CS 337: Algorithms: Design & Practice

Lab#3: Quicksort *vs.* Hybrid Quicksort *vs.* Library sort()

NOTE: All work should be entirely your own. You are not allowed to work with another person on any aspect of this Lab.

In this lab we will set up an experiment to measure and compare the performance of different implementations of the Quicksort algorithm: Jon Bentley's efficient version versus Hybrid Quicksort (Bentley's version of Quicksort+Insertion Sort). We will also see how our implementations compare to the library **`sort()`** function in the programming language of your choice (C, Go, Java, Python).

We will record execution times for sorting an array of integers of size, N ranging from 5,000,000 to 50,000,000 in steps of 5,000,000. This will give us ten data points to plot for each algorithm. We will experiment the sorting algorithm for average case performance, for non-repeating elements. Our first task, then, is to write a program that does the following:

```
Initialize an integer array of N elements so that A[i] = i.
```

Next, shuffle the array.

You can use the **Fisher-Yates Shuffle** algorithm:

```
for i from 0 to n−2 do
     j ← random integer such that i ≤ j < n
     exchange a[i] and a[j]
```

Task#1: Write a program that implements the above. Test it by printing out the results of creating and shuffling an array of 20 elements. (Aside, why use this algorithm rather than just draw that many random numbers as you did last week?)

Task#2: Bentley's Quicksort Algorithm

Jon Bentley[1] describes the following algorithm for efficiently implementing the Quicksort algorithm:

Given: A an array

left: an index in the array (left>=0) and left < len(A)

right: a second index in the array, right is usually greater than left.

Return: nothing. Execution of the the algorithm changes the order of items in A

```
quicksort(A, left, right) : Sort an array, A from index, left
                            up to index, right.
   We will use two variables: i, m
   if (left >= right) return

   // pivot is a random element in A[l]..A[u]
   swap(A[left], A[randomInt(left, right)])
   m = left
```

```
   for i from left+1 to i <= right do
      if (A[i] < A[left])
         swap(A[++m], A[i])

   swap(A[left], A[m])   // A is now partitioned at A[m]
   quicksort(A, left, m-1)
   quicksort(A, m+1, right)
```

Implement the above algorithm and test to ensure it results in a sorted array of size N=20 before proceeding.

Task#3: Insertion Sort

Implement the Insertion Sort algorithm exactly as in Cormen (pg 37). Again, as in Task#2, ensure that the algorithm is sorting correctly before proceeding.

Task#4: Hybrid Quicksort

Write a hybrid sort function using the following algorithm:
Given: as for quicksort along with

cutoff: a small positive integer typically between 10 and 100.

```
hybridSort(A, left, right, cutoff)
   quicksort(A, left, right, cutoff)
   insertionSort(A, left, right)
```

Use the following modified quicksort() algorithm:

```
quicksort(A, left, right, cutoff): Sort an array, A from
                           index, l up to index, u.
   We will use two variables: i, m
   if (right – left < cutoff) return

   // pivot is a random element in A[l]..A[u]
   swap(A[left], A[randomInt(left, right)])
   m = left
   for i from left+1 to i <= right do
      if (A[i] < A[left])
         swap(A[++m], A[i])

   swap(A[left], A[m])   // A is now partitioned at A[m]
   quicksort(A, left, m-1)
   quicksort(A, m+1, right)
```

Note that this version of quicksort is exactly equivalent to the previous when cutoff==1. It is normally a good idea to experiment a little to determine the cutoff value to determine the sweet spot. Or, you can just use cutoff=25. Once again, test and confirm that the array is being sorted before proceeding (make sure you use N > 25, say N = 100).

Task#5: What about library `sort()`?

Learn how to use the library **`sort()`** function provided in the language of your choice. Can you determine, from the documentation, what kind of sorting algorithm it employs? Just as you

tested the two versions of Quicksort above, write a program to use the library **`sort()`** to sort. Make sure it is correct.

NOTE: All work should be entirely your own. You are not allowed to work with another person on any aspect of this Lab.

Task#6: Performance Measurement

Now that you have two versions of Quicksort implemented correctly, you can time and measure the runtime for sorting arrays of sizes N=5,000,000 to 50,000,000 in increments of 5,000,000. For each N, try and run it several times to take a consistent runtime representative of that size. Why? Additionally, also obtain the timings for sorting using the library **`sort()`** function. You can use the table below to record your timings for each algorithm. Or, save the outputs in a file to load into Excel for plotting purposes.

N	Bentley's Quicksort	Hybrid Quicksort	Library sort()
5 million			
10 million			
15 million			
20 million			
25 million			
30 million			
35 million			
40 million			
45 million			
50 million			

Task#7: Visualization and Analysis

Plot the data in the table to visually see how each sort function compares.

Extra Credit [Optional]

Below is the dual pivot Quicksort algorithm by Vladimir Yaroslavskiy (Actually this is a slight simplification of that algorithm, but it still works well). Implement it and include its timing in the plot above. (Hybrid dual pivot is consistently a little slower that hybrid qSort on my Mac, but consistently a little quicker on a unix box)

What to hand in:

1. A printout of the plot from Task#7.
2. A printout of the table from Task#6 showing all the timings.
3. A **technical essay** that summarizes the observations you can make about the comparative performance of the sorting algorithms. Include as an appendix the implementations of each of your sort algorithms. Note that items 1 and 2 could be included as figures within your text or as appendices.

Here is the dual pivot algorithm (somewhat adapted and simplified from Yaroslavskiy). You can find Yaroslavskiy's Java code on the web. Use this algorithm instead.

```
Given:
     a: an array
     left: an index in the array
     right: an index in the array (these are exactly as in
qSort)
     cutoff: a small positive integer (1 for standard or 20 for
hybrid, dual-pivot qSort)
Return:
     nothing, the array a is modified in place.
```

1. if ((right-left) < cutoff), return.
2. if a[right] < a[left] swap right and left
3. let P1=a[left] and P2=a[right].
4. let L=left+1, G=right-1, K=L
5. Imagine that the array has 4 parts
   - part I with indices from left+1 to L-1. Contains which are less than P1 (initially empty),
   - part II with indices from L to K-1. Contains elements which are greater or equal to P1 and less or equal to P2 (initially empty),
   - part III with indices from G+1 to right-1. Contains elements greater than P2 (initially empty),
   - part IV with indices from K to G. Contains those elements that have not yet been placed in parts I, II or III. (initially contains everything other than P and P2. Empty at the conclusion of the step 6 loop).
6. While K ≤ G.
   1. The next element a[K] from the part IV is compared with two pivots P1 and P2, and placed to the corresponding part I, II, or III.

2. The pointers L, K, and G are changed in the corresponding directions in accordance with the placement.

7. The pivot element P1 is swapped with the last element from part I, the pivot element P2 is swapped with the first element from part III.

8. Recur on each of parts I, II and III.

[1] *The Most Beautiful Code I never Wrote*, Jon Bentley. In *Beauitful Code: Leading Programmers Explain How They Think*, Edited by Andy Oram & Greg Wilson, O'Reilly Publishers, 2007.