CS 61B Fall 2023

More Sorting

Exam-Level 13: November 13, 2023

1 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 <= $\frac{N}{100}$, 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. Note that the number of merging operations is actually constant (in particular, it takes about 7 splits and merges to get to an array of size $N/2^7$ = N / 128).

(b) 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, since we have to do N work to partition the array. However, it improves the worst case runtime, since we avoid the "bad" case where the pivot is on the extreme end(s) of the partition.

(c) 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.

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

- There are at most N inversions.

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

  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. Thus, the optimal sorting algorithm would be insertion sort. If K < N, then, insertion sort has the best and worst case runtime of $\Theta(N)$.

- There is exactly 1 inversion.

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

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

  First, we notice that if there is only 1 inversion, it could only involve 2 adjacent elements. Intuitively, if two elements that are apart form an inversion, than some element between these two would also form an inversion with one of the elements.

  (Optional) A formal argument is as follows: suppose the only inversion involves 2 elements that are not adjacent. Let's call their indices i and j, where $i < j$, and $a[i] > a[j]$ (definition of inversion). Because they are not adjacent, there exist some index k, such that $i < k < j$. In case 1, assume $a[k] > a[i]$. Then it follows that $a[k] > a[i] > a[j]$. Because $k < j$ but $a[k] > a[j]$, (k, j) forms an inversion, so we have a contradiction (we assumed ony 1 inversion). In case 2, assume $a[k] < a[i]$, but then we have $k > i$, so (k, i) also form an inversion, which is also a contradiction.

  Using this, we can just compare neighboring elements to find that exact inversion, and swap the 2 elements. If the inversion involves the first two elements, constant time is needed. If the inversion involves elements at the end, N time is needed.

- There are exactly $\frac{N(N-1)}{2}$ inversions

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

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

  If a list has $\frac{N(N-1)}{2}$ inversions, it means it is sorted in descending order. This is because every possible pair is an inversion (The total number of unordered pairs from N elements is $\binom{n}{2}$, or $\frac{N(N-1)}{2}$). 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 $\Theta(N)$.

2 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 stable 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;
}
/* Sorts the strings in `items` by their character at the `index` index alphabetically. */
private static void stableSort(List<String> items, int index) {
    // Implementation not shown
}
```

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<>();
    int start = 0;
    stableSort(items, index);
    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;
}

/* Sorts the strings in `items` by their character at the `index` index alphabetically. */
private static void stableSort(List<String> items, int index) {
    // Implementation not shown
}
```

3 Shuffled Exams

For this problem, we will be working with `Exam` and `Student` objects, both of which have only one attribute: `sid`, which is a integer like any student ID.

PrairieLearn thought it was ready for the final. It had meticulously created two arrays, one of `Exams` and the other of `Students`, and ordered both on `sid` such that the ith `Exam` in the `Exams` array has the same `sid` as the ith `Student` in the `Students` array. Note the arrays are not necessarily sorted by `sid`. However, PrairieLearn crashed, and the `Students` array was shuffled, but the `Exams` array somehow remained untouched.

Time is precious, so you must design a $O(N)$ time algorithm to reorder the Students array appropriately **without** changing the `Exams` array!

Hint: Begin by reordering **both** the `Students` and `Exams` arrays such that ith `Exam` in the `Exams` array has the same `sid` as the ith `Student` in the `Students` array.

Solution:
Let's begin by creating an `ExamWrapper` class that contains two attributes — an `Exam` instance and the `index` of the corresponding `Exam` in the `Exams` array. Next, for each `Exam`, create the corresponding `ExamWrapper` instance.

Run radix sort on the `ExamWrappers`, sorting them on the `sid` of the `Exam` instances. Similarly run radix sort on the list of `Students`, sorting them on `sid` as well. Note that both iterations of radix sort take linear time since the `sid` is of fixed length and of base 10.

At this point in the algorithm, we have "completed" the hint, but we still need move the ith `Student` to its proper place relative to the original `Exams` array. To acheive this, for the i-th `Student`, we will access the ith `ExamWrapper`, and set the index of the i-th `Student` as the `ExamWrapper's index` attribute.

.