CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2014

CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2014

Copyright © 2014 by Clifford A. Shaffer

Title page

Students should be familiar with inductive proofs, recursion, data structures, and programming at the CS3114 level.

CS5114: Theory of Algorithms

- Emphasis: Creation of Algorithms
- Less important:
  - Analysis of algorithms
  - Problem statement
  - Programming
- Central Paradigm: Mathematical Induction
  - Find a way to solve a problem by solving one or more smaller problems

CS5114: Theory of Algorithms

- Emphasis: Creation of Algorithms
- Less important:
  - Analysis of algorithms
  - Problem statement
  - Programming
- Central Paradigm: Mathematical Induction
  - Find a way to solve a problem by solving one or more smaller problems

Creation of algorithms comes through exploration, discovery, techniques, intuition: largely by **lots** of examples and **lots** of practice (HW exercises).
We will use Analysis of Algorithms as a tool.
Problem statement (in the software eng. sense) is not important because our problems are easily described, if not easily solved.
Smaller problems may or may not be the same as the original problem.
Divide and conquer is a way of solving a problem by solving one more more smaller problems.
Claim on induction: The processes of constructing proofs and constructing algorithms are similar.

Review of Mathematical Induction

- The paradigm of **Mathematical Induction** can be used to solve an enormous range of problems.
- **Purpose**: To prove a parameterized theorem of the form:
  Theorem: $\forall n \geq c, \mathbf{P}(n)$.
  - Use only positive integers $\geq c$ for n.
- Sample $\mathbf{P}(n)$:
  $n + 1 \leq n^2$

Review of Mathematical Induction

- The paradigm of Mathematical Induction can be used to solve an enormous range of problems.
- Purpose: To prove a parameterized theorem of the form:
  Theorem: $\forall n \geq c, \mathbf{P}(n)$.
  - Use only positive integers $\geq c$ for n.
- Sample $\mathbf{P}(n)$:
  $n + 1 \leq n^2$

First we will refresh/expand our our familiarity with induction.
Then we will try to apply an inductive approach to algorithm design.

$\mathbf{P}(n)$ is a statement containing n as a variable.

This sample $\mathbf{P}(n)$ is true for $n \geq 2$, but false for $n = 1$.

Principle of Mathematical Induction

- IF the following two statements are true:
  1. $\mathbf{P}(c)$ is true.
  2. For $n > c, \mathbf{P}(n-1)$ is true $\rightarrow \mathbf{P}(n)$ is true.

  ... **THEN** we may conclude: $\forall n \geq c$, $\mathbf{P}(n)$.
- The assumption "$\mathbf{P}(n-1)$ is true" is the **induction hypothesis**.
- Typical induction proof form:
  1. Base case
  2. State induction Hypothesis
  3. Prove the implication (induction step)
- What does this remind you of?

Principle of Mathematical Induction

- IF the following two statements are true:
  1. $\mathbf{P}(c)$ is true.
  2. For $n > c, \mathbf{P}(n-1)$ is true $\rightarrow \mathbf{P}(n)$ is true.

  ... THEN we may conclude: $\forall n \geq c, \mathbf{P}(n)$.
- The assumption "$\mathbf{P}(n-1)$ is true" is the induction hypothesis.
- Typical induction proof form:
  1. Base case
  2. State induction Hypothesis
  3. Prove the implication (induction step)
- What does this remind you of?

Important: The goal is to prove the **implication**, not the theorem! That is, prove that $\mathbf{P}(n-1) \rightarrow \mathbf{P}(n)$. **NOT** to prove $P(n)$. This is much easier, because we can assume that $\mathbf{P}(n-1)$ is true.
Consider the truth table for implication to see this. Since $A \rightarrow B$ is (vacuously) true when A is false, we can just assume that A is true since the implication is true anyway if A is false. That is, we only need to worry that the implication could be false if A is true.

The power of induction is that the induction hypothesis "comes for free." We often try to make the most of the extra information provided by the induction hypothesis.
This is like recursion! There you have a base case and a recursive call that must make progress toward the base case.

Induction Example 1

Theorem: Let

$$S(n) = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n.$$

Then, $\forall n \geq 1, S(n) = \frac{n(n+1)}{2}$.

Induction Example 1

Theorem: Let

$S(n) = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n$

Then, $\forall n \geq 1, S(n) = \frac{n(n+1)}{2}$.

Base Case: $\mathbf{P}(n)$ is true since $S(1) = 1 = 1(1+1)/2$.
Induction Hypothesis: $S(i) = \frac{i(i+1)}{2}$ for $i < n$.
Induction Step:

$$\begin{aligned} S(n) &= S(n-1) + n = (n-1)n/2 + n \\ &= \frac{n(n+1)}{2} \end{aligned}$$

Therefore, $\mathbf{P}(n-1) \rightarrow \mathbf{P}(n)$.
By the principle of Mathematical Induction,
$\forall n \geq 1, S(n) = \frac{n(n+1)}{2}$.
MI is often an ideal tool for **verification** of a hypothesis.
Unfortunately it does not help us to construct a hypothesis.

Induction Example 2

Theorem: $\forall n \geq 1, \forall$ real x such that $1 + x > 0$, $(1+x)^n \geq 1 + nx$.

Induction Example 2

Theorem: $\forall n \geq 1, \forall$ real x such that $1 + x > 0$, $(1+x)^n \geq 1 + nx$.

What do we do induction on? Can't be a real number, so must be n.
$\mathbf{P}(n) : (1+x)^n \geq 1 + nx$.

Base Case: $(1+x)^1 = 1 + x \geq 1 + 1x$
Induction Hypothesis: Assume $(1+x)^{n-1} \geq 1 + (n-1)x$
Induction Step:

$$\begin{aligned} (1+x)^n &= (1+x)(1+x)^{n-1} \\ &\geq (1+x)(1+(n-1)x) \\ &= 1 + nx - x + x + nx^2 - x^2 \\ &= 1 + nx + (n-1)x^2 \\ &\geq 1 + nx. \end{aligned}$$

Induction Example 3

Theorem: 2¢ and 5¢ stamps can be used to form any denomination (for denominations ≥ 4).

Induction Example 3

Theorem: 2¢ and 5¢ stamps can be used to form any denomination (for denominations ≥ 4).

Base case: $4 = 2 + 2$.

Induction Hypothesis: Assume $\mathbf{P}(k)$ for $4 \leq k < n$.

Induction Step:
Case 1: $n-1$ is made up of all 2¢ stamps. Then, replace 2 of these with a 5¢ stamp.

Case 2: $n-1$ includes a 5¢ stamp. Then, replace this with 3 2¢ stamps.

Colorings

4-color problem: For any set of polygons, 4 colors are sufficient to guarentee that no two adjacent polygons share the same color.

Restrict the problem to regions formed by placing (infinite) lines in the plane. How many colors do we need?
Candidates:

- 4: Certainly
- 3: ?
- 2: ?
- 1: No!

Let's try it for 2...

Colorings

4-color problem: For any set of polygons, 4 colors are sufficient to guarentee that no two adjacent polygons share the same color.

Restrict the problem to regions formed by placing (infinite) lines in the plane. How many colors do we need?
Candidates:

- 4: Certainly
- 3: ?
- 2: ?
- 1: No!

Let's try it for 2...

Induction is useful for much more than checking equations!

If we accept the statement about the general 4-color problem, then of course 4 colors is enough for our restricted version.

If 2 is enough, then of course we can do it with 3 or more.

Two-coloring Problem

Given: Regions formed by a collection of (infinite) lines in the plane.
Rule: Two regions that share an edge cannot be the same color.

Theorem: It is possible to two-color the regions formed by n lines.

Two-coloring Problem
Given: Regions formed by a collection of (infinite) lines in the plane.
Rule: Two regions that share an edge cannot be the same color.
Theorem: It is possible to two-color the regions formed by n lines.

Picking what to do induction on can be a problem. Lines? Regions? How can we "add a region?" We can't, so try induction on lines.
Base Case: $n = 1$. Any line divides the plane into two regions.
Induction Hypothesis: It is possible to two-color the regions formed by $n - 1$ lines.
Induction Step: Start with the regions formed from $n - 1$ lines and 2-color them. Now, introduce the n'th line.
This line cuts some colored regions in two.
Reverse the region colors on one side of the n'th line.
A valid two-coloring results.

- Any boundary surviving the addition still has opposite colors.
- Any new boundary also has opposite colors after the switch.

Strong Induction

IF the following two statements are true:

1. $\mathbf{P}(c)$
2. $\mathbf{P}(i), i = 1, 2, \cdots, n-1 \rightarrow \mathbf{P}(n)$,

... **THEN** we may conclude: $\forall n \geq c$, $\mathbf{P}(n)$.

Advantage: We can use statements other than $\mathbf{P}(n-1)$ in proving $\mathbf{P}(n)$.

Strong Induction
IF the following two statements are true:
1. $\mathbf{P}(c)$
2. $\mathbf{P}(i), i = 1, 2, \cdots, n-1 \rightarrow \mathbf{P}(n)$,

... **THEN** we may conclude: $\forall n \geq c$, $\mathbf{P}(n)$.
Advantage: We can use statements other than $\mathbf{P}(n-1)$ in proving $\mathbf{P}(n)$.

The previous examples were all very straightforward – simply add in the n'th item and justify that the IH is maintained.
Now we will see examples where we must do more sophisticated (creative!) maneuvers such as

- go backwards from n.
- prove a stronger IH.

to make the most of the IH.

Graph Problem

An **Independent Set** of vertices is one for which no two vertices are adjacent.

Theorem: Let $G = (V, E)$ be a **directed** graph. Then, G contains some independent set $S(G)$ such that every vertex can be reached from a vertex in $S(G)$ by a path of length at most 2.

Example: a graph with 3 vertices in a cycle. Pick any one vertex as $S(G)$.

Graph Problem
An **Independent Set** of vertices is one for which no two vertices are adjacent.
Theorem: Let $G = (V, E)$ be a **directed** graph. Then, G contains some independent set $S(G)$ such that every vertex can be reached from a vertex in $S(G)$ by a path of length at most 2.
Example: a graph with 3 vertices in a cycle. Pick any one vertex as $S(G)$.

It should be obvious that the theorem is true for an undirected graph. Pick any independent set. Then add any node not adjacent, one by one.
Naive approach: Assume the theorem is true for any graph of $n - 1$ vertices. Now add the nth vertex and its edges. But this won't work for the graph $1 \leftarrow 2$. Initially, vertex 1 is the independent set. We can't add 2 to the graph. Nor can we reach it from 1.
Going forward is good for proving existance.
Going backward (from an arbitrary instance into the IH) is usually necessary to prove that a property holds in all instances. This is because going forward requires proving that you reach all of the possible instances.

Graph Problem (cont)

Theorem: Let $G = (V, E)$ be a **directed** graph. Then, G contains some independent set $S(G)$ such that every vertex can be reached from a vertex in $S(G)$ by a path of length at most 2.
Base Case: Easy if $n \leq 3$ because there can be no path of length > 2.
Induction Hypothesis: The theorem is true if $|V| < n$.
Induction Step ($n > 3$):
Pick any $v \in V$.
Define: $N(v) = \{v\} \cup \{w \in V | (v, w) \in E\}$.
$H = G - N(v)$.
Since the number of vertices in H is less than n, there is an independent set $S(H)$ that satisfies the theorem for H.

Graph Problem (cont)
Theorem: Let $G = (V, E)$ be a **directed** graph. Then, G contains some independent set $S(G)$ such that every vertex can be reached from a vertex in $S(G)$ by a path of length at most 2.
Base Case: Easy if $n \leq 3$ because there can be no path of length > 2.
Induction Hypothesis: The theorem is true if $|V| < n$.
Induction Step ($n > 3$):
Pick any $v \in V$.
Define: $N(v) = \{v\} \cup \{w \in V | (v, w) \in E\}$.
$H = G - N(v)$.
Since the number of vertices in H is less than n, there is an independent set $S(H)$ that satisfies the theorem for H.

$N(v)$ is all vertices reachable (directly) from v. That is, the Neighbors of v.
H is the graph induced by $V - N(v)$.

OK, so why remove both v and $N(v)$ from the graph? If we only remove v, we have the same problem as before. If G is $1 \rightarrow 2 \rightarrow 3$, and we remove 1, then the independent set for H must be vertex 2. We can't just add back 1. But if we remove both 1 and 2, then we'll be able to do something...

Graph Proof (cont)

There are two cases:

1. $S(H) \cup \{v\}$ is independent.
   Then $S(G) = S(H) \cup \{v\}$.
2. $S(H) \cup \{v\}$ is not independent.
   Let $w \in S(H)$ such that $(w, v) \in E$.
   Every vertex in $N(v)$ can be reached by w with path of length ≤ 2.
   So, set $S(G) = S(H)$.

By Strong Induction, the theorem holds for all G.

Graph Proof (cont)

There are two cases:
1. $S(H) \cup \{v\}$ is independent.
   Then $S(G) = S(H) \cup \{v\}$.
2. $S(H) \cup \{v\}$ is not independent.
   Let $w \in S(H)$ such that $(w, v) \in E$.
   Every vertex in $N(v)$ can be reached by w with path of length ≤ 2.
   So, set $S(G) = S(H)$.

By Strong Induction, the theorem holds for all G.

"$S(H) \cup \{v\}$ is not independent" means that there is an edge from something in $S(H)$ to v.
IMPORTANT: There cannot be an edge from v to $S(H)$ because whatever we can reach from v is in $N(v)$ and would have been removed in H.
We need strong induction for this proof because we don't know how many vertices are in $N(v)$.
We must remove $N(v)$ instead of just v because of this case: We remove just v to yield H. S(H) turns out to have something that can be reached from v. So, when we add v back to reform G, v cannot become part of S(G) (because that would violate the definition of independent set). But if v is 3 steps away from anything in S(H), we must add it to satisfy the theorem. So are stuck.

Fibonacci Numbers

Define Fibonacci numbers inductively as:

$$\begin{aligned} F(1) &= F(2) = 1 \\ F(n) &= F(n-1) + F(n-2), n > 2. \end{aligned}$$

Theorem: $\forall n \geq 1, F(n)^2 + F(n+1)^2 = F(2n+1)$.

Induction Hypothesis:
$F(n-1)^2 + F(n)^2 = F(2n-1)$.

Fibonacci Numbers (2)

Expand both sides of the theorem, then cancel like terms:
$F(2n+1) = F(2n) + F(2n-1)$ and,

$$\begin{aligned} F(n)^2 + F(n+1)^2 &= F(n)^2 + (F(n) + F(n-1))^2 \\ &= F(n)^2 + F(n)^2 + 2F(n)F(n-1) + F(n-1)^2 \\ &= F(n)^2 + F(n-1)^2 + F(n)^2 + 2F(n)F(n-1) \\ &= F(2n-1) + F(n)^2 + 2F(n)F(n-1). \end{aligned}$$

Want: $F(n)^2 + F(n+1)^2 = F(2n+1) = F(2n) + F(2n-1)$
Steps above left us with needing to prove:
$F(2n) + F(2n-1) = F(2n-1) + F(n)^2 + 2F(n)F(n-1)$
So we need to show that: $F(2n) = F(n)^2 + 2F(n)F(n-1)$
To prove the original theorem, we must prove this. Since we must do it anyway, we should take advantage of this in our IH!

Fibonacci Numbers (3)

With a stronger theorem comes a stronger IH!

Theorem:
$F(n)^2 + F(n+1)^2 = F(2n+1)$ and
$F(n)^2 + 2F(n)F(n-1) = F(2n)$.

Induction Hypothesis:
$F(n-1)^2 + F(n)^2 = F(2n-1)$ and
$F(n-1)^2 + 2F(n-1)F(n-2) = F(2n-2)$.

Fibonacci Numbers (4)

$$\begin{aligned} & F(n)^2 + 2F(n)F(n-1) \\ &= F(n)^2 + 2(F(n-1) + F(n-2))F(n-1) \\ &= F(n)^2 + F(n-1)^2 + 2F(n-1)F(n-2) + F(n-1)^2 \\ &= F(2n-1) + F(2n-2) \\ &= F(2n). \end{aligned}$$

$$\begin{aligned} F(n)^2 + F(n+1)^2 &= F(n)^2 + [F(n) + F(n-1)]^2 \\ &= F(n)^2 + F(n)^2 + 2F(n)F(n-1) + F(n-1)^2 \\ &= F(n)^2 + F(2n) + F(n-1)^2 \\ &= F(2n-1) + F(2n) \\ &= F(2n+1). \end{aligned}$$

... which proves the theorem. The original result could not have been proved without the stronger induction hypothesis.

Another Example

Theorem: All horses are the same color.

Proof: $\mathbf{P}(n)$: If S is a set of n horses, then all horses in S have the same color.
Base case: $n = 1$ is easy.
Induction Hypothesis: Assume $\mathbf{P}(i), i < n$.
Induction Step:

- Let S be a set of horses, $|S| = n$.
- Let S' be $S - \{h\}$ for some horse h.
- By IH, all horses in S' have the same color.
- Let h' be some horse in S'.
- IH implies $\{h, h'\}$ have all the same color.

Therefore, $\mathbf{P}(n)$ holds.

Another Example

Theorem: All horses are the same color.
Proof: $\mathbf{P}(n)$: If S is a set of n horses, then all horses in S have the same color.
Base case: $n = 1$ is easy.
Induction Hypothesis: Assume $\mathbf{P}(i), i < n$.
Induction Step:
- Let S be a set of horses, $|S| = n$.
- Let S' be $S - \{h\}$ for some horse h.
- By IH, all horses in S' have the same color.
- Let h' be some horse in S'.
- IH implies $\{h, h'\}$ have all the same color.

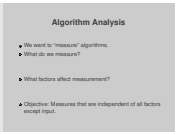
Therefore, $\mathbf{P}(n)$ holds.

The problem is that the base case does not give enough strength to give the **particular** instance of $n = 2$ used in the last step.

If it were true for 2, then the whole proof woudl work. But we cannot get from the base case to an arbitrary 2.

Algorithm Analysis

- We want to "measure" algorithms.
- What do we measure?
- What factors affect measurement?
- Objective: Measures that are independent of all factors except input.



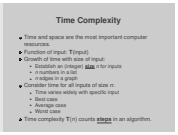
What do we measure?
Time and space to run; ease of implementation (this changes with language and tools); code size

What affects measurement?
Computer speed and architecture; Programming language and compiler; System load; Programmer skill; Specifics of input (size, arrangement)

If you compare two programs running on the same computer under the same conditions, all the other factors (should) cancel out.
Want to measure the relative efficiency of two algorithms without needing to implement them on a real computer.

Time Complexity

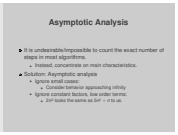
- Time and space are the most important computer resources.
- Function of input: **T**(input)
- Growth of time with size of input:
  - Establish an (integer) **size** n for inputs
  - n numbers in a list
  - n edges in a graph
- Consider time for all inputs of size n:
  - Time varies widely with specific input
  - Best case
  - Average case
  - Worst case
- Time complexity **T**(n) counts **steps** in an algorithm.



Sometimes analyze in terms of more than one variable.
Best case usually not of interest.
Average case is usually what we want, but can be hard to measure.
Worst case appropriate for "real-time" applications, often best we can do in terms of measurement.
Examples of "steps:" comparisons, assignments, arithmetic/logical operations. What we choose for "step" depends on the algorithm. Step cost must be "constant" – not dependent on n.

Asymptotic Analysis

- It is undesirable/impossible to count the exact number of steps in most algorithms.
  - Instead, concentrate on main characteristics.
- Solution: Asymptotic analysis
  - Ignore small cases:
    - Consider behavior approaching infinity
  - Ignore constant factors, low order terms:
    - $2n^2$ looks the same as $5n^2 + n$ to us.



Undesirable to count number of machine instructions or steps because issues like processor speed muddy the waters.

O Notation

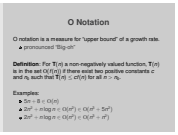
O notation is a measure for "upper bound" of a growth rate.

- pronounced "Big-oh"

Definition: For **T**(n) a non-negatively valued function, **T**(n) is in the set $\mathrm{O}(f(n))$ if there exist two positive constants c and n_0 such that **T**(n) $\leq cf(n)$ for all $n > n_0$.

Examples:

- $5n + 8 \in \mathrm{O}(n)$
- $2n^2 + n\log n \in \mathrm{O}(n^2) \in \mathrm{O}(n^3 + 5n^2)$
- $2n^2 + n\log n \in \mathrm{O}(n^2) \in \mathrm{O}(n^3 + n^2)$



Remember: The time equation is for some particular set of inputs – best, worst, or average case.

O Notation (cont)

We seek the "simplest" and "strongest" f.

Big-O is somewhat like "$\leq$":
$n^2 \in \mathrm{O}(n^3)$ and $n^2 \log n \in O(n^3)$, but

- $n^2 \neq n^2 \log n$
- $n^2 \in \mathrm{O}(n^2)$ while $n^2 \log n \notin \mathrm{O}(n^2)$

O Notation (cont)

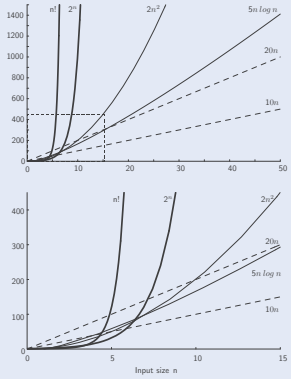
We seek the "simplest" and "strongest" f.

Big-O is somewhat like "≤":
$n^2 \in O(n^3)$ and $n^2 \log n \in O(n^3)$, but
- $n^2 \neq n^2 \log n$
- $n^2 \in O(n^2)$ while $n^2 \log n \notin O(n^2)$

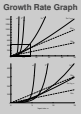
A common misunderstanding:

- "The best case for my algorithm is $n = 1$ because that is the fastest." WRONG!
- Big-oh refers to a growth rate as n grows to ∞.
- Best case is defined for the input of size n that is cheapest among all inputs of size n.

Growth Rate Graph



Growth Rate Graph



2^n is an exponential algorithm. $10n$ and $20n$ differ only by a constant.

Speedups

What happens when we buy a computer 10 times faster?

$\mathbf{T}(n)$	n	n'	Change	n'/n
$10n$	1,000	10,000	$n' = 10n$	10
$20n$	500	5,000	$n' = 10n$	10
$5n \log n$	250	1,842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	$--$

n: Size of input that can be processed in one hour (10,000 steps).

n': Size of input that can be processed in one hour on the new machine (100,000 steps).

Speedups

What happens when we buy a computer 10 times faster?

T(n)	n	n'	Change	n'/n
10n	1,000	10,000	n' = 10n	10
20n	500	5,000	n' = 10n	10
5nlog n	250	1,842	√10n < n' < 10n	7.37
2n²	70	223	n' = √10n	3.16
2^n	13	16	n' = n + 3	--

n: Size of input that can be processed in one hour (10,000 steps).

n': Size of input that can be processed in one hour on the new machine (100,000 steps).

How much speedup? 10 times. More important: How much increase in problem size for same time? Depends on growth rate.
For n^2, if $n = 1000$, then n' would be 1003.
Compare $\mathbf{T}(n) = n^2$ to $\mathbf{T}(n) = n \log n$. For $n > 58$, it is faster to have the $\Theta(n \log n)$ algorithm than to have a computer that is 10 times faster.

Some Rules for Use

Definition: f is **monotonically growing** if $n_1 \geq n_2$ implies $f(n_1) \geq f(n_2)$.
We typically assume our time complexity function is monotonically growing.

Theorem 3.1: Suppose f is monotonically growing.
$\forall c > 0$ and $\forall a > 1, (f(n))^c \in O(a^{f(n)})$
In other words, an **exponential** function grows faster than a **polynomial** function.

Lemma 3.2: If $f(n) \in O(s(n))$ and $g(n) \in O(r(n))$ then

- $f(n) + g(n) \in O(s(n) + r(n)) \equiv O(\max(s(n), r(n)))$
- $f(n)g(n) \in O(s(n)r(n))$.
- If $s(n) \in O(h(n))$ then $f(n) \in O(h(n))$
- For any constant k, $f(n) \in O(ks(n))$

Some Rules for Use

Definition: f is monotonically growing if n₁ ≥ n₂ implies f(n₁) ≥ f(n₂).
We typically assume our time complexity function is monotonically growing.

Theorem 3.1: Suppose f is monotonically growing.
∀c > 0 and ∀a > 1, (f(n))^c ∈ O(a^f(n))
In other words, an exponential function grows faster than a polynomial function.

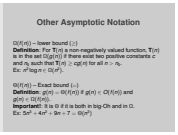
Lemma 3.2: If f(n) ∈ O(s(n)) and g(n) ∈ O(r(n)) then
- f(n) + g(n) ∈ O(s(n) + r(n)) ≡ O(max(s(n), r(n)))
- f(n)g(n) ∈ O(s(n)r(n)).
- If s(n) ∈ O(h(n)) then f(n) ∈ O(h(n))
- For any constant k, f(n) ∈ O(ks(n))

Assume monitonic growth because larger problems should take longer to solve. However, many real problems have "cyclically growing" behavior.
Is $O(2^{f(n)}) \in O(3^{f(n)})$? Yes, but not vice versa.
$3^n = 1.5^n \times 2^n$ so no constant could ever make 2^n bigger than 3^n for all n.
functional composition

Other Asymptotic Notation

$\Omega(f(n))$ – lower bound ($\geq$)
Definition: For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ for all $n > n_0$.
Ex: $n^2 \log n \in \Omega(n^2)$.

$\Theta(f(n))$ – Exact bound ($=$)
Definition: $g(n) = \Theta(f(n))$ if $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$.
Important!: It is Θ if it is both in big-Oh and in Ω.
Ex: $5n^3 + 4n^2 + 9n + 7 = \Theta(n^3)$



Ω is most userful to discuss cost of problems, not algorithms. Once you have an equation, the bounds have met. So this is more interesting when discussing your level of uncertainty about the difference between the upper and lower bound.

You have Θ when you have the upper and the lower bounds meeting. So Θ means that you know a lot more than just Big-oh, and so is perferred when possible.

A common misunderstanding:

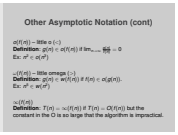
- Confusing worst case with upper bound.
- Upper bound refers to a growth rate.
- Worst case refers to the worst input from among the choices for possible inputs of a given size.

Other Asymptotic Notation (cont)

$o(f(n))$ – little o ($<$)
Definition: $g(n) \in o(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$
Ex: $n^2 \in o(n^3)$

$\omega(f(n))$ – little omega ($>$)
Definition: $g(n) \in w(f(n))$ if $f(n) \in o(g(n))$.
Ex: $n^5 \in w(n^2)$

$\infty(f(n))$
Definition: $T(n) = \infty(f(n))$ if $T(n) = O(f(n))$ but the constant in the O is so large that the algorithm is impractical.



We won't use these too much.

Aim of Algorithm Analysis

Typically want to find "simple" $f(n)$ such that $T(n) = \Theta(f(n))$.

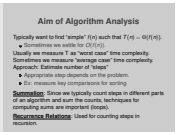
- Sometimes we settle for $O(f(n))$.

Usually we measure T as "worst case" time complexity.
Sometimes we measure "average case" time complexity.
Approach: Estimate number of "steps"

- Appropriate step depends on the problem.
- Ex: measure key comparisons for sorting

Summation: Since we typically count steps in different parts of an algorithm and sum the counts, techniques for computing sums are important (loops).

Recurrence Relations: Used for counting steps in recursion.



We prefer Θ over Big-oh because Θ means that we understand our bounds and they met. But if we just can't find that the bottom meets the top, then we are stuck with just Big-oh. Lower bounds can be hard. For **problems** we are often interested in Ω

– but this is often hard for non-trivial situations!

Often prefer average case (except for real-time programming), but worst case is simpler to compute than average case since we need not be concerned with distribution of input.

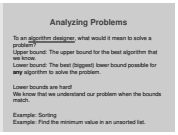
For the sorting example, key comparisons must be constant-time to be used as a cost measure.

Analyzing Problems

To an algorithm designer, what would it mean to solve a problem?
Upper bound: The upper bound for the best algorithm that we know.
Lower bound: The best (biggest) lower bound possible for **any** algorithm to solve the problem.

Lower bounds are hard!
We know that we understand our problem when the bounds match.

Example: Sorting
Example: Find the minimum value in an unsorted list.



Sorting: If you only know simple sorts, your upper bound is $O(n^2)$.
Then you learn better sorts and your upper bound is $O(n \log n)$
A naive lower bound is $\Omega(n)$. Later we learn the proof that no (general) sorting algorithm can have a worst case better than $\Omega(n \log n)$.
At that point, we know that sorting is $\Theta(n)$. Minimum Finding:

The upper bound is $O(n)$ because we know an algorithm to solve it in that time.
The lower bound is $\Omega(n)$ because we have to look at every value to be sure we have the answer

Summation: Guess and Test

Technique 1: Guess the solution and use induction to test.

Technique 1a: Guess the form of the solution, and use simultaneous equations to generate constants. Finally, use induction to test.

Summation: Guess and Test

Technique 1: Guess the solution and use induction to test.

Technique 1a: Guess the form of the solution, and use simultaneous equations to generate constants. Finally, use induction to test.

no notes

Summation Example

$$S(n) = \sum_{i=0}^{n} i^2.$$

Guess that $S(n)$ is a polynomial $\leq n^3$.
Equivalently, guess that it has the form
$S(n) = an^3 + bn^2 + cn + d$.

For $n = 0$ we have $S(n) = 0$ so $d = 0$.
For $n = 1$ we have $a + b + c + 0 = 1$.
For $n = 2$ we have $8a + 4b + 2c = 5$.
For $n = 3$ we have $27a + 9b + 3c = 14$.
Solving these equations yields $a = \frac{1}{3}, b = \frac{1}{2}, c = \frac{1}{6}$

Now, prove the solution with induction.

Summation Example

$S(n) = \sum_{i=0}^{n} i^2$

Guess that $S(n)$ is a polynomial $\leq n^3$.
Equivalently, guess that it has the form
$S(n) = an^3 + bn^2 + cn + d$.

For $n = 0$ we have $S(n) = 0$ so $d = 0$.
For $n = 1$ we have $a + b + c + 0 = 1$.
For $n = 2$ we have $8a + 4b + 2c = 5$.
For $n = 3$ we have $27a + 9b + 3c = 14$.
Solving these equations yields $a = \frac{1}{3}, b = \frac{1}{2}, c = \frac{1}{6}$

Now, prove the solution with induction.

This is Manber Problem 2.5.

We need to prove by induction since we don't know that the guessed form is correct. All that we **know** without doing the proof is that the form we guessed models some low-order points on the equation properly.

Technique 2: Shifted Sums

Given a sum of many terms, shift and subtract to eliminate intermediate terms.

$$G(n) = \sum_{i=0}^{n} ar^i = a + ar + ar^2 + \cdots + ar^n$$

Shift by multiplying by r.

$$rG(n) = ar + ar^2 + \cdots + ar^n + ar^{n+1}$$

Subtract.

$$G(n) - rG(n) = G(n)(1 - r) = a - ar^{n+1}$$

$$G(n) = \frac{a - ar^{n+1}}{1 - r} \quad r \neq 1$$

Technique 2: Shifted Sums

Given a sum of many terms, shift and subtract to eliminate intermediate terms.

$G(n) = \sum_{i=0}^{n} ar^i = a + ar + ar^2 + \cdots + ar^n$

Shift by multiplying by r.

$rG(n) = ar + ar^2 + \cdots + ar^n + ar^{n+1}$

Subtract.

$G(n) - rG(n) = G(n)(1 - r) = a - ar^{n+1}$

$G(n) = \frac{a - ar^{n+1}}{1 - r} \quad r \neq 1$

We often solve summations in this way – by multiplying by something or subtracting something. The big problem is that it can be a bit like finding a needle in a haystack to decide what "move" to make. We need to do something that gives us a new sum that allows us either to cancel all but a constant number of terms, or else converts all the terms into something that forms an easier summation.

Shift by multiplying by r is a reasonable guess in this example since the terms differ by a factor of r.

Example 3.3

$$G(n) = \sum_{i=1}^{n} i2^i = 1 \times 2 + 2 \times 2^2 + 3 \times 2^3 + \cdots + n \times 2^n$$

Multiply by 2.

$$2G(n) = 1 \times 2^2 + 2 \times 2^3 + 3 \times 2^4 + \cdots + n \times 2^{n+1}$$

Subtract (Note: $\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$)

$$\begin{aligned} 2G(n) - G(n) &= n2^{n+1} - 2^n \cdots 2^2 - 2 \\ G(n) &= n2^{n+1} - 2^{n+1} + 2 \\ &= (n-1)2^{n+1} + 2 \end{aligned}$$

Example 3.3

$G(n) = \sum_{i=1}^{n} i2^i = 1 \times 2 + 2 \times 2^2 + 3 \times 2^3 + \cdots + n \times 2^n$

Multiply by 2.

$2G(n) = 1 \times 2^2 + 2 \times 2^3 + 3 \times 2^4 + \cdots + n \times 2^{n+1}$

Subtract (Note: $\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$)

$\begin{aligned} 2G(n) - G(n) &= n2^{n+1} - 2^n \cdots 2^2 - 2 \\ G(n) &= n2^{n+1} - 2^{n+1} + 2 \\ &= (n-1)2^{n+1} + 2 \end{aligned}$

no notes

Recurrence Relations

- A (math) function defined in terms of itself.
- Example: Fibonacci numbers:
  $F(n) = F(n-1) + F(n-2)$ general case
  $F(1) = F(2) = 1$ base cases
- There are always one or more general cases and one or more base cases.
- We will use recurrences for time complexity of recursive (computer) functions.
- General format is $T(n) = E(T, n)$ where $E(T, n)$ is an expression in T and n.
  - $T(n) = 2T(n/2) + n$
- Alternately, an upper bound: $T(n) \leq E(T, n)$.

Recurrence Relations
- A (math) function defined in terms of itself.
- Example: Fibonacci numbers:
  $F(n) = F(n-1) + F(n-2)$ general case
  $F(1) = F(2) = 1$ base cases
- There are always one or more general cases and one or more base cases.
- We will use recurrences for time complexity of recursive (computer) functions.
- General format is $T(n) = E(T, n)$ where $E(T, n)$ is an expression in T and n.
  - $T(n) = 2T(n/2) + n$
- Alternately, an upper bound: $T(n) \leq E(T, n)$.

We won't spend a lot of time on techniques... just enough to be able to use them.

Solving Recurrences

We would like to find a closed form solution for $T(n)$ such that:

$$T(n) = \Theta(f(n))$$

Alternatively, find lower bound

- Not possible for inequalities of form $T(n) \leq E(T, n)$.

Methods:

- Guess (and test) a solution
- Expand recurrence
- Theorems

Solving Recurrences
We would like to find a closed form solution for $T(n)$ such that:
$T(n) = \Theta(f(n))$
Alternatively, find lower bound
- Not possible for inequalities of form $T(n) \leq E(T, n)$.
Methods:
- Guess (and test) a solution
- Expand recurrence
- Theorems

Note that "finding a closed form" means that we have $f(n)$ that doesn't include T.

Can't find lower bound for the inequality because you do not know enough... you don't know *how much bigger* $E(T, n)$ is than $T(n)$, so the result might not be $\Omega(T(n))$.

Guessing is useful for finding an asymptotic solution. Use induction to prove the guess correct.

Guessing

$T(n) = 2T(n/2) + 5n^2 \quad n \geq 2$
$T(1) = 7$
Note that T is defined only for powers of 2.

Guess a solution: $T(n) \leq c_1 n^3 = f(n)$
$T(1) = 7$ implies that $c_1 \geq 7$

Inductively, assume $T(n/2) \leq f(n/2)$.

$$\begin{aligned} T(n) &= 2T(n/2) + 5n^2 \\ &\leq 2c_1(n/2)^3 + 5n^2 \\ &\leq c_1(n^3/4) + 5n^2 \\ &\leq c_1 n^3 \text{ if } c_1 \geq 20/3. \end{aligned}$$

Guessing
$T(n) = 2T(n/2) + 5n^2 \quad n \geq 2$
$T(1) = 7$
Note that T is defined only for powers of 2.
Guess a solution: $T(n) \leq c_1 n^3 = f(n)$
$T(1) = 7$ implies that $c_1 \geq 7$
Inductively, assume $T(n/2) \leq f(n/2)$.
$$\begin{aligned} T(n) &= 2T(n/2) + 5n^2 \\ &\leq 2c_1(n/2)^3 + 5n^2 \\ &\leq c_1(n^3/4) + 5n^2 \\ &\leq c_1 n^3 \text{ if } c_1 \geq 20/3. \end{aligned}$$

For Big-oh, not many choices in what to guess.

$7 \times 1^3 = 7$

Because $\frac{20}{4 \cdot 3} n^3 + 5n^2 = \frac{20}{3} n^3$ when $n = 1$, and as n grows, the right side grows even faster.

Guessing (cont)

Therefore, if $c_1 = 7$, a proof by induction yields:
$T(n) \leq 7n^3$
$T(n) \in O(n^3)$

Is this the best possible solution?

Guessing (cont)
Therefore, if $c_1 = 7$, a proof by induction yields:
$T(n) \leq 7n^3$
$T(n) \in O(n^3)$
Is this the best possible solution?

No - try something tighter.

Guessing (cont)

Guess again.

$$T(n) \leq c_2 n^2 = g(n)$$

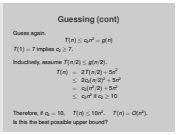
$T(1) = 7$ implies $c_2 \geq 7$.

Inductively, assume $T(n/2) \leq g(n/2)$.

$$\begin{aligned} T(n) &= 2T(n/2) + 5n^2 \\ &\leq 2c_2(n/2)^2 + 5n^2 \\ &= c_2(n^2/2) + 5n^2 \\ &\leq c_2 n^2 \text{ if } c_2 \geq 10 \end{aligned}$$

Therefore, if $c_2 = 10$, $T(n) \leq 10n^2$. $T(n) = O(n^2)$.

Is this the best possible upper bound?



Because $\frac{10}{2}n^2 + 5n^2 = 10n^2$ for $n = 1$, and the right hand side grows faster.

Yes this is best, since $T(n)$ can be as bad as $5n^2$.

Guessing (cont)

Now, reshape the recurrence so that T is defined for all values of n.

$T(n) \leq 2T(\lfloor n/2 \rfloor) + 5n^2 \qquad n \geq 2$

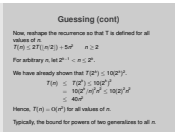
For arbitrary n, let $2^{k-1} < n \leq 2^k$.

We have already shown that $T(2^k) \leq 10(2^k)^2$.

$$\begin{aligned} T(n) &\leq T(2^k) \leq 10(2^k)^2 \\ &= 10(2^k/n)^2 n^2 \leq 10(2)^2 n^2 \\ &\leq 40n^2 \end{aligned}$$

Hence, $T(n) = \mathrm{O}(n^2)$ for all values of n.

Typically, the bound for powers of two generalizes to all n.



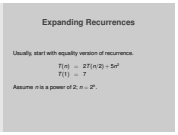
no notes

Expanding Recurrences

Usually, start with equality version of recurrence.

$$\begin{aligned} T(n) &= 2T(n/2) + 5n^2 \\ T(1) &= 7 \end{aligned}$$

Assume n is a power of 2; $n = 2^k$.

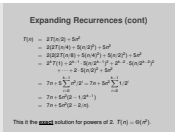


no notes

Expanding Recurrences (cont)

$$\begin{aligned} T(n) &= 2T(n/2) + 5n^2 \\ &= 2(2T(n/4) + 5(n/2)^2) + 5n^2 \\ &= 2(2(2T(n/8) + 5(n/4)^2) + 5(n/2)^2) + 5n^2 \\ &= 2^k T(1) + 2^{k-1} \cdot 5(n/2^{k-1})^2 + 2^{k-2} \cdot 5(n/2^{k-2})^2 \\ &\quad + \cdots + 2 \cdot 5(n/2)^2 + 5n^2 \\ &= 7n + 5\sum_{i=0}^{k-1} n^2/2^i = 7n + 5n^2 \sum_{i=0}^{k-1} 1/2^i \\ &= 7n + 5n^2(2 - 1/2^{k-1}) \\ &= 7n + 5n^2(2 - 2/n). \end{aligned}$$

This it the **exact** solution for powers of 2. $T(n) = \Theta(n^2)$.



no notes

Divide and Conquer Recurrences

These have the form:

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c \end{aligned}$$

... where a, b, c, k are constants.

A problem of size n is divided into a subproblems of size n/b, while cn^k is the amount of work needed to combine the solutions.

Divide and Conquer Recurrences

These have the form:

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c \end{aligned}$$

... where a, b, c, k are constants.

A problem of size n is divided into a subproblems of size n/b, while cn^k is the amount of work needed to combine the solutions.

no notes

Divide and Conquer Recurrences (cont)

Expand the sum; $n = b^m$.

$$\begin{aligned} T(n) &= a(aT(n/b^2) + c(n/b)^k) + cn^k \\ &= a^m T(1) + a^{m-1} c(n/b^{m-1})^k + \cdots + ac(n/b)^k + cn^k \\ &= ca^m \sum_{i=0}^{m} (b^k/a)^i \end{aligned}$$

$a^m = a^{\log_b n} = n^{\log_b a}$
The summation is a geometric series whose sum depends on the ratio

$$r = b^k/a.$$

There are 3 cases.

