CSM 61B Fall 2017

Sorting Algorithms

Mentoring 10: October 30, 2017

1 Asymptotics Potpourri

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.

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	$\Theta(N)$	$\Theta(N \log N)$	Hard
Quicksort	$\Theta(N)$	$\Theta(N^2)$	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.

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.

2 QuickCo

Malicious Mallory has been hired by a competitor to break into QuickCo, the world leader in sorting algorithms, and tamper with its Quicksort algorithms by making them as slow as possible. Mallory succeeded in unlocking the mainframe, but now she needs your help in slowing QuickCo's algorithms down to a halt!

For this question, assume that the Quicksort algorithm used has three steps.

1. Iterate through the array, to count how many elements are smaller than the pivot, larger than the pivot, and equal to the pivot.
2. Create 3 arrays for smaller, larger and equal elements of the correct size and then, in a second run through the array, place the appropriate elements in these arrays.
3. Finally, recurse on the smaller and larger arrays.

```
int[] data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

2.1 Mallory decides to change the way QuickCo chooses a pivot for Quicksort.

Given the `int[] data` of size `n`, what choice of pivot would cause the worst-case runtime for Quicksort? Expand out the first few steps of the quicksort algorithms in such a case.

First, last, or alternating first-last. The Quicksort algorithm will continue in the following fashion:

```
{ 1 }, { 2, 3, 4, 5, 6, 7, 8 }
{ 1 }, { 2 }, { 3, 4, 5, 6, 7, 8 }
{ 1 }, { 2 }, { 3 }, { 4, 5, 6, 7, 8 }
```

...

Since the size of our problem is decreasing only by one at each step, this will end up doing $O(N)$ work in the first step, $O(N-1)$ in the second step, $O(N-3)$ in the next, and so on. This will take $O(N^2)$ work.

2.2 Mallory finds an algorithm which always selects the middle element but she is unable to gain write access to it. If the array has an even number of elements, the algorithm picks the element to the left. However, Mallory discovers a way to intercept the incoming data and rearrange it before the algorithm runs.

Given the `int[] data` of size `n`, rearrange the numbers such that the algorithm will run in its worst-case time. Expand out the first few steps of the quicksort algorithms in such a case.

```
{ 6, 7, 8, 9, 1, 2, 3, 4, 5 }
```

In this case, the Quicksort algorithm will proceed as follows:

```
Smaller: { 6, 7, 8, 9, 2, 3, 4, 5 }; Equal: { 1 }
{ 1 }, { 6, 7, 8, 2, 3, 4, 5 }, { 9 }
{ 1 }, { 2 }, { 6, 7, 8, 3, 4, 5 }, { 9 }
```

...

With a similar reasoning as the previous case, this will also take $O(N^2)$ work.

2.3 Does the worst-case runtime of Quicksort depend on the array order, pivot choice, or both? Why?

The worst-case runtime of Quicksort depends on both the array order and the choice of pivots. The worst-case always occurs when the pivot's final position is on an end of the array, which means it was either the smallest or the largest element.

2.4 If the worst-case runtime of Quicksort is $O(N^2)$ while the worst-case runtime of Mergesort is $O(N \log N)$, why do we ever use Quicksort?

In most cases, Quicksort actually performs very well, with an average runtime of $O(N \log N)$. The probability that Quicksort ends up running in $O(N^2)$ is, in reality, very, very low. There is a mathematical discussion about this probability here.

Most implementations of MergeSort requires extra space (more arrays to be created). However, there are more complicated ways to write a MergeSort algorithm that doesn't require any extra space, but Quicksort is still often preferred when the number of elements being sorted isn't extremely large.

The general idea for this is that if all the data being sorted fits inside your RAM, then Quicksort is faster, since it doesn't need to access data stored in your Harddisk, which is slower. For reference, in 1GB RAM, we can hold an array of 32 million integers. This means, that for almost all regular purposes, Quicksort is faster. More information about this is available on the first and second answer here.

3 Vertigo

3.1 We have a list of N elements that should be sorted, but to our surprise we recently discovered that there are at most k pairs out of order, or k **inversions**, in the list. The list `{ 0, 1, 2, 6, 4, 5, 3 }`, for example, contains 5 inversions: $(6, 4), (6, 5), (6, 3), (4, 3), (5, 3)$.

For each value of k below, state the most efficient sorting algorithm and give a tight asymptotic runtime bound.

(a) $k \in O(\log N)$

Insertion sort is the most efficient in this case because its runtime is $O(N + k)$. The overall runtime bound for insertion sort in this scenario is $O(N)$.

(b) $k \in O(N)$

Insertion sort for the same reason above. The overall runtime bound for insertion sort in this scenario is $O(N)$.

(c) $k \in O(N^2)$

Merge sort, quicksort, or heap sort would be ideal here since the number of inversions causes insertion sort to run in $O(N^2)$ runtime. Using one of the three sorts listed earlier yields a runtime in $O(N \log N)$ in the normal case.

4 Getting to Know You

4.1 Run MSD and LSD radix sort on the following DNA sequence such that the output is sorted in alphabetical order ($A < C < G < T$).

Most-Significant Digit

