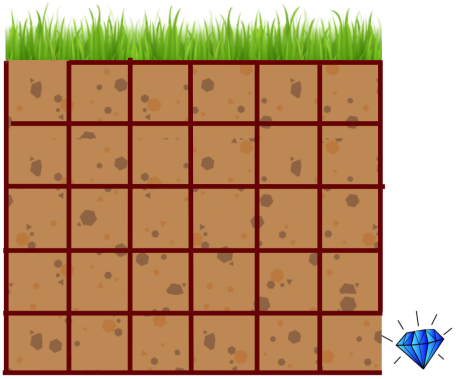
CS29003 ALGORITHMS LABORATORY
ASSIGNMENT 7 (Disjoint Set)
Date: 22nd October, 2020

Important Instructions

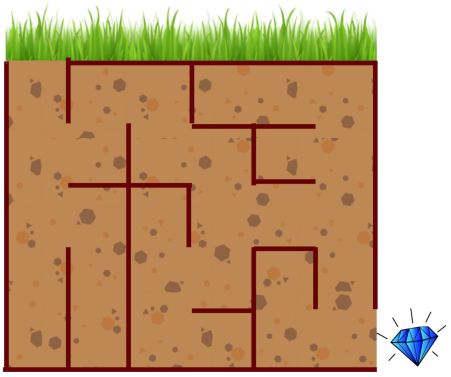
1. **File to be submitted:** ROLLNO_GroupNO_A7.c/.cpp
2. You are to **stick to the output formats strictly** as per the instructions.
3. Submission through **.zip files are not allowed**.
4. Write your name and roll number at the beginning of your program.
5. Do not use any global variable unless you are explicitly instructed so.
6. Use proper indentation in your code.
7. Please follow all the guidelines. Failing to do so will cause you to lose marks.
8. **There will be part marking.**

Problem Statement

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 $mn - 1$ times. The most precious treasure is located just beside the last chamber. Mr. Scrooge not only wants to find the treasure, 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 $mn - 1$ stone wall removal, 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.



(a) Initial grid



(b) Final grid after the stone-walls are removed

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.

Let C be a set of size s. Let $C_1, C_2, ..., C_t$ be pairwise disjoint subsets of C such that $C = \cup_{t=1}^{k} C_k$. For our application, C is the set of mn chambers, and C_k are the connected chambers. We represent each C_k as a rooted tree connected only by parent pointers. We assume that the parent of a root is the root itself. We identify the tree with the root, i.e., in order to find the subset C_k to which an individual element $c \in C$ belongs, we start at the node storing c, move up the tree following parent pointers, and return the root.

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.**

Input

The program should accept the number of rows m and columns n of the grid as command-line arguments. If no command line arguments are given, it should default to 10 rows by 10 columns.

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.

Main function

In the main function, do the followings:

- Read m and n from the console.
- Initialize the arrays C, H, and V. Print the initial grid.
- call *findtreasure* to remove the $mn - 1$ stone walls so that the first and final chambers are connected. Print the final grid.

You can verify that when less than $mn-1$ stone walls are removed, the chambers may not be fully connected even though the end chamber may be reached from the start chamber.

Sample 1:

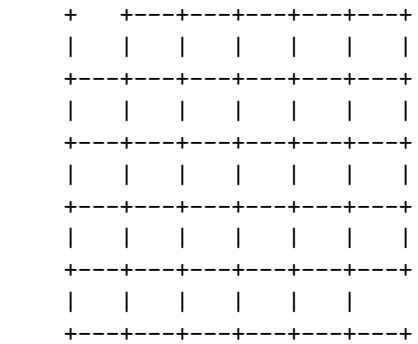
Input :

5 6

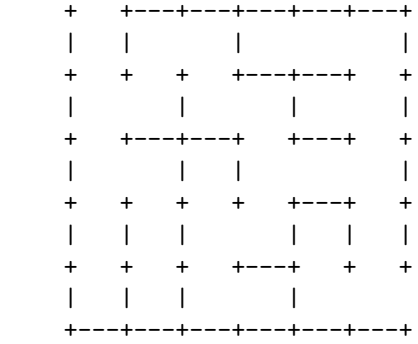
Output :

```
The final chamber can be reached from the start chamber.

Initial Grid
```



```
Final Grid
```



Notes

1. At the time of evaluation, the input data might be different from the one given here.
2. Your code should show the output in the given format in the console. You do not need to write the output to a file.
3. For randomization, you can use the *rand()* function to generate random numbers. At the beginning of your main function, you can add *srand((unsigned int)time(NULL))* to generate different random numbers in different runs of your code.