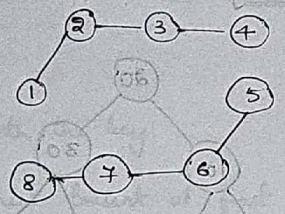
DISJOINT SETS

1. Disjoint sets & operations.
2. Detecting a cycle.
3. Graphical Representation.
4. Array Representation.
5. Weighted Union & collapsing find.

Operations :-

1. Find – to find the set in which the element belongs to.
2. Union.

Example of Disjoint sets :-

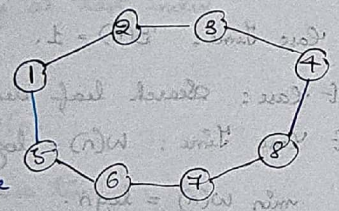


$S1 = \{1, 2, 3, 4\}$

$S2 = \{5, 6, 7, 8\}$.

$S1 \cap S2 = \emptyset$ — disjoint sets, don't have anything in common.

2. Union :-



$S1 = \{1, 2, 3, 4\}$

$S2 = \{5, 6, 7, 8\}$

* To perform union of disjoint sets, we connect edge: (4, 8) → connected edge.
* find 4: 4 in S1 find 8: 8 in S2.
* They belong to different sets to perform union.

$S1 \cup S2 = S3 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

* Perform another union using connected edge (1, 5)

we have $S3 = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

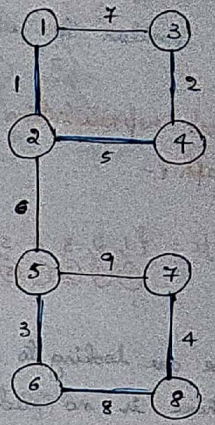
* find 1: 1 in S3 find 5: 5 in S3
* Both belong to same set. So there is a cycle.

* An Example to detect cycle in a graph :-

$U = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

Take a universal set, each number is set.

We will take edges one-by-one in increasing order of weights.



like a minimum cost spanning tree, Kruskal's.

① (1, 2) → find & remove 1, remove 2. perform union on 1 & 2.

$S_1 = \{1, 2\}$.

② (3, 4) find & remove 3, remove 4 perform union.

$S_2 = \{3, 4\}$

③ (5, 6)

$S_3 = \{5, 6\}$.

④ (7, 8)

$S_4 = \{7, 8\}$.

⑤ (2, 4) find 2. (S_1)
find 4 (S_2)
perform union $S_5 = S_1 \cup S_2 = \{1, 2, 3, 4\}$.

⑥ (2, 5) find 2. (S_5).
find 5 (S_3).
perform union $S_6 = S_3 \cup S_5 = \{1, 2, 3, 4, 5, 6\}$.

⑦ (1, 3) find 1. (S_6).
find 3. (S_6)
Both are in same set. There is a cycle.

⑧ (6, 8) find 6. (S_6)
find 8. (S_4)
Perform union $S_7 = S_6 \cup S_4 = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

⑨ (5,7) find 5 (S_7)

find 7 (S_7).

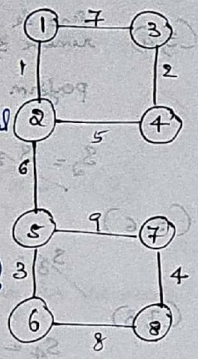
Cycle detected.

This is similar to working of Kruskal's algorithm.

* Graphical representation of Disjoint sets and finding cycle in a graph :-

$\mu = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

① ② ③ ④ ⑤ ⑥ ⑦ ⑧

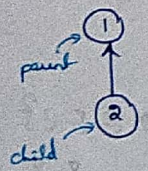


As we are looking to detect cycle in graph and there is no need to perform mathematical operations in our case, we will just do in an easy way — Graphical representation.

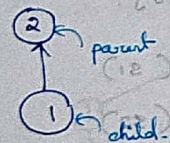
Instead of performing union on 1, 2 for (1,2), we will represent as follows:

① (1,2):

$S_1 = \{1, 2\}$.



(or)



② (3,4):

$S_2 = \{3, 4\}$.

③ (5,6):

$S_3 = \{5, 6\}$

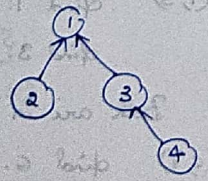
④ (7,8):

$S_4 = \{7, 8\}$

⑤ (2,4):

$S_5 = S_1 \cup S_2$

$\{1, 2, 3, 4\}$

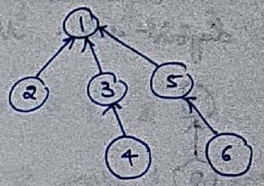


make parent of S_1 as parent of S_2 or viceversa.

