Summary of exam 2 practical section

In the practical portion of the exam, you will be given a pool of problems to work on individually. At the time the test is administered, the instructors *may* insist that all students attempt a specific problem, or one problem from a specific subset of problems; otherwise, each student may work on problems of his or her choice from the pool.

Points

Basic value

Each problem will have a specified point value; completing all of the required portions of a problem will earn that number of points.

In the summary below, problems with an indicated value of **low** will be worth 10–20 points on the exam. Problems in the **medium** value range will be worth 20–30 points. **High**-value problems are worth 30 or more points. (None of these ranges include any extra credit that may be available for a given problem.)

Unit tests

Most of the problems will have a unit testing component, included in the requirements for earning the stated value. A set of test cases will be provided for each of these problem. However, we reserve the right to include additional test cases beyond those in our grading process; any such additional cases will be in-line with the functional requirements given in the problem description.

Partial credit

Partial credit is available for all problems, and will be awarded based on the instructors' assessment of the work completed.

Please note: When examining code for the purpose of awarding partial credit, an instructor is (essentially) debugging the student's code, to assess how close that code is to solving the problem; this is almost always much easier to do on code that compiles than on code that doesn't compile.

Extra credit

Some problems will have extra credit portions, which can be completed for a specified point value.

In some (not all) cases, it will be possible to do *some* of the extra credit work without completing all elements of the basic problem. You won't be penalized for doing this—but it will generally be impossible to complete *all* of the extra credit portion without also completing all of the basic problem.

Completing the extra credit portion of a problem may (and often does) require that you modify the code you've already written for the basic problem. Thus, we strongly recommend that you commit (and, in general, push) your solution to the basic problem before attempting the extra credit portion of a problem.

Total points possible

This portion of the exam has a nominal value of 50 points; by completing more problems and/or extra credit portions of problems, a student may earn more than that. A maximum of 75 points will be awarded in this portion of the exam.

Submitting your work

The only work that will be graded is the work committed to Git **and** pushed to GitHub. **It is each student's responsibility to make sure that their work has actually been pushed to GitHub.** To that end, we *strongly* recommend you review the commits to your repositories in https://github.com/ddc-java-16 *well* before the test close time—which is 12:15 PM MST, 10 November, 2023.

Collaboration & independent work

Prior to the official start of the exam (7:45 AM MST, 10 November, 2023), we encourage you to do research online and collaborate with your classmates to work out approaches to the problems summarized below. During the exam, you're welcome to copy and paste code from work you've done up to this point in the class, code that you've written with your classmates in preparation for the exam, code you've found online related to an exam problem, or even code generated by an AI coding assistant. However, during the exam itself, *real- or near-real-time collaboration with others (in or out of the bootcamp) is not allowed.*

Use of relevant social networking sites (e.g., StackOverflow, Reddit) during an exam is allowed, *to an extent*: you may use them to search for solutions to similar problems—but during the exam, you may not engage in real- or near-real-time communication with other users of the sites for the purpose of advancing your work on the exam.

Simply stated, the exam is open-book, open-Internet, open-JShell, open-IDE; but it is *not* open-classmate, open-online-colleague, open-family-member, etc.

Problem summaries

Please use the summaries below for study, review, and practice coding, in preparation for the exam. However, do not assume that any code you write in your preparation will be suitable for dropping directly into the problems during the actual test: among other potential issues, the exam problems will specify method signatures, return types, and test cases that are not shown below. Similarly, extra credit opportunities aren't summarized here, beyond a simple "yes" or "no" for each problem.

Also, there is no guarantee that all of the problems summarized below will appear on the exam—nor is there a guarantee that the exam won't include additional problems. You should anticipate, however, that there will be a significant overlap between the problems here and those on the exam.

Next perfect square

- **Value**: Low
- **Unit tests**: Yes
- **Extra credit**: Yes

Perfect squares are positive integers which are the product of some integer multiplied by itself. For example, $1 = 1^2$, $4 = 2^2$, $9 = 3^2$, $16 = 4^2$, etc. Thus, $1, 4, 9, 16, \ldots$ are perfect squares.

Your task is to implement a method that takes a `long` input parameter, and—if the value of that parameter is a perfect square—returns the *next perfect square higher than the input value*. For example, $225 = 15^2$, and $256 = 16^2$, so the next perfect square after 225 is 256.

Please note: This is **not** the same problem that was used in preparation for exam 1. In this problem your task is not simply to say whether an input parameter value is a perfect square, but rather to say what the next perfect square *higher* than than the input value is—if the input value is itself a perfect square.

Sample casing

- **Value**: Medium
- **Unit tests**: Yes
- **Extra credit**: No

In this problem, you will need to implement both a constructor and a method in a class:

- The constructor takes a parameter that is a *sample* `String`, with zero or more characters. The casing of these letters dictate the behavior of the method you must implement (next).
- The method takes a single `String` input, and—based on the casing of the letters in the input to the constructor—returns a new `String` with possibly modified letter casing.

For example, if the sample casing `String` (passed to the constructor) is `"BaC"`, then the `String` returned by the method must not contain any lowercase `'b'` characters (since that letter is in uppercase in the sample), nor any uppercase `'A'` characters, nor any lowercase `'c'` characters. Any other characters in the input to the method must be included in the returned value without any changes to their casing.

Continuing the example, if the class was instantiated with a constructor argument of `"BaC"`, and the method is invoked with an argument of `"Algebra is crazy useful."`, the method must return `"algeBra is Crazy useful"`—that is, all of the lowercase `'b'` characters in the input have been converted to uppercase

`'B'` in the returned value, all of the uppercase `A` characters in the input have been converted to lowercase `'a'`, and all of the lowercase `'c'` characters have been converted to uppercase `'C'`.

