CS 5114: Theory of Algorithms

Clifford A. Shaffer
Department of Computer Science
Blacksburg, Virginia

Spring 2010

Copyright © 2010 by Clifford A. Shaffer

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:
  1. \( P(c) \) is true.
  2. 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:
  1. Base case
  2. State induction Hypothesis
  3. 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: \( \forall n \geq 1, \forall \text{real } x \text{ such that } 1 + x > 0, (1 + x)^n \geq 1 + nx. \)

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.

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.

\[ \text{Base Case: } n = 1. \text{ Any line divides the plane into two regions.} \]

\[ \text{Induction Hypothesis: } \text{It is possible to two-color the regions formed by } n - 1 \text{ lines.} \]

\[ \text{Induction Step: } \text{Introduce the } n^{\text{th}} \text{ line.} \]

This line cuts some colored regions in two. Reverse the region colors on one side of the \( n^{\text{th}} \) line. A valid two-coloring results.

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

Strong Induction

IF the following two statements are true:
\[ P(c) \]
\[ P(i), i = 1, 2, \ldots, n - 1 \rightarrow P(n), \]
... 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 \textbf{Independent Set} of vertices is one for which no two vertices are adjacent.

Theorem: Let \( G = (V, E) \) be a \textbf{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 \textbf{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.

\[ \text{Base Case: } \text{Easy if } n \leq 3 \text{ because there can be no path of length } > 2. \]

\[ \text{Induction Hypothesis: } \text{The theorem is true if } |V| < n. \]

\[ \text{Induction Step } (n > 3): \]

Pick any \( v \in V \).

Define: \[ N(v) = \{ v \} \cup \{ w \in V | (v, w) \in E \}. \]

\[ H = G - N(v). \]

Since the number of vertices in \( H \) is less than \( n \), there is an independent set \( S(H) \) that satisfies the theorem for \( H \).

N\((v)\) is all vertices reachable (directly) from \( v \). That is, the Neighbors of \( v \).

\( H \) is the graph induced by \( V - N(v) \).

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

\[
F(1) = F(2) = 1 \\
F(n) = F(n-1) + F(n-2), \quad n > 2.
\]

**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:**

\[
\begin{align*}
F(n)^2 + F(n+1)^2 &= F(2n+1) \quad \text{and} \\
F(n)^2 + 2F(n)F(n-1) &= F(2n).
\end{align*}
\]

**Induction Hypothesis:**

\[
\begin{align*}
F(n-1)^2 + F(n)^2 &= F(2n-1) \quad \text{and} \\
F(n-1)^2 + 2F(n-1)F(n-2) &= F(2n-2).
\end{align*}
\]

### 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 \(S' = S - \{h\}\) 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: $T(input)$
- 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.

**Solution:** Asymptotic analysis

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

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

**Asymptotic Analysis**

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

**O Notation**

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 "≤":
\( n^2 \in O(n^2) \) and \( n^2 \log n \in O(n^2) \), but
- \( n^2 \neq n^2 \log n \)
- \( n^2 \in O(n^2) \) while \( n^2 \log n \notin O(n^2) \)

Growth Rate Graph

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

Speedups

What happens when we buy a computer 10 times faster?

<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>1000</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>5nlog n</td>
<td>250</td>
<td>1,840</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{16n} )</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
- \( f(n) + g(n) = O(s(n) + r(n)) \equiv O(\max(s(n), r(n))) \)
- \( f(n)g(n) = O(s(n)r(n)) \)
- If \( s(n) \in O(h(n)) \) then \( f(n) \in O(h(n)) \)
- For any constant k, \( f(n) \in O(ks(n)) \)

Assume monotonically growing because larger problems should take longer to solve. However, many real problems have “cyclically growing” behavior.
Is \( O(2^{f(n)}) \in O(3^{g(n)}) \)? 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, \( T(n) \) 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 \( n \geq n_0 \).

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

\[ \Theta(f(n)) \] – Exact bound (\( \approx \))

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

\[ o(f(n)) \] – little o (\(<\))

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

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

\[ w(f(n)) \] – little omega (\( >\))

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

Ex: \( n^3 \in w(n^3) \)

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

**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 \( a + b + c = 1 \).

For \( n = 2 \) we have \( 8a + 4b + 2c = 5 \).

For \( n = 3 \) we have \( 27a + 9b + 3c = 14 \).

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

Now, prove the solution with induction.

