COURSENOTES

CS5114: Theory of Algorithms

Clifford A. Shaffer
Department of Computer Science
Virginia Tech

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

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 $\mathbf{P}(n)$:

$n + 1 \leq n^2$

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 $\rightarrow P(n)$ is true.

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

Step 1: Base case
Step 2: Induction step

The assumption "$\mathbf{P}(n-1)$ is true" is the **induction hypothesis**.

What does this remind you of?

Induction Example

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

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

Induction Example

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

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...

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.

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 $\mathbf{P}(n)$.

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 (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 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.

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 (cont)

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

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 induction hypothesis, all horses in S' have the same color.
- Let h' be some horse in S'.
- Induction hypothesis implies $\{h, h'\}$ have all the same color.

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

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.

Time Complexity

Time and space are the most important computer resources.

Function of input: $\mathbf{T}$(input)

Growth of time with size of input:

- Establish an (integer) **<u>size</u>** 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 $\mathbf{T}(n)$ counts **<u>steps</u>** in an algorithm.

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.

O Notation

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

- pronounced "Big-oh"

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

Examples:

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

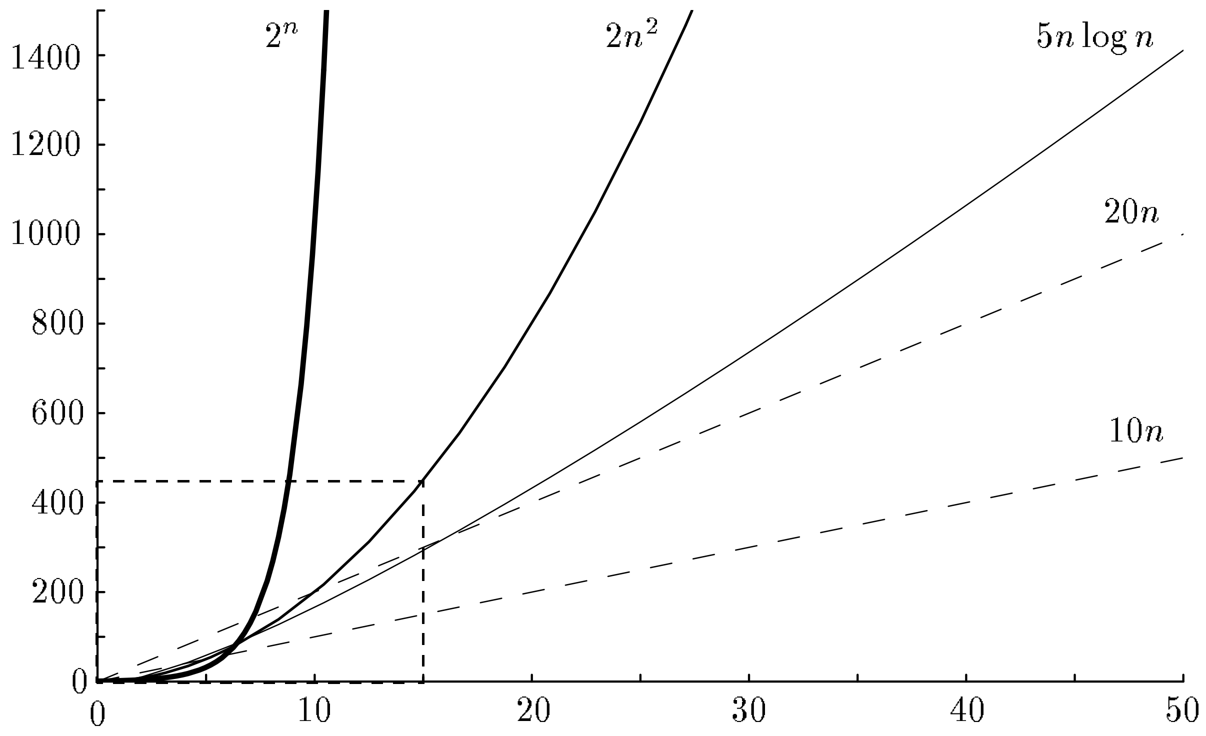
We seek the "simplest" and "strongest" f.

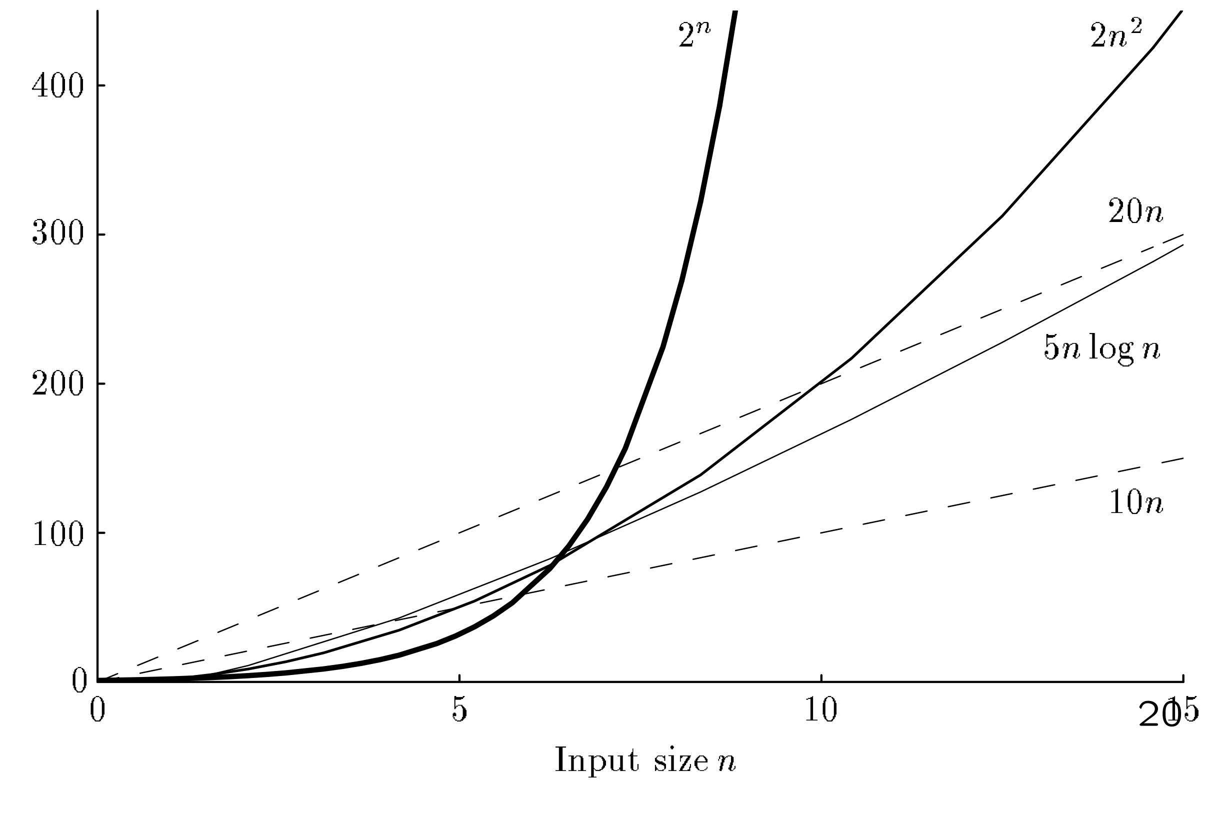
Note O is somewhat like "$\leq$":

$n^2 \in \mathrm{O}(n^3)$ 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 $n^2 \log n \notin \mathrm{O}(n^2)$

Growth Rate Graph





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).

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, $f(n) \in O(ks(n))$

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 $g(n) \in \Omega(f(n))$.

Ex: $5n^3 + 4n^2 + 9n + 7 = \Theta(n^3)$

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.

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.

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.

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.

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

Guess that $S(n) \leq n^3$.
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.

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.

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

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

Example 3.3

$$G(n) = \sum_{i=1}^{n} i2^i = 1\times2+2\times2^2+3\times2^3+\cdots+n\times2^n$$

Recurrence Relations

A function defined in terms of itself.

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.

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: $T(n) \leq E(T, n)$.

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 $T(n) \le E(T, n)$.

Methods:

- Guess a solution
- Expand recurrence
- Theorems

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 $T(n/2) \leq f(n/2)$.

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

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)

Guess again.

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

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

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

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

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

Is this the best possible upper bound?

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.

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; $n = 2^k$.

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

This it the <u>**exact**</u> solution for powers of 2.

$$T(n) = \Theta(n^2).$$

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 two solutions.

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.

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) $r = 1$

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

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

(3) $r > 1$

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

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

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

Summary

Theorem 3.4:

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

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

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:

$m_1 + m_2 \cdot n = m_1 + m_2 \cdot m_1$.

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:

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

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.

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.

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:

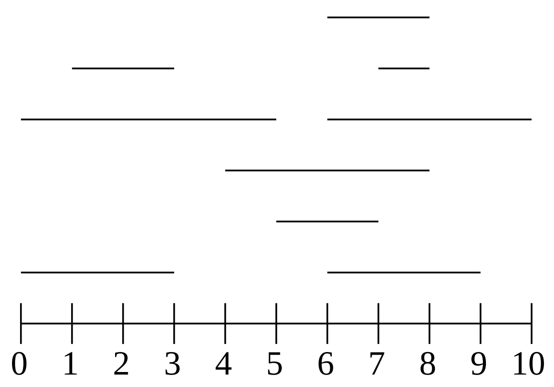
- Different inductive proofs are possible.
- We want the most **efficient** algorithm!

Interval Containment

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

Example:

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



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

Example:

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

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:

$T(n) = T(n-1) + cn$
$T(n) = \Theta(n^2)$

“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.

“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 $Q(n)$)

- Solve for first $n-1$ intervals recursively, and remember 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: $\Theta(n \log n) + \Theta(n)$.

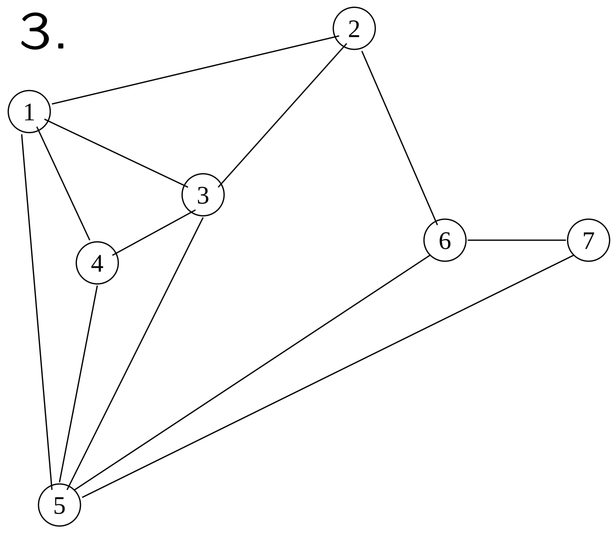
Lesson: Preprocessing, often sorting, can help sometimes.

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:

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 $\Theta(|V| + |E|)$.

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?

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?

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?

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: Every question eliminates one person as a non-celebrity.

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.

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

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

4 remains as candidate.

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

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 (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 $x_1 < 0 \Rightarrow$ 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)

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.

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) $+ x_n$).

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?

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

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

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 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.

Knapsack: New Induction

New Induction Hypothesis:

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

Resulting algorithm complexity:

$T(n) = 2T(n-1) + c \qquad n \geq 2$

$T(n) = \Theta(2^n)$ by expanding sum.

Alternate: change variables as $m = 2^n$.

$2T(m/2) + c_1 n^0$.

From Theorem 3.4, we get $\Theta(m^{\log_2 2}) = \Theta(2^n)$.

But, there are only $n(K+1)$ problems defined.

- Problems are being re-solved many times by this algorithm.

Try another hypothesis...

Improved Algorithm

Induction Hypothesis:

We know how to solve $P(i, k), 1 \leq i \leq n-1, 0 \leq k \leq K$.

```
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.
```

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 $\Theta(nK)$.

Example

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

Matrix M:

	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.

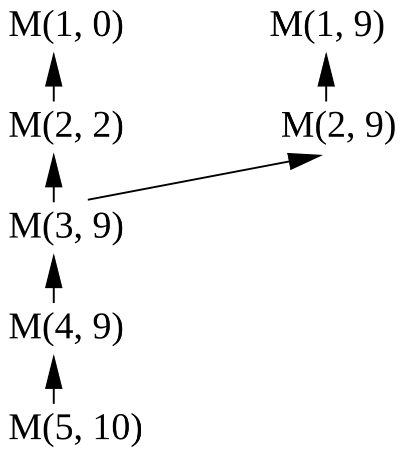
I/O Solutions for $P(i, k)$ both with i included and with i omitted.

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 $P(5, 10)$.


$$\begin{tikzcd}
M(1,0) & M(1,9) \\
M(2,2) \arrow[u] & M(2,9) \arrow[u] \\
M(3,9) \arrow[u] \arrow[ur] & \\
M(4,9) \arrow[u] & \\
M(5,10) \arrow[u] &
\end{tikzcd}$$

The result is an n-level DAG.

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**.

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

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:

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.

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

So,

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

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:

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

Ugh!

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

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 $1 \le i < j \le n$,

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

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?

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. Small number of subproblems, small amount of information to store for each subproblem.
2. Base case easy to solve.
3. Easy to solve one subproblem given solutions to smaller subproblems.

Sorting

Each record contains a field called the **<u>key</u>**.
Linear order: comparison.

