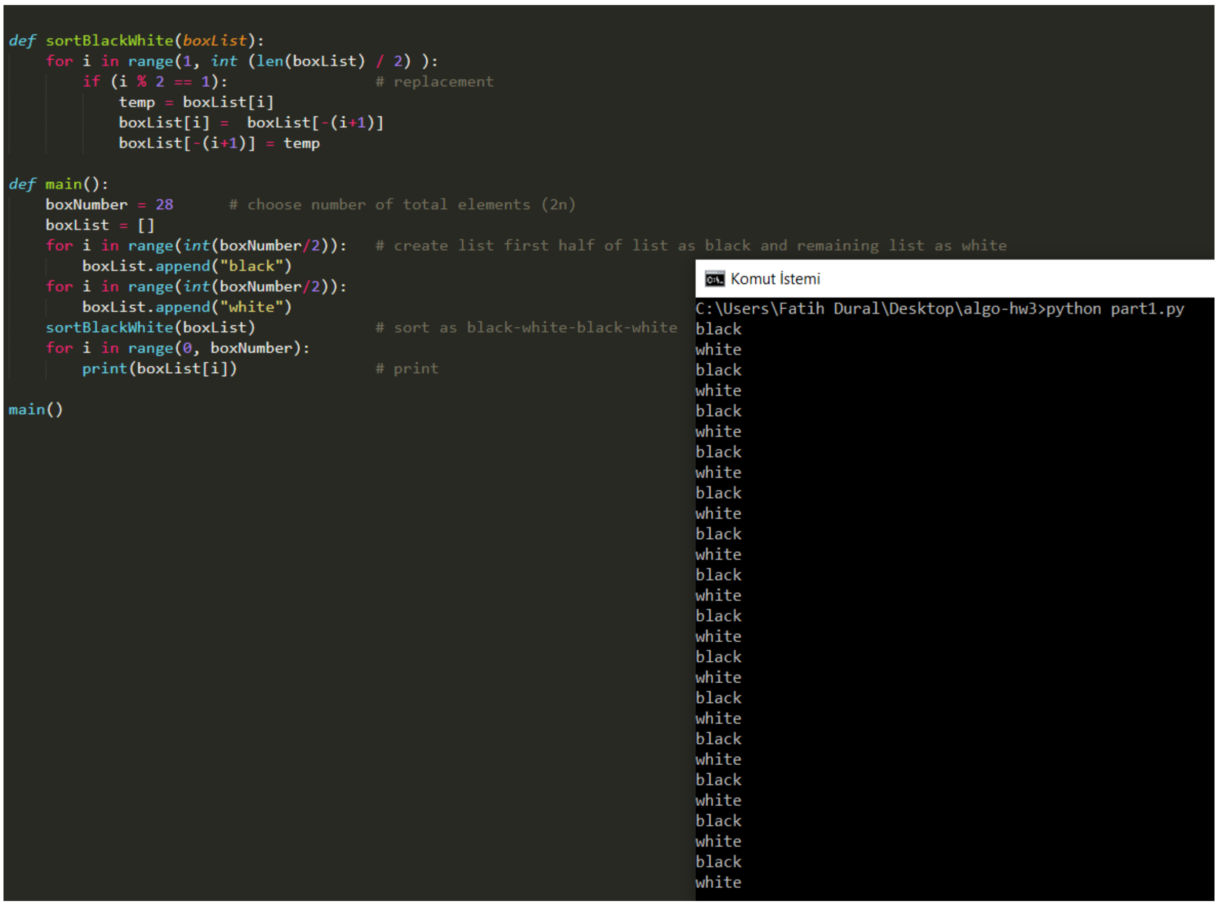
# CSE 321 Homework 3

# Fatih Dural

# 151044041

# Report

## Part1



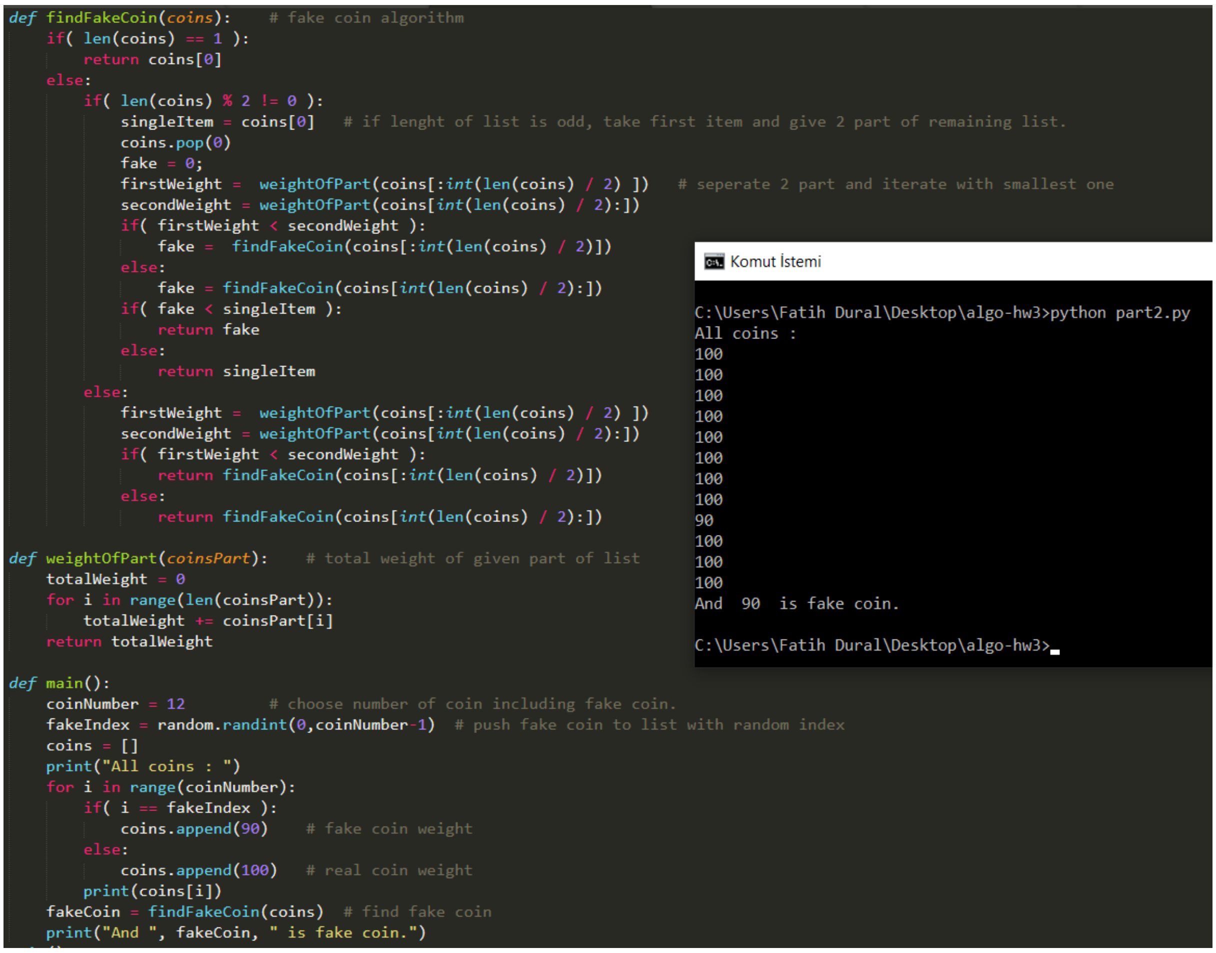
```python
def sortBlackWhite(boxList):
    for i in range(1, int (len(boxList) / 2) ):
        if (i % 2 == 1):                    # replacement
            temp = boxList[i]
            boxList[i] =  boxList[-(i+1)]
            boxList[-(i+1)] = temp

def main():
    boxNumber = 28      # choose number of total elements (2n)
    boxList = []
    for i in range(int(boxNumber/2)):   # create list first half of list as black and remaining list as white
        boxList.append("black")
    for i in range(int(boxNumber/2)):
        boxList.append("white")
    sortBlackWhite(boxList)             # sort as black-white-black-white
    for i in range(0, boxNumber):
        print(boxList[i])               # print

main()
```

