Solutions to Homework 1 Practice Problems

[DPV] Problem 6.1 – Maximum sum

First, a note: we typically would only ask for the maximum value, not the entire subsequence, eliminating the need for bookkeeeping and/or backtracking. This algorithm identifies and returns the value of the maximum contiguous subsequence.

(a) Define the entries of your table in words. E.g., $T(i)$ or $T(i, j)$ is ....

Let T(i) = the maximum sum achieved for the subsequence **ending at** a_i.

Note that since our subsequence is contiguous, the table entry includes the value of the number for the current index - this is an inclusive prefix. Our final answer will be the maximum value found in $T(\cdot)$

(b) State recurrence for entries of table in terms of smaller subproblems.

Our base case reflects the problem definition, the empty set has a maximum sum of zero:

$$T(0) = 0$$

Now, consider what happens at each element in S. If the previous sum is < 0, we want to restart our sequence; if the previous sum is > 0, the maximum sum achievable must include the prior position. Thus we have:

$$T(i) = a_i + \max(0, T(i-1)) \; \forall \; 1 \leq i \leq n$$

(c) Write pseudocode for your algorithm to solve this problem.

$T(0) = 0$
for $i = 1$ to n **do**
  $T(i) = a_i + \max(0, T(i-1))$
end for
return $\max(T(\cdot))$

To recreate the subsequence we add a second table, $B(i)$ which keeps track of the starting index for the subsequence ending at i.

$T(0) = 0$
$B(0) = 0$
for $i = 1$ to n **do**
  if $T(i-1) > 0$ **then**
    $T(i) = a_i + T(i-1)$
    $B(i) = B(i-1)$
  else
    $T(i) = a_i$
    $B(i) = i$
  end if
end for
$maxvalue = 0$
$maxindex = 0$
for $i = 1$ to n **do**
  if $T(i) > maxvalue$ **then**
    $maxindex = i$
    $maxvalue = T(i)$
  end if
end for
return $S[B(maxindex) \ldots maxindex]$

(d) Analyze the running time of your algorithm.

In both examples we have one loop through n inputs to set the table values, and a second loop to find the max. Overall run time is linear $O(n)$.

[DPV] Problem 6.4 – Dictionary lookup

(a) Define the entries of your table in words. E.g., $T(i)$ or $T(i, j)$ is ....

This subproblems consider prefixes but now the table just stores `TRUE` or `FALSE`. For $0 \leq i \leq n$, let $E(i)$ denote the `TRUE/FALSE` answer to the following problem: *$E(i)$: Can the string $s_1 s_2 \ldots s_i$ be broken into a sequence of valid words?* Whether the whole string can be broken into valid words is determined by the boolean value of $E(n)$.

(b) State recurrence for entries of table in terms of smaller subproblems.

The base case $E(0)$ is always `TRUE`: the empty string is a valid string. The next case $E(1)$ is simple: $E(1)$ is `TRUE` iff `dict`$(s[1])$ returns `TRUE`. We will solve subproblems $E(1), E(2), \ldots, E(n)$ in that order.

How do we express $E(i)$ in terms of subproblems $E(0), E(1), \ldots, E(i-1)$? We consider all possibilities for the last word, which will be of the form $s_j \ldots s_i$ where $1 \leq j \leq i$. If the last word is $s_j \ldots s_i$, then the value of $E(i)$ is `TRUE` iff *both* `dict`$(s_j \ldots s_i)$ and $E(j-1)$ are `TRUE`. Clearly the last word can be any of the i strings $s[j \ldots i], 1 \leq j \leq i$, and hence we have to take an "or" over all these possibilities. This gives the following recurrence relation *$E(i)$ is* `TRUE` *iff the following is* `TRUE` *for at least one $j \in [1, \ldots, i]$:* {`dict` $(s[j \ldots i])$ *is* `TRUE` *AND $E(j-1)$ is* `TRUE` }.

That recurrence can be expressed more compactly as the following where $\vee$ denotes boolean OR and $\wedge$ denotes boolean AND:

$$E(i) = \text{False} \bigvee_{j \in [1,\ldots,i]} \{\texttt{dict}(s[j \ldots i]) \wedge E(j-1)\} \;\; \forall\, 1 \leq i \leq n$$

with the base case:

$$E(0) = \texttt{TRUE}$$

(c) Write pseudocode for your algorithm to solve this problem.

