CS 577 # Assignment 7 – Dynamic Programming Fall 2022

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.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} > ℓ_i + ℓ_{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
```

Solution:
Counter-example:
$L := \langle 1, 1, 1 \rangle$
$H := \langle 1, 10, 100 \rangle$

For this instance, the algorithm JobSequence produces a schedule of: $\langle -, H, L \rangle$ with a value of 11, whereas the optimal schedule is $\langle L, -, H \rangle$ for a value of 101.

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

Solution:

- A $(n+2)$-element array s, where $s[i]$ contains the greatest possible profit over the first i weeks and the indices run from -1 to n.
- Bellman Equation:
$$s[i] = \max\{s[i-1] + \ell_i, s[i-2] + h_i\} \ ,$$
where $s[-1] = s[0] = 0$.
- The value of an optimal schedule for n weeks is found at $s[n]$. The schedule can be reconstructed by backtracing the decisions made at each max computation.

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

Solution: We prove that $s[n]$ is the profit of an optimal schedule (and that an optimal schedule can be reconstructed by backtracing) by strong induction over the weeks i.

Base case 1: $i = -1$ or 0. Nothing to schedule so optimal value is 0. The optimal schedule is the empty schedule.

Base case 2: $i = 1$. The possible schedules are either $(), (l_1)$, and (h_1). An optimal schedule is either (l_1) or (h_1), with the value of the optimal schedule being the maximum of h_1 and l_1. This agrees with the Bellman equation. Correctness of backtracing here is trivial; the choice of the job for week 1 determines the entire schedule.

Inductive Step: When scheduling week i, we can either schedule (1) a low stress job and combine it with the best schedule for the first $i-1$ weeks or (2) a high stress job combined with the best schedule for the first $i-2$ weeks. By the inductive hypothesis, we have that $s[i-1]$ and $s[i-2]$ are the values of optimal schedules for the first $i-1$ and $i-2$ weeks respectively. By correctness of $s[i-1]$ and $s[i-2]$, we have that the Bellman equation for $s[i]$ takes the max of the two possible options. Thus, the value of the optimal schedule for i weeks is found at $s[i]$.

Correctness of backtraced schedule follows from correctness of the backtraced schedules for $s[i-1]$ and $s[i-2]$. An optimal schedule will involve scheduling the low or high stress job, and copying an optimal solution for the first $i-1$ or $i-2$ weeks respectively. Since backtracing from $s[i-1]$ and $s[i-2]$ yields optimal schedules, the schedule constructed by backtracing from $s[i]$ is also optimal.

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.

Solution:
$M > 1$
$N := \langle 1, 3M, 1, 3M \rangle$
$S := \langle 3M, 1, 3M, 1 \rangle$
The optimal schedule is to start in NY and move between cities each month. The total cost of this schedule is $4 + 3M$. For any other schedule, you incur an operating cost of $3M$ for some month. If this other schedule involves no moves, the cost is $2 + 6M > 4 + 3M$. If the other schedule involves at least one move, then the cost is at least $3M + M + 3 = 4M + 3 > 4 + 3M$. In this case, a schedule with 3 moves is optimal.

(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
```

Solution:
Counter-example:
$NY := \langle 1, 2 \rangle$
$SF := \langle 2, 1 \rangle$
$M := 100$
The above algorithm will start in NY and then move to SF for month 2. The overall cost of this schedule is 102, whereas the optimal schedules are to stay in either NY or SF for both months, incurring an overall cost of 3.

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

Solution:

- A $2 \times n$-element matrix s, where $s[\{1,2\}][i]$ contains the optimal value over the first i months and being in NY for month i if the first coordinate is 1 and SF if the first coordinate is 2. The second index runs from 1 to n.
- $s[1][1] = N_1$ and $s[2][1] = S_1$.
- Bellman Equations for $i = 2$ to n,
  - For NY in month i:

$$s[1][i] = N_i + \min\{s[1][i-1], s[2][i-1] + M\}$$

  - For SF in month i:

$$s[2][i] = S_i + \min\{s[1][i-1] + M, s[2][i-1]\}$$

- The value of an optimal schedule for the n months is given by $\min_{j \in \{1,2\}} s[j][n]$.

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

Solution: We prove that $\min_{j \in \{1,2\}} s[j][n]$ gives the lowest possible cost by induction over the months i.

