LINEAR VS BINARY SEARCH

COP 3502
To convince you that this works, let’s look at an example:

- Power(5, 2):
  - Power(5, 2): return 5 * Power(5, 1)
  - Power(5, 1): return 5 * Power(5, 0)
  - Power(5, 0): return 1

Stack trace back to the original function call:
1
5 * 1 = 5
5 * 5 = 25
Recursion

- Why use recursion?
  - Some solutions are naturally recursive.
    - In these cases there might be less code for a recursive solution, and it might be easier to read and understand.

- Why NOT use recursion?
  - Every problem that can be solved with recursion can be solved iteratively.
  - Recursive calls take up memory and CPU time
    - Exponential Complexity – calling the Fib function uses $2^n$ function calls.
  - Consider time and space complexity.
Let’s do another example problem – Fibonacci Sequence
- 1, 1, 2, 3, 5, 8, 13, 21, ...

Let’s create a function `int Fib(int n)`
- we return the nth Fibonacci number
- `Fib(1) = 1, Fib(2) = 1, Fib(3) = 2, Fib(4) = 3, Fib(5) = 5, ...
- What would our base (or stopping) cases be?
Fibonacci

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

- Base (stopping) cases:
  - Fib(1) = 1
  - Fib(2) = 1,

- Then for the rest of the cases: Fib(n) = ?
  - Fib(n) = Fib(n-1) + Fib(n-2), for n>2

- So Fib(9) = ?
  - Fib(8) + Fib(7) = 21 + 13
Recursion - Fibonacci

- See if we can program the Fibonacci example...
Recursion - Fibonacci

Let’s say we called Fibo(5), we can visualize the calls to Fibo on the stack as a tree:
Recursion - Fibonacci

Let’s say we called `Fibo(5)`, we can visualize the calls to `Fibo` on the stack as a tree:
Recursion - Fibonacci

Let’s say we called Fibo(5), we can visualize the calls to Fibo on the stack as a tree:

```
3+2=5
  /   \
2+1=3  1+1=2
  |   |   |
1+1=2 1   1
  |   |   |
1   1   1
```
Write a recursive function that:
- Takes in 2 non-negative integers
- Returns the product
  - Does NOT use multiplication to get the answer

```c
int Multiply(int first, int second) {
    if ((second == 0) || (first == 0))
        return 0;
    else
        return (first + Multiply(first, second - 1));
}
```
In C Programming, we looked at the problem of finding a specified value in an array.

The basic strategy was:

- Look at each value in the array and compare to x
  - If we see that value, return true
  - else keep looking
  - If we’re done looking through the array and still haven’t found it, return false.

```c
int search(int array[], int len, int value) {
    int i;
    for (i = 0; i < len; i++) {
        if (array[i] == value)
            return 1;
    }

    return 0;
}
```
Linear Search

- For an unsorted array, this algorithm is optimal.
  - There’s no way you can definitively say that a value isn’t in the array unless you look at every single spot.

- But we might ask the question, could we find an item in an array faster if it were already sorted?

```c
int search(int array[], int len, int value) {
    int i;
    for (i = 0; i < len; i++) {
        if (array[i] == value)
            return 1;
    }
    return 0;
}
```
Consider the game you probably played when you were little:

- I have a secret number in between 1 and 100, make a guess and I’ll tell you whether your guess is too high or too low.
- Then you guess again, and continue guessing until you guess right.

What would a good strategy for this game be?
If you divide your search space in half each time, you won’t run the risk of searching \( \frac{3}{4} \) of the list each time.

For instance, if you pick 75 for your number, and you get the response “too high”,
Then your number is anywhere from 1-74...

So generally the best strategy is:
Always guess the number that is halfway between the lowest possible value in your search range and the highest possible value in your search range.
How can we adapt this strategy to work for search for a given value in an array?

Given the array:

Search for 19

Where is halfway in between?

One guess would be \((118+2) / 2 = 60\)

But 60 isn’t in the list and the closest value to 60 is 41 almost at the end of the list.

