Q1) This algorithm is a recursive function that attempts to find the path from a current coordinate to a destination coordinate on a given map that yields the maximum score, where the score is calculated by the sum of the values of the cells visited along the path.

The function takes the current coordinate, the destination coordinate, the map, the current score, the maximum score seen so far, the current path, and the maximum path seen so far as input, and returns the maximum score and the maximum path as output.

The function first checks if the current coordinate is equal to the destination coordinate. If it is, it compares the current score to the maximum score seen so far and updates the maximum score and the maximum path if necessary. If the current coordinate is not equal to the destination coordinate, the function considers two possibilities: moving right or moving down. For each possibility, the function calculates the new score by adding the value of the cell at the new coordinate to the current score, adds the new coordinate to the current path, and calls itself recursively with the updated information. After the recursive call returns, the function removes the last element from the current path, effectively backtracking to the previous step.

The worst-case time complexity of this algorithm is O(2^n), where n is the number of cells in the map. This is because the function may consider both possibilities for each cell in the map, leading to an exponentially increasing number of recursive calls.

Q2) This algorithm is a recursive function that finds the median value of a given array of numbers.

The function takes an array of numbers as input and returns the median value as output. It first checks if the array has a length of 0 or 1. If it does, it returns the only element in the array (or None if the array is empty). If the array has a length greater than 1, the function divides the array into two halves, finds the medians of the two halves recursively, and then returns the median of the array based on whether the length of the array is odd or even.

The worst-case time complexity of this algorithm is O(n), where n is the length of the array. This is because the function divides the array into two halves and calls itself recursively on each half until it reaches the base case where the array has a length of 0 or 1. The number of recursive calls is therefore equal to the number of times the array can be halved until it reaches a length of 0 or 1, which is logarithmic in the length of the array.

Q3)

a) This algorithm is implementing a circular linked list. In this problem, there are n people standing in a circle, and every mth person is eliminated until only one person is left standing. The last remaining person is the winner.

The algorithm first initializes a linked list with the players by setting the next and prev attributes of each player object. Then, it starts a loop that runs until there is only one player left in the circle. In each iteration of the loop, the player immediately after the current player (pointer) is eliminated, and the pointer is updated to the player after the eliminated player. This simulates the process of eliminating every mth player from the circle.

The time complexity of this algorithm is O(n), since the loop runs n times and the only operations performed in each iteration are a few attribute assignments and a single call to print. This is the worst-case time complexity, as it assumes that all players are eliminated, and the loop runs until there is only one player left.

b) This is a recursive function that finds the winning position in a game where n people are standing in a circle and counting off by k until only one person remains. The person who is left standing is the winner.

The function works by dividing the group of people into two subgroups: the first subgroup contains the first n // 2 people, and the second subgroup contains the rest of the people. The function then recursively finds the winning position within one of the subgroups, depending on which subgroup the survivor from the previous round belongs to. If the survivor's position is greater than n // 2, it means that they are in the second subgroup, so the function finds the winning position within the second subgroup. If the survivor's position is less than or equal to n // 2, it means that they are in the first subgroup, so the function finds the winning position within the first subgroup.

Once the winning position within the subgroup is found, the function maps the position back to the original group of n people by multiplying the position by 2 (for the first subgroup) or by multiplying the position by 2 and subtracting 1 (for the second subgroup).

