So let’s talk about these different patterns.

**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:

public static void printReverse(Node n) {

if (n == null) return;

printReverse(n.next);

System.out.println(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.

**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://el2.convertkit-mail2.com/c/n4uk94r23psvh5q53m/e0hph0/aHR0cHM6Ly93d3cuYnl0ZS1ieS1ieXRlLmNvbS8wMWtuYXBzYWNrLw==). 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):

public static List<List<Integer>> combinations(int[] n) {

List<List<Integer>> results = new LinkedList();

combinations(n, 0, results, new LinkedList<Integer>());

return results;

}

private static 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.

There is much more to this pattern than I can include in this email, but it is one of the most important patterns to understand, so it is worth taking time to go through several examples. [0-1 Knapsack](https://el2.convertkit-mail2.com/c/n4uk94r23psvh5q53m/75u7h2/aHR0cHM6Ly93d3cuYnl0ZS1ieS1ieXRlLmNvbS8wMWtuYXBzYWNrLw==), [Word Break](https://el2.convertkit-mail2.com/c/n4uk94r23psvh5q53m/o2ikhw/aHR0cHM6Ly9sZWV0Y29kZS5jb20vcHJvYmxlbXMvd29yZC1icmVhay8=), and [N Queens](https://el2.convertkit-mail2.com/c/n4uk94r23psvh5q53m/zefgh7/aHR0cHM6Ly9sZWV0Y29kZS5jb20vcHJvYmxlbXMvbi1xdWVlbnMvZGVzY3JpcHRpb24v) are some good places to start.

**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](https://el2.convertkit-mail2.com/c/n4uk94r23psvh5q53m/pxtehq/aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvQm9nb3NvcnQ=" \t "_blank) (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://el2.convertkit-mail2.com/c/n4uk94r23psvh5q53m/x6cph3/aHR0cHM6Ly93d3cuYnl0ZS1ieS1ieXRlLmNvbS9wZXJtdXRhdGlvbnMv):

public static List<List<Integer>> permutations(Set<Integer> n) {

List<List<Integer>> results = new LinkedList();

permutations(n, results, new LinkedList<Integer>());

return results;

}

private static 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.

**Subproblems**

I know this may come as a surprise, since at the beginning of the email I was bashing on people who tell you to just figure out the subproblems, but there is a time and a place for this. Although I find that many recursive problems don’t have such obvious subproblems as people would lead you to believe, there are some problems in which the subproblems are obvious.

One example of this that you certainly know is the Fibonacci problem. 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://el2.convertkit-mail2.com/c/n4uk94r23psvh5q53m/6zseho/aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvVG93ZXJfb2ZfSGFub2k=). 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, these types of problems don’t come up too often, so for the most part you can just memorize specific examples.

**Divide and Conquer**

Hopefully, if you know about some of the common applications of recursion, you 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.

I’m not going to go into too much depth on this pattern because it is primarily 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.

**Depth First Search**

This is our final pattern and I left it for last for two reasons:

1. In a lot of cases, this primarily applies to graphs and trees, which is somewhat out of the scope of our discussion on pure recursion.
2. Almost any other recursive problem can be framed as DFS.

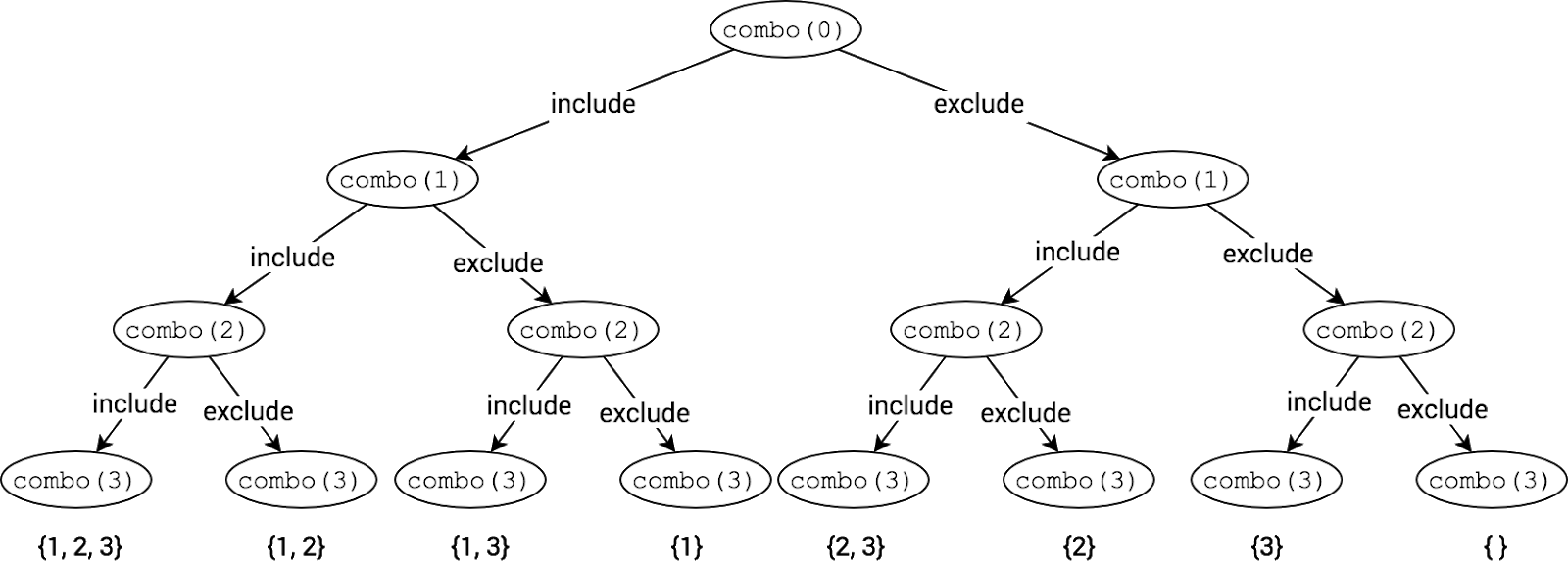
Now in terms of graphs and trees, I’m not going to go into any real detail. DFS is a basic technique that you should really know like the back of your hand. If you don’t, it’s time to get studying.

What I would like to discuss, however, is how we can apply other problems to DFS. Almost any of the problems that we’ve discussed above can be understood using DFS.

Let’s consider the example of brute force finding all combinations. In this problem our starting point for our DFS is an empty list. Then each step, we can search down one of two different paths: Either we include the current element or we don’t in our result.

Here is a diagram of what this looks like in a tree format. combo(i) refers to the recursive call with index i in our array:

*(If you don't see images below, click "Display images")*



As you can see along the bottom, we find every possible combination of elements from the input array {1, 2, 3}.

Whether you find this visual helpful, much of recursion can be most easily explained using a tree. Whether you use DFS or one of the other patterns above is relatively unimportant.

Phew. That was kind of a lot, wasn’t it? Here’s a quick recap of the 6 recursive patterns:

* Iteration
* Selection
* Ordering
* Subproblems
* Divide and Conquer
* Depth-first Search

If you truly understand these 6 patterns, you will have everything that you need to solve any recursive problem that might come up during your interview.

Now I know that this may still be a lot, and this long-ass email barely scratched the surface. Tomorrow, I’m going to show you a better way to learn all of this stuff so that you can really crush your interviews and get that dream job.

For now, though, pick one of these patterns and really study it. Code up one or two brute force examples and see if you can optimize them. Work through the code and make sure you really understand it. This will really help accelerate you towards understanding recursion.

Best,

Sam