Published Version: April 02 2021 9:25 AM

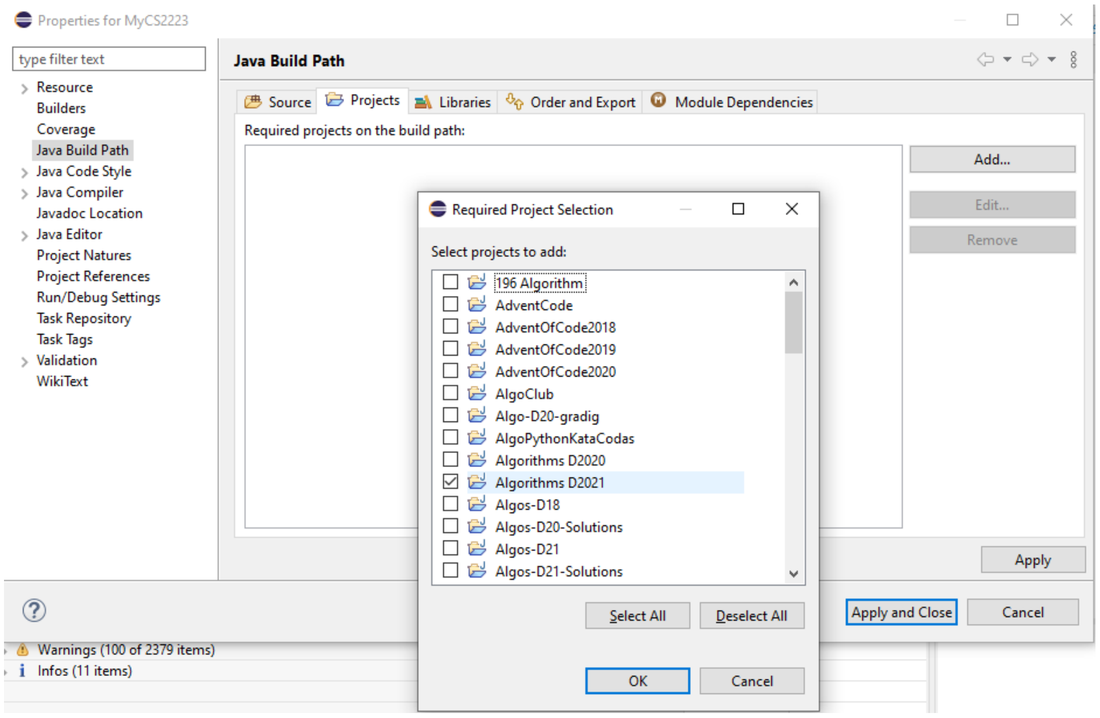
CS 2223 D21 Term. Homework 1 (100 pts.)

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/d21/#policies
- Due Date for this assignment is 10AM Monday April 5th. Submissions received after 10AM 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 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 D2021** 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.WrittenQuestions.txt → USERID.hw1.WrittenQuestions.txt`
- `algs.hw1.submission.* → Copy all classes in this package to USERID.hw1.*`

