CSE321 HOMEWORK 5 REPORT

1) In this algorithm algorithm starts with selecting the less costed place for first month.While iterating program compare if the move would be better choice or not by comparing the values operation cost of current place and operation cost of other place + movement cost.The algorithm selects the minimum on each iteration which will provide optimum solution.

```
def planOfMinCost(m,opcosts):
    place = 0
    count = 0
    minsolution = []
    if opcosts[0][0] < opcosts[1][0]:
        place = 0
        count = count + opcosts[0][0]
        minsolution.insert(len(minsolution),"NY")
    else:
        place = 1
        count = count + opcosts[1][0]
        minsolution.insert(len(minsolution),"SF")
    for i in range(1,len(opcosts[0])):
        if place == 0:
            if opcosts[0][i] < opcosts[1][i] + m:
                count = count + opcosts[0][i]
                minsolution.insert(len(minsolution),"NY")
            else:
                place = 1
                count = count + opcosts[1][i] + m
                minsolution.insert(len(minsolution),"SF")
        else:
            if opcosts[1][i] < opcosts[0][i] + m:
                count = count + opcosts[1][i]
                minsolution.insert(len(minsolution),"SF")
            else:
                place = 0
                count = count + opcosts[0][i] + m
                minsolution.insert(len(minsolution),"NY")
    print("Optimum Solution is below with total cost %d" %(count))
    print(minsolution)
```

If the size of input(operation cost list) is considered to be N,this algorithm always performs operation N times.The running time in the worst case,average case and best case is O(N).

2) In this particular problem,before processing the list ,the list is sorted by finish time in ascending order which will be the main idea of this greedy algorithm.After sorting operation,program starts from the head of list to select optimal symposiums.After adding a symposium,it's finish time is kept in variable in order to compare next symposiums start time is whether before it or after it.If the start time of the next symposium is smaller than the last added symposiums finish time then it's an overlap,which will lead to skipping that symposium without adding to optimal solution.

```
def sortByAscendingFinishTime(arr):
    for passnum in range(len(arr)-1,0,-1):
        for i in range(passnum):
            if arr[i][1]>arr[i+1][1]:
                temp = arr[i]
                arr[i] = arr[i+1]
                arr[i+1] = temp
def optimalSymposium():
    symposiums = [[13,15],[11,12],[9,11],[11,14],[13,16],[15,17]]
    sortByAscendingFinishTime(symposiums)
    lastFinished = [-1,-1]
    solution = []
    for i in range(len(symposiums)):
        if symposiums[i][0] >= lastFinished[1]:
            solution.insert(len(solution),symposiums[i])
            lastFinished = symposiums[i]
```

The function optimalSymposium without sorting takes O(n) time in the worst,average and best case.However before starting algortihm does bubblesort which can make n*(n-1) / 2 comparisons in the worst case.The running time in the worst case of this algorithm is O(n^2)

3) In this problem since the values of the given input list can be negative value,there will be a big table which will keep the sum values inside itself.The illustration of the table is below:

Min Max

Bool									

Min is the sum of negative values which is the minimum number of a subset can have as sum and the max is the sum of positive values as well.Each row shows the element number of that row with different sum values.

Since negative indexes can not be represented,minimum value is subtracted from index while operating.

We have the following recurrence:

dpTable[i][j-sumNeg] =

(arr[i] == j) **OR**
(dpTable[i-1][j - sumNeg]) **OR**
(dpTable[i-1][j - sumNeg - arr[i]])

During the operations if there is a true element found in (–min)th row (which represent sum of 0) then there is a subset which sum to 0.

```
def subArraySum(arr):
    sumNeg = 0
    sumPos = 0
    for i in range(0,len(arr)):
        if arr[i]<0:
            sumNeg = sumNeg + arr[i]
        else :
            sumPos = sumPos + arr[i]

    dpTable = [[None for i in range(sumPos-sumNeg+1)] for j in range(len(arr))]
    for i in range(0,len(arr)):
        for j in range(sumNeg,sumPos+1):
            if i == 0:
                dpTable[i][j-sumNeg] = (arr[i] == j)
            elif (sumNeg <= j - arr[i] and j - arr[i] <= sumPos):
                dpTable[i][j-sumNeg] = (arr[i] == j) or (dpTable[i-1][j - sumNeg]) or (dpTable[i-1][j - sumNeg - arr[i]])
                if (j == 0) and dpTable[i][j-sumNeg]:
                    return True
            else :
                dpTable[i][j - sumNeg] = (arr[i] == j) or (dpTable[i-1][j - sumNeg])
                if (j == 0) and dpTable[i][j-sumNeg]:
                    return True
    return False
```

The running time of this algorithm in worst case is O((max-min)*N)

4) This algorithms using a table with dynamic programming paradigm,the table shows the cost for each position.The approach is below:

On each iteration algorithm makes a choice according to calculations.Find the maximum of (the giving gap in the first sequence+gap penaly), (the giving gap in the second

sequence+gap penaly) or (staying with that position + (match or mismatch point whether equal or not))

```
def sequence_align(x, y,gap,match,mismatch):
    A = [[None]*(len(y)) for i in range(len(x))]
    maxValue = -999999
    A[0][0] = 0
    for i in range(1,len(x)):
        A[i][0] = i * gap
    for j in range(1,len(y)):
        A[0][j] = j * gap
    for i in range(1, len(x)):
        for j in range(1, len(y)):
            if x[i-1] == y[j-1]:
                temp = match
            else:
                temp = mismatch
            A[i][j] = max(A[i][j-1] + gap,A[i-1][j] + gap,A[i-1][j-1] + temp)
            if A[i][j] >= maxValue:
                maxValue = A[i][j]
    return maxValue
```

In the worst case,the running time of this algorithm is O(N,M)

If the length of the sequences x and y are N,M respectively.

5) In this problem approach is below:

In order to minimize the number of operation in an ancient computer,the calculation should have been done by ascending order.So the greedy algorithm find minimum each time and delete it after calculation.By this way,operation number is reduced.

```
def minOperation(arr):
    operation = [0]
    opNum = 0
    while (len(arr)>0):
        temp = findAndDeleteMin(arr)
        operation.insert(len(operation),operation[len(operation)-1]+temp)
    for i in range(0,len(operation)):
        opNum = opNum + operation[i]
    return [opNum,operation[len(operation)-1]]
def findAndDeleteMin(arr):
    min = [0,arr[0]]
    for i in range(1,len(arr)):
        if arr[i]< min[1] :
            min = [i,arr[i]]
    arr.pop(min[0])
    return (min[1])
```

If finding the maximum in list is considered to have O(n) complexity then the running time of the algorithm is O(n^2) in the worst case.