CS106B Handout #12
Summer 2013 July 15, 2013

Study Skills and Exam Strategies

Whenever engaging in a new area of study, an important question to ask is "how do I study?" It isn't necessarily the case that study skills developed for one area directly translate to another area. For example, when learning a new language memorization w is important, but in abstract mathematics simply memorizing proofs will not get you very far.

This handout seeks to provide you with a set of skills that will help you prepare for the exams in CS106B, but these skills will likely translate well to exams in other Computer Science, Mathematics and Engineering courses. Additionally, while the motivation for developing these skills is exam performance, they are very useful for learning problem solving methods in general.

The goal of all these strategies is to **force you to think more deeply about the material**.

Strategies for Understanding Code

When met with a foreign piece of code, a common question that one may ask is "What does this code do?" Being able to understand what a piece of code does is important in many "real world" contexts such as when contributing to a project with multiple people. With respect to CS106B, understanding code from lectures and the course reader is important because you're going to need to produce similar code on the assignments and exams.

When confronted with a new piece of code, you'll generally first read the code and try to "simulate" it in your brain to see if you can piece together it's functionality. For simple blocks of code this can work, but this can difficult if:

- The code is poorly commented and/or uses undescriptive variable/function names
- Lots of logic is packed into a few lines of code
- There is complicated recursion.

For the sake of concreteness, let's consider the last case. Consider the following code:

```
int maximizeCoverage(Vector<int> cellTowers) {
     if (cellTowers.size() == 0)
          return 0;
     if (cellTowers.size() == 1)
          return cellTowers[0];
     Vector<int> v1 = cellTowers;
     v1.remove(0);
     Vector<int> v2 = v1;
     v2.remove(0);
     return max(maximizeCoverage(v1),
                cellTowers[0] + maximizeCoverage(v2));
}
```

This is the code for maximizing cell phone tower coverage that we went over in class, but for the purposes of this handout let's assume we haven't seen this code before.

Optional Step 0) **Copy the code by hand onto a piece of paper**

Sometimes simply copying code by hand can be helpful in developing a better understanding. The reason for this is because the act of copy code by hand can force your brain to process the code at a deeper level than you may be doing when you simply read the code. As you are copying the code, ask yourself probing questions. *Do I understand what this line of code is doing? Do I understand what this loop does?* These types of questions can help you to determine if you are actually understanding the code.

Step 1) Walk through simple examples

Running through the code on paper with some simple input can be very helpful in developing an understanding of what the code does. For example, in the above code we could consider some of the extremely simple input such as a **`Vector<int>`** with 0 and 1 elements. This would give us a better appreciation of what the base cases do.

When working with recursive code, it can also be very helpful to generate a decision tree for some simple inputs. For example, we could construct the decision tree for **`maximizeCoverage()`** with 2 or 3 elements in the **`Vector<int>`**.

Step 2) Generate Pseudocode

In class we've only generated pseudocode as a way planning out an algorithm before implementing it. Doing the inverse – generated pseudocode from existing code – can be very helpful in developing an understanding of the code.

For example, having run through some examples with **`maximizeCoverage()`**, we might generate the following pseudocode:

```
BASE CASES:
        if no more towers
                return 0
        if only 1 tower
                return coverage of tower
        x = recurse on (all but the first tower)
        y = (coverage of first tower) + recurse on (all but first two towers)
        return max(x,y);
```

Some of you may be familiar with the study strategy of summarizing information as you read it (e.g. writing short summaries for each section of a Psychology textbook). Generating pseudocode from existing code is effective for similar reasons. First, it provides a valuable reference to use in the future. Second, and arguably more importantly, it forces you to develop a deeper understanding of the code.

Strategies for solving coding problems

When writing code, especially in a time-constrained environment (such as during an exam) it can be very tempting to immediately start writing code. **Avoid doing this!** Unless the code you need to write is simple for you, it's often valuable to plan out the code you will write before implementing it. The danger of immediately writing code is that the strategy you commit too a solution that will be either overly complicated or insufficient to solve the problem at hand.

