CMPS 201
Homework Assignment 5
Solutions

1. (This is a continuation of problem 6 on hw assignment 4, which was itself based on problem 8-5 on page 207 of CLRS 3rd edition.) Recall an array $A[1 \cdots n]$ is k-sorted if and only if the $(n-k+1)$ consecutive k-fold averages of $A[1 \cdots n]$ are sorted:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$

for $1 \leq i \leq n-k$. Observe that if $n = k$, then any array of length n is k-sorted, since there is only one k-fold average in this case. We may also *define* any $A[1 \cdots n]$ to be k-sorted if $n < k$, since in this case there are no k-fold averages. Modify Quicksort to produce an algorithm that k-sorts an array of any length. Write your algorithm in pseudo-code and call it Quick$[k]$sort(). Specifically, the call Quick$[k]$sort$(A, 1, n)$ will k-sort $A[1 \cdots n]$, but will not necessarily $(k-1)$-sort the same array. Prove the correctness of your algorithm, and analyze its worst case run time.

Solution:
The key to this problem is the observation that any array $A[1 \cdots n]$ is k-sorted, if $n \leq k$. We simply run the Quicksort algorithm, but only recur down to array size k, or smaller.

Quick$[k]$sort(A, p, r)
1. if $k < r - p + 1$
2. $\quad q = \text{Partition}(A, p, r)$
3. $\quad \text{Quick}[k]\text{sort}(A, p, q-1)$
4. $\quad \text{Quick}[k]\text{sort}(A, q+1, r)$

Proof of correctness:
We show by induction by induction on $m = r - p + 1$, that Quick$[k]$sort(A, p, r) correctly k-sorts any subarray $A[p \cdots r]$ of length $m \geq 1$.

I. The base cases $m = 1, 2, \ldots, k$ follow from the observations made in the problem statement, i.e. there is at most one k-fold average, and hence the consecutive k-fold averages cannot be out of order.

II. Let $m > k$ and assume that Quick$[k]$sort correctly k-sorts any subarray of length less than m. We must show that Quick$[k]$sort(A, p, r) correctly k-sorts the subarray $A[p \cdots r]$ of length $m = r - p + 1$. Since $m > k$ the test on line 1 is true, and lines 2, 3 and 4 are executed. The call to Partition(A, p, r) on line 2 rearranges $A[p \cdots r]$ and returns an integer q such that $p \leq q \leq r$ and

(*) $$A[p \cdots (q-1)] \leq A[q] < A[(q+1) \cdots r]$$

holds. Since both

$$\text{length}(A[p \cdots (q-1)]) = (q-1) - p + 1 = q - p \leq r - p < r - p + 1 = m$$

and

$$\text{length}(A[(q+1) \cdots r]) = r - (q+1) + 1 = r - q \leq r - p < r - p + 1 = m,$$

the induction hypothesis guarantees that $A[p \cdots (q-1)]$ and $A[(q+1) \cdots r]$ are k-sorted after the recursive calls on lines 3 and 4 are executed. We show that this fact, together with (*), implies the subarray $A[p \cdots r]$ is now k-sorted.

The result of problem 6c on hw4 says that $A[p \cdots r]$ is k-sorted if and only if $A[i] \le A[i+k]$ for all i in the range $p \le i \le r-k$. We have 3 cases to consider.

(1) $p \le i < i+k \le q-1$
In this case, the fact that $A[p \cdots (q-1)]$ is k-sorted implies $A[i] \le A[i+k]$.

(2) $q+1 \le i < i+k \le r$
Then $A[(q+1) \cdots r]$ being k-sorted implies $A[i] \le A[i+k]$.

(3) $p \le i < q \le i+k \le r$
In this case, the inequality (*) itself implies $A[i] \le A[i+k]$.

In all cases, $A[i] \le A[i+k]$ for all i in the range $p \le i \le r-k$, and hence the full subarray $A[p \cdots r]$ is k-sorted, as required.

The result follows for all $m \ge 1$ by the 2nd Principle of Mathematical Induction. ■

Analysis of worst case runtime
The recurrence for worst case number of array comparisons is very similar to that for ordinary Quicksort, the only difference being where (for which n) the recurrence halts. As before, the worst case behavior is elicited when one of the two subarrays on lines 3 or 4 is empty.

$$T(n) = \begin{cases} 0 & 1 \le n \le k \\ T(n-1) + (n-1) & n > k \end{cases}$$

The iteration method yields $T(n) = \sum_{i=1}^{s}(n-i) + T(n-s)$, stopping at the smallest s for which $1 \le n-s \le k$, that is $s = n-k$. Thus

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-k}(n-i) \\ &= \sum_{i=1}^{n-k} n - \sum_{i=1}^{n-k} i \\ &= n(n-k) - \frac{(n-k)(n-k+1)}{2} \\ &= \Theta(n^2). \end{aligned}$$

■

Remark
The average case runtime of $\mathrm{Quick}[k]\mathrm{sort}$ can be shown to be $\Theta(n\log(n))$, as one would expect. We leave this as a further exercise for the reader. ■

2. Read the Coin Changing Problem in the handout on Dynamic Programming. Its solution is presented below for reference. Recall this algorithm assumes that an unlimited supply of coins in each denomination $d = (d_1, d_2, \dots, d_n)$ are available.

