Radix sort

- Radix Sort is a non-comparative integer sorting algorithm that sorts numbers by processing individual digits.
- It works by sorting the numbers digit by digit, starting from the least significant digit to the most significant digit (LSD Radix Sort) or from the most significant digit to the least significant digit (MSD Radix Sort).

Radix sort

- Algorithm

radixSort(arr)

1. max = largest element in the given array
2. d = number of digits in the largest element (or, max)
3. Now, create d buckets of size 0 - 9
4. **for** i -> 0 to d
5. sort the array elements using counting sort (or any stable sort) according to the digits at the i^{th} place

Radix sort

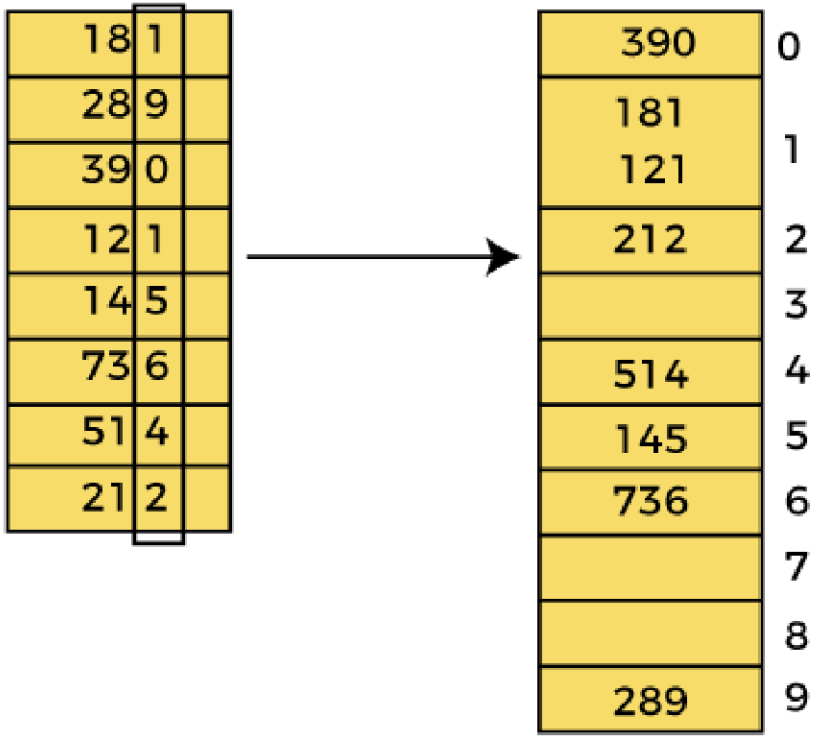
181	289	390	121	145	736	514	212

In the given array, the largest element is **736** that have **3** digits in it. So, the loop will run up to three times (i.e., to the **hundreds place**). That means three passes are required to sort the array.

Now, first sort the elements on the basis of unit place digits (i.e., **x = 0**). Here, we are using the counting sort algorithm to sort the elements.

Radix sort

- Pass 1:
- In the first pass, the list is sorted on the basis of the digits at 0's place.



