



LINEAR VS BINARY SEARCH

COP 3502

```

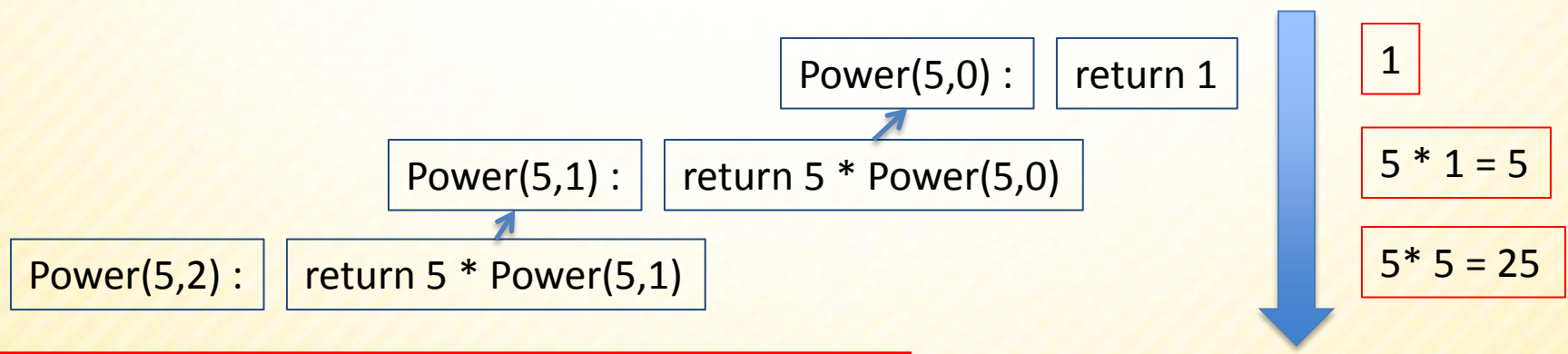
// Pre-conditions:  exponent is >= to 0
// Post-conditions: returns baseexponent

int Power(int base, int exponent) {

    if (exponent == 0)
        return 1;
    else
        return (base*Power(base, exponent - 1));
}

```

- To convince you that this works, let's look at an example:
 - Power(5,2):



STACK

Stack trace back
to the original function call



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.



Recursion Example

- 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
 - $\text{Fib}(1) = 1$, $\text{Fib}(2) = 1$, $\text{Fib}(3) = 2$, $\text{Fib}(4) = 3$, $\text{Fib}(5) = 5$, ...
- What would our base (or stopping) cases be?



Fibonacci

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- Base (stopping) cases:
 - $\text{Fib}(1) = 1$
 - $\text{Fib}(2) = 1,$
- Then for the rest of the cases: $\text{Fib}(n) = ?$
 - $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2),$ for $n > 2$
- So $\text{Fib}(9) = ?$
 - $\text{Fib}(8) + \text{Fib}(7) = 21 + 13$



