CS3000: Algorithms & Data — Summer I '21 — Drew van der Poel

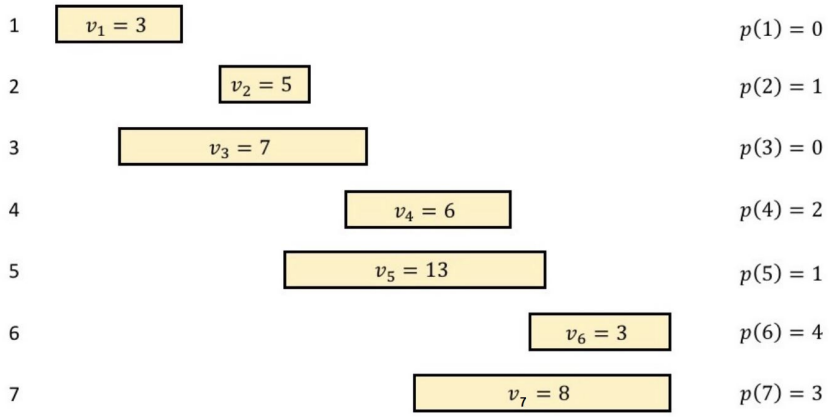
Homework 3
Due Sunday, June 6 at 11:59pm via Gradescope

Name:
Collaborators:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- This assignment is due Sunday, June 6 at 11:59pm via Gradescope. No late assignments will be accepted. Make sure to submit something before the deadline.
- Solutions must be typeset. If you need to draw any diagrams, you may draw them by hand as long as they are embedded in the PDF. I recommend using the source file for this assignment to get started.
- I encourage you to work with your classmates on the homework problems. *If you do collaborate, you must write all solutions by yourself, in your own words.* Do not submit anything you cannot explain. Please list all your collaborators in your solution for each problem by filling in the `yourcollaborators` command.
- Finding solutions to homework problems on the web, or by asking students not enrolled in the class is strictly forbidden.

Problem 1. *Interval Scheduling Recap (10 points)*

This problem will test your understanding of dynamic programming by having you run through the algorithm for interval scheduling that we saw in class. Consider the following input for the interval scheduling problem:



Fill out the dynamic programming table with the values of $\text{OPT}(i)$ for $i = 0, 1, \ldots, 7$. Label each value of $\text{OPT}(i)$ with whether or not the optimal schedule of length i contains interval i or not. Write the optimal schedule and its value.

Solution:

Recall that $\text{OPT}(i)$ is the *value* of the optimal schedule O_i for the first i intervals. The base case is $\text{OPT}(0) = 0$.

i	0	1	2	3	4	5	6	7
$\text{OPT}(i)$	0	3	8	8	14	16	17	17
O_i includes i?	–	yes	yes	no	yes	yes	yes	no

The optimal schedule is $\{1, 2, 4, 6\}$ with a value of 17. We can find the optimal schedule by unwinding the information in the dynamic programming table as

$$\begin{aligned} O_7 &= O_6 \\ &= \{6\} + O_4 \\ &= \{4, 6\} + O_2 \\ &= \{2, 4, 6\} + O_1 \\ &= \{1, 2, 4, 6\} \end{aligned}$$

Problem 2. *Boston Sports Pass (30 points)*

You have won a "Boston Sports Pass" as part of a student raffle! The pass works as follows, you get a free ticket to either one Bruins or one Celtics game each of the n weekends of the season. However, there is a catch. Every time you want to switch from one to the other, you have to pay a *switch fee*. So if you get Celtics tickets the first three weekends, and then Bruins tickets the remaining weekends, you would pay the fee once. If you switched every weekend, you would pay the fee $n-1$ times. You can choose to start with whichever team you like without paying any fee.

Unfortunately, because you are too busy studying, you determine you can't go to any games. Fortunately, you can turn this into a nice chunk of change, as your sports-afficionado Algorithms instructor offers to buy the tickets off of you. He tells you how much he would pay you for each game, but he isn't willing to pay the switch fee. Determine the maximum amount of profit you can make.

To formalize, the input to the problem is $2n+1$ numbers, $C_1,\ldots,C_n,B_1,\ldots,B_n,S>0$ where C_i is the amount you'd receive for choosing the Celtics game on weekend on i, B_i is the amount you'd receive for choosing the Bruins game on weekend i, and S is the switch fee. You will find the optimal set of weekends to select Celtics tickets.

