Homework 6

Due: Friday, Oct 17, 11:59 PM PT

For each of the questions below, design a dynamic programming based algorithm by answering the following sub-parts:

a) Define in plain English the **subproblems** that your dynamic programming algorithm solves. (In other words, your answer should be something like "Opt(x,y,z..) denotes the min/max/?? value obtained from x items/people/?? , with y time/money/?? , z __" and so on. Do not merely paraphrase the given problem or describe the high-level approach etc.) (4 points)

b) Define a recurrence relation that expresses the value of each subproblem in terms of the smaller subproblems. (6 points)

c) Using the recurrence formula in part b, write pseudocode to find the solution.
Make sure you specify all the base cases and their values, as well as the final answer to the problem. (6 points)

d) What is the time complexity of the solution? Further, determine whether it is an efficient solution or not. (2+2 points)

1. James wants to surprise his fiancée with exquisite jewellery made from gold. He owns many gold mines and wants the goldsmith to make different items. He gives the goldsmith a list of n different types of items, where each item type weighs w_i grams and each has a value of v_i. James' fiancée likes jewellery but only has space for some items, as her jewellery box has a maximum capacity of W grams. James wants the goldsmith to make items from the list and insists that the combined weight of all the items must be equal to or less than W grams, but at the same time, maximizing the total value of all the created items. Design a DP algorithm to output the total optimal value of the items, whose total weight doesn't exceed W grams. Note that the final jewellery box can contain multiple instances of the same jewellery. (20 points)

Solution

a) Let OPT(k, w) be the maximum value achievable using a knapsack of capacity $0 \leq w \leq W$ and with the first k types of items $1 \leq k \leq n$.
General tip: It is important to use different symbols to describe the general subproblem, from the ones that are given as input (**k** items vs **n** total items, capacity **w** vs total cap **W** etc.)

b) We find the recurrence relation of OPT(k, w) as follows. Since we have infinitely many items of each type, we choose between the following two cases:
• We include another item of type k and solve the sub-problem OPT(k, w− w_k). Notice that the first parameter is still k, since we can include more items of type k as we want.
• We do not include any item of type k and move to consider the next type of item, thus solving the sub-problem OPT(k − 1, w).
Therefore, we have

$$OPT(k, w) = \max\{OPT(k - 1, w), OPT(k, w - w_k) + v_k\}.$$

c) As a base case, we cover k=0, since otherwise the first term of the recurrence would call a subproblem with k = -1. Secondly, we would also need to take care of the second term calling subproblems with negative w, but we can take care of that simply when implementing the recurrence (see below).

```
For w= 0 to W:
        OPT[0, w] = 0          // Base case

For k = 1 to n:
        For w = 0 to W:
                If  w < w_k:    OPT[k, w] = OPT[k − 1, w]
                Else:           OPT[k, w] = max(OPT[k − 1, w],  OPT[k, w − w_k] + v_k)

Return OPT[n, W]               //Final answer
```

d) Given the double loop, the runtime complexity is $\Theta(nW)$. Since it depends polynomially on W, but not polynomially on its input size log W, this is Pseudo-polynomial, and thus, NOT efficient.

2. You are participating in a game show, where you are presented with a line of briefcases. On top of each briefcase, you can see the amount of money you can win by choosing that briefcase. However, after each time you pick out a briefcase, the host will remove the two briefcases adjacent to it (or the only adjacent briefcase if the corner-one was picked). Your goal is to maximize the amount of money you can win; however, if your final amount collected is more than K dollars, you will lose all your money. Design a dynamic programming algorithm to compute the maximum earnings possible without exceeding K dollars by picking suitcases as per the aforementioned rules.

   For example, if your starting set of briefcase values is [44, 51, 46, 55, 23, 12] and you want to make at most 100$. Here is how your selection can play out:

You pick the 44$ briefcase. The host removes the 51$ briefcase, and the remaining array would be [46, 55, 23, 12]. You now pick the 55$ briefcase and the host removes the 46$ and 23$ briefcases, and the remaining array would be [12]. You stop playing since you don't want to make more than 100$, therefore the game ends, and 99$ is the maximum amount you can make playing this game. (20 points)

Solution:

a) OPT(i, j) = The maximum money earned when using the first i briefcases, when you can at most make j dollars

b)

$$OPT(i,j) = \begin{cases} \max(OPT(i-1,j), \text{briefcase}[i] + OPT(i-2, j-\text{briefcase}[i])), & \text{if briefcase}[i] \leq j \\ OPT(i-1,j), & \text{if } j > \text{briefcase}[i] \end{cases}$$

Where briefcase[i] is the value of the ith briefcase (for convenience, assume briefcase[0] = 0)

We use OPT(i-2, j-briefcase[i]) since the i-1th gets removed by the host, and you add the value of the ith briefcase to your collection.

c)

Algorithm 1:

Input: An array $briefcase$ of n briefcase values, maximum possible earning K

Output: The maximum money earned by you that is less than K

```
for i ← 0 to n do
    OPT[i][0] ← 0;
end
for i ← 0 to K do
    OPT[0][i] ← 0;
end
for i ← 1 to n do
    for j ← 0 to K do
        if briefcase[i] ≤ j then
            use recursion from part 2
        else
            OPT[i][j] ← OPT[i − 1][j];
        end
    end
end
return OPT[n][K]
```

d) Runtime is **Θ**(nK), making this pseudo-polynomial (thus, not efficient), similar to Q1.

