UNIT – I

Introduction of Algorithm

Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the i/p into the o/p.

Formal Definition:

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

1. INPUT → Zero or more quantities are externally supplied.
2. OUTPUT → At least one quantity is produced.
3. DEFINITENESS → Each instruction is clear and unambiguous.
4. FINITENESS → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. EFFECTIVENESS → Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Issues or study of Algorithm:

- How to device or design an algorithm → creating and algorithm.
- How to express an algorithm → definiteness.
- How to analysis an algorithm → time and space complexity.
- How to validate an algorithm → fitness.
- Testing the algorithm → checking for error.

Algorithm Specification:

Algorithm can be described in three ways.

1. Natural language like English:
   When this way is choused care should be taken, we should ensure that each & every statement is definite.

2. Graphic representation called flowchart:
   This method will work well when the algorithm is small& simple.

3. Pseudo-code Method:
   In this method, we should typically describe algorithms as program, which resembles language like Pascal & algol.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,

```
Node. Record
{
  data type – 1   data-1;
        .
        .
        .
 data type – n  data – n;
 node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

<Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.

→ Logical Operators AND, OR, NOT
→Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed.

For, while and repeat-until

While Loop:

```
While < condition > do
{
        <statement-1>
              .
              .
              .

        <statement-n>
}
```

For Loop:

For variable: = value-1 to value-2 step step do

```
{
  <statement-1>
        .
        .
        .
<statement-n>
}
```

repeat-until:

```
repeat
        <statement-1>
                .
                .
                .
        <statement-n>
until<condition>
```

8. A conditional statement has the following forms.

   → If <condition> then <statement>
   → If <condition> then <statement-1>
   Else <statement-1>

   Case statement:

```
Case
{
        : <condition-1> : <statement-1>
                .
                .
                .
        : <condition-n> : <statement-n>
        : else : <statement-n+1>
}
```

9. Input and output are done using the instructions read & write.

10. There is only one type of procedure:
    Algorithm, the heading takes the form,

        Algorithm Name (Parameter lists)

    → As an example, the following algorithm fields & returns the maximum of 'n' given numbers:

```
1. algorithm Max(A,n)
2. // A is an array of size n
3. {
4. Result := A[1];
5. for I:= 2 to n do
6.    if A[I] > Result then
7.          Result :=A[I];
8.  return Result;
9. }
```

In this algorithm (named Max), A & n are procedure parameters. Result & I are Local variables.

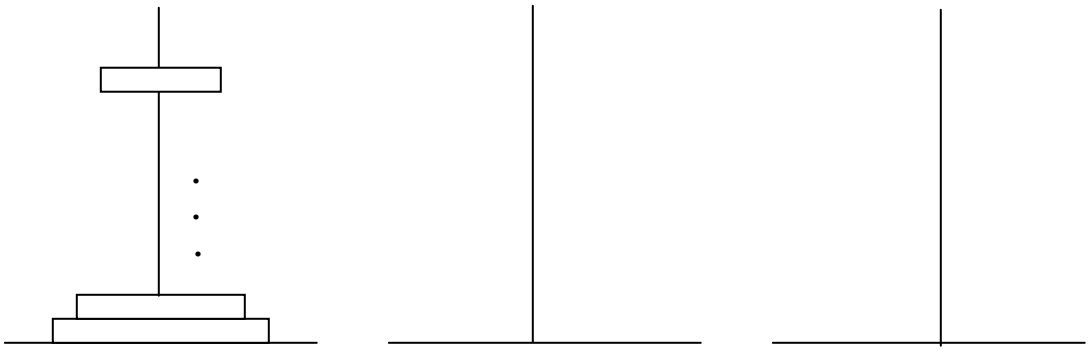
→ Next we present 2 examples to illustrate the process of translation problem into an algorithm.

Recursive Algorithms:

- A Recursive function is a function that is defined in terms of itself.
- Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is Direct Recursive.
- Algorithm 'A' is said to be Indirect Recursive if it calls another algorithm which in turns calls 'A'.
- The Recursive mechanism, are externally powerful, but even more importantly, many times they can express an otherwise complex process very clearly. Or these reasons we introduce recursion here.
- The following 2 examples show how to develop a recursive algorithms.

  → In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

1. **Towers of Hanoi:**



- It is Fashioned after the ancient tower of Brahma ritual.
- According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.
- The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.
- Besides these tower there were two other diamond towers(labeled B & C)
- Since the time of creation, Brehman priests have been attempting to move the disks from tower A to tower B using tower C, for intermediate storage.
- As the disks are very heavy, they can be moved only one at a time.
- In addition, at no time can a disk be on top of a smaller disk.
- According to legend, the world will come to an end when the priest have completed this task.
- A very elegant solution results from the use of recursion.
- Assume that the number of disks is 'n'.
- To get the largest disk to the bottom of tower B, we move the remaining 'n-1' disks to tower C and then move the largest to tower B.
- Now we are left with the tasks of moving the disks from tower C to B.
- To do this, we have tower A and B available.
- The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk scan be placed on top of it.

Algorithm:

