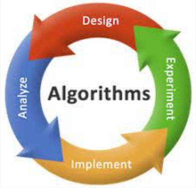
DESIGN AND ANALYSIS OF ALGORITHMS (DAA) (A34EC)

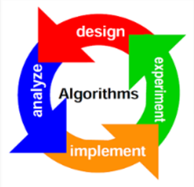
By :-

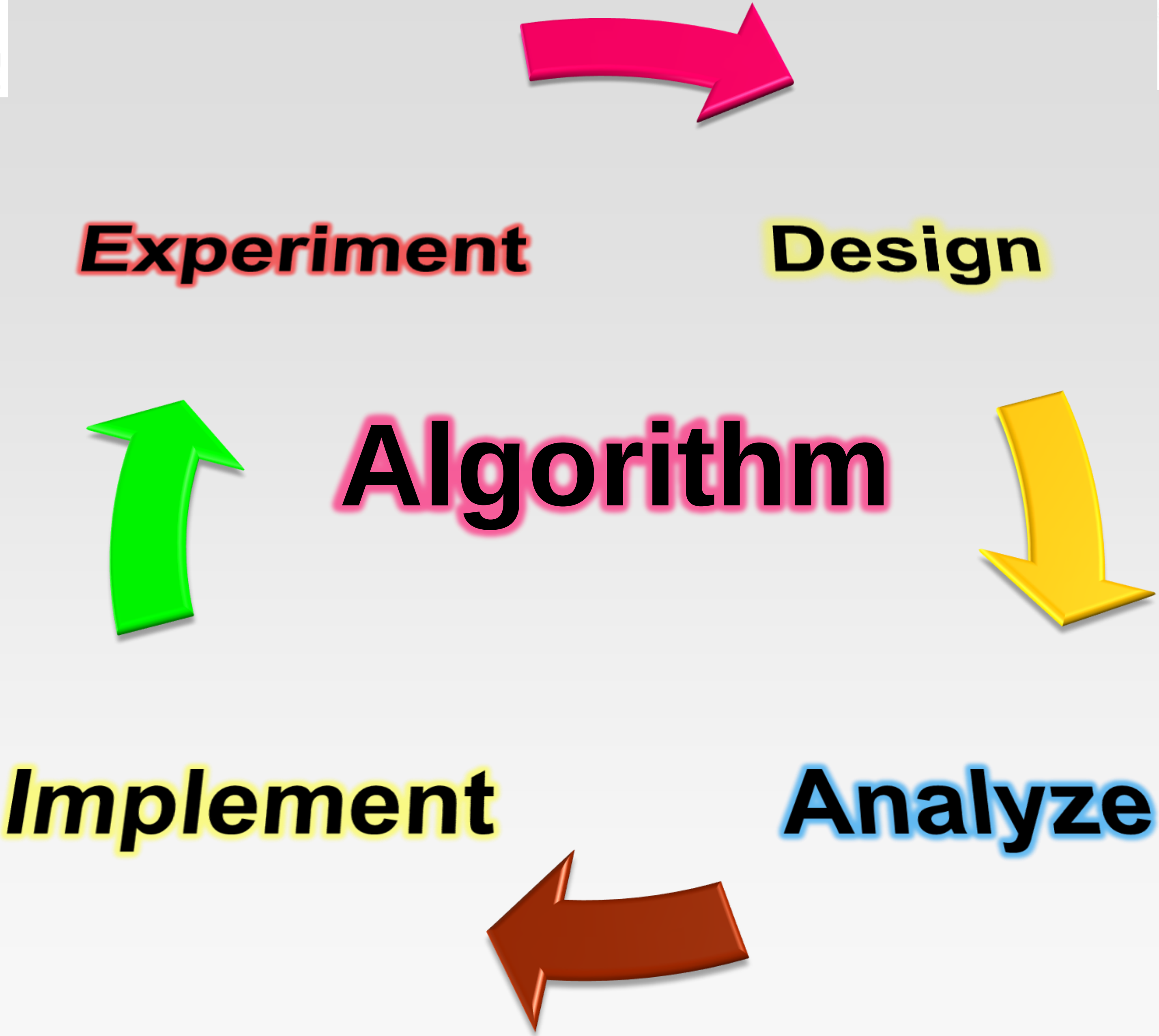
VIJAYKUMAR MANTRI,

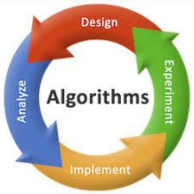
ASSOCIATE PROFESSOR.

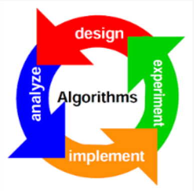
vijay_mantri.it@bvrit.ac.in



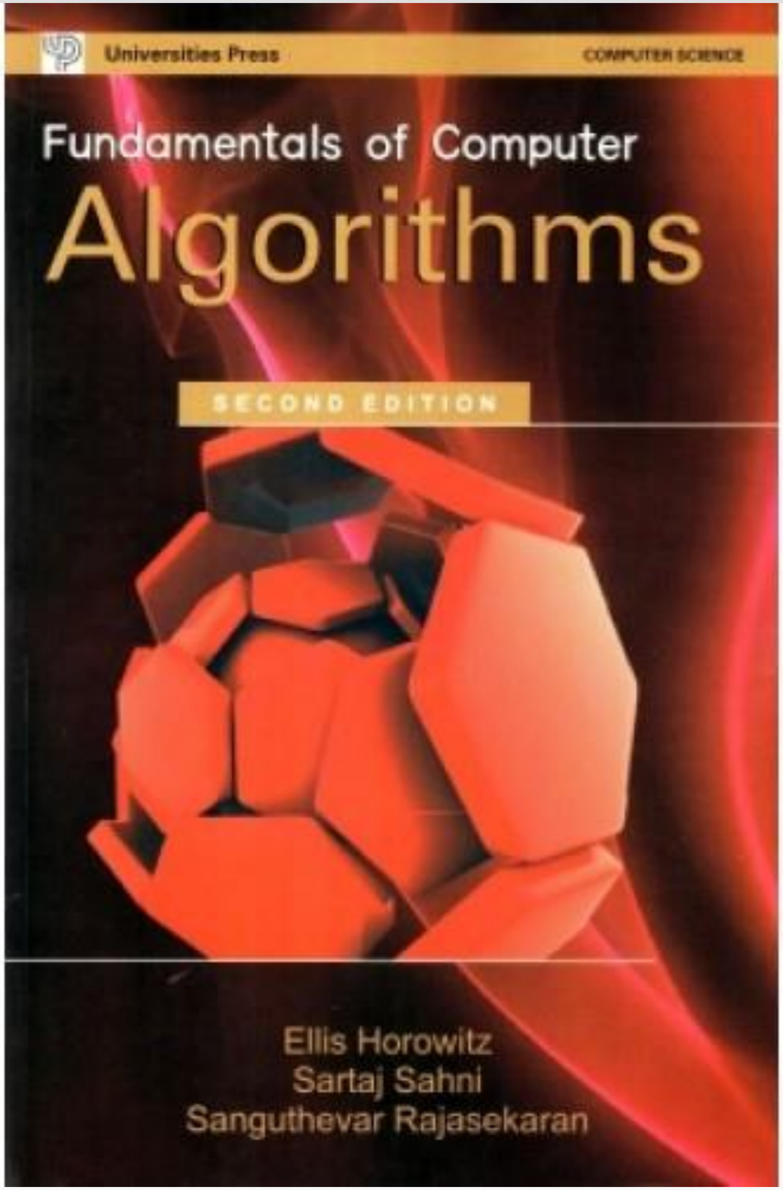



Experiment
Design
Algorithm
Implement
Analyze


Design
Experiment
Algorithms
Analyze
Implement

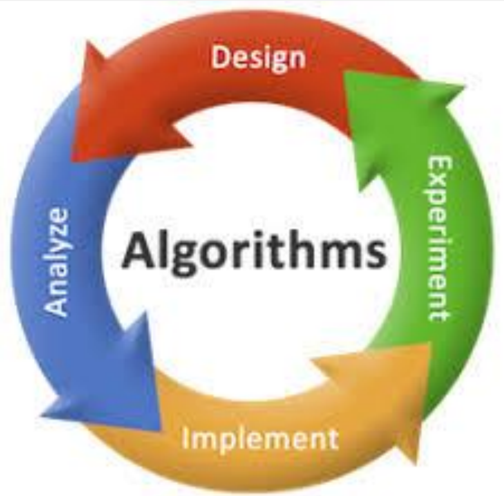

design
experiment
Algorithms
analyze
implement

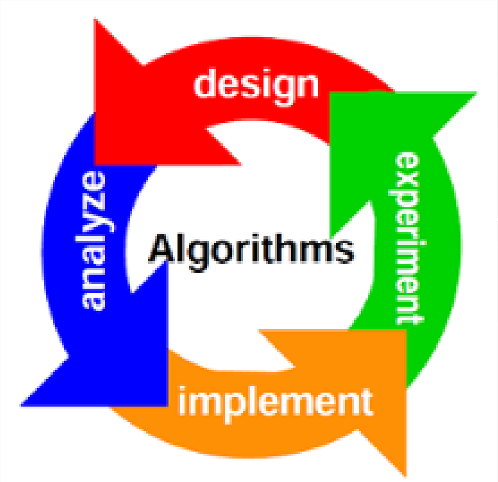
Textbook


Universities Press
COMPUTER SCIENCE
Fundamentals of Computer
Algorithms
SECOND EDITION
Ellis Horowitz
Sartaj Sahni
Sanguthevar Rajasekaran

DAA Unit II

Disjoint Sets Divide and Conquer





Unit II Syllabus

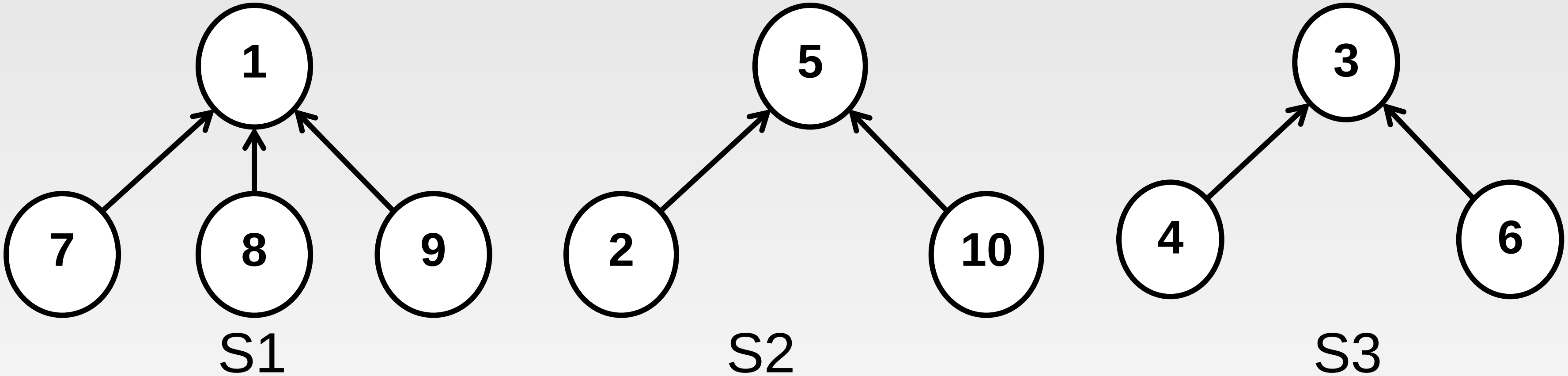
- Disjoint Sets:
  - Disjoint Sets,
  - Disjoint Set Operations,
  - Union and Find Algorithms,
  - Connected Components
  - and Bi-Connected Components.
- Divide and Conquer:
  - General method,
  - Applications Binary Search,
  - Merge Sort,
  - Strassen’s Matrix Multiplication.

Set and Disjoint Set

- A set is a collection data structure that stores certain values in a way that values are not repeated.
- Depending on whether these values are stored in an order or not, set is called ordered set or unordered set.
- It is an implementation of mathematical concept of Finite set.
- In this section we are going to see use of forests in the representation of sets.
- We assume that the elements of the sets are the numbers like 1, 2, 3, …., n.
- Also we assume that the sets being represented are pairwise **Disjoint** – that is if $\mathbf{S_i}$ **&** $\mathbf{S_j}$, $i \neq j$, are two sets then there is no element that is in both $\mathbf{S_i}$ **&** $\mathbf{S_j}$.

Set and Disjoint Set

- A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.
- For Example consider n=10, the elements can be partitioned into three different sets S1, S2, S3 like…



Possible tree representation of Sets

- Note that for each set we have linked the nodes from children to parent.

