Contents

Summary of exam 3 practical section

In the practical portion of the exam, you will be given a pool of problems to work on individually. 0 or 1 (our choice) of the problems will be assigned; everyone will be expected to work on any such assigned problem, and submit some work in full or partial completion of the problem; each student may also work on problems of his or her choice, in addition to any assigned problem.

Problem point value

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.

Unit tests

Some 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 in our grading process, beyond those that we include in the problem description; 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.

Extra credit

Some problems will have extra credit portions, which can be completed for a specified additional point value. Additionally, if a student completes more problems than are required to earn 50 points, the additional work will be counted as extra credit.

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.

In general, the summary descriptions below do not include details on the extra credit opportunities for each problem; these will be provided at the time of the test.

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 can be evaluated for grading purposes—even for partial credit—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. Do not assume that just because you have no uncommitted changes, all of your commits have been pushed to GitHub!** (To be sure of this, we recommend you review the pushed commits to

your repositories in https://github.com/deep-dive-coding-java-cohort-6 before 5:30 PM MDT on 11 April.)

Collaboration & independent work

You are welcome to collaborate in working out approaches to the problems summarized below. However, the work you complete and submit in the actual exam will be your own code: the exam is open book, open Internet, open IDE—but not open classmate.

Use of Slack or other messaging services—whether on your laptop or a phone—is not allowed during the test. The only exception to this rule is when the instructors publish information on the test (which may be updated as you are taking the test) on Slack.

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. Additionally, extra credit opportunities are described minimally, if at all.

Spatial sorting

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

When we sort primitive numeric values, or `String` instances, our common sense regarding *natural order* (typically ascending numerical or lexicographic—or alphabetical— order) usually serves us well. Additionally, the sorting methods of the `Arrays` and `Collections` classes provide easy-to-use facilities for natural-order sorting.

However, our concepts of natural order doesn't necessarily give us much to go on when sorting objects composed of multiple fields. Even if we're sorting (for example) objects representing points in 2-dimensional or 3-dimensional space, we might sometimes sort using primarily the X value, then the Y value, etc., while other requirements may dictate that we sort primarily in Y, then X, and so on. We may even sort by the distance from the origin, angle from the positive X-axis or plane, etc.

Fortunately, whatever the desired order, we can still use the sorting methods of the Java standard library in most cases, simply by implementing the `Comparable` interface in the class defining the instances to be sorted, or by implementing the `Comparator` interface (which we can often do with a lambda).

In this problem, your task will be to implement either `Comparable` or `Comparator`, as appropriate, in order to sort an array of 2- or 3-dimensional points, in a specified order.

Padovan sequence

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

The Padovan sequence resembles the Fibonacci sequence, in that each term is the sum of 2 terms appearing earlier in the sequence. However, while the Fibonacci sequence uses the sum of 2 adjacent terms to generate the term immediately following, the Padovan sequence uses the sum of 2 adjacent terms to generate the term that immediately follows the term immediately following the 2 that we add together.

In other words,

$$P = p_0, p_1, p_2, \ldots$$

where

$$\begin{aligned} p_0 &= 0, \\ p_1 &= 1, \\ p_2 &= 1, \\ p_n &= p_{n-3} + p_{n-2}, \text{for } n > 2. \end{aligned}$$

This recurrence relationship produces the sequence

$$P = 0, 1, 1, 1, 2, 2, 3, 4, 5, 7, \ldots$$

Your task in this problem will be to write code that produces the n^{th} item of the Padovan sequence, where n is a method parameter. For extra credit, your code will need to implement the `Iterable` interface, so that the first n terms of the Padovan sequence may be used in an *enhanced for* loop.

Zero-sum doubles

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

A common programming task is to identify subsets of values in a data set that satisfy some condition. This problem is a simple instance of that type of task.

You will start with a method that is declared with an `int[]` parameter, assumed to contain *distinct* values. Your task will be to complete the implementation of the method, so that it returns a list of all `int[]` pairs (i.e. arrays of 2 elements), where the elements in each pair are values in the input parameter, and the sum of each pair is 0.

For example, if the input parameter refers to the array

```
{1, 0, -2, 4, -3, -1, 3}
```

your method should return the list of pairs

```
[{1, -1}, {-3, 3}]
```

Your method might return this list in a different order than that shown above, and the order of elements in any given pair might be reversed from that shown above. However, no pair should appear twice in the list, even if the order is reversed from one appearance to the next. So, in the above example, the list returned should not contain both `{-3, 3}` and `{3, -3}`.

