Project 1

Due Date: Friday 12 February 2021 by 11:59 PM

General Guidelines.

The APIs given below describe the required methods in each class. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment.

Unless otherwise stated in this handout, you are welcome (and encouraged) to add to/alter the provided java files as well as create new java files as needed. Your solution must be coded in Java.

In general, you are not allowed to import any additional classes in your code without explicit permission from your instructor!

Note on academic dishonesty: Please note that it is considered academic dishonesty to read anyone else's solution code, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work! It is also considered academic dishonesty to share your code with another student. Anyone who is found to have violated this policy will be subject to consequences according to the syllabus and university policy.

Note on grading and provided tests: The provided tests are to help you as you implement the project and (in the case of auto-graded sections) to give you an idea of the number of points you are likely to receive. Please note that the points indicated when you run these tests locally are not your final grade. Your solution will be graded by the TAs after you submit. Please also note that these test cases are not likely to be exhaustive and find every possible error. Part of programming is learning to test and debug your own code, so if something goes wrong, we can help guide you in the debugging process but we will not debug your code for you.

Project Overview.

This project is divided into three parts. In Part 1, you will implement some pseudocode and use a count variable to analyze the runtime experimentally. You will then compare that runtime to a mathematical analysis of the runtime. In Part 2, you will implemented an array-based, dynamically re-sizable double-ended queue. In Part 3, you will use the double-ended queue to solve a particular problem involving searching a grid for "treasure."

Part 1. Analyzing Algorithms

Suggested Due Date: Monday 1 February by 11:59 PM

In this part, you will be implementing some short algorithms and running them on different input sizes to analyze the runtime. For each one, the cost model is given for you, and it is up to you to implement an accurate count according to an appropriate cost model. Most of the work on this part will be in the project report.

Note on counts: It is not the intention of this part of the project to drive you crazy with trying to get the counts right. You do need to get the counts right, but if you find that you are having trouble getting them exactly right, don't spend hours trying to figure it out. Just post your code privately to Piazza, and we can give you some quick feedback. A few things that might help:

- Read all the directions, as there are some specific notes here about counts.
- The *swap* function that I use does *not* swap if the two indexes are the same, so that would not add to the array access count.
- The *swap* function that I use uses 4 array accesses when a swap is actually made.

A. In a class called *Part1A.java*, implement the following pseudocode, including an accurate count that reflects the runtime of the algorithm. Your implementation should be in the main method, and the *N* value should be a command-line argument. After the algorithm runs, it should print out the runtime count. To test the accuracy of your count, you can use the values given in the table below.

```
for i from 1 to N:
    j = i
    while j >= 1:
        k = 1
        while k <= N:
            k = k * 2
        end while
        j = j - 1
    end while
end for
```

N	count
1	1
4	30
16	680
64	14560

The rest of the work for this part is in the Project Report.
B. In a class called *Part1B.java*, implement the following pseudocode, including an accurate count that reflects the runtime of the algorithm. Your implementation should be in the main method, and the *N* value should be a command-line argument. After the algorithm runs, it should print out the runtime count. To test the accuracy of your count, you can use the values given in the table below.

```
int sum = 0
while N > 0:
    for j from 1 to N:
        for k from 1 to j:
            sum += j + k
        end for
    end for
    N = N/3
end while
```

N	count
1	1
9	52
81	3751
729	299482

The rest of the work for this part is in the Project Report.

C. In a class called *Part1C.java*, implement the following pseudocode, including an accurate count that reflects the runtime of the algorithm. The cost model should be *the* *number of array accesses*. Your implementation should be in the main method, and the *N* value should be a command-line argument. After the algorithm runs, it should print out the runtime count. To test the accuracy of your count, you can use the values given in the table below. Note that it is up to you to determine what the best and worst case inputs should be and create them in your code. Your best and worst counts should match what are in the table below. You should also run the algorithm on an array of random integers to get an “expected” count. Your random count may not match what is in the table exactly, but it should be close. For generating random values, you may use *java.util.Random.*

```
Algorithm doSomething(int[] A: an array of N integers)
int x = A[0]
int p = 0
int f = 1
for(int i = 1; i < N; i++)
    if(A[i] < x)
        swap A[i] and A[f]
        swap A[p] and A[f]
        p++
        f++
    else if (A[i] == x)
        swap A[f] and A[i]
        f++
end for
end doSomething
```

Note on counting array accesses: You may want to use an extra variable to store the value of A[i] so that you don’t have to access that element more than once per loop. Usually, compilers will do a lot of that kind of work automatically anyway, but since we are counting accesses, we will have to do it explicitly.

