CSE 321 HW5
REPORT

1.) In this question we're asked to design minimum cost business plan. To do this task, we need to use dynamic programming. The concept of dynamic programming says that you don't need to calculate previous calculated subproblems. With this in mind, we can calculate for example month 4 by just deciding which city has minimum cost for that month and adding to previous cost, since we already have solution for if we have 3 months. We have DP table for n month with n+1 element. Since there is no month0 for 0th index we have 0 cost and then for month 1 we have this formula, and also for all months we have this formula again.

$$F[j] = F[j-1] + min(NY[j], SF[j])\ with\ moving\ cost$$

This formula says that, I know the solution for n-1 month. So just find minimum costly business office with cost if it's minimum and add to the solution. This gives you the solution for that particular month. And also store that value for next solutions. Also, since we have cost for moving office, we need to consider move cost when selecting minimum cost city. First, I select minimum city by looking costs of cities with considering moving cost. After that I stored selected city. For NY I stored 0 and for SF I stored 1. If we select location NY, we select last city as 0. If we select SF for location, we select last city as 1. After all, when algorithm works, it calculates the minimum cost for n month job schedule at the nth index of DP table with considering moving cost. We look cities like if we move there will be moving cost plus cost of city or we stay at same city and just add cost of that month and city. Minimum of them gives us the optimal solution for nth month on dp table's nth index.

```
F = []
for i in range(len(ny)):
    F.append(0)
F[1] = min(ny[1],sf[1])
print("Sequence of business plan:",end=" ")
if(F[1] == ny[1]):
    lastCity = 0
else:
    lastCity = 1
for j in range(2,len(ny)):
    if(lastCity == 0):
        print("NY", end=" ")
        op1 = ny[j]
        op2 = sf[j]+M
    else:
        op1 = ny[j]+M
        op2 = sf[j]
        print("SF", end=" ")
    F[j] = F[j-1] + min(op1,op2)
    if(min(op1,op2) == op1):
        lastCity = 0
    else:
        lastCity = 1
if(lastCity == 0):
    print("NY", end=" ")
else:
    print("SF", end=" ")
return F[n]
```

$\theta(n)$

$\theta(n)$

Complexity of algorithm is $\theta(n)$. First, we start selecting minimum costly city. After that, we select cities in order, by looking their individual costs and what if move another city by looking currently selected city. Minimum of these options gives us the minimum cost for that month and we store that solution for previous months.

2.) In this algorithm, we're asked to write an algorithm that finds maximum number of session that a student can attend. Since the student can't leave session until the session ends, we need to consider finish time of sessions. If we sort sessions with finish time, we can select the earliest finish time, in every check and we ignore that conflict with this interval. Also while selecting earliest session, we need to consider finish time of the session that the student lastly attend. With this, without any conflict we can attend maximum number of session and increase throughput.

In algorithm we have sessions dictionary. Sessions are unordered. Firstly, we sort all session with respect to finish times. After that we put -1 to lastTime variable since our time schedule start from 0 and we need to select one session at beginning. After selecting the session has the earliest finish time, we put it to optimal list and we store its finish time. This finish time help us to find next session start time. With this finish time we continue to traverse, session dictionary, since we sort these sessions with finish time, we only consider start times and if the start time of the currently inspected session is appropriate for our lastTime, we choose that session and process go on like that. Sort method of Python has $\theta(n \log n)$ complexity. Printings also has $\theta(n)$ complexity since it traverses dictionary and dictionary has element number that equal to the session count. The last for loop is the loop that selects sessions. This loop executes the algorithm. This loop also traverses the sessions and cost of loop will be again $\theta(n)$. In total we have $\theta(n \log n) + \theta(n) + \theta(n) + \theta(n) + \theta(n)$ and this will end up with total execution time of the algorithm which is $\theta(n \log n)$.

