Lab 6: Dynamic Programming
12:00 PM, Feb 28, 2018

Contents

1 Introduction 1
2 Pascal’s Triangle 1
3 Iron Man Loves Macarons! 3
4 Pascal’s Triangle: The Bottom-Up Approach 4
5 Macarons Part 2 4
6 Just For Fun: The Top-Down Approach 5

Objectives

By the end of this lab, you will know:

- what dynamic programming is

By the end of this lab, you will be able to:

- apply dynamic programming to implement faster solutions to problems with recursive solutions than a naive recursive implementation

Setup

Task: Before you begin, copy the support code to your javaproject/src/lab06/src directory:

```
cp /course/cs0180/src/lab06/src/* ~/course/cs0180/workspace/javaproject/src/lab06/src
```

1 Introduction

In today’s lab, you’ll be practicing dynamic programming. As you learned in lecture, dynamic programming is a technique that relies on storing intermediate solutions to subproblems in a table. There are two approaches to dynamic programming: top-down and bottom-up. The only difference between the two methods is how the values in this table are filled in.
These observations motivate this lab’s OOP design for solving dynamic programs, namely an abstract class that creates the table, extended by two classes, one which implements the fillTable method taking the top-down approach, and the other, bottom-up. As the table lives in the abstract class, so too can the methods that return the solutions to the problem, be it a particular value, like the $n$th Fibonacci number, or an optimal value coupled with an optimal path, as in an optimization problem like the optimal sequence of houses to visit while trick-or-treating.

In this lab, we will be asking you to solve a problem step-by-step, while providing you an example of each step using a problem called Pascal’s Triangle.

## 2 Pascal’s Triangle

Given non-negative integers $k \leq n$, we write \( \binom{n}{k} \) to denote the number of ways to choose $k$ elements from a set of $n$ distinct elements.

For example, suppose we have a set \{1, 2, 3\} and want to choose 2 elements from it. This means $n = 3$ and $k = 2$. We could choose \{1, 2\}, \{1, 3\}, or \{2, 3\}. Thus, the number of ways to choose 2 elements of 3, or \( \binom{3}{2} \), is 3.

You may be familiar with the following closed-form formula for computing \( \binom{n}{k} \):

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]

This formula requires computing factorials. Alternatively, one could use recurrence relation:

\[
\binom{n}{k} = \begin{cases} 
1, & \text{if } k = 0 \text{ or } k = n \\
\frac{n}{k} \binom{n-1}{k} + \frac{n-1}{k-1} \binom{n-1}{k-1}, & \text{otherwise}
\end{cases}
\] (1)

**Task:** Using pencil and paper (i.e., don’t implement anything), compute the value of \( \binom{n}{k} \) for all $n$ from 0 to 5 and $k$ from 0 to 5, $k \leq n$. You should have 21 values at the end.

**Hint:** If you arrange your solutions in a table with rows and columns corresponding to $n$ and $k$, Pascal’s triangle should emerge.

Now that you’ve seen how a table could be formed by hand, go ahead and open our coded setup to this problem (the source code you copied earlier). Open the AbsNChooseK and INChooseK files. AbsNChooseK is an abstract class that implements the INChooseK interface and contains everything that is needed to solve \( \binom{n}{k} \) dynamically, namely:

1. A table in which to store intermediate solutions;
2. An abstract method fillTable, to be overridden in the concrete classes that inherit from AbsNChooseK (so we can fill the table both in a top-down and bottom-up way);
3. The method choose(int n, int k), declared in the interface INChooseK, which returns the value of \( \binom{n}{k} \) by retrieving a value from the filled-in table.

Further, note that the AbsNChooseK constructor, which sets up our problem:
1. Takes as input the maximum set size $n$ that this class will support.

2. Checks that $n$ is valid (non-negative) and creates a 2D table of that size.

3. Calls two methods:
   
   (a) `initTable`, defined in `AbsNChooseK`, which initializes the values in the table (this is how the base case is set), and
   
   (b) `fillTable`, so the table is ready for any lookups (calls to the choose method).

**Task:** On a piece of paper or a text editor, explain what the purpose of the `AbsNChooseK` class is—why did we use this abstract class, rather than simply use the interface and two concrete classes for the two dynamic programming approaches?

---

**3 Iron Man Loves Macarons!**

Here’s a depiction of a store-counter of macarons:

<table>
<thead>
<tr>
<th>Chocolate</th>
<th>Strawberry</th>
<th>Vanilla</th>
<th>Pistachio</th>
<th>Raspberry</th>
</tr>
</thead>
</table>

Tony Stark wants to purchase a number of macarons, since they’re easy to store inside his Iron Man suit when he’s got work to do, but the shop owner has a particular rule: he may not purchase two adjacent macarons. For example, in the above arrangement, he cannot purchase both strawberry and vanilla macarons.

Each flavor of macaron has a non-negative tastiness value based on how tasty that flavor is:

- Flavor 1: Strawberry (value 35),
- Flavor 2: Salted Caramel (value 90),
- Flavor 3: Chocolate (value 40),
- ... 
- Flavor $n$: Lavender (value 15).

Your job is to help Tony figure out the best set of macarons to purchase— that is, the set of macarons with the maximum sum of their tasty values.

So, how will you do this?