Divide and Conquer Recurrences (cont)

Expand the sum; $n = b^m$.

$$\begin{aligned} T(n) &= a(aT(n/b^2) + c(n/b)^k) + cn^k \\ &= a^m T(1) + a^{m-1} c(n/b^{m-1})^k + \cdots + ac(n/b)^k + cn^k \\ &= ca^m \sum_{i=0}^{m} (b^k/a)^i \end{aligned}$$

$a^m = a^{\log_b n} = n^{\log_b a}$
The summation is a geometric series whose sum depends on the ratio

$$r = b^k/a.$$

There are 3 cases.

$n = b^m \Rightarrow m = log_b n$.

Set $a = b^{\log_b a}$. Switch order of logs, giving $(b^{\log_b n})^{\log_b a} = n^{\log_b a}$.

D & C Recurrences (cont)

(1) $r < 1$.

$$\sum_{i=0}^{m} r^i < 1/(1-r), \qquad \text{a constant.}$$

$$T(n) = \Theta(a^m) = \Theta(n^{\log_b a}).$$

(2) $r = 1$.

$$\sum_{i=0}^{m} r^i = m + 1 = \log_b n + 1$$

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^k \log n)$$

D & C Recurrences (cont)

(1) $r < 1$.

$$\sum_{i=0}^{m} r^i < 1/(1-r), \qquad \text{a constant.}$$

$$T(n) = \Theta(a^m) = \Theta(n^{\log_b a})$$

(2) $r = 1$.

$$\sum_{i=0}^{m} r^i = m + 1 = \log_b n + 1$$

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^k \log n)$$

When $r = 1$, since $r = b^k/a = 1$, we get $a = b^k$.
Recall that $k = log_b a$.

D & C Recurrences (Case 3)

(3) $r > 1$.

$$\sum_{i=0}^{m} r^i = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m)$$

So, from $T(n) = ca^m \sum r^i$,

$$\begin{aligned} T(n) &= \Theta(a^m r^m) \\ &= \Theta(a^m (b^k/a)^m) \\ &= \Theta(b^{km}) \\ &= \Theta(n^k) \end{aligned}$$

D & C Recurrences (Case 3)

(3) $r > 1$.

$$\sum_{i=0}^{m} r^i = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m)$$

So, from $T(n) = ca^m \sum r^i$,

$$\begin{aligned} T(n) &= \Theta(a^m r^m) \\ &= \Theta(a^m (b^k/a)^m) \\ &= \Theta(b^{km}) \\ &= \Theta(n^k) \end{aligned}$$

no notes

Summary

Theorem 3.4:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

Apply the theorem:
$T(n) = 3T(n/5) + 8n^2$.
$a = 3, b = 5, c = 8, k = 2$.
$b^k/a = 25/3$.

Case (3) holds: $T(n) = \Theta(n^2)$.

Summary

Theorem 3.4:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

Apply the theorem:
$T(n) = 3T(n/5) + 8n^2$.
$a = 3, b = 5, c = 8, k = 2$.
$b^k/a = 25/3$.

Case (3) holds: $T(n) = \Theta(n^2)$.

We simplify by approximating summations.

Examples

- Mergesort: $T(n) = 2T(n/2) + n$.
  $2^1/2 = 1$, so $T(n) = \Theta(n \log n)$.
- Binary search: $T(n) = T(n/2) + 2$.
  $2^0/1 = 1$, so $T(n) = \Theta(\log n)$.
- Insertion sort: $T(n) = T(n-1) + n$.
  Can't apply the theorem. Sorry!
- Standard Matrix Multiply (recursively):
  $T(n) = 8T(n/2) + n^2$.
  $2^2/8 = 1/2$ so $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$.

Examples

- Mergesort: $T(n) = 2T(n/2) + n$.
  $2^1/2 = 1$, so $T(n) = \Theta(n \log n)$.
- Binary search: $T(n) = T(n/2) + 2$.
  $2^0/1 = 1$, so $T(n) = \Theta(\log n)$.
- Insertion sort: $T(n) = T(n-1) + n$.
  Can't apply the theorem. Sorry!
- Standard Matrix Multiply (recursively):
  $T(n) = 8T(n/2) + n^2$.
  $2^2/8 = 1/2$ so $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$.

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

In the straightforward implementation, 2×2 case is:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

So the recursion is 8 calls of half size, and the additions take $\Theta(n^2)$ work.

Useful log Notation

- If you want to take the log of (log n), it is written log log n.
- $(\log n)^2$ can be written $\log^2 n$.
- Don't get these confused!
- $\log^* n$ means "the number of times that the log of n must be taken before $n \leq 1$.
  - For example, $65536 = 2^{16}$ so $\log^* 65536 = 4$ since $\log 65536 = 16$, $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$.

Useful log Notation

- If you want to take the log of (log n), it is written log log n.
- $(\log n)^2$ can be written $\log^2 n$.
- Don't get these confused!
- $\log^* n$ means "the number of times that the log of n must be taken before $n \leq 1$.
  - For example, $65536 = 2^{16}$ so $\log^* 65536 = 4$ since $\log 65536 = 16$, $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$.

no notes

Amortized Analysis

Consider this variation on STACK:

```
void init(STACK S);
element examineTop(STACK S);
void push(element x, STACK S);
void pop(int k, STACK S);
```

... where `pop` removes k entries from the stack.

"Local" worst case analysis for `pop`:
$O(n)$ for n elements on the stack.

Given m_1 calls to `push`, m_2 calls to `pop`:
Naive worst case: $m_1 + m_2 \cdot n = m_1 + m_2 \cdot m_1$.

Amortized Analysis

Consider this variation on STACK:

```
void init(STACK S);
element examineTop(STACK S);
void push(element x, STACK S);
void pop(int k, STACK S);
```

... where `pop` removes k entries from the stack.

"Local" worst case analysis for `pop`:
$O(n)$ for n elements on the stack.

Given m_1 calls to `push`, m_2 calls to `pop`:
Naive worst case: $m_1 + m_2 \cdot n = m_1 + m_2 \cdot m_1$.

no notes

Alternate Analysis

Use amortized analysis on multiple calls to `push`, `pop`:

Cannot pop more elements than get pushed onto the stack.

After many pushes, a single pop has high **potential**.

Once that potential has been expended, it is not available for future `pop` operations.

The cost for m_1 pushes and m_2 pops:

$$m_1 + (m_2 + m_1) = O(m_1 + m_2)$$

Alternate Analysis
Use amortized analysis on multiple calls to push, pop:
Cannot pop more elements than get pushed onto the stack.
After many pushes, a single pop has high potential.
Once that potential has been expended, it is not available for future pop operations.
The cost for m_1 pushes and m_2 pops:
$m_1 + (m_2 + m_1) = O(m_1 + m_2)$

Actual number of (constant time) push calls + (Actual number of pop calls + Total potential for the pops)

CLR has an entire chapter on this – we won't go into this much, but we use Amortized Analysis implicitly sometimes.

Creative Design of Algorithms by Induction

Analogy: Induction $\leftrightarrow$ Algorithms

Begin with a problem:

- "Find a solution to problem Q."

Think of Q as a set containing an infinite number of **problem instances**.

Example: Sorting

- Q contains all finite sequences of integers.

Creative Design of Algorithms by Induction
Analogy: Induction $\leftrightarrow$ Algorithms
Begin with a problem:
▲ "Find a solution to problem Q."
Think of Q as a set containing an infinite number of problem instances.
Example: Sorting
▲ Q contains all finite sequences of integers.

Now that we have completed the tool review, we will do two things:

1. Survey algorithms in application areas
2. Try to understand how to create efficient algorithms

This chapter is about the second. The remaining chapters do the second in the context of the first.

I $\leftarrow$ A is reasonably obvious – we often use induction to prove that an algorithm is correct. The intellectual claim of Manber is that I $\rightarrow$ A gives insight into problem solving.

Solving Q

First step:

- Parameterize problem by size: $Q(n)$

Example: Sorting

- $Q(n)$ contains all sequences of n integers.

Q is now an infinite sequence of problems:

- $Q(1), Q(2), ..., Q(n)$

Algorithm: Solve for an instance in $Q(n)$ by solving instances in $Q(i), i < n$ and combining as necessary.

Solving Q
First step:
▸ Parameterize problem by size: $Q(n)$
Example: Sorting
▸ $Q(n)$ contains all sequences of n integers.
Q is now an infinite sequence of problems:
▸ $Q(1), Q(2), ..., Q(n)$
Algorithm: Solve for an instance in $Q(n)$ by solving instances in $Q(i), i < n$ and combining as necessary.

This is a "meta" algorithm – An algorithm for finding algorithms!

Induction

Goal: Prove that we can solve for an instance in $Q(n)$ by assuming we can solve instances in $Q(i), i < n$.

Don't forget the base cases!

Theorem: $\forall n \geq 1$, we can solve instances in $Q(n)$.

- This theorem embodies the **correctness** of the algorithm.

Since an induction proof is mechanistic, this should lead directly to an algorithm (recursive or iterative).

Just one (new) catch:

- Different inductive proofs are possible.
- We want the most **efficient** algorithm!

Induction
Goal: Prove that we can solve for an instance in $Q(n)$ by assuming we can solve instances in $Q(i), i < n$.
Don't forget the base cases!
Theorem: $\forall n \geq 1$, we can solve instances in $Q(n)$.
▸ This theorem embodies the correctness of the algorithm.
Since an induction proof is mechanistic, this should lead directly to an algorithm (recursive or iterative).
Just one (new) catch:
▸ Different inductive proofs are possible.
▸ We want the most efficient algorithm!

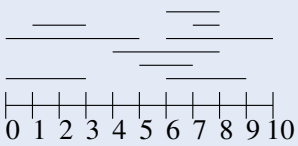
The goal is using Strong Induction.
Correctness is proved by induction.
Example: Sorting

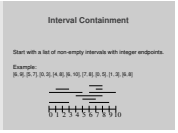
- Sort $n-1$ items, add nth item (insertion sort)
- Sort 2 sets of $n/2$, merge together (mergesort)
- Sort values $< x$ and $> x$ (quicksort)

Interval Containment

Start with a list of non-empty intervals with integer endpoints.

Example:
$[6,9], [5,7], [0,3], [4,8], [6,10], [7,8], [0,5], [1,3], [6,8]$



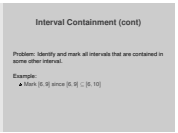


no notes

Interval Containment (cont)

Problem: Identify and mark all intervals that are contained in some other interval.

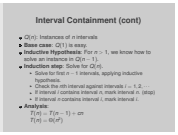
Example:
- Mark $[6,9]$ since $[6,9] \subseteq [6,10]$



$[5,7] \subseteq [4,8]$
$[0,3] \subseteq [0,5]$
$[7,8] \subseteq [6,10]$
$[1,3] \subseteq [0,5]$
$[6,8] \subseteq [6,10]$
$[6,9] \subseteq [6,10]$

Interval Containment (cont)

- $Q(n)$: Instances of n intervals
- **Base case**: $Q(1)$ is easy.
- **Inductive Hypothesis**: For $n > 1$, we know how to solve an instance in $Q(n-1)$.
- **Induction step**: Solve for $Q(n)$.
  - Solve for first $n-1$ intervals, applying inductive hypothesis.
  - Check the nth interval against intervals $i = 1, 2, \cdots$
  - If interval i contains interval n, mark interval n. (stop)
  - If interval n contains interval i, mark interval i.
- **Analysis**:
  $T(n) = T(n-1) + cn$
  $T(n) = \Theta(n^2)$



Base case: Nothing is contained

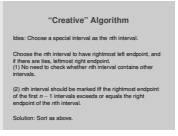
"Creative" Algorithm

Idea: Choose a special interval as the nth interval.

Choose the nth interval to have rightmost left endpoint, and if there are ties, leftmost right endpoint.
(1) No need to check whether nth interval contains other intervals.

(2) nth interval should be marked iff the rightmost endpoint of the first $n-1$ intervals exceeds or equals the right endpoint of the nth interval.

Solution: Sort as above.



In the example, the nth interval is $[7,8]$.
Every other interval has left endpoint to left, or right endpoint to right.
We must keep track of the current right-most endpont.

“Creative” Solution Induction

Induction Hypothesis: Can solve for $Q(n-1)$ AND interval n is the “rightmost” interval AND we know R (the rightmost endpoint encountered so far) for the first $n-1$ segments.

Induction Step: (to solve $Q(n)$)

- Sort by left endpoints
- Solve for first $n-1$ intervals recursively, remembering R.
- If the rightmost endpoint of nth interval is $\leq$ R, then mark the nth interval.
- Else R $\leftarrow$ right endpoint of nth interval.

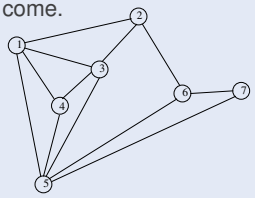
Analysis: $\Theta(n \log n) + \Theta(n)$.
Lesson: Preprocessing, often sorting, can help sometimes.

“Creative” Solution Induction
Induction Hypothesis: Can solve for $Q(n-1)$ AND interval n is the “rightmost” interval AND we know R (the rightmost endpoint encountered so far) for the first $n-1$ segments.
Induction Step: (to solve $Q(n)$)
Sort by left endpoints
Solve for first $n-1$ intervals recursively, remembering R.
If the rightmost endpoint of nth interval is $\leq$ R, then mark the nth interval.
Else R $\leftarrow$ right endpoint of nth interval.
Analysis: $\Theta(n \log n) + \Theta(n)$.
Lesson: Preprocessing, often sorting, can help sometimes.

We strengthened the induction hypothesis. In algorithms, this does cost something.
We must sort.
Analysis: Time for sort + constant time per interval.

Maximal Induced Subgraph

Problem: Given a graph $G = (V, E)$ and an integer k, find a maximal induced subgraph $H = (U, F)$ such that all vertices in H have degree $\geq k$.
Example: Scientists interacting at a conference. Each one will come only if k colleagues come, and they know in advance if somebody won't come.
Example: For $k = 3$.



Solution:

Maximal Induced Subgraph
Problem: Given a graph $G = (V, E)$ and an integer k, find a maximal induced subgraph $H = (U, F)$ such that all vertices in H have degree $\geq k$.
Example: Scientists interacting at a conference. Each one will come only if k colleagues come, and they know in advance if somebody won't come.
Example: For $k = 3$.
Solution:

Induced subgraph: U is a subset of V, F is a subset of E such that both ends of $e \in E$ are members of U.
Solution is: $U = \{1, 3, 4, 5\}$

Max Induced Subgraph Solution

$Q(s, k)$: Instances where $|V| = s$ and k is a fixed integer.

Theorem: $\forall s, k > 0$, we can solve an instance in $Q(s, k)$.

Analysis: Should be able to implement algorithm in time $\Theta(|V| + |E|)$.

Max Induced Subgraph Solution
$Q(s, k)$: Instances where $|V| = s$ and k is a fixed integer.
Theorem: $\forall s, k > 0$, we can solve an instance in $Q(s, k)$.
Analysis: Should be able to implement algorithm in time $\Theta(|V| + |E|)$.

Base Case: $s = 1$ H is the empty graph.
Induction Hypothesis: Assume $s > 1$. we can solve instances of $Q(s-1, k)$.
Induction Step: Show that we can solve an instance of $G(V, E)$ in $Q(s, k)$. Two cases:

(1) Every vertex in G has degree $\geq k$. $H = G$ is the only solution.
(2) Otherwise, let $v \in V$ have degree $< k$. $G - v$ is an instance of $Q(s-1, k)$ which we know how to solve.

By induction, the theorem follows.
Visit all edges to generate degree counts for the vertices. Any vertex with degree below k goes on a queue. Pull the vertices off the queue one by one, and reduce the degree of their neighbors. Add the neighbor to the queue if it drops below k.

Celebrity Problem

In a group of n people, a **celebrity** is somebody whom everybody knows, but who knows no one else.

Problem: If we can ask questions of the form “does person i know person j?” how many questions do we need to find a celebrity, if one exists?

How should we structure the information?

Celebrity Problem
In a group of n people, a celebrity is somebody whom everybody knows, but who knows no one else.
Problem: If we can ask questions of the form “does person i know person j?” how many questions do we need to find a celebrity, if one exists?
How should we structure the information?

no notes

Celebrity Problem (cont)

Formulate as an $n \times n$ boolean matrix M.
$M_{ij} = 1$ iff i knows j.

Example:
$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

A celebrity has all 0's in his row and all 1's in his column.

There can be at most one celebrity.

Clearly, $O(n^2)$ questions suffice. Can we do better?

The celebrity in this example is 4.

Efficient Celebrity Algorithm

Appeal to induction:

- If we have an $n \times n$ matrix, how can we reduce it to an $(n-1) \times (n-1)$ matrix?

What are ways to select the n'th person?

This induction implies that we go backwards. Natural thing to try: pick arbitrary n'th person.
Assume that we can solve for $n-1$. What happens when we add nth person?

- Celebrity candidate in $n-1$ – just ask two questions.
- Celebrity is n – must check $2(n-1)$ positions. $O(n^2)$.
- No celebrity. Again, $O(n^2)$.

So we will have to look for something special. Who can we eliminate? There are only two choices: A celebrity or a non-celebrity. It doesn't make sense to eliminate a celebrity. Is there an easy way to guarentee that we eliminate a non-celeberity on each question?

Efficient Celebrity Algorithm (cont)

Eliminate one person if he is a non-celebrity.

- Strike one row and one column.

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Does 1 know 3? No. 3 is a non-celebrity.
Does 2 know 5? Yes. 2 is a non-celebrity.
Observation: Each question eliminates one non-celebrity.

no notes

Celebrity Algorithm

Algorithm:

1. Ask $n-1$ questions to eliminate $n-1$ non-celebrities. This leaves one candidate who might be a celebrity.
2. Ask $2(n-1)$ questions to check candidate.

Analysis:

- $\Theta(n)$ questions are asked.

Example:

- Does 1 know 2? No. Eliminate 2
- Does 1 know 3? No. Eliminate 3
- Does 1 know 4? Yes. Eliminate 1
- Does 4 know 5? No. Eliminate 5

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

4 remains as candidate.

Why do we need to verify that 4 really is a celebrity?
Becasue we never checked it against 2 and 3, just against 1 and 5.

Maximum Consecutive Subsequence

Given a sequence of integers, find a contiguous subsequence whose sum is maximum.

The sum of an empty subsequence is 0.

- It follows that the maximum subsequence of a sequence of all negative numbers is the empty subsequence.

Example:
2, 11, -9, 3, 4, -6, -7, 7, -3, 5, 6, -2

Maximum subsequence:
7, -3, 5, 6 Sum: 15

Maximum Consecutive Subsequence
Given a sequence of integers, find a contiguous subsequence whose sum is maximum.
The sum of an empty subsequence is 0.
▸ It follows that the maximum subsequence of a sequence of all negative numbers is the empty subsequence.
Example:
2, 11, -9, 3, 4, -6, -7, 7, -3, 5, 6, -2
Maximum subsequence:
7, -3, 5, 6 Sum: 15

no notes

Finding an Algorithm

Induction Hypothesis: We can find the maximum subsequence sum for a sequence of $< n$ numbers.

Note: We have changed the problem.

- First, figure out how to compute the sum.
- Then, figure out how to get the subsequence that computes that sum.

Finding an Algorithm
Induction Hypothesis: We can find the maximum subsequence sum for a sequence of $< n$ numbers.
Note: We have changed the problem.
▸ First, figure out how to compute the sum.
▸ Then, figure out how to get the subsequence that computes that sum.

no notes

Finding an Algorithm (cont)

Induction Hypothesis: We can find the maximum subsequence sum for a sequence of $< n$ numbers.
Let $S = x_1, x_2, \cdots, x_n$ be the sequence.
Base case: $n = 1$
Either $x_1 < 0 \Rightarrow \text{sum} = 0$
Or sum $= x_1$.
Induction Step:

- We know the maximum subsequence SUM(n-1) for $x_1, x_2, \cdots, x_{n-1}$.
- Where does x_n fit in?
  - Either it is not in the maximum subsequence or it ends the maximum subsequence.
- If x_n ends the maximum subsequence, it is appended to trailing maximum subsequence of $x_1, \cdots, x_{n-1}$.

Finding an Algorithm (cont)
Induction Hypothesis: We can find the maximum subsequence sum for a sequence of $< n$ numbers.
Let $S = x_1, x_2, \cdots, x_n$ be the sequence.
Base case: $n = 1$
Either $x_1 < 0 \Rightarrow \text{sum} = 0$
Or sum $= x_1$.
Induction Step:
▸ We know the maximum subsequence SUM(n-1) for $x_1, x_2, \cdots, x_{n-1}$.
▸ Where does x_n fit in?
  ▸ Either it is not in the maximum subsequence or it ends the maximum subsequence.
▸ If x_n ends the maximum subsequence, it is appended to trailing maximum subsequence of $x_1, \cdots, x_{n-1}$.

That is, of the numbers seen **so far**.

Finding an Algorithm (cont)

Need: TRAILINGSUM(n-1) which is the maximum sum of a subsequence that ends $x_1, \cdots, x_{n-1}$.

To get this, we need a stronger induction hypothesis.

Finding an Algorithm (cont)
Need: TRAILINGSUM(n-1) which is the maximum sum of a subsequence that ends $x_1, \cdots, x_{n-1}$.
To get this, we need a stronger induction hypothesis.

no notes

Maximum Subsequence Solution

New Induction Hypothesis: We can find SUM(n-1) and TRAILINGSUM(n-1) for any sequence of $n-1$ integers.

Base case:
SUM(1) = TRAILINGSUM(1) = Max(0, x_1).

Induction step:
SUM(n) = Max(SUM(n-1), TRAILINGSUM(n-1) $+x_n$).
TRAILINGSUM(n) = Max(0, TRAILINGSUM(n-1) $+x_n$).

Maximum Subsequence Solution

New Induction Hypothesis: We can find SUM(n-1) and TRAILINGSUM(n-1) for any sequence of $n-1$ integers.

Base case:
SUM(1) = TRAILINGSUM(1) = Max(0, x_1).

Induction step:
SUM(n) = Max(SUM(n-1), TRAILINGSUM(n-1) $+x_n$).
TRAILINGSUM(n) = Max(0, TRAILINGSUM(n-1) $+x_n$).

no notes

Maximum Subsequence Solution (cont)

Analysis:
Important Lesson: If we calculate and remember some additional values as we go along, we are often able to obtain a more efficient algorithm.

This corresponds to strengthening the induction hypothesis so that we compute more than the original problem (appears to) require.

How do we find sequence as opposed to sum?

Maximum Subsequence Solution (cont)

Analysis:
Important Lesson: If we calculate and remember some additional values as we go along, we are often able to obtain a more efficient algorithm.

This corresponds to strengthening the induction hypothesis so that we compute more than the original problem (appears to) require.

How do we find sequence as opposed to sum?

$O(n)$. $T(n) = T(n-1) + 2$.
Remember position information as well.

The Knapsack Problem

Problem:

- Given an integer capacity K and n items such that item i has an integer size k_i, find a subset of the n items whose sizes exactly sum to K, if possible.
- That is, find $S \subseteq \{1, 2, \cdots, n\}$ such that

$$\sum_{i \in S} k_i = K.$$

Example:
Knapsack capacity $K = 163$.
10 items with sizes

$$4, 9, 15, 19, 27, 44, 54, 68, 73, 101$$

The Knapsack Problem

Problem:

- Given an integer capacity K and n items such that item i has an integer size k_i, find a subset of the n items whose sizes exactly sum to K, if possible.
- That is, find $S \subseteq \{1, 2, \cdots, n\}$ such that

$$\sum_{i \in S} k_i = K.$$

Example:
Knapsack capacity $K = 163$.
10 items with sizes

$$4, 9, 15, 19, 27, 44, 54, 68, 73, 101$$

This version of Knapsack is one of several variations.
Think about solving this for 163. An answer is:

$$S = \{9, 27, 54, 73\}$$

Now, try solving for $K = 164$. An answer is:

$$S = \{19, 44, 101\}.$$

There is no relationship between these solutions!

Knapsack Algorithm Approach

Instead of parameterizing the problem just by the number of items n, we parameterize by both n and by K.

$P(n, K)$ is the problem with n items and capacity K.

First consider the decision problem: Is there a subset S?

Induction Hypothesis:
We know how to solve $P(n-1, K)$.

Knapsack Algorithm Approach

Instead of parameterizing the problem just by the number of items n, we parameterize by both n and by K.

$P(n, K)$ is the problem with n items and capacity K.

First consider the decision problem: Is there a subset S?

Induction Hypothesis:
We know how to solve $P(n-1, K)$.

Is there a subset S such that $\sum S_i = K$?

Knapsack Induction

Induction Hypothesis:
We know how to solve $P(n-1, K)$.

Solving $P(n, K)$:

- If $P(n-1, K)$ has a solution, then it is also a solution for $P(n, K)$.
- Otherwise, $P(n, K)$ has a solution iff $P(n-1, K-k_n)$ has a solution.

So what should the induction hypothesis really be?

Knapsack Induction
Induction Hypothesis:
We know how to solve $P(n-1, K)$.
Solving $P(n, K)$:
- If $P(n-1, K)$ has a solution, then it is also a solution for $P(n, K)$.
- Otherwise, $P(n, K)$ has a solution iff $P(n-1, K-k_n)$ has a solution.
So what should the induction hypothesis really be?

But... I don't know how to solve $P(n-1, K-k_n)$ since it is not in my induction hypothesis! So, we must strengthen the induction hypothesis.

New Induction Hypothesis:
We know how to solve $P(n-1, k), 0 \leq k \leq K$.

Knapsack: New Induction

- **New Induction Hypothesis**:
  We know how to solve $P(n-1, k), 0 \leq k \leq K$.
- To solve $P(n, K)$:
  If $P(n-1, K)$ has a solution,
  Then $P(n, K)$ has a solution.
  Else If $P(n-1, K-k_n)$ has a solution,
  Then $P(n, K)$ has a solution.
  Else $P(n, K)$ has no solution.

Knapsack: New Induction
- New Induction Hypothesis:
  We know how to solve $P(n-1, k), 0 \leq k \leq K$.
- To solve $P(n, K)$:
  If $P(n-1, K)$ has a solution,
  Then $P(n, K)$ has a solution.
  Else If $P(n-1, K-k_n)$ has a solution,
  Then $P(n, K)$ has a solution.
  Else $P(n, K)$ has no solution.

Need to solve two subproblems: $P(n-1, k)$ and $P(n-1, k-k_n)$.

Algorithm Complexity

- Resulting algorithm complexity:
  $T(n) = 2T(n-1) + c \qquad n \geq 2$
  $T(n) = \Theta(2^n)$ by expanding sum.
- But, there are only $n(K+1)$ problems defined.
  - It must be that problems are being re-solved many times by this algorithm. Don't do that.

Algorithm Complexity
- Resulting algorithm complexity:
  $T(n) = 2T(n-1) + c \qquad n \geq 2$
  $T(n) = \Theta(2^n)$ by expanding sum.
- But, there are only $n(K+1)$ problems defined.
  - It must be that problems are being re-solved many times by this algorithm. Don't do that.

Problem: Can't use Theorem 3.4 in this form.

Efficient Algorithm Implementation

The key is to avoid re-computing subproblems.

Implementation:

- Store an $n \times (K+1)$ matrix to contain solutions for all the $P(i, k)$.
- Fill in the table row by row.
- Alternately, fill in table using logic above.

Analysis:
$T(n) = \Theta(nK)$.
Space needed is also $\Theta(nK)$.

Efficient Algorithm Implementation
The key is to avoid re-computing subproblems.
Implementation:
- Store an $n \times (K+1)$ matrix to contain solutions for all the $P(i, k)$.
- Fill in the table row by row.
- Alternately, fill in table using logic above.
Analysis:
$T(n) = \Theta(nK)$.
Space needed is also $\Theta(nK)$.

To solve $P(i, k)$ look at entry in the table.
If it is marked, then OK.
Otherwise solve recursively.
Initially, fill in all $P(i, 0)$.

Example

$K = 10$, with 5 items having size 9, 2, 7, 4, 1.

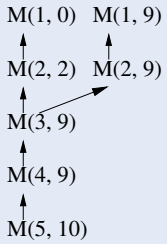
	0	1	2	3	4	5	6	7	8	9	10
$k_1 = 9$	O	–	–	–	–	–	–	–	–	I	–
$k_2 = 2$	O	–	I	–	–	–	–	–	–	O	–
$k_3 = 7$	O	–	O	–	–	–	–	I	–	I/O	–
$k_4 = 4$	O	–	O	–	I	–	I	O	–	O	–
$k_5 = 1$	O	I	O	I	O	I	O	I/O	I	O	I

Key:
– No solution for $P(i, k)$
O Solution(s) for $P(i, k)$ with i omitted.
I Solution(s) for $P(i, k)$ with i included.
I/O Solutions for $P(i, k)$ both with i included and with i omitted.

Example: M(3, 9) contains O because $P(2, 9)$ has a solution.
It contains I because $P(2, 2) = P(2, 9 - 7)$ has a solution.
How can we find a solution to $P(5, 10)$ from M?
How can we find **all** solutions for $P(5, 10)$?

Solution Graph

Find all solutions for $P(5, 10)$.



The result is an n-level DAG.

Alternative approach:
Do not precompute matrix. Instead, solve subproblems as necessary, marking in the array during backtracking.
To avoid storing the large array, use hashing for storing (and retrieving) subproblem solutions.

Dynamic Programming

This approach of storing solutions to subproblems in a table is called **dynamic programming**.

It is useful when the number of distinct subproblems is not too large, but subproblems are executed repeatedly.

Implementation: Nested `for` loops with logic to fill in a single entry.

Most useful for **optimization problems**.

no notes

Fibonacci Sequence

```
int Fibr(int n) {
  if (n <= 1) return 1;          // Base case
  return Fibr(n-1) + Fibr(n-2); // Recursion
}
```

- Cost is Exponential. Why?
- If we could eliminate redundancy, cost would be greatly reduced.

Essentially, we are making as many function calls as the value of the Fibonacci sequence itself. It is roughly (though not quite) two function calls of size $n - 1$ each.

Fibonacci Sequence (cont)

- Keep a table

```
int Fibrt(int n, int* Values) {
  // Assume Values has at least n slots, and
  // all slots are initialized to 0
  if (n <= 1) return 1;  // Base case
  if (Values[n] == 0)    // Compute and store
    Values[n] = Fibrt(n-1, Values) +
                Fibrt(n-2, Values);
  return Values[n];
}
```

- Cost?
- We don't need table, only last 2 values.
  - Key is working bottom up.

Fibonacci Sequence (cont)
▸ Keep a table
int Fibrt(int n, int* Values) { // Assume Values has at least n slots, and // all slots are initialized to 0 if (n <= 1) return 1; // Base case if (Values[n] == 0) // Compute and store Values[n] = Fibrt(n-1, Values) + Fibrt(n-2, Values); return Values[n]; }
▸ Cost?
▸ We don't need table, only last 2 values.
  ▸ Key is working bottom up.

no notes

Chained Matrix Multiplication

Problem: Compute the product of n matrices

$$M = M_1 \times M_2 \times \cdots \times M_n$$

as efficiently as possible.

If A is $r \times s$ and B is $s \times t$, then
COST$(A \times B) =$
SIZE$(A \times B) =$

If C is $t \times u$ then
COST$((A \times B) \times C) =$
COST$((A \times (B \times C))) =$

Chained Matrix Multiplication
Problem: Compute the product of n matrices
$M = M_1 \times M_2 \times \cdots \times M_n$
as efficiently as possible.
If A is $r \times s$ and B is $s \times t$, then
COST$(A \times B) =$
SIZE$(A \times B) =$
If C is $t \times u$ then
COST$((A \times B) \times C) =$
COST$((A \times (B \times C))) =$

$A \times B$:
COST: rst
SIZE: $r \times t$

$rst + (r \times t)(t \times u) = rst + rtu$.
$(r \times s)[(s \times t)(t \times u)] = (r \times s)(s \times u)$.
$rsu + stu$.

Order Matters

Example:

$$A = 2 \times 8; B = 8 \times 5; C = 5 \times 20$$

COST$((A \times B) \times C) =$
COST$(A \times (B \times C)) =$

View as binary trees:

Order Matters
Example:
$A = 2 \times 8; B = 8 \times 5; C = 5 \times 20$
COST$((A \times B) \times C) =$
COST$(A \times (B \times C)) =$
View as binary trees:

$2 \cdot 8 \cdot 5 + 2 \cdot 5 \cdot 20 = 280$.
$8 \cdot 5 \cdot 20 + 2 \cdot 8 \cdot 20 = 1120$.

Tree for $((A \times B) \times C) =: \cdot \cdot ABC$
Tree for $(A \times (B \times C) =: \cdot A \cdot BC$

We would like to find the optimal order for computation before actually doing the matrix multiplications.

Chained Matrix Induction

Induction Hypothesis: We can find the optimal evaluation tree for the multiplication of $\leq n-1$ matrices.

Induction Step: Suppose that we start with the tree for:

$$M_1 \times M_2 \times \cdots \times M_{n-1}$$

and try to add M_n.

Two obvious choices:
1. Multiply $M_{n-1} \times M_n$ and replace M_{n-1} in the tree with a subtree.
2. Multiply M_n by the result of $P(n-1)$: make a new root.

Visually, adding M_n may radically order the (optimal) tree.

Chained Matrix Induction
Induction Hypothesis: We can find the optimal evaluation tree for the multiplication of $\leq n-1$ matrices.
Induction Step: Suppose that we start with the tree for:
$M_1 \times M_2 \times \cdots \times M_{n-1}$
and try to add M_n.
Two obvious choices:
1. Multiply $M_{n-1} \times M_n$ and replace M_{n-1} in the tree with a subtree.
2. Multiply M_n by the result of $P(n-1)$: make a new root.
Visually, adding M_n may radically order the (optimal) tree.

Problem: There is no reason to believe that either of these yields the optimal ordering.

Alternate Induction

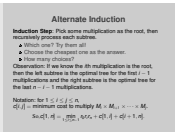
Induction Step: Pick some multiplication as the root, then recursively process each subtree.

- Which one? Try them all!
- Choose the cheapest one as the answer.
- How many choices?

Observation: If we know the ith multiplication is the root, then the left subtree is the optimal tree for the first $i-1$ multiplications and the right subtree is the optimal tree for the last $n-i-1$ multiplications.

Notation: for $1 \le i \le j \le n$,
$c[i,j]$ = minimum cost to multiply $M_i \times M_{i+1} \times \cdots \times M_j$.

$$\text{So, } c[1,n] = \min_{1 \le i \le n-1} r_0 r_i r_n + c[1,i] + c[i+1,n].$$



$n-1$ choices for root.

Analysis

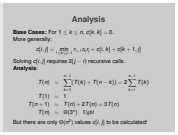
Base Cases: For $1 \le k \le n$, $c[k,k] = 0$.
More generally:

$$c[i,j] = \min_{1 \le k \le j-1} r_{i-1} r_k r_j + c[i,k] + c[k+1,j]$$

Solving $c[i,j]$ requires $2(j-i)$ recursive calls.
Analysis:

$$\begin{aligned} T(n) &= \sum_{k=1}^{n-1} (T(k) + T(n-k)) = 2\sum_{k=1}^{n-1} T(k) \\ T(1) &= 1 \\ T(n+1) &= T(n) + 2T(n) = 3T(n) \\ T(n) &= \Theta(3^n) \quad \text{Ugh!} \end{aligned}$$

But there are only $\Theta(n^2)$ values $c[i,j]$ to be calculated!



2 calls for each root choice, with $(j-i)$ choices for root. But, these don't all have equal cost.

$T(n+1) = 2\sum_{i=1}^{n} T(k)$
So:

$$\begin{aligned} T(n+1) - T(n) &= 2\sum_{i=1}^{n} T(k) - 2\sum_{i=1}^{n-1} T(k) \\ &= 2T(n) \\ T(n+1) &= 3T(n) \end{aligned}$$

Actually, since $j > i$, only about half that needs to be done.

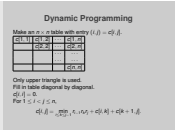
Dynamic Programming

Make an $n \times n$ table with entry $(i,j) = c[i,j]$.

$c[1,1]$	$c[1,2]$	$\cdots$	$c[1,n]$
	$c[2,2]$	$\cdots$	$c[2,n]$
		$\cdots$	$\cdots$
		$\cdots$	$\cdots$
			$c[n,n]$

Only upper triangle is used.
Fill in table diagonal by diagonal.
$c[i,i] = 0$.
For $1 \le i < j \le n$,

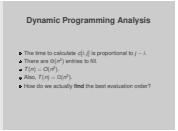
$$c[i,j] = \min_{i \le k \le j-1} r_{i-1} r_k r_j + c[i,k] + c[k+1,j].$$



The array is processed starting with the middle diagonal (all zeros), diagonal by diagonal toward the upper left corner.

Dynamic Programming Analysis

- The time to calculate $c[i,j]$ is proportional to $j-i$.
- There are $\Theta(n^2)$ entries to fill.
- $T(n) = O(n^3)$.
- Also, $T(n) = \Omega(n^3)$.
- How do we actually **find** the best evaluation order?



For middle diagonal of size $n/2$, each costs $n/2$.

For each $c[i,j]$, remember the k (the root of the tree) that minimizes the expression.
So, store in the table the next place to go.

Summary

- Dynamic programming can often be added to an inductive proof to make the resulting algorithm as efficient as possible.
- Can be useful when divide and conquer **fails** to be efficient.
- Usually applies to optimization problems.
- Requirements for dynamic programming:
  1. Repeated solution of subproblems
  2. Small number of subproblems, small amount of information to store for each subproblem.
  3. Base case easy to solve.
  4. Easy to solve one subproblem given solutions to smaller subproblems.

Summary
- Dynamic programming can often be added to an inductive proof to make the resulting algorithm as efficient as possible.
- Can be useful when divide and conquer fails to be efficient.
- Usually applies to optimization problems.
- Requirements for dynamic programming:
  1. Repeated solution of subproblems
  2. Small number of subproblems, small amount of information to store for each subproblem.
  3. Base case easy to solve.
  4. Easy to solve one subproblem given solutions to smaller subproblems.

no notes

Sorting

Each record contains a field called the **key**.
Linear order: comparison.

The Sorting Problem

Given a sequence of records $R_1, R_2, ..., R_n$ with key values $k_1, k_2, ..., k_n$, respectively, arrange the records into any order s such that records $R_{s_1}, R_{s_2}, ..., R_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$.

Measures of cost:

- Comparisons
- Swaps

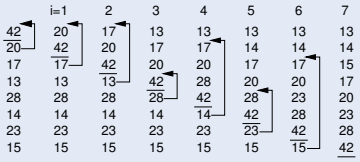
Sorting
Each record contains a field called the key.
Linear order: comparison.
The Sorting Problem
Given a sequence of records $R_1, R_2, ..., R_n$ with key values $k_1, k_2, ..., k_n$, respectively, arrange the records into any order s such that records $R_{s_1}, R_{s_2}, ..., R_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$.
Measures of cost:
- Comparisons
- Swaps

Linear order means: $a < b$ and $b < c \Rightarrow a < c$.

More simply, sorting means to put keys in ascending order.

Insertion Sort

```
void inssort(Elem* A, int n) { // Insertion Sort
  for (int i=1; i<n; i++)   // Insert i'th record
    for (int j=i; (j>0) && (A[j].key<A[j-1].key);
         j--)
      swap(A, j, j-1);
}
```



Best Case:
Worst Case:
Average Case:

Insertion Sort
Best Case:
Worst Case:
Average Case:

Best case is 0 swaps, $n - 1$ comparisons.
Worst case is $n^2/2$ swaps and compares.
Average case is $n^2/4$ swaps and compares.

Insertion sort has great best-case performance.

Exchange Sorting

- **Theorem**: Any sort restricted to swapping adjacent records must be $\Omega(n^2)$ in the worst and average cases.
- **Proof**:
  - For any permutation P, and any pair of positions i and j, the relative order of i and j must be wrong in either P or the inverse of P.
  - Thus, the total number of swaps required by P and the inverse of P MUST be

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Exchange Sorting
- Theorem: Any sort restricted to swapping adjacent records must be $\Omega(n^2)$ in the worst and average cases.
- Proof:
  - For any permutation P, and any pair of positions i and j, the relative order of i and j must be wrong in either P or the inverse of P.
  - Thus, the total number of swaps required by P and the inverse of P MUST be
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

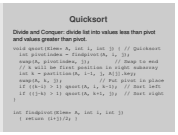
$n^2/4$ is the average distance from a record to its position in the sorted output.

Quicksort

Divide and Conquer: divide list into values less than pivot and values greater than pivot.

```
void qsort(Elem* A, int i, int j) { // Quicksort
  int pivotindex = findpivot(A, i, j);
  swap(A, pivotindex, j);         // Swap to end
  // k will be first position in right subarray
  int k = partition(A, i-1, j, A[j].key);
  swap(A, k, j);              // Put pivot in place
  if ((k-i) > 1) qsort(A, i, k-1);  // Sort left
  if ((j-k) > 1) qsort(A, k+1, j);  // Sort right
}

int findpivot(Elem* A, int i, int j)
  { return (i+j)/2; }
```

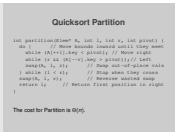



Initial call: `qsort(array, 0, n-1);`

Quicksort Partition

```
int partition(Elem* A, int l, int r, int pivot) {
  do {      // Move bounds inward until they meet
    while (A[++l].key < pivot); // Move right
    while (r && (A[--r].key > pivot));// Left
    swap(A, l, r);       // Swap out-of-place vals
  } while (l < r);       // Stop when they cross
  swap(A, l, r);         // Reverse wasted swap
  return l;     // Return first position in right
}
```

The cost for Partition is $\Theta(n)$.



