CS 577 # Assignment 7 – Dynamic Programming Spring 2024

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.

   low stress = ℓ_i payout
   * high stress = h_i payout → only if no job in week $i-1$

   (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

h_i ___ h_{i+2}
ℓ_i ℓ_{i+1} ℓ_{i+1}

weeks	1	2	3	4	5
job payout	$h_1 = 1$	$h_2 = 5$	$h_3 = 1000$		
	$l_1 = 2$	$l_2 = 2$	$l_3 = 5$		

take $i = 2$ for instance, and that $h_3 < l_2 + l_3$, but $h_4 > l_4 + l_5$

Basically, if $l_i + l_{i+1}$ is chosen, the case where $h_{i+2} > l_{i+1} + l_{i+2}$ is not accounted for, and thus we are not gaurenteed to get the max payout, as it may be the case where $h_i + h_{i+2} > l_i + l_{i+1} + l_{i+2}$, which is not accounted for in this algorithm.

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

Problem definition: For a given week i, determine the max profit schedule between scheduling high stress (h_i) and low stress (ℓ_i) jobs, for n weeks

Base case: Dimensions = 1 Dimension Array

$S[0] = 0$

$S[1] = \max\{h_1, \ell_1\}$

Bellman Equation: $S[i] = \max \begin{cases} S[i-2] + h_i, & (\text{if } i > 1) \\ S[i-1] + \ell_i \end{cases}$

Order of population: 1 to n

Solution can be found at $S[n]$, as it will find the max profit schedule through backtracing.

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

We will prove the algorithm is correct by conducting strong induction on the number of weeks n

Base case 1: $n=0$. When there are 0 weeks to schedule, then no profit can be made, thus the optimal solution is 0, which our algorithm outputs.

Base case 2: $n=1$. When only 1 week to schedule, the algorithm will choose the max between h_1 and ℓ_1, as we want optimal profit.

Inductive step: when $n=k$, we want to find the max profit schedule for k weeks. We assume for the inductive hypothesis that the lookups $S[k-2]$ and $S[k-1]$ return the optimal profit schedules for $k-2$ and $k-1$ weeks respectively. Then, the current week's h_k pay or ℓ_k pay is added to each returned schedule. Afterwards, the Bellman Equation for $S[k]$ will choose the max profit schedule between $S[k-2]+h_k$ and $S[k-1]+\ell_k$. Thus, the maximum profit solution is chosen by our solution.

We also assume that $S[k-2]$ and $S[k-1]$ return correct values, so $S[k]$ must also return a correct answer.

either NY or SF

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.

$N = \{n_1, n_2 \dots n_n\}$, $S = \{s_1, s_2, \dots, s_n\}$ where $n_n, s_n = kM$, where $k \in \mathbb{Z}^+$ and can change for N and S depending on operating costs

Would want to move on week i if $s_i > M + n_i$ / $n_i > M + s_i$

Scenario:

$N = \{M, 10M, M, 10M\}$

$S = \{10M, M, 10M, M\}$

In this scenario, we see that at each alternating week, the operating costs heavily outweigh the moving cost + operating cost of the other city.

Thus in this scenario, moving 3+ times is optimal to minimize cost.

OPTIMAL SCHEDULE WOULD BE $M + M + M + M =$ 4M

(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

This algorithm doesn't correctly solve the problem because it doesn't account for the moving cost M at all. Consider a scenario where M is far larger than any operating costs in S or N

$N = \{5, 10\}$

$S = \{10, 5\}$

$M = 20$

-> here, since $n_2 > s_2$, the algorithm would move to S, however the moving cost is 20, thus actually the total cost $M + s_2 > n_2$, ∴ moving is not minimizing the cost, therefore the algo is incorrect.

