CS 61B

Discussion 11: Sorting

Fall 2021

1 Sorting: Mechanical Practice

Show the steps taken by each sort on the following unordered list:

```
106, 351, 214, 873, 615, 172, 333, 564
```

(a) Quicksort. After each partition during the algorithm, write the ordering of the list, circle the pivot that was used for that partition, and box the sub-array being partitioned. Assume that the pivot is always the first item in the sublist being sorted and that the array is sorted in place.

In the solution, we will **bold** the next pivot(s) to be used and $\boxed{\text{circle}}$ the current unsorted elements of the list.

$\boxed{\mathbf{106}\ 351\ 214\ 873\ 615\ 172\ 333\ 564}$

$106\ \boxed{\mathbf{351}\ 214\ 873\ 615\ 172\ 333\ 564}$

$106\ \boxed{\mathbf{214}\ 172\ 333}\ 351\ \boxed{\mathbf{873}\ 615\ 564}$

$106\ 172\ 214\ 333\ 351\ \boxed{\mathbf{615}\ 564}\ 873$

$106\ 172\ 214\ 333\ 351\ 564\ 615\ 873$

Note that depending on how the partition method is implemented, you may have different orderings of elements on either side of your pivot after a partition. The only property that must hold is that all elements less than the selected pivot go to the left, and all elements greater than the selected pivot go to the right.

(b) Merge sort. Show the intermediate merging steps.

$\boxed{106}\ \boxed{351}\ \boxed{214}\ \boxed{873}\ \boxed{615}\ \boxed{172}\ \boxed{333}\ \boxed{564}$

$\boxed{106\ 351}\ \boxed{214\ 873}\ \boxed{172\ 615}\ \boxed{333\ 564}$

$\boxed{106\ 214\ 351\ 873}\ \boxed{172\ 333\ 564\ 615}$

$\boxed{106\ 172\ 214\ 333\ 351\ 564\ 615\ 873}$

(c) LSD radix sort. Show the ordering of the list after each round of counting sort.

$106\ 351\ 214\ 873\ 615\ 172\ 333\ 564$

$\boxed{351}\ \boxed{172}\ \boxed{873\ 333}\ \boxed{214\ 564}\ \boxed{615}\ \boxed{106}$

$\boxed{106}\ \boxed{214\ 615}\ \boxed{333}\ \boxed{351}\ \boxed{564}\ \boxed{172\ 873}$

$\boxed{106\ 172}\ \boxed{214}\ \boxed{333\ 351}\ \boxed{564}\ \boxed{615}\ \boxed{873}$

2 Sorting: 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.
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
```

Quicksort, using the first element as a pivot.

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

Insertion sort. The first four elements are sorted.

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
```

MSD radix sort. We have sorted by the first digit so far.

(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
```

Merge sort. We have finished 2 levels of merging stages.

(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
```

Heapsort. The second line creates a max-heap.

3 Runtimes, Part 1: Comparison Sorts

Fill in the best and worst case runtimes of the following *comparison* sorting algorithms with respect to N, the length of the list being sorted.

	Worst case	Best case	Stable? (Yes/No)
Selection sort	$\Theta(N^2)$	$\Theta(N^2)$	No
Insertion sort	$\Theta(N^2)$	$\Theta(N)$	Yes
Merge sort	$\Theta(N \log N)$	$\Theta(N \log N)$	Yes
Quicksort	$\Theta(N^2)$	$\Theta(N \log N)$	No
Heapsort	$\Theta(N \log N)$	$\Theta(N)$	No

- **Selection sort**
  - **Worst case**: On the first iteration of the selection sort algorithm, selection looks through all N items to find the smallest one. On the next iteration, it looks through $N-1$ items to find the second-smallest item. The same logic applies for the rest of the sorting iterations. The runtime can therefore be determined by counting the total number of times an item was compared throughout the sorting process, calculated as follows:

$$N + (N-1) + (N-2) + \ldots + 3 + 2 + 1 = \sum_{i=1}^{N} i = \frac{N(N-1)}{2} \in \Theta(N^2)$$

    Therefore, the worst case runtime for selection sort is $\Theta(N^2)$.
  - **Best case**: The selection sort algorithm always goes through the same steps regardless of the state of the list/array being sorted (i.e. selection sort does not notice or care if the list is sorted/partially sorted). Because selection sort will perform the same number of comparisons on all lists, regardless of their state, the best case runtime is the same as the worst case runtime: $\Theta(N^2)$.
  - **Stable?** No. The swapping process in selection sort is not guaranteed to preserve the original ordering of equivalent items.

    For example, consider sorting the list `[2a, 2b, 1]` where `2a` and `2b` both have the value of 2. After running selection sort, we get the sorted order `[1, 2b, 2a]`. The original ordering of elements `2a` and `2b` was not preserved, proving that selection sort is **not** a stable sort.

