CS 5114: Theory of Algorithms

Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2010

Copyright © 2010 by Clifford A. Shaffer

CS5114: Theory of Algorithms

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: \( \forall n \geq 2, \exists c. P(n) \).
  - Use only positive integers \( n \geq c \) for \( n \).
- Sample \( P(n) \):
  \( n + 1 \leq n^2 \)

Principle of Mathematical Induction

- IF the following two statements are true:
  1. \( P(c) \) is true.
  2. For \( n > c \), \( P(n - 1) \) is true \( \implies \) \( P(n) \) is true.
   ... THEN we may conclude: \( \forall n \geq c, P(n) \).
- The assumption “\( P(n - 1) \) is true” is the induction hypothesis.
- Typical induction proof form:
  - Base case
  - State induction Hypothesis
  - Prove the implication (induction step)
- What does this remind you of?

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

\( P(n) \) is a statement containing \( n \) as a variable.

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

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

The power of induction is that the induction hypothesis “comes for free.” We often try to make the most of the extra information provided by the induction hypothesis.
This is like recursion! There you have a base case and a recursive call that must make progress toward the base case.
Induction Example 1

**Theorem:** Let

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

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


Induction Example 2

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


Induction Example 3

**Theorem:** 2c and 5c 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 guarantee 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:
- $P(c)$
- $P(i), i = 1, 2, \cdots, n - 1 \rightarrow P(n)$,

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

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

---

**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 $H$.

---

Picking what to do induction on can be a problem. Lines?

Regions? How can we “add a region?” We can’t, so try induction on lines.

**Base Case:** $n = 1$. Any line divides the plane into two regions.

**Induction Hypothesis:** It is possible to two-color the regions formed by $n - 1$ lines.

**Induction Step:** Introduce the $n$th line.

This line cuts some colored regions in two. Reverse the region colors on one side of the $n$th line. A valid two-coloring results.

- Any boundary surviving the addition still has opposite colors.
- Any new boundary also has opposite colors after the switch.
Graph Proof (cont)

There are two cases:

- $S(H) \cup \{v\}$ is independent.
  - Then $S(G) = S(H) \cup \{v\}$.
- $S(H) \cup \{v\}$ 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 $\leq 2$.
  - So, set $S(G) = S(H)$.

By Strong Induction, the theorem holds for all $G$.

Fibonacci Numbers

Define Fibonacci numbers inductively as:

$$F(1) = F(2) = 1$$
$$F(n) = F(n-1) + F(n-2), n > 2.$$  

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: $P(n)$: 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 $P(i), i < n$.

Induction Step:

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

Therefore, $P(n)$ 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: $T(\text{input})$
- Growth of time with size of input:
  - Establish an (integer) size $n$ for inputs
  - $n$ numbers in a list
  - $n$ edges in a graph
- Consider time for all inputs of size $n$:
  - Time varies widely with specific input
  - Best case
  - Average case
  - Worst case
- Time complexity $T(n)$ counts steps in an algorithm.

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 $T(n)$ a non-negatively valued function, $T(n)$ is in the set $O(f(n))$ if there exist two positive constants $c$ and $n_0$ such that $T(n) \leq cf(n)$ for all $n \geq n_0$.

Examples:
- $5n + 8 \in O(n)$
- $2n^2 + n \log n \in O(n^2) \in O(n^3 + 5n^2)$
- $2n^2 + n \log n \in O(n^2) \in O(n^3 + n^2)$

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

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

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

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

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

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

We seek the “simplest” and “strongest” $f$.

Big-O is somewhat like $\leq$:
- $n^p \in O(n^q)$ and $n^q \log n \in O(n^p)$, but
  - $n^p \neq n^q \log n$
  - $n^p \in O(n^q)$ while $n^q \log n \notin O(n^p)$

Lemma 3.2

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

Speedups

What happens when we buy a computer 10 times faster?

<table>
<thead>
<tr>
<th>$T(n)$</th>
<th>$n$</th>
<th>$n'$</th>
<th>Change</th>
<th>$n'/n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10n$</td>
<td>1,000</td>
<td>10,000</td>
<td>$n' = 10n$</td>
<td>10</td>
</tr>
<tr>
<td>$20n$</td>
<td>500</td>
<td>5,000</td>
<td>$n' = 10n$</td>
<td>10</td>
</tr>
<tr>
<td>$5n \log n$</td>
<td>250</td>
<td>1,842</td>
<td>$\sqrt{10n} &lt; n' &lt; 10n$</td>
<td>7.37</td>
</tr>
<tr>
<td>$2n^2$</td>
<td>70</td>
<td>223</td>
<td>$n' = \sqrt{10n}$</td>
<td>3.16</td>
</tr>
<tr>
<td>$2^n$</td>
<td>13</td>
<td>16</td>
<td>$n' = n + 3$</td>
<td>---</td>
</tr>
</tbody>
</table>

$n$: 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 $\forall a > 1, (f(n))^c \in O(a^{n})$.

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 $k$, $f(n) \in O(k s(n))$

Assume monotonically growing because larger problems should take longer to solve. However, many real problems have “cyclically growing” behavior.

Is $O(2^{f(n)}) \in O(3^{f(n)})$? Yes, but not vice versa.

$3^n = 1.5^n \times 2^n$ so no constant could ever make $2^n$ bigger than $3^n$ for all $n$. Functional composition
**Other Asymptotic Notation**

\[ \Omega(f(n)) \] 
- lower bound (\( \geq \))

**Definition:** For \( T(n) \) a non-negatively valued function, \( T(n) \) is in the set \( \Omega(g(n)) \) if there exist two positive constants \( c \) and \( n_0 \) such that \( T(n) \geq cg(n) \) for all \( n > n_0 \).

Ex: \( n^2 \log n \in \Omega(n^2) \).

\[ \Theta(f(n)) \] 
- Exact bound (\( = \))

**Definition:** \( g(n) = \Theta(f(n)) \) if \( g(n) \in O(f(n)) \) and \( g(n) \in \Omega(f(n)) \).

**Important:** It is \( \Theta \) if it is both in big-Oh and in \( \Omega \).

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

\[ o(f(n)) \] 
- 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:** \( g(n) \in \omega(f(n)) \) if \( f(n) \in O(g(n)) \).

Ex: \( n^5 \in \omega(n^2) \)

\[ \infty(f(n)) \] 

**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. Sometimes we measure “average case” time complexity.

**Approach:** Estimate number of “steps”

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

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

**Recurrence Relations:** Used for counting steps in recursion.

**Summation: Guess and Test**

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

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

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

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

For \( n = 0 \) we have \( S(n) = 0 \) so \( d = 0 \).
For \( n = 1 \) we have \( a + b + c + 0 = 1 \).
For \( n = 2 \) we have \( 8a + 4b + 2c = 5 \).
For \( n = 3 \) we have \( 27a + 9b + 3c = 14 \).

Solving these equations yields \( a = \frac{1}{3} \), \( b = \frac{1}{2} \), \( c = \frac{1}{6} \).

Now, prove the solution with induction.

Technique 2: Shifted Sums

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

\[ G(n) = \sum_{i=0}^{n} ar^i = a + ar + ar^2 + \ldots + ar^n \]

Shift by multiplying by \( r \).

\[ rG(n) = ar + ar^2 + \ldots + ar^n + ar^{n+1} \]

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

\[ G(n) = \sum_{i=1}^{n} i2^i = 1 \times 2 + 2 \times 2^2 + 3 \times 2^3 + \ldots + n \times 2^n \]

Multiply by 2.

\[ 2G(n) = 1 \times 2^2 + 2 \times 2^3 + 3 \times 2^4 + \ldots + n \times 2^{n+1} \]

Subtract (Note: \( \sum_{i=1}^{n} i2^i = 2^{n+1} - 2 \))

\[ 2G(n) - G(n) = n2^{n+1} - 2^n \cdots 2^2 - 2 \]
\[ G(n) = \frac{n2^{n+1} - 2^{n+1} + 2}{1 - 2} \]
\[ (n - 1)2^{n+1} + 2 \]

Recurrence Relations

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

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

We would like to find a closed form solution for \( T(n) \) such that:
\[
T(n) = \Theta(f(n))
\]

Alternatively, find lower bound
- Not possible for inequalities of form \( T(n) \leq E(T, n) \).

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

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: \( T(n) \leq c_1n^3 = f(n) \)
\[
T(1) = 7 \text{ implies that } c_1 \geq 7
\]

Inductively, assume \( T(n/2) \leq f(n/2) \).
\[
T(n) = 2T(n/2) + 5n^2 \\
\leq 2c_1(n/2)^3 + 5n^2 \\
\leq c_1(n^3/4) + 5n^2 \\
\leq c_1n^3 \text{ if } c_1 \geq 20/3.
\]

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.
\[
T(n) \leq c_2n^2 = g(n) \\
T(1) = 7 \text{ implies } c_2 \geq 7
\]

Inductively, assume \( T(n/2) \leq g(n/2) \).
\[
T(n) = 2T(n/2) + 5n^2 \\
\leq 2c_2(n/2)^2 + 5n^2 \\
= c_2(n^2/2) + 5n^2 \\
\leq c_2n^2 \text{ if } c_2 \geq 10
\]

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

Is this the best possible upper bound?

Note that “finding a closed form” means that we have \( f(n) \) that doesn’t include \( T \).

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

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

For Big-oh, not many choices in what to guess.
\[
7 \times 1^3 = 7
\]

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

No - try something tighter.

Because \( \frac{10}{3}n^2 + 5n^2 = 10n^2 \) for \( n = 1 \), and the right hand side grows faster.

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

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

$$T(n) \leq 2T(n/2) + 5n^2 \quad n \geq 2$$

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

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

$$T(n) \leq T(2^k) \leq 10(2^k)^2$$

$$= 10(2^k/n)^2 n^2 \leq 10(2)^2 n^2$$

$$\leq 40n^2$$

Hence, $T(n) = O(n^2)$ 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.

$$T(n) = 2T(n/2) + 5n^2$$

$$T(1) = 7$$

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

Expanding Recurrences (cont)

$$T(n) = 2T(n/2) + 5n^2$$

$$= 2(2T(n/4) + 5(n/2)^2) + 5n^2$$

$$= 2(2T(n/8) + 5(n/4)^2) + 5(n/2)^2 + 5n^2$$

$$= 2^k T(1) + 2^{k-1} \cdot 5(n/2^{k-1})^2 + 2^{k-2} \cdot 5(n/2^{k-2})^2 + \cdots + 2 \cdot 5(n/2)^2 + 5n^2$$

$$= 7n + 5 \sum_{i=0}^{k-1} n^2/2^i = 7n + 5n^2 \sum_{i=0}^{k-1} 1/2^i$$

$$= 7n + 5n^2(2 - 1/2^{k-1})$$

$$= 7n + 5n^2(2 - 2/n).$$

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

Divide and Conquer Recurrences

These have the form:

$$T(n) = aT(n/b) + cn^k$$

$$T(1) = c$$

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

A problem of size $n$ is divided into $a$ subproblems of size $n/b$, while $cn^k$ is the amount of work needed to combine the solutions.
Divide and Conquer Recurrences (cont)

Expand the sum; \( n = b^r \).

\[
T(n) = a(aT(n/b^2) + c(n/b^3) + cr^h
\]

\[
= a^mT(1) + a^{m-1}c(n/b^{m-1})^h + \cdots + ac(n/b)^k + cr^h
\]

\[
= ca^m \sum_{i=0}^{m} (b^i/a)^i
\]

\( a^m = a^{b^m}n = n^{b^m} \)

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) \( r < 1 \).

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

\[
T(n) = \Theta(a^m) = \Theta(n^{b^m}).
\]

(2) \( r = 1 \).

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

\[
T(n) = \Theta(n^{b^m} \log n) = \Theta(n^k \log n)
\]

D & C Recurrences (Case 3)

(3) \( r > 1 \).

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

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

\[
T(n) = \Theta(a^m r^m) = \Theta(a^m (b^k/a)^m) = \Theta(b^{km}) = \Theta(n^k)
\]

Summary

Theorem 3.4:

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

Apply the theorem:

\[
T(n) = 3T(n/5) + 8n^2.
\]

\( a = 3, \ b = 5, \ c = 8, \ k = 2. \)

\( b^k/a = 25/3. \)

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

We simplify by approximating summations.
Examples

- Mergesort: \( T(n) = 2T(n/2) + n \).
- Binary search: \( T(n) = T(n/2) + 2 \).
- Insertion sort: \( T(n) = T(n-1) + n \).
- Standard Matrix Multiply (recursively):
  \[ T(n) = 8T(n/2) + n^2. \]
  \[ 2^3/8 = 1/2 \text{ so } T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3). \]

Useful log Notation

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

Amortized Analysis

Consider this variation on STACK:

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

... where \(\text{pop}\) removes \(k\) entries from the stack.

“Local” worst case analysis for \(\text{pop}\):

\(O(n)\) for \(n\) elements on the stack.

Given \(m_1\) calls to \(\text{push}\), \(m_2\) calls to \(\text{pop}\):

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 \(\text{push}, \text{pop}\):

Cannot push more elements than get pushed onto the stack.

After many pushes, a single \(\text{pop}\) has high potential.

Once that potential has been expended, it is not available for future \(\text{pop}\) 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 ↔ 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: ∀n ≥ 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]

This is a “meta” algorithm – An algorithm for finding algorithms!
Interval Containment (cont)

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

Example:
- Mark [6, 9] since [6, 9] ⊆ [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, ···
  - 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 if 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 < R, then mark the nth interval.
- Else R ← 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:

Induced subgraph: \( U \) is a subset of \( V \), \( F \) is a subset of \( E \) such that both ends of \( e \in E \) are members of \( U \).

Solution is: \( U = \{1, 3, 4, 5\} \)

**Max Induced Subgraph Solution**

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

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

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

**Celebrity Problem**

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

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

How should we structure the information?

