CS166
Spring 2019

Handout 03
April 2, 2019

Problem Set Policies

This handout contains information about the problem sets for CS166. Specifically, it contains

- ***submission instructions*** so you know how to turn in the problem sets;
- our ***Piazza policy*** for asking questions online;
- our ***collaboration policy*** with information about working in pairs;
- our ***problem set expectations***, with information about what we're looking for in your answers to theory and coding questions;
- our ***regrade policies***, which outlines our policy on regrading assignments; and
- our ***late policies***, which discusses late days and the like.

If you have any questions, please feel free to contact the course staff.

Submission Instructions

This quarter, we will be using GradeScope to handle problem set submissions and grading. To sign up for GradeScope, visit www.gradescope.com and enter this code:

`M42XBN`

Once you've signed up, you can submit your assignments by uploading them to GradeScope.

GradeScope only accepts electronic submissions. Because in the past we've had issues with low-resolution scans of handwritten work, you are required to type your assignment solutions and submit them as a PDF; scans of handwritten solutions will not be accepted. LaTeX is a great way to type up solutions.

When submitting on GradeScope, if you're working with a partner, please list both of your names on GradeScope in addition to on the PDF itself. To do so, have one person submit, then, after the submission completes, have them add the other student's name to the submission. Since we rely on GradeScope for our final grading spreadsheet, if you forget to include your partner on the submission – or if your partner forgets to list *you* on the submission – then only one person will get credit for the assignment. We ***strongly*** recommend that you always check to make sure that your assignment was submitted correctly, especially if you weren't the one submitting it, just in case your partner forgot to list you.

We ask that you submit your answers to programming questions separately from your written answers. You should submit your code electronically by `ssh`ing into `myth`, `cd`ing into the directory containing your solution files, then running

```
/usr/class/cs166/bin/submit
```

to submit your work. You'll be prompted for your name, whether you worked with a partner, and the problem set number. We'll test your code on the `myth` machines, so please make sure that your code works correctly there before submitting.

Piazza Policy

We have a Piazza forum (http://www.piazza.com) where you can ask questions and search for partners. You're welcome to ask questions online, and the course staff and other students can then provide answers.

Please exercise discretion when asking questions that might give away the answers to problem set questions. If you'd like to ask a question that you think would give away too much information about the solution to a problem, post your question privately.

Collaboration Policy

You are allowed to work on the problem sets individually or in pairs. Regardless of how many people you work with, your problem set will be graded on the same scale. You are not required to work with the same people on each problem set – you're welcome to work in a pair on one problem set, individually on the next, in a pair with a different partner the next time, etc. If you do work in a pair, please note that both members of the pair are responsible for ensuring that each assignment is completed and submitted on time.

If you submit in a pair, you should submit just a single set of solutions. Both members of the pair will earn the same grade on the problem set. That way, two or more TAs don't accidentally end up grading the same submission multiple times.

For more details about collaborating with other students, please read over our Honor Code policy.

Problem Set Expectations

CS166 as a course is designed with the assumption that you've finished the theory section of either the CS core (for undergrads) or foundations requirements (for graduate students). We expect that you've read and written many programs, proofs, and algorithms at this point, and we hope that you're starting to develop a good sense of what makes something easy to understand. Both so that you can specifically practice your coding / algorithm designing / proofwriting skills and as a courtesy to your TAs, we'll expect that any work you submit will be designed to be easy to read and understand (in addition to being correct, of course). For example:

- On programming assignments, you're expected to write beautiful, well-commented, well-decomposed code. If this is the sort of thing that goes without saying for you, great! But if you need convincing as to why we're requiring this, here are two reasons. First, the *altruistic* reason: you know that the TAs, who were not that long ago in your shoes, will be reading your code, and it's good to be respectful of them and their time. Therefore, you should try to make your code clear and easy to read so that if they find a bug somewhere, they can point out where it is without too much hassle. Second, the *selfish* reason: the programs we're going to ask you to write this quarter aren't very large, but they are pretty nuanced. You don't want to code yourself into a hole where you're forgetting which variables are indices and which are values, nor do you want to try to debug things by hunting through tons of `cout` and `printf` statements looking for the right one. So *of course* you'll write clean code, since it'll save a ton of headache when (not if) things break.
- If you're writing proofs, you're expected to do so in a clear and lucid way that helps guide the reader through your reasoning. We're expecting that you'll write in complete sentences and use mathematical notation where appropriate but not as a substitute for plain English. If you have a complex, multi-part proof, you should break it apart into smaller pieces and, ideally, include a brief introduction outlining the high-level approach you're going to be taking in your argument. If drawing pictures would make things easier to understand, go for it! If doing a small worked example before writing the formal proof would clarify things, do that! Remember that the TAs have to be able to read and understand what you're writing, and they really do want to hear what you have to say, so please try to make it easy for them. 😃