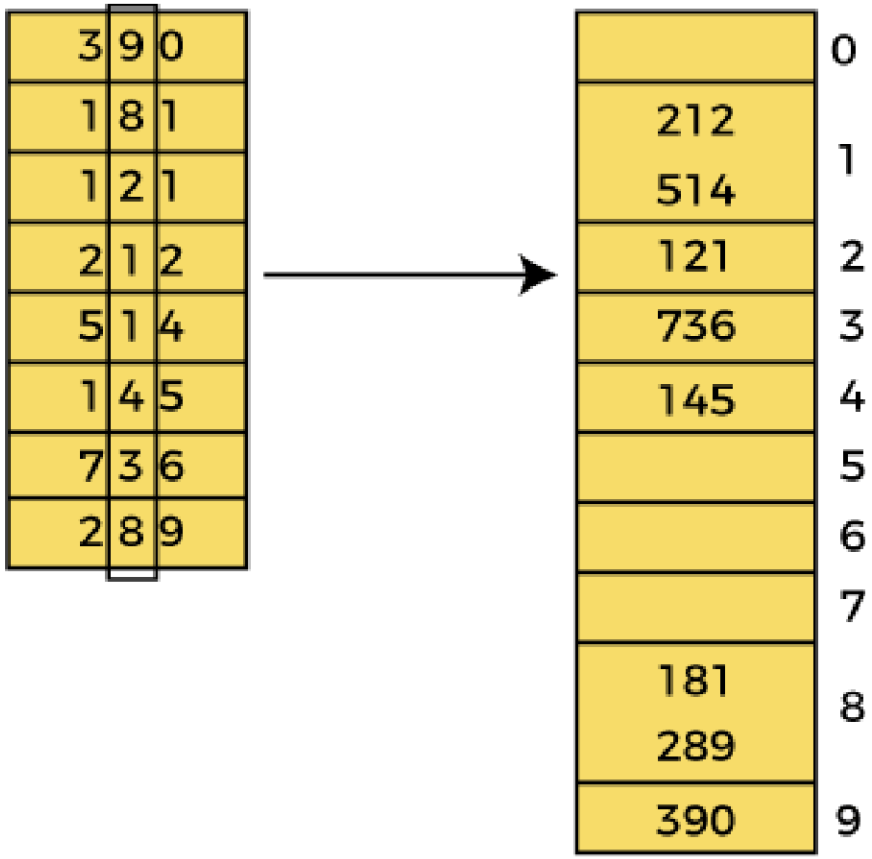
Radix sort

After the first pass, the array elements are –

390	181	121	212	514	145	736	289

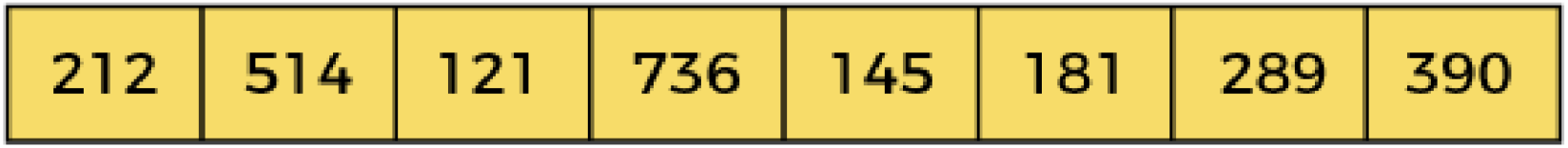
Radix sort

- Pass 2:
- In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10^{th} place).



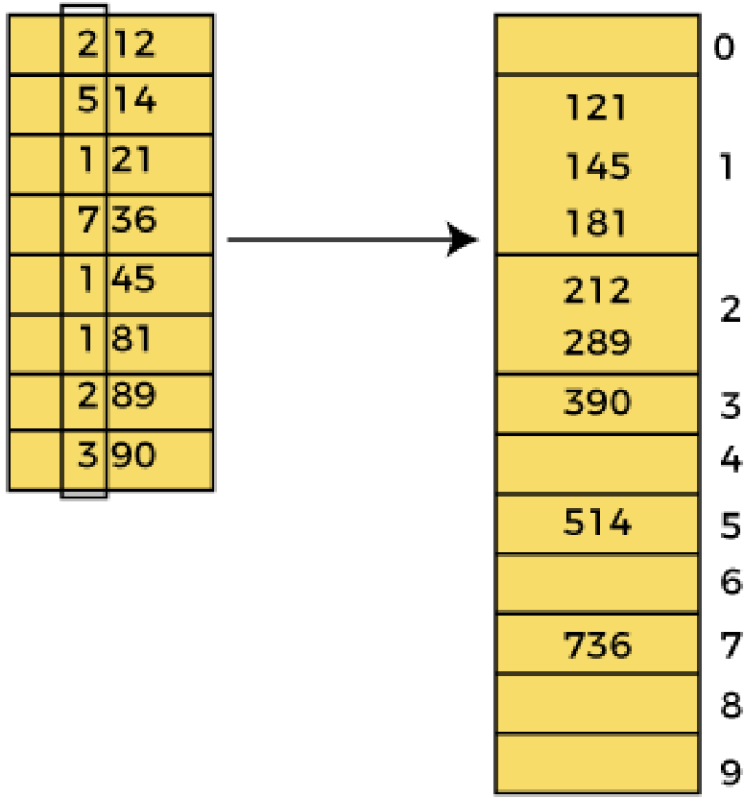
Radix sort

After the second pass, the array elements are –



Radix sort

- Pass 3:
- In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100^{th} place).