<u>CoinChange(d, N)</u>

```
n = length[d]
for i = 1 to n
    C[i, 0] = 0
for i = 1 to n
    for j = 1 to N
        if i = 1 and j < d[1]
            C[1, j] = ∞
        else if i = 1
            C[1, j] = 1 + C[1, j − d[1]]
        else if j < d[i]
            C[i, j] = C[i − 1, j]
        else
            C[i, j] = min(C[i − 1, j], 1 + C[i, j − d[i]])
return C[n, N]
```

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

Solution:

Note that array $d = (d_1, d_2, \dots, d_n)$ is needed as input in order to navigate the table C.

<u>PrintCoins(C, d, i, j)</u> Pre: $C[1 \cdots n; 0 \cdots N]$ was filled by CoinChange(d, N)

```
if j > 0
    if C[i, j] == ∞
        print("cannot pay 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 ≥ 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)
```

3. Read the Discrete Knapsack Problem in the handout on Dynamic Programming. A thief wishes to steal *n* objects having values $v_i > 0$ and weights $w_i > 0$ (for $1 \le i \le n$). His knapsack, which will carry the stolen goods, holds at most a total weight $W > 0$. Let $x_i = 1$ if object *i* is to be 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. Write pseudo-code for a dynamic programming algorithm that solves this problem. Your algorithm should take as input the value and weight arrays $v[\,]$ and $w[\,]$, and the weight limit *W*. It should generate a table $V[1 \cdots n; 0 \cdots W]$ of intermediate results. Each entry $V[i, j]$ will be the maximum value of the objects that can be stolen if the weight limit is *j*, and if we only include objects in the set $\{1, \dots, i\}$. 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: $v[1 \cdots n]$ and $w[1 \cdots n]$ 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. 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: $V[1 \cdots n; 0 \cdots W]$ was filled by Knapsack(v, w, W))

```
if i == 1
    if j < w_i
        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)
```

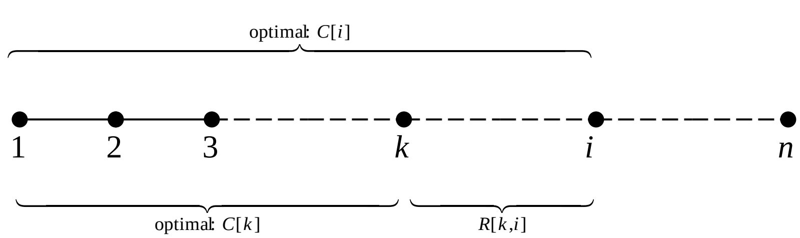
4. **Canoe Rental Problem.**
There are n trading posts numbered 1 to n as you travel downstream. At any trading post i you can rent a canoe to be returned at any of the downstream trading posts j, where $j \geq i$. You are given an array $R[i,j]$ defining the cost of a canoe that is picked up at post 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 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)
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 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)
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
```

```
PrintSequence(P, i)  (pre: 1 ≤ i ≤ length[P])
if i > 1
    PrintSequence(P, P[i])
    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 $\Theta(n)$. ■

5. **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 $1 \le j \le n$.

<u>OptimalSequence(p, n)</u>
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. $x_0 = (i - 1, j)$
6. $x_{-1} = \begin{cases} x_0 & \text{if } j = 1 \\ (i - 1, j - 1) & \text{if } j > 1 \end{cases}$
7. $x_1 = \begin{cases} x_0 & \text{if } j = n \\ (i - 1, j + 1) & \text{if } j < n \end{cases}$
8. $C[y] = C[x_{-1}] + p(x_{-1}, y)$
9. $P[y] = x_{-1}$
10.   if $C[x_0] + p(x_0, y) > C[y]$
11. $C[y] = C[x_0] + p(x_0, y)$
12. $P[y] = x_0$
13.   if $C[x_1] + p(x_1, y) > C[y]$
14. $C[y] = C[x_1] + p(x_1, y)$
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.

<u>PrintSequence$(P, (i, j))$</u> (pre: P was returned by OptimalSequence())
1. if $i \ge 2$
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. $\Theta(n)$. ∎

6. (This is Problem 15.2-1 on page 378 of CLRS.) Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $(5,10,3,12,5,50,6)$. Show the complete tables $m[i,j]$ and $s[i,j]$ described in the handout on Dynamic Programming.

Solution:
In this case $n=6$ and $p[0\cdots 6]=(5,10,3,12,5,50,6)$. Recall that

$$m[i,j]=\begin{cases} 0 & i=j \\ \min\limits_{i\le k<j}\left(m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\right) & i<j \end{cases}$$

for $1\le i\le j\le n$. Also $s[i,j]$ is defined to be the value of k at which the minimum occurs in cell $m[i,j]$, for $1\le i<j\le n$. With some effort, we have

$m[\ ]$		j 1	2	3	4	5	6
i	1	**0**	**150**	**330**	**405**	**1655**	**2010**
	2	-	**0**	**360**	**330**	**2430**	**1950**
	3	-	-	**0**	**180**	**930**	**1770**
	4	-	-	-	**0**	**3000**	**1860**
	5	-	-	-	-	**0**	**1500**
	6	-	-	-	-	-	**0**

and

$s[\ ]$		j 1	2	3	4	5	6
i	1	*	**1**	**2**	**2**	**4**	**2**
	2	-	*	**2**	**2**	**2**	**2**
	3	-	-	*	**3**	**4**	**4**
	4	-	-	-	*	**4**	**4**
	5	-	-	-	-	*	**5**
	6	-	-	-	-	-	*

The second table gives us the optimal parenthesization as

$$(A_1\cdot A_2)\cdot\big((A_3\cdot A_4)\cdot(A_5\cdot A_6)\big)$$

∎