⑥ (2,5): weighted union. { S_5 has more weight than S_3. So make S_3 child of S_5's root. }

$S_6 = S_3 \cup S_5$

$= \{1, 2, 3, 4, 5, 6\}$.



⑦ (1,3):

Parent of ① is ①

Parent of ③ is ①

Both belong to same set. So it's a cycle.

* Array Representation of Disjoint sets and finding cycle in a Graph :-

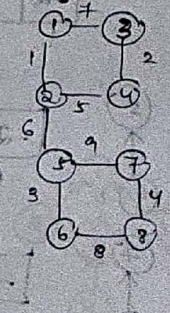
We are not performing union, intersection or set difference operations of mathematics or we don't need set naming.

We just do find and union in a simple way. Our objective is to detect cycle in graph.

We take a single array called parent and the indices are representing 8 vertices and each index having value -1 means each vertex is in its own vertex.

parent	-1	-1	-1	-1	-1	-1	-1	-1
	1	2	3	4	5	6	7	8

We will represent sets in an array and also show it graphically.



① (1,2).

find 1 → -1 (parent is itself)
find 2 → -1 (parent is itself) } in constant time, we "find".

So both are disjoint sets. Making ① as parent of ②.

-2	1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

We make value at index 1 as -2 as '-1' shows parent and '2' shows that there are 2 nodes.

We make value at index 2 as 1 as '1' is parent of '2'.

② (3,4)

find 3 → -1
find 4 → -1

Perform union making ③ as parent of ④.

-2	1	-2	3	-1	-1	-1	-1
1	2	3	4	5	6	7	8

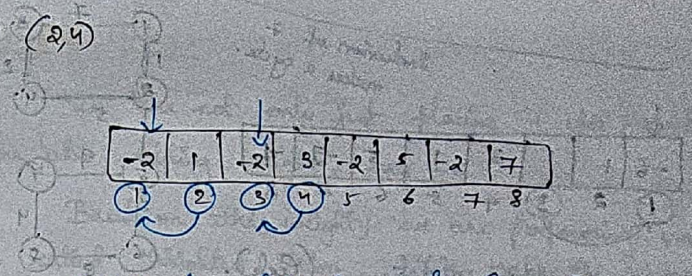
③

-2	1	-2	3	-2	5	-1	-1
1	2	3	4	5	6	7	8

④

-2	1	-2	3	-2	5	-2	7
1	2	3	4	5	6	7	8

⑤ (2,4)



-2	1	-2	3	-2	5	-2	7
1	2	3	4	5	6	7	8

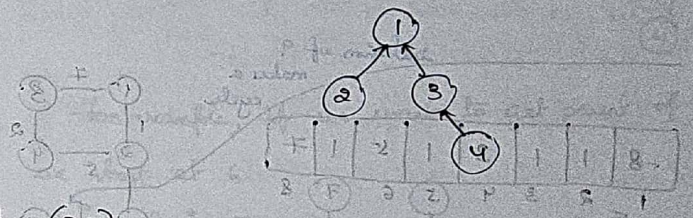
2 & 4 have different parents. (1 & 3).

Perform union in array ⇒ make 1 as parent of 3.

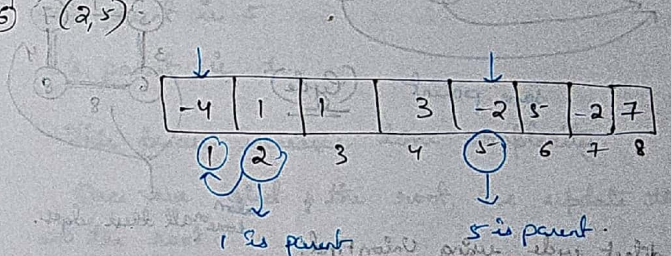
⇒ change value at 3 as 1.

⇒ update 1 as -4 (- is parent, 4 is weight).

-4	1	1	3	-2	5	-2	7
1	2	3	4	5	6	7	8



⑥ (2,5)



-4	1	1	3	-2	5	-2	7
1	2	3	4	5	6	7	8

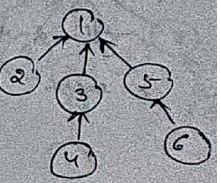
1 is parent
weight 4 or rank

>

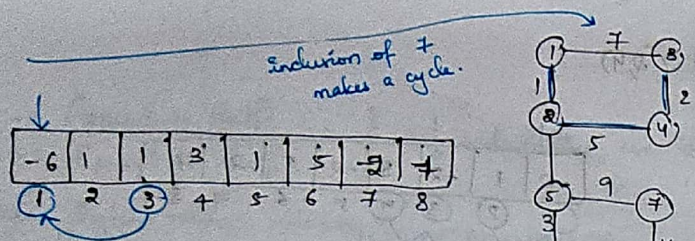
5 is parent.
weight 2 or rank

make 1 parent of 5.

