CS 61B Spring 2025

Sorting II

Exam-Level 12: April 21, 2025

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 $\frac{N}{100}$, 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 $\frac{N}{2^7} = \frac{N}{128}$).

(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 use any algorithm to sort the array knowing that:

- 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 only 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 $\frac{N(N-1)}{2}$, which is the number of inversions in a reverse-sorted list. 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 LSD Radix Sort

In this question, we are trying to sort a list of strings consisting of **only lowercase alphabets** using LSD radix sort. In order to perform LSD radix sort, we need to have a subroutine that sorts the strings based on a specific character index. We will use counting sort as the subroutine for LSD radix sort.

(a) Implement the method **`stableSort`** below. This method takes in **`items`** and an **`index`**. It sorts the strings in **`items`** by their character at the **`index`** index alphabetically. It is stable and should run in $O(N)$ time, where N is the number of strings in **`items`**.

```
/* Sorts the strings in `items` by their character at the `index` index alphabetically.
This should modify items instead of returning a copy of it. */
private static void stableSort(List<String> items, int index) {
    Queue<String>[] buckets = new Queue[26];
    for (int i = 0; i < 26; i++) { buckets[i] = new ArrayDeque<>();}
    for (String item : items) {

        char c = _________________;

        int idx = ____________________;

        ______________________________________;
    }
    int counter = 0;

    for (_____________________________________) {

        while (_____________________________________) {

            items.set(counter, bucket.poll());

            counter++;
        }
    }
}
```

Solution:

```
/* Sorts the strings in `items` by their character at the `index` index alphabetically.
This should modify items instead of returning a copy of it. */
private static void stableSort(List<String> items, int index) {
    Queue<String>[] buckets = new Queue[26];
    for (int i = 0; i < 26; i++) {
        buckets[i] = new ArrayDeque<>();
    }
    for (String item : items) {
        char c = item.charAt(index);
        int idx = c - 'a';
        buckets[idx].add(item);
    }

    int counter = 0;
    for (Queue<String> bucket : buckets) {
        while (!bucket.isEmpty()) {
            items.set(counter, bucket.poll());
            counter++;
        }
    }
}
```

(b) Now, using the **stableSort** method, implement the method **lsd** below. This method takes in a **List** of **Strings** and sorts them using LSD radix sort. It should run in $O(N \cdot M)$ time, where N is the number of strings in the list and M is the length of each string.

```
public static List<String> lsd(List<String> items) {
    int length = items.get(0).length();

    for (_____________________________________) {

        ____________________________________;
    }

    return items;
}
```

Solution:

```
public static List<String> lsd(List<String> items) {
    int length = items.get(0).length();

    for (int i = length - 1; i >= 0; i--) {
        stableSort(items, i);
    }

    return items;
}
```

3 MSD Radix Sort

Now, let's solve the same problem as the previous part, but using a different algorithm. 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, and all characters are lowercase alphabets. 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 from the previous question, 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, you've done this in the previous problem!
}
```