CS 61B Spring 2022 — Sorting

Exam Prep Discussion 12: April 11, 2022

1 Identifying Sorts

Here is a video walkthrough of the solutions.

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

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

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

Mergesort. One identifying feature of mergesort is that the left and right halves do not interactwith each other until the very end.

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

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

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

Quicksort. First item was chosen as pivot, so the first pivot is 1429, meaning the first iteration should break up the array into something like $| < 1429 | = 1429 | > 1429$

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

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

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

Insertion Sort. Insertion sort starts at the front, and for each item, move to the front as far as possible. These are the first few iterations of insertion sort so the right side is left unchanged

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

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

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

Heapsort. This one's a bit more tricky. Basically what's happening is that the second line is in the middle of heapifying this list into a maxheap. Then we continually remove the max and place it at the end.

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

Here is a video walkthrough of the solutions.

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.

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

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

Once we have 100 runs of size N/100, insertion sort will take best case $\Theta(N)$ and worst case $\Theta(N^2)$ time. The constant number of linear time merging operations don't add to the runtime.

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

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

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

The best case and worst case don't change since swapping at most doubles the work each iteration, which produces the same asymptotic runtime as normal selection sort.

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

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

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

Doing an extra N work each iteration of quicksort doesn't asymptotically change the best case runtime, but it improves the worst case runtime to be the same as the best case. Recall the best case runtime of quicksort is $\Theta(N \log(N))$. You may wonder, why don't we always do this then? Well, there are a couple reasons. First, if the initial array is randomly sorted, the worst case behavior is very improbably. Second, the added linear work per level doesn't change anything asymptotically, but it does slow down the algorithm in practice.

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

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

While a max-heap is better, we can make do with a min-heap by placing the smallest element at the right end of the list until the list is sorted in

descending order. Once the list is in descending order, it can be sorted in ascending order with a simple linear time pass.

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

- There are at most N inversions

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

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

  Recall that insertion sort takes $\Theta(N + K)$ time, where K is the number of inversions. If K is at most N, then, insertion sort has the best and worst case runtime of $\Theta(N)$. Here is an explanation for why no sorting algorithm can surpass this. Notice for our algorithm to terminate we *either* need to address every inversion or look at every element. Since there are at most N inversions, knowing that we have addressed every inversion would take us at least $\Theta(N)$ time. Looking at every element in the list would also take us $\Theta(N)$ time. In either case, we see the runtime of any sorting algorithm cannot be faster than $\Theta(N)$.

- There is exactly 1 inversion

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

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

  The inversion may be the first two elements, in which case constant time is needed. Or, it may involve elements at the end, in which case N time is needed. It can be proven quite simply that no sorting algorithm can achieve a better runtime than above for the best and worst case.

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

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

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

  If a list has $N(N - 1)/2$ inversions, it means it is sorted in descending order! So, it can be sorted in ascending order with a simple linear time pass. We know that reversing any array is a linear time operation, so the optimal runtime of any sorting algorithm is $\Theta(N)$.

3 MSD Radix Sort

Here is a video walkthrough of the solutions.

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.

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

    return __________________________________________________________________;
}

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

    if (____________________________________________________________________) {
        return items;
    }
    List<String> answer = new ArrayList<>();
    int start = 0;

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

        if (________________________________________________________________) {

            ________________________________________________________________;

            ________________________________________________________________;

            ________________________________________________________________;
        }
    }
    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)));
}
```

Solution:

```
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<>();
    stableSort(items, index);
    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)));
}
```

Explanation: MSD sort starts with the leftmost (most significant) digit, grouping and sorting all elements by that digit. It then proceeds recursively on each group. The helper function `msd(items, index)` tells us which index we're currently sorting items by, which is initialized to `0` by the original `msd` function. The base case is if there is 1 item or less (the list is already sorted), or if we've sorted every possible index.

Otherwise, we use `stablesort` to sort by the current index. Note that the subroutine to sort by index *must* be stable; otherwise we lose the ordering imposed by the previous indices we've already sorted.

Inside the loop, `start` and `end` track the start and end indices of our curent group (items that share the same value at `index`). If our current `end` differs from `start`, we must have reached an item with a different value at `index`, so we take everything from `start: end` (exclusive) to get the current group, recursively sorting that group on the next `index`.

4 Bears and Beds

Here is a video walkthrough of the solutions.

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.

Solution:
Our solution will modify quicksort. Let's begin by choosing a pivot from the Bears list. To avoid quicksort's worst case behavior on a sorted array, we will choose a random Bear as the pivot. Next we will partition the Beds into three groups — those less than, equal to, and greater than the pivot Bear. Next, we will select a pivot from the Beds list. This is very important — our pivot Bed will be the Bed that is equal to the pivot Bear. Given that the Beds and Bears have unique sizes, we know that **exactly** one Bed will be equal to the pivot Bear. Next we will partition the Bears into three groups — those less than, equal to, and greater than the pivot Bed.

Next, we will "match" the pivot Bear with the pivot Bed by adding them to the Bears and Beds lists at the same index, which is as easy as just adding to the end. Finally, in the same fashion as quicksort, we will have two recursive calls. The first recursive call will contain the Beds and Bears that are **less** than their respective pivots. The second recursive call will contain the Beds and Bears that are **greater** than their respective pivots.