As an example, if $S=15$, and the potential amounts received for $n=5$ weeks are

Weekend (i)	1	2	3	4	5
Celtics (C_i)	25	54	37	19	40
Bruins (B_i)	30	33	70	35	30

then the optimal schedule is [Celtics, Celtics, Bruins, Bruins, Bruins] with a net profit of $25+54+70-15+35+30=199$ where the -15 comes from paying the switch fee during the third weekend.

(a) **[10 points]** For $i=1,2,\ldots,n$, let $OPT_C(i)$ be the maximum net profit of any schedule for weekends $1,\ldots,n$ such that you take the Celtics ticket on weekend i. Likewise, let $OPT_B(i)$ be the maximum net profit of any schedule for weekends $1,\ldots,n$ such that you take the Bruins ticket on weekend i. Give recurences for computing the values $OPT_C(i)$ and $OPT_B(i)$. Provide short justification for the corectness of your recurrences. Don't forget your base cases!

Solution:

$OPT_C(1)=C_1$

$OPT_C(i)=\max(OPT_C(i-1),OPT_B(i-1)-S)+C_i$ where $1<i\leq n$

$OPT_B(1)=B_1$

$OPT_B(i)=\max(OPT_B(i-1),OPT_C(i-1)-S)+B_i$ where $1<i\leq n$

For the base case of $i=1$, we have to select the Celtics/Bruins ticket on the first weekend, so our profit will be C_1 or B_1, accordingly. In the general case, when computing $OPT_C(i)$, we know we have to select the Celtics ticket on weekend i and receive C_i from doing so. Thus, our only decision is whether it is optimal to select the Celtics or Bruins ticket on weekend $i-1$. Selecting the Celtics ticket would give a value of $OPT_C(i-1)$, while selecting the Bruins ticket would give $OPT_B(i-1)$, minus S, the switch fee. We follow the same logic for computing $OPT_B(i)$.

(b) **[10 points]** Describe in pseudocode a dynamic programming algorithm that computes the values $OPT_C(i)$ and $OPT_B(i)$ and determines the maximum net profit achievable by any schedule. Your algorithm may be either "bottom-up" or "top-down".

Solution:

Algorithm 1: Boston Sports Pass $(C_1, \ldots, C_n), (B_1, \ldots, B_n), S$

$T_B \leftarrow n \times 1$ array
$T_C \leftarrow n \times 1$ array
$T_B[1] \leftarrow B_1$
$T_C[1] \leftarrow C_1$
For $i = 2, \ldots, n$
  $T_B[i] = max(T_B[i-1], T_C[i-1] - S) + B_i$
  $T_C[i] = max(T_C[i-1], T_B[i-1] - S) + C_i$
Return $max(T_B[n], T_C[n])$

(c) **[5 points]** State the asymptotic running time of your algorithm, and briefly justify your answer. State the asymptotic space complexity (including the DP table(s) and inputs) and briefly justify your answer.

Solution:

The algorithm runs in $O(n)$ time. There are $n-1$ iterations of the for loop, each of which take $O(1)$ time. The initialization and return steps are also $O(n)$, thus the total runtime is $O(n)$.

The space complexity is also $O(n)$, as we have two n-dimensional arrays as DP tables and our inputs are two n-dimensional arrays and a single value (S). Thus, the total number of items stored would be $4n+1$, which is $O(n)$.

(d) **[5 points]** Show via pseudocode how you could use the values $OPT_C(i)$ and $OPT_B(i)$ to compute the schedule with the maximum total revenue. You can use your filled-in DP table(s) from part b. Describe your approach.

Solution:

We would first compare $T_B[n]$ and $T_C[n]$. If $T_B[n] \geq T_C[n]$, then we know we will select the Bruins ticket on the n-th weekend. In this case we would compare $T_B[n-1]$ and $T_C[n-1] - S$ to determine which ticket to select on the $n-1$-th weekend. If $T_C[n] > T_B[n]$, then we know we will select the Celtics ticket on the n-th weekend, and again use the recurrence from part (a) to determine which ticket to select on the $n-1$-th weekend.

We will repeat this process of looking at the recurrence for the team we select on weekend i

to determine which ticket to select on weekend $i-1$ until we have completed the schedule.