The Sorting Problem

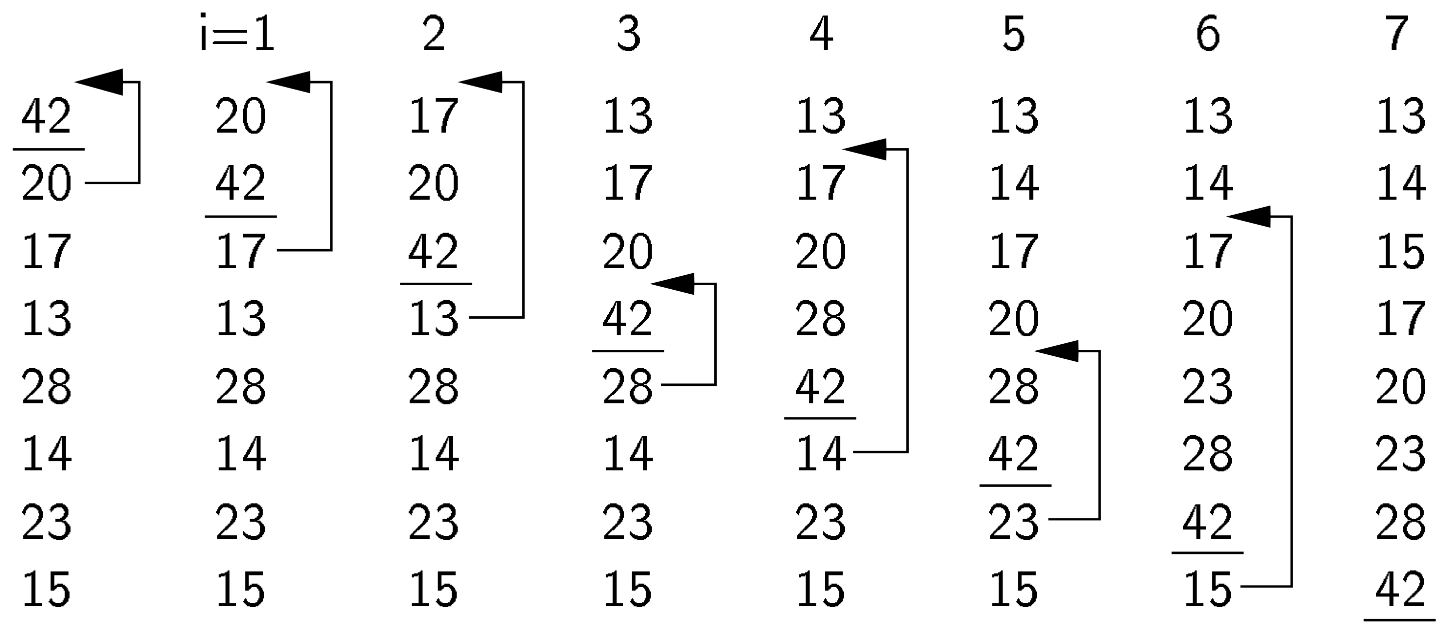
Given a sequence of records $R_1, R_2, ..., R_n$ with key values $k_1, k_2, ..., k_n$, respectively, arrange the records into any order s such that records $R_{s_1}, R_{s_2}, ..., R_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$.

Measures of cost:

- Comparisons
- Swaps

Insertion Sort

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



Best Case:

Worst Case:

Average Case:

Exchange Sorting

Theorem: Any sort restricted to swapping adjacent records must be $\Omega(n^2)$ in the worst and average cases.

Quicksort

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

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

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

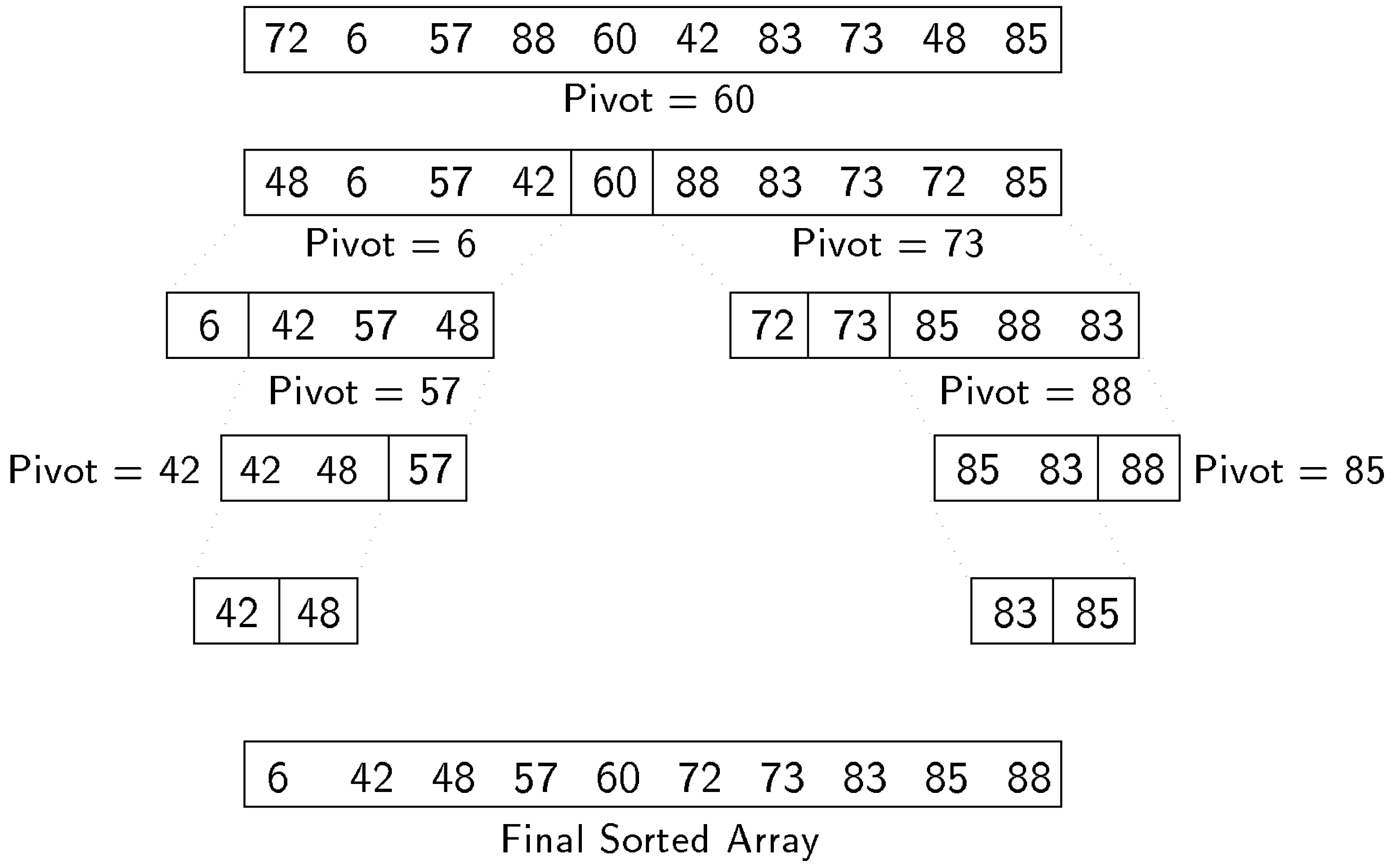
Quicksort Partition

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

```
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
                                 r    l

Swap 3            48   6    57   85   42   88   83   73   72   60
                                 r    l

Reverse Swap      48   6    57   42  |85   88   83   73   72   60
                                 r    l
```

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

Quicksort Example



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.

Quicksort Average Cost

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

Since the two halves of the summation are identical,

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

Multiplying both sides by n yields

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

Average Cost (cont.)

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

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

Average Cost (cont.)

Note that $\frac{2n}{n+1} \leq 2$ for $n \geq 1$. Expanding the recurrence, we get

$$\begin{aligned}
f(n+1) &\leq 2 + \frac{n+2}{n+1} f(n) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n} f(n-1)\right) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}\left(2 + \frac{n}{n-1} f(n-2)\right)\right) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \cdots + \frac{4}{3}(2 + \frac{3}{2} f(1))\right) \\
&= 2\left(1 + \frac{n+2}{n+1} + \frac{n+2}{n+1}\frac{n+1}{n} + \cdots \right. \\
&\qquad \left. + \frac{n+2}{n+1}\frac{n+1}{n} \cdots \frac{3}{2}\right) \\
&= 2\left(1 + (n+2)\left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2}\right)\right) \\
&= 2 + 2(n+2)\left(\mathcal{H}_{n+1} - 1\right) \\
&= \Theta(n \log n).
\end{aligned}$$

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));
}
```

$$36\quad 20\quad 17\quad 13\quad 28\quad 14\quad 23\quad 15$$

$$\boxed{20\quad 36}\ \boxed{13\quad 17}\ \boxed{14\quad 28}\ \boxed{15\quad 23}$$

$$\boxed{13\quad 17\quad 20\quad 36}\ \boxed{14\quad 15\quad 23\quad 28}$$

$$\boxed{13\quad 14\quad 15\quad 17\quad 20\quad 23\quad 28\quad 36}$$

Mergesort Implementation

Mergesort is tricky to implement.

```
void mergesort(Elem* array, Elem* temp,
               int left, int right) {
  int mid = (left+right)/2;
  if (left == right) return;        // List of one ELEM
  mergesort(array, temp, left, mid);   // Sort 1st half
  mergesort(array, temp, mid+1, right);// Sort 2nd half
  for (int i=left; i<=right; i++)      // Copy to temp
    temp[i] = array[i];
  // 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 sublist exhausted
      array[curr] = temp[i2++];
    else if (i2 > right)  // Right sublist exhausted
      array[curr] = temp[i1++];
    else if (key(temp[i1]) < key(temp[i2]))
      array[curr] = temp[i1++];
    else array[curr] = temp[i2++];
}}
```

Mergesort cost:

Mergesort is good for sorting linked lists.

Heaps

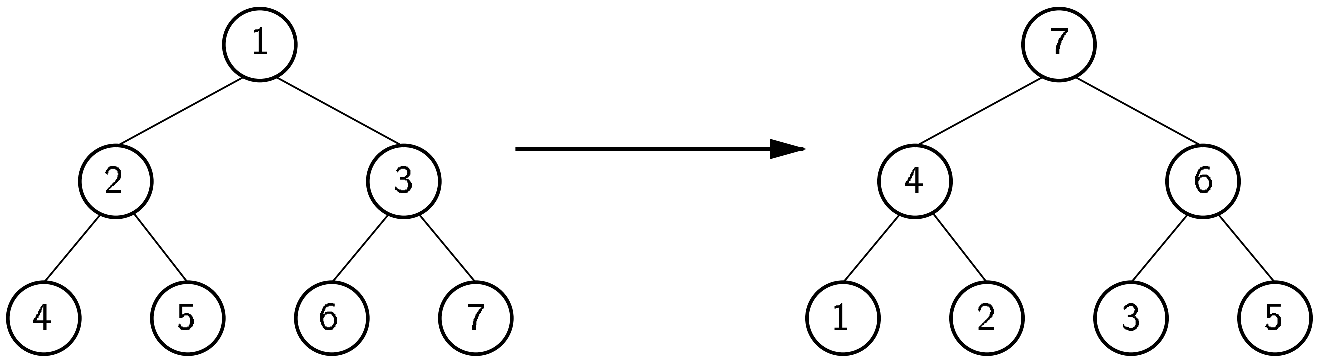
Heap: Complete binary tree with the
Heap Property:

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

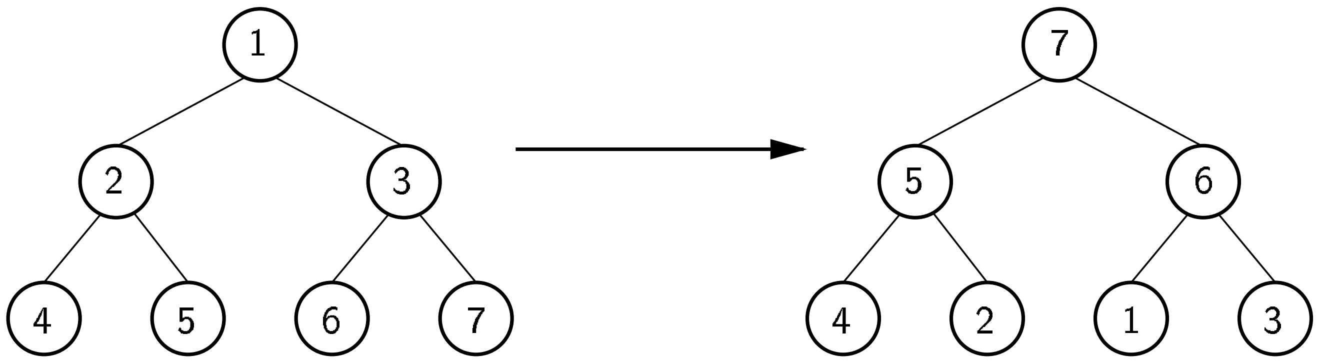
The values in a heap are **partially ordered**.

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

Building the Heap



(a)



(b)

(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).

Siftdown

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); }

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

Cost for heap construction:

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

Heapsort

Heapsort uses a max-heap.

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

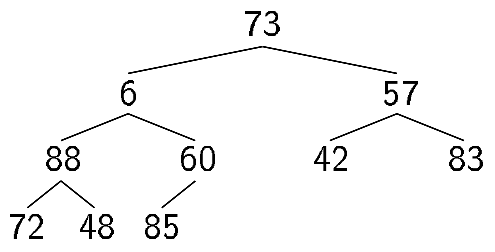
Cost of Heapsort:

Cost of finding k largest elements:

Heapsort Example

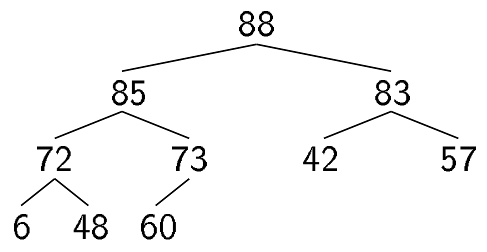
Original Numbers

73	6	57	88	60	42	83	72	48	85



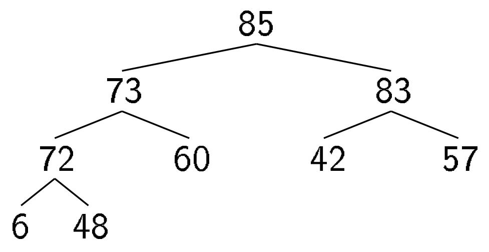
Build Heap

88	85	83	72	73	42	57	6	48	60



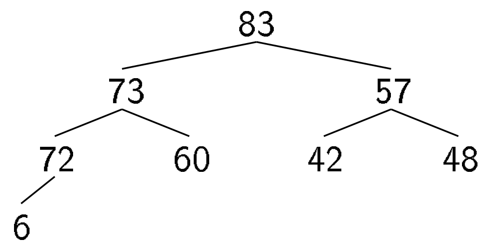
Remove 88

85	73	83	72	60	42	57	6	48	88



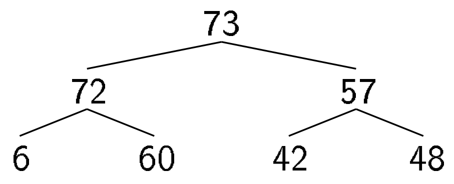
Remove 85

83	73	57	72	60	42	48	6	85	88



Remove 83

73	72	57	6	60	42	48	83	85	88



Binsort

A simple, efficient sort:

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

Ways to generalize:

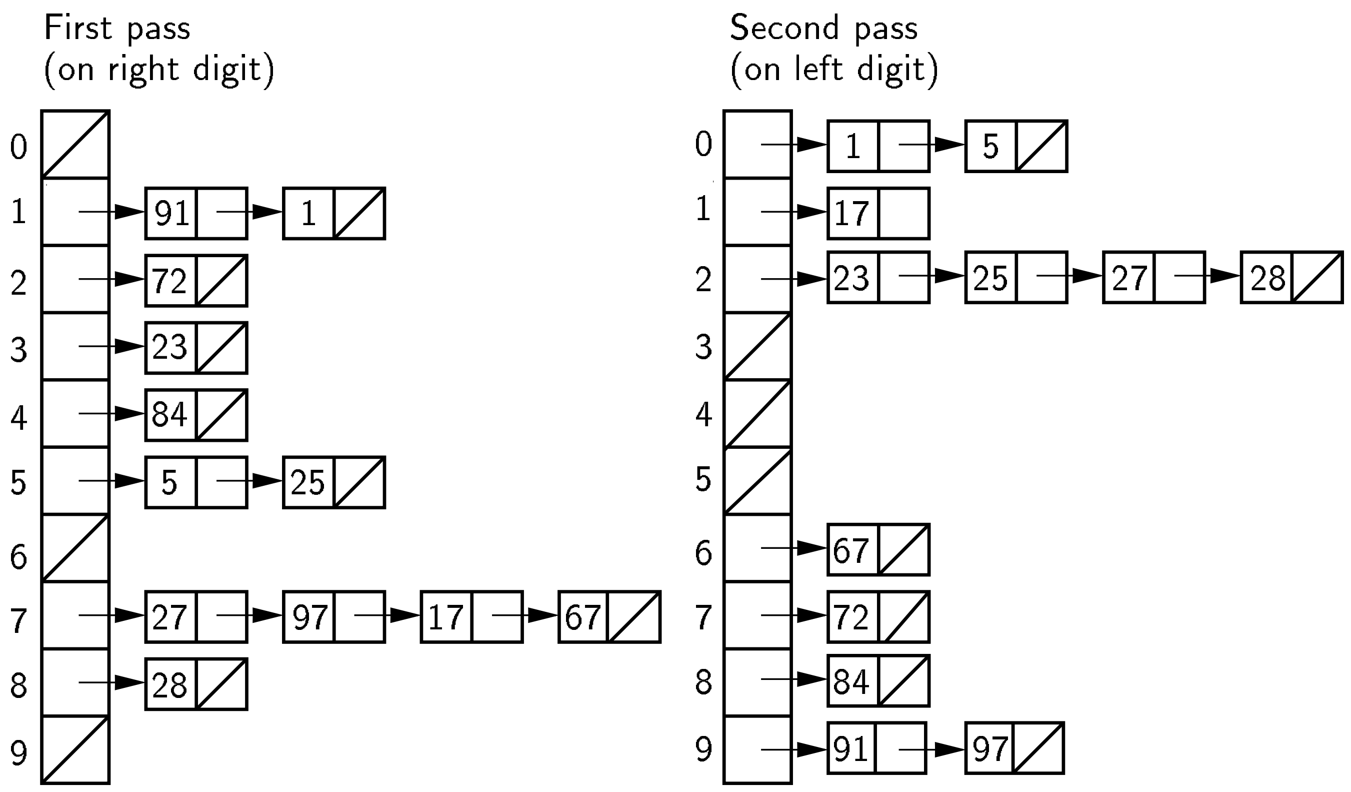
- Make each bin the head of a list.
- Allow more keys than records.

