Analysis of algorithm.

The analysis is a process of estimating the efficiency of an algorithm. There are fundamental parameters based on which we can analysis the algorithm.

→ Time Complexity — Time complexity is a function of input size n that refers to the amount of time needed by an algorithm to run to completion.

→ Space Complexity — The space complexity can be understood as the amount of ~~time needed~~ space required by an algorithm to run to completion.

We will be focusing more on time rather than space because time is instead a more limiting parameter in terms of the h/w. It is not easy to take a computer and changed its speed. Generally, we make three types of analysis, which is as follows

→ Worst case time complexity —

For 'n' input size, the worst-case time complexity can be defined as the maximum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the max^m number of steps performed on an instance having an input size of n.

→ Average-case time complexity —

For 'n' input size, the average-case time complexity can be defined as the average amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the average number of steps performed on an instance having an input size of n.

→ Best case time complexity —

For 'n' input size, the best case time complexity can be defined as the min^m amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the minimum no. of steps performed on an instance having an input size of n.

* Asymptotic means approching a value or curve arbitrarily closely.

☐ + ☐ = ☐

Asymptotic Notation.

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time time taken by a programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

1. Θ-(theta) notation
2. Big O notation
3. Ω (omega) notation.

1. Θ - notation.

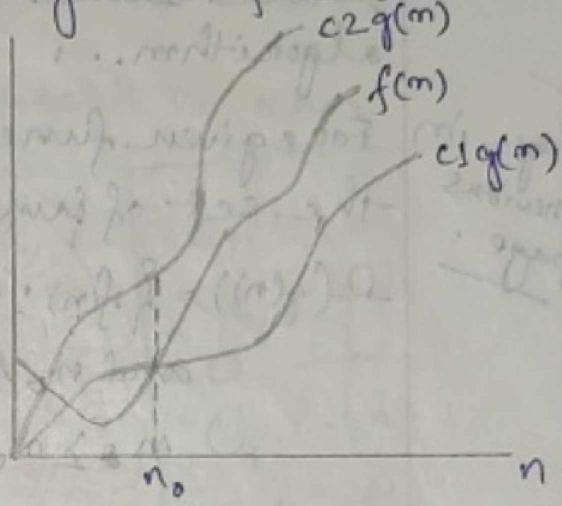
The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.

For a given function $g(n)$, we denote $\Theta(g(n))$ is following set of functions.

$\Theta(g(n)) = \{ f(n)$: there exist positive constants $c1$, $c2$ and n_0 such that.

$0 <= c1 * g(n) <= f(n) <= c2 * g(n), \ \forall \ n \geq n_0 \}$

The above defination means, if f(n) is theta of g(n), then the value f(n) is always betn c1 * g(n) and c2 * g(n) for large values of n. The defination of theta also requires that f(n) must be non-negative for values of n greater than n_0.



$f(n) = \Theta(g(n))$

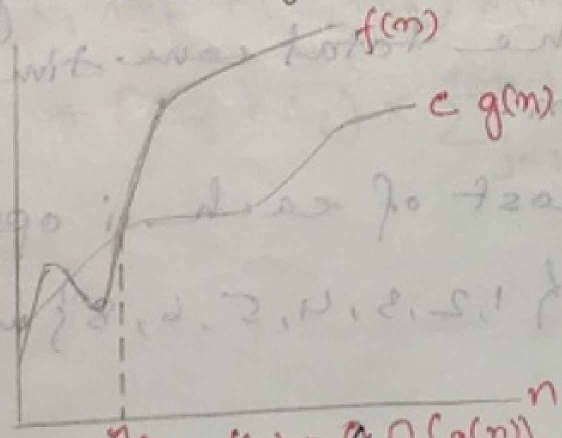
2. Big O notation

The big O notation defines an upper bound of an algorithm, it bounds a function only from above.

