# Dynamic Programming

## Problem (1 hour)

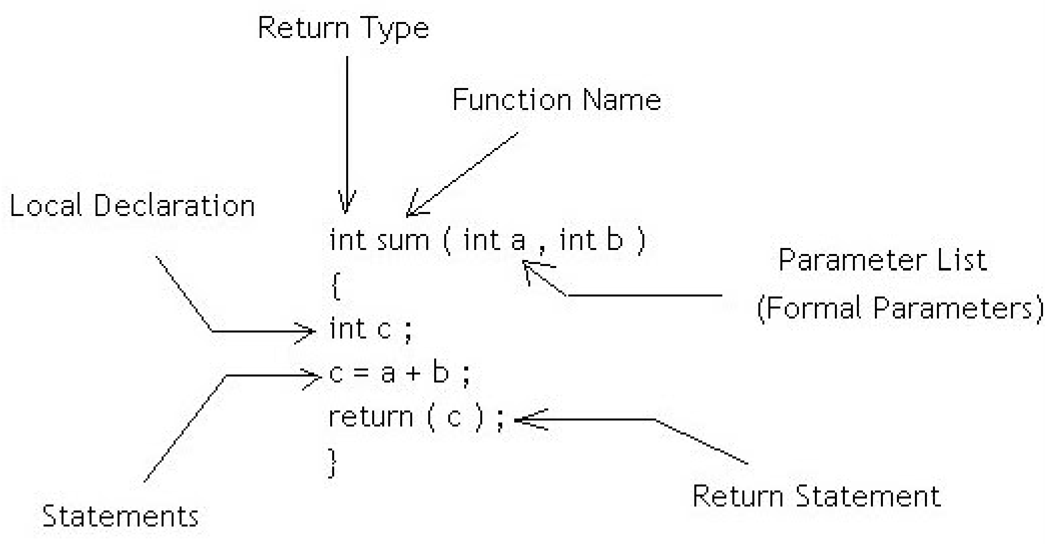
- Question - 1: Staircase Problem: https://www.geeksforgeeks.org/count-ways-reach-nth-stair-using-step-1-2-3/

```
Shubhendu wants to reach Nth stair. But he can only take 1 step, 2 steps or 3 steps at a time. So
in how many ways can Shubhendhu reach the Nth stair?

*For example, Subhendhu wants to reach to 4th stair and he knows only how to take 1 step, 2 step
and 3 step. The number of ways he can reach the 4th stair is 7*

1. 1+1+1+1
2. 1+1+2
3. 1+2+1
4. 2+1+1
5. 1+3
6. 3+1
7. 2+2
```

Solution: Recursive Method

- What are Functions ?

  Similar to Mathematical Functions. They take input and do some processing and returns an output.



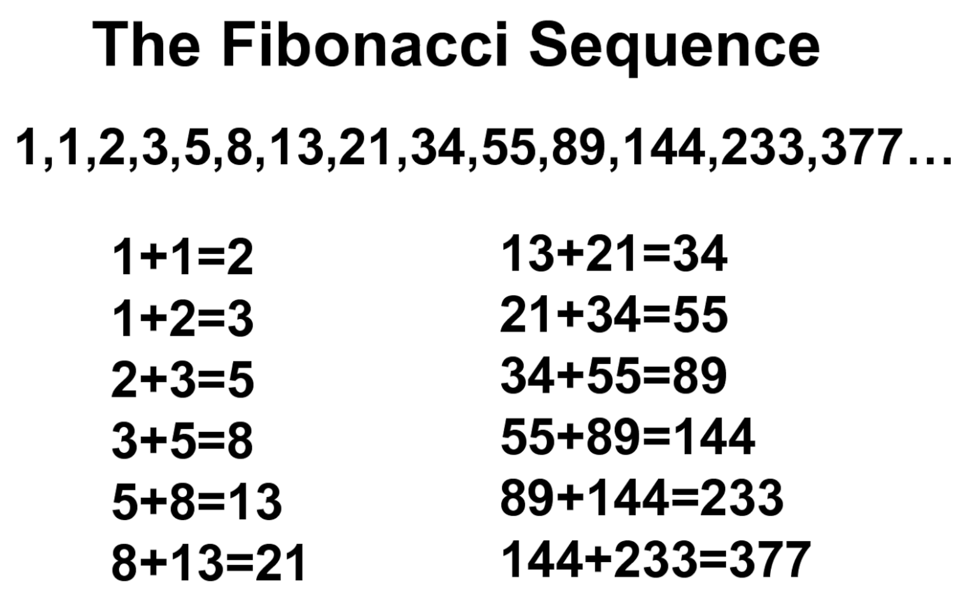
```
int sum ( int a , int b )
{
int c ;
c = a + b ;
return ( c ) ;
}
```

- What is Recursion ?

