**Fundamentals of Sorting:**

The fundamental problem of sorting is all about ordering a collection of items. How you order these items is entirely based on the method of comparison. Suppose you needed to sort a pile of books. If you are working on a home library, you might organize it by the author’s last name. But if you need to quickly transport the books, it might make sense to initially organize them based on the size of the book. Both of these problems are sorting problems, but a key takeaway is that sorting problems are necessarily tied to a method of comparison. Different methods of comparison may lead to different results. At the most basic level, sorting algorithms are all about rearranging elements in a collection based on a common characteristic of those elements.

In computer science, we have formal definitions of sorting with respect to ordering relations.

An **ordering relation** has two key properties: 1. Given two elements a and b, exactly one of the following must be true: *a*<*b*, *a*=*b*, or *a*>*b* ([**Law of Trichotomy**](https://en.wikipedia.org/wiki/Trichotomy_(mathematics))) 2. If *a*<*b* and *b*<*c*, then *a*<*c* ([**Law of Transitivity**](https://en.wikipedia.org/wiki/Transitive_relation))

A **sort** is formally defined as a rearrangement of a sequence of elements that puts all elements into a non-decreasing order based on the ordering relation.

Suppose you were given a list of strings [“hello”, “world”, “we”, “are”, “learning, “sorting”][“hello”, “world”, “we”, “are”, “learning, “sorting”]. One way to define an ordering relation might be based on the length of the string. One valid sort based on this ordering relation is [“we”, “are”, “hello”, “world”, “sorting”, “learning”][“we”, “are”, “hello”, “world”, “sorting”, “learning”]. For every pair of adjacent elements in the list, the length of the preceding string is always less than or equal to the length of the following string. Another ordering relation we could define is the number of vowels in the string. That would lead to the following sort: [“we”, “world”, “are”, “hello”, “sorting”, “learning”][“we”, “world”, “are”, “hello”, “sorting”, “learning”].

The ordering relation practically is defined as a method of comparison in programming languages. Most programming languages allow you to pass in custom functions for comparison whenever you want to sort a sequence of elements. In Java, for example, these are comparators. In Python, you can pass a comparison function as the key to the sort method.

import java.util.Arrays;

public class Solution {

public void sortByLength(String[] arr) {

// Sorts a list of string by length of each string

Arrays.sort(array, new StringCompare());

}

}

public class StringCompare implements Comparator<String> {

public int compare(String s1, String s2) {

if (s1.length() > s2.length()) {

return 1;

} else if (s1.length() < s2.length()) {

return -1;

}

return 0;

}

}

------------------Python3----------------------------------------------------------------------

class Solution:

def sort\_by\_length(self, lst: List[str]) -> None:

"""

Sorts a list of strings by the length of each string

"""

lst.sort(key=lambda x: len(x)) # Note we can also do lst.sort(key=len)

An important concept in sorting is **inversions**. An inversion in a sequence is defined as a pair of elements that are out of order with respect to the ordering relation. To understand this idea better, let's consider our earlier string example, where the ordering relation was defined by the length of the string:

[“are”, “we”, “sorting”, “hello”, “world”, “learning”][“are”, “we”, “sorting”, “hello”, “world”, “learning”]

Clearly, the above list is not sorted according to the lengths of strings, but what if you had to define a metric for how “out of sort” it was? Inversions provide a way to define that. In the above unsorted list, we have the following inversions:

(“are”, “we”)(“are”, “we”), (“sorting”, “hello”)(“sorting”, “hello”), and (“sorting”, “world”)(“sorting”, “world”)

The more inversions present, the more out of order the list is. In fact, the concept of inversions introduces an alternative definition of sorting: Given a sequence of elements with n inversions, a **sorting algorithm is a sequence of operations that reduces inversions to 0**.

The next important concept in sorting that we will refer back to is the **stability** of sorting algorithms. The key feature of a stable sorting algorithm is that it will preserve the order of equal elements. In our earlier string example with the string length ordering comparison, our original sequence was [“hello”, “world”, “we”, “are”, “learning, “sorting”][“hello”, “world”, “we”, “are”, “learning, “sorting”]

There are two valid sorts for this sequence:

1. [“we”, “are”, “hello”, “world”, “sorting”, “learning”][“we”, “are”, “hello”, “world”, “sorting”, “learning”]
2. [“we”, “are”, “world”, “hello”, “sorting”, “learning”][“we”, “are”, “world”, “hello”, “sorting”, “learning”]

We consider (1) to be a stable sort since the equal elements “hello” and “world” are kept in the same relative order as the original sequence.

