Timur Garifullin
Due: 11:59pm Thursday, Mar 28

Homework 8

Spring 2024, Bhakta
CS 315, Algorithms

This assignment covers Dynamic Programming.

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

- State how your subproblems are defined, explicitly state what the indices are and what their range is.
- 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 / code that will solve the problem in polynomial time. You MUST submit working code when requested, and you may and are encouraged to submit working code always.
- 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.

0. (0 points) (NOT OPTIONAL) Who did you work with on this homework? What sources did you consult? How long did this assignment take?

> **Solution:** Consulting: Artur Idrissov, Dr. Bhakta
> Sources: Class Notes, GeekForGeeks
> Time Spent: 21 hours

1. (Example) Find the length of the longest increasing subsequence in an array A of length n.
   (a) Define the subproblems in the recursion, make sure to state what the subproblems *mean* at a given index/indices.

   > **Solution:** Let $LIS(i)$ be the length of the longest increasing subsequence in A that ends at index i of A, for $0 \leq i < len(A)$.

   (b) What is the recursive relationship between the subproblems? What are the base cases? What should you return?

   > **Solution:** Consider a longest increasing subsequence that ends at i. This subsequence is either length 1, or it has some index $j < i$ as the element before i. It must be that $A[j] < A[i]$ because the sequence is increasing. The total length of this increasing subsequence that ends at element i is the length of this longest increasing subsequence that ends at element j plus 1. Out of all possible previous elements j, we are looking for the index j that maximizes the length of the longest increasing subsequence to i - so we choose the index j that maximizes the result. Therefore,
   >
   > $$LIS(i) = \max_{j<i : A[j]<A[i]} LIS(j) + 1$$

Here, if we take the max of an empty set to be 0, then we cover the case where there are no viable "previous" elements of i.

The longest increasing subsequence in the entire array must end somewhere in the array (not necessarily at $n-1$), so we will return the largest value of $LIS(i)$ over all i.

(c) Write pseudocode to solve this problem. You may submit actual working code as well.

Solution:

```
def LIS(A):
#Solutions array
Solutions = [0] * (len(A))
#hardcode base case
Solutions[0] = 1
#loop over the others
for i in range(1, len(A)):
    mx = 0
    for j in range(i): #all previous elements < i
        if A[j] < A[i]:
            mx = max(Solutions[j], mx)
    Solutions[i] = mx + 1
return max(Solutions)
```

(d) State the big-O final running time and space usage of the algorithm with an explanation of your calculation. You will get more partial credit for a correct but slower solution with proper analysis than for an incorrect, "allegedly" faster solution.

Solution: We have $O(n)$ subproblems and do $O(n)$ work per subproblem, so the total runtime is $O(n^2)$. We allocate space for the $O(n)$ subproblems, so we use $O(n)$ space.

2. (Optional) There are n rocks in a river arranged in a line, and a frog is using them to jump across. At each step, the frog can either jump one rock ahead or two rocks ahead. Also the frog must land on the first rock and the last rock. For example, there are 8 ways to cross 6 rocks in a row. They are: (1,2,3,4,5,6), (1,3,4,5,6), (1,2,4,5,6), (1,2,3,4,6), (1,2,3,5,6), (1,3,5,6), (1,3,4,6), and (1,2,4,6).

   Write an efficient dynamic programming algorithm, that takes a number of rocks and counts the number of the feasible paths that the frog can take.

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

   Solution:

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

   Solution:

(c) Write pseudocode to solve this problem. You may submit actual working code as well. (Recommended)

Solution:

(d) State the big-O final running time and space usage of the algorithm. You will get more partial credit for a correct but slower solution with proper analysis than for an incorrect, "allegedly" faster solution.

Solution:

3. (10 points) You are given an array A of n elements, as well as a distance parameter k. Your goal is to choose a subsequence of the elements of A that have maximum sum, with the constraint that all elements that you choose have to be *at least* k entries apart in the array A. In class, we looked at this problem for the $k = 1$ case.

   As an example, say the array was $[4, 2, 9, 8, 3, 6, 2, 3, 1, 2, 5]$ and $k = 2$. Then some possible valid choices are shown below (they appear bolded).

   $[\mathbf{4}, 2, 9, \mathbf{8}, 3, 6, \mathbf{2}, 3, 1, \mathbf{2}, 5]$ with total 16

   $[\mathbf{4}, 2, 9, \mathbf{8}, 3, 6, \mathbf{2}, 3, 1, 2, \mathbf{5}]$ with total 19

   $[4, 2, \mathbf{9}, 8, 3, \mathbf{6}, 2, 3, 1, 2, \mathbf{5}]$ with total 20

   The last one is the one with maximum sum for this problem.

