Introduction:

The given problem is about the **Coin Collection** game problem, which involves controlling a spaceship to collect the maximum number of coins in a grid and avoid danger from enemies. The grid moves downward, and we have the option to use a bomb to destroy enemies in a specific area. In our report we will, discuss the solving approach of this problem by using searching algorithm (BFS or DFS) in c++.

Problem Description:

In the Question, there is a 2D grid, there each cell contains three elements. This are:

- 1. Here (1) value contain a coin.
- 2. Here (2) value contain an enemy.
- 3. Here (0) value is an empty cell, that contains nothing.

The highlighted (yellow) zone is the control zone. S is a spaceship that we need to control so that we can get maximum coins the spaceship has the following movement options:

- Move left by one cell.
- Stay in the current position.
- Move right by one cell.

There is another part in this question is using a bomb. But this bomb can only be detonated once. When used, the bomb clears all enemies within a 5x5 area above the control area.

The game ends when any of the following conditions are met:

- If the spaceship encounters an enemy.
- If the whole grid has moved down, it is impossible to move anymore.
-

Problem Solving Approach:

According to the given solution.

Firstly, we have to define several variables and arrays globally.

i. **i** and **j,** for loop counting.

ii. **N** represents the number of rows in the grid.

iii. **m** = 5, for representing the number of columns in the grid.

iv. **Field** is a 2D array for representing the current state of the grid.

v. **copyField,** for storing a copy of the grid.

vi. **max_coin,** for keeping track of the maximum number of coins collected.

Secondly, we will create a **playGame** function. The **playGame** Function implements the core logic of the game. It recursively investigates diverse spaceship positions, coin collection procedures, and bomb utilization.

- **pos** represents the current spaceship position.
- **rowsLeft** represents the remaining rows in the grid.
- **currentCoin** is the number of coins collected so far.
- **bombStatus** indicates whether the bomb can still be used.

In the **playGame** function, it will check if there is any row left to process. If so, the function return that the game is end. If not, it will check again if the spaceship has reached the left or right boundary of the grid. If it is reached, the function return that the spaceship cannot move in that direction.

Then, it will check current cell containing value. If the value is 1, **currentCoin** will be incremented and **max_coin** updated. If the value is 0, the function will call recursively. And if the value is 2, it checks the availability of the bomb. If the bomb is available. It will explode the enemies, update bomb status, and continue the recursive call. After that, reset the grid using the value that stored in the **copyField** array.

Lastly, in the main function, we will take input from user according to the problem's input format. Then set **pos** = 2 (since, the center is the initial position of spaceship), **coin** = 0, **bombStatus** = 1 and **max_coin** = 0. After that, we will call the **playGame** function for three cases; do not move, left move and right move. At the end print **max collected coin.**

Code:

```
#include <iostream>
using namespace std;

int i, j, n, m = 5;
int Field[15][5], copyField[15][5];
int max_coin = 0;

void playGame(int pos, int rowsLeft, int currentCoin, int bombStatus)
{

    if (rowsLeft == -1)
    {
        return;
    }

    if (pos == m || pos == -1)
    {
        return;
    }

    if (Field[rowsLeft][pos] == 1)
    {
        currentCoin += 1;
        if (currentCoin > max_coin)
            max_coin = currentCoin;
        playGame(pos, rowsLeft - 1, currentCoin, bombStatus);
        playGame(pos - 1, rowsLeft - 1, currentCoin, bombStatus);
        playGame(pos + 1, rowsLeft - 1, currentCoin, bombStatus);
    }
    else if (Field[rowsLeft][pos] == 0)
    {
        playGame(pos, rowsLeft - 1, currentCoin, bombStatus);
        playGame(pos - 1, rowsLeft - 1, currentCoin, bombStatus);
        playGame(pos + 1, rowsLeft - 1, currentCoin, bombStatus);
    }

    else if (Field[rowsLeft][pos] == 2)
    {
        if (bombStatus == 0)
        {
            return;
        }
        else
```

