Assignment 2

CMPT307
Summer 2020
Assignment 2
Due Wed June 24 at 23:59
3 problems, 40 points.

1. Improve the Longest Common Subsequence (LCS) algorithm (10 points):

   (a) Show how to compute the length of an LCS using only $2 \min(m, n)$ entries in the $c$ table plus $O(1)$ additional space. Express in pseudocode. Then analyze the memory space usage of your algorithm. (5 points)

   **Solution:** (4 points for the algorithm, 1 point for the space analysis)
   - Since we only use the previous row of the $c$ table to compute the current row, we compute as normal, but only keep two rows of the table. When we go to compute row $k$, we overwrite row $k-2$ since we will never need it again to compute the length.
   - Below (Algorithm 1) is the pseudocode. We assume that $n \leq m$. (If $m < n$, then exchange $X$ and $Y$ to make the situation the same.)
   - The space usage includes $2 \min(m, n) + 2$ used by an 2-d array $a$ and $O(1)$ used by variables $m, n$.

   (b) Then show how to do the same thing, but using $\min(m, n)$ entries plus $O(1)$ additional space. Again, express in pseudocode, and analyze the memory space usage of your algorithm. (5 points)

   **Solution:** (4 points for the algorithm, 1 point for the space analysis)
   - To use $\min(m, n) + O(1)$ space, observe that to compute $c[i, j]$, all we need are the entries $c[i-1, j], c[i-1, j-1]$ and $c[i, j-1]$. Thus, we can free up entry-by-entry those from the previous row which we will never need again, reducing the space requirement to $\min(m, n)$.
   - Below is the pseudocode. We assume that $n \leq m$. (If $m < n$, then exchange $X$ and $Y$ to make the situation the same.) Notice that the key idea is, when dealing with the $\{i, j\}$ entry, $a[0 \ldots j]$ store the value of $\text{memo}[i, 0 \ldots j]$ of previous matrix, and $a[j + 1 \ldots n]$ corresponds to the $i-1$-th row, which is $\text{memo}[i-1, j+1 \ldots n]$. The space usage is $\min(m, n) + O(1)$, which include $\min(m, n) + 1$ used by an array $a$, and $O(1)$ used by the four variables $(m, n, topleft, tmp)$. 

1
Algorithm 1: LCS_2min(X, Y)

\[ m \leftarrow \text{length}(X); \]
\[ n \leftarrow \text{length}(Y); \]
allocate matrix \( a[0 : 1, 0 \ldots n] = 0; \)
for \( i \) in 0, \ldots, m do
  for \( j \) in 0, \ldots, n do
    if \( i == 0 \) or \( j == 0 \) then
      \( a[1, j] = 0; \)
    else
      if \( X[i] == Y[j] \) then
        \( a[1, j] = a[0, j - 1] + 1; \)
      else
        \( a[1, j] = \max(a[0, j], a[1, j - 1]); \)
    end
  end
end
\( a[0, 0 \ldots n] = a[1, 0 \ldots n]; \)
return \( a[1, n]; \)

Algorithm 2: LCS_min(X, Y)

\[ m \leftarrow \text{length}(X); \]
\[ n \leftarrow \text{length}(Y); \]
allocate array \( a[0, \ldots, n] = 0; \)
for \( i \) in 0, \ldots, m do
  topleft = 0;
  for \( j \) in 0, \ldots, n do
    if \( i == 0 \) or \( j == 0 \) then
      topleft = \( a[j]; \)
      \( a[j] = 0; \)
    else
      if \( X[i] == Y[j] \) then
        tmp = topleft;
        topleft = \( a[j]; \)
        \( a[j] = tmp + 1; \)
      else
        topleft = \( a[j]; \)
        \( a[j] = \max(a[j - 1], a[j]); \)
    end
  end
end
return \( a[n]; \)
2. Refer to the power of 2 problem (Lecture 12, slides p21) (10 points).

