# Sorting Algorithms

By: Anthony Nguyen

*Assume there's a global array and the swap function takes array indices and swaps the contents within.*

## Table of Contents

## 1. Introduction

Sorting is very useful. When an array is sorted, it makes finding an item in the array much easier and much faster (forget hashing for now). There are a bunch of sorting methods and some are better than others. Choosing the right one for your situation is crucial to writing the most efficient program.

We will assume that a sorted list always has values in increasing order.

There are five main sorting algorithms, some that have variations. We will be going through all of them. These are:

- Insertion sort
- Exchange sort
- Selection sort
- Merge sort
- Distribution sort

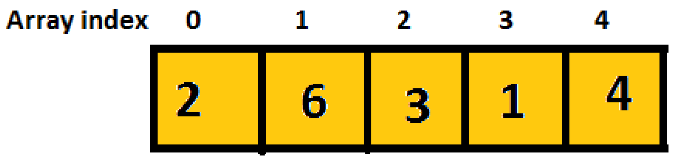
## 2. Insertion Sort

With insertion sort, the general idea is to take everything out of the array and insert elements one by one until the list has been filled. Each insertion makes sure the element is placed in the correct location.

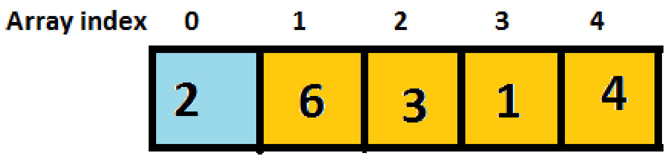
### 2.1 Straight Insertion Sort

Straight insertion sort involves re-inserting the elements from left to right. At each index in the array, we compare the index's value to the value to the left. If the value to the left is greater, then the values swap, and then the process repeats until the value to the left is not greater (the sorting loop).

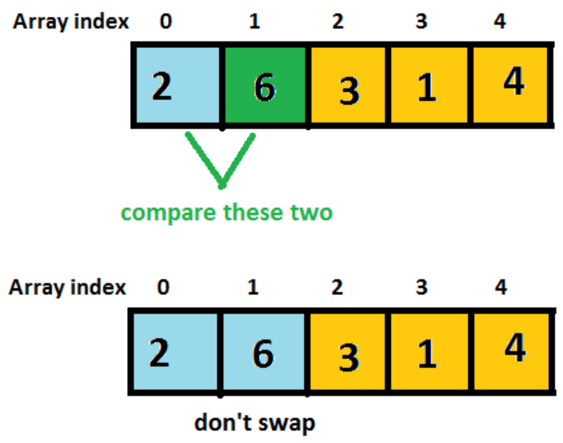
Let's see this in steps. This diagram will use light blue for the already sorted, green for the value we're "inserting", and orange for the non-sorted section. At first, the entire array is non-sorted.



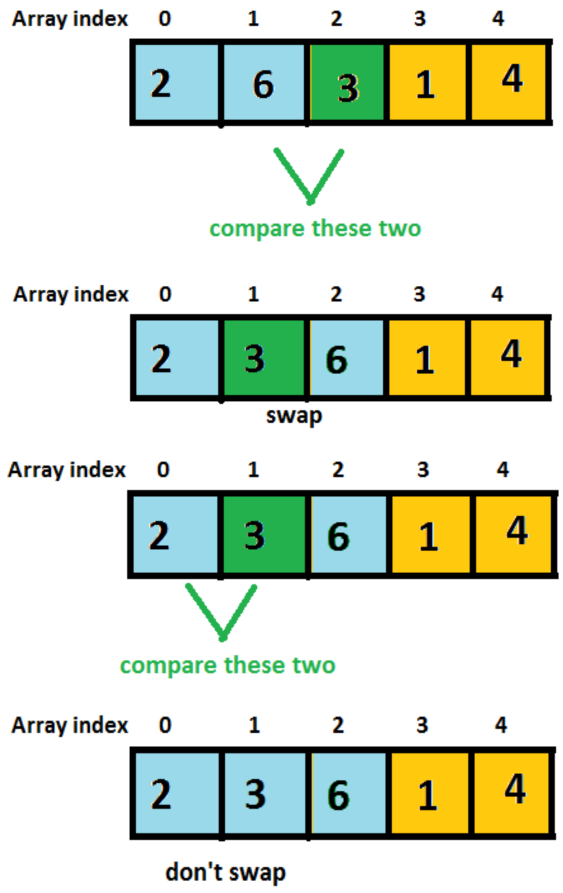
1. We "insert" the first array element. There is no value to the left to compare to, so there's no need to run the loop here. Moving right along.



2. We then "insert" array index 1. After inserting array index 1, we compare it with the value of index 0, the one to the left. If it is greater, we swap, if it isn't, we don't, and the loop breaks.



We then repeat step 2 on the next array index. We keep doing this until we get to the right of the array.



I’m not gonna keep doing the steps through a diagram. In the end the array will have all the numbers in increasing order (1 2 3 4 6, if you don’t believe me you can look at a number line).

Now we move onto coding this thing. We’ll start off with our array declaration and sort function declaration. These are pretty straightforward. You would pass off the array into the function by reference, because you want the changes in the function to apply permanently to the array. You also pass of the number of elements in the array.

```
void sort (int *array, int n)
{
```

This procedure is done as a **nested for loop**. This means we have to create two counter variables, which we will call i and j. The first counter variable **i** is to keep track of how far we are in the array for the entire sorting. The second counter variable **j** is to keep track of the specific array element’s position before and after swapping (if necessary).

```
 int i, j ;
```

We then create our first loop. This is the loop that runs the sorting algorithm on every index in the array, except the first one. We don’t run it on the first one because there is nothing to the left of the first index, hence why it is completely unnecessary. This is why i starts at 1. We can show this in our diagram.

```
 for ( i=1; i<n; i++ )
 {
```

At this stage, we can run our sorting algorithm. With the loop, the starting point is **i**, which is the index in the array we have reached in the previous loop.

```
 for ( j=i;
```

The second condition is the “keep going” condition, which tells the loop to keep going until one of the conditions breaks. Here, we compare the value of the current array index to the previous one. We want the loop to run while the index is greater than 0 (index of 0 means we reached the left of the array), and also while the value to the left of the current index is greater than that of the right.

```
 j>0 &&
 (array[j-1]>array[j]);
```

The third condition is the ‘simplifying’ condition which decreases the value of j by 1 on each iteration.

```
 j-- )
 {
```

If all conditions are satisfied, it means that the value to the left is greater than that of the right. This means we need to swap the two. After it does this, it will go back to checking the loop conditions.

