Q: Define the concept of and algorithm discuss the time analysis and space analysis of algorithms .

What is an Algorithm ?

The concept of an algorithm is basically it solve complex problems. An algorithm is a finite list of instructions, most often used in solving problems or performing tasks. You may have heard the term used in some fancy context about a genius using an algorithm to do something highly complex, usually in programming. Indeed, you've most likely heard the term used to explain most things related to computer processes. However, what would you say if I was to tell you that there is a very good chance that you, yourself, have followed an algorithm? You may have followed some algorithms hundreds or thousands of times.

A Real Life Algorithm

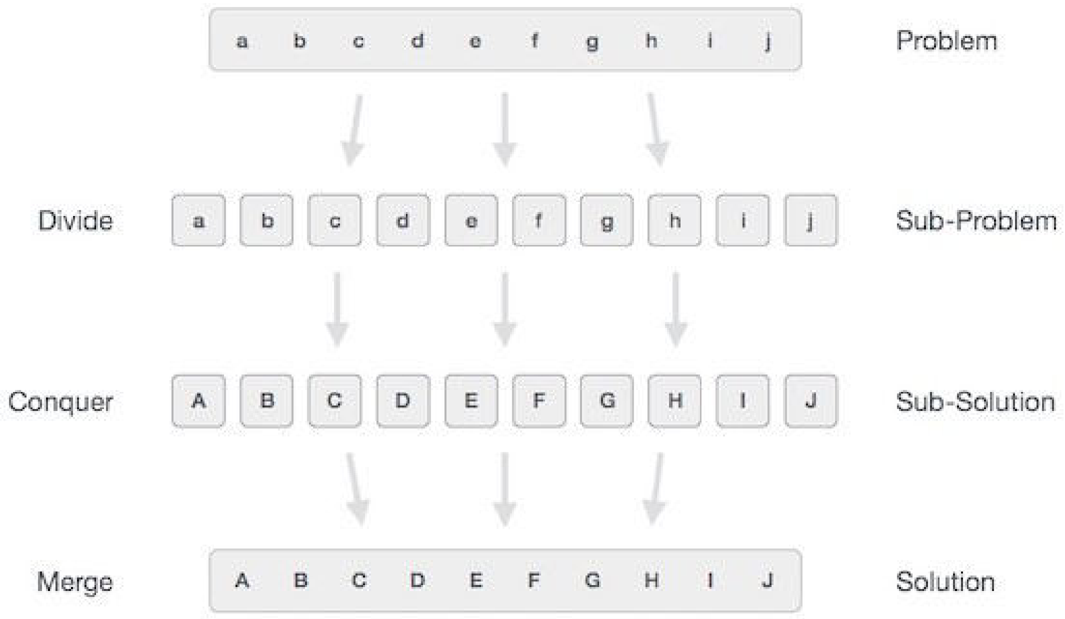
Have you ever baked or cooked something? One of the most obvious examples of an algorithm is a recipe. It's a finite list of instructions used to perform a task. For example, if you were to follow the algorithm to create brownies from a box mix, you would follow the three to five step process written on the back of the box.

Time analysis and space analysis of algorithms

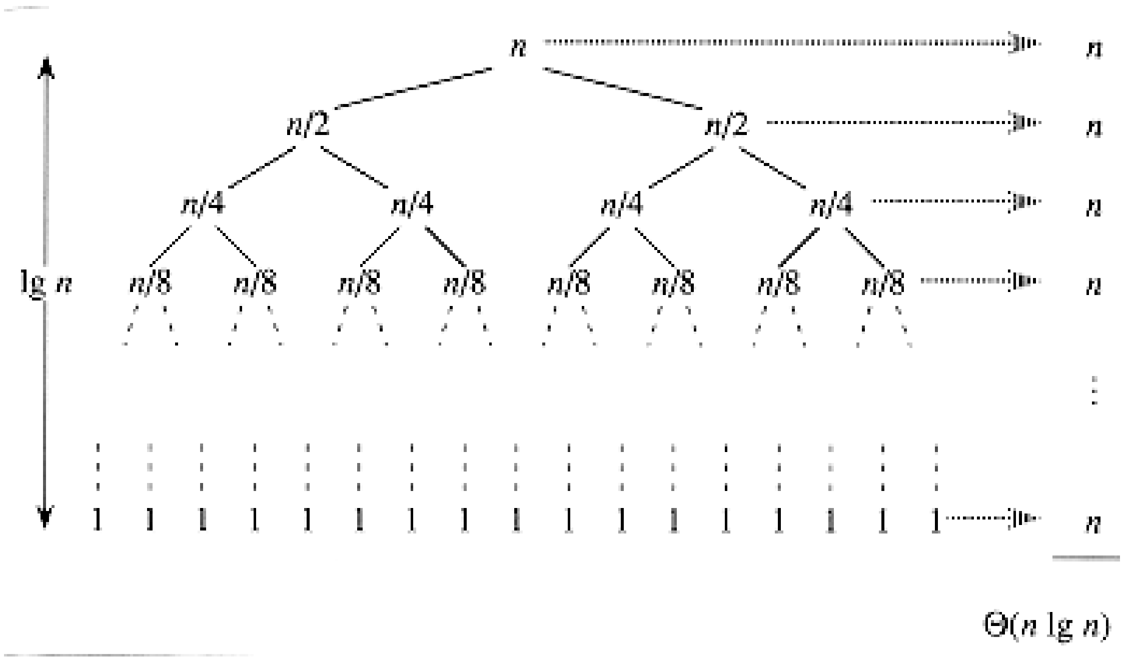
Time complexity is a function describing the amount
of time an algorithm takes in terms of the amount of input to the algorithm.
Space complexity is a function describing the amount of memory (space)
an algorithm takes in terms of the amount of input to the algorithm.