We want the middle INDEX of the array.

In this case: The lowest index is 0, the highest is 8, so the middle index is 4!
Binary Search

- Searching for 19:

Now we ask,

- Is 19 greater than, or less than, the number at index 4?
  - It is Less than, so now we only want to search from index 0 to index 3.
Binary Search

- Searching for 19:
  - The middle of 0 and 3 is 1 (since (3+0)/2 = 1)
  - So we look at array[1]
  - And ask is 19 greater than or less than 6?
    - Since it’s greater than 6, we next search halfway between 2 and 3, which is (2+3)/2 = 2
    - At index 2, we find 19!
Binary Search

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>n/2</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>x₁</td>
<td>x₂</td>
<td>x...</td>
<td>x...</td>
<td>x...</td>
<td>x...</td>
<td>x...</td>
<td>x...</td>
<td>xₙ</td>
</tr>
</tbody>
</table>

```c
int binsearch(int array[], int n, int value) {
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = (low + high)/2;

        if (value < array[mid])
            high = mid - 1;
        else if (value > array[mid])
            low = mid + 1;
        else
            return 1;
    }
    return 0;
}
```
Efficiency of Binary Search

- Now, let’s analyze how many comparisons (guesses) are necessary when running this algorithm on an array of $n$ items.

- First, let’s try $n = 100$:
  - After 1 guess, we have 50 items left,
  - After 2 guesses, we have 25 items left,
  - After 3 guesses, we have 12 items left,
  - After 4 guesses, we have 6 items left,
  - After 5 guesses, we have 3 items left,
  - After 6 guesses, we have 1 item left,
  - After 7 guesses, we have 0 items left.

- The reason we have to list that last iteration is because the number of items left represent the number of other possible values to search.
  - We need to reduce this number to 0!

  Also note that when $n$ is odd, such as when $n = 25$,
  We search the middle element #13,
  There are 12 elements smaller than it and 12 larger,
  So the number of items left is slightly less than 1/2.
Efficiency of Binary Search

- In the general case we get something like:
  - After 1 guess, we have \( \frac{n}{2} \) items left,
  - After 2 guesses, we have \( \frac{n}{4} \) items left,
  - After 3 guesses, we have \( \frac{n}{8} \) items left,
  - ...,
  - After \( k \) guesses, we have \( \frac{n}{2^k} \) items left,
In the general case we get something like:
- After 1 guess, we have \( n/2 \) items left,
- After 2 guesses, we have \( n/4 \) items left,
- After 3 guesses, we have \( n/8 \) items left,
- ...
- After \( k \) guesses, we have \( n/2^k \) items left,

If we can find the value that makes this fraction 1, then we know that in one more guess we’ll narrow down the item:
- \( \frac{n}{2^k} = 1 \), now we just solve for \( k \) (the # of guesses)
Efficiency of Binary Search

In the general case we get something like:

- After 1 guess, we have \( n/2 \) items left,
- After 2 guesses, we have \( n/4 \) items left,
- After 3 guesses, we have \( n/8 \) items left,
- ... 
- After \( k \) guesses, we have \( n/2^k \) items left,

If we can find the value that makes this fraction 1, then we know that in one more guess we’ll narrow down the item:

\[
\frac{n}{2^k} = 1, \text{ now we just solve for } k \text{ (the # of guesses)}
\]

\[
n = 2^k
\]

\[
k = \log_2 n
\]
Efficiency of Binary Search

In the general case we get something like:

- After 1 guess, we have \( \frac{n}{2} \) items left,
- After 2 guesses, we have \( \frac{n}{4} \) items left,
- After 3 guesses, we have \( \frac{n}{8} \) items left,
- ... 
- After \( k \) guesses, we have \( \frac{n}{2^k} \) items left,

If we can find the value that makes this fraction 1, then we know that in one more guess we’ll narrow down the item:

- \( \frac{n}{2^k} = 1 \), now we just solve for \( k \) (the # of guesses)
- \( n = 2^k \)
- \( k = \log_2 n \)

This means that a binary search roughly takes \( \log_2 n \) comparisons when search for a value in a sorted array of \( n \) items.

