Sorting

Data Structures – CS2001 – Fall 2021

Dr. Syed Ali Raza

School of Computer Science

National University of Computing & Emerging Sciences

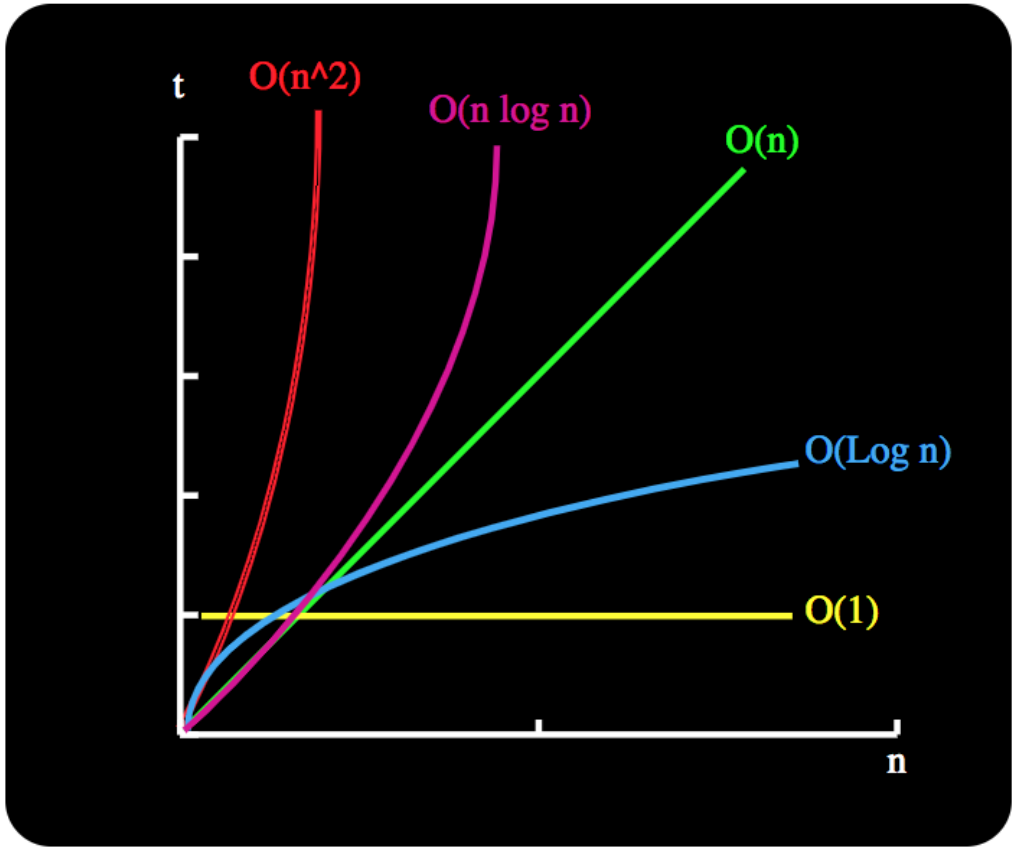
Karachi Campus

Outlines

- Elementary Sorting.
  - Bubble Sort.
  - Insertion Sort.
  - Selection Sort.
- Efficient Sorting.
  - Merge Sort.
  - Quick Sort.
  - Shell Sort.
  - Distribution Counting Sort.
  - Radix Sort.

Complexity Analysis

- Complexity analysis tells how the number of computations or the computation time (t) will grow (on y-axis) if number of elements in array (n) grow.
- O(n^2) is a bad complexity because the computation time grows exponentially as n grows.
- O(n logn) still grows exponentially but it is better than O(n^2).
- Generally, we want to avoid O(n^2) and want to achieve O(n) or better performance.
- Note, O(logn) is better than O(n) because for very large values of n, it almost becomes a constant.



Bubble Sort

- The array is scanned from the bottom up, and two adjacent elements are interchanged if they are found to be out of order with respect to each other.
- There are multiple passes of scanning.
- Time complexity is O(n^2) due to nested loops.
- It can be optimized by stopping the algorithm if the inner loop didn’t cause any swap.
- Elements at the end of the list start to become sorted first.
- It can be done in reverse: start from the back of the array and move the smallest to the front.