3. Suppose you have a rod of length N, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth p_i dollars. Devise a Dynamic Programming algorithm to determine the maximum amount of money you can get by cutting the rod strategically and selling the cut pieces. (20 points)

Solution:

a) Let r(n) be the maximum money that can be made from rod of length *n* by cutting it into pieces and selling ($0 \leq n \leq N$)

b) At each remaining length *n* of the rod, we can choose to cut the rod at any point, obtain money for the cut piece, and recursively compute the maximum money we can get for the remaining rod.

$$r(n) = \max_{1 \le j \le n} \{p_j + r(n - j)\}$$

Note that, it is important to specify the indices (subscript) for max when taking max over a series of terms.

c)

```
r[0] = 0                    //Base case
For n = 1 to N:
        q = −∞
        for j = 1 to n
                q = max(q, p[j] + r[n - j])
        end for
        r[n] = q
end for
return r[N]      //Final answer
```

d) The time complexity is $\theta(n^2)$ because of the double-nested for loop. Since the input size is also $\theta(n)$ (due to the n values p_i in the input), this is an efficient algorithm.

Ungraded Problems:

1. Given a set of numbers, design a DP algorithm to check whether it can be partitioned into two subsets such that the sum of the elements in both subsets is equal. (20 points)

Solution:
This problem is the same as Subset Sum problem with target sum t = S/2 (where S is the total sum). If the total sum is odd, we can simply return "False" (i.e. not possible) and run this DP algorithm if S is even.
Suppose the given set of numbers is denoted as $\{x_1, \ldots, x_n\}$

a) Suppose dp(j, s) = true if sum s can be achieved using first j numbers, false otherwise. ($0 \leq s \leq t$, $0 \leq j \leq n$)

b) Recurrence: $dp(j, s) = dp(j-1, s-x_j) \vee dp(j-1,s)$ ($\vee$ denotes logical OR)
The two terms of the recurrence correspond to whether x_j is chosen or not.

c) OPT[0,0] = True //Base case
For s = 1 to t:
OPT[0, s] = False // Base case

For j = 1 to n:
For s = 0 to t:
If $s < x_j$: OPT[j, s] = OPT[j − 1, s]
Else: $OPT[k, w] = OPT[j - 1, s] \vee OPT[j -1, s-x_j]$)

Return OPT[n, t] //Final answer

d) Runtime is O(nS), thus, pseudo-polynomial.

2. You are planning your holiday and have a list of events you'd like to attend. Each event is described by $[s_i, e_i, v_i]$, where s_i and e_i are the start day and end day of the event, and v_i represents how important you think the event is. Consider that:
    1. You can only attend one event at a time.
    2. If you choose to take one event, you must attend the entire event.
    3. You can attend k events at most.

   Design an algorithm that maximizes the total value of the events you attend. (8 points)

Solution:

Sort the events by end times.

a) dp(i, j) = max value from the first i events, selecting at most j events.

b) dp(i, j) = max(dp(i - 1, j), dp(p, j-1) + v_i)

Where p denotes the last event that's non-overlapping with event i.

P can be calculated in linear time at each subproblem (naive) OR in log n time with binary search for more efficiency or in O(1) time by computing the 'p' for each 'i' altogether with one linear pass initially (details left as exercise). If no target time complexity is specified, the naive approach is good enough.

Then we iterate through all possible event sets, using the following pseudo code:

1. Set i=0. Let n be the total number of events. Initialize dp as all zeros
2. For i=0; i<n; i++

    For j=0; j<k; j++

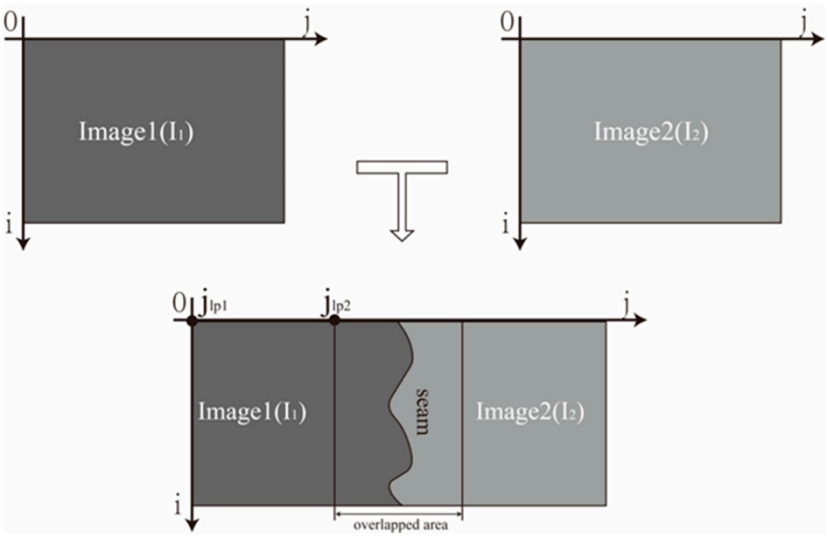
        a. Consider that we do not take the i th event, then we have:

