cs 5800 - hw5

Yijing Xiao

October 2021

1 Exercise 15.2-1

We define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_iA_{i+1}...A_j$ for $1 \le i \le j \le n$. Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$. Our definition for the minimum cost of parenthesizing the product $A_iA_{i+1}...A_j$ becomes

$$m[i,j] = \begin{cases} 0, & \textbf{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}, & \textbf{if } i < j \end{cases}$$

The $m[i,j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i,j]$ to be a value of k at which we split the product $A_iA_{i+1}...A_j$ in an optimal parenthesization. That is, $s[i,j]$ equals a value k such that $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

Instead of computing the optimal cost recursively, we compute it by using a tabular, bottom-up approach. We shall implement the tabular, bottom-up method in the procedure MATRIX-CHAIN-ORDER, which appears below. This procedure assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, ..., n$. Its input is a sequence $p =< p_0, p_1, , , , , p_n >$ where $p.length = n + 1$. The procedure uses an auxiliary table $m[1..n, 1..n]$ for storing the $m[i,j]$ costs and another auxiliary table $s[1..n-1, 2..n]$ that records which index of k achieved the optimal cost in computing $m[i,j]$.

We shall use the table s to construct an optimal solution. In order to implement the bottom-up approach, we must determine which entries of the table we refer to when computing $m[i,j]$. Thus, the algorithm should fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length. .

Pseudocode 1 Find optimal parenthesization of a matrix-chain product

```
function MATRIX-CHAIN-ORDER($p$)
    $n \leftarrow p.length - 1$
    let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
    for $i = 1$ to $n$ do
        $m[i, i] \leftarrow 0$
    end for
    for $l = 2$ to $n$ do                                  ▷ $l$ is the chain length
        for $i = 1$ to $n - l + 1$ do
            $j \leftarrow i + l - 1$
            $m[i, j] \leftarrow \infty$
            for $k = i$ to $j - 1$ do
                $q = m[i, k] + m[k + 1, i] + p_{i-1}p_kp_j$
                if $q < m[i, j]$ then
                    $m[i, j] \leftarrow q$
                    $s[i, j] \leftarrow k$
                end if
            end for
        end for
    end for
    return $m$ and $n$
end function

function PRINT-OPTIMAL-PARENS($s, i, j$)
    if $i == j$ then
        print "A"
    else
        print "("
        PRINT-OPTIMAL-PARENS($s, i, s[i, j]$)
        PRINT-OPTIMAL-PARENS($s, s[i, j] + 1, j$)
        print ")"
    end if
end function
```

For sequence of $\{5, 10, 3, 12, 5, 50, 6\}$**, let** $p_0 = 5, p_1 = 10, p_2 = 3, p_4 = 5, p_5 = 50, p_6 = 6$**. The corresponding matrix:** $A_1 = 5 \times 10, A_2 = 10 \times 3, A_3 = 3 \times 12, A_4 = 12 \times 5, A_5 = 5 \times 50, A_6 = 50 \times 6$**. According to the previous algorithm, for all** $m[i, j] = 0$**, we have:**

