<u>Solution Report by Rajkumar Ananthu for MathWorks India</u>

Definitions:
<u>Solution:</u> A path from R to D which consumes an energy which is less than or equal to the limit specified.
<u>Solution Space:</u> A set of all the possible paths which can be searched for a possible solution.

Assumptions:
- The input terrain will be in the form of either a square or a rectangular matrix.
- Energy consumed to cross a terrain cell is a single digit i.e., 0-9.
- There exists atleast one solution for the given input.
- All the possible solutions are of atleast length 2 (length of path > 1).

Answers to Advanced Questions:
<u>1) Time and Space Complexity:</u>

Here the problem statement doesn't talk about cycles in the path to find a solution. And it is possible to have a solution to the problem which can have a cycle in it. If cycles are to be considered, then the solution space to search for a valid solution will become infinite and it becomes impossible to solve the problem and estimate the complexities.

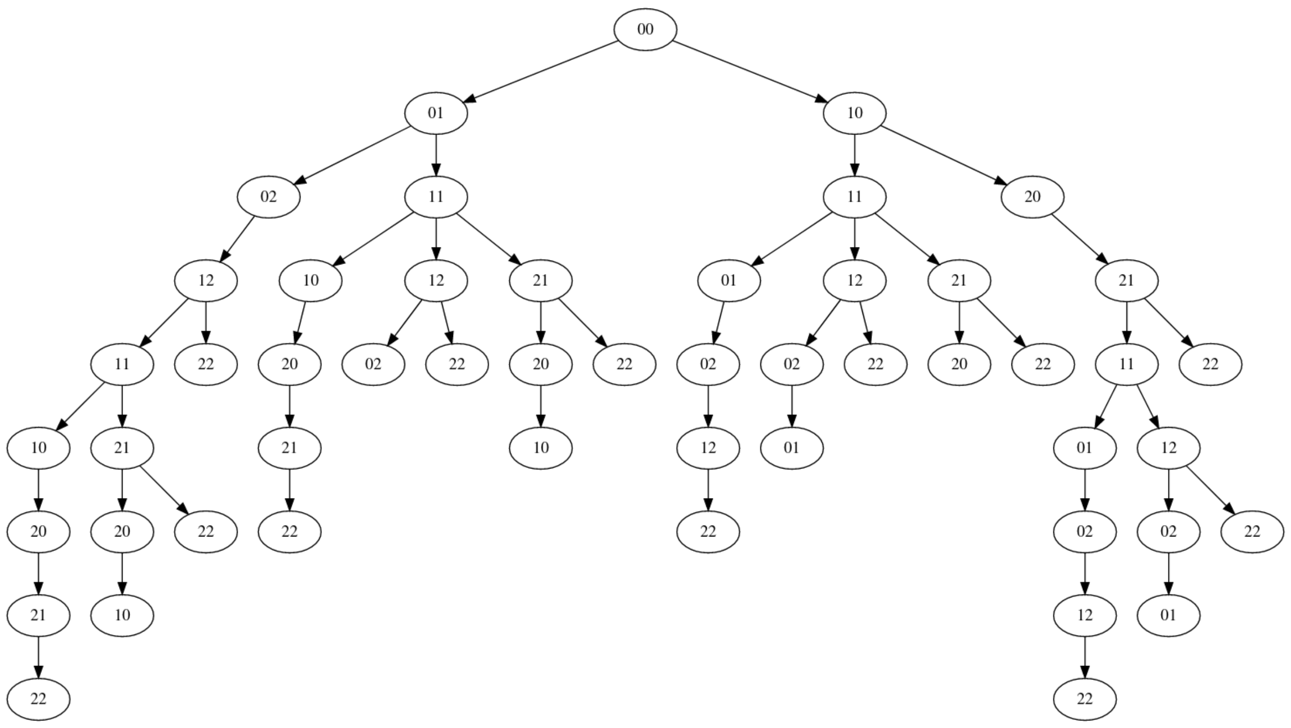
As the focus is on the shortest walk possible, paths which contain cycles in it can be removed from the solution space, after all if there is a solution A with cycle in it, then we can remove the cycle from solution A and arrive at solution B which is even a better solution.

