built on 2024/09/25 at 21:43:55 ***due:*** *wed oct 09 @ 11:59pm*

This assignment will give you practice with class design, references, and object-oriented programming, as well as thinking about the code's performance. You will implement a few Java programs, test them thoroughly, and hand them in. Additionally, you will do some math, write it up, and hand in a PDF. There is a starter package, which you must download to begin working on this assignment.

For each task, write helper functions as you see fit. Strive to write code that promotes clarity.

Typesetting Requirements. For written questions, you must ***properly typeset*** your answer, produce a PDF file called `hw2.pdf`, and add it to the zip file that you will upload to Canvas. No other format will be accepted. To typeset your homework, apart from Microsoft Word, there are LibreOffice and LaTeX, which we recommend. Note that a scan of your handwritten solution will not be accepted. You can find helpful resources about how to typeset your work on the course website.

Our policy on (gen)AI Usage for this assignment:

> Use of generative AI (genAI), including AI-assisted code completion, is prohibited. You must write every element of the code yourself.

When you hand in your work, your zip file will contain *no more than* the following files:

```
hw2.pdf
MinMax.java
RPal.java
mydeque/LinkedListDeque.java.
mydeque/ArrayDeque.java
```

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Be sure to indicate who you have worked with (refer to the hand-in instructions).

Logistics

We're using a script to grade your submission before any human being looks at it. Sadly, the script is not very forgiving. *So, make sure you follow the instructions strictly.* It's a bad omen when the course staff has to manually recover your file because the script doesn't like it. Hence:

- Save your work in a file as described in the task description. This will be different for each task. **Do not save your file(s) with names other than specified.**
- You'll zip these files into a single file called `a2.zip` and you will upload this one zip file to Canvas before the due date.
- Attempt to solve each task on your own first. After a day, if you still can't solve a task, come to office hours.
- Before handing anything in, you should thoroughly test everything you write.
- The course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online.

Task 1: Min and Max (8 points)

For this task, save your work in `MinMax.java`

Consider the following problem: given an array of n numbers, we want to find both the minimum and the maximum of these numbers. For such a problem, we often measure the cost in terms of the number of comparisons made—that is, if we compare any two numbers from the input, that's one comparison.

As an example, the following algorithm requires $n-1$ comparisons:

```
// assume a.length > 0
int maxArray(int[] a) {
    int maxSoFar = a[0];
    for (int i=1;i<a.length;i++) {
        if (a[i] > maxSoFar)
            maxSoFar = a[i];
    }
    return maxSoFar;
}
```

This is because in an array a of length n, *only* $a[1], a[2], \ldots, a[n-1]$ are compared with our `maxSoFar` in the `if` statement. Notice that $a[0]$ is *not* involved in the `if` statement.

You can use this algorithm to the find the maximum value and an almost-identical algorithm to find the minimum value. However, you'll need $2n-2$ comparisons ($n-1$ for max and another $n-1$ for min). *Only comparisons between input integers (either directly or indirectly) matter here,* which is why we don't count comparisons made by `i<a.length` in the above example. Your goal in this problem is to do better!

Subtask I: First, implement a function

```
public static double minMaxAverage(int[] numbers) {
    // your code goes here
    int myMin = ...;
    int myMax = ...;
    return (myMin + myMax)/2.0;
}
```

that takes in an array of integer numbers, finds the minimum and the maximum among these numbers, and returns the average of the minimum and the maximum (as the code above shows). For full credit, if input contains n numbers, your function must use ***fewer*** than $3n/2$ comparisons.

Subtask II: As a comment block in your code file, make a logical argument—as close to an airtight mathematical proof as possible—for why your code is indeed using *strictly* fewer than $3n/2$ comparisons. *(Hint: Remember the maximum-number problem from class? What happens after one round in the pairing-up algorithm?)*

Task 2: Fibonacci Growth (8 points)

For this task, save your work in `hw2.pdf`

In class, our ArrayList implementation uses the array doubling trick, which doubles the capacity of the array every time the array becomes full. Our analysis shows that starting with an array of capacity 1 with no data items initially, appending n data items ends up needing at most $2n$ copying steps.

This problem involves a more fancy array growing scheme. To describe this new approach, recall the Fibonacci sequence from Discrete Math, which is given by

$$F_{n+2} = F_{n+1} + F_n \qquad \text{for } n \geq 0,$$

with $F_1 = F_2 = 1$. In our new scheme, the capacity of the underlying array will strictly follow the Fibonacci sequence. Initially, the capacity is $F_2 = 1$. When full, we grow the capacity to $F_3 = 2$. When full again, we grow the capacity to $F_4 = 3$... then to $F_5 = 5$, then to $F_6 = 8$, and so on.

