Introduction To Algorithm Analysis

COMP 215 Lecture 1
Definitions

- **Problem**: A question we want an answer for.
- **Examples**:
  1) Sort a list $S$ of $n$ numbers in nondecreasing order.
     - Answer is the sorted sequence.
  2) Determine whether the number $x$ is in the list $S$ of $n$ numbers.
     - Answer is yes or no.

- $S$, $n$ and $x$ are **parameters** to these problems.
- Picking specific values for the parameters results in a problem **instance**.
- A **solution** to an instance of a problem is the answer to that instance.
More Definitions

- **Algorithm** – a clearly specified procedure for finding the solution to any instance of a given problem.
- Of course, there can be many algorithms for a single problem.
Search Example

Determine whether the number $x$ is in the list $S$ of $n$ numbers.
Answer is yes or no.
Sequential Search Pseudocode

```plaintext
//return the location of the item matching x, or 0 if //no such item is found.

index SequentialSearch(keytype[] S, int n, keytype x) {
    index location = 1;
    while (location <= n && S[location] != x) {
        location++;
    }
    if (location > n) {
        location = 0;
    }
    return location;
}
```
A Better Search Algorithm

- If we have $N$ items in the array, sequential search might need to make $N$ comparisons.
- There is a better way...
Binary Search Pseudocode

//return the location of the item matching x, or 0 if //no such item is found. S must be sorted.
index BinarySearch(keytype[] S, int n, keytype x)
{
    index low, high, mid, location;
    low = 1; high = n;
    location = 0;

    while (low <= high && location == 0) {
        mid = floor((low + high) / 2);
        if (x == S[mid])
            location = mid;
        else if (x < S[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
    return location
}
Efficiency Analysis

- We have two algorithms for searching.
- Which is better?
- What do we mean by better?
  - memory usage?
  - ease of implementation?
  - speed?
- In this class, we are almost always interested in speed.
- So... How do we measure speed?
The Experimental Approach

- Easy. Just code the algorithm and time how long it takes to run for different sized inputs.
- Problems:
  - Requires extrapolation. Extrapolation is dangerous.
  - Depends on hardware.
  - We get no insight into the workings of the algorithm.
The Analytic Approach

- Specify a basic operation.
- Specify a way of measuring input size.
- Determine the number of times the basic operation is performed as a function of input size.
- This might vary depending on the input.
- So we can perform:
  - Every-Case analysis: $T(n)$
  - Worst-Case analysis: $W(n)$
  - Average-Case analysis: $A(n)$
  - Best-Case analysis: $B(n)$
Sequential Search Analysis

• Basic operation?
• Input size?
• Every-case?
• Worst-case?
• Average-case?
• Best-case?
Binary Search Analysis

- Worst-case?
- Average-case?
- Best-case?
## Sequential vs. Binary

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Sequential $W(n)=n$</th>
<th>Binary $W(n)=\log(n)+1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td>1,024</td>
<td>1,024</td>
<td>11</td>
</tr>
<tr>
<td>1,048,576</td>
<td>1,048,576</td>
<td>21</td>
</tr>
<tr>
<td>4,294,967,296</td>
<td>4,294,967,296</td>
<td>33</td>
</tr>
</tbody>
</table>
Computing the $n$th Fibonacci Term

• The Fibonacci sequence:
  – $f_0 = 1$
  – $f_1 = 1$
  – $f_n = f_{n-1} + f_{n-2}$

• $1, 1, 2, 3, 5, 8, 13, 21...$
Recursive Algorithm

- The recursive definition leads naturally to a recursive algorithm:

```c
int fib(int n) {
    if (n <= 1) return 1;
    else return fib(n-1) + fib(n-2);
}
```
Recursion Tree...
Number of Terms Computed

<table>
<thead>
<tr>
<th>n</th>
<th>terms computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
</tr>
</tbody>
</table>
fib1 Analysis

• The # of terms computed more than doubles every time \( n \) increases by 2.
  \[
  T(n) > 2 \times T(n-2) \\
  > 2 \times 2 \times T(n-4) \\
  > 2 \times 2 \times 2 \times T(n-6) \\
  \vdots \\
  > 2 \times 2 \times 2 \times \ldots \times 2 \times T(0)
  \]

• Therefore \( T(n) > 2^{n/2} \)

• Can we prove that?
A Non-Recursive Algorithm

```c
int fib2(int n)
{
    index i;
    int f[0..n];
    f[0] = 1;
    f[1] = 1;
    if (n > 1) {
        for(i = 2; i <= n; i++)
            f[i] = f[i-1] + f[i-2];
    }
    return f[n]
}
```
# fib1 vs. fib2

- Running times, assuming it takes one ns to compute each term:

<table>
<thead>
<tr>
<th>n</th>
<th>$2^{n/2}$</th>
<th>fib2</th>
<th>fib1</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>1.0X10^6</td>
<td>40ns</td>
<td>1048 micro s</td>
</tr>
<tr>
<td>60</td>
<td>1.1X10^9</td>
<td>60ns</td>
<td>1 s</td>
</tr>
<tr>
<td>80</td>
<td>1.1X10^12</td>
<td>80ns</td>
<td>18 min</td>
</tr>
<tr>
<td>100</td>
<td>1.1X10^15</td>
<td>100ns</td>
<td>13 days</td>
</tr>
<tr>
<td>120</td>
<td>1.1X10^18</td>
<td>120ns</td>
<td>36 yrs</td>
</tr>
<tr>
<td>160</td>
<td>1.1X10^24</td>
<td>160ns</td>
<td>3.8 X 10^7 yrs</td>
</tr>
<tr>
<td>200</td>
<td>1.1X10^30</td>
<td>200ns</td>
<td>4 X 10^13 yrs</td>
</tr>
</tbody>
</table>
The Points

- The right algorithm can make the difference between fast and impossibly slow.
- Divide and Conquer was great for search. It's not always great.