Give the following array of strings ['hello', 'your', 'above', 'year', 'alone', 'friendly', 'crazy'] where the ordering relation is the length, select which sort is a stable sort?

['your', 'year', 'hello', 'crazy', 'above', 'alone', 'friendly']

['your', 'year', 'alone', 'hello', 'crazy', 'above', 'friendly']

['your', 'year', 'above', 'alone', 'crazy', 'hello', 'friendly']

['your', 'year', 'hello', 'above', 'alone', 'crazy', 'friendly']

How many inversions exist in the following list of integers: [3, 4, 6, 5, 2]

2

4

5

6

Which of the following are the key parts of an ordering relation? (Select all that apply)

If a < b and b < c, it must be true that a < c

There must exist an ordering of three elements such that a < b < c

It must be true that a < b or a = b or a > b

Removing an element from a collection affects the ordering relation between elements

**Comparison Based Sort**

Comparison based sorts are sorting algorithms that require a direct method of comparison defined by the ordering relation. In a sense, they are the most natural sorting algorithms since, intuitively, when we think about sorting elements, we instinctively think about comparing elements to each other. In the following sections, we’ll introduce some of the fundamental comparison based sorting algorithms.

Note that merge sort and quick sort are intentionally omitted from this Explore Card. In the future, these sorting algorithms will receive their own explore cards which will introduce each algorithm and related techniques that use the algorithm.

**Selection Sort**

Suppose you had to sort a pile of books by their weight, with the heaviest book on the bottom and the lightest book on the top. One reasonable method of sorting is to go through your books, find the heaviest book, and then place that at the bottom. After that, you can then find the next heaviest book in the remaining pile of books and place that on top of the heaviest book. You can continue this approach until you have a sorted pile of books. This concept is exactly what the selection sort does.

Suppose we had a collection of elements where every element is an integer. Selection sort will build up the sorted list by repeatedly finding the minimum element in that list and moving it to the front of the list through a swap. It will proceed to swap elements appropriately until the entire list is sorted.

In terms of simplicity, it is a highly intuitive algorithm and not too difficult to write. Unfortunately, it is pretty slow, requiring *O*(*n*2) time to sort the list in the worst case. In the worst case, we have to search the entire array to find the minimum element, meaning we can have up to n+(n−1)+(n−2)+…+1*n*+(*n*−1)+(*n*−2)+…+1 total operations, which is *O*(*n*2). The space complexity of selection sort is *O*(1) since we do not use any additional space during the algorithm (all operations are in-place).

It also is **not a stable** sorting algorithm. For example consider the collection [**4**, 2, 3, 4, 1]. After the first round of selection sort, we get the array [1, 2, 3, 4, **4**]. This array is sorted, but it does not preserve the ordering of equal elements.

public class Solution {

public void selectionSort(int[] arr) {

// Mutates arr so that it is sorted via selecting the minimum element and

// swapping it with the corresponding index

int min\_index;

for (int i = 0; i < arr.length; i++) {

min\_index = i;

for (int j = i + 1; j < arr.length; j++) {

if (arr[j] < arr[min\_index]) {

min\_index = j;

}

}

// Swap current index with minimum element in rest of list

int temp = arr[min\_index];

arr[min\_index] = arr[i];

arr[i] = temp;

}

}

}

--------------------------------------------------------------Python3-----------------------------------------------------------------

class Solution:

def selection\_sort(self, lst: List[int]) -> None:

"""

Mutates lst so that it is sorted via selecting the minimum element and

swapping it with the corresponding index

"""

for i in range(len(lst)):

min\_index = i

for j in range(i + 1, len(lst)):

# Update minimum index

if lst[j] < lst[min\_index]:

min\_index = j

# Swap current index with minimum element in rest of list

lst[min\_index], lst[i] = lst[i], lst[min\_index]

**Bubble Sort**

Conceptually, bubble sort is an implementation of a rather simple idea. Suppose we have a collection of integers that we want to sort in ascending order. Bubble sort proceeds to consider two adjacent elements at a time. If these two adjacent elements are out of or the (in this case, the left element is strictly greater than the right element), bubble sort will swap them. It then proceeds to the next pair of adjacent elements. In the first pass of bubble sort, it will process every set of adjacent elements in the collection once, making swaps as necessary. The core idea of ​​bubble sort is it will repeat this process until no more swaps are made in a single pass, which means the list is sorted.

