CS5114: Theory of Algorithms

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

Review of Mathematical Induction

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

Principle of Mathematical Induction

- **IF** the following two statements are true:
  - \( P(c) \) is true.
  - For \( n > c. P(n-1) \) is true \( \rightarrow P(n) \) 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 \( P(n-1) \rightarrow P(n) \). 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 \( A \rightarrow B \) 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: \( \forall n \geq 1, \forall x \) such that \( 1 + x > 0 \), \( (1 + x)^n \geq 1 + nx \).

Induction Example 3

Theorem: \( 2s \) 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...

Base Case: \( P(n) \) is true since \( S(1) = 1 = 1(1 + 1)/2 \).

Induction Hypothesis: \( S(i) = \frac{i(i+1)}{2} \) for \( i < n \).

Induction Step:

\[ S(n) = S(n-1) + n = (n-1)n/2 + n \]

Therefore, \( P(n-1) \rightarrow P(n) \).

By the principle of Mathematical Induction, \( \forall n \geq 1, S(n) = \frac{nx}{2} \).

MI is often an ideal tool for verification of a hypothesis. Unfortunately it does not help to construct a hypothesis.

What do we do induction on? Can’t be a real number, so must be \( n \).

\[ P(n) : (1 + x)^n \geq 1 + nx. \]

Base Case: \( (1 + x)^1 = 1 + x \geq 1 + 1x \)

Induction Hypothesis: Assume \( (1 + x)^{n-1} \geq 1 + (n - 1)x \)

Induction Step:

\[ (1 + x)^n = (1 + x)(1 + x)^{n-1} \]
\[ \geq (1 + x)(1 + (n - 1)x) \]
\[ = 1 + nx - x + nx^2 - x^2 \]
\[ = 1 + nx + (n - 1)x^2 \]
\[ \geq 1 + nx. \]

Base case: \( 4 = 2 + 2 \).

Induction Hypothesis: Assume \( P(k) \) for \( 4 \leq k < n \).

Induction Step:

Case 1: \( n - 1 \) is made up of all \( 2c \) stamps. Then, replace 2 of these with a \( 5c \) stamp.

Case 2: \( n - 1 \) includes a \( 5c \) stamp. Then, replace this with 3 \( 2c \) stamps.

Induction is useful for much more than checking equations!

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

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

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

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

**Strong Induction**

IF the following two statements are true:
1. \( P(c) \)
2. \( P(i), i = 1, 2, \ldots, n-1 \rightarrow P(n) \)

... 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** (\( n \geq 3 \)):
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.

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

- go backwards from \( n \).
- prove a stronger IH.

to make the most of the IH.

It should be obvious that the theorem is true for an undirected graph.

Naive approach: Assume the theorem is true for any graph of \( n - 1 \) vertices. Now add the \( n \)th vertex and its edges. But this won’t work for the graph \( 1 \rightarrow 2 \). Initially, vertex 1 is the independent set. We can’t add 2 to the graph. Nor can we reach it from 1.

Going forward is good for proving existence.
Going backward (from an arbitrary instance into the IH) is usually necessary to prove that a property holds in all instances. This is because going forward requires proving that you reach all of the possible instances.

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

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

There are two cases:
- \( S(H) \cup \{v\} \) 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:
\[
F(n)^2 + F(n+1)^2 = F(2n+1) \text{ and } \\
F(n)^2 + 2F(n)F(n-1) = F(2n).
\]

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 \( S' \) be \( S - \{h\} \) for some horse \( h \).
- By IH, all horses in \( S' \) have the same color.
- Let \( h' \) be some horse in \( S' \).
- IH implies \( \{h, h'\} \) have all the same color.

Therefore, \( P(n) \) 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 \( n > n_0 \).

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 $\leq$:

$n^2 \in O(n^2)$ and $n^2 \log n \in O(n^2)$, but

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

In other words, an $O$ notation is a “best case”.

A common misunderstanding:

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

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


For $n^2$, if $n = 1000$, then $n'$ would be 1003.

