CS 61B
Spring 2021

More Sorting

Discussion 13: April 19, 2021

1 Quicksort

(a) Sort the following unordered list using stable 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
```

```
-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 happen to result in better runtime?

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's 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, 3, 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 $1 + 2 + \cdots + N$.

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

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

2 Comparison Sorts Summary

(a) When choosing an appropriate algorithm, there are often several trade-offs that we need to consider. Complete the chart for the following sorting algorithms: give the expected time complexity in the worst case, in the best case, and whether or not each sort is stable.

	Time Complexity (Best)	Time Complexity (Worst)	Stability	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	Mostly

For all of these algorithms, note that there are many variants, and it's possible to find implementations with different stats than given above. For example, mergesort CAN be implemented in place, but its terrible and complicated, so in practice, we use a version that is not in place. Also note that most implementations of Quicksort use $O(log(n))$ additional space, which many people consider in-place. However, some people define "in place" to mean a constant amount of extra space.

(b) For each selection sort, give an example of a list where the order of equivalent items is not preserved.

In the following example, we only care about the number. The letter is to distinguish equal objects. We've given examples for heapsort and quicksort (our other unstable sorts) too, if you're interested!

```
Selection Sort: 3a, 3b, 3c, 1

[3a 3b 3c *1*]
1 [3b 3c 3a]
1 3b [3c 3a]
1 3b 3c [3a]

Heapsort: 1a, 1b, 1c

Quicksort: 3, 5a, 2, 5b, 1
[-3- *5a* 2 5b ˜1˜]
[-3- 1 2 5b 5a]
[-3- 1 *2* ˜5b˜ 5a]
[-3- 1 2 *˜5b˜* 5a]
[-3- 1 ˜2˜ *5b* 5a]
"L" and "R" pointers cross, swap pivot.
[1 2] 3 [5b 5a]
```

```
[-1- 2] 3 [-5b- 5a]
[-1- *~2~*] 3 [-5b- *~5a~*]
[-1- ~~ *2*] 3 [-5b- ~~ *5a*]
[-1-] [-2-] 3 [-5b-] [5a]
1 2 3 5b 5a
```

Note that if using Quicksort that randomizes the array, any array could yield instability.

(c) Notice that the worst-case runtime in the comparison sorts on an N element array listed above are lower bounded by $\Theta(N \log N)$. Can there be a sort that runs faster than $\Theta(N \log N)$ in the worst-case?

Yes, if we can avoid sorts that require comparisons, otherwise no. Given N elements, there are $N!$ possible permutations. Using a comparison sort, we will need at least $log_2(N!) \in \Omega(N \log N)$ comparisons. This is because one comparison could eliminate at most half of the possible permutations–when comparing two elements A and B, if you decide A should come first, then you've eliminated all the other permutations where B came first. However, with counting sorts, we can avoid the need for comparisons, and get a runtime that is linear with respect to the number of elements in the list, though its runtime is greatly dependent on other factors like radix and word size.

3 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	4309$\underline{2}$	3039$\underline{5}$	3031$\underline{5}$	3032$\underline{6}$
2	303$\underline{15}$	303$\underline{26}$	430$\underline{92}$	303$\underline{95}$
3	43$\underline{092}$	30$\underline{315}$	30$\underline{326}$	30$\underline{395}$
4	3$\underline{0315}$	3$\underline{0326}$	3$\underline{0395}$	4$\underline{3092}$
5	$\underline{30315}$	$\underline{30326}$	$\underline{30395}$	$\underline{43092}$

(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 three rounds are already filled for you.

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

	30395	30326	40392	30315
1	$\underline{3}$0395	$\underline{3}$0326	$\underline{3}$0315	| $\underline{4}$0392
2	$\underline{30}$395	$\underline{30}$326	$\underline{30}$315	| $\underline{4}$0392
3	$\underline{303}$95	$\underline{303}$26	$\underline{303}$15	| $\underline{4}$0392
4	| $\underline{3031}$5	| $\underline{3032}$6	| $\underline{3039}$5	| $\underline{4}$0392
5	| $\underline{30315}$	| $\underline{30326}$	| $\underline{30395}$	| $\underline{4}$0392

(c) Give the best case runtime, worst case runtime, and whether or not the sort is stable 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	$\Theta(W(N+R))$	$\Theta(W(N+R))$	Yes
MSD Radix Sort	$\Theta(N+R)$	$\Theta(W(N+R))$	Yes

(d) We just saw above that radix sort has great runtime with respect to the number of elements in the list. Given this fact, should 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.

4 Bounding Practice *Extra*

Given an array of n elements, the heapification operation permutes the elements of the array into a heap. There are many solutions to the heapification problem. One approach is bottom-up heapification, which treats the existing array as a heap and rearranges all nodes from the bottom up to satisfy the heap invariant. Another is top-down heapification, which starts with an empty heap and inserts all elements into it.

(a) Why can we say that any solution for heapification requires $\Omega(n)$ time?

