QUESTION 1

First I take n and m as inputs from the user and then fill the 2D array randomly.

Then I keep 2 arrays and 1 variable for maximum score:

```
path = []
points = [[0, 0]]
score = pathFinder(array, 0, 0, path)
```

pathFinder function takes 2D array, current row and column and also the path array. In this function firstly I call the same pathFinder function 2 times for the 2 movements. Then I choose between these two paths according to their total scores that they have returned and then add this path to my path and change the score accordingly. I send 2 different arrays for 2 different paths to these functions and then fetch the data inside this arrays properly. If I am at the end of the row, I continue downwards; if I am at the end of the column, I continue to the right. At the end, I return total score and path array is changed via side effect.

path array is not in very well form but I handle it and take the data inside and also fill points array according to the path.

I have used brute force algorithm that traverses every possible path so this is costly. Worst case is:

- $\theta(2^{n*m})$

  because game array's size is n*m. In each recursive call, the function can either go down or right, so, at each step, the number of possible paths doubles, leading to this complexity.

QUESTION 2

Here I have used the LomutoPartition and QuickSelect algorithms that we have seen at lectures. First I take the array size as input from the user and then fill the array randomly. I have set the k to half of the array size to find k^{th} smallest element (median) in the array. For even number of elements, I have also found the $k^{th}+1$ smallest element so that I can find the average of these two.

In LomutoPartition, I do the partition by choosing rightmost array element and returning its proper position.

In QuickSelect, first I do the partition to find the correct place of the current element. After comparisons, I continue either with left part of the pivot or right part of the pivot recursively.

An extremely unbalanced partition results in worst case time complexity (array's 1 part empty and other part is with n-1 elements while doing partitioning). This can happen on each step:

- Cworst(n) = (n-1) + (n-2) + ... + 1 + 0 = n(n-1)/2 = $\theta(n^2)$

QUESTION 3

(a)

I have created a CircularLinkedList class in addition to Node class that keeps data. This class has 3 methods other than constructor:

- add
  - Adds data to the list. This addition doesn't ruin the circular property by handling the head node properly (assigning next node of the lastly added node to head).
- addPlayers
  - Adds given number of players to the list.
- winner
  - Eliminates one player at each iteration of the given loop by going two next element in the list until node's next is itself (until there is only 1 element left)

Adding players to list takes linear time, and choosing winner of the game takes linear time too because the algorithm iterates through the linked list until there is only one node left. Each iteration removes one player from the game, so the number of iterations is equal to number of players-1. This results in linear time.

(b)

For this algorithm, I need to reduce the problem size each time. My base case is returning 1 if there is only 1 player in the game (she/he is the winner). At each round, we are eliminating $\left\lfloor \frac{n}{2} \right\rfloor$ players. Hence, I have controlled if player number is even or not and accordingly call the function recursively by dividing the player number by 2 or dividing player number - 1 by 2. If we start with n players, $\left\lfloor \frac{n}{2} \right\rfloor$ players left after the first round. The winner will be the person in position, let's say, x now. That person was at position 2x – 1 (or 2x + 1 for odd number of players) formerly. That means winner is at the position 2x – 1 formerly. Hence, I recursively continue with using this formulation.

T(n) = T(n/2) + 1 ---> at each step, I decrease the size by half. So this cut in half process makes the running time logarithmic.

QUESTION 4

Binary Search:

- T(n) = T(n/2) + 2 (2 is for 2 comparisons) ----------> $O(\log_2 n)$
- 2 comparisons:
  - if middle element equals to searching element
  - if middle element greater than (or less than) the searching element

Ternary Search:

- T(n) = T(n/3) + 4 (4 is for 4 comparisons) ----------> $O(\log_3 n)$
- 4 comparisons:
  - if first middle element equals to searching element
  - if second middle element equals to searching element
  - if first middle element greater than the searching element
  - if second middle element less than the searching element

There are more comparisons at each step of the ternary search. There may be less number of steps at the ternary search on average but more number of comparisons at each step.

With bigger divisor, we may finish the searching in less steps but with way more comparisons and comparison number is the most important thing in the searching. For example, if we divide the array with size n into n parts, we need to do 1 step but n comparisons so time complexity would be linear: $\theta(n)$

QUESTION 5

(a)

Best-case scenario of interpolation search happens when searching value is at the middle of the search range. We find on the first iteration and time complexity becomes O(1).

(b)

Interpolation search is similar to binary search but instead of going to the middle element, interpolation search may go to different locations by estimating the position of the target value within the range according to searching element.

Best-case scenario of both of them is constant when the searching value is at the middle.

Worst-case scenario of interpolation search is linear because it may require more iterations to shrink the search range (when items are distributed exponentially). It is worse than the binary search's worst case which is logarithmic. Interpolation search can take longer to find the target value if it is not located middle of the range, as it may require additional iterations to narrow down the search range.

On the other hand, if values are uniformly distributed or searching value is likely to be located near middle of the range, interpolation search can be useful. On average, binary search takes O(logn) time while interpolation search takes O(log(logn)) time. This makes interpolation search better for average case.