- **Insertion sort**
  - **General case**: The runtime of insertion sort is $\Theta(N+K)$, where K is the total number of inversions (recall that an inversion is defined as *any pair* of elements that are not in the correct sorted order). We get the value N because we must look at all N items in the list in order to sort the list. We get the value K because we perform one pairwise swap for each inversion that we find in our array.
  - **Worst case**: We get a worst case scenario when we try to sort a list with the maximum possible number of inversions, i.e. fully reversed list. For each item at index i, we must shift its position to the left by i places. To get the worst case runtime we simply sum up the total number of swaps needed to sort items from index 0 through $n-1$. We can compute this as follows:

    $$0+1+2+\ldots+(N-1)=\sum_{i=1}^{N-1} i=\frac{(N-1)(N-2)}{2}\in\Theta(N^2)$$

    Note that we perform a swap each time we encounter an inversion in the array. Counting the number of swaps performed is therefore the same as counting the number of inversions in the array. Because we calculated that $\Theta(N^2)$ swaps are needed to sort the array, we can conclude that we have $K\in\Theta(N^2)$ inversions. By plugging in this result into our general equation for insertion sort runtime ($\Theta(N+K)$), we get a worst case runtime of $\Theta(N+N^2),\in\Theta(N^2)$.
  - **Best case**: If the list is almost sorted and has only $\Theta(n)$ inversions, then we only have to do $\Theta(n)$ swaps over all the items, giving us a best case runtime of $\Theta(n)$.
  - **Stable?** Yes. Insertion sort is a stable sort. During the selection sort process, we will only swap the ordering of any two items if the item on the right is less than the item to its left. Therefore, the ordering of two equivalent items will always be preserved in insertion sort.
- **Merge sort**
  - **Worst case**: At each level of our tree, we split the list into two halves, resulting in a tree with a height of $\log N$. At each one of the $\log N$ merge levels, we merge a total of N items together. As a result of performing N comparisons at $\log N$ levels, we get a worst case runtime of $\Theta(N\log N)$.

    Another approach is to sum up the amount of work done at each level of the call tree. Each level has 2^i nodes, and each node does $\frac{N}{2^i}$ amount of work from the linear-time merging operation. So that total amount of work is $\sum_{i=0}^{\log N} 2^i(\frac{N}{2^i})\in\Theta(N\log N)$.
  - **Best case**: The merge sort algorithm does the same amount of work regardless of the initial ordering of the list being sorted. We still have to do all of the comparisons between items regardless of their initial ordering, so the best case runtime is the same as the worst case runtime:($\Theta(N\log N)$).
  - **Stable?** Yes. During the merging process, we preserve the ordering of equivalent objects. If the first element of the left list being merged is equivalent to the first element

of the right list, we always insert the element from the left list before the right list. Through this process we always preserve the inital relative ordering of equivalent items in the list, resulting in a stable sorting algorithm.

- **Quicksort**
  - **Worst case**: In the quicksort algorithm, the work done at each iteration comes from moving the unsorted items to the correct side of the pivot. In the worst case we choose bad pivots, i.e. pivots that do not split the unsorted array into relatively even halves. In the worst case, we consistently choose partitions that are either the max or min of the unsorted items. When we choose bad partitions, our quicksort algorithm begins to look a bit like selection sort. For the first iteration, we move N items. For the second iteration, we shift $N-1$ items. For the third iteration, we shift $N-2$ items. For a list of length N we end up recursing N times. We sum up the total amount of work to get our worst case runtime as follows:

    $$N+(N-1)+(N-2)+...+2+1=\sum_{i=i}^{N} i \in \Theta(N^2)$$

    Therefore the worst case runtime for quicksort is $\Theta(N^2)$.

    If we end up always choosing the smallest item or largest item for our next pivot, each partition of a sub-array of size m will leave us with only one non-empty partition of size $m-1$, since every other item will be larger than or smaller than our pivot. With a total of n elements, we end up recursing n times, each time doing a linear time partition, for a total of $\Theta(n^2)$ time.
  - **Best case**: If we can always partition our sub-arrays roughly in half, our recursive call tree is identical to that of merge sort, resulting in a best case runtime of $\Theta(N \log N)$
  - **Stable?** For in-place implementations of quicksort, no. Through the process of moving items across the partition, there is no guarantee that equivalent items will stay in the same relative positioning as the initial unsorted list. For not in-place implementations of quicksort it is easier to implement stable quicksort. There are existing implementations of in-place quicksort that are stable, but those require extra logic and are therefore a bit more annoying to implement.

- **Heapsort**
  - **General case**: In-place heapification (bottom-up bubble down heapification) takes $O(N)$ time (naive in-place heapification, or top-bottom bubble up heapification, takes $\Theta(N \log N)$ time). After heapification, we simply remove from the max-heap. The runtime of heapsort is $\Theta(N+(timetoremoveallitems))$. The time it takes to remove all N items from the heap depends on the values in the heap, which differ in the worst and best case.
  - **Worst case**: As mentioned above, the runtime of heapsort is $\Theta(N+(timetoremoveallitems))$, where the $\Theta(N)$ comes from the in-place heapification procedure. In the worst case, each removal from the max-heap results in the new root being bubbled down all the

way down to the bottom level of the heap. As a result, each removal takes $\Theta(\log N)$ time from bubbling the new root from the top level to the $\log N$th level. The removal of N items, each taking $\Theta(\log N)$ time, takes $\Theta(N \log N)$ time. Therefore, the worst case runtime is $\Theta(N + (timetoremoveallitems))$, or $\Theta(N + N \log N) \in \Theta(N \log N)$.

