CS 323 Homework 3

Due by Thursday, February 16, 2017 8:45 PM

Submission instructions

Submit your assignment through the QTest system, using course ID: **CS323** and exam ID: **hw03**. Write all your code into the code boxes provided in QTest, and make sure that it works correctly by pressing the "Execute" button. If your program is composed of multiple classes, make the first one public and all the other ones not public (just omit the public visibility modifier for all classes except the first one). No email submissions are accepted. No late submissions are accepted. Include a collaboration statement in which you acknowledge any collaboration, help, or resource you used or consulted to complete this assignment. This section must be written even if you worked on the assignment alone.

1 Heaps

Implement the following methods to visualize and manipulate binary heaps. All the methods are static, i.e., this is not and Object Oriented program. You will call your methods directly from the main method of your test class. Notice that in the traditional explanations of heaps (e.g., Chapter 6 of your textbook) array indexes start at 1; however in Java array indexes start at zero. Your program should correctly handle arrays starting from index 0.

- **(10 points)** `public static String toTreeString(String[] x, int n)` This method takes an array of strings `x` and an integer `n` representing the number of elements actually used in `x` (assume `n` $\leq$ `x.length`). Assuming that the array x represents a binary heap, this method returns a string representation of the current heap in the following recursive format:
  `(key (left-subtree) (right-subtree))`
- **(5 points)** `public static void buildMaxHeap(String[] x, int n)` This method takes an array of strings `x` and an integer `n` representing the number of elements actually used in `x` (assume `n` $\leq$ `x.length`). The method transforms array `x` in-place into a max-heap.
- **(5 points)** `public static void buildMinHeap(String[] x, int n)` This method takes an array of strings `x` and an integer `n` representing the number of elements actually used in `x` (assume `n` $\leq$ `x.length`). The method transforms array `x` in-place into a min-heap.
- **(10 points)** `public static boolean addToHeap(String s, String[] x, int n)`. This method takes a string `s`, an array of strings `x` representing a valid binary heap, and an integer `n` representing the number of elements actually used in `x` (assume `n` $\leq$ `x.length`). The method automatically detects if `x` is a max-heap or a min-heap, and adds string `s` to it. If there is not enough data to automatically detect if it is a max-heap or min-heap, make it a min-heap by default. The method returns `true` if the insertion was successful, or `false` if there was not enough extra space in the array to insert the new element.

(5 points) In the main method of your program, write plenty of test cases to accurately demonstrate the functionality of your program.

2 Heap Sort, Merge Sort, Selection Sort

Implement the following methods:

- **(10 points)** `public static void heapSort(String[] x, int n, boolean descending)` This method takes an array of strings `x`, an integer `n` representing the number of elements actually used in `x` (assume `n` $\leq$ `x.length`), and a boolean `descending`. Using the Heap Sort algorithm, the method sorts the array `x` in-place. If `descending` is `true`, the elements are sorted high to low; otherwise they are sorted low to high.
- **(10 points)** `public static void mergeSort1(String[] x, int n, boolean descending)` This method takes an array of strings `x`, an integer `n` representing the number of elements actually used in `x` (assume `n` $\leq$ `x.length`), and a boolean `descending`. Using the Top-Down Merge Sort algorithm, the method sorts the array `x` in-place. IIf `descending` is `true`, the elements are sorted high to low; otherwise they are sorted low to high.
- **(10 points)** `public static void mergeSort2(String[] x, int n, boolean descending)` This method takes an array of strings `x`, an integer `n` representing the number of elements actually used in `x` (assume `n` $\leq$ `x.length`), and a boolean `descending`. Using the Bottom-Up Merge Sort algorithm, the method sorts the array `x` in-place. If `descending` is `true`, the elements are sorted high to low; otherwise they are sorted low to high.
- **(10 points)** `public static void selectionSort(String[] x, int n, boolean descending)` This method takes an array of strings `x`, an integer `n` representing the number of elements actually used in `x` (assume `n` $\leq$ `x.length`), and a boolean `descending`. Using the Selection Sort algorithm, the method sorts the array `x` in-place. If `descending` is `true`, the elements are sorted high to low; otherwise they are sorted low to high.

(5 points) In the main method of your program, write plenty of test cases to accurately demonstrate the functionality of your program.

3 Comparison of Sorting Algorithms

Implement the following methods:

- **(2 points)** `public static String[] randomArray(int n, int m)` This method creates and returns an array of `n` random strings of `m` random alphanumeric characters each (alphanumeric characters: A-Z, a-z, 0-9).
- **(2 points)** `public static double measureHeapSort(int n)` This method measures the total execution time (in seconds) of your `heapSort` method by repeating 1,000 times the following cycle:

  1. Create an array of `n` random strings of 5 characters each.
  2. Record the start time.
  3. Call the method to be measured.
  4. Record the end time.
  5. Calculate the difference between end and start time, and add this quantity to an accumulation variable.

  At the end of the loop, the accumulation variable is returned. Notice that the time taken to create the random input arrays is not included in the total.

- **(2 points)** `public static double measureMergeSort1(int n)` This method measures the total execution time (in seconds) of your `mergeSort1` method in the same way as before.
- **(2 points)** `public static double measureMergeSort2(int n)` This method measures the total execution time (in seconds) of your `mergeSort2` method in the same way as before.
- **(2 points)** `public static double measureSelectionSort(int n)` This method measures the total execution time (in seconds) of your `selectionSort` method in the same way as before.

(10 points) Use your measurement methods to run experiments with various values of `n` (collect measurements for at least 5 different values of `n` for each algorithm). Plot those measurements on a chart, and verify that your experiments are compatible with the expected big-O asymptotic complexity of the respective algorithms. Include your chart and a brief discussion of the results in a PDF file generated using LaTeX.

Grading criteria

- Programs that do not compile in QTest get zero points.
- -10 points for missing collaboration statement.
- -10 points for missing comments or bad usage of comments.