CSE 321 HW4

Burak Kocausta

1901042605

Note: Explanation of the test menu is end of the report. After built in results executed, menu will show up.

Q1) Brute Force Route Game Algorithm

Explanation:

The algorithm that I designed is encapsulated in RouteGame class. This class has initializer, print and finding route functions. Map is held at 2d grid, if a 2d grid is passed it is initialized according to the user input, if not random map is generated.

In the algorithm part, brute force is requested, so my approach is determining all routes, then decide which has greatest point. Because of being brute force, algorithm doesn't have any smart decisions during recursive calls. I use sub procedure as main algorithm, this algorithm fills the sum_route hash map. In the hash map sums are keys, route lists are values. Base case of the algorithm is reaching the lower right element. There are 2 recursive calls, which are going right and going down. These 2 operations only condition is only border checking. Therefore, if borders are not reached call will definitely happen. Before the call is made, each condition has process step. In these steps, sum and route list is propagated. Because the base condition is reached for every route, in every reach these two values is saved to sum_route hash map. After the algorithm finishes, outer procedure prints the proper result.

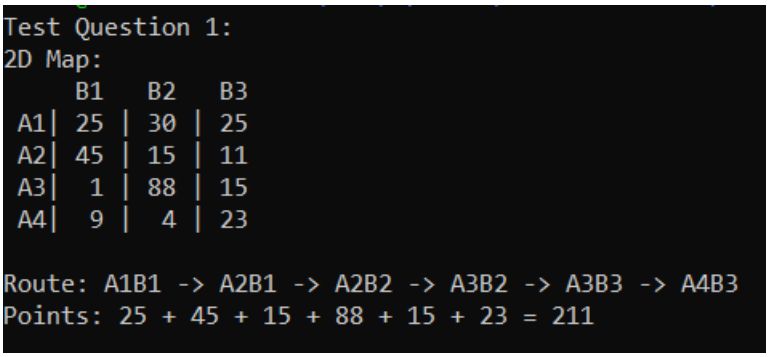
Analysis:

The algorithm that I designed with using brute force approach checks all the possible routes. Every recursive call is made at one point, all points in the map have only 2 options. Going right or down. There are (number of rows * number of columns) points. If we call this n*m. There are 2^(n*m) decisions. This algorithm performs all decisions without condition. As a result, it is performed 2^(n*m) times.

$T(n) = \Theta(2^{n*m})$

Test for question 1:

```
print("Test Question 1: ")
routeGame = RouteGame(4, 3, [[25, 30, 25], [45, 15, 11], [1, 88, 15], [9, 4, 23]])
print("2D Map: ")
routeGame.print_map()
routeGame.find_max_point_route()
print()
```



Q2) Decrease and Conquer Algorithm to Find Median of Unsorted Array

Explanation:

For finding median of unsorted array, initial thought could be first sorting the array, then finding the middle element, but this makes unnecessary operations to determine median. More direct approach can be made using decrease and conquer. I thought quick select algorithm to achieve this, because it is also a decrease and conquer algorithm. It selects the element using partition. It returns the element which is the nth element after ordering. To find this, sorting all the array is unnecessary. Partition algorithm returns the pivot index which is also position of that element after sorting. Quick Select algorithm uses that to find wanted number. So, I thought if I use quick select to find middle element, it means median will be found.

Algorithm is decrease and conquer algorithm, because it decreases the problem, till target is found. There is no merging in this algorithm, if merge operation happened, then it would be divide and conquer algorithm. Algorithm looks, left or right sub array till target is found.

To implement this firstly I implemented the Lomuto Partition algorithm. It is linear algorithm. Arranges the array, then returns the pivot index. In my algorithm I selected the rightmost element as pivot. Then, the correct position of pivot is returned. In the find_median function, there two base cases, one of them is if array consumed before finding the median, or median is found condition. Recursive calls are made according to the pivot index's position, if it is middle of the array, then it can be said that it is median. Till it is found, problem is reduced with using left sub array or right sub array.