Set and Disjoint Set

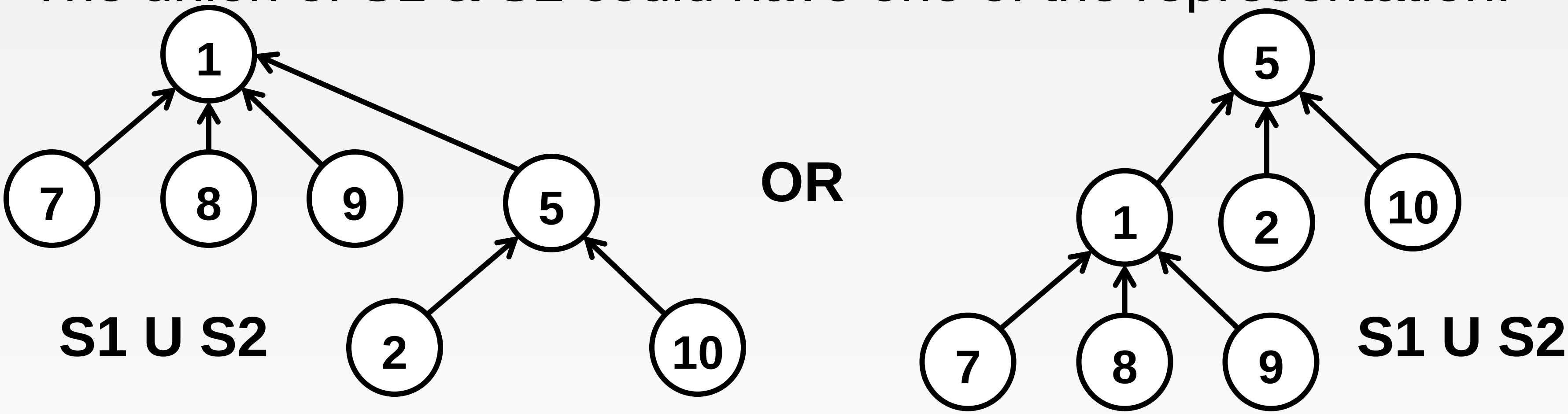
- The operations we wish to perform on these sets are :

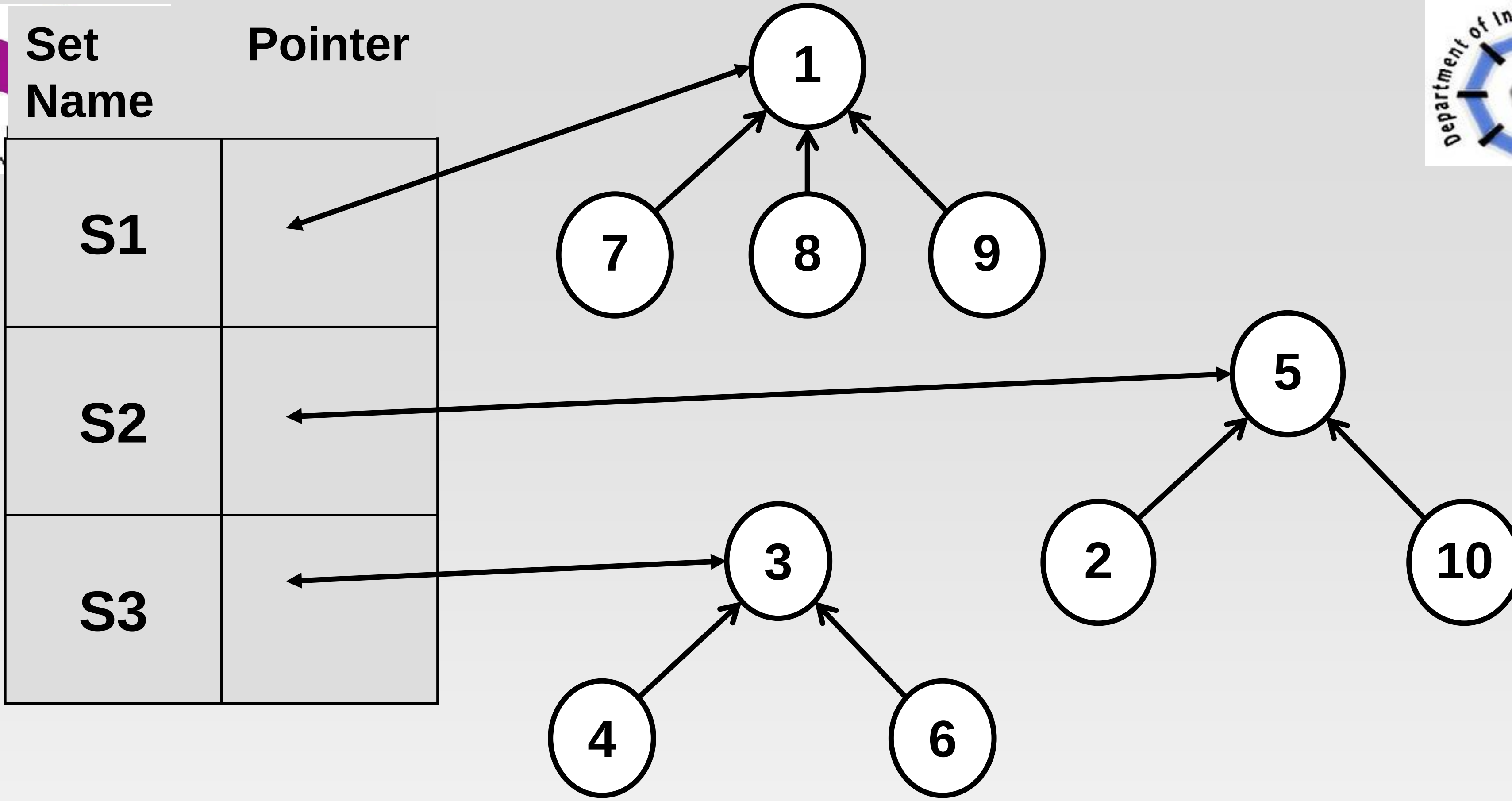
1. **Disjoint Set Union :** Join two subsets into a single subset. If $\mathbf{S_i}$ **&** $\mathbf{S_j}$ are two disjoint sets, then their union

   $\mathbf{S_i}$ **U** $\mathbf{S_j}$ = all elements x such that x is in $\mathbf{S_i}$ or $\mathbf{S_j}$.

2. **Find(i) :** Given the element i, find the set containing i. Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

- **Union and Find O**perations : Consider Union operation first The union of S1 & S2 could have one of the representation.





Data representation for S1, S2 & S3

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

Array representation for S1, S2 & S3

Union and Find Algorithms

- Here are the algorithm for Union & Find operations.

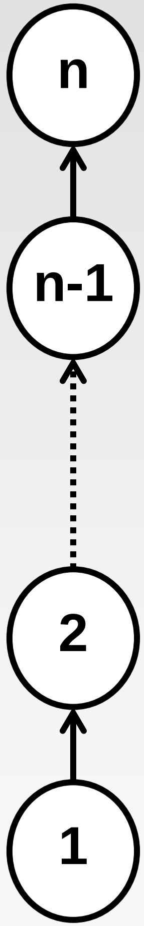
1. **Algorithm SimpleUnion (i, j)**
2. **{**
3. **p [i] := j;**
4. **}**

1. **Algorithm SimpleFind (i)**
2. **{**
3. **while (p [i] >=0) do**
4. **i := p [i];**
5. **return i;**
6. **}**

- Even these algorithms are very easy to state, their performance are not very good.

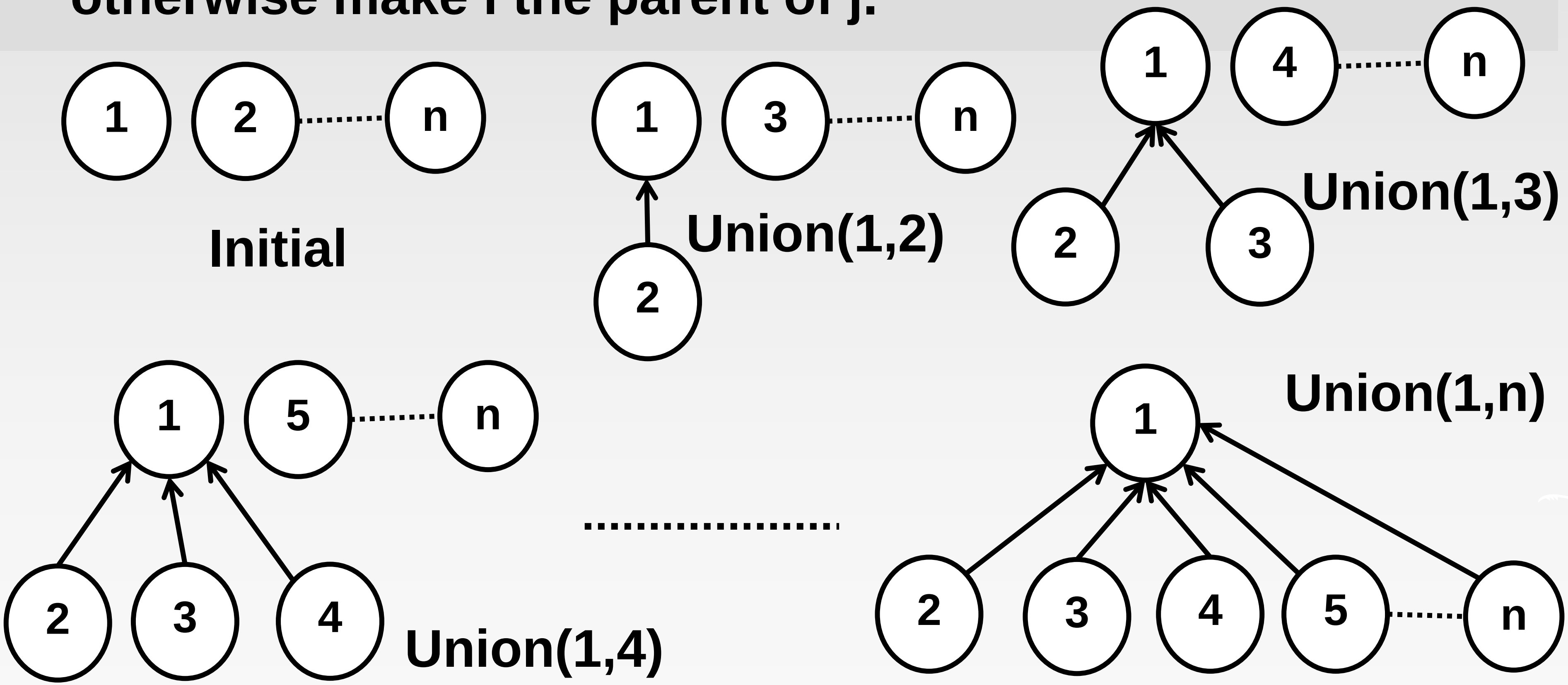
Union and Find Algorithms

- For example if we start with **q** elements each in set of its own i.e. $\mathbf{S_i = \{ i \}, 1 \le i \le q}$ then initial configuration consists of a forest with q nodes and $\mathbf{p[\,i\,] = 0, 1 \le i \le q}$
- Now let us process following sequence of union-find operations
  - **Union(1,2), Union(2,3), Union(3,4), .., Union(n-1, n)**
  - **Find(1), Find(2), Find(3), Find(4), ….., Find(n)**
- This will results in the degenerate tree.
- Since time taken for union is constant, the **n-1** union can be processed in time **O(n)**.
- 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 n finds is $\mathbf{O(\sum_{i=1}^{n} i) = O(n^2)}$



Union and Find Algorithms

- We can improve performance of Union & Find algorithms using weighting rule for **Union(i, j)**.
- 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.**



Union and Find Algorithms

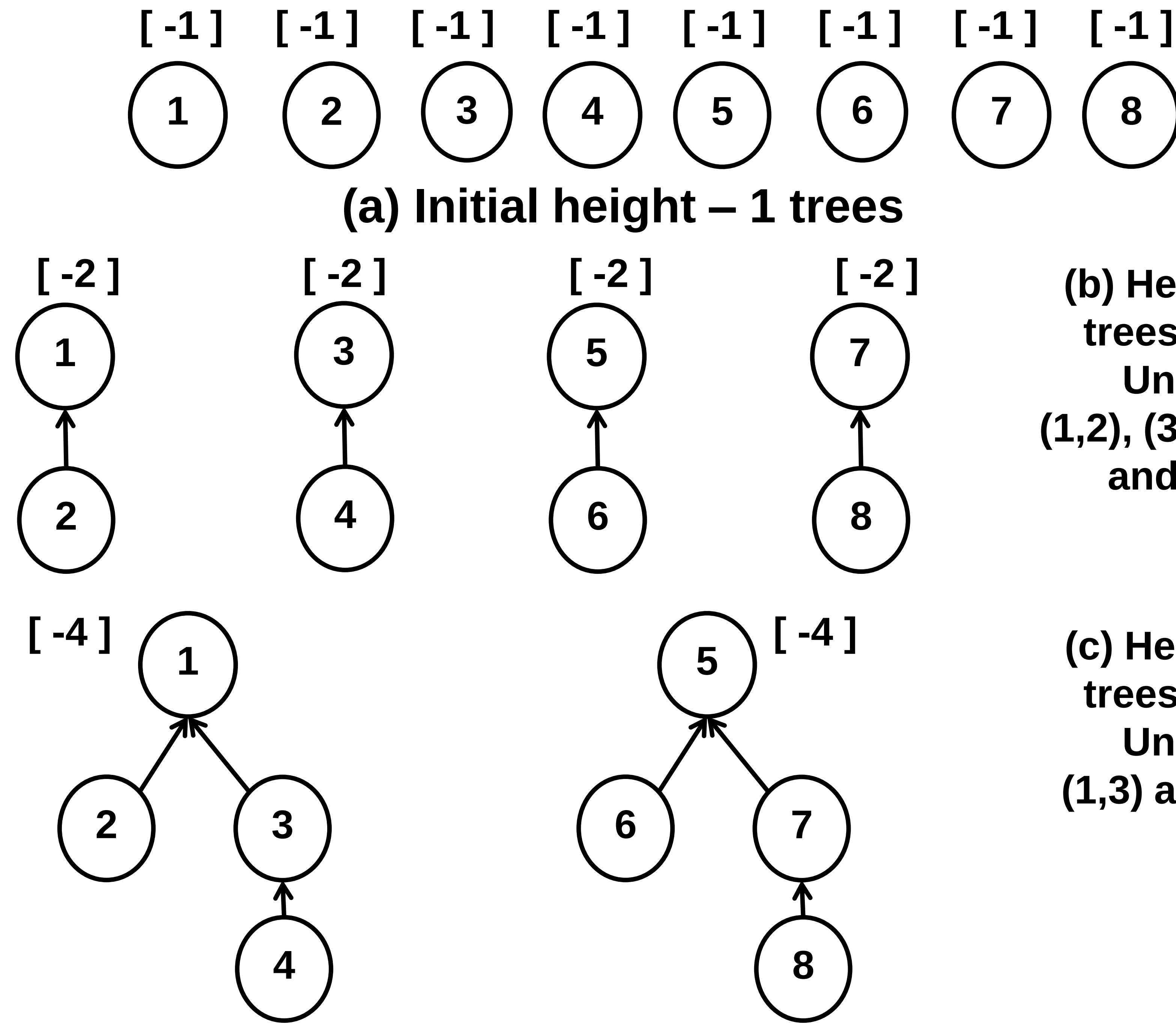
```
Algorithm WeightedUnion (i, j)
// p [ i ] = - count[ i ] and p [ j ] = - count[ j ]
{
  temp := p [ i ] + p [ j ];
   if(p [ i ] > p [ j ]) then
   { // i  has fewer nodes
      p [ i ] := j; p [ j ] := temp;
   }
   else
    { // j  has fewer or equal nodes
      p [ j ] := i; p [ i ] := temp;
   }
 }
```

