Stephen Oliver

Dr. Jie Jiu

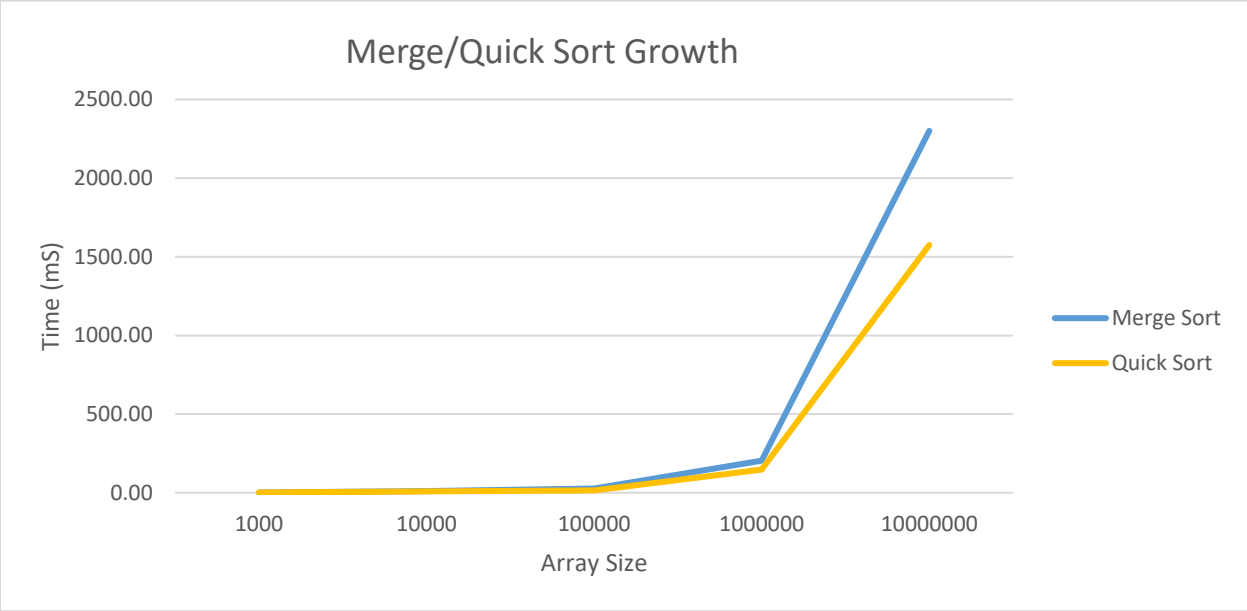
Algorithms

28 January 2017

Lab 1 Overview and Analysis

As discussed in class and assigned reading, both the quick-sort and merge-sort have the same complexity. The complexity of quick-sort and merge-sort is O(n log n); however, merge-sort seems to take more time once n become substantially large. Further adding to the confusion, quick-sort, unlike merge-sort, has a worst-case of $O(n^2)$. Merge-sort seems to have one great weakness that quick-sort lacks; merge-sort requires a whole separate array to use as temporary storage, but quick-sort only requires one array utilizing pointers to specific indexes. Merge-sort should, however, prove more stable on even larger data-sets (i.e. greater that the data-set given to us at 10000000 positive integers). This is due to merge-sort not having a disclosed worst-case like quick-sort does. In the end, it may benefit the merge-sort algorithm to be to have its division limited to a number of elements that can be quickly sorted by another stable sorting algorithm such as insertion-sort.

