CSE321

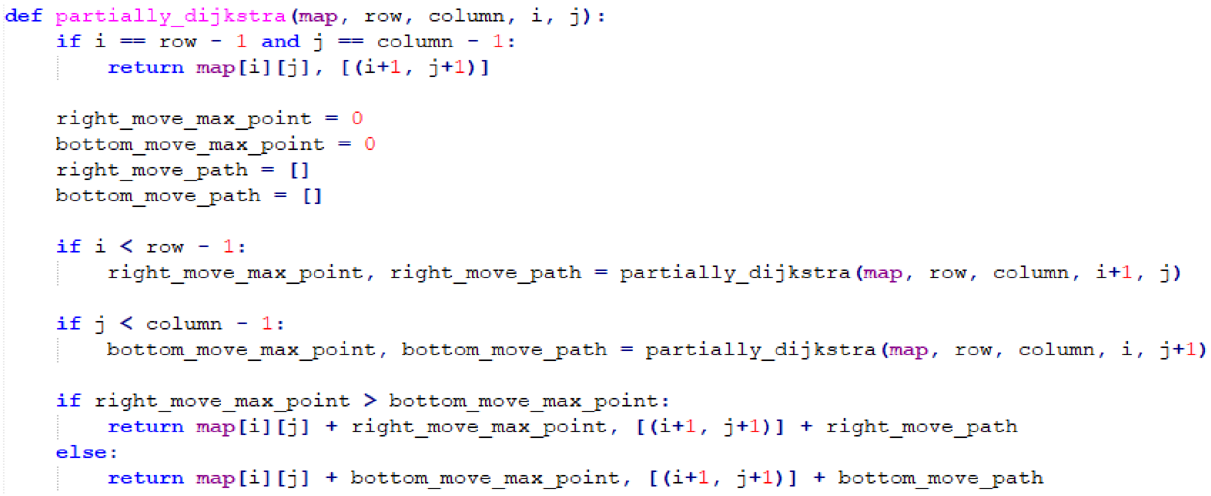
Homework #4

Abdullah Çelik - 171044002

1)

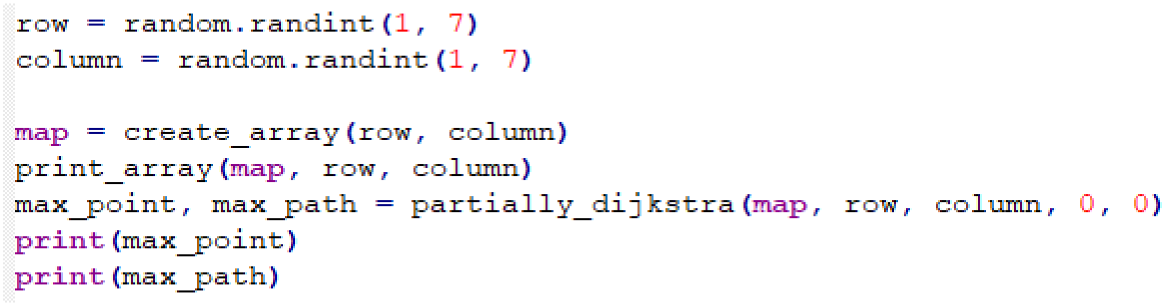
Design: I will make use power of recursive functions

- Am I at the target point now?
    - So return the score and position of the cell I am in
- If you can go the right way, go and calculate the total cost (make recursion call)
- If you can go the bottom way, go and calculate the total cost (make recursion call)
- Which way I get more points when I go(right or bottom), return the total score and full path



```
def partially_dijkstra(map, row, column, i, j):
    if i == row - 1 and j == column - 1:
        return map[i][j], [(i+1, j+1)]

    right_move_max_point = 0
    bottom_move_max_point = 0
    right_move_path = []
    bottom_move_path = []

    if i < row - 1:
        right_move_max_point, right_move_path = partially_dijkstra(map, row, column, i+1, j)

    if j < column - 1:
        bottom_move_max_point, bottom_move_path = partially_dijkstra(map, row, column, i, j+1)

    if right_move_max_point > bottom_move_max_point:
        return map[i][j] + right_move_max_point, [(i+1, j+1)] + right_move_path
    else:
        return map[i][j] + bottom_move_max_point, [(i+1, j+1)] + bottom_move_path
```