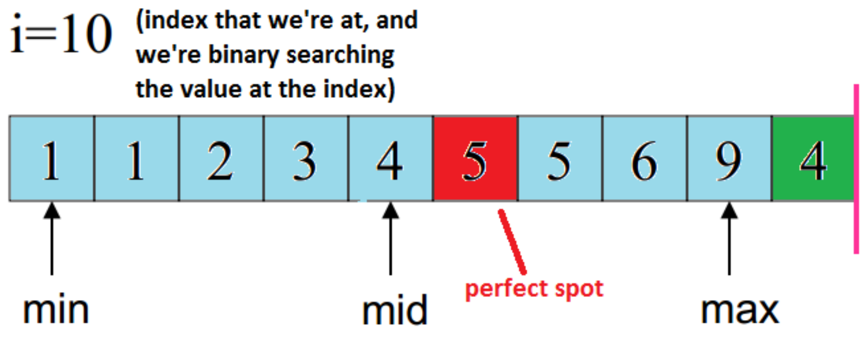
```
 swap(array[j], array[j-1]);
 }
 }
}
```

In its best case, if the entire list is sorted already, the runtime of this sort algorithm will be O(n) because it still needs to go through all the array elements and check the previous. In its worst case, if the entire list is in reverse order, the runtime of this sort algorithm will be O(n^2).

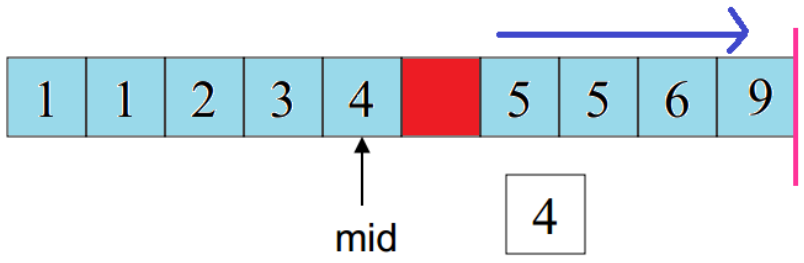
### 2.2 Binary Insertion Sort

Binary insertion sort is another insertion algorithm. The difference between this algorithm and the previous one is that for this one, we look for the perfect place to "insert" the element. Here are the steps:

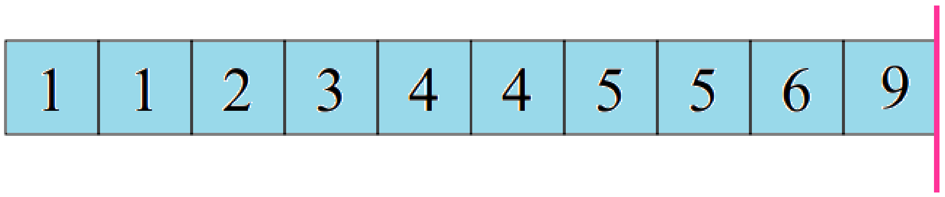
1. Run a for loop to go through every index in the array. The next steps will be run on the specific index in the array.
2. Use binary search (http://www.youtube.com/watch?v=wNVCJj642n4) to find the perfect place to insert the array element. The (slightly modified) binary search function returns some array index as the perfect location.
    a. If the binary search finds the value, it will return the existing value's array index plus one.
    b. If the binary search does not find the value, the binary search's 'middle' will hold some value that would be to the left or right of the 'inserted' value. If the value in the 'middle' index is greater than that of the 'inserted' value, the function will return the middle. If it is less, then the function returns the middle plus 1.


i=10
(index that we're at, and we're binary searching the value at the index)
1 1 2 3 4 5 5 6 9 4
min
mid
perfect spot
max

3. Shift all values in the array, from the 'perfect location', up until the end of the partial list, up one. This is done in a for loop, starting at the top element (i-1), and ending at the perfect spot.



4. Place the value we want to insert into the perfect location.



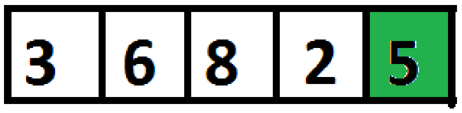
We probably won't have to code this on the quiz. It's not part of the assignment (only lab).

## 3. Exchange Sort

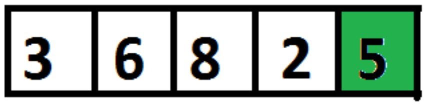
### 3.1 Bubble Sort

Bubble sort is where, on each iteration of the function, the lowest value on each iteration bubbles to the top. Here are the steps:

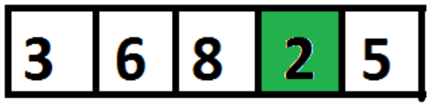
1. We first want to look for a value for the first array index (we'll call it i=0 for the first iteration). We start at the end of the array (the process of looking for this value is done in a loop that is in descending order).

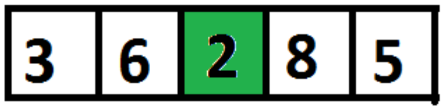


2. Compare the current array index to the one to the left (index – 1).
    a. If the value to the left is greater, swap the current and left array index values.
    b. If the value to the left is less, don't do anything.

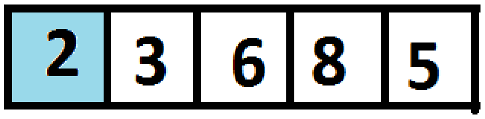


3. Change the current array index to the one to the left, and then repeat step 2. Keep going until we reach the first array index.





After running steps 2 and 3 repeatedly until we reach the first array index…



4. Now, check to see if there was any swapping done in the pass. If there wasn't any swapping done, that means the **list is already sorted**, and you can just exit sort. If not, then continue on.
5. Repeat steps 1-4, starting once again at the end of the array. This time, the 'first array index' will be the previous 'first array index' plus one.

The code looks like this. This assumes an array that starts at 0.

```
void bubbleSort(int *array, int n)
{
```

We declare a couple of variables for our loops as well as a variable called **swapped** to keep track of if there was any swapping.

```
int i, j, swapped ;
```

This is where we do an ascending for loop to decide on our ‘first array index’ mentioned earlier. We start at i = 0, and then end at the array size minus 1. The ‘first array index’ has the variable ***i***.

```
for (i=0; i<=n-1; i++)
 {
```

We then assigned the *swapped* variable a value of 0 at the beginning of each pass, to signify no swaps yet.

```
        swapped = 0;
```

Now we do the swapping part in a nested loop. The ‘current array index’ described earlier has the variable ***j***. The switching of the current array index is done by decreasing the value of *j* by 1 on each iteration.

```
        for (j=n-1; j>i; j--)
        {
```

If the value to the left to the current array index is greater than that of the left, then the two values swap and the swap gets recorded.

```
        if (array[j] < array[j-1])
        {
        swap(j, j-1) ;
        swapped++;
        }
        }
```

Now we have exited our j loop. If a swap has not been recorded, then it won’t bother running the next iteration of the *i* loop.

```
        if(swapped == 0)
        {
            break;
            }
    }
}
```

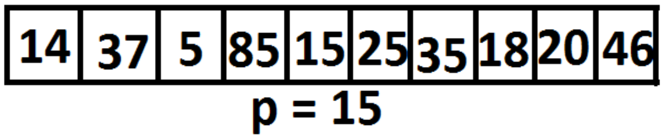
## 3.2 Quicksort

Quicksort is a little more complicated than bubble sort but is very fast for many applications. There are two parts to quicksort: the partitioning and the quicksorting.

