

A Tutorial on- Disjoint Sets

Algorithms – 1, Autumn 2021
Presented by- Siddharth D Jaiswal, CSE, IIT Kgp
[siddsjaiswal@kgpian.iitkgp.ac.in]

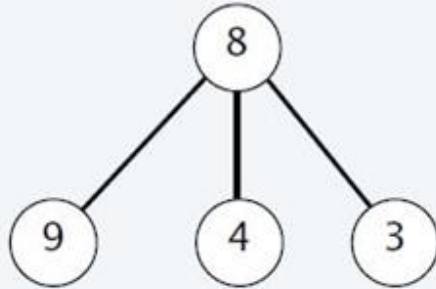
Agenda

- Union By Rank
- Path Compression
- Problem Statement from Autumn, 2020
- Solution
- Oral Discussion

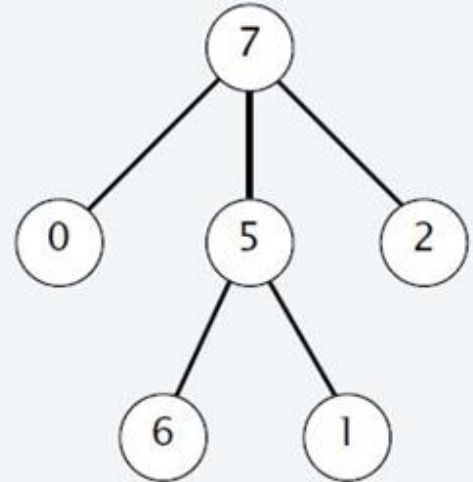
Union By Rank

UNION(5, 3)

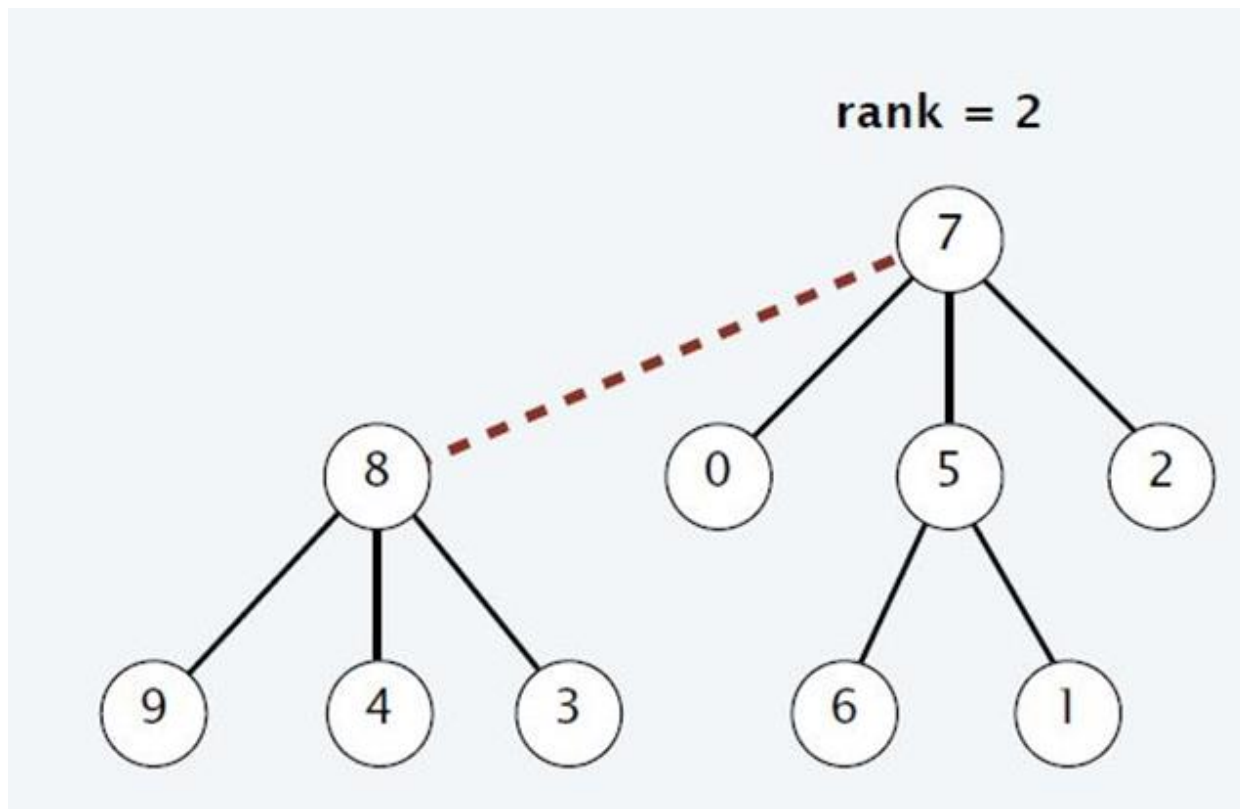
rank = 1



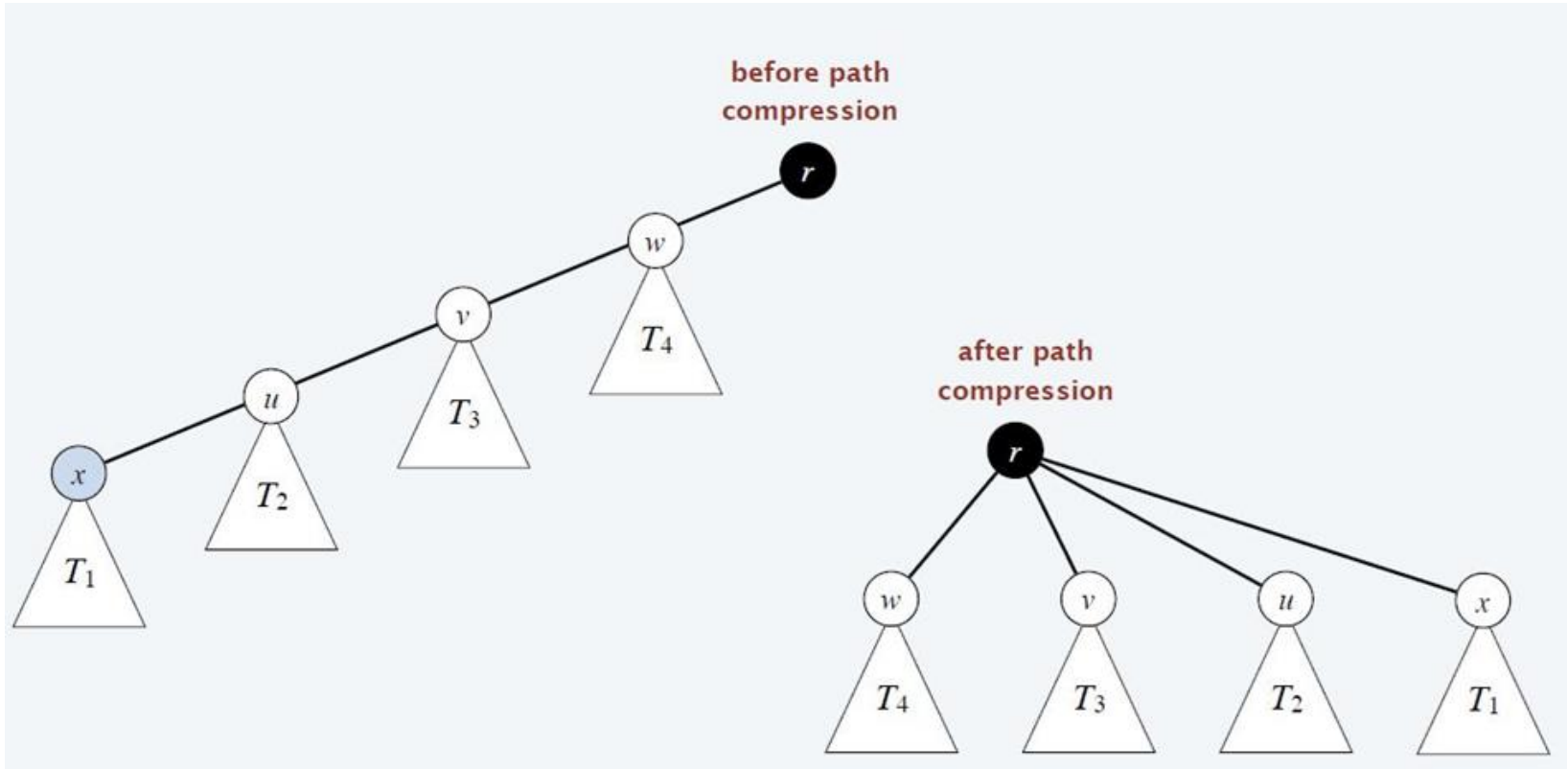
rank = 2



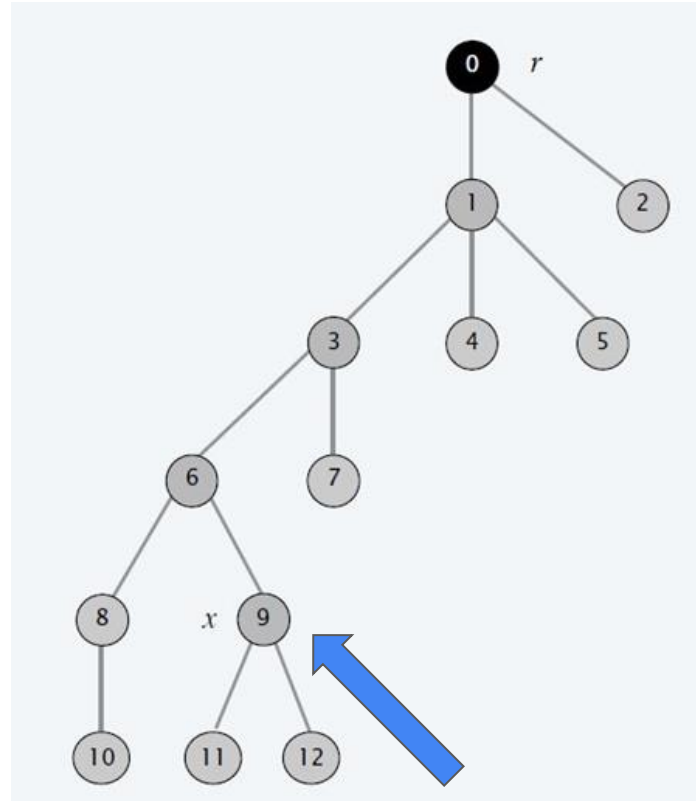
Union By Rank



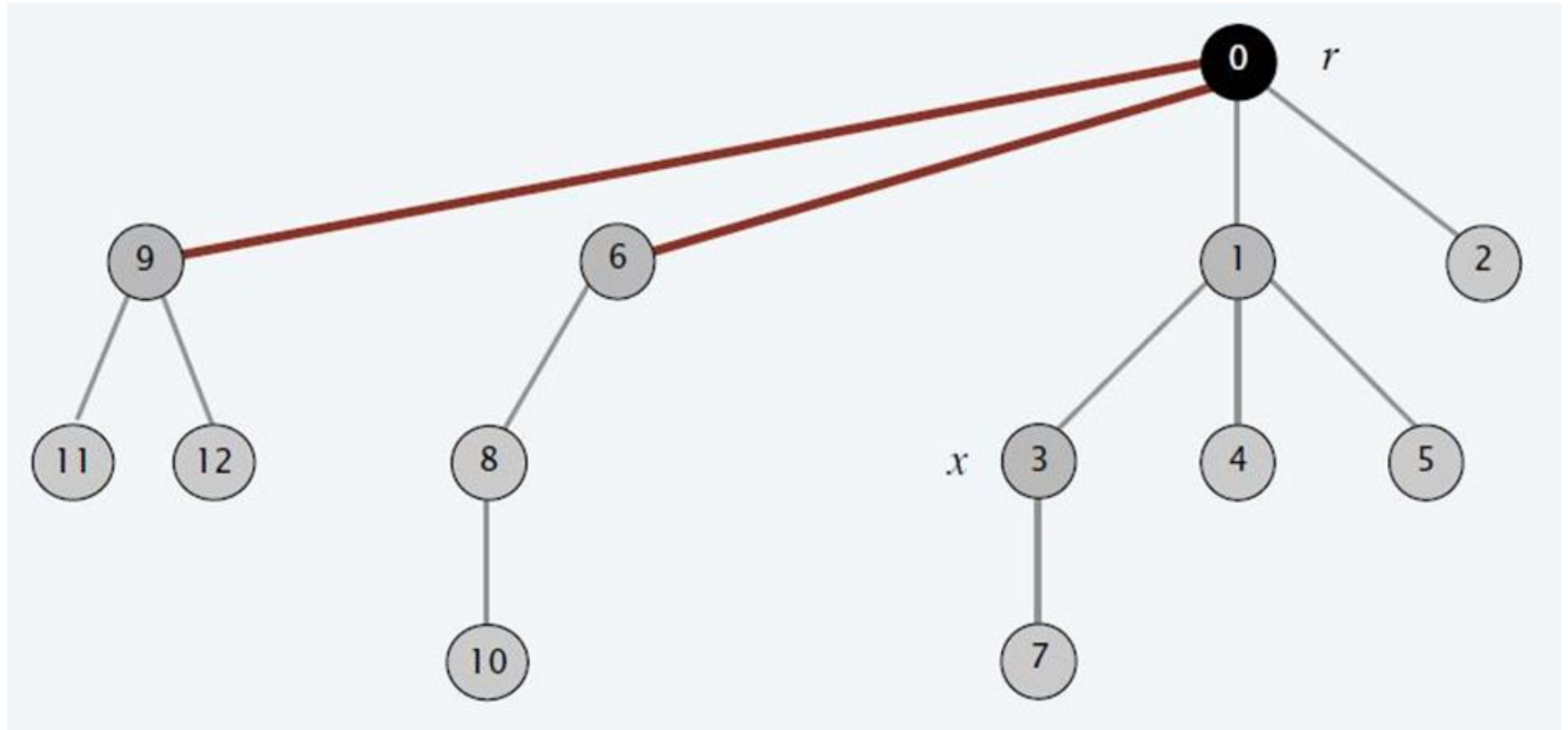
Path Compression



Path Compression



Path Compression



Code for Union by Rank and Path Compression

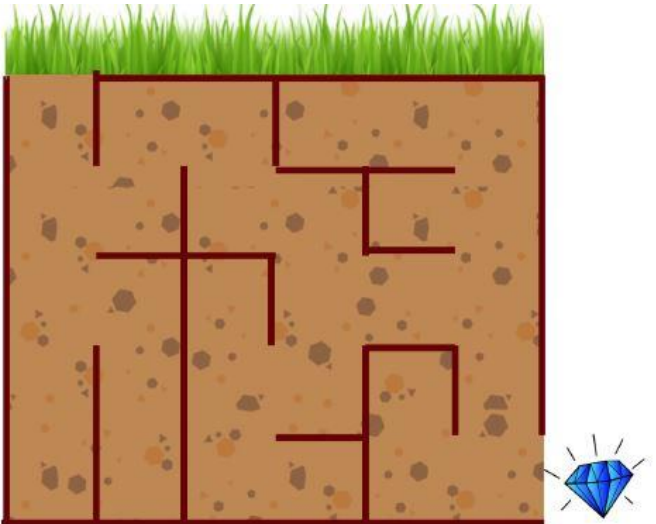
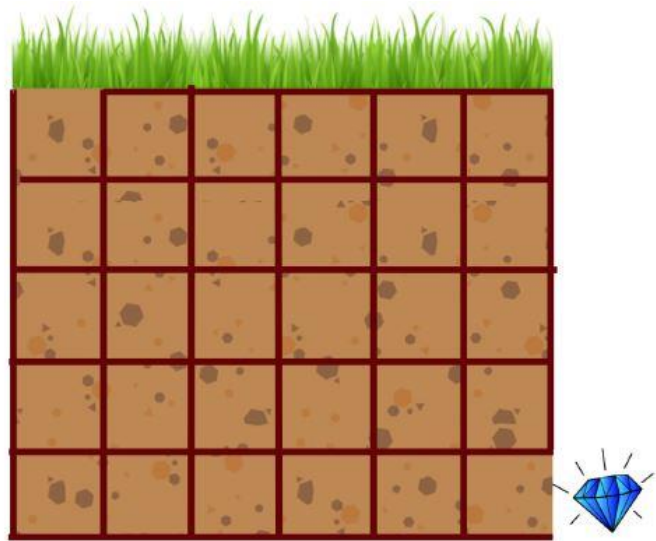
```
//PATH COMPRESSION
node *findset ( node *x)
{
    if (x -> parent != x)
        x->parent = findset(x->parent);
    return x->parent;
}
```

```
int mergeset (node *x, node *y )
{
    if (x == y) return 0;
    if (x -> rank > y -> rank)
        y -> parent = x;
    else if (x -> rank < y -> rank)
        x -> parent = y;
    else
    {
        x -> parent = y;
        y -> rank += 1;
    }
    return 1;
}
```


Problem Statement from Autumn 2020

Mr. Scrooge is a renowned treasure hunter. Recently, he came across **a map which shows possible treasure** buried deep inside the earth. The place has **$m \times n$ chambers with stone walls between every two adjacent chambers**. The map also contains a **spell** which can be used to remove the stone walls, but it can only be **used at most $(m \times n - 1)$ times**. The most precious treasure is located just beside the last chamber. Mr. Scrooge not only wants to find the treasure, but he **also wants to explore each of the chambers** since he thinks there might be some valuables in other chambers as well. He does not know which stone walls have to be removed to access each of the chambers. If he cannot find the treasure **within $(m \times n - 1)$ stone wall removals**, the spell will lose its potential. **Luckily, Mr. Scrooge knows about the disjoint set data structure which he can effectively apply to solve this problem.**

Problem Statement from Autumn 2020



Problem Statement from Autumn 2020

You can think of the chambers like **a grid**. The above figure illustrates the initial and the final grid. In the initial grid, all the **chambers are separated by stone walls**. There is **one entry point** in the top left of the grid and the treasure is right beside the bottom right chamber. In the final grid, Mr. Scrooge has successfully found the treasure and all the chambers are accessible from the start chamber.

