Lecture 8: Sorting I

built on 2019/05/15 at 22:42:43

The main theme of this lecture: sorting—i.e., how to arrange a given sequence into an ordered sequence.

As a running example, say we have a list of numbers:

```
[54, 26, 93, 17, 77, 31, 44, 55, 20]
```

and we're interested in producing a list that is a sorted copy of the initial list—as efficiently as possible! The sorted copy of this list is:

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

There are many reasons that we're interested in the sorting problem: it's not only a basic problem in computer science but also a handy technique that you'll use as a subroutine to solve other problems. Moreover, as you'll see, it can be done pretty fast.

Rules of the game. To be concrete, we're given an array of elements that can be compared and we want to order them from small to large using this ordering. In Java, we model this as an array of `Comparable`. Therefore, the algorithm specifiation is

```
void sort(Comparable[] a)
```

1 Bubble Sort

The first algorithm we'll look at will be a simple one. To motivate this algorithm, think about how you can test if a given list is sorted. Perhaps, you'll write a piece of code that goes over the list, which checks if adjacent elements are in the right order (i.e. $a[i] \leqslant a[i+1]$).

If the list is not sorted, you'll find adjacent elements that violate $a[i] \leqslant a[i+1]$. To fix this, we will swap them. Although it is unclear a priori if this will help anything overall, it is intuitively clear that it fixes one violation.

Building on this idea, bubble sort makes passes over the input list. In each pass, it compares adjacent elements and swap them if they are out of order.

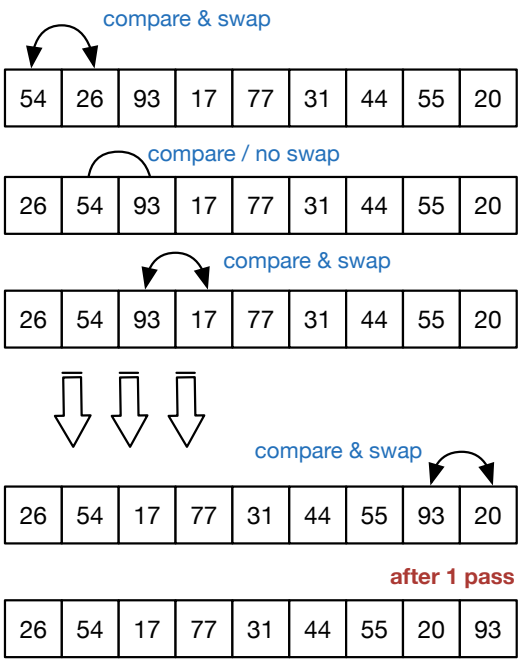
More concretely, we'll repeat these steps until the list is sorted:

> For $i = 0, 1, ..., n-2$: compare and swap $A[i], A[i+1]$ as necessary

We know that the list is sorted once we aren't making any more swaps.

The curious thing is, *why would this algorithm sort the list eventually?* An important observation is that for each pass through the list, the next largest value will be put in the proper spot. In fact, each element floats, or well "bubbles", up to the correct location.

To see how this works in action, the figure to the right illustrates the first pass through our running example:



Turning this idea into code, we'll first write a function `swap` that exchanges a pair of elements:

```
void swp(Object[] a, int i, int j) {
  Object temp=a[i]; a[i]=a[j]; a[j]=temp;
}
```

1.1 Is A Less Than B?

If u and v are `Comparable`, how can we tell whether u is less than v? The answer is simple: if `u.compareTo(v) < 0`. We can even write a function for that:

```
boolean less(Comparable u, Comparable v) {
  return u.compareTo(v) < 0;
}
```

