Homework 03 — Solutions

CS 624, 2024 Spring

1. Problem 15.4-5 on p397, modified.

   > Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

   Do not write out the algorithm. Instead, do the following:

   (a) We will start with a related problem: Given a sequence $X[1 \,..\, n]$, what is the longest increasing subsequence of $X[1 \,..\, n]$ that ends with $X[n]$? I'll call this the LISE problem.

   State the *optimal substructure property* for the LISE problem. You are not required to prove it.

   > Suppose that $Z[1 \,..\, k]$ is a longest increasing subsequence of $X[1 \,..\, n]$ ending in $X[n]$.
   >
   > Then there is some j such that $X[j] < X[n]$ and $Z[1 \,..\, k{-}1]$ is a longest increasing subsequence for $X[1 \,..\, j]$ ending in $X[j]$.

   (b) Define a recursive function $LISE(k)$ that returns the *length* of the longest monotonically increasing subsequence for $X[1 \,..\, k]$ that ends in $X[k]$. Assume that the original sequence of numbers X is fixed/global.

   > $$LISE(k) = \max(\{1 + LISE(j) \mid 1 \le j < k,\ X[j] < X[k]\} \cup \{0\})$$
   >
   > That is, we maximize over the longest increasing subsequences ending in a value less than $X[n]$. We add 0 to the set so if there are no such candidates, the call to max is defined and returns 0.

   (c) Briefly explain how to solve the original problem by adapting the LISE-solving algorithm.

   > $$LIS(n) = \max\{LISE(j) \mid 1 \le j \le n\}$$

   (d) Briefly explain the running time of the algorithm, assuming memoization.

   > If the original sequence has n elements, then the size of the memo table is n, and filling in each slot takes $O(n)$ time, so the running time is $O(n^2)$.

   Note that there are other ways of solving the original problem. You should think about that, but for the homework you are required to use the structure above.

   > Here's another approach: Make a copy Y of the original array. Sort Y and remove duplicate elements. A sequence is an increasing subsequence of X if it is a subsequence of both X and Y. So we can find the longest increasing subsequence of X by finding the longest common subsequence (LCS) of X and Y, which is also $O(n^2)$ time.

2. Problem 15-2 on p405, modified.

   > A *palindrome* is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).
   >
   > Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input

string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

Do not write out the algorithm. Instead, do the following:

(a) State the *optimal substructure property* for the longest palindrome subsequence problem. You are not required to prove it.

Suppose $P[1 \,..\, k]$ is the longest palindrome subsequence of $A[1 \,..\, n]$.

- If $A[1] = A[n]$, then $P[1] = A[1] = A[n] = P[k]$, and $P[2 \,..\, k-1]$ is the longest palindrome sequence for $A[2 \,..\, n-1]$.
- If $A[1] \neq A[n]$, then $P[1 \,..\, k]$ is the longest palindrome subsequence of either $A[1 \,..\, n-1]$ or $A[2 \,..\, n]$.

(b) Define a recursive function $LPS(??)$ which returns the *length* of the longest palindrome subsequence for a given subproblem. Assume the original string is fixed/global. You must decide what arguments identify each subproblem. Briefly explain the meaning of the arguments.

Let $A[1 \,..\, n]$ be the original sequence. We'll treat A as global.
Let $LPS(i, j)$ represent the length of the longest palindrome subsequence of $A[i \,..\, j]$.

$$LPS(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ 2 + LPS(i+1, j-1) & \text{if } A[i] = A[j] \\ \max(LPS(i+1, j), LPS(i, j-1)) & \text{otherwise} \end{cases}$$

(c) Briefly explain the running time of your algorithm, assuming memoization.

Each subproblem is identified by a pair of indices i and j in $1 \,..\, n$ where $i \leq j$. So the size of the memo table is $O(n^2)$. Filling in each slot of the memo table is $O(1)$ time. So the total running time is $O(n^2)$.

3. Problem 16.2-2 on p427, modified.

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

Do not write out the algorithm. Instead, do the following:

(a) State the *optimal substructure property* for the 0-1 knapsack problem. You are not required to prove it.

Version 1

Suppose that T is an optimal set of items to take given available items A and weight budget W. That is, it maximizes $\sum_{i \in T} v_i$ while satisfying $\sum_{i \in T} w_i \leq W$.
Suppose $j \in T$. Then $T - \{j\}$ is an optimal set of items to take given available items $A - \{j\}$ with weight budget $W - w_j$.