Radix sort

After the third pass, the array elements are –

121	145	181	212	289	390	514	736

Now, the array is sorted in ascending order.

Radix sort

Time complexity:

n is the number of elements, and k is the range of the digits (10 for decimal numbers).

The number of digits in the maximum number determines the number of passes we need to perform Sort.

Radix sort

Overall Time Complexity

Combining these, the overall time complexity of Radix Sort is:

- $O(d \times (n+k))$
- For decimal numbers, k is 10 (the range of digits 0-9). Therefore, the time complexity becomes:
- $O(d \times (n+10))$
- $O(d \times n)$ (since 10 is a constant and can be ignored in Big-O notation)

Bitonic sort

- Bitonic Sort is a parallel algorithm for sorting that is particularly well-suited for implementation on parallel computing architectures.
- It works by recursively constructing bitonic sequences (sequences that first increase then decrease, or vice versa) and then merging them into a sorted sequence.

Bitonic sort

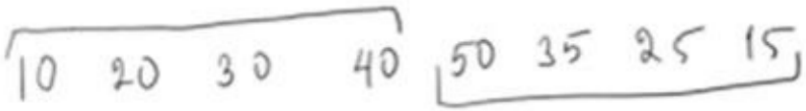
BITONIC SORTING

- Bitonic sequence is an array A[0 … (n-1)] if there is an index value i within the range 0 to n-1. For which the value of A[i] is greatest in the array. i.e.

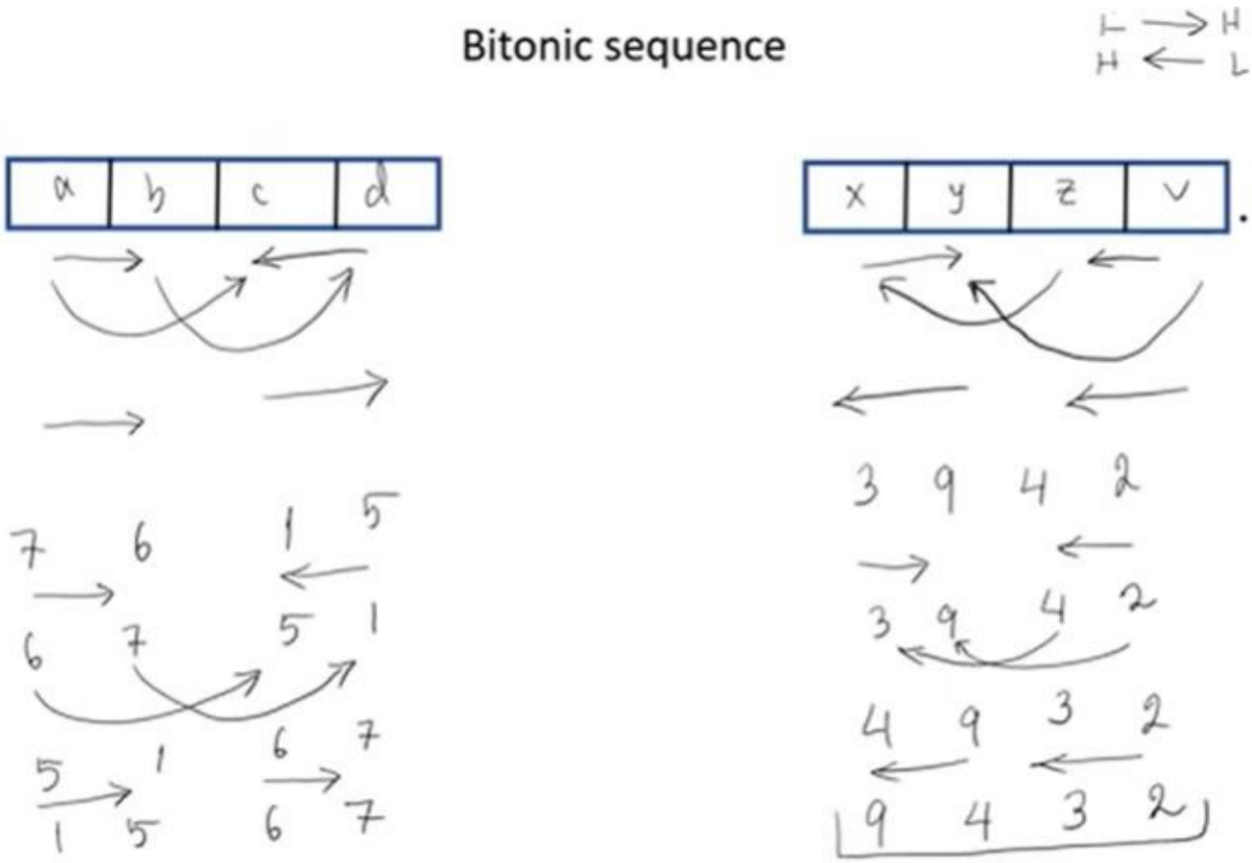
$$A[0] <= A[1] \ldots <= A[i] \text{ and } A[i] >= A[i+1] \ldots >= A[n-1]$$