Algorithm 2: Compute Schedule

▷ We use B_i to represent selecting the Bruin's ticket on weekend i, and likewise for C_i and the Celtics

Function FINDSCHEDULE(S, T_B, T_C, X, i)**:**

- **If** $i \leq 0$ **:**
  - **Return** $\emptyset$
- **If** $X == B$ **:**
  - ▷ we select the Bruins on weekend i
  - **If** $T_B[i-1] \geq T_C[i-1] - S$ **:**
    - **Return** $B_i \cup$ FINDSCHEDULE($S, T_B, T_C, B, i-1$)
  - **Else**
    - **Return** $B_i \cup$ FINDSCHEDULE($S, T_B, T_C, C, i-1$)
- **Else**
  - ▷ we select the Celtics on weekend i
  - **If** $T_C[i-1] \geq T_B[i-1] - S$ **:**
    - **Return** $C_i \cup$ FINDSCHEDULE($S, T_B, T_C, C, i-1$)
  - **Else**
    - **Return** $C_i \cup$ FINDSCHEDULE($S, T_B, T_C, B, i-1$)

If $T_B[n] \geq T_C[n]$ **:**

- **If** $T_B[n-1] \geq T_C[n-1] - S$ **:**
  - **Return** $B_n \cup$ FINDSCHEDULE($S, T_B, T_C, B, n-1$)
- **Else**
  - **Return** $B_n \cup$ FINDSCHEDULE($S, T_B, T_C, C, n-1$)

Else

- **If** $T_C[n-1] \geq T_B[n-1] - S$ **:**
  - **Return** $C_n \cup$ FINDSCHEDULE($S, T_B, T_C, C, n-1$)
- **Else**
  - **Return** $C_n \cup$ FINDSCHEDULE($S, T_B, T_C, B, n-1$)

Problem 3. *Armageddon (24 points)*

The NASA Near Earth Object Program lists potential future Earth impact events that the JPL Sentry System has detected based on currently available observations. Sentry is a highly automated collision monitoring system that continually scans the most current asteroid catalog for possibilities of future impact with Earth over the next 100 years.

This system allows us to predict that i years from now, there will be x_i tons of asteroid material that has near-Earth trajectories. In the mean time, we can build a space laser that can blast asteroids. However, each laser blast will require exajoules of energy, and so there will need to be a recharge period on the order of years between each use of the laser. The longer the recharge period, the stronger the blast—after j years of charging, the laser will have enough power to obliterate d_j tons of asteroid material. You must find the best way to use the laser.

The input to the algorithm consists of the vectors $(x_1, \ldots, x_n)$ and $(d_1, \ldots, d_n)$ representing the incoming asteroid material in years 1 to n, and the power of the laser d_i if it charges for i years. The output consists of the optimal schedule for firing the laser to obliterate the most material.

Example: Suppose $(x_1, x_2, x_3, x_4) = (1, 10, 10, 1)$ and $(d_1, d_2, d_3, d_4) = (1, 2, 4, 8)$. The best solution is to fire the laser at times $3, 4$. This solution blasts a total of 5 tons of asteroids.

(a) Construct an input on which the following "greedy" algorithm returns the wrong answer:

Algorithm 3: The BADLASER Algorithm

Function BADLASER$(x_1, \ldots, x_n), (d_1, \ldots, d_n)$**:**
- Compute the smallest j such that $d_j \geq x_n$, or set $j = n$ if no such j exists
- Shoot the laser at time n
- **If** $n > j$ **:**
  - **Return** BADLASER$(x_1, \ldots, x_{n-j}), (d_1, \ldots, d_{n-j})$

Intuitively, the algorithm figures out how many years j are needed to blast all the material in the last time slot. It shoots during that last time slot, and then accounts for the j years required to recharge for that last slot, and recursively considers the best solution for the smaller problem of size $n - j$.

Solution:

Consider an instance where $(x_1, x_2, x_3, x_4) = (1, 2, 100, 2)$ and $(d_1, d_2, d_3, d_4) = (1, 2, 100, 101)$. BADLASER$(1, 2, 100, 2), (1, 2, 100, 101)$ would shoot at time 4 and blast $x_4 = 2$ tons of asteroids. Since the smallest number of years j such that $d_j \geq 2$ is $j = 2$, the recursive call would be BADLASER$(1, 2), (1, 2)$. The laser would choose to shoot again at time 2 and blast another $x_2 = 2$ tons of asteroids. The smallest number of years j such that $d_j \geq 2$ is $j = 2$ so the algorithm would terminate. Overall BADLASER would manage to destroy 4 tons of asteroids. However, the optimal solution is to wait 3 years and shoot at time 3 blasting 100 tons of asteroids, and then again at time 4 blasting 2 tons of asteroids. Therefore, BADLASER is not the right algorithm for this problem.

(b) Let OPT(j) be the maximum amount of asteroid we can blast from year 1 to year j. Give a recurrence to compute OPT(j) from OPT$(1), \ldots,$ OPT$(j-1)$. Give a few sentences justifying why your recurrence is correct.