   In class, we solved the $k = 1$ case (Maximum independent set). Following the logic in class, how should the subproblem definition and recursion change to handle this case?

   Give the recursive solution, base cases, pseudocode or code, and analysis of the runtime in terms of n and k.

   Hint: this is a fairly small change.

Solution: Subproblem Definition:
$DP[i]$ represents the maximum sum of a subsequence ending at index i, where elements in the subsequence are at least k entries apart.

Recurrence Relation:
$DP[i] = \max(DP[j] + A[i])$ for all j such that $i - j \geq k$ and $0 \leq j < i$.

Base Cases:
$DP[0] = A[0]$ and $DP[1] = \max(A[0], A[1])$.

Code:

```
def maxSumWithDistanceApart(A, k):
    n = len(A)
    DP = [0] * n

    # Base cases
    DP[0] = A[0]
```

```
    DP[1] = max(A[0], A[1])

    for i in range(2, n):
        maxSum = 0
        for j in range(i - k, i):
            if j >= 0:
                maxSum = max(maxSum, DP[j])
        DP[i] = maxSum + A[i]

    return max(DP)

# Example usage
A = [4, 2, 9, 8, 3, 6, 2, 3, 1, 2, 5]
k = 2
print(maxSumWithDistanceApart(A, k))  # Output: 20
```

Runtime Analysis:

The algorithm iterates through the array once, and for each element, it looks back at most k elements. Therefore, the runtime complexity is $O(nk)$, where n is the number of elements in the array A and k is the distance parameter.

4. (40 points) Given two sequences X and Y, let $C(X, Y)$ denote the number of times that X appears as subsequence of Y (all the letters of X, in order). For instance, the sequence ABC appears 27 times as a subsequence of $AAABBBCCC$.

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

   Solution:
   Subproblem:
   Let's define $DP[i][j]$ as the number of times the substring $X[0 : i]$ appears as a subsequence in the string $Y[0 : j]$. Here, $X[0 : i]$ represents the substring of X from index 0 to i, and $Y[0 : j]$ represents the substring of Y from index 0 to j.

   (b) (10 points) What is the recursive relationship between the subproblems? Briefly 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: Defining the recursive relationship between subproblems:

   - When $X[i-1] == Y[j-1]$:
     If the last characters of X and Y match, it means we can either include this character in forming subsequences or ignore it.
     If we include the character, we add the count of subsequences formed by considering both strings without these characters (i.e., $DP[i-1][j-1]$).
     If we ignore the character, the count of subsequences remains the same as what

we had without this character in Y (i.e., $DP[i][j-1]$).
Therefore, the total count of subsequences is the sum of these two options: $DP[i][j] = DP[i-1][j-1] + DP[i][j-1]$.

- When $X[i-1] \neq Y[j-1]$:
If the last characters don't match, it means we cannot include this character in forming subsequences.
In this case, the count of subsequences remains the same as what we had without this character in Y (i.e., $DP[i][j] = DP[i][j-1]$).

Defining base cases:

- $DP[0][j] = 1$ for all j:
When X is an empty string, it is considered a subsequence of any string Y exactly once.
Therefore, for all lengths of Y, there is exactly one way to form an empty subsequence.

- $DP[i][0] = 0$ for all $i > 0$:
If Y is an empty string (excluding the case where both X and Y are empty), no non-empty string X can be a subsequence of it.
Therefore, the count of subsequences for any non-empty X is zero when Y is empty.

- $DP[0][0] = 1$:
Both empty strings are considered equal, so an empty string X is a subsequence of an empty string Y exactly once.

(c) (20 points) Write pseudocode (or working code) to solve this problem.
You should write a self contained function that takes two strings X, Y as input. The function should return the number of times X appears as a substring of Y.
Example Input 1:
ABC
AAABBBCCC

Example Output 1:
27

Example Input 2:
ABA
AABBAABBAA

Example Output 2:
32

Example Input 3:

ABC
ADABDADCBDC

Example Output 3:
7

Example Input 4:

GATTACA
AGAGGAATGCTCGATTTAAGGATAGATAAGAATAGGGCTCCCCCTCGATCAGCTCAAAACA

Example output 4:
147807

Example Input/Output 5 can be found on Blackboard. It may take a second.

