CS 577: Introduction to Algorithms Fall 2020

How to Develop a Dynamic Programming Algorithm

Instructor: Dieter van Melkebeek TA: Ryan Moreno

This handout is meant to provide a basic structure to the problem solving and write-up steps for dynamic programming. Please note that these are not required patterns; they are merely suggestions. Any solution that is clear, correct, sufficiently efficient, and analyzed properly (correctness and complexity) will get full credit. However, if you are stuck on a problem, going through the steps outlined below can be helpful.

Suggested Write-Up Format

1. Clearly define the sub-problems. This is often the bottleneck in the algorithm development. See section 2 for tips.
2. Clearly state a recurrence for solving the sub-problems.
3. Explain the intuition and key ideas behind your recurrence. We should understand your basic approach without having to reverse-engineer your recurrence.
4. Analyze the correctness of your recurrence and base cases. This doesn't need to be a formal inductive proof, but needs to include the key ideas that would have been in a formal inductive step.
5. Describe how to organize the computation and obtain the final result (using either recursion & memoization or an iterative approach).
6. Analyze the time and space complexity of your algorithm.

Overview

Dynamic Programming is useful in making recursive algorithms with over-lapping sub-problems more efficient. Such recursive algorithms often make the same calls multiple times, wasting resources. Dynamic Programming takes a problem that is recursive in nature, minimizes the number of unique recursive calls, 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 generic steps to create a DP solution. For each step, I'll give an example of how it would look for the Coin Problem, stated below.

Coin Problem

Given a set of k of positive, integer coin values $C = \{c_1 = 1, c_2, c_3 ... c_k\}$, and a non-negative integer 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? Multiple coins of a given coin value c_j can be used. Write an algorithm to answer this question efficiently. Note that there is always a way to sum to T since a coin with value 1 is always provided.

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

1 Getting Intuition

State your goal

When getting intuition for a problem, I like to start with restating the goal of the algorithm. Writing this out explicitly can be helpful in coming up with a recurrence as well as understanding how your final answer is found. We will start by only considering the optimization question (what is the minimum number of coins), leaving retrieving the solution path (what coins did we use in the solution) for later. In the case of the optimization question, the goal is to return the minimum number of coins that sum to T.

First Choice/Question

Setting aside algorithms for a minute, as a human what is the first choice you would make if you were trying to answer the question at hand? This will often give intuition about what your sub-problems should be. The first decision I would make in solving an instance of the Coin Problem is to choose some first coin to be included in my pile of coins that will sum to T.

Remaining Problem

After making that first decision, what do you still need to know in order to answer the original "goal" problem (what is the minimum number of coins that sum to T)? Assuming that I had chosen a coin of value c_j as the first in my pile, the question that remains is the minimum number of coins necessary to sum to what remains of the target value, $T - c_j$.

2 Building a Recurrence

Defining the Sub-problems

Now that we have built intuition for how to tackle the problem, let's define the sub-problems. This step is essentially a problem specification, describing the sub-problem's inputs and outputs. Defining the sub-problem is one of the key steps, and it can take some time. Your goal is to come up with a sub-problem definition such that the number of distinct sub-problems is small. For example, in the Interval Scheduling problem we saw in class, we sort the meeting time slots by start time. This minimizes the amount of information necessary to represent the state after some number of previous choices (whether to attend a meeting or not). Through this re-organization, our state

only consists of the total value our previous choices accrued and the end time of the last meeting we chose to attend. This way, we only need sub-problems that consider the subsets of meetings starting after the end time of the last meeting we chose. In the Coin Problem, you might at first think that the state of the problem is the list of coins already chosen. However, we can reduce the size of this state by only remembering the total value of the coins we've chosen.

If you're having trouble defining your sub-problem's inputs, try thinking about it in another way. Thinking about what information needs to be passed into a recursive call solving the sub-problem is analogous to thinking about what information is necessary to represent the state of the problem after some number of decisions. Another tip if you get stuck is to start writing the recurrence relation until you realize that you need some piece of information, adding it as an input parameter.

In the case of the Coin Problem, the minimum information we need to describe the current state of the problem after some number of coin choices is the value our previous choices sum to. Then, our remaining problem is to find the minimum number of coins that sum to what's left of our target value. Note that the coin values never change, so we don't need to pass this information. We will define our sub-problem NumCoins(i) as the minimum number of coins necessary to sum to some integer i. What's left of the target sum is exactly the input necessary for the recursive call solving the sub-problem. By subtracting i from our original target value, you can also think of i as providing the current state of the problem after some number of decisions (our current sum).