The worst-case time complexity of this function is O(log n), because each recursive call reduces the size of the input by half.

```
##################### Q1 ######################
Enter the number of rows: 4
Enter the number of columns: 3
Enter the element for row 1 and column 1: 25
Enter the element for row 1 and column 2: 30
Enter the element for row 1 and column 3: 25
Enter the element for row 2 and column 1: 45
Enter the element for row 2 and column 2: 15
Enter the element for row 2 and column 3: 11
Enter the element for row 3 and column 1: 1
Enter the element for row 3 and column 2: 88
Enter the element for row 3 and column 3: 15
Enter the element for row 4 and column 1: 9
Enter the element for row 4 and column 2: 4
Enter the element for row 4 and column 3: 23
Maximum score: 211
Maximum score path: [(1, 0), (1, 1), (2, 1), (2, 2), (3, 2)]
zeroday@zeroday-Lenovo-V330-15IKB:~/Desktop/cse321_algorithm/hw4$
```

```
###################### Q2 ########################
Enter the size of the array: 4
Enter the element for index 0: 34
Enter the element for index 1: 657
Enter the element for index 2: 21
Enter the element for index 3: 65
median:  49.5
```

```
##################### Q3 part1 ######################
Enter count of players: 6
Enter player 0 name: p1
Enter player 1 name: p2
Enter player 2 name: p3
Enter player 3 name: p4
Enter player 4 name: p5
Enter player 5 name: p6
p1 eliminates p2
p3 eliminates p4
p5 eliminates p6
p1 eliminates p3
p5 eliminates p1
p5 is the winner
```

```
##################### Q3 part2 ######################
Enter count of players: 6
Enter player p3 name:  is the winner
zeroday@zeroday-Lenovo-V330-15IKB:~/Desktop/cse321_algori
```

Q4) In the case of binary search, the time complexity is O(log2n), which means that the number of steps required to perform the search is logarithmic in the size of the array. This means that as the size of the array increases, the number of steps required to perform the search increases more slowly.

On the other hand, the time complexity of ternary search is O(log3n), which means that the number of steps required to perform the search is logarithmic in the size of the array with a base of 3. This means that as the size of the array increases, the number of steps required to perform the search increases even more slowly than in the case of binary search.

The divisor, or base, of the logarithm affects the complexity of the search algorithm because it determines how quickly the number of steps required to perform the search increases as the size of the array increases. A larger base means that the number of steps required to perform the search increases more slowly, while a smaller base means that the number of steps required to perform the search increases more quickly.

If we divide the array into n parts at the beginning, the time complexity of the search algorithm becomes O(n). This is because in this case, the algorithm would need to compare the value to every element in the array in order to find the target value, which would take n steps. This means that the time complexity of the algorithm would be linear in the size of the array, which is much worse than the logarithmic time complexity of both binary search and ternary search.

Q5) Interpolation search is a search algorithm that is used to find a specific value in a sorted array. It works by using an interpolation formula to estimate the position of the target value in the array, and then checking the value at that position to see if it is the target value. If the value at the estimated position is not the target value, the algorithm can use the same interpolation formula to estimate the position of the target value in a subarray and repeat the process until the target value is found or it is determined that the target value is not present in the array.

a) The best-case scenario for interpolation search is when the target value is found at the estimated position in the first step of the algorithm. In this case, the time complexity of interpolation search is O (1), which means that the algorithm takes a constant number of steps to find the target value.

In general, the time complexity of interpolation search depends on the distribution of values in the array. If the values in the array are uniformly distributed, then the time complexity of interpolation search is O(log(n)) in the average case, as mentioned earlier. However, if the values in the array are not uniformly distributed, the time complexity of interpolation search may be worse than O(log(n)) in the average case. For example, if the values in the array are concentrated at one end of the array, the time complexity of interpolation search may be closer to O(n) in the average case.

b) In terms of manner of work, binary search works by dividing the array into two parts at each step and comparing the target value to the middle element of the array. Interpolation search, on the other hand, uses an interpolation formula to estimate the position of the target value in the array and then checks the value at that position.

The time complexity of binary search is O(log(n)) in the worst case and average case, which means that the number of steps required to perform the search is logarithmic in the size of the array. The time complexity of interpolation search is O(n) in the worst case, but it can be O(log(n)) in the average case for a uniformly distributed array. This means that interpolation search may be more efficient than binary search in the average case for a uniformly distributed array, but it is not as efficient as binary search in the worst case.