UNIT-I

Introduction: Algorithm, Pseudo code for expressing algorithms, Performance Analysis-Space complexity, Time complexity, Asymptotic Notation- Big oh notation, Omega notation, Theta notation and Little oh notation, Disjoint Sets- disjoint set operations, union and find operations.

Divide and conquer: General method, applications-Binary search, Quick sort, Merge sort.

****

Algorithm:

History of Algorithm

The word **algorithm** comes from the name of a persian, **Muhammad ibn Musa Al-Khwarizmi**, dubbed the father of algebra (al-jabr).

- He is credited with providing the step-by-step rules for adding, subtracting, multiplying, and dividing ordinary decimal numbers.
- When written in Latin, the name became Algorismus, from which algorithm is a small step
- This word has taken on a special significance in computer science, where "algorithm" has come to refer to a method that can be used by a computer for the solution of a problem
- Between 400 and 300 B.C., the ancient Greek mathematician invented: Euclid's algorithm for calculating the greatest common divisor of two integers.

.

- *"An **algorithm** is a series of step-by-step instructions that aim to solve a particular problem. "*
- *"A finite set of instruction that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems is called an algorithm."*
- *"An algorithm is an abstraction of a program to be executed on a physical machine (model of computation)."*
- *"An algorithm is defined as set of instructions to perform a specific task within finite no. of steps."*

An Algorithm is a finite sequence of instructions or steps (i.e. inputs) to achieve a particular task. All algorithms must satisfy the following criteria (**Properties of Algorihm**)

1. Input	There are zero or more quantities, which are externally supplied;
2. Output	At least one quantity is produced;
3. Definiteness	Each instruction must be clear and unambiguous;
4. Finiteness	If we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;
5. Effectiveness	Every instruction must be sufficiently basic i.e. simple instruction. It is not enough that each operation be definite, but it must also be feasible.

Algorithms can be considered to be procedural solutions to problems. There are certain steps to be followed in designing and analyzing an algorithm

1. Understanding the problem:

The problem given should be understood completely. Check if it is similar to some standard problems and if a Known algorithm exists, otherwise a new algorithm has to be devised.

2. Ascertain the capabilities of the computational device:

Once a problem is understood we need to know the capabilities of the computing device this can be done by knowing the type of the architecture, speed and memory availability.

3. Exact /approximate solution:

Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs.

4. Deciding data structures :

Data structures play a vital role in designing and analyzing the algorithms. Some of the algorithm design techniques also depend on the structuring data specifying a problem's instance.

Algorithm + Data structure = Programs

5. Algorithm design techniques:

Creating an algorithm is an art which may never be fully automated. By mastering these design strategies, it will become easier for you to devise new and useful algorithms.

6. Prove correctness:

- ✓ Correctness has to be proved for every algorithm. For some algorithms, a proof of correctness is quite easy; for others it can be quite complex.
- ✓ A technique used for proving correctness by mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
- ✓ But we need one instance of its input for which the algorithm fails. If it is incorrect, redesign the algorithm, with the same decisions of data structures design technique etc

7. Analyze the algorithm

There are two kinds of algorithm efficiency: time and space efficiency. Time efficiency indicates how fast the algorithm runs; space efficiency indicates how much extra memory the algorithm needs.

8. Coding

- ✓ Programming the algorithm by using some programming language. Formal verification is done for small programs. Validity is done by testing and debugging.
- ✓ Inputs should fall within a range and hence require no verification.
- ✓ Some compilers allow code optimization which can speed up a program by a constant factor whereas a better algorithm can make a difference in their running time.
- ✓ The analysis has to be done in various sets of inputs.

Approaches of Algorithm:

The following are the approaches used after considering both the theoretical and practical importance of designing an algorithm:

- ➢ **Brute force algorithm:** The general logic structure is applied to design an algorithm. It is also known as an exhaustive search algorithm that searches all the possibilities to provide the required solution. Such algorithms are of two types:
  - ✓ **Optimizing:** Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate if the best solution is known.
  - ✓ **Sacrificing:** As soon as the best solution is found, then it will stop.
- ➢ **Divide and conquer:** It is a very implementation of an algorithm. It allows you to design an algorithm in a step-by-step variation. It breaks down the algorithm to solve the problem in

different methods. It allows you to break down the problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.

- **Greedy algorithm:** It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution. It is easy to implement and has a faster execution time. But, there are very rare cases in which it provides the optimal solution.
- **Dynamic programming:** It makes the algorithm more efficient by storing the intermediate results. It follows five different steps to find the optimal solution for the problem:
    1. It breaks down the problem into a sub problem to find the optimal solution.
    2. After breaking down the problem, it finds the optimal solution out of these sub problems.
    3. Stores the result of the sub problems is known as memorization.
    4. Reuse the result so that it cannot be recomputed for the same sub problems.
    5. Finally, it computes the result of the complex program.
- **Branch and Bound Algorithm:** The branch and bound algorithm can be applied to only integer programming problems. This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.
- **Randomized Algorithm:** As we have seen in a regular algorithm, we have predefined input and required output. Those algorithms that have some defined set of inputs and required output, and follow some described steps are known as deterministic algorithms. What happens that when the random variable is introduced in the randomized algorithm?. In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.
- **Backtracking:** Backtracking is an algorithmic technique that solves the problem recursively and removes the solution if it does not satisfy the constraints of a problem.

Pseudo code for expressing algorithms:

Pseudo code is a term which is often used in programming and algorithm based fields. It is a methodology that allows the programmer to represent the implementation of an algorithm. Simply, we can say that it's the cooked up representation of an algorithm

Advantages of Pseudo code

- Improves the readability of any approach. It's one of the best approaches to start implementation of an algorithm.
- Acts as a bridge between the program and the algorithm or flowchart. Also works as a rough documentation, so the program of one developer can be understood easily when a pseudo code is written out. In industries, the approach of documentation is essential. And that's where a pseudo-code proves vital.
- The main goal of a pseudo code is to explain what exactly each line of a program should do, hence making the code construction phase easier for the programmer.
- Let us first understand the conventions used for writing an algorithm using pseudo-code.

1. Algorithm is a procedure consisting of heading and body. The heading consists of name of the procedure and parameter list. The syntax is

```
        Algorithm name_of_procedure (parameter1, parameter2 ...parametern)
```

2. Single line comments are written using // as beginning of comment
3. The delimiters **;** are used at the end of each statement.
4. Blocks are indicated with matching braces **{** and **}**

```
{
        -------
        --------
        --------
}
```

5. The identifier should begin by letter and not by digit. An identifier can be a combination of alphanumeric string.
    - It is not necessary to write data types explicitly for identifiers. It will be represented by the context itself.
    - Basic data types used are integer, float, char, boolean and so on. The pointer type is also used to point memory location.
    - The compound data type such as structure or record can also be used.

6. Assignment of values to variables is done using the assignment statement.

   <Variable> := <expression>;

Example:

```
Node = Record
{
            datatype_1 data_1;
            . .
            . .
            datatype_n data_n;
            node *link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

7. There are two Boolean values TRUE and FALSE.

   Logical Operators **AND, OR, NOT**

   Relational Operators **<, ≤, >, ≥, =, ≠**

8. The input and output can be done using read and write.
9. A conditional statement has the following forms.

- **if statement:**

  syntax:

```
if <condition> then
   <statement>
```

- **if-else statement:**

  syntax:

```
if <condition> then
   <statement_1>
else
   <statement_2>
```

- **case statement:**

  syntax:

```
case{
      :<condition_1>: <statement_1>
                  .
                  .
      :<condition_n>: <statement_n>
      :else: <statement_n+1>
}
```

10. The following looping statements: while, repeat-until and For.

- **While Loop:**

  syntax:

```
while (condition) do
{
      statement_1
      .
      .
      statement_n
}
```

- **repeat-until:**

  syntax:

```
repeat{
      statement_1
      .
      .
      statement_n
}until (condition)
```

- **For Loop:**

  syntax:

```
for variable: = starting_value to target_value step step_count do
{
      statement_1
      .
      .
      statement_n
}
```

11. Elements of multidimensional arrays are accessed using [and]. example: **A[i,j]**

Example-1: Pseudo code to perform addition

```
Algorithm Add(a, b)
   // Step 1: Calculate the sum of a and b
   c := a + b

   // Step 2: Return the result (sum)
   Return c
End
```

Example-2: Pseudo code to finds and returns the maximum of n given numbers

```
Function Max(numbers, n)

  max := numbers[0]  // Initialize max with the first element of the list

  for i = 1 to n - 1

    if numbers[i] > max then

      max := numbers[i]  // Update max if a larger number is found

    end if

  end for

  return max  // Return the maximum value
```

Example-3 : Pseudo code to finds and returns the sum of n given numbers

```
Function SumArray(arr, n)
  sum := 0  // Initialize sum to 0

  for i = 0 to n - 1
    sum := sum + arr[i]  // Add each element of the array to sum
  end for

  return sum  // Return the total sum
```

Example 4: Write an algorithm to check whether given number is even or odd

```
Algorithm eventest  (val)
   {
      if (val%2==0) then
            Write ("given number is even");
      else
            Write ("given number is odd");
    }
```

Performance Analysis

Algorithm Analysis:

The algorithm can be analyzed in two levels, i.e., first is before creating the algorithm, and second is after creating the algorithm. The following are the two analysis of an algorithm:

- **Priori Analysis**: Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm. Various factors can be considered before implementing the algorithm like processor speed, which has no effect on the implementation part.
- **Posterior Analysis**: Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing the algorithm using any programming language. This analysis basically evaluate that how much running time and space taken by the algorithm.

❖ Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

- ✓ Space required to complete the task of that algorithm (Space Complexity).
- ✓ Time required to complete the task of that algorithm (Time Complexity)

Space Complexity:

Definition:

The space complexity of an algorithm is the amount of memory it needs to execute and produce the result.

The Space needed by each of these algorithms is seen to be the sum of the following component.

1. A fixed part that is independent of the characteristics (eg:number,size) of the inputs and outputs. The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on. It is also known as **Constant space complexity**
2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space. It is also known as **Linear space complexity**

The space requirement s(p) of any algorithm p may therefore be written as,

S(P) = c+ Sp(Instance characteristics)

Where ‘c’ is a constant.

Example of Constant Space Complexity

```
Algorithm abc(a,b,c)
{
  return a + b * c + (a+b-c) / (a+b) + 4.0;
}
```

- The problem instance is characterized by the specific values of a,b, and c.

- Making the assumption that one word is adequate to store the values of each of a,b,c and the result, so word count is constant that is 3.
- There is no dependent Instance characteristics so Sp=0.

  S(P) = 3+0 = 3

Example of Linear Space Complexity

```
Algorithm sum(a,n)
{
   s:=0.0;
   for i:=1 to n do
             s:= s+a[i];
   return s;
}
```

- The problem instances for Algorithm characterized by n, the number of elements to be summed.
- The space needed by n is one word; since it is of type integer.
- The space needed by a is the space needed by variables of type array of floating point numbers.
- This is at least n words,
- Since a must be large enough to hold the n elements to be summed.
- So,we obtain S_{sum} (n) >=(n+3) (**n for a[],one each for n,i and s**)

Example of Recursiv Space Complexity

```
Algorithm R_Sum(a,n)
{
if (n < 0) then return 0.0;
else return R_Sum (a,n-1)+ a[n];
 }
```

1. Let us consider the algorithm RSum .As in the case of Sum, the instances are characterized by n.
2. The recursion stack space includes space for the formal parameters, the local variables, and the return address .
3. Assume that the return address requires only one word of memory.
4. Each call to R_{Sum} requires at least three words (including space for the values of n, the return address, and a pointer to a[]).

Since the depth of recursion is n + 1,the recursion stack space needed is >=**3(n+ 1)**

Time Complexity

Definition:

The time complexity of an algorithm is the amount of computer time it needs to execute and produce the result.

- The time T(p) taken by a program P is the **sum of the compile time and the run time**(execution time). The compile time does not depend on the instance characteristics.
- Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because the configuration changes from one system to another system.
- To solve this problem, we must assume a model machine with a specific configuration.
- So that, we can able to calculate generalized time complexity according to that model machine.

To calculate the time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. It is a Single processor machine
2. It is a 32 bit Operating System machine
3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways

1. Step count:

- We introduce a variable, count into the program statement to increment count with initial value 0.
- Statement to increment count by the appropriate amount are introduced into the program.
- This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

Example-1:

```
Algorithm sum(A,n)
{
        s:= 0.0;
        count := count+1; // count is global; it is initially zero
        for i:=1 to n do
        {
                count := count+1; //for for loop
```

```
                  s:=s+A[i];
                  count=count+1;  //for assignment
         }
         count=count+1;  //for last time of for
         count=count+1;  //for the return
         return s;
}
```

If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum executes a total of **2n+3 steps.**

Now adding the steps from initialization and the return statement:

1. Initialization of s: 1 step.
2. It is important to note that the frequency of the **for** statement is n + 1and not n. This is so because i has to be incremented to **n+1** before the for loop can terminate
3. N time's assignment: n steps.
4. Step 3 (Return statement): 1 step.

Thus, the total number of steps becomes:

Total Steps= 1 +2n+1+1 =2n+3

Example-2:

```
Algorithm Rsum(A,n)
{
         count := count+1;     //for if condition
         if(n ≤ 0) then
           count := count+1;  //for return
           retrun 0;
         else
           count := count+1;  //for return
           return Rsum(A,n-1)+A[n];
}
```

$$t_{\text{Rsum}}(n) = \begin{cases} 2 & if\, n = 0 \\ 2 + t_{\text{Rsum}}(n\text{-}1) & if\, n > 0 \end{cases}$$

Recursive formulas are referred to as **Recurrence relations**

To solve this, make repeated substitutions.

$$= 2 + t_{\text{Rsum}}(n\text{-}1)$$
$$= 2 + 2 + t_{\text{Rsum}}(n\text{-}2)$$
$$= 4 + 2 + t_{\text{Rsum}}(n\text{-}3)$$

$$= n(2) + t_{\text{Rsum}}(0)$$

$$= 2n + 2$$

2. Frequency Count Method:

- The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.
- First determine the number of **steps per execution(s/e)** of the statement and the total number of times (ie., frequency) each statement is executed.
- By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

statement	S/e	Frequency	Total
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. s:=0.0;	1	1	1
4. for i:=1 to n do	1	n+1	n+1
5. s:=s+a[i];	1	n	n
6. return s;	1	1	1
7.}	0	-	0
Total			2n+3

Statement	S/e	Frequency		Total	
		n=0	n>0	n=0	n>0
1. Algorithm Rsum(A,n)	0	-	-	0	0
2. {	0	-	-	0	0
3. if(n ≤ 0) then	1	1	1	1	1
4. retrun 0;	1	1	0	1	0
5. else	0	-	-	0	0
6. return Rsum(A,n-1)+A[n];	1+x	0	1	0	1+x
7. }	0	-	-	0	0
Total				2	2+x

$$x = t_{\text{Rsum}}(n-1)$$

Asymptotic Notation

Complexity of Algorithms:

- The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'.
- Complexity shall refer to the running time of the algorithm.
- The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. **Best Case** : The minimum possible value of f(n) is called the best case.
2. **Average Case** : The expected value of f(n).
3. **Worst Case** : The maximum value of f(n) for any key possible input.

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

Big 'oh'(O)- worst case:

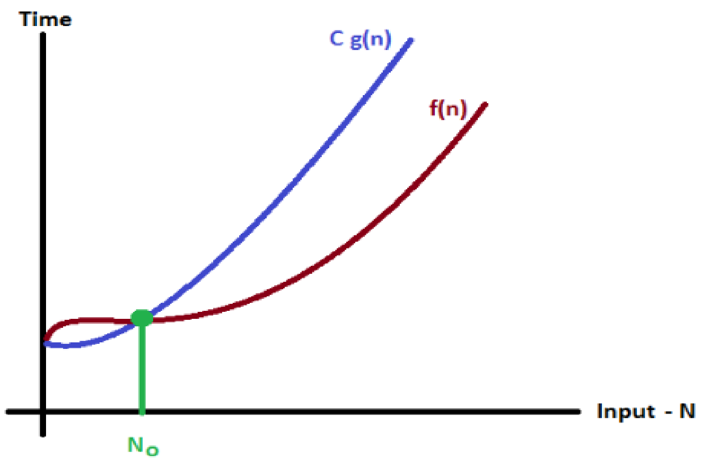
The notation O(n) is the formal way to express the **upper bound of an algorithm's** running time. It is the most commonly used notation. It measures the **worst case time complexity** or the longest amount of time an algorithm can possibly take to complete.

Definition

A function f(n) can be represented is the order of g(n) that is O(g(n)), if there exists a value of positive integer n as n_0 and a positive constant c such that –

$f(n) \leq c^*g(n)$ for all $n \geq n_0$.

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



- Then f(n) = O(g(n)) as growth of g(n) is faster than that of f(n).
- f(n) = O(g(n)) is read as f(n) is equal to Big Oh of g(n).
- Mathematically, we can say that $f(n) \leq c.g(n)$ for constants c and n0 where c>0 and n>n0.
- This implies that the function g(n) is an upper bound on the function f(n) i.e. f(n) cannot grow faster than g(n).

Let us try to understand this concept with the help of an example:

Ex-1: **3n+2 = O(n)**

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent f(n) as O(g(n)) then it must satisfy **f(n) <= C g(n)** for all values of C > 0 and n_0>= 1

f(n) <= C g(n)

⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 4 and n >= 2.**

By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

Ex-2: **$n^2 + n = O(n^3)$**

Proof:

- Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$
- Notice that if $n \geq 1$, $n \leq n^3$ is clear.
- Also, notice that if $n \geq 1$, $n^2 \leq n^3$ is clear.
- **Therefore**, $n^2 + n \leq n^3 + n^3 = 2n^3$
- We have just shown that $n^2 + n \leq 2n^3$ for all $n \geq 1$
- Thus, we have shown that $n^2 + n = O(n^3)$
- (by definition of Big- O, with **n_0 = 1, and c = 2**.)

Note: In general, if $a \leq b$, then $na \leq nb$ whenever $n \geq 1$. This fact is used often in these types of proofs.

As illustrated by the previous examples, the statement f(n) = O(g(n)) states only that g(n) is an upper bound on the value of f(n) for all n, n ≥ no. It does not say anything about how good this bound is.

Order	Name of Time Complexity	Examples
O(1)	Constant	Few algorithms without any loops
O(log n)	Logarithmic	Searching algorithms
O(n)	Linear	Algorithms that scan a list of size n
O(n log n)	n-log-n	Divide and conquer algorithms
$O(n^2)$	Quadratic	Operations on n by n matrices
$O(n^3)$	Cubic	Nontrivial algorithms from linear algebra
$O(2^n)$	Exponential	Algorithms that generate all subsets of a set
O(n!)	Factorial	Algorithms that generate permutations of a set

If algorithm is run on same complexity on same type of data, but with higher magnitude of n, the resulting time is less than some constant time f(n).

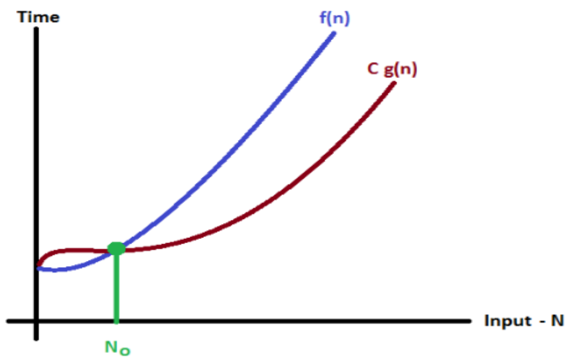
Thus: **$O(1) < O(lon\ n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$**

Big Omega(Ω)- **best-case**

- This notation is denoted by 'Ω', and it is pronounced as '**Big Omega**'. Big Omega notation defines lower bound for the algorithm.
- It means the running time of algorithm cannot be less than its asymptotic **lower bound** for any random sequence of data.

Definition:

The function f(n) = Ω(g(n)) iff there exist positive constants c and n_0 such that **f(n) ≥ c*g(n)** for all n ≥ n_0.



Examples:

1. $3n+1 = \Omega(n)$

   as $3n+1 \geq 3n$ for all $n \geq 1$.

   $c = 3$ & $n_0 = 1$

2. $3n+4 = \Omega(n)$

   as $3n+4 \geq 3n$ for all $n \geq 1$.

   $c = 3$ & $n_0 = 1$

- Big Oh and Big Omega notations provide **the upper and lower running bounds of the algorithm**. These measures tell us that the running time of an algorithm cannot be more or less than their respective bounds.
- Big Oh notation provides the **worst-case** running time, whereas Big Omega provides the **best-case** running time of the algorithm.
- The running time of the algorithm cannot be better than Big Omega and it cannot be worse than Big Oh.