1.2 Let's Write Some Code

Because for every pass that we run, we bring the next largest element to the right spot, we know that after at most $n-1$ passes, we will be done. (Exercise: convince yourself this.) The main bubble sort code is pretty straightforward (left: do it $n-1$ passes, right: stop as soon as sorted).

```
void bubbleSort(Comparable[] a) {
  int n=a.length;
  for (int rep=0;rep<n-1;rep++) {
    for (int i=0;i<n-1;i++)
      if (less(a[i+1], a[i]))
        swp(a,i,i+1);
  }
}
```

```
void bubbleSort(Comparable[] a) {
  int n=a.length;
  for (;;) {
    boolean change=false;
    for (int i=0;i<n-1;i++)
      if (less(a[i+1], a[i])) {
        swp(a,i,i+1);
        change=true;
      }
    if (!change) break;
  }
}
```

Running time: $O(n^2)$ Why?

2 Insertion Sort

Let's look at another implementation that is probably equally intuitive. Remember that problem from last term... where you were asked to write a function that takes a sorted list and an extra element x and returns a new list which adds x to the input list at a location that will keep it sorted. As it turns out, this routine can be used to develop a sorting algorithm known in the literature as insertion sort.

Let's first think about this problem:

> We have an array a of length n. Assume that $a[0], a[1], \ldots, a[i-1]$ is sorted. What to do to make $a[0], a[1], \ldots, a[i-1], \underline{a[i]}$ also sorted?

As an example, let's look at this example:

> 1 4 7 $\underline{2}$

In other words: What we want to do is to push 2 downward until it's in the right position. As we do so, the other numbers will "bubble up." In fact, we could do the following:

```
for (int j=i;j>0 && less(a[j],a[j-1]);j--)
    swp(a, j, j-1);
```

What this code does is precisely take a partially sorted list and "insert" a new element into it while maintaining sortedness. That's why it's called insertion sort.

With this idea, we can write insertion sort as follows:

```
void insertionSort(Comparable a[]) {
  int n=a.length;
  // invariant: a[0]...a[i-1] is sorted
  for (int i=0;i<n;i++) {
    for (int j=i;j>0 && less(a[j],a[j-1]);j--)
      swp(a, j, j-1);
  }
}
```

What's the running time? $O(n^2)$. Why?

2.1 Selection Sort

While the previous algorithm keeps inserting elements into a sorted list one by one, we can think of a different way to sort a list:

- First, find the minimum element; call this e_0
- Then, find the minimum element after excluding e_0; call this e_1
- Then, find the minimum element after excluding e_0 and e_1; call this e_2;
- ...

In code:

```
void selectionSort(Comparable a[]) {
  int n=a.length;
  // invariant: a[0]...a[i-1] is sorted
  for (int i=0;i<n;i++) {
    // goal: mi is the min-elt index excluding a[0],a[1],...,a[i-1]
    int mi=i;
    for (int j=i+1;j<n;j++)
      if (less(a[j],a[mi])) mi=j;
    if (mi!=i) swp(a,mi,i);
  }
}
```

All of the algorithms we have looked at have quadratic running times. Is it possible break that barrier? Could we sort faster than $O(n^2)$ in the general case?

3 Breaking The Quadratic Barrier: Divide and Conquer

Divide and conquer is a highly versatile technique that generally lends itself well to the design of fast algorithms. We have already seen the divide-and-conquer technique many times without knowing its name. This time we'll give the technique a label and look at a few examples.

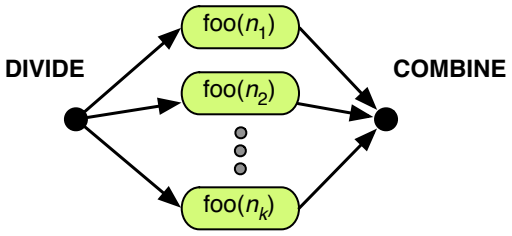
The structure of a divide-and-conquer algorithm follows the structure of a proof by (strong) induction. This makes it easy to show correctness and also to figure out cost bounds. The general structure looks as follows:

— **Base Case:** When the problem is sufficiently small, we return the trivial answer directly or resort to a different, usually simpler, algorithm, which works great on small instances.

