You're expected to work on the discussion problems before coming to the lab. Discussion session is not meant to be a lecture. TA will guide the discussion and correct your solutions if needed. We may or may not release 'official' solutions. If you're better prepared for discussion, you will learn more. TAs will record names of the students who actively engage in discussion and report them to the instructor; they are also allowed to give some extra points to those students at their discretion. The instructor will factor in participation in final grade.

1. (Basic) Recall that the The Fibonacci Sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, $\cdots$. Formally, $f(0) = 0, f(1) = 1$ and $f(i) = f(i-1) + f(i-2)$ for all $i \geq 2$, where $f(i)$ denotes the ith Fibonacci number.

   We saw the following naive pseudo-code/implementation of the function returns the n the Fibonacci number:

   ```
   int F(n)
     if n == 0 return 0
     if n == 1 return 1
     return F(n-2) + F(n-1)
   ```

   However, the running time is exponential in n. We can improve the running time to $O(n)$ using DP. There are typically two equivalent ways to implement a DP approach, i) *bottom-up method* and ii) *top-down with memoization*. Give a pseudo-code of each implementation.

   Sol.

   ```
   bottom up:
   int F(n)
     Array A[0 ... n]
     A[0] = 0, A[1] = 1
     for i = 2; i <= n ; i++
        A[i] = A[i-1] + A[i-2]
     return A[i]

   top down:
   int F(n)
     Array A[0 ... n]
     A[0] = 0, A[1] = 1
     A[2] = A[3]  = ... = A[n] = - infinity
     return Aux-F(A, n)

   int Aux-F(n)
     if A[n] >=  0 return A[n]
     A[n] = Aux-F(A, n-1) + Aux-F(A, n-2)
     return A[n]
   ```

(Note: If you're asked to give a dynamic programming and decided to use a bottom-up method, you can describe it as follows: We set up a DP table $a[0], a[1], ..., a[n]$ where $a[i]$ denotes the ith Fibonacci number. Using the given recurrence, we fill out the table in the order of $a[0], a[1], ..., a[n]$, and return $a[n]$. If you're asked to analyze the running time, you can say: there're $O(n)$ table entries, and computing each table entry takes $O(1)$ time, so the running time is $O(n)$.)

2. (Basic) Consider a sequence of integers defined by the following recurrence: $f(0) = 0, f(1) = 1$, and $f(i) = f(i-1) + f(\lfloor i/2 \rfloor)$ for $i \geq 2$. We would like to compute $f(n)$ using a bottom-up DP method. Give a pseudo-code.

   Sol. Omitted.

3. (Basic) Consider a sequence of integers defined by the following recurrence: $f(0) = 0, f(1) = 1$, and $f(i) = f(i-1) + f(n-2) + f(n-3) + ... + f(1) + f(0)$ for $i \geq 2$. We would like to compute $f(n)$ using a bottom-up DP method. Describe the DP and analyze its running time.

   Sol. Algorithm description: We set up a DP table with entries $f[0], f[1], ..., f[n]$, where $f[i]$ is the entry for $f(i)$. We fill the entries in increasing order of i using the recursion. At the end, we return $f[n]$. omitted. $O(n)$ entires. $O(n)$ time for each entry computation. RT: $O(n^2)$.

4. (Intermediate) Consider a modification of the rod-cutting problem (see CLRS Ch 15.1 page 360) in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. For simplicity, we will be only interested in computing the optimum.

   (a) Recall that if $c = 0$, then we would have the same recursion we had for the original rod-cutting problem: $r_n = \max_{1 \leq i \leq n}(p_i + r_{n-i})$. How would you change this recursion?

   Sol. $r_n = \max\{\max_{1 \leq i \leq n-1}(p_i + r_{n-i} - c), p_n\}$.

   (b) Give a dynamic-programming algorithm to solve this modified problem.

   Sol. Set up array (DP table entries) $r[0...n]$. Compute the entries using the recursion in this order, $r[0], r[1], ..., r[n]$. Return $r[n]$.

5. (Advanced) Limited-Rod-Cutting (again, see CLRS Ch 15.1 page 360). Let's consider another variant of the rod-cutting problem. The input is exactly the same, $p_1, p_2, \cdots p_n$. The only difference is that we're allowed to make *at most 10* cuts. In other words, you want to maximize your revenue by cutting a rod of length n into at most 11 pieces.

Our goal is to give a $O(n^2)$ time algorithm for this task. As before, let's focus on computing the optimum.

(a) Give a simple $O(n^{10})$ time algorithm.

Sol. There are at most $O(n^{10})$ ways of cutting. So, we can solve it in a brute-force manner.

(b) We need to extend subproblems. Let $r_{i,k}$ denote the max revenue you can get from a rod of length i by making at most k cuts. We will be interested in computing $r_{n,10}$. Give a recurrence for computing $r_{i,k}$.

Sol. $r_{i,k} = \max_{1 \le j \le i}(p_j + r_{i-j,k-1})$ for all $i \ge 1$ and $k \ge 1$; and $r_{i,0} = p_i$ for all $i \ge 1$; and $r_{i,k} = 0$ if $i = 0$.

(c) Give a DP. You can simply state how the DP table is defined, and in which order you will compute the DP entries. What is the number of DP entries? How much time do you need to compute each entry? What is the running time of your dp?

Sol.

Table entry $r_{i,k}(0 \le i \le n, 0 \le k \le 10)$ denotes the max revenue one can get from a rod of length i by making at most k cuts. We compute the entries using double for loops:
for $0 \le k \le 10$
for $0 \le i \le n$
compute $r_{i,k}$ using the recurrence.
Return $r_{n,10}$.