Komut İstemi

```
C:\Users\Fatih Dural\Desktop\algo-hw3>python part1.py
black
white
black
white
black
white
black
white
black
white
black
white
black
white
black
white
black
white
black
white
black
white
black
white
black
white
black
white
```

There are 2n boxes; the first n of them shown as “black” named items in the boxList and remaining n boxes as “white” named items in the boxList. Decrease-and-conquer algorithm is made by sortBlackWhite function. The function turns size / 2 times. When iterating element is odd, it is replaced by an item that moves the same amount from behind(e.g first element and last element). In this way, some elements from the first and seconds parts are replaced. So, the elements are sorted as black-white-black-white. The list length(boxNumber) is optional, the user can change as he wishes. Looking at the complexity, it would be Θ(n) – theta(n), because loop turns size / 2 times and does swap operations. It does not vary by list length. The worst-case, best-case and average case is the same.

## Part2



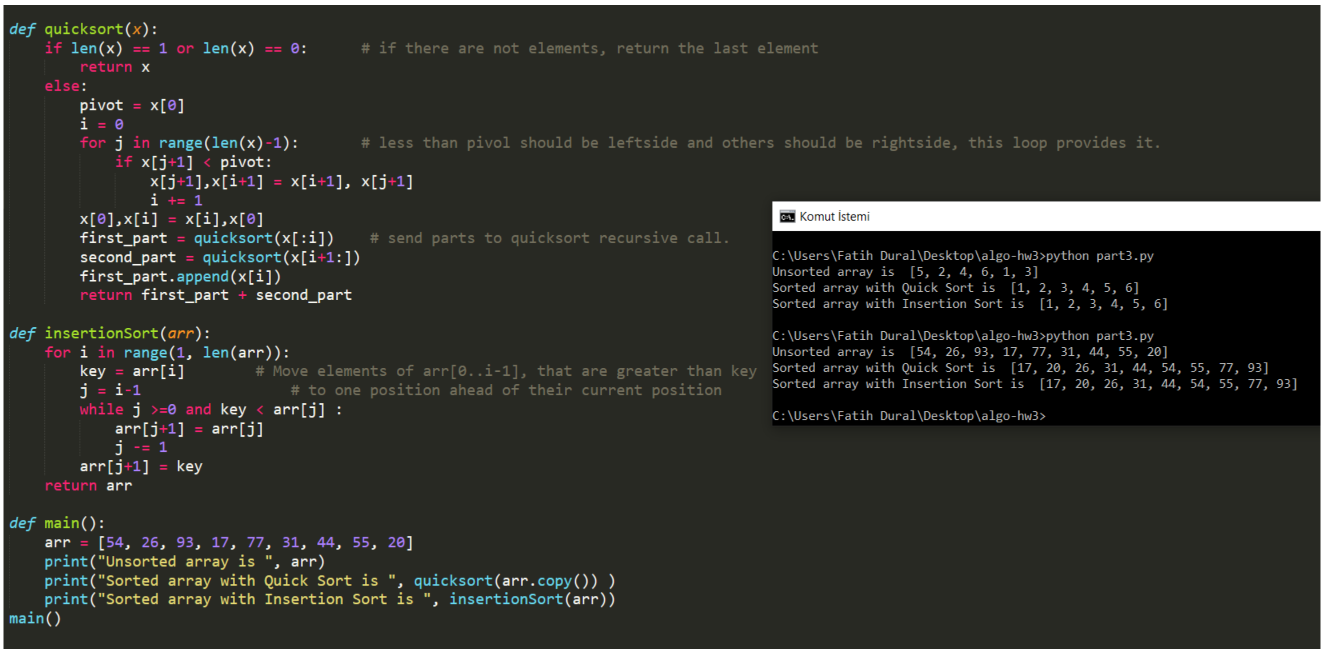
```python
def findFakeCoin(coins):    # fake coin algorithm
    if( len(coins) == 1 ):
        return coins[0]
    else:
        if( len(coins) % 2 != 0 ):
            singleItem = coins[0]   # if lenght of list is odd, take first item and give 2 part of remaining list.
            coins.pop(0)
            fake = 0;
            firstWeight =  weightOfPart(coins[:int(len(coins) / 2) ])   # seperate 2 part and iterate with smallest one
            secondWeight = weightOfPart(coins[int(len(coins) / 2):])
            if( firstWeight < secondWeight ):
                fake =  findFakeCoin(coins[:int(len(coins) / 2)])
            else:
                fake = findFakeCoin(coins[int(len(coins) / 2):])
            if( fake < singleItem ):
                return fake
            else:
                return singleItem
        else:
            firstWeight =  weightOfPart(coins[:int(len(coins) / 2) ])
            secondWeight = weightOfPart(coins[int(len(coins) / 2):])
            if( firstWeight < secondWeight ):
                return findFakeCoin(coins[:int(len(coins) / 2)])
            else:
                return findFakeCoin(coins[int(len(coins) / 2):])

def weightOfPart(coinsPart):    # total weight of given part of list
    totalWeight = 0
    for i in range(len(coinsPart)):
        totalWeight += coinsPart[i]
    return totalWeight

def main():
    coinNumber = 12         # choose number of coin including fake coin.
    fakeIndex = random.randint(0,coinNumber-1)  # push fake coin to list with random index
    coins = []
    print("All coins : ")
    for i in range(coinNumber):
        if( i == fakeIndex ):
            coins.append(90)    # fake coin weight
        else:
            coins.append(100)   # real coin weight
        print(coins[i])
    fakeCoin = findFakeCoin(coins)  # find fake coin
    print("And ", fakeCoin, " is fake coin.")
```