Solution:

```
def count_subsequences(X, Y):
    m, n = len(X), len(Y)
    DP = [[0] * (n + 1) for _ in range(m + 1)]
    # Base case: an empty string is a subsequence of any string
    for j in range(n + 1):
        DP[0][j] = 1
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]:
                DP[i][j] = DP[i - 1][j - 1] + DP[i][j - 1]
            else:
                DP[i][j] = DP[i][j - 1]
    return DP[m][n]
# Test cases
X1 = "ABC"
Y1 = "AAABBBCCC"
print(count_subsequences(X1, Y1)) # Output: 27
X2 = "ABA"
Y2 = "AABBAABBAA"
print(count_subsequences(X2, Y2)) # Output: 32
X3 = "ABC"
Y3 = "ADABDADCBDC"
print(count_subsequences(X3, Y3)) # Output: 7
X4 = "GATTACA"
Y4 = "AGAGGAATGCTCGATTTAAGGATAGATAAGAATAGGGCTCCCCCTCGATCAGCTCAAAACA"
print(count_subsequences(X4, Y4)) # Output: 147807

Test case from Blackboard (the inputs were too long to display here)
```

```
X5 = "TTTTGTGCGAATGCAATGACACGTCTAGCCAGCGTCGAGCAGGTAC..."
Y5 = "GGCGACAAGGGACCCTAGAACACCCACAACATCTAAAGTAAATGAATATAA..."
print(count_subsequences(X5, Y5))
# Output: 22646581178657640392892987594567291895644734673710178895960
75729442072559891477587748721176239477970536469457321374669485906273 9
92006870582449209109034909346228037536535656646341533216797532990769 7
94758703935134813005739860345084404568686066173668362379609176407061
28769283005496795834483718581334537802473696403366614589869295400197
30047275208098781822915496951387404079416191037674957689612208308712 8
78552516385838709553424831580761920321904748513346492228410064636152 9
84977936124546378386401730572349817460481253147203892166027009888292 7
97199674168979166343991435299978354484161082559179129402842709854373 0
07664636182470438729831125421659926651639178681158013036703383341911 0
69263310719946393776962045341390627204924719135085227314785626924533 2
09707185030416224426864964261445914682769146329354496229 2
```

(d) (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 (although that's not really accurate since these numbers get huge). You will get more partial credit for a correct but slower solution with proper analysis than for an incorrect, "allegedly" faster solution.

Solution:
Constructing the DP table requires iterating through all positions of both strings X and Y, which takes $O(m \cdot n)$ time, where m is the length of string X and n is the length of string Y. Therefore, the time complexity of the algorithm is $O(m \cdot n)$, assuming constant-time integer addition and multiplication operations.

The space complexity of the algorithm is determined by the size of the DP table, which is of size $(m+1) \times (n+1)$. Therefore, the space complexity is $O(m \cdot n)$.

5. (50 points) You are a gold miner that just arrived in town. At time $t = 0$, you have no gold, but you do have a standard iron pickaxe, with which you can get $1oz$ of gold per hour. Once you have enough gold, you can use that gold to buy better pickaxes to earn more gold per hour. There are n pickaxes available for purchase, and each pickaxe i can earn v_i oz per hour of gold, and costs c_i oz of gold to buy (no returns or resales). A more productive pickaxe is more expensive, and you can assume that they are given to you in sorted order of price / productivity. It takes 1 hour to go to the store and buy the new pickaxe, so for the hour that you are upgrading your pickaxe, you aren't earning money by mining.

   In the very long run, it would sense to eventually buy the most expensive and productive pickaxe and use it forever. However, you are only in town for t hours, at which point you will abandon all your tools and leave town with just your gold.

   You want to figure out: what is the most gold that you can possibly have at the end of t hours?

   (a) (Optional) Construct a scenario (costs and productivity of available pickaxes, total time t), with one pickaxe where it does not make sense to buy the pickaxe, even though you can afford it at some point in your mining. Explain your construction.

Solution:

(b) (Optional) Construct a scenario (costs and productivity of available pickaxes, total time t), with two available pickaxes where it does not make sense to buy the first pickaxe in the list, but it makes sense to buy the second, even though the productivity / value ratio of both pickaxes is the same. Explain your construction.

Solution:

(c) (10 points) We will first solve this problem using BFS. Construct a graph as follows.

As with similar problems that you've done in class, each vertex represents your "state" - in this case, how much gold you have and which pickaxe you have.

(Note that an upper bound on the amount of gold you might have is $v_{max} * t$.)

Create vertices (i, k), where i represents the current pickaxe that you own at that state, and k represents the current amount of gold that you own at that state.

Figure out how to add directed edges to this graph to connect each state to other states in a way that BFS on this graph can help you determine the most gold you can possibly have after t hours. Explain your construction and how BFS will help answer the question.

How big is the graph that you made? What is the runtime of this procedure, in terms of n, t, and v_{max}?

Solution: Directed Edges:
For each state (i, k), if buying the next pickaxe $(i + 1)$ is feasible (i.e., $i + 1 \leq n$ and $k \geq c_{i+1}$), add a directed edge from (i, k) to $(i + 1, k - c_{i+1})$. This represents the action of buying the next pickaxe.