To explore the complete solution space, there is a need to traverse all the possible path from a Robot cell, and as it is a 2D terrain there are four possibilities at each cell i.e., left cell, right cell, top cell, down cell. After each possibility, there is a need to backtrack to current cell and traverse the next possibility. The time complexity starting from R can be specified as follows, as there can be 4 possibilities from R:

$$\begin{aligned}\text{Time Complexity} &= \text{Size of the solution space}\\ &= T(\text{left}) + T(\text{right}) + T(\text{top}) + T(\text{down})\\ &= 4 * T(\text{left})\end{aligned}$$

Where T(left) is the time required to search for all possibile paths starting with left of the Robot cell.

Recursion based approach is used to solve the problem, so a sample run of the algorithm on a 2D terrain of size 3x3 with R at (0,0) and D at (2,2) will result in the following recursion tree:
label of nodes are xy – representing the (x,y) co-ordinates of cell traversed



From here:

Size of the solution space = number of leaf nodes in the recursion tree

After the first level, at each node there are at most 3 possiblities only, this is because traversing in reverse direction is not considered as the paths which have any cycles/loop are to be eliminated.

To simplify things, say that the recursion tree from second level onwards is a complete 3-ary tree in which all the levels are complete filled. 3-ary representing the 3 possibilities of traversal at a second node after moving from R.

So the time complexity of the alogorithm can be constructed as follows:

Time Complexity = Size of the solution space
= 4 * T(left)
= 4 * number of leaf-nodes in a complete 3-ary tree
= 4 * 3^h (h represents height of complete 3-ary tree)

The max height is the longest path between R and D, which is $O(n^2)$.

So the final worst case time complexity can be represented by

$$T(n) = O(3^{n\text{^}2})$$

If the terrain is a rectangular matrix of size mxn then the time complexity can be given by:

$$T(n) = O(3^{mn})$$

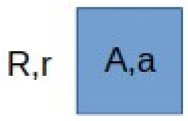
And for the space complexity as recursion is used, space complexity can be calculated as the maximum number of recursive calls made.

Space Complexity = maximum number of recursive calls made
= h (height of the complete 3-ary tree) + 1
= $O(n^2)$

2) Optimizaions:

Optimization 1:

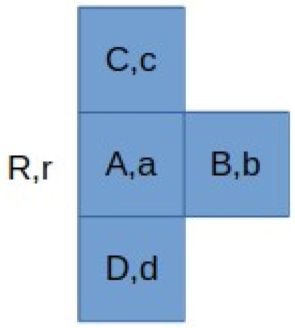
- As per the problem statment and below diagram, assume that amount spent so far by R is ‘r’ and it needs to spent ‘a’ joules to traverse through the terrain cell A.



- If the sum of energy spent so far by the Robot and the energy that is needed to cross the terrain cell A is greater than the limit, i.e., r + a > energyLimit, then that is not a possible solution, so stop traversing further down that path and can backtrack to other possibility.

Optimization 2:

- Along with optimization1, check for a look ahead cell as explained below to further more optimize solution.



- Assume the amount spent so far by R is ‘r’ and it needs ‘a’ joules to traverse through the terrain cell A, ‘b’ joules to traverse through terrain cell B, ‘c’ joules to traverse through terrain cell C, ‘d’ joules to traverse through terrain cell D. *And none of B,C,D is a destination cell*.

