Module-3: Time Complexity and Sorting

CRUx Bootstrap

3.A: Efficiency

Your code might be inefficient in two ways - **time** and **memory** - and such inefficiencies result in corresponding negative verdicts - **Time Limit Exceeded**, and **Memory Limit Exceeded**.

However, when we are generally looking at efficiency in competitive coding, we are dealing with **time-based efficiency only**.

Note that efficiency generally does not come into picture at beginner-level problems, but as you advance, efficiency becomes ever more important, and **dictates the way in which you approach a problem**.

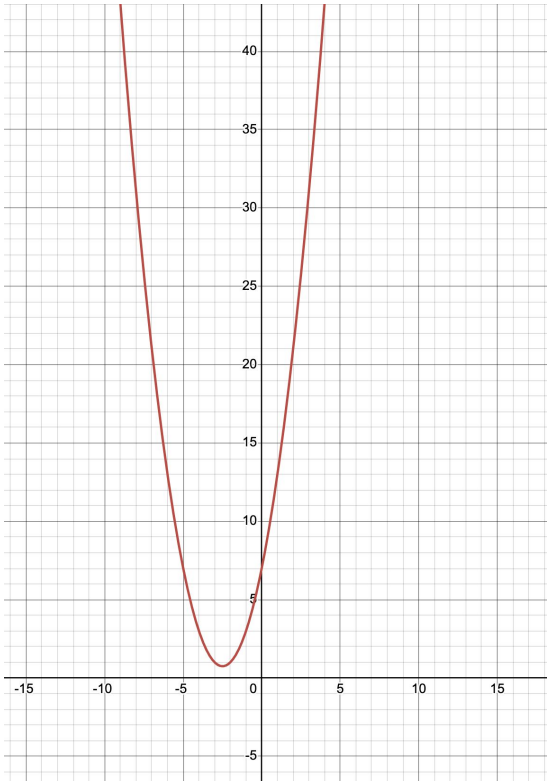
3.B: Complexity

Complexity, is in a sense, a measure of how efficient your code is.

Mathematically, it tells us how the time your code takes to run varies with the size of variables.

As an example, let us say that the time taken by your code to run, as a function of the input variable **n**, is **n2+5n+7**.

As you can see in the graph on the right, the time taken increases at a rapid rate.



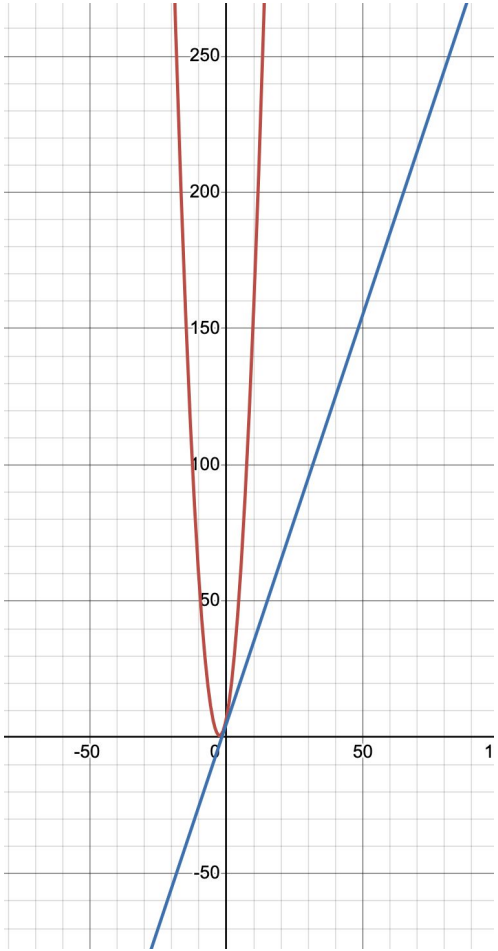
Now, let us say that a different program written to solve the same problem takes **3n+5** units of time.

As you can see, it does not grow as quickly as the previous function.

This is something you would expect, as n2+5n+7 is a **quadratic** function of n, whereas, 3n+5 is **linear**.

It seems like in general, functions which include higher powers of input variables grow faster than others.

And this is precisely how we define complexity - it is **the order of the fastest-growing term** in a function.



3.C: Big-O notation

Big-O notation is used to denote the complexity of a function.

For example, the complexity of **n2+5n+7** and **3n+5** would be **O(n2)** and **O(n)** respectively.

When we write big-O notation, we **ignore coefficients** and simply write the order of the fastest growing term.

Function Name	Big-O Notation
Constant	O(c)
Linear	O(n)
Quadratic	O(n^2)
Cubic	O(n^3)
Exponential	O(2^n)
Logarithmic	O(log(n))
Log Linear	O(nlog(n))

3.D: Program complexity

So far, you have learnt how to find the complexity of a mathematical function in terms of big-O notation.

Naturally, the next question is, given a program, how do I derive a function for the time it takes to run in the first place?

Time taken is **roughly proportional** to the **number of lines** processed in the execution of your program.

If there are 5 lines in a for loop that repeats 3 times, that would count as 15 lines.

To understand how to find the complexity of your program in depth, read this article by GFG.

3.E: Application of complexity

Now, you know how to find the complexity of a particular program.

Everything that was covered until now is stuff that you would have already learnt, if you've taken the DSA or FDSA course.

But what we will now cover is something new, and something strictly relevant to solving competitive coding problems.

Given a problem statement, how do you determine the required complexity of your solution code?

To do this, you have to observe the constraints on the problem's input variables, and match them with the below table.

Upper bound on input	**Required time complexity**
Greater than 10^6	O(1) or O(logn)
10^6	O(n)
2.10^5	O(nlogn)
5000	O(n2)
500	O(n3)
20	O(2^n)
10	O(n!)

As an exercise, try to determine the required complexity for the set of practice problems on the next slide, and the complexity of your solution program, as you solve them.

3.1: Codeforces 139A: Petr and Book

3.2: Codeforces 237A: Free Cash

3.3: Codeforces 1676D: X-Sum

3.4: Codeforces 550C: Divisibility by Eight

3.F: Sorting

Sorting is the act of rearranging a list of numbers in increasing or decreasing order.

You might have learnt the different approaches to sorting in DSA or FDSA.

However, in competitive coding, we are not concerned with the exact mechanics of sorting; all we are concerned with is its application.

All you need to know is how to use the **in-built C++ sort() function on a vector**, and the fact that it runs at **O(nlogn)** complexity, and you are good to go!

1. Basics of sorting
2. Sorting a vector of pairs

3.5: Codeforces 1760A: Medium Number

3.6: Codeforces 456A: Laptops

3.7: Codeforces 1742B: Increasing

3.8: Codeforces 1849B: Monsters

3.9: Codeforces 1857A: Assembly via Minimums

3.10: Codeforces 1760C: Advantage