Recursion

Recursion

- Recursion happens when a function calls itself.

- It is a very useful way to solve certain problems.
- However, each call to itself takes up function stack space (as does any call to a function).
- As a result, the recursion depth is limited (due to limited stack space available to programs).

- Fun analogy:
  - The movie Inception shows a dream within a dream, which itself is within another dream, and so on.
  - So I guess you can call that recursive dreaming ☺

- Now, because a function calls itself, there has to be a stopping condition where the call to itself would have to stop (otherwise the program will run out of stack space and crash).

- If you run the program below, it is bound to crash pretty quickly ☺
- That's because the recursion there does NOT have a stopping condition, so it will recurse indefinitely (not really, because it will crash soon)

```
void MyNonStoppingRecursiveFunction()
{
        MyNonStoppingRecursiveFunction();
}
int main()
{
        MyNonStoppingRecursiveFunction();
}
```

- Following conditions should be satisfied in a recursion:

- There should be a base case which is where the recursion stops (i.e., the function does not call itself).
- The recursive calls to itself must take it closer to the base case / terminating condition.
- In the base case, the function returns a known value and doesn't call itself, thereby stopping the recursion.
- Factorial(3) = 1 * 2 * 3
- Factorial(5) = 1 * 2 * 3 * 4 * 5 = 120

- Lets look at a simple common example: (Assume n is >= 0. If n is < 0, this just returns 1).

```
int Factorial( int n )
{
        if (n <= 1)                    ← Base case or terminating condition.
            return 1;
        else
            return n * Factorial (n – 1);  ← Calling Factorial (n-1) is taking it closer to base case.  n>1
}
```

Calling Factorial (4) will look like:

Factorial (4)

↓

Factorial (3)

↓

Factorial (2)

↓

Factorial (1)

So, each call to the function just says my value is the value of N passed to me, multiplied by whatever the next one down returns…. Unless my value is 1, in which case I wont ask the next one down, but will just return 1, as I am the base case and I know what I need to return, and I don't need to ask anyone else.

Note: In factorial, n == 1 or n== 0 are both base cases.

```
factorial(4)
    factorial(3)
        factorial(2)
            factorial(1)
            return 1
        return 2*1 = 2
    return 3*2 = 6
return 4*6 = 24
```

Big O complexity

- What is the Big O complexity of the factorial function?

Iterative implementation

- Lets take a quick look at the iterative implementation of factorial

```
int Factorial( int n )
{
        int fac = 1;                      // initialize to 1, not 0

        for (int ii = n; ii >= 1; --ii)   // could also make the condition ii >= 2
                fac *= ii;                // Same as fac = fac * ii;

        return fac;
}
```

What is the time complexity of iterative implementation?
What is the space complexity of iterative implementation?

Iterative implementation is more efficient than recursive.
Why?

Fibonacci sequence

- Fibonacci sequence is a well-known Mathematical sequence.

- It is defined as:
  - *a sequence of numbers where :*
    - *first two numbers are 0 and 1*
    - *each subsequent number is the sum of previous two numbers*.

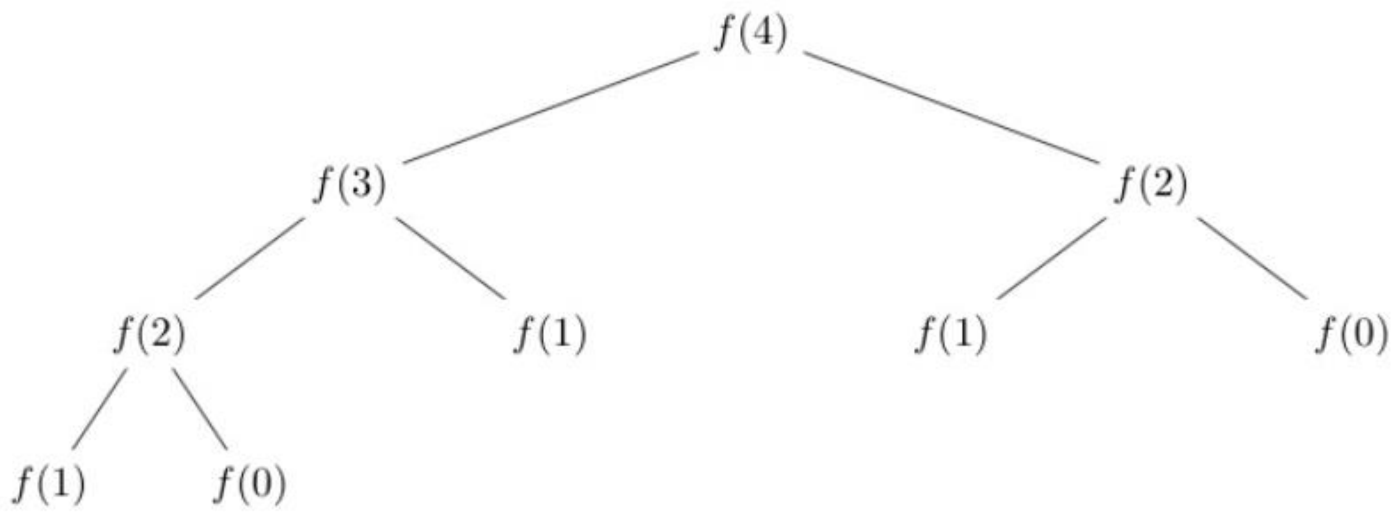
    - In other words, the first two numbers are special, but every number after that is the sum of previous two numbers.

