CSCI 270: Dynamic Programming

Overview

Dynamic Programming is useful in optimizing recursive algorithms. Recursive algorithms use the solutions to smaller versions of their problem in order to solve the entire problem. Often, recursive algorithms make the same calls multiple times, wasting resources. Dynamic Programming takes a problem that is recursive in nature and stores the solutions to smaller versions of the problem along the way, eliminating the waste of recalculating these values. In this handout, I'll provide the generic steps to create a DP solution. For each step, I'll give an example of how it would look for the Coin Problem from lecture.

Coin Problem

Given a set of k coin values $C = \{c_1 = 1, c_2, c_3 ... c_k\}$, and a target value T, what is the minimum number of coins needed to sum to the target value? What are the coins used in the optimal solution? Note that multiple coins of a given coin value c_j can be used. Write an algorithm to answer this question efficiently.

For example, given the coins $C = \{1, 5, 8\}$ and the target value $T = 15$, the optimal solution would take three coins, using three 5s.

Step 1: What is the Goal?

The first step in solving a DP problem is to state what the goal of your algorithm is. Writing this out explicitly can be helpful in coming up with a recursive function as well as knowing where your answer is stored in the end. I like to start by only considering the optimization question (what is the minimum number of coins), leaving the traceback question (what coins did we use in the solution) for later.

In the case of the Coin Problem, the goal is to return the minimum number of coins that sum to T.

Step 2: Define the Recursive Function

First we will create a recursive function for the problem at hand; we will translate this into an iterative solution in the next step. Defining your recursive function has two parts. First, you need to define your function's inputs and outputs explicitly in English. Second, you need to write pseudocode for your function.

In the case of the Coin Problem, we will define a function NumCoins(i) that returns the minimum number of coins needed to sum to i.

$\text{NumCoins}(i) = 1 + \min_{1 \le j \le k}\{\ \text{NumCoins}(i - c_j)\ \}$

This function is based on breaking the coins that optimally sum to i into two sets: one coin of any of the values c_j and the coins that optimally sum to what remains of the target value, $i - c_j$. So, the number of coins necessary to optimally sum to i is 1 + the number of coins in the second set. Since we already defined NumCoins to return the minimum number of coins needed to sum to the input value, we recursively call NumCoins($i - c_j$) to compute the number of coins in the second set. Note that If $i - c_j < 1$ the min subroutine will need to ignore option j because the value of the coin c_j is already greater than i, so we wouldn't use it in the sum.

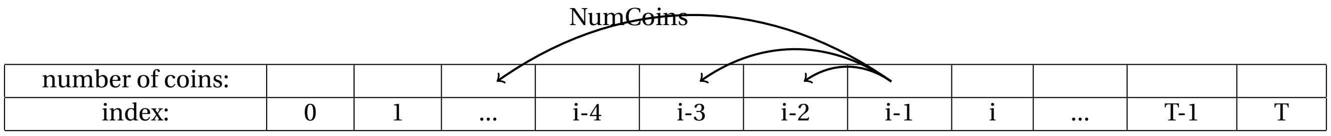
Step 3: Translate from Recursive to Iterative

Now that we've defined the recursive function, we want to make it iterative. This will consist of creating the array, deciding what direction to fill it in, and deciding the base case(s).

For clarity, we will change from a function NumCoins(i) that returns the minimum number of coins needed to sum to i to having an array NumCoins of size T such that NumCoins[i] stores the minimum number of coins needed to sum to i. We keep the same formula to fill in NumCoins: $\text{NumCoins}[i] = 1 + \min_{1 \le j \le k}\{\ \text{NumCoins}[i - c_j]\ \}$.

Next we need to consider what direction the array is filled in. When we fill in NumCoins[i], we need to have already calculated NumCoins[$i - c_j$]. So, we will fill in the array from left to right so that the dependencies are available.

I think that it's helpful to visualize the array. Since our recursive function only took one input, our array is 1D. For the sake of the diagram, consider the specific case where the set of possible coins is $C = \{1, 2, 4\}$. The arrows in the picture below represent which values we will look at when filling in NumCoins[i].



number of coins:											
index:	0	1	...	i-4	i-3	i-2	i-1	i	...	T-1	T

Next we need to define our base cases. In this Coin Problem, the base cases include all of the ways to use only one coin, NumCoins[1] = NumCoins[2] = NumCoins[4] = 1. Additionally, NumCoins[0] = 0.

