# GTU Department of Computer Engineering
# CSE 321 – Fall 2022
# Homework #4

**Hasan Mutlu**
**1801042673**

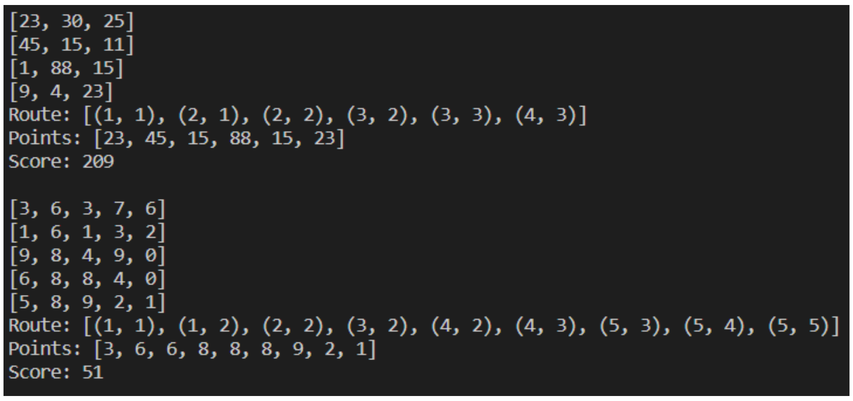
## Question 1

The first question consists of two methods. The first method find_max_score takes a 2D array as input. It first creates and initializes max_score and score variables and path and max_path tuple arrays. Then, it calls the count_scores method for the first cell in the array.

The count_scores is a recursive method that takes the grid, indices, current score, max score, current path, and the max path as arguments. First, it checks if the destination cell is reached. If it is, it compares the current path's score and the previously found max score. If current path has a higher score, it updates max_score and max_path variables and returns them. If destination cell isn't reached yet, it calls itself recursively for the next and below cell, if possible. While making the next call, it adds the score and indices of the next cell to the current ones in the function call. Then, it assigns the method returnees into max_score and max_path and returns these information.

After the first called count_scores method is returned to the find_max_score method, it prints the results and also returns them.

Here are example outputs. The first one is the example in the PDF and the second one is randomly generated.



```
[23, 30, 25]
[45, 15, 11]
[1, 88, 15]
[9, 4, 23]
Route: [(1, 1), (2, 1), (2, 2), (3, 2), (3, 3), (4, 3)]
Points: [23, 45, 15, 88, 15, 23]
Score: 209

[3, 6, 3, 7, 6]
[1, 6, 1, 3, 2]
[9, 8, 4, 9, 0]
[6, 8, 8, 4, 0]
[5, 8, 9, 2, 1]
Route: [(1, 1), (1, 2), (2, 2), (3, 2), (4, 2), (4, 3), (5, 3), (5, 4), (5, 5)]
Points: [3, 6, 6, 8, 8, 8, 9, 2, 1]
Score: 51
```

**Worst Case Time Complexity**

In any case, assuming that the matrix is NxM, there are O(N.M) possible paths to go from each cell. Every cell in the grid is covered as well, so that makes a time complexity of O(N^2.M^2).

## Question 2

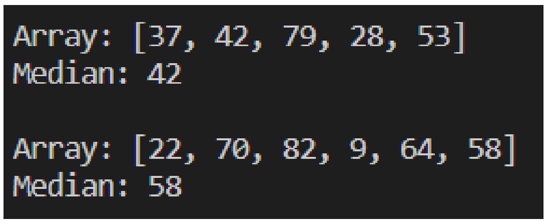
The second question consists of three methods and makes use of Quickselect algortihm, which is an algortihm to find the kth smallest value in a given array. In order to find the median, the value k must be the half of the array size.

The wrapper method find_median takes the array as argument. It checks if the array has odd or even number of elements. If there are odd number of elements, it sends the index of element in the middle; if there are even number of elements, it sends the index of the element on the left of the middle pair.

The quickSelect method takes the array, starting index, end index and the k value as arguments. First it checks if the index k is within the bounds of given start and end indices. Then it partitions the array, selecting the end point as the pivot. Partition method returns the lst index of the pivot element.

quickSelect method  checks if the pivot element is the kth smallest element and returns it if so. If the pivot element is larger than the kth smallest element, the function recursively calls itself with the subarray to the left of the pivot as the index range. If the pivot element is smaller than the kth smallest element, the function recursively calls itself with the subarray to the right of the pivot as the input, and the adjusted value of k that takes into account the number of elements in the left subarray.

