2️⃣

002 - Recursion

Run-time Stack:

- A run-time stack, also known as a runtime call stack or just a stack, is a specialized data structure used by computer programs during their execution.
- The stack is a last-in, first-out (LIFO) data structure that is used to keep track of function calls, local variables, and other temporary data that is used during the execution of a program.
- Think of the run-time stack as a stack of plates. With each function call, a new "plate" is placed onto the stacks. local variables and parameters are placed onto this plate.
- Variables from the calling function are no longer accessible (because they are on the plate below).
- When a function terminates, the local variables and parameters are removed from the stack. Control is then given back to the calling function.
- Your programming language often manages the call stack, which exists outside of any specific function. This call stack tracks the ordering of the different function invocations, so the **last function to enter the call stack is the first function to exit the call stack**.

Q) What is Recursion?

→ Recursion is a programming technique in which a function calls itself in order to solve a problem or perform a task. In other words, recursion is a process in which a function or a method is called within itself repeatedly until a base condition is met.

Q) Why do we need Recursion?

→ It helps us in solving bigger and complex problems in a simpler way. You can convert recursion solutions into iteration and vice versa. Point to note is that the space complexity is not constant.

STEPS TO WRITE RECURSIVE FUNCTIONS

1. State the base case. What argument values will lead to a solution so simple that you can simply return the value.
2. State the recursive case. If you are given something other than the base case, how can you use the function itself to get to the base case.
3. Recursion is about letting the problem solve itself. The solution to the current problem consists of taking a small step and restarting the prooblem.

BASE CASE AND RECURSIVE STEP

Recursion has two fundamental aspects: the base case and the recursive step.

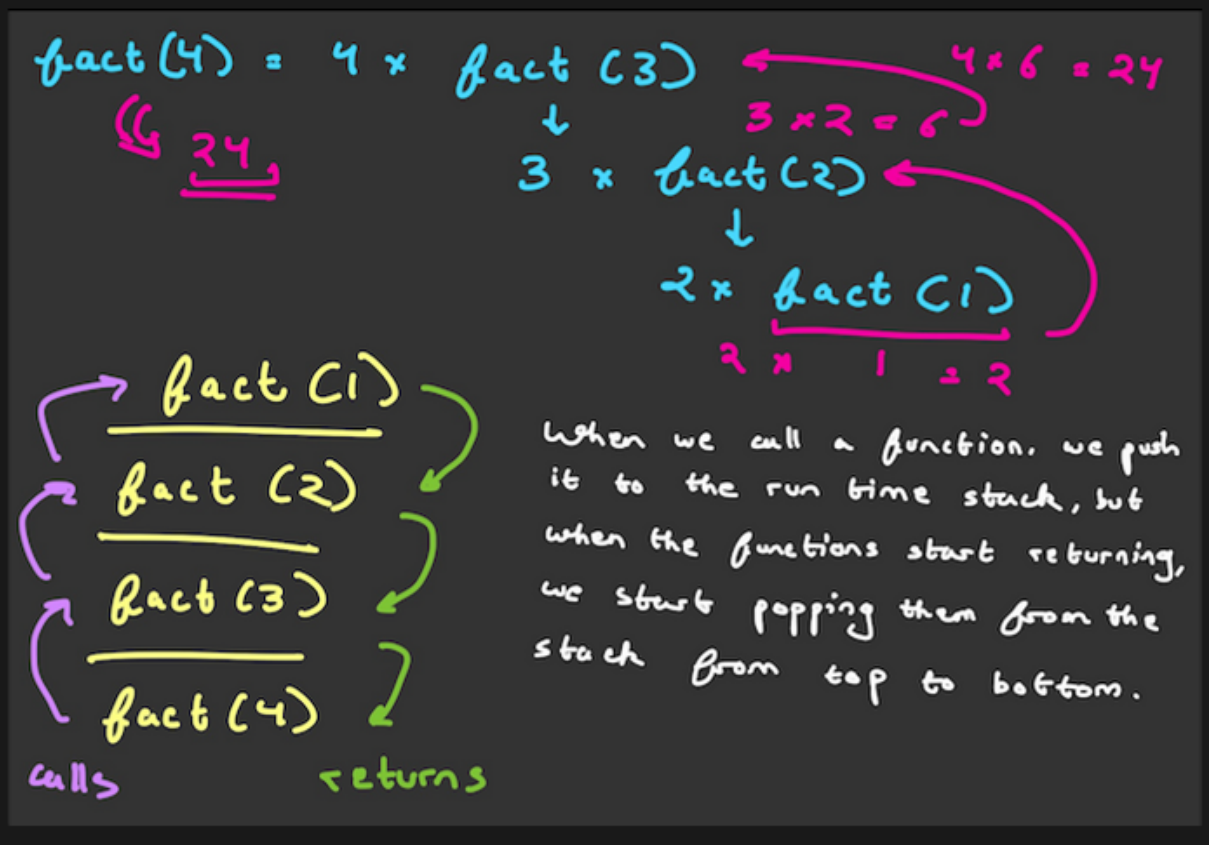
1. The base caseis like the counting variable in iteration as it decides whether the function will recurse. Without a base case, it would be like an infinite loop.
2. This will result in a "stack overflow" error as the stack memory would be full at some point because of the function calls in it and the computer will crash.
3. Recursive step is the portion of the function that moves us one step closer to the solution.
4. As we dont have incrementing/decrementing counting variables in case of recursion, we have to use arguments to the functions.
5. If we're counting down to 0, for example, our base case would be the function call that receives `0` as an argument. We might design a recursive step that takes the argument passed in, decrements it by one, and calls the function again with the decremented argument. In this way, we would be moving towards `0` as our base case.
6. Analyzing the Big O runtime of a recursive function is very similar to analyzing an iterative function. Substitute iterations of a loop with recursive calls. For example, if we loop through once for each element printing the value, we have a `O(N)` or linear runtime. Similarly, if we have a single recursive call for each element in the original function call, we have a `O(N)` or linear runtime.

