UNIT – I

Analysis of Algorithm:

INTRODUCTION – ANALYZING CONTROL STRUCTURES-AVERAGE CASE ANALYSIS-SOLVING RECURRENCES.

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.

Selection Sort:

- Suppose we Must devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
- A Simple solution given by the following.
- (From those elements that are currently unsorted ,find the smallest & place it next in the sorted list.)

Algorithm:

```
1. For i:= 1 to n do
2. {
3.              Examine a[I] to a[n] and suppose the smallest element is at a[j];
4.              Interchange a[I] and a[j];
5.  }
```

→ Finding the smallest element (sat a[j]) and interchanging it with a[i]

- We can solve the latter problem using the code,

```
t   := a[i];
a[i]:=a[j];
a[j]:=t;
```

- The first subtask can be solved by assuming the minimum is a[I];checking a[I] with a[I+1],a[I+2]…….,and whenever a smaller element is found, regarding it as the new minimum. a[n] is compared with the current minimum.

- Putting all these observations together, we get the algorithm Selection sort.

Theorem:

Algorithm selection sort(a,n) correctly sorts a set of n>=1 elements .The result remains is a a[1:n] such that a[1] <= a[2] ….<=a[n].

Selection Sort:

Selection Sort begins by finding the least element in the list. This element is moved to the front. Then the least element among the remaining element is found out and put into second position. This procedure is repeated till the entire list has been studied.

Example:

LIST L = 3,5,4,1,2

1 is selected , → 1,5,4,3,2
2 is selected, →1,2,4,3,5
3 is selected, →1,2,3,4,5
4 is selected, →1,2,3,4,5

Proof:

- We first note that any I, say I=q, following the execution of lines 6 to 9,it is the case that a[q] Þ a[r],q<r<=n.
- Also observe that when 'i' becomes greater than q, a[1:q] is unchanged. Hence, following the last execution of these lines (i.e. I=n).We have a[1] <= a[2] <=……a[n].
- We observe this point that the upper limit of the for loop in the line 4 can be changed to n-1 without damaging the correctness of the algorithm.

Algorithm:

```
1. Algorithm selection sort (a,n)
2. // Sort the array a[1:n] into non-decreasing order.
3.{
4.      for I:=1 to n do
5.      {
6.              j:=I;
7.              for k:=i+1 to n do
8.                      if (a[k]<a[j])
9.                      t:=a[I];
10.                     a[I]:=a[j];
11.                     a[j]:=t;
```

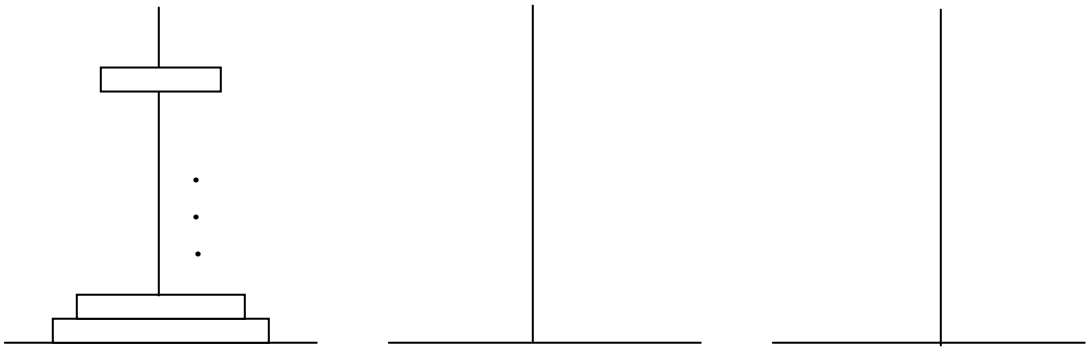
```
12.     }
13. }
```

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)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	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.

ELEMENTARY DATA STRUCTURES
SETS AND DISJOINT SET UNION

Introduction

In this section we study the use of forests in the representation of sets. We shall assume that the elements of the sets are the numbers 1, 2, 3, ..., n. These numbers might, in practice, be indices into a symbol table in which the names of the elements are stored. We assume that the sets being represented are pairwise disjoint (that is, if S_i and S_j, $i \neq j$, are two sets, then there is no element that is in both S_i and S_j). 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 $S_3 = \{3, 4, 6\}$. Fig shows one possible representation for these sets. In this representation, each set is represented as a tree. Notice that for each set we have linked the nodes from the children to the parent, rather than our usual method of linking from the parent to the children. The reason for this change in linkage becomes apparent when we discuss the implementation of set operations.