Union and Find Algorithms

- The Find algorithm remains unchanged.
- The maximum time to perform a find will be calculated as
- Assume that we start with a forest of trees, each having one node. Let T be a tree with m nodes created using WeightedUnion. The height of T is no greater than $\mathbf{\log_2 m + 1}$
- Now consider creation of tree using WeightedUnion algorithm for **8** elements with initial configuration

$$p[\, i \,] = -\, count[\, i \,] = -1,\ 1 \le i \le 8 = n$$

  - **Union(1,2), Union(3,4), Union(5,6), Union(7,8)**
  - **Union(1,3), Union(5,7), and finally Union(1,5)**
- This will results in tree as in next slides.
- We found that height of each tree with m nodes is $\mathbf{\log_2 m + 1.}$
- The time to process a find is **O(log m)** if there are m elements
- If an intermixed sequence of **u-1** union and **f** find operations, the time becomes **O(u + f log u)** along with **O(n)** to initialize **n-tree** forest.

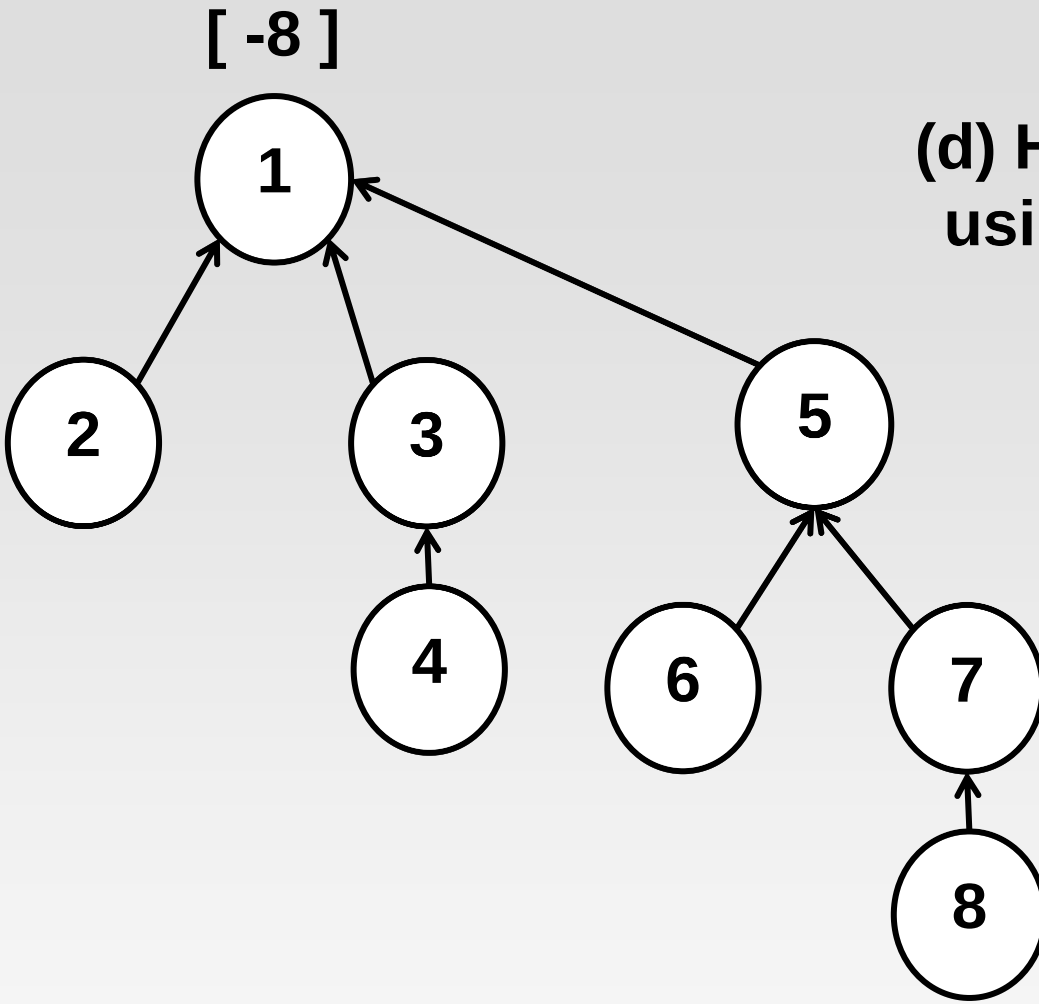


(a) Initial height – 1 trees

(b) Height – 2 trees using Unions (1,2), (3,4), (5,6), and (7,8)

(c) Height – 3 trees using Unions (1,3) and (5,7)

(d) Height – 4 trees using Union (1,5)



Trees achieving worst-case bound

Union and Find Algorithms

- Still further improvement is possible in the Find algorithm using Collapsing Rule.
- **Collapsing Rule :** If j is node on the path from i to its root and **p[i] ≠ root[i]**, then set **p[j] to root [i].**

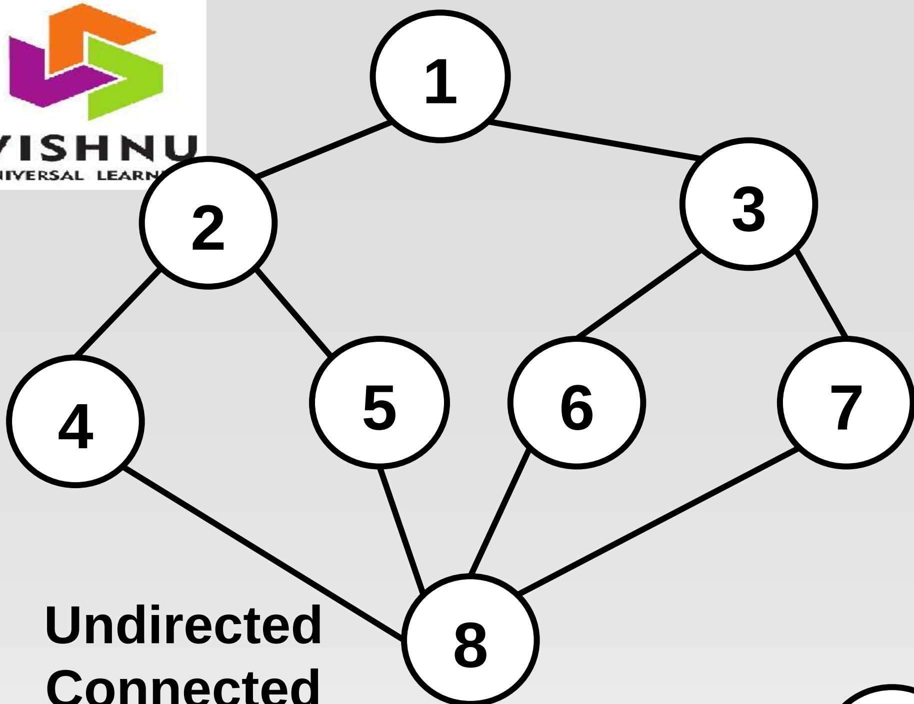
```
Algorithm CollapsingFind (i)
{
    r := i;
    while (p [ r ] > 0) do
        r := p [ r ];
    while ( i ≠ r ) do
    {
        s := p [ i ];
        p [ i ] := r;
        i := s;
    }
return r
}
```

Union and Find Algorithms

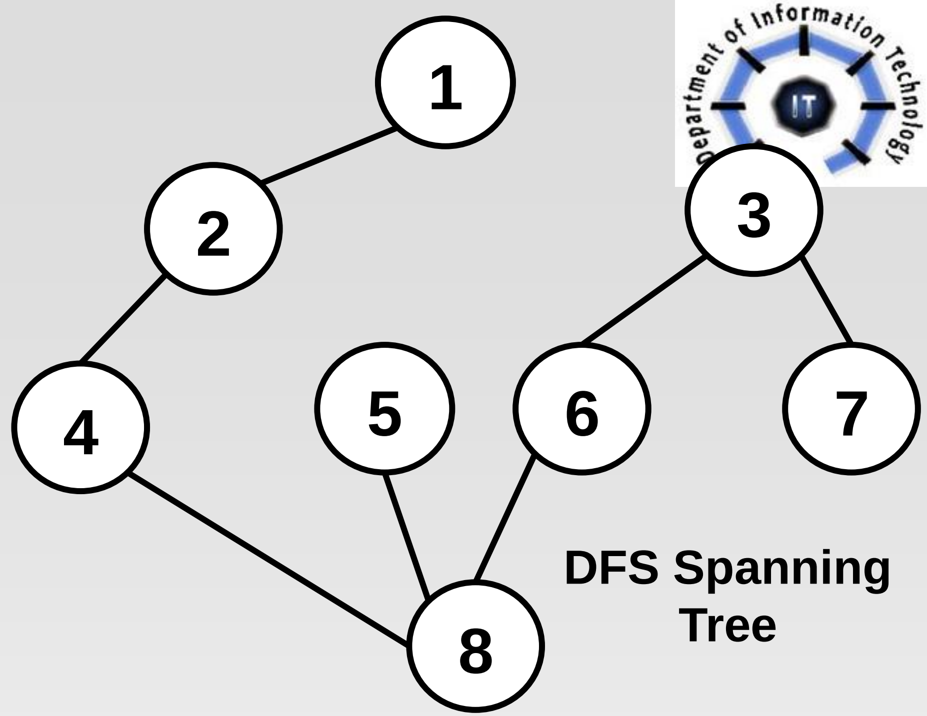
- Consider the tree created by WeightedUnion as seen in previous example. Now we will process the following ten finds to search element 8:
  - **Find(8), Find(8), Find(8), Find(8), ….., Find(8)**
- If we use **SimpleFind**, each Find(8) requires going up three parent link fields for a total of 30 moves to process ten finds.
- When we use **CollapsingFind** the first Find(8) requires going up **three** links and then resetting **two** links (Actually three links as it will reset parent of 5 to 1).
- Each remaining nine Find requires going up by only one link field.
- The total cost is now only **15** moves.

Connected Components

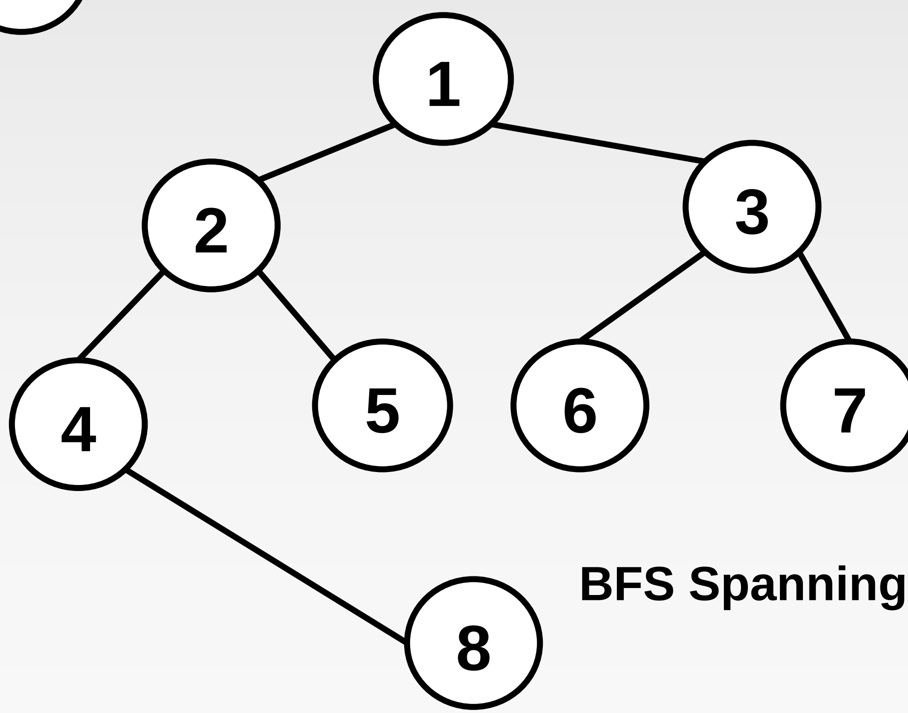
- **Graph** : A graph **G** consists of two sets **V** and **E** where **V** is set of vertices and **E** is a set of pairs of edges. **G=(V, E).**
- In an undirected graph **G**, two vertices **u** & **v** are said to be connected iff there is a path in **G** from **u** to **v**.
- An undirected graph is said to be connected iff for every pair of distinct vertices **u** & **v** in **V(G)**, there is a path from **u** to **v** in **G**.
- A connected component of an undirected graph is a maximal connected subgraph. Maximal meaning that **G** contains no other subgraph.
- If graph **G** is connected undirected graph, then all vertices of **G** will get visited on the first call to Breadth First Search (BFS). If not then connected, then at least two calls to BFS.



