Published Version: 3/21/2023 11:56 AM

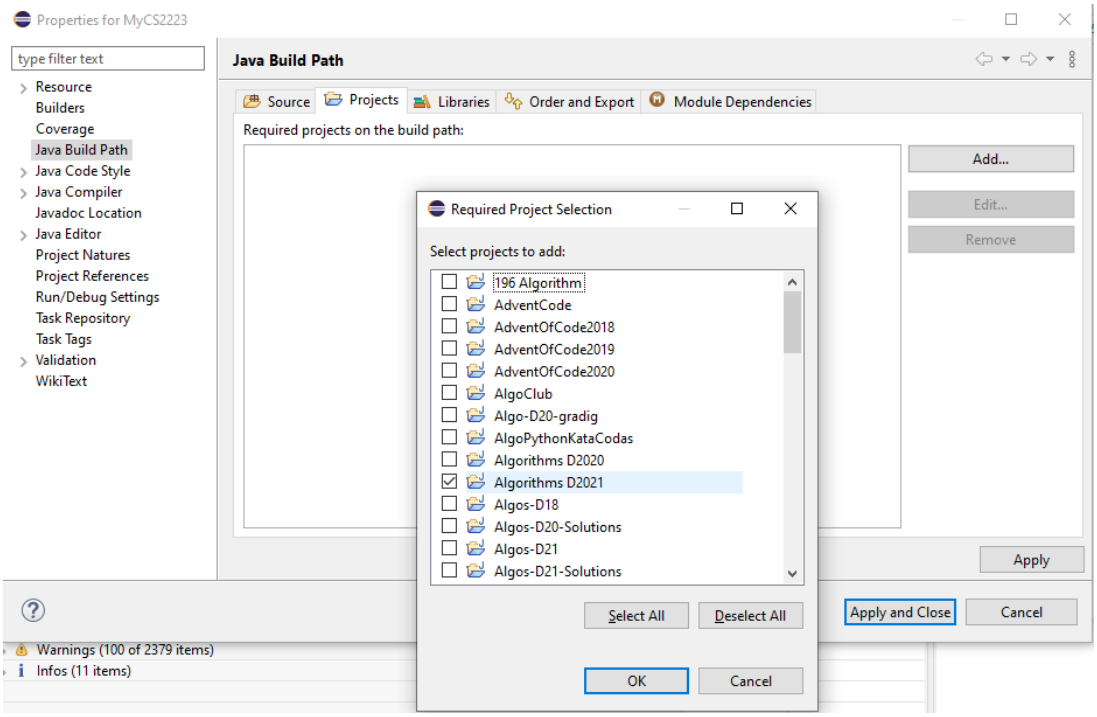
CS 2223 D23 Term. Homework 1 (100 pts.)

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/d23/#policies
- The 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.hw1`** 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 necessary files from the **git** repository that you will modify 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 D2023** project to your build path.



Once done, create the package **USERID.hw1** inside this project, which is where you will complete your work. 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. I will provide a video explanation that goes over this assignment and will post to Canvas.

This homework has a total of **100** 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.

This homework assesses the following skills:

- How to use debugger to inspect information as a program executes
- How to implement a stack in a fixed size array (stacks can grow to the right or grow to the left)
- How to implement a queue in a fixed size array (queues can grow to the right or grow to the left)
- How to use a stack as part of an algorithm
- How to work with two-dimensional arrays
- How to use Binary Array Search in a number of ways: (a) determine whether an ordered array contains a value; (b) When an ordered array contains duplicate values, determine the location of a (possibly duplicated) value with the lowest index; and determine the location of a (possibly duplicated) value with the highest index

This assignment has a total of 100 points (with four potential bonus points)

- Q1 has 20 points
- Q2 has 30 points and 1 bonus point question
- Q3 has 15 points and 1 bonus point question
- Q4 has 25 points and 1 bonus point question
- Q5 has 10 points and 1 bonus point question

All told, you will copy the following files into your **USERID.hw1** package:

- **ComputeRectangle.java**
- **DigitRepresentation.java**
- **EmpiricalEvaluation.java**
- **Evaluate.java**
- **SolveSearch.java**
- **StackConverter.java**
- **Staque.java**
- **Strassen.java**
- **WrittenQuestions.txt**

Published Version: 3/21/2023 11:56 AM

Q1. Stack Experiments (20 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: 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 [].