  Subhendhu is sitting at the last row and I ask him "what row number are you sitting in?" But as we all know he is a lazy guy but is a bit intelligent. So he thinks that whatever is the row number of the guy sitting in front of him has and if he adds 1 to that that number then that will be his row number. Basically his `rowNumber(N) = rowNumber(N-1) + 1` So he asks the guy sitting in front of him "What is his row number?"

  Then the person sitting in front of Shubhendhu asked the same thing to the the person sitting in front of him. This happens till the person in second row asks the person sitting in first row "What row number is he sitting in?". The first guy knows that he is sitting in the First Row. Therefore, he "returns 1" and then the second row guy will "return `1+1` ". And Third row guy will "return `2+1` ". And so on.....

- Fibonacci Series (Yet another Sequence!!!!)



The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

| | |
|---|---|
| 1+1=2 | 13+21=34 |
| 1+2=3 | 21+34=55 |
| 2+3=5 | 34+55=89 |
| 3+5=8 | 55+89=144 |
| 5+8=13 | 89+144=233 |
| 8+13=21 | 144+233=377 |

- Let us try to code the Fibonacci Series using recursion!!!!! Way simpler!
- What is the complexity ? Think of better solution !!

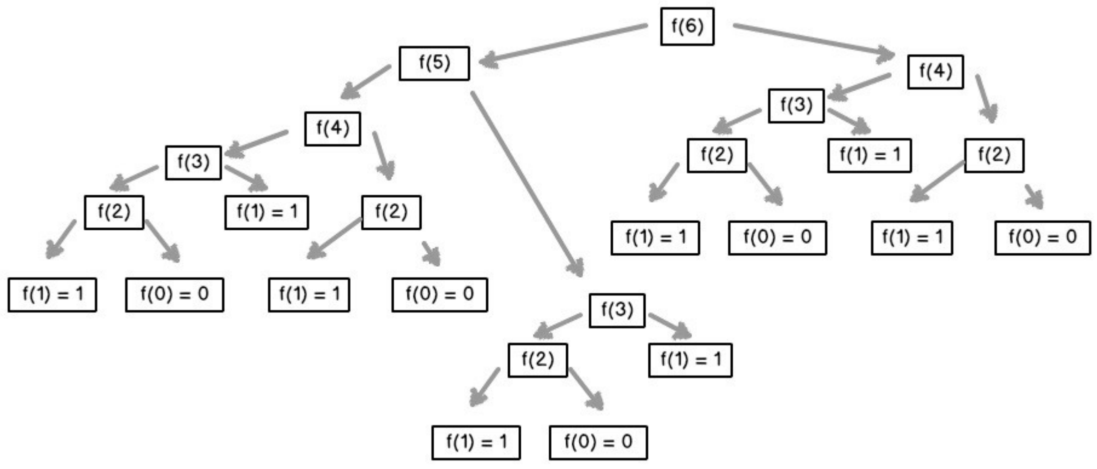
  The Time complexity of this solution is `O(2^n)`

  WHICH IS WAY TOO MUCH!!!!!!!!!!!!!!!!!!!

## Better Solution (Building the intuition) (30 minutes)

- O(n) Solution:

  Let us rethink what we are doing



**We can clearly see that we are doing repetitive tasks. See that we are recalculating f(4), f(3), f(2), and so on...**

So what we can do when we see a repetitive tasks happening? what does your intuition say about this?

*Answer: Store the results once you have calculated so that you don't have to calculate the same thing again and again, you can just go and fetch the value at the index where you've stored the result.*

Idea:

We will form a table (array) called `fib` which will store the Fibonacci Numbers i.e. i$^{th}$ Fibonacci Number will be stored at `fib[i]` . For example, the 4th Fibonacci number i.e. 5 will be stored at `fib[4]` .

