Chapter 1 : The beginning

1.1 What *is* an algorithm

Say you are on a road trip with your friends, somewhere far far away. Your goal is simple: get to the destination safely. Now you are pretty sure you are lost, so you decide to open your window and ask a bystander for the way to your destination (lets just say you had no reception for your google maps). The guy tells you which path to go in, what turns to make, quite often with some landmarks to ensure you get it right. You follow whatever *instructions* he gave you, and arrive at your destination, safe and sound, thus, fulfilling your goal.

Now lets think about this from a different approach. Your goal (problem at hand) was to reach the destination. You were given the instructions (rules) to follow to reach the destination. Finally, you reach the destination (problem solved!). This is what an *algorithm* is.

In short, an algorithms is a set of rules or instructions to be followed for doing a calculation, or solving a problem. Here, the instructions given by the bystander is the algorithm: not following this, or even changing the order of these instruction may lead to you not reaching the destination.

1.2 How to compare algorithms

Taking the same example of giving directions, suppose there are two routes leading to your destination: one being longer than the other. Keep in mind that the shorted route and the longer route have a different set of instructions, or, as we formally defined, algorithms. Usually, we prefer to take the shortest route to the destination. But it is also possible that the longer route is more well-built. Thus, if we are in a hurry, we might choose the shorter one, but those who value comfort over speed would prefer the longer route.

There are many ways to compare algorithms. The primary comparison is based on two factors:

- time taken for the algorithm to solve the problem, a.k.a time complexity
- space complexity

1.2.1 Time complexity

As the name suggests, it has to do with the time taken by the algorithm to solve the problem. In the above example, in terms of *time*, the shorter path might be the way to go, and hence, a person in a hurry (or not) prefers to use the shorter path. Formally, time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. It is denoted by O(t) , which is called the "Big-O" notation. In general, an algorithm with time complexity $O(t_1)$ is better than an algorithm with time complexity $O(t_2)$, when $t_1 < t_2$ (by better, it just means that it has a quicker solving time)

1.2.2 Space complexity

Again, from the name, it has to do with the space taken by the algorithm to solve the problem. Formally space complexity is a measure of how efficient your code is in terms of memory used, with respect to input size. Similar to time complexity, an algorithm with space complexity $O(s_1)$ is better than an algorithm with space

complexity $O(s_2)$, when $s_1 < s_2$ (by better, it just means that it has a lesser space requirement for the same input). (it is also represented in the big O notation)

Ideally, a good algorithm must have low time as well as space complexity. Unfortunately, such an algorithm seldom exists in the real world, and the choice of which algorithm to use depends largely on what/where it is being used. For instance, consider sorting algorithms. Selection sort and merge sort are among the famous sorting algorithms. Selection sort has a time complexity of $O(N^2)$ and a space complexity of O(1) [O(1) means constant, and is independent of the size of the input], where N is the number of elements to be sorted. Merge sort has a time complexity of O(NlogN) and a space complexity of O(N). Suppose we have a large number of elements to be sorted (large N). Then, if we run the sorting on a system with very limited memory, we have to use selection sort, as it has a lesser memory need (space complexity) than mergesort, and when we want the sorting to be done fast, we use mergesort, by compromising on memory. Thus, there is most often a compromise on which algorithm to use to solve a problem.

1.3 Algorithmic Paradigms

All of us might probably have found an instance of where multiple ways of thinking helped different people to solve a same problem, or to arrive at the same conclusion. A very common example is math ,where there is "no one particular method" to arrive at the answer. A person who has no depth in integration, but is well versed in differentiation, can obtain the answer for a multiple choice question by differentiating all the options provided, and still get the same answer as a guy who used actual integration to solve the question.

Any basic, commonly used approach in designing algorithms could be considered an algorithmic paradigm. Formally, an algorithmic paradigm or algorithm design paradigm is a generic model or framework which underlies the design of a class of algorithms.

For instance, we have a letter to be delivered to a specific locker. Now, the letter only has the name of the person it has to be delivered to, and there are many lockers. One way of delivering the letter, would be to manually go through the locker room, checking each locker to see a name that fits the letter. This particular "paradigm" is called **Brute Force** paradigm, which is systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement. Another example is guessing the password of an iphone: we might have to try all the 10^4 combinations for the 4 digits.

Now, say 2 of our friends arrived to our aid. We can split up the locker room into 3 sections, and send one person to each section. This way, the amount of work done by one person reduces, and takes a lesser amount of time to cover more lockers (3 times faster, obviously). This way of dividing the problem at hand to smaller subproblems, and solving them recursively is called **Divide and conquer**. Mergesort is an excellent example of this paradigm.

There are a variety of other algorithm paradigms, like the **greedy method** (Find solution by always making the choice that looks optimal at the moment- don’t look ahead, never go back. As we shall learn later, the Huffman coding is a greedy algorithm), **Dynamic programming**, **sliding window paradigm** (an array, for example, is divided into a subarray. The subarray is visualized like a sliding window as it movies from one end of the array to another. The LZ algorithm, as we shall see later, belongs to this category.) etc.

1.4 Summary

By the end of this chapter, the reader must be familiar with the basics regarding algorithms, how to compare algorithms and a few important algorithm paradigms.