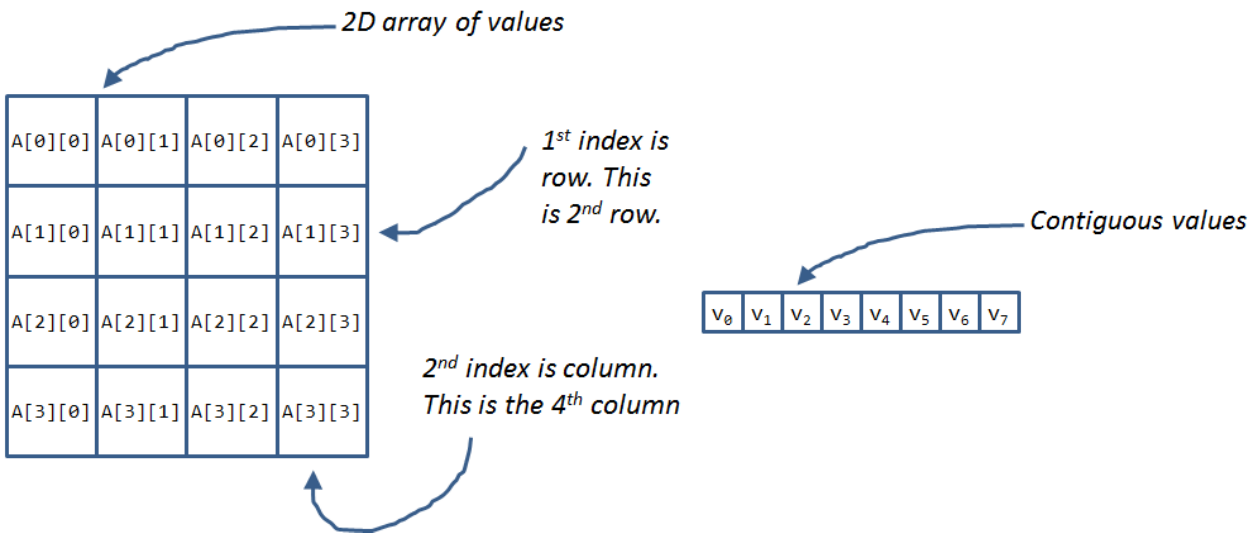
In this way, I can provide sample code for you to easily modify and submit for your assignment.

- Q1 is worth 30 points
- Q2 is worth 40 points
- Q3 is worth 20 points
- Q4 is worth 10 points

This homework has a total of **107** points. If you do all bonus work (do not attempt until completing the full homework!!) you can earn an additional 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 FuzzySquare Finder** 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 **107** points is exceptional.

This homework is concerned with how to efficiently access data stored in a 1XN one-dimensional array or an NXN two-dimensional square array. **Q1** uses a **`FixedCapacityStack`** (introduced in lecture 4).



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 `"( 5 8 * / 4 )"`
1.2. **(4 pts.)** Run `Evaluate` on input `"( 5 + + 8 )"` (there is a space between the plus signs)
1.3. **(4 pts.)** Run `Evaluate` on input `"- 71"` (there is a space between the minus sign and the 7)
1.4. **(4 pts.)** Run `Evaluate` on input `"( 1 * ( 2 + ( 3 + 4"`

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

1.5. **(4 pts.)** Run `Evaluate` on input `"( ( 6 * 4 ) / ( ( 3 * 7 ) / ( 5 - 1 ) ) )"`

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. ← FIXED
   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.

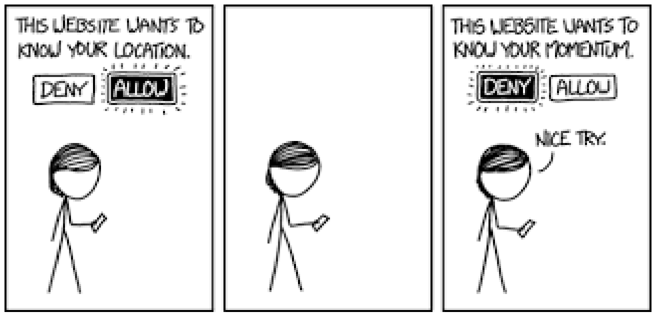
Note: If, to an empty stack, you push the value "1", "2" and then "3", the state of this stack is represented as ["1", "2", "3"] where the top of the stack contains the value "3" on the right, and the bottommost element of the stack, the value "1", is on the left. An empty stack is represented as [].

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 (40 pts.)

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

This particular homework assignment presents a philosophical challenge inspired by Heisenberg's uncertainty principle of quantum mechanics.


THIS WEBSITE WANTS TO KNOW YOUR LOCATION.
DENY
ALLOW
THIS WEBSITE WANTS TO KNOW YOUR MOMENTUM.
DENY
ALLOW
NICE TRY.