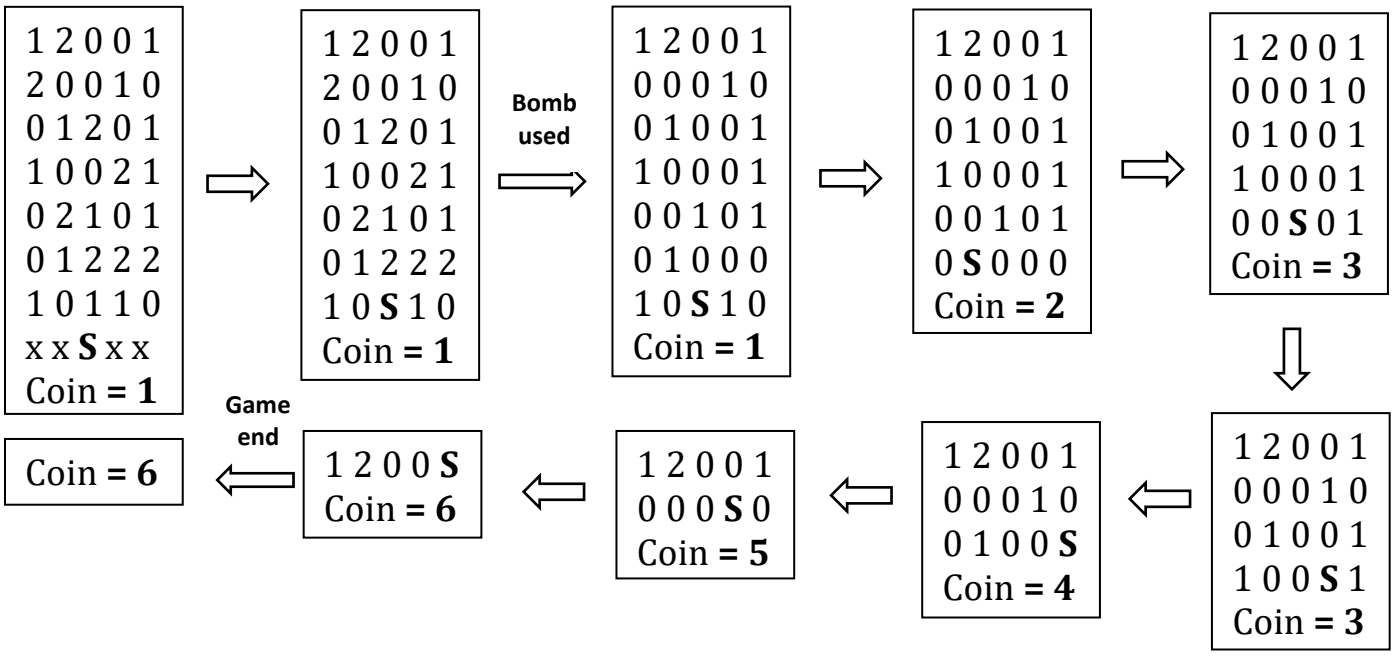
```
        {
            for (i = rowsLeft; i >= 0 && i > rowsLeft - 5; i--)
            {
                for (j = pos - 2; j <= pos + 2; j++)
                {
                    if (j >= 0 && j < m && Field[i][j] == 2)
                    {
                        Field[i][j] = 0;
                    }
                }
            }
            bombStatus = 0;
            playGame(pos, rowsLeft - 1, currentCoin, bombStatus);
            playGame(pos - 1, rowsLeft - 1, currentCoin, bombStatus);
            playGame(pos + 1, rowsLeft - 1, currentCoin, bombStatus);
            for (i = rowsLeft; i >= 0 && i > rowsLeft - 5; i--)
            {
                for (j = pos - 2; j <= pos + 2; j++)
                {
                    if (j >= 0 && j < m)
                    {
                        Field[i][j] = copyField[i][j];
                    }
                }
            }
        }
    }
}

int main()
{
    int copyT, t;
    cin >> t, copyT = t;
    while (t--)
    {
        cin >> n;
        for (i = 0; i < n; i++)
        {
            for (j = 0; j < m; j++)
            {
                cin >> Field[i][j];
                copyField[i][j] = Field[i][j];
            }
        }
        int pos = 2, coin = 0, bombStatus = 1;
```

