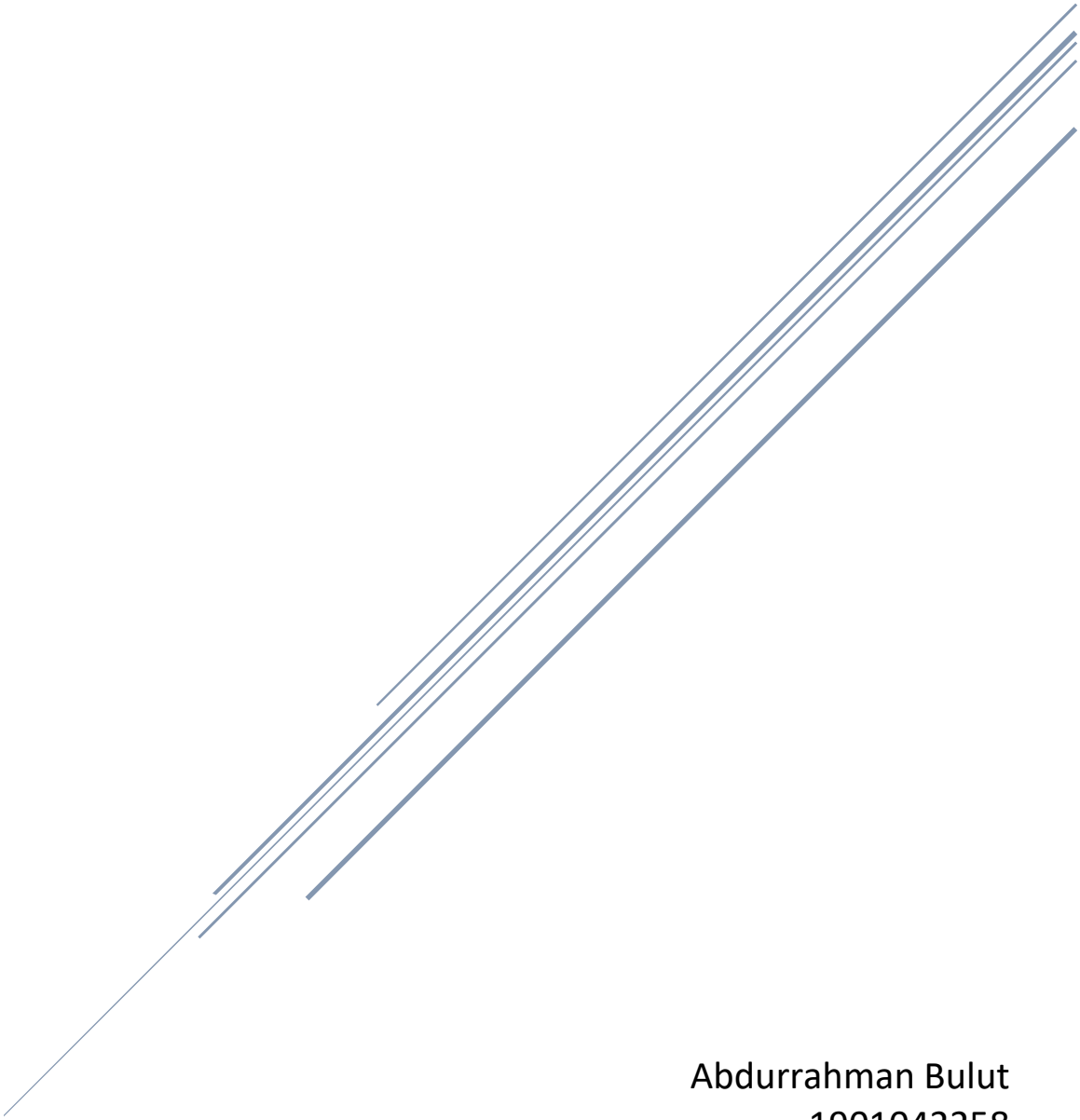
25/12/2022

CSE321

Introduction to Algorithm Design – Homework 4



Abdurrahman Bulut
1901042258

Question 1:

Algorithm: find_maximum_points

Input: board (2D array of point values)

Output: path (list of coordinates of the maximum points route), max_points (maximum points)

1. Set max_points to 0 , path to an empty list, n to number of rows and m to number of columns.(n and m may be taken by inputs.)

2. For i in 0 to n-1:

a. For j in 0 to m-1:

i. Set route to [(0, 0)] and points to the value at coordinate (0, 0) in board.

ii. Set x and y to 0.

iii. While x < n-1 or y < m-1:

1. If x < n-1 and y < m-1:

a. If the value at coordinate (x+1, y) in board is greater than the value at coordinate (x, y+1) in board:

i. Set x to x+1.

ii. Else:

Set y to y+1.

2. Else if x < n-1:

Set x to x+1.