– **Best case**: In the rare event where all the items in the list are the same, removing the maximum valued item takes $\Theta(1)$ time (we don't have to bubble the new root down whenever we remove an item). Performing N removals that each take $\Theta(1)$ time gives us $\Theta(N)$ time to remove all items from the heap. Therefore, the best case runtime for heapsort is $\Theta(N + N) \in \Theta(N)$.

– **Stable?** No. Through the insertion and removal process, with all the bubbling up and bubbling down there is no guarantee of stability.

4 Runtimes, Part 2: Counting Sorts

Fill in the best and worst case runtimes of the following *counting* sorting algorithms with respect to N, the length of the list being sorted. Assume we are sorting integers and L is the average number of digits in the integers being sorted. Let R be the size of the radix being used.

	Worst case	Best case	Stable? (Yes/No)
Distribution counting	$\Theta(N)$	$\Theta(N)$	Yes
LSD radix sort	$\Theta(NL)$	$\Theta(NL)$	Yes
MSD radix sort	$\Theta(NL)$	$\Theta(NL)$	Yes

- **Distribution counting**
  – **Worst case**: For the distribution counting algorithm, we take one linear pass through the list to calculate the indices for each group. Then we take another linear pass through the list to populate the final sorted list based off the indices calculated in the previous state. Therefore, the worst case runtime for distribution counting is $\Theta(N + N) \in \Theta(N)$.

  – **Best case**: Same as worst case. The runtime of distribution counting does not depend on the ordering of the list being sorted, yielding a best case runtime of $\Theta(N)$.

  – **Stable?** Yes! When we're populating the final sorted array, we process and add items in the order in which we encounter them, from index 0 to length-1. Because we're only adding equivalent items in the order in which they initially appeared, distribution counting is a stable sorting algorithm.

- **LSD Radix Sort**
  - **Worst Case**: There are N items, and each have approximately L digits. For each of these digits, we have to look through all N numbers and sort them by that digit. Since there are L digits and N integers, this gives us a runtime of $\Theta(NL)$.
  - **Best Case**: For LSD, you still have to look through all N items L times regardless of the input, resulting in a best case runtime of $\Theta(NL)$.
  - **Stable?** Recall that distribution counting is a stable sorting algorithm. Since LSD is just a result of running distribution counting across all the digits of the numbers being sorted, LSD radix sort is also a stable sorting algorithm. Note that LSD radix sort would not work if distribution counting was not a stable sorting algorithm.
- **MSD Radix Sort**
  - **Worst Case**: Same explanation as the worst case runtime for LSD radix sort. In the worst case we must perform distribution counting across all the digits of the N digits being sorted, giving us a worst case runtime of $\Theta(NL)$.
  - **Best Case**: MSD radix sort can short-circuit once each item is in its own bucket. If $N \leq R$ (R is the size of the radix), then it's possible to get each item in its own bucket after only 1 pass of distribution counting. This yields a runtime of $\Theta(N)$. However, this is not a strictly correct asymptotic runtime. As N grows towards infinity and becomes greater than R, it's not possible to put every item in its own bucket after looking at only the most significant digit. We need to look at at least $\log_R N$ digits to get all of our items in their own buckets and end MSD early in the best case. If $\log_R N$ is greater than L, the length of our items, we can't end MSD early and will simply go through the entire L iterations of distribution counting, yielding a best case runtime of $\Theta(NL)$.
  - **Stable?** Just like LSD sort, since MSD radix sort is just distribution counting applied across each digit of the numbers being sorted, MSD radix sort is also a stable sort.

5 Comparing Algorithms

(a) Give an example of a situation where using insertion sort is more efficient than using merge sort.

Insertion sort performs better than merge sort for lists that are already almost in sorted order (i.e. if the list has only a few elements out of place or if all elements are within k positions of their proper place and $k < \log N$, as this implies that there are less than $N \log N$ inversions).

(b) When might you decide to use radix sort over a comparison sort, and vice versa?

Radix sort gives us Nk and comparison sorts can be no faster than $N \log N$. When what we're trying to sort is bounded by a small k (such as short binary sequences), it might make more sense to run radix sort. Comparison sorts are more general-purpose, and are better when the items you're trying to sort don't make sense from a lexicographic perspective or can't be split up into individual "digits" on which you can run counting sort. However, if comparisons take a long time, radix sort might be a better option. Consider sorting many strings of very long length that are very similar. Using a comparison sort will take at least $N \log N$ comparisons, but each comparison may require us to iterate through entire strings, giving us a runtime of $NL \log N$, where L is the average length of our strings. Meanwhile, radix sort will give us a better runtime of NL. On the other hand, if our long strings are very dissimilar, our comparisons will take constant time because we can quickly determine that 2 strings are unequal. In this case, a comparison sort's runtime can be $N \log N$, which will likely be smaller than a radix sort's NL runtime.