-6	1	1	3	1	5	-2	7
1	2	3	4	5	6	7	8

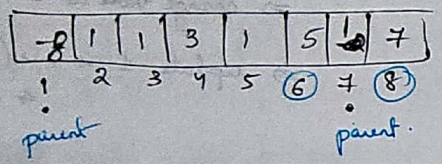


(7) (1,3)



Both have same parent. (Cycle).

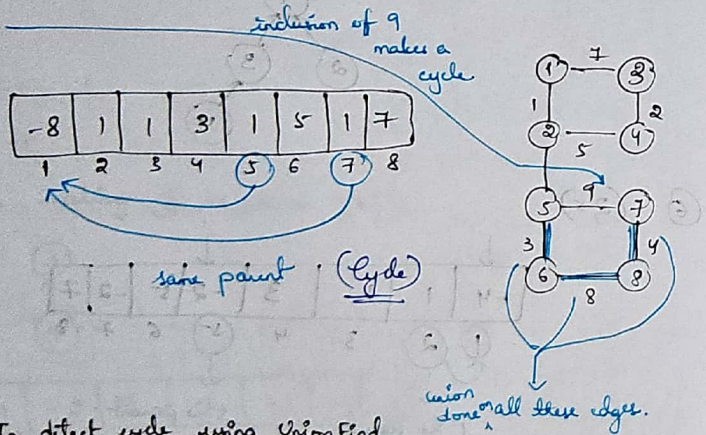
(8) (6,8)



rank = 6 > rank = 2

make 7 child of 1.

(9) (5,7)



same parent (Cycle)

union done on all these edges.

Conclusion:- To detect cycle using Union Find.

If 2 vertices of an edge belong to two disjoint sets, then perform a union.

If 2 vertices of an edge belong to same set, then there is a cycle.

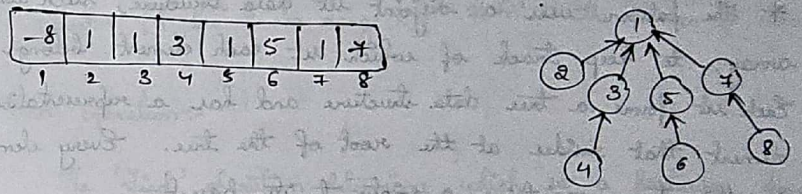
Weighted Union:-

We are not only just placing -1 for parent but we are also adding weight, say -6.

Based on this weight, we are performing union. This is called Weighted Union. We make the parent with highest weight as parent of another set.

Collapsing Find:-

Suppose we have an array and graph as follows:



For example, if we want to get parent of 6,

we start at 6.

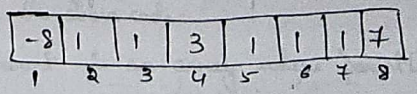
6's parent is 5

5's parent is 1.

This process increases time complexity.

Once we find the root, we update the value of the node as the root.

So, in above example, instead of 5, we update as 1 at index 6.



The process of directly linking a node to the direct parent of a set is called as collapsing find.

With collapsing find, we can reduce the time for finding same value next time.

* Union Find also known as Disjoint Set Union (DSU)

* A disjoint-set data structure is defined as one that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

* The Union Find Pattern or algorithm is used to group elements into sets based on a property. Each set is non-overlapping, that is, it contains unique elements that are not present in any other set.

* The pattern uses a disjoint set data structure, such as an array, to keep track of which set each element belongs to. Each set forms a tree data structure and has a representative element that resides at the root of the tree. Every element in this tree maintains a pointer to its parent.

* The Union Find algorithm performs two useful operations.

① Find: Determine which subset a particular element is in.

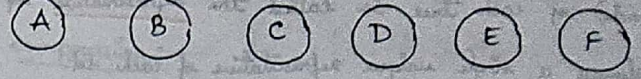
② Union: Join two subsets into a single subset.

Operations on Disjoint Set Data Structures:-

Creating Disjoint Sets:

* Each element points to a parent.
* Initially, each element is its own parent, forming 'n' disjoint sets.
* A "root" element is one that points to itself, identifying the set's representative.

Initial State:



Element	A	B	C	D	E	F
Parent	A	B	C	D	E	F

```
# Initialize 'n' elements where each element is its own parent.
parent = [i for i in range(n)]  # 'n' is the number of elements.
```

2. Find Operation :-

The 'Find' operation determines which set a particular element belongs to. This can be used to determine if two elements are in the same set.

```
def find(i):
    # If the element is its own parent, it's the representative
    # of its set.
    if parent[i] == i:
        return i
    # Otherwise, recursively find the representative of the set.
    else:
        return find(parent[i])
```

In the above code:

* "parent" is an array where "parent[i]" is the parent of "i".
* If "parent[i] == i," then "i" is the root of the set and hence the representative.
* The "find" function follows the chain of parents for i until it reaches the root.