"What we observe is not nature itself, but nature exposed to our method of questioning."
Werner Heisenberg

You will be given Java objects that store a one-dimensional array (1XN) or a two-dimensional square NxN array. Your task is to determine where a particular target value is contained within the object's storage, but you only have a limited number of mechanisms to inspect these objects' data. Each inspection probes an object's state, and a running total of all probes is maintained by these objects.

For each of these problems, you are to develop an algorithm that properly locates target values (under specific restrictions imposed). You will receive credit based on correctly locating all targets **and** meeting specific constraints regarding the total number of probes your algorithm requires on specific problem instance sizes.

In a one-dimensional array, a value's location is uniquely determined by its index, which I commonly abbreviate as `idx`, which is a value between `0` and N-1 inclusively. In a two-dimensional array, a value is located at a given row and column location (using zero-indexing, thus the left-most column is column 0 and the bottom-most row is row N-1). I use an `algs.hw1.Coordinate(row, column)` object to represent the location of a value in a two-dimensional array.

Each Java object has a `toString()` method that will produce a string representation of its state, but doing so will randomized its state, so this method is only useful for debugging.

For each of the following problems, I provide code that generates a table recording the total number of accumulated probes for different trial runs of size N, when calling the location methods that you define. This table will be used by the grading staff to evaluate your assignment,.

Q2.1 Slicer

A **slicer** object creates a two-dimensional NxN array, A, containing the integers from 0 to N*N–1 in some random arrangement. The following is **`slicer = new Slicer(5, 99)`** using random seed 99:

	c_0	c_1	c_2	c_3	c_4
r_0	4	10	6	17	22
r_1	20	5	24	18	14
r_2	9	11	16	3	21
r_3	0	19	8	1	23
r_4	15	13	7	2	12

This array cannot be directly queried, but there are two supported inspection methods, each one counting as a probe of the underlying data:

- **`inLeft(c, target)`** determines whether the target value is found in any of the columns between column 0 and including column **`c`**. In the **slicer** object above, `inLeft(2, 20)` and `inLeft(2, 7)` are `true` but `inLeft(2, 14)` is `false`.
- **`inTop(r, target)`** determines whether the target value is found in any of the rows between row 0 and including row **`r`**. In the **slicer** object above, `inTop(1, 20)` and `inTop(0, 17)` are `true` but `inTop(1, 9)` is `false`.

Copy **`algs.hw1.submission.SlicerFinder`** into your **`USERID.hw1`** package and complete the implementation of `Coordinate find(Slicer s, int target)` which returns a `Coordinate` object for the location of `target` in `s` (or **`null`**, if the `target` does not exist). This `find()` method must use `inLeft(c, target)`and `inTop(r, target)` probes to determine where target exists. As an example, using the **slicer** object above, **`find(s, 15)`** must return a `Coordinate` object (row=4, column=0).

Execute your class to generate a table showing how many accumulated probes you needed to properly locate all NxN values in a **Slicer**(N) object. A total of N*N ~~probes~~ targets are ~~made~~ searched for in each trial.

Task 2.1 (10 points): Complete `find()` method in **`SlicerFinder`** and ensure that on a 13x13 array, the sample trial executed by **`SlicerFinder.main`**() completes with fewer than 3,000 probes.

Q2.1.1 Bonus (+1 bonus point)

Only attempt the bonus point after you have completed the first part. Can you achieve (or do better than) **`1,274`** total probes on the sample 13x13 array. This is the number reported by **`slicer.solver(new algs.hw1.submission.SlicerFinder())`**that you can see in the `main()` method of **`SlicerFinder`**.

