Timur Garifullin
Due: 11:59pm Friday, Nov 4

Homework 9

Fall 2022, Bhakta
CS 315, Algorithms

Consulting: Artur, Dr. Bhakta
Source: GeekForGeeks
Time Spent: 24 hours

This assignment covers more advanced dynamic programming and recovering the solution.

Directions: For all of the dynamic programming questions, we want *all* the following information:

- State how your subproblems are defined, make sure to state what the indexes mean.
- Describe the main recursive idea of your solution, with an explanation of your reasoning. State what the base cases are, and justify your recurrence.
- Write pseudocode that will solve the problem in polynomial time *and extract the solution* if needed. State any assumptions that you are making about the given data structures.
- State the final running time of your algorithm, with some explanation. You will get more partial credit for a correct but slower solution with the proper analysis than for an incorrect, allegedly faster solution.

1. (20 points) You are in charge of a social student organization, and you are trying to figure out which days to throw parties for your members. Throwing parties makes you happy, but takes away your energy and money.

   You start the semester fully rested with E energy, a party budget of D dollars for the semester, and 0 happiness. There is one potential party you can throw on each of n days in the semester.

   If you throw the party on day i, your happiness will go up by $h_i \geq 0$, but your energy will go down by $e_i \geq 0$ and it will cost $d_i \geq 0$ dollars.

   If you don't throw the party on day i, your happiness and money will not change, and your energy level will recover by a fixed r per rest day.

   Neither your energy levels or your dollars are allowed to go below 0. Your energy levels also cannot go above E with additional resting.

   You are given as input E, D, r, n and the length-n arrays $HGain = [h_i]$, $ECost = [e_i]$, and $DCost = [d_i]$.

   Your goal is to maximize total happiness gained from throwing parties across the semester, provided you don't run out of money and your energy doesn't go below 0.

   Design a dynamic programming solution to this problem.

   (a) (5 points) Clearly define the meaning of your subproblems. Your subproblems should be a smaller version of the above "big" problem, and often captures the idea of an intermediate state.

   Hint: Adapt a similar reasoning behind the knapsack subproblem, but to a 3 dimensional subproblem.

   Solution: Let Hapiness[i][j][b] be the max value of happiness of throwing a party up to the day i, with remaining energy $j >= 0$, and budget spent $b >= 0$.

(b) (10 points) What is the recursive solution? It helps to think about the various cases/choices available to you. Feel free to use shorthand if describing minimums/maximums/sums/subsets. Give justification for your recursive solution. Don't forget to include the base cases!

Hint: Because 3D reasoning is hard, don't try to work out a small subproblem for intuition or patterns. Instead, think abstractly. What are your choices?

Solution:
Case 1 (Not throwing a party):
if j == E:
consider $Happiness[i-1][j-x][b]$ where x = [0, r) and (j-x > 0)
else:
consider $Happiness[i-1][j-r][b]$ where (j-r >= 0)

Case 2 (Throwing a party):
if $j + e_j <= E$ and $b + d_j <= D$:
$Happiness[i][j][b] = Happiness[i-1][j+e_i][b+d_i] + h_j)$

$Happiness[i][j][b] = max(Case1, Case2)$

Recursive Idea:

The solution maximizes happiness while managing the remaining energy and budget spent, considering whether to throw a party or not.There two cases to consider:

Case 1 (Not throwing a party):
If the remaining hours j exactly equal the total available hours E, it means there are several ways we could have go here. Therefore, we consider the maximum happiness obtained from the previous $i-1$ days with x subtracted from remaining energy j. x iterates from 0 inclduing to r excluding. The budget is the amount of money spent b.
Otherwise, you consider the maximum happiness from the previous $i-1$ days with the energy set to remaining energy $j-r$, and budget b.
Case 2 (Throwing a party):
If throwing a party is feasible, we consider the respective constraints and add the happiness h_i gained from the party on day i activity to the maximum happiness obtained from the previous $i-1$ days with adjusted energy $j+e_i$ and budget $b+d_i$.

Finally, we take the maximum value between the happiness obtained from not throwing a party (Case 1) and the happiness obtained from throwing a party (Case 2).

Base cases:
$Happiness[0][E][D] = 0$
$Happiness[i][E-j_0][D-b_0] = h_0$
$Happiness[i][j][b] = \infty$

(c) (5 points) You don't need to implement anything. What is the final runtime *if* this were

implemented as an efficient DP instead of with recursion?

> **Solution:** We have a 3D table with dimensions $n * (E + 1) * (D + 1)$. Each cell in this table involves constant-time operations leading to a total of $O(1)$ operations per cell. Also, since some days will be for energy recovery we get $E + r$. Thus, the overall time complexity which accounts for all possible combinations of days, budgets, energy levels, and energy recovery is $O(nD(E + r))$.

