Overview

- HW5 Due & HW6 assignment will be online soon!
- HW6 is due in 1.5 weeks on 11/3 Tuesday

- Last Lecture (Lec15 on 10/15 and 20)
  - Sequence, Subsequence, Summation
  - Theory of Counting
  - Theory of Counting: real number is not countable
  - Pigeonhole Principle
  - Generalized Pigeonhole Principle

- This Lecture: Algorithm Analysis
  - Definition of algorithms
  - Complexity function
  - Algorithm complexity
  - Big-O notation

Chapter 5. Introduction to Analysis of Algorithms
What is Algorithm?

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

We will use a pseudocode to specify algorithms.

**Example 1**: an algorithm that finds the maximum element in a finite sequence

**Function max** (List[1], List[2],…, List[i],…, List[n]) // a list of n integers

max = List[1]
for i = 2 to n
if, max < List[i], then, max = List[i]

// max is the largest element in List[1…n]

**Example 2**: a linear search algorithm, that is, an algorithm that linearly solves a problem of searching a sequence for a particular element.

**Function linear_search** (List[1], List[2],…, List[n], x)

i = 1 // May assume all numbers in List are distinct
while (i ≤ n and x ≠ List[i])
    i = i + 1
if i ≤ n then location = i
else location = -1
// if location = -1, x is not found; otherwise List[location] = x

If the numbers in List are ordered (or sorted), a binary search algorithm is more efficient than linear search.
**Function binary_search(List[1], List[2], ..., List[n], x)**

// numbers in List are sorted in increasing order

\( i = 1 \)  // 1st index in List

\( j = n \)  // last index in List

while (\( i < j \))

\( m = \lfloor (i + j)/2 \rfloor \)

if \( x > \text{List}[m] \) then \( i = m+1 \)  // upper half

else if \( x < \text{List}[m] \) then \( j = m-1 \)  // lower half

else \( i = j = m \)  // found \( x \)

if \( x = \text{List}[i] \) then \( \text{location} = i \)

else \( \text{location} = -1 \)

---

1. Write Pseudocode
2. Complexity function

- Given an algorithm with an input sequence with \( n \) elements,

- A complexity function specifies **the number of basic operations** (e.g., if-condition comparison) to be executed in order to complete the computation with the \( n \) elements.

- Described as a function of \( n \), unique to a given algorithm

- **Can be directly derived from a pseudocode!**
Complexity Function Examples

Maximum difference between any two numbers in input sequence

**Function** \texttt{max\_diff(List[1], List[2], ..., List[n])}

\[ m = 0 \]

\[ \text{for } i = 1 \text{ to } n-1 \]

\[ \text{for } j = i + 1 \text{ to } n \]

\[ \text{if } | \text{List}[i] - \text{List}[j] |= m \text{ then} \]

\[ m = | \text{List}[i] - \text{List}[j] | \]

Comparisons: \( n-1 + n-2 + n-3 + \ldots + 1 = \frac{(n-1)n}{2} = 0.5n^2 - 0.5n \)

Another algorithm solving the same problem:

**Function** \texttt{max\_diff(List[1], List[2], ..., List[n])}

\[ \text{min} = \text{List}[1] \]

\[ \text{max} = \text{List}[1] \]

\[ \text{for } i = 2 \text{ to } n \]

\[ \text{if } \text{List}[i] < \text{min} \text{ then } \text{min} = \text{List}[i] \]

\[ \text{else if } \text{List}[i] > \text{max} \text{ then } \text{max} = \text{List}[i] \]

\[ m = | \text{max} - \text{min} | \]

Comparisons: \( 2(n-1) = 2n - 2 \)

Why not counting \texttt{min=List[1]} as a one step?

Well, there are three extra steps so really \( 2(n-1) + 3 = 2n + 1 \)!

But this does not matter (you see it later)
**Algorithm Complexity by complexity function**

- **Time complexity**: a measure of the time required (or total steps) to solve a problem of a particular size.

- **Space complexity**: a measure of the space required (or total memory) to solve a problem of a particular size. (will not discuss this….)

- In general, we are not so much interested in the time and space complexity for small inputs.

- For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with \( n = 10 \); but it is gigantic for \( n = 2^{30} \).

- For example, let us assume two algorithms \( A \) and \( B \) that solve the same class of problems.

  - The complexity function of \( A \) is \( 5,000n \), the one for \( B \) is \( \lceil 1.1^n \rceil \) for an input with \( n \) elements.

  - For \( n = 10 \), \( A \) requires 50,000 steps, but \( B \) only 3, so \( B \) seems to be superior to \( A \).

  - For \( n = 1000 \), however, \( A \) requires 5,000,000 steps, while \( B \) requires \( 2.5 \cdot 10^{47} \) steps.

  - This means that algorithm \( B \) cannot be used for large inputs, while algorithm \( A \) is still feasible.

