Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Ki Min Kang Wisc id: 908 404 5062

More Dynamic Programming

Do **NOT** write pseudocode when describing your dynamic programs. Rather give the Bellman Equation, describe the matrix, its axis and how to derive the desired solution from it.

1. *Kleinberg, Jon. Algorithm Design (p. 327, q. 16).*

   In a hierarchical organization, each person (except the ranking officer) reports to a unique superior officer. The reporting hierarchy can be described by a tree T, rooted at the ranking officer, in which each other node v has a parent node u equal to his or her superior officer. Conversely, we will call v a direct subordinate of u.

   Consider the following method of spreading news through the organization.

   - The ranking officer first calls each of her direct subordinates, one at a time.
   - As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates, one at a time.
   - The process continues this way until everyone has been notified.

   Note that each person in this process can only call *direct* subordinates on the phone.

   We can picture this process as being divided into rounds. In one round, each person who has already heard the news can call one of his or her direct subordinates on the phone. The number of rounds it takes for everyone to be notified depends on the sequence in which each person calls their direct subordinates.

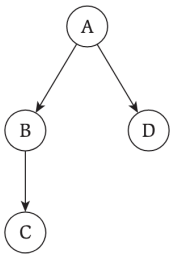

Figure 1: A hierarchy with four people. The fastest broadcast scheme is for A to call B in the first round. In the second round, A calls D and B calls C. If A were to call D first, then C could not learn the news until the third round.

Give an efficient algorithm that determines the minimum number of rounds needed for everyone to be notified, and outputs a sequence of phone calls that achieves this minimum number of rounds by answering the following:

(a) Give a recursive algorithm. (The algorithm does not need to be efficient)

1. If an officer doesn't have any subordinates, the process is over.
2. For an officer with subordinates, the algorithm calls itself for each subordinate to find out how many rounds it takes for them to get the news
3. It keeps track of all these numbers and then figures out the longest time it would take for any one subordinate to be informed.
4. It sorts all the subordinates based on how quickly they can spread the news, so the one who can do it the fastest gets called first.
5. In the end, the algorithm tells the maximum number of rounds needed and the best order to make the calls to spread the news ASAP

(b) Give an efficient dynamic programming algorithm.

Let R(v) as the minimum number of rounds it takes for the message to reach all nodes in a subtree with a root node v.

we will use the Bellman equation: $R(v) = \max_{i=1,\dots,n} (i + R(S_v[i]))$, where R(v) (the minimum rounds for node v) is the maximum value of i (the order of a subordinate) plus R(S[i]) (the rounds needed for the subordinate)

For the implementation, we think of the S_v as a tree and traverse this tree using a stack. we add nodes to the stack and take them off as we go, ensuring we visit each node and calculate the minimum rounds using the equation.

When it comes to the runtime, we have n nodes and sorting costs us $O(n \log n)$ each time

(c) Prove that the algorithm in part (b) is correct.

We will prove the algorithm by strong induction.

Suppose we have a sequence of calls. Each call is to subordinate who will need some rounds to pass the message along. The sequence tells us the order in which the person makes these calls. Imagine the sequence isn't sorted, which means we might not be calling the quickest informers first. If we swap two people in the sequence and this swap doesn't increase the total rounds needed, it means the order isn't crucial for those two. But if arranging them in decreasing order gives us the smallest number, then this order is best one.

So if we have two subordinates, and one can inform their team faster than the other, calling the faster one is always better or as good as than calling slower one first.

This proves that the best strategy is to always call the quickest informers first, indicating the algorithm is correct.

2. Consider the following problem: you are provided with a two dimensional matrix M (dimensions, say, $m \times n$). Each entry of the matrix is either a **1** or a **0**. You are tasked with finding the total number of square sub-matrices of M with all **1**s. Give an $O(mn)$ algorithm to arrive at this total count by answering the following:

   (a) Give a recursive algorithm. (The algorithm does not need to be efficient)

Assume we have a grid M with rows and columns filled with 1s and 0s.

