Solutions for [DPV] Practice Dynamic Programming Problems

[DPV] Problem 6.8 – Longest common substring
Solution:

Here we are doing the longest common substring (LCStr), as opposed to the longest common subsequence (LCS). First, we need to figure out the subproblems. This time, we have two sequences instead of one. Therefore, we look at the longest common substring (LCStr) for a prefix of X with a prefix of Y. Since it is asking for substring which means that the sequence has to be continuous, we should define the subproblems so that the last letters in both strings are included. Notice that the subproblem only makes sense when the last letters in both strings are the same.

Let us define the subproblem for each i and j as:

$$P(i,j) = \text{length of the LCStr for } x_1x_2...x_i \text{ with } y_1y_2...y_j$$
$$\text{where we only consider substrings with } a = x_i = y_j \text{ as its last letter.}$$

For those i and j such that $x_i \neq y_j$, we set $P(i,j) = 0$.

Now, let us figure out the recurrence for $P(i,j)$. Assume $x_i = y_j$. Say the LCStr for $x_1 \dots x_i$ with $y_1 \dots y_j$ is the string $s_1 \dots s_\ell$ where $s_a = x_i = y_j$. Then $s_1 \dots s_{\ell-1}$ is the LCStr for $x_1 \dots x_{i-1}$ with $y_1 \dots y_{j-1}$. Hence, in this case $P(i,j) = 1 + P(i-1, j-1)$. Therefore, the recurrence is the following:

$$P(i,j) = \begin{cases} 1 + P(i-1,j-1) & \text{if } x_i = y_j \\ 0 & \text{if } x_i \neq y_j \end{cases}$$

The base cases are simple, $P(0,j) = P(i,0) = 0$ for any i, j.

The running time is $O(nm)$.

Algorithm 1 LCStr($x_1, x_2, \ldots, x_n,\ y_1, y_2, \ldots, y_m$)

```
for i = 0 to n do
    P(i, 0) = 0.
end for
for j = 0 to m do
    P(0, j) = 0.
end for
for i = 1 to n do
    for j = 1 to m do
        if x_i = y_j then
            P(i, j) = 1 + P(i − 1, j − 1)
        else
            P(i, j) = 0
        end if
    end for
end for
return max_{i,j}{P(i, j)}
```

[DPV] Problem 6.17 – Making change I
Solution:

In this problem, you have n denominations $x_1, \ldots, x_n$ (with unlimited supply of each) and a value V, and you are asked to determine in $O(nV)$ time whether there is a set of coins with total value V. This problem is similar to the knapsack problem (with repetition).

We make a one dimensional table. For $0 \le w \le V$, let

$S(w) = \{$`TRUE` or `FALSE` whether there is a subset of coins with total value $w\}$.

By considering the last coin used, we get that $S(w)$ is `TRUE` if there is a denomination $1 \le i \le n$ where $S(w - x_i)$ is `TRUE`. Hence, we get the following recurrence, where $\bigvee$ denotes OR. For $0 \le w \le V$,

$$S(w) = \bigvee_i \{S(w - x_i) : 1 \le i \le n, x_i \le w\}$$

This yields an $O(nV)$ time algorithm with 2 for-loops.

Algorithm 2 Coin Changing I

```
S(0) = TRUE.
for j = 1 to v do
    S(j) = FALSE.
end for
for i = 1 to v do
    for j = 1 to n do
        if i − x_j ≥ 0 then
            S(i) ← S(i − x_j) ∨ S(i)
        end if
    end for
end for
return S(v)
```

[DPV] Problem 6.18 – Making change II

Solution: This problem is very similar to the knapsack problem without repetition that we saw in class.

First of all, let's identify the subproblems. Since each denomination is used at most once, consider the situation for x_n. There are two cases, either

- We do not use x_n then we need to use a subset of $x_1, \ldots, x_{n-1}$ to form value v;
- We use x_n then we need to use a subset of $x_1, \ldots, x_{n-1}$ to form value $v - x_n$. Note this case is only possible if $x_n \leq v$.

If either of the two cases is `TRUE`, then the answer for the original problem is `TRUE`, otherwise it is `FALSE`. These two subproblems can depend further on some subproblems defined in the same way recursively, namely, a subproblem considers a prefix of the denominations and some value.

We define a $n \times v$ sized table D defined as:

$$D(i,j) = \{\texttt{TRUE} \text{ or } \texttt{FALSE} \text{ where there is a subset of the coins of denominations } x_1,\ldots,x_i \text{ to form the value } j.\}$$

Our final answer is stored in the entry $D(n, v)$.

Analogous to the above scenario with denomiation x_n we have the following recurrence relation for $D(i, j)$. For $i > 0$ and $j > 0$ then we have:

$$D(i,j) = \begin{cases} D(i-1,j) \vee D(i-1,j-x_i) & \text{if } x_i \leq j \\ D(i-1,j) & \text{if } x_i > j. \end{cases}$$

(Recall, $\vee$ denotes Boolean OR.)

The base cases are $D(0,0) =$ `TRUE` and for all $j = 1, 2, \ldots, v$, $D(0, j) =$ `FALSE`.

The algorithm for filling in the table is the following.