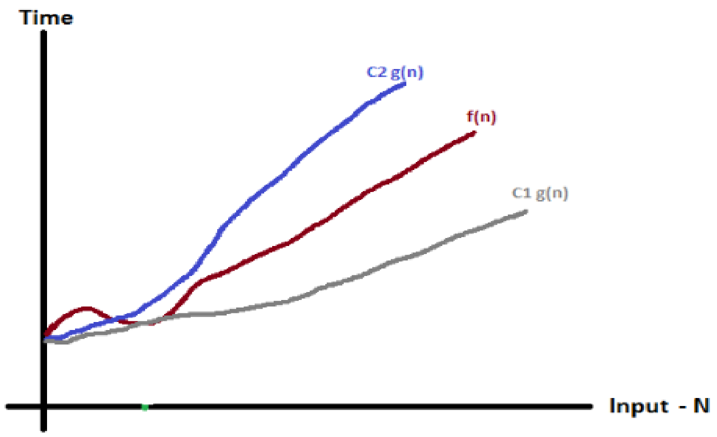
Big Theta(Θ)- **average-case**

This notation is denoted by 'Θ', and it is pronounced as "Big Theta". Big Theta notation defines tight bound for the algorithm. It means the running time of algorithm is **between** upper bound and lower bound. It is used for analyzing the **average-case** complexity of an algorithm.

Definition

The function $f(n)=\Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that

$c_1*g(n) \leq f(n) \leq c_2*g(n)$ for all $n \geq n_0$.



The theta notation is more precise that both the big oh and omega notations. The function $f(n) = \Theta(g(n))$ iff g(n) is both an upper and lower bound of f(n).

Examples:

1. $3n+2 = \Theta(n)$

   as $3n+2 \geq 3n$ and $3n+2 \leq 4n$, for $n \geq 2$.

   $c_1=3$, $c_2=4$, and $n_0=2$

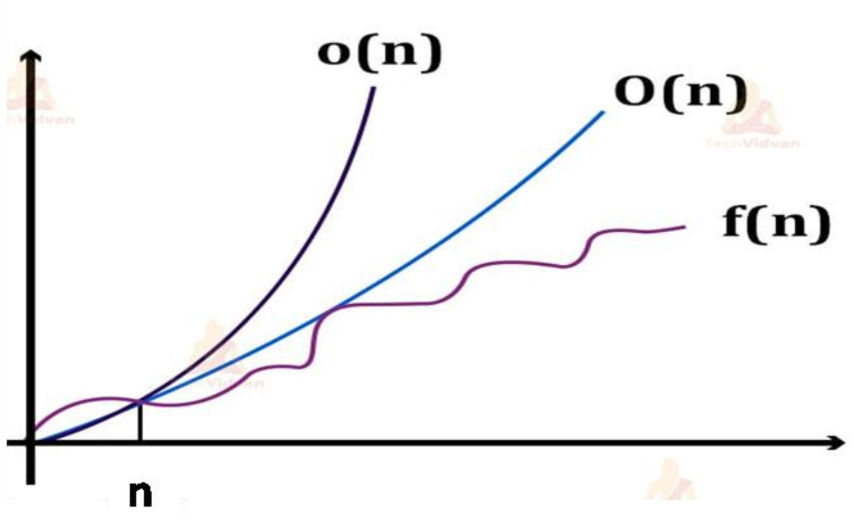
Important Notes

- For analysis (best case, worst case and average) we try to give upper bound (O) and lower bound (Ω) and average running time (Θ).
- From the above examples, it should also be clear that, for a given function (algorithm) getting upper bound (O) and lower bound (Ω) and average running time (Θ) **may not be possible always**.
- **For example,** if we are discussing the best case of an algorithm, then we try to give upper bound (O) and lower bound (Ω) and average running time (Θ).
- In the remaining chapters we generally concentrate on upper bound (O) because knowing lower bound (Ω) of an algorithm is of no practical importance.

Little (o):

- Big O is used as a tight upper bound on the growth of an algorithm's effort (this effort is described by the function f(n)), even though, as written, it can also be a loose upper bound.
- "Little o" (o) notation is used to describe an **upper bound that cannot be tight**.
- In the domain of algorithm analysis, the little o notation is a valuable tool used to describe the behavior of functions as they approach certain limits.

A function f(n) can be represented is the order of g(n) that is o(g(n)), if there exists a value of positive integer n as n_0 and a positive constant c such that **$f(n) < c*g(n)$ for all values of $c>0$ and for all values of n where $n \geq n_0$** ***and c and n_0 are constants***



Mathematical Relation of Little o notation

Using mathematical relation, we can say that f(n) = o(g(n)) means,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Example -1:

If f(n) = n^2 and g(n) = n^3 then check whether f(n) = o(g(n)) or not.

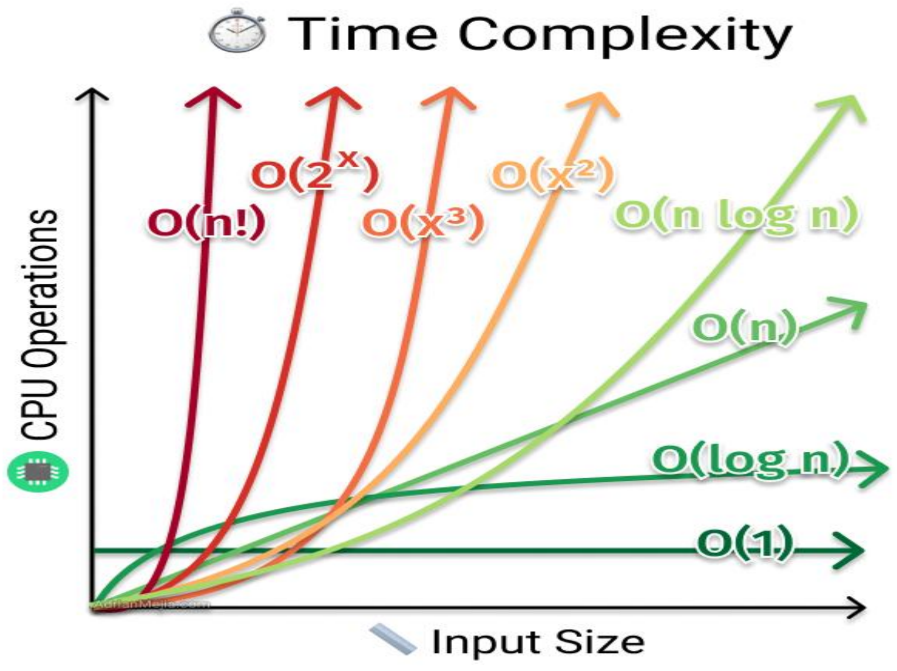
$$\lim_{n\to\infty} \frac{n^2}{n^3}$$

$$= \lim_{n\to\infty} \frac{1}{n}$$

$$= \frac{1}{\infty}$$

$$= 0$$

The result is 0, and it satisfies the equation mentioned above. So we can say that f(n) = o(g(n))



Disjoint Sets- disjoint set operations, union and find operations.

- The disjoint set data structure is also known as union-find data structure and merge-find set. It is a data structure that contains a collection of disjoint or non-overlapping sets.
- The disjoint set means that when the set is partitioned into the disjoint subsets. The various operations can be performed on the disjoint subsets.
- In this case, we can add new sets, we can merge the sets, and we can also find the representative member of a set. It also allows to find out whether the two elements are in the same set or not efficiently..

Disjoint Set Operations:

The disjoints sets are those do not have any common element.

For example S1={1,7,8,9} and S2={2,5,10}, then we can say that S1 and S2 are two disjoint sets.

The disjoint set operations are

1. Union:

If Si and Sj are two disjoint sets, then their union Si U Sj consists of all the elements x such that x is in S_i or S_j.

Example:

S1={1,7,8,9} S2={2,5,10}

S1US2={1,2,5,7,8,9,10}

2. Find:

Given the element I, find the set containing I.
Example:

S1={1,7,8,9}

Then, S2={2,5,10} S3={3,4,6}

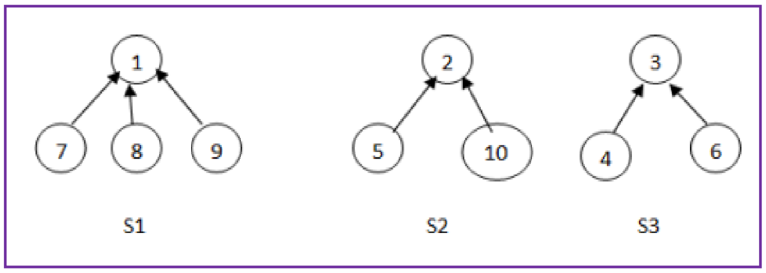
Find(4)=S3 Find(5)=S2 Find(7)=S1

Set Representation:

- The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address.
- In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

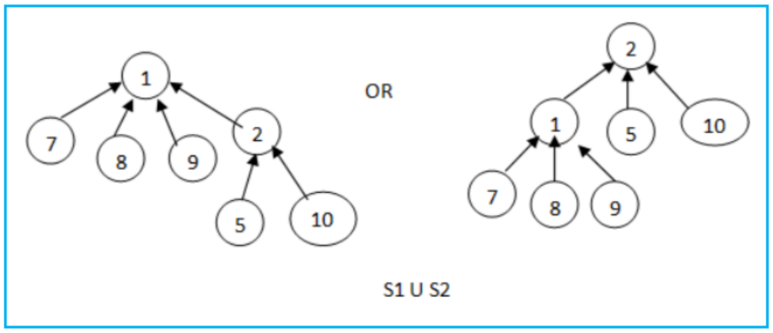
Example:
S1={1,7,8,9} S2={2,5,10} s3={3,4,6}

Then these sets can be represented as



Disjoint Union:

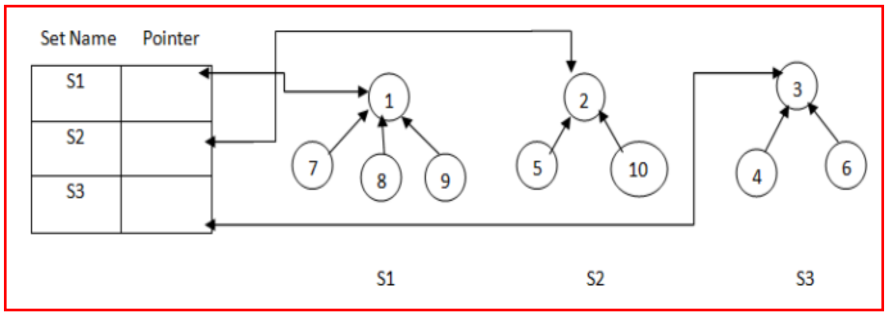
- To perform disjoint set union between two sets Si and Sj can take any one root and make it sub-tree of the other.
- Consider the above example sets S1 and S2 then the union of S1 and S2 can be represented as any one of the following



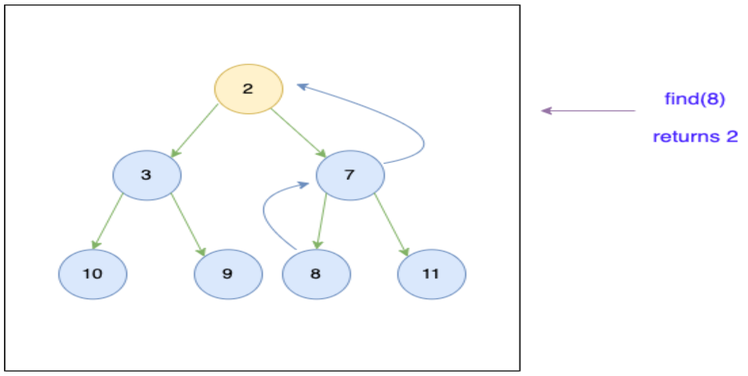
Disjoint Find:

- To perform find operation, along with the tree structure we need to maintain the **name of each set**. So, we require one more data structure to store the set names.
- The data structure contains two fields. One is the set name and the other one is the pointer to root

.



We give an element, say "x" which should be a part of one of the disjoint sets and the operation will return us the element representing that set (**Leader of that Set**).

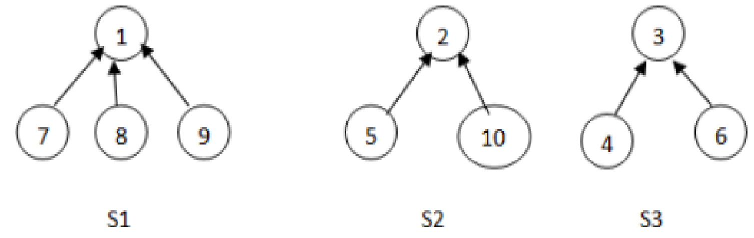


Union and Find Algorithms:

- In presenting Union and Find algorithms, we ignore the set names and identify sets just by the roots of trees representing them.
- To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets.
- The index values represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be‘-1’.

Example:

For the following sets the array representation is as shown below



I	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
P	-1	-1	-1	3	2	3	1	1	1	2

Algorithm for Union operation:

To perform union the SimpleUnion(i,j) function takes the inputs as the set roots i and j . And make the parent of i as j i.e, make the second root as the parent of first root.