Here are some example outputs with randomly generated arrays:



```
Array: [37, 42, 79, 28, 53]
Median: 42

Array: [22, 70, 82, 9, 64, 58]
Median: 58
```

**Worst Case Time Complexity**

If the selected pivot element is always either the smallest or the largest element in the given indices, the next subarray would have one less element than the current subarray. This would result in a need of N recursive calls and each recursive call would take O(N) for the partitioning, resulting in a worst time complexity of O(N^2).

## Question 3.1

findWinner1 method takes a circular linked list as argument. It chooses the head node as the current node and starts a loop. In each step, it deletes the next node from the list and prints that the current node eliminated the next node. Then, it moves to the next node. The loop goes on until there is only one element left in the list.

**Worst Case Time Complexity**

In any case, the algorithm iterates the list several times, the size of the list is reducing by half in each step. The deletion takes O(1) time since the deleted node’s connections are directly broken in place. So, it takes O(n) time in overall.

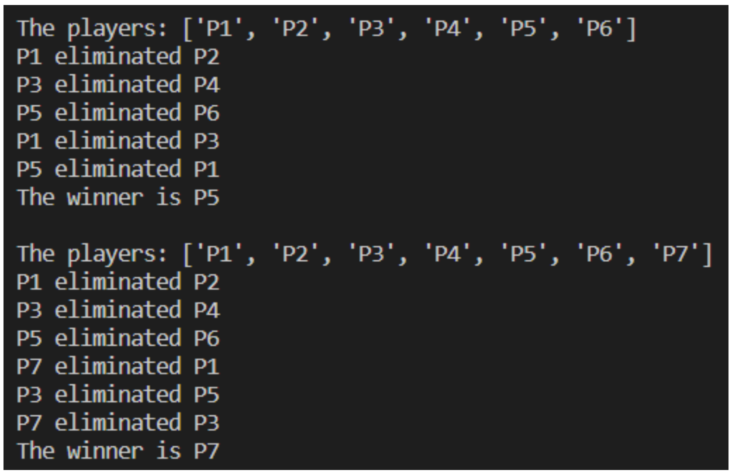
## Question 3.2

findWinner2 method takes a circular linked list as argument. It creates a new list and adds the players with odd indices to this new list. If the list has odd number of players, the last player is assigned as the head of the new list as it didn’t eliminate a rival yet. And then the method is recursively called for the new list until there is only one element left.

**Worst Case Time Complexity**

In any case, the list is halved in each step until there is only one element left so it takes O(logn) time.

Example output of the two algorithms. They give the same output.



```
The players: ['P1', 'P2', 'P3', 'P4', 'P5', 'P6']
P1 eliminated P2
P3 eliminated P4
P5 eliminated P6
P1 eliminated P3
P5 eliminated P1
The winner is P5

The players: ['P1', 'P2', 'P3', 'P4', 'P5', 'P6', 'P7']
P1 eliminated P2
P3 eliminated P4
P5 eliminated P6
P7 eliminated P1
P3 eliminated P5
P7 eliminated P3
The winner is P7
```

## Question 4

The divisor in a search algorithm refers to the number of parts into which the input is divided at each step. In ternary search, the array is divided into 3 parts at each step, while in binary search it is divided into 2 parts. The divisor affects the time complexity of the algorithm because the number of operations required increases as the divisor decreases. This is because a smaller divisor means that the algorithm has to perform more operations (divisions) in order to find the target element.

For example, if there are 6 elements in a list, binary search would require at least 3 recursions while ternary search would only require 2. This makes difference for large inputs.

Dividing the array in n elements would result in a time complexity of O(logn(n)) and that is equal to O(1).

## Question 5

The best-case scenario for interpolation search is when the target element is at the midpoint of the array. In this case, the algorithm will find the target element in the first iteration, making it the most efficient scenario.

Interpolation search uses an estimate of the position of the target element based on the value of the element and the values of the elements around it. This allows it to skip over large sections of the array that are not likely to contain the target element. Binary search, on the other hand, always divides the array into two equal parts and checks which part the target element is likely to be in.

Interpolation search tends to be faster than binary search in practice, especially when the array is large and the target element is closer to the midpoint of the array. This is because interpolation search is able to narrow down the search area more quickly by using an estimate of the position of the target element.