INTRODUCTION TO COMPUTER SCIENCE I
PROJECT 5C
Divide and conquer, part II

1 Seeing the future

You walk into Frost Library to return a book.[1] You decide to look up Knuth's *The Art of Computer Programming*. Seeing that it is down in C-level, you descend the stairs. Strangely, though, when you get to C-level, you notice, in the semi-darkness, that the stairs **continue down another level**. You have never seen these stairs before, thinking that C-level was the bottom of this architectural eyesore.[2]

Feeling adventurous, you descend to the little-known and, as it turned out, magical D-level. In it, you find stacks of periodicals—newspapers, magazines, etc. You approach a stack of Wall Street Journals, and pick up the one on the top. It's date is *Nov 10, 2012*. "But . . . that's *next Sunday*!" you exclaim. Digging through the stack, each Journal is marked as being yet another day into the future. Disbelieving, you write down the titles of a couple of articles: *Microsoft Surface Hot Item for Holidays*, and *Congress Defers Fiscal Cliff to June*. There's even a sports article: *Amherst Dominates Biggest Little Game*—a game scheduled for this coming Saturday, Nov 10!

Saturday comes, and you watch Amherst whoop Williams on a beautiful afternoon, winning by the exact score in the paper that you saw on D-level. You wake up on Sunday morning, run to Hastings, and buy a copy of the WSJ—and there are those headlines that you copied while in D-level, verbatim. Wow. You run back to Frost, sprinting down the stairs into D-level again. Monday's paper is there, and so is Tuesdays, Wednesdays, etc. Even more oddly, you seem to be the only person in the room. It's not clear anyone else sees it; indeed, perhaps nobody else has ever seen it.

Sensing opportunity, you grab the next few days of the journal, and you look at the stock prices. Wow. If these newspapers are what they seem to be, you could make a killing! So, you grab your laptop out of your backpack, and you pick a stock—Apple continues to ride high—and start writing down its prices for tomorrow, the next day, and so on. After many hours of labor, you have a lengthy list of Apple share prices, day by day, going years into the future.

Now you realize that you need to take advantage of this magical information in order to make as much money as possible. Good thing you're taking COSC 111! Otherwise, you would have trouble figuring out how to calculate, from this list of numbers, on which day you should buy as much Apple stock as you can afford, and on which later day you should sell it.[3] You settle in to write a program that will read that list of Apple stock prices, and after a few seconds of calculating, print out the **day to buy**, the **day to sell**, and the **factor increase in your money**.[4]

[1] You know, the old codex things made of stacked and bound paper.

[2] Not that I have an opinion on the matter.

[3] In the WSJ from Nov 10, 2012, Congress and the Obama administration outlaw short selling, so you know that you must buy first and sell later.

[4] That is, if the stock price increases from \$25 per share on your buy-day to \$125 per share on your sell-day, then your increase is a factor of $\frac{125}{25} = 5$.

2 What you must do

Login and change to your `project-5` directory. Then, grab some starter code from me:

```
$ cp ~sfkaplan/public/COSC-111/project-5c/PickEm.java .
```

Open the Java source code file, `PickEm.java`, with *Emacs*. You will see, as usual, that it contains a fully-formed `main()`, which calls on a methods to generate a random array of "stock prices" (floating point numbers) of some given length (on the command line). It then calls on the method `slowFindBuySell()`, passing it the array of stock prices. This method is expected to return, as a two-element `int` array:

- `[0]` The day number on which one ideally should buy the stock, and
- `[1]` The day number on which one should sell it.

Therefore, this method should **always return an array of length 2**, where the value at index `0` contains the *buy-day*, and the value at index `1` contains the *sell-day*.

2.1 Required steps: A basic version

Write the `slowFindBuySell()` method. Specifically, employ a *brute force algorithm* that considers every possible buy/sell day pairings (remembering that the buy-day must **precede** the sell-day), and chooses the best.

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

```
$ java PickEm 1
Day = 0, price = 811.3915355625351
Buy on 0 at 811.3915355625351
Sell on 0 at 811.3915355625351
Factor profit = 1.0

$ java PickEm 5
Day = 0, price = 811.3915355625351
Day = 1, price = 27.67945220153589
Day = 2, price = 359.17924706247993
Day = 3, price = 149.29240060832484
Day = 4, price = 814.3190563057799
Buy on 1 at 27.67945220153589
Sell on 4 at 814.3190563057799
Factor profit = 29.419623277826087
```

You likely want to test your solution on as lengthy a list of prices as you can examine by hand. Notice that, if you choose a sufficiently high number of days, then the output will be too long to view in your terminal window. So, you can employ *shell redirection*, sending the output that normally appears within the terminal window into a file instead. Having done so, you can then open the file with *Emacs* and examine the whole output:

```
$ java PickEm 500 >& output
$ emacs output &
```

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 a large input. Specifically, with the slow approach, you should run your program with $250,000$ days:

```
$ java PickEm 250000
```

If a sane output results (although it is hard to know if it is *correct*), then you are done. I will run your program with this input size to test it.

