INTRODUCTION TO COMPUTER SCIENCE I
LAB 6
Divide and Conquer

1 Making a fortune, take II

Recall, from Lab-5, the tale of how you found a list of the closing stock prices for one particular stock, going n days into the future. In that assignment, you devised a solution that could compute the ideal days on which to buy and sell that stock, assuming one single purchase followed by one single sale. This solution, which required $O(n^2)$ operations, was tractable so long as n was in the low-hundreds-of-thousands. Since n represented a number of days, that solution was certainly sufficient, given that $n = 100,000$ corresponds to future prices for the next 274 years.

But buying and then selling on a given *day* is soooo twentieth century. These are the days of high-speed algorithmic trading, where stock prices change by the millisecond. In keeping with that change, you now find, on the fabled D-level in Frost, a computer that allows you to connect to a database of future stock prices given at millisecond intervals throughout. At that rate, $n = 100,000$, where n is just a one millisecond of trading activity, corresponds to a mere 1 minute and 40 seconds.

Given this change in trading speed (and the price records that go with it), we need an algorithm that can handle much larger n in a reasonable amount of time...

2 What you must do

Create a directory for this project, change into it, and grab source code:
`bit.ly/COSC-111-lab-6-source`

Open the Java source code file, `FastPickEm.java`, with *Emacs/Aquamacs*. There, you will find something nearly identical to the starting code from Lab-5. Indeed, other than changing *days* into *milliseconds*, the rest is identical. Again, the key method with which you should concern yourself is `findBuySell()`, which, given an array of future prices, calculates and returns an array that contains:

- `[0]` The millisecond at which one ideally should buy the stock, and
- `[1]` The millisecond at which one should sell it.

Therefore, this method must return an array of length **at least 2**, where the value at index `[0]` contains the *buy-millisecond*, and the value at index `[1]` contains the *sell-millisecond*.[1]

[1] Notice the use of the phrase, *at least*. You could return an array that is longer, carrying additional information in those additional entries, This idea is a big hint.

2.1 A fast solution

Write the `findBuySell()` method. Specifically, employ the *recursive divide-and-conquor approach* discussed during lab. For this approach, in order to find the ideal buy/sell times given a list of n prices, the solution should:

1. Split the array into halves. Notice that this splitting can be literal—two new half-sized arrays can be created, and the values from the original array copied into the respective halves—or the splitting can be conceptual—index ranges can be designated as marking the left and right halves of the array.

2. For each of the left and right halves, recursively obtain the following information:

   - The best buy time,
   - The best sell time,
   - The time when the price is at its minimum, and
   - The time when the price is at its maximum.

3. Given this information for each half, choose the best overall buy/sell times from the three following possibilities, one of which yields the greatest profit:

   - The best buy/sell pair of times within the left half.
   - The best buy/sell pair of times within the right half.
   - The best buy/sell pair of times that bridge the halves. That is, the result of buying at the time of the left half's minimum, and selling at the time of the right half's maximum.

4. Return the same quartet of information, listed above, to the caller.

Once you write such a method, **test it**, just as you did with the slower method. Add code to print the array of prices, and then run the program with small but increasing numbers of milliseconds (which you get to specify on the command line).

Once you have determined that your program is working correctly, then **remove the debugging code** that prints the array of prices, and run the program on larger inputs. Begin with $250,000$ milliseconds, which was about the practical maximum for the slow solution:

```
$ java FastPickEm 250000
```

How long does this large a value of n now take? How large can you make n before the program takes more than a few minutes? **Bonus question:** What is the *Big-O* number of operations that this divide-and-conquor solutions requires?[2]

[2]This is truly a bonus question. It's not all that easy to analyze this kind of thing until you take *Data Structures* and then *Algorithms*. But hey, give it your best shot.

3 Submitting your work

Submit your `FastPickEm.java` file with the CS submission system, using one of the two methods:

- **Web-based:** Visit the CS submission systems web page at:
  `www.cs.amherst.edu/submit`

- **Command-line based:** Use the `cssubmit` command at your shell prompt. (WARNING: This method works only on `remus/romulus`.)

This assignment is due on Thursday, Apr-11, 11:59 pm.