CSE 321 - Introduction to Algorithm Design

Homework 4 – Report

Ömer Faruk Bitikçioğlu

161044010

1. To find the sequence of steps that leads to the highest possible score in this 2D computer game using a brute force approach, we can check every possible path through the map and select the one with the most points.

The FindMaxPoint2DGame class has a constructor that initializes the map, dimensions of the map, maximum number of points, and path.

The dfs function is a recursive function that uses a depth-first search (DFS) algorithm to find the path with the maximum number of points. It takes in five parameters: the current row index, the current column index, the current number of points, a list of visited coordinates, and a list of the current path.

The function first checks if the current number of points is greater than the maximum number of points. If it is, it updates the maximum number of points and the path. Then it checks if the current coordinate is the last coordinate in the map. If it is, it returns.

If the current coordinate is not the last coordinate, the function calls itself recursively with the coordinates one step down or one step to the right, if those coordinates have not been visited yet. It also updates the number of points and the visited coordinates list for each recursive call.

The find_max_points function is a wrapper function that starts the DFS algorithm from the first coordinate (0, 0) and returns the path with the maximum number of points.

The time complexity of the dfs function in the FindMaxPoint2DGame class is O(2^(n+m)), where n is the row and m is the column of destination point in the map.

This is because the function makes a maximum of two recursive calls at each step, and it continues making recursive calls until it reaches the last coordinate in the map. Therefore, the maximum number of recursive calls is 2^(n+m), which is the worst-case time complexity of the function.

2. The goal of this implementation is to find the median of an unsorted array using a decrease and conquer algorithm. The algorithm works by dividing the array into smaller parts and selecting only one part at a time.

First, the function quickselect_median is defined. It takes in an array as input and checks if the array is empty. If it is, it returns None. If the array has an even number of elements, it returns the average of the kth smallest element and the (k+1)th smallest element, where k is the index of the median. If the array has an odd number of elements, it returns the kth smallest element, where k is the index of the median.

Next, the function quickselect is defined. It takes in an array and an integer k as input. It first checks if the array is empty and returns None if it is. It then checks if all the elements in the array are the same and returns the first element if they are. Otherwise, it selects a pivot element from the array and divides the array into two parts based on the pivot: one part with elements that are smaller than the pivot and another part with elements that are larger. If k is less than the number of elements in the left part, the function returns the kth smallest element from the left part. If k is greater than the number of elements in the left part, the function returns the (k-len(left)-1)th smallest element from the right part. If k is equal to the number of elements in the left part, the pivot is the kth smallest element and the function returns it.

In the worst case scenario, the quickselect algorithm will take O(n^2) time to find the median of an unsorted array. This can occur if the pivot element is consistently selected as the smallest or largest

element in the array. This will result in the algorithm having to repeatedly divide the array into smaller and smaller parts, until the pivot is eventually in the middle of the array.

In order to avoid the worst case scenario, it is important to select a random pivot element from the array. This will help to ensure that the pivot is not consistently the smallest or largest element, and will reduce the risk of the algorithm taking O(n^2) time to find the median.

3. a) The find_winner function uses a circular linked list to implement the game described. The nodes in the circular linked list represent the players in the game.

The find_winner function takes a list of players as input and creates a circular linked list with the players as the nodes. It then initializes a variable current to the head of the list (i.e., the first player) and enters a loop to eliminate players until there is only one player left.

At each iteration of the loop, the find_winner function sets current's next node as current's next's next node. By doing this, it removes the current's next node. Then it proceed with the next node (current's next's next node). This process continues until there is only one player left in the game. Then the function returns the value of the remaining player as the winner of the game.

The time complexity of this algorithm is O(n), where n is the number of players in the game. This is because the algorithm eliminates one player at a time, and the number of players decreases by one at each iteration of the loop. Therefore, the time complexity of the algorithm is linear in the number of players.

3. b) This is a function that finds the winner of a game using a decrease and conquer approach. It takes a list of players as input, and the goal is to find the player who will be the last one standing.

The function first defines a variable called "turn" which is initially set to "First", meaning that it is the first player's turn. It then enters a while loop that will run until there is only one player left in the list.

Inside the loop, the function defines a new list called "players_old" which is a copy of the original list of players. It then uses array slicing to remove every other player from the list, depending on whether it is the first player's turn or the last player's turn.

After this, the function checks if the length of the original list of players was odd or even. If it was odd, the value of "turn" is flipped to the opposite value (either "First" or "Last").

Finally, the function returns the only element remaining in the "players" list, which is the winner. It also prints out the current list of players and the current turn at each iteration of the while loop, to help with debugging and understanding the process.

Worst case complexity is O(log(n)). While loop will iterate until there is only one player left, and in each iteration, the number of players is halved. This means that the while loop will iterate log(n) times, and the slicing operation performed within the loop has a time complexity of O(1). Therefore, the overall time complexity is O(log(n)).

4) At first look, it seems like Ternary Search takes less time than Binary Search.

The following is the recursive formula for counting comparisons in the worst case of Binary Search.

$T(n) = T(n/2) + 2, T(1) = 1$

The following is the recursive formula for counting comparisons in the worst case of Ternary Search.

$T(n) = T(n/3) + 4, T(1) = 1$

In binary search, there are $2\log_2(n) + 1$ comparisons in the worst case. In ternary search, there are $4\log_3(n) + 1$ comparisons in the worst case.

Therefore, the comparison of time complexity for the two algorithms transformed into the comparison of expressions $2\log_3(n)$ and $\log_2(n)$. $2\log_3(n)$ can be written as $(2/\log_2(3)) * \log_2(n)$. Since the value of $2/\log_2(3)$ is larger than 1, the time required for the Ternary Search in the worst-case scenario is larger than Binary Search.

The time complexity of a search algorithm depends on the count of items compared. Dividing the input into two or three parts reduces the number of comparisons. But the number of comparisons increases when the count of divisions increases. So it is more accurate to divide the input into two halves.

5. a) The best-case scenario for interpolation search occurs when the target element is located at the middle of the array. In this case, interpolation search will find the target element in just one comparison. The best-case time complexity of interpolation search is $O(1)$.

5. b) Interpolation search and binary search are both search algorithms that are used to find a target element in a sorted array. However, they differ in the way they work and their time complexity.

Interpolation search works by estimating the position of the target element in the array based on the values of the elements around it. It does this by calculating a value called the interpolation index which is used to determine the next position to search in the array. Interpolation search has a time complexity of $O(\log n)$.

Binary search, on the other hand, works by dividing the array into two parts at each step of the search. It does this by comparing the target element to the element in the middle of the array. If the target element is smaller than the middle element, it searches the left half of the array. If the target element is larger than the middle element, it searches the right half of the array. Binary search has a time complexity of $O(\log_2(n))$.

In general, interpolation search may be faster than binary search in some cases, particularly when the data is well-distributed and the target element is likely to be located near the middle of the array. However, binary search may be faster in other cases, particularly when the data is poorly distributed or the target element is located near the beginning or end of the array.