> *This is true, but it leads to a very large search space.*
>
> *Recall the refinement we made to the change-making problem to focus on the choice of the first coin given. Similarly, in this problem, let's focus on the first choice made.*

Version 2

Suppose that T is an optimal set of items to take given available items $A = \{k, \ldots, n\}$ and weight budget W. That is, it maximizes $\sum_{i \in T} v_i$ while satisfying $\sum_{i \in T} w_i \leq W$. Then

- If $k \in T$, then $T - \{k\}$ is an optimal set of items to take from $\{k+1, \ldots, n\}$ with weight

budget $W - w_i$.

- If $k \notin T$, then T is an optimal set of items to take from $\{k+1, \ldots, n\}$ with weight budget W.

(b) Define a recursive function $KS(??)$ which returns the maximum *total value* of items in the knapsack for a given subproblem. Briefly explain what parts of the original problem are fixed/global and what parts are subproblem-specific.

Let the v, w sequences be global. Let n, the number of items, be global.

Let $KS(k, W)$ represent the greatest value achievable by taking items from the set $\{k, \ldots, n\}$ with a "weight budget" of W.

$$KS(k, W) = \begin{cases} -\infty & \text{if } W < 0 \\ 0 & \text{if } W \geq 0 \text{ and } k > n \\ \max(v_k + KS(k+1, W - w_k), KS(k+1, W)) & \text{if } W \geq 0 \text{ and } k \leq n \end{cases}$$

A result of $-\infty$ indicates that the knapsack's weight limit has been exceeded. (We choose $-\infty$ to represent an unviable result because we are *maximizing* over subproblem solutions.)

The solution to the original problem is $KS(1, W)$, where W is the original weight limit of the knapsack.

Alternative solution:

We could also build solutions by starting at the end of the array. That is, $KS'(k, W)$ would represent the optimal solution for the set $\{1, \ldots, k\}$ with weight budget W. Then

$$KS'(k, W) = \begin{cases} -\infty & \text{if } W < 0 \\ 0 & \text{if } W \geq 0 \text{ and } k = 0 \\ \max(v_k + KS'(k-1, W - w_k), KS'(k-1, W)) & \text{if } W \geq 0 \text{ and } k > 0 \end{cases}$$

The initial call would be $KS'(n, W)$ instead.

(c) Briefly explain the running time of your algorithm, assuming memoization.

The memo table has nW slots, and filling each slot takes $O(1)$ time, so the running time is $O(kW)$.

Depending on the actual item weights, it may not be necessary to fill the memo table completely. If every weight is even, for example, then half of the memo table is unused. This algorithm may thus benefit from top-down memoization.

4. The Fibonacci numbers are defined by the following recurrence:

$$F_0 = 0 \qquad F_1 = 1 \qquad F_{n+2} = F_{n+1} + F_n$$

(a) Write pseudocode for a *naive* recursive function that computes the n^{th} Fibonacci number.

```
function fibonacci(n) {
  if (n = 0) {
    return 0;
  } else if (n = 1) {
    return 1;
  } else {
    return n * fibonnaci(n-1);
  }
}
```

(b) Write pseudocode for a *top-down memoized* version of the same function. Your main function

should still take one integer argument and produce one integer result; it should use a memoized helper function.

```
function fibonnaci(n) {
  let memo = new array[0..n];
  return fibm(n,memo);
}

function fibm(n,memo) {
  if !(memo has key n) {
    if (n = 0) {
      memo[n] = 0;
    } else if (n = 1) {
      memo[n] = 1;
    } else {
      memo[n] = fibm(n-1,memo) + fibm(n-2,memo);
    }
  }
  return memo[n];
}
```

(c) Write pseudocode for a *bottom-up memoized* version of the Fibonacci function. Your main function should still take one integer argument and produce one integer result.

```
function fibonnaci(n) {
  let memo = new array[0..n];
  memo[0] = 0;
  memo[1] = 1;
  for (i = 2 to n) {
    memo[i] = memo[i-1] + memo[i-2];
  }
  return memo[n];
}
```

If you are using LaTeX, the `verbatim` or `alltt` environment might be helpful for writing the pseudocode.