The array will look like `fib[]= {1,1,2,3,5,8,13,21,...}`

So we'll loop till the Nth Fibonacci number so that we can return `fib[N]`

**Let's code this solution**

We will use Arrays (Data Structure) to store the results.

So, Now Time Complexity will be **O(n) (why?) and Space Complexity will be O(n)**

Is their a O(1) solution?

**Another approach:**

`Fn = {[(√5 + 1)/2] ^ n} / √5`

- Space vs Time trade off!

  So what we have done here is we traded time with space. We have stored the results of one thing so that we don't have to calculate again and again. In this case we have saved computation time but it requires memory to store these values!!!!

- This is DP!!

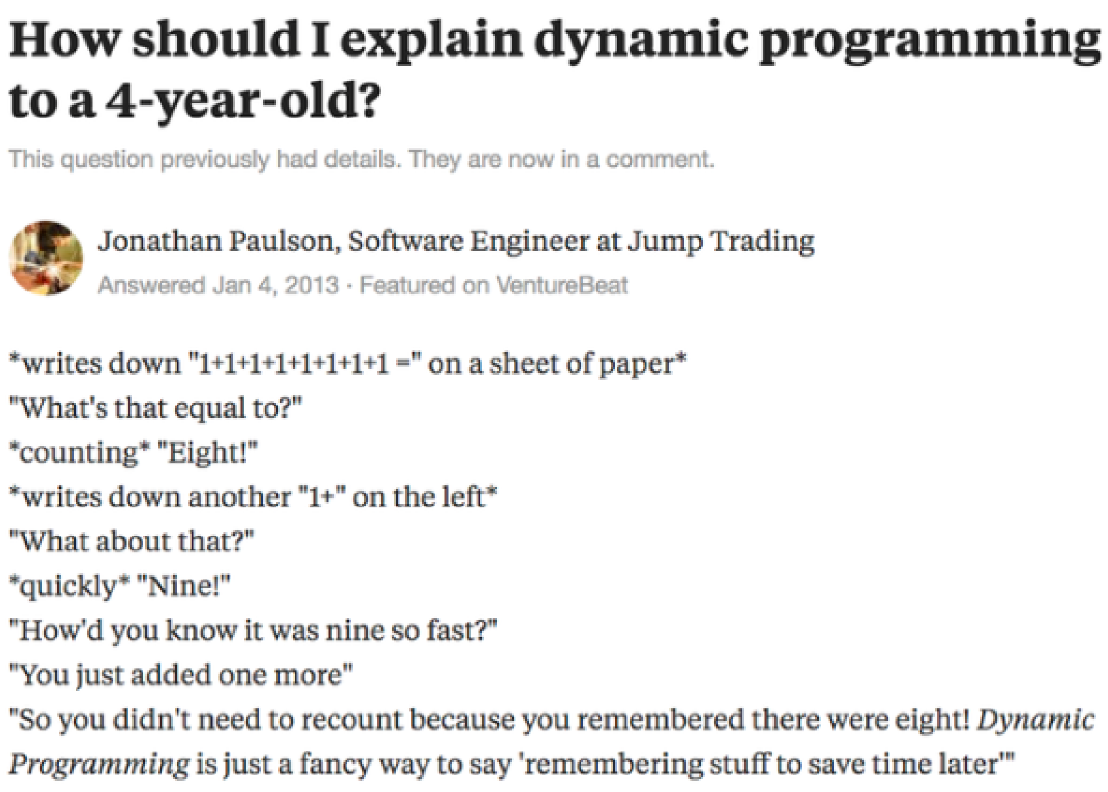
  What we just did was DP!!!!

***Hold your horses, because you are going to go through a lot of information AKA a lot of theory***

## What is DP (30 minutes)

- What is DP = (recursion+common sense)

  So to understand what DP is let's start with this image:



# How should I explain dynamic programming to a 4-year-old?

This question previously had details. They are now in a comment.