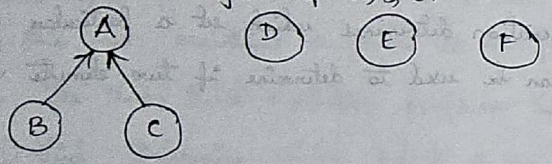
We use "Tree" to represent disjoint set. The root node (i.e the topmost node) of each tree is called the "representative" of the set. There is always a single unique representative of each set.

A simple rule to identify a representative is "if 'i' is the representative of a set, then Parent[i] = i. If 'i' is not the representative of this set, then it can be found by traversing up the tree until we find the representative.

3. Union Operation :-

The Union Operation merges two disjoint sets. It takes two elements as input, finds the representatives of their sets using the "Find" operation, and finally merges the two sets.

Taking union of A, B, C.



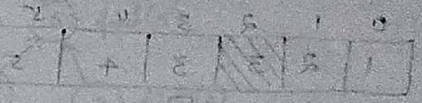
Updated Parent elements after taking union of A, B, C.

Element	A	B	C	D	E	F
Parent	A	A	A	D	E	F

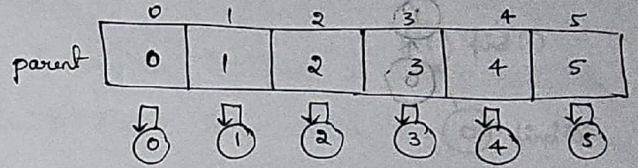
```
def union(i, j):
    # find the representative of two sets
    irep = find(i)
    jrep = find(j)
    # if they are already in the same set, return.
    if irep == jrep:
        return
    # make representative of the second the representative of first
    parent[irep] = jrep
```

Optimizations :-

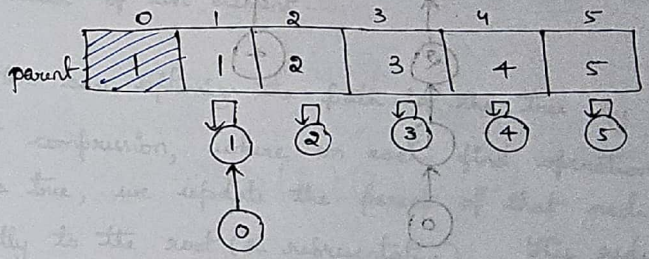
1. Path compression :-



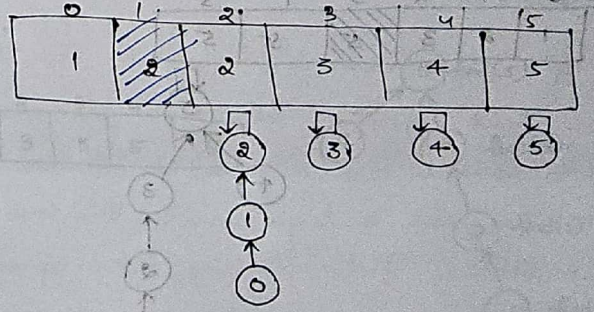
Example :-



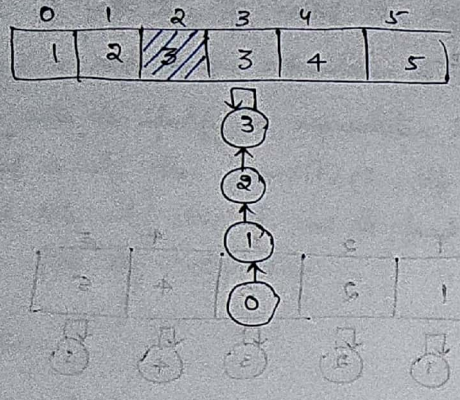
↳ union(0,1) merge tree containing 0 (find(0)) with tree containing 1 (find(1)) and update the representative of tree with 0 to the representative of tree with 1.



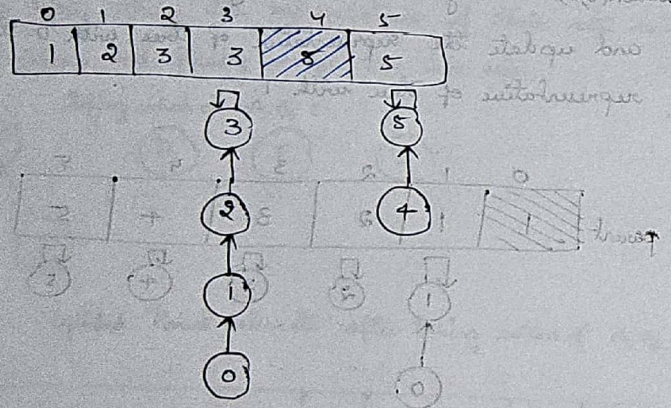
↳ union(1,2) find(1) = 1 find(2) = 2.
representative of tree with 1 = representative of tree with 2.