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

Solution Matrix: $n \times 2$ matrix C where $C[(N,S),i]$ contains the minimum possible operating cost over i months, where if the first parameter N is New York, and the 2nd parameter S is San Francisco. Also, changing locations costs M. The problem runs from 1 to n, and is populated accordingly. (>i.e: $C[N,1] = n_1$, $C[S,1] = s_1$)

Bellman Equation:

For N in month i:

$$C[N,i] = n_i + \min\{C[N,i-1], C[S,i-1] + M\}$$

For S in month i:

$$C[S,i] = s_i + \min\{C[S,i-1], C[N,i-1] + M\}$$

Base Case: $S[k][1] = \max\{n_1, s_1\}$ where $k = N/S$

The value of the optimal solution can be found in $C[k,n]$ (where $k = N$ if $n_i \leq s_i$ or $k = S$ if $s_i < n_i$ — start with minimum cost)

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

Will prove that this algorithm minimizes the operating costs over n months using induction on n number of months.

Base Case: $n = 1$

When we only want to minimize costs for 1 month, then we don't need to move at all, and can just start at the area with the smaller operating costs, which our algo does, which is correct.

Inductive Step: $n = k$

We can either be in New York or SF on our $k-1^{st}$ month, thus assume that the operating costs are the minimum they can be for the $k-1^{st}$ month. If we are at NY on the $k-1^{st}$ month, for the k^{th} month the algorithm will calculate between staying at NY for the k^{th} month vs. moving cost of going to SF, and the operating costs at SF, and then return the minimum of the two, thus minimizing the total operating costs for the k^{th} month. The same idea applies flipped if we started at SF on our $k-1^{st}$ month. Thus, our Bellman equation minimizes our operating cost, as shown.

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

d_i = # of trucks expected to be sold on month i

i = month

s = unsold trucks to sell next month $(i+1)$

j = number of trucks stored / ordered

$c(i,j)$ = cost of storing j trucks on month i.

$k(i,j)$ = cost of ordering j trucks on month i.

Consider the following inventory planning problem. You are running a company that sells trucks and have good predictions for how many trucks you sell per month. Let d_i denote the number of trucks you expect to sell in month i. If you have unsold trucks any given month, you can store up to s of them in inventory to sell the next month instead. Storage cost is described by a function $c(i,j)$, where i is the month and j is the number of trucks stored. Every month you can buy any number of trucks, and each order has an associated ordering fee $k(i,j)$, that is a function of month i and number of trucks ordered j. Trucks ordered in month i can both be used to fulfill demand in month i, or be stored for future months. You start out with no trucks in storage. The problem is to design an algorithm that decides how many trucks to order each month to satisfy all the demands $\{d_i\}$, and minimize the total cost.

To summarize, every month you pay a storage cost $c(i,j)$, where i is the month and j is the number of trucks stored. Additionally, for each order you place you pay an ordering fee $k(i,j)$, where i is the month and j is the number of trucks ordered.You have to satisfy the demand for trucks d_i each month by either ordering trucks or having trucks in storage. In any month you can store at most s trucks.

(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.)

can store S trucks at most

C = cost of storage for j trucks at ith month

K = ordering fee of j trucks at ith month

d_i = sequence of trucks needing to be sold.

```
TruckCost(i, j): where i = month, j = # of trucks, returns minimum cost of storing j
                 trucks from month 1 to i.
Base case: i == 0:
    return 0  //no cost for 0 months
Recursive case:
  if j ≥ di:
    return c(i,j) + TruckCost(i-1, j)

  else:
    return c(i,j) + K(i, di-j) + TruckCost(i-1, j+(di-j))
```

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

Solution Matrix: A 2 dimensional matrix where we want to plan for i months of inventory, and we have j unsold trucks at the start of each month, and we can store s trucks at most, and storage of j trucks or ordering j trucks both have a respective cost of $c(i,j)$ and $K(i,j)$, and lastly an expected demand of trucks for the next i months, denoted by d_i

$DP[i,j]$ = minimum cost to satisfy demand of up to i months with j trucks in storage at the start of each month