Analysis:

Algorithm uses lomuto partition, and specific version of quick select. In each step lomuto partition finds the pivot's position, and it is linear algorithm. So, execution occurs at least n times. After that, for each case that median is not found, half of the array is eliminated. Then problem size for partition algorithm becomes n/2 for each step. So in the worst case, it becomes $n + n/2 + n/4 + \dots 1$, and this is equal to $2n-1$. There are $2n-1$ execution for n-size problems in the worst case. We ignore the constants, therefore it is linear algorithm. Best case is $O(n)$, worst case is $O(2n-1)=O(n)$.

T(n) = O(2n-1)

Test for Question 2:

```
# test find_median function
print("Test Question 2: ")
print("Unsorted Array: ", end="")
arr = [3, 1, 7, 5, 2, 4, 6, 8, 12]
print(arr)
print("Median: ", end="")
print(find_median(arr, 0, len(arr)-1))
print()
```

```
Test Question 2:
Unsorted Array: [3, 1, 7, 5, 2, 4, 6, 8, 12]
Median: 5
```

Q3-a) Josephus Game Circular Array Solution

Explanation:

Firstly, I implemented the Circular Linked List data structure, and named it PeopleCLL. This class have, insert, and print functions. Holds person as inner class. Then I used this class to implement the algorithm. The algorithm directly performs the game as we understand. It means that, algorithm always started from P1, it looks the next of that person, then deletes the person from circular linked list. After that current person becomes next person. This process is performed till next person becomes that person. Then winner is the last person.

In the implementation of the algorithm, I initially insert the whole people to that circular linked list. After that, algorithm starts from the first person which is P1. In while loop condition is next person is the current person, Inside the loop elimination result is printed, and next person becomes next persons next. So, it means that first next person is eliminated. With this way, circle becomes smaller, and smaller, in the end there is only 1 person left. And that person is winner.

Analysis:

Algorithms first part is inserting all the elements to the linked list, this takes Θ(n) time. After that game is started. During the game, linked list has n elements initially, but in each step one person is eliminated. This means, when there is only one person left, whole circular list is iterated one time. This part is also Θ(n). In the end algorithm is linear.

T(n) = Θ(n)

Test for question 3a:

```
# test question 3a
print("Test Question 3a: ")
josephus_problem(7)
print()
```

```
Test Question 3a:
people: P1->P2->P3->P4->P5->P6->P7

P1 eliminates P2
P3 eliminates P4
P5 eliminates P6
P7 eliminates P1
P3 eliminates P5
P7 eliminates P3
P7 is the winner
```

Q3-b) Josephus Game Decrease and Conquer Solution

Explanation:

While designing the algorithm, I initially thought the base situation in the problem. Firstly, it can be seen that, even numbers never win the game, and the result is determined according to the number of elements for the main problem. If somehow, I carry the 2 odd number from subproblems to the main problem, result can be found.

Generalization Explanation:

Pa, Pc are odd numbers. Pb, Pd are even numbers.

First Case: {Pa} - there is only one person, so winner is Pa.

Second Case: {Pa, Pb, Pc} – there are 3 people winner is Pc.

Third Case: {Pa, Pb, Pc, Pd} – there are 4 people, winner is Pa.