```
Algorithm SimpleUnion(i,j)

{

P[j]:=i;

}
```

Algorithm for find operation:

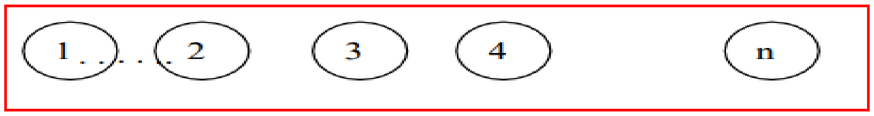
The SimpleFind(i) algorithm takes the element i and finds the root node of i. It starts at i until it reaches a node with parent value-1.

```
Algorithm SimpleFind(i)
{
while( P[i]≥0)
i:=P[i];
return i;
}
```

Analysis of SimpleUnion(i,j) and SimpleFind(i):

Although the SimpleUnion(i,j) and SimpleFind(i) algorithms are easy to state, their performance characteristics are not very good.

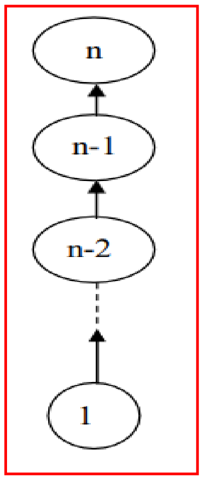
For example, consider the sets



Then if we want to perform following sequence of operations –

Union(1,2) ,Union(2,3)……. Union(n-1,n) and sequence of Find(1), Find(2)………Find(n)

The sequence of Union operations results the **degenerate tree** as below

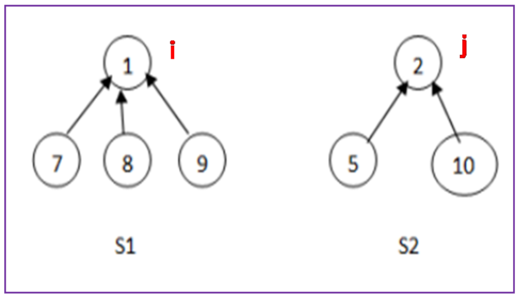


Since, the time taken for a Union is constant; the n-1 sequence of union can be processed in time O(n). And for the sequence of Find operations it will take for an element at level i of a tree is O(i),the total time needed to process the n finds is $O(\sum_{i=1}^{n} i) = O(n^2)$.

We can improve the performance of union and find by avoiding the creation of degenerate tree by applying **weighting rule** for Union(i,j).

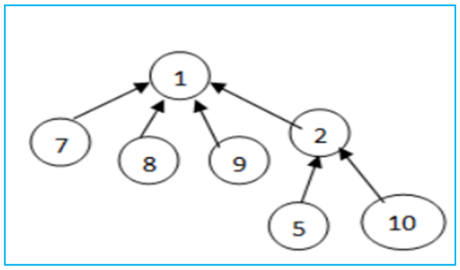
Weighting rule for Union:

- If the number of nodes in the tree with root **i** is less than the number in the tree with the root j, then make 'j' the parent of i; otherwise make 'i' the parent of j.
- To implement weighting rule we need to know how many nodes are there in every tree.
- To do this we maintain "count" field in the root of every tree. If **'i'** is the root then count[i] equals to number of nodes in tree with root **i**.
- Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in **P** field of the root as negative number.



i	1	7	8	9
P[i]	-4	1	1	1

j	2	5	10
P[j]	-3	2	2



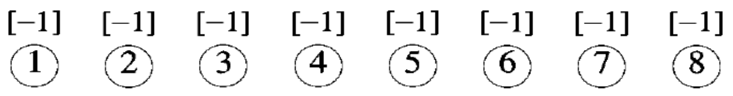
i	1	7	8	9	2	5	10
P[i+j]	-7	1	1	1	1	2	2

