CS 3510: Design & Analysis of Algorithms 10/16/2023

Practice Exam 3: Dynamic Programming

Abrahim Ladha

- This is the CS 3510 practice exam for exam 3.
- Topics include: Dynamic Programming on graphs, strings, and sequences.
- The number of questions on this practice exam are roughly what you could expect on the real exam.

1. A solution to a dynamic programming problem involves a table size of $\mathcal{O}(n^2)$, and each cell in the table takes $\mathcal{O}(n)$ time. The runtime for this algorithm will be $\mathcal{O}(n^3)$.

   [✓ **True**]

   ◯ False

2. For dynamic programming problems, we always create a table and our solution to the problem is always given by exactly one cell in the table.

   ◯ True

   [✓ **False**]

3. Consider the rod cutting problem:

   Input: A rod of length n inches and a table P of prices for lengths of the rod, that is, the money you make by selling a piece of the rod of that length.

   Output: The maximum profit which can be made by cutting the rod.

   While there could be many solutions, one of the following recurrence relations will yield a working DP algorithm. Select the correct recurrence relation.

   Assume the price of a length 0 rod is 0.

   - ○ $dp[i] = \max_{j \leq i}\{1 + dp[i-j]\}$
   - ○ $dp[i] = \max_{j \leq i}\{dp[i] + dp[i-j]\}$
   - [✓ $dp[i] = \max_{j \leq i}\{P_j + dp[i-j]\}$]
   - ○ $dp[i] = \max_{j \leq i}\{P_i + dp[i-j]\}$
   - ○ $dp[i] = \max_{j \leq i}\{dp[j] + dp[i-j]\}$
   - ○ $dp[i] = \max_{j \leq i}\{i + dp[i-j]\}$

4. Consider the largest sub-array sum problem.

   Input: A sequence of numbers $A = a_1, a_2, ..., a_n$.

   Output: The largest sum from a continuous sub-array.

   Select the correct recurrence relation and result.For simplicity, assume that $dp[i] = 0$ for $i \leq 0$. Also assume that the sum of an empty sub-array is 0.

   - ○ $dp[i] = \max(A_i + dp[i-1], 0), dp[n]$
   - ○ $dp[i] = \max(A_i + dp[i-2], dp[i-1]), dp[n]$
   - ○ $dp[i] = \max(A_i + dp[i-1], dp[i-1]), dp[n]$
   - ○ $dp[i] = \max(A_i + dp[i-2], dp[i-1]), \max_{1 \leq i \leq n}\{dp[i]\}$
   - [✓ $dp[i] = \max(A_i + dp[i-1], 0), \max_{1 \leq i \leq n}\{dp[i]\}$]
   - ○ $dp[i] = \max(A_i + dp[i-1], dp[i-1]), \max_{1 \leq i \leq n}\{dp[i]\}$

5. Suppose you're a frog,[1] crossing a river over a series of lilypads. Each lilypad can has a durability measuring how far you can jump to it; some lilypads are strong and can endure a big impact, but some are weak and if you jump to them, you'll sink into the pond! To formalize the problem, you're given an array of durabilities and you want to check whether you can get from the first lilypad to the final lilypad.

   Consider the following table. You can jump from lily pads in the following order to reach the end: 1, 2, 3, 5, 6, 7.

lilypad index	1,	2,	3,	4,	5,	6,	7
lilypad durability	1,	3,	1,	0,	2,	1,	1

   Select the correct recurrence relation. For simplicity, assume that $dp[i] = \text{False}$ for $i \leq 0$.

   - ○ $dp[i] = \bigvee_{1 \leq j \leq i}(dp[j] \wedge (A[i] > j))$
   - ○ $dp[i] = \bigvee_{1 \leq j \leq i}(dp[j] \wedge (A[j] > i))$
   - ○ $dp[i] = \bigvee_{1 \leq j \leq i}(dp[j] \wedge (A[i] > i - j - 1))$
   - ○ $dp[i] = \bigvee_{1 \leq j \leq i}(dp[j] \wedge (A[i] > j - 1))$
   - [✓ $dp[i] = \bigvee_{1 \leq j \leq i}(dp[j] \wedge (A[i] > i - j))$]
   - ○ $dp[i] = \bigvee \cdot_{1 \leq j \leq i}(dp[j] \wedge (A[i] > i - 1))$