3.) Let's say we have a sequence of numbers in increasing order and we're trying find the nonempty subset that sum of elements is equal to 0. And if we define a Boolean function $Q(i, s)$ which implies that there is a nonempty subset of $x_{1,\ldots\ldots\ldots\ldots} x_i$ and the sum of subset is equal to the s. For our problem we have for that function $Q(n, 0)$ n is the last index of that sequence. Let's also we have A and B respectively sum of positive and sum of negative numbers of that sequence. For our DP table we need store for every index's trueness or falseness for that equality of given sum. Our DP table is 2D array. Indexes for array from $0 \leq i \leq lenght\ of\ whole\ set$ and for columns we have the array indexes from $A \leq s \leq B$. Initially, we fill first row with the value of Boolean $x_1 == s$. S is the target sum is here and we check that set that include first element is reaching target sum or not. After that we have main case that will give us whole solution. If current sum index for dp table after subtracting current element of array is between $A \leq s \leq B$ we can say that for current dp table index can be these things:

   a. Current element of sequence can be equal to target sum.
   b. Previous row index can be true and this adding may not effect total sum.
   c. After subtracting current sequence element, we can also have target sum still.

   For other case we have again case a and case b.
   We can think algorithm just like knapsack problem. Weight of values is 1. We decide that can we create this sum with these elements or not. Just like knapsnack but instead of values we say true or false. After that we inspect the the dp table to find subset. We start last element of dp table and decide that is the previously indexed element is true for current target sum. If it not, we take current element to subset and we decrease target sum by that element to find other sets.

```
S = [-1, 6, 4, 2, 3, -7, -5]
S = sorted(S)
neg = 0
pos = 0
target = 0
for i in range(len(S)):
    if(S[i] > 0):
        pos += S[i]
    elif(S[i] < 0):
        neg += S[i]

dp = [[True for i in range(pos-neg+1)] for j in range(len(S))]

for i in range(len(S)):
    for j in range(neg, pos+1):
        if(i == 0):
            dp[i][j-neg] = (S[i] == j)
        elif(neg <= j - S[i] and j-S[i] <= pos):
            dp[i][j-neg] = (S[i] == j) or dp[i-1][j-neg] or dp[i-1][j-neg-S[i]]
        else:
            dp[i][j-neg] = (S[i] == j) or dp[i-1][j-neg]

subset = []

for i in range(len(S)-1, -1, -1):
    if(S[i] == target):
        subset.append(S[i])
    elif(not(dp[i-1][target-neg])):
        subset.append(S[i])
        target = target-S[i]
```


```
for i in range(len(dp)):
    if(dp[i][-neg] == True):
        isEqual = True
```

This segment looks index of 0 in dp table and decides is there a sum that equals to zero or not. If sum exist, prints sum to terminal. Otherwise prints no sum.

Complexity of algorithm changes since it's a NP problem. Complexity of sort is $\theta(n \log n)$. Complexity of other loop's is $\theta(n)$ but for dp table creator loop we have pseudo-polynomial time. Which means we don't know complexity actually since B-A not polynomial. In total we have the $O(n(B-A))$ complexity. If we say there all values are $O(n^k)$ for some k, then all required time is $O(n^{k+2})$.

4.) For that particular problem I used Needleman-Wunsch algorithm. First, we need to create array for storing results of previous solution of problem (i.e. DP table). If we go over an example. Let's think words ALGORITHM and ALGORITMA.

		A	L	G	O	R	I	T	H	M
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18
A	−2	None	None	None	None	None	None	None	None	None
L	−4	None	None	None	None	None	None	None	None	None
G	−6	None	None	None	None	None	None	None	None	None
O	−8	None	None	None	None	None	None	None	None	None
R	−10	None	None	None	None	None	None	None	None	None
I	−12	None	None	None	None	None	None	None	None	None
T	−14	None	None	None	None	None	None	None	None	None
M	−16	None	None	None	None	None	None	None	None	None
A	−18	None	None	None	None	None	None	None	None	None

At first, we fill table like that. So, for 0 word we all have that gap and we fill for gap score. We start from second cell and second column and move row by row and at each cell we calculate score by comparing results of top, left or top-left cell cell and we add right score to previous scores.
Recursive formula for DP algorithm is:

$$F[i][j] = \max\left(F[i-1][j-1] + score\ of\ match\ or\ mismatch, F[i][j-1] + gapScore + F[i-1][j] + gapScore\right)$$

Also, we have the initial conditions that

$$F[0][j] = gapScore * j$$
$$F[i][j] = gapScore * i$$

The first table above comes from that initial conditions.
After that filling table with this recursive dp table formula, last cell of the table will give us the exact score for that matching.

