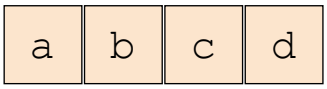
Permutations

This chapter shows how one can reason about recursive problems without extensive mathematical knowledge

Finding permutations of a given character or integer array is a common interview problem. Assume we have the following character array of size 4 with no repeating characters, and we need to find out all the permutations of size 4 for the below array.



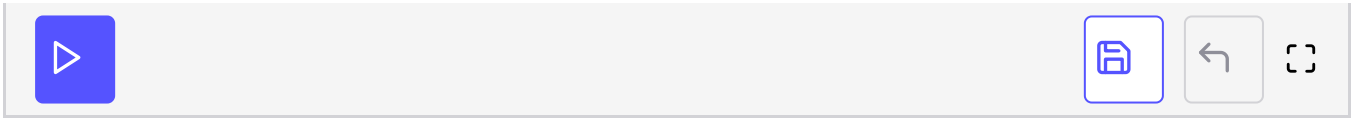
Character Array to Permute

The permutations will be generated using recursion and we'll give a solution that is easy to understand, even though it may not be the optimal one in terms of space usage. Our emphasis is on learning to reason about the complexity of algorithms. Take a minute to read through the code listing below before we attempt to figure out the time complexity.

Permutation Implementation

```
class Demonstration {
    public static void main( String args[] ) {
        char[] array = new char[] { 'a', 'b', 'c', 'd' };
        char[] perm = new char[array.length];
        boolean[] used = new boolean[256];
        permute(array, perm, 0, used);
    }

    /**
     * Note that we are using a boolean array to remember what
     * character we have already used and another array perm
     * which contains the actual permutation. We can actually
     * generate permutations without these additional arrays but
     * for now the desire is to make the code easier to understand.
     */
    static void permute(char[] array, char[] perm, int i, boolean[] used) {

        // base case
        if (i == perm.length) {
            System.out.println(perm);
```

```
            return;
        }

        for (int j = 0; j < array.length; j++) {

            if (!used[array[j] - 'a']) {
                perm[i] = array[j];
                used[array[j] - 'a'] = true;
                permute(array, perm, i + 1, used);
                used[array[j] - 'a'] = false;
            }
        }
    }
}
```



Permutations of a String

The number of permutations for a string of size n is n! (pronounced as n factorial). A factorial of a number is the product of all the numbers starting from 1 up to n. Below are a few examples:

0! = 1
3! = 3 x 2 x 1 = 6
5! = 5 x 4 x 3 x 2 x 1 = 120
10! = 10 x 9 x 8 .... 1 = 3628800

The number of ways to permute a string of size n is ***n!***. One way to rationalize is to think of n empty slots.

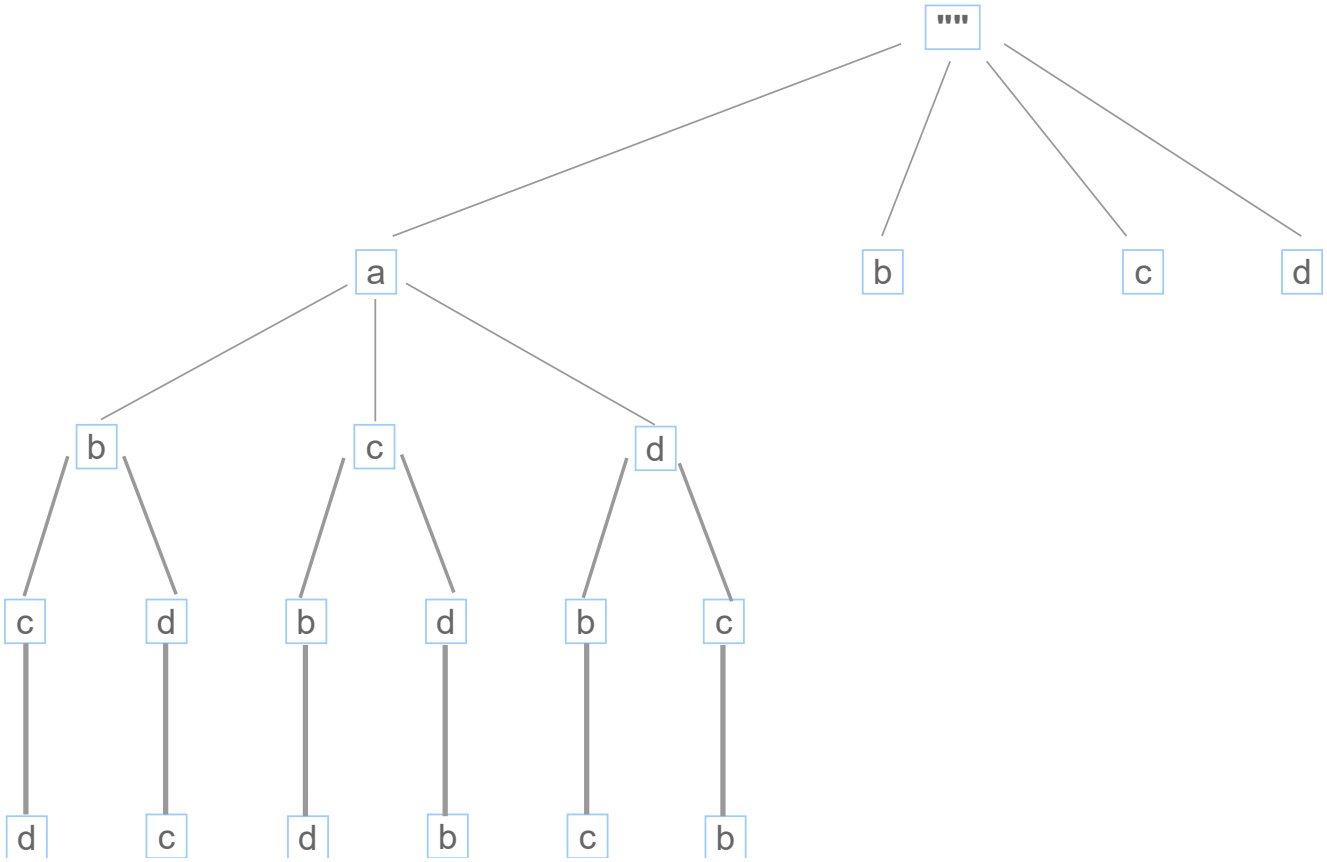
We can pick any of the n characters and put it in the first slot. Once the first slot if fixed, we can fill the second slot in any of the n-1 ways using the remaining n-1 characters of the string. In the same vein of reasoning, we can fill the third slot in n-2 ways, and so on and so forth. By the time we get to the nth slot, there will only be one element left and we will only have one way to fill it up. Thus the total number of ways we can fill up n empty slots (i.e. arrange n elements in a different order or by the number of permutations) is:

$$n * (n-1) * (n-2)...1$$

$$= n!$$

We can immediately see that the time complexity for generating all the permutations is also O(n!). Note that even though this is a recursive problem, it doesn't lend itself to the divide-and-conquer strategy; it can't be broken up into smaller subproblems that can then be combined. The solution is to find every possible permutation, and that would take n! time.

Having said that, the intent of this chapter is to reason about complexity using recursion trees, so let us draw out the recursion tree and see how one can deduce the complexity from the tree.



Recursion Tree

The above diagram shows the complete recursion tree when the first slot is filled with the character 'a'. The subtrees for b, c and d would be similar when chosen as first characters. Each node of the tree represents a recursive call to the permute function itself. The root of the tree is an imaginary empty string to ease the thought process. Once we analyze the complexity of the subtree rooted at "a", we can simply multiply it with n to get the final complexity. This stems from the fact that there will be *n* such subtrees rooted with each of the n characters in the string.

If we can find the total recursive calls made and the cost incurred in each

If we can find the total recursive calls made and the cost incurred in each invocation, we can multiply the two to get the total cost for the subtree rooted at "a".

Cost of each *permute* invocation

The permute method in the given code consists of the base case, where we detect that a permutation is complete and output it. The rest of the code caters to the regular case, where we made a recursive call but still have not formed a permutation. In the base case we are outputting the array, which is an O(n) operation behind the scenes since every character needs to be traversed in order to be output to the console.

In the regular case, note that the loop runs for the entire length of the array, though it may skip those characters that have already been used. The cost incurred is still O(n) as the loop runs the length of the array. In summary, we can say that the cost of each invocation of the *permute* method will be O(n).

Number of recursive calls

Once we fix "a" as the root, then there are only 3 choices left for the second slot. Therefore, the fan-out from the root element is 3. Once the second slot is filled, we are left with only 2 elements, and thus the fan-out from the nodes at the second level of the tree (rooted at a) is 2. On filling the third slot, there's only one element left and the fan-out is only one. We can generalize that the fan-out would decrease in the following order:

$$n$$

$$n-1$$

$$n-2$$

$$.$$

$$.$$

$$1$$

Note that the fan-out from our imaginary empty string root element is n, which is the number of ways we can fill up the first slot.

Now to determine the total number of invocations, which are represented by

the nodes in the recursion tree, we need to find the nodes at each level of the tree and sum them up. At any given level we'll multiply the fan-out for that level with the number of nodes that appear in the previous level.

Level	Fanout	Fanout * Nodes in Previous Level
1	n	n * 1
2	n - 1	(n-1) * n
3	n - 2	(n-2) * (n-1) * n
4	n - 3	(n-3) * (n-2) * (n-1) * n
.	.	.
.	.	.
nth	1	1 * 2 ... (n-3) * (n-2) * (n-1) * n

If we sum up the third column of the above table we'll get the total number of nodes or invocations for the permute method. However, the summation of all the terms in the third column will require mathematical gymnastics on our end. As software developers, we need to find a ballpark or asymptotic behavior rather than the exact expression representing the time complexity for a given algorithm. If you look carefully, at the nth level, the number of nodes is exactly n! and for each of the previous levels the number of nodes will necessarily be less than n!. There are a total of n levels, excluding the root level which we only introduced for instructional purposes. We can then say that the total number of nodes is capped by:

$$Total\ number\ of\ nodes = n * n!$$

This insight will help us determine the big O for the algorithm.

Tying it together

Now we know the total invocations of the permute method and the cost of each invocation. The running time of the algorithm is given as

$$T(n) = Recursive\ calls\ made * Cost\ of\ each\ call$$

$$T(n) = (n * n!) * n$$

$$T(n) = n^2 * n!$$

To further simplify, we can reason that n^2 is less than the product of (n+1)* (n+2)

$$T(n) = n^2 * n! < (n+2) * (n+1) * n!$$

$$T(n) = n^2 * n! < (n+2)!$$

$$T(n) = (n+2)!$$

Dropping constants

$$T(n) = O(n!)$$

1 If we tweaked the permutation problem to produce all the permutations of a string of size 1 to n, then what would be the complexity of the resulting algorithm?

2

What would be the time complexity of generating combinations of size 2 for a character array of size n?

3

Without writing code, can you guess what would be the time complexity of generating combinations of all sizes for a character array of size n?

Check Answers