6. Will's trying to host Thanksgiving dinner for the family, but he's never hosted it before. He plans to cook all the food, but with his busy lifestyle he can't cook all the Thanksgiving dinner food. Then, he has the family vote on what they would most like to eat, and he estimates how long it would take to cook the food. He's ok with only pleasing some portion of the family

   In other words, given a list of N dishes, the total votes v_i for each dish, and how long each dish takes to prepare t_i, figure out a set of dishes that can be prepared in the least amount of time and still meet or exceed the vote threshold V. Note that only one dish can be in progress at one time.

   Which of the following transformations to knapsack would result in a correct result; where in knapsack, C is the capacity, w_i is the weight and g_i is the value of the good.

   - [✓ $C = -V, w_i = -v_i, g_i = -t_i$]
   - ○ $C = -V, w_i = v_i, g_i = t_i$
   - ○ $C = -V, w_i = t_i, g_i = v_i$
   - ○ $C = V, w_i = -t_i, g_i = -v_i$
   - ○ $C = V, w_i = t_i, g_i = -v_i$
   - ○ $C = -V, w_i = -v_i, g_i = t_i$

[1] If you were a frog, would you have a good life?

7. Bob is giving two presentations this week and he has two ordered sets of note cards for each presentation. He's a bit of a clutz though and he accidentally dropped and mixed his two sets of cards.

   This is formalized with two strings A and B (which represent the note cards for a presentation) which get jumbled where the letters of A and B are interspersed while remaining in the same order. Give an algorithm to check whether string C is a jumbled version of A and B.

   For instance the words "ALGORITHM" and "AUTOMATA" could be jumbled into

   "ALGAUORITOMTHATA"

   (a) Define your table and the entries of your table.

   Solution: Create a two way table, T, indexed by i and j in the bounds n and m respectively where n and m are the lengths of strings A and B respectively.

   Each entry in the table is either valid of invalid. The entry is valid if $C[1..i+j]$ is a valid shuffle of $A[1..i]$ and $B[1..j]$.

   Any index into the table which is out of bounds will be treated as invalid.

   (b) Define your base cases and the recurrence relation. Justify your answers.

   Solution: The base case is $T[0,0] =$ valid because this is when all strings are empty, making the shuffle valid. At each recursive step we're going to look if we expanded one letter from A or one letter from B. If we've expanded one letter, we need to index into the table at one minus the current position with respect to the string and the current letter needs to equal the new letter in C.

   For A and B this is done with the equations $T[i-1,j] \wedge A[i] == C[i+j]$, and $T[i,j-1] \wedge B[j] == C[i+j]$. If either of these are true, then the shuffle at this step can be valid because it could be formed from previous shuffles.

$$T[i,j] = T[i-1,j] \wedge A[i] == C[i+j] \vee T[i,j-1] \wedge B[j] == C[i+j]$$

   (c) Find the runtime of your algorithm:

   Solution: $\mathcal{O}(mn)$

8. Consider two possible investments A and B; each year, an oracle tells you the return of the investments for the next year (you can assume that these are always correct). Each year you can place your money in either of the two assets; there's a caveat though that if you move your money from A to B or B to A (i.e. not keep your money in the same asset), you lose half your money.

   For instance the expected returns for investment A are 4x, 2x, 3x, 1x. The returns for investment B are 1x, 2x, 3x, 3x. The optimal strategy is to place your money in investment A in years 1, 2, and 3 (to get 24x) and to place money in investment 4 in year 4 to get 36x.

   (a) Define your table and the entries of your table.

   Solution: We'll define a two way table indexed by the investment opportunity and the year. This will then be a table of shape $(2, n)$ where n is the number of years. [2]

   (b) Define your base cases and recurrence. Justify your answer.

   Solution: Both investments start with the same unit amount of money. So our base cases will be $T[A, 0] = 1$ and $T[B, 0] = 1$.

   At each step we have two choices, we can either switch from the other investment and suffer the cost or stay with the same investment. This can be done with the following equation for A: $T[A, i] = \max(T[A, i-1]a_i, T[B, i-1]\frac{b_i}{2})$. The same for B: $T[B, i] = \max(T[B, i-1]b_i, T[A, i-1]\frac{a_i}{2})$

   (c) Find the runtime of your algorithm.

