CS 61B Spring 2021 — More Sorting

Exam Prep Discussion 13: April 19, 2021

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 $size <= N/100$, we perform insertion sort on them.

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

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

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

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.

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

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

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

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

  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 $\Theta(N)$.

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 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:

```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<>();
    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)));
}
```

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 number 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 ith `Student`, we will access the ith `ExamWrapper`, and set the index of the ith `Student` as the `ExamWrapper's index` attribute.

.