Published Version: March 16 2022 8:30 PM

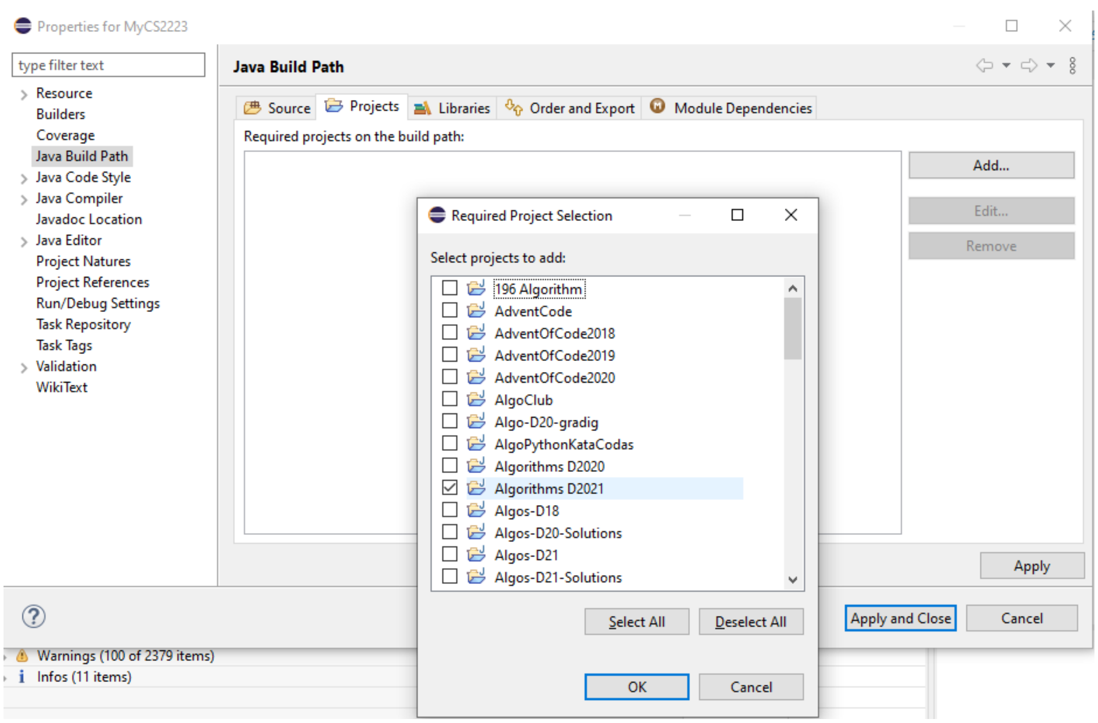
CS 2223 D22 Term. Homework 1 (100 pts.)

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/d22/#policies
- Due Date for this assignment can be found in canvas. Late Submissions received after the deadline are penalized 25% and can be submitted for up to 48 hours.
- 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 WPI user name (the letters before the @wpi.edu in your email address). **You will lose FIVE POINTS (or 5% of your assignment) if you don't do this. Pay Attention!!!**

First Steps

Your first task is to copy all of the necessary 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 D2022** 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 file into your **USERID.hw1** package.

- `hw1.WrittenQuestions.txt` → `USERID.hw1.WrittenQuestions.txt`
- Other files will be copied over, as described in each question

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

- Q1 is worth 30 points
- Q2 is worth 30 points (+4 bonus points)
- Q3 is worth 30 points (+3 bonus points)
- Q4 is worth 10 points (+2 bonus points)

This homework has a total of **109** points. You can earn additional bonus points, but sometimes the extra bonus questions require some extensive work so be sure to complete regular homework first.

Published Version: March 16 2022 8:30 PM

Q1. Stack Experiments (30 pts.)

On page 129 of the book there is an implementation of a calculator algorithm using two stacks to evaluate an expression, invented by Dijkstra (one of the most famous designers of algorithms). I have created the **algs.hw1.`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 ")".