❖ Bitonic sequence can be rotated back to bitonic sequence.

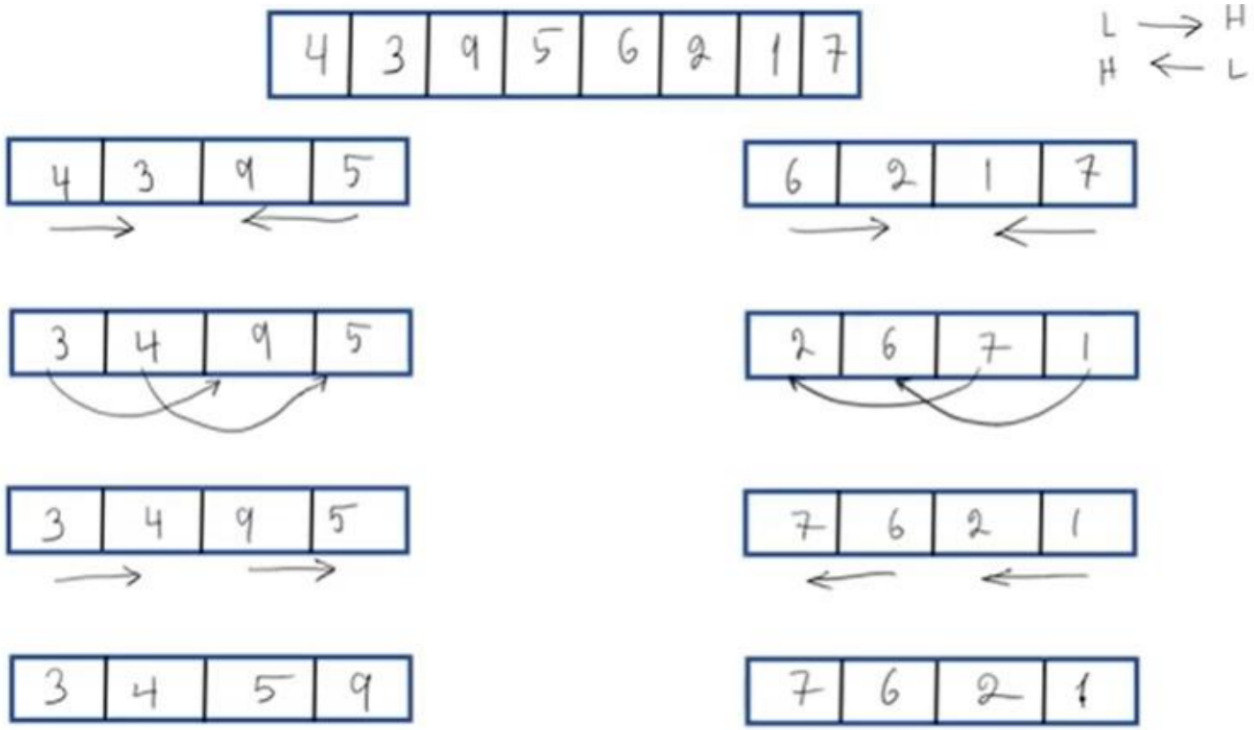
❖ A sequence with elements in increasing and decreasing order is a bitonic sequence.



