Bubble-sort

Bubble Sort

```
procedure BUBBLESORT(var A: arraytype; n: integer);
var
   i, j: integer;
   temp: itemtype;
begin {BUBBLESORT}
   for i ← 1 to n - 1 do
     for j ← 1 to n - i do
       if A[j] > A[j + 1] then
         begin
           temp ← A[j];
           A[j] ← A[j + 1];
           A[j + 1] ← temp;
         end;
end {BUBBLESORT};
```

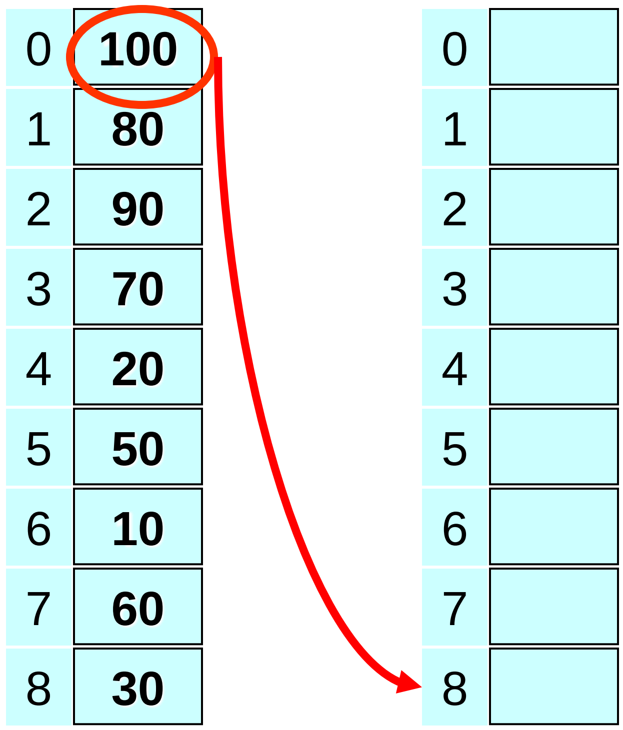
We can modify this algorithm so as to stop early if no swaps happen in a pass (indicating the array is already sorted): The best case time complexity then will be O(n), but worst and average remains $O(n^2)$

- Back to our old question
- How to build a heap from the array
- Remove elements from the heap one by one and insert them back into the array

```
procedure HEAPSORT(A:arraytype; n:integer);
var i:integer; item: itemtype;
begin
    HEAPIFY(A,n)
    for i ← n downto 2 do
    begin
        item ← A[i];
        A[i] ← A[1];
        A[1] ← item;
        ADJUST(A,1,i-1);
    end;
end.
```

0	**100**
1	**80**
2	**90**
3	**70**
4	**20**
5	**50**
6	**10**
7	**60**
8	**30**