The following inputs are all improperly formatted, but I am curious what will be output:

1.1. **(4 pts.)** Run `Evaluate` on input "`( 3 2 * / 5 )`"
1.2. **(4 pts.)** Run `Evaluate` on input "`( 4 + + 1 )`" (there is a space between the plus signs)
1.3. **(4 pts.)** Run `Evaluate` on input "`- 76`" (there is a space between the minus sign and the 76)
1.4. **(4 pts.)** Run `Evaluate` on input "`( 8 * ( 9 + ( 3 + 4`"

The following input is more complicated but has the right format.

1.5. **(4 pts.)** Run `Evaluate` on input "`( ( 3 + 1 ) / ( ( 4 * 1 ) / ( 5 - 9 ) ) )`"

Now modify Evaluate to support new operations:

1.6. **(5 pts.)** Modify `Evaluate` to support two new operations:
   a. Add a new binary operation "n **exp** b" that computes n^b.
   b. Add a new binary operation "n **log** b" which computes $\log_b(n)$.

1.7. **(5 pts.)** Run your modified `Evaluate` on input "`( 2 exp ( 17 log 4 ) )`" and be sure to explain the result of the computation in your `WrittenQuestions.txt` file. Hint: be sure to explain what happens when processing each of the ')' closing parentheses.

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.

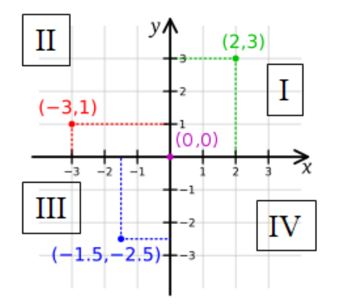
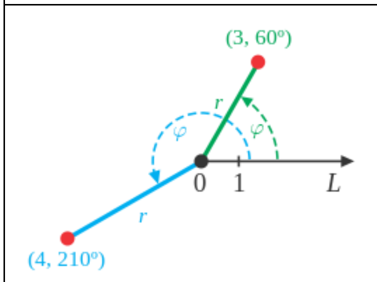
Note: If, to an empty stack, you push the value "1", "2" and then "3", the state of this stack is represented as ["1", "2", "3"] where the top of the stack contains the value "3" on the right, and the bottommost element of the stack, the value "1", is on the left. An empty stack is represented as [].

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.

Q2. Searching Programming Exercise (30 pts.)

Many algorithms are concerned with searching for values within a given data structure, and this homework assignment is no exception.

For this assignment you will be working with collections of points in a two-dimensional plane. From high school mathematics, you should be familiar with the following two types of points:

	Cartesian points are defined using (x, y) coordinates. The (0,0) origin is typically shown in the middle of the page. There are four quadrants: Quadrant **I** in the upper right corner, Quadrant **II** is in the upper left. Quadrant **III** is in the lower left, and Quadrant **IV** in the lower right. The x-axis increases in value to the right and the y-axis increases in value up the page.
	Polar points are defined using (r, theta) where r is a magnitude distance away from the central origin point (0) on the polar axis L. Theta is the angular coordinate in counter-clockwise motion away from L. Theta is described using degrees from 0 (which aligns with the L axis) all the way up to, but not including 360. Note that r can never be negative and theta must be in the range [0, 360). Quadrant **I** contains points with 0 < theta < 90, Quadrant **II** has points with 90 < theta < 180. Quadrant **III** has points with 180 < theta < 270. Quadrant **IV** has points with 270 < theta < 360.

For this assignment, all Cartesian and Polar points will be defined using integer values for x, y, r and theta.

Given a collection of unique points stored in an array there are some operations of interest:

- **Exists** – check whether a point exists in the array
- **CountInQuadrant(q)** – count the number of points in quadrant 1, 2, 3 or 4
- **CountBetween(min, max)** – draw two rays from the origin to infinity at angles `min` and `max`, each counter-clockwise from the horizontal axis, **L**. Count the number of points between the two of them. *While this could also be applied to Cartesian points, for simplicity, this will only be applied to Polar Points.*

These operations can be made more efficient if you order the points in specific ways. For this question, you are to construct solutions for both Cartesian and Polar Points. You will make use of an `OrderedArray<T>` class that stores an array of elements that have been sorted in a specific way.

