# Solutions to Problem Set 6

## Ekkapot Charoenwanit Efficient Algorithms

#### Problem 6.1. Fibonacci numbers

1) The pseudocode given in Algorithm 1 implements a function computing the Fibonacci numbers in a top-down manner with the help of memoization.

### Algorithm 1 computes the Fibonacci numbers with memoization.

```
1: procedure FIB(n, F[1...n])
2: if n \le 1 then
3: return n
4: else
5: if F[n] > 0 then
6: return F[n]
7: F[n] = FIB(n-1, F) + FIB(n-2, F)
8: return F[n]
```

What is the recursion depth and what is the space complexity of FIB(n)?

**Solution:** The recursion depth is the same as the number of distinct subproblems generated by the call Fib(n). Figure 1 shows that there are exactly n distinct subproblems. Therefore, the recursion depth is n. Time complexity T(n) is linear in the number of distinct subproblems times work per subproblem. Work per subproblem is  $\Theta(1)$ . Thus,  $T(n) = \Theta(n) \cdot \Theta(1) = \Theta(n)$ .

Space complexity S(n) is linear in the number of elements of the table F[1...n] and is also linear in the recursion depth, which determines the maximum number of nested stack frames during the execution of Fib(n). Thus,  $S(n) = \Theta(n)$ .

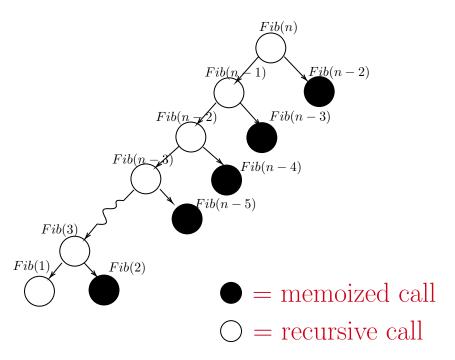


Figure 1: Memoized Fibonacci

2) Optimize the bottom-up implementation given in Algorithm 2 such that it uses  $\Theta(1)$  space.

### Algorithm 2 computes the Fibonacci numbers bottom-up.

```
1: procedure FIB(n)

2: F = NEW TABLE[0...n]

3: F[0] = 0

4: F[1] = 1

5: for i = 2 \rightarrow n do

6: F[i] = F[i-1] + F[i-2]

7: return F[n]
```

**Solution:** Algorithm 2 computes FIB(n) by unnecessarily allocating  $n + 1 = \Theta(n)$  array cells. In fact, we can compute FIB(n) with only **two** arrays cells, hence  $\Theta(1)$  space, as shown in Algorithm 3.

### **Algorithm 3** computes the Fibonacci numbers bottom-up using $\Theta(1)$ space.

```
1: procedure FIB(n)
       F = \text{NEW TABLE}[0...1]
2:
       F[0] = 0
3:
       F[1] = 1
4:
       for i = 2 \rightarrow n do
5:
           \mathsf{TEMP} = F[1]
6:
           F[1] = F[0] + F[1]
7:
           F[0] = \text{TEMP}
8:
       return F[1]
```

### Problem 6.2. Matrix Chain Multiplication

1) Find an optimal parenthesization for the chain product of 5 matrices with dimensions  $6 \times 7$ ,  $7 \times 8$ ,

 $8 \times 3$ ,  $3 \times 10$  and  $10 \times 6$  in a bottom-up approach.

#### **Solution: Base Cases:**

(0): 
$$M[1,1] = M[2,2] = M[3,3] = M[4,4] = M[5,5] = 0$$

#### **Inductive Cases:**

The table is filled in the following order:  $(1) \rightarrow (2) \rightarrow (3) \rightarrow (4)$ 

- (1): M[1,2], M[2,3], M[3,4], M[4,5]
- (2): M[1,3], M[2,4], M[3,5]
- (3): M[1,4], M[2,5]
- (4): M[1,5]

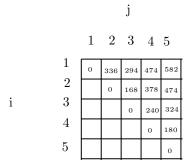


Figure 2: Bottom-up Table

M[1,3] is the optimal number of scalar multiplications for the matrix chain  $A_1 \cdot A_2 \cdot A_3$ . The parenthesization  $A_1 \cdot (A_2 \cdot A_3)$  yields the optimal value of 294.

M[2,4] is the optimal number of scalar multiplications for the matrix chain  $A_2 \cdot A_3 \cdot A_4$ . The parenthesization  $(A_2 \cdot A_3) \cdot A_4$  yields the optimal value of 378.

M[3,5] is the optimal number of scalar multiplications for the matrix chain  $A_3 \cdot A_4 \cdot A_5$ . The parenthesization  $A_3 \cdot (A_4 \cdot A_5)$  yields the optimal value of 324.

M[1,4] is the optimal number of scalar multiplications for the matrix chain  $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ . The parenthesization  $(A_1 \cdot A_2 \cdot A_3) \cdot A_4$  yields the optimal value of 474.

M[2,5] is the optimal number of scalar multiplications for the matrix chain  $A_2 \cdot A_3 \cdot A_4 \cdot A_5$ . The parenthesization  $(A_2 \cdot A_3) \cdot (A_4 \cdot A_5)$  yields the optimal value of 474.

M[1,5] is the optimal number of scalar multiplications for the matrix chain  $A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5$ . The parenthesization  $(A_1 \cdot A_2 \cdot A_3) \cdot (A_4 \cdot A_5)$  yields the optimal value of 582.

The optimal number of scalar multiplications can be harvested from M[1,5]. Here, the optimal number of scalar multiplications is 582, and an optimal parenthesization is  $(A_1 \cdot (A_2 \cdot A_3))(A_4 \cdot A_5)$ .

