CSE321 Introduction to Algorithm Design

Homework3 Report

Part1

To solve this problem, I created a Boxes class. It has boxList and size. boxList is used to store black “B” and white “W”, size is used for amount of black and white. In constructor of Boxes class, I appended “B” and “W” by order into the boxList.

To test this part, you have to create a Boxes object with size “n”. After that you have to call **makePattern** method. It makes the pattern with “B” and “W”. To see result, you have to call **printBoxList** method. It prints the new pattern.

makePattern method calls **makePatternRecursive** method. makePatternRecursive method takes index of boxList’s start and index of boxList’s end. Best case of this recursive method is, start index reaches middle of list. Else, swap “B” and “W” if start index is odd and end index is even, then increase start index by one and decrease end index by one and recall makePatternRecursive method recursively.

Usage and Results of Part1

```
def testPart1():

    printSeperator()
    print("Algorithm HW3 Part1")
    print("Your Box Object : ")
    boxes = Boxes(3)
    boxes.printBoxList()
    print("After Pattern : ")
    boxes.makePattern()
    boxes.printBoxList()
    print()
    print("Your Box Object : ")
    boxes2 = Boxes(6)
    boxes2.printBoxList()
    print("After Pattern : ")
    boxes2.makePattern()
    boxes2.printBoxList()
    printSeperator()
```

```
================================================================================
Algorithm HW3 Part1
Your Box Object :
['B', 'B', 'B', 'W', 'W', 'W']
After Pattern :
['B', 'W', 'B', 'W', 'B', 'W']

Your Box Object :
['B', 'B', 'B', 'B', 'B', 'B', 'W', 'W', 'W', 'W', 'W', 'W']
After Pattern :
['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']
================================================================================
```

Analysis of Part1

-Best Case: The list has 2n items. Best case occurs if n equals to 1. When n is 1 the boxList is like [“B”, ”W”] and it is already patterned. And recursive method just returns the list. So that best-case **B(n) Ɛ Θ (1).**

-Worst Case: My algorithm traverses the list starting from begin and starting from end. And it finishes when start point and end point reach each other. In other words, it traverse list. So that worst-case **W(n) Ɛ Θ (n).**

-Average Case: The recurrence relation of my algorithm is **T(n) = T(n-2) + O(1)**

I control the list from start and from end so that it is T(n-2). O(1) is some operations such as increasing start index value, decreasing end index value…

T(n) = T(n-2) + 1 ------> T(n-2) = T(n-4) + 1, T(n-4) = T(n-6) + 1, T(n-6) = T(n-8) + 1

T(n) = T(n-4) + 1 + 1

T(n) = T(n-8) + 1 + 1 + 1

After n/2 steps…

T(n) = T(0) + 1 + 1 + .. + 1 (n/2 times 1)

So shat average-case **A(n) Ɛ Θ (n).**

Part2

To solve this problem, I created a Coin class. It has name and weight. name is used to specify name of Coin and weight is used to give weight for Coin. It is 10 for all Coin.

To test this part, you have to specify an integer number which is represents the amount of Coins, and you have to create a list to store Coin object. You have to call **createCoins** with amount and list. It creates Coin with name and weight 10. Then it appends these Coins into the list. Then it creates a random number for index of fake coin. Then makes the weight of Coin, which is stored in the random index, fakeWeight (404). Finally, we have n Coins and one of them is fake coin. You have to call **findFakeCoin** method with the list that just created. This method returns fake coin. Then you have to see fake coin, call **printFakeCoin** method with return value of findFakeCoin.

findFakeCoin calls **findFakeCoinRecursive** method. This method takes start index, end index and coinsList. Best-cases of this method are reaching start index to end index and size of coinsList is less than or equal to 2. In this case we cannot understand which coin is fake. Else, it controls weight of index start and weight of index start + 1. If these are not equal it controls, the weight of index start + 2. Which is equal to weight of index start + 2, the other is definitely fake coin and it returns fake coin. Otherwise it calls **helperRecursive** method.

helperRecursive takes coinsList, notFakeWeight and index. Best-case of this method is index is greater or equal to size of coinsList. Else, control the weight with notFakeWeight. If these are equal to each other, increase the index by one and recall helperRecursive method, otherwise return the fake coin. We know the coinsList has only one fake coin and if these weight are not equal to each other; fake coin is founded. Weighbridge logic is used to check two coins' weight.

