In Worst-case (when the elements are in descending order), the time complexity is $O(n^2)$.
In Best-case (when the elements are in ascending order), the time complexity is O(n).
In Average-case, the time complexity is $O(n^2)$.

Disjoint Sets

Suppose we have some finite universe of n elements, U, out of which sets will be constructed. These sets may be empty or contain any subset of the elements of U. We shall assume that the elements of the sets are the numbers 1, 2, 3, …, n.

We assume that the sets being represented are pair wise disjoint, i.e., if S_i and S_j ($i \neq j$) are two sets, then there is no element that is in both $\mathbf{S_i}$ and $\mathbf{S_j}$.

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 $S_3 = \{3,4,6\}$.

→The following two operations are performed on the disjoint sets:

1) Union:

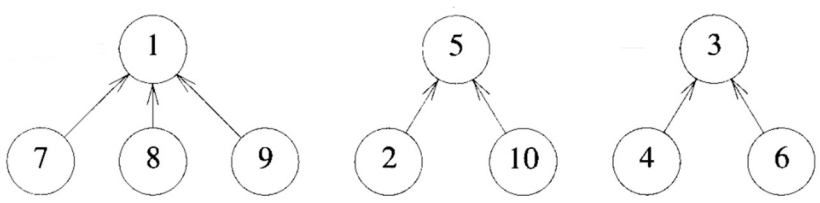
If $\mathbf{S_i}$ and $\mathbf{S_j}$ are two disjoint sets, then their union $\mathbf{S_i} \cup \mathbf{S_j}$ = {all elements **x** such that **x** is in $\mathbf{S_i}$ or $\mathbf{S_j}$}. Thus, in our example, $\mathbf{S_1} \cup \mathbf{S_2} = \{1, 7, 8, 9, 2, 5, 10\}$.

Since we have assumed that all sets are disjoint, we can assume that following the union of $\mathbf{S_i}$ and $\mathbf{S_j}$, the sets $\mathbf{S_i}$ and $\mathbf{S_j}$ do not exist independently; that is, they are replaced by $\mathbf{S_i} \cup \mathbf{S_j}$ in the collection of sets.

2) Find(i): Given the element i, find the set containing i.
Thus, in our example, 4 is in set S_3, and 9 is in set S_1.
So, Find(4) = S_3
Find(9) = S_1

To carry out these two operations efficiently, we represent each set by a tree.

One possible representation for the sets $S_1 = \{1,7, 8,9\}$, $S_2 = \{2,5,10\}$, and $S_3 = \{3,4,6\}$ using trees is given below:



Note: For each set, we have linked the nodes from the children to the parent.

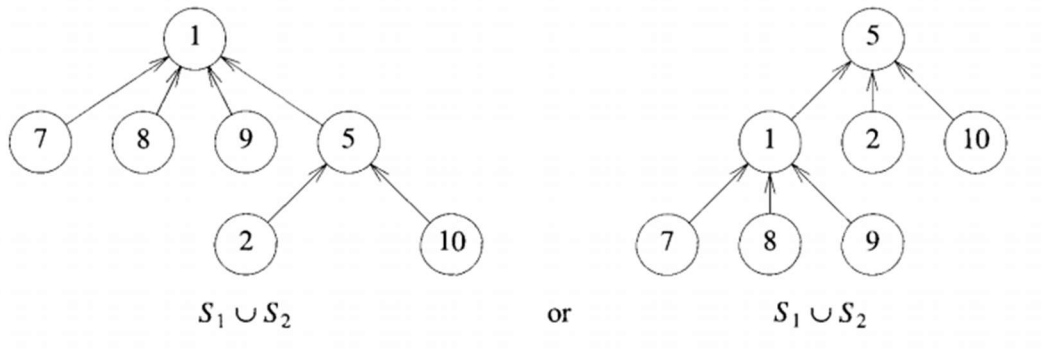
In presenting the UNION and FIND algorithms, we ignore the set names and identify sets just by the roots of the trees representing sets. This simplifies the discussion.

The operation of Find(i) now becomes:
Determine the root of the tree containing element 'i'.