- This is much much faster than searching linearly!
Efficiency of Binary Search

- Let’s look at a comparison of a linear search to a logarithmic search:

<table>
<thead>
<tr>
<th>n</th>
<th>log n</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
</tr>
<tr>
<td>65536</td>
<td>16</td>
</tr>
<tr>
<td>1048576</td>
<td>20</td>
</tr>
<tr>
<td>33554432</td>
<td>25</td>
</tr>
<tr>
<td>1073741824</td>
<td>30</td>
</tr>
</tbody>
</table>
Recursion

COP 3502
The iterative code is not the easiest to read, if we look at the recursive code
- It’s MUCH easier to read and understand

```c
int binsearch(int *values, int low, int high, int searchVal) {
    int mid;
    if (!terminating condition) {
        // Recursive call
        return 0;
    }
    return 0;
}
```
Recursive Binary Search

- We need a stopping case:
  - We have to STOP the recursion at some point
- Stopping cases:
  1. We found the number!
  2. Or we have reduced our search range to nothing – the number wasn’t found 😞
     - The search range would be empty when low > high

```c
int binsearch(int *values, int low, int high, int searchVal) {
    int mid;
    if (low <= high){
        mid = (low+high)/2;
        if (searchVal == values[mid])
            return 1;
        else if (searchVal > values[mid])
            // Do something else
        else
            // Do something
    }
    return 0;
}
```
Recursive Binary Search

- What are our recursive calls going to be?
  - We need to change what low and high are
  - So we get the following:

```c
int binsearch(int *values, int low, int high, int searchVal) {
    int mid;
    if (low <= high){
        mid = (low+high)/2;
        if (searchVal == values[mid])
            return 1;
        else if (searchVal > values[mid])
            // Do something else
        else
            // Do something else
    }
    return 0;
}
```
Recursive Binary Search

- Binary Search Code summary (using recursion):
  - If the value is found,
    - return 1
  - Otherwise
    - if (searchVal > values[mid])
      - Recursively call binsearch to the right
    - else if (searchVal < values[mid])
      - Recursively call binsearch to the left
  - If low is ever greater than high
    - The value is not in the array return 0
Why Recursion?

- Recursion – behind the scenes
  - Every time we recurse, we are doing another function call, this results in manipulating the run-time stack in memory, passing parameters, and transferring control
    - So recursion costs us both in time and memory usage
Why Recursion?

Recursive Solution

```c
int fact(int n) {
    if (n==1)
        return 1;
    return n*fact(n-1);
}
```

Iterative Solution

```c
int fact(int n) {
    int result = 1;
    while (n > 1) {
        result *= n--;
    }
}```

- More elegant – easier to read:
- But we aren’t seeing the stack manipulations which require:
  - pushing a new n,
  - space for the function’s return value, and updating the stack pointer register
  - and popping off the return value and n when done
Why Recursion?

- If recursion is harder to understand and less efficient, why use it?
  - It leads to elegant solutions – less code, less need for local variables, etc
  - If we can define a function mathematically, the solution is easy to codify
  - Some problems require recursion
    - Tree traversals
    - Graph traversals
    - Search problems
    - Some sorting algorithms (quicksort, mergesort)
      - Note: this is not strictly speaking true, we can accomplish a solution without recursion by using iteration and a stack, but in effect we would be simulating recursion, so why not use it?
      - In some cases, an algorithm with a recursive solution leads to a lesser computational complexity than an algorithm without recursion
        - Compare Insertion Sort to Merge Sort for example
Practice Problem

- Write a recursive function that:
  - Takes in 2 non-negative integers
  - Returns the product
    - Does NOT use multiplication to get the answer

- So if the parameters are 6 and 4
  - We get 24
  - Not using multiplication, we would have to do 6+6+6+6