Amazingly this turns out to work quite well! You'll prove a few facts about Fibonacci numbers and apply them to analyze the total copying steps.

Subtask I Using mathematical induction, prove that for $n \geq 1$,

$$1 + F_1 + F_2 + \ldots F_n = F_{n+2}$$

You're expected to write a solid, rigorous proof based on the definition of Fibonacci numbers (above).

Subtask II Prove, e.g., using direct proof that for $n \geq 1$,

$$\frac{1}{F_n}\left(1 + \sum_{k=1}^{n} F_k\right) \leq 3$$

Subtask III Suppose we start with no data items in our ArrayList. If we use the Fibonacci growth scheme and add in n data items, give a detailed analysis of the total number of copy steps (like we did in class) of this scheme. State your argument carefully. To simplify matters, you may wish to assume that $n = F_r + 1$ for some integer $r \geq 2$.

Task 3: Palindromic and Recursive (8 points)

For this task, save your work in `RPal.java`

This problem will give you more practice in writing recursive programs, in the context of solving a wacky problem. Let $N > 0$ be an integer. We say that a list X of positive integers is a *partition* of N if the elements of X add up to exactly N. For example, each of $[1,2,4]$ and $[2,3,2]$ is a partition of 7.

As you might know already, a list is *palindromic* if it reads the same forward and backward. Of the above example partitions, $[1,2,4]$ is not palindromic, but $[2,3,2]$ is palindromic. What's more, we know that if X is palindromic, then the *first half* (precisely the first `len(X)/2` numbers) is the reverse of the last half (precisely the last `len(X)/2` numbers).

In this task, we're interested in partitions that are palindromic recursively. A partition is *recursively palindromic* if it is palindromic itself and its first half is recursively palindromic or empty. For example, there are 6 recursively palindromic partitions of 7:

```
[7], [1,5,1], [2,3,2], [1,1,3,1,1], [3,1,3], [1,1,1,1,1,1,1]
```

The end goal of this problem is a working implementation of `allRPals(int n)` inside a public class `RPal`. Specifically, the method `allRPals(n)` will return a list of all recursively palindromic lists that sum to `n`. To boost performance, `allRPals` maintains a dictionary (a look-up table) on the side: if it knows the answer already, then it returns that answer right away. Otherwise, it calls upon `computeAllRPals(int n)` to generate the answer.

Your task: Inside a public class `RPal`, implement a private method

```
private List<List<Integer>> computeAllRPals(int n)
```

that returns a list of all recursively palindromic lists that sum to `n`. In your implementation, you are encouraged to call `allRPals` on smaller values of `n`. It is such recursion magic that will help you solve this problem!

Performance Expectations: You will only be tested with $1 \leq n \leq 224$. We expect your code to be reasonably fast. For the largest n (i.e., $n = 224$), your program should not take more than 2.5 seconds.

(Hint: There are 9,042 *partitions that are recursively palindromic for* $n = 99$*. Also, there are* 355,906 *partitions that are recursively palindromic for* $n = 224$*.)*

Additional Challenge: If you feel like learning a new trick, how would you rewrite code inside `allRPals` to achieve the same effect in just a single line using `storageAllRPals.computeIfAbsent(...)`? Read the Java documentation to learn more about `computeIfAbsent`.

Task 4: Deque Using Doubly Linked Lists and Arrays (16 points)

For this task, save your work in `mydeque/`

In this problem, you will build implementations of a "double-ended queue" using both lists and arrays. This extends the discussion of our linked list and array list from class. You can look at a more detailed explanation in Chapter 5 of the book[1].

(The next assignment will deal with repackaging it in a more Java proper way.)

The Deque API

The *double ended queue* is similar to the linked list and array list data structures that you have seen in class. For a more authoritative definition (from cplusplus.com), a *deque* (usually pronounced like "deck") is an irregular acronym of double-ended queue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

For our needs, any deque implementation must have exactly the following operations:

```
// Adds an item of type T to the front of the deque.
public void addFirst(T item)

// Adds an item of type T to the back of the deque.
public void addLast(T item)

// Returns true if deque is empty, false otherwise.
public boolean isEmpty()

// Returns the number of items in the deque.
public int size()

// Returns a string showing the items in the deque from first to last,
// separated by a space.
public String toString()

// Removes and returns the item at the front of the deque.
// If no such item exists, returns null.
```

[1] https://introds.philinelabs.net

```
public T removeFirst()

// Removes and returns the item at the back of the deque.
// If no such item exists, returns null.
public T removeLast()

// Gets the item at the given index, where 0 is the front, 1 is the next item,
// and so forth. If no such item exists, returns null. Must not alter the deque!
public T get(int index)
```

Your class should accept any generic type (not just integers).