  - So what is important is the **growth** of the complexity functions.

  - The growth of time and space complexity with increasing input size \( n \) is a suitable measure for the comparison of algorithms.
Comparison: time complexity of algorithms A and B

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Algorithm A</th>
<th>Algorithm B</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>5,000n</td>
<td>\sqrt{1.1^n}</td>
</tr>
<tr>
<td>10</td>
<td>50,000</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>500,000</td>
<td>13,781</td>
</tr>
<tr>
<td>1,000</td>
<td>5,000,000</td>
<td>2.5 \times 10^4</td>
</tr>
</tbody>
</table>
How to derive big-O notation of a complexity function?

> choose \( g(n) \) out of the list then find \( C \) and \( k \! \)

- Example: Show that \( f(x) = x^2 + 2x + 1 \) is \( O(x^2) \).
  
  For \( x > 1 \) we have:
  \[
  x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 \quad \Rightarrow \quad x^2 + 2x + 1 \leq 4x^2
  \]
  Therefore, for \( C = 4 \) and \( k = 1 \), \( f(x) \leq Cx^2 \) whenever \( x > k \).

- Example: Show that \( f(n) = 6*2^n + n^2 \) is \( O(2^n) \).
  
  We have:
  \[
  6*2^n + n^2 \leq 6*2^n + 2^n \quad \text{for all } n \geq 4
  \]
  \[
  6*2^n + n^2 \leq 7*2^n
  \]
  Therefore, for \( C = 7 \) and \( k = 4 \), \( f(n) \leq C2^n \) whenever \( x > k \).

- Question: If \( f(x) \) is \( O(x^2) \), is it also \( O(x^3) \)?
  
  Yes. \( x^2 \) grows faster than \( x^3 \), so \( x^3 \) grows also faster than \( f(x) \).

Therefore, we always have to find the smallest simple function \( g(x) \) for which \( f(x) \) is \( O(g(x)) \).
2-Step Complexity Analysis: Choose \( g(n) \) and Prove \( O(g) \)!

“Popular” reference functions \( g(n) \) are:

\[ 1 < \log n < n - \log n < n^2 < n^3 < n^n < 2^n < 10^n < n! \]

(above are listed from slowest to fastest growth)

A problem that can be solved with polynomial worst-case complexity is called **tractable**.

Problems of higher complexity are called **intractable**.

Problems that no algorithm can solve are called **unsolvable**. (more on this later...)

You will find out more about this in future courses.

\[ f(n) \leq Cg(n) \text{ for } n \geq k \]

---

**Useful Rules for Big-O**

- For any polynomial \( f(x) = a_nx^n + a_{n-1}x^{n-1} + ... + a_0 \), where \( a_n, a_{n-1}, ..., a_0 \) are real numbers, \( f(x) \) is \( O(x^n) \).

- If \( f_1(x) \) is \( O(g_1(x)) \) and \( f_2(x) \) is \( O(g_2(x)) \), then \( (f_1 + f_2)(x) \) is \( O(\max(g_1(x), g_2(x))) \).

- If \( f_1(x) \) is \( O(g(x)) \) and \( f_2(x) \) is \( O(g(x)) \), then \( (f_1 + f_2)(x) \) is \( O(g(x)) \).

- If \( f_1(x) \) is \( O(g_1(x)) \) and \( f_2(x) \) is \( O(g_2(x)) \), then \( (f_1f_2)(x) \) is \( O(g_1(x)g_2(x)) \).
Complexity Function Examples

Function `max_diff(List[1], List[2], ..., List[n])`

\[
\begin{align*}
m &= 0 \\
\text{for } i &= 1 \text{ to } n-1 \\
\quad &\text{for } j = i + 1 \text{ to } n \\
\quad &\quad \text{if } |List[i] - List[j]| > m \text{ then} \\
\quad &\quad \quad m = |List[i] - List[j]| \\
\end{align*}
\]

// m is the maximum difference between any
// two numbers in the input sequence

Comp. Func.: \( n-1 + n-2 + n-3 + ... + 1 = (n - 1)n/2 = 0.5n^2 - 0.5n \)

Time complexity is \( O(n^2) \).

Another algorithm solving the same problem:

Function `max_diff(List[1], List[2], ..., List[n])`

\[
\begin{align*}
\text{min} &= \text{List[1]} \\
\text{max} &= \text{List[1]} \\
\text{for } i &= 2 \text{ to } n \\
\quad &\quad \text{if } \text{List}[i] < \text{min} \text{ then } \text{min} = \text{List}[i] \\
\quad &\quad \text{else if } \text{List}[i] > \text{max} \text{ then } \text{max} = \text{List}[i] \\
\quad \quad m &= |\text{max} - \text{min}| \\
\end{align*}
\]

Comp. Func.: \( 2(n-1) = 2n - 2 \)

Time complexity is \( O(n) \).