Compare $T(n) = n^2$ to $T(n) = n \log n$. For $n > 58$, it is faster to have the $O(n \log n)$ algorithm than to have a computer that is 10 times faster.

Some Rules for Use

**Definition:** $f$ is **monotonically growing** if $n_1 \geq n_2$ implies $f(n_1) \geq f(n_2)$.

We typically assume our time complexity function is monotonically growing.

**Theorem 3.1:** Suppose $f$ is monotonically growing.

$\forall c > 0$ and $\forall a > 1, (f(n))^c \in O(a^{f(n)})$.

In other words, an **exponential** function grows faster than a **polynomial** function.

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

- $f(n) + g(n) \in O\left(s(n) + r(n)\right) \equiv O\left(\max\left(s(n), r(n)\right)\right)$
- $f(n)g(n) \in O\left(s(n)\cdot r(n)\right)$
- If $s(n) \in O(h(n))$ then $f(n) \in O(h(n))$
- For any constant $k$, $f(n) \in O\left(k s(n)\right)$
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: \( 5n^2 + 4n^2 + 9n + 7 = \Theta(n^2) \).

Other Asymptotic Notation (cont)

\( o(f(n)) \) – little o (\( < \))

**Definition:** \( g(n) \in o(f(n)) \) if \( \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0 \)

Ex: \( n^2 \in o(n^3) \)

\( \omega(f(n)) \) – little omega (\( > \))

**Definition:** \( g(n) \in w(f(n)) \) if \( f(n) \in o(g(n)) \).

Ex: \( n^5 \in w(n^6) \)

\( \infty(f(n)) \)

**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 < \( n^3 \).
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

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

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

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

Recurrence Relations

- A (math) function defined in terms of itself.
- Example: Fibonacci numbers:
  \[ F(n) = F(n-1) + F(n-2) \quad \text{general case} \]
  \[ F(1) = F(2) = 1 \quad \text{base cases} \]
- 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: \( T(n) \leq E(T, n) \).

This is Manber Problem 2.5.

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

Recurrence Relations

- A (math) function defined in terms of itself.
- Example: Fibonacci numbers:
  \[ F(n) = F(n-1) + F(n-2) \quad \text{general case} \]
  \[ F(1) = F(2) = 1 \quad \text{base cases} \]
- 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: $T(n) \leq c_1 n^3 = f(n)$

$T(1) = 7$ implies that $c_1 \geq 7$

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

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

$$\leq 2c_1(n/2)^3 + 5n^2$$

$$\leq c_1(n^3/4) + 5n^2$$

$$\leq c_1n^3$$ if $c_1 \geq 20/3$.

---

Guessing (cont)

Therefore, if $c_1 = 7$, a proof by induction yields:

$$T(n) \leq 7n^3$$

$$T(n) \in O(n^3)$$

Is this the best possible solution?

---

Guessing (cont)

Guess again.

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

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

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

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

$$\leq 2c_2(n/2)^2 + 5n^2$$

$$= c_2(n^2/2) + 5n^2$$

$$\leq c_2n^2$$ 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 $20/7 n^3 + 5n^2 = 20/3 n^3$ when $n = 1$, and as $n$ grows, the right side grows even faster.

No - try something tighter.

Because $10 n^2 + 5n^2 = 10n^2$ 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 $T(2^k) \leq 10(2^k)^2$.

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

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

$$\leq 40n^2$$

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)

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

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

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

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

$$\cdots + 2 \cdot 5(n/2)^2 + 5n^2$$

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

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

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; \( n = b^m \).

\[
T(n) = a(T(n/b^2) + c(n/b^k) + cr^k) = a^n T(1) + a^{n-1} c(n/b^{m-1}) + \ldots + ac(n/b^k) + cr^k = ca^n \sum b^k/a^i
\]

\[a^m = 2^{\log_b n} = n^{\log_b 2}\]

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) \( r < 1 \).

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

\[T(n) = \Theta(a^n) = \Theta(n^{\log_b a})\]

(2) \( r = 1 \).

