CSM 61B Spring 2019

More Sorting!

Mentoring 11: April 15, 2019

1 Stability

Stability is a property of some sorting algorithms. Stability essentially means that if we have two elements that are equal, then their relative ordering in the sorted list is the same as the ordering in the unsorted list. For instance, let's say that we had an array of integers.

```
{ 1, 2, 1, 3, 1, 2, 4 }
```

Since we have multiple 1 and 2s, let's label these.

```
{ 1A, 2A, 1B, 3, 1C, 2B, 4 }
```

A stable sort would result in the final list being

```
{ 1A, 1B, 1C, 2A, 2B, 3, 4 }
```

Why is this desirable? Say that we have an Excel spreadsheet where we are recording the names of people who log in to CSM Scheduler. The first column contains the timestamps, and the second column contains their username. The timestamps are already ordered in increasing order. If we wanted to sort the username, so that we could group the list to see when each username logs in, we would want that the timestamps maintain their relative order. This is precisely what a stable sort ensures.

1.1 Why does Java's built-in `Array.sort` method use quicksort for **`int`**, **`long`**, **`char`**, or other primitive arrays, but merge sort for all `Object` arrays?

Fast, in-place solutions for quicksort are unstable, meaning that, for any two equivalent keys, their final order in the output is not guaranteed to be the same. Merge sort has good asymtotic behavior without sacrificing on stability.

2 Pivot Choice

2.1 For each pivot selection strategy below, what is the best, average and worst case runtime?

The best case and average case runtime for quicksort in each scenario is still in $\Theta(N \log N)$.

(a) Always choose the first value in the list.

If we always select the first value as a pivot, the values in the list will determine the runtime. If we have a sorted or reverse sorted array, selecting the first value will only reduce the problem size by a single value during each call: we essentially have selection sort! The runtime in this worst-case scenario is $\Theta(N^2)$.

But if the values in the list are randomly shuffled, then choosing the first value not degrade into the worst-case runtime.

(b) Always find and choose the median value in the list. Assume finding the median takes $O(N)$ time where N is the length of the list.

The worst case runtime in this scenario will be in $\Theta(N \log N)$ including the time to find the median (which can be found in linear time using the median of medians algorithm). In reality, the additional cost associated with finding the median is usually not worth it over simply selecting a random value.

Even though median-finding takes linear time, quicksort normally needs to spend linear time bucketing values into less-than, equal-to, and greater-than buckets anyways so the asymptotic runtime is not affected.

(c) Always choose a random pivot.

It is possible for this to degrade to $\Theta(N^2)$ runtime in the worst case as well. If we pick a random pivot from an array we have a $\frac{1}{N}$ probability of selecting the smallest value and, after that, a $\frac{1}{N-1}$ of selecting the next smallest, and so forth. The total probability of selecting the smallest value every time is $\frac{1}{N!}$.

Although it is possible for for quicksort to run in $\Theta(N^2)$, it becomes less and less probable as N, or the size of the list, increases.

3 Even More Sorting

In the table below, the runtimes of the sorts gone over in last week's section are written for you. Fill out the best-case and worst-case runtimes for Quicksort as well as whether all of the sorts we've seen so far are stable or not.

Algorithm	Best-case	Worst-case	Stable
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	Depends
Insertion Sort	$\Theta(N)$	$\Theta(N^2)$	Yes
Merge Sort	$\Theta(N \log N)$	$\Theta(N \log N)$	Yes
Heapsort	$\Theta(N)$	$\Theta(N \log N)$	No
Quicksort	$\Theta(N \log N)$	$\Theta(N^2)$	Depends

Selection Sort In selection sort, we loop through the array to find the smallest element. Next, we swap the element at index-0 with the smallest element. Next, we repeat this procedure, but only looking at the array starting at index-1.

