1: Printing Paragraph

Analysis: : Let $s(i,j) = M - j + i - \sum_{k=i}^{j} l_k$

be the number of extra spaces required to put words from i to j on the same line. When s(i,j) is greater or equal to 0, we can fit words into the line. Otherwise, we cannot do that. Therefore, we can summurize the cost c(i,j) as:

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

The dynamic programming method: Let C(j) be the optimal method for printing n words from word 1 to word j, Then we have
$C(0) = 0$
$C(j) = \min_{j \geq i \geq 0} C(i) + C(i,j) \quad for \quad j > 0$
for our algorithm, the input is an integer array l[ ], with slot i representing the length of word i M is the maximum length per line, and n is the size of our input array.

Running time: : The complexity of the algorithm is $O(n^2)$ The space requirement for the algorithm is about $O(n)$ complexity

Algorithm: :

$EX[n+1][n+1]$ ▷ number of extra spaces on a single line for word i to j
$LC[n+1][n+1]$ ▷ the cost for a single line
$C[n+1]$ ▷ the total cost of our optimal solution for word from 1 to j
$p[n+1]$ ▷ used to print the optimal solution
i and j
▷ calculate the extra space for each word when they are put into a single line
for i from 1 to n **do**
  $EX[i][i]$ = M - $l[i-1]$
  for j from $i+1$ to n **do**
    $EX[i][j]$ = $EX[i][j-1]$ - $l[j-1]$ - 1
▷ calculate the corresponding line costs
for i from 1 to n **do**
  for j from 1 to n **do**
    if $EX[i][j]$ ¡ 0 **then**
      $LC[i][j]$ = infinity
    else if j = n and EX[i][j] greater or equal to 0 **then**
      $LC[i][j]$ = 0
    else
      $LC[i][j]$ = $EX^3[i][j]$
▷ calculate minimum cost
$C[0]$ = 0
for j from 1 to n **do**

$C[j]$ = `infinity`
for i `from` 1 `to` j **do**
if `C[i-1] != infinity and LC[i][j] != infinity and C[i][j] + LC[i][j] < C[j]` **then**
$C[j]$ = $C[i-1]$ + $LC[i-1]$
$P[j]$ = i

2: Palindrome subsequence

Analysis: : approach the problem using bottom-up approach, by beginning with subsequences with length 1 all the way up to the string with length equal to the input string length. If the input string has length n, The data structure used is a n by n matrix.

Suppose that the input is a string s with length n

Running time: : The Time complexity of the algorithm is $O(n^2)$

Algorithm: :

$chars[n]$ ←s ▷ convert the input string to a char array
R ←R[n][n] ▷ Initialize a new n by n matrix
if $n = 0$ **then**
return 0 ▷ empty string
else
for `i from` 1 `to` n **do**
`R[i][i] = 1` ▷ add characters to the matrix column
for `sublen from` 2 `to` n **do**
for `i from` 0 `to` n-$sublen$ **do**
j = i + $sublen$ - 1
if chars[i] = chars[j] and $sublen = 2$ **then**
`R[i][j] = 2`
else if chars[i] = chars[j] **then**
`R[i][j] = R[i+1][j-1] + 2`
else
`R[i][j] = max(R[i+1][j] and LP[i][j-1])`
`return R[0][n-1]`

3: Minimum Cost of Cutting a String

Analysis: : the input of the problem is a stirng s with length n, and a set of m indices to break the string according to the indexes we need a best ordering of the breaks to minimize the cost of cutting the string pick index i and j, where i is less than j, and i, j are between 0 and n denote the minimum cost for cutting the substring from index i to j as C(i,j) Recurrence: C(i,j) = mininum of (length of substring + C(i,k) + C(k,j) where k is between i and j) Build a n by n table and find the minimum cost of each slot (i,j) where i and j are between 0 and n

Running time: : building up the table takes $O(n^2)$ at most we need to look up n items and compare, which gives $O(n)$ total cost is about $O(n^3)$

Algorithm: :
  $C[n][n]$ ▷ the cost table that stores cost for substring(i,j)

4: Schedule to minimize the lateness

Part(a) : Suppose that we have an optional solution J, and suppose we have an inversion in the optimal solution: if job i is scheduled before job j but $d_i > d_j$, in this case, the finish time will still be before d_i and the swapping will not change finish time for other jobs, this is still optimal after the swapping.

Any permutation of the optimal solution is a combination of swappings in the optimal soliution. Since swapping doesn't change optimality, we can have an optimal solution where the dealines are in increasing order.

Part(b): Suppose that $d_1 \leq d_2 \leq ... \leq d_n = D$, since in part (a), we have proven that there is an optimal solution in increasing order. so we recognize subproblems as $OPT(i, j)$, which means the maximal number of schedulable jobs among $1, 2, ..., i$ under the constraint that all the chosen job must finish by time j. Then $OPT(i, j)$ is achieved either by including job i in the schedule (at the end) or not, i.e.,

$$OPT(i, j) = \max \{OPT(i-1, j-t_i) + 1, OPT(i-1, j)\} \tag{1}$$

where $i \geq 1$ and $j \geq 0$, and if $j < t_i$, job i cannot be involved in the schedule, so $OPT(i, j) = OPT(i-1, j)$ in this case. And the base case is

$$OPT[0][j] = 0 \text{ for } j = 0, 1, ..., D \tag{2}$$

Apply dynamic programming, we use a 2D array `OPT[0..n][0..D]` to store the values of $OPT(i, j)$ In the end, $OPT[n][D]$ will be our final solution.

Algorithm: :
  $OPT[n][D]$
  for `i from 0 to D` **do**
    `OPT[0][j] = 0`
  for `i from 1 to n` **do**
    for `j from 1 to D` **do**
      if t_i< j **then**
        $OPT[i][j]$ = $OPT[i-1][j]$
      else
        max(OPT[i-1][j-t_i] and OPT[i-1][j])

Running time: : after we have constructed the table, we need backtracking to find the value $OPT[n][D]$,if $OPT[i][j] = OPT[i-1][j]$, job i is not included in the schedule, if OPT[i][j] = OPT[i-1][j-t_i]+1, job i is included in the schedule. Do this backward tracing until we reach$OPT[0][0]$ sorting the elements takes $O(n \log n)$, filling table takes constant time $O(D)$. backtracking takes $O(n)$ thus we need $O(nD)$