Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4

по «Алгоритмам и структурам данных»

Яндекс контест

Выполнил:

Студент группы P3233

Перевозчиков И. С.

Преподаватели:

Косяков М. С.

Тараканов Д. С.

Санкт-Петербург

2022

Задача M. Цивилизация

```
#include <iostream>
#include <vector>
#include <map>
#include <string>

using namespace std;

enum {
	START = 0,
	MEADOW = 1,
	FOREST = 2,
	WATER = 1000 * 1000 * 2 + 1,
	NOT_VISITED = 1000 * 1000 * 2 + 2,
	NOT_EXIST = 1000 * 1000 * 2 + 3
};

enum {
	NORTH = 100,
	EAST = 200,
	WEST = 300,
	SOUTH = 400
};

void init(
	vector <vector <int>>& worldMap,
	vector <vector <int>>& minWayMap,
	vector <int>& path,
	multimap<int, int>& currVertices,
	int x1, int y1)
{
	int n = worldMap.size();
	int m = worldMap[0].size();
	minWayMap[x1][y1] = START;

	if (x1 > 0) {
		if (worldMap[x1 - 1][y1] != WATER) {
			currVertices.emplace(worldMap[x1 - 1][y1], (x1 - 1) * m + y1);
			path[(x1 - 1) * m + y1] = SOUTH;
			minWayMap[x1 - 1][y1] = worldMap[x1 - 1][y1];
		}
		else {
			minWayMap[x1 - 1][y1] = WATER;
			path[(x1 - 1) * m + y1] = WATER;
		}
	}

	if (x1 < n - 1) {
		if (worldMap[x1 + 1][y1] != WATER) {
			currVertices.emplace(worldMap[x1 + 1][y1], (x1 + 1) * m + y1);
			path[(x1 + 1) * m + y1] = NORTH;
			minWayMap[x1 + 1][y1] = worldMap[x1 + 1][y1];
		}
		else {
			minWayMap[x1 + 1][y1] = WATER;
			path[(x1 + 1) * m + y1] = WATER;
		}
	}

	if (y1 > 0) {
		if (worldMap[x1][y1 - 1] != WATER) {
			currVertices.emplace(worldMap[x1][y1 - 1], x1 * m + y1 - 1);
			path[x1 * m + y1 - 1] = EAST;
			minWayMap[x1][y1 - 1] = worldMap[x1][y1 - 1];
```

```
                }
                else {
                        minWayMap[x1][y1 - 1] = WATER;
                        path[x1 * m + y1 - 1] = WATER;
                }
        }

        if (y1 < m - 1) {
                if (worldMap[x1][y1 + 1] != WATER) {
                        currVertices.emplace(worldMap[x1][y1 + 1], x1 * m + y1 + 1);
                        path[x1 * m + y1 + 1] = WEST;
                        minWayMap[x1][y1 + 1] = worldMap[x1][y1 + 1];
                }
                else {
                        minWayMap[x1][y1 + 1] = WATER;
                        path[x1 * m + y1 + 1] = WATER;
                }
        }
}

int initUpperNeighbor(int currX, int currY, vector <vector <int>>& minWayMap, vector
<vector <int>>& worldMap) {
        if (currX > 0) {
                if (worldMap[currX - 1][currY] == WATER) {
                        return WATER;
                }
                return minWayMap[currX - 1][currY];
        }
        else {
                return NOT_EXIST;
        }
}

int initBottomNeighbor(int currX, int currY, vector <vector <int>>& minWayMap,
vector <vector <int>>& worldMap) {
        if (currX < minWayMap.size() - 1) {
                if (worldMap[currX + 1][currY] == WATER) {
                        return WATER;
                }
                return minWayMap[currX + 1][currY];
        }
        else {
                return NOT_EXIST;
        }
}

int initLeftNeighbor(int currX, int currY, vector <vector <int>>& minWayMap, vector
<vector <int>>& worldMap) {
        if (currY > 0) {
                if (worldMap[currX][currY - 1] == WATER) {
                        return WATER;
                }
                return minWayMap[currX][currY - 1];
        }
        else {
                return NOT_EXIST;
        }
}

int initRightNeighbor(int currX, int currY, vector <vector <int>>& minWayMap, vector
<vector <int>>& worldMap) {
        if (currY < minWayMap[0].size() - 1) {
                if (worldMap[currX][currY + 1] == WATER) {
                        return WATER;
                }
```

```
                return minWayMap[currX][currY + 1];
        }
        else {
                return NOT_EXIST;
        }
}

void dijkstra(
        vector <vector <int>>& worldMap,
        vector <vector <int>>& minWayMap,
        vector <int>& path,
        multimap<int, int>& currVertices,
        int x1, int y1)
{
        int n = worldMap.size();
        int m = worldMap[0].size();
        int currDistance = currVertices.begin()->first;
        int currX = currVertices.begin()->second / m;
        int currY = currVertices.begin()->second % m;

        currVertices.erase(currVertices.begin());



        int upperNeighbor = initUpperNeighbor(currX, currY, minWayMap, worldMap);
        int bottomNeighbor = initBottomNeighbor(currX, currY, minWayMap, worldMap);
        int leftNeighbor = initLeftNeighbor(currX, currY, minWayMap, worldMap);
        int rightNeighbor = initRightNeighbor(currX, currY, minWayMap, worldMap);



        int currMinNeighbor = NOT_EXIST;

        if (path[currX * m + currY] != NORTH &&
                upperNeighbor != START &&
                upperNeighbor != WATER &&
                upperNeighbor != NOT_VISITED &&
                upperNeighbor != NOT_EXIST
                ) {
                currMinNeighbor = upperNeighbor;
        }

        if (path[currX * m + currY] != SOUTH &&
                bottomNeighbor != START &&
                bottomNeighbor != WATER &&
                bottomNeighbor != NOT_VISITED &&
                bottomNeighbor != NOT_EXIST
                ) {
                currMinNeighbor = bottomNeighbor;
        }

        if (path[currX * m + currY] != WEST &&