In terms of the running time of the algorithm, bubble sort’s runtime is entirely based on the number of passes it must make in the array until it’s sorted. If the array has *n* elements, each pass will consider (*n*−1) pairs. In the worst case, when the minimum element is at the end of the list, it will take (*n*−1) passes to get it to the proper place at the front of the list, and then one more additional pass to determine that no more swaps are needed. Bubble sort, as a result, has worst case runtime of *O*(*n*2). The space complexity of bubble sort is *O*(1). All sorting operations involve swapping adjacent elements in the original input array, so no additional space is required.

Bubble sort is also a **stable** sorting algorithm since equal elements will never have swapped places, so their relative ordering will be preserved.

Overall, bubble sort is fairly simple to implement, and it's stable, but outside of that, this algorithm does not have many desirable features. It's fairly slow for most inputs and, as a result, it is rarely used in practice.

Below is the implementation of bubble sort:

**public class Solution {**

**public void bubbleSort(int[] arr) {**

**// Mutates arr so that it is sorted via swapping adjacent elements until**

**// the arr is sorted.**

**boolean hasSwapped = true;**

**while (hasSwapped) {**

**hasSwapped = false;**

**for (int i = 0; i < arr.length - 1; i++) {**

**if (arr[i] > arr[i + 1]) {**

**// Swap adjacent elements**

**int temp = arr[i];**

**arr[i] = arr[i + 1];**

**arr[i + 1] = temp;**

**hasSwapped = true;**

**}**

**}**

**}**

**}**

**}**

**class Solution:**

**def bubble\_sort(self, lst: List[int]) -> None:**

**"""**

**Mutates lst so that it is sorted via swapping adjacent elements until**

**the entire lst is sorted.**

**"""**

**has\_swapped = True**

**# if no swap occurred, lst is sorted**

**while has\_swapped:**

**has\_swapped = False**

**for i in range(len(lst) - 1):**

**if lst[i] > lst[i + 1]:**

**# Swap adjacent elements**

**lst[i], lst[i + 1] = lst[i + 1], lst[i]**

**has\_swapped = True**

**Insertion Sort**

Going back to our pile of books analogy, where we attempted to sort by weight, let's explore another approach to sorting the pile of books. We'll start at the top of the pile and iterate over the books one by one. Every time we encounter a book that is lighter than the book above it, we'll move the book up until it is in its appropriate place. Repeating this for the entire pile of books, we will get the books in sorted order.

This is the core intuition behind insertion sort. Given a collection of integers, you can sort the list by proceeding from the start of the list, and every time you encounter an element that is out of order, you can continuously swap places with previous elements until it is inserted in its correct relative location based on what you’ve processed thus far. This process is best understood with a visual example.

In terms of efficiency of this approach, the worst possible input is a reversed list, where every element has to be inserted at the very beginning of the list, which leads to a total of 1+2+…+(*n*−1) or *O*(*n*2) swaps. The space complexity of insertion sort is *O*(1). All operations are performed in-place.

Despite the *O*(*n*2) time complexity, in practice, there are a couple of advantages to insertion sort.

For one, it is a **stable sort**. By design of its implementation, we will never swap an element later in the list with an equal element earlier in the list. But more importantly, there are cases where insertion sort may actually be the best sort.

Generally, on almost sorted arrays where the number of inversions is relatively small compared to the size of the array, insertion sort will be quite fast since the number of swaps required will be low on almost sorted arrays.

Next, insertion sort can also be the best choice on small arrays. This is more of an empirical observation based on experiments, but it is one that you should be aware of. Many sorting functions have a quick check for the size of the collection and if that value is below a threshold, the program will default to insertion sort. Java's [official implementation of Arrays.sort()](http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/be44bff34df4/src/share/classes/java/util/Arrays.java#l1345) performs such a check before performing more theoretically optimal sorts.

In terms of disadvantages, on larger collections with many inversions, other sorts will generally outperform insertion sort. However, of all the sorts we have covered thus far, insertion sort is the first that is practically used, depending on the context.