Hint:* Binary Array Search *can come to your rescue

Q2.1.2 Bonus (+1 bonus point)

Develop a formula C(N) – where N is a power of 2 – that counts the number of total probes as reported in the following table, which represents the output from my best solution. The total number of probes is the accumulation of searching for all N^2 integers in the range from 0 to N*N – 1. For example, C(16) should compute 2048. Note that the formula only needs to be accurate when N is a power of 2.

N	Total Probes by Slicer
2	8
3	30
4	64
5	120
6	192
7	280
8	384
9	522
10	680
11	858
12	1056
13	1274
14	1512
15	1770
16	2048
17	2380
18	2736
19	3116

Q2.2 ManhattanSquare

A **`ManhattanSquare`** object creates a two-dimensional NxN array, `A`, containing the integers from 0 to N*N–1 in random locations. The following is **`ms = new ManhattanSquare(5, 99)`** using random seed 99:

	c_0	c_1	c_2	c_3	c_4
r_0	4	10	6	17	22
r_1	20	5	24	18	14
r_2	9	11	16	3	21
r_3	0	19	8	1	23
r_4	15	13	7	2	12

This array cannot be directly queried; instead, use **`int distance (r, c, target)`** to return the *Manhattan distance*[1] from location `A[r][c]` to the actual location of `target` within the array; if `target` does not exist in A then return `-1`, otherwise the distance is returned in terms of the number of horizontal and vertical steps from `A[r][c]` to target's location. You are to develop code that determines the `Coordinate` for a target value's location in `A` or return **`null`** to signal that value is not present.

- `distance(2, 1, 17)` returns `4` because you need at least this many up/down/left/right movements to get from `A[2][1]` to the location that contains `17`, in this case `A[0][3]`.
- `distance(3, 4, 23)` returns `0` because `A[3][4]` contains that value; `distance(0,0,99)` returns `-1` since that value does not exist in `A`.

Copy **`algs.hw1.submission.ManhattanSquareFinder`** into your `USERID.hw1` package and complete the implementation of `Coordinate find(ManhattanSquare ms, int target)` which returns a `Coordinate` object for the location of `target` in `ms` (or **`null`**, if the `target` does not exist). Using the **`ms`** object above, `find(ms, 2)` must return a `Coordinate` object (row=2, column=4).

Once you implement `find()`, execute your class to generate a table showing how many accumulated probes you needed to properly locate all NxN values in a **`ManhattanSquare`**(N) object. A total of N*N ~~probes~~ targets are ~~made~~ searched for in each trial.

Task 2.2 (10 points): Complete `find()` method in **`ManhattanSquareFinder`** and ensure that on a 13x13 array, the sample trial completes with fewer than 1,000 probes.

Q2.2.1 Bonus (+1 bonus point)

Only attempt the bonus point after you have completed the first part. Can you achieve (or do better than) **337** total probes on the sample 13x13 array. This is the number reported by **`ms.solver(new algs.hw1.submission.ManhattanSquareFinder())`**that you can see in the `main()` method of **`ManhattanSquareFinder`**.

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

Q2.3 HeisenbergFinder

`h = new Heisenberg(N)` creates a one-dimensional 1xN array, `A`, of random integers in ascending order drawn from the range `[0` to `10*N]` (inclusive). The following is **`h = new Heisenberg(7, 99)`** using random seed 99:

4	10	17	24	28	37	45

You can inspect a given index position using **`int inspect(idx)`** which returns the value of `A[idx]`. However, after every call to `inspect()`, **the values stored in A are perturbed** as follows:

- all values in index positions `0` to `idx–1` are reduced by one
- all values in index positions `idx+1` to `N-1` are increased by one
- the value at index position `idx` is randomly incremented or decremented by one

Copy **`algs.hw1.submission.HeisenbergFinder`** into your `USERID.hw1` package and **improve upon** the default implementation of `int find(Heisenberg h, int target)` which returns an index location, `idx`, if `A[idx] == target`, or `-1` if target does not exist in `h`. Using the **h** object above, **`find(h, 10)`** must return the integer 1.