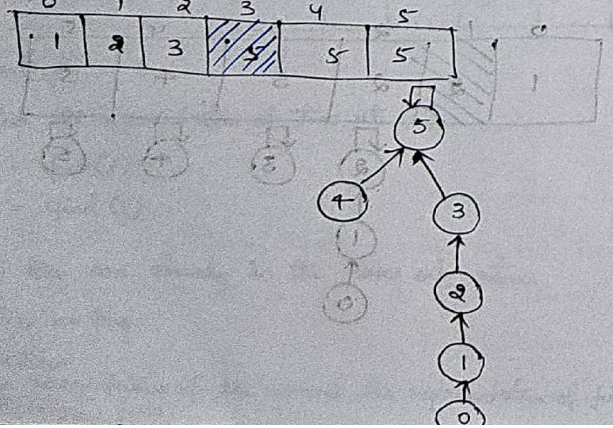
union(2,3):



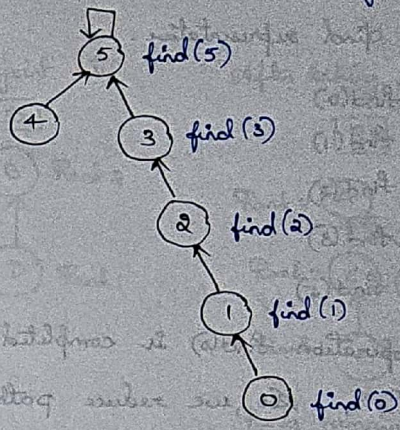
union(4,5):



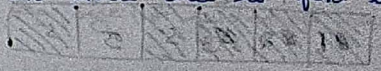
union(3,5):



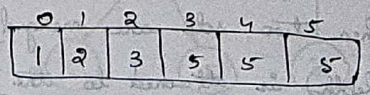
find(0): Find representative of tree having 0.

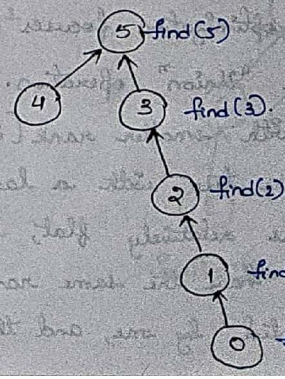


The worst-case time complexity of this approach is O(n) because we might have to traverse the entire tree to find the representative of an element.

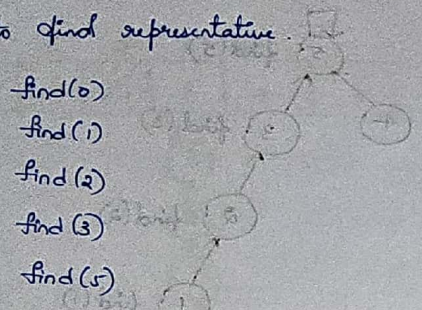


We can optimize the path of the tree by using Path compression, where on each find operation on a node of a tree, we update the parent of that node to point directly to the root (or representative). This reduces the length of the path of that node to the root, ensuring we don't have to travel all the intermediate nodes on future find operations.

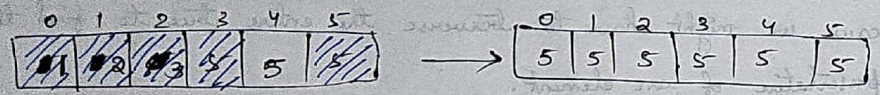


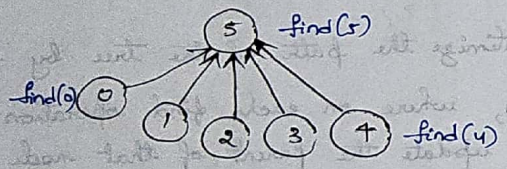


Suppose, we first do a find(0) operation, it would take 5 operations to find representative.



After this operation find(0) is completed, we now have representative as "5". So we reduce path to "5" directly by updating all parents of nodes 0, 1, 2, 3, 5 as representative which is "5".





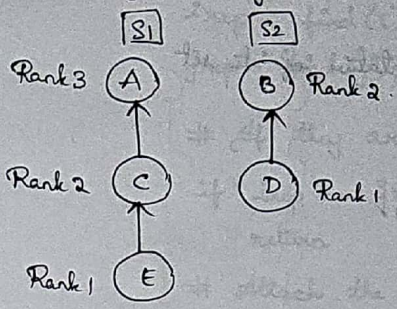
5 operations reduced to 2.

