CSC 207 2014S: Extra Session, Week 6

Overview

- Admin
- You ask questions.
- I try to give answers.

Admin

- Welcome to the people who live on the 3rd floor, or at least seem to.

Questions

Why don't you like syntax coloring?

Distracting. Sometimes confusing.

Talk about good code design

- Both process and product.
  - Thinking while you write
  - What results when you've thought
  - And what you look for after you've written (can I do this better?)
- I was taught to always write general code.
  - You think more deeply when writing general code.
- Agile suggests write code for a specific purpose; only generalize when you are tempted to copy and paste.
  - You're more likely to get it right / understand it in the specific-purpose case
  - And it takes effort to generalize
  - General code isn't always needed, so it's wasted effort
- Having your own library of general code can make you a much more efficient programmer - just use one of your library routines when you need it.

Process concepts:

- Generality - Will this code work in multiple situtations; How do I make it do so?
- Efficiency - How fast is this code likely to run?
  - Premature optimization is a waste of time
  - But you need to think about how slow/fast the methods you're calling are, and try to avoid recalling
- Know and use library.
- Be your own navigator.

- Consider clarity.
- Break the problem into smaller problems
  - But revisit that decision regularly
  - Small functions that interact nicely.
- Have expectations of what the state of your code will be at various points in the program so that you can check them.
  - In C, assert can be helpful.
  - In all languages, comments can be helpful.
- "Know your assumptions"
- Design the algorithm in "English", then go back and code

A bad example

for (int i = 0; i < str.length(); i++) { this.contents[start+i] = str.toCharArray()[i]; } // for

Problems

- We don't know the cost of str.length(), but as C programmers we should worry.
  - Conflicting issue: What if the string is changing? Sometimes it's better to start with something correct and then make it more efficient.
  - Many people worry about function calls in the header of for loops
- We don't know the cost of str.toCharArray(), we can assume it's O(n), which would make our overall algorithm O(n^2), where n is the length of str.
  - This is a big enough improvement that I don't even need to profile.
  - That is, it is NOT premature optimization.

A better version

int strlen = str.length(); char[] strchars = str.toCharArray(); for (int i = 0; i < strlen; i++) { this.contents[start+i] = strchars[i]; } / for

But, doesn't show knowledge of the library. E.g., String.charAt, which we could assume is constant time.

int strlen = str.length(); for (int i = 0; i < strlen; i++) { this.contents[start+i] = str.charAt(i); } / for

But there's more to the library. System.arraycopy is intended to do quick copying of values.

```
System.arraycopy(str.toCharArray(), 0, this.contents, start, str.length());
```

The result is shorter (in terms of code we write), more likely to be correct, likely to be faster, and potentially more readable.

An example of clarity

Problem: Insert an array into the middle of another array

First model: Three loops (or three calls to System.arraycopy)

```
// Deal with the stuff before the insertion point
// Deal with the stuff after the insertion point
// Deal with the inserted stuff
```

Second model: One loop, asking where we are at each point

```
for each position in the target array
   if we're before the insertion point ...
   if we're in the insertion section
   if we're after the insertion section
```

In case it's not clear, Sam would suggest the first.

Thinking about replace

- It will be useful to figure out how big the new array should be before you copy stuff over.
  - Otherwise, you'll expand again and again and again.
- You can do that by finding out how many times the patterns appears in the string, and then doing appropriate calculations.

// Copy the characters from the replacment // Shift the remaining stuff over (vague)

O(logn) algorithms

Although we say the constant multipler doesn't really matter, it makes a HUGE difference for O(log_2(n)) algorithms.

Suppose we have an O(log_2(n)) algorithm that can process a million-item collection in a minute

How big a collection can an algorithm that is twice as fast process in a minute? Now, we can do billion-element collection.

Copyright (c) 2013-14 Samuel A. Rebelsky.

This work is licensed under a Creative Commons Attribution 3.0 Unported License. To view a copy of this license, visit `http://creativecommons.org/licenses/by/3.0/` or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.