INTRODUCTION TO COMPUTER SCIENCE I
LAB 8
Divide and Conquer

1 Making a fortune, take II

Recall, from Lab 7, 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 proportional to 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, where each time step is just a one millisecond of trading activity, $n = 100,000$, corresponds to a mere 1 minute and 40 seconds.

Given this change in trading speed, 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:

```
$ mkdir lab8
$ cd lab8
$ wget -nv --trust-server-names https://goo.gl/X8nakZ
```

Open the Java source code file, `FastPickEm.java`, with *Emacs*. There, you will find something nearly identical to the starting code from Lab 7. 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 2, where the value at index `[0]` contains the *buy-millisecond*, and the value at index `[1]` contains the *sell-millisecond*.

2.1 A fast solution

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

1. A base case. What is the smallest number of milliseconds one could consider? What is the solution in this case?
2. For each of the left and right halves of the prices array[1], recursively obtain the following information:
   - The best buy time,
   - The best sell time,
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. (You will need to do some extra work to compute this.)
4. Return the same two pieces of information, listed above in step 2, to the caller.

You may find that you would like to have some extra input parameters to indicate that you are considering a particular sub-range of the array (like we did for mergesort and binary search). You should ***NOT*** change the header I have given you for `findBuySell()`. Instead, you should create a new method with all the input parameters you need, and call this method from within `findBuySell()`.

Once you write your `findBuySell()` method and any additional methods you might need, **test it**, just as you did with the method you wrote in Lab 7. 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 solution you wrote in Lab 7:

```
$ java FastPickEm 250000
```

Answer the following questions in comments at the top of your `FastPickEm.java` file:

1. How long does the program take to run when $n = 250,000$? (You do not need to measure this precisely. Does it take less than a second? A few seconds? Minutes? Longer?)

[1]One "half" may be one element longer than the other if n is an odd number.

2. How large can you make n before the program takes more than a minute or so?
3. **Bonus question:** In terms of n, what is the number of operations that this recursive solution requires?[2]

3 Submitting your work

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

- **Web-based:** Visit the submission system web page.
- **Command-line based:** Use the `cssubmit` command at your shell prompt.

This assignment is due on Thursday, November 16, 11:59 pm.

[2]This is truly a bonus question. It's not all that easy to analyze this kind of thing until you take *COSC-211: Data Structures* and then *COSC-311: Algorithms*. But hey, give it your best shot.