UNIT-II

Disjoint sets & Divide and conquer

Objective: To get acquainted with disjoint sets and divide and conquer technique**.**

Syllabus: Disjoint Sets - Disjoint set operations, Union and Find algorithms, Bi-connected components.

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

Learning outcomes:

At the end of the unit, students will be able to

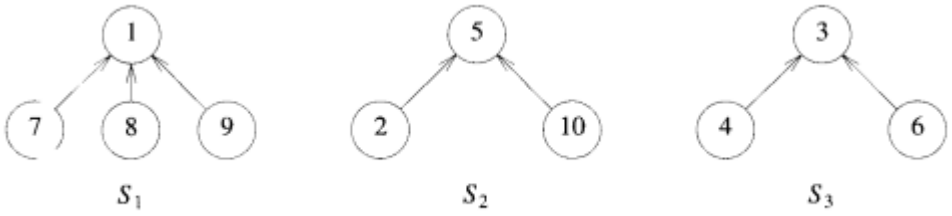
- Develop algorithms to perform union and find operations on disjoint sets.
- Find articulation points and bi-connected components of a graph.
- Design and analyze algorithms of binary search, quick sort and merge sort.
- Solve recurrence relations.

LEARNING MATERIAL

Disjoint Sets

- Assume that the elements of the sets are the numbers 1,2,3,..
- **Disjoint Sets**: If Si and S_j , i not equal to j, are two sets, then there is no element that is in both Si and Sj.
- For example, when n = 10,the elements can be partitioned into three disjoint sets,
  S_1 = {1,7,8,9}, S_2 = {2,5,10}, and S_3= {3,4,6}

- Each set is represented as tree .we have linked the nodes from the children to the parent, rather than our usual method of linking from the parent to the children.



Operations on Disjoint Sets

- The operations we wish to perform on these sets are:

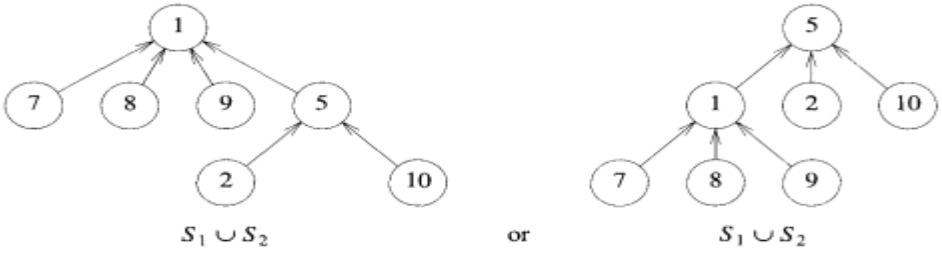
  1.Disjoin tset union : If Si and Sj are two disjoint sets, then their union Si U Sj= all elements x such that x is in Si or Sj. Thus, SiU S2 = {1,7, 8,9,2,5,10}.

  Since we have assumed that all sets are disjoint, we can assume that following the union of Si and Sj, the sets Si and Sj do not exist independently; that is, they are replaced by Si U Sj i.e. the collection of sets.

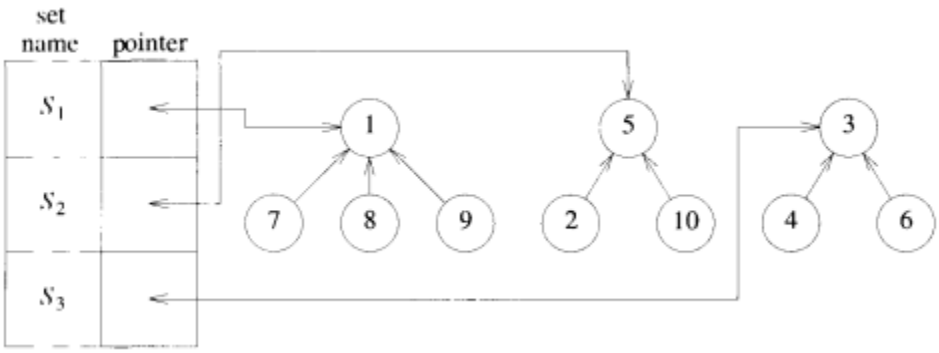
  2.Find(i): Given the element i, find the set containing i. Thus, 4 is in set S3,and 9 is in setS1.

Union and Find Operations

- Union: Suppose that we wish to obtain the union of ,Si and S2 .
- Since we have linked the nodes from children to parent, we simply make one of the trees a sub tree of the other.



- To obtain the union of two sets, all that has to be done is to set the parent field of one of the roots to the other root.
- This can be accomplished by keeping a pointer to the root of the tree representing that set.
- In addition, each root has a pointer to the set name, then to determine which set an element is currently in, we follow parent links to the root of its tree and use the pointer to the set name.



- If we determine that element i is in a tree with root j, and j has a pointer to entry k in the set name table, then the set name is just name[k].
- If we wish to unite sets S_i, and Sj, then we wish to unite the trees with roots FindPointer(S_i) and FindPointer(Sj).
- Here FindPointer is a function that takes a set name and determines the root of the tree that represents it.
- This is done by an examination of the [set name, pointer] table. In many applications the set name is just the element at the root.
- The operation of Find(i) now becomes: Determine the root of the tree containing element i. The function Union (i,j) requires two trees with roots i and j be joined.
- Since the set elements are numbered 1through n, we represent the tree nodes using an array p[1: n], where n is the maximum number of elements.

- The i^{th} element of this array represents the tree node that contains element i. This array element gives the parent pointer of the corresponding tree node. root nodes have a parent of -1.

