Version: 11-11-2015 11:10 PM

CS 2223 B15 Term. Homework 2

Homework Instructions

- This homework is to be completed individually. If you have any questions as to what constitutes improper behavior, review the examples as I have posted online http://web.cs.wpi.edu/~heineman/html/teaching_/cs2223/b15/#policies .
- Due Date for this assignment is 2PM November 13th. Homeworks received after 2PM receive a 25% late penalty. Homeworks received after 6PM will receive zero credit.
- Submit your assignments electronically using the blackboard site for CS2223. Login to **my.wpi.edu** and go to CS2223 under "My Courses" then go to "Assignments" and submit your homework under "HW2". 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 packager USERID where USERID is your CCC user id.

Q1. Sorting Experiments (30 pts)

The best way to evaluate comparison-based sorting algorithms is to empirically evaluate their performance on randomized trials. You are to develop a program, similar to what you see in the book, which reports essential statistics of the following sorting algorithms:

- Selection Sort
- Insertion Sort
- MergeSort
- QuickSort as presented in the book using the `partition` function shown on p. 291
- QuickAlternate as presented in the book using alternate partition as included in repository. Find in algs.hw2.QuickAlternate

For each algorithm, you are to execute T=10 trials and report the lowest range ~~(low and high)~~ of the following statistics for input size ranging from 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, and 8192. The relevant statistics are:

- Number of `exch` invocations
- Number of `less` comparisons
- Stopwatch performance for time of execution

You will sort randomized collections of uniformly distributed floating point values, as constructed using the **`StdRandom.uniform()`** method call (as you see on p. 256 of the book). As a side note, observe that all sorting algorithms presented sort "in place", so you will need to generate a new array of values for each invocation of sort.

Version: 11-11-2015 11:10 PM

To get started on this question, copy the relevant files from the **`algs.days.dayNN`** packages that already implement the various sorting algorithms. Then you should modify them to record the number of **`exch`** invocations as well as the number of **`less`** operations.

Hint: You need to determine the proper place in your code where you reset your counters for these operations.

(15 pts) The report should following the following output: ~~TBA~~. I have provided a template class, **`SortComparison`**, in **`algs.hw2`** which you should copy into your own project and modify accordingly. This class will properly produce the output that is expected.

(15 pts) For each algorithm, I am asking you to specifically investigate the number of comparisons made by each algorithm and identify the order of growth based on its performance. That is, what is the Tilde equation that you would use to define the number of comparisons made by each algorithm on a data set of size N.

Q2. Data Type Exercise (35 pts)

This assignment gives you a chance to demonstrate your ability to program with Linked Lists. You are to implement the following data type which maintains a unique bag of items, that is, it contains no duplicates.

```
package USERID.hw2;

public class UniqueBag<Item extends Comparable<Item>> {

  class Node<Item> {
    private Item       item;
    private Node<Item> next;
  }

  public UniqueBag() { }
  public UniqueBag(Item[] initial) { }
  public int size() { ... }
  public boolean identical (UniqueBag other) { ... }
  public Item[] toArray() { ... }
  public boolean add(Item it) { ... }
  public boolean remove(Item it) { ... }
  public boolean contains(Item it) { ... }
  public UniqueBag<Item> intersects(UniqueBag<Item> other) { ... }
  public UniqueBag<Item> union(UniqueBag<Item> other) { ... }
```

The implementation must conform to performance specifications that are included in the sample template found in **algs.hw2** in the Git repository. More documentation is found in the sample file.

You are to write benchmark code that evaluates the execution performance of these methods as well.

We will validate the output against a set of test cases that we develop for the grading. Individual breakdown of points is found on the rubric. We will release the test utility that will validate the correctness of your implementation as well as the execution performance.

Q3. Heap Exercise (10 pts)

The heap data structure can be used to implement a Max Priority Queue (p. 318). From this base code, add the resizing logic that we used earlier for allowing a queue to support arbitrary-sized data. There are a number of propositions (see p. 319) that you can empirically validate if you instrument the code to count the number of comparisons during execution:

- In an N-key priority queue, the heap requires no more than *1 + log N* compares for insert
- In an N-key priority queue, the heap requires no more than *2 Log N* compares for remove maximum

Modify the **`HeapExercise`** class. You will process randomized collections of uniformly distributed floating point values, as constructed using the **`StdRandom.uniform`**`()` method call (as you see on p. 256 of the book).

You are to execute T=10 trials and report the highest range ~~(low and high)~~ of the above statistics for input size ranging from N=4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192. Complete this evaluation as follows:

1. Randomly construct a Heap priority queue containing N values using **StdRandom**.
2. Once constructed, perform 1000 iterations of the following sequence
    a. Remove maximum (during which you should count the number of comparisons)
    b. Insert random value (during which you should count the number of comparisons)

You are to report on the highest range ~~(low and high)~~ of these statistics for each of the Heap sizes. Do your statistics support the propositions stated above?

Q4. MergeSort Variations (25 pts)

MergeSort as constituted uses 2-way merges. Modify **MergeSort** to create a 3-way merge. That is, the sort method divides the array `a[lo..hi]` into thirds, sorts each one, and combines uses a 3-way merge.

Hint: Try to pattern your merge operation on the 2-way merge. Thus it will have the Java signature of:

```
// 3way-merge sorted results a[lo..left] with a[left+1..right] and
// a[right+1..hi] back into a
static void merge (Comparable[] a, int lo, int left, int right, int hi) { … }
```

You don't have to handle any special cases within the `sort()` method if your merge works as it should.

You are to:

- **(15 pts)** Write a complete **MergeSortThreeWay** class that implements the 3-way merge sort
- **(10 pts)** Develop two propositions to match the **F** and **G** propositions that you find on page 272 and 275 of the book. These will compute the number of compares C(n) needed to sort an array of length N, where you can assume N is a power of three, or 3^k. And then compute A(n), the maximum number of array access to sort an array of length N.