Analysis of Algorithms

• Efficiency measure
  – how long the program runs - time complexity
  – how much memory it uses - space complexity

• Why analyze at all?
  – Confidence: algorithm will work well in practice
  – Insight : alternative, better algorithms

Time Complexity

• We count number of abstract steps
  – Not physical runtime in seconds
  – Not every machine instruction

• What is one abstract step?
  – Each of these statements is one step
    • count = count + 1
    • y = a*x^3 + b*x + c
    • if (n ≥ 2 sqrt(m) || n ≤ 0.5 sqrt(m)) ...

Complexity of Algorithms

• Example : X is a set with n elements - some labeled red and others labeled black.

• Problem - find the number of subsets of X that contain at least 1 red element.

• Analysis : Since X has n elements, it has 2^n subsets so an algorithm that generates and examines the subsets of X will require at least 2^n units of time (cycles) to execute.

• Even for relatively small values on n, this can be a large execution time.
  – Note : the time to compute is a complex issue depending on speed of the machine, nature of coding, size of the problem, compilers,…

Algorithm Development Algorithm

AlgorithmDesign(informal problem)
1 formalize problem (mathematically) [Step 0]
2 repeat
3 devise algorithm [Step 1]
4 analyze correctness [Step 2]
5 analyze efficiency [Step 3]
6 refine
7 until algorithm good enough
8 return algorithm
**Example: Finding the Maximum of 3 Numbers**

**Input:** three numbers a, b, c

**Output:** x, the maximum of the 3 numbers (a, b, c)

**Procedure max (a, b, c):**

1. Set x := a
2. If b > x then (b is larger than x, update x)
   - x := b
3. If c > x then (c is larger than x, update x)
   - x := c
4. Return (x)

**Algorithm: Procedure for calculating n!**

**Input:** n, n \(\in\mathbb{Z}_+\)

**Output:** n!

**Procedure factorial (n):**

1. If n = 0 then
   - Return (1)
2. Else
   - Return (n \(\times\) factorial (n - 1))

**Recursive Algorithms**

- A recursive procedure is a procedure that invokes itself.

- A **recursive algorithm** is an algorithm that contains a recursive procedure.
  - e.g. divide and conquer techniques: divide a problem into a collection of problems of the same type as the original problem

**Algorithm - Recursive Computation of GCD**

**Input:** a, b \(\in\mathbb{Z}_{>0}\), not both zero

**Output:** gcd (a, b)

**Procedure gcd_recurse (a, b):**

1. Make a the larger
2. If a < b then
   - Swap (a, b)
3. Else
   - If b = 0 then
     - Return (a)
   - Else
     - // divide a by b to obtain a = qb + r, r \(\in\mathbb{Z}_+\) and r < b
     - Return (gcd_recurse(b, r))
4. End gcd_recurse
Asymptotic Analysis

- Complexity as a function of input size $n$
  
  - $T(n) = 4n + 5$
  
  - $T(n) = 0.5 \cdot n \cdot \log n - 2n + 7$
  
  - $T(n) = 2^n + n^3 + 3n$

- **What happens as $n$ grows?**

---

Rates of Growth

- Suppose we can execute $10^{10}$ ops/sec

<table>
<thead>
<tr>
<th>$n$</th>
<th>$T(n)$</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$10^{-9}s$</td>
<td>$10^{-8}s$</td>
<td>$10^{-7}s$</td>
<td>$10^{-6}s$</td>
<td></td>
</tr>
<tr>
<td>$n \log_2 n$</td>
<td>$10^{-9}s$</td>
<td>$10^{-8}s$</td>
<td>$10^{-7}s$</td>
<td>$10^{-6}s$</td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td>$10^{-7}s$</td>
<td>$10^{-5}s$</td>
<td>$10^{-3}s$</td>
<td>$10^{-1}s$</td>
<td></td>
</tr>
<tr>
<td>$n^3$</td>
<td>$10^{-5}s$</td>
<td>$10^{-3}s$</td>
<td>$0.1s$</td>
<td>$100s$</td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td>$10^{-5}s$</td>
<td>$10^{20}s$</td>
<td>$10^{291}s$</td>
<td>forever!</td>
<td></td>
</tr>
</tbody>
</table>

$10^{5}s = 2.8$ hrs \quad $10^{18}s = 30$ billion years

---

Why do we care?

