BYJU'S EXAM PREP

# Vision 2023

## A Course for GATE & PSUs

## Computer Science Engineering

## Algorithm

**CHAPTER 3**

## Divide and Conquer

# CHAPTER 3 ALGORITHM

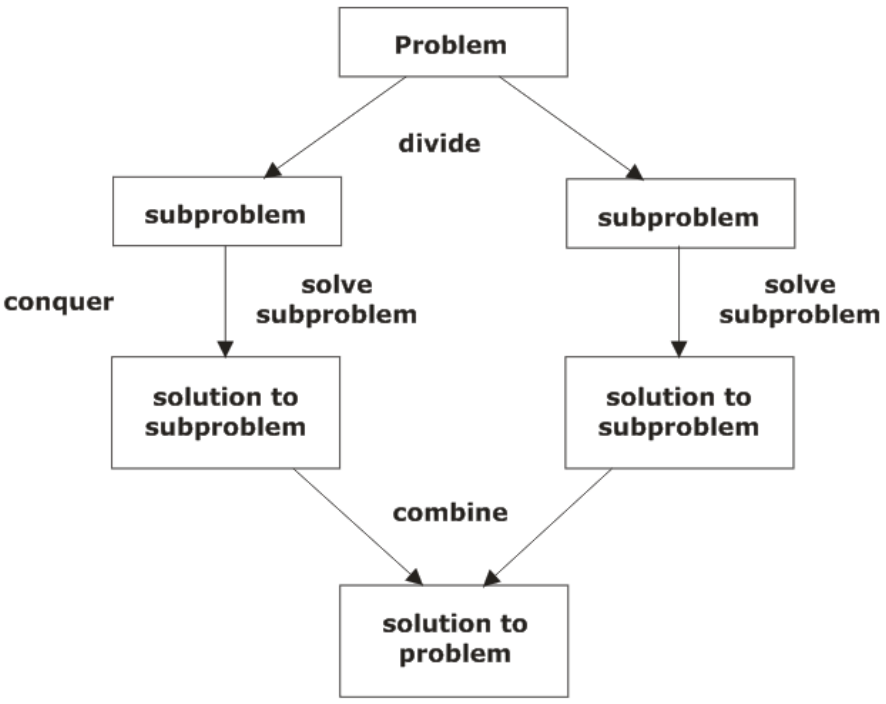
# DIVIDE AND CONQUER

**Divide and Conquer**

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

The divide and conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems. Divide steps define that break the problem into smaller sub-problems. Subproblems represent a subpart of the actual problem. This step generally takes recursively to divide the problem till no sub-problem occurs. Sub-problems become atomic in nature and represent a sub-part of the original problem.
2. **Conquer:** Solve every subproblem individually, recursively. Conquer includes many smaller subproblems for solving. The smaller problems are already solved on their own.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem. After solving the sub problems, sub problems are recursively combined all until they solve the original problem.



**Fundamental of Divide & Conquer Strategy:**

There are two fundamentals of Divide & Conquer Strategy:

- Relational Formula
- Stopping Condition

1. **Relational Formula**: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.
2. **Stopping Condition**: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

**Divide And Conquer algorithm :**

```
DAC(a, i, j)
{
if(small(a, i, j))                    // stopping Condition
return(solution(a, i, j))
   else
     m = divide(a, i, j)             // f1(n)
     b = DAC(a, i, mid)               // T(n/2)
     c = DAC(a, mid+1, j)            // T(n/2)
     d = combine(b, c)                // f2(n)
  return(d)
}
```

**Recurrence Relation for DAC algorithm :**

This is the recurrence relation for the above program.

$T(n) = O(1)$ if n is small

$T(n) = f1(n) + 2T(n/2) + f2(n)$

**Applications Of Divide and Conquer-**

- Max Min problem
- Power of an element
- Binary Search
- Merge Sort
- Quick Sort
- Selection Procedure
- Strassen' s matrix multiplication
- Finding Inversion

1. **Max - Min Problem :**

Analyze the algorithm to find the maximum and minimum element from an array.

**Maximum Minimum**

Find Maximum and minimum element of an array using minimum number of comparisons.

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is y−x+1y−x+1, where **y** is greater than or equal to **x**.

Max−Min(x,y)Max−Min(x,y) will return the maximum and minimum values of an array numbers[x...y]numbers[x...y].

