ESO207A : Data Structures and Algorithms
Assignment 4 Solutions

Team Name : Invariantly_Yours

Yatharth Goswami: 191178
Sarthak Rout : 190772
Devanshu Singla : 190274

January 11, 2021

I Problem 1 - Part (c)

Algorithm 1: PseudoCode - Bottom - Up Dynamic Programming Algorithm

Input: A list of words L and a number M

Output: 'Neat' printout of words and minimum cost

```
1  L -> List of words
2  n -> Length of L
3  l -> Array containing lengths of corresponding words from L
4  cost -> Array containing cost of printing neatly words from current index to n
5  last -> Array storing the position of the last word which should appear on the first line of
      the optimal solution of L_k to L_n.
6  Function Bottom-Up(L):
7      for i = n down to 1 do
8          if \sum_{k=i}^{n} l_k + (n - k) < M then
9              cost[i] = 0
10             continue
11         cost_min = \infty
12         for j = 1 to min(n-i, m) do
13             temp = M - (\sum_{k=1}^{j} l_{i+k} + j - 1)
14             if temp >= 0 and temp*temp*temp + cost[i+j+1] < cost_min then
15                 cost_min = temp * temp * temp + cost[i + j + 1]
16                 last[i] = i + j
17         cost[i] = cost_min
18 Function PrintNeatly(last, cost):
19     end = last[0]
20     prev = 0
21     while prev != n do
22         for i = prev+1 to end do
23             Print L[i]
24         prev = end
25         end = last[end + 1]
26     Print cost[0]
```

Time Complexity for algorithm 1

The function Bottom-Up represents the main algorithm, so will analyse this function only. In function Bottom-Up, lines 8-17 execute n times. The condition in line 8 takes at most $(n-i)c_1$ time, for some c_1, to execute. If the condition is true the statements 9, 10 are executed taking constant time say c_2. Otherwise, statements 11-17 are executed. Statements 11 and 17 take constant time say c_3. Statements 12-16 run $\min(m, n-i)$ times taking time of at most $\min(m, n-i)c_4$. Hence, one loop executes in either at most $(n-i)c_1 + c_2$ time or at most $(n-i)c_1 + c_3 + \min(m, n-i)c_4$, which implies one loop take at most $c\min(m, n-i)$ time for some c. Therefore, if time needed to execute algorithm be T(n), then:

$$T(n) \leq \sum_{i=1}^{n} \text{c}\min(m, n-i)$$

$$\implies T(n) \leq \text{c}\min(mn, \frac{n(n-1)}{2})$$

Therefore, time complexity of algorithm is $O(n\min(m, n))$.

Space Complexity of algorithm 1

Space taken by predefined arrays like L, l, cost and last have space complexity of $O(n)$. The local variables in the function Bottom-Up occupy constant space, hence space complexity due to them is $O(1)$. So, overall space complexity = $O(n) + O(1)$ = $O(n)$.

Algorithm 2: PseudoCode - Top - Down Dynamic Programming Algorithm

```
Input: A list of words L and a number M
Output: 'Neat' printout of words and minimum cost
N = 1e9 + 7
MAX_SIZE = 100005
PrefixSums[MAX_SIZE]
DP[MAX_SIZE][2]
Function Precompute(L):
    for i in L.size do
        PrefixSums[i+1] = PrefixSums[i] + L[i]
    for i = 1 to MAX_SIZE do
        DP[i][0] = ∞
        DP[i][1] = -1
    return
Function LengthSum(a, b):
    if a == 0 then
        return PrefixSums[b]
    else
        return PrefixSums[b] - PrefixSums[a-1]
Function PrintWords(DP):
    idx = 1
    while idx < n do
        nidx = dp[idx][1]; for i =idx to nidx do
            Print L[idx-1]
        idx = nidx
    if idx <= n then
        for i = idx to nidx do
            Print L[idx-1]
Function ResolveDP(i, n):
    if i >= n then
        return 0
    else if  DP[i][0] != ∞ then
        return DP[i][0]
    else
        for k = i + 1 to n + 1 do
            z = M - LengthSum (i, k-1) - (k-1 - i)
            if z < 0 then
                break
            part_sum = ResolveDP(k, n)
            if  k == n + 1 then
                DP[i][0] = 0
                DP[i][1] = k
            else if z*z*z + part_sum < DP[i][0] then
                DP[i][0] = z*z*z + part_sum
                DP[i][1] = k
        DP[i][0] = DP[i][0]%N
        return DP[i][0]
    return
Function GetNeatWords(L):
    Precompute(L)
    minimised_quantity = ResolveDP(1, L.size)
    PrintWords(DP)
```

Time Complexity for algorithm 2

For function Precompute, the for loop in line 6-7 will execute for n times, hence requires O(n) time. Similarly, for loop in lines 8-9 executes MAX_SIZE time which is $O(n)$, hence time complexity for "for loop" is $O(n)$.