Usage and Results of Part2

```
def testPart2():
    printSeperator()
    print("Algorithm HW3 Part2")
    numberOfCoins = 5
    coinsList = []
    createCoins(coinsList, numberOfCoins)
    print("Your Coins : ")
    printCoinsList(coinsList)
    FakeCoin = findFakeCoin(coinsList)
    print("Fake Coin is : ")
    FakeCoin.printCoin()
    print()
    numberOfCoins = 8
    coinsList = []
    createCoins(coinsList, numberOfCoins)
    print("Your Coins : ")
    printCoinsList(coinsList)
    FakeCoin = findFakeCoin(coinsList)
    print("Fake Coin is : ")
    FakeCoin.printCoin()
    printSeperator()
```

```
================================================================================
Algorithm HW3 Part2
Your Coins :
Name : Coin_0 Weight : 10
Name : Coin_1 Weight : 10
Name : Coin_2 Weight : 10
Name : Coin_3 Weight : 404
Name : Coin_4 Weight : 10
Fake Coin is :
Name : Coin_3 Weight : 404

Your Coins :
Name : Coin_0 Weight : 10
Name : Coin_1 Weight : 10
Name : Coin_2 Weight : 10
Name : Coin_3 Weight : 10
Name : Coin_4 Weight : 10
Name : Coin_5 Weight : 404
Name : Coin_6 Weight : 10
Name : Coin_7 Weight : 10
Fake Coin is :
Name : Coin_5 Weight : 404
================================================================================
```

Analysis of Part2

-Best Case: Best case of my algorithm is constant time. If fake coin is in the first index of the list or second index of the list, I can always find the fake coin with checking third index of the list. In this case no needs to call recursive method or loop, only checking first 3 index is enough. So that best-case **B(n) Ɛ Θ (1).**

-Worst Case: Worst case of my algorithm occurs if the fake coin in the end of the list. My algorithm traverses the list starting from begin. And it finishes when start point reaches the end of the list. This method checks index and index + 1 elements of the list. So, it makes n comparisons to find fake coin. So that worst-case **W(n) Ɛ Θ (n).**

-Average Case: The recurrence relation of my algorithm is **T(n) = T(n-1) + O(1)**

O(1) is some operations such as increasing start index value, check the 2 coins have same weight or not…

T(n) = T(n-1) + 1 ------> T(n-2) = T(n-3) + 1, T(n-3) = T(n-4) + 1, T(n-4) = T(n-5) + 1

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

T(n) = T(n-3) + 1 + 1 + 1

After n steps…

T(n) = T(0) + 1 + 1 + .. + 1 (n times 1)

So shat average-case **A(n) Ɛ Θ (n).**

Part3

To solve this problem, I implemented **insertionSort, partition, quickSortRecursive** and **quicksort** methods. insertionSort is for insertion sort and others are for quick sort.

To test this part, create lists to sort. Call insertionSort with list to sort list by insertion sort. It returns sorted list and amount of swap operation. Print them and see the results.
Call quicksort with list to sort list by quick sort. It returns sorted list and amount of swap operation.

insetionSort takes a list and traverses it from begin to end. If next item smaller than current item goes backward to find bigger than itself. Repeats it until the list is sorted.

Quicksort method takes a list and call quickSortRecursive with list, start, end and coutOfSwap. quickSortRecursive method calls partition to find pivot index.

partition method takes last element as pivot, places the pivot element at its correct position in sorted array, and places all items, which are smaller than pivot, to left of pivot and all bigger items to right of pivot.

quickSortRecursive separately sort elements before partition and after partition.