In order to check that an array satisfies the heap invariant, we have to at least look at every element, which takes linear time.

(b) The worst-case runtime for top-down heapification is in $\Theta(n \log n)$. Why does this mean that the optimal solution for heapification takes $O(n \log n)$ time?

Since at least one solution for heapification takes $O(n \log n)$ time, the optimal solution for heapification takes $O(n \log n)$ time. Big O describes an upper bound on an operation, or in other words, the fastest rate at which it could possibly grow with respect to the input.

(c) In contrast, bottom-up heapification is an $O(n)$ algorithm. Is bottom-up heapfication asymptotically-optimal?

Since the running time of bottom-up heapify is $\Theta(n)$ and any solution for heapification requires $\Omega(n)$, bottom-up heapification is asymptotically optimal.

(d) Show that the worst-case runtime for top-down heapification is in $\Theta(n \log n)$.

For top-down heapification, where n elements are inserted into a Max Heap and subsequently popped off, the worst case is when a node needs to swim all the way up from the bottom at every element inserted.

Intuitively, it takes, at worst, $\log n$ work to insert a single element into a max heap, and we have n elements to insert, totalling to $n \log n$ work to create the heap. This logic alone is sufficient within the scope of 61B.

For a more mathematical explanation: inserting the first element into the 0^{th} level will require some work. Inserting the second (and third) element will require swimming up a level into the 1^{st} level, with 2 nodes at that level, results in a total of $(2^1 * (1))$ work on that level. Likewise, inserting the 4th (and 5th, 6th, and 7th) node requires swimming up two levels for a total work of $(2^2 * (2))$ work at the third level. At the ith level, there is a total work of $(2^i * i)$ In a heap with n elements, there are $\log n$ levels. The total work done is the summation of the work to insert all the nodes into a max heap where the insertion requires a node to swim from the bottom-most row to the top, such as inserting an array elements that are already in order (1,2,3,4,5...). Then, we get

$$\begin{aligned}
\sum_{i=0}^{\log_2(n)} i2^i &\leq \sum_{i=0}^{\log_2(n)} log_2(n)2^i \\
&= \log_2(n) \sum_{i=0}^{\log_2(n)} 2^i \\
&= \log_2(n) * n \texttt{ //note } \sum_{i=0}^{\log_2(n)} 2^i = 1 + 2 + 4 + ... + 2^{\log_2(n)} \in \Theta(n) \\
&= \Theta(n \log n)
\end{aligned}$$

(e) Show that the running time of bottom-up heapify is $\Theta(n)$. *This runtime derivation is definitely out of scope for this class–don't worry if you don't get the math!*

Some useful facts:

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

Taking the derivative:

$$\frac{d}{dx}(\sum_{i=0}^{\infty} x^i) = \frac{1}{(1-x)^2}$$

In bottom up heapification, we call `swimDown()` on every node, from bottom to top (or right to left in the underlying array). We want to sum the total work done in `swimDown()` at every node in the heap, so we can do this by summing the work done at each layer/level of the heap.

The number of levels in a heap is $log(n)$.

For each call to `swimDown()`, the number of work we'll do is proportional to how far that node would have to swim down. Nodes at the bottom of the heap don't have to move down at all, for example. So nodes at layer i would have to do i work, and the number of layers ranges from 0 to $log(n)$, where 0 is the leaf level and the root is at level $log(n)$.

How many nodes are at leach layer? Doing some quick math, we see that it is $2^{log(n)-i-1}$, where i is the layer number. Again layer i=0 is the leaves!

Putting this all together, we want to sum the amount of work done per node on a given layer (proportional to i) multiplied by the amount of nodes in that given layer (proportional to $2^{log(n)-i-1}$). We want to sum this all up for all layers, which means we should do a summation from 0 to $log(n)$, because that's the height of a heap!

Then the running time of heapify is:

$$\begin{aligned}
\sum_{i=0}^{\log n} i2^{log(n)-i-1} &= \sum_{i=0}^{\log n} i\frac{n}{2^{i+1}} \\
&= \frac{n}{4}\sum_{i=0}^{\log n} i(\frac{1}{2})^{i-1}
\end{aligned}$$

For ease of calculation, let's substitute $x = \frac{1}{2}$ and continue where we left off.

$$\begin{aligned}
\frac{n}{4}\sum_{i=0}^{\log n} ix^{i-1} &\leq \frac{n}{4}\sum_{i=0}^{\infty} i\,(x)^{i-1} \\
&= \frac{n}{4}\frac{d}{dx}\sum_{i=0}^{\infty} x^i \\
&= \frac{n}{4}\left(\frac{1}{(1-x)^2}\right) \\
&= \frac{n}{4}\left(\frac{1}{(1-1/2)^2}\right) \\
&= \Theta(n)
\end{aligned}$$

Essentially, the idea is just that each level roughly doubles the work, so the total runtime dependency on n is linear.