```
ACAG ACAG ACAG ACAG ACAA
CTAG ACAA ACAA ACAA ACAG
ACAA CTAG CCTC CCTC CCTC
TGAG CCTC CTAG CTAG CTAG
CCTC GAGT GAGT GAGT GAGT
GAGT TGAG TGAG TGAG TGAG
```

Least-Significant Digit

```
ACAG ACAA ACAA GAGT ACAA
CTAG CCTC ACAG ACAA ACAG
ACAA ACAG CTAG ACAG CCTC
TGAG CTAG TGAG CCTC CTAG
CCTC TGAG GAGT TGAG GAGT
GAGT GAGT CCTC CTAG TGAG
```

4.2 Performing Radix Sort seems to be a fast sorting algorithm. Why don't we always use it?

Radix sort is only possible when the elements being compared have some radix or base. For strings, they can be broken up into individual characters and separately compared, as we did above. We can also do the same for integers. However, what if we wanted to compare 10 `Cat` objects? We cannot compare `Cat` objects by breaking them up into any smaller pieces or radices.

4.3 Why does Least-Significant Digit radix sort work?

The key is to maintain stability during sorting. If we make sure that at each step we sort stably, then the following proof explains why LSD sort works.

Consider the final step of the algorithm, where we are sorting the first digit.

If two strings differ on the first character, then their ordering will now be fixed, and since the previous steps of the algorithm relatively sorted the rest of the string, the ordering will be fully correct.

If two strings do not differ on the first character, e.g., we having CAT and CBX, we know that CAT would appear before CBX because of the previous steps. All we have to ensure is that this final step doesn't change the ordering of CAX and CBX. This will be ensured, since we make sure we sort stably.

5 More Asymptotics Potpourri *Extra Practice*

Algorithm	Best-case	Worst-case	Stable
Counting Sort	$\Theta(N+R)$	$\Theta(N+R)$	Yes
LSD Radix Sort	$\Theta(W(N+R))$	$\Theta(W(N+R))$	Yes
MSD Radix Sort	$\Theta(N+R)$	$\Theta(W(N+R))$	Depends

Where N is the length of the list, R is the size of the alphabet (radix), and W is the length of the longest word.

MSD radix sort can be made stable with additional space for a buffer.

6 Berkeley Bytes Buffet *Extra Practice*

You are the proud owner of Berkeley Bytes Buffet and business is good! You have a policy where children under 8 eat free and seniors eat 50 percent off. Since you're a savvy business owner, you keep the ages of all your customers.

6.1 For taxes, you must to submit a list of the ages of your customers in sorted order. Define `ageSort`, which takes an `int[] array` of all customers' ages and returns a sorted array. Assume customers are less than 150 years old.

```
public class BerkeleyBytes {
    private static int maxAge = 149;
    public static int[] histogram(int[] ages) {
        int[] ageCounts = new int[maxAge + 1];
        for (int age : ages) {
            ageCounts[age] += 1;
        }
        return ageCounts;
    }
    public static int[] ageSort(int[] ages) {
        int[] ageCounts = histogram(ages);
        int[] result = new int[ages.length];
        int index = 0;
        for (int age = 0; age < maxAge; age++) {
            for (int count = 0; count < ageCounts[age]; count++) {
                result[index] = age;
                index += 1;
            }
        }
        return result;
    }
}
```

6.2 Time passes and your restaurant is doing well. Unfortunately, our robot overlords advanced medicine to the point where humans are now immortal.

(a) How could we extend the algorithm to accept a list of any ages?

Radix sort. Sort the customers using the above algorithm, looking at only the last digit of their age. We would need 10 buckets, since the digit can only have 10 values. Repeat with the second to last digit, and so on, until the first digit sorted.

(b) When would we be able to use this type of sort?

The keys we want to sort must have some **base** (or radix). The type of item must be some combination of symbols.