CS2040 Tutorial 5 Suggested Solution

Week 7, starting 26 Sep 2022

Q1 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, but 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 just once, 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. As there are many CA components, the marks have no fixed precision

(d) For the same data in part (c), you realize the marks already seem to be in 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]

[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

(d) Insertion sort. Insertion sort has an O(n) best-case time, which occurs when elements are already in **almost sorted order**. Suppose half of the students have swapped place with the next student, i.e. everyone is in the wrong place but they are almost sorted. We will then make only n/2 extra key comparisons and n/2 shifts. Hence, we still get O(n) time

Q2 Missing Family Members

The Addams family had just gone on a fishing trip and taken a photo of all of their family members standing side by side in a long line. Each member of the family will wear a distinct shirt labelled 1..**N**, **N** being the number of people in the family and can be large. Unfortunately, the photo was edited with some family members being removed (but no one has their positions changed/swapped)

If the original sequence of shirts in the photograph is always the first permutation of 1..**N** that contains the sequence of all family members in the edited photo, find an **efficient** way to reproduce this original sequence

For example, suppose **N**=6 and [1, 4, 6, 3] was observed in the edited photo. The original sequence is [**1**, 2, **4**, 5, **6**, **3**] as that is the first permutation of {1,2,3,4,5,6} that contains [1, 4, 6, 3] in that order

```
int[] findOrigSeq(int N, ArrayList<Integer> editedSeq)
```

Answer

Let the first permutation of 1..**N** that contains the sequence of all family members in it be [p1, p2, p3, … pN]. p1 will be as small as possible (as it is the most significant digit), followed by p2 being as small as possible after filling in p1, … p_k as small as possible after filling in p_{k-1}

Therefore, we just need to merge (yes the merge used within mergeSort) the missing people in sorted order with the sequence of people in the photograph. This ensures the smallest possible number will always be brought down first, without disrupting the sequence of people in the photograph

No sorting is required here. Using comparison-based sort results in a worse time complexity

```
int[] findOrigSeq(int N, ArrayList<Integer> editedSeq) {
   boolean[] present = new boolean[N+1];
   for (int edited : editedSeq)
      present[edited] = true;
   ArrayList<Integer> absent = new ArrayList<>();
   for (int idx=1; idx <= N; idx++)
      if (!present[idx])
         absent.add(idx);
   return merge(absent, editedSeq);
}
```

```
int[] merge(ArrayList<Integer> left, ArrayList<Integer> right) {
   int[] result = new int[left.size() + right.size()];
   int resultIdx = 0, leftIdx = 0, rightIdx = 0;
   while (leftIdx < left.size() && rightIdx < right.size()) {
      if (left.get(leftIdx) <= right.get(rightIdx))
         result[resultIdx++] = left.get(leftIdx++);
      else
         result[resultIdx++] = right.get(rightIdx++);
   }
   while (leftIdx < left.size())
         result[resultIdx++] = left.get(leftIdx++);
   while (rightIdx < right.size())
         result[resultIdx++] = right.get(rightIdx++);
   return result;
}
```

Q3 Waiting for Doctor

You work at a 'high-tech' clinic with only ONE doctor. Before the clinic opens for the day, **ALL N** patients will be waiting outside the clinic, and have made an appointment to visit the doctor

As the appointment system keeps track of a patient's history and medical condition, it is able to assign for each patient **i**, the time $\mathbf{t_i}$ required to serve patient **i**. As the doctor can only see 1 patient at any time, how do you minimize the total waiting time of all patients?

(a) Implement an **efficient algorithm**, returning the total waiting time of all patients for the day. The waiting time of a patient refers to the time elapsed from the start of the day to the point the patient starts to visit the doctor

(b) If each serving time $\mathbf{t_i}$ is an integer in [1, 3**N**), can we do **better** than O(**N** log **N**) time?

```
int computeMinWait(List<Integer> servingTimes)
```

Answer

(a) There is a **greedy solution** to this problem. We can rearrange the input such that at every step taken to solve the problem, taking the 'obvious best choice' at that one step leads to the best solution to the entire problem. This algorithm or selection policy is called Shortest Job First (SJF).

Suppose we queue patients according to shortest $\mathbf{t_i}$ first. For any two patients **a** and **b** in which $\mathbf{t_a} < \mathbf{t_b}$, **a** will visit the doctor before **b**, according to our selection policy. Swapping the places of **a** and **b** will just lead to a net increase in waiting time of $\mathbf{t_b} - \mathbf{t_a}$ for at least one person. This proves that the SJF policy leads to an optimal solution, i.e. it minimizes the waiting time of all patients.

Brute-forced SJF: O($\mathbf{N}^2$)
Merge sort patients by waiting time ASC then SJF: O(**N** log **N**)

```
int computeMinWait(List<Integer> servingTimes) {
   ArrayList<Integer> copy = new ArrayList<Integer>(servingTimes);
   Collections.sort(copy); // avoid mutating the input
   int wait = 0, ppl = servingTimes.size();
   for (int idx = 0; idx < ppl; idx++)
      wait += (ppl-1-idx) * copy.get(idx); // affects every1 after idx
   return wait;
}
```

Alternatively, the current time can be added to the total waiting time and maintained, so that there is no need to transform the problem by looking ahead

(b) In radix sort, for each of the **d** digits starting from the least significant digit: **b** buckets are created, the **N** elements are thrown into **b** buckets in a stable manner, then the elements are pulled out from the **b** buckets in sequence

To be very precise, the time complexity of radix sort is O(**d**(**N**+**b**)) where **N** is the number of elements, **b** the number of buckets and **d** the number of digits. Usually **b** is introduced as a small constant so not much attention is paid to it

Where is the bottleneck and how can we get rid of it? If **b**=10, then **d**$\approx \log_{10}(3\mathbf{N})$ so we will get O(**N** log **N**) time from radix sort. But if we change the idea of a digit to be somewhere between **N** and 3**N**, i.e. **N** $\leq$ **d** $\leq$ 3**N**, then we will require only **d**=1 or 2 passes

When **b**=3**N**, there will only be **d**=1 pass, and this algorithm is known as bucket sort

Question 4 (Online Discussion) – Still Waiting for Doctor?

As in Q3, if now each serving time $\mathbf{t_i}$ is an integer in [1, $\mathbf{N}^3$), can we do **better** than O(**N** log **N**) time?