CS166 Handout 03
Spring 2023 April 4, 2023

Assignment Policies

This handout contains information about the assignments and individual assessments for CS166. Specifically, it contains

- ***submission instructions*** so you know how to turn in the problem sets;
- our ***late policies*** for work submitted past the deadline;
- our ***EdStem policy*** for asking questions online;
- our ***collaboration policy*** with information about working in pairs;
- our ***solution 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.

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

Submission Instructions

This quarter, we will be using GradeScope to handle written assignment submissions. By virtue of being enrolled in CS166, you should have access to the course GradeScape page.

GradeScope only accepts electronic submissions. 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.

Programming questions are submitted separately from 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.

Late Policy

All problem sets in this class is due at noon Pacific time on the day it's due. Each student has three "late days" that they may use throughout the quarter. Each late day automagically extends an assignment deadline by 24 hours. You may use at most two late days on any one assignment, and they're charged automatically; you don't need to get our advance approval before using them. Just submit late.

No work may be submitted more than 48 hours after the stated deadline without prior approval by the course staff. If you submit an assignment fewer than 48 hours late but have run out of late days, your score on that assignment will be capped at 70%. That means it's still better to submit late than not at all.

Late days are tracked per student rather than per pair. So, for example, suppose you and your partner submit an assignment 12 hours late. You have a late day left and your partner does not. You'll then be charged that late day and will receive the full score for your assignment, but your partner will be capped at 70% on the assignment score.

Late days, once used, can't be shifted to other assignments.

EdStem Policy

We have an EdStem forum where you can ask questions on the problem sets 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 assignment 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. We grade assignments uniformly regardless of whether they're submitted individually or jointly. 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, ***submit 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.

Coding Expectations

You're expected to write beautiful, well-commented, well-decomposed code. This is both to make it easier for you to debug and so that it's easier for the TAs to review your code. 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:

- ***Comment your code***. It is difficult to read code someone else wrote if that person didn't leave comments 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.
- ***Decompose problems***. 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.
- ***Use clear variable names***. There's a tension between the mathematical side of things, where single-letter variable names are the norm, 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.
- ***Remove dead code*** It's fine to sprinkle some `cout` or `printf` statements throughout your code when you're testing things. It's also reasonable to comment out an old implementation if 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.
- ***Don't be cute, unless you need to***. You'll be writing code in C++, where it's possible to write both elegant code and obfuscated labyrinths of twisted logic. Aim for clarity above efficiency unless there's a compelling reason not to. For example, please divide by two by writing `value / 2`, not `value >> 1`, unless there's a specific reason that bit-shifts more clearly express your intent. You 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.

Here's an example of a piece of code that we would consider to be totally beautiful. It's an implementation of binary search over an array:

```
/* Returns whether the element key exists in the sorted vector elems in the
 * index range [low, high). Note that low is inclusive, while high is exclusive.
 */
bool binarySearchRec(const std::vector<int>& elems, int key,
                     size_t low, size_t high) {
    /* Base case: If the range is empty (low >= high), the key does not exist in
     * the range [low, high).
     */
    if (low >= high) return false;

    /* Probe the middle element. The use of low + (high - low) / 2 is to avoid
     * integer overflow that could result from computing (low + high) / 2.
     */
    size_t mid = low + (high - low) / 2;

    /* If the element is at the midpoint, we've found it. */
    if (key == elems[mid]) return true;

    /* Otherwise, discard half the elements and search
     * the appropriate section.
     */
    if (key < elems[mid]) {
        return binarySearchRec(elems, key, low, mid);
    } else {
        return binarySearchRec(elems, key, mid + 1, high);
    }
}

bool binarySearch(const std::vector<int>& elems, int key) {
    return binarySearchRec(elems, key, 0, elems.size());
}
```

Proofwriting Expectations

You're expected to write proofs clearly and lucidly. Your proofs should not just convey the mathematical argument; they should 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.

