Section 3: Sort Algorithms

Stable vs. Unstable Sort Algorithms

- Affects duplicate values
- It does not matter for integers, but it can affect objects
- **NOTE:** A stable algorithm can be made unstable if the implementation is not correct, make sure to account for this

Unstable

- It is unstable if the relative ordering of the duplicate items is not preserved. The duplicates will swap, the order is not the same

Stable

- The order of the duplicate items is preserved, they will not be swapped

In-Place Algorithm

- Means that we are not creating a new array. We are just partitioning the original array into sorted and unsorted
- It is in-place if the extra memory that you need does not depend on the number of items you are sorting

Recursion

- A method is recursive when it calls itself
- There must be a case that will end the recursion, or else it will keep going

Bubble Sort

- We keep track of the end of the unsorted partition. We start left and compare adjacent elements, if the one before it is greater, then we swap them. We continue swapping until it reaches the end where we decrement the end of the unsorted partition and repeat
- **In-place algorithm**
- **Quadratic $O(n^2)$**
- **Stable**

https://www.youtube.com/watch?v=xli_FI7CuzA

Selection Sort

- Divides the array into sorted and unsorted partitions
- Find the largest element and save the index as you are traversing the array and comparing, at the end swap the largest with the last element in the array, decrease the end of the unsorted partition and repeat
- Does not require as much swapping as bubble sort does, only once per traversal
- **In-place algorithm**
- **Quadratic $O(n^2)$**
- **Unstable**

https://www.youtube.com/watch?v=g-PGLbMth_g

Insertion Sort

- Divides the array in partitions, but here it will build up the sorted partition. We compare the selected element at the index to each element in the sorted partition until we have found another element that is greater than it. Then at that point we will insert our chosen element before the element we have compared it to
- **In-place algorithm**
- **Quadratic $O(n^2)$**
- **Stable**

https://www.youtube.com/watch?v=JU767SDMDvA

Shell Sort

- Variation of insertion sort
- Insertion sort chooses which element to insert using a gap of 1, however shell sort starts out using a larger gap value
- As the algorithm runs the gap is reduced, the goal is to reduce the amount of shifting required
- The array is partially sorted; therefore, we are doing less shifting
- **In-place algorithm**
- **Quadratic $O(n^2)$** (can perform better)
- **Unstable**

https://www.youtube.com/watch?v=SHcPqUe2GZM

Merge Sort

- Divide and conquer algorithm, it involves splitting the array you want to sort, into a bunch of smaller arrays, it is implemented using recursion
- Splitting phase leads to faster sorting during the merging phase
- We don't create new arrays. We use indices to keep track of where the array has been split
- **Not in-place algorithm** (creating temporary arrays)
- **O(n * log(n))**
- **Stable**

https://www.youtube.com/watch?v=4VqmGXwpLqc

Splitting Phase

- Start with an unsorted array
- Divide the array into two arrays, which are unsorted. The first array is the left array, and the second is the right array
- Split the left and right arrays into two arrays each
- Keep splitting until all the arrays have only one element each – these arrays are sorted

Merging Phase

- Merge every left/right pair of sibling arrays into a sorted array
- After the first merge, we'll have a bunch of 2-element sorted arrays
- Then merge those sorted arrays (left/right siblings) to end up with a bunch of 4-element sorted arrays
- Repeat until you have a single sorted array

Merging Process

- We merge sibling left and right arrays
- We create a temporary array large enough to hold all the elements in the arrays we're merging
- We set i to the first index of the left array, and j to the first index of the right array
- We compare left[i] to right[j]. If left is smaller, we copy it to the temp array and increment i by 1. If right is smaller, we copy it to the temp array and increment j by 1
- We repeat this process until all elements in the two arrays have been processed
- At this point, the temporary array contains the merged values in sorted order
- We then copy this temporary array back to the original input array, at the correct positions
- If the left array is at position x to y, and the right array is at positions y + 1 to z, then after copy, positions x to z will be sorted in the original array

Quick Sort

- Divide and conquer algorithm
- Recursive
- Uses a pivot element to partition the array into two parts
- Elements < pivot to its left, elements > pivot to its right
- Pivot will then be in its correct sorted position
- Process is now repeated for the left array and right array
- Eventually, every element has been the pivot, so every element will be in its correct sorted position
- As with merge sort, we'll end up partitioning the array into a series of 1-element arrays
- **In-place algorithm**
- **O(n * log(n))**
- **Unstable**

https://www.youtube.com/watch?v=Hoixgm4-P4M

Counting Sort

- Makes assumptions about the data, so far, the previous algorithms did not
- Doesn't use comparisons
- Counts the number of occurrences of each value
- Only works with non-negative discrete values (can't work with floats, strings)
- Values must be within a specific range
- **Not in-place algorithm**
- **Linear O(n)**
- **Stable** (only if we add extra steps)

https://www.youtube.com/watch?v=7zuGmKfUt7s

Radix Sort

- Makes assumptions about the data
- Data must have the same radix and width
- Because of this, data must be integers or strings
- Sort based on each individual digit or letter position
- Start at the rightmost position
- Must use a stable sort algorithm at each stage, and counting sort is often used as the sort algorithm for radix sort
- **In-place algorithm**
- **Linear O(n)**
- **Stable**

https://www.youtube.com/watch?v=nu4gDuFabIM

Counting Sort (Stable)

- Requires extra steps
- Can calculate where values should be written back to the original array
- Writing the values into the array in backwards order ensures stability
- It works because we traverse the input array from right to left, and we write duplicate values into the temp array from right to left
- If we know that duplicate values will go into positions 3 and 4, we write the rightmost value in the input array into position 4, and the leftmost value into position 3
- This preserves the relative positioning of duplicate values
- By adjusting the counting array after the initial pass, we can map values to indices in the temp array
- Can also use linked lists to make counting sort stable

Java JDK Sort Methods

static void	**sort**(byte[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(byte[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(char[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(char[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(double[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(double[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(float[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(float[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(int[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(int[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(long[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(long[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(Object[] a) Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.
static void	**sort**(Object[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements.
static void	**sort**(short[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(short[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static <T> void	**sort**(T[] a, Comparator<? super T> c) Sorts the specified array of objects according to the order induced by the specified comparator.
static <T> void	**sort**(T[] a, int fromIndex, int toIndex, Comparator<? super T> c) Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.