ESO207A
Assignment 4

Rahul Sethi (190668)
Rohan Baijal (190714)
Vivek Kumar Singh (190988)

The reader may skip directly to page 6 (click here) for the time and space complexity results.

Contents

1 Sketch of the Algorithm

Borrowing notation from the question (CLRS), define

$$E(i,j) = M - j + i + 1 - \sum_{k=i+1}^{j} l_k$$

where $E(i,j)$ denotes the number of extra space characters at the end of the line containing words from $i+1$ to j (both inclusive). Since a line cannot have more than M characters, we require $E(i,j) \geq 0$. We wish to minimize the sum of cubes of $E(i,j)$ over all lines except the last. Let us define a **penalty** term $P(i,j)$ as follows:

$$P(i,j) = \begin{cases} 0 & \text{if } j = n \text{ and } E(i,j) \geq 0 \\ \infty & \text{if } E(i,j) < 0 \\ (E(i,j))^3 & \text{otherwise} \end{cases}$$

Let $C(j)$ be the **optimal cost** of printing words from 1 to j, such that the word j ends *some line*. Then, the base case is

$$C(0) = 0$$

and for $j > 0$ we have

$$C(j) = \min_{0 \leq i < j} \{C(i) + P(i,j)\}$$

To actually *print neatly*, we will have to store i that minimizes $C(j)$ for each j. This tells us what the last word on the previous line is, i.e. the line before the line that ends in j. Clearly, $C(j)$ exhibits optimal substructure, and contains overlapping subproblems. The verification of the same is left to the reader.

2 Pseudocode

The pseudocodes in this section may not exactly conform with the sketch of the algorithm in the previous one (in terms of notation, indices, etc.), however, the idea remains the same. Care has been taken to establish close correspondence between the pseudocodes and implementations done in C++ (submitted via HackerRank). Comments have been added in places to ensure clarity.

2.1 Q1(a)

Algorithm 1 NeatPrinting$(C, L, P, \text{prefix}, j, n)$

Input: An array L of words, an integer array **prefix**, and integers j and n. prefix$[i]$ denotes the sum of lengths of words from 1 to i. $C[i]$ is the optimal cost of printing words from 1 to i. $P[i]$ assumes that the last word of current line is $L[i]$, and its value denotes the last word in the previous line. Note that prefix$[0] = 0$ and $C[0] = 0$.

```
1: if C[j] < ∞ then
2:     return C[j]                                   ▷ subproblem already solved
3: i ⟵ j
4: while i ≥ 1 do
5:     spaces ⟵ M − j + i − (prefix[j] − prefix[i − 1])        ▷ words from i to j
6:     if spaces < 0 then
7:         break                                     ▷ spaces< 0 always hereafter
8:     else
9:         a ⟵ NeatPrinting(C, L, prefix, i − 1, n)
10:        b ⟵ spaces^3
11:        if j = n then
12:            b ⟵ 0                                 ▷ zero cost for last line
13:        C[j] ⟵ min(C[j], a + b)
14:        if a + b ≤ C[j] then
15:            P[j] = i − 1
16:    i −= 1
17: return C[j]
```

Algorithm 2 PrintSolution(P, L, end)

Input: Arrays P and L as defined earlier, and an integer **end**.

```
1: if end= 0 then
2:     return
3: PrintSolution(P[end])
4: i ⟵ P[end] + 1
5: while i ≤ end do
6:     print(L[i])
7:     i += 1
```

Continued on next page

Algorithm 3 TopDown(n, M)

Input: M is the maximum number of characters in a line, and n is the number of words.

```
1: prefix[0, 1, ..., n + 1]                          ▷ prefix array of size n + 1
2: C[0, 1, ..., n + 1]                               ▷ optimal cost array of size n + 1
3: P[1, 2...., n]                                    ▷ integer array of size n
4: L[1, 2...., n]                                    ▷ array of strings of size n
5: prefix[0] ⟵ 0
6: C[0] ⟵ 0
7: i ⟵ 1
8: while i ≤ n do
9:     read prefix[i]                                ▷ take integer input into prefix[i]
10:    prefix[i] +=prefix[i − 1]
11:    P[i] = 0
12:    read L[i]                                     ▷ read the word into L[i]
13:    C[i] = ∞
14:    i += 1
15: s ⟵NeatPrinting(prefix,n, n)
16: PrintSolution(P, L, n)
17: print(S  mod 10^9 + 7)
```

A closer look at the **prefix** array suggests that

$$\text{prefix}[i] = \sum_{k=1}^{i} l[k]$$

where $l[k]$ denotes the length of the k^{th} word. Storing this sum for all $i = 1, 2, ..., n$ saves us from a $\mathcal{O}(n^3)$ (overall) algorithm, as Line 5 in NeatPrinting turns into a constant-time operation.

2.2 Q1(b)

Algorithm 4 ExtraSpaces(l, i, j)

Auxilary function to calculate extra spaces left at the end of the line after printing the words $L[i + 1]$ to $L[j]$.

