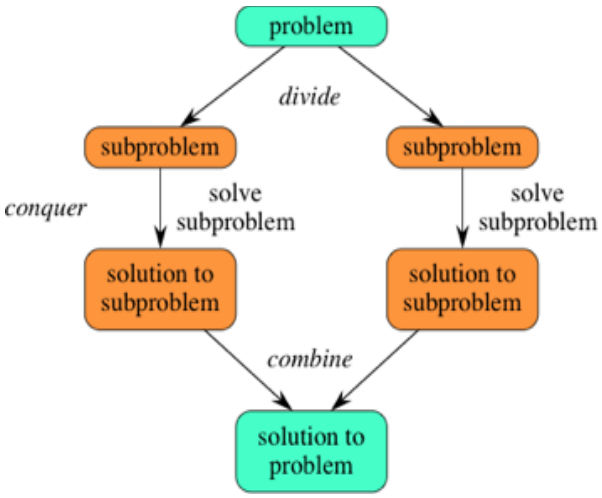
Module-2

Sl. No	Questions	Marks
1	Explain general method of divide and conquer problem.	8
2	Apply binary search algorithm for following array elements: 3,10,15,20,35,40,60. And analysis the same.	14
3	Construct the algorithm for search and insert an element in a Binary search tree. Tabulate the best, worst, and average-case complexity of the binary search tree for each operation	7
4	Construct the algorithm for the Breadth-First Search (BFS) traversal of a directed graph. Mention the best, worst, and average-case complexity of the BFS	7
4	Construct the algorithm for Depth-First Search (DFS) traversal of a directed graph. Mention the best, worst, and average-case complexity of the DFS	8
5	List Difference between BFS and DFS	7
6	Construct the binary search tree for the following data elements: 45, 15, 79, 90, 10, 55, 12, 20, 50 Trace the algorithm which you have constructed in 3a for every insertion and depict the final tree	7
7	Design an algorithm to sort the n number of elements using the divide and conquer technique and provide the time complexities for best average, and worst case.	8
8	Sort the following elements using the above-designed divide and conquer algorithm 100, 25, 98, 54, 79, 64, 84, 26, 48 and16. Trace the algorithm for each and every process	8
9	A man has 10 varieties of fish: fish1, fish2 …… fish10 namely, in his aquarium. The number of fishes from fish1, fish2…. fish10 are as follows: 48, 56, 24, 96, 68, 59, 71, 84, 99, and 35. Apply the divide and conquer methodology to find the name of the fish which is maximum and minimum in the aquarium	7
10	Suggest and design the suitable algorithm for the above problem and also provide the time complexity for that algorithm	8
11	Discuss Quick Sort Algorithm and Explain it with example. Derive Worst case and Average Case Complexity.	14
12	Explain Strassen's matrix multiplication. Evaluate it's efficiency.	14

Q1. Explain general method of divide and conquer problem.

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1<k<=n, yielding 'k' sub problems.
- These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.

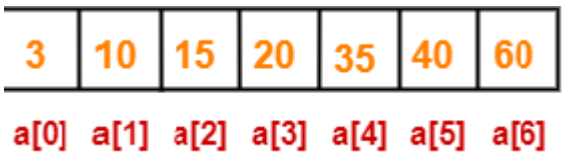
- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.
- For those cases the re application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.
- D And C(Algorithm) is initially invoked as D and C(P), where 'p' is the problem to be solved.
- Small(P) is a Boolean-valued function that determines whether the i/p size is small enough that the answer can be computed without splitting.
- If this so, the function 'S' is invoked.
- Otherwise, the problem P is divided into smaller sub problems.



- Divide the problem into a number of sub-problems that are smaller instances of the same problem.
- Conquer the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- Combine the solutions to the sub-problems into the solution for the original problem.

Q2. Apply binary search algorithm for following array elements 3,10,15,20,35,40,60. And analysis the same.

Consider an array consists of the following numbers: 3,10,15,20,35,40,60. Search for 15 using binary search.



Binary Search Algorithm works in the following steps-

Step-01:

We take low=0 and high=6.