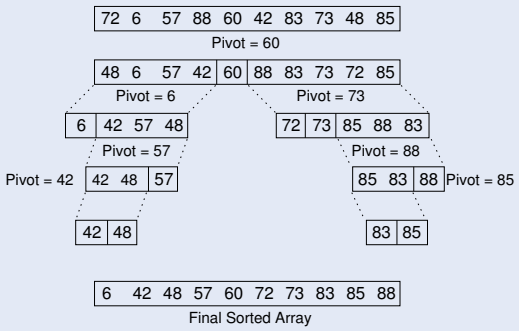
no notes

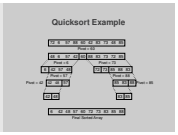
Partition Example

Initial	72	6	57	88	85	42	83	73	48	60
	l									r
Pass 1	72	6	57	88	85	42	83	73	48	60
		l							r	
Swap 1	48	6	57	88	85	42	83	73	72	60
		l							r	
Pass 2	48	6	57	88	85	42	83	73	72	60
				l		r				
Swap 2	48	6	57	42	85	88	83	73	72	60
				l		r				
Pass 3	48	6	57	42	85	88	83	73	72	60
					l,r					

no notes

Quicksort Example





no notes

Cost for Quicksort

Best Case: Always partition in half.

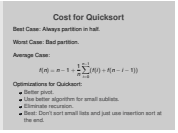
Worst Case: Bad partition.

Average Case:

$$f(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(f(i) + f(n-i-1))$$

Optimizations for Quicksort:

- Better pivot.
- Use better algorithm for small sublists.
- Eliminate recursion.
- Best: Don't sort small lists and just use insertion sort at the end.



Think about when the partition is bad. Note the FindPivot function that we used is pretty good, especially compared to taking the first (or last) value.
Also, think about the distribution of costs: Line up all the permuations from most expensive to cheapest. How many can be expensive? The area under this curve must be low, since the average cost is $\Theta(n \log n)$, but some of the values cost $\Theta(n^2)$. So there can be VERY few of the expensive ones.

This optimization means, for list threshold T, that no element is more than T positions from its destination. Thus, insertion sort's best case is nearly realized. Cost is at worst nT.

Quicksort Average Cost

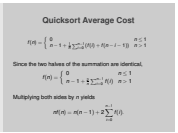
$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(f(i) + f(n-i-1)) & n > 1 \end{cases}$$

Since the two halves of the summation are identical,

$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} f(i) & n > 1 \end{cases}$$

Multiplying both sides by n yields

$$nf(n) = n(n-1) + 2\sum_{i=0}^{n-1} f(i).$$



This is a "recurrence with full history".

Think about what the pieces correspond to.
To do Quicksort on an array of size n, we must:

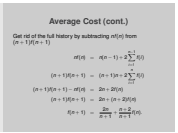
- Partation: Cost n
- Findpivot: Cost c
- Do the recursion: Cost dependent on the pivot's final position.

These parts are modeled by the equation, including the average over all the cases for position of the pivot.

Average Cost (cont.)

Get rid of the full history by subtracting $nf(n)$ from $(n+1)f(n+1)$

$$\begin{aligned} nf(n) &= n(n-1) + 2\sum_{i=1}^{n-1} f(i) \\ (n+1)f(n+1) &= (n+1)n + 2\sum_{i=1}^{n} f(i) \\ (n+1)f(n+1) - nf(n) &= 2n + 2f(n) \\ (n+1)f(n+1) &= 2n + (n+2)f(n) \\ f(n+1) &= \frac{2n}{n+1} + \frac{n+2}{n+1}f(n). \end{aligned}$$

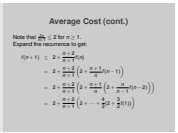


no notes

Average Cost (cont.)

Note that $\frac{2n}{n+1} \leq 2$ for $n \geq 1$.
Expand the recurrence to get:

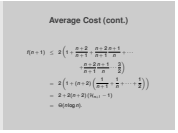
$$\begin{aligned} f(n+1) &\leq 2 + \frac{n+2}{n+1}f(n) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}f(n-1)\right) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}\left(2 + \frac{n}{n-1}f(n-2)\right)\right) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \cdots + \frac{4}{3}(2 + \frac{3}{2}f(1))\right) \end{aligned}$$



no notes

Average Cost (cont.)

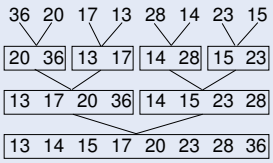
$$
\begin{aligned}
f(n+1) &\leq 2\left(1+\frac{n+2}{n+1}+\frac{n+2}{n+1}\frac{n+1}{n}+\cdots \right.\\
&\qquad \left. +\frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right)\\
&= 2\left(1+(n+2)\left(\frac{1}{n+1}+\frac{1}{n}+\cdots+\frac{1}{2}\right)\right)\\
&= 2+2(n+2)\left(\mathcal{H}_{n+1}-1\right)\\
&= \Theta(n\log n).
\end{aligned}
$$

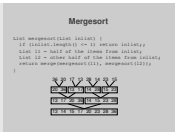


$\mathcal{H}_{n+1} = \Theta(\log n)$

Mergesort

```
List mergesort(List inlist) {
  if (inlist.length() <= 1) return inlist;;
  List l1 = half of the items from inlist;
  List l2 = other half of the items from inlist;
  return merge(mergesort(l1), mergesort(l2));
}
```



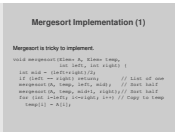


no notes

Mergesort Implementation (1)

Mergesort is tricky to implement.

```
void mergesort(Elem* A, Elem* temp,
               int left, int right) {
  int mid = (left+right)/2;
  if (left == right) return;        // List of one
  mergesort(A, temp, left, mid);    // Sort half
  mergesort(A, temp, mid+1, right);// Sort half
  for (int i=left; i<=right; i++) // Copy to temp
    temp[i] = A[i];
```

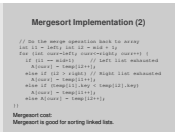



This implementation requires a second array.

Mergesort Implementation (2)

```
  // Do the merge operation back to array
  int i1 = left; int i2 = mid + 1;
  for (int curr=left; curr<=right; curr++) {
    if (i1 == mid+1)     // Left list exhausted
      A[curr] = temp[i2++];
    else if (i2 > right) // Right list exhausted
      A[curr] = temp[i1++];
    else if (temp[i1].key < temp[i2].key)
      A[curr] = temp[i1++];
    else A[curr] = temp[i2++];
}}
```

Mergesort cost:
Mergesort is good for sorting linked lists.



Mergesort cost: $\Theta(n \log n)$

Linked lists: Send records to alternating linked lists, mergesort each, then merge.

Heaps

Heap: Complete binary tree with the **Heap Property**:

- Min-heap: all values less than child values.
- Max-heap: all values greater than child values.

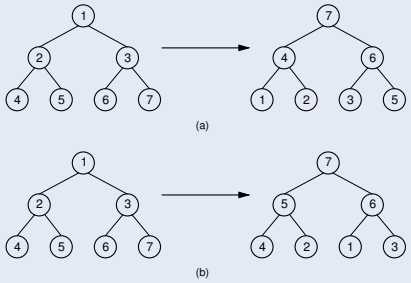
The values in a heap are **partially ordered**.

Heap representation: normally the array based complete binary tree representation.

Heaps
Heap: Complete binary tree with the Heap Property:
Min-heap: all values less than child values.
Max-heap: all values greater than child values.
The values in a heap are partially ordered.
Heap representation: normally the array based complete binary tree representation.

no notes

Building the Heap



(a) requires exchanges (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6).
(b) requires exchanges (5-2), (7-3), (7-1), (6-1).

Building the Heap
(a) requires exchanges (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6).
(b) requires exchanges (5-2), (7-3), (7-1), (6-1).

This is a Max Heap
How to get a good number of exchanges? By induction.
Heapify the root's subtrees, then push the root to the correct level.

Siftdown

```
void heap::siftdown(int pos) { // Sift ELEM down
  assert((pos >= 0) && (pos < n));
  while (!isLeaf(pos)) {
    int j = leftchild(pos);
    if ((j<(n-1)) &&
        (Heap[j].key < Heap[j+1].key))
      j++; // j now index of child with > value
    if (Heap[pos].key >= Heap[j].key) return;
    swap(Heap, pos, j);
    pos = j;  // Move down
  }
}
```


Siftdown

no notes

BuildHeap

For fast heap construction:

- Work from high end of array to low end.
- Call `siftdown` for each item.
- Don't need to call `siftdown` on leaf nodes.

```
void heap::buildheap()        // Heapify contents
  { for (int i=n/2-1; i>=0; i--) siftdown(i); }
```

Cost for heap construction:

$$\sum_{i=1}^{\log n} (i-1)\frac{n}{2^i} \approx n.$$

BuildHeap
For fast heap construction:
Work from high end of array to low end.
Call siftdown for each item.
Don't need to call siftdown on leaf nodes.
Cost for heap construction:

$(i-1)$ is number of steps down, $n/2^i$ is number of nodes at that level.

The intuition for why this cost is $\Theta(n)$ is important. Fundamentally, the issue is that nearly all nodes in a tree are close to the bottom, and we are (worst case) pushing all nodes down to the bottom. So most nodes have nowhere to go, leading to low cost.

Heapsort

Heapsort uses a max-heap.

```
void heapsort(Elem* A, int n) { // Heapsort
  heap H(A, n, n);              // Build the heap
  for (int i=0; i<n; i++)       // Now sort
    H.removemax(); // Value placed at end of heap
}
```

Cost of Heapsort:

Cost of finding k largest elements:

2014-05-02 CS 5114 └Heapsort

Cost of Heapsort: $\Theta(n \log n)$
Cost of finding k largest elements: $\Theta(k \log n + n)$.

- Time to build heap: $\Theta(n)$.
- Time to remove least element: $\Theta(\log n)$.

Compare Heapsort to sorting with BST:

- BST is expensive in space (overhead), potential bad balance, BST does not take advantage of having all records available in advance.
- Heap is space efficient, balanced, and building initial heap is efficient.

Binsort

A simple, efficient sort:

```
for (i=0; i<n; i++)
  B[key(A[i])] = A[i];
```

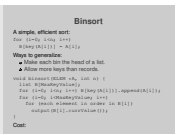
Ways to generalize:

- Make each bin the head of a list.
- Allow more keys than records.

```
void binsort(ELEM *A, int n) {
  list B[MaxKeyValue];
  for (i=0; i<n; i++) B[key(A[i])].append(A[i]);
  for (i=0; i<MaxKeyValue; i++)
    for (each element in order in B[i])
      output(B[i].currValue());
}
```

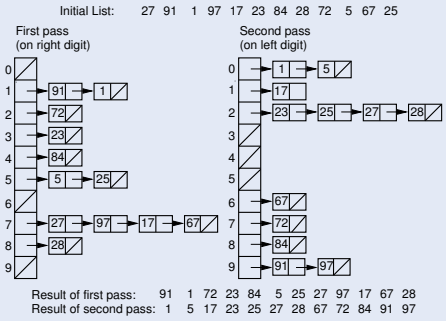
Cost:

2014-05-02 CS 5114 └Binsort

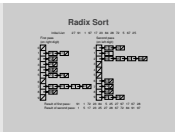


The simple version only works for a permutation of 0 to $n-1$, but it is truly $O(n)$!
Support duplicatesI.e., larger key spaceCost might look like $\Theta(n)$.
Oops! It is ctually, $\Theta(n * \text{Maxkeyvalue})$.
Maxkeyvalue could be $\mathrm{O}(n^2)$ or worse.

Radix Sort



2014-05-02 CS 5114 └Radix Sort



no notes

Radix Sort Algorithm (1)

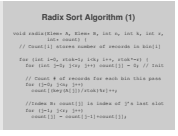
```
void radix(Elem* A, Elem* B, int n, int k, int r,
           int* count) {
  // Count[i] stores number of records in bin[i]

  for (int i=0, rtok=1; i<k; i++, rtok*=r) {
    for (int j=0; j<r; j++) count[j] = 0; // Init

    // Count # of records for each bin this pass
    for (j=0; j<n; j++)
      count[(key(A[j])/rtok)%r]++;

    //Index B: count[j] is index of j's last slot
    for (j=1; j<r; j++)
      count[j] = count[j-1]+count[j];
```


2014-05-02 CS 5114 └Radix Sort Algorithm (1)



no notes

Radix Sort Algorithm (2)

```
    // Put recs into bins working from bottom
    //Bins fill from bottom so j counts downwards
    for (j=n-1; j>=0; j--)
      B[--count[(key(A[j])/rtok)%r]] = A[j];
    for (j=0; j<n; j++) A[j] = B[j]; // Copy B->A
  }
}
```

Cost: $\Theta(nk + rk)$.

How do n, k and r relate?

Radix Sort Algorithm (2)

// Put recs into bins working from bottom
//Bins fill from bottom so j counts downwards
for (j=n-1; j>=0; j--)
B[--count[(key(A[j])/rtok)%r]] = A[j];
for (j=0; j<n; j++) A[j] = B[j]; // Copy B->A
}
}

Cost: Θ(nk + rk).

How do n, k and r relate?

r can be viewed as a constant.
$k \geq \log n$ if there are n distinct keys.

Radix Sort Example

Radix Sort Example

no notes

Sorting Lower Bound

Want to prove a lower bound for *all possible* sorting algorithms.

Sorting is $\mathrm{O}(n \log n)$.

Sorting I/O takes $\Omega(n)$ time.

Will now prove $\Omega(n \log n)$ lower bound.

Form of proof:

- Comparison based sorting can be modeled by a binary tree.
- The tree must have $\Omega(n!)$ leaves.
- The tree must be $\Omega(n \log n)$ levels deep.

Sorting Lower Bound

Want to prove a lower bound for all possible sorting algorithms.

Sorting is O(n log n).

Sorting I/O takes Ω(n) time.

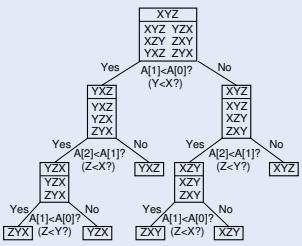
Will now prove Ω(n log n) lower bound.

Form of proof:

- Comparison based sorting can be modeled by a binary tree.
- The tree must have Ω(n!) leaves.
- The tree must be Ω(n log n) levels deep.

no notes

Decision Trees



- There are $n!$ permutations, and at least 1 node for each.
- A tree with n nodes has at least $\log n$ levels.
- Where is the worst case in the decision tree?

Decision Trees

- There are n! permutations, and at least 1 node for each.
- A tree with n nodes has at least log n levels.
- Where is the worst case in the decision tree?

no notes

Lower Bound Analysis

$$\log n! \leq \log n^n = n \log n.$$

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} \geq \frac{1}{2}(n \log n - n).$$

- So, $\log n! = \Theta(n \log n)$.
- Using the decision tree model, what is the average depth of a node?
- This is also $\Theta(\log n!)$.

Lower Bound Analysis

$\log n! \leq \log n^n = n \log n$.

$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} \geq \frac{1}{2}(n \log n - n)$.

- So, $\log n! = \Theta(n \log n)$.
- Using the decision tree model, what is the average depth of a node?
- This is also $\Theta(\log n!)$.

$\log n -$ (1 or 2).

A Search Model (1)

Problem:
Given:

- A list L, of n elements
- A search key X

Solve: Identify one element in L which has key value X, if any exist.

Model:

- The key values for elements in L are unique.
- One comparison determines $<, =, >$.
- Comparison is our only way to find ordering information.
- Every comparison costs the same.

A Search Model (1)

Problem:
Given:

- A list L, of n elements
- A search key X

Solve: Identify one element in L which has key value X, if any exist.

Model:

- The key values for elements in L are unique.
- One comparison determines $<, =, >$.
- Comparison is our only way to find ordering information.
- Every comparison costs the same.

What if the key values are not unique? Probably the cost goes down, not up. This is an assumption for *analysis*, not for implementation.

We would have a slightly different model (though no asymptotic change in cost) if our only comparison test was $<$. We would have a very different model if our only comparison was $= / \neq$.

A comparison-based model.

String data might require comparisons with very different costs.

A Search Model (2)

Goal: Solve the problem using the minimum number of comparisons.

- Cost model: Number of comparisons.
- (Implication) Access to every item in L costs the same (array).

Is this a reasonable model and goal?

A Search Model (2)

Goal: Solve the problem using the minimum number of comparisons.

- Cost model: Number of comparisons.
- (Implication) Access to every item in L costs the same (array).

Is this a reasonable model and goal?

- We are assuming that the # of comparisons is proportional to runtime.
- Might not always share an array (assumption that all accesses are equal). For example, linked lists.
- We assume there is no relationship between value X and its position.

Linear Search

General algorithm strategy: Reduce the problem.

- Compare X to the first element.
- If not done, then solve the problem for $n-1$ elements.

```
Position linear_search(L, lower, upper, X) {
  if L[lower] = X then
    return lower;
  else if lower = upper then
    return -1;
  else
    return linear_search(L, lower+1, upper, X);
}
```

What equation represents the worst case cost?

Linear Search

General algorithm strategy: Reduce the problem.

- Compare X to the first element.
- If not done, then solve the problem for $n-1$ elements.

```
Position linear_search(L, lower, upper, X) {
  if L[lower] = X then
    return lower;
  else if lower = upper then
    return -1;
  else
    return linear_search(L, lower+1, upper, X);
}
```

What equation represents the worst case cost?

$$f(n) = \begin{cases} 1 & n = 1 \\ f(n-1) + 1 & n > 1 \end{cases}$$

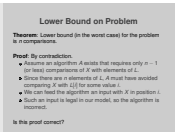
Lower Bound on Problem

Theorem: Lower bound (in the worst case) for the problem is n comparisons.

Proof: By contradiction.

- Assume an algorithm A exists that requires only $n-1$ (or less) comparisons of X with elements of L.
- Since there are n elements of L, A must have avoided comparing X with $L[i]$ for some value i.
- We can feed the algorithm an input with X in position i.
- Such an input is legal in our model, so the algorithm is incorrect.

Is this proof correct?

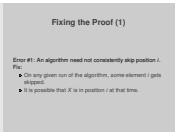


Be careful about assumptions on how an algorithm might (must) behave.
After all, where do new, clever algorithms come from? From different behavior than was previously assumed!

Fixing the Proof (1)

Error #1: An algorithm need not consistently skip position i.
Fix:

- On any given run of the algorithm, *some* element i gets skipped.
- It is possible that X is in position i at that time.

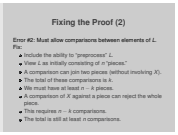


no notes

Fixing the Proof (2)

Error #2: Must allow comparisons between elements of L.
Fix:

- Include the ability to "preprocess" L.
- View L as initially consisting of n "pieces."
- A comparison can join two pieces (without involving X).
- The total of these comparisons is k.
- We must have at least $n-k$ pieces.
- A comparison of X against a piece can reject the whole piece.
- This requires $n-k$ comparisons.
- The total is still at least n comparisons.



no notes

Average Cost

How many comparisons does linear search do on average?

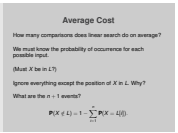
We must know the probability of occurrence for each possible input.

(Must X be in L?)

Ignore everything except the position of X in L. Why?

What are the $n+1$ events?

$$\mathbf{P}(X \notin L) = 1 - \sum_{i=1}^{n} \mathbf{P}(X = L[i]).$$



No, X might not be in L! What is this probability?

The actual values of other elements is irrelevent to the search routine.

$L[1], L[2], ..., L[n]$ and *not found*.

Assume that array bounds are $1..n$.

Average Cost Equation

Let $k_i = i$ be the number of comparisons when $X = L[i]$.
Let $k_0 = n$ be the number of comparisons when $X \notin L$.

Let p_i be the probability that $X = L[i]$.
Let p_0 be the probability that $X \notin L[i]$ for any i.

$$\begin{aligned} f(n) &= k_0 p_0 + \sum_{i=1}^{n} k_i p_i \\ &= np_0 + \sum_{i=1}^{n} i p_i \end{aligned}$$

What happens to the equation if we assume all p_i's are equal (except p_0)?

Average Cost Equation

Let $k_i = i$ be the number of comparisons when $X = L[i]$.
Let $k_0 = n$ be the number of comparisons when $X \notin L$.

Let p_i be the probability that $X = L[i]$.
Let p_0 be the probability that $X \notin L[i]$ for any i.

$$\begin{aligned} f(n) &= k_0 p_0 + \sum_{i=1}^{n} k_i p_i \\ &= np_0 + \sum_{i=1}^{n} i p_i \end{aligned}$$

What happens to the equation if we assume all p_i's are equal (except p_0)?

no notes

Computation

$$\begin{aligned} f(n) &= p_0 n + \sum_{i=1}^{n} ip \\ &= p_0 n + p \sum_{i=1}^{n} i \\ &= p_0 n + p \frac{n(n+1)}{2} \\ &= p_0 n + \frac{1-p_0}{n} \frac{n(n+1)}{2} \\ &= \frac{n+1+p_0(n-1)}{2} \end{aligned}$$

Depending on the value of p_0, $\frac{n+1}{2} \leq f(n) \leq n$.

Computation

$$\begin{aligned} f(n) &= p_0 n + \sum_{i=1}^{n} ip \\ &= p_0 n + p \sum_{i=1}^{n} i \\ &= p_0 n + p \frac{n(n+1)}{2} \\ &= p_0 n + \frac{1-p_0}{n} \frac{n(n+1)}{2} \\ &= \frac{n+1+p_0(n-1)}{2} \end{aligned}$$

Depending on the value of p_0, $\frac{n+1}{2} \leq f(n) \leq n$.

$$p = \frac{1-p_0}{n}$$

.
Show a graph of p_0 vs. cost for $0 \leq p_0 \leq 1$, with y axis going from 0 to n.

Problems with Average Cost

- Average cost is usually harder to determine than worst cost.
- We really need also to know the variance around the average.
- Our computation is only as good as our knowledge (guess) on distribution.

Problems with Average Cost

- Average cost is usually harder to determine than worst cost.
- We really need also to know the variance around the average.
- Our computation is only as good as our knowledge (guess) on distribution.

Example: Quicksort variance is rather low. For this linear search, the variances is higher (normal curve).

Sorted List

Change the model: Assume that the elements are in ascending order.

Is linear search still optimal? Why not?

Optimization: Use linear search, but test if the element is greater than X. Why?

Observation: If we look at $L[5]$ and find that X is bigger, then we rule out $L[1]$ to $L[4]$ as well.

More is Better: If we look at $L[n]$ and find that X is bigger, then we know in one test that X is not in L. Great!

- What is wrong here?

Sorted List

Change the model: Assume that the elements are in ascending order.

Is linear search still optimal? Why not?

Optimization: Use linear search, but test if the element is greater than X. Why?

Observation: If we look at $L[5]$ and find that X is bigger, then we rule out $L[1]$ to $L[4]$ as well.

More is Better: If we look at $L[n]$ and find that X is bigger, then we know in one test that X is not in L. Great!

- What is wrong here?

We have more information a priori.

Can quit early.
What is best, worst, average cost? 1, n, $n/2$, respectively.
Effectively eliminates case of x not on list.

If we find that x is smaller, we only rule out one element.
Cost is 1 either way, but we don't get much information in worst case.
Small probability for big information, but big probability for small information.

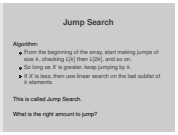
Jump Search

Algorithm:

- From the beginning of the array, start making jumps of size k, checking $L[k]$ then $L[2k]$, and so on.
- So long as X is greater, keep jumping by k.
- If X is less, then use linear search on the last sublist of k elements.

This is called Jump Search.

What is the right amount to jump?



no notes

Analysis of Jump Search

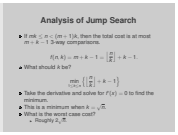
- If $mk \leq n < (m+1)k$, then the total cost is at most $m + k - 1$ 3-way comparisons.

$$f(n,k) = m + k - 1 = \left\lfloor \frac{n}{k} \right\rfloor + k - 1.$$

- What should k be?

$$\min_{1 \leq k \leq n} \left\{ \left\lfloor \frac{n}{k} \right\rfloor + k - 1 \right\}$$

- Take the derivative and solve for $f'(x) = 0$ to find the minimum.
- This is a minimum when $k = \sqrt{n}$.
- What is the worst case cost?
  - Roughly $2\sqrt{n}$.



m is number of big steps, k is size of big step.

Lessons

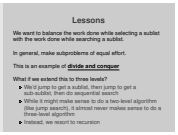
We want to balance the work done while selecting a sublist with the work done while searching a sublist.

In general, make subproblems of equal effort.

This is an example of **divide and conquer**

What if we extend this to three levels?

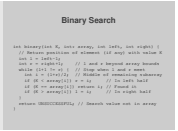
- We'd jump to get a sublist, then jump to get a sub-sublist, then do sequential search
- While it might make sense to do a two-level algorithm (like jump search), it almost never makes sense to do a three-level algorithm
- Instead, we resort to recursion



This could lead us to binary search. It could also lead us to interpolation search.

Binary Search

```
int binary(int K, int* array, int left, int right) {
  // Return position of element (if any) with value K
  int l = left-1;
  int r = right+1;    // l and r beyond array bounds
  while (l+1 != r) {  // Stop when l and r meet
    int i = (l+r)/2;  // Middle of remaining subarray
    if (K < array[i]) r = i;     // In left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```

$$f(n) = \begin{cases} 1 & n = 1 \\ f(\lfloor n/2 \rfloor) + 1 & n > 1 \end{cases}$$

Lower Bound (for Problem Worst Case)

How does n compare to $\sqrt{n}$ compare to $\log n$?

Can we do better?

Model an algorithm for the problem using a decision tree.

- Consider only comparisons with X.
- Branch depending on the result of comparing X with $L[i]$.
- There must be at least n leaf nodes in the tree. (Why?)
- Some path must be at least $\log n$ deep. (Why?)

Thus, binary search has optimal worst cost under this model.

Lower Bound (for Problem Worst Case)
How does n compare to $\sqrt{n}$ compare to $\log n$?
Can we do better?
Model an algorithm for the problem using a decision tree.
- Consider only comparisons with X.
- Branch depending on the result of comparing X with $L[i]$.
- There must be at least n leaf nodes in the tree. (Why?)
- Some path must be at least $\log n$ deep. (Why?)

Thus, binary search has optimal worst cost under this model.

Assumption: A deterministic algorithm: For a given input, the algorithm always does the same comparisons.

Since L is sorted, we already know the outcome of any comparisons between elements in L, so such comparisons are useless.

There must be some point in the algorithm, for each position in the array, where only that position remains as the possible outcome. Each such place corresponds to a (leaf) node.

Because a tree of n nodes requires at least this depth.

Average Cost of Binary Search (1)

An estimate given these assumptions:

- X is in L.
- X is equally likely to be in any position.
- $n = 2^k$ for some non-negative integer k.

Cost?

- One chance to hit in one probe.
- Two chances to hit in two probes.
- 2^{i-1} to hit in i probes.
- $i \leq k$.

Average cost is $\log n - 1$.

Average Cost of Binary Search (1)
An estimate given these assumptions:
- X is in L.
- X is equally likely to be in any position.
- $n = 2^k$ for some non-negative integer k.

Cost?
- One chance to hit in one probe.
- Two chances to hit in two probes.
- 2^{i-1} to hit in i probes.
- $i \leq k$.

Average cost is $\log n - 1$.

no notes

Average Cost Lower Bound

- Use decision trees again.
- **Total Path Length**: Sum of the level for each node.
- The cost of an outcome is the level of the corresponding node plus 1.
- The average cost of the algorithm is the average cost of the outcomes (total path length/n).
- What is the tree with the least average depth?
- This is equivalent to the tree that corresponds to binary search.
- Thus, binary search is optimal.

Average Cost Lower Bound
- Use decision trees again.
- **Total Path Length**: Sum of the level for each node.
- The cost of an outcome is the level of the corresponding node plus 1.
- The average cost of the algorithm is the average cost of the outcomes (total path length/n).
- What is the tree with the least average depth?
- This is equivalent to the tree that corresponds to binary search.
- Thus, binary search is optimal.

(In worst case.)

Fill in tree row by row, left to right. So node i is at depth $\lfloor \log i \rfloor$.

Interpolation Search

(Also known as Dictionary Search) Search L at a position

that is appropriate to the value of X.

$$p = \frac{X - L[1]}{L[n] - L[1]}$$

Repeat as necessary to recalculate p for future searches.

Interpolation Search
(Also known as Dictionary Search) Search L at a position
that is appropriate to the value of X.
$$p = \frac{X - L[1]}{L[n] - L[1]}$$
Repeat as necessary to recalculate p for future searches.

That is, readjust for new array bounds.

Note that p is a fraction, so $\lfloor pn \rfloor$ is an index position between 0 and $n - 1$.

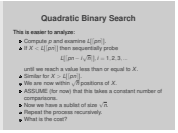
Quadratic Binary Search

This is easier to analyze:

- Compute p and examine $L[\lceil pn \rceil]$.
- If $X < L[\lceil pn \rceil]$ then sequentially probe

$$L[\lceil pn - i\sqrt{n} \rceil], i = 1, 2, 3, ...$$

  until we reach a value less than or equal to X.
- Similar for $X > L[\lceil pn \rceil]$.
- We are now within $\sqrt{n}$ positions of X.
- ASSUME (for now) that this takes a constant number of comparisons.
- Now we have a sublist of size $\sqrt{n}$.
- Repeat the process recursively.
- What is the cost?



This is following the induction in a different way than Binary Search. Binary Search says break down list by (repeatedly) splitting in half. Interpolation search says break down list by (repeatedly) finding a square root-sized sublist.

We will come back and examine this assumption.

How many times can we take the square root of n?
Keep dividing the exponent by 2 until we reach 1 – that is, take the log of the *exponent*.
What is the exponent? It is $\log n$.
$\log \log n$ is the number of times that we can take the square root.

QBS Probe Count (1)

Cost is $\Theta(\log \log n)$ IF the number of probes on jump search is constant.

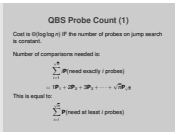
Number of comparisons needed is:

$$\sum_{i=1}^{\sqrt{n}} i\mathbf{P}(\text{need exactly } i \text{ probes})$$

$$= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}}$$

This is equal to:

$$\sum_{i=1}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

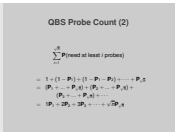


no notes

QBS Probe Count (2)

$$\sum_{i=1}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

$$\begin{aligned} &= 1 + (1 - \mathbf{P}_1) + (1 - \mathbf{P}_1 - \mathbf{P}_2) + \cdots + \mathbf{P}_{\sqrt{n}} \\ &= (\mathbf{P}_1 + ... + \mathbf{P}_{\sqrt{n}}) + (\mathbf{P}_2 + ... + \mathbf{P}_{\sqrt{n}}) + \\ &\quad (\mathbf{P}_3 + ... + \mathbf{P}_{\sqrt{n}}) + \cdots \\ &= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}} \end{aligned}$$



no notes

QBS Probe Count (3)

We require at least two probes to set the bounds, so cost is:

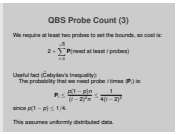
$$2 + \sum_{i=3}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

Useful fact (Čebyšev's Inequality):
The probability that we need probe i times ($\mathbf{P}_i$) is:

$$\mathbf{P}_i \leq \frac{p(1-p)n}{(i-2)^2 n} \leq \frac{1}{4(i-2)^2}$$

since $p(1-p) \leq 1/4$.

This assumes uniformly distributed data.



Original C's Inequality $\leq$ the result of recognizing that $p(1-p) \leq 1/4$.

Important assumption!

QBS Probe Count (4)

Final result:

$$2 + \sum_{i=3}^{\sqrt{n}} \frac{1}{4(i-2)^2} \approx 2.4112$$

Is this better than binary search?

What happened to our proof that binary search is optimal?

QBS Probe Count (4)

Final result:

$2 + \sum_{i=3}^{\sqrt{n}} \frac{1}{4(i-2)^2} \approx 2.4112$

Is this better than binary search?

What happened to our proof that binary search is optimal?

The assumption of uniform distribution (resulting in constant number of probes on average) is much stronger than the assumptions used by the lower bounds proof.

Comparison (1)

Let's compare $\log\log n$ to $\log n$.

n	$\log n$	$\log\log n$	Diff
16	4	2	2
256	8	3	2.7
64K	16	4	4
2^{32}	32	5	6.4

Now look at the actual comparisons used.

- Binary search $\approx \log n - 1$
- Interpolation search $\approx 2.4 \log\log n$

n	$\log n - 1$	$2.4 \log\log n$	Diff
16	3	4.8	worse
256	7	7.2	$\approx$ same
64K	15	9.6	1.6
2^{32}	31	12	2.6

Comparison (1)

Let's compare $\log\log n$ to $\log n$.

n	$\log n$	$\log\log n$	Diff
16	4	2	2
256	8	3	2.7
64K	16	4	4
2^{32}	32	5	6.4

Now look at the actual comparisons used.

- Binary search $\approx \log n - 1$
- Interpolation search $\approx 2.4 \log\log n$

n	$\log n - 1$	$2.4 \log\log n$	Diff
16	3	4.8	worse
256	7	7.2	$\approx$ same
64K	15	9.6	1.6
2^{32}	31	12	2.6

no notes

Comparison (2)

Not done yet! This is only a count of comparisons!

- Which is more expensive: calculating the midpoint or calculating the interpolation point?

Which algorithm is dependent on good behavior by the input?

Comparison (2)

Not done yet! This is only a count of comparisons!

- Which is more expensive: calculating the midpoint or calculating the interpolation point?

Which algorithm is dependent on good behavior by the input?

Taking an interpolation point.

QBS

Order Statistics

Definition: Given a sequence $S = x_1, x_2, \cdots, x_n$ of elements, x_i has **rank** k in S if x_i is the kth smallest element in S.

- Easy to find for a sorted list.
- What if list is not sorted?
- **Problem**: Find the maximum element.
- **Change the model**: Count exact number of comparisons
- **Solution**:

Order Statistics

Definition: Given a sequence $S = x_1, x_2, \cdots, x_n$ of elements, x_i has **rank** k in S if x_i is the kth smallest element in S.

- Easy to find for a sorted list.
- What if list is not sorted?
- **Problem**: Find the maximum element.
- **Change the model**: Count exact number of comparisons
- **Solution**:

Finding max: Compare element n to the maximum of the previous $n-1$ elements. Cost: $n-1$ comparisons. This is optimal since you must look at every element to be sure that it is not the maximum.

Two problems

- Find the max and the min
- Find (max and) the second biggest value

Is one of these harder than the other?

Two problems
- Find the max and the min
- Find (max and) the second biggest value

Is one of these harder than the other?

Of course both can be done in $\Theta(n)$ time, but we want to count exact number of comparisons.

Both can also be done by finding max, then finding min or second max. So both can be done in 2n-1 comparisons.

Finding the Second Best

In a single-elimination tournament, is the second best the one who loses in the finals?

Simple algorithm:
- Find the best.
- Discard it.
- Now, find the second best of the $n-1$ remaining elements.

Cost? Is this optimal?

Finding the Second Best
In a single-elimination tournament, is the second best the one who loses in the finals?
Simple algorithm:
- Find the best.
- Discard it.
- Now, find the second best of the $n-1$ remaining elements.

Cost? Is this optimal?

As we discuss this problem, we consider *exact* counts, not asymptotics.

Not necessarily – the best 2 could compete in the first round! Note that we ignore variations in performance, the outcome between two players will always be the same.

$2n-3$.

To know, need a lower bound on the problem.
Naive: $\approx n$ might work. Clearly not optimal here! But, tighten lower bound.

Lower Bound for Second (1)

Lower bound:
- Anyone who lost to anyone who is not the max cannot be second.
- So, the only candidates are those who lost to max.
- `Find_max` might compare max to $n-1$ others.
- Thus, we might need $n-2$ additional comparisons to find second.
- Wrong!

Lower Bound for Second (1)
Lower bound:
- Anyone who lost to anyone who is not the max cannot be second.
- So, the only candidates are those who lost to max.
- `Find_max` might compare max to $n-1$ others.
- Thus, we might need $n-2$ additional comparisons to find second.
- Wrong!

What is wrong with this argument?
It relies on the behavior of a particular algorithm.

Lower Bound for Second (2)

The previous argument exhibits the **necessity fallacy**:
- Our algorithm does something, therefore all algorithms solving the problem must do the same.

Alternative: Divide and conquer
- Break the list into two halves.
- Run `Find_max` on each half.
- Compare the winners.
- Run `Find_max` on the winner's half for second.
- Compare that second to second winner.

Cost: $\lceil 3n/2 \rceil - 2$.
Is this optimal?
What if we break the list into four pieces? Eight?

Lower Bound for Second (2)
The previous argument exhibits the **necessity fallacy**:
- Our algorithm does something, therefore all algorithms solving the problem must do the same.

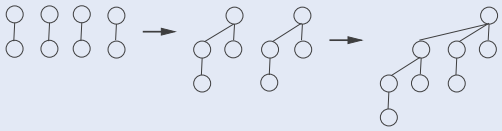
Alternative: Divide and conquer
- Break the list into two halves.
- Run `Find_max` on each half.
- Compare the winners.
- Run `Find_max` on the winner's half for second.
- Compare that second to second winner.

Cost: $\lceil 3n/2 \rceil - 2$.
Is this optimal?
What if we break the list into four pieces? Eight?

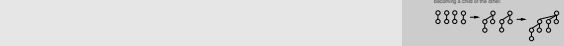
In particular, it is not necessary that the max element compare with $n-1$ others, even in the worst case.
$\lfloor n/2 \rfloor - 1 + \lceil n/2 \rceil - 1 \ldots + 1 = n-1$.
Worst case: $\lceil n/2 \rceil - 1$ elements, since winner need not compete again.
$+1$.
Cost of $\lceil 3n/2 \rceil - 2$ just closed half of the gap between our old lower bound and our old algorithm – pretty good progress!
4: about $5/4$.
8: $n-1+\lceil n/8 \rceil - 1 = \lceil 9n/8 \rceil - 2$.
What if we do this recursively?
$f(n) = 2f(n/2) + 2; f(1) = 0$ which is $3n/2 - 2$, which is no better than halves. So recursive divide & conquer (in a naive way) does not work! Quarters would be better!

Binomial Trees (1)

- Pushing this idea to its extreme, we want each comparison to be between winners of equal numbers of comparisons.
- The only candidates for second are losers to the eventual winner.
- A **binomial tree** of height m has 2^m nodes organized as:
  - a single node, if $m = 0$, or
  - two height $m-1$ binomial trees with one tree's root becoming a child of the other.



Binomial Trees (1)
- Pushing this idea to its extreme, we want each comparison to be between winners of equal numbers of comparisons.
- The only candidates for second are losers to the eventual winner.
- A binomial tree of height m has 2^m nodes organized as:
  - a single node, if $m = 0$, or
  - two height $m-1$ binomial trees with one tree's root becoming a child of the other.



but, we want as *few* of these as possible.

Binomial Trees (2)

Algorithm:

- Build the tree.
- Compare the $\lceil \log n \rceil$ children of the root for second.

Cost?

Binomial Trees (2)

Algorithm:
- Build the tree.
- Compare the $\lceil \log n \rceil$ children of the root for second.

Cost?

$n + \lceil \log n \rceil - 2$.

Adversarial Lower Bounds Proof (1)

Many lower bounds proofs use the concept of an **adversary**.

The adversary's job is to make an algorithm's cost as high as possible.

The algorithm asks the adversary for information about the input.

The adversary may never lie.

Adversarial Lower Bounds Proof (1)

Many lower bounds proofs use the concept of an adversary.

The adversary's job is to make an algorithm's cost as high as possible.

The algorithm asks the adversary for information about the input.

The adversary may never lie.

no notes

Adversarial Lower Bounds Proof (2)

Imagine that the adversary keeps a list of all possible inputs.

- When the algorithm asks a question, the adversary answers, and crosses out all remaining inputs inconsistent with that answer.
- The adversary is permitted to give any answer that is consistent with at least one remaining input.

Examples:

- Hangman.
- Search an unordered list.

Adversarial Lower Bounds Proof (2)

Imagine that the adversary keeps a list of all possible inputs.
- When the algorithm asks a question, the adversary answers, and crosses out all remaining inputs inconsistent with that answer.
- The adversary is permitted to give any answer that is consistent with at least one remaining input.

Examples:
- Hangman.
- Search an unordered list.

Adversary maintains dictionary, and can give any answer that conforms with at least one entry in the dictionary.

Adversary always says "not found" until last element.

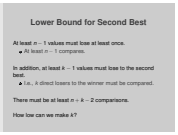
Lower Bound for Second Best

At least $n-1$ values must lose at least once.
- At least $n-1$ compares.

In addition, at least $k-1$ values must lose to the second best.
- I.e., k direct losers to the winner must be compared.

There must be at least $n+k-2$ comparisons.

How low can we make k?



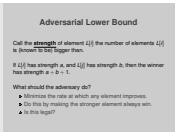
What does your intuition tell you as a lower bound for k? $\Omega(n)$? $\Omega(\log n)$? $\Omega(c)$?

Adversarial Lower Bound

Call the **strength** of element $L[i]$ the number of elements $L[i]$ is (known to be) bigger than.

If $L[i]$ has strength a, and $L[j]$ has strength b, then the winner has strength $a+b+1$.

What should the adversary do?
- Minimize the rate at which any element improves.
- Do this by making the stronger element always win.
- Is this legal?



The winner has now proved stronger than $a+b+$ the one who just lost.

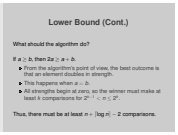
Yes. The adversary cannot "fix" the fight to give contradictory answers. But, it *can* give answers consistent with *some* legal input.

Lower Bound (Cont.)

What should the algorithm do?

If $a \geq b$, then $2a \geq a+b$.
- From the algorithm's point of view, the best outcome is that an element doubles in strength.
- This happens when $a = b$.
- All strengths begin at zero, so the winner must make at least k comparisons for $2^{k-1} < n \leq 2^k$.

Thus, there must be at least $n + \lceil \log n \rceil - 2$ comparisons.



Need to get the final strength up to $n-1$.
These k losers are candidates for 2nd place.

Min and Max

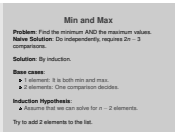
Problem: Find the minimum AND the maximum values.
Naive Solution: Do independently, requires $2n-3$ comparisons.

Solution: By induction.

Base cases:
- 1 element: It is both min and max.
- 2 elements: One comparison decides.

Induction Hypothesis:
- Assume that we can solve for $n-2$ elements.

Try to add 2 elements to the list.



We are adding items n and $n-1$.

Conceptually: ? compares for $n-2$ elements, plus one compare for last two items, plus cost to join the partial solutions.

Min and Max (2)

Induction Hypothesis:
- Assume that we can solve for $n-2$ elements.

Try to add 2 elements to the list.
- Find min and max of elements $n-1$ and n (1 compare).
- Combine these two with $n-2$ elements (2 compares).
- Total incremental work was 3 compares for 2 elements.

Total Work:

What happens if we extend this to its logical conclusion?

Min and Max (2)
Induction Hypothesis:
Assume that we can solve for $n-2$ elements.
Try to add 2 elements to the list.
Find min and max of elements $n-1$ and n (1 compare).
Combine these two with $n-2$ elements (2 compares).
Total incremental work was 3 compares for 2 elements.
Total Work:
What happens if we extend this to its logical conclusion?

Total work is about $3n/2$ comparisons.

It doesn't get any better if we split the sequence into two halves. The recurrence is:

$$T(n) = \begin{cases} 1 & n = 2 \\ 2T(n/2) + 2 & n > 2 \end{cases}$$

This is $3/2n - 2$ for n a power of 2.

The Lower Bound (1)

Is $\lceil 3n/2 \rceil - 2$ optimal?

Consider all states that a successful algorithm must go through: The **state space** lower bound.

At any given instant, track the following four categories:
- Novices: not tested.
- Winners: Won at least once, never lost.
- Losers: Lost at least once, never won.
- Moderates: Both won and lost at least once.

The Lower Bound (1)
Is $\lceil 3n/2 \rceil - 2$ optimal?
Consider all states that a successful algorithm must go through: The **state space** lower bound.
At any given instant, track the following four categories:
Novices: not tested.
Winners: Won at least once, never lost.
Losers: Lost at least once, never won.
Moderates: Both won and lost at least once.

no notes

The Lower Bound (2)

Who can get ignored?

What is the initial state?

What is the final state?

How is this relevant?

The Lower Bound (2)
Who can get ignored?
What is the initial state?
What is the final state?
How is this relevant?

Moderates – Can't be min or max.

Initial: (n, 0, 0, 0).

Final: (0, 1, 1, n-2).

We must go from the initial state to the final state to solve the problem.
So, we can analyze how this gets done.

Lower Bound (3)

Every algorithm must go from $(n, 0, 0, 0)$ to $(0, 1, 1, n-2)$.

There are 10 types of comparison.

Comparing with a moderate cannot be more efficient than other comparisons, so ignore them.

Lower Bound (3)
Every algorithm must go from $(n, 0, 0, 0)$ to $(0, 1, 1, n-2)$.
There are 10 types of comparison.
Comparing with a moderate cannot be more efficient than other comparisons, so ignore them.

That gets rid of 4 types of comparisons.

Lower Bound (3)

If we are in state (i, j, k, l) and we have a comparison, then:

$N:N$	$(i-2,$	$j+1,$	$k+1,$	$l)$
$W:W$	$(i,$	$j-1,$	$k,$	$l+1)$
$L:L$	$(i,$	$j,$	$k-1,$	$l+1)$
$L:N$	$(i-1,$	$j+1,$	$k,$	$l)$
or	$(i-1,$	$j,$	$k,$	$l+1)$
$W:N$	$(i-1,$	$j,$	$k+1,$	$l)$
or	$(i-1,$	$j,$	$k,$	$l+1)$
$W:L$	$(i,$	$j,$	$k,$	$l)$
or	$(i,$	$j-1,$	$k-1,$	$l+2)$

Lower Bound (3)

If we are in state (i, j, k, l) and we have a comparison, then:

$N:N$	$(i-2,$	$j+1,$	$k+1,$	$l)$
$W:W$	$(i,$	$j-1,$	$k,$	$l+1)$
$L:L$	$(i,$	$j,$	$k-1,$	$l+1)$
$L:N$	$(i-1,$	$j+1,$	$k,$	$l)$
or	$(i-1,$	$j,$	$k,$	$l+1)$
$W:N$	$(i-1,$	$j,$	$k+1,$	$l)$
or	$(i-1,$	$j,$	$k,$	$l+1)$
$W:L$	$(i,$	$j,$	$k,$	$l)$
or	$(i,$	$j-1,$	$k-1,$	$l+2)$

no notes

Adversarial Argument

What should an adversary do?

- Comparing a winner to a loser is of no value.

Only the following five transitions are of interest:

$N:N$	$(i-2,$	$j+1,$	$k+1,$	$l)$
$L:N$	$(i-1,$	$j+1,$	$k,$	$l)$
$W:N$	$(i-1,$	$j,$	$k+1,$	$l)$
$W:W$	$(i,$	$j-1,$	$k,$	$l+1)$
$L:L$	$(i,$	$j,$	$k-1,$	$l+1)$

Only the last two types increase the number of moderates, so there must be $n-2$ of these.

The number of novices must go to 0, and the first is the most efficient way to do this: $\lceil n/2 \rceil$ are required.

Adversarial Argument

What should an adversary do?

- Comparing a winner to a loser is of no value.

Only the following five transitions are of interest:

$N:N$	$(i-2,$	$j+1,$	$k+1,$	$l)$
$L:N$	$(i-1,$	$j+1,$	$k,$	$l)$
$W:N$	$(i-1,$	$j,$	$k+1,$	$l)$
$W:W$	$(i,$	$j-1,$	$k,$	$l+1)$
$L:L$	$(i,$	$j,$	$k-1,$	$l+1)$

Only the last two types increase the number of moderates, so there must be $n-2$ of these.

The number of novices must go to 0, and the first is the most efficient way to do this: $\lceil n/2 \rceil$ are required.

Minimize information gained.

Adversary will just make the winner win – No new information is provided.

This provides an algorithm.

Kth Smallest Element

Problem: Find the kth smallest element from sequence S.

(Also called **selection**.)

Solution: Find min value and discard (k times).

- If k is large, find $n-k$ max values.

Cost: $O(\min(k, n-k)n)$ – only better than sorting if k is $\mathrm{O}(\log n)$ or $O(n - \log n)$.

Kth Smallest Element

Problem: Find the kth smallest element from sequence S.

(Also called **selection**.)

Solution: Find min value and discard (k times).

- If k is large, find $n-k$ max values.

Cost: $O(\min(k, n-k)n)$ – only better than sorting if k is $\mathrm{O}(\log n)$ or $O(n - \log n)$.

no notes

Better Kth Smallest Algorithm

Use quicksort, but take only one branch each time.

Average case analysis:

$$f(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(f(i-1))$$

Average case cost: $\mathrm{O}(n)$ time.

Better Kth Smallest Algorithm

Use quicksort, but take only one branch each time.

Average case analysis:

$$f(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(f(i-1))$$

Average case cost: $\mathrm{O}(n)$ time.

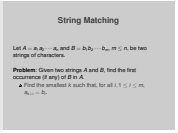
Like Quicksort, it is possible for this to take $O(n^2)$ time!!
It is possible to guarentee average case $O(n)$ time.

String Matching

Let $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$, $m \leq n$, be two strings of characters.

Problem: Given two strings A and B, find the first occurrence (if any) of B in A.

- Find the smallest k such that, for all $i, 1 \leq i \leq m$, $a_{k+i} = b_i$.



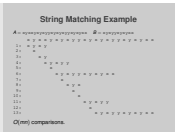
no notes

String Matching Example

$A =$ `xyxxyxyxyyxyxyxyyxyxyxx` $B =$ `xyxyyxyxyxx`

