Overview

- HW5 due on Tuesday 4/7
- MT2 in one week on 4/9 (Please prepare!)
- Next lecture on 4/7 will be on the MT2 review

- Last Lecture
  - Sequence, Subsequence, Summation
  - Theory of Counting
    - Theory of Counting: real number is not countable

- This Lecture
  - Pigeonhole Principle
  - Generalized Pigeonhole Principle
  - Introduction to algorithm analysis
  - Complexity function
  - Algorithm complexity
  - Big-O notation

Pigeonhole Principle: If $x$ items are placed into $y$ bins where $x > y$, then there is one bin which contains at least two items.

Note: textbook specified this property by using function notation

Proof by contradiction: Suppose $x$ (> $y$) items are placed into $y$ bins and suppose no bin with two or more items. Each bin has 0 or 1 item. The maximum total number of items in $y$ bins is then $y$, but $y < x$. This contradiction proves the result

Example: How many people must be in a room to guarantee that two people have last names that begin with the same letter?

What is the minimum $x$ so that $x > 26$?

There are 26 letters (or bins). If there are 27 people, then at least 2 people will have last names beginning with the same letter.
Example: The population of city x is 40,000. If each resident has three initials, is it true that there must be at least 2 individuals with the same initials?

How many possible combination of 3 letters?

\[26 \times 26 \times 26 = 17,576 < 40,000\]

By Pigeonhole principle, there must be at least 2 individuals with the same initials.

**Generalized Pigeonhole Principle:** If x items are placed into y bins where \(x > y\), then there is one bin which contains at least \(\lceil \frac{x}{y} \rceil\) items.

Example: For above example, it is true that there must be at least 3 individuals with the same initials, i.e. \(\lceil \frac{40000}{17576} \rceil = 3\).

Example: What is the minimum number of people in a group there must be so that there are at least 3 who were born in the same month?

There are 12 months (bins). With > 12 people, at least 2 people were born in the same month.

With > 24 people, at least 3 people who were born in the same month.

So, minimum number is 25 people.

Example: Assume you have a drawer containing a random distribution of a dozen brown socks and a dozen black socks. It is dark, so how many socks do you have to pick to be sure that among them there is a matching pair?

Ans: ??
Application - Problem with divisors:
Let \( m \in \mathbb{N} \). Given \( m \) integers \( a_1, a_2, \ldots, a_m \), there exist \( i \) and \( j \) with \( 0 < i < j \leq m \) such that \( a_{i+1} + a_{i+2} + \ldots + a_j \) is divisible by \( m \).

Proof (in two cases):
• Consider \( m \) sums:
  \[
  a_1, \ a_1 + a_2, \ a_1 + a_2 + a_3, \ldots, a_1 + a_2 + \ldots + a_m
  \]
• 1) If any of these sums is divisible by \( m \), then we are done!
• 2) Suppose not, each sum has a nonzero remainder when divided by \( m \).
• The possible remainders are 1, 2, 3,\ldots, \( m-1 \).
• By Pigeonhole principle, there are at least 2 sums with same remainder (\( r \)). We have:
  \[
  a_1 + a_2 + \ldots + a_i = cm + r \quad (i)
  \]
  \[
  a_1 + a_2 + \ldots + a_j = dm + r \quad (ii)
  \]
  where \( c, d \) and \( r \) are integers and assume \( i < j \)
• \( a_{i+1} + a_{i+2} + \ldots + a_j = (d-c) m \quad (ii) - (i) \)
It is divisible by \( m \).
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)

```plaintext
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)

```plaintext
// numbers in List are sorted in increasing order
i = 1  // 1st index in List
j = n  // last index in List
while (i < j)
    m = ⌊(i + j)/2⌋
    if x > List[m] then i = m + 1  // upper half
    else if x < List[m] then j = m - 1  // lower half
    else i = j = m  // found x
if x = List[i] then location = i
else location = -1
```
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 \( \text{max_diff}(\text{List}[1], \text{List}[2], \ldots, \text{List}[n]) \)

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

Comparisons: \((n-1) + (n-2) + (n-3) + \ldots + 1 = (n - 1)n/2\)
\[
= 0.5n^2 - 0.5n
\]
Another algorithm solving the same problem:

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

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

**Comparisons: 2(n-1) = 2n – 2**

Why not counting ‘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 $[1.1^n]$ 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 $2.5 \cdot 10^{41}$ 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>$[1.1^n]$</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>$2.5 \cdot 10^{41}$</td>
</tr>
<tr>
<td>1,000,000</td>
<td>$5 \times 10^9$</td>
<td>$4.8 \times 10^{41392}$</td>
</tr>
</tbody>
</table>
The Growth of Functions

• The growth of functions is usually described using the big-O notation.

• Definition of big-O: Let $f$ and $g$ be functions from the integers to the real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants $C$ and $k$ such that

$$|f(x)| \leq C|g(x)| \text{ whenever } x > k.$$  

(upper-bound)

• To analyze the growth of complexity functions $f(n)$ we determine its big-O with reference function $g(n)$ ($1 < \log n < n < \log n < n^2 < n^3 < 2^n < n!$). Both $f(n)$ and $g(n)$ are always positive. Therefore, we can simplify the big-O requirement to:

$$f(n) \leq C \cdot g(n) \quad \text{whenever } n > k.$$  

Note: If we want to show that $f(n)$ is $O(g(n))$, we only need to find one pair $(C, k)$ (which is never unique).

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:
  $$x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 \Rightarrow x^2 + 2x + 1 \leq 4x^2$$
  Therefore, for $C = 4$ and $k = 1$, $f(x) \leq Cx^2$ whenever $x > k$.

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

• Question: If $f(x)$ is $O(x^2)$, is it also $O(x^3)$?
  Yes. $x^3$ grows faster than $x^2$, 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 $O(g(x))$. 
• “Popular” reference functions g(n) are:
  1, log-n, 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.

**Useful Rules for Big-O**

• For any polynomial \( f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 \), where \( a_0, a_1, \ldots, a_n \) 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 \( O(g_1(x) \cdot g_2(x)) \).
Complexity Function Examples

Function max_diff(List[1], List[2], …, List[n])

\[
m = 0
\]

for i = 1 to n - 1

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

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

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

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

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

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

Another algorithm solving the same problem:

Function max_diff(List[1], List[2], …, List[n])

\[
\text{min} = a_1
\]

\[
\text{max} = a_1
\]

for i = 2 to n

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

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

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

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

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