- Here are the first few Fibonacci numbers:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

- Lets look at an implementation:

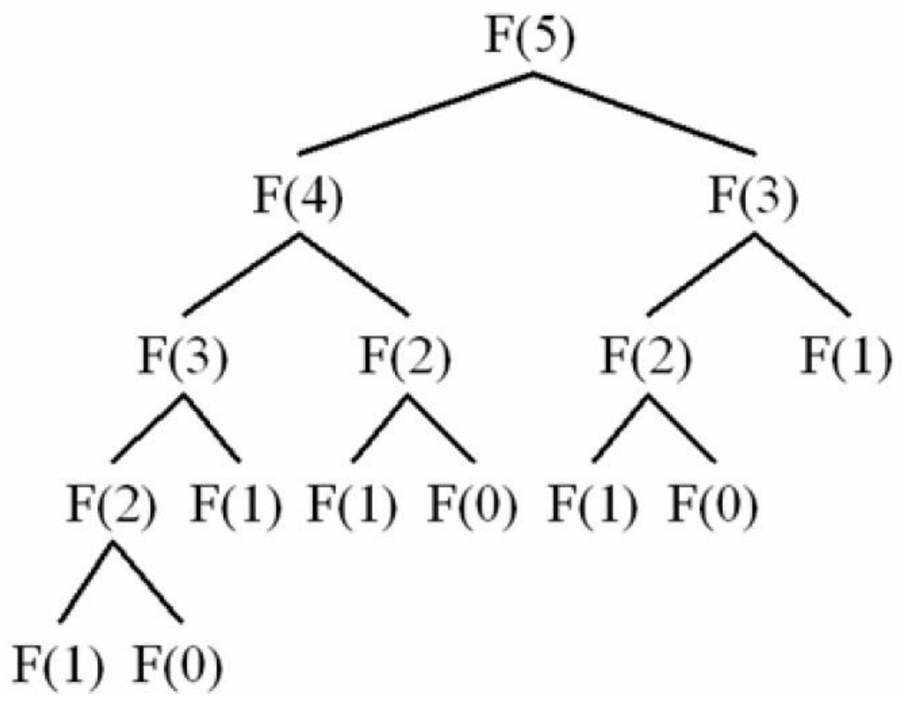
```
// Assumption: n >= 0, i.e., some other code checks and errors if n < 0
int Fibonacci( int n )
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return  Fibonacci (n – 1) + Fibonacci( n – 2 );
}
```

Fibonacci (4) call tree



- When computing Fibonacci(4), we have two calls to Fibonacci(2).