0	
1	
2	
3	
4	
5	
6	
7	
8	


0 100
1 80
2 90
3 70
4 20
5 50
6 10
7 60
8 30
0
1
2
3
4
5
6
7
8

0	
1	**80**
2	**90**
3	**70**
4	**20**
5	**50**
6	**10**
7	**60**
8	**30**

0	
1	
2	
3	
4	
5	
6	
7	
8	**100**

0	**90**
1	**80**
2	**50**
3	**70**
4	**20**
5	**30**
6	**10**
7	**60**
8	

0	
1	
2	
3	
4	
5	
6	
7	
8	**100**

0	**90**
1	**80**
2	**50**
3	**70**
4	**20**
5	**30**
6	**10**
7	**60**
8	

0	
1	
2	
3	
4	
5	
6	
7	
8	**100**

0	
1	**80**
2	**50**
3	**70**
4	**20**
5	**30**
6	**10**
7	**60**
8	

0	
1	
2	
3	
4	
5	
6	
7	**90**
8	**100**

0	**80**
1	**70**
2	**50**
3	**60**
4	**20**
5	**30**
6	**10**
7	
8	

0	
1	
2	
3	
4	
5	
6	
7	**90**
8	**100**

0	**80**
1	**70**
2	**50**
3	**60**
4	**20**
5	**30**
6	**10**
7	
8	

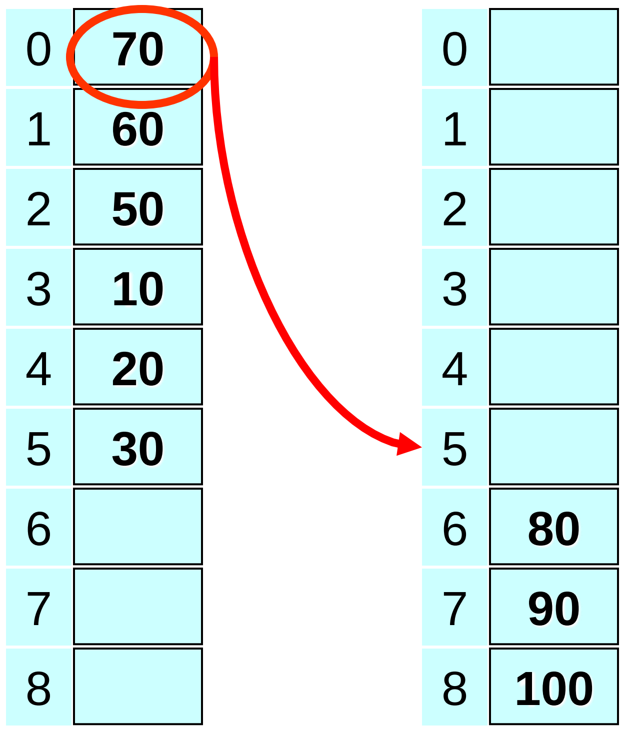
0	
1	
2	
3	
4	
5	
6	
7	**90**
8	**100**

0	
1	**70**
2	**50**
3	**60**
4	**20**
5	**30**
6	**10**
7	
8	

0	
1	
2	
3	
4	
5	
6	**80**
7	**90**
8	**100**

0	**70**
1	**60**
2	**50**
3	**10**
4	**20**
5	**30**
6	
7	
8	

0	
1	
2	
3	
4	
5	
6	**80**
7	**90**
8	**100**

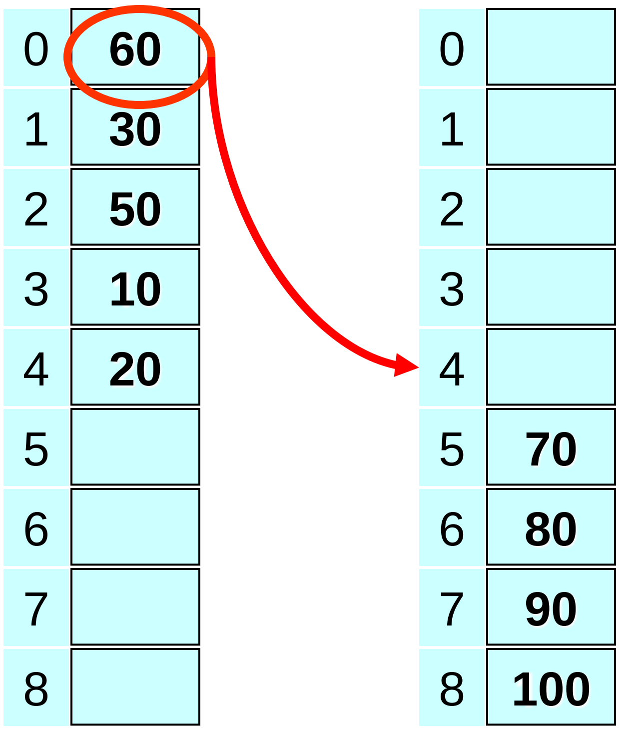

0
70
1
60
2
50
3
10
4
20
5
30
6
7
8
0
1
2
3
4
5
6
80
7
90
8
100

0	
1	**60**
2	**50**
3	**10**
4	**20**
5	**30**
6	
7	
8	

0	
1	
2	
3	
4	
5	**70**
6	**80**
7	**90**
8	**100**

0	**60**
1	**30**
2	**50**
3	**10**
4	**20**
5	
6	
7	
8	

0	
1	
2	
3	
4	
5	**70**
6	**80**
7	**90**
8	**100**

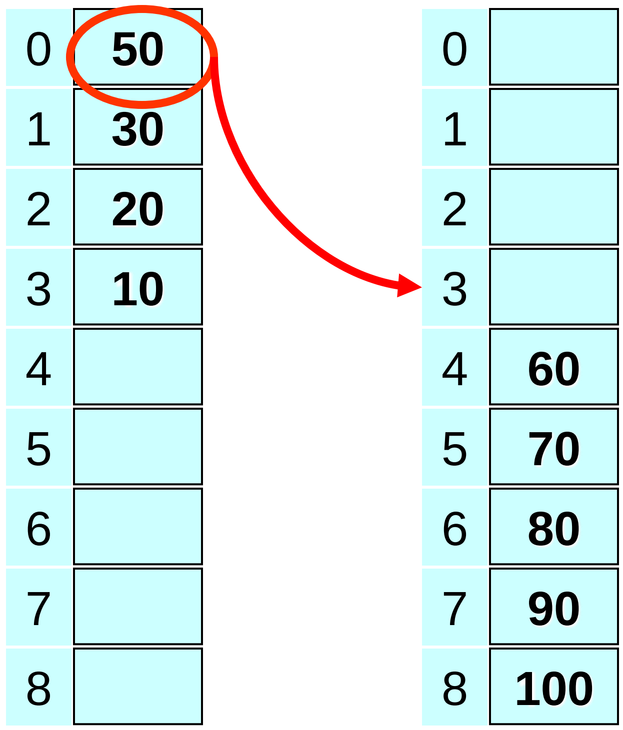

0 60
1 30
2 50
3 10
4 20
5
6
7
8
0
1
2
3
4
5 70
6 80
7 90
8 100

0	
1	**30**
2	**50**
3	**10**
4	**20**
5	
6	
7	
8	

0	
1	
2	
3	
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

0	**50**
1	**30**
2	**20**
3	**10**
4	
5	
6	
7	
8	

0	
1	
2	
3	
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**


0 50
1 30
2 20
3 10
4
5
6
7
8
0
1
2
3
4 60
5 70
6 80
7 90
8 100

0	
1	**30**
2	**20**
3	**10**
4	
5	
6	
7	
8	

0	
1	
2	
3	**50**
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

0	**30**
1	**10**
2	**20**
3	
4	
5	
6	
7	
8	

0	
1	
2	
3	**50**
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

0	**30**
1	**10**
2	**20**
3	
4	
5	
6	
7	
8	

0	
1	
2	
3	**50**
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

0	
1	**10**
2	**20**
3	
4	
5	
6	
7	
8	

0	
1	
2	**30**
3	**50**
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

0	**20**
1	**10**
2	
3	
4	
5	
6	
7	
8	

0	
1	
2	**30**
3	**50**
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

0	**20**
1	**10**
2	
3	
4	
5	
6	
7	
8	

0	
1	
2	**30**
3	**50**
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

0	
1	**10**
2	
3	
4	
5	
6	
7	
8	

0	
1	**20**
2	**30**
3	**50**
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

0	**10**
1	
2	
3	
4	
5	
6	
7	
8	

0	
1	**20**
2	**30**
3	**50**
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

0	**10**
1	
2	
3	
4	
5	
6	
7	
8	

0	
1	**20**
2	**30**
3	**50**
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

0	
1	
2	
3	
4	
5	
6	
7	
8	

0	**10**
1	**20**
2	**30**
3	**50**
4	**60**
5	**70**
6	**80**
7	**90**
8	**100**

- Though the call of HEAPIFY requires only *O*(n) operations, ADJUST possible requires *O*(log n) operations for each invocation.

- **Thus the worse case time is O(n log n).**

- **Sets and disjoint unions**

- **Problem:** Suppose we have a finite universe of n elements U, out of which sets will be created.

- **Representation:** SET(1:n) such that SET(i) = 1 if i^{th} element of U is present, otherwise 0.

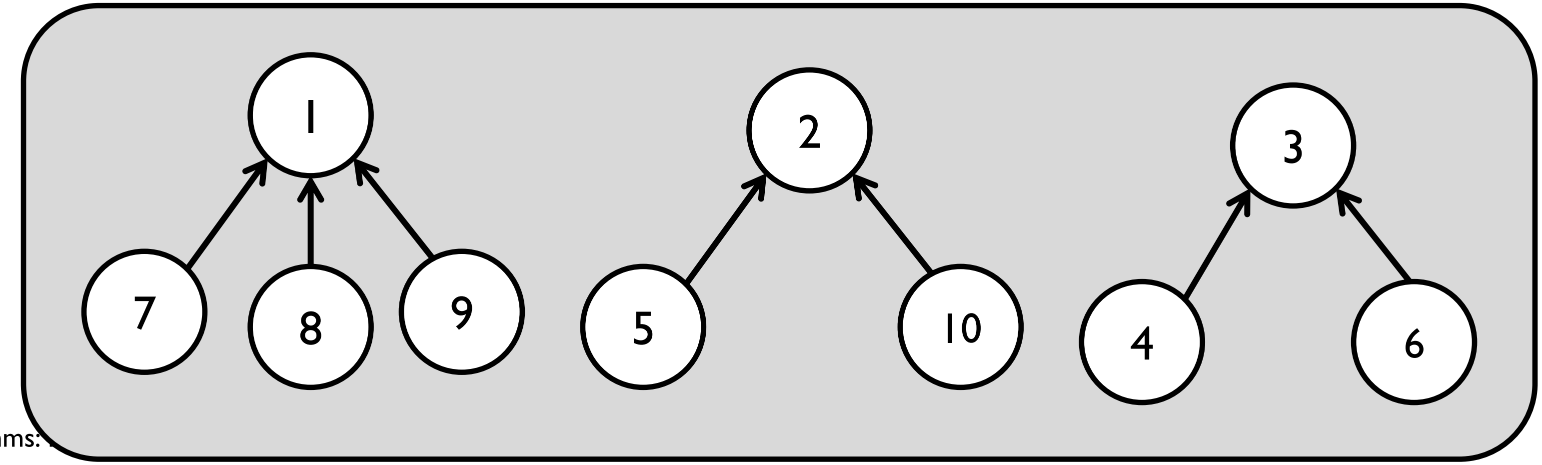
- This array is called the ***characteristic vector*** for the set.

- **Advantages:** It can be easily determined whether or not a particular element i is present.
- *Union* *and* *Intersection* can be done using "logical and" and "logical or".
- **Disadvantages:** This representation is inefficient when the value of n is large and the size of the set is smaller compare to U.
- The time will be proportional to n rather than the number of elements in the set.

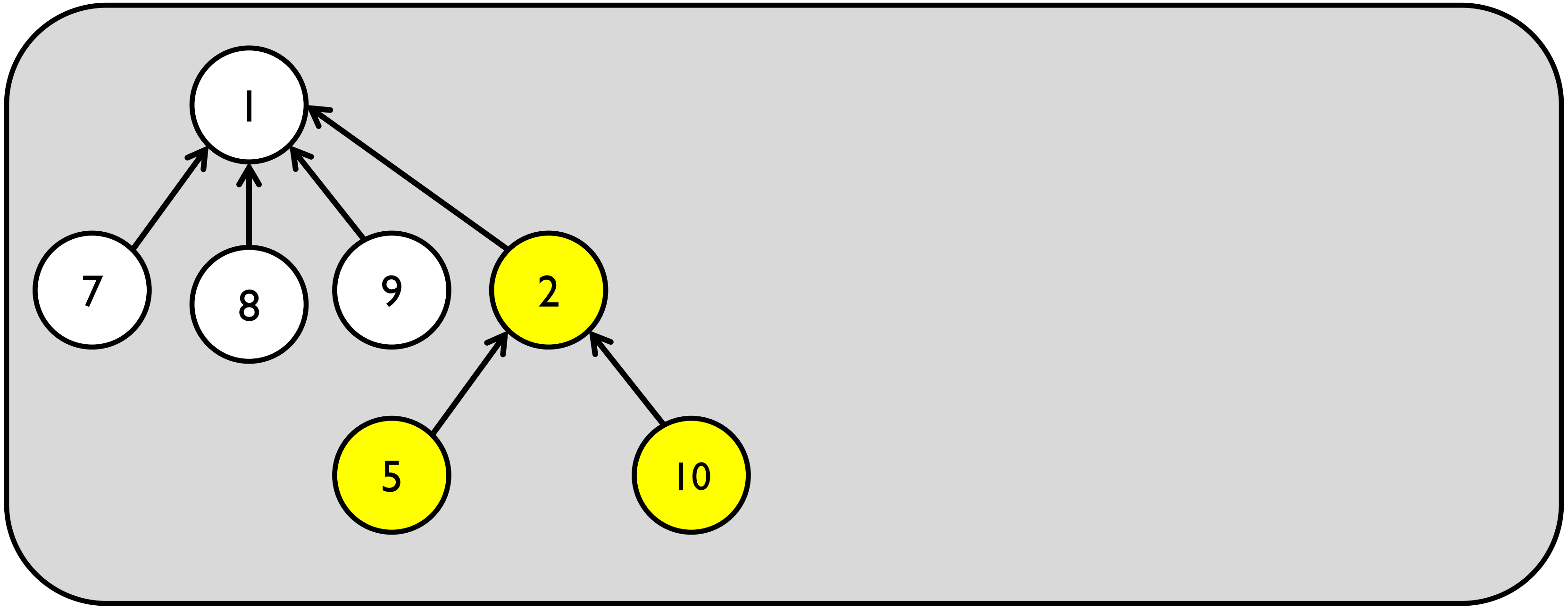
- **Alternate representation:** Represent each set by its element (assuming m in first and n in second). If there is any ordering relationship between them, then the operation such as union and intersection can be carried out in time proportional to the length of the sets.
- (Can you write code that does this in $O(m+n)$ time?)