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

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

D & C Recurrences (Case 3)

(3) \( r > 1 \).

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

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

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

Summary

Theorem 3.4:

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

Apply the theorem:

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

\[a = 3, \ b = 5, \ c = 8, \ k = 2\]

\[b^k/a = 25/3\]

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

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

Useful log Notation

- If you want to take the log of \((\log n)\), it is written \(\log\log n\).
- \((\log n)^2\) can be written \(\log^2 n\).
- Don’t get these confused!
- \(\log^n n\) 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 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: \(m_1 + m_2 \cdot n = m_1 + m_2 \cdot n_1\).

Alternate Analysis

Use amortized analysis on multiple calls to \(\text{push}\\, \text{pop}\\):

Cannot \(\text{pop}\\) 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

An analogy to 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:
- \( Q(1), Q(2), \ldots, Q(n) \)

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

Induction

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

Don’t forget the base cases!

Theorem: \( \forall n \geq 1 \), we can solve instances in \( Q(n) \).
- This theorem embodies the correctness of the algorithm.

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

Just one (new) catch:
- Different inductive proofs are possible.
- We want the most efficient algorithm!

Interval Containment

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

Example:
- \([6, 9], [5, 7], [0, 3], [4, 8], [6, 10], [7, 8], [0, 5], [1, 3], [6, 8]\)

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

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

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

I \( \rightarrow \) A is reasonably obvious – we often use induction to prove that an algorithm is correct. The intellectual claim of Manber is that I \( \rightarrow \) A gives insight into problem solving.
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 \( n \)th interval against intervals \( i = 1, 2, \ldots \)
  - 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 \( n \)th interval.

Choose the \( n \)th interval to have rightmost left endpoint, and if there are ties, leftmost right endpoint.
1. No need to check whether \( n \)th interval contains other intervals.
2. \( n \)th interval should be marked iff the rightmost endpoint of the first \( n-1 \) intervals exceeds or equals the right endpoint of the \( n \)th 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 \( n \)th interval is \( \leq R \), then mark the \( n \)th interval.
- Else \( R \leftarrow \) right endpoint of \( n \)th interval.

Analysis: \( \Theta(n\log n) + \Theta(n) \).

Lesson: Preprocessing, often sorting, can help sometimes.
Maximal Induced Subgraph

**Problem:** Given a graph $G = (V, E)$ and an integer $k$, find a maximal induced subgraph $H = (U, F)$ such that all vertices in $H$ have degree $\geq k$.

Example: Scientists interacting at a conference. Each one will come only if $k$ colleagues come, and they know in advance if somebody won’t come.

Example: For $k = 3$.

Solution:

Max Induced Subgraph Solution

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

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

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

Celebrity Problem

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

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

How should we structure the information?

Celebrity Problem (cont)

Formulate as an $n \times n$ boolean matrix $M$.

$M_{ij} = 1$ iff $i$ knows $j$.  

Example:

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

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

There can be at most one celebrity.

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

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

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.

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

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:
- $O(n)$ 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 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: \(\text{TRAILINGSUM}(n-1)\) 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 \(\text{TRAILINGSUM}(n-1)\) for any sequence of \(n-1\) integers.

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

**Induction step:**
\[\text{SUM}(n) = \text{Max}(\text{SUM}(n-1), \text{TRAILINGSUM}(n-1) + x_n).\]
\[\text{TRAILINGSUM}(n) = \text{Max}(0, \text{TRAILINGSUM}(n-1) + x_n).\]
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?