For each state (i, k), add a directed edge from (i, k) to $(i, k + v_i)$ representing the action of mining with the current pickaxe. The BFS algorithm on this graph will help us determine the most gold we can possibly have after t hours because it explores all possible states in a breadth-first manner, ensuring that we consider all possible sequences of actions (buying pickaxes or mining) within the time limit.

Size of the Graph:
The number of vertices in the graph is $O(nt)$ because there are n pickaxes and each state can have up to t different amounts of gold. The number of directed edges is also $O(nt)$ because each state can have at most two outgoing edges (one for mining and one for buying a pickaxe).

Runtime Analysis:
The runtime of this procedure is primarily determined by the BFS algorithm, which has a time complexity of $O(V + E)$, where V is the number of vertices, $O(v_{max} * t * n)$, and E is the number of edges in the graph, $O(v_{max} * t * n^2)$. The reason why n^2 is because in the worst case when we have enough gold to purchase all other available pickaxes we will have n edges from that vertex going to those pickaxes that we can possibly purchase. So the overall runtime complexity of $O(V + E)$ is $O(v_{max} * t * n^2)$.

(d) (15 points) We will now try to solve this problem a different way using dynamic programming. Use the following subproblem definition:

Let MostGold(i,j) be the most gold that you can possibly have after j hours while having pickaxe i. (Pickaxe 0 is your starting pickaxe). If it is impossible to have pickaxe i at time j, then this should be $-\infty$.

Find a recurrence for this subproblem: some way to express MostGold(i,j) in terms of smaller values of MostGold.

Hint: Remember to think *backwards* not forwards. The question is not "I am in state (i,j). Where do I go next?". Instead, the question is "I am in state (i,j). How did I get here?".

Solution: To find how we got to find MostGold at time j with a pickaxe i we have two cases to consider:

Case 1 (we kept digging with the same pickaxe):
In this case, we added the productivity of that pickaxe i (the same one) to the amount we have at $j-1$

```
MostGold[i][j] = MostGold[i][j-1] + p[i]
```

Case 2 (we just bought a new pickaxe j):
In this case, we subtracted the cost of the pickaxe i from the most amount of gold we could make at time $j-1$

```
MaxGold[i][j] = biggestVal - c[i]
```

We run another loop to find the $biggestVal$ - biggest value of gold we could make at time $j-1$:

```
for k in range(i):
    biggestVal = max(biggestVal, MaxGold[k][j-1])
```

(e) (20 points) Implement code to solve this problem. Your function should take three inputs: the time t, an array of the set of pickaxe costs c and an array of the set of pickaxe productivities p.

Solution:

```
def ReadInput():
    #inputs
    C=[]
    P=[]
    n, t = map(int, input().split())

    for _ in range(0, n):
        line = input()
```

```
            c, p = map(int, line.split())
            C.append(c)
            P.append(p)

        print(MostGold(C,P,t))


    def MostGold(c,p,t):
        # initializing dpTable to -inf
        dpTable = [[float('-inf') for j in
            range(t + 1)] for i in range(len(c) + 1)]
        # setting the first value to 0
        dpTable[0][0] = 0


        # main rucursive idea
        for i in range(len(c)):
            for j in range(1,t + 1):
                # resetting biggestVal for each new (i, j) pair
                biggestVal = float('-inf')

                for k in range(i):
                    biggestVal = max(biggestVal, dpTable[k][j-1])

                if(biggestVal >= c[i]) and dpTable[i][j-1] < 0:
                    dpTable[i][j] = biggestVal - c[i]

                else:
                    dpTable[i][j] = dpTable[i][j-1] + p[i]

        # Printing the Table (excluded from the Runtime Calculations)
        maxGold = max(row[-1] for row in dpTable)
        print(dpTable)

        return maxGold


    ReadInput()
```

(f) (5 points) Analyze the runtime of this algorithm. For which settings would you prefer the dynamic programming solution to the graph based solution?

Solution: Let's denote n as the length of the list c.
t stays the same.

The runtime of the algorithm is $O(n * t * n)$ or $O(n^2 * t)$

This is because there are three loops in the algorithm:
- The outer loop iterates over i from 0 to the length of list c : $O(n)$ runtime
- The middle loop iterates over j from 1 to t : $O(t)$ runtime
- The inner loop iterates over k from 0 to i, because i can go up to n : $O(n)$ runtime

For smaller inputs, it doesn't matter which algorithm to run, because the runtime difference would be negligible. However, for big inputs, the DP algorithm is going to be faster (and more preferable) because it is $O(n^2 * t)$ as opposed to $O(v_{max} * t * n^2)$ (in the graph-based algorithm).

6. (Bonus) [50] 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, an explanation of your solution with subproblems, recursion, code, and runtime, and a screen shot showing that you passed all the tests on kattis.