Base case:

$DP[0,0] = 0$, since there's no cost for 0 months

Bellman Equation:

$$DP[i,j] = \min \begin{cases} DP[i-1, \max\{0, j-d_i\}] + K(i, d_i - j) & \text{(if } j < d_i\text{)} \\ DP[i-1, j-d_i+] + c(i, j-d_i) & \text{(if } j \geq d_i\text{)} \end{cases}$$

prev month

remaining $j-d_i$ trucks in storage, with max s trucks in storage

ordering fee of d_i trucks required

cost of storing $j-d_i$ trucks

Populate the sol. from 1 to n, since we backtrace.

Solution can be found at $DP[n][0]$, where n = number of total months, with 0 inventory at the start, since we are backtracing in our solution.

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

* We will prove through induction on the number of months n the correctness of our algorithm.

Base Case: n=0, thus 0 months of inventory so there's no cost, which our algorithm correctly returns.

Inductive step: Assume that the algorithm returns the correct result for month k=n-1, i.e: it returns correctly the minimum cost for the number of months k-1 and j number of remaining trucks.

Now, there are di number of trucks expected to be sold, so we check the value of j:

* if $j < di$, then more trucks expected to be sold then there are available, so we would order $di - j$ trucks, and also calculate the cost of $DP[i-1, 0/j-di]$, which we know through the inductive hypothesis is correct. Thus, this is the min possible cost for this scenario.

* if $j \geq di$, then we have more trucks stored than we do required to sell, thus we simply subtract the amount we want to sell and call DP again, which we know returns the correct value. Also, we add the storage cost of $j - di$ trucks remaining.

(> The bellman equation gives us the correct value in both possible scenarios, thus in our algorithm is correct.

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: 3 dimensional array, stacks A, B, and C, where matrix: the max element out of all 3 stacks (i,j, or k) will be chosen, where i,j,k shows remaining number of coins in each stack.

* AliceOpt[i,j,k] = Alice's choices, BobOpt[i,j,k], Bob's rational choices, for his turn.

$$\text{AliceOpt}[i,j,k] = \max \begin{cases} a_i + \text{BobOpt}[i-1,j,k] & \text{if } i>0 \\ b_j + \text{BobOpt}[i,j-1,k] & \text{if } j>0 \\ c_k + \text{BobOpt}[i,j,k-1] & \text{if } k>0 \end{cases}$$

$$\text{BobOpt}[i,j,k] = \min \begin{cases} a_i + \text{AliceOpt}[i-1,j,k] & \text{if } i>0 \\ b_j + \text{AliceOpt}[i,j-1,k] & \text{if } j>0 \\ c_k + \text{AliceOpt}[i,j,k-1] & \text{if } k>0 \end{cases}$$

Base case: AliceOpt[0,0,0] = 0 No coins to take

↳ Solution is located at A[n,n,n]

↳ uses $O(n^3)$ space

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

We will prove by strong induction that AliceOpt[i,j,k] correctly returns the max value of coins that Alice can obtain from the 3 stacks.

Base case: When there are 0 coins in every stack, Alice can't get any more coins, thus 0 is correctly returned.

Inductive Step: We will prove the correctness of AliceOpt[i,j,k], so assume it is Alice's turn and there are i,j,k coins in stacks A, B, and C. If Alice chooses to take from stack A, we will add a_i to BobOpt[i-1,j,k] as it is Bob's turn next, and he will choose rationally in this case. By the inductive hypothesis, BobOpt returns the optimal value, thus a_i + BobOpt[i-1,j,k] returns the optimal possible scenario when taking a_i out of stack A. The same applies to cases with stack B and C. Then, our algo chooses the max value returned from the 3 stacks + BobOpt's response, thus optimal.

BobOpt is correct cause its the same idea as AliceOpt but with the previous element in one stack out of 3.

5. **Coding Question: WIS**

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.