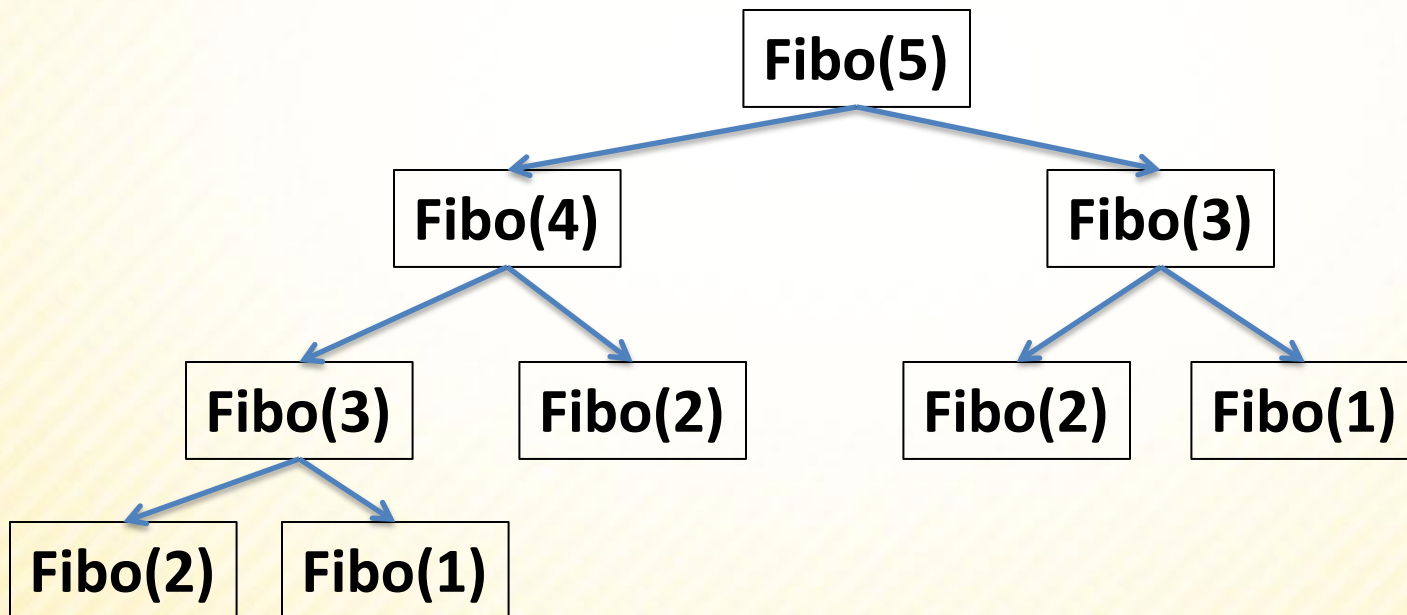
Recursion - Fibonacci

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



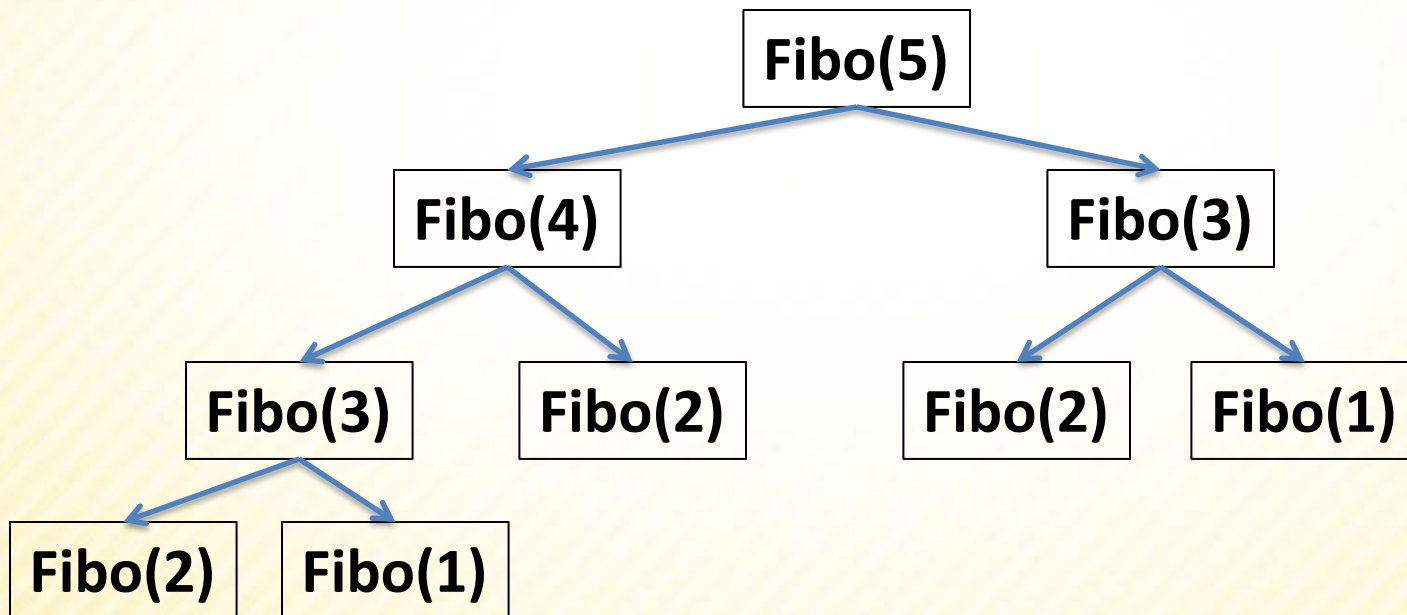
Recursion - Fibonacci

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



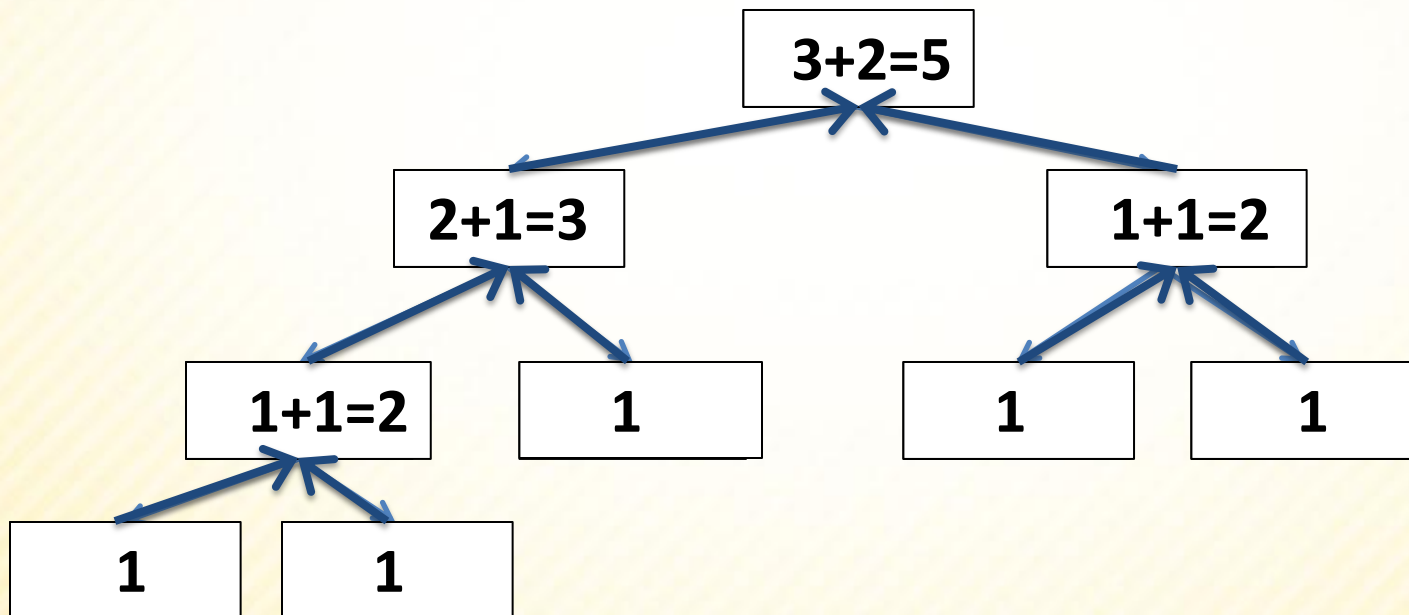
Recursion - Fibonacci

- 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 $\text{Fibo}(5)$, we can visualize the calls to Fibo on the stack as a tree:



Linear Search

- 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.

```
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?

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

    return 0;
}
```



Binary Search

- 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?



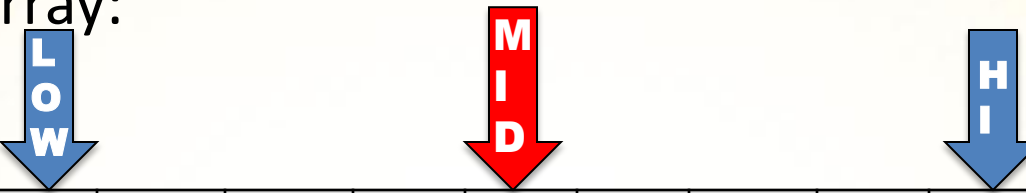
Binary Search

- 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.



