COMPETITIVE PROGRAMMING
COVID CODING PROGRAM

MENTOR: MR. RAHUL LATHER
sachinlather49@gmail.com

07 April 2020, Day-2

Content To Be Discussed Today!

- Recursion
    - What is Recursion?
    - Why use Recursion?
    - Practice Problems
- Divide & Conquer
    - What is Divide & Conquer?
    - Structure of Divide & Conquer.
    - Practice Problems

What is Recursion?

Recursion is a wonderful programming tool. It provides a simple, powerful way of approaching a variety of problems.

In order to say exactly what recursion is, we first have to answer “What is recursion?”.

Basically, a function is said to be recursive if it calls itself. Below is pseudocode for a recursive function that prints the phrase “Hello World” a total of count times:

You may be thinking this is not terribly exciting, but this function demonstrates some key considerations in designing a recursive algorithm:

```
function HelloWorld(count) {

  if (count < 1) return

  print("Hello World!")

  HelloWorld(count - 1)
}
```

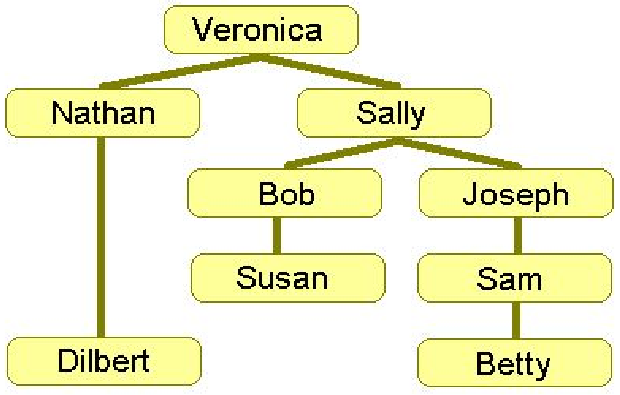
1. **It handles a simple "base case" without using recursion.** In this example, the base case is "HelloWorld(0)"; if the function is asked to print zero times then it returns without spawning any more "HelloWorld"s.
2. **Avoids Cycles (Infinite Calls).** Imagine if "HelloWorld(10)" called "HelloWorld(10)" which called "HelloWorld(10)." You'd end up with an infinite cycle of calls, and this usually would result in a "stack overflow" error while running. In many recursive programs, you can avoid cycles by having each function call be for a problem that is somehow smaller or simpler than the original problem.

Why use Recursion?

The problem we illustrated above is simple, and the solution we wrote works, but we probably would have been better off just using a loop instead of bothering with recursion.

Recursion can be applied to pretty much any problem, but there are certain scenarios for which you'll find it's particularly helpful.

Scenario #1: Hierarchies, Networks, or Graphs



Now suppose we are given the task of writing a function that looks like "countEmployeesUnder(employeeName)". This function is intended to tell us how many employees report (directly or indirectly) to the person named by **employeeName**. For example, suppose we're calling "**countEmployeesUnder('Sally')**" to find out how many employees report to Sally.

To start off, it's simple enough to count how many people work directly under her. To do this, we loop through each database record, and for each employee whose manager is Sally we increment a counter variable. Implementing this approach, our function would return a count of 2: Bob and Joseph. This is a start, but we also want to count people like Susan or Betty who are lower in the hierarchy but report to Sally indirectly. This is awkward because when looking at the individual record for Susan, for example, it's not immediately clear how Sally is involved.

A good solution, as you might have guessed, is to use recursion.

Remember that each time you make a recursive call, you get a new copy of all your local variables. This means that there will be a separate copy of counter for each call.

```
{
  declare variable counter
  counter = 0
  for each person in employeeDatabase {
    if (person.manager == employeeName) {
      counter = counter + 1
      counter = counter + countEmployeesUnder(person.name)
    }
  }
  return counter
}
```

Scenario #2: Explicit Recursive Relationships

You may have heard of the Fibonacci number sequence. This sequence looks like this: 0, 1, 1, 2, 3, 5, 8, 13... After the first two values, each successive number is the sum of the previous two numbers. We can define the Fibonacci sequence like this:

```
function fib(n) {
  if (n < 1) return 0
  if (n == 1) return 1
  return fib(n - 2) + fib(n - 1)
}
```

Scenario #3: Multiple Related Decisions

When our program only has to make one decision, our approach can be fairly simple. We loop through each of the options for our decision, evaluate each one, and pick the best. If we have two decisions, we can have nest one loop inside the other so that we try each possible combination of decisions. However, if we have a lot of decisions to make (possibly we don't even know how many decisions we'll need to make), this approach doesn't hold up.

For example, one very common use of recursion is to solve mazes. In a good maze we have multiple options for which way to go. Each of those options may lead to new choices, which in turn may lead to new choices as the path continues to branch. In the process of getting from start to finish, we may have to make a number of related decisions on which way to turn. Instead of making all of these decisions at once, we can instead make just one decision. For each option we try for the first decision, we then make a recursive call to try each possibility for all of the remaining decisions.

How memory is allocated to different function calls in recursion?

Practice Problems

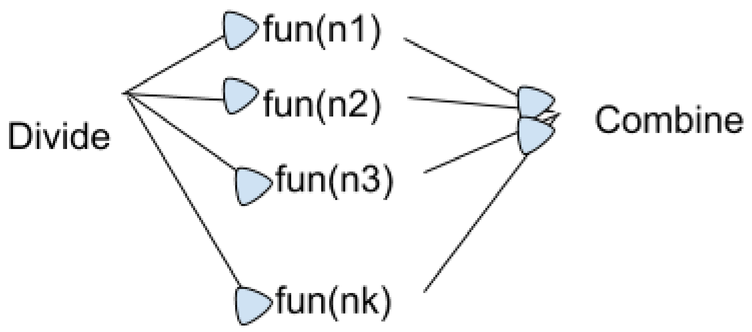
1. Recursive program to find all Indices of a Number.
2. Return Keypad Code

What is Divide & Conquer?

In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.

This technique can be divided into the following three parts:

1. **Divide:** This involves dividing the problem into some sub problem.
2. **Conquer:** Sub problem by calling recursively until sub problem solved.
3. **Combine:** The Sub problem Solved so that we will get find problem solution.



Practice Problems:

3. Frequency of an integer in the given array using Divide and Conquer

4. Count Inversions