(a) Redo the problem using the accounting method. (5 points)

Solution:

<table>
<thead>
<tr>
<th>Table 1:</th>
<th>Operation</th>
<th>Actual cost</th>
<th>Amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 2^k$</td>
<td>$i$</td>
<td>$2$</td>
<td></td>
</tr>
<tr>
<td>$i \not= 2^k$</td>
<td>$1$</td>
<td>$3$</td>
<td></td>
</tr>
</tbody>
</table>

or

<table>
<thead>
<tr>
<th>Table 2:</th>
<th>Operation</th>
<th>Actual cost</th>
<th>Amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 2^k$</td>
<td>$i$</td>
<td>$3$</td>
<td></td>
</tr>
<tr>
<td>$i \not= 2^k$</td>
<td>$1$</td>
<td>$3$</td>
<td></td>
</tr>
</tbody>
</table>

We prove that for Table 1.

To verify the correctness, we should prove that $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$.

If $i$ is not power of 2, the $i$-th operation has cost 1 and is paid 3, so it remains 2 credits.

We start from $n = 1$, obviously we have a credit of 2 and $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ holds.

Now suppose $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ holds for $n = 2^k$, so after $2^k$-th operation, the credit $\geq 0$. Now consider $n = 2^{k+1}$, there are $2^k - 1$ numbers between $2^k$ and $2^{k+1}$, thus the accumulated credit is $2^{k+1} - 2$. Then the $2^{k+1}$-th operation is paid 2, the total credit now is $2^{k+1}$, which equals to the cost $2^{k+1}$.

(b) Redo the problem using the potential method. (5 points)

Solution: We define the potential function $\Phi$ which satisfies $\Phi(D_0) = 0$ and

$$\Phi(D_i) = \begin{cases} 
  k + 3 & \text{for } i = 2^k \\
  \Phi(D_{2^k}) + 2(i - 2^k) & \text{for } otherwise
\end{cases}$$

where $k$ is the largest integer such that $2^k \leq i$ for the second sub-function.

Then we discuss $\hat{c}_i$ in two cases:

- **case 1**: if $i$ is not a power of 2,

  $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

  $$= c_i + \Phi(D_{2^k}) + 2(i - 2^k) - \Phi(D_{2^k}) - 2((i-1) - 2^k)$$

  $$= 1 + 2$$

  $$= 3$$
cast 2: if $i$ is a power of 2, that is, $i = 2^k$,

$$
\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
= c_i + (k + 3) - (\Phi(D_{2^k-1}) + 2(i - 1 - 2^{k-1})) \\
= c_i + (k + 3) - (k-1 + 3 + 2i - 2 - 2^k) \\
= c_i + (k + 3) - (k - 1 + 3 + 2i - 2 - 2^k) \\
= c_i + 3 - 2^k \\
= 3
$$

(3)

or

$$\Phi(D_i) = 2i - 2^k$$

(4)

where $k$ is the smallest integer such that $2^k > i$. Then we discuss $\hat{c}_i$ as follow:

case 1: if $i$ is not a power of 2,

$$
\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
= c_i + (2i - 2^k) - (2(i - 1) - 2^k) \\
= c_i + 2 \\
= 3
$$

(5)

case 2: if $i$ is a power of 2, that is, $i = 2^k$,

$$
\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
= c_i + (2i - 2^{k+1}) - (2(i - 1) - 2^k) \\
= c_i + 2 - 2^k \\
= 2^k + 2 - 2^k \\
= 2
$$

(6)
3. Coin changing (20 points):
   Consider the problem of making change for \( n \) cents using the fewest number of coins. Assume that each coin’s value is an integer.

   (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution. (7 points)

   **Solution:** (3 points for the greedy algorithm, 4 points for the proof of optimal solution.)

   \( Make\_change([25, 10, 5, 1], v) \)

   **Algorithm 3:** Make\_change(coins, v)

   ```
   n = length(coins);
   numcoins[0, ..., n - 1] = 0;
   surplus = v;
   for i in 0, ..., n - 1 do
     numcoins[i] = floor(surplus/coins[i]);
     surplus = surplus - numcoins[i]*coins[i];
   end
   return numcoins;
   ```

   To prove it provides an optimal solution:

   To make change for \( n \) cents using 25, 10, 5, 10, at most 2 dimes, 1 nickle and 4 pennies will be used. And the change made by dimes, nickels and pennies must be less than 25 cents. For example, if 2 nickles are used, it can be replaces by a dime and with 1 less coin, this wouldn’t happen in our greedy algorithm.