3. Else if y < m-1:

Set y to y+1.

iv. Append coordinate (x, y) to route.

v. Add the value at coordinate (x, y) in board to points.

iv. If points is greater than max_points:

Set max_points to points and path to route.

3. Return path and max_points.

Starting from A_1B_1, this algorithm tries every possible path, selecting at each step the coordinate with the greater point value for each path. The eventual result is the path with the most points. Due to the (nm) potential pathways, this technique has an O(nm) time complexity. The maximum size of the route list is nm, which is why the space complexity is O(nm).

Examples:

Test1:

board = [[25, 30, 25],[45, 15, 11], [1, 88, 15], [9, 4, 23]]

Result:

Route: $A_1B_1 \rightarrow A_2B_1 \rightarrow A_2B_2 \rightarrow A_3B_2 \rightarrow A_3B_3 \rightarrow A_4B_3$

Points: 25 + 45 + 15 + 88 + 15 + 23 = 211

Test2:

board = [[25, 30, 25],[45, 15, 11], [90, 88, 15], [93, 4, 23]]

Result:

Route: $A_1B_1 \rightarrow A_2B_1 \rightarrow A_3B_1 \rightarrow A_4B_1 \rightarrow A_4B_2 \rightarrow A_4B_3$

Points: 25 + 45 + 90 + 93 + 4 + 23 = 280

Question 2:

Algorithm: find_median(arr)

Input: arr: a list of integers

Output: the median of arr

1. If arr has 0 or 1 element, return the only element or None if arr is empty

2. If arr has an odd number of elements, return quick_select(arr, len(arr) / 2)

3. If lst has an even number of elements, return (quick_select(arr, len(arr) / 2 - 1) + quick_select(arr, len(arr) / 2)) / 2

Algorithm: quick_select(arr, k)

Input: arr: a list of integers

k: an integer

Output: the kth smallest element of arr

1. If arr has 0 or 1 element, return the only element or None if arr is empty

2. Select a pivot element randomly from arr

3. Partition arr into three parts: the elements that are smaller than the pivot, the elements that are equal to the pivot, and the elements that are larger than the pivot

4. If k is among the elements that are smaller than the pivot, return quick_select(smallers, k)

5. If k is among the elements that are equal to the pivot, return the pivot

6. If k is among the elements that are larger than the pivot, return quick_select(highers, k - len(smallers) - len(pivots))

This implementation of the median of an unsorted array uses the Quick Select algorithm, which is decrease-and-conquer to find the median and it is a variation of the Quicksort algorithm that is used to find the kth smallest element in an array. The find_median() function first checks for the base case where the array has 0 or 1 element. In these cases, the median is the only element in the array or None if the array is empty.

If the array has more than 1 element, the function checks whether the array has an odd or even number of elements. If the array has an odd number of elements, it calls the quick_select() function to find the element at the middle position (index len(arr) / 2). If the array has an even number of elements, it calls the quick_select() function to find the two middle elements (indices len(arr) / 2 - 1 and len(arr) / 2) and returns their average.

The quick_select() function uses the Quick Select algorithm to find the kth smallest element in the array. It first checks for the base case where the array has 0 or 1 element, in which case it returns the only element or None if the array is empty.

Then, the function selects a pivot element randomly from the array and partitions the array into three parts: the elements that are smaller than the pivot, the elements that are equal to the pivot, and the elements that are larger than the pivot.

If the kth smallest element is among the elements that are smaller than the pivot, the function recursively calls itself on the smaller half. If the kth smallest element is among the elements that are equal to the pivot, it returns the pivot. If the kth smallest element is among the elements that are larger than the pivot, it recursively calls itself on the larger half, adjusting the value of k to account for the elements that have been removed from the array.

The time complexity of this implementation of the median of an unsorted array using the Quick Select algorithm is O(n) in the average case and O(n^2) in the worst case.

The Quick Select algorithm works by partitioning the array around a pivot and recursively selecting the kth smallest element from the appropriate half of the array. At each step, the size of the array is halved, so the algorithm has a time complexity of O(n) in the average case when the pivot is chosen randomly and the array is partitioned evenly into two halves.

However, in the worst case, the pivot is always chosen as the smallest or largest element in the array, and the array is not partitioned evenly. In this case, the time complexity is O(n^2) because the size of the array is not halved at each step.

The space complexity of this implementation is O(n), because we are creating three new lists (smallers, pivots, and highers) at each step.

