Analysis of Algorithm

While writing an algorithm, our main concern is its performance and this is the entire point of designing an efficient algorithm. But how do we measure the performance? Let's jump into this chapter and find out.

Imagine a person writing a new algorithm and then checking its performance. He is running it in a computer and if the performance is not up to mark, he is modifying the code and again testing for the performance in a computer. One can't test each and every modification on a computer again and again, so there should be some way through which we can theoretically predict the performance of our algorithm while writing it and we are going to study the same in this chapter.

How to predict the performance of an algorithm?

To predict the performance of an algorithm, the best way is to follow the steps our computer is going to take in the process of executing the algorithm and predict the performance of each step. Still, this requires the knowledge of each step our computer is going to take and also the performance of each step. No doubt, our computer is a very complex machine and we can't predict the performance of these steps and hence the performance of our algorithm precisely.

The solution is to model our computer into something simpler using some set of rules and use this model to predict the performance. Generally, we use **RAM (Random Access Machine) model** to do so. The rules or the instructions of the RAM model are:

1. Arithmetic operations like addition(+), subtraction(-), multiplication(*), division(/), floor, ceiling, etc. take a constant amount of time.
2. Memory access like read, save, copy, etc. take a constant amount of time.
3. Subroutine calls, control, return, etc. take a constant amount of time.

This model is similar to that of a real computer but yet someone can think that multiplying two small numbers and two large number is not going to take an equal amount of time. Of course, this model is not going to give the exact precise time our algorithm is going to take. For a moment, forget about computers, even the same algorithm written in different languages are going to perform differently. But we are also not concerned with the exact performance, only the idea of whether an algorithm is going to take tens of second or thousands of second is quite an useful information to us and this model gives us the idea about the performance of our algorithm and does it very well.

Just think about high school Physics, we often ignore the airspeed, air drag, etc. while calculating the trajectory of a stone but yet we get an idea of the path the stone is going to follow. Similarly, we can deduce very useful information from our RAM model and thus analyze our algorithm.

Let's have a look at an example to see how to use this RAM model.

```
DEMO(A)
  for i in 1 to A.length
    for j in i to A.length
      x = a[i]+a[j]
```

Before moving further to the analysis of the performance, let's first look at the code written above.

Pseudocode

This type of code is called pseudocode and it used to represent the logic of any code in a language-independent way. We can easily implement these logics in any language of our choice. **However, codes in some of the languages like C/C++, Java and Python are also provided along with the pseudocode in this course**. We will be using the following conventions for pseudocode in this course:

- `for`, `if`, `else`, `return` and `while` are similar to most of the languages like C, Java, Python, etc.
- `//` is used for commenting.
- `and`, `or` & `not` are boolean operators and are similar to that of `&&`, `||` & `!` of C and Java respectively.
- We are using indentation to represent the body block.
- `floor()` and `ceiling()` are used for the floor and ceiling functions respectively.

  floor(2.5) = $\lfloor 2.5 \rfloor$ = 2

  ceiling(2.5) = $\lceil 2.5 \rceil$ = 3
- *DEMO* is the name of the function with the argument '*A*'.
- `==`, `!=`, `<`, `>`, `<=` and `>=` are used to compare values.
- `A[l,r]` means a subarray of A, indexed from 'l' to 'r'.

Explanation of the code

`DEMO(A)` - *DEMO* is the name of the function and *A* is the argument of the function which in this case is an array.

`for i in 1 to A.length` - We are iterating over the array A using a for loop. `A.length` is the length of the array *A*.

</>
For simplicity, through this entire course, we will assume that our array is starting from index 1 and not 0 and goes up to the length of the array and not length-1.

`for j in i to A.length` - *j* is iterating from the value of *i* to the length of the array *A* for each iteration of the outer loop. If the value of *i* is 2 then *j* will iterate from 2 to *A.length*.

</>
for i in 1 to A.length - Both 1 and A.length are inclusive here. It means that the iteration will include both the upper limit (A.length) and the lower limit (1) in the iteration.

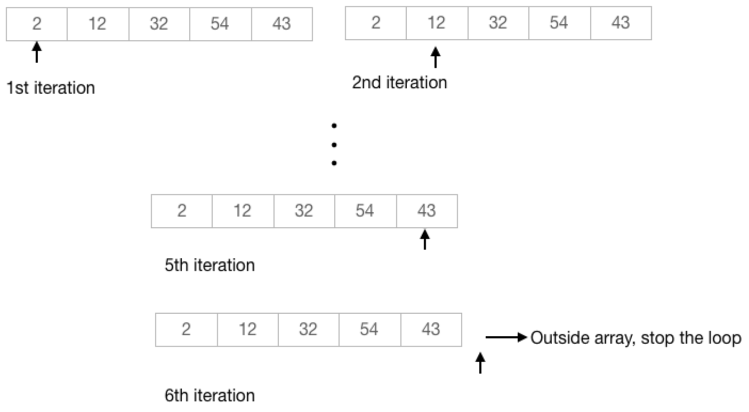
`x = a[i]+a[j]` - We are performing addition on the values of a[i] and a[j] and then copying the value to the variable x.

