cs 5800 - hw6

Yijing Xiao

October 2021

1 Jars on a ladder problem

1.1

We could consider this problem by using dynamic programming. When the first jar is drooped in ith rung, if the jar break then we don't need to consider $[i+1, n]th$ rung and we only have $k-1$ jars now. We shall user $MinT(i-1, k-1)+1$ to express the minimum number of dropping trials has to make from 1 to $n-1$ rung; Otherwise, if the jar not break in ith rung, then we don't consider $[1, i]th$ rung any more, and we still have k jars left. Thus. the minimum number of dropping trails has to make is $MinT(n-i, k)+1$ where $1 \leq i \leq n$. Therefore, in different cases the minimum number of dropping trails is:

$$\begin{cases} n\ rungs\ and\ k\ jars\ left: \ MinT(n-i,k)+1 \\ dropping\ on\ the\ ith\ rung\ and\ jar\ break: \ MinT(n.k) \\ dropping\ on\ the\ ith\ rung\ and\ jar\ not\ break: \ MinT(n-i,k) \end{cases}$$

Thus, q could be expressed as:

$$q = MinT(n.k) = \min_{1 \leq i \leq n} \{\max[MinT(i-1, k-1), MinT(n-i, k)] + 1\}$$

and $MinT(n, 1) = n, MinT(0, k) = 0$

1.2

Assume that $MaxR(k, q)$ = the highest ladder size n doable with k jars and maximum q trails. When we drop the jar from rung, if the jar break then we need to use $k-1$ jars to get the highest ladder in $q-1$ trials: $MaxR(k-1, q-1)$. Otherwise, we need to use k jars to get the highest ladder in $q-1$ trails: $MaxR(k, q-1)$. Therefore, $MaxR(1, q) = MaxR(k-1, q-1) + MaxR(k, q-1) + 1$.

1.3

Pseudocode 1 Bottom-up non-recursive computation

```
1: function MinT(n, k)
2:     let f[0..n + 1, 0..k + 1] be a new 2D table
3:     Initialize the table f and f[0][0] = 0
4:     for i = 1 to n + 1 do
5:         f[i][1] ← i
6:     end for
7:     for j = 1 to k + 1 do
8:         f[1][j] ← 1
9:     end for
10:    for i = 2 to n + 1 do
11:        for j = 2 to k + 1 do
12:            for m = 1 to i + 1 do
13:                f[i][j] = min(f[i][j], max(f[m − 1][j − 1], f[i − m][j]) + 1
14:            end for
15:        end for
16:    end for
17:    return f[n][k]
18: end function
```

1.4

Algorithm 2 Top-down recursive using memorization

```
1: function MinT(n, k)
2:     let f[0..n + 1, 0..k + 1] be a new 2D table
3:     return Helper(n, k)
4: end function
5:
6: function Helper(n, k)
7:     if n == 0 or n == 1 then
8:         return n
9:     end if
10:    if k == 1 then
11:        return n
12:    end if
13:    if f[n][k]! = NULL then
14:        return f[n][k]
15:    end if
16:    res ← ∞
17:    for i = 1 to n + 1 do
18:        q = max(Helper(i − 1, k − 1), Helper(n − i, k) + 1)
```

```
        res = min(res, q)
    end for
    return f[n][k] = res
end function
```

1.5

Algorithm 3 Auxiliary structure to determine the optimal sequence of drops

```
function TRACE(n, k)
    let dp[0..n+1, 0..k+1] be a new 2D table
    Initialize the dp table
    i ← 0
    while dp[i][k] < n do
        i++
        for j = 1 to k + 1 do
            dp[i][j] = dp[i − 1][j] + dp[i − 1][j − 1] + 1
        end for
    end while
    return i
end function
```

2 Problem 15.2 Longest palindrome subsequence

Assume that the sequence of n numbers is $X = \{x_0, x_1, ..., x_n\}$, makes a copy of X and then reverse the copy called $Y = \{y_0, y_1, ..., y_n\}$. Procedure LPS-Length takes X and Y as inputs. It stores the $c[i, j]$ values in a table $c[0..n, 0..n]$ and it computes the entries in row-major order(That is, the procedure fills in the first row of c from left to right, then the second row, and so on). The procedure also maintains the table $b[1..n, 1..n]$ to help us construct an optimal solution. Intuitively, $b[i, j]$ points to the optimal subproblem solution chosen when computing $c[i, j]$. The procedure returns the b and c tables.