$$m[1, 2] = m[1, 1] + m[2, 2] + p_0p_1p_2 = 0 + 5 \times 10 \times 3 = 150$$

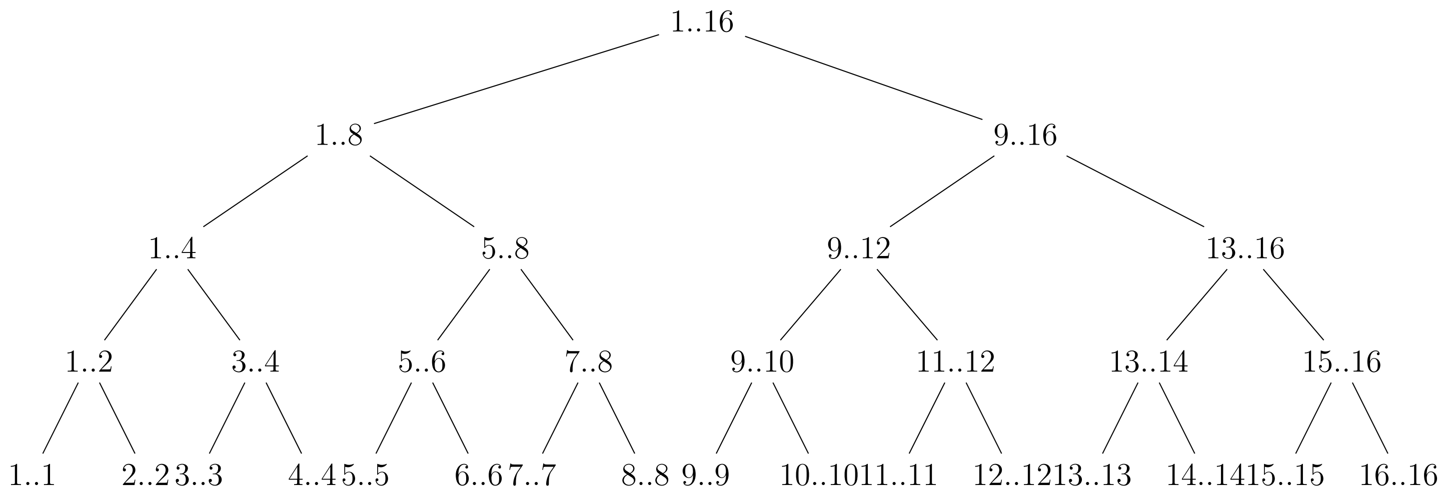
...

The m-table and s-table are shown as follows: .

$$\text{m-table} = \begin{array}{ccccccc} 6 & 0 & & & & & \\ 5 & 1500 & 0 & & & & \\ 4 & 1860 & 3000 & 0 & & & \\ 3 & 1770 & 930 & 180 & 0 & & \\ 2 & 1950 & 2430 & 330 & 360 & 0 & \\ 1 & 2010 & 1655 & 405 & 330 & 150 & 0 \\ i & 6 & 5 & 4 & 3 & 2 & 1 & j \end{array} \qquad \text{s-table} = \begin{array}{cccccc} 5 & 5 & & & & \\ 4 & 4 & 4 & & & \\ 3 & 4 & 4 & 3 & & \\ 2 & 2 & 2 & 2 & 2 & \\ 1 & 2 & 4 & 2 & 2 & 1 \\ i & 6 & 5 & 4 & 3 & 2 & j \end{array}$$

According to the $s-table$, the optimal parenthesization is $(A_1A_2)((A_3A_4)(A_5A_6))$

2 Exercise 15.3-2



An optimal problem must have for dynamic programming to apply is that the space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems. In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion. Dynamic programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table when it can be looked up when needed, using constant time per lookup. The MERGE-SORT procedure does not have overlapping subproblems, that is, all nodes of the recursion tree are distinct. Since we cannot reuse the results of recursive, the dynamic programming cannot improve the efficiency.

3 Exercise 15.3-5

The optimal substructure of section 15.1 whose recursion is shown below:

$$r_n = \max_{1\leq i\leq n}(p_i + r_{n-i})$$

Now, we are going to apply this recursion directly to this problem. Cut a rod of length n, assuming that the first cut is length i. When solving the subproblem, that is, when cutting rod of length $n-i$, it is assumed that the number of short rod of length i cut is exactly the maximum l_i. Hence, in the cutting scheme of the whole rod, there are l_i+1 short rods of length i, which exceeds the maximum limit of l_i. Therefore, applying this recursion directly may result in a solution that does not meet the constraints.
In other words, when solving the subproblem, that is, when cutting rod of length $n-i$, the number of short rods of length i cut cannot exceeds $l_i - 1$ (minus 1 is because a short rod with length i has been cut before solving the subproblem). Therefore, the subproblem is not only smaller than the original problem, but the constraints are also changed. It leads to a problem that even if a subproblem of the same size is solved, different results will be produced due to different constraints, that is, a subproblem of the same size may need to be solved multiple times due to the change of constraints. This does not conform to the principle of optimal substructure "the same subproblem only needs to be solved once".

