Project 2: Sorting & Searching

Due Date: Thursday 15 October 2020 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 receive an automatic 0 on the assignment and may be subject to further consequences according to the university's policies.

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 two Parts. In Parts 1 & 2, you will implement two sorting algorithms, which are variations of sorting algorithms we covered in class.

In Part 3, you will extend the Clinic part of Project 1 to allow new methods in the PatientQueue class. Part of this will involve building a Hashtable for quick searching.

Part 1. SortMerge (16 points)

In this part, you will implement a class called *SortMerge*, which uses a variation of MergeSort to sort an array. In the typical *MergeSort* algorithm, an array is recursively divided into halves, which are then recursively sorted and merged together. In this variation, we expect that many of our input arrays are made up of smaller, already sorted arrays, and we seek to take advantage of that by finding the sorted subarrays and merging them together.

Example:

Input: [2, 5, 8, 10, 1, 3, 6, 7, 9, 11, 2, 4, 4, 5, 6, 1, 2]
This array is made up of 4 sorted subarrays: {2, 5, 8, 10}, {1, 3, 6, 7, 9, 11}, {2, 4, 4, 5, 6}, and {1, 2}. With regular MergeSort, the array would be divided by half for each recursive call, regardless of the actual sorted subarrays. In our version, *SortMerge*, we will take advantage of these subarrays and merge them together, following these steps:

1. Find the subarrays (starting and ending indices for each one). This should take no more than $O(N)$ time where *N* is the input size.
2. Merge pairs of subarrays together until the whole array is sorted.

Using the example above, this would look like:

1. Scan the array to find the starting and ending indices for each subarray. These are: (0, 3), (4, 9), (10, 14), and (15, 16).
2. Merge the subarrays together in pairs until the whole thing is sorted.
    a. Merge the first two subarrays together creating a new sorted subarray from index 0 to index 9: {1, 2, 3, 5, 6, 7, 8, 9, 10, 11}
    b. Merge the third and fourth subarrays together creating a new sorted subarray from index 10 to index 16: {1, 2, 2, 4, 4, 5, 6}
    c. Merge the two new subarrays together, sorting the entire array: {1, 2, 2, 2, 3, 4, 4, 5, 5, 6, 6, 7, 8, 9, 10, 11}

Implementation Requirements

- The runtime for your entire implementation should be no worse than $O(NlogN)$.
- The space usage for your implementation should be no worse than $O(N)$. (Hint: You will probably want to use an extra array of size *N*.)
- You may find it useful to use an extra data structure for keeping track of the indices of the subarrays that are to be merged. One way of doing this is with a Queue, so a *Queue.java* class is provided for you. Note that you are not required

to use this, but if you do not, your implementation still has to meet the overall requirements of the project (including not importing unapproved classes!)

API for *SortMerge.java*

static void *sort*(int[] *array*)	sorts *array* from smallest to largest in the method described above

Testing

We provide *SortMergeTest.java* to help you test the accuracy and robustness of your solution. Keep in mind that this test may not be exhaustive, and you should consider adding more robust testing to ensure that you handle all possible inputs correctly and efficiently

Part 2. Five-way Quicksort (16 Points)

In this part, you will implement a class called *FiveQuick*, which uses a variation of Quicksort to sort an array. In the typical three-way *Quicksort* algorithm, the first element is chosen as a pivot and the array is divided into three parts: what is less than the pivot, what is equal to the pivot, and what is greater than the pivot. Then a recursive call is made on the less-than section and the greater-than section.

In *FiveQuick*, you must implement a five-way quicksort algorithm where two pivots are chosen. I recommend you pick the first element as pivot 1 and the last element as pivot 2. Note that you should first make sure that *pivot 1 <= pivot 2* . Then the array is divided into *<less than pivot 1><equal to pivot 1><between pivot 1 and pivot 2><equal to pivot 2><greater than pivot 2>.* Finally, three recursive calls should be made. (You should be able to figure out which sections require recursive calls.) To get started, you may want to review the solutions to the Midterm, as one of those questions gave pseudocode for a pivot method (with a single pivot).

Implementation Requirements

- The runtime for your entire implementation should be no worse than $O(NlogN)$.
- Your algorithm should work in-place.

API for *FiveQuick.java*

static void *sort*(int[] *array*)	sorts *array* from smallest to largest in the method described above

Testing
We provide *FiveQuickTest.java* to help you test the accuracy and robustness of your solution. Keep in mind that this test may not be exhaustive, and you should consider adding more robust testing to ensure that you handle all possible inputs correctly and efficiently.