The Big-O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

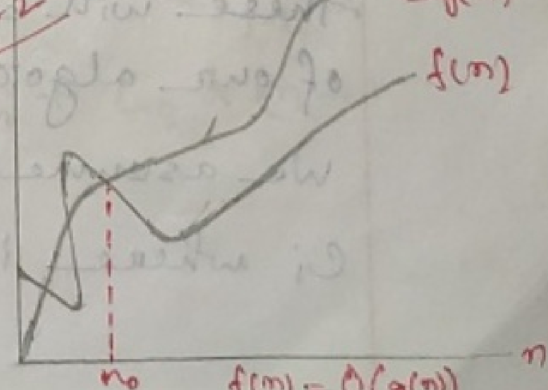
$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c.g(n) \ \forall \ n \geq n_0 \}$

fig-3



$f(n) = \cancel{\Theta} \ \Omega(g(n))$

fig-2



$f(n) = O(g(n))$

3. Ω-notation

Just as Big-O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω notation can be useful when we have lower bound on time complexity of an algorithm.

fig. in previous page.

For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of function.

$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c\,g(n) \leq f(n) \text{ for all } n \geq n_0 \}$$

Insertion Sort analysis

Running time for any algorithm depends on the number of operations executed. We could see in the Pseudocode that there are precisely 7 operations under this algorithm. So, our task is to find the cost or Time complexity of each and trivially sum of these will be the total ~~com~~ time complexity of our algorithm.

We assume cost of each i operation as c_i where $i \in \{1, 2, 3, 4, 5, 6, 8\}$ and compute the no. of times these are executed. Therefore the total cost for one such operation would be the product of cost of one operation and the no. of times it is executed.

We could list them below.

cost of line	No. of Times it is run.
c_1	n
c_2	$n-1$
c_3	$n-1$
c_4	$\sum_{j=1}^{n-1} t_j$
c_5	$\sum_{j=1}^{n-1} t_j - 1$
c_6	$\sum_{j=1}^{n-1} t_j - 1$
c_8	$n-1$

c_1 – n
c_2 – $n-1$
0 – $n-1$
c_4 – $n-1$
c_5
c_6
c_7 $\quad c_7 = 0$
c_8

Then total Running time of Insertion sort

$$T(n) = c_1 \times n + (c_2 + c_3) \times (n-1) + c_4 \times \sum_{j=1}^{n-1} t_j + (c_5 + c_6) \times \sum_{j=1}^{n-1} t_j - 1 + c_8 \times (n-1)$$

Best Case analysis

In Best case i.e, when the array is already sorted, $t_j = 1$.

Therefore

$$T(n) = C_1 \times n + (C_2 + C_3) \times (n-1) + C_4 \times (n-1) + (C_5 + C_6) \times (n-2) + C_8 \times (n-1)$$

which when further simplified has dominating factor of n and gives

$$T(n) = C \times (n) \quad \text{or} \quad O(n)$$

Worst Case Analysis

In worst case i.e., when the array is reversely sorted, $t_j = j$

Therefore

$$T(n) = C_1 \times n + (C_2 + C_3) \times (n-1) + C_4 \times \frac{(n-1)n}{2} + (C_5 + C_6) \times \left(\frac{n(n-1)}{2} - 1\right) + C_8 \times (n-1)$$

which when further simplified has dominating factor of n^2 and gives

$$T(n) = C \times (n^2) \quad \text{or} \quad O(n^2).$$

insertion sort analysis

Average case analysis.

Let's assume that $t_j = \frac{j-1}{2}$ to calculate the average case.

Therefore

$$T(n) = c_1 \times n + (c_2 + c_3) \times (n-1) + \frac{c_4}{2} \times \frac{n(n-1)}{2} + \frac{(c_5 + c_6)}{2} \times \left(\frac{n(n-1)}{2} - 1\right) + c_8 \times (n-1)$$

which when further simplified has dominating factor of n^2 and gives

$$T(n) = c \times (n^2) \text{ or } O(n^2).$$

Selection sort

Algorithm	Cost	Time
$n \leftarrow$ length[A]	c_1	1
for $j \leftarrow 1$ to $n-1$	c_2	$n-1$
do smallest $\leftarrow j$	c_3	$n-1$
for $i \leftarrow j+1$ to n	c_4	$\sum_{j=1}^{n-1} (n-j+1)$
$\approx \frac{n^2}{2}$ comparisons, do if $A[i] < A[\text{smallest}]$	c_5	$\sum_{j=1}^{n-1} (n-j)$
then smallest $\leftarrow i$	c_6	$\sum_{j=1}^{n-1} (n-j)$
$\approx n$ exchanges, exchange $A[j] \leftrightarrow A[\text{smallest}]$	c_7	$n-1$

Total running time

$$T(n) = (c_1 * 1) + (c_2 + c_3) \times (n-1) + c_4 \times \sum_{j=1}^{n-1} (n-j+1)$$
$$+ (c_5 + c_6) \times \sum_{j=1}^{n-1} (n-j) + c_7 \times (n-1)$$

which when further simplified has dominating factor of n^2 and gives

$$T(n) = c \times (n^2) \text{ or } O(n^2)$$

— · —

Analyzing divide-and conquer algorithm.

