CS 61B Spring 2025

Sorting II

Discussion 12: April 21, 2025

1 Quicksort

(a) Sort the following unordered list using Quicksort. Assume that we always choose first element as the pivot and that we use the 3-way merge partitioning process described in lecture. Show the steps taken at each partitioning step.

`18, 7, 22, 34, 99, 18, 11, 4`

Solution:

```
-18-, 7, 22, 34, 99, 18, 11, 4
-7-, 11, 4 | 18, 18 | 22, 34, 99
4, 7, 11, 18, 18 | -22-, 34, 99
4, 7, 11, 18, 18, 22 | -34-, 99
4, 7, 11, 18, 18, 22, 34, 99
```

(b) What is the best and worst case running time of Quicksort with Hoare partitioning on N elements? Given the two lists `[4, 4, 4, 4, 4]` and `[1, 2, 3, 4, 5]`, assuming we pick the first element as the pivot every time, which list would result in better runtime?

Solution: Best: $\Theta(N \log N)$ Running Quicksort on a list that has a pivot splits the partition exactly in half will result in $\Theta(\log N)$ levels, with the same amount work as above (i.e. $\Theta(N)$ at each level). For example, `[3, 1, 2, 5, 4]`. An alternative case is when we have all of the same element in the array (i.e. `[4, 4, 4, 4, 4]`), since the two pointers in Hoare partitioning always end up in the middle.

Worst: $\Theta(N^2)$. In general, the worst case is such that the partioning scheme repeatedly partions an array into one element and the rest.

Running Quicksort on a sorted list will take $\Theta(N^2)$ if the pivot chosen is always the first or last in the subarray: `[1, 2, 3, 4, 5]`. At each level of recursion, you will need to do $\Theta(N)$ work, and there will be $\Theta(N)$ levels of recursion. This sums up to $1 + 2 + ... + N$.

(c) What are two techniques that can be used to reduce the probability of Quicksort taking the worst case running time?

Solution:

1. Randomly choose pivots.
2. Shuffle the list before running Quicksort.

2 Radix Sorts

(a) Sort the following list using LSD Radix Sort with counting sort. Show the steps taken after each round of counting sort. The first row is the original list and the last two rounds are already filled for you.

	30395	30326	43092	30315
1				
2				
3				
4	30315	30326	30395	43092
5	30315	30326	30395	43092

Solution:
The underlined sections denote the digits that have already been sorted.

	30395	30326	43092	30315
1	43092	30395	30315	30326
2	30315	30326	43092	30395
3	43092	30315	30326	30395
4	30315	30326	30395	43092
5	30315	30326	30395	43092

(b) Sort the following list using MSD Radix Sort with counting sort. Show the steps taken after each round of counting sort. The first row is the original list and the first round is already filled for you.

	21295	22316	30753	21248	30751
1	21295	22316	21248	30753	30751
2					
3					
4					
5					

Solution:

	21295	22316	30753	21248	30751
1	21295	22316	21248	30753	30751
2	21295	21248	22316	30753	30751
3	21295	21248	22316	30753	30751
4	21248	21295	22316	30753	30751
5	21248	21295	22316	30751	30753

(c) Give the best case runtime, worst case runtime, and stability for both LSD and MSD radix sort. Assume we have N elements, a radix R, and a maximum number of digits in an element W.

	Time Complexity (Best)	Time Complexity (Worst)	Stability
LSD Radix Sort			
MSD Radix Sort			

Solution:

	Time Complexity (Best)	Time Complexity (Worst)	Stability
LSD Radix Sort	$\Theta(W(N+R))$	$\Theta(W(N+R))$	Yes
MSD Radix Sort	$\Theta(N+R)$	$\Theta(W(N+R))$	Yes

(d) We saw in part (c) that radix sort has good runtime with respect to the number of elements in the list. Given this fact, can we say that radix sort is the best sort to use?

No. Though radix sort runs linear with respect to the number of elements in the list, the runtime also depends on the size of the radix R and the length of the longest "word" W (or the number of digits in a number). Additionally, it is not always possible to use radix sort, because not all objects can be split up into digits. However, comparison sorts can be used on *any* object that defines a `compareTo` method, and would work well with `compareTo` methods that are fast.

3 Sort Identification

Match the sorting algorithms to the sequences, each of which represents several intermediate steps in the sorting of an array of integers. Assume that for quicksort, the pivot is always the first item in the sublist being sorted. Note that these steps are not necessarily the first few intermediate steps and there may be steps which are skipped.

