General Tips:

- Practice early so you don't cram the day before! A clear head is essential to doing well on a CSE 143 exam, as there will be a lot of logic involved.
- Practice-It and past final exams posted on the course website are perhaps the best resources for studying. Make sure to write your solutions on paper first to mimic the setting of the exam (give yourself 60 minutes for a printed practice exam, without notes). It's good to practice testing your code by hand (on paper) since you won't have Practice-It to tell you 20/20 on the exam.
- As you practice, write down your 2-3 most common mistakes for each category. This will **really help** as you'll prepare yourself to avoid what you'll most commonly make a mistake on!
- Which problems are you avoiding, cringing at the thought of returning to? Do those. You don't want to get to the test without feeling any more confident about them.
- It can be helpful to have go-to strategies to have when you're stuck. Having a go-to (even if it's not end-all be-all) can help you feel more comfortable and get you started in the right direction without wasting too much time.
  - One common strategy includes drawing additional example inputs and considering what the return/result should be (case analysis - try simple inputs and complex ones!). By understanding what the method does, you'll have a more intuitive grasp on the overall idea and may start seeing some more patterns.
  - It might also be useful to reread the prompt / come back to the problem with a fresh mind later to try and find another hint in the problem statement that will send you in the right direction. Active reading with pens (circling, underlining, highlighting, etc.) can help you catch/remember important details, such as any important ordering, exceptions to throw, specified parameters and returns, data structures/methods not allowed, empty cases, etc.
- Give yourself space between statements to account for any last-minute fixes/special cases.
- Make sure to review the exams information page! There are a number of points to keep in mind when studying. The exam cheat sheet will also be useful to familiarize yourself with.
- Writing something is better than nothing. Most programming problems have some form of partial credit so even if your solution isn't complete, you may receive points for parts of your method that would also appear in a correct solution.

Common in-the-moment mistakes

- Forgetting proper interfaces (e.g. with `Queues`)
- Mixing up size/length methods and whether to include parentheses:
  - `String word : word.length()` - *method*
  - `List<Integer> names: names.size()` - *method*
  - `int[] numbers : numbers.length` - *field*
- Forgetting to declare a variable before using it.
- Forgetting to return (in all cases) what you've worked so hard to solve (for methods with returns).
- Forgetting to "build" an answer during a recursive case, and ultimately only returning your base case.
- Reversing order of things (especially common with recursion returns and Stacks/Queues)

Tips for Different Problem Types:

ArrayIntList

- You're writing a method that would be inside the `ArrayIntList` class (see Section 02 handouts)
- Use **only** the fields/methods given to you (you probably won't be able to use any methods other than the ones you write)
- Start with one case (front, middle, empty, or end). Then approach other cases. The bullet points will probably help start you off with some cases (ie. you may assume that your list isn't null)
- It's been a while since we've covered `ArrayIntList` programming problems, so don't underestimate the value of practicing a few of these problems!

Stacks and Queues

- We haven't practiced these in a while, and students often underestimate the difficulty of Stack and Queue midterm problems. Make sure to practice these especially!
- You almost always need to use an auxiliary data structure (often the opposite of the original one you are working with). In many problem descriptions the required structure will be specified.
- Declare the auxiliary before you do anything else (unless there's an exception or special case to account for)
- Make sure to use Integer and Character capitalized names in your Stack/Queue (if storing ints or chars)
- Remember interfaces: e.g. `Queue<Integer> q = new LinkedList<Integer>();` `Stack<Integer> s = new Stack<Integer>();`
- Double check that you use the correct methods, many students call `push` on a queue or `remove` on a stack
- Remember to use stack-to-queue and queue-to-stack while loops to make sure you preserve original order (one round of queue to stack and stack to queue will reverse the order of the original queue, so you'll need to do it a couple of times). Use a cycle diagram to keep track of order!
- Make sure you don't call `.pop()` on an empty Stack, or you will get an exception. Sometimes you may have to surround large parts of your code with a conditional to avoid this.

LinkedList Before and After

- **Draw arrows!** Solve each problem in this way, make sure that you don't use any nodes and use temp ListNodes as necessary. Recall how to make a temp (take care not to create a new node unless a new one appears in the After diagram):
  - `ListNode temp = list.next (example)`
- For each arrow you draw in this process, write the number corresponding to the step
- Transfer your steps to code, looking at what you are doing at each numbered step
- This will really help make sure that you don't lose nodes anywhere and you remember to move everything around
- Remember to set any last nodes in the list to null if they are still pointing to another node
- **There will be no LinkedList programming problems on the midterm. So just practice the before/after problems :)**

Recursive Tracing

- Practice recursive tracing and find a diagram method that helps you the most. It will help you save time and points when you have a process you can count on and are familiar with (e.g. the arrow-numbering method used in section)
- If you can find a pattern, use it to solve longer problems more quickly, but make sure it is really the right pattern first
- Make sure you are comfortable working for different types of recursive problems (ones that return vs. ones that output, ones that works with ints vs. ones with strings, etc.)

Recursion Programming

- Do you have a base case? Recursive case? (you may have more than one of each)
- Is your base case the simplest case possible? Do you build your answer with your recursive case (if needed)?
- Run through your code after you finish (if you have time) to make sure you have the correct output (reverse output or only base case output is a common error)
- If you have no idea how to start, try the leap of faith trick, and see what your output would be.
- When practicing these problems, try to use a visualization strategy for each. Recall examples you may have seen in lecture/section:
  - The column-based example which worked well with symmetrical output
  - The tree-based approach that worked will with two recursive calls in one recursive case
  - The linear method chain approach which worked well when we percolate a result back up the "call chain", possibly performing a calculation on the sub-call result before returning it back up. This can help you have a go-to visualization when approaching a challenging recursive programming problem on the exam.

Collection Programming

- Make sure you're familiar with Map and Set methods on the exam reference sheet - if you forget the syntax of the return or parameters, use the sheet!
- Make sure to review how loops work with Iterators, how to add/remove with Iterators, the difference between TreeSets vs. HashSets, how to use a Map with a List or Set as its value type, etc. - any of these types of problems may be included. Recall that an Iterator is just an object that lets us traverse elements of a Collection, and access/remove elements while looking at one at a time.
- For Maps, make sure you're familiar with keys and values, and be able to handle the case of reversing keys to values, counting number of unique values, etc. Refer to Section 9 for different problem types!
- Remember to identify whether you're using a TreeMap or HashMap. If the keys must be sorted, you have to use a TreeMap. Same goes with Sets.
- For problems using Maps, compare the differences between adding to a Map that has a data structure as its value type (e.g. Set<Integer>) and one that doesn't (e.g. Integer)
  - A common pattern with nested data structures / maps: a special case for the first time you put something in the map. ex: if (!containsKey) then put the key with a new ArrayList, TreeSet, etc. See problems like `groupExchanges` from section.
- It can be tricky to correctly identify the types of the parameters and returns if you're not careful. Be sure to find that info in the problem statement and circle it before you begin. Otherwise, you may be solving a different problem.
- Remember that Sets don't have indices, so usually you'll want to use a for-each loop (or Iterator if you're modifying the Set)