2. (25 points) You and Dr. Bhakta are playing a two player game. There are n cards, face up in front of you in a row, and each card has a positive integer labeled on it. You take turns either taking a card from either the beginning or end of the list of cards.

   For example, if you were looking at the cards [222315], you would have to option of picking up either the last 5, leaving [22231] or the first 2, leaving [22315]. Then it would be Dr. Bhakta's turn to choose a card, then yours, etc.

   Play ends when all cards are gone. Your goal is to get the highest sum possible for the cards you take. Note that this is a zero sum game - all the cards eventually are shared between you and your opponent, so maximizing your score is equivalent to minimizing your opponent's score.

   (a) (5 points) First, show that the greedy algorithm of always choosing the larger of the two cards available to you fails to be optimal on some small counterexample. Note that this means that you need to give a situation where a greedy sequence of moves against a perfect player yields a worse result than some other sequence of moves against a perfect player.

   > **Solution:** if I move first and the cards are = [4, 9, 3, 2]
   >
   > Greedy:
   > I pick 4, Dr. Bhakta picks 9, I pick 3, Dr. Bhakta picks 2;
   > Outcome: $7 < 11$, Dr. Bhakta wins
   >
   > Optimal:
   > I pick 2, Dr. Bhakta picks 4, I pick 9, Dr. Bhakta picks 3;
   > Outcome: $7 < 11$, I win

   Now we'll design a dynamic programming algorithm to help us play the game.

   We'll use two subproblems, that we'll solve simultaneously.

   Let MYTURN[i,j] be the maximum amount that the *Player* can win when faced with the set of cards from $i...j-1$ *and it's currently the player's turn.*

   and

   Let NOTMYTURN[i,j] be the maximum amount that the *Player* can win when faced with the set of cards from $i...j-1$ and *it's currently Dr. Bhakta's turn.*

   (b) (10 points) What is the recursive relationship between the subproblems? The two subproblems may depend on each other. Briefly explain your reasoning. Make sure to specify the base cases. As with many dynamic programming questions, thinking about the cases/choices available to you for your problem may help.

Solution: The recursive relationship between MYTURN[i,j] and NOTMYTURN[i,j]:

MYTURN[i,j] = the maximum amount that the *Player* can win when faced with the set of cards from $i...j-1$ *and it's currently the player's turn.*

- If it's the player's turn, they have two choices: either pick the card at index i or pick the card at index j-1.
- If they pick the card at index i, the amount they win is the value of card[i] plus the maximum amount they can win from the remaining cards NOTMYTURN[i+1, j].
- If they pick the card at index j-1, the amount they win is the value of card[j-1] plus the maximum amount they can win from the remaining cards NOTMYTURN[i, j-1].
- The player's goal is to maximize their winnings, so the recursive relationship is:
  MYTURN[i, j] =
  max(card[i] + NOTMYTURN[i+1, j], card[j-1] + NOTMYTURN[i, j-1])

NOTMYTURN[i,j] = maximum amount the player can win when it's Dr. Bhakta's turn and the cards are from index i to j-1:

- If it's Dr. Bhakta's turn, he also has two choices: either pick the card at index i or pick the card at index j-1.
- If Dr. Bhakta picks the card at index i, the amount he wins is the negative value of card[i] (since it's subtracted from the player's winnings) plus the maximum amount he can win from the remaining cards MYTURN[i+1, j].
- If Dr. Bhakta picks the card at index j-1, the amount he wins is the negative value of card[j-1] plus the maximum amount he can win from the remaining cards MYTURN[i, j-1].
- Dr. Bhakta's goal is to minimize the player's winnings, so the recursive relationship is:
  NOTMYTURN[i, j] =
  min(-card[i] + MYTURN[i+1, j], -card[j-1] + MYTURN[i, j-1])

Base Cases:

- When there are no cards left ($i >= j$), MYTURN[i, j] and NOTMYTURN[i, j] are both 0 since there are no cards to pick.
- When there is only one card left ($i + 1 == j$), MYTURN[i, j] is simply the value of that card, and NOTMYTURN[i, j] is the negative value of that card (since it's Dr. Bhakta's turn).

(c) (10 points) Write pseudocode or code (code is recommended) to solve this problem efficiently. What is the final runtime?

Solution:
Assuming the Player is the one to pick first.

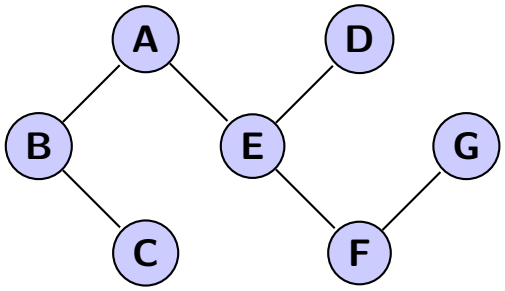
```
def CardGame(cards):
    n = len(cards)
    dpTable = [[0] * n for _ in range(n)]

    for length in range(1, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if length == 1:
                dpTable[i][j] = cards[i]
            elif length == 2:
                dpTable[i][j] = max(cards[i], cards[j])
            else:
                dpTable[i][j] = max(
                    cards[i] + min(dpTable[i + 2][j], dpTable[i + 1][j - 1]),
                    cards[j] + min(dpTable[i + 1][j - 1], dpTable[i][j - 2])
                )

    return dpTable[0][n - 1]
```

Runtime Complexity:
The final runtime of the card game is $O(n^2)$. n is the number of cards in the input list. This runtime comes from the nested loops used to fill in the table called "dpTable". Within each iteration of the outer loop (length), there are constant-time operations performed to update the table, giving us an overall $O(n^2)$ runtime complexity.

3. (15 points) A *vertex cover* of a graph $G = (V, E)$ is a subset of the vertices $S \subset V$ that includes at least one endpoint of every edge in E. Give a linear-time dynamic programming algorithm for the following task.

   Input: An undirected tree $T = (V, E)$.
   Output: The **size** of the smallest vertex cover of T.

   For instance, in the following tree, possible vertex covers include $\{A, B, C, D, E, F, G\}$ and $\{A, C, D, F\}$ but not $\{C, E, F\}$. The smallest vertex cover has size 3: $\{B, E, G\}$.



   (a) (5 points) State how your subproblems are defined, make sure to state what the indexes mean.

Hint: Adapt the solution for independent set done in class. Optionally work in some of the ideas from the previous question's subproblems for an extra easy time.

Solution: Subproblem:
Let's define SmallestVertexCover[v][0] as the minimum size of a vertex cover of the subtree rooted at node v, where v is not included in the vertex cover.
Let's define SmallestVertexCover[v][1] as the minimum size of a vertex cover of the subtree rooted at node v, where v is included in the vertex cover.
Indexes:
v: is the current node in the subtree traversal. 0 and 1: if the current node is not in the vertex cover then 0, if the current node is included then 1.

(b) (10 points) What is the recursive relationship between the subproblems? What should you return?

Solution: For each non-leaf node v, we can update the SmallestVertexCover[v][0] and SmallestVertexCover[v][1] based on its children nodes u.

- SmallestVertexCover[v][0] can be updated as the sum of the min vertex cover sizes of all children nodes where the children nodes are included in the vertex cover ($\sum_{u \in children(v)}$SmallestVertexCover[v][1])
- SmallestVertexCover[v][1] can be updated as 1 + the sum of the min vertex cover sizes of all children nodes
  ($1+\sum_{u \in children(v)}$min(SmallestVertexCover[$u$][0],SmallestVertexCover[u][1])

The algorithm would return the minimum of the SmallestVertexCover[*root*][0] and SmallestVertexCover[*root*][1]. Here *root* is the root node of the tree. This will give us the smallest vertex cover for the entire tree.

4. (40 points) There was a terrible accident at registration, and it turns out that you were enrolled in an algorithms class all semester and you didn't know it! What a nightmare! It's the end of the semester now, and the registrar won't let you withdraw because they don't believe your story. Luckily you know some algorithms already, and Dr. Bhakta will give you a chance to get as many points as possible and hopefully pass the class.

   There were n homeworks assigned over the semester, and each is worth 100 points. Because you can't hope to finish all the homeworks completely, you have to try to maximize your partial credit on each homework in the time you have to spend. You have h hours total to spend among these n homeworks.

   You are given an $nx(h+1)$ array *points*, where *points*[i][j] is the amount of partial credit that you will get on homework i if your spend exactly j hours on that homework.

   You are given that *points*[i][0] = 0 (0 time = 0 points), and that *points*[i][j] <= *points*[i][$j+1$] (more time on an assignment doesn't decrease points).

   For example, for $n = 3, h = 6$, you might be given the below input:

```
points = [ [0, 50, 75,  90,  100, 100, 100],
           [0, 80, 100, 100, 100, 100, 100],
```

```
[0, 40, 70,  80,  90,  95,  100] ]
```

Your goal is to allocate your available hours to these assignments in order to maximize your total points. The total amount of time spent across all assignments should be at most h.

(a) (5 points) One greedy algorithm would be to start with allocating 0 hours to each assignment. Then, for h iterations, add one hour to the assignment that will give you the largest *increase* in your total points.