NumCoins

number of coins:	0	1	1		1					
index:	0	1	2	3	4	...	i	...	T-1	T

Step 4: Define the Answer

Often, the answer to the original question we restated in Step 1 can be found in a specIfic cell in the array. Other times, we might have to do some extra work to find the answer. For example, the answer could be the minimum value in the array. One way to recognize where the answer is stored is to compare the original question from Step 1 to the English definition of our array from Step 3.

In the case of the Coin Problem, our original goal was to return the minimum number of coins that sum to T. Our recursive array NumCoins[i] stores the minimum number of coins needed to sum to i. Through a simple substitution of the target value For i, we find that our array stores the solution in cell T; NumCoins[T] stores the minimum number of coins needed to sum to T.

Step 5: Traceback

In some problems, you will also need to traceback the path that led you to the optimal answer. This usually consists of storing extra information along the way so that when you find your optimal solution you remember how you got there. Sometimes the array you created to solve the optimization will be enough to traceback the solution and you don't need to store extra information. Notice that the way you traceback the path can affect the overall runtime and space complexity of your solution.

In the case of the Coin Problem, we could store extra information along the way, keeping a 2D array that includes a row for each coin value c_j that storing how many of the coins we used to reach that sum. If NumCoins[$i - c_j$] was the minimum value when we calculated NumCoins[i], we would copy over the coins used to sum to $i - c_j$ and add one more coin to the number of c_j coins we used. By storing this extra information, we change the space complexity of the problem, requiring a 2D array of size $T \cdot k$.

It turns out that in this problem we don't actually need to store extra information. We can traceback the solution from the array we've already stored to answer the optimization question. Starting from NumCoins[T], we look back at the values in NumCoins[$T - c_j$] for all values of c_j. We know that we would have chosen the minimum of these values, so we record that we used one coin of value c_j where c_j was the coin that minimized the number of coins needed. We then proceed to examine NumCoins[$T - c_j$] in the same way, checking what previous value we would have used. This process takes $\mathcal{O}(T \cdot k)$ time since we examine k spots at each step backwards, and we could take at most T steps backwards.

Step 6: Write the Algorithm Explicitly

In order to make sure that you've completed all the necessary steps, you can write out your DP algorithm explicitly. This will include the base cases, directions for filling in the array, the formula to calculate the standard case, and where to find the answer. If the problem requires traceback, store any extra information you need and include the steps to traceback your path.

```
function CoinProblem($T$, $C = \{c_1 = 1, c_2, c_3 ... c_k\}$)
    NumCoins $\leftarrow$ empty array of size $T + 1$
    NumCoins[0] $\leftarrow$ 0                                         ▷ base cases
    for $j \leftarrow 1 \dots k$ do
        NumCoins[$c_j$] $\leftarrow$ 1
    end for

    for $i \leftarrow 0 \dots T$ do                                    ▷ fill in array
        if NumCoins[$i$] $\neq$ null then                              ▷ don't overwrite base cases
            continue
        min $\leftarrow \infty$                                        ▷ find the minimum previous case
        for $j \leftarrow 1 \dots k$ do
            if $i - c_j < 1$ then                                      ▷ coin value too large
                continue
            prev $\leftarrow$ NumCoins[$i - c_j$]
            if prev < min then
                min $\leftarrow$ prev
        end for
        NumCoins[$i$] $\leftarrow 1 +$ min                            ▷ our formula for the standard case
    end for

    $i \leftarrow T$                                                   ▷ backtrace
    CoinsUsed $\leftarrow$ array size $k$ filled with zeroes   ▷ CoinsUsed[$j$] is the number of coins of value $c_j$ used
    loop
        min $\leftarrow \infty$                                        ▷ find the minimum previous case
        minIndex $\leftarrow -1$
        for $j \leftarrow 1 \dots k$ do
            if $i - c_j < 1$ then
                continue
            prev $\leftarrow$ NumCoins[$i - c_j$]
            if prev < min then
                min $\leftarrow$ prev
                minIndex $\leftarrow j$
        end for
        CoinsUsed[minIndex] $\leftarrow$ CoinsUsed[minIndex] + 1
        $i \leftarrow i - j$
        if min = 1 then                                                ▷ back to base case
            CoinsUsed[i] $\leftarrow$ CoinsUsed[i] + 1
            break
    end loop

    return NumCoins[$T$], CoinsUsed                  ▷ return minimum number of coins and coins used
end function
```