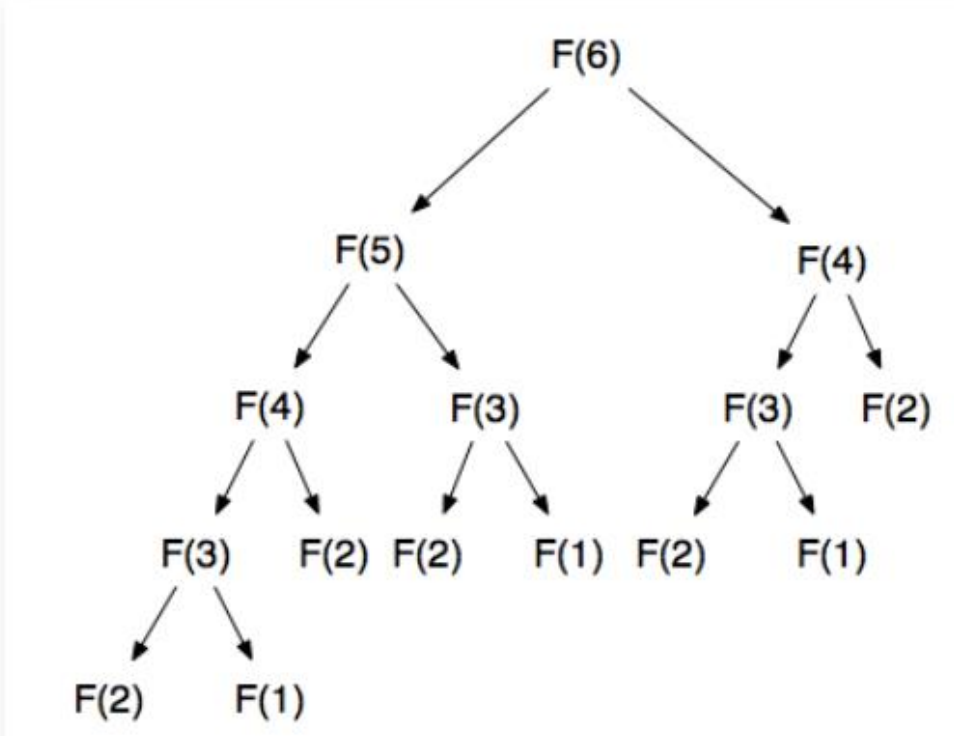
Fibonacci(5) call tree



When computing Fibonacci(5), we have

- Two calls to Fibonacci(3)
- Three calls to Fibonacci(2)

Fibonacci(6) call tree



When computing Fibonacci(6), we have

- Two calls to Fibonacci(4)
- Three calls to Fibonacci(3)
- Five calls to Fibonacci(2)

Big O

- So, as you can see, there are a lot of repeated calls for the same value of N.
  - We will revisit this fact in a few slides.

- Lets look at how many nodes we have in the Fibonacci call tree, we notice the following:
  - Root level has 1 node ($2^{0)}$
  - Level below root has 2 nodes(2^1 then 4(2^2), then 8, and so on.
    - Note: yes, when we get towards the bottom, not all levels will be completely filled with nodes.

- Now, Fibonacci(n) will have an n level call tree.
- So, in general, we can say that a level n tree will have 2^n nodes.

The algorithm we just saw has a time complexity of O(2^n)

- Exponential algorithm ☹… and as we saw before, that is not great.

Big O

- Space complexity:
  - The space here is taken up by the function call stack.
  - The max depth of the function call stack is n.

  - So the space complexity is O (n).

Memoization

- As we saw, there were a lot of repeated calls to values that had already been computed before.
- To prevent this, we can cache the values that are freshly computed, and then read from the cache.

- The code would look something like:
- Assumption: cache in pseudocode below can be an array or dictionary or whatever(some storage), and is assumed to be allocated outside of the function.
  - It is filled up inside the function, as we make the calls to Fibonacci function.

```
// Assumption: n >= 0, i.e., some other code checks and errors if n < 0
int Fibonacci( int n )
{
        if (n == 0)
                return 0;
        else if (n == 1)
                return 1;
        else
        {
                if (n in cache)
                        return cache[n];
                cache[n] =  Fibonacci (n – 1) + Fibonacci( n – 2 );
                return cache [n];
        }
}
```

Memoization

Or you could rewrite it with a minor change:

```
// Assumption: n >= 0, i.e., some other code checks and errors if n < 0
int Fibonacci( int n )
{
        if (n == 0)
                return 0;
        else if (n == 1)
                return 1;
        else
        {
                if  NOT (n in cache)
                                cache[n] =  Fibonacci (n – 1) + Fibonacci( n – 2 );

                return cache [n];
        }
}
```

Memoization drastically reduces the number of recursive calls to Fibonacci.

Time complexity

We now have to compute each node only once, because we cache the values.

So now we have a Big O of O(N)

Space complexity

We cached all the values, so the Big O here is O(N).

LAB

1. Code up a *recursive* solution to Fibonacci with memoization.

2. Code up an *iterative* solution to Fibonacci.
    1. What is the time and space complexity of the iterative solution?
    2. Which implementation of Fibonacci is most efficient for: (1) time, and for (2) space.
        1. Recursive
        2. Recursive with memorization
        3. Iterative

LAB code in IDE

- First lets look at code for this lab in the IDE, then we can look at the bullets below

- So, we have, in order of increasing speed, the following
  - Lets fill in the blanks:

  - Recursive Fibonacci with **no** memorization. **Slowest**
    - Time:
    - Space:

  - Recursive Fibonacci **with** memorization
    - Time:
    - Space:

  - Iterative Fibonacci
    - Time:
    - Space:

  - Recursive Fibonacci **with** memorization, but calls made repeatedly on the **same** object
    - Lets talk about the calls made **AFTER the first call** (when we have stored the Fibonacci values)
      - Time:
      - Space: