HW4: Divide & Conquer (Coding)

Objectives:
To practice designing and implementing algorithms using the divide and conquer approach.

Due date: Friday, February 2nd by 11:00pm.

Materials: starter code (hw4-divide-and-conquer.tar.gz)

What to turn in:

- Your final version of `SidewaysTrend.java` and any other Java files that you create or change.
- A PDF file containing your answers to the questions from problems 2 (and optionally 4) below.

How to submit:
Use the canvas page (https://canvas.uw.edu/courses/1124392/assignments/4077927) to submit your solution.

Overview

In this assignment, you will design and implement a divide and conquer algorithm to find "sideways trends" in price data. Rather than looking for the range of days where the price increases the most, as we did in class (with the maximum single-sell profit problem), you are instead looking for ranges of days where the prices do not change very much.

In particular, given an array of prices, called `prices`, we will say that the index range `[i .. j]` defines a **sideways trend of percent `p`** if no two prices in that range differ by more than `p` percent. That is, for any two indices, `s` and `t`, lying in that range, neither is more than `p` percent larger than the other:

```
prices[s] <= (1 + p/100) prices[t]
```

(Note that this is true for all prices in the range if and only if it is true when `s` is the index of the largest price and `t` is the index of the smallest price in that range.)

The problem you will solve in the assignment is to find the **longest** sideways trend in the given data, i.e., the range of indices `[i .. j]` defining a sideways trend of percent `p` with `j - i` *as large as possible*.

Provided Code

Most of the code needed for the program we are making in this assignment is provided for you. In particular, you are already given code for parsing the data files, parsing the command-line arguments, and for keeping track of information about index ranges. You will only need to fill in the bodies of a few methods that implement the core logic of the algorithms.

You will need to familiarize yourself with two files in the provided code: `SidewaysTrend.java` and `Range.java`. While the other code will be used by your program, you will not need to use it directly.

`SidewaysTrend.java` contains the entry point for the program. The `main` method parses the command-line arguments and loads the data from the passed-in data file. `SidewaysTrend` also contains the three method bodies that **you will fill in** during the assignment. (Search for "TODO" in the file to find them.)

`Range.java` is a helper class that represents a range of indices. It also stores information about the prices over that range so that it can quickly check whether the range defines a sideways trend. `Range` provides the following methods that you will need to use in your solution:

- **`static`** `Range` **`fromOneIndex`**`(int index, List<Integer> prices)`
  Returns a new `Range` that represents the index range `[index .. index]`, i.e., a range containing just the index passed in.
- `Range` **`concat`**`(Range other)`
  Returns a `Range` that represents the concatenation of this range with the one passed as the argument `other`. Note that `other` **must** begin immediately after this range ends — no gaps are allowed. Otherwise, the method will throw an exception.
- `boolean` **`percentChangeAtMost`**`(double pctChanged)`
  Checks whether this range defines a sideways trend of percent `pctChanged`.

Note that all of these operations run in O(1) time.

Problems

1. Start by implementing the method `findLongestSidewaysTrendNaive` in the starter code. This method is passed in the list of `prices` and the limit, `maxPctChange`, on how much prices can change within a sideways trend (what we called "p" above).

   Your implemention should find the longest sideways trend using a naive, brute-force algorithm, which potentially tries every pair of indexes, `i` and `j`, with `i <= j`, to see if they define a sideways trend. Rather than implementing these calculations from scratch, you should make use of the provided `Range` class. In particular, you can build up a `Range` object for any range of indices by constructing `Range`s for each individual index using `Range.forOneIndex` and calling `concat` to concatenate adjacent ranges into longer ones. Once you have the complete range, the method `percentChangeAtMost` will tell you if it defines a sideways trend. Your implementation should run in $O(n^2)$ time on price data of length n.

   You can invoke your method by running the `ant run-naive` command from the `build.xml` file or by running the program with an argument of "`--naive`" (in addition to the argument with the prices data file).

2. In this part, we will design a divide and conquer algorithm for this problem.

   At a high level, the algorithm will operate as follows. Divide the array of prices into the first and second halves and solve the problem recursively on each half. That will give us the longest sideways trends that stay entirely in one half of the data, leaving us to consider only ranges starting in the first half and ending in the second half (i.e., those that cross the mid-way point) in our combine step. Once we have found the longest sideways trend of all three types, we return the longest of those three as our solution.

a. What is the base case for this recursion? How do you solve the problem in that case in O(1) time?

b. As described in the next section, our combine step will use a "two finger" algorithm with each finger pointing to a range in one of the two lists and maintaining the invariant that the range under the right finger points to the *longest one* that can be concatenated with the range under the left finger to produce a sideways trend.