1.1. **(2 pts.)** Run **`Evaluate`** on input "`( ( 4 + 1 ) / ( ( 8 * 2 ) / ( 3 - 7 ) ) )`" and state the observed output

1.2. **(4 pts.)** **Modify `Evaluate`** to support two new operations:
   a. Add a new binary operation "n **mod** m" that computes n % m, where % is the modulo operator. For example, "5 mod 2" is equal to 1
   b. Add a new binary operation "n **choose** k" which computes the binomial coefficient C(n, k). For example, "5 **choose** 2" is equal to 5!/(3!*2!) = 10. Note that the arguments to **choose** are first converted to integers before being processed. Thus **5.2912 choose 2.18** first becomes **5 choose 2**. This operation has no value if **n** or **k** is negative. When grading we will only use positive integers such that 0 < **k** <= **n**.

1.3. **(2 pts.)** Run your modified **`Evaluate`** on input "`( ( 9 choose 4 ) mod 13 )`" and state the observed output (note that use the 'mod' operator in the input expression!)

The following inputs are **all improperly formatted**, but aren't you curious what will be output? For each of these questions below, be sure to (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. If you set a breakpoint at line 57 in **`Evaluate`** you can use the debugger to find the values.

1.4. **(3 pts.)** Run **`Evaluate`** on input "`( 2 3 * / 5 )`"
1.5. **(3 pts.)** Run **`Evaluate`** on input "`( 6 + + 1 )`" (there is a space between the plus signs)
1.6. **(3 pts.)** Run **`Evaluate`** on input "`- 34`" (there is a space between the minus sign and the 34)
1.7. **(3 pts.)** Run **`Evaluate`** on input "`( 4 * ( 5 + ( 8 + 9`"

Write the answers to these questions in the **`WrittenQuestions.txt`** text file. For question **1.2,** 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 2D arrays of integer values. I have created a helper class **`TwoDimensionalStorage`** which you will use to get the value at a given row and column or set a value in a particular row and column. I have done this because it also records the total number of times any value was read. The purpose of this question is to have you design efficient algorithms that try to minimize the number of array entries that are read.

Q2.1 Searching Values [15 pts]

Consider a **Permutation Array,** a specially constructed two-dimensional array of integers with R rows and C columns that contains each of the R x C integers from 1 to R x C in a permuted arrangement such that the values in each row appear in ascending order. The following is a sample array:

```
int[][] sample = new int[][] {
   { 1, 2, 4,  10, 11},
   { 6, 7, 12, 13, 15},
   { 3, 5, 8,  9,  14},
}
```

Copy the **`algs.hw1.SolveSearch`** into your **`USERID.hw1`** package where you will complete the implementation of the following two functions:

- **`int[] less(TwoDimensionalStorage storage, int target)`** returns the number of integers in each row of the array that are strictly smaller than **`target`**. For example, **`less(sample, 5)`** produces the one-dimensional array of **`int[]{ 3, 0, 1 }`**
- **`int contains(TwoDimensionalStorage storage, int value)`** returns the row that contains value. For example, **`contains(sample, 5)`** returns **`2`**

Once you have a working **`SolveSearch`**, execute it to confirm that it works on this simple example and a smaller trial of ten (if it fails any of these trials then an error is reported.)

Task 2.1 (15 points): Complete implementation of **`SolveSearch`** and execute it to reproduce the following table of runs on specific random 8 x 8 arrays (which appears in the output)

```
[2, 3, 8, 4, 7, 7, 8, 6] for 46
[3, 5, 5, 8, 8, 8, 7, 8] for 53
[1, 1, 2, 3, 4, 1, 1, 1] for 15
[7, 6, 4, 4, 4, 4, 0, 8] for 38
[5, 7, 8, 3, 7, 4, 4, 1] for 40
[1, 3, 0, 1, 2, 3, 2, 6] for 19
[5, 5, 3, 2, 4, 6, 3, 6] for 35
[0, 0, 2, 1, 2, 1, 0, 0] for 7
[8, 8, 3, 6, 2, 5, 3, 2] for 38
[5, 8, 5, 2, 8, 7, 7, 7] for 50
```

It will also produce a leaderboard computation. To get full credit, you need fewer than 1,082,400,000 total inspections. DM to alert me if you outperform the leaderboard champion.

