Published Version: March 28 2020

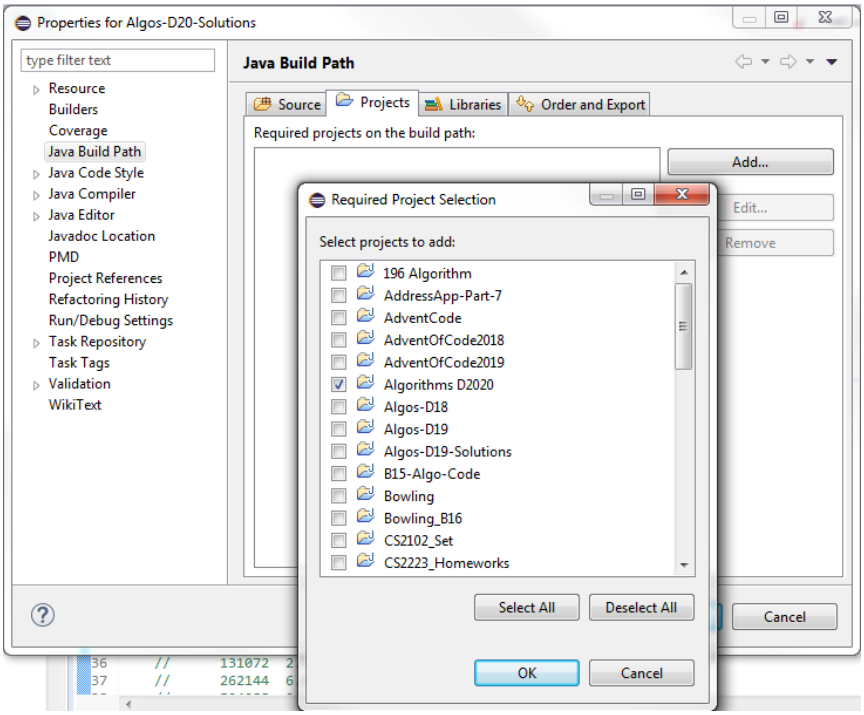
CS 2223 D20 Term. Homework 1 (100 pts.)

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/d20/#policies
- Due Date for this assignment is 2PM ~~Friday March 27st~~ Monday April 6th. Submissions received after 2PM are penalized 25%. Submissions 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 package USERID where USERID is your CCC user id. **You will lose TEN POINTS (or 10% of your assignment) if you don't do this. Pay Attention!!!**

First Steps

Your first task is to copy all of the files from the Git repository that you will be modifying/using for homework 1. 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 **Algorithms D2020** project to your build path.



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.RowOrderedArraySolution` → `USERID.hw1.RowOrderedArraySolution`
- `algs.hw1.arraysolution.DiagonalArraySolution`→ `USERID.hw1.DiagonalArraySolution`
- `algs.hw1.arraysolution.NestedArraySolution`→ `USERID.hw1.NestedArraySolution`
- `algs.hw1.Evaluate` → `USERID.hw1.Evaluate`
- `algs.hw1.arraysolution.Q3` → `USERID.hw1.Q3`
- `algs.hw1.Q4` → `USERID.hw1.Q4`
- `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.

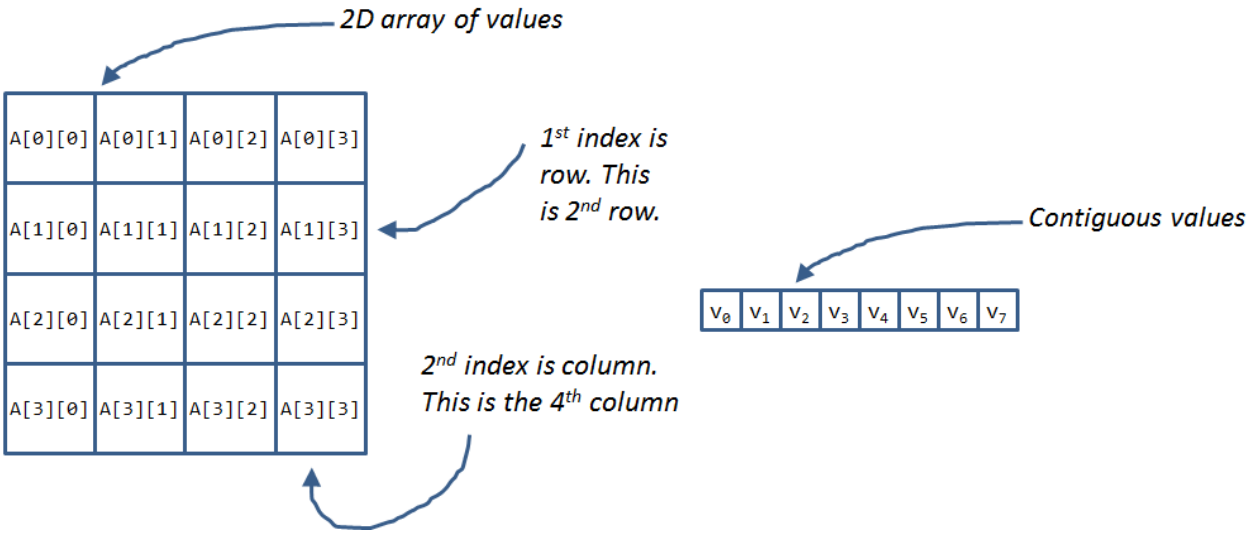
This homework has a total of 106 points. If you do all bonus work (do not attempt until completing the full homework!!) you can earn an additional six points. Note that the amount of work to complete the bonus points is not proportional to the few paltry points that you will achieve.