        dp[i][j] = max(dp[i][j], dp[i - 1][j])

        b. Consider we take the i th event. Use either approach mentioned above to find the last possible non-overlapping event that we can take. Let it be p. Then we have:

        dp[i][j] = max(dp[i][j], dp[p][j-1] + v[i])
3. Return dp[n][k]

3. Shawn decided to go on a trip for spring break, so he decided to visit San Francisco and went to see his favorite monument, the Golden Gate Bridge. Immediately, Shawn decided to take a picture of the magnificent bridge; however, due to its size, he was unable to fit the entire bridge in one picture. Relying on his expertise as a computer scientist, Shawn decided to take two pictures from the bridge and stitch the images together when he got back home. Now Shawn is back from his trip and can't wait to show his friends the pictures from his trip. He knows that simply putting his two images together would show the lack of his photographic skills to his friends, so he decides to stitch the two images together such that the stitching is not obvious. To do this, he knows that the two images have the same height H and share 5 columns of pixels together, so he calculates the absolute difference between the pixels of the two images in the shared area. His goal is to find a path through the pixels in the shared area that minimizes the total sum of pixel distances. Furthermore, he knows that once he selects a pixel, he can select one that is either directly above it or diagonally above it as the next pixel, because otherwise the stitching becomes way too obvious. Help Shawn find the best path to stitch his images so that he can show off in front of his friends. The figure below provides an abstract representation of what Shawn is attempting to do. (10 points)



Solution:

a) OPT[i, j]= The minimum sum of pixel distance between two images from the bottom of the shared part to the pixel [i,j]

b)

$$OPT(i,j) = \begin{cases} \min(OPT(i-1,j), OPT(i-1,j-1), OPT(i-1,j+1)), & \text{if } 1 < i < 5 \\ \min(OPT(i-1,j), OPT(i-1,j+1)), & \text{if } i == 1 \\ \min(OPT(i-1,j), OPT(i-1,j-1)), & \text{if } i == 5 \end{cases} + \text{diff(i,j)}$$

c) For demonstration, the pseudo-code below includes the computation of the actual path that minimizes the values, rather than only compute the minimum value

Algorithm 1:

Input: An 2-d array $diff$ that displays the absolute diff between pixels of two images, height of the image H
Output: The path with minimum distance
for $i \leftarrow 1$ **to** 5 **do**
  $OPT[0][i] \leftarrow diff[0][i]$;
end
for $i \leftarrow 1$ **to** $H - 1$ **do**
  for $j \leftarrow 1$ **to** 5 **do**
    use the recursion from part 2
  end
end
min_val $\leftarrow \infty$;
min_index $\leftarrow -1$;
for $j \leftarrow 1$ **to** 5 **do**
  min_val$\leftarrow$ min($OPT[H-1][j]$, min_val)
  if $min\_val = OPT[H-1][j]$ **then**
    min_index $\leftarrow$ j;
  end
end
path $\leftarrow$ [min_index]
for $i \leftarrow H - 2$ **to** 0 **do**
  based on the recursion from part 2 find the index
  OPT[i + 1][min_index] was generated from
  update the value of min_index
  append min_index to path
end
return path

d) O(H), polynomial time (since we have the 'diff' array of 5H values in the input).

1. Solve Kleinberg and Tardos, Chapter 6, Exercise 5. (10 points)

Solution:
Let $Y_{i,k}$ denote the substring $y_i y_{i+1} \ldots y_k$.

a) Let OPT(k) denote the quality of an optimal segmentation of the substring $Y_{1,k}$.

b) An optimal segmentation of this substring $Y_{1,k}$ will have quality equal to the quality of the last word (say $y_i y_{i+1} \ldots y_k$) in the segmentation plus the quality of an optimal solution to the substring $Y_{1,i}$. Otherwise, we could use an optimal solution to $Y_{1,i}$ to improve OPT(k), which would lead to a contradiction.

$$OPT(k) = \max_{0<i<k} OPT(i) + \text{quality}(Y_{i+1,k})$$

c) We can begin solving the above recurrence with the initial condition that OPT(0) = 0 and then go on to compute OPT(k) for k = 1, 2, . . . , n, similar to graded Q3.
The segmentation corresponding to OPT(n) is the final answer

d) The runtime is $\Theta(n^2)$, assuming the function quality() takes O(1) time. If quality() takes time linear in length of its argument, then runtime would be $\Theta(n^3)$. In either case, polynomial time.