As in the matrix-chain multiplication problem, our recursive solution to the LPS problem involves establishing a recurrence for the value of an optimal solution. If either $i = 0$ or $j = 0$, one of the sequence has length 0, and so the LPS has length 0. The optimal substructure of the LIS problem gives the recursive formula

$$c[i,j] = \begin{cases} 0, & \textbf{if } i = 0 \textbf{ or } j = 0 \\ c[i-1, j-1] + 1, & \textbf{if } i, j > 0 \textbf{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]), & \textbf{if } i, j > 0 \textbf{ and } x_i \neq y_j \end{cases}$$

Algorithm 4 Find the longest palindrome subsequence

```
1: function LPS-LENGTH(X, Y)
2:     n ← X.length
3:     let b[1..n, 1..n] and c[0..n, 0..n] be two new tables
4:     for i = 1 to n do
5:         c[i, 0] ← 0
6:     end for
7:     for j = 0 to n do
8:         c[0, j] ← 0
9:     end for
10:    for i = 1 to n do
11:        for j = 1 to n do
12:            if x_i == y_i then
13:                c[i, j] ← c[i − 1, j − 1] + 1
14:                b[i, j] =↖
15:            else
16:                if c[i − 1, j] ≥ c[i, j − 1] then
17:                    c[i, j] ← c[i − 1, j]
18:                    b[i, j] =↑
19:                else
20:                    c[i, j] ← c[i, j − 1]
21:                    b[i, j] =←
22:                end if
23:            end if
24:        end for
25:    end for
26:    return c and b
27: end function
```

The b table returned by LPS-Length enables us to quickly construct an LPS of X. We simply begin at $v[n, n]$ and trace through the table by following the arrows. Whenever we encounter a "↖" in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the longest palindrome sequence that LPS-Length found. With this method, we encounter the elements of this LPS of X. The initial call is $PRINT - LPS(b, X, X.length, X.length)$.

Algorithm 5 Print the longest palindrome subsequence

```
1: function PRINT-LPS(b, X, i, j)
2:     if i == 0 or j == 0 then return
3:     end if
4:     if b[i, j] == " ↖ " then
5:         PRINT-LPS(b, X, i − 1, j − 1)
6:         print x_i
7:     else
```

```
        if b[i, j] == " ↑ " then
            PRINT-LPS(b, X, i − 1, j)
        else
            PRINT-LPS(b, X, i, j − 1)
        end if
    end if
end function
```

The running time of the procedure is $\Theta(n^2)$, since each table entry takes $\Theta(1)$ time to compute. The procedure of PRINT-LPS takes time $O(n + n) = O(2n)$, since it decrements at least one of i and j in each recursive call.

Therefore, we can solve the palindrome problem in time $O(n^2)$.

3 Exercise 15.4-6

Assume that we have a sequence S and its the longest monotonically increasing subsequence is S', then the last element of S' of length n is at least as large as the last element of S' of length $n-1$. Each time we iterate over S, we only need to do the following to find S':

1. If x is larger than all elements of S', that is x can be placed at the end of S', then append x to the S' and increase the length of S' by 1.

2. If $S'[i-1] < x \leq s'[i]$, that is x needs to replace the preceding number greater than x in order to ensure that S' are an increasing sequence.

Algorithm 6 $O(n \lg n)$-time to find the longest monotonically increasing subsequence

```
function LIS(S)
    n = S.length
    S'[0] ← −1
    for i = 0 to n do
        if S[i] > S'[−1] then
            S'.append(S[i])
        else
            Use binary search to replace the preceding number greater than x
        end if
    end for
    return S'
end function
```

Therefore, the total running time is $O(n \lg n)$.

4 Problem 15-4 Printing neatly

Since single word cannot be printed separately, we assume that the length of word k does not exceed the number of characters that can be contained in a line, that is $l_k \leq M$ where $k = 1, 2, ..., n$. For any line, assuming that it begins with the word i and end with word j, the number of spaces left at the end of the line is

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

we denote $line - cubes(i,j)$ as the sum of the cubes of the numbers of extra space characters at the end of a line. If $e(i,j) < 0$, that means a line cannot contain all words through i to j, so it is not appropriate. If $e(i,j) \geq 0$ and $j = n$, that means this line is last, so $line - cubes(i,j) = 0$ since the last line is not necessary to calculate the sum of cubes. If $e(i,j) \geq 0$ and $j < n$, that means it is not last line, so $line - cubes(i,j) = (e(i,j))^3$. Thus, we obtains:

$$line - cubes(i,j) = \begin{cases} \infty, & e(i,j) < 0 \\ 0, & e(i,j) \geq 0 \text{ and } j = n \\ (e(i,j))^3, & e(i,j) \geq 0 \text{ and } j < n \end{cases}$$

Assuming that the optimal solution of the sum if cubes of extra space characters at the ends of all lines is $cubes(j)$ from first word to word j. If $j < n$, then we don't count the last line. If we assume that word i is the beginning of the last line where $1 \leq i \leq j$, then the cube is $line - cubes(i,j)$ and other words $1 - i - 1$ is $cubes(i-1)$. Thus, we obtains:

$$cubes(j) = \begin{cases} 0, & j = 0 \\ \min\limits_{1 \leq i \leq j} \left\{ \max line - cubes(i,j), cubes(i-1) \right\}, & j > 0 \end{cases}$$