— **Inductive Step:** First, the algorithm **divides** the current instance I into parts, commonly referred to as *subproblems*, each smaller than the original problem. Then, it **recurses** on each of the parts to obtain answers for the parts. In proofs, this is where we assume inductively that the answers for these parts are correct, and based on this assumption, it **combines** the answers to produce an answer for the original instance I.

So far, this process is identical to what we have seen as our recipe for designing recursive algorithms. The signature of divide-and-conquer algorithms is that we attempt to make each part a large fraction of the starting instance—not just one smaller or a constant smaller.

Schematically, the process works as follows: To solve a problem `foo` on an instance of size n,



4 Merge Sort

Almost invariably, the main idea for reducing the time complexity so far has been to aggressively reduce the problem size: whereas in essence both insertion sort and bubble sort reduce the problem size by 1, we wish to do better, perhaps cutting the size down in half—like we did for binary search. Following that pattern, we'll attempt to write a recursive algorithm that splits a given list in half.

Like before, there are two questions we need to answer:

(Q1) **How to solve small instances?** As with our previous problems, we measure the size of the input by the length of the input list.

Do we know how to sort an empty list and a list of length 1? This should be easy. Both the empty list and a list of length 1 are already sorted. Duh. Hence, `if len(in_list)<=1: return in_list`.

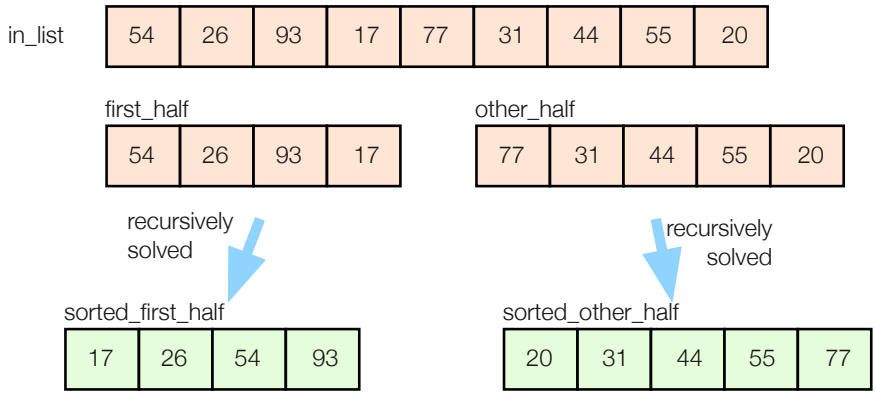
(Q2) **How to tackle an instance in terms of smaller instances?** Say, inside the call where the input has length n, we already know how to sort any input list of length $< n$. How can we use this to our advantage?

We'll try the pattern we have used before. Let's break the input list into two (roughly) equal-sized lists—`first_half` and `other_half`—at midpoint. This is straightforward and we have done it before.

Furthermore, both these lists are shorter than n, in fact, just half of that. Hence, we could call ourselves on both of them, yielding a sorted copy of `first_half` and `other_half`, respectively.

What we would like to do is to combine these two sorted lists into a single one that is sorted.

Pictorially, we have something like this:



To simplify it a little, we'll implement the version of merge sort that operates on integer arrays.

In code, we have:

```
static Integer[] mergeSort(Integer[] a) {
  int n=a.length;
  if (n <= 1) return a;

  Integer[] left = Arrays.copyOfRange(a, 0, n/2);
```

```
    Integer[] right = Arrays.copyOfRange(a, n/2, n);
    return merge(mergeSort(left), mergeSort(right));
  }
```

