UNIT-2

ALGORITHM

An algorithm is a set of instructions designed to perform a specific task. It takes a set of input(s) and produces the desired output. When designing algorithms, it is important to consider not only the correctness of the algorithm, but also its efficiency.

Algorithm Analysis is the process of determining the efficiency and correctness of an algorithm. It is a fundamental part of computer science, helping to evaluate the **performance** of an algorithm in terms of time and space. The main goal of analyzing an algorithm is to ensure that it can handle large inputs efficiently and scale well. Two key measures of efficiency are **time complexity** and **space complexity**.

TIME AND SPACE COMPLEXITY

Space Complexity

Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle. Space needed by an algorithm is equal to the sum of the following two components

A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.

A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

Space complexity S(p) of any algorithm p is S(p) = c + Sp(I) Where c is treated as the fixed part and S(I) is treated as the variable part of the algorithm which depends on instance characteristic I.

Example:-

1. Algorithm

```
abc(x,y,z)
{
return x*y*z+(x-y)
}
```

Here we have three variables x, y and z. x,y and z each variable requires one unit of memory.
S(p) = c + sp
S(p) = 3 + 0
S(p)=3

2. Algoritham sum(x,n)
{
Total:=0
For i←1 to n do
Total:=total + x[i]
}
S(p)=c+sp
Here c = x,n,total (each one requires one unit of memory)=3
Sp=x[i] and its depends on n
S(p)=3+n

Time complexity

The time complexity of an algorithm is defined as the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

1. 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:-

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

Now we can calculate execution statement count. Initially count = 0. Then

```
Algorithm sum(a,n)
{
S: = 0.0
Count : = count+1;
For i: = 1 to n do
{
Count : = count+1;
S: = S + a[i];
Count : = count+1;

}
Count : = count+1;  // for last exexution
return S;
Count : = count+1;
}
```

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.

2. 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**

Common Time Complexities (from most efficient to least efficient):

- **O(1)**: Constant time – the algorithm takes the same amount of time regardless of input size.
- **O(log n)**: Logarithmic time – time grows logarithmically with input size (e.g., binary search).
- **O(n)**: Linear time – time grows linearly with input size.
- **O(n log n)**: Quasilinear time – typical for efficient sorting algorithms like merge sort.
- **O(n^2)**: Quadratic time – often seen in algorithms with nested loops, such as bubble sort.
- **O(2^n)**: Exponential time – very slow, used in algorithms that solve combinatorial problems.
- **O(n!)**: Factorial time – extremely slow, often associated with brute force algorithms.

Worst-case, Average-case, Best-case Time Complexity

Worst-case running time

This denotes the behaviour of an algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

Average-case running time

The average-case running time of an algorithm is an estimate of the running time for an 'average' input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

Best-case running time

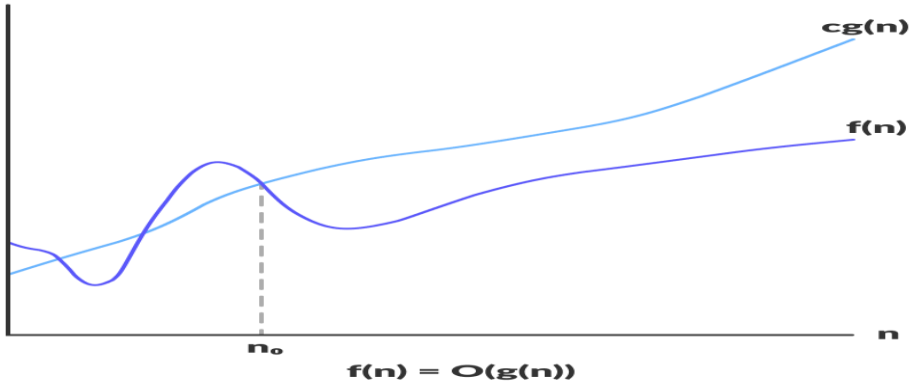
The term 'best-case performance' is used to analyze an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list. However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance. It is always recommended to improve the average performance and the worst-case performance of an algorithm.

ASYMPTOTIC NOTATIONS

Asymptotic notations are used to represent the complexities of algorithms for asymptotic analysis. These notations are mathematical tools to represent the complexities. There are three notations that are commonly used.

Big O Notation (O): It represents the **upper bound** of the runtime of an algorithm. Big O Notation's role is to calculate the longest time an algorithm can take for its execution, i.e., it is used for calculating the **worst-case time** complexity of an algorithm.