- If you're designing and analyzing an algorithm or data structure, you should present the algorithm in the clearest way that you can. For reference, writing pseudocode for the whole algorithm and saying "here's what I would do!" is *usually not* the clearest way to describe an algorithm – ask anyone who's been a CS161 TA if you doubt us on this one. Instead, give a high-level overview of the algorithm or data structure, pointing out any bits that you think the reader might find unexpected or unusual. Then, go a little lower-level, describing the steps in the algorithm. Feel free to bust out pseudocode here if you think it would help, but if you do, make sure that you've written it in a way that's easy to follow and doesn't require knowledge of any terms you haven't yet defined. From there, feel free to proceed to the analysis.

Learning to communicate these sorts of ideas effectively is incredibly valuable, especially if you find yourself collaborating on a team either in industry or in academia. If this is something you find challenging, no worries! Come talk to us. We're happy to provide pointers and advice.

If you're the sort of person that likes step-by-step checklists, here's a bare minimum set of requirements for any code you submit:

- ***Your code should be well-commented***. From experience, it is very difficult to read code that someone else wrote if they didn't leave comments behind describing what it is that they were trying to do. At a bare minimum, you should include comments describing what all your helper functions and helper types do, how edge cases are handled, etc. Ideally, you should also include comments on any dense sections of code explaining what that code does. If there are any spots where your code is handling some particular task in an unusual or non-obvious way, please leave some notes so we know what you're doing.
- ***Your code should be well-decomposed***. There's a bad tendency in algorithms and data structures contexts to see hundred-line monster functions. Please do not do this. Break larger pieces of code apart into smaller, bite-sized chunks that all perform a single task. If you find yourself working with some concept that's logically separate from other concepts, make it into its own type and use that type where appropriate.
- ***You should have clear variable names***. There's a tension between the mathematical side of things, where single-letter variable names are the norm (so much so that we've raided the English, Greek, and Hebrew alphabets looking for more letters to use!), and the programming side of things, where single-letter variable names make things nigh impossible to read. With the exception of loop indices in cases where the index can easily be inferred from context, please do not use single-letter names. Describe what your variables represent, and do so in a way that makes the code lucid.
- ***Don't be cute, unless you need to***. We'll be writing most of our code in C and C++, where it's possible to write both extremely elegant code and highly obfuscated labyrinths of twisted logic. Aim for clarity above efficiency unless there's a compelling reason not to do so. For example, please divide by two by writing `value / 2`, not `value >> 1`, unless there's some specific reason that a bit-shift more clearly expresses your intent. You probably should never need to `#define` *anything* in this class, and you should *especially* not use `#define` to make shorthands for iterating over things. Language features like inline functions, `const`, enhanced for loops, etc. have mostly eliminated the need to do things like this.

  If you do find yourself needing to pull all the stops out in order to improve performance, that's fine. But *be sure to extensively comment any aggressively-optimized sections of your code* so that the TA reading over it can appreciate the beautiful ideas you used to squeeze out a little bit more efficiency. In the past, we've had people turn in optimized code that neither we *nor the code's original authors* could fully understand. That's not good for anyone. 😃
- ***Don't leave in dead code***. It's perfectly fine to sprinkle some `cout` or `printf` statements liberally throughout your code when you're testing things. It's also totally reasonable to comment out an old implementation of something if you think you found a better way to achieve the same result. But please don't submit code that has debug printouts in it or which has commented-out blocks. It makes things harder to read and can mess up our autograding infrastructure.

The above rules cover our coding expectations, but there's a lot more to this class than the code you'll be writing. In many cases, we're going to ask you to design data structures that solve various problems, and either for the sake of time or for the sake of clarity we're going to ask you to express your resulting data structure purely in prose. If you need to write up a data structure or algorithm in plain English, you're welcome to do so however you'd like as long as it's lucid. If you think you know how to do that already, great! Go do whatever sparks the most joy. But if that's something you've never really had to think about before, here's a little template you can follow:

