Julie Zelenski
CS106B

Handout #23J
January 28, 2013

Exam Strategies and Tactics

This handout was written by (a) Julie Zelenski, (b) Julie Zelenski, (c) Julie Zelenski, or (d) all of the above.

The exams in the CS106 courses can be challenging and even a bit intimidating. Hopefully you have been keeping up in lecture and doing well on the assignments, but may be unsure of how to make sure your skills will translate well to the exam setting. The practice midterm gives an idea of what to expect and this handout gives some sage advice gathered from our current and past staff members. We hope you will find our tips useful when preparing for and conquering this upcoming challenge!

The rationale behind pencil and paper exams

Students often suggest that exams should be done more like assignments: using a compiler, having code completion and searchable documentation, being able to run, test, and debug, etc. The logistics of an online exam add serious challenges in terms of fairness and security, but we did try this experimentally and it didn't work they way we'd hoped. For a start, much valuable time was lost to dorking with details (proper `#includes`, syntax issues) that we don't even count in an exam situation. In a time-restricted situation, immediate feedback from the compiler can be more of an impediment than an advantage. Imagine this, you read the first problem, have a good idea how to solve it, write your solution, and trace its operation and feel good. In a paper exam, you then move on to the next problem. In an online exam, you compile and test it. Suppose it exhibits a bug—even though it may only be a minor issue, you can see your answer is wrong so you hunker down and rework and retest until perfect. Writing your first solution took 20 minutes and would have earned, say, 17 of 20 points. You spent another 20 minutes debugging to earn those remaining 3 points. Bad deal! The rest of your exam also suffers because you used up so much time. We had students who never made it past the first or second problem in the online exam. Being confronted with clear evidence of bugs made it impossible to move on. We tried again and warned students about this effect, but resistance was futile. Given the limited time available, we want you to write your best answer and move on and paper seems to be the means to encourage exactly that.

We know that writing on paper is not the same as working with a compiler, and we account for that in how we design and grade the exam. We are assessing your ability to think logically and use appropriate problem-solving techniques. We expect you to express yourself in reasonably correct C++, but we will be quite lenient with errors that are syntactic rather than conceptual.

How to prepare for the exam

- "**Open book" doesn't mean "don't study".** The exam is open-book/open-notes and you can bring along the reader, your notes from lecture, course handouts, and printouts of all your assignments. We don't expect you to memorize minute details and the exam will not focus on them. However, this doesn't mean you shouldn't prepare.

There certainly isn't enough time during the exam to *learn* the material. To do well, you must be experienced at working problems efficiently and accurately without needing to repeatedly refer to your resources.

- **Practice, practice, practice.** A good way to study for the programming problems is to take a problem (lecture or section example, chapter exercise, sample exam problem) and write out your solution under test-like conditions, e.g. on a blank sheet of paper using a pencil with a short amount of time. This is much more valuable than a passive review of the problem and its solution where it is too easy to conclude "ah yes, I would have done that" only to find yourself adrift during the real exam when there is no provided solution to guide you!
- **Get your questions answered.** If there is a concept you're a bit fuzzy on, or you'd like to check your answer to a chapter exercise, or you wonder why a solution is written a particular way, get those questions answered before the exam. Swing by the LaIR, come to office hours, or send an email and we're happy to help.

How to take the exam

- **Scan the entire exam first.** Quickly peruse all questions before starting on any one. This allows you to "multitask"—as you are writing the more mundane parts of one answer, your mind can be brainstorming strategies or ideas for another problem in the background. You can also sketch out how to allocate your time between questions in the first pass.
- **Spend your time wisely.** There are only a handful of questions, and each is worth a significant amount. Don't get stuck on any particular problem. There is much opportunity for partial credit, so it's better to make good efforts on all problems than to perfect an answer to one leaving others untouched.
- **Consider the point value of each question.** Divide the total minutes by the total number of points to figure the time per point and use that as guide when allocating your time across the problems. You may want to reserve a little time for checking your work at the end as well.
- **Leverage the materials you bring with you**. If you know of a function in the reader or a handout that would help, you can simply cite the source and use it. You do not need to rewrite it. If you have a function you wrote for an assignment that you would like to use, you can copy it from your assignment printouts (hence, why we suggest you bring them).
- **Style and decomposition are secondary to correctness.** Unlike the assignments where we hold you to high standards in all areas, for an exam, the correctness of the answers that dominates the grading. Decomposition and style are thus somewhat de-emphasized. However, good design may make it easier for you to get the functionality correct and require less code, which takes less time and has fewer opportunities for error. Comments are never required unless specifically indicated by a problem. When a solution is incorrect, commenting may help us determine what you were trying to do and award partial credit.
- **Answer in pseudocode, but only if you must**. If the syntax of C++ is somehow in your way, you can answer in pseudocode for partial credit. There is a wide variation