- We represent sets as trees.
- We assume that they are pair wise disjoint and perform these operations.

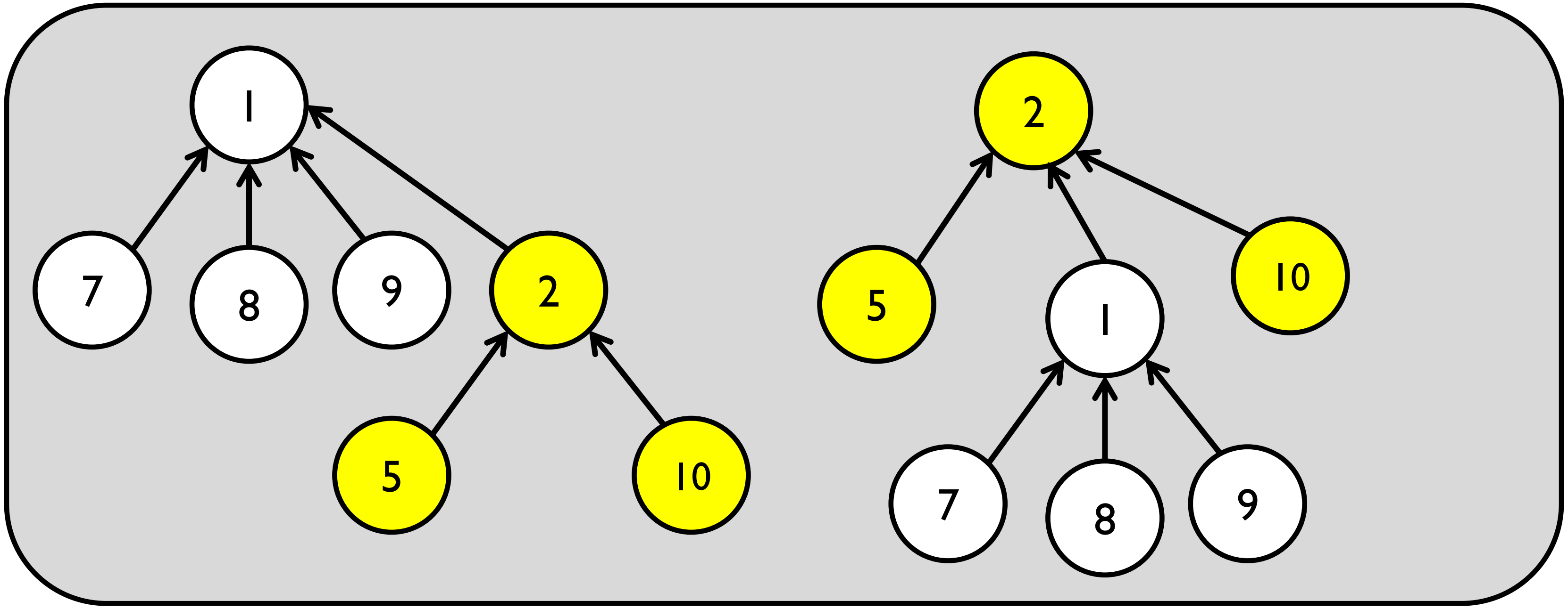
- **Disjoint Union:** $S_i \cup S_j$ = {all elements x such that x is in S_i or S_j}.
- **Find(i):** find a set containing element i.
- Challenge is to device data representation for disjoint sets so that these operation can be performed efficiently.
- One possible representation of S_1, S_2, S_3 can be given by



- ***Union:*** Make one of the trees a subtree of the other



- ***Union:*** Make one of the trees a subtree of the other



- In order to find Union of two sets all one has to do is, set the parent field of the root to the other root.

- We identify the sets by the index of roots.

- The operation F(i) will find the root of the tree containing element i. U(i,j) require two trees with roots i, j to be joined.

procedure U(i,j);

var i,j : integer;

begin

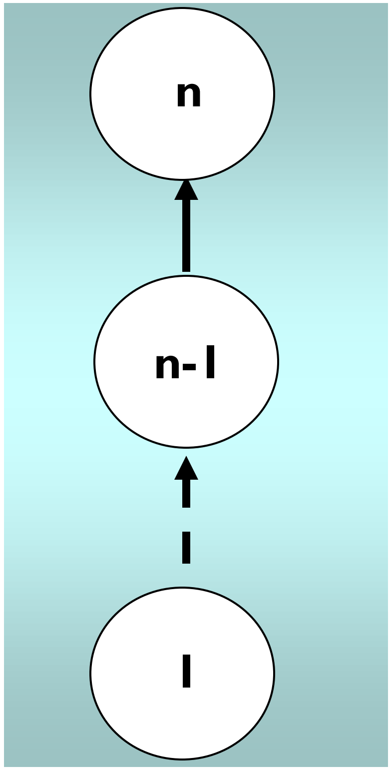
parent(i) ← j;

end.

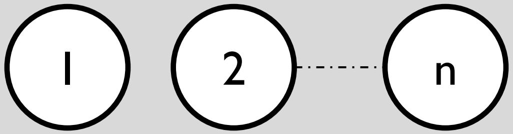
```
procedure F(i);
var i,j : integer;
begin
    j←i;
    While (parent(j) > 0) do
        j ← parent(j)
    return(j)
end.
```

- performance of U and F is **not good**, for ex. $S_i = \{i\}$, $1 \le i \le n$, then there is forest of n nodes with parent(i) = 0, $1 \le i \le n$.

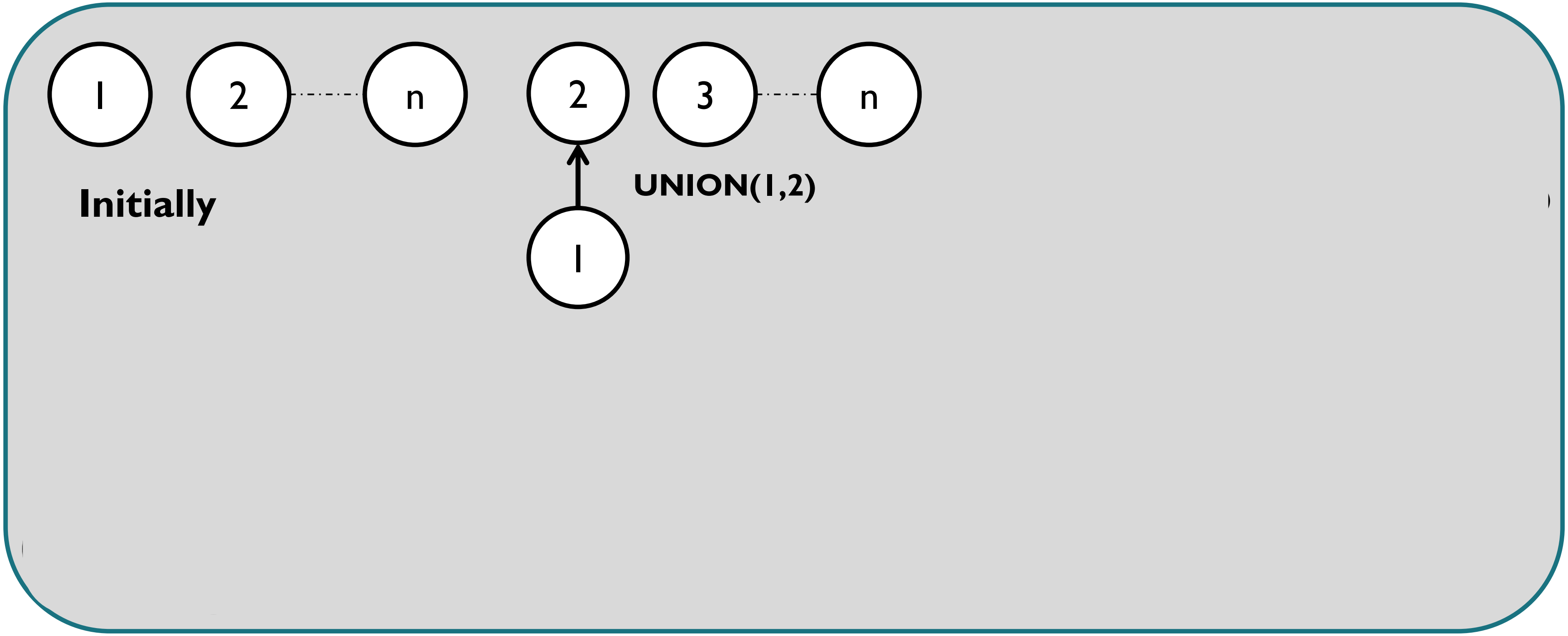
- If we perform these sequence of Union and Find-
- U(1,2), F(1), U(2,3), F(1), U(3,4), F(1),........, U(n-1,n).
- results in the degenerate tree.