2. Union by Rank:

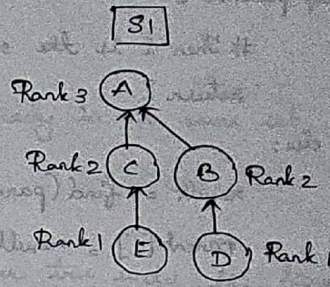
This optimization focuses on keeping the tree balanced during the "Union" operation. When two sets are merged, the tree with smaller rank (or depth) is attached to the root of the tree with a larger rank. This ensures that the tree remains relatively flat, optimizing future operations. If both trees have the same rank, the rank of one of them is incremented by one, and the other tree is attached to it.

Example:-

Initial Disjoint sets



After taking union of S1 and S2.



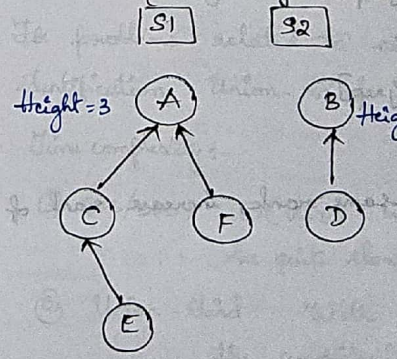
Ranks remain same after union.

3. Union by Size :-

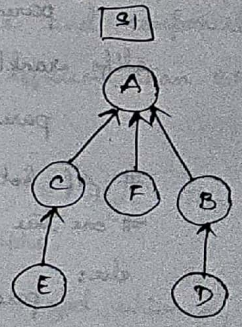
This optimization considers the size (or number of nodes) of trees when merging them.

The primary goal is to ensure that the smaller tree (in terms of nodes) is always attached to the larger tree. This approach helps in maintaining a balanced tree structure, ensuring that operations on the data structure remain efficient.

Initial Disjoint sets



After taking union of S1 and S2



Smaller tree is attached to larger tree.

* def find(i):

```
    # If i is the parent of itself
    if parent[i] == i:
        # Then i is the representative of its set
        return i
    else:
        result = find(parent[i])
        parent[i] = result
        return result
```

* def unionbyrank(i, j):

```
    # Find representatives of two sets
    irep = find(i)
    jrep = find(j)

    # If they are already in same set, return
    if irep == jrep:
        return

    # Attach the tree with a smaller rank under the
    # tree with a larger rank.
    if rank[irep] < rank[jrep]:
        parent[irep] = jrep
    elif rank[irep] > rank[jrep]:
        parent[jrep] = irep
    # If both trees have same rank, increase rank of
    # one tree by 1.
    else:
        parent[irep] = jrep
        rank[jrep] += 1
```

* def unionbysize(i, j):

```
    # Find the representatives of the two sets.
    irep = find(i)
    jrep = find(j)

    # If they are already in the same set, return.
    if irep == jrep:
        return
    # Attach the smaller tree under the larger tree
    if size[irep] < size[jrep]:
        parent[irep] = jrep
        size[jrep] += size[irep]
    else:
        parent[jrep] = irep
        size[irep] += size[jrep]
```

This brings the time complexity down to $O(\log n)$ in worst case.

Why choose Union Find Over BFS/DFS?

You can solve similar problems using BFS & DFS but the Union Find algorithm provides the optimal solution over them. For problems related to connectivity checks and component identification, Union-Find is often more efficient than BFS/DFS.

Time complexity:-

① BFS/DFS: $O(V+E)$ where V - vertices, E - edges. This can be quite slow for large graphs.

② Union-Find: With path compression and union by rank, the amortized time complexity for union and find operations can be approximated as $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. This $\alpha(n)$ grows very slowly, making Union-Find operations almost constant time.

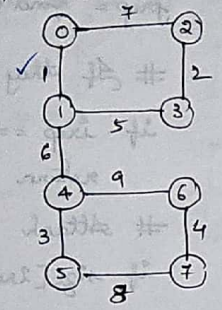
Find & Union Operation :-

Program :-

```
public int find(int i) {
    if (parent[i] == i)
        return i;
    return find(parent[i]);
}
```

	0	1	2	3	4	5	6	7
parent	0	0	0	2	0	4	0	6

graph :-




```
public void union(int u, int v)
    int urep = find(u);
    int vrep = find(v);
    parent[vrep] = urep;
}
```

Dry Run :-

(0,1)
find(0) = 0 [parent[0]==0]
find(1) = 1 [parent[1]==1]
union(0,1) ⇒ parent[1] = 0.

(2,3)
find(2) = 2 [parent[2]==2]
find(3) = 3 [parent[3]==3]
union(2,3) ⇒ parent[3] = 2

(4,5)
find(4) = 4 [parent[4]==4]
find(5) = 5 [parent[5]==5]
union(4,5) ⇒ parent[5] = 4

(6,7)
find(6) = 6 [parent[6]==6]
find(7) = 7 [parent[7]==7]
union(6,7) ⇒ parent[7] = 6.