in the scoring for psuedocode. Some pseudocode is vague and content-less and does little more than restate the problem description ("I would recursively try all the subsequences to find the longest") and thus is worth next to nothing. The more details it provides, the better. We typically award at most half of the points for clear psuedocode precisely describing a correct algorithm. But truthfully, very good pseudocode contains so much information (e.g. "scan the next token from input, if it the first char is a digit, then convert to an integer") that it typically would have been easier and more concise to just write in C++ in the first place.

- **Pay attention to specific instructions.** A problem statement may include detailed constraints and hints such as "you must solve this recursively" or "you should not destructively modify the input list" or "you do not have to handle the case when the string is empty" or "this operation must run in constant time." You may want to underline or highlight these instructions to be sure you don't overlook them. These constraints are not intended to make things difficult, typically we are trying to guide you in the direction of a straightforward and simple solution. If you disregard these instructions, you are likely to lose points, either for not meeting the problem specification and/or for errors introduced when attempting a convoluted alternative.
- **Syntax is not that important if it is clear what you mean.** We won't trouble you about most syntax as long as your intentions are clear. But if there is ambiguity in your attempt, correct syntax can help us get the correct meaning. For example, if we see a `for` statement followed by two lines, where both lines are vaguely indented or a third line has been added in after the fact, we may be confused. If there are braces around all the lines, it will be clear you intended both to be a part of the loop body, but without the braces, we can't be sure and it may make your answer incorrect.
- **Write in pencil.** CS exams done in pen are often messy and difficult to grade. Your first draft may have "typos" (e.g. missing arguments in function call, statements out of order), and in pencil, you can easily erase to make the necessary corrections, in pen, it is hard to make such changes and still keep your intentions clear.
- **Cross-out abandoned attempts rather than erase them.** As it usually turns out on a CS exam, you will have false starts on a problem—you try one strategy and hit a dead end. You try something else and then realize you actually were closer to the right solution the first time. If you haven't erased your first attempt, you can always go back to it. Once you work out a better answer, cross out your earlier attempt. When you cross out work, please direct us to where you have written the solution you want graded instead. If you forget to cross out your bad attempt and hence appear to have two answers, we will grade which ever is closer to the problem statement.
- **Save a little time for checking your work.** Before handing in your exam, reserve a few minutes to go back over your work. Check for missing initialization/return statements, correct parameters passed to functions, etc. We try not to deduct points for minor things if it is obvious what you meant, but sometimes it is difficult to decipher your true intention. You might save yourself a few lost points by tidying up the details at the end.

Advice on specific question types

The rest of this handout goes through some of the more common question types for the exams (not all of which appear on this midterm) and offers some specific advice for each question type.

C++ coding

Writing C++ code shows up all over the exam, but sometimes we include problems that are focused on C++ coding in isolation— manipulating strings, streams, pointers, structs, and so on. These problems take various forms (tracing code, drawing memory diagrams, finding/correcting bugs, writing functions, etc.) Some things to consider:

- Sometimes we ask you to trace existing code and either show the output or draw a memory diagram at a given point. There are no shortcuts for these, you just need to carefully walk through the code line-by-line and update the state of the stack and heap as assignments are made. Parameters passed by reference can be a stumbling block, so be careful working through such code. Be sure to show your work so that we can give you partial credit even if your end result is wrong.
- Bookmark the reader pages that list library functions for strings, files, random, etc. so you can quickly look up the interface when needed. During the exam, you can also come ask us if you can't remember/find the function you need to use.

