Introduction to Algorithm

CSE 321

Homework #4

Berkan AKIN

171044073

Q1) One way to solve this problem using a brute-force approach is to try every possible sequence of steps and keep track of the highest score achieved. This algorithm tries every possible sequence of steps by recursively calling find_maximum_score_from_coordinate starting from each coordinate in the map. The function tries moving to the right and moving down, and returns the maximum score achieved by either move. The base case is when the player has reached the last coordinate (i == n and j == m), in which case the current score is returned.

This algorithm has a time complexity of O(n^2 * m^2), since it tries every possible sequence of steps and the length of each sequence is bounded by the size of the map (n * m).

Note: The above pseudocode uses -99999 as a sentinel value to indicate that a move is not possible. This is just a convention and any other value that is guaranteed to be smaller than any possible score can be used instead.

Algorithm Analaysis

The brute-force algorithm for finding the maximum score in the map has a time complexity of O(n^2 * m^2), since it tries every possible sequence of steps and the length of each sequence is bounded by the size of the map (n * m).

For each coordinate in the map, the algorithm recursively calls itself to try moving to the right and moving down. The recursion continues until the player has reached the last coordinate, at which point the current score is returned.

Since the algorithm tries every possible sequence of steps, it is guaranteed to find the maximum score. However, the time complexity of the algorithm makes it impractical for large maps, as it may take a long time to complete.

To improve the performance of the algorithm, you can use other techniques such as dynamic programming, which allows you to store the results of subproblems and reuse them to avoid recalculating them multiple times. This can significantly reduce the time complexity of the algorithm and make it more practical for larger maps.

Q2) One way to design a decrease and conquer algorithm to find the median of an unsorted array is to use the Quickselect algorithm. Quickselect is a variant of Quicksort, a well-known sorting algorithm, that can be used to find the kth smallest element in an array in expected linear time.

Here is a high-level outline of how the Quickselect algorithm works:

1) Choose a pivot element from the array.
2) Partition the array into two subarrays, with all elements less than the pivot element in one subarray and all elements greater than the pivot element in the other subarray
3) If the pivot element is the kth smallest element, return it.
4) If the pivot element is less than the kth smallest element, recursively apply Quickselect to the subarray with elements greater than the pivot element.

5) If the pivot element is greater than the kth smallest element, recursively apply Quickselect to the subarray with elements less than the pivot element.

To find the median of an unsorted array using Quickselect, we can set k to be the middle element of the array (for an odd-length array) or the average of the two middle elements (for an even-length array).

The partition function is responsible for partitioning the array into two subarrays based on the pivot element. It can be implemented using two pointers, one pointing to the start of the array and the other pointing to the end. The two pointers move towards each other, swapping elements as needed, until the array is fully partitioned.

Overall, the Quickselect algorithm has an average-case time complexity of O(n) and a worst-case time complexity of O(n^2), making it an efficient algorithm for finding the median of an unsorted array.

Algorithm Analaysis

The time complexity of Quickselect is dependent on the choice of the pivot element and how well it partitions the array into two balanced subarrays.
In the average case, Quickselect has a time complexity of O(n), where n is the size of the array. This is because the pivot element is chosen such that it splits the array into two subarrays of roughly equal size, resulting in a balanced partition. This leads to a logarithmic decrease in the size of the

subarrays with each recursive call, resulting in a linear time complexity overall.
In the worst case, Quickselect has a time complexity of O(n^2). This occurs when the pivot element is always chosen as the minimum or maximum element in the array, resulting in one subarray being empty and the other subarray having n-1 elements. This leads to an unbalanced partition, causing the size of the subarrays to decrease by only 1 element with each recursive call. As a result, the time complexity becomes quadratic.
Space complexity:

The space complexity of Quickselect is O(1), as it does not use any additional space beyond the input array and a few variables for storing the pivot element and the two subarrays.
Overall, Quickselect is a useful algorithm for finding the kth smallest element in an array in expected linear time. However, it has a worst-case time complexity of O(n^2) and is not suitable for sorting the entire array.

Q3) One way to find the winner of the game using a circular linked list is to use the Josephus problem algorithm. This algorithm works by repeatedly eliminating every second player until only one player is left.
Here is how the algorithm works:

1. Create a circular linked list with the players as the nodes. The first player should be the head of the list.
2. Initialize a pointer to the head of the list.
3. While there is more than one player left, do the following:
   - Move the pointer to the next player in the list.
   - If the pointer is at the head of the list, move it to the next player.
   - Remove the player pointed to by the pointer from the list.