Part 3. A new Patient Queue (33 Points)

In Project 1, you implemented two methods of managing patients in a Clinic. In this part, you will extend the Priority Queue method to allow for changes to be made efficiently to a patient's status. For example, a patient may experience an emergency while waiting and will need to be seen more quickly. Alternatively, a patient may walk out before being seen. Using just a priority queue to manage this is not efficient because priority queues are not made for searching. So, instead, we will use a priority queue along with a hashtable so that values in the priority queue can be quickly updated. You'll notice that there are a few differences between this part and the stuff you did in Project 1, so be sure to read the instructions carefully.

The Patient class
The *Patient.java* class is already implemented and provided for you. You may add to the class, but do not change any of the variables and methods already implemented. It is highly recommended that you take some time to look through this file to see what operations are already available to you. They may be useful later on. The following API summarizes the methods that are available in *Patient.java*.

`Patient.java API`

`Patient(String name, int urgency, long time_in)`	`constructor: creates a Patient with the given name*, urgency level, and time_in`
`String name()`	`returns Patient's name`
`int urgency()`	`returns Patient's urgency level`
`void setUrgency(int urgency)`	`sets a Patient's urgency level`
`long time_in()`	`returns Patient's time in`
`long compareTo(Patient other)`	`compares two Patients according to prioritization (positive result means higher priority`

	relative to *other*); priority is based on (1) urgency level and (2) time in, meaning that a higher urgency level automatically gets priority; if the urgency levels are equal, an earlier time in gets priority
String toString()	returns a String containing Patient’s name, urgency level, and time in
String toStringWithPos()	returns a String containing Patient’s name, urgency level, and position in queue
String toStringForTesting()	returns a String containing Patient’s name and urgency level
int posInQueue()	returns the index at which the Patient is stored in PatientQueue (-1 if the Patient is not in the PQ)
void setPosInQueue(int pos)	sets Patient’s *posInQueue* variable to the passed in *posy*

*You can assume that Patient names are unique.

The Hashtable

A simple binary heap PQ has good runtime for both *insert* and *delMax*, but this alone is not ideal for the clinic’s queue because sometimes patients who start out with a low emergency level might experience an emergency while waiting. Likewise, some patients may get tired of waiting and walk out. This requires us to be able to *update* patients in the queue and *remove* specific patients from the queue. This means, though, that we need to first find the specific patient in the queue (based on the name), and PQs are not built for fast searching. The good news is that with the help of another data structure, we can do this in reasonably fast time.

In this section, you will implement a basic hashtable that will store the patients for quick searching. The key value is the patient’s name, and you should use the built-in Java hashcode function for Strings, as well as modular hashing.

You will implement the hashtable using separate chaining to handle collisions. However, since we know the capacity of the clinic, we can set the array size to be that capacity, which will make our *expected* runtime for *put* and *get* and *remove* all $O(1)$.

You may use ArrayList objects to implement the chains. You should only initialize a new list when necessary. Otherwise, leave the array element null.

The following methods must be implemented in PHashtable. There are also two helper methods provided for getting an appropriate prime number.

PHashtable.java API

PHashtable()	Constructor: create a new PHashtable with the underlying array having size *11*
Patient get(String name)	return the Patient with the given name from the table; if the Patient is not in the table, return *null*
void put(Patient p)	put the Patient *p* into the table; if the Patient already exists in the table, do nothing; if the number of Patients in the queue becomes 2 times the underlying capacity, resize (and rehash) the table to be the first prime number that is greater than or equal double the capacity (e.g. if the capacity is 11, then after inserting the 22nd patient, you should resize the array to be of size 23 and rehash all the values)
Patient remove(String name)	remove and return the Patient with the given name; if the Patient does not exist in the table, return *null*
int size()	return the number of patients in the table

Note: For the hash function, you should use Java's hashcode function for Strings and use modular hashing on the result to make it fit into the table. However, sometimes you may get a negative number. You should deal with that by adding the size of the modulus to the negative value. For example, if *s* is the String key, and *n* is the size of the hashtable, and $h = s.hashCode()\%n < 0$, you should reset *h* to $h + n$.

A New Patient Queue

NOTE: If you choose to use your own *MaxPQ* class from Project 1, that is fine as long as it works with the test cases. Otherwise, feel free to use the *MaxPQ.java* class provided for you. You will likely need to make a few minor changes to the existing methods, but the bulk of the work will be to implement the two new methods described in the API below.

Now that you have a Hashtable for the patients, you can store pointers to each patient both in the PQ and in the HT. This allows fast searching (by patient name), and once you retrieve the Patient object, you can access the patient's position in the PQ and thereby access it quickly, allowing for more advanced operations in the PQ in reasonably fast time.

NOTE: Assuming that *get* and *put* and *remove* are all expected to be $O(1)$ operations in your hashtable (a reasonable assumption based on the specifications), each of the following functions should be no worse than $O(logN)$ where *N* is the number of patients in the PatientQueue. Also, remember, that the binary heap must always be in the correct order after every operation!

