CMSC 451: Dynamic Programming Examples, Fall 2002

1. **Finding the Maximum Consecutive Subsequence.** Given a sequence $x_1, x_2, \ldots, x_n$ of real numbers (not necessarily positive), a maximum subsequence is a subsequence x_i, x_{i+1}, ..., x_j such that the sum of the numbers in it is maximum over all possible subsequences of consecutive elements. The sum of an empty subsequence is defined to be 0. The task is to find the sum of numbers in a maximum subsequence.

   We can solve this problem by divide-and-conquer, but let's use dynamic programming here. Let MaxV(i) denote the sum of a maximum subsequence in $x_1, x_2, \ldots, x_i$. The base case is simple where MaxV(0)=0 which corresponds to an empty subsequence. A maximum subsequence of $x_1, x_2, \ldots, x_i$ either contains x_i or does not contain x_i. It is tempting to write the recurrence formula as MaxV(i) = max (MaxV($i-1$), MaxV($i-1$)+x_i). But this recurrence formula is incorrect. Consider the sequence $\{2, -1, 3\}$. MaxV(2) = 2 but MaxV(3) = 4 $\neq$ MaxV(2) + 3. The problem here is that x_i may create a maximum subsequence in $x_1, \ldots, x_i$ by extending a non-maximum subsequence in $x_1, \ldots, x_{i-1}$. On the other hand, we would be able to compute MaxV(i) correctly if we have SMaxV($i-1$) where SMaxV($i-1$) corresponds to the sum of a maximum subsequence including x_{i-1} in $x_1, \ldots, x_{i-1}$. The correct recurrence formula is MaxV(i) = max (MaxV($i-1$), x_i, SMaxV($i-1$)+x_i). We need to compute SMaxV(i) as well where SMaxV(i) = max(x_i, SMaxV($i-1$)+x_i). In the algorithm depicted in figure 1, we compute SMaxV(i) and MaxV(i) from $i = 1$ upto n.

   The run time for the algorithm is clearly $O(n)$ since there is only one single for loop in line 3. In addition, note that we can reduce the space requirement for the algorithm. This follows since in computing SMaxV(i) and MaxV(i), we are only interested in the value of SMaxV($i-1$) and MaxV($i-1$). There is no need to use one-dimensional arrays to store the values for SMaxV($i-2$), MaxV($i-2$), SMaxV($i-3$),MaxV($i-3$), ...

```
(1)   SMaxV(0) ← 0
(2)   MaxV(0) ← 0
(3)   for i = 1 to n
(4)       do SMaxV(i) ← max(x_i, SMaxV(i − 1)+ x_i)
(5)          MaxV(i) ← max(MaxV(i − 1), x_i, SMaxV(i − 1)+x_i)
(6)   return MaxV(n)
```

Figure 1: Algorithm for Question 1

2. **Minimum Edit distance.** Let $x[1..m]$ and $y[1..n]$ be two text strings. We would like to change x character by character such that it becomes equal to y. We allow three types of changes (or edit steps), and we assign a cost of 1 to each: (1) *insert* – insert a character into the string, (2) *delete* – delete a character from the string, (3) *replace* – replace one character with a different character. For example, to change the string *abbc* into the string *babb*, we can delete the first *a*, forming the string *bbc*, then insert an *a* between the two *b*s (*babc*), and then replace the last *c* with a *b* for a total of three changes. However, we can also insert a new *b* at the beginning (forming *babbc*), and then delete the last *c*, for a total of two changes. The goal is to minimize the number of edit steps (minimum edit distance).

   Denote $C[i, j]$ as the minimum edit distance from $x[1..i]$ to $y[1..j]$. Suppose we know the values of $C[i', j']$ for all $i' \leq i$ and $j' \leq j$ except $C[i, j]$ which we want to compute. Consider two cases.

   - **Case I:** $x[i] = y[j]$: Here, we claim that $C[i, j] = C[i-1, j-1]$: we prove that reducing the problem to transforming $x[1..i-1]$ to $y[1..j-1]$ is a valid reduction, as follows.

     For a valid transformation from $x[1..i]$ to $y[1..j]$, the last character of the transformed string must equal $y[j]$. If a transformation does not match $x[i]$ to $y[j]$, then there could be two subcases.