		0	1	2	3	4	5	6	7
i = 0	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i=1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i=2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i=3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i=4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i=5	0	1	2	3	4				
	1	1	2	3					
i=6	0	1	2	3					
		1	2						

Bubble Sort

- We will sort by bubbling largest number to the end of the array.
- For this, we need two loops.
- An outer loop for the number of scans and an inner loop to make comparisons in each scan.
- The outer loop goes from 0 to n-1.
- The inner loop goes from 0 to n-1-(index of outer loop).

		0	1	2	3	4	5	6	7
i = 0	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i=1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i=2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i=3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i=4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i=5	0	1	2	3	4				
	1	1	2	3					
i=6	0	1	2	3					
		1	2						

Selection Sort

- The element with the lowest value is selected and exchanged with the element in the first position.
- Then, the smallest value among the remaining elements data[1], . . . , data[n-1] is found and put in the second position.
- It continues until all elements are in their proper positions.
- Worst case as well as best case time complexity is O(n^2) due to nested loops.

```
selectionsort(data[],n)
    for i = 0 to n-2
        select the smallest element among data[i],...,data[n-1];
        swap it with data[i];
```

Selection Sort

arr[] = 64 25 12 22 11
// Find the minimum element in arr[0...4]
// and place it at beginning s
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 **12** 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 **22** 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 **25** 64

Selection Sort

- Needs nested loops.
- The outer loop goes from 0 to n-2 and is used to move the boundary of unsorted part of the array.
- The inner loop goes from (index of outer loop) to n-1 and is used to find the minimum in the unsorted part of the array.

Insertion Sort

- An *insertion sort* starts by considering the two first elements of the array data, which are data[0] and data[1]. If they are out of order, an interchange takes place.
- Then, the third element, data[2], is considered. If data[2] is less than data[0] and data[1], these two elements are shifted by one position; data[0] is placed at position 1, data[1] at position 2, and data[2] at position 0.
- If data[2] is less than data[1] and not less than data[0], then only data[1] is moved to position 2 and its place is taken by data[2]. If, finally, data[2] is not less than both its predecessors, it stays in its current position.

Insertion Sort

Given array: 12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array) i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12
11, 12, 13, 5, 6

- for i = 2, 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
11, 12, 13, 5, 6

- for i = 3, 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
5, 11, 12, 13, 6

- for i = 4, 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
5, 6, 11, 12, 13

Insertion Sort

- We need two loops.
- An outer loop which goes from 1 to n-1 (say, using variable i).
- And an inner loop which goes from (index of outer loop)-1 to 0 (say, using variable j) if using a while loop.
- Hold ith element in a temporary variable (say, k), it will create a hole at the ith position.
- Then see if jth element is larger than ith element. If yes, then move jth element one position to right i.e. to j+1 position. This will create a hole at jth position.
- Keep doing it until jth element is not larger than ith element.
- Lastly, fill the hole at the j+1 position will k (i.e., insertion).

Insertion Sort – Complexity Analysis

- The best case is when the data are already in order. Only one comparison is made for each position *i,* so there are *n* - 1 comparisons, which is *O*(*n*), and 2(*n* - 1) moves, all of them redundant.
- The worst case is when the data are in reverse order. In this case, for each *i,* the item data[i] is less than every item data[0], . . . , data[i-1], and each of them is moved by one position.
- For each iteration *i* of the outer *for* loop, there are *i* comparisons, and the total number of comparisons for all iterations of this loop is,

$$\sum_{i=1}^{n-1} i = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

Insertion Sort – Complexity Analysis

- The number of times the assignment in the inner *for* loop is executed can be computed using the same formula. The number of times temporary variable is loaded and unloaded in the outer *for* loop is added to that, resulting in the total number of moves:

$$\frac{n(n-1)}{2} + 2(n-1) = \frac{n^2 + 3n - 4}{2} = O(n^2)$$

Insertion Sort – Complexity Analysis

- For every iteration *i* of the outer *for* loop, the number of comparisons depends on how far away the item data[i] is from its proper position in the currently sorted subarray data[0 . . . i-1].
- Generally, if it is *j* positions away from its proper location, data[i] is compared with *j* + 1 other elements. This means that, in iteration *i* of the outer for loop, there are either 1, 2, . . . , or *i* comparisons.