Q2.2 Searching for Rectangles [15 pts]

Consider an **Embedded Rectangle**, a specially constructed two-dimensional array of integers with R rows and C columns where:

- Both R and C are odd values
- The only values stored in the array are 0 and 1 values
- The 1 values form a contiguous `r x c` rectangle, where `r` > R/2 and `c` > C/2; note that `r` and `c` may be odd or even
- All other values in the array are 0

The following R=5 row, C=7 column array is an example with `r`=4 and `c`=5 whose upper-left corner is (`startr`=1, `startc`=2):

```
int[][] sample = new int[][] {
   { 0, 0, 0, 0, 0, 0, 0 },
   { 0, 0, 1, 1, 1, 1, 1 },
   { 0, 0, 1, 1, 1, 1, 1 },
   { 0, 0, 1, 1, 1, 1, 1 },
   { 0, 0, 1, 1, 1, 1, 1 },
}
```

Copy the **`algs.hw1.ComputeRectangle`** class into your **`USERID.hw1`** package and complete the implementation of the **`search(TwoDimensionalStorage)`** method. The goal is to devise an algorithm that computes (**`startr, startc, numRows, numCols`**) given a specially constructed two-dimensional arrays with an **Embedded Rectangle**.

I have provided the implementation of **`algs.hw1.fixed.er.SweepFindRectangle`**, which is a naïve solution you can execute. Your goal is to write code that outperforms this naïve implementation.

Task 2.2 [15 points]: Complete implementation of **`ComputeRectangle`** and execute it to reproduce the following table of runs on specific random 7 x 15 arrays (which appears in the output):

```
start=(0,0) with 5 rows and 14 columns in 11
start=(1,2) with 4 rows and 9 columns in 10
start=(0,0) with 6 rows and 14 columns in 12
start=(0,1) with 6 rows and 11 columns in 11
start=(1,0) with 5 rows and 9 columns in 11
start=(1,1) with 5 rows and 12 columns in 11
start=(2,0) with 4 rows and 12 columns in 11
start=(1,3) with 5 rows and 10 columns in 11
start=(0,1) with 6 rows and 11 columns in 11
start=(1,2) with 4 rows and 11 columns in 10
```

It will produce a computation for the leaderboard. I have provided a simple **`SweepFindRectangle`** algorithm which requires 2,639,301,797 array inspections. To receive half credit for this assignment, your final leaderboard computed number of inspections must be smaller than **326,000,000**. To receive full credit, your total number of inspections must be smaller than **56,000,000**.

Q2.2.1 BONUS Searching for Knight's Path [1 point]

This bonus question is only worth 1 point, so don't even attempt until you have completed the entire assignment. Copy **`algs.hw1.CheckKnightPath`** to your **`USERID.hw1`** package and complete its implementation. Once done execute your code and see if you can beat my performance of 22,249:

```
BE PATIENT. This might take awhile.

A total of 500 boards explored.
31.068 on average for valid 5x5 board
13.43 on average for invalid 5x5 board
Total of 22249
```

Q3. Stack Programming [20 pts.]

The stack data type provides the functions `push()` and `pop()`. While you can determine if a stack is empty, there typically is no way to determine the size of a stack.

For this question your toArray() method can assume that the `FixedCapacityStack` it receives as an argument will never have more than 256 items.

Q3.1 Stack to Array [10 pts.]

It is often useful to take a stack of values and return an array of those values, in order from oldest to youngest, while ensuring that when done the original stack is reconstituted as it was.

```
FixedCapacityStack<Integer> stack = new FixedCapacityStack<>(256);
stack.push(926);
stack.push(415);
stack.push(31);
int vals[] = StackConverter.toArray(stack);

// vals is the array int[] { 926, 415, 31 }
```

Copy **`algs.hw1.StackConverter`** into your **`USERID.hw1`** package and complete its implementation.

- **`static int[] toArray(FixedCapacityStack<Integer> stack)`**

When **`toArray()`** completes, the stack that was passed as an argument **must contain the original elements in the same order**.

Validate that your implementation is correct by executing the **`main()`** method in this class.

Task 3.1 [10 points]: Complete implementation of **`toArray()`**

Q3.2 Working with Stacks [10 pts.]

From (https://en.wikipedia.org/wiki/Numeral_system): In the decimal system (base 10), the numeral 4,327 means (**4**×10^3) + (**3**×10^2) + (**2**×10^1) + (**7**×10^0), noting that $10^0 = 1$. In general, if $b > 1$ is the base, one writes a number in the numeral system of base b by expressing it in the form $a_n b^n + a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + \ldots + a_0 b^0$ and writing the enumerated digits $a_n a_{n-1} a_{n-2} \ldots a_0$ in descending order. The digits are natural numbers between 0 and $b - 1$, inclusive.