```
Algorithm WeightedUnion(i,j)
  //Union sets with roots i and j, i≠j (disjoint sets) using the weighted rule
  // P[i]=-count[i] and p[j]=-count[j]
  {
  temp:=P[i]+P[j];
  if (P[i]>P[j]) then
  {
  // i has fewer nodes
  P[i]:=j;
  P[j]:=temp;
  }
  else
  {
  // j has fewer nodes
  P[j]:=i;
  P[i]:=temp;
  }
 }
```

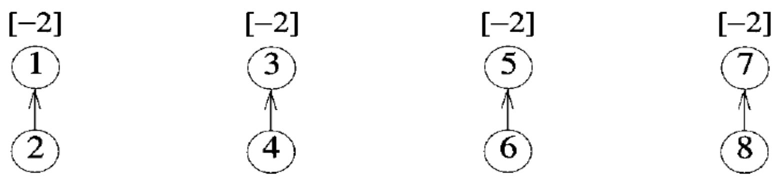
- ❖ In this algorithm the time required to perform a union has increased some what but is still bounded by a constant (that is, it is O(1)). The find algorithm remains unchanged
- ❖ Further improvement is possible. This time the modification is made in the find algorithm using the **collapsing rule.**

Collapsing rule for find:

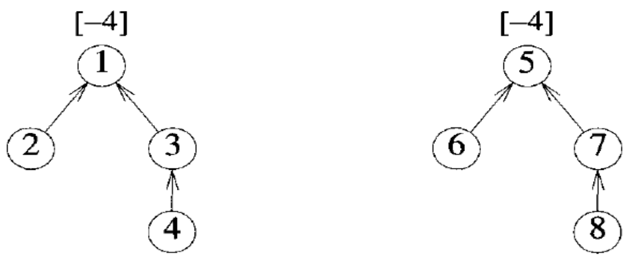
- If j is a node on the path from i to its root and p[i]≠root[i], then set P[j] to root[i].
- Consider the tree created by WeightedUnion() on the sequence of1≤i≤8.
- Union(1,2), Union(3,4), Union(5,6) and Union(7,8)



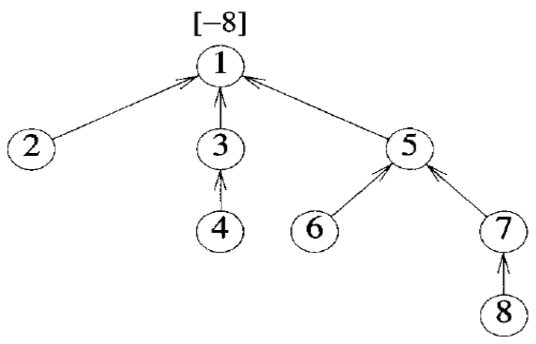
(a) Initial height-1 trees



(b) Height-2 trees following *Union*(1,2), (3,4), (5,6), and (7,8)



(c) Height-3 trees following *Union*(1,3) and (5,7)

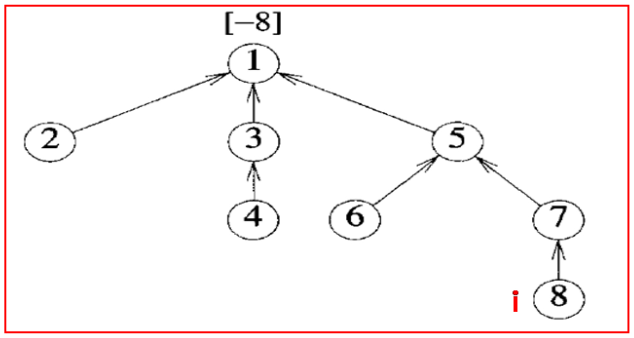


(d) Height-4 tree following *Union*(1,5)

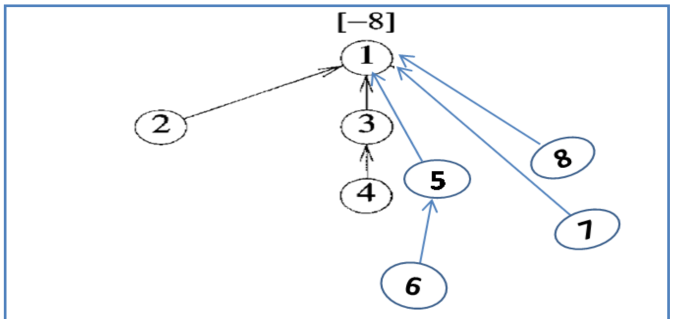
Now process the following **eight find** operations

Find(8),Find(8)................................. Find(8)

- If SimpleFind() is used each Find(8) requires going up **three** parent link fields for a total of 24 moves.
- When Collapsing find is used the first Find(8) requires going up three links and resetting three links. Each of remaining seven finds require going up only one link field.
- Then the total cost is now only 13 moves.(3 going up + 3 resets + 7 remaining finds)



i	1	2	3	4	5	6	7	8
P[i]	-8	1	1	3	1	5	5	7



i	1	2	3	4	5	6	7	8
P[i]	-8	1	1	3	1	5	1	1

```
Algorithm CollapsingFind(i) // Find the root of the tree containing element i. Use the

// collapsing rule to collapse all nodes from i to the root .
{
            r := i;
            while (p[r] >0) do
                    r := p[r]; / Find the root,
               while (i!=r) do // Collapse nodes from i to root r,
               {
                           s:=p[i];
                            p[i]:=r;
                           i:=s;
               }
}
```

Divide and conquer

Divide and Conquer Algorithm is a problem-solving technique used to solve problems by dividing the main problem into sub problems, solving them individually and then merging them to find solution to the original problem. In this article, we are going to discuss how Divide and Conquer Algorithm is helpful and how we can use it to solve problems.

Working of Divide and Conquer Algorithm:

Divide and Conquer Algorithm can be divided into three steps: **Divide**, **Conquer** and **Merge** .

1. Divide:

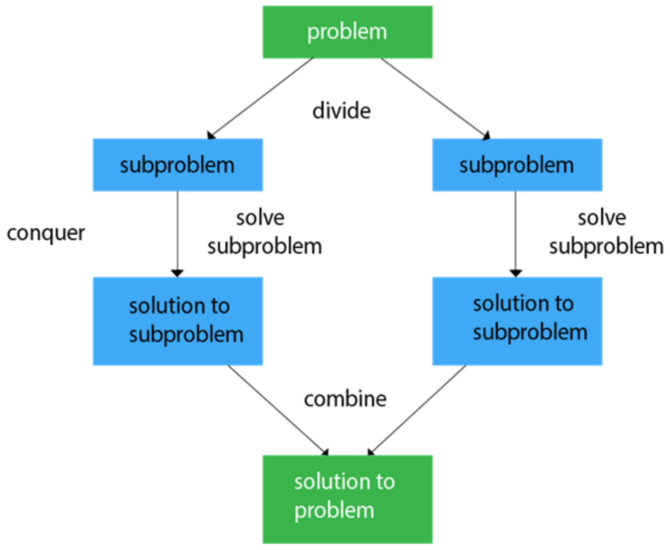
- Break down the original problem into smaller sub problems.
- Each sub problem should represent a part of the overall problem.
- The goal is to divide the problem until no further division is possible.

2. Conquer:

- Solve each of the smaller sub problems individually.
- If a subproblem is small enough (often referred to as the "base case"), we solve it directly without further recursion.
- The goal is to find solutions for these sub problems independently.

3. Merge:

- Combine the sub-problems to get the final solution of the whole problem.
- Once the smaller sub problems are solved, we recursively combine their solutions to get the solution of larger problem.
- The goal is to formulate a solution for the original problem by merging the results from the sub problems



General Method

- Divided into smaller sub problems.
- These sub problems are solved independently.
- Combining all the solutions of sub problems into a solution of the whole.
- If the sub problems are still relatively large, then divide-and-conquer is reapplied. The generated sub problems are usually of same type as the original problem. Hence recursive algorithm are used.

Algorithm:

```
Algorithm (P)
{
        if small(P) then
                return S(P);
        else
        {
                divide P into smaller instances     P1, P2 .. Pk, k>=1;
                Apply DAndC to each of these sub problems;
                return combine (DAndC(P1), DAndC(P2),....,DAndC(Pk));
        }
}
```

- small (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting.
- If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p1, p2, . . . , pk are solved by recursive application of DAndC .
- Combine is a function that determines the Solution to P using the solutions to the k sub problems. If the size of P is n and the sizes of the k sub problems are $n_1,n_2,....,n_k$ respectively, then the computing time of DAndC is described by the recurrence relation

The computing time of above procedure of divide and conquer is given by the recurrence relation.

$$T(n) = \begin{cases} g(n) & \text{n small} \\ T(n_1) + T(n_2) + \dots + T(n_k) + f(n) & \text{otherwiser} \end{cases}$$

Where T(n) is the time for divide and conquer of size n. The g(n) is the computing time required to solve small inputs. The F(n) is the time required in dividing problem P and combining the solution to sub problems.

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) \begin{cases} T(n) = T(1) \text{ if } n=1 \\ aT(n/b)+f(n) \text{ if } n>1 \end{cases}$$

Where:

- $T(n)$ is the time (or cost) to solve the problem of size n,
- a is the number of subproblems in the recursive division,
- $\frac{n}{b}$ is the size of each subproblem (i.e., the problem size is divided by b),
- $f(n)$ is the cost of dividing the problem and combining the results of the subproblems (i.e., the "work" outside of recursion).

Example: Consider the case in which a = 2 and b = 2. Let T(1)=2 and f(n) = n.

We have one of the methods for solving any such recurrence relation is called the **substitution method**. This method repeatedly makes substitution for each occurrence of the function Tin the right-hand side until all such occurrences disappear.

Base Case:
T(1)= 2 specifies the solution for the smallest possible input, where the problem size is 1.

In other words, when the problem size is 1, the cost or number of operations is 2.

The general recurrence relation for a Divide and Conquer algorithm is:

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

For this specific case, we have:

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

Step 1: Start with the initial recurrence

The recurrence is:

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

We are going to expand this recurrence using the **substitution method**.

Step 2: Expand the recurrence once

We first substitute $T\left(\frac{n}{2}\right)$ with its recurrence relation:

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

Substitute this back into the original recurrence:

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

Simplify the equation:

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

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

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

Base Case:
f(n)= n specifies the represents the work done outside of the recursive calls at each level.

Interpretation:
f(n)=n means that at each level of recursion, the non-recursive work (such as **dividing the problem, merging results, or doing some other work**) takes n operations.

Step 3: Expand the recurrence a second time

Now expand $T\left(\frac{n}{4}\right)$:

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$$

Substitute this back:

$$T(n) = 2^2\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$$

Simplifying:

$$T(n) = 2^3T\left(\frac{n}{8}\right) + 2^2 \cdot \frac{n}{4} + 2n$$

$$T(n) = 2^3T\left(\frac{n}{8}\right) + n + 2n$$

$$T(n) = 2^3T\left(\frac{n}{8}\right) + 3n$$

Step 4: Generalize the recurrence

By continuing the process, we see the following pattern forming:

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

At each step, the exponent of 2 increases, and the total cost for merging increases linearly with k.

Step 5: Base case

The recursion stops when the subproblem size becomes 1, i.e., when:

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

Solving for k:

$$2^k = n \quad \Rightarrow \quad k = \log_2 n$$

So, after $k = \log_2 n$ expansions, the recursion will reach the base case.

Step 6: Final substitution

Now, substitute $k = \log_2 n$ into the general formula for $T(n)$:

$$T(n) = 2^{\log_2 n}T(1) + n \cdot \log_2 n$$

Since $2^{\log_2 n} = n$, we get:

$$T(n) = n \cdot T(1) + n \cdot \log_2 n$$

Using the base case value $T(1) = 2$:

$$T(n) = n \cdot 2 + n \cdot \log_2 n$$

Thus:

$$T(n) = 2n + n\log_2 n$$

Final Time Complexity

The dominant term in the expression $2n + n\log_2 n$ is $n\log_2 n$. So, the time complexity of this recurrence is:

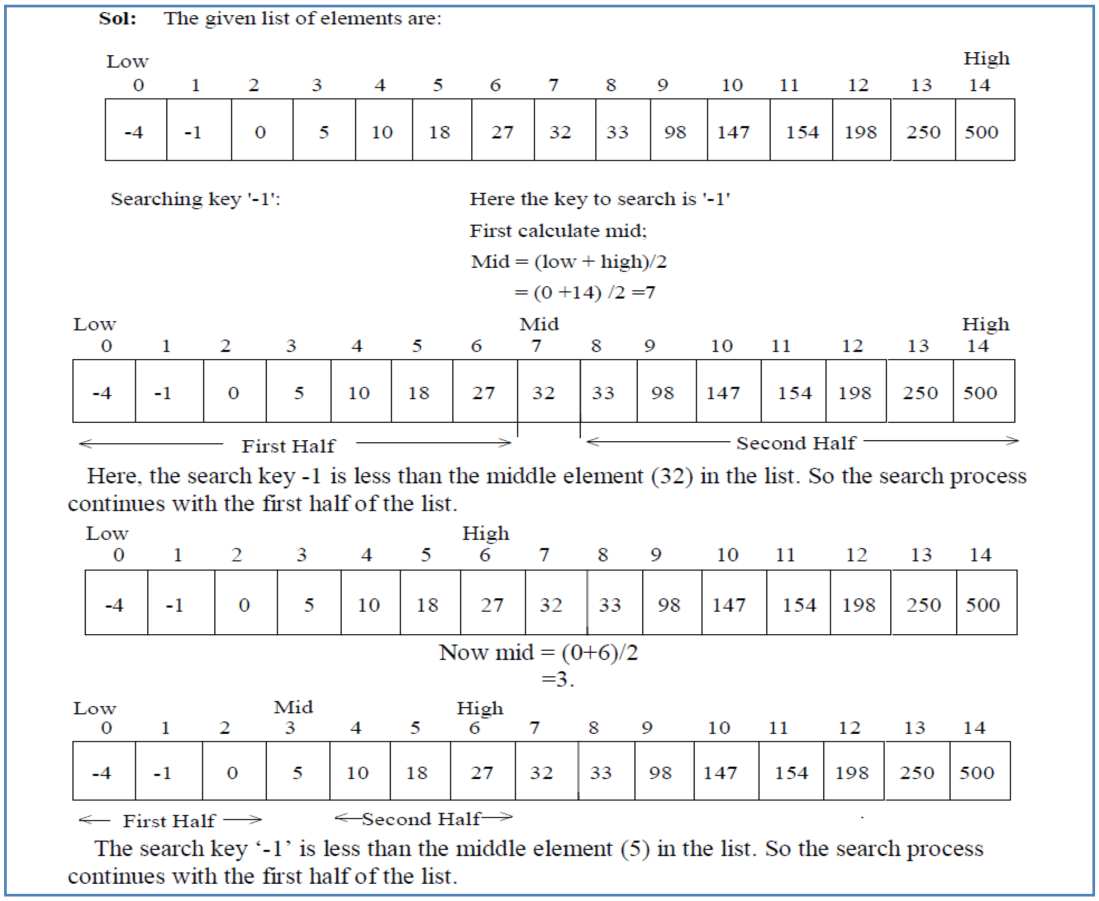
$$T(n) = O(n \log n)$$

Binary Search:

- If we have ‘n’ records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element ‘x’, binary search is used to find the corresponding element from the list. In case ‘x’ is present, we have to determine a value ‘j’ such that a[j] = x (successful search). If ‘x’ is not in the list then j is to set to zero (un successful search).
- In Binary search we jump into the middle of the file, where we find key a[mid], and compare ‘x’ with a[mid]. If x = a[mid] then the desired record has been found.
- If x < a[mid] then ‘x’ must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of ‘x’ with a[mid] will eliminate roughly half the un-searched portion from consideration.
- Since the array size is roughly halved often each comparison between ‘x’ and a[mid], and since an array of length ‘n’ can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Example-1:

Consider the list of elements: -4, -1, 0, 5, 10, 18, 32, 33, 98, 147, 154, 198, 250, 500. Trace the binary search algorithm searching for the element -1.



Sol: The given list of elements are:

Low														High
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Searching key '-1': Here the key to search is '-1'

First calculate mid;

Mid = (low + high)/2

= (0 +14) /2 =7

Low							Mid							High
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

← First Half → ← Second Half →

Here, the search key -1 is less than the middle element (32) in the list. So the search process continues with the first half of the list.

Low						High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

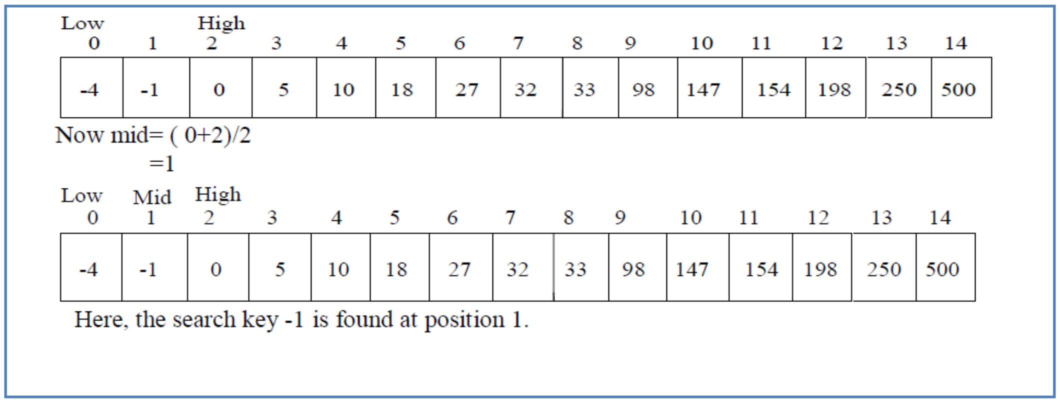
Now mid = (0+6)/2

=3.

Low			Mid			High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

← First Half → ← Second Half →

The search key ‘-1’ is less than the middle element (5) in the list. So the search process continues with the first half of the list.



Low 0	1	High 2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid= (0+2)/2
=1

Low 0	Mid 1	High 2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Here, the search key -1 is found at position 1.

Algorithm Algorithm - **Recursive Binary Search**