There are multiple ways to sort an array of points. For this assignment, two will be relevant:

- *comparePolarByTheta* – sorts Polar points counter-clockwise around the origin, O, breaking ties by sorting using distance, r, from the origin. Using this sorting scheme, the following nine points would be sorted from 1 to 9:

```
             ^
      4      |      2
             | 3
       5     |     1
<------------O------------->
  6          |    9
       7     |
             |     8
             V
```

- *compareCartesianByQuadrant* – sorts Cartesian points by quadrant: RIGHT to LEFT from Quadrant **I** to **II**, and then LEFT to RIGHT from Quadrant **III** to **IV**. The same points above would be ordered as follows:

```
             ^
      5      |      1
             | 3
       4     |     2
<------------O------------->
  6          |    8
       7     |
             |     9
             V
```

The points will be randomly computed. Cartesian points will never appear on either the X- or Y-axis while Polar points will never have a theta of 0, 90, 180 or 270.

The code I provide for this question generates random Cartesian points in the range from x=[-500, 499] and y=[-500, 499] using only integer coordinates. No point will be duplicated, and since no points will be generated on the axes, the total number of potential points that could be generated is 998,001.

When generating random Polar points, r is drawn from the range [0, 499] and theta is [0, 359]. Since no points will be generated with r=0 or theta of 0, 90, 180, 270, the total number of potential points that could be generated is 89*4*499 = 177,644.

Published Version: March 16 2022 8:30 PM

Q2.1 Cartesian Trials

For this question, you are asked to write an efficient `countCartesiansInQuadrant(points, q)` method that counts the number of points in the given quadrant (1, 2, 3, or 4). Assume that `cartesians` has already been sorted by the *`compareCartesianByQuadrant`* comparator.

Copy **`algs.hw1.QuadrantCounting`** into your **`USERID.hw1`** package and complete the implementation of `int countCartesiansInQuadrant(OrderedArray<Point> cartesians, int q)`, which returns the number of points in the given quadrant.

The `cartesians` object cannot be directly accessed like an array, but you can call the following methods:

- **`length()`** returns the size of the collection.
- **`get(idx)`** returns the `Point` stored at the given index location, **`idx`**.

Execute your class to generate a table that outperforms the following result (which you can observe by executing **`algs.hw1.fixed.CartesianTrials`**).

N	Q1	Q2	Q3	Q4	QAll	#Inspections
255	68	56	65	66	255	1020
511	135	137	124	115	511	2044
1023	224	269	266	264	1023	4092
2047	518	534	500	495	2047	8188
4095	1040	1021	1033	1001	4095	16380
8191	2098	2053	1984	2056	8191	32764
16383	4108	4124	4096	4055	16383	65532
32767	7988	8229	8217	8333	32767	131068
65535	16285	16436	16364	16450	65535	262140

Each row above reports the results of counting the points in the four quadrants – showing how many were found in each one – and summarizing under **`QAll`** the total number (which must equal **N**). The final column, labeled **#Inspections**, determines the total number of **`get(idx)`** calls that were used. As you can see, this number is always 4xN, because the naïve brute-force algorithm simply checks each point to determine whether it is in quadrant q, and `countCartesiansInQuadrant(q)` is called four times.

Task 2.1 (10 points): Complete `countCartesiansInQuadrant()` method in **`QuadrantCounting`** and produce the following output (which is correct, yet requires FAR FEWER inspections):

N	Q1	Q2	Q3	Q4	QAll	#Inspections
255	68	56	65	66	255	48
511	135	137	124	115	511	54
1023	224	269	266	264	1023	60
2047	518	534	500	495	2047	66
4095	1040	1021	1033	1001	4095	72
8191	2098	2053	1984	2056	8191	78
16383	4108	4124	4096	4055	16383	84
32767	7988	8229	8217	8333	32767	90
65535	16285	16436	16364	16450	65535	96

Q2.1.1 Bonus (+1 bonus point)

Develop a formula C(N) – where N is one less than a power of 2 – that counts the number of Inspections in the table shown in Task 2.1 above, for any N ≥ 255.

