Ahmed Semih Ozmekik

Question 1

1. Given a value for the moving cost M and sequences of operating costs N_1 ... N_n and S_1 ... S_n , find a plan of minimum cost, which is being the optimal cost. We need a dynamic programming algorithm for that problem, here is the algorithm:

```
def optimal_cost(n, M, ny, sf):
    min_cost = [[0] * n, [0] * n]
    min_cost[0][0] = ny[0]
    min_cost[1][0] = sf[0]

    for i in range(1, n):
        min_cost[0][i] = min(min_cost[0][i - 1] + ny[i], min_cost[1][i - 1]
            + sf[i] + M)
        min_cost[1][i] = min(min_cost[0][i - 1] + ny[i] + M, min_cost[1][i
            - 1] + sf[i])

    return min(min_cost[0][n-1], min_cost[1][n-1])
```

$mincost[0][i]$ means minimum cost in New York, while $mincost[1][i]$ means minimum cost in San Francisco. We have constructed our matrix, and we take the minimum of these to get the optimal cost.

In our algorithm, we have a main for loop which is run n times. Inside it, we have constant time operations, so in worst case we will run $\mathcal{O}(n)$ times.

Hence, $T(n) \in \mathcal{O}(n)$.

Question 2

1. Supposing we'd like to attend a symposium which has many simultaneous sessions. The lengths of the sessions are not fixed and they begin at various times. We want to join as much sessions as possible. For that we need to design a greedy algorithm that finds the optimal list of sessions with the maximum number of sessions. Remembering that we can be at only one session at the same time and we cannot leave any session before it is over. Here is implementation for this algorithm design:

```
def optimal_list_schedule(S):
    S = sorted(S, key=lambda x: x[1], reverse=False)
```

```
    optimal = []
    joined = -1
    for s in S:
        if joined == -1 or optimal[joined][1] <= s[0]:
            optimal.append(s)
            joined += 1
    return optimal
```

This problem is a general problem called interval scheduling problem. As we discussed this in our class, the simple, greedy approach to solve this problem is taking the class which finishes earlier. In our implementation choice, I have considered our input to be like this: (0_{th} index being the start time and 1_{th} index being the end time of the class.)

```
def driver_optimal_list_schedule():
    schedule = [[1, 3], [11, 1], [12, 3], [12, 2], [1, 4]]
```

2. **Worst Case**: We considered the finish time in this algorithm. And in our basic operation, it will run n times, for n is the number of elements in S. With sorting, it would make the complexity $\mathcal{O}(nlogn)$. Also we have $\mathcal{O}(n)$ and in total it is $\mathcal{O}(nlogn + n)$.

   Hence, $T(n) \in \mathcal{O}(nlogn)$.

Question 3

1. Solution kept in the end of document for scaling and alignment purposes.

Question 4

1. We have given cost equation with some scores. Regarding to the relations between two given strings, we need to find this scores and calculate the scores; for different gaps and alignment. After that, we need to find the specific alignment which has the max score. Here is our algorithm implementation for this problem:

```
def align_score(A, B):
    # scores are hardcoded, could be changed.
    match, miss, gap = 2, -2, -1

    n = len(A) + 1
    m = len(B) + 1
    F = ([[0 for i in range(n)] for i in range(m)])
```

```
    for i in range(m):
        F[i][0] = gap * i
    for i in range(n):
        F[0][i] = gap * i

    for i in range(1, m):
        for j in range(1, n):
            score = miss
            if B[i - 1] == A[j - 1]:
                score = match
            no_gap_score = F[i - 1][j - 1] + score
            F[i][j] = max(no_gap_score, F[i - 1][j] + gap, F[i][j - 1] +
                gap)
    return max(*[F[i][n - 1] for i in range(m)], *[F[m - 1][i] for i in
        range(n)])
```

In this dynamic programming algorithm approach, we consider $seq1$ to be the long string. If not we swap them. $score[i, j]$ means the cost of aligning $seq1[i]$ to $seq2[j]$. Our max score will be $max(score[i][n-1])$. Our basic operation is, for loop. m and n being the length of the sequences, our complexity becomes $O(m.n)$ because of the foor loop.

