DAA Unit-1

CSE 4th(SEM)

DAA Algorithm

The word algorithm has been derived from the Persian author's name, Abu Ja 'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who has written a textbook on Mathematics. The word is taken based on providing a special significance in computer science. The algorithm is understood as a method that can be utilized by the computer as when required to provide solutions to a particular problem.

An algorithm can be defined as a finite set of steps, which has to be followed while carrying out a particular problem. It is nothing but a process of executing actions step by step.

An algorithm is a distinct computational procedure that takes input as a set of values and results in the output as a set of values by solving the problem. More precisely, an algorithm is correct, if, for each input instance, it gets the correct output and gets terminated.

An algorithm unravels the computational problems to output the desired result. An algorithm can be described by incorporating a natural language such as English, Computer language, or a hardware language.

Characteristics of Algorithms

- **Input:** It should externally supply zero or more quantities.
- **Output:** It results in at least one quantity.
- **Definiteness:** Each instruction should be clear and ambiguous.
- **Finiteness:** An algorithm should terminate after executing a finite number of steps.
- **Effectiveness:** Every instruction should be fundamental to be carried out, in principle, by a person using only pen and paper.
- **Feasible:** It must be feasible enough to produce each instruction.
- **Flexibility:** It must be flexible enough to carry out desired changes with no efforts.

- **Efficient:** The term efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space meeting the acceptable limit of development time.
- **Independent:** An algorithm must be language independent, which means that it should mainly focus on the input and the procedure required to derive the output instead of depending upon the language.

Advantages of an Algorithm

- **Effective Communication:** Since it is written in a natural language like English, it becomes easy to understand the step-by-step delineation of a solution to any particular problem.
- **Easy Debugging:** A well-designed algorithm facilitates easy debugging to detect the logical errors that occurred inside the program.
- **Easy and Efficient Coding:** An algorithm is nothing but a blueprint of a program that helps develop a program.
- **Independent of Programming Language:** Since it is a language-independent, it can be easily coded by incorporating any high-level language.

Disadvantages of an Algorithm

- Developing algorithms for complex problems would be time-consuming and difficult to understand.
- It is a challenging task to understand complex logic through algorithms.

Pseudocode

Pseudocode refers to an informal high-level description of the operating principle of a computer program or other algorithm. It uses structural conventions of a standard programming language intended for human reading rather than the machine reading.

Advantages of Pseudocode

- Since it is similar to a programming language, it can be quickly transformed into the actual programming language than a flowchart.

- The layman can easily understand it.
- Easily modifiable as compared to the flowcharts.
- Its implementation is beneficial for structured, designed elements.
- It can easily detect an error before transforming it into a code.

Disadvantages of Pseudocode

- Since it does not incorporate any standardized style or format, it can vary from one company to another.
- Error possibility is higher while transforming into a code.
- It may require a tool for extracting out the Pseudocode and facilitate drawing flowcharts.
- It does not depict the design.

Difference between Algorithm and the Pseudocode

An algorithm is simply a problem-solving process, which is used not only in computer science to write a program but also in our day to day life. It is nothing but a series of instructions to solve a problem or get to the problem's solution. It not only helps in simplifying the problem but also to have a better understanding of it.

However, Pseudocode is a way of writing an algorithm. Programmers can use informal, simple language to write pseudocode without following any strict syntax. It encompasses semi-mathematical statements.

Problem: Suppose there are 60 students in the class. How will you calculate the number of absentees in the class?

Pseudo Approach:

1. Initialize a variable called as **Count** to zero, **absent** to zero, **total** to 60
2. FOR EACH Student PRESENT DO the following:
   Increase the **Count** by One
3. Then Subtract **Count** from **total** and store the result in **absent**
4. Display the number of absent students

Algorithmic Approach:

1. Count <- 0, absent <- 0, total <- 60

2. REPEAT till all students counted
   Count <- Count + 1
3. absent <- total - Count
4. Print "Number absent is:" , absent

Need of Algorithm

1. To understand the basic idea of the problem.

2. To find an approach to solve the problem.

3. To improve the efficiency of existing techniques.

4. To understand the basic principles of designing the algorithms.

5. To compare the performance of the algorithm with respect to other techniques.

6. It is the best method of description without describing the implementation detail.

7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.

8. A good design can produce a good solution.

9. To understand the flow of the problem.

10. To measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases)

11. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.

12. With the help of algorithm, we convert art into a science.

13. To understand the principle of designing.

14. We can measure and analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design.

Analysis of algorithm

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

- **Space Complexity:** The space complexity can be understood as the amount of space required by an algorithm to run to completion.
- **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.

Let's understand it with an example.

Suppose there is a problem to solve in Computer Science, and in general, we solve a program by writing a program. If you want to write a program in some programming language like C, then before writing a program, it is necessary to write a blueprint in an informal language.

Or in other words, you should describe what you want to include in your code in an English-like language for it to be more readable and understandable before implementing it, which is nothing but the concept of Algorithm.

In general, if there is a problem **P1**, then it may have many solutions, such that each of these solutions is regarded as an algorithm. So, there may be many algorithms such as $\mathbf{A_1}$, $\mathbf{A_2}$, $\mathbf{A_3}$, …, $\mathbf{A_n}$.

Before you implement any algorithm as a program, it is better to find out which among these algorithms are good in terms of time and memory.

It would be best to analyze every algorithm in terms of **Time** that relates to which one could execute faster and **Memory** corresponding to which one will take less memory.

So, the Design and Analysis of Algorithm talks about how to design various algorithms and how to analyze them. After designing and analyzing, choose the best algorithm that takes the least time and the least memory and then implement it as a program in C.

In this course, we will be focusing more on time rather than space because time is instead a more limiting parameter in terms of the hardware. It is not easy to take a computer and change its speed. So, if we are running an algorithm on a particular platform, we are more or less stuck with the performance that platform can give us in terms of speed.

However, on the other hand, memory is relatively more flexible. We can increase the memory as when required by simply adding a memory card. So, we will focus on time than that of the space.

The running time is measured in terms of a particular piece of hardware, not a robust measure. When we run the same algorithm on a different computer or use different programming languages, we will encounter that the same algorithm takes a different time.

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 maximum 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 minimum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the minimum number of steps performed on an instance having an input size of n.

Complexity of Algorithm

The term algorithm complexity measures how many steps are required by the algorithm to solve the given problem. It evaluates the order of count of operations executed by an algorithm as a function of input data size.

To assess the complexity, the order (approximation) of the count of operation is always considered instead of counting the exact steps.

O(f) notation represents the complexity of an algorithm, which is also termed as an Asymptotic notation or "**Big O**" notation. Here the f corresponds to the function whose size is the same as that of the input data. The complexity of the asymptotic computation O(f) determines in which order the resources such as CPU time, memory, etc. are consumed by the algorithm that is articulated as a function of the size of the input data.

The complexity can be found in any form such as constant, logarithmic, linear, n*log(n), quadratic, cubic, exponential, etc. It is nothing but the order of constant, logarithmic, linear and so on, the number of steps encountered for the completion of a particular algorithm. To make it even more precise, we often call the complexity of an algorithm as "running time".

Typical Complexities of an Algorithm

- **Constant Complexity:**
  It imposes a complexity of **O(1)**. It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem. The count of operations is independent of the input data size.

- **Logarithmic Complexity:**
  It imposes a complexity of **O(log(N))**. It undergoes the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2. For N = 1,000,000, an algorithm that has a complexity of O(log(N)) would undergo 20 steps (with a constant precision). Here, the logarithmic base does not hold a necessary consequence for the operation count order, so it is usually omitted.
- **Linear Complexity:**
  - It imposes a complexity of **O(N)**. It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements. For example, if there exist 500 elements, then it will take about 500 steps. Basically, in linear complexity, the number of elements linearly depends on the number of steps. For example, the number of steps for N elements can be N/2 or 3*N.
  - It also imposes a run time of **O(n*log(n))**. It undergoes the execution of the order N*log(N) on N number of elements to solve the given problem. For a given 1000 elements, the linear complexity will execute 10,000 steps for solving a given problem.
- **Quadratic Complexity:** It imposes a complexity of **O(n^2)**. For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem.
  If N = 100, it will endure 10,000 steps. In other words, whenever the order of operation tends to have a quadratic relation with the input data size, it results in quadratic complexity. For example, for N number of elements, the steps are found to be in the order of $3*N^2/2$.
- **Cubic Complexity:** It imposes a complexity of **O(n^3)**. For N input data size, it executes the order of N^3 steps on N elements to solve a given problem.
  For example, if there exist 100 elements, it is going to execute 1,000,000 steps.
- **Exponential Complexity:** It imposes a complexity of **O(2^n), O(N!), O(n^k), ….** For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.
  For example, if N = 10, then the exponential function 2^N will result in 1024. Similarly, if N = 20, it will result in 1048 576, and if N = 100, it will result in a number having 30

digits. The exponential function N! grows even faster; for example, if N = 5 will result in 120. Likewise, if N = 10, it will result in 3,628,800 and so on.

Since the constants do not hold a significant effect on the order of count of operation, so it is better to ignore them. Thus, to consider an algorithm to be linear and equally efficient, it must undergo N, N/2 or 3*N count of operation, respectively, on the same number of elements to solve a particular problem.

How to approximate the time taken by the Algorithm?

