Project 3: Sorting

Handed out: **October 5, 2012**
Due: **October 18, 2012 at 11:59pm**

Description

This project is concerned with sorting. The goal is two-fold: First, you will acquire hands-on experience with an advanced sorting algorithm. Second, you will learn how to use sorting to solve a practical problem efficiently. In the first section you will need to implement an enhanced version of the quick sort that we briefly saw in class. In addition, you will visualize your sorting steps to give you insight into the progression of the algorithm. In the second part you will use sorting to solve a real world application.

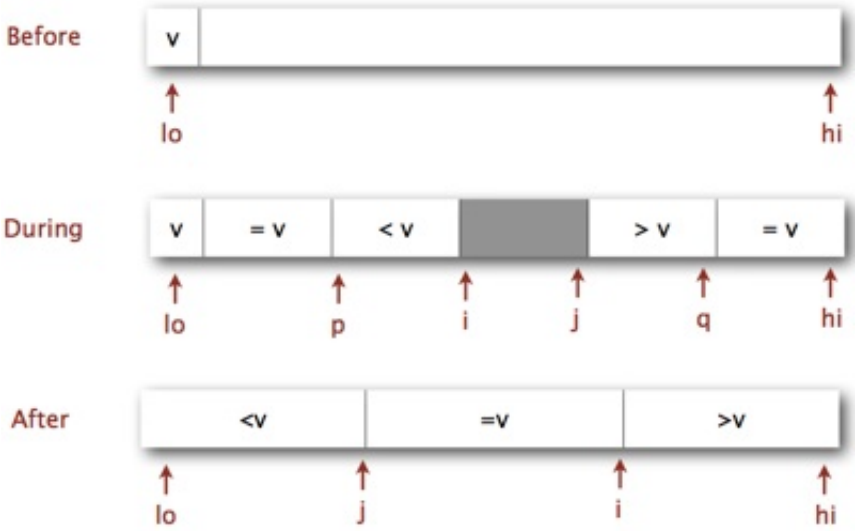
Part 1 - Fast three-way partitioning. (J. Bentley and D. McIlroy)

Algorithm

You will implement an entropy-optimal sort Quick3wayBM.java based on keeping equal keys at both the left and right ends of the sub-array `a[lo..hi]`. Practically, you need to maintain indices that enforce following invariants:

- indices `p` and `q` are such that `a[lo..p-1]` that `a[q+1..hi]` are all equal to `a[lo]`,
- an index `i` is such that `a[p..i-1]` are all less than `a[lo]`
- an index `j` is such that `a[j+1..q]` are all greater than `a[lo]`

These notations are illustrated below.



In your code you should add to the inner partitioning loop code to swap `a[i]` with `a[p]` (and increment `p`) if it is equal to `v` and to swap `a[j]` with `a[q]` (and decrement `q`) if it is equal to `v` before the usual comparisons of `a[i]` and `a[j]` with `v`. After the partitioning loop has terminated, add code to swap the items with equal keys into position.

Improvements

In your code you must implement the following improvements:

• Cutoff to insertion sort. Its usually pays off to switch to insertion sort for tiny arrays. The optimum value of the cutoff in your code MUST be 8.

• Median-of-three partitioning. A second improvement is to use the median of a small sample of items taken from the array as the partitioning item. Doing so will give a slightly better partition, but at the cost of computing the median. It turns out that most of the available improvement comes from choosing a sample of size 3 (and then partitioning on the middle item). You should ONLY use this improvement if the current number of elements to sort is (<= 40).

• Tukey ninther. Use *Tukey ninther* technique as partitioning element when the previous optimizations do not apply.

Input / Output

Your program should read the input from the standard command line. It should process the input using the following command line:

```
% java -cp .:stdlib Quick3wayBM.java < numbers1.txt
```

The file contains a list of numbers in the following format (<number><space>). The first number is the number of elements on the file.