```
1.  Algorithm TowersofHanoi(n,x,y,z)
2. //Move the top 'n' disks from tower x to tower y.
3. {

        .
        .
        .

4.if(n>=1) then
5. {
6.      TowersofHanoi(n-1,x,z,y);
7.      Write("move top disk from tower " X ,"to top of tower " ,Y);
8.  Towersofhanoi(n-1,z,y,x);
9.  }
10. }
```

2. Permutation Generator:

- Given a set of n>=1elements, the problem is to print all possible permutations of this set.

- For example, if the set is {a,b,c} ,then the set of permutation is,

  { (a,b,c),(a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a)}
- It is easy to see that given 'n' elements there are n! different permutations.
- A simple algorithm can be obtained by looking at the case of 4 statement(a,b,c,d)
- The Answer can be constructed by writing

1. a followed by all the permutations of (b,c,d)
2. b followed by all the permutations of(a,c,d)
3. c followed by all the permutations of (a,b,d)
4. d followed by all the permutations of (a,b,c)

Algorithm:

```
Algorithm perm(a,k,n)
{
if(k=n) then write (a[1:n]); // output permutation
else   //a[k:n] ahs more than one permutation
        // Generate this recursively.
for I:=k to n do
{
t:=a[k];
a[k]:=a[I];
a[I]:=t;
perm(a,k+1,n);
//all permutation of a[k+1:n]
t:=a[k];
a[k]:=a[I];
a[I]:=t;
}
}
```

Performance Analysis:

1. **Space Complexity:**

   The space complexity of an algorithm is the amount of money it needs to run to compilation.

2. **Time Complexity:**

   The time complexity of an algorithm is the amount of computer time it needs to run to compilation.

Space Complexity:

Space Complexity Example:

Algorithm abc(a,b,c)

```
{
return a+b++*c+(a+b-c)/(a+b) +4.0;
}
```

→ The Space needed by each of these algorithms is seen to be the sum of the following component.

1.A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs.

The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

- The space requirement s(p) of any algorithm p may therefore be written as,
  S(P) = c+ Sp(Instance characteristics)

Where 'c' is a constant.

Example 2:

```
Algorithm sum(a,n)
{
    s=0.0;
    for I=1 to n do
    s= s+a[I];
    return s;
}
```

- The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by 'n' is one word, since it is of type integer.
- The space needed by 'a'a is the space needed by variables of tyepe array of floating point numbers.
- This is atleast 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
- So,we obtain Ssum(n)>=(n+s)

[n for a[],one each for n,I a& s]

Time Complexity:

The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

→The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

→ The number of steps any problem statemn t is assigned depends on the kind of statement.

For example, comments → 0 steps.
Assignment statements → 1 steps.
[Which does not involve any calls to other algorithms]

Interactive statement such as for, while & repeat-until→ Control part of the statement.

1. We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.

   This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

Algorithm:

```
Algorithm sum(a,n)
{
        s= 0.0;
        count = count+1;
        for I=1 to n do
        {
         count =count+1;
        s=s+a[I];
        count=count+1;
        }
        count=count+1;
        count=count+1;
        return s;
        }
```

→ If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

→First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.
→By combining these two quantities, the total contribution of all statements, the

step count for the entire algorithm is obtained.

Statement	*S/e*	*Frequency*	*Total*
1. Algorithm Sum(a,n) 2.{ 3. S=0.0; 4. for I=1 to n do 5. s=s+a[I]; 6. return s; 7. }	0 0 1 1 1 1 0	- - 1 n+1 n 1 -	0 0 1 n+1 n 1 0
Total			2n+3

AVERAGE –CASE ANALYSIS

- Most of the time, average-case analysis are performed under the more or less realistic assumption that all instances of any given size are equally likely.
- For sorting problems, it is simple to assume also that all the elements to be sorted are distinct.
- Suppose we have 'n' distinct elements to sort by insertion and all n! permutation of these elements are equally likely.
- To determine the time taken on a average by the algorithm ,we could add the times required to sort each of the possible permutations ,and then divide by n! the answer thus obtained.
- An alternative approach, easier in this case is to analyze directly the time required by the algorithm, reasoning probabilistically as we proceed.
- For any I,$2 \leq I \leq n$, consider the sub array, T[1….i].
- The partial rank of T[I] is defined as the position it would occupy if the sub array were sorted.
- For Example, the partial rank of T[4] in [3,6,2,5,1,7,4] in 3 because T[1….4] once sorted is [2,3,5,6].
- Clearly the partial rank of T[I] does not depend on the order of the element in
- Sub array T[1…I-1].

Analysis

Best case:
This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Worst case:
This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Average case:

This type of analysis results in average running time over every type of input.

Complexity:

Complexity refers to the rate at which the storage time grows as a function of the problem size

Asymptotic analysis:

Expressing the complexity in term of its relationship to know function. This type analysis is called asymptotic analysis.

Asymptotic notation:

Big 'oh': the function f(n)=O(g(n)) iff there exist positive constants c and no such that f(n)≤c*g(n) for all n, n ≥ no.

Omega: the function f(n)=Ω(g(n)) iff there exist positive constants c and no such that f(n) ≥ c*g(n) for all n, n ≥ no.

Theta: the function f(n)=ɵ(g(n)) iff there exist positive constants c1,c2 and no such that c1 g(n) ≤ f(n) ≤ c2 g(n) for all n, n ≥ no.

Recursion:

Recursion may have the following definitions:
-The nested repetition of identical algorithm is recursion.
-It is a technique of defining an object/process by itself.
-Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

When to use recursion:

Recursion can be used for repetitive computations in which each action is stated in terms of previous result. There are two conditions that must be satisfied by any recursive procedure.

1. Each time a function calls itself it should get nearer to the solution.
2. There must be a decision criterion for stopping the process.

In making the decision about whether to write an algorithm in recursive or non-recursive form, it is always advisable to consider a tree structure for the problem. If the structure is simple then use non-recursive form. If the tree appears quite bushy, with little duplication of tasks, then recursion is suitable.

The recursion algorithm for finding the factorial of a number is given below,

Algorithm : factorial-recursion

Input : n, the number whose factorial is to be found.
Output : f, the factorial of n
Method : if(n=0)
f=1
else
f=factorial(n-1) * n
if end
algorithm ends.

The general procedure for any recursive algorithm is as follows,

1. Save the parameters, local variables and return addresses.

2. If the termination criterion is reached perform final computation and goto step 3 otherwise perform final computations and goto step 1

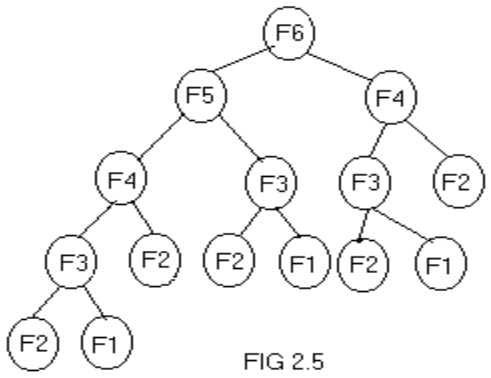


FIG 2.5

3. Restore the most recently saved parameters, local variable and return address and goto the latest return address.

Iteration v/s Recursion:

Demerits of recursive algorithms:

1. Many programming languages do not support recursion; hence, recursive mathematical function is implemented using iterative methods.

2. Even though mathematical functions can be easily implemented using recursion it is always at the cost of execution time and memory space. For example, the recursion tree for generating 6 numbers in a Fibonacci series generation is given in fig 2.5. A Fibonacci series is of the form 0,1,1,2,3,5,8,13,…etc, where the third number is the sum of preceding two numbers and so on. It can be noticed from the fig 2.5 that, f(n-2) is computed twice, f(n-3) is computed thrice, f(n-4) is computed 5 times.

3. A recursive procedure can be called from within or outside itself and to ensure its proper functioning it has to save in some order the return addresses so that, a return to the proper location will result when the return to a calling statement is made.

4. The recursive programs needs considerably more storage and will take more time.

Demerits of iterative methods :

- Mathematical functions such as factorial and Fibonacci series generation can be easily implemented using recursion than iteration.
- In iterative techniques looping of statement is very much necessary.

