# Amortization Table

Rachel Hamilton, Jake Cousino

Wednesday, June 20

## Contents

**Rubric**

25 pts commented and readable code ________________

25 pts elegant source code ________________

25 pts concise design ________________

25 pts results during testing ________________

# Part I
# Source Code

mazeSolver.cpp

```cpp
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string>

using namespace std;


int backtrackingMazeSolver(int i, int j);
int greedyMazeSolver(int i, int j, int endX, int endY);
std::vector<int> GetClosestNodeToFinish(int i, int j, int endX, int endY);
int divideAndConquerMazeSolver(int i, int j);
int dynamicProgrammingMazeSolver(int i, int j);
int randomizedMazeSolver(int i, int j);
bool isEmpty(char **myMaze, int i, int j, int height, int width);
bool hasBeenChecked(char **myMaze, int i, int j, int height, int width);
int bruteForceMazeSolver(char **myMaze, int i, int j, bool oneShot, int startX, int startY, struct maze structMaze);
void printArray(char **array, int row, int col);
bool bruteCheckForEmpty(char **myMaze, int i, int j, struct maze structMaze);
bool bruteCheckForTraveled(char **myMaze, int i, int j, struct maze structMaze);
bool isFinishAdjacent(char **myMaze, int i, int j);

struct maze
{
        int rows;
        int cols;
        char matrix [100][100];
};

maze myMaze;
int bruteForceCount = 0;

int main()
{
        //required variables
        ifstream in;
        in.open("maze.txt");
        char line;

        //read the matrix using plain c code, character by character
        in >> myMaze.rows;
        in >> line;
        in >> myMaze.cols;
        cout << "Reading␣a␣" << myMaze.rows << "␣by␣" << myMaze.cols << "␣matrix." << endl;
        //Burn the end of line character
        in.ignore(200,'\n');
        for(int i=0; i<myMaze.rows; i++)
        {
                for(int j=0; j<myMaze.cols; j++)
                {
                        in.get( myMaze.matrix[i][j] );
                }
                //Burn the end of line character
                in.ignore(200,'\n');
        }

        //Print the empty maze
        for(int i=0; i<myMaze.rows; i++)
        {
                for(int j=0; j<myMaze.cols; j++)
                cout << myMaze.matrix[i][j];
                cout << endl;
        }
        int x=1,y=1;

        //Find starting coordinates
        for(int i=0; i<myMaze.rows; i++)
        for(int j=0; j<myMaze.cols; j++)
        if( myMaze.matrix[i][j] == 'S' ){
                x=i;
                y=j;
        }

        Find Finish coordinates
        for(int i=0; i<myMaze.rows; i++)
        for(int j=0; j<myMaze.cols; j++)
        if( myMaze.matrix[i][j] == 'F' ){
                endX=j;
                endY=i;
        }

        //Call a recursive mazeSolver
        FIXME:RH:int bfDistance = bruteForceMazeSolver(x,y);      //brute force? dnc?
        int btDistance = backtrackingMazeSolver(x,y);      //brute force? dnc?
        int gDistance = greedyMazeSolver(x,y,endX,endY);
        int dncDistance = divideAndConquerMazeSolver(x,y);
        int dpDistance = dynamicProgrammingMazeSolver(x,y);
        int rDistance = randomizedMazeSolver(x,y);

        cout << "Brute␣force␣distance:␣" << bfDistance << "␣units␣away!" << endl;
        cout << "Backtracking␣distance:␣" << btDistance << "␣units␣away!" << endl;
        cout << "Greedy␣distance:␣" << gDistance << "␣units␣away!" << endl;
        cout << "Divide␣and␣conquer␣distance:␣" << dncDistance << "␣units␣away!" << endl;
        cout << "Dynamic␣programming␣distance:␣" << dpDistance << "␣units␣away!" << endl;
        cout << "Randomized␣distance:␣" << rDistance << "␣units␣away!" << endl;

        //Print solved maze - x2
        // for(int i=0; i<myMaze.rows; i++)
        // {
        //      for(int j=0; j<myMaze.cols; j++)
        //      cout << myMaze.matrix[i][j];
        //      cout << endl;
        // }

        // ****************************************
        // Begin Student Written Section
        // ****************************************


        char **mazeArray;
        bool alwaysTrue = true;

        mazeArray = new char *[myMaze.rows];
        for(int i=0 ; i<myMaze.rows ; i++){
                mazeArray[i] = new char[myMaze.cols];
                for(int j=0 ; j<myMaze.cols ; j++)
                mazeArray[i][j] = myMaze.matrix[i][j];
        }

        bruteForceMazeSolver(mazeArray, 1, 1, !alwaysTrue, x, y, myMaze);

        return 0;
}

int bruteForceMazeSolver(char **myMaze, int i, int j, bool oneShot, int startX, int startY,
    struct maze structMaze)
{
