- See previous practice problem sets for instructions.
- I repeat one very important instruction: You must explicitly state all *non-trivial* assumptions that you make. When in doubt, over-communicate.

1. For $n \in \mathbb{N}$ the n^{th} Fibonacci number F_n is defined by: $F_0 = F_1 = 1, F_i = F_{i-1} + F_{i-2}$; $i \geq 2$.

   (a) Translate the following pseudocode for computing F_n into code in your favourite procedural programming language, and *time* it[1] on inputs $n = 35, n = 40, n = 45, n = 50$.

```
SimpleFib($n$)
if $n \leq 1$
    return 1
$F1 \leftarrow$ SimpleFib($n - 1$)
$F2 \leftarrow$ SimpleFib($n - 2$)
return $F1 + F2$
```

   (b) The following pseudocode is what we get from a straightforward memoization of SimpleFib. Translate this pseudocode into code in your favourite procedural programming language, and time it on inputs $n = 35, n = 40, n = 45, n = 50$.

```
SmartFib($n$)
if $n \leq 1$
    return 1
$F \leftarrow$ a global array of length $n + 1$
$F[0] = F[1] = 1$
for $i \leftarrow 2$ to $n$
    $F[i] = 0$
$Fn \leftarrow$ FibHelper($n$)
return $Fn$
```

```
FibHelper($i$)
if $F[i] \neq 0$
    return $F[i]$
$F1 \leftarrow$ FibHelper($i - 1$)
$F2 \leftarrow$ FibHelper($i - 2$)
$F[i] \leftarrow F1 + F2$
return $F[i]$
```

[1]That is: find the clock-time taken by the code.

Do you notice a significant difference? If you would like to *really* understand what is going on here: Try adding print statements to inspect the arguments passed in to the recursive calls in lines 3 and 4 of FIBHELPER.

2. Recall the following recursive algorithm that we saw in class, for solving the ROD CUTTING problem. Here n is the length of the rod to be cut up and sold, and P is an array of selling prices for different lengths of rod[2].

```
CUTROD($n, P$)
1  if $n == 0$
2      return 0
3  $maxRevenue \leftarrow -1$
4  for $i \leftarrow 1$ to $n$
5      $iFirstRevenue = P[i] + $ CUTROD($(n-i), P$)
6      if $iFirstRevenue > maxRevenue$
7          $maxRevenue \leftarrow iFirstRevenue$
8  return $maxRevenue$
```

(a) Prove that CUTROD correctly solves the ROD CUTTING problem. Note that this recursive algorithm has a loop from inside which the recursive calls are made. So it is not enough to assume that the recursive calls return the correct values; you also need to come up with a useful invariant for the loop.

(b) Note that—assuming that each array access and each operation involving up to two numbers can be done in constant time—the running time of the algorithm is proportional to the *number of times* that CUTROD is invoked, starting with the initial call CUTROD(n, P). Write a recurrence for this number, and solve it to get an asymptotically tight (that is: $\Theta()$) estimate of this number.

(c) Write pseudocode that *memoizes* the above recursive solution. Derive an upper bound on the running time of this memoized algorithm, as a function of n. How does this bound compare to the bound that you derived for the recursive solution in part (b)?

3. We derived the following dynamic programming solution to ROD CUTTING in class:

[2] See the Dynamic Programming chapter in CLRS for more details on this problem.

```
CutRod-DP$(n, P)$
    $R \leftarrow$ an array of length $n+1$    // $R = R[0 \ldots n]$
    $R[0] = 0$
    for $j \leftarrow 1$ to $n$
        $jMaxRevenue = -1$
        for $i = 1$ to $j$
            $iRevenue = P[i] + R[j-i]$
            if $iRevenue > jMaxRevenue$
                $jMaxRevenue \leftarrow iRevenue$
        $R[j] \leftarrow jMaxRevenue$
    return $R[n]$
```

We have now seen three solutions to this problem: the recursive solution given as part of Question 2, the memoized version of Question 2(c), and the above DP. Each of these algorithms only gave us the maximum revenue; they did not tell us *where/how to cut* the input rod to *realize* this maximum revenue.