4 Problem 15-5

4.1 a)

Assume that the optimal cost for transferring $x[1..i]$ where $0 \leq i \leq m$ to $y[1..j]$ where $0 \leq j \leq n$, and $x[1..0], y[1..0]$ is null.

case 1: if $i > 0$ and $j > 0$

Thinking of the last operation from $x[1..i]$ transfer to $y[1..j]$, there has six possible options. Note that some operations can only be performed if conditions are met.

1. Copy
Conditions: $x[i] = y[i]$
Costs: $d_1[i,j] = cost(copy) + c[i-1, j-1]$
2. Replace
It is necessary to assume that the cost of replace is greater than the cost of copy. Otherwise, we could use replace instead of copy so that copy will never be user at all.
Conditions: $x[i] \neq y[i]$
Costs: $d_2[i,j] = cost(replace) + c[i-1, j-1]$

3. Delete

Conditions: None

Costs: $d_3[i,j] = cost(delete) + c[i-1,j]$

4. Insert

Conditions: None

Costs: $d_4[i,j] = cost(insert) + c[i,j-1]$

5. Twiddle

Conditions: $i \leq 2, j \leq 2, x[i-1] = y[j], x[i] = y[j-1]$

Costs: $d_5[i,j] = cost(twiddle) + c[i-2, j-21]$

6. Kill

Kill can only be the last operation, so there must have $i = m$ and $j = n$ and the transformation-operation before kill is $x[1..k]$ to $y[1..n]$ where $0 \leq k \leq m$. Note that if the kill operation is executed, the number of characters processed by the x needs to be record.

Conditions: $i = m, j = n$

Costs: $d_6[i,j] = \min\limits_{0 \leq k \leq m} \{cost(kill) + c[k,n]\}$

The processing costs of the above six operations are calculated respectively, and the one with the minimum is selected as the optimal cost:

$$c[i,j] = \min_{1 \leq k \leq m6} \{d_k[i,j]\}$$

Case 2: if $i = 0$ or $j = 0$

1. $i = 0$, that is, $x[1..0]$ to $y[1..j]$

Since $x[1..0]$ is empty, we only can make j insert operation to get $y[1..j]$

$$c[0,j] = j * cost(insert)$$

2. $j = 0$, that is, $x[1..i]$ to $y[1..0]$

Since $y[1..0]$ is empty, if $i < m$ or $n > 0$, we only can make i delete operation; if $i = m$ and $n = 0$, then the kill operation can be performed in one step or delete operation can be performed in m steps, take the one with lowest cost.

$$c[i,0] = \begin{cases} i * cost(delete), & i < j \textbf{ or } n > 0 \\ \min\{cost(kill), i * cost(delete)\}, & i = m \textbf{ and } n = 0 \end{cases}$$

.

Algorithm 2 Dynamic programming algorithm that finds the edit distance

Input: Sequence X, sequence Y, m(the length of X), n(the length of Y), the cost of each operation

