## How to understand any recursive code, step by step <https://www.byte-by-byte.com/recursion/>

Before we even get into writing our own recursive code, it’s critical that we have a clear understanding of how recursive code actually works. After all, how can you write something that you don’t understand.

The problem here is that recursive code isn’t exactly straightforward.

Most code executes in a linear fashion. We start at the top of the file and keep running until we get to the bottom. If there’s a loop, we loop through until we ultimately exit the loop, but then we keep going. Without recursion (and go-to statements, so help me God), that is as complicated as our non-recursive code will get.

But as soon as we throw in recursion, all bets are off.

As soon as we make a recursive call, we are jumping out of the original function and computing something completely different before we ever even come back to the original function.

Let’s look at a simple example of how the order of execution for recursive functions changes.

Consider the following code:



|  |  |
| --- | --- |
| 1  2  3  4  5 | void f(int n) {     if (n == 0) return;     f(n-1);     print(n);  } |

What is the output of this code?

If we read the code in the order that it’s written, then we might think that it would print the output of f(5) as 5, 4, 3, 2, 1. Afterall, the outer function prints 5 first, and then it subsequently gets decremented.

Unfortunately, though, this would be wrong. We have to evaluate the recursive call first before we can continue to the end of the function.

Therefore, our code executes as follows:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | f(5)     f(4)         f(3)             f(2)                 f(1)                     f(0)                     print(1)                 print(2)             print(3)         print(4)     print(5) |

As you can see, the proper output of this function is 1,2,3,4,5.

The key thing to understand here is that our output is not always going to occur in the order that it appears in our code.

**So how do we accurately compute the output?**

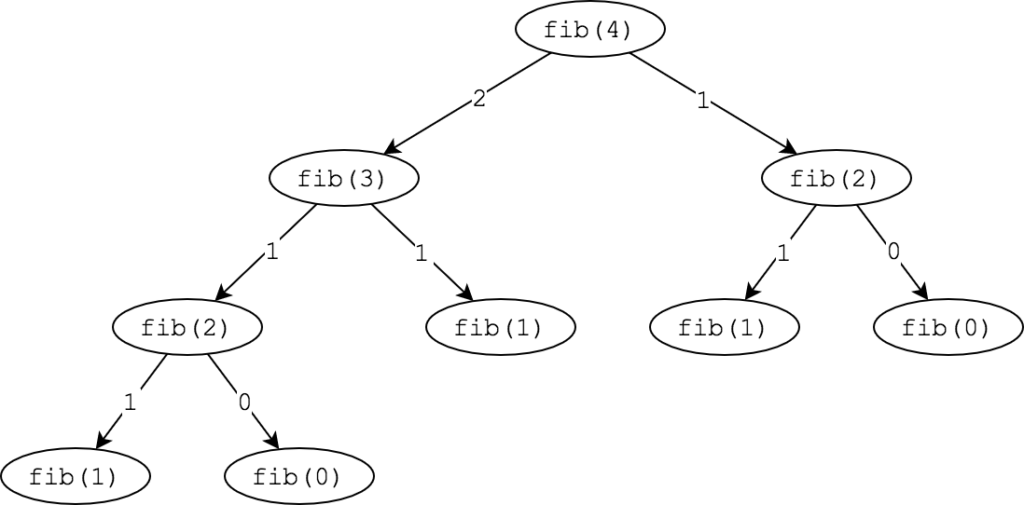
The key to understanding any recursive code is to walk through the code step-by-step. You are now the compiler.



The most important thing to do is to walk through the code exactly as the compiler would. As soon as you start assuming how the code will behave and not reading through it line-by-line, you’re screwed.

In this video, I show you exactly how to do this:

Here are the most important tips to doing this effectively:

* **Draw the recursive tree**. Recursive functions can act like a tree. The parent is the main function call and each recursive call made is a child node. The tree structure can be really useful because it helps you to easily keep track of variables. Here is what a tree might look like for the Fibonacci function.
* **Do not lose track of your variables**. This is absolutely critical. As functions get more complicated it becomes harder and harder to actually store everything in your head, especially since you have to do your recursion out of order. The tree will help you with this.
* **Practice**. I know this may not sound like the most fun task, but understanding recursive code is critically important. If you can understand the recursive code that other people write and why it works it will make it exponentially easier for you to write your own recursive code.