Finally, we get the following dynamic programming algorithm for checking whether $s[\cdot]$ can be reconstituted as a sequence of valid words: set $E(0)$ to `TRUE`. Solve the remaining problems in the order $E(1), E(2), E(3), \ldots, E(n)$ using the above recurrence relation.

The second part of the problem asks us to give a reconstruction of the string if it is *valid*, i.e., if $E(n)$ is `TRUE`. To reconstruct the string, we add an additional bookkeeping device here: for each subproblem $E(i)$ we compute an additional quantity $prev(i)$, which is the index j such that the expression $\texttt{dict}(s_j, \ldots, s_i) \wedge E(j-1)$ is `TRUE`. We can then compute a valid reconstruction by following the $prev(i)$ "pointers" back from the last problem $E(n)$ to $E(1)$, and outputting all the charaters between two consecutive ones as a valid word.

Here is the pseudocode.

$E(0) =$ `TRUE`.
for $i = 1$ to n **do**
  $E(i) =$ `FALSE`.
  for $j = 1$ to i **do**
    if $E(j-1) =$ `TRUE` and $dict(S[j \ldots i] =$ `TRUE`$)$ **then**
      $E(i) =$ `TRUE`
      $prev(i) = j$
    end if
  end for
end for
return $E(n)$

To output the partition into words after running the above algorithm we use a recursive procedure to work back through the list:

---

if $E(n) =$ `FALSE` **then**
  return `FALSE`
else
  return (Reconstruct$(S[1 \ldots prev(n) - 1]), S[prev(n) \ldots n])$
end if

(d) Analyze the running time of your algorithm.

The run time is dominated by the nested for-loops, each of which is bounded by n, for $O(n^2)$ total time.

[DPV] Problem 6.8 – Longest common substring

(a) Define the entries of your table in words. E.g., $T(i)$ or $T(i,j)$ is ....

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:

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

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

(b) State recurrence for entries of table in terms of smaller subproblems.

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 $s_\ell = x_i = y_j$. 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:

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

where $1 \leq i \leq n$ and $1 \leq j \leq m$

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

(c) Write pseudocode for your algorithm to solve this problem.

---

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 $\max(P(\cdot, \cdot))$

(d) Analyze the running time of your algorithm.

The two base case loops are linear $O(n)$ and $O(m)$. The nested loops to establish the values for P and find the maximum dominate the run time, which is $O(nm)$.

[DPV] Problem 6.17 – Making change I (unlimited supply)

(a) Define the entries of your table in words. E.g., $T(i)$ or $T(i, j)$ is ....

Our subproblem considers making change for ever increasing amounts until we get to the value v: for all $1 \leq w \leq v$, if can make change w with coins of denominations $x_1, \ldots, x_n$ we let $T(w) = \texttt{TRUE}$, otherwise $T(w) = \texttt{FALSE}$.

(b) State recurrence for entries of table in terms of smaller subproblems.

Informally, the recurrence does the following: for each denomination x_i, if $T(w - x_i)$ is `TRUE` for at least one value of i, set $T(w)$ to be `TRUE`. Else set it `FALSE`. Given the base case $T(0) = \texttt{TRUE}$, the formal recurrence is

$$T(w) = \text{False} \bigvee_i T(w - x_i) \text{ where } x_i \leq w \; \forall \; 1 \leq i \leq n$$

where $\vee$ is the OR logical operator.

(c) Write pseudocode for your algorithm to solve this problem.

$T(0) = \texttt{TRUE}$
for $w = 1$ to v **do**
  $T(w) = \texttt{FALSE}$
  for $i = 1$ to n **do**
    if $x_i \leq w$ **then**
      $T(w) = T(w) \vee T(w - x_i)$
    end if
  end for
end for
return $T(v)$

(d) Analyze the running time of your algorithm.

The first step is $O(1)$, The nested loops are $O(v)$ and $O(n)$, with updates in order $O(1)$, this has running time $O(nv)$. The final step is $O(1)$ so the running time is $O(nv)$.

[DPV] Problem 6.18 – Making change II

(a) Define the entries of your table in words. E.g., $T(i)$ or $T(i,j)$ is ....