- ***Begin with a short, high-level description of the idea behind the data structure***. This should be a two or three sentence paragraph describing the intuition behind the data structure. This will help the TAs get a better sense for how the data structure works.
- ***Describe the representation of the data structure***. Give some details about how the data structure is actually put together. You can do this with details such as "store two max heaps called `a` and `b`," or by describing a modification of an existing data structure, such as "store a Fibonacci heap, but where each node stores a pointer into a balanced binary search tree."
- ***Describe any invariants or accounting schemes for the data structure***. Some data structures maintain strict invariants on their internal representation. For example, a binary min-heap data structure ensures that each node always stores a value no larger than its children and that the tree is a complete binary tree. If your data structure doesn't have any invariants, you don't need to list anything. When we begin discussing amortized analysis, you can also list any charging schemes or potential functions here.
- ***Describe each of the operations and give their runtimes***. For each operation, describe how that operation is performed. We'd prefer explanations in plain English, but if you think that pseudocode would be better, you can use that if you'd like. *Just make sure that your description is complete – there shouldn't be any ambiguities in how to perform each operation.* Then, explain why these operations are correct and justify why the data structure meets specified time bounds. You don't need to write a formal proof of correctness unless asked.

For example, consider the following problem:

> Design a data structure that supports the following operations: **insert(*x*)**, which inserts real number x into the data structure and runs in time O(log n), where n is the number of elements in the data structure, and **find-median()**, which returns the median of the data set if it is nonempty and runs in time O(1).

This is great problem to work through if you haven't seen it before. We have a sample solution on the next page, so try this problem out before moving on. As a hint, try using heaps.

Here is a possible answer to this problem and a sample writeup. Note that you don't need to include section headers like these; we're just doing this because in this case we think it's easier to read.

Overview:

This data structure works by storing the data in a min-heap and a max-heap such that the two middle values are at the top of each heap. Each time we insert elements, we shuffle elements between the heaps to keep the middle values at the top. Fortunately, the combination of the min-heap and max-heap make it so that we don't need to move too many elements around to find the median.

Representation:

A max-heap *left* and a min-heap *right*.

Invariants:

There are two invariants: the *ordering invariant*, which says that all elements in *left* are less than or equal to all elements in *right*, and the *size invariant*, which says that size(*left*) = size(*right*) if there are an even number of elements, and otherwise the sizes of *left* and *right* differ by only one. These guarantees mean that if there are an even number of elements in the data structure, the median is the average of *max*(*left*) and *min*(*right*), and otherwise the median is the *max* or *min* value of whichever heap is larger.

Operations:

insert(*x*): First, determine which heap should contain *x* to maintain the ordering invariant. If $x < max(left)$, then add *x* to *left*; otherwise add it to *right*. This may break the size invariant. The size invariant can only be violated if before adding the value, there were an odd number of entries in the data structure (since if previously there were an even number of values, the heaps would have to have the same size). Therefore, if after inserting the value there are an even number of elements, and if additionally and one heap has exactly two more elements than the other, dequeue from that heap and enqueue the appropriate value into the other heap. This operation preserves the ordering invariant, since the value removed is either the biggest value from *left* or the smallest value from *right*. This operation requires only O(1) heap inserts or deletes, so it runs in time O(log *n*).

find-median(): If there are an odd number of elements in the data structure, one of the two heaps must have one more element than the other. If it's the maximum element of *left*, then that element is greater than half the elements (namely, the other elements of *left*) and smaller than half the elements (the elements in *right*), so it's the median. Therefore, return *max*(*left*). By similar reasoning, if the odd element is in *right*, then *min*(*right*) is the median, so we can return it.

Otherwise, there are an even number of elements in the data structure. This means that the median element is the average of the two elements closest to the median point. Using reasoning analogous to the odd case, we know that *min*(*right*) and *max*(*left*) are those two elements, so we can return the average of *min*(*right*) and *max*(*left*).

Both of these operations only require calling *min* or *max* in *right* and *left*, and therefore run in time O(1).

Some of the questions on the problem set will be theory questions that ask you to prove various mathematical results that are relevant for the analysis of data structures. For questions like these, we expect that you'll write a formal mathematical proof of the result. If you have enough proofwriting experience that you feel comfortable writing lucid mathematical prose, go for it! If you're still learning how to do this and would like some guidance, here's a recommendation about how to structure things:

- ***Give a high-level description of your analysis or proof***. If you're writing a proof, you might give a two or three sentence description of the main insight behind the proof and how you'll turn that insight into a proof. If you're asked to perform a calculation of some sort, you can explain how you went about performing that calculation.
- ***Write the proof or calculation***. This is where you'll either write a formal mathematical proof or work through the steps in a calculation in detail.

