BFS :

DataStructure used :

- Queue: contain the nodes that had been visitied but their child not yet.

- Array of LinkedLists : contain the path for each visitied node from it to the start point.

- Boolean array: to determine if the node is visited or not.

Algoritm structure:

First : I search for the start point and keep its position in variables to indicate it easily later.
Second : for the four child nodes if :
- out of boundx => escape.
- contain visited ==> escape.
- is wall => escape.

is target :
- mark it as visited.
- add the parent path to its linkedlist.
- add its position to its linkedlist.
- transfer the linkedlist to an array.
- return the result.

is empty cell:
- mark it as visited.
- add the parent path to its linkedlist.
- add it to the queue.
- else throw null.

Third:While the Queue is not empty:
- pop the first node
- for the four child nodes if:
  - out of boundx => escape.
  - contain visited ==> escape.
  - is wall => escape.

  target :
  - mark it as visited.
  - add the parent path to its linkedlist.
  - add its position to its linkedlist.
  - transfer the linkedlist to an array.
  - return the result.

  empty cell:
  - mark it as visited.
  - add the parent path to its linkedlist.
  - add it to the queue.
  - else throw null.

DFS :

DataStructure used :

- Stack : contain the nodes that had been visitied but their child not yet.

- LinkedLists : contain the path for each visitied node from it to the start point.

- Boolean array: to determine if the node is visited or not.

Algoritm structure:

First : I search for the start point and keep its position in variables to indicate it easily later.

Second : for the four child nodes if :
- out of boundx => escape.
- contain visited ==> escape.
- is wall => escape.

is target :
- mark it as visited.
- add its position to its linkedlist.
- transfer the linkedlist to an array.
- return the result.

is empty cell:
- mark it as visited.
- add it to the queue.

Third:While the Queue is not empty:
- pop the first node
- for the four child nodes if:
  - out of boundx => escape.
  - contain visited ==> escape.
  - is wall => escape.

  target :
  - mark it as visited.
  - add its position to its linkedlist.
  - transfer the linkedlist to an array.
  - return the result.

  empty cell:
  - mark it as visited.
  - add it to the queue.
- else throw null.

Comparsion

BFS	DFS
BFS Stands for "**Breadth First Search**".	**DFS** stands for "**Depth First Search**".
BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node.	DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise.
Breadth First Search can be done with the help of **queue** i.e. **FIFO** implementation.	Depth First Search can be done with the help of **Stack** i.e. **LIFO** implementations.
This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once. BFS is **slower** than DFS.	This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped-off. DFS is faster than BFS.
BFS requires **more** memory compare to DFS. **Applications of BFS** > To find Shortest path > Single Source & All pairs shortest paths > In Spanning tree > In Connectivity	DFS require **less** memory compare to BFS. **Applications of DFS** > Useful in Cycle detection > In Connectivity testing > Finding a path between V and W in the graph. > useful in finding spanning trees & forest.
BFS is useful in finding shortest paths can be used to find the shortest distance between some starting node and the remaining nodes of the graph. **Example:**  A, B, C, D, E, F	DFS in not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (**backtrack**) to add branches also as long as possible. **Example:**  A, B, D, C, E, F

Sample Space

First :

```
3 5
##.S.
..E..
..#..
```

```
03
02
12
```

```
03
13
23
24
14
12
```

Second:

```
10 10
##.S.##..#
.##..##..#
..##.##..#
##...###.#
...#.###..
##.####...
..E..###..
..#.....##
###..####.
..##.##...
```

```
03
13
14
24
34
44
33
32
42
52
62
```

```
03
04
14
24
34
33
32
42
52
62
```

Third :

```
3 3
.#.
S.#
..E
```

```
10
11
21
22
```

```
10
11
21
22
```

Test no target :

```
3 3
.#.
S.#
...
```

```
Exception in thread "main" java.lang.NullPointerException
        at eg.edu.alexu.csd.datastructure.maze.cs55.Maze.sss(Maze.java:183)
        at eg.edu.alexu.csd.datastructure.maze.cs55.Maze.solveBFS(Maze.java:46)
        at eg.edu.alexu.csd.datastructure.maze.cs55.Ssssssss.main(Ssssssss.java:27)
```

```
Exception in thread "main" java.lang.NullPointerException
        at eg.edu.alexu.csd.datastructure.maze.cs55.Maze.solveDFS(Maze.java:234)
        at eg.edu.alexu.csd.datastructure.maze.cs55.Ssssssss.main(Ssssssss.java:27)
```