Below is the implementation of insertion sort:

**public class Solution {**

**public void insertionSort(int[] arr) {**

**// Mutates elements in arr by inserting out of place elements into appropriate**

**// index repeatedly until arr is sorted**

**for (int i = 1; i < arr.length; i++) {**

**int currentIndex = i;**

**while (currentIndex > 0 && arr[currentIndex - 1] > arr[currentIndex]) {**

**// Swap elements that are out of order**

**int temp = arr[currentIndex];**

**arr[currentIndex] = arr[currentIndex - 1];**

**arr[currentIndex - 1] = temp;**

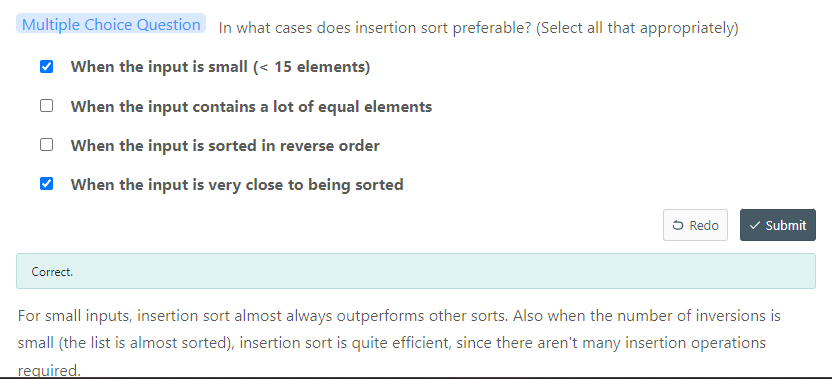
**currentIndex -= 1;**

**}**

**}**

**}**

}

****

Given the head of a singly linked list, sort the list using **insertion sort**, and return *the sorted list's head*.

The steps of the **insertion sort** algorithm:

1. Insertion sort iterates, consuming one input element each repetition and growing a sorted output list.
2. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list and inserts it there.
3. It repeats until no input elements remain.

The following is a graphical example of the insertion sort algorithm. The partially sorted list (black) initially contains only the first element in the list. One element (red) is removed from the input data and inserted in-place into the sorted list with each iteration.

**Heap Sort**[Report Issue](https://github.com/LeetCode-Feedback/LeetCode-Feedback/issues)

When we discussed selection sort, the basic principle involved finding the minimum element and moving it to the front. We repeated this continuously until we sorted the entire list. But as we saw, selection sort has a running time of �(�2)*O*(*n*2), since for every iteration, we need to find the minimum element in the list which takes �(�)*O*(*n*) time. We can improve upon this by using a special data structure called a heap.

To review the basics of the heap data structure, you can visit the [Heap Explore Card](https://leetcode.com/explore/featured/card/heap/). The core concept of the heap sort involves constructing a heap from our input and repeatedly removing the minimum/maximum element to sort the array. A naive approach to heapsort would start with creating a new array and adding elements one by one into the new array. As with previous sorting algorithms, this sorting algorithm can also be performed in place, so no extra memory is used in terms of space complexity.

The key idea for in-place heapsort involves a balance of two central ideas:  
(a) Building a heap from an unsorted array through a “bottom-up heapification” process, and  
(b) Using the heap to sort the input array.

Heapsort traditionally uses a max-heap to sort the array, although a min-heap also works, but its implementation is a little less elegant.

Algorithm for “bottom-up heapification” of input into max-heap. Given an input array, we can represent it as a binary tree. If the parent node is stored at index i, the left child will be stored at index 2i + 1 and the right child at index 2i + 2 (assuming the indexing starts at 0).  
To convert it to a max-heap, we proceed with the following steps:

1. Start from the end of the array (bottom of the binary tree).
2. There are two cases for a node
   * It is greater than its left child and right child (if any).
     + In this case, proceed to next node (one index before current array index)
   * There exists a child node that is greater than the current node
     + In this case, swap the current node with the child node. This fixes a violation of the max-heap property
     + Repeat the process with the node until the max-heap property is no longer violated
3. Repeat step 2 on every node in the binary tree from bottom-up.

