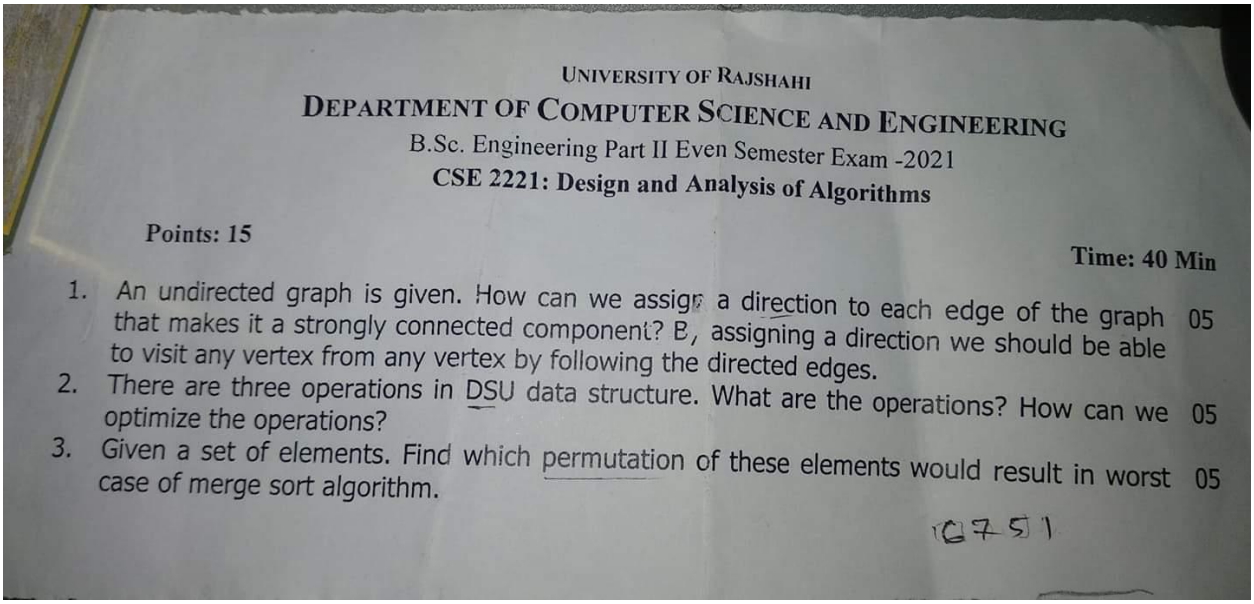


UNIVERSITY OF RAJSHAHI

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.Sc. Engineering Part II Even Semester Exam -2021

CSE 2221: Design and Analysis of Algorithms

Points: 15 **Time: 40 Min**

1. An undirected graph is given. How can we assign a direction to each edge of the graph that makes it a strongly connected component? By assigning a direction we should be able to visit any vertex from any vertex by following the directed edges. 05
2. There are three operations in DSU data structure. What are the operations? How can we optimize the operations? 05
3. Given a set of elements. Find which permutation of these elements would result in worst case of merge sort algorithm. 05

0751

1.
Convert undirected connected graph to strongly connected directed graph

Given an undirected graph of **N** vertices and **M** edges, the task is to assign directions to the given M Edges such that the graph becomes Strongly Connected Components. If a graph cannot be converted into Strongly Connected Components then print **"-1"**.

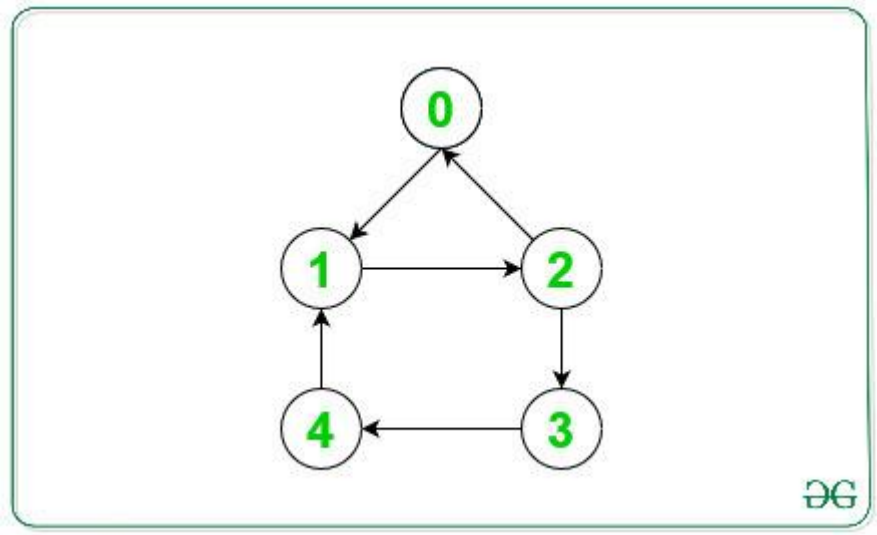
Examples:

