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	$\Theta(N)$	$\Theta(N^2)$	yes	yes
Heapsort	$\Theta(N)$	$\Theta(N\log N)$	no	yes
Mergesort	$\Theta(N\log N)$	$\Theta(N\log N)$	yes	no (usually)
Quicksort (w/ Hoare Partitioning)	$\Theta(N\log N)$	$\Theta(N^2)$	no (usually)	yes (logN space)

Comparison sorts cannot run faster than $\Theta(N\log N)$! 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

1A 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:

1A 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:

Once the runs in merge sort are of size < N/100, we perform insertion sort on them.
Best case: Θ( )
Worst case: Θ( )

1A 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:

Once the runs in merge sort are of size < N/100, we perform insertion sort on them.
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 $N / 2^7 \sim N / 100$).

1B 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:

We use a linear time median finding algorithm to select the pivot in quicksort.
Best case: Θ( )
Worst case: Θ( )

1B 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:

We use a linear time median finding algorithm to select the pivot in quicksort.
Best case: Θ(N log N)
Worst case: Θ(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.

1C 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:

We implement heapsort with a min-heap instead of a max-heap. You may modify heapsort but must maintain constant space complexity.
Best case: Θ( )
Worst case: Θ( )

1C 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:

We implement heapsort with a min-heap instead of a max-heap. You may modify heapsort but must maintain constant space complexity.
Best case: Θ(N log N)
Worst case: Θ(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.

1D 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:

We run an optimal sorting algorithm of our choosing knowing there are at most N inversions.
Best case: Θ( )
Worst case: Θ( )

1D 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:

We run an optimal sorting algorithm of our choosing knowing there are at most N inversions.
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 $\Theta(N)$.

1D 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:

We run an optimal sorting algorithm of our choosing knowing there is exactly 1 inversion.
Best case: Θ( )
Worst case: Θ( )

1D 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:

We run an optimal sorting algorithm of our choosing knowing there is exactly 1 inversion.
Best case: $\Theta(1)$
Worst case: $\Theta(N)$

That one inversion must only involve two adjacent elements. 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.

1D 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:

We run an optimal sorting algorithm of our choosing knowing there are N(N - 1)/2 inversions.
Best case: Θ( )
Worst case: Θ( )

1D 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:

We run an optimal sorting algorithm of our choosing knowing there are N(N - 1)/2 inversions.
Best case: Θ(**N**)
Worst case: Θ(**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 Θ(N).

2 MSD Radix Sort

```
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;
}
```


2 MSD Radix Sort

```
public static List<String> msd(List<String> items) {
    return msd(items, 0);
}
```


2 MSD Radix Sort

```
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;
}
```


2 MSD Radix Sort

```
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;
    ____________________________________________________________________;
    for (int end = 1; end <= items.size(); end += 1) {
        if (_____________________________________________________________) {
            ____________________________________________________________;
            ____________________________________________________________;
            ____________________________________________________________;
        }
    }
    return answer;
}
```


2 MSD Radix Sort

```
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 (_____________________________________________________________) {
            ____________________________________________________________;
            ____________________________________________________________;
            ____________________________________________________________;
        }
    }
    return answer;
}
```


2 MSD Radix Sort

```
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)) {
            ____________________________________________________________;
            ____________________________________________________________;
            ____________________________________________________________;
        }
    }
    return answer;
}
```


2 MSD Radix Sort

```
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);
            ____________________________________________________________;
            ____________________________________________________________;
        }
    }
    return answer;
}
```


2 MSD Radix Sort

```
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));
            ____________________________________________________________;
        }
    }
    return answer;
}
```


2 MSD Radix Sort

```
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;
}
```


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 an integer like any student ID, and is at most 10 digits long.

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.

3 Shuffled Exams

```
Exams = [499, 317, 351, 409, 150]
Students = [409, 351, 317, 150, 499]
```

Goal: `Students = [499, 317, 351, 409, 150]`

3 Shuffled Exams

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.

```
Exams = [E499, E317, E351, E409, E150]
Students = [S409, S351, S317, S150, S499] // shuffled
ExamWrappers = [(499, 0), (317, 1), (351, 2), (409, 3), (150, 4)]
```


3 Shuffled Exams

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.

Exams = [E_{499}, E_{317}, E_{351}, E_{409}, E_{150}]
Students = [S_{150}, S_{317}, S_{351}, S_{409}, S_{499}]
ExamWrappers = [(150, 4), (317, 1), (351, 2), (409, 3), (499, 0)]

3 Shuffled Exams

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.

```
Exams = [E499, E317, E351, E409, E150]
Students = [S150, S317, S351, S409, S499]
StudentsCopy = [, , , , S150]
ExamWrappers = [(150, 4), (317, 1), (351, 2), (409, 3), (499, 0)]

StudentsCopy[4] = Students[0]
```


3 Shuffled Exams

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.

`Exams` = [E_{499}, E_{317}, E_{351}, E_{409}, E_{150}]
`Students` = [S_{150}, S_{317}, S_{351}, S_{409}, S_{499}]
`StudentsCopy` = [, S_{317}, , , S_{150}]
`ExamWrappers` = [(150, 4), (317, 1), (351, **2**), (409, 3), (499, 0)]

```
StudentsCopy[1] = Students[1]
```


3 Shuffled Exams

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.

`Exams` = [E_{499}, E_{317}, E_{351}, E_{409}, E_{150}]
`Students` = [S_{150}, S_{317}, S_{351}, S_{409}, S_{499}]
`StudentsCopy` = [, S_{317}, S_{351}, , S_{150}]
`ExamWrappers` = [(150, 4), (317, 1), (351, **2**), (409, 3), (499, 0)]

```
StudentsCopy[2] = Students[2]
```


3 Shuffled Exams

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.

`Exams` = [E_{499}, E_{317}, E_{351}, E_{409}, E_{150}]
`Students` = [S_{150}, S_{317}, S_{351}, S_{409}, S_{499}]
`StudentsCopy` = [, S_{317}, S_{351}, S_{409}, S_{150}]
`ExamWrappers` = [(150, 4), (317, 1), (351, 2), (409, **3**), (499, 0)]

```
StudentsCopy[3] = Students[3]
```


3 Shuffled Exams

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.

$\texttt{Exams} = [E_{499}, E_{317}, E_{351}, E_{409}, E_{150}]$
$\texttt{Students} = [S_{150}, S_{317}, S_{351}, S_{409}, S_{499}]$
$\texttt{StudentsCopy} = [S_{499}, S_{317}, S_{351}, S_{409}, S_{150}]$
`ExamWrappers = [(150, 4), (317, 1), (351, 2), (409, 3), (499, 0)]`

```
StudentsCopy[0] = Students[4]
```