So, to find it out, we shall first understand the types of the algorithm we have. There are two types of algorithms:

1. **Iterative Algorithm:** In the iterative approach, the function repeatedly runs until the condition is met or it fails. It involves the looping construct.
2. **Recursive Algorithm:** In the recursive approach, the function calls itself until the condition is met. It integrates the branching structure.

However, it is worth noting that any program that is written in iteration could be written as recursion. Likewise, a recursive program can be converted to iteration, making both of these algorithms equivalent to each other.

But to analyze the iterative program, we have to count the number of times the loop is going to execute, whereas in the recursive program, we use recursive equations, i.e., we write a function of F(n) in terms of F(n/2).

Suppose the program is neither iterative nor recursive. In that case, it can be concluded that there is no dependency of the running time on the input data size, i.e., whatever is the input size, the running time is going to be a constant value. Thus, for such programs, the complexity will be **O(1)**.

For Iterative Programs

Consider the following programs that are written in simple English and does not correspond to any syntax.

Example1:

In the first example, we have an integer i and a for loop running from i equals 1 to n. Now the question arises, how many times does the name get printed?

1. A()

2. {
3. int i;
4. for (i=1 to n)
5. printf("Edward");
6. }

Since i equals 1 to n, so the above program will print Edward, n number of times. Thus, the complexity will be **O(n)**.

Example2:

1. A()
2. {
3. int i, j:
4. for (i=1 to n)
5. for (j=1 to n)
6. printf("Edward");
7. }

In this case, firstly, the outer loop will run n times, such that for each time, the inner loop will also run n times. Thus, the time complexity will be **O(n^2)**.

Example3:

1. A()
2. {
3. i = 1; S = 1;
4. while (S<=n)
5. {
6. i++;
7. SS = S + i;
8. printf("Edward");
9. }
10. }

As we can see from the above example, we have two variables; i, S and then we have while S<=n, which means S will start at 1, and the entire loop will stop whenever S value reaches a point where S becomes greater than n.

Here i is incrementing in steps of one, and S will increment by the value of i, i.e., the increment in i is linear. However, the increment in S depends on the i.

Initially;

i=1, S=1

After 1st iteration;

i=2, S=3

After 2nd iteration;

i=3, S=6

After 3rd iteration;

i=4, S=10 … and so on.

Since we don't know the value of n, so let's suppose it to be k. Now, if we notice the value of S in the above case is increasing; for i=1, S=1; i=2, S=3; i=3, S=6; i=4, S=10; …

Thus, it is nothing but a series of the sum of first n natural numbers, i.e., by the time i reaches k, the value of S will be k(k+1)/2.

Asymptotic Analysis of algorithms (Growth of function)

Resources for an algorithm are usually expressed as a function regarding input. Often this function is messy and complicated to work. To study Function growth efficiently, we reduce the function down to the important part.

Let $f(n) = an^2+bn+c$

In this function, the n^2 term dominates the function that is when n gets sufficiently large.

Dominate terms are what we are interested in reducing a function, in this; we ignore all constants and coefficient and look at the highest order term concerning n.

Asymptotic notation:

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

Asymptotic analysis

It is a technique of representing limiting behavior. The methodology has the applications across science. It can be used to analyze the performance of an algorithm for some large data set.

1. In computer science in the analysis of algorithms, considering the performance of algorithms when applied to very large input datasets

The simplest example is a function $f(n) = n^2+3n$, the term 3n becomes insignificant compared to n^2 when n is very large. The function "$f(n)$ is said to be **asymptotically equivalent** to n^2 as $n \to \infty$", and here is written symbolically as $f(n) \sim n^2$.

Asymptotic notations are used to write fastest and slowest possible running time for an algorithm. These are also referred to as 'best case' and 'worst case' scenarios respectively.

"In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n)"

"These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms."

Why is Asymptotic Notation Important?

1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

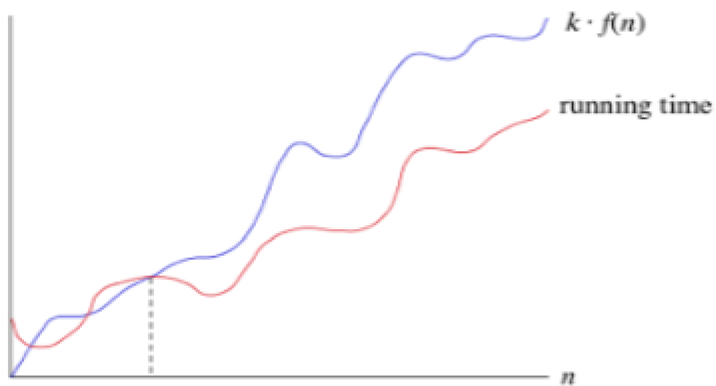
Asymptotic Notations:

Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm:

1. Big-oh notation: Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if exist positive constant c and such that

1. f (n) ⩽ k.g (n)f(n)⩽k.g(n) **for** n>n0n>n0 in all **case**

Hence, function g (n) is an upper bound for function f (n), as g (n) grows faster than f (n)



ASYMPTOTIC UPPER BOUND

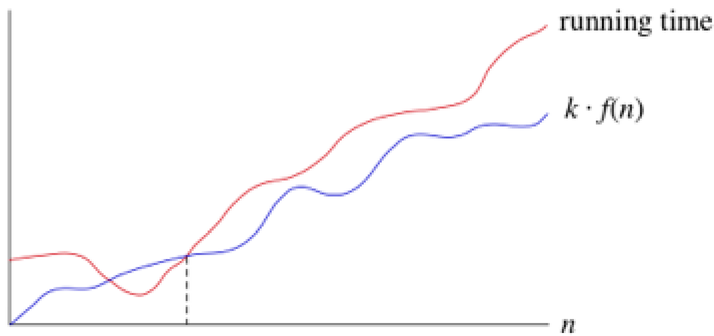
For Example:

1. 1. $3n+2=O(n)$ as $3n+2\leq4n$ **for** all $n\geq2$
2. 2. $3n+3=O(n)$ as $3n+3\leq4n$ **for** all $n\geq3$

Hence, the complexity of **f(n)** can be represented as O (g (n))

2. Omega () Notation: The function f (n) = **Ω** (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that

$F(n) \geq k^* g(n)$ for all n, $n\geq n_0$



ASYMPTOTIC LOWER BOUND

For Example:

$f(n) = 8n^2+2n-3 \geq 8n^2-3$

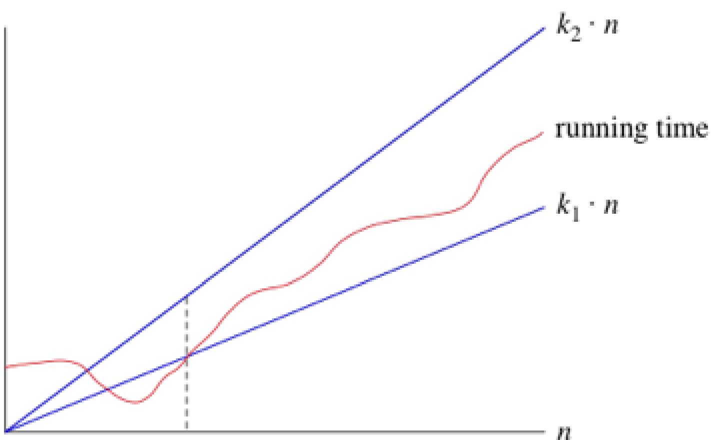
$= 7n^2+(n^2-3) \geq 7n^2$ (g(n))

Thus, $k_1=7$

Hence, the complexity of **f (n)** can be represented as **Ω** (g (n))

3. Theta (θ): The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant k_1, k_2 and k_0 such that

$k_1 * g(n) \leq f(n) \leq k_2\, g(n)$ for all n, $n \geq n_0$



ASYMPTOTIC TIGHT BOUND

For Example:

$3n+2 = \theta(n)$ as $3n+2 \geq 3n$ and $3n+2 \leq 4n$, for n

$k_1=3, k_2=4$, and $n_0=2$

Hence, the complexity of f (n) can be represented as θ (g(n)).

The Theta Notation is more precise than both the big-oh and Omega notation. The function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.

To stop the loop, has to be greater than n, and when we solve this equation, we will get **> n**. Hence, it can be concluded that we get a complexity of **O(√n)** in this case.

For Recursive Program

Consider the following recursive programs.

Example1:

1. A(n)
2. {
3. if (n>1)
4. return (A(n-1))
5. }

Solution;

Here we will see the simple Back Substitution method to solve the above problem.

T(n) = 1 + T(n-1) …Eqn. (1)

Step1: Substitute n-1 at the place of n in Eqn. (1)

T(n-1) = 1 + T(n-2) ...Eqn. (2)

Step2: Substitute n-2 at the place of n in Eqn. (1)

T(n-2) = 1 + T(n-3) …Eqn. (3)

Step3: Substitute Eqn. (2) in Eqn. (1)

$T(n)= 1 + 1+ T(n-2) = 2 + T(n-2)$ …Eqn. (4)

Step4: Substitute eqn. (3) in Eqn. (4)

$T(n) = 2 + 1 + T(n-3) = 3 + T(n-3) = …... = k + T(n-k)$ …Eqn. (5)

Now, according to Eqn. (1), i.e. $T(n) = 1 + T(n-1)$, the algorithm will run until $n>1$. Basically, n will start from a very large number, and it will decrease gradually. So, when $T(n) = 1$, the algorithm eventually stops, and such a terminating condition is called anchor condition, base condition or stopping condition.

Thus, for $k = n-1$, the T(n) will become.