Driver:



```
row = random.randint(1, 7)
column = random.randint(1, 7)

map = create_array(row, column)
print_array(map, row, column)
max_point, max_path = partially_dijkstra(map, row, column, 0, 0)
print(max_point)
print(max_path)
```

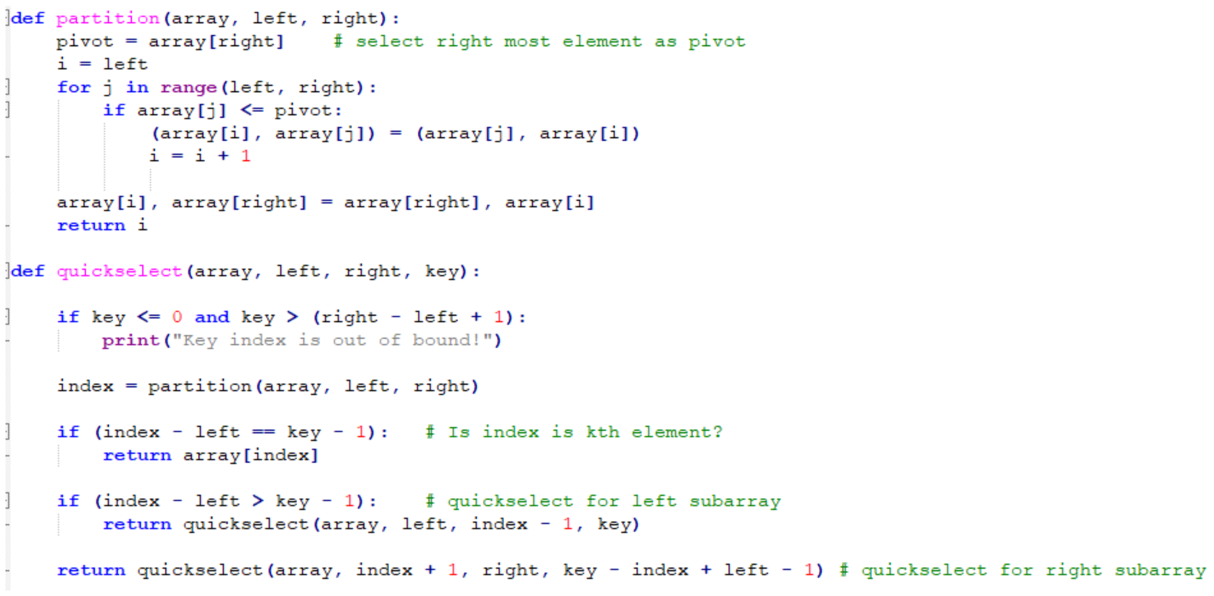
Here the algorithm generates all possible paths between the start and end point. And the most recent recursion call returns with base condition. The recursion call that calls the most recent recursion returns score and path according to the coming from the right way and bottom way. Thus, after continuing, the first recursion call selects the max score and path returns it. All possible paths are tried in best case and worst case. All possible subset is 2^{n*m}. Then time complexity Tbest = Tworst = Tavg = $\theta(2^{n+m})$.

2) The median is the medium element in an ordered array. In our problem, since our array is not ordered, I cannot directly access the middle element in the array. Instead, a solution using the decrease and conquer approach where I can find the (kth) element is required. The first thing that comes to my mind here is the quickselect algorithm. This algorithm allows us to find the kth element in an unordered array. The difference from quicksort is that after partitioning, quicksort works for both subarrays. This logic is called divide and conquer. Quickselect works for subarray containing kth element after partitioning. This is the decrease and conquer approach. The working steps of the algorithm are as follows:

- Select a pivot and partition as in the quick sort algorithm (elements smaller than pivot places in left subarray, greater than pivot places in right subarray).
- If the pivot element is equal to the kth element, the searched element is found. If not:
    - If pivot element index is greater than the kth element, recursion call for the left subarray.
    - If pivot element index is smaller than the kth element, recursion call for the right subarray.

