CSE 321 HOMEWORK 5

FURKAN OZEV

161044036

1) Find Optimal Plan and Cost (Dynamic Programming):

ALGORITHM:

- function optimalplan (NY, SF, M):
    - optNY[0:n] <- 0
    - optSF[0:n] <- 0
    - n <- length of NY or SF
    - for i = 1,…,n:
        - optNY[i] <- NY[i-1] + min(optNY[i-1], (M + optSF[i-1])
        - optSF[i] <- SF[i-1] + min(optSF[i-1], (M + optNY[i-1])
    - end for
    - Return the smaller of optNY(n) and optSF(n)

ANALYZE COMPLEXITY:

1- Creation two lists and initiliaze take O(1) time.
2- Calculate length of array take O(1) time:
3- The loop has n iterations: O(n)
   a. Comparison in min function takes O(1) time.
   b. Addition and initialization takes O(1) time.
4- Comparison and return operation take O(1) time.

Worst Case:

T(n) = O(1) + O(1) + O(n) + O(1)

T(n) = O(n)

- The algorithm has n iterations, and each takes constant time.
- Thus the running time is O(n)

O(n)

EXPLANATION ALGORITHM:

- The basic observation is that the optimal plan ends in either NY or SF.
- optNY(j) denotes the minimum cost of a plan on months 1,..,j ending in NY
- optSF(j) denotes the minimum cost of a plan on months 1,..,j ending in SF
- If it ends in NY, it will pay expense in the nth month plus one of the following two amounts:
    - The cost of the optimal plan on n-1 months, ending in NY, or
    - The cost of the optimal plan on n-1 months, ending in SF, plus a moving cost of M
    - optNY(n) = NY(n) + min(optN(n-1), M+optSF(n-1))
- A similar sitiuation applies, if the optimal plan ends in SF.
    - optSF(n) = SF(n) + min(optSF(n-1), M+optNY(n-1))
- It is preferred that the cost of the last finished city is less.

2) Find Optimal List of Sessions (Greedy Algorithm):

ALGORITHM:

- function optimalSession (begins, lengths):
  - Calculate the finish times of the sessions in the given session set with using begins and length times.
  - Sort sessions by finish time.
  - Keep indexes of sessions in indexlist before sorting.
  - Call helper function with sorted sessions' begin and finish times.
  - Then, return function results.

- function helper(sbegins, sfinishes, indexlist):
  - Calculate session amount
  - Create session list to keep optimal sessions
  - Append first element of sorted session list
  - i <- 0
  - For j = 0,…,n-1:
    - If the start time of the next element in the sorted session list is after the end time of the current session:
      - This session is added to the optimal session list
      - i <- j
  - Return session list

ANALYZE COMPLEXITY:

1- Calculation the finish times of the session takes O(n) time.
2- Sorting sessions by finish time of sessions and Keeping indexes of sessions takes O(n) time.
3- helper function takes f(n) time. O(n)
   a. Calculation session list length take O(1) time.
   b. Creation list take O(1) time.
   c. Append element in list operation take O(1) time.
   d. The loop has n iteration: O(n)
      i. Comparison, append, assignment operation take O(1) time
      ii. So loop take O(n).
   e. f(n) = O(1) + O(1) + O(1) + O(n) = O(n)
4- Return session list take O(1) time.

Worst Case:

$T(n) = O(n) + O(n) + O(n) + O(1)$
$T(n) = O(n)$

O(n)

EXPLANATION ALGORITHM:

- The list of given sessions is sorted so that the finish time is at the earliest.
- Greedy selection is always to choose the first ending session.
- First session always provides one of the optimal solutions.
- Because the first session is the shortest session.
- The main purpose: End the session earlier and join more sessions.

- After the current session is finish, it enters the session that has not yet started and has the earliest finish time.

3) Subset with Total Sum of Elements Equal to Zero (Dynamic Programming):

ALGORITHM:

- function part3 (arr):
    - Create 2 2D list and initialize element with 0. (First list to store variable, Other list to states of value.)
    - Call helper function to find possible subset with arr, some star value and some start list.
    - If result of this fuction is a list:
        - Print this list
    - Else:
        - Print message about there is no subset

- function helper(i=0, sum=0, arr, arr2=[], n, dp, visit):
    - If i equal n: (Base Case)
        - If sum of current subset (sum) equal 0:
            - If length of current subset (arr2) not equal 0:
                - Return current subset (arr2)
            - Else:
                - Return 1
        - Else:
            - Return 0
    - If a state is already solved:
        - Return the value
    - Change state with already sorted.
    - Copy current subset, and append next item in arr.
    - There is a 2 possible situation:
        - Next subset contains item in current index of arr
        - Or, Next subset does not contain item in current index of arr
    - Recursively call this function for first situation.
    - If first situation result is a list:
        - Return result
    - Recursively call this function for second situation.
    - If secont situation result is a list:
        - Return result
    - Sum first and second result for, then assign array.
    - Return value

