CS 3510 Section B: Design & Analysis of Algorithms October 13th 2024

Problem Set 6: Dynamic Programming II

YOUR NAME HERE Due: October 21st 2024

- Please type your solutions using LaTeX or any other software. Handwritten solutions will not be accepted.
- If we ask for a specific running time, a correct solution achieving it will receive full credit even if a faster solution exists. If no target running time is provided, faster solutions will earn more credit.

Problem 1: Beam Cutting *(33 points)*

You have finally secured permission to build a monorail across the Georgia Tech campus! To begin construction, you purchase a large linear beam whose length is the circumference of the Georgia Tech campus. However, you realize that the stations ought to use a different material than the rest of the track, so you want to slice this beam at various points to create gaps to eventually insert stations.
The beam of length n has points labeled 0 to n from left-to-right (where point 0 is at the left end and point n is at the right end). A classmate of yours does some measurements on campus to find exact locations along the beam where you should cut, and gives you a **sorted** list `cuts` of **distinct** integers at which to perform the cuts. Each cut will be on a point between 1 and $n-1$, and any cut on one beam will split it into two beams. You can perform the cuts at any order, but must cut at exactly the locations specified (due to the color scheme, it is **not** valid to change these locations, even if the resultant component lengths are the same). The cost of cutting a component of the beam into slices of length l and r is $l+r$, but Georgia Tech makes an administrative error and accidentally gives you a budget of $l \times r$ for each such cut. This is an arbitrage opportunity!
You want to optimally order your cuts to maximize your profit, the difference between the amount of money Georgia Tech gives you and the amount of money you actually spend cutting the beam. Given n (the length of your beam), and a sorted list `cuts` of size c of integer locations at which to make the cuts, write a polynomial-time algorithm that returns the (possibly negative) maximum amount of profit you can generate.

(a) Define the entries of your table in words. E.g. $T(i)$ or $T(i, j)$ is ...

Solution: $T(i, j)$ is the maximum profit achievable by making cuts at all the specified points in the sub-beam starting at index i and ending at index j of the cuts array (inclusive).

(b) State recurrence for entries of the table in terms of smaller subproblems. Briefly explain in words why it is correct. Remember to state your base case(s) as well.

Solution: Base Case: T(i, i) = (cuts[i] - cuts[i-1]) * (n - cuts[i]) The profit from cutting the beam between cuts[i-1] and cuts[i] Recursive Case: T(i, j) = maxT(i, k) + T(k + 1, j) + (cuts[j] - cuts[i - 1]) * (n - cuts[j]) for all k where $i \leq k < j$

(c) Write **pseudocode** for your algorithm to solve this problem.

```
c = len(cuts)
T = [-infinity for all values in range(c + 1) for in range(c + 1)]

# Base cases: single cuts (between two consecutive cut points)
for i in range(1, c + 1):
  T[i][i] = (cuts[i - 1] - cuts[i - 2]) * (n - cuts[i - 1])

# Iterate over sub-beam lengths (similar to MCM chain length)
```

```
for length in range(2, c + 1):
  for i in range(1, c - length + 2):
    j = i + length - 1
    for k in range(i, j):
      cost = (cuts[j - 1] - cuts[i - 2])
      profit = cost * (n - cuts[j - 1])
      T[i][j] = max(T[i][j], T[i][k] + T[k + 1][j] + profit)

return T[1][c]  # Return the maximum profit for the entire beam
```

(d) Analyze the running time of your algorithm.

Solution: The time complexity is O(c^3) where 'c' is the number of cuts, due to the three nested loops.

Problem 2: Debt Collection *(33 points)*

You're an upstanding citizen, and your secret stash of money has n different types of bills labeled 1 to n. For each type of bill i, it is worth v_i dollars per bill, and the number of those types of bills you have is c_i. As the upstanding citizen you are, you're trying to pay back a debt of value of exactly D dollars. You don't want to overpay this debt or underpay this debt. Design an algorithm to determine the number of different selections of all the bills to exactly pay off your debt. Note that different bills of the same type are considered identical, so two selections of bills are considered the same if and only if they use the same number of bills for every bill type.