**Algorithm: Max - Min(x, y)**

if y – x ≤ 1 then

return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))

else

(max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)

(max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)

return (max(max1, max2), min(min1, min2))

Analysis

Let ***T(n)*** be the number of comparisons made by Max−Min(x,y)Max−Min(x,y), where the number of elements n=y−x+1n=y−x+1.

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

T(n)=⎧⎩⎨⎪⎪T(⌊n2⌋)+T(⌈n2⌉)+210forn>2forn=2forn=1T(n)={T(⌊n2⌋)+T(⌈n2⌉)+2forn>21forn=20forn=1

Let us assume that ***n*** is in the form of power of **2**. Hence, $\mathbf{n = 2^k}$ where **k** is height of the recursion tree.

So,

T(n)=2.T(n2)+2=2.(2.T(n4)+2)+2.....=3n2−2

• Recurrence Relation: $T(n) = \begin{cases} 0; & \text{if } n = 1 \\ 1; & \text{if } n = 2 \\ T(n/2) + T(n/2) + 1; & \text{if } n > 2 \end{cases}$

Time complexity with DAC = O(n)

(i) Using linear search = O(n)

(ii) Using Tournament method: Time complexity = O(n)

Number of Comparison: When n is power of 2 then

T(n) = 2 T(n/2) + 2 = (3n/2) -2 else more than (3n/2) – 2.

(iii) Compare in Pairs: Time complexity = O(n)

Number of comparison

(a) When n is even = (3n/2) – 2,

(b) Else n is odd = (3n – 1)/2.

**Example**

Consider the problem of computing min-max in an unsorted array where min and max are minimum and maximum elements of the array. Algorithm A1 can compute min-max in a1 comparisons without divide and conquer. Algorithm A2 can compute min-max in a2 comparisons by scanning the array linearly. What could be the relation between a1 and a2 considering the worst case scenarios?

(A) a1 < a2

(B) a1 > a2

(C) a1 = a2

(D) Depends on the input

Answer: (B)

Explanation: When Divide and Conquer is used to find the minimum-maximum element in an array, Recurrence relation for the number of comparisons is

T(n) = 2T(n/2) + 2 where 2 is for comparing the minimums as well the maximums of the left and right subarrays

On solving, T(n) = 1.5n – 2.

While doing linear scan, it would take 2*(n-1) comparisons in the worst case to find both minimum as well maximum in one pass.

**2. <u>Power of an element-</u>**

- $T(n) = \begin{cases} 1 & , \quad \text{if } n = 0 \\ a & , \quad \text{if } n = 1 \\ T(n/2) + c & , \quad \text{otherwise} \end{cases}$

Time complexity = $O(\log_2 n)$

C Program for calculating Power of an Element

```
/* Extended version of power function that can work
for float x and negative y*/
#include<stdio.h>
float power(float x, int y)
{
        float temp;
        if( y == 0)

return 1;
```

```
        temp = power(x, y/2);
        if (y%2 == 0)
                return temp*temp;
        else
        {
                if(y > 0)
                        return x*temp*temp;
                else
                        return (temp*temp)/x;
        }
}

/* Program to test function power */
int main()
{
        float x = 2;
        int y = -3;
        printf("%f", power(x, y));
        return 0;
}
```

**3. Binary Search –**

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.

Binary Search Algorithm can be applied only on Sorted arrays.

So, the elements must be arranged in-

- Either ascending order if the elements are numbers.
- Or dictionary order if the elements are strings.

To apply binary search on an unsorted array,

- First, sort the array using some sorting technique.
- Then, use binary search algorithm.

**Binary Search Algorithm-**

Consider-

- There is a linear array 'a' of size 'n'.
- Binary search algorithm is being used to search an element 'item' in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.

- Variables beg and end keeps track of the index of the first and last element of the array or sub array in which the element is being searched at that instant.
- Variable mid keeps track of the index of the middle element of that array or sub array in which the element is being searched at that instant.

Then, Binary Search Algorithm is as follows-

```
Begin
Set beg = 0
Set end = n-1
Set mid = (beg + end) / 2
while ( (beg <= end) and (a[mid] ≠ item) ) do
if (item < a[mid]) then
Set end = mid - 1
else
Set beg = mid + 1
endif
Set mid = (beg + end) / 2
endwhile
if (beg > end) then
Set loc = -1
else
Set loc = mid
endif
End
```

**Explanation**

Binary Search Algorithm searches an element by comparing it with the middle most element of the array.