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	−1	5	−1	3	−1	3	1	1	1	5

- We can now implement Find(i) by following the indices, starting at i until we reach a node with parent value - 1.
- For example, Find(6) starts at 6 and then moves to 6'sparent, 3. Since p[3] is negative, we have reached the root.
- The operation Union(i,j) is equally simple. We pass in two trees with roots i and j. Adopting the convention that the **first tree becomes a sub tree of the second**, the statement p[i] :=j; accomplishes the union.

```
Algorithm SimpleUnion(i, j)
{
    p[i] := j;
}

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

Performance of Union and Find operations

- The two algorithms are very easy but their performance characteristics are not very good.
- For instance, if we start with q elements each in a set of its own (that is,Si = {i}, 1< i < q), then the initial configuration consists of a forest with q nodes, and p[i] = 0,1<= i<= q).

  Union(l,2),Union(2,3), Union(3,4), Union(4,5),... , Union(n- l, n)

  Find(1),Find{2),..., Find(n)

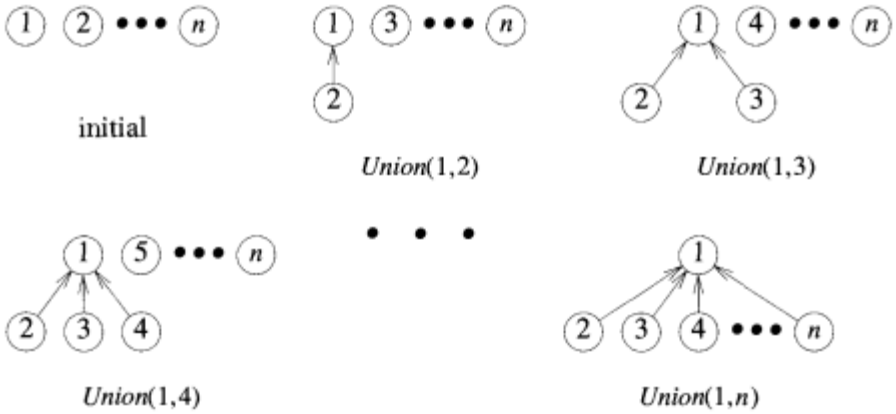
- This results in the following degenerate tree:

- Since the time taken for a union is constant, the n-1unions can be processed in time **O(n).**
- Each find requires following a sequence of parent pointers from the element to be found to the root.
- Since the time required to process a find for an element at level i of a tree is **O(i),** the total time needed to process the n finds is **$O(n^2)$.**

Weighting rule for Union

- We can improve the performance of our union and find algorithms by avoiding the creation of degenerate trees.
- To accomplish this, we make use of a **weighting rule for Union (i, j).**
- **Weighting rule: If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i; otherwise make i the parent of j.**
- When we use the weighting rule to perform the sequence of set unions given before, we obtain the following:
- To implement the weighting rule, we need to know how many nodes there are in every tree.
- To do this easily, we maintain a **count field** in the root of every tree.
- If i is a root node, then **count[i]** equals the number of nodes in that tree.

- Since all nodes other than the roots of trees have a positive number in the **p field**, we can maintain the count in the p field of the roots as a negative number.
- Time required to perform a union has increased somewhat but is still bounded by a constant, it is O(l)).The find algorithm remains unchanged.



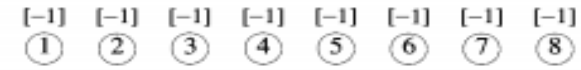

```
Algorithm WeightedUnion(i, j)
// Union sets with roots i and j, i ≠ j, using the
// weighting rule. p[i] = -count[i] and p[j] = -count[j].
{
    temp := p[i] + p[j];
    if (p[i] > p[j]) then
    { // i has fewer nodes.
        p[i] := j; p[j] := temp;
    }
    else
    { // j has fewer or equal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

- Let T be a tree with m nodes created by **WeightedUnion.** Consider the last union operation performed, Union(k,j).
- Let **a** be the number of nodes in tree j, and **m-a** be the number in k. Without loss of generality we can assume1 < = a < =m/2. Then the height of T is either the same as that of k or is one more than that of j.

- **The height of T is not more than $\log_2 m+1$.**
- Consider the behavior of Weighted Union on the following sequence of unions starting from the initial configuration p[i] = - count[i] = -1,1<= i<=8=n.

  Union(l,2),Union(3,4), Union(5,6), Union(7,8),

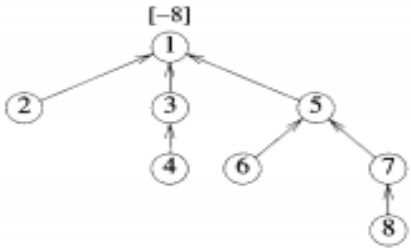
  Union(1,3),Union(5,7), Union(1,5)



(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)

Collapsing rule for Find

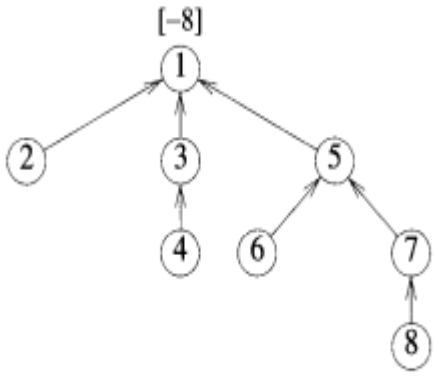
- The time to process a find is O(log m) if there are m elements in a tree.
- If an inter mixed sequence of u-1 union and f find operations are to be processed, the time becomes O(u+f log u),as no tree has more than u nodes in it.

- We need O(n) additional time to initialize the n-tree forest.
- The modification can be made in the find algorithm using the **collapsing rule.**
- **Collapsing rule: 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].**

```
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;
    }
    return r;
}
```

- Now process the following eight finds:

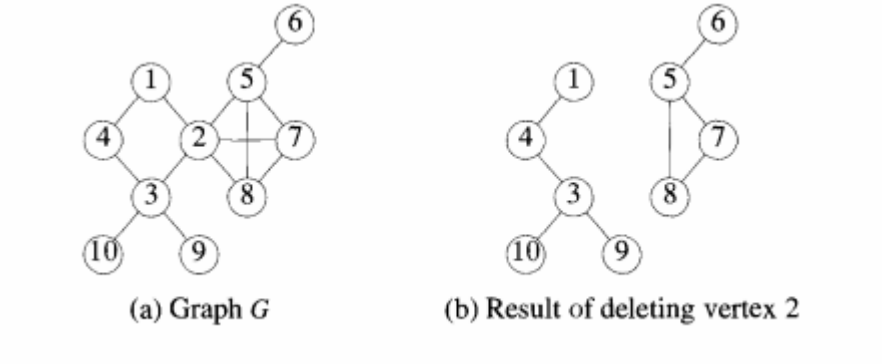
  Find(8),Find(8),...,Find(8)
- If **SimpleFind** is used,each Find(8) requires going up three parent link fields for a total of 24 moves to process all eight finds.
- When **CollapsingFind** is used, the first Find(8) requires going up three links and then resetting two links.
- Note that even though only two parent links need to be reset, CollapsingFind will reset three(the parent of 5 is reset to 1).
- Each of the remaining seven finds requires going up only one link field. The total cost is now only 13 moves.