A key property of this method is that by processing the nodes from the bottom-up, once we are at a specific node in our heap, it is guaranteed that all child nodes are also heaps. Once we have “heapified” the input, we can begin using the max-heap to sort the list. To do so, we will:

1. Take the maximum element at index 0 (we know this is the maximum element because of the max-heap property) and swap it with the last element in the array (this element's proper place).
2. We now have sorted an element (the last element). We can now ignore this element and decrease heap size by 1, thereby omitting the max element from the heap while keeping it in the array.
3. Treat the remaining elements as a new heap. There are two cases:
   * The root element violates the max-heap property
     + Sink this node into the heap until it no longer violates the max-heap property. Here the concept of "sinking" a node refers to swapping the node with one its children until the heap property is no longer violated.
   * The root element does not violate the max-heap property
     + Proceed to step (4)
4. Repeat step 1 on the remaining unsorted elements. Continue until all elements are sorted.

The key aspect that makes heapsort better than selection sort is the running time of the algorithm is now *O*(*N*log*N*). This is a result of the fact that removing the max element from the heap, which is the central operation in the sort is a *O*(log*N*) operation, which has to be performed in the worst case *N*−1 times. Note that in-place heapification is an *O*(*N*) [operation](https://stackoverflow.com/questions/9755721/how-can-building-a-heap-be-on-time-complexity), so it has no impact on the worst-case time complexity of heapsort.

In terms of space complexity, since we are treating the input array as a heap and creating no extra space (all operations are in-place), heapsort is *O*(1).

**public class Solution {**

**public void heapSort(int[] arr) {**

**// Mutates elements in lst by utilizing the heap data structure**

**for (int i = arr.length / 2 - 1; i >= 0; i--) {**

**maxHeapify(arr, arr.length, i);**

**}**

**for (int i = arr.length - 1; i > 0; i--) {**

**// swap last element with first element**

**int temp = arr[i];**

**arr[i] = arr[0];**

**arr[0] = temp;**

**// note that we reduce the heap size by 1 every iteration**

**maxHeapify(arr, i, 0);**

**}**

**}**

**private void maxHeapify(int[] arr, int heapSize, int index) {**

**int left = 2 \* index + 1;**

**int right = 2 \* index + 2;**

**int largest = index;**

**if (left < heapSize && arr[left] > arr[largest]) {**

**largest = left;**

**}**

**if (right < heapSize && arr[right] > arr[largest]) {**

**largest = right;**

**}**

**if (largest != index) {**

**int temp = arr[index];**

**arr[index] = arr[largest];**

**arr[largest] = temp;**

**maxHeapify(arr, heapSize, largest);**

**}**

**}**

**}**

The main advantage of heapsort is it's generally much faster than the other comparison based sorts on sufficiently large inputs as a consequence of the running time. However, there are a few undesirable qualities in the algorithm. For one, it is **not a stable** sort. It also turns out that in practice, this algorithm performs worse than other *O*(*N*log*N*) sorts as a result of bad [cache locality](https://en.wikipedia.org/wiki/Locality_of_reference) properties. Heapsort swaps elements based on locations in heaps, which can cause many read operations to access indices in a seemingly random order, causing many cache misses, which will result in practical performance hits.

**Quiz!**

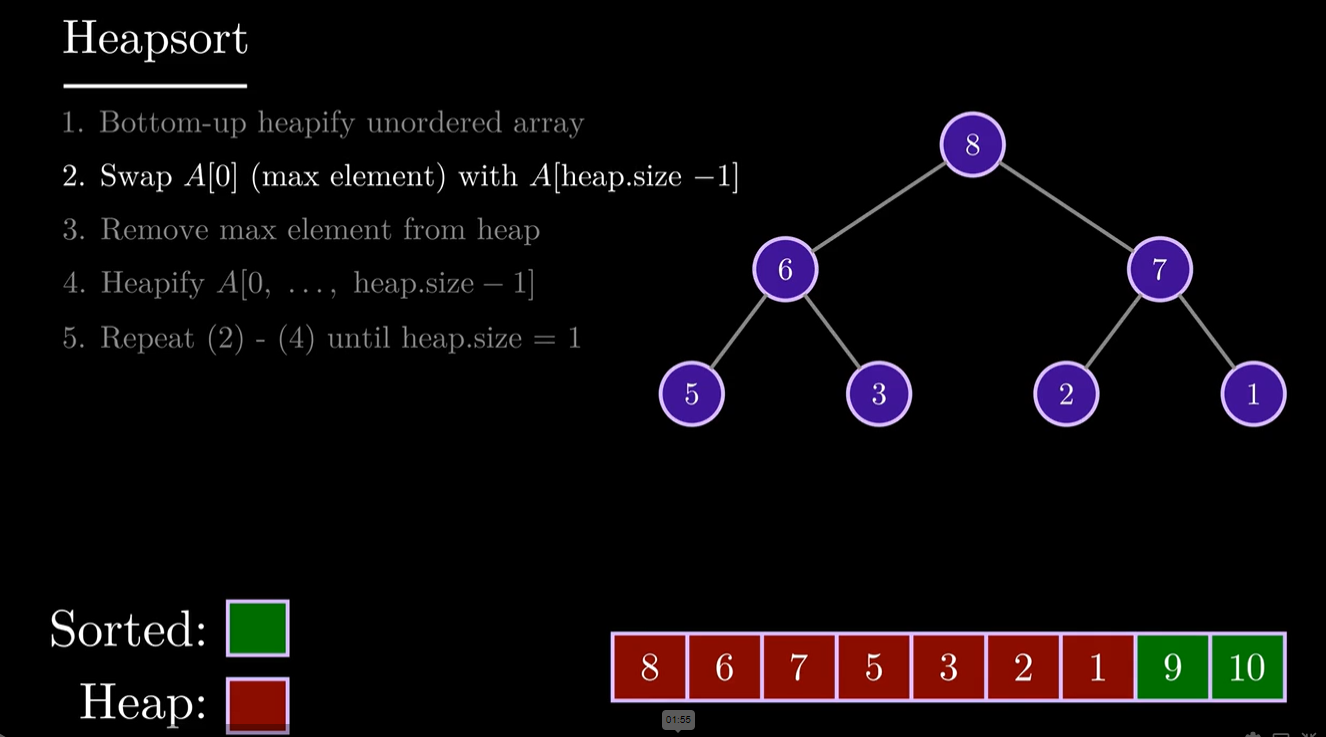
Multiple Choice Question

Which sort is heapsort a direct optimization of?

Insertion Sort

Bubble Sort

Selection Sort

**Counting Sort**

In the world of non-comparison based sorts, one of the simplest building blocks is counting sort.

Let’s start with a simple example and build up to the full counting sort algorithm. Let’s define an array *A*=[1,5,0,3,6,4,2]

Could you come up with a clever approach to sort this special array in a single pass?

A special property of this array is that the maximum element is 6, the minimum element is 0, and each value in between shows up exactly once in the array. So, for this specific array, a rather simple algorithm can sort it in one pass:

1. Initialize an array output of size 7
2. For every element *A*[*i*]
   * output [*A*[*i*]]=*A*[*i*]

All we have to do is map each element*A*[*i*] to index *A*[*i*] in the output array. This algorithm runs in O(N) time and O(N) extra space. It is guaranteed to work for any array with the following properties: 1. Each element in the array *A* is between 00 and *N*−1 inclusive (0≤ *A*[*i*] ≤ *N*−1) 2. No element is repeated 3. The array is size N (all elements from 0 to N - 1 show up exactly once)

The above algorithm in a sense is the simplest version of counting sort. Counting sort is all about using a predefined range of “keys” (in the above example, the keys map one-to-one to an index) to construct a sorted input. The above example is the basic idea, but there are some natural extensions to it. 1. Counting sort can handle non-unique keys (input array can have duplicate elements) 2. Counting sort can handle non-consecutive keys (input array can have elements that don’t exist within the predefined range of values) 3. Counting sort can handle non-numerical keys as long as they are constrained within an alphabet of constrained size (e.g characters, objects with a predefined set of values)

Suppose now that the minimum possible value of the array is set to 0 and the maximum possible value in the array is K. The main idea required to handle steps (1) and (2) is to track the frequency of each value in the range 0 to K.

Suppose the input array was *A*=[5,4,5,5,1,1,3]

As we did earlier, we can initialize a new array counts of size equal to the max(*A*)+1. Then, the core concept involves mapping each index *i* of the counts array to the number of occurrences of *i* in the original array *A*.

In this example, the counts array would be  [0, 2, 0, 1, 1, 3, 0]

From this counts array, we can determine the starting index for each element in the original array. The starting indices can be found by calculating the cumulative sum of our previous counts array (index *i* is the sum of all preceding indices).  startingIndices=[0,0,2,2,3,4,7]

An easy interpretation of this array is to take any element in the original input *A* and find its index in startingIndicesstartingIndices. That index is where the first instance of that element should be placed. Couple of examples below:

1. The first instance of element 1 should be placed at index 0 (which makes sense because it is the minimum element).
2. The first instance of element 5 should be placed at index 4. This must be true since, when sorted, there are three total instances of 5's, so the starting index of the first instance of 5 should be at index 4. On every iteration, we will increment the index as we process elements. So the second and third instances of element 5 will be placed at indices 5 and 6, respectively.