- We compute location of the middle element as-

mid= (low + high) / 2 = (0 + 6) / 2 = 3

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

Step-02:

Since a[mid] = 20 > 15, so we take high = mid – 1 = 3 – 1 = 2 whereas low remains unchanged.

- We compute location of the middle element as-

mid= (low + high) / 2 = (0 + 2) / 2 = 1

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

Step-03:

Since a[mid] = 10 < 15, so we take low = mid + 1 = 1 + 1 = 2 whereas high remains unchanged.

- We compute location of the middle element as: mid= (low + high) / 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.

Analysis of Binary Search:

- Binary Search can be analysed with the best, worst, and average case number of comparisons. These analyses are dependent upon the length of the array, so let N =|A| denote the length of the Array A.
- The numbers of comparisons for the recursive and iterative versions of Binary Search are the same, if comparison counting is relaxed slightly. For Recursive Binary Search, count each pass through the if-then- else block as one comparison. For Iterative Binary Search, count each pass through the while block as one comparison.
- **Best case - O (1) comparisons**: In the best case, the item X is the middle in the array A. A constant number of comparisons (actually just 1) are required.
- **Worst case - O (log n) comparisons**: In the worst case, the item X does not exist in the array A at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done ceiling (log n) times. Thus, ceiling (log n) comparisons are required.
- **Average case - O (log n) comparisons:** To find the average case, take the sum over all elements of the product of number of comparisons required to find each element and the probability of searching for that element. To simplify the analysis, assume that no item

which is not in A will be searched for, and that the probabilities of searching for each element are uniform.

Q3. Construct the algorithm for search and insert an element in a Binary search tree. Tabulate the best, worst, and average-case complexity of the binary search tree for each operation. 7M

Searching Algorithm

Input: key elements
Output: return the elements if found, otherwise returns null

```
struct node* search(int data){
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data){

      if(current != NULL) {
         printf("%d ",current->data);

         //go to left tree
         if(current->data > data){
            current = current->leftChild;
         }  //else go to right tree
         else {
            current = current->rightChild;
         }

         //not found
         if(current == NULL){
            return NULL;
         }
      }
   }

   return current;
}
```

Inserting Algorithm

Input: key elements
Output: insert key elements in the existing BST or create a new BST if the tree is empty

```
void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;
```

```
    //if tree is empty
    if(root == NULL) {
      root = tempNode;
    } else {
      current = root;
      parent = NULL;

      while(1) {
        parent = current;

        //go to left of the tree
        if(data < parent->data) {
          current = current->leftChild;
          //insert to the left

          if(current == NULL) {
            parent->leftChild = tempNode;
            return;
          }
        }  //go to right of the tree
        else {
          current = current->rightChild;

          //insert to the right
          if(current == NULL) {
            parent->rightChild = tempNode;
            return;
          }
        }
      }
    }
  }
```

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Q4. Construct the algorithm for the Breadth-First Search (BFS) traversal of a directed graph. Mention the best, worst, and average-case complexity of the BFS.7M

Input − The list of vertices, and the start vertex.

Output − Traverse all of the nodes, if the graph is connected.

```
BFS(graph, start_node, end_node):
    frontier = new Queue()
```

```
frontier.enqueue(start_node)
explored = new Set()

while frontier is not empty:
   current_node = frontier.dequeue()
   if current_node in explored: continue
   if current_node == end_node: return success

   for neighbor in graph.get_neigbhors(current_node):
      frontier.enqueue(neighbor)
   explored.add(current_node)
```

Time complexity will be O(V+E) for all the cases.

Q5. Construct the algorithm for Depth-First Search (DFS) traversal of a directed graph. Mention the best, worst, and average-case complexity of the DFS.8M

Input − The list of vertices, and the start vertex.

Output − Traverse all of the nodes, if the graph is connected

```
DFS(graph, start_node, end_node):
   frontier = new Stack()
   frontier.push(start_node)
   explored = new Set()
   while frontier is not empty:
      current_node = frontier.pop()
      if current_node in explored: continue
      if current_node == end_node: return success

      for neighbor in graph.get_neigbhors(current_node):
         frontier.push(neighbor)
explored.add(current_node)
```

Time complexity will be O(V+E) for all the cases.