Once you implement `find()`, execute your class to generate a table showing how many accumulated probes you needed to properly check whether all 10N+1 values in a **`HeisenbergFinder`**(N) object. A total of 10*N+1 ~~probes~~ targets are searched for ~~made~~ in each trial.

Task 2.3 (10 points): Complete `find()` method in **`HeisenbergFinder`** and ensure that on a 1x13 array, the sample trial completes with fewer than 1,000 probes. Even though the Heisenberg object changes after every probe, each of the 10N+1 trials will start with the exact same initial 1xN Heisenberg state. I have provided a sample **`HeisenbergFinder`** that you must modify to work more efficiently.

Hint: Can BINARY ARRAY SEARCH come to your rescue?

Q2.3.1 Bonus (+1 bonus point)

When you execute the original, inefficient `algs.hw1.submission.HeisenbergFinder`, the following table shows the total number of inspections required to run a trial that checks whether all 10N+1 possible values are present.

N	Total Probes by Default HeisenbergFinder
2	41
3	90
4	158
5	245
6	351
7	476
8	620
9	783
10	965
11	1166
12	1386
13	1625
14	1883
15	2160
16	2456
17	2771
18	3105
19	3458

What is the formula HF(N) that predicts the total number of probes in the above table for any N?

Q2.3.2 Bonus (+1 bonus point)

The following reports my Best HeisenbergFinder solution. Can you develop a formula BHF(N) that predicts the values of BHF(N) when N is 1 less than a power of 2, such as 3, 7, 15, or 31?

N	Total Probes by Best HeisenbergFinder
3	61
7	209
15	593
31	1529
63	3729
127	8777
255	20161

Q2.4 FuzzyFinder

A **`FuzzySquare`** object creates a two-dimensional NxN array, `A`, of random integers drawn from the range from `0` to `10*N*N` inclusive that are in ascending order within each row, and each subsequent row contains values greater than values in lower rows. The following **`fs = new FuzzySquare(5, 99)`** using random seed 99.

	c_0	c_1	c_2	c_3	c_4
r_0	4	8	11	17	24
r_1	28	35	40	47	48
r_2	51	57	60	65	71
r_3	74	77	80	85	91
r_4	99	103	111	116	118

This array cannot be directly queried; instead, use **`int probe3x3(r, c, target)`** that inspects a 3x3 region centered at `(r, c)` . The above image visualizes a probe3x3() inspection with r=2 and c=3. It returns a special integer status code:

- `0 [FuzzySquare.FOUND]` – the target value is contained within the 3x3 probe region; using a **`target`** of `60` or `91`, for example, in the above 3x3 region would return this status code.
- `5 [FuzzySquare.NOT_PRESENT]` – the target value cannot be contained in `A`. In the above probe, for example, if target were equal to 45, `FuzzySquare` would report that this value cannot possibly exist, because of the sorted nature of the array and the fact that the probe region has both 40 and 47, but nothing in between. If you search for a target that is less than `0` OR greater than `10*N*N` this status code will also be returned.

You only need to know these status return codes to complete this question to receive its 10 points.

Copy **`algs.hw1.submission.FuzzyFinder`** into your `USERID.hw1` package and complete the implementation of `Coordinate find(FuzzySquare fs, int target)` which returns a `Coordinate` where `target` is found, or **`null`** if target does not exist in `fs`. Using the **fs** object above, **find(fs, 111)** must return a `Coordinate` object of (row=4, column=2)

When you successfully complete **`probe3x3`**`()`, execute your class to generate a table showing how many accumulated probes you needed to properly check whether all 10*N*N+1 values exist in **fs**. A total of 10*N*N +1 ~~probes~~ targets are ~~made~~ searched for in each trial.

Task 2.4 (10 points): Complete `find()` method in **`FuzzyFinder`** and ensure that on a 13x13 array, the sample trial completes with fewer than 200,000 total probes.