Komut İstemi

```
C:\Users\Fatih Dural\Desktop\algo-hw3>python part2.py
All coins :
100
100
100
100
100
100
100
100
90
100
100
100
And  90  is fake coin.

C:\Users\Fatih Dural\Desktop\algo-hw3>
```

In this part, there are real coins and one of them is fake. Real coins have same weight. We can compare their weight. I assume that the fake coin is lighter than others. The algorithm is divide the list into two parts and compare their weight. The side with fake coin will be lighter. Continue iteration with lighter one. In code, coins are created with integer numbers. The placed of fake coin is determined with randomly. Number of coins may vary. findFakeCoin function takes coins and compare two part of weight with helper function weightOfPart until one element remains. Because of always given small list, the last element would become fake one. Sometimes size of the list can be odd, it causes computing wrong weight because of the list cannot be fully split. We solve this problem by removing an element and comparing it with the returning value from recursive call. Because of dividing by two, time complexity is the same for worst-case, best-case and average case and it would be Θ(log(2n)).

## Part3



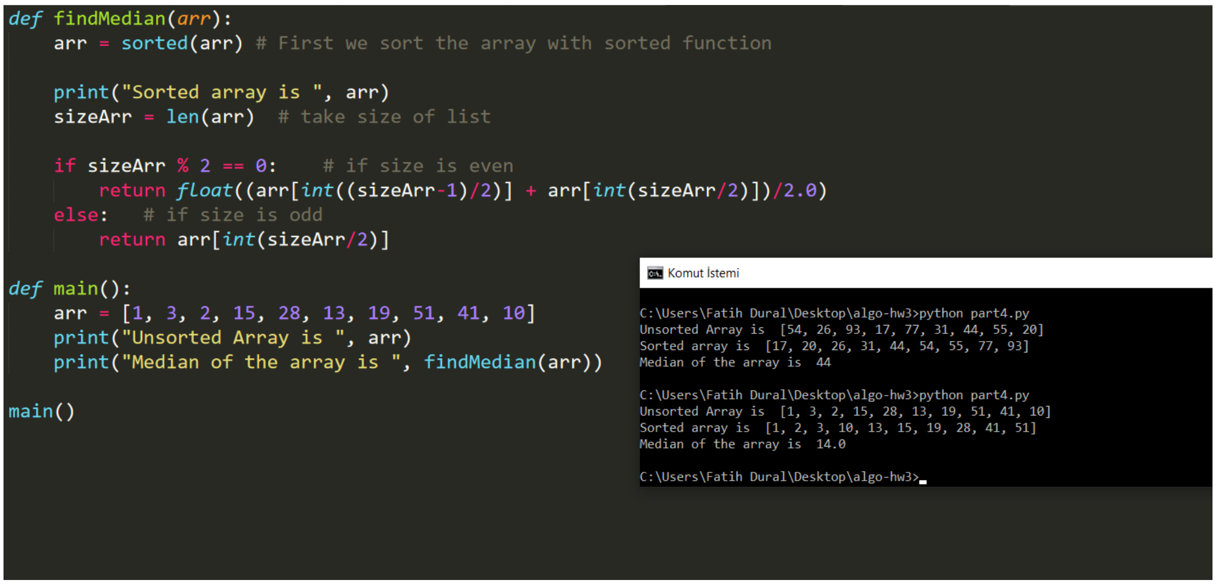
```python
def quicksort(x):
    if len(x) == 1 or len(x) == 0:      # if there are not elements, return the last element
        return x
    else:
        pivot = x[0]
        i = 0
        for j in range(len(x)-1):       # less than pivol should be leftside and others should be rightside, this loop provides it.
            if x[j+1] < pivot:
                x[j+1],x[i+1] = x[i+1], x[j+1]
                i += 1
        x[0],x[i] = x[i],x[0]
        first_part = quicksort(x[:i])    # send parts to quicksort recursive call.
        second_part = quicksort(x[i+1:])
        first_part.append(x[i])
        return first_part + second_part

def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]        # Move elements of arr[0..i-1], that are greater than key
        j = i-1                 # to one position ahead of their current position
        while j >=0 and key < arr[j] :
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
    return arr

def main():
    arr = [54, 26, 93, 17, 77, 31, 44, 55, 20]
    print("Unsorted array is ", arr)
    print("Sorted array with Quick Sort is ", quicksort(arr.copy()) )
    print("Sorted array with Insertion Sort is ", insertionSort(arr))
main()
```

Komut İstemi

```
C:\Users\Fatih Dural\Desktop\algo-hw3>python part3.py
Unsorted array is  [5, 2, 4, 6, 1, 3]
Sorted array with Quick Sort is  [1, 2, 3, 4, 5, 6]
Sorted array with Insertion Sort is  [1, 2, 3, 4, 5, 6]

C:\Users\Fatih Dural\Desktop\algo-hw3>python part3.py
Unsorted array is  [54, 26, 93, 17, 77, 31, 44, 55, 20]
Sorted array with Quick Sort is  [17, 20, 26, 31, 44, 54, 55, 77, 93]
Sorted array with Insertion Sort is  [17, 20, 26, 31, 44, 54, 55, 77, 93]

C:\Users\Fatih Dural\Desktop\algo-hw3>
```

In this part, we compare quicksort and insertion sort. QuickSort is one of the most efficient sorting algorithms and is based on the splitting of an array into smaller ones. The name comes from the fact that, quick sort is capable of sorting a list of data elements significantly faster than any of the common sorting algorithms. It picks an element as pivot and partitions the given array around the picked pivot. Insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The analogy can be understood from the style we arrange a deck of cards. Insertion sort counting the number of swap is bigger than quick sort generally. Quicksort faster than insertion sort. But, if number of element n is small, insertion sort may faster, because Quick sort has extra overhead from the recursive calls. Also insertion sort is more stable than Quick Sort and requires less memory. The average-case complexity of Quick Sort is O(nlogn) and the average-case complexity of Insertion Sort is O(n^2). So, quick sort is better than insertion sort with small input number, in other case insertion sort can be better.