Step5: Substitute $k = n-1$ in eqn. (5)

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

Hence, $T(n) = n$ or **O(n).**

Algorithm Design Techniques

The following is a list of several popular design approaches:

1. Divide and Conquer Approach: It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

- Divide the original problem into a set of subproblems.
- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.

2. Greedy Technique: Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.

- Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.
- The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

3. Dynamic Programming: Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.

This is particularly helpful when the number of copying subproblems is exponentially large. Dynamic Programming is frequently related to **Optimization Problems**.

4. Branch and Bound: In Branch & Bound algorithm a given subproblem, which cannot be bounded, has to be divided into at least two new restricted subproblems. Branch and Bound algorithm are methods for global optimization in non-convex problems. Branch and Bound algorithms can be slow, however in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the method coverage with much less effort.

5. Randomized Algorithms: A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

6. Backtracking Algorithm: Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

7. Randomized Algorithm: A randomized algorithm uses a random number at least once during the computation make a decision.

Example 1: In Quick Sort, using a random number to choose a pivot.

Example 2: Trying to factor a large number by choosing a random number as possible divisors.

Loop invariants

This is a justification technique. We use loop invariant that helps us to understand why an algorithm is correct. To prove statement S about a loop is correct, define S concerning series of smaller statement S_0 S_1....S_k where,

- The initial claim so is true before the loop begins.
- If S_{i-1} is true before iteration i begin, then one can show that S_i will be true after iteration i is over.
- The final statement S_k implies the statement S that we wish to justify as being true.

Complexity of Algorithm

The term algorithm complexity measures how many steps are required by the algorithm to solve the given problem. It evaluates the order of count of operations executed by an algorithm as a function of input data size.

To assess the complexity, the order (approximation) of the count of operation is always considered instead of counting the exact steps.

O(f) notation represents the complexity of an algorithm, which is also termed as an Asymptotic notation or "**Big O**" notation. Here the f corresponds to the function whose size is the same as that of the input data. The complexity of the asymptotic computation O(f) determines in which order the resources such as CPU time, memory, etc. are consumed by the algorithm that is articulated as a function of the size of the input data.

The complexity can be found in any form such as constant, logarithmic, linear, n*log(n), quadratic, cubic, exponential, etc. It is nothing but the order of constant, logarithmic, linear and so on, the number of steps encountered for the completion of a particular algorithm. To make it even more precise, we often call the complexity of an algorithm as "running time".

Typical Complexities of an Algorithm

- **Constant Complexity:**
  It imposes a complexity of **O(1)**. It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem. The count of operations is independent of the input data size.
- **Logarithmic Complexity:**
  It imposes a complexity of **O(log(N))**. It undergoes the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2.
  For N = 1,000,000, an algorithm that has a complexity of O(log(N)) would undergo 20 steps (with a constant precision). Here, the logarithmic base does not hold a necessary consequence for the operation count order, so it is usually omitted.
- **Linear Complexity:**
  - It imposes a complexity of **O(N)**. It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements. For example, if there exist 500 elements, then it will take about 500 steps. Basically, in linear complexity, the number of elements linearly depends on the number of steps. For example, the number of steps for N elements can be N/2 or 3*N.
  - It also imposes a run time of **O(n*log(n))**. It undergoes the execution of the order N*log(N) on N number of elements to solve the given problem. For a given 1000 elements, the linear complexity will execute 10,000 steps for solving a given problem.

- **Quadratic Complexity:** It imposes a complexity of $\mathbf{O(n^2)}$. For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem.
  If N = 100, it will endure 10,000 steps. In other words, whenever the order of operation tends to have a quadratic relation with the input data size, it results in quadratic complexity. For example, for N number of elements, the steps are found to be in the order of $3*N^2/2$.
- **Cubic Complexity:** It imposes a complexity of $\mathbf{O(n^3)}$. For N input data size, it executes the order of N^3 steps on N elements to solve a given problem.
  For example, if there exist 100 elements, it is going to execute 1,000,000 steps.
- **Exponential Complexity:** It imposes a complexity of $\mathbf{O(2^n)}$**,** $\mathbf{O(N!)}$**,** $\mathbf{O(n^k)}$**, ….** For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.
  For example, if N = 10, then the exponential function 2^N will result in 1024. Similarly, if N = 20, it will result in 1048 576, and if N = 100, it will result in a number having 30 digits. The exponential function N! grows even faster; for example, if N = 5 will result in 120. Likewise, if N = 10, it will result in 3,628,800 and so on.

Since the constants do not hold a significant effect on the order of count of operation, so it is better to ignore them. Thus, to consider an algorithm to be linear and equally efficient, it must undergo N, N/2 or 3*N count of operation, respectively, on the same number of elements to solve a particular problem.

How to approximate the time taken by the Algorithm?

So, to find it out, we shall first understand the types of the algorithm we have. There are two types of algorithms:

1. **Iterative Algorithm:** In the iterative approach, the function repeatedly runs until the condition is met or it fails. It involves the looping construct.
2. **Recursive Algorithm:** In the recursive approach, the function calls itself until the condition is met. It integrates the branching structure.

However, it is worth noting that any program that is written in iteration could be written as recursion. Likewise, a recursive program can be converted to iteration, making both of these algorithms equivalent to each other.

But to analyze the iterative program, we have to count the number of times the loop is going to execute, whereas in the recursive program, we use recursive equations, i.e., we write a function of F(n) in terms of F(n/2).

Suppose the program is neither iterative nor recursive. In that case, it can be concluded that there is no dependency of the running time on the input data size, i.e., whatever is the input size, the running time is going to be a constant value. Thus, for such programs, the complexity will be **O(1)**.

For Iterative Programs

Consider the following programs that are written in simple English and does not correspond to any syntax.

Example1:

In the first example, we have an integer i and a for loop running from i equals 1 to n. Now the question arises, how many times does the name get printed?

```
1. A()
2. {
3. int i;
4. for (i=1 to n)
5. printf("Edward");
6. }
```

Since i equals 1 to n, so the above program will print Edward, n number of times. Thus, the complexity will be **O(n)**.

Example2:

```
1. A()
2. {
3. int i, j:
4. for (i=1 to n)
5. for (j=1 to n)
6. printf("Edward");
7. }
```

In this case, firstly, the outer loop will run n times, such that for each time, the inner loop will also run n times. Thus, the time complexity will be **O(n^2)**.

Example3:

```
1.  A()
2.  {
3.  i = 1; S = 1;
4.  while (S<=n)
5.  {
6.  i++;
7.  SS = S + i;
8.  printf("Edward");
9.  }
10. }
```

As we can see from the above example, we have two variables; i, S and then we have while S<=n, which means S will start at 1, and the entire loop will stop whenever S value reaches a point where S becomes greater than n.

Here i is incrementing in steps of one, and S will increment by the value of i, i.e., the increment in i is linear. However, the increment in S depends on the i.

Initially;

i=1, S=1

After 1st iteration;

i=2, S=3

After 2nd iteration;

i=3, S=6

After 3rd iteration;

i=4, S=10 … and so on.

Since we don't know the value of n, so let's suppose it to be k. Now, if we notice the value of S in the above case is increasing; for i=1, S=1; i=2, S=3; i=3, S=6; i=4, S=10; …

Thus, it is nothing but a series of the sum of first n natural numbers, i.e., by the time i reaches k, the value of S will be k(k+1)/2.

To stop the loop, $\frac{k(k+1)}{2}$ has to be greater than n, and when we solve this equation, we will get $\frac{k^2+k}{2}$ **> n**. Hence, it can be concluded that we get a complexity of **O(√n)** in this case.

For Recursive Program

Consider the following recursive programs.

Example1:

1. A(n)
2. {
3. if (n>1)
4. return (A(n-1))
5. }

Solution;

Here we will see the simple Back Substitution method to solve the above problem.

T(n) = 1 + T(n-1) …Eqn. (1)

Step1: Substitute n-1 at the place of n in Eqn. (1)

T(n-1) = 1 + T(n-2) ...Eqn. (2)

Step2: Substitute n-2 at the place of n in Eqn. (1)

T(n-2) = 1 + T(n-3) …Eqn. (3)

Step3: Substitute Eqn. (2) in Eqn. (1)

T(n)= 1 + 1+ T(n-2) = 2 + T(n-2) …Eqn. (4)

Step4: Substitute eqn. (3) in Eqn. (4)

T(n) = 2 + 1 + T(n-3) = 3 + T(n-3) = …... = k + T(n-k) …Eqn. (5)

Now, according to Eqn. (1), i.e. T(n) = 1 + T(n-1), the algorithm will run until n>1. Basically, n will start from a very large number, and it will decrease gradually. So, when T(n) = 1, the algorithm eventually stops, and such a terminating condition is called anchor condition, base condition or stopping condition.

Thus, for k = n-1, the T(n) will become.

Step5: Substitute k = n-1 in eqn. (5)

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

Hence, $T(n) = n$ or **O(n).**

Recurrence Relation

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

$T(n) = \theta(1)$ if n=1

$2T\left(\frac{n}{2}\right) + \theta(n)$ if n>1

There are four methods for solving Recurrence:

1. Substitution Method
2. Iteration Method
3. Recursion Tree Method
4. Master Method

1. Substitution Method:

The Substitution Method Consists of two main steps:

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

For Example1 Solve the equation by Substitution Method.

$$T(n) = T\left(\frac{n}{2}\right) + n$$

We have to show that it is asymptotically bound by O (log n).

