1 Introduction

In today’s lab, you’ll be practicing dynamic programming (DP). As you learned in lecture, DP is a technique that relies on storing intermediate solutions to subproblems in a table. In this lab, we’ll use bottom-up DP.
In this lab, you will be solving a new kind of problem using DP. Fibonacci just had to compute a single answer (the $i^{th}$ value in the sequence). DP saved our intermediate values, and all the intermediate values contributed to the final solution. For today’s problem, we are instead going to look for an optimal solution out of many possible solutions. Furthermore, the solution we return will contain two pieces of information: a specific value that the solution maximizes, and the set of choices needed to get the maximal solution.

Such problems, featuring optimization and path-finding, are common in computer science. Fortunately, they share a similar architecture to what we had for Fibonacci, so we really will just be extending on what we did in lecture.

2 The Problem: Brock Loves Macarons!

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

| Chocolate | Strawberry | Vanilla | Pistachio | Raspberry |

Brock wants to purchase a number of macarons, since they’re easy to carry around 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 positive (non-negative and nonzero) 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 Brock figure out the best set of macarons to purchase— that is, the set of macarons with the maximum sum of their tasty values while following the shop owner’s rule. The first thing to find is the maximum tastiness sum you can achieve, given the problem constraints and the array of tastiness values. Later, we’ll add code to also compute which specific macarons he should purchase.

So, how will you do this?

We’ll take you through this step-by-step, but for reference, we have also provided you with a fully worked example solution to a different problem called Maximum Subarray Sum.

While there are no required tasks in the Maximum Subarray Sum example, you may find it helpful to spend some time looking over the max subarray solution as you work out the corresponding steps for the Macaron problem.
3 The Best Tastiness

**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 the item in index 7, you cannot subsequently pick items in indices 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 this 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.

Since being greedy won’t work, we’ll have to systematically search for a solution instead. How might that work? We won’t do anything fancy here. We’re simply going to go down the sequence of tastiness values and ask ourselves whether we are better off keeping or skipping the current macaron.

Let’s see the idea with an example. Assume that we have five macaron options with the following tastiness values: [3, 10, 12, 16, 4].

**Task:** Which combinations of macarons could you select given the shop owner’s non-adjacency rule? Which combination would give you the maximal tastiness score?

Now, we have to figure out how to implement that search systematically. Consider the Raspberry macaron (from the example figure): if we select it, we must skip Pistachio and instead select from the first three macarons. If we skip Raspberry, then we can look for our best option among the first four macarons. So if we knew the maximal tastiness within the first three and first four macarons, we can compare the maximum from the first four to the max from the first three plus Raspberry to decide whether to skip the Raspberry macaron.

The src folder for lab provides a naive recursive solution to maximizing tastiness, following the approach outlined above. Go ahead and copy over our source code, which contains this recursive solution, among other things (note the -r option, which copies subfolders):

```
 cp -r /course/cs0180/src/lab05/src/* ~/course/cs0180/workspace/javaproject/src/lab05/src
```

Open up RecMacaron.java and take a look at the optimize function, which is the recursive solution to this problem. Things to notice about the code:

- The tastiness values are provided in an array, as opposed to a list as you are more used to with recursive solutions.

- This means that the input to the recursive function is an index into the array, not sublists as you are more used to. We’re still using recursion, but we’re effectively passing a position into a data structure as opposed to the entire (sub) data structure on the recursive calls.

Study RecMacaron.java to make sure you understand how the macaron computation works before we migrate this to DP. Ask a TA if you have questions.
4 Migrating to DP

4.1 Finding a Recurrence

We know that in DP, we need to fill out a table of values representing intermediate solutions to our problem. To do that, we have to figure out what the intermediate problems (also known as subproblems) are.

Task: Based on the code in RecMacaron.java, write down what you think the subproblems are. That is, if we wanted to create a table T to store intermediate results in DP, what would an arbitrary spot T[i] represent in the problem? Next, write down the dimensions of your table, assuming the input array of macaron tastiness values is an array of length n.