Example: Let's say you have $n = 2$ types of bills and you need to pay a debt of $D = 15$. Bill type 1 has a value of 5 dollars per bill and you have 3 bills of this type ($v_1 = 5, c_1 = 3$). Bill type 2 has a value of 1 dollar per bill and you have 10 bills of this type ($v_2 = 1, c_2 = 10$). There are 3 selection of the bills to pay off this debt:

One bill of type 1, ten bills of type 2 (1(5) + 10 (1) = 15)
Two bills of type 1, five bills of type 2 (2(5) + 5(1) = 15)
Three bills of type 1, zero bills of type 2 (3(5) + 0(1) = 15)

(a) Define the entries of your table in words. E.g. $T(i)$ or $T(i, j)$ is ...

Solution: T[i][j]: The number of combinations to reach amount j using the first i bill types.

(b) State recurrence for entries of the table in terms of smaller subproblems. Briefly explain in words why it is correct. Remember to state your base case(s) as well.

Solution: base case: T[i][j] = 0 for all i and all j except 0. T[i][0] = 1 for all i recursive case: If values[i - 1] ¿ j: T[i][j] = T[i - 1][j] If values[i - 1] ≤ j: T[i][j] = T[i - 1][j] + T[i - 1][j - k * values[i - 1]] for k from 1 to min(counts[i - 1], j // values[i - 1]). considering using the bill k times.

(c) Write **pseudocode** for your algorithm to solve this problem.

```
T = new Array(n + 1)
for i from 0 to n:
  T[i] = new Array(D + 1)
  for j from 0 to D:
    T[i][j] = 0

for i from 0 to n:
  T[i][0] = 1

for i from 1 to n:
  for j from 1 to D:
```

```
            T[i][j] = T[i - 1][j]
          if values[i - 1] <= j:
            for k from 1 to min(counts[i - 1], j // values[i - 1]):
              T[i][j] = T[i][j] + T[i - 1][j - k * values[i - 1]]

      return T[n][D]
```

(d) Analyze the running time of your algorithm.

Solution: The algorithm has a time complexity of O(n * D * K) with n being the number of types of bills, D being total dollars, and K being max(c).

Problem 3: Eating Bananas *(33 points)*

There is a monkey trying to maximize the amount of bananas it can eat. The monkey has M seconds to eat from N bags of bananas labeled 1 to N. The monkey knows that bag i has b_i bananas and it takes t_i seconds to finish all the bananas in the bag. However, the monkey cannot eat more than K bags of bananas, and each bag that the monkey chooses must have all of its bananas eaten. Design a dynamic programming algorithm to determine the maximum amount of bananas the monkey can eat.

Big Hint: Start by ignoring the K constraint, then try to add it in as a separate dimension.

(a) Define the entries of your table in words. E.g. $T(i)$ or $T(i, j)$ is ...

Solution: T is a 3D array where T[i][j][k] stores the maximum number of bananas the monkey can eat by considering the first i bags, with a maximum time of j seconds, and by eating at most k bags.

(b) State recurrence for entries of the table in terms of smaller subproblems. Briefly explain in words why it is correct. Remember to state your base case(s) as well.

Solution: Base case: T[0][0][0] = 0 Implicitly, when i, j, or k is 0 as no bananas can be eaten. recursive case: t[i - 1] ¿ j: If the time to eat the current bag exceeds the remaining time, the monkey cannot eat it. So, the maximum bananas remain the same as with the previous bags: T[i][j][k] = T[i - 1][j][k]. t[i - 1] ¡= j: If the monkey has enough time to eat the current bag, it has two options: Not eating the current bag: T[i - 1][j][k] Eating the current bag: b[i - 1] + T[i - 1][j - t[i - 1]][k - 1]

(c) Write **pseudocode** for your algorithm to solve this problem.

```
N = length(b)  // Number of bags

T = new Array(N + 1)
for i from 0 to N:
  T[i] = new Array(M + 1)
  for j from 0 to M:
    T[i][j] = new Array(K + 1)
    for k from 0 to K:
      T[i][j][k] = 0

// Iterate through each bag, time, and bag count
for i from 1 to N:
  for j from 1 to M:
    for k from 1 to K:
      if t[i - 1] > j:  // If eating the current bag takes too long
        T[i][j][k] = T[i - 1][j][k]
      else:
            // Choose the maximum between not eating and eating the current bag
            T[i][j][k] = max(T[i - 1][j][k], b[i - 1] + T[i - 1][j - t[i - 1]][k - 1])

    return T[N][M][K]
```

(d) Analyze the running time of your algorithm.

Solution: Time Complexity is O(N * M * K) b/c of the three nested loops.