- Most algorithms are fast for small $n$
  
  - Time difference too small to be noticeable
  
  - External things dominate (OS, disk I/O, …)

- $n$ is typically large in practice
  
  - Databases, internet, graphics, …

- Time difference really shows up as $n$ grows!

---

ASYMPTOTIC GROWTH

We want to compare algorithms to decide which we prefer.

1. But, there are many dimensions along which we might compare. So, we decide that what really matters is running time.

2. But, running time depends on input. So, we decide that what really matters is running time as a function of input size.

3. But, this can be hard to characterize. So, we decide that what really matters is worst-case running time as a function of input size.

4. But, one algorithm might be better on small inputs and the other on large inputs. So, we decide that what really matters is worst-case running time as a function of input size for large inputs.

5. But, this can still be quite hard to determine precisely. So, we decide that what really matters is worst-case running time as a function of input size for large inputs, ignoring constant factors.
Asymptotic Growth

• This leads us directly to a notion of asymptotic growth.
• Function \( g(n) \) is asymptotically bigger than function \( f(n) \) if, for any scaling factor \( c > 0 \), there is some threshold \( n_0 \) such that:
  
  \[
  g(n) > cf(n) \text{ for all } n > n_0.
  \]

By this definition, \( g(n) = n/2 \) is asymptotically bigger than \( f(n) = n^{1/2} \).

Why? Give me any scaling factor \( c \) can I determine \( n_0 \) so that

\[
\frac{n_0}{2} > c\left(n_0^{1/2}\right)
\]

\[
n_0 > 16c n_0^{1/2}
\]

\[
n_0^{1/2} > 16c
\]

\[
n_0 > 256c^2
\]

So, any \( n_0 \) at least that big \( (256c^2) \) will satisfy the conditions of the defn.

On the other hand,

\( g(n) = 2n^2 \) is not asymptotically bigger than \( f(n) = n^2 - n + 5 \).

Why?

Take the constant \( c = 3 \).

We have \( g(n) < cf(n) \) for all \( n \).

Further, \( f(n) \) is not asymptotically bigger than \( g(n) \) either.

Asymptotic Growth

• The notion of asymptotic growth can be used to partition the space of functions.
• Two functions \( f(n) \) and \( g(n) \) are “equal” if neither is asymptotically bigger than the other.
  
  – We write this as \( f(n) = \Theta\left(g(n)\right) \).
• Formally, \( f(n) = \Theta\left(g(n)\right) \) if and only if there exists scaling factors \( c_1 \) and \( c_2 \) and threshold \( n_0 \) such that
  
  \[
  0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n > n_0.
  \]

  read this as “eventually”
**Obtaining Asymptotic Bounds**

- **Eliminate low order terms**
  - $4n + 5 \Rightarrow 4n$
  - $0.5 \, n \log n - 2n + 7 \Rightarrow 0.5 \, n \log n$
  - $2^n + n^2 + 3n \Rightarrow 2^n$

- **Eliminate coefficients**
  - $4n \Rightarrow n$
  - $0.5 \, n \log n \Rightarrow n \log n$
  - $n \log n^2 = 2 \, n \log n \Rightarrow n \log n$

---

**Race Against Time!**

<table>
<thead>
<tr>
<th>Race #</th>
<th>$T_1(n)$</th>
<th>$T_2(n)$</th>
<th>Which grows faster?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$n^3 + 2n^2$</td>
<td>$100n^2 + 1000$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$n^{0.1}$</td>
<td>$\log n$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$n + 100n^{0.1}$</td>
<td>$2n + 10 \log n$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$5n^2$</td>
<td>$n!$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$n^{-15}2^n/100$</td>
<td>$1000n^{15}$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$82\log n$</td>
<td>$3n^7 + 7n$</td>
<td></td>
</tr>
</tbody>
</table>
Race 3

\[ n + 100n^{0.1} \quad \text{vs.} \quad 2n + 10 \log n \]

