CS1020E: Data Structures and Algorithms I

Tutorial 9 – Sorting

(Week 11, starting 24 October 2016)

1. *Divide and Conquer!*

Explore the differences between Quick Sort and Merge Sort. While there are a number of different implementations available, use the version of Quick Sort, partition, and Merge Sort discussed in lectures. You are given this array as sample input. Do note that there are two different elements with key value 8.

5	9	12	7	**8**	51	31	8

(a) One advantageous property of quick sort is that it is __________, while merge sort is __________.

Answer

Quick Sort: In place; Merge Sort: Stable

(b) Display the contents of the array after each `partition()` call completes. Mark the pivot. What do you notice about the two elements with the same key value, and what causes this to happen?

Answer

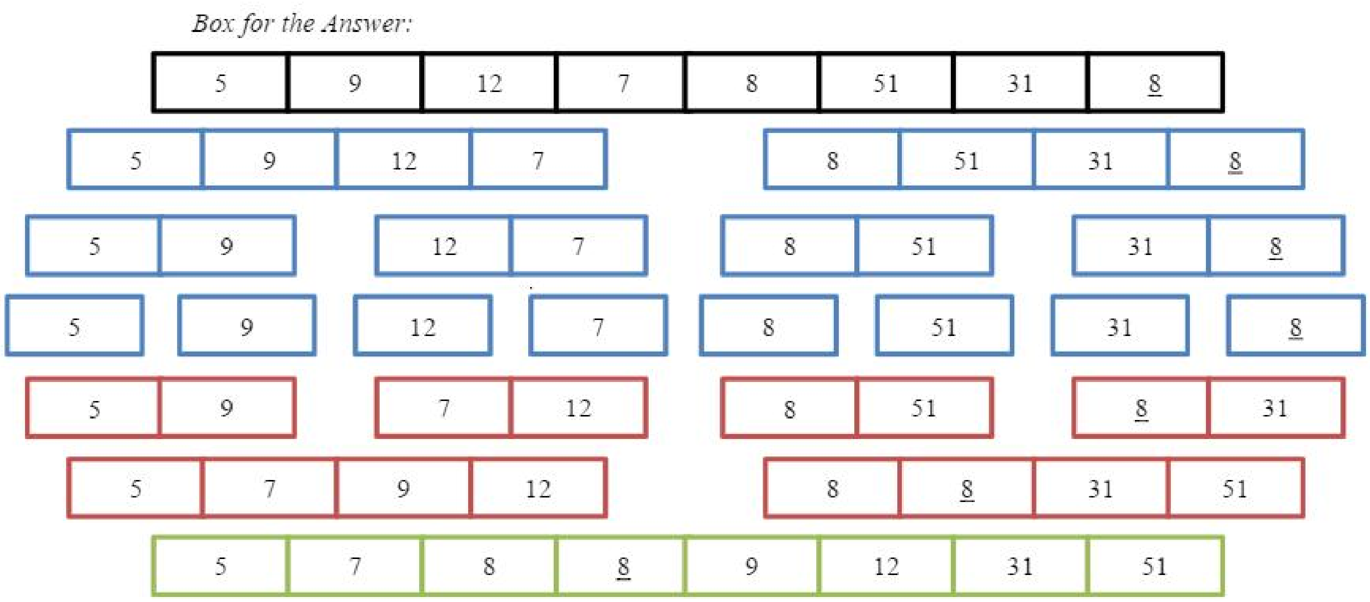
1st Step	5	9	12	7	**8**	51	31	8
2nd Step	5	8	7	**8**	9	51	31	12
3rd Step	5	7	8	**8**	9	51	31	12
4th Step	5	7	8	**8**	9	12	31	51
5th Step	5	7	8	**8**	9	12	31	51
6th Step								

They change their orderings relative to one another, i.e. sort is unstable. Two causes:

- Last value of the left partition finally swaps place with pivot, as in this example
- One element is swapped to the right, over another element of the same value, e.g. [5, 2, **8**, 8, 3]
- Use https://visualgo.net/sorting?create=5,9,12,7,8,50,31,8&mode=Quick to double check (PS: VisuAlgo's sorting vertical bars are limited to 50, so we change 51 to 50)

(c) Using the same array, trace the execution of Merge Sort. Which boxes represent the temporary arrays, and which boxes are just logical (contents of the original array)?

Answer



The **blue** boxes show the subsequent recursive calls to `mergesort()` after the first, so they are **logical**.

The **red** boxes show the contents of each **temporary array** made by a call to `merge()`. These values are then **copied back** to the original array before each call ends.

The **green** box shows the contents of the **final temporary array** created. The values in this array are then **copied back** to the original array before the last `merge()` (or the first `mergesort()`) call ends.

Use https://visualgo.net/sorting?create=5,9,12,7,8,50,31,8&mode=Merge to view the animation of this process (the VisuAlgo animation is showing the bottom half of the merging process, but excluding the top half of the trivial divide process). Again, we replace 51 with 50 in VisuAlgo.

2. Choice of Sorting Algorithm

In this question, consider only 4 sorting algorithms: Insertion Sort, Quick Sort, Merge Sort, and Radix Sort. Choose the fastest sorting method that is suitable for each scenario.

(a) You are compiling a list of students (ID, weight) in Singapore, for your CCA. However, due to budget cut, you are facing a problem in the amount of memory available for your computer. After loading all students in memory, the extra memory available can only hold up to 20% of the total students you have! Which sorting method should be used to sort all students based on weight (no fixed precision)?