Q6. List Difference between BFS and DFS. 7M

BFS	DFS
BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
BFS is better when target is closer to Source.	DFS is better when target is far from source.
As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
BFS is slower than DFS.	DFS is faster than BFS.
Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

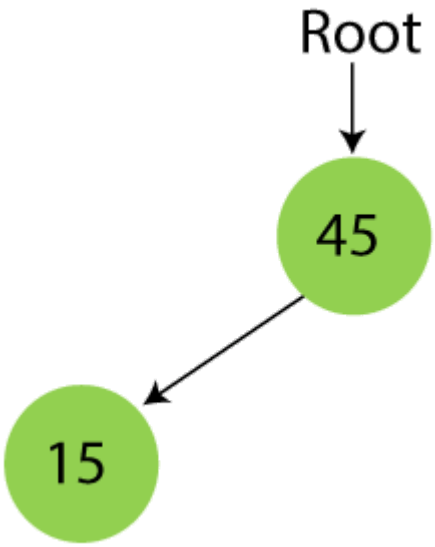
Q7. Construct the binary search tree for the following data elements:
45, 15, 79, 90, 10, 55, 12, 20, 50
Trace the algorithm which you have constructed in 3a for every insertion and depict the final

tree.7M

Step 1 - Insert 45.

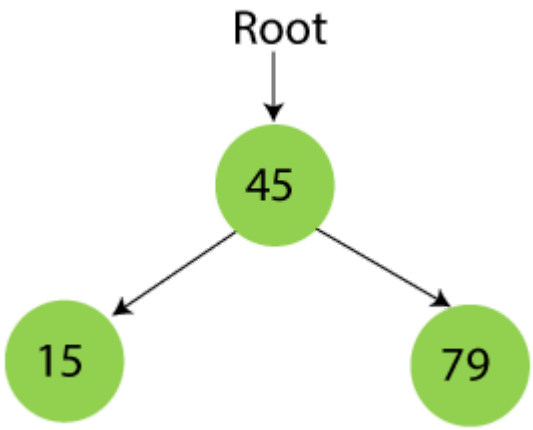
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



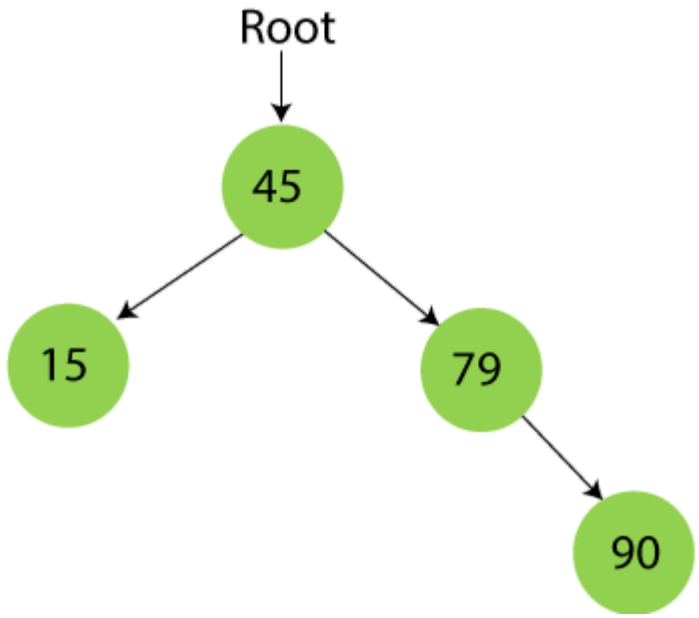
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



