CS5800 Algorithms

Recitations from Oct 14, 2019 to Oct 18, 2019

Ravi Sundaram

Dynamic Programming Solution Guidelines

Dynamic programming can be very tricky and this template will help guide you through solving new problems. Get in the habit of going through the list and filling everything out step by step. We will grade harshly on missing items. And if there's no english description of the two items mentioned below then the solution will get an **AUTOMATIC 0** on homeworks and exams with no exceptions.

1. (**AUTOMATIC 0 IF MISSING**) English description of the
   (a) problem you are actually solving with your recursion.
   (b) logic behind your recursion.
2. The actual recursion. And don't forget the base cases!
3. Brief description of how an iterative algorithm would loop through the memoization array and fill the values (make sure your order makes sense, and that values are already filled when you call them). Pseudocode isn't required, just a couple sentences.
4. The number of subproblems you have to solve.
5. How much time each subproblem takes to solve (usually constant or linear).
6. Final running time.
7. Final value we have to return.

Problem 1 (Dynamic Programming)

1. You are given a number t and k piles of infinitely many coins of values $d_1, d_2, ..., d_k$. Return the minimum number of coins whose values sum to t. For example, if $t = 9$, $k = 3$, and the values are $d_1 = 1, d_2 = 6, d_3 = 4$, then the minimum number of coins needed is 3 (the combination 4,4,1).
2. You are about to go on a journey, and you are thinking of what to take in your bag (of limited size). You are given a set of n items. Each item has a value v_i and a weight w_i that are always in $\{0, 1, 2, ..., n\}$. You are given a target value V and a target weight W. Describe a polynomial time algorithm that determines if there exists a subset S of the the items such that $\sum_{i \in S} v_i \geq V$ and $\sum_{i \in S} w_i \leq W$
3. Given an array $A[1...n]$ of letters, compute the length of the longest palindrome subsequence of A. A sequence $P[1...m]$ is a palindrome if $P[i] = P[m - i + 1]$ for every i. For example, RACECAR is a palindrome, while REAR is not a palindrome. And the longest palindrome subsequence in the string ABETEB is BEEB.

4. Given an array $A[1...n]$ of integers, compute the length of the longest alternating subsequence. A sequence $P[1..m]$ is alternating if $P[i] < P[i-1]$ for every even index $i \geq 2$, and if $P[i] > P[i-1]$ for every odd index $i \geq 3$. For example, in the sequence

$$(3, 1, 4, 1, 5, 9, 2, 6, 5, 2, 5, 8)$$

the longest alternating subsequence is

$$(3, 1, 4, 1, 5, 2, 6, 5, 8)$$

so we should return 9.

Solution:

1. We will be filling out $OPT[j]$, where $OPT[j]$ contains the minimum number of coins whose values sum to the current j. When we are filling this out, we can get to j by adding one coin of value d_i to some group of coins that has value $j - d_i$. So we check the minimum of among all $OPT[j - d_i]$ for all d_i.

   Recurrence Relation (and Base Case)

$$OPT[j] = \begin{cases} \infty & \text{if } j < 0 \\ 0 & \text{if } j = 0 \\ min_{1 \leq i \leq k}(OPT[j - d_i] + 1) \end{cases}$$

   We start at $j = 0$ and go up to t. There are n subproblems, but at each step we need to find the min out of k elements. So there's $O(k)$ work done per subproblem. So the runtime is $O(kt)$, and we return $OPT[t]$ at the end.

2. The subproblem we're going to be filling out is a boolean table $S(i, x, y)$ which is true if there's a subset of the first i items such that the sum of the values is equal to x and the sum of the weight is equal to y. If we're at item i, and we add it to some subset, then we add the item's weight and value to the running totals of the susbsets value and weight. So to get to the values x, y, we check if $S[i-1, x-v_i, y-w_i]$ is true. We can also choose not to add item i, and then the running totals are unchanged and we check if $S[i-1, x, y]$ is true.

   Recurrence Relation (and Base Case)

$$S[i, x, y] = \begin{cases} 0 & \text{if } x < 0 \vee y < 0 \\ 1 & \text{if } x = y = 0 \\ S[i-1, x, y] \vee S[i-1, x-v_i, y-w_i] \end{cases}$$

   In the outermost loop we loop over i, starting from 0. Then the next two loops are over x, y, starting from 0 to n^2. So there are $O(n^5)$ subproblems, and in each subproblem we spend constant time since we just look up the values in the array. The runtime is $O(n^5)$. Note that for our return value, we look over all values of $S(n, v, w)$ such that $v \geq V$ and $w \leq W$, and if anyone of them are true we return true. Otherwise we return false.

3. SOLUTION 1 (incomplete)

- English Description of problem: We create a new array where $Pal[i,j]$ will contain the length of the longest palindrome subsequence of the the subarray of A that starts at index i and ends at index j.
- English Description of logic: It's easy to see that if $A[i] = A[j]$, for some $i < j$, we can include that in palindrome subsequence and increase i by 1 and decrease j by 1. And if they're not equal, we either increase i or decrease j, but not both.
- Recurrence Relation (and Base Case)

$$Pal[i,j] = \begin{cases} Pal[i,j] = 1 & i = j \\ max(Pal[i,j+1], Pal[i-1,j]) & A[i] \neq A[j] \\ 2 + Pal[i-1,j+1] & A[i] = A[j] \end{cases}$$

- Iterative Algorithm: Now it's not immediately clear how we can memoize $Pal[i,j]$ correctly. The next solution is an alternative way we can look at the recursion that uses length as a parameter, and has a natural iterative algorithm.

