ESO207

Data Structure and Algorithms

Indian Institute of Technology, Kanpur

Group Number: Runtime Terror

Tejesh Vaish (190908), Shorya Kumar (190818), Mandar Bapat (190475)

Assignment 4

Date of Submission: 17th November, 2020

---

1 Question 1

1.1 Part C

1.1.1 Code for Top Down approach

```
/*
For solving the problem corresponding to a list of words we will break it into first row and a identical smaller sub problem , for instance if we have to solve for a list s[i,n] , then once we fix the first row s[i,j] , then we are left with a identical smaller problem corresponding to s[j+1,n] , extrapolating this nature of the problem we just have to take the minimum of ( no of characters including spaces in the row corresponding to partition [i,j] + solution for s[j+1,n] )from all valid partitions of s[i,j] , a partition is valid if total character count of s[i,j] is less than or equal to M . We use memorization to store values of answer for s[i,n] in mi[i] and the end index j for the optimal partition in en[i].
*/

ll mi[100004];//this is used to store the answer for the sub problem [i...n] (i.e minimum value of sum of the cubes ..)
ll en[100004];   // this is used to store the end index for the row beginning at index i for the optimal case
ll p[100004];    // this is the prefix sum of lengths of the words such that p[i] stores the sum of lengths of first i                         // words , it is used so the sum of length of words can be quickly calculated

```

```

ll m;            // max number of characters that can come in one row

ll ans(int i,int n){  // this returns the answer for the sub problem s[i...n]
    if(i>n){             // if starting index is greater than n then no row is formed and this returns 0
        return 0;
    }
    if(mi[i]!=-1){       // if mi[i] is not equal to -1 , which means answer to the sub problem is already determined
        return mi[i];
    }
    if(n-i+p[n]-p[i-1] <= m){//if (n-i) number of spaces + (p[n]-p[i-1]) the sum length of words from i to n is less than
            //equal to M , hence only one row is needed and again the answer to the sub problem turns                    //out to be zero
        en[i]=n;                 // end index for all such i is n
        mi[i]=0;                 // answer is zero
        return 0;
    }

    // if none of the above condition holds then at least two rows will be formed and we need the optimised partition of the       problem s[i,n] into s[i,j] and s[j+1,n] such that ( ans(j+1,n)+ pow(m-(j-i+p[j]-p[i-1]),3) ) is minimum among all j          from i to n , so we recursively check for all the valid partitions j such that total length of characters                    corresponding to the row starting at i and end at j should not be more than m.

    ll t = _LONG_LONG_MAX_;  // temporary variable to find the minimum

    // now checking for all valid partitions [i,j] that we can make starting at i


    for(int j = i ; (j<=n && (j-i+p[j]-p[i-1] <= m)) ;j++){//till j<=n and (j-i+p[j]-p[i-1] <= m)(length constraint)
        ll te = ans(j+1,n)+ pow(m-(j-i+p[j]-p[i-1]),3);    // computing the answer corresponding to partition [i,j]
```

```
34         if(t > te ){                                              // if te if less than t
35             en[i]= j;                                     // endpoint for segment starting at i is revised to j
36             mi[i]=te;                                     // and the ans for s[i,n] is revised to te
37             t=te;                                          // t is revised to te
38
39         }
40
41     }
42     // at the end of the loop mi[i] contains minimum possible value for the sub problem [i,n]
43     return mi[i];     // returning the ans
44 }
45
46
47
```

1.1.2 Time Complexity for Top Down approach

We are interested in finding worst-case time complexity
of the *ans()* function.
Note that we are making 2 kinds of recursive calls in the
function:
(a.)The calls which are returned at line numbers 13,16 and 19
in the above code which take O(1) time.
(b.)Those in which for loop is executed

Note that call for each subp roblem falls in 2nd kind at most once , as once its computed its stored in mi(dp) array and returned from line 7 if called again. Hence the internal for loop is called at most n times as there are total n sub problems.