   Solution: $\mathcal{O}(n)$

[2]Note that since we're indexing by investment and their are a fixed number of investments, this takes linear space.

9. Given two strings **word1** and **word2** of length m and n respectively, design an algorithm to return the minimum number of operations required to convert **word1** to **word2**. You have the following three operations permitted on a word:

   1. Insert a character
   2. Delete a character
   3. Replace a character

   Example:
   Input: word1 = "horse", word2 = "ros"
   Output: 3
   Explanation:
   horse $\Longrightarrow$ rorse (replace 'h' with 'r')
   rorse $\Longrightarrow$ rose (remove 'r')
   rose $\Longrightarrow$ ros (remove 'e')

   (a) Define your table and the entries of your table.
   Solution: Define $T[i][j]$ of size $m \times n$ to be the minimum number of operations to convert word1[0..i) to word2[0..j).

   (b) Define your base cases and recurrence. Justify your answer.
   Solution: For the base case, to convert a string to an empty string, the mininum number of operations (deletions) is just the length of the string. So we have $T[i][0] = i$ and $T[0][j] = j$. For the recurrence, if word1[i - 1] == word2[j - 1], then there is no additional step to convert the two strings, so

   $$T[i][j] = T[i-1][j-1]$$

   . Otherwise,

   $$T[i][j] = \min(T[i-1][j], T[i][j-1], T[i-1][j-1]) + 1$$

   (c) Find the runtime of your algorithm.
   Solution: $\mathcal{O}(mn)$

10. Let's define a multiplication operation on three symbols a, b, and c according to the following table; thus $ab = b$, $ba = c$, and so on. Notice that the multiplication operation defined by the table is neither associative nor commutative. This means that a multiplication of several symbols could have multiple different results! For example, abc could be either $(ab)c = a$, or $a(bc) = b$.

    Find an efficient algorithm that examines a string of these symbols, say $bbbbac$, and decides whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is a. For example, on input $bbbbac$ your algorithm should return yes because $((b(bb))(ba))c = a$.

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

    (a) Define your table and the entries of your table.

    Solution: We'll define a three way table $T[i, j, \ell]$ where each entry represents true or false if it is possible to get letter ℓ with elements $s_i ... s_j$. [3]

[3] Again like the problem above, since there are a finite number of letters, we do not count this is a factor. The lecture solution illustrates how you can do this without that constant factor if you're curious.

(b) Define your base cases and recursive step:

Solution: If there is no string, we're not limited in the letters we can have from that, so we're going to say where all indices are equal we can make all strings.

$T[i, j, \ell] =$ true if $i == j$ and $\ell = s_i$

At each recursive step we check if we can make the specified letter with letters created by partitions of indices. This is essentially asking, can we make some letter with a partition at 1, a partition at 2, a partition at 3?

Then at each entry we have one of the three recurrences.

$$\begin{aligned}
T[i,j,a] = & \bigvee_{i<k<i-1} (T[i,k,a] \wedge T[k+1,j,c]) \\
& \bigvee_{i<k<i-1} (T[i,k,b] \wedge T[k+1,j,c]) \\
& \bigvee_{i<k<i-1} (T[i,k,c] \wedge T[k+1,j,a]) \\
T[i,j,b] = & \bigvee_{i<k<i-1} (T[i,k,a] \wedge T[k+1,j,a]) \\
& \bigvee_{i<k<i-1} (T[i,k,a] \wedge T[k+1,j,b]) \\
& \bigvee_{i<k<i-1} (T[i,k,b] \wedge T[k+1,j,b]) \\
T[i,j,c] = & \bigvee_{i<k<i-1} (T[i,k,b] \wedge T[k+1,j,a]) \\
& \bigvee_{i<k<i-1} (T[i,k,c] \wedge T[k+1,j,b]) \\
& \bigvee_{i<k<i-1} (T[i,k,c] \wedge T[k+1,j,c])
\end{aligned}$$

(c) Find the runtime of your algorithm.

Solution: $\mathcal{O}(n^2)$