Answer to Question - 1

First approach is that:
Selecting the bigger one in each selection didn't give correct result each time. Because, if the last element of a path is too big from the other smaller numbers, algorithm will give the result of bigger-begginning algorithm. So that first we need to traverse all possible results.Holding the subproblems from somewhere and solving them is called: dynamic programming.

Every time we have two options: right or down. This remind me the tree structure. If I put the each branching into nodes and construct a tree from them, I can easily traverse the tree with one of the searches, and easily find the max path. So I decided to use tree structure to represent the possible paths.

- Create a tree to represent the possible paths through the map. The root node of the tree represents the starting position, which is (0,0)
- Use breadth-first search to traverse the tree and populate it with the possible paths.
- Until all possible paths added to the tree, add the right and left (down) children of each node to the queue, if movement is valid.
- Traverse the tree and find the maximum value using depth-first search, by adding the right and left(down) children of each node to the stack.
- Update the maximum value if necessary
- Return the maximum value

Analyze the worst-case time complexity:

It is O(n*m), where n and m are the dimensions of the board.

The algorithm traverses the entire tree, which has a total of n*m nodes, and operation performed takes constant time at each node. Therefore, the time complexity is depends on the number of nodes in the tree, which is n*m.

Worst case is the case where the board is full, the algorithm will have to visit every node in the tree. This results time complexity of O(n*m).

Answer to Question - 2

Each iteration of the variable-size-decrease technique results in a different size-reduction pattern. By sorting the list first and choosing the kth element from the output, the kth smallest element in a list can be found. Sorting is the most expensive component of the algorithm. Since the problem just requests for the kth smallest element in the list to be found, sorting the full list is probably unnecessary. To reduce complexity, an efficient sorting algorithm should be select. For example mergesort (dividing an array's elements into two) can be as efficient as 0(n log n).

Here we describe the Lomuto partitioning for dividing an array.
Consider a subarray A[l..r] where (0 ≤ l ≤ r ≤ n − 1) as being formed of three adjacent segments:

Create an array and call Quickselect function. By doing that, median should be send as ceiling of the value, after it divided with 2. Continue until the return value that comes from the partitioning is equal to the k-1 (k: (1 ≤ k ≤ r − l + 1))

According to the coming s value is bigger than or smaller than k value, update lower bound and upper bound and send again to the Quickselect function that new values.

Analyze the worst-case time complexity:

It is O(n*m), where n and m are the dimensions of the board.

Answer to Question - 3

A) Design an algorithm that finds the winner of the game, by using a circular linked list. Makesure your algorithm runs in linear time.

First start by taking the input for the player amount. After that, create a circular linked list. Assign the all elements (Node's values) of the circular linked list with 0.

0: Non-eliminated players
1:Eliminated players

Call the eliminatePlayerGame and send the created circular ll.
This function should start from the index that is head of the circular LL, and continue until there exists only one player, that has value 0 in it's data part. So that traversing whole list once is obligation. And this will be perform in the outer while loop. So that we don't have chance to use any complex time operation inside that while loop. But we have this case: When current player is 0 but the next player has 1, so that current player should look next of next. And may be need to traverse all next of nexts. How can we perform it in efficeist way?

Using dictionary. Hold a dictionary, to store the count of 1s in the list. Find the first zero-value player with searching in the dict, but it will return the index of the first player that has value 0. So that it will takes less time than O(n). So I use recursion for that.

```
def find_first_zero(ones_count, next_zero, next_index, mod):
    i = 0
    for i, value in ones_count.items():
        if value == 0:
            return i
        else:
            find_first_zero(ones_count, next_zero.next, (next_index + 1) % mod, mod)
```

This A part is implemented in the eliminatePlayerGame/eliminatePlayerGame_linear.

B) Design a decrease-and-conquer algorithm that finds the winner of the game. Make sureyour algorithm runs in logarithmic time.

To be able to perform decrease and conquer, we need to convert our functions to recursive versions. This B part is implemented in the eliminatePlayerGame_recursion.

Answer to Question - 4

- Compare the time complexities of ternary search and binary search:

In ternary search, the array is split into three equal parts, whereas in binary search, the array is split into two equal parts at each step. As a result, ternary search requires one additional division at each step than binary search. Binary search has an $O(\log_2 n)$ time complexity, while ternary search has an $O(\log_3 n)$ time complexity. As the base of the logarithm increases, the time complexity of the algorithm decreases, $log_3 n < log_2 n$. But we can say that their complexities are same, due to they're both logarithmic time.

- Explain how the divisor affects the complexity of the search algorithm.

Divisor means the number of parts into which the array is divided at each step, in a search algorithm. The amount of divisions the algorithm makes at each stage is determined by divisor. Base of the logarithm represents the "how many parts that array is divided". The effect of the divisor is when it is getting larger, less divisions will be performed. This leads to gets faster to find the desired element.

- Assuming the array has n elements, what does the time complexity of the algorithm become if we divide it into n parts at the beginning?

O(n). Dividing the array into n is the same thing with linear search or sequential search. The array is not just divided into parts at the beginning, but rather it is divided into smaller parts at each step. It is not possible to continue divide the individual sections into n parts during iterations after dividing the array into n parts at the beginning. The individual components are now just elements rather than arrays.

Answer to Question - 5

C) What is the best-case scenario of interpolation search? What is the time complexity of it?

It is O(1), because there is possibility of finding the element at first lookup. According to below formula:

$$x = l + \left\lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \right\rfloor$$

where;

x is the index to be searched,
l is the lowest index,
r is the highest index,
A[l] is the element in the lowest index,
A[r] is the element in the highest index,
v is the search key's value.

D) What is the difference between interpolation search and binary search in terms of the manner of work and the time complexity?

Interpolation search is a variable-size-decrease algorithm, that for searching in a sorted array of n uniformly distributed values. Binary search is also works on sorted array. In interpolation search, we're starting from a calculated index, according to our value to be searched; whereas in binary search we're starting from middle index. Due to interpolation search starts from a "closer" estimated index, it is more efficient than binary search in average case. But in worst case,like non-linear arrays, interpolation needs more comparison because interpolation algorithm assumes that the array values increase linearly. In that case binary search is more efficient.

	Interpolation Search	Binary Search
Best Case	O(1)	O(1)
Worst Case	O(n)	O(log n)
Average Case	O(log log n)	O(log n)