Insertion Sort – Complexity Analysis

Under the assumption of equal probability of occupying array cells, the average number of comparisons of `data[i]` with other elements during the iteration i of the outer `for` loop can be computed by adding all the possible numbers of times such tests are performed and dividing the sum by the number of such possibilities. The result is

$$\frac{1 + 2 + \cdots + i}{i} = \frac{\frac{1}{2}i(i + 1)}{i} = \frac{i + 1}{2}$$

To obtain the average number of all comparisons, the computed figure has to be added for all is (for all iterations of the outer `for` loop) from 1 to $n - 1$. The result is

$$\sum_{i=1}^{n-1} \frac{i + 1}{2} = \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} \frac{1}{2} = \frac{\frac{1}{2}n(n - 1)}{2} + \frac{1}{2}(n - 1) = \frac{n^2 + n - 2}{4}$$

which is $O(n^2)$ and approximately one-half of the number of comparisons in the worst case.

Activate Window

Shell Sort

- Improvement of Insertion sort.
- The idea is to start with comparing elements which are distant apart, which exist at a gap, and are not adjacent.
- Then gradually decrease the size of the gap and make comparisons.
- Finally, make comparisons of adjacent elements, just like Insertion sort.
- The complexity of worst case is O(n^2).
- The average case depends on the gap but can be better than O(n^2).

Shell Sort Example

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
Input data	62	83	18	53	07	17	95	86	47	69	25	28
After 5-sorting	17	28	18	47	07	25	83	86	53	69	62	95
After 3-sorting	17	07	18	47	28	25	69	62	53	83	86	95
After 1-sorting	07	17	18	25	28	47	53	62	69	83	86	95

Quick Sort

- Quick Sort is a divide and conquer algorithm.
- First, choose a pivot value (preferably the last element of the array).
- Then, arrange the smaller values towards the left side of the pivot and higher values towards the right side of the pivot.
- This operation is then repeated recursively for the two subarrays, 1) array of numbers less than pivot and 2) array of numbers greater than pivot.
- It continues until there is only one element left in an array.
- It is an in-place sorting algorithm (no extra space needed).

Quick Sort

- Best case complexity is O(nlogn), worst case is O(n^2), and the average case is O(nlogn).
- An advantage of quick sort is that we can almost always avoid worst case and achieve average case performance.
- It makes quick sort a practically useful sorting algorithm compared to previously seen algorithms.
- Most of the programming language libraries, implement default sorting using quick sort.