```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i : l]$ of elements in nondecreasing
// order, $1 \leq i \leq l$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    if ($l = i$) then  // If Small($P$)
    {
        if ($x = a[i]$) then return $i$;
        else return 0;
    }
    else
    { // Reduce $P$ into a smaller subproblem.
        $mid := \lfloor (i + l)/2 \rfloor$;
        if ($x = a[mid]$) then return $mid$;
        else if ($x < a[mid]$) then
                  return BinSrch($a, i, mid - 1, x$);
              else return BinSrch($a, mid + 1, l, x$);
    }
}
```

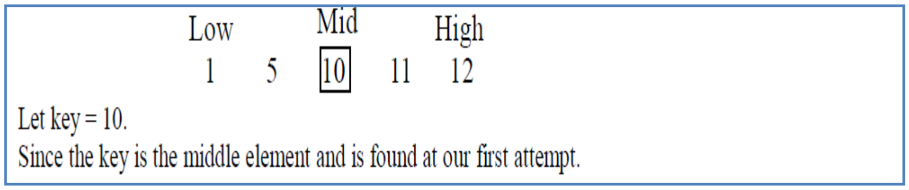
Algorithm Algorithm -**Iterative binary Search**

```
Algorithm BinSearch($a, n, x$)
// Given an array $a[1 : n]$ of elements in nondecreasing
// order, $n \geq 0$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    $low := 1$; $high := n$;
    while ($low \leq high$) do
    {
        $mid := \lfloor (low + high)/2 \rfloor$;
        if ($x < a[mid]$) then $high := mid - 1$;
        else if ($x > a[mid]$) then $low := mid + 1$;
             else return $mid$;
    }
    return 0;
}
```

Efficiency of Binary Search: To evaluate binary search, count the number of comparisons in the best case, average case, and worst case.

Best Case: The best case occurs if the middle element happens to be the key element. Then only one comparison is needed to find it. Thus the efficiency of binary search is O(1).

Ex: Let the given list is: 1, 5, 10, 11, 12.



Let key = 10.
Since the key is the middle element and is found at our first attempt.

$$T(n) = O(1)$$

Worst Case:

The worst case of Binary Search occurs when: The element is to search is in the first index or last index

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= T(n/4)+1+1 = T(n/2^2)+2 \\ &= T(n/8)+1+1+1 = T(n/2^3)+3 \\ &---- \\ &----- \\ &= T(n/2^k) + k \end{aligned}$$

Using recurrence Relation of Divide and conquer method

So $T(n)= T(n/n)+\log n$
$= 1+\log n$
$T(n)=\log n$

Therefore **$T(n)=O(\log n)$**

Let $n=2^k$

$\log n=\log 2^k$

[Apply Both side log]

$\log n=k\log 2$

$\log n=k$

$k=\log n$

Average Case: In binary search, the average case efficiency is near to the worst case efficiency. So the average case efficiency will be taken as O(log n).

$$T(n) = O(\log n)$$

The conclusion of our Time and Space Complexity analysis of Binary Search is as follows:

- Best Case Time Complexity of Binary Search: O(1)
- Average Case Time Complexity of Binary Search: O(logN)
- Worst Case Time Complexity of Binary Search: O(logN)
- Space Complexity of Binary Search: O(1) for iterative, O(logN) for recursive.

Merge Sort:

- ❖ **Merge Sort** is a popular and efficient sorting algorithm. It works on the principle of **Divide and Conquer** strategy.
- ❖ The fundamental operation in merge sort algorithm is merging two sorted lists. This is also known as **two-way merge sort**. Merge sort runs in O(n log n) worst-case running time, and the number of comparisons is nearly optimal. It is a good example of recursive algorithm.
- ❖ We assume to sort the given array **a[n]** into ascending order. We split it into two subarrays: **a[0]...a[n/2]** and **a[n/2)+1]...a[n-1]**.
- ❖ Each subarray is individually sorted, and the resulting sorted sub arrays are merged to produce a single sorted array of n elements.
- ❖ This is an **ideal** example of divide-and-conquer strategy. First, left half of the array a[0]...a[n/2] elements is being split and merged; and next second half of the array a[n/2)+1]...a[n-1] elements is processed Notice how the splitting continues until sub arrays containing a single element are produce.
- ❖ This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array)

Here's a step-by-step explanation of the Merge Sort process

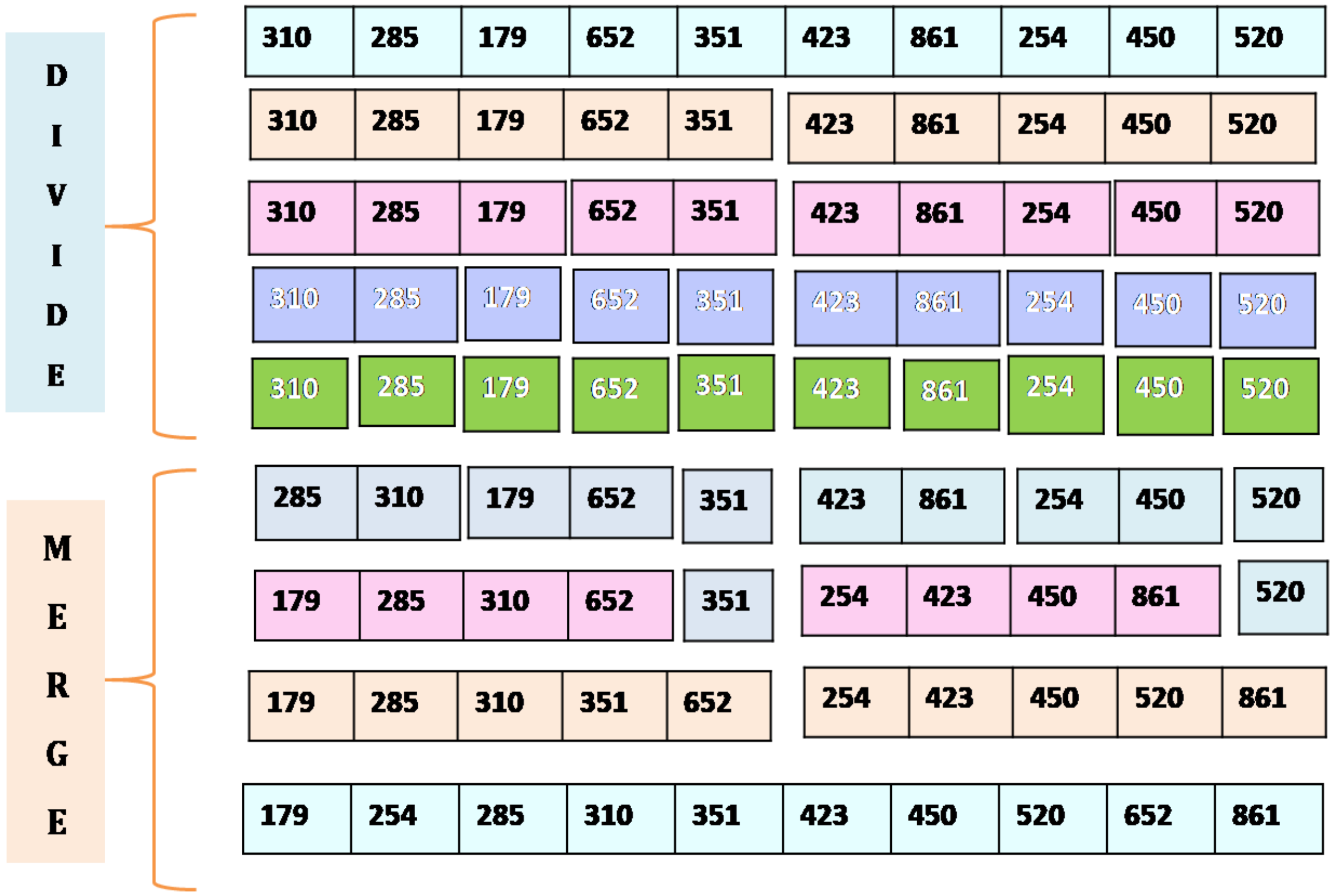
- **Divide:** The array is divided into two halves (sub-arrays) recursively until each sub-array contains a single element. This is done by finding the middle of the array using the formula

  middle=(low + high)/2

- **Conquer:** Each pair of single-element sub-arrays is merged into a sorted array of two elements; these sorted arrays are then merged into sorted arrays of four elements, and so on.
- **Merge:** Finally, all elements are merged into a single sorted array.

Consider the array often elements a[1:10]= **(310, 285,179, 652,351,423,861, 254, 450,520).**

- ✓ Algorithm Merge Sort begins by splitting a[ ] into two sub arrays each of size five (a[l:5] and a[6:10]).
- ✓ The elements in a[l:5] are then split into two sub arrays of size three(a[l:3])and two (a[4 : 5]).
- ✓ Then the items in a[1 : 3] are split into sub arrays of size two (a[l: 2]) and one (a[3 : 3]).
- ✓ The two values in a[1: 2] are split a final time into one-element sub arrays, and now the merging begins.
- ✓ Note that no movement of data has yet taken place. A record of the sub arrays is implicitly maintained by the recursive mechanism. Pictorially the file can now be viewed as



- The below tree that represents the sequence of recursive calls that are produced by Merge Sort when it is applied to ten elements.
- The pair of values in each node is the values of the parameters low and high.
- Notice how the splitting continues until sets containing a single element are produced.

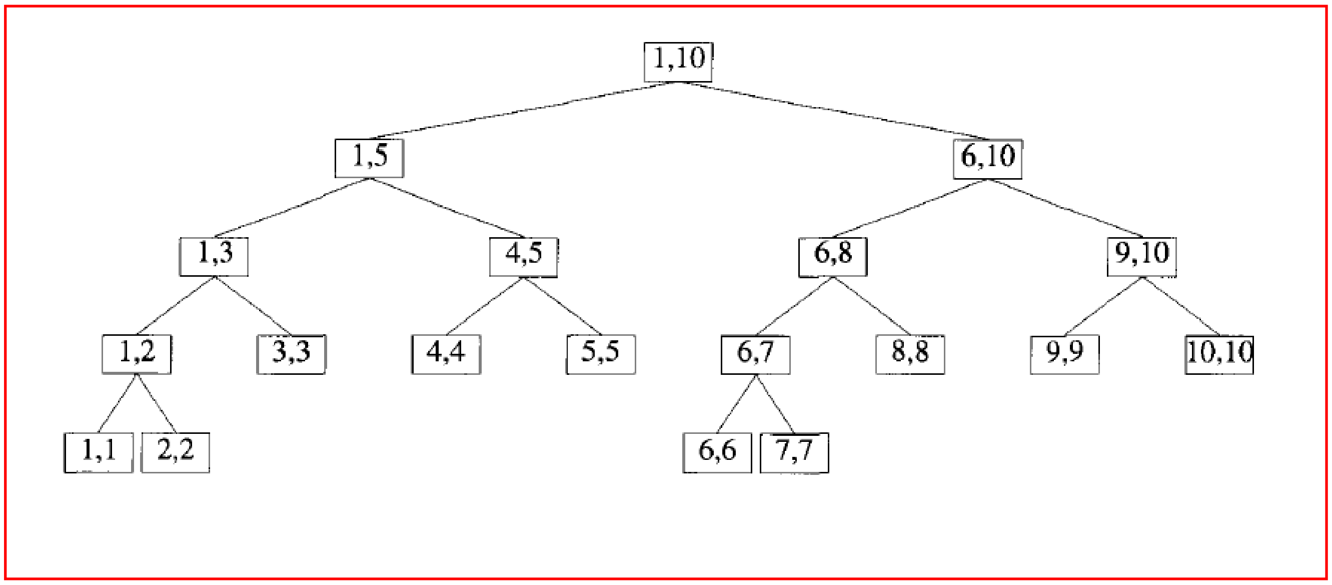
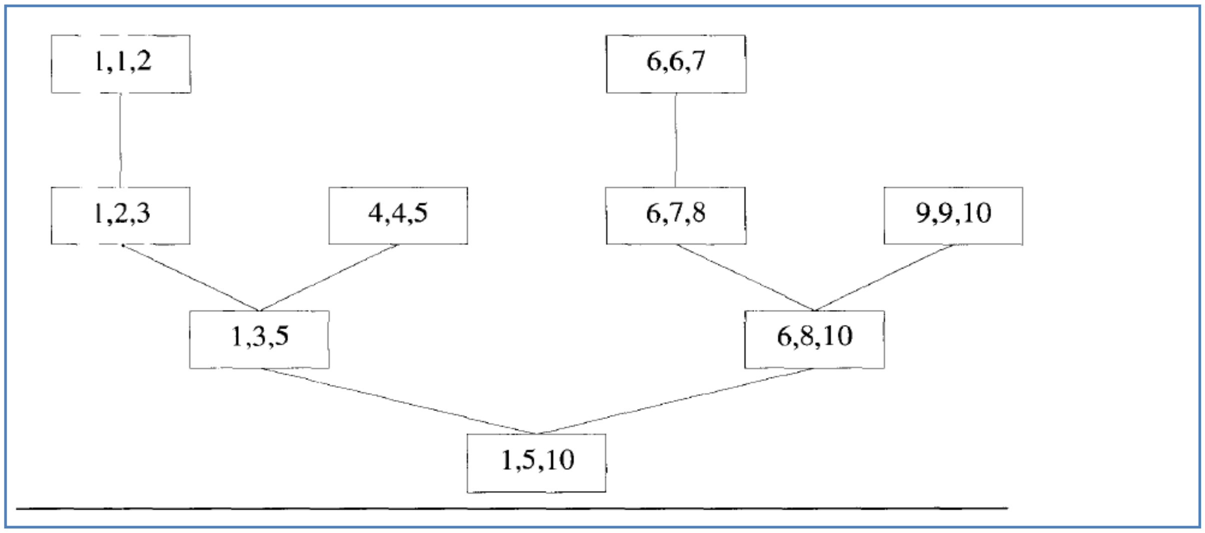


Figure: Tree of calls of Merge

The below Figure is a tree representing the calls to procedure Merge by Merge Sort.

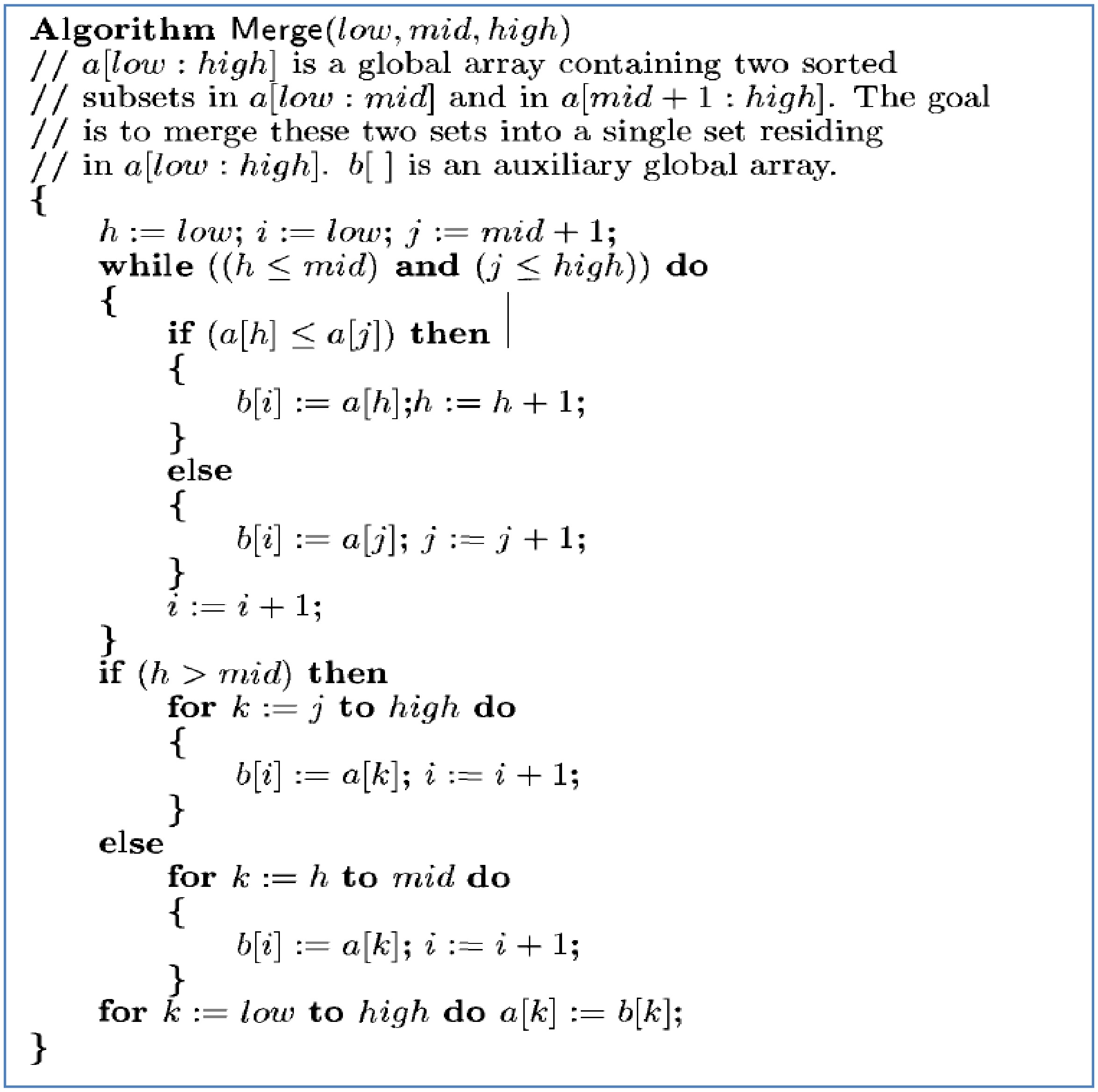
For example, the node containing1,2, and 3 represents the merging of a[l:2] with a[3].

- The below figure shows the divide-and-conquer nature of Merge sort, there remain several inefficiencies that can and should be eliminated.
- We present these refinements in an attempt to produce a version of Merge sort that is good enough to execute. Despite these improvements the algorithm's complexity remains **O(nlogn).**



Merge Sort Algorithm - using recursion

```
Algorithm MergeSort($low, high$)
// $a[low : high]$ is a global array to be sorted.
// Small($P$) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if ($low < high$) then  // If there are more than one element
    {
        // Divide $P$ into subproblems.
            // Find where to split the set.
                $mid := \lfloor (low + high)/2 \rfloor$;
        // Solve the subproblems.
            MergeSort($low, mid$);
            MergeSort($mid + 1, high$);
        // Combine the solutions.
            Merge($low, mid, high$);
    }
}
```



