CS170Hw7

Trustin Nguyen

October 21, 2024

Study Group

DP Solution Writing Guidelines

When writing your solutions to DP problems only, please follow the **4-part solution** template below (unless specified otherwise):

1. Define a function $f(\cdot)$ in words. Make sure to include how many parameters there are and what they mean, and tell us what inputs you feed into f to get the answer to your problem.
2. Write the recurrence relation for f, as well as the "base cases".
3. Prove that the recurrence correctly solves the problem. In almost all cases, you will want to use induction to prove the correctness of a DP algorithm.
4. Analyze the runtime and space complexity of your final DP algorithm. Note that the top-down and bottom-up approaches to DP have the same runtime complexity; however, bottom-up can potentially yield a better space complexity.

Egg Drop

You are given m identical eggs and an n story building. You need to figure out the highest floor $b \in \{0, 1, 2, \ldots, n\}$ that you can drop an egg from without breaking it. Each egg will never break when dropped from floor b or lower, and always breaks if dropped from floor $b + 1$ or higher. ($b = 0$ means the egg always breaks). Once an egg breaks, you cannot use it any more. However, if an egg does not break, you can reuse it.

Let $f(n, m)$ be the minimum number of egg drops that are needed to find b (regardless of the value of b)

(a) Find $f(1, m)$, $f(0, m)$, $f(n, 1)$, and $f(n, 0)$. Briefly explain your answers.

Answer. $f(1, m) = 1$. We are one a one-story building, so either $b = 0$ or $b = 1$. Then there are two cases: either the egg always breaks ($b = 0$) or it never breaks $b = 1$. So if we drop it once from floor 1, then if it does not break, we know that $b = 1$, otherwise, $b = 0$.

$f(0, m) = 0$. Since $b \in \{0, 1, \ldots, n\}$, we have $b \in \{0\}$. So $b = 0$.

$f(n, 1) = n$. If we have only one egg, then we need to go iteratively from floor 1 to n dropping it at every floor until it breaks. In the worst case, it will never break, so $b = n$. If it breaks at floor 1, we know that $b = 0$, otherwise, $b \geqslant 1$.

$f(n, 0) = \infty$. Since we have 0 eggs, we won't be able to find out the highest floor b that it breaks. So it is ∞.

(b) Consider dropping an egg at floor h when there are n floors and m eggs left. Then, it either breaks, or doesn't break. In either scenario, determine the minimum remaining number of egg drops that are need to find b in terms of $f(\cdot, \cdot)$, n, m, and/or h

Answer. If the egg breaks, then we know that it will break in one of the lower floors or not, so we test with our remaining $m - 1$ eggs: $f(h - 1, m - 1)$. If the egg does not break, we need to find out if it breaks at one of the higher floors. Notice that this is analogous to setting the floor h as floor 0 and now repeating the same process where $n \Rightarrow n - h$: $f(n - h, m)$

(c) Find a recurrence relation for $f(n, m)$.

Answer. Let the base case be $f(n, 1) = n$. Then we choose the worst case out of the egg cracking and the egg not cracking when dropping at the $n/2$ floor:

$$f(n, m) = \min_{h}(\max(f(n - h, m), f(h - 1, m - 1))) + 1$$

So this is the recurrence relation.

(d) If we want to use dynamic programming to compute $f(n, m)$ given n and m, in what order do we solve the subproblems?

Answer. We should have the look $i = 1 : m$ on the outside and $j = 1 : n$ as the inner loop. This is because if $i = 1$, this is just our base case $f(j, 1)$. And as we get more eggs, we see that

$$f(n, m) = \max(f(n/2, m), f(n/2 - 1, m - 1)) + 1$$

we can use the previous values of $n/2, m$ and $n/2 - 1, m - 1$.

(e) Based on your responses to previous parts, analyze the runtime complexity of your DP algorithm.

Answer. The runtime will be $O(mn)$ since we have two for loops.

(f) Analyze the space complexity of your DP algorithm.

Answer. The space complexity is also $O(mn)$ because we will be storing an $m \times n$ array over the combinations of m, n.

(g) Is it possible to modify your algorithm above to use less space? If so, describe your modification and re-analyze the space complexity. If not, briefly justify.

Answer. Yes it is possible. Since we only cut n by 2 each level, we do not need to store all values of n. We can denote the space complexity as:

$$S(n, m) = S(n//2, m) + S(n - n//2, m - 1) + 2$$

Each node in our tree will represent a constant amount of space being added to store the result. So $S(n//2, m)$ has some complexity that might be large, but as a node $n//2, m$ in our tree, it adds a complexity of 1 to store the result of the calculation of $f(n//2, m)$. Then to get the complexity, we need to count the number of nodes. The height of the tree is at worst $\log n$. So the number of nodes is at most $2^{\log n}$. So space complexity is $O(n)$.