Start with some simple problems. Try to do this for a Fibonacci or factorial problem and then work your way up. You can find tons of examples of recursive code that others have written. Grab a random piece of code and try to interpret what it’s doing.

Then compare your results to what you get when you actually run the code. You’ll gain so many insights on how recursive code works through this process.

## Tail recursion, backtracking, and other core recursive concepts

Okay now we know what recursion is, we know why we need it, and we know how to understand a piece of recursive code when we see it.

It’s about time that we start getting into the meat of how to write our own recursive code.

In this section we’re going to start with some of the most important and asked about (although not necessarily both, ahem tail recursion) topics related to recursion.

### Return types

To start with, it is critical that we go over strategies for returning values from our recursive function. While some recursive functions may just print the results, in many cases, we are going to want to return a specific result to the caller. There are multiple different ways that we can do this recursively, some of which are better than others.

**Global Variable**

The first option (and the only one here that I’d say you should probably never use) is to store your result into a global variable. We all know that global variables aren’t a great solution when there is another option, so try to avoid this if possible.

For demonstration purposes, let’s look at a function that counts the number of even values in an array. Here’s how we could write this code using a global variable:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | int globalResult;    void countEvenGlobal(int[] arr) {     globalResult = 0;     countEvenGlobal(arr, 0);  }    void countEvenGlobal(int[] arr, int i) {     if (i >= arr.length) return;     if (arr[i] % 2 == 0) globalResult++;     countEvenGlobal(arr, i+1);  } |

The key here is that we will simply create a global variable and update the value as we recurse through our code. Whenever we find an even value in our array, we just increment the global variable, similar to how we might solve this problem iteratively.

**Passed Variable**

A better option that is similar to using a global variable is to use a passed variable. In this case we are going to pass a variable to our recursive function that we will update with the result as we go. Essentially this is like having a global variable except that it is scoped to our function.

Our code looks like this:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class ResultWrapper {     int result;  }    int countEvenPassed(int[] arr) {     ResultWrapper result = new ResultWrapper();     result.result = 0;     countEvenPassed(arr, 0, result);     return result.result;  }    void countEvenPassed(int[] arr, int i, ResultWrapper result) {     if (i >= arr.length) return;     if (arr[i] % 2 == 0) result.result++;     countEvenPassed(arr, i+1, result);  } |

Note in this code, if we are using a language like Java, we have to use some sort of ResultWrapper class because we cannot directly pass a pointer to our function and we will be updating the value as we go.

The advantage of this approach is that it is generally the easiest for most people to understand. It also gives us the opportunity to do tail recursion, if that is something that our language (and the problem) supports.

**Build the result as we return**

Our third strategy is a bit more difficult to understand because we are essentially doing the work that we need to do in reverse order. We will return partial values as we return from our recursive calls and combine them into the result that we want.

The reason that this strategy is critical is that it will make it possible for us to use the FAST Method to solve dynamic programming problems. If we use one of the previous two approaches, we are not isolating our subproblems and so it becomes impossible for us to actually cache the values that we want.

Here is what our solution might look like:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int countEvenBuiltUp(int[] arr) {     return countEvenBuiltUp(arr, 0);  }    int countEvenBuiltUp(int[] arr, int i) {     if (i >= arr.length) return 0;     int result = countEvenBuiltUp(arr, i+1);     if (arr[i] % 2 == 0) result++;     return result;  } |

For the most part, you can use these return strategies interchangeably for different recursive problems.

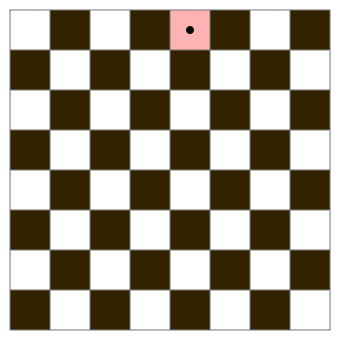
Don’t bother practicing using global variables, since you really should never use that in your interview. However, I would recommend practicing both of the others. Try doing a practice problem both ways. See what the similarities and differences are. The extra practice will do you good!