```
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 (B[i].first(); B[i].isInList(); B[i].next())
      output(B[i].currValue());
}
```

Cost:

Radix Sort

Initial List: 27 91 1 97 17 23 84 28 72 5 67 25



Result of first pass: 91 1 72 23 84 5 25 27 97 17 67 28
Result of second pass: 1 5 17 23 25 27 28 67 72 84 91 97

Cost of Radix Sort

```
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) {// k digits
    for (int j=0; j<r; j++) count[j] = 0;   // Init

    // Count number 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 for last slot of j.
    for (j=1; j<r; j++) count[j] = count[j-1]+count[j];

    // Put recs into bins working from bottom of bin.
    // Since bins fill from bottom, 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 to A
  }
}
```

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

How do n, k and r relate?

Radix Sort Example

Initial Input: Array A

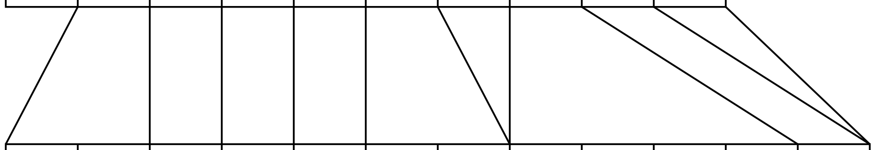
27	91	1	97	17	23	84	28	72	5	67	25

First pass values for Count.
`rtok` = 1.

0	1	2	3	4	5	6	7	8	9
0	2	1	1	1	2	0	4	1	0

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
0	2	3	4	5	7	7	11	12	12



End of Pass 1: Array A.

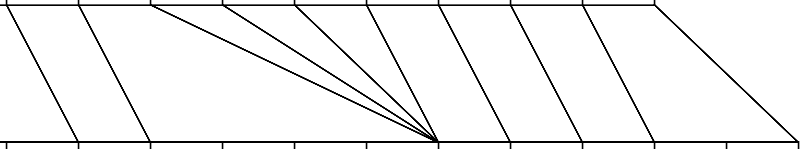
91	1	72	23	84	5	25	27	97	17	67	28

Second pass values for Count.
`rtok` = 10.

0	1	2	3	4	5	6	7	8	9
2	1	4	0	0	0	1	1	1	2

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
2	3	7	7	7	7	8	9	10	12



End of Pass 2: Array A.

1	5	17	23	25	27	28	67	72	84	91	97

Sorting Lower Bound

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

Sorting is $O(n \log n)$.

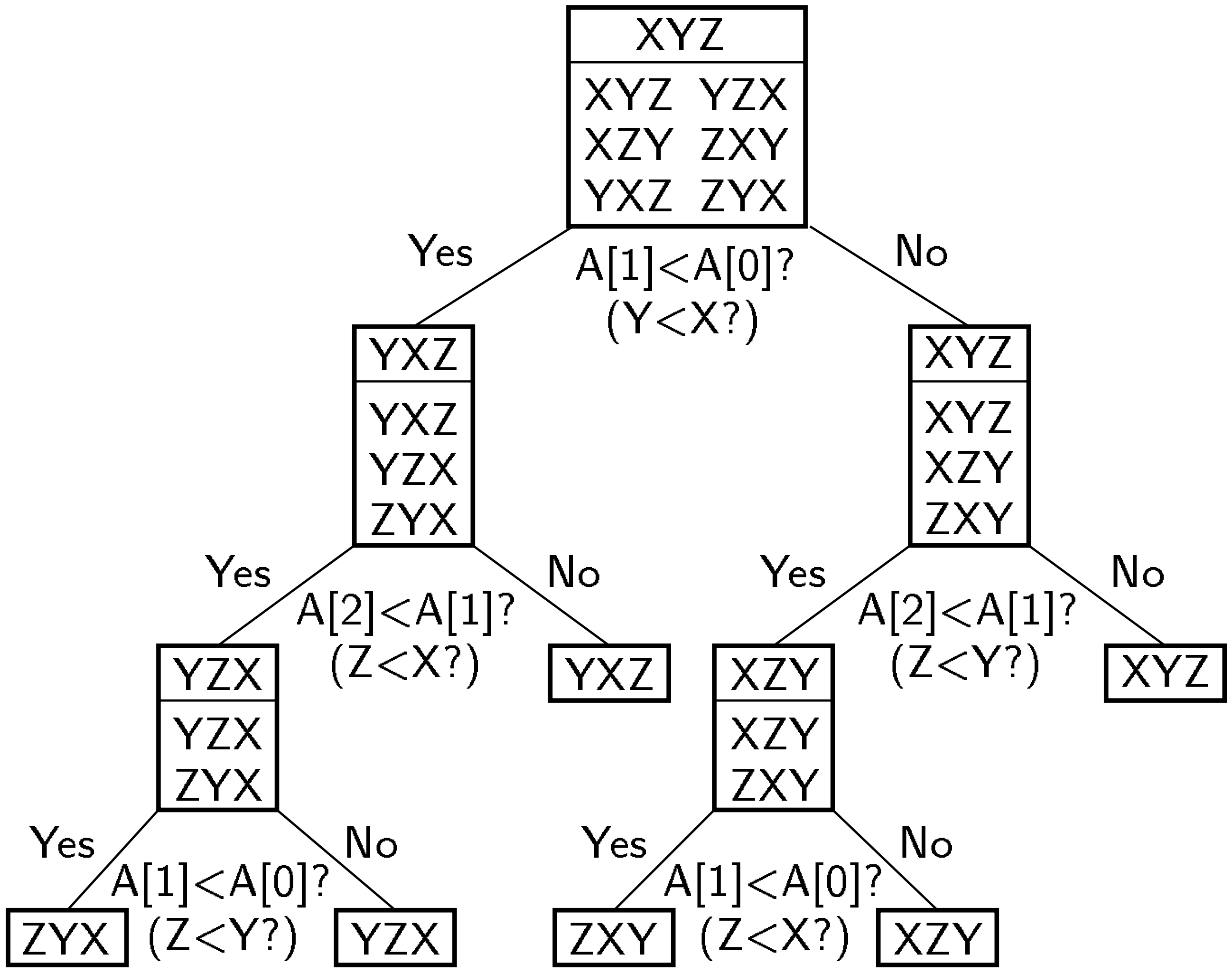
Sorting I/O takes $\Omega(n)$ time.

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.

Decision Trees



There are $n!$ permutations, and at least 1 node for each permutation.

A tree with n nodes has at least $\log n$ levels.

Where is the worst case in the decision tree?

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 $\Theta(\log n!)$.

External Sorting

Problem: Sorting data sets too large to fit in main memory.

- Assume data stored on disk drive.

To sort, portions of the data must be brought into main memory, processed, and returned to disk.

An external sort should minimize disk accesses.

Model of External Computation

Secondary memory is divided into equal-sized **blocks** (512, 2048, 4096 or 8192 bytes are typical sizes).

The basic I/O operation transfers the contents of one disk block to/from main memory.

Under certain circumstances, reading blocks of a file in sequential order is more efficient. (When?)

Typically, the time to perform a single block I/O operation is sufficient to Quicksort the contents of the block.

Thus, our primary goal is to minimize the number fo block I/O operations.

Most workstations today must do all sorting on a single disk drive.

Key Sorting

Often records are large while keys are small.

- Ex: Payroll entries keyed on ID number.

Approach 1: Read in entire records, sort them, then write them out again.

Approach 2: Read only the key values, store with each key the location on disk of its associated record.

If necessary, after the keys are sorted the records can be read and re-written in sorted order.

External Sort: Simple Mergesort

Quicksort requires random access to the entire set of records.

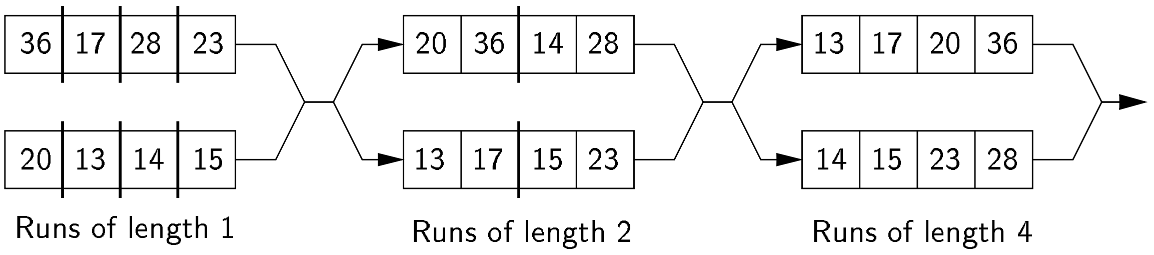
Better: Modified Mergesort algorithm

- Process n elements in $\Theta(\log n)$ passes.

1. Split the file into two files.
2. Read in a block from each file.
3. Take first record from each block, output them in sorted order.
4. Take next record from each block, output them to a *second* file in sorted order.
5. Repeat until finished, alternating between output files. Read new input blocks as needed.
6. Repeat steps 2-5, except this time the input files have groups of two sorted records that are merged together.
7. Each pass through the files provides larger and larger groups of sorted records.

A group of sorted records is called a **run**.

Problems with Simple Mergesort



Is each pass through input and output files sequential?

What happens if all work is done on a single disk drive?

How can we reduce the number of Mergesort passes?

In general, external sorting consists of two phases:

1. Break the file into initial runs.
2. Merge the runs together into a single sorted run.

Breaking a file into runs

General approach:

- Read as much of the file into memory as possible.
- Perform and in-memory sort.
- Output this group of records as a single run.

Replacement Selection

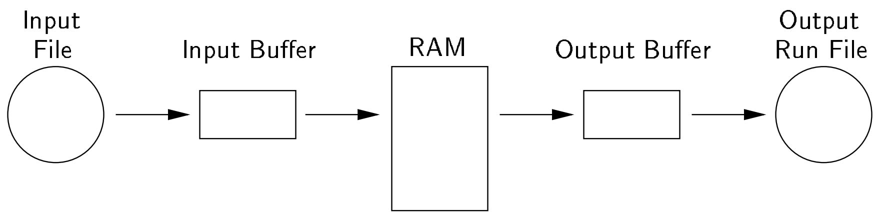
1. Break available memory into an array for the heap, an input buffer and an output buffer.
2. Fill the array from disk.
3. Make a min-heap.
4. Send the smallest value (root) to the output buffer.
5. If the next key in the file is greater than the last value output, then

       Replace the root with this key.

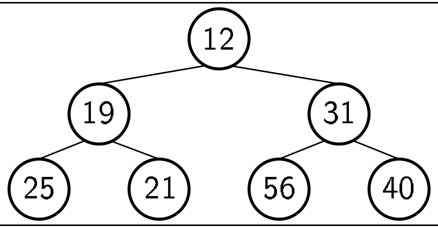
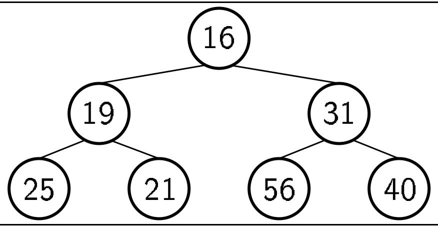
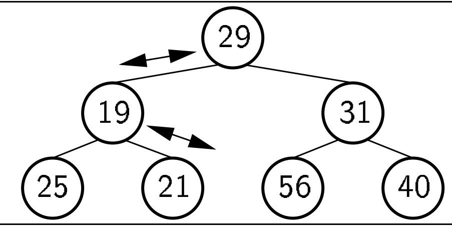
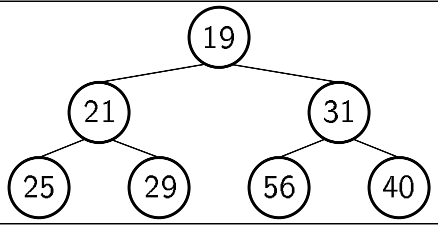
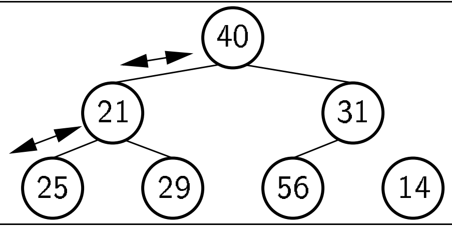
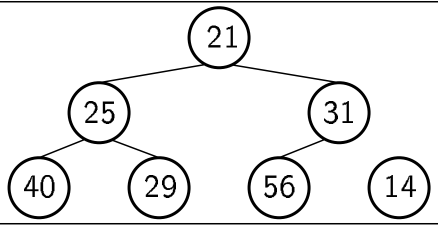
   else

       Replace the root with the last key in the array.

       Add the next record in the file to a new heap (actually, stick it at the end of the array).



Example of Replacement Selection

Input	Memory	Output
16		12
29		16
		
14		19
		
35		21

Benefit from Replacement Selection

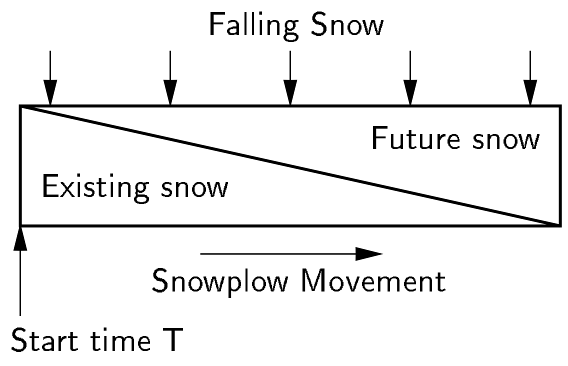
Use double buffering to overlap input, processing and output.

How many disk drives for greatest advantage?

Snowplow argument:

- A snowplow moves around a circular track onto which snow falls at a steady rate.
- At any instant, there is a certain amount of snow S on the track. Some falling snow comes in front of the plow, some behind.
- During the next revolution of the snowplow, all of this is removed, plus 1/2 of what falls during that revolution.
- Thus, the plow removes $2S$ amount of snow.

Is this always true?



Simple Mergesort may not be Best

Simple Mergesort: Place the runs into two files.

- Merge the first two runs to output file, then next two runs, etc.

This process is repeated until only one run remains.

- How many passes for r initial runs?

Is there benefit from sequential reading?

Is working memory well used?

Need a way to reduce the number of passes.

Multiway Merge

With replacement selection, each initial run is several blocks long.

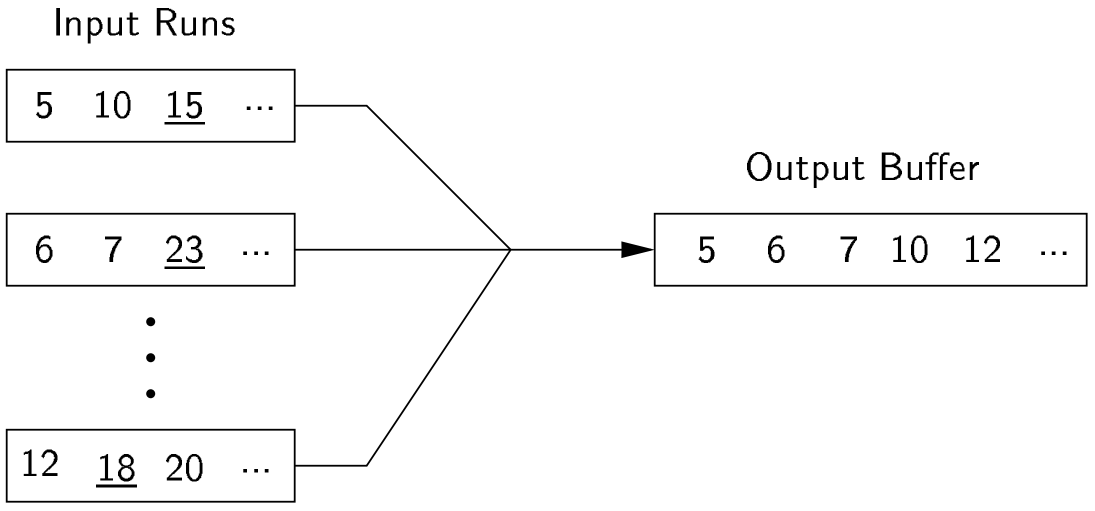
Assume that each run is placed in a separate disk file.

We could then read the first block from each file into memory and perform an r-way merge.

When a buffer becomes empty, read a block from the appropriate run file.

Each record is read only *once* from disk during the merge process.

In practice, use only one file and seek to appropriate block.



Limits to Single Pass Multiway Merge

Assume working memory is b blocks in size.

How many runs can be processed at one time?

The runs are $2b$ blocks long (on average).

How big a file can be merged in one pass?

Larger files will need more passes – but the run size grows quickly!

This approach trades $\Theta(\log b)$ (possibly) sequential passes for a single or a very few random (block) access passes.

General Principals of External Sorting

In summary, a good external sorting algorithm will seek to do the following:

- Make the initial runs as long as possible.
- At all stages, overlap input, processing and output as much as possible.
- Use as much working memory as possible. Applying more memory usually speeds processing.
- If possible, use additional disk drives for more overlapping of processing with I/O, and allow for more sequential file processing.

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 $i, 1 \leq i \leq m$, $a_{k+i} = b_i$.

Example: Search for a certain word or pattern in a text file.

Brute force approach:

$A =$ 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.

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`

Finding a Better Algorithm

Find $B =$ `xyxyyxyxyxx` in

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

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?

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

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? $b_{i-1} = b_{next(i-1)+1}$.
- I.e., 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.

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).

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 B(3) to this
             -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.

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
```

