CIS 5511. Programming Techniques

Dynamic Programming

1. The idea

When a problem is solved by reducing into subproblems, sometimes the same subproblem appears multiple times.

For example, Fibonacci numbers are defined by the following recurrence:

```
F(0) = 0
F(1) = 1
F(i) = F(i-1) + F(i-2)
```

It is easy to directly write a recursive algorithm according to this definition:

```
Fibonacci-R(i)
if i == 0
    return 0
if i == 1
    return 1
return Fibonacci-R(i-1) + Fibonacci-R(i-2)
```

This algorithm contains repeated calculations, so is not efficient. For example, to get F(4), it calculates F(3) and F(2) separately, though in calculating F(3), F(2) is already calculated. The complexity of this algorithm is O(2^i).

It is easy to see that the algorithm can be rewritten in a "bottom-up" manner with the intermediate results saved in an array. This gives us a "dynamic programming" version of the algorithm:

```
Fibonacci-DP(i)
    A[0] = 0        // assume 0 is a valid index
    A[1] = 1
    for j = 2 to i
        A[j] = A[j-1] + A[j-2]
    return A[i]
```

Since each subsolution is only calculated once, the complexity is O(i).

From this simple example, we see the basic elements of *dynamic programming*, that is, though the solution is still defined in a "top-down" manner (from solution to subsolution), it is built in a "bottom-up" manner (from subsolution to solution) with the subsolutions remembered to avoid repeated calculations.

Memoization is variation of dynamic programming that keeps the efficiency, while maintaining a top-down structure.

```
Fibonacci-M(i)
A[0] = 0        // assume 0 is a valid index
A[1] = 1
for j = 2 to i
    A[j] = -1
```

```
5  Fibonacci-M2(A, i)

Fibonacci-M2(A, i)
1  if A[i] < 0
2      A[i] = Fibonacci-M2(A, i-1) + Fibonacci-M2(A, i-2)
3  return A[i]
```

The complexity of this algorithm is also O(i).

Dynamic programming is often used in optimization where the optimal solution corresponds to a special way to divide the problem into subproblems. When different ways of division are compared, it is quite common that some subproblems are involved in multiple times, therefore dynamic programming provides a more efficient algorithm than (direct) recursion.

The complexity of dynamic programming comes from the need of keeping intermediate results, as well as remembering the structure of the optimal solution, i.e., where the solution comes from in each step.

2. Example: planning matrix-chain multiplication

The following algorithm multiplies two matrices A (p-by-q) and B (q-by-r) to get C (p-by-r):

```
RECTANGULAR-MATRIX-MULTIPLY(A, B, C, p, q, r)
for $i = 1$ to $p$
    for $j = 1$ to $r$
        for $k = 1$ to $q$
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
```

The result C contains p*r elements, each calculated from q scalar multiplications and q − 1 additions. If we count the number of scalar multiplications and use it to represent running time, then for such an operation, it is p*q*r.

To calculate the product of a matrix-chain $A_1A_2...A_n$, n − 1 matrix multiplications are needed. Though in each step any pair of matrices can be multiplied, different choices have different costs. For matrix-chain $A_1A_2A_3$, if the three have sizes 10-by-2, 2-by-20, and 20-by-5, respectively, then the cost of $(A_1A_2)A_3$ is 10*2*20 + 10*20*5 = 1400, while the cost of $A_1(A_2A_3)$ is 2*20*5 + 10*2*5 = 300.

In general, for matrix-chain $A_i...A_j$ where each A_k has dimensions P_{k-1}-by-P_k, then the minimum cost of the product *m[i,j]* can be obtained by looking for the best way to cut it into $A_i...A_k$ and $A_{k+1}...A_j$. Therefore we have

```
m[i, j] = 0                                                 if i = j
          min{ m[i,k] + m[k+1,j] + $P_{i-1}P_kP_j$ }              if i < j
```

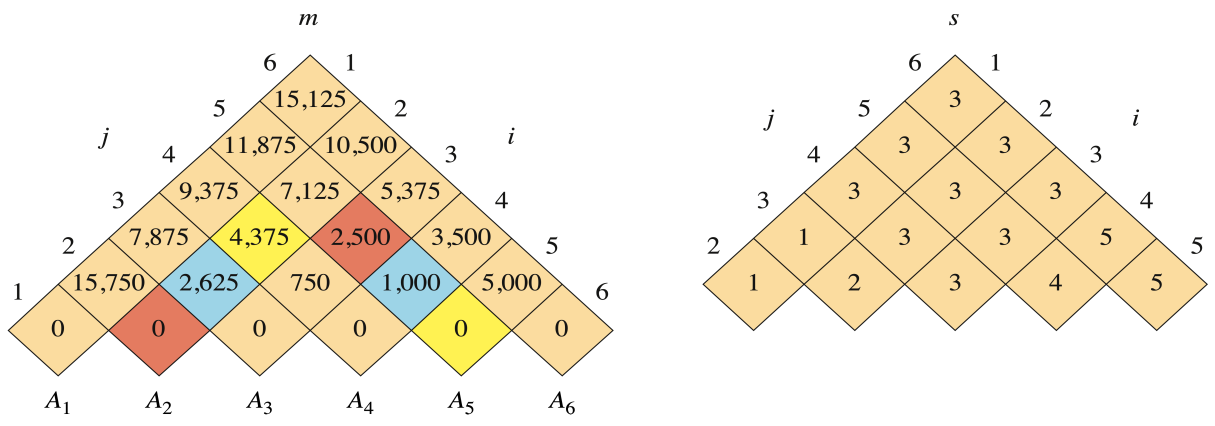
Since certain *m[i,j]* will be used again and again, we use dynamic programming to find the minimum number of operation by calculating this *m* value for sub-chains with increasing length, and using another matrix *s* to keep the cutting point *k* for each *m[i,j]*.

```
MATRIX-CHAIN-ORDER(p, n)
let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables
for $i = 1$ to $n$                        // chain length 1
    $m[i,i] = 0$
for $l = 2$ to $n$                        // $l$ is the chain length
    for $i = 1$ to $n - l + 1$            // chain begins at $A_i$
        $j = i + l - 1$                   // chain ends at $A_j$
        $m[i,j] = \infty$
        for $k = i$ to $j - 1$            // try $A_{i:k} A_{k+1:j}$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
            if $q < m[i,j]$
                $m[i,j] = q$              // remember this cost
                $s[i,j] = k$              // remember this index
return $m$ and $s$
```