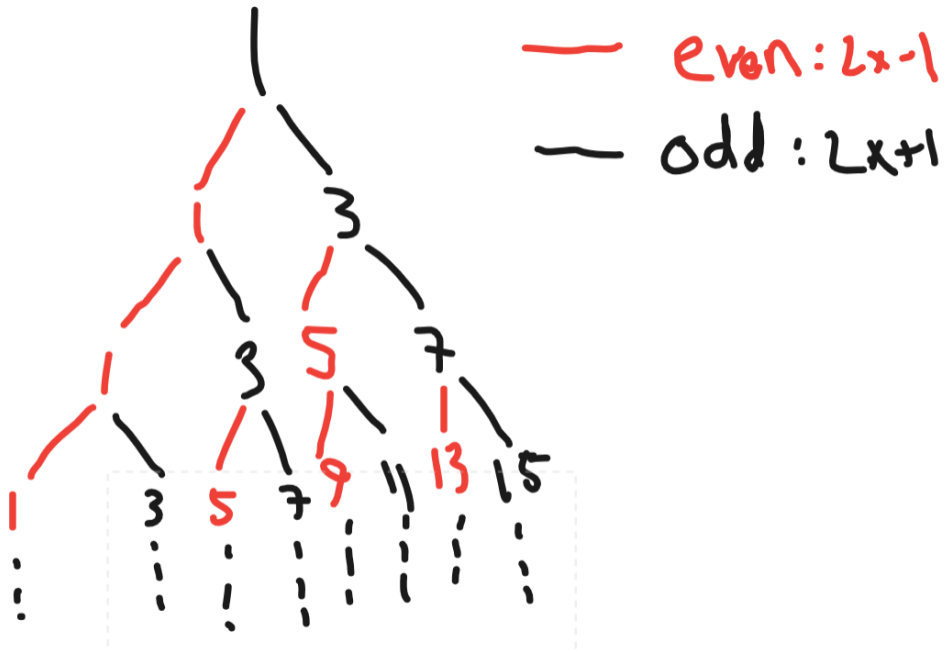
It can be seen there is only 2 possibilities for determining the winner. Except from the base case which is n = 1. Also, there is another case which is there are only 2 people, in this case first case occurs, so I didn't include it in the generalization.

When the number of people is odd (2*n+1)th person is the winner.

When the number of people is even (2*n-1)th person is the winner.

Therefore, before returning the result in the algorithm, situation must be one of these cases. To achieve that, I decided to decrease the problem into two subproblems. In each problem, result is carried to the general case. In every call dividing the size by two, reduces the number of people, with this way base case is reached. After n = 1 case is reached, algorithm evaluates the reduced result, and reaches the generalized cases. In every returning of the recursion, subproblems result will be returned. This result is propagated in each step according to each subproblems size (even or odd), at the end result will be found with original size.

Possible Result Tree of the Algorithm:



For example, if there are 6 people, steps are:

f(6)

6 -> 6 is even, result is 2*f(3)-1

3 -> 3 is odd, result is 2*f(1) + 1

1 -> 1

then evaluation starts:

3-> 2*1 - 1 -> 3

6-> 2*3 - 1 -> 5

P5 is winner.

Analysis:

For each recursive call problem size is divided by 2, so 2^T(n) = n, T(n) = logn. Because of being eliminated half of the problem in each step, algorithm becomes logarithmic.

T(n) = Θ(logn)

Test for Question 3:

```
# test question 3b
print("Test Question 3b: ")
print("input: n = 7")
print(f'{"P"}{josephus_problem_dc(7)}', "is the winner")
print()
```

```
Test Question 3b:
input: n = 7
P7 is the winner
```

Q4) Binary – Ternary Search, Search with Dividing N-Parts

Time complexity improvement is not correct if we consider the number of comparisons. If we implement ternary search, this means there must be 2 middle points, and 3 sections. If the element that searched is one of that 2 middle point, target is found this makes 2 comparisons. If target is not found, selection of where to go for that 3 section is the next. It can be left of left middle, between middles, or rightmost section. This requires 2 comparisons, because last condition is on the else branch. Therefore, there must be at most 4 comparisons to decide where to go during recursive calls. For the **worst case**, in each recursive call **4 comparisons** will be made.

In binary search, there is only 1 middle point, and 2 sections. So, in every call middle point is checked, then left or right direction is decided with comparing target and middle. So, in the **worst case** there are **2 comparisons** for each call.

According to these comparison numbers,

It is said that in the question, ternary search is $O(\log_3 n)$, binary search is $O(\log_2 n)$. If we consider worst case, and previously calculated number of comparisons, we can say that total number of comparisons for ternary search it can be said $O(4\log_3 n)$, and for binary search it is $O(2\log_2 n)$. Because there are n number of comparison for every call, and it is done in total call time, so we can multiply them, and ignore the other constants in big O notation.

