CSE 102 Spring 2021
Homework Assignment 5

Jaden Liu
University of California at Santa Cruz
Santa Cruz, CA 95064 USA

May 11, 2021

1 HW5

1. 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 = length[d]
2.  for i = 1 to n
3.      C[i, 0] = 0
4.  for i = 1 to n
5.      for j = 1 to N
6.          if i = 1 and j < d[1]
7.              C[1, j] = ∞
8.          else if i = 1
9.              C[1, j] = 1 + C[1, j − d[1]]
10.         else if j < d[i]
11.             C[i, j] = C[i − 1, j]
12.         else
13.             C[i, j] = min(C[i − 1, j], 1 + C[i, j − d[i]])
14. return C[n, N]
```

Write a recursive algorithm that, given the filled table $C[1 \cdots n; 0 \cdots N]$ as input, prints a sequence of $C[n, N]$ coin values whose total value is N. If it happens that $C[n, N] = \infty$, print a message to the effect that no such disbursement of coins is possible.

Solution. Since C[i][j] will always comes from C[i-1][j] or C[i][j-d[i]], so we just need to recursively see back the process then we will find all sequence of C[n,N].

If C[i][j] = C[i-1][j], it means that we didn't increase any more coin, so we don't need to print the coin now

If C[i][j] =C[i][j-d[i]], it means that we find a better solution by adding the present coin, so we need to print it out.

Keep finding until we reach the end of the table.
Here is pseudo code:

```
coin(C[][], n,N)
        if C[n][N] == $\infty$
                print("no such disbursement")
        if n<=0 or N<=0
                return
        if C[n][N] == C[n][N-d[n]]:
                print d[n]
                coin(C[][], n, N-d[n])
        else:
                coin(C[][], n-1, N)
```

□

2.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.)

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 for a.

```
knapsack(v[],w[],w,n)
        V[][] = int[w+1][v+1]
        for i=1 to n:
                for j=1 to w:
                        if j>w[i]:
                                V[i][j] = max(V[i-1][j],V[i-1][j-w[i]]+v[i])
                        else:
```

```
                                V[i][j] = V[i-1][j]
        return V[n][w]
```

□

Solution for b. The solution is similar to question 1. Since all V[i][j] comes from either V[i-1][j] or V[i-1][j-[wi]]+v[i]. Then we only needs to compare V[i][j] with these two values and we will get the souce of the value, or the objects to stolen.

If V[i][j] = V[i-1][j], that means we cannot put it in the bag in the subproblem. Thus, we don't need to save the object.

If V[i][j] = V[i-1][j-[wi]]+v[i], which means we have put the object i into our bag, save the value.

Keep recursively find in either V[i-1][j], or V[i-1][j-[wi]]+v[i] in the situation above until we reach the end of the table. □

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 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.

Proof for 3.1. Let P to be the optimal soltuion for $C[i]$, and let P' to be the subsequence of P = R(k,i)+C[k] in the path P. Assume, to the contradiction, that P' is not the optimal. Then there exist a P"=R(s,i)+C[s]. Then we can replace P' with P" to obtain a shorter path than P. This contradicts that P is the shortest path in the initial. Therefore, it satisfy the principle of optimality. □

Recurrence. $C[i] = min(R(k,i)+C[k])$, $1 < k < n$, so that k is the optimal minimum cost that makes R(k,i) + C[k] least. □

Algorithm.

```

fill(C[],i):
       for j=1 to i:
              if j<=1:
                     return 0;
```

```
                min = $\infty$;
                for l=1 to j:
                        temp = R(l,i)+C[i]
                        if temp < min:
                                min = temp
                C[j] = min
        return c[]
```

□

Parellel array. Adding a flag to denote the k in step min(R(k,i)+C[k]), and saving it into P[i]. Since P[i] was saved the last post, so we continue find in P[P[i]] until we reach the first post. The time to find single C[i] is O(n), the outer loop that find C[i] to C[n] also needs O(n), therefore, the runtimes for determining the cost of a cheapset seqeuence is $O(n^2)$.
While for the algorithm that determines the sequence, it cost O(n) in the worst case. O(1) in the best case, depending on the array P. □

(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 $c[1], \ldots, c[n]$.

- (3 pts) First, find a justify a recurrence (with boundary conditions) giving the optimal cost for completing the tasks.
- (1 pts) Give an $O(n)$-time recursive algorithm with memoization for calculating the value of the recurrence.
- (1 pt) Give an $O(n)$-time bottom-up algorithm for filling in the array
- (2 pts) Describe how to determine which tasks to do yourself, and which tasks to hire the handyman for in an optimal solution.

Solution for a. Let the optimal total cost is in array o[n]
$o[i] = o[i-4] + min(h, c[i-3]+c[i-2]+c[i-1]+c[i])$
If $i < 4$, $o[i] = sum$(c[j: for j=0 to i]).
Since $o(n)$ be the optimal solution, let $o(n-4)$ be the optimal solution for that. □

Solution for b. Set up a array o[n] to save the value during the recurrence.