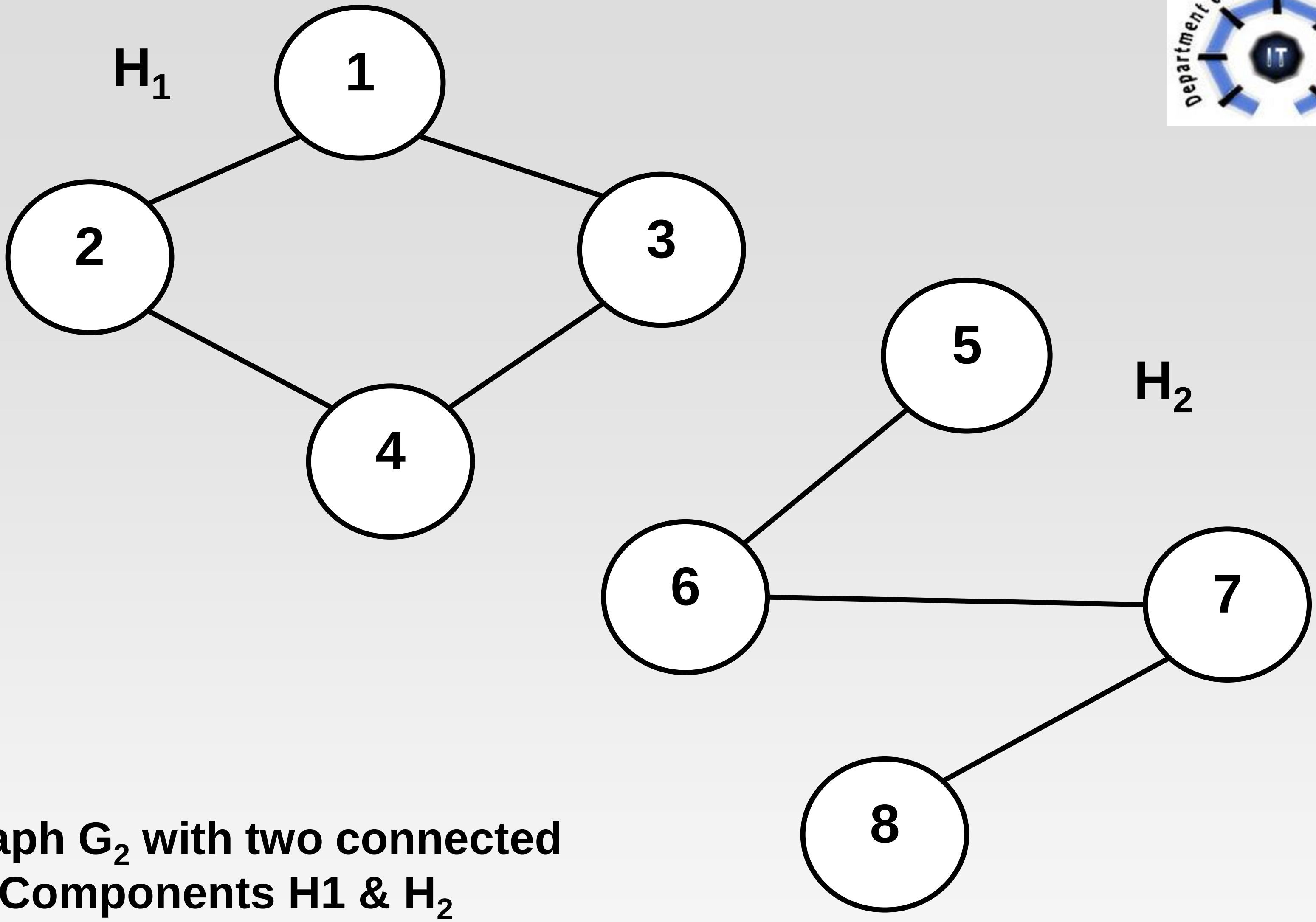
Undirected Connected Graph G_1



DFS Spanning Tree



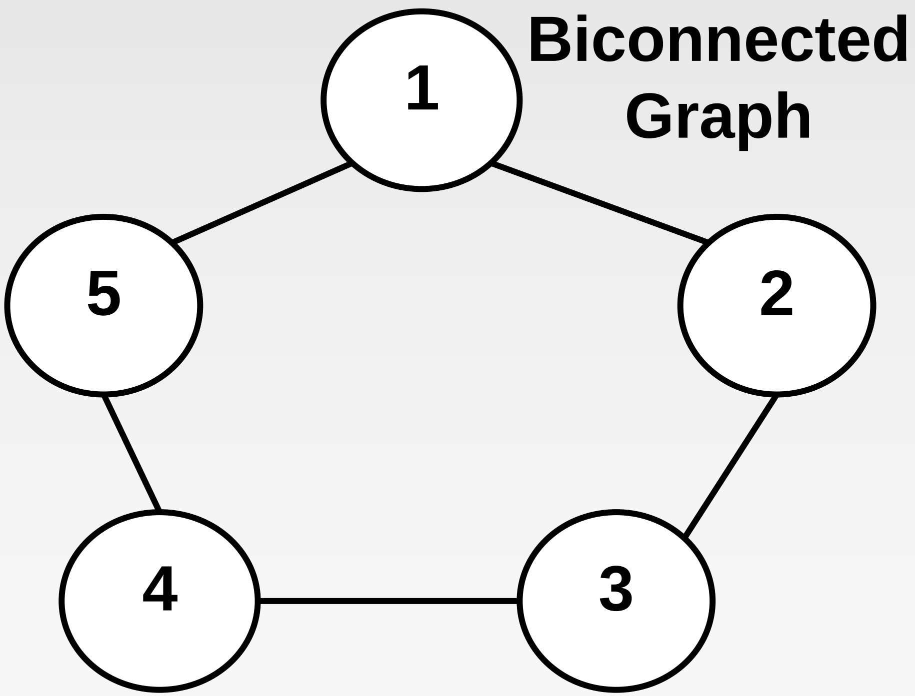
BFS Spanning Tree

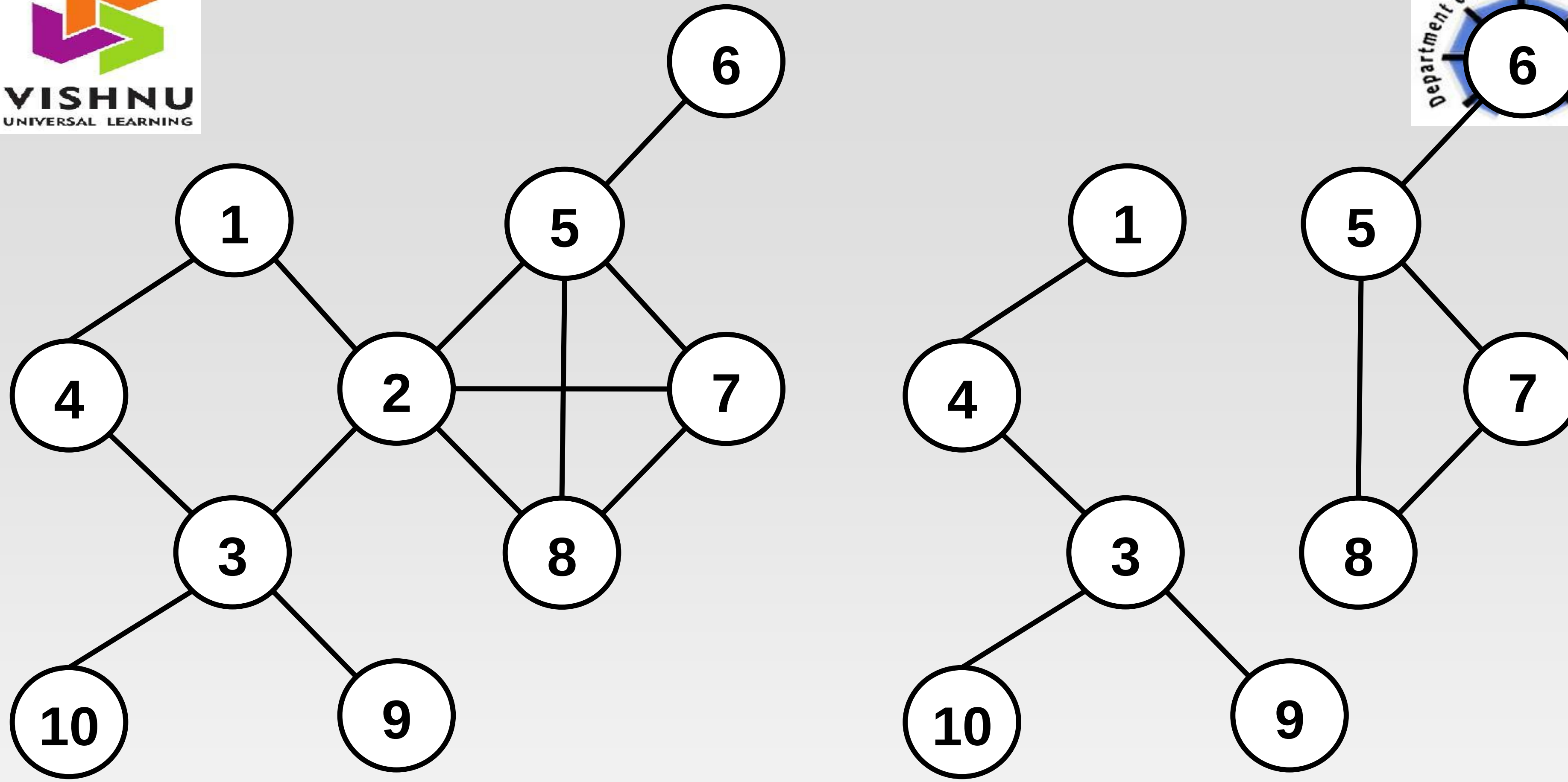


A Graph G_2 with two connected Components H1 & H_2

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.
- **Biconnected** : A graph **G** is biconnected if and only if it contains no articulation points.
- The presence of articulation points in connected graph is undesirable feature.
- The graph shown in figure is biconnected graph example.
- The graph shown in next slide figure is not biconnected graph as deleting vertex 2 we will get two Graphs.