Linked List Deque

For this part, save your work in a file called `LinkedListDeque.java`.

Your task is to build a `LinkedListDeque` class, which will be (doubly) linked list based. Your operations are subject to the following rules:

- `.add` and `.remove` family of operations must not involve any looping or recursion. Hence, a single such operation must take "constant time." That is to say, its execution time should not depend on the size of the deque.
- `.get` must use iteration, not recursion.
- `.size` must take constant time.
- You must not have extraneous/dangling nodes. Specifically, the amount of memory that your program uses at any given time must be proportional to the number of items. For example, if you add 1,000 items to the deque, and then remove 999 items, the resulting size should be more like a deque with 1 item than 1,000. This means that you must **not** maintain references to items that are no longer in the deque.

You will implement all the methods listed above in "The Deque API" section (above), together the following constructors:

```
// Creates an empty linked list deque.
public LinkedListDeque()
// Creates a deep copy of other.
public LinkedListDeque(LinkedListDeque<T> other)
```

Note that creating a deep copy means that you create an entirely new `LinkedListDeque`, with the exact same items. However, they are copies so they should be different objects. A good litmus test is, if you change other, the "copied" `LinkedListDeque` should not change.

NOTE: You are not allowed to use Java's built-in `LinkedList` data structure (or any data structure from `java.util.*`) in your implementation.

NOTE #2: We're providing a very simple simple sanity check in `LinkedListDequeTest.java`. For your benefit, you must write more comprehensive tests. Passing the given tests does not necessarily mean that you will pass our test or receive full credit.

NOTE #3: You may wish to implement a `printDeque()` method, which unlike `toString`, will print a detailed view of your internal representation—make it print whatever you wish to see when implementing/debugging the code. This is to help you debug and make sense of your deque structure. You are *not* required to hand this in, but we recommend that you write one to help you work through the task.

Array Deque

For this part, save your work in a file called `ArrayDeque.java`.

As another deque implementation, you'll build an `ArrayDeque` class. This deque must use fixed-size arrays as the core data structure. You'll implement all the methods listed above in the Deque API. Other than that, your operations are subject to the following rules:

- The `.add` and `.remove` family of operations must take constant time, except during resizing (grow and perhaps shrink) operations.
- `.get` and `.size` must take constant time.
- The starting size of your array should be 8. The amount of memory that your program uses at any given time must be proportional to the number of items. For example, if you add 10,000 items to the deque, and then remove 9,999 items, you shouldn't still be using an array of length 10,000ish.
- In addition, for arrays of length 16 or more, your array utilization (the ratio between array cells that are used compared to the total array capacity) always be at least 25%. For smaller arrays, your usage factor can be arbitrarily low.

You will also implement the following constructors:

```
// Creates an empty array deque.
public ArrayDeque()

// Creates a deep copy of other.
public ArrayDeque(ArrayDeque<T> other)
```

Like before, creating a deep copy means that creating an entirely new `ArrayDeque`, with the exact same items as `other`. However, they should be different objects, i.e. if you change other, the new `ArrayDeque` you created should not change as well. You may add any private helper classes or methods in the same file as you see fit.

TIPS #1: The biggest challege for this part is, how to support the add and remove operations in constant time (independent of the size)? We strongly recommend that you learn about the circular buffer. Chapter 5.3 of the book explains this for queues and has some code examples. You might also find more inspirations from Wikipedia (`https://en.wikipedia.org/wiki/Circular_buffer`). That is, you'll treat your array as circular. This means, for example, if your front pointer is at position zero, and you `addFirst`, the front pointer should loop back around to the end of the array (so the new front item in the deque will be the last item in the underlying array). Simiarly, if the rear end of the deque is at the last slot of the array and you `addLast`, it should wrap around and stores the item at position 0 (unless already full, in which case you'd resize).

TIPS #2: Consider not doing resizing at all until you know your code works without it. Resizing is a performance optimization (but it is required for full credit). And when you do resizing, make sure you think carefully about what happens if the data structure goes from empty, to some non-zero size (e.g. 4 items) back down to zero again, and then back to some non-zero size. Pro tip: $0 \times 2 = 0$, but it might not be what you want.

TIPS #3: Chapter 4.3 of the book discusses some idea(s) for resizing both for growing and shrinking the underlying array.

TIPS #4: Like in the linked list version, you may wish to implement a `printDeque()` method, which will print a detailed view of your internal representation—make it print whatever you wish to see when implementing/debugging the code. This is to help you debug and make sense of your deque structure. You are *not* required to hand this in, but we recommend that you write one to help you work through the task.