Ex: Find(1)=1, Find(7)=1, Find(5)=5, Find(2)=5, Find(3)=3, find(6)=3, and so on.

To obtain the union of two sets, all that has to be done is to link one of the roots to the other root. The function Union(i, j) requires two trees with roots i and j to be joined.

The possible representations of $S_1 \cup S_2$:



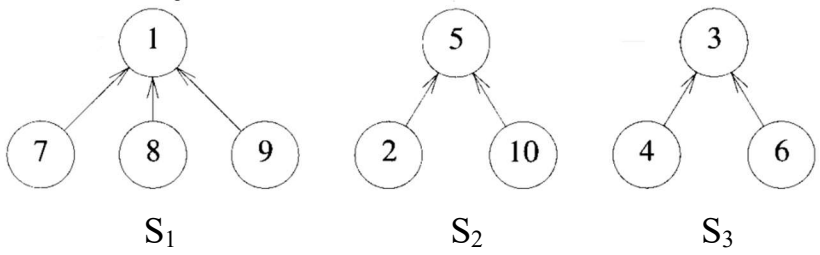
Representing the tree nodes of all disjoint sets using a single array:

Since the universal set elements are numbered 1 through n, we represent the tree nodes of all sets using an array P[1:n], where P stands for parent.

The i^{th} index of this array represents the tree node for element *i*. The array element at index *i* gives the parent of the corresponding tree node.

Note: We assume that the parent of root node of disjoint set tree is -1.

Ex: Suppose the tree representations of disjoint sets S_1, S_2 and S_3 are as follow:



S_1 S_2 S_3

Array representation of trees corresponding to sets S_1, S_2 and S_3:

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

→We can now implement **Find(i)** by following the indices starting at i until we reach a node with parent value -1.

Simple algorithm for **FIND(i)**

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

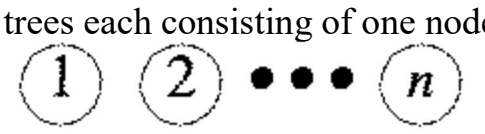
→The operation **Union(i, j)** is equally simple. Adopting the convention that the first tree becomes a subtree of the second tree (i.e., root of the first tree is linked to the root of the second tree), the statement P[i] := j accomplishes the Union.

Simple algorithm for **Union (i, j)**

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

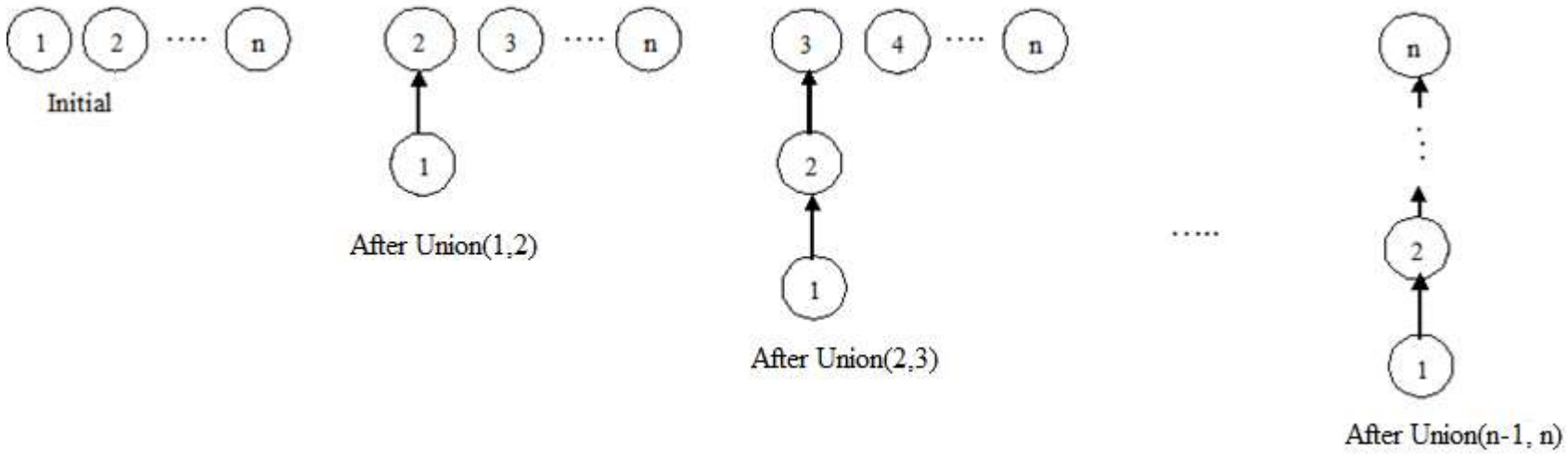
→Although these two algorithms are very easy to state, their performance characteristics are not very good.