Race 5

\[ n^{-15} \frac{2^n}{100} \quad \text{vs.} \quad 1000n^{15} \]

Race 4

\[ 5n^5 \quad \text{vs.} \quad n! \]

Race 6

\[ 8^2 \log(n) \quad \text{vs.} \quad 3n^7 + 7n \]
Race Against Time! (2)

<table>
<thead>
<tr>
<th>Race #</th>
<th>( T_1(n) )</th>
<th>( T_2(n) )</th>
<th>Which grows faster?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( n^3 + 2n^2 )</td>
<td>( 100n^2 + 1000 )</td>
<td>( T_2 ) : O( (n^2) )</td>
</tr>
<tr>
<td>2</td>
<td>( n^{0.1} )</td>
<td>log ( n )</td>
<td>( T_2 ) : O( (\log n) )</td>
</tr>
<tr>
<td>3</td>
<td>( n + 100n^{0.1} )</td>
<td>( 2n + 10 \log n )</td>
<td>Tie: O( (n) )</td>
</tr>
<tr>
<td>4</td>
<td>( 5n^5 )</td>
<td>( n! )</td>
<td>( T_1 ) : O( (n^5) )</td>
</tr>
<tr>
<td>5</td>
<td>( n^{-15}2n/100 )</td>
<td>( 100n^{15} )</td>
<td>( T_2 ) : O( (n^{15}) )</td>
</tr>
<tr>
<td>6</td>
<td>( 8^{2\log n} )</td>
<td>( 3n^7 + 7n )</td>
<td>( T_1 ) : O( (n^6) )</td>
</tr>
</tbody>
</table>

Terminology

- \( g(n) \in O(f(n)) \)
  - \( \exists \) constants \( c \) and \( n_0 \) such that \( g(n) \leq c f(n) \) \( \forall n \geq n_0 \)
  - e.g. 1, \( \log n \), \( n \), \( 100n \) \( \in O(n) \)

- \( g(n) \in \Omega(f(n)) \)
  - \( \exists \) constants \( c \) and \( n_0 \) such that \( g(n) \geq c f(n) \) \( \forall n \geq n_0 \)
  - e.g. \( n/10 \), \( n^2 \), \( 100 \cdot 2^n \), \( n^3 \log n \) \( \in \Omega(n) \)

- \( g(n) \in \Theta(f(n)) \)
  - \( g(n) \in O(f(n)) \) and \( g(n) \in \Omega(f(n)) \)
  - e.g. \( n^4 \), \( 2n \), \( 100n \), \( 0.01 n + \log n \) \( \in \Theta(n) \)

Typical Growth Rates

- constant: \( O(1) \)
- logarithmic: \( O(\log n), \log_2 n, \log n^2 \in O(\log n) \)
- poly-log: \( O(\log^k n) \)
- linear: \( O(n) \)
- log-linear: \( O(n \log n) \)
- superlinear: \( O(n^{1+c}) \) (\( c \) is a constant > 0)
- quadratic: \( O(n^2) \)
- cubic: \( O(n^3) \)
- polynomial: \( O(n^k) \) (\( k \) is a constant)
- exponential: \( O(c^n) \) (\( c \) is a constant > 1)

Terminology (2)

- \( g(n) \in o(f(n)) \)
  - \( g(n) \in O(f(n)) \) and \( g(n) \not\in \Theta(f(n)) \)
  - e.g. 1, \( \log n \), \( n^{0.99} \) \( \in o(n) \)

- \( g(n) \in \omega(f(n)) \)
  - \( g(n) \in \Omega(f(n)) \) and \( g(n) \not\in \Theta(f(n)) \)
  - e.g. \( n^{1.01} \), \( n^2 \), \( 100 \cdot 2^n \), \( n^3 \log n \) \( \in \omega(n) \)
Terminology (3)

- Roughly speaking, the correspondence is
  \[ O \leq \Omega \geq \theta \leq o < \omega > \]