ANALYZE COMPLEXITY:

1- Creation and initialization array takes O(1) time.
2- helper function takes f(n) time. O(2^n)
   a. Comparison and return operations take O(1) time.
   b. Assignment operations take O(1) time.
   c. Copy list operation take O(n) time.
   d. Append element in list operation take O(1) time.

e. There are 2 recursive calling with i+1 (like size-1)

f. Sum and return operation take O(1) time

F(n) = O(1) + O(1) + O(n) + O(1) + 2*F(n-1)

F(n) = 2*F(n-1) + O(n)

$$
\begin{aligned}
T(n) &= 2T(n) + n \quad \rightarrow (1)\\
&= 2^2T(n-2) + 2(n-1) + n\\
&= 2^3T(n-3) + 2^2(n-2) + 2(n-1) + n\\
&.\\
&.\\
&.\\
&= 2^{n-1}T(n-(n-1)) + 2^{n-2}(n-(n-2)) + 2^{n-3}(n-(n-3)) + \cdots + 2(n-1)\\
&+ n\\
&= 2^{n-1}T(1) + 2^{n-2}.2 + 2^{n-3}.3 + \cdots + 2(n-1) + n
\end{aligned}
$$

Now multiply $T(n)$ By 2

$$2T(n) = 2^n + 2^{n-1}.2 + 2^{n-2}.3 + \cdots + 2^2(n-1) + 2n \rightarrow (2)$$

Now $(2) - (1) \implies$

$$
\begin{aligned}
T(n) &= 2^n + 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^2 + 2 - n\\
&= 2^n + 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^2 + 2 - n\\
&= 2.\frac{(2^n-1)}{(2-1)} \text{ (Sum to n terms of GP with a = r = 2) } - n\\
&= 2^{n+1} - 2 - n\\
&= \Theta(2^n)
\end{aligned}
$$

F(n) = O(2^n)

3- Printing message takes O(1) time.

Worst Case:

T(n) = O(1) + O(2^n) + O(1)

T(n) = O(2^n)

O(2^n)

EXPLANATION ALGORITHM:

- Recursively check sum of all possible subsets until sum of elements of current subset equal 0.
- While doing this, the basic logic is to include the current element in the subset sum or not.
- To better explain:
- Let's suppose sum of all the items we have selected up to index 'i-1' is 'S'.
- So, starting with "i", we need to find the subarray {i, N-1}, where the sum is equal to S.
- Let's define dp[i][S].
- It means number of the subset of the subarray{i, N-1} of 'arr' with sum equals '-S'.
- If we are at i th index, we have two choices:
  - Include it in the sum
  - Or dont include in the sum

4) Find Alignment Between Two Strings With Minimum Cost (Dynamic Programming):

ALGORITHM:

- function alignment (str1, str2, match, mismatch, gap):
  - Calculate length of strings
  - Create 2D array to store optimal subsrtucture then, initliaze with 0.
  - Initialize the 2D array
  - for i = 0,…,(len1+len2):
    - arr[i][0] <- i * gap
    - arr[0][i] <- i * gap
  - Calculating the minimum penalty:

- for i = 1,…,len1:
  - for j = 1,…,len2:
    - If the letters of the strings in this index match:
      - In 2D array, increase the value of this position with match score.
    - Else:
      - Find maximum mismatch score: (There are 3 situation)
        - Assume 2 letter are mismatch.
        - Assume first string has a gap.
        - Assume second string has a gap
      - In 2D array, increase the value of this position with calculated max score.
- Reconstruct the solution
- l <- len1 + len2
- i <- len1, j <- n
- xpos <- l, ypos <- l
- Create xans[l+1], yans[l+1]
- Determine final answer for the respective strings
- While i not equal 0 and j not equal 0:
  - If x[i-1] = y[j-1]:
    - xans[xpos] <- str1[i-1]
    - yans[ypos] <- str2[j-1]
    - i--, j--, xpos--, ypos--
  - Else if arr[i-1][j-1] + mismatch = D[i][j]:
    - xans[xpos] <- str1[i-1]
    - yans[ypos] <- str2[j-1]
    - i--, j--, xpos--, ypos--
  - Else if arr[i-1][j] + gap = D[i][j]:
    - xans[xpos] <- str1[i-1]
    - yans[ypos] <- "_"
    - i--, xpos--, ypos--
  - Else if arr[i][j-1] + gap = D[i][j]:
    - xans[xpos] <- "_"
    - yans[ypos] <- str2[j - 1]
    - j--, xpos--, ypos--
- While xpos > 0:
  - İf i > 0:
    - i <- i – 1
    - xans[xpos] <- str1[i]
    - xpos <- xpos -1
  - Else:
    - xans[xpos} <- "_"
    - xpos <- xpos – 1
- While ypos > 0:
  - İf j > 0:
    - j <- j – 1
    - yans[ypos] <- str2[j]

