CSE 102
Homework Assignment 6
Solutions

1. **Canoe Rental Problem.**
   There are n trading posts numbered 1 to n as you travel downstream. At any trading post i you can rent a canoe to be returned at any of the downstream trading posts j, where $j \geq i$. You are given an array $R[i,j]$ defining the cost of a canoe that is picked up at post i and dropped off at post j, for i and j in the range $1 \leq i \leq j \leq n$. Assume that $R[i,i] = 0$, and that you can't take a canoe upriver (so perhaps $R[i,j] = \infty$ when $i > j$). Your problem is to determine a sequence of canoe rentals that start at post 1, end at post n, and which has a minimum total cost. As usual there are really two problems: determine the cost of a cheapest sequence, and determine the sequence itself.

   Design a dynamic programming algorithm for this problem. First, define a 1-dimensional table $C[1 \cdots n]$, where $C[i]$ is the cost of an optimal (i.e. cheapest) sequence of canoe rentals that starting at post 1 and ending at post i. Show that this problem, with subproblems defined in this manner, satisfies the principle of optimality, i.e. state and prove a theorem that establishes the necessary optimal substructure. Second, write a recurrence formula that characterizes $C[i]$ in terms of earlier table entries. Third, write an iterative algorithm that fills in the above table. Fourth, alter your algorithm slightly so as to build a parallel array $P[1 \cdots n]$ such that $P[i]$ is the trading post preceding i along an optimal sequence from 1 to i. In other words, the last canoe to be rented in an optimal sequence from 1 to i was picked up at post $P[i]$. Write a recursive algorithm that, given the filled table P, prints out the optimal sequence itself. Determine the asymptotic runtimes of your algorithms.

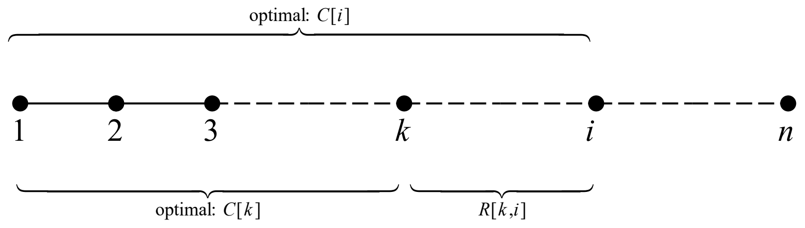
   Solution:
   One way to solve this problem would be to construct a 2-dimensional table whose i^{th} row and j^{th} column is the cost of an optimal sequence of canoe rentals starting at post i and ending at post j. This approach works, but one soon discovers that a 2-dimensional table is not really necessary. The entries in each row depend only on other entries in the same row, and since we are seeking an optimal sequence from 1 to n, only the first row is needed. Accordingly we define a 1-dimensional table $C[1 \cdots n]$ where $C[i]$ is the cost of an optimal sequence of canoe rentals starting at post 1 and ending at post i, for $1 \leq i \leq n$. When this table is filled, we simply return the value $C[n]$.

   Clearly $C[1] = 0$ since one need not rent any canoes to get from post 1 to 1. Let $i > 1$, and suppose we have found an optimal sequence taking us from 1 to i. In this sequence, there is some post k at which the final canoe was rented, where $1 \leq k < i$. In other words, our optimal sequence ends with a single canoe ride from post k to post i, whose cost is $R[k,i]$.

   Claim: The subsequence of canoe rentals starting at 1 and ending at k is also optimal.

   Proof: We prove this by contradiction. Assume that the above mentioned subsequence is not optimal. Then it must be possible to find a less costly sequence which takes us from 1 to k. Following that sequence by a single canoe ride from k to i, again of cost $R[k,i]$, yields a sequence taking us from 1 to i costing less than our original optimal one, a contradiction. Therefore the subsequence of canoe rides from 1 to k, obtained by deleting the final canoe ride in our optimal sequence from 1 to i, is itself optimal.