4. The remaining player is the winner.
   This algorithm runs in linear time, as it only needs to visit each player once.
   Here is some sample code in Python that demonstrates the algorithm:

Algorithm Analaysis

The Josephus problem algorithm has a time complexity of O(n), as it only needs to visit each player once. The space complexity is also O(n), as it requires a circular linked list to store the players.
The algorithm begins by creating a circular linked list with the players as the nodes. It then initializes a pointer to the head of the list.
In the while loop, the algorithm moves the pointer to the next player in the list, and removes the player pointed to by the pointer from the list. If the pointer is at the head of the list, it is moved to the next player. This process is repeated until there is only one player left.
At the end of the loop, the remaining player is the winner.
Overall, the Josephus problem algorithm is a simple and efficient solution for finding the winner of the game using a circular linked list. It runs in linear time and has a relatively low space complexity.

Q4) Ternary search is a divide and conquer algorithm that is used to find the minimum or maximum value in a unimodal function. A unimodal function is a function that has a single

peak or valley, meaning that it increases up to a certain point and then decreases or vice versa.
The time complexity of ternary search is $O(log3n)$, which means that the number of steps required to perform the search grows logarithmically with respect to the size of the input array. In contrast, the time complexity of binary search is $O(log2n)$, which means that it grows at a slower rate with respect to the size of the input array.
The divisor in the time complexity of a search algorithm refers to the number of parts into which the array is divided at each step of the search. In the case of ternary search, the divisor is 3, while in the case of binary search, it is 2.
If we divide the array into n parts at the beginning, the time complexity of the algorithm would be $O(logn)$, which means that the number of steps required to perform the search would grow logarithmically with respect to the size of the input array. This would be slower than both ternary search and binary search, as $logn > log2n$ and $logn > log3n$.
It is important to note that the time complexity of an algorithm is just an estimate of how long it takes to run, and it does not take into account the specific implementation or the hardware on which the algorithm is being run. In practice, the actual performance of an algorithm may be different from the theoretical time complexity.

Q5)

Part 1) Interpolation search is an algorithm used to search for a specific value in a sorted list or array. It works by calculating the midpoint of the search range based on the value being searched for, rather than simply taking the middle element as in a binary search. This can allow the algorithm to search through the list more efficiently, particularly if the list is large and the value being searched for is likely to be closer to the ends of the list rather than the middle.

The best-case scenario for interpolation search is when the value being searched for is the first element in the list. In this case, the algorithm will find the value on the first iteration and the time complexity will be O(1), or constant time.

In the worst-case scenario, when the value being searched for is not in the list, interpolation search will take O(log n) time, where n is the size of the list. This is because the algorithm will halve the size of the search range on each iteration, so the number of iterations required to search the entire list will be logarithmic in the size of the list.

Overall, the time complexity of interpolation search is O(log log n) in the average case, making it more efficient than binary search, which has a time complexity of O(log n) in the average case. However, interpolation search is less predictable in terms of performance and may not always be the best choice for searching large lists, particularly if the list is not uniformly distributed.

Part 2) Interpolation search and binary search are both algorithms used to search for a specific value in a sorted list or array. Both algorithms work by dividing the search range in half on each iteration and comparing the value being searched for to the middle element of the search range.

The main difference between the two algorithms is in the way that they determine the midpoint of the search range. Binary search always takes the middle element of the search range as the midpoint, while interpolation search calculates the midpoint based on the value being searched for and the values at the ends of the search range. This allows interpolation search to potentially search through the list more efficiently, particularly if the list is large and the value being searched for is likely to be closer to the ends of the list rather than the middle.

In terms of time complexity, interpolation search has a slightly better average-case performance than binary search. The average-case time complexity of interpolation search is O(log log n), while the average-case time complexity of binary search is O(log n). This means that, on average, interpolation search will take fewer iterations to search a list of size n than binary search.

However, interpolation search is less predictable in terms of performance and may not always be the best choice for searching large lists, particularly if the list is not uniformly distributed. Binary search, on the other hand, has a more predictable performance and is generally considered to be a more reliable choice for searching large lists.