Q2.1.2 Bonus (+1 bonus point)

Only attempt this bonus point after you have completed the first part. Can you come up with an optimization that improves performance to be:

N	Q1	Q2	Q3	Q4	QAll	#Inspections
255	68	56	65	66	255	48
511	135	137	124	115	511	53
1023	224	269	266	264	1023	59
2047	518	534	500	495	2047	64
4095	1040	1021	1033	1001	4095	71
8191	2098	2053	1984	2056	8191	77
16383	4108	4124	4096	4055	16383	83
32767	7988	8229	8217	8333	32767	88
65535	16285	16436	16364	16450	65535	95

Published Version: March 16 2022 8:30 PM

Q2.2 Polar Trials

For this question, you are asked to write efficient methods below. Assume that `polars` has already been sorted by the *comparePolarByTheta* comparator.

- `boolean existsThetaOrdered(OrderedArray<Point> polars, PolarPoint p)`, which determines whether `p` is in the `polars OrderedArray`.
- `int countBetweenThetaOrdered(OrderedArray<PolarPoint> points,int min,int max)` which counts the number of PolarPoints between angles `min` and `max` (inclusive on both ends).

Copy **`algs.hw1.PolarPointTrials`** into your **`USERID.hw1`** package and complete the implementation of two methods:

The `polars` object cannot be directly accessed like an array, but you can call the following methods:

- **`length()`** returns the size of the collection.
- **`get(idx)`** returns the Point stored at the given index location, **`idx`**.

Execute your class to generate a table that outperforms the following result (which you can observe by executing **`algs.hw1.fixed.PolarTrials`**).

```
N       #Found  Exists-I        #Betw.  Between-I
255     105     16698554        255     15300
511     170     33447889        511     30660
1023    374     66849068        1023    61380
2047    773     133330107       2047    122820
4095    1511    265239878       4095    245700
8191    2949    524900393       8191    491460
16383   6077    1023467126      16383   982980
32767   12131   1947821462      32767   1966020
65535   23839   3516530722      65535   3932100
```

Each row above reports the results of (a) 65,536 invocations of `existsThetaOrdered()` using random `PolarPoints`; and (b) invoking `countBetweenThetaOrdered()` 60 times using different 6 degree slices, for example, (min=0, max=5), then (min=6, max=11), then (min=12, max=17) all the way up to (min=354, max=359). The value in **#Betw.** must exactly match N since all possible angles are involved.

The values of **`Exists-I`** and **`Between-I`** reflect the total number of inspections required to perform each trial. Your challenge is to solve the question more efficiently.

Note: Make sure that your local copy of `PolarPointTrials` has its main method to match this:

```
/** Do not change this function. Just execute it. */
public static void main(String[] args) {
      new PolarPointTrials().runTrial();
}
```

Task 2.2 (20 points): Complete `existsThetaOrdered()` and `countBetweenThetaOrdered()` methods in **`PolarPointTrials`** to produce the following improved results (note: the values under **#Found** and **#Betw.** must exactly match the earlier table).

N	#Found	Exists-I	#Betw.	Between-I
255	105	524184	255	960
511	170	589655	511	1080
1023	374	655014	1023	1200
2047	773	720117	2047	1320
4095	1511	784783	4095	1440
8191	2949	849091	8191	1560
16383	6077	911388	16383	1680
32767	12131	970783	32767	1800
65535	23839	1024892	65535	1920

Q2.2.1 Bonus (+1 bonus point)

Only attempt this bonus point after you have completed the first part. Can you achieve (or do better than) the following results for **`Between-I`**, which reduces the number of inspections needed to:

N	#Found	Exists-I	#Betw.	Between-I
255	105	524184	255	885
511	170	589655	511	997
1023	374	655014	1023	1124
2047	773	720117	2047	1243
4095	1511	784783	4095	1363
8191	2949	849091	8191	1477
16383	6077	911388	16383	1597
32767	12131	970783	32767	1718
65535	23839	1024892	65535	1841

Q2.2.2 Bonus (+1 bonus point)

Given the tabular output above, do the **#Found** numbers seem reasonable? That is, since random Polar points are being generated and random Polar points are being searched for, compute the expected totals for this **#Found** column (using basic statistics) and compare against the output above.

