Report:

**Used data structures:**

* **Stack:for depth-first-search:**
* **Queue:for breadth-first-search**

**Algorithms:**

1. **Dfs algorithm:**

**Let S be a stack**

**s.push (sourcepoint)**

**while (stack is not empty)**

**testingpoint = s.pop()**

**if testingpoint is the end point**

**show final path**

**else if isVisited or a wall#**

**continue**

**else**

**mark as visited and search**

**west direction :if ! visited and not outofbounds,s.push(point)**

**south direction :if ! visited and not outofbounds,s.push(point)**

**east direction :if ! visited and not outofbounds,s.push(point)**

**north direction :if ! visited and not outofbounds,s.push(point)**

**2-BFS algoritm:**

Let q be a queue

q.enqueue(sourceppoint)

while(!q.isEmpty())

**testingpoint = q.dequeue()**

**if testingpoint is the end point**

**show final path**

**else if isVisited or a wall#**

**continue**

**else**

**mark as visited and search**

**north direction :if ! visited and not outofbounds,** q.enqueue **(pt)**

**east direction :if ! visited and not outofbounds,** q.enqueue **(pt)**

**south direction :if ! visited and not outofbounds,** q.enqueue **(pt)**

**west direction :if ! visited and not outofbounds,** q.enqueue **(pt)**

**Comparison between both algorithms and how they operate?:**

**Depth first search:**

* It starts at the  root and explores as far as possible along each branch before [backtracking](https://en.wikipedia.org/wiki/Backtracking).
* takes time Θ(|V| + |E|)
* **Vertex orderings**

It is possible to use the depth-first search to linearly order the vertices of the original graph (or tree).

* A **preordering** is a list of the vertices in the order that they were first visited by the depth-first search algorithm.
* A **postordering** is a list of the vertices in the order that they were *last* visited by the algorithm.
* A **reverse postordering** is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. Reverse postordering is not the same as preordering.

**Breadth first search:**

* It starts at the [tree root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) and explores the neighbor nodes first, before moving to the next level neighbors.
* The non-recursive implementation is similar to the non-recursive implementation of [depth-first search](https://en.wikipedia.org/wiki/Depth-first_search), but differs from it in two ways:

1. it uses a [queue](https://en.wikipedia.org/wiki/Queue_(abstract_data_type)) instead of a [stack](https://en.wikipedia.org/wiki/Stack_(abstract_data_type)) and
2. it checks whether a vertex has been discovered before enqueueing the vertex rather than delaying this check until the vertex is dequeued from the queue.

* The *distance* attribute of each vertex (or node) is needed for example when searching for the shortest path between nodes in a graph
* The time complexity can be expressed as O(|V|+|E|)

**Details needed to be clarified:**

* Dfs : the search order needed to be reversed because the ADT used in the dfs search is a stack so the search was in this order: west, south, east, north in order to find the path according to the the required order
* before searching in the four allowed directions of a cell;the indices of the cell were stored in the parent array which will be used to determine the final path by iterating backwards over it

in the dfs and bfs

* a multiple entering points or multiple target points are considerd an error and an exception is throwed
* a maze without target or entry points is an error and an exception is throwed
* when no path is found ,both search algorithms return null meaning that the parent array is empty
* reading the maze inputs from a file where the first line contains the dimensions of the 2d array where the file inputs will be stored

**Sample runs for both algorithms:**

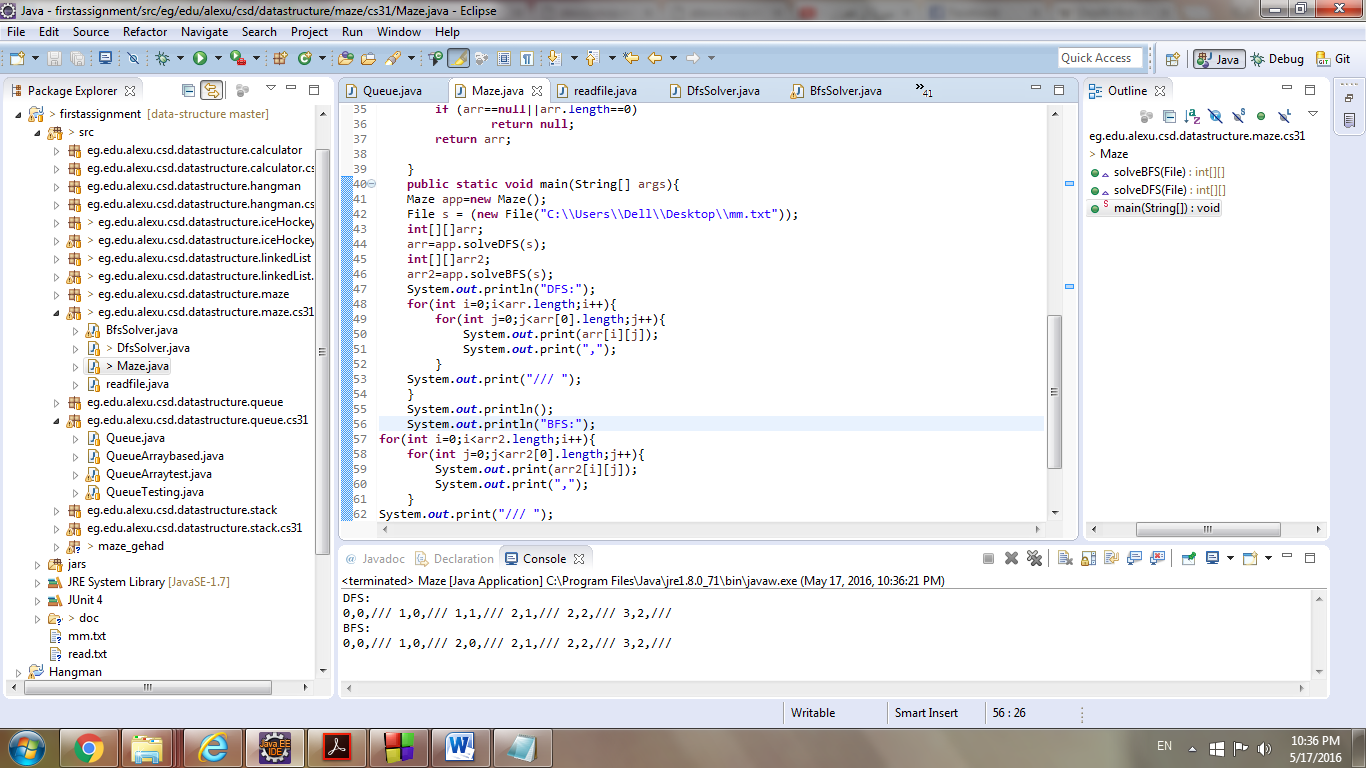
**For example:solving this maze is different in both algorithms:**

**4 5**

**S#..#**

**..##.**

**...#.**

**##E#.**