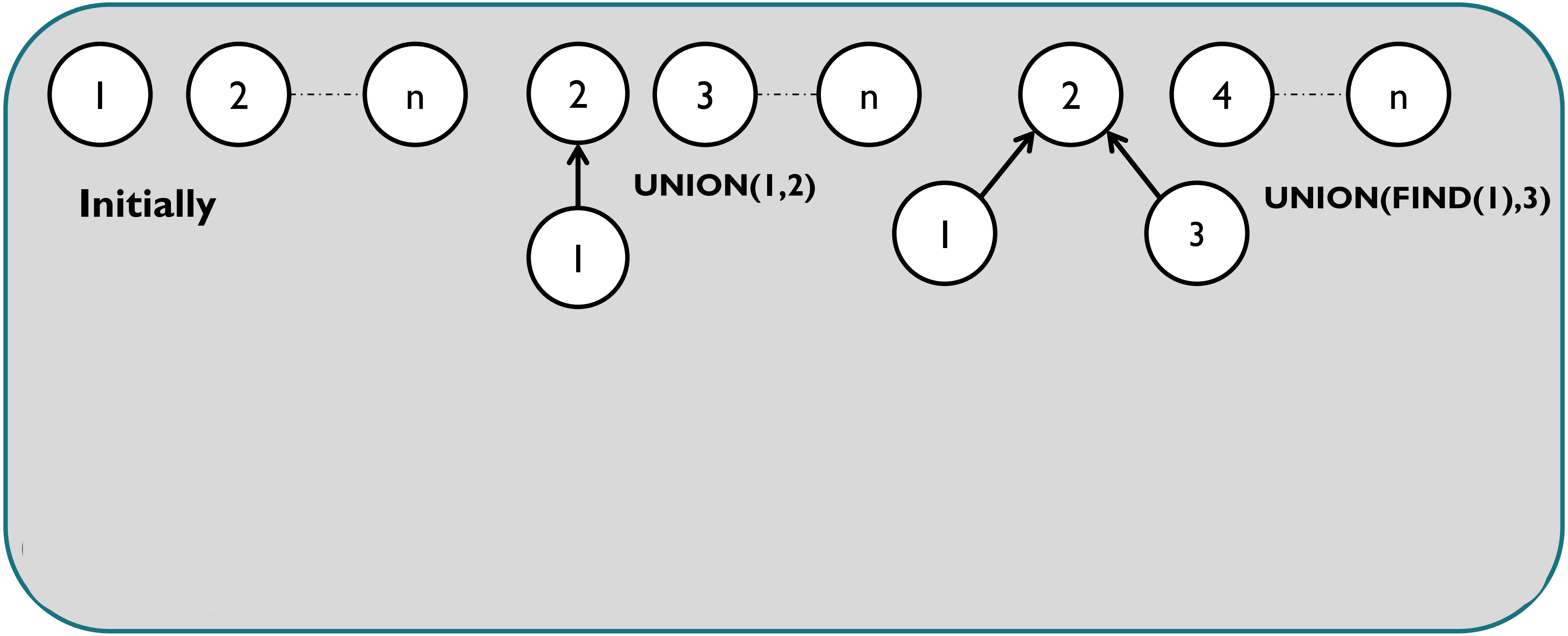


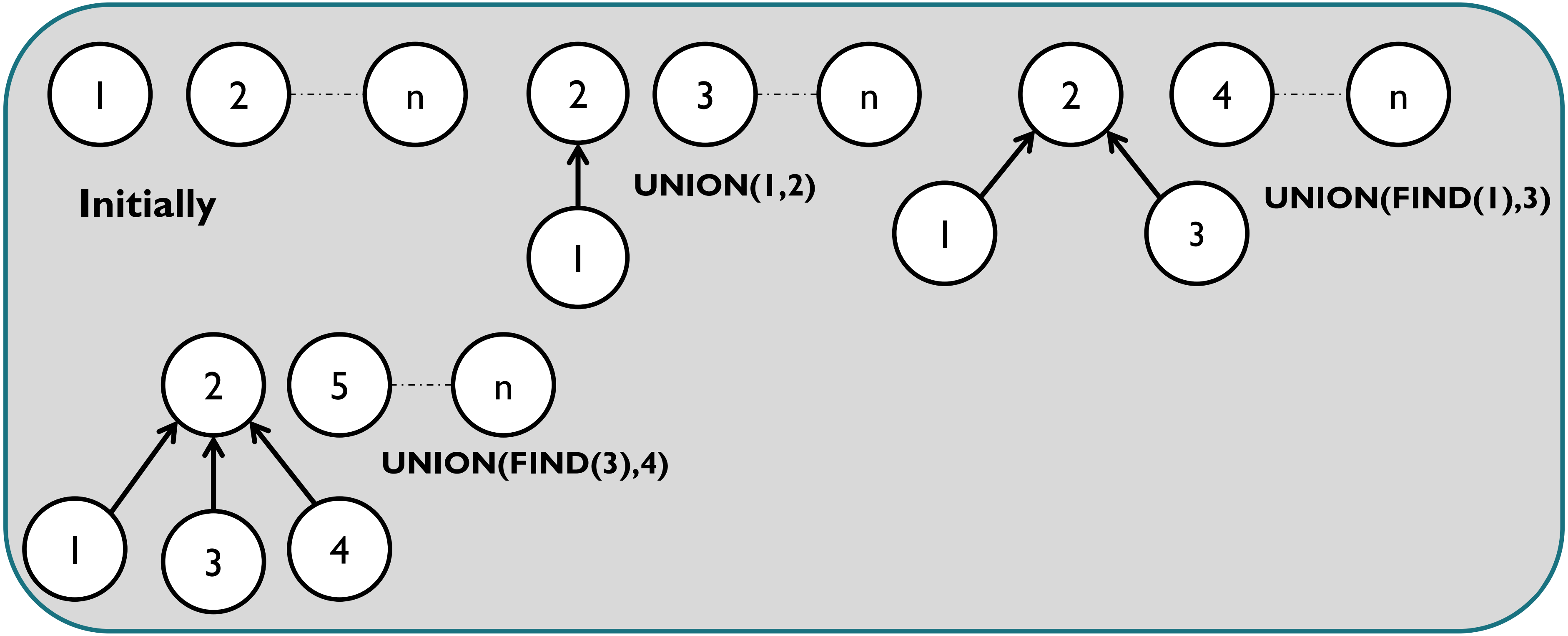
- Since the time taken for union is constant, n-1 calls to U can be processed in $O(n)$ time. Time required to process F at level i is $O(i)$.
- n-2 calls of find takes $O(n^2)$ time.
- ***Weighting rule: "if the number of nodes in tree i is less than the number of nodes in tree j, then make j the parent of i, else make i, the parent of j".***
- using the rule on the data set given earlier, and using the same sequence of operations we have-