Solution:

For T (n) = O (log n)

We have to show that for some constant c

1. T (n) ≤c logn.

Put this in given Recurrence Equation.

$$T(n) \le c \log\left(\frac{n}{2}\right) + 1$$

$$\le c \log\left(\frac{n}{2}\right) + 1 = c \log n - c\log_2 2 + 1$$

$$\le c \log n \text{ for } c \ge 1$$

Thus **T (n) =O logn**.

Example2 Consider the Recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad n>1$$

Find an Asymptotic bound on T.

Solution:

We guess the solution is O (n (logn)).Thus for constant 'c'.

T (n) ≤c n logn

Put this in given Recurrence Equation.

Now,

$$T(n) \le 2c\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right) + n$$

$$\le cn\log n - cn\log 2 + n$$

$$= cn \log n - n(c\log 2 - 1)$$

$$\le cn \log n \text{ for } (c \ge 1)$$

Thus **T (n) = O (n logn)**.

2. Iteration Methods

It means to expand the recurrence and express it as a summation of terms of n and initial condition.

Example1: Consider the Recurrence

1. $T(n) = 1$ **if** $n=1$
2. $= 2T(n-1)$ **if** $n>1$

Solution:

$$
\begin{aligned}
T(n) &= 2T(n-1) \\
&= 2[2T(n-2)] = 2^2T(n-2) \\
&= 4[2T(n-3)] = 2^3T(n-3) \\
&= 8[2T(n-4)] = 2^4T(n-4) \quad \text{(Eq.1)}
\end{aligned}
$$

Repeat the procedure for i times

$T(n) = 2^i\, T(n-i)$
Put $n-i=1$ or $i= n-1$ in (Eq.1)
$T(n) = 2^{n-1}\, T(1)$
$= 2^{n-1} .1$ $\{T(1) =1 \text{ .....given}\}$
$= 2^{n-1}$

Example2: Consider the Recurrence

1. $T(n) = T(n-1) +1$ and $T(1) = \theta(1)$.

Solution:

$$
\begin{aligned}
T(n) &= T(n-1) +1 \\
&= (T(n-2) +1) +1 = (T(n-3) +1) +1+1 \\
&= T(n-4) +4 = T(n-5) +1+4 \\
&= T(n-5) +5= T(n-k) + k
\end{aligned}
$$

Where $k = n-1$
$T(n-k) = T(1) = \theta(1)$
$T(n) = \theta(1) + (n-1) = 1+n-1=n= \theta(n)$.

Recursion Tree Method

1. Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.

2. In general, we consider the second term in recurrence as root.

3. It is useful when the divide & Conquer algorithm is used.

4. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem.

5. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.

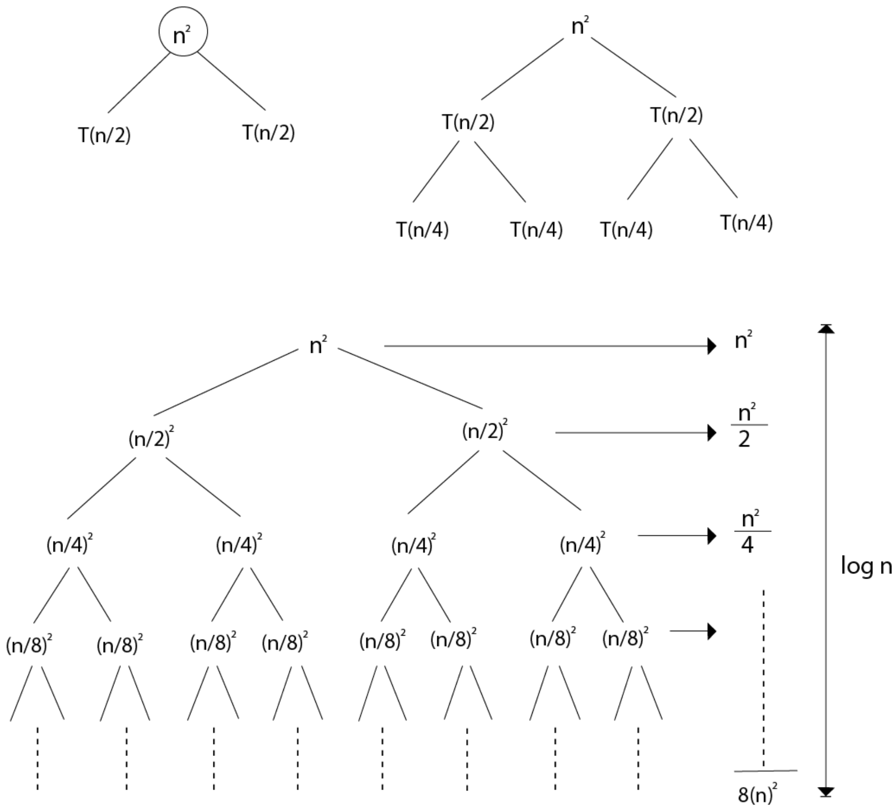
6. A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.

Example 1

Consider $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

We have to obtain the asymptotic bound using recursion tree method.

Solution: The Recursion tree for the above recurrence is



$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \log n \text{ times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

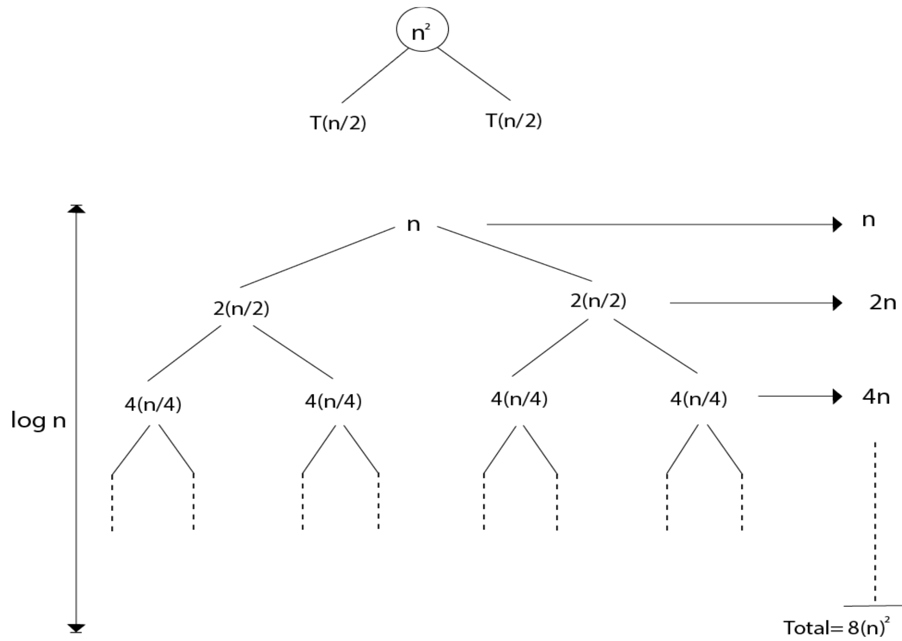
$$\mathbf{T(n) = \theta n^2}$$

Example 2: Consider the following recurrence

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

Solution: The recursion trees for the above recurrence



We have $n + 2n + 4n + \ldots \ldots \log_2 n$ times

$$= n\,(1 + 2 + 4 + \ldots \ldots \log_2 n \text{ times})$$

$$= n\,\frac{(2\ \log_2 n - 1)}{(2-1)} = \frac{n(n-1)}{1} = n^2 - n = \theta(n^2)$$

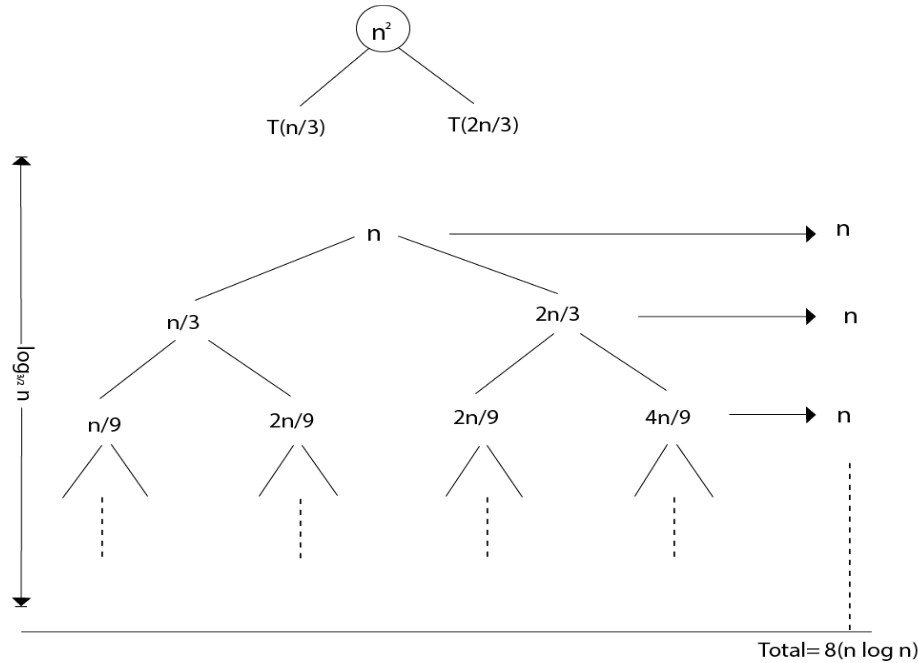
$$\mathbf{T\ (n) = \theta(n^2)}$$

Example 3: Consider the following recurrence

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

Solution: The given Recurrence has the following recursion tree



When we add the values across the levels of the recursion trees, we get a value of n for every level. The longest path from the root to leaf is

$$n \longrightarrow \frac{2}{3}n \longrightarrow \left(\frac{2}{3}\right) n \longrightarrow \ldots .1$$

Since $\left(\frac{2}{3}\right)$ n=1 when i=$\log\frac{3}{2}$ n.