Input: *N = 5, Edges[][] = { { 0, 1 }, { 0, 2 }, { 1, 2 }, { 1, 4 }, { 2, 3 }, { 3, 4 } }*

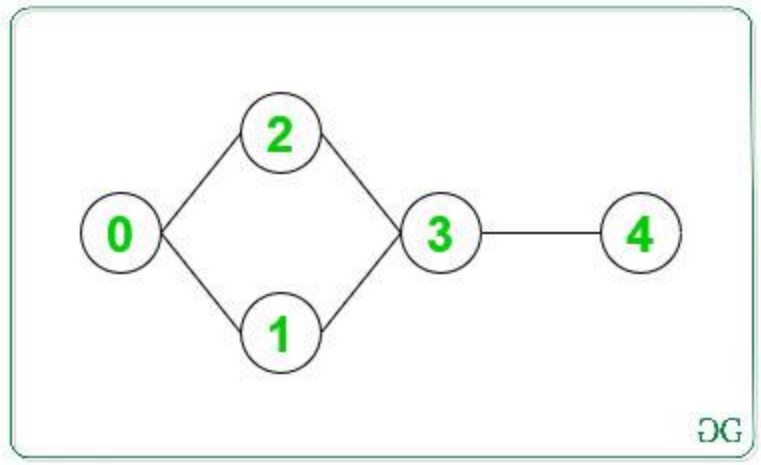
Output:

0->1
2->0
4->1
3->4
2->3
1->2

Explanation:
Below is the assigned edges to the above undirected graph:



Input: *N = 5, Edges[][] = { { 0, 1 }, { 0, 2 }, { 1, 3 }, { 2, 3 }, { 3, 4 } }*
Output: *-1*
Explanation:
Below is the graph for the above information:



Since there is a bridge present in the above-undirected graph. Therefore, this graph can't be converted into SCCs.

Approach: We know that in any directed graph is said to be in **Strongly Connected Components(SCCs)** iff all the vertices of the graph are a part of some cycle. The given undirected graph doesn't form SCCs if and only if the graph contains any bridges in it. Below are the steps:

- We will use an array **mark[]** to store the visited node during DFS Traversal, **order[]** to store the index number of the visited node, and **bridge_detect[]** to store any bridge present in the given graph.
- Start the DFS Traversal from vertex **1**.
- Traverse the Adjacency list of current Node and do the following:
    - If any edges are traverse again while any DFS call then ignore that edges.
    - If the order of child Node(**Node u**) is greater than the order of parent node(**node v**), then ignore this current edges as as **Edges(v, u)** is already processed.
    - If any Back Edge is found then update the Bridge Edges of the current parent node(**node v**) as:

```
bridge_detect[v] = min(order[u], bridge_detect[v]);
```

- Else do the DFS Traversal for the current child node and repeat step 3 for the current node.
- Update the bridges detect after DFS call for the current node as:

```
bridge_detect[v] = min(bridge_detect[u], bridge_detect[v])
```

- Store the current pair of **Edges(v, u)** as directed Edges from Node v to Node u in an array of pairs(say **arr[][]**).
- If there is any bridge present in the given graph then print **"-1"**.
- Else print the directed Edges stored in **arr[][]**.

Time Complexity: O(N)
Auxiliary Space: O(N)

https://www.geeksforgeeks.org/convert-undirected-connected-graph-to-strongly-connected-directed-graph/

2. Answer:

Operations

Disjoint-set data structures support three operations: Making a new set containing a new element; Finding the representative of the set containing a given element; and Merging two sets.

Making new sets

The `MakeSet` operation adds a new element into a new set containing only the new element, and the new set is added to the data structure. If the data structure is instead viewed as a partition of a set, then the `MakeSet` operation enlarges the set by adding the new element, and it extends the existing partition by putting the new element into a new subset containing only the new element.

In a disjoint-set forest, `MakeSet` initializes the node's parent pointer and the node's size or rank. If a root is represented by a node that points to itself, then adding an element can be described using the following pseudocode:

```
function MakeSet(x) is
    if x is not already in the forest then
        x.parent := x
        x.size := 1     // if nodes store size
        x.rank := 0     // if nodes store rank
    end if
end function
```

This operation has constant time complexity. In particular, initializing a disjoint-set forest with n nodes requires $O(n)$ time.