### Backtracking

What is backtracking?

Backtracking is an essential strategy that we use in recursion. It essentially allows us to try different possible results without actually knowing ahead of time what the right result is.

Consider the Knight’s Tour problem. In this problem, we want to find a path that a knight can follow on a chessboard where it will visit each square exactly once.



We don’t actually know what the right more is to make ahead of time and a knight can make up to 8 different moves from any one square, so we just have to pick a move at random.

But obviously not all combinations of moves will take us through a valid path. Some moves might trap us in a corner that we can’t get out of without visiting the same square twice.

So what do we do? We “backtrack”.

The idea behind backtracking is simply that we retrace our steps backwards and then try a different path. This allows us to try every different path until we ultimately find one that is valid.

Another example of this that you’re likely already familiar with is depth-first search in a tree. When we are looking for a node in a tree, we consider the leftmost branch first. Then what happens if we don’t find the node we’re looking for? We backtrack.

We go back up one level and try the other child of the parent node. If we still haven’t found what we are looking for, we go back up the tree even further. This continues until we find what we are looking for or have gone through the entire tree.

Some common backtracking problems include:

* [0-1 Knapsack](https://www.byte-by-byte.com/01knapsack/)
* [Sudoku Solving](https://leetcode.com/problems/sudoku-solver/)
* [N Queens](https://leetcode.com/problems/n-queens/)
* [And many more…](https://leetcode.com/tag/backtracking/)

**A note on backtracking**: A lot of people hear “backtracking” and they stress out about it, but my guess is that if you’ve done any recursion in the past, you’re already familiar with the basic concept. Don’t stress too much about this one specifically. Just make sure you understand how to implement the 6 recursive patterns and you’ll learn how to do backtracking as a side-effect.

### Tail recursion

Okay this is the most unnecessarily worried about concept in all of recursion. Tail recursion almost never comes up in an interview and isn’t even supported by [most major programming languages](https://en.wikipedia.org/wiki/Tail_call).

That said, I’m not opposed to giving it it’s fair representation here. So let’s cover the basics of tail recursion.

Tail recursion is an optimization technique that you can use to improve the space complexity of a small subset of recursive problems. To understand how this works, we do need to talk about space usage for a second.

When you make recursive calls, the computer needs to save the state of the current function (all the variables, the current position that you’re executing at, etc) so that you can return back after the end of the recursive call.

Therefore, for every recursive call you make, you are using some space on the call stack. The more recursive calls you make, the more space you’re using.

Tail recursion allows us to avoid having to use extra space. Imagine that you write a function where the very last line of the function is the recursive call that immediately returns. Then do you really need to save the previous function state on the call stack?

No. You can just return the value from the deepest level of your recursion directly to the top.

That is what tail recursion does for us. If we have a single recursive call that is the very last thing before we return, then we can optimize out the extra stack space.

To look at our simple example from earlier, which of these is tail recursive?



|  |  |
| --- | --- |
| 1  2  3  4  5 | void f1(int n) {     if (n == 0) return;     f1(n-1);     print(n);  } |

Or



|  |  |
| --- | --- |
| 1  2  3  4  5 | void f2(int n) {     if (n == 0) return;     print(n);     f2(n-1);  } |

Based on our definition, you can see that in f2(), the very last thing we do is our recursive call. f1() prints after we make the recursive call so we can’t optimize it because we are going to need to return back to the previous function so that we can call print(n).

So in limited circumstances, tail recursion can be useful. It is particularly useful while doing functional programming, since you are often doing basic things like iteration recursively that can be easily optimized.

However, most of the time, tail recursion won’t be helpful to us. Chances are the language that you’re using won’t support it and on top of that, many of the examples that we will see in our interviews will require us to make multiple recursive calls within our function. If you have multiple calls, they can’t both be the last thing before we return and so tail recursion is out of the question.

### Recursive Time and Space complexity

Time and space complexity for recursive problems tends to pose quite a challenge. Because recursive problems are so hard to parse in the first place, it is often non-obvious how we would compute the complexity.

However on the bright side, there are a couple of heuristics that we can use to help us.