   On the next iteration, when we advance the left finger to the next range in the left list, the right finger will always to move in one particular direction (or not move at all). What direction is that (to lower indexes or higher ones), and why do we know this is the case?

c. Why is the total number of finger moves O(n)? Furthermore, if we implement this with the two fingers stored as integer indexes in the code, why does all the work of moving the indexes, concatenating ranges, and checking whether they define sideways trends take only O(n) time?

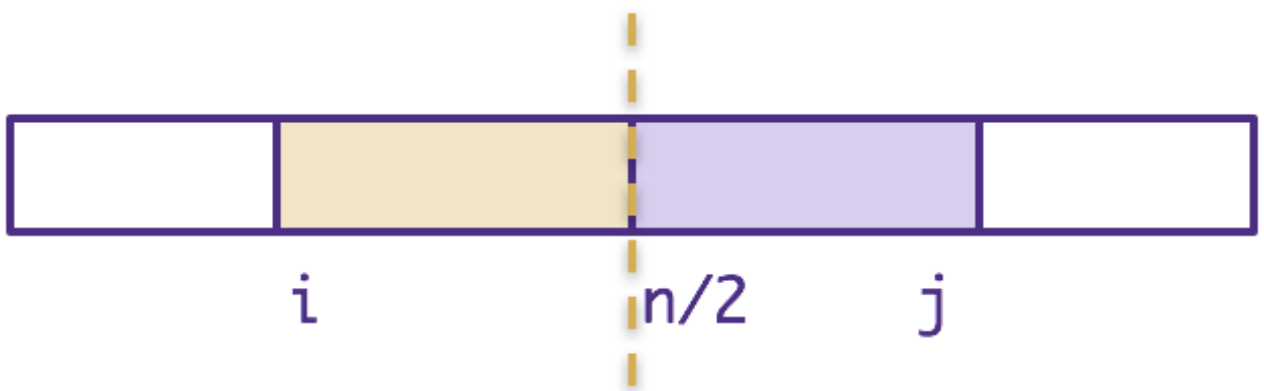
d. Why do we know that the longest sideways trend crossing the midpoint is one that we get by concatenating a range under the left finger and a range under the right finger satisfying the invariant above? (I.e., why is this algorithm correct when we're not actually considering *every* combination of gold and purple ranges?)

3. Implement the divide and conquer algorithm described in problem 2 in the `findLongestSidewaysTrend` method of the source code. Your implementation of the combine step should go in the seperate `findLongestSidewaysTrendCrossingMidpoint` method of the source code.

4. **Extra Credit**: As usual when designing a divide and conquer algorithms, now that we are done, we should step back and consider whether the problem is truly easier to solve with divide and conquer than it is to solve directly. Would you say that that true in this case? Make a convincing argument for why that is or is not the case here.

Combine Algorithm

The two recursive calls of our divide and conquer algorithm will return to us the longest sideways trends from the first and the second half of the data. In the combine step, it only remains to consider ranges that start in the first half and end in the second half. Any such range crosses the mid-way point, drawn here as a dashed line:



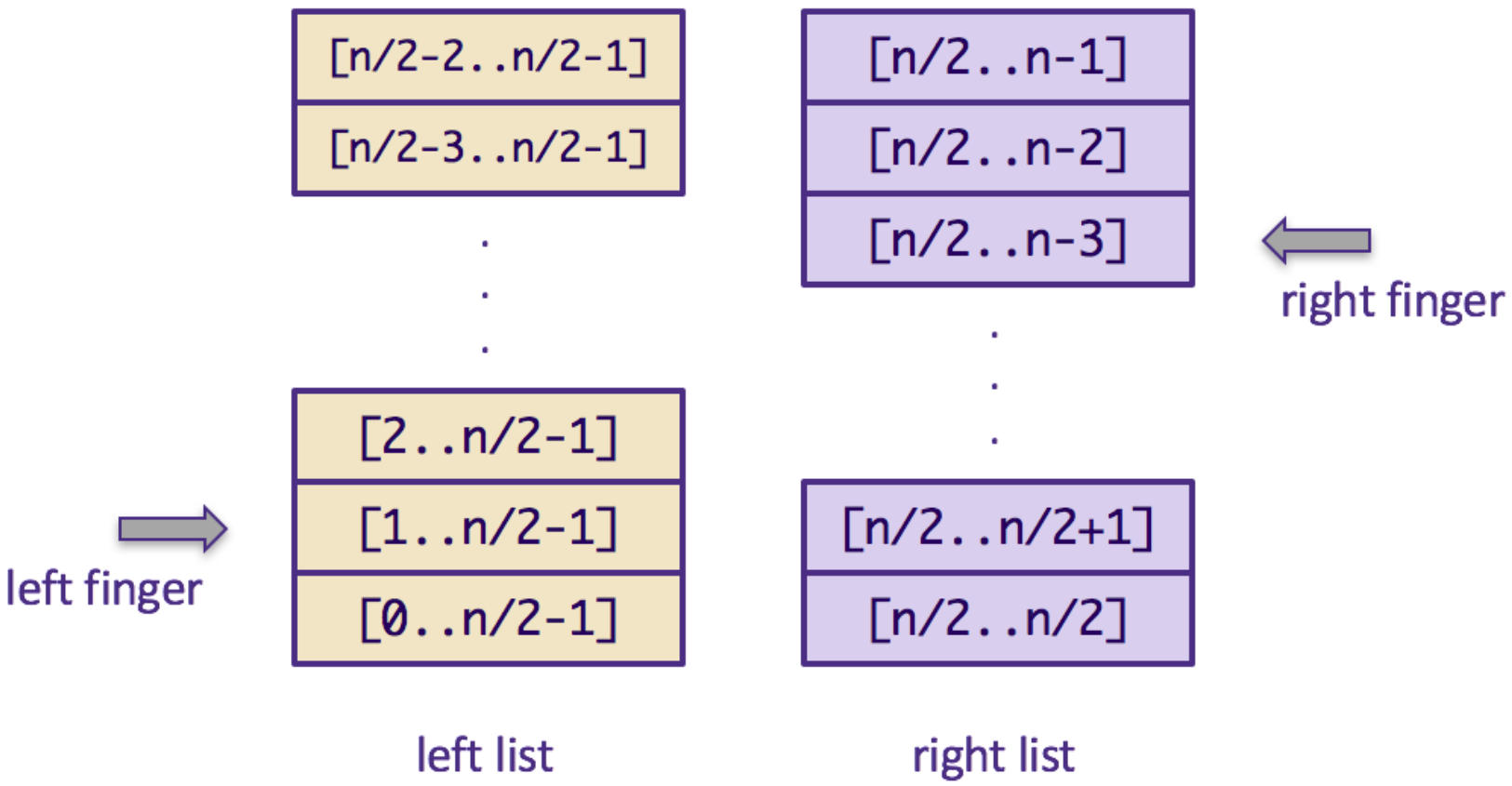
Furthermore, as we discussed for the maximum sub-array sum problem, any range that crosses the mid-way point can be broken into two pieces, as shown in the picture: a gold range to the left of the mid-way point and a purple range to the right of the mid-way point. Hence, we can restrict our attention

considering only ranges that are created by concatenating a gold range with a purple one. (Any such range crosses the mid-way point and any range that crosses the mid-way point can be constructed in that manner.)

Our combine algorithm will start by creating a list of all the gold ranges and all the purple range. There are only n/2 of each.

We will sort the list of gold ranges by their starting index, so the longest ranges are in the front of the list. And we will sort the list of purple ranges by their ending index, so the shortest ones are in the front of the list.

The resulting lists will look like this:



We will use a "two finger" algorithm to construct the ranges that we want to consider. Our left finger will advance through the elements of the left list, starting from the front. Each time the left finger moves, we move the right finger to the **longest purple range** that, when concatenated with the range under the left finger, defines a sideways tend (if any such range exists). That may require moving the right finger many positions in the list from where it was before.

(Note that it is not really necessary to include gold or purple ranges that are not themselves sideways trends, as concatenating them with any other range would also fail to be a sideways trend. Nonetheless, the algorithm will still run in O(n) time even with those ranges included.)

You will complete the description of this algorithm in problem 2 above...