CSE 373: Data Structures and Algorithms

Lecture 5: Math Review/Asymptotic Analysis III
Growth rate terminology

- $T(N) = O(f(N))$
  - $f(N)$ is an **upper bound** on $T(N)$
  - $T(N)$ grows no faster than $f(N)$

- $T(N) = \Omega(g(N))$
  - $g(N)$ is a **lower bound** on $T(N)$
  - $T(N)$ grows at least as fast as $g(N)$

- $T(N) = \Theta(g(N))$
  - $T(N)$ grows at the same rate as $g(N)$

- $T(N) = o(h(N))$
  - $T(N)$ grows strictly slower than $h(N)$
More about asymptotics

• Fact: If \( f(N) = O(g(N)) \), then \( g(N) = \Omega(f(N)) \).

• Proof: Suppose \( f(N) = O(g(N)) \).
  Then there exist constants \( c \) and \( n_0 \) such that
  \[ f(N) \leq c \cdot g(N) \]
  for all \( N \geq n_0 \).

  Then \( g(N) \geq (1/c) \cdot f(N) \) for all \( N \geq n_0 \),
  and so \( g(N) = \Omega(f(N)) \).
Facts about big-Oh

• If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
  $- T_1(N) + T_2(N) = O(f(N) + g(N))$
  $- T_1(N) \times T_2(N) = O(f(N) \times g(N))$

• If $T(N)$ is a polynomial of degree $k$, then:
  $T(N) = \Theta(N^k)$
  $- example: 17n^3 + 2n^2 + 4n + 1 = \Theta(n^3)$

• $\log^k N = O(N)$, for any constant $k$
Complexity cases

• **Worst-case complexity:** “most challenging” input of size \( n \)

• **Best-case complexity:** “easiest” input of size \( n \)

• **Average-case complexity:** random inputs of size \( n \)

• **Amortized complexity:** \( m \) “most challenging” *consecutive* inputs of size \( n \), divided by \( m \)
Bounds vs. Cases

Two orthogonal axes:

• Bound
  – Upper bound ($O, o$)
  – Lower bound ($\Omega$)
  – Asymptotically tight ($\Theta$)

• Analysis Case
  – Worst Case (Adversary), $T_{\text{worst}}(n)$
  – Average Case, $T_{\text{avg}}(n)$
  – Best Case, $T_{\text{best}}(n)$
  – Amortized, $T_{\text{amort}}(n)$

One can estimate the bounds for any given case.
Example

List.contains(Object o)
• returns true if the list contains o; false otherwise
• Input size: $n$ (the length of the List)
• $T(n)$ = “running time for size $n”$
• But $T(n)$ needs clarification:
  – Worst case $T(n)$: it runs in at most $T(n)$ time
  – Best case $T(n)$: it takes at least $T(n)$ time
  – Average case $T(n)$: average time
Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

<table>
<thead>
<tr>
<th>Class</th>
<th>Big-Oh</th>
<th>If you double N, ...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>O(1)</td>
<td>unchanged</td>
<td>10ms</td>
</tr>
<tr>
<td>logarithmic</td>
<td>O(log₂ N)</td>
<td>increases slightly</td>
<td>175ms</td>
</tr>
<tr>
<td>linear</td>
<td>O(N)</td>
<td>doubles</td>
<td>3.2 sec</td>
</tr>
<tr>
<td>log-linear</td>
<td>O(N log₂ N)</td>
<td>slightly more than doubles</td>
<td>6 sec</td>
</tr>
<tr>
<td>quadratic</td>
<td>O(N²)</td>
<td>quadruples</td>
<td>1 min 42 sec</td>
</tr>
<tr>
<td>cubic</td>
<td>O(N³)</td>
<td>multiplies by 8</td>
<td>55 min</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>exponential</td>
<td>O(2^N)</td>
<td>multiplies drastically</td>
<td>5 * 10^{61} years</td>
</tr>
</tbody>
</table>
Recursive programming

• A method in Java can call itself; if written that way, it is called a *recursive method*

• The code of a recursive method should be written to handle the problem in one of two ways:
  – **base case**: a simple case of the problem that can be answered directly; does not use recursion.
  – **recursive case**: a more complicated case of the problem, that isn't easy to answer directly, but can be expressed elegantly with recursion; makes a recursive call to help compute the overall answer
Recursive power function

- Defining powers recursively:

  \[ \begin{align*}
  \text{pow}(x, 0) &= 1 \\
  \text{pow}(x, y) &= x \times \text{pow}(x, y-1), \quad y > 0
  \end{align*} \]

// recursive implementation
public static int pow(int x, int y) {
    if (y == 0) {
        return 1;
    } else {
        return x * pow(x, y - 1);
    }
}
Searching and recursion

• Problem: Given a sorted array $a$ of integers and an integer $i$, find the index of any occurrence of $i$ if it appears in the array. If not, return -1.
  – We could solve this problem using a standard iterative search; starting at the beginning, and looking at each element until we find $i$
  – What is the runtime of an iterative search?