(b) After your success in creating the list for your CCA, you are hired as an intern in NUS to manage a student database. There are student records, already sorted by name. However, we want a list of students first ordered by age. For all students with the same age, we want them to be ordered by name. In other words, we need to preserve the ordering by name as we sort the data by age.

(c) After finishing internship in NUS, you are invited to be an instructor for CS1010E. You have just finished marking the final exam papers randomly. You want to determine your students' grades, so you need to sort the students in order of marks *(yes, that Bell Curve system).* As there are many CA components, the marks have no fixed precision.

(d) Before you used the sorting method in (c), you realize the marks are already in *near* sorted order. However, just to be very sure that you did not cut and paste a student record in the wrong order, you still want to sort the result.

Answer

(a) Quick Sort. Due to **memory** constraint, you will need an **in-place** sorting algorithm. Hence, a sorting algorithm that is both in-place and **works for floating point** is Quick Sort. Do note that: The system requires some extra space on the call stack, due to the recursive implementation of Quick Sort (and similarly for Merge Sort), although we say that Quick Sort is in-place.

(b) Radix Sort. The requirements call for a **stable** sorting algorithm, so that the **ordering by name is not lost**. Since memory is not an issue, Radix Sort can be used. Radix Sort has a lower time complexity than comparison based sorts here, O(dn) where d = 2, vs O(n log n) for Merge Sort.

(c) Quick Sort. Being a **comparison-based** sort, Quick Sort is able to sort **floating point** numbers, unlike Radix Sort. Quick Sort is also a good choice because the grades are **randomly distributed**, resulting in O(n log n) average-case time.

Comparing Quick Sort with Merge Sort here, Quick sort is **in-place**, and *may* run faster[1].

(d) Insertion sort. Insertion sort has an O(n) best-case time, which occurs when elements are already in **almost sorted order**. Suppose there are 10 students that have swapped place with the next student. We will then only make 20 extra key comparisons and 10 shifts. Hence, we get O(n) time.

[1] Extra: As long as we don't hit many cases where one partition size is near 0, Quick Sort often runs faster than Merge Sort. It *may* be able to take advantage of caching better, due to the fact that consecutive elements read are located near to each other. See accepted answer at http://stackoverflow.com/questions/70402/why-is-quicksort-better-than-mergesort

3. Comparison vs Non-Comparison Sort

`arr` is an **unsorted** integer array of length N (very long), containing only non-negative values.

(a) Print all integers that appear at least k times in the array. **Requirement:** Time complexity of O(N log N)

(b) Now, the array contains only integers within the range [0, 1000]. Print the integer that appears the most number of times. If more than one integer appears the same number of times, print the first to reach that number of occurrences. The array has to be sorted in ascending order afterwards. **Requirement:** Time complexity of **O(N)**…

Yikes! =O [Hint: Count…]

Answer

(a)

Sort the array using an O(N log N) algorithm, e.g. merge sort. Sequentially, check if an element also appears k-1 elements ahead. If so, print the element, and save it to avoid duplicated printing. When this happens, we can save on iterations by jumping k elements ahead. Take care not to check past the end of the array.

```
// pre-condition: k (minRecurrences) is positive
void printRecurringElm(int arr[], int size, int minRecurrences) {
   sort(arr, arr + size); // O(N log N)
   int lastPrinted = arr[0] - 1, currIdx = 0;
   while (currIdx <= size - minRecurrences) { // O(N) worst case
      if (arr[currIdx] == arr[currIdx + minRecurrences -1]) { // k times
         if (arr[currIdx] != lastPrinted) { // skip if already printed
            cout << arr[currIdx] << " ";
            lastPrinted = arr[currIdx];
         }
         currIdx += minRecurrences;
      } else {
         currIdx++;
      }
   }
}
```

(b)
Do NOT use a comparison-based sort, as the worst-case time complexity would be at least O(N log N). Maintain an array of occurrences (**count**) of each value. To avoid iterating through that entire array as we read each number, record the value that appears the most number of times and check whether it needs to be replaced. (Similar to finding the maximum value in an array)

To sort the array afterwards, just overwrite each element in the **input** array with the number of times each number appeared in count, traversing count in ascending order.

```
// pre-cond: Every input[i] is in range [0, uppLimit], array size not 0
void countingSort(int input[], int size, int uppLimit) {
   int count [uppLimit + 1];
   for (int i = 0; i <= uppLimit; i++)
      count[i] = 0;

   int mostOften = 0, inputIdx = 0; // assume input[0] occurs most
   while (inputIdx < size) { // count, O(N)
      count[input[inputIdx]]++;
      if (count[input[inputIdx]] > count[mostOften])
         mostOften = input[inputIdx];
      inputIdx++;
   }

   cout << mostOften << " appears " <<
      count[mostOften] << " times." << endl;

   int currValue = 0; // now "sort" the original array
   inputIdx = 0;
   while (currValue <= uppLimit) { // O(N + range), NOT O(N*range)
      while (count[currValue]-- > 0)
         input[inputIdx++] = currValue;
      currValue++;
   }
}
```

Good practice: Declare variables in as narrow a scope as possible, and just before where you want to use them. This helps others identify where each variable is used, and narrows the portion of code where the variable could be used by mistake.

O(1)

more weeks to go!!!