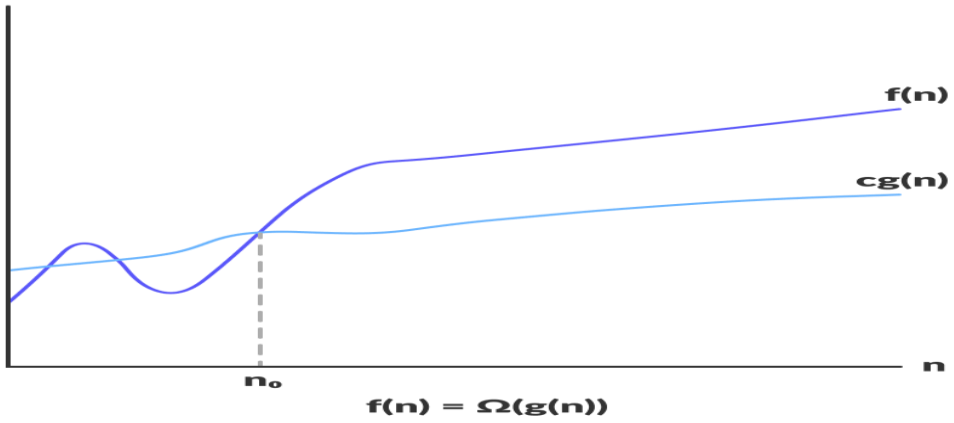
Let g and f be functions from the set of natural numbers to itself. The function f is said to be O(g) (read big-oh of g), if there is a constant **c > 0** and a natural number $\mathbf{n_0}$ such that f(n) ≤ cg(n) for all $n \geq n_0$.



Omega Notation (Ω(n)): It represents the **lower bound** of the runtime of an algorithm. It is used for calculating the best time an algorithm can take to complete its execution, i.e., it is used for measuring the **best case time** complexity of an algorithm.

We write f(n) = Ω(g(n)), If there are positive constants n0 and c such that, to the right of n_0 the f(n) always lies on or above c*g(n).

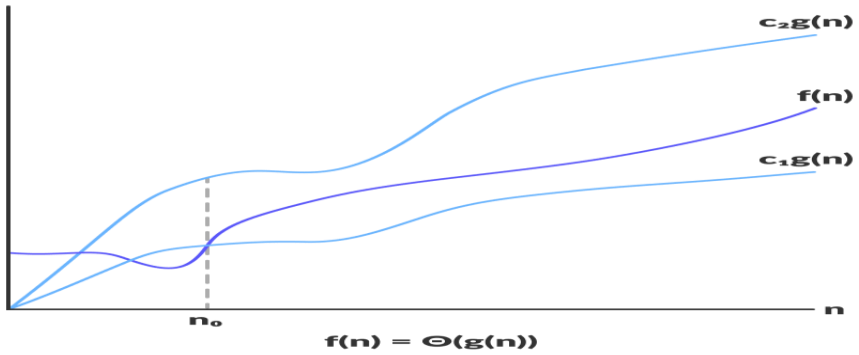
Ω(g(n)) = { f(n) : There exist positive constant c and n0 such that $0 \leq c\ g(n) \leq f(n)$, for all $n \leq n_0$}



Theta Notation (Θ(n)): It carries the middle characteristics of both Big O and Omega notations as it represents the **lower and upper bound** of an algorithm.

We write f(n) = Θ(g(n)), If there are positive constants n0 and c_1 and c_2 such that, to the right of n_0 the f(n) always lies between c_1*g(n) and c_2*g(n) inclusive.

Θ(g(n)) = {f(n) : There exist positive constant c_1, c_2 and n_0 such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$, for all $n \geq n_0$}

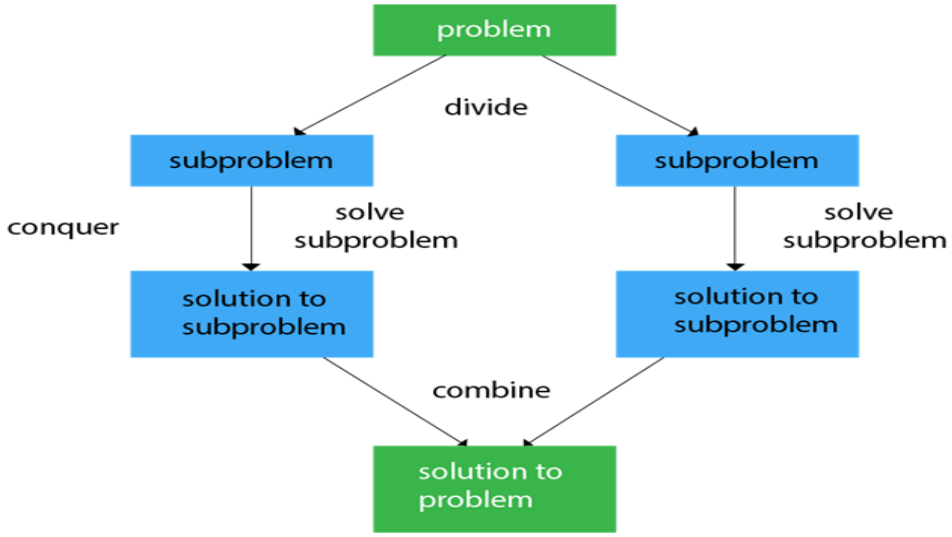


DIVIDE AND CONQUER

Divide and Conquer is a fundamental algorithm design paradigm used to solve problems by breaking them down into smaller subproblems, solving the subproblems independently, and then combining their solutions to solve the original problem. It is widely applied in various algorithms, particularly those that deal with recursion.

Key Concepts in Divide and Conquer:

1. **Divide**: The problem is divided into smaller sub-problems of the same type.
2. **Conquer**: Solve the sub-problems recursively. If the sub-problem size is small enough, solve it directly (base case).
3. **Combine**: Merge the solutions of the sub-problems to produce the solution to the original problem.




```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P1, P2, ..., Pk, k ≥ 1;
7           Apply DAndC to each of these subproblems;
8           return Combine(DAndC(P1),DAndC(P2),...,DAndC(Pk));
9       }
10  }
```

Time Complexity

The complexity of the divide and conquer algorithm is calculated using the master theorem which is as follow.