Algorithms: *Quicksort, Merge Sort, Heapsort, MSD Radix Sort, Insertion Sort*

(a)
```
12, 7, 8, 4, 10, 2, 5, 34, 14
7, 8, 4, 10, 2, 5, 12, 34, 14
4, 2, 5, 7, 8, 10, 12, 14, 34
```

Solution: Quicksort. A pattern we can see is that most of the elements remain in the same order relative to one another, but the first element keeps moving around. Taking a closer look at this element, we note that everything to the left is less than it, and everything to the right is greater than, indicating quicksort.

(b)
```
23, 45, 12, 4, 65, 34, 20, 43
4, 12, 23, 45, 65, 34, 20, 43
```

Solution: Insertion Sort. A surefire way of identifying insertion sort is the fact it slowly builds a partially sorted array on the left hand side, which is exactly what occurs here.
Another solution is merge sort (with a recursive implementation), where the left half has already been fully merge-sorted

(c)
```
12, 32, 14, 11, 17, 38, 23, 34
12, 14, 11, 17, 23, 32, 38, 34
```

Solution: MSD Radix Sort. We first see that the numbers have been moved all over the place, which is odd, but taking a closer look at how the numbers are oriented, we see that all the tens digits are ordered in ascending order (10s, then 20s and 30s). This is immediately indicative of MSD Radix Sort, sorting from the leftmost digit first.

(d)
```
45, 23, 5, 65, 34, 3, 76, 25
23, 45, 5, 65, 3, 34, 25, 76
5, 23, 45, 65, 3, 25, 34, 76
```

Solution: Merge Sort. We notice that the numbers stay relatively close to where they begin, but we also see little sorted runs inside the array, like (23, 45) and (5, 65) for example. These small sorted subarrays and the merging of the subarrays in halves of the array is indicative of Merge Sort.

(e)
```
23, 44, 12, 11, 54, 33, 1, 41
54, 44, 33, 41, 23, 12, 1, 11
44, 41, 33, 11, 23, 12, 1, 54
```

Solution: Heapsort. Heapsort always heapifies the array first, which on the second line we see the maximum element, 54, at the start of the array. Heapifying usually shuffles around the array, which we also see. Placing the maximum element at the back of the array and bubbling up the 44 seals the deal that this is heapsort.

4 Conceptual Comparison Sorts (Extra)

Answer the following questions regarding various sorting algorithms that we've discussed in class. If the question is T/F and the statement is true, provide an explanation. If the statement is false, provide a counterexample.

(a) Give a 5 integer array that causes the worst case runtime for insertion sort.

Solution: A simple example is `[5, 4, 3, 2, 1]`. Any 5 integer array in descending order would work.

(b) Give some reasons as to why someone would use merge sort over quicksort.

Solution:
Some possible answers:
- Merge sort has $\Theta(N \log N)$ worst case runtime versus quicksort's $\Theta(N^2)$.
- Merge sort is stable, whereas quicksort typically isn't.
- Merge sort can be highly parallelized because as we saw in the first problem the left and right sides don't interact until the end.
- Merge sort is preferred for sorting a linked list.

(c) You are given the following options:

(A) Quicksort (in-place using Hoare partitioning and choose the leftmost item as the pivot)
(B) Merge Sort
(C) Selection Sort
(D) Insertion Sort
(E) Heapsort
(F) None of the above

For each of the statements below, list all letters that apply. Each option may be used multiple times or not at all. Note that all answers refer to the entire sorting process, not a single step of the sorting process, and assume that N indicates the number of elements being sorted.

_____ Bounded by $\Omega(N \log N)$ lower bound.

Solution: A, B, C. Both insertion sort and heapsort both have a lower bound of $\Omega(N)$.

_____ Worst case runtime that is asymptotically better than quicksort's worst case runtime.

Solution: B, E. Quicksort has a worst case runtime of $O(N^2)$, while both merge sort and heapsort have a worst-case runtime of $O(N \log N)$.

_____ In the worst case, performs $\Theta(N)$ pairwise swaps of elements.

Solution: C. When thinking of pairwise swaps, both selection and insertion sort come to mind. Selection sort does at most $\Theta(N)$ swaps, while it is possible for insertion sort to need $\Theta(N^2)$ swaps (for example, a reverse sorted array).

_____ Never compares the same two elements twice.

Solution: A, B, D. Notice for quicksort and merge sort that once we do a comparison between two elements (the pivot for quicksort and elements within a recursive subarray in merge sort), we will never compare those two elements again. For example, we won't compare the pivot against any other element again, and a sorted subarray will never have elements within it compared against one another. Insertion sort is much the same, as we bubble down elements into their respective places, comparing only against elements to the "left" of them.

_____ Runs in best case $\Theta(\log N)$ time for certain inputs.

F. The best case runtime for a sorting algorithm cannot be faster than $\Theta(N)$. This is because at the very least, we need to check if all elements are sorted, and since there are N elements, we can't have an algorithm that sorts faster than $\Theta(N)$.