CS 577 # Assignment 4 – Dynamic Programming Summer 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: Gurusharan Kunusoth Wisc id: KUNUSOTH

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.313 q.2).*

   Suppose you are managing a consulting team and each week you have to choose one of two jobs for your team to undertake. The two jobs available to you each week are a low-stress job and a high-stress job.

   For week i, if you choose the low-stress job, you get paid ℓ_i dollars and, if you choose the high-stress job, you get paid h_i dollars. The difference with a high-stress job is that you can only schedule a high-stress job in week i if you have no job scheduled in week $i-1$.

   Given a sequence of n weeks, determine the schedule of maximum profit. The input is two sequences: $L := \langle \ell_1, \ell_2, \ldots, \ell_n \rangle$ and $H := \langle h_1, h_2, \ldots, h_n \rangle$ containing the (positive) value of the low and high jobs for each week. For Week 1, assume that you are able to schedule a high-stress job.

   (a) Show that the following algorithm does not correctly solve this problem.

Algorithm: JOBSEQUENCE

Input : The low (L) and high (H) stress jobs.
Output: The jobs to schedule for the n weeks
for *Each week i* **do**
  if $h_{i+1} > \ell_i + \ell_{i+1}$ **then**
    Output "Week i: no job"
    Output "Week i+1: high-stress job"
    Continue with week i+2
  else
    Output "Week i: low-stress job"
    Continue with week i+1
  end
end

$L = \langle 1, 2, 3 \rangle$

$H = \langle 100, 5, 4 \rangle$

By algorithm, we can say that

$0 + h_2 + L_3 = 8$

and with schedule $= h_1 + L_2 + L_3 = 107$

(b) Give an efficient algorithm that takes in the sequences L and H and outputs the greatest possible profit.

We can say max profit for each week using this recurrence relation: maxProfit(i) = max(maxProfit(i-1) + H[i], maxProfit(i-2) + L[i])
The base cases are:
maxprofit(1) = H[1] (since it is given that a high stress job can be scheduled)
maxprofit(2) = H[2] (choosing high stress job in week 2)
To implement this, use array dp[]. dp[1] & dp[2] are the best cases mentioned above. Then we iterate from i = 3 to n, applying the recurrence relation to compute dp[i]. Finally return dp[n], which represents the maximum profit for the entire sequence of n weeks.

(c) Prove that your algorithm in part (c) is correct.

Use induction. Base cases: Algorithm correctly initializes dp[1] and dp[2] according to the base cases. So, base cases hold.
Inductive step: Assume algo correct for first i-1 weeks, where i >= 3. By induction, dp[i-1] represents the maximum profit for the first i-1 weeks. Consider 2 cases:
Case1: choosing a high-stress job in week i: Total profit to week i is dp[i-2] + L[i]. This is valid because we can schedule a high-stress job in week i.
Case2: choosing a low-stress job in week i: total profit upto week i is dp[i-2] + L[i]. valid because we cannot schedule a high-stress job in week i, but we can choose a low-stress job.

By considering both cases, we compute dp[i] as the maximum between dp[i-1] + H[i] and dp[i-2] + L[i], which represents the max profit for the first i weeks. Therefore, by induction, the algorithm correctly computes the greatest possible profit for the given sequence of weeks.

2. *Kleinberg, Jon. Algorithm Design (p.315 q.4).*

Suppose you're running a small consulting company. You have clients in New York and clients in San Francisco. Each month you can be physically located in either New York or San Francisco, and the overall operating costs depend on the demands of your clients in a given month.

Given a sequence of n months, determine the work schedule that minimizes the operating costs, knowing that moving between locations from month i to month $i+1$ incurs a fixed moving cost of M. The input consists of two sequences N and S consisting of the operating costs when based in New York and San Francisco, respectively. For month 1, you can start in either city without a moving cost.

(a) Give an example of an instance where it is optimal to move at least 3 times. Explain where and why the optimal must move.

$N = \langle 0, 10M, 0, 10M, 0, 10M \rangle$

$S = \langle 10M, 0, 10M, 0, 10M, 10 \rangle$

So, it is optimal to switch to 0 operating cost each month.

(b) Show that the following algorithm does not correctly solve this problem.

Algorithm: WORKLOCSEQ

Input : The NY (N) and SF (S) operating costs.
Output: The locations to work the n months
for *Each month i* **do**
  if $N_i < S_i$ **then**
    Output "Month i: NY"
  else
    Output "Month i: SF"
  end
end

$N = \langle \frac{M}{2}, M, \frac{M}{2} \rangle$

$S = \langle M, \frac{M}{2}, M \rangle$

The algorithm would cost. $\frac{M}{2} + M + \frac{M}{2} + M + \frac{M}{2} = \frac{7}{2}M$.

