CS 577 # Assignment 7 – Dynamic Programming Fall 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: Riya Kore Wisc id: rykore

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: $L := \langle \ell_1, \ell_2, \dots, \ell_n \rangle$ and $H := \langle h_1, h_2, \dots, h_n \rangle$ 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
```

Solution:
Counter-example:
Let the sequence of low-stress jobs be: L $= <1, 1, 1>$
Let the sequence of high-stress jobs be: H $= <1, 5, 15>$

For this instance, the algorithm JOBSEQUENCE produces a schedule of $< -, H, L >$ with a value of 6, but the optimal schedule is $< L, -, H >$ for a value of 16.

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

Solution:
1) For an (n+2) - element array a, here a[i] contains the greatest possible profit over the first i weeks, and the indies (i) goes from -1 to n.
2) Bellman equation:

$$a[i] = max\{a[i-1] + l_i,\ a[i-2] + h_i\}$$

where a[-1] = a[0] = 0
3) The value of an optimal schedule for n weeks is found at a[n]. The schedule can be reconstructed by going back in steps for the decisions made while computing the max value from the Bellman Equation.

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

Solution:
We need to prove that a[n] is the profit of an optimal schedule (also that an optimal schedule can be reconstructed by going back on your steps) using strong induction over the weeks i.

Base case 1: i = -1 or 0. Nothing to schedule, so a[n] = 0. The optimal schedule is empty.

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. Backtracking becomes important because the choice of the job for week 1 determines the entire schedule.

Inductive Hypothesis: Let's assume that all the schedules till week i - 1 have optimal schedules, so the profit a[i - 1] is optimal.

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

To check the correctness of a[i] using backtracking, you have already achieved that from the correctness of the backtracking of a[i - 1] and a[i - 2]. An optimal schedule will involve scheduling the low or high stress job, and copying an optimal solution for the first i - 1 and i - 2 weeks respectively. Since backtracking from a[i - 1] and a[i - 2] gives optimal schedules, the schedule constructed by backtracking from a[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

For New York: N = $< 1, 3M, 1, 3M >$
For San Francisco: S = $< 3M, 1, 3M, 1 >$

The optimal schedule will be to start in New York 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 from some month. If this other schedule involves no moves, 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:

For New York: $< 1, 2 >$
For San Francisco: $< 2, 1 >$
M = 100

The above algorithm will start in New York and then move to San Francisco for month 2. The overall cost of this schedule is 102, whereas the optimal schedules are to stay in either New York or San Francisco 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:
1) A 2xn - element matrix a, where a[{1,2}][i] contains the optimal value over the first i months. Being in New York for month i if the first co-ordinate is 1 and San Francisco if the first co-ordinate is 2. The second index runs from 1 to n.
2) Base cases will be: a[1][1] = N_1 and a[2][1] = S_1
3) Using the Bellman equation for i = 2 to n,
- For New York in month i:

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

For San Francisco in month i:

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

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

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

Solution:
We need to prove that the minimum value of a[j][n], where j ∈ 1,2, gives the lowest possible cost by doing induction over the months i.

Base Case: i = 1. There are two possible schedules in this case, one where you start in New York and the other where you start in San Francisco. The cost when starting in New York is N_1 and the cost when starting in San Francisco is S_1. This corresponds to the definition of a[1][1] and a[2][1]. The lowest possible cost corresponds to the minimum of these two values.

Inductive Hypothesis: Let's assume that all the schedules for New York and San Francisco till i - 1 months are optimal schedules, so a[1][i-1] and a[2][i-1] are both the lowest possible costs.

Inductive Step: Without loss of generality (WLOG), lets consider the situation that an optimal schedule is in New York for month i. When scheduling the month i in New York, we will have either spent the previous months in New York or in San Francisco. If we spent the previous month in New York, our schedule will include the minimum cost schedule for the first i - 1 months that end in New York. Using the information from inductive hypothesis, the cost of out schedule for the first i months will be the cost of month i in New York plus the cost of our optimal i - 1 month schedule that ends in New York.
If we spent the previous months in San Francisco, our schedule for the i months will include our optimal cost schedule for the first i - 1 months that ends in San Francisco (given by the inductive hypothesis) plus the cost to move from San Francisco to New York plus the cost of month i in New York. The Bellman equation takes the minimum of the optimal costs of our two options, producing the optimal overall value for a[1][i]. The same argument hold for San Francisco and a[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 a(i,j) be the minimum cost starting from the ith month, given that j trucks were stored in the previous month. The recursive algorithm will look at all possible values $j^{'}$ of trucks to store for the next month, and choose whichever gives the minimum answer. If j $< d_j + j^{'}$, we incur the ordering fee of k, while if j $= d_j + j^{'}$, there is no ordering fee. The case of j $> d_j + j^{'}$ is not possible, because all the unsold trucks must be stored.