In practice, `MakeSet` must be preceded by an operation that allocates memory to hold X. As long as memory allocation is an amortized constant-time operation, as it is for a good dynamic array implementation, it does not change the asymptotic performance of the random-set forest.

Finding set representatives

The `Find` operation follows the chain of parent pointers from a specified query node x until it reaches a root element. This root element represents the set to which x belongs and may be x itself. `Find` returns the root element it reaches.

Performing a `Find` operation presents an important opportunity for improving the forest. The time in a `Find` operation is spent chasing parent pointers, so a flatter tree

leads to faster `Find` operations. When a `Find` is executed, there is no faster way to reach the root than by following each parent pointer in succession. However, the parent pointers visited during this search can be updated to point closer to the root. Because every element visited on the way to a root is part of the same set, this does not change the sets stored in the forest. But it makes future `Find` operations faster, not only for the nodes between the query node and the root, but also for their descendants. This updating is an important part of the disjoint-set forest's amortized performance guarantee.

There are several algorithms for `Find` that achieve the asymptotically optimal time complexity. One family of algorithms, known as **path compression**, makes every node between the query node and the root point to the root. Path compression can be implemented using a simple recursion as follows:

```
function Find(x) is
    if x.parent ≠ x then
        x.parent := Find(x.parent)
        return x.parent
    else
        return x
    end if
end function
```

This implementation makes two passes, one up the tree and one back down. It requires enough scratch memory to store the path from the query node to the root (in the above pseudocode, the path is implicitly represented using the call stack). This can be decreased to a constant amount memory by performing both passes in the same direction. The constant memory implementation walks from the query node to the root twice, once to find the root and once to update pointers:

```
function Find(x) is
    root := x
    while root.parent ≠ root do
        root := root.parent
    end while

    while x.parent ≠ root do
        parent := x.parent
        x.parent := root
        x := parent
    end while

    return root
end function
```

Tarjan and Van Leeuwen also developed one-pass `Find` algorithms that retain the same worst-case complexity but are more efficient in practice.[4] These are called path splitting and path halving. Both of these update the parent pointers of nodes on the path between the query node and the root. **Path splitting** replaces every parent pointer on that path by a pointer to the node's grandparent:

```
function Find(x) is
    while x.parent ≠ x do
        (x, x.parent) := (x.parent, x.parent.parent)
    end while
    return x
end function
```

Path halving works similarly but replaces only every other parent pointer:

```
function Find(x) is
    while x.parent ≠ x do
        x.parent := x.parent.parent
        x := x.parent
    end while
    return x
end function
```

Merging two sets

The operation `Union(x, y)` replaces the set containing x and the set containing y with their union. `Union` first uses `Find` to determine the roots of the trees containing x and y. If the roots are the same, there is nothing more to do. Otherwise, the two trees must be merged. This is done by either setting the parent pointer of x's root to y's, or setting the parent pointer of y's root to x's.

The choice of which node becomes the parent has consequences for the complexity of future operations on the tree. If it is done carelessly, trees can become excessively tall. For example, suppose that `Union` always made the tree containing x a subtree of the tree containing y. Begin with a forest that has just been initialized with elements and execute $\text{Union}(1, 2)$, $\text{Union}(2, 3)$, ..., $\text{Union}(n - 1, n)$. The resulting forest contains a single tree whose root is n, and the path from 1 to n passes through every node in the tree. For this forest, the time to run `Find(1)` is $O(n)$.

In an efficient implementation, tree height is controlled using **union by size** or **union by rank**. Both of these require that a node store information besides just its parent pointer.

This information is used to decide which root becomes the new parent. Both strategies ensure that trees do not become too deep.

In the case of union by size, a node stores its size, which is simply its number of descendants (including the node itself). When the trees with roots x and y are merged, the node with more descendants becomes the parent. If the two nodes have the same number of descendants, then either one can become the parent. In both cases, the size of the new parent node is set to its new total number of descendants.

```
function Union(x, y) is
    // Replace nodes by roots
    x := Find(x)
    y := Find(y)

    if x = y then
        return  // x and y are already in the same set
    end if

    // If necessary, rename variables to ensure that
    // x has at least as many descendants as y
    if x.size < y.size then
        (x, y) := (y, x)
    end if

    // Make x the new root
    y.parent := x
    // Update the size of x
    x.size := x.size + y.size
end function
```

The number of bits necessary to store the size is clearly the number of bits necessary to store n. This adds a constant factor to the forest's required storage.