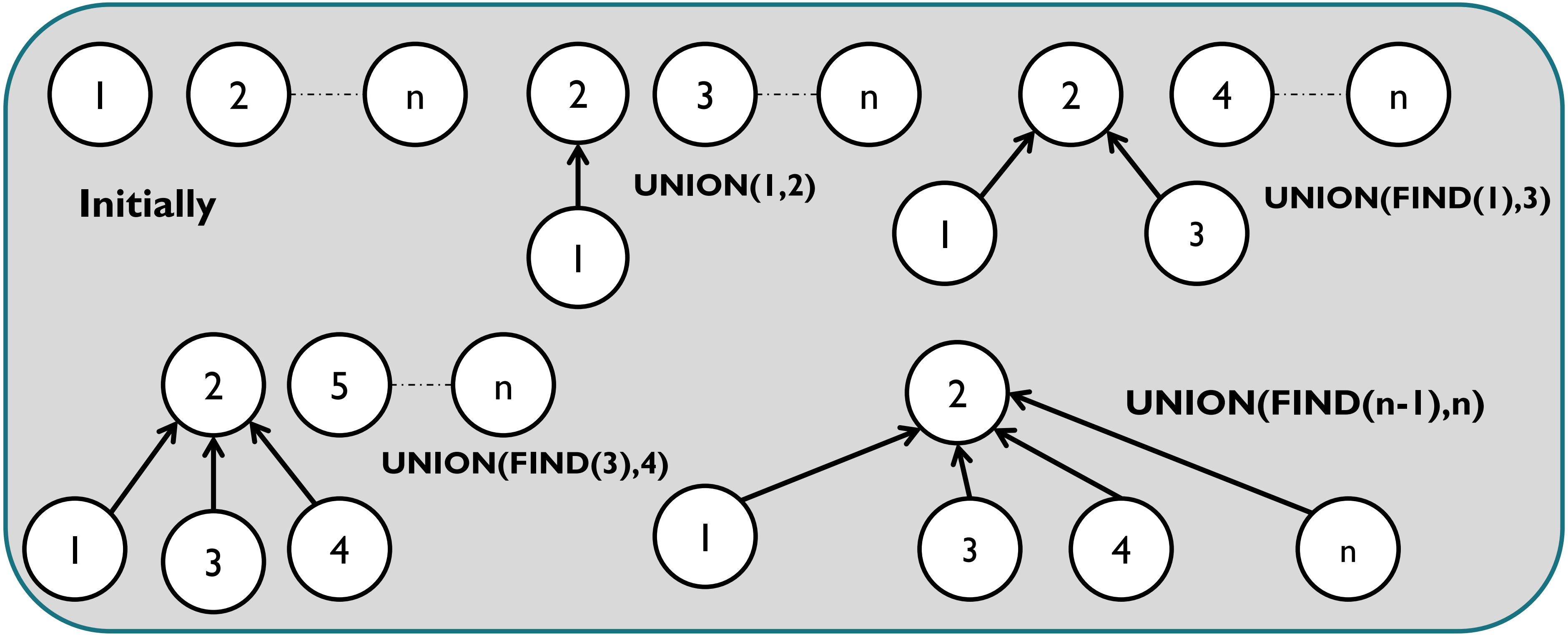


Initially


1
2
n
2
3
n
UNION(1,2)
Initially
1


1
2
n
Initially
2
3
n
1
UNION(1,2)
2
4
n
1
3
UNION(FIND(1),3)


1
2
n
Initially
2
1
3
n
UNION(1,2)
2
1
3
4
n
UNION(FIND(1),3)
2
1
3
4
5
n
UNION(FIND(3),4)



- In order to ***implement the weighting rule***, we need to know how many nodes are there in a tree. We maintain a COUNT field in the root of every tree.

- If i is the root of the tree then COUNT(i) = number of nodes in that tree. COUNT can be maintained in the PARENT field as a negative number.
- Because we can **store both parent and size in the same array** without needing a separate data structure.
- The PARENT **array** stores either:
  - A **positive number** = the index of the parent node
  - A **negative number** = means **this is a root**, and the absolute value is the **size of the tree.**

```
procedure UNION(i,j);
//PARENT(i) = -COUNT(i), PARENT(j) = -COUNT(j), //
var
i,j,x: integer;
begin
        x ← PARENT(i) + PARENT(j);
        if (PARENT(i) > PARENT(j)) then
                PARENT(i) ←j;
                PARENT(j) ←x;
        else
                PARENT(j) ←i;
                PARENT(i) ←x;
end.
```

- You have to trace it as an exercise
- Initially, each node is its own root:
  PARENT[i] = -1 for all i = 1 to 8

- Time required by UNION is still bounded by constant. The maximum time required by FIND is given by the lemma-

- ***Lemma:*** Let T be a tree with n nodes created as a result of algorithm UNION. No node in the tree has a level greater than $\lfloor \log n \rfloor + 1$.

Proof: Theorem is true for n = 1, assume that it is true for all trees with i nodes, $i \le n-1$. We show that it is true for i = n. Consider the last operation performed, UNION(k,j). Let m be number of nodes in tree j and n-m are number of nodes in k. We may assume $1 \le m \le n/2$. The maximum level of any node in T is

- **either is same as that in k or**
- **is one more than that in j**

- If first is the case then maximum level

$$T \leq \lfloor \log(n-m) \rfloor + 1$$

$$\leq \lfloor \log n \rfloor + 1$$

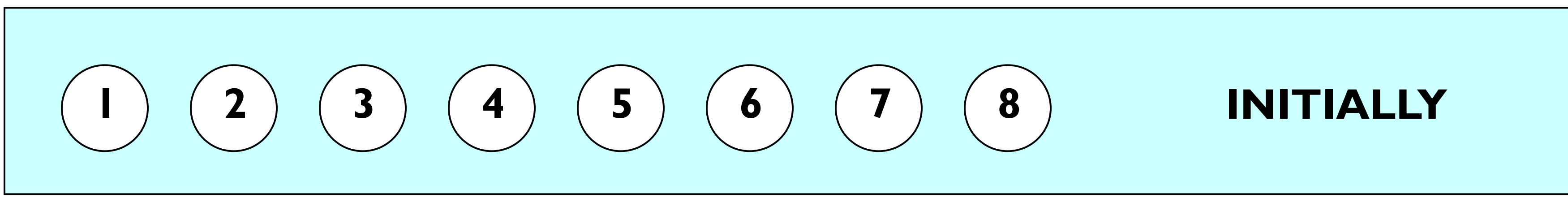
- later is the case than it is

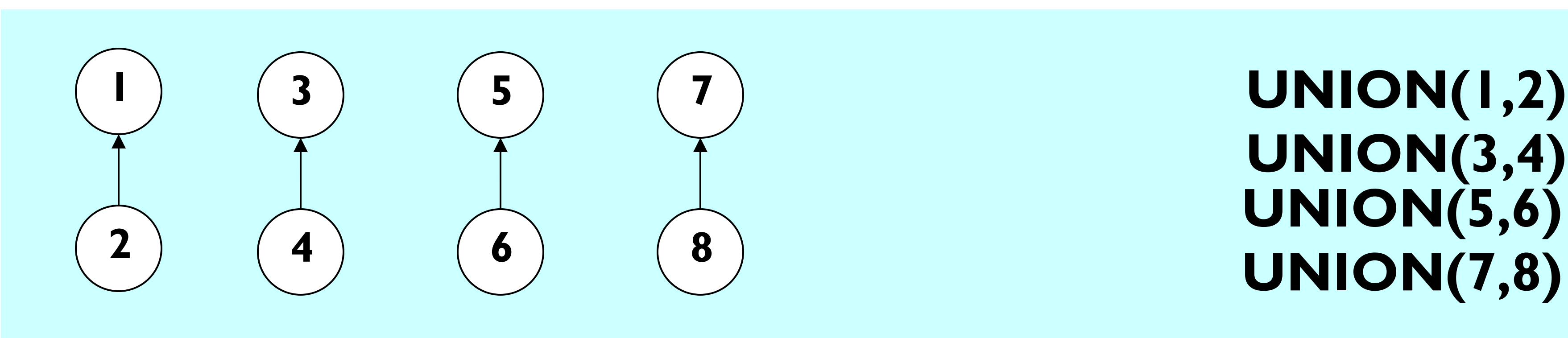
$$\leq \lfloor \log m \rfloor + 1 + 1$$

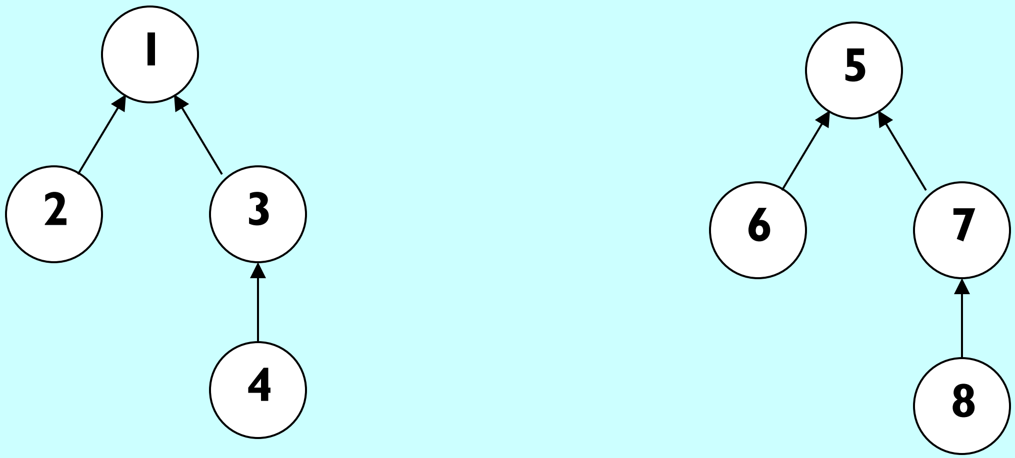
$$\leq \left\lfloor \log \frac{n}{2} \right\rfloor + 1$$