## Part4



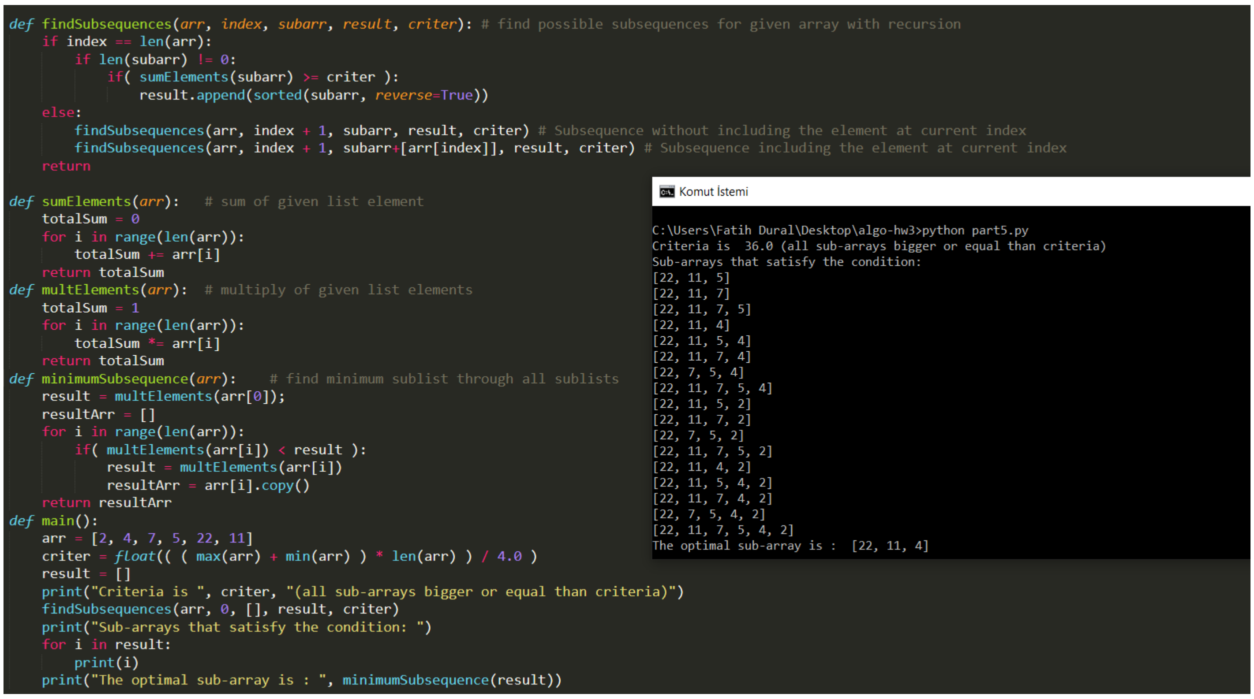
```python
def findMedian(arr):
    arr = sorted(arr) # First we sort the array with sorted function

    print("Sorted array is ", arr)
    sizeArr = len(arr)  # take size of list

    if sizeArr % 2 == 0:    # if size is even
        return float((arr[int((sizeArr-1)/2)] + arr[int(sizeArr/2)])/2.0)
    else:   # if size is odd
        return arr[int(sizeArr/2)]

def main():
    arr = [1, 3, 2, 15, 28, 13, 19, 51, 41, 10]
    print("Unsorted Array is ", arr)
    print("Median of the array is ", findMedian(arr))

main()
```