Note: As you can see, the algorithm did not work for part of the subarray after dividing the array. That's the decrease approach. It works to split again for the part containing the kth element. This is the conquer approach.

Algorithm:



```python
def partition(array, left, right):
    pivot = array[right]    # select right most element as pivot
    i = left
    for j in range(left, right):
        if array[j] <= pivot:
            (array[i], array[j]) = (array[j], array[i])
            i = i + 1

    array[i], array[right] = array[right], array[i]
    return i

def quickselect(array, left, right, key):

    if key <= 0 and key > (right - left + 1):
        print("Key index is out of bound!")

    index = partition(array, left, right)

    if (index - left == key - 1):   # Is index is kth element?
        return array[index]

    if (index - left > key - 1):    # quickselect for left subarray
        return quickselect(array, left, index - 1, key)

    return quickselect(array, index + 1, right, key - index + left - 1) # quickselect for right subarray
```

In worst case scenario is that we select the largest or smallest element in each iteration and sort for each element. Sorting n times for n elements. Tworst = $\theta(n^2)$.

In best case scenarion is when the pivot is the sought element after partitioning for the selected pivot. In this case, partitioning is done only once. Tbest = $\theta(n)$.

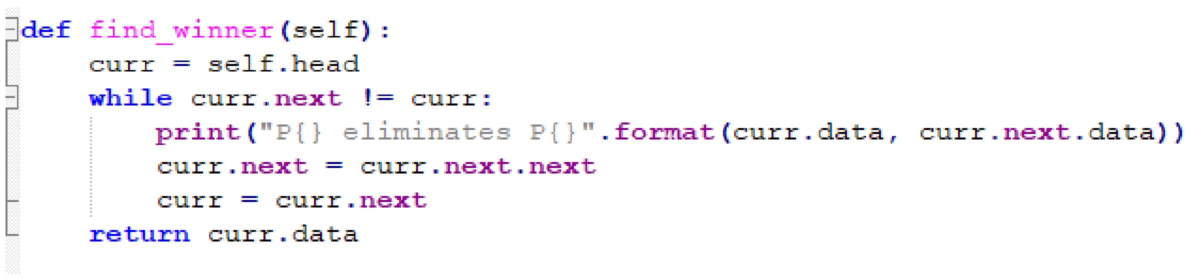
By choosing a random pivot, we can greatly reduce our worst-case chance. Thus, it can work in the average case. Tavg = O(n).

3)

(a) After creating the Circular Linked List structure, the algorithm performs the following operations.

- Assign the head node of the circular linked list to the variable named current node
- Assign the current’s next node to current’s next node’s next node, unless the current node is equal to the node referenced by the next reference (meaning that there is only one node in the circular linked list).

Thus, each node will delete the next node (eliminated from the game). When only one node remains, that is, when the next node of the current node becomes current, the winner will be found. Since the algorithm removes one node from the list, iteration number is one less than the size of the circular linked list. It is constant time to remove the node from the list. For this reason, Tbest = Tworst = Tavg = θ(n)



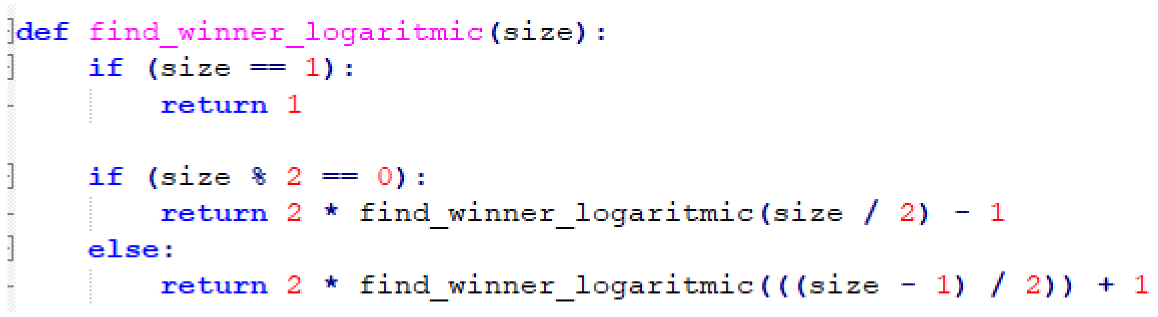
```
def find_winner(self):
    curr = self.head
    while curr.next != curr:
        print("P{} eliminates P{}".format(curr.data, curr.next.data))
        curr.next = curr.next.next
        curr = curr.next
    return curr.data
```