Tests:

```
print(find_median([1, 2, 3, 4, 5]))

print(find_median([5, 4, 3, 2, 1]))

print(find_median([1, 3, 5, 2, 7, 1]))

print(find_median([1, 2, 3, 4, 5, 6]))

print(find_median([10, 52, 44, 13, 25, 55]))

print(find_median([1]))

print(find_median([]))
```

Outputs:

```
3

3

2.5

3.5

34.5

1

None
```

Question 3-a:

This problem known as the Josephus problem that is a problem in which a group of people are standing in a circle waiting to be executed. Starting at a certain point in the circle, people are eliminated in a set order until only one person remains. The problem is to find the position of the last surviving person. For example, consider a group of 7 people standing in a circle and numbered as follows: 1, 2, 3, 4, 5, 6, 7. If every third person is eliminated, the survivors will be: 3, 6, 4, 5, 7, 3, 5, 7, ..., and the last surviving person will be 5.

Steps: by using a circular linked list

Create a circular linked list with n players, where each player is represented by a node in the list.

Initialize a pointer to the first node (P1).

While there is more than one node in the list, do the following:

a. Remove the node to the left of the pointer (the nearest player on the left).

b. Move the pointer to the node on the right of the removed node.

Return the remaining node as the winner.

This algorithm runs in linear time, O(n), because each player is only eliminated once and the pointer only moves forward in the list.

Code explanation:

The Node class represents a node in the circular linked list. It has two attributes: player, which is a string representing the player, and next, which is a reference to the next node in the list. The CircularLinkedList class represents the circular linked list itself. It has three attributes: head, which is a reference to the first node in the list, size, which is the number of nodes in the list, and find_winner(), which is a method that finds the winner of the game. The __init__() method of the CircularLinkedList class is used to initialize the circular linked list. It takes an integer n as input and creates a circular linked list with n players. Each player is represented by a node in the list. The head attribute is set to the first node in the list and the size attribute is set to n.

The find_winner() method is used to find the winner of the game. It initializes a pointer to the first node in the list (P1). It then enters a loop that runs while there is more than one node in the list. Inside the loop, it removes the node to the left of the pointer (the nearest player on the left). It then moves the pointer to the node on the right of the removed node. When the loop ends, there is only one node left in the list, which is returned as the winner. Finally, it creates an instance of the CircularLinkedList class, passing the value of n as an argument to the __init__() method. It then calls the find_winner() method on the instance and prints the result.

The time complexity of the code is O(n). This is because the find_winner() method runs in linear time, O(n). It only removes each player from the list once and moves the pointer forward in the list once for each player, so the number of operations is directly proportional to the number of players, n. The __init__() method also runs in linear time, O(n). It creates a node for each player and links them together in a circular linked list, which takes a constant amount of time per player. Therefore, the overall time complexity of the code is O(n).

Question 3-b:

Steps: decrease-and-conquer algorithm

This problem is also known as Josephus problem. One way to solve the this problem using a decrease-and-conquer approach is to use a binary search to find the position of the survivor. This algorithm has a time complexity of O(log n), where n is the number of people in the circle.

```
def find_winner(n: int, k: int) -> int:
    if n == 1:
        return 0
    else:
        return (find_winner (n - 1, k) + k) % n
#“k is fixed to number 2”
def find_winner(n: int) -> int:
    if n == 1:
        return 0
    else:
        return (find_winner (n - 1) + 2) % n
```

This function return result – 1. So in the driver function I added 1 to result.

In the pseudocode I provided, k is a parameter that represents the number to be counted off before a person is removed from the circle. For example, if k is 3, then the algorithm will count off 1, 2, 3 and remove the person at the third position. Then it will continue counting off from the fourth position, and so on, until only one person remains. In the assignment , it is asked that every player kills the next player. So, k is fixed to number 2.

For example, if we want to find the survivor in a group of 7 people and k is set to 3, then the survivor will be the person at position 3 (counting from 0). If k is set to 4, then the survivor will be the person at position 1. At each step, the position of the survivor in the smaller group is used to calculate the position of the survivor in the larger group.

To see how this works, let's consider an example with n = 7 and k = 3. We start by calling find_winner (7, 3). This calls find_winner (6, 3), which calls find_winner (5, 3), which calls find_winner (4, 3), and so on. Eventually, we reach the base case of n = 1, which returns 0.

The call to find_winner (4, 3) returns (0 + 3) % 4 = 3, which is the position of the survivor in the group of 4 people. The call to find_winner (5, 3) returns (3 + 3) % 5 = 3, which is the position of the survivor in the group of 5 people. The call to find_winner (6, 3) returns (3 + 3) % 6 = 0, which is the position of the survivor in the group of 6 people. Finally, the call to find_winner (7, 3) returns (0 + 3) % 7 = 3, which is the position of the survivor in the group of 7 people.

This algorithm runs in O(log n) time because the size of the input is halved at each step of the recursion.

Question 4:

The time complexity of a search algorithm is a measure of how the running time of the algorithm scales with the size of the input. In the case of search algorithms, the input size is typically the number of elements in the array being searched. In the case of binary search and ternary search, the time complexity is determined by the number of times the search range is divided in half or into three equal parts in the case of ternary search. The time complexity is expressed as a function of the number of divisions, which is related to the divisor used to divide the search range. First of all, lets look at these algorithms and time complexities of each.