• However, in this case, the array is sorted, so does that help us solve this problem more intelligently? Can recursion also help us?
Binary search algorithm

• Algorithm idea: Start in the middle, and only search the portions of the array that might contain the element $i$. Eliminate half of the array from consideration at each step.
  – can be written iteratively, but is harder to get right

• called **binary search** because it chops the area to examine in half each time
  – implemented in Java as method
    ```java
    Arrays.binarySearch in java.util package
    ```
Binary search example

i = 16

```
0  4   min
1  7
2  16
3  20   mid (too big!)
4  37
5  38
6  43   max
```
Binary search example

i = 16

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>16</td>
<td>20</td>
<td>37</td>
<td>38</td>
<td>43</td>
</tr>
</tbody>
</table>

min
mid (too small!)
max
Binary search example

\[ i = 16 \]

\[
\begin{array}{c}
0 & 4 \\
1 & 7 \\
2 & 16 \\
3 & 20 \\
4 & 37 \\
5 & 38 \\
6 & 43 \\
\end{array}
\]

\text{min, mid, max (found it!)}
Binary search pseudocode

binary search array $a$ for value $i$:
  if all elements have been searched,
    result is -1.
  examine middle element $a[mid]$.
  if $a[mid]$ equals $i$,
    result is $mid$.
  if $a[mid]$ is greater than $i$,
    binary search left half of $a$ for $i$.
  if $a[mid]$ is less than $i$,
    binary search right half of $a$ for $i$. 
• How do we analyze the runtime of binary search and recursive functions in general?

• binary search either exits immediately, when input size <= 1 or value found (base case), or executes itself on 1/2 as large an input (rec. case)
  – $T(1) = c$
  – $T(2) = T(1) + c$
  – $T(4) = T(2) + c$
  – $T(8) = T(4) + c$
  – ...
  – $T(n) = T(n/2) + c$

• How many times does this division in half take place?
Divide-and-conquer

• **divide-and-conquer algorithm**: a means for solving a problem that first separates the main problem into 2 or more smaller problems, then solves each of the smaller problems, then uses those sub-solutions to solve the original problem
  – 1: "divide" the problem up into pieces
  – 2: "conquer" each smaller piece
  – 3: (if necessary) combine the pieces at the end to produce the overall solution

  – binary search is one such algorithm
Recurrences, in brief

• How can we prove the runtime of binary search?

• Let's call the runtime for a given input size $n$, $T(n)$. At each step of the binary search, we do a constant number $c$ of operations, and then we run the same algorithm on $1/2$ the original amount of input. Therefore:

  - $T(n) = T(n/2) + c$
  - $T(1) = c$

• Since $T$ is used to define itself, this is called a recurrence relation.
Solving recurrences

• **Master Theorem:**
  A recurrence written in the form
  \[ T(n) = a \times T(n / b) + f(n) \]
  (where \( f(n) \) is a function that is \( O(n^k) \) for some power \( k \))
  has a solution such that

  \[ T(n) = O(n^{\log_b a}), \quad a > b^k \]

  \[ T(n) = O(n^k \log n), \quad a = b^k \]

  \[ O(n^k), \quad a < b^k \]

• This form of recurrence is very common for divide-and-conquer algorithms
Runtime of binary search

• Binary search is of the correct format:
  \[ T(n) = a \times T(n/b) + f(n) \]
  
  - \( T(n) = T(n/2) + c \)
  - \( T(1) = c \)

  - \( f(n) = c = O(1) = O(n^0) \) ... therefore \( k = 0 \)
  - \( a = 1, b = 2 \)

• \( 1 = 2^0 \), therefore:
  \[ T(n) = O(n^0 \log n) = O(\log n) \]

• (recurrences not needed for our exams)
Which Function Dominates?

<table>
<thead>
<tr>
<th>f(n)</th>
<th>g(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^3 + 2n^2$</td>
<td>$100n^2 + 1000$</td>
</tr>
<tr>
<td>$n^{0.1}$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>$n + 100n^{0.1}$</td>
<td>$2n + 10 \log n$</td>
</tr>
<tr>
<td>$5n^5$</td>
<td>$n!$</td>
</tr>
<tr>
<td>$n^{-15} 2^n/100$</td>
<td>$1000n^{15}$</td>
</tr>
<tr>
<td>$8^{2\log n}$</td>
<td>$3n^7 + 7n$</td>
</tr>
</tbody>
</table>

What we are asking is: is $f = O(g)$? Is $g = O(f)$?
Race I

\[ f(n) = n^3 + 2n^2 \quad \text{vs.} \quad g(n) = 100n^2 + 1000 \]
Race II

\( n^{0.1} \) vs. \( \log n \)
Race III

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

$5n^5$ vs. $n!$
Race V

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

$8^{2 \log(n)}$ vs. $3n^7 + 7n$
A Note on Notation

You'll see...

\[ g(n) = O(f(n)) \]

and people often say...

\[ g(n) \text{ is } O(f(n)). \]

These really mean

\[ g(n) \in O(f(n)). \]

That is, \( O(f(n)) \) represents a set or class of functions.