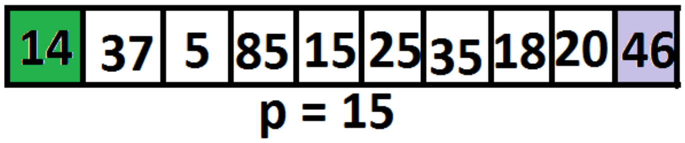
### How to Partition an Array

Partitioning an array means having assigning an array pivot, having all elements to the left of the pivot be less than the pivot, and having all elements to the right of the pivot be greater than the pivot. Here are the steps to doing this:

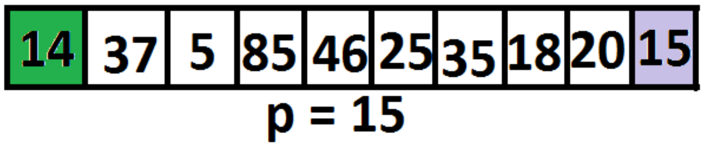
1. Pick a pivot. This is done arbitrarily, and there are different ways to pick one based on the set of data, but with our method we're just going to take the middle for simplicity.

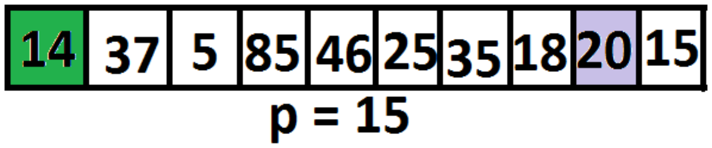


2. Now, assign a **left and right selector**. The left selector should start at the left most side of the array, and the right selector should start at the right. In the diagram, the left selector is green, and the right selector is purple.

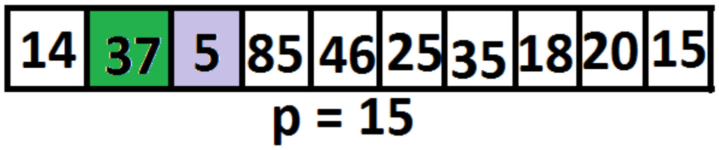


3. Swap the locations of the pivot and whatever is in the right selector. Move the right selector over one.

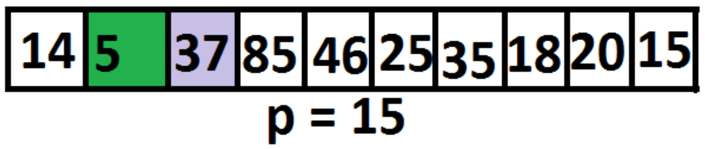




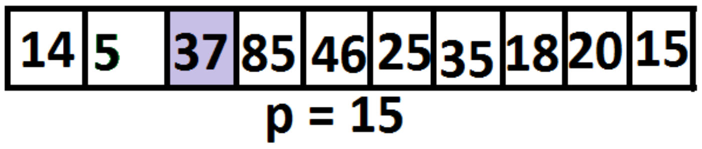
4. Check the value at each selector. Move the left selector to the right **until you reach a value that is greater than or equal to the pivot**. Move the right selector to the left **until you reach a value that is less than or equal to the pivot**.

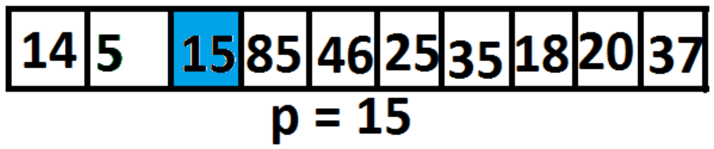


5. Swap the values of the left selector and the right selector, if at the same time, the left selector has a value greater than that of the pivot, and the right selector has a value less than that of the pivot.



6. Eventually, the selectors will be at the same index in the array. At this point, swap the locations of the current selector's value and the pivot, but **only if the value of the pivot less than that of the selector** (if the value of the pivot is greater, that means that the pivot is already in the correct spot being at the very end of the array).





Now, all the values to the left of the pivot are less than the pivot, and all the values to the right of the pivot are greater than the pivot.

## The Entire QuickSort method

1. If the array has 1 element or less, there is no need to run the quicksort method, and it skips everything.
2. It then partitions the array. At the end of the partitioning, we will be left with a pivot that is in the right location.
3. We treat the group of elements to the left and right of the pivot as smaller arrays. We recursively call the quicksort method on the array to the left of the pivot, and the on the array to the right of the pivot.

Now here's the code. Right now, I'm assuming the array is declared as a global variable, so we're not taking that as input. I'm also assuming an array that starts at 1. The left and right of the array, or the indexes of the first and last elements in the array, are passed off to the function.

```
void quickSort(int left, int right)
{
```

Now we check to see if the array has more than one element. We do this by subtracting the left index from the right index (essentially the array size), and seeing if it is greater than 0. If right and left give a difference of 0, it means that the array either has one element or none. This also acts as our terminating condition.

```
 if (right – left > 0)
 {
```

If the condition is passed, then we get to move onto our partitioning. We select a pivot index *p* with some function that will select a pivot index. The function is defined somewhere else, can have whatever method of pivot selection you want, and takes the left and right array bounds as input.

```
int p = selectPivot(left, right) ;
```

We then get to the step where we swap the values inside the pivot index and the right index.

```
swap(p, right) ;
```

We then assign another variable to hold the value of the pivot.

```
int pivot = array[right] ;
```

Now, we assign two variables for the left and right *selector indexes*. The left selector index (i) is given the left of the array, and the right selector index is given the right of the array, subtract 1 (basically moving the right selector over one).

```
int i= left ;
int j= right -1;
```

Now we do a for loop with two while conditions, to do our selector moving and swapping. The first while condition is for moving the left selector (moves it to the right when the value at the left selector is still less than the pivot), and the second while condition is for moving the right selector (moves it to the left when the value at the right selector is still greater than the pivot).

```
for (;;)
{
while(i < j && array[i]< pivot) i++ ;
while(i < j && array[j]> pivot) j-- ;
```

If the left and right selectors are at the same index, the for loop will stop.

```
if ( i >= j ) break ;
```

If the value at the left selector is greater than that of the pivot, and the value at the right selector is less than that of the pivot, we swap the values in the left and right selectors, and increment by 1 after.

```
swap(i++, j--) ;
}
```

Now, once the left and right selectors are at the same spot, it will check to see if the selector value is greater than the pivot value. If it is, then we can swap the pivot and selector values.

```
17.  if (array[i]> pivot )
18.  swap(i, right) ;
```