Counting Targets

We call a sequence of n integers $x_1, \ldots, x_n$ *valid* if each x_i is in $\{1, \ldots, m\}$.

(a) Give a dynamic programming-based algorithm that takes in n, m and "target" T as input and outputs the number of distinct valid sequences such that $x_1 + \cdots + x_n = T$. Your algorithm should run in time $O(m^2n^2)$. Note that you can assume $T \leqslant mn$ since no valid sequences sum to more than mn.

Please provide a 4-part solution.

Answer. (Part I) We can define a function $f(n, m, T)$ such that n is the fixed sequence length of sequence $S = x_1, \ldots, x_n$, m is the value such that $x_i \in \{1, \ldots, m\}$, and T is a number. The function will output the number of unique sequences S such that $x_1 + \cdots + x_n = T$.

(Part II) There are several cases for the base case:

$$f(n, m, T) = \begin{cases} 0 & \text{if } T < n \\ 0 & \text{if } T > m \cdot n \\ 1 & \text{if } n = 1 \text{ and } T \in \{1, \ldots, m\} \end{cases}$$

The recurrence relation is:

$$f(n, m, T) = \sum_{i=1}^{m} f(n-1, m, T-i)$$

and the full equation:

$$f(n, m, T) = \begin{cases} 0 & \text{if } T < n \\ 0 & \text{if } T > m \cdot n \\ 1 & \text{if } n = 1 \text{ and } T \in \{1, \ldots, m\} \\ \sum_{i=1}^{m} f(n-1, m, T-i) & \text{otherwise} \end{cases}$$

(Part III) We will use induction on n. For the base case, if $n = 1$, we see that $S = x_1$, and we want to find the number of unique sequences such that $x_1 = T$. This is 1 if $T \in \{1, \ldots, m\}$ and 0 otherwise (which we get from the cases $T < n$, $T > m \cdot n$).

Now suppose that $n > 1$, we want to show that this holds. Then the number of unique sequences $S = x_1, \ldots, x_n$ such that

$$x_1 + \cdots + x_n = T$$

is the same number such that

$$x_1 + \cdots + x_{n-1} = T - x_n$$

So now iterating over every case, we get $f(n, m, T) = \sum_{x_n=1}^{m} f(n-1, m, T-x_n)$. Finally, each case is disjoint because we are selecting a different x_n value for each. So this gives the number of unique sequences on n, as expected.

(Part IV) To get the runtime, we need to describe the algorithm:

Start with a $dp = n \times T$ array of zeros. We will iterate by row. For the current row: r, for all columns at positions $c = 1, \ldots, T$, we calculate the number of ways such that a sequence of r numbers can add up to c. For the first row, we get:

	1	2	...	m	...	T
1	1	1	...	1	0	0

For the subsequent rows, we draw upon the value of the previous row. For example, in row 2, for each column $c = 1, \ldots, T$, we set the value of the second row at c to be $\sum_{i=1}^{m} dp[1, c - i]$. The solution will be the element in the last row, last column of the matrix: $dp[n, T]$.

Runtime Complexity: Since we iterate over the entire matrix, and for each element, we take a sum $\sum_{i=1}^{m} dp[r-1, c-i]$, this gives $m \cdot n \cdot T$ runtime complexity or $O(m^2n^2)$, since $T = mn$

Space Complexity: We used an $n \times T$ array, so the complexity here is $n \cdot nm = O(mn^2)$.

(b) (Extra Credit) Give an algorithm for the problem in part (a) that runs in time $O(mn^2)$.

Answer. Instead of doing m additions for each table entry, we can apply a sliding window along the previous row for the entries in the current row. This means we wont have duplicate additions. Then the runtime for each row is m to construct the window and 1 for each subsequent entry. There are mn entries per row so runtime per row is $m + nm = nm$. There are n rows, total runtime will be mn^2.

String Shuffling

Let x, y, and z be strings. We want to know if z can be obtained only from x and y by interleaving the characters from x and y such that the characters in x appear in order and the characters in y appear in order.

For example, if x = **prasad** and y = **SANJAM** then it is true for z = **praSANsadJAM**, but false for z = **prasadSANJAMpog** (extra characters), z = **prasSANJAad** (missing the final **M**), and z = **prasadASNJAM** (out of order). How can we answer this query efficiently? Your answer must be able to efficiently deal with strings with lots of overlap, such as x = **aaaaaaaaaab** and y = **aaaaaaaac**.

(a) Design an efficient algorithm to solve the above problem, briefly justify its correctness and analyze its runtime. You do *not* need to provide a space complexity analysis.