Recursion

Recursion exam questions are similar in format/structure to those you saw on the recursion problem set. Some things to consider:

- Many section leaders agree that recursion is one of the most difficult exam topics. They suggest that you read any recursion problems early in the exam, but if you don't see the answer quickly, move on to other problems. This way, you can be thinking about it in the background the entire time, and you're more likely to hit on the insight needed to solve the problem.
- Between the recursion problem set, the section handouts, and the exercises in the course reader, you have a lot of problems for practice. Practice is an excellent way to strengthen your skills for recursion.
- Many recursion problems are variations on classic themes (permutations, subsets, and so on). When you first read the problem, try to identify what other problems it resembles that you've already seen. How could adapt a previous solution to help you solve this new problem?
- Most likely the most difficult tasks will be coming up with an appropriate recursive decomposition. Focus on these three things: what part of the problem can be handled at the current step, what are the smaller, simpler recursive call(s) to make from here, and how to combine the results from the current step and the recursive call(s) to solve the current version.
- Try to come up with the simplest base cases possible.
- Trace your recursive cases on some sample input and ensure you are always making progress toward a base case and solving the whole problem.

- Sometimes you'll need a wrapper. If it looks like you don't have enough data given the parameters as given, think of what other data might be necessary and see if you can incorporate this into additional arguments passed by the wrapper to the real recursive function.

Client use of classes

These questions ask you to use a class or classes as client to accomplish some task. The CS106B classes (`Vector`, `Queue`, `Map`, `TokenScanner`, and so forth) are likely to make an appearance in such questions. Some things to consider:

- Being on the client end of a class, the class provides you with a lot of functionality for free. Bookmark the tables of methods in the reader so that you can quickly look up features as needed.
- There is some crufty syntax required to be a client of a template class. Only a small amount of points will be reserved for those minor details of syntax. Those are easy points to earn if you are careful, but also not too much to lose if you need to focus your attention on writing the correct algorithm.
- Often we are explicit about the algorithm/data structures that we want you to use, so be sure to read our problem statement carefully and follow our instructions in order to maximize credit.
- Often these questions are some of the "easier" ones. If the client code correctly leverages the features provided by the given classes, the task it needs to accomplish can be relatively straightforward. Let's hear it for the power of abstraction!

Algorithmic thinking

For some problems we are quite prescriptive about the approach we want you to take, but you can expect that at least one problem in the exam will ask you to come up with your own algorithmic approach to solve a problem. For this we are trying to gauge your ability to think algorithmically and make use of various foundational techniques (C++ coding, use of libraries, client use of classes, recursion, etc.) in a new context.

- The challenging part of these questions is usually in coming up with an approach. Once you have that, expressing the algorithm in code is usually not difficult. However, if you're pressed for time, pseudo-code that describes a working strategy can earn some partial credit.
- There is often more than one valid approach for these questions (in fact, it is often our intention to provide that freedom). Some of the strategies are more straightforward, and thus easier to write, but an alternative that is correct and meets the specification of the problem can earn full credit.
- There may be information in the problem statement that attempts to guide you toward the more promising strategies. If we provide a hint or suggestion (e.g. "you may find it helpful to…" or "consider using…") then you are free to take or leave that advice. However, stronger statements (e.g., "your solution must . . ." or "your solution cannot . . .") must be respected to earn full credit.

Big-O and algorithmic analysis

Big-O tends to be sprinkled throughout the test. Sometimes we give you mystery functions that you need to analyze for big-O performance. Other times you are asked to report the big-O for functions you wrote or designed or we might ask you to choose an algorithm that meets a specific big-O runtime requirement. Some things to consider:

- The big-O-mystery questions are usually not overly complex, so if you're finding yourself needing to solve equations that are significantly more difficult than what was done in lecture and in the reader, you're probably making the problem harder than it needs to be.
- When we ask you to "justify" your analysis, we accept many varied ways for you to show how you arrived at your answer: a formal mathematical proof, prose describing informal reasoning, describing the tree branching/depth for a set of recursive calls, by analogy to other algorithms, etc. Use whatever you're comfortable with.
- The most common $O(N)$ functions are those you've seen in the reader and in class: $O(1)$, $O(\log N)$, $O(N)$, $O(N \log N)$, $O(N^2)$, and $O(2^N)$.