Then, following three cases are possible-

Case-01

If the element being searched is found to be the middle most element, its index is returned.

Case-02

If the element being searched is found to be greater than the middle most element, then its search is further continued in the right sub array of the middle most element.

Case-03

If the element being searched is found to be smaller than the middle most element, then its search is further continued in the left sub array of the middle most element.

This iteration keeps on repeating on the sub arrays until the desired element is found or size of the sub array reduces to zero.

**Time Complexity Analysis-**

Binary Search time complexity analysis is done below-

- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

Thus, we have-

> The Time Complexity of a Binary Search Algorithm is $O(\log_2 n)$.
> Here, n is the number of elements in the sorted linear array.

This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

**Binary Search Example-**

Consider-

- We are given the following sorted linear array.
- Element 15 has to be searched in it using Binary Search Algorithm.

| 3 | 10 | 15 | 20 | 35 | 40 | 60 |
|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] |

**Binary Search Example**

Binary Search Algorithm works in the following steps-

**Step-01:**

- To begin with, we take beg=0 and end=6.
- We compute location of the middle element as-

mid
= (beg + end) / 2
= (0 + 6) / 2
= 3

- Here, a[mid] = a[3] = 20 ≠ 15 and beg < end.
- So, we start the next iteration.

**Step-02:**

- Since a[mid] = 20 > 15, so we take end = mid – 1 = 3 – 1 = 2 whereas beg remains unchanged.
- We compute location of the middle element as-

mid
= (beg + end) / 2

= (0 + 2) / 2

= 1

- Here, a[mid] = a[1] = 10 ≠ 15 and beg < end.
- So, we start next iteration.

**Step-03:**

- Since a[mid] = 10 < 15, so we take beg = mid + 1 = 1 + 1 = 2 whereas end remains unchanged.
- We compute location of the middle element as-

mid

= (beg + end) / 2

= (2 + 2) / 2

= 2

- Here, a[mid] = a[2] = 15 which matches to the element being searched.
- So, our search terminates in success and index 2 is returned.

**Binary Search Algorithm Advantages-**

The advantages of binary search algorithm are-

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

**Binary Search Algorithm Disadvantages-**

The disadvantages of binary search algorithm are-

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.

(because of its random access nature)

**Important Note-**

For in-memory searching, if the interval to be searched is small,

- Linear search may exhibit better performance than binary search.
- This is because it exhibits better locality of reference.

**4. Merge Sort-**

- Merge sort is a famous sorting algorithm.
- It uses a divide and conquer paradigm for sorting.

- It divides the problem into sub problems and solves them individually.
- It then combines the results of sub problems to get the solution of the original problem.
- Comparison based sorting.
- Stable sorting algorithm but outplace.
- Recurrence Relation: $T(n) = \begin{cases} c \text{ if } n = 1 \\ T(n/2) + T(n/2) + cn \text{ if } n > 1 \end{cases}$

Time complexity: (i) Worst case = O(nlogn) ,
(ii) Best case = O(nlogn),
((i)) Average case = O(nlogn).
Space complexity: S(n) = 2n + logn + c; Worst case = O(n)

Merge Sort Algorithm works in the following steps-

- It divides the given unsorted array into two halves- left and right sub arrays.
- The sub arrays are divided recursively.
- This division continues until the size of each sub array becomes 1.
- After each sub array contains only a single element, each sub array is sorted trivially.
- Then, the above discussed merge procedure is called.
- The merge procedure combines these trivially sorted arrays to produce a final sorted array.

The division procedure of merge sort algorithm which uses recursion is given below-

```
// A : Array that needs to be sorted
MergeSort(A)
{
n = length(A)
if n<2 return
mid = n/2
left = new_array_of_size(mid) // Creating temporary array for left
right = new_array_of_size(n-mid) // and right sub arrays
for(int i=0 ; i<=mid-1 ; ++i)
{
left[i] = A[i] // Copying elements from A to left
}
for(int i=mid ; i<=n-1 ; ++i)
{
right[i-mid] = A[i] // Copying elements from A to right
}
MergeSort(left) // Recursively solving for left sub array
MergeSort(right) // Recursively solving for right sub array
merge(left, right, A) // Merging two sorted left/right sub array to final array
}
```

**Properties-**

Some of the important properties of merge sort algorithm are-

- Merge sort uses a divide and conquer paradigm for sorting.
- Merge sort is a recursive sorting algorithm.
- Merge sort is a stable sorting algorithm.
- Merge sort is not an in-place sorting algorithm.
- The time complexity of merge sort algorithm is Θ(nlogn).
- The space complexity of merge sort algorithm is Θ(n).