Please note that this algorithm is used to determine the best order to carry out the matrix multiplications, without doing the multiplications themselves.

Example: Given the following matrix dimensions:
A_1 is 30-by-35
A_2 is 35-by-15
A_3 is 15-by-5
A_4 is 5-by-10
A_5 is 10-by-20
A_6 is 20-by-25
P: [30, 35, 15, 5, 10, 20, 25]

then the output of the program is



which means that $A_1A_2A_3A_4A_5A_6$ should be calculated as $(A_1(A_2A_3))((A_4A_5)A_6)$, and the number of scalar multiplication is 15,125.

This problem can be solved by either a bottom-up dynamic programming algorithm or a top-down, memoized dynamic-programming algorithm. The running time of both algorithms is $\Theta(n^3)$, and the algorithm needs $\Theta(n^2)$ space.

3. Longest Common Subsequence

From a given sequence, a *subsequence* can be obtained by removing any number of its elements. For sequence X and Y, their Longest Common Subsequence (LCS) is not necessarily continuous in either X or Y. For example, if X is ABCBDAB and Y is BDCABA, a LCS of them is BCBA with length 4, and BCAB is another one. So the LCS is not unique, though their lengths are the same.

LCS can be recursively constructed. Let us define c[i, j] to be the length of a LCS of the sequences X_i and Y_j (to index i and j in X and Y, respectively), then

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

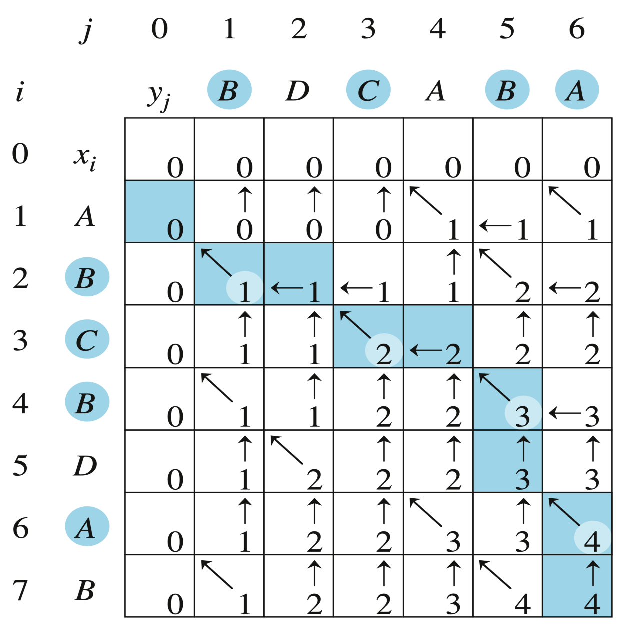
Algorithm:

```
LCS-LENGTH(X, Y, m, n)
 1  let b[1 : m, 1 : n] and c[0 : m, 0 : n] be new tables
 2  for i = 1 to m
 3      c[i, 0] = 0
 4  for j = 0 to n
 5      c[0, j] = 0
 6  for i = 1 to m          // compute table entries in row-major order
 7      for j = 1 to n
 8          if x_i == y_j
 9              c[i, j] = c[i − 1, j − 1] + 1
10              b[i, j] = "↖"
11          elseif c[i − 1, j] ≥ c[i, j − 1]
12              c[i, j] = c[i − 1, j]
13              b[i, j] = "↑"
14          else c[i, j] = c[i, j − 1]
15              b[i, j] = "←"
16  return c and b
```

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return               // the LCS has length 0
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i            // same as y_j
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Time cost: Θ(m*n). Space cost: Θ(m*n).

Example:



	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

4. Optimal Binary Search Tree

If the keys in a constant BST have different probabilities to be searched, then a balanced tree may not give the lowest average cost (though it gives the lowest worst cost). Using dynamic programming, BST with optimal average search time can be built by comparing the expected cost of each candidate (i.e., with each node as root), in a way similar to the matrix-chain-order algorithm. The input of this algorithm are the probabilities for each key values to be searched, as well as those of the other values not in the tree, for each ranges between the values in the tree.