$$a(i,j) = min_{max\{0,j-d_j\} \leq j' \leq a} \begin{cases} a(i+1,j^{'}) + cj^{'} & \text{if j} = d_i + j^{'} \\ a(i+1,j^{'}) + cj^{'} + k & \text{if j} < d_i + j^{'} \end{cases}$$

a(1,0) gives the full solution.

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

Solution: 1) We backtrack from the last month, and at each step we decide how many trucks to keep in storage for the next month, and using the lowest costs already calculated for the future months.
2) We use a 2D grid called 'a' to keep track of the minimum cost for each combination of months and stored trucks. Each cell 'a[i][j]' tells us the minimum cost from month 'i' to the end of our planning n if we keep 'j' trucks in storage from month i - 1 to i. The matrix contains (n x (s + 1)) elements and the indices go from 1 to n for i and 0 to s for j.
3) Initialize a[n][j] = k + c*j for all j $< d_n$, a[n][j] = c*j for j = d_n, and, to guarantee that in month n we do not have any extra trucks, a[n][j] = ∞ for all j $> d_n$.
4) Define f(i,j,j') = 0 if j is >= di + j' and 1 otherwise. f is an indicator function, 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).
5) Bellman equation for i = n - 1 to 1 and j = 0 to s,

$$a[i][j] = c * j + min_{j':max\{0,j-di\} \leq j' \leq s}(k * f(i,j,j') + a[i+1][j'])$$

6) The minimum value for the n months is at a[1][0], i.e. month 1 with no trucks stored from month 0 to 1.
The optimal cost is given by a[1][0] and the optimal schedule can be determined by going back on your steps.

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

Solution:
We can use reverse induction over the months i such that a[i][j] is the lowest possible cost from months i to n given that j trucks are stored from month i - 1 to month i. Correctness of the backtracked solution follows a similar argument.

Base case: i = n. The equation as defined above calculates the minimum cost for each $0 \leq j \leq s$.

Inductive Hypothesis: Let's assume that the above argument holds true for months i.

Inductive Step: For each j in a[i][j], by the problem's definition, the storage cost is c*j. Finding the minimum in Bellman's equation is valid for all possibilities for the number of trucks j' to store for month i + 1 given j.
For j > dj, j' cannot be less than j - dj. If j' is large enough to require an order, the cost k i included as determined by f(i, j ,j'). The part considered in the minimizer is a[i+1][j'] (for the i+1 months). This is the minimal value for the parameters i + 1 and j' from the inductive hypothesis.

Running time: The matrix consists pf (n * (s+1)) cells and, for each of the cells, we consider O(s) cells for the previous month. Overall, we have a runtime of O($n * s^2$)

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 $a_1, \dots, a_n$. Similarly, the coins in stack B have values $b_1, \dots, b_n$, and the coins in stack C have values $c_1, \dots, c_n$. Both players try to play optimally in order to maximize the total value of their coins.

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

Solution:
Lets define two 3D dynamic programming arrays:
AliceOpt[x][y][z] represents the maximum value of remaining coins Aline can get if it is currently Alice's turn, and there are x, y, z coins left in piles A, B, C, respectively.
Similarly, we have BobOpt[x][y][z] with the only difference being that it is B0b's turn. (In particular, we want Bob to still count the value of Alice's coins.)
Bellman equations for the arrays: We set both cases AliceOpt[0][0][0] = BobOpt[0][0][0] = 0.

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

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

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 this by using strong induction on the triple x,y,z such 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 can also prove the correctness of BobOpt in the same way.

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 both correct.

Inductive Hypothesis: Let's assume that Alice gains a value of BobOpt[x-1][y][z] in her turn and vice versa for Bob.

Inductive step: Suppose its Alice's turn and there are x, y, z coins in the piles A, B, C. If Alice chooses to take a coin from pile A, she gains a_x in value, and now it is Bob's turn and there are x - 1, y, z coins in piles A, B, C. Using inductive hypothesis, she can gain a max of $a_x + BobOpt[x-1][y][z]$ value in total. We can use a similar method for the cases where she takes a coin from pile B or c. Since it is Alice's turn, the options 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 at all, 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 use similar reasoning for all the other plans. Since it is Bob's turn, the option that results in the minimum (for Alice's value) is chosen.

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.