$T(n) = aT(n/b) + f(n)$,
where,
n = size of input
a = number of sub-problems in the recursion
n/b = size of each sub-problem. All sub-problems are assumed to have the same size.
f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Now let's take an example by finding the time complexity of a recursive problem
$T(n) = aT(n/b) + f(n)$
$= 2T(n/2) + O(n)$
Where,
a = 2 (each time, a problem is divided into 2 sub-problems)
n/b = n/2 (size of each sub problem is half of the input)
f(n) = time taken to divide the problem and merging the sub-problems
$T(n/2) = O(n \log n)$ (To understand this, please refer to the master theorem.)

Now, **$T(n) = 2T(n \log n) + O(n) \approx O(n \log n)$**

QUICK SORT

Quick sort is a sorting algorithm that uses the divide and conquers strategy. In this method division is dynamically carried out. The three steps of quick sort are as follows:
Divide: split the array into two sub arrays that each elements in the left sub array is less than or equal the middle element and each element in the right is greater than the middle element. The splitting of the array into two sub arrays is based on pivot element. All the elements that are less that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right sub array.
Conquer: Recursively sort the two sub array.
Combine: Combine all the sorted elements in a group to from a list of sorted elements.

The quick sort algorithm works as follows:
1. Select an element pivot from the array elements.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called **the *partition*** operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.) After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array. Thus, the main task is to find the pivot element, which will

partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

Technique

Quick sort works as follows:

1. Set the index of the first element in the array to loc and left variables. Also, set the index of the last element of the array to the right variable. That is, loc = 0, left = 0, and right = n–1 (where n in the number of elements in the array)

2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc. That is, a[loc] should be less than a[right].

(a) If that is the case, then simply continue comparing until right becomes equal to loc. Once right = loc, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to Step 3.

(c) Set loc = right

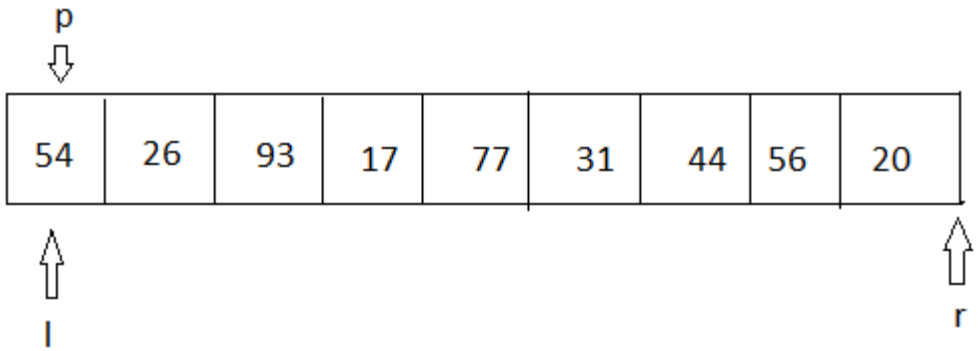
3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc. That is, a[loc] should be greater than a[left].

(a) If that is the case, then simply continue comparing until left becomes equal to loc. Once left = loc, it means the pivot has been placed in its correct position.

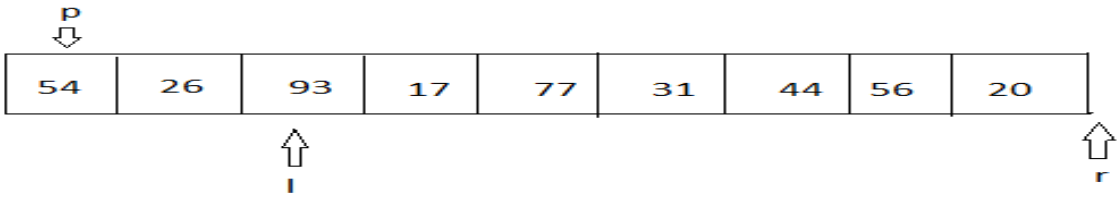
(b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to Step 2.

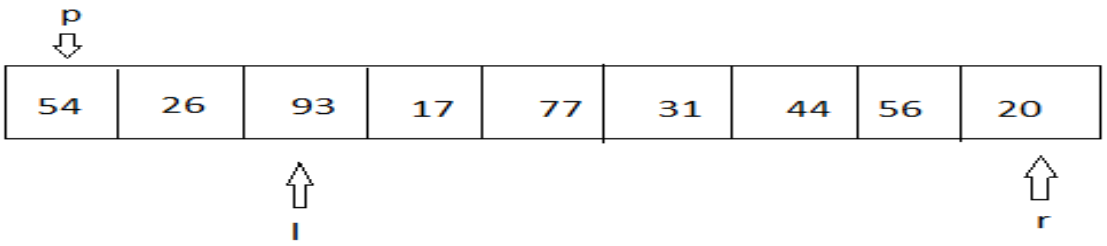
(c) Set loc = left.