For union by rank, a node stores its *rank*, which is an upper bound for its height. When a node is initialized, its rank is set to zero. To merge trees with roots x and y, first compare their ranks. If the ranks are different, then the larger rank tree becomes the parent, and the ranks of x and y do not change. If the ranks are the same, then either one can become the parent, but the new parent's rank is incremented by one. While the rank of a node is clearly related to its height, storing ranks is more efficient than storing heights. The height of a node can change during a `Find` operation, so storing ranks avoids the extra effort of keeping the height correct. In pseudocode, union by rank is:

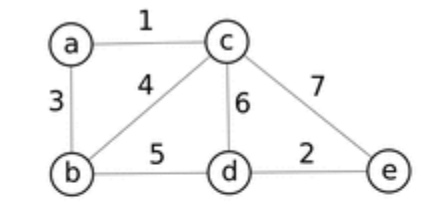
```
function Union(x, y) is
    // Replace nodes by roots
    x := Find(x)
```

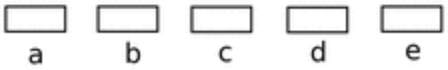
```
    y := Find(y)

    if x = y then
        return  // x and y are already in the same set
    end if

    // If necessary, rename variables to ensure that
    // x has rank at least as large as that of y
    if x.rank < y.rank then
        (x, y) := (y, x)
    end if

    // Make x the new root
    y.parent := x
    // If necessary, increment the rank of x
    if x.rank = y.rank then
        x.rank := x.rank + 1
    end if
end function
```

It can be shown that every node has rank or less.[10] Consequently the rank can be stored in $O(\log \log n)$ bits, making it an asymptotically negligible portion of the forest's size.

It is clear from the above implementations that the size and rank of a node do not matter unless a node is the root of a tree. Once a node becomes a child, its size and rank are never accessed again.

Applications





A demo for Union-Find when using Kruskal's algorithm to find minimum spanning tree.

Disjoint-set data structures model the partitioning of a set, for example to keep track of the connected components of an undirected graph. This model can then be used to determine whether two vertices belong to the same component, or whether adding an edge between them would result in a cycle. The Union–Find algorithm is used in high-performance implementations of unification.[15]

This data structure is used by the Boost Graph Library to implement its Incremental Connected Components functionality. It is also a key component in implementing Kruskal's algorithm to find the minimum spanning tree of a graph.

Note that the implementation as disjoint-set forests does not allow the deletion of edges, even without path compression or the rank heuristic.

Sharir and Agarwal report connections between the worst-case behavior of disjoint-sets and the length of Davenport–Schinzel sequences, a combinatorial structure from computational geometry.[16]

Disjoint Set(DS) data structures are representations of sets (which are all disjoint, sharing no elements) with certain functions:

FindSet(x): finds the set of element x
UniteSets(x,y): unites the sets x and y
MakeSet(x): makes a set with element x
Disunion(list): removes all elements from other sets and makes a new set with these elements
(Note: the disunion operation is not as commonly used as the other operations and is not implemented efficiently, with an O(n) runtime.)

DS is used in various algorithms, such as Kruskal's minimum spanning tree finder.

This post will proceed with a naive implementation, and then progress to an efficient solution.