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 $D(n,v)$.

(b) State recurrence for entries of table in terms of smaller subproblems.

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

$$D(0,0) = \texttt{TRUE}$$
$$D(0,j) = \texttt{FALSE} \; \forall \; 1 \leq j \leq v$$

(c) Write pseudocode for your algorithm to solve this problem.

The algorithm for filling in the table is the following.

$D(0,0) =$ `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**
      $D(i,j) \leftarrow D(i-1,j) \vee D(i-1, j-x_i)$
    else
      $D(i,j) \leftarrow D(i-1,j)$
    end if
  end for
end for
return $D(n,v)$

(d) Analyze the running time of your algorithm.

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

[DPV] Problem 6.20 – Optimal Binary Search Tree

(a) Define the entries of your table in words. E.g., $T(i)$ or $T(i,j)$ is ....

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 $1 \leq i \leq j \leq n$, let

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

(b) State recurrence for entries of table in terms of smaller subproblems.

The base case is when $i = j$, and the expected cost is simply the word p_i, hence $C(i,i) = p_i$. Let's also set for $j < i$ $C(i,j) = 0$ for all $0 \leq j < i$ since such a tree will be empty. These entries where $i > j$ will be helpful for simplifying our recurrence; we need to include $i = n+1$ to ensure that all references are established. Formally, the base case is:

$T(i,i) = p_i$ for all $1 \leq i \leq n$
$T(i,j) = 0$ for all $1 \leq i \leq (n+1)$ and $0 \leq j \leq n$ and $j < i$

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 $1 \leq i < j \leq n$ we have:

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

To fill the table C we do so by increasing width $w = j - i$. Finally we output the entry $C(1,n)$.

(c) Write pseudocode for your algorithm to solve this problem.

Here's our pseudocode to identify the lowest cost. The backtracking to produce the tree is left as an exercise for you to consider:

---

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

(d) Analyze the running time of your algorithm.

There are $O(n^2)$ entries in the table and each entry takes $O(n)$ time to fill, hence the total running time is $O(n^3)$.

[DPV] Problem 6.26 – Alignment

(a) Define the entries of your table in words. E.g., $T(i)$ or $T(i,j)$ is ....

This is similar to the Longest Common Subsequence (LCS) problem, not the Longest Common Substring from this homework, just a bit more complicated. Let

$P(i,j) =$ maximum score of an alignment of $x_1x_2 \ldots x_i$ with $y_1y_2...y_j$.

(b) State recurrence for entries of table in terms of smaller subproblems.

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:

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

where $1 \leq i \leq n$ and $1 \leq j \leq m$. 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 $P(0,j) = P(0, j-1) + \delta(-, y_j)$.

(c) Write pseudocode for your algorithm to solve this problem.

$P(0,0) = 0$.
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 $P(n,m)$

(d) Analyze the running time of your algorithm.

The running time is $O(nm)$, the time required to fill our $n \times m$ table.

(Jumping Frog)

(a) Define the entries of your table in words. E.g., $T(i)$ or $T(i, j)$ is ....

Let $T[i]$ = the number of ways René can go from rock 1 to rock i

(b) State recurrence for entries of table in terms of smaller subproblems.

Lets start with a base case $T[1] = 1$, giving René credit for being on the first rock. What about the rest of the rocks? For any given rock i, René can land on that rock from either rock $i - 1$ or rock $i - 4$. Let's give René credit each time that prior rock has been visited (meaning that the value at the prior rock is > 0). And if there were two ways to get to rock $i - 1$, there would be two ways to get to rock i - so we accumulate the totals at each prior rock. This gives us the recurrence:

$$T[i] = T[i - 1] \text{ (when } 2 \leq i \leq 4)$$
$$T[i] = T[i - 1] + T[i - 4] \text{ (when } 5 \leq i \leq n)$$

(c) Write pseudocode for your algorithm to solve this problem.

$T[1] = 1$

for $i = 2 \rightarrow n$
    $T[i] = T[i - 1]$
    if $i > 4$ then
        $T[i] = T[i] + T[i - 4]$

return $T[n]$

(d) Analyze the running time of your algorithm.

We have a single loop through the n rocks, thus a linear run time $O(n)$

Final note: this could also be framed as a graph problem, do you see how?