**Celebrity Problem (cont)**

Formulate as an \( n \times n \) boolean matrix \( M \).

\[
M_{ij} = 1 \text{ iff } i \text{ knows } j.
\]

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?

What are ways to select the \( n \)th person?

Efficient Celebrity Algorithm (cont)

Eliminate one person if he is a non-celebrity.
- Strike one row and one column.

\[
\begin{bmatrix}
1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\end{bmatrix}
\]

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

Celebrity Algorithm

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

Analysis:
- \( \Theta(n) \) questions are asked.

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

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 \(S = x_1, x_2, \ldots, x_n\) be the sequence.

Base case: \(n = 1\)

- Either \(x_1 < 0 \Rightarrow \text{sum} = 0\)
- Or \(\text{sum} = x_1\).

Induction Step:

- We know the maximum subsequence \(\text{SUM}(n-1)\) for \(x_1, x_2, \ldots, x_{n-1}\).
- 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 \(x_1, \ldots, x_{n-1}\).

Finding an Algorithm (cont)

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

To get this, we need a stronger induction hypothesis.

Maximum Subsequence Solution

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

Base case:
\[
\text{SUM}(1) = \text{TRAILINGSUM}(1) = \max(0, x_1).
\]

Induction step:
\[
\text{SUM}(n) = \max(\text{SUM}(n-1), \text{TRAILINGSUM}(n-1) + x_n).
\]
\[
\text{TRAILINGSUM}(n) = \max(0, \text{TRAILINGSUM}(n-1) + x_n).
\]
**Maximum Subsequence Solution (cont)**

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

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

How do we find sequence as opposed to sum?

**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 \( S \subseteq \{1, 2, \ldots, n\} \) 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.

So what should the induction hypothesis really be?
Knapsack: New Induction

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

Algorithm Complexity

- Resulting algorithm complexity:
  
  
  \[
  T(n) = 2T(n - 1) + cn & n \geq 2 \\
  T(n) = \Theta(2^n) & \text{by expanding sum.}
  \]

- Alternate: change variable from $n$ to $m = 2^n$.
  
  
  \[
  2T(m/2) + cn^2.
  \]

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

- But, there are only $n(K + 1)$ problems defined.
  - It must be that problems are being re-solved many times by this algorithm. Don’t do that.

Efficient Algorithm Implementation

The key is to avoid re-computing subproblems.

**Implementation:**

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

**Analysis:**

\[
T(n) = \Theta(nK).
\]

Space needed is also $\Theta(nK)$.

Example

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_1$</td>
<td>9</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_2$</td>
<td>2</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_3$</td>
<td>7</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_4$</td>
<td>4</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k_5$</td>
<td>1</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>I</td>
</tr>
</tbody>
</table>

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.

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

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

This form uses $n^2$ because we also need an exponent of $n$ to fit the form of the theorem.

To solve $P(i, k)$ look at entry in the table.

If it is marked, then OK.

Otherwise solve recursively.

Initially, fill in all $P(i, 0)$.

Example: $M(3, 9)$ 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 \( P(5, 10) \).

\[
\begin{align*}
&M(1, 0) \quad M(1, 9) \\
&M(2, 2) \quad M(2, 9) \\
&M(3, 9) \\
&M(4, 9) \\
&M(5, 10)
\end{align*}
\]

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.

Fibonacci Sequence

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

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

Fibonacci Sequence (cont)

- Keep a table

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

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

$$\text{COST}(A \times B) =$$

$$\text{SIZE}(A \times B) =$$

If $C$ is $t \times u$ then

$$\text{COST}((A \times B) \times C) =$$

$$\text{COST}(A \times (B \times C)) =$$

**Order Matters**

Example:

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

$$\text{COST}((A \times B) \times C) =$$

$$\text{COST}(A \times (B \times C)) =$$

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:

- Multiply $M_{n-1} \times M_n$ and replace $M_{n-1}$ in the tree with a subtree.
- 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 $i$th 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 $1 \leq i \leq j \leq n$,

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

So, $c[1,n] = \min_{1 \leq i < n - 1} \text{cost} + c[1,i] + c[i+1,n]$.
Analysis

Base Cases: For \(1 \leq k \leq n\), \(c[k, k] = 0\).

More generally:
\[
c[i, j] = \min_{1 \leq k < j} n_{i-1} r_n + c[i, k] + c[k + 1, j]
\]

Solving \(c[i, j]\) requires \(2(j - i)\) recursive calls.

Analysis:
\[
T(n) = \sum_{k=1}^{n-1} (T(k) + T(n-k)) = 2 \sum_{k=1}^{n-1} T(k)
\]
\[
T(1) = 1
\]
\[
T(n+1) = T(n) + 2T(n) = 3T(n)
\]
\[
T(n) = \Theta(3^n) \text{ Ugh!}
\]

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

Dynamic Programming

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

\[
\begin{array}{cccc}
c[1, 1] & c[1, 2] & \cdots & c[1, n] \\
c[2, 2] & \cdots & \cdots & c[2, n] \\
\vdots & \vdots & \ddots & \vdots \\
c[n, n] & & & \\
\end{array}
\]

Only upper triangle is used.
Fill in table diagonal by diagonal.
\(c[i, i] = 0\).
For \(1 \leq i < j \leq n\),
\[
c[i, j] = \min_{i \leq k < j} n_{i-1} r_n + c[i, k] + c[k + 1, j].
\]

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

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

Actually, since \(j > i\), only about half that needs to be done.

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

For middle diagonal of size \(n/2\), each costs \(n/2\).

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

Each record contains a field called the key.

Linear order: comparison.

The Sorting Problem

Given a sequence of records \( R_1, R_2, ..., R_n \) with key values \( k_1, k_2, ..., k_n \), respectively, arrange the records into any order so that records \( R_s, R_{s+1}, ..., R_n \) have keys obeying the property \( k_s \leq k_{s+1} \leq ... \leq k_n \).

Measures of cost:
- Comparisons
- Swaps

Insertion Sort

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

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.
- **Proof:**
  - For any permutation \( P \), and any pair of positions \( i \) and \( j \), the relative order of \( i \) and \( j \) must be wrong in either \( P \) or the inverse of \( P \).
  - Thus, the total number of swaps required by \( P \) and the inverse of \( P \) MUST be
  \[
  \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.
  \]

Quicksort

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

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

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

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

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

Cost for Quicksort

Best Case: Always partition in half.

Worst Case: Bad partition.

Average Case:

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

Optimizations for Quicksort:

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

Think about when the partition is bad. Note the FindPivot function that we used is pretty good, especially compared to taking the first (or last) value.

Also, think about the distribution of costs: Line up all the permutations from most expensive to cheapest. How many can also, think about the distribution of costs: Line up all the permutations from most expensive to cheapest. How many can very few of the expensive ones.

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

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

Since the two halves of the summation are identical,

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

Multiplying both sides by \( n \) yields

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

This is a “recurrence with full history”.

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

- Partition: Cost \( n \)
- FindPivot: Cost \( c \)
- Do the recursion: Cost dependent on the pivot’s final position.

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

Since the two halves of the summation are identical,

\[ \sum_{i=0}^{n-1} f(i) = \sum_{i=0}^{n-1} f(i) \]

\[ (n+1)f(n+1) = (n+1)n + 2 \sum_{i=1}^{n} f(i) \]

\[ (n+1)f(n+1) - nf(n) = 2n + 2f(n) \]

\[ (n+1)f(n+1) = 2n + (n+2)f(n) \]

\[ f(n+1) = \frac{2n}{n+1} + \frac{n+2}{n+1} f(n). \]

Note that \( \frac{2n}{n+1} \leq 2 \) for \( n \geq 1 \).

Expand the recurrence to get:

\[ f(n+1) \leq 2 + \frac{n+2}{n+1} f(n) \]

\[ = 2 + \frac{n+2}{n+1} \left( 2 + \frac{n+1}{n} f(n-1) \right) \]

\[ = 2 + \frac{n+2}{n+1} \left( 2 + \frac{n+1}{n} \left( 2 + \frac{n}{n-1} f(n-2) \right) \right) \]

\[ = 2 + \frac{n+2}{n+1} \left( 2 + \cdots + \frac{4}{3} \left( 2 + \frac{1}{2} f(1) \right) \right) \]

\[ = \Theta(n \log n). \]
**Mergesort**

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

Mergesort Implementation (1)

Mergesort is tricky to implement.

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

Mergesort Implementation (2)

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

Mergesort cost:
Mergesort is good for sorting linked lists.

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) 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**

```cpp
def siftdown(pos):
    if not isLeaf(pos):
        j = leftchild(pos)
        if (j < (n-1)) and (Heap[j].key < Heap[j+1].key):
            j += 1  # j now index of child with > value
        if Heap[pos].key >= Heap[j].key:
            return
        swap(Heap, pos, j)
        siftdown(j)
```

**BuildHeap**

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

```
def buildheap():
    for i in range(n//2-1, -1, -1):
        siftdown(i)
```

Cost for heap construction:

\[
\sum_{i=1}^{\log n} \left( \frac{n}{2^i} \right) \approx n.
\]

**Heapsort**

Heapsort uses a max-heap.

```
deck heapsort(Elem* A, int n) {
    heap H(A, n, n);  // Build the heap
    for (int i=n/2-1; i>=0; i--) siftdown(i);  // Now sort
    H.removemax();  // Value placed at end of heap
}
```

Cost of Heapsort:

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

**Compare Heapsort to sorting with BST:**
- BST is expensive in space (overhead), potential bad balance, BST does not take advantage of having all records available in advance.
- Heap is space efficient, balanced, and building initial heap is efficient.
Heapsort Example (1)

Heapsort Example (2)

Binsort

A simple, efficient sort:

\[
\text{for } (i=0; i<n; i++) \\
\quad B[\text{key}(A[i])] = A[i];
\]

Ways to generalize:

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

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

Cost:

Radix Sort

The simple version only works for a permutation of 0 to \( n - 1 \), but it is truly \( O(n! \) not \( O(n^2) \).

Support duplicates (i.e., larger key space).

Cost might look like \( \Theta(n \times \text{Maxkeydown}) \).

Oops! It is actually \( \Theta(n \times \text{Maxkeydown}) \).

Maxkeydown could be \( O(n^2) \) or worse.
Radix Sort Algorithm (1)

```c
void radix(Elem * A, Elem * B, int n, int k, int r, int* count) {
    // Count[i] stores number of records in bin[i]
    for (int i=0, rtok=1; i<k; i++, rtok *=r) {
        for (int j=0; j<r; j++) count[j] = 0; // Init
        // Count # of records for each bin this pass
        for (j=0; j<n; j++)
            count[(key(A[j])/rtok)%r]++;
        //Index B: count[j] is index of j’s last slot
        for (j=1; j<r; j++)
            count[j] = count[j-1]+count[j];
    }
    // Put recs into bins working from bottom
    //Bins fill from bottom so j counts downwards
    for (j=n-1; j>=0; j--)
        B[--count[(key(A[j])/rtok)%r]] = A[j];
    for (j=0; j<n; j++) A[j] = B[j]; // Copy B->A
}
```

Cost: $\Theta(nk + rk)$. How do $n$, $k$, and $r$ relate?

Radix Sort Algorithm (2)

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

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 $\Omega(r \log n)$ levels deep.
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 of block I/O operations.
- Most workstations today must do all sorting on a single disk drive.

**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**

Can efficiently read block sequentially when:

1. Adjacent logical blocks of file are physically adjacent on disk
2. No competition for I/O head.

The algorithm presented here is geared toward these conditions.
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.

But, this is not usually done.
1. It is expensive (random access to all records).
2. If there are multiple keys, there is no “correct” order.

Internal → External Sort

Why not just use an internal sort on a large virtual memory?
- Quicksort requires random access to the entire set of records.
- Mergesort is more geared toward sequential processing of records.
  - Process $n$ elements in $\Theta(\log n)$ passes.
- Better: Modify Mergesort for the purpose.

Try #1: Simple Mergesort

- Split the file into two files.
- Read in a block from each file.
- Take first record from each block, output them in sorted order.
- Take next record from each block, output them to a second file in sorted order.
- Repeat until finished, alternating between output files.
- Repeat steps 2-5, except this time the input files have groups of two sorted records that are merged together.
- 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:
  - Break the file into initial runs.
  - Merge the runs together into a single sorted run.

Yes, each pass is sequential.
But competition for I/O head eliminates advantage of sequential processing.
We could read in a block (or several blocks) and do an in-memory sort to generate large initial runs.
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

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

Replacement Selection (cont)

Example of Replacement Selection
Benefit from Replacement Selection

- Double buffer to overlap input, processing, 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 amount $S$ snow on the track. Some snow falls 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 2$S$ 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,
  - 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.

Multiway Merge (cont)
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 \( 2^b \) 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 \( i, 1 \leq i \leq m \), \( a_{k+i} = b_i \).

String Matching Example

\[
A = xyxxyxxykxyxkxyxy \\
B = xyxxyxxy
\]

1: \( \times \) 2: \( \times \) 3: \( \times \) 4: \( \times \) 5: \( \times \) 6: \( \times \) 7: \( \times \) 8: \( \times \) 9: \( \times \) 10: \( \times \) 11: \( xyxxy \\
O(mn) \) comparisons in worst case.
String Matching Worst Case

Brute force isn’t too bad for small patterns and large alphabets. However, try finding: \texttt{yyyyyx} in: \texttt{yyyyyyyyyyyyyyyx}

Alternatively, consider searching for: \texttt{xyyyyy}

Finding a Better Algorithm

Find \( B = xyxyyxyxyxx \) in \( A = xyxxyxyxyyxyxyxyyxyxyxx \)

When things go wrong, focus on what the prefix might be.

- \( xyxxyxyxyyxyxyxyyxyxyxx \) -- 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 \( B \).
  - \( \text{next}(i) = \) the maximum \( j \) (\( 0 < j < i - 1 \)) such that \( b_i b_{i-j} \cdots b_{i-1} = B(j) \), and 0 if no such \( j \) exists.
  - We define \( \text{next}(1) = -1 \) to distinguish it.
  - \( \text{next}(2) = 0 \). Why?