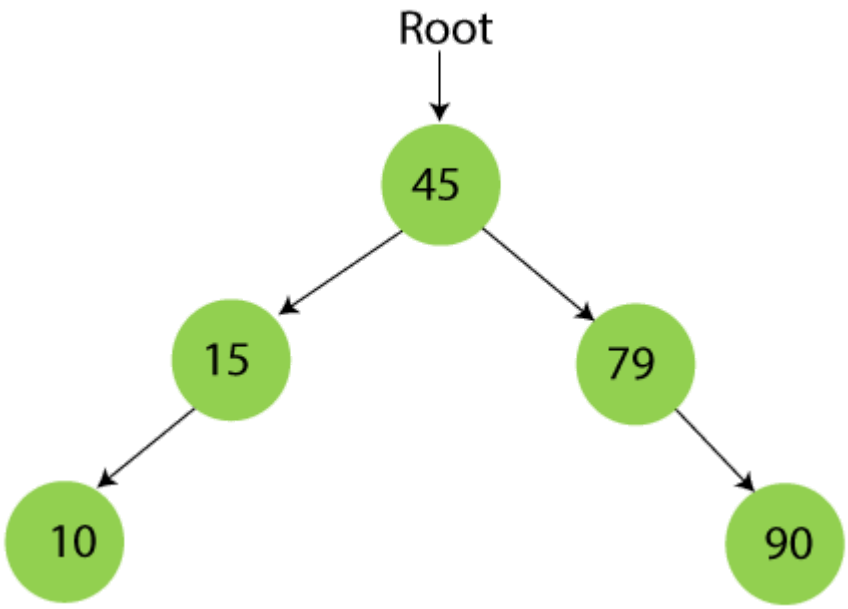
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



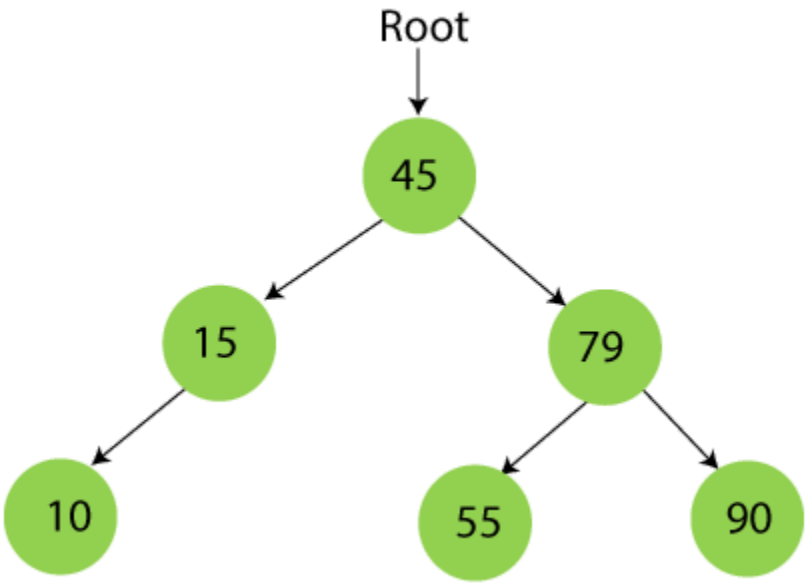
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



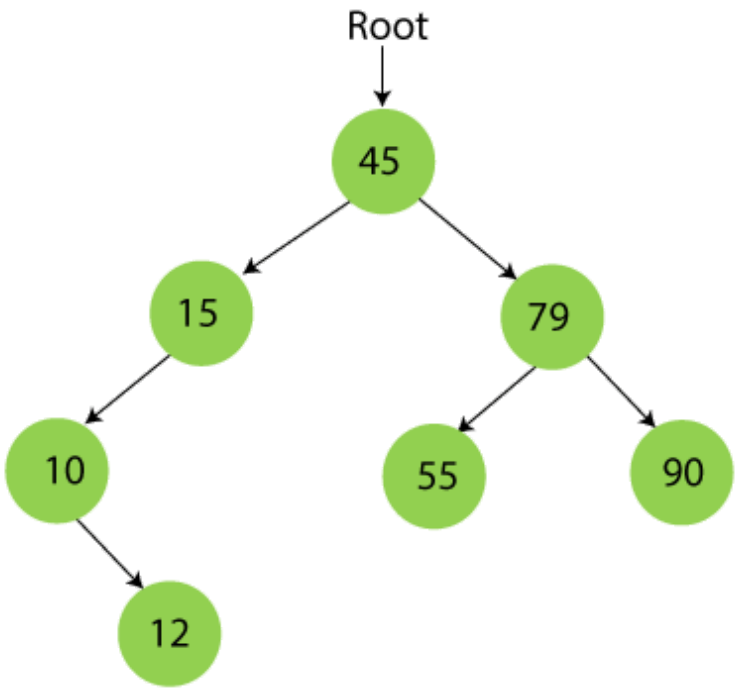
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