If the grid is empty then there are no squares, the answer is 0.
Otherwise, it looks for squares in the rest of the grid by calling itself recursively.
Then for each column in the first row, it counts how many squares of 1s can be formed with top-left corner at the first row and current column. It does this by checking smaller squares until it hits a 0 or runs out of grid. It adds up all these counts to give us the total number of squares.

   (b) Give an efficient dynamic programming algorithm.

We will use a helper matrix C, where each element C[i,j] represents the size of the largest submatrix ending at M[i,j] that is filled with all 1s.

The dynamic programming relation we'll use to fill out C is:
If we are at the first row or first column of M, then C[i,j] can only be the value of M[i,j] itself because there are no other elements to form a larger square.

If the element M[i,j] is 0, then no square submatrix with all 1s can end at that element, so C[i,j] is 0. If the element M[i,j] is 1, then we check the sizes of the largest squares directly to the left of it, above and to the left. The size of the largest square ending at M[i,j] is 1 plus the smallest of these 3 numbers.

Once we have C filled out, the total number of square submatrices in M with 1s is simply the sum of all the values in C.

The runtime of the algorithm is O(mn) because we do only a constant amount of work for each element in M.

(c) Prove that the algorithm in part (b) is correct.

- the size of the largest square sub-matrix ending at cell M[i][j] depends on the sizes of the largest square sub-matrices ending at M[i-1][j], M[i][j-1] and M[i-1][j-1] because any larger square sub-matrix at M[i][j] must expand.
- If M[i][j] is 0, then the largest possible square sub-matrix at this cell is also 0, because a square of all 1s can't have a 0 in any corner.
- If M[i][j] is 1, the largest square sub-matrix is limited by the smallest of the three adjacent squares because if one cell can only support a smaller square, adding M[i][j] to that square won't make it any bigger.
- We find the minimum of the three values C[i-1][j], C[i][j-1] and C[i-1][j-1] and add 1 for the cell itself if M[i][j] is 1. This gives us the size of the largest square sub-matrix at M[i][j]

We calculate this recursively for every cell and the sum of all C[i,j] values will give us the total number of squares of all 1s in the matrix.

(d) Furthermore, how would you count the total number of square sub-matrices of M with all **0**s?

1. change every 1 in matrix M to a 0, and every 0 to a 1. Now, our new matrix M' is the opposite of M.
2. Apply the algorithm that we use to find submatrices fulls of ones to M.

3. *Kleinberg, Jon. Algorithm Design (p. 329, q. 19).*

String x' is a *repetition* of x if it is a prefix of x^k (k copies of x concatenated together) for some integer k. So $x' = 10110110110$ is a repetition of $x = 101$. We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences x' and y', so that x' is a repetition of x and y' is a repetition of y. For example, if $x = 101$ and $y = 00$, then $s = 100010010$ is an interleaving of x and y, since characters $1, 2, 5, 8, 9$ form 10110—a repetition of x—and the remaining characters $3, 4, 6, 7$ form 0000—a repetition of y.

Give an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y by answering the following:

(a) Give a recursive algorithm. (The algorithm does not need to be efficient)

IsInterleaved(s, x, y):

- If 's' is empty, then return True because an empty string is an interleaving of two empty strings
- If the first character of 's' doesn't match the first character of 'x' or 'y', then return False. Because 's' cannot start with a character that isn't in 'x' or 'y'.
- Initialize two variables, 'interleaveX' and 'interleaveY', as False. These will be used to store the results of recursive calls
- If the first character of 's' matches the first character of 'x', call the function recursively with the rest of 's' and 'x', keeping 'y' the same. Store the result in 'interleaveX'.
- Similarly, if the first charater of 's' matches the first character of 'y', call the function recursively with the rest of 's' and 'y', keeping 'x' the same. Store the result in 'interleaveY'.
- Return the logical OR of 'interleaveX' and 'interleaveY'. This means that if either recursive call returns True, the function will return True, indicating that 's' is an interleaving of 'x' and 'y'.