                leftNeighbor != START &&
                leftNeighbor != WATER &&
                leftNeighbor != NOT_VISITED &&
                leftNeighbor != NOT_EXIST
                ) {
                currMinNeighbor = leftNeighbor;
        }

        if (path[currX * m + currY] != EAST &&
                rightNeighbor != START &&
                rightNeighbor != WATER &&
                rightNeighbor != NOT_VISITED &&
                rightNeighbor != NOT_EXIST
```

```
            ) {
                  currMinNeighbor = rightNeighbor;
            }


            if (currMinNeighbor != START &&
                  currMinNeighbor != WATER &&
                  currMinNeighbor != NOT_VISITED &&
                  currMinNeighbor != NOT_EXIST &&
                  currMinNeighbor + worldMap[currX][currY] < currDistance) {
                  minWayMap[currX][currY] = currMinNeighbor + worldMap[currX][currY];
                  currDistance = currMinNeighbor + worldMap[currX][currY];

                  if (currMinNeighbor == upperNeighbor) {
                        path[currX * m + currY] = NORTH;
                  }
                  else if (currMinNeighbor == bottomNeighbor) {
                        path[currX * m + currY] = SOUTH;
                  }
                  else if (currMinNeighbor == leftNeighbor) {
                        path[currX * m + currY] = WEST;
                  }
                  else {
                        path[currX * m + currY] = EAST;
                  }
            }



            if (upperNeighbor == NOT_VISITED) {
                  minWayMap[currX - 1][currY] = currDistance + worldMap[currX -
1][currY];
                  path[(currX - 1) * m + currY] = SOUTH;
                  currVertices.emplace(currDistance + worldMap[currX - 1][currY], (currX
- 1) * m + currY);
            }
            else if (upperNeighbor == WATER) {
                  minWayMap[currX - 1][currY] = WATER;
                  path[(currX - 1) * m + currY] = WATER;
            }

            if (bottomNeighbor == NOT_VISITED) {
                  bottomNeighbor = currDistance + worldMap[currX + 1][currY];
                  minWayMap[currX + 1][currY] = currDistance + worldMap[currX +
1][currY];
                  path[(currX + 1) * m + currY] = NORTH;
                  currVertices.emplace(currDistance + worldMap[currX + 1][currY], (currX
+ 1) * m + currY);
            }
            else if (bottomNeighbor == WATER) {
                  minWayMap[currX + 1][currY] = WATER;
                  path[(currX + 1) * m + currY] = WATER;
            }

            if (leftNeighbor == NOT_VISITED) {
                  leftNeighbor = currDistance + worldMap[currX][currY - 1];
                  minWayMap[currX][currY - 1] = currDistance + worldMap[currX][currY -
1];
                  path[currX * m + currY - 1] = EAST;
                  currVertices.emplace(currDistance + worldMap[currX][currY - 1], currX *
m + currY - 1);
            }
            else if (leftNeighbor == WATER) {
                  minWayMap[currX][currY - 1] = WATER;
                  path[currX * m + currY - 1] = WATER;
```

```
	}

	if (rightNeighbor == NOT_VISITED) {
		rightNeighbor = currDistance + worldMap[currX][currY + 1];
		minWayMap[currX][currY + 1] = currDistance + worldMap[currX][currY +
1];
		path[currX * m + currY + 1] = WEST;
		currVertices.emplace(currDistance + worldMap[currX][currY + 1], currX *
m + currY + 1);
	}
	else if (rightNeighbor == WATER) {
		minWayMap[currX][currY + 1] = WATER;
		path[currX * m + currY + 1] = WATER;
	}
}

string parsePath(int hash1, int hash2, int m, vector <int>& path) {
	string minPath = "";
	while (hash2 != hash1) {
		if (path[hash2] == NORTH) {
			minPath += 'S';
			hash2 -= m;
		}
		else if (path[hash2] == SOUTH) {
			minPath += 'N';
			hash2 += m;
		}
		else if (path[hash2] == WEST) {
			minPath += 'E';
			hash2--;
		}
		else {
			minPath += 'W';
			hash2++;
		}
	}

	return minPath;
}



int main()
{
	int n, m;
	cin >> n >> m;
	int x1, y1, x2, y2;
	cin >> x1 >> y1 >> x2 >> y2;
	x1--;
	y1--;
	x2--;
	y2--;
	vector <vector <int>> worldMap(n, vector <int>(m));
	vector <vector <int>> minWayMap(n, vector <int>(m, NOT_VISITED));
	vector <int> path(n * m, NOT_VISITED);
	multimap <int, int> currVertices;

	for (int i = 0; i < n; i++) {
		for (int j = 0; j < m; j++) {
			char block;
			cin >> block;
			switch (block) {
			case '.':
				worldMap[i][j] = MEADOW;
				break;
```

```
                        case 'W':
                                worldMap[i][j] = FOREST;
                                break;
                        case '#':
                                worldMap[i][j] = WATER;
                        }
                }
        }

        init(worldMap, minWayMap, path, currVertices, x1, y1);

        while (!currVertices.empty()) {
                dijkstra(worldMap, minWayMap, path, currVertices, x1, y1);
        }

        if (minWayMap[x2][y2] != WATER && minWayMap[x2][y2] != NOT_VISITED) {
                cout << minWayMap[x2][y2] << '\n';
                string minPath = parsePath(x1 * m + y1, x2 * m + y2, m, path);
                for (int i = minPath.size() - 1; i >= 0; i--) {
                        cout << minPath[i];
                }
        }
        else {
                cout << -1;
        }

        return 0;
}
```