Q3. Stack Programming And Recursion Exercise (30 pts.)

Q3.1 Evaluate and Convert a Postscript Expression into an Infix Expression [10pt]

Question 1 contains the `Evaluate` class that demonstrates how to use stacks to compute the value of an infix expression, where a binary operator appears between its arguments, like "((4 + 5) * 7)", using parentheses to disambiguate sub-expressions.

Expressions can be represented without parentheses using **postfix notation**[1], where a binary operator appears after its arguments. The infix expression "((4 + 5) * 7)" is represented as "4 5 + 7 *" using postfix. You can read this from left to right as "Given values 4 and 5, sum them and leave the result as 9, which together with 7 is multiplied to make 63". The elegance of postfix notation is that you do not need parentheses! Forget PEMDAS! This is the future of computation.

Copy the **`algs.hw1.PostFixToInfix`** class into your **`USERID.hw1`** package and complete the implementation so it converts postfix expressions into an infix expression, using parentheses for each sub-expression. While doing this conversion, you should also compute the value's expression, using similar logic to what you saw in Evaluate – you only need to support the standard mathematical operations of +, -, /, and *.

It should process a single a postfix notation input (with spaces between all values and operators) using **`FixedCapacityStack`**. Output the corresponding infix expression for the input.

Sample Input	Sample Output
`2 6 +`	`(2 + 6) = 8.0`
`3 1 + 4 * 1 5 - /`	`(((3 + 1) * 4) / (1 - 5)) = -4.0`
`9 8 7 6 5 * / - +`	`(9 + (8 - (7 / (6 * 5)))) = 16.766666…`

[1] https://en.wikipedia.org/wiki/Reverse_Polish_notation

Q3.2 There is a deep relationship between Recursion and Stacks [10pt]

The Fibonacci Sequence has been studied extensively throughout history.

Fibonacci Sequence (A00045)	The Zeckendorf sum for n is computed as follows:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$	`while n > 0:`
	`  f = largest Fibonacci number smaller than n`
	`  print (f)`
	`  n = n - f`

Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, …. Note that F_8 = 21 since the first entry is F_0.	

According to the Zeckendorf theorem, every positive integer can be represented uniquely as the sum of *one or more* distinct Fibonacci numbers in such a way that the sum does not include any two consecutive Fibonacci numbers.

For example, the Zeckendorf sum for 64 is 55 + 8 + 1. This can be converted into a Zeckendorf representation by using 0 or 1 to determine which Fibonacci numbers are included in the sum. For example:

64 = 1 (55) + 0 (34) + 0 (21) + 0 (13) + 1 (8) + 0 (5) + 0 (3) + 0 (2) + 1 (1)

Which means the Zeckendorf representation is 100010001. For any given positive integer, the Zeckendorf sum can be found using a greedy algorithm, as shown above. Your job is to compute the Zeckendorf string representation for any positive integer.

Copy the **`algs.hw1.Zeckendorf`** class into your **`USERID.hw1`** package and complete the implementation so it generates the following table:

```
N       Zeck. Repr.
1       1
2       10
3       100
4       101
5       1000
6       1001
7       1010
8       10000
9       10001
10      10010
11      10100
12      10101
13      100000
14      100001
15      100010
```

Task 3.2 (10 points): Complete `implementation and` generate above table **[10 pts]**

Q3.2.1 Bonus Question (1 pt)

Can you determine the number of Zeckendorf representations of length N? For example, there are three encodings of length four (i.e., `1000, 1001, 1010`).

Q3.2.2 Bonus Question (1 pt)

What is the Fibonacci encoding for `9223372036854775807`?

Published Version: March 16 2022 8:30 PM

Q3.3 Double Stack Implementation [10pt]

I will describe the implementation of `FixedCapacityStack<T>` on **`day03`** of the class. For this question, you are to implement a `DoubleStack` that uses a single array to store two independent stacks of **`int`** values, one that grows from the left upwards into the array, while the other grows from the right downwards into the array.