As a result, $4\log_3 n > 2\log_2 n$. This means that ternary search makes more comparison. Therefore, **binary search is faster.**

For dividing n parts, it can be seen that increasing the sub-section number comes with a cost when binary and ternary is compared. Reason of that is number of comparisons as indicated. It is also true for this part. If we divide n elements to n part, this means that there are n-1 middles. In each comparison, only 1 element is eliminated. It goes to the result one by one. Therefore n-1 comparison will be done in worst case, and this means that it becomes linear since O(n-1) = **O(n).**

Q5-a) Interpolation Search Best Case

Interpolation search is a search algorithm which is applied on sorted array. It is based on estimation. It calculates a position on every call, this calculation is the estimation part, then checks that position, if target is not found, it eliminates some of the subsections according to the comparison result.

For this algorithm best case is of course **O(1).** Result can be found directly, as similar to other search algorithms like binary search. For the best scenario, algorithm makes more accurate estimations if given collection is uniformly distributed. Average case is **log(log(n))** for that algorithm. If all estimations are incorrect, worst case will occur and it is linear O(n).

Q5-b) Comparison of Interpolation Search and Binary Search

Both algorithms perform on the sorted arrays. But interpolation search requires more precondition to perform efficiently than binary search. As explained above, interpolation performs best when elements are uniformly distributed. Binary Search does not have a property like that. Binary search provides more guaranteed time complexity. Because, interpolation search is based on estimation, there is a calculation part in the interpolation algorithm to estimate position of the element. Because of that calculation it requires more precondition to perform better. It is likely better when elements are uniformly distributed, and time complexity of this is O(log(logn)). For the worst case it behaves like linear search, and it is O(n). For the binary search average case, and worst case are O(logn). Both algorithms are O(1) in the best case.

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

As a result, interpolation search should be preferred when elements are more likely to be uniformly distributed to achieve better average performance. Binary search is better when, to get guaranteed performance, It is more general than interpolation search.

Test Menu for Questions 1-2-3:

(continues on the other page)

Question 1: Takes row, and column generates elements randomly, prints the result.

```
Test Menu:
1. Test Question 1
2. Test Question 2
3. Test Question 3a
4. Test Question 3b
5. Exit

input(1-5): 1

Question 1:

Enter number of rows: 4
Enter number of columns: 5

2D Map:
     B1   B2   B3   B4   B5
 A1| 18 | 60 | 86 | 56 | 44
 A2| 69 | 48 | 49 | 92 | 18
 A3| 45 | 85 | 41 | 41 | 57
 A4| 67 | 56 | 47 | 34 | 28

Route: A1B1 -> A1B2 -> A1B3 -> A1B4 -> A2B4 -> A3B4 -> A3B5 -> A4B5
Points: 18 + 60 + 86 + 56 + 92 + 41 + 57 + 28 = 438
```

Question 2: Gets the size of the array, then generates elements randomly, and prints the result.

```
Test Menu:
1. Test Question 1
2. Test Question 2
3. Test Question 3a
4. Test Question 3b
5. Exit

input(1-5): 2

Question 2:

Enter size of array: 9

Generated Unsorted Array: [21, 32, 7, 14, 38, 52, 36, 69, 84]
Median: 36
```

Question 3a: Gets the total number of the people, and prints the result.

```
Test Menu:
1. Test Question 1
2. Test Question 2
3. Test Question 3a
4. Test Question 3b
5. Exit

input(1-5): 3

Question 3a:

Enter number of people: 6

people: P1->P2->P3->P4->P5->P6

P1 eliminates P2
P3 eliminates P4
P5 eliminates P6
P1 eliminates P3
P5 eliminates P1
P5 is the winner
```

Question 3b: Gets the total number of the people and prints the result.

```
Test Menu:
1. Test Question 1
2. Test Question 2
3. Test Question 3a
4. Test Question 3b
5. Exit

input(1-5): 4

Question 3b:

Enter number of people: 6

P5 is the winner
```

Terminate menu with 5.