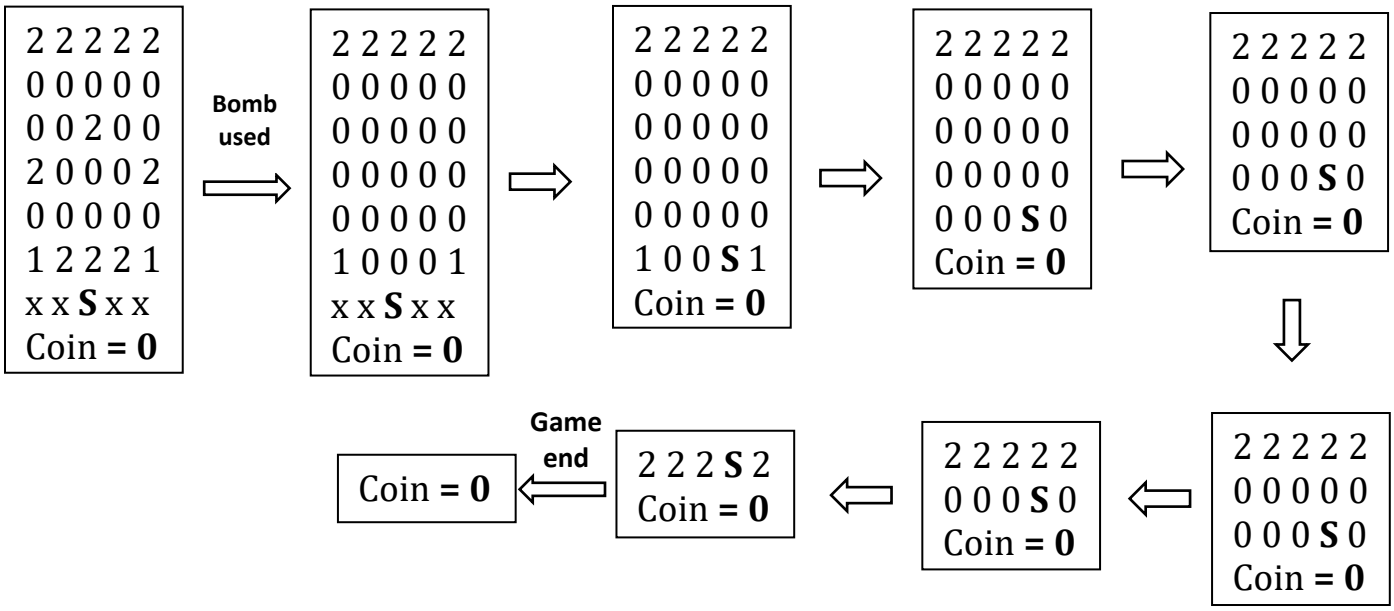
```
        max_coin = 0;
        playGame(pos, n - 1, coin, bombStatus);
        playGame(pos - 1, n - 1, coin, bombStatus);
        playGame(pos + 1, n - 1, coin, bombStatus);
        cout << "#" << (copyT - t) << " " << max_coin << endl;
    }
    return 0;
}
```

Simulation:

For 1st input:



For 2nd input:



Depth First Search:

DFS, Depth First Search, is an edge-based technique. It uses the Stack data structure and performs two stages, first visited vertices are pushed into the stack, and second if there are no vertices then visited vertices are popped.[R1].

Breadth-First Search:

BFS, Breadth-First Search, is a vertex-based technique for finding the shortest path in the graph. It uses a Queue data structure that follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS. [R1].

Here our given solution , he/she applied the **dfs** approach to solve this given question. Now the question is that ,why he/she applied the **dfs** approach instead of **bfs ?**

Advantages of using DFS:

DFS explores a single path as deeply as possible before backtracking .In our given question DFS can simulate scenarios like collecting coins, avoiding

enemies, and utilizing the bomb with precision. Also, DFS reduce the Time Complexity.

In solution, The main mechanism for DFS exploration is the recursive function **playGame()**. The function considers three alternative spaceship movements: remaining in place, moving left, and moving right, starting from the spaceship's initial position and systematically exploring numerous possibilities.

Disadvantages of using BFS:

BFS explores all possibilities at a given level before proceeding to the next level of level order traversal. It may use more memory than DFS. Its time complexity also increases a lot.

Consolation:

To sum up, the question of collecting coins in the game has been solved using DFS methods. The DFS solution efficiently searches for paths and makes choices to maximize the collection of coins

Reference:

Difference between BFS and DFS - GeeksforGeeks[R1]