CMPS 201
Spring 2010
Homework Assignment 9 Review Only – Solutions Included

1. Recall the coin changing problem discussed in class. We present the dynamic programming solution below for reference. Note that this algorithm assumes an unlimited supply of coins in each denomination $d[1], \ldots, d[n]$ are available.

<u>CoinChange(d, N)</u>
$n \leftarrow \text{length}[d]$
for $i \leftarrow 1$ to n
  $C[i,0] \leftarrow 0$
for $i \leftarrow 1$ to n
  for $j \leftarrow 1$ to N
    if $i = 1$ and $j < d[i]$
      $C[1, j] \leftarrow \infty$
    else if $i = 1$
      $C[1, j] \leftarrow 1 + C[1, j - d[i]]$
    else if $j < d[i]$
      $C[i, j] \leftarrow C[i-1, j]$
    else
      $C[i, j] \leftarrow \min\left(C[i-1, j],\ 1 + C[i, j - d[i]]\right)$
return $C[n, N]$

Write a recursive algorithm which given the filled table $C[1\ldots n; 0\ldots N]$ generated by the above algorithm, prints out a sequence of $C[n, N]$ coin values which disburse N monetary units. In the case $C[n, N] = \infty$, print a message to the effect that no such disbursal is possible.

Solution:
Note that array $d[1..n]$ is needed as input in order to navigate the table C.

<u>PrintCoins(C, d, i, j)</u> (Pre: $C[1..n; 0\ldots N]$ was filled by CoinChange(d, N))
if $j > 0$
  if $C[i, j] = \infty$
    print "Cannot pay the amount " j
    return
  if $i = 1$
    print "Pay one coin of denomination " $d[1]$
    PrintCoins(C, d, 1, $j - d[1]$)
  else if $j < d[i]$
    PrintCoins(C, d, $i-1$, j)
  else // both $i > 1$ and $j \geq d[i]$
    if $C[i, j] = C[i-1, j]$
      PrintCoins(C, d, $i-1$, j)
    else // $C[i, j] = 1 + C[i, j - d[i]]$

```
            print "Pay one coin of denomination "  d[i]
            PrintCoins(C, d, i, j − d[i])
```

2. Recall the Discrete Knapsack Problem described in class. A thief wishes to steal n objects having values $v_i > 0$ and weights $w_i > 0$ $(1 \le i \le n)$. His knapsack, which will carry the stolen goods, holds at most a total weight W. Let $x_i = 1$ if object i is taken, and $x_i = 0$ if object i is not taken $(1 \le i \le n)$. The thief's goal is to maximize the total value $\sum_{i=1}^{n} x_i v_i$ of the goods stolen, subject to the constraint $\sum_{i=1}^{n} x_i w_i \le W$.

   a. (10 Points) Write pseudo-code for a dynamic programming algorithm that solves this problem. Your algorithm should take as input the value and weight arrays $v[1..n]$ and $w[1..n]$, and the weight limit W. It should generate a table $V[1 .. n; 0 .. W]$ of intermediate results. Each entry $V[i, j]$ will be the maximum value of the objects which can be stolen if the weight limit is j, and if we only include objects in the set $\{1, \ldots, i\}$. Your algorithm should return the maximum possible value of the goods which can be stolen from the full set of objects, i.e. the value $V[n, W]$. (Alternatively you may write your algorithm to return the whole table.)

   Solution:
   <u>Knapsack(*v*, *w*, *W*)</u> (Pre: $v[1..n]$ and $w[1..n]$ contain positive numbers)

```
n ← length[v]
for j ← 0 to W   //  fill in first row
    if j < w[1]
        V[1, j] ← 0
    else
        V[1, j] ← v[1]
for i ← 2 to n   //  fill remaining rows
    for j ← 0 to W
        if j < w[i]
            V[i, j] ← V[i−1, j]
        else
            V[i, j] ← max(V[i−1, j], v[i] + V[i−1, j − w[i]])
return V[n, W]
```

   b. (10 Points) Write an algorithm that, given the filled table generated in part (a), prints out a list of exactly which objects are to be stolen.

   Solution:
   <u>PrintObjects(*V*, *w*, *i*, *j*)</u> (Pre: $V[1..n; 0..W]$ was filled by Knapsack(v, w, W))

```
if i = 1
    if j < w[1]
        print "Do not include object " 1
    else
```

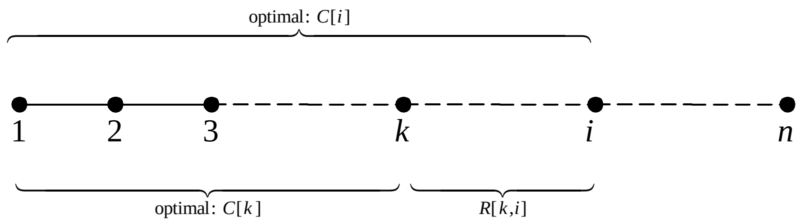
```
            print "Include object " 1
else  //  i > 1
      if V[i, j] = V[i−1, j]
            PrintObjects(V, w, i−1, j)
            print "Do not include object " i
      else   //  both j ≥ w[i] and V[i, j] = v[i] + V[i−1, j−w[i]]
            PrintObjects(V, w, i−1, j−w[i])
            print "Include object " i
```