Recursion is a top down approach to problem solving. It divides the problem into pieces or selects out one key step, postponing the rest.
Iteration is more of a bottom up approach. It begins with what is known and from this constructs the solution step by step. The iterative function obviously uses time that is O(n) where as recursive function has an exponential time complexity.
It is always true that recursion can be replaced by iteration and stacks. It is also true that stack can be replaced by a recursive program with no stack.

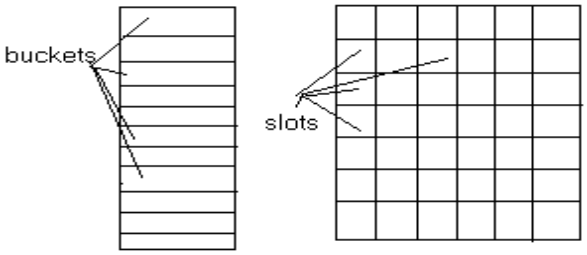


Fig 2.6

SOLVING RECURRENCES :-(Happen again (or) repeatedly)

- The indispensable last step when analyzing an algorithm is often to solve a recurrence equation.
- With a little experience and intention, most recurrence can be solved by intelligent guesswork.
- However, there exists a powerful technique that can be used to solve certain classes of recurrence almost automatically.
- This is a main topic of this section the technique of the characteristic equation.

1. Intelligent guess work:

This approach generally proceeds in 4 stages.

1. Calculate the first few values of the recurrence
2. Look for regularity.
3. Guess a suitable general form.
4. And finally prove by mathematical induction(perhaps constructive induction).

Then this form is correct.
Consider the following recurrence,

$$T(n) = \begin{cases} 0 & \text{if } n=0 \\ 3T(n \div 2)+n & \text{otherwise} \end{cases}$$

- First step is to replace n ÷ 2 by n/2
- It is tempting to restrict 'n' to being ever since in that case n÷2 = n/2, but recursively dividing an even no. by 2, may produce an odd no. larger than 1.
- Therefore, it is a better idea to restrict 'n' to being an exact power of 2.
- First, we tabulate the value of the recurrence on the first few powers of 2.

probabilistic analysis of algorithms :

is an approach to estimate the computational complexity of algorithm or a computational problem. It starts from an assumption about a probabilistic distribution of the set of all possible inputs. This assumption is then used to design an efficient algorithm or to derive the complexity of a known algorithm.
In probabilistic analysis of probabilistic (randomized) algorithms, the distributions or averaging for all possible choices in randomized steps are also taken into an account, in addition to the input distributions.

amortized analysis :

is a method of analyzing algorithms that considers the entire sequence of operations of the program. It allows for the establishment of a worst-case bound for the performance of an algorithm irrespective of the inputs by looking at all of the operations. This analysis is most commonly discussed using big O notation.

At the heart of the method is the idea that while certain operations may be extremely costly in resources, they cannot occur at a high enough frequency to weigh down the entire program because the number of less costly operations will far outnumber the costly ones in the long run, "paying back" the program over a number of iterations.[1] It is particularly useful because it guarantees worst-case performance rather than making assumptions about the state of the program.

There are generally three methods for performing amortized analysis: the aggregate method, the accounting method, and the potential method. All of these give the same answers, and their usage difference is primarily circumstantial and due to individual preference.

- Aggregate analysis determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the amortized cost to be $T(n) / n$.

- The accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. Usually, many short-running operations accumulate a "debt" of unfavorable state in small increments, while rare long-running operations decrease it drastically.
- The potential method is like the accounting method, but overcharges operations early to compensate for undercharges later

Disjoint Sets

Disjoint-set data structure is a data structurethat keeps track of a set of elements partitioned into a number of disjoint (nonoverlapping) subsets.

A **union-find algorithm**is an algorithm that performs two useful operations on such a data structure:

- *Find*: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.
- *Union*: Join two subsets into a single subset.

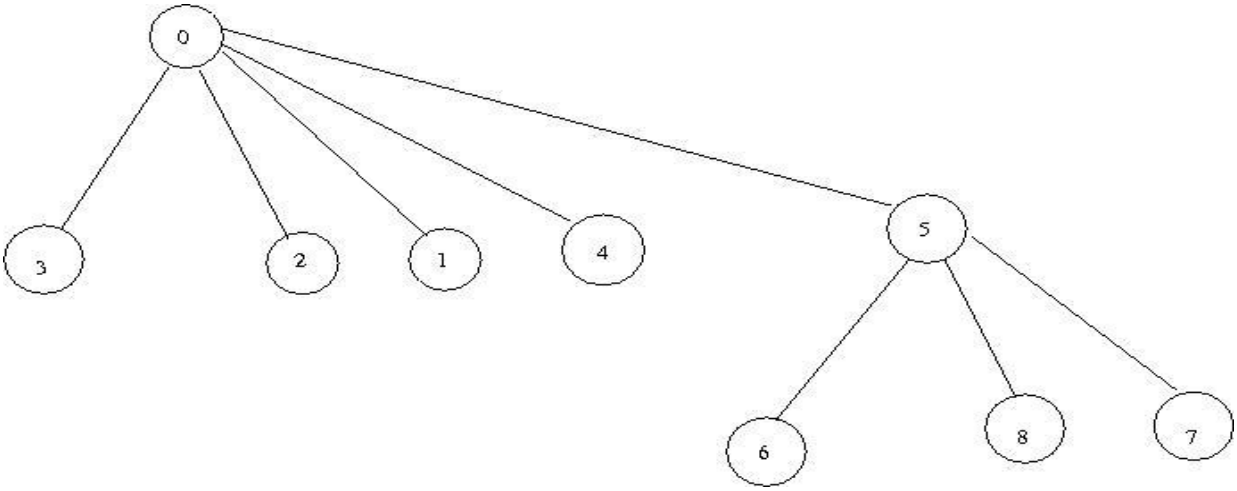
Because it supports these two operations, a disjoint-set data structure is sometimes called a *union-find data structure* or *merge-find set*. The other important operation, *MakeSet*, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved

Equivalence Relations

A binary relation R over a set S is called an
equivalence relation if it has following properties
1. Reexivity: for all element x, xRx
2. Symmetry: for all elements x and y, xRy if and only if yRx
3. Transitivity: for all elements x, y and z,if xRy and yRz then zRz
The relation \is related to" is an equivalence relation over the set of people

S1= {0,1,2,3,4}; S2 = { 5,6,7,8} and SJ ={3, 4, 6}.
The operations we wish to perform on these sets are:
(a) **Disjoint set union** ... if S; and Sj are two disjoint sets, then their union S; U S j = { all elements x such that x is in S; or S j}. Thus, S 1 U S 2 = {0,1,2,3,4,5,6,7,8}. Since we have assumed that all sets are disjoint, following the union of S; and Sj we can assume that the sets S; and Sj no longer exist independently, i.e. they are replaced by S; U Sj in the collection of sets.
(b) **Find (i) ...** find the set containing element i. Thus, 4 is in set S1 and 8 is in set S 2.

S1 U S2 in the tree format



Simple Union Method