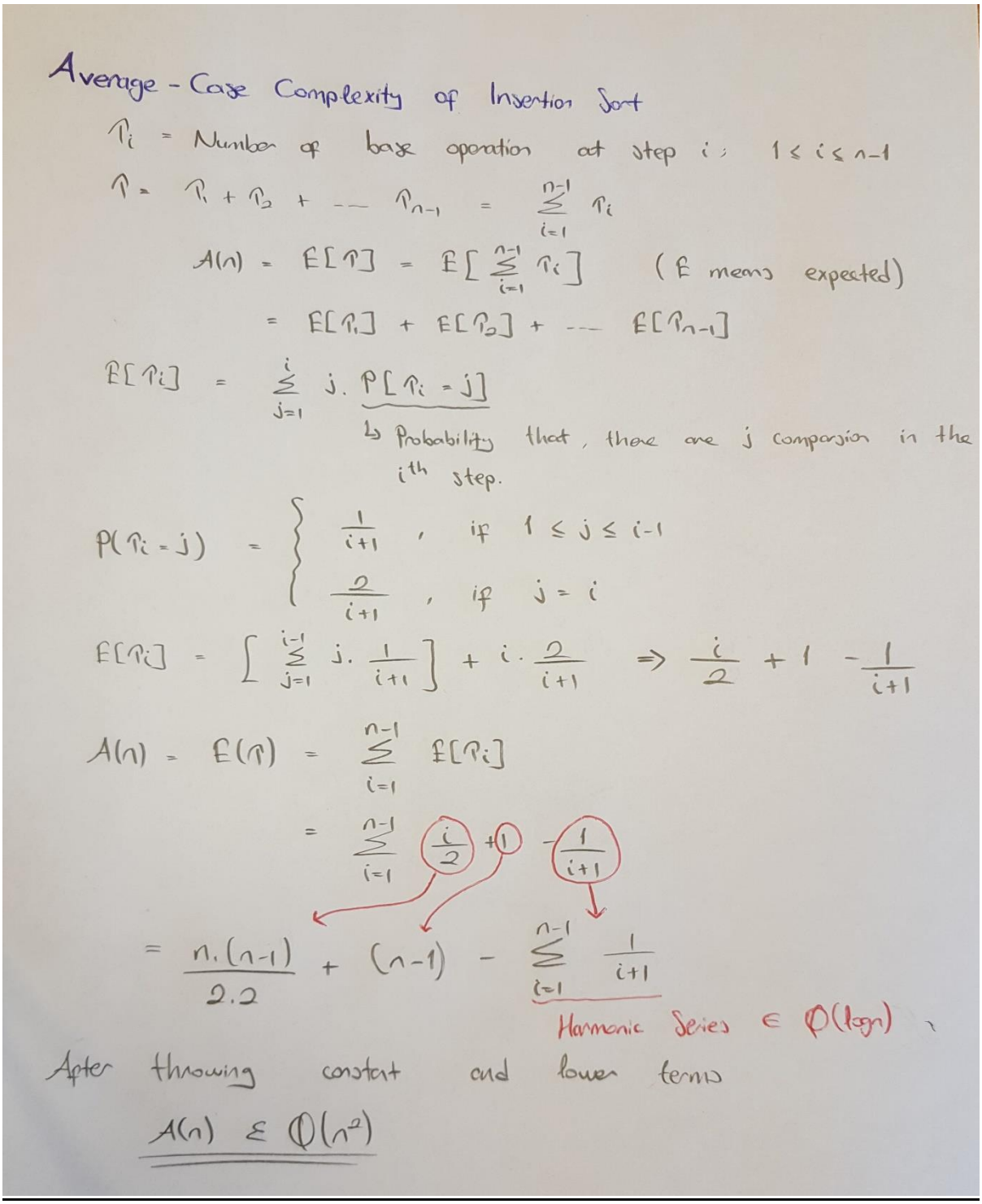
Usage and Results of Part3

```
def testPart3():
    printSeperator()
    print("Algorithm HW3 Part3")
    list_1 = [1, 2, 3, 4, 5]
    print("List :", list_1)
    sortedList_1, countOfSwap1 = insertionSort(list_1)
    print("Sorted List (Insertion):", sortedList_1)
    print("countOfSwap :", countOfSwap1)
    print()
    list_2 = [1, 2, 3, 4, 5]
    print("List :", list_2)
    sortedList_2, countOfSwap2 = quickSort(list_2)
    print("Sorted List (Quick):", sortedList_2)
    print("countOfSwap :", countOfSwap2)
    print()
    list_3 = [6, 7, 9, 8, 10]
    print("List :", list_3)
    sortedList_1, countOfSwap1 = insertionSort(list_3)
    print("Sorted List (Insertion):", sortedList_1)
    print("countOfSwap :", countOfSwap1)
    print()
    list_4 = [6, 7, 9, 8, 10]
    print("List :", list_4)
    sortedList_2,countOfSwap2 = quickSort(list_4)
    print("Sorted List (Quick):", sortedList_2)
    print("countOfSwap :", countOfSwap2)
    print()
    list_5 = [10, 7, 8, 9, 1, 5]
    print("List :", list_5)
    sortedList_1, countOfSwap1 = insertionSort(list_5)
    print("Sorted List (Insertion):", sortedList_1)
    print("countOfSwap :", countOfSwap1)
    print()
    list_6 = [10, 7, 8, 9, 1, 5]
    print("List :", list_6)
    sortedList_2,countOfSwap2 = quickSort(list_6)
    print("Sorted List (Quick):", sortedList_2)
    print("countOfSwap :", countOfSwap2)
    printSeperator()
```

```
================================================================================
Algorithm HW3 Part3
List : [1, 2, 3, 4, 5]
Sorted List (Insertion): [1, 2, 3, 4, 5]
countOfSwap : 0

List : [1, 2, 3, 4, 5]
Sorted List (Quick): [1, 2, 3, 4, 5]
countOfSwap : 14

List : [6, 7, 9, 8, 10]
Sorted List (Insertion): [6, 7, 8, 9, 10]
countOfSwap : 1

List : [6, 7, 9, 8, 10]
Sorted List (Quick): [6, 7, 8, 9, 10]
countOfSwap : 10

List : [10, 7, 8, 9, 1, 5]
Sorted List (Insertion): [1, 5, 7, 8, 9, 10]
countOfSwap : 11

List : [10, 7, 8, 9, 1, 5]
Sorted List (Quick): [1, 5, 7, 8, 9, 10]
countOfSwap : 5
================================================================================
```