But if we stay in New York for 3 months, we cost $\frac{M}{2} + \frac{M}{2} + M = 2M$

(c) Give an efficient algorithm that takes in the sequences N and S and outputs the value of the optimal solution.

1. create two arrays, DP_N and DP_S of length n+1 to store the optimal costs for being in New York and San Fran.
2. Initialize DP_N[1] and DP_S[1] with the corresponding operating costs for month 1.
3. For each month i from 2 to n, calculate the optimal costs for being in NY & SF: DP_N[i] = min(DP_N[i-1], DP_S[i-1]+M) + N[i]
DP_S[i] = min(DP_S[i-1], DP_N[i-1] + M) + S[i].
4. The optimal solution will be minimum value between DP_N[n] & DP_S[n].

(d) Prove that your algorithm in part (c) is correct.

We need to show that the optimal substructure & overlapping subproblems properties hold.
optimal structure: consider the optimal solutions for month j. The cost of being in new york for month j will be the min. of 2 options:
1. The cost of being in New york for month i-1 plus the operating cost N[i] for month i.
2. The cost of being in S.F for months i-1 plus the moving cost M and the operating cost N[i] for month j.
The same logic applies to the cost of being in S.F for month i.
Therefore, optimal solution for month i depends on the optimal solution for months i-1, and the subproblem solutions leads to the overall optimal solutions.
Overlapping subproblems:
Algorithm calculates the optimal costs for each month by considering the optimal costs for the previous months. As a result, the algorithm solves the same subproblems multiple times. However, by storing the computed values in the DP_N & DP_S arrays, the algorithm avoids redundant calculations and ensures that each subproblem is solved only once.

3. *Kleinberg, Jon. Algorithm Design (p.333, q.26).*

Consider the following inventory problem. You are running a company that sells trucks and predictions tell you the quantity of sales to expect over the next n months. Let d_i denote the number of sales you expect in month i. We'll assume that all sales happen at the beginning of the month, and trucks that are not sold are stored until the beginning of the next month. You can store at most s trucks, and it costs c to store a single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee k each time you place an order (regardless of the number of trucks you order). You start out with no trucks. The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands $\{d_i\}$, and minimize the costs. In summary:

- There are two parts to the cost: (1) storage cost of c for every truck on hand; and (2) ordering fees of k for every order placed.
- In each month, you need enough trucks to satisfy the demand d_i, but the number left over after satisfying the demand for the month should not exceed the inventory limit s.

(a) Give a recursive algorithm that takes in s, c, k, and the sequence $\{d_i\}$, and outputs the minimum cost. (The algorithm does not need to be efficient.)

Recursive formula:
MinCost(i, t) = min(minCost(i-1, t+x) + c(t+x-di) + k) = Max(0, di-t) to s-
Minimum cost is obtained by considering all possible values of x, calculating the cost for each scenario & choosing the maximum.

(b) Give an algorithm in time that is polynomial in n and s for the same problem.

Iterative algorithm:
```
function minCost(s, c, k, d)
    n = length(d)
    dp = create a 2D array of size (n+1) x (s+1)
    for t=0 to s:
        dp[0][t] = 0
    for i = 1 to n:
        for t=0 to s:
            min_cost = infinity
            for x = max(0, di - t) to s:
                cost = dp[i-1][t+x] + c(t+x-di) + k
                min_cost = min(min_cost, cost)
            dp[i][t] = min_cost
    return dp[n][0].
```

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

need to show that the computed dp[n][0] represents the minimum cost required to satisfy all demands $\{d_1, d_2 \cdots d_n\}$ with no excess trucks in inventory.

using induction on the month i:

1) Base case: For $i=0$, there are no demands to satisfy, and the cost is 0. $dp[0][t]=0$ for all t.

2) Inductive step:

Assuming $dp[i-1][t']$ represents the minimum cost for satisfying demands $\{d_1 \cdots d_{i-1}\}$ with t' trucks in inventory, we show that $dp[i][t]$ represents min cost for satisfying demands $\{d_1 \cdots d_i\}$ —

- For each t, we consider all possible values of x from $\max(0, d_i - t)$ to s. This ensures that the number of trucks left over after satisfying the demand d_i doesn't exceed the inventory limit s.
- we calculate the cost of ordering x trucks, storing them for one month and satisfying the demand d_i, while considering the min cost obtained so far.
- By storing the min cost in $dp[i][t]$, we ensure that the computed value represents the min cost for satisfying demands $\{d_1, d_2 \cdots d_i\}$ with t trucks in inventory.