Q2.4.1 Bonus (+1 bonus point)

Be warned. I spent about 10 hours solving this bonus problem to get it working right, but it is only worth one additional point. Well done in advance if it takes you less time!

The **`int probe3x3(r, c, target)`** method that inspects a 3x3 region centered at `(r, c)` returns different status codes that you can use to more efficiently search for a target value. Given that `fs.probe3x3(2, 2, target)` was invoked below, there are different status codes that are returned:

	c_0	c_1	c_2	c_3	c_4
r_0	4	8	11	17	24
r_1	28	35	40	47	48
r_2	51	57	60	65	71
r_3	74	77	80	85	91
r_4	99	103	111	116	118

Status Codes

1	1	1	1	1
1				2
2				3
3				4
4	4	4	4	4

- `1 [FuzzySquare.ABOVE]` – Marked with Yellow. The target value is contained above the probe region, which means (a) any row above the 3x3 region; or (b) to the left of the top row for the probe region. [Using the above 3x3 probe region and searching for a value in the range from 0 to 34 would return `ABOVE`].
- `2 [FuzzySquare.M1]` – Marked with Orange. The target value is contained "between" the first row and the middle row of the probe 3x3 range. This means either (a) extending to the right of the 3x3 probe region from the top row of the probe region; or (b) extending to the left of the 3x3 probe region from the middle row of the probe region. [Using the above 3x3 probe region and searching for a value in the range from 48 to 56 would return `M1`].
- `3 [FuzzySquare.M2]` – Marked with Blue. The target value is contained "between" the middle row and the bottom row of the probe 3x3 range. This means either (a) extending to the right of the 3x3 probe region from the middle row of the probe region; or (b) extending to the left of the 3x3 probe region from the bottom row of the probe region. [Using the above 3x3 probe region and searching for a value in the range from 66 to 76 would return `M2`].
- `4 [FuzzySquare.BELOW]` – Marked with Green. The target value is contained below the probe region, which means (a) any row below the 3x3 region; or (b) to the right of the bottom row for the probe region. [Using the above 3x3 probe region and searching for a value in the range from 86 to 10*5*5 (or 250) would return `BELOW`].

Bonus Task (1 points) Complete `find()` method in **`FuzzyFinder`** and ensure that on a 13x13 array, the sample trial completes with 23,402 or fewer total probes.

Q3. Stack Programming And Recursion Exercise (20 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**[2], 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!

Copy the **`algs.hw1.submission.Q3_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
`3 6 +`	`(3 + 6) = 9`
`3 6 + 5 * 8 2 - /`	`(((3 + 6) * 5) / (8 - 2)) = 7.5`
`9 8 7 6 5 + * / -`	`(9 - (8 / (7 * (6 + 5)))) = 8.896103…`

[2] 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)	**Lucas Numbers (A00032)**
$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$	$L_n = \begin{cases} 2 & n = 0 \\ 1 & n = 1 \\ L_{n-1} + L_{n-2} & n > 1 \end{cases}$
Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, …. Note that F_8 = 21 since the first entry is F_0.	**Sequence:** 2, 1, 3, 4, 7, 11, 18, 29, 47 … Note that L_8 = 47 since first entry is L_0.

A fundamental identify for Lucas numbers is that:

$$L_n = F_{n-1} + F_{n+1} \text{ for } n > 1$$

For example, 18 = L_6 = F_5 + F_7 = 5 + 13. One surprising identity[3] is that:

$$F_{x+y} = ((F_x * L_y) + (F_y * L_x))/2$$

For example, F_{16} = (F_7*L_9 + F_9*L_7)/2 = (13*76 + 34*29)/2 = (988 + 986)/2 = 987. You can confirm that this is the appropriate Fibonacci number.