Analysis of Part3

Analysis of Insertion Sort



Average - Case Complexity of Insertion Sort

T_i = Number of base operation at step i; $1 \le i \le n-1$

$$T = T_1 + T_2 + \cdots T_{n-1} = \sum_{i=1}^{n-1} T_i$$

$$A(n) = E[T] = E\left[\sum_{i=1}^{n-1} T_i\right] \quad (E \text{ means expected})$$

$$= E[T_1] + E[T_2] + \cdots E[T_{n-1}]$$

$$E[T_i] = \sum_{j=1}^{i} j \cdot P[T_i = j]$$

↳ Probability that, there are j comparsion in the i^{th} step.

$$P(T_i = j) = \begin{cases} \frac{1}{i+1}, & \text{if } 1 \le j \le i-1 \\ \frac{2}{i+1}, & \text{if } j = i \end{cases}$$

$$E[T_i] = \left[\sum_{j=1}^{i-1} j \cdot \frac{1}{i+1}\right] + i \cdot \frac{2}{i+1} \Rightarrow \frac{i}{2} + 1 - \frac{1}{i+1}$$

$$A(n) = E(T) = \sum_{i=1}^{n-1} E[T_i]$$

$$= \sum_{i=1}^{n-1} \frac{i}{2} + 1 - \frac{1}{i+1}$$

$$= \frac{n \cdot (n-1)}{2 \cdot 2} + (n-1) - \sum_{i=1}^{n-1} \frac{1}{i+1}$$

Harmonic Series $\in \Theta(\log n)$

After throwing constant and lower terms

$$A(n) \in \Theta(n^2)$$

Analysis of Quick Sort

Average-Case Complexity of Quick Sort

$T = T_1 + T_2$ (T_1 : Rearrange , T_2 : Recursive Call)

$A(n) = E[T] = E[T_1] + E[T_2]$

$E[T_2] = \sum_x E[T_2 \mid \bar{x} = x] \cdot P(\bar{x} = x)$, $E[T_1] = n+1$

$A(n) = n+1 + \sum_{i=1}^{n} E[T_2 \mid \bar{x} = i] \cdot \underbrace{P(\bar{x} = i)}_{1/n}$