Recurrence

Now that we have defined what the sub-problems *mean* we need to describe *how* to calculate their values. Our recurrence represents testing a decision about a small piece of the problem (*i.e.*, the "first choice") and then combining that with the optimal solution for the rest of the problem (*i.e.*, the "remaining problem"). Don't forget to include the base case(s) as well!

In the Coin Problem, our decision is picking one coin to include in the sum. After we pick one coin, we can depend on the recursive function to decide the optimal solution for the remaining target value ($i - c_j$). The function then chooses to include whichever "first" coin results in the minimal number of coins needed to sum to the remaining target value. Our base case is that it takes 0 coins to sum to a target value of 0. The recurrence is as follows:

$$\text{NumCoins}(i) = \begin{cases} \min_{\substack{1 \le j \le k, \\ i \ge c_j}} \{1 + \text{NumCoins(i} - \text{c}_\text{j})\}, & \text{if } i \geq 1 \\ 0, & \text{if } i = 0 \end{cases}$$

3 Correctness of Recurrence

At this point in the course, we assume that you understand how to write formal inductive proofs of correctness, and you are not expected to write one for your homework problems (or exam problems for that matter). However, you do need to explain the base cases and the key ideas of correctness that would have appeared in a formal proof's inductive step.

We only need to prove the correctness of our recurrence relation and base cases because the correctness of our recursive and/or iterative algorithms follow directly from the recurrence relation and base cases (as you will see in the next steps, the recursive and iterative algorithms are direct translations of the recurrence relation and base cases).

Below is an informal correctness argument that would suffice for your homework. A formal inductive proof of correctness is included in the appendix A.2.

The above recurrence is based on breaking the coins that optimally sum to i into two sets: the set containing one coin of any of the values c_j and another set that contains the coins that optimally sum to what remains of the target value, $i - c_j$. So, for a given first coin of value c_j, 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(i) 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. NumCoins(i) then returns the minimum number of coins necessary considering all possible values for the "first coin." Note that if $i < c_j$ the min subroutine will need to ignore option j because the coin value is already larger than the target value, so we wouldn't use it in the sum. Our base case is that it takes 0 coins to sum to a target value of 0. We don't need to consider the case that it's impossible to sum to the target value because the problem states that our set of coin values always includes a coin of value 1.

4 Algorithm

Next we need to describe how to execute the recurrence relation and base cases to achieve efficient running time. For details on the difference between a recursive approach using memoization and an iterative appoach, please see the appendix A.1.

Recursion & Memoization

The most straightforward way to execute the recurrence is to simply use recursion with memoization. Recursion sometimes has the added benefit of being faster than its iterative counterpart because it can skip sub-problems that it doesn't use to calculate the top-level recursive call (whereas in the iterative strategy we calculate all sub-problems). To use recursion and memoization, we develop a recursive algorithm identical to the recurrence relation and base cases we defined earlier. We then rely on memoization to store solutions to sub-problems each time a recursive call is made so that the same recursive call is never executed more than once.

In the case of the Coin Problem, it is enough to say that our recurrence can be executed recursively using memoization.

Iterative Approach

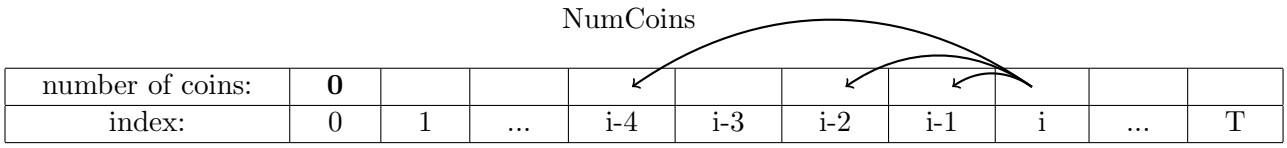
Translating our recurrence into an iterative approach can help to understand how our answer is calculated efficiently. Furthermore, an iterative approach makes retrieving the solution path simpler and sometimes yields better space complexity because we may only need to keep track of a small

portion of our storage array as we fill it in. The iterative approach will consist of creating an array to store solutions to our sub-problems and deciding what direction to fill the array in.

