Section 08: Solutions

1. Relaxation Zone

A timeless classic: `https://www.youtube.com/watch?v=YvTW7341kpA`

Solution:

Yes.

2. Mechanical 1: Sorting Algorithm Steps

Here is your input array: `32, 15, 2, 17, 19, 26, 41, 17, 17.`

Show the steps taken on this array for each sort as we have learned in lecture. Sort elements so that the result is ordered from smallest to largest. Also remark on whether each sort is **stable** or not.

(a) Insertion sort:

Solution:

See section slides.

(b) Selection sort:

Solution:

See section slides.

(c) **In-place** Heapsort (You may want to draw out the heap. Make sure the first step you do is the `buildHeap` step!):

Solution:

See section slides.

(d) Merge sort:

Solution:

See section slides.

(e) Quicksort (assume that we always choose the left-most element as the pivot):

Solution:

See section slides.

3. Mechanical 2: More Sorting Algorithm Steps

Show the steps taken for each sort as we have learned in lecture. Sort elements so that the result is ordered from smallest to largest.

(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) **In-place** Heapsort on `0, 6, 2, 7, 4`. (You may want to draw out the heap. Make sure the first step you do is the `buildHeap` 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:

```
[0 4 2 7 6 1 3 5]
[0 4 2 7] [6 1 3 5]
[0 4] [2 7] [6 1] [3 5]
[0] [4] [2] [7] [6] [1] [3] [5]
[0 4] [2 7] [1 6] [3 5]
[0 2 4 7] [1 3 5 6]
```

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

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

Solution:

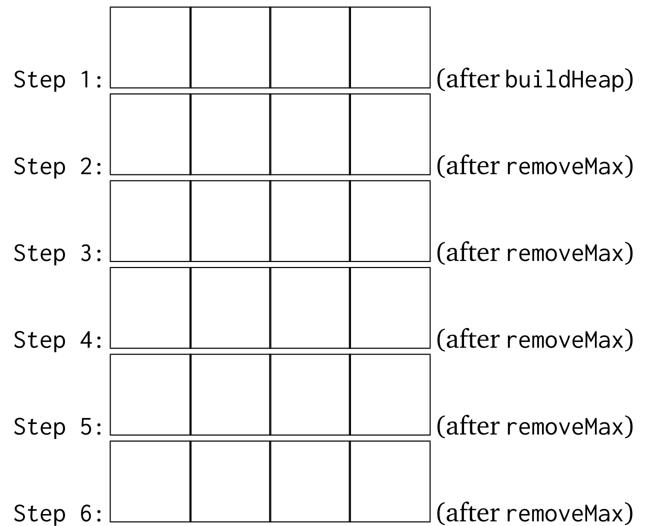
```
-18-, 7, 22, 34, 99, 18, 11, 4
-7-, 11, 4, 18, 18, 22, 34, 99
4, 7, 11, 18, -18-, 22, 34, 99
4, 7, 11, 18, 18, -22-, 34, 99
4, 7, 11, 18, 18, 22, 34, 99
```

4. Mechanical 3: More Heapsort

List out the steps of sorting the array [5, 0, 1, 3] into ascending order using **in-place** heap sort with a **max heap**.

The first step should be the array after the intial buildHeap, and each successive step should be the array after removeMax. You may not need every line provided to write your solution.

(Tip: Draw out the heap. Make sure your first step is buildHeap!)

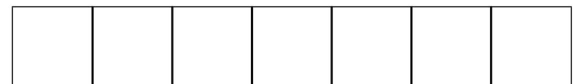


Solution:

5 - 3 - 1 - 0 : buildHeap turns the array into a valid heap
3 - 1 - 0 - 5 : removeMax(5), then percolate 0 down
1 - 0 - 3 - 5 : removeMax(3), then percolate 0 down
0 - 1 - 3 - 5 : removeMax(1)
0 - 1 - 3 - 5 : removeMax(0)

5. Runtime 1: Insertion Sort Worst-Case Input

Give the worst possible order of input for insertion sort with the following integers: 1, 2, 3, 4, 5, 6, 7. Assume that the result of sorting should be in ascending order.



Solution:

[7,6,5,4,3,2,1]

6. Runtime 2: All Worst-Case Inputs

When choosing an appropriate algorithm, there are often several trade-offs that we need to consider. Inspect the chart for the following sorting algorithms. Then, for each sort, provide an example of **both** a best-case and worst-case input.

	Runtime (best)	Runtime (worst)	Stable? (Y/N)	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	No	
Insertion Sort	$\Theta(N)$	$\Theta(N^2)$	Yes	
Heapsort	$\Theta(N)$	$\Theta(N \log N)$	No	
Merge Sort	$\Theta(N \log N)$	$\Theta(N \log N)$	Yes	
Quicksort	$\Theta(N \log N)$	$\Theta(N^2)$	No	

Solution:

See the **section slides** for interactive examples as well!