Note that the *swap* function exchanges the two values in the array. You will need to implement this function. At most, it should take about three lines of code and 4 array accesses. Be sure to include those accesses in the count.

N	best case count	worst case count	random case count
10	9	45	53

50	49	245	381
250	249	1245	2029
1250	1249	6245	4777

The rest of the work for this part is in the Project Report.

D. In a class called *Part1D.java*, implement the following pseudocode, including an accurate count that reflects the runtime of the algorithm. The cost model should be *the number of array accesses*. Your implementation should be in the main method, and the *N* value should be a command-line argument. After the algorithm runs, it should print out the runtime count. To test the accuracy of your count, you can use the values given in the table below. Note that it is up to you to determine what the best and worst case inputs should be and create them in your code. Your best and worst counts should match what are in the table below. You should also run the algorithm on an array of random integers to get an "expected" count. Your random count may not match what is in the table exactly, but it should be close. For generating random values, you may use *java.util.Random.*

```
Algorithm doSomething(int[] A: an array of N integers)
for i from 1 to N-1:
    j = i
    k = j-1
    while k >= 0 && A[j] < A[k]:
        swap A[j] and A[k]
        j--
        k--
    end while
end for
end doSomething
```

Note on counting array accesses: You may want to use an extra variable to store the value of A[i] so that you don't have to access that element more than once per loop. Usually, compilers will do a lot of that kind of work automatically anyway, but since we are counting accesses, we will have to do it explicitly.

Note that the *swap* function exchanges the two values in the array. You will need to implement this function. At most, it should take about three lines of code and 4 array accesses. Be sure to include those accesses in the count.

N	best case count	worst case count	random case count
10	18	198	78
50	98	4998	2342
250	498	124998	62002
1250	2498	3124998	1573478

The rest of the work for this part is in the Project Report.

E. In a class called *Part1E.java*, implement the following pseudocode, including an accurate count that reflects the runtime of the algorithm. The cost model should be *the number of array accesses*. Your implementation should be in the main method, and the *N* value should be a command-line argument. After the algorithm runs, it should print out the runtime count. To test the accuracy of your count, you can use the values given in the table below. Note that it is up to you to determine what the best and worst case inputs should be and create them in your code. Your best and worst counts should match what are in the table below. You should also run the algorithm on an array of random integers to get an "expected" count. Your random count may not match what is in the table exactly, but it should be close. For generating random values, you may use *java.util.Random.*

```
Algorithm doSomething(int[] A: an array of N integers)
for(int i = N/2; i >= 0; i--):
     foo(A, i)
end for
end doSomething

procedure foo(int[] A, int i)
     int l = 2*i+1
     int r = 2*i+2
     if l >= A.length && r >= A.length:
          return
```

```
        if r >= A.length && A[i] < A[l]:
            swap A[i] and A[l]
        else if A[r] > A[l] && A[r] > A[i]:
            swap A[i] and A[r]
            foo(A, r)
        else if A[l] > A[i]:
            swap A[i] and A[l]
            foo(A, l)
end foo
```

Note that the *swap* function exchanges the two values in the array. You will need to implement this function. At most, it should take about three lines of code and 4 array accesses. Be sure to include those accesses in the count.

N	best case count	worst case count	random case count
10	18	60	34
50	98	370	246
250	498	1938	1646
1250	2498	9930	8406

The rest of the work for this part is in the Project Report.

Part 2. Deque

Suggested Due Date: Friday 5 February by 11:59 PM

A. A Deque is a double-ended queue, which means that it combines the functionality of a Stack with the functionality of a Queue. In this part, you will implement an array-based Deque. **You must use an array to implement this or you will not receive credit!** The Deque should be generic, meaning that it can store any kind of object, and it should be dynamically resizable. The required methods are described below. Do not change any of the method signatures or your code may not work with the test code.

