

QUESTION – 1

In this question, we must implement a brute force algorithm. These algorithms are simple to implement, I think. So, in this question, I go to the end in every possible way recursively and return the total points. If returning value (total points) of going 'right side' function call is greater than downside function call, I got the first one and add the path and total points. If I simplify it, I go to final cell in every possible way and select the one that has more points. I run example of PDF firstly, then I got the input.

I must inform you about a point. `findPath` calculates the total points correct. But, in the path there were only a few unnecessary locations. So, I cleaned them in the function called *initiliazePath*. I use a simple algorithm in it. Unrelated cells deleted and the rest path is correct path. My all results in this question are correct. You can test part-1. It will generate random table and showed the maximum point path. Regards.

Analyse:

```
def findPath(board, rown, coln, total, string, points):
    if(rown>=len(board) or coln>=len(board[0])):
        return total

    move1 = "A" + str(rown+1+1) + "B" + str(coln+1)
    move2 = "A" + str(rown+1) + "B" + str(coln+1+1)

    total1 = -1
    total2 = -1

    if(rown+1<len(board)):
        total1 = findPath(board, rown+1, coln, total+board[rown+1][coln], string, points)
    if(coln+1<len(board[0])):
        total2 = findPath(board, rown, coln+1, total+board[rown][coln+1], string, points)

    if ((rown+1<len(board)) and (coln+1<len(board[0]))):
        if(total1>total2):
            if move1 not in string:
                string.insert(0, move1)
                points.insert(0, board[rown+1][coln])
            return total1
        else:
            if move2 not in string:
                string.insert(0, move2)
                points.insert(0, board[rown][coln+1])
            return total2
    else:
        return total+board[len(board)-1][len(board[0])-1]
```

Handwritten notes:

- Next to the first recursive call: $T(n-1)$
- Next to the second recursive call: $T(n-1)$
- Next to the first branch of the conditional: $O(n)$
- Next to the second branch of the conditional: $O(n)$

We have a recurrence relation in WORST case:

$$T(n) = 2T(n-1) + n$$

$$T(n) = 2(2T(n-2) + (n-1)) + n$$

$$T(n) = 4T(n-2) + 3n - 2$$

$$T(n) = 4(2T(n-3) + (n-2)) + 3n - 2$$

$$T(n) = 8T(n-3) + 7n - 6$$

$$T(n) = 2^k * T(n-k) + (k+1)n - 2*(2^k - 1)$$

We can set $k = n-1$, to reach the base case.

In this case, $T(n) = n^2 - n$

So, the algorithm is *quadratic*. $O(n^2)$ in the worst case

QUESTION – 2

While studying the textbook of the course ("Anany Levitin - Introduction to the Design and Analysis of Algorithms") I have seen this problem. Page 158 (Computing a Median and the Selection Problem).

To solve the problem, as book suggests, we must use one of partition algorithms. The problem looks like sorting problems. But there is a difference. The sorting problems are divide-and-conquer algorithms. I mean, you must divide problem into pieces, and you must solve all of them separately to combine later. But in this case, we do not have to sort the list. Of course, we can, but it decreases the performance. We need only median. So, I use Lamuto Partition Algorithm and QuickSelect Algorithm in this question as textbook suggests. The main function is QuickSelect. I implemented it to find any index, but in the driver function I send the median index. QuickSelect makes a partition using Lamuto Partition. It finds the pivot. Then compares the pivot with the target. If there is no match, it divides the problem and continue. It leaves the other part of data. So, it is decrease-and-conquer algorithm. (Page 159, 160, 161)

Analyse Worst Case:

Partition algorithm can make a partition that we really do not want. For example, the pivot can always in the first index.

A partition algorithm that gets n-size input, makes n-1 comparisons. So, if the pivot is always in the first index as I told above, and moreover, if we cannot find it until the end, we must call the partition algorithm as many as data size. In summary, if we have n element array,

Worst Case will be $= (n-1) + (n-2) + (n-3) + \dots + 1 = O(N^2)$, *Quadratic*.

(n-1 is first partition, n-2 is second partition ... etc.)

It is *quadratic*.

QUESTION – 3

a.)

Using circular linked list, the question seems so simple. We will start from the head of the linked list. We must traverse the whole list. So, I create an iterator-like variable. I traverse the list. In every step, the next node of the iterator is deleted (this process is constant time). Also, in every step, I check that how many elements I have. If there is only 1 element, it is the survivor. Game over.

Analyse:

We must only traverse the whole list with an iterator. So, complexity is linear.

Assume that we have 5 elements: {1, 2, 3, 4, 5}

Step-1 -> Iterator points the number 1, Number 2 is deleted, Forward the iterator

Step-2 -> Iterator points the number 3, Number 4 is deleted, Forward the iterator

Step-3 -> Iterator points the number 5, Number 1 is deleted, Forward the iterator

Step-4 -> Iterator points the number 3, Number 5 is deleted. There is only 1 element. The winner is 3!

The process will be the same in the different datasets. If we have 5 elements, we need 4 iterations. The time complexity is linear: $O(n)$

b.)