Copy the **`algs.hw1.DoubleStack`** class into your **`USERID.hw1`** package and complete the implementation. For example, after creating a `DoubleStack` of size 7, its storage array looks like the following:

--	--	--	--	--	--	--

Now issue the following commands:

- `pushLeft(5)`
- `pushLeft(3)`
- `pushRight(7)`
- `pushRight(2)`
- `pushLeft(1)`

The resulting storage should look like the following:

5	3	1	--	--	2	7

Naturally, your `DoubleStack` must support the following methods:

- `pushLeft(v)` and `pushRight(v)` – Pushes value to top of either left or right side
- `isFull()` – is there room to push an element (either on the left or right side?)
- `sizeLeft()` and `sizeRight()` to determine the number of elements on either side
- `popLeft()` and `popRight()` to remove an element from either side
- `exchange()` – if left side and right side contain at least one element each, swap top values

You can confirm your implementation is complete by copying **`algs.hw1.TestDoubleStack`** from the **`test/`** source folder in **Algorithms D2022** into your **`USERID.hw1`** directory. Then execute it as a JUnit test case and all test cases should pass.

Q3.3.1 Bonus Question (1 pt)

Create a `DoubleStackIterator` that drains all values from the Double stack by popping and returning all values from the left side; then once the size of the left is 0, pop and return all values from the right side. When the Iterator completes, the `DoubleStack` is empty.

Q4 Big O Notation [10 points]

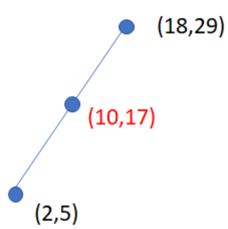
In lecture, I will present the Big O notation used to classify the worst case runtime performance of an algorithm. The ThreeSum program on page 173 offers a classic example of an $O(N^3)$ algorithm. Upon inspecting the code, you can see the triply-nested **for** loop that ensures that each different possible triple (i, j, k) is checked. With a little bit of mathematical help, you can evaluate that the number of times the **if** statement executes is $\frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3}$. For n=10, for example, this results in (1000/6 – 100/2 + 10/3) = 120. You can read more about this on p. 181 of the textbook. In lecture I will explain asymptotics to explain behavior: as N grows larger and larger, the n^3 term dominates and determines the order of growth.

```
public static int count(int[] a) {
  int N = a.length;
  int ct = 0;
  for (int i = 0; i < N; i++) {
    for (int j = i+1; j < N; j++) {
      for (int k = j+1; k < N; k++) {
        if (a[i] + a[j] + a[k] == 0) { ct++; }
      }
    }
  }
  return ct;
}
```

You will try another problem and use Big O notation to classify its worst-case runtime performance as well as provide empirical evidence.

There is a well-known Sylvester Line Problem, which states that in every finite set of points in the Euclidean plane, there is a line that either (a) passes through exactly two of the points; or (b) it passes through all of them.

Copy the **`algs.hw1.LineProblem`** class into your **`USERID.hw1`** package and complete its implementation. I have provided a helper method to compute the greatest common divisor between two integers. You will find this useful when determining whether three points are collinear. As you have already seen, these random Cartesian points all have integer coordinates, which makes it easy to



determine whether three points are collinear. In the figure on the left, for example, if you select points p1=(2,5) and p2=(18,29), you can determine that the slope of this line is $\frac{(29-5)}{(18-2)}$ or $\frac{24}{16}$ which can be simplified to $\frac{3}{2}$.

Now, the middle point p3=(10,17) is on this line, because you can compute the slope between points p1 and p3 as $\frac{(5-17)}{(2-10)}$ or $\frac{-12}{-8}$ which can be simplified to $\frac{3}{2}$ so you know it is on the same line as the other two points.

