# 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

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5
6  using namespace std;
7
8
9  int backtrackingMazeSolver(int i, int j);
10 int greedyMazeSolver(int i, int j, int endX, int endY);
11 std::vector<int> GetClosestNodeToFinish(int i, int j, int endX, int endY);
12 int divideAndConquerMazeSolver(int i, int j);
13 int dynamicProgrammingMazeSolver(int i, int j);
14 int randomizedMazeSolver(int i, int j);
15 bool isEmpty(char **myMaze, int i, int j, int height, int width);
16 bool hasBeenChecked(char **myMaze, int i, int j, int height, int width);
17 int bruteForceMazeSolver(char **myMaze, int i, int j, bool oneShot, int startX, int startY,
       struct maze structMaze);
18 void printArray(char **array, int row, int col);
19 bool bruteCheckForEmpty(char **myMaze, int i, int j, struct maze structMaze);
20 bool bruteCheckForTraveled(char **myMaze, int i, int j, struct maze structMaze);
21 bool isFinishAdjacent(char **myMaze, int i, int j);
22
23 struct maze
24 {
25         int rows;
26         int cols;
27         char matrix [100][100];
28 };
29
30 maze myMaze;
31 int bruteForceCount = 0;
32
33 int main()
34 {
35         //required variables
36         ifstream in;
37         in.open("maze.txt");
38         char line;
39
40         //read the matrix using plain c code, character by character
41         in >> myMaze.rows;
42         in >> line;
43         in >> myMaze.cols;
44         cout << "Reading a " << myMaze.rows << " by " << myMaze.cols << " matrix." << endl;
45         //Burn the end of line character
46         in.ignore(200,'\n');
47         for(int i=0; i<myMaze.rows; i++)
48         {
49                 for(int j=0; j<myMaze.cols; j++)
50                 {
51                         in.get( myMaze.matrix[i][j] );
52                 }
53                 //Burn the end of line character
54                 in.ignore(200,'\n');
55         }
56
57         //Print the empty maze
58         for(int i=0; i<myMaze.rows; i++)
59         {
60                 for(int j=0; j<myMaze.cols; j++)
```

```
61                      cout << myMaze.matrix[i][j];
62                      cout << endl;
63              }
64              int x=1,y=1;
65
66              //Find starting coordinates
67              for(int i=0; i<myMaze.rows; i++)
68              for(int j=0; j<myMaze.cols; j++)
69              if( myMaze.matrix[i][j] == 'S' ){
70                      x=i;
71                      y=j;
72              }
73
74              Find Finish coordinates
75              for(int i=0; i<myMaze.rows; i++)
76              for(int j=0; j<myMaze.cols; j++)
77              if( myMaze.matrix[i][j] == 'F' ){
78                      endX=j;
79                      endY=i;
80              }
81
82              //Call a recursive mazeSolver
83              FIXME:RH:int bfDistance = bruteForceMazeSolver(x,y);      //brute force? dnc?
84              int btDistance = backtrackingMazeSolver(x,y);      //brute force? dnc?
85              int gDistance = greedyMazeSolver(x,y,endX,endY);
86              int dncDistance = divideAndConquerMazeSolver(x,y);
87              int dpDistance = dynamicProgrammingMazeSolver(x,y);
88              int rDistance = randomizedMazeSolver(x,y);
89
90              cout << "Brute force distance: " << bfDistance << " units away!" << endl;
91              cout << "Backtracking distance: " << btDistance << " units away!" << endl;
92              cout << "Greedy distance: " << gDistance << " units away!" << endl;
93              cout << "Divide and conquer distance: " << dncDistance << " units away!" << endl;
94              cout << "Dynamic programming distance: " << dpDistance << " units away!" << endl;
95              cout << "Randomized distance: " << rDistance << " units away!" << endl;
96
97              //Print solved maze - x2
98              // for(int i=0; i<myMaze.rows; i++)
99              // {
100             //      for(int j=0; j<myMaze.cols; j++)
101             //      cout << myMaze.matrix[i][j];
102             //      cout << endl;
103             // }
104
105             // ***************************************
106             // Begin Student Written Section
107             // ***************************************
108
109
110             char **mazeArray;
111             bool alwaysTrue = true;
112
113             mazeArray = new char *[myMaze.rows];
114             for(int i=0 ; i<myMaze.rows ; i++){
115                     mazeArray[i] = new char[myMaze.cols];
116                     for(int j=0 ; j<myMaze.cols ; j++)
117                     mazeArray[i][j] = myMaze.matrix[i][j];
118             }
119
120             bruteForceMazeSolver(mazeArray, 1, 1, !alwaysTrue, x, y, myMaze);
121
122             return 0;
123     }
124
125     int bruteForceMazeSolver(char **myMaze, int i, int j, bool oneShot, int startX, int startY,
        struct maze structMaze)
126     {
127             bool isEmptySpace, isTraveledSpace;
```