Note that Question **2.3 NestedArraySolution** is likely the hardest question on this assignment. On each homework, I typically have one such question, which guarantees that only the most dedicated students will get 100 points on each assignment.

Naturally, getting the full 106 points is exceptional.

Homework Context

This homework is concerned with how to efficiently access data stored either in a one-dimensional array (**Q4**) or a two-dimensional array (**Q2**). **Q1** uses a `FixedCapacityStack` (introduced on lecture 3).



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 `Evaluate` on input "`( 9 8 * / 3 )`"
1.2. **(4 pts.)** Run `Evaluate` on input "`( 9 - - 2 )`" (there is a space between the minus signs)
1.3. **(4 pts.)** Run `Evaluate` on input "`- 99`" (there is a space between the minus sign and the 9)
1.4. **(4 pts.)** Run `Evaluate` on input "`( 6 * ( 3 + ( 8 + 4`"
1.5. **(4 pts.)** Run `Evaluate` on input "`( ( 3 * 7 ) / ( ( 8 * 2 ) + ( 3 / 8 ) ) )`"
1.6. **(5 pts.)** Modify `Evaluate` to support two new operations
   a. Add a new equals operation "**=**" that computes whether two double values are equal. If the values are equal then 1.0 is the result, otherwise 0.0
   b. Add a new round operation "**round**" which computes the closest integer to x. This operator is a unary operator (like **sqrt** for square root). The `Math.round(double)` method will be useful for you.
1.7. **(5 pts.)** Run your modified `Evaluate` on input "`( round ( 8 / 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.

Q1.8 Bonus (+1 bonus point)

Can you modify `Evaluate` to be able to detect when the user input is not a valid infix expression? You should throw a `RuntimeException` (with helpful error message) for each of the input cases 1.1 through 1.5. **Hint: you may need to keep track of extra state while processing the operations and values.**

Q2. ArraySearch Programming Exercise (30 pts.)

You are given an *nxn* two dimensional (2D) square array of unique, positive integer values. The array represents a lower-triangle array, that is, all values above the main diagonal are zero; all other A[r][c] values in the array are greater than 0 and smaller than or equal to $\frac{(n-1)*n*(n+1)}{2}$; you can inspect the value of any cell in the array by calling `inspect(r, c)` where *r* is the desired row ($0 \le r < n$) and *c* is the desired column ($0 \le c < n$). **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 lower-triangle array or it will return **`null`** if `target` 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 <= r; c++) {    // only need to inspect lower triangle
      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 $\frac{n*(n+1)}{2}$ values. **algs.hw1.arraysolution.UnknownArraySolution** creates a 13x13 lower-triangular array and looks for all numbers from 1 to $\frac{(n-1)*n*(n+1)}{2}$; in this case, *n*=13, so it will call *locate* on the 1,092 values from 1 to 1,092. As you can see from the program output, it requires a total of 95,277 inspections of the array to locate each of these values (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, `trial` searches for 12 values; create a 5x5 array and it searches for 60 values. Below are the two arrays that are created:

	0	1	2
0	2		
1	4	6	
2	8	10	12

	0	1	2	3	4
0	4				
1	8	12			
2	16	20	24		
4	28	32	36	40	
5	44	48	52	56	60

As you can see, the smallest number (upper left corner) is *n-1*, the largest value (lower right corner) is $(n-1)*n*(n+1)/2$ and the difference between subsequent numbers (from left to right, and from top row to bottom row) is $n-1$. **Your task is to write more efficient `locate` methods if you know the specific structure *of the nxn* array**. 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.***

Q2.1 RowOrderedArraySolution

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

1. The array is a lower-triangular array where all values above the main diagonal are zero.
2. Each row contains ascending values from left to right (ignoring the zero values of course).
3. Each of the values in row $0 \leq k < (n\text{-}1)$ are smaller than the values in row $(k+1)$.