(b) Give an efficient dynamic programming algorithm.

- Initialize a table 'dp' where 'dp[i][j]' will be true if the first 'i+j' characters of 's' can be created by interleaving the first 'i' and 'j' characters
- We will fill in the table starting from 'dp[0][0]', which is true.
- If the 'i'th character of 's' is the same as the 'i'th character of 'x', and if dp[i-1][j] is true, then 'dp[i][j]' should also be ture. If the 'j'th character of 's' is the same as the 'j'th character of 'y', and if 'dp[i][j-1]' is true, then 'dp[i][j]' should also be true.
- If neighter condition is met, we set 'dp[i][j]' to false.

(c) Prove that the algorithm in part (b) is correct.

The base cases obviously work.

We know that for some parts of 's', we can shuffle smaller pieces of 'x' and 'y' to match. If we can add one more character from 's' to the mix, and it fits by coming next in either 'x' or 'y', then the algorithm still holds.

Our algorithm relies on the logical step: If a smaller part works, and we can add the next step without breaking the rules, then the whole thing works. We repeat the reasoning for every character in 's', which proves our method is correct.

4. *Kleinberg, Jon. Algorithm Design (p. 330, q. 22).*

To assess how "well-connected" two nodes in a directed graph are, one can not only look at the length of the shortest path between them, but can also count the number of shortest paths.

This turns out to be a problem that can be solved efficiently, subject to some restrictions on the edge costs. Suppose we are given a directed graph $G = (V, E)$, with costs on the edges; the costs may be positive or negative, but every cycle in the graph has strictly positive cost. We are also given two nodes $v, w \in V$.

Give an efficient algorithm that computes the number of shortest $v - w$ paths in G. (The algorithm should not list all the paths; just the number suffices.)

Assume we have a map of one-way streets connecting various points and want to find the quickest way to get from point A to point B, but also how many quickest routes there are.

1. Initialize two lists for every point on the map. One keeps the shortest distance to get from A to the point, and the other list counts how many shortest routes are there.
2. At the start, say the shortest distance from A to A is 0 and there's 1 way to stay put. For all other points, say we don't know the shortest distance yet, and there are 0 ways to get there
3. Now, go through the streets. If a street leads directly to B, note down the distance and say there's 1 way to get there. If a street goes somewhere else, check if it offers a shorter route to that point or if it's another shortest route to the point. Keep updating the lists
4. After we've checked all streets, look at list for point B. The distance tells how quickly we can get there, and the count tells how many ways to get there.

This way, we systemically can find the quickest routes.

5. The following is an instance of the Knapsack Problem. Before implementing the algorithm in code, run through the algorithm by hand on this instance. To answer this question, generate the table, indicate the maximum value, and recreate the subset of items.

item	weight	value
1	4	5
2	3	3
3	1	12
4	2	4

Capacity: 6

	W=0	W=1	W=2	W=3	W=4	W=5	W=6
Item 0	0	0	0	0	0	0	0
Item 1	0	0	0	0	5	5	5
Item 2	0	0	0	3	5	5	5
Item 3	0	12	12	12	15	17	17
Item 4	0	12	12	16	16	17	19

Item 4 with W=6 contains the maximum value 19.

6. **Coding Question: Knapsack**

   Implement the algorithm for the Knapsack Problem in either C, C++, C#, Java, Python, or Rust. Be efficient and implement it in $O(nW)$ time, where n is the number of items and W is the capacity.

   The input will start with an positive integer, giving the number of instances that follow. For each instance, there will two positive integers, representing the number of items and the capacity, followed by a list describing the items. For each item, there will be two nonnegative integers, representing the weight and value, respectively.

   A sample input is the following:

```
2
1 3
4 100
3 4
1 2
3 3
2 4
```

   The sample input has two instances. The first instance has one item and a capacity of 3. The item has weight 4 and value 100. The second instance has three items and a capacity of 4.

   For each instance, your program should output the maximum possible value. The correct output to the sample input would be:

```
0
6
```