Now it checks to see if the selector is at the left of the array. If it isn’t, then the program will recursively run quicksort on the smaller array to the left of the selector (passing off the left as the left, and the selector minus 1 to as the right).

```
19.  if (left < i)
20.  quicksort( left, i-1 ) ;
```

It then checks to see if the selector is at the right of the array. If it isn’t, then the program will recursively run quicksort on the smaller array to the right of the selector (passing off the right as the right, and the selector plus 1 as the left).

```
21.  if (right > i)
22.  quicksort( i+1, right ) ;
23.  }
24. }
```

### Cutoff???

Quicksort sucks when the set of data is small. If we want to use some other sorting method for smaller arrays, we change the condition at line 3 to be if the right – left + 1 is greater than the cutoff.

```
3.  if (right - left + 1 > cutoff)
```

We then add an else condition after the if condition, and then send the left and right to some other function that uses some other sorting method. That looks like this:

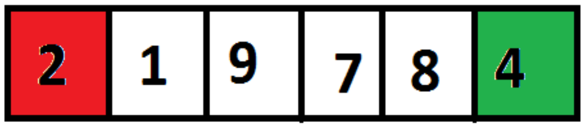
*else someOtherSort( left, right ) ;*

## 4. Selection Sort

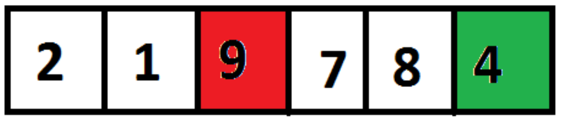
### 4.1 Straight Selection Sort

Straight selection sort is easy. It is similar to insertion sort where you re-insert all of the values, but this time you pick which values you want to insert. We insert from right to left based on the max value to the left of the index. Here are the steps:

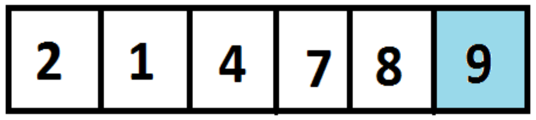
1. Start the insertion at the right side of the array, with the initial max being on the left side of the array. We’re going to use green as the array index we’re inserting into (insertion index), red for the current maximum value in the array, and light blue for the already sorted section.



2. It will then traverse the entire array to the left of the insertion index, to find the actual max.



3. Once we find the max, we swap the value of the max and the value inside the insertion index **only if the max is greater than the value already in the insertion index**.



4. Repeat steps 1-3, but with the insertion index being to the left of the previous one.

## Code

In this code we assume an array that starts at index 1, and an array that is defined globally.

```
void straightSelectionSort()
{
```

We create a loop for the insertion index. We start it at n (the array size), and move the index down until it is at index 1, which is the last index.

```
 for (int i=n; i>1; i--)
 {
```

We then declare a max index. We first make it equal to 0 because array index 0 holds nothing, and any value in the array will be greater than the value in the max index (this will make sense trust me).

```
 int max = 0 ;
```

Now it does a linear traversal of all elements in the array up to the inserted index. It checks to see if the value of the index, at that point, is greater than the value inside the max index. If it is, then it changes the max index to the new index.

```
 for (int j=1; j<i; j++ )
 {
 if (array[j] > (array[max]))
 max = j ;
 }
```

Finally, we swap the value in the max index with the value in the insertion index, minus 1 (we start at n, but n doesn’t exist in the array, so we have to go down).

```
 swap(i-1, max) ;
 }
}
```

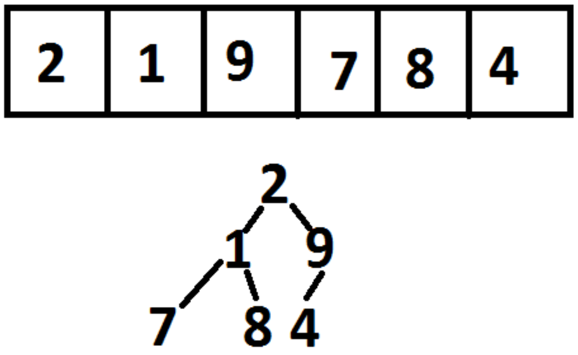
### 4.2 Heap Sorting

Remember our good buddy the heap? Yeah well I hope you haven't forgotten it because we actually implement a heap on an unsorted array to help sort our array. Since we are assuming an array index of 1, the root will be at array index 1, the left child will be at (parent index * 2) and the right child will be at (parent index * 2 + 1).

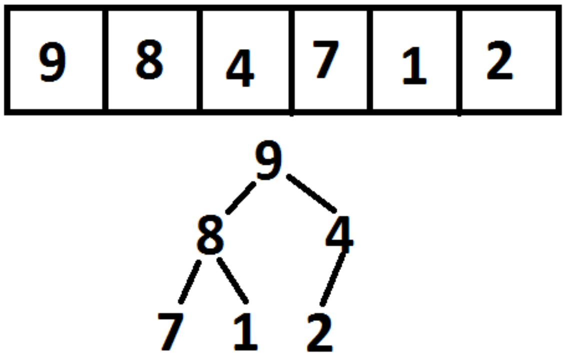
## Part 1: Making our unsorted array into a heap

We want our array to be sorted into a **max heap** (refresher: this means that parents are always greater than their children).

1. At first, our tree looks something like this:



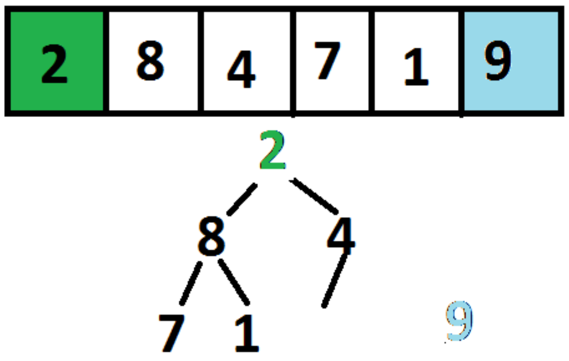
2. Next, we percolate. Percolating is when you take the parent node, compare it to the children nodes, and swap with the larger child (if and only if the parent node is less than the child). You then repeat the process on the parent's new location.
   It is best to start percolating at index (n/2) where n is the size of the array (round down if there is a decimal result). You then work your way back to the root (i.e. percolate at the previous index until you get to the root), and eventually the entire tree will satisfy the heap condition.



## Part 2: Sorting

Now, we begin to remove items from the tree and place them in the array from left to right.

1. Swap the root node with the last node that is still in the tree. Remove the old root node from the tree (this is done by cutting down the number of elements in the tree with a loop).



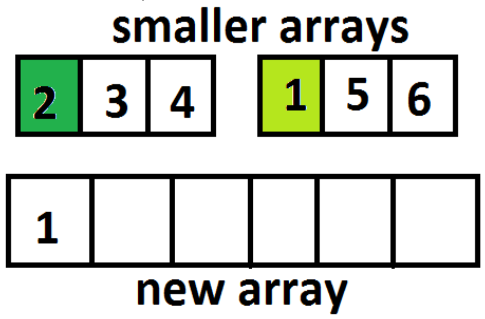
2. Percolate the new root node to its correct position.
3. Repeat steps 1-2. The elements 'removed' from the tree are **no longer part of the tree**.