Array slicing

- **Value**: Medium
- **Unit tests**: Yes
- **Extra credit**: Yes

Given an `int[]`, return a specified *slice* of the array. Rather than a simple sub-array, a *slice* contains the elements starting from `beginIndex`, up to (but not including) `endIndex`, skipping elements according to a *step* or *stride* parameter value (`stride`).

For example, a slice of `{1, 1, 2, 3, 5, 8, 13, 21, 34, 55}`, with a `beginIndex` value of 1, an `endIndex` of `9`, and a `stride` of 2, would return `{1, 3, 8, 21}`—that is, the array containing the elements from the original array starting at position 1, advancing by 2 positions in each step (thus, including values from positions 1, 3, 5, etc. of the original), including each such value up to, but not including the element in position 9.

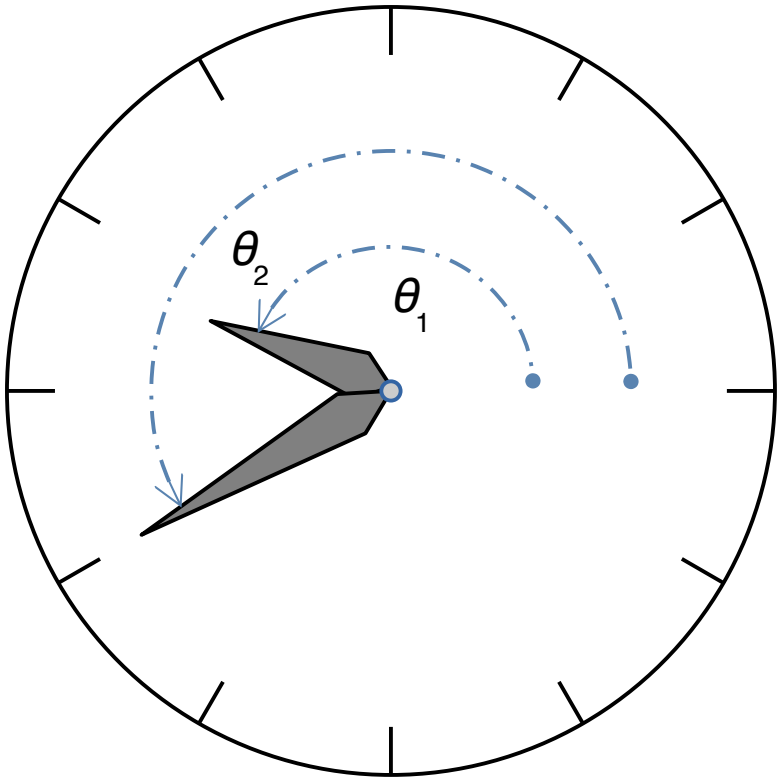
Clock angles (radians)

- **Value**: Low
- **Unit tests**: Yes
- **Extra credit**: Yes

(Sound familiar? Maybe so—but best not to jump to conclusions.)

Given an integer number of hours between 0 and 23, (inclusive), and a floating-point number of minutes in the half-open interval $[0, 60)$ (i.e. including 0, but not including 60), what angle in **radians** is measured **counter-clockwise** from 3 o'clock to the hour hand? What is the angle between 3 o'clock and the minute hand? So, this problem is indeed similar to the clock angles problem in exam 1—but instead of returning angles measured as on a compass (clockwise degrees), this time we need to return the angles we deal with much more frequently in programming: counter-clockwise radians, starting from the positive X-axis.

In the example below, the time is 9:40 AM or 9:40 PM (both appear the same on a 12-hour clock face). The angle to the hour hand is θ_1; in radians, it measures approximately 2.7925. The angle to the minute hand is θ_2, which measures about 3.6652 radians.



Please note that this problem can be solved without using trigonometric functions. Instead, the main thing you need to keep in mind is that there are 2π radians in one trip around a circle; moving part-way around a circle traces out an angle proportionate to the given fraction of the circle.

Social security number (SSN) formatting and parsing

- **Value**: Medium
- **Unit tests**: No (see below)
- **Extra credit**: Yes

For full credit on this problem, you must complete the implementation of methods to perform the following:

- Format an `int[]` (presumably containing 3 elements, each) as a social security number (SSN) `String`, using the `'-'` delimiter between digit groups, and with leading zeros as necessary, to form the required digit groups of 3, 2, and 4 digits each.

  For example, the `int[]` containing the values `{16, 64, 512}` would be formatted and returned as `"016-64-0512"`

- Parse an SSN from a `String` to an `int[]`, splitting the input `String` on the `'-'` (or possibly other) delimiter characters.

This problem will not require you to write unit tests; instead, there are already unit tests written—several of which are failing. Correct implementation of the two methods summarized above will be required in order to pass all of the unit tests.

Android temperature conversion

- **Value**: High
- **Unit tests**: No
- **Extra credit**: Yes

For this problem, you must build a single-activity Android app, essentially from scratch, with the following features:

- 3 editable fields for Fahrenheit, Celsius, and Kelvin temperature values.
- A text prompt or hint with each field.
- Buttons for converting any of the 3 values to its equivalent values in the other 2 scales, and filling in the fields accordingly.
- All elements positioned according to detailed specifications.

You will not be required to follow the Google- or DDC-recommended architecture; in particular, you could write all of the code in a single activity class (along with a corresponding resource file for the layout), without fragments, viewmodels, repositories, etc.