In each recursive sorting call of the array you have to output the following as a tuple following the exact format shown below. Your program should simply print the output to the command line. In the case you used the insertion sort just output:

```
(Insertion_Sort, <lo>, <hi>)
```

otherwise, you should output:

```
(<Improvement_used>, <lo>, <v>, <p>, <i>, <j>, <q>, <hi>)
```

where `<Improvement_used>` will be either "`median of 3`" or "`Tukey ninther`", depending on the case.

As an example of the former scenario your program should output:

```
(Insertion Sort, 1, 4)
```

As two example of the latter scenario your program should output:

```
(median of 3, 1, 10, 4, 15, 16, 20, 25)
(Tukey Ninether, 6, 20, 4, 18, 17, 50, 70)
```

This output should take place before you move the equal values from the two ends of the array to the middle.

Visualization

In this programming assignment you will use the **StdDraw** library included in the stdlib.jar. You will augment your Quick3wayBM.java code and submit another file that provides visualization of the sorting steps, call it Quick3wayBM_V.java. In other words, your Quick3wayBM.java should ONLY provide text output while your Quick3wayBM_V.java should graphically visualize the sorting step WITHOUT text output.

To get a full grade in this part of the assignment you MUST adhere to the following guidelines (you might find it helpful to check some of the attached examples while reading this part):

✓ Your visualization should represent the elements as a histogram.
✓ When drawing a line that is proportional to the element scale the (y) parameter by (0.3).
✓ You should start by visualizing the current elements in their order and end up with all the elements in their sorted order.
✓ There should be no visualization if you use the insertion sort in any step.
✓ At each sorting step, visualize the elements after moving the equal elements in the two ends of the array to the middle.

At each sorting step, use the **black color** (StdDraw.BLACK) to draw a line representing the current portion of the array values your considering. Use the **red color** (StdDraw.BOOK_RED) to shoe the elements equal to v after putting them in the middle. Use **light grey color** (StdDraw.LIGHT_GRAY) to represent the elements that you're not looking at in this round.

Implementation

Implement following API. In addition, implement a main function that reads from the standard input a list of numbers separated by a space. The first number represents the number of integers to be sorted, followed by those integers.

```
public class Quick3wayBM {
    public static void sort(Comparable[] a) { ... }
    private static void sort(Comparable[] a, int lo, int hi) {
        ...
        //insertion sort
        ...
        // median-of-3
        ...
        // use Tukey ninther as partitioning element
        ...
        // Bentley-McIlroy 3-way partitioning
        ...
    }
```

```
}
```

Setting up the visualization parameters.

You must use the following code to set up your visualization parameters. **Note**: the values highlighted in red have been modified on 10/11/2012.

```
StdDraw.setCanvasSize(800, 1400);
StdDraw.show(0);
StdDraw.setXscale(-1, N+1); // N is the number of integers
StdDraw.setPenRadius(.006);
StdDraw.show(0);
```

In addition, using the following parameters to set up the Y-Scale whenever you want to print:

```
StdDraw.setYscale(-(21)*(21-frame), (21)*frame + 10);
```

Where frame must be increased by 1 every time you print. It MUST start at 0.

To make sure that your visualization follows the guidelines correctly, use the examples listed below and save the output as jpg format. Then use diff between the sample visualization and the visualization your program output and you saved. The diff command should return nothing if your output is correct.

Testing with an example

With this document there is three examples of numbers as a way for you to test your code and make sure you are following the right format. These files are numbers1.txt, numbers2.txt and numbers3.txt where each of them provide two different output (output1.txt, numbers1.jpg), (output2.txt, numbers2.jpg) and (output3.txt, numbers3.jpg) respectively.