For this assignment, you will improve upon the following standard recursive implementation and provide empirical evidence to show the improvement.

```
static long fibonacci_recursive(long n) {
  numRecursiveCalls++;    // count the number of recursive calls
  // base case
  if (n == 0) { return 0; }
  if (n == 1) { return 1; }

  return fibonacci_recursive(n-1) + fibonacci_recursive(n-2);
}
```

Copy the **`algs.hw1.submission.Q3_Fibonacci`** class into your **`USERID.hw1`** package and complete the implementation so it generates the following table. Columns **Fn** and **Ln** refer to the Nth Fibonacci and Lucas numbers. **Frec** contains the total number of recursive calls needed to compute **Fn** using `fibonacci_recursive()` shown above. **Firec** contains the total number of recursive calls **of both** `fibonacci_improved()` and `lucas_improved()` since they are both needed.

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

[3] https://core.ac.uk/download/pdf/53189783.pdf

N	Fn	Ln	Frec	Firec
0	0	2	1	1
1	1	1	1	1
2	1	3	3	1
3	2	4	5	1
4	3	7	9	9
5	5	11	15	17
6	8	18	25	25
7	13	29	41	41
8	21	47	67	57
9	34	76	109	81
10	55	123	177	105
11	89	199	287	137
12	144	322	465	169
13	233	521	753	209
14	377	843	1219	249
15	610	1364	1973	305
16	987	2207	3193	361
17	1597	3571	5167	425
18	2584	5778	8361	489
19	4181	9349	13529	569
20	6765	15127	21891	649
21	10946	24476	35421	737
22	17711	39603	57313	825
23	28657	64079	92735	929
24	46368	103682	150049	1033
25	75025	167761	242785	1145
26	121393	271443	392835	1257
27	196418	439204	635621	1393
28	317811	710647	1028457	1529
29	514229	1149851	1664079	1681

Q3.2.1 Bonus Question (1 pt)

Develop a formula TR(N) that computes the total number of recursive invocations in column **Frec** above using Fibonacci and/or Lucas numbers.

For N=15, TR(N) = 1973

Q4 Big O Notation

In lecture, I will present the Big O notation used to classify the worst case runtime performance of an algorithm. As an introduction, I have pulled together the resulting total number of probes for each of the implementations from Question 2. These are the TOTAL accumulated probes after a number of search requests were made. Note that I have provided the results for two different FuzzySquare and Heisenberg implementations – my First solution and my Best solution.

N	Manhattan Square	Heisenberg Best	Slicer Best	Heisenberg First	FuzzySquare-Best	FuzzySquare-First
3	31	61	30	90	111	135
4	49	104	64	158	362	700
5	71	139	120	245	1384	2275
6	97	175	192	351	2101	5652
7	127	209	280	476	2825	11851
8	161	274	384	620	3772	22120
9	199	315	522	783	4835	37935
10	241	365	680	965	7951	61000
11	287	408	858	1166	13471	93247
12	337	456	1056	1386	17881	136836
13	391	501	1274	1625	23402	194155
14	449	549	1512	1883	27593	267820
15	511	593	1770	2160	32074	360675
16	577	697	2048	2456	36746	475792
17	647	756	2380	2771	41867	616471
18	721	815	2736	3105	46964	786240

- **HeisenbergFirst** contains the results from `algs.hw1.submission.HeisenbergFinder` default implementation
- **FuzzySquareFirst** contains my first solution to the problem (which will be revealed once HW1 is done)
- **FuzzySquareBest** and **HeisenbergBest** contain the results of my best solutions for these problems (again, you can see the code once HW1 is done).

4.1 Produce Normalized Table for Average probe/request cost [5 pts]

You want to compute the probe/request average, so you will need to go back and determine the total number of locate requests conducted so you can normalize the above values to compute the average number of probes per locate request. This will result in another table which you need to submit as part of your assignment, inside the `WrittenQuestions.txt` file. These values are a good indication as to the runtime performance of each algorithm in solving a problem instance of size N.