## 5. Merge Sort

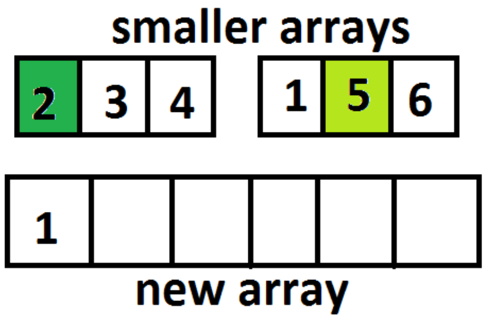
### How to Merge Two Arrays

When you merge two arrays, basically you take two already sorted arrays and insert them into a new array. Here's how you do it:

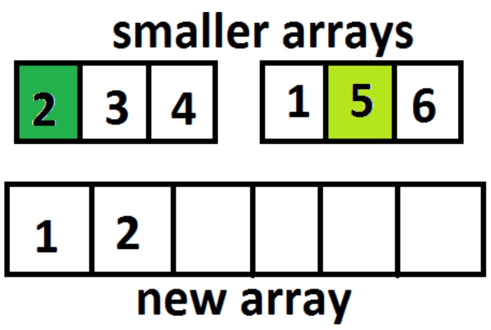
1. Start at the left most element in each array. Compare the values. Insert the smaller value into the new array, at the left most unfilled spot.



2. Move the index one over, for the array that just inserted into the new array.



3. Compare values again, insert again, and then repeat step 2. Keep going until the new array is filled.

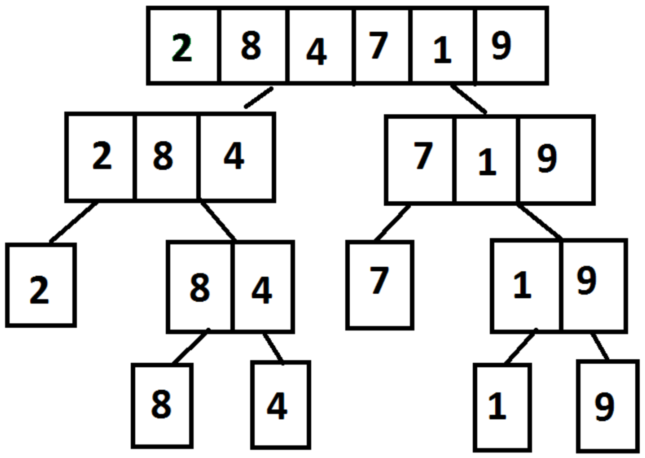


And so on…

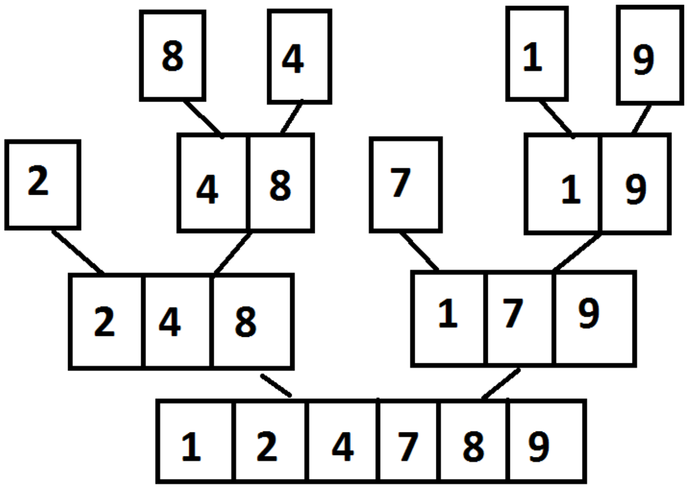
## How to Merge Sort

Merge sort is when you split your array into small arrays of size 1, and then merge all of those arrays together. The steps look like this.

1. Recursively split up your array into smaller arrays of size 1. Do this by splitting the array in half recursively until you can't split up the array anymore. **Remember the path taken when doing this (especially if you're doing this by hand)**. Note that when you split an odd numbered array in half, you always round down, so the array to the right always has one more element.



2. Now, merge arrays that came from the same 'parent array', using the merging steps that were described earlier.



I pulled code off of Wikipedia (edited names a little), because it makes a lot more sense, and shows the entire thing. **This time we assume an array that starts at index 0 (which I found out is just like our lab…. There's a big hint on how to change the lab code).**

## Splitting Function

First, we set up our splitting function. Since our splitting function will also call the merge function, it is more suitable to call this function SplitMerge. We pass off the array, the index for the beginning of the array (iBegin), the index for the end of the array (iEnd), and a temporary array for copying.

```
void SplitMerge(array[], iBegin, iEnd, tempArray[])
{
```

We then check to see if the size of the array is less than 2. If it is, then we consider it already broken down into its simplest form and we end the split merge.

```
    if(iEnd - iBegin < 2)                       // if run size == 1
        return;                                 //   consider it sorted
```

Now, if it passes the first condition, it will find a middle to split the array. Note that this is an integer, so decimal numbers are rounded down.

```
    int iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
```

At this point, it will recursively split the left and right sub arrays. To recursively split the left array, we pass off the entire array, the beginning index (as the start of the array), the middle index - 1 (as the end of the array), and the temporary array.

```
    SplitMerge(array, iBegin,  (iMiddle – 1), tempArray);  // split / merge left  half
```

To recursively split the right array, we pass off the entire array, the middle index (as the start of the array), the end index (as the end of the array), and the temporary array.

```
    SplitMerge(array, iMiddle, iEnd,    tempArray);  // split / merge right half
```

Now, we call our merging function. The merging function takes five arguments: the actual array, the start of the array, the middle of the array, the end of the array, and a temporary array to copy values. It will be described in a bit.

```
    Merge(array, iBegin, iMiddle, iEnd, tempArray);  // merge the two half runs
```

Finally, it is necessary to copy our values from the temporary array back into the original array.

```
    CopyArray(tempArray, iBegin, iEnd, array);              // copy the merged runs back to A
}
```

### Merge Function

Now we write our merge function. The input arguments were described before.

```
void Merge(array[], iBegin, iMiddle, iEnd, tempArray[])
{
```

We declare some variables to keep track of the left sub array and the right sub array, called i0 and i1 (I'm calling these the track indexes, basically selectors).

```
    int i0 = iBegin, int i1 = iMiddle;
```

We then run a for loop to fill in each index of our temporary array. We want it to start at the first index and end at the last.

```
    for (j = iBegin; j < iEnd; j++) {
```

Now we have to decide which sub array element we want to put in. We want to insert the element from the left sub array if:

- Our left sub array has not reached its end (i.e. the left track index is still below the middle)
- Our right sub array **has** reached its end (i.e. right track index >= end of array) **OR** the current value at the left array is less than that of the current value at the right array (at their respective trackers)

When we insert, we want to increment the track index by 1 for whichever array just inserted a value.

```
        // If left run head exists and is <= existing right run head.
        if (i0 < iMiddle && (i1 >= iEnd || array[i0] <= array[i1]))
            tempArray[j] = array[i0++];  // Increment i0 after using it as an index.
```

If the condition isn't satisfied, it means we need to insert from the right sub array.

```
        else
            tempArray[j] = array[i1++];  // Increment i1 after using it as an index.
    }
}
```

Because this is done recursively, everything is done in the correct order.