Now analysing the inner for loop , we pay attention to the break condition of the loop , (i.e (j<=n && (j-i+p[j]-p[i-1] <= m)) in line 22 . As we can see j starts from i and j<=n hence at most n iterations are

executed as j increases after each iteration of the loop.
, whereas it is obvious from second term of the condition
that (j-i)<=m as p[j]-p[i-1] is non negative, again as j increases by
one after each iteration and it starts from i and is less than equal
to m hence the loop runs for atmost m times (once we
find a word that makes the chracter sum of the row more than
m , then there is no reason to place further words
on that line and no need to further
increase j) . From the above arguments
its clear that for loop iterates for atmost min(n,m) times. And in each
iteration we make one call in line 23 . Hence at most min(n,m) calls
are made from the for loop.
As for loop is reached in at most n calls of the function and each
for loop makes at most min(n,m) calls , note that these min(n,m) calls are
of first kind as for all the calls of second kind are already taken care of
when we considered total n calls of 2nd kind. hence time complexity is
of the order of n*(c*min(n,m))

Hence the total time complexity is O(n*min(n,m)).
which for n>m reduces to O(n*m)
In the view of the problem where m is limited to 80 and n can range to as large as
10 raise to 5 , for such cases m can be treated as a bounded constant and we can say
for very large inputs this algorithm reduces to O(80*n) which is O(n).

1.1.3 Code for Bottom Up approach

```
int main() {
    ll n;              // number of words
    cin>>n>>m;         // taking number of words and M as input
    ll l[n+1];         // array to store length of the words
    string s[n+1];     // array to store the words
    for(int i=1;i<=n;i++){
        cin>>l[i];         //    taking length as input in l[i]
        cin>>s[i];         // taking word as string and storing it in s[i]

    }
```

```
 p[0]=0;                    // initializing p[0]=0;
 for(int i=1;i<=n;i++){     // initializing the prefix array in which p[i] contains the sum of lengths of
     p[i]= p[i-1]+l[i];     // first i words , it will be used in quickly calculating no of characters in a row
 }
 for(int i=1;i<=n;i++){     // initializing mi and en to -1, -1 signifies that the sub problem has not been called yet
     en[i]=-1;
     mi[i]= -1;
 }

 /*
 next it the code for filling the array mi[ ]  and en[ ] in bottom up style , starting from i=n
 */
 for(int i=n;i>=1;i--){                 // i from n to 1
     if(n-i+p[n]-p[i-1] <= m){          // till the sum of all characters in the row is less than or equal to m as the
         en[i]=n;                       // last row is not counted ending index is n and the mi[i] is zero.
         mi[i]=0;
     }
     else{                             // else we take the minimum of mi[j+1]+ pow(m-(j-i+p[j]-p[i-1]),3); for all j from i to n where j denotes the ending index if row starting from i

         ll t = _LONG_LONG_MAX_;
         for(int j = i ; (j<=n && (j-i+p[j]-p[i-1] <= m)) ;j++){// checking for all valid first rows [i,j]
         ll te = mi[j+1]+ pow(m-(j-i+p[j]-p[i-1]),3);
             if(t > te ){   //if te is less than t than updating the optimal tables with new values
                 en[i]= j;   // ending index with j
                 mi[i]=te;   // the minimum sum of cubes with te
                 t=te;       // t stores the minimum till now

             }

         }
```

```
41
42             // at the end mi[i] contains the minimum possible answer for the sub problem s[i,n]
43         }
44     }
45
46 }
```

1.1.4 Time Complexity for Bottom Up approach

We are interested in finding worst-case time complexity of the algorithm.This basically requires us to analyze the time complexity of nested for loops that are used to fill our mi and en (dp arrays) in lines 24-44 .

The outer *for* loop runs *n* times.
The inner for loop breaks at 2 conditions similar to the *for* loop in the recursive *ans()* function of the *Top Down* approach.Hence the inner loop runs for *min(n,m)* times as before. And time taken in every iteration of inner loop is constant hence T.C of the loop is of order of c*min(n,m) and since such for loops are executed at most n times hence T.C is of the order of n*c*min(n,m) .

Hence the total time complexity is O(n.min(n,m)).

Again if n is very large compared to m , the algorithm essentially becomes linear in n .

1.1.5 Difference for Large Inputs

The time taken by bottom up approach will be **less** than the top down approach , as is evident by the algorithm that top down approach uses **recursion** where as bottom up approach uses **iterative method** which is faster than recursion , so this will cause the top down version to take more time in case LARGE inputs .

In practical scenario , when we run the program over the input word array of length 10^5, the bottom up approach took on average **16926.7** micro seconds(on 10000 samples) , where as top down approach take **17548.1** micro seconds(on 10000 samples), which is slower by 1 millisecond , **the difference in time taken will further increase as it will take even longer time to run larger input on the algorithm , hence in this case , bottom up approach takes tad less time than top down approach !**

References

lecture notes