**Time Complexity**

Remember when we represented our recursion as a tree structure? Well that tree structure can show us very clearly the number of recursive calls we’re making.

And simply put, the time complexity is going to be O(number of recursive calls \* work per recursive call).

With this formula, we are now able to simplify dramatically into two components that we can more easily calculate.

First, let’s talk about the number of recursive calls.

How do we estimate the total number of recursive calls without drawing out the whole tree? Well if we wanted to compute the number of nodes in a tree, we can look at the **height of the tree** and the **branching factor**.

The height of the tree is simply how deep our recursion goes. For example if you keep recursively calling f(n-1) and your base case is when n == 0, then our depth is going to be O(n), since we keep decrementing until n reaches 0. If we have multiple different recursive calls, we just consider whatever the maximum depth is (remember that Big Oh is an upper bound).

The branching factor is also fairly straightforward to figure out. The branching factor is defined as the maximum number of child nodes any parent has in a tree. For example, if we have a binary tree, the branching factor will be 2.

To find the branching factor of our recursive tree, we simply need to look at the maximum number of recursive calls. In our Fibonacci function, we call fibonacci(n-1) and fibonacci(n-2) every time, which gives us a branching factor of 2.

If we said:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int f(n) {     if (n == 0) return 0;       int sum = 0;     for (int i = 0; i < n; i++) {         sum += f(n-i);     }       return sum;  } |

What would be the branching factor here?

At first glance, it seems like it will be hard to compute. After all, the branching factor depends on n and n keeps changing.

But remember we only need the worst case here. So we don’t need to consider every case just what is the worst case branching factor, which in this case is n.

Finally, now that we have the branching factor and the height of our recursive tree, how can we use these to find the time complexity of our recursive function.

We can use the following heuristic:

Number of nodes = O(branching\_factordepth\_of\_recursion)

Knowing the number of nodes, we get that our total time complexity is:

O(branching\_factordepth\_of\_recursion \* work\_per\_recursive\_call)

Work per recursive call is simply the amount of work that we’re doing in our function other than when we call it recursively. If we have any sort of for loop, such as in the example above, our work per recursive call will be proportional to that loop, or O(n) in this case.

That gives us a time complexity of O(nn \* n) for the function above.

**Space complexity**

Recursive space complexity is a bit easier for us to compute, but is also not exactly trivial.

The space does not depend on the branching factor, only on the height and then amount of space used per recursive call.

When we think about this for a minute, it makes sense. The branching factor leads to multiple different paths through our recursive tree. We have to consider each of these separately for the time complexity because time is additive. There is no way to reuse time.

Space, however, can be reused. Each time we recurse down, we will be using more and more space. However, once we return from our recursive function, we clear up space that can be reused for the next path.

This gives us a formula for our space complexity of simply:

O(depth\_of\_recursion \* space\_per\_recursive\_call)

It is, however, important to remember that recursion does actually use up space, since that is something that many people often tend to forget.erview..

## The 6 core recursive patterns

One of the main reasons why people tend to find recursion so difficult is that they don’t have a mental model for how to view recursion as a whole.

When you look at every recursive problem and see how different they are, it can be really difficult to figure out what is going on.

Not only that, but if every problem seems completely different, how can you ever feel confident that you will be able to solve a problem in your interview.

After combing through dozens of common recursive problems, I’ve categorized all recursive problems into one of 6 categories. These 6 categories, based around the core pattern used to solve the problem, allow us to put a finite bound on the scope of recursive problems that we could need to solve.

If you understand each of these patterns and how to code them, then you can apply these concepts to almost any recursive problems that might come up in your interview. These 6 patterns are **Iteration**, **Subproblems**, **Selection**, **Ordering**, **Divide & Conquer**, and **Depth First Search**. For the remainder of this section, we will go into each in more detail.

### Iteration

As you hopefully know, any problem that can be solved recursively can also be solved iteratively and vice versa. This is a fundamental concept behind functional languages, where you use recursion to do everything; there are no loops.

While this is not something that you’re often going to need to do recursively, this pattern can come in useful once in awhile, particularly if you want to be able to refer back to items you’ve looped through previously.