For example, if we start off with 'n' elements each in a set of its own (that is, $S_i = \{i\}$, $1 \le i \le n$), then the initial configuration consists of a forest with 'n' trees each consisting of one node, and P[i] = -1, $1 \le i \le n$ as shown below:



i	1	2	…	n-1	N
P[i]	-1	-1	…	-1	-1

➔ Now imagine that we process the following sequence of UNION operations in the worst case: Union(1,2), Union(2,3), …, Union(n-1, n).



This sequence of union operations results in the **degenerate tree** as shown above.

The time taken for a union operation is constant and so, the n-1 Union operations can be processed in **O(n)** time.

→Now suppose we process the following sequence of FIND operations:

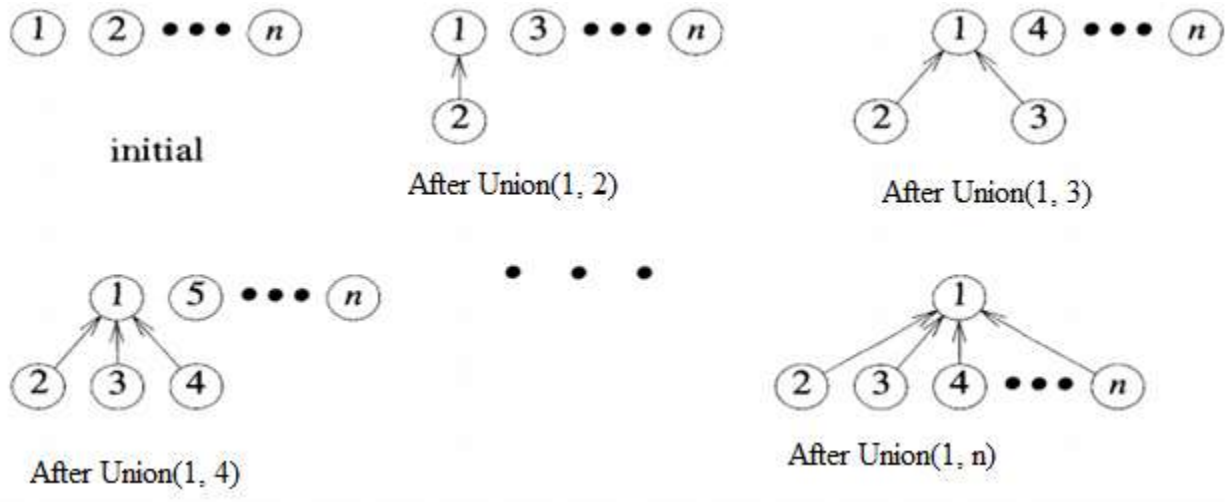
Find(1), Find(2), …, Find(n).

Each FIND requires following a chain of the parent links from the element to be found up to the root.
Since the time required to process a FIND for an element at level 'i' of the tree is O(i), the total time needed to process the 'n' FIND operations is $\sum_{i=1}^{n} i = \mathbf{O(n^2)}$.

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).

Weighting rule for Union(i, j):

If the number of nodes in the tree with root *i* is less than the number of nodes in the tree with root *j*, then make *'j'* as the parent of *'i'*; otherwise make *'i'* as the parent of *'j'*.
→When we use the weighting rule to perform the sequence of UNION operations Union(1,2), Union(1,3), …, Union(1,n), we obtain the trees as shown below:



To implement the weighting rule, we need to know how many nodes are there 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]* = number of nodes in the tree.