$$\leq \lfloor \log n \rfloor + 1$$

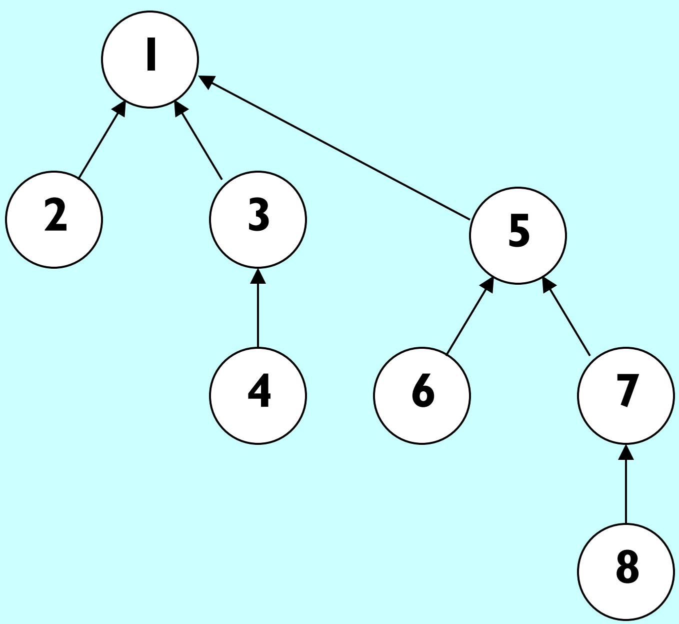
- Action of **UNION(1,2), UNION(3,4), UNION(5,6), UNION(7,8),**
  UNION(1,3), UNION(5,7),
  UNION(1,5)







UNION(1,3)
UNION(5,7)



UNION(1,5)

- As a result of the lemma, the maximum time to process a find is *O*(log n).
- If there are n elements in the tree sequence of ***n*** unions and ***m*** finds is bounded by *O*(m log n).
- Further improvement is possible if we make use of collapsing rule.
- **Collapsing rule:** ***If j is the node on the path from i to its root then set PARENT(j) ← ROOT(i).***

```
procedure FIND(i);
var j:integer;
begin
        j← i;
        while( PARENT(j) > 0) do
                j ← PARENT(j);
        k ← i;
        while (k ≠ j) do
                t ← PARENT(k);
                PARENT(k) ← j;
                k ← t;
        return(j);
end.
```

- processing FIND(8), FIND(8), FIND(8), FIND(8)
  FIND(8), FIND(8), FIND(8), FIND(8)

- using old F(8) we need 24 moves and with FIND we need 13 moves.

Example use

- Kruskal's Algorithm – Minimum Spanning Tree (MST)
- Cycle Detection in Undirected Graph
- Connected Components in Undirected Graph
- Dynamic Connectivity (offline queries)
- Network Connectivity Tracking
- Grouping Social Network Users / Friend Circles
- Merging Accounts, Names, or Labels
- Image Processing – Connected Component Labeling
- Equivalence of Equations (e.g., a == b, b == c, check a == c)

Equivalence Relations

- A relation R is defined on a set S if for every pair of elements (a, b) with $a, b \in S$, **a R b is either true or false**. If $a\ R\ b$ is true, we say that "a is related to b".
- An equivalence relation is a relation R that satisfies three properties
  - (Reflexive) $a\ R\ a$ for all $a \in S$
  - (Symmetric) $a\ R\ b$ if and only if $b\ R\ a$
  - (Transitive) $a\ R\ b$ and $b\ R\ c$ implies that $a\ R\ c$

Equivalence Relation Examples

- "=", but not "≤"
- Students with the same CGPA
- All cities in the same country
- Computers connected in a network

Equivalence Classes

- The equivalence class for an element $a \in$ S is the subset of S that contains all the elements that are related to a.
- The subsets that represent the equivalence classes will be "disjoint"
- Example
  - All students in Algorithms who are from Ph D

Examples

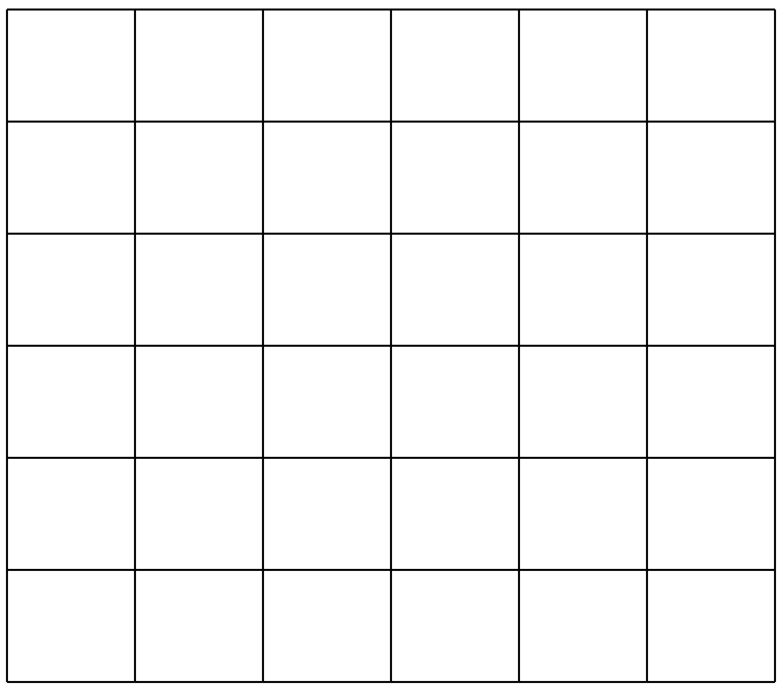
- **Same Birth Year (Students in Algorithms Class)**
- **Equivalence Relation**: a ~ b if both were born in the same year.
- **Equivalence Classes**:
  - 2004: {Rahul, Anjali, Neha}
  - 2005: {Sohan, Ria}
  - 2006: {Deepak}

Examples

- **Anagram Groups (Strings)**
- **Equivalence Relation:** Two strings are equivalent if they are anagrams of each other.
- **Equivalence Classes:**
  - {listen, silent, enlist}
  - {rat, tar, art}
  - {dusty, study}

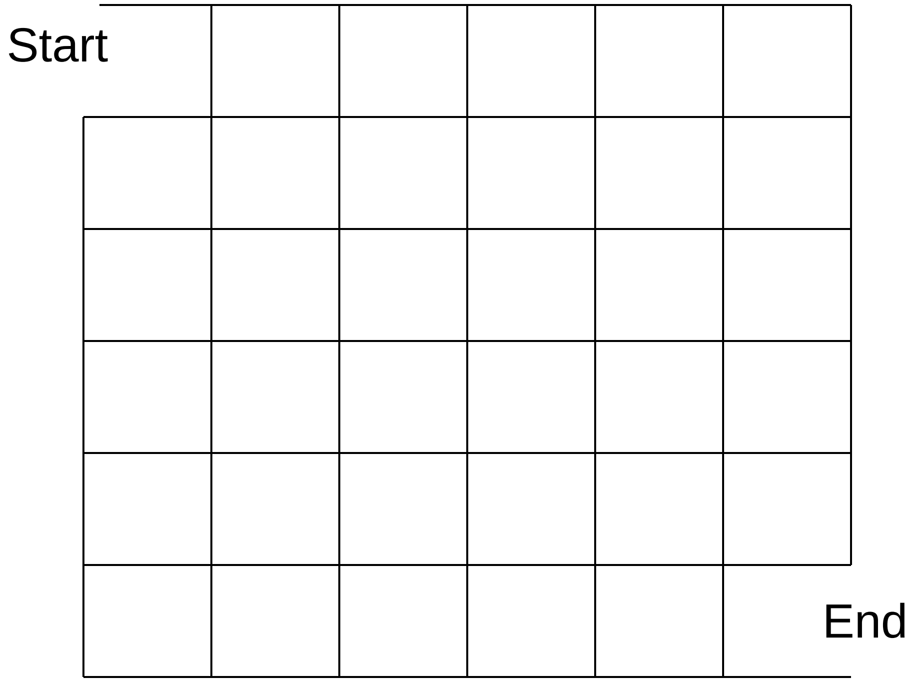
Maze Application

- Build a random maze by erasing edges.