Stone walls should be removed to achieve the following characteristics:

- Walls should be **selected randomly** as candidates for removal.
- A stone wall **should not be removed if the two chambers on either side of the wall are already connected by some other path**.
- The chambers are fully connected, i.e., **every chamber is reachable from the start chamber**.

Problem Statement from Autumn 2020

Write the following three functions for implementing the disjoint-set data structure with union-by-rank and path compression.

- **makeset** : Initially, the chambers are isolated from one another, i.e., each chamber belongs to a singleton set. Create an **$m \times n$ array C of nodes**. Let the parent pointer of each node point to itself. Also set the rank field of each node to zero.
- **findset** : Given **a node, i.e., (i, j) -th chamber, locate and return the root of the tree** (connected area) to which the (i, j) -th chamber belongs. Also **apply path compression technique while finding the root**.
- **mergeset** : Given two distinct **root pointers x and y** , make the **root of the smaller tree a child of the root of the larger tree**. If both roots have the same rank, increase rank of new root by one.

To randomly select walls for removal, you will also **need to maintain a separate list of walls eligible for removal**. There are **two types of walls : horizontal walls and vertical walls**. Create an **$(m - 1) \times n$ array H** for the horizontal walls, and an **$m \times (n - 1)$ array V** for the vertical walls. Note that the **outer walls need not be removed**, so these arrays do not consider outer walls.

Problem Statement from Autumn 2020

Part-1

Initialize C , H , and V corresponding to the chambers, horizontal wall, and vertical walls, respectively. Implement the disjoint-set data structure. Use the following definition for a node.

```
typedef struct _node {  
    int rank;  
    struct _node *parent;  
} node;
```

Part-2

Write a function *findtreasure* that removes $mn - 1$ stone walls such that all the chambers (including the first and the final chambers) are in the same set.

At this point, call *findset* for the first and final chambers and verify that these are in the same set. Print appropriate message in console.

Part-3

Write a function *printgrid* for printing the initial and final grid. The grid is printed in ASCII using the vertical bar (|) and dash characters (---) to represent stone walls, and plus (+) for corners. The start and end chambers should have exterior openings as shown in the sample output.

A deeper look at the problem- Part 1

Part-1

Initialize C , H , and V corresponding to the chambers, horizontal wall, and vertical walls, respectively. Implement the disjoint-set data structure. Use the following definition for a node.

```
typedef struct _node {  
    int rank;  
    struct _node *parent;  
} node;
```

makeset : Initially, the chambers are isolated from one another, i.e., each chamber belongs to a singleton set. **Create an $m \times n$ array C of nodes.** Let the **parent pointer of each node point to itself**. Also set the **rank field of each node to zero**.

A deeper look at the problem- Part 1

1. Create a 2-D array of the nodes.
 - a. Set rank of each node as 0
 - b. Set parent of each node to point to its own address.
2. Create horizontal walls array*
3. Create vertical walls array*
4. Print the grid!

* Any guesses on the data types of the nodes?

** VERIFY EACH FUNCTION'S CORRECTNESS BEFORE MOVING ON TO THE NEXT ONE!

Solution- Part 1 [main()]

```
int main ( int argc, char *argv[] )
{
    int m, n;
    node **C;
    char **H, **V;
    int i,j,u,v;
    node *x,*y;

    if (argc >= 3) {
        m = atoi(argv[1]);
        n = atoi(argv[2]);
    }
    else {
        scanf("%d%d", &m, &n);
    }

    printf("    m = %d\n    n = %d\n", m, n);

    srand((unsigned int)time(NULL));

    C = makeset(m,n);
    H = inithorwalls(m,n);
    V = initverwalls(m,n);
```



Solution- Part 1 [makeset()]


```
node **makeset ( int m, int n )
{
    node **C;
    int i, j;

    C = (node **)malloc(m * sizeof(node *));
    for (i=0; i<m; ++i) {
        C[i] = (node *)malloc(n * sizeof(node));
        for (j=0; j<n; ++j) {
            C[i][j].rank = 0;
            C[i][j].parent = &(C[i][j]);
        }
    }
    return C;
}
```