Thus the height of the tree is $\log\frac{3}{2}$ n.

$$T\ (n) = n + n + n + \ldots\ldots + \log\frac{3}{2}\,n \text{ times.} = \boldsymbol{\theta(n\ logn)}$$

Master Method

The Master Method is used for solving the following types of recurrence

T (n) = a T $\left(\frac{n}{b}\right)$ + f (n) with a≥1 and b≥1 be constant & f(n) be a function and $\frac{n}{b}$ can be interpreted as

Let T (n) is defined on non-negative integers by the recurrence.

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.
- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

Master Theorem:

It is possible to complete an asymptotic tight bound in these three cases:

$$T(n)=\begin{cases}\Theta\left(n^{\log_b a}\right) & f(n)=O\left(n^{\log_b a-\varepsilon}\right)\\ \Theta\left(n^{\log_b a}\log n\right) & f(n)=\Theta\left(n^{\log_b a}\right)\\ \Theta\left(f(n)\right) & f(n)=\Omega\left(n^{\log_b a+\varepsilon}\right)\text{AND}\\ & af(n/b)<cf(n)\ \text{for large}\ n\end{cases}\quad \begin{matrix}\varepsilon>0\\ c<1\end{matrix}$$

Case1: If f (n) = $O\left(n^{\log_b a-\varepsilon}\right)$ for some constant ε >0, then it follows that:

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

Example:

T (n) = 8 T $\left(\frac{n}{2}\right) + 1000n^2$ apply master theorem on it.

Solution:

Compare T (n) = 8 T $\left(\frac{n}{2}\right) + 1000n^2$ with

T (n) = a T $\left(\frac{n}{b}\right) + f(n)$ *with* $a \geq 1$ *and* $b > 1$

a = 8, b=2, f (n) = 1000 n^2, $\log_b a = \log_2 8 = 3$

Put all the values in: f (n) = $O\left(n^{\log_b a-\varepsilon}\right)$

1000 n^2 = O ($n^{3-\varepsilon}$)

If we choose ε=1, we get: 1000 n^2 = O (n^{3-1}) = O (n^2)

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

T (n) = Θ $\left(n^{\log_b a}\right)$

Therefore: T (n) = Θ (n^3)

Case 2: If it is true, for some constant k ≥ 0 that:

F (n) = Θ $\left(n^{\log_b a} \log^k n\right)$ then it follows that: T (n) = Θ $\left(n^{\log_b a} \log^{k+1} n\right)$

Example:

T (n) = 2 $T\left(\frac{n}{2}\right) + 10n$, solve the recurrence by using the master method.

As compare the given problem with T (n) = a T $\left(\frac{n}{b}\right) + f(n)$ *with* $a \geq 1$ *and* $b > 1$ a = 2, b=2, k=0, f (n) = 10n, $\log_b a = \log_2 2 = 1$

Put all the values in f (n) =Θ $\left(n^{\log_b a} \log^k n\right)$, we will get

10n = Θ (n^1) = Θ (n) which is true.

Therefore: T (n) = Θ $\left(n^{\log_b a} \log^{k+1} n\right)$

$= \Theta(n \log n)$

Case 3: If it is true $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ for some constant ε >0 and it also true that: $a f\left(\frac{n}{b}\right) \le cf(n)$ for some constant c<1 for large value of n ,then :

1. $T(n) = \Theta((f(n))$

Example: Solve the recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

Solution:

Compare the given problem with $T(n) = a T\left(\frac{n}{b}\right) + f(n) \text{ with } a \ge 1 \text{ and } b > 1$

$a= 2$, $b =2$, $f(n) = n^2$, $\log_b a = \log_2 2 =1$

Put all the values in $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ ..... (Eq. 1)

If we insert all the value in (Eq.1), we will get

$n^2 = \Omega(n^{1+\varepsilon})$ put ε =1, then the equality will hold.

$n^2 = \Omega(n^{1+1}) = \Omega(n^2)$

Now we will also check the second condition:

$$2\left(\frac{n}{2}\right)^2 \le cn^2 \Rightarrow \frac{1}{2}n^2 \le cn^2$$

If we will choose c =1/2, it is true:

$$\frac{1}{2}n^2 \le \frac{1}{2}n^2 \quad \forall n \ge 1$$

So it follows: $T(n) = \Theta((f(n))$

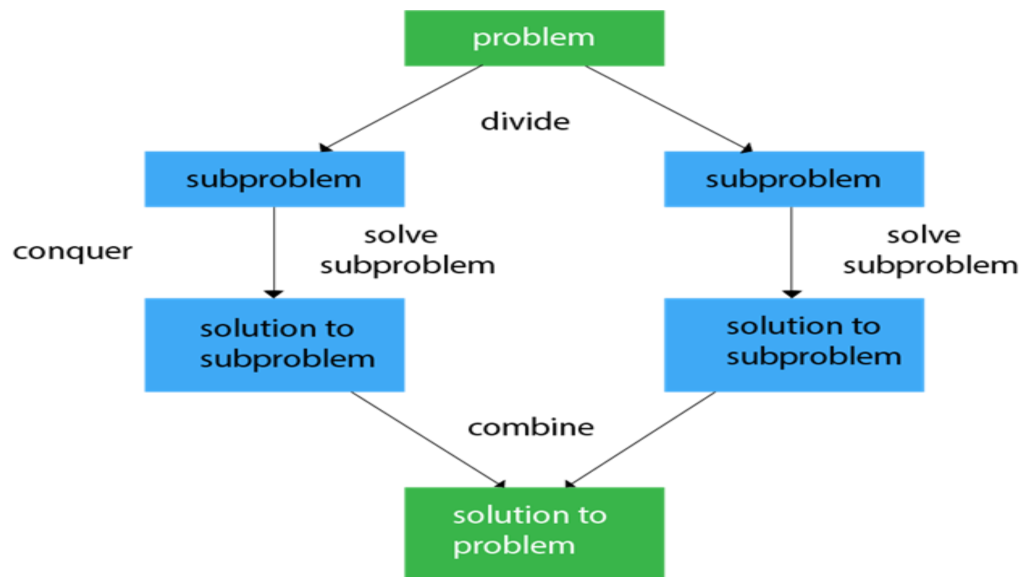
$T(n) = \Theta(n^2)$

Divide and Conquer Introduction

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

44.2M
777
Prime Ministers of India | List of Prime Minister of India (1947-2020)

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.
4. **Closest Pair of Points:** It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric space, given n points, such that the distance between the pair of points should be minimal.

5. **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.
6. **Cooley-Tukey Fast Fourier Transform (FFT) algorithm:** The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of O(nlogn).
7. **Karatsuba algorithm for fast multiplication:** It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got published in 1962. It multiplies two n-digit numbers in such a way by reducing it to at most single-digit.

Binary Search

1. In Binary Search technique, we search an element in a sorted array by recursively dividing the interval in half.

2. Firstly, we take the whole array as an interval.

3. If the Pivot Element (the item to be searched) is less than the item in the middle of the interval, We discard the second half of the list and recursively repeat the process for the first half of the list by calculating the new middle and last element.

4. If the Pivot Element (the item to be searched) is greater than the item in the middle of the interval, we discard the first half of the list and work recursively on the second half by calculating the new beginning and middle element.

5. Repeatedly, check until the value is found or interval is empty.

Analysis:

1. **Input:** an array A of size n, already sorted in the ascending or descending order.
2. **Output:** analyze to search an element item in the sorted array of size n.
3. **Logic:** Let T (n) = number of comparisons of an item with n elements in a sorted array.

- Set BEG = 1 and END = n
- Find mid = $\text{int}\left(\frac{beg + end}{2}\right)$
- Compare the search item with the mid item.

Case 1: item = A[mid], then LOC = mid, but it the best case and T (n) = 1

Case 2: item ≠A [mid], then we will split the array into two equal parts of size $\frac{n}{2}$.

And again find the midpoint of the half-sorted array and compare with search element.

Repeat the same process until a search element is found.

T (n) = $T\left(\frac{n}{2}\right)+1$ ...... (Equation 1)

{Time to compare the search element with mid element, then with half of the selected half part of array}

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + 1$, putting $\frac{n}{2}$ in place of n.

Then we get: T (n) = $(T\left(\frac{n}{2^2}\right) + 1) + 1$...........By putting $T\frac{n}{2}$ in (1) equation

T (n) = $T\left(\frac{n}{2^2}\right) + 2$...................... (Equation 2)

$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1$................... Putting $\frac{n}{2}$ in place of n in eq 1.

T (n) = $T\left(\frac{n}{2^3}\right) + 1 + 2$

T (n) =$T\left(\frac{n}{2^3}\right) + 3$................................ (Equation 3)

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$................. Putting $\frac{n}{3}$ in place of n in eq1

Put $T\left(\frac{n}{2^3}\right)$ in eq (3)

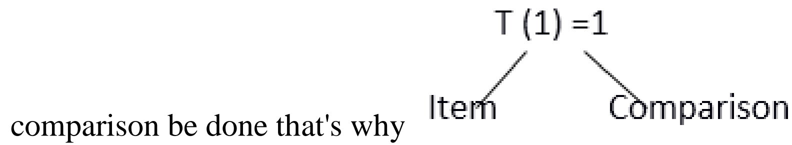
T (n) = $T\left(\frac{n}{2^4}\right) + 4$

Repeat the same process ith times

T (n) = $T\left(\frac{n}{2^i}\right) + i$ .....