   Suppose there exists a \( n \) that the greedy algorithm doesn’t give the optimal solution. Let \( n_0, n_1, n_2, n_3 \) represent the number of coins used by the greedy algorithm, corresponding to quarter, dime, nickel, penny. Let \( n'_0, n'_1, n'_2, n'_3 \) be the number of coins used by the optimal solution.

   Since the greedy algorithm uses as many quarters as possible at the beginning, we have \( n'_0 \leq n_0 \). Now we show that \( n'_0 < n_0 \) is not true. If \( n'_0 < n_0 \) and \( e = n_0 - n'_0 \), it means that the optimal solution will make change for \( e \times 25 \) cents using dimes, nickles and pennies. In this case, the total number of coins used for \( e \times 25 \) coins cannot be less than \( 3e \) (2 dimes and 1 nickle), replace it using \( e \) quarters can obvious provide a better solution. If a better solution exists, \( n'_0 \) is not the optimal solution. So we have \( n'_0 = n_0 \).

   Analyze the usage of dimes, nickles and pennies using the above idea, we will have \( n'_1 = n_1, n'_2 = n_2 \) and \( n'_3 = n_3 \). The greedy algorithm provides an optimal solution.
(b) Suppose that the available coins are in the denominations that are powers of $c$, i.e., the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution. (4 points)

**Solution:** Given an optimal solution \{x_0, x_1, \ldots, x_k\} where $x_i$ indicates the number of coins of denomination $c^i$.

We will show that we must have $x_i < c$ for $x_i$ by $c$ and increase $x_{i+1}$ by 1. This collection of coins has the same value and has $c - 1$ fewer coins, so the original solution must be non-optimal.

This configuration of coins is exactly the same as you would get if you kept greedily picking the largest coin possible. This is because to get a total value of $V$, you would pick $x_k = \lfloor Vc^{-k} \rfloor$ and for $i < k$, $x_i = \lfloor (V \mod c^{i+1})c^{-i} \rfloor$. This is the only solution that satisfies the property that there aren’t more than $c$ of any but the largest denomination because the coin amounts are a base $c$ representation of $V \mod c^k$.

(c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of $n$. (3 points)

**Solution:** For example, $[1, 3, 4]$ to make change for 6; $[1, 5, 6]$ to make change for 10; etc.

(d) Give an $O(nk)$-time algorithm that makes change for any set of $k$ different coin denominations, assuming that one of the coins is a penny. (6 points)

**Solution:** use dynamic programming. See algorithm 4 below.
Algorithm 4: Make_change(S, v)

\[ \text{numcoins}[0, \ldots, v - 1] = 0; \]
\[ \text{coin}[0, \ldots, v - 1] = 0; \]
\[ \text{for } i \text{ in } 0, \ldots, v \text{ do} \]
\[ \quad \text{bestcoin} = -1; \]
\[ \quad \text{bestnum} = \infty; \]
\[ \quad \text{for } c \text{ in } S \text{ do} \]
\[ \quad \quad \text{if } \text{numcoins}[i - c] + 1 < \text{bestnum} \text{ then} \]
\[ \quad \quad \quad \text{bestnum} = \text{numcoins}[i - c] + 1; \]
\[ \quad \quad \quad \text{bestcoin} = c; \]
\[ \quad \text{end} \]
\[ \text{numcoins}[i] = \text{bestnum}; \]
\[ \text{coin}[i] = \text{bestcoin}; \]
\[ \text{end} \]
\[ \text{let change be an empty set;} \]
\[ \text{iter} = v; \]
\[ \text{while } \text{iter} > 0 \text{ do} \]
\[ \quad \text{add } \text{coin}[\text{iter}] \text{ to change;} \]
\[ \quad \text{iter} = \text{iter} - \text{coin}[\text{iter}]; \]
\[ \text{end} \]
\[ \text{return change} \]