Answer. For the algorithm, construct an $|x| \times |y|$ array like shown:

	p	r	a	s	a	d	-
S	1	0	0	0	0	0	0
A	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0
M	0	0	0	0	0	0	0
-	0	0	0	0	0	0	0

Base Case: Ensure that $\text{len}(z) = \text{len}(x) + \text{len}(y)$. Start at the top left corner.

Inductive Case: We will then have an outer for loop that iterates over the elements of z after the first from $0 : \text{len}(z)$. Then for the i-th character of z, iterate along the i-th diagonal of the table. If the current entry is a 1, then proceed to check if z_i is equal to the row label. If it is, then set the entry below the current cell to be a 1. Also check if z_i is equal to the current column label. If it is, then set the entry in the cell to the right of the current cell to be a 1.

Correctness:

Base case: If z is of length 1, and $\text{len}(x) + \text{len}(y) = \text{len}(z)$, let $x = x_1, y = ''$ wlog our table is:

	x_1	-
-	1	0

Then we check that the current cell 1, 1 is a 1. In this case it is. Then on the column to the right, the value is set to $z_1 == x_1$

	x_1	-
-	1	$z_1 == x_1$

So this works.

Inductive case: Suppose the algorithm works for $n + m > \text{len}(z) > 1$. Let:

$$x = x_1 x_2 \dots x_n$$
$$y = y_1 y_2 \dots y_m$$

and $\text{len}(z) = n + m$, $z = z_1 z_2 \dots z_{n+m}$. Now remove z_{n+m} and denote the new string as $\bar{z}$. Define the notation:

$$\text{IL}_z(x, y) = \begin{cases} 1 & \text{if } x, y \text{ interleaves } z \\ 0 & \text{if otherwise} \end{cases}$$

We have two cases:

(a) Check that $\overline{x}, y$ interleaves $\overline{z}$. We know that the table correctly checks if $\overline{x}, y$ interleaves $\overline{z}$. So the bottom right of the table has the value $\text{IL}_{\overline{z}}(\overline{x}, y)$

(b) Check that $x, \overline{y}$ interleaves $\overline{z}$. We know that the table correctly checks if $x, \overline{y}$ interleaves $\overline{z}$. So the bottom right of the table has the value $\text{IL}_{\overline{z}}(x, \overline{y})$

Putting these two together, our table looks something like:

	x_1	x_2	$\dots$	x_n	$-$
y_1	?	?	$\dots$	?	?
y_2	?	?	$\dots$	?	?
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
y_m	?	?	$\dots$	?	$\text{IL}_{\overline{z}}(x, \overline{y})$
$-$	?	?	$\dots$	$\text{IL}_{\overline{z}}(\overline{x}, y)$	?

Now the bottom right corner will be the statement:

$$[\text{IL}_{\overline{z}}(\overline{x}, y) \wedge (z_{m+n} == x_n)] \vee [\text{IL}_{\overline{z}}(x, \overline{y})) \wedge (z_{n+m} == y_m)]$$

We see that this exactly captures if x, y interleaves z, because the bottom corner will check if either the last character of z is equal to x_n and the remaining characters of $\overline{x}, y$ interleave $z_1 \dots z_{m+n-1}$ or the other symmetric case where $z_{m+n} = y_m$.

Runtime: The runtime is $O(nm)$ because we look at each table entry once, and we do constant work per entry.

(b) Consider a bottom-up approach to our DP algorithm in part (a). Naively if we want to keep track of every solved sub-problem, this requires $O(|x||y|)$ space (double check to see if you understand why this is the case). How can we reduce the amount of space our algorithm uses?

Answer. Notice that we only need to keep track of which elements are 0 and which ones are 1, to run our condition. So instead, we can initialize a queue which looks like $[(0, 0)]$ at first. The queue holds the position of the table that is 1. Recall that in our base case, we set the top left to be 1.

Now we see that our queue stores tuples of the form (i, j). The current row label is then given by $y[j]$ and the current column label is given by $x[i]$. The current diagonal is given by $i + j$.

Then our check becomes

$$x[i] == z[i + j] \vee y[j] == z[i + j]$$

If $x[i] == z[i + j]$, we know to move our pointer one character over in x. So we would push $(i+1, j)$ into the queue. If $y[j] == z[i+j]$, we know to move the pointer over by 1 for our y string. So we would push $(i, j + 1)$ into the queue. Only push elements to the queue if the element to be pushed is not equal to the last element of the queue. This ensures we do not push duplicate entries. We iterate until our queue is empty and check that the last element is $(\text{len}(x), \text{len}(y))$. This denotes that the bottom right corner is a 1 in our DP table. We have at most two diagonals worth of items in the queue at a time, so the space complexity is $O(2 \cdot \max(x, y))$.