Since all nodes other than the roots of the trees have a positive number in their corresponding positions in the P[ ] array, we can maintain the negative of count of a tree in the corresponding position of its root in P[ ] array to distinguish root from other nodes.

Union algorithm with weighting rule :

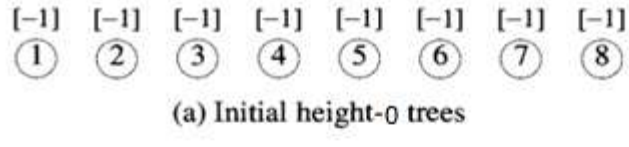
```
Algorithm WeightedUnion(i, j)
{
        // Unite sets with the roots i and j (i ≠ j) using weighting rule.
        // P[i] = -count[ i ] and P[ j ] = -count[ j ]
        temp := P[ i ] + P[ j ];
        if(P[ i ] > P[ j ]) then       // if tree 'i' has lesser number of nodes than tree 'j'
        {
                P[i ] := j;      // make i as subtree of j
                P[ j ] := temp;         //update count of  tree j
        }
        else   // if tree 'i' has more or same number of nodes than tree 'j'
        {
                P[ j ] := i;     // make j as subtree of  i
                P[ i ] := temp;         //update count of  tree i

        }
}
```

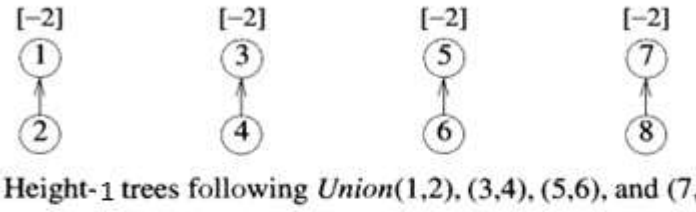
The time taken for WeightedUnion(i, j) is also constant, that is, **O(1)**.

Note:
Assume that we start off with a forest of trees, each having one node. Let *T* be a tree with *'n'* nodes created as a result of a sequence of UNION operations each performed using WeightedUnion. The height of *T* will not be more than$\lfloor \log_2 n \rfloor$. So, the worst-case time complexity of FIND is **O(logn)**.

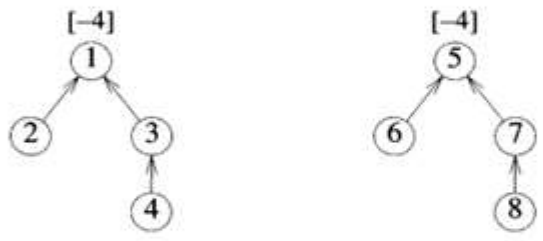
Example:
Consider the behavior of WeightedUnion on the following sequence of UNION operations starting from the initial configuration, P[i] = - count[i] = -1, 1≤i≤ 8 :
Union(1,2), Union(3,4), Union(5,6), Union(7,8), Union(1,3), Union(5,7), Union(1,5):



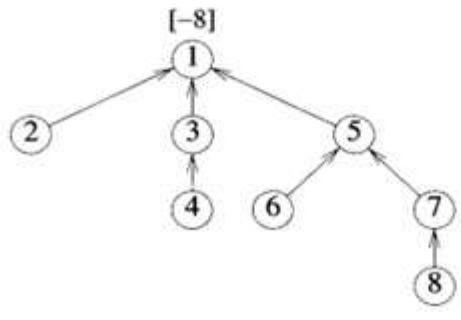
(a) Initial height-0 trees



(b) Height-1 trees following *Union*(1,2), (3,4), (5,6), and (7,8)



(c) Height-2 trees following *Union*(1,3) and (5,7)



(d) Height-3 tree following *Union*(1,5)

To further reduce the time taken over a sequence of FIND operations, we make the modifications in the FIND algorithm using the Collapsing Rule.