Example Sort the elements given in the following array using quick sort algorithm Let us consider an array A. 54, 26, 93, 17, 77, 31, 44, 56, 20

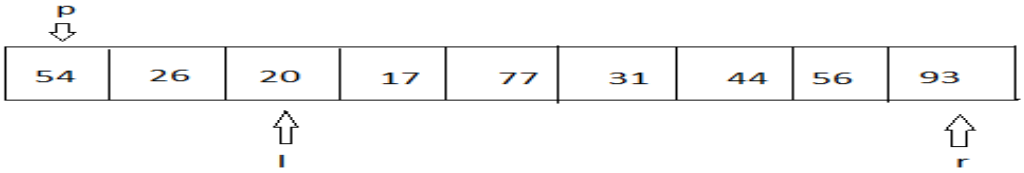


First element is considered as pivot element. l = A [0] and r = A [n]. l = l+1 until l > p and r = r - 1 until r < p, this is done until l > r

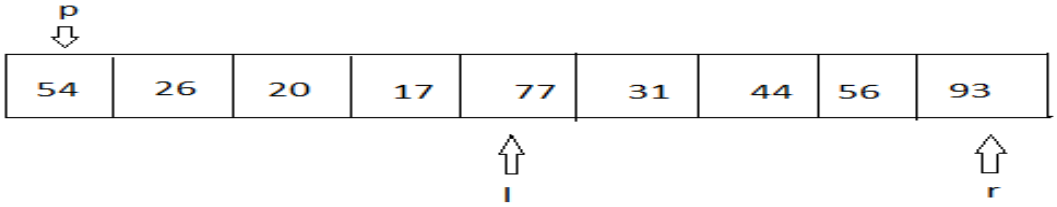




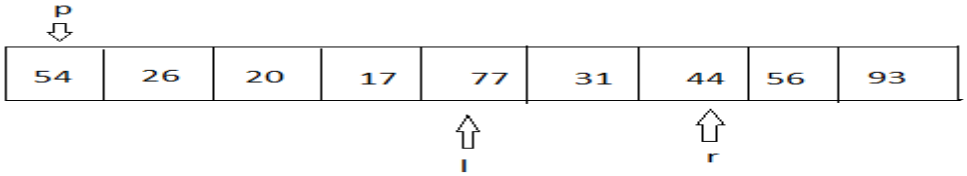
Now l = 93 and r = 20 then swap (l, r) i.e. swap (93, 20) then



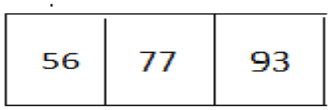
l = l+1 until l > p



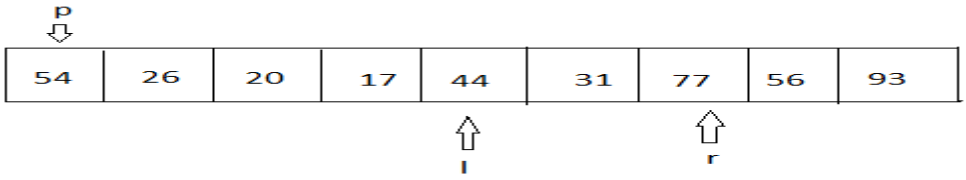
r = r-1 until r < p



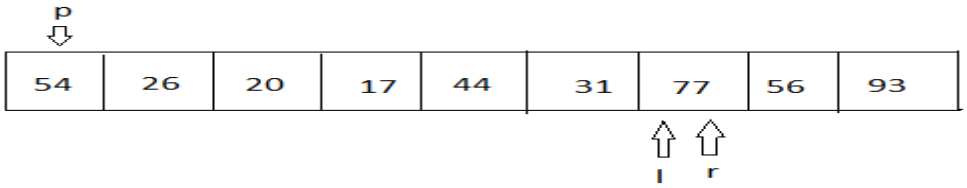
Now l = 77 and r = 44, then swap the positions of l and r i.e. swap (77, 44) then the array is



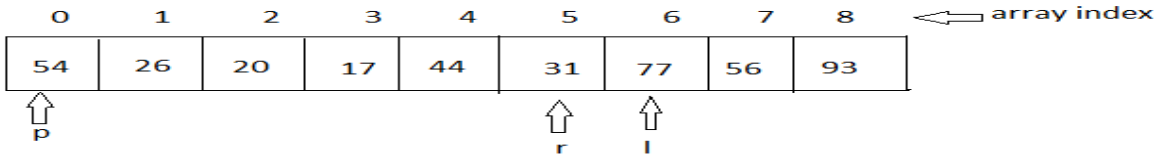
Then combine these two sorted arrays we get the sorted array as