Q: Explain the Divide and conquer Approach . Draw the recurrence tree to analyze the divide and conquer Approach.

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub problems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



I am taking Quicksort algorithm as a Example for Recurrence tree .



There are 4 comparison to find the element which is actually suitable time complexity.

Log(n) .

Q : Discuss the asymptotic notations with the help of worst case , average case and best case.

Asymptotic analysis of an algorithm refers to defining the mathematical foundation / framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Usually, the time required by an algorithm falls under three types −

- Best Case − Minimum time required for program execution.
- Average Case − Average time required for program execution.
- Worst Case − Maximum time required for program execution.

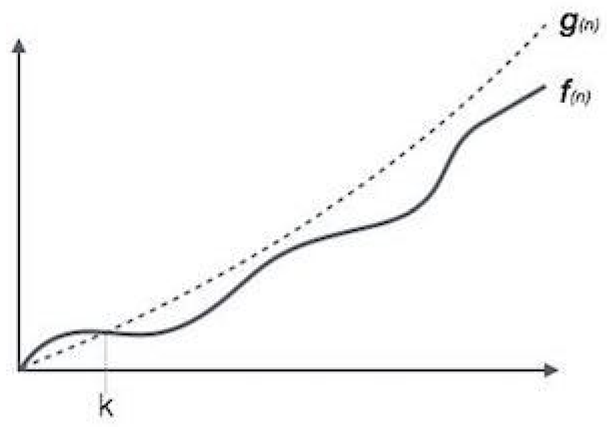
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

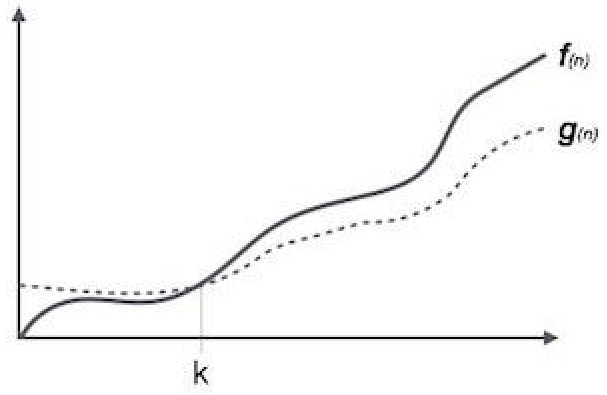
Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



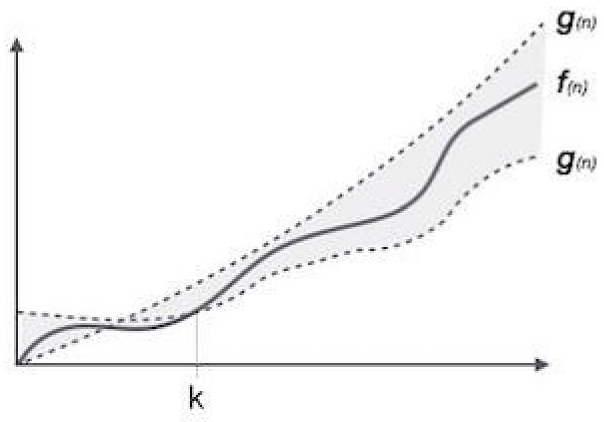
Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



Common asymptotic notations.

constant	−	$O(1)$
logarithmic	−	$O(\log n)$
linear	−	$O(n)$
n log n	−	$O(n \log n)$
quadratic	−	$O(n^2)$
cubic	−	$O(n^3)$
polynomial	−	$n^{O(1)}$
exponential	−	$2^{O(n)}$

Q:Solve the recurrence equation using substitution method and master method

The substitution method is a condensed way of proving an asymptotic bound on a recurrence by induction. In the substitution method, instead of trying to find an exact closed-form solution, we only try to find a closed-form bound on the recurrence. This is often much easier than finding a full closed-form solution, as there is much greater leeway in dealing with constants.