Jonathan Paulson, Software Engineer at Jump Trading
Answered Jan 4, 2013 · Featured on VentureBeat

*writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper*
"What's that equal to?"
*counting* "Eight!"
*writes down another "1+" on the left*
"What about that?"
*quickly* "Nine!"
"How'd you know it was nine so fast?"
"You just added one more"
"So you didn't need to recount because you remembered there were eight! *Dynamic Programming* is just a fancy way to say 'remembering stuff to save time later'"

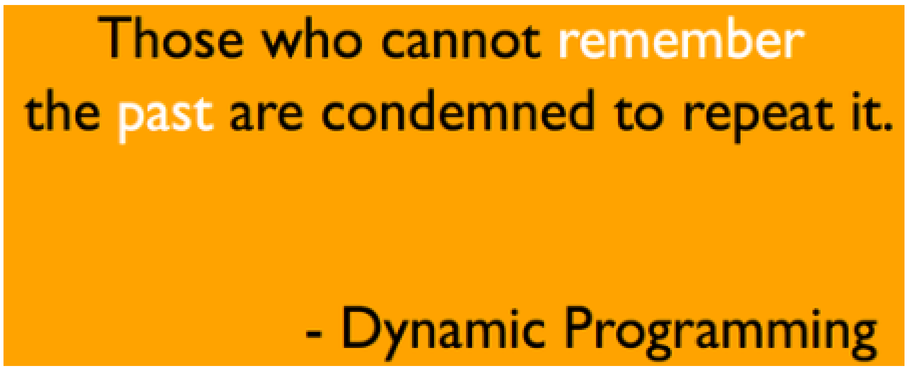
So DP is nothing just "Remembering the Past", why would anyone want to redo (re-calculate) the same thing they just did. (Highly Intuitive, right?)

No, it's not just that!!!

There are basically two things in DP:

1. The given problem should be able to be broken down into smaller sub-problems. In other words, the solution to each problem will depend on the solution of its smaller sub-problems and so on... AKA recursion
2. Storing the solution of each sub-problem as and when you solve it so that you don't have to redo it

- Why DP?



Those who cannot remember the past are condemned to repeat it.

- Dynamic Programming

- Two Properties of DP: Optimal Substructure and Overlapping Subproblems

  There are mainly two properties of DP:

  1. Overlapping Subproblem

     A problem has overlapping subproblems if finding its solution involves solving the *same* subproblem multiple times. Dynamic Programming is mainly used when solutions of the same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don’t have to be recomputed again. Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again.

     Example - Fibonacci Sequence

2. Optimal Substructure

   The term optimal substructure has two components — optimal and substructure. Optimal means best or most favourable, and a substructure simply means a subproblem of the main problem. A problem is said to have an optimal substructure if an optimal solution to the main problem can be constructed efficiently from optimal solutions of its subproblems.

   Example - Suppose we have a network of roads and we are tasked to go from City A to City B by taking the shortest path. And if there is a city between A and B, say C. The optimal solution i.e. shortest path from A to B depends on the shortest path from A to C and C to B then this is nothing but optimal substructure property.

   In other words, if the optimal solution to our main problem (the shortest path from A to B) is composed of optimal solutions of smaller subproblems such as the shortest paths between two intermediate cities. Then, this problem is said to have an optimal structure

- Two ways to conquer DP problems: Top-Down vs Bottom-Up

  There are two approaches to perform DP )in short remembering the old stuff):

  1. Memoization (Top-Down)

     Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive.

     *"Storing On-Demand"*

     It starts from top and perform action as and when required (i.e. when the result is not there already in the table)

  2. Tabulation (Bottom-up)

     Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem.

     *"Storing everything"*

     Start by computing the result for the smallest subproblem (base case). Using the subproblem result, solve another subproblem and finally solve the whole problem. Another way of understanding this would be: Try solving the sub-problems first and use their solutions to build on and arrive at solutions to bigger sub-problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems.

  **The O(n) solution of Fibonacci Sequence was Tabulation or Memoization?**

  (HW: Try solving Fibonacci problem using memoization)

  **How DP is different from Divide and Conquer?**

  (A: That has non-overlapping subproblems)

- How to identify a problem can be solved using DP

  (Source: GFG)

  - All dynamic programming problems satisfy the overlapping subproblems property and most of the classic dynamic problems also satisfy the optimal substructure property. Once, we observe these properties in a given problem, be sure that it can be solved using DP.
  - Typically, all the problems that require to maximize or minimize certain quantity or counting problems that say to count the arrangements under certain condition or certain probability problems can be solved by using Dynamic Programming.