```
128
129         bruteForceCount++;
130
131         // will not activate unless this is the first iteration
132         // gives starting location
133         if(!oneShot){
134                 i = startX;
135                 j = startY;
136                 oneShot = true;
137                 myMaze[i][j] = 't';
138         }
139
140         // Check if adjacent spot is F
141         if(isFinishAdjacent(myMaze,i,j)){
142                 myMaze[i][j] = 't';
143                 printArray(myMaze, structMaze.rows, structMaze.cols);
144                 cout << "Brute distance: " << bruteForceCount << endl;
145         }
146         else{
147                 // if a finish spot is not nearby, check for empty space
148                 isEmptySpace = bruteCheckForEmpty(myMaze,i,j,structMaze);
149                 if(!isEmptySpace){
150                         // if there are no empty spaces, check for already traveled spaces
151                         isTraveledSpace = bruteCheckForTraveled(myMaze,i,j, structMaze);
152                         if(!isTraveledSpace){
153                                 cout << "ERROR";
154                         }
155                 }
156         }
157
158         return -1;
159 }
160 int backtrackingMazeSolver(int i, int j)
161 {
162         //algorithm goes here
163         return -1;
164 }
165 int greedyMazeSolver(int i, int j, int endX, int endY)
166 {
167         if(myMaze.matrix[i][j] == 'F') return 1;
168
169         std::vector<int> nextNode = GetClosestNodeToFinish(i, j, endX, endY);
170         myMaze.matrix[nextNode[0]][nextNode[1]] = 'X';
171         return greedyMazeSolver(nextNode[0], nextNode[1], endX, endY);
172
173   // return -1;
174 }
175
176 std::vector<int> GetClosestNodeToFinish(int i, int j, int endX, int endY)
177 {
178         std::vector<int> north = {i, j-1};
179         std::vector<int> east = {i+1, j};
180         std::vector<int> south = {i, j+1};
181         std::vector<int> west = {i-1, j};
182
183         std::vector<std::vector<int>> directions;
184         directions.push_back(north);
185         directions.push_back(east);
186         directions.push_back(south);
187         directions.push_back(west);
188
189         std::vector<int> clostestNode = south;
190         for(int i = 0; i < 4; i++)
191         {
192                 int nodeDistance = std::abs(clostestNode[0] - endX) + std::abs(clostestNode
                        [1] - endY);
193                 if(nodeDistance > (std::abs(directions[i][0] - endX) + std::abs(directions[i
                        ][1] - endY)))
```

```
194                 {
195                         if(myMaze.matrix[directions[i][0]][directions[i][1]] != '*')
196                         {
197                                 clostestNode = directions[i];
198                         }
199                 }
200         }
201         return clostestNode;
202 }
203 int divideAndConquerMazeSolver(int i, int j)
204 {
205         //algorithm goes here
206         return -1;
207 }
208 int dynamicProgrammingMazeSolver(int i, int j)
209 {
210         //algorithm goes here
211         return -1;
212 }
213 int randomizedMazeSolver(int i, int j)
214 {
215         //algorithm goes here
216         return -1;
217 }
218
219 // Added by RH
220 // given the maze, and the space to check, will check
221 // for to see if the lacation is valid first, then if so
222 // will check the character stored is a space character
223 // (not in the Master Chief sorta way), then set the return
224 // condition to true
225 bool isEmpty(char **myMaze, int i, int j, int height, int width){
226         bool isEmpty = false;
227         bool isValidLocation = true;
228
229         // check if valid
230         if(height < i){
231                 cout << "Not a valid row lacation";
232                 isValidLocation = !isValidLocation;
233         }
234         if(width < j){
235                 cout << "Not a valid col lacation";
236                 isValidLocation = !isValidLocation;
237         }
238
239         // check if space
240         if(isValidLocation){
241                 if(myMaze[i][j] == ' ')
242                 {
243                         isEmpty = true;
244                 }
245         }
246
247         return isEmpty;
248
249 }
250
251 // Added by RH
252 // Same as isValid, but for check the char 't'
253 bool hasBeenChecked(char **myMaze, int i, int j, int height, int width){
254         bool isTracked = false;
255         bool isValidLocation = true;
256
257         // check if valid
258         if(height < i){
259                 cout << "Not a valid row lacation";
260                 isValidLocation = !isValidLocation;
261         }
```