Output: The array of optimal cost C, the array of the last opeartion op, the number of processed character p

```
function EDIT-DISTANCE($X, Y, m, n, cost$)
    create 2 new 2-D arrays $C[0..m, 0..n]$ and $op[0..m, 0..n]$
    create a new 1-D array $d[1..6]$
    $C[0, 0] \leftarrow 0$
    $op[0, 0] \leftarrow 0$
    for $j = 1$ to $n$ do
        $C[0, j] \leftarrow j * cost[4]$                                ▷ insert
        $op[0, j] \leftarrow 4$
    end for
    for $i = 1$ to $m$ do
        $C[i, 0] \leftarrow j * cost[3]$                                ▷ delete
        $op[i, 0] \leftarrow 3$
    end for
    if $n = 0$ and $cost(kill) < C[m, 0]$ then
        $C[m, 0] \leftarrow cost[6]$
        $op[i, 0] \leftarrow 6$
        $p \leftarrow 0$
    end if
    for $i = 1$ to $m$ do
        for $j = 1$ to $n$ do
            for $k = 1$ to 6 do
                $d[k] \leftarrow \infty$
            end for
            if $X[i] = Y[j]$ then
                $d[1] = cost[1] + C[i-1, j-1]$                          ▷ copy
            else
                $d[2] = cost[2] + C[i-1, j-1]$                          ▷ replace
            end if
            $d[3] = cost[3] + C[i-1, j]$                                ▷ delete
            $d[4] = cost[4] + C[i, j-1]$                                ▷ insert
            if $i \geq 2$ and $j \geq 2$ and $X[i-1] = Y[j]$ and $X[i] = Y[j-1]$ then
                $d[5] = cost[5] + C[i-2, j-2]$                          ▷ twiddle
            end if
            if $i = m$ and $j = n$ then
                for $k = 0$ to $m - 1$ do
                    if $cost[6] + C[k, n] < d[6]$ then
                        $d[6] = cost[6] + C[k, n]$                      ▷ kill
                        $p = k$
                    **end if**
                **end for**
            **end if**
            $C[i, j] \leftarrow \infty$
            **for** $k = 1$ to $6$ **do**
                **if** $d[k] < C[i, j]$ **then**
                    $C[i, j] \leftarrow d[k]$
                    $op[i, j] \leftarrow k$
                **end if**
            **end for**
        **end for**
    **end for**
    **return** $C, OP$ and $p$
    **end function**
```

Pseudocode 3 Print the operation

Input: The array op, the number of processed character p, i(the length of X), j(the length of Y)

```
**function** PRINT-OPERATION$(op, p, i, j)$
    If $op[i, j] = 0$ **return**
    Else if $op[i, j] = 1$
        PRINT-OPERATION$(op, p, i - 1, j - 1)$
        print *copy* $i \to j$
    Else if $op[i, j] = 2$
        PRINT-OPERATION$(op, p, i - 1, j - 1)$
        print *replace* $i \to j$
    Else if $op[i, j] = 3$
        PRINT-OPERATION$(op, p, i - 1, j)$
        print *delete* $i$
    Else if $op[i, j] = 4$
        PRINT-OPERATION$(op, p, i, j - 1)$
        print *insert* $j$
    Else if $op[i, j] = 5$
        PRINT-OPERATION$(op, p, i - 2, j - 2)$
        print *twiddle* $i - 1 \to j, i \to j - 1$
    Else
        PRINT-OPERATION$(op, p, p, j)$
        print *kill* $p$
**end function**
```

The running time of the algorithm is $O(mn)$.

4.2 b)

The DNA alignment problem can be transfermed into an edit distance problem by the following method:

If $x'[i] = y'[j]$ and neither is a space, it is same as copy operation, the $score(copy) = +1$

If $x'[i] \neq y'[j]$ and neither is a space, it is same as replace operation, the $score(replace) = -1$

If $x'[i] = space$, it is same as insert operation, the $score(insert) = -2$

If $y'[i] = space$, it is same as delete operation, the $score(delete) = -2$

The twiddle and kill operations cannot be used. The pseudocode is mostly same. Besides, the edit distance problem is to minimize the cost, while the problem of two DNA sequences is to maximize the score.