Input: l is an integer array containing lengths of words in L. i, j are integers.

```
1: sum⟵ l[j] − l[i]
2: return M − j + i + 1−sum
```

Algorithm 5 PrintingCost(l, i, j)

Returns the cost of printing exactly the words indexed from $i + 1$ to j in the same line.

Input: l is an integer array containing lengths of words in L. i, j are integers.

```
1: E_ij ⟵ ExtraSpaces(l, i, j)
2: if E_ij < 0 then
3:     return ∞
4: if j = n then
5:     E_ij ⟵ 0
6: return E^3_ij
```

Continued on next page

Algorithm 6 NeatPrinting(P, C, l, j)

Input: l is an integer array containing lengths of words in L. j is an integer.

```
1: temp⟵ ∞
2: i ⟵ j − 1
3: while i ≥ 0 do
4:     if ExtraSpaces(l, i, j)< 0 then
5:         break                                  ▷ extra spaces are always negative hereafter
6:     curr ⟵ C[i]+PrintingCost(l, i, j)
7:     if curr<temp then
8:         temp⟵curr
9:         P[j] ⟵ i
10:    i −= 1
11: C[j] ⟵ temp
12: return
```

Algorithm 7 PrintSolution$(P, L,$ end$)$

Input: Arrays P and L as defined earlier, and an integer **end**.

```
1: if end= 0 then
2:     return
3: PrintSolution(P[end])
4: i ⟵ P[end] + 1
5: while i ≤ end do
6:     print(L[i])
7:     i += 1
```

Algorithm 8 BottomUp(n, M)

Input: Arrays P and L as defined earlier, and an integer **end**.

```
1: prefix[0, 1, ..., n + 1]                       ▷ prefix array of size n + 1
2: C[0, 1, ..., n + 1]                            ▷ optimal cost array of size n + 1
3: P[1, 2...., n]                                 ▷ integer array of size n
4: L[1, 2...., n]                                 ▷ array of strings of size n
5: prefix[0] ⟵ 0
6: C[0] ⟵ 0
7: i ⟵ 1
8: while i ≤ n do
9:     read prefix[i]                             ▷ take integer input into prefix[i]
10:    prefix[i] +=prefix[i − 1]
11:    P[i] = 0
12:    read L[i]                                  ▷ read the word into L[i]
13:    C[i] = ∞
14:    i += 1
15: i ⟵ 1
16: while i ≤ n do
17:    NeatPrinting(P, C, l, i)
18:    i += 1
19: PrintSolution(n)
20: print(C[n]  mod 10^9 + 7)
```

3 Complexity Analysis

3.1 Q1(c)

3.1.1 The Top-Down Approach

The top-down approach is described by Algorithm 3, which is accompanied by auxiliary algorithms 1 and 2. First, let us discuss the time-complexity of NeatPrinting. For brevity, we shall denote NeatPrinting(C, L, P,prefix, j, n) by $f(j)$. Line 15 of Algorithm 3 calls $f(n)$.

There are two possibilities for $f(j)$ (for all $0 \leq j \leq n$). $f(j)$ may return from Line 2 (of Algorithm 1), or enter the loop on Line 4. If $f(j)$ terminates on Line 2, it is a constant time (say c) operation. If not, we enter the while loop and loop through it *at most* j times (as i descends from j to 1). It is easy to see that all operations except the one on Line 9 in the while loop, are constant-time operations.

Note 1: Before the first call to NeatPrinting, $C[j] = \infty$ for all $j \neq 0$.

Note 2: Once the while loop in $f(j)$ terminates, $C[j]$ is set to some finite value.

Note 3: prefix$[i_1] \leq$prefix$[i_2]$ for all $0 \leq i_1 \leq i_2 \leq n+1$

Note 4: In Algorithm 1, if spaces< 0 (condition in Line 6 is true), then $M-j+i <$prefix$[j]-$prefix$[i-1]$, i.e. prefix$[i-1] <$prefix$[j] + j - i - M$. Since i decreases in subsequent iterations, from Note 3 it is clear that prefix$[i-1] <$prefix$[j] + j - i - M$ continues to hold for all i here forward.

The first call to NeatPrinting is $f(n)$. In view of Note 1, we enter the while loop. Whenever Line 9 is visited inside the while loop, a recursive call to f is made, and these are in the order $f(n-1), f(n-2), ..., f(n-k+1)$.

Claim 5: k in the above statement is such that $k \leq \min\left(\left\lceil \frac{M}{2} \right\rceil, n\right)$

Proof of Claim 5:

- If $M \leq n$:
  k is essentially the number of times the while loop in Algorithm 1 runs without breaking on Line 7. The maximum of $\lceil \frac{M}{2} \rceil$ is achieved when all the words have length 1. In this case, a line can contain at most $\lceil \frac{M}{2} \rceil$ many words after taking into account white-space characters between any two pair of consecutive words. In all other cases, we *exhaust* the limit of M characters in a line relatively quickly, since at least one word is of length ≥ 1. This shows $k \leq \lceil \frac{M}{2} \rceil$.
- If $M \geq n$:
  The number of iterations of the while loop in Lines 4-16 is upper bounded by n, so $k \leq n$.