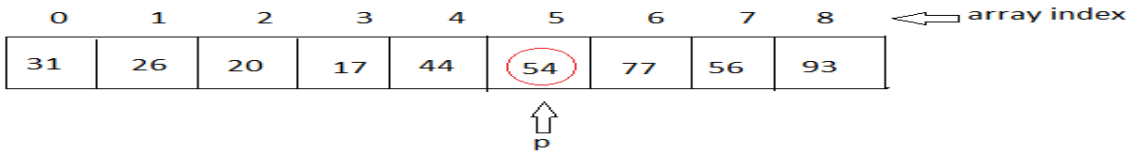
l = l+1 until l > p



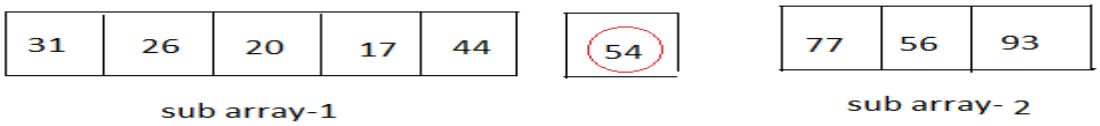
r = r-1 until r < p



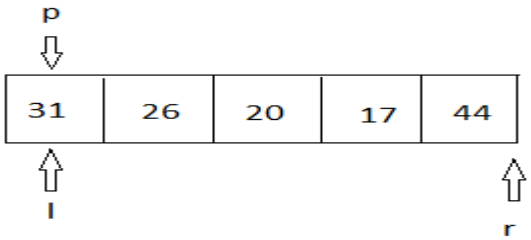
Here l = 77, r = 31. But in this position l > r then swap (p, r) i.e. swap (54, 33)



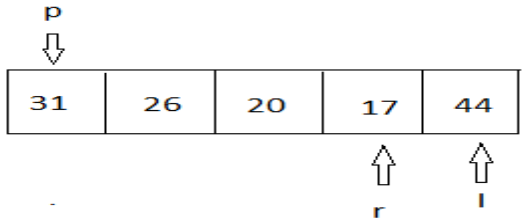
Here 54 are a partition element and fix the 54 position. In this position the left side elements of 54 are arranged less than 54 and the right side elements of 54 are arranged greater than 54. Now we can divide the array into two sub arrays.



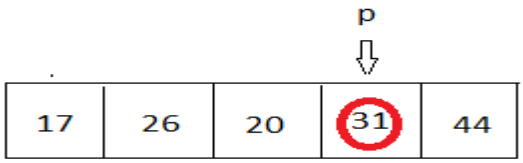
Partition – 1(sub array -1)



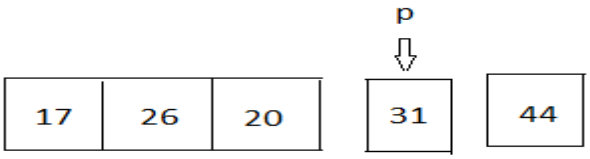
l = l+1 until l > p and r = r-1 until r < p then



Here l > r then swap (p, r)



Here p is a partition element



Continue the same process finally we can get the sorted sub array-1 as

Algorithm for quick sort (A [l….r])
If l < r there is some element
S← partition [l….r]
Quick sort (A [l…S-1])
Quick sort (A [S+1…r])

Algorithm partition (A [l….r])
// AIM: - partition a sub array by using its first element as a pivot
// Input: - A sub array A [l….r] of A [0…n-1] defined by left and right indices l and r (l < r)
i← l
j← r+1
repeat
repeat i = i + 1 until A [i] >= p
repeat j = j - 1 until A [i] <= p
swap (A[i], A[j])
until i > j
swap (A[l], A[j])
return j

Complexity of Quick Sort

In the average case, the running time of quick sort can be given as O(n log n). The partitioning of the array which simply loops over the elements of the array once uses O(n) time. In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only log n nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is O(log n). And because at each level, there can only be O(n), the resultant time is given as O(n log n) time.

Practically, the efficiency of quick sort depends on the element which is chosen as the pivot. Its worst-case efficiency is given as O(n2). The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot. However, many implementations randomly choose the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of O(n log n).