Binary Search

- How can we adapt this strategy to work for search for a given value in an array?
- Given the array:




Index	0	1	2	3	4	5	6	7	8
Value	2	6	19	27	33	37	38	41	118

- 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:



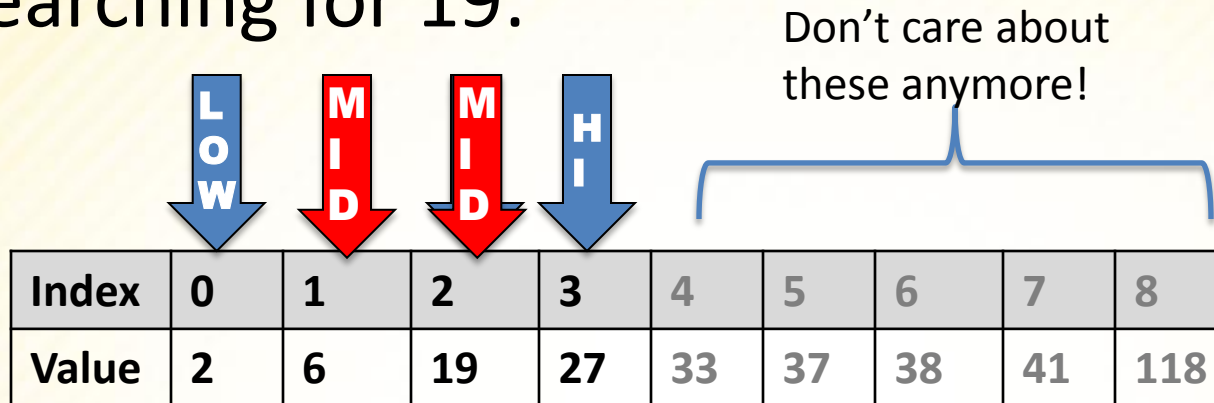
Index	0	1	2	3	4	5	6	7	8
Value	2	6	19	27	33	37	38	41	118

- 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

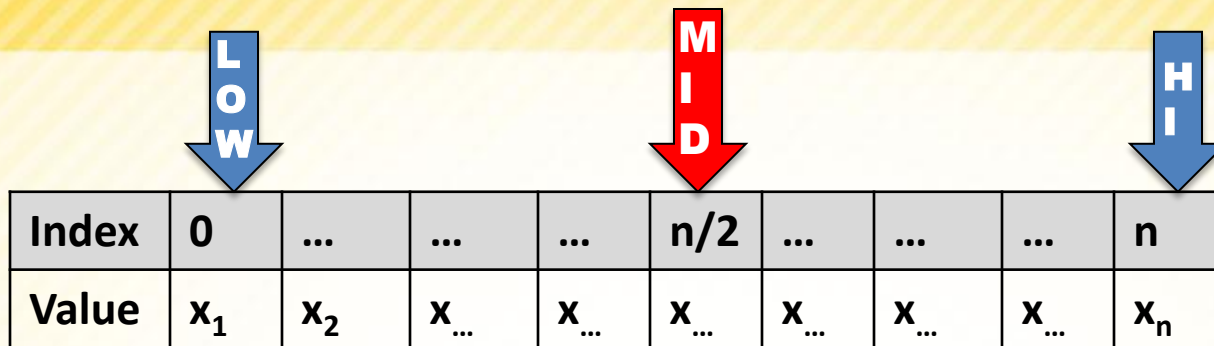
- Searching for 19:



- 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



```
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.

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 $1/2$.

- 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!



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,

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:
 - $\frac{n}{2^k} = 1$, *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:
 - **$\frac{n}{2^k} = 1$, now we just solve for k (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 **$n/2$ 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)**
 - **$n = 2^k$**
 - **$k = \log_2 n$**
- 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:

n	log n
8	3
1024	10
65536	16
1048576	20
33554432	25
1073741824	30





RECURSION

COP 3502

Recursive Binary Search

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

```
int binsearch(int *values, int low, int high, int searchVal) {
    int mid;
    if (!terminating condition){

    }
    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 $low > high$

```
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:

```
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

- 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**



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 $6+6+6+6$



Practice Problem

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

```
int Multiply(int first, int second) {  
    if (( second == 0 ) || ( first = 0))  
        return 0;  
    else  
        return (first + Multiply(first, second - 1));  
}
```

