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

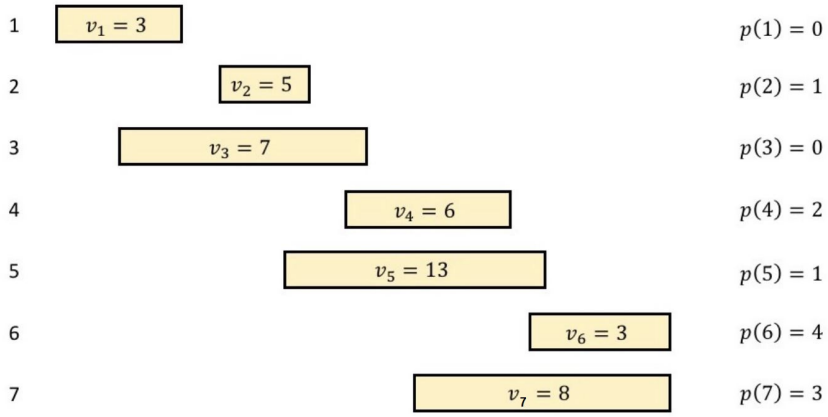
Homework 3
Due Sunday, June 6 at 11:59pm via Gradescope

Name: Gabriel Peter
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:
TABLE CREATION (-1 = Empty)

$M = [0, 5, 7, 11, 13, 14, 15]$

SCHEDULE OPTIMIZATION:
$FindSched(7) \rightarrow [7, 3]$
$FindSched(6) \rightarrow [6, 4, 2, 1]$
$FindSched(5) \rightarrow [5, 1]$
$FindSched(4) \rightarrow [4, 2, 1]$
$FindSched(3) \rightarrow [3]$
$FindSched(2) \rightarrow [2, 1]$
$FindSched(1) \rightarrow [1]$
$FindSched(0) \rightarrow []$
All $Opt(i)$ contain i in the schedule!
$OptimalSchedule = [7, 3]$ with a value of $Opt(7) = 15$

Problem 2. *Boston Sports Pass (24 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) **[8 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:

$OPT_C(0)=0$
$OPT_C(1)=C_1$
$OPT_C(i)=C_i+max(OPT_C(i-1),OPT_B(i-1)-S)$

We can assert the correctness of this recurrence by noting the two cases for it:
Either, CASE 1: $OPT_C(i-1)$ is not part of the solution and $OPT_B(i-1)$.
Or, CASE 2: $OPT_C(i-1)$ is part of the solution and $OPT_B(i-1)$ is not.
There is no possibility for both $OPT_C(i-1)$ and $OPT_B(i-1)$ to be part of the same solution.
Similarly there is not such thing as an empty set solution due to the nature of the problem.

$OPT_B(0)=0$
$OPT_B(1)=B_1$

$OPT_B(i) = B_i + max(OPT_B(i-1), OPT_C(i-1) - S)$

We can assert the correctness of this recurrence by noting the two cases for it:
Either, CASE 1: $OPT_B(i-1)$ is not part of the solution and $OPT_C(i-1)$.
Or, CASE 2: $OPT_B(i-1)$ is part of the solution and $OPT_C(i-1)$ is not.
There is no possibility for both $OPT_B(i-1)$ and $OPT_C(i-1)$ to be part of the same solution.
Similarly there is not such thing as an empty set solution due to the nature of the problem.

Additionally, I analyze the solution at $(i-1)$ instead of i because the problem's unique nuance that 'You can choose to start with whichever team without paying any fee.' Which means we receive the first $week_i$ for free.
This also matches my pseudo-code implementation which was optionally tested.

(b) **[8 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:

```
def OPT_C(n):
    if M_C[n] != None:
        return M_C[n]
    else:
        M_C[n] = C[n] + max(OPT_C(n-1), OPT_B(n-1)-S)
        return M_C[n]

def OPT_B(n):
    if M_B[n] != None:
        return M_B[n]
    else:
        M_B[n] =  B[n] + max(OPT_B(n-1), OPT_C(n-1)-S)
        return M_B[n]
```

For this problem, I decided to implemnent the soltuion in a '$top-down$' approach. Based off the recurences written above, I incorporated them into functions in order to populate their respect M_B M_C DP tables which calculate the maximum soltuion if you initially start with B or C for game 'n'. In a funny way, solutions for $M_C[i]$ can be compromised from M_B values or visa-versa such that we know the value of revenue we receive by choosing the repsective team on $week_i$ regardless of how many switches are made later on for $week_{i-1,2,3,...}$ as figuring out WHEN to switch is done in FindSched().

$M_C = [0, 25, 79, 116, 138, 194]$
$M_B = [0, 30, 63, 134, 169, 199]$

The soltuion schedule is then constructed from the DP table by taking the $max(M_B[n], M_C[n])$. Depending on which is bigger, this signifies that you would choose that respective team for week. This is the seed for the initial call of FindSchedule in the next question in order to determine which team to start off with for free.