4. SOLUTION 2 (complete)

- English Description of problem: $P[i,k]$ will contain the length of the longest palindrome subsequence of the subarray $A[i...k-i+1]$. Note that k is the length of the string we are looking at.
- English Description of what we're doing: In essence the logic is same as above. If $A[i] = A[j]$, for some $i < j$, we can include that in palindrome subsequence and decrease the length of the subarray we're considering by 2, and increment i by 1. And if they're not equal, we either decrement the right index (which we can do by keeping i the same but decrementing k), or we decrement the left index (so we increment i, and then to stay at the same right index we need to decrement k).
- Recurrence Relation (and Base Case)

$$P[i,k] = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ 2 + P[i+1,k-2] & \text{if } A[i] = A[i+k-1], k \geq 2 \\ max(P[i,k-1], P[i+1,k-1]) & \text{otherwise} \end{cases}$$

- Iterative Algorithm - Now it's easier to see how we can fill P in the correct order. Our outer for loop will increment k starting from 0, and our inner loop will increment on i in the inner loop. Now, the following items become easier to analyze.
- Number of subproblems - The length goes up to n, and the index goes up to n. We have a subproblem for each pair of indices, so there are $O(n^2)$ subproblems.
- Time per subproblem - Checking if $A[i] = A[j]$ is constant time, and we might need to look up the values $P[i+1,k-2], P[i,k-1], P[i+1,k-1]$ but these will already be computed and stored in P so the lookup is constant time.
- Total Runtime - We get $O(n^2)$.
- Final Return Value - $P[1,n]$

5. Let $Alt[i, b, j]$ be the longest alternating subsequence in $A[j...n]$ whose first value is greater than $A[i]$ if $b = 1$, and whose first value is less than $A[i]$ if $b = 0$.

   Note that when $b = 1$, if $A[j] \leq A[i]$, then we can immediately throw out $A[j]$ and move on to $A[j + 1]$.

   However if $A[j] > A[i]$, then we might include $A[j]$ in the optimal alternating subsequence, or we might ignore it anyways. If we include it, then b switches from 1 to 0 (to maintain the alternating condition), and we increment j.

   When $b = 0$, we flip the inequalities between $A[i] and A[j]$ and we get a similar analysis.

   Recurrence Relation (and Base Case)

$$Alt[i, b, j] = \begin{cases} 0 & \text{if } j > n \\ Alt[i, b, j + 1] & \text{if } (b = 1 \wedge A[j] \leq A[i]) \vee (b = 0 \wedge A[j] \geq A[i]) \\ max(Alt[i, b, j + 1], 1 + Alt[j, 1 - b, j + 1]) & \text{otherwise} \end{cases}$$

   For our iterative algorithm, our outermost loop will be on j, and we start at $j = n$ and work our way downwards. Then our next loop will be over i, starting at $i = j$ and working downwards. Then our innermost loop is over b in any order. There are $O(n^2)$ subproblems, and each subproblem takes constant time since we're just looking up values in an array. So total runtime is $O(n^2)$. For the final answer we return $Alt[0, 1, 1]$ (and we set $A[0] = -\infty$ so the edge case works out).

...

■

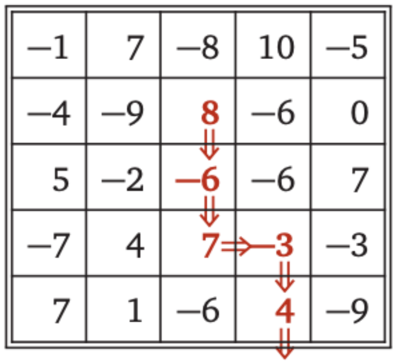
Problem 2 (Extras...)

1. Given an array $A[1...n]$ of integers, compute the length of the longest decreasing subsequence. A sequence $P[1...m]$ is decreasing if $P[i] < P[i - 1]$ for $2 \leq i \leq m$.

2. Given an array $A[1...n]$ of letters, give a dynamic programming solution to finding the length of the shortest supersequence that's a palindrome. For example, if we are given the string INTERNET, the shortest supersequence that's a palindrome is INTERNRETNI, so your solution should return 11.

3. (Taken from Jeff Erickson's book) Vankins Mile is an American solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.For example, given the grid below, the player can score $8-6+7-3+4 = 10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is not the best possible score for this grid of numbers.)

   In the European version of this game, appropriately called Vankins Kilometer, the player can move the token either one square down, one square right, or one square left in each turn. However, to prevent infinite scores, the token cannot land on the same square more than once. Describe and



−1	7	−8	10	−5
−4	−9	**8**	−6	0
5	−2	**−6**	−6	7
−7	4	**7**	**−3**	−3
7	1	−6	**4**	−9

analyze an efficient algorithm to compute the maximum possible score for a game of Vankins Kilometer, given the $n \times n$ array of values as input.