O(n). $T(n) = T(n - 1) + 2$.
Remember position information as well.
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:
  \[
  T(n) = 2T(n-1) + c\quad n \geq 2 \]
  \[
  T(n) = \Theta(2^n) \quad \text{by expanding sum.}
  \]
- Alternate: change variable from \( n \) to \( m = 2^n \).
  \[
  2T(m/2) + cn^2.
  \]
  From Theorem 3.4, we get \( \Theta(m^{log_2 3}) = \Theta(2^n) \).
- 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>( k )</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>I</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>O</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>I</td>
<td>–</td>
<td>I/O</td>
<td>–</td>
</tr>
<tr>
<td>( k_4 )</td>
<td>4</td>
<td>O</td>
<td>–</td>
<td>I</td>
<td>–</td>
<td>I</td>
<td>O</td>
<td>–</td>
<td>O</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>( k_5 )</td>
<td>1</td>
<td>O</td>
<td>I</td>
<td>O</td>
<td>I</td>
<td>O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/O</td>
<td>I/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 \( P(n - 1, k - k) \).

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 \( P(2, 2) = P(2, 3) = 7 \) 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 \( P(5, 10) \).

\[
\begin{array}{c c c c c c}
M(1, 0) & M(1, 9) \\
M(2, 2) & M(2, 9) \\
M(3, 9)
\end{array}
\]

The result is an \( n \)-level DAG.

Dynamic Programming

This approach of storing solutions to subproblems in a table is called \textit{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 \textit{optimization problems}.

Fibonacci Sequence

```c
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

```c
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
\[
\text{COST}((A \times B) \times C) = \text{COST}(A \times (B \times C)) =
\]

View as binary trees:

Order Matters

Example:
\[
A = 2 \times 8; B = 8 \times 5; C = 5 \times 20
\]
\[
\text{COST}((A \times B) \times C) = \text{COST}(A \times (B \times C)) =
\]

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, \( c[1, n] = \min_{1 \leq i \leq n-1} r_i + c[1, i] + c[i + 1, n] \).
Analysis

Base Cases: For $1 \leq k \leq n$, $c[k, k] = 0$.
More generally:
$$c[i, j] = \min_{1 \leq k \leq j-1} n_{i-1, j} n_i + c[i, k] + c[k+1, j]$$
Solving $c[i, j]$ requires $2(j-i)$ recursive calls.
Analysis:
$$T(n) = \sum_{k=1}^{n-1} (T(k) + T(n-k)) = 2\sum_{k=1}^{n-1} T(k)$$
$$T(1) = 1$$
$$T(n+1) = T(n) + 2T(n) = 3T(n)$$
$$T(n) = \Theta(3^n)$$ Ugh!
But there are only $\Theta(n^2)$ values $c[i, j]$ to be calculated!

Dynamic Programming

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

<table>
<thead>
<tr>
<th></th>
<th>c[1, 1]</th>
<th>c[1, 2]</th>
<th>...</th>
<th>c[1, n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>c[2, 2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td>c[n, n]</td>
</tr>
</tbody>
</table>

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

Dynamic Programming Analysis

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

Summary

- Dynamic programming can often be added to an inductive proof to make the resulting algorithm as efficient as possible.
- Can be useful when divide and conquer fails to be efficient.
- Usually applies to optimization problems.
- Requirements for dynamic programming:
  - 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.
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 such that records $R_{k_1}, R_{k_2}, ..., R_{k_n}$ have keys obeying the property $k_1 \leq k_2 \leq ... \leq k_n$.

Measures of cost:
- Comparisons
- Swaps

Insertion Sort

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

Best Case:
Worst Case:
Average Case:

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.