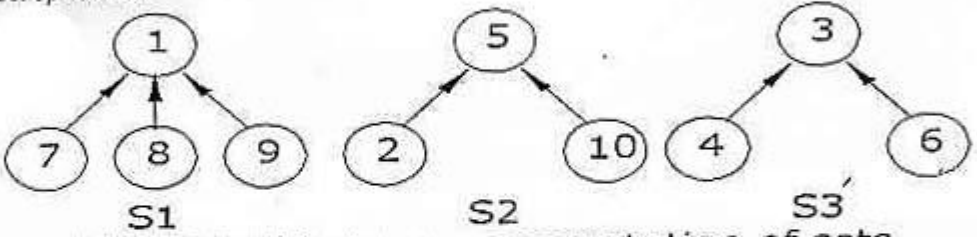


Fig: Possible tree representation of sets

The operations we wish to perform on these sets are:

1. Disjoint set union. If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = all elements x such that x is in S_i or S_j. Thus, $S_i \cup S_j = \{1, 7, 8, 9, 2, 5, 10\}$. Since we have assumed that all sets are disjoint, we can assume that following the union of S_i and S_j, the sets S_i and S_j do not exist independently; that is, they are replaced by $S_i \cup S_j$ in the collection of sets.
2. **Find(i).** Given the element i, find the set containing i. Thus, 4 is in set S_3, and 9 is in set S_1.

Union and Find Operations

Let us consider the union operation first. Suppose that we wish to obtain the union of S_1 and S_2. Since we have linked the node from children to parent, we simply make one of the trees a sub-tree of the other. $S_1 \cup S_2$ could have one of the representations of the following figures.

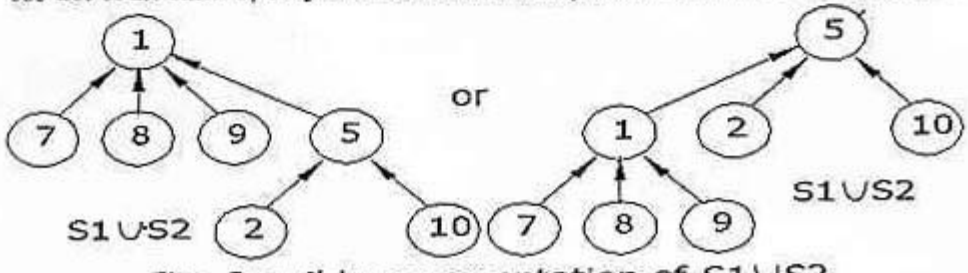


Fig: Possible representation of S1∪S2

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 easily if, with each set name, we keep a pointer to the root of the tree representing that set. If, 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. The data representation for S_1, S_2 and S_3 may then take the form shown in Figure.

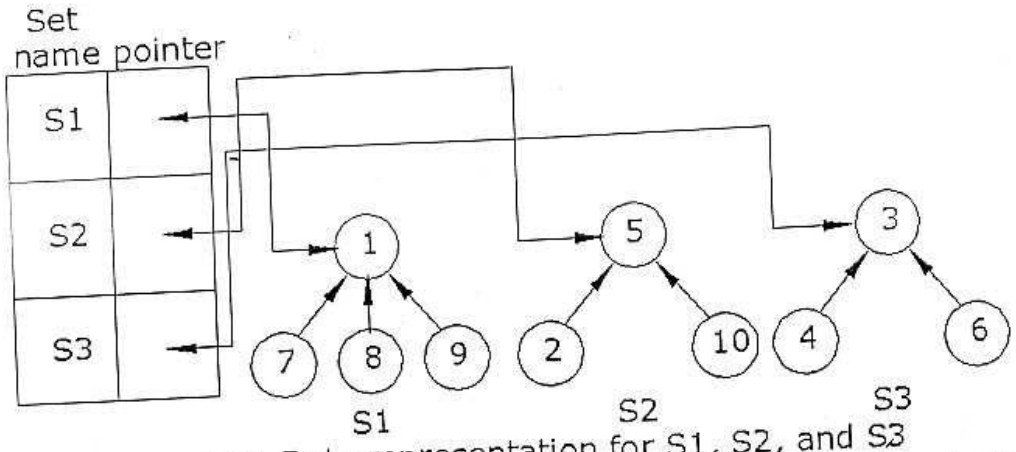


Fig: Data representation for S1, S2, and S3

In presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them. This simplifies the discussion. If we wish to unite sets S_i and S_j, then we wish to unite the trees with roots FindPointer(S_i) and FindPointer(S_j). 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. Also to simplify, assume that the set elements are the numbers 1 through n.