For clarity, we will change our notation from a recurrence NumCoins(i) that returns the minimum number of coins needed to sum to i into an array NumCoins of size $T + 1$ such that NumCoins[i] stores the minimum number of coins needed to sum to i. The recurrence we defined earlier be used to fill in this array.

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. The dependencies will always be available since our base case is $i = 0$.

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].



Finding the Answer

Often, the answer to our "goal" question can be found using a specific recursive call (in the recursive approach) or can be found in a specific cell in the array (in the iterative approach). Other times, we might have to do some extra work to find the answer. For example, the answer could be the minimum value in our storage array in the iterative approach. If the location of the answer is not clear, it can help to compare the original "goal" question to the definition of the sub-problems.

In the case of the Coin Problem, our original goal was to return the minimum number of coins that sum to T. Our recurrence NumCoins(i) is the minimum number of coins needed to sum to i. Through a simple substitution of the target value T for i, we can see that the recursive call to NumCoins(T) (in the recursive approach) or the cell at NumCoins[T] will return the solution to the "goal" question, the minimum number of coins needed to sum to T.

5 Retrieving the Solution Path

In some problems, you will be asked not only to solve the optimization question (e.g. the minimum number of coins needed), but also to retrieve the path that led to the optimal answer (which coins were used). Sometimes solution retrieval will be faster if you use the iterative approach and store

extra information along the way so that when you find your optimal solution you remember how you got there.

Other times, you will directly retrieve a solution without storing extra information. In fact, you can always retrieve a solution by tracing the recursive calls/array look-ups your algorithm made in reverse, seeing which choices it would have made at each step. We will see an example of this shortly. Notice that the way you retrieve the solution path can affect the overall run-time 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, storing how many of the coins we used to reach a given sum of i. 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 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 $(T + 1) \cdot k$.

However, we don't actually need to store the extra information. We can retrieve a solution path 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 our recurrence 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 total coins needed (breaking ties arbitrarily). We then proceed to examine NumCoins$[T - c_j]$ in the same way, checking what previous value we would have used. This process takes $O(T \cdot k)$ time since we examine k spots at each step backwards, and we could take at most T steps backwards.

It turns out we can optimize our retrieval time complexity without significant change to our space complexity by storing one extra value per cell in the array. In addition to storing the number of coins used so far, we will store the value of the last coin used. Starting from NumCoins$[T]$, we can record what coin value c_j was last used. We then examine the index $(T - c_j)$ to see the second to last coin used, etc. This process takes $O(T)$ time, which is an improvement from our last approach. However, since filling in the array took $O(T \cdot k)$ time, both both of these two retrieval approaches result in the same asymptotic run-time of the overall algorithm.

6 Time and Space Complexity Analysis

Recursion & Memoization Complexity

The time complexity for a recursive implementation can be calculated as (number of unique recursive calls) $\cdot$ (local time to compute a single recursive call) + (time for retrieving the solution path).

In the context of the Coin Problem, there are $T + 1$ unique inputs possible to our recurrence, so there are $O(T)$ unique recursive calls. The local time to compute the recurrence is $O(k)$ because we take the minimum over the k values corresponding to k different choices of the "first" coin. Retrieving the solution path takes $O(n \cdot k)$ if we trace the recursive calls the algorithm made in reverse, seeing which choices it would have made at each step. This results in a time complexity of

$O(T \cdot k)$

The space complexity is the size of the array used for memoization. In the case of the Coin Problem, this is $T + 1$ which is $O(T)$.

Iterative Complexity

The time complexity for an iterative implementation can be calculated as (size of the array) · (time to calculate a single array cell) + (time to find the answer in the array) + (time to retrieve the solution path). In the Coin Problem, calculating an array cell takes $O(k)$ (because the recurrence takes the minimum over k values, each of which take $O(1)$ time to look up in the array). The size of the array is $O(T)$. Finding the answer takes one look-up in the array, which is $O(1)$. As described earlier, the retrieval step takes $O(T)$ or $O(T \cdot k)$ depending on the implementation we use. So, the total time complexity is $O(T \cdot k)$.

The space complexity is the size of the array, which may include space to store extra information for retrieving the solution path. In the Coin Problem, the array is size $T+1$ (or $2(T+1)$ depending on how we implement the retrieval step), so the space complexity is $O(T)$.