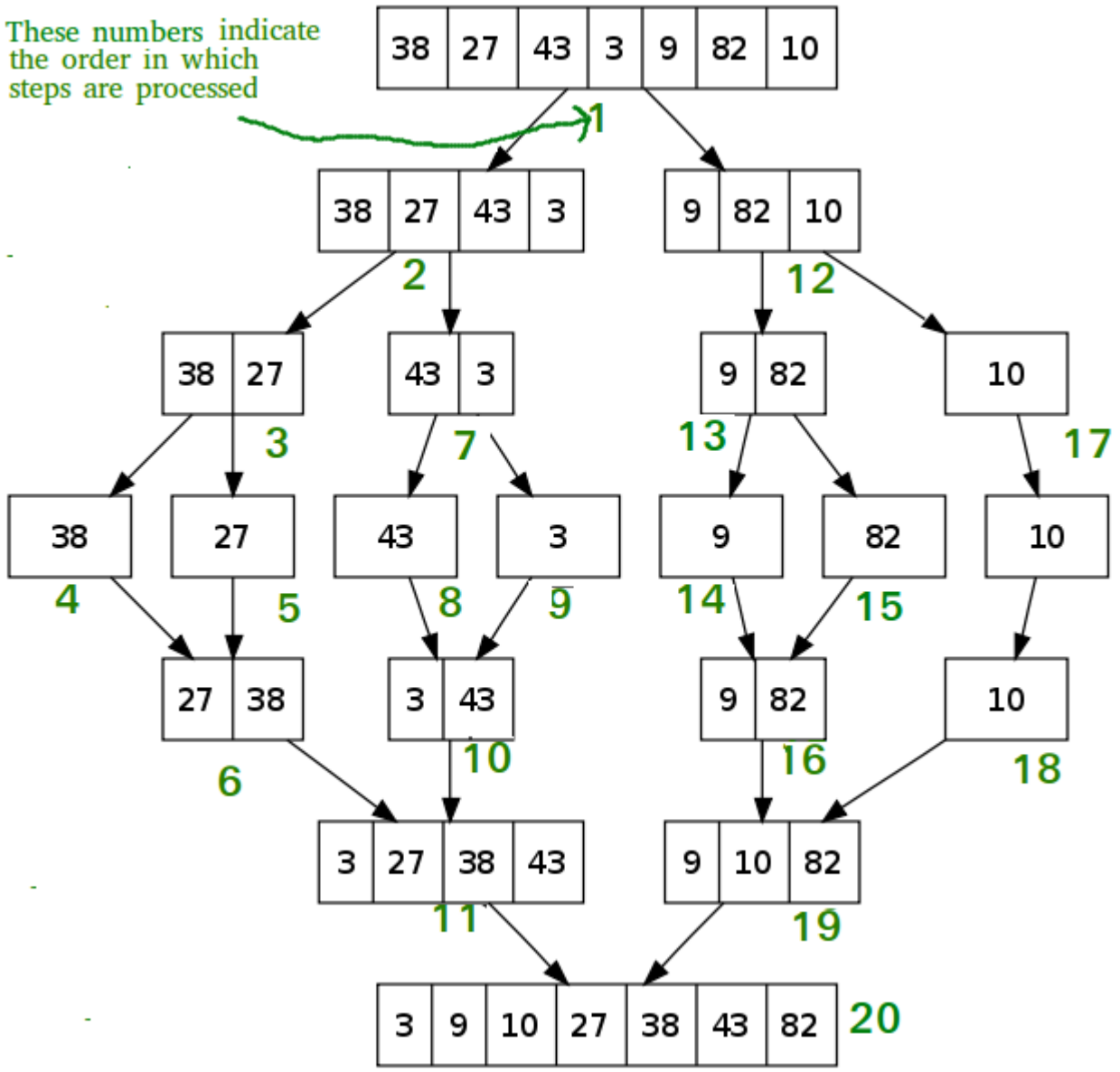
Merge Sort

- It is based on the divide and conquer technique.
- Divide an array from the middle into two subarrays (left and right).
- Keep dividing each subarray into two more subarrays recursively, until each subarray is of size one (base condition).
- Then, each of these one element based array is already sorted.
- Now, start combining left and right subarrays, until single sorted array is achieved.