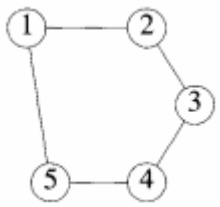
- In the algorithms WeightedUnion and CollapsingFind, use of the collapsing rule roughly doubles the time for an individual find. However, it reduces the worst-case time over a sequence of finds.

Biconnected Components

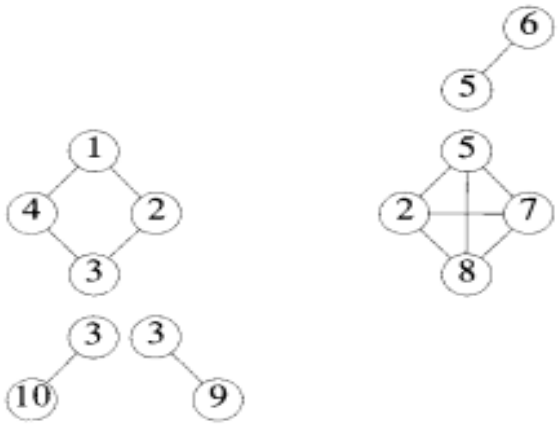
- **Articulation Point:** A vertex v in a connected graph G is an articulation point if and only if the deletion of vertex v together with all edges incident to v disconnects the graph into two or more nonempty components.



- In the connected graph **vertex 2 is an articulation point** as the deletion of vertex 2 and edges (1,2),(2,3),(2,5),(2,7), and (2,8) leaves behind two disconnected nonempty components Graph G has two other articulation points: vertex 5 and vertex 3.
- Note that if any of the remaining vertices is deleted from G, then exactly one component remains.
- **Biconected Graph: A graph G is biconnected** if and only if it contains no articulation points.



- The **presence of articulation points in a connected graph is an undesirable feature** in many cases.
- For example, if G represents a communication network with the vertices representing communication stations and the edges communication lines, then the failure of a communication station i that is an articulation point would result in the loss of communication to points other than i too.
- On the other hand, if G has no articulation point, then if any station i fail, we can still communicate between every two stations not including station i.
- We have to develop an efficient algorithm to test whether a connected graph is biconnected. For the case of graphs that are not biconnected:
  - Identify all the articulation points.
  - Determine a set of edges whose inclusion makes the graph biconnected.
  - Determining such a set of edges is facilitated if we know the **maximal sub graphs** of G that are biconnected.
  - G'= (V',E') is a maximal biconnected sub graph of G if and only if G has no biconnected subgraph G" = (V", E") such that V□V" and E'CE".
- **Biconnected Component:** A maximal biconnected subgraph is a **biconnected component.**
- **Two biconnected components can have at most one vertex in common** and this vertex is an articulation point.
- Every biconnected component of a connected graph G contains at least two vertices.
- The following are the biconnected components regarding the articulation points: **2 3 and 5.**



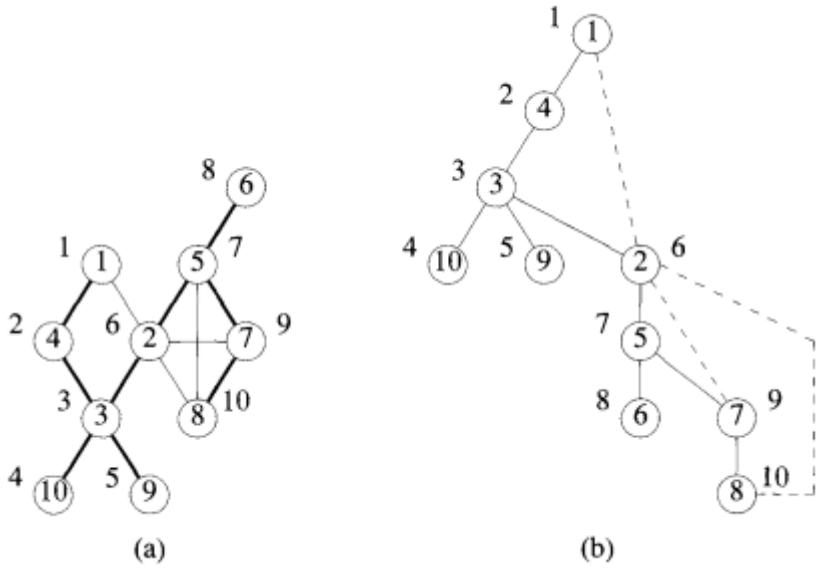
Algorithm to construct a bi connected Graph

```
for each articulation point a do
{
    Let B1, B2, ..., Bk be the biconnected
    components containing vertex a;
    Let vi, vi ≠ a, be a vertex in Bi, 1 ≤ i ≤ k;
    Add to G the edges (vi, vi+1), 1 ≤ i < k;
}
```

- By using the above scheme, we have to add edges (4,10) and (10,9) (corresponding to the articulation point 3),edge(1,5) (corresponding to the articulation point 2), and edge(6,7) (corresponding to point 5).
- Once the edges (V_i, V_{i+1}) are added, **vertex a** is no longer an articulation point.
- If the edges corresponding to all articulation points are added, G has no articulation points and so is biconnected.
- **If G has p articulation points and b biconnected components, then it introduces exactly b - p new edges into G**.
- But the scheme is using more than the minimum number of edges needed to make G biconnected.

Depth First Spanning Tree

- The problem can be efficiently solved by considering a depth first spanning tree of G by identifying the articulation points and biconnected



components of a connected graph G with n > 2 vertices.

