Topic 7.1: Recursion

Learning Goals (Week 8):

- Create and implement recursive solutions to simple problems such as simple mathematical calculations and list traversals.
- Write a recursive solution to a problem with and without a helper function.
- Identify and explain the base case and recursive step components of a recursive algorithm.

Iterative vs Recursive

- All the previous examples are ones we could have solved with a loop pretty easily
  - this means, at this point, we haven't seen a reason to solve a problem recursively instead of doing it iteratively
- When a solution to a problem would require **multiple recursive calls**, the iterative solution is CONSIDERABLY less elegant than a recursive one
  - often referred to as a **naturally recursive problem**

Binary Search

- Remember our friend binary search? It’s actually naturally recursive!
- Base Case(s):
  - hi < lo: we didn’t it

    ```
    if (hi < lo) { return -1 }
    ```
  - array[mid] == target: we found it, return the index we found it at

    ```
    else if(array[mid] == target) { return mid; }
    ```
- Recursive Case(s):
  - array[mid] > target: recursive call with mid to hi end points

    ```
    if (array[mid] > target) { return binaryRec(lo, mid, target, arr); }
    ```
  - array[mid] < target: recursive call with lo to mid end points

    ```
    else if(array[mid] < target) { return binaryRec(mid+1, hi, target, arr); }
    ```

Another: Towers of Hanoi

- Objective:
  - Move the disks to the right (or center) peg
- Rules:
  - Only one can be moved at a time
  - Only the top one from one peg can be moved
  - No disk can ever be placed on a smaller one

Image: woodenpuzzle.com

Recursive Solution

- Surprisingly easy
- To move disks **1 to n**: from peg A to peg B (using peg C):
- Base Case
  - If n=1: just move disk 1
- Recursive Case
  - Move disks 1 to (n-1) from A to C (using B temporarily)
  - Move disk n from A to B
  - Move disks 1 to (n-1) from C to B (using A temporarily)

Image: woodenpuzzle.com

Recursive Solution

means move n discs from A to B using C

```
public static void solveHanoi(int n, String A, String B, String C)
{
    if(n==1) System.out.println("move 1 from "+A+" to "+B);
    else {
        solveHanoi(n-1,A,C,B);
        System.out.println("move "+n+" from "+A+" to "+B);
        solveHanoi(n-1,C,B,A);
    }
}
```

Reminder

- Every instance of the method must be independent
- With its own complete set of variables (received as parameters)
  - you need to receive parameters:
    - that’s how you can send information to each recursive call!

- Let’s look at the power of recursion in an example
- and then an example downfall

Subsets

- Choosing all possible subsets of k things chosen from n things?
- Example: choose 3 things from 5 things

- Recursion:
  - Choose k-1 things from n-1 things
  - Tack on the first thing to all of these
  - Choose k things from n-1 things

1	**2**	**3**	4	5
1	**2**	3	**4**	5
1	**2**	3	4	**5**
1	2	**3**	**4**	5
1	2	**3**	4	**5**
1	2	3	**4**	**5**
1	**2**	**3**	**4**	5
1	**2**	**3**	4	**5**
1	**2**	3	**4**	**5**
1	2	**3**	**4**	**5**

Subsets

- Base Case?
  - Can be tricky:
  - In the top recursion (option 1), we'd go from:
    - 3 of 5 ⇒ 2 of 4 ⇒ 1 of 3 ⇒ 0 of 2.
      - Stop there! There's one solution to take 0 of anything.
  - But in the bottom recursion (option 2), we'd go from:
    - 3 of 5 ⇒ 3 of 4 ⇒ 3 of 3 ⇒ 3 of 2.
    - Stop! 3 of 2 is clearly impossible. No solutions! Quit!

- This time, we'll simply print the results as we get them
- Programming it can be a bit tricky for two reasons:
- The main problem is not directly recursive any more. We have to pass the things we've already chosen as an extra parameter

Subsets

- Final Note on the trickery of this problem:
  - When we shrink the list, we have to make sure that the original list is NOT affected!
  - DON'T DESTROY YOUR PARAMETERS!
  - Objects can easily be changed! (Integer and String are OK, because immutable)

Let’s look at the code in Subset.java

Tricky: Permutations

Great for permutations, subsets, game strategies, mazes, space filling, fractals, etc. etc.

- Example: Generate all possible permutations of an ArrayList
  - producing a list of ArrayLists –
  - stored as an ArrayList
  - That's a 2D ArrayList!

1	**2**	**3**	**4**
1	**2**	**4**	**3**
1	**3**	**2**	**4**
1	**3**	**4**	**2**
1	**4**	**3**	**2**
1	4	2	3
2	1	3	4
2	1	4	3
…	…	…	…
2	1	3	4
3	…	…	…
4	…	…	…

Tricky: Permutations

Great for permutations, subsets, game strategies, mazes, space filling, fractals, etc. etc.

- Example: Generate all possible permutations of an ArrayList
  - producing a list of ArrayLists –
  - stored as an ArrayList
  - That's a 2D ArrayList!