■



The above argument shows that this problem exhibits the required optimal substructure necessary for dynamic programming. It also shows how to define $C[i]$ in terms of earlier table entries. Indeed its clear that $C[i] = C[k] + R[k, i]$. Since we do not know the post k beforehand, we take the minimum of this expression over all k in the range $1 \le k < i$. Define

$$C[i] = \begin{cases} 0 & i = 1 \\ \min\limits_{1 \le k < i}(C[k] + R[k, i]) & 1 < i \le n \end{cases}$$

With this formula, the algorithm for filling in the table is straightforward.

```
CanoeCost($R$)
1.  $n$ = #rows[$R$]
2.  $C[1] = 0$
3.  for $i = 2$ to $n$
4.      min = $R[1, i]$
5.      for $k = 2$ to $i - 1$
6.          if $C[k] + R[k, i] <$ min
7.              min = $C[k] + R[k, i]$
8.      $C[i]$ = min
9.  return $C[n]$
```

There are two equally valid approaches to determining the actual sequence of canoe rentals which minimizes cost. One approach would be to back-track through the array $C[1 \cdots n]$. The other method is to alter the CanoeCost() algorithm so as to construct the optimal sequence while $C[\,1 \cdots n]$ is being filled. We take the second approach in the algorithm below, where we maintain an array $P[1 \cdots n]$ whose i^{th} entry, $P[i]$ is defined to be the post k at which the final canoe is rented, in an optimal sequence from 1 to i. Note that the definition of $P[1]$ can be arbitrary, since it is never used. Array P is then used to recursively print out the sequence.

```
CanoeCost($R$)
1.  $n$ = #rows[$R$]
2.  $C[1] = 0$
3.  $P[1] = 0$
4.  for $i = 2$ to $n$
5.      min = $R[1, i]$
6.      $P[i] = 1$
7.      for $k = 2$ to $i - 1$
8.          if $C[k] + R[k, i] <$ min
9.              min = $C[k] + R[k, i]$
10.             $P[i] = k$
11.     $C[i]$ = min
```

```
12. return $P$
```

```
PrintSequence($P, i$)  (pre: $1 \le i \le \text{length}[P]$)
1. if $i > 1$
2.     PrintSequence($P, P[i]$)
3.     print("rent a canoe at ", $P[i]$, " and drop it off at ", $i$)
```

Both CanoeCost() and CanoeSequence() run in time $\Theta(n^2)$, since the inner for loop performs $i - 2$ comparisons to determine $C[i]$, and

$$\sum_{i=2}^{n}(i-2) = \sum_{i=1}^{n-2} i = \frac{(n-1)(n-2)}{2} = \Theta(n^2)$$

The cost of the top level call PrintSequence(P, n) is the depth of the recursion, which is in turn, the number of canoes rented in an optimal sequence from post 1 to n. Thus PrintSequence() has worst case runtime in $\Theta(n)$. ■

2. **Moving on a checkerboard** (This is problem 15-6 on page 368 of the 2nd edition of CLRS.) Suppose that you are given an $n \times n$ checkerboard and a single checker. You must move the checker from the bottom (1st) row of the board to the top (n^{th}) row of the board according to the following rule. At each step you may move the checker to one of three squares:

   - the square immediately above,
   - the square one up and one to the left (unless the checker is already in the leftmost column),
   - the square one up and one right (unless the checker is already in the rightmost column).

   Each time you move from square x to square y, you receive $p(x, y)$ dollars. The values $p(x, y)$ are known for all pairs (x, y) for which a move from x to y is legal. Note that $p(x, y)$ may be negative for some (x, y).

   Give an algorithm that determines a set of moves starting at the bottom row, and ending at the top row, and which gathers as many dollars as possible. Your algorithm is free to pick any square along the bottom row as a starting point, and any square along the top row as a destination in order to maximize the amount of money collected. Determine the runtime of your algorithm.

   Solution:
   Define $C[i, j]$ to be the maximum amount of money that can be collected in this process by moving a checker from any square on row 1, to the square at row i , column j. Obviously $C[1, j] = 0$ for all j in the range $1 \le j \le n$, since at least one move must be made to collect any money. Once the table entry $C[i, j]$ is known for all i and j ($1 \le i \le n, 1 \le j \le n$), the maximum amount of money that can be collected by moving from row 1 to row n is computed as $\max_{1 \le j \le n} C[n, j]$.

   Observe that if one knows an optimal sequence of moves leading to square $y = (i, j)$, where $i > 1$, then the last move in that sequence must originate in one of the three neighboring squares in row $i - 1$. These three squares have coordinates $(i - 1, j - 1)$ (if $j > 1$), $(i - 1, j)$ and $(i - 1, j + 1)$ (if $j < n$). Let that preceding square be denoted x.