```
void SimpleUnion(int i, int j)
{parent[i] = j;}
```

The time complexity O(1)

Simple Find Method

```
int SimpleFind(int i)
{
while (parent[i] >= 0)
i = parent[i]; // move up the tree
return i;
}
```

Time Complexity of SimpleFind()

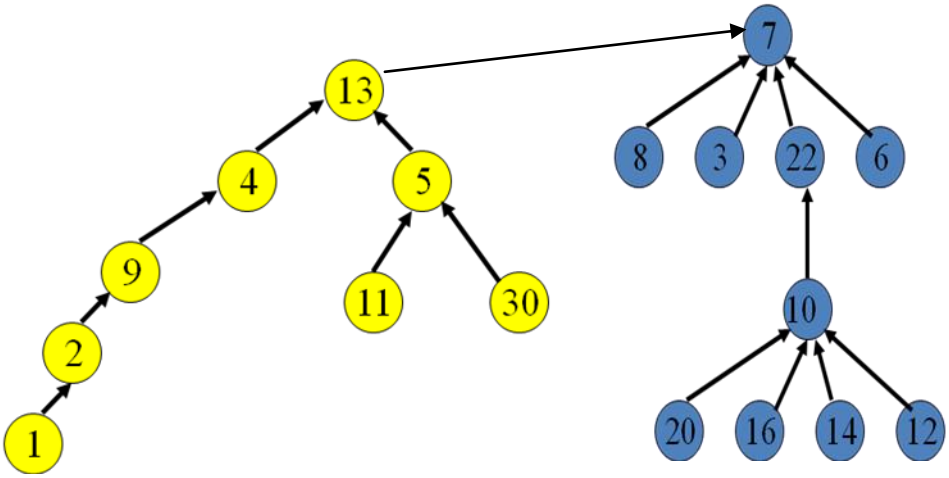
• Tree height may equal number of elements n in tree.

Union(2,1), Union(3,2), Union(4,3), Union(5,4)…

So complexity is O(n).

Weighted Rule

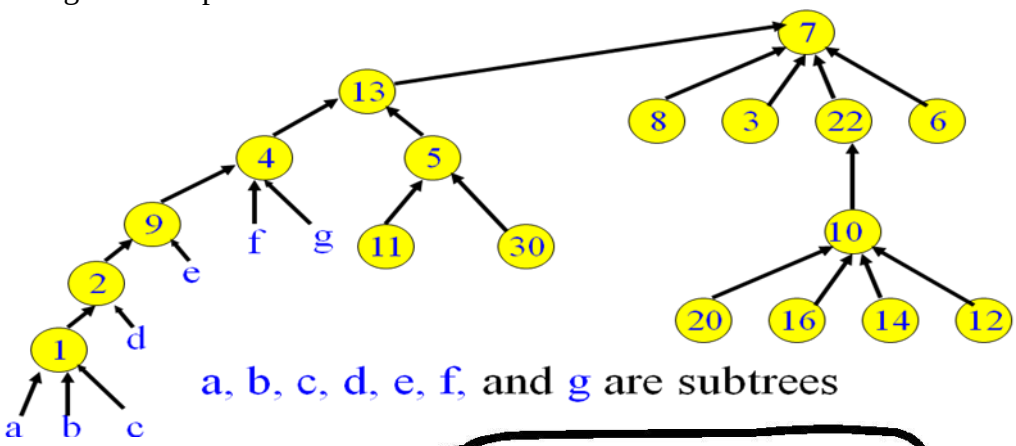
Make tree with fewer number of elements a subtree of the other tree.

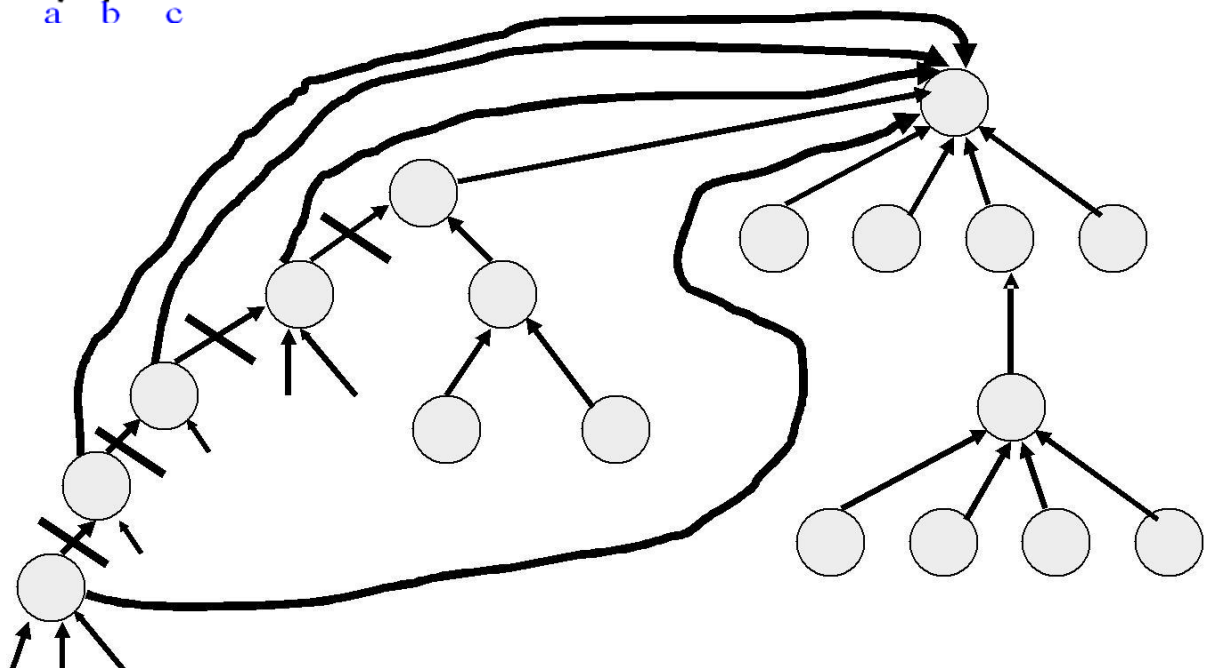


- Root of each tree must record either its height or the number of elements in the tree.
- When a union is done using the height rule, the height increases only when two trees of equal height are united.
- When the weight rule is used, the weight of the new tree is the sum of the weights of the trees that are united.

Algorithm WIGHTEDUNION(i j)
//union sets with roots i andj, i ;it. j, using the weighting rule./ /
//PARENT(i) = -COUNT(i) and PARENT(;) = - COUNT(j)./ /
integer i,j,x
x <-- PARENT(i) + PARENT(j)
if. PARENT(i) > PARENT(j)
then PARENT(i) <-- j //i has fewer nodes//