Computing the table

\[
\begin{array}{cccccccccccc}
B = & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
& x & y & x & y & y & x & y & x & y & x & x \\
& -1 & 0 & 0 & 1 & 2 & 0 & 1 & 2 & 3 & 4 & 3 \\
\end{array}
\]

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

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

In the second example, we can quickly reject a position - no backtracking.
How to Compute Table?

- By induction.
- **Base cases:** `next(1)` and `next(2)` already determined.
- **Induction Hypothesis:** Values have been computed up to `next(i - 1)`.
- **Induction Step:** For `next(i)`: at most `next(i - 1) + 1`.
  - When? `b_{i-1} = b_{next(i-1)+1}`.
  - That is, largest suffix can be extended by `b_{i-1}`.
  - If `b_{i-1} ≠ 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_i`, then only `i` mismatches are possible.
  - But, for each mismatch, we had to go forward a character to get to `b_i`.
  - Since there are always `n` forward moves, the total cost is `O(n)`.

Example Using Table

```
<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
</table>
B | x | y | x | y | y | x | y | y | x | x | x |
-1 | 0 | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 4 | 3 |
A | x | y | x | y | x | y | y | x | y | y | y | y | x | x
  | y | y | next(4) = 1, compare B(2) to this
  | -x | y | next(2) = 0, compare B(1) to this
  | x | y | y | next(5) = 2, compare to B(3)
  | -x | y | y | y | x | y | x | x | next(11) = 3
  | -x | -y | x | y | 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 $S = x_1, x_2, \ldots, x_n$ of elements, $x_i$ has rank $k$ in $S$ if $x_i$ is the $k$th 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?

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 $q_1, q_2, p_1, p_2$.

$$T(2n) = 2T(n) + 2; T(2) = 1.$$  
$$T(n) = 3n/2 - 2.$$  

- 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 
  \[ \text{compare } p_2 \text{ and } q_1 \text{ [ignore } q_2] \]
- Else 
  \[ \text{compare } p_1 \text{ and } q_2 \text{ [ignore } p_2] \]
- 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 
  \[ \text{new}_1 = p_1; C_{\text{new}} = C_P \cup q_1 \]
- Else 
  \[ \text{new}_1 = q_1; C_{\text{new}} = C_Q \cup p_1 \]
- At end, look through set of candidates that remains.
- What is size of \( C \)?
- **Total cost:**

### Lower Bound for Second Best

At least \( n - 1 \) values must lose at least once.
- At least \( n - 1 \) compares.

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

There must be at least \( n + k - 2 \) comparisons.

How low can we make \( k \)?

### Adversarial Lower Bound

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

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

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

What should the algorithm do?

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

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

\[ \text{K^{th Smalllest Element}} \]

\textbf{Problem}: Find the \( k \)-th smallest element from sequence \( S \).

(Also called \textbf{selection}.)

\textbf{Solution}: Find min value and discard \( k \) times.
- If \( k \) is large, find \( n - k \) max values.

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

\[ \text{Better K^{th Smallest Algorithm}} \]

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.

\[ \text{Probabilistic Algorithms} \]

All algorithms discussed so far are \textbf{deterministic}.

\textbf{Probabilistic} algorithms include steps that are affected by \textbf{random} events.

Example: Pick one number in the upper half of the values in a set.
- Pick maximum: \( n - 1 \) comparisons.
- Pick maximum from just over 1/2 of the elements: \( n/2 \) comparisons.

Can we do better? Not if we want a \textbf{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 $S_1, S_2, \ldots, S_k$ 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 $1 - 2^{-k}$.
  - If it doesn't work, try again!

**Transforming to Deterministic Alg**

- First, generalize the problem:
  - Let $S_1, S_2, \ldots, S_k$ 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:
  $$F(n) = 2^r / 2^{S_i}$$
- If $F(n) < 1$, then there exists a coloring that solves the problem.
- **Strategy:** Color one element of $S$ at a time, always choosing color that gives lower probability of failure.

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

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

This principle is why, for example, the Skip List data structure has much more reliable performance than a BST. The BST’s performance depends on the input data. The Skip List’s performance depends entirely on chance. For random data, the two are essentially identical. But you can’t trust data to be random.

$k, r$ picked to make calculation easy. Note the sets are distinct, not disjoint. So just make sure that $r$ is “big enough” compared with $k$. There is always a valid coloring, since $r$ is chosen “big enough.”

Probability $1/2^r$ that a subset is all red, $1/2^r$ that a subset is all blue, so probability $1/2^{r-1}$ that the subset is all one color. There are $k$ chances for this to happen.

For example, $S_i = 3$. 1/8 all red. 1/8 all blue. 1/4 failure.

We selected $r$ and $k$ so that this must be true.
Deterministic Algorithm

- Let \( S = \{x_1, x_2, \ldots, x_n\} \).
- Suppose we have colored \( x_{j+1}, x_{j+2}, \ldots, x_n \) 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:
  - whether \( x_j \) is a member of \( S_i \).
  - whether \( S_i \) contains a blue element.
  - whether \( S_i \) contains a red element.
  - the number of elements in \( S_i \) yet to be colored.

\[ P_R(i,j) = P(i,j) \]

Deterministic Algorithm (cont)

Result:
- If \( x_j \) is not a member of \( S_i \), probability is unchanged.
- \( P_R(i,j) = P(i,j) \).
- If \( S_i \) contains a blue element, then \( P_R(i,j) = 0 \).
- If \( S_i \) contains no blue element and some red elements, then \( P_R(i,j) = 2P(i,j) \).
- 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 \( P_R(i,j) \), the probability of failure for set \( S_i \) if \( x_j \) is colored blue.
- Sum the failure probabilities as follows:
  \[
  F_R(j) = \sum_{i=1}^{k} P_R(i,j) \\
  F_B(j) = \sum_{i=1}^{k} P_R(i,j)
  \]
- Claim: \( F_R(n-1) + F_B(n-1) \leq 2F(n) \).
  \[
  P_R(i,j) + P_B(i,j) \leq 2P(i,j) 
  \]

Deterministic Algorithm (cont)

- 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 \( j = n \) down to 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

- Most computers systems use a deterministic algorithm to select pseudorandom numbers.
- **Linear congruential method:**
  - Pick a seed \( r(1) \). Then,
    \[
    r(i) = (r(i - 1) \times b) \mod t.
    \]
- 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:
\[
\begin{align*}
  r(i) &= 6r(i - 1) \mod 13 = \\
    &\quad \cdots \quad 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1 \cdots \\
  r(i) &= 7r(i - 1) \mod 13 = \\
    &\quad \cdots \quad 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1 \cdots \\
  r(i) &= 5r(i - 1) \mod 13 = \\
    &\quad \cdots \quad 1, 5, 12, 8, 1 \cdots \\
    &\quad \cdots \quad 2, 10, 11, 3, 2 \cdots \\
    &\quad \quad \cdots \quad 4, 7, 9, 6, 4 \cdots \\
    &\quad \quad \quad \cdots \quad 0, 0 \cdots
\end{align*}
\]

The last one depends on the start value of the seed.
Suggested generator: \( r(i) = 16807r(i - 1) \mod 2^{31} - 1 \)

Graph Algorithms

Graphs are useful for representing a variety of concepts:
- Data Structures
- Relationships
- Families
- Communication Networks
- Road Maps
- 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
- Labels for edges vs. weights for edges
- Multiple edges, loops
- Cycle, Circuit, path, simple path, tours
- Bipartite, acyclic, connected
- Rooted tree, unrooted tree, free tree
A Tree Proof

- **Definition:** A 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:

- Eulerian tours
  - Traverse each edge exactly once
- Depth-first search
  - Keeps vertices on a stack
- Breadth-first search
  - Keeps vertices on a queue

Eulerian Tours

A circuit that contains every edge exactly once.

Example:

```
        f
       /|
      a/b  c/d
```

Tour: b a f c d e.

Example:

```
        f
       /|
      a/b  c/d
```

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.

Why no tour? Because some vertices have odd degree. All even nodes is a necessary condition. Is it sufficient?
**Depth First Search**

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

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

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

---

**Depth First Search Example**

The directions are imposed by the traversal. This is the Depth First Search Tree.

**DFS Tree**

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

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

**Proof:** Consider the first time an edge $(v, w)$ is examined, with $v$ the current vertex.
- If $w$ is unmarked, then $(v, w)$ is in $T$.
- If $w$ is marked, then $w$ has a smaller DFS number than $v$ AND $(v, w)$ is an unexamined edge of $w$.
- Thus, $w$ is still on the stack. That is, $w$ is on a path from $v$.

**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 vertex's neighbors before going deeper in tree.

Breadth First Search Algorithm

```c
void BFS(Graph G, int start) {
    Queue Q(G.n());
    Q.enqueue(start);
    G.setMark(start, VISITED);
    while (!Q.isEmpty()) {
        int v = Q.dequeue(); // Take appropriate action
        for (Edge w = each neighbor of v)
            if (G.getMark(G.v2(w)) == UNVISITED) {
                G.setMark(G.v2(w), VISITED);
                Q.enqueue(G.v2(w));
            }
        PostVisit(G, v); // Take appropriate action
    }
}
```

Breadth First Search Example

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.

Topological Sort Algorithm

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

void tophelp(Graph G, int v) { // Helper function
    G.setMark(v, VISITED);
    for (Edge w = each neighbor of v)
        if (G.getMark(G.v2(w)) == UNVISITED)
            tophelp(G, G.v2(w));
    printout(v); // PostVisit for Vertex v
}
```

Queue-based Topological Sort

```java
void topsort(Graph G) { // Top sort: Queue
    Queue Q(G.n()); int Count[G.n()];
    for (int v=0; v<G.n(); v++) Count[v] = 0;
    for (v=0; v<G.n(); v++) // Process every edge
        for (Edge w each neighbor of v)
            Count[G.v2(w)]++; // Add to v2’s count
    for (v=0; v<G.n(); v++) // Initialize Queue
        if (Count[v] == 0) Q.enqueue(v);
    while (!Q.isEmpty()) { // Process the vertices
        int v = Q.dequeue(); // PreVisit for v
        for (Edge w = each neighbor of v)
            if (Count[G.v2(w)]==0) 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.

w(A, D) = 20; d(A, D) = 10 (through ACBD).

Single Source Shortest Paths

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

Try 1: Visit all vertices in some order, compute shortest paths for all vertices seen so far, then add the shortest path to next vertex x.

Problem: Shortest path to a vertex already processed might go through x.

Solution: Process vertices in order of distance from s.

Dijkstra's Algorithm Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>0</td>
<td>\infty</td>
<td>\infty</td>
<td>\infty</td>
<td>\infty</td>
</tr>
<tr>
<td>Process A</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>20</td>
<td>\infty</td>
</tr>
<tr>
<td>Process C</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>Process B</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Process D</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Process E</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
</tbody>
</table>

Dijkstra's Algorithm: Array (1)

```c
void Dijkstra(Graph G, int s) { // Use array
    int D[G.n()];
    for (int i=0; i<G.n(); i++)  // Initialize
        D[i] = INFINITY;
    D[s] = 0;
    for (i=0; i<G.n(); i++) { // Process vertices
        int v = minVertex(G, D);
        if (D[v] == INFINITY) return; // Unreachable
        G.setMark(v, VISITED);
        for (Edge w = each neighbor of v)
            if (D[G.v2(w)] > (D[v] + G.weight(w)))
                D[G.v2(w)] = D[v] + G.weight(w);        }
} `
Dijkstra's Algorithm: Array (2)

```c
void Dijkstra(Graph G, int s) { // priority queue
    int key(Elem x) { return x.dist; }
    class Elem { public: int vertex, dist; }
    int D[G.n()]; Elem E[G.e()];
    int v; Elem temp;
    D[s] = 0; for (i=0; i<G.n(); i++) D[i] = INFINITY;
    total cost: $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$.
    Approach 1: Scan the table on each pass for closest vertex.
    if (G.getMark(i) == UNVISITED) { v = i; break; }
    for (i++; i<G.n(); i++) // Find smallest D val
    for (Edge w = each neighbor of v)
        if (D[G.v2(w)] > (D[v] + G.weight(w))) {
            temp.vertex = G.v2(w);
            temp.dist = D[G.v2(w)];
            D[G.v2(w)] = D[v] + G.weight(w);
            H.insert(temp); // Insert new distance
    return v;
}
```

Dijkstra's Algorithm: Priority Queue (1)

```c
class Elem { public: int vertex, dist; }
int key(Elem x) { return x.dist; }
void Dijkstra(Graph G, int s) { // priority queue
    int temp; int D[G.n()]; Elem E[G.e()];
    temp.dist = 0; temp.vertex = s; E[0] = temp;
    D[s] = 0;
    for (i=0; i<G.n(); i++) { // Get distances
        do { temp = H.removemin(); v = temp.vertex; }
        while (G.getMark(v) == VISITED);
        if (D[v] == INFINITY) return; // Unreachable
        G.setMark(v, VISITED);
        for (i++; i<G.n(); i++) // Find smallest D val
            if ((G.getMark(i)==UNVISITED) && (D[i]<D[v]))
                v = i;
        for (Edge w = each neighbor of v)
            if (D[G.v2(w)] > (D[v] + G.weight(w))) {
                temp.vertex = G.v2(w);
                temp.dist = D[G.v2(w)];
                D[G.v2(w)] = D[v] + G.weight(w);
                H.insert(temp); // Insert new distance
        return v;
    }
```

Dijkstra's Algorithm: Priority Queue (2)

```c
Dijkstra's Algorithm: Array (2)
```

All Pairs Shortest Paths

- For every vertex $u, v \in V$, calculate $d(u, v)$.
- Could run Dijkstra's Algorithm $|V|$ times.
- Better is Floyd's Algorithm.
- Define a $k$-path from $u$ to $v$ to be any path whose intermediate vertices all have indices less than $k$.