MERGE SORT

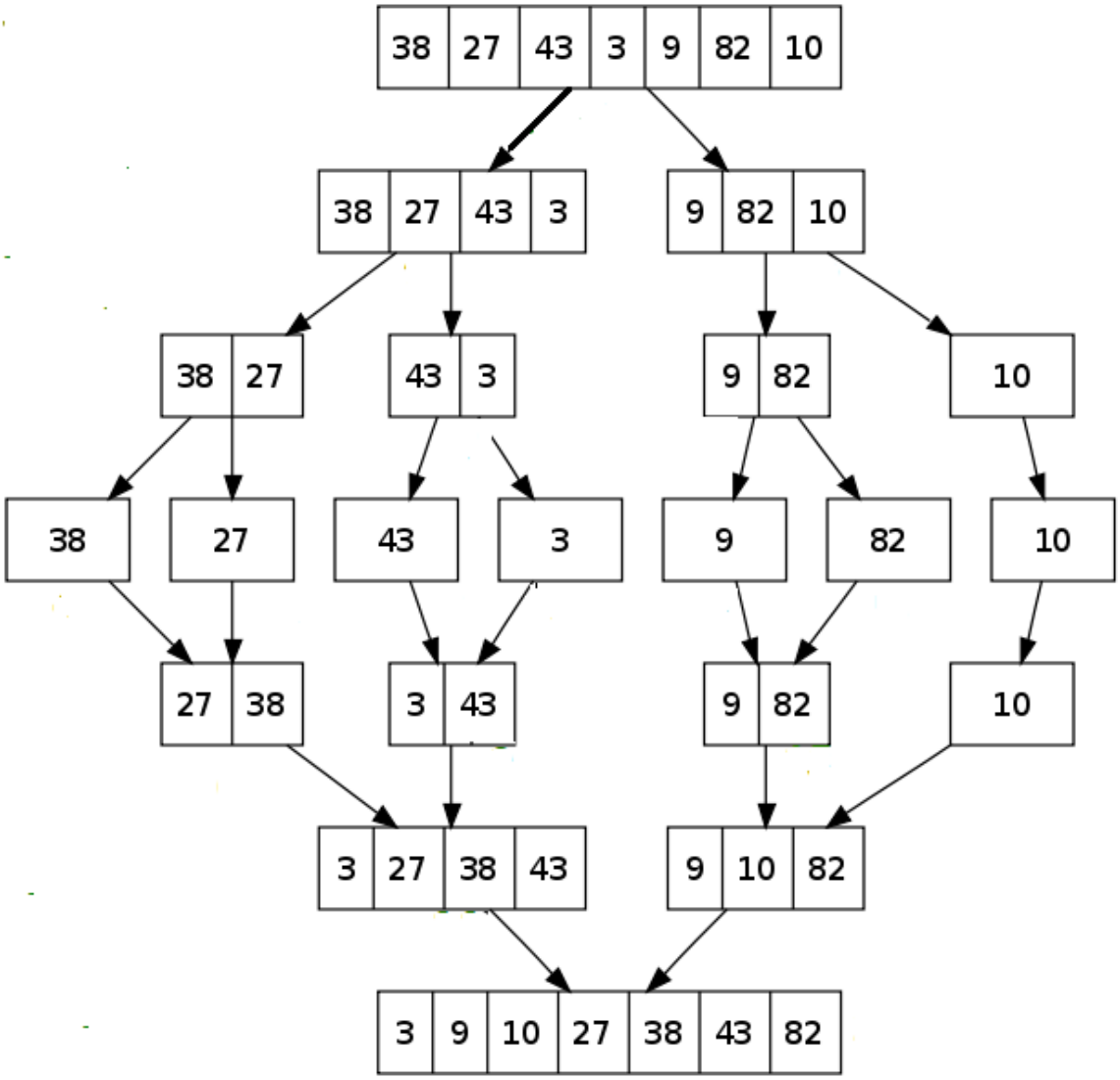
Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm. ***Divide*** means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.
Conquer means sorting the two sub-arrays recursively using merge sort.
Combine means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.
Merge sort algorithm focuses on two main concepts to improve its performance (running time):

- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

Example:-
Sort the array given below using merge sort. 38, 27, 43, 3, 9, 82, 10



Algorithm for merge sort

MERGE_SORT (ARR, BEG, END)

Step 1: IF BEG < END

SET MID = (BEG + END)/2

CALL MERGE_SORT (ARR, BEG, MID)

CALL MERGE_SORT (ARR, MID + 1, END)

MERGE (ARR, BEG, MID, END) [END OF IF]

Step 2: END

MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
IF ARR [I] < ARR [J]
SET TEMP [INDEX] = ARR [I]
SET I = I + 1
ELSE
SET TEMP [INDEX] = ARR [J]
SET J = J + 1
[END OF IF]
SET INDEX = INDEX + 1
[END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
IF I > MID
Repeat while J <= END
SET TEMP [INDEX] = ARR [J]
SET INDEX = INDEX + 1,
SET J = J + 1
[END OF LOOP]
[Copy the remaining elements of left sub-array, if any]
ELSE
Repeat while I <= MID
SET TEMP [INDEX] = ARR [I]
SET INDEX = INDEX + 1,
SET I = I + 1
[END OF LOOP]
[END OF IF]
Step 4: [Copy the contents of TEMP back to ARR]
SET K= 0
Step 5: Repeat while K < INDEX
SET ARR [K] = TEMP [K]
SET K = K + 1
[END OF LOOP]
Step 6: END

STRASSEN'S MATRIX MULTIPLICATION

Consider two matrices A and B each of the size N*N. We want to calculate the resultant matrix C which is formed by multiplying the given two matrices i.e, A and B. Matrix A has a size N*M and matrix B has a size A*B. Given two matrices are multiplied only when the number of columns in the first matrix is equal to the number of rows in the second matrix. Therefore, we can say that matrix multiplication is possible only when M = = A.

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Let us take the example the multiplication of two matrix of size 3 by 3 is

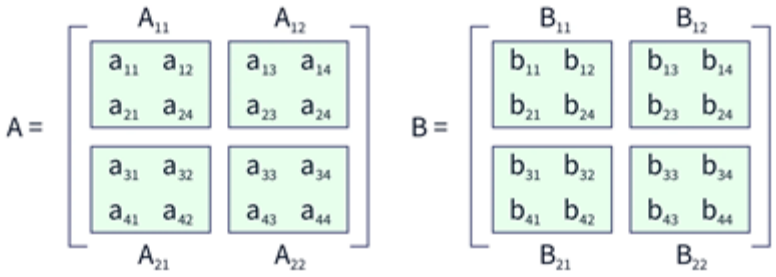
$$A \times B = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix}$$

$$A \times B = \begin{bmatrix} (aj + bm + cp) & (ak + bn + cq) & (al + bo + cr) \\ (dj + em + fp) & (dk + en + fq) & (dl + eo + fr) \\ (gj + hm + ip) & (gk + hn + iq) & (gl + ho + ir) \end{bmatrix}$$

Where the given matrices are square matrices of size is N*N each. For multiplying every column with every element in the row from the given matrices uses two loops, and adding up the values takes another loop. Therefore the overall time complexity turns out to be O(N^3).

