CS204: Design and Analysis of Algorithms

220001001, 220001002, 220001003

January 12, 2024

1 Introduction

Analysis of algorithms includes - time and space analysis.

In today's date, we cannot increase the processor speed. Thus time is a limiting parameter. So we focus more on time complexity than space.

2 Time Complexity

2.1 Ways we can measure time complexity

- We can measure processing time in seconds or milliseconds, but this won't be a fair measure because one person may have an i5 processor and another one can have an i9 processor. This might give the wrong measure of the efficiency of the program.
- Another method would be to count the actual number of steps in the algorithm. For example, in sorting for 10 elements we will need $\binom{10}{2}$, and for sorting of 100 elements we will need $\binom{100}{2}$ comparisons.
- This step is a function of input size: $step = f(I|p\ size)$. But also, the number of steps may vary for sorting two arrays of the same size, so it can be a little tricky to measure the exact number of steps.
- Thus we analyze the worst-case time.

2.2 Asymptotic time complexity

- When we analyze time complexity we are NOT going to look at ALL the steps.
- We will see asymptotic time complexity: The complexity of the algorithm depends on input size. Assignment and Comparison steps do not much affect the time.

2.3 Example

Consider the following algorithm to find the maximum element of an array A:

```
max = A[0]
i = 1
while i < A.length
    if A[i] > max
        max = A[i]
```

Here there are three types of steps: initialization steps ($max = A[0], i = 1$), assignment ($max = [i]$) and comparison ($i < A.length$), these are constant time operations, and taking into account the whole algorithm these can be ignored when compared to a much larger time-taking steps like the while loop. This measure of operational time is called **asymptotic time complexity** where constant time operations can be ignored.

In the above code, number of steps can be approximated as:
$2 + n + n + \mathbf{n}$

'2' for first two steps, **first n** for checking while loop condition, **second n** for checking if condition inside the while loop and **last n** for is for worst-case time complexity for the assignment step. For the best case, it would be just 1 instead of n.
$\therefore$ We can write it as $3n + 2 \leq 4n$
Here n is the dominating factor thus as n $\uparrow$ time complexity $\uparrow$.

NOTE: In asymptotic analysis, we ignore the constant part and the coefficient.

FUN FACTS:

- Current CPU capacity is 10^8 instructions per second (or ins/sec).
- For sorting 10^9 elements:
  - For the algorithm having number of steps proportional to n^2 we need 10^{18} steps
    Therefore, time $\rightarrow \frac{10^{18}}{10^8} = 10^{10} \equiv 300$ years!!
  - For the algorithm having number of steps proportional to $nlogn$ we need $10^9 \log(10^9) \equiv 10^9 \log((10^3)^3) \equiv 3 \times 10^9 \log(2^{10}) = 3 \times 10^{10}$ steps
    Therefore, time $\rightarrow \frac{3 \times 10^{10}}{10^8} = 300$ sec

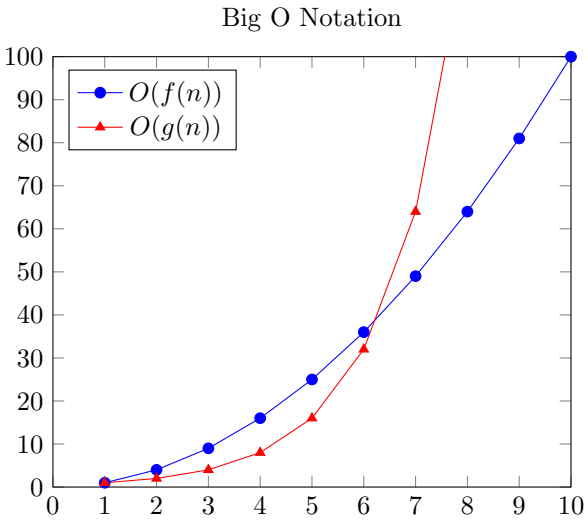
2.4 Some common time complexities and their comparisons:

- $O(1) \rightarrow$ constant time complexity: The time complexity, or the number of steps, does not depend upon the size of the input.
- $\log n \rightarrow$ logarithmic time complexity
  - steps $\propto \log n$
  - binary search
  - finding no. of digits in a number
- Polynomial complexities:
  $\sqrt{n},\ n,\ n^2,\ n^3$ .... n^k
- Exponential complexities:
  $2^n,\ n!$

input	$\log(n)$	$\sqrt{n}$	n	n^2	2^n	$n!$
10	3	3	10	100	1000	1000000
10^2	6	10	100	10000	10^{30}	10^{157}
10^3	9	30	1000	10^6	-	-
10^4	12	100	10000	10^8	-	-
10^5	15	300	100000	10^{10}	-	-
10^6	18	10^3	10^6	10^{12}	-	-
10^7	21	3×10^3	10^7	10^{14}	-	-
10^8	24	10^4	10^8	10^{16}	-	-
10^9	27	3×10^4	10^9	10^{18}	-	-

2.5 The Big O notation

The Big O notation provides an upper bound on the growth rate of an algorithm. It gives us an idea about the worst-case scenario of an algorithm.



$$O(g(n)) = \{f(n) \mid \exists c_1 \in \mathbb{R}^+, n_0 \in \mathbb{Z}^+ \text{ such that } 0 \leq f(n) \leq g(n) \text{ for all } n \geq n_0\}$$

Example 1: $f(n) = n + 3\log(n) + 5$

$$0 < n + 3\log(n) + 5 < c \cdot n$$

Now we can say:

$$n + 3\log(n) + 5 \leq n + 3n + 5n \text{ for } n \geq 1$$

$$n + 3\log(n) + 5 \leq 9n$$

Hence, we can conclude that $n_0 = 1$ and $c = 9$.

Example 2: $f(n) = 3n + 9$

$$3n + 9 \leq 4 \cdot n \text{ for all } n \geq n_0$$

Hence, $c = 4$.

$$9 \leq 4n - 3n$$

$$9 \leq n$$

Hence, we can conclude that $n_0 = 9$ and the value of $c = 4$.

Worst Case Time complexity is not a good way to analyze because it occurs once in a while. The better way is to use the average time complexity but it is not possible every time since it's not always possible to know the nature of input and hence we cannot find the time complexity. Thus for such types of algorithms, it would be better to say that the time complexity of it lies between some values.