Hence, $T(n) \in \mathcal{O}(m.n)$

Question 5

1. Supposing we have an ancient computer system that needs to do A + B operation to add numbers A and B. For example, the computer does 13 operation to add 5 and 8. We are given an array of integers to calculate the sum of its elements. Regarding to this definitions, we will design a greedy algorithm to calculate the sum of the array with the minimum number of operations. Here is the implementation:

```
def ancient_sum(arr):
    op = 0
    n = len(arr)
    while n > 1:
        for i in range(0, n, 2):
            arr[i] += arr[i + 1]
            op += arr[i]
        arr = [arr[i] for i in range(n) if i % 2 == 0]
        n = len(arr)
```

```
    print(op, 'number of operation.')
    return arr[0]
```

For the definition, we need to perform addition operation between very small numbers. We can't achieve this by adding the numbers straightforward. We will sum them up by pairs. This approach is kind of similar to divide and conquer style. We add the A[0] and A[1] and put to A[0]. Similarly, we add A[2] and A[3] and put to A[2]. And then, sum them up too. This continues like this and constructs the idea of tree which has it's root appears in the final step.

2. **Worst Case**:

$$T(n) = \frac{n}{2} + \frac{n}{4} + ... + 1 \tag{1}$$

$$T(n) = \sum_{i=1}^{logn} \frac{n}{2^i} \tag{2}$$

$$T(n) = n \times \sum_{i=1}^{logn} 2^{-i} \tag{3}$$

$$T(n) = n \times (2 - \frac{2}{n}) \tag{4}$$

$$T(n) = 2n - 2 \tag{5}$$

$$T(n) = n \tag{6}$$

$$\tag{7}$$

Hence, $T(n) \in \mathcal{O}(n)$.

Question 3

We are given a set of integers and we will design a dynamic programming algorithm to check whether there is a subset with the total sum of elements equal to zero. If the algorithm finds such a subset, then we print the elements of that subset and terminate the algorithm.
dp or in our case $F(i, j)$ will be such function that elements from 0 to i will return the the *sum* j. Here is the algorithm:

Algorithm 1 Finding the subset having zero sum with DP.

```
1:  procedure zeroSum(S)
2:      S ← [i − min(S) for i in S ]
3:      m ← sum(S)
4:      n ← len(S)
5:      F ← [[False] ∗ m for i in range(0,n) ]
6:      p1 ← [[(0, 0) ∗ n] for i in range(0,m) ]
7:      p2 ← [[0] ∗ m for i in range(0,n) ]
8:      i from 0 to n:
9:          F[i][0] ← True
10:         F[0][S[0]] ← True
11:     i from 0 to m:
12:         p1[0][i] ← (−1, i)
13:     F[i][0] ← [True for i in range(0,n)]
14:     F[0][S[0]] ← True
15:     ...
```

Algorithm 2 'cont.

```
1: procedure zeroSum(S)
2:     ...
3:     i from 1 to n:
4:         j from 0 to m:
5:             if j < S[i] then
6:                 F[i][j] ← F[i − 1][j]
7:                 if F[i][j]] then
8:                     p1[i][j] ← (i − 1, j)
9:                     p2[i][j] ← p2[i − 1][j]
10:                else then
11:                    p1 ← (−1, j)
12:                    p2[i][j] ← 0
13:            else then
14:                F[i][j] ← F[i − 1][j] + F[i − 1][j − S[i]]
15:            if F[i − 1][j] and S[i]! = 0 then
16:                p1[i][j] = (i − 1, j)
17:                p2[i][j] = p2[i − 1][j]
18:            else if F[i − 1][j − S[i]] then
19:                p1[i][j] ← (i − 1, j − S[i])
20:                p2[i][j] ← p2[i − 1][j − S[i]] + 1
21:                if F[i][j] and j! = 0 then and p2[i][j] ∗ min(S) = j
22:                    a, b ← i, j
23:                    p3 ← p2[a][b]
24:                    p3_index ← a
25:                    while a! = −1 then
26:                        if p2[a][b]! = p3 then
27:                            p3_index = a
28:                            p3 = p2[a][b]
29:                            a, b = p1[a][b] return
30:            else then
31:                p1[i][j] ← (−1, j)
32:                p2[i][j] ← 0
33:            end if
34:        end loop
35:    end loop
36: end procedure =0
```

We will the dynamic programming table. In that computation, nested loops run $m.n$ times.

Hence, $T(n) \in \mathcal{O}(m.n)$.