Multiple runs of Dijkstra’s algorithm Cost: $|V||E|\log |V| = |V|^{2+\log |V|}$ for dense graph.

The issue driving the concept of “k paths” is how to efficiently check all the paths without computing any path more than once.

0,3 is a 0-path. 2,0,3 is a 1-path. 0,2,3 is a 3-path, but not a 2 or 1 path. Everything is a 4 path.
**Floyd's Algorithm**

```c
void Floyd(Graph G) { // All-pairs shortest paths
    int D[G.n()][G.n()]; // Store distances
    for (int i=0; i<G.n(); i++) // Initialize D
        for (int j=0; j<G.n(); j++)
            D[i][j] = G.weight(i, j);
    for (int k=0; k<G.n(); k++) // Compute k paths
        for (int i=0; i<G.n(); i++)
            for (int j=0; j<G.n(); j++)
                if (D[i][j] > (D[i][k] + D[k][j]))
                    D[i][j] = D[i][k] + D[k][j];
}
```

**Minimum Cost Spanning Trees**

Minimum Cost Spanning Tree (MST) Problem:
- Input: An undirected, connected graph G.
- Output: The subgraph of G that
  - has minimum total cost as measured by summing the values for all of the edges in the subset, and
  - 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 \((v_1, v_2)\).
Prim's MST Algorithm (1)

```c
void Prim(Graph G, int s) { // Prim's MST alg
    int D[G.n()]; int V[G.n()]; // Distances
    for (int i=0; i<G.n(); i++) // Initialize
        D[i] = INFINITY;
    D[s] = 0;
    for (i=0; i<G.n(); i++) { // Process vertices
        int v = minVertex(G, D);
        G.setMark(v, VISITED);
        if (v != s) AddEdgetoMST(V[v], v);
        if (D[v] == INFINITY) return; //v unreachable
        for (Edge w = each neighbor of v)
            if (D[G.v2(w)] > G.weight(w)) {
                D[G.v2(w)] = G.weight(w); // Update dist
                V[G.v2(w)] = v; // who came from
            }
    }
}
```

Prim's MST Algorithm (2)

```c
int minVertex(Graph G, int * D) {
    int v; // Initialize v to any unvisited vertex
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED)
            { v = i; break; }
    for (i=0; i<G.n(); i++) // Find smallest value
        if ((G.getMark(i)==UNVISITED) && (D[i]<D[v]))
            v = i;
    return v;
}
```

This is an example of a greedy algorithm.

Alternative Prim's Implementation (1)

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

```c
void Prim(Graph G, int s) { // Prim's MST alg
    int v; // The current vertex
    int D[G.n()]; // Distance array
    int V[G.n()]; // Who's closest
    Elem temp;
    Elem E[G.e()]; // Heap array
    temp.distance = 0; temp.vertex = s;
    E[0] = temp; // Initialize heap array
    heap H(E, 1, G.e()); // Create the heap
    for (int i=0; i<G.n(); i++) D[i] = INFINITY;
    D[s] = 0;
    do { temp = H.removemin(); v = temp.vertex; }
        while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v);
    if (D[v] == INFINITY) return; // Unreachable
    for (Edge w = each neighbor of v)
        if (D[G.v2(w)] > G.weight(w)) { // Update D
            D[G.v2(w)] = G.weight(w);
            V[G.v2(w)] = v; // Who came from
            temp.distance = D[G.v2(w)];
            temp.vertex = G.v2(w);
            H.insert(temp); // Insert dist in heap
        }
}
```

Alternative Prim's Implementation (2)

```c
for (i=0; i<G.n(); i++) { // Now build MST
    do { temp = H.removemin(); v = temp.vertex; }
        while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v);
    if (D[v] == INFINITY) return; // Unreachable
    for (Edge w = each neighbor of v)
        if (D[G.v2(w)] > G.weight(w)) { // Update D
            D[G.v2(w)] = G.weight(w);
            V[G.v2(w)] = v; // Who came from
            temp.distance = D[G.v2(w)];
            temp.vertex = G.v2(w);
            H.insert(temp); // Insert dist in heap
        }
}
```
Kruskal’s MST Algorithm (1)

Kruskål(Graph G) { // Kruskal’s MST algorithm
    Gentree A(G.n()); // Equivalence class array
    Elem E[G.e()]; // Array of edges for min-heap
    int edgecnt = 0;
    for (int i=0; i<G.n(); i++) { // Put edges into E
        G.isEdge(w); w = G.next(w)) {
            E[edgecnt].weight = G.weight(w);
            E[edgecnt++].edge = w;
        }
    }
    heap H(E, edgecnt, edgecnt); // Heapify edges
    int numMST = G.n(); // Init w/ n equiv classes
    for (i=0; numMST>1; i++) { // Combine
        union A(v, u); // Combine
        H.removemin(); // Next cheap edge
        G.isEdge(w); w = G.next(w)) {
            E[edgecnt++].weight = G.weight(w);
            E[edgecnt].edge = w;
        }
    }
}

Kruskal’s MST Algorithm (2)

for (i=0; numMST>1; i++) { // Combine
    Elem temp = H.removemin(); // Next cheap edge
    int v = G.v1(w); int u = G.v2(w);
    if (A.differ(v, u)) { // If different
        A.UNION(v, u); // Combine
        AddEdgetoMST(G.v1(w), G.v2(w)); // Add
        numMST--; // Now one less MST
    }
}

How do we compute function MSTof(v)?
Solution: UNION-FIND algorithm (Section 4.3).

Kruskal’s Algorithm Example
Total cost: Θ(|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 includes every vertex.

An example:
(1-3) is a matching.
(1-3) (5, 4) is both maximal and maximum.
Take away the edge (5-4). Then (3, 2) would be maximal but not a maximum matching.
**Very Dense Graphs (1)**

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

**Proof:** Suppose that $G$ does not contain a perfect matching.
- Let $M \subseteq E$ be a max matching. $|M| < n$.
- Therefore, there must be two unmatched vertices $v_1, v_2$ that are not adjacent.
- Every vertex adjacent to $v_1$ or to $v_2$ is matched.
- Let $M' \subseteq M$ be the set of edges involved in matching the neighbors of $v_1$ and $v_2$.
- There are $\geq 2n$ edges from $v_1$ and $v_2$ to vertices covered by $M'$, but $|M'| < n$.

**Very Dense Graphs (2)**

**Proof:** (continued)
- Thus, some edge of $M'$ is adjacent to 3 edges from $v_1$ and $v_2$.
- Let $(u_1, u_2)$ be such an edge.
- Replacing $(u_1, u_2)$ with $(v_1, u_2)$ and $(v_2, u_1)$ results in a larger matching.
- Theorem proven by contradiction.

**Generalizing the Insight**

- $v_1, u_2, u_1, v_2$ is a path from an unmatched vertex to an unmatched vertex such that alternate edges are unmatched and matched.
- In one step, switch unmatched and matched edges.
- Let $G = (V, E)$ be an undirected graph and $M \subseteq E$ a matching.
- An alternating path $P$ goes from $v$ to $u$, consists of alternately matched and unmatched edges, and both $v$ and $u$ are not in the match.

**Matching Example**

1, 2, 3 is NOT an alternating path (it does not start with an unmatched vertex).

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

Observation: If a matching has an alternating path, then the size of the matching can be increased by one by switching matched and unmatched edges along the alternating path.
The Alternating Path Theorem (1)

**Theorem:** A matching is maximum if and only if it has no alternating paths.

**Proof:**
- Clearly, if a matching has alternating paths, then it is not maximum.
- Suppose $M$ is a non-maximum matching.
- Let $M'$ be any maximum matching. Then, $|M'| > |M|$.
- Let $M \oplus M'$ be the symmetric difference of $M$ and $M'$.
- $G' = (V, M \oplus M')$ is a subgraph of $G$ having maximum degree $\leq 2$.

The Alternating Path Theorem (2)

**Proof:** (continued)
- Therefore, the connected components of $G'$ are either even-length cycles or a path with alternating edges.
- Since $|M'| > |M|$, there must be a component of $G'$ that is an alternating path having more $M'$ edges than $M$ edges.

Bipartite Matching

- A **bipartite graph** $G = (U, V, E)$ consists of two disjoint sets of vertices $U$ and $V$ together with edges $E$ such that every edge has an endpoint in $U$ and an endpoint in $V$.
- Bipartite matching naturally models a number of assignment problems, such as assignment of workers to jobs.
- Alternating paths will work to find a maximum bipartite matching. An alternating path always has one end in $U$ and the other in $V$.
- If we direct unmatched edges from $U$ to $V$ and matched edges from $V$ to $U$, then a directed path from an unmatched vertex in $U$ to an unmatched vertex in $V$ is an alternating path.

Bipartite Matching Example

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

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

Naive algorithm: Find a maximal matching (greedy algorithm).

For each vertex:
- Do a DFS or other search until an alternating path is found.
- Use the alternating path to improve the match.

$|V|(|V| + |E|)$
Algorithm for Maximum Bipartite Matching

Construct BFS subgraph from the set of unmatched vertices in $U$ until a level with unmatched vertices in $V$ is found.

Greedily select a maximal set of disjoint alternating paths.

Augment along each path independently.

Repeat until no alternating paths remain.

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

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 Graph

Network Flow Definitions

A flow in a network is a function $f : V \times V \to R$ with the following properties.

(i) Skew Symmetry:

$$\forall v, w \in V, \quad f(v, w) = -f(w, v).$$

(ii) Capacity Constraint:

$$\forall v, w \in V, \quad f(v, w) \leq c(v, w).$$

If $f(v, w) = c(v, w)$ then $(v, w)$ is saturated.

(iii) Flow Conservation:

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

Order doesn’t matter. Find a path, remove its vertices, then repeat. Augment along the paths independently since they are disjoint.
**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.

A **min cut** is a cut of minimum capacity.

**Cut Flows**

For any flow \( f \), the flow across a cut is:

\[ f(X, X') = \sum_{v \in X, w \in X'} c(v, w). \]

**Lemma:** For all flows \( f \) and all cuts \((X, X')\), \( f(X, X') = |f| \).

- Clearly, the flow out of \( s \) = \( |f| \) = the flow into \( t \).
- It can be proved that the flow across every other cut is also \( |f| \).

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

---

**Residual Graph**

Given any flow \( f \), the residual capacity of the edge is

\[ res(v, w) = c(v, w) - f(v, w) \geq 0. \]

**Residual graph** is a network \( R = (V, E_R) \) where \( E_R \) contains edges of non-zero residual capacity.

\( R \) is the network after \( f \) has been subtracted.

Saturated edges do not appear.

Some edges have larger capacity than in \( G \).
Observations

- Any flow in $R$ can be added to $F$ to obtain a larger flow in $G$.
- In fact, a max flow $f'$ in $R$ plus the flow $f$ (written $f + f'$) is a max flow in $G$.
- 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.

Max-flow Min-cut Theorem (2)

(ii) $\Rightarrow$ (iii):

- Suppose $f$ has no augmenting path in $R$.
- Let $X$ be the subset of $V$ reachable from $s$ and $X' = V - X$.
- Then $s \in X$, $t \in X'$, so $(X, X')$ is a cut.
- $\forall v \in X$, $w \in X'$, $res(v, w) = c(v, w) - f(v, w) = 0$.
- $f(X, X') = \sum_{v \in X, w \in X'} f(v, w) = \sum_{v \in X, w \in X'} c(v, w) = c(X, X')$.
- By Lemma, $|f| = c(X, X')$ and $(X, X')$ is a min cut.

Max-flow Min-cut Theorem (3)

(iii) $\Rightarrow$ (i):

- Let $f$ be a flow such that $|f| = c(X, X')$ for some (min) cut $(X, X')$.
- By Lemma, all flows $f'$ satisfy $|f'| \leq c(X, X') = |f|$.

Thus, $f$ is a max flow.
Max-flow Min-cut Corollary

**Corollary:** The value of a max flow equals the capacity of a min cut.

This suggests a strategy for finding a max flow.

\[ R = G; f = 0; \]
\[ \text{repeat} \]
\[ \text{find a path from } s \text{ to } t \text{ in } R; \]
\[ \text{augment along path to get a larger flow } f; \]
\[ \text{update } R \text{ for new flow}; \]
\[ \text{until } R \text{ has no path } s \text{ to } t. \]

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 \( R \) with minimum number of edges.

Performance: \( O(|V|^3) \).

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

Network flow remains an active research area.

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 \(p_1, p_2, \ldots, p_n\), and the line segments \(p_1 – p_2, p_2 – p_3, \ldots, p_{n-1} – p_n\) 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.

Problem with Ford-Fulkerson:

Draw graph with nodes nodes s, t, a, and b. Flow from S to a and b is M, flow from a and b to t is M, flow from a to b is 1.

Now, pick s-a-b-t.
Then s-b-a-t. (reverse 1 unit of flow).
Repeat M times.
M is unrelated to the size of V, E, so this is potentially exponential.

Geometric Algorithms

Same principles often apply to 3D, but it may be more complicated.
We will avoid continuous problems such as polygon intersection.

Special cases: Geometric programming is much like other programming in this sense. But there are a LOT of special cases! Co-point, co-linear, co-planar, horizontal, vertical, etc.

Numeric stability: Each intersection point in a cascade of intersections might require increasing precision to represent the computed intersection, even when the point coordinates start as integers. Floating point causes problems!

Definitions

Line alternate representation: slope and intercept.
For polygons, order matters. A left-handed and right-handed triangle are not the same even if they occupy the same space.
Definitions (cont)

- **Simple Polygon**: The corresponding path does not intersect itself.
  - A simple polygon encloses a region of the plane INSIDE the polygon.
- **Basic operations**, assumed to be computed in constant time:
  - Determine intersection point of two line segments.
  - Determine which side of a line that a point lies on.
  - Determine the distance between two points.