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.

Solution:

Problem: Find the minimum AND the maximum elements.

Solution: Do independently.

- Requires $2n - 3$ comparisons.
- Is this best?

Min and Max

Problem: Find the minimum AND the maximum values.

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.

Min and Max

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?

Kth Smallest Element

Problem: Find the kth smallest element from sequence S.

Also called **order statistics** or **selection**.

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

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

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

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: O(n) time.

Two Largest Elements in a Set

Problem: Given a set S of n numbers, find the two largest.

Want to minimize comparisons.

Assume n is a power of 2.

Solution: Divide and Conquer

Induction Hypothesis: We can find the two largest elements of $n/2$ elements (lists P and Q).

Using two more comparisons, we can find the two largest of q_1, q_2, p_1, p_2.

$$\begin{aligned} T(2n) &= 2T(n) + 2; T(2) = 1. \\ T(n) &= 3n/2 - 2. \end{aligned}$$

Much like finding the max and min of a set. Is this best?

A Closer Examination

Again consider comparisons.

```
If $p_1 > q_1$ then
    compare $p_2$ and $q_1$        [ignore $q_2$]
Else
    compare $p_1$ and $q_2$        [ignore $p_2$]
```

We need only ONE of p_2, q_2.

Which one? It depends on p_1 and q_1.

Approach: Delay computation of the second largest element.

Induction Hypothesis: Given a set of size $< n$, we know how to find the maximum element and a "small" set of candidates for the second maximum element.

Algorithm

Given set S of size n, divide into P and Q of size $n/2$.

By induction hypothesis, we know p_1 and q_1, plus a set of candidates for each second element, C_P and C_Q.

```
If $p_1 > q_1$ then
    $new_1 = p_1; C_{new} = C_P \cup q_1$.
Else
    $new_1 = q_1; C_{new} = C_Q \cup p_1$.
```

At end, look through set of candidates that remains.

What is size of C?

Total cost:

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**.

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.

Probabilistic Quicksort

Quicksort runs into trouble on highly structured input.

Solution: Randomize input order.

- Chance of worst case is then $2/n!$.

Coloring Problem

Let S be a set with n elements, let $S_1, S_2, \cdots, S_k$ be a collection of distinct subsets of S, each containing exactly r elements, $k \leq 2^{r-2}$.

Problem: Color each element of S with one of two colors, red or blue, such that each subset S_i contains at least one red and at least one blue.

Probabilistic solution:

- Take every element of S and color it either red or blue at random.

This may not lead to a valid coloring, with probability

$$\frac{k}{2^{r-1}} \leq \frac{1}{2}.$$

If it doesn't work, try again!

Transforming to Deterministic Alg

First, generalize the problem:

Let $S_1, S_2, \cdots, S_k$ be distinct subsets of S. Let $s_i = |S_i|$.
Assume $\forall i, s_i \geq 2, |S| = n$. Color each element of S red or blue such that every S_i contains a red and blue element.

The probability of failure is at most:

$$F(n) = \sum_{i=1}^{k} 2/2^{S_i}$$

If $F(n) < 1$, then there exists a coloring that solves the problem.

Strategy: Color one element of S at a time, always choosing the color that gives the lower probability of failure.

Deterministic Algorithm

Let $S = \{x_1, x_2, \cdots, x_n\}$.

Suppose we have colored $x_{j+1}, x_{j+2}, \cdots, x_n$ and we want to color x_j. Further, suppose $F(j)$ is an upper bound on the probability of failure.

How could coloring x_j red affect the probability of failing to color a particular set S_i?

Let $P_R(i, j)$ be this probability of failure.

Let $P(i, j)$ be the probability of failure if the remaining colors are randomly assigned.

$P_R(i, j)$ depends on these factors:

1. whether x_j is a member of S_i.
2. whether S_i contains a blue element.
3. whether S_i contains a red element.
4. the number of elements in S_i yet to be colored.

Deterministic Algorithm (cont)

Result:

1. If x_j is not a member of S_i, probability is unchanged.

$$P_R(i,j) = P(i,j).$$

2. If S_i contains a blue element, then $P_R(i,j) = 0$.
3. If S_i contains no blue element and some red elements, then

$$P_R(i,j) = 2P(i,j).$$

4. If S_i contains no colored elements, then probability of failure is unchanged.

$$P_R(i,j) = P(i,j)$$

Deterministic Algorithm (cont)

Similarly analyze $P_B(i,j)$, the probability of failure for set S_i if x_j is colored blue.

Sum the failure probabilities as follows:

$$\begin{aligned} F_R(j) &= \sum_{i=1}^{k} P_R(i,j) \\ F_B(j) &= \sum_{i=1}^{k} P_B(i,j) \end{aligned}$$

Claim: $F_R(n-1) + F_B(n-1) \leq 2F(n)$.

$$P_R(i,j) + P_B(i,j) \leq 2P(i,j).$$

Suffices to show that $\forall i$,

$$P_R(i,j) + P_B(i,j) \leq 2P(i,j).$$

This is clear except in case (3) when $P_R(i,j) = 2P(i,j)$.
But, then case (2) applies on the blue side, so $P_B(i,j) = 0$.

Final Algorithm

```
For $j = n$ downto 1 do
    calculate $F_R(j)$ and $F_B(j)$;
    If $F_R(j) < F_B(j)$ then
      color $x_j$ red
    Else
      color $x_j$ blue.
```

By the claim, $1 \geq F(n) \geq F(n-1) \geq \cdots \geq F(1)$.

This implies that the sets are successfully colored, i.e., $F(1) = 0$.

Key to transformation: We can calculate $F_R(j)$ and $F_B(j)$ efficiently, combined with the claim.

Random Number Generators

Reference: CACM, October 1998.

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 $r(i) = r(j)$?

t should be prime.

Random Number Generators (cont)

Some examples:

$$\begin{aligned}
r(i) &= 6r(i-1) \text{ mod } 13 = \\
&\cdots 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1 \cdots \\
r(i) &= 7r(i-1) \text{ mod } 13 = \\
&\cdots 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1 \cdots \\
r(i) &= 5r(i-1) \text{ mod } 13 = \\
&\cdots 1, 5, 12, 8, 1 \cdots \\
&\cdots 2, 10, 11, 3, 2 \cdots \\
&\cdots 4, 7, 9, 6, 4 \cdots \\
&\cdots 0, 0 \cdots
\end{aligned}$$

The last one depends on the start value of the seed.

Suggested generator:

$$r(i) = 16807r(i-1) \text{ mod } 2^{31} - 1$$

Mode of a Multiset

Multiset: not (necessarily) distinct elements.

A **mode** of a multiset is an element that occurs most frequently (there may be more than one).

The number of times that a mode occurs is its **multiplicity**

Problem: Find the mode of a given multiset S.

Solution: Sort, and then scan in sequential order counting multiplicities.

$O(n \log n + n)$. Is this best?

Mode Induction

Induction Hypothesis: We know the mode of a multiset of $n-1$ elements.

Problem: The nth element may break a tie, creating a new mode.

Stronger IH: Assume that we know ALL modes of a multiset with $n-1$ elements.

Problem: We may create a new mode with the nth element.

What if the nth element is chosen to be special?

- Example: nth element is the maximum element
- Better: Remove ALL occurrences of the maximal element.

Still too slow – particularly if elements are distinct.

New Approach

Use divide and conquer:

- Divide the multiset into two approximately equal, disjoint parts.

Note that we can find the median (position $n/2$) in O(n) time.

This makes 3 multilists: less than, equal to, and greater than the median.

Solve for each part.

$$T(n) \leq 2T(n/2) + O(n), T(2) = 1.$$

Result: O($n \log n$). No improvement.