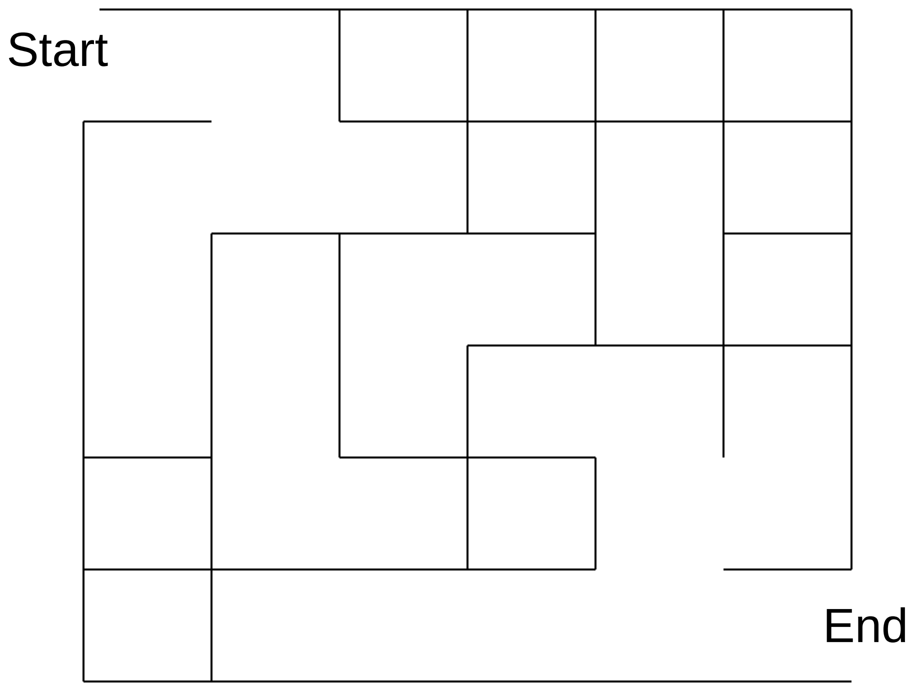
Maze Application

- Pick Start and End



Maze Application

- Repeatedly pick random edges to delete.



Equivalence Relations

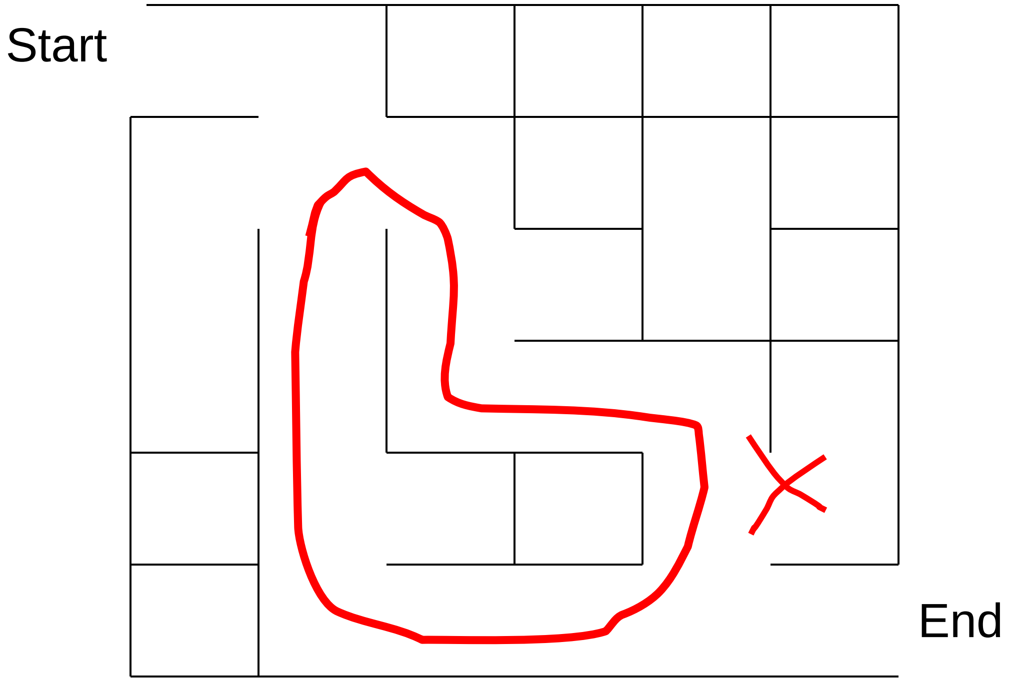
Connection between rooms is an equivalence relation

- any room is connected to itself
- if room **a** is connected to room **b**, then room **b** is connected to room **a**
- if room **a** is connected to room **b** and room **b** is connected to room **c**, then room **a** is connected to room **c**

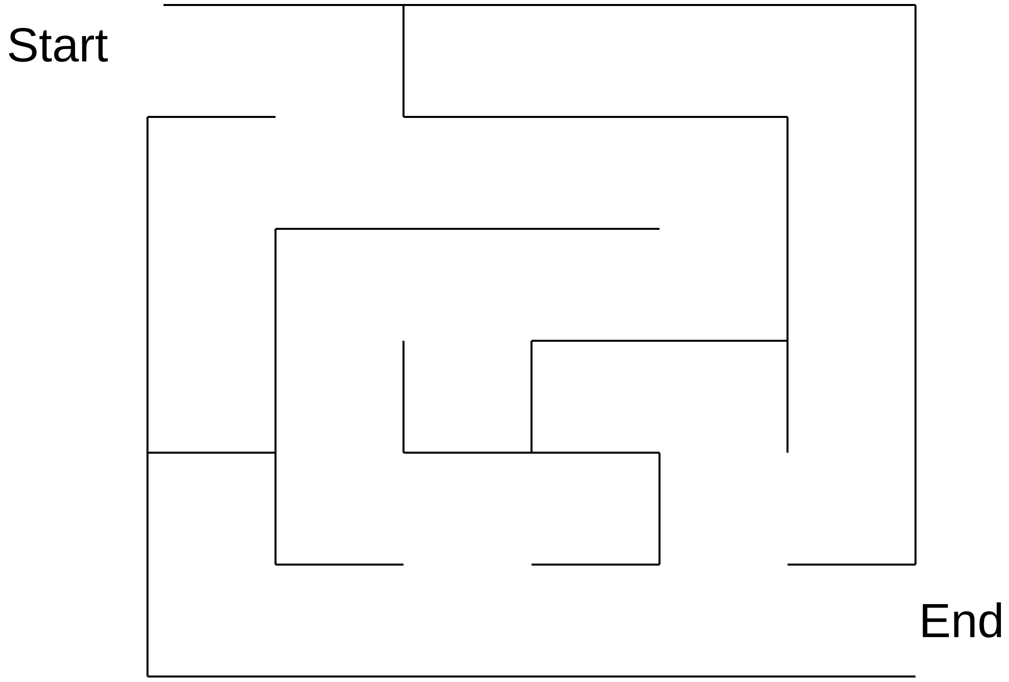
Maze Generator

- None of the boundary is deleted
- Randomly remove walls until the Start and End cells are in the same set.
- Removing a wall is the same as doing a union operation.
- Do not remove a randomly chosen wall if the cells it separates are already in the same set.
- There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.

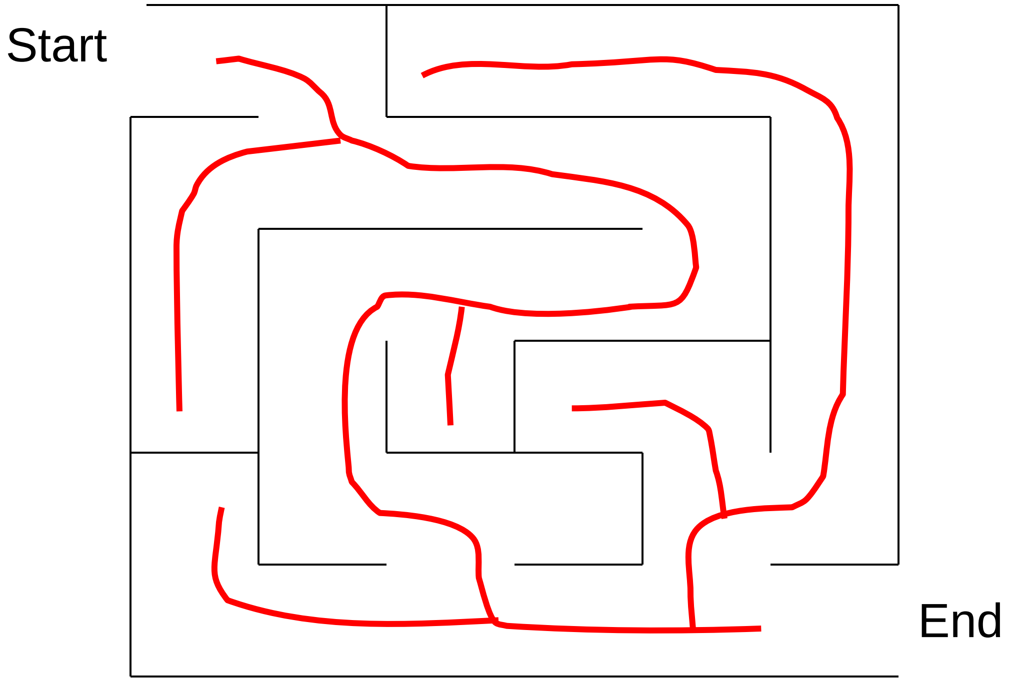
A Cycle



A Good Solution



A Hidden Tree



Number the Cells

We have disjoint sets S ={ {1}, {2}, {3}, {4},… {36} } each cell is unto itself.
We have all possible edges E ={ (1,2), (1,7), (2,8), (2,3), … } 60 edges total.

