Problem 1: The following problem arises in the context of DNA analysis. Let $\Sigma$ be an alphabet. Given two strings $X = <x_1, x_2, \ldots, x_n>$ and $Y = <y_1, \ldots, y_m>$ over $\Sigma$ we are looking for the longest common subsequence of $X$ and $Y$.

We say that $Z = <z_1, \ldots, z_k>$ is a subsequence of $X$ if there exists a strictly increasing sequence of indices $<i_1, \ldots, i_k>$ such that $x_{i_j} = z_j$ for all $j = 1, \ldots, k$. Note that a subsequence does not have to be formed of adjacent elements in the original string: for example if $X = <a, b, c, b, d, a, b>$ then $Z = <b, c, d, b>$ is a valid subsequence (the indices that correspond to $Z$ in $X$ are $2, 3, 5, 7$).

Given a dynamic programming algorithm to compute the longest common subsequence of $X$ and $Y$.

Solution: We use dynamic programming. Let $LCS[i, j]$ be the LCS (longest common subsequence) of the string $X_i = <x_1, x_2, \ldots, x_i>$ and $Y_j = <y_1, \ldots, y_j>$. Obviously $LCS[1, j]$ is equal to 1 if $x_1$ appears in $Y_j$, and 0 otherwise. Similarly $LCS[i, 1]$ is equal to 1 if $y_1$ appears in $X_i$, and 0 otherwise.

Now consider what happens if we want to fill $LCS(i, j)$ with $i, j \geq 2$. There are two cases to examine:

- If $x_i = y_j$ then $LCS(i, j) = 1 + LCS(i-1, j-1)$. That’s because the LCS of
- If $x_i \neq y_j$ then $LCS(i, j) = \max[LCS(i-1, j), LCS(i, j-1)]$

Both equations follow from the fact that the LCS problem has the ”optimal subproblem structure”. Indeed if $x_i = y_j$ then let $s = LCS(i, j)$; it must be that $s$ ends with $x_i$ (which is equal to $y_j$) otherwise it would not be the LCS($i, j$). Let $\hat{s}$ be the string $s$ after you drop $x_i$ from it. Obviously this must be the LCS($i-1, j-1$) indeed if $s' \neq \hat{s}$ with $|s'| \geq |\hat{s}|$ was the LCS($i-1, j-1$) then $s'$ concatenated with $x_i$ would be a longer LCS($i, j$) and that contradicts $s = LCS(i, j)$. Similarly for the case $x_i \neq y_j$: if $s = LCS(i, j)$ it must be that $s$ is either the LCS($i-1, j$) or LCS($i, j-1$) because $x_i$ and $y_j$ don’t match so there is no LCS that takes them both.

This gives us a dynamic programming algorithm to compute LCS($n, m$) by filling the LCS($\cdot, \cdot$) by row and column order (i.e. we fill row 2, then column 2, then row 3, then column 3 – since we are assured to always have the entries that we need). The running time is $O(nm)$ since filling the first row takes $O(nm)$, similarly for the first column, and then filling the rest of the matrix takes constant work per entry.

Problem 2: You are working at the cash register at the local supermarket and you have to make change for $n$ cents. To save time (and be nice to your customers) you want to do that with the fewest number of coins.

- Describe a greedy algorithm to make change when the coins in the drawer are the usual U.S. coin denominations: quarters, dimes, nickels and pennies. Argue that the greedy algorithm yields an optimal solution.
- What if you were working in the country of Powerpolis where coins denominations are $c ^ 0, c ^ 1, c ^ 2, \ldots, c ^ k$ for some integer $c > 1$ and $k \geq 1$. Does the greedy algorithm still yields an optimal solution?
- Show an example of a set of coin denominations for which the greedy algorithm does not yield an optimal solution.

Solution:

- Let $q, d, n, p$ be the number of quarters, dimes, nickels and pennies respectively in the optimal solution when you have to make changes for $N$ cents. First we are going to prove that the optimal solution must satisfy certain properties.

  1. $p < 5$. If $p \geq 5$ then the solution could be improved by removing 5 pennies and replacing them with a nickel.
2. If \( n \geq 2 \) then the solution could be improved by removing 2 nickels and replacing them with a dime.

3. If \( d = 2 \) then \( n = 0 \). Indeed if \( d \geq 3 \) then the solution could be improved by removing 3 dimes and replacing them with a quarter and a nickel. Moreover if \( d = 2 \) and \( n \geq 1 \) then the solution could be improved by removing 2 dimes and a nickel and replacing them with a quarter.

We now prove that there is only one possible solution satisfying the above, and that this solution is the greedy solution. The first three conditions imply that \( d + n + p < 25 \). Therefore \( q = \lfloor N/25 \rfloor \) which is exactly the choice made by the greedy algorithm. Conditions 1-2 imply that \( n + p < 10 \) therefore \( d = \lfloor (N - 25q)/10 \rfloor \) which also is the choice made by the greedy algorithm. Since \( p < 5 \) then we have that \( n = \lfloor (N - 25q - 10d)/10 \rfloor \) again the choice made by the greedy algorithm. Obviously \( p \) is now fully determined as the one in the greedy algorithm.

- The analysis in this case is the same as in the previous case. First of all we prove that if \( n_i \) is the number of coins of denomination \( c^i \) in the optimal solution, then it must be that \( n_i < c \) (otherwise the solution could be improved by removing \( c \) coins of denomination \( c^i \) and replacing them with a single coin of denomination \( c^{i+1} \)). Then we prove that there is only one solution that satisfy this condition and that’s the greedy solution. Indeed the above means that for all \( i \)

\[
\sum_{j=0}^{i-1} n_j c^j < c^i
\]

Note that this uniquely determines \( n_k \) as \( \lfloor N/c^k \rfloor \). This the choice made by the greedy algorithm. You can continue to argue this in descending order of \( i \).

- There are many ways to solve this, but one possible solution is denominations 1, 11, 12 and \( N = 22 \). The optimal solution uses 2 coins, but then greedy algorithm uses 11.

**Problem 3:** A sequence of \( n \) operations is performed on a data structure. The \( i^{th} \) operation costs \( i \) if \( i \) is a power of 2, otherwise it costs 1. Determine the total cost of the sequence of operations, and the amortized cost per operation. You can assume that \( n \) is a power of 2.

**Solution:** Assume \( n = 2^k \) and therefore \( k = \log n \). Let \( A \) be the set of integers between 1 and \( n \) which are not powers of 2. Obviously \( A = \{j : j \neq 1, 2, \ldots, 2^i, \ldots, 2^k \} \) and therefore \( |A| = n - k \). We have that the total cost is

\[
\sum_{i \in A} 1 + \sum_{i=0}^{k} 2^i = (n - k) + \frac{2^{k+1} - 1}{2 - 1} = n - k + 2^{k+1} - 1 = O(n)
\]

the last equation comes from the fact that \( k = \log n \). This means that the amortized cost per operation is \( O(1) \).