Step 7: Analyze Runtime and Space Complexity

Finally, you need to analyze the efficiency of your algorithm using $\mathcal{O}$-notation. The runtime for a DP algorithm is (the time to calculate a single array cell) $\cdot$ (the size of the array) + (the time to find the answer in the array) + (the time to perform the traceback). The space complexity is the size of the array, which may include space to store extra information for the traceback step.

In the case of the Coin Problem, calculating an array cell takes $\mathcal{O}(k)$. The size of the array is $\mathcal{O}(T)$. As described earlier, the traceback step takes $\mathcal{O}(T \cdot K)$. So, the total runtime of our algorithm is $\mathcal{O}(T \cdot K)$. The space complexity is $\mathcal{O}(T)$.

Conclusion

I'm hopeful that these steps can serve as a scaffolding for how to approach a generic DP problem. Some of the steps that I've given are not strictly necessary to get full credit for a problem. For example, you would of course not lose points for leaving out Step 1, where we restated the problem. The minimum that you need to include is the inputs and outputs of your recursive function, a definition for your recursive function, the base cases, how to find the final answer, and the runtime analysis. Although these are the minimum requirements, if you're stuck on a problem, going through each step outlined above can be helpful.

Addendum: Recursive vs Iterative Implementation

The difference between a recursive implementation and an iterative implementation in DP is nuanced and all the more confusing because these terms are overloaded. All DP solutions are recursive in nature in the sense that they build upon optimal sub-solutions. The implementation itself can be either recursive or iterative in the programming sense of the words. If you follow the steps above, you first create a recursive function that answers the question at hand. Then, you translate this recursive function into an iterative implementation. The iterative implementation is conceptually identical to the recursive function; the formula to fill in a given spot in the storage array is the same formula the recursive function computes and returns. We translate the problem into an iterative implementation purely for clarity; by examining the way the array is filled in, you can ensure that your base cases provide the dependencies to fill in the rest of the array.

That being said, translating your recursive function to an iterative implementation is not necessary. If you prefer, you can call the recursive function top-down, storing the sub-solutions into a storage array as you go. The recursive function will first check the storage array to see if you've already made this calculation. If you have, the function will short-circuit and return the value you calculated and stored earlier instead of making further recursive calls. This recursive implementation provides the exact same benefit of DP as using an iterative implementation – it takes a problem that is recursive in nature and stores the solutions to smaller versions of the problem along the way, eliminating the waste of recalculating these values. Below is an example of what a recursive implementation to the Coin Problem would look like.

```
function CoinProblem($T$, $C = \{c_1 = 1, c_2, c_3 ... c_k\}$)
    StorageArray ← empty array of size $T + 1$        ▷ stores minimum number of coins to sum to index
    StorageArray[0] ← 0        ▷ base cases
    for $j \leftarrow 1 \dots k$ do
        StorageArray[$c_j$] ← 1
    end for

    return RecursiveHelper($T$)        ▷ top level recursive call will return the answer
end function

function RecursiveHelper($t$)        ▷ returns minimum number of coins to sum to $t$
    if $t < 0$ then        ▷ invalid; target value too small
        return NULL
    if StorageArray[$t$] ≠ null then        ▷ already performed this calculation; short circuit
        return StorageArray[$t$]

    min ← ∞        ▷ find the minimum previous case
    for $j \leftarrow 1 \dots k$ do
        prev ← RecursiveHelper($t - c_j$)
        if prev ≠ NULL and prev < min then
            min ← prev
    end for

    StorageArray[$t$] ← 1 + min        ▷ Store calculation in array
    return 1 + min
end function
```