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

Initial call: `qsort(array, 0, n-1);`
## 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
        swap(A, l, r); // Swap out-of-place vals
        while (l < r); // Stop when they cross
        swap(A, l, r); // Reverse wasted swap
    } while (l < r); // Stop when they cross
    return l; // Return first position in right
}
```

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

### Partition Example

<table>
<thead>
<tr>
<th>Initial</th>
<th>72 6 57 88 85 42 83 73 48 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 1</td>
<td>72 6 57 88 85 42 83 73 48 60</td>
</tr>
<tr>
<td>Pass 2</td>
<td>48 6 57 88 85 42 83 73 48 60</td>
</tr>
<tr>
<td>Pass 3</td>
<td>48 6 57 42 85 88 83 73 48 60</td>
</tr>
<tr>
<td>Reverse Swap</td>
<td>48 6 57 42 85 88 83 73 48 60</td>
</tr>
</tbody>
</table>

### Quicksort Example

<table>
<thead>
<tr>
<th>72 657884283734860</th>
<th>Pivot = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 657426088838560</td>
<td>Pivot = 6</td>
</tr>
<tr>
<td>42 578385838583</td>
<td>Pivot = 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6 48 57 83 85 86</th>
<th>Final Sorted Array</th>
</tr>
</thead>
</table>

### 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 be expensive? The area under this curve must be low, since the average cost is $\Theta(n \log n)$, but some of the values cost $\Theta(n^2)$. So there can be VERY few of the expensive ones.

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

---

**Notes:**

- No notes for Quicksort Partition or Quicksort Example.
## Quicksort Average Cost

The recursive function is:

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

Since the two halves of the summation are identical,

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

Multiplying both sides by \( n \) yields:

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

### Average Cost (cont.)

To get rid of the full history by subtracting \( nf(n) \) from \( (n+1)f(n+1) \)

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

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

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

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

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

### Average Cost (cont.)

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

Expand the recurrence to get:

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

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

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

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

### Average Cost (cont.)

\[
f(n+1) \leq 2 \left( 1 + \frac{n+2}{n+1} + \frac{n+2}{n+1} + \ldots + \frac{3}{2} \right)
\]

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

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

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

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

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.

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

Cost for heap construction:

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

Heapsort

Heapsort uses a max-heap.

```c
void heapsort(Elem* A, int n) { // Heapsort
    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: \( \Theta(k \log n) \).

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:

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

Ways to generalize:

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

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

Cost:

The simple version only works for a permutation of 0 to n – 1, but it is truly $O(n!)$!
Support duplicates, i.e., larger key space Cost might look like $\Theta(n^2)$.
Oops! It is actually, $\Theta(n \times \text{Maxkeyvalue})$.
Maxkeyvalue could be $O(n^2)$ or worse.

Radix Sort

Initial list:

<table>
<thead>
<tr>
<th>First pass (on 1st digit)</th>
<th>Second pass (on 2nd digit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 7 9 1 1 7 3 3 3 3 0 7 5 6 7 2 5</td>
<td>1 7 9 1 1 7 3 3 3 3 0 7 5 6 7 2 5</td>
</tr>
</tbody>
</table>

Radix Sort
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.

Radix Sort Example

Sorting Lower Bound

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

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

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

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

Form of proof:

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

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

Lower Bound Analysis

\[
\log n! \leq \log n^n = n \log n.
\]

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

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

External Sorting

Problem: Sorting data sets too large to fit in main memory.
- Assume data stored on disk drive.

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

An external sort should minimize disk accesses.

Model of External Computation

- Secondary memory is divided into equal-sized blocks (512, 2048, 4096 or 8192 bytes are typical sizes).
- The basic I/O operation transfers the contents of one disk block to/from main memory.
- Under certain circumstances, reading blocks of a file in sequential order is more efficient. (When?)
- Typically, the time to perform a single block I/O operation is sufficient to Quicksort the contents of the block.
- Thus, our primary goal is to minimize the number of block I/O operations.
- Most workstations today must do all sorting on a single disk drive.

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.
- Read new input blocks as needed.
- 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.
  else
  Replace the root with the last key in the array.
  Add the next record in the file to a new heap (actually, stick it at the end of the array).

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 $2b$ blocks long (on average).
- How big a file can be merged in one pass?
- Larger files will need more passes – but the run size grows quickly!
- This approach trades $\Theta(\log b)$ (possibly) sequential passes for a single or a very few random (block) access passes.

General Principals of External Sorting

In summary, a good external sorting algorithm will seek to do the following:
- Make the initial runs as long as possible.
- At all stages, overlap input, processing and output as much as possible.
- Use as much working memory as possible. Applying more memory usually speeds processing.
- If possible, use additional disk drives for more overlapping of processing with I/O, and allow for more sequential file processing.