(1) $y[j]$ is added to the transformed string by insertion or by replacing some character $x[i']$ for some $i' < i$. Consider two further subcases: (a) $x[i]$ is not retained and is deleted and (b) $x[i]$ is retained and matched to $y[j']$ for some $j' < j$. For (a), the transformation is not optimal as the cost of transformation can be reduced by matching $x[i]$ to $y[j]$. For (b), the transformation will involve insertion of $y[j'+1], \ldots, y[j]$ and matching $y[j']$ to $x[i]$. Instead of inserting $y[j]$ and matching $y[j']$ to $x[i]$, we could have a transformation of the same cost by first matching $y[j]$ to $x[i]$ and then inserting $y[j']$ through $y[j-1]$.
(2) $y[j]$ is matched to $x[i']$ for some $i' < i$. The transformation will involve deletion of $x[i'+1], \ldots, x[i]$ and matching $y[j]$ to $x[i']$. Similar to subcase 1b, we could have a transformation of the same cost by first matching $y[j]$ to $x[i]$, then deletion of $x[i'], \ldots, x[i-1]$.
Combining the two cases, we can see that there exists at least one optimal transformation that involves matching the character $x[i]$ to $y[j]$; hence $C[i,j] = C[i-1, j-1]$.

- **Case II:** $x[i] \neq y[j]$: Here, we only need consider making a change at the end of the substring $x[1..i]$. The reason is that we must handle $x[i]$ somehow. It is either deleted, or becomes some character in y. In the latter case, either $x[i]$ is modified to $y[j]$ or $x[i]$ is modified to $y[j']$ for some $j' < j$. Making changes somewhere else first does not save us the trouble or the cost of dealing with $x[i]$. Thus, there are three ways of reducing the problem of converting $x[1..i]$ to $y[1..j]$ to a smaller subproblem: (i) Insert the character $y[j]$ after $x[i]$. The problem becomes converting $x[1..i]$ to $y[1..j-1]$. (ii) Delete the character $x[i]$. The problem becomes converting $x[1..i-1]$ to $y[1..j]$. (iii) Replace the character $x[i]$ by $y[j]$. The problem becomes converting $x[1..i-1]$ to $y[1..j-1]$. Combining the three cases for $x[i] \neq y[j]$, we get $C[i,j] = \min(C[i,j-1], C[i-1,j], C[i-1,j-1]) + 1$.

The base cases are less obvious this time. Besides $C[0,0] = 0$, the base cases also include $C[0,j] = j$ for all j (which corresponds to the conversion from a null string to $y[1..j]$, addition of j characters.) Similarily, $C[i,0] = i$ for all i (deletion of i characters).

The algorithm depicted in figure 2 uses a top-down approach. It will only compute $C[i,j]$ when it is needed and has not been computed before. Each $C[i,j]$ is computed only once and it takes constant time to compute from $C[i-1,j-1], C[i-1,j]$ and $C[i,j-1]$. In addition, each $C[i,j]$ is referenced at most three times (in computing $C[i+1,j]$, $C[i+1,j+1]$ and $C[i,j+1]$). Therefore the running time of the algorithm is $O(mn)$.

The algorithm depicted in figure 3 uses a bottom-up approach. It computes the value in the matrix C from left to right, bottom to top. It is easy to see that the run time of the algorithm is also $O(mn)$.

3. **Bitonic Euclidean traveling-salesman problem.** Problem 15-1 in page 364 of the textbook.

   First of all, we sort the points in ascending order of their x-coordinates and label them as $P_1, P_2, \ldots, P_n$. This step can be done in $O(n \log n)$ time. We define a bitonic walk from P_i to P_j as a walk that starts at P_i, goes strictly right to left to P_1, and then goes strictly left to right to P_j with the constraint that it contains all the points $P_1, \ldots, P_{max(i,j)}$ exactly once. Denote $d(i,j)$ as the length of the shortest bitonic walk from P_i to P_j. Due to symmetry, we only need to consider the values of $d(i,j)$ for $i > j$.

   Denote $\Delta(i,j)$ as the straight-line distance between P_i and P_j. If we have computed the values of $d(i,j)$ for any pair of i,j $(i > j)$, then the length of the shortest bitonic tour is simply $\min(d(n,1) + \Delta(n,1), d(n,2) + \Delta(n,2), \ldots, d(n,n-1) + \Delta(n,n-1))$.