	Runtime (best)	Runtime (worst)	Stable? (Y/N)	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	No	Same best case and worst case. Not stable if we use swapping method from lecture, 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, which occurs with a 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 percolating the heap).
Merge Sort	$\Theta(N \log N)$	$\Theta(N \log N)$	Yes	
Quicksort	$\Theta(N \log N)$	$\Theta(N^2)$	No	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").

7. Runtime 3: Quicksort Pivots

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

Solution:

(a) Randomly choose pivots.

(b) Shuffle the list before running Quicksort.

8. The Legacy of Robbie: A Sorting Mystery

Consider the following sorting algorithm in pseudocode. Note that, in this case, the upper bound of each for loop is *inclusive*, so they run up to and including $i = A.\text{length} - 1$ and $j = i - 1$.

1: **function** MysterySort(A)
2:     **for** $i = 1$ to A.length $- 1$ **do**

3: for $j = 0$ to $i - 1$ **do**
4: **if** $A[j] \geq A[i]$ **then**
5: $x = A[i]$
6: shift every item from j to $i - 1$ right by one
7: $A[j] = x$
8: **break**

(a) Is MysterySort most similar to insertion sort, merge sort, quick sort, or selection sort?

Solution:

MysterySort is most similar to insertion sort. After every iteration of the i for-loop, a new item has been *inserted* somewhere in the sorted section of the array. Unlike selection sort, the new item could be inserted anywhere in the sorted section, not just at the end. This may involve shifting items over to the right to make room.

(b) Is MysterySort a stable sorting algorithm? Why or why not?

Solution:

MysterySort is unstable. This is due to a combination of reasons:

(i) The j for-loop goes over the array from left-to-right;

(ii) The if-condition that inserts the item does not use strict inequality.

This means MysterySort will insert each new item *before* the first item that is greater than or *equal* to it. Since each successive new item came *after* the last one in the original array, this reverses the order of equal items, which means MysterySort is unstable.

Note that insertion sort is normally a stable sorting algorithm. This could be considered a bug in MysterySort, since stability is a desirable property among sorting algorithms. You could make MysterySort stable by changing line 4 to check if $A[j] > A[i]$ instead of $A[j] \geq A[i]$.

(c) What is the best-case runtime (as a tight big-$\mathcal{O}$ bound) for MysterySort? Why is this the best case?

Hint: What happens when MysterySort is given an array that is already sorted?

Solution:

The best-case runtime is $\mathcal{O}(n^2)$.

Consider that for an array A that is already sorted, $A[i] \geq A[j]$ for all $i > j$. This means that the if-condition will never be true (ignoring equal items for now). The j for-loop will then always run i times, which is $\mathcal{O}(n^2)$.

For an array A that is sorted in *reverse* order, $A[i] \leq A[j]$ for all $i > j$. This means that the if-condition will always be true, so the loop runs only once. But when it runs, it shifts every item from j to $i - 1$ right by one; since $j = 0$, this takes i operations. So it is also $\mathcal{O}(n^2)$.

Any array will be some combination of these two extremes: the i for-loop will run for k iterations before shifting $i - k$ items over. Either way, it has to visit $k + (i - k) = i$ items every time, so it is always $\mathcal{O}(n^2)$.

Note that insertion sort normally runs in $\mathcal{O}(n)$ time in the best case, which is an already sorted array. You could fix MysterySort by reversing the j for-loop so it goes from $j = i - 1$ to 0, and changing the if-condition so it checks if $A[i] \geq A[j]$. This is always true if A is already sorted, so the loop runs once; but this time, it tries to shift over $(i - 1) - j = (i - 1) - (i - 1) = 0$ items! This takes constant time, so MysterySort will run in $\mathcal{O}(n)$.

9. Sorting Design Decisions 1

For each scenario below, fill in all squares with the letter corresponding to the sorting algorithm that would perform the best. Some of the questions may have multiple answers, in which case you should fill in multiple squares.

A: Insertion Sort, B: In-place Heapsort, C: Merge Sort, D: Selection Sort, E: Quicksort, F: None of the Above

(a) Java integer array with a length of 5. A ☐ B ☐ C ☐ D ☐ E ☐ F ☐

(b) An array of comparable UW classes (objects), that are already sorted by number and need to be additionally sorted by department. For example, the input `[CSE 142, CHEM 142, CSE 143, CSE 373]` would result in the output `[CHEM 142, CSE 142, CSE 143, CSE 373]`.

A ☐ B ☐ C ☐ D ☐ E ☐ F ☐

(c) We need to sort integers in $\Theta(\log N)$ time for the best case. A ☐ B ☐ C ☐ D ☐ E ☐ F ☐

(d) Sorting doubles with a $\Omega(N \log N)$ best-case lower bound. A ☐ B ☐ C ☐ D ☐ E ☐ F ☐

Solution:

(a) A. Insertion sort is fast for low N values

(b) A, C. Because the values are not primitives, we need a stable sort, so insertion sort and merge sort do the job!

(c) F. None of the above sorts can sort in $\Theta(\log N)$ time

(d) C, D, E. Merge sort, selection sort, and quicksort.

10. Sorting Design Decisions 2

For each of the following scenarios, say which sorting algorithm you think you would use and why. 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.