CS 577 # Assignment 8 – Dynamic Programming 2 Spring 2023

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: ____________________ Wisc id: ____________________

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.

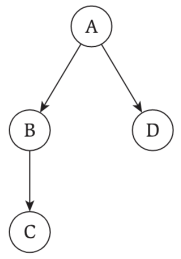

Figure 1: A hierarchy with four people. The fastest broadcast scheme is for A to call B in the first round. In the second round, A calls D and B calls C. If A were to call D first, then C could not learn the news until the third round.

The questions are on the next page.

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)

Solution:

We will represent schedule as a list of lists S that contains the calls done at round i at $S[i]$.

Algorithm 1: CallSchedule

```
Input  : An officer $r$ with subordinates $[s_1, \dots, s_n]$
Output: (Number of rounds required, schedule of call)
if $n = 0$ then
    return $(0, [])$;
end
$L \leftarrow []$;
foreach $s_i$ do
    $L.append($CallSchedule($s_i$)$)$
end
Sort $L$ by the first entry;
$N \leftarrow \max(L[i][1] + i)$;
$S \leftarrow$ schedule: at round 1, $s_1$; at round 2, $s_2$ and round 1 of of $s_1$'s schedule, ...;
return $(N, S)$;
```

(b) Give an efficient dynamic programming algorithm.

Solution:

Let $R(v)$ denote the minimum number of rounds neede to contact all nodes in the subtree rooted at v. Let $S_v[1 \dots n]$ denote the list of direct subordinates i of v ordered by decreasing value of $R(i)$. Then, the Bellman equation can be give by:

$$R(v) = \max_{i=1,\dots,n} \Big(i + R(S_v[i]) \Big) \tag{1}$$

Then, the solution to the problem is R(ranking officer).

To retrieve the schedule, we can think of S_v memoized as a tree. We can keep a stack of nodes such that its children are not exhausted. We initialize the stack with the top officer. At each round we pop off the all the nodes in the list. For all each of those nodes v, we add the next children c in S_v not yet considred to the schedule at that round. And we add c to the stack. And if there are still more children left for v, we add it back to the stack. We repeat until the stack is empty.

Let m denote the number of nodes in the tree. A naïve analysis of runtim can give $O(m^2 \log m)$, since there are m entries, and we sort each time taking $O(m \log m)$. A more careful analysis can yield $O(m \log m)$. Note that the work we actually do to compute each $R(v)$ is $O(n_v \log n_v)$ if n_v denotes the number of children of v. Since each node has a unique parent, the total work is $\sum_v O(n_v \log n_v)$ with $\sum_v n_v \approx m$. Note that for any $a, b > 0$, we have $a \log a + b \log b \leq a \log(a+b) + b \log(a+b) \leq (a+b) \log(a+b)$. So, the total work done is $O(m \log m)$ for computing R. The backtracing step takes linear time because the number of times each node gets added to the stack is equal to the number of children it has. In other words, each node adds its parent to the stack at most once, so it takes linear time with respect to m. Therefore in total, the runtime is $O(m \log m)$.

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

Solution:

The correctness follows from the correctness of the Bellman equation. We can prove the correctness by strong induction. Suppose we are considering the sequence of the calls for a node v. Given any sequence of calls by v, say $(u_1, ..., u_n)$, the minimal possible rounds needed for the sequence is at least each of $i + R(u_i)$, since u_i needs to make calls for its subordinates after it gets called by v at the ith round. Therefore, $R(v) = \min_{(u_1,...,u_n)} \max_{i=1,...,n} i + R(u_i)$.
Now, suppose $U = (u_1, ..., u_n)$ is not sorted by the decreasing R values. That means there is some indicies $i < j$ such that $R(u_i) < R(u_j)$ and also $i + R(u_i) < j + R(u_j)$. Consider a new sequence $W = (w_1, ..., w_n)$ that is same as U except we swap the u_i and u_j. Then, $R(w_i) + i = R(u_j) + i < R(u_j) + j$ and $R(w_j) + j = R(u_i) + j < R(u_j) + j$. Therefore, removing an inversion in the sequence U does not increase the maximum value achieved. By the exchange argument, we can conclude that the minimal value is achieved when the sequence is sorted by the decreasing R value.

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)

Solution:

```
Algorithm 2: SqSub
  Input  : $m \times n$ binary matrix $M$
  Output: Number of square submatrix of $M$ with all 1
  if $m = 0$ then
      return 0;
  end
  $N \leftarrow$ SqSub($M[2 \dots m][1 \dots n]$);
  foreach $i \in [1 \dots n]$ do
      $k \leftarrow$ number of square submatrix with all 1 having $M[1, i]$ as its upper left corner;
      $N \leftarrow N + k$;
  end
  return $N$;
```

   (b) Give an efficient dynamic programming algorithm.

Solution:

Let $C(i, j)$ denote the side length of the biggest square submatrix with **1**s with bottom-right corner at $M[i][j]$. Note that this number $C(i, j)$ also counts the number of square submatrix with **1**s with bottom-right corner at $M[i][j]$.

$$C(i,j) = \begin{cases} M[i][j] & \text{if } i = 0 \text{ or } j = 0 \\ 0 & \text{if } M[i][j] = 0 \\ \min(C(i-1,j), C(i,j-1), C(i-1,j-1)) + 1 & \text{if } M[i][j] = 1 \end{cases}$$

We can solve this recurrence relation directly using a recursive algorithm. The solution is then $\sum_{i,j} C(i, j)$.
We are essentially filling up a table of size $m \times n$ with constant amount of work for each entry. Therefore, the runtime is $O(mn)$.