Note that we can sometimes reduce the space complexity by only keeping the part of the array necessary to fill in the remaining cells. In the Coin Problem, we look back a maximum of $\max(C)$ cells where $\max(C)$ is the largest coin value in C, the set of coins from the problem statement. So, as we fill in the array, we only need to keep $\max(C)$ cells (rather than our original $(T+1)$ requirement). Depending on the coin denominations and value of T in a particular problem instance, this could save significant space. Our space complexity drops to $O(\max(C))$.

Appendix

A.1 Recursive vs Iterative Implementation

The difference between a recursive implementation (often called a "top-down" approach because we start with the top-level recursive call) and an iterative implementation (often called a "bottom-up" approach because we start by filling in the base cases and work our way up) 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 recurrence that answers the question at hand. Then, you translate this recurrence into either a recursive or iterative implementation. The iterative and recursive implementations are conceptually identical to the recurrence; in the iterative implementation, the formula to fill in a given spot in the storage array is the same formula the recurrence computes.

We often translate the problem into an iterative implementation 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. Also, the iterative implementation can be useful for saving space and for retrieiving the solution path.

That being said, translating your recursive function to an iterative implementation is not always necessary. If you prefer, you can call the recursive function top-down, relying on memoization to store the sub-solutions as you go. Furthermore, the recursive approach is sometimes faster than the iterative approach because it can skip recursive calls to sub-solutions that don't help calculate the top-level call. The recursion and memoization approach provides the 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.

Memoization is an abstract concept in which the results of recursive calls are stored so that they are not computed more than once. Memoization can be implemented in many ways, using a variety of data structures. In fact, you can rely on the compiler for memoization and do not need to concern yourself with how it is implemented. However, as an illustrative example, one way to imagine memoization is having an array to store sub-solutions. Each time our recursive algorithm returns, we first store the result for the specific input x in this array. Before making a recursive call with input y, we check the array to see if this specific recursive call has already been made, returning the result stored in the array if it has. Note that the number of array cells used is the number of unique inputs passed into the recursive algorithm. Below is pseudo-code for a recursive implementation of the Coin Problem using this style of memoization.

```
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 case
     return RecursiveHelper(T)                    ▷ top level recursive call will return the answer

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
        return StorageArray[t]

    min ← ∞                                       ▷ find the minimum previous case
    for j ← 1 ... k  do
        if c_j ≤ t then                           ▷ make sure coin value is not too large
            sub_soln ← RecursiveHelper(t − c_j)
            if subSoln ≠ NULL and subSoln < min then
                min ← subSoln

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

A.2 Inductive Proof of Correctness

We use induction to prove the correctness of our recurrence relation and base cases. The correctness of the recursive and iterative approaches follow immediately from the correctness of the recurrence and base cases because they are direct translations. Here is what an inductive proof of correctness would look like in the Coin Problem.

$$\text{NumCoins}(i) = \begin{cases} \min_{\substack{1 \le j \le k, \\ i \ge c_j}} \{1 + \text{NumCoins(i} - \text{c}_\text{j})\}, & \text{if } i \geq 1 \\ 0, & \text{if } i = 0 \end{cases}$$

Base case: $i = 0$**.** NumCoins(i) returns 0 in this case, which is correct because it takes 0 coins to sum to a target value of 0.

Inductive hypothesis: $0 \leq i \leq k$**.** We assume NumCoins(i) is correct, meaning that it returns the minimum number of coins needed to sum to i.

Inductive step: $i = k+1$**.** We are not in the base case, so we know that summing to $i = k+1$ will require at least one coin. Our recurrence considers all possible choices for this one coin c_j. By the IH, the call to NumCoins($i - c_j$) correctly computes the minimum number of coins needed to sum to $i - c_j$. Note that we can apply the inductive hypothesis because we have a call to $i' = i - c_j$ where $0 \leq i' \leq k$. This is true because $c_j > 0$, so $i - c_j < k + 1$ and we know from the constraints in the recurrence that $i \geq c_j$, so $i - c_j \geq 0$. So, for a given value of c_j, we know that $1 + \text{NumCoins}(i - c_j)$ properly computes the minimum value of coins necessary to sum to i assuming that there is at least one coin of value c_j used. Since we know that summing to i will require at least one coin, by taking the minimum (over all values of c_j that aren't themselves larger than the target sum) of the optimal answers assuming the fixed coin is c_j, our recurrence correctly calculates the minimum number of coins needed to sum to $i = k + 1$ with no constraints on the "first" coin.

By induction, our recurrence relation and base cases are correct for all target values $i \geq 0$.