Therefore, time complexity of function Precompute $= O(n) + O(n) = O(n)$.

In function LengthSum(a, b), all lines run in constant time, which implies time complexity of LengthSum is $O(1)$.

In function ResolveDP(i, n), if $i >= n$ or DP$[i][0]$ has been changed from initial value '∞' then it executes in constant time otherwise lines 32-44 are executed. When $DP[i][0] \neq \infty$. Since initially DP$[i][0] = \infty, \forall i$, hence first call at line 48 to ResolveDP$(1, L.size)$ will execute line 32-45. While executing ResolveDP$(1, L.size)$, it will first call ResolveDP(k, n) in line 36 where $k = 2$ and $n = L.size$. This will continue to happen, in the execution of ResolveDP(i, n), ResolveDP(k, n) is executed in line 36 where $k = i+1, n = L.size$, till k becomes equal to n. Since, execution of lines 32-45, either line 38 or line 41 execute once and hence DP$[i][0]$ changes, it implies further calls to ResolveDP(i, n) will execute lines 27-30 and hence will be constant time operation. Let $T(i)$ denote the time complexity for the call ResolveDP(i, n) when DP$[j, n]$ for $i \leq j < n$ has not been changed, where $n = L.size$. During execution of first loop in ResolveDP(i, n), the time taken for statement 36 is $T(i+1)$ and rest of statements is constant time c_1. After execution of ResolveDP$(i+1, n)$, DP$[j][0]$ have been changed for $i < j < n$, hence in further execution of loops, line 36 will run in constant time. Therefore for atmost M loops(in case loop breaks at 33) or atmost (n-i-1) loops (in case for loop completes successfully without breaking), lines 33-44 execute in constant time, say upper limit be c_2. Therefore,

$$T(i) \leq T(i+1) + c_1 + \min(M, (n-i-1))c_2$$

$$\implies T(i) \leq T(i+1) + \min(M, n-i)c \text{ , for some } c \text{ independent of } i$$

$$\implies T(i) \leq T(j) + \sum_{k=i}^{j-1} \min(M, n-k)c, \text{ for some } l \leq n$$

Since execution of ResolveDP(n, n) take constant time say c_3, putting $j = n$ in above equation,

$$T(i) \leq c_3 + \sum_{k=i}^{n-1} \min(M, n-k)c$$

$$\implies T(i) \leq c_3 + \min(M(n-1), \frac{(n-i)(n-i+1)}{2})c$$

Since execution of Resolve$(1, n)$ takes $T(1) \leq c_3 + \min(M(n-1), \frac{(n-1)(n)}{2})c$ time, it implies time complexity of Resolve$(1, n)$ is $O(n \min(M, n))$.

Since the main part of algorithm is line 47 and 48, therefore the time complexity of algorithm = $O(n) + O(n\min(M, n)) = O(n\min(M, n))$, where $n = L.size$.

Space Complexity of algorithm 2

In the algorithm, MAX_SIZE refers to the maximum number of lines possible which is obviously less than n. Hence the space complexity of global variables in lines 1-4 is $O(n)$.

In functions Precompute and LengthSum , the local variables occupy constant space, hence space complexity due to them is $O(1)$.

While executing ResolveDP(1,n), where $n = L.size$, it can be seen from above analysis the maximum depth of recursion is n, hence at max n stack frames of the function ResolveDP are created and since the local variables in function ResolveDP occupy constant space, it implies space complexity for the local variables in the call ResolveDP(1,n) is $O(n)$.

So, total space complexity due to algorithm = $O(1) + O(n) + O(n) = O(n)$.

II Observations

We generated random text of 10000 words online with $M = 20$. For measuring time, we used <chrono> and <ctime> library of C++. The PC set up used had 8th gen Intel i5 processor with 8 GB of RAM.

Table 1: Table of time taken

Top-Down Approach	Bottom Up Approach
0.012998	0.035828
0.009974	0.030913
0.010972	0.041188
0.008978	0.034870
0.015990	0.040331
0.008976	0.024934
0.008973	0.026261
0.008989	0.033172
0.008978	0.032910
0.008960	0.039381

Averaging over 10 observations, the **Top-Down** approach took **0.0104** s and the **Bottom-Up** approach took **0.0339** s.
The bottom-up approach took about 3 times more time than top-down approach to solve the problem. We understand it in this way that, since, in Top-Down approach, we only calculate the relevant sub-problems once, which contribute to the dominant $O(n^2)$ term and in Bottom-Up approach, we fill the whole table for all states, it takes some more time by a constant factor of about 3.