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

Solution:

The recurrence relation given above is correct because the largest square **1** matrix with $M[i][j]$ at the bottom-right corner is constrained by the largest such matrices at $M[i-1][j]$, $M[i][j-1]$, and $M[i-1][j-1]$. More formally, suppose length ℓ square is the largest **1** square matrix we can fit with bottom-right corner at $M[i][j]$. This square also includes three length $\ell - 1$ **1** squares with bottom-right corner at $M[i-1][j]$, $M[i][j-1]$, and $M[i-1][j-1]$. Therefore, $C(i,j) \leq C(i-1,j)+1$, $C(i,j) \leq C(i,j-1)+1$, and $C(i,j) \leq C(i-1,j-1)+1$, so $C(i,j) \leq \min(C(i-1,j), C(i,j-1), C(i-1,j-1)) + 1$. For the other direction, we can see one of the above three inequalities must be actually equality, since if not, we would be able to fit a length ℓ squares at each of those three corners, giving us a length $\ell + 1$ square fitting at the corner $M[i][j]$. That is equivalent to taking the minimum of the three C values.

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

Solution:

We just need to convert M to M' that has opposite entries of M and use our algorithm on 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)

Solution:

Assume we have long enough $X = x^k$ and $Y = y^k$ for some big enough k.

Algorithm 3: Interleave

```
Input  : Strings $s$, $X$, and $Y$
Output: Whether $s$ is an interleaving of $X$ and $Y$ or not
if $length(s) = 0$ then
    return True;
end
if $s[0] \neq X[0] \wedge s[0] \neq Y[0]$ then
    return False;
end
$b_X, b_Y \leftarrow$ False;
if $s[0] = X[0]$ then
    $b_X \leftarrow$ Interleave($s, X[2\ldots], Y$);
end
if $s[0] = Y[0]$ then
    $b_Y \leftarrow$ Interleave($s, X, Y[2\ldots]$);
end
return $b_X \vee b_Y$;
```

(b) Give an efficient dynamic programming algorithm.

Solution:

Let x^* and y^* be the infinite repetition of x and y respectively. Let $S(i, j)$ denote whether the substring $s_1 s_2 \ldots s_{i+j}$ is an interleaving of $x_1^* \ldots x_i^*$ and $y_1^* \ldots y_j^*$.

$$S(i,j) = \Big[S(i-1,j) \wedge \big(s_{i+j} == x_i^*\big)\Big] \vee \Big[S(i,j-1) \wedge \big(s_{i+j} == y_j^*\big)\Big] \tag{2}$$

with base case $S(0, 0) = 1$, $S(i, 0) = (s_i == x_i^*)$ and $S(0, j) = (s_j == y_j^*)$. The output will be true if there is some $i + j = \ell$ with $S(i, j)$ true. Note that we only need finite intial segment of x^* and y^*. Also, note that $x_i^* = x_{i \bmod \ell}$ where ℓ is the length of x, so it can be computed in constant time. Therefore, it takes $O(n^2)$ time since i and j are in the range 0 to n where n is the length of s.

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

Solution:

The base cases are clearly true. Assume that $S(i, j-1)$ and $S(i-1, j)$ return the correct result. The string $s_1 \ldots s_{i+j}$ can be interleaved with $x_1^* \ldots x_i^*$ and $y_1^* \ldots y_j^*$ if and only if the last character, s_{i+j} matches with one of x_i^* or y_j^*, and the substring $s_1 \ldots s_{i+j-1}$ can be interleaved with the remaining of x^* and y^*. The recurrence relation expresses this statement. Also, we check for all possible repeitions of x and y.

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.)

Solution:

We keep track of two $2D$ matrices C and M such that $C[n][i]$ the cost of the shortest path from i to w of length exactly n, and $M[n][i]$ is the number of such paths. We initialize $C[1][i] = c_{iw}$ and $M[1][i] = 1$ if there is an edge $(i, w) \in E$ and $C[1][i] = \infty$ and $M[1][i] = 0$ otherwise.

We update the entries of M and C as follows: Let

$$C[n][i] = \min_{j \in V} \{c_{ij} + C[n-1][j]\}$$

and

$$M[n][i] = \sum_j M[n-1][j] \quad \text{for } j \in V \text{ with } c_{ij} + C[n-1][j] = C[n][i]$$

To find the solution, we first compute the cost of the shortest path from v to w, which is $c = \min_{1 \le n \le |V|-1} C[n][v]$. Then, for each j with $C[j][v] = c$, we return $m = \sum_j M[j][v]$.

We show correctness. Fix some n. Suppose c is the cost of the shortest path from i to w of length n. And suppose m is the number of such paths. Then, if $p_1, p_2, \ldots, p_m$ are those paths, the cost of each one of them are c and they all have length n. We can partition them disjointly by the first vertex visited after i. Suppose without loss of generality that $p_1, \ldots, p_k$ start with the edge (i, j). If $p'_1, \ldots, p'_k$ denote the paths obtained by removing the first edge, then they must be the shortest path from j to w of length $n-1$. Then, by an inductive argument, $k = M[n-1][j]$ and the cost of each of the must be $C[n-1][j] = c - c_{ij}$. Also, since there is no negative cycle, all shortest paths must have length less than n, so the algorithm counts them all.

5. The following is an instance of the Knapsack Problem. Before implementing the algorithm, 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

Solution:

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

Max value: 19

Items used: 2, 3, 4

6. Implement the algorithm for the Knapsack Problem in either C, C++, C#, Java, or Python. 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
```