Homework 3

CS 325

Daniel Kim

---

**Problem 1.** (4 points)

What does dynamic programming have in common with divide-and-conquer? What is a principal difference between them?

---

Both dynamic programming and divide-and-conquer break a large problem into easier subproblems to obtain a solution. Also, both methods solve a problem via subproblems recursively.

Divide-and-conquer is a top-down approach that partitions the problem into disjoint or non-overlapping and independent subproblems and solves the subproblems recursively. In dynamic programming, subproblems overlap and share subproblems of subproblems, which allows increase efficiency. This sharing can be done in a top-down (memoization) or bottom-up approach and means that the same subproblem is not solved more than once. Subproblems for dynamic programming are not independent, and each subproblem depends upon the solution of respective subproblems. For dynamic programming, we have two requirements: optimal substructure and overlapping subproblems.

---

**Problem 2.** (6 points)

Shortest path counting: A chess rook can move horizontally or vertically to any square in the same row or the same column of a chessboard. Find the number of shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner. The length of a path is measured by the number of squares it passes through, including the first and the last squares. Solve the problem:

a) by a dynamic programming algorithm.

b) by using elementary combinatorics.

---

Let SP (i,j) = the number of rook's shortest paths from square(1,1) to square (i,j)

Rows: labeled 1 to 8

Columns: labeled 1 to 8

Initial state: a rook at the lower left corner of a chessobard at position at (1,1)

column, where 1 < i, j < 8.

Base case:

SP (i, 1) = SP (i,j) = 1 $for\ any\ 1 \leq i,\ j \leq 8$.

With a rook, we can only move in one direction at a time. For example, if we choose to move our rook up, it can only move up and not diagonally, which is both vertical and horizontal movement, across the board. In terms of row=i column=j, our rook with each move can only move by changing i with j constant or by changing j with i constant.

Each “shortest path” comprises of up movements and right movements (no backwards movements). As a result, shortest path or optimal path to SP(i,j) will be from the square with position (i-1,j) or (i,j-1) which are previous shortest paths.

So, we can define the shortest path to SP(i,j) as a recurrence:

SP (i,j) = SP (i , j-1) + SP (i-1,j) $for\ any\ 1 < i,\ j \leq 8$

SP (i, 1) = SP (1 ,j) = 1 $for\ 1 \leq i,\ j \leq 8$

Using the recurrence, we can compute the values of SP (i,j) for each square (i,j) of the board.

| | | | | | | | | **j** |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 8 | **1** | **8** | 36 | 120 | 330 | 792 | 1716 | Finish |
| 7 | **1** | **7** | 28 | 84 | 210 | 462 | 924 | 1716 |
| 6 | **1** | **6** | 21 | 56 | 126 | 252 | 462 | 792 |
| 5 | **1** | **5** | 15 | 35 | 70 | 126 | 210 | 330 |
| 4 | **1** | **4** | 10 | 20 | 35 | 56 | 84 | 120 |
| 3 | **1** | **3** | **6** | 10 | 15 | 21 | 28 | 36 |
| 2 | **1** | **2** | **3** | 4 | 5 | 6 | 7 | 8 |
| 1 | **Start** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| **i** | | | | | | | | |

| | | | | | | | | j |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 8 | **1** | **8** | 36 | 120 | 330 | 792 | 1716 | 3432 |
| 7 | **1** | **7** | 28 | 84 | 210 | 462 | 924 | 1716 |
| 6 | **1** | **6** | 21 | 56 | 126 | 252 | 462 | 792 |
| 5 | **1** | **5** | 15 | 35 | 70 | 126 | 210 | 330 |
| 4 | **1** | **4** | 10 | 20 | 35 | 56 | 84 | 120 |
| 3 | **1** | **3** | **6** | 10 | 15 | 21 | 28 | 36 |
| 2 | **1** | **2** | **3** | 4 | 5 | 6 | 7 | 8 |
| 1 | **Start** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| **i** | | | | | | | | |

An example of different paths to the top right corner.