```
     x y x x y x y x y y x y x y x y y x y x y x x
 1:  x y x y
 2:    x
 3:      x y
 4:        x y x y y
 5:          x
 6:            x y x y y x y x y x x
 7:              x
 8:                x y x
 9:                  x
10:                    x
11:                      x y x y y
12:                        x
13:                          x y x y y x y x y x x
```

$O(mn)$ comparisons.

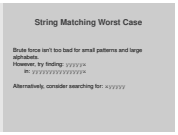


$O(mn)$ comparisons in worst case.

String Matching Worst Case

Brute force isn't too bad for small patterns and large alphabets.
However, try finding: `yyyyyx`
in: `yyyyyyyyyyyyyyyx`

Alternatively, consider searching for: `xyyyyy`



Our example was a little pessimistic... but it wasn't worst case!

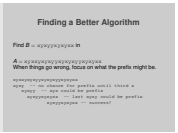
In the second example, we can quickly reject a position - no backtracking.

Finding a Better Algorithm

Find $B =$ `xyxyyxyxyxx` in

$A =$ `xyxxyxyxyyxyxyxyyxyxyxx`
When things go wrong, focus on what the prefix might be.

```
xyxxyxyxyyxyxyxyyxyxyxx
xyxy  -- no chance for prefix until third x
   xyxyy  -- xyx could be prefix
     xyxyyxyxyxx  -- last xyxy could be prefix
            xyxyyxyxyxx -- success!
```

Not only can we skip down several letters if we track the potential prefix, we don't need even to repeat the check of the prefix letters – just start that many characters down.

Knuth-Morris-Pratt Algorithm

- Key to success:
  - Preprocess B to create a table of information on how far to slide B when a mismatch is encountered.
- Notation: $B(i)$ is the first i characters of B.
- For each character:
  - We need the **maximum suffix** of $B(i)$ that is equal to a prefix of B.
- $next(i) =$ the maximum j ($0 < j < i - 1$) such that $b_{i-j}b_{i-j+1} \cdots b_{i-1} = B(j)$, and 0 if no such j exists.
- We define $next(1) = -1$ to distinguish it.
- $next(2) = 0$. Why?

Knuth-Morris-Pratt Algorithm
- Key to success:
  - Preprocess B to create a table of information on how far to slide B when a mismatch is encountered.
- Notation: $B(i)$ is the first i characters of B.
- For each character:
  - We need the maximum suffix of $B(i)$ that is equal to a prefix of B.
- $next(i) =$ the maximum j ($0 < j < i - 1$) such that $b_{i-j}b_{i-j+1} \cdots b_{i-1} = B(j)$, and 0 if no such j exists.
- We define $next(1) = -1$ to distinguish it.
- $next(2) = 0$. Why?

In all cases other than $B[1]$ we compare current A value to appropriate B value. The test told us there was no match at that position. If $B[1]$ does not match a character of A, that character is completely rejected. We must slide B over it.

Why? All that we know is that the 2nd letter failed to match. There is no value j such that $0 < j < i - 1$. Conceptually, compare beginning of B to current character.

Computing the table

$B =$

1	2	3	4	5	6	7	8	9	10	11
x	y	x	y	y	x	y	x	y	x	x
−1	0	0	1	2	0	1	2	3	4	3

- The third line is the "next" table.
- At each position ask "If I fail here, how many letters before me are good?"

Computing the table
$B =$

1	2	3	4	5	6	7	8	9	10	11
x	y	x	y	y	x	y	x	y	x	x
−1	0	0	1	2	0	1	2	3	4	3

- The third line is the "next" table.
- At each position ask "If I fail here, how many letters before me are good?"

no notes

How to Compute Table?

- By induction.
- **Base cases**: $next(1)$ and $next(2)$ already determined.
- **Induction Hypothesis**: Values have been computed up to $next(i-1)$.
- **Induction Step**: For $next(i)$: at most $next(i-1)+1$.
  - When? $b_{i-1} = b_{next(i-1)+1}$.
  - That is, largest suffix can be extended by b_{i-1}.
- If $b_{i-1} \neq b_{next(i-1)+1}$, then need new suffix.
- But, this is just a mismatch, so use *next* table to compute where to check.

How to Compute Table?
- By induction.
- **Base cases**: $next(1)$ and $next(2)$ already determined.
- **Induction Hypothesis**: Values have been computed up to $next(i-1)$.
- **Induction Step**: For $next(i)$: at most $next(i-1)+1$.
  - When? $b_{i-1} = b_{next(i-1)+1}$.
  - That is, largest suffix can be extended by b_{i-1}.
- If $b_{i-1} \neq b_{next(i-1)+1}$, then need new suffix.
- But, this is just a mismatch, so use *next* table to compute where to check.

Induction step: Each step can only improve by 1.

While this is complex to understand, it is efficient to implement.

Complexity of KMP Algorithm

- A character of A may be compared against many characters of B.
  - For every mismatch, we have to look at another position in the table.
- How many backtracks are possible?
- If mismatch at b_k, then only k mismatches are possible.
- But, for each mismatch, we had to go forward a character to get to b_k.
- Since there are always n forward moves, the total cost is $O(n)$.

Complexity of KMP Algorithm
- A character of A may be compared against many characters of B.
  - For every mismatch, we have to look at another position in the table.
- How many backtracks are possible?
- If mismatch at b_k, then only k mismatches are possible.
- But, for each mismatch, we had to go forward a character to get to b_k.
- Since there are always n forward moves, the total cost is $O(n)$.

no note

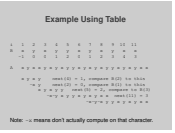
Example Using Table

```
i   1   2   3   4   5   6   7   8   9  10  11
B   x   y   x   y   y   x   y   x   y   x   x
   -1   0   0   1   2   0   1   2   3   4   3

A   x y x x y x y x y y x y x y x y y x y x y x x

    x y x y    next(4) = 1, compare B(2) to this
      -x y     next(2) = 0, compare B(1) to this
         x y x y y   next(5) = 2, compare to B(3)
            -x-y x y y x y x y x x  next(11) = 3
                        -x-y-x y y x y x y x x
```

Note: $-x$ means don't actually compute on that character.

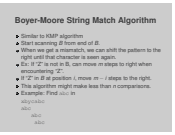


no note

Boyer-Moore String Match Algorithm

- Similar to KMP algorithm
- Start scanning B from end of B.
- When we get a mismatch, we can shift the pattern to the right until that character is seen again.
- Ex: If "Z" is not in B, can move m steps to right when encountering "Z".
- If "Z" in B at position i, move $m - i$ steps to the right.
- This algorithm might make less than n comparisons.
- Example: Find `abc` in

```
xbycabc
abc
   abc
    abc
```

Better for larger alphabets.

Probabilistic Algorithms

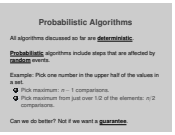
All algorithms discussed so far are **deterministic**.

Probabilistic algorithms include steps that are affected by **random** events.

Example: Pick one number in the upper half of the values in a set.

1. Pick maximum: $n - 1$ comparisons.
2. Pick maximum from just over 1/2 of the elements: $n/2$ comparisons.

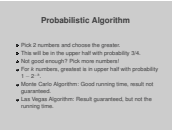
Can we do better? Not if we want a **guarantee**.



no notes

Probabilistic Algorithm

- Pick 2 numbers and choose the greater.
- This will be in the upper half with probability 3/4.
- Not good enough? Pick more numbers!
- For k numbers, greatest is in upper half with probability $1 - 2^{-k}$.
- Monte Carlo Algorithm: Good running time, result not guaranteed.
- Las Vegas Algorithm: Result guaranteed, but not the running time.



Pick k big enough and the chance for failure becomes less than the chance that the machine will crash (i.e., probability of even getting an answer from a deterministic algorithm).

Rather have no answer than a wrong answer? If k is big enough, the probability of a wrong answer is less than any calamity with finite probability – with this probability independent of n.

Searching Linked Lists

Assume the list is sorted, but is stored in a linked list.

Can we use binary search?

- Comparisons?
- "Work?"

What if we add additional pointers?

Searching Linked Lists

Assume the list is sorted, but is stored in a linked list.

Can we use binary search?
- Comparisons?
- "Work?"

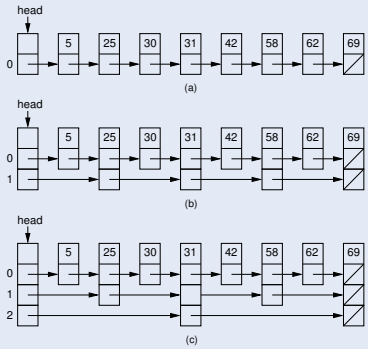
What if we add additional pointers?

Same. Is this a good model? No.

Much higher since we must move around a lot (without comparisons) to get to the same position.

Might get to desired position faster.

"Perfect" Skip List

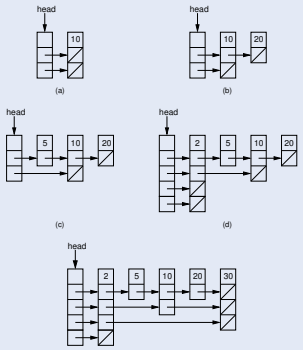


"Perfect" Skip List

What is the access time? log n.
We can insert/delete in log n time as well.

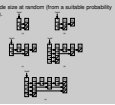
Building a Skip List

Pick the node size at random (from a suitable probability distribution).



Building a Skip List

Pick the node size at random (from a suitable probability distribution).



no notes

Skip List Analysis (1)

What distribution do we want for the node depths?

```
int randomLevel(void) { // Exponential distrib
  for (int level=0; Random(2) == 0; level++);
  return level;
}
```

What is the worst cost to search in the "perfect" Skip List?

What is the average cost to search in the "perfect" Skip List?

What is the cost to insert?

What is the average cost in the "typical" Skip List?

Skip List Analysis (1)

What distribution do we want for the node depths?

```
int randomLevel(void) { // Exponential distrib
  for (int level=0; Random(2) == 0; level++);
  return level;
}
```

What is the worst cost to search in the "perfect" Skip List?

What is the average cost to search in the "perfect" Skip List?

What is the cost to insert?

What is the average cost in the "typical" Skip List?

Exponential decay. 1 link half of the time, 2 links one quarter, 3 links one eighth, and so on.

log n.

Close to log n.

log n.

log n.

Skip List Analysis (2)

How does this differ from a BST?

- Simpler or more complex?
- More or less efficient?
- Which relies on data distribution, which on basic laws of probability?

Skip List Analysis (2)

How does this differ from a BST?
- Simpler or more complex?
- More or less efficient?
- Which relies on data distribution, which on basic laws of probability?

About the same.

On average, about the same *if* data are well distributed.

BST relies on data distribution, while skiplist merely relies on chance.

Probabilistic Quicksort

Quicksort runs into trouble on highly structured input.

Solution: Randomize input order.

- Chance of worst case is then $2/n!$.

Probabilistic Quicksort

Quicksort runs into trouble on highly structured input.

Solution: Randomize input order.
- Chance of worst case is then $2/n!$.

This principle is why, for example, the Skip List data structure has much more reliable performance than a BST. The BST's performance depends on the input data. The Skip List's performance depends entirely on chance. For random data, the two are essentially identical. But you can't trust data to be random.

Random Number Generators

- Most computers systems use a deterministic algorithm to select **pseudorandom** numbers.
- **Linear congruential method**:
  - Pick a **seed** $r(1)$. Then,

$$r(i) = (r(i-1) \times b) \bmod t.$$

- Must pick good values for b and t.
- Resulting numbers must be in the range:
- What happens if $r(i) = r(j)$?

Random Number Generators

- Most computers systems use a deterministic algorithm to select **pseudorandom** numbers.
- **Linear congruential method**:
  - Pick a **seed** $r(1)$. Then,

$$r(i) = (r(i-1) \times b) \bmod t.$$

- Must pick good values for b and t.
- Resulting numbers must be in the range:
- What happens if $r(i) = r(j)$?

Lots of "commercial" random number generators have poor performance because they don't get the numbers right. Must be in range 0 to $t - 1$.

They generate the same number, which leads to a cycle of length $|j - i|$.

Random Number Generators (cont)

Some examples:

$$\begin{aligned} r(i) &= 6r(i-1) \bmod 13 = \\ &\quad \cdots 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1 \cdots \\ r(i) &= 7r(i-1) \bmod 13 = \\ &\quad \cdots 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1 \cdots \\ r(i) &= 5r(i-1) \bmod 13 = \\ &\quad \cdots 1, 5, 12, 8, 1 \cdots \\ &\quad \cdots 2, 10, 11, 3, 2 \cdots \\ &\quad \cdots 4, 7, 9, 6, 4 \cdots \\ &\quad \cdots 0, 0 \cdots \end{aligned}$$

The last one depends on the start value of the seed.
Suggested generator: $r(i) = 16807r(i-1) \bmod 2^{31} - 1$

Random Number Generators (cont)

Some examples:

$$\begin{aligned} r(i) &= 6r(i-1) \bmod 13 = \\ &\quad \cdots 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1 \cdots \\ r(i) &= 7r(i-1) \bmod 13 = \\ &\quad \cdots 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1 \cdots \\ r(i) &= 5r(i-1) \bmod 13 = \\ &\quad \cdots 1, 5, 12, 8, 1 \cdots \\ &\quad \cdots 2, 10, 11, 3, 2 \cdots \\ &\quad \cdots 4, 7, 9, 6, 4 \cdots \\ &\quad \cdots 0, 0 \cdots \end{aligned}$$

The last one depends on the start value of the seed.
Suggested generator: $r(i) = 16807r(i-1) \bmod 2^{31} - 1$

no notes

Graph Algorithms

Graphs are useful for representing a variety of concepts:

- Data Structures
- Relationships
- Families
- Communication Networks
- Road Maps

Graph Algorithms

Graphs are useful for representing a variety of concepts:
- Data Structures
- Relationships
- Families
- Communication Networks
- Road Maps

- A **graph** $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ consists of a set of **vertices** V, and a set of **edges** E, such that each edge in E is a connection between a pair of vertices in V.
- Directed vs. Undirected
- Labeled graph, weighted graph
- Labels for edges vs. weights for edges
- Multiple edges, loops
- Cycle, Circuit, path, simple path, tours
- Bipartite, acyclic, connected
- Rooted tree, unrooted tree, free tree

A Tree Proof

- **Definition**: A **free tree** is a connected, undirected graph that has no cycles.
- **Theorem**: If T is a free tree having n vertices, then T has exactly $n - 1$ edges.
- **Proof**: By induction on n.
- **Base Case**: $n = 1$. T consists of 1 vertex and 0 edges.
- **Inductive Hypothesis**: The theorem is true for a tree having $n - 1$ vertices.
- **Inductive Step**:
  - If T has n vertices, then T contains a vertex of degree 1.
  - Remove that vertex and its incident edge to obtain T', a free tree with $n - 1$ vertices.
  - By IH, T' has $n - 2$ edges.
  - Thus, T has $n - 1$ edges.

A Tree Proof

- **Definition:** A free tree is a connected, undirected graph that has no cycles.
- **Theorem:** If T is a free tree having n vertices, then T has exactly $n - 1$ edges.
- **Proof:** By induction on n.
- **Base Case:** $n = 1$. T consists of 1 vertex and 0 edges.
- **Inductive Hypothesis:** The theorem is true for a tree having $n - 1$ vertices.
- **Inductive Step:**
  - If T has n vertices, then T contains a vertex of degree 1.
  - Remove that vertex and its incident edge to obtain T', a free tree with $n - 1$ vertices.
  - By IH, T' has $n - 2$ edges.
  - Thus, T has $n - 1$ edges.

This is close to a satisfactory definition for free tree. There are several equivalent definitions for free trees, with similar proofs to relate them.

Why do we know that some vertex has degree 1? Because the definition says that the Free Tree has no cycles.

Graph Traversals

Various problems require a way to **traverse** a graph – that is, visit each vertex and edge in a systematic way.

Three common traversals:

1. Eulerian tours
   Traverse each edge exactly once
2. Depth-first search
   Keeps vertices on a stack
3. Breadth-first search
   Keeps vertices on a queue

Graph Traversals

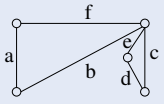
Various problems require a way to **traverse** a graph – that is, visit each vertex and edge in a systematic way.

Three common traversals:
1. Eulerian tours
   Traverse each edge exactly once
2. Depth-first search
   Keeps vertices on a stack
3. Breadth-first search
   Keeps vertices on a queue

a vertex may be visited multiple times

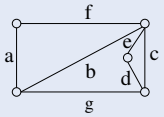
Eulerian Tours

A circuit that contains every edge exactly once.
Example:



Tour: b a f c d e.

Example:



No Eulerian tour. How can you tell for sure?

Eulerian Tours

A circuit that contains every edge exactly once.
Example:
Tour: b a f c d e
Example:
No Eulerian tour. How can you tell for sure?

Why no tour? Because some vertices have odd degree.

All even nodes is a necessary condition. Is it sufficient?

Eulerian Tour Proof

- **Theorem**: A connected, undirected graph with m edges that has no vertices of odd degree has an Eulerian tour.
- **Proof**: By induction on m.
- **Base Case**:
- **Inductive Hypothesis**:
- **Inductive Step**:
  - Start with an arbitrary vertex and follow a path until you return to the vertex.
  - Remove this circuit. What remains are connected components G_1, G_2, ..., G_k each with nodes of even degree and $< m$ edges.
  - By IH, each connected component has an Eulerian tour.
  - Combine the tours to get a tour of the entire graph.

Base case: 0 edges and 1 vertex fits the theorem.
IH: The theorem is true for $< m$ edges.
Always possible to find a circuit starting at any arbitrary vertex, since each vertex has even degree.

Depth First Search

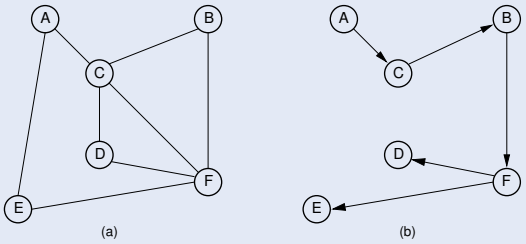
```
void DFS(Graph G, int v) { // Depth first search
  PreVisit(G, v);     // Take appropriate action
  G.setMark(v, VISITED);
  for (Edge w = each neighbor of v)
    if (G.getMark(G.v2(w)) == UNVISITED)
      DFS(G, G.v2(w));
  PostVisit(G, v);    // Take appropriate action
}
```

Initial call: `DFS(G, r)` where r is the **root** of the DFS.

Cost: $\Theta(|\mathrm{V}| + |\mathrm{E}|)$.

no notes

Depth First Search Example



The directions are imposed by the traversal. This is the Depth First Search Tree.

DFS Tree

If we number the vertices in the order that they are marked, we get **DFS numbers**.

Lemma 7.2: Every edge $e \in E$ is either in the DFS tree T, or connects two vertices of G, one of which is an ancestor of the other in T.

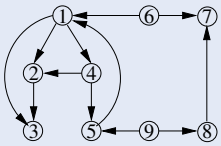
Proof: Consider the first time an edge (v, w) is examined, with v the current vertex.

- If w is unmarked, then (v, w) is in T.
- If w is marked, then w has a smaller DFS number than v AND (v, w) is an unexamined edge of w.
- Thus, w is still on the stack. That is, w is on a path from v.

Results: No “cross edges.” That is, no edges connecting vertices sideways in the tree.

DFS for Directed Graphs

- Main problem: A connected graph may not give a single DFS tree.



- Forward edges: (1, 3)
- Back edges: (5, 1)
- Cross edges: (6, 1), (8, 7), (9, 5), (9, 8), (4, 2)
- **Solution**: Maintain a list of unmarked vertices.
  - Whenever one DFS tree is complete, choose an arbitrary unmarked vertex as the root for a new tree.

DFS for Directed Graphs
- Main problem: A connected graph may not give a single DFS tree.
- Forward edges: (1, 3)
- Back edges: (5, 1)
- Cross edges: (6, 1), (8, 7), (9, 5), (9, 8), (4, 2)
- Solution: Maintain a list of unmarked vertices.
  - Whenever one DFS tree is complete, choose an arbitrary unmarked vertex as the root for a new tree.

no notes

Directed Cycles

Lemma 7.4: Let G be a directed graph. G has a directed cycle iff every DFS of G produces a back edge.

Proof:

1. Suppose a DFS produces a back edge (v, w).
   - v and w are in the same DFS tree, w an ancestor of v.
   - (v, w) and the path in the tree from w to v form a directed cycle.
2. Suppose G has a directed cycle C.
   - Do a DFS on G.
   - Let w be the vertex of C with smallest DFS number.
   - Let (v, w) be the edge of C coming into w.
   - v is a descendant of w in a DFS tree.
   - Therefore, (v, w) is a back edge.

Directed Cycles
Lemma 7.4: Let G be a directed graph. G has a directed cycle iff every DFS of G produces a back edge.
Proof:
1. Suppose a DFS produces a back edge (v, w).
   - v and w are in the same DFS tree, w an ancestor of v.
   - (v, w) and the path in the tree from w to v form a directed cycle.
2. Suppose G has a directed cycle C.
   - Do a DFS on G.
   - Let w be the vertex of C with smallest DFS number.
   - Let (v, w) be the edge of C coming into w.
   - v is a descendant of w in a DFS tree.
   - Therefore, (v, w) is a back edge.

See earlier lemma.

Breadth First Search

- Like DFS, but replace stack with a queue.
- Visit vertex's neighbors before going deeper in tree.

Breadth First Search
- Like DFS, but replace stack with a queue.
- Visit vertex's neighbors before going deeper in tree.

no notes

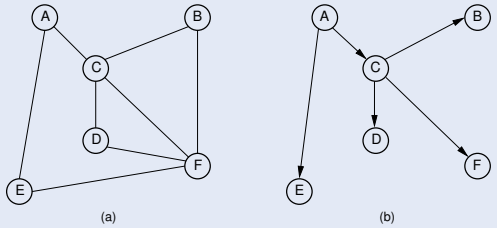
Breadth First Search Algorithm

```
void BFS(Graph G, int start) {
  Queue Q(G.n());
  Q.enqueue(start);
  G.setMark(start, VISITED);
  while (!Q.isEmpty()) {
    int v = Q.dequeue();
    PreVisit(G, v);   // Take appropriate action
    for (Edge w = each neighbor of v)
      if (G.getMark(G.v2(w)) == UNVISITED) {
        G.setMark(G.v2(w), VISITED);
        Q.enqueue(G.v2(w));
      }
    PostVisit(G, v);  // Take appropriate action
}}
```

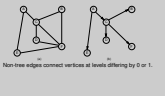

Breadth First Search Algorithm
void BFS(Graph G, int start) {
Queue Q(G.n());
Q.enqueue(start);
G.setMark(start, VISITED);
while (!Q.isEmpty()) {
int v = Q.dequeue();
PreVisit(G, v); // Take appropriate action
for (Edge w = each neighbor of v)
if (G.getMark(G.v2(w)) == UNVISITED) {
G.setMark(G.v2(w), VISITED);
Q.enqueue(G.v2(w));
}
PostVisit(G, v); // Take appropriate action
}}

no notes

Breadth First Search Example



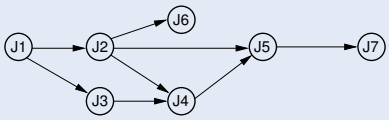
Non-tree edges connect vertices at levels differing by 0 or 1.

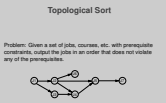


We know this because if an edge had connected to a deeper level, then that target node would have been placed on the queue when the edge was encountered.

Topological Sort

Problem: Given a set of jobs, courses, etc. with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites.

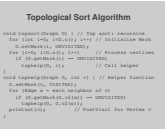




no notes

Topological Sort Algorithm

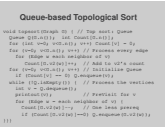
```
void topsort(Graph G) { // Top sort: recursive
  for (int i=0; i<G.n(); i++) // Initialize Mark
    G.setMark(i, UNVISITED);
  for (i=0; i<G.n(); i++)     // Process vertices
    if (G.getMark(i) == UNVISITED)
      tophelp(G, i);          // Call helper
}
void tophelp(Graph G, int v) { // Helper function
  G.setMark(v, VISITED);
  for (Edge w = each neighbor of v)
    if (G.getMark(G.v2(w)) == UNVISITED)
      tophelp(G, G.v2(w));
  printout(v);          // PostVisit for Vertex v
}
```

Prints in reverse order.

Queue-based Topological Sort

```
void topsort(Graph G) { // Top sort: Queue
  Queue Q(G.n());  int Count[G.n()];
  for (int v=0; v<G.n(); v++) Count[v] = 0;
  for (v=0; v<G.n(); v++) // Process every edge
    for (Edge w each neighbor of v)
      Count[G.v2(w)]++;  // Add to v2's count
  for (v=0; v<G.n(); v++) // Initialize Queue
    if (Count[v] == 0) Q.enqueue(v);
  while (!Q.isEmpty()) {  // Process the vertices
    int v = Q.dequeue();
    printout(v);          // PreVisit for v
    for (Edge w = each neighbor of v) {
      Count[G.v2(w)]--;   // One less prereq
      if (Count[G.v2(w)]==0) Q.enqueue(G.v2(w));
}}}
```

no notes

Shortest Paths Problems

Input: A graph with **weights** or **costs** associated with each edge.

Output: The list of edges forming the shortest path.

Sample problems:

- Find the shortest path between two specified vertices.
- Find the shortest path from vertex S to all other vertices.
- Find the shortest path between all pairs of vertices.

Our algorithms will actually calculate only **distances**.

Shortest Paths Problems
Input: A graph with weights or costs associated with each edge.
Output: The list of edges forming the shortest path.
Sample problems:
Find the shortest path between two specified vertices.
Find the shortest path from vertex S to all other vertices.
Find the shortest path between all pairs of vertices.
Our algorithms will actually calculate only distances.

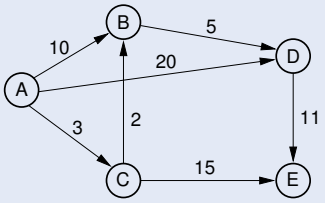
no notes

Shortest Paths Definitions

d(A, B) is the **shortest distance** from vertex A to B.

w(A, B) is the **weight** of the edge connecting A to B.

- If there is no such edge, then w(A, B) $= \infty$.



Shortest Paths Definitions
d(A, B) is the shortest distance from vertex A to B.
w(A, B) is the weight of the edge connecting A to B.
If there is no such edge, then w(A, B) = ∞.

w(A, D) $= 20$; d(A, D) $= 10$ (through ACBD).

Single Source Shortest Paths

Given start vertex s, find the shortest path from s to all other vertices.

Try 1: Visit all vertices in some order, compute shortest paths for all vertices seen so far, then add the shortest path to next vertex x.

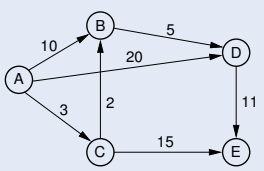
Problem: Shortest path to a vertex already processed might go through x.
Solution: Process vertices in order of distance from s.

Single Source Shortest Paths
Given start vertex s, find the shortest path from s to all other vertices.
Try 1: Visit all vertices in some order, compute shortest paths for all vertices seen so far, then add the shortest path to next vertex x.
Problem: Shortest path to a vertex already processed might go through x.
Solution: Process vertices in order of distance from s.

no notes

Dijkstra's Algorithm Example

	A	B	C	D	E
Initial	0	∞	∞	∞	∞
Process A	0	10	3	20	∞
Process C	0	5	3	20	18
Process B	0	5	3	10	18
Process D	0	5	3	10	18
Process E	0	5	3	10	18



Dijkstra's Algorithm Example

no notes

Dijkstra's Algorithm: Array (1)

```
void Dijkstra(Graph G, int s) { // Use array
  int D[G.n()];
  for (int i=0; i<G.n(); i++)   // Initialize
    D[i] = INFINITY;
  D[s] = 0;
  for (i=0; i<G.n(); i++) {  // Process vertices
    int v = minVertex(G, D);
    if (D[v] == INFINITY) return; // Unreachable
    G.setMark(v, VISITED);
    for (Edge w = each neighbor of v)
      if (D[G.v2(w)] > (D[v] + G.weight(w)))
        D[G.v2(w)] = D[v] + G.weight(w);
  }
}
```


Dijkstra's Algorithm: Array (1)

no notes

Dijkstra's Algorithm: Array (2)

```
// Get mincost vertex
int minVertex(Graph G, int* D) {
  int v;  // Initialize v to an unvisited vertex;
  for (int i=0; i<G.n(); i++)
    if (G.getMark(i) == UNVISITED)
      { v = i; break; }
  for (i++; i<G.n(); i++) // Find smallest D val
    if ((G.getMark(i)==UNVISITED) && (D[i]<D[v]))
      v = i;
  return v;
}
```

Approach 1: Scan the table on each pass for closest vertex.
Total cost: $\Theta(|\mathrm{V}|^2 + |\mathrm{E}|) = \Theta(|\mathrm{V}|^2)$.

Dijkstra's Algorithm: Array (2)

no notes

Dijkstra's Algorithm: Priority Queue (1)

```
class Elem { public: int vertex, dist; };
int key(Elem x) { return x.dist; }
void Dijkstra(Graph G, int s) { // priority queue
  int v;   Elem temp;
  int D[G.n()];  Elem E[G.e()];
  temp.dist = 0; temp.vertex = s; E[0] = temp;
  heap H(E, 1, G.e());          // Create the heap
  for (int i=0; i<G.n(); i++) D[i] = INFINITY;
  D[s] = 0;
  for (i=0; i<G.n(); i++) {     // Get distances
    do { temp = H.removemin(); v = temp.vertex; }
      while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (D[v] == INFINITY) return; // Unreachable
```


Dijkstra's Algorithm: Priority Queue (1)

no notes

Dijkstra's Algorithm: Priority Queue (2)

```
    for (Edge w = each neighbor of v)
      if (D[G.v2(w)] > (D[v] + G.weight(w))) {
        D[G.v2(w)] = D[v] + G.weight(w);
        temp.dist = D[G.v2(w)];
        temp.vertex = G.v2(w);
        H.insert(temp);  // Insert new distance
}}}
```

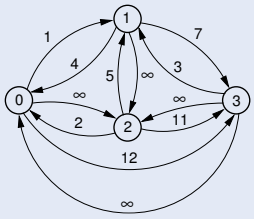
- Approach 2: Store unprocessed vertices using a min-heap to implement a priority queue ordered by D value. Must update priority queue for each edge.
- Total cost: $\Theta((|\mathrm{V}| + |\mathrm{E}|) \log |\mathrm{V}|)$.

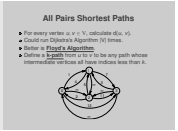
Dijkstra's Algorithm: Priority Queue (2)

no notes

All Pairs Shortest Paths

- For every vertex $u, v \in V$, calculate d(u, v).
- Could run Dijkstra's Algorithm |V| times.
- Better is **Floyd's Algorithm**.
- Define a **k-path** from u to v to be any path whose intermediate vertices all have indices less than k.





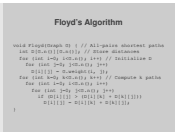
Multiple runs of Dijkstra's algorithm Cost:
$|V||E| \log |V| = |V|^3 \log |V|$ for dense graph.

The issue driving the concept of "k paths" is how to efficiently check all the paths without computing any path more than once.

0,3 is a 0-path. 2,0,3 is a 1-path. 0,2,3 is a 3-path, but not a 2 or 1 path. Everything is a 4 path.

Floyd's Algorithm

```
void Floyd(Graph G) { // All-pairs shortest paths
  int D[G.n()][G.n()]; // Store distances
  for (int i=0; i<G.n(); i++) // Initialize D
    for (int j=0; j<G.n(); j++)
      D[i][j] = G.weight(i, j);
  for (int k=0; k<G.n(); k++) // Compute k paths
    for (int i=0; i<G.n(); i++)
      for (int j=0; j<G.n(); j++)
        if (D[i][j] > (D[i][k] + D[k][j]))
          D[i][j] = D[i][k] + D[k][j];
}
```

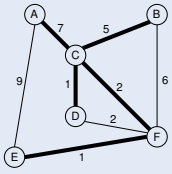



no notes

Minimum Cost Spanning Trees

Minimum Cost Spanning Tree (MST) Problem:

- Input: An undirected, connected graph G.
- Output: The subgraph of G that
  1. has minimum total cost as measured by summing the values for all of the edges in the subset, and
  2. keeps the vertices connected.



no notes

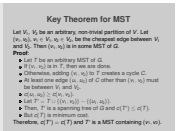
Key Theorem for MST

Let V_1, V_2 be an arbitrary, non-trivial partition of V. Let (v_1, v_2), $v_1 \in V_1, v_2 \in V_2$, be the cheapest edge between V_1 and V_2. Then (v_1, v_2) is in some MST of G.

Proof:

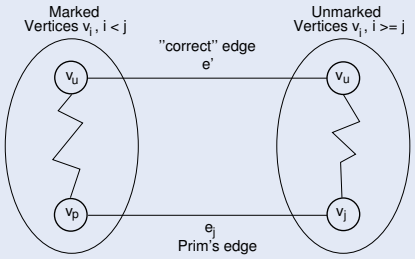
- Let T be an arbitrary MST of G.
- If (v_1, v_2) is in T, then we are done.
- Otherwise, adding (v_1, v_2) to T creates a cycle C.
- At least one edge (u_1, u_2) of C other than (v_1, v_2) must be between V_1 and V_2.
- $c(u_1, u_2) \geq c(v_1, v_2)$.
- Let $T' = T \cup \{(v_1, v_2)\} - \{(u_1, u_2)\}$.
- Then, T' is a spanning tree of G and $c(T') \leq c(T)$.
- But $c(T)$ is minimum cost.

Therefore, $c(T') = c(T)$ and T' is a MST containing (v_1, v_2).

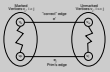


There can only be multiple MSTs when there are edges with equal cost.

Key Theorem Figure



2014-05-02 CS 5114 — Key Theorem Figure

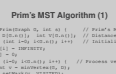


no notes

Prim's MST Algorithm (1)

