Section 08: Solutions

Comparison Sorts

Show the steps taken for each sort as we have learned in lecture.

(a) Insertion sort on `0, 4, 2, 7, 6, 1, 3, 5`.

Solution:

```
0 | 4 2 7 6 1 3 5
0 4 | 2 7 6 1 3 5
0 2 4 | 7 6 1 3 5
0 2 4 7 | 6 1 3 5
0 2 4 6 7 | 1 3 5
0 1 2 4 6 7 | 3 5
0 1 2 3 4 6 7 | 5
0 1 2 3 4 5 6 7 |
```

(b) Selection sort on `0, 4, 2, 7, 6, 1, 3, 5`.

Solution:

```
0 | 4 2 7 6 1 3 5
0 1 | 2 7 6 4 3 5
0 1 2 | 7 6 4 3 5
0 1 2 3 | 6 4 7 5
0 1 2 3 4 | 6 7 5
0 1 2 3 4 5 | 7 6
0 1 2 3 4 5 6 | 7
0 1 2 3 4 5 6 7 |
```

(c) Heapsort on `0, 6, 2, 7, 4`. (You may want to draw out the heap. Make sure the first step you do is the heapification step!)

Solution:

```
7 6 2 0 4 (turns the array into a valid heap)
6 4 2 0 7 ('delete' 7, then sink 4)
4 0 2 6 7 ('delete' 6, then sink 0)
2 0 4 6 7 ('delete' 4, then sink 2)
0 2 4 6 7 ('delete' 2)
0 2 4 6 7 ('delete' 0)
```

(d) Merge sort on `0, 4, 2, 7, 6, 1, 3, 5`.

Solution:

$\boxed{\text{0 4 2 7 6 1 3 5}}$

$\boxed{\text{0 4 2 7}}\ \boxed{\text{6 1 3 5}}$

$\boxed{\text{0 4}}\ \boxed{\text{2 7}}\ \boxed{\text{6 1}}\ \boxed{\text{3 5}}$

$\boxed{0}\ \boxed{4}\ \boxed{2}\ \boxed{7}\ \boxed{6}\ \boxed{1}\ \boxed{3}\ \boxed{5}$

$\boxed{\text{0 4}}\ \boxed{\text{2 7}}\ \boxed{\text{1 6}}\ \boxed{\text{3 5}}$

$\boxed{\text{0 2 4 7}}\ \boxed{\text{1 3 5 6}}$

$\boxed{\text{0 1 2 3 4 5 6 7}}$

(e) Quicksort on `18, 7, 22, 34, 99, 18, 11, 4`. (Assume that we always choose first element as the pivot. Show the steps taken at each partitioning step.)

Solution:

$\boxed{\text{-18-, 7, 22, 34, 99, 18, 11, 4}}$

$\boxed{\text{-7-, 11, 4}}\ \boxed{\text{18}}\ \boxed{\text{18, 22, 34, 99}}$

$\boxed{4}\ \boxed{7}\ \boxed{11}\ \boxed{18}\ \boxed{\text{-18-, 22, 34, 99}}$

$\boxed{4}\ \boxed{7}\ \boxed{11}\ \boxed{18}\ \boxed{18}\ \boxed{\text{-22-, 34, 99}}$

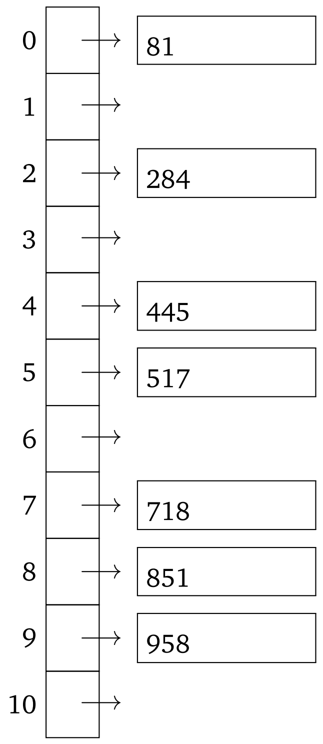
$\boxed{4}\ \boxed{7}\ \boxed{11}\ \boxed{18}\ \boxed{18}\ \boxed{22}\ \boxed{\text{34, 99}}$

