More Sorting

Exam-Level 12

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	4/15 Project 3A due					
	4/22 Project 3B/C due					

Content Review

Quicksort - More review

3 Way Partitioning or 3 scan partitioning is a simple way of partitioning an array around a pivot. You do three scans of the list, first putting in all elements less than the pivot, then putting in elements equal to the pivot, and finally elements that are greater. This technique is NOT in place, but it is stable.

3 1 2 5 4

Quicksort - More review

Hoare Partitioning is an unstable, in place algorithm for partitioning. We use a pair of pointers that start at the left and right edges of the array, skipping over the pivot.

The left pointer likes items < the pivot, and the right likes items > the pivot. The pointers walk toward each other until they see something they don't like, and once both have stopped, they swap items.

Then they continue moving towards each other, and the process completes once they have crossed. Finally, we swap the pivot with the pointer that originated on the right, and the partitioning is completed.

3 1 2 5 4

Link to Hoare partitioning demo used in lecture

Comparison Sorts Summary

	Best case	Worst case	Stable?	In Place?
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	no	yes
Insertion Sort	Θ(N)	$\Theta(N^2)$	yes	yes
Heapsort	Θ(N)	Θ(NlogN)	no	yes
Mergesort	Θ(NlogN)	Θ(NlogN)	yes	no (usually)
Quicksort (w/ Hoare Partitioning)	Θ(NlogN)	$\Theta(N^2)$	no (usually)	yes (logN space)

Comparison sorts cannot run faster than Θ(NlogN)! What about counting sorts?

Some radix vocabulary

A **radix** can be thought of as the alphabet or set of digits to choose from in some system. Properly, it is defined as the base of a numbering system. The **radix size** of the English alphabet is 26, and the radix size of Arabic numerals is 10 (0 through 9).

Radix sorts work by using **counting sorts** to sort the list, one digit at a time. This contrasts with what we've learned with **comparison sorts**, which compares elements in the list directly.

LSD Radix Sort

LSD sorts numbers by sorting them by digit from lowest digit to largest digit. We'll see an example of this on the worksheet.

```
120
923
112
342
199
```

General Runtime: $\Theta(W(N + R))$, where:

- W = width of longest key in list
- N = # elements being sorted
- R = radix size

MSD Radix Sort

MSD sorts numbers by sorting them by digit from largest digit to smallest digit. We'll see an example of this on the worksheet.

```
120
923
112
342
199
```

General Runtime: O(W(N + R))

Worksheet

CS 61B
Spring 2024

More Sorting
Exam-Level 12: April 15, 2024

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.

↙ Run merge sort 7 times, then insertion sort on each piece

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

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

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

Similar analysis to mergesort
↳ guarantees best split, maximizing size of each set (max. entropy)

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

Reverse the list after
"Negate" every element before/after > both N time, dominated by NlogN

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

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

- There are at most N inversions.

  Insertion Sort: $\Theta(N+k)$

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

- There is exactly 1 inversion.

  ↙ hard to find the right inversion

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

- There are exactly $\frac{N(N-1)}{2}$ inversions - in reverse order, since every combination pairwise is flipped

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

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 msd(items, 0)_______________________________________________________;
}
                                                         index of the radix sort, what to sort on
private static List<String> msd(List<String> items, int index) {
                 nothing to sort                 end of radix
    if (_items.size()<=1___||__index >= items.get(0).length()___________________________) {
        return items;
    }
    List<String> answer = new ArrayList<>();
    int start = 0;
                                    does the sort
    _stableSort(items, index)___________________________________________________________;
    for (int end = 1; end <= items.size(); end += 1) {
                                  go to next                 difference at radix means new bucket to be created
        if (_end == items.size() || items.get(start).charAt(index) != index.get(end).charAt(index) {
what triggers
making a
new sublist?  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: While you cannot modify the `Exams` array, you can sort a copy of the `Exams` array with some added information. Think about what information would be useful to put back the `Students` array in the same order as the exams.

Option 1 (not intended Solution): Hashmaps

1. Make hashmap of sid → Student instance in N time
2. Go through exams, find corresponding Student, then Sort!

↳ into StudentsCopy

Option 2 (intended Solution): Radix Sort

1. Make an ExamWrapper containing Exam and its index
2. Radix sort on these EW - sid fixed length, base 10
3. Move corresponding students to index in EW