Modify each of these three algorithms for Rod Cutting so that it also returns a collection of lengths (starting with zero at one end of the rod, and ending with n at the other end) where we can cut the input rod of length n to realize the maximum possible revenue. A sample output for $n = 10$ and some array P of prices might look like: "$2, 4, 7, 9$".

Hint: Simplify and conquer. See if it suffices to find, for each length $0 \leq i \leq n$, the best place to make the first *cut on a rod of length i.*

4. Recall the problem of efficiently multiplying a chain of matrices that we saw in class:

Least Cost Matrix-chain Multiplication

Input: An array $D[0 \ldots n]$ of $n+1$ positive integers.
Output: The least cost (total number of arithmetic operations required) for computing the matrix product $A_1 A_2 \cdots A_n$ where each A_i ; $1 \leq i \leq n$ has dimensions $D[i-1] \times D[i]$, given that multiplying a $p \times q$ matrix with a $q \times r$ matrix requires $\Theta(pqr)$ arithmetic operations.

Let $OPT(s, t)$ denote the least ("optimum") cost for multiplying the sub-sequence $A_s, A_{s+1}, \ldots, A_t$. If $s = t$ then $OPT(s, t) = 0$.

(a) Explain why there must exist an index $1 \leq i < n$ such that $OPT(1, n) = OPT(1, i) + OPT(i+1, n) + d_0 d_i d_n$.

Note: The main thing to show is *not* the fact that such an i exists, but the claim that it is OK to just add together the OPT() values of the sub-sequences. In particular: Why is it that the global optimum is not *smaller* than this sum?

(b) Write the pseudocode for a recursive algorithm that solves LEAST COST MATRIX-CHAIN MULTIPLICATION, using the claim proved in part (a). Argue that this algorithm correctly solves the problem.

(c) Prove that your algorithm from part (b) makes $\Omega(2^n)$ recursive calls.

(d) Memoize your algorithm from part (c). Show that it now runs in $\mathcal{O}(n^c)$ time for some fixed constant c. What is the value of c that you get?

(e) Write the pseudocode for a *non-recursive* algorithm that solves this problem using *dynamic programming*, based on the idea of projecting onto the "breakpoint" index i of part (a).

Argue that your procedure correctly solves the problem.

What is the running time of your algorithm in the asymptotic notation?

5. The 0-1 KNAPSACK WITHOUT REPETITION problem is defined as follows:

0-1 Knapsack Without Repetition

Input:	A non-negative integer n; a set of n items $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ where item I_j has value v_j and weight w_j; and a maximum weight capacity W. The values, weights, and W are all integers.
Output:	The maximum sum, taken over all subsets of $\mathcal{I}$ of total weight at most W, of the total value of the items in that set.

That is, the goal is to maximize

$$\sum_{i=1}^{n} v_i x_i$$

subject to the conditions

$$\left(\sum_{i=1}^{n} w_i x_i\right) \leq W,$$

and

$$x_i \in \{0,1\} \text{ for } 1 \leq i \leq n.$$

For this question, assume that the weights and values are given, respectively, as arrays $Value[1 \ldots n]$ and $Weight[1 \ldots n]$, where $Value[j]$ is the value and $Weight[j]$ is the weight of item I_j.

(a) Write the pseudocode for a *recursive* procedure NoRepKnapSackRec which solves the 0-1 Knapsack Without Repetition problem for these inputs.

Argue that your procedure correctly solves the problem.

Write a recurrence for the running time of the algorithm, and solve it to obtain a worst-case upper bound on the running time of the algorithm on these inputs.

(b) Memoize your algorithm of part (a). What is the running time of this version?

(c) Write the pseudocode for a *non-recursive* procedure NoRepKnapSackDP which solves the 0-1 Knapsack Without Repetition problem for these inputs. The procedure should be a *dynamic programming* algorithm based on the idea of projecting onto a *prefix* of the list of items. That is, for each $1 \leq i \leq n$, let projection S_i denote the set of all solutions which pick items *only* from the subset $\{I_1, I_2, \ldots, I_i\}$.

Argue that your procedure correctly solves the problem.