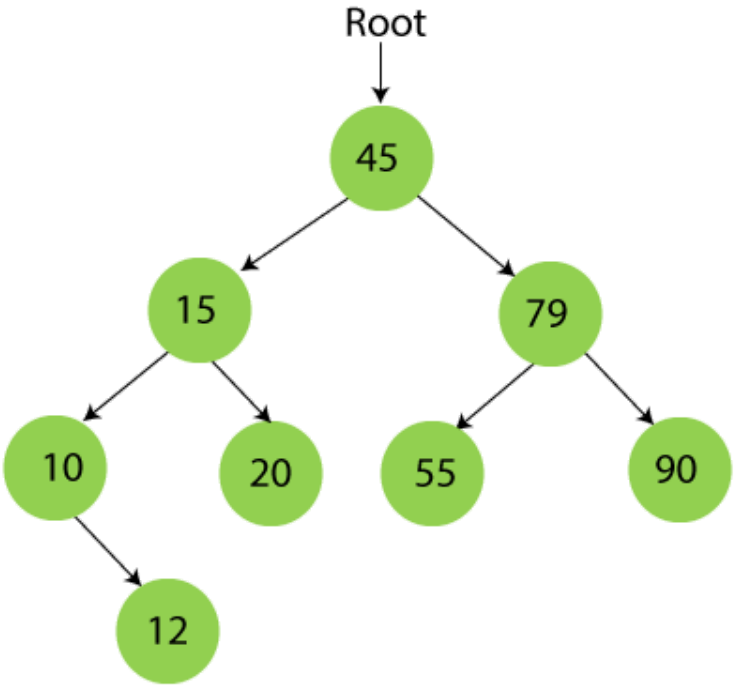
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



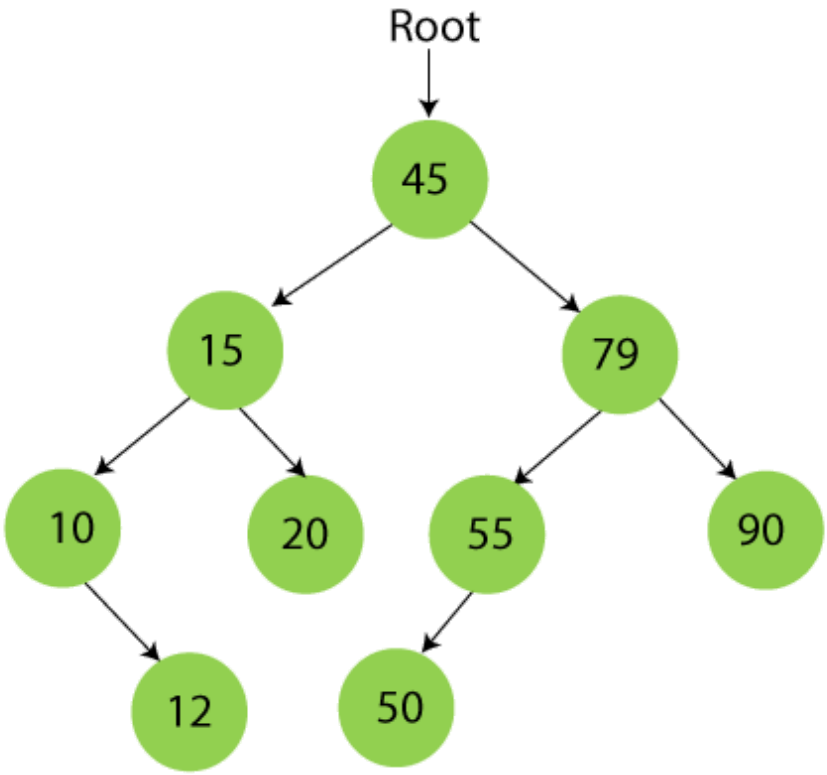
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed.