For example, if $h = 6$, the greedy algorithm would allocate, in order,

- 1 hour to assignment 1, for 80 points
- 1 hour to assignment 0, for 50 points
- 1 hour to assignment 2 for 40 points
- 1 hour to assignment 2 for an additional 30 points
- 1 hour to assignment 0 for an additional 25 points
- 1 hour to assignment 1, for an additional 20 points

This happens to be the optimal solution for $h = 6$ for this input to the problem. However, this doesn't always lead to an optimal allocation.

Find an example input (points array, h) where the greedy algorithm fails to find the optimal solution.

Solution: Greedy algorithm fails to find the optimal solution with this input:

points array =

$$\begin{bmatrix}[0, & 11, & 17], \\ [0, & 6, & 91]\end{bmatrix}$$

h = 2

Greedy Algorithm:
1 hour to assignment 0, for 11 points
1 hour to assignment 0, for 6 points
Total points = 17 points

Optimal:
1 hour to assignment 1, for 6 points
1 hour to assignment 1, for 85 points
Total points = 91 points

Now, we will solve the problem using dynamic programming.

(b) (5 points) State your subproblem definition for this dynamic program. Be sure to specify the range of indices for this subproblem.

Solution: Let MaxPoints $[i][j]$ represent the maximum points that can be obtained by spending j hours on the first i homework assignments, $1 <= i <= n$ and $0 <= j <= h$

(c) (10 points) What is the recursive relationship between the subproblems? Explain your reasoning. Make sure to specify the base cases. As with many dynamic programming questions, thinking about the cases/choices for your problem may help.

Solution:
Recursive relationship:
MaxPoints[i][j] = max (MaxPoints[i-1][j-k] + points[i][k])
where k varies from 0 to j, representing the hours spent on the ith homework assignment.

Reasoning:

- To maximize the points for the ith homework assignment, we need to consider all possible hours k spent on that assignment.
- The maximum points obtained for the ith assignment and j total hours can be achieved by choosing the maximum of:
  The maximum points obtained for the first i - 1 assignments and j - k total hours, plus the points obtained for spending k hours on the ith assignment (points[i][k]).
  This accounts for the fact that we can spend any number of hours k on the ith assignment, and we want to maximize the total points considering all possible choices.

Base cases:

- MaxPoints[0][j] = 0 for all j, since there are no homework assignments to consider.
- MaxPoints[i][0] = 0 for all i, since there is no more time to spend on homework assignments. Spending 0 hours on homework gives 0 points.

(d) (15 points) Write WORKING code to solve this problem. You may write your program in Java, C++, or Python.

You should write your code so that it is a self contained function that takes an array/vector/list as input, as well as an integer k. The function should return the total cost as described above *as well as* return a list containing exactly how much time you should spend on each assignment. This will require backtracking through your computed subproblems when the main loop has ended.

You should also include tests, so that when I compile/run your file, I will see the execution of your function on all your tests. I will provide some tests shortly.

Note: when grading, I will add my own secret tests on large arrays, and your code should finish within a few seconds.

(e) (5 points) State the big-O final running time and space usage of the algorithm. You may assume that integer addition and multiplication is happening in constant time. You will get more partial credit for a correct but slower solution with proper analysis than for an incorrect, "allegedly" faster solution. This should be a polynomial in n, h.

Hint: I am expecting a runtime larger than $\Theta(nh)$.

Solution: Big O time complexity:

- The algorithm iterates through every homework assignment and each possible time allocation up to h.
- For each homework i and each time allocation j, the algorithm computes the maximum points by iterating through time spent k, from 0 to j.
- Thus, the time complexity is $O(n * h^2)$. n total homework assignments, h total hours available.

Space usage:

- The algorithm uses the "dpTable" table of size (n + 1)*(h + 1) to store the maximum points for each subproblem.
- It also uses the choices table of the same size to track the time allocation for each subproblem.
- Thus, the space usage is $O(n * h)$.

5. (Bonus) This bonus problem is combined between homeworks 8 and 9.

   Create an account on open.kattis.com, a programming contest archive. Show me that you've submitted working code for any problem that you solve using dynamic programming, (10 points each) up to 50 bonus points total. Examples: (You don't have to pick these, there are so, so many):

   - https://open.kattis.com/problems/bachetsgame
   - https://open.kattis.com/problems/goodcoalition
   - https://open.kattis.com/problems/purplerain
   - https://open.kattis.com/problems/spiderman
   - https://open.kattis.com/problems/everythingisanail
   - https://open.kattis.com/problems/trainsorting
   - https://open.kattis.com/problems/bagoftiles
   - https://open.kattis.com/problems/princeandprincess
   - https://open.kattis.com/problems/coke

   Submit your code for any of these problems as well as a screen shot showing that you passed all the tests on kattis. I'm putting this here so you can start early if you want - some of these are tricky.