Алгоритм Дейкстры

Алгоритмическая сложность: $O(n * m)$.

Задача N. Свинки-копилки

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <string>

using namespace std;

int main()
{
	int n;
	cin >> n;

	vector <int> moneyBoxes(n);
	vector <int> cycles(n, 0);

	for (int i = 0; i < n; i++) {
		int x;
		cin >> x;
		moneyBoxes[i] = x - 1;
	}

	int currCycle = 1;

	for (int i = 0; i < n; i++) {
		int j = i;

		bool cycle = cycles[j] == 0;

		if (cycle) {
			while (cycles[j] == 0) {
				cycles[j] = currCycle;
				j = moneyBoxes[j];
			}

			if (cycles[j] == currCycle) {
				currCycle++;
			}
			else {
				int k = i;
				int prevCycle = cycles[j];
				while (cycles[k] != prevCycle) {
					cycles[k] = prevCycle;
					k = moneyBoxes[k];
				}
			}
		}
	}

	cout << currCycle - 1;
}
```

Нужно узнать количество цепочек, таких что, в каждой цепочке ключ от текущей копилки хранится в предыдущей. При этом такие цепочки могут образовывать циклы. Количество цепочек и есть ответ.

Алгоритмическая сложность: $O(n^2)$.

Задача O. Долой списывание!