(b) I couldn’t implement an algorithm for this part. As a result of my research, I couldn't find a decrease and conquer algorithm that works in logarithmic time and finds the result by printing that one player eliminates the other player as in option a. But I was able to find an algorithm like this. This algorithm uses an approach that divides the size into subproblems depending on whether the size of the problem is odd or even (decrease approach). And after it decreases, it reaches the result by making a recursion call (conquer). Mathematics is used in the solution here. For example, if your size is even, Player 1 will eliminate Player 2 and Player 3 will eliminate Player 4. As a result, all even-numbered players will be eliminated. In other words, n/2 players will be eliminated and it will be the turn of player 1 again and the remaining numbers will be 1, 3, 5, … n/2. To express this as a reccurence relation

T(n) = T(n/2) – 1   for n is even
T(n) = T((n-1) / 2) + 1   for n is odd

If n is 1, player 1 will win. Base condition T(1) = 1

Algorithm:



```
def find_winner_logaritmic(size):
    if (size == 1):
        return 1

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

Algorithm divides the problem into two parts each time and makes a recursion call.

Tbest = Tworst = Tavg = ϴ(logn)

4)

Dividing the search space into parts at each step seems to provide improvements in time complexity, but in depth, increasing the number of partitions increases the number of comparasions to be made. For example, ternary search makes 4 comparisions at each step, while binary search makes 2 comparisions.

There is an delusion here. Constant terms are ignored when calculating the time complexity of an algorithm. This makes the idea wrong when looking at the time complexity of the algorithm as in ternary search. But the time complexity of an algorithm refers to the amount of time it takes for the algorithm to complete. Increasing the number of divisors will increase the number of comparasions made at each step. This will reduce the execution time of the algorithm.

Dividing the algorithm into n parts will make our time complexity O(1), but when we look at the number of comparasions, we will see that dividing into n parts is no different than doing a linear search. Because after dividing into n parts, we will search for the key in these n parts, respectively. This means that all elements are searched in a worst case scenario.

5)

a) Best Case: The best-case scenario for interpolation search occurs when the target value is found at the first position in the array, or when the value is equal to the value at the midpoint of the array. In either of these cases, the time complexity of interpolation search is θ(1), since the search will be completed in a single step.

b) The main difference between interpolation search and binary search in the manner of work is the way they narrow the search space. Binary search works by continuously dividing the search area in half, comparing the target value with the value at the midpoint of the current search space. Interpolation search, on the other hand, uses an estimate of the target value's position to determine the next position to search. This estimate is based on the values of other elements in the array and can potentially allow interpolation search to find the target value faster than a binary search in some cases.

Time Complexity

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

Space Complexity

	Best Case	Worst Case	Average Case
Binary Search	Θ(1)	Θ(1)	Θ(1)
Interpolation Search	Θ(1)	Θ(1)	Θ(1)

Another difference is in terms of time complexity. In the best-case scenario, both interpolation search and binary search have a time complexity of θ(1), since the search will be completed in a single step. However, in the worst-case scenario, interpolation search can work θ(n) if the search space aren’t uniformly distributed. But if our search space is uniformly distributed, interpolation search average case is better than binary search in terms of the time complexity.

Overall, both interpolation search and binary search can be useful searching methods for searching large, sorted arrays, and the best choice between the two will depend on the specific needs of the application.