**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 (1)**

**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.

**Point in Polygon Analysis (2)**

Two general principles for geometrical and graphical algorithms:
- 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.
- 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 \( q \).

**Special cases**:
- Line intersects polygon at a vertex, goes in to out.
- Line intersects poly. at inflection point (stays in or stays out).
- Line intersects polygon through a line.

Simplify calculations by making line horizontal.

Accuracy of calculations is not a problem with integer coordinates for points and a horizontal line. But think about representing the intersection point for two arbitrary line segments (from a polygon intersection operation). Cascading intersections can lead to ever-increasing demand for precision in coordinate representation.

“Fast checks” take time. When they “win” (they rule something out), they save time. When they “lose” (they fail to rule something out) they add extra time. So they have to “win” often enough so that the time savings outweighs the cost of the check.
Constructing Simple Polygons

**Problem:** Given a set of points, connect them with a simple closed path.

**Approaches:**
- Randomly select points.
- Use a scan line:
  - Sort points by y value.
  - Connect in sorted order.
- 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 \( p_1, \ldots, p_n \) by the order in which they are encountered by the scan line.
- For all \( i, 1 \leq i < n \), edge \( p_i \rightarrow p_{i+1} \) is in a distinct slice of the circle formed by a rotation of the scan line.
- Thus, edge \( p_i \rightarrow p_{i+1} \) does not intersect any other edge.
- Exception: If the angle between points \( p_i \) and \( p_{i+1} \) is greater than 180°.

**Implementation**

How do we find the point for the scan line center?

Actually, we don’t care about angle – slope will do.

Select \( z \);

for \( i = 2 \) to \( n \)  
compute the slope of line \( z \rightarrow p_i \).
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 \( p_1, p_2, \ldots, p_n \) such that connecting \( p_1 \) and \( p_n \) results in a convex polygon.
- The convex hull for a set of points is the smallest convex polygon enclosing all the points.
  - imagine placing a tight rubberband around the points.
- The point **belongs** to the hull if it is a vertex of the hull.
- **Problem:** Compute the convex hull of \( n \) points.
Simple Convex Hull Algorithm

IH: Assume that we can compute the convex hull for < \( n \) points, and try to add the \( n \)th point.

- \( n \)th point is inside the hull.
  - No change.
- \( n \)th point is outside the convex hull
  - “Stretch” hull to include the point (dropping other points).

Subproblems (1)

Potential problems as we process points:
- Determine if point is inside convex hull.
- Stretch a hull.

The straightforward induction approach is inefficient. (Why?)

Our standard induction alternative: Select a special point for the \( n \)th point – some sort of min or max point.

If we always pick the point with max \( x \), what problem is eliminated?
Stretch:
  - Find vertices to eliminate
  - Add new vertex between existing vertices.

Subproblems (2)

Supporting line of a convex polygon is a line intersecting the polygon at exactly one vertex.

Only two supporting lines between convex hull and max point \( q \).

These supporting lines intersect at “min” and “max” points on the (current) convex hull.

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 $q$th 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 = []; Point p = the point in S with largest x coordinate;
    P = P ∪ 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 ∪ q;
        L = -p - q-;
        p = q;
    }
}
```

**Gift Wrapping Analysis**

Complexity:
- To add $k$th point, find the min angle among $n - k$ lines.
- Do this $h$ times (for $h$ the number of points on hull).
- Often good in average case.
- Could be bad in worst case.

$O(n^2)$. Actually, $O(hn)$ where $h$ is the number of edges to hull.
Graham’s Scan

- Approach:
  - Start with the points ordered with respect to some maximal point.
  - Process these points in order, adding them to the set of processed points and its convex hull.
  - Like straightforward algorithm, but pick better order.
- Use the Simple Polygon algorithm to order the points by angle with respect to the point with max $x$ value.
- Process points in this order, maintaining the convex hull of points seen so far.

Graham’s Scan (cont)

Induction Hypothesis:
- Given a set of $n$ points ordered according to algorithm Simple Polygon, we can find a convex path among the first $n - 1$ points corresponding to the convex hull of the $n - 1$ points.

Induction Step:
- Add the $4$th 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 GrahamScan(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($q_{m-1}, q_m, q_{m+1}$) $\leq 180^\circ$) do
            $m = m - 1$;
            $m = m + 1$;
            $q_m = p_k$;
        }
    }
}

Graham’s Scan Analysis

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 parabola $y = x^2$.
- 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 $O(n \log n)$.

“Black Box” Model

A Sorting Algorithm:
- keys → points: $O(n)$
- Convex Hull
- CH Polygon → Sorted Keys: $O(n)$

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 $n$th 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.

Assume $n = 2^k$ points.

Note: We will actually compute smallest distance, not pair of points with smallest distance.
Closest Pair Algorithm

Induction Hypothesis:

- We can solve closest pair for two sets of size \( n/2 \) named \( P_1 \) and \( P_2 \).

Let minimal distance in \( P_1 \) be \( d_1 \), and for \( P_2 \) be \( d_2 \).

- Assume \( d_1 \leq d_2 \).

Only points in the strip of width \( d_1 \) to either side of the dividing line need to be considered.

Worst case: All points are in the strip.

Closest Pair Algorithm (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.

Sweep Line Algorithms (1)

Problem: Given a set of \( n \) horizontal and \( m \) vertical line segments, find all intersections between them.
- Assume no intersections between 2 vertical or 2 horizontal lines.

Straightforward algorithm: Make all \( n \times m \) comparisons.

If there are \( n \times m \) intersections, this cannot be avoided.

However, we would like to do better when there are fewer intersections.

Solution: Special order of induction will be imposed by a sweep line.

Sweep Line Algorithms (2)

Plane sweep or sweep line algorithms pass an imaginary line through the set of objects.

As objects are encountered, they are stored in a data structure.

When the sweep passes, they are removed.

Preprocessing Step:
- Sort all line segments by \( x \) coordinate.

Sweep Line Algorithms (3)

Inductive approach:
- We have already processed the first \( k - 1 \) end points when we encounter endpoint \( k \).
- Furthermore, we store necessary information about the previous line segments to efficiently calculate intersections with the line for point \( k \).

Possible approaches:
- Store vertical lines, calculate intersection for horizontal lines.
- Store horizontal lines, calculate intersection for vertical lines.

Since we processed by \( x \) coordinate (i.e., sweeping horizontally) do (2). When we process a vertical line, it is clear which horizontal lines would be relevant (the ones that cross that include the \( x \) coordinate of the vertical line), and so could hope to find them in a data structure. If we stored vertical lines, when we process the next horizontal line, it is not so obvious how to find all vertical lines in the horizontal range.
Organizing Sweep Info

What do we need when encountering line \( L \)?
- NOT horizontal lines whose right endpoint is to the left of \( L \).
- Maintain \textit{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:
- \((k + 1)\)th endpoint is right endpoint of horizontal line.
  - Delete horizontal line.
- \((k + 1)\)th endpoint is left endpoint of horizontal line.
  - Insert horizontal line.
- \((k + 1)\)th endpoint is vertical line.
  - Find intersections with stored horizontal lines.

Data Structure Requirements (1)

To have an efficient algorithm, we need efficient
- Intersection
- Deletion
- 1 dimensional range query

Example solution: Balanced search tree
- Insert, delete, locate in \( \log n \) time.
- Each additional intersection calculation is of constant cost beyond first (traversal of tree).

Data Structure Requirements (2)

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.

See Figure 8.17 in Manber.

y coordinates of the active horizontal lines.
Reductions

A reduction is a transformation of one problem to another.

Purpose: To compare the relative difficulty of two problems.

Example:
Sorting reals reduces to (in linear time) the problem of finding a convex hull in two dimensions.
- Use CH as a way to solve sorting.

We argued that there is a lower bound of $\Omega(n \log n)$ on finding the convex hull since there is a lower bound of $\Omega(n \log n)$ on sorting.

Reduction Notation

- We denote names of problems with all capital letters.
  - Ex: SORTING, CONVEX HULL
- What is a problem?
  - A relation consisting of ordered pairs (I, SLN).
  - I comes from the set of instances (allowed inputs).
  - SLN is the solution to the problem for instance I.
- Example: SORTING = (I, SLN).
  - I is a finite subset of R.
  - Prototypical instance: \{x_1, x_2, ..., x_n\}.
- SLN is the sequence of reals from I in sorted order.

Black Box Reduction (1)

The job of an algorithm is to take an instance I and return a solution SLN, or to report that there is no solution.

A reduction from problem A(I, SLN) to problem B(I', SLN') requires two transformations (functions) T, T'.

\[ T: I \Rightarrow I' \]
- Maps instances of the first problem to instances of the second.

\[ T': SLN \Rightarrow SLN' \]
- Maps solutions of the second problem to solutions of the first.

Black Box Reduction (2)

Black box idea:
- Start with an instance I of problem A.
- Transform to an instance I' = T(I), an instance of problem B.
- Use a “black box” algorithm for B as a subroutine to find a solution SLN' for B.
- Transform to a solution SLN = T(SLN'), a solution to the original instance I for problem A.

This example we have already seen.

NOT reduce CH to sorting – that just means that we can make CH as hard as sorting! Using sorting isn’t necessarily the only way to solve the CH problem, perhaps there is a better way. So just knowing that sorting is ONE WAY to solve CH doesn’t tell us anything about the cost of CH. On the other hand, by showing that we can use CH as a tool to solve sorting, we know that CH cannot be faster than sorting.
More Notation

If \((I, \text{SLN})\) reduces to \((I', \text{SLN}')\), write:

\((I, \text{SLN}) \leq (I', \text{SLN}')\).

This notation suggests that \((I, \text{SLN})\) is no harder than \((I', \text{SLN}')\).

Examples:

- \text{SORTING} \leq \text{CONVEX HULL}

The time complexity of \(T\) and \(T'\) is important to the time complexity of the black box algorithm for \((I, \text{SLN})\).

If combined time complexity is \(O(g(n))\), write:

\((I, \text{SLN}) \leq O(g(n)) (I', \text{SLN}')\).

Reduction Example

\text{SORTING} = (I, \text{SLN})

\text{CONVEX HULL} = (I', \text{SLN}') .

\begin{enumerate}
\item \(I = \{x_1, x_2, \ldots, x_n\}\),
\item \(T(I) = I' = \{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}\),
\item Solve \text{CONVEX HULL} for \(I'\) to give solution \(\text{SLN}' = \{(x_{i[1]}, x_{i[1]}^2), (x_{i[2]}, x_{i[2]}^2), \ldots, (x_{i[k]}, x_{i[k]}^2)\}\).
\item \(T\) finds a solution to \(I\) from \(\text{SLN}\) as follows:
\begin{itemize}
\item Find \((x_{j[k]}, x_{j[k]}^2)\) such that \(x_{j[k]}\) is minimum.
\item \(Y = x_{j[k]}, x_{j[k+1]}, \ldots, x_{j[k]}, x_{j[k+1]}, \ldots, x_{j[k]}\).
\item For a reduction to be useful, \(T\) and \(T'\) must be functions that can be computed by algorithms.
\end{itemize}
\item An algorithm for the second problem gives an algorithm for the first problem by steps 2 – 4.
\end{enumerate}

Notation Warning

Example: \text{SORTING} \leq O(n) \text{CONVEX HULL}.

WARNING: \leq is NOT a partial order because it is NOT antisymmetric.

\text{SORTING} \leq O(n) \text{CONVEX HULL}.

\text{CONVEX HULL} \leq O(n) \text{SORTING}.

But, \text{SORTING} \not\leq \text{CONVEX HULL}.

Bounds Theorems

\textbf{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:

- \text{SORTING} \leq O(n) \text{CONVEX HULL}.
- \(g(n) = n, h(n) = n \log n, g(n) = o(h(n))\).
- Theorem gives \(\Omega(n \log n)\) lower bound on \text{CONVEX HULL}.

\textbf{Upper Bound Theorem:} If \(P_3\) has time complexity \(O(h(n))\) and \(P_1 \leq O(g(n)) P_2\), then \(P_1\) has time complexity \(O(g(n) + h(n))\).

Notice \(o\), not \(O\). So, given good transformations, both problems take at least \(\Omega(P_1)\) and at most \(O(P_2)\).
System of Distinct Representatives (SDR)

**Instance:** Sets $S_1, S_2, \ldots, S_n$.
**Solution:** Set $R = \{r_1, r_2, \ldots, r_n\}$ 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.
- $\text{SDR} \leq \mathcal{O}(n)$ \textsc{Bipartite Matching}.
- Given an instance of $S_1, S_2, \ldots, S_n$ of SDR, transform it to an instance $G = (U, V, E)$ of \textsc{Bipartite Matching}.
- Let $S = \bigcup_{i=1}^{n} S_i$, $U = \{S_1, S_2, \ldots, S_n\}$,
- $V = S$, $E = \{(S, x_i) | x_i \in S_i\}$.

A solution to SDR is easily obtained from a maximum matching in $G$ of size $k$.

**Simple Polygon Lower Bound (1)**

- SIMPLE POLYGON: Given a set of $n$ points in the plane, find a simple polygon with those points as vertices.
- SORTING $\leq \mathcal{O}(n)$ SIMPLE POLYGON.
- Instance of SORTING: $\{x_1, x_2, \ldots, x_n\}$.
  - In linear time, find $M = \max |x_i|$.
  - Let $C$ be a circle centered at the origin, of radius $M$.
- Instance of SIMPLE POLYGON:
  - $\{(x_1, \sqrt{M^2 - x_1^2}), \ldots, (x_n, \sqrt{M^2 - x_n^2})\}$.

All these points fall on $C$ in their sorted order.
- The only simple polygon having the points on $C$ as vertices is the convex one.

**Simple Polygon Lower Bound (2)**

- As with CONVEX HULL, the sorted order is easily obtained from the solution to SIMPLE POLYGON.
- By the Lower Bound Theorem, SIMPLE POLYGON is $\Omega(n \log n)$.
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.

$$\text{MATRIX MULTIPLY} \leq O(n^2) \text{ SYM.}$$

Clearly SYM is not harder than MM. Is it easier? No...

So, having a good SYM would give a good MM. The other way of looking at it is that SYM is just as hard as MM.

Matrix Squaring

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

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

Linear Programming (LP)

Maximize or minimize a linear function subject to linear constraints.
Variables: vector $X = (x_1, x_2, \ldots, x_n)$.

Objective Function: $c \cdot X = \sum c_i x_i$.
Inequality Constraints: $A_i \cdot X \leq b_i$, $1 \leq i \leq k$.
Equality Constraints: $E_i \cdot X = d_i$, $1 \leq i \leq m$.

Non-negative Constraints: $x_i \geq 0$ for some $i$.

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.
Network Flow Reduction (1)

- Reduce NETWORK FLOW to LP.
- Let $x_1, x_2, \ldots, x_n$ be the flows through edges.
- Objective function: For $S = \text{edges out of the source}$, maximize
  \[ \sum_{i \in S} x_i. \]
- Capacity constraints: $x_i \leq c_i$ for $1 \leq i \leq n$.
- Flow conservation:
  - For a vertex $v \in V - \{s, t\}$, let $Y(v) = \text{set of } x_i \text{ for edges leaving } v$.
  - $Z(v) = \text{set of } x_i \text{ for edges entering } v$.
  \[ \sum_{v \in S} x_i - \sum_{v \in Y(v)} x_i = 0. \]
- Integer constraints makes this INTEGER LINEAR PROGRAMMING (ILP).

Network Flow Reduction (2)

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

Maximize: $x_1 + x_4$ subject to:

\[
\begin{align*}
x_1 & \leq 4 \\
x_2 & \leq 3 \\
x_3 & \leq 2 \\
x_4 & \leq 5 \\
x_5 & \leq 7 \\
x_1 + x_2 - x_3 &= 0 \\
x_4 - x_3 - x_5 &= 0 \\
x_1, \ldots, x_5 &\geq 0
\end{align*}
\]

Matching

- Start with graph $G = (V, E)$.
- Let $x_1, x_2, \cdots, x_n$ represent the edges in $E$.
  - $x_i = 1$ means edge $i$ is matched.
- Objective function: Maximize
  \[ \sum_{i=1}^{n} x_i. \]
- Subject to: (Let $N(v)$ denote edges incident on $v$)
  \[ \sum_{N(v)} x_i \leq 1 \]
  \[ x_i \geq 0 \quad 1 \leq i \leq n \]

- Integer constraints: Each $x_i$ must be an integer.
- Integer constraints makes this INTEGER LINEAR PROGRAMMING (ILP).

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 **tractable** if an algorithm exists to solve it with polynomial time complexity: $O(p(n))$.

• It is said to be **intractable** if the best known algorithm requires exponential time.

Examples:
• Sorting: $O(n^2)$
• Convex Hull: $O(n^2)$
• Single source shortest path: $O(n^2)$
• All pairs shortest path: $O(n^3)$
• Matrix multiplication: $O(n^3)$

Log-polynomial is $O(n \log n)$

Like any simple rule of thumb for categorizing, in some cases the distinction between polynomial and exponential could break down. For example, one can argue that, for practical problems, $1.01^n$ is preferable to $n^2$. But the reality is that very few polynomial-time algorithms have high degree, and exponential-time algorithms nearly always have a constant of 2 or greater. Nearly all algorithms are either low-degree polynomials or “real” exponentials, with very little in between.
Tractable Problems (cont)

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

- A special kind of reduction.
- Nondeterminism.

Decision Problems

\((I, S)\) such that \(S(X)\) is always either “yes” or “no.”

- Usually formulated as a question.

**Example:**

- Instance: A weighted graph \(G = (V, E)\), two vertices \(s\) and \(t\), and an integer \(K\).

  - Question: Is there a path from \(s\) to \(t\) of length \(\leq K\)? In this example, the answer is “yes.”

Decision Problems (cont)

Can also be formulated as a language recognition problem:

- Let \(L\) be the subset of \(I\) consisting of instances whose answer is “yes.” Can we recognize \(L\)?

The class of tractable problems \(\mathcal{P}\) is the class of languages or decision problems recognizable in polynomial time.

Polynomial Reducibility

Reduction of one language to another language.

Let \(L_1 \subseteq I_1\) and \(L_2 \subseteq I_2\) be languages. \(L_1\) is **polyominally reducible** to \(L_2\) if there exists a transformation \(T : I_1 \rightarrow I_2\), computable in polynomial time, such that

\[
f(x) \in L_2 \text{ if and only if } x \in L_1.
\]

We write: \(L_1 \leq_p L_2\) or \(L_1 \leq L_2\).

Polynomial Reducibility (cont)

Or one decision problem to another.

Specialized case of reduction from Chapter 10.
Examples

- CLIQUE $\leq_p$ INDEPENDENT SET.
- An instance $I$ of CLIQUE is a graph $G = (V, E)$ and an integer $K$.
- The instance $I' = f(I)$ of INDEPENDENT SET is the graph $G' = (V, E')$ and the integer $K$, were an edge $(u, v) \in E'$ iff $(u, v) \not\in E$.
- $f$ 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;
}
```