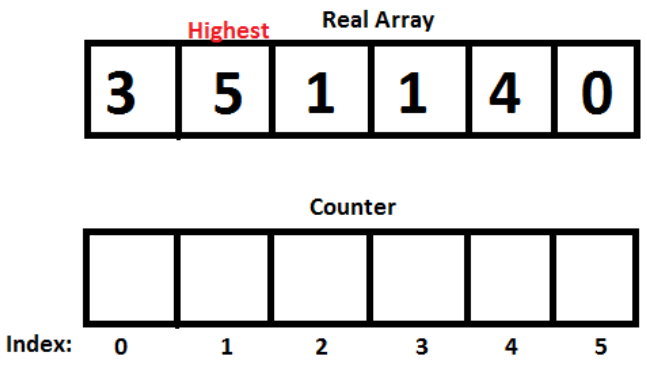
## 6. Distribution Sort

### 6.1 Bucket Sort

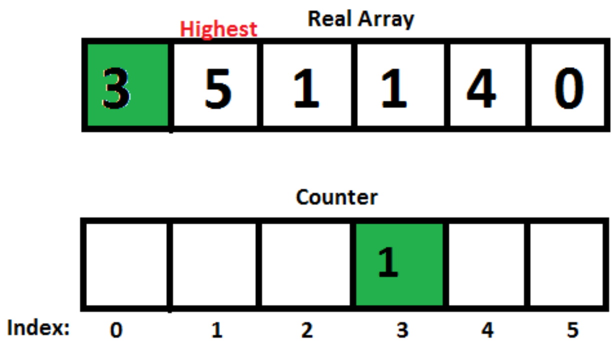
Bucket sort is great for when you know, exactly, the bounds of the values in your array. It should be done when the bounds are small, otherwise the operation takes up a lot of space. The idea is that we count the number of times each value appears in the array, and then insert based on that.

Here are the steps:

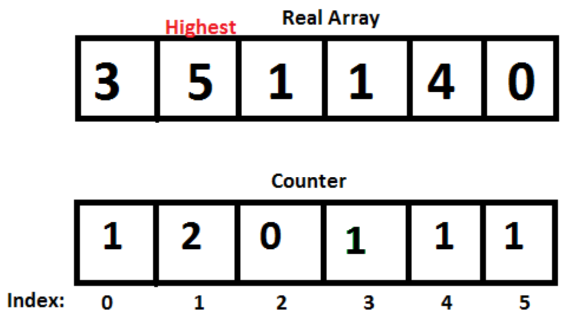
1. Create a **counter** array to count the number of times the index value appears. The end of the counter array should be the highest value in the array.



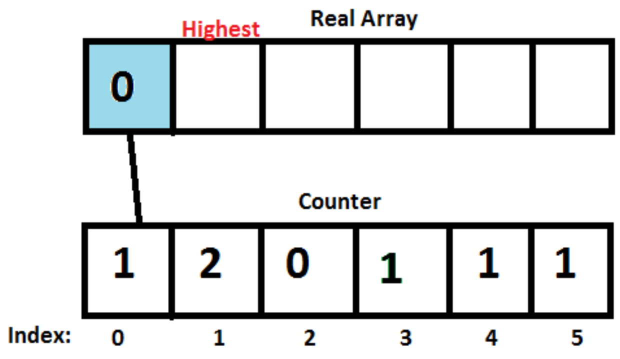
2. Traverse the array. At each element of the real array, add one the index of the counter array equal to the value. In our example, the first element has a value of 3, **so we add one to the counter of index 3.**



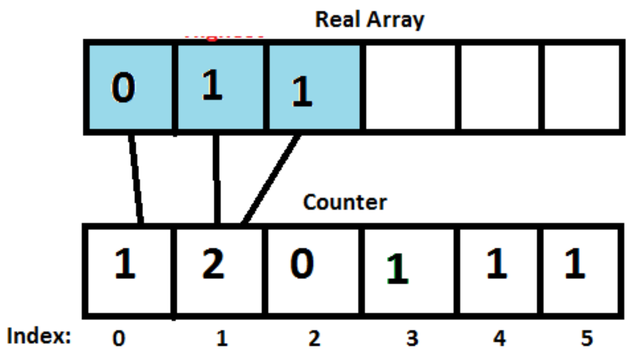
Several steps later…



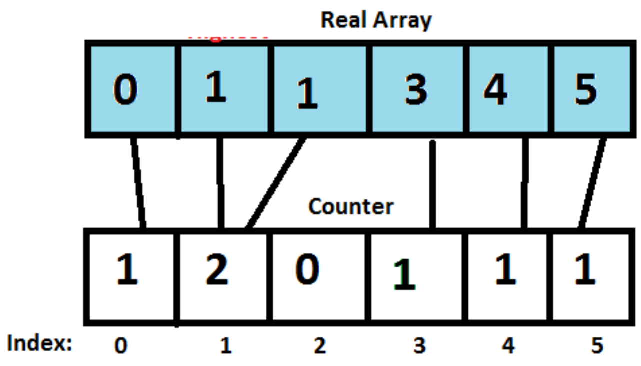
3. Now, re-fill the real array. This is done by inserting the index the number of times it occurred in the original array. In our example, in the counter array, the index of 0 has a value of 1, meaning it occurred once. We insert that in first.



The index of 1 has a value of 2, meaning it occurred twice. We insert 1 into our real array twice.



Keep going until we get….



## Coding

```
void BucketSorter()
{
```

Create a variable m which contains the largest value of the array. The largest value isn't found in the code but yeah just find it yourself using a linear traversal or something.

```
 int m; // finite universe range
```

We now declare a counter array, with size m (largest value of the array).

```
int count[m];
```

Now we fill out our array with zeroes first, in a for loop.

```
for (int i = 0; i < m; ++i)
count [i] = 0;
```

Now we do that counting. At each element of the regular array, we add one to the value of the **index equal to the value at the regular array**. Yeah I can't really come up with a great sentence for this, refer to diagrams for what I mean and it will make sense.

```
for (int j = 0; j < n; ++j)
++count [array [j]];
```

Now we go back to filling in our regular array. We do a for loop with four statements:

- The starting index of the counter array, which we will call i
- The starting index of the regular array, which we will call j
- We want to keep going until the index of the counter array reaches its max value
- We want to increment the index of the counter array by 1 on each iteration

```
for (int i = 0, j = 0; i < m; ++i)
```

We then have a for loop that basically says add the counter index number to the regular array until the value inside the counter index is equal to 0. When inserting, we do j++ because we want the regular array index to increment by 1 on each insertion.