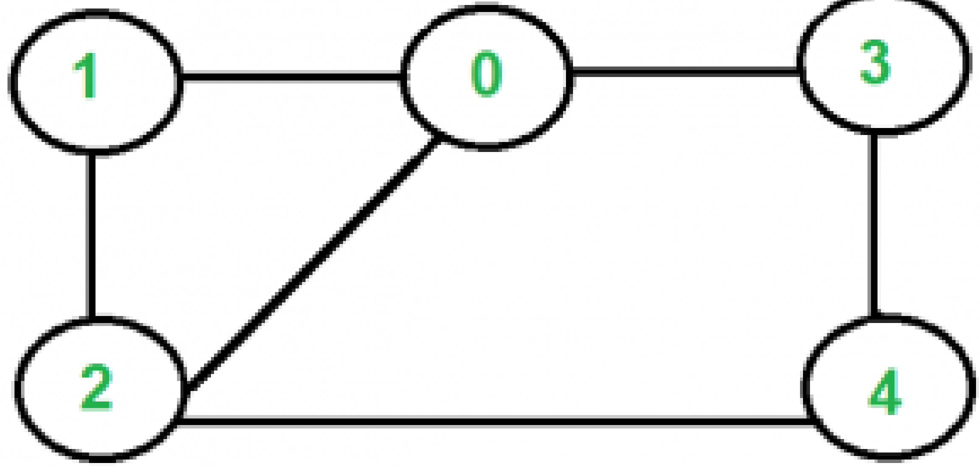


(a) Graph G

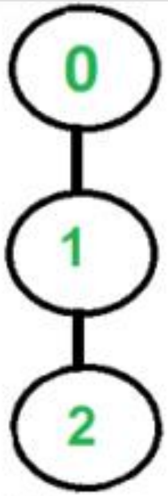
(b) Result of deleting vertex 2

An Example Graph G – Not Biconnected Graph

Biconnected

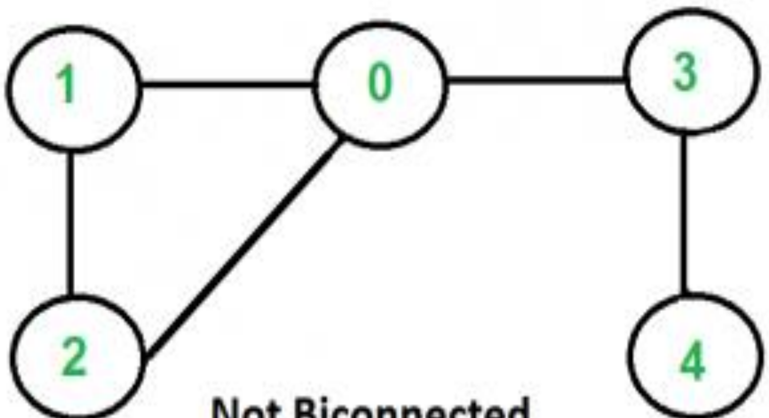


Biconnected

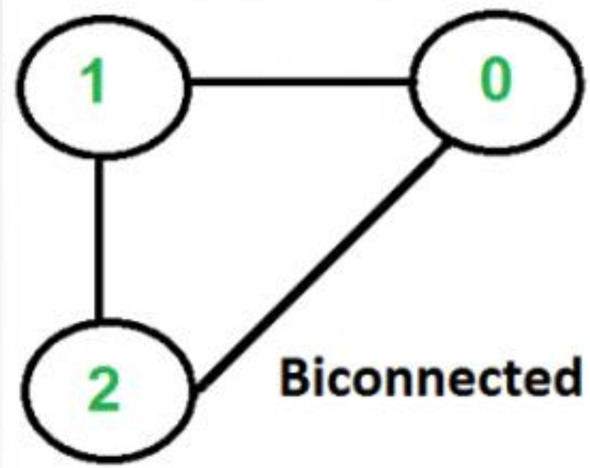


Not Biconnected

Few more Biconnected & Non Biconnected Graphs



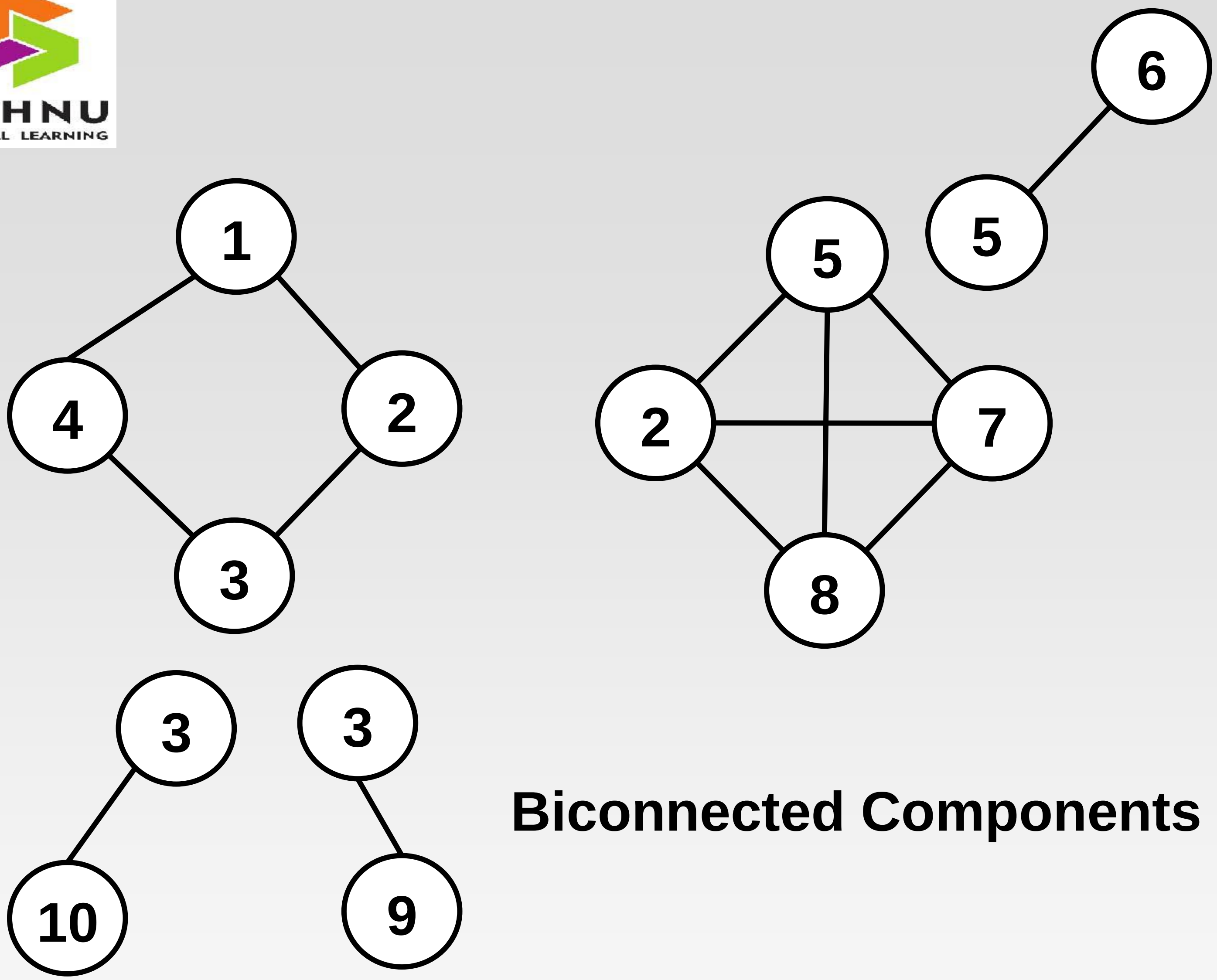
Not Biconnected



Biconnected

Biconnected Components

- For example, if **G** represents a communication network with the vertices representing communication stations and edges communication lines, then 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** fails, we can still communicate between every two stations not including station **i**.
- We will see an efficient algorithm to test whether a connected graph is biconnected.
- For the case of graphs that are not biconnected, this algorithm will identify all articulation points.
- Next slide shows the biconnected components of previous example Graph **G**.



Biconnected Components of Graph G

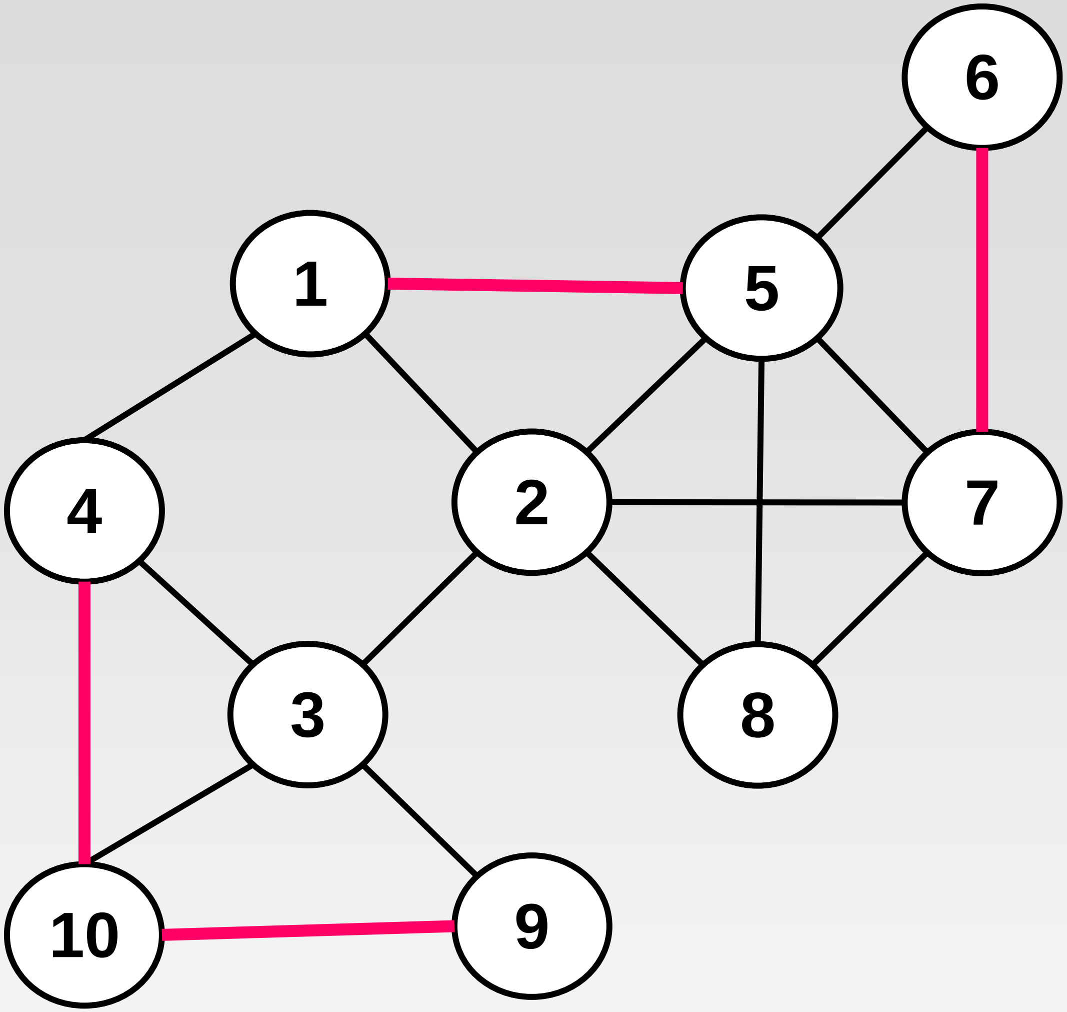
Biconnected Components

- It is easy to show that, Two biconnected component can have at most one vertex in common and this vertex as an articulation point.
- The graph **G** can be transformed into a biconnected graph by using edge addition scheme algorithm.

1. **for** each articulation point a **do**
2. **{**
3. Let $\mathbf{B_1, B_2, \ldots., B_k}$ be the biconnected
4. components containing vertex a;
5. Let $\mathbf{v_i, v_i \neq a,}$ be a vertex in $\mathbf{B_i, 1 \leq i \leq k;}$
6. Add to **G** the edges $\mathbf{(v_i, v_{i+1}), 1 \leq i < k;}$
7. **}**

Biconnected Components

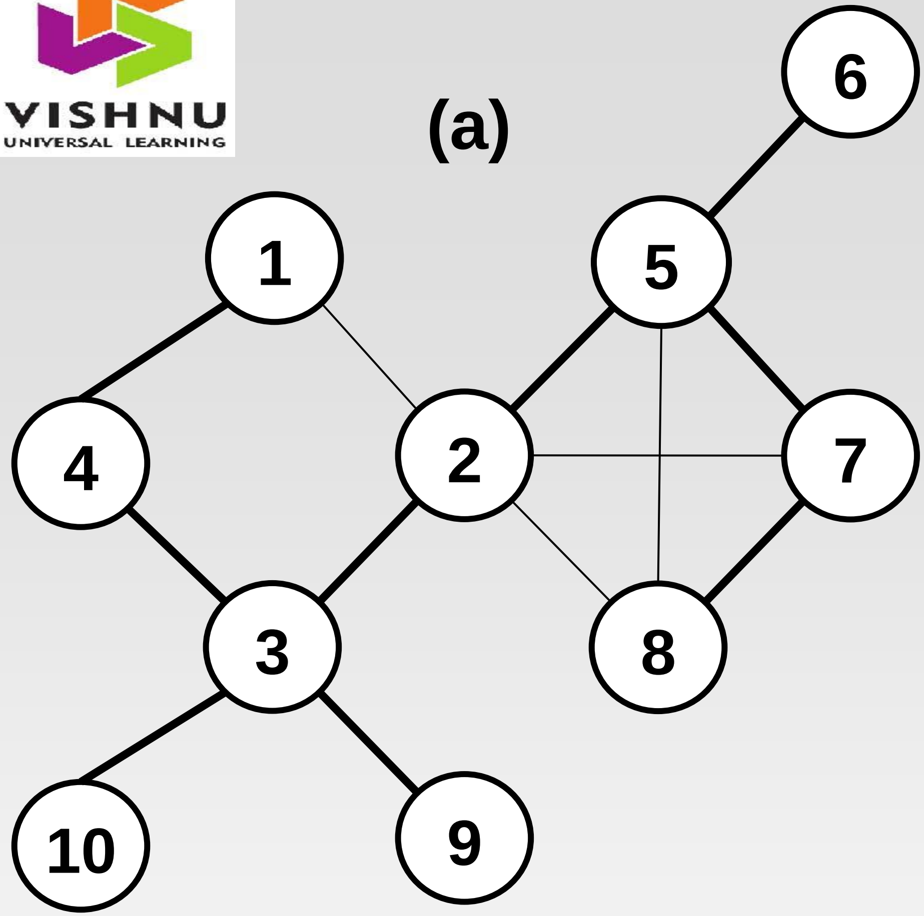
- Using the above scheme to transform the graph **G** seen in previous example into a biconnected graph requires us to…
- Add edges (4, 10) and (10, 9) corresponding to articulation point 3.
- Addedge (1, 5) corresponding to articulation point 2.
- And add edge (6, 7) corresponding to articulation point 5.
- Note that once the edges (v_i, v_{i+1}) as per line 6 (of algorithm in previous slide) are added, vertex **a** is no longer an articulation point.
- We can conclude that addition of the edges corresponding to all articulation points, G has no articulation point and so it is biconnected graph