Let's alanyze our code

	Cost	Times
DEMO (A)		
for i in 1 to A.length	c_1	n+1
for j in i to A.length	c_2	$\sum(n-i+1)+1$
x = a[i]+a[j]	c_3	$\sum n-i+1$

Here, the cost represents the cost of running the statements a single time which according to the RAM model is a constant time operation. For example, to execute the statement `for i in 1 to A.length` for single time, c_1 amount of time will be taken.

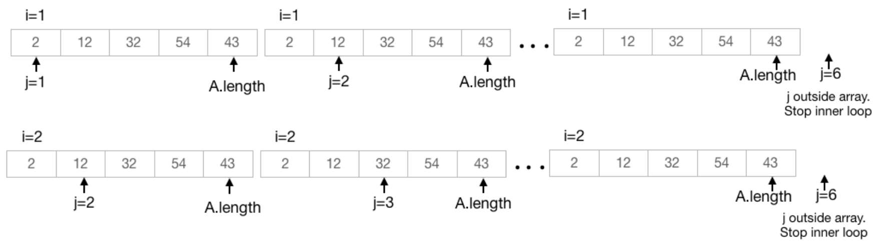
Now, the first loop runs **n+1** times (the length of the array is *n* and one more time when it will just check the condition of the loop and will fail). So, the **total cost** will be **c_1*(n+1)**.



For the outer loop,

$$cost\,(C_1) = c_1 * (n+1)$$

Also, for the first iteration of the outer loop, the inner loop will run *n+1* times (*i* is 1 in the first iteration, so j will go from 1 to A.length i.e., n times and one more time when it will check the condition of the loop and will fail). Similarly, *n* times for the second iteration of the outer loop, *n-1* for the third and so on. Thus, for the last iteration of the outer loop, the inner loop will run only single time (*i* is *A.length* i.e., *n* so, *j* will iterate only one time).



For the outer loop,

$$cost\,(C_2) = c_2 * [(n+1) + (n) + (n-1) + ... + 1] = c_2 * \frac{(n+1)(n+2)}{2}$$

</>

The sum of first n natural numbers (1 + 2 + ... + n) is $\frac{n*(n+1)}{2}$

Similarly, the last statement (`x = a[i]+a[j]`) would execute one time less than the number of times inner loop is executed because this statement is not going to run one extra time when the loop statement was just executed to check the condition and failed.

So for the last statement,

$$cost\,(C_3) = c_3 * [(n) + (n-1) + (n-2) + ... + 1] = c_3 * \frac{n*(n+1)}{2}$$

Thus, the total cost is the summation of the above three costs i.e.,

$$C = C_1 + C_2 + C_3$$

Putting the values of C_1, C_2 and C_3,

$$or,\ C = \left[(c_1 * (n+1)) + \left(c_2 * \frac{(n+1)(n+2)}{2}\right) + \left(c_3 * \frac{n*(n+1)}{2}\right)\right]$$

By solving the equation,

$$or,\ C = \left[(nc_1 + c_1) + \left(\frac{c_2n^2}{2} + \frac{3nc_2}{2} + c_2\right) + \left(\frac{c_3n^2}{2} + \frac{c_3n}{2}\right)\right]$$

$$or,\ C = n^2\left(\frac{c_2 + c_3}{2}\right) + n\left(c_1 + \frac{3c_2 + c_3}{2}\right) + (c_1 + c_2)$$

</>
We have tried to simplify each and every equation used in this course. So, just put your effort and you will be able to understand all the equations of the course.

So, we got a function which is quadratic with the input size *n*. And this is the way we are going to compare our algorithms. One could think of some other possibilities like comparing execution time or counting the number of statements executed to compare algorithms but our solution of expressing the running time as a function of input size n i.e., $f(n)$ is the most ideal solution to compare different algorithms. Think about comparing execution times, they would vary from computer to computer and also with languages. Also, the number of statements would vary with different languages and also with different programmers because of different writing styles. But our derived function (C) is independent of all these factors.

One thing to be noted here is that in this case, the input size *n* is the number of elements in the array *A* and this depends entirely on the problem we are solving. For example, for solving a graph, the most suitable inputs would be the vertices and the edges of the graph, the polynomial degree for equations, or number of elements in a matrix while dealing with a matrix, etc. So, the input will vary from problem to problem.