What is the running time of your algorithm in the asymptotic notation?

6. The 0-1 Knapsack With Repetition problem is defined as follows:

0-1 Knapsack With Repetition

Input: A non-negative integer n; a set of n item *types* $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ where each item of type T_j has value v_j and weight w_j; and a maximum weight capacity W. The values, weights, and W are all integers.

Output: The maximum sum, taken over all multisubsets of $\mathcal{T}$ of total weight at most W, of the total value of the items represented by that multiset.

That is, the goal is to maximize

$$\sum_{i=1}^{n} v_i x_i$$

subject to the conditions

$$\left(\sum_{i=1}^{n} w_i x_i\right) \leq W,$$

and

$$x_i \in (\mathbb{N} \cup \{0\}) \text{ for } 1 \leq i \leq n.$$

The difference from 0-1 Knapsack Without Repetition is that here we are allowed to pick more than one item of each type into the collection.

Assume, as before, that the weights and values are given, respectively, as arrays $Value[1 \dots n]$ and $Weight[1 \dots n]$, where $Value[j]$ is the value and $Weight[j]$ is the weight of item type T_j.

(a) Write the pseudocode for a *recursive* procedure RepKnapSackRec which solves the 0-1 Knapsack With Repetition problem for these inputs.

Argue that your procedure correctly solves the problem.

Write a recurrence for the running time of the algorithm, and solve it to obtain a worst-case upper bound on the running time of the algorithm on these inputs.

(b) Memoize your algorithm of part (a). What is the running time of this version?

(c) Write pseudocode for a *non-recursive* procedure RepKnapSack1 which solves the 0-1 Knapsack With Repetition problem for these inputs. The procedure should implement a *dynamic programming* algorithm based on the idea of projecting onto the *item types* in a solution.

Argue that your procedure correctly solves the problem.

What is the running time of your algorithm in the asymptotic notation?

(d) Write pseudocode for a *non-recursive* procedure RepKnapSack2 which solves the 0-1 Knapsack With Repetition problem for these inputs. The procedure should be a dynamic programming algorithm based on the idea of projecting onto the *number of items* in a solution.

Argue that your procedure correctly solves the problem.

What is the running time of your algorithm in the asymptotic notation?

7. A *subsequence* of an array A is any sub-array of A, obtained by deleting zero or more elements of A *without* changing the order of the remaining elements. An *increasing subsequence* of an integer array A is a sub-sequence of A which is in *strict* increasing order.

The input to the *Longest Increasing Subsequence* problem consists of an integer array A, and the goal is to find an increasing subsequence of A with the most number of elements. See Chapters 2 and 3 of Jeff Erickson's book for various solutions to this problem. In this question we will deal with the (slightly) simpler problem of finding the *length* of a longest increasing subsequence of the input array.

(a) Write the pseudocode for a *recursive* algorithm LenLis(A) that takes an integer array $A[1 \dots n]$ of length n as input and returns the length of a longest increasing subsequence of A. As always, simplifying the task makes it easier to solve:

i. First, write the pseudocode for LenLis(A) *assuming* that you have access to a function LenLisBigger(A, i, j) which takes A and two indices $1 \leq i <$

$j \le n$ as inputs, and returns the length of a longest increasing subsequence S of $A[j \dots n]$ with the property that every element of S is *larger* than $A[i]$.

  ii. Now write *recursive* pseudocode for LENLISBIGGER(A, i, j). Make sure that you correctly deal with the base cases.

(b) Write a recurrence for the running time of your LENLIS(A) function and solve it.

(c) Memoize your LENLIS(A) function. Derive an upper bound on the running time of the memoized version. How does this compare with the running time of the pure recursive version?

(d) Write pseudocode for a *non-recursive* procedure LENLISDP(A) that finds the length of a longest increasing subsequence of integer array $A[1 \dots n]$. The procedure should be a dynamic programming algorithm based on the idea of projecting onto *index pairs* (i, j) ; $i < j$, where the projection mimics the idea behind the recursive solution from part(a).

Argue that your procedure correctly solves the problem.

What is the running time of your algorithm in the asymptotic notation?