CS 61B
Spring 2021

ADTs & Sorting

Discussion 12: November 9, 2020

1 All Sorts Of Sorts

Show the steps taken by each sort on the following unordered list:

```
0, 4, 2, 7, 6, 1, 3, 5
```

(a) Insertion sort

```
0 | 4 2 7 6 1 3 5
0 4 | 2 7 6 1 3 5
0 2 4 | 7 6 1 3 5
0 2 4 7 | 6 1 3 5
0 2 4 6 7 | 1 3 5
0 1 2 4 6 7 | 3 5
0 1 2 3 4 6 7 | 5
0 1 2 3 4 5 6 7 |
```

(b) Selection sort

```
0 | 4 2 7 6 1 3 5
0 1 | 2 7 6 4 3 5
0 1 2 | 7 6 4 3 5
0 1 2 3 | 6 4 7 5
0 1 2 3 4 | 6 7 5
0 1 2 3 4 5 | 7 6
0 1 2 3 4 5 6 | 7
0 1 2 3 4 5 6 7 |
```

(c) Merge sort

$\boxed{0\ 4\ 2\ 7\ 6\ 1\ 3\ 5}$

$\boxed{0\ 4\ 2\ 7}\ \boxed{6\ 1\ 3\ 5}$

$\boxed{0\ 4}\ \boxed{2\ 7}\ \boxed{6\ 1}\ \boxed{3\ 5}$

$\boxed{0}\ \boxed{4}\ \boxed{2}\ \boxed{7}\ \boxed{6}\ \boxed{1}\ \boxed{3}\ \boxed{5}$

$\boxed{0\ 4}\ \boxed{2\ 7}\ \boxed{1\ 6}\ \boxed{3\ 5}$

$\boxed{0\ 2\ 4\ 7}\ \boxed{1\ 3\ 5\ 6}$

$\boxed{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7}$

(d) Use heapsort to sort the following array (hint: draw out the heap). Draw out the array at each step:

```
0, 6, 2, 7, 4
```

```
7 6 2 0 4 (turns the array into a valid heap)
6 4 2 0 7 ('delete' 7, then sink 4)
4 0 2 6 7 ('delete' 6, then sink 0)
```

```
2 0 4 6 7 ('delete' 4, then sink 2)
0 2 4 6 7 ('delete' 2)
0 2 4 6 7 ('delete' 0)
```

2 Sorta Interesting, Right?

(a) What does it mean to sort "in place", and why would we want this?

In general, we consider a sorting operation to be done in place if it does not require significant extra space. "Significant extra space" is often defined as linear or greater, with respect to the number of elements we are sorting. For example, if our sorting algorithm requires making a whole new array and copying elements over to it, then it is NOT in place because we had to allocate significant space for this new array. Some algorithms that can be implemented in place are selection sort, insertion sort, and heap sort. Mergesort technically can be implemented in place, but it's rather complex.

Doing operations in place is beneficial because we typically want to use as little memory/computer resources as possible. Though in this class, we focus on time efficiency, space efficiency is important too in the real world!

(b) What does it mean for a sort to be "stable"? Which sorting algorithms that we have seen are stable?

If given a list with equivalent elements (according to whatever metric we're using to compare items), a stable sorting algorithm will leave those elements in the same order as they were originally. Some stable sorts we've seen are insertion sort and merge sort.

(c) Which algorithm would run the fastest on an already sorted list?

Insertion sort would run the fastest on an already sorted list, because there would be no inversions! Then at every step, insertion sort would see that there is no inversion and simply move on to the next element. After going through the whole list like this, insertion sort would terminate, resulting in linear time!

(d) Given any list, what is the ideal pivot for quicksort?

The ideal pivot will be the median, or the element that WOULD be at the exact halfway point if the list were to be sorted. This is because it will split the list exactly in half after partitioning around it. When we are able to break our subproblem perfectly in half at every time step, it gives is the recursive tree height of $\log(n)$. With n work done at each level, we achieve $\theta(n \log n)$ runtime in this best case where we pick the median at each step. Note that the worst case pivot choice is choosing the minimum or maximum element (apply the same logic we've used above to see why!), in which we'd have $\theta(n^2)$! Then technically, the upper bound or Big O limit on quicksort is $O(n^2)$. On average though, with random pivot selection, we often see behavior that closely resembles our best case.

(e) So far, in class, we've mostly applied our sorts to lists of numbers. In practice, how would we typically make sure our sorts can be applied to other types?

In practice, if we want our items to be sortable by a comparison sort, we make sure they implement the `Comparable` interface! This involves defining a `compareTo()` method, such that a sorting algorithm has a way to compare our elements, just as naturally as it can compare numbers. This is only relevant for comparison sorts (all of the sorts in this discussion), but not counting sorts (e.g. radix sorts).

3 Zero One Two-Step

(a) Given an array that only contains 0's, 1's and 2's, write an algorithm to sort it in linear time. You may want to use the provided helper method, `swap`.

The solution below is designed in the style of comparison swaps we have seen thus far (indeed, we see it has flavor similar to quicksort with 1 as the pivot, and other sorts that use swapping). Note that there is also a completely valid counting sort approach, but counting sorts are not the focus of this discussion.

```
public static int[] specialSort(int[] arr) {
    int front = 0;
    int back = arr.length - 1;
    int curr = 0;

    while (curr <= back) {
        if (arr[curr] < 1) {
            swap(arr, curr, front);
            front += 1;
            curr += 1;
        } else if (arr[curr] > 1) {
            swap(arr,  curr, back);
            back -= 1;
        } else {
            curr += 1;
        }
    }
}

private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

(b) We just wrote a linear time sort, how cool! Can you explain in a sentence or two why we can't always use this sort, even though it has better runtime than Mergesort or Quicksort?

While our algorithm is super cool, we were only able to write it because we knew there were exactly 3 possible values that could be in our array! For the general case, when the collections we're sorting have much more variety, we can't make these kinds of guarantees.