The substitution method is a powerful approach that is able to prove upper bounds for almost all recurrences. for certain types of recurrences, the master method (see below) can be used to derive a tight bound with less work. In those cases, it is better to simply use the master method, and to save the substitution method for recurrences that actually need its full power.

Note that the substitution method still requires the use of induction. The induction will always be of the same basic form, but it is still important to state the property you are trying to prove, split into one or more base cases and the inductive case, and note when the inductive hypothesis is being used.

Substitution method example

Consider the following reccurence relation, which shows up fairly frequently for some types of algorithms:

$T(1) = 1$
$T(n) = 2T(n-1) + c_1$

By expanding this out a bit (using the "iteration method"), we can guess that this will be $O(2^n)$. To use the substitution method to prove this bound, we now need to guess a closed-form upper bound based on this asymptotic bound. We will guess an upper bound of $k2^n - b$, where b is some constant. We include the b in anticipation of having to deal with the constant c_1 that appears in the recurrence relation, and because it does no harm. In the process of proving this bound by induction, we will generate a set of constraints on k and b, and if b turns out to be unnecessary, we will be able to set it to whatever we want at the end.

Our property, then, is $T(n) \leq k2^n - b$, for some two constants k and b. Note that this property logically implies that $T(n)$ is $O(2^n)$, which can be verified with reference to the definition of O.

Base case: $n = 1$. $T(1) = 1 \leq k2^1 - b = 2k - b$. This is true as long as $k \geq (b + 1)/2$.

Inductive case: We assume our property is true for $n - 1$. We now want to show that it is true for n.

$$
\begin{aligned}
T(n) &= 2T(n-1) + c_1 \\
&\leq 2(k2^{n-1} - b) + c_1 \quad \text{(by IH)} \\
&= k2^n - 2b + c_1 \\
&\leq k2^n - b
\end{aligned}
$$

This is true as long as $b \geq c_1$.

So we end up with two constraints that need to be satisfied for this proof to work, and we can satisfy them simply by letting $b = c_1$ and $k = (b + 1)/2$, which is always possible, as the definition of O allows us to choose any constant. Therefore, we have proved that our property is true, and so $T(n)$ is $O(2^n)$.

The biggest thing worth noting about this proof is the importance of adding additional terms to the upper bound we assume. In almost all cases in which the recurrence has constants or lower-order terms, it will be necessary to have additional terms in the upper bound to "cancel out" the constants or lower-order terms. Without the right additional terms, the inductive case of the proof will get stuck in the middle, or generate an impossible constraint; this is a signal to go back to your upper bound and determine what else needs to be added to it that will allow the proof to proceed without causing the bound to change in asymptotic terms.

MASTER METHOD

The master method gives us a quick way to find solutions to recurrence relations of the form T(n) = aT(n/b) + h(n), where a and b are constants, a ≥ 1 and b > 1. Conceptually, a represents how many recursive calls are made, b represents the factor by which the work is reduced in each recursive call, and h(n) represents how much work is done by each call apart from the recursion, as a function of n.

Once we have a recurrence relation of that form, the master method tells us the solution based on the relation between a, b, and h(n), as follows:

Case 1: h(n) is O(nlogba − ε), which says that h grows more slowly than the number of leaves. In this case, the total work is dominated by the work done at the leaves, so T(n) is Θ(nlogb a).

Case 2: h(n) is Θ(nlogba), which says that h grows at the same rate as the number of leaves. In this case, T(n) is Θ(nlogb a log n).

Case 3: h(n) is Ω(nlogba + ε), which says that h grows faster than the number of leaves. For the upper bound, we also need an extra smoothness condition on f, namely ah(n/b) ≤ ch(n) for some c < 1 and large n. In this case, the total work is dominated by the work done at the root, so T(n) is Θ(h(n)).