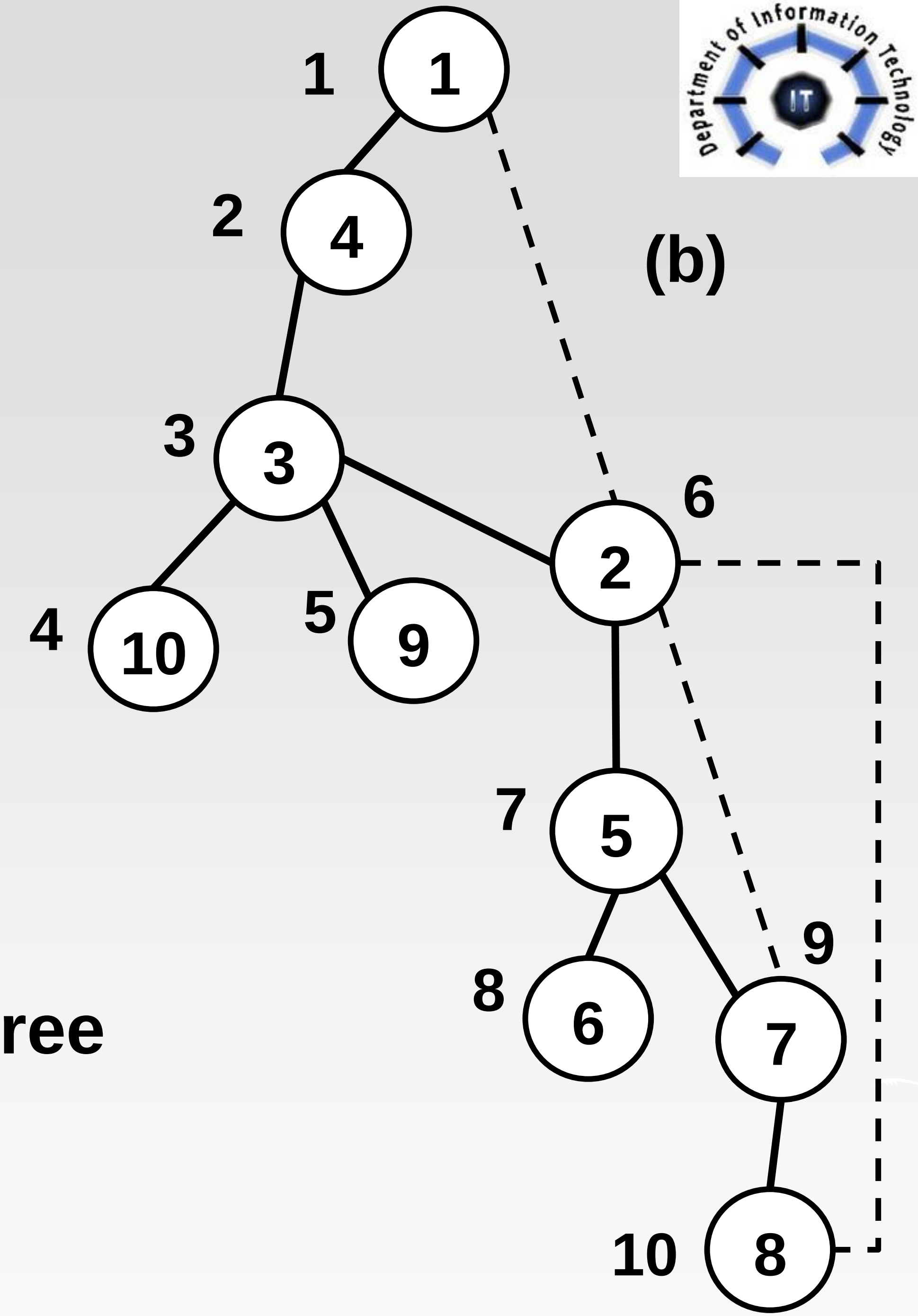


Biconnected graph corresponding to Graph G

Biconnected Components

- Now, we will see how to identify the articulation points and biconnected components of a connected graph **G** with **n >= 2** vertices.
- The problem is efficiently solved by using **Depth First Search (DFS)** Spanning Tree.
- The depth first spanning tree of the Graph **G** is shown in next slide.
- There is a number outside each vertex.
- These numbers correspond to the order in which a depth first search visits these vertices and are referred to as **depth first numbers (dfns)** of the vertex.
- The **solid edges** form the depth first spanning tree are called as **tree edges** and **broken edges** (all other edges) are called **back edges**.





A depth first spanning tree of the Graph G

Biconnected Components

- The **root node** of a depth first spanning tree is an **articulation point** iff it has **at least two children**
- Also if **u** 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 descendants of **w** and a back edge.
- Note that if this cannot be done for some child **w** of **u**, them the deletion of vertex **u** leaves behind at least two nonempty components, this observation leads to a simple rule to identify articulation points.
- For each vertex ***u***, define ***L***[***u***] as follows :

$$L[\,u\,] = \min \{\, dfn[\,u\,],\ \min \{\, L[\,w\,] \mid w \text{ is a child of } u \,\},\ \min \{\, dfn[\,w\,] \mid (u, w) \text{ is a back edge} \,\}\,\}$$

- If ***u*** is not the root, then ***u*** is an articulation point iff ***u*** has a child w such that $L[\,w\,] >= dfn[\,u\,]$

Biconnected Components

- **Example :** For spanning tree of Graph G - as shown in previous slide fig (b) the L values are

L [1:10] = { 1, 1, 1, 1, 6, 8, 6, 6, 5, 4 }

And dfn [1:10] = { 1, 6, 3, 2, 7, 8, 9, 10, 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

- Vertex 5 is an articulation point as child 6 has

  L[6] = 8 and dfn[5] = 7

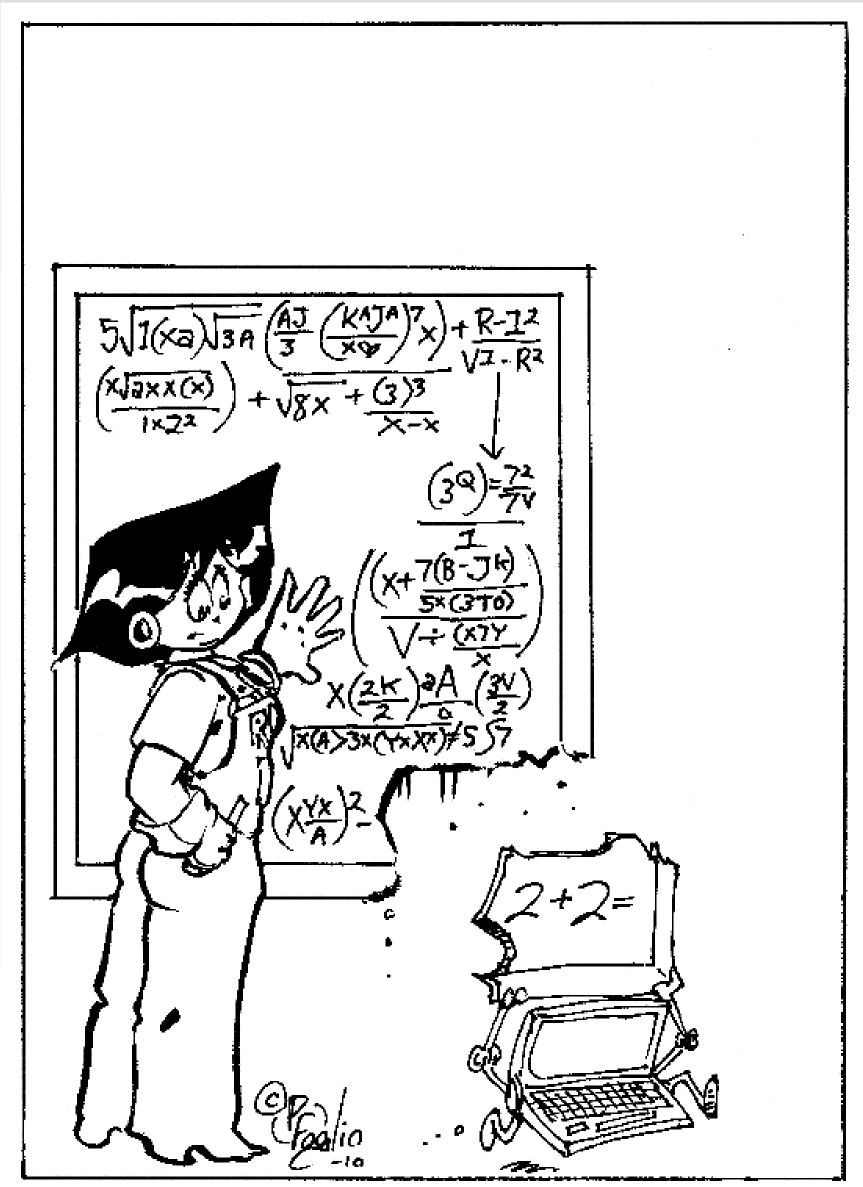
```
Algorithm Art(u, v)
// u is a start vertex for depth first search. v is its parent if any
// in the depth first spanning tree. It is assumed that the global
// array dfn is initialized to zero and that the global variable
// num is initialized to 1. n is the number of vertices in G.
{
    dfn[u] := 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]);
    }
}
```

Algorithm 6.9 Pseudocode to compute *dfn* and *L*

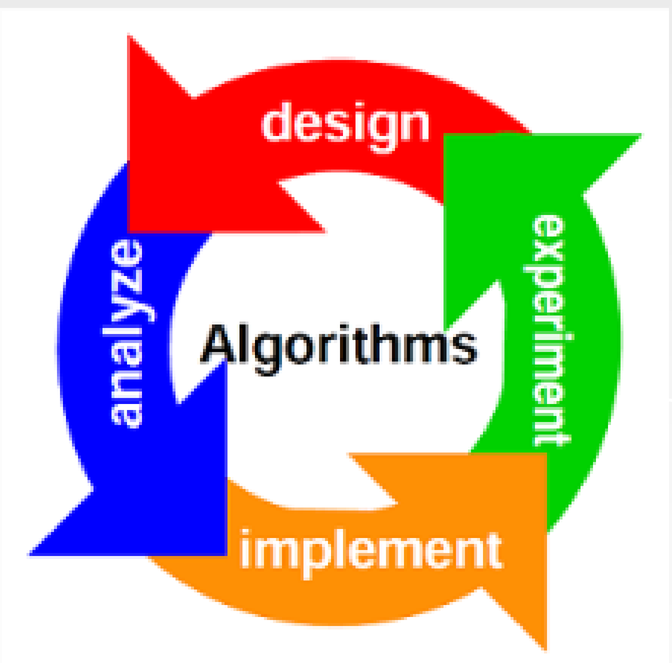
```
Algorithm BiComp(u, v)
// u is a start vertex for depth first search. v is its parent if
// any in the depth first spanning tree. It is assumed that the
// global array dfn is initially zero and that the global variable
// num is initialized to 1. n is the number of vertices in G.
{
    dfn[u] := 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]);
    }
}
```

Algorithm 6.10 Pseudocode to determine bicomponents

Divide-and-Conquer



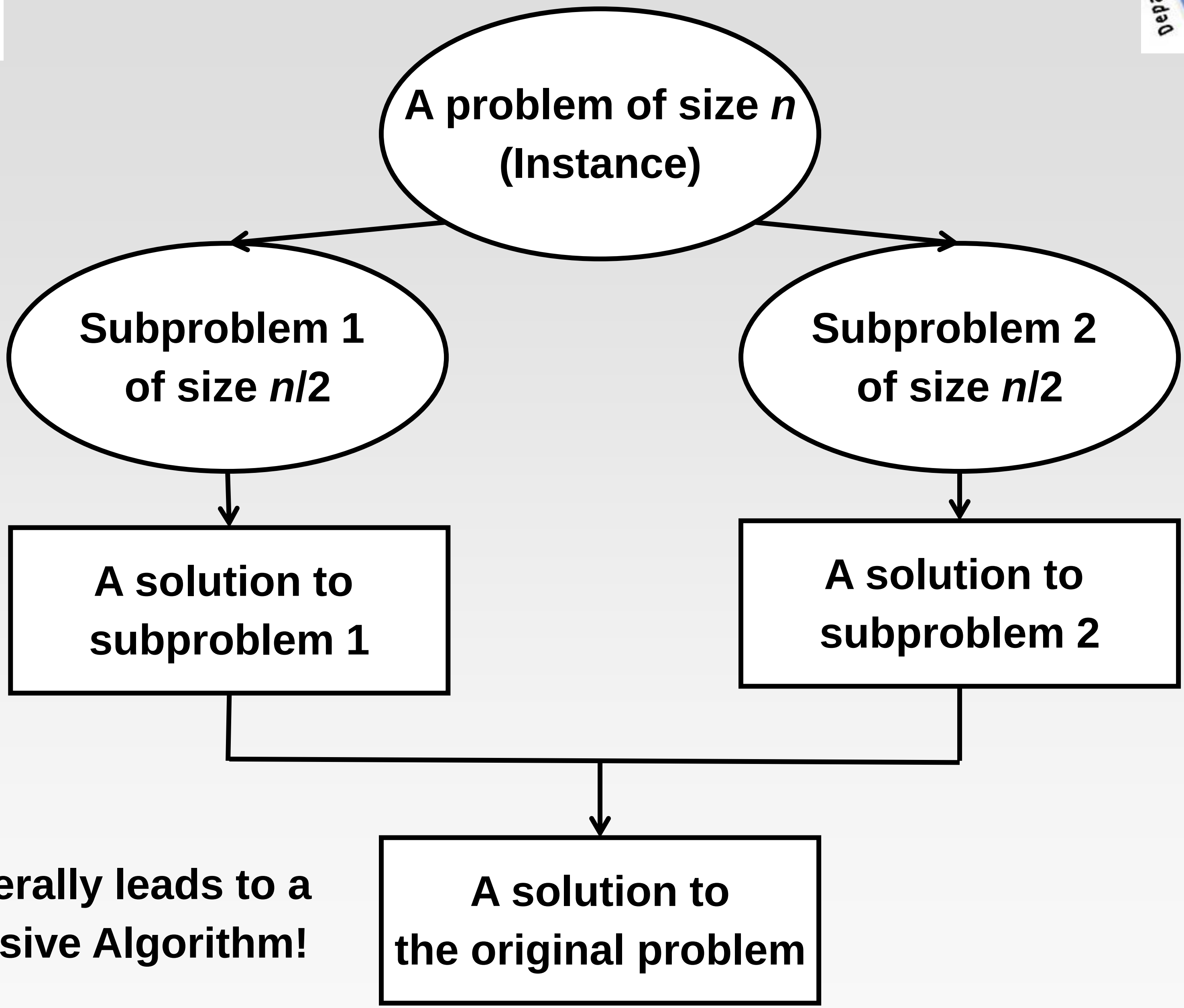
- **Binary Search**
- **Merge Sort**
- **Strassen's Matrix Multiplication.**



Divide-and-Conquer

- The most-well known algorithm design strategy.
- As its name implies **Divide-and-Conquer** involves dividing a problem into smaller problems that can be more easily solved.
- While the specifics vary from one application to another, divide-and-conquer always includes the following 3 steps in some form:
- **Divide** - Typically this steps involves splitting one problem into two problems of approximately ½ the size of original problem.
- **Conquer** - The divide step is repeated (usually recursively as subproblems are same type as original problem) until individual problem sizes are small enough to be solved (conquered) directly.
- **Recombine** - The solution to the original problem is obtained by combining all the solutions to the sub-problems.
- Divide and Conquer is not applicable to every problem class.
- Even when Divide and Conquer works it may not provide for an efficient solution.