PARENT(j) <-- x
else PARENT(j) <-- i // j has fewer nodes/ /
PARENT(i) - x
endif
end WIGHTEDUNION

Collapsing Rule: If j is a node on the path from i to its root then set PARENT(j) - root (i). The new algorithm is procedure FIND

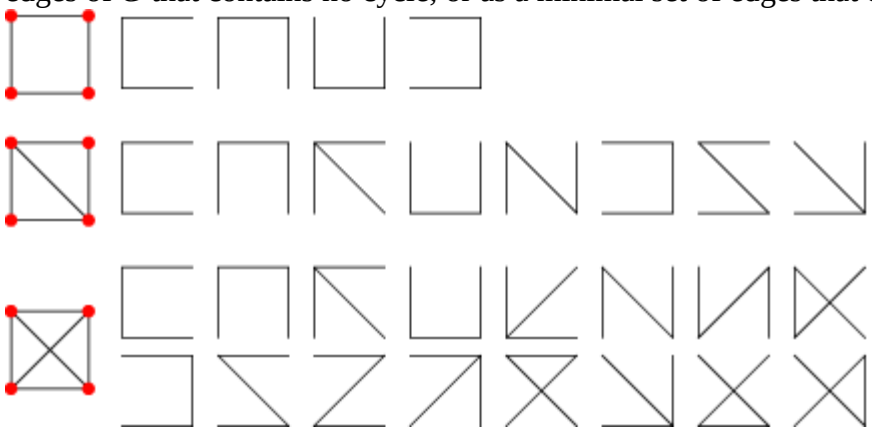




Algorithm COLLAPSINGFIND(i)
//Find the root of the tree containing element i. Use the/ I
//collapsing rule to collapse all nodes from i to the rootj//
j <-- i
while PARENT(j) > 0 do / /find root/ /
j <-- PARENT(j)
repeat
k <-- i
while k ;it. j do //collapse nodes from i to rootj//
t <-- PARENT(k)
PARENT(k) → j
k - t
repeat
retum(j)
end COLLAPSINGFIND

Spanning Trees

A tree is a connected undirected graph with no cycles. It is a spanning tree of a graph *G* if it spans *G* (that is, it includes every vertex of *G*) and is a sub-graph of *G* (every edge in the tree belongs to *G*). A spanning tree of a connected graph *G* can also be defined as a maximal set of edges of *G* that contains no cycle, or as a minimal set of edges that connect all vertices



Breadth-First Search Algorithmis a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.

Pseudocode