Examples for the master method

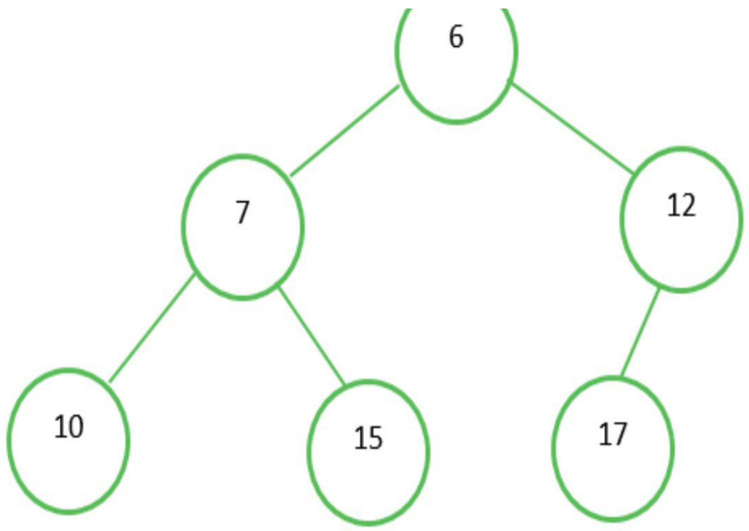
Example 1: Say you have derived the recurrence relation $T(n) = 8T(n/2) + cn^2$, where c is some positive constant. We see that this has the appropriate form for applying the master method, and that a=8, b=2, and $\mathrm{h(n)} = cn^2$. cn^2 is $O(n^{\log_2 8 - \varepsilon}) = O(n^{3-\varepsilon})$ for any $\varepsilon \leq 1$, so this falls into case 1. Therefore, $T(n)$ is $\Theta(n^3)$.

Example 2: Say you have derived the recurrence relation $T(n) = T(n/2) + cn$, where c is some positive constant. We see that this has the appropriate form for applying the master method, and that a=1, b=2, and $\mathrm{h(n)} = cn$. Then $h(n)$ is $\Omega(n^{\log_2 1 + \varepsilon}) = \Omega(n^{\varepsilon})$ for any $\varepsilon \leq 1$, so this falls into case 3. And $ah(n/b) = cn/2 = ½h(n)$, therefore $T(n)$ is $\Theta(n)$.

Example 3: Say you have derived the recurrence relation $T(n) = 8T(n/4) + cn^{3/2}$, where c is some positive constant. We see that this has the appropriate form for applying the master method, and that a=8, b=4, and $\mathrm{h(n)} = cn^{3/2}$. $cn^{3/2}$ is $\Theta(n^{\log_4 8}) = \Theta(n^{3/2})$, so this falls into case 2. Therefore, $T(n)$ is $\Theta(n^{3/2}\log n)$.

Q: Explain the concept of heap sorting using min heap of an array and define the insertion and deletion.

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements



DELETION IN HEAP

Given a Binary Heap and an element present in the given Heap. The task is to delete an element from this Heap.

Process of Deletion:

Since deleting an element at any intermediary position in the heap can be costly, so we can simply replace the element to be deleted by the last element and delete the last element of the Heap.

- Replace the root or element to be deleted by the last element.
- Delete the last element from the Heap.

•Since, the last element is now placed at the position of the root node. So, it may not follow the heap property.

Therefore, heapify the last node placed at the position of root.

Suppose the Heap is a Max-Heap as:

```
      10

    /    \

   5      3

  / \

 2   4
```

The element to be deleted is root, i.e. 10.

```
Process:
The last element is 4.


Step 1: Replace the last element with root, and delete
it.
      4

    /    \

   5      3

  /

 2


Step 2: Heapify root.
Final Heap:

      5
      /     \
     4       3
    /
   2
```

INSERTION IN HEAP

The insertion operation is also similar to that of the deletion process.

> *Given a Binary Heap and a new element to be added to this Heap. The task is to insert the new element to the Heap maintaining the properties of Heap.*

Process of Insertion: Elements can be inserted to the heap following a similar approach as discussed above for deletion. The idea is to:

- First increase the heap size by 1, so that it can store the new element.
- Insert the new element at the end of the Heap.
- This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap, heapify this newly inserted element following a bottom-up approach.

```
Suppose the Heap is a Max-Heap as:
      10
    /    \
   5      3
  / \
 2   4


The new element to be inserted is 15.

Process:
Step 1: Insert the new element at the end.
      10

    /    \

   5      3

  / \    /

 2   4  15


Step 2: Heapify the new element following bottom-up
         approach.

-> 15 is more than its parent 3, swap them.

       10

    /    \

   5      15

  / \    /

 2   4  3


-> 15 is again more than its parent 10, swap them.

       15

    /    \

   5      10

  / \    /
 2   4  3
```

Therefore, the final heap after insertion is:

```
        15
     /     \
    5       10
   / \     /
 2   4  3
```