$A(n) = n+1 + \sum_{i=1}^{n} \underbrace{[A(i-1) + A(n-i)]}_{2.[A(0) + \cdots A(n-1)]} \cdot \frac{1}{n}$

$n.A(n) = n.[(n+1) + 2.[A(0) + A(1) + \cdots A(n-1)]]$

$-(n-1).A(n-1) = (n-1).[n + 2.[A(0) + A(1) + \cdots A(n-2)]]$

$n.A(n) - (n-1).A(n-1) = 2n + 2.(A(n-1))$

$\frac{A(n)}{n+1} = \frac{2}{n+1} + \frac{A(n-1)}{n+1}$ } Reccurance formula ($t(n) = \frac{A(n)}{n+1}$)

$t(n) = \frac{2}{n+1} + t(n-1)$

$t(n) = \frac{2}{2} + t(0)$

$t(0) = \frac{2}{3} + t(1)$

$t(n) = \frac{2}{n+1} + t(n-1)$

$+$

$t(n) = t(0) + 2. \sum_{i=1}^{n} \frac{1}{i+1}$ Harmonic Series ($\log n$)

$t(n) = 2.\log n$

$A(n) = (n+1).t(n)$ → $A(n) = (n+1).2\log n$

$A(n) \in \Theta(n \log n)$

Comparing Insertion Sort and Quick Sort with Swap Count

Comparing these two algorithms with their swap count, gives us an information which algorithm is more efficient in which case.

Insertion sort is more efficient if the list is sorted or nearly sorted. If the list is sorted there is no swap, only traverses all list from begin to end one time. If the list is nearly sorted, then traverses the all list and find the value and goes back to find its correct position. In this case count of swap also very less. But the list is reverse sorted or very far from sorted, it occurs the count of swap very high. In each iteration it must find correct position for current element.

Quick sort is more efficient if the list is far from the sorted. If pivot is selected good, it makes the algorithm fast. In other words, less count of swap is when the pivot element divides the list into two equal halves by coming exactly in the middle position. If the list is already sorted or nearly sorted, this makes the count of swap higher.

As you see top of the usage of part3, insertion sort made less swap for [1,2,3,4,5] and [6,7,9,8,10] but quick sort makes higher swap. Quick sort makes less swap for [10, 7, 8, 9, 1, 5].

If list is nearly sorted you should choose insertion sort, otherwise choose quick sort.

Part4

To solve this problem, I implemented **findPivotIndex**, **find_kthSmallestItem** and **findMedian** methods.

To test this part, you have to create a list of numbers. This list has to unsorted. Then call findMedian method with list. This method returns the median of the list.

There are some ways to find median of a list. For examples, first of all, sorting the list and getting the middle element is median of the list, or using quick select algorithm. I used quick select algorithm. This algorithm finds the k-th smallest element in an unsorted list. If list's size is odd number and if I can find len(list)/2 smallest item, it will be median of the list. If list's size is even number and if I can find len(list)/2 smallest item and len(list)/2 - 1 smallest item, average of these two items will be median of the list.

findPivotIndex considers last element as pivot and adds element left which value is less than itself, adds right element which values is bigger than itself. And swap the pivot its final position. kthSmallestItem is used to find k-th smallest element without sorting. If pivotIndex is equal to k, we have found the k-th smallest element and we return it. If pivotIndex is bigger than k, then we recur for left part of the list. If pivotIndex is smaller than k, then we recur for right part.

findMedian method calls kthSmallestItem method inside of itself. If size of the list is odd, then call kthSmallestItem with k is len(list)/2 and return median, else call kthSmallestItem with k is len(list)/2 and len(list)/2 – 1. And average this result and return it as a median.