Ternary search works by dividing the given range into three equal parts and evaluating the function at the two extreme points and the midpoint, so the divisor is 2. Based on the values at these three points, the algorithm decides which of the two subranges to search next. The process is then repeated on the chosen subrange until the minimum or maximum is found.

Pseudocode:

```
function ternary_search(f, left, right):
  while left < right:
    mid1 = left + (right - left) // 3
    mid2 = right - (right - left) // 3
    if f(mid1) < f(mid2):
      right = mid2
    else:
      left = mid1
  return left
```

Ternary search has a time complexity of $O(\log_3 n)$, which is slower than binary search $O(\log_2 n)$, but faster than linear search (O(n)). It is useful for functions that are too expensive to evaluate at every point in a range, but can be evaluated quickly at the three points chosen by the algorithm.

On the other hand, Binary search is an algorithm for finding the position of a target value within a sorted array. It works by repeatedly dividing the search range in half, until the target value is found or it is clear that the value is not present in the array, so the divisor is 3.

```
function binary_search(array, target):
  left = 0
  right = len(array) - 1
  while left <= right:
    mid = left + (right - left) // 2
```

```
    if array[mid] == target:
        return mid
    elif array[mid] < target:
        left = mid + 1
    else:
        right = mid - 1
  return -1
```

Binary search has a time complexity of $O(\log_2 n)$, which is faster than linear search O(n). It requires that the input array be sorted, but is efficient for large arrays and can be used to find a specific value or to determine the position at which a value should be inserted to maintain sorted order.

In general, the time complexity of a search algorithm is affected by the divisor used to divide the array at each step. The larger the divisor, the fewer the number of steps required to search the entire array, and the lower the time complexity of the algorithm. For example, if we divide the array into n parts at the beginning, the time complexity of the algorithm would be O(logn), which is the same as the time complexity of a linear search and significantly lower than both the time complexity of ternary search and binary search . This is because the algorithm would only need to make a single division to determine the position of the target value. It's important that while ternary search may have a lower time complexity than binary search, it is not always the best choice for searching a sorted array. Binary search is generally more efficient and easier to implement, and it is also more widely used in practice.

Question 5:

a- Interpolation search is an algorithm that is used to search for an element in a sorted array. It works by using the value of the element being searched for to estimate the position where it is likely to be found in the array, and then performing a binary search at that position. The best-case scenario for interpolation search is when the element being searched for is found at the estimated position in the first iteration. It happens when the target value is found at the midpoint of the search range on the first iteration. In other words, It happens when the target value is exactly at the midpoint of the search range or when it is the only value in the array. In this case, the time complexity of the algorithm is O(1), because the element is found in a single step. However, in the worst-case scenario, interpolation search has a time complexity of O(n), which is the same as binary search. This occurs when the target value is not present in the array or when it is at one of the extreme ends of the search range. If the data is sorted and uniformly distributed, the time complexity of interpolation search is O(loglog n) in the average case where n is the number of elements in the array. It’s not using any extra memory, so the space complexity is constant O(1).

Overall, interpolation search is a useful algorithm for searching ordered lists, particularly when the list is large and the key being searched for is likely to be located near the middle of the list.

b- Both interpolation search and binary search are algorithms for searching for a given key within an ordered list. The main difference between the two algorithms lies in the way they determine the position of the key being searched for within the list. The difference between interpolation search and binary search is that interpolation search works by calculating the midpoint of the sorted array based on the key values, while binary search works by dividing the array into two parts and searching the appropriate subarray. The time complexity of interpolation search is O(loglog n), while the time complexity of binary search is O(log n). One key advantage of interpolation search over binary search is that it is more efficient when the key being searched for is likely to be located near the middle of the list. In this case, interpolation search can find the key in fewer comparisons than binary search, due to its more precise estimate of the key's position.

As example:
Suppose we have the following ordered list of integers: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

We want to search for the key 14 using both interpolation search and binary search.

Interpolation search:
We start by using the interpolation function to estimate the position of the key within the list. Suppose the function returns an estimate of position 6. We compare the element at position 6 (14) to the key being searched for (14). Since they are equal, the search terminates and we have found the key.

Binary search:
We start by comparing the element at the midpoint of the list (10) to the key being searched for (14). Since 14 is greater than 10, we know the key is likely to be in the second half of the list. We compare the element at the midpoint of the second half of the list (14) to the key being searched for (14). Since they are equal, the search terminates and we have found the key. In this example, interpolation search was able to find the key in a single comparison, while binary search required two comparisons. This illustrates how interpolation search can be more efficient than binary search when the key is likely to be located near the middle of the list.