3. **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 which is picked up at post *i* and dropped off at post *j*, for $1 \leq i \leq j \leq n$. 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 rentals which start at post 1 and 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 starts at post 1 and ends 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 $P[1 \ldots n]$ 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 which starts at post *i* and ends 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 $C[1 \ldots n]$ where $C[i]$ is the cost of an optimal sequence of canoe rentals that starts at post 1 and ends 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 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 last 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 which costs 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 a dynamic programming solution. 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

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

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

<u>CanoeCost(*R*)</u>

```
n ← #rows[R]
C[1] ← 0
for i ← 2 to n
    min ← R[1,i]
    for k ← 2 to i−1
        if C[k] + R[k,i] < min
            min ← C[k] + R[k,i]
    C[i] ← min
return C[n]
```

There are two equally valid approaches to determining the actual sequence of canoe rentals which minimizes cost. One approach is to alter the CanoeCost() algorithm so as to construct the optimal sequence while the table $C[1 \ldots n]$ is being filled. In the following algorithm we maintain an array $P[1 \ldots n]$ where $P[i]$ is defined to be the post k at which the last 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.

<u>CanoeSequence(*R*)</u>

```
n ← #rows[R]
C[1] ← 0, P[1] ← 0
for i ← 2 to n
    min ← R[1,i]
    P[i] ← 1
```

```
      for k ← 2 to i−1
          if C[k]+R[k,i] < min
               min ← C[k]+R[k,i]
               P[i] ← k
      C[i] ← min
return P
```

<u>PrintSequence(*P*, *i*)</u> (Pre: $1 \le i \le \text{length}[P]$)

```
if i > 1
      PrintSequence(P, P[i])
      print "Rent a canoe at post " P[i] " and drop it off at post " i
```

Both CanoeCost() and CanoeSequence() run in time $\Theta(n^2)$, since the inner for loop performs $i-2$ comparisons in order to determine $C[i]$, and $\sum_{i=2}^{n}(i-2) = \frac{(n-1)(n-2)}{2} = \Theta(n^2)$. The top level call to PrintSequence(*P*, *n*) has a cost that is the depth of the recursion, which is in turn, the number of canoes rented in the optimal sequence from post 1 to post *n*. Thus in worst case, PrintSequence() runs in time $\Theta(n)$. ///

4. **Moving on a checkerboard** (This is problem 15-6 on page 368.) 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 (nth) 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 in the first row, to the square at row *i* column *j*. Obviously $C[1,j]=0$ for all j ($1 \le j \le n$), since at least one move must be made to collect any money. Once $C[i,j]$ 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 easily computed as $\max_{1 \le j \le n} C[n,j]$.

   Observe that if one knows an optimal (i.e. highest value) sequence of moves leading to square (*i*, *j*), where $i > 1$, then the last move in that sequence must originate in one of the three squares in row

$i-1$ and in either column $j-1$ (provided $j>1$), or column j, or column $j+1$ (provided $j<n$). Denote that preceding square by x, and let $y=(i, j)$. **Claim:** The subsequence of moves leading to square x is itself an optimal sequence of moves from row 1 to square 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 argument shows that this problem exhibits the required optimal substructure for a dynamic programming solution. Furthermore, 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

$$C[i, j]=C[y]=\max_{x}\big(C[x]+p(x, y)\big),$$

where the maximum is taken over all of the (at most 3) possible predecessors 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 $1 \le j \le n$.

<u>OptimalSequence(p, n)</u> (Pre: $p(x, y)=$ the value of a transition from square x to square y)

$C[1, 1\cdots n] \leftarrow (0\cdots 0)$
for $i \leftarrow 2$ to n
    for $j \leftarrow 1$ to n
        $y \leftarrow (i, j)$
        $x_0 \leftarrow (i-1, j)$
        $x_{-1} \leftarrow \begin{cases} x_0 & \text{if } j=1 \\ (i-1, j-1) & \text{if } j>1 \end{cases}$
        $x_1 \leftarrow \begin{cases} x_0 & \text{if } j=n \\ (i-1, j+1) & \text{if } j<n \end{cases}$
        $C[y] \leftarrow C[x_{-1}]+p(x_{-1}, y)$
        $P[y] \leftarrow x_{-1}$
        if $C[x_0]+p(x_0, y) > C[y]$
            $C[y] \leftarrow C[x_0]+p(x_0, y)$
            $P[y] \leftarrow x_0$
        if $C[x_1]+p(x_1, y) > C[y]$
            $C[y] \leftarrow C[x_1]+p(x_1, y)$
            $P[y] \leftarrow x_1$
$k \leftarrow 1$
for $j \leftarrow 2$ to n
    if $C[n, j] > C[n, k]$
        $k \leftarrow j$
return $C[n, k]$, k, and P

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

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)

This algorithm prints out an optimal sequence ending at square (i, j). To print an optimal sequence starting at row 1 and ending at row n, call PrintSequence(P, (n, k)), where P and k were returned by OptimalSequence(). ///