2) Show that the number of ways of parenthesization C(n) is  $\Omega(2^n)$ , where n is the matrix chain length. You may use induction to show that  $C(n) \ge c \cdot 2^n$  for some  $c \in \mathbb{R}^+$  and  $n_0 \in \mathbb{Z}^+$  for all  $n \ge n_0$ .

**Proof:** We know that

$$C(n) = \begin{cases} 1 & n = 1\\ \sum_{k=1}^{n-1} C(k)C(n-k) & n \ge 2 \end{cases}$$

We would like to show that

$$C(n) = \Omega(2^n)$$

Suppose that we choose  $n_0 = 1$  and  $c = \frac{1}{4}$ .

In other words, we want to show

$$C(n) \ge \frac{1}{4} \cdot 2^n \quad \forall n \ge 1$$

Base Case: n=1

$$C(1) \ge \frac{1}{4} \cdot 2^1 = \frac{1}{2}$$

Since C(1) = 1, the proposition is true when n = 1.

**Induction Hypothesis:** Assume true for k = 1, 2, 3, ..., n - 1.

$$C(k) \ge \frac{1}{4} \cdot 2^k$$

Inductive Step:

$$C(n) = \sum_{k=1}^{n-1} C(k)C(n-k)$$

$$\geq C(1)C(n-1) + C(n-1)$$

$$= 2C(1)C(n-1)$$

$$= 2C(n-1) \qquad [C(1) = 1]$$

$$\geq 2 \cdot \frac{1}{4} \cdot 2^{(n-1)} \qquad [Invoking I.H. at n-1]$$

$$= \frac{1}{4} \cdot 2^{n}$$

Thus, we have shown that the proposition is true for all  $n \ge 1$  with  $c = \frac{1}{4}$ . Therefore,  $C(n) = \Omega(2^n)$ .  $\square$ 

3) Calculate the exact number of distinct subproblems for a matrix chain of length n.

**Solution:** Let M[i,j] be the minimum number of scalar multiplications required to compute the matrix chain  $A_i \cdot A_{i+1} \cdot ... A_{j-1} \cdot A_j$ .

For i = 1, there are n ways to choose j.

For i = 2, there are n - 1 ways to choose j.

For i = 3, there are n - 2 ways to choose j.

• • •

For i = n, there are 1 ways to choose j.

The number of ways to choose i and j is the number of distinct subproblems. Therefore, there are  $n+(n-1)+(n-2)+...+1=\frac{n(n+1)}{2}$  distinct subproblems.  $\square$ 

4) Does the maximum matrix chain problem also exhibit optimal substructure? If it is the case, prove your claim using a cut-and-paste argument.

**Proof:** The maximum chain problem also exhibits optimal substructure as we will show using a cut-and-paste argument as follows.

It is given that M[i,j] is an optimal solution to  $A_i...A_j$ . Suppose the solution  $M[i,k] = m_{i,k}$  to the prefix subchain  $A_i...A_k$  is not optimal. We can then replace this solution to  $A_i...A_k$  with a better solution (better means larger in value)  $M[i,k] = m'_{i,k} > m_{i,k}$  to obtain a better solution  $M[i,j] = m'_{i,j}$  to  $A_i...A_j$ :

$$m'_{i,j} = m'_{i,k} + m_{k+1,j} + p_i p_k p_j > m_{i,j}$$

,which contradicts the optimality of the solution  $m_{i,j}$  to  $A_i...A_j$ . An identical cut-and-paste argument can be used to show optimality of the suffix subchain  $A_{k+1}...A_j$ .  $\square$ .

### Problem 6.3. Longest Common Subsequence

1) Give a memoized version of LCS-LENGTH that runs in  $\mathcal{O}(mn)$  time.

**Solution:** The table c in Algorithm 4 is assumed to be a **global** variable, and all c[i,j] are initialized to -1. Given two sequences X and Y of length m and n, respectively, we run MEMOIZED-LCS(X,Y,m,n) to compute the length of a longest-common subsequence.

### Algorithm 4 Longest-Common Subsequence

```
1: procedure MEMOIZED-LCS(X, Y, i, j)
       if c[i, j] > -1 then
 2:
           return c[i,j]
 3:
       if i = 0 \lor j = 0 then
 4:
           c[i,j] = 0
 5:
           return 0
 6:
       if X[i] == Y[i] then
 7:
           c[i, j] = 1 + \text{Memoized-LCS}(X, Y, i - 1, j - 1)
 8:
9:
           return c[i,j]
       c[i,j] = \max(\text{Memoized-LCS}(X,Y,i-1,j), \text{Memoized-LCS}(X,Y,i,j-1))
10:
       return c[i, j]
11:
12:
```

Since there are  $\Theta(mn)$  distinct subproblems, and each subproblem requires  $\mathcal{O}(1)$  time (addition, max etc all require constant time), the algorithm takes O(mn) time with memoization.

2) What is the maximum recursion depth of LCS?

**Solution:** Observe that, at each recursive call LCS(X,Y,i,j), either i or j or both decrease by one.

An unlucky case can happen, for example, when i decreases by one at each recursive call whereas j stays at n all the way until i = 1, from which point i stays at 1 whereas j decreases by one at each recursive call.

This means that it takes at most m + n recursive calls to bring either i or j (whichever one becomes 0 first does not matter) to 0, which is a base case, where recursion bottoms out.

Thus, this means that the maximum number of recursive calls is m + n, which means the maximum recursion depth is also m + n.