CS 61B Spring 2022 — Sorting

Exam Prep Discussion 12: April 11, 2022

1 Identifying Sorts

Below you will find intermediate steps in performing various sorting algorithms on the same input list. The steps do not necessarily represent consecutive steps in the algorithm (that is, many steps are missing), but they are in the correct sequence. For each of them, select the algorithm it illustrates from among the following choices: insertion sort, selection sort, mergesort, quicksort (first element of sequence as pivot), and heapsort. When we split an odd length array in half in mergesort, assume the larger half is on the right.

Input list: 1429, | 3291, | 7683, | 1337, | 192, | 594, | 4242, | 9001, | 4392, | 129, | 1000

(a) 1429, 3291, 7683, 192, 1337, | 594, 4242, 9001, 4392, 129, 1000 Merge Sort

1429, 3291, 192, 1337, 7683, | 594, 4242, 9001, 129, 1000, 4392

192, 1337, 1429, 3291, 7683, | 129, 594, 1000, 4242, 4392, 9001

(b) 1337, 192, 594, 129, 1000, $\boxed{1429}$, 3291, 7683, 4242, 9001, 4392 Quick Sort (First item is chosen as pivot)

192, 594, 129, 1000, $\boxed{1337}$, 1429, 3291, 7683, 4242, 9001, 4392

129, $\boxed{192}$, 594, 1000, 1337, 1429, 3291, 4242, 4392, 7683, 9001

(c) 1337, 1429, 3291, 7683, 192, 594, 4242, 9001, 4392, 129, 1000 Insertion Sort

192, 1337, 1429, 3291, 7683, 594, 4242, 9001, 4392, 129, 1000

192, 594, 1337, 1429, 3291, 7683, 4242, 9001, 4392, 129, 1000

(d) 1429, 3291, 7683, 9001, 1000, 594, 4242, 1337, 4392, 129, 192 Heap Sort

7683, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 129, 9001 ← Max Heap

129, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 7683, 9001

In all these cases, the final step of the algorithm will be this:

129, 192, 594, 1000, 1337, 1429, 3291, 4242, 4392, 7683, 9001

2 Sorted Runtimes

We want to sort an array of N **unique** numbers in ascending order. Determine the best case and worst case runtimes of the following sorts:

(a) Once the runs in merge sort are of *size* $<= N/100$, we perform insertion sort on them. Constant number of splits (up to 7) then insertion sort then constant merges.

Best Case: $\Theta(N)$, Worst Case: $\Theta(N^2)$

(b) We can only swap adjacent elements in selection sort.

Best Case: $\Theta(N^2)$, Worst Case: $\Theta(N^2)$

(c) We use a linear time median finding algorithm to select the pivot in quicksort.

Best Case: $\Theta(N \log N)$, Worst Case: $\Theta(N \log N)$ Best pivot have to scan anyway to find elements < and > so adding one more scan is fine

(d) We implement heapsort with a min-heap instead of a max-heap. You may modify heapsort but must maintain constant space complexity.

Best Case: $\Theta(N)$, Worst Case: $\Theta(N \log N)$ Reverse at the end

(e) We run an optimal sorting algorithm of our choosing knowing:

- There are at most N inversions

  Best Case: $\Theta(N)$, Worst Case: $\Theta(N)$

- There is exactly 1 inversion

  Best Case: $\Theta(1)$, Worst Case: $\Theta(N)$

- There are exactly $(N^2 - N)/2$ inversions

  Best Case: $\Theta(N)$, Worst Case: $\Theta(N)$

3 MSD Radix Sort

Recursively implement the method `msd` below, which runs MSD radix sort on a `List` of `Strings` and returns a sorted `List` of Strings. For simplicity, assume that each string is of the same length. You may not need all of the lines below.

In lecture, recall that we used counting sort as the subroutine for MSD radix sort, but any sort works! For the subroutine here, you may use the `stableSort` method, which sorts the given list of strings in place, comparing two strings by the given index. Finally, you may find following methods of the `List` class helpful:

1. `List<E> subList(int fromIndex, int toIndex)`. Returns the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
2. `addAll(Collection<? extends E> c)`. Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.

```java
public static List<String> msd(List<String> items) {

    return msd(items, 0);
}

private static List<String> msd(List<String> items, int index) {

    if (items.size() <= 1 || index >= items.get(0).length()) {
        return items;
    }
    List<String> answer = new ArrayList<>();
    int start = 0;

    ________________________________________________________________________;
    for (int end = 1; end <= items.size(); end += 1) {

        if (end = items.size() || items.get(start).charAt(index) != items.get(end).charAt(index)) {

            List<String> sublist = items.subList(start, end);

            answer.addAll(msd(sublist, index +1));

            start = end;
        }
    }
    return answer;
}
/* You don't need to understand the implementation of this method to use it! */
private static void stableSort(List<String> items, int index) {
    items.sort(Comparator.comparingInt(o -> o.charAt(index)));
}
```

Represents the active index to sort by

Creates the "buckets" to then recursively sort on, by the values of start/end marking the different digits in the index being compared

4 Bears and Beds

The hot new Cal startup AirBearsnBeds has hired you to create an algorithm to help them place their customers in the best possible homes to improve their experience. They are currently in their alpha stage so their only customers (for now) are bears. Now, a little known fact about bears is that they are very, very picky about their bed sizes: they do not like their beds too big or too little - they like them just right. Bears are also sensitive creatures who don't like being compared to other bears, but they are perfectly fine with trying out beds.

The Problem:
Given a list of Bears with unique but unknown sizes and a list of Beds with corresponding but also unknown sizes (not necessarily in the same order), return a list of Bears and a list of Beds such that that the ith Bear in your returned list of Bears is the same size as the ith Bed in your returned list of Beds. Bears can only be compared to Beds and we can get feedback on if the Bed is too large, too small, or just right. In addition, Beds can only be compared to Bears and we can get feedback if the Bear is too large for it, too small for it, or just right for it.

The Constraints:
Your algorithm should run in $O(N \log N)$ time on average. It may be helpful to figure out the naive $O(N^2)$ solution first and then work from there.

Naive solution:

Use two loops, one through bears, one through beds. Create an output set of arrays - when two items match, mark them as used, and add to the output.

Faster (NlogN) solution:

Note each bed can only be matched with one bear, and vice versa.

So, use quicksort on either bears or beds (random partition), then find the corresponding pivot in the other list. In this way, we are sorting through both lists in a synchronized manner.