Since the set elements are numbered 1 through n, we represent the tree nodes using an array p[1 : n], where n is the maximum number of elements. The ith element of this array represents the tree node that contains element i. This array element gives the parent pointer of the corresponding tree node. Following figure shows this representation of the sets S_1, S_2 and S_3. Notice that 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

Fig: Array representation of S1, S2 and S3

Algorithm: Simple algorithms for union-and find are

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

```
1  Algorithm SimpleFind(i)
2  {
3     while (p[i] ≥ 0) do i := p[i];
4     return i;
5  }
```

These two algorithms are very easy to state, 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, $S_i = \{i\}$, $1 \le i \le q$), then the initial configuration consists of forest with q nodes, and $p[i] = 0$, $1 \le i \le q$. Now let us process the following sequence of union-find operations:

Union(1,2), Union(2,3), Union(3,4), Union(4,5), …., Union(n - 1, n)

Find(1), Find(2), Find(3), …., Find(n).

This sequence results in the degenerate tree of following figure.

Degenerate tree

Since the time taken for a union is constant, the n - 1 unions can be processed in time O(n). However, 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

$$O\left(\sum_{i=1}^{n} i\right) = O(n^2).$$

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).

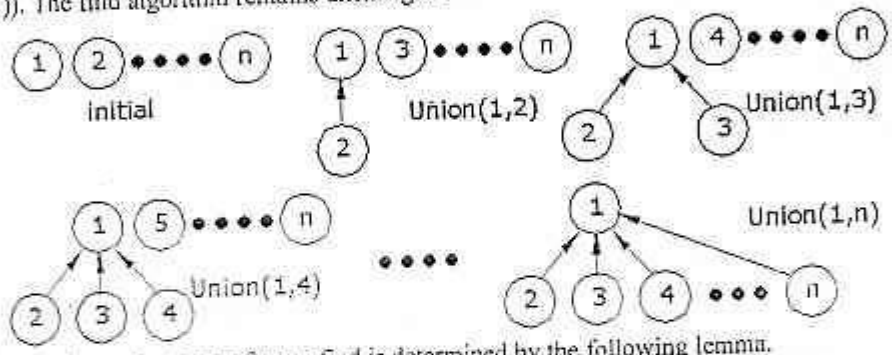
Sets & Disjoint Set - Union

Definition [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.

When we use the weighting rule to perform the sequence of set unions given before, we obtain the trees of following figure. In this figure, the unions have been modified so that the input parameter values correspond to the roots of the trees to be combined.

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.

Using this convention, we obtain following algorithm. In this algorithm the time required to perform a union has increased somewhat but is still bounded by a constant (that is, it is O(1)). The find algorithm remains unchanged.



The maximum time to perform a find is determined by the following lemma.

```
1  Algorithm WeightedUnion(i,j)
2  // Union sets with roots i and j, i ≠ j, using the
3  // weighting rule. p[i] = -count[i] and p[j] = -count[j].
4  {
5     temp := p[i] + p[j];
6     if (p[i] > p[j]) then
7       {// i has fewer nodes.
8          p[i] := j; p[j] := temp;
9       }
10    else
11      {// j has fewer or equal nodes.
12         p[j] := i; p[i] := temp;
13      }
14 }
```

Algorithm: Union algorithm with weighting rule

Lemma: Assume that we start with a forest of trees, each having one node. Let T be a tree with m nodes created as a result of a sequence of unions each performed using WeightedUnion. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$.

Proof: The lemma is clearly true for m = 1. Assume it is true for all trees with i nodes, $i \leq m - 1$. We show that it is also true for i = m. 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 the number in k. Without loss of generality we can assume $1 \leq a \leq m/2$. Then the height of T is either the same as that of k or is one more than that of j. If the former is the case, the height of T is $\leq \lfloor \log_2(m-a) \rfloor + 1 \leq \lfloor \log_2 m \rfloor + 1$. However, if the latter is the case, the height of T is $\leq \lfloor \log_2 a \rfloor + 2 \leq \lfloor \log_2 m/2 \rfloor + 2 \leq \lfloor \log_2 m \rfloor + 1$.