```
void Prim(Graph G, int s) {     // Prim's MST alg
  int D[G.n()];  int V[G.n()];  // Distances
  for (int i=0; i<G.n(); i++)   // Initialize
    D[i] = INFINITY;
  D[s] = 0;
  for (i=0; i<G.n(); i++) {   // Process vertices
    int v = minVertex(G, D);
    G.setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v);
    if (D[v] == INFINITY) return; //v unreachable
    for (Edge w = each neighbor of v)
      if (D[G.v2(w)] > G.weight(w)) {
        D[G.v2(w)] = G.weight(w); // Update dist
        V[G.v2(w)] = v;          // who came from
}}}
```


2014-05-02 CS 5114 — Prim's MST Algorithm (1)



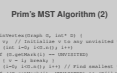
no notes

Prim's MST Algorithm (2)

```
int minVertex(Graph G, int* D) {
  int v;  // Initialize v to any unvisited vertex
  for (int i=0; i<G.n(); i++)
    if (G.getMark(i) == UNVISITED)
      { v = i; break; }
  for (i=0; i<G.n(); i++) // Find smallest value
    if ((G.getMark(i)==UNVISITED) && (D[i]<D[v]))
      v = i;
  return v;
}
```

This is an example of a **greedy** algorithm.

2014-05-02 CS 5114 — Prim's MST Algorithm (2)



no notes

Alternative Prim's Implementation (1)

Like Dijkstra's algorithm, can implement with priority queue.

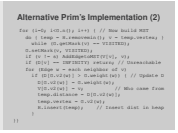
```
void Prim(Graph G, int s) {
  int v;                  // The current vertex
  int D[G.n()];           // Distance array
  int V[G.n()];           // Who's closest
  Elem temp;
  Elem E[G.e()];          // Heap array
  temp.distance = 0; temp.vertex = s;
  E[0] = temp;            // Initialize heap array
  heap H(E, 1, G.e());    // Create the heap
  for (int i=0; i<G.n(); i++) D[i] = INFINITY;
  D[s] = 0;
```


2014-05-02 CS 5114 — Alternative Prim's Implementation (1)

no notes

Alternative Prim's Implementation (2)

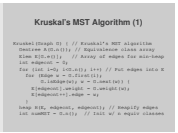
```
  for (i=0; i<G.n(); i++) { // Now build MST
    do { temp = H.removemin(); v = temp.vertex; }
      while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v);
    if (D[v] == INFINITY) return; // Unreachable
    for (Edge w = each neighbor of v)
      if (D[G.v2(w)] > G.weight(w)) { // Update D
        D[G.v2(w)] = G.weight(w);
        V[G.v2(w)] = v;          // Who came from
        temp.distance = D[G.v2(w)];
        temp.vertex = G.v2(w);
        H.insert(temp);    // Insert dist in heap
      }
}}
```

no notes

Kruskal's MST Algorithm (1)

```
Kruskel(Graph G) { // Kruskal's MST algorithm
  Gentree A(G.n()); // Equivalence class array
  Elem E[G.e()];   // Array of edges for min-heap
  int edgecnt = 0;
  for (int i=0; i<G.n(); i++) // Put edges into E
    for (Edge w = G.first(i);
         G.isEdge(w); w = G.next(w)) {
      E[edgecnt].weight = G.weight(w);
      E[edgecnt++].edge = w;
    }
  heap H(E, edgecnt, edgecnt); // Heapify edges
  int numMST = G.n();  // Init w/ n equiv classes
```

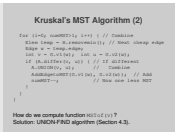



no notes

Kruskal's MST Algorithm (2)

```
  for (i=0; numMST>1; i++) { // Combine
    Elem temp = H.removemin(); // Next cheap edge
    Edge w = temp.edge;
    int v = G.v1(w);  int u = G.v2(w);
    if (A.differ(v, u)) { // If different
      A.UNION(v, u);      //   Combine
      AddEdgetoMST(G.v1(w), G.v2(w));  // Add
      numMST--;           // Now one less MST
    }
  }
}
```

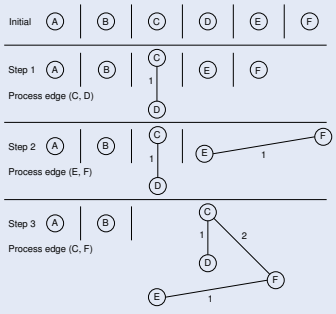
How do we compute function `MSTof(v)`?
Solution: UNION-FIND algorithm (Section 4.3).

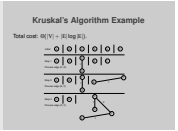


no notes

Kruskal's Algorithm Example

Total cost: $\Theta(|\mathrm{V}| + |\mathrm{E}| \log |\mathrm{E}|)$.





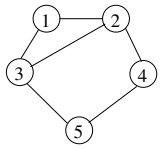
Cost is dominated by the edge sort.
Alternative: Use a min heap, quit when only one set left.
"Kth-smallest" implementation.

Matching

- Suppose there are n workers that we want to work in teams of two. Only certain pairs of workers are willing to work together.
- **Problem**: Form as many compatible non-overlapping teams as possible.
- Model using G, an undirected graph.
  - Join vertices if the workers will work together.
- A **matching** is a set of edges in G with no vertex in more than one edge (the edges are independent).
  - A **maximal matching** has no free pairs of vertices that can extend the matching.
  - A **maximum matching** has the greatest possible number of edges.
  - A **perfect matching** includes every vertex.

Matching
- Suppose there are n workers that we want to work in teams of two. Only certain pairs of workers are willing to work together.
- Problem: Form as many compatible non-overlapping teams as possible.
- Model using G, an undirected graph.
  - Join vertices if the workers will work together.
- A matching is a set of edges in G with no vertex in more than one edge (the edges are independent).
  - A maximal matching has no free pairs of vertices that can extend the matching.
  - A maximum matching has the greatest possible number of edges.
  - A perfect matching includes every vertex.

An example:
(1-3) is a matching.
(1-3) (5, 4) is both maximal and maximum.
Take away the edge (5-4). Then (3, 2) would be maximal but not a maximum matching.



Very Dense Graphs (1)

Theorem: Let $G = (V, E)$ be an undirected graph with $|V| = 2n$ and every vertex having degree $\geq n$. Then G contains a perfect matching.

Proof: Suppose that G does not contain a perfect matching.

- Let $M \subseteq E$ be a max matching. $|M| < n$.
- There must be two unmatched vertices v_1, v_2 that are not adjacent.
- Every vertex adjacent to v_1 or to v_2 is matched.
- Let $M' \subseteq M$ be the set of edges involved in matching the neighbors of v_1 and v_2.
- There are $\geq 2n$ edges from v_1 and v_2 to vertices covered by M', but $|M'| < n$.

Very Dense Graphs (1)
Theorem: Let $G = (V, E)$ be an undirected graph with $|V| = 2n$ and every vertex having degree $\geq n$. Then G contains a perfect matching.
Proof: Suppose that G does not contain a perfect matching.
- Let $M \subseteq E$ be a max matching. $|M| < n$.
- There must be two unmatched vertices v_1, v_2 that are not adjacent.
- Every vertex adjacent to v_1 or to v_2 is matched.
- Let $M' \subseteq M$ be the set of edges involved in matching the neighbors of v_1 and v_2.
- There are $\geq 2n$ edges from v_1 and v_2 to vertices covered by M', but $|M'| < n$.

There must be two unmatched vertices not adjacent:
Otherwise it would either be perfect (if there are no 2 free vertices) or we could just match v_1 and v_2 (because they are adjacent).

Every adjacent vertex is matched, otherwise the matching would not be maximal.

See Manber Figure 3.76.

Very Dense Graphs (2)

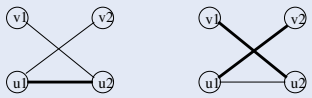
Proof: (continued)

- Thus, some edge of M' is adjacent to 3 edges from v_1 and v_2.
- Let (u_1, u_2) be such an edge.
- Replacing (u_1, u_2) with (v_1, u_2) and (v_2, u_1) results in a larger matching.
- Theorem proven by contradiction.

Very Dense Graphs (2)
Proof: (continued)
- Thus, some edge of M' is adjacent to 3 edges from v_1 and v_2.
- Let (u_1, u_2) be such an edge.
- Replacing (u_1, u_2) with (v_1, u_2) and (v_2, u_1) results in a larger matching.
- Theorem proven by contradiction.

Pigeonhole Principle

Generalizing the Insight

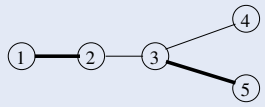


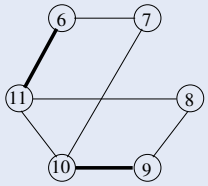
- v_1, u_2, u_1, v_2 is a path from an unmatched vertex to an unmatched vertex such that alternate edges are unmatched and matched.
- In one step, switch unmatched and matched edges.
- Let $G = (V, E)$ be an undirected graph and $M \subseteq E$ a matching.
- An **alternating path** P goes from v to u, consists of alternately matched and unmatched edges, and both v and u are not in the match.

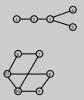
Generalizing the Insight
- v_1, u_2, u_1, v_2 is a path from an unmatched vertex to an unmatched vertex such that alternate edges are unmatched and matched.
- In one step, switch unmatched and matched edges.
- Let $G = (V, E)$ be an undirected graph and $M \subseteq E$ a matching.
- An alternating path P goes from v to u, consists of alternately matched and unmatched edges, and both v and u are not in the match.

no notes

Matching Example







1, 2, 3, 5 is NOT an alternating path (it does not start with an unmatch vertex).

7, 6, 11, 10, 9, 8 is an alternating path with respect to the given matching.

Observation: If a matching has an alternating path, then the size of the matching can be increased by one by switching matched and unmatched edges along the alternating path.

The Alternating Path Theorem (1)

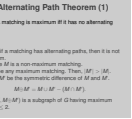
Theorem: A matching is maximum iff it has no alternating paths.

Proof:

- Clearly, if a matching has alternating paths, then it is not maximum.
- Suppose M is a non-maximum matching.
- Let M' be any maximum matching. Then, $|M'| > |M|$.
- Let $M \oplus M'$ be the symmetric difference of M and M'.

$$M \oplus M' = M \cup M' - (M \cap M').$$

- $G' = (V, M \oplus M')$ is a subgraph of G having maximum degree ≤ 2.



The first point is the obvious part of the iff. If there is an alternating path, simply switch the match and umatched edges to augment the match.

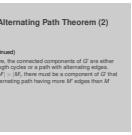
Symmetric difference: Those in either, but not both.

The max degree is ≤ 2 because a vertex matches one different vertex in M and M'.

The Alternating Path Theorem (2)

Proof: (continued)

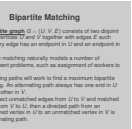
- Therefore, the connected components of G' are either even-length cycles or a path with alternating edges.
- Since $|M'| > |M|$, there must be a component of G' that is an alternating path having more M' edges than M edges.



no notes

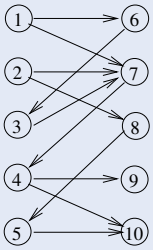
Bipartite Matching

- A **bipartite graph** $G = (U, V, E)$ consists of two disjoint sets of vertices U and V together with edges E such that every edge has an endpoint in U and an endpoint in V.
- Bipartite matching naturally models a number of assignment problems, such as assignment of workers to jobs.
- Alternating paths will work to find a maximum bipartite matching. An alternating path always has one end in U and the other in V.
- If we direct unmatched edges from U to V and matched edges from V to U, then a directed path from an unmatched vertex in U to an unmatched vertex in V is an alternating path.



no notes

Bipartite Matching Example



2, 8, 5, 10 is an alternating path.

1, 6, 3, 7, 4, 9 and 2, 8, 5, 10 are **disjoint** alternating paths that we can augment **independently**.

Bipartite Matching Example
2, 8, 5, 10 is an alternating path.
1, 6, 3, 7, 4, 9 and 2, 8, 5, 10 are disjoint alternating paths that we can augment independently

Naive algorithm: Find a maximal matching (greedy algorithm).

For each vertex:
Do a DFS or other search until an alternating path is found.
Use the alternating path to improve the match.

$|V|(|V| + |E|)$

Algorithm for Maximum Bipartite Matching

Construct BFS subgraph from the set of unmatched vertices in U until a level with unmatched vertices in V is found.

Greedily select a maximal set of disjoint alternating paths.

Augment along each path independently.

Repeat until no alternating paths remain.

Time complexity $O((|V| + |E|)\sqrt{|V|})$.

Algorithm for Maximum Bipartite Matching
Construct BFS subgraph from the set of unmatched vertices in U until a level with unmatched vertices in V is found.
Greedily select a maximal set of disjoint alternating paths.
Augment along each path independently.
Repeat until no alternating paths remain.
Time complexity $O((|V| + |E|)\sqrt{|V|})$.

Order doesn't matter. Find a path, remove its vertices, then repeat.Augment along the paths independently since they are disjoint.

Network Flows

Models distribution of utilities in networks such as oil pipelines, water systems, etc. Also, highway traffic flow.

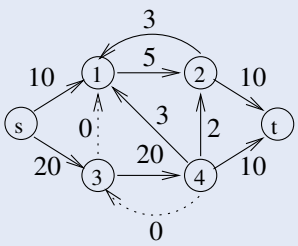
Simplest version:

A **network** is a directed graph $G = (V, E)$ having a distinguished source vertex s and a distinguished sink vertex t. Every edge (u, v) of G has a **capacity** $c(u, v) \geq 0$. If $(u, v) \notin E$, then $c(u, v) = 0$.

Network Flows
Models distribution of utilities in networks such as oil pipelines, water systems, etc. Also, highway traffic flow.
Simplest version:
A network is a directed graph $G = (V, E)$ having a distinguished source vertex s and a distinguished sink vertex t. Every edge (u, v) of G has a capacity $c(u, v) \geq 0$. If $(u, v) \notin E$, then $c(u, v) = 0$.

no notes

Network Flow Graph



Network Flow Graph

no notes

Network Flow Definitions

A **flow** in a network is a function $f : V \times V \to R$ with the following properties.

(i) **Skew Symmetry**:

$$\forall v, w \in V, \quad f(v, w) = -f(w, v).$$

(ii) **Capacity Constraint**:

$$\forall v, w, \in V, \quad f(v, w) \leq c(v, w).$$

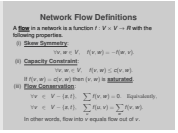
If $f(v, w) = c(v, w)$ then (v, w) is **saturated**.

(iii) **Flow Conservation**:

$$\forall v \in V - \{s, t\}, \quad \sum f(v, w) = 0. \quad \text{Equivalently,}$$

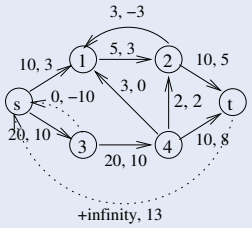
$$\forall v \in V - \{s, t\}, \quad \sum_u f(u, v) = \sum_w f(v, w).$$

In other words, flow into v equals flow out of v.



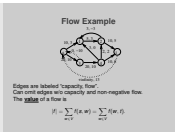
no notes

Flow Example



Edges are labeled "capacity, flow".
Can omit edges w/o capacity and non-negative flow.
The **value** of a flow is

$$|f| = \sum_{w \in V} f(s, w) = \sum_{w \in V} f(w, t).$$



3, -3 is an illustration of "negative flow" returning. Every node can be thought of as having negative flow. We will make use of this later – augmenting paths.

Max Flow Problem

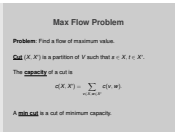
Problem: Find a flow of maximum value.

Cut (X, X') is a partition of V such that $s \in X, t \in X'$.

The **capacity** of a cut is

$$c(X, X') = \sum_{v \in X, w \in X'} c(v, w).$$

A **min cut** is a cut of minimum capacity.



no notes

Cut Flows

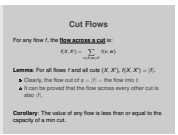
For any flow f, the **flow across a cut** is:

$$f(X, X') = \sum_{v \in X, w \in X'} f(v, w).$$

Lemma: For all flows f and all cuts (X, X'), $f(X, X') = |f|$.

- Clearly, the flow out of $s = |f| =$ the flow into t.
- It can be proved that the flow across every other cut is also $|f|$.

Corollary: The value of any flow is less than or equal to the capacity of a min cut.



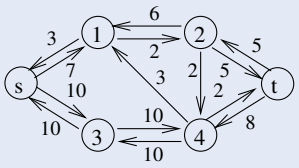
no notes

Residual Graph

Given any flow f, the **residual capacity** of the edge is

$$res(v, w) = c(v, w) - f(v, w) \geq 0.$$

Residual graph is a network $R = (V, E_R)$ where E_R contains edges of non-zero residual capacity.



2014-05-02 CS 5114 └Residual Graph

Residual Graph
Given any flow f, the residual capacity of the edge is $res(v, w) = c(v, w) - f(v, w) \geq 0$.
Residual graph is a network $R = (V, E_R)$ where E_R contains edges of non-zero residual capacity.

R is the network after f has been subtracted.
Saturated edges do not appear.
Some edges have larger capacity than in G.

Observations

1. Any flow in R can be added to F to obtain a larger flow in G.
2. In fact, a max flow f' in R plus the flow f (written $f + f'$) is a max flow in G.
3. Any path from s to t in R can carry a flow equal to the smallest capacity of any edge on it.
   - Such a path is called an **augmenting path**.
   - For example, the path

$$s, 1, 2, t$$

   can carry a flow of 2 units $= c(1, 2)$.

2014-05-02 CS 5114 └Observations

Observations
1. Any flow in R can be added to F to obtain a larger flow in G.
2. In fact, a max flow f' in R plus the flow f (written $f + f'$) is a max flow in G.
3. Any path from s to t in R can carry a flow equal to the smallest capacity of any edge on it.
   - Such a path is called an augmenting path.
   - For example, the path
$$s, 1, 2, t$$
   can carry a flow of 2 units $= c(1, 2)$.

no notes

Max-flow Min-cut Theorem

The following are equivalent:

(i) f is a max flow.

(ii) f has no augmenting path in R.

(iii) $|f| = c(X, X')$ for some min cut (X, X').

Proof:

(i) $\Rightarrow$ (ii):

- If f has an augmenting path, then f is not a max flow.

2014-05-02 CS 5114 └Max-flow Min-cut Theorem

Max-flow Min-cut Theorem
The following are equivalent:
(i) f is a max flow.
(ii) f has no augmenting path in R.
(iii) $|f| = c(X, X')$ for some min cut (X, X').
Proof:
(i) $\Rightarrow$ (ii):
- If f has an augmenting path, then f is not a max flow.

no notes

Max-flow Min-cut Theorem (2)

(ii) $\Rightarrow$ (iii):

- Suppose f has no augmenting path in R.
- Let X be the subset of V reachable from s and $X' = V - X$.
- Then $s \in X, t \in X'$, so (X, X') is a cut.
- $\forall v \in X, w \in X'$, $res(v, w) = c(v, w) - f(v, w) = 0$.
- $f(X, X') = \sum_{v \in X, w \in X'} f(v, w) = \sum_{v \in X, w \in X'} c(v, w) = c(X, X')$.
- By Lemma, $|f| = c(X, X')$ and (X, X') is a min cut.

2014-05-02 CS 5114 └Max-flow Min-cut Theorem (2)

Max-flow Min-cut Theorem (2)
(ii) $\Rightarrow$ (iii):
- Suppose f has no augmenting path in R.
- Let X be the subset of V reachable from s and $X' = V - X$.
- Then $s \in X, t \in X'$, so (X, X') is a cut.
- $\forall v \in X, w \in X'$, $res(v, w) = c(v, w) - f(v, w) = 0$.
- $f(X, X') = \sum_{v \in X, w \in X'} f(v, w) = \sum_{v \in X, w \in X'} c(v, w) = c(X, X')$.
- By Lemma, $|f| = c(X, X')$ and (X, X') is a min cut.

Line 4: Because no augmenting path.
Line 5: Because we know the residuals are all 0.

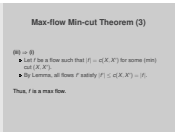
In other words, look at the capacity of G at the cut separating s from t in the residual graph. This must be a min cut (for G) with capacity $|f|$.

Max-flow Min-cut Theorem (3)

(iii) $\Rightarrow$ (i)

- Let f be a flow such that $|f| = c(X, X')$ for some (min) cut (X, X').
- By Lemma, all flows f' satisfy $|f'| \leq c(X, X') = |f|$.

Thus, f is a max flow.



no notes

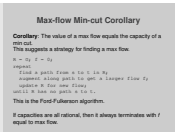
Max-flow Min-cut Corollary

Corollary: The value of a max flow equals the capacity of a min cut.
This suggests a strategy for finding a max flow.

```
R = G; f = 0;
repeat
  find a path from s to t in R;
  augment along path to get a larger flow f;
  update R for new flow;
until R has no path s to t.
```

This is the Ford-Fulkerson algorithm.

If capacities are all rational, then it always terminates with f equal to max flow.



Problem with Ford-Fulkerson:
Draw graph with nodes nodes s, t, a, and b. Flow from S to a and b is M, flow from a and b to t is M, flow from a to b is 1.

Now, pick s-a-b-t.
Then s-b-a-t. (reverse 1 unit of flow).
Repeat M times.
M is unrelated to the size of V, E, so this is potentially exponential.

Edmonds-Karp Algorithm

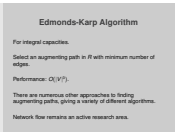
For integral capacities.

Select an augmenting path in R with minimum number of edges.

Performance: $O(|V|^3)$.

There are numerous other approaches to finding augmenting paths, giving a variety of different algorithms.

Network flow remains an active research area.



no notes

Geometric Algorithms

Potentially **large** set of objects to manipulate.

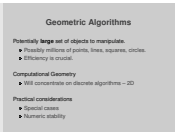
- Possibly millions of points, lines, squares, circles.
- Efficiency is crucial.

Computational Geometry

- Will concentrate on discrete algorithms – 2D

Practical considerations

- Special cases
- Numeric stability



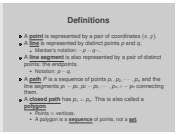
Same principles often apply to 3D, but it may be more complicated.
We will avoid continuous problems such as polygon intersection.

Special cases: Geometric programming is much like other programming in this sense. But there are a LOT of special cases! Co-point, co-linear, co-planar, horizontal, vertical, etc.

Numeric stability: Each intersection point in a cascade of intersections might require increasing precision to represent the computed intersection, even when the point coordinates start as integers. Floating point causes problems!

Definitions

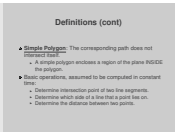
- A **point** is represented by a pair of coordinates (x, y).
- A **line** is represented by distinct points p and q.
  - Manber's notation: $-p - q-$.
- A **line segment** is also represented by a pair of distinct points: the endpoints.
  - Notation: $p - q$.
- A **path** P is a sequence of points $p_1, p_2, \cdots, p_n$ and the line segments $p_1 - p_2, p_2 - p_3, \cdots, p_{n-1} - p_n$ connecting them.
- A **closed path** has $p_1 = p_n$. This is also called a **polygon**.
  - Points $\equiv$ vertices.
  - A polygon is a **sequence** of points, not a **set**.



Line alternate representation: slope and intercept.
For polygons, order matters. A left-handed and right-handed triangle are not the same even if they occupy the same space.

Definitions (cont)

- **Simple Polygon**: The corresponding path does not intersect itself.
  - A simple polygon encloses a region of the plane INSIDE the polygon.
- Basic operations, assumed to be computed in constant time:
  - Determine intersection point of two line segments.
  - Determine which side of a line that a point lies on.
  - Determine the distance between two points.



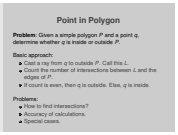
no notes

Point in Polygon

Problem: Given a simple polygon P and a point q, determine whether q is inside or outside P.

Basic approach:
- Cast a ray from q to outside P. Call this L.
- Count the number of intersections between L and the edges of P.
- If count is even, then q is outside. Else, q is inside.

Problems:
- How to find intersections?
- Accuracy of calculations.
- Special cases.



Special cases:

- Line intersects polygon at a vertex, goes in to out.
- Line intersects poly. at inflection point (stays in or stays out).
- Line intersects polygon through a line.

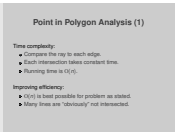
Simplify calculations by making line horizontal.

Accuracy of calculations is not a problem with integer coordinates for points and a horizontal line. But think about representing the intersection point for two arbitrary line segements (from a polygon intersection operation). Cascading intersections can lead to ever-increasing demand for precision in coordinate representation.

Point in Polygon Analysis (1)

Time complexity:
- Compare the ray to each edge.
- Each intersection takes constant time.
- Running time is $\mathrm{O}(n)$.

Improving efficiency:
- $\mathrm{O}(n)$ is best possible for problem as stated.
- Many lines are "obviously" not intersected.

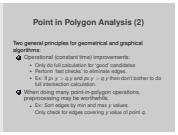


no notes

Point in Polygon Analysis (2)

Two general principles for geometrical and graphical algorithms:

1. Operational (constant time) improvements:
   - Only do full calculation for 'good' candidates
   - Perform 'fast checks' to eliminate edges.
   - Ex: If $p_1.y > q.y$ and $p_2.y > q.y$ then don't bother to do full intersection calculation.
2. When doing many point-in-polygon operations, preprocessing may be worthwhile.
   - Ex: Sort edges by min and max y values. Only check for edges covering y value of point q.



Spatial data structures can help.

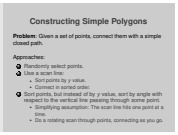
"Fast checks" take time. When they "win" (they rule something out), they save time. When they "lose" (they fail to rule something out) they add extra time. So they have to "win" often enough so that the time savings outweighs the cost of the check.

Constructing Simple Polygons

Problem: Given a set of points, connect them with a simple closed path.

Approaches:

1. Randomly select points.
2. Use a scan line:
   - Sort points by y value.
   - Connect in sorted order.
3. Sort points, but instead of by y value, sort by angle with respect to the vertical line passing through some point.
   - Simplifying assumption: The scan line hits one point at a time.
   - Do a rotating scan through points, connecting as you go.



(1) Could easily yield an intersection.

(2) The problem is connecting point p_n back to p_1. This could yield an intersection.

Simplifying assumption is that the points are not colinear w.r.t. the scan line.

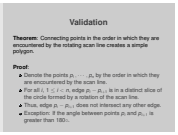
See Manber Figure 8.6.

Validation

Theorem: Connecting points in the order in which they are encountered by the rotating scan line creates a simple polygon.

Proof:

- Denote the points $p_1, \cdots, p_n$ by the order in which they are encountered by the scan line.
- For all i, $1 \leq i < n$, edge $p_i - p_{i+1}$ is in a distinct slice of the circle formed by a rotation of the scan line.
- Thus, edge $p_i - p_{i+1}$ does not intersect any other edge.
- Exception: If the angle between points p_i and p_{i+1} is greater than 180$\circ$.



So, the key is to pick a point for the center of the rotating scan that guarentees that the angle never reachese 180$\circ$.

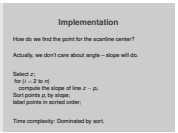
Implementation

How do we find the point for the scanline center?

Actually, we don't care about angle – slope will do.

Select z;
for ($i = 2$ to n)
  compute the slope of line $z - p_i$.
Sort points p_i by slope;
label points in sorted order;

Time complexity: Dominated by sort.



Pick as z the point with greatest x value (and least y value if there is a tie). See Manber Figure 8.7.

The next point is the next largest angle between $z - p_i$ and the vertical line through z. It is important to use the slope, because then our computation is a constant-time operation with no transendental functions.

z is the point with greatest x value (minimum y in case of tie)

So, time is $\Theta(n \log n)$

Convex Hull

- A **convex hull** is a polygon such that any line segment connecting two points inside the polygon is itself entirely inside the polygon.
- A **convex path** is a path of points $p_1, p_2, \cdots, p_n$ such that connecting p_1 and p_n results in a convex polygon.
- The convex hull for a set of points is the smallest convex polygon enclosing all the points.
  - imagine placing a tight rubberband around the points.
- The point **belongs** to the hull if it is a vertex of the hull.
- **Problem**: Compute the convex hull of n points.

Convex Hull
- A convex hull is a polygon such that any line segment connecting two points inside the polygon is itself entirely inside the polygon.
- A convex path is a path of points $p_1, p_2, \cdots, p_n$ such that connecting p_1 and p_n results in a convex polygon.
- The convex hull for a set of points is the smallest convex polygon enclosing all the points.
  - Imagine placing a tight rubberband around the points.
- The point belongs to the hull if it is a vertex of the hull.
- Problem: Compute the convex hull of n points.

no notes

Simple Convex Hull Algorithm

IH: Assume that we can compute the convex hull for $< n$ points, and try to add the nth point.

1. nth point is inside the hull.
   - No change.
2. nth point is outside the convex hull
   - "Stretch" hull to include the point (dropping other points).

Simple Convex Hull Algorithm
IH: Assume that we can compute the convex hull for $< n$ points, and try to add the nth point.
1. nth point is inside the hull.
   - No change.
2. nth point is outside the convex hull
   - "Stretch" hull to include the point (dropping other points).

See Manber Figure 8.9.

Subproblems (1)

Potential problems as we process points:
1. Determine if point is inside convex hull.
2. Stretch a hull.

The straightforward induction approach is inefficient. (Why?)

Our standard induction alternative: Select a special point for the nth point – some sort of min or max point.

If we always pick the point with max x, what problem is eliminated?
Stretch:
1. Find vertices to eliminate
2. Add new vertex between existing vertices.

Subproblems (1)
Potential problems as we process points:
1. Determine if point is inside convex hull.
2. Stretch a hull.
The straightforward induction approach is inefficient. (Why?)
Our standard induction alternative: Select a special point for the nth point – some sort of min or max point.
If we always pick the point with max x, what problem is eliminated?
Stretch:
1. Find vertices to eliminate
2. Add new vertex between existing vertices.

Why? Lots of points don't affect the hull, and stretching is expensive.

Subproblem 1 can be eliminated: the max is always outside the polygon.

Subproblems (2)

Supporting line of a convex polygon is a line intersecting the polygon at exactly one vertex.

Only two supporting lines between convex hull and max point q.

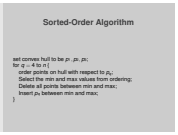
These supporting lines intersect at "min" and "max" points on the (current) convex hull.

Subproblems (2)
Supporting line of a convex polygon is a line intersecting the polygon at exactly one vertex.
Only two supporting lines between convex hull and max point q.
These supporting lines intersect at "min" and "max" points on the (current) convex hull.

"Min" and "max" with respect to the angle formed by the supporting lines.

Sorted-Order Algorithm

set convex hull to be p_1, p_2, p_3;
for $q = 4$ to n {
  order points on hull with respect to p_q;
  Select the min and max values from ordering;
  Delete all points between min and max;
  Insert p_q between min and max;
}



Sort by x value.

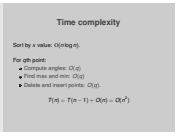
Time complexity

Sort by x value: $\mathrm{O}(n \log n)$.

For qth point:

- Compute angles: $O(q)$
- Find max and min: $O(q)$
- Delete and insert points: $O(q)$.

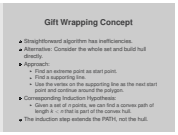
$$T(n) = T(n-1) + O(n) = O(n^2)$$



no notes

Gift Wrapping Concept

- Straightforward algorithm has inefficiencies.
- Alternative: Consider the whole set and build hull directly.
- Approach:
  - Find an extreme point as start point.
  - Find a supporting line.
  - Use the vertex on the supporting line as the next start point and continue around the polygon.
- Corresponding Induction Hypothesis:
  - Given a set of n points, we can find a convex path of length $k < n$ that is part of the convex hull.
- The induction step extends the PATH, not the hull.

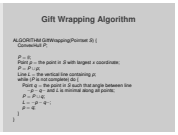


Straightforward algorithm spends time to build convex hull with points interior to final convex hull.

Gift Wrapping Algorithm

ALGORITHM GiftWrapping(Pointset S) {
  ConvexHull P;

  $P = \emptyset$;
  Point p = the point in S with largest x coordinate;
  $P = P \cup p$;
  Line L = the vertical line containing p;
  while (P is not complete) do {
    Point q = the point in S such that angle between line $-p-q-$ and L is minimal along all points;
    $P = P \cup q$;
    $L = -p-q-$;
    $p = q$;
  }
}



no notes

Gift Wrapping Analysis

Complexity:

- To add kth point, find the min angle among $n - k$ lines.
- Do this h times (for h the number of points on hull).
- Often good in average case.
- Could be bad in worst case.

Gift Wrapping Analysis

Complexity:
- To add kth point, find the min angle among $n - k$ lines.
- Do this h times (for h the number of points on hull).
- Often good in average case.
- Could be bad in worst case.

$O(n^2)$. Actually, $O(hn)$ where h is the number of edges to hull.

Graham's Scan

- Approach:
  - Start with the points ordered with respect to some maximal point.
  - Process these points in order, adding them to the set of processed points and its convex hull.
  - Like straightforward algorithm, but pick better order.
- Use the Simple Polygon algorithm to order the points by angle with respect to the point with max x value.
- Process points in this order, maintaining the convex hull of points seen so far.

Graham's Scan

- Approach:
  - Start with the points ordered with respect to some maximal point.
  - Process these points in order, adding them to the set of processed points and its convex hull.
  - Like straightforward algorithm, but pick better order.
- Use the Simple Polygon algorithm to order the points by angle with respect to the point with max x value.
- Process points in this order, maintaining the convex hull of points seen so far.

See Manber Figure 8.11.

Graham's Scan (cont)

Induction Hypothesis:

- Given a set of n points ordered according to algorithm Simple Polygon, we can find a convex path among the first $n - 1$ points corresponding to the convex hull of the $n - 1$ points.

Induction Step:

- Add the kth point to the set.
- Check the angle formed by p_k, p_{k-1}, p_{k-2}.
- If angle $< 180\circ$ with respect to inside of the polygon, then delete p_{k-1} and repeat.

Graham's Scan (cont)

Induction Hypothesis:
- Given a set of n points ordered according to algorithm Simple Polygon, we can find a convex path among the first $n - 1$ points corresponding to the convex hull of the $n - 1$ points.

Induction Step:
- Add the kth point to the set.
- Check the angle formed by p_k, p_{k-1}, p_{k-2}.
- If angle $< 180\circ$ with respect to inside of the polygon, then delete p_{k-1} and repeat.

no notes

Graham's Scan Algorithm