Observation: Don't look at lists smaller than size M where M is the multiplicity of the mode.

Implementation

Look at each submultilist.

If all contain more than one element, subdivide them all.

$$\begin{aligned} T(n) &\leq 2T(n/2) + O(n), T(M) = O(M). \\ T(n) &= O(n \log(n/M)). \end{aligned}$$

This may be superior to sorting, but only if M is “large” and comparisons are expensive.

Graph Algorithms

Graphs are useful for representing a variety of concepts:

- Data Structures
- Relationships
- Families
- Communication Networks
- Road Maps

Graph Terminology

A **graph** $G = (V, E)$ 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

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.

Graph Traversals

Various problems require a way to **traverse** a graph – that is, visit each vertex and edge in a systematic way.

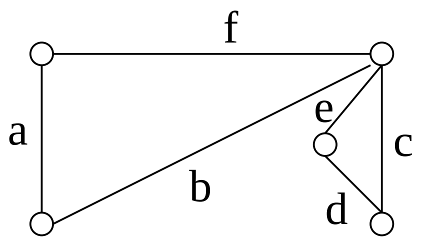
Three common traversals:

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

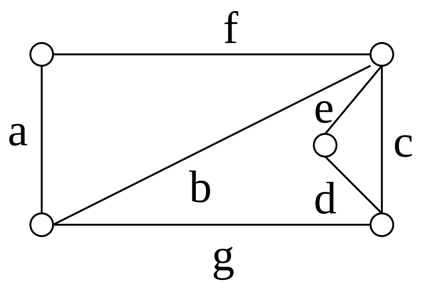
Eulerian Tours

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 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:

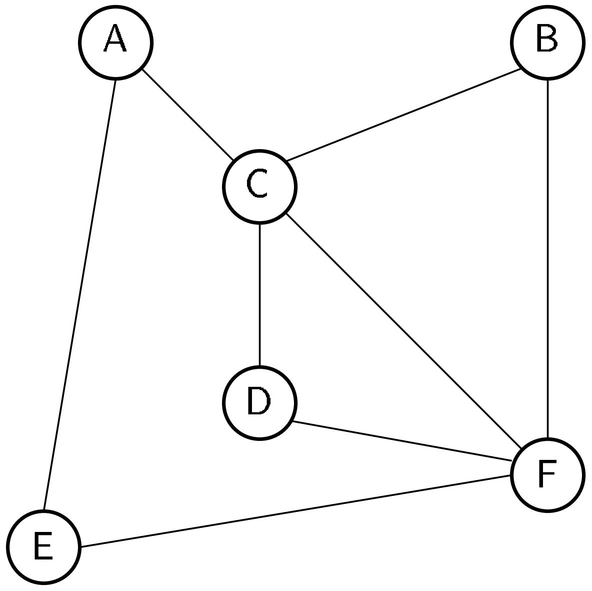
- Start with an arbitrary vertex and follow a path until you return to the vertex.
- Remove this circuit. What remains are connected components G_1, G_2, ..., G_k each with nodes of even degree and $< m$ edges.
- By IH, each connected component has an Eulerian tour.
- Combine the tours to get a tour of the entire graph.

Depth First Search

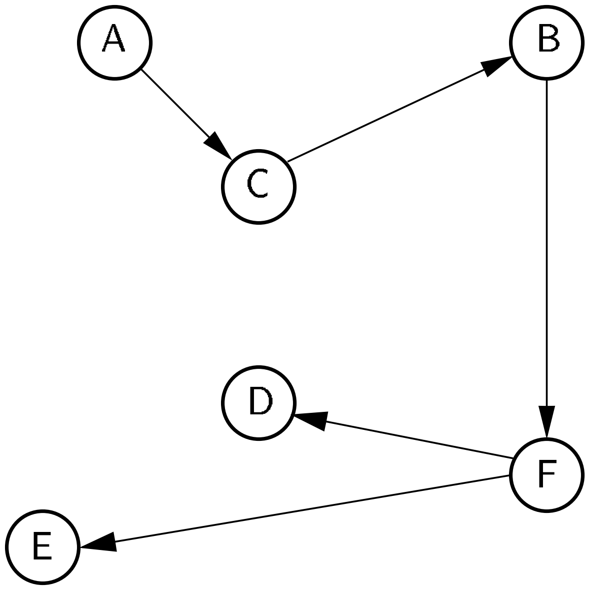
```
void DFS(Graph& G, int v) {  // Depth first search
  PreVisit(G, v);           // Take appropriate action
  G.setMark(v, VISITED);
  for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
    if (G.getMark(G.v2(w)) == UNVISITED)
      DFS(G, G.v2(w));
  PostVisit(G, v);          // Take appropriate action
}
```

Initial call: `DFS(G, r)` where r is the **root** of the DFS.

Cost: $\Theta(|V| + |E|)$.



(a)



(b)

DFS Tree

If we number the vertices in the order that they are marked, we get **DFS numbers**.

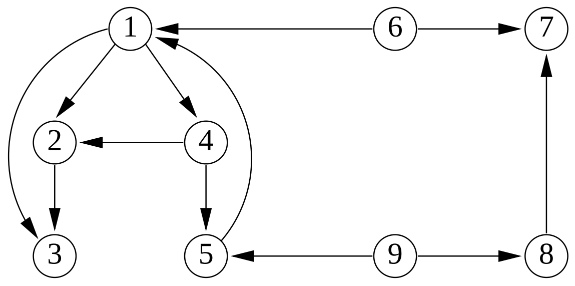
Lemma 7.2: Every edge $e \in E$ is either in the DFS tree T, or connects two vertices of G, one of which is an ancestor of the other in T.

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.

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.

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:

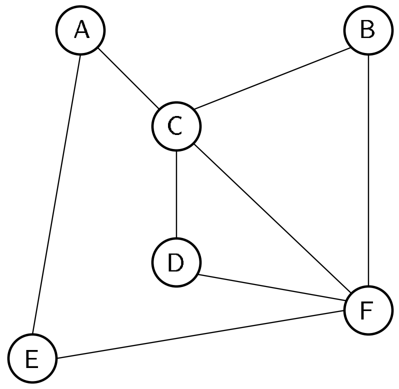
1. Suppose a DFS produces a back edge (v, w).
   - v and w are in the same DFS tree, w an ancestor of v.
   - (v, w) and the path in the tree from w to v form a directed cycle.
2. Suppose G has a directed cycle C.
   - Do a DFS on G.
   - Let w be the vertex of C with smallest DFS number.
   - Let (v, 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.

Breadth First Search

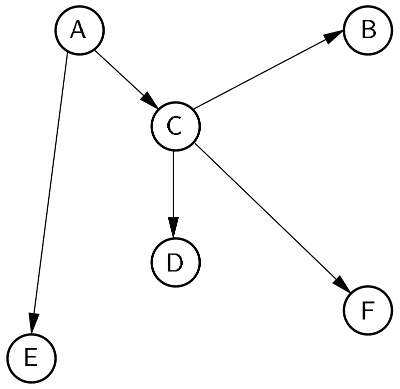
Like DFS, but replace stack with a queue.

Visit the vertex's neighbors before continuing deeper in the tree.

```
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 = G.first(v); G.isEdge(w); w=G.next(w))
      if (G.getMark(G.v2(w)) == UNVISITED) {
        G.setMark(G.v2(w), VISITED);
        Q.enqueue(G.v2(w));
      }
    PostVisit(G, v);         // Take appropriate action
}}
```



(a)

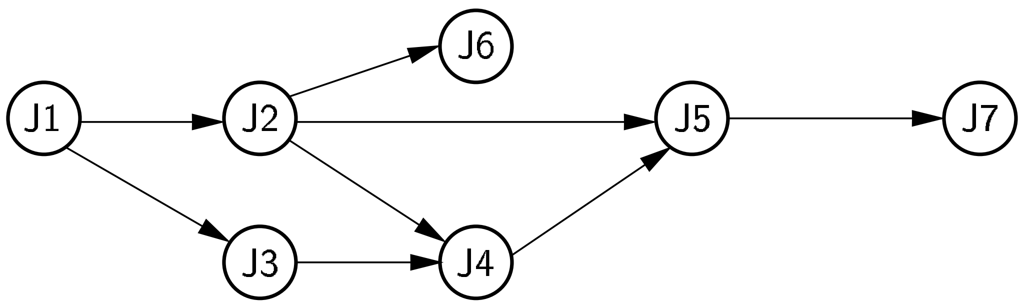


(b)

Non-tree edges connect vertices at levels differing by 0 or 1.

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.



```
void topsort(Graph& G) { // Topological sort: recursive
  for (int i=0; i<G.n(); i++) // Initialize Mark array
    G.setMark(i, UNVISITED);
  for (i=0; i<G.n(); i++)     // Process all vertices
    if (G.getMark(i) == UNVISITED)
      tophelp(G, i);          // Call helper function
}

void tophelp(Graph& G, int v) { // Helper function
  G.setMark(v, VISITED);
  // No PreVisit operation
  for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
    if (G.getMark(G.v2(w)) == UNVISITED)
      tophelp(G, G.v2(w));
  printout(v);          // PostVisit for Vertex v
}
```

Queue-based Topological Sort

```
void topsort(Graph& G) {   // Topological sort: Queue
  Queue Q(G.n());
  int Count[G.n()];

  for (int v=0; v<G.n(); v++) Count[v] = 0; // Init
  for (v=0; v<G.n(); v++)  // Process every edge
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
       Count[G.v2(w)]++;   // Add to v2's prereq count
  for (v=0; v<G.n(); v++)  // Initialize Queue
    if (Count[v] == 0)     // Vertex has no prereqs
      Q.enqueue(v);
  while (!Q.isEmpty()) {   // Process the vertices
    int v = Q.dequeue();
    printout(v);           // PreVisit for Vertex V
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w)) {
      Count[G.v2(w)]--;    // One less prerequisite
      if (Count[G.v2(w)] == 0) // Vertex is now free
        Q.enqueue(G.v2(w));
    }
  }
}
```

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.

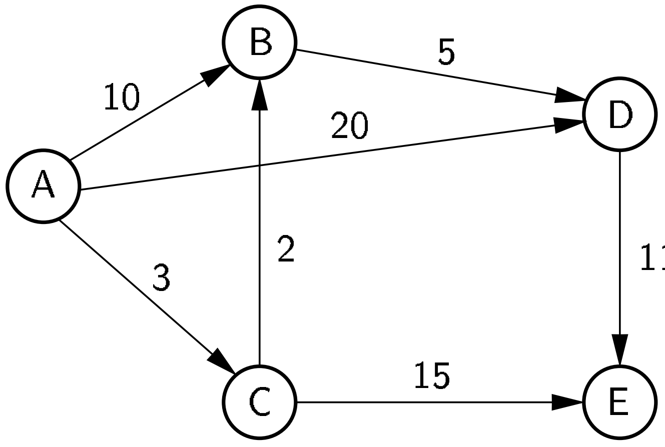
Our algorithms will actually calculate only **distances**.

Shortest Paths 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) $= \infty$.



Single Source Shortest Paths

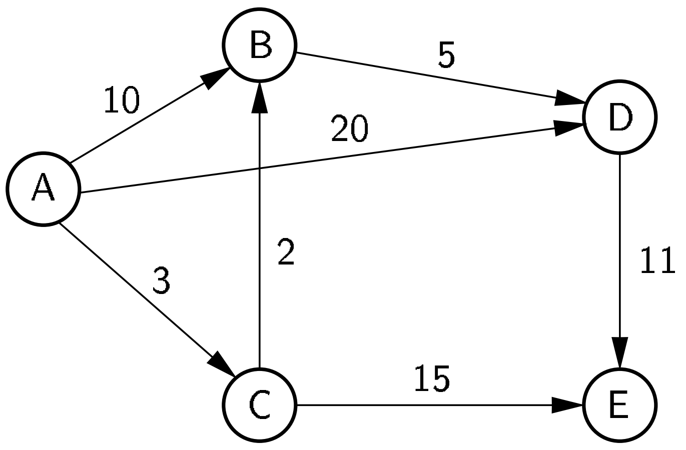
Given start vertex s, find the shortest path from s to all other vertices.

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.

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: Array

```
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 = G.first(v); G.isEdge(w); w=G.next(w))
      if (D[G.v2(w)] > (D[v] + G.weight(w)))
        D[G.v2(w)] = D[v] + G.weight(w);
  }
}

int minVertex(Graph& G, int* D) { // Get mincost vertex
  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++; i<G.n(); i++)  // Now find smallest D value
    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: $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$.

Dijkstra's Algorithm: Priority Queue

```
class Elem { public: int vertex, dist; };
int key(Elem x) { return x.dist; }
void Dijkstra(Graph& G, int s) {  // W/ priority queue
  int v;                          // The current vertex
  int D[G.n()];                   // Distance array
  Elem temp;
  Elem E[G.e()];                  // Heap array
  temp.dist = 0; temp.vertex = s;
  E[0] = temp;                    // Initialize heap
  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++) {       // Now, get distances
    do { temp = H.removemin();  v = temp.vertex; }
      while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (D[v] == INFINITY) return; // Rest unreachable
    for (Edge w = G.first(v); G.isEdge(w); w=G.next(w))
      if (D[G.v2(w)] > (D[v] + G.weight(w))) {
        D[G.v2(w)] = D[v] + G.weight(w); // Update D
        temp.dist = D[G.v2(w)]; temp.vertex = G.v2(w);
        H.insert(temp);  // Insert new distance in heap
}}}
```

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: $\Theta((|V| + |E|) \log |V|)$.

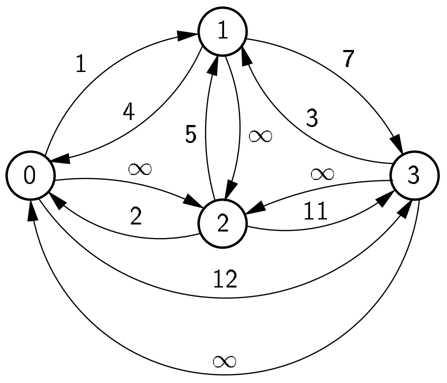
All Pairs Shortest Paths

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.



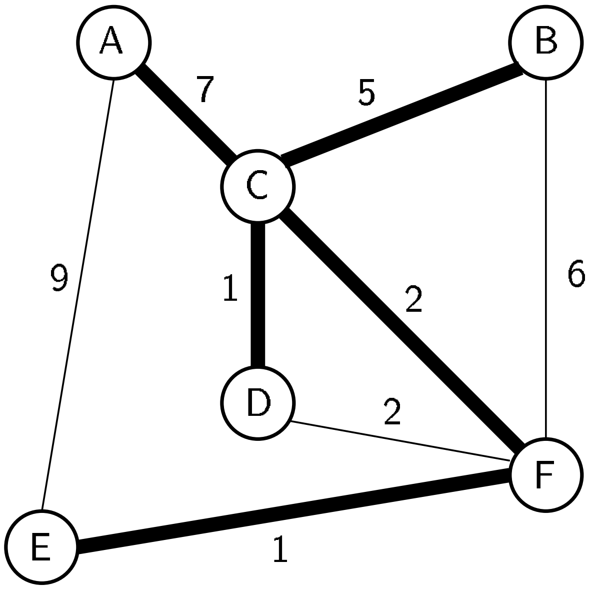
Floyd’s Algorithm