You must also keep track of the number of array accesses for analysis purposes. Two of the methods below have to do with the access count.

Deque API

Method	**Description**
Deque()	constructor: creates an empty Deque with a starting capacity of 10
Deque(int *n*)	constructor: creates an empty Deque with a starting capacity of *n*
void addToFront(Item *item*)	adds *item* to the front of the deque; if the deque is full, first resize it by doubling the size of the array
void addToBack(Item *item*)	adds *item* to the back of the deque; if the deque is full, first resize it by doubling the size of the array
Item getFirst()	remove and return the Item that is at the front of the deque; if the deque is empty, throw an EmptyDequeException; if removing the item makes the size fall below ¼ of the array’s capacity, resize the deque by cutting the array size in half unless half the array size would be less than the initial capacity
Item getLast()	remove and return the Item that is at the back of the deque; if the deque is empty, throw an EmptyDequeException; if removing the item makes the size fall below ¼ of the array’s capacity, resize the deque by cutting the array size in half unless half the array size would be less than the initial capacity
Item peekFirst()	return the Item that is at the front of the deque; if the deque is empty, throw an EmptyDequeException
Item peekLast()	return the Item that is at the back of the deque; if the deque is empty, throw an EmptyDequeException

`boolean isEmpty()`	`return` *`true`* `if the deque is empty and` *`false`* `otherwise`
`int size()`	`return the size of the deque (i.e. the number of elements in the deque)`
`int getAccessCount()`	`return the array access count`
`void resetAccessCount()`	`reset the access count to 0`
`Item[] getArray()`	`return the underlying array for testing purposes`

B. In the *main* method of *Deque.java*, implement an experiment for calculating the amortized runtime of the *addToFront/addToBack* operations. You should have two command line arguments: *N*, the number of operations, and *c*, the starting capacity of the deque array. The experiment should calculate the number of array accesses needed to perform *N addToFront/addToBack* operations on an initially empty Deque with a starting capacity of *c*. Then you should reset the access count and calculate the number of array accesses needed to remove all the elements from the deque. You should run this experiment on many different values of *N* and *c*. You will use the results in the Project Report.

Part 3. Drive-thru Vaccination Simulation

Suggested Due Date: Thursday 11 February by 11:59 PM

In this section, you will use the Deque class to implement a simulation of a drive-thru vaccination site. The following describes the scenario and the variables involved:

- The site can have up to 5 lines (Deques), which means that newcomers could be put at the back of a line or pushed to the front of a line. The number of deques is the first command line argument.
- The site has a set number of vaccines available. This is the second command line argument.
- The number of people who show up is a random number from the interval [0, 1000).
- Time is simulated by a counter.
- When people show up, they are evaluated and assigned a ranking based on their risk level (1 = high-risk, 2 = medium-risk, 3 = low-risk) and are then directed to either the back or the front of one of the lines.
- In the simulation, we keep track of the total number of people who are processed (that is, the number of people who show up), the total number

of people who are vaccinated (keep in mind that the number of vaccines is limited, so not everyone may get a vaccine), the total amount of time it takes to get through everyone, and the statistics by risk level. That is, we we want to know the percentage of people at each risk level who received a vaccination.

- Note that there are two ways the simulation might end:
    - All the people are processed and all the vaccines are used.
    - Fewer people show up than the number of vaccines, so they are all vaccinated, but the simulation ends because of a time limit.
- The main parts of the simulation are:
    - Distribute the vaccines among the lines. This does not have to be an even distribution.
    - Run through the simulation. Each time through the loop, the time variable is increased. Also, a new person shows up and is processed with 75% probability, and we go through each Deque and vaccinate a person with 25% probability. We do this until the simulation ends.

Simulation1.java

You are provided with a basic simulation of this situation to use as a baseline. In this design, the number of Deques depends on a command-line argument (but keep in mind that 5 is the upper limit). The vaccine is distributed evenly (more or less) among the deques, and people are assigned to the back of a line according to the following criteria:

- We loop through the lines to distribute people evenly.
- We keep track of the number of people in the line so that if we know a line will be out of vaccine, we do not add anyone new to that line.