The digit sequence for **N** in a given base **10** is computed as follows:

```
power = 10
while n > 0:
  digit = (n % 10)    // use modulo arithmetic to extract digit from 0 .. 9
  print (digit)
  n = n / 10          // divide and truncate
  power = power * 10
```

When this program is run with N=4327, the output will be 7234, which produces the digits in reverse order. Instead of just printing the digits, you must produce a **`FixedCapacityStack<Integer>`**

containing the digits in reverse order, so that popping the digits off and printing them each one at a time would produce the proper left-to-right digit representation.

Copy the **algs.hw1.DigitRepresentation** class into your **USERID.hw1** package and complete the implementation of the **main()** method and the following method:

- **static** FixedCapacityStack<Integer> reverseRepresentation(**int** n, **int** b)

This function will construct a stack containing the digits (in base **b**) of **n** in reverse order such that the topmost value on the returned stack will be the leftmost digit representation of n in base b. Validate that your function works by completing the **main()** method in this class that when executed produces the following table for n=21 and bases from 2 up to and including 10.

```
b      21 in base b
---------------------
2      10101
3      210
4      111
5      41
6      33
7      30
8      25
9      23
10     21
```

Task 3.2 (10 points): Complete implementation of **reverseRepresentation ()** and generate above table.

Q3.2.1 Bonus Question [1 pt]

Consider the digit representation of a number, N, in any base 1 < b < N. Some of these representations are palindromes **with at least two digits**. For example, the number 21 has four palindrome representations, namely 10101_2 and 111_4 and 33_6 and finally 11_{20}. Note it can be hard to write down a digit representation for numbers with very large base **b**, and so instead use integer values to store these digits rather than characters. For example, 96,036,604 in base 200 is equal to $12 \times 200^3 + 183 \times 200^1 + 4$ = 96,036,604 or new **int[] { 12, 183, 4 }.**

Write a program that computes the smallest N that has 50 palindrome representations with at least two digits using base 1 < b < N. A representation is palindromic if it reads the same from left to right as it does from right to left. **What is this value of N?** Submit your program and include in **WrittenQuestions.txt** the value of N you computed.

After looking at the results of this effort, can you see why working with palindromes is not an interesting numeric problem?

Q4. Staque Implementation [25 pts.]

I will describe the implementation of **`FixedCapacityStack`** on **`day03`** of the class. For this question, you are to implement a made-up data type that I call a **`Staque`**, which uses a single storage array to store two independent sub-structures:

- a stack of **char[]** values that starts just to the left of the middle of storage and grows to the left with each push and shrinks to the right with each pop. You can request to push up to 255 **`char`** at a time
- a queue of **`char[]`** values that starts just to the right of the middle of storage and grows to the right with each enqueue. Note that each dequeue shrinks the queue from the left

Copy the **`algs.hw1.Staque`** class into your **`USERID.hw1`** package and complete its implementation. In my lecture on the implementation of stacks and queue data types, I describe how these classes define extra class attributes to record information; you should do the same for your **`Staque`** class.

The **`Staque`** constructor has a single **`size`** parameter which represents the number of **`char`** set aside for the stack (and similarly for the queue); the total size of internal storage array is **`2*size+1`**. For example, after creating a **`Staque`** with **`size=7`**, its storage array of **15 `char`** looks like the following. Note that the **`char`** at address 7 is never used.

--	--	--	--	--	--	--	X	--	--	--	--	--	--	--
0	Stack ←						7	Queue →						14

The **`Staque`** must support the following operations (review the starting file to see the full set):

- **`push(char[])`** – push a sequence of **`char`** values (from 1 to 255 of them) to the top of the stack
- **`pop()`** – pop and return the most recently pushed sequence of **`char`** values
- **`enqueue(char[])`** – enqueue a sequence of **`char`** values (from 1 to 255 of them) to the tail of the queue
- **`dequeue()`** – dequeue and return the oldest enqueued sequence of **`char`** values.

Consider the following sequence of commands using **`c3 = new char[] { 'a', 'b', 'c' }`** and **`c4 = new char[] { 'w', 'x', 'y', 'z' }`**.