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

1	0	0	0	0
13	23	0	0	0
35	36	37	0	0
48	52	54	55	0
63	72	77	78	79

Copy **`algs.hw1.arraysolution.RowOrderedArraySolution`** into your `USERID.hw1` package and 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 10,000 array inspections.

Hint: **BINARY ARRAY SEARCH** ***can come to your rescue, thinking vertically***

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) **`7,600`** array inspections on the sample 13x13 array.

Q2.2 DiagonalSolution

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

1. The smallest value is in the top left cell A[0][0].
2. The largest value is in the lower left cell A[*n*–1][0].
3. There are *n* diagonal bands, each running from a cell in the first column "southeast" to the bottom row.
4. Diagonal band D_0 contains n values in ascending order along the main diagonal.
5. Diagonal band D_1 contains $n-1$ values in ascending order along the diagonal from A[1][0] to A[n-1][n-2]. Each successive band has one fewer element. D_{n-1} has just a single element.

Here is a sample 5x5 **DiagonalArray**. D_0 has the values 4 to 20 while D_1 has the values 24 to 36.

4	0	0	0	0
24	8	0	0	0
40	28	12	0	0
52	44	32	16	0
60	56	48	36	20

Copy **`algs.hw1.arraysolution.DiagonalArraySolution`** into your `USERID.hw1` package and write a more efficient `locate` method that takes advantage of the structure of a **DiagonalArray**.

Task 2.2 (10 points): Complete `locate` method in **DiagonalArraySolution** 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; thinking vertically?

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) `7,569` array inspections on the sample 13x13 array. ***Hint: Can BINARY ARRAY SEARCH come to your rescue again, this time thinking diagonally?***

Q2.3 NestedArraySolution

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

1. The smallest positive integer is in the upper left cell A[0][0].
2. The lower triangular array is decomposed into nested triangles. All other values in A are 0.
3. The outermost triangle, T_0, is composed of $3 * (n - 1)$ integers in increasing order in clockwise fashion. That is, running diagonally "south-east" diagonally, then "west" horizontally, then "north" vertically.
4. There are $k = 1 + \lfloor \frac{n-1}{3} \rfloor$ nested triangles. All values in triangle T_i are smaller than all values in inner Triangle T_{i+1}.
5. The next inner triangle, T_1, has $3 * (n - 4)$ integers. The next inner triangle, T_2, has $3 * (n - 7)$ integers, and so on (note: sometimes the final T_s computes zero; when that happens, set to 1).

Here are sample **NestedArrays**, with each nested triangle colored a different color.

1	0	0	0
9	2	0	0
8	10	3	0
7	6	5	4

1	0	0	0	0
12	2	0	0	0
11	13	3	0	0
10	15	14	4	0
9	8	7	6	5

1	0	0	0	0	0
15	2	0	0	0	0
14	16	3	0	0	0
13	21	17	4	0	0
12	20	19	18	5	0
11	10	9	8	7	6

1	0	0	0	0	0	0
18	2	0	0	0	0	0
17	19	3	0	0	0	0
16	27	20	4	0	0	0
15	26	28	21	5	0	0
14	25	24	23	22	6	0
13	12	11	10	9	8	7

Copy **`algs.hw1.arraysolution.NestedArraySolution`** into your `USERID.hw1` package and write a more efficient `locate` method that takes advantage of the structure of a **NestedArray**. Hint a naïve solution that (a) locates the proper triangle and then (b) checks all values in that triangle will be sufficient for Task 2.3.

Task 2.3 (10 points): Complete `locate` method in **NestedArraySolution** 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 a bit 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) `8,188` array inspections on the sample 13x13 array.

Q2.3.2 Bonus (+1 bonus point)

Given an *nxn* **NestedArray**, which cell, A[r][c], has the largest value? State your answer in terms of *n*, that is, state *r* and *c* in terms of *n*.

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 $\frac{(n-1)*n*(n+1)}{2}$.

If you execute **`algs.hw1.arraysolution.Q3`**, it will constrct *nxn* square arrays, of different sizes, and you will see the following output values:

n	# Array Inspections
3	57
4	255
5	795
6	1995
…	…
13	95277

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) = 95277.

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

Now review the results from **ImprovedUnknownArraySolution** which are the second part of the **Q3** output. 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	57	2	12
4	245	3	30
5	765	4	60
6	1932	5	105
…	…	…	…
13	94367	12	1092

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 lower-triangular array. Confirm you have the proper g(n) by validating that it computes g(13) = 94367.

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

Q4. Stack Programming Exercise (20 pts.)

There is a deep relationship between stacks and functional recursion. Copy **`algs.hw1.Q4`** into your **`USERID.hw1`** package and finish the partially-completed `fibonacci(FixedCapacityStack<Long> stack)` and `gcd(FixedCapacityStack<Long> stack)` functions.

