Lecture 9: Sorting II

built on 2019/05/21 at 11:51:50

Last time, we saw a few sorting algorithms, starting with a few "natural" $O(n^2)$-time algorithms such as bubble sort and insertion sort. We also saw how merge sort could improve the running time to $O(n \log n)$ using the familiar pattern of splitting in (roughly) half and recursively solve the problem. This lecture continues on with the theme of sorting.

1 Quick Sort

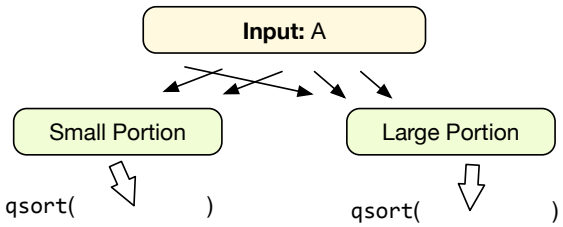
We saw that merge sort achieves $O(n \log n)$ running time by using divide and conquer. Specifically, it divides the problem in roughly half in a natural way—splitting in the middle, recursively solves the two subproblems, and pieces them together. While the divide step is really easy, the combine step is somewhat involved, at least relative to the divide step.

It is reasonable to ask: Can we derive an algorithm with a trivial combine step perhaps at the cost of a more complex divide step?

Say we want a dirt simple combine strategy: If the recursive solutions are X and Y, we want to simply concatenate them. The layout of the algorithm, therefore, looks as follows:

```
def qsort(A):
  if len(A) <= 1: return A
  else:
     # do something to derive
     # say X, Y
     return X + Y
```

But *what strategy could we use to divide up the problem* to satisfy this requirement? Because we wish to simply concatenate the parts, we have imposed certain conditions as to what the parts have to look like, relatively to each other. Everything in X must be less than anything in Y.



1.1 Split At a Pivot Value

It is unclear a priori how to split an input list A into small and large portions while ensuring that the two portions have roughly the same size. We'll start with a simple idea: We'll pick a value called the pivot p, and split the input list A into *three* parts:

- `lt` — items that are strictly less than p;
- `eq` — Items that are exactly p; and
- `gt` — Items that are strictly greater than p.

The `eq` part is there for reasons that will soon be apparent. Can we write code for this quickly?

```
void split(List<Integer> a, Integer p,
           List<Integer> lt, List<Integer> eq, List<Integer> gt) {
   for (Integer elt : a) {
       int cmp = elt.compareTo(p);
       if (cmp < 0)        { lt.add(elt); }
       else if (cmp == 0)  { eq.add(elt); }
       else                { gt.add(elt); }
```

```
    }
}
```

What's the running time of `split`? $O(\texttt{len}(A))$ because each list-comprehension expression simply goes over the list taking things that match the condition, and we do this 3 times.

The key question that will influence the performance of quick sort is, *how to select the pivots?* For now, let's say we'll use A[0], the first element in the input list as our pivot. Hence, we have our initial version of quick sort:

```
static ArrayList<Integer> qsort(ArrayList<Integer> xs) {
  if (xs.size() <= 1) return xs;
  else {
    ArrayList<Integer> lt = new ArrayList<>(),
                       eq = new ArrayList<>(),
                       gt = new ArrayList<>();

    Integer p = xs.get(0);
    split(xs, p, lt, eq, gt);


    ArrayList<Integer> out = qsort(lt);
    out.addAll(eq);
    out.addAll(qsort(gt));
    return out;
  }
}
```

Exercise: Show that this code is correct.

1.2 How (not) to Select Pivots?

Our initial attempt at writing quick sort was a sucess in that the algorithm correctly sorts a given input list; however, from a performance point of view, it wasn't great. To understand what's going on, let's write a recurrence for the above algorithm:

$$T(0) = T(1) = 1$$
$$T(n) = T(n_{lt}) + T(n_{gt}) + O(n),$$

where $n_{lt} = \texttt{len}(lt)$ and $n_{gt} = \texttt{len}(gt)$.

Q: Can we come up with an example where this `qsort` will not perform well? As in it will deteriorate into something like insertion/selection sort.

It's not too hard to convince ourselves that if the input is already sorted, like $A = [1,2,3,4,5]$. Our current implementation will suffer because each time it divides the list into $lt = [], eq = [A[0]], gt =$ the rest, resulting in $T(n) = T(n-1) + O(n)$, which is an $O(n^2)$ algorithm.

How can we fix this? Use last element? Use the middle element? Use the median of first, middle, and last?

As it stands, it may look like no matter what we deterministically choose, an adversary can force our hands so this algorithm runs slowly. (Well, not quite true as we'll see later). But we want something simple, right?

How about picking an element from that list to be a pivot at random? Yes, we're going to pick one of the elements randomly.

1.3 Randomized Quick Sort

This will be our first algorithm that makes random choices. We'll need help from a random-number generator (RNG), which most languages, including Java and Python, provide. In a way, by making random choices, the

adversary can't know ahead of time what choices of pivots we're using, so it can't construct an example that's truly bad for our code. So we just change how we pick the pivot, though this has to be paired with a random number generator instance (`java.util.Random`).