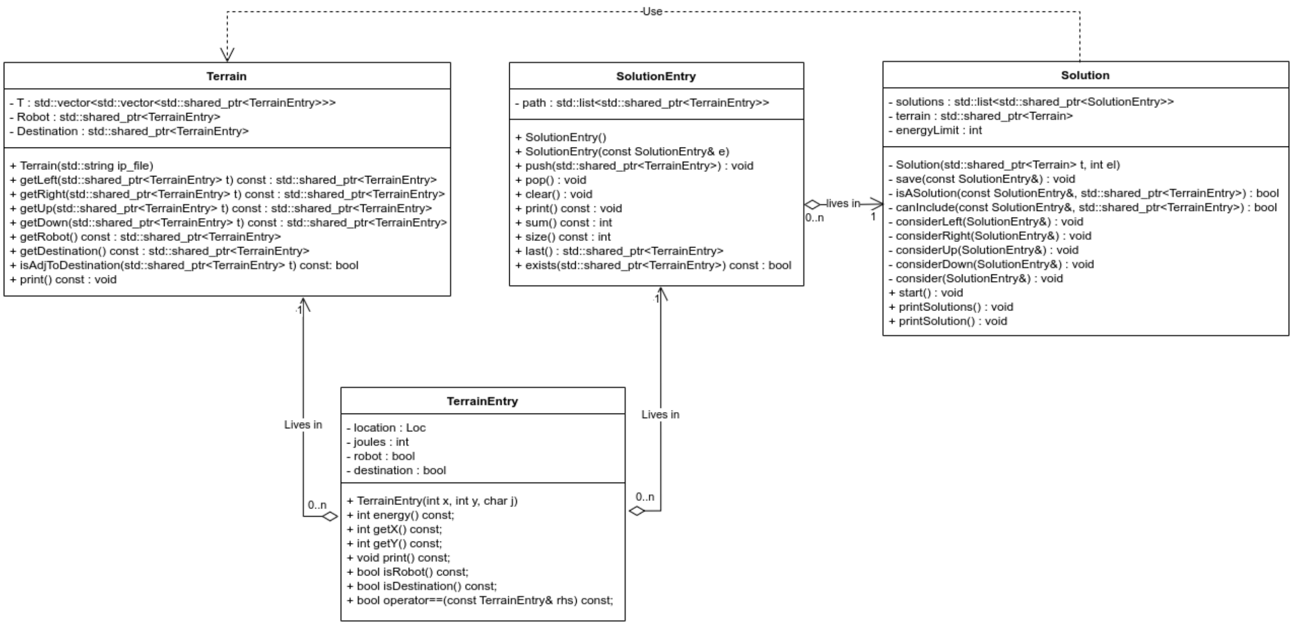
- Now consider temporarily that we cross the cell A, then we only have 3 more possibilities, traverse to cell B, C, or D.
But if all the below equations hold true, then there is no point in crossing the terrain cell A:

$r + a + b > energyLimit$ &&
$r + a + c > energyLimit$ &&
$r + a + d > energyLimit$

- Thus using a look ahead, solution can be optimized more.

3) Class Diagrams and Data strucutures used:

Class Diagram:
Note: Please refer to the image class_diagram.png in the zip folder if diagram is not clear.


Use
Terrain
- T : std::vector<std::vector<std::shared_ptr<TerrainEntry>>>
- Robot : std::shared_ptr<TerrainEntry>
- Destination : std::shared_ptr<TerrainEntry>
+ Terrain(std::string ip_file)
+ getLeft(std::shared_ptr<TerrainEntry> t) const : std::shared_ptr<TerrainEntry>
+ getRight(std::shared_ptr<TerrainEntry> t) const : std::shared_ptr<TerrainEntry>
+ getUp(std::shared_ptr<TerrainEntry> t) const : std::shared_ptr<TerrainEntry>
+ getDown(std::shared_ptr<TerrainEntry> t) const : std::shared_ptr<TerrainEntry>
+ getRobot() const : std::shared_ptr<TerrainEntry>
+ getDestination() const : std::shared_ptr<TerrainEntry>
+ isAdjToDestination(std::shared_ptr<TerrainEntry> t) const: bool
+ print() const : void
SolutionEntry
- path : std::list<std::shared_ptr<TerrainEntry>>
+ SolutionEntry()
+ SolutionEntry(const SolutionEntry& e)
+ push(std::shared_ptr<TerrainEntry>) : void
+ pop() : void
+ clear() : void
+ print() const : void
+ sum() const : int
+ size() const : int
+ last() : std::shared_ptr<TerrainEntry>
+ exists(std::shared_ptr<TerrainEntry>) const : bool
lives in
0..n
1
Solution
- solutions : std::list<std::shared_ptr<SolutionEntry>>
- terrain : std::shared_ptr<Terrain>
- energyLimit : int
- Solution(std::shared_ptr<Terrain> t, int el)
- save(const SolutionEntry&) : void
- isASolution(const SolutionEntry&, std::shared_ptr<TerrainEntry>) : bool
- canInclude(const SolutionEntry&, std::shared_ptr<TerrainEntry>) : bool
- considerLeft(SolutionEntry&) : void
- considerRight(SolutionEntry&) : void
- considerUp(SolutionEntry&) : void
- considerDown(SolutionEntry&) : void
- consider(SolutionEntry&) : void
+ start() : void
+ printSolutions() : void
+ printSolution() : void
1
Lives in
0..n
1
Lives in
0..n
TerrainEntry
- location : Loc
- joules : int
- robot : bool
- destination : bool
+ TerrainEntry(int x, int y, char j)
+ int energy() const;
+ int getX() const;
+ int getY() const;
+ void print() const;
+ bool isRobot() const;
+ bool isDestination() const;
+ bool operator==(const TerrainEntry& rhs) const;