To optimize the matrix multiplication Strassen's matrix multiplication algorithm can be used. It reduces the time complexity for matrix multiplication. Strassen's Matrix multiplication can be performed only on square matrices where N is a power of 2. And also the order of both of the matrices is N × N.

The main idea is to use the divide and conquer technique in this algorithm. We need to divide matrix A and matrix B into 8 sub matrices and then recursively compute the submatrices of the result.



For each multiplication of size N2×N2$N2 \times N2$, follow the below recursive function as

```
MatMul ( A, B, n ) {
    it ( n ≤ 2 ) return
C11= a11+ b11+ a12+ b21
C12= a11+ b12+ a12+ b22
C21= a21+ b11+ a22+ b21
C22= a21+ b21+ a22+ b22
}

MatMul ( A11 ,B11 , n/2 ) + MatMul ( A12 ,B21 , n/2 )
MatMul ( A11 ,B12 , n/2 ) + MatMul ( A12 ,B22 , n/2 )
MatMul ( A21 ,B11 , n/2 ) + MatMul ( A22 ,B21 , n/2 )
MatMul ( A21 ,B12 , n/2 ) + MatMul ( A22 ,B22 , n/2 )
```

Where the resultant matrix is C and can be obtained in the following way. Now we need to store the result as follows:

C11=A11∗B11+A12∗B21$C11 = A11 * B11 + A12 * B21$
C12=A11∗B12+A12∗B22$C12 = A11 * B12 + A12 * B22$
C21=A21∗B11+A22∗B21$C21 = A21 * B11 + A22 * B21$
C22=A21∗B12+A22∗B22$C22 = A21 * B12 + A22 * B22$

The recurrence relation obtained is: T(N)=8T(N2)+4O(N2)$T(N) = 8T(N_2) + 4O(N2)$

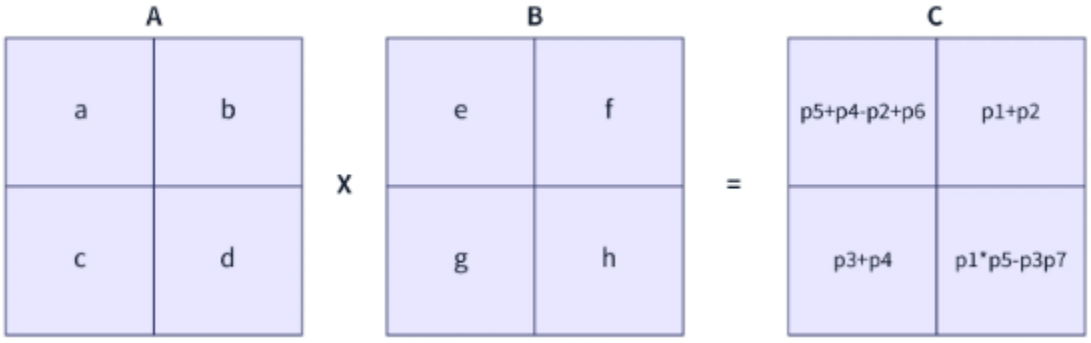
To optimize it further, we use Strassen's Matrix Multiplication where we don't need 8 recursive calls, as we can solve them using 7 recursive calls and this requires some manipulation which is achieved using addition and subtraction.

Strassen's 7 calls are as follows:

- $p_1 = (A_{11}) * (B_{11} - B_{22})$
- $p_2 = (A_{11} + A_{12}) * (B_{22})$
- $p_3 = (A_{21} + A_{22}) * (B_{11})$
- $p_4 = (A_{22}) * (B_{21} + B_{22})$
- $p_5 = (A_{11} + A_{22}) * (B_{11} + B_{22})$
- $p_6 = (A_{12} - A_{21}) * (B_{21} + B_{22})$
- $p_7 = (A_{11} - A_{21}) * (B_{11} + B_{12})$

Now the resultant matrix can be obtained in the following way:

$$Mat_A \times Mat_B = \begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & (p_1 * p_5) - (p_3 * p_7) \end{bmatrix}$$



Complexity Analysis

Time Complexity: O(N^log(7)), where N * N is the order of square matrices given.

We just need 7 recurrences Strassen's Matrix Multiplication algorithm and some addition subtraction has to be performed to manipulate the answer. The recurrence relation obtained is: $T(N)=7T(N_2)+O(N^2)$

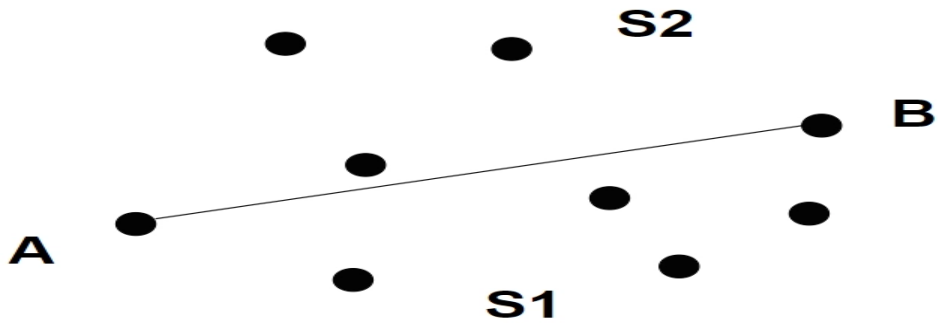
By solving the above relation, we get the overall time complexity of Strassen's Matrix Multiplication is $O(N^{\log(7)})$. By simplifying that we get the overall time complexity of approximately is $O(N^{2.8074})$ which is better than $O(N^3)$.