- The numbers outside each vertex correspond to the order in which a depth first search visits these vertices and are referred to as the **depth first numbers (dfn's)** of the vertex. Thus, dfn[l]= 1,dfn[4] = 2,dfn[6] = 8,and so on.
- In solid edges form the depth first spanning tree. These edges are called **"tree edges"**. Broken edges (i.e., all the remaining edges) are called **"back edges".**
- If (u,v) is any edge in G, then relative to the depth first spanning tree t, either u is an ancestor of v or v is an ancestor of it. So, there are **no cross edges** relative to a depth first spanning tree
- An edge **(u,v) is a cross edge** relative to t if and only if u is not an ancestor of v and v is not an ancestor of u.
- The **root node of a depth first spanning tree is an articulation point iff it has at least two children.**

- If it is **any other vertex, then it is not an articulation point iff from every child w of u, it is possible to reach an ancestor of u using only a path made up of descendents of w and a back edge.**
- If this cannot be done for some child w of u, then the deletion of vertex u leaves behind at least two nonempty components -one containing the root and the other containing vertex w.
- For each vertex u, define L[u] as follows:

  L[u] = min {dfn[u], min {L[w]\\ w is a child of u},min {dfn[w}\\ (u,w) is a back edge}}
- L[u] is the lowest depth first number that can be reached from u using a path of descendents followed by at most one back edge.
- If u is not the root, then it is an articulation point iff it has a child w such that

  L[w] > dfn[u].
- **For spanning tree in Fig (a):** L values are L[l: 10]= {1,1,1,1,6,8,6,6,5,4}. Vertex 3 is an articulation point as child 10 has L[10] = 4 and dfn[3]= 3. Vertex 2 is an articulation point as child 5 has L[5]= 6 and dfn[2] = 6.The only other articulation point is vertex 5; child 6 has L[6]= 8 and dfn[5] = 7.
- To determine the articulation points, it is necessary to perform a depth first search of the graph G and visit the nodes in the resulting depth first spanning tree in post order.
- Once L[l:n] has been computed, the articulation points can be identified in O(e) time. The total complexity is O(n+e), where e is the number of edges and n is the number of nodes in G, the articulation points of G can be determined in O(n+e) time.
- **The following algorithm calculates the articulation points by using L values and dfn's:**

```
Algorithm Art(u, v)
// u is a start vertex for depth first search. v is its parent if any
// in the depth first spanning tree. It is assumed that the global
// array dfn is initialized to zero and that the global variable
// num is initialized to 1. n is the number of vertices in G.
{
    dfn[u] := num; L[u] := num; num := num + 1;
    for each vertex w adjacent from u do
    {
        if (dfn[w] = 0) then
        {
            Art(w, u); // w is unvisited.
            L[u] := min(L[u], L[w]);
        }
        else if (w ≠ v) then L[u] := min(L[u], dfn[w]);
    }
}
```

- **The following algorithm constructs the biconnected components:**

```
Algorithm BiComp(u, v)
// u is a start vertex for depth first search. v is its parent if
// any in the depth first spanning tree. It is assumed that the
// global array dfn is initially zero and that the global variable
// num is initialized to 1. n is the number of vertices in G.
{
    dfn[u] := num; L[u] := num; num := num + 1;
    for each vertex w adjacent from u do
    {
        if ((v ≠ w) and (dfn[w] < dfn[u]))then
            add (u, w) to the top of a stack s;
        if (dfn[w] = 0) then
        {
            if (L[w] ≥ dfn[u]) then
            {
                write ("New bicomponent");
                repeat
                {
                    Delete an edge from the top of stack s;
                    Let this edge be (x, y);
                    write (x, y);
                } until (((x, y) = (u, w)) or ((x, y) = (w, u)));
            }
            BiComp(w, u); // w is unvisited.
            L[u] := min(L[u], L[w]);
        }
        else if (w ≠ v) then L[u] := min(L[u], dfn[w]);
    }
}
```

Divide and conquer (D & C)

- **Divide and conquer** (**D&C**) is an algorithm design paradigm based on multi-branched recursion.
- A D &C algorithm works by recursively breaking down a problem into smaller sub-problems and then each problem is solved independently.
- The solutions to the sub-problems are then combined to give a solution to the original problem.
- The D&C can be of three steps.
    - Divide / Break
    - Conquer / Solve
    - Merge / Combine
- The D & C applications- Binary search, Quick sort, Merge sort.

GENERAL METHOD

- 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.
- Often the sub problems resulting from a divide-and conquer design are of the same type as the original problem.
- Divide-and-conquer principle is naturally expressed by a recursive algorithm.
- **Control Abstarction:** It is procedure where the primary operations are specified by other procedures whose precise meanings are left undefined.
- **DAndC algorithm** is initially invoked as DAndC(P), where **P is the problem to be solved**. **Small(P)** is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this is so, the **function S** is invoked.

- Otherwisethe problem P is divided into smaller sub problems. The sub problems P1, P2, P3, ….,Pk are solved by recursive applications of DAndC.
- **Combine** is a function that determines the solution to P using the solutions to the k sub problems.

Control abstraction for divide-and-conquer

```
1   AlgorithmDAndC(P)
2   {
3           if Small(P)thenreturnS(P);
4           else
5           {
6                   divide P into smallerinstancesPi,P2,..P.k,, k > 1;
7                   Apply DAndC to eachof thesesubproblems;
8                   returnCombine(DAndC(Pi),DAndC(P..2,D)rAndC(Pfc));
9           }
10  }
```

- If the size of P is n and the sizes of the k sub problems are $n_1, n_2, n_3 \ldots n_k$, respectively,
  then the computing time of DAndC is described by the recurrence relation.

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

Where

T(n) is the time for Algorithm on any input of size n and
g(n) is the time to compute the answer directly for small inputs.

The function f(n) is the time for dividing P and combining the solutions to sub problems.

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

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

Where a and b are known constants. We assume that T(l)is known and n is a power of b (i.e.n,= b^k).

Example:

$$T(n) = \begin{cases} T(1) & n = 1 \\ T(n/2) + c & n > 1 \end{cases}$$

Example:

Consider the case in which a = 2 and 6 = 2.

Let T(l) and f(n) = n. We have

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

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

= 4T(n/4) +2n

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

= 8T(n/8)+ 3n

In general, we see that $T(n) = 2^i T(n/2^i) + i\,n$, for any $\log_2 n \geq i \geq 1$.

Applications of Divide and Conquer approach:

1. Binary Search
2. Merge Sort
3. Quick Sort

BINARYSEARCH

- **Binary search**, also known as **half-interval search** or **logarithmic search**,
- It finds the position of a target value within a sorted array.