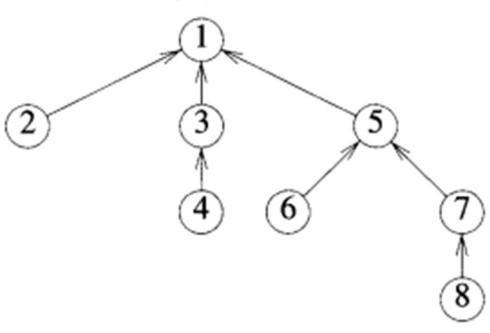
Collapsing Rule :

If 'j' is a node on the path from 'i' to its root 'r' and P[i] ≠ r then set P[j] = r.

Find algorithm with Collapsing Rule :

```
Algorithm CollapsingFind(i)
{
        // Find the root of the tree containing element i. Use the collapsing rule to collapse all nodes from i to root.
        r :=i;
        while(P[r] > 0) do  // find the root of the tree containing i.
                r := P[ r ];
        // At this point, r is the root of the tree containing i. Now collapsing has to done.
        while (i ≠ r) do
        {
                S := P[ i ] ;
                P[ i ] := r;     // link i to r directly
                i := S;
        }
        return r;
}
```

Example: Consider the following tree:

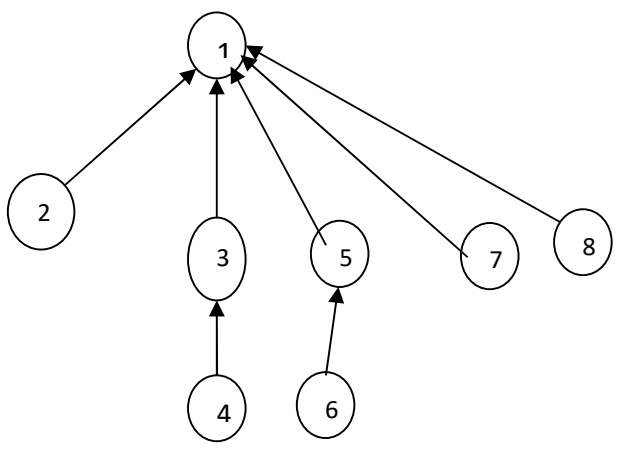


i	1	2	3	4	5	6	7	8
P [i]	-8	1	1	3	1	5	5	7

Now process the following eight FIND operations:
Find(8), Find(8), Find(8), Find(8), Find(8), Find(8), Find(8), Find(8).

If SimpleFind( ) is used, each Find(8) requires going up 3 parent link fields for a total of 24 moves to process all the eight FIND operations.

When CollapsingFind( ) is used, the first Find(8) requires going up 3 parent links and the resetting of 3 parent links. The tree after performing the first Find(8) operation will be as follows:

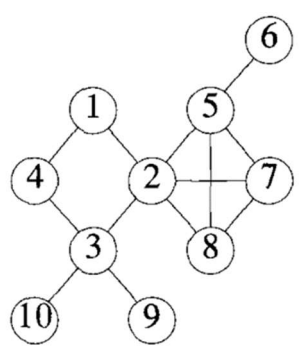


Each of the remaining seven Find(8) operations require going up only one parent link field. The total cost is now only 3+3+7=13 moves.

Articulation Points

A vertex V in a connected graph G is said to be 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 non-empty components.

Example: Consider the following connected graph G:

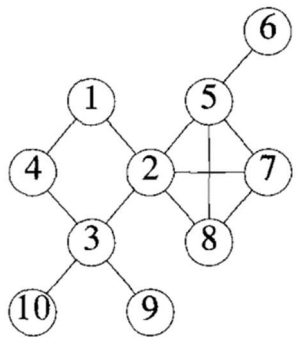


The articulation points in the graph G are: 2, 3 and 5.

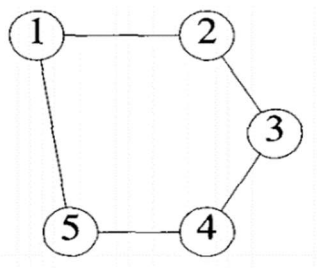
Biconnected Graph: A graph G is biconnected if and only if it contains no articulation points.