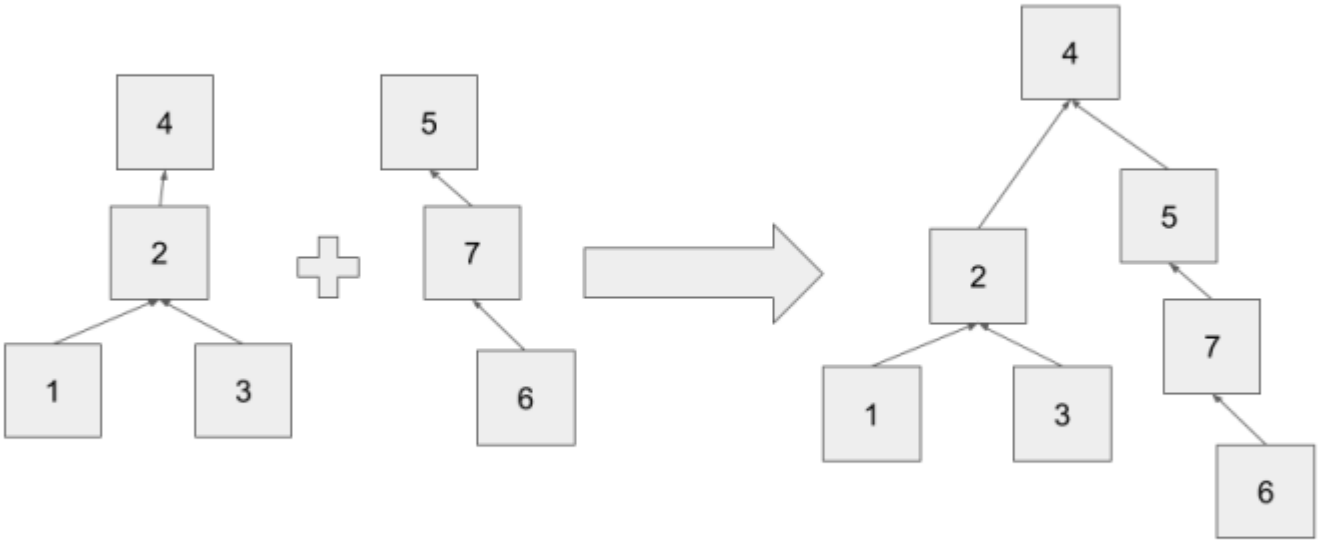
The Naive Solution

We construct a forest (group of disjoint trees). Each element has a parent node and a list of children. (The list of children can be omitted if we are not intending to disunite sets.)

Define P(x): a function which maps an element to its "parent" element. In each set there will be exactly one parent, and P(parent) = parent, this is how a set is referred to.

Each element e starts as being in its own set, with P(e)=e. We can make an element easily with this element, and we can unite two sets X,Y with the following method.

Given two elements x,y, iterate P on each until P(x)=x, P(y)=y. Then set P(x)=y and add x to y's children. Now all elements that were children of x are children of y, so we have successfully united the two sets.



To disunite two sets, iterate through every node N (from farthest from root to root) that we are intending to breakaway. Change P of N's children to P(N) unless N=P(N) and N has more than one child or P(N) is going to be removed as well. If N=P(N)& N has more than one child (if it had one it could be removed with no consequence) instead make one of P(N)'s children the root. (Make sure not to select a child that is going to be removed, if all children are going to be removed go one level deeper.) If P(N) is going to be moved, repeat the procedure for P(N), with all of N's children included under P(N). As every node is processed at most once, the runtime is O(N(cost of removing from children + cost of adding to children)+(cost of building the new set, or n^2)).

Lets look at their runtimes:
FindSet : O(n), as in worst case the structure is a linked list.
UniteSets: O(n), as its runtime is determined by two FindSet calls.
MakeSet: O(1).
Disunion: O(n(cost of removing from children + cost of adding to children)+ n^2).
Don't get too caught up memorizing the above algorithm, it is unwieldy and slow compared to the actual solution.

Optimization 1: Balanced Trees with log N height

Looking at the first version of our DS one possible optimization becomes appearant. Our DS is a forest, so if all the trees were balanced, with a height of log N (something like a binary tree) then both FindSet and, as a result, UniteSets, will take O(log N) time.

As we build our sets from singular elements, our tree structure is determined by UniteSets. This points to potentially editing UniteSets to improve our runtime. Instead of arbitarily combining two sets, we can be smarter. Notice the height of the resulting tree is max(height(x)+1,height(y)) as height(x) and height(y)-1 are the heights of the two subtrees of root y after unification. Looking at this equation we notice some inefficency, if height(x)>height(y) then the height of the resulting tree is greater than if x became the new root. Thus, if we choose the taller tree's root as the final root, we have a more efficent solution.

It turns out that this makes the tree's have a height of O(log N). Let the current maximum height of the whole DS be h. To increase h, we need to combine a tree with height h with another tree of height h (otherwise the max height wouldn't change.) Thus, a tree of height h+1 requires the sum of the elements in the trees of height h. Thus we get a recurrence, # of nodes in tree of height h+1 = 2*# of nodes in tree of height h. As it takes 2^n nodes to form a tree of height n, the height of a tree with N nodes is log N.