Binary search (Iterative)

```
1   Algorithm BinSearch(a, n, x)
2   // Given an array a[1 : n] of elements in nondecreasing
3   // order, n ≥ 0, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       low := 1; high := n;
7       while (low ≤ high) do
8       {
9           mid := ⌊(low + high)/2⌋;
10          if (x < a[mid]) then high := mid − 1;
11          else if (x > a[mid]) then low := mid + 1;
12                else return mid;
13      }
14      return 0;
15  }
```

Binary search (Recursive)

```
1   Algorithm BinSrch(a, i, l, x)
2   // Given an array a[i : l] of elements in nondecreasing
3   // order, 1 ≤ i ≤ l, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       if (l = i) then
7       {
8           if (x = a[i]) then return i;
9           else return 0;
10      }
11      else
12      {
13          mid := ⌊(i + l)/2⌋;
14          if (x = a[mid]) then return mid;
15          else if (x < a[mid]) then
16                    return BinSrch(a, i, mid − 1, x);
17              else return BinSrch(a, mid + 1, l, x);
18      }
19  }
```

- **Example :**

-15,-6, 0, 7, 9, 23,54, 82,101,112,125,131,142,151

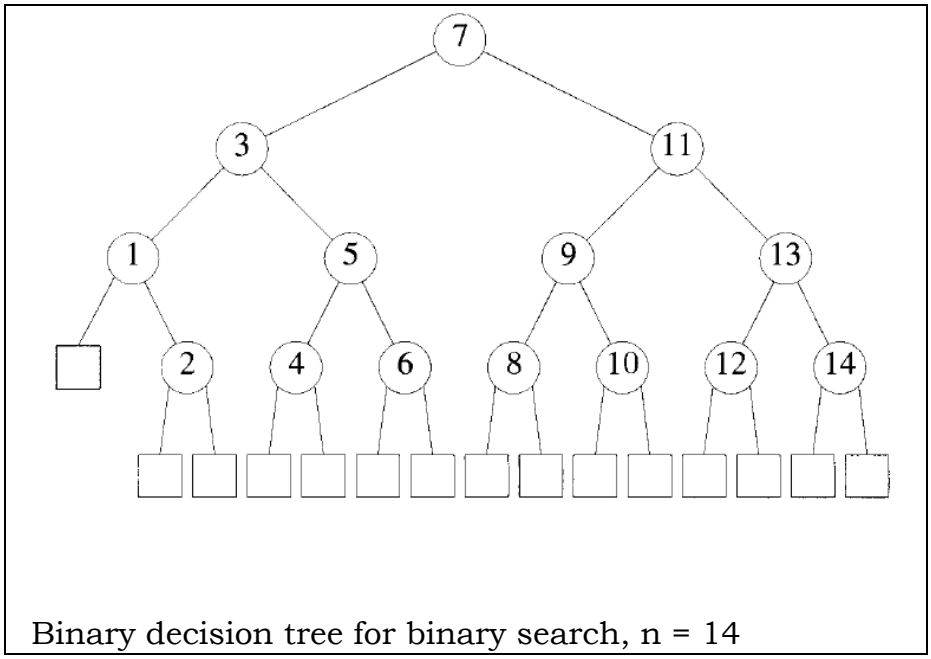
Place them in a[l:14],and simulate the steps that BinSearch goes through as it searches for different values of x.

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

Binary decision tree for binary search:



Binary decision tree for binary search, n = 14

- The first comparison is x with a[7].If x a[7],then the next comparison is with a[3].
- Each path through the tree represents a sequence of comparisons in the binary search method.
- If x is present, then the algorithm will end at one of the circular nodes that lists the index into the array where x was found.
- If x is not present, the algorithm will terminate at one of the square nodes. Circular nodes are called internal nodes, and square nodes are referred to as external nodes.

Succesful and Unsuccessful search:

- If n is in the range $(2^{k-1}, 2^k)$ then BinSearch makes at most k element comparisons for a successfulsearch and either k-1 or k comparisons for an unsuccessful search .
- In otherwords the time for a successful search is O(log n) and for an unsuccessful search is O(log n).
- Consider the binary decision tree describing the action of BinSearch on n elements. All successful lsearches end at a circula rnode where as all unsuccessful searches end at a square node.
- If $2^{k-1} <= n < 2^k$, then all circular nodes are at levels1,2,..., k where as all square nodes are at levels k and k + 1 (note that the root is at level 1). The number of element comparisons needed to terminate at a circular node on level i is i whereas the number of element comparisons needed to terminate at a square node at level i is only i- 1.
- To determine the average behavior, we need to look more closely at the binary decision tree and equate its size to the number of element comparisons in the algorithm. The distance of a node from the root is one less than its level. The **internal path length I** is the sum of the distances of all internal nodes from the root.

- The **external path length E** is the sum of the distances of all external nodes from the root. It is easy to show by induction that for any binary tree with n internal nodes, E and I are related by the formula:

  E = I +2n
- There is a simple relationship between E, I, and the average number of comparisons in binary search. Let **$A_s(n)$ be the average number of comparisons in a successful search, and $A_u(n)$ the average number of comparisons in an unsuccessful search**.
- The number of comparisons needed to find an element represented by an internal node is one more than the distance of this node from the root. Hence,

  $A_s(n) = 1+I/n$
- The number of comparisons on any path from the root to an external node is equal to the distance between the root and the external node. Since every binary tree with n internal nodes has n +1 external nodes, it follows that:

  $A_u(n) = E/(n+1)$
- Using these three formulas for E,$A_s(n)$, and $A_u(n)$, we find that:

  $A_s(n) = (1+1/n)A_u(n)-1$
- From this formula we see that $A_s(n)$ and $A_u(n)$ are directly related.
- The minimum value of $A_s(n)$ (and hence Au(n)) is achieved by an algorithm whose binary decision tree has minimum external and internal path length.
- This minimum is achieved by the binary tree all of whose external nodes are on adjacent levels, and this is precisely the tree that is produced by binary search.
- E is proportional to **nlog n**.Using this in the preceding formulas, we conclude that $A_s(n)$ and $A_u(n)$ are both proportional to **log n**.
- Thus we conclude that the average-and worst-case numbers of comparisons for binary search are the same to within a constant factor.