Divide-and-Conquer Technique



It generally leads to a Recursive Algorithm!

Divide-and-Conquer

- **Control Abstraction** : A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.
- The control abstraction for divide and conquer technique is DAndC (P), where P is the problem to be solved.

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

- For divide-and-conquer Small (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting.
- If this is so function ‘S’ is invoked otherwise, the problem ‘p’ into smaller sub problems.
- These sub problems $\mathbf{P_1, P_2, \ldots. P_k}$ are solved by recursive application of DAndC.
- Combine is a function that determines the solution to P using the solutions to the k subproblems.
- If the size of P is n and sizes of the k subproblems are $\boldsymbol{n_1, n_2, \ldots. n_k}$ then the computing time of DAndC is:

$$\mathbf{T}(\boldsymbol{n}) = \begin{cases} \boldsymbol{g(n)} & \boldsymbol{n\ small} \\ \boldsymbol{T(n_1) + T(n_2) + \ \ldots\ldots + T(n_k) + f(n)} & \boldsymbol{Otherwise} \end{cases}$$

- Where, **T(*n*)** is the time for DAndC on ‘*n*’ inputs,
- **g (*n*)** is the time to complete the answer directly for small inputs
- The function **f(*n*)** is the time for Dividing P and Combing solutions to subproblems.

- The complexity of many divide-and-conquer algorithms is given by recurrences of the form
- **T(*n*) =** $\begin{cases} T(1) & n = 1 \\ a\,T(n/b) + f(n) & n > 1 \end{cases}$
- Where, a and b are known constants.
- One of the method for solving any such recurrence relation is substitution method.
- For example consider the case where a = 2 and b = 2.
- Let T(1) = 2 and $f(n)$ = n, we have

$$T\,(n) = 2\,T(n/2) + n$$
$$= 2\,[\,2\,T(n/4) + n/2] + n$$
$$= 4\,T(n/4) + 2n$$
$$= 4\,[\,2\,T(n/8) + n/4] + 2n \quad ....$$
$$= 8\,T(n/8) + 3n$$

- In general, we see that

$$T\,(n) = 2^i\,T(n/2^i) + i\,n \qquad for\ any\ log_2\,n \geq i \geq 1$$

Binary Search

- Let $a_i, 1 \leq i \leq n$, be a list of elements that are sorted in order.
- When we are given a element 'x', binary search is used to find the corresponding element from the list.
- In case 'x' is present, we have to determine a value 'j' such that $a_j = x$ (successful search).
- If 'x' is not in the list then 'j' is to set to Zero (unsuccessful search).
- Divide-and-conquer can be used to solve this problem.
- Let Small(P) be true if $n = 1$.
- In this case, S(P) will take the value $i \; if \; x = ai$; otherwise it will take the value 0.
- If P has more than one element, it can divided (or reduced) into a new subproblem.

```
Algorithm BinSearch (a, n, x )
// Given an array a[1: n] of elements in increasing order,
// n ≥ 0, determine whether 'x' is present, and if so,
// return 'j' such that x = a[j] else return 0.
{
 low := 1; high := n;
  while (low ≤ high) do
  {
   mid := ⌊(low + high)/2⌋ ;
    if (x < a[mid]) then high := mid − 1;
    else if (x > a[mid]) then low := mid + 1;
            else return mid;
  }
   return 0;
 }
```

Example : Consider 14 elements

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Elements	- 15	- 6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons	3	4	2	4	3	4	1	4	3	4	2	4	3	4

Low	High	Mid
Case 1 : Search $x = 151$		
1	14	7
8	14	11
12	14	13
14	14	14
		Found
Case 3 : Search $x = 9$		
1	14	7
1	6	3
4	6	5
		Found

Low	High	Mid
Case 2 : Search $x = -14$		
1	14	7
1	6	3
1	2	1
2	2	2
2	1	Not Found
Case 4 : Search $x = -43$		
1	14	7
1	6	3
1	2	1
1	0	Not Found

Binary Search

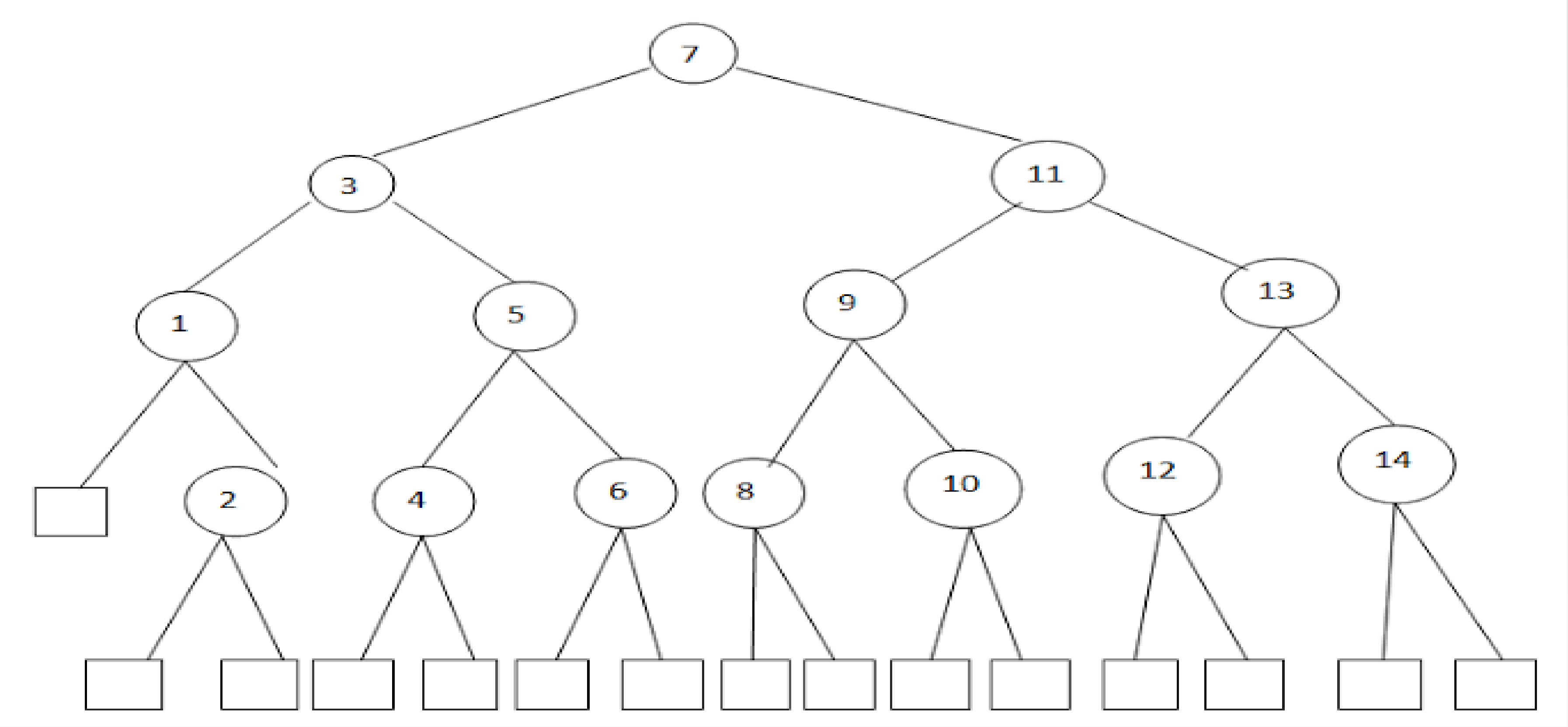
- It is found that No element requires more than 4 comparisons to be found (See Case 1 & 3).
- The average is obtained by summing the comparisons (Mentioned below elements) needed to find all 14 elements and dividing by 14.

$\mathbf{45/14 \approx 3.21}$ comparisons per successful search on average.

- There are 15 possible ways that an unsuccessful search may terminate depending on the value of $\boldsymbol{x}$.
- If $\boldsymbol{x < a[1]}$ the algorithm requires 3 elements comparison (See case 4) to determine that $\boldsymbol{x}$ is not present.
- For all remaining possibilities, BinSearch requires 4 elements comparison (See case 2) to determine that $\boldsymbol{x}$ is not present
- Thus the average number of elements comparisons for an unsuccessful search is $\mathbf{(3 + 14 \ * 4)\ /15 = 59\ /15 \approx 3.93}$.

Binary Search

- A better way to understand the algorithm is to consider the sequence of values for mid that are produced by BinSearch for all possible values of x.
- These values are nicely described using a binary decision tree.



Binary decision tree for binary search, $n = 14$

Binary Search

- If n is in the range of $[2^{k-1}, 2^k]$ then BinSearch makes at most k elements comparisons for a successful search and
- either $k - 1$ or k comparisons for an unsuccessful search.
- The computing time of binary search by giving formulas that describe the Best, Average, and Worst cases :

Successful Searches				**Unsuccessful Searches**
O (1)	**O ($\log n$)**	**O ($\log n$)**		**O ($\log n$)**
Best	**Average**	**Worst**		**Best, Average, Worst**

Merge Sort

- Merge Sort is an another classical example of a Divide-and-Conquer algorithm which has nice property that in the worst case its complexity is **O ($\boldsymbol{n}\log \boldsymbol{c}$)**.
- Given a sequence of $\boldsymbol{n}$ elements $\boldsymbol{a}[\mathbf{1}], \boldsymbol{a}[\mathbf{2}], \ldots\ldots \boldsymbol{a}[\boldsymbol{n}]$ to be sorted in increasing order.
- The merge sort algorithm divides input array in two sets, $\boldsymbol{a}[\mathbf{1}], \boldsymbol{a}[\mathbf{2}], \ldots\ldots \boldsymbol{a}[\lfloor \boldsymbol{n}/\mathbf{2} \rfloor]$ and $\boldsymbol{a}[\lfloor \boldsymbol{n}/\mathbf{2} \rfloor + \mathbf{1}], \ldots\ldots \boldsymbol{a}[\boldsymbol{n}]$
- And each set is individually sorted and resulting sorted sequences are merged to produce a single sorted sequence of $\boldsymbol{n}$ elements.
- Thus we can use Divide-and-Conquer strategy in which splitting is into two equal-sized sets and combining operation is merging of two sorted sets into one.

Merge Sort Algorithm

Algorithm MergeSort ($low, high$)
//$a[low : high]$ is a global array to be sorted.
{
  if ($low < high$) then // If there are more than one element
  {
    // Divide into subproblems.
    mid := $\lfloor (low + high)/2 \rfloor$; // Finds where to split the set
    MergeSort (low, mid**)** // Sort one subset
    MergeSort ($mid, high$**)** // Sort the other subset
    Merge ($low, mid, high$**)** // Combine the solutions
  }
}

Merge Algorithm

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

```
if (h > mid) then
 for k := j to high do
 {
b[i] := a[k]; i := i + 1;
}
else for k := h to mid do
{
 b[i] := a[k]; i := i + l;
}
for k := low to high do
   a[k] := b[k];
}
```

Merging two sorted subarrays using auxiliary storage

Merge Sort Example

- Consider an array of ten elements

  a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520).
- Algorithm MergeSort begins by splitting a[ ] into two subarrays each of size five (a[1:5] and a[6:10]).
- The elements in a[1:5] are then split into two subarrays of size three (a[1:3]) and two (a[4:5]).
- Then the items in a[1:3] are split into subarrays of size two (a[1:2]) and one (a[3:3]).
- The two values in a[1:2] are split final time into one-element subarrays , and now the merging begins.
- A record of the subarrays is implicitly maintained by the recursive mechanism.

Merge Sort Example

- Pictorially the file can be viewed as

  (310 | 285 |179 | 652 , 351 | 423 , 861 , 254 , 450 , 520)

- where vertical bars indicate the boundaries of subarrays.
- Elements a[1] and a[2] are merged to yield

  (**285 , 310** |179 | 652 , 351 |423 , 861 , 254 , 450 , 520)

- Then a[3] is merged with a[1:2] and

  (**179 , 285 , 310** |652 , 351 | 423 , 861 , 254 , 450 , 520)

- Next elements a[4] and a[5] are merged:

  (179 , 285 , 310 |**351 , 652** | 423 , 861 , 254 , 450 , 520)

- And then a[1:3] and a[4:5]:

  (**179 , 285 , 310 , 351 , 652** | 423 , 861 , 254 , 450 , 520)

- At this point the algorithm has returned to the first invocation of MergeSoft and is about to process second recursive call.