**NOTE**

Merge sort is the best sorting algorithm in terms of time complexity Θ(nlogn) if we are not concerned with auxiliary space used.

## PRACTICE PROBLEMS BASED ON MERGE SORT ALGORITHM-

### Problem 1-

Assume that a merge sort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes?

1. 256
2. 512
3. 1024
4. 2048

### Solution-

We know, time complexity of merge sort algorithm is Θ(nlogn).

**Step-01:**

It is given that a merge sort algorithm in the worst case takes 30 seconds for an input of size 64.

So, we have-

k x nlogn = 30 (for n = 64)

k x 64 log64 = 30

k x 64 x 6 = 30

From here, k = 5 / 64.

**Step-02:**

Let n be the maximum input size of a problem that can be solved in 6 minutes (or 360 seconds).

Then, we have-

k x nlogn = 360

(5/64) x nlogn = 360 { Using Result of Step-01 }

nlogn = 72 x 64

nlogn = 4608

On solving this equation, we get n = 512.

Option B is correct.

**Problem 2-**

T(n)=2T(n/2)+n

=2[2T(n/4)+n/2]+n

=2[2[2T(n/8)+n/4]+n/2]+n

=2^3T(n/8)+2n+n

=2^k T(n/2^k)+kn

so, n/2^k=1

k=logn

nk=n*logn

T(n)=O(n*logn)

No of levels = k = logn

and at each level the time complexity = n

time complexity= n*logn

5. **Quick Sort-**

- Comparison based sorting.
- 2 to 3 times faster than merge and heap sort.
- Not a stable sorting algorithm but inplace.

Choosing pivot is the most important factor.

As Start element T(n) = $O(n^2)$, As End element T(n) = $O(n^2)$

As Middle element T(n) = $O(n^2)$, As Median element T(n) = $O(n^2)$

As Median of Median T(n) = O(n logn)

**Time complexity:**

(i) Best case: each partition splits array into two halves then

T(n) = T(n/2) + T(n/2) + n = O(nlogn)

(ii) Worst case: each partition gives unbalanced splits we get

T(n) T(n – k) + T(k – 1) + cn = $O(n^2)$

(iii) Average case: In the average case, we donot know where the split happens for this reason we take all the possible value of split locations.

$$T(n) = \frac{1}{n}\sum_{i=1}^{n} T(i-1) + (n - i) + n + 1 = O(nlogn)$$

- Space complexity: S(n) = 2n = n + logn = O(n).
- Randomized quick sort choose pivot from random places make worst case O(n logn) but for array with same element worst case $O(n^2)$.

Then, Quick Sort Algorithm is as follows-

```
Partition_Array (a , beg , end , loc)
Begin
Set left = beg , right = end , loc = beg
Set done = false
While (not done) do
While ( (a[loc] <= a[right] ) and (loc ≠ right) ) do
Set right = right - 1
end while
if (loc = right) then
Set done = true
else if (a[loc] > a[right]) then
Interchange a[loc] and a[right]
Set loc = right
end if
if (not done) then
While ( (a[loc] >= a[left] ) and (loc ≠ left) ) do
Set left = left + 1
end while
if (loc = left) then
Set done = true
else if (a[loc] < a[left]) then
Interchange a[loc] and a[left]
Set loc = left
end if
end if
end while
End
```

Quick Sort is also based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting(major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element(Central element)
3. Elements greater than the pivot element

**Pivot** element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as **pivot**.

**For example:** In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as **pivot**. So after the first pass, the list will be changed like this.

{6 8 17 14 **25** 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements **smaller** to it on its left and all the elements **larger** than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sunarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

**How Quick Sorting Works**

Following are the steps involved in quick sort algorithm:

1. After selecting an element as pivot, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this partitioning. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the pivot will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the pivot element will be at its final sorted position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of pivot and elements on the right of pivot, and we perform partitioning on them by choosing a pivot in the subarrays.

**Quick Sort Analysis**-

To find the location of an element that splits the array into two parts, O(n) operations are required.