Claim: The subsequence of moves ending at x is itself an optimal sequence from row 1 to x.
Proof: Suppose there exists a more valuable sequence from row 1 to square x. Then by following that sequence with a single move from x to y, we obtain a more valuable sequence from row 1 to square y than our original optimal one, a contradiction. Therefore any optimal sequence ending at $y = (i, j)$ consists of an optimal sequence to the predecessor x of y, followed by a single move from x to y. ■
This problem therefore exhibits the required optimal substructure for a dynamic programming solution. Using the same notation as above, it is evident that $C[i, j] = C[y] = C[x] + p(x, y)$. Since the predecessor x is not known in advance, we have

$$C[i, j] = C[y] = \max_x (C[x] + p(x, y)),$$

where the maximum is taken over all (at most 3) possible predecessors x, of y. It is now a simple matter to write an iterative algorithm to fill in the table C. Since we also wish to print out an optimal sequence of moves, it is worthwhile to keep track of the predecessors as we fill in the table. Define $P[i, j]$ to be the predecessor of square (i, j) along an optimal sequence of moves starting in row 1, and ending at square (i, j), for $2 \le i \le n$ and $1 \le j \le n$.

```
OptimalSequence($p, n$)
1.  $C[1;\ 1 \cdots n] = (0, \ldots, 0)$    // the first row is initialized to all zeros
2.  for $i = 2$ to $n$
3.      for $j = 1$ to $n$
4.          $y = (i, j)$
5.          $x_0 = (i-1, j)$                       // the middle of 3 possible predecessors
6.          $x_{-1} = \begin{cases} x_0 & \text{if } j = 1 \\ (i-1, j-1) & \text{if } j > 1 \end{cases}$   // the leftmost of three 3 possible predecessors
7.          $x_1 = \begin{cases} x_0 & \text{if } j = n \\ (i-1, j+1) & \text{if } j < n \end{cases}$   // the rightmost of three 3 possible predecessors
8.          $C[y] = C[x_{-1}] + p(x_{-1}, y)$        // start with the leftmost
9.          $P[y] = x_{-1}$
10.         if $C[x_0] + p(x_0, y) > C[y]$         // if the middle is better
11.             $C[y] = C[x_0] + p(x_0, y)$        //     change it
12.             $P[y] = x_0$
13.         if $C[x_1] + p(x_1, y) > C[y]$         // if the rightmost is better
14.             $C[y] = C[x_1] + p(x_1, y)$        //     change it
15.             $P[y] = x_1$
16. $k = 1$
17. for $j = 2$ to $n$
18.     if $C[n, j] > C[n, k]$
19.         $k = j$
20. return $(C[n, k], k, P)$
```

Lines 4-7 initialize y and its three possible predecessors: x_{-1}, x_0, x_1, which reduce to two when either $j = 1$ or $j = n$. Lines 8-15 determine the larger of $C[x_{-1}] + p(x_{-1}, y)$, $C[x_0] + p(x_0, y)$ and $C[x_1] + p(x_1, y)$, then set $C[y]$ and $P[y]$ accordingly. Lines 16-19 determine the maximum value in the n^{th} row of C, which is the value of an optimal sequence from row 1 to row n. That value, the column k where it is found, and the table of predecessors P are returned on line 20.

```
PrintSequence($P$, $(i, j)$)  (pre: $P$ was returned by OptimalSequence())
1. if $i \geq 2$
2.     PrintSequence($P, P[i, j]$)
3.     print("move to square ", $(i, j)$)
4. else
5.     print("start at square ", $(i, j)$)
```

PrintSequence$(P, (i, j))$ prints an optimal sequence starting at row 1 and ending at square (i, j). To print an optimal sequence ending at row n, call PrintSequence$(P, (n, k))$ where P, n and k were returned by OptimalSequence(). OptimalSequence() clearly runs in time $\Theta(n^2)$. The cost of PrintSequence() is the depth of the recursion, which is simply the number of print statements executed, i.e. $\Theta(n)$. ■

3. (Read the **Rod-Cutting Problem** in 15.1 pp. 360-369 of CLRS 3rd edition. This is 15.1-2 on p. 370.) Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the *density* of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

   Counter Example
   Let $n = 3$ and $p = (p_1, p_2, p_3) = (1, 10, 12)$. Then the densities p_i/i are $(1, 5, 4)$. The greedy strategy first cuts a rod of length 2 having price 10, leaving a rod of length 1 having price 1. The total price for this sequence of cuts is then $10 + 1 = 11$. This sequence is not optimal however, since the whole rod of length 3 (with no cuts) is worth 12. Therefore the greedy strategy does not produce an optimal sequence of rod cuts in this instance. ■