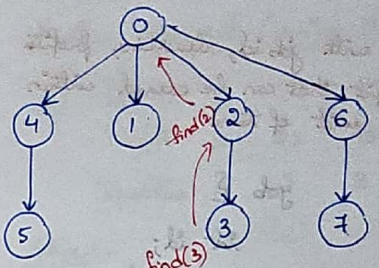
(1,3) find(1) ⇒ parent[1] ≠ 0 ⇒ find(0) = 0 [parent[0]==0].
find(3) ⇒ parent[3] = 2 ⇒ find(2) = 2 [parent[2]==2].
union(1,3) ⇒ parent[2] = 0

(1,4) find(1) ⇒ parent[1] = 0 ⇒ find(0) = 0 [parent[0]==0]
find(4) ⇒ parent[4] = 4
union(1,4) ⇒ parent[4] = 0

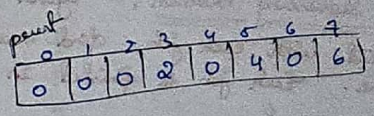
(0,2) find(0) ⇒ parent[0] == 0.
find(2) ⇒ parent[2] == 0 // cycle.

(5,7) find(5) ⇒ parent[5] = 4 ⇒ find(4) = 0 ⇒ find(0) = 0
find(7) ⇒ parent[7] = 6 ⇒ find(6) = 6
union(5,7) ⇒ parent[6] = 0 ⇒ find(6) = 6

(4,6) find(4) ⇒ parent(4) = 0 ⇒ find(0) = 0
find(6) ⇒ parent[6] = 0 ⇒ find(0) = 0 // cycle.

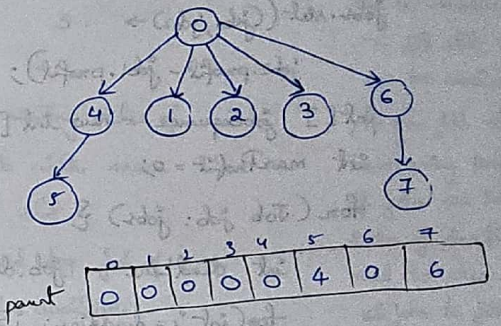


↳ find(3)
parent[3] = 2
↳ find(2)
parent[2] = 0.



find(3)
parent[3] = 2
0 ← find(2)
parent[3] = 0.

collapsing find (or) path compression



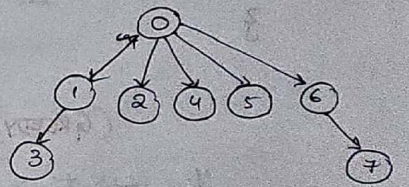
Collapsing find :-

```
public int cfind(int i) {
    if (parent[i] == i)
        return i;
    return parent[i] = cfind(parent[i]);
}
```

Dry Run :-

	0	1	2	3	4	5	6	7
parent	0	0	0	1	0	0	0	6

(0,1) - 0 1
(2,3) - 2 3
(1,4) - 0 4
(0,2) - 0 0 cycle.
(5,7) 0 6
(4,6) 0 0 cycle.
(4,5) - 4 5
(6,7) - 6 7
(1,3) - 0 2



Job Sequencing Problem:- Given job with job id, deadline, profits associated with each job, find max Profit that can be earned within deadline. Each job is finished in one unit of time.

```
main() {
    ans = bookSlots(jobs, 3);
    print ans;
}

bookSlots(List<Job> jobs,
          int maxDeadline) {
    jobs.sort((job1, job2) ->
        job2.profit - job1.profit);
    int[] jobSequence = new int[maxDeadline+1];
    int maxProfit = 0;
    for(Job job: jobs) {
        int deadLine = job.deadline;
        for(int j = deadline; j >=1; j--) {
            if(jobSequence[j]==0) {
                jobSequence[j] = job.id;
                maxProfit += job.profit;
                break;
            }
        }
    }
    return new int[][] { new int[] {maxProfit}, jobSequence};
}
```

```
Job {
    int id;
    int profit;
    int deadline;
    public Job(id, profit, dl){
        id = id;
        profit = profit;
        deadline = dl;
    }
}
```

(GREEDY)

Time Complexity: $O(N^2)$

Space complexity: $O(N)$

Another approach (using Heap):-

Q) Job ID: J_1 J_2 J_3 J_4 J_5

Profit: 3 25 15 15 4

Deadline: 5 3 3 1 2

Steps:-

① Sort jobs based on deadlines

1	2	3	3	5
J_4	J_5	J_2	J_3	J_1
15	4	25	15	3

profits:

② Fill slots with decreasing order of deadlines so that job with max deadline can fill last slot.

slot 5 can be filled by job of deadline 5 if the job is available.

slot 3 can be filled by jobs with deadline ≥ 3 if available.