Run	Array Size	MergeSort (mS)	Quick Sort (mS)
1	1000	2.00	1.00
1	10000	7.00	8.00
1	100000	24.00	17.00
1	1000000	210.00	143.00
1	10000000	2374.00	1583.00
2	1000	2.00	2.00
2	10000	12.00	5.00
2	100000	22.00	17.00
2	1000000	197.00	143.00
2	10000000	2242.00	1595.00
3	1000	1.00	2.00
3	10000	13.00	14.00
3	100000	34.00	11.00
3	1000000	203.00	153.00
3	10000000	2286.00	1547.00
AVG	1000	1.67	1.67
AVG	10000	10.67	9.00
AVG	100000	26.67	15.00
AVG	1000000	203.33	146.33
AVG	10000000	2300.67	1575.00



CODE SEGMENTS

STEP 2:

```
/*
 * Sort the elements between the startIndex and endIndex using Merge Sort.
 * Code altered from:
 *http://www.sanfoundry.com/java-program-implement-merge-sort/
 * @param array The given array to be sorted
 * @param startIndex Index to start sorting at
 * @param endIndex Index to end sorting at
 */
public void auxMergeSort (int[] array, int startIndex, int endIndex)
{
	int n = endIndex - startIndex;
	if (n <= 1)
	{
		return;
	}
	int midIndex = startIndex + n/2;
	auxMergeSort (array, startIndex, midIndex);
	auxMergeSort (array, midIndex, endIndex);
	merge(array, n, startIndex, midIndex, endIndex);
}
private void merge (int[] array, int n, int startIndex, int midIndex,
				int endIndex)
{
	int[] temp = new int[n];
	int i = startIndex;
	int j = midIndex;
	for (int k = 0; k < n; k++)
	{
		if (i == midIndex)
		{
			temp[k] = array[j++];
		}
		else if (j == endIndex)
		{
			temp[k] = array[i++];
		}
		else if (array[j] < array[i])
		{
			temp[k] = array[j++];
		}
		else
		{
			temp[k] = array[i++];
		}
	}
	for (int k = 0; k < n; k++)
	{
		array[startIndex + k] = temp[k];
	}
}
```

STEP 3:

```
/*
 * Sort the elements between the startIndex and endIndex using Quick Sort with
 * the pivot to be the average of the values at startIndex, endIndex, and the
 * middle element between startIndex and endIndex.
 * @param array The given array to be sorted
 * @param startIndex Index to start sorting at
 * @param endIndex Index to end sorting at
 */
public void auxQuickSort (int[] array, int startIndex, int endIndex)
{
	if (startIndex < endIndex)
	{
		int q = partition(array, startIndex, endIndex);
		auxQuickSort(array, startIndex, q-1);
		auxQuickSort(array, q+1, endIndex);
	}
}
private int partition (int[] array, int startIndex, int endIndex)
{
	int mid = (startIndex + endIndex)/2;
	int pivot = (startIndex + mid + endIndex)/3;
	exchange(array, pivot, endIndex);
	int x = array[endIndex];
	int i = startIndex - 1;
	for (int j = startIndex; j <= (endIndex -1); j++)
	{
		if (array[j] <= x)
		{
			i++;
			exchange(array, i, j);
		}
	}
	exchange(array, (i+1), endIndex);
	return i + 1;
}
public void exchange(int[] array, int i, int j)
{
	int temp = array[j];
	array[j] = array[i];
	array[i] = temp;
}
```

STEP 4:

```
/*
 * Check if a given array is sorted in increasing order. (recursive)
 * STACKOVERFLOW on 100,000+ sized arrays (fix if time allows).
 * @param array The given array to check.
 * @return true if and only if the array is sorted in increasing order.
 */
public boolean flgIsSorted (int[] array)
{
	int n = array.length;
	if (n <= 5000)
	{
		return flgIsSortedRecursion(array, n);
	}
	else
	{
		if (flgIsSortedRecursion(Arrays.copyOfRange(array, 0, 5000),
			5000) == true)
		{
			return flgIsSortedLooper(Arrays.copyOfRange(array, 4999,
				n));
		}
		return false;
	}
}
private boolean flgIsSortedRecursion (int[] array, int n)
{
	if (n == 1)
	{
		return true;
	}
	else if (array[n-2] > array[n-1])
	{
		return false;
	}
	return flgIsSortedRecursion (array, n-1);
}
private boolean flgIsSortedLooper(int[] array)
{
	int prevNum = 0;
	for (int num : array)
	{
		if (num < prevNum)
		{
			return false;
		}
		prevNum = num;
	}
	return true;
}
```