4. Alice and Bob are playing another coin game. This time, there are three stacks of n coins: A, B, C. Starting with Alice, each player takes turns taking a coin from the top of a stack – they may choose any nonempty stack, but they must only take the top coin in that stack. The coins have different values. From bottom to top, the coins in stack A have values $a_1, \ldots, a_n$. Similarly, the coins in stack B have values $b_1, \ldots, b_n$, and the coins in stack C have values $c_1, \ldots, c_n$. Both players try to play optimally in order to maximize the total value of their coins.

   (a) Give an algorithm that takes the sequences $a_1, \ldots, a_n, b_1, \ldots, b_n, c_1, \ldots, c_n$, and outputs the maximum total value of coins that Alice can take. The runtime should be polynomial in n.

- initialize scoreA, scoreB, scoreC to 0;
- Initialize pointers A, B, C to n-1, the current player to Alice
- ~~initialize~~ Repeat until all stacks are empty:
  a. if currentplayer is Alice:
     calculate maxScore = max(scoreA + a_i, scoreB + b_i, scoreC + c_i)
     Add maxscore to Alice total score
     Increment the pointer corresponding to maxscore
     Reset scoreA, B, C to 0
     set current player to Bob
  b. if currentplayer is Bob; calculate maxscore = max(scoreA - a_i, scoreB - b_i, scoreC - c_i)
     incre
  Return Alice total score.

   (b) Prove the correctness of your algorithm in part (a).

For n = 1, the algo correctly chooses the stack with the highest-valued coin. Assume the algorithm works correctly for n=k, where k > 1. For n = k+1, Alice chooses the stack with the highest value. Bob then minimizes alices score. In the next round, Alice has two options: 1) choosing the stack with the second highest valued coin. 2) Choosing the stack with the highest-value coin, where bob is forced to choose the second highest valued coin.
In both cases, alice maximizes her score by choosing the stack that gives her the highest value.
By induction, the algorithm correctly determines the optimal strategy for alice to maximize her score.

5. *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.

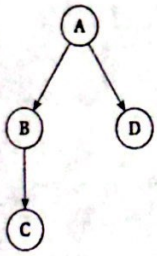

Figure 1: A hierarchy with four people. The fastest broadcast scheme is for A to call B in the first round. In the second round, A calls D and B calls C. If A were to call D first, then C could not learn the news until the third round.

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)

we define a minRounds(v) that calculates min. number of rounds.
- if "v" is a leaf node (has no direct subordinates), return 0.
- Otherwise, initialize a variable "minRound" to ∞.
- For each direct subordinate "u" of "v", calculate the number of rounds needed to notify everyone starting from "u" recursively.
- update "minRound" to the minimum value between "minRound" and "rounds".
- Return minRound.

minRounds(root) → ranking officer

(b) Give an efficient dynamic programming algorithm.

Function DP(v) where v is a node:
- create a memorization table "memo" to store the min. number of rounds for each node. Initialize all entries in "memo" to ∞ except for the leaf nodes, which are initialized to 0.
- Traverse the heirarchy tree in a bottom-up fashion, starting from leaf nodes and moving towards the root.
- For each node "v", calculate the number of rounds needed to notify everyone starting from "v" by considering each direct subordinate "u" of "v". The number of rounds needed is given by "rounds = 1+memo[u]."
update memo[v] to the minimum value among all "rounds" calculated in the previous step.
- Finally return memo[root], where root represents the ranking officer.

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

Above algorithm satisfies both optimal substructure and overlapping subproblems. optimal substructure: Algorithm recursively calculates the min. number of rounds needed for each subproblem. the min no. of rounds for each node is based on the min. no. of rounds needed by it's direct subordinates. The above algorithm guarantees that the solution obtained is the min. number of rounds required to notify everyone in the organization. Overlapping subproblems: By traversing the heirarchy tree in a bottom-up fashion, algo ensures the min no of rounds for each node is the only calculated one and stored in memo. when calculating min no. of rounds, the algo access the results of its direct subordinates directly from the memo. This eliminates redundant calculations and improves efficiency of the program.

6. 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)

Function countSubmatrices(M,i,j):
- If M[i][j] is 0, return 0 since it is not possible to form a square sub-matrix with all 1s ~~start~~ starting from this cell.
- If i or j is equal to 0, return 1 since each cell in the first row or first column itself forms a square sub-matrix with all 1s.
- Otherwise, calculate the total number of square sub-matrices with all 1s, including the current cell (i,j), by taking the min value among the following:
  1) countSubmatrices(M,i-1,j) {counting sub-matrices above current cell}
  2) " (M,i,j-1), (" " left of current cell)
  3) " (M, i-1,j-1), (" " diagonally above ~~and~~ to left of current cell).
- Add 1 to the min value obtained in step 3 to include the current cell itself
- Return the total count calculated in step 4.