Please write in complete sentences. A great way to see if you're doing this is pick up your proof and read it aloud, with all mathematical symbols and jargon replaced with the phrase "mugga mugga." If what you get back is grammatical – every sentence has a subject, a verb, and, optionally, an object – then you're doing it right! If not, figure out which sentences don't parse, then go fix them. Another cool trick: if you can't find a subject for the sentence, have the subject be "we" (you and the reader), and the verb be something like "see," "learn," "infer," "discover," etc.

As an example, consider the following problem:

> Consider a binary heap B with n elements, where the elements of B are drawn from a totally-ordered set. Give the best lower bound you can on the runtime of any comparison-based algorithm for constructing a binary search tree from the elements of B.

Here is one possible solution:

Proof Idea: The lower bound is $\Omega(n \log n)$, and this is a tight bound. We'll prove this by first showing that there's an $O(n \log n)$-time, comparison-based algorithm for constructing a BST from the elements of an n-element heap. Then, we'll show that any $o(n \log n)$-time, comparison-based algorithm for doing the conversion would make it possible to sort n elements in time $o(n \log n)$ using only comparisons, which we know is impossible.

Proof: First, we'll show that there is an $O(n \log n)$-time, comparison-based algorithm for constructing a BST out of the elements of B. Specifically, just iterate across the n elements of B and insert each into a balanced binary search tree. This does $O(n)$ insertions into a balanced binary search tree, which will take time $O(n \log n)$. This algorithm is also comparison-based because binary search tree insertion is comparison-based.

Next, we'll show that no $o(n \log n)$-time, comparison-based algorithm for constructing a BST from a binary heap exists. Assume for the sake of contradiction that such an algorithm exists. Then consider the following algorithm on an array of length n:

- Construct a binary heap B from the array elements in time $O(n)$.
- Create a binary search tree T from B in time $o(n \log n)$.
- Do an inorder traversal of T and output the elements in the order visited in time $O(n)$.

Note that the runtime of this algorithm is $o(n \log n)$, and each step is comparison-based. However, this algorithm will sort the elements of the array, because doing an inorder traversal over a BST will list off the elements of that BST in sorted order. This is impossible, since there is no $o(n \log n)$-time, comparison-based sorting algorithm. Therefore, no $o(n \log n)$-time, comparison-based algorithm exists for converting a binary heap into a binary search tree. ■

Regrade Policies

We do our best in this course to grade as accurately and as thoroughly as possible. We understand how important it is for your grades to be fair and correct, especially since the graders' comments will be our main vehicle for communicating feedback on your progress. That said, we sometimes make mistakes while grading – we might misread what you've written and conclude that your reasoning is invalid, or we might forget that you proved a key result earlier in your answer. In cases like these – where we've misread or misinterpreted your proof – you're encouraged to contact the course staff and ask for a regrade. We want to make sure that your grade is accurate and will try to correct any errors we've made.

Late Policies

We figure that anyone who's signed up to take CS166 likely has a lot of other fun and exciting commitments going on in parallel with the course. Plus, in a course of this size, it's almost a certainty that someone will have something come up (an illness, a family emergency, an exciting job interview, etc.) that prevents them from finishing one of the assignments on time.

The mechanism we have in place to make this work is the ***late period***. Everyone has ***two*** free late periods they can use on the assignments. Each late period is an automatic 48-hour extension on the assignment. You don't need to tell us you're using a late period – we'll charge one automatically if you submit past the assignment deadline. (As a clarification, because assignment submissions are handled using automated tools, we will defer all decisions of lateness to the tools. And because those tools are powered by computers, which keep accurate time down to the nanosecond, "late" means "$T_{submitted} - T_{due} \geq 0$," regardless of what the left-hand side is.)

Regardless of how many late periods you have left, we won't accept any submissions for any assignment more than 48 hours past the initial deadline. Remember – the TAs have to go and grade things, and it's a real hassle if they don't know how many assignments they'll need to grade for several days after the stated deadline. Plus, this will let us get solutions out to you a lot sooner.

So what happens if you run out of late periods? In that case, you are still welcome to submit the assignment up to 48 hours late. However, if you do, we'll cap your score on that assignment at a 70%. That means that it's better for you to submit late than to not submit at all, but it's even better to submit on time than late.

Each person has an independent bank of late periods available to use. If you and a partner submit jointly and submit late, each of you will get charged one late period. If you still have a late period available to spend but your partner doesn't, your grade on that assignment will be whatever it normally would be, and your partner will have their score capped at 70%. That is, the fact that they're out of late periods won't impact you. However, you should definitely plan on having a frank conversation with your partner if they're out of late periods and you want to submit late, as that imposes an asymmetric burden on them.