CS1020E: Data Structures and Algorithms I

Tutorial 9 – Sorting

8 April 2016

# 1. Choosing Between Sorting Algorithms

In this question, we will only consider the 4 sorting algorithms: *Insertion Sort*, *Quick Sort*, *Merge Sort*, and *Radix Sort*. Given the 4 algorithms above, consider the following scenarios and choose the **fastest** sorting method to be used for each scenario.

(a) You are making a database of the *weight* of the students in NUS for your CCA. However, because of the budget cut, you are facing a problem in the amount of *memory* available for your computer. The memory available is just barely enough to put the entire values in the memory. You should be able to sort the database based on weight.

(b) Since your success in creating the database for your CCA, you are hired as an intern in SoC. The database that you are creating now is a student database. It should contains the *name* and *age*. The sorting requirement is that you should sort the student firstly by *name*, then by *age*. That means all the students with the same age will be displayed in a sorted order.

(c) After finishing intern in SoC, you are invited to be a guest lecturer on sorting class. You just had final exam for the module, and now you just finished marking the papers *randomly*. You want to display the *result* to your students in a sorted order.

(d) Little did you know that you were already marking the *grades* in an *almost sorted order*. However, being a very peculiar skeptics, you still want to sort the result.

# 2. QuickSort with tri-partitioning

In standard Quicksort, we partition an array A[1..n] about a pivot element x. Prof. Triple suggests that we might be able to do better if we use two pivot elements, instead of just one.

a) Suppose that you choose two first elements x and y (such that x<=y) to partition the array A[1..n] into 3 partitions: the left partition contains all element <= x, the middle partition contains all elements >x but <=y, and the right partition contains all elements that >y. Design and implement an algorithm to perform this tri-partitioning with two pivots.

pair<int, int> tripartition( int A[], int low, int high ) {

int x=A[low];

int y=A[low+1];

//swap x and y if needed

//Do tri-partitioning on elements A[low .. high-1] so that

//each element in A[low .. px] <= x;

//x < each element in A[px+1 .. py] <= y;

//y < each element in A[py+1 .. high-1]

return make\_pair(px, py);

}

What is the running time and space complexity?

b) Implement quick sort algorithm with tri-partition procedure in part a.

c) What is worst case and best case running time of your algorithm in part b.

# 3. Pre-processing

Often, we sort data not because we need the final result to be sorted, but because it makes our algorithm more efficient. For each of the following problems, implement two solutions, one without sorting and one with sorting, and analyse the time-complexity for each. You can use std::sort(), which has an average case performance N log(N).

(a) You are given an input array of random integers. Print out all the integers that appear in the array at least k-times.

(b) You are given two input arrays of random integers without duplicates, i.e. within each array, the values are unique. Print out all the integers that appear in both arrays, i.e. the intersection of the two sets.