- *Runtime, Best, Worst Case:* Since it takes $O(N)$ time to loop through the array, and we loop through the array N times, this algorithm has a runtime of $\Theta(N^2)$. Note that even if the array is already sorted, we need to iterate through it to find the minimum, and then iterate through it again, and again, N times.
- *Stability:* Consider an array `{ 3A, 2, 3B, 1 }`, where the 3s have been labeled to differentiate between them. The algorithm will find 1 to be the smallest, and will swap it with $3A$, pushing $3A$ after $3B$, making it not stable. However, it is also possible to make it stable if we implement Selection Sort in a different way, which involves creating a new array instead of swapping the minimum elements.

Insertion Sort This is the way an adult would normally sort a pack of cards. Iterating through the array, swapping each element left-wards.

- *Best Case:* Given a sorted array, `{ 1, 2, 3, 4 }`, this algorithm would iterate through the array just once, and do 0 swaps, since all elements are already as left-wards as they can be.

- *Worst Case:* Given a fully unsorted array, `{ 4, 3, 2, 1 }`, this algorithm would first swap $(3, 4)$, then to move 2 left-wards, it needs to do 2 swaps. Finally to move 1 left-wards, it needs to do 3 swaps. This is of the ordering of $O(n^2)$ swaps.
- *Stability:* Consider an array `{ 3A, 2, 3B, 1 }`. We would get the following steps: `{ 2, 3A, 3B, 1 }`, `{ 1, 3, 3A, 3B, }`. In general, this algorithm is stable, because given a $3A, 3B$, we would never swap them with each other.

Merge Sort Given an array, divide it into two equal halves, and call merge-sort recursively on each half. Take the recursive leap of faith and assume that each half is now sorted. Merge the two sorted halves. Merging takes a single iteration through both arrays, and takes $O(N)$ time. The base case is if the input list is just 1 element long, in which case, we return the list itself.

- *Best case, Worst Case, Runtime:* Since the algorithm divides the array and recurses down, this takes $\Theta(N \log N)$ time, no matter what.
- *Stability:* Merge sort is made stable by being careful during the merging step of the algorithm. If deciding between 2 elements that are the same, one in the left half and one in the right half, pick the one from the left half first.

Heap Sort Place all elements into a heap. Remove elements one by one from the heap, and place them in an array.

- *Recall:* Creating a heap of N elements takes $N \log N$ time, because we have to bubble-up elements. Removing an element from a heap takes $\log N$ time, also because of bubbling and sinking.
- *Best Case:* Say that all the elements in the input array are equal. In this case, creating the heap only takes $O(N)$ time, since there is no bubbling-down to be done. Also, removing from the heap takes constant time for the same reason. Since we remove N elements, and creating the heap takes $O(N)$ time, the overall runtime is $O(N)$.
- *Worst Case:* Any general array would require creating the heap with bubbling which itself takes $N \log N$ time.
- *Runtime:* HeapSort is not stable. Consider two elements(3a and 3b) that are considered equal. Based on the implementation, we pop the max element from the heap and add it to the end. This naturally puts 3a after 3b in the array after the two pops.

For example, say the original array is already in heap structure: `{3(a),3(b),2,1}`. After the first pop of the heap and add it to the end, it will look like: `{3(b),1,2,3(a)}`. And the end result would be `{1,2,3(b),3(a)}`.

Quicksort Based on some pivot-picking strategy, pick a pivot. Divide the array up into 3 groups: elements smaller than the pivot, larger than the pivot and equal to the pivot. Recursively sort the first and second group.

- *Runtime:* Analyzed in detail in the next question.
- *Stability:* QuickSort is generally not implemented as a stable algorithm, assuming we are using Tony Hoare's in-place partitioning implementation. It is not stable because the algorithm swaps non-adjacent elements. However, if we use an extra space of $O(N)$, we can implement a stable QuickSort.

3.1 Run the quicksort algorithm. Assume we pick the middle element as the pivot; if there is no exact middle, pick the element to the right of the middle.

```
{ 1, 3, 8, 2, 6, 4, 5, 9 }

{ 1, 3, 2, 4, 5 }, { 6 }, { 8, 9 }
{ 1 }, { 2 }, { 3, 4, 5 }, { 6 }, { 8 }, { 9 }
{ 1 }, { 2 }, { 3 }, { 4 }, { 5 }, { 6 }, { 8 }, { 9 }
```

4 Sorting Out My Head!

4.1 Web developers use many different sorts for the different types of lists that they might want to sort. For each of these, provide the best sorting algorithm amongst the following: Mergesort, Quicksort (with Hoare Partitioning), Insertion Sort, LSD Sort. Also, state the worst-case runtime.