Solution- Part 1 [inithorwalls() & initverwalls()]

```
#define CLOSED 0  
#define OPEN 1
```



```
char **inithorwalls ( int m, int n )  
{  
    int i, j;  
    char **H;  
  
    H = (char **)malloc((m-1) * sizeof(char *));  
    for (i=0; i<=m-2; ++i) {  
        H[i] = (char *)malloc(n * sizeof(char));  
        for (j=0; j<n; ++j) H[i][j] = CLOSED;  
    }  
    return H;  
}
```

```
char **initverwalls ( int m, int n )  
{  
    int i, j;  
    char **V;  
  
    V = (char **)malloc(m * sizeof(char *));  
    for (i=0; i<m; ++i) {  
        V[i] = (char *)malloc((n-1) * sizeof(char));  
        for (j=0; j<=n-2; ++j)  
            V[i][j] = CLOSED;  
    }  
    return V;  
}
```

Solution- Part 1 [printgrid()]

```
void printgrid ( char **H, char **V, int m, int n )
{
    int i, j;

    printf("    +");
    printf("    +");
    for (j=0; j<n-1; ++j) printf("---+"); printf("\n");
    for (i=0; i<m; ++i)
    {
        // if (i==0)
        //     printf("    ");
        // else printf(" |");
        printf("    |");
        for (j=0; j<=n-2; ++j)

            printf("    %c", (V[i][j] == CLOSED) ? '|' : ' ');

        if (i==m-1)
            printf("    \n");
        else
            printf(" | \n");

        if (i != m-1) {
            printf("    ");
            for (j=0; j<n; ++j) printf("+%s", (H[i][j] == CLOSED) ? "---" : " ");
            printf("+\n");
        }
    }

    printf("    +"); for (j=0; j<n; ++j) printf("---+"); printf("\n");
}
```

A deeper look at the problem- Part 2

Part-2

Write a function *findtreasure* that removes $mn - 1$ stone walls such that all the chambers (including the first and the final chambers) are in the same set.

At this point, call *findset* for the first and final chambers and verify that these are in the same set. Print appropriate message in console.

- **findset** : Given a node, i.e., (i; j)-th chamber, locate and return the root of the tree (connected area) to which the (i; j)-th chamber belongs. Also **apply path compression technique while finding the root**.
- **mergeset** : Given two distinct **root pointers x and y**, make the **root of the smaller tree a child of the root of the larger tree**. If both roots have the same rank, increase rank of new root by one.

A deeper look at the problem- Part 2

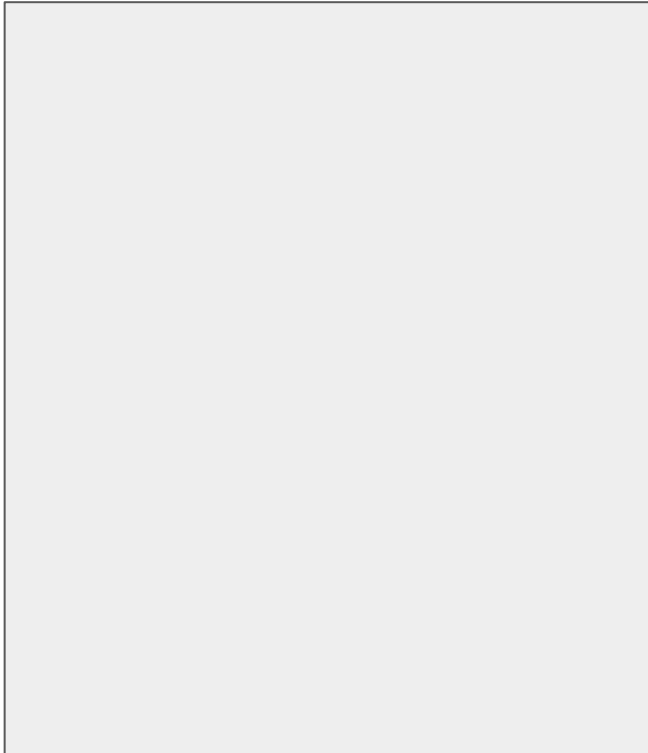
1. Create the findtreasure function
2. Within that, iterate until $(m \times n - 1)$ walls have not been removed
3. Choose a wall randomly
4. Call the findset function and open the wall*
5. Call the mergeset* function and join the different cells into a single path

* Any guesses on how findset and mergeset work here?