5 Cross-country road-trip problem

Since the problem is similar with chess board problem, we use the Dynamic programming. Table of costs given as a matrix $p_{ij}; i = 1 : n, j = 1 : n$

$C[i,j]$ = minimum cost from row 1 to cell $[i,j]$ and $C[i,j] = p_{ij}$ if $i = 1$.

```
1: $C[1,j] = P_{1j}$ for all j
2: for $i = 2$ to $n$ do
3:     for $j = 1$ to $n$ do
4:         $C[i,j] \leftarrow p_{ij} + \min(C[i-1,j-1], C[i-1,j], C[i-1,j+1])$
5:         $y \leftarrow x_i + 1$
6:     end for
7: end for
8: return $C$
```

After the array C computed, we will trace the solution. Find the minimum column $j = argmin(C[m,:])$ on the last row, output the cell $C[n,j]$.

```
1: $i \leftarrow n$
2: while $i > 1$ do
3:     $j_b elow \leftarrow argmin(C[i-1,j-1], C[i-1,j], C[i-1,j+1])$
4:     output cell $C[i-1, j_b elow]$
5:     $i \leftarrow i - 1$
6:     $j \leftarrow j_b elow$
7: end while
```

Since the outer loop has n iterations, inner loop has n iterations, and we also have 3 comparisons that is constant time, our algorithm total running time is $\Theta(n^2)$.

6 Exercise 15.4-5

Assume that the sequence of n numbers is $X = \{x_0, x_1, ..., x_n\}$, makes a copy of X called Y and sorted T by increasing order $Y = \{y_0, y_1, ..., y_n\}$. 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; $c[n, n]$ contains the length of an longest monotonically increasing subsequence of X.

As in the matrix-chain multiplication problem, our recursive solution to the LIS 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 LIS 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 monotonically increasing subsequence

```
function LIS-LENGTH($X, Y$)
    let $b[1..n, 1..n]$ and $c[0..n, 0..n]$ be new tables
    for $i = 1$ to $n$ do
        $c[i, 0] \leftarrow 0$
    end for
    for $j = 0$ to $n$ do
        $c[0, j] \leftarrow 0$
    end for
    for $i = 1$ to $n$ do
        for $j = 1$ to $n$ do
            if $x_i == y_i$ then
                $c[i, j] \leftarrow c[i-1, j-1] + 1$
                $b[i, j] = \nwarrow$
            else
                if $c[i-1, j] \geq c[i, j-1]$ then
                    $c[i, j] \leftarrow c[i-1, j]$
                    $b[i, j] = \uparrow$
                else
                    $c[i, j] \leftarrow c[i, j-1]$
                    $b[i, j] = \leftarrow$
                end if
            end if
        end for
    end for
    return $c$ and $b$
end function
```

The b table returned by LIS-Length enables us to quickly construct an LIS of $X = \{x_0, x_1, ..., x_n\}$ and $Y = \{y_0, y_1, ..., y_n\}$. We simply begin at $v[n, n]$ and trace through the tbale by following the arrows. Whenever we encounter a "$\nwarrow$" in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LIS that Lis-Length found. With this method, we encounter the elements of this LIS in the proper, forward order. The initial call is $PRINT - LIS(b, X, i, j)$. .

Algorithm 5 Print the longest monotonically increasing subsequence

```
function PRINT-LIS($b, X, i, j$)
    if $i == 0$ or $j == 0$ then return
    end if
    if $b[i, j] ==$ " $\nwarrow$ " then
        PRINT-LIS($b, X, i - 1, j - 1$)
        print $x_i$
    else
        if $b[i, j] ==$ " $\uparrow$ " then
            PRINT-LIS($b, X, i - 1, j$)
        else
            PRINT-LIS($b, X, i, j - 1$)
        end if
    end if
end function
```

The LIS-Length is based on an exponential-time recursive algorithm to compute the length of an LIS of sequences, so the LIS-Length takes $\Theta(n^2)$ time, since each table entry takes $\Theta(1)$ time to compute.
The procedure of PRINT-LIS takes time $O(n + n) = O(2n)$, since it decrements at least one of i and j in each recursive call.

Therefore, the running time of LIS problem is $\Theta(n^2)$.