Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

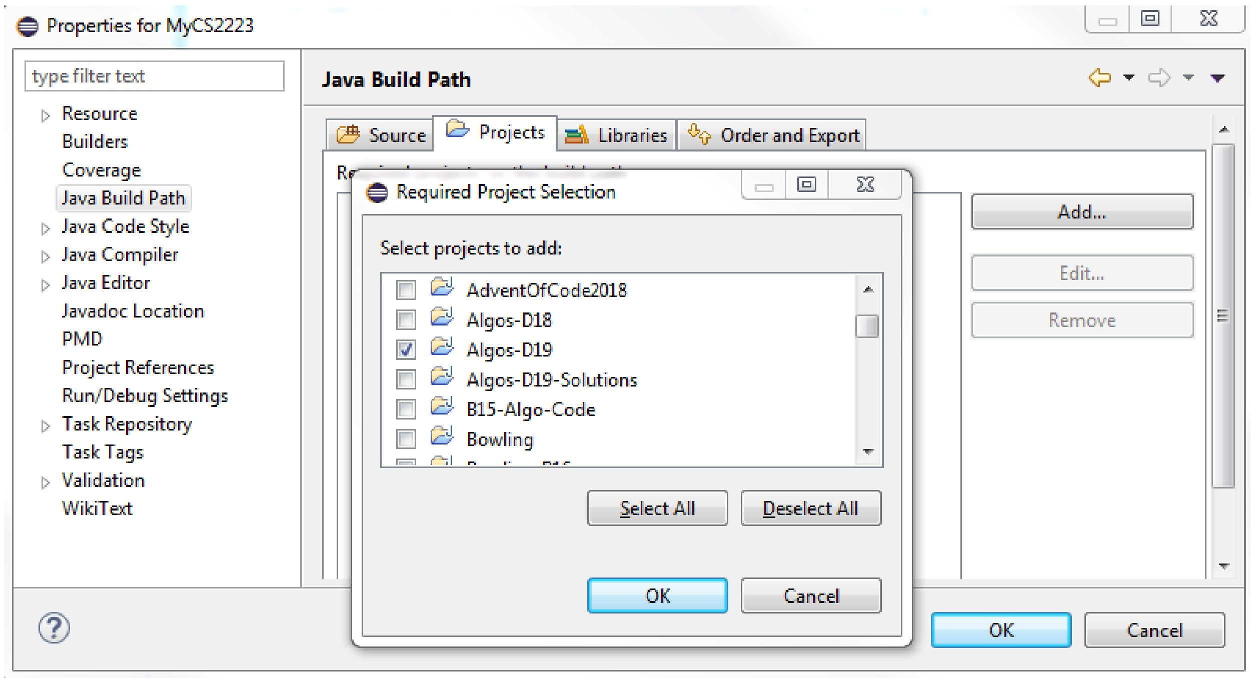
CS 2223 D19 Term. Homework 1 (100 pts.)

Homework Instructions

- This homework is to be completed individually. If you have any questions as to what constitutes improper behavior, review the examples I have posted online http://web.cs.wpi.edu/~heineman/html/teaching_/cs2223/d19/#policies
- Due Date for this assignment is 2PM Friday March 22nd. Homeworks received after 2PM are penalized 25%. Homeworks received after 6PM receive zero credit. Solutions are posted at 6PM.
- Submit your assignments electronically using the canvas site for CS2223. You must submit a single ZIP file that contains all of your code as well as the written answers to the assignment.
- All of your Java classes must be defined in a packager USERID where USERID is your CCC user id.

First Steps

Your first task is to copy all of the files from the Git repository that you will be modifying/using for homework1. First, make sure you have created a Java Project within your workspace (something like MyCS2223). Be sure to modify the build path so this project will have access to the shared code I provide in the **git** repository. To do this, select your project and right-click on it to bring up the Properties for the project. Choose the option **Java Build Path** on the left and click the Projects tab. Now **Add...** the **Algos-D19** project to your build path.



Select projects to add:

- [ ] AdventOfCode2018
- [ ] Algos-D18
- [✓] Algos-D19
- [ ] Algos-D19-Solutions
- [ ] B15-Algo-Code
- [ ] Bowling

Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

Once done, create the package **`USERID.hw1`** inside this project which is where you will complete your work (for the whole term. You likely will have packages for each of the homework assignments). Start by copying the following files into your **`USERID.hw1`** package.