For example, consider the problem of printing a linked list in reverse order. There are plenty of approaches we can take for this problem, but recursion is uniquely concise:



|  |  |
| --- | --- |
| 1  2  3  4  5 | void printReverse(Node n) {     if (n == null) return;     printReverse(n.next);     print(n.val);  } |

For iteration, we simply make our recursive call on the remainder of our inputs, either by passing in the next node (as we did above) or by incrementing some index. While this doesn’t come up too often, this is one of the simplest applications of recursion.

### Subproblems

Here we have what most people probably think of when they think of “recursion”: Breaking down a problem into its subproblems. Technically, you could use this pattern to solve any recursive problem, but the advantage of the other patterns is that they get much more specific about helping you do that.

With all that being said, there are some problems that just lend themselves to being broken down into subproblems.

One example of this is the Fibonacci problem that we’ve talked about earlier in this post. In that case, even the mathematical definition of a Fibonacci sequence is recursive.

Another problem that frequently comes up is the [Towers of Hanoi problem](https://en.wikipedia.org/wiki/Tower_of_Hanoi). In this case, we can simply break down our problem by considering the subproblem of moving the top n-1 disks. If we can move the top n-1 disks, we just move all of them to the middle pin, move our bottom disk to the destination pin, and then move the above disks again over to the destination pin. This works because all of the disks above our bottom pin are not affected by any pins below them.

In problems like this, you have to look for those subproblems with which to break up the problem. However, there aren’t too many problems in this category, since most can be more explicitly solved with one of the following patterns.

### Selection

This is my favorite pattern to test people on because it is one of the most common patterns to come up in recursion (and dynamic programming). In this pattern, we are simply finding all of the combinations of our input that match a certain criteria.

Consider for example the [0-1 Knapsack Problem](https://www.byte-by-byte.com/01knapsack/). In this problem, we have a series of items that have weights and values. We want to figure out what the maximum value is that we can achieve while remaining under some fixed weight.

Many people recognize this as a dynamic programming problem, since it’s a classic example, but let’s look at it from a recursive standpoint.

What would be a brute force solution to this problem? Well we can easily validate for a given combination of items whether it is under the maximum weight, and we can also easily compute the weight for any combination. That means if we could just generate all the different combinations, we could figure out the optimal answer.

Figuring out all the combinations is the core of a selection problem. I like the term “selection” because the way our code works is to simply include/exclude, or “select”, each item in our list.

The brute force code to generate all combinations might look something like this (this is simplified, but you get the idea):



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | List<List<Integer>> combinations(int[] n) {     List<List<Integer>> results = new LinkedList();     combinations(n, 0, results, new LinkedList<Integer>());     return results;  }    void combinations(int[] n, int i, List<List<Integer>> results, List<Integer> path) {     if (i == n.length) {         results.add(path);         return;     }       List<Integer> pathWithCurrent = new LinkedList(path);     pathWithCurrent.add(n[i]);       // Find all the combinations that exclude the current item     combinations(n, i+1, results, path);       // Find all the combinations that include the current item     combinations(n, i+1, results, pathWithCurrent);  } |

Once we understand this basic pattern, we can start to make optimizations. In many cases, we may actually be able to filter out some combinations prematurely. For example, in the Knapsack problem, we can limit our recursion to only consider combinations of items that stay below the prescribed weight.

If you spend the majority of your time on any one pattern, it should be this one. It comes up so frequently in so many different forms. Some good problems to get your started are [0-1 Knapsack](https://www.byte-by-byte.com/01knapsack/), [Word Break](https://leetcode.com/problems/word-break/), and [N Queens](https://leetcode.com/problems/n-queens/description/).

### Ordering

This pattern is the permutation to Selection’s combination. Essentially here we’re looking at any case in which we want to consider different orderings of our values. The most straightforward problem here is just to figure out all of the permutations of a given set of elements, although just like with selection, we may add in additional restrictions.

Some examples of problems that fall under this category are Bogosort (sorting a list of items by generating all permutations and determining which is sorted), finding all numbers that can be made from a set of digits that match a certain property, determine all palindromic strings that can be made from a set of characters, and more.