What makes this different than random guessing is that all choices happen "in parallel."
Nondeterministic Acceptance

- \((G, K)\) is in the “language” CLIQUE iff there exists a sequence of nd-choice guesses that causes nd-CLIQUE to accept.
- Definition of acceptance by a nondeterministic algorithm:
  - An instance is accepted iff there exists a sequence of nondeterministic choices that causes the algorithm to accept.
- An unrealistic model of computation.
  - There are an exponential number of possible choices, but only one must accept for the instance to be accepted.
- Nondeterminism is a useful concept
  - It provides insight into the nature of certain hard problems.

Class \(NP\)

- The class of languages accepted by a nondeterministic algorithm in polynomial time is called \(NP\).
- There are an exponential number of different executions of nd-CLIQUE on a single instance, but any one execution requires only polynomial time in the size of that instance.
- Time complexity of nondeterministic algorithm is greatest amount of time required by any one of its executions.

Alternative Interpretation:

- \(NP\) 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, \(P \subset NP\).

Extra Credit Problem:

- Prove or disprove: \(P = NP\).

This is important because there are many natural decision problems in \(NP\) for which no \(P\) (tractable) algorithm is known.
\(N^P\)-completeness

A theory based on identifying problems that are as hard as any problems in \(N^P\).

The next best thing to knowing whether \(P = N^P\) or not.

A decision problem \(A\) is \(N^P\)-hard if every problem in \(N^P\) is polynomially reducible to \(A\), that is, for all
\[
B \in N^P, \quad B \leq_p A.
\]

A decision problem \(A\) is \(N^P\)-complete if \(A \in N^P\) and \(A\) is \(N^P\)-hard.

**Proof:**

1. Since \(SA\) \(\in N^P\), \(\leq P\) \(SA\).
2. If \(SA\) \(\leq P\) \(A\), then \(A\) \(\in N^P\).
3. If \(A \in N^P\) and \(B \in N^P\), then \(B \leq_p A\).

**Satisfiability**

Let \(E\) be a Boolean expression over variables \(x_1, x_2, \ldots, x_n\) in conjunctive normal form (CNF), that is, an AND of ORs.

\[
E = (x_0 + x_1 + x_5) \cdot (x_2 + x_3) \cdot (x_1 + x_3 + x_6).
\]

A variable or its negation is called a literal. Each sum is called a clause.

SATISFIABILITY (SAT):
- Instance: A Boolean expression \(E\) over variables \(x_1, x_2, \ldots, x_n\) in CNF.
- Question: Is \(E\) satisfiable?

**Cook’s Theorem:** \(SA\) \(\in N^P\)-complete.

**Proof Sketch**

\(SAT \leq P N^P\):
- A non-deterministic algorithm guesses a truth assignment for \(x_1, x_2, \ldots, x_n\) and checks whether \(E\) is true in polynomial time.
- It accepts if there is a satisfying assignment for \(E\).

\(SAT\) \(\leq P N^P\)-hard:
- Start with an arbitrary problem \(B \in N^P\).
- 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 \(SA\) \(\in N^P\)-complete, we have not defined an empty concept.
2. If \(SA \leq P\), then \(P = N^P\).
3. If \(P = N^P\), then \(SA \in P\).
4. If \(A \in N^P\) and \(B \in N^P\)-complete, then \(B \leq P A\) implies \(A\) \(\in N^P\)-complete.

Proof:
- Let \(C \in N^P\).
- Then \(C \leq P B\) since \(B\) \(\in N^P\)-complete.
- Since \(B \leq P A\) and \(\leq P\) is transitive, \(C \leq P A\).
- Therefore, \(A\) \(\in N^P\)-hard and, finally, \(N^P\)-complete.

A is not permitted to be harder than \(N^P\). For example, Tower of Hanoi is not in \(N^P\). It requires exponential time to verify a set of moves.
Implications (cont)

(5) This gives a simple two-part strategy for showing a decision problem $A$ is $\mathcal{NP}$-complete.
(a) Show $A \in \mathcal{NP}$.
(b) Pick an $\mathcal{NP}$-complete problem $B$ and show $B \leq_p A$.

$\mathcal{NP}$-completeness Proof Paradigm

To show that decision problem $B$ is $\mathcal{NP}$-complete:
- $B \in \mathcal{NP}$
  - Give a polynomial time, non-deterministic algorithm that accepts $B$.
  - Given an instance $X$ of $B$, guess evidence $Y$.
  - Check whether $Y$ is evidence that $X \in B$. If so, accept $X$.
- $B$ is $\mathcal{NP}$-hard.
  - Choose a known $\mathcal{NP}$-complete problem, $A$.
  - Describe a polynomial-time transformation $T$ of an arbitrary instance of $A$ to a [not necessarily arbitrary] instance of $B$.
  - Show that $X \in A$ if and only if $T(X) \in B$.

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.
Choosing SAT to be the known $\mathcal{NP}$-complete problem.

1. 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 $(y_i)$ with $(y_1 + v + w) \cdot (y_1 + v + w) \cdot (y_1 + v + w)$
     where $v$ and $w$ are new variables.

(b) $j = 2$
   - Replace $(y_i + y_2)$ with $(y_1 + y_2 + z) \cdot (y_1 + y_2 + z)$ where $z$ is a new variable.

(c) $j > 3$
   - Replace $(y_i + y_2 + \cdots + y_j)$ with
     
     $(y_1 + y_2 + z_1) \cdot (y_3 + z_2 + z_3) \cdots (y_{j-2} + z_{j-4} + z_{j-3}) \cdot (y_{j-1} + y_j + z_{j-3})$
     
     where $z_1, z_2, \cdots, z_{j-3}$ are new variables.
   - After replacements made for each $C_i$, a Boolean expression $E'$ results that is an instance of 3SAT.
     - The replacement clearly can be done by a polynomial-time deterministic algorithm.

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

- 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.

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

- **Tree of Reductions**
  - **Reductions go down the tree.**
  - **Proofs that each problem ∈ NP are straightforward.**

### Perspective

The reduction tree gives us a collection of 12 diverse NP-complete problems. The complexity of all these problems depends on the complexity of any one:

- If any 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 NP-complete.

**Observation:** If we find a problem is NP-complete, then we should do something other than try to find a P-time algorithm.

### SAT ≤p CLIQUE

1. Easy to show CLIQUE in NP.
2. An instance of SAT is a Boolean expression
   
   \[ B = C_1 \cdot C_2 \cdot \ldots \cdot C_m, \]
   
   where
   
   \[ C_i = y[i, 1] + y[i, 2] + \ldots + y[i, k]. \]

   Transform this to an instance of CLIQUE \( G = (V, E) \) and \( K \).
   
   \[ V = \{ v[i, j] \mid 1 \leq i \leq m, 1 \leq j \leq k \} \]

   Two vertices \( v[i, j] \) and \( v[i, k] \) are adjacent in \( G \) if \( i \neq k \) and EITHER \( y[i, j] \) and \( y[i, k] \) are the same literal OR \( y[i, j] \) and \( y[i, k] \) have different underlying variables. \( K = m. \)

### SAT ≤p CLIQUE (cont)

**Example:** \( B = (x_1 + x_2) \cdot (x_1 + x_2 + x_3). \)

\( K = 2. \)

(3) \( B \) is satisfiable iff \( G \) has clique of size \( \geq K \).

- \( B \) is satisfiable implies there is a truth assignment such that \( y[i, j] \) is true for each \( i. \)
- But then \( v[i, j] \) 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 \( v[i, j] \) in the clique for each \( i. \)
- There is a truth assignment making each \( y[i, j] \) true.

That truth assignment satisfies \( B. \)

We conclude that CLIQUE is \( \mathcal{NP} \)-hard, therefore \( \mathcal{NP} \)-complete.

**Refer to handout of \( \mathcal{NP} \)-complete problems**

**Hundreds of problems, from many fields, have been shown to be \( \mathcal{NP} \)-complete.**

**More on this observation later.**

**One vertex for each literal in \( B. \)**

**No join if one is the negation of the other**

**Need figure here. Another example is shown in Manber Figure 11.3.**

**It must connect to the other \( m - 1 \) literals that are also true.**

**No clique can have more than one member from the same clause, since there are no links between members of a clause.**
PARTITION ≤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 our \(O(KN)\) dynamic prog algorithm?
- Input size for KNAPSACK is \(O(N \log K)\)
  - Thus \(O(KN)\) is exponential in \(N \log K\).
- The dynamic programming algorithm counts through numbers \(1, \ldots, K\). Takes exponential time when measured by number of bits to represent \(K\).
- If \(K\) is “small” \((K = O(p(N)))\), then algorithm has complexity polynomial in \(N\) and is truly polynomial in input size.
- An algorithm that is polynomial-time if the numbers IN the input are “small” (as opposed to number OF inputs) is called a pseudo-polynomial time algorithm.