- The best-case analysis is easy. For a successful search only one element comparison is needed. For an unsuccessfulsearch, [logn] element comparisons are needed in the best case.
- The computing time of binary search by giving formulas that describe the best, average, and worst cases:

successful searches			unsuccessful searches
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

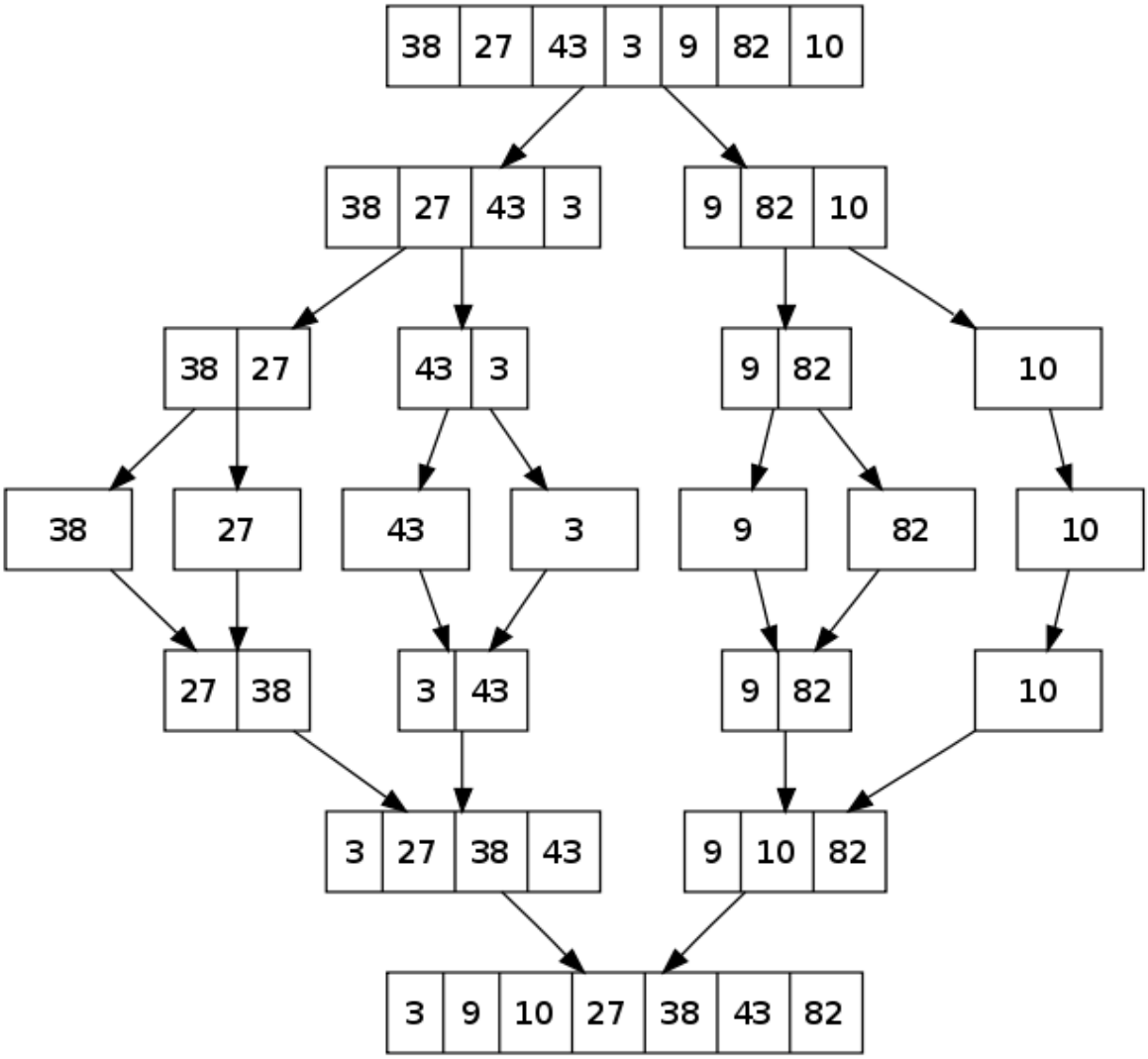
Asymptotic analysis :

- **Best Case :**
  - The element is exactly in the middle of the list.
  - Time complexity is O(1).
- **Worst Case :**
  - The element is found at last attempt.
  - Time complexity is O(log n).
- **Average Case :**
  - Time complexity is O(log n).

MERGE SORT:

- **Merge Sort:**Merge sort is a sorting technique based on divide and conquer technique.
- A merge sort works as follows:
  - Divide the unsorted list into n sub lists, each containing 1 element.
  - Repeatedly merge sub lists to produce new sorted sub lists until there is only 1 sub list remaining. This will be the sorted list.
- Consider the following example which sorts a list of seven elements using merge sort.

The unsorted list is : 38, 27,43,3,9,82,10



Algorithm Mergesort(low, high)

// a[low: high] is a global array to be sorted.

{

if(low<high) then

{

//Divide Problem into sub problems.

//Find where to split the list.

mid:= (low + high)/2;

```
        //Solve the sub problems
                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 sub lists in a [low : mid] and
// in a[mid+1,high].

//b[] is an auxiliary global array.
{
        i := low;
        j := high;
 while(( i<= mid) and (j<=high) do
{
    if(a[i] <= a[j]) then
    {
        b[k] := a[i];
        i=i+1;
    }
    else
    {
        b[k] := a[j];
        j=j+1;
      }
      k = k+1;
    }
```

```
        if ( i> mid) then
        {
                // copy all the remaining elements of the sub list to auxiliary
    array b[].
                for  p=j to high do
                {
                        b[k] := a[p];
                        k := k+1;
                }
                else
                {
                        for p:= i to mid do
                        {
                                b[k] := a[p];
                                k := k+1;
                        }
                }
              // Copy all the elements form auxiliary array b[] to  original
array a[].
     for i:= low to high do
        {
                a[i] := b[j];
        }

    }
```

- If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation.

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

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

= 2(2 T(n/4) +cn) + cn

= 4 T(n/4) + 2cn

= 4(2 T(n/8) + cn) + 2cn

= 8 T(n/8) +3cn

.

.

.

= 2^k T(1) +kcn

= an + cn log n

Therefore

T(n) = O(nlogn)

- **Advantages of Merge Sort**:
  - One of the fastest algorithms on average.
  - Does not need additional memory (the sorting takes place in the array - this is called **in-place** processing).
  - Compare with merge sort: Merge sort needs additional memory for merging.

- **Disadvantages of Merge Sort:** The worst-case complexity is $O(n^2)$

QUICKSORT

- Quick sort is also called as called partition-exchange sort.
- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- Quick sort follows divide and conquer approach.
- In Quick sort, the division into two sub arrays is made so that the sorted sub arrays do not need to be merged later.

The steps are:

1. **Pick** an element, called a pivot, from the array.
2. **partition operation:**
   Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition operation.**
3. **Recursively apply the above steps** to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Algorithm QuickSort(p, q)

// Sorts the elements a[p],..a[q,] which reside in the global array a[l:n] into ascending order.
// a[n+ 1]is considered to be defined and must be ≥ all the elements in a[l:n].
{
    if (p <q) then //If there are more than one element
    {
        // divide P into two sub problems.

j :=Partition(a, p, q+ 1); 10 // j is the position of the partitioning element.