- If \( P_k(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \)
  is a polynomial of degree \( k \) and each \( a_i \) is nonnegative, then
  \[ P_k(n) = \Theta(n^k) \]

Types of Analysis

Three orthogonal axes:
- **Bound flavor**
  - upper bound \((O, o)\)
  - lower bound \((\Omega, \omega)\)
  - asymptotically tight \((\theta)\)
- **Analysis case**
  - worst case (adversary)
  - average case
  - best case
- **Analysis quality**
  - loose bound (most true analyses)
  - tight bound (\( \exists \) no better bound which is asymptotically different)

Analyzing Code

- **General guidelines**
  - Simple C / C++ operations - constant time
  - consecutive statements - sum of times per stmt
  - conditionals - sum of branches and condition
  - loops - sum over iterations
  - function calls - cost of function body

Simple loops

\[
\begin{align*}
\text{sum} & = 0 \\
\text{for } i = 1 \text{ to } n \text{ do} & \\
\text{for } j = 1 \text{ to } n \text{ do} & \\
\text{sum} & = \text{sum} + 1
\end{align*}
\]
**Conditionals and While Loop**

- **Conditional**
  
  \[
  \begin{align*}
  \text{if } C & \text{ then } S_1 \\
  \text{else } S_2 & 
  \end{align*}
  \]

- **Loops**
  
  \[
  \text{while } C \text{ do } S
  \]

---

**Example: Factorial**

**Analysis by simple calculation**

1. \( T(n) \leq c + c + T(n - 2) \)  
   (by substitution)
2. \( T(n) \leq c + c + c + T(n - 3) \)  
   (by substitution, again)
3. \( T(n) \leq kc + T(n - k) \)  
   (extrapolating \( 0 < k \leq n \))
4. \( T(n) \leq nc + T(0) = nc + b \)  
   (setting \( k = n \))

So, \( T(n) \in \) ?

---

**Recursion**

- **Recursion**
  
  - Almost always yields a recurrence
  - Recursive max

- **Example: Factorial**
  
  \[
  \begin{align*}
  \text{fac}(n) & \\
  \text{if } n = 0 & \text{ return } 1 \\
  \text{else return } n \cdot \text{fac}(n - 1) & 
  \end{align*}
  \]

\[
\begin{align*}
T(0) & = 1 \\
T(n) & \leq c + T(n - 1) \quad \text{if } n > 0
\end{align*}
\]

---

**Example: Mergesort**

- **Mergesort algorithm**
  
  - If list has 1 element, return
  - Otherwise split list in half, sort first half, sort second half, merge sorted "halves" together

\[
\begin{align*}
T(1) & = 1 \\
T(n) & \leq 2T(n/2) + cn \quad \text{if } n > 1
\end{align*}
\]

- Sorting the two halves recursively
- Splitting and merging
Example: Mergesort (2)

Analysis by simple calculation

\[ T(n) \leq 2T(n/2) + cn \]
\[ \leq 2(2T(n/4) + c(n/2)) + cn \]
\[ = 4T(n/4) + cn + cn \]
\[ \leq 4(2T(n/8) + c(n/4)) + cn + cn \]
\[ = 8T(n/8) + cn + cn + cn \]
\[ \vdots \]
\[ \leq 2^kT(n/2^k) + kcn \quad \text{(extrapolating } 1 < k \leq n) \]
\[ \leq nT(1) + cn \log n \quad \text{(for } 2^k = n \text{ or } k = \log n) \]

- \( T(n) \in ? \)

Summary

- Determine what characterizes a problem’s input size
- Express how much resources (time, memory, etc.) an algorithm requires as a function of input size using \( O(\cdot), \Omega(\cdot), \theta(\cdot) \)
  - worst case
  - best case
  - average case
  - common case

- A problem that does not have a worst-case polynomial time algorithm is said to be **intractable**.
- A problem for which there is no algorithm at all is said to be an **unsolvable problem**.
- There are many problems that have not yet been determined, i.e. they are thought to be intractable but no one has yet proved it to be intractable - these problems are called **NP (complete)**