- Steps to tackle a DP problem - the secret to solve any problem

  1. Identify if it is a DP problem
  2. Decide a state expression with least parameters
  3. Formulate state relationship
  4. Do tabulation (or add memoization)

*Let's follow these steps for solving the below problems for understanding them better*

## Solving some questions (0.5- 1 hour)

- Question - 1: http://www.spoj.com/problems/COINS/
- Question - 2: https://www.geeksforgeeks.org/ugly-numbers/

## General Coding Interview Tips (10 - 20 minutes)

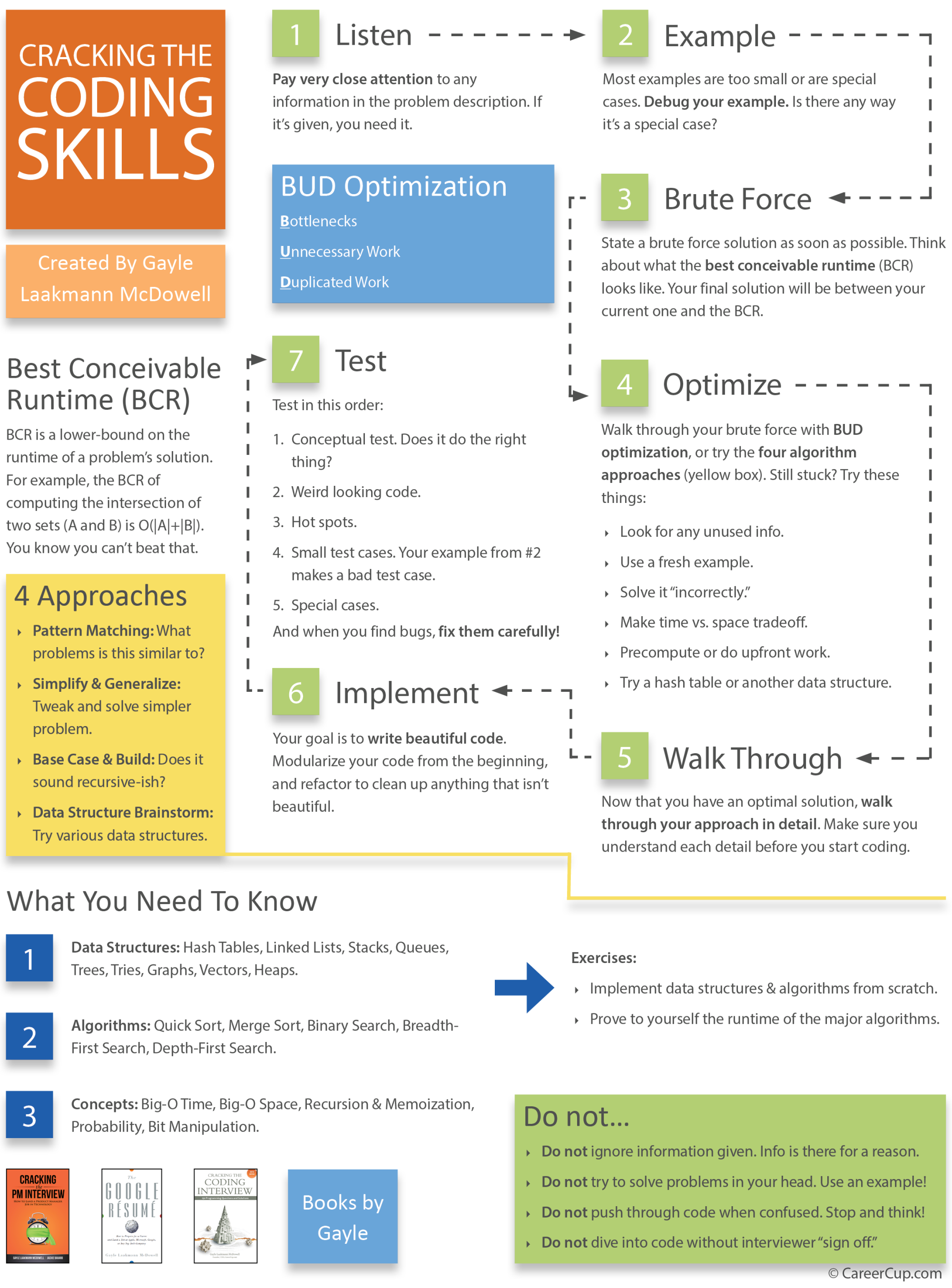
- How to approach any problem?