Stopping Condition: T (1) =1

At least there will be only one term left that's why that term will compare out, and only one comparison be done that's why



Is the last term of the equation and it will be equal to 1

$$\frac{n}{2^i} = 1$$

$\frac{n}{2^i}$ Is the last term of the equation and it will be equal to 1

$$n = 2^i$$

Applying log both sides

$$\log n = \log_2 i$$

$$\text{Log } n = i \log 2$$

$$\frac{\log n}{\log 2} = 1$$

$$\log_2 n = i$$

$$T(n) = T\left(\frac{n}{2^i}\right) + i$$

$\frac{n}{2^i}$ = 1 as in eq 5

$= T(1) + i$

$= 1 + i$.................................. $T(1) = 1$ by stopping condition

$= 1 + \log_2 n$

$= \log_2 n$ .... (1 is a constant that's why ignore it)

Therefore, binary search is of order o ($\log_2 n$)

Merge Sort

Merge sort is yet another sorting algorithm that falls under the category of Divide and Conquer technique. It is one of the best sorting techniques that successfully build a recursive algorithm.

Divide and Conquer Strategy

In this technique, we segment a problem into two halves and solve them individually. After finding the solution of each half, we merge them back to represent the solution of the main problem.

Suppose we have an array **A**, such that our main concern will be to sort the subsection, which starts at index **p** and ends at index **r**, represented by **A[p..r]**.

Divide

If assumed **q** to be the central point somewhere in between **p** and **r**, then we will fragment the subarray **A[p..r]** into two arrays **A[p..q]** and **A[q+1, r]**.

Conquer

After splitting the arrays into two halves, the next step is to conquer. In this step, we individually sort both of the subarrays **A[p..q]** and **A[q+1, r]**. In case if we did not reach the base situation, then we again follow the same procedure, i.e., we further segment these subarrays followed by sorting them separately.

Combine

As when the base step is acquired by the conquer step, we successfully get our sorted subarrays **A[p..q]** and **A[q+1, r]**, after which we merge them back to form a new sorted array **[p..r]**.

Merge Sort algorithm

The MergeSort function keeps on splitting an array into two halves until a condition is met where we try to perform MergeSort on a subarray of size 1, i.e., **p == r**.

And then, it combines the individually sorted subarrays into larger arrays until the whole array is merged.

1. ALGORITHM-MERGE SORT
2. 1. If p<**r**
3. 2. Then q → (p+ r)/2
4. 3. MERGE-SORT (A, p, q)
5. 4. MERGE-SORT (A, q+1,r)

6. 5. MERGE (A, p, q, r)

   Here we called **MergeSort(A, 0, length(A)-1)** to sort the complete array.

   As you can see in the image given below, the merge sort algorithm recursively divides the array into halves until the base condition is met, where we are left with only 1 element in the array. And then, the merge function picks up the sorted sub-arrays and merge them back to sort the entire array.

   The following figure illustrates the dividing (splitting) procedure.

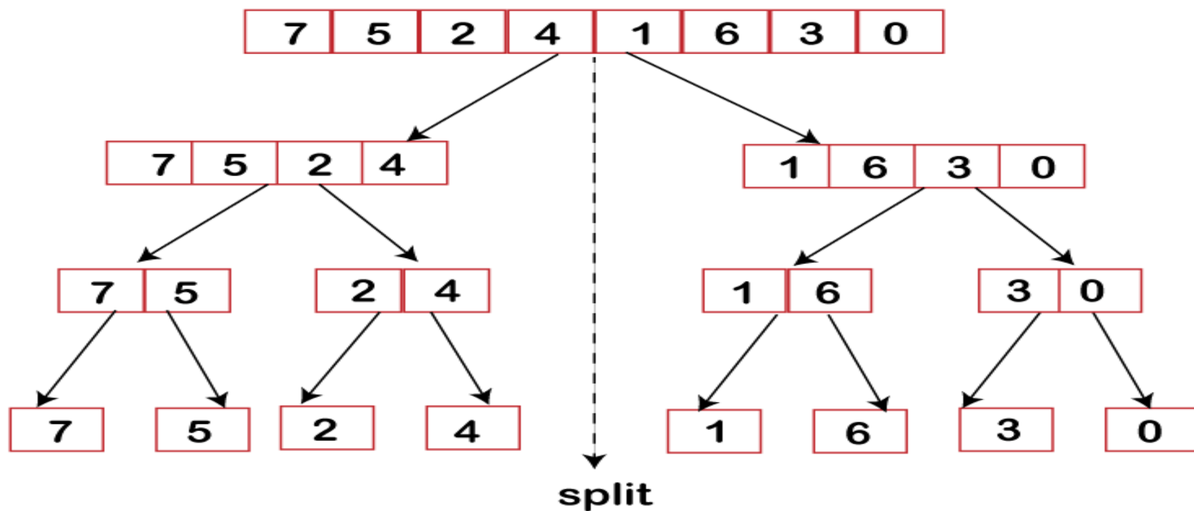


Figure 1: Merge Sort Divide Phase

1. FUNCTIONS: MERGE (A, p, q, r)
2.
3. 1. n 1 = q-p+1
4. 2. n 2= r-q
5. 3. create arrays [1.....n 1 + 1] and R [1.....n 2 +1]
6. 4. for i ← 1 to n 1
7. 5. do [i] ← A [p+ i-1]
8. 6. for j ← 1 to n2
9. 7. do R[j] ← A[q + j]
10. 8. L [n 1+ 1] ← ∞
11. 9. R[n 2+ 1] ← ∞
12. 10. I ← 1
13. 11. J ← 1

14. 12. For k ← p to r
15. 13. Do if L [i] ≤ R[j]
16. 14. then A[k] ← L[i]
17. 15. i ← i +1
18. 16. else A[k] ← R[j]
19. 17. j ← j+1

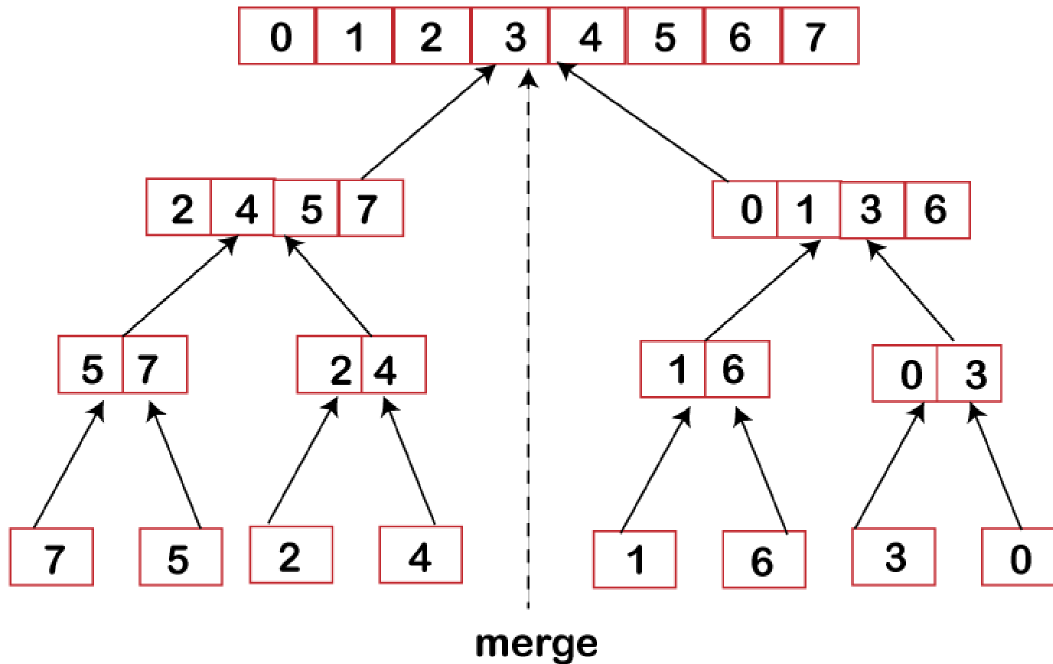


Figure 2: Merge Sort Combine Phase

The merge step of Merge Sort

Mainly the recursive algorithm depends on a base case as well as its ability to merge back the results derived from the base cases. Merge sort is no different algorithm, just the fact here the **merge** step possesses more importance.

To any given problem, the merge step is one such solution that combines the two individually sorted lists(arrays) to build one large sorted list(array).

The merge sort algorithm upholds three pointers, i.e., one for both of the two arrays and the other one to preserve the final sorted array's current index.

1. Did you reach the end of the array?
2. No:

3. Firstly, start with comparing the current elements of both the arrays.
4. Next, copy the smaller element into the sorted array.
5. Lastly, move the pointer of the element containing a smaller element.
6. Yes:
7. Simply copy the rest of the elements of the non-empty array

Merge( ) Function Explained Step-By-Step

Consider the following example of an unsorted array, which we are going to sort with the help of the Merge Sort algorithm.