Let's look at one more example of deriving this function for another algorithm.

```
SUM(A)
  sum = 0
  for i in 1 to A.length
     sum = sum+A[i]
```

This is a simple function to calculate the sum of all elements of an array. Let's analyze this code.

	Cost	Times
SUM (A)		
sum = 0	c_1	1
for i in 1 to A.length	c_2	n+1
sum = sum+a[i]	c_3	n

`sum = 0` - This is going to run only one time and thus the total cost of this line would be c_1.

`for i in 1 to A.length` - As discussed in the previous example, this is going to run *n+1* times, *n* time as *i* will iterate from 1 to *A.length* and one more time when it will check the condition and fail. Thus, it will have a cost of $c_2(n+1)$.

`sum = sum+a[i]` - This line will be executed *n* times, one time in each iteration except when the loop will run one extra time to check the condition and thus will have a cost of c_3n.

Thus, the total cost would be

$$C = c_1 + c_2(n+1) + c_3n$$

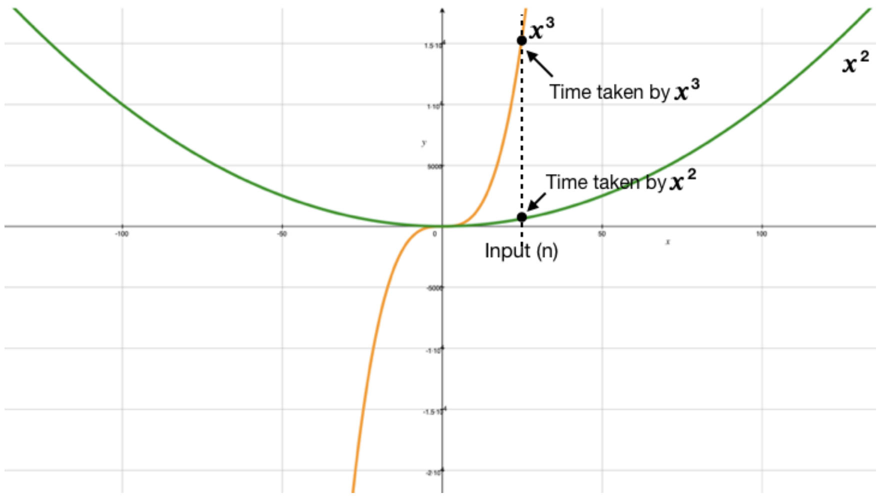
$$or,\ C = n(c_2 + c_3) + (c_1 + c_2)$$

As you can see, this time we have got a function which is linear. Also, you will see later in this course that we are only concerned about the order of these function and we also ignore the coefficients e.g., $c_2 + c_3$ and $c_1 + c_2$ in the above equation assuming the values of these coefficients are neither too high nor too low to make one term dominate over others and this assumption is also practically valid for most of the cases.

So, what is the significance of obtaining this function of cost?

The most important significance is that it represents the growth of the algorithm i.e., how the cost is going to increase with the increase in the size of the array. Also, we are not interested when the size is small because it doesn't matter whether our computer is taking 0.04s or 0.045s but the things get really prominent when the size of the input increases and then the rate of the growth matters. For example, let's take a hypothetical case where an algorithm on a computer takes 0.000001s for the input size of 1. And suppose the algorithm has a function of order n^2, then for the input size of 1,000,000 (1 million), it is going to take $1,000,000^2 * 0.000001s$ i.e., around 11 days but with a function which is linear i.e., have an order of n, it will only take $1,000,000 * 0.000001$ i.e., 1s.

So, now you have the idea of how the order of the function is going to affect the performance of an algorithm. Just imagine if the order of an algorithm is cubic and how worse it would perform with an input of large size and this is the reason why while studying basic programming languages, we are told to avoid writing codes which requires nesting of three loops because it will have a cubic order. So, the entire point of the discussion was that we are concerned about the rate of the growth i.e., the order of the function because it is what going to matter.



This is the graphical representation of how functions with different orders grow with n. Also you can see that for $n < 1$, the function n^2 is taking less time than that of n but as told earlier, with this small input size, the difference in the performance is not considerable and we are concerned about the input of large sizes.

So this chapter gave you an idea of basic analysis of an algorithm and how important the rate of the growth is, the next chapter (/course/algorithms-growth-of-a-function/) discusses the growth functions in details with also the notations we generally use.

“ When something is important enough, you do it even if the odds are not in your favour. ”

- Elon Musk

PREV (/course/algorithms-introduction/) (/course/algorithms-growth-of-a-function/) NEXT