```

o[] = int [n+1]
find(c[],n,h):
        if n<4:
                for j=0 to n:
                        sum += c[j]
                o[n] = sum
```

```
        else:
                o[n] = find(c[],n-4,h)+min(h,c[n-3]+c[n-2]+c[n-1]+c[n])
        return o[n]
```

□

Solution for c. Fill the array just use the recurrence in the solution a.

```

        fill(c[],n):
                o[] = int [n+1]
                for i=1 to n:
                        if i<4:
                                o[i] = sum(c[j: for j in range(i)])
                        else:
                                o[i] = o[i-4]+min(h,c[i-3]+c[i-2]+c[i-1]+c[i])
                return o[n]
```

□

Solution for d. First we need an array P and initialize all element to 0 to save the result. Thinking from the task after the last element, which can be considered as "n+1" task. Using the recurrence before, then we can compare task "n", "n-1", "n-2", "n-3" with h(if $n > 4$). If the first one is faster, then mark the four task to 1 in array P to represent these tasks needs to do by myself. Otherwise, stay them to 0. □

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. 1. Proof of optimality This problem exhibits optimal substructure. We can prove by contradiction. Let q[a][n] is optimal solution for the final result, that a is the value that make the profit largest in last column, q[b][n-1] be the optimal solution for the next to last line. Assume there

exist other optimal solution q[c][n-1], then we can replace q[b][n-1] with q[c][n-1] that contradicts that q[a][n] is the optimal solution, since q[a][n] is based on q[b][n-1]. 2. Recurrence Since no matter what, we need to move up a line. We can consider the optimal solution is saved in a 2-dimensional q[][], then the final solution is max(q[][n]) that is the largest value in the last column. The recurrence is to find the optimal solution in line below, e.g., q[n][n] = max(max[q[][n-1]]+p(dot_x,[n][n])). x represent the dot position that make the whole equation maximum. The pseudo code to find the maximum is like this:

```
1
2               len = len(p[][])
3   find(x,y):
4           if x<1 or x>len or y<1 or y>len:
5                   return negative infinity # represent illegal move
6           elif x=1:
7                   return 0
8           else:
9                   return max(find(x-1,y-1),find(x-1,y),find(x-1,y+1))
```

3. Algorithm We need to first fill up the checkerboard with the optimal profit for each square, then we can find the maximum profit path. The pseudo code to find maximum is like this:

```
1               #fill the checkerboard c[][]
2               len = len(p[][])
3   fill(p[][]):
4           for i=0 to len:
5                   for j=0 to len:
6                           if i = 0:
7                                   c[i][j] = 0
8                           else:
9                                   # first save the square pos
10                                  dot_x = max(c[i+1][j-1],c[i][j-1],c[i-1][j-1])
11                                  c[i][j] = p(dot_x, dot_[i][j]) + \
12                                  max(c[i+1][j-1],c[i][j-1],c[i-1][j-1])
```

4. Sequence Then if we want to know the path, just save the every result in each iteration when doing *find* in the first part.

```
1
2               len = len(p[][])
3               print_path(x,y):
4                       if c[x][y]=0:
5                               return
6                       print(x,y)
7                       print_path(max(c[i+1][j-1],c[i][j-1],c[i-1][j-1]))
```

5. Runtime The runtime of my algorithm is $O(n^2)$ □

References

[1] https://www.chegg.com/homework-help/questions-and-answers/2-read-coin-changing-problem-handout-dynamic-programming-solution-presented-reference-reca-q44597504