- `push(c3)` `-- push 3 char to the stack`
- `enqueue(c4)` `-- enqueue 4 char in the queue`

The internal storage array will be updated as follows:

--	--	--	\3	a	b	c	X	\4	w	x	y	z	--	--
0	Stack ←						7	Queue →						14

As you can see, the length of the respective **char[]** arrays is stored just prior to the actual **`char[]`** values (using the notation **\#** where **#** is the length) that had been push'd (or enqueue'd) to the stack (or

queue). Because we have to stuff an integer (i.e., the length of the **char** sequence) into a **char**, we must limit the size of these **char[]** arrays to 255 characters.

This **Staque** object can support the following two additional requests with **c1 = new char[] {'M'}** and **c2 = new char[] { 'x', 'y'}**.

- `push(c2)` -- push 2 **char** to the stack
- `enqueue(c1)` -- enqueue 1 **char** in the queue

The internal storage array will be updated as follows:

\2	x	y	\3	a	b	c	X	\4	w	x	y	z	\1	M
0	Stack ←						7	Queue →						14

If you execute the following two commands:

- `pop()` -- pop and return **char[]** {'x', 'y'} that had been pushed
- `dequeue()` -- dequeue and return **char[]** {'w','x','y','z'} from head

The internal storage array will be updated as follows:

--	--	--	\3	a	b	c	X	--	--	--	--	--	\1	M
0	Stack ←						7	Queue →						14

You can confirm your implementation works for this example by executing the **main()** method in **Staque**. Your output will look something like this:

```
Note that in the output, the encoded lengths may appear as a strange
character (like ▯) because of how characters appear.

    0:[ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]     <empty>
    1:[ ,  ,  , ▯, a, b, c,  ,  ,  ,  ,  ,  ,  ,  ]     <pushed 3>
    2:[ ,  ,  , ▯, a, b, c,  , ▯, w, x, y, z,  ,  ]     <enqueued 4>
    3:[ ,  ,  , ▯, a, b, c,  , ▯, w, x, y, z, ▯, M]     <enqueued 1>
    4:[▯, x, y, ▯, a, b, c,  , ▯, w, x, y, z, ▯, M]     <pushed 2>
    5:[ ,  ,  , ▯, a, b, c,  ,  ,  ,  ,  , ▯, ▯, M]     <pop'd and dequed>
```

If you are running on Linux or from the command line, your output might look like this:

```
Note that in the output, the encoded lengths may appear as a strange
character (like ^G) because of how characters appear.

    0:[ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ]       <empty>
    1:[ ,  ,  , ^C, a, b, c,  ,  ,  ,  ,  ,  ,  ,  ]       <pushed 3>
    2:[ ,  ,  , ^C, a, b, c,  , ^D, w, x, y, z,  ,  ]     <enqueued 4>
    3:[ ,  ,  , ^C, a, b, c,  , ^D, w, x, y, z, ^A, M]     <enqueued 1>
    4:[^B, x, y, ^C, a, b, c,  , ^D, w, x, y, z, ^A, M]   <pushed 2>
    5:[ ,  ,  , ^C, a, b, c,  ,  ,  ,  ,  , ^A, ^A, M]     <pop'd and dequeued>
```

Q4.1 Bonus Question (1 pt)

The **Staque** queue will eventually become full, because memory for the queue is not reclaimed when elements are dequeued. Implement a circular queue (using ideas from **day05**) but be careful about the edge cases (i.e., it can be confusing to detect the difference between a full queue and an empty queue).

When you complete this implementation, validate it works by writing a class with the following **main()** method that moves values back and forth between the stack and the queue a few hundred times.

```
public static void main(String[] args) {
  Staque stq = new StaqueCircular(64);

  stq.push("parting".toCharArray());
  stq.push("is".toCharArray());
  stq.push("such".toCharArray());
  stq.push("sweet".toCharArray());
  stq.push("sorrow".toCharArray());

  for (int i = 0; i < 500; i++) {
    System.out.println(i + "...");
    while (stq.canPop()) {
      char[] bb = stq.pop();
      System.out.println(String.valueOf(bb));
      stq.enqueue(bb);
    }

    while (stq.canDequeue()) {
      char[] bb = stq.dequeue();
      System.out.println(String.valueOf(bb));
      stq.push(bb);
    }
  }
}
```

Q5. Big O Notation [10 points]

In lecture, I will present the Big O notation used to classify the **upper bound** of the asymptotic complexity of an algorithm. Empirically, I will demonstrate evidence to support the Big O classification of an algorithm on a worst case input, by (a) counting the number of times a key operation executes; (b) timing the run of a Java program. The number of executions of a key operation often correlates directly with the execution time.