Usage and Results of Part4

```
def testPart4():
    printSeperator()
    print("Algorithm HW3 Part4")
    unSortedArray_1 = [14,55,27,11,20,19,49,47,88]
    unSortedArray_2 = [48,24,30,8,11,29,1,4]

    print("Your Unsorted Array is :")
    print(unSortedArray_1)
    median = findMedian(unSortedArray_1)
    print("Median is :",median)
    print()
    print("Your Unsorted Array is :")
    print(unSortedArray_2)
    median = findMedian(unSortedArray_2)
    print("Median is :",median)
    printSeperator()
```

```
================================================================================
Algorithm HW3 Part4
Your Unsorted Array is :
[14, 55, 27, 11, 20, 19, 49, 47, 88]
Median is : 27

Your Unsorted Array is :
[48, 24, 30, 8, 11, 29, 1, 4]
Median is : 17.5
================================================================================
```

Analysis of Part4

-Worst Case: Time complexity is changeable with choosing pivot item. For example, if you want to find minimum item in the list with choosing last item as a pivot, it will be worst case. And it is $W(n) \in \Theta(n^2)$.

Worst case of my algorithm. Occurs if the size of the list is even and medians are in the first and second indexes, choosing last item as a pivot. If the size is even, I must call find_kthSmallestElement two times and its cost is in this case $\Theta(n^2)$.

So that worst-case **$W(n) \in \Theta(n^2)$.**

Part5

To solve this problem, I implemented **finMaxAndMinItem, isGreaterThanOrEqualSumB findSUM_B, findSubsets, findMinimumNumberOfMultiplicationSubset, findRecursive** and **multiplicationOfArray** methods.

To test this part, create a list of integer item. Call findMinimumNumberOfMultiplicationSubset with the list. findMinimumNumberOfMultiplicationSubset returns a list of subset of A. This subset is optimal subset which is asked in homework pdf.

First of all, I have to find SUM_B. So that I find max(A) and min(A). To find max and min A, I used **finMaxAndMinItem method.** Inside of this method I used **find_kthSmallestItem** method which was implemented for Part4. Min(A) is equal to 0^{th} index. Smallest element on the list so that call find_kthSmallestItem with 0. Max(A) is equal len(list) -1th index so that call find_kthSmalletItem with len(list) -1. Now we have max and min item of A. Then I calculate SUM_B. To do that I used **findSUM** method. To find all subsets, I implemented **findSubsets** method. This method finds all subsets which sum of all elements is greater than or equal to SUM_B. To find subsets with given condition I use a controller method with name **isGreaterThanOrEqualSumB.** This method takes a list and SUM_B that is found just before. Returns true if sum of all elements in the list is greater than or equal to SUM_B, else return false. Now I have all subset with given condition. Then I implemented **findMinimumNumberOfMultiplicationSubset** to find optimal subset.

findMinimumNumberOfMultiplicationSubset method uses **multiplicationOfArray** and **findRecursive** methods. multiplicationOfArray takes a list and returns the multiplication of all elements. findRecursive method is takes a list, index and minMulltiplication list. It traverses list from begin to and finds the minMulltiplication list inside of list and returns it.

Usage and Results of Part5

```
def testPart5():
    printSeperator()
    print("Algorithm HW3 Part5")
    A = [2, 4, 7, 5, 22, 11]

    #allSubsetsOfGivenCondition = findSubsets(A)    #These lines print all subsets which are >= SUM_B.
    #printSubsets(allSubsetsOfGivenCondition)

    optimalSubSet = findMinimumNumberOfMultiplicationSubset(A)
    print("The optimal sub-set is",optimalSubSet)
    printSeperator()
```

```
================================================================================
Algorithm HW3 Part5
The optimal sub-set is [4, 11, 22]
================================================================================
```

Analysis of Part5

-Worst Case: To make analysis of this algorithm find all method's time complexity which are used in this problem. First of all, to run this part you only call findMinimumNumberOfMultiplicationSubset method. Inside this method, calling findSubsets method. This method has nested for loops, first loop works 2^n, other loops works n times. Also, inside the first loop there is also a method which works size of current subset time. We know it is smaller than real list so we can ignore it. So that its time complexity is $O(n * 2^n)$.

findMinimumNumberOfMultiplicationSubset also calls findRecursive method. It runs size of list which is parameter of findRecursive method. We know that it can not be greater than all subset of list A (which is parameter of findMinimumNumberOfMultiplicationSubset method). Let suppose that findRecursive algorithm works m times, total time complexity of findMinimumNumberOfMultiplicationSubset method is $n^* 2^n + m$. In this case m is like constant value. So that worst-case **$W(n) \in \Theta (n^* 2^n)$.**

ALİ HAYDAR KURBAN

151044058