2.2 Optional steps: A challenging version

In contrast to the *brute force algorithm*, there is a *divide-and-conquer algorithm* that can be devised for this problem. Doing so requires some substantial thought, so implementing this variation is **not required, but will earn you more credit for the assignment.** The structure and thinking of this solution is in some ways quite like that used for *mergesort* and *binary search*.

Your mission, should you choose to accept it, is to write a new method—`fastFindBuySell()`—that solves this problem. It's signature (the parameter list and return type) should be the same as for `slowFindBuySell()`. Once you write it, you should modify `main()` to call it and print the results, just as it does for `slowFindBuySell()`. Test your program with small numbers of days. Leave the use of `slowFindBuySell()` in place so that you can verify that both versions are yielding the same result.

When you have verified that the new, faster version produces the same results as your slower version, *comment-out*—that is, add comment markers (`//`) in front of the lines in `main()` that call `slowFindBuySell()` and print its results, thus removing the use of the slower solution from your program (for now). Then, if it has been done correctly, you should be able to run the program on an input of $5,000,000$, noting that it takes far less time than the slow solution on an input $\frac{1}{20}^{th}$ the size:

```
$ java PickEm 5000000
```

2.3 More optional steps: Theoretical vs. empirical analysis

If you have gotten both solutions working, you may then take on yet another additional task for even more credit. In particular, you should try to figure out just how slow/fast your solutions are. In particular, there are two approaches that you can take, and you should try both.

Theoretical analysis: For our sorting and searching algorithms, we calculated, ignoring constants, roughly how many operations each algorithm required in terms of the length of the array on which it operated. For example, for an array of n elements, *bubble sort* required $O(n^2)$ operations, but *mergesort* required only $O(n\ lg\ n)$.

Can you perform this analysis for the slow and fast algorithms that you've devised here? Attempt to do so. Write down your solutions. To the extent that you are able, write down your reasoning behind your solutions.[5]

Empricial analysis: Want to know how long each algorithm takes to run? Then **time them!**[6] Specifically, in `main()` remove the commented-out code that calls `slowFindBuySell()` and prints its results, thus enabling your program to compute the solution using both algorithms. Then, use a special method:

```
public static long System.nanoTime ()
```

That is, if your program calls the method `System.nanoTime()`, that method will return, as a `long` integer, the computer's current time in nanoseconds. Add to `main()` a call to this method immediately **before** `slowFindBuySell()` is called, and immediate `after` it returns, storing the results in two different variables. The difference between these two variables is the time elapsed, in nanoseconds, in computing the solution using the slow algorithm. If you then add code to do the same for the fast algorithm, and then print both timing results, you can compare the time required.

With this timing code added to your program, you can then run the program with increasing numbers of days, recording the time required for each algorithm.[7] Use a spreadsheet or a statistics program (e.g., R) to record each timing and the number of days that yielded that timing. You can then create a plot that shows the relationship between the length of the input (the number of days) and the computational time for each algorithm. If you want to really be adventurous, you can attempt some curve fitting to see what mathematical function matches the curves.

Do your empirical observations match your theoretical analysis? Do you need to revise your theoretical analysis?

3 How to submit your work

Use the `cs111-submit` command:

```
cs111-submit project-5c PickEm.java
```

[5] We have only a little practice, and no formal background, in performing this type of analysis. That's OK. Use whatever intuitions and reasoning you may have developed. If you want to be able to perform this kind of analysis more precisely and on more complex algorithms, then take *Data Structures and Algorithms I* and *II*.

[6] It turns out that timing computations is actually a difficult and complex task. For our purposes here, we can ignore the complexities. If you want to know more about the challenges and solutions for measuring computational performance, take *Advanced Operating Systems*.

[7] Of course, there is a limit on how large an input you can use, since the slow algorithm starts to take many minutes at about $300,000$ or $400,000$ days. Your patience and available time will dictate the largest input for which you obtain a measurement.

If you have taken on any of the extra challenges, and you have additional files that you would like to submit, then include them in the list of files provided to the `cs111-submit` command. They will be included along with your Java code.

Part C is due on Nov 15/16, at the start of lab