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.

CoinChange(d, N)
1. $n = \text{length}[d]$
2. for $i = 1$ to n
3. $\quad C[i, 0] = 0$
4. for $i = 1$ to n
5. $\quad$ for $j = 1$ to N
6. $\quad\quad$ if $i = 1$ and $j < d[1]$
7. $\quad\quad\quad C[1, j] = \infty$
8. $\quad\quad$ else if $i = 1$
9. $\quad\quad\quad C[1, j] = 1 + C\big[1, j - d[1]\big]$
10. $\quad\quad$ else if $j < d[i]$
11. $\quad\quad\quad C[i, j] = C[i - 1, j]$
12. $\quad\quad$ else
13. $\quad\quad\quad C[i, j] = \min\big(C[i - 1, j], 1 + C\big[i, j - d[i]\big]\big)$
14. 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*.

PrintCoins(C, d, i, j) Pre: $C[1 \cdots n; 0 \cdots N]$ was filled by CoinChange(d, N)
1. if $j > 0$
2. $\quad$ if $C[i, j] == \infty$
3. $\quad\quad$ print("cannot pay amount ", j)
4. $\quad\quad$ return
5. $\quad$ if $i == 1$
6. $\quad\quad$ print("pay one coin of denomination ", d_1)
7. $\quad\quad$ PrintCoins$(C, d, 1, j - d_1)$
8. $\quad$ else if $j < d_i$
9. $\quad\quad$ PrintCoins$(C, d, i - 1, j)$
10. $\quad$ else // both $i > 1$ and $j \geq d_i$
11. $\quad\quad$ if $C[i, j] == C[i - 1, j]$
12. $\quad\quad\quad$ PrintCoins$(C, d, i - 1, j)$
13. $\quad\quad$ else // $C[i, j] == 1 + C[i, j - d_i]$
14. $\quad\quad\quad$ print("pay one coin of denomination ", d_i)
15. $\quad\quad\quad$ 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 $\{1, \ldots, i\}$. 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:

```
Knapsack(v, w, W)  (pre:  v[1 ··· n] and w[1 ··· n] contain positive numbers)
1.  n = length[v]
2.  for j = 0 to W   //  fill in first row
3.      if j < w_1
4.          V[1, j] = 0
5.      else
6.          V[1, j] = v_1
7.  for i = 2 to n   //  fill remaining rows
8.      for j = 0 to W
9.          if j < w_i
10.             V[i, j] = V[i − 1, j]
11.         else
12.             V[i, j] = max( V[i − 1, j],  v_i + V[i − 1, j − w_i] )
13. 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:

```
PrintObjects(V, w, i, j)  (pre: V[1 ··· n; 0 ··· W] was filled by Knapsack(v, w, W)
1.  if i == 1
2.      if j < w_i
3.          print("do not include object ", 1)
4.      else
5.          print("include object ", 1)
6.  else  // i > 1
7.      if V[i, j] == V[i − 1, j]
8.          PrintObjects(V, w, i − 1, j)
9.          print("do not include object ", i)
10.     else  //  both j ≥ w_i and V[i, j] == v_i + V[i − 1, j − w_i]
11.         PrintObjects(V, w, i − 1, j − w_i)
12.         print("include object ", i)
```

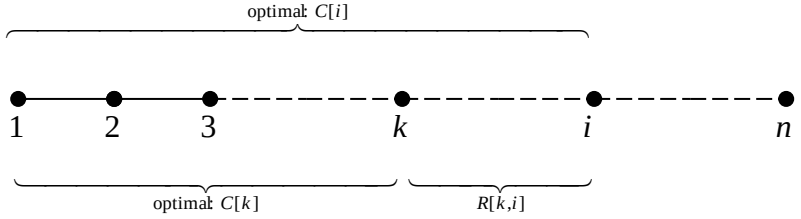
4. **Canoe Rental Problem.**
There are n trading posts numbered 1 to n as you travel downstream. At any trading post i you can rent a canoe to be returned at any of the downstream trading posts j, where $j \geq i$. You are given an array $R[i,j]$ defining the cost of a canoe that is picked up at post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

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

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 ≤ i ≤ 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 $\text{PrintSequence}(P,n)$ 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 $1 \le j \le n$.