Komut İstemi

```
C:\Users\Fatih Dural\Desktop\algo-hw3>python part4.py
Unsorted Array is  [54, 26, 93, 17, 77, 31, 44, 55, 20]
Sorted array is  [17, 20, 26, 31, 44, 54, 55, 77, 93]
Median of the array is  44

C:\Users\Fatih Dural\Desktop\algo-hw3>python part4.py
Unsorted Array is  [1, 3, 2, 15, 28, 13, 19, 51, 41, 10]
Sorted array is  [1, 2, 3, 10, 13, 15, 19, 28, 41, 51]
Median of the array is  14.0

C:\Users\Fatih Dural\Desktop\algo-hw3>
```

In this part, we find the median of an unsorted array. Median is found in sorted arrays, so first we should sort given array. Median is the middle number in the even numbers and half the sum of the two middle numbers in the odd numbers. In code, array is specified and give the findMedian function as a parameter. Function sort items with python sort function. (I use python standart function because it is not forbidden). This sort function use timsort algorithm that splits the array into small pieces and executes insertion sort. Then, median is returned in according to size is even or odd. The worst-case time complexity is O(nlogn), because the sorting algorithm is so.

## Part5



```python
def findSubsequences(arr, index, subarr, result, criter): # find possible subsequences for given array with recursion
    if index == len(arr):
        if len(subarr) != 0:
            if( sumElements(subarr) >= criter ):
                result.append(sorted(subarr, reverse=True))
    else:
        findSubsequences(arr, index + 1, subarr, result, criter) # Subsequence without including the element at current index
        findSubsequences(arr, index + 1, subarr+[arr[index]], result, criter) # Subsequence including the element at current index
    return

def sumElements(arr):   # sum of given list element
    totalSum = 0
    for i in range(len(arr)):
        totalSum += arr[i]
    return totalSum
def multElements(arr):  # multiply of given list elements
    totalSum = 1
    for i in range(len(arr)):
        totalSum *= arr[i]
    return totalSum
def minimumSubsequence(arr):    # find minimum sublist through all sublists
    result = multElements(arr[0]);
    resultArr = []
    for i in range(len(arr)):
        if( multElements(arr[i]) < result ):
            result = multElements(arr[i])
            resultArr = arr[i].copy()
    return resultArr
def main():
    arr = [2, 4, 7, 5, 22, 11]
    criter = float(( ( max(arr) + min(arr) ) * len(arr) ) / 4.0 )
    result = []
    print("Criteria is ", criter, "(all sub-arrays bigger or equal than criteria)")
    findSubsequences(arr, 0, [], result, criter)
    print("Sub-arrays that satisfy the condition: ")
    for i in result:
        print(i)
    print("The optimal sub-array is : ", minimumSubsequence(result))
```

Komut İstemi

```
C:\Users\Fatih Dural\Desktop\algo-hw3>python part5.py
Criteria is  36.0 (all sub-arrays bigger or equal than criteria)
Sub-arrays that satisfy the condition:
[22, 11, 5]
[22, 11, 7]
[22, 11, 7, 5]
[22, 11, 4]
[22, 11, 5, 4]
[22, 11, 7, 4]
[22, 7, 5, 4]
[22, 11, 7, 5, 4]
[22, 11, 5, 2]
[22, 11, 7, 2]
[22, 7, 5, 2]
[22, 11, 7, 5, 2]
[22, 11, 4, 2]
[22, 11, 5, 4, 2]
[22, 11, 7, 4, 2]
[22, 7, 5, 4, 2]
[22, 11, 7, 5, 4, 2]
The optimal sub-array is :  [22, 11, 4]
```

In the last part, there is an array which consists of n integer values. There are sub-arrays that meet a specific criterion. In these sub-arrays, the optimal sub-array is the least multiplying elements compared to others. In code, criter is specified with (max(array) + min(array)) * size / 4. findSubsequence function is found all subsequence that complying the criteria. Subsequences is appended to result list. Then, result list is sended to minimumSubsequence for finding to the optimalest one. This function traverses subsequences, computes multiplying of the elements and return the smallest one. The worst-case complexity would be O(n^2) because of recursive call turn twice and every time turns n times.

**FATİH DURAL**

**151044041**