```
ALGORITHM GrahamsScan(Pointset P) {
  Point p1 = the point in P with largest x coordinate;
  P = SimplePolygon(P, p1); // Order points in P
  Point q1 = p1;
  Point q2 = p2;
  Point q3 = p3;
  int m = 3;
  for (k = 4 to n) {
    while (angle(−qm−1 − qm−, −qm − pk−) ≤ 180°) do
      m = m − 1;
    m = m + 1;
    qm = pk;
  }
}
```


Graham's Scan Algorithm

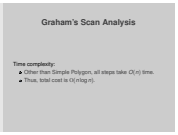
```
ALGORITHM GrahamsScan(Pointset P) {
  Point p1 = the point in P with largest x coordinate;
  P = SimplePolygon(P, p1); // Order points in P
  Point q1 = p1;
  Point q2 = p2;
  Point q3 = p3;
  int m = 3;
  for (k = 4 to n) {
    while (angle(−qm−1 − qm−, −qm − pk−) ≤ 180°) do
      m = m − 1;
    m = m + 1;
    qm = pk;
  }
}
```

no notes

Graham's Scan Analysis

Time complexity:

- Other than Simple Polygon, all steps take $O(n)$ time.
- Thus, total cost is $\mathrm{O}(n \log n)$.



no notes

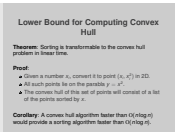
Lower Bound for Computing Convex Hull

Theorem: Sorting is transformable to the convex hull problem in linear time.

Proof:

- Given a number x_i, convert it to point (x_i, x_i^2) in 2D.
- All such points lie on the parabla $y = x^2$.
- The convex hull of this set of points will consist of a list of the points sorted by x.

Corollary: A convex hull algorithm faster than $\mathrm{O}(n \log n)$ would provide a sorting algorithm faster than $\mathrm{O}(n \log n)$.



WARNING: These are the most important two slides of the semester!

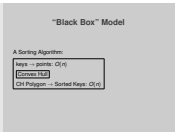
"Black Box" Model

A Sorting Algorithm:

keys $\rightarrow$ points: $O(n)$

Convex Hull

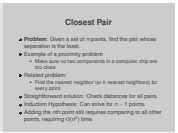
CH Polygon $\rightarrow$ Sorted Keys: $O(n)$



This is the fundamental concept of a reduction. We will use this constantly for the rest of the semester.

Closest Pair

- **Problem**: Given a set of n points, find the pair whose separation is the least.
- Example of a proximity problem
  - Make sure no two components in a computer chip are too close.
- Related problem:
  - Find the nearest neighbor (or k nearest neighbors) for every point.
- Straightforward solution: Check distances for all pairs.
- Induction Hypothesis: Can solve for $n - 1$ points.
- Adding the nth point still requires comparing to all other points, requiring $\mathrm{O}(n^2)$ time.



Next try: Ordering the points by x value still doesn't help.

Divide and Conquer Algorithm

- Approach: Split into two equal size sets, solve for each, and rejoin.
- How to split?
  - Want as much valid information as possible to result.
- Try splitting into two disjoint parts separated by a dividing plane.
- Then, need only worry about points close to the dividing plane when rejoining.
- To divide: Sort by x value and split in the middle.

Divide and Conquer Algorithm
- Approach: Split into two equal size sets, solve for each, and rejoin.
- How to split?
  - Want as much valid information as possible to result.
- Try splitting into two disjoint parts separated by a dividing plane.
- Then, need only worry about points close to the dividing plane when rejoining.
- To divide: Sort by x value and split in the middle.

Assume $n = 2^k$ points.

Note: We will actually compute smallest distance, not pair of points with smallest distance.

Closest Pair Algorithm

Induction Hypothesis:

- We can solve closest pair for two sets of size $n/2$ named P_1 and P_2.

Let minimal distance in P_1 be d_1, and for P_2 be d_2.

- Assume $d_1 \leq d_2$.

Only points in the strip of width d_1 to either side of the dividing line need to be considered.

Worst case: All points are in the strip.

Closest Pair Algorithm
Induction Hypothesis:
- We can solve closest pair for two sets of size n/2 named P_1 and P_2.
Let minimal distance in P_1 be d_1, and for P_2 be d_2.
- Assume $d_1 \leq d_2$.
Only points in the strip of width d_1 to either side of the dividing line need to be considered.
Worst case: All points are in the strip.

See Manber Figure 8.13

Closest Pair Algorithm (cont)

Observation:

- A single point can be close to only a limited number of points from the other set.

Reason: Points in the other set are at least d_1 distance apart.

Sorting by y value limits the search required.

Closest Pair Algorithm (cont)
Observation:
- A single point can be close to only a limited number of points from the other set.
Reason: Points in the other set are at least d_1 distance apart.
Sorting by y value limits the search required.

See Manber Figure 8.14

Closest Pair Algorithm Cost

$\mathrm{O}(n \log n)$ to sort by x coordinates.

Eliminate points outside strip: $\mathrm{O}(n)$.

Sort according to y coordinate: $\mathrm{O}(n \log n)$.

Scan points in strip, comparing against the other strip: $\mathrm{O}(n)$.

$T(n) = 2T(n/2) + O(n \log n)$.

$T(n) = O(n \log^2 n)$.

Closest Pair Algorithm Cost
O(n log n) to sort by x coordinates.
Eliminate points outside strip: O(n).
Sort according to y coordinate: O(n log n).
Scan points in strip, comparing against the other strip: O(n).
$T(n) = 2T(n/2) + O(n \log n)$.
$T(n) = O(n \log^2 n)$.

no notes

A Faster Algorithm

The bottleneck was sorting by y coordinate.

If solving the subproblem gave us a sorted set, this would be avoided.

Strengthen the induction hypothesis:

- Given a set of $< n$ points, we know how to find the closest distance and how to output the set ordered by the points' y coordinates.

All we need do is merge the two sorted sets – an $\mathrm{O}(n)$ step.

$T(n) = 2T(n/2) + O(n)$.
$T(n) = O(n \log n)$.

A Faster Algorithm
The bottleneck was sorting by y coordinate.
If solving the subproblem gave us a sorted set, this would be avoided.
Strengthen the induction hypothesis:
- Given a set of $< n$ points, we know how to find the closest distance and how to output the set ordered by the points' y coordinates.
All we need do is merge the two sorted sets – an $O(n)$ step.
$T(n) = 2T(n/2) + O(n)$.
$T(n) = O(n \log n)$.

no notes

Horizontal and Vertical Segments

- Intersection Problems:
  - Detect if any intersections ...
  - Report any intersections ...

  ... of a set of <line segments>.
- We can simplify the problem by restricting to vertical and horizontal line segments.
- Example applications:
  - Determine if wires or components of a VLSI design cross.
  - Determine if they are too close.
    - Solution: Expand by 1/2 the tolerance distance and check for intersection.
  - Hidden line/hidden surface elimination for Computer Graphics.

Horizontal and Vertical Segments
- Intersection Problems:
  - Detect if any intersections ...
  - Report any intersections ...

  ... of a set of <line segments>.
- We can simplify the problem by restricting to vertical and horizontal line segments.
- Example applications:
  - Determine if wires or components of a VLSI design cross.
  - Determine if they are too close.
    - Solution: Expand by 1/2 the tolerance distance and check for intersection.
  - Hidden line/hidden surface elimination for Computer Graphics.

no notes

Sweep Line Algorithms (1)

Problem: Given a set of n horizontal and m vertical line segments, find all intersections between them.

- Assume no intersections between 2 vertical or 2 horizontal lines.

Straightforward algorithm: Make all $n \times m$ comparisons.

If there are $n \times m$ intersections, this cannot be avoided.

However, we would like to do better when there are fewer intersections.

Solution: Special order of induction will be imposed by a **sweep line**.

Sweep Line Algorithms (1)
Problem: Given a set of n horizontal and m vertical line segments, find all intersections between them.
- Assume no intersections between 2 vertical or 2 horizontal lines.

Straightforward algorithm: Make all $n \times m$ comparisons.
If there are $n \times m$ intersections, this cannot be avoided.
However, we would like to do better when there are fewer intersections.
Solution: Special order of induction will be imposed by a **sweep line**.

This is a "classic" computational geometry problem/algorithm

Sweep Line Algorithms (2)

Plane sweep or **sweep line** algorithms pass an imaginary line through the set of objects.

As objects are encountered, they are stored in a data structure.

When the sweep passes, they are removed.

Preprocessing Step:

- Sort all line segments by x coordinate.

Sweep Line Algorithms (2)
Plane sweep or **sweep line** algorithms pass an imaginary line through the set of objects.
As objects are encountered, they are stored in a data structure.
When the sweep passes, they are removed.
Preprocessing Step:
- Sort all line segments by x coordinate.

The induction here is to add a special nth element.

Sweep Line Algorithms (3)

Inductive approach:

- We have already processed the first $k-1$ end points when we encounter endpoint k.
- Furthermore, we store necessary information about the previous line segments to efficiently calculate intersections with the line for point k.

Possible approaches:

1. Store vertical lines, calculate intersection for horizontal lines.
2. Store horizontal lines, calculate intersection for vertical lines.

Sweep Line Algorithms (3)

Inductive approach:
- We have already processed the first $k-1$ end points when we encounter endpoint k.
- Furthermore, we store necessary information about the previous line segments to efficiently calculate intersections with the line for point k.

Possible approaches:
1. Store vertical lines, calculate intersection for horizontal lines.
2. Store horizontal lines, calculate intersection for vertical lines.

Since we processed by x coordinate (i.e., sweeping horizontally) do (2). When we process a vertical line, it is clear which horizontal lines would be relevent (the ones that cross that include the x coordinate of the vertical line), and so could hope to find them in a data structure. If we stored vertical lines, when we process the next horizontal line, it is not so obvious how to find all vertical lines in the horizontal range.

Organizing Sweep Info

What do we need when encountering line L?

- NOT horizontal lines whose right endpoint is to the left of L.
- Maintain **active** line segments.

What do we check for intersection?

Induction Hypothesis:

- Given a list of k sorted coordinates, we know how to report all intersections among the corresponding lines that occur to the left of $k.x$, and to eliminate horizontal lines to the left of k.

Organizing Sweep Info

What do we need when encountering line L?
- NOT horizontal lines whose right endpoint is to the left of L.
- Maintain active line segments.

What do we check for intersection?

Induction Hypothesis:
- Given a list of k sorted coordinates, we know how to report all intersections among the corresponding lines that occur to the left of $k.x$, and to eliminate horizontal lines to the left of k.

See Figure 8.17 in Manber.

y coordinates of the active horizontal lines.

Sweep Line Tasks

Things to do:

1. $(k+1)$th endpoint is right endpoint of horizontal line.
   - Delete horizontal line.
2. $(k+1)$th endpoint is left endpoint of horizontal line.
   - Insert horizontal line.
3. $(k+1)$th endpoint is vertical line.
   - Find intersections with stored horizontal lines.

Sweep Line Tasks

Things to do:
1. $(k+1)$th endpoint is right endpoint of horizontal line.
   - Delete horizontal line.
2. $(k+1)$th endpoint is left endpoint of horizontal line.
   - Insert horizontal line.
3. $(k+1)$th endpoint is vertical line.
   - Find intersections with stored horizontal lines.

Deleting horizontal line is $\mathrm{O}(\log n)$.

Inserting horizontal line is $\mathrm{O}(\log n)$.

Finding intersections is $O(\log n + r)$ for r intersections.

Data Structure Requirements (1)

To have an efficient algorithm, we need efficient

- Intersection
- Deletion
- 1 dimensional range query

Example solution: Balanced search tree

- Insert, delete, locate in $\log n$ time.
- Each additional intersection calculation is of constant cost beyond first (traversal of tree).

Data Structure Requirements (1)

To have an efficient algorithm, we need efficient
- Intersection
- Deletion
- 1 dimensional range query

Example solution: Balanced search tree
- Insert, delete, locate in $\log n$ time.
- Each additional intersection calculation is of constant cost beyond first (traversal of tree).

no notes

Data Structure Requirements (2)

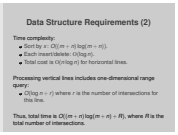
Time complexity:

- Sort by x: $O((m+n)\log(m+n))$.
- Each insert/delete: $\mathrm{O}(\log n)$.
- Total cost is $\mathrm{O}(n \log n)$ for horizontal lines.

Processing vertical lines includes one-dimensional range query:

- $O(\log n + r)$ where r is the number of intersections for this line.

Thus, total time is $O((m+n)\log(m+n)+R)$, where R is the total number of intersections.



no notes

Reductions

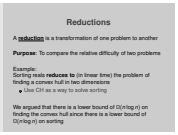
A **reduction** is a transformation of one problem to another

Purpose: To compare the relative difficulty of two problems

Example:
Sorting reals **reduces to** (in linear time) the problem of finding a convex hull in two dimensions

- Use CH as a way to solve sorting

We argued that there is a lower bound of $\Omega(n \log n)$ on finding the convex hull since there is a lower bound of $\Omega(n \log n)$ on sorting

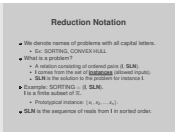


This example we have already seen.

NOT reduce CH to sorting – that just means that we can make CH as hard as sorting! Using sorting isn't necessarily the only way to solve the CH problem, perhaps there is a better way. So just knowing that sorting is ONE WAY to solve CH doesn't tell us anything about the cost of CH. On the other hand, by showing that we can use CH as a tool to solve sorting, we know that CH cannot be faster than sorting.

Reduction Notation

- We denote names of problems with all capital letters.
  - Ex: SORTING, CONVEX HULL
- What is a problem?
  - A relation consisting of ordered pairs (**I**, **SLN**).
  - **I** comes from the set of **instances** (allowed inputs).
  - **SLN** is the solution to the problem for instance **I**.
- Example: SORTING = (**I**, **SLN**).
  I is a finite subset of $\mathcal{R}$.
  - Prototypical instance: $\{x_1, x_2, ..., x_n\}$.
- **SLN** is the sequence of reals from **I** in sorted order.



no notes

Black Box Reduction (1)

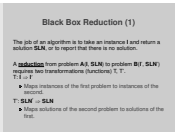
The job of an algorithm is to take an instance **I** and return a solution **SLN**, or to report that there is no solution.

A **reduction** from problem **A**(**I**, **SLN**) to problem **B**(**I**', **SLN**') requires two transformations (functions) T, T'.
T: $\mathbf{I} \Rightarrow \mathbf{I}'$

- Maps instances of the first problem to instances of the second.

T': $\mathbf{SLN}' \Rightarrow \mathbf{SLN}$

- Maps solutions of the second problem to solutions of the first.



no notes

Black Box Reduction (2)

Black box idea:

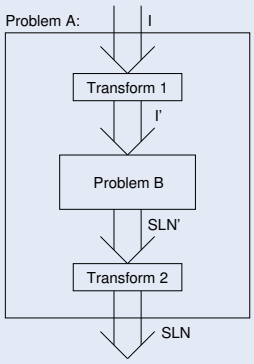
1. Start with an instance **I** of problem **A**.
2. Transform to an instance **I**' = T(**I**), an instance of problem **B**.
3. Use a "black box" algorithm for **B** as a subroutine to find a solution **SLN**' for **B**.
4. Transform to a solution **SLN** = T'(**SLN**'), a solution to the original instance **I** for problem **A**.

Black Box Reduction (2)

Black box idea:
1. Start with an instance **I** of problem **A**.
2. Transform to an instance **I**' = T(**I**), an instance of problem **B**.
3. Use a "black box" algorithm for **B** as a subroutine to find a solution **SLN**' for **B**.
4. Transform to a solution **SLN** = T'(**SLN**'), a solution to the original instance **I** for problem **A**.

no notes

Black Box Diagram



Black Box Diagram

no notes

More Notation

If (**I**, **SLN**) reduces to (**I**′, **SLN**′), write:
(**I**, **SLN**) $\leq$ (**I**′, **SLN**′).

This notation suggests that (**I**, **SLN**) is **no harder than** (**I**′, **SLN**′).

Examples:
- SORTING $\leq$ CONVEX HULL

The time complexity of T and T' is important to the time complexity of the black box algorithm for (**I**, **SLN**).

If combined time complexity is $O(g(n))$, write:
(**I**, **SLN**) $\leq_{O(g(n))}$ (**I**′, **SLN**′).

More Notation

If (**I**, **SLN**) reduces to (**I**′, **SLN**′), write:
(**I**, **SLN**) $\leq$ (**I**′, **SLN**′).

This notation suggests that (**I**, **SLN**) is **no harder than** (**I**′, **SLN**′).

Examples:
- SORTING $\leq$ CONVEX HULL

The time complexity of T and T' is important to the time complexity of the black box algorithm for (**I**, **SLN**).

If combined time complexity is $O(g(n))$, write:
(**I**, **SLN**) $\leq_{O(g(n))}$ (**I**′, **SLN**′).

Sorting is no harder than Convex Hull. Conversely, Convex Hull is *at least as hard as* Sorting.

If T or T' is expensive, then we have proved nothing about the relative bounds.

Reduction Example

SORTING = (**I**, **SLN**)
CONVEX HULL = (**I**', **SLN**').

1. **I** $= \{x_1, x_2, ..., x_n\}$.
2. T(**I**) = **I**' $= \{(x_1, x_1^2), (x_2, x_2^2), ..., (x_n, x_n^2)\}$.
3. Solve CONVEX HULL for **I**' to give solution **SLN**' $= \{(x_{i[1]}, x_{i[1]}^2), (x_{i[2]}, x_{i[2]}^2), ..., (x_{i[n]}, x_{i[n]}^2)\}$.
4. T' finds a solution to **I** from **SLN**' as follows:
   1. Find $(x_{i[k]}, x_{i[k]}^2)$ such that $x_{i[k]}$ is minimum.
   2. Y $= x_{i[k]}, x_{i[k+1]}, ..., x_{i[n]}, x_{i[1]}, ..., x_{i[k-1]}$.

- For a reduction to be useful, T and T' must be functions that can be computed by algorithms.
- An algorithm for the second problem gives an algorithm for the first problem by steps 2 – 4.

Reduction Example

SORTING = (**I**, **SLN**)
CONVEX HULL = (**I**', **SLN**').

1. **I** $= \{x_1, x_2, ..., x_n\}$.
2. T(**I**) = **I**' $= \{(x_1, x_1^2), (x_2, x_2^2), ..., (x_n, x_n^2)\}$.
3. Solve CONVEX HULL for **I**' to give solution **SLN**' $= \{(x_{i[1]}, x_{i[1]}^2), (x_{i[2]}, x_{i[2]}^2), ..., (x_{i[n]}, x_{i[n]}^2)\}$.
4. T' finds a solution to **I** from **SLN**' as follows:
   1. Find $(x_{i[k]}, x_{i[k]}^2)$ such that $x_{i[k]}$ is minimum.
   2. Y $= x_{i[k]}, x_{i[k+1]}, ..., x_{i[n]}, x_{i[1]}, ..., x_{i[k-1]}$.

- For a reduction to be useful, T and T' must be functions that can be computed by algorithms.
- An algorithm for the second problem gives an algorithm for the first problem by steps 2 – 4.

no notes

Notation Warning

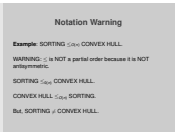
Example: SORTING $\leq_{O(n)}$ CONVEX HULL.

WARNING: $\leq$ is NOT a partial order because it is NOT antisymmetric.

SORTING $\leq_{0(n)}$ CONVEX HULL.

CONVEX HULL $\leq_{O(n)}$ SORTING.

But, SORTING $\neq$ CONVEX HULL.



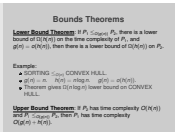
no notes

Bounds Theorems

Lower Bound Theorem: If $P_1 \leq_{O(g(n))} P_2$, there is a lower bound of $\Omega(h(n))$ on the time complexity of P_1, and $g(n) = o(h(n))$, then there is a lower bound of $\Omega(h(n))$ on P_2.

Example:
- SORTING $\leq_{O(n)}$ CONVEX HULL.
- $g(n) = n$. $h(n) = n \log n$. $g(n) = o(h(n))$.
- Theorem gives $\Omega(n \log n)$ lower bound on CONVEX HULL.

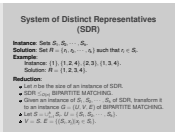
Upper Bound Theorem: If P_2 has time complexity $O(h(n))$ and $P_1 \leq_{O(g(n))} P_2$, then P_1 has time complexity $O(g(n) + h(n))$.



Notice o, not O.So, given good transformations, both problems take at least $\Omega(P_1)$ and at most $O(P_2)$.

System of Distinct Representatives (SDR)

Instance: Sets $S_1, S_2, \cdots, S_k$.
Solution: Set $R = \{r_1, r_2, \cdots, r_k\}$ such that $r_i \in S_i$.
Example:
Instance: $\{1\}, \{1,2,4\}, \{2,3\}, \{1,3,4\}$.
Solution: $R = \{1,2,3,4\}$.
Reduction:
- Let n be the size of an instance of SDR.
- SDR $\leq_{O(n)}$ BIPARTITE MATCHING.
- Given an instance of $S_1, S_2, \cdots, S_k$ of SDR, transform it to an instance $G = (U, V, E)$ of BIPARTITE MATCHING.
- Let $S = \cup_{i=1}^{k} S_i$. $U = \{S_1, S_2, \cdots, S_k\}$.
- $V = S$. $E = \{(S_i, x_j) | x_j \in S_i\}$.



Since it is a set, there are no duplicates.

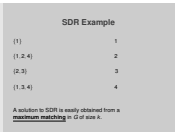
Or, $R = \{1, 4, 2, 3\}$

U is the sets.
V is the elements from all of the sets (union the sets).
E matches elements to sets.

SDR Example

$\{1\}$	1
$\{1,2,4\}$	2
$\{2,3\}$	3
$\{1,3,4\}$	4

A solution to SDR is easily obtained from a **maximum matching** in G of size k.



Need better figure here.

Simple Polygon Lower Bound (1)

- SIMPLE POLYGON: Given a set of n points in the plane, find a simple polygon with those points as vertices.
- SORTING $\leq_{O(n)}$ SIMPLE POLYGON.
- Instance of SORTING: $\{x_1, x_2, \cdots, x_n\}$.
  - In linear time, find $M = \max |x_i|$.
  - Let C be a circle centered at the origin, of radius M.
- Instance of SIMPLE POLYGON:

$$\{(x_1, \sqrt{M^2 - x_1^2}), \cdots, (x_n, \sqrt{M^2 - x_n^2})\}.$$

  All these points fall on C in their sorted order.
- The only simple polygon having the points on C as vertices is the convex one.

Simple Polygon Lower Bound (1)
▸ SIMPLE POLYGON: Given a set of n points in the plane, find a simple polygon with those points as vertices.
▸ SORTING $\leq_{O(n)}$ SIMPLE POLYGON.
▸ Instance of SORTING: $\{x_1, x_2, \cdots, x_n\}$.
  ▸ In linear time, find $M = \max |x_i|$.
  ▸ Let C be a circle centered at the origin, of radius M.
▸ Instance of SIMPLE POLYGON:
$\{(x_1, \sqrt{M^2 - x_1^2}), \cdots, (x_n, \sqrt{M^2 - x_n^2})\}$.
All these points fall on C in their sorted order.
▲ The only simple polygon having the points on C as vertices is the convex one.

Need a figure here showing the curve.

Simple Polygon Lower Bound (2)

- As with CONVEX HULL, the sorted order is easily obtained from the solution to SIMPLE POLYGON.
- By the Lower Bound Theorem, SIMPLE POLYGON is $\Omega(n \log n)$.

Simple Polygon Lower Bound (2)
▸ As with CONVEX HULL, the sorted order is easily obtained from the solution to SIMPLE POLYGON.
▸ By the Lower Bound Theorem, SIMPLE POLYGON is $\Omega(n \log n)$.

no notes

Matrix Multiplication

Matrix multiplication can be reduced to a number of other problems.

In fact, certain special cases of MATRIX MULTIPLY are equivalent to MATRIX MULTIPLY in asymptotic complexity.

SYMMETRIC MATRIX MULTIPLY (SYM):
- Instance: a symmetric $n \times n$ matrix.

MATRIX MULTIPLY $\leq_{O(n^2)}$ SYM.

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix}$$

Matrix Multiplication
Matrix multiplication can be reduced to a number of other problems.
In fact, certain special cases of MATRIX MULTIPLY are equivalent to MATRIX MULTIPLY in asymptotic complexity.
SYMMETRIC MATRIX MULTIPLY (SYM):
▸ Instance: a symmetric $n \times n$ matrix.
MATRIX MULTIPLY $\leq_{O(n^2)}$ SYM.
$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix}$

Clearly SYM is not harder than MM. Is it easier? No...

So, having a good SYM would give a good MM. The other way of looking at it is that SYM is just as hard as MM.

Matrix Squaring

Problem: Compute A^2 where A is an $n \times n$ matrix.

MATRIX MULTIPLY $\leq_{O(n^2)}$ SQUARING.

$$\begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}^2 = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}$$

Matrix Squaring
Problem: Compute A^2 where A is an $n \times n$ matrix.
MATRIX MULTIPLY $\leq_{O(n^2)}$ SQUARING.
$\begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}^2 = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}$

no notes

Linear Programming (LP)

Maximize or minimize a linear function subject to linear constraints.
Variables: vector $\mathbf{X} = (x_1, x_2, \cdots, x_n)$.

Objective Function: $\mathbf{c} \cdot \mathbf{X} = \sum c_i x_i$.
Inequality Constraints: $\mathbf{A_i} \cdot \mathbf{X} \leq b_i \quad 1 \leq i \leq k$.
Equality Constraints: $\mathbf{E_i} \cdot \mathbf{X} = d_i \quad 1 \leq i \leq m$.

Non-negative Constraints: $x_i \geq 0$ for some is.

Linear Programming (LP)

Maximize or minimize a linear function subject to linear constraints.
Variables: vector $\mathbf{X} = (x_1, x_2, \cdots, x_n)$.

Objective Function: $\mathbf{c} \cdot \mathbf{X} = \sum c_i x_i$.
Inequality Constraints: $\mathbf{A_i} \cdot \mathbf{X} \leq b_i \quad 1 \leq i \leq k$.
Equality Constraints: $\mathbf{E_i} \cdot \mathbf{X} = d_i \quad 1 \leq i \leq m$.

Non-negative Constraints: $x_i \geq 0$ for some is.

Example of a "super problem" that many problems can reduce to.

Objective function defeinse what we want to minimize.

$\mathbf{A_i}$ is a vector – k vectors give the k b's.

Not all of the constraint types are used for every problem.

Use of LP

Reasons for considering LP:

- Practical algorithms exist to solve LP.
- Many real-world optimization problems are naturally stated as LP.
- Many optimization problems are reducible to LP.

Use of LP

Reasons for considering LP:
- Practical algorithms exist to solve LP.
- Many real-world optimization problems are naturally stated as LP.
- Many optimization problems are reducible to LP.

no notes

Network Flow Reduction (1)

- Reduce NETWORK FLOW to LP.
- Let $x_1, x_2, \cdots, x_n$ be the flows through edges.
- Objective function: For $S =$ edges out of the source, maximize
$$\sum_{i \in S} x_i.$$
- Capacity constraints: $x_i \leq c_i \quad 1 \leq i \leq n$.
- Flow conservation:
  For a vertex $v \in V - \{s, t\}$,
  let $Y(v) =$ set of x_i for edges leaving v.
  $Z(v) =$ set of x_i for edges entering v.
$$\sum_{Z(V)} x_i - \sum_{Y(V)} x_i = 0.$$

Network Flow Reduction (1)

- Reduce NETWORK FLOW to LP.
- Let $x_1, x_2, \cdots, x_n$ be the flows through edges.
- Objective function: For $S =$ edges out of the source, maximize
$\sum_{i \in S} x_i$.
- Capacity constraints: $x_i \leq c_i \quad 1 \leq i \leq n$.
- Flow conservation:
  For a vertex $v \in V - \{s, t\}$,
  let $Y(v) =$ set of x_i for edges leaving v.
  $Z(v) =$ set of x_i for edges entering v.
$\sum_{Z(V)} x_i - \sum_{Y(V)} x_i = 0$.

Obviously, maximize the objective function by maximizing the X_i's!! But we can't do that arbirarily because of the constraints.

Network Flow Reduction (2)

Non-negative Constraints: $x_i \geq 0 \quad 1 \leq i \leq n$.
Maximize: $x_1 + x_4$ subject to:

$$\begin{aligned} x_1 &\leq 4 \\ x_2 &\leq 3 \\ x_3 &\leq 2 \\ x_4 &\leq 5 \\ x_5 &\leq 7 \\ x_1 + x_3 - x_2 &= 0 \\ x_4 - x_3 - x_5 &= 0 \\ x_1, \cdots, x_5 &\geq 0 \end{aligned}$$

Network Flow Reduction (2)

Non-negative constraints: $x_i \geq 0 \quad 1 \leq i \leq n$.
Maximize: $x_1 + x_4$ subject to:

$$\begin{aligned} x_1 &\leq 4 \\ x_2 &\leq 3 \\ x_3 &\leq 2 \\ x_4 &\leq 5 \\ x_5 &\leq 7 \\ x_1 + x_3 - x_2 &= 0 \\ x_4 - x_3 - x_5 &= 0 \\ x_1, \cdots, x_5 &\geq 0 \end{aligned}$$

Need graph:
Vertices: s, a, b, t.

Edges:

- s $\rightarrow$ a with capacity $c_1 = 4$.
- a $\rightarrow$ t with capacity $c_2 = 3$.
- a $\rightarrow$ b with capacity $c_3 = 2$.
- s $\rightarrow$ b with capacity $c_4 = 5$.
- b $\rightarrow$ t with capacity $c_5 = 7$.

Matching

- Start with graph $G = (V, E)$.
- Let $x_1, x_2, \cdots, x_n$ represent the edges in E.
  - $x_i = 1$ means edge i is **matched**.
- Objective function: Maximize
$$\sum_{i=1}^{n} x_i.$$
- subject to: (Let $N(v)$ denote edges incident on v)
$$\sum_{N(V)} x_i \leq 1$$
$$x_i \geq 0 \quad 1 \leq i \leq n$$
- Integer constraints: Each x_i must be an integer.
- Integer constraints makes this INTEGER LINEAR PROGRAMMING (ILP).

Matching
- Start with graph $G = (V, E)$.
- Let $x_1, x_2, \cdots, x_n$ represent the edges in E.
  - $x_i = 1$ means edge i is matched.
- Objective function: Maximize $\sum_{i=1}^{n} x_i$.
- subject to: (Let $N(v)$ denote edges incident on v) $\sum_{N(V)} x_i \leq 1$, $x_i \geq 0 \quad 1 \leq i \leq n$.
- Integer constraints: Each x_i must be an integer.
- Integer constraints makes this INTEGER LINEAR PROGRAMMING (ILP).

no notes

Summary

NETWORK FLOW $\leq_{O(n)}$ LP.

MATCHING $\leq_{O(n)}$ ILP.

Summary

NETWORK FLOW $\leq_{O(n)}$ LP.

MATCHING $\leq_{O(n)}$ ILP.

no notes

Summary of Reduction

Importance:

1. Compare difficulty of problems.
2. Prove new lower bounds.
3. Black box algorithms for "new" problems in terms of (already solved) "old" problems.
4. Provide insights.

Warning:

- A reduction **does not** provide an algorithm to solve a problem – only a transformation.
- Therefore, when you look for a reduction, you are **not** trying to solve either problem.

Summary of Reduction

Importance:
1. Compare difficulty of problems.
2. Prove new lower bounds.
3. Black box algorithms for "new" problems in terms of (already solved) "old" problems.
4. Provide insights.

Warning:
- A reduction **does not** provide an algorithm to solve a problem – only a transformation.
- Therefore, when you look for a reduction, you are **not** trying to solve either problem.

no notes

Another Warning

The notation $P_1 \leq P_2$ is meant to be suggestive.

Think of P_1 as the easier, P_2 as the harder problem.

Always transform from instance of P_1 to instance of P_2.

Common mistake: Doing the reduction backwards (from P_2 to P_1).

DON'T DO THAT!

Another Warning

The notation $P_1 \leq P_2$ is meant to be suggestive.

Think of P_1 as the easier, P_2 as the harder problem.

Always transform from instance of P_1 to instance of P_2.

Common mistake: Doing the reduction backwards (from P_2 to P_1).

DON'T DO THAT!

no notes

Common Problems used in Reductions

NETWORK FLOW

MATCHING

SORTING

LP

ILP

MATRIX MULTIPLICATION

SHORTEST PATHS

Common Problems used in Reductions
NETWORK FLOW
MATCHING
SORTING
LP
ILP
MATRIX MULTIPLICATION
SHORTEST PATHS

no notes

Tractable Problems

We would like some convention for distinguishing tractable from intractable problems.
A problem is said to be **tractable** if an algorithm exists to solve it with polynomial time complexity: $O(p(n))$.

- It is said to be **intractable** if the best known algorithm requires exponential time.

Examples:

- Sorting: $O(n^2)$
- Convex Hull: $O(n^2)$
- Single source shortest path: $O(n^2)$
- All pairs shortest path: $O(n^3)$
- Matrix multiplication: $O(n^3)$

Tractable Problems
We would like some convention for distinguishing tractable from intractable problems.
A problem is said to be tractable if an algorithm exists to solve it with polynomial time complexity: $O(p(n))$.
- It is said to be intractable if the best known algorithm requires exponential time.
Examples:
- Sorting: $O(n^2)$
- Convex Hull: $O(n^2)$
- Single source shortest path: $O(n^2)$
- All pairs shortest path: $O(n^3)$
- Matrix multiplication: $O(n^3)$

Log-polynomial is $O(n \log n)$

Like any simple rule of thumb for catagorizing, in some cases the distinction between polynomial and exponential could break down. For example, one can argue that, for practical problems, 1.01^n is preferable to n^{25}. But the reality is that very few polynomial-time algorithms have high degree, and exponential-time algorithms nearly always have a constant of 2 or greater. Nearly all algorithms are either low-degree polynomials or "real" exponentials, with very little in between.

Tractable Problems (cont)

The technique we will use to classify one group of algorithms is based on two concepts:

1. A special kind of reduction.
2. Nondeterminism.

Tractable Problems (cont)
The technique we will use to classify one group of algorithms is based on two concepts:
1. A special kind of reduction.
2. Nondeterminism.

no notes

Decision Problems

(I, S) such that S(X) is always either "yes" or "no."

- Usually formulated as a question.

Example:

- Instance: A weighted graph $G = (V, E)$, two vertices s and t, and an integer K.
- Question: Is there a path from s to t of length $\leq K$? In this example, the answer is "yes."

Decision Problems
(I, S) such that S(X) is always either "yes" or "no."
- Usually formulated as a question.
Example:
- Instance: A weighted graph $G = (V, E)$, two vertices s and t, and an integer K.
- Question: Is there a path from s to t of length $\leq K$? In this example, the answer is "yes."

Need a graph here.

Decision Problems (cont)

Can also be formulated as a language recognition problem:

- Let L be the subset of I consisting of instances whose answer is "yes." Can we recognize L?

The class of tractable problems $\mathcal{P}$ is the class of languages or decision problems recognizable in polynomial time.

Decision Problems (cont)

Can also be formulated as a language recognition problem:
- Let L be the subset of I consisting of instances whose answer is "yes." Can we recognize L?

The class of tractable problems $\mathcal{P}$ is the class of languages or decision problems recognizable in polynomial time.

Following our graph example: It is possible to translate from a graph to a string representation, and to define a subset of such strings as corresponding to graphs with a path from s to t. This subset defines a language to "recognize."

Polynomial Reducibility

Reduction of one language to another language.

Let $L_1 \subset I_1$ and $L_2 \subset I_2$ be languages. L_1 is **polynomially reducible** to L_2 if there exists a transformation $f : I_1 \to I_2$, computable in polynomial time, such that $f(x) \in L_2$ if and only if $x \in L_1$.
We write: $L_1 \leq_p L_2$ or $L_1 \leq L_2$.

Polynomial Reducibility

Reduction of one language to another language.

Let $L_1 \subset I_1$ and $L_2 \subset I_2$ be languages. L_1 is **polynomially reducible** to L_2 if there exists a transformation $f : I_1 \to I_2$, computable in polynomial time, such that $f(x) \in L_2$ if and only if $x \in L_1$.
We write: $L_1 \leq_p L_2$ or $L_1 \leq L_2$.

Or one decision problem to another.

Specialized case of reduction from Chapter 10.

Examples

- CLIQUE $\leq_p$ INDEPENDENT SET.
- An instance I of CLIQUE is a graph $G = (V, E)$ and an integer K.
- The instance $I' = f(I)$ of INDEPENDENT SET is the graph $G' = (V, E')$ and the integer K, were an edge $(u, v) \in E'$ iff $(u, v) \notin E$.
- f is computable in polynomial time.

Examples

- CLIQUE $\leq_p$ INDEPENDENT SET.
- An instance I of CLIQUE is a graph $G = (V, E)$ and an integer K.
- The instance $I' = f(I)$ of INDEPENDENT SET is the graph $G' = (V, E')$ and the integer K, were an edge $(u, v) \in E'$ iff $(u, v) \notin E$.
- f is computable in polynomial time.

no notes

Transformation Example

- G has a clique of size $\geq K$ iff G' has an independent set of size $\geq K$.
- Therefore, CLIQUE $\leq_p$ INDEPENDENT SET.
- **IMPORTANT WARNING:** The reduction does not **solve** either INDEPENDENT SET or CLIQUE, it merely transforms one into the other.

Transformation Example

- G has a clique of size $\geq K$ iff G' has an independent set of size $\geq K$.
- Therefore, CLIQUE $\leq_p$ INDEPENDENT SET.
- **IMPORTANT WARNING:** The reduction does not **solve** either INDEPENDENT SET or CLIQUE, it merely transforms one into the other.

Need a graph here.

If nodes in G' are independent, then no connections. Thus, in G they all connect.

Nondeterminism

Nondeterminism allows an algorithm to make an arbitrary choice among a finite number of possibilities.

Implemented by the "nd-choice" primitive:
nd-choice(ch_1, ch_2, ..., ch_j)
returns one of the choices ch_1, ch_2, ... **arbitrarily**.

Nondeterministic algorithms can be thought of as "correctly guessing" (choosing nondeterministically) a solution.

Alternatively, nondeterminsitic algorithms can be thought of as running on super-parallel machines that make all choices simultaneously and then reports the "right" solution.

Nondeterminism

Nondeterminism allows an algorithm to make an arbitrary choice among a finite number of possibilities.

Implemented by the "nd-choice" primitive:
nd-choice(ch1, ch2, ..., chj)
returns one of the choices ch1, ch2, ... arbitrarily.

Nondeterministic algorithms can be thought of as "correctly guessing" (choosing nondeterministically) a solution.

Alternatively, nondeterminsitic algorithms can be thought of as running on super-parallel machines that make all choices simultaneously and then reports the "right" solution.

no notes

Nondeterministic CLIQUE Algorithm

```
procedure nd-CLIQUE(Graph G, int K) {
  VertexSet S = EMPTY;  int size = 0;
  for (v in G.V)
    if (nd-choice(YES, NO) == YES) then {
      S = union(S, v);
      size = size + 1;
    }
  if (size < K) then
    REJECT;      // S is too small
  for (u in S)
    for (v in S)
      if ((u <> v) && ((u, v) not in E))
        REJECT;  // S is missing an edge
  ACCEPT;
}
```


Nondeterministic CLIQUE Algorithm

procedure nd-CLIQUE(Graph G, int K) {
VertexSet S = EMPTY; int size = 0;
for (v in G.V)
if (nd-choice(YES, NO) == YES) then {
S = union(S, v);
size = size + 1;
}
if (size < K) then
REJECT; // S is too small
for (u in S)
for (v in S)
if ((u <> v) && ((u, v) not in E))
REJECT; // S is missing an edge
ACCEPT;
}

What makes this different than random guessing is that **all** choices happen "in parallel."

Nondeterministic Acceptance

- (G, K) is in the "language" CLIQUE iff there exists a sequence of nd-choice guesses that causes nd-CLIQUE to accept.
- Definition of acceptance by a nondeterministic algorithm:
  - An instance is accepted iff there exists a sequence of nondeterministic choices that causes the algorithm to accept.
- An unrealistic model of computation.
  - There are an exponential number of possible choices, but only one must accept for the instance to be accepted.
- Nondeterminism is a useful concept
  - It provides insight into the nature of certain hard problems.

Nondeterministic Acceptance

- (G, K) is in the "language" CLIQUE iff there exists a sequence of nd-choice guesses that causes nd-CLIQUE to accept.
- Definition of acceptance by a nondeterministic algorithm:
  - An instance is accepted iff there exists a sequence of nondeterministic choices that causes the algorithm to accept.
- An unrealistic model of computation.
  - There are an exponential number of possible choices, but only one must accept for the instance to be accepted.
- Nondeterminism is a useful concept
  - It provides insight into the nature of certain hard problems.

no notes

Class $\mathcal{NP}$

- The class of languages accepted by a nondeterministic algorithm in polynomial time is called $\mathcal{NP}$.
- There are an **exponential** number of different executions of nd-CLIQUE on a single instance, but any one execution requires only **polynomial time** in the size of that instance.
- Time complexity of nondeterministic algorithm is greatest amount of time required by any **one** of its executions.

Class $\mathcal{NP}$

- The class of languages accepted by a nondeterministic algorithm in polynomial time is called $\mathcal{NP}$.
- There are an **exponential** number of different executions of nd-CLIQUE on a single instance, but any one execution requires only **polynomial time** in the size of that instance.
- Time complexity of nondeterministic algorithm is greatest amount of time required by any **one** of its executions.

Note that Towers of Hanoi is not in $\mathcal{NP}$.

Class $\mathcal{NP}$(cont)

Alternative Interpretation:

- $\mathcal{NP}$ is the class of algorithms that — never mind how we got the answer — can check if the answer is correct in polynomial time.
- If you cannot verify an answer in polynomial time, you cannot hope to find the right answer in polynomial time!

Class $\mathcal{NP}$(cont)

Alternative Interpretation:
- $\mathcal{NP}$ is the class of algorithms that — never mind how we got the answer — can check if the answer is correct in polynomial time.
- If you cannot verify an answer in polynomial time, you cannot hope to find the right answer in polynomial time!

This is worded a bit loosely. Specifically, we assume that we can get the answer fast enough – that is, in polynomial time non-deterministically.

How to Get Famous

Clearly, $\mathcal{P} \subset \mathcal{NP}$.

Extra Credit Problem:

- Prove or disprove: $\mathcal{P} = \mathcal{NP}$.

This is important because there are many natural decision problems in $\mathcal{NP}$ for which no $\mathcal{P}$ (tractable) algorithm is known.

How to Get Famous

Clearly, $\mathcal{P} \subset \mathcal{NP}$.

Extra Credit Problem:
- Prove or disprove: $\mathcal{P} = \mathcal{NP}$.

This is important because there are many natural decision problems in $\mathcal{NP}$ for which no $\mathcal{P}$ (tractable) algorithm is known.

no notes

$\mathcal{NP}$-completeness

A theory based on identifying problems that are as hard as any problems in $\mathcal{NP}$.

The next best thing to knowing whether $\mathcal{P} = \mathcal{NP}$ or not.

A decision problem A is **$\mathcal{NP}$-hard** if every problem in $\mathcal{NP}$ is polynomially reducible to A, that is, for all

$$B \in \mathcal{NP}, \quad B \leq_p A.$$

A decision problem A is **$\mathcal{NP}$-complete** if $A \in \mathcal{NP}$ and A is $\mathcal{NP}$-hard.

$\mathcal{NP}$-completeness

A theory based on identifying problems that are as hard as any problems in $\mathcal{NP}$.

The next best thing to knowing whether $\mathcal{P} = \mathcal{NP}$ or not.

A decision problem A is $\mathcal{NP}$-hard if every problem in $\mathcal{NP}$ is polynomially reducible to A, that is, for all

$$B \in \mathcal{NP}, \quad B \leq_p A.$$

A decision problem A is $\mathcal{NP}$-complete if $A \in \mathcal{NP}$ and A is $\mathcal{NP}$-hard.

A is not permitted to be harder than $\mathcal{NP}$. For example, Tower of Hanoi is not in $\mathcal{NP}$. It requires exponential time to verify a set of moves.

Satisfiability

Let E be a Boolean expression over variables $x_1, x_2, \cdots, x_n$ in conjunctive normal form (CNF), that is, an AND of ORs.

$$E = (x_5 + x_7 + \overline{x_8} + x_{10}) \cdot (\overline{x_2} + x_3) \cdot (x_1 + \overline{x_3} + x_6).$$

A variable or its negation is called a **literal**.
Each sum is called a **clause**.

SATISFIABILITY (SAT):

- Instance: A Boolean expression E over variables $x_1, x_2, \cdots, x_n$ in CNF.
- Question: Is E satisfiable?

Cook's Theorem: SAT is $\mathcal{NP}$-complete.

Satisfiability

Let E be a Boolean expression over variables $x, x_2, \cdots, x_n$ in conjunctive normal form (CNF), that is, an AND of ORs.

$$E = (x_5 + x_7 + \overline{x_8} + x_{10}) \cdot (\overline{x_2} + x_3) \cdot (x_1 + \overline{x_3} + x_6).$$

A variable or its negation is called a literal.
Each sum is called a clause.

SATISFIABILITY (SAT):
- Instance: A Boolean expression E over variables $x_1, x_2, \cdots, x_n$ in CNF.
- Question: Is E satisfiable?

Cook's Theorem: SAT is $\mathcal{NP}$-complete.

Is there a truth assignment for the variables that makes E true?

Cook won a Turing award for this work.

Proof Sketch

SAT $\in \mathcal{NP}$:

- A non-deterministic algorithm **guesses** a truth assignment for $x_1, x_2, \cdots, x_n$ and **checks** whether E is true in polynomial time.
- It accepts iff there is a satisfying assignment for E.

SAT is $\mathcal{NP}$-hard:

- Start with an arbitrary problem $B \in \mathcal{NP}$.
- We know there is a polynomial-time, nondeterministic algorithm to accept B.
- Cook showed how to transform an instance X of B into a Boolean expression E that is satisfiable if the algorithm for B accepts X.

Proof Sketch

SAT $\in \mathcal{NP}$:
- A non-deterministic algorithm **guesses** a truth assignment for $x_1, x_2, \cdots, x_n$ and **checks** whether E is true in polynomial time.
- It accepts iff there is a satisfying assignment for E.

SAT is $\mathcal{NP}$-hard:
- Start with an arbitrary problem $B \in \mathcal{NP}$.
- We know there is a polynomial-time, nondeterministic algorithm to accept B.
- Cook showed how to transform an instance X of B into a Boolean expression E that is satisfiable if the algorithm for B accepts X.

The proof of this last step is usually several pages long. One approach is to develop a nondeterministic Turing Machine program to solve an arbitrary problem B in $\mathcal{NP}$.

Implications

(1) Since SAT is $\mathcal{NP}$-complete, we have not defined an empty concept.

(2) If SAT $\in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

(3) If $\mathcal{P} = \mathcal{NP}$, then SAT $\in \mathcal{P}$.

(4) If $A \in \mathcal{NP}$ and B is $\mathcal{NP}$-complete, then $B \leq_p A$ implies A is $\mathcal{NP}$-complete.
Proof:

- Let $C \in \mathcal{NP}$.
- Then $C \leq_p B$ since B is $\mathcal{NP}$-complete.
- Since $B \leq_p A$ and $\leq_p$ is transitive, $C \leq_p A$.
- Therefore, A is $\mathcal{NP}$-hard and, finally, $\mathcal{NP}$-complete.

Implications

(1) Since SAT is $\mathcal{NP}$-complete, we have not defined an empty concept.

(2) If SAT $\in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

(3) If $\mathcal{P} = \mathcal{NP}$, then SAT $\in \mathcal{P}$.

(4) If $A \in \mathcal{NP}$ and B is $\mathcal{NP}$-complete, then $B \leq_p A$ implies A is $\mathcal{NP}$-complete.
Proof:
- Let $C \in \mathcal{NP}$.
- Then $C \leq_p B$ since B is $\mathcal{NP}$-complete.
- Since $B \leq_p A$ and $\leq_p$ is transitive, $C \leq_p A$.
- Therefore, A is $\mathcal{NP}$-hard and, finally, $\mathcal{NP}$-complete.

no notes

Implications (cont)

(5) This gives a simple two-part strategy for showing a decision problem A is $\mathcal{NP}$-complete.

(a) Show $A \in \mathcal{NP}$.

(b) Pick an $\mathcal{NP}$-complete problem B and show $B \leq_p A$.

Implications (cont)

(5) This gives a simple two-part strategy for showing a decision problem A is $\mathcal{NP}$-complete.
(a) Show $A \in \mathcal{NP}$.
(b) Pick an $\mathcal{NP}$-complete problem B and show $B \leq_p A$.

Proving $A \in \mathcal{NP}$ is usually easy.

Don't get the reduction backwards!

$\mathcal{NP}$-completeness Proof Template

To show that decision problem B is $\mathcal{NP}$-complete:

1. $B \in \mathcal{NP}$
   - Give a polynomial time, non-deterministic algorithm that accepts B.
     1. Given an instance X of B, **guess** evidence Y.
     2. **Check** whether Y is evidence that $X \in B$. If so, accept X.
2. B is $\mathcal{NP}$-hard.
   - Choose a known $\mathcal{NP}$-complete problem, A.
   - Describe a polynomial-time transformation T of an **arbitrary** instance of A to a [not necessarily arbitrary] instance of B.
   - Show that $X \in A$ if and only if $T(X) \in B$.

$\mathcal{NP}$-completeness Proof Template

To show that decision problem B is $\mathcal{NP}$-complete:
1. $B \in \mathcal{NP}$
   - Give a polynomial time, non-deterministic algorithm that accepts B.
     1. Given an instance X of B, **guess** evidence Y.
     2. **Check** whether Y is evidence that $X \in B$. If so, accept
2. B is $\mathcal{NP}$-hard.
   - Choose a known $\mathcal{NP}$-complete problem, A.
   - Describe a polynomial-time transformation T of an **arbitrary** instance of A to a [not necessarily arbitrary] instance of B.
   - Show that $X \in A$ if and only if $T(X) \in B$.

$B \in \mathcal{NP}$ is usually the easy part.

The first two steps of the $\mathcal{NP}$-hard proof are usually the hardest.

3-SATISFIABILITY (3SAT)

Instance: A Boolean expression E in CNF such that each clause contains exactly 3 literals.

Question: Is there a satisfying assignment for E?

A special case of SAT.

One might hope that 3SAT is easier than SAT.

3-SATISFIABILITY (3SAT)

Instance: A Boolean expression E in CNF such that each clause contains exactly 3 literals.

Question: Is there a satisfying assignment for E?

A special case of SAT.

One might hope that 3SAT is easier than SAT.

What about 2SAT? This is in $\mathcal{P}$.

Effectively a 2-coloring graph problem. Join 2 vertices if they are in same clause, also join x_i and $\overline{x_i}$. Then, try to 2-color the graph with a DFS.

How to solve 1SAT? Answer is "yes" iff x_i and $\overline{x_i}$ are not both in list for any i.

3SAT is $\mathcal{NP}$-complete

(1) 3SAT $\in \mathcal{NP}$.