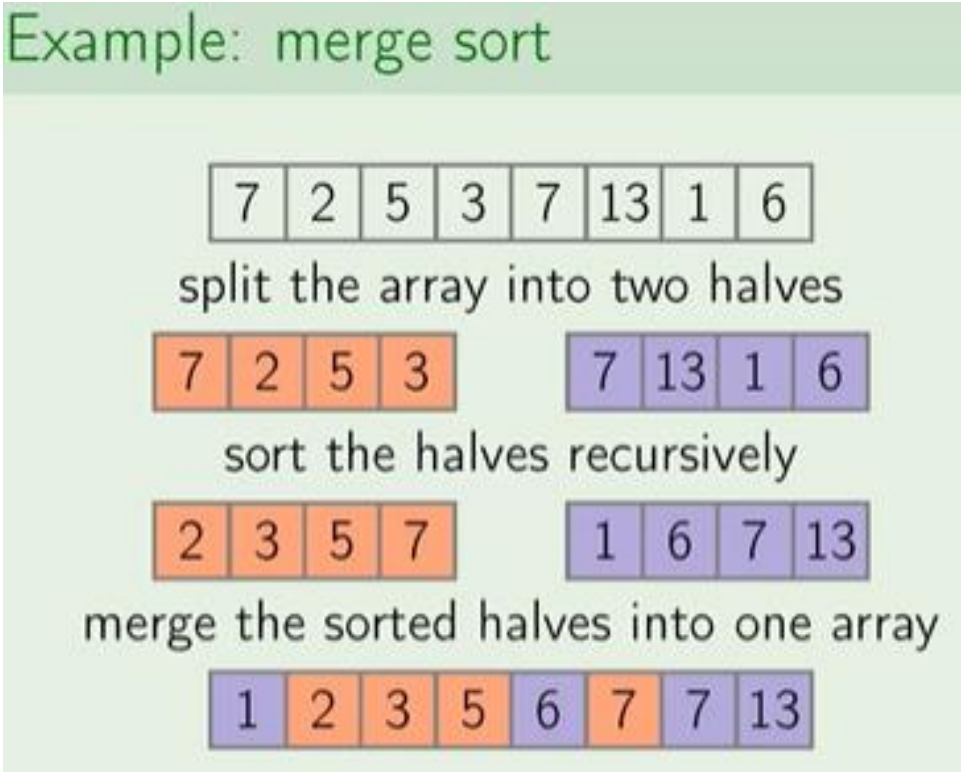
Merge Sort

- Time complexity of Merge Sort is O(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
- However, it requires an additional memory space of O(n) for the temporary array.

Merge Sort Example



Merge Sort Example



Distribution Counting Sort

- The main idea is to count the number of times each element occurs and use those counts to find the right position of the element in the sorted array.
- It is non-comparison based algorithm.
- It assumes that each of the elements is an integer in the range 1 to *K*, for some integer *K.*
- Find the max of given array.
- Make a temporary count array of size max+1.
- Fill the count array with the count of elements in the given array.

Distribution Counting Sort

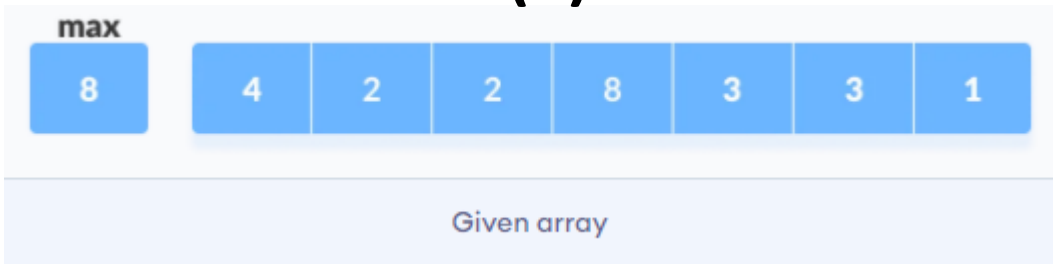
- Treat indexes of the count array as the elements in the given array. For example, if one element is 4 in the given array then the number of times 4 occurs in the given array is stored at index ‘4’ in the count array.
- Compute the cumulative counts in the count array. It directly tells the number of elements in the given array which should be placed before an element (which is the index of the count array).
- Use the cumulative counts to find the right position for each element in the given array.

Distribution Counting Sort

- It is efficient if the range of input digits is not significantly greater than the number of elements in the given array.
- Its average, best and worst time complexity is O(n+K) where n is the number of elements in the given array and K is the range of inputs.
- For example, if the range of inputs is from 10 to 10,000 and the array is {10,20,8000,10000}, then the time complexity will be high.
- Counting sort can be extended to work for negative inputs also (find the minimum element and store count of that minimum element at zero index).
- It is often used as a sub-routine to another sorting algorithm like Radix sort.