Algorithm 7 Print a paragraph of n words neatly on a printer

```
1:  function NEATLY-PRINT(l, n, M)
2:      let e[1..n, 1..n] and line − cubes[1..n, 1..n] be two new tables
3:      let cubes[0..n] and p[1..n] be two new arrays
4:      for i = 1 to n do
5:          e(i, j) = M − l_i
6:          for j = i + 1 to n do
7:              e(i, j) = e(i, j − 1) − l_i − 1
8:          end for
9:      end for
10:     for i = 1 to n do
11:         for j = 1 to n do
```

```
            if e(i, j) < 0 then
                line − cubes(i, j) = ∞
            else
                if j = n then
                    line − cubes(i, j) = 0
                else
                    line − cubes(i, j) = (e(i, j))^3
                end if
            end if
        end for
    end for
    cubes(0) = 0
    for j = 1 to n do
        cubes(j) = ∞
        for i = 1 to j do
            if line − cubes(i, j) + cubes(i − 1) < cubes(j) then
                cubes(j) = line − cubes(i, j) + cubes(i − 1)
                p(j) = i
            end if
        end for
    end for
    return cubes(n)
end function
```

Since $NEATLY - PRINT$ include two double loops and one single loop, the running time is $O(n^2)$

5 Problem 15-10 Planning an investment strategy

5.1

There are three cases:

1. Move your money every year

In this case, it is obvious that you should choose the investment with the highest return rate each year to maximize your 10-year total amount.

2. Do not move your money every year

Suppose that the 10-year total amount of money of choosing the first investment product is E_1, choosing the second investment product is E_2, ..., and choosing the nth investment product is E_n. If we choose two investment product k_1 and k_2, then the 10-year total amount must be the weighted sum of E_{k1} and E_{k2} which is not greater than the biggest one of E_{k1} and E_{k2}. In this case, only choose one investment product each year which can maximize the return.

3. Some years move the money and some years not

If we consider some years which move the money as consecutive years, then it will

be regarded as a subproblem like case 1.If we consider some years which do not move the money as consecutive years, then it will be regarded as a subproblem like case 2.

Therefore, there exists an optimal investment strategy that, in each year, puts all the money into a single investment.

5.2

$e[i, j]$ **denote that the maximized total earning before** jth **year if we choose** ith **investment product in** jth **year. Analysis of** $dp[i][j]$ **needs to consider two cases:**

1. Do not move the money in jth **year**

In this case, we must choose ith **investment product in** $(j-1)th$ **year.**

$$dp_1[i][j] = (dp[i][j-1] - f_1) * r_{ij}$$

2. Move the money in jth **year**

In this case, we can select any investment product except ith **investment product. Of course, we need to select the product which can maximize our total amount of money before** jth **year.**

$$dp_2 = \max_{1 \le k \le n} (dp[k][j] - f_2) * r_{ij} \text{ and } k \neq i$$

The $dp[i][j]$ **should choose the biggest one between case 1 and case 2, that is** $dp[i][j] = \max\{dp_1[i][j], dp_2[i][j]\}$

Therefore, the problem of planning your optimal investment strategy exhibits optimal substructure since the maximizing returns in jth **year depends on the maximizing returns in** $(j-1)th$ **year.**

5.3

Algorithm 8 Optimal investment strategy

```
1: function INVEST-STRATEGY(r, n, c)
2:     let e[1..n, 1..10] and p[1..n, 2..10]be two new 2D tables
3:     for i = 1 to n do
4:         e[i, 1] = c * r_{i1}
5:     end for
6:     for j = 2 to 10 do
7:         for i = 1 to n do
8:             e[i, j] = (e[i, j − 1] − f_1) * r_{ij}
9:             p[i, j] ← i
10:            for k = 1 to n do
11:                if k ≠ i then t = (e[k, j − 1] − f_2) * r_{ij}
12:                    if t > e[i, j] then
```

```
                    e[i, j] = t
                    p[i, j] ← k
                end if
            end if
        end for
    end for
    max ← e[1, 10]
    s_1 0 ← 1
    for k = 1 to n do
        if e[k, 10] > max then
            max = e[k, 10]
            s_1 0 ← k
        end if
    end for
    return e, p and s_1 0
end function

function PRINT-INVESTMENT(p, j, s)
    if j > 0 then
        if j > 1 then PRINT-INVESTMENT(p, j, s)
        end if
        print s
    end if
end function
```

The running time is $O(n^2)$.

5.4

When there has a limitation on the amount you can invest, the amount you have to invest in the next year becomes relevant. If we have an optimal investment strategy, that means we have to solve the subproblem for every possible initial amount of money. Since the limitation of a single investment, we have to make multiple investments so the optimal substructure no longer exists.