In the course textbook ("Anany Levitin - Introduction to the Design and Analysis of Algorithms"), in the page 154, you will see Josephus Problem. I have seen it while studying the course. This problem is exactly the same as with this Josephus Problem. It really helped me. Let me explain the algorithm:

- We must consider the cases of even and odd data sizes separately. (Page 155)
- If n is even, in the first iteration, we got the exact same problem, but half its initial size. The positions of people in the new case will change. If a person's number is 4 in the beginning of this iteration, it will be 7. We need to multiply its position by 2 and then add 1. (Page 155)
- The case of an odd n : The first iteration kills people in even positions. If we add to this the elimination of the person in position 1 right after that, we are left with an instance of size k . In this case, numbers of people will change too. If you want to find the new number of a person, you must multiply it by 2, then subtract 1. (Page 155)

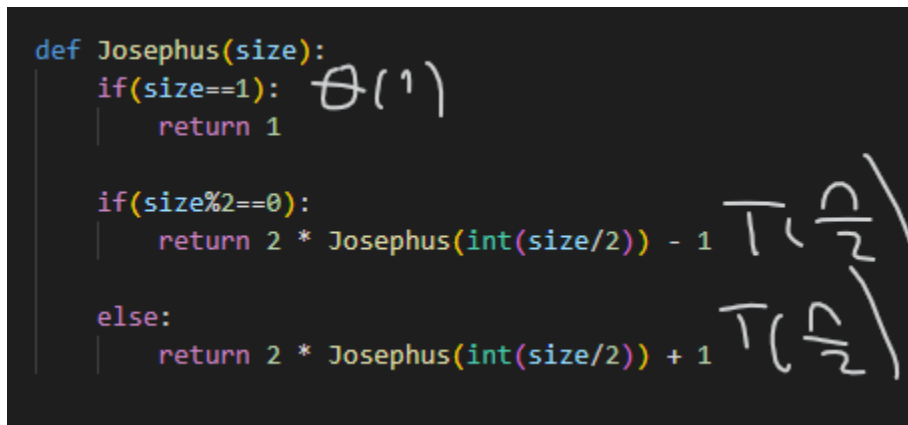
I use the only append and remove functions of Linked List. I modify the remove function. It gets a node and deleted the next node of it. It really helped me. Also, I locate the Josephus problem inside Linked List implementation. It was helpful too.

My code implies the partial function below:

$$\text{Josephus}(2k) = 2J(k) - 1$$

$$\text{Josephus}(2k+1) = 2J(k) + 1$$

Analyse Worst Case:



```
def Josephus(size):  
    if(size==1):  
        return 1  
  
    if(size%2==0):  
        return 2 * Josephus(int(size/2)) - 1  
  
    else:  
        return 2 * Josephus(int(size/2)) + 1
```

It is really simple algorithm. We always divide the problem to its half. You can easily see that the complexity is logarithmic.

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

If we solve the recurrence relation above, we will see that the complexity is $O(\log n)$. And there is no stopping condition. It will always run $\log n$ time. So, the worst case is $\theta(\log n)$

QUESTION – 4

Binary search divides the problem 2 part and cuts one part. So, it goes by dividing the problem 2 parts. At the end, complexity will be $O(\log_2 n)$. Ternary Search divides the problem 3 parts and cuts 2 parts of them. At the end complexity will be $O(\log_3 n)$.

Mathematically, it looks that $\log_2 n$ is greater than $\log_3 n$. However, in every step, binary search makes 1 comparison, but ternary search makes 2 comparisons. And, mathematically, $2\log_3(x) > \log_2(x)$. So, binary search is more efficient practically.

If we think theoretically, we may think that dividing the problem to more parts will make the algorithm better. However, practically for larger N values, things get worse. if we divide the data that has n elements to n parts, you must easily see that we must make more comparisons as I told above. So, dividing the data that has n elements to n parts, makes the algorithm linear. $O(n)$. We are going to MUST compare every element with target in worst case.

QUESTION – 5

b.)

I want to answer this 'B' question first. Binary Search always compares the target with middle element and divide the array 2 parts. So, its time complexity is always logarithmic: $O(\log n)$. On the other hand, Interpolation Search, starts searching from the parts where the searched item is likely to be found. This is like searching for a word starting with the letter Y in the dictionary and opening the dictionary from the ends. If the array is uniformly distributed array, Interpolation Search is much more efficient than Binary Search. In this case its complexity is $O(\log \log n)$.

However, if the array is not uniformly distributed performance of Interpolation Search will decrease. So, it is clear that Binary Search is more reliable than Interpolation Search although Interpolation Search looks better.

a.)

Now the question 5.A can be answered. As I wrote above, Interpolation Search, starts searching from the parts where the searched item is likely to be found. It uses the formula below to find the position:

```
pos = lo + ((hi - lo) // (arr[hi] - arr[lo]) * (x - arr[lo]))
```

lo is leftmost index, hi is rightmost index, x is target element.

Assume that we have an array: {1, 2, 3, 4, 5, 6, 7} and we are looking for the middle element 4.

In this case, in the first iteration of Interpolation Search will calculate the likely position as:

$$\text{pos} = 0 + ((6-0) // (7-1) * (4 - 1)) = 3$$

$$\text{pos} = 3$$

As you see, if the target element is in the middle, Interpolation Search finds it in constant time. $O(1)$