  The systematic approach:

  1. Clarification of Problem
  2. Identification of Test Cases, Edge Cases & Manual Solution
  3. Formulate Algorithm (PseudoCode)
  4. Dry Run the algorithm with the test cases
  5. Time & Space Complexity
  6. Optimizing the algorithm and check whether test cases are passing with the new solution
  7. Convert the logic into the actual working code. Write test cases
  8. Refactor code and make more object-oriented
- What 2nd Year should start?

  1. Look for internships (shortlist some companies)
  2. Contribute to Open Source
  3. Code daily for at least an hour
  4. Start projects to put in resume
  5. 2nd YEAR is really crucial: STUDY THE CONCEPTS REALLY WELL
  6. START CP TODAY! (if you haven't yet)
- What 3rd Year should do?

  1. WHAT 2nd years have to do
  2. Solve technical challenges on hackerrank or hackerearth
  3. Start making better projects
  4. Start choosing your SINGLE domain for expertise (SINGLE ROI)
- THE ULTIMATE FLOWCHART



CRACKING THE CODING SKILLS

Created By Gayle Laakmann McDowell

**1 Listen**

**Pay very close attention** to any information in the problem description. If it's given, you need it.

**2 Example**

Most examples are too small or are special cases. **Debug your example.** Is there any way it's a special case?

**BUD Optimization**

**B**ottlenecks
**U**nnecessary Work
**D**uplicated Work

**3 Brute Force**

State a brute force solution as soon as possible. Think about what the **best conceivable runtime** (BCR) looks like. Your final solution will be between your current one and the BCR.

**Best Conceivable Runtime (BCR)**

BCR is a lower-bound on the runtime of a problem's solution. For example, the BCR of computing the intersection of two sets (A and B) is O(|A|+|B|). You know you can't beat that.

**7 Test**

Test in this order:

1. Conceptual test. Does it do the right thing?
2. Weird looking code.
3. Hot spots.
4. Small test cases. Your example from #2 makes a bad test case.
5. Special cases.

And when you find bugs, **fix them carefully!**

**4 Optimize**

Walk through your brute force with **BUD optimization**, or try the **four algorithm approaches** (yellow box). Still stuck? Try these things:

- Look for any unused info.
- Use a fresh example.
- Solve it "incorrectly."
- Make time vs. space tradeoff.
- Precompute or do upfront work.
- Try a hash table or another data structure.

**4 Approaches**

- **Pattern Matching:** What problems is this similar to?
- **Simplify & Generalize:** Tweak and solve simpler problem.
- **Base Case & Build:** Does it sound recursive-ish?
- **Data Structure Brainstorm:** Try various data structures.

**6 Implement**

Your goal is to **write beautiful code**. Modularize your code from the beginning, and refactor to clean up anything that isn't beautiful.

**5 Walk Through**

Now that you have an optimal solution, **walk through your approach in detail**. Make sure you understand each detail before you start coding.

**What You Need To Know**

1. **Data Structures:** Hash Tables, Linked Lists, Stacks, Queues, Trees, Tries, Graphs, Vectors, Heaps.
2. **Algorithms:** Quick Sort, Merge Sort, Binary Search, Breadth-First Search, Depth-First Search.
3. **Concepts:** Big-O Time, Big-O Space, Recursion & Memoization, Probability, Bit Manipulation.

**Exercises:**

- Implement data structures & algorithms from scratch.
- Prove to yourself the runtime of the major algorithms.

**Do not...**

- **Do not** ignore information given. Info is there for a reason.
- **Do not** try to solve problems in your head. Use an example!
- **Do not** push through code when confused. Stop and think!
- **Do not** dive into code without interviewer "sign off."

Books by Gayle

© CareerCup.com

- See Cracking the Coding Interview (Page 30)

## QnA

## Mock Coding Round

- 45 Minutes
- 3 Questions