Below is an animation of counting sort:

****

In the actual implementation of this algorithm, we will overwrite the original counts array with the startingIndicesstartingIndices array to reduce the amount of additional space used.

With this intuition in hand, below is the algorithm for counting sort on a set of integers from 0 to K (not all values have to be present and some values can be duplicated).

**public class Solution {**

**public void countingSort(int[] arr) {**

**// Sorts an array of integers where minimum value is 0 and maximum value is K**

**int K = Arrays.stream(arr).max().getAsInt();**

**int[] counts = new int[K + 1];**

**for (int elem : arr) {**

**counts[elem] += 1;**

**}**

**// we now overwrite our original counts with the starting index**

**// of each element in the final sorted array**

**int startingIndex = 0;**

**for (int i = 0; i < K + 1; i++) {**

**int count = counts[i];**

**counts[i] = startingIndex;**

**startingIndex += count;**

**}**

**int sortedArray[] = new int[arr.length];**

**for (int elem : arr) {**

**sortedArray[counts[elem]] = elem;**

**// since we have placed an item in index counts[elem], we need to**

**// increment counts[elem] index by 1 so the next duplicate element**

**// is placed in appropriate index**

**counts[elem] += 1;**

**}**

**// common practice to copy over sorted list into original arr**

**// it's fine to just return the sortedArray at this point as well**

**for (int i = 0; i < arr.length; i++) {**

**arr[i] = sortedArray[i];**

**}**

**}**

**}**

The time complexity of counting sort is *O*(*N*+*K*) where *N* is the size of the input array and *K* is the maximum value in the array. Again, a key assumption in the above version of counting sort is that the minimum possible value in the array is 0 (no negative numbers) and the maximum value is some positive integer K. If this is not the case, it's possible to perform a mapping step at the beginning and then remap the values to the original array at the end. For example, an array with values between -5 and 10 can be mapped to values between 0 and 15, perform counting sort, and then remap to the original -5 to 10 range.

Below is a slight modification to counting sort to handle shifting of values when values are between a range of two general integers.

**import java.util.Arrays;**

**public class Solution {**

**public void countingSort(int[] arr) {**

**// Sorts an array of integers (handles shifting of integers to range 0 to K)**

**int shift = Arrays.stream(arr).min().getAsInt();**

**int K = Arrays.stream(arr).max().getAsInt() - shift;**

**int[] counts = new int[K + 1];**

**for (int elem : arr) {**

**counts[elem - shift] += 1;**

**}**

**// we now overwrite our original counts with the starting index**

**// of each element in the final sorted array**

**int startingIndex = 0;**

**for (int i = 0; i < K + 1; i++) {**

**int count = counts[i];**

**counts[i] = startingIndex;**

**startingIndex += count;**

**}**

**int sortedArray[] = new int[arr.length];**

**for (int elem : arr) {**

**sortedArray[counts[elem - shift]] = elem;**

**// since we have placed an item in index counts[elem], we need to**

**// increment counts[elem] index by 1 so the next duplicate element**

**// is placed in appropriate index**

**counts[elem - shift] += 1;**

**}**

**// common practice to copy over sorted list into original arr**

**// it's fine to just return the sortedArray at this point as well**

**for (int i = 0; i < arr.length; i++) {**

**arr[i] = sortedArray[i];**

**}**

**}**

**}**

The space complexity of counting sort is also *O*(*N*+*K*) since we have to initialize a new array of size *N* and a counts array of size *K*+1.

Another important constraint is that counting sort is only viable on inputs that have a fixed size (integers in a range, characters, etc.). If the possible set of inputs is an array of strings, counting sort is not a viable option.

Advantages of using counting sort: 1. It is a **stable sort**. 2. It can be significantly faster than other comparison based sorts on larger collections of integers with a relatively small range of values.

Disadvantages of using counting sort: 1. It requires extra memory, while many comparison sorts can be implemented without requiring any extra memory. 2. When the range of possible values K is large compared to N, counting sort may actually perform worse than a theoretically slower *O*(*N*log*N*) sort as a result of the extra memory overhead and additional K operations that need to be performed.