Data Structures Used:

1. Adjacency List:
To represent the 2D terrain (matrix) structure. This is achieved using vector of vectors (std::vector<std::vector<std::shared_ptr<TerrainEntry>>>).

2. Linked List:
This data structure is used at 2 places, one is to hold each solution (std::list<std::shared_ptr<TerrainEntry>>) other is to hold all the possible list of solutions (std::list<std::shared_ptr<SolutionEntry>>).

Sample Runs/Output Screenshots/Memory leak checks:

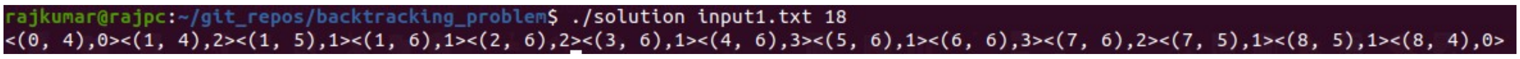
Compilation command:

```
g++ main.cpp solution.cpp terrain.cpp --std=c++11 -g -o solution
```

Output is the shortest possible path from R to D which will satisfy the constraint.

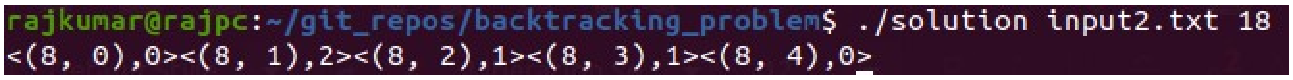
<(x,y),j> = (x,y) represent the co-ordinates of the terrain cell, j represents joules required to travel through that particular terrain cell.

Run1:



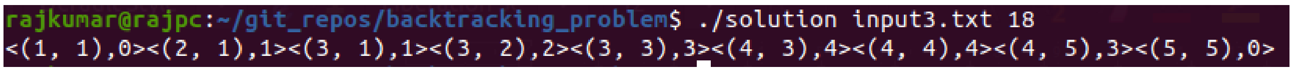
```
rajkumar@rajpc:~/git_repos/backtracking_problem$ ./solution input1.txt 18
<(0, 4),0><(1, 4),2><(1, 5),1><(1, 6),1><(2, 6),2><(3, 6),1><(4, 6),3><(5, 6),1><(6, 6),3><(7, 6),2><(7, 5),1><(8, 5),1><(8, 4),0>
```

Run2:



```
rajkumar@rajpc:~/git_repos/backtracking_problem$ ./solution input2.txt 18
<(8, 0),0><(8, 1),2><(8, 2),1><(8, 3),1><(8, 4),0>
```

Run3:



```
rajkumar@rajpc:~/git_repos/backtracking_problem$ ./solution input3.txt 18
<(1, 1),0><(2, 1),1><(3, 1),1><(3, 2),2><(3, 3),3><(4, 3),4><(4, 4),4><(4, 5),3><(5, 5),0>
```

Command to check for memory leaks if any using valgrind:

```
valgrind --leak-check=full --show-leak-kinds=all --log-file=valgrind.txt ./solution input1.txt 18
```

The log(valgrind.txt) is in the solution zip folder passed.

Contents of the zip folder:
Source files:- main.cpp, terrain.cpp, terrain.h, solution.cpp, solution.h
Test input files:- input1.txt, input2.txt, input3.txt
Problem Statement:- InterviewProblem.docx
Solution Report:- SolutionReport.pdf
valgrind log:- valgrind.txt
Class diagram:- class_diagram.png