The crux of the matter is then, *how to merge two sorted lists into one?* The trick is to realize that the two lists we're merging are already sorted, so we can "peel off" the front of either list whichever one is smaller. To implement this, we keep two fingers, one at each point, pointing to the location we haven't peeled off. We compare the elements where the fingers are pointing and peel off the smaller one. Eventually one list will be empty. We simply append the nonempty one to the end output. In code:

```
  static Integer[] merge(Integer[] a, Integer[] b) {
    Integer[] out = new Integer[a.length + b.length];
    int ai=0,bi=0,oi=0;

    while (ai < a.length && bi < b.length) {
      if (less(a[ai], b[bi])) { out[oi++] = a[ai++]; }
      else                    { out[oi++] = b[bi++]; }
    }
    while (ai<a.length) {out[oi++] = a[ai++];}
    while (bi<b.length) {out[oi++] = b[bi++];}

    return out;
  }
```

What's the running time of `merge`? Convince yourself that it is $O(\texttt{len}(X) + \texttt{len}(Y))$.

Therefore, `msort` follows the recurrence $T(0) = T(1) = 1$ and $T(n) = 2T(n/2) + O(n)$, which solves to $O(n \log n)$. This makes sense intuitively also because if we were to draw a tree, we'll get a nice bushy tree of height $\log(n)$ and each level does roughly speaking $O(n)$ work for a total of $O(n \log n)$.

5 Sort/Merge-Inspired Algorithms

Many problems don't look anything like a sorting problem at first glance. Nonetheless, the sheer fact that the instance is sorted and in some cases, a merge-like routine can be of great help to solving them. Here we'll look at two examples.

5.1 Duplicate Removal

In our first example, we are given a list of elements (think about numbers if it's any easier) and we want to produce a list where each element in the original list shows up once (i.e. removing all the extra copies). The output can be in any order. For example, if the input is `[3,7,3,8,8,7,1,4,3]`, one possible output is `[3,7,8,1,4]`.

Let's think about this for a moment. Hint: Things look nicer when sorted :)

One way to solve this problem is to realize the following:

The Crux: Once sorted, all copies of the same element are next to each other.

For instance, on the example input, sorting `[3,7,3,8,8,7,1,4,3]` yields `[1, 3, 3, 3, 4, 7, 7, 8, 8]`, which makes it easy to tease out unique elements.

Turning this idea into code is not difficult:

```
static ArrayList<Integer> mkUnique(ArrayList<Integer> xs) {
    ArrayList<Integer> elements = new ArrayList<>(xs);
    Collections.sort(elements);
    ArrayList<Integer> out = new ArrayList<>();
```

```
    Integer prev=null;
    for (Integer elt : elements) {
        if (!elt.equals(prev)) {
            prev = elt;
            out.add(elt);
        }
    }
    return out;
}
```

5.2 Intersection

We'll now turn our attention to another problem. Let A and B be two lists. We'll assume for now that they individually contain no duplicates, an assumption we can conveniently satisfy using our previous example. We are interested in producing a list which is the intersection of the two lists—that is, an element will be there if it is in both A and B. Again, we don't care about the order in which the elements are listed in the output. As a few examples:

- On input `A = [1, 4, 2]` and `B = [7,1,9,5,2]`, one possible output is `[1, 2]`.
- On input `A = [1, 5, 3]` and `B = [6, 4, 2]`, the output is `[]`.

How to solve this problem quickly? At first glance, it has nothing to do with sorting or merging. But one thing we notice when we wrote the merge routine was that identical elements from different sides get compared, so we're going to take advantage of this observation:

```
// a and b must be sorted ascendingly
static List<Integer> intersectSorted(List<Integer> a, List<Integer> b) {
    List<Integer> common = new ArrayList<>();

    int ix=0, iy=0;
    while (ix < a.size() && iy < b.size()) {
        if (a.get(ix).equals(b.get(iy))) {
            common.add(a.get(ix));
            ix++; iy++;
        } else if (a.get(ix).compareTo(b.get(iy)) < 0) { ix++; }
          else                                         { iy++; }
    }

    return common;
}
```