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, each of which takes time $O(\log n)$, for a net runtime of $O(n \log n)$. 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 exists for constructing a BST from a binary heap. 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. ■

Algorithm and Data Structure Expectations

If you're designing and analyzing an algorithm or data structure, you should present the algorithm in the clearest way that you can. Here are our recommendations for how to do this:

- ***Start at a high level, then go deeper***. One of the most effective ways to convey a complex algorithm or data structure is in stages. Begin by detailing the idea at a very high level, then make a second pass over the approach and give more details, and finally drop down to low-level specifics. This approach lets the reader get context for the overarching idea behind your solution, then get more and more clarity on the approach as you go.
- ***Use pseudocode only as a last resort***. It is hard to understand an algorithm or data structure purely from pseudocode – ask anyone who's TAed CS161 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. You should only use pseudocode if it is absolutely necessary to convey an idea.
- ***Use the runtime analysis to fill in details***. Some data structures or algorithms are based on a reasonable, intuitive idea that makes sense at a high level but gets trickier as you get into the details. One way to make these structures easier to explain is to describe the high-level operation of the data structure ("concatenate these lists," "find the third-largest element in the binary heap", etc.), and then to only talk specifics in the runtime analysis. For example, you might describe at a high level how you'll maintain a collection of lists, then in the runtime analysis reveal that the lists are circularly, doubly-linked lists that store pointers to their maximum elements. Only divulging that detail in the runtime analysis keeps the discussion easier and leaves the tricky implementation details to the spots where they matter.

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).

If you haven't encountered this before, it's a great problem! Take a minute to think through it, then check the next page for an example of a writeup that we think is at the appropriate level of detail.

Here's one way to write up a solution to this problem:

At a high level, the data structure consists of two binary heaps that each store roughly half of the elements. The first heap is a max heap that stores the smallest half of the elements, and the second heap is a min heap that stores the larger half of the elements. The median of the values can then be found by looking at the tops of the two heaps.

More specifically, we'll maintain two heaps called *left* and *right*. The *left* heap is a max-heap that will store the smallest $\lfloor n/2 \rfloor$ elements in the data structure, where n is the total number of elements inserted. The *right* heap is a min-heap that stores the largest $\lceil n/2 \rceil$ largest elements in the data structure.

To implement the operation `insert(x)`, we compare x against the tops of the two heaps. If x is less than or equal to the top of the *left* heap, then we insert x into *left*. Otherwise, we insert x into *right*. This ensures that all the elements of *left* are less than or equal to all the elements of *right*, but may result in there being too many elements in one of the two heaps. If this happens, then we either remove the maximum element from *left* and add it to *right*, or remove the minimum element of *right* and add it to *left*. Doing so doesn't change the fact that all elements of *left* are less than or equal to the elements of *right* (since we've either moved the largest element of *left* to *right* or the smallest element of *right* to *left*), and ensures that the two heaps have the right number of elements.

(An edge case we need to handle: if either heap is empty, we insert into *left*. If the data structure was previously empty, this results in 1 element in *left* and 0 elements in *right*, matching the expected counts. If the data structure was not empty, then we must have had 1 element in *left* and 0 in *right*; otherwise, there would be an element in *right*. After adding to *left* we have 2 elements in *left* and 0 in *right*, and we can move an element from *left* to *right* as above.)

To implement the operation `find-median()`, we consider two cases. First, it might be the case that the number of elements n in the heap is even. In that case, the median value is the average of the two elements closest to the center were the elements to be written in sorted order. Since in this case both *left* and *right* contain exactly $n / 2$ elements, any element of *left* that's greater than or equal to all the other elements of *left* could appear just before position $n / 2$ in the sorted sequence, since that element is greater than or equal to half the elements (the elements of *left*) and less than or equal to half the elements (the elements of *right*). A similar argument establishes that any element less than or equal to all the elements of *right* could appear just after position $n / 2$ in the sorted sequence. Therefore, we can compute the average over the maximum element of *left* and the minimum element of *right* to get the median.

On the other hand, if n is odd, then the maximum element of *left* is the median. To see this, note that if $n = 2k+1$, then there are $k+1$ elements in *left* and k in right. The maximum element of *left* is then greater than or equal to k elements (the smaller elements of *left*) and less than or equal to k elements (the elements of of *right*), so it's the median.

Looking at the runtime: each `insert` call does a heap enqueue, followed possibly by a heap dequeue and second enqueue. These heaps each have size $O(n)$, so this takes time $O(\log n)$. Each call to `find-median` looks at the tops of at most two heaps, which takes time $O(1)$.

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. ***Regrade requests must be received within one week of the graded work being returned to you***.