| | | | | | | | | j |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 8 | 1 | 8 | 36 | 120 | 330 | 792 | 1716 | 3432 |
| 7 | 1 | **7** | 28 | 84 | 210 | 462 | 924 | 1716 |
| 6 | 1 | **6** | 21 | 56 | 126 | 252 | 462 | 792 |
| 5 | 1 | **5** | 15 | 35 | 70 | 126 | 210 | 330 |
| 4 | 1 | **4** | 10 | 20 | 35 | 56 | 84 | 120 |
| 3 | 1 | **3** | **6** | 10 | 15 | 21 | 28 | 36 |
| 2 | 1 | **2** | **3** | 4 | 5 | 6 | 7 | 8 |
| 1 | **Start** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| i | | | | | | | | |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 8 | 1 | 8 | 36 | 120 | 330 | 792 | 1716 | 3432 |
| 7 | 1 | **7** | 28 | 84 | 210 | 462 | 924 | 1716 |
| 6 | 1 | **6** | 21 | 56 | 126 | 252 | 462 | 792 |
| 5 | 1 | **5** | 15 | 35 | 70 | 126 | 210 | 330 |
| 4 | 1 | **4** | 10 | 20 | 35 | 56 | 84 | 120 |
| 3 | 1 | **3** | **6** | 10 | 15 | 21 | 28 | 36 |
| 2 | 1 | **2** | **3** | 4 | 5 | 6 | 7 | 8 |
| 1 | **Start** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| i | | | | | | | | |

| | | | | | | | | j |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 8 | **1** | **8** | 36 | 120 | 330 | 792 | (I, j-1) | Finish |
| 7 | **1** | **7** | 28 | 84 | 210 | 462 | 924 | (i-1,j) |
| 6 | **1** | **6** | 21 | 56 | 126 | 252 | 462 | 792 |
| 5 | **1** | **5** | 15 | 35 | 70 | 126 | 210 | 330 |
| 4 | **1** | **4** | 10 | 20 | 35 | 56 | 84 | 120 |
| 3 | **1** | **3** | **6** | 10 | 15 | 21 | 28 | 36 |
| 2 | **1** | **2** | **3** | 4 | 5 | 6 | 7 | 8 |
| 1 | Start | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| i | | | | | | | | |

b. Combinatorics:

Any shortest path to (8,8) will consist of 7 up moves and 7 right moves.

There are 14 total moves to get to the top right corner square at (8,8).

The different shortest paths to that position consists of 7 up moves.

So, the total possible shortest paths to the top right position is $_{14}C_7$ or C(14,7).

---

**Problem 3.** (6 points)

Maximum square submatrix: Given an $m \times n$ Boolean matrix $B$, find its largest square submatrix whose elements are all zeros. Design a dynamic programming algorithm and indicate its time efficiency. (The algorithm may be useful for, say, finding the largest free square area on a computer screen or for selecting a construction site.)

---

Subproblems:

- Given an m x n Boolean matrix B, we can say the smallest Boolean matrix of 0's would look like this:

| | j | |
|---|---|---|
| | 0 | 1 |
| 0 | **0** | **0** |
| 1 | **0** | **0** |
| i | | |

with dimensions i x j where i = j

The bottom right corner of this matrix at (i ,j) = (1,1) can only be part of a square matrix of 0's if

1. Position (i-1,j) = 0 ;
2. Position (i,j-1) = 0;

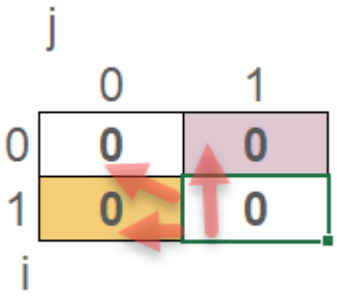
3. Position (i-1,j-1) = 0

In terms of the Boolean matrix B,

1. B[i-1][j]=0
2. B[i][j-1] = 0
3. B[i-1][j-1] =0

and the dimensions of that smallest sub-square has dimensions of i x j

where i > 1 and j >1 and i = j




- We need to somehow find the dimension of a sub-square S when a 0 is not the bottom-right corner of a sub- square of 0's.
- Let C be a i x j matrix (initialized to equal 0) that indexes the dimensions of a sub-square matrix with a bottom-right corner position at B[i][j] with dimensions of that sub-square being C[i][j] by C[i][j] (1 x 1 is not a square).
- At (1,1) the 0 is not a bottom-right corner of a sub-square of 0's. At (2,2), the square is the bottom-right corner of a 2 x 2 square. At (3,3), the square is of a 3 x 3 sub-square of 0's. To account for these squares:
    - if B[0][j] == 1 then C[i][j] = 0
    - If B[i][j] == 0 then C[i][j] = min(C[i-1][j], C[i][j-1], C[i-1][j-1]) + 1

Start of Matrix C

| i \ j | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | **0** | **0** | **0** |
| 2 | 0 | **0** | **0** | **0** |
| 3 | 0 | **0** | **0** | **0** |

Boolean Matrix B

| i \ j | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | **0** | **0** | **0** |
| 2 | **0** | **0** | **0** | **0** |
| 3 | **0** | **0** | **0** | **0** |

**Indexing Matrix C**

| i \ j | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | **1** | **2** | **2** |
| 2 | 1 | **1** | **2** | **3** |
| 3 | 1 | **2** | **2** | **3** |