```
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 all 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];
}
```

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.



Key Theorem for MST

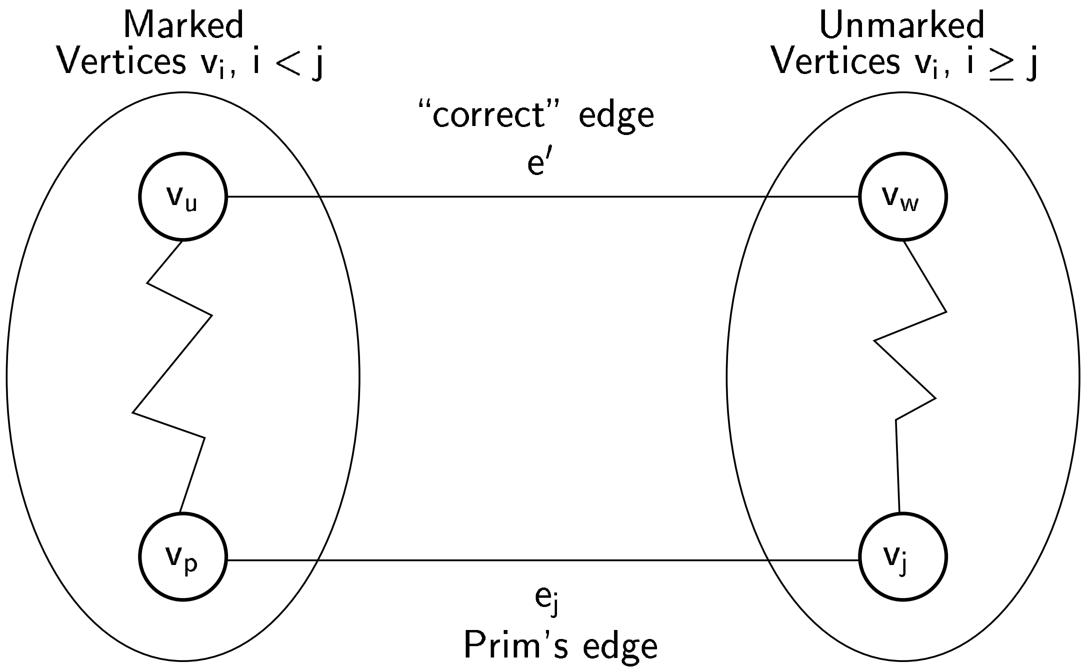
Let V_1, V_2 be an arbitrary, non-trivial partition of V. Let (v_1, v_2), $v_1 \in V_1, v_2 \in V_2$, be the cheapest edge between V_1 and V_2. Then (v_1, v_2) is in some MST of G.

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).

Key Theorem Figure



Prim's MST Algorithm

```
void Prim(Graph& G, int s) {       // Prim's MST alg
  int D[G.n()];                    // Distance vertex
  int V[G.n()];                    // Who's closest
  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); // Add to MST
    if (D[v] == INFINITY) return; // Rest unreachable
    for (Edge w = G.first(v); G.isEdge(w); w=G.next(w))
      if (D[G.v2(w)] > G.weight(w)) {
        D[G.v2(w)] = G.weight(w);  // Update distance,
        V[G.v2(w)] = v;            //   who came from
}}}

int minVertex(Graph& G, int* D) { // Min cost vertex
  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++)  // Now find smallest value
    if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))
      v = i;
  return v;
}
```

This is an example of a **greedy** algorithm.

Alternative Prim's Implementation

Like Dijkstra's algorithm, we can implement Prim's algorithm with a priority queue.

```
void Prim(Graph& G, int s) {  // W/ priority queue
  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;  // Init
  D[s] = 0;
  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); // Add to MST
    if (D[v] == INFINITY) return; // Rest unreachable
    for (Edge w = G.first(v); G.isEdge(w); w=G.next(w))
      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 distance in heap
      }
}}
```

Kruskal's MST Algorithm

```
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 on array
    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 the edges
  int numMST = G.n();        // Init w/ n equiv classes
  for (i=0; numMST>1; i++) {   // Combine equiv classes
    Elem temp = H.removemin(); // Get next cheap edge
    Edge w = temp.edge;
    int v = G.v1(w);  int u = G.v2(w);
    if (A.differ(v, u)) { // If different equiv classes
      A.UNION(v, u);      // Combine equiv classes
      AddEdgetoMST(G.v1(w), G.v2(w));  // Add to MST
      numMST--;           // One less MST
    }
  }
}
```

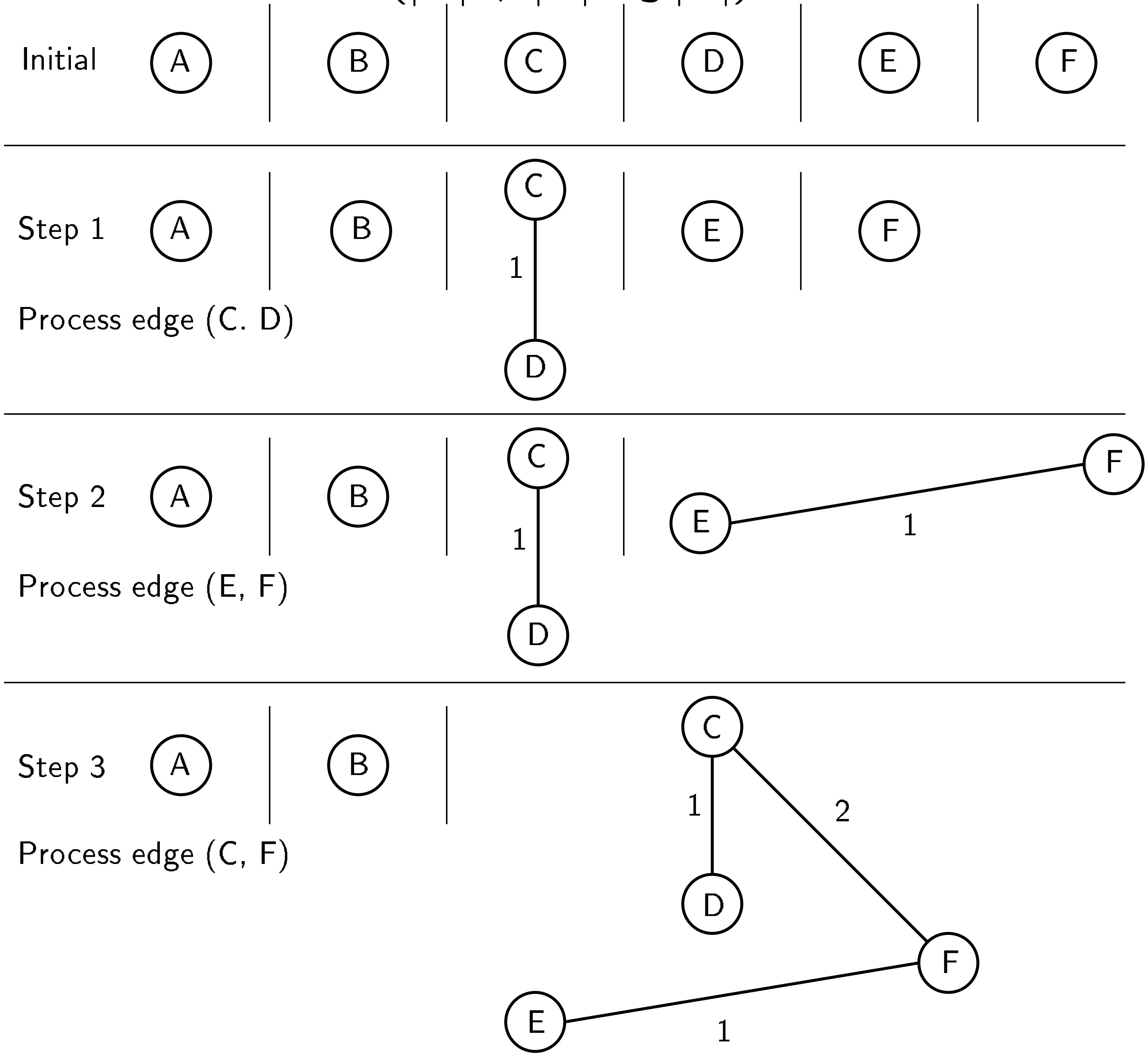
How do we compute function `MSTof(v)`?

Solution: UNION-FIND algorithm (Section 4.3).

Kruskal's Algorithm Example

Time dominated by cost for initial edge sort.

Total cost: $\Theta(|V| + |E| \log |E|)$.



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.

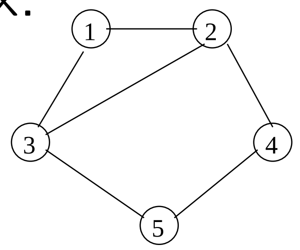
Problem: Form as many compatible non-overlapping teams as possible.

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** is a matching that includes every vertex.



Very Dense Graphs

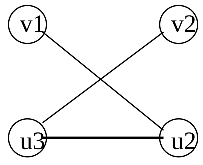
Theorem: Let $G = (V, E)$ be an undirected graph with $|V| = 2n$ and every vertex having degree $\geq n$. Then G contains a perfect matching.

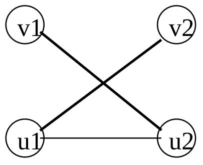
Proof: Suppose that G does not contain a perfect matching.

- 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 M', but $|M'| < n$.
- Thus, some element of M' is adjacent to 3 edges from v_1 and v_2.
- Let (u_1, u_2) be such an element.
- Replacing (u_1, u_2) with (v_1, u_2) and (v_2, u_1) results in a larger matching.
- Theorem proven by contradiction.

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.

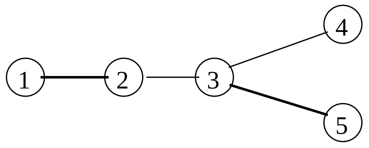
In one basic step, the unmatched and matched edges are switched.

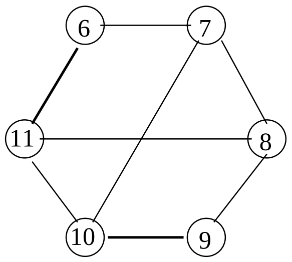
We have motivated the following definitions:

Let $G = (V, E)$ be an undirected graph and $M \subseteq E$ a matching.

A path P that consists of alternately matched and unmatched edges is called an **alternating path**. An alternating path from one unmatched vertex to another is called an **augmenting path**.

Matching Example





1, 2, 3, 5 is an alternating path but NOT an augmenting path.

7, 6, 11, 10, 9, 8 is an augmenting path with respect to the given matching.

Observation: If a matching has an augmenting path, then the size of the matching can be increased by one by switching matched and unmatched edges along the augmenting path.

The Augmenting Path Theorem

Theorem: A matching is maximum iff it has no augmenting paths.

Proof:

- If a matching has augmenting 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.

- Therefore, the connected components of G' are either even-length cycles or alternating paths.
- Since $|M'| > |M|$, there must be a component of G' that is an alternating path having more M' edges than M edges.
- This is an augmenting path for M.

Bipartite Matching

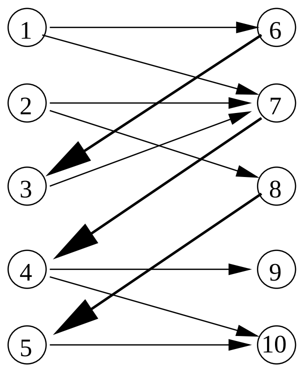
A **bipartite graph** $G = (U, V, E)$ consists of two disjoint sets of vertices U and V together with edges E such that every edge has an endpoint in U and an endpoint in V.

Bipartite matching naturally models a number of assignment problems, such as assignment of workers to jobs.

Augmenting paths will work to find a maximum bipartite matching. An augmenting 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 augmenting path.

Bipartite Matching Example



2, 8, 5, 10 is an augmenting path.

1, 6, 3, 7, 4, 9 and 2, 8, 5, 10 are **disjoint** augmenting paths that we can augment **independently**.

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 augmenting paths.

Augment along each path independently.

Repeat until no augmenting paths remain.

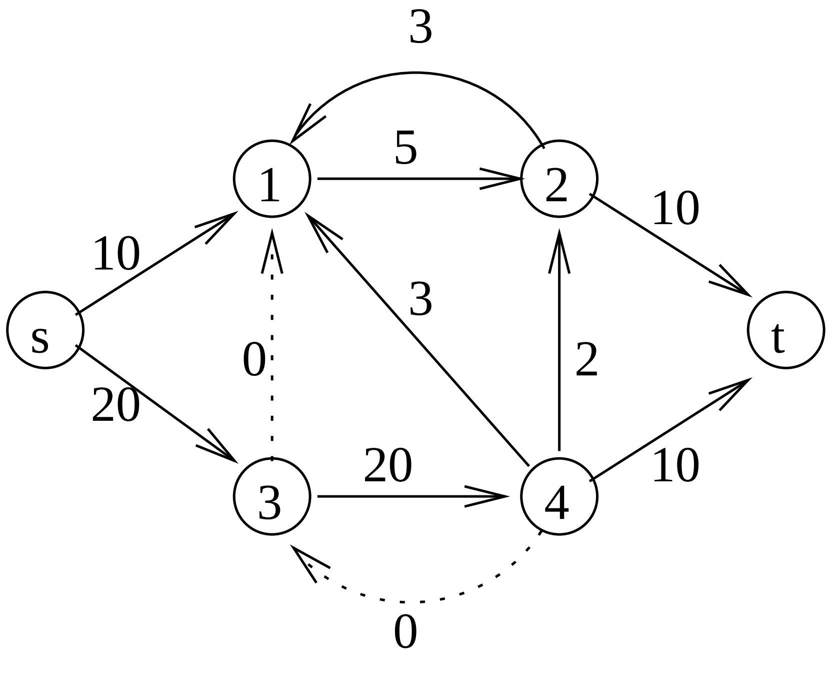
Time complexity $O((|V| + |E|)\sqrt{|V|})$.

Network Flows

Models distribution of utilities in networks such as oil pipelines, waters 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 Flow Definitions

A **flow** in a network is a function

$$f : V \times V \rightarrow R$$

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**:

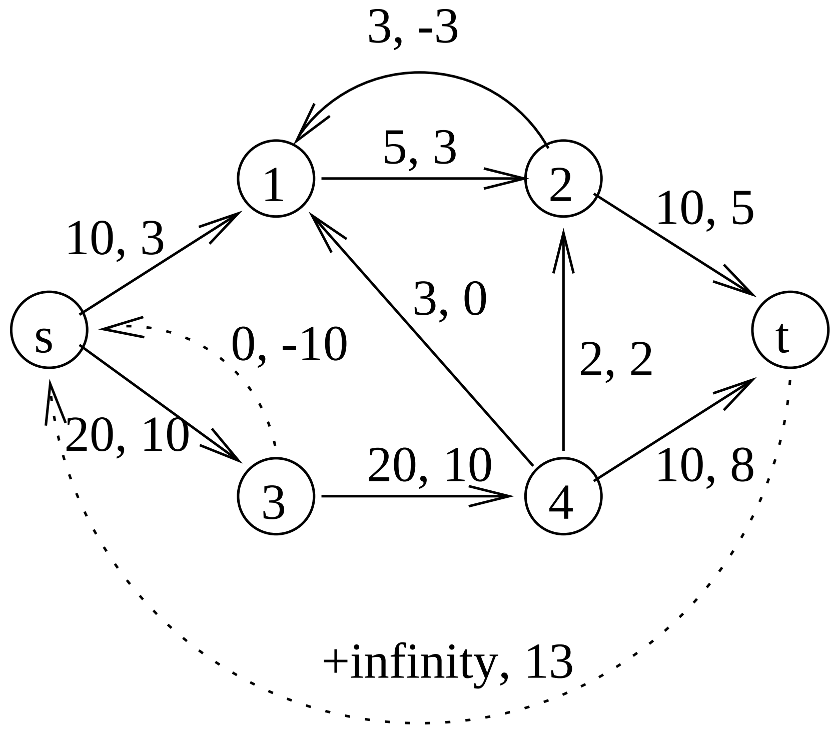
$$\forall v \in V - \{s, t\}, \quad \sum f(v, w) = 0.$$

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.

Flow Example



Edges are labeled "capacity, flow".

Can omit edges w/o capacity and non-negative flow.

The **value** of a flow is

$$|f| = \sum_{w \in V} f(s, w) = \sum_{w \in V} f(w, t).$$

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.

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'),

$$f(X,X') = |f|.$$

Proof:

$$\begin{aligned} f(X,X') &= \sum_{v\in X, w\in X'} f(v,w) \\ &= \sum_{v\in X, w\in V} f(v,w) - \sum_{v\in X, w\in X} f(v,w) \\ &= \sum_{w\in V} f(s,w) - 0 \\ &= |f|. \end{aligned}$$

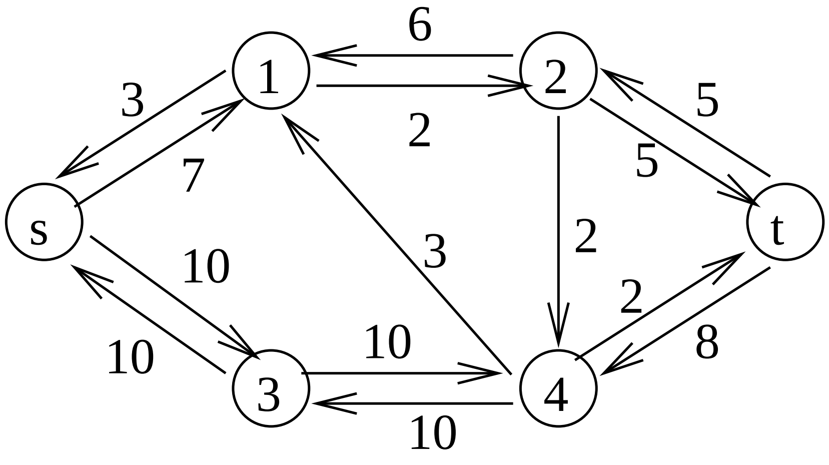
Corollary: The value of any flow is less than or equal to the capacity of a min cut.

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.



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 $= c(1, 2)$.

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.

(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.

Proof (cont)

(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.

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.

Edmonds-Karp Algorithm

For integral capacities.

Select an augmenting path in R of minimum length.

Performance: $O(|V|^3)$ where c is an upper bound on capacities.

There are numerous other approaches to finding augmenting paths, giving a variety of different algorithms.

Network flow remains an active research area.

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

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**.

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.

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.

Point in Polygon Analysis

Time complexity:

- Compare the ray to each edge.
- Each intersection takes constant time.
- Running time is O(n).

Improving efficiency:

- O(n) is best possible for problem as stated.
- Many lines are "obviously" not intersected.

Two general principles for geometrical and graphical algorithms:

1. Operational (constant time) improvements:
   - Only do full calc. 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. For 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.

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.

Simple Polygons (cont)

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.

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 $180\circ$.

Implementation

How do we find the next point?

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.

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 is 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.

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).

Subproblems

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?)

- Standard alt: 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.

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.

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;
}
```