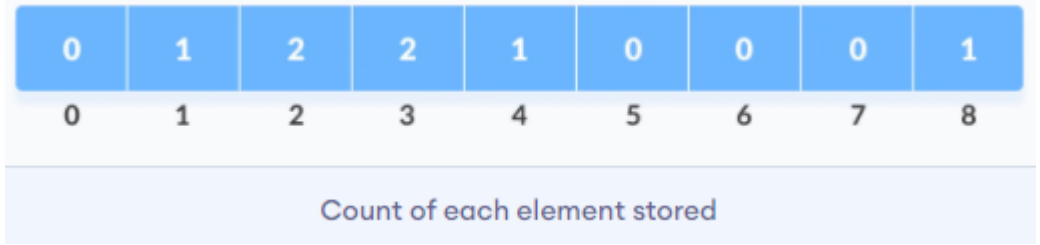
Distribution Counting Sort

(1)

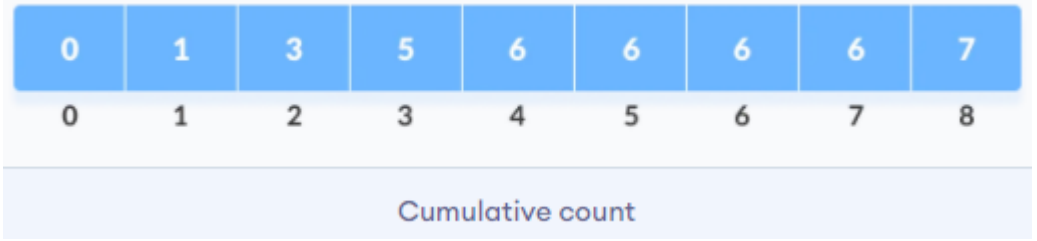


(2)

(3)



(4)

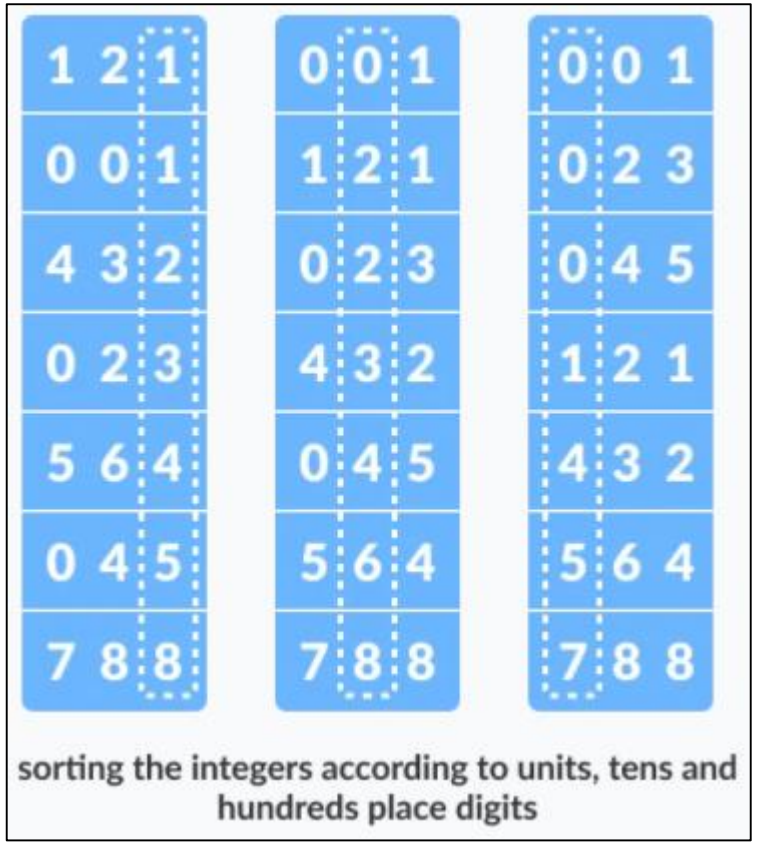


(5)

Radix Sort

- Like Counting Sort, it is a non-comparative sorting algorithm.
- It sorts by creating and distributing elements into buckets according to their radix.
- In a positional numeral system, the radix or base is the number of unique digits, including the digit zero, used to represent numbers. For example, for the decimal/denary system (the most common system in use today) the radix (base number) is ten, because it uses the ten digits from 0 through 9. (Wikipedia)
- The runtime for radix sort is O(nk), where n is the length of the array, k is the maximum number of digits.
- Both Count and Radix sort are **stable sort**, which means they preserve the relative order of elements that have the same key value.

Radix Sort Example



sorting the integers according to units, tens and hundreds place digits

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives:

[*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives:

[*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by the most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802