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 asymptotic concepts 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 evaluate another problem and use Big O notation to classify its worst-case runtime performance as well as provide empirical evidence of the same.

Given two square matrices, Matrix Multiplication is a binary operation that computes a new matrix from these two matrices (to simplify this question, you will only consider multiplication of square matrices).

The naïve matrix multiplication shown below computes the product of **A** and **B**, and it has been *instrumented* to keep track of the total number of multiplications and additions.

```
public static int[][] multiply (int[][] A, int[][] B) {
  numMultiplications = 0;                          // reset count of multiplications and additions
  numAdditions = 0;
  int [][] result = new int[A.length][B.length];
  for (int rA = 0; rA < A.length; rA++) {
    for (int cB = 0; cB < B[0].length; cB++) {
      for (int cA = 0; cA < A[0].length; cA++) {
        result[rA][cB] += A[rA][cA] * B[cA][cB];
        numAdditions++;                            // instrument to count both the multiplication…
        numMultiplications++;                      // …and the addition
      }
    }
  }
  return result;
}
```

In the naïve multiplication algorithm, the number of additions (`result[rA][cB] += …`) is the same as the number of multiplications (`A[rA][cA] * B[cA][cB]`).

Note: By convention, you do not count the number of times the **for** loop variables are incremented (i.e., `rA++`); you only count the number of additions or multiplications of the array elements.

In 1969, Volker Strassen invented an algorithm that outperforms naive matrix multiplication. Known as *Strassen's algorithm*, the asymptotic complexity for multiplying two N x N matrices where $N=2^n$ is $O(N^{\log_2 7})$ which is approximately $O(N^{2.8074\ldots})$. For this question you will copy **`algs.hw1.Strassen`** into your **`USERID.hw1`** package and instrument it (similar to how you have seen how **`MatrixMultiply`** is instrumented) to account for all the integer multiplications as well as integer additions; for simplicity, you should count both the number of integer subtractions and integer additions in the same counter. Then copy **`algs.hw1.EmpiricalEvaluation`** into your **`USERID.hw1`** and complete it.

Task 5. [10 points]: Complete implementation of **`EmpiricalEvaluation`** and generate following table. In **`WrittenQuestions.txt`** be sure to:

(a) Write a formula that computes #SM Mult given N
(b) Write a formula that computes #SM Add/Sub given N

```
BE PATIENT! This might take a few minutes to run...
           N         #MM MULT          #MM ADD         #SM MULT      #SM ADD/SUB    TIME MM    TIME SM
           4               64               64               49              198   0.000000   0.000000
           8              512              512              343             1674   0.000000   0.000000
          16             4096             4096             2401            12870   0.000000   0.015625
          32            32768            32768            16807            94698   0.000000   0.000000
          64           262144           262144           117649           681318   0.015625   0.046875
         128          2097152          2097152           823543          4842954   0.000000   0.343750
         256         16777216         16777216          5764801         34195590   0.031250   2.296875
         512        134217728        134217728         40353607        240548778   0.265625  16.343750
        1024       1073741824       1073741824                0                0   2.109375  -1.000000
        2048       8589934592       8589934592                0                0  66.000000  -1.000000
```

The first column, **N**, declares the problem size. Column **`#MM MUL`** and **`#MM ADD`** record the number of multiplications and additions for a matrix multiplication problem of size **N**. Similarly, **`#SM MUL`** and **`#SM ADD/SUB`** do the same for **`Strassen`**.

To complete this question, come up with a formula for **`#SM MULT`** and for **`#SM ADD/SUB`**. For example, the entry for **`#MM MULT`** is N^3.