Time complexity:

Sort by x value: O($n \log n$).

For qth point:

- Compute angles: $O(q)$
- Find max and min: $O(q)$
- Delete and insert points: $O(q)$.

$$T(n) = T(n-1) + O(n) = O(n^2)$$

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.

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;
      $P = P \cup q$;
      $p = q$;
   }
}
```

Complexity:

- To add kth point, find the min angle among $n - k$ lines.

Often good in average case.

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.

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.

Graham's Scan (cont)

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 Algorithm

```
ALGORITHM GrahamsScan(Pointset P) {
  Point $p_1$ = the point in $P$ with largest $x$ coordinate;
  $P$ = SimplePolygon($P, p_1$); // Order points in $P$
  Point $q_1 = p_1$;
  Point $q_2 = p_2$;
  Point $q_3 = p_3$;
  int $m = 3$;
  for ($k = 4$ to $n$) {
    while (angle($-q_{m-1} - q_m-$, $-q_m - p_k-$) $\leq 180^\circ$) do
      $m = m - 1$;
    $m = m + 1$;
    $q_m = p_k$;
  }
}
```

Time complexity:

- Other than Simple Polygon, all steps take $O(n)$ time.
- Thus, total cost is O($n \log n$).

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 O($n \log n$) would provide a sorting algorithm faster than O($n \log n$).

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 O(n^2) time.

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.

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 need be considered.

Worst case: All points are in the strip.

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 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).

$T(n) = 2T(n/2) + O(n \log n)$.

$T(n) = O(n \log^2 n)$.

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.

$T(n) = 2T(n/2) + O(n)$.

$T(n) = O(n \log n)$.

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.

Sweep Line Algorithms

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**.

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.

Sweep Line Algorithms (cont)

Preprocessing Step:

- Sort all line segments by x coordinate.

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.

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.

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.

Data Structure Requirements

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).

Time complexity:

- Sort by x: $O((m+n)\log(m+n))$.
- Each insert/delete: $O(\log n)$.
- Total cost is $O(n \log n)$ 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.

Voronoi Diagrams

For some point P in S, P's **Voronoi polygon** is the locus of points closer to P than to any other point in S.

- Alt: The Voronoi polygon is the intersection of the half-planes formed by the bisectors of lines connecting P and P_i in $\{S\}$ for $P_i \neq P$.

Given: A set S of points in the plane.

All of the points in S together partition the plane into a set of disjoint regions called the **Voronoi Diagram** for S.

Problem: Determine Voronoi Diagram of S.

Voronoi Diagram Properties

Assumption: No four points of the original set S are cocircular.

Theorem: Every vertex of the Voronoi diagram is the common intersection of exactly three edges.

Proof:

- A vertex is equidistant from all points in S bisected by an edge touching that vertex.
- By the assumption, there can be at most 3 such points.
- If there were only two edges, both would be bisectors for the same pair of points.

Voronoi Diagram Size

Theorem: The straight-line dual of the Voronoi diagram is a trianglulation of S.

Theorem: A Voronoi diagram on N points has at most $3N - 6$ edges.

Proof:

- Each edge in the straight-line dual corresponds to a unique Voronoi edge.
- The dual is a planar graph on N vertices.
- By Euler's formula, it has at most $3N - 6$ edges.

Corollary: The Voronoi diagram can be stored in $\Theta(N)$ space.

Voronoi Diagram Construction

Induction Hypothesis: We can construct the Voronoi diagram for a set of points with size $< n$.

Approach: Divide and conquer.

- Partition set S into 2 equal-sized subsets S_1 and S_2 by means of a vertical median cut.
- Construct Voronoi diagrams VOR(S_1) and VOR(S_2).
- Construct a **polygonal chain** σ separating S_1 and S_2.
- Discard all edges of VOR(S_1) to the right of σ and all edges of VOR(S_2) to the left of σ.

Constructing the Dividing Chain

Assume (by induction) that we have the convex hull for S_1 and S_2.

The two ends of the dividing chain will be semi-infinite rays.

These rays are the bisectors for supporting segments for the two convex hulls.

- These segments can be found in O(n) time.

Constructing the Dividing Chain (cont)

Procedure:

- Move inwards along one of the rays until an edge of either VOR(S_1) or VOR(S_2) is encountered.
- At this point, the chain enters a region where it must form the bisector for a different pair of points, requiring it to change direction.
- Continue modifying the direction of the chain until the second ray is reached.

With proper organization of the Voronoi diagrams, we can crate the chain in $O(N)$ time.

Cost of Voronoi Construction

Intial Sort:

Time to partition the set:

Time to solve the two subproblems:

Time to create the chain:

Time to eliminate extra edges:

Total cost:

Lower Bound for Voronoi Diagram Construction

Theorem: Given the Voronoi diagram on N points, their convex hull can be found in linear time.

Proof:

1. We constructed the convex hull as part of the Voronoi diagram construction algorithm.
2. Look at each point, checking its edges until a ray is found.
   - This point is a hull vertex.
   - Find its counter-clockwise neighbor which is also a hull point.
   - Follow the hull points around the entire set.
   - No edge needs to be looked at more than 3 times.

All Nearest Neighbors

Problem: All Nearest Neighbors (ANN): Find the nearest neighbor for each point in a set S of points.

Theorem: Given the Voronoi diagram for S, the ANN problem can be solved in linear time.

Proof:

- Every nearest neighbor of a point p_i defines an edge of the Voronoi diagram.
- Since every edge belongs to two Voronoi polygons, no edge will be examined more than twice.

Corollary: ANN can be solved in O($n \log n$) time.

Reductions

A **reduction** is a transformation of one problem to another.

Purpose: To compare the relative difficulty of two problems.

Examples:

1. Sorting reals reduces in linear time to the problem of finding a convex hull in two dimensions.

   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.

2. Finding a convex hull in two dimensions reduces in linear time to the problem of finding a Voronoi diagram in two dimensions.
   Again, there is an $\Omega(n \log n)$ lower bound on finding the Voronoi diagram.

Reduction Notation

We denote names of problems with all capital letters.

- Ex: SORTING, CONVEX HULL, VORONOI

What is a problem?

- A relation consisting of ordered pairs (**I**, **S**).
- **I** is the set of **instances** (allowed inputs).
- **S** is a relation that gives the correct solutions.

Example: SORTING = (**I**, **S**).
I = set of finite subsets of R.

- Prototypical instance: $\{x_1, x_2, ..., x_n\}$.

S is a function that takes a finite set of real numbers and returns the numbers in sorted order.

S($\{x_1, x_2, ..., x_n\}$) $= x_{i[1]}, x_{i[2]}, ..., x_{i[n]}$ such that $x_{i[k]} \in \{x_1, x_2, ..., x_n\}$ and $x_{i[1]} < x_{i[2]} < ... < x_{i[n]}$.

Black Box Reduction

The job of an algorithm is to take an instance of X $\in \mathbf{I}$ and return any solution in $\mathbf{S}$(X) or to report that there is no solution, $\mathbf{S}(X) = \emptyset$.

A **reduction** from problem ($\mathbf{I}$, $\mathbf{S}$) to problem ($\mathbf{I}'$, $\mathbf{S}'$) consists of two transformations (functions) T, T'.

T: $\mathbf{I} \Rightarrow \mathbf{I}'$

- Maps instances of the first problem to instances of the second.

T': $\mathbf{S}' \Rightarrow \mathbf{S}$

- Maps solutions of the second problem to solutions of the first.

Black box idea:

1. Start with an instance X $\in \mathbf{I}$.
2. Transform an instance X' = T(X) $\in \mathbf{I}'$.
3. Use a "black box" algorithm as a subroutine to find a solution Y' $\in \mathbf{S}'$(X').
4. Transform to a solution Y = T'(Y') $\in$ $\mathbf{S}$(X).

More Notation

If ($\mathbf{I}$, **S**) reduces to ($\mathbf{I}'$, **S**$'$), write:
 ($\mathbf{I}$, **S**) $\leq$ ($\mathbf{I}'$, **S**$'$).

This notation suggests that ($\mathbf{I}$, **S**) is **no harder than** ($\mathbf{I}'$, **S**$'$).

Examples:

- SORTING $\leq$ CONVEX HULL
- CONVEX HULL $\leq$ VORONOI

The time complexity of T and T' is important to the time complexity of the black box algorithm for ($\mathbf{I}$, **S**).

If combined time complexity is $O(g(n))$, write:
 ($\mathbf{I}$, **S**) $\leq_{O(g(n))}$ ($\mathbf{I}'$, **S**$'$).

Reduction Example

SORTING = ($\mathbf{I}$, $\mathbf{S}$)
CONVEX HULL = ($\mathbf{I}'$, $\mathbf{S}'$).

1. S = $\{x_1, x_2, ..., x_n\} \in \mathbf{I}$.
2. T(X) = X'
   $= \{(x_1, x_1^2), (x_2, x_2^2), ..., (x_n, x_n^2)\} \in \mathbf{I}'$.
3. Solve CONVEX HULL for X' to give solution Y'
   $= (x_{i[1]}, x_{i[1]}^2), (x_{i[2]}, x_{i[2]}^2), ..., (x_{i[n]}, x_{i[n]}^2)$.
4. T' finds a solution to X from Y' as follows:
   (a) Find $(x_{i[k]}, x_{i[k]}^2)$ such that $x_{i[k]}$ is minimum.
   (b) Y $= x_{i[k]}, x_{i[k+1]}, ..., x_{i[n]}, x_{i[1]}, ..., x_{i[k-1]}$.

For a reduction to be useful in the context of algorithms, 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 above.

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.

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 $O(g(n) + h(n))$.

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 $S = \cup_{i=1}^{k} S_i$. $U = \{S_1, S_2, \cdots, S_k\}$.
- $V = S$. $E = \{(S_i, x_j) | x_j \in S_i\}$.

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.

Simple Polygon Lower Bound

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:

$$\{(x_1, \sqrt{M^2 - x_i^2}), \cdots, (x_n, \sqrt{M^2 - x_n^2})\}.$$

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.

As with CONVEX HULL, the sorted order is easily obtained from the solution to SIMPLE POLYGON.

By the Lower Bound Theorem, SIMPLE POLYGON is $\Omega(n \log n)$.

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 Squaring

Problem: Compute A^2 where A is an $n \times n$ matrix.

MATRIX MULTIPLY $\leq_{O(n^2)}$ SQUARING.

$$\left[\begin{array}{cc} 0 & A \\ B & 0 \end{array} \right]^2 = \left[\begin{array}{cc} AB & 0 \\ 0 & BA \end{array} \right]$$

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.

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.

Example: NETWORK FLOW

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: $x_i \le c_i \quad 1 \le i \le n$.

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 (cont)

Non-negative constraints: $x_i \geq 0 \quad 1 \leq i \leq n$.

Maximize: $x_1 + x_4$ subject to:

$$\begin{aligned}
x_1 &\leq 4 \\
x_2 &\leq 3 \\
x_3 &\leq 2 \\
x_4 &\leq 5 \\
x_5 &\leq 7 \\
x_1 + x_3 - x_2 &= 0 \\
x_4 - x_3 - x_5 &= 0 \\
x_1, \cdots, x_5 &\geq 0
\end{aligned}$$

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 <u>**matched**</u>.

Objective function: Maximize

$$\sum_{i=1}^{n} x_i.$$

subject to:

- Let $N(v)$ be the variable for 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).

Summary

NETWORK FLOW $\leq_{O(n)}$ LP.

MATCHING $\leq_{O(n)}$ ILP.

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.

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!

Common Problems used in Reductions

NETWORK FLOW

MATCHING

SORTING

LP

ILP

MATRIX MULTIPLICATION

SHORTEST PATHS

Tractable Problems

We would like some convention for distinguishing tractable from intractable problems.

A problem is said to be **<u>tractable</u>** if an algorithm exists to solve it with polynomial time complexity: $O(p(n))$.

- It is said to be **<u>intractable</u>** 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: $O(n^3)$

The technique we will use to classify one group of algorithms is based on two concepts:

1. A special kind of reduction.
2. Nondeterminism.

Decision Problems

(I, SOL) such that SOL(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."

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.