Algorithm 3 Coin Changing II

$D(0,0) =$ `TRUE`.
for $j = 1$ to v **do**
  $(0, j) =$ `FALSE`.
end for
for $i = 1$ to n **do**
  for $j = 0$ to v **do**
    if $x_i \leq j$ **then**
      $D(i,j) \leftarrow D(i-1,j) \vee D(i-1,j-v_i)$
    else
      $D(i,j) \leftarrow D(i-1,j)$
    end if
  end for
end for
return $D(n,v)$

Each entry takes $O(1)$ time to compute, and there are $O(nv)$ entries. Hence, the total running time is $O(nv)$.

[DPV] Problem 6.26 – Alignment
Solution: This is similar to the Longest Common Subsequence (LCS) problem, not the Longest Common Substring from this homework, and also to the Edit Distance problem that we did in class, just a bit more complicated. Let us use a similar way to what we did for edit distance to define the subproblem. Let

$$P(i, j) = \text{maximum score of an alignment of } x_1 x_2 \ldots x_i \text{ with } y_1 y_2 ... y_j.$$

Now, we figure out the dependency relationship. What subproblems does $P(i, j)$ depend on? There are three cases:

- Match x_i with y_j, then $P(i, j) = \delta(x_i, y_j) + P(i - 1, j - 1)$;
- Match x_i with $-$, then $P(i, j) = \delta(x_i, -) + P(i - 1, j)$;
- Match y_j with $-$, then $P(i, j) = \delta(-, y_j) + P(i, j - 1)$.

The recurrence then is the best choice among those three cases:

$$P(i, j) = \max\{\delta(x_i, y_j) + P(i-1, j-1), \delta(x_i, -) + P(i-1, j), \delta(-, y_j) + P(i, j-1)\}.$$

For the base case, we have to be a bit careful, there is no problem with assigning $P(0, 0) = 0$. But how about $P(0, j)$ and $P(i, 0)$? Can they also be zero? The answer is no, they should not even be the base case and should follow the recurrence of assigning $P(0, 1) = \delta(-, y_1)$ and generally $P(0, j) = P(0, j - 1) + \delta(-, y_j)$.
The running time is $O(nm)$.

Algorithm 4 Alignment($x_1, x_2, \ldots, x_n,\ y_1, y_2, \ldots, y_m$)

$P(0,0) = 0$.
for $i = 1$ to n **do**
  $P(i,0) = P(i-1,0) + \delta(x_i, -)$.
end for
for $j = 1$ to m **do**
  $P(0,j) = P(0,j-1) + \delta(-, y_j)$.
end for
for $i = 1$ to n **do**
  for $j = 1$ to m **do**
    $P(i,j) = \max\{\delta(x_i, y_j) + P(i-1, j-1), \delta(x_i, -) + P(i-1, j), \delta(-, y_j) + P(i, j-1)\}$
  end for
end for
return $P(n,m)$

[DPV] Problem 6.20 – Optimal Binary Search Tree
Solution:

This is similar to the chain matrix multiply problem that we did in class. Here we have to use substrings instead of prefixes for our subproblem. For all i, j where $1 \le i \le j \le n$, let

$$C(i,j) = \text{minimum cost for a binary search tree for words } p_i, p_{i+1}, \ldots, p_j.$$

The base case is when $i = j$, and then the expected cost is 1 for the search for word p_i, hence $C(i,i) = p_i$. Let's also set for $j < i$ $C(i,j) = 0$ since such a tree will be empty. These entries where $i > j$ will be helpful for simplifying our recurrence.

To make the recurrence for $C(i,j)$ we need to decide which word to place at the root. If we place p_k at the root then we need to place $p_i, \ldots, p_{k-1}$ in the left-subtree and $p_{k+1} \ldots, p_j$ in the right subtree. The expected number of comparisons involves 3 parts: words $p_i, \ldots, p_j$ all take 1 comparison at the root , the remaining cost for the left-subtree is $C(i, k-1)$, and for the right-subtree it's $C(k+1, j)$. Therefore, for $i < j$ we have:

$$C(i,j) = \min_{i \le k \le j} (p_i + \cdots + p_j) + C(i, k-1) + C(k+1, j)$$

To fill the table C we do so by increasing width $w = j - i$. Finally we output the entry $C(1,n)$. There are $O(n^2)$ entries in the table and each entry takes $O(n)$ time to fill, hence the total running time is $O(n^3)$.

The pseudocode for filling the table is on the following page.

Algorithm 5 BST$(p_1, p_2, \ldots, p_n)$

---

for $i = 1$ to n **do**
    $C(i,i) = p_i$
end for
for $i = 1$ to n **do**
    for $j = 1$ to $i-1$ **do**
        $C(i,j) = 0$
    end for
end for
for $w = 1$ to $n-1$ **do**
    for $i = 1$ to $n-w$ **do**
        $j = i + w$
        $C(i,j) = \infty$
        for $k = i$ to j **do**
            $cur = (p_i + \cdots + p_j) + C(i,k-1) + C(k+1,j)$
            if $C(i,j) > cur$ **then**
                $C(i,j) = cur$
            end if
        end for
    end for
end for
return $(C(1,n))$