Hint: If we were computing the n\textsuperscript{th} Fibonacci using DP, the answers to the above problem would be:

- Each spot T[i] in the table would contain the i+1\textsuperscript{th} Fibonacci number, because of zero-indexing, so the 0\textsuperscript{th} spot is the first Fibonacci number, etc.
- The table would have one dimension of length n.

Hint: Zero-indexing played a part in the Fibonacci example above, but will not play a part in your answers!

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

Task: Now that we know the meaning of an individual entry in our table, we write down a recurrence relation to capture how subproblems contribute to solving larger problems. You saw recurrence relations in the fall as part of doing big-O analysis, but now you will see recurrence relations being used to capture computations other than runtime.

In the DP case, recurrence relations instead summarize the recursive computation at the heart of the problem. They include any base cases, as well as the heart of the inductive case. If we were doing this task for Fibonacci, our recurrence relation would look like this:

\[ T[i] = \begin{cases} 
0, & \text{if } i = 0 \\
1, & \text{if } i = 1 \\
T[i - 2] + T[i - 1], & \text{otherwise}
\end{cases} \]

Using notation as above, write the recurrence relation for our macaron problem. It will indicate how to fill out the i\textsuperscript{th} index of your table.

Hint: It might help to revisit our recursive solution, as it already embodies the recurrence relation you’ll need! As you get more experience with these problems, you start by writing the recurrence relation directly, rather than first working through a recursive solution.

Task: Now, assuming you had filled out the table, where in it would one find the final answer to the problem? Consider the meaning of the cell contents, and also consider the case where the input array size is 0, meaning there are no macarons. Don’t forget that in an array of length n, the last entry is in spot n − 1!
Hint: For Fibonacci, the final answer would be located in $T[n-1]$, the last entry in our array.

Task: What would be the run time of a naive, recursive implementation of your recurrence relation? State your answer in big-$O$ notation. Give a brief justification; a formal proof is not necessary.

Task: What would be the run time of your dynamic programming approach? Again, no formal proof is necessary.

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

Now that we have our recurrence relation and a better sense of how we’ll use the table, it’s time to start coding! First, we’ll have you set up the class (leaving most of the methods returning dummy answers). After a TA checks your setup, you’ll go on to implement the actual methods.

4.2 Setting up the class for your code

Task: We have provided a stencil class for you, BUMacaron (for bottom-up macaron). At this time, go ahead and copy that over:

```
cp /course/cs0180/sol/lab05/sol/* ~/course/cs0180/workspace/javaproject/sol/lab05/sol
```

This stencil contains the following:

- a constructor that takes in the array of (int) macaron values. The constructor should be assigning values to the following fields:
  - `int[] itemValues` - The array of macaron tastiness values that was passed in/given.
  - `OptionalInt[] optimalValues` - The array of optimal values given the problem constraint to fill in. This is of type `OptionalInt` because you do not want to initialize an array to all `null` values. After this is initialized, `initTable()` and `fillTable()` should be called to actually fill in the values.
- `public void initTable()`, a method which should fill in the base cases in `optimalValues` and sets the rest of the entries in `optimalValues` equal to `OptionalInt.empty()`.
- `public void fillTable()`, a method which will use your recurrence relation to fill out `optimalValues`. **Do not implement this yet!**
- `public int optimizeValue`, a method that uses the filled table to return the maximum sum macaron tastiness value. We’ll fill this out later.

Task: Fill out the constructor code, as well as `initTable()`, per the specifications above.

Hint: To initialize an array in Java, you can do something like this:

```java
OptionalInt[] myArray = new OptionalInt[values.length];
```
You’ve reached a checkpoint! Please call over a lab TA to review your work.

4.3 Implementing the DP Algorithm

Now that you have the recurrence relation and class structure ready, all you need is to create and fill in the methods that would solve the problem in a bottom-up way. We recommend at this time that you take a look at our example solution for the maximal subarray problem (located in the src code for this lab, in BUMaxSubarray.java)!

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

Task: Inside BUMacaron, implement the following methods:

- fillTable() - A method that fills out the optimalValues array.
- optimizeValue() - A method that given a filled table should return the optimal value as an int. Don’t forget to account for the case where the input table of macarons was of length 0!