This is because every element in the array is compared to the partitioning element.

After the division, each section is examined separately.

If the array is split approximately in half (which is not usually), then there will be log2n splits.

Therefore, total comparisons required are f(n) = n x log2n = O(nlog2n).

Order of Quick Sort = O(nlog2n)

**Worst Case-**

Quick Sort is sensitive to the order of input data.

It gives the worst performance when elements are already in the ascending order.

It then divides the array into sections of 1 and (n-1) elements in each call.

Then, there are (n-1) divisions in all.

Therefore, here total comparisons required are f(n) = n x (n-1) = O(n2).

Order of Quick Sort in worst case = O(n2)

**Advantages of Quick Sort-**

The advantages of quick sort algorithm are-

Quick Sort is an in-place sort, so it requires no temporary memory.

Quick Sort is typically faster than other algorithms.

(because its inner loop can be efficiently implemented on most architectures)

Quick Sort tends to make excellent usage of the memory hierarchy like virtual memory or caches.

Quick Sort can be easily parallelized due to its divide and conquer nature.

**Disadvantages of Quick Sort-**

The disadvantages of quick sort algorithm are-

The worst case complexity of quick sort is O(n2).

This complexity is worse than O(nlogn) worst case complexity of algorithms like merge sort, heap sort etc.

It is not a stable sort i.e. the order of equal elements may not be preserved.

**6. Matrix Multiplication-**

- Using DAC: $T(n) = \begin{cases} O(1), & \text{for } n = 1 \\ 8\ T(n/2) + O(n^2), & \text{for } n > 1 \end{cases}$

  Time complexity = $O(n^3)$

- Strassen's Matrix Multiplication:

  $T(n) = \begin{cases} O(1) & , \quad \text{for } n = 1 \\ 7\ T(n/2) + an^2, & \text{for } n > 1 \end{cases}$

  Time complexity = $O(n^{2.81})$ by Strassen's.

  Time complexity = $O(n^{2.37})$ by Coppersmith and Winograd.

- Karatsuba algorithm for fast multiplication of two n-digit numbers required exactly $n^{\log_2 3}$ (when n is a power of 2) using Divide and Conquer.

**Sorting Techniques**

**1. Bubble Sort-**

Bubble sort is the easiest sorting algorithm to implement.

It is inspired by observing the behaviour of air bubbles over foam.

It is an in-place sorting algorithm.

It uses no auxiliary data structures (extra space) while sorting.

**How Bubble Sort Works**

Bubble sort uses multiple passes (scans) through an array.

In each pass, bubble sort compares the adjacent elements of the array.

It then swaps the two elements if they are in the wrong order.

In each pass, bubble sort places the next largest element to its proper position.

In short, it bubbles down the largest element to its correct position.

**Bubble Sort Algorithm-**

The bubble sort algorithm is given below-

```
for(int pass=1 ; pass<=n-1 ; ++pass)     // Making passes through array
{
   for(int i=0 ; i<=n-2 ; ++i)
   {
      if(A[i] > A[i+1])              // If adjacent elements are in wrong order
         swap(i,i+1,A);              // Swap them
   }
}
//swap function : Exchange elements from array A at position x,y
void swap(int x, int y, int[] A)
{
   int temp = A[x];
   A[x] = A[y];
   A[y] = temp;
   return ;
}
```

**Time Complexity Analysis-**

Bubble sort uses two loops- inner loop and outer loop.

The inner loop deterministically performs O(n) comparisons.

**Worst Case–**

In worst case, the outer loop runs O(n) times.

Hence, the worst case time complexity of bubble sort is O(n x n) = O(n2).

**Best Case-**

In best case, the array is already sorted but still to check, bubble sort performs O(n) comparisons.

Hence, the best case time complexity of bubble sort is O(n).

**Average Case-**

In average case, bubble sort may require (n/2) passes and O(n) comparisons for each pass.

Hence, the average case time complexity of bubble sort is O(n/2 x n) = Θ(n2).

The following table summarizes the time complexities of bubble sort in each case-

**Time Complexity**

Best Case O(n)

Average Case Θ(n2)

Worst Case O(n2)

From here, it is clear that bubble sort is not at all efficient in terms of time complexity of its algorithm.

**Space Complexity Analysis-**

Bubble sort uses only a constant amount of extra space for variables like flag, i, n.

Hence, the space complexity of bubble sort is O(1).