③ Save these jobs in Max Heap so that (profit) job available with highest profit can be prioritized first.

④ Fill the slot only if slot is available. Compare deadline of current & previous, the difference should be greater than 0.

Idea:

A job with deadline 5 can fill slots from 1 to 5 – 1 2 3 4 5

To fill slot 2, J_4, J_5 can be considered.

To fill slot 3, J_1, J_2, J_3 can be considered.

If you fill slots in increasing order & even if jobs are in increasing order of deadlines, you need to check 2 to 3 jobs to fill.

For slot n, analyse deadlines of value ≥ n. At worst case, for a slot, n comparisons are required.

Time complexity $O(n^2)$.

So fill slots from last to first.

```
printJobScheduling(Job arr[], int n){
    int n = arr.size();
    Collections.sort(arr, (a,b)->{return a.deadline-b.deadline;});
    List<Job> result = new ArrayList<>();
    PriorityQueue<Job> maxHeap = new PriorityQueue<>((a,b) -> b.profit-a.profit);
```

Next page is the end of this book.

```
for(int i=n-1; i>=0; i--) {
    int slot_available;
    if(i==0) {
        slot_available = arr.get(i).deadline;
    } else {
        slot_available = arr.get(i).deadline - arr.get(i-1).deadline;
    }
    max_Heap.add(arr.get(i));
    while(slot_available>0 && maxHeap.size()>0) {
        Job job = maxHeap.remove();
        slot_available--;
        result.add(job);
    }
}
Collections.sort(result, (a,b) -> a.deadline - b.deadline);
return result;
}
```

Time Complexity: $O(N \log N)$

Auxiliary Space: $O(N)$.

Job Sequencing using Union Find:-

	J_1	J_2	J_3	J_4	J_5
Profit:	3	25	15	15	4
Deadline:	5	3	3	1	2

→ Decreasing order of profit →

	J_2	J_3	J_4	J_5	J_1
P:	25	15	15	4	3
d:	3	3	1	2	5

maxDeadline = 5

⇒ 5 slots

S_1	S_2	S_3	S_4	S_5
1	2	3	4	5

① $J_2 \rightarrow d = 3$, So assign time slot S_3 to J_2

find(3) = 3 // ⇒ S_3 is available

perform union of S_2, S_3.

S_1	S_2	S_3	S_4	S_5
1	2	2	4	5

Now find(d) ⇒ find(3) = 2.

⇒ 2 is next available slot.

② $J_3 \rightarrow d = 3$, find(3) = 2 ⇒ S_2 is available.

fill S_2 with J_3 ⇒ union(S_1, S_2) = union(1,2)

S_1	S_2	S_3	S_4	S_5
1	1	2	4	5

③ $J_4 \rightarrow d = 1$, find(1) = 1 ⇒ S_1 is available

fill S_1 with J_4 ⇒ union(S_0, S_1) ⇒ $S_1 = 0$.

S_1	S_2	S_3	S_4	S_5
0	1	2	4	5

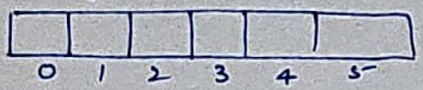
④ $J_5 \rightarrow d = 2$, find(2) = 0 ⇒ no slot available.

S_1	S_2	S_3	S_4	S_5
0	1	2	4	4

⑤ $J_1 \rightarrow d = 5$, find(5) = 5 ⇒ S_5 available. union(4,5)

Union Find (Continuation) for Job Sequencing Problem

Using Disjoint Set for Job Sequencing:-

* All time slots are individual sets initially.
* We first find max deadline of all jobs m.
* We create m+1 individual sets 

* If a job is assigned a time slot of t where t >= 0, then the job is scheduled during [t-1, t].
* We need to keep track of the greatest time slot available which can be allotted to a given job having deadline. We use the parent array of Disjoint Set Data Structures for this purpose. The root of the tree is always the latest available slot.
* If for a deadline d, there is no slot available, then root would be 0.

How come find returns the latest available time slot?

* Initially, all the time slots are individual slots. So the time slot returned is always maximum.
* When we assign a time slot 't' to a job, we do union of 't' with 't-1' in a way that 't-1' becomes the parent of 't'. To do this we call union(t-1, t).
* This means that all future queries for time slot t would now return the latest time slot available for set represented by t-1.

```
printJobScheduling(ArrayList<Job> arr) {
    Collections.sort(arr, new Job());
    int maxDeadline = findMaxDeadline(arr);
    UnionFind uf = new UnionFind(maxDeadline);
    for (Job temp : arr) {
        int availableSlot = uf.find(temp.deadline);
        if (availableSlot > 0) { uf.merge(uf.find(availableSlot-1), availableSlot); }
    }
}
```

TC: O(NlogD)
SC: O(D)