When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation or recurrence, which describes the ~~total~~ overall running time on a problem of size n in terms of the running time on smaller inputs.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. We let $T(n)$ be the running time on a problem of size n. If the problem size is small enough, say $n \leq c$ for some constant c, the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields 'a' subproblems, each of which is $1/b$ the size of the original. (For merge sort, both a and b are 2.). It takes time $T(n/b)$ to solve one problem of size n/b, and so it takes time $aT(n/b)$ to solve a of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solution to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

Merge Sort Analysis

Although the pseudocode of Merge-Sort works correctly when the no. of elements is not even, our recurrence based analysis is simplified if we assume that the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly $n/2$.

We reason as follows to set up recurrence for $T(n)$, the worst-case running time of merge-sort on n nos. Merge-sort on just one element takes constant time. When we have $n>1$ elements, we break down the running time as follows

- Divide — The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = O(1)$

- Conquer — We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- Combine — We have already noted that the Merge procedure on an n-element subarray takes time $O(n)$, and so $C(n) = O(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $O(n)$ and a function that is $O(1)$. This sum is a linear function of n, that is, $O(n)$. Adding it to the $2T(n/2)$ term from the "conquer" steps gives the recurrence for the worst-case running time $T(n)$ of merge sort.

5

$$\therefore \quad T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1. \end{cases}$$

Using master theorem the solution to the recurrence is $T(n) = O(n \log n)$

Quick Sort Analysis.

In quick sort, we pick an element called the pivot in each steps and re-arrange the array in such a way that all elements less than the pivot now appear to the left of the pivot, and all elements larger than the pivot appear on the right side of the pivot. In all subsequent iterations of the sorting algorithm the position of this pivot will remain unchanged, because it has been put in its correct place. The total time taken to re-arrange the array as just described, is always $O(n)$, or αn where α is some constant. Let us suppose that the pivot we just chose has divided the array into two parts - one of size k and the other of size $n-k$. Notice that both these parts still need to be sorted. This gives us the following relation:

$$T(n) = T(k) + T(n-k) + \alpha n .$$

Worst case analysis.

Now consider the case, when the pivot happened to be the least element of the array, so that we had $k=1$ and $n-k=n-1$. In such a case, we have

$$T(n) = T(1) + T(n-1) + \alpha n$$

Substituting $T(n-1)$ by $[T(n-2) + T(1) + \alpha(n-1)]$

Now let us analyse the time complexity of quick sort in such a case in detail by solving the recurrence as follows -

$$T(n) = T(n-1) + T(1) + \alpha n$$
$$= [T(n-2) + T(1) + \alpha(n-1)] + T(1) + \alpha n$$
$$= T(n-2) + 2T(1) + \alpha(n-1+n)$$ (by simplifying & grouping terms together)
$$= [T(n-3) + T(1) + \alpha(n-2)] + 2T(1) + \alpha(n-1+n)$$
$$= T(n-3) + 3T(1) + \alpha(n-2+n-1+n)$$
$$= T(n-i) + iT(1) + \alpha(n-i+1+ \dots + n-2+n-1+n)$$

(counting likewise till the i^{th} steps.)

$$= T(n-i) + iT(1) + \alpha\left(\sum_{j=0}^{i-1}(n-j)\right)$$

Now ~~careful~~ clearly such a recurrence can only go on until $i = n-1$. So, substitute $i = n-1$ in the above equation, which gives us:

$$T(n) = T(1) + (n-1)T(1) + \alpha\sum_{j=0}^{n-2}(n-j)$$
$$= nT(1) + \alpha\left(n(n-2) - (n-2)(n-1)/2\right)$$
$$= O(n^2)$$

This is the worst case of quick sort, which happens when the pivot we picked turns out to be the least element of the array to be sorted, in every step.