CONVEX HULL

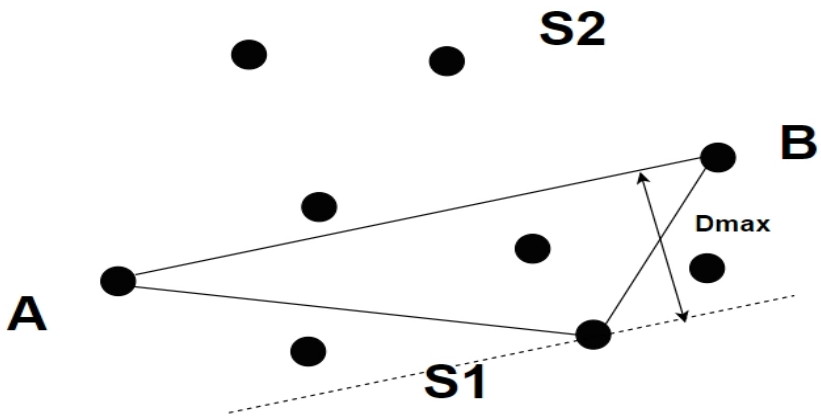
A convex hull is the smallest convex polygon that completely encloses a set of points in a two-dimensional or three-dimensional space. It can be thought of as the "envelope" or "wrapper" of the points. We use the divide and conquer approach to solve this by recursively calling the function for smaller parameters.

Here is an illustration of our approach:

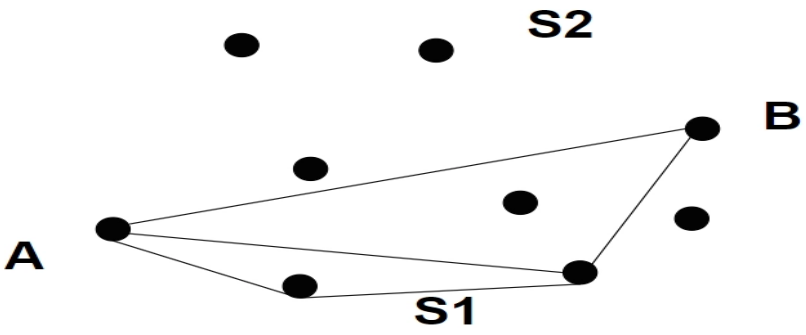
The first step is to find out the farthest two points in the plane:



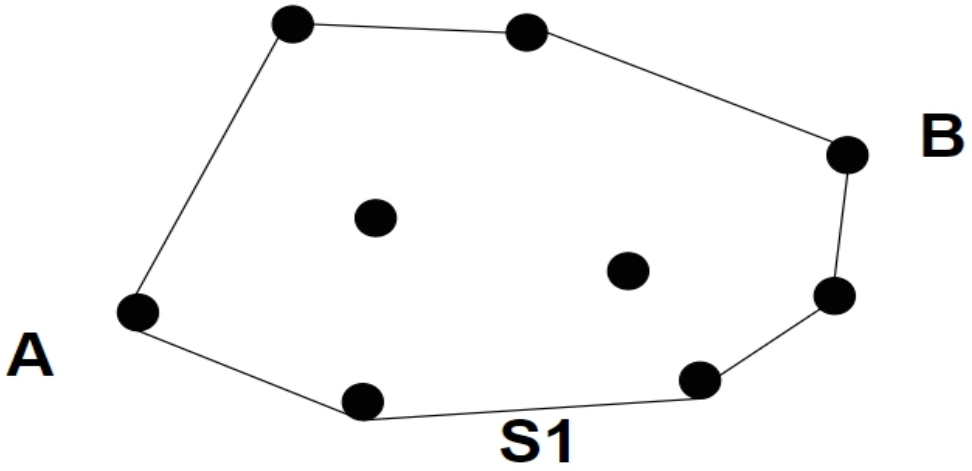
Then, in the two spaces S1 and S2, we will find out the farthest point:



We will continue to do operations



Finally, Our resultant polygon would look something like this:



Algorithm

The steps that we'll follow to solve the problem are:

1. First, we'll sort the vector containing points in ascending order (according to their x-coordinates).
2. Next, we'll divide the points into two halves S1 and S2. The set of points S1 contains the points to the left of the median, whereas the set S2 contains all the points that are right to the median.
3. We'll find the convex hulls for the set S1 and S2 individually. Assuming the convex hull for S1 is C1, and for S2, it is C2.
4. Now, we'll merge C1 and C2 such that we get the overall convex hull C.