```
262         if(width < j){
263                 cout << "Not a valid col lacation";
264                 isValidLocation = !isValidLocation;
265         }
266
267
268         // check if space
269         if(isValidLocation){
270                 if(myMaze[i][j] == 't'){
271                         isTracked = true;
272                 }
273         }
274
275         return isTracked;
276
277 }
278
279 void printArray(char **array, int row, int col){
280         for(int i=0 ; i<row ; i++){
281                 for(int j=0 ; j<col ; j++){
282                         cout << array[i][j];
283                 }
284                 cout << endl;
285         }
286 }
287
288
289 // Checks for empty spots in direction order east, north, south, west
290 // Marks a 't' on every traveled spot
291 bool bruteCheckForEmpty(char **myMaze, int i, int j, struct maze structMaze){
292         if(isEmpty(myMaze,i,j+1,structMaze.rows,structMaze.cols)){
293                 myMaze[i][j] = 't';
294                 bruteForceMazeSolver(myMaze, i, j+1, true, 0, 0, structMaze);
295         }
296         else{
297                 if(isEmpty(myMaze,i+1,j,structMaze.rows,structMaze.cols)){
298                         myMaze[i][j] = 't';
299                         bruteForceMazeSolver(myMaze, i+1, j, true, 0, 0, structMaze);
300                 }
301                 else{
302                         if(isEmpty(myMaze,i-1,j,structMaze.rows,structMaze.cols)){
303                                 myMaze[i][j] = 't';
304                                 bruteForceMazeSolver(myMaze, i-1, j, true, 0, 0, structMaze)
                                        ;
305                         }
306                         else{
307                                 if(isEmpty(myMaze,i,j-1,structMaze.rows,structMaze.cols)){
308                                         myMaze[i][j] = 't';
309                                         bruteForceMazeSolver(myMaze, i, j-1, true, 0, 0,
                                                structMaze);
310                                 }
311                                 else{
312                                         return false;
313                                 }
314                         }
315                 }
316         }
317         return true;
318 }
319
320
321 // Runs the same thing as bruteCheckForEmpty, except places 'x' and
322 // does not run if the finish is adjacent. Without the check, the program
323 // defaults to looking for 't' spaces rather than stopping
324 bool bruteCheckForTraveled(char **myMaze, int i, int j, struct maze structMaze){
325         if(!isFinishAdjacent(myMaze,i,j)){
326                 if(hasBeenChecked(myMaze,i,j+1,structMaze.rows,structMaze.cols)){
327                         myMaze[i][j] = 'x';
```

```
328                         bruteForceMazeSolver(myMaze, i, j+1,true, 0, 0, structMaze);
329                 }
330                 else{
331                         if(hasBeenChecked(myMaze,i+1,j,structMaze.rows,structMaze.cols)){
332                                 myMaze[i][j] = 'x';
333                                 bruteForceMazeSolver(myMaze, i+1, j, true, 0, 0, structMaze)
                                        ;
334                         }
335                         else{
336                                 if(hasBeenChecked(myMaze,i-1,j,structMaze.rows,structMaze.
                                        cols)){
337                                         myMaze[i][j] = 'x';
338                                         bruteForceMazeSolver(myMaze, i-1, j, true, 0, 0,
                                                structMaze);
339                                 }
340                                 else{
341                                         if(hasBeenChecked(myMaze,i,j-1,structMaze.rows,
                                                structMaze.cols)){
342                                                 myMaze[i][j] = 'x';
343                                                 bruteForceMazeSolver(myMaze, i, j-1, true,
                                                        0, 0, structMaze);
344                                         }
345                                         else{
346                                                 return false;
347                                         }
348                                 }
349                         }
350                 }
351         }
352         return true;
353 }
354
355 // checks all directions for adjacent 'F'
356 bool isFinishAdjacent(char **myMaze, int i, int j){
357
358         if(myMaze[i+1][j]=='F'){
359                 return true;
360         }
361         if(myMaze[i-1][j]=='F'){
362                 return true;
363         }
364         if(myMaze[i][j+j]=='F'){
365                 return true;
366         }
367         if(myMaze[i][j-j]=='F'){
368                 return true;
369         }
370
371         return false;
372 }
373
374 //recursion!!
375 //Mark current location
376 //Base Case: Look north, south, east, west for victory!
377 //Mark our path
378 //Try going south if it is open
379 //Try going north if it is open
380 //Try going east if it is open
381 //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***ttt***
**  *******t***t*****
**     ***t***t*****
******ttttxx*t******
*   ttt********t*****
*** t****** ******t***
*****t***   ******t***
*****tttttt*******t***
**********tttttt******
********************
Brute distance: 91
```