1	2	3	4	
1	2	4	3	
1	3	2	4	234 Permutations
1	3	4	2	
1	4	3	2	
1	4	2	3	
2	1	3	4	
2	1	4	3	134 Permutations
…	…	…	…	
2	1	3	4	
3	…	…	…	124 Permutations
4	…	…	…	123 Permutations

Tricky: Permutations

It may seem daunting but:

Recursive Case:

- for each element in the list
  - choose that element to be first
  - find all the sub-permutations of the rest
  - add that first element to all the sub-permutations
  - add them all to the list of solutions

Base Case:

- The size of the list to permute through is smaller each time so eventually it must be:
  - ONE!
  - That's trivial and requires no recursion (there’s only 1 permutation of size 1)
  - You have to generate a list of 1 solution, and that 1 solution is a list of the 1 element.
  - It still must be a “2D ArrayList”

When this can go wrong:

Fibonacci numbers are usually defined recursively:

$$fib(0) = fib(1) = 1$$

$$fib(n) = fib(n-2)+fib(n-1) \text{ [for } n \geq 2]$$

When this can go wrong:

Fibonacci numbers are usually defined recursively:

```
fib(0) = fib(1) = 1

fib(n) = fib(n-2)+fib(n-1) [for n≥2]
```

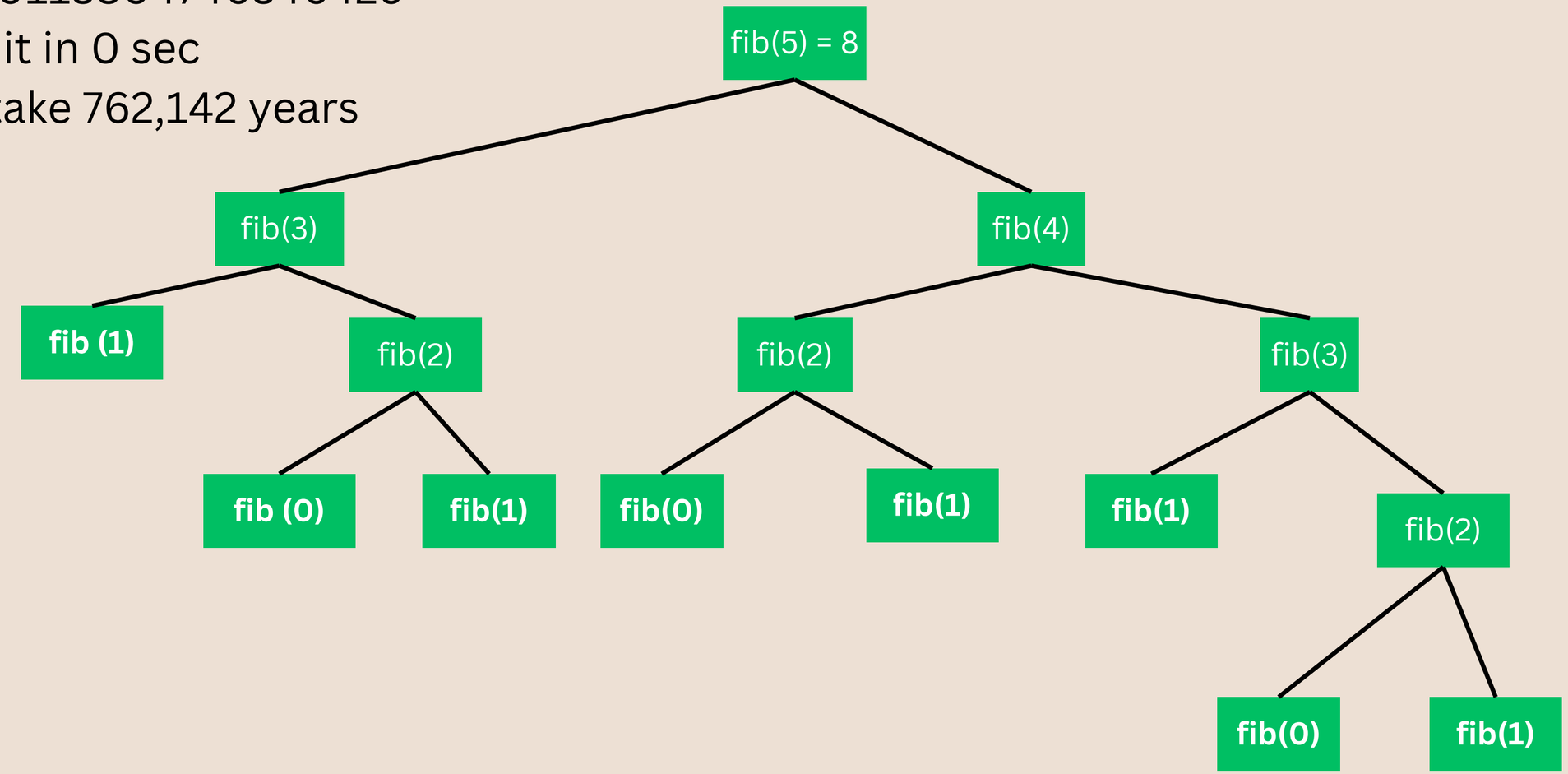
CRAZY EASY TO DO RECURSIVELY

```
public static long fibR(int n) {
   if(n<=1)
     return 1;
   else
     return fibR(n-2)+fibR(n-1);
}
```

Let’s see the timing on this…(checkout Fib.java)

When this can go wrong:

- For fib(91)=7540113804746346429
- A loop will find it in 0 sec
- Recursion will take 762,142 years (roughly)



Algorithmic Complexity Takeaways:

- The simplicity or complexity of the code is not an indication of the speed of the algorithm!

- You have to understand and analyze the number of actions/steps/operations that an algorithm will do

- This can require significant amounts of mathematical analysis at times

- We'll do a little bit in the next part of COMP 1020
    - For the real story, take COMP 2080

Recursion Solving Takeaways:

- Building a recursive method involves answering two questions:
    - How can I solve a big problem by solving smaller instances of the same problem?
    - How can I stop the recursion with a simple/easy/base case, for which I know the answer?