Co-\(\mathcal{NP}\)

- Note the asymmetry in the definition of \(\mathcal{NP}\).
  - The non-determinism can identify a clique, and you can verify it.
  - But what if the correct answer is “NO”? How do you verify that?
- Co-\(\mathcal{NP}\): The complements of problems in \(\mathcal{NP}\).
  - Is a boolean expression always false?
  - Is there no clique of size \(k\)?
- It seems unlikely that \(\mathcal{NP} = \text{co-}\mathcal{NP}\).

Is \(\mathcal{NP}\)-complete \(\equiv \mathcal{NP}\)?

- It has been proved that if \(P \neq \mathcal{NP}\), then \(\mathcal{NP}\)-complete \(\neq \mathcal{NP}\).
- The following problems are not known to be in \(P\) or \(\mathcal{NP}\), but seem to be of a type that makes them unlikely to be in \(\mathcal{NP}\).
  - GRAPH ISOMORPHISM: Are two graphs isomorphic?
  - COMPOSITE NUMBERS: For positive integer \(K\), are there integers \(m, n > 1\) such that \(K = mn\)?
  - LINEAR PROGRAMMING

These problems seem easier than typical \(\mathcal{NP}\)-complete problems, but are still probably harder than \(P\). They are obviously in \(\mathcal{NP}\), but don’t appear to be “hard” enough to solve any \(\mathcal{NP}\)-complete problem.

This is an important point, about the input size. It has to do with the “size” of a number (a value). We represent the value \(n\) with \(\log n\) bits, or more precisely, \(\log n\) bits where \(N\) is the maximum value. In the case of KNAPSACK, \(K\) (the knapsack size) is effectively the maximum number. We will use this observation frequently when we analyze numeric algorithms.
“Practical” Problems (cont)

- Lesson: While KNAPSACK is \( \mathcal{NP} \)-complete, it is often not that hard.
- Many \( \mathcal{NP} \)-complete problems have no pseudo-polynomial time algorithm unless \( P = \mathcal{NP} \).

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:
- Each edge represents a particular nondeterministic choice.
- 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:

- A leaf that accepts is found.
- A partial solution that could not possibly lead to acceptance is reached.

Example:

There cannot possibly be a set \( S \) of cardinality \( \geq 2 \) 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

\[ |V'| \leq 2OPT \]

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

Need Figure here.

When the corresponding decision problem is \( \mathcal{NP} \)-complete.
Bin Packing

We have numbers $x_1, x_2, \ldots, x_n$ 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 $\frac{3}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{8}, \frac{2}{3}, \frac{1}{2}, \frac{1}{4}$.

Optimal solution: $[\frac{3}{4}, \frac{1}{8}], [\frac{1}{2}, \frac{1}{3}], [\frac{1}{2}, \frac{1}{4}], [\frac{2}{3}]$.

First Fit Algorithm

Place $x_i$ 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 $\epsilon$ be very small, e.g., $\epsilon = 0.0001$.

Numbers (in this order):
- 6 of $(\frac{1}{7} + \epsilon)$.
- 6 of $(\frac{1}{3} + \epsilon)$.
- 6 of $(\frac{1}{2} + \epsilon)$.

First fit returns:
- 1 bin of $[6 \text{ of } \frac{1}{7} + \epsilon]$
- 3 bins of $[2 \text{ of } \frac{1}{3} + \epsilon]$
- 6 bins of $[\frac{1}{2} + \epsilon]$

Optimal solution is 6 bins of $[\frac{1}{7} + \epsilon, \frac{1}{3} + \epsilon, \frac{1}{2} + \epsilon]$.

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:

  \[
  \text{Is it } \mathcal{NP}\text{-complete?} \iff \text{Is it in } \mathcal{P}?
  \]

- Alternately think about each question. Lack of progress on either question might give insights into the answer to the other question.
- Once an affirmative answer is obtained to one of these questions, one of two strategies is followed.

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 \textit{bits}.
  - Algebraic operations:
    - Measure size of input in terms of \textit{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:

\[
p = 1; \\
\text{for } (i=1 \text{ to } k) \\
p = p * n;
\]

Analysis:

- Input size: \( \Theta(\log n + \log k) \).
- Time complexity: \( \Theta(k) \) multiplications.
- This is \textit{exponential} in input size.
Faster Exponentiation

Write $k$ as:
$$k = b_12^1 + b_{-1}2^{-1} + \cdots + b_2 + b_0, \ b \in \{0, 1\}.$$ 
Rewrite as:
$$k = ((\cdots (b_2 + b_{-1})2 + \cdots + b_2)2 + b_1)2 + b_0.$$ 
New algorithm:
$$p = n;$$
for ($i = t-1$ downto 0)
$$p = p \cdot p \cdot \exp(n, b[i])$$
Analysis:
- Time complexity: $\Theta(t) = \Theta(\log k)$ multiplications.
- This is exponentially better than before.

Greatest Common Divisor

- The Greatest Common Divisor (GCD) of two integers is the greatest integer that divides both evenly.
- Observation: If $k$ divides $n$ and $m$, then $k$ divides $n - m$.
- So,
$$f(n, m) = f(n - m, n) = f(m, n - m) = f(m, n).$$
- Observation: There exists $k$ and $l$ such that
$$n = km + l \text{ where } m > l \geq 0.$$ 
$$n = \lfloor n/m \rfloor m + n \mod m.$$ 
- So,
$$f(n, m) = f(m, l) = f(m, n \mod m).$$

GCD Algorithm

$$f(n, m) = \begin{cases} 
  n & m = 0 \\
  f(m, n \mod m) & m > 0 
\end{cases}$$

```c
int LCF(int n, int m) {
  if (m == 0) return n;
  return LCF(m, n % m);
}
```

Analysis of GCD

- How big is $n \mod m$ relative to $n$?
$$n \geq m \Rightarrow \frac{n}{m} \geq 1$$
$$2\lfloor n/m \rfloor > n/m$$
$$m\lfloor n/m \rfloor > n/2$$
$$n - n/2 > n - m\lfloor n/m \rfloor = n \mod m$$
$$n/2 > n \mod m$$
- The first argument must be halved in no more than 2 iterations.
- Total cost:
$$\text{Total cost:}$$
$$\text{Analysis of GCD}$$

Can split in half log $n$ times. So $2 \log n$ is upper bound.

Note that this is linear on problem size, since problem size is $2 \log n$ (2 numbers).
Given:

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

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

**Multiplying Polynomials (2)**

- **Divide and Conquer:**

\[ P_1 = \sum_{i=0}^{n/2-1} p_i x^i \]
\[ P_2 = \sum_{i=n/2}^{n-1} p_i x^{i-n/2} \]
\[ Q_1 = \sum_{i=0}^{n/2-1} q_i x^i \]
\[ Q_2 = \sum_{i=n/2}^{n-1} q_i x^{i-n/2} \]

\[ PQ = (P_1 + x^{n/2} P_2)(Q_1 + x^{n/2} Q_2) = P_1 Q_1 + x^{n/2}(Q_1 P_2 + P_1 Q_2) + x^n P_2 Q_2. \]

- **Recurrence:**

\[ T(n) = 4T(n/2) + O(n). \]
\[ T(n) = \Theta(n^2). \]

**Multiplying Polynomials (3)**

Observation:

\[ (P_1 + P_2)(Q_1 + Q_2) = P_1 Q_1 + (Q_1 P_2 + P_1 Q_2) + P_2 Q_2. \]
\[ (Q_1 P_2 + P_1 Q_2) = (P_1 + P_2)(Q_1 + Q_2) - P_1 Q_1 - P_2 Q_2. \]

Therefore, \( PQ \) can be calculated with only 3 recursive calls to a polynomial multiplication procedure.

**Recurrence:**

\[ T(n) = 3T(n/2) + O(n) \]
\[ = aT(n/b) + cn^\log_b a \approx 1.59. \]
\[ T(n) = \Theta(n^{\log_2 3}). \]

**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: \( \Omega(n^2) \).
Strassen's Algorithm

(1) Trade more additions/subtractions for fewer multiplications in 2 × 2 case.

(2) Divide and conquer.

In the straightforward implementation, 2 × 2 case is:

\[ c_{11} = a_{11}b_{11} + a_{12}b_{21} \]
\[ c_{12} = a_{11}b_{12} + a_{12}b_{22} \]
\[ c_{21} = a_{21}b_{11} + a_{22}b_{21} \]
\[ c_{22} = a_{21}b_{12} + a_{22}b_{22} \]

Requires 8 multiplications and 4 additions.

Another Approach (1)

Compute:

\[ m_1 = (a_{12} - a_{22})(b_{21} + b_{22}) \]
\[ m_2 = (a_{11} + a_{22})(b_{21} + b_{22}) \]
\[ m_3 = (a_{11} - a_{21})(b_{11} + b_{22}) \]
\[ m_4 = (a_{11} + a_{12})b_{22} \]
\[ m_5 = a_{11}(b_{12} - b_{22}) \]
\[ m_6 = a_{22}(b_{21} - b_{11}) \]
\[ m_7 = (a_{21} + a_{22})b_{11} \]

7 multiplications and 18 additions/subtractions.

Another Approach (2)

Then:

\[ c_{11} = m_1 + m_2 - m_4 + m_6 \]
\[ c_{12} = m_4 + m_5 \]
\[ c_{21} = m_5 + m_7 \]
\[ c_{22} = m_2 - m_3 + m_5 - m_7 \]

Strassen's Algorithm (cont)

Divide and conquer step:

Assume \( n \) is a power of 2.

Express \( C = A \times B \) in terms of \( \frac{n}{2} \times \frac{n}{2} \) matrices.

\[
\begin{bmatrix}
c_{11} & c_{12} \\
c_{21} & c_{22}
\end{bmatrix}
= \begin{bmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{bmatrix}
\begin{bmatrix}
b_{11} & b_{12} \\
b_{21} & b_{22}
\end{bmatrix}
\]
Strassen’s Algorithm (cont)

By Strassen’s algorithm, this can be computed with $7$ multiplications and $18$ additions/subtractions of $n/2 \times n/2$ matrices.

Recurrence:

$$T(n) = 7T(n/2) + 18(n/2)^2$$

$$T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.81}).$$

Current “fastest” algorithm is $\Theta(n^{2.376})$

Open question: Can matrix multiplication be done in $O(n^2)$ time?

Introduction to the Sliderule

Compared to addition, multiplication is hard.

In the physical world, addition is merely concatenating two lengths.

Observation:

$$\log nm = \log n + \log m.$$ 

Therefore,

$$nm = \text{antilog}(\log n + \log m).$$

What if taking logs and antilogs were easy?

Introduction to the Sliderule (2)

The sliderule does exactly this!

- It is essentially two rulers in log scale.
- Slide the scales to add the lengths of the two numbers (in log form).
- The third scale shows the value for the total length.

Representing Polynomials

A vector $\mathbf{a}$ of $n$ values can uniquely represent a polynomial of degree $n - 1$

$$P_a(x) = \sum_{i=0}^{n-1} a_i x^i.$$ 

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 \( B \): \( 2x^3 - x + 1 \).
Polynomial \( AB \): \( 2x^4 - x^3 + 3x^2 - x + 1 \).

Notice:
\[
AB(-1) = (2)(4) = 8 \\
AB(0) = (1)(1) = 1 \\
AB(1) = (2)(2) = 4 
\]

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 \( \omega \) is a primitive \( n \)th root of unity if
- \( \omega^n = 1 \) and
- \( \omega^k \neq 1 \) for \( 0 < k < n \).

\( \omega^0, \omega^1, \ldots, \omega^{n-1} \) are the \( n \)th roots of unity.

Example:
- For \( n = 4 \), \( \omega = i \) or \( \omega = -i \).
- For the first circle, \( n = 4 \), \( \omega = i \).
- For the second circle, \( n = 8 \), \( \omega = \sqrt[8]{1} \).
Nth Root of Unity (cont)

\[ n = 4, \; \omega = i. \]
\[ n = 8, \; \omega = \sqrt{i}. \]

Discrete Fourier Transform

Define an \( n \times n \) matrix \( V(\omega) \) with row \( i \) and column \( j \) as

\[ V(\omega) = (\omega^j)^i. \]

Example: \( n = 4, \; \omega = i: \)

\[
V(\omega) = \begin{bmatrix}
1 & 1 & 1 & 1 \\
i & -1 & -1 & i \\
i & 1 & 1 & -i \\
-1 & i & i & -1
\end{bmatrix}
\]

Let \( \mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T \) be a vector.
The Discrete Fourier Transform (DFT) of \( \mathbf{a} \) is:

\[ F_\omega = V(\omega)\mathbf{a} = \mathbf{v}. \]

This is equivalent to evaluating the polynomial at the \( n \)th roots of unity.

Array example

For \( n = 8, \; \omega = \sqrt{i}, \; V(\omega) = \)

\[
\begin{align*}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
i & \sqrt{i} & i & -\sqrt{i} & i & -i & -i & i \\
i & 1 & -1 & i & 1 & -1 & -i & i \\
i & -1 & 1 & -1 & 1 & 1 & -1 & i \\
i & -i & 1 & i & -1 & -i & 1 & -i \\
i & \sqrt{i} & -i & -\sqrt{i} & -i & i & i & \sqrt{i}
\end{align*}
\]

Inverse Fourier Transform

The inverse Fourier Transform to recover \( \mathbf{a} \) from \( \mathbf{v} \) is:

\[ F_\omega^{-1} \mathbf{a} = [V(\omega)]^{-1} \mathbf{v}. \]

\[ [V(\omega)]^{-1} = \frac{1}{n} V(1/\omega). \]

This is equivalent to interpolating the polynomial at the \( n \)th roots of unity.

An efficient divide and conquer algorithm can perform both the DFT and its inverse in \( \Theta(n \lg n) \) time.

Observe the sharable parts in the matrix.

In the array, indexing begins with 0.

Example:

\[ 1 + 2x + 3x^2 + 4x^3 \]

Values to evaluate at: \( 1, i, -1, -i. \)
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 $AB$.


FFT Algorithm

\[
\text{FFT}(n, a_0, a_1, ..., a_{n-1}, \omega, \text{var } V);
\]
Output: $V[0..n-1]$ of output elements.

\[
\begin{align*}
\text{begin} & \\
\text{if } n=1 & \text{ then } V[0] = a_0; \\
\text{else} & \\
\text{FFT}(n/2, a_0, a_2, ... a_{n-2}, \omega^2, U); & \\
\text{FFT}(n/2, a_1, a_3, ... a_{n-1}, \omega^2, W); & \\
\text{for } j=0 \text{ to } n/2-1 & \\
V[j+n/2] & = U[j] - \omega^j W[j]; \\
V[j] & = U[j] + \omega^j W[j]; \\
\text{end} & \\
\end{align*}
\]

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 $\infty$ 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

- Would need a new algorithm for every $p$!

Approach (2): Pick best algorithm for $p = \infty$, then convert to run on $p$ processors.

Hopefully, if $T(n, p) = X$, then $T(n, p/k) \approx kX$ for $k > 1$.

Using one processor to emulate $k$ processors is called the parallelism folding principle.
Parallel Algorithm Design (2)

Some algorithms are only good for a large number of processors.

\[
\begin{align*}
T(n, 1) &= n \\
T(n, n) &= \log n \\
S(n) &= n / \log n \\
E(n, n) &= 1 / \log n
\end{align*}
\]

For \( p = 256, n = 1024 \).

\( T(1024, 256) = 4 \log 1024 = 40. \)

For \( p = 16, \) running time \( = 1024/16 \times \log 1024 = 640. \) Speedup < 2, efficiency \( = 1024/(16 \times 640) = 1/10. \)

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:

\[
\text{Speedup} = \frac{(S + P)}{(S + P / N)}
\]

\[
= \frac{1}{(S + P / N)} \leq \frac{1}{S}.
\]

for \( S = \) serial fraction, \( P = \) parallel fraction, \( S + P = 1. \)

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, \( S = f(n) \) with \( S \) shrinking as \( n \) grows.

Instead of fixing problem size, fix execution time for increasing number \( N \) processors (and thus, increasing problem size).

Scaled Speedup = \( \frac{(S + P \times N)}{(S + P)} \)

\[
= \frac{S + P \times N}{S + P}
\]

\[
= \frac{S + (1 - S) \times N}{N} + \frac{(1 - N) \times S}{S}.
\]

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.

Good in terms of speedup 1024/256, assuming one processor emulates 4 in 4 times the time.

\( E(1024, 256) = 1024/(256 + 40) = 1/10. \)

But note that efficiency goes down as the problem size grows.


Speedup is Serial / Parallel.

Draw graph, speed up is Y axis, Sequential is X axis. You will see a nonlinear curve going down.

How long sequential process would take / How long for \( N \) processors.

Since \( S + P = 1 \) and \( P = 1 - S \).

The point is that this equation drops off much less slowly in \( N \): Graphing (sequential fraction for fixed \( N \)) vs. speedup, you get a line with slope \( 1 - N \).

All of this seems to assume the same algorithm for sequential and parallel. But that’s OK – we want to see how much parallelism is possible for the parallel algorithm.

Vector: IBM 3090, Cray

Pipelined: Graphics coprocessor boards

MIMD: Modern clusters.
MIMD Communications (1)

Interconnection network:
- Each processor is connected to a limited number of neighbors.
- Can be modeled as (undirected) graph.
- Examples: Array, mesh, N-cube.
- It is possible for the cost of communications to dominate the algorithm (and in fact to limit parallelism).
- **Diameter**: Maximum over all pairwise distances between processors.
- Tradeoff between diameter and number of connections.

MIMD Communications (2)

Shared memory:
- Random access to global memory such that any processor can access any variable with unit cost.
- In practice, this limits number of processors.
- Exclusive Read/Exclusive Write (EREW).
- Concurrent Read/Exclusive Write (CREW).
- Concurrent Read/Concurrent Write (CRCW).

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 \( i \)th 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.

Parallel Addition

Two possibilities: carry or not carry.

Also, for each a boolean indicating if it returns a carry.

If right has carry then
\[
\text{Sum} = L_c | R
\]
Else
\[
\text{Sum} = L | R
\]
If Sum has carry then
\[
\text{Carry} = \text{TRUE}
\]
For Sum_c

Do the same using \( R_c \) since it is computing value having received carry.
Parallel Addition (2)

The $n/2$-size problems are independent. Given enough processors,

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

We need only the EREW memory model.

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.

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

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

More Efficient EREW Algorithm (2)

But what could we do with more processors?
A parallel algorithm is static if the assignment of processors to actions is predefined.
- We know in advance, for each step $i$ of the algorithm and for each processor $p_i$, the operation and operands $p_i$ 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 $T(n,s/t) \in O(t)$.

Proof:
- Let $a_i, 1 \leq i \leq t$, be the total number of steps performed by all processors in step $i$ of the algorithm.
- \[ \sum_{i=1}^{t} a_i = s \]
- If $a_i \leq s/t$, then there are enough processors to perform this step without change.
- Otherwise, replace step $i$ with $\lceil a_i / (s/t) \rceil$ steps, where the $s/t$ processors emulate the steps taken by the original $p$ processors.

Thus, the running time is still $O(t)$.

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

Brent’s Lemma (2)

- The total number of steps is now
  \[ \sum_{i=1}^{t} \lceil a_i / (s/t) \rceil \leq \sum_{i=1}^{t} (a_i t / s + 1) \]

\[ = t + (t/s) \sum_{i=1}^{t} a_i = 2t. \]

Thus, the running time is still $O(t)$.

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 $p_i$ 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 $v$, $1 \leq i \leq n$.
  - The processor assigned to the max element writes that value to MAX.

Efficiency of this algorithm is very poor!

- Divide and crush.

Maximum-finding: CRCW (2)

More efficient (but slower) algorithm:
- Given: $n$ processors.
- Find maximum for each of $n/2$ pairs in constant time.
- Find max for $n/8$ groups of 4 elements (using 8 proc/group) each in constant time.
- Square the group size each time.
- Total time: $O(\log \log n)$.

If $s$ is sequential complexity, then the modified algorithm has $O(1)$ efficiency.

Maximum-finding: CRCW

- Need $O(n^2)$ processors
- Need only constant time.
- Efficiency is $1/n$.

$n/2$ processors
- $n$ processors, using previous “divide and crush” algorithm.

This leaves $n/8$ elements which can be broken into $n/128$ groups of 16 elements with 128 processors assigned to each group. And so on.

Efficiency is $1 / \log \log n$. 
Parallel Prefix

- Let * 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 \( PR(i, j) = x_i \cdot x_{i+1} \cdot \ldots \cdot x_j \).
- We want to compute \( PR(1, k) \) for \( 1 \leq k \leq n \).
- We want to compute \( PR(1, k) \) for \( 1 \leq k \leq n \).
- Sequential alg: Compute each prefix in order
  - \( O(n) \) time required (using previous prefix)
- Approach: Divide and Conquer
  - IH: We know how to solve for \( n/2 \) elements.
  - \( PR(1, k) \) and \( PR(n/2 + 1, n/2 + k) \) for \( 1 \leq k \leq n/2 \).
  - \( PR(1, m) \) for \( n/2 < m \leq n \) comes from \( PR(1, n/2) \cdot PR(n/2 + 1, m) \) – from IH.

Complexity:

- \( T(n, n) = O(\log n) \):
  - \( E(n, n) = O(1/\log n) \).
- Brent's lemma no help: \( O(n \log n) \) total steps.

Better Parallel Prefix

- \( E \) is the set of all \( x \)'s with \( i \) even.
- If we know \( PR(1, 2i) \) for \( 1 \leq i \leq n/2 \) then \( PR(1, 2i + 1) = PR(1, 2i) \cdot x_{2i+1} \).
- Algorithm:
  - Compute in parallel \( x_{2i} = x_{2i-1} \cdot x_{2i} \) for \( 1 \leq i \leq n/2 \).
  - Solve for \( E \) (by induction).
  - Compute in parallel \( x_{2i+1} = x_{2i} \cdot x_{2i+1} \).
- Complexity:
  - \( T(n, n) = O(\log n) \).
  - \( S(n) = S(n/2) + n - 1 \), so \( S(n) = O(n) \).
  - for \( S(n) \) the total number of steps required to process \( n \) elements.
- So, by Brent's Lemma, we can use \( O(n/\log n) \) processors for \( O(1) \) efficiency.

Routing on a Hypercube

Goal: Each processor \( P_i \) simultaneously sends a message to processor \( P_{i(i)} \) 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.

We don't just want the sum or min of all – we want all the partials as well.

That is – no processors are “excessively” idle.

Since the E's already include their left neighbors, all info is available to get the odds.

There is only one recursive call, instead of two in the previous algorithm.

Need EREW model for Brent's Lemma.

Need a figure
Randomizing Switching Algorithm

It can be shown that any deterministic algorithm is $\Omega(2^n)$ for some $a > 0$, where $2^n$ is the number of messages.

A node $i$ (and its corresponding message) has binary representation $i = k_1 k_2 \cdots k_n$.

Randomization approach:
(a) Route each message from $i$ to $j$ to a random processor $r$ (by a randomly selected route).
(b) Continue the message from $r$ to $j$ by the shortest route.

n-dimensional hypercube has $2^n$ nodes.

Remember that we want parallel algorithms with cost $\log n$, not cost $n^a$!
The distance from any processor $i$ to another processor $j$ is only $\log n$ steps.

Randomized Switching (2)

Phase (a):
for (each message at $i$)
  cobegin
    for (k = 1 to n)
      $T[i, k] = \text{RANDOM}(0, 1)$;
    for (k = 1 to n)
      if ($T[i, k] = 1$)
        Transmit $i$ along dimension $k$;
  coend;

Randomized Switching (3)

Phase (b):
for (each message $i$)
  cobegin
    for (k = 1 to n)
      $T[i, k] = \text{Current}[i, k] \text{ EXCLUSIVE OR Dest}[i, k]$;
    for (k = 1 to n)
      if ($T[i, k] = 1$)
        Transmit $i$ along dimension $k$;
  coend;

Randomized Switching (4)

With high probability, each phase completes in $O(\log n)$ time.
- It is possible to get a really bad random routing, but this is unlikely.
- In contrast, it is very possible for any correlated group of messages to generate a bottleneck.
Sorting on an array

Given: \( n \) processors labeled \( P_1, P_2, \ldots, P_n \) 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 \text{ArraySort}(X, n) \{
    \text{do in parallel } \lceil n/2 \rceil \text{ times } \{
        \text{Exchange-compare}(P[2i-1], P[2i]); // Odd}
        \text{Exchange-compare}(P[2i], P[2i+1]); // Even
    \}
\}

A simple algorithm, but will it work?

Parallel Array Sort

Any algorithm that correctly sorts 1’s and 0’s by comparisons will also correctly sort arbitrary numbers.

Correctness of Odd-Even Transpose

Theorem 12.2: When Algorithm ArraySort terminates, the numbers are sorted.

Proof: By induction on \( n \).

Base Case: 1 or 2 elements are sorted with one comparison/exchange.

Induction Step:
- Consider the maximum element, say \( x_m \).
- Assume \( m \) odd (if even, it just won’t exchange on first step).
- This element will move one step to the right each step until it reaches the rightmost position.

Correctness (2)

- The position of \( x_m \) follows a diagonal in the array of element positions at each step.
- Remove this diagonal, moving comparisons in the upper triangle one step closer.
- The first row is the \( n \)th 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 $a_{\text{odd}}$ with $b_{\text{odd}}, a_{\text{even}}$ with $b_{\text{even}}$, yielding $o_1, o_2, \cdots, o_n$ and $e_1, e_2, \cdots, e_n$ respectively.

Batcher's Algorithm Correctness

**Theorem 12.3:** For all $i$ such that $1 \leq i \leq n-1$, we have $x_{2i} = \min(o_{i+1}, e_i)$ and $x_{2i+1} = \max(o_{i+1}, e_i)$.

**Proof:**

- Since $e_i$ is the $i$th element in the sorted even sequence, it is at least $i$ even elements.
- For each even element, $e_i$ is also at least an odd element.
- So, $e_i \geq 2i$ elements, or $e_i \geq x_{2i}$.
- In the same way, $o_{i+1} \geq i+1$ odd elements, $\geq$ at least $2i$ elements all together.
- So, $o_{i+1} \geq x_{2i}$.
- By the pigeonhole principle, $e_i$ and $o_{i+1}$ must be $x_{2i}$ and $x_{2i+1}$ (in either order).

See Manber Figure 12.11.
**Batcher Sort Complexity**

- Total number of comparisons for merge:
  
  \[ T_M(2n) = 2T_M(n) + n - 1; \quad T_M(1) = 1. \]

  Total number of comparisons is \( O(n \log n) \), but the depth of recursion (parallel steps) is \( O(\log n) \).

- Total number of comparisons for the sort is:
  
  \[ T_S(2n) = 2T_S(n) + O(n \log n); \quad T_S(2) = 1. \]

  So, \( T_S(n) = O(n \log^2 n) \).

- The circuit requires \( n \) processors in each column, with depth \( O(\log^2 n) \), for a total of \( O(n \log^2 n) \) processors and \( O(\log^2 n) \) time.

- The processors only need to do comparisons with two inputs and two outputs.

---

**Matrix-Vector Multiplication**

**Problem:** Find the product \( x = Ab \) of an \( m \) by \( n \) matrix \( A \) with a column vector \( b \) 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.

See Manber Figure 12.17.