You must complete the following implementations:

- `Solution compute(Point[] points)` – compute a `Solution` to the Sylvester Line Problem by returning a structure that contains two points and the number of other points that are collinear to these points (which handles the odd case where ALL points are collinear)
- `int findAllJustTwo(Point[] points)` – compute the number of potential lines that contain only two collinear points from `points`.

Task 4.1 (10 points): Complete `implementation and` generate following table **[10 pts]**

```
This will take over a minute to complete. Be patient!
N       Time    TimeCt  #Found
32      0.000   0.000   496
64      0.000   0.000   2009
128     0.000   0.016   8095
256     0.000   0.125   32335
512     0.000   0.953   128360
1024    0.000   8.109   506149
2048    0.000   67.844  1979530
```

The first column, **Time**, reports the time to find a single pair of points that have no other collinear point. The second column, **TimeCt**, reports the time it took to find the total number of pairs of points that share no other collinear points (also reported in **#Found**).

As you can see, **TimeCt**, appears to increase by a factor of 8 whenever the problem instance size, N, doubles. This strongly indicates a $O(N^3)$ implementation.

Q4.1.1 Bonus Question (1 pt)

Determine a formula F(N) that predicts **#Found** for a given problem size, N. It won't be exact (since these are randomly generated points) but it should do a good job in predicting this actual number.

Q4.1.2 Bonus Question (1 pt)

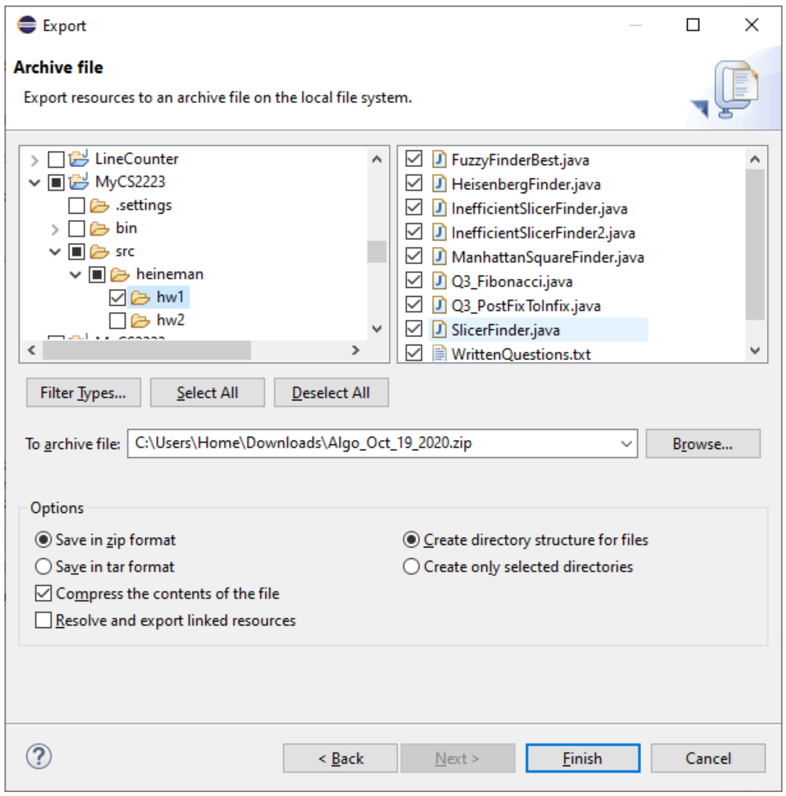
What is the Big O classification for `compute(Point[] points)`?

Published Version: March 16 2022 8:30 PM

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 discord.

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

1. Deadline is meant to be at the start of class, which is 10AM, but you can turn in up to 6PM.

2. If, for question 3.3, you copy algs.hw1.TestDoubleStack into your USERID.hw1 folder, then you will need to modify the build properties for the project to include the JUnit 5 runtime libraries, as I describe in the video I posted further describing this homework assignment.
3. For question 2.2, you have to make sure that YOUR local copy of "PolarPointTrials" has a main method that instantiates your local PolarPointTrials object, like this:

```
/** Do not change this function. Just execute it. */
public static void main(String[] args) {
  new PolarPointTrials().runTrial();
}
```