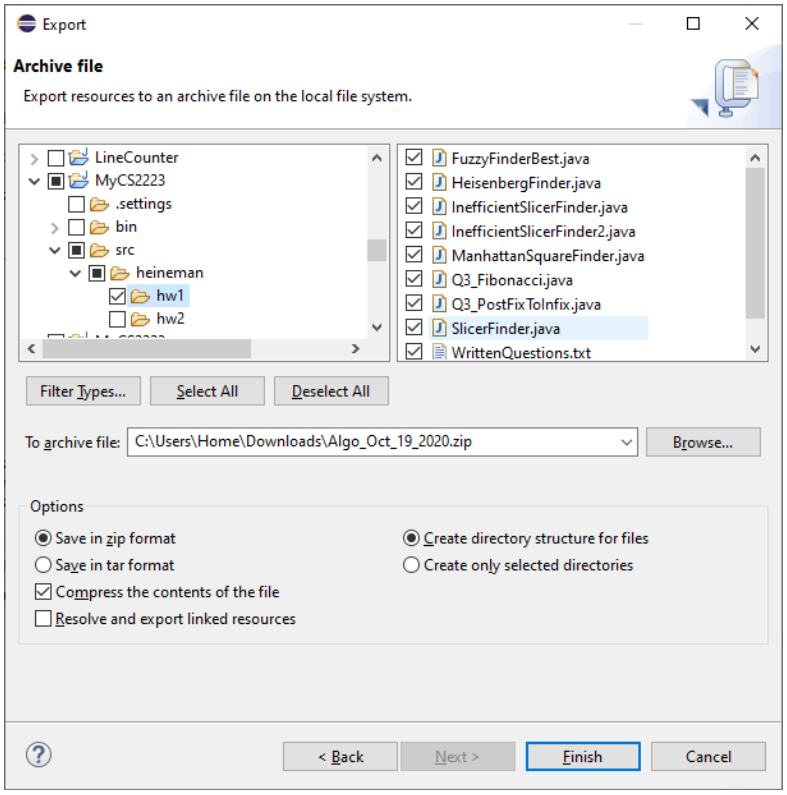
4.2 Classify these different algorithm implementations from empirical data [5pts]

Classify the expected runtime performance for each of these algorithm implementations using the notation I introduce during lectures 4 and 5 (Monday March 29th and Tuesday March 30th).

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**.



Export

Archive file

Export resources to an archive file on the local file system.

- [ ] LineCounter
- [■] MyCS2223
  - [ ] .settings
  - [ ] bin
  - [■] src
    - [■] heineman
      - [✓] hw1
      - [ ] hw2

- [✓] FuzzyFinderBest.java
- [✓] HeisenbergFinder.java
- [✓] InefficientSlicerFinder.java
- [✓] InefficientSlicerFinder2.java
- [✓] ManhattanSquareFinder.java
- [✓] Q3_Fibonacci.java
- [✓] Q3_PostFixToInfix.java
- [✓] SlicerFinder.java
- [✓] WrittenQuestions.txt

Filter Types... | Select All | Deselect All

To archive file: C:\Users\Home\Downloads\Algo_Oct_19_2020.zip | Browse...

Options

- [✓] Save in zip format
- [ ] Save in tar format
- [✓] Compress the contents of the file
- [ ] Resolve and export linked resources
- [✓] Create directory structure for files
- [ ] Create only selected directories

< Back | Next > | Finish | Cancel

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.
2. Also the deadline is MONDAY APRIL 5TH AS IT IS IN CANVAS.
3. Mistake in Question 1 regarding Exponentation. (2 exp 3) should be 8.0\

4. Added “Hint: be sure to explain what happens when processing each of the ‘)’ closing parentheses.” to question 1.7
5. In the description for Heisenberg (2.3) I mistakenly referred to an NxN array, when it should be a 1xN array.
6. Q3 is only worth 20 points (not 30 as it had showed in its header by mistake).
7. For all searching questions, I was using the wrong terms when describing my trials, which are targets to be searched for.