        bool isEmptySpace, isTraveledSpace;

        bruteForceCount++;

        // will not activate unless this is the first iteration
        // gives starting location
        if(!oneShot){
                i = startX;
                j = startY;
                oneShot = true;
                myMaze[i][j] = 't';
        }

        // Check if adjacent spot is F
        if(isFinishAdjacent(myMaze,i,j)){
                myMaze[i][j] = 't';
                printArray(myMaze, structMaze.rows, structMaze.cols);
                cout << "Brute distance: " << bruteForceCount << endl;
        }
        else{
                // if a finish spot is not nearby, check for empty space
                isEmptySpace = bruteCheckForEmpty(myMaze,i,j,structMaze);
                if(!isEmptySpace){
                        // if there are no empty spaces, check for already traveled spaces
                        isTraveledSpace = bruteCheckForTraveled(myMaze,i,j, structMaze);
                        if(!isTraveledSpace){
                                cout << "ERROR";
                        }
                }
        }

        return -1;
}
int backtrackingMazeSolver(int i, int j)
{
        //algorithm goes here
        return -1;
}
int greedyMazeSolver(int i, int j, int endX, int endY)
{
        if(myMaze.matrix[i][j] == 'F') return 1;

        std::vector<int> nextNode = GetClosestNodeToFinish(i, j, endX, endY);
        myMaze.matrix[nextNode[0]][nextNode[1]] = 'X';
        return greedyMazeSolver(nextNode[0], nextNode[1], endX, endY);

  // return -1;
}

std::vector<int> GetClosestNodeToFinish(int i, int j, int endX, int endY)
{
        std::vector<int> north = {i, j-1};
        std::vector<int> east = {i+1, j};
        std::vector<int> south = {i, j+1};
        std::vector<int> west = {i-1, j};

        std::vector<std::vector<int>> directions;
        directions.push_back(north);
        directions.push_back(east);
        directions.push_back(south);
        directions.push_back(west);

        std::vector<int> clostestNode = south;
        for(int i = 0; i < 4; i++)
        {
                int nodeDistance = std::abs(clostestNode[0] - endX) + std::abs(clostestNode[1] - endY);
                if(nodeDistance > (std::abs(directions[i][0] - endX) + std::abs(directions[i][1] - endY)))
                {
                        if(myMaze.matrix[directions[i][0]][directions[i][1]] != '*')
                        {
                                clostestNode = directions[i];
                        }
                }
        }
        return clostestNode;
}
int divideAndConquerMazeSolver(int i, int j)
{
        //algorithm goes here
        return -1;
}
int dynamicProgrammingMazeSolver(int i, int j)
{
        //algorithm goes here
        return -1;
}
int randomizedMazeSolver(int i, int j)
{
        //algorithm goes here
        return -1;
}

// Added by RH
// given the maze, and the space to check, will check
// for to see if the lacation is valid first, then if so
// will check the character stored is a space character
// (not in the Master Chief sorta way), then set the return
// condition to true
bool isEmpty(char **myMaze, int i, int j, int height, int width){
        bool isEmpty = false;
        bool isValidLocation = true;

        // check if valid
        if(height < i){
                cout << "Not␣a␣valid␣row␣lacation";
                isValidLocation = !isValidLocation;
        }
        if(width < j){
                cout << "Not␣a␣valid␣col␣lacation";
                isValidLocation = !isValidLocation;
        }

        // check if space
        if(isValidLocation){
                if(myMaze[i][j] == '␣')
                {
                        isEmpty = true;
                }
        }

        return isEmpty;

}

// Added by RH
// Same as isValid, but for check the char 't'
bool hasBeenChecked(char **myMaze, int i, int j, int height, int width){
        bool isTracked = false;
        bool isValidLocation = true;

        // check if valid
        if(height < i){
                cout << "Not␣a␣valid␣row␣lacation";
                isValidLocation = !isValidLocation;
        }
        if(width < j){
                cout << "Not␣a␣valid␣col␣lacation";
                isValidLocation = !isValidLocation;
        }


        // check if space
        if(isValidLocation){
                if(myMaze[i][j] == 't'){
                        isTracked = true;
                }
        }

        return isTracked;

}

void printArray(char **array, int row, int col){
        for(int i=0 ; i<row ; i++){
                for(int j=0 ; j<col ; j++){