Q5.1 Bonus Question [1 pt]

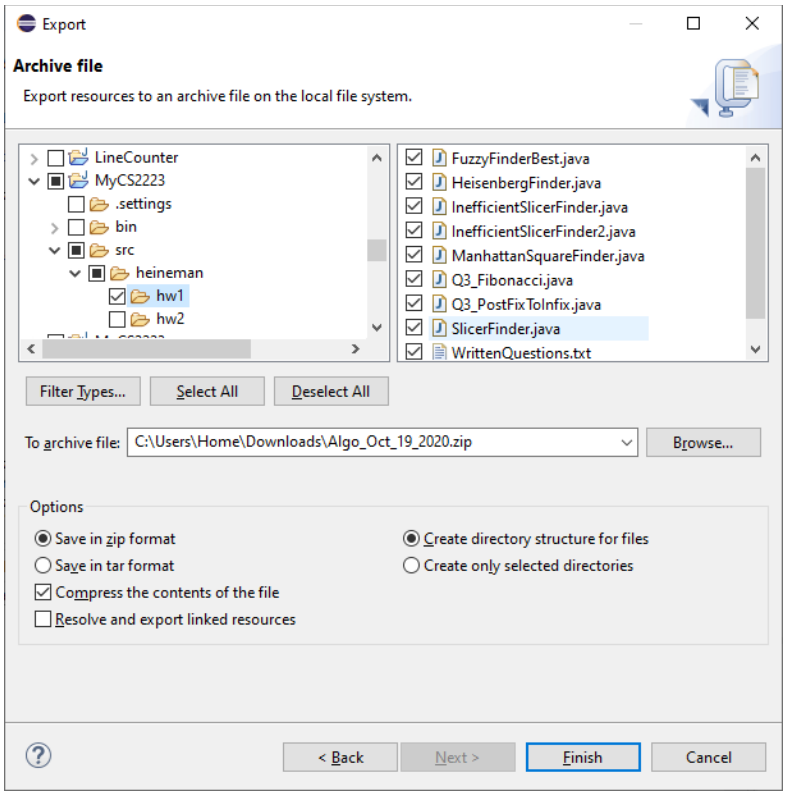
Given the 4x4 array shown below:

```
A = {  {1, 0, 0, 1},
       {0, 1, 1, 0},
       {0, 1, 1, 0},
       {1, 0, 0, 1} };
```

If $\mathbf{A}^N$ represents the product **A** x **A** x … x **A** where **A** is multiplied N-1 times, write a program to compute the value of **A**[1][1] for any N. Can you define this value in terms of N?

Submission Details

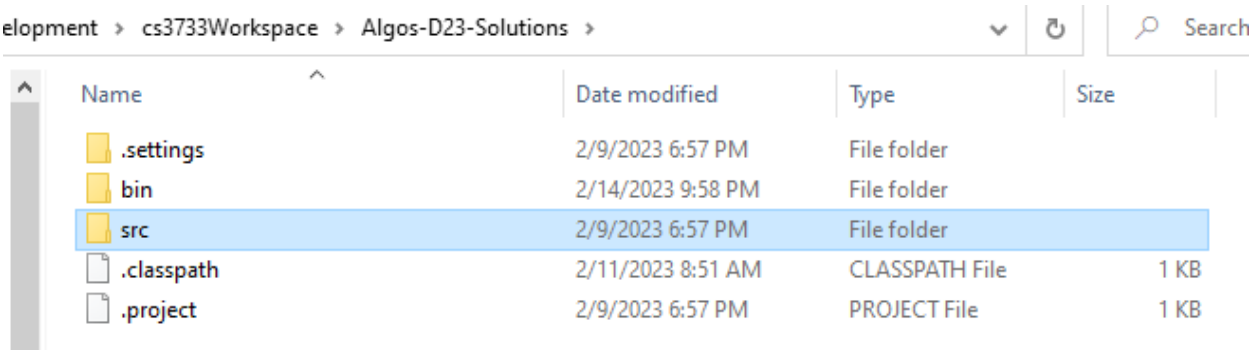
Each student is to submit a single ZIP file that will contain the implementations. In addition, there is a file "WrittenQuestions.txt" in which you are to complete the short answer problems on the homework.



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 as your homework1 submission.

If you would rather just Zip your files from your computer directly, then use the windows explorer to locate the **`src/`** folder which contains your code and zip this folder up and submit it.



Published Version: 3/21/2023 11:56 AM

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. For Q1 you can assume that when the **choose** operator executes its arguments are converted to integers first before being processed. Thus "(5.31 choose 2.11)" is equivalent to "(5 choose 2)"
2. For Q1, make sure spaces between every token
3. Confirm for Q1 that "(n choose k)" will only be called with integer values such that 0 < k <= n. you should convert the given values into integers before attempting to compute this operation. If you want to write a defensive implementation (NO NEED TO DO SO), you could simply have "n choose k" evaluate to 0 if it is given any invalid input.
4. Fixed errant comment regarding the enqueue of c1 which, of course, enqueues a char[] of a single char to the queue.