(c) **[4 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 asymptotic space complexity of inputs is $O(2n+1) = O(n)$
The space complexity is $O(2n) = O(n)$ each M_C and M_B had n elements corresponding to the availble weeks.
The running time complexity for recurences: $OPT_C/B = O(n^2)$ because the only perform calculations for $2n$ slots in the DP tables, while the number of calls technically is more than $O(n)$, they are hitting the line if $M_C[n]! = None : returnM_C[n]$ which is an $O(1)$ operation.

THUS, the runtime complexity for the entire algorithm is also $O(n)$ as the FindSchedule Function simply iterates down by $(n-1)$ resulting in n loops.
$O(n^2) + O(n) = O(n^2)$

(d) **[4 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:

```
def FindOpt(M_C, M_B, n, current_team):
    if n == 0:
        return []
    celtics_choice = M_C[n]
    bruins_choice = M_B[n]
    if current_team == 'Celtics':
        bruins_choice = M_B[n] - S
    else:
        celtics_choice = M_C[n] - S
    if celtics_choice > bruins_choice:
        return FindOpt(M_C, M_B, n-1, 'Celtics') + ['Celtics']
    else:
        return FindOpt(M_C, M_B, n-1, 'Bruins') + ['Bruins']
```

Continuing with the $top-down$ approach, the FindSched(...) now considered the current team and the cost for switching teams. If $M_switchTeam[n] - S > M_currentTeam[n]$ we know that by switching teams, we would profit more than not-switching and avoiding the 'S' fee. This recurrence is made for the entire DP table until the base cases are hit.

Based on the provided DP tables from OPT_C and OPT_B, we are provded with the following:
$['Celtics','Celtics','Bruins','Bruins','Bruins']$

Problem 3. *Armageddon (32 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 asteroid material present in years 1 to n, and the power of the laser d_i if it charges for i continuous 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) **[4 points]** Construct an input on which the following "greedy" algorithm returns the wrong answer:

Algorithm 1: 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:

x = [2, 2, 2, 5]
d = [2, 3, 4, 5]
This would yeild the wrong results as the algorithm is obligated to shoot each asteroid that it physically has the charge for. Meanwhile the correct solution would wait until the last asteroid to shoot it for maximum tons destroyed.
Thus, it require all 4 years to blast, $x_4 = 5$ as it is greedy to pick the first. However, the lowest power values are weighted higher (proportionally) so the best solution is to fire at i=1,2,3,4 and the value will be 8 tons!

(b) **[8 points]** Let $\text{OPT}(j)$ be the maximum amount of asteroid we can blast from year 1 to year j. Give a recurrence to compute $\text{OPT}(j)$ from $\text{OPT}(1),\ldots,\text{OPT}(j-1)$. Give a few sentences justifying why your recurrence is correct.

Solution:
$n == j$
$OPT(0) = 0$
$OPT(1) = min(x_1, d_1)$
$OPT(2) = max(min(x_2, d_2), min(x_2, d_1) + OPT(1)$
$OPT(n) = max_{2<x<n}(min(x_n, d_x) + OPT(n-x), OPT(n-1))$

The blaster can either shoot at n, and iterates at n-x or shoots at n-1. Those are the only two options for the laser; therefore, the maximum value of asteroid debris destroyed is a summation of those two options until n == 0 or n-j == 0. + Base cases.

(c) **[8 points]** 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:

```
n = 4
x = [0, 1, 10, 10, 1]
d = [0, 1, 2, 4, 8]
M = [0, min(d[1],x[1])] + [None] * (n-1)

# This function build the dp table getting
# the max asteroid mass that can destroyed
# in 'n' years
def OPT(n):
    for i in range(2,n+1):
        # subproblems = []
        best = 0
        for j in range(1, i+1):
            y = min(x[i], d[j]) + M[i-j]
            if y >= best:
                best = y
        M[i] = best

OPT(n)
print(M) # [0, 1, 2, 4, 5]

# This function retrieves the schedule
# to achieve the max asteroid destruction within
# 'n' years
```

```
def FindSched(M, n):
    if n == 0: return ''
    else:
        for j in range(1, n+1):
            if M[n] - min(x[n], d[j]) == M[n-j]:
                return FindSched(M, n-j) + f'{n} '
        return FindSched(M, n-1)

print(FindSched(M, n)) # 3 4
```

I used a $'bottom-up'$ approach to solve this problem. The first function, OPT(n), creates the memoization table, which interates 2,...,n (avoiding the two base cases). Within each iteration a nested for loop finds the maximum amount of asteroid possible after a given amount of charge. This is a combination of the previous dp entries and/or simply a full charge of j years. This allows for all cases of charging to occur in order to find the optimal soltuion.

Secondly, the FindSched(M, n) function utilizes the DP to create the schedule for the optimal solution. This is done by iterating through all the possible charge amounts at year 'n' then sees which previous charge would have been possible with the maximum charge. For example, at n = 4, the only value of min(x[n], d[j]) that has a corresponding lower value in the dp table is when J=1, which means it only needed a single year to charge and can calculate at n=n-j=3 with a max of 4 tons of asteroid according to the DP table.. The proccess terminates here because there is a perfect value of j to where min(x[n], d[j]) = M[n], providing us with the proper schedule.