(b) Give an efficient dynamic programming algorithm.

countSubmatrices(M)
- create a new matrix DP, of the same dimensions as M and initialize it with all zeroes.
- Initialize a variable totalCount to 0, to keep track of totalCount of square sub-matrices
- Iterate over each cell (i,j) in M:
  1) If M[i][j] is 0, set DP[i][j] to 0 & continue to the next cell.
  2) If i or j equal to 0, set DP[i][j] to 1.
  3) Otherwise set DP[i][j] to the min. value among DP[i-1][j], DP[i][j-1] & DP[i-1][j-1] plus 1.
  4) update totalCount by adding the value of DP[i][j]
- Return totalCount.

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

Algorithm satisfies optimal substructure & overlapping subproblems properties.
Optimal substructure: Algo calculates total number of subsquare matrices with all 1s for each subproblem and combines the results to obtain the solution for the overall problem. The total count for each cell is based on the min. value among the counts of its neighboring cells, considering the sub-matrices, the algorithm guarantees that the solution obtained is the total count of square sub-matrices with all 1s in the entire matrix.
overlapping subproblems: Dynamic programming algorithm uses a matrix DP to store & reuse the results of previously solved subproblems. By iterating over each cell in the matrix M in a systematic manner, the algorithm ensures that the count for each cell is only calculated once and stored in DP. When calculating the count for a cell, the algorithm accesses the results of its neighboring cells directly from 'DP'. This eliminates redundant calculations and improves the efficiency of the algorithm.
Therefore, this algorithm is correct & provides the total count of square submatrices with all 1s in the given matrix

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

To count the total number of square submatrices of M with all 0s, we can follow a similar approach using either the recursive or dynamic programming algorithm. The main difference is in the condition checks & initializations. In the recursive algorithm, if M[i][j] is 1, we return 0 since it isn't possible to form a square submatrix with all 0s starting from this cell. In the dynamic programming algorithm, if M[i][j] is 1, we set DP[i][j] to 0 instead of 1 in the initialization step. The rest of the algorithm remains the same & the total count obtained will represent the number of square sub-matrices with all 0s in the given matrix.

7. *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)

```
def is_interleaving_recursive(s, x, y):
    if len(s) == 0:
        return len(x) == 0 & len(y) == 0
    elif len(x) == 0:
        return s == y
    elif len(y) == 0
        return s == x

    else:
      if s[0] == x[0] & s[0] == y[0]:
        return is_inter..(s[1:], x[1:]y) or
               is_inter..(s[1:], x, y[1:])
      elif s[0] == x[0]:
          return is_inter...(s[1:], x[1:], y)
      elif s[0] == y[0]:
          return is_inter..(s[1:], x, y[1:])
      else
          return False
```

(b) Give an efficient dynamic programming algorithm.

```
def is_interleaving_dp(s, x, y):
    m, n = len(x), len(y)
    if len(s) != m+n:
        return false
    table = [[False] + (n+1) for _ in range(m+1)]
    table[0][0] = True
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 & j == 0
                continue
            + table[i][j] = (table[i-1][j] & x[i-1] == s[i+j-1]) or
                            (table[i][j-1] & y[j-1] == s[i+j-1])
    return table[m][n]
```

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

The dynamic programming algo handles base cases correctly when either x or y is empty. For the inductive step, assuming algo correctly determines whether s is an interleaving of x and y, where s is a substring of s and x and y are substrings of x and y respectively. By comparing last character of s with x & y, algo updates table[len(x)][len(y)] to indicate whether s is interleaving.
This algorithm guarantees correctness by considering all possible interleavings of x & y in the table.

8. *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.)

we first modify our shortest path algorithm. we modify the dichotomy from • use $\leq i-1$ edges to use i edges
and • use $\leq i$ edges to use $i-1$ edges.
we want to find v-w path. we have 2-D matrix of M. Number of edges × vertices.
$M(i,t)$ is shortest path from t to w using exact i edges $M(i,w) = 0$. we have an extra destination
$N(i,-6)$ is the number of path from t to w.
Bellman equation: $M(i,t) = \min_{s \in V} (M(i-1,s) + (t,s))$
$N(i,t) = \sum_{\text{satisfy}} N(i-1,s)$. Once we solve the matrix M, N, we have shortest path $v\text{-}w = \min_{i \leq |V|} M(i,v)$
The number of shortest path:
$= \sum_{i \text{ satisfy}} N(i,v)$

9. 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

$i = 1 \ldots 4 \quad w = 0 \ldots 6 \quad X_{i,w} = 1 \ (w_i \le w)$

$V(i,w) = \max \{ V(i-1,w),\ X_{i,w}(V(i-1, w-w_i) + v_i) \}$

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

maximum value is 19.

Trace back and we know

we take 4 (since $19 > 17$)

$3\ (15 > 5)$

$2\ (3 > 0)$

The subset is

2. 3. 4.