Solution:

We have to decide how many years we will wait and charge in order to shoot at year j. If we wait $i = 1, \ldots, j$ years, we will have enough power to obliterate $d_1, \ldots, d_j$ tons, respectively. No matter how many years we wait, the maximum number of tons we could obliterate in year j is x_j. Therefore, if we decide to wait i years and shoot at j, we will obliterate $\min\{x_j, d_i\}$ tons that year. Since we would need to charge for i years, the most recent past year we could have shot again is $j - i$ and the maximum number of asteroid tons we can obliterate from year 1 to year $j - i$ is $\text{OPT}(j - i)$. This way, we would blast a total of $\min\{x_j, d_i\} + \text{OPT}(j - i)$ tons from year 1 to year j. To decide how many years i we will wait, we take the option that maximizes the total tons of asteroids we blast. Therefore, the recurrence is:

$$\text{OPT}(j) = \max_{1 \le i \le j} \{\min\{x_j, d_i\} + \text{OPT}(j - i)\}$$

$$\text{OPT}(0) = 0$$

(c) Using your recurrence, design a dynamic programming algorithm to output the optimal set of times to fire the laser. You may use either a top-down or bottom-up approach. Remember that your algorithm needs to output the optimal set of times to fire the laser.

Solution:

The (bottom-up) algorithm for the problem is the following:

Algorithm 4: The GOODLASER Algorithm $(x_1, \ldots, x_n), (d_1, \ldots, d_n)$

```
M ← array of zeros
M[0] ← 0
Function FINDOPT(n):
    For j = 1, ..., n
        M[j] ← 0
        For i = 1, ..., j                          ▷ finds max_{1≤i≤j}
            If M[j] < min{x_j, d_i} + M[j − i] :
                M[j] ← min{x_j, d_i} + M[j − i]
    Return M[n]

Function FINDSHOOTINGTIMES(M, n):
    If n = 0 :
        Return ∅
    Else
        For i = 1, ..., n
            If M[n] = min{x_n, d_i} + OPT(n − i) :
                Return {n} + FINDSHOOTINGTIMES(M, n − i)
```

We use FINDOPT to find the maximum tons of asteroids we can blast from year 1 to year n (i.e. $\text{OPT}(n)$) and store it in $M[n]$. We use FINDSHOOTINGTIMES to find the times we need to fire the laser. If $M[n] = \min\{x_n, d_i\} + \text{OPT}(n - i)$ for some i, then this means that the optimal

solution for years 1 to n shoots at time n and charges for i years. Therefore, we add shooting time n and recurse to find the rest of the shooting times included in the optimal solution, the latest of which could be in year $n - i$.

(d) Analyze the running time and space usage of your algorithm.

Solution:

- Time: In FINDOPT, each iteration of the nested loop takes constant time. Therefore, the running time of FINDOPT is asymptotically bounded by the total number of iterations, which is $\sum_{j=1}^{n} j = \Theta(n^2)$. Each call of the FINDSHOOTINGTIMES(M, j) function makes at most j iterations and a recursive call. The total number of recursive calls is at most n. Thus, the running time of FINDSHOOTINGTIMES is $O(\sum_{j=1}^{n} j) = O(n^2)$. Overall the algorithm has $\Theta(n^2)$ running time.

- Space: The algorithm stores the $n + 1$ values of array M. Hence, the space usage of the algorithm is $\Theta(n)$.

Problem 4. *Strategery (40 points)*

Alice and Bob play the following game. There is a row of n tiles with values $a_1, \ldots, a_n$ written on them. Starting with Alice, Alice and Bob take turns removing either the first or last tile in the row and placing it in their pile until there are no tiles remaining. For example, if Alice takes tile 1, Bob can take either tile 2 or tile n on the next turn. At the end of the game, each player receives a number of points equal to the sum of the values of their tiles minus that of the other player's tiles. Specifically, if Alice takes tiles $A \subseteq \{1, \ldots, n\}$ and Bob takes tiles $B = \{1, \ldots, n\} \setminus A$, then their scores are

$$\sum_{i \in A} a_i - \sum_{i \in B} a_i \quad \text{and} \quad \sum_{i \in B} a_i - \sum_{i \in A} a_i,$$

respectively. For example, if $n = 3$ and the tiles have numbers $10, 2, 8$ then taking the first tile guarantees Alice a score of at least $10 + 2 - 8 = 4$, whereas taking the last tile would only guarantee Alice a score of at least $8 + 2 - 10 = 0$.