Complexity of algorithm is $O(mn)$. M and n is lengths of strings in here. If we look code below there is nested for loop. One of them runs for first string and the other one runs for second string. Since their individual complexities $O(m)\ and\ O(n)$ the total complexity of algorithm will be multiple of them and that is $O(mn)$. Other calculations don't affect the total complexity.

To find exact alignment we need to do backtracking. This is reverse process starts from last cell of DP table and redoing summations again and if this is the case of that summation, it puts words in alignment string. If there is need a gap, it puts "-" for gap. This code second code segment of next page.

```python
s1 = "ALGORITHM"
s2 = "ALGORITMA"

DP_ROW_N = len(s1)+1
DP_COLUMN_N = len(s2)+1
match = 2
mismatch = -2
gap = -1

dp = [[None for y in range(DP_COLUMN_N)] for x in range(DP_ROW_N)]

for i in range(DP_ROW_N):
    dp[i][0] = mismatch*i
for j in range(DP_COLUMN_N):
    dp[0][j] = mismatch*j

for i in range(len(dp)):
    for j in range(len(dp[0])):
        print("%5s"%(dp[i][j]), end=" ")
    print()

for i in range(1,DP_ROW_N):
    for j in range(1,DP_COLUMN_N):
        score = (match if (s1[i-1] == s2[j-1]) else mismatch)
        h = dp[i][j-1] + gap
        d = dp[i-1][j-1] + score
        v = dp[i-1][j] + gap
        dp[i][j] = max(h,d,v)
```

$\theta(mn)$

$\theta(m) + \theta(n)$

$\theta(mn)$

```python
while (i > 0 or j > 0):
    if (i > 0 and j > 0 and dp[i][j] == dp[i-1][j-1] + (match if (s1[i-1] == s2[j-1]) else mismatch)):
        a1 = s1[i-1] + a1
        a2 = s2[j-1] + a2
        i = i - 1
        j = j - 1
    elif (i > 0 and dp[i][j] == dp[i-1][j] + gap):
        a1 = s1[i-1] + a1
        a2 = "-" + a2
        i = i - 1
    else:
        a1 = "-" + a1
        a2 = s2[j-1] + a2
        j = j - 1
```

$O(n+m)$

5.) For that algorithm I think two approaches. Both approaches start with sorting array. This is takes in python sort function $\theta(n \log n)$. After that my first approach is that:

a. Start one index from start of list and start one index from end of list. Sum indexed element and add sum operation cost to total cost and also add sum to total sum.

i. For example for sorted 1,2,3,4 first we compute $1 + 4 = 5$ and then $2 + 3 = 5$ in total now we have 10 operation and if we sum this to sum we end up with total sum 10 and total operation 5+5 and last one will be 10 operation from $5 + 5, 5 + 5 + 10 = 20$ operation.

ii. For that array second approach is firstly sum first two element of array. This will be 1+2 = 3 and 3 operation in total. After that sum all elements in order and add sum to total operation and total sum. If we continue

1. $1 + 2 = 3\ Total\ sum = 3\ Total\ operation = 3$
2. $3 + 3 = 6\ Total\ sum = 6\ Total\ operation = 6 + 3 = 9$
3. $6 + 4 = 10\ Total\ sum = 10\ Total\ Operation = 19$

This approach has a smaller number of operation and will be the optimal solution for this problem.

```
arr = sorted(arr)                    θ(n log n)

sum = 0
operation = 0

if(len(arr) < 2):
    print("No element to sum")
else:
    sum = arr[0]+arr[1]
    operation += sum

    for j in range(2,len(arr)):
        sum += arr[j]                          θ(n)
        operation += sum
    print("Total sum:", end=" ")
    print(sum)
    print("Total operation:", end=" ")
    print(operation)
```

Complexity of algorithm is for sorting $\theta(n \log n)$ and for finding total operation $\theta(n)$. In total we have $\theta(n \log n)$.