- `algs.hw1.arraysolution.BandedArraySolution` → `USERID.hw1.BandedArraySolution`
- `algs.hw1.arraysolution.RowOrderedArraySolution` → `USERID.hw1.RowOrderedArraySolution`
- `algs.hw1.arraysolution.SpiralArraySolution` → `USERID.hw1.SpiralArraySolution`
- `algs.hw1.q4.Computation` → `USERID.hw1.Computation`
- `algs.hw1.Evaluate` → `USERID.hw1.Evaluate`
- `algs.hw1.WrittenQuestions.txt` → `USERID.hw1.WrittenQuestions.txt`

In this way, I can provide sample code for you to easily modify and submit for your assignment.

This homework has a total of 104 points. If you do all bonus work (do not attempt until completing the full homework!!) you can earn an additional four points. Note that the amount of work to complete the bonus points is not proportional to the few points that you will achieve.

Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

Q1. Stack Experiments (30 pts.)

On page 129 of the book there is an implementation of a rudimentary calculator using two stacks for expression evaluation. I have created the `Evaluate` class which you should copy into your **`USERID.hw1`** package. Note that all input (as described in the book) must have spaces that cleanly separate all operators and values. Note 1.2 has a space before final closing ")".

1.1. **(4 pts.)** Run this program on input "`( 2 + 3 + 4 )`"
1.2. **(4 pts.)** Run this program on input "`( 9 - - 2 )`" (there is a space between the minus signs)
1.3. **(4 pts.)** Run this program on input "`- 99`" (there is a space between the minus sign and the 9).
1.4. **(4 pts.)** Run this program on input "`( 2 * 3 + 7 / 4 )`"
1.5. **(4 pts.)** Run this program on input "`( ( ( 3 * 7 ) + ( 8 * 2 ) ) / 8 )`"
1.6. **(5 pts.)** Modify `Evaluate` to support two new operations
   a. Add a new modulo operation "A % B" that computes the remainder when dividing A by B. Note this also works for floating point values, thus "5.5 % 1.5" should equal 1.0
   b. Add a new floor operation **ceiling** which computes the smallest integer greater than or equal to x. This operator is a unary operator (like **sqrt** for square root). The *Math.ceil(double)* method will be useful for you.
1.7. **(5 pts.)** Run your program on input "`( ceiling ( 8.625 % 3.5 ) )`" and be sure to explain the result of the computation in your `WrittenQuestions`.txt file.

For each of these questions (a) state the observed output; (b) describe the state of the **ops** stack when the program completes; (c) describe the state of the **vals** stack when the program completes.

Note: If, to an empty stack, you push the value "1", "2" and then "3", the state of this stack is represented as ["3", "2", "1"] where the top of the stack contains the value "3" on the left, and the bottommost element of the stack, the value "1", is on the right. An empty stack is represented as [].

Write the answers to these questions in the "WrittenQuestions.txt" text file. For question **1.6**, modify your copy of the **`Evaluate`** class and be sure to include this revised class in your submission.

Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

Q2. ArraySearch Programming Exercise (30 pts.)

You are given an *nxn* two dimensional (2D) square array of unique, positive integer values. Each value, *a[r][c]* in the array, is greater than 0 and smaller than or equal to n^3. You can inspect the value of any cell in the array by calling *inspect(r, c)* where *r* is the desired row ($0 \leq r < n$) and *c* is the desired column ($0 \leq c < n$). **Without knowing any information about the way values are stored in the array**, the following *locate* method will identify the row and column of a desired target value (if it exists in the array) or simply return **null** if it can't be found.

```
public int[] locate(int target) {
  int n = this.length();
  for (int r = 0; r < n; r++) {
    for (int c = 0; c < n; c++) {
      if (inspect(r,c) == target) {
        return new int[] { r, c };    // return (row, col) where found
      }
    }
  }
  return null;  // not found
}
```

In the best case, the target value you are looking for is in (row=0, col=0) and so only one array inspection is required. In the worst case, the target value is not in the array and you have to inspect all n^2 values.