HINT: Recall findset and union function we just discussed

** VERIFY EACH FUNCTION'S CORRECTNESS BEFORE MOVING ON TO THE NEXT ONE!

Solution- Part 2 [main()]



```
printf("\n Initial grid\n");  
printgrid(H,V,m,n);
```

```
findtreasure(C,H,V,m,n);  
printf("\n Final grid\n");  
printgrid(H,V,m,n);
```

```
x = findset(&C[m-1][n-1]);  
y = findset(&C[0][0]);
```

```
if (x==y)  
|   printf("Found\n");  
else  
|   printf("NOT Found\n");  
exit(0);
```

```
}
```

Solution- Part 2 [findtreasure()]

```
void findtreasure ( node **C, char **H, char **V, int m, int n )
{
    int i1, i2, j1, j2, s, nmerge;
    node *x, *y;

    s = 1; nmerge = 0;

    while (s < (m*n)) {
        if (rand() % 2) {
            /* Break horizontal stone wall */
            i1 = rand() % (m-1);
            i2 = i1 + 1;
            j1 = j2 = rand() % n;

            if (H[i1][j1] == OPEN) {
                continue;
            }

            x = findset(&C[i1][j1]);
            y = findset(&C[i2][j2]);
            if (x == y) {
                continue;
            }
            H[i1][j1] = OPEN;
        } else {
```

```
        } else {
            /* Break vertical stone wall */
            i1 = i2 = rand() % m;
            j1 = rand() % (n-1);
            j2 = j1 + 1;

            if (V[i1][j1] == OPEN) {
                continue;
            }

            x = findset(&C[i1][j1]);
            y = findset(&C[i2][j2]);

            if (x == y) {
                continue;
            }

            V[i1][j1] = OPEN;
        }

        if (mergeset(x,y)==1)
        {s++;
        ++nmerge;
        }
    }

    printf("\n grid created after %d stone wall removals\n", nmerge);
}
```

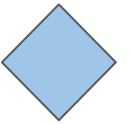
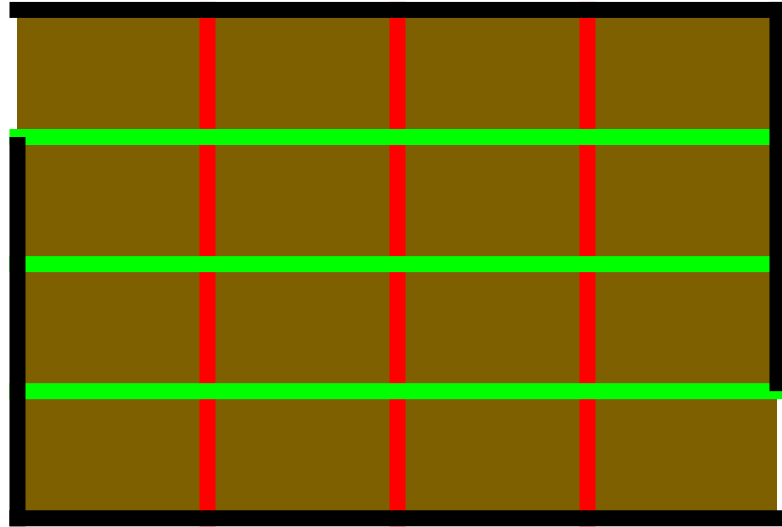
An example grid

$$m = 4,$$

$$n = 4$$


$$H = 12$$

$$V = 12$$

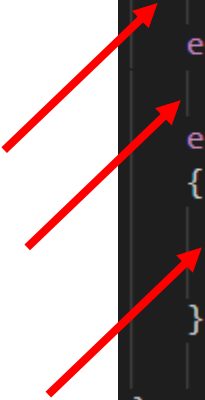


Solution- Part 2 [findset() & mergeset()]

```
//PATH COMPRESSION
node *findset ( node *x)
{
    if (x -> parent != x)
        x->parent = findset(x->parent);
    return x->parent;
}
```



```
int mergeset (node *x, node *y )
{
    if (x == y) return 0;
    if (x -> rank > y -> rank)
        y -> parent = x;
    else if (x -> rank < y -> rank)
        x -> parent = y;
    else
    {
        x -> parent = y;
        y -> rank += 1;
    }
    return 1;
}
```