```
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <string>

using namespace std;

enum color {
	COLORLESS,
	WHITE,
	BLACK
};

void dfs(int lkshonok, int prev, vector <vector <int>>& lkshata, vector <int>&
colors) {
	if (colors[prev] == BLACK) {
		colors[lkshonok] = WHITE;
	}
	else {
		colors[lkshonok] = BLACK;
	}

	for (int i = 0; i < lkshata[lkshonok].size(); i++) {
		if (lkshata[lkshonok][i] == prev) {
			continue;
		}

		if (colors[lkshata[lkshonok][i]] == COLORLESS) {
			dfs(lkshata[lkshonok][i], lkshonok, lkshata, colors);
		}
	}
}

bool containCycle(vector <vector <int>>& lkshata, vector <int>& colors) {
	for (int i = 0; i < lkshata.size(); i++) {
		for (int j = 0; j < lkshata[i].size(); j++) {
			if (colors[lkshata[i][j]] == colors[i]) {
				return true;
			}
		}
	}
	return false;
}


int main()
{
	int n, m;
	cin >> n >> m;
	vector <vector <int>> lkshata(n);
	vector <int> colors(n, COLORLESS);

	for (int i = 0; i < m; i++) {
		int a, b;
		cin >> a >> b;
		a--;
		b--;
		lkshata[a].push_back(b);
		lkshata[b].push_back(a);
	}

	for (int i = 0; i < n; i++) {
```

```
			if (colors[i] == COLORLESS) {
				dfs(i, i, lkshata, colors);
			}
		}

		if (containCycle(lkshata, colors)) {
			cout << "NO";
		}
		else {
			cout << "YES";
		}

}
```

С помощью DFS заполняем цикл двумя цветами, далее проверяем на двудольность. Если граф не двудольный, то разделить лкшат невозможно.

Алгоритмическая сложность: $O(n + m)$.

Задача P. Авиаперелеты

```
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <string>

using namespace std;


enum visited {
	VISITED,
	NOT_VISITED,
	IN,
	OUT
};

void dfsOut(int minTubSize, int curr, int inOrOut, vector <vector <int>>&
bubundiaMap, vector <int>& visitedCities) {
	visitedCities[curr]= VISITED;
	for (int i = 0; i < visitedCities.size(); i++) {
		if (i != curr && visitedCities[i] == NOT_VISITED &&
bubundiaMap[curr][i] <= minTubSize) {
			dfsOut(minTubSize, i, IN, bubundiaMap, visitedCities);
		}
	}
}

void dfsIn(int minTubSize, int curr, int inOrOut, vector <vector <int>>&
bubundiaMap, vector <int>& visitedCities) {
	visitedCities[curr] = VISITED;
	for (int i = 0; i < visitedCities.size(); i++) {
		if (i != curr && visitedCities[i] == NOT_VISITED &&
bubundiaMap[i][curr] <= minTubSize) {
			dfsIn(minTubSize, i, OUT, bubundiaMap, visitedCities);
		}
	}
}

bool isAllCitiesVisited(vector <int>& visitedCities) {
	for (int i = 0; i < visitedCities.size(); i++) {
		if (visitedCities[i] == NOT_VISITED) {
			return false;
		}
	}
	return true;
}


int main()
{
	int n;
	cin >> n;

	vector <vector <int>> bubundiaMap(n, vector <int>(n));

	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++) {
			int a;
			cin >> a;
			bubundiaMap[i][j] = a;
		}
	}
```

```
	int l = 0;
	int r = INT32_MAX;

	while (l < r) {
		vector <int> visitedCitiesIn(n, NOT_VISITED);
		vector <int> visitedCitiesOut(n, NOT_VISITED);
		dfsIn((l + r) / 2, 0, OUT, bubundiaMap, visitedCitiesIn);
		dfsOut((l + r) / 2, 0, OUT, bubundiaMap, visitedCitiesOut);
		if (isAllCitiesVisited(visitedCitiesIn) &&
isAllCitiesVisited(visitedCitiesOut)) {
			r = (l + r) / 2;
		}
		else {
			l = (l + r) / 2 + 1;
		}
	}

	cout << l;
}
```

Бинарный поиск по ответу.

С помощью bfsIn проверяем, можно ли попасть в каждую вершину из заданной с текущим размером бака. С Помощью bfsOut проверяем можно ли попасть из каждой вершины в заданную. Если при обоих dfs-ах все вершины пройдены, то бак подходит, иначе не подходит.

Алгоритмическая сложность: $O(n^2 \cdot \log(n))$.