$O(n)$ entries. $O(n)$ time for each entry computation. The running time is $O(n^2)$.

(d) In the above, we only computed $r_{n,10}$. How do you find a solution that gives the revenue?

Sol. We compute $s[i,k]$, which indicates the length of the first piece in an optimal solution for length i with at most k cuts. Then,

```
k = 10
while(n >=0 && k >=0)
  output s[n,k]
  n  = n - s[n, k]
  k = k-1
```

6. (Basic) Matrix Chain Multiplication. Suppose we are given four matrices, A_1, A_2, A_3, A_4 with $p_0 = 3, p_1 = 2, p_2 = 3, p_3 = 2, p_4 = 3$. Fill out the DP table of entries $m[i,j]$, where $1 \le i \le j \le 4$, and return $m[i,j]$; see the lecture slides or textbook for the

definition of $m[i, j]$. Also show all $s[i, j]$, where $1 \leq i \leq j \leq 4$ and show an optimal full parenthesization using them.

Sol. $m[1, 1], m[2, 2], m[3, 3], m[4, 4] = 0$ as they are base cases.

$m[1, 2] = p_0 p_1 p_2 = 18$, $m[2, 3] = p_1 p_2 p_3 = 2 \cdot 3 \cdot 2 = 12$, $m[3, 4] = p_2 p_3 p_4 = 18$.

$m[1, 3]$: There are two cases depending on the value of k in the recursion.
case 1. $k = 1$. $m[1, 1] + m[2, 3] + p_0 p_1 p_3 = 0 + 12 + 12 = 24$
case 2. $k = 2$. $m[1, 2] + m[3, 3] + p_0 p_2 p_3 = 18 + 0 + 18 = 36$
Taking the minimum, we have $m[1, 3] = 24$ and we set $s[1, 3] = 1$

$m[2, 4]$: There are two cases depending on the value of k in the recursion.
case 1. $k = 2$. $m[2, 2] + m[3, 4] + p_1 p_2 p_4 = 0 + 18 + 18 = 36$
case 2. $k = 3$. $m[2, 3] + m[4, 4] + p_1 p_3 p_4 = 12 + 0 + 12 = 24$
Taking the minimum, we have $m[2, 4] = 24$ and we set $s[2, 4] = 3$

$m[1, 4]$: There are three cases depending on the value of k in the recursion.
case 1. $k = 1$. $m[1, 1] + m[2, 4] + p_0 p_1 p_4 = 0 + 24 + 18 = 32$
case 2. $k = 2$. $m[1, 2] + m[3, 4] + p_0 p_2 p_4 = 18 + 18 + 27 = 63$
case 3. $k = 3$. $m[1, 3] + m[4, 4] + p_0 p_3 p_4 = 24 + 0 + 18 = 32$
Taking the minimum, we have $m[1, 4] = 36$ and we set $s[1, 4] = 1$ (or 3)

So, $(A_1((A_2 A_3)A_4))$ is an optimal solution, which can computed using $m[1, 4] = 32$ multiplications.

7. (Basic) We would like to find an LCS of two sequences $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$ using the DP we learned. Fill out the DP table as shown on page 395 of the textbook.

Sol.

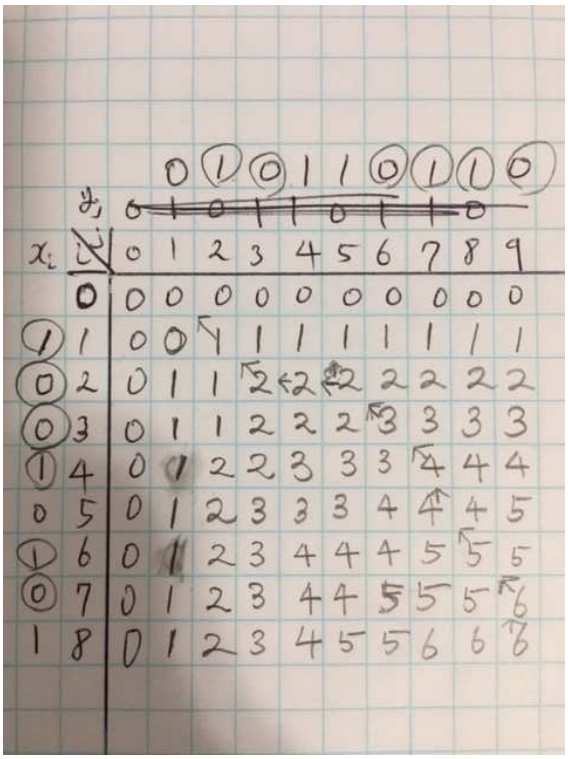

Figure 1: LCS

8. (Advanced) Give an algorithm for computing an LCS of three given sequences, X, Y, and Z; for simplicity, assume that all the tree sequences are of length n. You have to define subproblems, give a recursion, and explain in which order the DP table is filled out. What is the running time of your algorithm?

   Sol. Let X_i denote X's prefix of length i. Y_j and Z_k are similarly defined. Define $c[i, j, k]$ as the length of LCS of X_i, Y_j, Z_k. Then, we have the following recursion:

$$c[i,j,k] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \text{ or } k = 0 \\ c[i-1, j-1, k-1] + 1 & \text{if } i, j, k \geq 1 \text{ and } x_i = y_j = z_k \\ \max\{c[i-1,j,k], c[i,j-1,k], c[i,j,k-1]\} & \text{otherwise} \end{cases} \tag{1}$$

   For example, for k = 0 to n
   for i = 0 to n
   for j = 0 to n
   compute $c[i, j, k]$ using the recursion.

   RT: $O(n^3)$ since there are $O(n^3)$ DP entries and computing each entry takes $O(1)$ time.