Counting Sorts

(a) Show how the following list would look after one iteration of radix sort starting at the most significant digit with a radix of 10: [718, 445, 284, 517, 958, 851, 81]

Solution:

Using a radix of 10, the most decimal places required is 3. We will therefore look at the hundredths place of each number to sort the list. We can get a stable sort by putting the elements into a hash set with table size 10. We get the following table:



When we iterate through, the following order is achieved: [81, 284, 445, 517, 718, 851, 958]

(b) Now consider how the answer to the previous question would change if we used the least significant digit instead. Which of the following is the order after one iteration?

☐ $851, 81, 284, 445, 517, 958, 718$

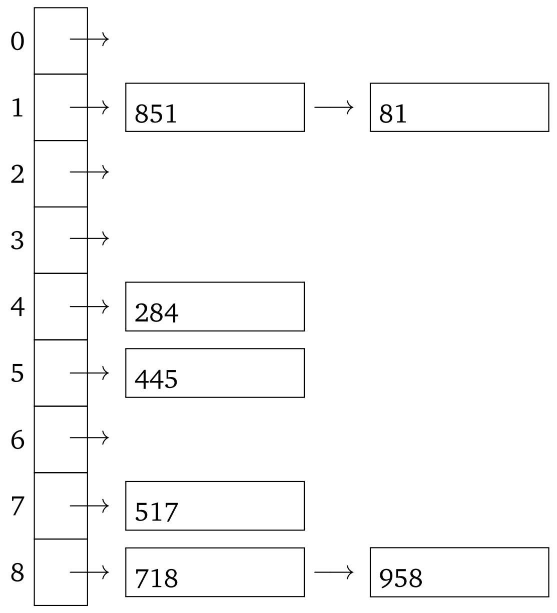
☐ $81, 851, 284, 445, 517, 718, 958$

☐ $851, 81, 284, 445, 517, 718, 958$

☐ $81, 851, 284, 445, 517, 958, 718$

Solution:

Using least significant digit, we would sort based on the ones place.



When we iterate through this table, we must remember to list numbers in the same bucket in the order they were added since radix sort requires the sort in each iteration to be stable. We will get the following list after one iteration of radix sort (LSD): [851, 81, 284, 445, 517, 718, 958]

(c) A friend proposes using radix sort to sort alphabetic strings. They would treat each string's characters as a base 26 digit and perform a radix sort using LSD. Would this algorithm correctly sort alphabetic strings? Why or why not?

Solution:

This algorithm would only work in some scenarios. The issue lies with how radix sort treats strings of different length. Radix sort on numbers of different length puts the number with fewer digits in front of the one with more (e.g. 30 is sorted before 101). When applying radix sort to strings, we do not want this to occur (e.g. ”bee” should be sorted after ”acorn”). To fix this, we can set all the strings to the same length by padding strings with fewer letters on the right then sorting and removing the padding.

(d) In order to speed up your radix sort algorithm, someone suggests choosing a hash table size as big as your largest value. Describe inputs where this would be advantageous and inputs where this is a disadvantage.

Solution:

On inputs with small values with dense distributions, this can be effective. For instance when sorting test scores, this can reduce the amount of iterations and potentially speed up the sort significantly.

On inputs where the values are large, this is not the right choice. For instance, if we are sorting students in 332 by their student id number, the size of the hash table would far exceed the size of the class. We would need to iterate over a lot of empty spaces before reaching any non-empty buckets.

These examples are two extremes, but the takeaway is that your choice of radix will have a significant effect on the runtime of counting sort algorithms.

Sorting Decisions

For each of the following scenarios, say which sorting algorithm you think you would use and why. As with the design decision problems, there may be more than one right answer.

(a) Suppose we have an array where we expect the majority of elements to be sorted "almost in order". What would be a good sorting algorithm to use?

Solution:

Merge sort and quick sort are always predictable standbys, but we may be able to get better results if we try using something like insertion sort, which is $\mathcal{O}(n)$ in the best case.