Q8. Design an algorithm to sort the n number of elements using the divide and conquer technique and provide the time complexities for best average, and worst case.

Design an algorithm to sort the n number of elements using the divide and conquer technique and provide the time complexities for best average, and worst case. 8M

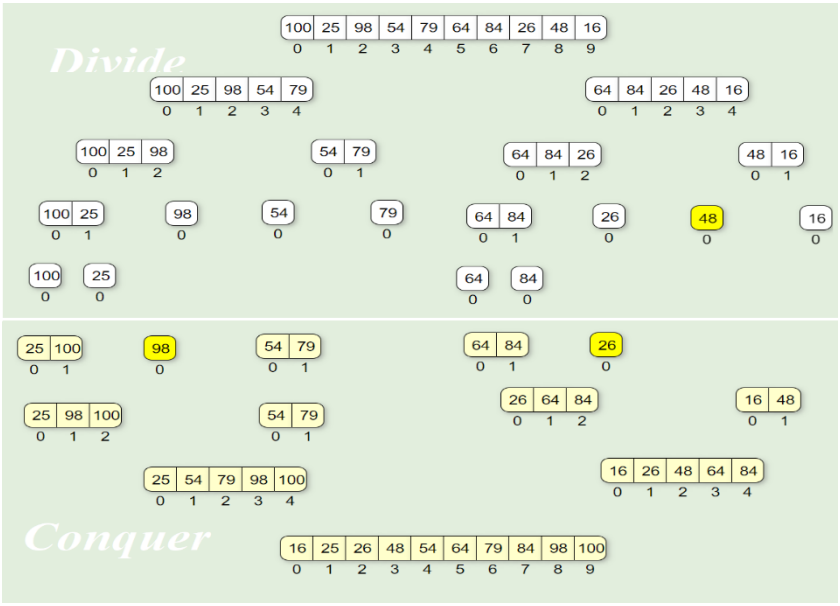
Either we can use Merge sort or Quick sort

Input: unsorted array of elements
Output: sorted array of elements

```
void merge(array, low, mid, high )
{
        int temp[MAX];
        int i = low;
        int j = mid +1 ;
        int k = low ;

        while( (i <= mid) && (j <=high) )
        {
                if(array[i] <= array[j])
                        temp[k++] = array[i++] ;
                else
                        temp[k++] = array[j++] ;
        }/*End of while*/

        while( i <= mid )
                temp[k++]=array[i++];

        while( j <= high )
                temp[k++]=array[j++];

        for(i= low; i <= high ; i++)
                array[i]=temp[i];

}/*End of merge()*/

void merge_sort( int low, int high )
{
        int mid;
        if( low != high )
        {
                mid = (low+high)/2;
                merge_sort( low , mid );
                merge_sort( mid+1, high );
                merge( low, mid, high );
        }
}/*End of merge_sort*/
```

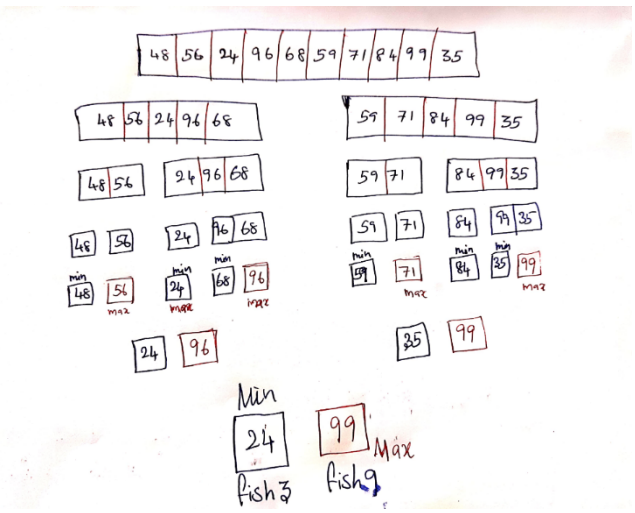
Time complexity of merge sort will be O(n log n) for all the cases.