1) Figure out what you function needs as input and output

Before doing anything else, figure out what your function is going to return and what it needs to take as input. This may seem obvious, but it is very important because in some cases if you don't give your function what it needs, then you won't be able to successful solve the problem. This is especially important for recursive functions.

2) Generate pseudocode

Generating pseudocode can be valuable because it frees you from the nitty gritty implementation details and lets you think about how you can solve the problem at a very high level. Stuff like getting indices correct and picking what data structures to use can get in the way of developing the overall algorithm you want to implement.

As you're going through this process you may realize that you need additional parameters or do not need all parameters from step (1). One of the great things about this process is that making these sorts of changes is super easy because we haven't committed to writing any code yet!

3) Pick the best data structures for the problem

Your pseudocode probably mentions some sort of data structure, and now is the time to pick which ones to use. This is an extremely important step because your choice of data structure will have a huge impact on the complexity of your code. For example, let's say you need to store a collection of distinct integers. A natural choice for this is a **`Set<int>`**, but let's say you instead use a **`Vector<int>`** - what impact will this have on your code? Unlike the **`Set`**, the **`Vector`** does not have a method that checks if it contains a certain element or not, so this means whenever you want to perform an operation like **`Set::contains()`**, you'll need to perform a linear search of the **`Vector`**. In this case, using a Vector creates additionally work for you to do.

4) Write the code

Only now that we have selected our parameters, return type, generated pseudocode and chosen data structures are we ready to write code. What's great about doing steps (1) through (3) first is that all the difficult algorithmic choices have been made so all we need to do is worry about syntax and implementation details.

Tips for writing code by hand

1) Don't worry about "minor" syntax errors

We don't care about "minor" syntax errors, so neither should you. Semicolons, minor typos, unbalanced parentheses – as long as we understand what your code says then you're good. That said, syntax errors can lead to ambiguity, which ***is*** an issue.

That said, what you write should be C++ code (i.e. it shouldn't be pseudocode), but it doesn't have to have perfect syntax.

In summary – your code doesn't have to compile, we just need to understand what you're doing.

2) Invest time decomposing your code

While we don't grade your exam on style, good decomposition can save you a ton of work (and prevent bugs), particularly when it saves you from "copy-pasting-by-hand" code. Think ahead, and if you suspect that you'll need to run the same block of code several times, then it may be worth implementing it in a separate method or using some form of iteration (for/while/foreach)

For example consider the following code, which takes as input a location in a Grid<int> and sums the 8 surrounding cells:

```
int sumCross(Grid<int> &g, int row, int col) {
     int sum = 0;
     if (g.inBounds(row-1,col))
          sum += g[row-1][col];
     if (g.inBounds(row,col-1))
          sum += g[row][col-1];
     if (g.inBounds(row+1,col))
          sum += g[row+1][col];
     if (g.inBounds(row,col+1))
          sum += g[row][col+1];
     if (g.inBounds(row-1,col+1))
          sum += g[row-1][col];
     if (g.inBounds(row+1,col-1))
          sum += g[row][col-1];
     if (g.inBounds(row+1,col-1))
          sum += g[row+1][col];
     if (g.inBounds(row,col+1))
          sum += g[row-1][col+1];
     return sum;
}
```

This code is functionally correct, but it has a lot of "copy pasted" code. Copying and pasting code is bad practice in general, but it is especially bad when writing code by hand because:

- It wastes time (you cannot simply "Control-C, Control-V").
- It increases the probability of introducing errors.

Consider the following, decomposed version of this function:

```
int sumCross(Grid<int> &g, int row, int col) {
     int sum = 0;
     for (int dRow = -1; dRow <= 1; dRow++)
          for (int dCol = -1; dCol <= 1; dCol++) {
               int r = row+dRow;
               int c = col+dCol;
               if (dRow != 0 || dCol != 0) {
                    if (g.inBounds(r,c))
                         sum += g[r][c];
               }
          }
     return sum;
}
```

This code will take much less time to write down as well as being easier to check for errors.