Zero-sum triples

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

A slightly more difficult version of the problem above—and one more often used in programming challenges—is that of finding *zero-sum triples*—combinations of 3 values from an input data set, where each such combination sums to 0.

The basis structure of this problem is the same as that above, except that the required output is a list of `int[]` triples.

For example, taking the same input array as before,

```
{1, 0, -2, 4, -3, -1, 3}
```

your method should return the list of triples

```
[{1, 0, -1}, {0, -3, 3}, {4, -3, -1}, {-2, -1, 3}]
```

Lawn mowing patterns

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

Your employer has purchased a programmable lawnmower. Your task is not to program it, however, but to determine whether a desired pattern of grass heights can be mowed, subject to the limitations of the lawnmower. In short, these limitations are:

- The lawnmower can only be used on rectangular lawns.
- It can only mow in straight, rectilinear (i.e parallel or perpendicular to the rectangle's sides) paths that go directly from one of the four sides of the lawn directly to the opposite edge.
- The height of the cutting blade can be set prior to each straight-line pass over the lawn; it *can't* be changed in the middle of a mowing pass.
- The width of each mowing pass is 1 meter.

Your specific task will be to implement a method that takes an `int[][] heights` (i.e. a 2-dimensional array of `int` values) as an input parameter. These represent the desired grass heights (in cm) on 1 m2 patches of lawn, whether the total dimensions of the lawn (in meters) are `heights.length` X `heights[0].length`.

Your code must return a `boolean` value from the method, indicating whether the lawn can be mowed to the desired heights, subject to the above restrictions.

For example, we might have

```
int[][] heights = {
    {2, 1},
    {1, 2}
}
```

This represents a 2m X 2m lawn (pretty small), where the desired heights form a checkerboard pattern of 1cm and 2cm. However, since the lawnmower makes complete passes from one side of the lawn to the other, we can easily see that there's no way to cut both the NE and SW patches to a height of 1cm, without also cutting either the NW patch or the SE patch to a height of 1cm as well.

On the other hand, we could have

```
int[][] heights = {
    {2, 1, 2},
    {1, 1, 1}
}
```

This pattern *is* feasible, with three passes of the lawnmower:

1. From NW to NE, at a height of 2cm.
2. From SW to SW, at a height of 1cm.
3. From N to S, in the center, at a height of 1cm.

Parsing Roman numerals

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

In this problem, you must write a method that parses a `String` containing Roman numerals and returns an `int`, according to the rules at https://www.factmonster.com/math/numbers/roman-numerals.

For extra credit, the overbar character (described in rule 4 of the page referenced above) must be supported.

JavaFX debugging

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

Quite often, a working programmer (especially at the entry level) may be given the task to fix some code base that has long been known to have bugs or performance issues, but none of the other members of the development teams has had the time to address.

In this problem, you will be given a repository containing the code of JavaFX-based game containing one or more fatal bugs. Of course, you won't have seen this code before (and you haven't worked much with JavaFX), but you'll be expected to apply the same debugging tools and skills that you've been using in non-JavaFX apps:

- Examining a stack trace available after a crash, to figure out what line of code directly produced the crash, where the method containing that line of code was invoked from, etc.
- Setting breakpoints that allow you to step through the code in the vicinity of the crash conditions.
- Examine the stack frame as you step through the code, to see the values of the local variables, instance fields, and static fields.
- Setting watchpoints to detect unexpected changes in key variables.
- Form and test hypotheses about the underlying cause(s) of the crash.
- Fix the identified culprit code.

Color picker

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

This problem involves the construction (from scratch) of a single-activity Android app, with the following basic UI elements:

- 3 inputs (`EditText`, `SeekBar`, etc.) for levels of red, green, and blue, ranging from 0 to 255.
- A rectangle displaying a color "swatch".

All of the above elements must be positioned precisely according to specifications stated in the problem.

As the input values change, the color filling the swatch region must change to the color made up of the combination of the current red, green, and blue values.

Random password service

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

For this problem, you must implement a Spring Boot-based REST-like service (from scratch) with the following features:

- A single controller class (annotated with `@RestController` and—optionally— `@RequestMapping`), with a single method (annotated with `@GetMapping`).
- The single controller method will be defined with 2 parameters:
  - `String pool`
  - `int length`
- The service must return a `String` containing `length` characters, randomly selected (with replacement) from `pool`.

(Of course, there will be some additional specifications and constraints, provided with the problem description during the test.)