```
procedure nd-3SAT(E) {
  for (i = 1 to n)
    x[i] = nd-choice(TRUE, FALSE);
  Evaluate E for the guessed truth assignment.
  if (E evaluates to TRUE)
    ACCEPT;
  else
    REJECT;
}
```

nd-3SAT is a polynomial-time nondeterministic algorithm that accepts 3SAT.

3SAT is $\mathcal{NP}$-complete

(1) 3SAT $\in \mathcal{NP}$.

```
procedure nd-3SAT(E) {
  for (i = 1 to n)
    x[i] = nd-choice(TRUE, FALSE);
  Evaluate E for the guessed truth assignment.
  if (E evaluates to TRUE)
    ACCEPT;
  else
    REJECT;
}
```

nd-3SAT is a polynomial-time nondeterministic algorithm that accepts 3SAT.

no notes

Proving 3SAT $\mathcal{NP}$-hard

1. Choose SAT to be the known $\mathcal{NP}$-complete problem.
   - We need to show that SAT $\leq_p$ 3SAT.
2. Let $E = C_1 \cdot C_2 \cdots C_k$ be any instance of SAT.

Strategy: Replace any clause C_i that does not have exactly 3 literals with two or more clauses having exactly 3 literals.

Let $C_i = y_1 + y_2 + \cdots + y_j$ where $y_1, \cdots, y_j$ are literals.
(a) $j = 1$

- Replace (y_1) with

$$(y_1 + v + w) \cdot (y_1 + \overline{v} + w) \cdot (y_1 + v + \overline{w}) \cdot (y_1 + \overline{v} + \overline{w})$$

where v and w are new variables.

Proving 3SAT $\mathcal{NP}$-hard

1. Choose SAT to be the known $\mathcal{NP}$-complete problem.
   - We need to show that SAT $\leq_p$ 3SAT.
2. Let $E = C_1 \cdot C_2 \cdots C_k$ be any instance of SAT.

Strategy: Replace any clause C_i that does not have exactly 3 literals with two or more clauses having exactly 3 literals.

Let $C_i = y_1 + y_2 + \cdots + y_j$ where $y_1, \cdots, y_j$ are literals.
(a) $j = 1$

- Replace (y_1) with

$$(y_1 + v + w) \cdot (y_1 + \overline{v} + w) \cdot (y_1 + v + \overline{w}) \cdot (y_1 + \overline{v} + \overline{w})$$

where v and w are new variables.

SAT is the only choice that we have so far!

Replacing (y_1) with $(y_1 + y_1 + y_1)$ seems like a reasonable alternative. But some of the theory behind the definitions rejects clauses with duplicated literals.

Proving 3SAT $\mathcal{NP}$-hard (cont)

(b) $j = 2$

- Replace $(y_1 + y_2)$ with $(y_1 + y_2 + z) \cdot (y_1 + y_2 + \overline{z})$ where z is a new variable.

(c) $j > 3$

- Relace $(y_1 + y_2 + \cdots + y_j)$ with

$$(y_1 + y_2 + z_1) \cdot (y_3 + \overline{z_1} + z_2) \cdot (y_4 + \overline{z_2} + z_3) \cdots$$
$$(y_{j-2} + \overline{z_{j-4}} + z_{j-3}) \cdot (y_{j-1} + y_j + \overline{z_{j-3}})$$

where $z_1, z_2, \cdots, z_{j-3}$ are new variables.

- After replacements made for each C_i, a Boolean expression E' results that is an instance of 3SAT.
- The replacement clearly can be done by a polynomial-time deterministic algorithm.

Proving 3SAT $\mathcal{NP}$-hard (cont)

(b) $j = 2$

- Replace $(y_1 + y_2)$ with $(y_1 + y_2 + z) \cdot (y_1 + y_2 + \overline{z})$ where z is a new variable.

(c) $j > 3$

- Relace $(y_1 + y_2 + \cdots + y_j)$ with

$$(y_1 + y_2 + z_1) \cdot (y_3 + \overline{z_1} + z_2) \cdot (y_4 + \overline{z_2} + z_3) \cdots$$
$$(y_{j-2} + \overline{z_{j-4}} + z_{j-3}) \cdot (y_{j-1} + y_j + \overline{z_{j-3}})$$

where $z_1, z_2, \cdots, z_{j-3}$ are new variables.

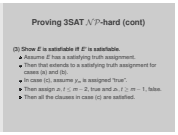
- After replacements made for each C_i, a Boolean expression E' results that is an instance of 3SAT.
- The replacement clearly can be done by a polynomial-time deterministic algorithm.

no notes

Proving 3SAT $\mathcal{NP}$-hard (cont)

(3) Show E is satisfiable iff E' is satisfiable.

- Assume E has a satisfying truth assignment.
- Then that extends to a satisfying truth assignment for cases (a) and (b).
- In case (c), assume y_m is assigned "true".
- Then assign $z_t, t \leq m-2$, true and $z_k, t \geq m-1$, false.
- Then all the clauses in case (c) are satisfied.

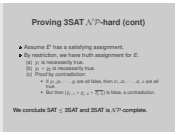


no notes

Proving 3SAT $\mathcal{NP}$-hard (cont)

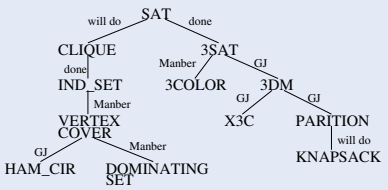
- Assume E' has a satisfying assignment.
- By restriction, we have truth assignment for E.
  (a) y_1 is necessarily true.
  (b) $y_1 + y_2$ is necessarily true.
  (c) Proof by contradiction:
    - ⋆ If $y_1, y_2, \cdots, y_j$ are all false, then $z_1, z_2, \cdots, z_{j-3}$ are all true.
    - ⋆ But then $(y_{j-1} + y_{j-2} + \overline{z_{j-3}})$ is false, a contradiction.

We conclude SAT $\leq$ 3SAT and 3SAT is $\mathcal{NP}$-complete.



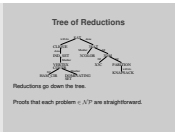
no notes

Tree of Reductions



Reductions go down the tree.

Proofs that each problem $\in \mathcal{NP}$ are straightforward.



Refer to handout of $\mathcal{NP}$-complete problems

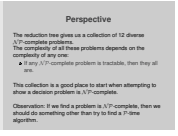
Perspective

The reduction tree gives us a collection of 12 diverse $\mathcal{NP}$-complete problems.
The complexity of all these problems depends on the complexity of any one:

- If any $\mathcal{NP}$-complete problem is tractable, then they all are.

This collection is a good place to start when attempting to show a decision problem is $\mathcal{NP}$-complete.

Observation: If we find a problem is $\mathcal{NP}$-complete, then we should do something other than try to find a $\mathcal{P}$-time algorithm.



Hundreds of problems, from many fields, have been shown to be $\mathcal{NP}$-complete.

More on this observation later.

SAT $\leq_p$ CLIQUE

(1) Easy to show CLIQUE in $\mathcal{NP}$.
(2) An instance of SAT is a Boolean expression

$$B = C_1 \cdot C_2 \cdots C_m,$$

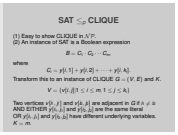
where

$$C_i = y[i,1] + y[i,2] + \cdots + y[i,k_i].$$

Transform this to an instance of CLIQUE $G = (V, E)$ and K.

$$V = \{v[i,j] | 1 \leq i \leq m, 1 \leq j \leq k_i\}$$

Two vertices $v[i_1, j_1]$ and $v[i_2, j_2]$ are adjacent in G if $i_1 \neq i_2$ AND EITHER $y[i_1, j_1]$ and $y[i_2, j_2]$ are the same literal OR $y[i_1, j_1]$ and $y[i_2, j_2]$ have different underlying variables.
$K = m$.



One vertex for each literal in B.

No join if one is the negation of the other

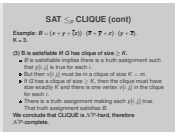
SAT $\leq_p$ CLIQUE (cont)

Example: $B = (x + y + \overline{(z)}) \cdot (\overline{x} + \overline{y} + z) \cdot (y + \overline{z})$.
K = 3.

(3) B is satisfiable iff G has clique of size $\geq K$.

- B is satisfiable implies there is a truth assignment such that $y[i, j_i]$ is true for each i.
- But then $v[i, j_i]$ must be in a clique of size $K = m$.
- If G has a clique of size $\geq K$, then the clique must have size exactly K and there is one vertex $v[i, j_i]$ in the clique for each i.
- There is a truth assignment making each $y[i, j_i]$ true. That truth assignment satisfies B.

We conclude that CLIQUE is $\mathcal{NP}$-hard, therefore $\mathcal{NP}$-complete.



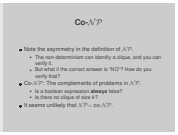
See Manber Figure 11.3.

It must connect to the other $m - 1$ literals that are also true.

No clique can have more than one member from the same clause, since there are no links between members of a clause.

Co-$\mathcal{NP}$

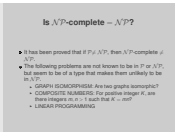
- Note the asymmetry in the definition of $\mathcal{NP}$.
  - The non-determinism can identify a clique, and you can verify it.
  - But what if the correct answer is "NO"? How do you verify that?
- Co-$\mathcal{NP}$: The complements of problems in $\mathcal{NP}$.
  - Is a boolean expression **always** false?
  - Is there no clique of size k?
- It seems unlikely that $\mathcal{NP} =$ co-$\mathcal{NP}$.



Co-$\mathcal{NP}$might be a bigger ("harder") class that includes $\mathcal{NP}$.

Is $\mathcal{NP}$-complete $= \mathcal{NP}$?

- It has been proved that if $\mathcal{P} \neq \mathcal{NP}$, then $\mathcal{NP}$-complete $\neq \mathcal{NP}$.
- The following problems are not known to be in $\mathcal{P}$ or $\mathcal{NP}$, but seem to be of a type that makes them unlikely to be in $\mathcal{NP}$.
  - GRAPH ISOMORPHISM: Are two graphs isomorphic?
  - COMPOSITE NUMBERS: For positive integer K, are there integers $m, n > 1$ such that $K = mn$?
  - LINEAR PROGRAMMING



These problems seem easier than typical $\mathcal{NP}$-complete problems, but are still probably harder than $\mathcal{P}$. They are obviously in $\mathcal{NP}$, but don't appear to be "hard" enough to solve any $\mathcal{NP}$-complete problem.

PARTITION $\leq_p$ KNAPSACK

PARTITION is a special case of KNAPSACK in which

$$K = \frac{1}{2}\sum_{a \in A} s(a)$$

assuming $\sum s(a)$ is even.

Assuming PARTITION is $\mathcal{NP}$-complete, KNAPSACK is $\mathcal{NP}$-complete.

PARTITION $\leq_p$ KNAPSACK

PARTITION is a special case of KNAPSACK in which

$K = \frac{1}{2}\sum_{a \in A} s(a)$

assuming $\sum s(a)$ is even.

Assuming PARTITION is $\mathcal{NP}$-complete, KNAPSACK is $\mathcal{NP}$-complete.

The assumption about PARITION is true, though we do not prove it.

The "transformation" is simply to pass the input of PARTITION to KNAPSACK.

"Practical" Exponential Problems

- What about our $O(KN)$ dynamic prog algorithm?
- Input size for KNAPSACK is $O(N \log K)$
  - Thus $O(KN)$ is exponential in $N \log K$.
- The dynamic programming algorithm counts through numbers $1, \cdots, K$. Takes exponential time when measured by number of bits to represent K.
- If K is "small" ($K = O(p(N))$), then algorithm has complexity polynomial in N and is truly polynomial in input size.
- An algorithm that is polynomial-time if the numbers IN the input are "small" (as opposed to number OF inputs) is called a **pseudo-polynomial** time algorithm.

"Practical" Exponential Problems

- What about our $O(KN)$ dynamic prog algorithm?
- Input size for KNAPSACK is $O(N \log K)$
  - Thus $O(KN)$ is exponential in $N \log K$.
- The dynamic programming algorithm counts through numbers $1, \cdots, K$. Takes exponential time when measured by number of bits to represent K.
- If K is "small" ($K = O(p(N))$), then algorithm has complexity polynomial in N and is truly polynomial in input size.
- An algorithm that is polynomial-time if the numbers IN the input are "small" (as opposed to number OF inputs) is called a pseudo-polynomial time algorithm.

This is an important point, about the input size. It has to do with the "size" of a number (a value). We represent the value n with $\log n$ bits, or more precisely, $\log N$ bits where N is the maximum value. In the case of KNAPSACK, K (the knapsack size) is effectively the maximum number. We will use this observation frequently when we analyze numeric algorithms.

"Practical" Problems (cont)

- Lesson: While KNAPSACK is $\mathcal{NP}$-complete, it is often not that hard.
- Many $\mathcal{NP}$-complete problems have no pseudo-polynomial time algorithm unless $\mathcal{P}=\mathcal{NP}$.

"Practical" Problems (cont)

- Lesson: While KNAPSACK is $\mathcal{NP}$-complete, it is often not that hard.
- Many $\mathcal{NP}$-complete problems have no pseudo-polynomial time algorithm unless $\mathcal{P}=\mathcal{NP}$.

The issue is what size input is practical. The problems we want to solve for Traveling Salesman are not practical.

Coping with $\mathcal{NP}$-completeness

(1) Find subproblems of the original problem that have polynomial-time algorithms.

(2) Approximation algorithms.

(3) Randomized Algorithms.

(4) Backtracking; Branch and Bound.

(5) Heuristics.
- Greedy.
- Simulated Annealing.
- Genetic Algorithms.

Coping with $\mathcal{NP}$-completeness

(1) Find subproblems of the original problem that have polynomial-time algorithms.

(2) Approximation algorithms.

(3) Randomized Algorithms.

(4) Backtracking; Branch and Bound.

(5) Heuristics.
- Greedy.
- Simulated Annealing.
- Genetic Algorithms.

The subproblems need to be "significant" special cases.

Approximation works for optimization problems (and there are a LOT of those).

Randomized Algorithms typically work well for problems with a lot of solutions.

(4) gives ways to (relatively efficiently) implement nd-choice.

Subproblems

Restrict attention to special classes of inputs.
Examples:

- VERTEX COVER, INDEPENDENT SET, and CLIQUE, when restricted to bipartite graphs, all have polynomial-time algorithms (for VERTEX COVER, by reduction to NETWORK FLOW).
- 2-SATISFIABILITY, 2-DIMENSIONAL MATCHING and EXACT COVER BY 2-SETS all have polynomial time algorithms.
- PARTITION and KNAPSACK have polynomial time algorithms if the numbers in an instance are all $O(p(n))$.
- However, HAMILTONIAN CIRCUIT and 3-COLORABILITY remain $\mathcal{NP}$-complete even for a planar graph.

Subproblems
Restrict attention to special classes of inputs.
Examples:
▲ VERTEX COVER, INDEPENDENT SET, and CLIQUE, when restricted to bipartite graphs, all have polynomial-time algorithms (for VERTEX COVER, by reduction to NETWORK FLOW).
▶ 2-SATISFIABILITY, 2-DIMENSIONAL MATCHING and EXACT COVER BY 2-SETS all have polynomial time algorithms.
▲ PARTITION and KNAPSACK have polynomial time algorithms if the numbers in an instance are all $O(p(n))$.
▶ However, HAMILTONIAN CIRCUIT and 3-COLORABILITY remain $\mathcal{NP}$-complete even for a planar graph.

Assuming the subclass covers the inputs you are interested in!

Backtracking

We may view a nondeterministic algorithm executing on a particular instance as a tree:

1. Each edge represents a particular nondeterministic choice.
2. The checking occurs at the leaves.

Example:

Each leaf represents a different set S. Checking that S is a clique of size $\geq K$ can be done in polynomial time.

Backtracking
We may view a nondeterministic algorithm executing on a particular instance as a tree:
1. Each edge represents a particular nondeterministic choice.
2. The checking occurs at the leaves.
Example:
Each leaf represents a different set S. Checking that S is a clique of size $\geq K$ can be done in polynomial time.

Example for k-CLIQUE

Need a figure here. Manber Figure 11.7 has a similar example.

Backtracking (cont)

Backtracking can be viewed as an in-order traversal of this tree with two criteria for stopping.

1. A leaf that accepts is found.
2. A partial solution that could not possibly lead to acceptance is reached.

Example:

There cannot possibly be a set S of cardinality ≥ 2 under this node, so backtrack.

Since $(1, 2) \notin E$, no S under this node can be a clique, so backtrack.

Backtracking (cont)
Backtracking can be viewed as an in-order traversal of this tree with two criteria for stopping.
1. A leaf that accepts is found.
2. A partial solution that could not possibly lead to acceptance is reached.
Example:
There cannot possibly be a set S of cardinality ≥ 2 under this node, so backtrack.
Since $(1, 2) \notin E$, no S under this node can be a clique, so backtrack.

Need Figure here.

Need Figure here.

Branch and Bound

- For optimization problems.
  More sophisticated kind of backtracking.
- Use the best solution found so far as a **bound** that controls backtracking.
- Example Problem: Given a graph G, find a minimum vertex cover of G.
- Computation tree for nondeterministic algorithm is similar to CLIQUE.
  - Every leaf represents a different subset S of the vertices.
- Whenever a leaf is reached and it contains a vertex cover of size B, B is an upper bound on the size of the minimum vertex cover.
  - Use B to prune any future tree nodes having size $\geq B$.
- Whenever a smaller vertex cover is found, update B.

Branch and Bound
▲ For optimization problems.
More sophisticated kind of backtracking.
▲ Use the best solution found so far as a bound that controls backtracking.
▲ Example Problem: Given a graph G, find a minimum vertex cover of G.
▶ Computation tree for nondeterministic algorithm is similar to CLIQUE.
- Every leaf represents a different subset S of the vertices.
◀ Whenever a leaf is reached and it contains a vertex cover of size B, B is an upper bound on the size of the minimum vertex cover.
- Use B to prune any future tree nodes having size $\geq B$.
▶ Whenever a smaller vertex cover is found, update B.

When the corresponding decision problem is $\mathcal{NP}$-complete.

Branch and Bound (cont)

- Improvement:
  - Use a fast, greedy algorithm to get a minimal (not minimum) vertex cover.
  - Use this as the initial bound B.
- While Branch and Bound is better than a brute-force exhaustive search, it is usually exponential time, hence impractical for all but the smallest instances.
  - ... if we insist on an optimal solution.
- Branch and Bound often practical as an approximation algorithm where the search terminates when a "good enough" solution is obtained.

Branch and Bound (cont)
- Improvement:
  - Use a fast, greedy algorithm to get a minimal (not minimum) vertex cover.
  - Use this as the initial bound B.
- While Branch and Bound is better than a brute-force exhaustive search, it is usually exponential time, hence impractical for all but the smallest instances.
  - ... if we insist on an optimal solution.
- Branch and Bound often practical as an approximation algorithm where the search terminates when a "good enough" solution is obtained.

no notes

Approximation Algorithms

Seek algorithms for optimization problems with a guaranteed bound on the quality of the solution.

VERTEX COVER: Given a graph $G = (V, E)$, find a vertex cover of minimum size.

Let M be a maximal (not necessarily maximum) matching in G and let V' be the set of matched vertices.
If OPT is the size of a minimum vertex cover, then

$$|V'| \leq 2\mathrm{OPT}$$

because at least one endpoint of every matched edge must be in **any** vertex cover.

Approximation Algorithms
Seek algorithms for optimization problems with a guaranteed bound on the quality of the solution.
VERTEX COVER: Given a graph $G = (V, E)$, find a vertex cover of minimum size.
Let M be a maximal (not necessarily maximum) matching in G and let V' be the set of matched vertices.
If OPT is the size of a minimum vertex cover, then
$|V'| \leq 2\mathrm{OPT}$
because at least one endpoint of every matched edge must be in **any** vertex cover.

Vertex cover: A set of vertices such that every edge is incident on at least one vertex in the set.

Then every edge will be have at least one matched vertex (i.e., vertex in the set). Thus the matching qualifies as a vertex cover.

Since a vertex of M cannot cover more than one edge of M.
In fact, we always know how far we are from a **perfect** cover (though we don't always know the size of OPT).

Bin Packing

We have numbers $x_1, x_2, \cdots, x_n$ between 0 and 1 as well as an unlimited supply of bins of size 1.

Problem: Put the numbers into as few bins as possible so that the sum of the numbers in any one bin does not exceed 1.

Example: Numbers 3/4, 1/3, 1/2, 1/8, 2/3, 1/2, 1/4.

Optimal solution: [3/4, 1/8], [1/2, 1/3], [1/2, 1/4], [2/3].

Bin Packing
We have numbers $x_1, x_2, \cdots, x_n$ between 0 and 1 as well as an unlimited supply of bins of size 1.
Problem: Put the numbers into as few bins as possible so that the sum of the numbers in any one bin does not exceed 1.
Example: Numbers 3/4, 1/3, 1/2, 1/8, 2/3, 1/2, 1/4.
Optimal solution: [3/4, 1/8], [1/2, 1/3], [1/2, 1/4], [2/3].

Optimal in that the sum is 3 1/8, and we packed into 4 bins.
There is another optimal solution with the first 3 bins packed, but this is more than we need to solve the problem.

First Fit Algorithm

Place x_1 into the first bin.

For each $i, 2 \leq i \leq n$, place x_i in the first bin that will contain it.

No more than 1 bin can be left less than half full.
The number of bins used is no more than twice the sum of the numbers.

The sum of the numbers is a lower bound on the number of bins in the optimal solution.

Therefore, first fit is no more than twice the optimal number of bins.

First Fit Algorithm
Place x_1 into the first bin.
For each $i, 2 \leq i \leq n$, place x_i in the first bin that will contain it.
No more than 1 bin can be left less than half full.
The number of bins used is no more than twice the sum of the numbers.
The sum of the numbers is a lower bound on the number of bins in the optimal solution.
Therefore, first fit is no more than twice the optimal number of bins.

Otherwise, the items in the second half-full bin would be put into the first!

First Fit Does Poorly

Let ϵ be very small, e.g., $\epsilon = .00001$.
Numbers (in this order):

- 6 of $(1/7 + \epsilon)$.
- 6 of $(1/3 + \epsilon)$.
- 6 of $(1/2 + \epsilon)$.

First fit returns:

- 1 bin of [6 of $1/7 + \epsilon$]
- 3 bins of [2 of $1/3 + \epsilon$]
- 6 bins of [$1/2 + \epsilon$]

Optimal solution is 6 bins of $[1/7 + \epsilon, 1/3 + \epsilon, 1/2 + \epsilon]$.

First fit is 5/3 larger than optimal.

no notes

Decreasing First Fit

It can be proved that the worst-case performance of first-fit is 17/10 times optimal.

Use the following heuristic:

- Sort the numbers in decreasing order.
- Apply first fit.
- This is called **decreasing first fit**.

The worst case performance of decreasing first fit is close to 11/9 times optimal.

no notes

Summary

- The theory of $\mathcal{NP}$-completeness gives us a technique for separating tractable from (probably) intractable problems.
- When faced with a new problem requiring algorithmic solution, our thought process might resemble this scheme:

Is it $\mathcal{NP}$-complete?	$\rightleftharpoons$	Is it in $\mathcal{P}$?

- Alternately think about each question. Lack of progress on either question might give insights into the answer to the other question.
- Once an affirmative answer is obtained to one of these questions, one of two strategies is followed.

no notes

Strategies

(1) The problem is in $\mathcal{P}$.

- This means there are polynomial-time algorithms for the problem, and presumably we know at least one.
- So, apply the techniques learned in this course to analyze the algorithms and improve them to find the lowest time complexity we can.

(2) The problem is $\mathcal{NP}$-complete.

- Apply the strategies for coping with $\mathcal{NP}$-completeness.
- Especially, find subproblems that are in $\mathcal{P}$, or find approximation algorithms.

That is the only way we could have proved it is in $\mathcal{P}$.

Algebraic and Numeric Algorithms

- Measuring cost of arithmetic and numerical operations:
  - Measure size of input in terms of **bits**.
- Algebraic operations:
  - Measure size of input in terms of **numbers**.
- In both cases, measure complexity in terms of basic arithmetic operations: $+, -, *, /$.
  - Sometimes, measure complexity in terms of bit operations to account for large numbers.
- Size of numbers may be related to problem size:
  - Pointers, counters to objects.
  - Resolution in geometry/graphics (to distinguish between object positions).

Algebraic and Numeric Algorithms
- Measuring cost of arithmetic and numerical operations:
  - Measure size of input in terms of bits.
- Algebraic operations:
  - Measure size of input in terms of numbers.
- In both cases, measure complexity in terms of basic arithmetic operations: + , − , ∗ , / .
  - Sometimes, measure complexity in terms of bit operations to account for large numbers.
- Size of numbers may be related to problem size:
  - Pointers, counters to objects.
  - Resolution in geometry/graphics (to distinguish between object positions).

no notes

Exponentiation

Given positive integers n and k, compute n^k.

Algorithm:

```
p = 1;
for (i=1 to k)
  p = p * n;
```

Analysis:

- Input size: $\Theta(\log n + \log k)$.
- Time complexity: $\Theta(k)$ multiplications.
- This is **exponential** in input size.

Exponentiation
Given positive integers n and k, compute n^k.
Algorithm:
```
p = 1;
for (i=1 to k)
  p = p * n;
```
Analysis:
- Input size: $\Theta(\log n + \log k)$.
- Time complexity: $\Theta(k)$ multiplications.
- This is exponential in input size.

no notes

Faster Exponentiation

Write k as:

$$k = b_t 2^t + b_{t-1} 2^{t-1} + \cdots + b_1 2 + b_0, b \in \{0, 1\}.$$

Rewrite as:

$$k = ((\cdots(b_t 2 + b_{t-1})2 + \cdots + b_2)2 + b_1)2 + b_0.$$

New algorithm:

```
p = n;
for (i = t-1 downto 0)
  p = p * p * exp(n, b[i])
```

Analysis:

- Time complexity: $\Theta(t) = \Theta(\log k)$ multiplications.
- This is **exponentially** better than before.

Faster Exponentiation
Write k as:
$k = b_t 2^t + b_{t-1} 2^{t-1} + \cdots + b_1 2 + b_0, b \in \{0, 1\}$.
Rewrite as:
$k = ((\cdots(b_t 2 + b_{t-1})2 + \cdots + b_2)2 + b_1)2 + b_0$.
New algorithm:
```
p = n;
for (i = t-1 downto 0)
  p = p * p * exp(n, b[i])
```
Analysis:
- Time complexity: $\Theta(t) = \Theta(\log k)$ multiplications.
- This is exponentially better than before.

no notes

Greatest Common Divisor

- The Greatest Common Divisor (GCD) of two integers is the greatest integer that divides both evenly.
- Observation: If k divides n and m, then k divides $n - m$.
- So,

$$f(n, m) = f(n - m, n) = f(m, n - m) = f(m, n).$$

- Observation: There exists k and l such that

$$\begin{aligned} n &= km + l \text{ where } m > l \geq 0. \\ n &= \lfloor n/m \rfloor m + n \mod m. \end{aligned}$$

- So,

$$f(n, m) = f(m, l) = f(m, n \mod m).$$

Greatest Common Divisor
- The Greatest Common Divisor (GCD) of two integers is the greatest integer that divides both evenly.
- Observation: If k divides n and m, then k divides n − m.
- So,
$f(n, m) = f(n - m, n) = f(m, n - m) = f(m, n)$.
- Observation: There exists k and l such that
$n = km + l$ where $m > l \geq 0$.
$n = \lfloor n/m \rfloor m + n \mod m$.
- So,
$f(n, m) = f(m, l) = f(m, n \mod m)$.

Assuming $n > m$, then $n = ak$, $m = bk$, $n - m = (a - b)k$ for integers a, b.

This comes from definition of mod .

GCD Algorithm

$$f(n,m) = \begin{cases} n & m = 0 \\ f(m, n \bmod m) & m > 0 \end{cases}$$

```
int LCF(int n, int m) {
  if (m == 0) return n;
  return LCF(m, n % m);
}
```


GCD Algorithm

$$f(n,m) = \begin{cases} n & m = 0 \\ f(m, n \bmod m) & m > 0 \end{cases}$$

```
int LCF(int n, int m) {
  if (m == 0) return n;
  return LCF(m, n % m);
}
```

no notes

Analysis of GCD

- How big is n mod m relative to n?

$$\begin{aligned} n \geq m &\Rightarrow n/m \geq 1 \\ &\Rightarrow 2\lfloor n/m \rfloor > n/m \\ &\Rightarrow m\lfloor n/m \rfloor > n/2 \\ &\Rightarrow n - n/2 < n - m\lfloor n/m \rfloor = n \bmod m \\ &\Rightarrow n/2 > n \bmod m \end{aligned}$$

- The first argument must be halved in no more than 2 iterations.
- Total cost:

Analysis of GCD

- How big is n mod m relative to n?

$$\begin{aligned} n \geq m &\Rightarrow n/m \geq 1 \\ &\Rightarrow 2\lfloor n/m \rfloor > n/m \\ &\Rightarrow m\lfloor n/m \rfloor > n/2 \\ &\Rightarrow n - n/2 < n - m\lfloor n/m \rfloor = n \bmod m \\ &\Rightarrow n/2 > n \bmod m \end{aligned}$$

- The first argument must be halved in no more than 2 iterations.
- Total cost:

Can split in half $\log n$ times. So $2 \log n$ is upper bound.

Note that this is linier on problem size, since problem size is $2 \log n$ (2 numbers).

Multiplying Polynomials (1)

$$P = \sum_{i=0}^{n-1} p_i x^i \qquad Q = \sum_{i=0}^{n-1} q_i x^i.$$

- Our normal algorithm for computing PQ requires $\Theta(n^2)$ multiplications and additions.

Multiplying Polynomials (1)

$$P = \sum_{i=0}^{n-1} p_i x^i \qquad Q = \sum_{i=0}^{n-1} q_i x^i.$$

- Our normal algorithm for computing PQ requires $\Theta(n^2)$ multiplications and additions.

no notes

Multiplying Polynomials (2)

- Divide and Conquer:

$$P_1 = \sum_{i=0}^{n/2-1} p_i x^i \qquad P_2 = \sum_{i=n/2}^{n-1} p_i x^{i-n/2}$$

$$Q_1 = \sum_{i=0}^{n/2-1} q_i x^i \qquad Q_2 = \sum_{i=n/2}^{n-1} q_i x^{i-n/2}$$

$$\begin{aligned} PQ &= (P_1 + x^{n/2} P_2)(Q_1 + x^{n/2} Q_2) \\ &= P_1 Q_1 + x^{n/2}(Q_1 P_2 + P_1 Q_2) + x^n P_2 Q_2. \end{aligned}$$

- Recurrence:

$$\begin{aligned} T(n) &= 4T(n/2) + O(n). \\ T(n) &= \Theta(n^2). \end{aligned}$$

Multiplying Polynomials (2)

- Divide and Conquer:

$$P_1 = \sum_{i=0}^{n/2-1} p_i x^i \qquad P_2 = \sum_{i=n/2}^{n-1} p_i x^{i-n/2}$$

$$Q_1 = \sum_{i=0}^{n/2-1} q_i x^i \qquad Q_2 = \sum_{i=n/2}^{n-1} q_i x^{i-n/2}$$

$$\begin{aligned} PQ &= (P_1 + x^{n/2} P_2)(Q_1 + x^{n/2} Q_2) \\ &= P_1 Q_1 + x^{n/2}(Q_1 P_2 + P_1 Q_2) + x^n P_2 Q_2 \end{aligned}$$

- Recurrence:

$$\begin{aligned} T(n) &= 4T(n/2) + O(n). \\ T(n) &= \Theta(n^2). \end{aligned}$$

Do this to make the subproblems look the same.

Multiplying Polynomials (3)

Observation:

$$(P_1 + P_2)(Q_1 + Q_2) = P_1Q_1 + (Q_1P_2 + P_1Q_2) + P_2Q_2$$
$$(Q_1P_2 + P_1Q_2) = (P_1 + P_2)(Q_1 + Q_2) - P_1Q_1 - P_2Q_2$$

Therefore, PQ can be calculated with only 3 recursive calls to a polynomial multiplication procedure.

Recurrence:

$$\begin{aligned} T(n) &= 3T(n/2) + O(n) \\ &= aT(n/b) + cn^1. \end{aligned}$$

$\log_b a = log_2 3 \approx 1.59$.
$T(n) = \Theta(n^{1.59})$.

Multiplying Polynomials (3)
Observation:
$(P_1 + P_2)(Q_1 + Q_2) = P_1Q_1 + (Q_1P_2 + P_1Q_2) + P_2Q_2$
$(Q_1P_2 + P_1Q_2) = (P_1 + P_2)(Q_1 + Q_2) - P_1Q_1 - P_2Q_2$
Therefore, PQ can be calculated with only 3 recursive calls to a polynomial multiplication procedure.
Recurrence:
$T(n) = 3T(n/2) + O(n)$
$= aT(n/b) + cn^1.$
$\log_b a = log_2 3 \approx 1.59$.
$T(n) = \Theta(n^{1.59})$.

In the second equation, the sums in the first term are half the original problem size, and the second two terms were needed for the first equation.

$$PQ = P_1Q_1 + X^{n/2}((P_1 + P_2)(Q_1 + Q_2) - P_1Q_1 - P_2Q_2) + x^nP_2Q_2$$

A significant improvement came from algebraic manipulation to express the product in terms of 3, rather than 4, smaller products.

Matrix Multiplication

Given: $n \times n$ matrices A and B.

Compute: $C = A \times B$.

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}.$$

Straightforward algorithm:

- $\Theta(n^3)$ multiplications and additions.

Lower bound for any matrix multiplication algorithm: $\Omega(n^2)$.

Matrix Multiplication
Given: $n \times n$ matrices A and B.
Compute: $C = A \times B$.
$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$
Straightforward algorithm:
▲ $\Theta(n^3)$ multiplications and additions.
Lower bound for any matrix multiplication algorithm: $\Omega(n^2)$.

no notes

Strassen's Algorithm

(1) Trade more additions/subtractions for fewer multiplications in 2×2 case.

(2) Divide and conquer.

In the straightforward implementation, 2×2 case is:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

Requires 8 multiplications and 4 additions.

Strassen's Algorithm
(1) Trade more additions/subtractions for fewer multiplications in 2×2 case.
(2) Divide and conquer.
In the straightforward implementation, 2×2 case is:
$c_{11} = a_{11}b_{11} + a_{12}b_{21}$
$c_{12} = a_{11}b_{12} + a_{12}b_{22}$
$c_{21} = a_{21}b_{11} + a_{22}b_{21}$
$c_{22} = a_{21}b_{12} + a_{22}b_{22}$
Requires 8 multiplications and 4 additions.

no notes

Another Approach (1)

Compute:

$$\begin{aligned} m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\ m_4 &= (a_{11} + a_{12})b_{22} \\ m_5 &= a_{11}(b_{12} - b_{22}) \\ m_6 &= a_{22}(b_{21} - b_{11}) \\ m_7 &= (a_{21} + a_{22})b_{11} \end{aligned}$$

Another Approach (1)
Compute:
$m_1 = (a_{12} - a_{22})(b_{21} + b_{22})$
$m_2 = (a_{11} + a_{22})(b_{11} + b_{22})$
$m_3 = (a_{11} - a_{21})(b_{11} + b_{12})$
$m_4 = (a_{11} + a_{12})b_{22}$
$m_5 = a_{11}(b_{12} - b_{22})$
$m_6 = a_{22}(b_{21} - b_{11})$
$m_7 = (a_{21} + a_{22})b_{11}$

no notes

Another Approach (2)

Then:

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6 \\ c_{12} &= m_4 + m_5 \\ c_{21} &= m_6 + m_7 \\ c_{22} &= m_2 - m_3 + m_5 - m_7 \end{aligned}$$

7 multiplications and 18 additions/subtractions.

Another Approach (2)

Then:

$c_{11} = m_1 + m_2 - m_4 + m_6$
$c_{12} = m_4 + m_5$
$c_{21} = m_6 + m_7$
$c_{22} = m_2 - m_3 + m_5 - m_7$

7 multiplications and 18 additions/subtractions.

$$\begin{aligned} c_{12} &= m_4 + m_5 \\ &= (a_{11} + a_{12})b_{22} + a_{11}(b_{12} + b_{22}) \\ &= a_{11}b_{22} + a_{11}b_{12} - a_{11}b_{22} \\ &= a_{12}b_{22} + b_{11}b_{12} \end{aligned}$$

Strassen's Algorithm (cont)

Divide and conquer step:

Assume n is a power of 2.

Express $C = A \times B$ in terms of $\frac{n}{2} \times \frac{n}{2}$ matrices.

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Strassen's Algorithm (cont)

Divide and conquer step:

Assume n is a power of 2.

Express $C = A \times B$ in terms of $\frac{n}{2} \times \frac{n}{2}$ matrices.

$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$

no notes

Strassen's Algorithm (cont)

By Strassen's algorithm, this can be computed with 7 multiplications and 18 additions/subtractions of $n/2 \times n/2$ matrices.

Recurrence:

$$\begin{aligned} T(n) &= 7T(n/2) + 18(n/2)^2 \\ T(n) &= \Theta(n^{\log_2 7}) = \Theta(n^{2.81}). \end{aligned}$$

Current "fastest" algorithm is $\Theta(n^{2.376})$
Open question: Can matrix multiplication be done in $O(n^2)$ time?

Strassen's Algorithm (cont)

By Strassen's algorithm, this can be computed with 7 multiplications and 18 additions/subtractions of $n/2 \times n/2$ matrices.

Recurrence:

$T(n) = 7T(n/2) + 18(n/2)^2$
$T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.81})$.

Current "fastest" algorithm is $\Theta(n^{2.376})$
Open question: Can matrix multiplication be done in $O(n^2)$ time?

But, this has a high constant due to the additions. This makes it rather impractical in real applications.

But this "fastest" algorithm is even more impractical due to overhead.

Introduction to the Sliderule

Compared to addition, multiplication is hard.

In the physical world, addition is merely concatenating two lengths.

Observation:

$$\log nm = \log n + \log m.$$

Therefore,

$$nm = \text{ antilog}(\log n + \log m).$$

What if taking logs and antilogs were easy?

Introduction to the Sliderule

Compared to addition, multiplication is hard.

In the physical world, addition is merely concatenating two lengths.

Observation:

$\log nm = \log n + \log m$.

Therefore,

$nm =$ antilog$(\log n + \log m)$.

What if taking logs and antilogs were easy?

no notes

Introduction to the Sliderule (2)

The sliderule does exactly this!

- It is essentially two rulers in log scale.
- Slide the scales to add the lengths of the two numbers (in log form).
- The third scale shows the value for the total length.

Introduction to the Sliderule (2)

The sliderule does exactly this!
- It is essentially two rulers in log scale.
- Slide the scales to add the lengths of the two numbers (in log form).
- The third scale shows the value for the total length.

This is an example of a transform. We do transforms to convert a hard problem into a (relatively) easy problem.

Representing Polynomials

A vector **a** of n values can uniquely represent a polynomial of degree $n-1$

$$P_{\mathbf{a}}(x) = \sum_{i=0}^{n-1} \mathbf{a}_i x^i.$$

Alternatively, a degree $n-1$ polynomial can be uniquely represented by a list of its values at n distinct points.

- Finding the value for a polynomial at a given point is called **evaluation**.
- Finding the coefficients for the polynomial given the values at n points is called **interpolation**.

Representing Polynomials

A vector **a** of n values can uniquely represent a polynomial of degree $n-1$

$$P_{\mathbf{a}}(x) = \sum_{i=0}^{n-1} \mathbf{a}_i x^i.$$

Alternatively, a degree $n-1$ polynomial can be uniquely represented by a list of its values at n distinct points.
- Finding the value for a polynomial at a given point is called evaluation.
- Finding the coefficients for the polynomial given the values at n points is called interpolation.

That is, a polynomial can be represented by it coefficients.

Multiplication of Polynomials

To multiply two $n-1$-degree polynomials A and B normally takes $\Theta(n^2)$ coefficient multiplications.

However, if we evaluate both polynomials, we can simply multiply the corresponding pairs of values to get the values of polynomial AB.

Process:

- Evaluate polynomials A and B at enough points.
- Pairwise multiplications of resulting values.
- Interpolation of resulting values.

Multiplication of Polynomials

To multiply two $n-1$-degree polynomials A and B normally takes $\Theta(n^2)$ coefficient multiplications.

However, if we evaluate both polynomials, we can simply multiply the corresponding pairs of values to get the values of polynomial AB.

Process:
- Evaluate polynomials A and B at enough points.
- Pairwise multiplications of resulting values.
- Interpolation of resulting values.

no notes

Multiplication of Polynomials (2)

This can be faster than $\Theta(n^2)$ IF a fast way can be found to do evaluation/interpolation of $2n-1$ points (normally this takes $\Theta(n^2)$ time).

Note that evaluating a polynomial at 0 is easy, and that if we evaluate at 1 and -1, we can share a lot of the work between the two evaluations.

Can we find enough such points to make the process cheap?

Multiplication of Polynomials (2)

This can be faster than $\Theta(n^2)$ IF a fast way can be found to do evaluation/interpolation of $2n-1$ points (normally this takes $\Theta(n^2)$ time).

Note that evaluating a polynomial at 0 is easy, and that if we evaluate at 1 and -1, we can share a lot of the work between the two evaluations.

Can we find enough such points to make the process cheap?

no notes

An Example

Polynomial A: $x^2 + 1$.
Polynomial B: $2x^2 - x + 1$.
Polynomial AB: $2x^4 - x^3 + 3x^2 - x + 1$.

Notice:

$$\begin{aligned} AB(-1) &= (2)(4) = 8 \\ AB(0) &= (1)(1) = 1 \\ AB(1) &= (2)(2) = 4 \end{aligned}$$

But: We need 5 points to nail down Polynomial AB. And, we also need to interpolate the 5 values to get the coefficients back.

An Example

Polynomial A: $x^2 + 1$.
Polynomial B: $2x^2 - x + 1$.
Polynomial AB: $2x^4 - x^3 + 3x^2 - x + 1$.

Notice:

$$\begin{aligned} AB(-1) &= (2)(4) = 8 \\ AB(0) &= (1)(1) = 1 \\ AB(1) &= (2)(2) = 4 \end{aligned}$$

But: We need 5 points to nail down Polynomial AB. And, we also need to interpolate the 5 values to get the coefficients back.

	-1	0	1
A	2	1	2
B	4	1	2
AB	8	1	4

Nth Root of Unity

The key to fast polynomial multiplication is finding the right points to use for evaluation/interpolation to make the process efficient.

Complex number ω is a **primitive nth root of unity** if

1. $\omega^n = 1$ and
2. $\omega^k \neq 1$ for $0 < k < n$.

$\omega^0, \omega^1, ..., \omega^{n-1}$ are the **nth roots of unity**.

Example:

- For $n = 4$, $\omega = i$ or $\omega = -i$.

Nth Root of Unity

The key to fast polynomial multiplication is finding the right points to use for evaluation/interpolation to make the process efficient.

Complex number ω is a **primitive nth root of unity** if

1. $\omega^n = 1$ and
2. $\omega^k \neq 1$ for $0 < k < n$.

$\omega^0, \omega^1, ..., \omega^{n-1}$ are the **nth roots of unity**.

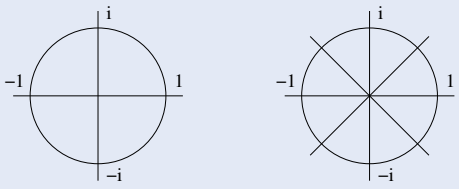
Example:

- For $n = 4$, $\omega = i$ or $\omega = -i$.

For the first circle, $n = 4, \omega = i$.

For the second circle, $n = 8, \omega = \sqrt{i}$.

Nth Root of Unity (cont)



$n = 4, \omega = i$.
$n = 8, \omega = \sqrt{i}$.

Nth Root of Unity (cont)

$n = 4, \omega = i$.
$n = 8, \omega = \sqrt{i}$.

no notes

Discrete Fourier Transform

Define an $n \times n$ matrix $V(\omega)$ with row i and column j as

$$V(\omega) = (\omega^{ij}).$$

Example: $n = 4$, $\omega = i$:

$$V(\omega) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Let $\overline{a} = [a_0, a_1, ..., a_{n-1}]^T$ be a vector.
The **Discrete Fourier Transform** (DFT) of $\overline{a}$ is:

$$F_\omega = V(\omega)\overline{a} = \overline{v}.$$

This is equivalent to evaluating the polynomial at the nth roots of unity.

Discrete Fourier Transform

Define an $n \times n$ matrix $V(\omega)$ with row i and column j as

$$V(\omega) = (\omega^{ij}).$$

Example: $n = 4$, $\omega = i$:

$$V(\omega) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Let $\overline{a} = [a_0, a_1, ..., a_{n-1}]^T$ be a vector.
The **Discrete Fourier Transform** (DFT) of $\overline{a}$ is:

$$F_\omega = V(\omega)\overline{a} = \overline{v}$$

This is equivalent to evaluating the polynomial at the nth roots of unity.

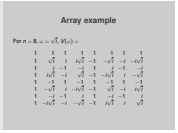
In the array, indexing begins with 0.

Example:
$1 + 2x + 3x^2 + 4x^3$
Values to evaluate at: $1, i, -1, -i$.

Array example

For $n = 8$, $\omega = \sqrt{i}$, $V(\omega) =$

$$\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & \sqrt{i} & i & i\sqrt{i} & -1 & -\sqrt{i} & -i & -i\sqrt{i} \\
1 & i & -1 & -i & 1 & i & -1 & -i \\
1 & i\sqrt{i} & -i & \sqrt{i} & -1 & -i\sqrt{i} & i & -\sqrt{i} \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & -\sqrt{i} & i & -i\sqrt{i} & -1 & \sqrt{i} & -i & i\sqrt{i} \\
1 & -i & -1 & i & 1 & -i & -1 & i \\
1 & -i\sqrt{i} & -i & -\sqrt{i} & -1 & i\sqrt{i} & i & \sqrt{i}
\end{array}$$



no notes

Inverse Fourier Transform

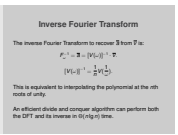
The inverse Fourier Transform to recover $\overline{a}$ from $\overline{v}$ is:

$$F_\omega^{-1} = \overline{a} = [V(\omega)]^{-1} \cdot \overline{v}.$$

$$[V(\omega)]^{-1} = \frac{1}{n} V(\frac{1}{\omega}).$$

This is equivalent to interpolating the polynomial at the nth roots of unity.

An efficient divide and conquer algorithm can perform both the DFT and its inverse in $\Theta(n \lg n)$ time.



Just replace each ω with $1/\omega$

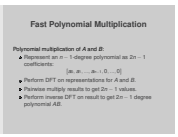
After substituting $1/\omega$ for ω.

Observe the sharable parts in the matrix.

Fast Polynomial Multiplication

Polynomial multiplication of A and B:

- Represent an $n-1$-degree polynomial as $2n-1$ coefficients:
$$[a_0, a_1, ..., a_{n-1}, 0, ..., 0]$$
- Perform DFT on representations for A and B.
- Pairwise multiply results to get $2n-1$ values.
- Perform inverse DFT on result to get $2n-1$ degree polynomial AB.



$\Theta(n \log n)$

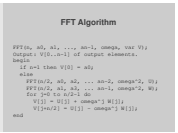
$\Theta(n)$

$\Theta(n \log n)$

Total time: $\Theta(n \log n)$.

FFT Algorithm