Q6. Sort the following elements using the above-designed divide and conquer algorithm 100, 25, 98, 54, 79, 64, 84, 26, 48 and16. Trace the algorithm for each and every process.7M



Q9. A man has 10 varieties of fish:
fish1, fish2 …… fish10 namely, in his aquarium. The number of fishes from fish1, fish2…. fish10 are as follows:
48, 56, 24, 96, 68, 59, 71, 84, 99, and 35.
Apply the divide and conquer methodology to find the name of the fish which is maximum and minimum in the aquarium.8M

We can apply the Maxmin algorithm to find the minimum and the maximum number of fish in the aquarium in minimum time complexity using the divide and conquer technique.



Q10. Suggest and design the suitable algorithm for the above problem and also provide the time complexity for that algorithm. 7M

Algorithm MAXMIN (A, low, high)

```
// Description : Find minimum and maximum element from array using divide and conquer approach
// Input : Array A of length n, and indices low = 0 and high = n - 1
// Output : (min, max) variables holding minimum and maximum element of array

if n == 1 then
  return (A[1], A[1])
else if n == 2 then
  if A[1] < A[2] then
    return (A[1], A[2])
  else
    return (A[2], A[1])
else
  mid ← (low + high) / 2
  [LMin, LMax] = DC_MAXMIN (A, low, mid)
  [RMin, RMax] = DC_MAXMIN (A, mid + 1, high)
  if LMax > RMax then   // Combine solution
    max ← LMax
  else
    max ← RMax
  end
  if LMin < RMin then    // Combine solution
    min ← LMin
  else
    min ← RMin
  end
  return (min, max)
end
```

Q11. Discuss Quick Sort Algorithm and Explain it with example. Derive Worst case and Average Case Complexity.

Quick Sort and divide and conquer

• Divide: Partition array A[l..r] into 2 subarrays, A[l..s-1] and A[s+1..r] such that each element of the first array is ≤A[s] and each element of the second array is ≥ A[s]. (Computing the index of s is part of partition.)

• Implication: A[s] will be in its final position in the sorted array.

• Conquer: Sort the two subarrays A[l..s-1] and A[s+1..r] by recursive calls to quicksort

• Combine: No work is needed, because A[s] is already in its correct place after the partition is done, and the two subarrays have been sorted.

✓ **Steps in Quicksort**

• Select a pivot w.r.t. whose value we are going to divide the sublist. (e.g., p = A[l])

• Rearrange the list so that it starts with the pivot followed by a ≤ sublist (a sublist whose elements are all smaller than or equal to the pivot) and a ≥ sublist (a sublist whose elements are all greater than or equal to the pivot) Exchange the pivot with the last element in the first sublist(i.e., ≤ sublist) – the pivot is now in its final position

• Sort the two sublists recursively using quicksort.

The Quicksort Algorithm

ALGORITHM Quicksort(A[l..r])

//Sorts a subarray by quicksort

//Input: A subarray A[l..r] of A[0..n-1],defined by its left and right indices l and r
//Output: The subarray A[l..r] sorted in
nondecreasing order if l < r
s Partition (A[l..r]) // s is a
split position Quicksort(A[l..s1])
Quicksort(A[s+1..r]
ALGORITHM Partition (A[l ..r])
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n-1], defined by its left and right indices l and r (l < r)
//Output: A partition of A[l..r], with the split position returned as this
function's value P A[l]
i =l; j =r + 1;
Repeat
repeat i= i + 1 until A[i]>=p //leftright scan repeats j= j – 1 until A[j] <= p//right-left scan
if (i < j) //need to continue
with the scan swap(A[i], a[j])
until i >= j //no
need to scan swap(A[l], A[j])
return j

✓ Efficiency of Quicksort

Based on whether the partitioning is balanced.