Hint: To find the maximum of two values, you can do Math.max(num1, num2).

Hint: optimizeValue() shouldn’t be performing any computations, but rather, accessing the table which has already been filled out!

Note: When you copied the solution code earlier, you also copied our test file! It tests optimizeValue, and the tests for optimizeSubSeq have been commented for now; you’ll only need them later.

Task: Using our test file, test out your implementation!

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

5 Just for Fun: Which Macarons?

Now, Brock 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.

As your current implementation goes along, it does figure out which macarons to compute: whenever you check whether to skip a macaron or to use it and skip the next one, your code makes a choice about which macaron to purchase. Our current implementation, however, doesn’t store that information anywhere. We therefore need to augment the solution to also store the chosen macarons.

Task: We’re going to use a second array, which stores which macarons to buy for the optimal tastiness value. The array will contain List<Integer>, representing all of the macaron indices that you should include in the optimal solution. Go ahead and make another class variable, optimalSeq, of type List<Integer>[], to accomplish this.

Hint: You can declare it like this:

```java
protected List<Integer>[] optimalSeq;
```

Task: We also need to initialize the base cases and default values for this table, inside our initTable method. Go ahead and add to that code, adding in the base cases (for just the 0th macaron, which
macarons should you pick? What about when looking at just the first two macarons?). Here, the default value in any non-base case entry should simply be an empty list.

**Hint:** Don’t forget to initialize your new array! You can initialize it in the constructor like this:

```java
this.optimalSeq = new LinkedList[inputValues.length];
```

You will also have to add one line of code directly above your constructor, because you can’t tell an array what type of LinkedList it contains:

```java
@SuppressWarnings("unchecked")
```

Now, we’ll need to go back to `fillTable` and also update entries in `optimalSeq`. Similar to your prior recurrence relation, you’ll need to make the decision: is it better to include the macaron I’m currently on (add the current index to a previous stored solution) or not (use a different previous stored solution)?

**Task:** Upgrade your `fillTable()` method so that in addition to filling in the `optimalValues` array, it also fills in the `optimalSeq` array - an array of LinkedLists of the current optimal selection at each index.

**Question:** If the optimal sequence doesn’t change from index `i` to index `j`, can you just set `optimalSeq[i] = optimalSeq[j]` and keep adding to the same list? Or do you need to set `optimalSeq[i]` to a copy of `optimalSeq[j]`? Think about the consequences of each decision: do both give the correct answer? Are there other tradeoffs?

**Hint:** If you want to copy the list, write `optimalSeq[i].addAll(optimalSeq[j])`.

**Task:** In the `BUMacaron` class, implement the `optimizeSubSeq` method which returns an optimal subsequence of the given sequence, represented as an array of booleans.

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:** You should be using the `optimalSeq` array that you have filled in. Check out `RecMacaron.java`’s `optimizeSubSeq` method to see how we did this!

**Hint:** By default, all entries in a boolean array in Java are initialized to false.

**Task:** Test your implementation by uncommenting all the commented tests in `MacaronTest.java`!

### 6 Stepping Back

Before you go, step back and make sure you understand the steps and pieces that went into writing this DP-based solution:

- Think through cases of the problem statement to help you identify possible subproblems that you can combine into larger solutions
- Write a recurrence relation that summarizes the computation for the problem
- Figure out the dimension of your table and what each cell of the table represents relative to the overall problem.
• In the class that will contain your DP implementation, set up a field for the table of answers to each subproblem. If you also need to return the path of decisions that got to the subproblem, set up another table to hold the paths.

• Implement methods to initialize the table, to populate it through DP, and to extract the optimal value and/or sequence of choices, as needed in the problem.

Make sure you understand what the pieces of your **BUMacaron** class do towards each of these goals (you’ll have to do this for yourself on the homework). If you want to see additional examples, study the Maximum Subarray Sum example we gave you. Discuss it with your partner and/or the TAs until you are comfortable with what is going on. There’s no task to check off here, just warnings to use the chance now to understand what’s going on before the DP homework is released on Friday.

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)