```
FFT(n, a0, a1, ..., an-1, omega, var V);
Output: V[0..n-1] of output elements.
begin
  if n=1 then V[0] = a0;
  else
    FFT(n/2, a0, a2, ... an-2, omega^2, U);
    FFT(n/2, a1, a3, ... an-1, omega^2, W);
    for j=0 to n/2-1 do
      V[j] = U[j] + omega^j W[j];
      V[j+n/2] = U[j] - omega^j W[j];
end
```

no notes

Parallel Algorithms

- **Running time**: $T(n, p)$ where n is the problem size, p is number of processors.
- **Speedup**: $S(p) = T(n, 1)/T(n, p)$.
  - A comparison of the time for a (good) sequential algorithm vs. the parallel algorithm in question.
- Problem: Best sequential algorithm might not be the same as the best algorithm for p processors, which might not be the best for ∞ processors.
- Efficiency: $E(n, p) = S(p)/p = T(n, 1)/(pT(n, p))$.
- Ratio of the time taken for 1 processor vs. the total time required for p processors.
  - Measure of how much the p processors are used (not wasted).
  - Optimal efficiency $= 1 =$ speedup by factor of p.

Parallel Algorithms
- Running time: $T(n, p)$ where n is the problem size, p is number of processors.
- Speedup: $S(p) = T(n, 1)/T(n, p)$.
  - A comparison of the time for a (good) sequential algorithm vs. the parallel algorithm in question.
- Problem: Best sequential algorithm might not be the same as the best algorithm for p processors, which might not be the best for ∞ processors.
- Efficiency: $E(n, p) = S(p)/p = T(n, 1)/(pT(n, p))$.
- Ratio of the time taken for 1 processor vs. the total time required for p processors.
  - Measure of how much the p processors are used (not wasted).
  - Optimal efficiency $= 1 =$ speedup by factor of p.

As opposed to $T(n)$ for sequential algorithms.

Question: What algorithms should be compared?

$pT(n, p)$ is total amount of "processor power" put into the problem.

If $E(n, p) > 1$ then the sequential form of the parallel algorithm would be faster than the sequential algorithm being compared against – very suspicious!

So there are differing goals possible: Absolute fastest speedup vs. efficiency.

Parallel Algorithm Design

Approach (1): Pick p and write best algorithm.
- Would need a new algorithm for every p!

Approach (2): Pick best algorithm for $p = \infty$, then convert to run on p processors.

Hopefully, if $T(n, p) = X$, then $T(n, p/k) \approx kX$ for $k > 1$.

Using one processor to **emulate** k processors is called the **parallelism folding principle**.

Parallel Algorithm Design
Approach (1): Pick p and write best algorithm.
- Would need a new algorithm for every p!

Approach (2): Pick best algorithm for $p = \infty$, then convert to run on p processors.

Hopefully, if $T(n, p) = X$, then $T(n, p/k) \approx kX$ for $k > 1$.

Using one processor to **emulate** k processors is called the **parallelism folding principle**.

no notes

Parallel Algorithm Design (2)

Some algorithms are only good for a large number of processors.

$$\begin{aligned} T(n, 1) &= n \\ T(n, n) &= \log n \\ S(n) &= n/\log n \\ E(n, n) &= 1/\log n \end{aligned}$$

For $p = 256$, $n = 1024$.
$T(1024, 256) = 4 \log 1024 = 40$.
For $p = 16$, running time $= (1024/16) * \log 1024 = 640$.
Speedup < 2, efficiency $= 1024/(16 * 640) = 1/10$.

Parallel Algorithm Design (2)
Some algorithms are only good for a large number of processors.

$$\begin{aligned} T(n, 1) &= n \\ T(n, n) &= \log n \\ S(n) &= n/\log n \\ E(n, n) &= 1/\log n \end{aligned}$$

For $p = 256$, $n = 1024$.
$T(1024, 256) = 4 \log 1024 = 40$.
For $p = 16$, running time $= (1024/16) * \log 1024 = 640$.
Speedup < 2, efficiency $= 1024/(16 * 640) = 1/10$.

Good in terms of speedup.

$1024/256$, assuming one processor emulates 4 in 4 times the time.
$E(1024, 256) = 1024/(256 * 40) = 1/10$.

But note that efficiency goes down as the problem size grows.

Amdahl's Law

Think of an algorithm as having a **parallelizable** section and a **serial** section.

Example: 100 operations.
- 80 can be done in parallel, 20 must be done in sequence.

Then, the best speedup possible leaves the 20 in sequence, or a speedup of $100/20 = 5$.

Amdahl's law:

$$\begin{aligned} \text{Speedup} &= (\mathcal{S} + \mathcal{P})/(\mathcal{S} + \mathcal{P}/N) \\ &= 1/(\mathcal{S} + \mathcal{P}/N) \leq 1/\mathcal{S}, \end{aligned}$$

for $\mathcal{S}$ = serial fraction, $\mathcal{P}$ = parallel fraction, $\mathcal{S} + \mathcal{P} = 1$.

Amdahl's Law
Think of an algorithm as having a **parallelizable** section and a **serial** section.

Example: 100 operations.
- 80 can be done in parallel, 20 must be done in sequence.

Then, the best speedup possible leaves the 20 in sequence, or a speedup of $100/20 = 5$.

Amdahl's law:

$$\begin{aligned} \text{Speedup} &= (\mathcal{S} + \mathcal{P})/(\mathcal{S} + \mathcal{P}/N) \\ &= 1/(\mathcal{S} + \mathcal{P}/N) \leq 1/\mathcal{S}, \end{aligned}$$

for $\mathcal{S}$ = serial fraction, $\mathcal{P}$ = parallel fraction, $\mathcal{S} + \mathcal{P} = 1$.

See John L. Gustafson "Reevaluating Amdahl's Law," CACM 5/88 and follow-up technical correspondance in CACM 8/89.

Speedup is Serial / Parallel.
Draw graph, speed up is Y axis, Sequential is X axis. You will see a nonlinear curve going down.

Amdahl's Law Revisited

However, this version of Amdahl's law applies to a fixed problem size.

What happens as the problem size grows?
Hopefully, $\mathcal{S} = f(n)$ with $\mathcal{S}$ shrinking as n grows.

Instead of fixing problem size, fix execution time for increasing number N processors (and thus, increasing problem size).

$$\begin{aligned} \text{Scaled Speedup} &= (\mathcal{S} + \mathcal{P} \times N)/(\mathcal{S} + \mathcal{P}) \\ &= \mathcal{S} + \mathcal{P} \times N \\ &= \mathcal{S} + (1 - \mathcal{S}) \times N \\ &= N + (1 - N) \times \mathcal{S}. \end{aligned}$$

Amdahl's Law Revisited
However, this version of Amdahl's law applies to a fixed problem size.
What happens as the problem size grows?
Hopefully, $\mathcal{S} = f(n)$ with $\mathcal{S}$ shrinking as n grows.
Instead of fixing problem size, fix execution time for increasing number N processors (and thus, increasing problem size).
Scaled Speedup $= (\mathcal{S} + \mathcal{P} \times N)/(\mathcal{S} + \mathcal{P})$
$= \mathcal{S} + \mathcal{P} \times N$
$= \mathcal{S} + (1 - \mathcal{S}) \times N$
$= N + (1 - N) \times \mathcal{S}$.

How long sequential process would take / How long for N processors.

Since $S + P = 1$ and $P = 1 - S$.

The point is that this equation drops off much less slowly in N: Graphing (sequential fraction for fixed N) vs. speedup, you get a line with slope $1 - N$.

All of this seems to assume the same algorithm for sequential and parallel. But that's OK – we want to see how much parallelism is possible for the parallel algorithm.

Models of Parallel Computation

Single Instruction Multiple Data (SIMD)
- All processors operate the same instruction in step.
- Example: Vector processor.

Pipelined Processing:
- Stream of data items, each pushed through the same sequence of several steps.

Multiple Instruction Multiple Data (MIMD)
- Processors are independent.

Models of Parallel Computation
Single Instruction Multiple Data (SIMD)
All processors operate the same instruction in step.
Example: Vector processor.
Pipelined Processing:
Stream of data items, each pushed through the same sequence of several steps.
Multiple Instruction Multiple Data (MIMD)
Processors are independent.

Vector: IBM 3090, Cray

Pipelined: Graphics coprocessor boards

MIMD: Modern clusters.

MIMD Communications (1)

Interconnection network:
- Each processor is connected to a limited number of neighbors.
- Can be modeled as (undirected) graph.
- Examples: Array, mesh, N-cube.
- It is possible for the cost of communications to dominate the algorithm (and in fact to limit parallelism).
- **Diameter**: Maximum over all pairwise distances between processors.
- Tradeoff between diameter and number of connections.

MIMD Communications (1)
Interconnection network:
Each processor is connected to a limited number of neighbors.
Can be modeled as (undirected) graph.
Examples: Array, mesh, N-cube.
It is possible for the cost of communications to dominate the algorithm (and in fact to limit parallelism).
Diameter: Maximum over all pairwise distances between processors.
Tradeoff between diameter and number of connections.

no notes

MIMD Communications (2)

Shared memory:
- Random access to global memory such that any processor can access any variable with unit cost.
- In practice, this limits number of processors.
- Exclusive Read/Exclusive Write (EREW).
- Concurrent Read/Exclusive Write (CREW).
- Concurrent Read/Concurrent Write (CRCW).

MIMD Communications (2)
Shared memory:
Random access to global memory such that any processor can access any variable with unit cost.
In practice, this limits number of processors.
Exclusive Read/Exclusive Write (EREW).
Concurrent Read/Exclusive Write (CREW).
Concurrent Read/Concurrent Write (CRCW).

no notes

Addition

Problem: Find the sum of two n-bit binary numbers.

Sequential Algorithm:

- Start at the low end, add two bits.
- If necessary, carry bit is brought forward.
- Can't do ith step until $i-1$ is complete due to uncertainty of carry bit (?).

Induction: (Going from $n-1$ to n implies a sequential algorithm)

Addition

Problem: Find the sum of two n-bit binary numbers.

Sequential Algorithm:
- Start at the low end, add two bits.
- If necessary, carry bit is brought forward.
- Can't do ith step until i − 1 is complete due to uncertainty of carry bit (?).

Induction: (Going from n − 1 to n implies a sequential algorithm)

no notes

Parallel Addition

Divide and conquer to the rescue:

- Do the sum for top and bottom halves.
- What about the carry bit?

Strengthen induction hypothesis:

- Find the sum of the two numbers **with** or **without** the carry bit.

After solving for $n/2$, we have L, L_c, R, and R_c.

Can combine pieces in constant time.

Parallel Addition

Divide and conquer to the rescue:
- Do the sum for top and bottom halves.
- What about the carry bit?

Strengthen induction hypothesis:
- Find the sum of the two numbers with or without the carry bit.

After solving for n/2, we have L, L_c, R, and R_c.

Can combine pieces in constant time.

Two possibilities: carry or not carry.

Also, for each a boolean indicating if it returns a carry.

If right has carry then
  Sum $= L_c|R$
Else
  Sum $= L|R$
If Sum has carry then
  Carry = TRUE
For Sum$_c$

Do the same using R_c since it is computing value having received carry.

Parallel Addition (2)

The $n/2$-size problems are independent.
Given enough processors,

$$T(n,n) = T(n/2,n/2) + O(1) = O(\log n).$$

We need only the EREW memory model.

Parallel Addition (2)

The n/2-size problems are independent.
Given enough processors,

$$T(n,n) = T(n/2,n/2) + O(1) = O(\log n).$$

We need only the EREW memory model.

Not $2T(n/2, n/2)$ because done in parallel!

Maximum-finding Algorithm: EREW

"Tournament" algorithm:

- Compare pairs of numbers, the "winner" advances to the next level.
- Initially, have $n/2$ pairs, so need $n/2$ processors.
- Running time is $O(\log n)$.

That is faster than the sequential algorithm, but what about efficiency?

$$E(n,n/2) \approx 1/\log n.$$

Why is the efficiency so low?

Maximum-finding Algorithm: EREW

"Tournament" algorithm:
- Compare pairs of numbers, the "winner" advances to the next level.
- Initially, have n/2 pairs, so need n/2 processors.
- Running time is O(log n).

That is faster than the sequential algorithm, but what about efficiency?

$$E(n,n/2) \approx 1/\log n$$

Why is the efficiency so low?

Since $\frac{T(n,1)}{nT(n,n)} = \frac{n}{n\log n}$

Lots of idle processors after the first round.

More Efficient EREW Algorithm

Divide the input into $n/\log n$ groups each with $\log n$ items.

Assign a group to each of $n/\log n$ processors.

Each processor finds the maximum (sequentially) in $\log n$ steps.

Now we have $n/\log n$ "winners".

Finish tournament algorithm.
$T(n, n/\log n) = O(\log n)$.
$E(n, n/\log n) = O(1)$.

More Efficient EREW Algorithm
Divide the input into $n/\log n$ groups each with $\log n$ items.
Assign a group to each of $n/\log n$ processors.
Each processor finds the maximum (sequentially) in $\log n$ steps.
Now we have $n/\log n$ "winners".
Finish tournament algorithm.
$T(n, n/\log n) = O(\log n)$.
$E(n, n/\log n) = O(1)$.

In $\log n$ time.

More Efficient EREW Algorithm (2)

But what could we do with more processors?
A parallel algorithm is **static** if the assignment of processors to actions is predefined.

- We know in advance, for each step i of the algorithm and for each processor p_j, the operation and operands p_j uses at step i.

This maximum-finding algorithm is static.

- All comparisons are pre-arranged.

More Efficient EREW Algorithm (2)
But what could we do with more processors?
A parallel algorithm is **static** if the assignment of processors to actions is predefined.
- We know in advance, for each step i of the algorithm and for each processor p_j, the operation and operands p_j uses at step i.
This maximum-finding algorithm is static.
- All comparisons are pre-arranged.

Cannot improve time past $O(\log n)$.

Doesn't depend on a specific input value.

As an analogy to help understand the concept of static: Bubblesort and Mergesort are static in this way. We always know the positions to be compared next.
In contrast, Insertion Sort is not static.

Brent's Lemma

Lemma 12.1: If there exists an EREW static algorithm with $T(n, p) \in O(t)$, such that the total number of steps (over all processors) is s, then there exists an EREW static algorithm with $T(n, s/t) \in O(t)$.

Proof:

- Let $a_i, 1 \leq i \leq t$, be the total number of steps performed by all processors in step i of the algorithm.
- $\sum_{i=1}^{t} a_i = s$.
- If $a_i \leq s/t$, then there are enough processors to perform this step without change.
- Otherwise, replace step i with $\lceil a_i/(s/t) \rceil$ steps, where the s/t processors emulate the steps taken by the original p processors.

Brent's Lemma
Lemma 12.1: If there exists an EREW static algorithm with $T(n, p) \in O(t)$, such that the total number of steps (over all processors) is s, then there exists an EREW static algorithm with $T(n, s/t) \in O(t)$.
Proof:
- Let $a_i, 1 \leq i \leq t$, be the total number of steps performed by all processors in step i of the algorithm.
- $\sum_{i=1}^{t} a_i = s$.
- If $a_i \leq s/t$, then there are enough processors to perform this step without change.
- Otherwise, replace step i with $\lceil a_i/(s/t) \rceil$ steps, where the s/t processors emulate the steps taken by the original p processors.

Note that we are using t as the actual number of steps, as well as the variable in the big-Oh analysis, which is a bit informal.

Brent's Lemma (2)

- The total number of steps is now

$$\begin{aligned}\sum_{i=1}^{t} \lceil a_i/(s/t) \rceil &\leq \sum_{i=1}^{t} (a_i t/s + 1) \\ &= t + (t/s) \sum_{i=1}^{t} a_i = 2t.\end{aligned}$$

Thus, the running time is still $O(t)$.

Intuition: You have to split the s work steps across the t time steps somehow; things can't **always** be bad!

Brent's Lemma (2)
- The total number of steps is now
$$\begin{aligned}\sum_{i=1}^{t} \lceil a_i/(s/t) \rceil &\leq \sum_{i=1}^{t} (a_i t/s + 1) \\ &= t + (t/s) \sum_{i=1}^{t} a_i = 2t.\end{aligned}$$
Thus, the running time is still $O(t)$.
Intuition: You have to split the s work steps across the t time steps somehow; things can't **always** be bad!

If s is sequential complexity, then the modified algorithm has $O(1)$ efficiency.

Maximum-finding: CRCW

- Allow concurrent writes to a variable only when each processor writes the same thing.
- Associate each element x_i with a variable v_i, initially "1".
- For each of $n(n-1)/2$ processors, processor p_{ij} compares elements i and j.
- First step: Each processor writes "0" to the v variable of the smaller element.
  - Now, only one v is "1".
- Second step: Look at all $v_i, 1 \leq i \leq n$.
  - The processor assigned to the max element writes that value to MAX.

Efficiency of this algorithm is **very** poor!

- "Divide and crush."

Maximum-finding: CRCW
- Allow concurrent writes to a variable only when each processor writes the same thing.
- Associate each element x_i with a variable v_i, initially "1".
- For each of $n(n-1)/2$ processors, processor p_{ij} compares elements i and j.
- First step: Each processor writes "0" to the v variable of the smaller element.
  - Now, only one v is "1".
- Second step: Look at all $v_i, 1 \leq i \leq n$.
  - The processor assigned to the max element writes that value to MAX.

Efficiency of this algorithm is **very** poor!
- "Divide and crush."

Need $O(n^2)$ *processors*
Need only constant time.
Efficiency is $1/n$.

Maximum-finding: CRCW (2)

More efficient (but slower) algorithm:

- Given: n processors.
- Find maximum for each of $n/2$ pairs in constant time.
- Find max for $n/8$ groups of 4 elements (using 8 proc/group) each in constant time.
- Square the group size each time.
- Total time: $O(\log \log n)$.

Maximum-finding: CRCW (2)
More efficient (but slower) algorithm:
- Given: n processors.
- Find maximum for each of $n/2$ pairs in constant time.
- Find max for $n/8$ groups of 4 elements (using 8 proc/group) each in constant time.
- Square the group size each time.
- Total time: $O(\log \log n)$.

n/2 processors
n processors, using previous "divide and crush" algorithm.

This leaves n/8 elements which can be broken into n/128 groups of 16 elements with 128 processors assigned to each group. And so on.

Efficiency is $1/\log \log n$.

Parallel Prefix

- Let $\cdot$ be any associative binary operation.
  - Ex: Addition, multiplication, minimum.
- Problem: Compute $x_1 \cdot x_2 \cdot \ldots \cdot x_k$ for all $k, 1 \leq k \leq n$.
- Define $\mathrm{PR(i, j)} = \mathrm{x_i \cdot x_{i+1} \cdot \ldots \cdot x_j}$.
  We want to compute $\mathrm{PR(1, k)}$ for $1 \leq k \leq n$.
- Sequential alg: Compute each prefix in order
  - $O(n)$ time required (using previous prefix)
- Approach: Divide and Conquer
  - IH: We know how to solve for $n/2$ elements.
  1. $\mathrm{PR(1, k)}$ and $\mathrm{PR(n/2 + 1, n/2 + k)}$ for $1 \leq k \leq n/2$.
  2. $\mathrm{PR(1, m)}$ for $n/2 < m \leq n$ comes from $\mathrm{PR(1, n/2) \cdot PR(n/2 + 1, m)}$ – from IH.

Parallel Prefix
- Let $\cdot$ be any associative binary operation.
  - Ex: Addition, multiplication, minimum.
- Problem: Compute $x_1 \cdot x_2 \cdot \ldots \cdot x_k$ for all $k, 1 \leq k \leq n$.
- Define $\mathrm{PR(i, j)} = \mathrm{x_i \cdot x_{i+1} \cdot \ldots \cdot x_j}$.
  We want to compute $\mathrm{PR(1, k)}$ for $1 \leq k \leq n$.
- Sequential alg: Compute each prefix in order
  - $O(n)$ time required (using previous prefix)
- Approach: Divide and Conquer
  - IH: We know how to solve for $n/2$ elements.
  1. $\mathrm{PR(1, k)}$ and $\mathrm{PR(n/2 + 1, n/2 + k)}$ for $1 \leq k \leq n/2$.
  2. $\mathrm{PR(1, m)}$ for $n/2 < m \leq n$ comes from $\mathrm{PR(1, n/2) \cdot PR(n/2 + 1, m)}$ – from IH.

We don't just want the sum or min of all – we want all the partials as well.

We have the lower half done, and the upper half values are each missing the contribution from the lower half.

Parallel Prefix (2)

- **Complexity**: (2) requires $n/2$ processors and CREW for parallelism (all read middle position).
- $T(n, n) = O(\log n)$; $E(n, n) = O(1/\log n)$.
  Brent's lemma no help: $O(n \log n)$ total steps.

Parallel Prefix (2)
- **Complexity**: (2) requires $n/2$ processors and CREW for parallelism (all read middle position).
- $T(n, n) = O(\log n)$; $E(n, n) = O(1/\log n)$.
  Brent's lemma no help: $O(n \log n)$ total steps.

That is – no processors are "excessively" idle. This is because we needed to copy PR(1, $n/2$) into $n/2$ positions on the last step.

$$E = \frac{n}{n \cdot \log n} = \frac{1}{logn}$$

Better Parallel Prefix

- E is the set of all x_is with i even.
- If we know $\mathrm{PR}(1, 2i)$ for $1 \leq i \leq n/2$ then $\mathrm{PR}(1, 2i+1) = \mathrm{PR}(1, 2i) \cdot x_{2i+1}$.
- Algorithm:
  - Compute in parallel $x_{2i} = x_{2i-1} \cdot x_{2i}$ for $1 \leq i \leq n/2$.
  - Solve for E (by induction).
  - Compute in parallel $x_{2i+1} = x_{2i} \cdot x_{2i+1}$.
- Complexity:
  $T(n, n) = O(\log n)$.
  $S(n) = S(n/2) + n - 1$, so $S(n) = O(n)$ for $S(n)$ the total number of steps required to process n elements.
- So, by Brent's Lemma, we can use $O(n/\log n)$ processors for $O(1)$ efficiency.

Better Parallel Prefix
- E is the set of all x_is with i even.
- If we know $\mathrm{PR}(1, 2i)$ for $1 \leq i \leq n/2$ then $\mathrm{PR}(1, 2i+1) = \mathrm{PR}(1, 2i) \cdot x_{2i+1}$.
- Algorithm:
  - Compute in parallel $x_{2i} = x_{2i-1} \cdot x_{2i}$ for $1 \leq i \leq n/2$.
  - Solve for E (by induction).
  - Compute in parallel $x_{2i+1} = x_{2i} \cdot x_{2i+1}$.
- Complexity:
  $T(n, n) = O(\log n)$.
  $S(n) = S(n/2) + n - 1$, so $S(n) = O(n)$ for $S(n)$ the total number of steps required to process n elements.
- So, by Brent's Lemma, we can use $O(n/\log n)$ processors for $O(1)$ efficiency.

Since the E's already include their left neighbors, all info is available to get the odds.

There is only one recursive call, instead of two in the previous algorithm.

Need EREW model for Brent's Lemma.

Routing on a Hypercube

Goal: Each processor P_i simultaneously sends a message to processor $P_{\sigma(i)}$ such that no processor is the destination for more than one message.

Problem:

- In an n-cube, each processor is connected to n other processors.
- At the same time, each processor can send (or receive) only one message per time step on a given connection.
- So, two messages cannot use the same edge at the same time – one must wait.

Routing on a Hypercube
Goal: Each processor P_i simultaneously sends a message to processor $P_{\sigma(i)}$ such that no processor is the destination for more than one message.
Problem:
- In an n-cube, each processor is connected to n other processors.
- At the same time, each processor can send (or receive) only one message per time step on a given connection.
- So, two messages cannot use the same edge at the same time – one must wait.

Need a figure

Randomizing Switching Algorithm

It can be shown that any deterministic algorithm is $\Omega(2^{n^a})$ for some $a > 0$, where 2^n is the number of messages.

A node i (and its corresponding message) has binary representation $i_1 i_2 \cdots i_n$.

Randomization approach:

(a) Route each message from i to j to a random processor r (by a randomly selected route).
(b) Continue the message from r to j by the shortest route.

Randomizing Switching Algorithm
It can be shown that any deterministic algorithm is $\Omega(2^{n^a})$ for some $a > 0$, where 2^n is the number of messages.
A node i (and its corresponding message) has binary representation $i_1 i_2 \cdots i_n$.
Randomization approach:
(a) Route each message from i to j to a random processor r (by a randomly selected route).
(b) Continue the message from r to j by the shortest route.

n-dimensional hypercube has 2^n nodes.

Remember that we want parallel algorithms with cost log n, not cost n^a!
The distance from any processor i to another processor j is only log n steps.

Randomized Switching (2)

Phase (a):

```
for (each message at i)
cobegin
  for (k = 1 to n)
    T[i, k] = RANDOM(0, 1);
  for (k = 1 to n)
    if (T[i, k] = 1)
      Transmit i along dimension k;
coend;
```

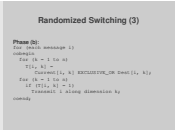

Randomized Switching (2)
Phase (a):
```
for (each message at i)
cobegin
  for (k = 1 to n)
    T[i, k] = RANDOM(0, 1);
  for (k = 1 to n)
    if (T[i, k] = 1)
      Transmit i along dimension k;
coend;
```

no notes

Randomized Switching (3)

Phase (b):

```
for (each message i)
cobegin
  for (k = 1 to n)
    T[i, k] =
       Current[i, k] EXCLUSIVE_OR Dest[i, k];
  for (k = 1 to n)
    if (T[i, k] = 1)
      Transmit i along dimension k;
coend;
```

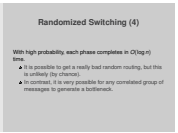



no notes

Randomized Switching (4)

With high probability, each phase completes in $O(\log n)$ time.

- It is possible to get a really bad random routing, but this is unlikely (by chance).
- In contrast, it is very possible for any correlated group of messages to generate a bottleneck.



no notes

Sorting on an array

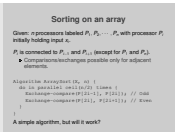
Given: n processors labeled $P_1, P_2, \cdots, P_n$ with processor P_i initially holding input x_i.

P_i is connected to P_{i-1} and P_{i+1} (except for P_1 and P_n).

- Comparisons/exchanges possible only for adjacent elements.

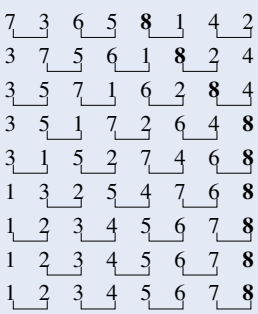
```
Algorithm ArraySort(X, n) {
  do in parallel ceil(n/2) times {
    Exchange-compare(P[2i-1], P[2i]); // Odd
    Exchange-compare(P[2i], P[2i+1]); // Even
  }
}
```

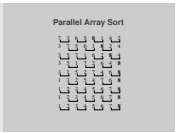
A simple algorithm, but will it work?



Any algorithm that correctly sorts 1's and 0's by comparisons will also correctly sort arbitrary numbers.

Parallel Array Sort





Manber Figure 12.8.

Correctness of Odd-Even Transpose

Theorem 12.2: When Algorithm ArraySort terminates, the numbers are sorted.

Proof: By induction on n.

Base Case: 1 or 2 elements are sorted with one comparison/exchange.

Induction Step:

- Consider the maximum element, say x_m.
- Assume m odd (if even, it just won't exchange on first step).
- This element will move one step to the right each step until it reaches the rightmost position.

Correctness of Odd-Even Transpose
Theorem 12.2: When Algorithm ArraySort terminates, the numbers are sorted.
Proof: By induction on n.
Base Case: 1 or 2 elements are sorted with one comparison/exchange.
Induction Step:
Consider the maximum element, say x_m.
Assume m odd (if even, it just won't exchange on first step).
This element will move one step to the right each step until it reaches the rightmost position.

no notes

Correctness (2)

- The position of x_m follows a diagonal in the array of element positions at each step.
- Remove this diagonal, moving comparisons in the upper triangle one step closer.
- The first row is the nth step; the right column holds the greatest value; the rest is an $n-1$ element sort (by induction).

Correctness (2)
The position of x_m follows a diagonal in the array of element positions at each step.
Remove this diagonal, moving comparisons in the upper triangle one step closer.
The first row is the nth step; the right column holds the greatest value; the rest is an $n-1$ element sort (by induction).

Map the execution of n to an execution of $n-1$ elements.

See Manber Figure 12.9.

Sorting Networks

When designing parallel algorithms, need to make the steps independent.

Ex: Mergesort split step can be done in parallel, but the join step is nearly serial.

- To parallelize mergesort, we must parallelize the merge.

Sorting Networks
When designing parallel algorithms, need to make the steps independent.
Ex: Mergesort split step can be done in parallel, but the join step is nearly serial.
To parallelize mergesort, we must parallelize the merge.

no notes

Batcher's Algorithm

For n a power of 2, assume $a_1, a_2, \cdots, a_n$ and $b_1, b_2, \cdots, b_n$ are sorted sequences.

Let $x_1, x_2, \cdots, x_{2n}$ be the final merged order.

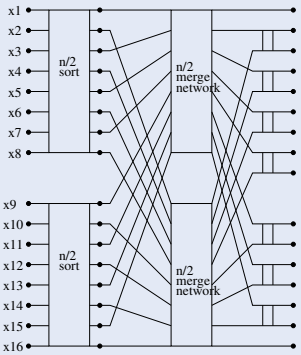
Need to merge disjoint parts of these sequences in parallel.

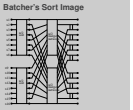
- Split a, b into odd- and even- index elements.
- Merge a_{odd} with b_{odd}, a_{even} with b_{even}, yielding $o_1, o_2, \cdots, o_n$ and $e_1, e_2, \cdots, e_n$ respectively.

Batcher's Algorithm
For n a power of 2, assume $a_1, a_2, \cdots, a_n$ and $b_1, b_2, \cdots, b_n$ are sorted sequences.
Let $x_1, x_2, \cdots, x_{2n}$ be the final merged order.
Need to merge disjoint parts of these sequences in parallel.
Split a, b into odd- and even- index elements.
Merge a_{odd} with b_{odd}, a_{even} with b_{even} yielding $o_1, o_2, \cdots, o_n$ and $e_1, e_2, \cdots, e_n$ respectively.

No notes

Batcher's Sort Image





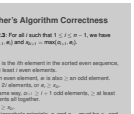
No notes

Batcher's Algorithm Correctness

Theorem 12.3: For all i such that $1 \leq i \leq n-1$, we have $x_{2i} = \min(o_{i+1}, e_i)$ and $x_{2i+1} = \max(o_{i+1}, e_i)$.

Proof:

- Since e_i is the ith element in the sorted even sequence, it is $\geq$ at least i even elements.
- For each even element, e_i is also $\geq$ an odd element.
- So, $e_i \geq 2i$ elements, or $e_i \geq x_{2i}$.
- In the same way, $o_{i+1} \geq i+1$ odd elements, $\geq$ at least $2i$ elements all together.
- So, $o_{i+1} \geq x_{2i}$.
- By the pigeonhole principle, e_i and o_{i+1} must be x_{2i} and x_{2i+1} (in either order).



See Manber Figure 12.11.

Batcher Sort Complexity

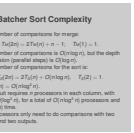
- Total number of comparisons for merge:

$$T_M(2n) = 2T_M(n) + n - 1; \quad T_M(1) = 1.$$

  Total number of comparisons is $O(n \log n)$, but the depth of recursion (parallel steps) is $O(\log n)$.
- Total number of comparisons for the sort is:

$$T_S(2n) = 2T_S(n) + O(n \log n), \quad T_S(2) = 1.$$

  So, $T_S(n) = O(n \log^2 n)$.
- The circuit requires n processors in each column, with depth $O(\log^2 n)$, for a total of $O(n \log^2 n)$ processors and $O(\log^2 n)$ time.
- The processors only need to do comparisons with two inputs and two outputs.



$\mathrm{O}(\log n)$ sort steps, with each associated merge step counting $\mathrm{O}(\log n)$.

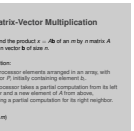
Matrix-Vector Multiplication

Problem: Find the product $x = A\mathbf{b}$ of an m by n matrix A with a column vector $\mathbf{b}$ of size n.

Systolic solution:

- Use n processor elements arranged in an array, with processor P_i initially containing element b_i.
- Each processor takes a partial computation from its left neighbor and a new element of A from above, generating a partial computation for its right neighbor.

Cost: $\mathrm{O}(n+m)$



See Manber Figure 12.17.