```
1  procedure BFS(G,v) is
2      create a queue Q
3      create a set V
4      enqueue v onto Q
5      add v to V
6      while Q is not empty loop
```

```
7           t ← Q.dequeue()
8           if t is what we are looking for then
9               return t
10          end if
11          for all edges e in G.adjacentEdges(t) loop
12              u ← G.adjacentVertex(t,e)
13              if u is not in V then
14                  add u to V
15                  enqueue u onto Q
16              end if
17          end loop
18      end loop
19      return none
20  end BFS
```

Space complexity

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as $O(|V|)$ where $|V|$ is the cardinality of the set of vertices. If the graph is represented by an Adjacency list it occupies $\Theta(|V| + |E|)$ space in memory, while an Adjacency matrix representation occupies $\Theta(|V|^2)$.

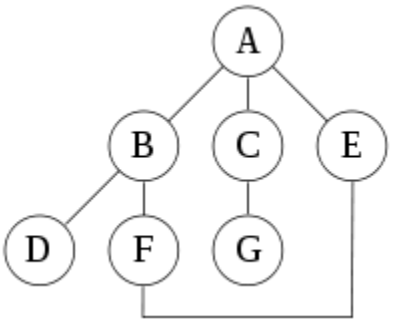
Time complexity

The time complexity can be expressed as $O(|V| + |E|)$ since every vertex and every edge will be explored in the worst case. Note: $O(|E|)$ may vary between $O(|V|)$ and $O(|V|^2)$, depending on how sparse the input graph is (assuming that the graph is connected).

Depth First Search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.Key difference between BFS and DFS is the order *discovered* (adjacent) vertices are explored.

BFS places discovered vertices in FIFO *queue*, exploring vertices in the order discovered.
DFS places discovered vertices in LIFO *stack*, exploring vertices as discovered.
For the following graph:



a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges traversed in this search form a Trémaux tree, a structure with important applications in graph theory.

Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

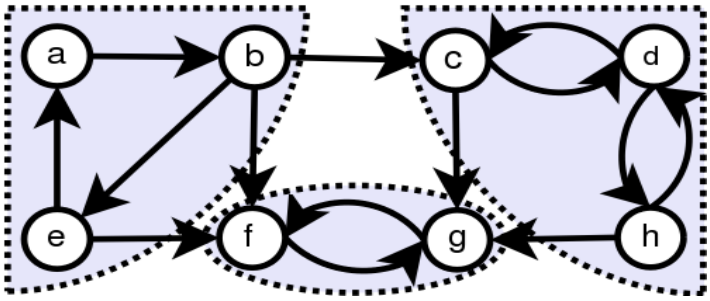
Iterative deepening is one technique to avoid this infinite loop and would reach all nodes.

PSEUDOCODE