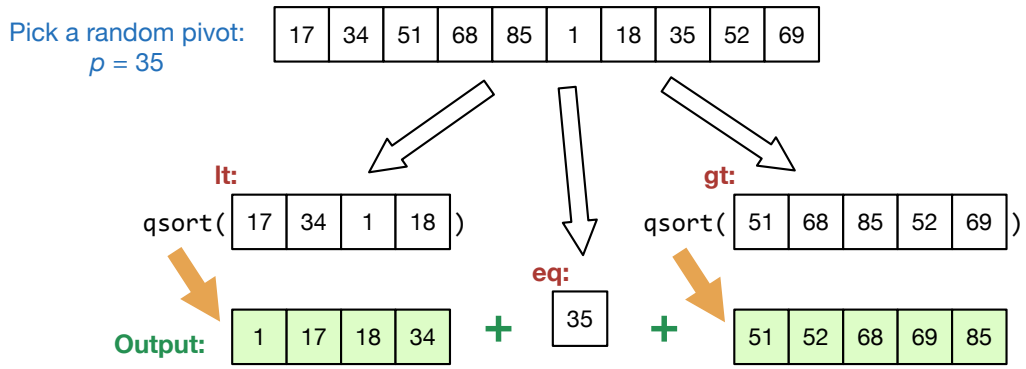
```
static Random RNG = new Random();  // declared "globally"

// how to pick the pivot
Integer p = xs.get(RNG.nextInt(xs.size()));
```

How fast is this algorithm? If we handwave and say the split is balanced—i.e., we get a perfect split $n/2$ and $n/2$ (which we actually don't). Then, the recurrence becomes: $T(n) = 2T(n/2) + O(n)$, which solves to $O(n \log n)$.

In reality, the split is unlikely to be perfect; however, it can't be so uneven, either. This intuition can be formalized, allowing us to prove an $O(n \log n)$-time bound, which holds true in probability.

Before we move on, let's see this in action:



2 Java Built-In Sort

Java, as with most modern languages, has built-in capabilities for sorting an array/collection. I'd like to talk about `Collections.sort`, which sorts a collection, and `Arrays.sort`, which sorts an array.

This means, if you have an array `int[] a`, you can sort it like so:

```
int[] a = {3, 5, 1, 9, 8};
Arrays.sort(a);
```

Or if you have an `ArrayList<Integer>` (which is a collection), you can sort it like this:

```
ArrayList<Integer> a = ...;
Collections.sort(a);
```

2.1 Sorting By a Key Function

In many situations, we wish to sort a list of elements according to some order other than the "natural" order. For example, say we have an array

```
String[] colors = { "violet", "red" , "blue" , "teal" , "green",
   "pink", "magenta" , "orange" }
```

If we just sort it using `Arrays.sort`, we would get the array:

```
 {"blue", "green", "magenta", "orange", "pink", "red", "teal", "violet"}
```

Suppose we're interested in sorting the colors by the length of the color name. How can we do that? In particular, we wish to sort by an alternate attribute:

Real Key	"violet"	"red"	"blue"	"teal"	"green"	"pink"	"magenta"	"orange"
Sort Key	6	3	4	4	5	4	7	6

As it turns out, we could specify how we wish the data elements to be ordered by, by giving the `sort` function an extra parameter.

```
Arrays.sort(colors, (String x, String y) -> x.length() - y.length());
```

This extra parameter is called the Comparator (see Java doc for details). This is to indicate to the sorting function that to compare x and y, compare their lengths instead.

Exercise: Think how you would modify our merge sort (or quick sort) to support sorting by a different key.

3 Discussion and Remarks

Is $O(n \log n)$ the best you can hope for in general? Let me explain this question a bit before I attempt to answer it. Sorting, as we have discussed so far, assumes that you learn whether or not an item a is less than an item b by comparing them. This is known as the *comparison model*. In the comparison model, it can be proved that the best sorting algorithm requires $O(n \log n)$ in general. That is to say, no matter what the algorithm is, there will be an input that causes the algorithm to spend at least $n \log n$ time.

Puzzle: I have n numbers $a_0, a_1, \ldots, a_{n-1}$, and I promise that each a_i is an integer $\in [0, 1000]$. How fast can you sort them? Hint: $O(n)$. Let's try to solve this for a bit. Here's a pointer if you want to learn more: https://en.wikipedia.org/wiki/Counting_sort.

4 Aside: Quick Sort's Probabilistic Recurrence

Here's the recurrence I mentioned during our digression. Say we're working with a list of length n. If we pick the pivot randomly, we have a probability of $1/n$ to pick a pivot that splits the list into i and $n - i + 1$ for any i. Therefore, if $\bar{T}(n)$ is the average running time for quick sort on n elements, then

$$\bar{T}(n) = O(n) + \sum_{i=0}^{n-1} \frac{1}{n} \Big(\bar{T}(i) + \bar{T}(n-i+1) \Big),$$

which solves to $\bar{T}(n) = O(n \log n)$.

Exercise: (for the brave) Solve this recurrence.