Combining the above two cases, we get $k \leq \min\left(\left\lceil \frac{M}{2} \right\rceil, n\right)$.

In view of Notes 1-4 and Claim 5, we see that when $f(n)$ is called, the while loop is executed n times, but Line 9 is reached only k times, where k is bounded as above. The time complexity of Algorithm 1 is thus $\mathcal{O}\left(\min\left(\left\lceil \frac{M}{2} \right\rceil n, n^2\right)\right)$ which is just $\mathcal{O}\left(\min\left(Mn, n^2\right)\right)$.

PrintSolution (Line 16 of TopDown(n, M)) takes $\mathcal{O}(n)$ time and Lines 1-14,17 in TopDown(n,M) also take $\mathcal{O}(n)$ time. In the sketch above, we saw that NeatPrinting(prefix,n, n) takes $\mathcal{O}\left(\min\left(Mn, n^2\right)\right)$ time.

Therefore, **TopDown(n, M) has $\mathcal{O}\left(\min\left(Mn, n^2\right)\right)$ time complexity.**

TopDown(n, M) has $\mathcal{O}(n)$ space complexity. This is easily seen from Lines 1-4 (initializing arrays) of Algorithm 3. Apart from these arrays, all other variables occupy constant space.

Continued on next page

3.1.2 The Bottom-Up Approach

Note 6: Algorithm 4 and 5 take constant time, and Algorithm 7 takes $\mathcal{O}(n)$ time.

As in the Top-Down approach taken in the previous section, the time complexity of BottomUp(n, M) also primarily depends on NeatPrinting(P, C, l, j), i.e. Algorithm 6. In BottomUp(n, M), Lines 1-7 take constant time and Lines 8-15 take linear time. In this section, we continue to refer to NeatPrinting(P, C, l, j) as $f(j)$.

Note 7: prefix$[i_1] \leq$prefix$[i_2]$ for all $0 \leq i_1 \leq i_2 \leq n+1$

Note 8: In Algorithm 6, if spaces< 0 (condition in Line 4 is true), then $M-j+i <$prefix$[j]-$prefix$[i-1]$, i.e. prefix$[i-1] <$prefix$[j] + j - i - M$. Since i decreases in subsequent iterations, from Note 7 it is clear that prefix$[i-1] <$prefix$[j] + j - i - M$ continues to hold for all i here forward.

Similar to the Top-Down Approach, the loop in Algorithm 6 will be executed k times, where $k \leq \min\left(\left\lceil \frac{M}{2} \right\rceil, n\right)$. The proof is also similar to that of Claim 5. In Algorithm 8, the loop starting on Line 16 runs for n iterations. In view of Notes 6-8, the overall complexity of BottomUp(n, M) is therefore $\mathcal{O}\left(\min\left(\left\lceil \frac{M}{2} \right\rceil n, n^2\right)\right)$ which is just $\mathcal{O}\left(\min\left(Mn, n^2\right)\right)$.

Therefore, **TopDown(n, M) has $\mathcal{O}\left(\min\left(Mn, n^2\right)\right)$ time complexity.**

BottomUp(n, M) has $\mathcal{O}(n)$ space complexity. This is easily seen from Lines 1-4 (initializing arrays) of Algorithm 8. Apart from these arrays, all other variables occupy constant space.

3.1.3 Conclusion

Q1(c): *What is the complexity of your algorithms in (a) and (b). Do you see a difference in the time taken by programs in (a) and (b), in practice on large inputs?*

In (a), i.e. the top-down approach:

1. Time complexity is $\mathcal{O}\left(\min\left(Mn, n^2\right)\right)$. If $M < n$ (more practical case), it is $\mathcal{O}(Mn)$.
2. Space complexity is $\mathcal{O}(n)$

In (b), i.e. the bottom-up approach:

1. Time complexity is $\mathcal{O}\left(\min\left(Mn, n^2\right)\right)$. If $M < n$ (more practical case), it is $\mathcal{O}(Mn)$.
2. Space complexity is $\mathcal{O}(n)$

where n equals the number of words to be printed neatly in the paragraph, given the constraints.

CPU Runtime:

We measured the runtime for the Top-Down and Bottom-Up approaches. We considered $n = 10000$. Observation:

- **Q1(a)** Top-Down: 0.007407 s
- **Q1(b)** Bottom-Up: 0.006716 s

Both algorithms have the same complexity, i.e. $\mathcal{O}\left(\min\left(Mn, n^2\right)\right)$ as discussed at length in 3.1.1 and 3.1.2. This minor difference can be attributed to the recursive nature of Top-Down approach, which leads to a slight overhead. On the other hand, the Bottom-Up approach doesn't lead to recursive function calls to solve smaller subproblems.