Best case: split in the middle — Θ (n log n)

C(n) = 2C(n/2) + Θ(n) //2 subproblems of size n/2 each

Worst case: sorted array! — $\Theta(n^2)$

C(n) = C(n-1) + n+1 //2 subproblems of size 0 and n-1 respectively

Average case: random arrays — Θ (n log n)

Q12. Explain Strassen's matrix multiplication. Evaluate its efficiency.

1. Let A and B be two nn matrices, that is, each having n rows and n columns. If C=AB, then the product matricx C will also have n rows and n columns.
2. An element C[i, j] can now be found using the formula

   C[i,j]=∑k=0n−1A[i,k]*B[k,j]C[i,j]=∑k=0n−1A[i,k]*B[k,j]

3. The method of multiplying two matrices is known as the classic matrix multiplication method. Using the above formula, we may write the following statements:

for(i=0; i < n; i++)

for(j=0; j<n; j++)

```
{
   C[i] [j] =0;
   for( k=0;k<n;k++)
   C[i][j]=c[i][j]+a[i][k]*b[k][j];
}
```

It can be easily verified that the complexity of the method is $O(n^3)$.

Strassen's matrix multiplication can be used only when n is a power of 2i.e. $n=2^k$. If n is not a power of 2 then enough rows and columns of zeros can be added to both A and B so that the resulting dimension are a power of 2.

In this method matrix A and B are splitted into 4 squares sub-matrices where each sub-matrices has dimension of n/2. Now the product is found as shown:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$\begin{aligned}
P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
Q &= (A_{21} + A_{22})B_{11} \\
R &= A_{11}(B_{12} - B_{22}) \\
S &= A_{22}(B_{21} - B_{11}) \\
T &= (A_{11} + A_{12})B_{22} \\
U &= (A_{21} - A_{11})(B_{11} + B_{22}) \\
V &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
C_{11} &= P + S - T + V \\
C_{12} &= R + T \\
C_{21} &= Q + S \\
C_{22} P + R - Q + U
\end{aligned}$$

It is observed that all c [i, j] can be calculated using a total of 7 multiplications and 18 additions or subtractions.

It is not necessary that n should be always 2. If n>2 then the same formula can be used but now it will be done recursively. The same formula will be continuously applied on smaller sized matrices till n gets reduced to 2. When n=2, this will be terminating condition of recursion and now multiplication is done directly.

Analysis:

Recurrence relation for Strassen's Algorithm is:

$T(n) = 7T(n/2) + \Theta(n2)$

Applying the Master Theorem to $T(n) = a\ T(n/b) + f(n)$ with a=7, b=2, and $f(n)=\Theta(n^2)$.

Since $f(n) = O(n^{\log b(a)}-\varepsilon) = O(n^{\log 2(7)}-\varepsilon)$, case a) applies and

we get $T(n)= \Theta(n^{\log b(a)}) = \Theta(n^{\log 2(7)}) = O(n^{2.81})$.

Example:

A= [1,3] B=[6,8]

[7,5] [4,2]

// Step 1: Split A and B into half-sized matrices of size 1x1 (scalars).

a11=1, a12=3, a21=7, a22=5, b11=6, b12=8, b21=4, b22=2

// Define the "S" matrix.

s1=b12-b22 // 6

s2=a11+a12 // 4

s3= a21+a22 // 12

s4= b21-b11 // -2

s5=a11+a22 // 6

s6=b11+b22 //8

s7=a12-a22 //-2

s8=b21+b22 //6

s9=a11-a21 // -6

s10=b11 + b12 // 14

// Define the "P" matrix.

p1=a11*s1 // 6

p2=s2*b22 //8

p3=s3*b11 //72

p4=a22*s4//-10

p5=s5*s6 // 48

p6=s7*s8 //-12

p7=s9*s10 // -84

// Fill in the resultant "C" matrix.

c11 = p5 + p4 - p2 + p6 // 18

c12 = p1 + p2 // 14

c21 = p3 + p4 // 62

c22 = p5 + p1 - p3 - p7 // 66

C=[18,14]