> 💡 **Execution Context: An execution context refers to the state of a particular function call at a given point in time. It includes information such as the function's local variables, parameters, return address, and any other relevant data associated with that specific invocation of the function.**

FACTORIAL FUNCTION

```
def fact(n):
    if n == 1:
        return 1
    return n * fact(n - 1)
print(fact(4))
```

- Base Case: **1! = 1**
- Working: 5! = 5 * 4 * 3 * 2 * 1
- Therefore, 5! = 5 * 4! ⇒ n! = n * (n - 1)!



TIME COMPLEXITY ANALYSIS:

```
def fact(n):
    if n == 1: # 1
        return 1 # 1
    return n * fact(n - 1) # 2 + T(n - 1)
print(fact(4))
```

- Let 'n' represent the number that we are calculating the factorial of.
- Let T(n) represent the number of operations it takes to find n.
- T(0) and T(1) = 2
- T(n) = 2 + 2 + T(n - 1)
- T(n - 1) = 4 + T(n - 2) and so on....
- T(n) = 4 + 4 + 4 + ... 4 + 2
- T(n) = 4(n - 1) + 2 ⇒ 4n - 2
- Time Complexity: O(n)

DRAWBACKS OF RECURSION

- Recursion isn't the best way of writing code. If you are writing code recursively, you are probably putting on extra overhead. For example the factorial function could be easily written using a simple for loop. If the code is straight forward an iterative solution is likely faster. In some cases, recursive solutions are much slower.
- **Use recursion iff:**
  - The problem is naturally recursive
  - A relatively straight forward iterative solution is not available
- The reason is that recursion makes use of the run time stack. If you don't write code properly, your program can easily run out of stack space. You can also run out of stack space if you have a lot of data. You may wish to write it another way that doesn't involve recursion so that this doesn't happen.

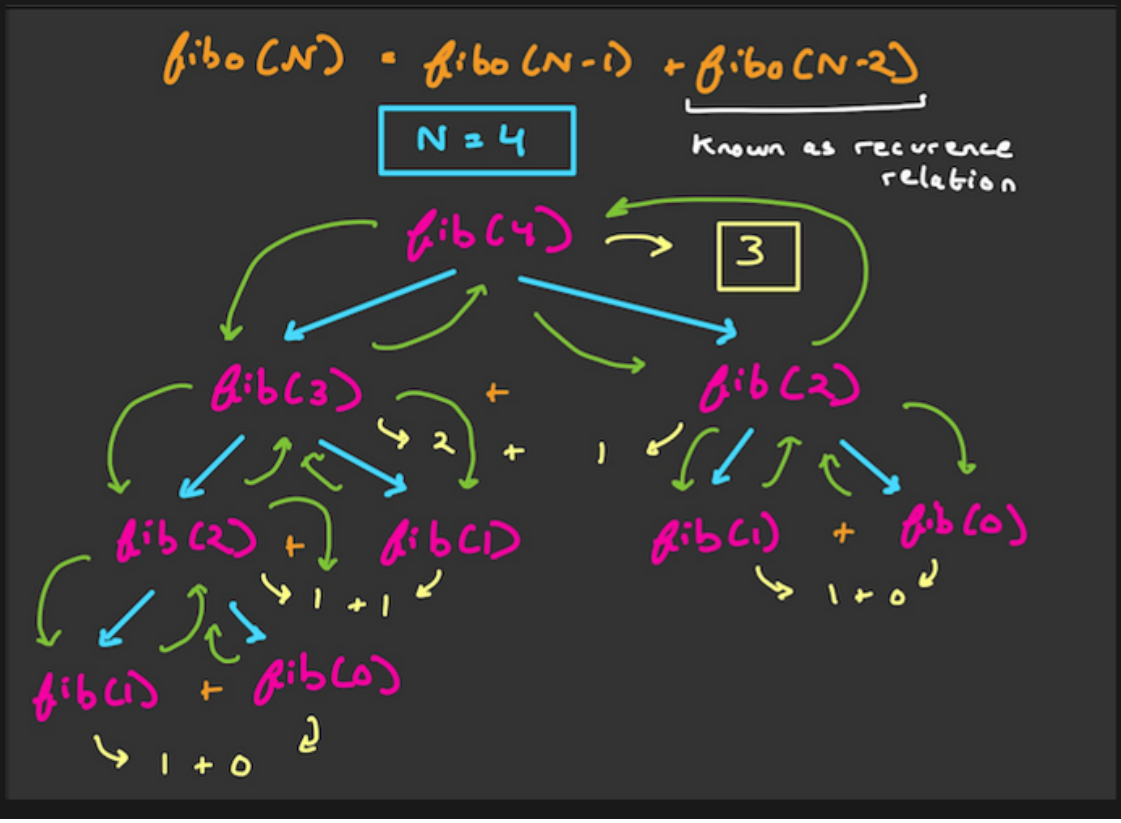
UNDERSTANDING AND APPROACHING A RECURSION PROBLEM