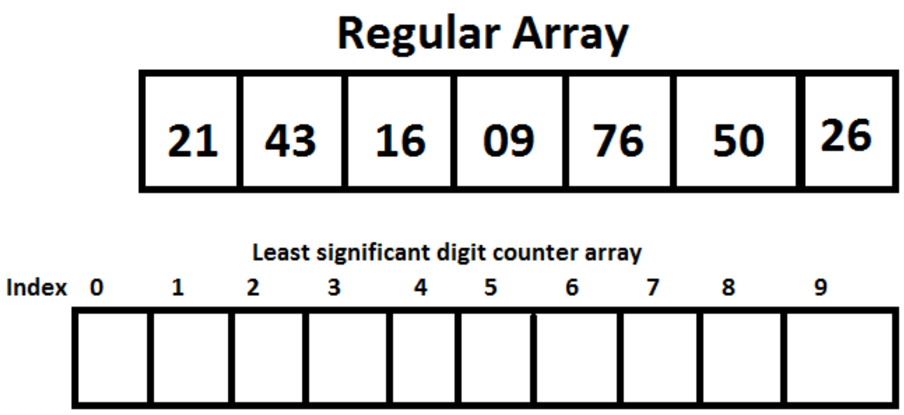
```
for ( ; count [i] > 0; --count [i])
array [j++] = i;

}
```

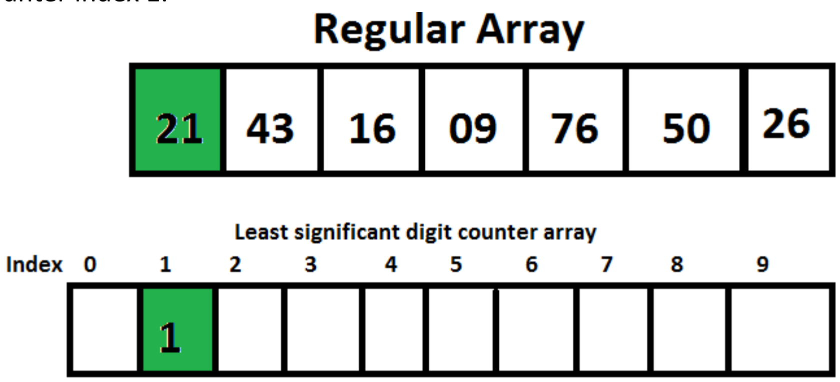
### 6.2 Radix Sort

Radix sort has a similar idea to the bucket sort in the sense that it uses a counter for the least significant digit. Then, it does an offset and it gets pretty messy but I'm gonna do my best to explain it. This method is better because you only need to know your bounds in base 10. It does, however, take up more space. Here are the steps:

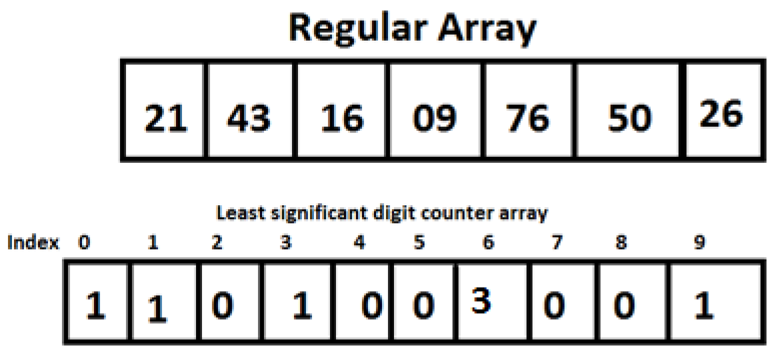
1. Create a counter index of 10 elements (assuming we start at index 0).


Regular Array
21 43 16 09 76 50 26
Least significant digit counter array
Index 0 1 2 3 4 5 6 7 8 9

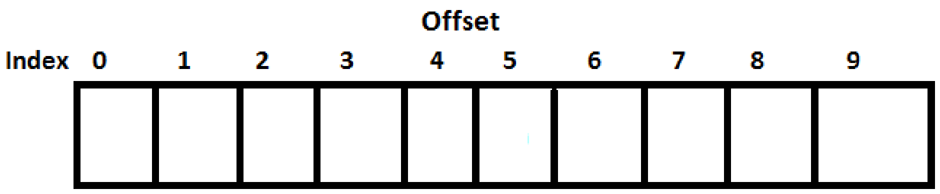
2. Fill in the least significant digit counter array like we did before, except count one at the counter index equal to the value of the LSD (we're treating it as if there's a decimal at the end so 0 counts as a significant digit). In our example, the first array element has a LSD of 1, so we add one at the counter index 1.


Regular Array
21 43 16 09 76 50 26
Least significant digit counter array
Index 0 1 2 3 4 5 6 7 8 9
1

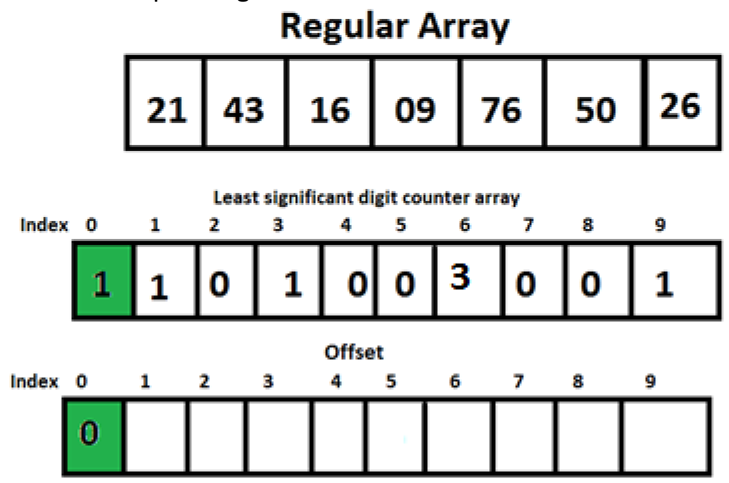
A few steps later…


Regular Array
21 43 16 09 76 50 26
Least significant digit counter array
Index 0 1 2 3 4 5 6 7 8 9
1 1 0 1 0 0 3 0 0 1