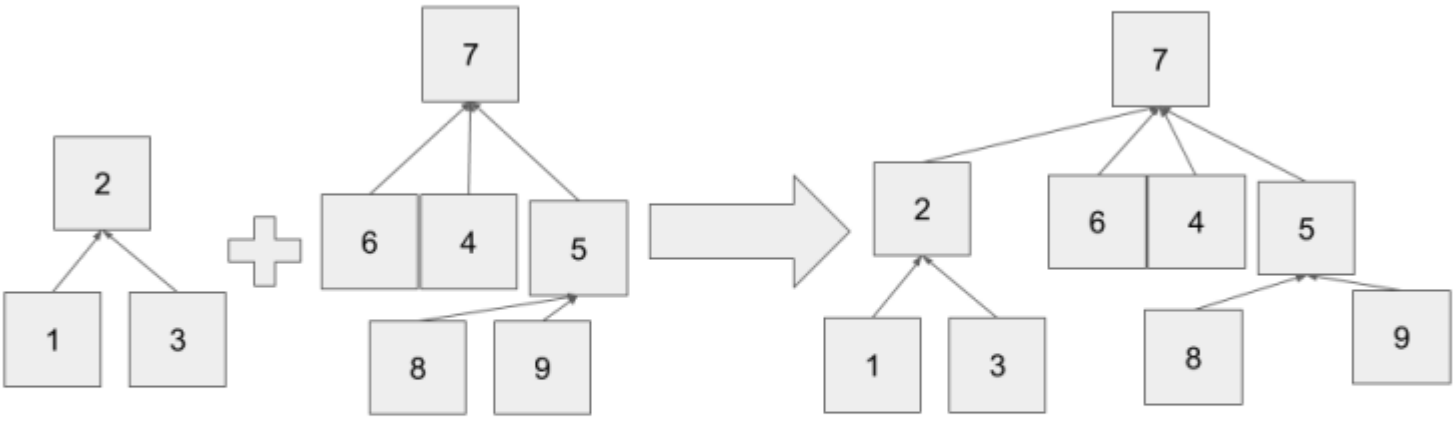
Lets look at the updated runtimes:
FindSet : O(log n), as in worst case the structure is a binary tree.
UniteSets: O(log n), as its runtime is determined by two FindSet calls.
MakeSet: O(1).
Disunion: O(n(cost of removing from children + cost of adding to children)).

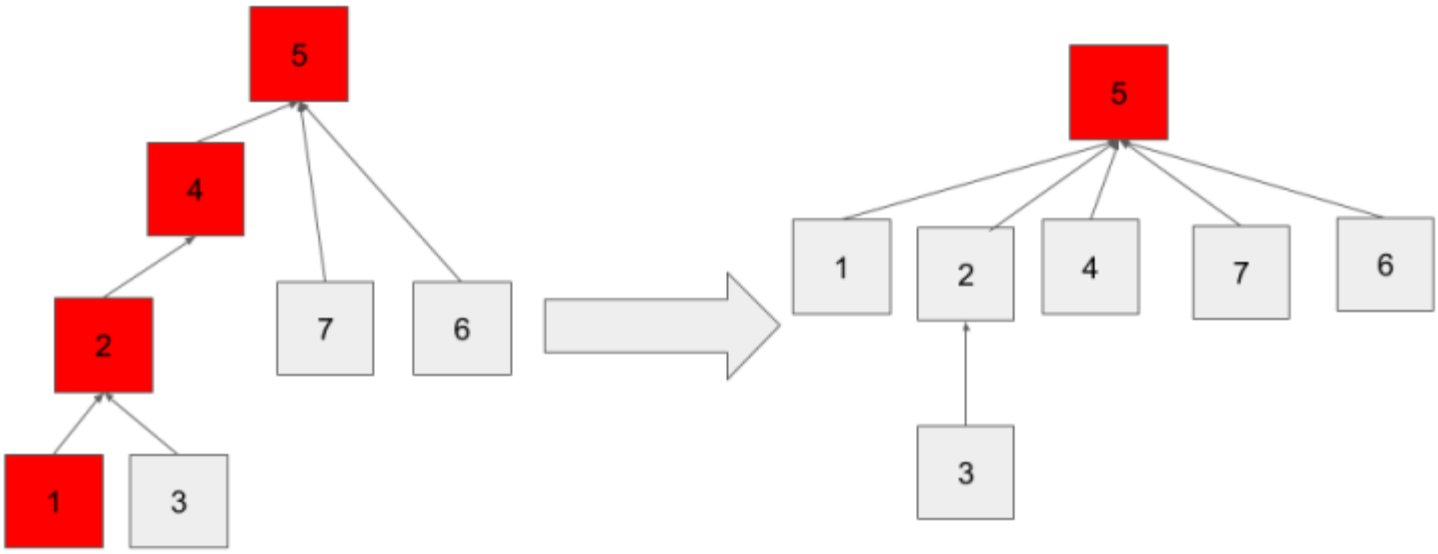
Optimization 2: Path Compression

The runtimes with optimization 1 are pretty good (Disunion will always be O(n) at worst as removing n arbitrary nodes requires touching all n nodes), but can we make them better? With path compression, we can make them even faster (and decrease the memory needed.) However, this heuristic makes analyzing the runtimes kind of complicated based on what you are using the DS for.