Merge Sort Example

- Repeated recursive calls are invoked producing following subarrays

  (179 , 285 , 310 , 351 , 652 | 423 | 861 | 254 | 450 , 520)

- Elements a[6] and a[7] are merged. Then a[8] is merged with a[6:7]

  (179 , 285 , 310 , 351 , 652 | **254 , 423 , 861** | 450 , 520)

- Next a[9] and a[10] are merged and then a[6:8] and a[9:10]:

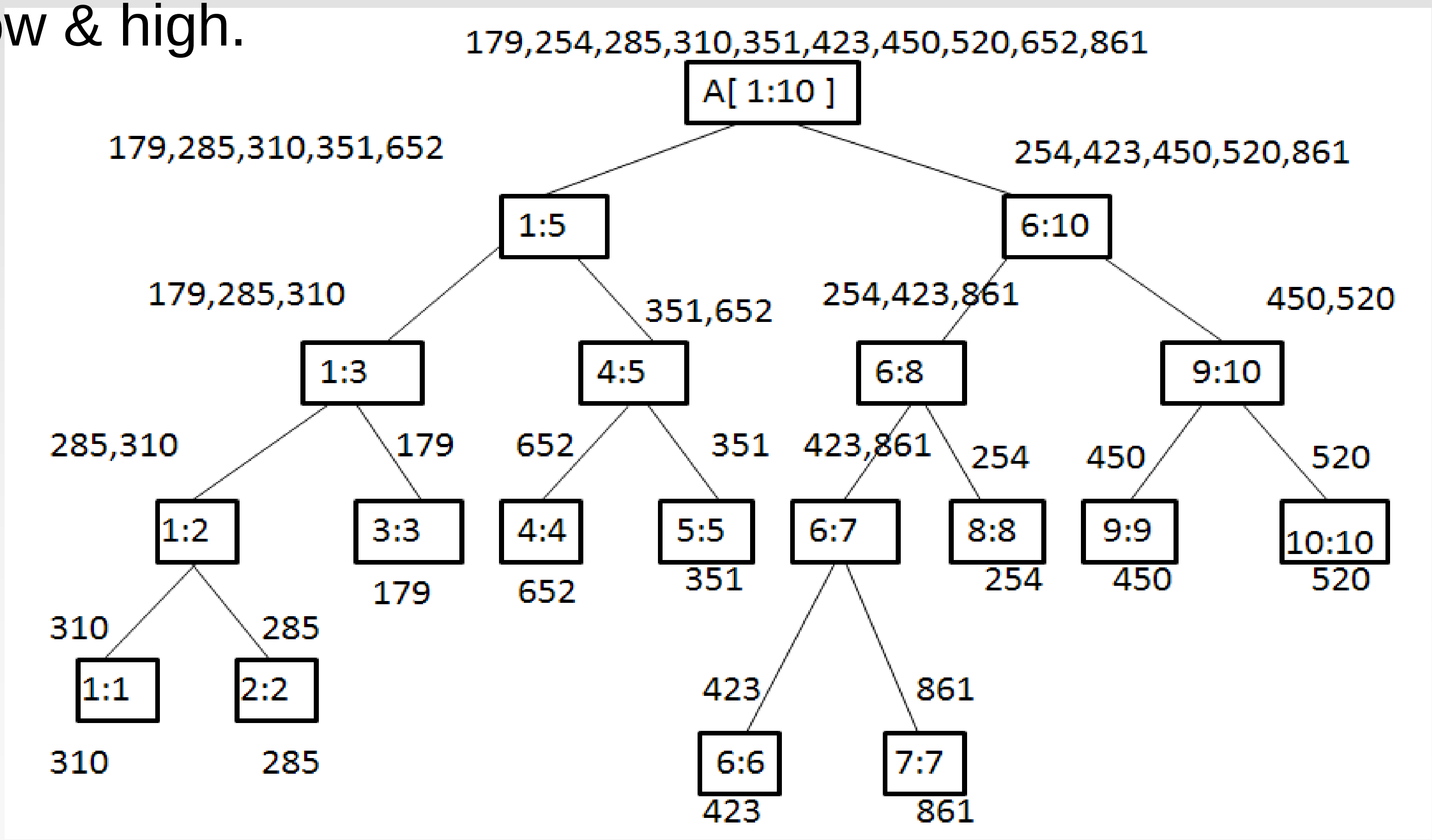
  (179 , 285 , 310 , 351 , 652 | **254, 423, 450, 520, 861**)

- At this point there are two sorted subarrays and the final merge produces fully sorted result.

  (179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

Merge Sort Example

- The tree represents the sequence of recursive call that are produced by MergeSort.
- The pairs of values in each node are the values of parameters low & high.

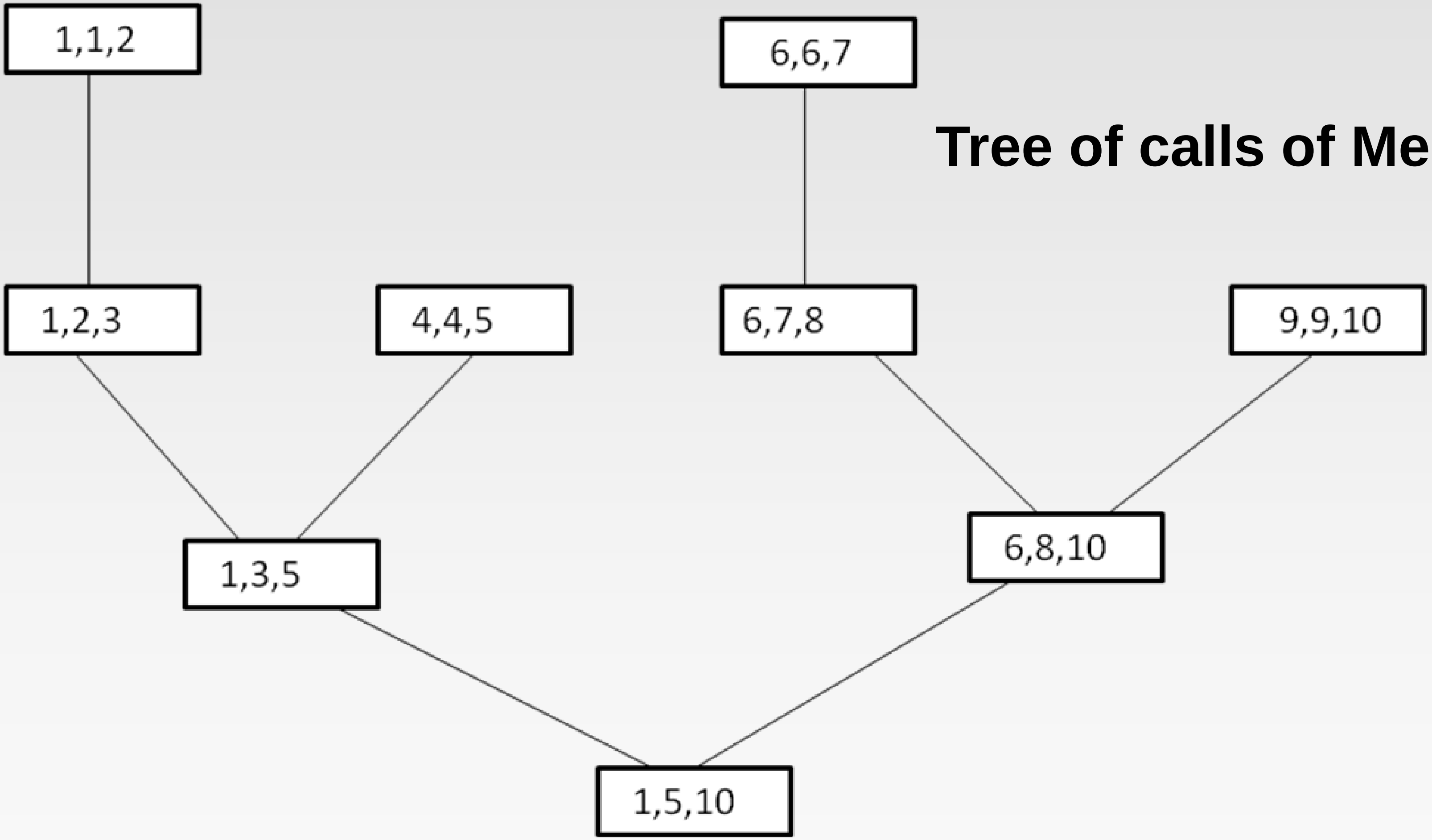


Tree of calls of MergeSort (1, 10)

Merge Sort Example

- The tree represents calls to procedure **Merge** by MergeSort.
- Reading Tree → Example : The node containing 1, 2, and 3 represents the merging of a[1:2] with a[3].

Tree of calls of Merge



Time Complexity of Merge Sort

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

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

- When n is a power of 2, $n = 2^k$, (i.e. $k = \log n$) we can solve equation by substitutions

$$\begin{aligned} T(n) &= 2\,T(n/2) + c * n \\ &= 2\,T(2\,T(n/4) + c * n/2) + c * n \\ &= 4\,T(n/4) + 2\,c * n \\ &= 4\,(2\,T(n/8) + c * n/4) + 2 * c * n \\ &= 8\,T(n/8) + 3 * c * n \\ &\;\vdots \\ &= 2^k T(1) + k * c * n \\ &= a * n + c * n \log n \end{aligned}$$

- It is easy to see that is $2^k < n \leq 2^{k+1}$, $\textit{then } T(n) \leq T(2^{k+1})$.
- Therefore $T(n) = O(n \log n)$

Strassen's Matrix Multiplication

- Let A and B be two $\boldsymbol{n\ X\ n}$ matrices. The product matrix $\boldsymbol{C = AB}$ is calculated by using the formula,

  $$\boldsymbol{C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j)}$$

- To compute $\boldsymbol{C(i,j)}$ we need $\boldsymbol{n}$ multiplications. As the matrix has $\boldsymbol{n^2}$ elements, the time complexity for the Matrix Multiplication is $\boldsymbol{O(n^3)}$
- The Divide-and-Conquer strategy suggest another way to find matrix multiplication of two $\boldsymbol{n\ X\ n}$ matrices.
- We assume that $\boldsymbol{n}$ is a power of 2, that is, there exists a integer $\boldsymbol{k}$ such that $\boldsymbol{n = 2^k}$
- Imagine that A and B are each partitioned into four square submatrices, each submatrix having dimension $\boldsymbol{\frac{n}{2}\ X\ \frac{n}{2}}$

Strassen's Matrix Multiplication

- If $n = 2$ then the product AB can be computed using above formula.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Then

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$
$$C_{12} = A_{11} B_{12} + A_{12} B_{22}$$
$$C_{21} = A_{12} B_{11} + A_{22} B_{21}$$
$$C_{22} = A_{21} B_{12} + A_{22} B_{22}$$

- For $n > 2$ the elements of C can be computed using matrix multiplication and addition operations applied to matrix of size $n/2 \ X \ n/2$.
- Since 'n' is a power of 2, these product can be recursively computed using the same formula .This Algorithm will continue applying itself to smaller sub matrix until 'n' become suitable small (n =2) so that the product is computed directly.

Strassen's Matrix Multiplication

- To compute $\boldsymbol{AB}$, we need eight multiplication and four additions of $\boldsymbol{n/2\ X\ n/2}$ matrices.
- Since two $\boldsymbol{n/2\ X\ n/2}$ matrices can be added in time $\boldsymbol{cn^2}$ for some constant c, the overall computing time $\boldsymbol{T(n)}$ of the resulting divide-and-conquer is given by the recurrence

$$\boldsymbol{T(n) = \begin{cases} b & n \leq 2 \\ 8\,T(n/2) + cn^2 & n > 2 \end{cases}}$$

- Where b and c are constants.
- This recurrence can be solved to obtain $\mathbf{T}(\boldsymbol{n}) = \boldsymbol{O}(\boldsymbol{n^3})$, hence no improvement over conventional method.
- Since matrix multiplications are more expensive than matrix additions, we can attempt to reformulate the equations for $\boldsymbol{C_{ij}}$ so as to have fewer multiplications and possibly more additions.

Strassen's Matrix Multiplication

- Volker Strassen has discovered a way to compute C_{ij} using only 7 multiplications and 18 additions.
- The idea of **Strassen's method** is to reduce the number of multiplications to 7.
- Strassen's method is similar to simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size $n/2 \ X \ n/2$, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

- His method involves first computing seven terms P, Q, R, S, T, U and V using 7 multiplications & 10 additions/subtractions.

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22})B_{11}$$
$$R = A_{11}(B_{12} - B_{22})$$
$$S = A_{22}(B_{21} - B_{11})$$
$$T = (A_{11} + A_{12})B_{22}$$
$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

- The C_{ij} require an additional 8 additions or subtractions.

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

- The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \le 2 \\ 7\,T(n/2) + an^2 & n > 2 \end{cases}$$

where a and b are constants.

$$\begin{aligned}
T(n) &= 7\,T(n/2) + an^2 \\
&= 7\,(7\,T(n/4) + a(n/2)^2) + an^2 \\
&= 7^2 T(n/4) + a\,n^2(1 + \tfrac{7}{4}) \\
&= 7^3 T(n/8) + a\,n^2(1 + \tfrac{7}{4} + (\tfrac{7}{4})^2) \\
&= 7^k T(n/2^k) + a\,n^2\left(1 + \tfrac{7}{4} + \left(\tfrac{7}{4}\right)^2 \ldots. + \left(\tfrac{7}{4}\right)^{k-1}\right) \\
&(As\ n = 2^k \ \rightarrow \mathbf{k = log\ n\ \&\ T(1) = b}) \\
&\le 7^{log\,n}\, b + a\,n^2 \left(\tfrac{7}{4}\right)^{log\,n} \\
&\le b\,n^{log\,7} + a\,n^2\, n^{log(7/4)} \\
&\le b\,n^{log\,7} + a\,n^{2 + log\,7 - log\,4} \\
&\le b\,n^{log\,7} + a\,n^{log\,7}
\end{aligned}$$

$$\mathrm{T}(n) = O\left(n^{log\,7}\right) = O(n^{2.81})$$

Next - Unit III

Greedy Method