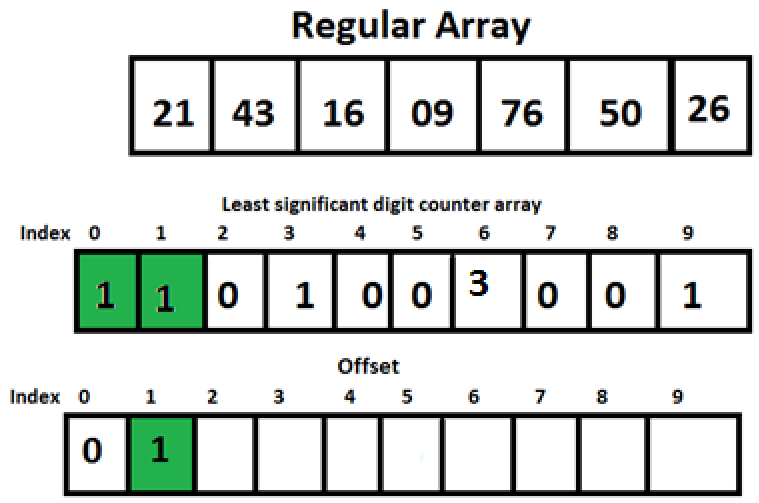
3. This part is a little tricky. Now, we create an array of 10 elements called the **offset array**.



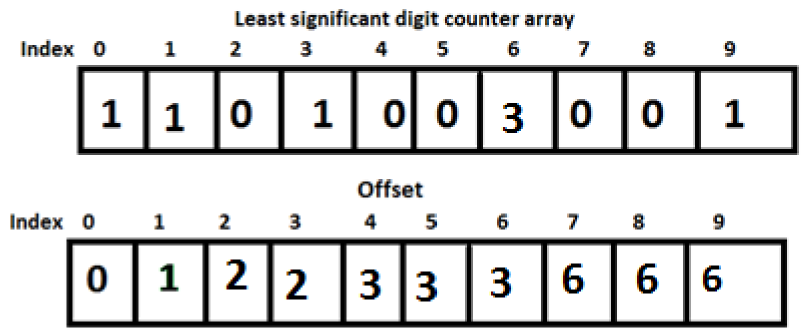
At each index of the offset, you add the first occurrence of the least significant digit in a sorted list. That's how it works technically, but here's the way I like to do it. Add all of the counter values to the left of the corresponding offset index.


Regular Array
21 43 16 09 76 50 26
Least significant digit counter array
Index 0 1 2 3 4 5 6 7 8 9
1 1 0 1 0 0 3 0 0 1
Offset
Index 0 1 2 3 4 5 6 7 8 9
0

The sum of the values to the left of counter index 0 is 0, so we place zero there.

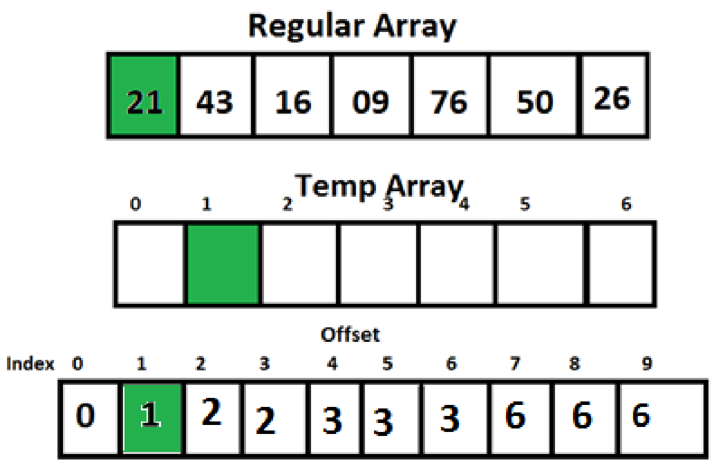

Regular Array
21 43 16 09 76 50 26
Least significant digit counter array
Index 0 1 2 3 4 5 6 7 8 9
1 1 0 1 0 0 3 0 0 1
Offset
Index 0 1 2 3 4 5 6 7 8 9
0 1

At offset index 1, the sum of the values to the left of index 1 is 1, so we put a 1 there.
Keep going until eventually, we get this:

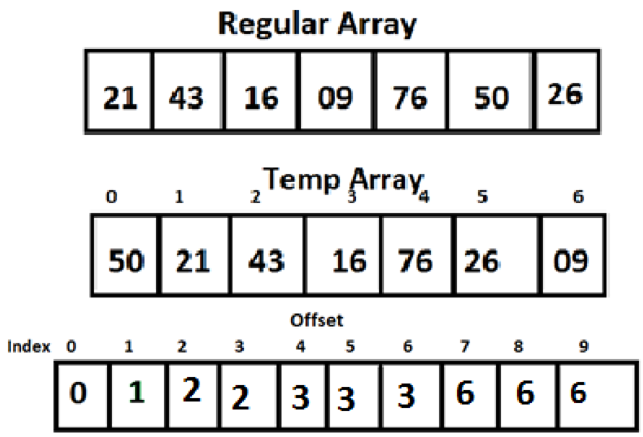

Least significant digit counter array
Index 0 1 2 3 4 5 6 7 8 9
1 1 0 1 0 0 3 0 0 1
Offset
Index 0 1 2 3 4 5 6 7 8 9
0 1 2 2 3 3 3 6 6 6

4. Using the offset, we can now insert elements into another temporary array. We do this by traversing the original array and inspecting the LSD. Then, we go to the index equal to the LSD in

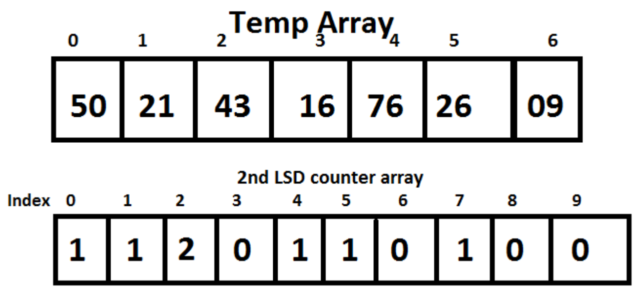
the offset array. We then place the element at the index equal to the value stored inside the offset array. For example,



Our first element has a LSD of 1. It goes to the offset array at index 1, and sees that the value here is 1. Therefore, the element is inserted at index 1 of the temp array.
In the event of a collision, we move over one array index in the temp array.
The final result looks like this:



5. Now the elements are in order for LSD. We now replace the regular array with the temp array. Now, we need to repeat steps 1-4 for the *next least significant digit*. We keep going until we reach our most significant digit for the set of data, and we stop. When we finish, we will have a sorted list.
   a. Step 1 and 2



   b. Step 3

2nd LSD counter array

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 2 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

Offset

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 6 | 7 | 7 |

c. Step 4

Temp Array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 50 | 21 | 43 | 16 | 76 | 26 | 09 |

Temp Array 2

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 09 | 16 | 21 | 26 | 43 | 50 | 76 |

Offset

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 6 | 7 | 7 |

Temp array 2 is now our fully sorted array.