The inspiration for this optimization comes from inefficency in FindSet, namely that calling FindSet(x) with the same x multiple times still requires iterating up the tree. This inspires a dp solution, where we memoize the final parent of x. Then, realize that we kind of already do that with P. Instead of creating a seperate array, we can just update P.

When we do path compression we do just that, for every node n from x to the root, set P(n)=root. That way, each of the nodes on the root are only 1 step from finding their parent. For these nodes FindSet becomes O(1).



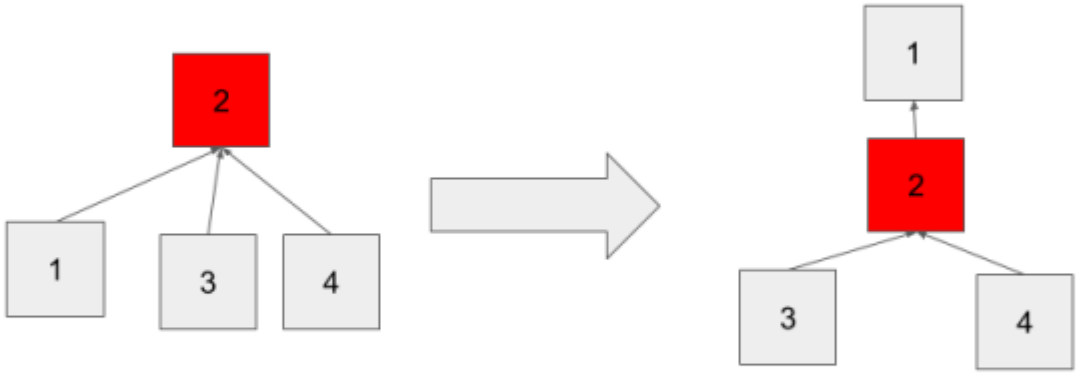


The above image shows path compression after calling findSet(1).

When we unite this set with other sets the distance from these nodes to their root could increase again, and thus our O(1) time progressively becomes ruined. Regardless, allowing path compression results in a much flatter tree, and thus on average reduces the runtime.

Note that with path compression, we can improve our disunion algorithm. We no longer need to store the children of each node. To disunion, run FindSet on every node in order to compress every node. This makes every tree have a height of two. Removing most nodes is now trivial, as they are leaves. Removing the roots is a little harder, but this can be accomplished intelligently. After compressing all the nodes, we again look at every node, and if a we process a node that is not a root whose parent is being disunioned and a root, instead make the non-root node the new root. Then, all the trees have a root that is not being removed (unless the whole

tree is being removed, in which case we don't have to worry about it). After compressing all the nodes again, all disunioned nodes can be removed trivially and heights can be recalculated.



Lets look at our final runtimes:
FindSet : O(α(n)) (This and UniteSets' worst case is the reverse Ackermann function, as uniting two sets with path compression shortens the trees every union. This is effectively constant.)
UniteSets: O(α(n))
MakeSet: O(1).
Disunion: O(n).

All with O(n) space!

3. Answer:

Find a permutation that causes worst case of Merge Sort

Given a set of elements, find which permutation of these elements would result in worst case of Merge Sort.
Asymptotically, merge sort always takes O(n Log n) time, but the cases that require more comparisons generally take more time in practice. We basically need to find a permutation of input elements that would lead to maximum number of comparisons when sorted using a typical Merge Sort algorithm.

Example:

```
Consider the below set of elements

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,

 13, 14, 15, 16}


Below permutation of the set causes 153

comparisons.

{1, 9, 5, 13, 3, 11, 7, 15, 2, 10, 6,

 14, 4, 12, 8, 16}


And an already sorted permutation causes

30 comparisons.


See this for a program that counts
comparisons and shows above results.
```

Now how to get worst case input for merge sort for an input set?

Lets us try to build the array in bottom up manner
Let the sorted array be {1,2,3,4,5,6,7,8}.

In order to generate the worst case of merge sort, the merge operation that resulted in above sorted array should result in maximum comparisons. In order to do so, the

left and right sub-array involved in merge operation should store alternate elements of sorted array. i.e. left sub-array should be {1,3,5,7} and right sub-array should be {2,4,6,8}. Now every element of array will be compared at-least once and that will result in maximum comparisons. We apply the same logic for left and right sub-array as well. For array {1,3,5,7}, the worst case will be when its left and right sub-array are {1,5} and {3,7} respectively and for array {2,4,6,8} the worst case will occur for {2,6} and {4,8}.