// Solve the sub problems.

QuickSort(p,j-1);

QuickSort(j+ 1, q);

// There is no need for combining solutions.

}

} // End of Algorithm

```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., 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] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
     v := a[m]; i := m; j := p;
     repeat
     {
          repeat
               i := i + 1;
          until (a[i] ≥ v);

          repeat
               j := j - 1;
          until (a[j] ≤ v);

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

     } until (i ≥ j);

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

Analysis of Quicksort :

- In analyzing Quick Sort,we count only the number of element comparisons C(n).It is easy to see that the frequency count of other operations is of the same order as C(n).
- We make the following assumptions: the n elements to be sorted are distinct, and the input distribution is such that the partition element **v = a[m]** in the call to Partition (a,m,p) has an equal probability of being the ith smallest element,1<=i<=p-m, a[m:p-1]
- **Worst case analysis:** Let the worst-case value be $C_w(n)$ of C(n).The number of element comparisons in each call of Partition is at most p- m + 1. Let r be the total number of elements in all the calls to Partition at any level of recursion.
- At level one only one call,Partition (a,l,n+l), is made and r = n; at level two at most two calls are made and r = n-1;and so on.
- At each level of recursion, O(r) element comparisons are made by Partition. At each level, r is at least one less than the r at the previous level as the partitioning elements of the previous level are eliminated. Hence $C_w(n)$ is the sum on r as r varies from 2 to n, or $O(n^2$
- **Average case analysis:** The average value $C_A(n)$ of C(n) is much less than $C_w(n)$. Under the assumptions made earlier, the partitioning element v has an equal probability of being the ith-smallest element 1<=i<=p-m, a[m:p-1]
- Hence the two sub arrays remaining to be sorted are **a[m:j] and a[j+1:p-1]**with probability 1/(p- m), m<=j<p. From this we obtain the recurrence:

$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \le k \le n} [C_A(k-1)) + C_A(n-k)]$$

- The number of element comparisons required by Partition on its first call is n + 1.Note that $C_A(0)= C_A(1)= 0$.Multiplying both sides of above equation by n, we obtain:

$$nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n-1)]$$

- Replacing n by n-1, we get:

$$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + \cdots + C_A(n-2)]$$

- Subtracting the above equations, we get:

$$\begin{aligned} nC_A(n) - (n-1)C_A(n-1) &= 2n + 2C_A(n-1) \\ & or \\ C_A(n)/(n+1) &= C_A(n-1)/n + 2/(n+1) \end{aligned}$$

- Repeatedly using this equation to substitute for $C_A(n-1), C_A(n-2), \ldots$, we get:

$$\begin{aligned} \frac{C_A(n)}{n+1} &= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \\ &= \frac{C_A(1)}{2} + 2\sum_{3 \le k \le n+1} \frac{1}{k} \\ &= 2\sum_{3 \le k \le n+1} \frac{1}{k} \end{aligned}$$

since

$$\sum_{3 \le k \le n+1} \frac{1}{k} \le \int_2^{n+1} \frac{1}{x}\, dx = \log_e(n+1) - \log_e 2$$

- The above equation becomes:

$$C_A(n) \le 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)$$

- The average time is only **O(n log n).**
- **Worst-case: $O(n^2)$**
  - This happens when the pivot is the smallest (or the largest) element.
  - Then one of the partitions is empty, and we repeat recursively the procedure for N-1 elements.
- **Best-case : O(nlogn)**
  - The best case is when the pivot is the median of the array, and then the left and the right part will have same size.
  - There are **log n** partitions, and to obtain each partition we do n comparisons (And not more than n/2 swaps). Hence the complexity is **O(nlog n)**
- **Average-case: O(n log n)**

Analysis

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

The time to sort the file is equal to

The time to sort the left partition with i elements, plus

The time to sort the right partition with N-i-1 elements, plus

The time to build the partitions

- **Worst case analysis**

In Worst case, the pivot is the smallest element.

T(n) = T(n-1) + c n , n > 1

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

$T(n-2) = T(n-3) + c(n-2)$

$T(n-3) = T(n-4) + c(n-3)$

.

.

.

$T(2) = T(1) + c.2$

Add all equations:

$T(n) + T(n-1) + T(n-2) + \ldots + T(2) =$

$= T(n-1) + T(n-2) + \ldots + T(2) + T(1) + c(n) + c(n-1) + c(n-2) + \ldots + c.2$

$T(n) = T(1) + c(2 + 3 + \ldots + n)$

$T(n) = 1 + c(n\ (n+1)/2 - 1)$

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

- **Best-case analysis:**

  Best-case is when the pivot is at the middle. So, pivot element divides the list in to two equal parts.

  $T(n) = 2T(n/2) + c\ n$

Divide by n:

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

$T(n/2) / (n/2) = T(n/4) / (n/4) + c$

$T(n/4) / (n/4) = T(n/8) / (n/8) + c$

……

$T(2) / 2 = T(1) / (1) + c$

Add all equations:

$T(n)/N + T(n/2) / (n/2) + T(n/4) / (n/4) + …. + T(2) / 2 =$

$= (n/2) / (n/2) + T(n/4) / (n/4) + … + T(1) / (1) + c.\log n$

After crossing the equal terms:

$T(n)/n = T(1) + c \text{ Log } n = 1 + c \text{ Log } n$

$T(n) = n + n \text{ c Log } n$

Therefore T(n) = O(n log n)

UNIT – II

Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. Which of the following algorithms is NOT a divide & conquer algorithm by nature? [ ]
   A) Euclidean algorithm to compute the greatest common divisor
   B) Heap sort C) Merge sort D) Quick sort
2. Time required to merge two sorted lists of size m and n is [ ]
   A) O(m/n) B) O(m+n) C) O(mn) D) O(m-n)
3. The best-case time complexity of binary search is [ ]
   A) O(1) B) O(log n) C) $O(n^2)$ D) O(n)
4. The average-case time complexity of binary search is [ ]
   A) O(1) B) O(log n) C) $O(n^2)$ D) O(n)
5. The worst-case time complexity of binary search is [ ]
   A) O(1) B) O(log n) C) $O(n^2)$ D) O(n)
6. The worst-case time complexity of merge sort is [ ]
   A) □(1) B) □(log n) C) □(n^2) D) □(nlog n)
7. The average-case time complexity of merge sort is [ ]
   A) □(n) B) □(log n) C) □(n^2) D) □(nlog n)
8. Which of the following is not a limitation of binary search algorithm? [ ]
   A) Must use a sorted array
   B) Requirement of sorted array is expensive when a lot of insertion and deletions are needed
   C) There must be a mechanism to access middle element directly
   D) Binary search algorithm is not efficient when the data elements are more than 1500.

9. Quick sort exhibits worst-case time complexity when the data is already in sorting order [True/False]

10. The average-case time complexity of quick sort is [ ]

A) □(n) B) □(log n) C) □(n^2) D) □(nlog n)

11. The worst-case time complexity of quick sort is [ ]

A) □(n) B) □(log n) C) □(n^2) D) □(nlog n)

12. The time complexity for calculating the articulation points of a graph G with 'n' vertices and 'e' edges once the L values are determined is [ ]

A) O(n*e) B) O(n+e) C) O(n-e) D) O(n/e)

13. If a tree has 'm' nodes which is created as a sequence of unions performed by weighted union, then the height of the tree is not greater than [ ]

A) Log 2 m+1 B) Log 2 m-1 C) Log 2 m D) Log 2 m+c

14. LetP be a quicksort program to sort numbers in ascending order. Let t1 and t2 be the time taken by the program for the inputs [1 2 3 4] and [5 4 3 2 1] respectively. Which of the following holds [ ]

A) t1 = t2 B) t1 > t2 C) t1 < t2 D) t1 = t2+5log5

15. The solution to the recurrence T(n) = T(n/2)+n is [ ]

A) O(log n) B) O(n log n) C) O(n) D) O(n^2)

16. The recurrence relation that arises in relation with the complexity of binary search is [ ]

A) T(n) = T(n/2) + k, k is a constant

B) T(n) = 2T(n/2) + k, k is a constant

C) T(n) = T(n/2) + logn D) T(n) = T(n/2) + n

17. A sorting technique is called stable if [ ]

A) it takes O(n logn) time

B) it maintains the relative order of occurrence of non-distinct elements

C) it uses divide and conquer paradigm

D) it takes O(n) space

18. The recurrence relation [ ]

T(1) = 2

T(n) = 3T(n/4)+n

Has the solution T(n) equal to

A) O(n) B) O(logn) C) $O(n^{3/4})$ D) None of these

19. In the following C function, let n>=m