If you execute **algs.hw1.arraysolution.UnknownArraySolution**, it will create a sample 13x13 array and run a trial to look for all numbers from 1 to n^3; in this case, *n*=13, so it will call *locate* on the 2,197 values from 1 to 2,197. As you can see from the program output, it requires a total of 357,907 inspections of the array to locate each value (that is, find its location in the array or determine it is not present).

If you modify the `main` method in **UnknownArraySolution** to create a 3x3 array, the trial searches for 27 values; create a 5x5 array and it searches for 125 values. Below are the two arrays that are created:

	0	**1**	**2**
0	3	5	7
1	9	11	13
2	15	17	19

	0	**1**	**2**	**3**	**4**
0	5	9	13	17	21
1	25	29	33	37	41
2	45	49	53	57	61
4	65	69	73	77	81
5	85	89	93	97	101

As you can see, the smallest number (upper left corner) is *n*, the largest value is $n^2 * (n-1) + 1$ and the difference between subsequent numbers is $n - 1$. On a 13x13 array, trial requires 357,097 inspections. I am asking you to write several more efficient *locate* methods if you know the *nxn* array has a specific structure. Draw inspiration from **BINARY ARRAY SEARCH** presented in class for searching for a value within a one-dimensional array of sorted values. Your *locate* method must call *inspect(r,c)* every time it checks the value of the cell in row *r* and column *c*. This is done to properly count the number of times your code inspects the array. ***Trying to work around this restriction might cause you to lose points on this question.***

Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

Q2.1 RowOrderedArraySolution

A **RowOrderedArray** is a two-dimensional, square *nxn* array of unique, positive integers with the following properties:

1. Each row contains ascending values from left to right.
2. Each of the values in row $0 \leq k < (n-1)$ are smaller than the values in row $(k+1)$

Here is a sample 5x5 **RowOrderedArray**, with each row colored differently.

1	3	9	11	12
13	23	24	25	29
35	36	37	44	47
48	52	54	55	60
63	72	77	78	79

Your goal is to modify **`RowOrderedArraySolution`** to write a more efficient `locate` method that takes advantage of the structure of a **RowOrderedArray**.

Task 2.1 (10 points): Complete `locate` method in **`RowOrderedArraySolution`** and ensure that on a 13x13 array, the sample trial completes with fewer than 20,000 array inspections.

Hint:* BINARY ARRAY SEARCH *can come to your rescue

Q2.1.1 Bonus (+1 bonus point)

Only attempt the bonus point after you have completed the first part. Get an extra bonus point if you achieve (or do better than) `16,394` array inspections on the sample 13x13 array.

Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

Q2.2 BandedArraySolution

A **BandedArray** is a two-dimensional, square *nxn* array of unique, positive integers with the following properties:

1. The smallest value is in the upper left cell A[0][0]
2. The largest value is in the upper right cell A[0][n-1]
3. Each of the upper-left *k*-by-*k* cells ($0 < k < n$) is smaller than any of the other $n*n - k*k$ cells
4. There are k "bands" in the array containing values in ascending order, each starting on one of the left-most cells A[r][0], heading right to A[r][r], then heading up to stop at A[0][r].

Here is a sample 5x5 **BandedArray**, with each band colored a different color.

1	4	9	16	25
2	3	8	15	24
5	6	7	14	23
10	11	12	13	22
17	18	19	20	21

Your goal is to modify **`BandedArraySolution`** to write a more efficient `locate` method that takes advantage of the structure of a **BandedArray**.

Task 2.2 (10 points): Complete `locate` method in **`BandedArraySolution`** and ensure that on a 13x13 array, the sample trial completes with fewer than 20,000 array inspections.

Hint: Can Binary Array Search come to your rescue; this time perhaps think diagonally?

Q2.2.1 Bonus (+1 bonus point)

Only attempt the bonus point after you have completed the first part. Get an extra bonus point if you achieve (or do better than) `17,253` array inspections on the sample 13x13 array.

Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

Q2.3 SpiralArraySolution

A **SpiralArray** is a two-dimensional, square *nxn* array of unique, positive integers where *n* is an odd number. The properties of a **SpiralArray** are as follows:

1. The smallest value is in the innermost cell A[*n*/2][*n*/2]
2. The largest value is in the lower right cell A[*n*–1][*n*–1]
3. All values appear in ascending order in a counterclockwise spiral from the innermost cell to the lower right cell.