```
OptimalSequence(p, n)
1.  C[1; 1 ··· n] = (0, ..., 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} = { x_0              if j = 1
                     { (i − 1, j − 1)   if j > 1
7.          x_1 = { x_0              if j = n
                  { (i − 1, j + 1)   if j < n
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.

```
PrintSequence(P, (i, j))  (pre: P was returned by OptimalSequence())
1.  if i ≥ 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. $\Theta(n)$. ■

2. (7 pts) Assume that you have a list of n home maintenance/repair tasks (numbered from 1 to n) that must be done *in list order* on your house. You can either do each task i yourself at a positive cost (that includes your time and effort) of $c[i]$. Alternatively, you could hire a handyman who will do the next 4 tasks on your list for the fixed cost h (regardless of how much time and effort those 4 tasks would cost you). You are to create a dynamic programming algorithm that finds a minimum cost way of completing the tasks. The inputs to the problem are h and the array of costs $c[1], \ldots, c[n]$.

   (a) (3 pts) First, find a justify a recurrence (with boundary conditions) giving the optimal cost for completing the tasks.
   Clarification: The handyman must be hired for exactly 4 tasks, so cannot be hired if fewer tasks remain. (Note that students may answer the harder version where the handyman can be hired even if fewer than 4 tasks remain).
   Consider any optimal solution S to some n-task instance of the problem. Define $M(j)$ to be the minimum cost required to do the first j tasks. Since S is optimal, the cost of S is $M(n)$. Optimal solution S either has the owner doing the last task, or the handyman doing the last task. First consider the case that $n < 4$, since there are not enough tasks for the handyman, the owner must do all of them with cost the sum of the $c(i)$'s. (Note that if $n = 0$ the sum of the $c(i)$'s is also 0.)
   Now assume that $n \geq 4$, and examine who does the last task.
   Case 1: The owner does the last task. In this case the cost of the optimal solution is $c(n)$ plus the cost of solution S's way of doing all $n - 1$ previous tasks.
   Claim: this must be the minimum cost for doing the $n - 1$ previous tasks.
   Proof: Assume to the contrary that the previous $n-1$ tasks can be done more cheaply than they are done in S. Modifying S to do the remaining tasks in this cheaper way results in a solution S' that is cheaper than S, contradicting the optimality of S.
   The cost of S is thus $M(n) = M(n-1) + c(n)$ in this case.
   case 2: The handyman does the last task.

If the handyman does last task, the handyman must do the last 4 tasks. Since S is optimal, it must also contain a minimum-cost way of doing the remaining $n-4$ tasks. Otherwise one could hire the handyman for the last four tasks and use a cheaper way of doing the other $n-4$ tasks to contradict the optimality of S. This implies the cost of S is $M(n) = h + M(n-4)$ in this case.

Combining cases 1 and 2 gives:

$$M(n) = \min\{c(n) + M(n-1); h + M(n-4)\}$$

with the boundary condition that for $0 \leq n < 4$

$$M(n) = \sum_{1 \leq i \leq n} c(i)$$

Thus $M(0) = 0$.

Students may solve an earlier version of the problem where the handyman is allowed to be used for fewer than 4 tasks at the end of the sequence for full credit. There are a couple of ways to modify the solution if the handyman is allowed to finish the tasks even if there are less than 4 remaining. The difficulty is that the original problem and the subproblems are no longer the same: the original problem for tasks 1 through n can use the handyman on the last 2 or 3 tasks while the subproblems for tasks 1 through $j < n$ cannot.

One way is to first prove that no optimal solution has the owner doing a task j followed by the handyman doing all the remaining tasks (from $j+1$ to n) unless the number of remaining tasks is a multiple of 4. This can be proven by contradiction similar to the correctness of a greedy algorithm. The recurrence could then assume that the handyman only does tasks in groups of n, but an additional step at the end would be needed to consider the case where the handyman is hired to do all of the tasks.

Another way to handle this is to re-define the subproblems, letting $M[i]$ be the minimum cost to do tasks from i to n (rather than from 1 to i). Now the subproblems all end with the partial-completion flexibility and the recurrence would be derived by considering how an optimal schedule could do the first task (rather than the last one). This will result in a recurrence like: $M[j] = \min\{c[j] + M[j+1]; h + M[j+4]\}$ with the boundary conditions something like $M[j] = 0$ if $j > n$. In the iterative algorithm, the $M[]$ array would then be filled in from the back (i.e. $M[n]$ first and $M[1]$ last).

A third way is padding the original sequence of tasks with three "dummy" cost-0 tasks so there are now $n+3$ tasks. This padding will not change the cost of the optimal solution, but the dummy tasks give a way to assign the handyman exactly 4 tasks at the end.

(b) (1 pts) Give an $O(n)$-time recursive algorithm with memoization for calculating the value of the recurrence.

```
Initialize M(0) through M(3) as described above
return CALCM(n)

function CALCM(j)
    if M(j) initialized then
        return M(j)
    else                                          ▷ j must be at least 4
        M(j) = min{c(j) + CALCM(j − 1); h + CALCM(n − 4)}
        return M(j)
    end if
end function
```

(c) (1 pt) Give an $O(n)$-time bottom-up algorithm for filling in the array

```
Initialize M(0) through M(3) as described above
for j = 4, . . . , n do
    M(j) = min{c(j) + M(j − 1); h + M(n − 4)}
end for
```

(d) (2 pts) Describe how to determine which tasks to do yourself, and which tasks to hire the handyman for in an optimal solution.

Trace back through the calculated array $M()$ to find when the handyman is used.

```
1: function TRACEBACK(j)
2:     if j=0 then
3:         return
4:     else if j < 4 then
5:         print owner does first j tasks
6:         return
7:     else
8:         if M(j) = c(j) + M(j − 1) then
9:             TRACEBACK(j − 1)
10:            print owner does task j
11:        else
12:            TRACEBACK(j − 4)
13:            print hire handyman for tasks j − 3 to j
14:        end if
15:    end if
16: end function
```

(Note: this order of the **print** statements and TRACEBACK calls causes the information to printed in task order. This is not necessary for the assignment.)