            - ypos <- ypos -1
        - Else:
            - yans[ypos} <- "_"
            - ypos <- ypos - 1
    - Remove the extra gaps in the starting id represents the index from which the arrays xans, yans are useful.
    - i <- l
    - while i >= 1:
        - if yans[i] == "_" and xans[i] == "_":
            - id = i + 1
            - break
        - i <- i – 1
    - i < i + 1
    - result of sequence1 is xans[i:]
    - result of sequence2 is yans[i:]
    - return (arr[len1][len2], res1, res2)

ANALYZE COMPLEXITY:

1- Assume length of sequence1 is n and length of sequence2 is m.
2- Creation array takes O(1) time.
3- First for loop for gap initialization has (n+m) iteration. Each iterations take constant time. So it takes O(n+m) time.
4- Second for loop for initialization has (n+m) iteration. Each iterations take constant time. So it takes O(n+m) time.
5- Assignment operations take O(1) time.
6- Creation xans yans list and initialization has (n+m) iteration. So it takes O(n+m) time.
7- First while loop for reconstructing has maximum (n+m) iteration. Each iterations take constant time. So it takes O(n+m) time.
8- Second while loop has maximum n iteration. Each iterations take constant time. So it takes O(n) time.
9- Third while loop has maximum m iteration. Each iterations take constant time. So it takes O(m) time.
10- First while loop for removing gap has (n+m) iteration. Each iterations take constant time. So it takes O(n+m) time.
11- Return takes O(1) time.

Worst Case:

T(n) = O(1)+O(n+m) + O(n+m) +O(1)+ O(n+m) + O(n+m) + O(n) + O(m) + O(n+m) + O(1)
O(n+m) > O(n) and O(n+m) > O(n) and O(n+m) > O(1)
So, T(n) = O(n+m) for worst case
T(n) = O(n+m)
O(n+m)

EXPLANATION ALGORITHM:

- Given as an input two strings, X = x1, x2,...,xn, and Y = y1,y2,...,ym, output the alignment of the strings, character by character, so that the net penalty is minimised.
- A reward is given to match the X and Y characters.
- A gap penalty occurs if a gap is inserted between the string.
- A mismatch penalty occurs if the X and Y characters match incorrectly.

- The applicable solution is to insert gaps in the strings to equalize the lengths.
- It can easily be proved that the addition of extra gaps after the lengths have been equalized will only result in increased penalties.
- It can be seen that the optimal solution narrows with only three candidates.
    - xn and ym
    - xn and gap
    - gap and ym
- To Reconstruct
- Trace back through the filled table, starting arr[n][m].
- When(i,j):
    - If was filled in using state 1, go to: (i-1, j-1)
    - If was filled in using state 2, go to: (i-1, j)
    - If was filled in using state 3, go to: (i, j-1)
- If i = 0 or j = 0, match the remaining substring with spaces.

5) Minimum Number of Operations (Greedy Algorithm):

ALGORITHM:

- function minOperation (arr):
    - opCount <- 0
    - Until 1 item remains in the list:
        - Select the 2 minimum elements in list.
        - Delete these elements from the list.
        - Sum these 2 numbers and add this result to the list.
        - Increase the opCount by the sum of these 2 numbers. (opCount += sum)
    - Return sum and opCount.

ANALYZE COMPLEXITY:

- Assume n equal length of arr
- Initialize opCount takes O(1) time.
- Loop has (n-1) iteration: O(n^2)
    - Getting minimum element in list takes O(n) time.
    - Removing element in list takes O(1) time.
    - Sum and append operations take O(1) time.

    F(n) = O(n) + O(1) + O(1) = O(n)

    Each iteration takes O(n) time

    So, Loop takes O(n^2) time

- Return operation takes O(1) time

  Worst Case:

  T(n) = O(1) + O(n^2) + O(1)
  T(n) = O(n^2)

  O(n^2)

EXPLANATION ALGORITHM:

a. We know that to sum two numbers, requires operations as much as the sum of two numbers.
b. Greedy selection is always to choose the smallest elements.
c. So we have to sum the smallest numbers as possible.
d. Until 1 element remains in the list (ie result sum of array):
    - We take the smallest 2 numbers and sum it.
    - The result is added to the list.
    - Reason for adding the result to the list: The result may be less than any number in the list.
    - It can be one of the minimum numbers for the next operation. Or not.
e. So, we solve the problem by doing fewer operations than summing the numbers one by one.