In its most simple form, this is one way that we can brute force generate all permutations of a list. You can also see an [alternative approach here](https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/):



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | List<List<Integer>> permutations(Set<Integer> n) {     List<List<Integer>> results = new LinkedList();     permutations(n, results, new LinkedList<Integer>());     return results;  }    void permutations(Set<Integer> n, List<List<Integer>> results, List<Integer> path) {     if (n.isEmpty()) {         results.add(path);         return;     }       for (int i : n) {         // For now ignore concurrent modification issue         n.remove(i);         List<Integer> pathWithCurrent = new LinkedList(path);         pathWithCurrent.add(i);         permutations(n, results, pathWithCurrent);         n.add(i);     }  } |

As you can hopefully see, there is a lot of similarity between this solution and the combinations solution above. By understanding these two patterns and their variants, you can cover a huge number of the different possible recursive problems you might be asked in your interview.

### Divide and Conquer

If you know about some of the common applications of recursion, you probably saw this one coming. Divide and conquer is the backbone to how we use techniques such as mergesort, binary search, depth first search, and more. In this technique, we attempt to break the problem space in half, solve each half separately, and then come back together.

Most frequently, this pattern is used as part of common algorithms that you should already know, such as the one I mentioned above, but there are a handful of problems for which this can be valuable.

For example, consider trying to determine all of the unique binary trees that we can generate from a set of numbers. If you pick a root node, you simply need to generate all of the possible variations of a left and right subtree.

Consider trying to find the maximum and minimum value of a list using the minimum number of comparisons. One way that we can do this is by splitting the list repeatedly, much the same as how we would do mergesort.

This technique generally applies to tree and sorting/searching problems where we will be splitting the problem space and then recombining the results.

There is also a slightly different case in which D&C is very useful: Trying to find all of the ways to group a set. Imagine, for example, that you have a mathematical function. You want to determine all of the different ways that you can group the arguments. This can be done by recursively trying splitting your formula at every different point. Here is an example of what this code might look like. It uses a string, but demonstrates the same concept:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | List<String> parentheses(String s) {     if (s.length() == 1) {         List<String> result = new LinkedList<String>();         result.add(s);         return result;     }       List<String> results = new LinkedList<String>();       for (int i = 1; i < s.length(); i++) {         List<String> left = parentheses(s.substring(0, i));         List<String> right = parentheses(s.substring(i, s.length()));           for (String s1 : left) {             for (String s2 : right) {\                 results.add("(" + s1 + s2 + ")");             }         }     }       return results;  } |

In this example, we take the string and we try finding every different midpoint. For example “abcd” becomes “(a)(bcd)”, “(ab)(cd)”, and “(abc)(d)”. Then we recursively apply this to each of the halves of the string.

### Depth First Search

Depth first search is our final pattern that our recursive functions can fall under. It’s also one that we’ll likely find ourselves using a lot. That’s because DFS (and BFS) are used extensively whenever we are doing anything with trees or graphs.

In terms of the details of trees and graphs, I’m not going to go in depth here. There is way too much to cover so I’m going to leave them to their own study guides.

The main important thing with DFS is to understand how not only to search for a node in a tree or graph but also how to actually find the path itself. If you can’t do that, you’re severely limiting what you can actually do.

Here is what the code might look like to find the path to a specific node in a tree:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | List<Node> pathToNode(Node root, int val) {     if (root == null) return null;     if (root.value == val) {         List<Node> toReturn = new LinkedList<Node>();         toReturn.add(root);         return toReturn;     }       List<Node> left = pathToNode(root.left, val);     if (left != null) {         left.add(0, root);         return left;     }       List<Node> right = pathToNode(root.right, val);     if (right != null) {         right.add(0, root);         return right;     }       return null;  } |

And there’s not that much more to it than that. In most cases, you will use depth first search as part of a larger problem, rather than actually having to modify it in any significant way. Therefore the most important thing is understanding the core of how DFS works.

With these 6 recursive patterns, you will be able to solve almost any recursive problem that you could see in your interview. The key is to look for those patterns in the problems that you are solving. If you see the pattern, use it. If not, try something else. Different patterns will work better for different people, so do what feels right to you.