- Identify if you can break down the problem into smaller problems.
- Write the recusive relation if needed.
- Draw the recursive tree.
- **About the tree:**
  - See the flow of functions, how are they getting in stack.
  - Identify and focus on left tree calls and right tree calls.
  - Draw the tree and pointers again and again and use a debugger to see the steps.
  - See how the values are returned from each step. See where the function call will come out of.

QUESTIONS

Q) Find nth fibonacci number

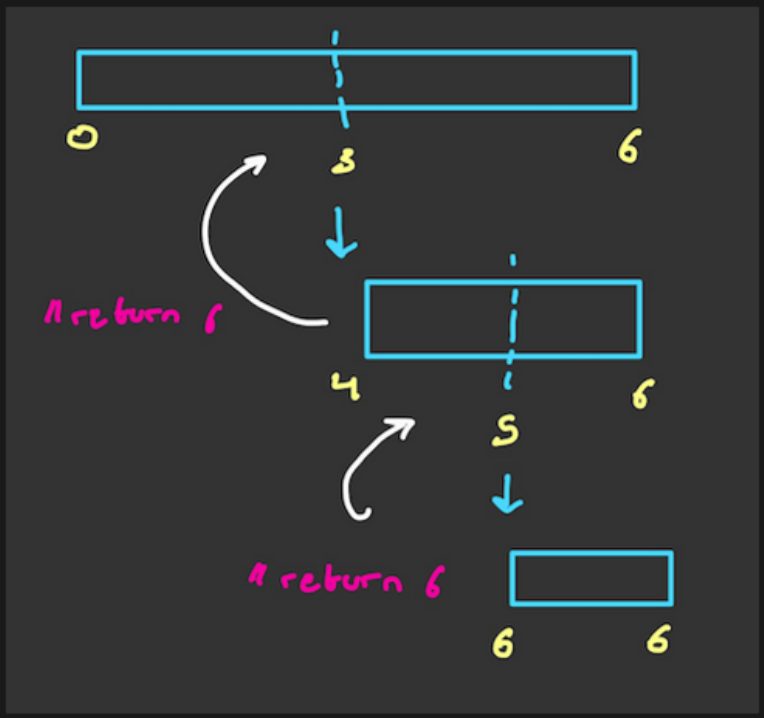
```python
# Find nth fibonacci number
def fibo(n):
    if (n < 2):
        return n
    return fibo(n - 1) + fibo(n - 2)
print(fibo(4)) # Output: 3
```



Q) Recursive Binary Search

```
def recursiveBinarySearch(arr, target, start, end):
    if (start > end):
        return -1
    middle = start + int((end - start) / 2)
    if (arr[middle] == target):
        return middle
    if (target < arr[middle]):
        return recursiveBinarySearch(arr, target, start, middle - 1)
    elif (target > arr[middle]):
        return recursiveBinarySearch(arr, target, middle + 1, end)

print(recursiveBinarySearch([1, 2, 3, 4, 5], 2, 0, 4))
```



Notes and Tips:

- **Types of recurrence relation:**
  - Linear Recurrence Relation: Search set reduces search set linearly
  - Divide and Conquer: Search set gets reduced by a factor
- Variables:
  - Variables that are supposed to be used in the future of the recursive function (variables that are used in the next function) have to be put as the arguments.
  - Variables that are specific to a function call have to be in the function body.
- Always return if the function has a return type.