```
Algorithm Merge($low, mid, high$)
// $a[low : high]$ is a global array containing two sorted
// subsets in $a[low : mid]$ and in $a[mid + 1 : high]$. The goal
// is to merge these two sets into a single set residing
// in $a[low : high]$. $b[\,]$ is an auxiliary global array.
{
    $h := low$; $i := low$; $j := mid + 1$;
    while (($h \le mid$) and ($j \le high$)) do
    {
        if ($a[h] \le a[j]$) then
        {
            $b[i] := a[h]$;$h := h + 1$;
        }
        else
        {
            $b[i] := a[j]$; $j := j + 1$;
        }
        $i := i + 1$;
    }
    if ($h > mid$) then
        for $k := j$ to $high$ do
        {
            $b[i] := a[k]$; $i := i + 1$;
        }
    else
        for $k := h$ to $mid$ do
        {
            $b[i] := a[k]$; $i := i + 1$;
        }
    for $k := low$ to $high$ do $a[k] := b[k]$;
}
```

Algorithm- Merging two sorted sub arrays using auxiliary storage

Time Complexity:

- Since **Merge Sort** always divides the array into two halves and then merges them, the time complexity for the best, worst, and average cases is identical, i.e., O(nlogn).
- If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

Where:

- $T(n)$ is the time complexity for an input of size n.
- $2T\left(\frac{n}{2}\right)$ is the time to solve two subproblems of size $n/2$ (this occurs because we divide the problem into two halves).
- cn is the time taken to combine the two subproblems (in **Merge Sort**, this is the time taken for the merging step, which is linear in n).

The goal is to solve this recurrence using **successive substitution** to determine the time complexity of the algorithm.

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \end{aligned}$$

- After k substitutions (where $n = 2^k$), we get:

$$T(n) = 2^k T(1) + k \cdot cn$$

Since $T(1) = O(1)$ (i.e., the base case takes constant time), we can simplify:

$$T(n) = O(1) + k \cdot cn$$

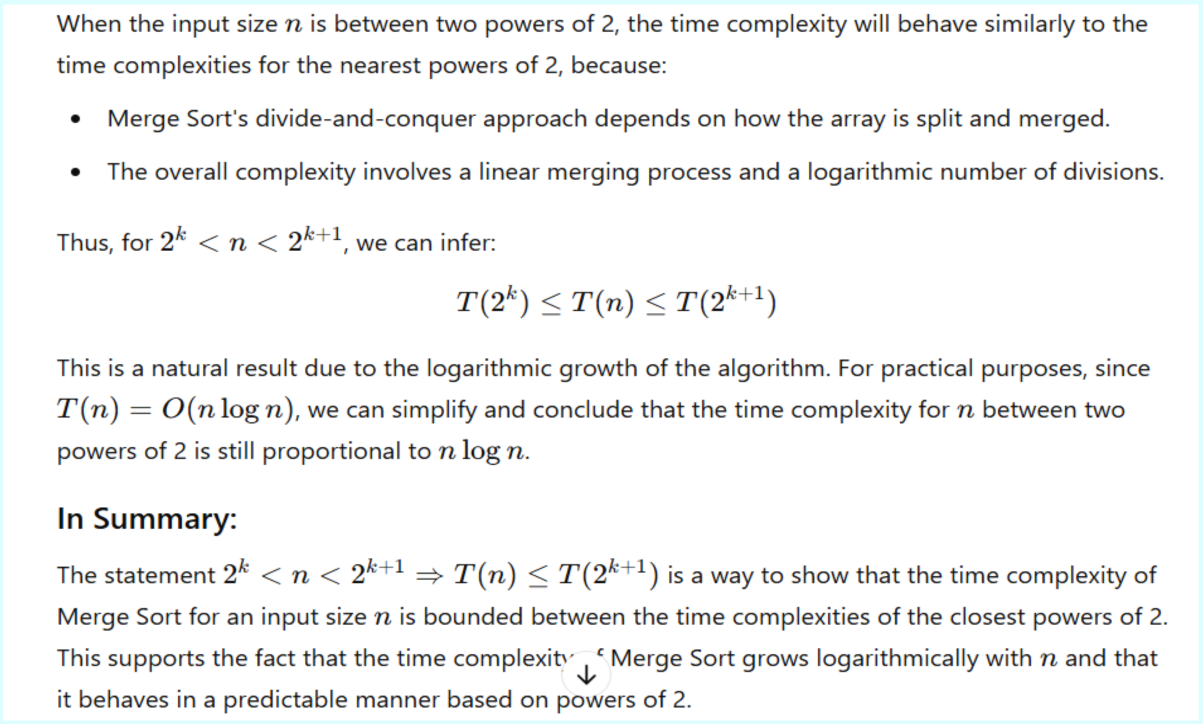
Recall that $k = \log_2 n$ (because $n = 2^k$), so the equation becomes:

$$T(n) = O(n \log n)$$

Thus, the time complexity of the algorithm is $O(n \log n)$.

T(n) = O (n log n)

NOTE:


When the input size n is between two powers of 2, the time complexity will behave similarly to the time complexities for the nearest powers of 2, because:

- Merge Sort's divide-and-conquer approach depends on how the array is split and merged.
- The overall complexity involves a linear merging process and a logarithmic number of divisions.

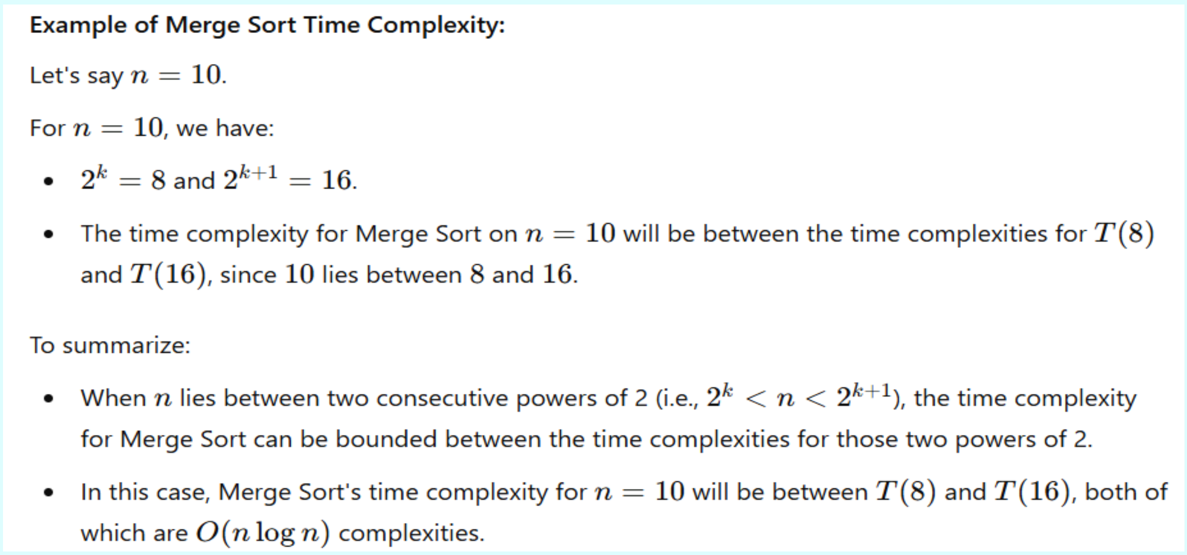
Thus, for $2^k < n < 2^{k+1}$, we can infer:

$$T(2^k) \leq T(n) \leq T(2^{k+1})$$

This is a natural result due to the logarithmic growth of the algorithm. For practical purposes, since $T(n) = O(n \log n)$, we can simplify and conclude that the time complexity for n between two powers of 2 is still proportional to $n \log n$.

In Summary:

The statement $2^k < n < 2^{k+1} \Rightarrow T(n) \leq T(2^{k+1})$ is a way to show that the time complexity of Merge Sort for an input size n is bounded between the time complexities of the closest powers of 2. This supports the fact that the time complexity of Merge Sort grows logarithmically with n and that it behaves in a predictable manner based on powers of 2.


Example of Merge Sort Time Complexity:

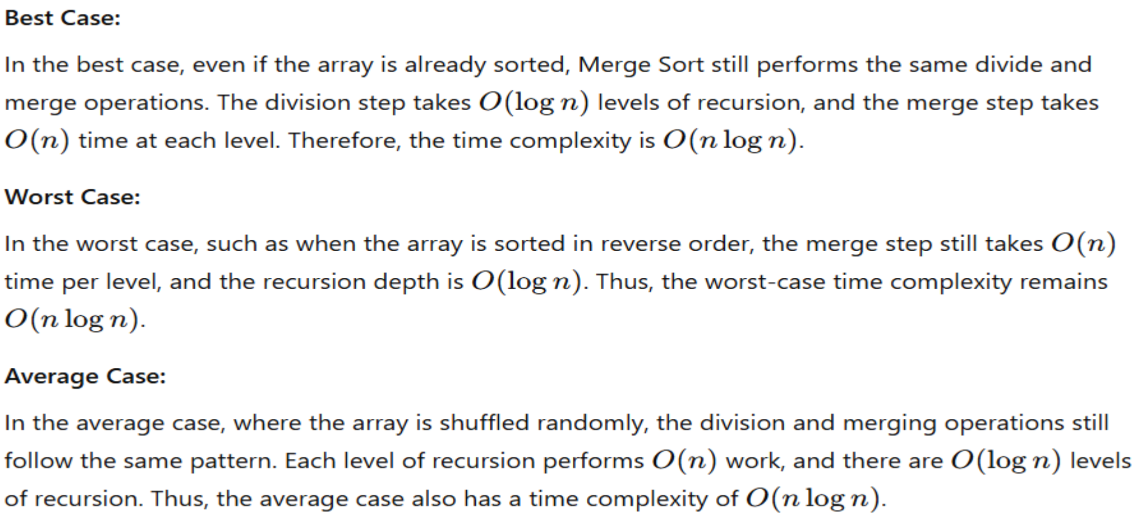
Let's say $n = 10$.

For $n = 10$, we have:

- $2^k = 8$ and $2^{k+1} = 16$.
- The time complexity for Merge Sort on $n = 10$ will be between the time complexities for $T(8)$ and $T(16)$, since 10 lies between 8 and 16.

To summarize:

- When n lies between two consecutive powers of 2 (i.e., $2^k < n < 2^{k+1}$), the time complexity for Merge Sort can be bounded between the time complexities for those two powers of 2.
- In this case, Merge Sort's time complexity for $n = 10$ will be between $T(8)$ and $T(16)$, both of which are $O(n \log n)$ complexities.


Best Case:

In the best case, even if the array is already sorted, Merge Sort still performs the same divide and merge operations. The division step takes $O(\log n)$ levels of recursion, and the merge step takes $O(n)$ time at each level. Therefore, the time complexity is $O(n \log n)$.

Worst Case:

In the worst case, such as when the array is sorted in reverse order, the merge step still takes $O(n)$ time per level, and the recursion depth is $O(\log n)$. Thus, the worst-case time complexity remains $O(n \log n)$.

Average Case:

In the average case, where the array is shuffled randomly, the division and merging operations still follow the same pattern. Each level of recursion performs $O(n)$ work, and there are $O(\log n)$ levels of recursion. Thus, the average case also has a time complexity of $O(n \log n)$.

Quick sort:

- In quick sort, the division into two sub arrays is made so that the sorted sub arrays do not need to be merged later. This is accomplished by rearranging the elements in a[1:n] such that a[i] < a[j] for all **i** between**1** and **m** and all **j** between **m + 1** and **n** for some m, 1< m < n.
- Thus, the elements in **a[1:m] and a[m+1:n]** can be independently sorted. **No merge is needed**.
- The rearrangement of the elements is accomplished by picking Some element of a[ ], say **t = a[s],**and then reordering the other elements So that all elements appearing **before t** in a[1:n] are less than or equal to **t** and all elements appearing **after t** are greater than or equal to t.
- This rearranging is referred to as partitioning.

Example: As an example of how Partition works, consider the following array of nine elements. The function is initially invoked as Partition(a,1,10).

- ❖ The element **a[1]= 65 is the partitioning element** and it is eventually (in the sixth row) determined to be the fifth smallest element of the set. Notice that the remaining elements are unsorted but partitioned about a[5] = 65.
- ❖ Function Partition of **Algorithm-1** accomplishes an **in-place partitioning** of the elements of a[m:p-1].
- ❖ It is assumed that a[p] >=a[m] and that **a[m] is the partitioning element.**
- ❖ If m = 1and n=p-1, then **a[n + 1] must be defined and must be greater than or equal to all elements in a[1:n].**

m,i								j	
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
65	70	75	80	85	60	55	50	45	+∞

Until a[i]>a[m] do i++ then stop
Unitil a[j]<a[m] do j-- then stop

m,	i							j	
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
65	70	75	80	85	60	55	50	45	+∞

m,	i							j	
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
65	45	75	80	85	60	55	50	70	+∞

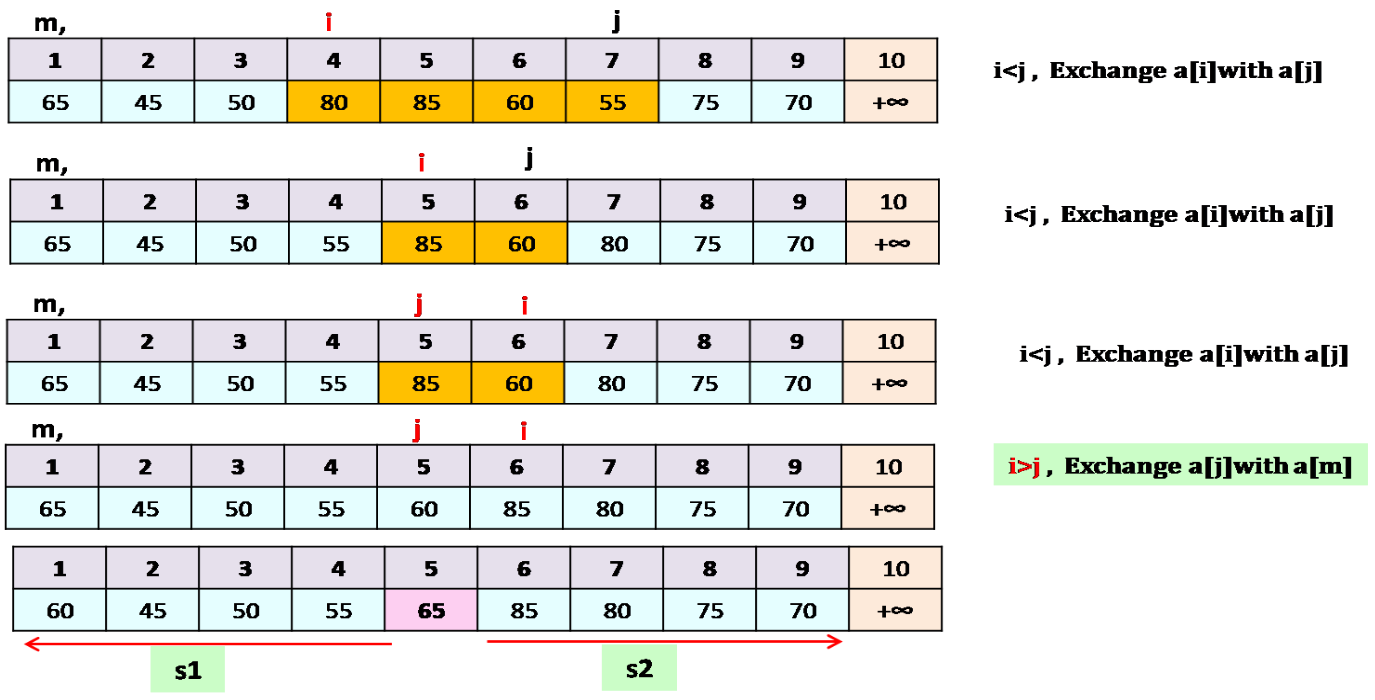
i<j , Exchange a[i]with a[j]

m,		i					j		
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
65	45	75	80	85	60	55	50	70	+∞

i<j , Exchange a[i]with a[j]

m,		i					j		
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
65	45	50	80	85	60	55	75	70	+∞

i<j , Exchange a[i]with a[j]


m,
i
j
i<j , Exchange a[i]with a[j]
m,
i
j
i<j , Exchange a[i]with a[j]
m,
j
i
i<j , Exchange a[i]with a[j]
m,
j
i
i>j , Exchange a[j]with a[m]
s1
s2

1	2	3	4	5	6	7	8	9	10
65	45	50	80	85	60	55	75	70	+∞

1	2	3	4	5	6	7	8	9	10
65	45	50	55	85	60	80	75	70	+∞

1	2	3	4	5	6	7	8	9	10
65	45	50	55	85	60	80	75	70	+∞

1	2	3	4	5	6	7	8	9	10
65	45	50	55	60	85	80	75	70	+∞

1	2	3	4	5	6	7	8	9	10
60	45	50	55	65	85	80	75	70	+∞

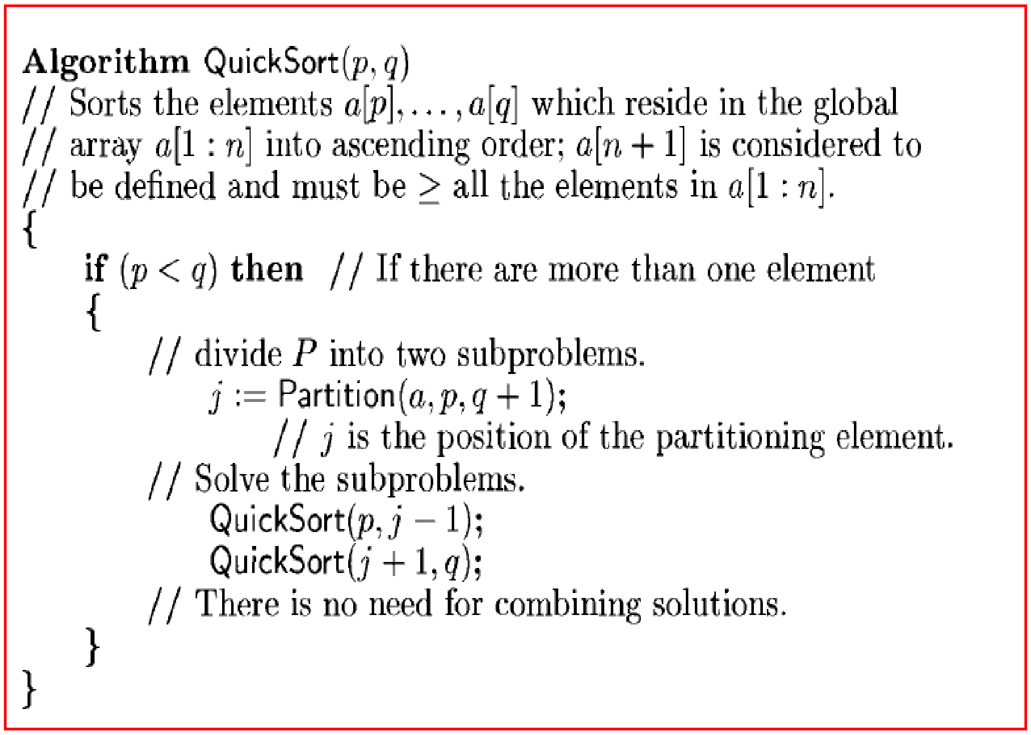
- The assumption that **a[m]** is the partition element is merely for convenience; other choices for the partitioning element than the first item in the set are better in practice.
- The function Interchange(a,i,j) exchanges a[i]with a[j].