Examples:

1. The following graph is not a biconnected graph since it has articulation points.



2. The following graph is a Biconnected Graph since it doesn't have articulation points.



→The presence of articulation points in a connected graph can be undesirable feature in many cases.
For example, if G represents a communication network with the vertices representing communication stations and the edges representing communication lines, then the failure of a communication station i that is an articulation point would result in the loss of communication to other points also and makes the entire communication system down.

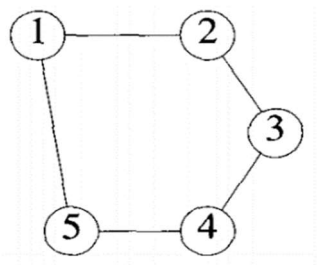
On the other hand, if G has no articulation point, then if any station i fails, we can still communicate between any two stations excluding station i.

Once it has been determined that a connected graph *G* is not biconnected, it may be desirable to determine a set of edges whose inclusion will make the graph biconnected. Determining such a set of edges is facilitated if we know the maximal subgraphs of *G* that are biconnected, (i.e., biconnected components of G).

Biconnected Components:

A biconnected component of a graph G is a maximal subgraph of G that is biconnected. That means, it is not contained in any larger subgraph of G that is biconnected.

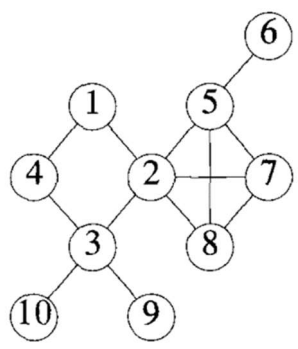
Ex: Consider the following biconnected graph:



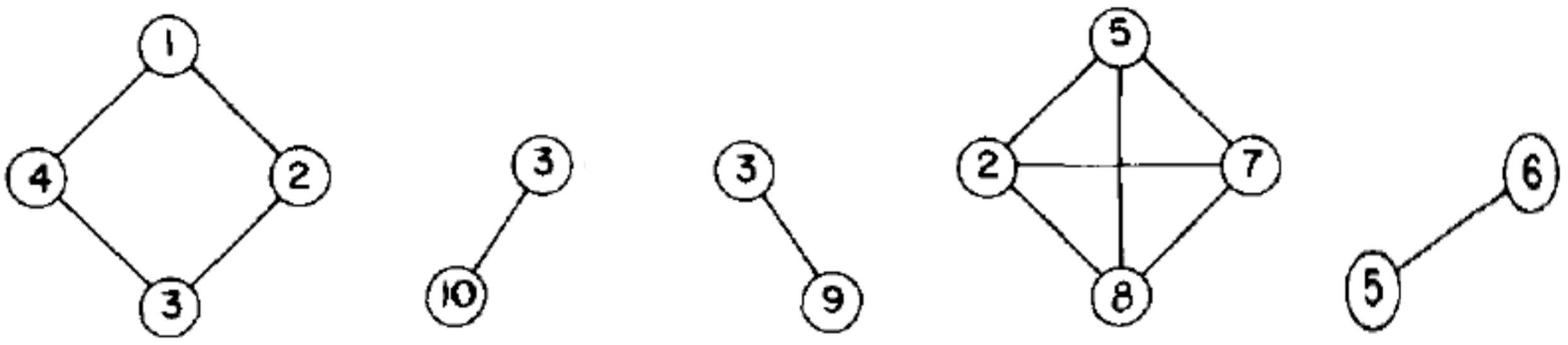
This graph has only one biconnected component (i.e., the entire graph itself).

So, a biconnected graph will have only one biconnected component, whereas a graph which is not biconnected consists of several biconnected components.