Here is a sample 5x5 **SpiralArray**.

17	16	15	14	13
18	5	4	3	12
19	6	1	2	11
20	7	8	9	10
21	22	23	24	**25**

Your goal is to modify **`SpiralArraySolution`** to write a more efficient `locate` method that takes advantage of the structure of a **SpiralArray**.

Task 2.3 (10 points): Complete `locate` method in **`SpiralArraySolution`** and ensure that on a 13x13 array, the sample trial completes with fewer than 10,000 array inspections.

Hint: Can Binary Array Search come to your rescue; again thinking diagonally?

Q2.3.1 Bonus (+1 bonus point)

Only attempt the bonus point after you have completed the first part. Get an extra bonus point if you achieve (or do better than) `7,273` array inspections on the sample 13x13 array.

Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

Q3. Counting Computations Exercise (20 pts.)

To understand the performance of an algorithm, one must count the number of times that key operations are executed. The **UnknownArraySolution** program is concerned with how many times you inspect the array using *inspect(r, c)* while trying to locate numbers from 1 to n^3.

If you modify the main method of **UnknownArraySolution** to change the size of the *nxn* square array under consideration, you will see the following output values:

n	# Array Inspections
3	207
4	904
5	2825
6	7146
…	…
13	357097

Task 3.1 (10 points): Construct a function f(n) that accurately models the number of array inspections required by **UnknownArraySolution** for an *nxn* square array. Confirm you have the proper f(n) by validating that it computes f(13) = 357097.

For example, f(n) = $n^5 - 4n^2$ works for n=3, but doesn't properly compute the other values in the table.

Now review the **ImprovedUnknownArraySolution** program. This solution first identifies the smallest and largest values in the *nxn* array (shown in the table below) and uses this information to reduce the number of array inspections.

n	# Array Inspections	min	max
3	126	3	19
4	632	4	49
5	2150	5	101
6	5742	6	181
…	…	…	…
13	326846	13	2029

Task 3.2 (10 points): Construct a function g(n) that accurately models the number of array inspections required by **ImprovedUnknownArraySolution** for an *nxn* square array. Confirm you have the proper g(n) by validating that it computes g(13) = 326846.

As a hint, observe that in the *nxn* array created by *UnknownArraySearch.create(n)*, its smallest value is *n* and its largest value is $n^2 * (n-1) + 1$.

Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

Q4. Stack Programming Exercise (20 pts.)

In the **algs.hw1.q4** package there is a **Computation** class that you must copy into your `USERID.hw1` package and modify to work properly. There are two parts to this question:

4.1 (10 points) Complete the implementation of `factorize(n)` which returns a stack of integer values, `Stack<Long>,` with the resulting factors on the stack in reverse order (assume that *n* > 1).

Invocation	Stack contents on return
factorize (2913732)	[23, 23, 17, 3, 3, 3, 3, 2, 2]
factorize (2913731)	[2039, 1429]
factorize (2913727)	[2913727]

When reading the stack representation above, recall that the left-most element is the top of the stack while the right-most element is at the bottom of the stack.

4.2(10 points) Complete the implementation of `isSquare(Stack<Long> factors)` which takes a stack of factors (as produced by `factorize`) and returns **`true`** if the factors (when multiplied together) represents a perfect square.

Invocation	Is perfect square	Actual number was…
isSquare ([3,3,2,2,2,2])	true	144
isSquare ([5,3,2,2,2])	false	120
isSquare ([199,199])	true	39601
isSquare ([5,5,3,3,2,2])	true	900
isSquare ([1707,1707])	True	2913849

Once your program is done, be sure to validate that it works on the values in the above table.

Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

Bonus Questions

In developing this homework, I encountered some nice problems that I decided to make optional and only worth 1 extra point. Please do not attempt these questions until you have completed the rest of the homework. Also, understand that it may take hours to complete these questions…

BQ5. FractalArraySolution (+1 bonus point)

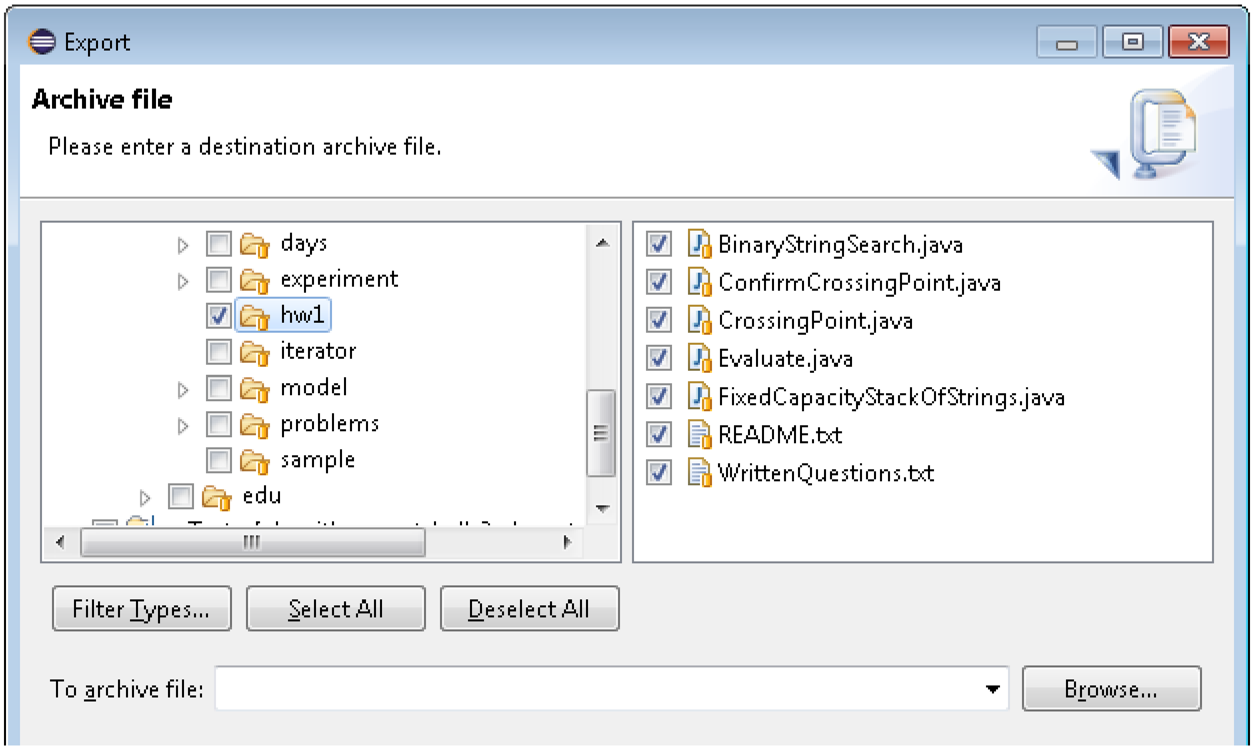
If you want an even more challenging bonus problem, copy **`algs.hw1.arraysearch.bonus.FractalArraySolution`** into your project area and modify it to work. Can you achieve (or do better than) **`6,739`** array inspections on a sample 14x14 array. Note that the arrays **FractalArraySolution** must be of a specific size (n = 2, 6, 14, 30, 62, …)

Published Version: 8:46 AM March 12 2019 (revised 11:30 AM 3/14/19)

Submission Details

Each student is to submit a single ZIP file that will contain the implementations. In addition, there is a file "WrittenQuestions.txt" in which you are to complete the short answer problems on the homework.

The best way to prepare your ZIP file is to export your entire **`USERID.hw1`** package to a ZIP file using Eclipse. Select your package and then choose menu item "**Export…**" which will bring up the Export wizard. Expand the **General** folder and select **Archive File** then click **Next**.



You will see something like the above. Make sure that the entire "hw1" package is selected and all of the files within it will also be selected. Then click on **Browse…** to place the exported file on disk and call it USERID-HW1.zip or something like that. Then you will submit this single zip file in my.wpi.edu as your homework1 submission.

Addendum

If you discover anything materially wrong with these questions, be sure to contact the professor or TA/SAs posting to the discussion forum for HW1 on piazza.

When I make changes to the questions, I enter my changes in red colored text as shown here.