We would solve our problem as follows:

```
MAX-SUBARRAY(B, m, n)
 initialize all index of C[m][n] = 0
 int max = 1 (because 1 x 1 matrix cannot be a square matrix)
 for i = 0 to m
    for j = 0 to n
        if B[i][j] == 1
            C[i][j] = 0
        If B[i][j] == 0
            C[i][j] = min(C[i-1][j], C[i][j-1], C[i-1][j-1]) + 1
            if C[i][j] > max
                max = C[i][j]
 return max
```

**Time Efficiency:**

- **Time complexity: O(m, n)** single pass.
- **Space complexity: O(m*n)** for another indexing matrix C with same dimensions as matrix B.

**Problem 4.** (14 points)

Consider the following instance of the knapsack problem with capacity W = 6

| Item | Weight | Value |
|---|---|---|
| 1 | 3 | $25 |
| 2 | 2 | $20 |
| 3 | 1 | $15 |
| 4 | 4 | $40 |
| 5 | 5 | $50 |

a) Apply the bottom-up dynamic programming algorithm to that instance.

b) How many different optimal subsets does the instance of part (a) have?

c) In general, how can we use the table generated by the dynamic programming algorithm to tell whether there is more than one optimal subset for the knapsack problem's instance?

d) Implement the bottom-up dynamic programming algorithm for the knapsack problem. The program should read inputs from a file called "data.txt", and the output will be written to screen, indicating the optimal subset(s).

e) For the bottom-up dynamic programming algorithm, prove that its time efficiency is in Θ(nW), its space efficiency is in Θ(nW) and the time needed to find the composition of an optimal subset from a filled dynamic programming table is in O(n).

---

a.

| | | | **Capacity = j** | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **$w_i$** | **$v_i$** | **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| | | **0** | **0** | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 25 | **1** | 0 | 0 | 0 | 25 | 25 | 25 | 25 |
| 2 | 20 | **2** | 0 | 0 | 20 | 25 | 25 | 45 | 45 |
| 1 | 15 | **3** | 0 | 15 | 20 | 35 | 40 | 45 | **60** |
| 4 | 40 | **4** | 0 | 15 | 20 | 35 | 40 | 55 | 60 |
| 5 | 50 | **5** | 0 | 15 | 20 | 35 | 40 | 55 | **65** |

**Max subset = V[5,6] = 65**
**With item 3, item 5**

| Item | Weight | Value |
|---|---|---|
| 1 | 3 | $25 |
| 2 | 2 | $20 |
| 3 | 1 | $15 |
| 4 | 4 | $40 |
| 5 | 5 | $50 |

b. The instance of W = 6 with the item weight and value combinations has a unique optimal solution only if the algorithm for obtaining an optimal subset, which can be traced from the final calculated value of V [n, W]=65, does not find an identical valid value from V [i − 1, j] to vi + V [i − 1, j − wi]. In this case, we can see that only 1 optimal subset exists because tracing back from 65 when i = 5 leads only to i = 4 then i = 3 where the item is weight3 = 1 and v3 = 15. There is no other combination that is equivalent or exceeds the value of 65.

c. As mentioned in part b, we can use the table to traceback from the optimal solution. If we reach a path where there are multiple optimal subsets that equal to our value of final optimal value. Then, that would be our optimal solution is not unique.

d. Submitted on TEACH

e) Executing comparisons in the conditional statements is spending time Θ(1). Moving through the array (table) with dimensions columns = (W+1) and rows = ( n+1) takes Θ(nW) time.

Implementation: (actual code on TEACH)

```
7  def knapsack_BU(W, weight, value, n):
8      #V is a 2D array with [values][weight]
9      cols = W + 1
10     rows = n + 1
11     V = [[0 for x in range(cols)] for x in range(rows)]
12     for i in range(rows):
13         for w in range(cols):
14             if i == 0 or w == 0:
15                 V[i][w] = 0
16             elif weight[i-1] <= w:
17                 V[i][w] = max( V[i-1][w-weight[i-1]] + value[i-1],  V[i-1][w])
18             else:
19                 V[i][w] = V[i-1][w]
20     return V[n][W]
```

---

EXTRA CREDIT (4 points)

Implement an algorithm that finds the composition of an optimal subset from the table generated by the bottom-up dynamic programming algorithm for the knapsack problem.

---

**Programs can be written in C, C++ or Python, but all code must run on the OSU engr servers. Submit to TEACH a copy of all your code files and a README file that explains how to compile and run your code in a ZIP file. We will only test execution with an input file named data.txt.**

EXTRA CREDIT: Uploaded to TEACH with algorithm code