A= (36,25,40,2,7,80,15)

Step1: The merge sort algorithm iteratively divides an array into equal halves until we achieve an atomic value. In case if there are an odd number of elements in an array, then one of the halves will have more elements than the other half.

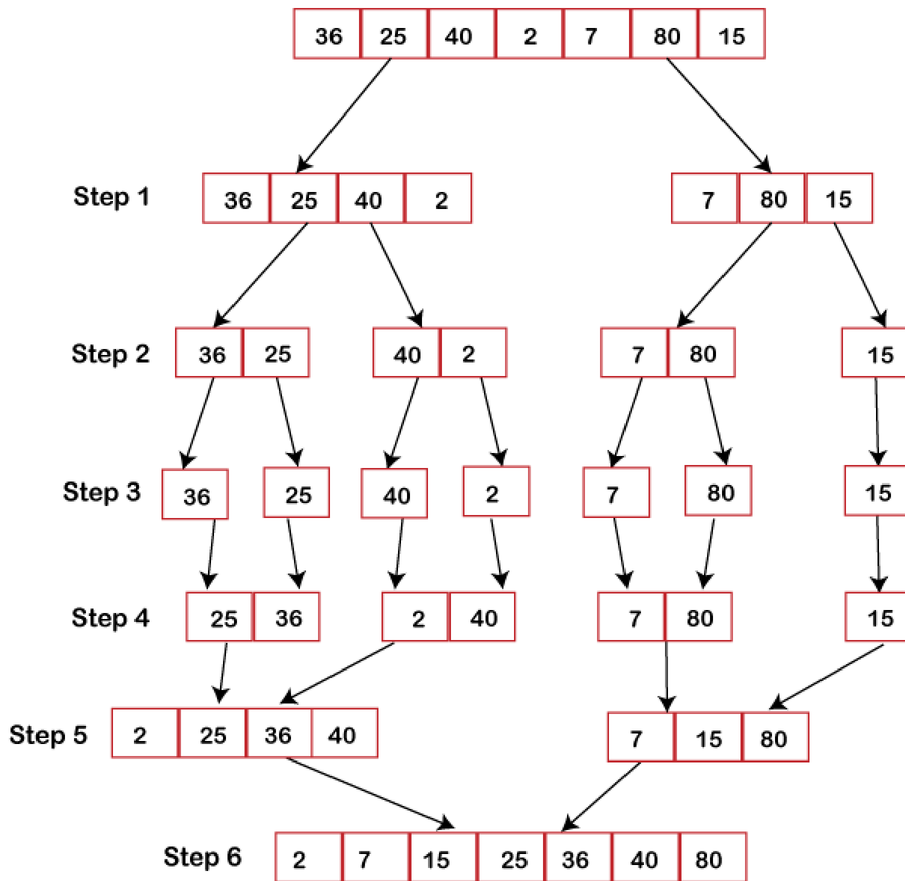
Step2: After dividing an array into two subarrays, we will notice that it did not hamper the order of elements as they were in the original array. After now, we will further divide these two arrays into other halves.

Step3: Again, we will divide these arrays until we achieve an atomic value, i.e., a value that cannot be further divided.

Step4: Next, we will merge them back in the same way as they were broken down.

Step5: For each list, we will first compare the element and then combine them to form a new sorted list.

Step6: In the next iteration, we will compare the lists of two data values and merge them back into a list of found data values, all placed in a sorted manner.



Hence the array is sorted.

Analysis of Merge Sort:

Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most $2T\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

But we ignore '-1' because the element will take some time to be copied in merge lists.

So $T(n) = 2T\left(\frac{n}{2}\right) + n$...equation 1

Putting $n=\frac{n}{2}$ in place of n in ....equation 1

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$...................equation2

Put 2 equation in 1 equation

$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$

$= 2^2T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n$

$T(n) = 2^2\, T\left(\frac{n}{2^2}\right) + 2n$......................equation 3

Putting $n=\frac{n}{2^2}$ in equation 1

$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$......................equation4

Putting 4 equation in 3 equation

$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

$T(n) = 2^3\ T\left(\frac{n}{2^3}\right) + n + 2n$

$T(n) = 2^3\ T\left(\frac{n}{2^3}\right) + 3n$............................equation5

From eq 1, eq3, eq 5.......we get

$T(n) = 2^i\, T\left(\frac{n}{2^i}\right) + in$..............................equation6

From Stopping Condition:

$\frac{n}{2^i} = 1$ And $T\left(\frac{n}{2^i}\right)=0$

$n = 2^i$

Apply log both sides:

log $n=\log_2 i$

logn= i log2

$$\frac{\log n}{\log 2}=i$$

$\log_2 n=i$

From 6 equation

$$T(n) = 2^i\, T\left(\frac{n}{2^i}\right) + in$$

$$= 2^i \times 0 + \log_2 n \,.\, n$$

$$\mathbf{= T(n) = n.\log n}$$

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

Space Complexity: The space complexity of merge sort is **O(n)**.

Merge Sort Applications

The concept of merge sort is applicable in the following areas:

- Inversion count problem
- External sorting
- E-commerce applications

Quick sort

It is an algorithm of Divide & Conquer type.

Divide: Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub- array is larger than the middle element.

Conquer: Recursively, sort two sub arrays.

Combine: Combine the already sorted array.

Algorithm:

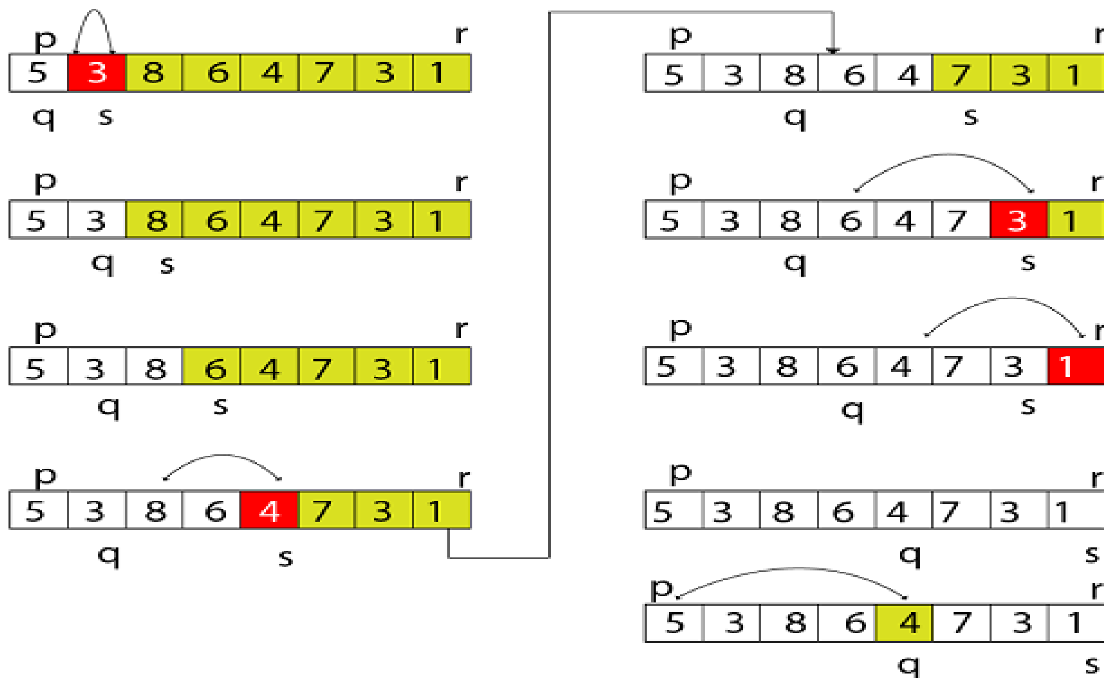
1. QUICKSORT (array A, **int** m, **int** n)
2. 1 **if** (n > m)
3. 2 then
4. 3 i ← a random index from [m,n]
5. 4 swap A [i] with A[m]
6. 5 o ← PARTITION (A, m, n)
7. 6 QUICKSORT (A, m, o - 1)
8. 7 QUICKSORT (A, o + 1, n)

Partition Algorithm:

Partition algorithm rearranges the sub arrays in a place.

1. PARTITION (array A, **int** m, **int** n)
2. 1 x ← A[m]
3. 2 o ← m
4. 3 **for** p ← m + 1 to n
5. 4 **do if** (A[p] < x)
6. 5 then o ← o + 1
7. 6 swap A[o] with A[p]
8. 7 swap A[m] with A[o]
9. 8 **return** o

Figure: shows the execution trace partition algorithm



Example of Quick Sort:

1. 44 33 11 55 77 90 40 60 99 22 88

Let **44** be the **Pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. As **22** is smaller than **44** so swap them.

22	33	11	55	77	90	40	60	99	**44**	88

Now comparing **44** to the left side element and the element must be **greater** than 44 then swap them. As **55** are greater than **44** so swap them.

22	33	11	**44**	77	90	40	60	99	**55**	88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element **44** & one right from pivot element.

22	33	11	**40**	77	90	**44**	60	99	55	88

Swap with 77:

22	33	11	40	**44**	90	**77**	60	99	55	88

Now, the element on the right side and left side are greater than and smaller than **44** respectively.

Now we get two sorted lists:

22 33 11 40 **44** 90 77 66 99 55 88

Sublist1 (22 33 11 40) **Sublist2** (90 77 66 99 55 88)

And these sublists are sorted under the same process as above done.

These two sorted sublists side by side.