Simulation2.java

Your task in this part is to implement *Simulation2.java.* I recommend that you start by copying *Simulation1.java* into a new file (and changing the names accordingly). Then you will change the overall design with the following goals in mind:

- We want to give vaccination priority to those who are of higher risk. If you run *Simulation1.java*, you'll see that for the most part, the vaccines are fairly evenly distributed among each of the three risk categories. We want to instead try to give first priority to Category 1 and second priority to Category 2.
- You must still limit yourself to using 1 to 5 Deques.
- We still want to minimize the overall time.

Part 4. Project Report

A. Related to Part 1

1. (5.5 points) For each of the pseudocode implementations, run your code on many input sizes and record the results. Then graph the results with the input size on the horizontal axis and the count value on the vertical axis. Please note that you need to have enough values with a large enough range so that the trend in the graph is clear. Also note that hand-drawn graphs may not receive full credit. If applicable, you should have three graphs--one for best-case, one for average case, and one for worst-case. This is only necessary if best and worst case are different.
2. (10 points) For each piece of pseudocode, give a thorough mathematical analysis of the runtime, and compare this to the experimental results with a short discussion. If applicable, include a description of the inputs you used for best and worst case analysis with an explanation of why these inputs produced the best/worst case.

B. Related to Part 2

1. (6 points) Run the main method of *Deque.java* on many values of *N* given each of the following starting capacities. You need to use enough *N* values and a large enough range so that your results can clearly illustrate the correct kind of growth in the array access count. You should do this experiment on each of the following starting capacities:
    a. c = 1
    b. c = 25
    c. c = 100

    For each capacity value, you should show your results in two graphs. The first graph should have *N* on the horizontal axis and the number of array accesses for adding elements to the Deque on the vertical axis. The second graph should have *N* on the horizontal axis and the number of array accesses for removing elements from the Deque on the vertical axis.
2. (2.5 points) In a short paragraph, discuss your results. Were they what you expected? Do the experimental results match with a mathematical analysis of the same question?

C. Related to Part 3

1. (7 points) Explain your approach in your implementation of *Simulation2.java*. What was your set-up and why?
2. (4 points) For at least 5 sets of different input values, run both Simulations and record the results here. Then in a short paragraph, compare your design to the

baseline design (Simulation1). You should make a statement about which design is better and back it up with the results from your experiments.

Submission Procedure.

To submit, please upload the following files to **lectura** by using **turnin**. Once you log in to lectura, you can submit using the following command:

turnin cs345-fall20-p1 Stack1.java Stack2.java Queue.java MaxPQ.java

Upon successful submission, you will see this message:

```
Turning in:
    Stack1.java -- ok
    Stack2.java -- ok
    Queue.java -- ok
    MaxPQ.java -- ok
All done.
```

Note: Your submission must be able to run on **lectura**, so make sure you give yourself enough time before the deadline to check that it runs and do any necessary debugging. I recommend finishing the project locally 24 hours before the deadline to make sure you have enough time to deal with any submission issues.

Grading.

The following rubric gives the basic breakdown of your grade for this Project.

Item	**Points**
Part 1 code compiles and runs	10
Part 2 code: tests from DequeTest	32
Part 2 code: main method of Deque	8
Part 3 code (Simulation2.java) compiles and runs	10
Coding Style	5
Project Report	35
Total	**100**

Other notes about grading:

- If you do not follow the directions (e.g. importing classes without permission, not using an array for the Deque, etc.), you may not receive credit for that part of the assignment.
- Good Coding Style includes: using good indentation, using meaningful variable names, using informative comments, breaking out reusable functions instead of repeating them, etc.
- If you implement something correctly but in an inefficient way, you may not receive full credit.
- If you do not submit code, your Project Report will not be graded.
- In cases where efficiency is determined by counting something like array accesses, it is considered academic dishonesty to *intentionally* try to avoid getting more access counts, so if you are in doubt, it is always best to ask. If you are asking, then we tend to assume that you are trying to do the project correctly.
- If you have questions about your graded project, you may contact the TAs and set up a meeting to discuss your grade with them in person. Regrades on programs that do not work will only be allowed under limited circumstances. All regrade requests should be submitted according to the guidelines in the syllabus and within the allotted time frame.