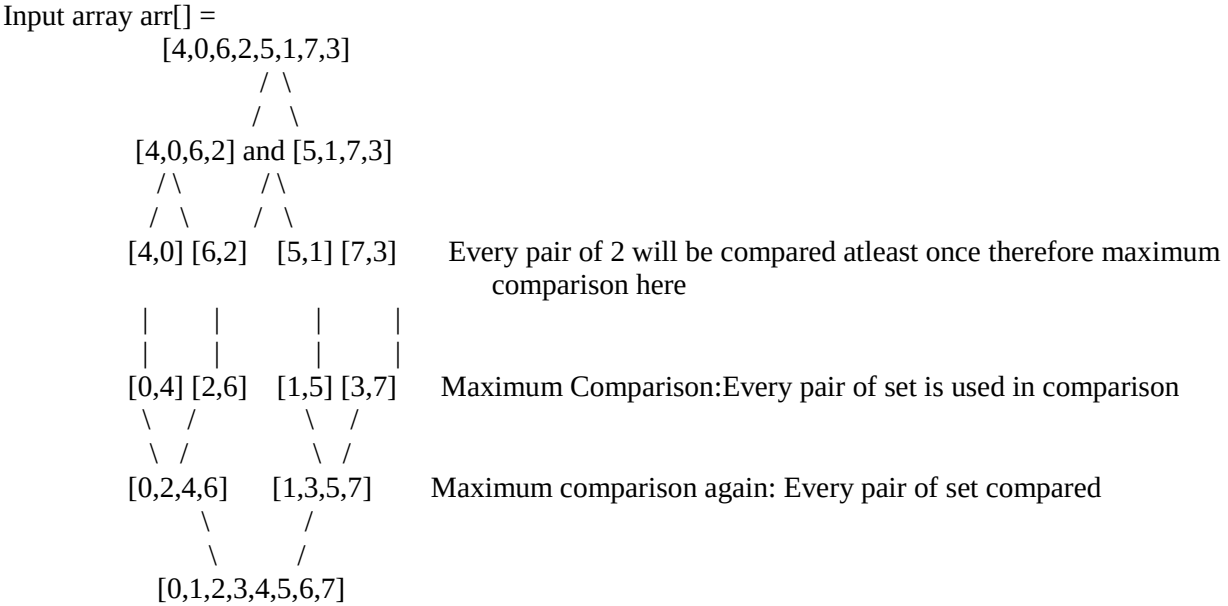
Complete Algorithm –

GenerateWorstCase(arr[])

1. Create two auxiliary arrays left and right and store alternate array elements in them.
2. Call GenerateWorstCase for left subarray: GenerateWorstCase (left)
3. Call GenerateWorstCase for right subarray: GenerateWorstCase (right)
4. Copy all elements of left and right subarrays back to original array.

Another example

Applying Merge Sort using Divide and Conquer



Below is the implementation of the idea

```
// Java program to generate Worst Case of Merge Sort

import java.util.Arrays;

class GFG
{
    // Function to join left and right subarray
    static void join(int arr[], int left[], int right[],
                     int l, int m, int r)
    {
        int i;
        for (i = 0; i <= m - l; i++)
            arr[i] = left[i];

        for (int j = 0; j < r - m; j++)
            arr[i + j] = right[j];
    }

    // Function to store alternate elements in left
    // and right subarray
    static void split(int arr[], int left[], int right[],
                      int l, int m, int r)
    {
        for (int i = 0; i <= m - l; i++)
            left[i] = arr[i * 2];

        for (int i = 0; i < r - m; i++)
            right[i] = arr[i * 2 + 1];
    }

    // Function to generate Worst Case of Merge Sort
    static void generateWorstCase(int arr[], int l, int r)
    {
        if (l < r)
        {
            int m = l + (r - l) / 2;

            // create two auxiliary arrays
            int[] left = new int[m - l + 1];
            int[] right = new int[r - m];

            // Store alternate array elements in left
            // and right subarray
            split(arr, left, right, l, m, r);

            // Recurse first and second halves
            generateWorstCase(left, l, m);
            generateWorstCase(right, m + 1, r);
```

```
   // join left and right subarray
            join(arr, left, right, l, m, r);
        }
    }

    // driver program
    public static void main (String[] args)
    {
        // sorted array
        int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9,
                      10, 11, 12, 13, 14, 15, 16 };
        int n = arr.length;
        System.out.println("Sorted array is");
        System.out.println(Arrays.toString(arr));

        // generate Worst Case of Merge Sort
        generateWorstCase(arr, 0, n - 1);

        System.out.println("\nInput array that will result in \n"+
             "worst case of merge sort is \n");

        System.out.println(Arrays.toString(arr));
    }
}
```