

- See previous practice problem sets for instructions.

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* it¹ on inputs $n = 35, n = 40, n = 45, n = 50$.

SIMPLEFIB(n)

```

1  if  $n \leq 1$ 
2      return 1
3  F1  $\leftarrow$  SIMPLEFIB( $n - 1$ )
4  F2  $\leftarrow$  SIMPLEFIB( $n - 2$ )
5  return F1 + F2

```

- (b) Write the pseudocode for a *memoized* version of SIMPLEFIB. Translate this pseudocode into code in your favourite procedural programming language, and time it on inputs $n = 35, n = 40, n = 45, n = 50$.

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 your memoized version.

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 rod².

CUTROD(n, P)

```

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

```

- (a) Explain why CUTROD correctly solves the ROD CUTTING problem.

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

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

- (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 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:

`CUTROD-DP(n, P)`

```

1  R ← an array of length n + 1    // R = R[0...n]
2  R[0] = 0
3  for j ← 1 to n
4      jMaxRevenue = -1
5      for i = 1 to j
6          iRevenue = P[i] + R[j - i]
7          if iRevenue > jMaxRevenue
8              jMaxRevenue ← iRevenue
9      R[j] ← jMaxRevenue
10 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 $D[0 \dots n]$ of $n + 1$ positive integers.

Output: The least cost (total number of arithmetic operations required) for computing the matrix product $A_1 A_2 \dots A_n$ 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 $\mathcal{O}(pqr)$ arithmetic operations.

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

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

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 $\text{OPT}()$ 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?

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 $\mathcal{I} = \{I_1, I_2, \dots, I_n\}$ 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 $\text{Value}[1 \dots n]$ and $\text{Weight}[1 \dots n]$, where $\text{Value}[j]$ is the value and $\text{Weight}[j]$ 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?

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* $\mathcal{T} = \{T_1, T_2, \dots, T_n\}$ 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 $\text{Value}[1 \dots n]$ and $\text{Weight}[1 \dots n]$, where $\text{Value}[j]$ is the value and $\text{Weight}[j]$ 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?

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:
- 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 $1 \leq i < j \leq n$ 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 $A[i]$.
 - 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?