(b) You are writing code to run on the next Mars rover to sort the data gathered each night (Think about sorting with limited memory and computational power).

Solution:

Since each memory stick costs thousands (millions?) of dollars to send to Mars, an in-place sort is probably your best bet. Among in-place sorts, heap sort is a great choice (since it is guaranteed $\mathcal{O}(n \log n)$ time and doesn't even use much stack memory). Insertion sort meets memory needs, but wouldn't be fast.

(c) You're writing the backend for the website `SortMyNumbers.com`, which sorts numbers given by users.

Solution:

Do you trust your users? I wouldn't. Because of that, I want a worst-case $\mathcal{O}(n \log n)$ sort. Heap sort or Merge sort would be good choices.

(d) Your artist friend says for a piece she wants to make a computer sort every possible ordering of the numbers $1, 2, \ldots, 15$. Your friend says something special will happen after the last ordering is sorted, and you'd like to see that ASAP.

Solution:

Since you're going to sort all the possible lists, you want to optimize for the average case – Quick sort has the best average case behavior, which makes it a really good choice. Merge sort and heapsort also have average speed of $\mathcal{O}(n \log n)$ but they're usually a little slower on average (depending on the exact implementation).

She didn't appreciate your snarky suggestion to "just print $[1, 2, \ldots, 15]$ $15!$ times." Something about not accurately representing the human struggle.

(e) (1, 2, 3, 4, 5, 6, 7)

Solution:

Insertion sort has O(N) runtime if the input is completely sorted.

(f) A random generated integer data set with arbitrary size (could be really large). Ex. (65, 97, 85, 31, 72, 5, 98)

Solution:

If we know that the input is relatively large, (arbitrary size), the memory size should be under consideration. It is possible that the machine memory is not enough to make multiple copies of the data set, so a in-place sort will work better in this case.
There might be another case that one machine can't take over even one full data set. What would happen? Think about only take part of the data and sort it first. Then, merge it with the rest of the data set. That's merge sort.

(g) Kevin collects your QuickCheck worksheet and tries to sort it by names. What kind of sort would he use? Ex. (Erik, Louis, Velocity, Erika, Alieen, Brain)

Solution:

Since this is a "paper work" and the TAs might not want to print copies and waste papers, in-place sorting is preferable. An example is that Kevin find four TAs and split the papers in five groups. Each person sorts their stack and merge them back together.

Sorting Algorithm Analysis

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

Solution:

(i) Randomly choose pivots.

(ii) Shuffle the list before running Quicksort.

(b) When choosing an appropriate algorithm, there are often several trade-offs that we need to consider. Complete the chart for the following sorting algorithms by writing down the best case and worst case runtimes in $\Theta()$ notation and whether or not the sort is stable. You may write any notes about the sort in the "Notes" column; this column will not be graded.

Solution:

	Runtime (best)	Runtime (worst)	Stable? (Y/N)	Notes (not graded)
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	No	Same best case and worst case. Not stable if we use swapping method from lab, but there is a possibility of stability if we use an auxiliary array .
Insertion Sort	$\Theta(N)$	$\Theta(N^2)$	Yes	Best case is using a sorted array or an array of all duplicates. Worst case is using a reverse-sorted array. "Average" case is $\Theta(N^2)$. The runtime of insertion sort depends on the number of inversions; if we know the number of inversions is k, then the runtime of insertion sort is $\Theta(N+k)$ (the best case is when k is the smallest and is equal to 0 (sorted array or array of duplicates); the worst case is when k is the largest and is equal to $\frac{(N(N-1)}{2}$ (maximum number of inversions, reverse sorted array)).
Heapsort	$\Theta(N)$	$\Theta(N \log N)$	No	Best case is if the heap contains all duplicate items (every remove operation wouldn't require bubbling the heap).
Merge Sort	$\Theta(N \log N)$	$\Theta(N \log N)$	Yes	
Quicksort (Naive)	$\Theta(N \log N)$	$\Theta(N^2)$	Yes	Quicksort can also be implemented so it's unstable if Hoare's partitioning is used (no auxiliary arrays are created, and it is done "in-place").