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 $L_1 \leq L_2$.

Example:

- 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.

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.

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.

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 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.

Class $\mathcal{NP}$

The class of languages accepted by a nondeterministic algorithm in polynomial time is called $\mathcal{NP}$.

While there are an **exponential** number of different executions of nd-CLIQUE on a single instance, any one execution requires only **polynomial time** in the size of that instance.

The time complexity of a nondeterministic algorithm is the greatest amount of time required by any **one** of its executions.

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!

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.

$\mathcal{NP}$-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 $\underline{\mathcal{NP}\textbf{-hard}}$ 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 $\underline{\mathcal{NP}\textbf{-complete}}$ if $A \in \mathcal{NP}$ and A is $\mathcal{NP}$-hard.

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.

$$E = (x_5 + x_7 + \bar{x_8} + x_{10}) \cdot (\bar{x_2} + x_3) \cdot (x_1 + \bar{x_3} + x_6).$$

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.

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.

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.

(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 $B \leq_p A$.

$\mathcal{NP}$-completeness Proof Paradigm

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.
   (a) Given an instance X of B, **guess** evidence Y.
   (b) **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 an instance of B.
   - Show that $X \in A$ if and only if $T(X) \in B$.

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.

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.

Proving 3SAT $\mathcal{NP}$-hard

1. Choose $A =$ 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

  $$(y_1+v+w)\cdot(y_1+\bar{v}+w)\cdot(y_1+v+\bar{w})\cdot(y_1+\bar{v}+\bar{w})$$

  where v and w are new variables.

Proving 3SAT $\mathcal{NP}$-hard (cont)

(b) $j = 2$

- Replace $(y_1 + y_2)$ with

$$(y_1 + y_2 + z) \cdot (y_1 + y_2 + \bar{z})$$

where z is a new variable.

(c) $j > 3$

- Relace $(y_1 + y_2 + \cdots + y_j)$ with

$$(y_1 + y_2 + z_1) \cdot (y_3 + \bar{z_1} + z_2) \cdot (y_4 + \bar{z_2} + z_3) \cdots$$

$$(y_{j-2} + \bar{z_{j-4}} + z_{j-3}) \cdot (y_{j-1} + y_j + \bar{z_{j-3}})$$

where $z_1, z_2, \cdots, z_{j-3}$ are new variables.

After appropriate replacements have been 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)

(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.

- 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 $(y_{j-1} + y_{j-2} + \bar{z}_{j-3})$ is false, a contradiction.

We conclude SAT $\leq$ 3SAT and 3SAT is $\mathcal{NP}$-complete.

Tree of Reductions

Reductions go down the tree.

Proofs that each problem $\in \mathcal{NP}$ are straightforward.

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.

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.

$K = m$.

SAT $\leq_p$ CLIQUE (cont)

Example:

$$B = (x_1 + x_2) \cdot (\bar{x_1} + x_2 + x_3).$$

K = 2.

(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.

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.

“Practical” Exponential Problems

What about $O(MN)$ dynamic prog algorithm?

Input size for KNAPSACK is $O(N \log M)$

- Thus $O(MN)$ is exponential in $\log M$.

The dynamic prog algorithm counts through numbers $1, \cdots, M$. Takes exponential time when measured by number of bits to represent M.

If M is “small” ($M = O(p(N))$), 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.

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 $\mathcal{P} = \mathcal{NP}$.

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.

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.

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 (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.

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 (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.

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

$$|V'| \leq 2\text{OPT}$$

because at least one endpoint of every matched edge must be in **any** vertex cover.

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].

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 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 $[6 \text{ of } 1/7 + \epsilon]$
- 3 bins of $[2 \text{ of } 1/3 + \epsilon]$
- 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.

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.

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:

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.

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.

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).

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.

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.

Multiplying Polynomials

$$P = \sum_{i=0}^{n-1} p_i x^i \qquad Q = \sum_{i=0}^{n-1} q_i x^i.$$

Our normal algorithm for computing PQ requires $\Theta(n^2)$ multiplications and additions.

Divide and Conquer:

$$P_1 = \sum_{i=0}^{n/2-1} p_i x^i \qquad P_2 = \sum_{i=n/2}^{n-1} p_i x^{i-n/2}$$

$$Q_1 = \sum_{i=0}^{n/2-1} q_i x^i \qquad Q_2 = \sum_{i=n/2}^{n-1} q_i x^{i-n/2}$$

$$\begin{aligned} PQ &= (P_1 + x^{n/2}P_2)(Q_1 + x^{n/2}Q_2) \\ &= P_1Q_1 + x^{n/2}(Q_1P_2 + P_1Q_2) + x^nP_2Q_2. \end{aligned}$$

Recurrence:

$$\begin{aligned} T(n) &= 4T(n/2) + O(n). \\ T(n) &= \Theta(n^2). \end{aligned}$$

Multiplying Polynomials (cont)

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:

$$\begin{aligned} T(n) &= 3T(n/2) + O(n) \\ &= aT(n/b) + cn^1. \end{aligned}$$

$\log_b a = log_2 3 \approx 1.59$.
$T(n) = \Theta(n^{1.59})$.

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: $\Omega(n^2)$.

Another Approach

Compute:

$$\begin{aligned}
m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\
m_4 &= (a_{11} + a_{12})b_{22} \\
m_5 &= a_{11}(b_{12} - b_{22}) \\
m_6 &= a_{22}(b_{21} - b_{11}) \\
m_7 &= (a_{21} + a_{22})b_{11}
\end{aligned}$$

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.

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 (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.

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?

Boolean Matrix Multiplication

A and B are $n \times n$ matrices with entries 0 or 1.

Substitute OR for addition and AND for multiplication.

Four Russians' Algorithm:

- Assume $\log_2 n$ divides n. Let $k = n/\log_2 n$.
- Partition into strip submatrices, each strip $\log_2 n$ wide.

$$C = A \times B = \sum_{i=1}^{k} A_i \times B_i.$$

Four Russians' Algorithm

Concentrate on $A_i \times B_i$.

- Each row of A_i has $\log n$ entries.
- There are at most $2^{\log n} = n$ distinct row combinations.
- Any row of $A_i \times B_i$ is the sum of a subset of the rows of B_i, depending on the 1s in a row of A_i.
- We can precompute all n possible sums of rows B_i in time $O(n^2)$ using dynamic programming.

  Base case: An empty set of rows.
  sum($\emptyset$) = 0.

  General case: $S' = S \cup \{t\}$
  sum(S') = sum(S) + row(t).

Four Russians' Algorithm (cont)

After precomputation of sums, compute each row of $A_i \times B_i$ by table lookup in time $O(n^2)$.

Analysis:

- Precomputation: $(n/\log n)\Theta(n^2)$.
- Table lookup: $(n/\log n)\Theta(n^2)$.
- Algorithm: $\Theta(n^3/\log n)$.

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?

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.

Representing Polynomials

A vector $\mathbf{a}$ 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**.

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.

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?

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.

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.

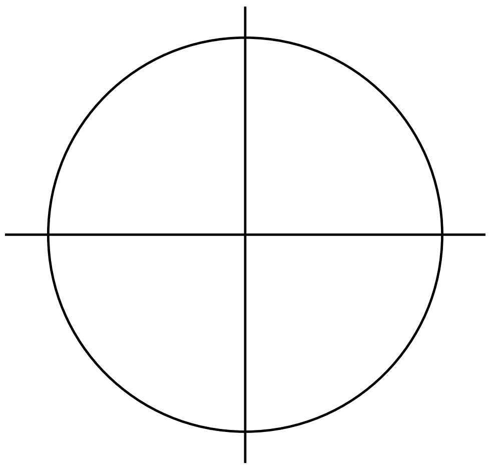
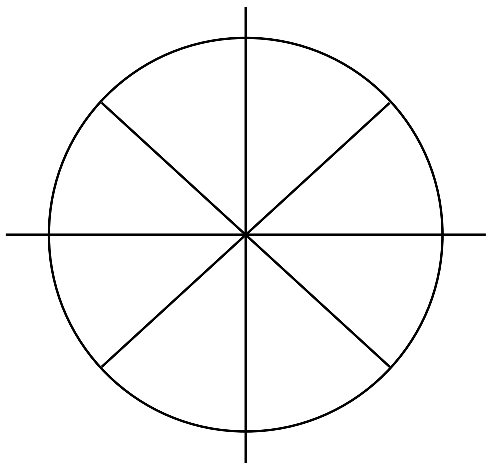
Complex number ω is a **primitive nth root of unity** if

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 $\omega = -i$.

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.

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.

Example: For $n = 8$, $\omega = \sqrt{i}$, $V(\omega) =$

$$\begin{array}{rrrrrrrr}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & \sqrt{i} & i & i\sqrt{i} & -1 & -\sqrt{i} & -i & -i\sqrt{i} \\
1 & i & -1 & -i & 1 & i & -1 & -i \\
1 & i\sqrt{i} & -i & \sqrt{i} & -1 & -i\sqrt{i} & i & -\sqrt{i} \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & -\sqrt{i} & i & -i\sqrt{i} & -1 & \sqrt{i} & -i & i\sqrt{i} \\
1 & -i & -1 & i & 1 & -i & -1 & i \\
1 & -i\sqrt{i} & -i & -\sqrt{i} & -1 & i\sqrt{i} & i & \sqrt{i}
\end{array}$$

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.

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 may not be the same as the best algorithm for p processors, which may 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 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**.

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 $= 640$, speedup < 2, efficiency $< 1/2$.

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 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.

$$\begin{aligned}
\text{Scaled Speedup} &= (\mathcal{S} + \mathcal{P} \times N)/(\mathcal{S} + \mathcal{P}) \\
&= \mathcal{S} + \mathcal{P} \times N \\
&= \mathcal{S} + (1 - \mathcal{S}) \times N \\
&= N + (1 - N) \times \mathcal{S}.
\end{aligned}$$

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.

MIMD Communications

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.

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).

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)

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.

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.

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?

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.
$T(n, n/\log n) = O(\log n)$.
$E(n, n/\log n) = O(1)$.

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.

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 $T(n, s/t) = O(t)$.

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.
- 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 $O(t)$.

Brent's Lemma (cont)

Intuition: You have to split the s work steps across the t time steps somehow; things can't **always** be bad!

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."

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: $O(\log\log n)$.

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.

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.

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.

Better Parallel Prefix

E is the set of all x_is with i even.

If we know $\mathsf{PR}(1, 2i)$ for $1 \le i \le n/2$ then $\mathsf{PR}(1, 2i+1) = \mathsf{PR}(1, 2i) \cdot x_{2i+1}$.

Algorithm:

- Compute in parallel $x_{2i} = x_{2i-1} \cdot x_{2i}$ for $1 \le i \le n/2$.
- 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.

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.

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.

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 (cont)

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;
```

With high probability, each phase completes in $O(n)$ time.

- It is possible to get a really bad random routing, but this is unlikely.
- However, it is very possible for a permutation of messages to be correlated, causing bottlenecks.

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?

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.
- 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).

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.

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 $x_1, x_2, \cdots, x_n$ 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 (cont)

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).

Batcher Sort Complexity

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.

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.