An example grid- Breaking a horizontal wall

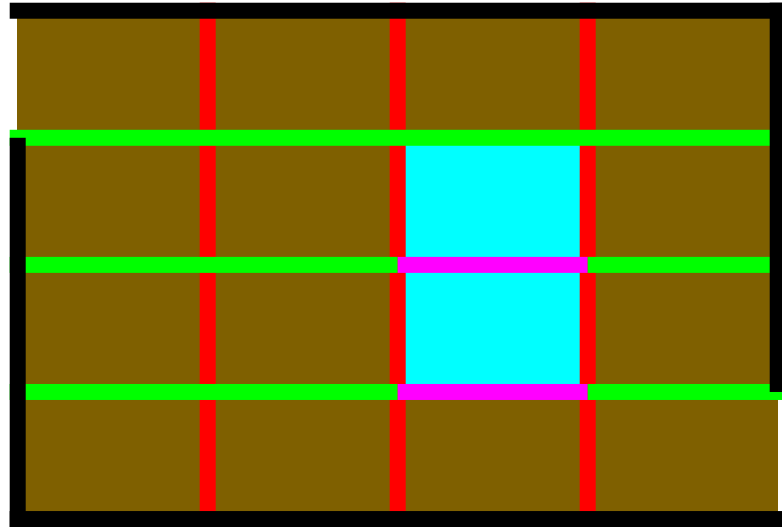
For---

$i1 = 1$

$j1 = 2$

$i2 = 2$

$j2 = 2$



An example grid- Breaking a horizontal wall

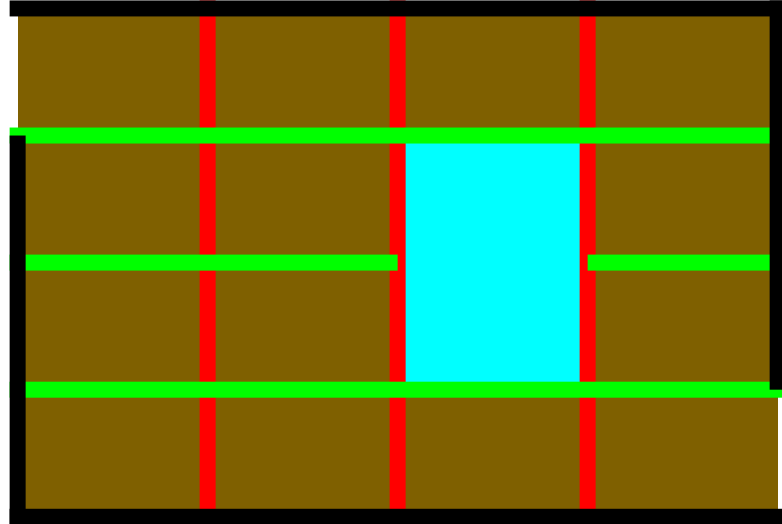
For---

$i1 = 1$

$j1 = 2$

$i2 = 2$

$j2 = 2$



A deeper look at the problem- Part 3

Part-3

Write a function *printgrid* for printing the initial and final grid. The grid is printed in ASCII using the vertical bar (|) and dash characters (---) to represent stone walls, and plus (+) for corners. The start and end chambers should have exterior openings as shown in the sample output.

```
+   +---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
```

Solution- Part 1 [printgrid()]

```
void printgrid ( char **H, char **V, int m, int n )
{
    int i, j;

    printf("    +");
    printf("    +");
    for (j=0; j<n-1; ++j) printf("---+"); printf("\n");
    for (i=0; i<m; ++i)
    {
        // if (i==0)
        //     printf("    ");
        // else printf(" |");
        printf("    |");
        for (j=0; j<=n-2; ++j)

            printf("    %c", (V[i][j] == CLOSED) ? '|' : ' ');

        if (i==m-1)
            printf("    \n");
        else
            printf(" | \n");

        if (i != m-1) {
            printf("    ");
            for (j=0; j<n; ++j) printf("+%s", (H[i][j] == CLOSED) ? "---" : " ");
            printf("+\n");
        }
    }

    printf("    +"); for (j=0; j<n; ++j) printf("---+"); printf("\n");
}
```

Some *other* things to take care of

- Be careful about the input and the output formats mentioned in the question. If in doubt, ask. If still in doubt, make assumptions and **mention them clearly**.
- Always, ALWAYS, **ALWAYS test each function's logic for correctness before moving on to the next one**.
- Implement more functions that do a single thing instead of having a single function that does a lot of things.
 - Don't shy away from writing more code, if it simplifies your and our understanding of your implementation logic.
- Finally, NEVER over-optimize your code. First ensure the correctness and only then focus on the optimization.

Thank You
Any further questions?