---

**Technique 2: Shifted Sums**

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

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

Shift by multiplying by \( r \).

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

Subtract.

\[ G(n) - rG(n) = G(n)(1 - r) = a - ar^{n+1} \]

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

---

**Example 3.3**

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

Multiply by 2.

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

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

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

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

---

**Recurrence Relations**

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

\[
T(1) = 7 \quad \text{implies that} \quad c_1 \geq 7
\]

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

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

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

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

\[
\leq c_1 n^3 \quad \text{if} \quad c_1 \geq 20/3.
\]

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.

\[
T(n) \leq c_2 n^2 = g(n)
\]

\[
T(1) = 7 \quad \text{implies} \quad c_2 \geq 7.
\]

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

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

\[
\leq 2c_2(n/2)^2 + 5n^2
\]

\[
= c_2 (n^2/2) + 5n^2
\]

\[
\leq c_2 n^2 \quad \text{if} \quad c_2 \geq 10
\]

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

\[
T(n) = O(n^2).
\]

Is this the best possible upper bound?
Expanding Recurrences (cont)

For arbitrary \( n \), let \( 2^{k-1} < n < 2^k \).

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)

\[
T(n) = 2T(n/2) + 5n^2
= 2(2T(n/4) + 5(n/2)^2) + 5n^2
= 2(2(2T(n/4) + 5(n/4)^2) + 5(n/2)^2) + 5n^2
= 2^k T(1) + 2^{k-1} \cdot 5(n/2^{k-1})^2 + 2^{k-2} \cdot 5(n/2^{k-2})^2 + \cdots + 2 \cdot 5(n/2)^2 + 5n^2
= 7n + 5n^2 \sum_{i=0}^{k-1} 2^i
= 7n + 5n^2(2 - 1/2^{k-1})
= 7n + 5n^2(2 - 2/n).
\]

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; \( n = b^m \).

\[
T(n) = aT(n/b^k) + c(n/b^k)^k + cr^k
\]

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

\[
a^m = a^{b^m} = n^{b^m}
\]

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

\[
T(n) = \Theta(a^m r^m)
\]

\[
= \Theta(a^m (b^k/a)^m)
\]

\[
= \Theta(b^m)
\]

\[
= \Theta(n^k)
\]

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: \( T(n) = 2T(n/2) + n \). 
  \( 2^{1}/2 = 1 \), 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): 
  \( T(n) = 8 T(n/2) + n^2 \). 
  \( 2^{2}/8 = 1/2 \), so \( T(n) = \Theta(n^2) \).

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!
  - \(\log^* n\) means “the number of times that the log of \(n\) must 
    be taken before \(n \leq 1\).
    - For example, \(\text{loglog} 65536 = 2^{16}\) 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 pop more elements than get pushed onto the stack.

After many pushes, a single pop has high potential.

Once that potential has been expended, it is not available for future \(\text{pop}\) operations.

The cost for \(m_1\) pushes and \(m_2\) pops:

\[ m_1 + (m_2 + m_1) = \Theta(m_1 + m_2) \]
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:

- \( Q(1), Q(2), \ldots, Q(n) \)

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:

\[ [6, 9], [5, 7], [0, 3], [4, 8], [6, 10], [7, 8], [0, 5], [1, 3], [6, 8] \]
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 if 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 $M$.

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

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: \(\text{TRAILINGSUM}(n-1)\) 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:
\(\text{SUM}(1) = \text{TRAILINGSUM}(1) = \max(0, x_1)\).

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 $S \subseteq \{1, 2, \cdots, n\}$ such that $\sum_{i \in S} k_i = K$.

Example:
Knapsack capacity $K = 163$.
10 items with sizes
4, 9, 15, 19, 27, 44, 54, 68, 73, 101

Knapsack Algorithm Approach

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

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

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

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

Knapsack Induction

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

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

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 $P(n - 1, K - k_n)$ has a solution, then $P(n, K)$ has a solution.
- Else $P(n, K)$ has no solution.

Algorithm Complexity

- Resulting algorithm complexity:
  $T(n) = 2T(n - 1) + cn$ for $n \geq 2$
  $T(n) = \Theta(2^n)$ by expanding sum.
- Alternate: change variable from $n$ to $m = 2^n$.
  $2T(m/2) + cn^0$.
  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.

Problem: Can’t use Theorem 3.4 in this form.
This form uses $n^0$ because we also need an exponent of $n$ to fit the form of the theorem.