```
Algorithm Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \geq v$);

        repeat
            $j := j - 1$;
        until ($a[j] \leq v$);

        if ($i < j$) then Interchange($a, i, j$);

    } until ($i \geq j$);

    $a[m] := a[j]$; $a[j] := v$; return $j$;
}

Algorithm Interchange($a, i, j$)
// Exchange $a[i]$ with $a[j]$.
{
    $p := a[i]$;
    $a[i] := a[j]$; $a[j] := p$;
}
```

Algorithm-1

❖ Following a call to the **function Partition**, two sets **S1** and **S2** are produced. All elements in **S1** are less than or equal to the element **S2** .

❖ Hence S1 and S2 can be sorted independently. Each set is sorted by reusing the function Partition. **Algorithm -2** describes the complete process.



```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j − 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

Algorithm -2

Time Complexity:

Average case:

To understand the average case, we need to consider how QuickSort works and how the choice of pivot affects the time complexity. QuickSort works by repeatedly selecting a pivot element and partitioning the array into two subarrays:

- One subarray with elements smaller than the pivot.
- One subarray with elements larger than the pivot.

After the partitioning, the algorithm recursively sorts the two subarrays.

The **average case** occurs when the pivot divides the array in such a way that, on average, each subarray has approximately half the elements of the original array. This balanced partitioning ensures that the recursion tree grows logarithmically.

Breakdown of the Average Case:

1. **Pivot Selection**:
   - In the average case, we assume that the pivot is chosen randomly or in a manner that it does not always pick the smallest or largest element (as it might in the worst case).
   - This random selection leads to an expected balanced partition, meaning the pivot typically divides the array into two subarrays of roughly equal size.

2. **Recursion Tree**:
   - Suppose we have n elements in the array.
   - In the first partitioning step, the pivot divides the array into two subarrays. On average, the pivot will divide the array into two subarrays of sizes about $n/2$ each.
   - At the next level of recursion, the two subarrays of size $n/2$ will each be partitioned into two subarrays of size $n/4$, and so on.
   - This process continues, leading to a recursion tree with a depth of approximately $\log n$.
3. **Work per Level**:
   - At each level of recursion, we need to perform a **partitioning step**. In the partition step, every element is compared to the pivot, so the number of comparisons at each level is proportional to the number of elements in that level.
   - At the first level, we compare all n elements to the pivot. At the second level, we compare approximately $n/2$ elements in total (as the array has been divided into two subarrays). At the third level, we compare $n/4$ elements, and so on.
   - The total number of comparisons at each level is proportional to n.
4. **Total Work Across Levels**:
   - The total work across all levels of recursion can be calculated by summing the work at each level:

$$\text{Total work} = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots$$

This is a geometric series with the sum:

$$\text{Total work} = O(n)$$

The number of comparisons at each level adds up to $O(n)$, and since the recursion tree has a depth of $\log n$, the total work across all levels is $O(n \log n)$.

The key idea here is that Quick Sort is efficient in the average case because the array is split into roughly equal-sized sub arrays at each step. This leads to a logarithmic recursion depth and linear work at each level. The partitioning step takes O(n) work, and there are O(logn) levels, leading to an overall time complexity of:

T(n)=O(nlogn)

Best case:

The best case arises when the pivot always divides the array into two **equal-sized sub arrays**. This balanced partitioning minimizes the number of recursive calls and ensures that the total work grows logarithmically in the number of levels, each level performing O(n) comparisons.

- **Balanced Partitioning**: The key to the best-case scenario is that the array is split in half at each level, ensuring that each recursive call works with progressively smaller arrays of roughly equal size. This leads to a recursion depth of **logn.**
- **Total Work**: The work at each level is O(n) (since each partitioning step compares every element), and there are O(logn) levels in the recursion tree. Hence, the total time complexity in the best case is O(nlogn).

T(n)=O(nlogn)

Worst-Case

1. **Pivot Selection in the Worst Case**:

- Suppose that the pivot selected is always the **smallest** or **largest** element of the subarray being processed.
- After partitioning, the pivot will be in its final position, but the **left subarray** (with elements smaller than the pivot) will contain no elements, and the **right subarray** (with elements larger than the pivot) will contain almost all the elements.
- Essentially, every partition step only reduces the problem by one element (the pivot), and the recursion depth becomes **linear** in the size of the array.

2. **Example of Worst Case:**

- Let's consider an array of n elements: $[1, 2, 3, 4, 5, 6, 7, 8, 9]$.
- If the pivot is always chosen as the **smallest element**, say 1, the partitioning will look like this:
  - First partition: 1 is placed at the first position, and the array becomes $[1]$ and $[2, 3, 4, 5, 6, 7, 8, 9]$. Only the right subarray remains.
  - Second partition: 2 is placed at the second position, and the array becomes $[2]$ and $[3, 4, 5, 6, 7, 8, 9]$.
  - This process continues until we are left with a single element in the left subarray and all the rest in the right subarray.
- Notice that at each level of recursion, we only reduce the problem by one element. Therefore, the recursion depth becomes n, and the total work done at each level adds up to $O(n^2)$.

3. **Recursion Tree in the Worst Case:**

- In the worst-case scenario, the recursion tree is **highly unbalanced**. It essentially forms a **linear chain** instead of a balanced tree.
- Each level of recursion only processes one fewer element than the previous level, leading to a depth of n (where each level processes just one element).
- At the first level, you perform $O(n)$ comparisons to partition the array. At the second level, you perform $O(n-1)$ comparisons, at the third level, $O(n-2)$ comparisons, and so on.

So, the total work across all levels is:

$$O(n) + O(n-1) + O(n-2) + \cdots + O(1) = O(n^2)$$

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

Key Difference between Best-Case and Average-Case:

- **Best case**: The pivot divides the array perfectly into two equal parts at each level.
- **Average case**: The pivot, chosen randomly, will divide the array into two roughly equal parts on average, but there might be some imbalance in certain cases. However, the imbalance does not grow large enough to lead to quadratic time complexity, and on average, the recursion tree will still be balanced enough to yield **O(nlogn).**

Space Complexity

1. Best and Average Case:

- In the best and average cases, where the pivot splits the array into two roughly equal subarrays, the recursion depth is **logarithmic**.
  - The maximum depth of recursion is $O(\log n)$, where n is the number of elements in the array.
  - At each level of recursion, the algorithm works on a subarray of size n, but the actual space used for recursion is proportional to the number of recursive calls, not the size of the array.
  - Therefore, in the **best-case** and **average-case**, the space complexity is dominated by the depth of the recursion tree, which is $O(\log n)$.

Space Complexity in Best/Average Case: $O(\log n)$

2. Worst Case:

- In the worst case, where the pivot consistently results in highly unbalanced partitions (e.g., the pivot is always the smallest or largest element), the recursion depth becomes **linear**. This occurs when the algorithm effectively sorts one element at a time, leading to a recursion tree of height n.
  - In this case, the space complexity will be proportional to the recursion depth, which is $O(n)$.

Space Complexity in Worst Case: $O(n)$

Applications of Quick Sort Algorithm

1. **Numerical analysis:**For accuracy in calculations most of the efficiently developed algorithm uses a **priority queue** and quick sort is used for sorting.
2. **Call Optimization:** It is tail-recursive and hence all the call optimization can be done.
3. **Database management:** Quicksort is the fastest algorithm so it is widely used as a better way of searching.

Advantages of Quick Sort Algorithm

1. **Efficiency:** It is efficient on large data sets.
2. **In-place sorting:** Quick sort performs **in-place sorting**, meaning it doesn't require additional memory to store temporary data or intermediate results.
3. **Simple implementation:** Quick sort's algorithmic logic is relatively straightforward, making it easier to understand and implement compared to some other complex sorting algorithms

Disadvantages of Quick Sort Algorithm

1. **Unstable Sorting:** Quick sort is an **unstable sorting algorithm**, meaning that it does not guarantee the relative order of equal elements after the sorting process.
2. **Worst-case Performance:** Its worst-case **time complexity** is O(n^2). It occurs when the pivot chosen is the smallest or largest element in the array, causing unbalanced partitions.
3. **Dependency on Pivot Selection:** The choice of pivot significantly affects the performance of quick sort. This issue is particularly prominent when dealing with already sorted or nearly sorted arrays.
4. **Recursive Overhead:** **Recursive function** calls consume additional memory and can lead to **stack overflow** errors when dealing with large data sets.
5. **Not Adaptive:** Quick sort does not take advantage of any pre-existing order or partially sorted input. This means that even if a portion of the array is already sorted, the quick sort will still perform partitioning operations on the entire array.
6. **Inefficient for Small Data Sets:** Quick sort has additional overhead in comparison to some simpler sorting algorithms, especially when dealing with small data sets. The recursive nature of quick sort and the constant splitting of the array can be inefficient for sorting small arrays.