```
1  procedure DFS(G,v):
2      label v as discovered
3      for all edges from v to w in G.adjacentEdges(v) do
4          if vertex w is not labeled as discovered then
5              recursively call DFS(G,w)
```

Connected component

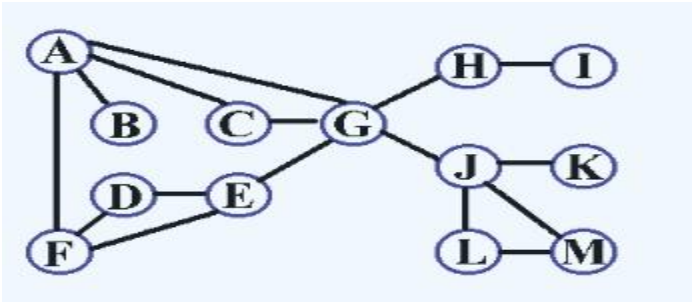
In graph theory, a **connected component** (or just **component**) of an undirected graph is a subgraph in which any two vertices are connected to each other bypaths, and which is connected to no additional vertices in the super graph. For example, the graph shown in the illustration on the right has three connected components. A graph that is itself connected has exactly one connected component, consisting of the whole graph
a graph is said to be **strongly connected** if every vertex is reachable from every other vertex. The **strongly connected components** of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. It is possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time



Articulation Point

An articulation point of a graph is a vertex v such that if v and its incident edges are removed, a connected component of the graph is broken into two or more pieces
a connected component with no articulation points is said to be biconnected
the dfs can be used to help find the biconnected components of a graph (how?)
finding articulation points is one problem concerning the connectivity of graphs

- ***Biconnected graph:*** A graph with no articulation point called biconnected. In other words, a graph is biconnected if and only if any vertex is deleted, the graph remains connected.
- ***Biconnected component:*** A biconnected component of a graph is a maximal biconnected subgraph- a biconnected subgraph that is not properly contained in a larger biconnected subgraph.
- A graph that is not biconnected can divide into biconnected components, sets of nodes mutually accessible via two distinct paths.



- **Articulation points: A, H, G, J**
- **Biconnected components: {A, C, G, D, E, F}、{G, J, L, B}、B、H、I、K**