It is an in-place sorting algorithm i.e. it modifies elements of the original array to sort the given array.

**Properties-**

Some of the important properties of bubble sort algorithm are-

Bubble sort is a stable sorting algorithm.

Bubble sort is an in-place sorting algorithm.

The worst case time complexity of bubble sort algorithm is O(n2).

The space complexity of bubble sort algorithm is O(1).

Number of swaps in bubble sort = Number of inversion pairs present in the given array.

Bubble sort is beneficial when array elements are less and the array is nearly sorted.

**Problem-01:**

The number of swapping needed to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order using bubble sort is-

A. 11

B. 12

C. 13

D. 10

Solution-

In bubble sort, Number of swaps required = Number of inversion pairs.

Here, there are 10 inversion pairs present which are-

(8,7)

(22,7)

(22,9)

(8,5)

(22,5)

(7,5)

(9,5)

(31,5)

(22,13)

(31,13)

Thus, Option (D) is correct.

**Problem-02:**

When will bubble sort take worst-case time complexity?

A. The array is sorted in ascending order.

B. The array is sorted in descending order.

C. Only the first half of the array is sorted.

D. Only the second half of the array is sorted.

Solution-

In bubble sort, Number of swaps required = Number of inversion pairs.

When an array is sorted in descending order, the number of inversion pairs = n(n-1)/2 which is maximum for any permutation of array.

Thus, Option (B) is correct.

## 2. Selection Sort-

- Selection sort is one of the easiest approaches to sorting.
- It is inspired from the way in which we sort things out in day to day life.
- It is an in-place sorting algorithm because it uses no auxiliary data structures while sorting.

### How Selection Sort Works

Consider the following elements are to be sorted in ascending order using selection sort-

6, 2, 11, 7, 5

Selection sort works as-

- It finds the first smallest element (2).
- It swaps it with the first element of the unordered list.
- It finds the second smallest element (5).
- It swaps it with the second element of the unordered list.
- Similarly, it continues to sort the given elements.

As a result, sorted elements in ascending order are-

2, 5, 6, 7, 11

### Selection Sort Algorithm-

Let A be an array with n elements. Then, selection sort algorithm used for sorting is as follows-

```
for (i = 0 ; i < n-1 ; i++)
{
index = i;
for(j = i+1 ; j < n ; j++)
{
if(A[j] < A[index])
index = j;
}
```

```
temp = A[i];
A[i] = A[index];
A[index] = temp;
}
```

Here,

- i = variable to traverse the array A
- index = variable to store the index of minimum element
- j = variable to traverse the unsorted sub-array
- temp = temporary variable used for swapping

**Time Complexity Analysis-**

- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has $O(n^2)$ time complexity.

| | Time Complexity |
|---|---|
| Best Case | $n^2$ |
| Average Case | $n^2$ |
| Worst Case | $n^2$ |

**Space Complexity Analysis-**

- Selection sort is an in-place algorithm.
- It performs all computation in the original array and no other array is used.
- Hence, the space complexity works out to be O(1).
-

**3. Insertion Sort-**

- Insertion sort is an in-place sorting algorithm.
- It uses no auxiliary data structures while sorting.
- It is inspired from the way in which we sort playing cards.

**Insertion Sort Algorithm-**

Let A be an array with n elements. The insertion sort algorithm used for sorting is as follows-

```
for (i = 1 ; i < n ; i++)
{
key = A [ i ];
j = i - 1;
```

```
while(j > 0 && A [ j ] > key)
{
A [ j+1 ] = A [ j ];
j--;
}
A [ j+1 ] = key;
}
```

Here,

- i = variable to traverse the array A
- key = variable to store the new number to be inserted into the sorted sub-array
- j = variable to traverse the sorted sub-array

**Time Complexity Analysis-**

- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has $O(n^2)$ time complexity.

| | **Time Complexity** |
|---|---|
| Best Case | n |
| Average Case | $n^2$ |
| Worst Case | $n^2$ |

**Space Complexity Analysis-**

- Selection sort is an in-place algorithm.
- It performs all computation in the original array and no other array is used.
- Hence, the space complexity works out to be O(1).

**Important Notes-**

- Insertion sort is not a very efficient algorithm when data sets are large.
- This is indicated by the average and worst case complexities.
- Insertion sort is adaptive and number of comparisons are less if array is partially sorted.

**Example-**

12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13

11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

****