`MaxPQ.java API (advanced)`

Patient remove(String s)	remove and return the Patient with name *s* from the PatientQueue
void update(String s, int urgency)	update the emergency level of the patient with name *s* to *urgency* in the PatientQueue

The Clinic

The Clinic class puts everything together to simulate the clinic. This time we include patients who experience emergencies and patients who walk out before being seen. This is already implemented for you, but you should familiarize yourself with the class.

Clinic.java API

Clinic(int er_threshold)	Constructor: create a new Clinic with the given *er_threshold*
int er_threshold()	return the *er_threshold*
String process(String name, int urgency)	Process a new patient with the given name and urgency level; if the patient's urgency level exceeds *er_threshold* send them directly to the ER and return null; otherwise, add the new Patient into the queue and return the Patient's name
void seeNext()	The patient with highest precedence that is currently in the queue will get seen by a doctor; return the name of the patient who is seen (or null if the queue is empty)
boolean handle_emergency(String name, int urgency)	Patient with name *name* experiences an emergency, causing their urgency level to increase to *urgency;* if their urgency level exceeds *er_threshold,* send them to the ER and return true; else, update their *urgency* level in the PatientQueue and return false (i.e. return true if they are removed from the queue)
void walk_out(String name)	Patient with name *name* walks out; remove them from the PatientQueue

The private methods *sendToER(Patient p)* and *seeDoctor(Patient p)* are provided for you and should not be changed! There are also getter methods for the four counter variables that should not be changed.

Testing

You are provided with the following test files.

- Text files:
    - *patient_file_1.txt*

- *PHashtableTest.java*: This tests *PHashtable.java*.
- *NewPatientQueueTest.java:* This tests *NewPatientQueue.java*.
- *Simulation.java*: Run simulation to see the results of using your new classes to simulate a Clinic.
  `<er_threshold> <pWalkin> <pDocAvailable> <pEmergency> <pWalkOut>`
    - `er_threshold: the threshold emergency level for sending Patients to the ER (an integer)`
    - `pWalkin: the probability (in percentage form) that a patient will walk in (i.e. 45 for 45%)`
    - `pDocAvailable: the probability (in percentage form) that a doctor is available (i.e. 56 for 56%)`
    - `pEmergency: the probability (in percentage form) that a patient experiences an emergency (i.e. 17 for 17%)`
    - `pWalkOut: the probability (in percentage form) that a patient walks out (i.e. 34 for 34%)`

It is recommended that you run the simulation several times with several different sets of arguments in order to make sure your results look correct and to help you with your report.

Part 4. Project Report (27 Points)

For the project report, you should provide typed responses to the following questions and submit it as a PDF to Gradescope. Keep in mind that your report will not be graded if you do not submit code.

Questions about Part 1

(1) Explain briefly how your algorithm finds the subarrays and give the runtime of that portion of the algorithm.
(2) Explain briefly how your algorithm knows which subarrays to merge. Did you use a Queue as suggested or manage it in another way? How does this affect the runtime of your overall algorithm?
(3) Fully analyze your SortMerge algorithm. This means discussing the best and worst case (both the runtime and the type of input that would produce the best and worst case). Be sure to discuss all parts of the algorithm, even if the runtime of a particular part might be insignificant in the big-Oh.
(4) Compare this algorithm to regular MergeSort. Do you think this algorithm is better or worse than Mergesort? Explain your reasoning.

Questions about Part 2

(1) Analyze the runtime of Five-way Quicksort. Your analysis should include a discussion of both the best and worst case runtimes. The discussion of each should include a recurrence relation (explained), the runtime in big-Oh notation (explained), and a description of inputs that would result in those runtimes.
(2) Compare Five-way Quicksort to Three-way Quicksort. Do you think one is better than the other? Explain your reasoning.

Questions about Part 3
(1) In this part, you added two methods to the MaxPQ: *remove* and *update*. Explain why we needed the hashtable to allow these to be done in logarithmic time.
(2) Analyze the runtime of each of the following methods in *Clinic.java* with respect to the data structures being used. Let the number of patients currently in the Clinic be *N*. You should discuss best, worst, and expected runtime.

- *process*
- *seeNext*
- *handle_emergency*
- *walk_out*

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-p2 SortMerge.java FiveQuick.java PHashtable.java MaxPQ.java
```

Upon successful submission, you will see this message:

```
Turning in:
        SortMerge.java -- ok
        FiveQuick.java -- ok
        PHashtable.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.

SortMerge Tests	16 points
FiveQuick Tests	16 points
PHashtable Tests	14 points
MaxPQ Tests	12 points
Coding Style	5 points
Project Report	27 points

Other notes about grading:

- Any violation of the academic integrity guidelines will result in a 0 on the assignment.
- If you do not follow the directions (e.g. sorting with another sorting method rather than the one described in the handout), you may not receive credit for that part of the assignment.
- Good Coding Style includes: using 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.
- 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.