(d) **[4 points]** Analyze the running time and space usage of your algorithm.

Solution:

OPT(n) and FindSched(M, n) run consecutively; therefore their runtime complexities are additive. For OPT function we have two nested for loops at a max iteration of 'n' therefore it has an $O(n^2)$.

Meanwhile FindSched(M, n) also has a worst-runtime of $O(n^2)$ as it checks each $d_1, ...d_n$ and there is a max of 'n' * FindSched(M, n-1) calls.

This leaves us a $O(n^2) + O(n^2) \rightarrow O(n^2)$
Meanwhile, the space complexity is only $O(n)$ as the DP has a length of 'n' elements from an input of $O(2n+1) \rightarrow O(n)$

(e) **[8 points]** In order to better your familiarity with dynamic programming algorithms, we have created a hackerrank challenge (www.hackerrank.com/cs3000-summer1-2021-programming-assignment-3). Please implement your dynamic programming strategy and submit it to the challenge. Your grade for this part will depend on (a) how many test cases your implementation passes and (b) actually implementing a dynamic programming strategy (we will check!).

In order to allow for efficient grading, please write your hackerrank account below.

Solution:
peter_g

Problem 4. *Strategery (30 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) **[6 points]** Describe the set of subproblems that your dynamic programming algorithm will consider. Your solution should look something like "For every ..., we define OPT(...) to be ..."

Solution:
For every turn of the game, we define OPT(n) to be the maximum value Alice can achieve from an n-piece board, which results in a complete win of the game. This "maximum value" accounts for her opponents sum of tile values being subtracted from her's.

(b) **[8 points]** Give a recurrence expressing the solution to each subproblem in terms of the solution to smaller subproblems.

Solution:

$OPT(0) = 0$
$OPT(1) = a_1$
$OPT(2) = max(a_1 - a_2, a_2 - a_1)$

For every turn of the game, $OPT(n) = max(a.pop(1) - OPT(n-1), a.pop(n) - OPT(n-1))$

This recurrence will yeild the maximum score Alice can receive, which is the sum of her tiles - the sum of Bob's tiles. We can assert this is true because there are only two options availble for a player's turn: a_1 or a_n so one of them must be part of the solution. Subsequentially, Bob will choose his most obtimized move and that tile's value is subtracted from her's. The next turn will have Alice pick either a_1 or a_n for her max score. Luckily the math works our such that:
Alice's - (Bob's - Alice's) == Alice's + Alice's + ... - (Bob's - ....)

Based on the function flow. pop() asserts that the board will update with the removed pieces, but the actually pseudocode benifited from simply keeping track of left and right bounds. Bob's logic and goal is identical to Alice's which allows for this minimal implementation.

(c) **[4 points]** Explain in English a valid order to fill your dynamic programming table in a "bottom-up" implementation of the recurrence.

Solution:

In order to fill the DP in a bottom-up fashion you will need to start with the last two pieces of the game, which would be in the center surrounding or equal to the midpoint pivot. Then for each consecutive iteration of i, M[i] will be the maximum value of either of the next/next first or last tiles. To which the inner nested for-loop would simulate bob's turn as he is then going to pick the next best value for a maximum game. The nuances are similar to the full, top-down implementation below...
The recurrence stops well all tiles are picked! and M[n] is the value of Alice's score.

(d) **[8 points]** 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:

```
board = [10, 2, 8, 5]
n = len(board)
M = [0] + [None] * (n+1)
M_B = [0, ]

def OPT(l, r, n):
    if n <= 0: return 0
    if n == 1: return board[l]
    if n == 2:
        return max(board[l]-board[r], board[r]-board[l])
    if M[n] != None: return M[n]
    else:
        M[n] = max(
            board[l] - OPT(l+1, r, n-1),
            board[r] - OPT(l, r-1, n-1))
        return M[n]


OPT(0, n-1, n)
print(M)
```

For this problem I implemented a $top - down$ approach, which properly calculates the maximum score Alice can make while Bob also manages to optimize his solution. Essentially a win with this strategy is guaranteed at the beginning by who starts the match. The 'else' statement is the meat of the algorightm as alice has to choose the right-bound tile or the left-bound tile, knowing that bob's turn will choose the maximum total of the new resulting board size of n-1. Bob's choices are then $OPT(l+1, r, n-1)$ or $OPT(l, r-2, n-1)$ respectively. n is used to keep track of the end game and solve the base cases.

The variables: l, r are needed in order to assert the boundaries of the board without mutating the object directly.

(e) **[4 points]** Analyze the running time and space usage of your algorithm.

Solution:

The space complexity of this soltuion is $O(n)$ as M, the DP memoization table, have n slots corresponding to the maximum score the protagonist can receive with the n-tiles left in the game.

The runtime complexity is $O(n^2)$ as for each turn (iteration of n), results in a n iterations from the nested for-loop (noted in bottom-up), however, topDown will have a similar runtime.