22 33 11 40 **44** **90** 77 60 99 55 88

11 33 **22** 40 **44** 88 77 60 99 55 **90**

11 **22** 33 40 **44** 88 77 60 **90** 55 **99**

First sorted list

88 77 60 **55** **90** 99

Sublist3 (88 77 60 55) **Sublist4** (90 99)

55 77 60 **88** 90 99

Sorted (90 99)

55 **77** **60**

55 60 77

Sorted

Merging Sublists:

11	**22**	**33**	**40**	**44**	**55**	**60**	**77**	**88**	**90**	**99**

SORTED LISTS

Worst Case Analysis: It is the case when items are already in sorted form and we try to sort them again. This will takes lots of time and space.

Equation:

1. T (n) =T(1)+T(n-1)+n

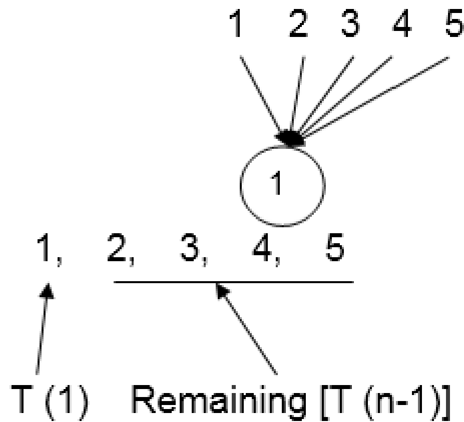
T (1) is time taken by pivot element.

T (n-1) is time taken by remaining element except for pivot element.

N: the number of comparisons required to identify the exact position of itself (every element)

If we compare first element pivot with other, then there will be 5 comparisons.

It means there will be n comparisons if there are n items.



Relational Formula for Worst Case:

T (n) =T (1) + T (n-1) + n...................... (1)

T (n-1) =T (1) +T (n-1-1) + (n-1)

By putting (n-1) in place of n in equation1

Put T (n-1) in equation1

T(n)=T(1)+T(1)+(T(n-2)+(n-1)+n......................(ii)

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

T (n-2) =T (1) +T (n-3) + (n-2)

By putting (n-2) in place of n in equation1

Put T (n-2) in equation (ii)

T(n)=2T(1)+T(1)+T(n-3)+(n-2)+(n-1)+n

T(n)=3T(1)+T(n-3)+)+(n-2)+(n-1)+n

T (n-3) =T (1) +T (n-4) +n-3

By putting (n-3) in place of n in equation1

T(n)=3T(1)+T(1)+T(n-4)+(n-3)+(n-2)+(n-1)+n

=4T(1)+T(n-4))+(n-3)+(n-2)+(n-1)+n..........................(iii)

T(n)=(n-1) T(1) + T(n-(n-1))+(n-(n-2))+(n-(n-3))+(n-(n-4))+n
T (n) = (n-1) T (1) + T (1) + 2 + 3 + 4+............n
T (n) = (n-1) T (1) +T (1) +2+3+4+...........+n+1-1

[Adding 1 and subtracting 1 for making AP series]

T (n) = (n-1) T (1) +T (1) +1+2+3+4+........ + n-1

$$T(n) = (n-1)\,T(1) + T(1) + \frac{n(n+1)}{2} - 1$$

Stopping Condition: T (1) =0

Because at last there is only one element left and no comparison is required.

$$T(n) = (n-1)(0) + 0 + \frac{n(n+1)}{2} - 1$$

$$T(n) = \frac{n^2 + n - 2}{2}$$

Avoid all the terms expect higher terms n^2

$$T(n) = O(n^2)$$

Worst Case Complexity of Quick Sort is T (n) =O (n^2)

Randomized Quick Sort [Average Case]:

Generally, we assume the first element of the list as the pivot element. In an average Case, the number of chances to get a pivot element is equal to the number of items.

1. Let total time taken =T (n)
2. For eg: In a given list
3. p 1, p 2, p 3, p 4............pn
4. If p 1 is the pivot list then we have 2 lists.
5. I.e. T (0) and T (n-1)
6. If p2 is the pivot list then we have 2 lists.
7. I.e. T (1) and T (n-2)
8. p 1, p 2, p 3, p 4............pn
9. If p3 is the pivot list then we have 2 lists.
10. I.e. T (2) and T (n-3)
11. p 1, p 2, p 3, p 4............p n

So in general if we take the **Kth** element to be the pivot element.

Then,

$$T(n) = \sum_{K=1}^{n} T(K-1) + T(n-k)$$

Pivot element will do n comparison and we are doing average case so,

$$T(n) = n+1 + \frac{1}{n} \left(\sum_{K=1}^{n} T(K-1) + T(n-k)\right)$$

N comparisons (n+1) — Average of n elements ($\frac{1}{n}(\sum_{K=1}^{n} T(K-1) + T(n-k))$)

So Relational Formula for Randomized Quick Sort is:

$$T(n) = n+1 + \frac{1}{n} \left(\left(\sum_{K=1}^{n} T(K-1) + T(n-k)\right)\right)$$

$$= n+1 + \frac{1}{n} (T(0)+T(1)+T(2)+...T(n-1)+T(n-2)+T(n-3)+...T(0))$$

$$= n+1 + \frac{1}{n} \times 2 (T(0)+T(1)+T(2)+...T(n-2)+T(n-1))$$

1. n T (n) = n (n+1) +2 (T(0)+T(1)+T(2)+...T(n-1)........eq 1

   Put n=n-1 in eq 1

1. (n -1) T (n-1) = (n-1) n+2 (T(0)+T(1)+T(2)+...T(n-2)......eq2

   From eq1 and eq 2

   n T (n) - (n-1) T (n-1)= n(n+1)-n(n-1)+2 (T(0)+T(1)+T(2)+?T(n-2)+T(n-1))-2(T(0)+T(1)+T(2)+...T(n-2))
   n T(n)- (n-1) T(n-1)= n[n+1-n+1]+2T(n-1)
   n T(n)=[2+(n-1)]T(n-1)+2n
   n T(n)= n+1 T(n-1)+2n

$$\frac{n}{n+1} T(n) = \frac{2n}{n+1} + T(n-1) \qquad \text{[Divide by n+1]}$$

$$\frac{1}{n+1} T(n) = \frac{2}{n+1} + \frac{T(n-1)}{n} \qquad \text{[Divide by n] ....eq 3}$$

Put n=n-1 in eq 3

$$\frac{1}{n}T\ (n\text{-}1) = \frac{2}{n} + \frac{T(n-2)}{n-1} \text{..........eq4}$$

Put 4 eq in 3 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{T(n-2)}{n-1} \text{............eq5}$$

Put n=n-2 in eq 3

$$\frac{T(n-2)}{n-1} = \frac{2}{n-1} + \frac{2}{n} + \frac{T(n-3)}{n-2} \text{..........eq6}$$

Put 6 eq in 5 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{T(n-3)}{n-2} \text{.........eq7}$$

Put n=n-3 in eq 3

$$\frac{T(n-3)}{n-2} = \frac{2}{n-2} + \frac{T(n-4)}{n-3} \text{............eq8}$$

Put 8 eq in 7 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \frac{T(n-4)}{n-3} \text{............eq9}$$

2 terms of $\frac{2}{n+1} + \frac{2}{n} = T(n-2)$

3 terms of $\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} = T(n-3)$

4 terms of $\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} = T(n\text{-}4)$

From 3eq, 5eq, 7eq, 9 eq we get

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots + \frac{2}{3} + \frac{T(n-(n-1))}{n-(n-2)} \ldots\ldots\ldots \text{eq10}$$

From 3 eq $\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{T(n-1)}{n}$

Put n=1

$$\frac{T(1)}{2} = \frac{2}{2} + \frac{T(0)}{1}$$

$$\frac{T(1)}{2} = 1$$

From 10 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots + \frac{2}{3} + \frac{T(1)}{2}$$

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots + \frac{2}{3} + 1$$

Multiply and divide the last term by 2

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots\ldots + \frac{2}{3} + 2 \times \frac{1}{2}$$

$$= 2\left[\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \ldots\ldots + \frac{1}{n} + \frac{1}{n+1}\right]$$

$$= 2 \sum_{2 \le k \le n+1}^{n} \frac{1}{k} = 2 \int_{2}^{n+1} \frac{1}{k}$$

Multiply & divide k by n

$$= 2 \int_{2}^{n+1} \frac{1}{\frac{k.n}{n}}$$

Put $\frac{k}{n} = \text{x and } \frac{1}{n} = \text{dx}$

$$\frac{T(n)}{n+1} = 2 \int_{2}^{n+1} \frac{1}{x} \, dx$$

$$= 2 \log x \int_{2}^{n+1}$$

NOTE: $\int \frac{1}{x} dx = \log x$

=2[log (n+1)-log2]

T (n) =2(n+1) [log (n+1)-log2]

Ignoring Constant we get

T (n) =n log n

T (n) =O (n log n)

Is the average case complexity of quick sort for sorting n elements.

3. Quick Sort [Best Case]: In any sorting, best case is the only case in which we don't make any comparison between elements that is only done when we have only one element to sort.

Method Name	Equation	Stopping Condition	Complexities
1.Quick Sort[Worst Case]	T(n)=T(n-1)+T(0)+n	T(1)=0	$T(n)=n^2$
2.Quick Sort[Average Case]	$T(n)=n+1+\frac{1}{n}\left(\sum_{K=1}^{n} T(k-1)+T(n-k)\right)$		T(n)=nlogn