Deliverables

After writing all the required code use your Quick3wayBM_V.java and run it on numbers.txt and save the output as numbers.jpg. After that use the submission guidelines below to submit the following files for this part of the assignment:

- Quick3wayBM.java (which prints text output ONLY to the command line)
- Quick3wayBM_V.java (which provide the visualization only).
- numbers.jpg (the visualization steps of the numbers.txt file)
- readme.txt (if needed).
- stdlib.jar

Part 2. Sorting Application - Rhyming Words

Overview

For your poetry class, you would like to tabulate a list of rhyming words. For example, if you had the following words (electrics, ethnic, clinic, coughed, laughed, metrics), your program should produce the following output

```
[ cli|nic, eth|nic ]
[ elec|trics, me|trics ]
[ co|ughed, la|ughed ]
```

Write a program Rhyming.java that reads in a sequence of words from standard input and prints them out in the order specified above. Specifically, we define here rhyming words as a group of words sharing sharing a rhyme, and we define that rhyme as being the longest possible suffix common to all words in the group. For instance, in the example above, we do not consider “ic”, “ics” or “ed” to be rhymes because they are suffixes of longer common suffixes.

The output order is defined as follows.
1) Words are listed in increasing order of suffix length
2) Among suffixes of same length, words are listed in alphabetical suffix order
3) Words with common suffix (rhyming words) are listed in alphabetical order

Looking back at the previous example: the suffixes are **nic** (length=3), **trics**(length=5), **ughed** (length=5). They appear in that order as a result of rule 1) (length 3 before length 5) and rule 2) (**trics** comes before **ughed** alphabetically). Finally, each set of rhyming words are listed alphabetically (rule 3). In addition, groups of rhyming words are separated by a newline.

Note that the same word can rhyme with different groups of words: For instance: friendly, lucky, murky, rapidly would lead to following output:

```
[ friendl|y, luck|y, murk|y, rapidl|y ]
[ luc|ky, mur|ky ]
[ frien|dly, rapi|dly ]
```

Again, while “y” is not a rhyme for “lucky” and “murky” (it is a suffix of the actual rhyme “ky”) it is a rhyme for the group “friendly”, “lucky”, “murky”, and “rapidly”, as the longest common suffix.

Input / Output

Your program should read the input from the standard command line. It should process the input using the following command line:

```
% java -cp .:stdlib Rhyming.java < list1.txt
```

The file contains a list of strings, one string per line. Your program should output to standard output all the words sorted as the example stated above. To facilitate the visual assessment of your result, start each new line (corresponding to a series of rhyming words in alphabetical order) with an opening square bracket followed by a space character and, symmetrically, terminate each new line with a space character and closing square bracket. In addition, each word should contain a pipe symbol (‘|’) that indicates the beginning of the identified suffix. Again, refer to the examples above and to the solution files provided below.

Testing with an example

Three test files are available with corresponding solution. Use them to verify that your code complies with all the guidelines listed above: list1.txt, list2.txt and list3.txt and loutput1.txt, loutput2.txt and loutput3.txt respectively.

Deliverables

Use the submission guidelines below to submit the following files for this part of the assignment:

- Rhyming.java
- readme.txt (if needed)
- stdlib.jar

Submission

Submit your solution on or before October 18, 2012. turnin will be used to submit this assignment. The submission procedure is the same as for the first project. Inside your working directory for this project on data (e.g., ~/cs251/project3), create a folder in which you will include all source code used and libraries needed to compile and run your code. DO NOT use absolute paths in your files since they will become invalid once submitted. Optionally, you can include a README file to let us know about any known issues with your code (like errors, special conditions, etc).
After logging into data.cs.purdue.edu, please follow these steps to submit your assignment:

- Enter the working directory for this project

```
% cd ~/cs251/project3
```

- Make a directory named <your_first_name>_<your_last_name> and copy all the files needed to compile and run your code there.
- While still in the working directory of your project (e.g., ~/cs251/project3) execute the following turnin command

```
% turnin -c cs251 -p project3 <your_first_name>_<your_last_name>
```

Keep in mind that old submissions are overwritten with new ones whenever you execute this command. You can verify the contents of your submission by executing the following command:

```
% turnin -v -c cs251 -p project3
```

Do not forget the -v flag here, as otherwise your submission would be replaced with an empty one.