In this question, you will design an algorithm to determine the maximum score that Alice can guarantee for herself, assuming Bob plays optimally to maximize his score. Note that the sum of their scores is always 0, so if Bob is playing optimally to maximize his own score, then he is also playing optimally to minimize Alice's score.

(a) (**8 points**) Describe the set of subproblems that your dynamic programming algorithm will consider. Your solution should look something like "For every ..., we define $\text{OPT}(\ldots)$ to be ..."

Solution: For every i and $j \geq i$ denoting the first and last index of a sequence of tiles $a_i, \ldots, a_j$, we define $\text{OPT}(i, j)$ to be the maximum score that the first player can achieve given this sequence. Thus, each of our subproblems is defined by a subsequence of $a_1, \ldots, a_n$.

(b) Give a recurrence expressing the solution to each subproblem in terms of the solution to smaller subproblems.

Solution:

Let's assume we have a sequence of tiles $a_i, \ldots, a_j$ and that it's Alice's turn. She has two options:

- **Case 1:** She can pick the first tile a_i and place it in her pile. She immediately increases her score by a_i. Then the other player, Bob, is left with the instance $a_{i+1}, \ldots, a_j$. We have assumed that both players play optimally so the score that Bob will achieve from this sequence is the maximum possible, that is, $\text{OPT}(i + 1, j)$. Therefore Alice will get $-\text{OPT}(i + 1, j)$ since the sum of their scores is always 0. So, with this move, Alice will score $a_i - \text{OPT}(i + 1, j)$ overall.
- **Case 2:** She can pick the last tile a_j and place it in her pile. In this case, she immediately increases her score by a_j and leaves Bob with the sequence $a_i, \ldots, a_{j-1}$. His score on this sequence will be $\text{OPT}(i, j - 1)$ and Alice's will be $-\text{OPT}(i, j - 1)$. So, with this move, Alice will score $a_j - \text{OPT}(i, j - 1)$ overall.

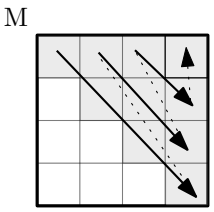
Notice that Alice and Bob are interchangeable in the previous argument: Bob has the same options if it is his turn to play. The goal of the player is to make the choice that will maximize their score. Therefore, the recurrence is the following:

$$\text{OPT}(i,j) = \max\{a_i - \text{OPT}(i+1,j), a_j - \text{OPT}(i,j-1)\} \text{ , if } i > j$$
$$\text{OPT}(i,i) = a_i$$

(c) Explain in English a valid order to fill your dynamic programming table in a "bottom-up" implementation of the recurrence.

Solution:

We need to fill an $n \times n$ table M, where each element is $M(i,j) = \text{OPT}(i,j)$, in order to find $M(1,n)$ which is the maximum score Alice can achieve with the original sequence $a_1, \ldots, a_n$. We only need to fill in the upper triangle of the matrix, since $M[i,j]$ does not make sense whenever $i > j$. First, we fill in the diagonal of the matrix which is $M[i,i] = a_i \ \forall i$. To fill in the value of $M[i,j]$ we need to know the values of $M[i+1,j]$ and $M[i,j-1]$. Therefore, we fill the cells $M[i,i+1]$ next since we know the values $M[i,i]$ and $M[i+1,i+1]$ on which they depend. We continue filling in the table this way until we reach the $M[1,n]$ cell. The order we just described is shown in the following image.



(d) Describe in pseudocode an algorithm that finds the maximum score that Alice can guarantee for herself. Your implementation may be either "bottom-up" or "top-down."

Solution:

The "bottom-up" algorithm for the problem is the following:

Algorithm 5: Strategery $(a_1, \ldots, a_n)$

```
M ← matrix n × n of zeros
For k = 0, ..., n − 1
    For i = 1, ..., n − k
        If k = 0 :
            M[i, i] ← a_i
        Else
            j = i + k
            M[i, j] ← max{a_i − M[i + 1, j], a_j − M[i, j − 1]}
Return M[1, n]
```

(e) Analyze the running time and space usage of your algorithm.

Solution:

- Time: Our algorithm takes constant time in each iteration of the nested loop. The number of iterations is $\sum_{k=0}^{n-1}(n-k) = \sum_{k=1}^{n} k = \Theta(n^2)$. Another way to see this is that it takes constant time to fill each of the $\frac{n(n+1)}{2}$ cells of the upper triangle of M, thus having $\Theta(n^2)$ running time overall.
- Space: The size of M is $n \times n$ so the space usage of our algorithm is $\Theta(n^2)$.