   In computing $d(i,j)$ for $i > j$, the base case is $d(2,1) = \Delta(2,1)$. Suppose $d(i',j')$ is known for any $i' < i$ and $j' < i'$. As P_{i-1} must be included in the bitonic walk from P_i to P_j, if $j < i-1$, then the edge (P_i, P_{i-1}) is present in the walk. This implies $d(i,j) = d(i-1,j) + \Delta(i,i-1)$. For $j = i-1$, exactly one of following edges is present in the walk: $(P_i, P_{i-2}), (P_i, P_{i-3}), \ldots, (P_i, P_1)$. Therefore it follows that $d(i,i-1) = \min(d(i-1,i-2) + \Delta(i-2,i), d(i-1,i-3) + \Delta(i-3,i), \ldots, d(i-1,1) + \Delta(1,i))$.

```
(1)   ComputeC(i, j)
(2)   if C[i, j] ≠ −1 then
(3)       return C[i, j]
(4)   if x[i] = y[j] then
(5)       C[i, j] ← ComputeC(i − 1, j − 1)
(6)   else
(7)       C[i, j] ← min(ComputeC(i − 1, j − 1), ComputeC(i − 1, j),
(8)                     ComputeC(i, j − 1)) + 1
(9)   return C[i, j]
(10)
(11)  Main
(12)  for i = 0 to m
(13)      do for j = 0 to n
(14)          do C[i, j] ← −1
(15)  for i = 0 to m
(16)      do C[i, 0] ← i
(17)  for j = 0 to n
(18)      do C[0, j] ← j
(19)  return ComputeC(m, n)
```

Figure 2: Top-down approach for Question 2

```
(1)   for i = 0 to m
(2)       do C[i, 0] ← i
(3)   for j = 0 to n
(4)       do C[0, j] ← j
(5)   for i = 1 to m
(6)       do for j = 1 to n
(7)           do if x[i] = y[j] then
(8)                   C[i, j] ← C[i − 1, j − 1]
(9)              else
(10)                  C[i, j] ← min(C[i − 1, j], C[i − 1, j − 1], C[i, j − 1])+1
(11)   return C[m, n]
```

Figure 3: Bottom up approach for Question 2

In order to recover the optimal bitonic tour, we assign a variable $\pi(i)$ for $i \geq 3$ to denote how we arrived at the value $d(i, i-1)$: i.e,. $d(i, i-1) = d(i-1, \pi(i)) + \Delta(\pi(i), i)$. There is no need to record how we arrived at the value $d(i, j)$ for $j < i-1$, since there is only one way to do so – it is from $d(i-1, j) + \Delta(i-1, i)$.

In the algorithm depicted in figure 4, lines $1-10$ are used to compute the d values. Lines $11-18$ are for finding the length of the shortest bitonic tour. Lines $19-40$ are for backtracking the optimal bitonic tour. The variable lr is true when we are appending an edge to the tour in the left-to-right direction. Otherwise it is false.

Run-time analysis: The double for loops in lines $2-10$ take $O(n^2)$ time. The single for loop in lines $14-18$ takes $O(n)$ time. The backtracking for the tour takes $O(n)$ time since there are $n+1$ edges in the tour. Therefore the total run time for the algorithm is $O(n^2)$.

```
(1)   d(2,1) ← Δ(2,1)
(2)   for i = 3 to n
(3)       do for j = 1 to i − 2
(4)              do d(i,j) ← d(i−1,j) + Δ(i−1,i)
(5)          d(i,i−1) ← d(i−1,i−2) + Δ(i−2,i)
(6)          π(i) ← i − 2
(7)          for k = 1 to i − 3
(8)              do if d(i,i−1) > d(i−1,k) + Δ(k,i) then
(9)                       d(i,i−1) ← d(i−1,k) + Δ(k,i)
(10)                      π(i) ← k
(11)  shortest ← d(n,n−1) + Δ(n,n−1)
(12)  tour ← (n,n−1)
(13)  j ← n − 1
(14)  for i = 1 to n − 2
(15)      do if shortest > d(n,i) + Δ(n,i) then
(16)                 shortest ← d(n,i) + Δ(n,i)
(17)                 j ← i
(18)                 tour ← (n,i)
(19)  i ← n
(20)  lr ← true
(21)  while (j ≠ 1 or i ≠ 2) do
(22)      while (i − 1 > j) do
(23)          if lr then
(24)              tour ← (i − 1,i)||tour
(25)          else
(26)              tour ← tour||(i,i − 1)
(27)          i ← i − 1
(28)      if i > 2 then
(29)          if lr then
(30)              tour ← (π(i),i)||tour
(31)          else
(32)              tour ← tour||(i,π(i))
(33)      i ← π(i)
(34)      swap(i,j)
(35)      lr ←!lr
(36)  if lr then
(37)      tour ← (1,2)||tour
(38)  else
(39)      tour ← tour||(2,1)
(40)  return tour
```

Figure 4: Algorithm for Question 3