**Greedy:** You might try a *greedy* approach. You start by choosing the item with the highest value. This rules out the adjacent items (i.e., if you pick item 7, you cannot subsequently pick items 6 or
8). Next, you pick the item of maximal value among those remaining. You repeat this until you cannot add any more items.

**Task:** Using pencil and paper, come up with a simple example showing that the above greedy algorithm is not optimal. That is, show an example where the greedy approach gives you a solution that doesn't have the maximum sum of tasty values.

**Task:** Write a recurrence relation that finds the optimal value of a subsequence of non-adjacent items given a sequence of \( n \) items with values \( v_1, \ldots, v_n \).

**Note:** A macaron array of length 0 is valid input!

**Hint:** First consider how to find the optimal value out of a sequence of 0 items, then 1 item, then 2 items, and so on.

**Task:** What would be the run time of a naive, recursive implementation of your recurrence relation? State your answer in \( \mathcal{O} \) notation. Give a brief justification; a formal proof is not necessary.

**Task:** Create the `AbsMacaron` class, which implements the `IMacaron` interface we have provided. We recommend you look back at the Pascal’s Triangle section as a reference to make sure you include all the necessary methods and fields in `AbsMacaron`! `AbsMacaron` should handle all functionality that is shared by top-down and bottom-up dynamic programming solutions.

**Note:** At this time, only write the constructor and the `initTable` method, and create whatever data structures you need. Don’t forget to declare the `fillTable` method! You should also override the `IMacaron` methods, but return default values for now (i.e. -1 or `null`).

You’ve reached a checkpoint! Please call over a lab TA to review your work.

Now that you have the recurrence relation and OOP class structure ready, all you need is to create the concrete classes that actually fill out the table in the two approaches. We’ll first show you how to do this with our Pascal’s Triangle example.

### 4 Pascal’s Triangle: The Bottom-Up Approach

Open the source file called `BUNChooseK` (short for bottom-up NChooseK). This class extends `AbsNChooseK` and overrides the `fillTable` method to fill the table values in a bottom-up fashion. Observe that the 2D table is being filled out from the top to bottom, left to right, using the recurrence relation. By the end of this function, all the values in the table that we need will be filled in; that is, all values where \( k \leq n \) and \( n, k \leq \) the maximum \( n \) specified by the constructor are filled in.

**Note:** We have also created a constructor for this bottom-up class. Note that it simply calls the abstract class’ constructor, as the abstract class contained everything needed for both dynamic programming approaches other than the specifics of the way `fillTable` works.

We have also created a tester class, called `NChooseKTest`; feel free to take a look at that for ideas on testing your macaron solution once you are ready.

**Note:** Notice how we use `OptionalInt` rather than regular `ints` to fill in our table. This is to avoid accidentally using uninitialized values without realizing it! You should do the same when writing your solutions for the macaron problem.
5 Macarons Part 2

Now that you’ve seen an example of a concrete bottom-up class, you’re ready to help Tony with optimizing his macaron experience!

**Task:** Using the recurrence relation from earlier, implement the **BU**Macaron class. This class should include only what is specific to a bottom-up implementation.

**Task:** Inside **Abs**Macaron, implement the optimizeValue method (originally defined in **IMacaron**), which returns the optimal value that can be achieved, given a sequence of tasty values.

**Hint:** Take a look at **AbsN**ChooseK’s choose method! This should be a very short method.

**Note:** As always, don’t forget to test your functions before you move on!

**Note:** The constructor should take an array of ints, representing the tastiness values of the macarons. However, be sure your dp solution fills out an array of OptionalInts!

| You’ve reached a checkpoint! Please call over a lab TA to review your work. |

Now, Tony can figure out the maximum tastiness value he can get from a sequence of macarons! However, that doesn’t help him much in figuring out which macarons he should buy.

**Task:** In the **Abs**Macaron class, implement the optimizeSubSeq method defined in **IMacaron**, which returns an optimal subsequence of the given sequence, represented as an array of booleans. Don’t forget to test!

For example, given the sequence ⟨40, 50, 60, 30⟩, the optimal subsequence is ⟨true, false, true, false⟩.

**Note:** The optimal subsequence need not be unique. Your method can return any subsequence that sums to the optimal sum. All are equally valid solutions to this problem.

**Hint:** Once you’ve filled in your table, you can use it to backtrack and ask, “which macaron did I pick before this?” starting at the last macaron you “picked” and working your way to the start using algebra.

| You’ve reached a checkpoint! Please call over a lab TA to review your work. |

6 Just For Fun: The Top-Down Approach

Now that we’ve seen one approach for dynamic programming, let’s explore the other. Open the source file called **TD**NChooseK (short for top-down NChooseK). Like **BUN**ChooseK, this class extends **AbsN**ChooseK and overrides the fillTable method, but it fills the table using a top-down method. It tries to fill the entry where \( n, k \) = the maximum \( n \) specified the constructor, and fills in the other entries of the table as it needs them to calculate this value.

**Note:** Once again, the constructor for this approach just calls the constructor for the abstract class.

**Task:** Implement the **TD**Macaron class. Like **BU**Macaron, this should include only what is specific to the top-down implementation.

**Hint:** You should only have to implement one method, and maybe a helper. The rest should come from **Abs**Macaron.
Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)