Base case: $i = 1$. Are two possible schedules are to start in NY or to start in SF. The cost when starting in NY is N_1 and the cost of starting in SF is S_1, which correspond to the definition of $s[1][1]$ and $s[2][1]$. The lowest possible cost corresponds to the minimum of these two values.

Inductive Step: WLOG consider the situation that an optimal schedule is in NY for month i. When scheduling month i in NY, we will have either spent the previous month in NY or in SF. If we spent the previous month in NY, our schedule will include the minimum cost schedule for the first $i-1$ months that ends in NY. The cost of our schedule for the first i months will be the cost of month i in NY plus the cost of our optimal $i-1$-month schedule that ends in NY (given by $s[1][i-1]$ by the inductive hypothesis). Similarly, if we spent the previous month in SF, our schedule will include the minimum cost schedule for the first $i-1$ months that ends in SF. The cost of our schedule for the first i months will be the cost of month i in NY, plus the cost to move from SF to NY, plus the cost of our optimal $i-1$-month schedule that ends in SF (given by $s[2][i-1]$ by the inductive hypothesis). The Bellman equation takes the minimum of the optimal costs for our two options, producing the optimal overall value for $s[1][i]$. An analogous argument holds for SF and $s[2][i]$.

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.)

Solution: Let $m(i, j)$ be the minimum cost starting from the ith month, given that j trucks were stored in the previous month. Our recursion will look at all possible values j' of trucks to store for the next month, and choose whichever gives the minimum answer. If $j < d_i + j'$, we incur the ordering fee of k, while if $j = d_i + j'$, there is no ordering fee. The case of $j > d_i + j'$ is not possible, as all unsold trucks must be stored.

$$m(i,j) = \min_{\max\{0, j-d_i\} \le j' \le s} \begin{cases} m(i+1, j') + cj' & \text{if } j = d_i + j' \\ m(i+1, j') + cj' + k & \text{if } j < d_i + j' \end{cases}$$

The full solution is given by $m(1, 0)$.

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

Solution:

- We work backwards from the last month, at each step deciding how many trucks to keep in storage for the next month, and using the lowest costs already calculated for future months.
- A 2D matrix m containing $(n \times (s+1))$-elements, where $m[i][j]$ contains the minimal cost accrued from months i through n given that we stored j trucks from month $i-1$ to i, and the indices are 1 to n for i and 0 to s for j.
- Initialize $m[n][j] = k + c \cdot j$ for all $j < d_n$, $m[n][j] = c \cdot j$ for $j = d_n$, and, in order to guarantee that in month n we do not have any extra trucks, $m[n][j] = \infty$ for all $j > d_n$.
- Define $f(i, j, j') = 0$ if $j \ge d_i + j'$ and 1 otherwise. That is, f is an indicator function, where i is the current month, j is the storage from month $i-1$ to i, and j' is the storage from month i to $i+1$. f is 1 if we need to order more trucks (if our current storage j is not enough to cover both the demand for next month, and the number of trucks we plan to store after next month).
- Bellman Equations for $i = n-1$ to 1 and $j = 0$ to s,

$$m[i][j] = c \cdot j + \min_{j' :\ \max\{0, j-d_i\} \le j' \le s} (k \cdot f(i, j, j') + m[i+1][j'])$$

- The minimum value for the n months is at $m[1][0]$, i.e., month 1 with no trucks stored from month 0 to 1.

The optimal cost is given by $m[1][0]$; the optimal schedule can be determined by backtracing.

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

Solution: We prove by reverse induction over the months i that $m[i][j]$ is the lowest possible cost accrued from months i to n given that j trucks are stored from month $i-1$ to month i. Correctness of the backtraced solution follows a similar argument.
Base case 1: $i = n$**.** The equation as defined above calculates the minimum cost for each $0 \leq j \leq s$.
Inductive Step: For each j for $m[i][j]$, by definition of the problem, the storage cost is $c \cdot j$. The minimizer portion of the Bellman equation considers all valid possibilities for the number of trucks j' to store for month $i+1$ given j. Notice that for a j larger than d_i, j' cannot be less than $j - d_i$. If j' is large enough to require an order, the cost k is included as determined by $f(i, j, j')$. The last part considered in the minimizer is $m[i+1][j']$ which is the minimal value for the parameters $i+1$ and j' from the induction hypothesis.
Running time: The matrix consists of $(n \cdot (s+1))$ cells and, for each cell, we consider $O(s)$ cells from the previous month. Overall, we have a runtime of $O(n \cdot s^2)$.