```
intgcd(n,m)
{       if(n%m ==0) return m;
        n=n%m;
    return gcd(m,n);
 }
```

How many recursive calls are made by this function? [ ]

A) □(logn) B) Ω(n) C) □(log logn) D) □($\sqrt{n}$)

20. The worst-case running times of Insertion sort, Merge sort and Quick sort, respectively, are [ ]

A) □(n logn), □(n logn), and □(n^2) B) □(n^2), □(n^2), and □(n logn)

C) □(n^2), □(n logn), and □(n logn) D) □(n^2), □(n logn), and □(n^2)

SECTION-B

SUBJECTIVE QUESTIONS

1. Define disjoint set. Describe tree and array representations of disjoint sets.
2. Design algorithms for simple union and find operations. Also discuss problems associated with these algorithms.
3. Develop algorithms for union using weighting rule and find using collapsing rule.
4. What is an articulation point? Write an algorithm to find articulation points.
5. Write and explain the control abstraction for divide and conquer.
6. Write an algorithm for quick sort and analyze its worst-case time complexity.
7. Write an algorithm for merge sort and analyze its time complexity.
8. Write recursive binary search algorithm and analyze its time complexity.

9. Show how binary search algorithm works for searching 151, -14 and 9 in the following set of elements:

   -15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

10. Draw the tree of calls of merge sort and merge for the following set.

    (35, 25,15,10,45, 75, 85, 65, 55, 5, 20, 18)

11. Derive the average case time complexity of a Quick sort.

12. Sort the records with the following index values in ascending order using quick sort algorithm.

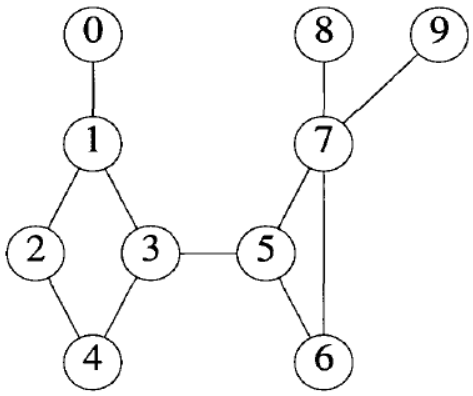
    65,70,75,80, 33, 60,55, 22, 50,45, 11

13. A sorting method is said to be stable if at the end of the method, identical elements occur in the same order as in the original unsorted ser. Is merge sort a stable sorting method? Show this with a suitable example.

14. Sort the following data in ascending order using merge sort

    35, 25,15,10,45, 75, 85, 65, 55, 5, 20, 18

15. Identify the articulation points and draw the biconnected components for the following graph:



SECTION-C

QUESTIONS AT THE LEVEL OF GATE

1. Suppose T(n)=2T(n/2)+n, T(0)=T(1) which one of the following is false? [GATE 2005] [ ]

   A) T(n) =O(n^2) B) T(n) = □(n log n)

   C) T(n)=□(n) D) T(n) = O(n log n)

2. In quick sort, for sorting n elements, the $(n/4)^{th}$ smallest element is selected as pivot using an O(n) time algorithm. What is the worst case time complexity of the quick sort?[GATE 2009] [ ]

   A) □(n) B) □(n log n) C)□(n^2) D) □(n^2 log n)

3. Which of the following correctly determines the solution of the recurrence relation with T(1) = 1?[GATE 2014] [ ]

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

   A)□(n) B) □(n log n) C) □(n^2) D) □(log n)

4. The recurrence relation [GATE 2004]

   T(1) = 1

   T(n) = 2T(n-1) + n, n≥2

   Evaluates to [ ]

   A) $2^{n+1} - n - 2$ B) $2^n - n$ C) $2^{n+1} - 2n - 2$ D) $2^n + n$

5. The running time of an algorithm is represented by the following recurrence relation;

   $$T(n) = \begin{cases} n, & n \leq 3 \\ T\left(\frac{n}{3}\right) + cn, & otherwise \end{cases}$$

   Which of the following represents the time complexity of the algorithm? [GATE 2009] [ ]

   A) □(n) B) □(n log n) C) □(n^2) D) □(n^2 log n)