Ex: Consider the following graph which is not biconnected:



Biconnected components of the above graph are:



(i) (ii) (iii) (iv) (v)

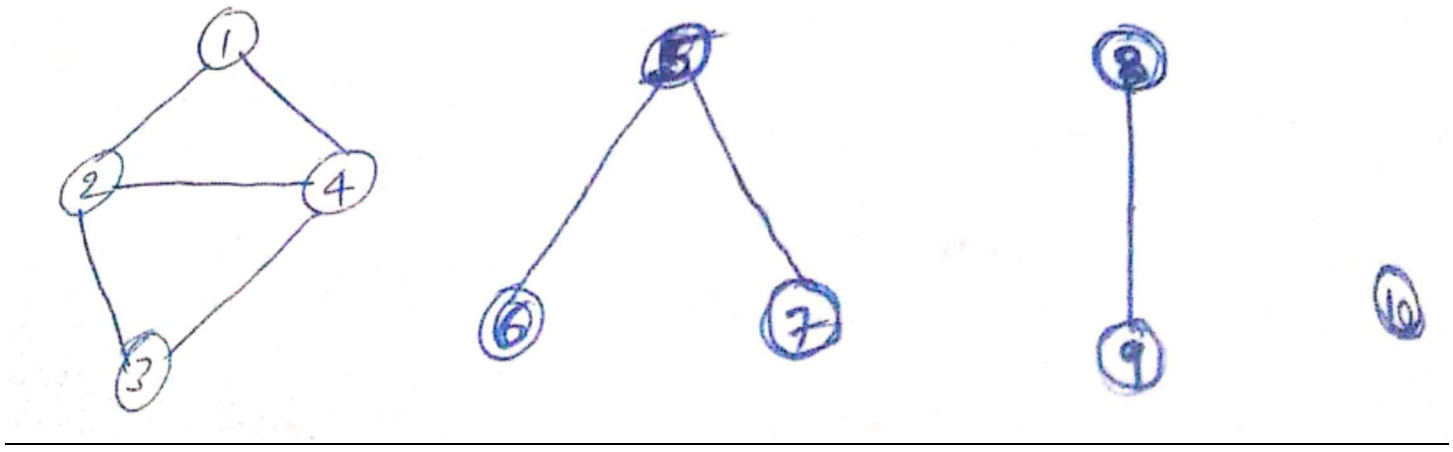
Note: Two biconnected components can have at most one vertex in common and this vertex is an articulation point.

Connected Components:

A connected component of a graph G is a maximal subgrph of G that is connected. That means, it is not contained in any larger subgraph of G that is connected.

A connected graph consists of just one connected component (i.e., the entire graph), whereas a disconnected graph consists of several connected components.

Ex: A disconnected graph of 10 vertices and 4 connected components:



Amortized Analysis

In an amortized analysis, we average the time required to perform a sequence of data-structure operations.
With amortized analysis, we can show that the average cost of any operation in the sequence is cheap although a single operation in the sequence might be expensive.

Amortized analysis applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure where the costs of those operations vary over a period of time.
It turns out that in some situations a single operation can be expensive, but the total time for an entire sequence of n such operations is always significantly lesser than the worst-case time complexity of that single operation multiplied by n.

There are three most common techniques used in amortized analysis:
Aggregate analysis
Accounting method
Potential method

(1) Aggregate analysis:

We determine an upper bound T(n) on the total cost of a sequence of n operations. The average cost per operation is then T(n)/n. We take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

$$\text{Amortized cost of each operation} = \frac{Sum\ of\ the\ actual\ costs\ of\ n\ operations\ in\ a\ sequence}{n}$$

(2) Accounting method:

The accounting method is aptly named because it borrows ideas and terms from accounting.

Here, each operation is charged a cost called the **amortized cost**. Some operations can be charged more or less than they actually cost. If an operation's amortized cost exceeds its actual cost, the surplus is added to a **credit** (which is like a bank balance). Credit can be used later to help pay for other operations whose amortized cost is less than their actual cost. Total credit can never be negative in any sequence of operations.

We must choose the amortized cost of each operation carefully.

Different types of operations (like *push* and *pop* in stack data structure or *insert* and *delete* in queue data structure) may have different amortized costs. This method differs from aggregate analysis wherein all types of operations have the same amortized cost.

We must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence of operations. That means, the total amortized cost of a sequence of operations must be greater than or equal to the total actual cost of the sequence of operations.

If we denote the actual cost of the i^{th} operation by c_i and the amortized cost of the i^{th} operation by $\hat{c}_i$, we require

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

for all sequences of n operations.