                        cout << array[i][j];
                }
                cout << endl;
        }
}


// Checks for empty spots in direction order east, north, south, west
// Marks a 't' on every traveled spot
bool bruteCheckForEmpty(char **myMaze, int i, int j, struct maze structMaze){
        if(isEmpty(myMaze,i,j+1,structMaze.rows,structMaze.cols)){
                myMaze[i][j] = 't';
                bruteForceMazeSolver(myMaze, i, j+1, true, 0, 0, structMaze);
        }
        else{
                if(isEmpty(myMaze,i+1,j,structMaze.rows,structMaze.cols)){
                        myMaze[i][j] = 't';
                        bruteForceMazeSolver(myMaze, i+1, j, true, 0, 0, structMaze);
                }
                else{
                        if(isEmpty(myMaze,i-1,j,structMaze.rows,structMaze.cols)){
                                myMaze[i][j] = 't';
                                bruteForceMazeSolver(myMaze, i-1, j, true, 0, 0, structMaze);
                        }
                        else{
                                if(isEmpty(myMaze,i,j-1,structMaze.rows,structMaze.cols)){
                                        myMaze[i][j] = 't';
                                        bruteForceMazeSolver(myMaze, i, j-1, true, 0, 0, structMaze);
                                }
                                else{
                                        return false;
                                }
                        }
                }
        }
        return true;
}


// Runs the same thing as bruteCheckForEmpty, except places 'x' and
// does not run if the finish is adjacent. Without the check, the program
// defaults to looking for 't' spaces rather than stopping
bool bruteCheckForTraveled(char **myMaze, int i, int j, struct maze structMaze){
        if(!isFinishAdjacent(myMaze,i,j)){
                if(hasBeenChecked(myMaze,i,j+1,structMaze.rows,structMaze.cols)){
                        myMaze[i][j] = 'x';
```

```
            bruteForceMazeSolver(myMaze, i, j+1,true, 0, 0, structMaze);
        }
        else{
            if(hasBeenChecked(myMaze,i+1,j,structMaze.rows,structMaze.cols)){
                myMaze[i][j] = 'x';
                bruteForceMazeSolver(myMaze, i+1, j, true, 0, 0, structMaze);
            }
            else{
                if(hasBeenChecked(myMaze,i-1,j,structMaze.rows,structMaze.cols)){
                    myMaze[i][j] = 'x';
                    bruteForceMazeSolver(myMaze, i-1, j, true, 0, 0, structMaze);
                }
                else{
                    if(hasBeenChecked(myMaze,i,j-1,structMaze.rows,structMaze.cols)){
                        myMaze[i][j] = 'x';
                        bruteForceMazeSolver(myMaze, i, j-1, true, 0, 0, structMaze);
                    }
                    else{
                        return false;
                    }
                }
            }
        }
    }
    return true;
}

// checks all directions for adjacent 'F'
bool isFinishAdjacent(char **myMaze, int i, int j){

    if(myMaze[i+1][j]=='F'){
        return true;
    }
    if(myMaze[i-1][j]=='F'){
        return true;
    }
    if(myMaze[i][j+j]=='F'){
        return true;
    }
    if(myMaze[i][j-j]=='F'){
        return true;
    }

    return false;
}

//recursion!!
//Mark current location
//Base Case: Look north, south, east, west for victory!
//Mark our path
//Try going south if it is open
//Try going north if it is open
//Try going east if it is open
//Try going west if it is open
```

# Part II
# Output

```
Reading a 20 by 20 matrix.
********************
*******S    ********
*****   ***     ****
***** *      ** ****
*     ******     ***
* ******************
*   ****************
***    *********F***
*    *     ***** ***
*  ******* ***** ***
**   ***** ***   ***
**  ****** *** *****
**     *** *** *****
******       * *****
*      ******* *****
***  ***** ***   ***
**** ***   ***** ***
****      ****   ***
*********      *****
********************
Backtracking distance: -1 units away!
Divide and conquer distance: -1 units away!
Dynamic programming distance: -1 units away!
Randomized distance: -1 units away!
********************
*******ttttt********
*****ttt***ttttt****
*****t*tttttt**t****
*ttttt******ttttx***
*t******************
*ttt****************
***tttt*********F***
*    *ttttt*****t***
*  *******t*****t***
**   *****t***ttt***
**  ******t***t*****
**     ***t***t*****
******tttttxx*t*****
*   ttt*******t*****
*** t***** ***ttt***
****t***   *****t***
****tttttt****ttt***
*********tttttt*****
********************
Brute distance: 91
```