Start

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

End

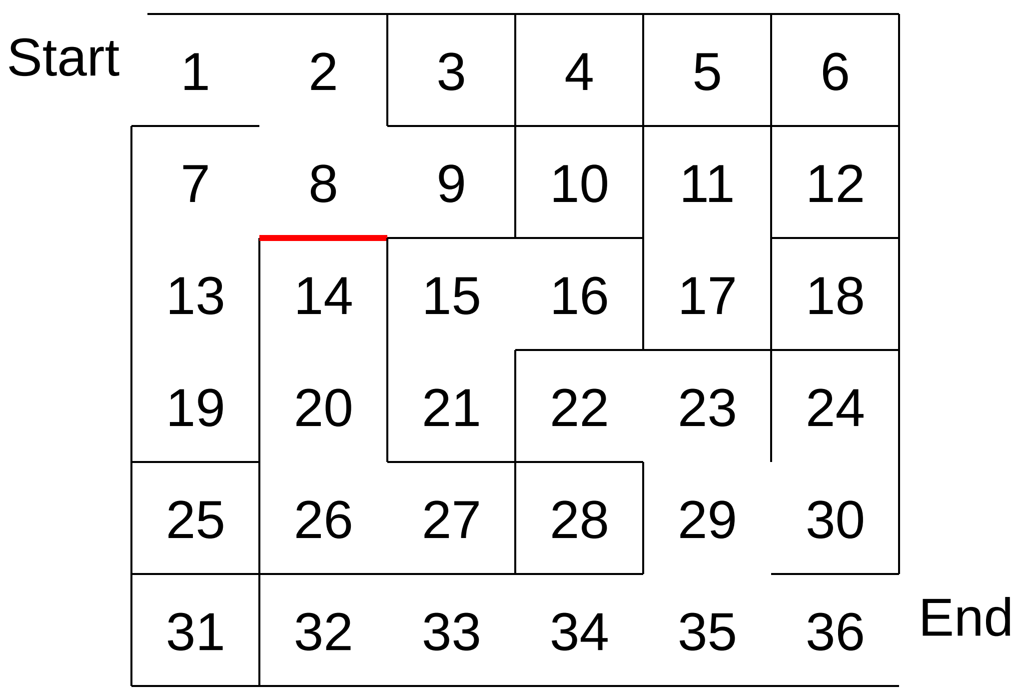
Basic Algorithm

- S = set of sets of connected cells
- E = set of edges
- Maze = set of maze edges initially empty

```
While there is more than one set in S
   pick a random edge (x,y) and remove from E
   u ← Find(x);
   v ← Find(y);
   if u ≠ v then
      Union(u,v)
   else
       add (x,y) to Maze
All remaining members of E together with Maze form the maze
```

Example Step

Pick (8,14)



S

{1,2,7,8,9,13,19}

{3}

{4}

{5}

{6}

{10}

{11,17}

{12}

{14,20,26,27}

{15,16,21}

.

.

{22,23,24,29,30,32
33,34,35,36}

Example

S
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

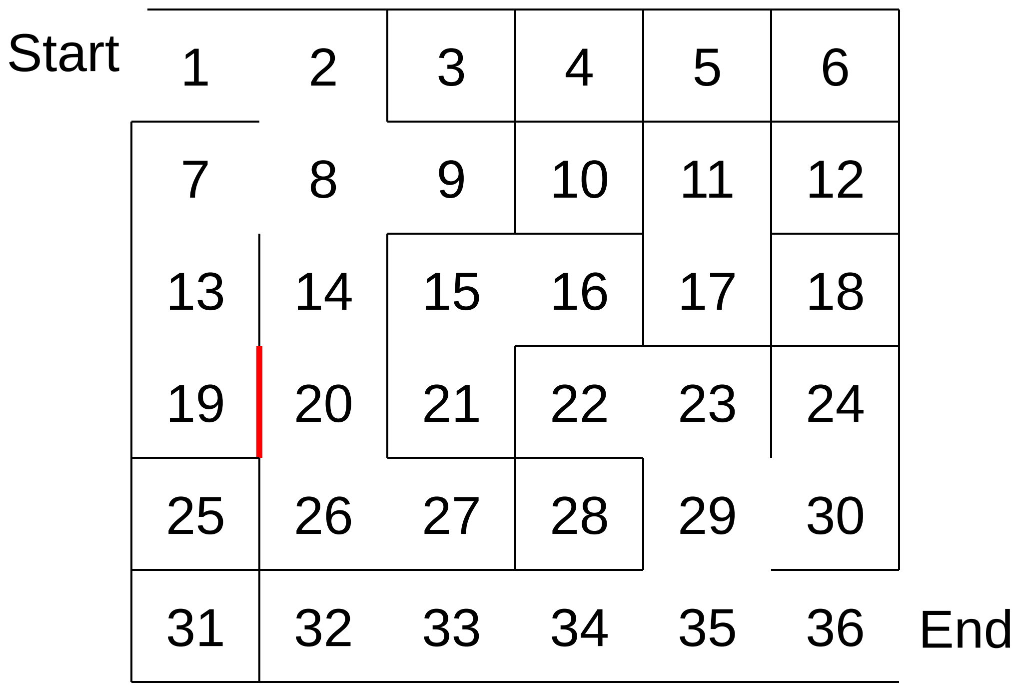
Find(8) = 7
Find(14) = 20

Union(7,20)

S
{1,2,7,8,9,13,19,14,20 26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Example

Pick (19,20)



S

{1,2,7,8,9,13,19
14,20,26,27}

{3}

{4}

{5}

{6}

{10}

{11,17}

{12}

{15,16,21}

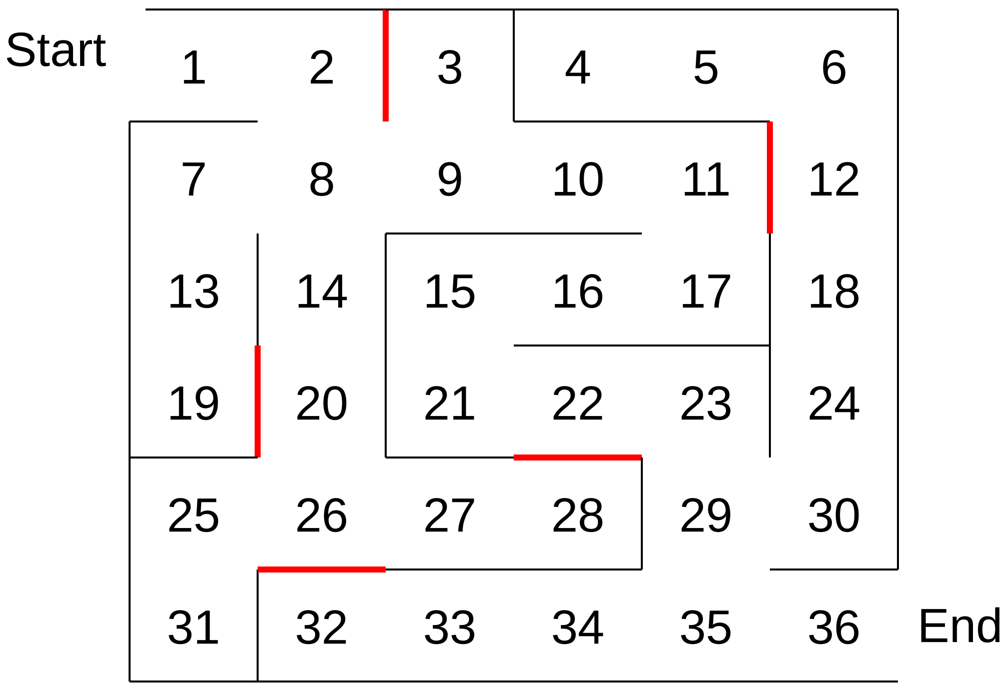
.

.

{22,23,24,29,39,32
33,34,35,36}

Example at the End

S
{1,2,3,4,5,6,7,… 36}



E
Maze

End of Chapter