(a) A list of N packets received by a server over time. Each packet has the timestamp at which the sender sent it. However, some packets may be dropped or arrive out-of-order due to the faulty network. Sort this list by that timestamp (sent time).

Since we expect the list to be largely sorted by time already, with a few packets out of place, we should use insertion sort. Worst case runtime is $O(N^2)$.

(b) A list of N websites. Each website has the number of total visitors. Sort this list by visitor count.

Quicksort, since it's generally pretty fast for sorting what is effectively a random list of numbers. Worst case runtime is $O(N^2)$. Could also argue for LSD sort since there might be some limit k on the total number of visitors, but less preferable.

(c) After sorting by visitor count, we now want to sort by webpage file size. If websites have the same file size, they should be ordered by visitor count.

Mergesort, since we want to sort stably. Worst case runtime is $O(N \log N)$.

(d) A list of 20 names. Sort in alphabetical order.

Insertion sort, since it has the least overhead and is fastest for small lists. Worst case runtime is $O(1)$, since 20 is a constant, and we assume that all names are shorter than some fixed constant as well.

5 QuickSort vs. Merge Sort

5.1 (a) What are the advantages and disadvantages of quicksort?

Advantages:

- Faster on average by a constant factor than merge sort
- In-place variant for $\Theta(1)$ memory complexity. Including the average-case call stack brings up space complexity to $\Theta(\log N)$
- Multi-pivot quicksort offers greater constant factor optimizations

Disadvantages:

- Most efficient implementations (in-place) are not stable
- Worst-case runtime is in $\Theta(N^2)$, thus making it a poor choice for adversarial datasets
- Not very good for external sorting

(b) What are the advantages and disadvantages of merge sort?

Advantages:

- Stable
- Good for external sorting
- Constant space usage for sorting linked lists

Disadvantages:

- Slower than quicksort on average
- Higher space complexity for array allocation

6 Cheapest Flights Within K Stops *Extra Practice*

6.1 This question was adapted from LeetCode: cheapest-flights-within-k-stops

There are n cities connected by m flights. Each flight starts from city u and arrives at v with a price w. Thus a city is represented as [u, v, w].

Given all the cities and flights, together with starting city `src` and the destination `dst`, your task is to find the cheapest price from `src` to `dst` with up to K stops. If there is no such route, output -1.

The solution was adapted from: https://leetcode.com/problems/cheapest-flights-within-k-stops/discuss/180314/Dijkstra-with-Modification/187297/.

META: This question is "EXTRA HARD". The complete walkthrough of the codes is not recommended, instead, please let students fully understand the following:

We can modify Dijkstra's to solve this problem! Dijkstra's Algorithm by itself doesn't work because we need to limit our solution to K stops. Instead of adding explored nodes to a closed set, then, we simply check if the current count is over K. See code below:

```
public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {
    Map<Integer, Map<Integer, Integer>> map = new HashMap<>();
    for (int[] flight : flights) {
        map.putIfAbsent(flight[0], new HashMap<>());
        map.get(flight[0]).put(flight[1], flight[2]);
    }
    PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) ->  a[1] - b[1] );
    pq.offer(new int[]{ src, 0, 0 });
    while (!pq.isEmpty()) {
        int[] current = pq.poll();
        int city = current[0]; int cost = current[1]; int count = current[2];
        if (count > K + 1) continue;
        if (city == dst) {return cost;}
        if (map.containsKey(city)) {
            Map<Integer, Integer> nexts = map.get(city);
            for (int nextCity : nexts.keySet()) {
                pq.offer(new int[]{ nextCity, cost + nexts.get(nextCity), count + 1 });
            }
        }
    }
    return -1;
}
```