My Dog Ate My Homework

One morning, you wake up to realize that your dog ate some of your CS 170 homework paper, which is an $x \times y$ rectangular grid of squares. Some of the squares have holes chewed through them, and you cannot use paper that has a hole in it. You would like to cut the paper into pieces so as to separate all the tattered squares from all the clean, unbitten squares. You want to do this so that you can save as much as your work as possible.

Formally, the problem is as follows:

- **Input**: Dimensions of the paper $x \times y$ and an array $P[i, j]$ such that $P[i, j] = 1$ if and only if the ij^{th} square has holes bitten into it.
- **Goal**: Find the minimum number of cuts needed so that the $P[i, j]$ values of each piece are either all 0 or all 1.

Design a DP-based algorithm to find the smallest number of cuts needed to separate all the bitten parts out in $O(x^3y^3)$ time. For **extra credit**, try to optimize your algorithm to achieve a runtime of $O((x + y)x^2y^2)$

(a) Define your subproblem.

Answer. If we make a random cut, this will split the paper into two pieces. So the subproblem that could be solved is the minimum number of cuts to cut a sub rectangle of the homework into a valid partition. We can define $f(P)$ to be the minimum number of cuts to cut P into a valid partition.

(b) Write down the recurrence relation for your subproblems. A fully correct recurrence relation will always have the base cases specified.

Answer. First, we need to see if the cut is actually needed, or that the cells along the cut differ. So if we cut the paper at the i-th column for example, it is a necessary cut if $P[:, i-1]! = P[:, i+1]$. Then the recurrence relation becomes:

$$
\begin{aligned}
f(P) = \min\Big(& \\
& \min_{i=1}^{y-1}[(P[i-1,:]! = P[i+1,:]) \cdot (f(P[:i,:]) + f(P[i:,:]) + 1)], \\
& \min_{j=1}^{x-1}[(P[:,j-1]! = P[:,j+1]) \cdot (f(P[:,:j]) + f(P[j:,:]) + 1)] \\
\Big) &
\end{aligned}
$$

The base case for $f(P)$ is that $f(P) = 0$ when P is a 1×1 array.

(c) Describe the order in which we should solve the subproblems in your DP algorithm.

Answer. We need to solve the subproblems in order of increasing size of the array. So this means that if $x_1 \times y_1$ and $x_2 \times y_2$ were two arrays of different sizes, we would solve the one with the smaller value of $x_i + y_i$ first. We see that this is a valid ordering because if $x_1 \times y_1$ is a sub rectangle of the current problem we want to solve $x_2 \times y_2$, then $x_1 \leqslant x_2$ and $y_1 \leqslant y_2$, so we require $x_1 + y_1 \leqslant x_2 + y_2$.

(d) What is the runtime complexity of your DP algorithm? Provide a justification.

Answer. To get the runtime complexity, the algorithm needs to be described:

Algorithm: Each sub rectangle of our homework paper can be described by the position of the top left corner and the bottom right corner of the sub rectangle. There are a total of $(x-1)(y-1)$ possible positions for either corner. Create an

array called DP of size $xy \times xy$:

tl/br	0	1	...	xy
0	0	0	...	0
1	0	0	...	0
⋮	⋮	⋮	⋱	⋮
xy	0	0	...	0

where tl denotes the position of the top left corner and br denotes the position of the bottom right. Then tl is a tuple (t_i, t_j) where i is the row of the corner and j is the column. Assuming row-major order, we index into DP by $t_i \cdot y + t_j$ for the given tuple (t_i, t_j).

Recall that we iterate in order of increasing size of $x_i + y_i$. The size is given by $br - tl = (b_i - t_i, b_j - t_j)$, and this gives at most x^2y^2 iterations because each entry of DP corresponds to one instance of br, tl.

Then for each entry of the DP array, we iterate over all the horizontal cuts first and use our relation:

$$f(P) = (P[i-1,:]! = P[i+1,:]) \cdot (f(P[:i,:]) + f(P[i:,:]) + 1)$$

So we check that the cells along the cut are not all equal. We have at most y cells to check per cut over x horizontal cuts. So this is a runtime of xy. And doing the same over the vertical cuts is also xy runtime. Then indexing into the DP array is linear time. So runtime per DP cell is xy.

The total runtime is therefore $xy \cdot x^2y^2 = O(x^3y^3)$.

(e) What is the space complexity of your algorithm? Provide a justification.

Answer. From the algorithm description of the previous problem, the space complexity is $O(x^2y^2)$ for the DP array.