Example: Consider the behavior of WeightedUnion on the following sequence of unions starting from the initial configuration p[i] = -count[i] = -1, $1 \leq i \leq 8 = n$:

Union(1,2), Union(3,4), Union(5,6), Union (7, 8), Union(1,3), Union(5,7), Union(1,5)

The trees of following figures are obtained. As is evident, the height of each tree with m nodes is $\lfloor \log_2 m \rfloor + 1$.

From Lemma, it follows that the time to process a find is O(log m) if there are m elements in a tree. If an intermixed sequence of u - 1 union and f find operations is to be processed, the time becomes O(u + f log u), as no tree has more than u nodes in it. Of course, we need O(n) additional time to initialize the n-tree forest.

Surprisingly, further improvement is possible. This time the modification is made in the find algorithm using the collapsing rule.

Definition [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].

CollapsingFind incorporates the collapsing rule.

```
1  Algorithm CollapsingFind(i)
2  // Find the root of the tree containing element i. Use the
3  // collapsing rule to collapse all nodes from i to the root.
4  {
5     r:= i;
6     while (p[r] > 0) do r := p[r]; // Find the root.
7     while (i ≠ r) do      // Collapse nodes from i to root r.
8     {
9        s:= p[i]; p[i] := r; i := s;
10    }
11    return r;
12 }
```

Algorithm: Find algorithm with collapsing rule

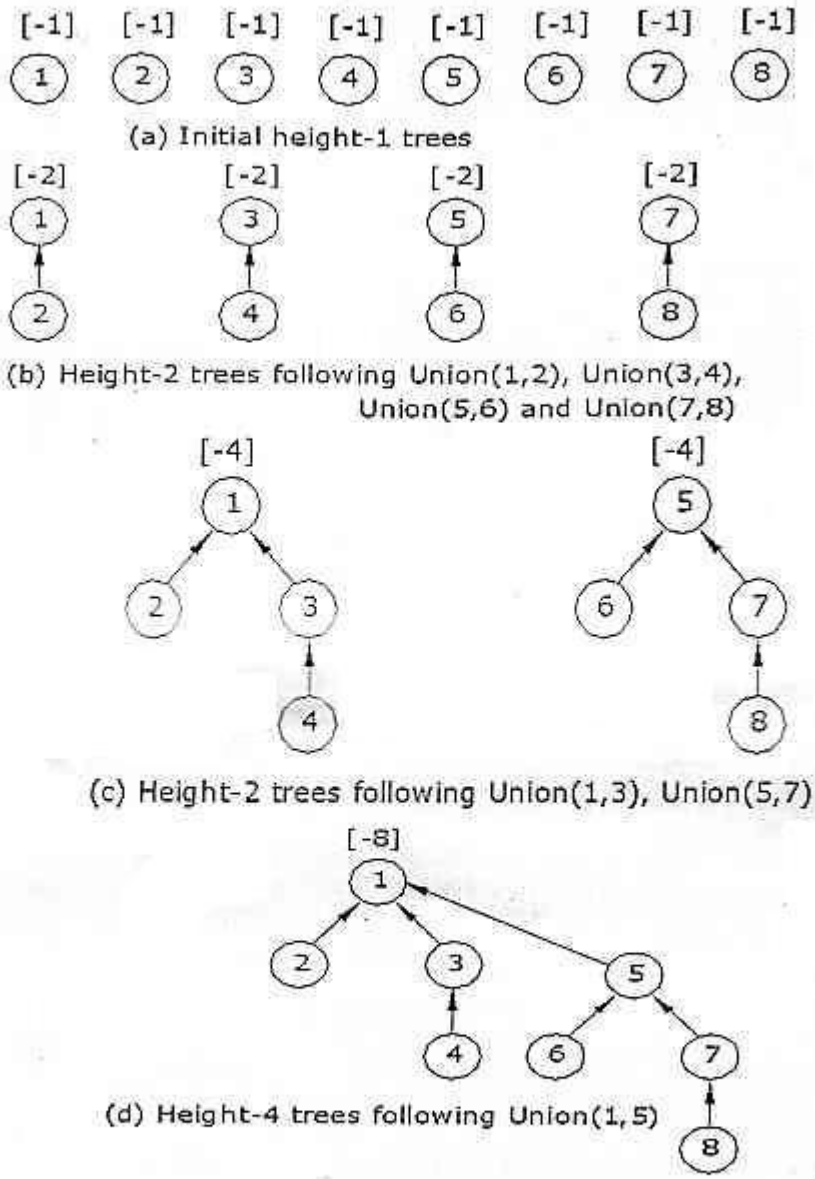


Fig: Trees achieving worst-case bound