4.1 (10 points) Consider the well-known Fibonacci series: 1, 1, 2, 3, 5, 8, 13, 21, 34 and so on. Each new number in the series is formed by adding the two prior numbers. The series is initialized starting with F(0)=1 and F(1)=1. This can be defined, recursively, as F(n) = F(n-1) + F(n-2). To compute the Nth Fibonacci number using just a standard `FixedCapacityStack`, push n onto a stack, `fcs`, and call `fibonacci(fcs)`. This function cannot add any additional variables to the method and you cannot call any other functions other than `push` and `pop`.

Think of it this way: Use the stack to represent Fibonacci numbers to be computed and keep a running sum when you encounter F(0) or F(1). Try a hand-computation for using stack to compute F(4)=5.

4.2 (10 points) Around 300 B.C. (more than 2,300 years ago) the Greek mathematician Euclid designed one of the first algorithms ever invented, called `gcd(a,b)` which computes the greatest common divisor of two integers `a` and `b`. In reviewing the definition of gcd on Wikipedia, you can see that it, too, can be defined recursively:

```
function gcd(a, b)
  if b = 0                  // BASE CASE
    return a
  else                      // RECURSIVE CASE
    return gcd(b, a mod b)
```

Where **mod** is the modulo operator, "%" in Java, that returns the remainder when dividing `a` by `b`.

For this assignment, you must complete the partially-completed `gcd(FixedCapacityStack<Long> stack)` function to compute the greatest common divisor of the two long values that have been pushed to the stack.

Given the existing code, you cannot add any additional variables to the method and you cannot call any other functions other than `push` and `pop`.

Think of it this way: Use the stack to store two numbers for which the gcd is to be computed. When two numbers are popped from the stack, either the BASE CASE occurs (in which case the value is computed) or the RECURSIVE CASE occurs, and two numbers are pushed back onto the stack to repeat.

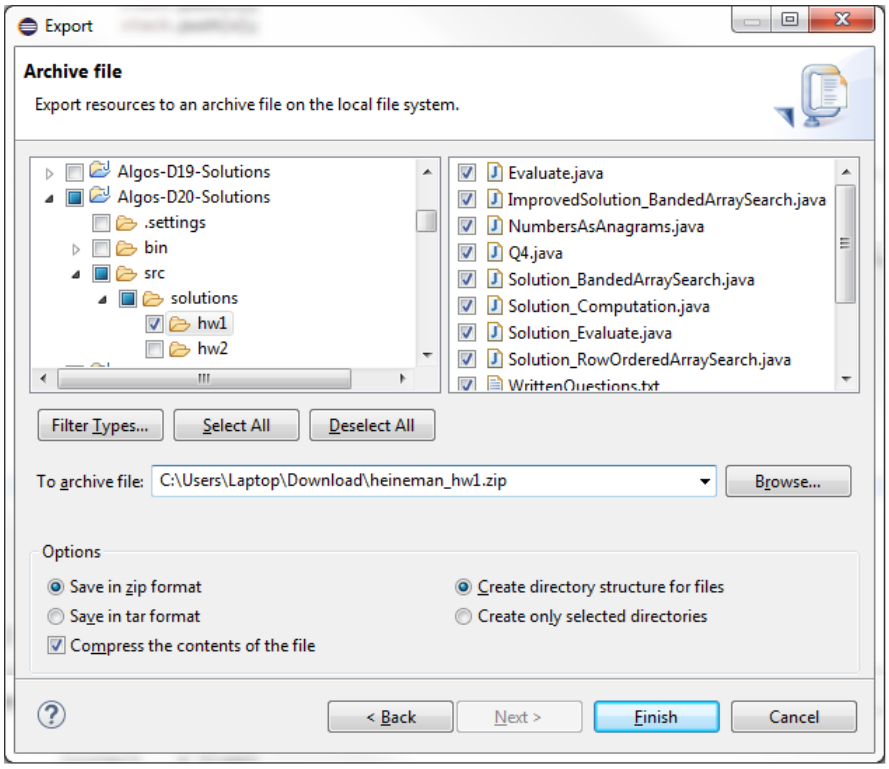
Q4.3 Bonus (+1 bonus point)

After you have completed the implementation of `fibonacci`, the output of the third column contains the minimum time to compute F_n. Compute the ratio of minTime(n+1)/minTime(n) which determines how much longer it takes to compute F_{n+1} when compared to the time it takes to compute F_n. You should be able to see that this ratio converges. What is the value to which this ratio converges?

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 canvas.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.

1. HW still had original due date (2PM ~~Friday~~ ~~March 27st~~ Monday April 6th) now fixed.
2. Also include request to copy Q3 into your USERID.hw1.Q3