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.

Solution:
We define two 3D DP arrays: AliceOpt$[x][y][z]$ represents the maximum value of remaining coins Alice can get if it is currently Alice's turn, and there are x, y, z coins left in piles A, B, C, respectively. We define BobOpt$[x][y][z]$ similarly, with the only difference being that it is Bob's turn. (In particular, we want BobOpt to still count the value of Alice's coins.)

Bellman Equations: We set the base cases AliceOpt$[0][0][0]$ = BobOpt$[0][0][0] = 0$.

$$\text{AliceOpt}[x][y][z] = \max \begin{cases} a_x + \text{BobOpt}[x-1][y][z] & \text{if } x > 0 \\ b_y + \text{BobOpt}[x][y-1][z] & \text{if } y > 0 \\ c_z + \text{BobOpt}[x][y][z-1] & \text{if } z > 0 \end{cases}$$

$$\text{BobOpt}[x][y][z] = \min \begin{cases} \text{AliceOpt}[x-1][y][z] & \text{if } x > 0 \\ \text{AliceOpt}[x][y-1][z] & \text{if } y > 0 \\ \text{AliceOpt}[x][y][z-1] & \text{if } z > 0 \end{cases}$$

The optimal value for the problem is given by AliceOpt$[n][n][n]$. The algorithm uses $\Theta(n^3)$ time and space, which is polynomial in n.

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

Solution: We prove by strong induction on the triple x, y, z that AliceOpt$[x][y][z]$ correctly describes the maximum value of coins that Alice can gain, given it is Alice's turn, and x, y, z coins remain in piles A, B, C, respectively. We also prove the correctness of BobOpt.

Base Case: When there are 0 coins in each pile, Alice cannot gain any more value, regardless of whose turn it is. So AliceOpt$[0][0][0]$ and BobOpt$[0][0][0]$ are correct.

Inductive Step: We prove the correctness of AliceOpt$[x][y][z]$. Suppose it is Alice's turn and there are x, y, z coins in piles A, B, C. If Alice chooses to take a coin from pile A, she gains a_x value, and now it is Bob's turn and there are $x-1$, y, z coins in piles A, B, C. By the inductive hypothesis, she can gain a maximum of $a_x + \text{BobOpt}[x-1][y][z]$ value in total. We can use similar reasoning for the cases where she takes a coin from pile B or C. Since it is Alice's turn, the option that results in the maximum is chosen.

Now we prove the correctness of BobOpt$[x][y][z]$. If Bob takes a coin from pile A, Alice gains no value, and now it is Alice's turn and there are $x-1$, y, z coins in piles A, B, C. By our inductive hypothesis, Alice will gain AliceOpt$[x-1][y][z]$ value. We can once again use similar reasoning for the other piles. Since it is Bob's turn, the option that results in the minimum (for Alice's value) is chosen.

5. Implement the optimal algorithm for Weighted Interval Scheduling (for a definition of the problem, see the slides on Canvas) in either C, C++, C#, Java, Python, or Rust. Be efficient and implement it in $O(n^2)$ time, where n is the number of jobs. We saw this problem previously in HW3 Q2a, where we saw that there was no optimal greedy heuristic.

   The input will start with an positive integer, giving the number of instances that follow. For each instance, there will be a positive integer, giving the number of jobs. For each job, there will be a trio of positive integers i, j and k, where $i < j$, and i is the start time, j is the end time, and k is the weight.

   A sample input is the following:

```
2
1
1 4 5
3
1 2 1
3 4 2
2 6 4
```

   The sample input has two instances. The first instance has one job to schedule with a start time of 1, an end time of 4, and a weight of 5. The second instance has 3 jobs.

   The objective of the problem is to determine a schedule of non-overlapping intervals with maximum weight and to return this maximum weight. For each instance, your program should output the total weight of the intervals scheduled on a separate line. Each output line should be terminated by exactly one newline. The correct output to the sample input would be:

```
5
5
```

   or, written with more explicit whitespace,

```
"5\n5\n"
```

   Notes:

   - Endpoints are exclusive, so it is okay to include a job ending at time t and a job starting at time t in the same schedule.
   - In the third set of tests, some outputs will cause overflow on 32-bit signed integers.