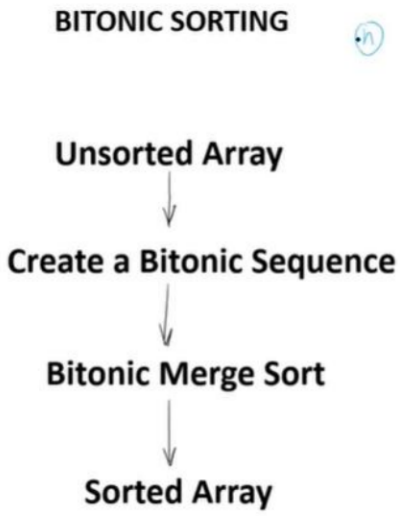
Bitonic sort



Bitonic sort



Bitonic sort



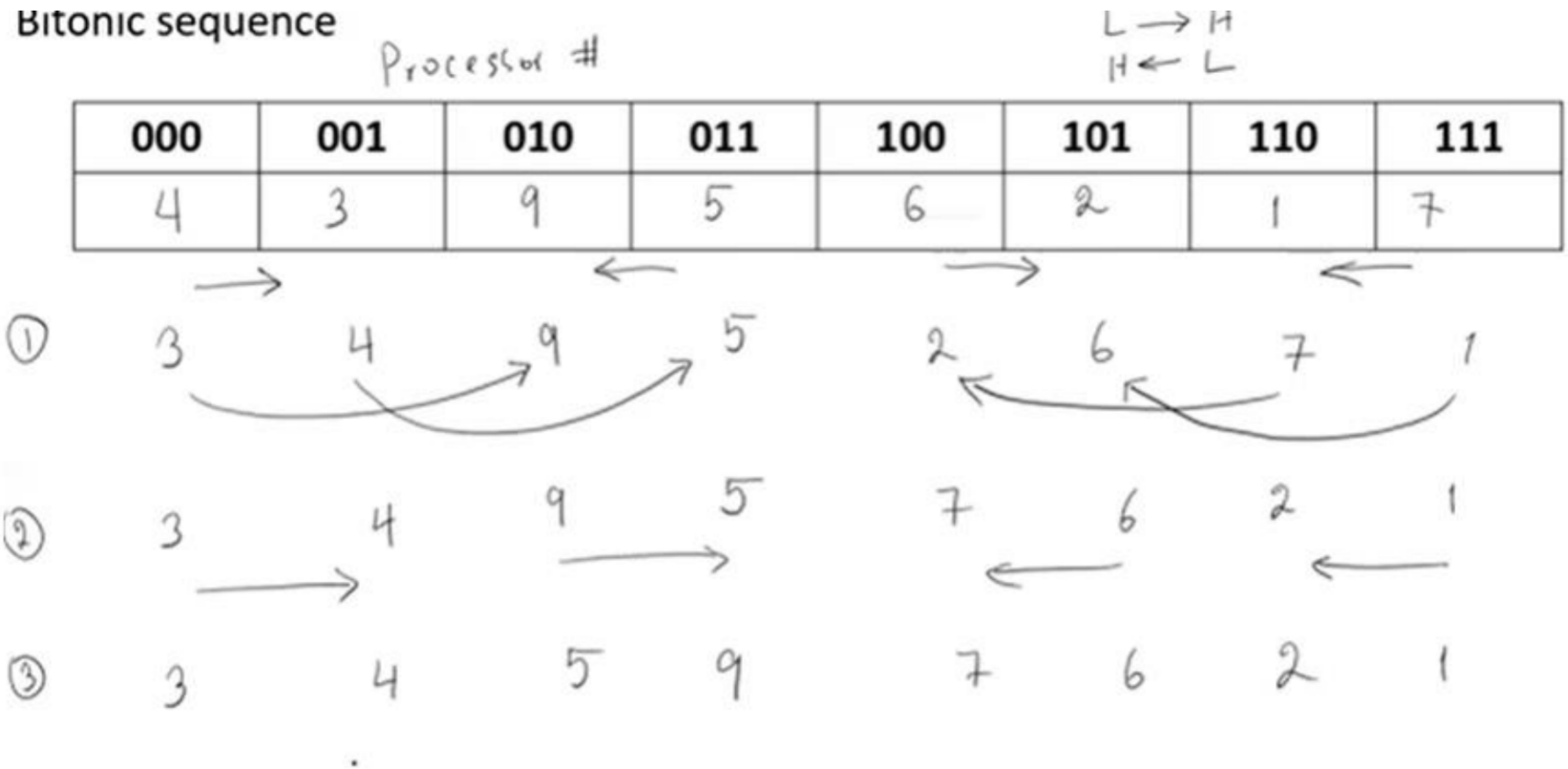
BITONIC SORTING

Steps in bitonic sorting:

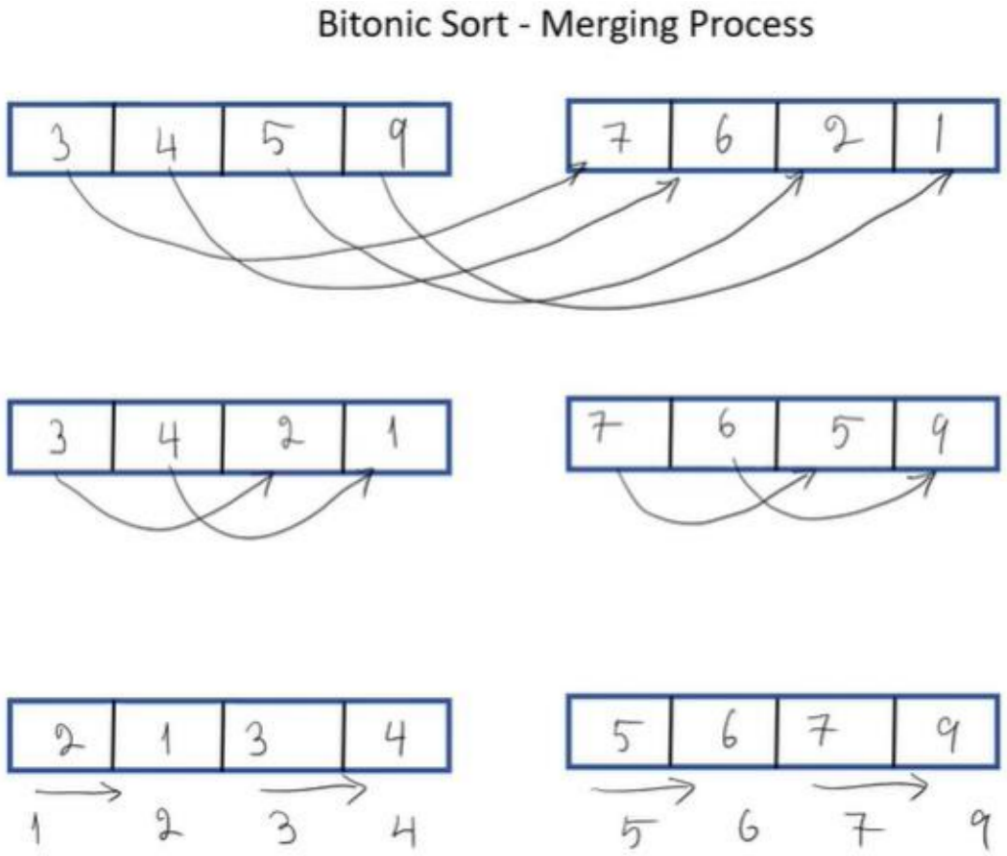
- Create a bitonic sequence.
  (one part in increasing order and others in decreasing order.)
- Compare and swap the first elements of both halves. Then second, third, fourth elements for them.
- We will compare and swap, every second element of the sequence.
- Compare and swap adjacent elements of the sequence.

After all swaps, we will get the sorted array.

Bitonic sort



Bitonic sort



Bitonic sort

- Time complexity:

1. Bitonic Sequence Construction:

In this phase, the array is recursively divided into smaller subarrays and then these subarrays are arranged into bitonic sequences. This process takes $O(\log n)$ steps because each step divides the array into halves.

2. Bitonic Merge:

- Each bitonic merge step involves comparing and swapping elements to create a sorted sequence from a bitonic sequence.
- The bitonic merge process also takes $O(\log n)$O steps.
- Each comparison and swap operation involves a linear scan of the current subarray, which takes $O(n)$ time in total for each level of recursion.

Bitonic sort

- Since both the bitonic sequence construction and the bitonic merge take O(logn) steps, and each step involves O(n) work, the overall time complexity is:
- $O(n \log n \log n) = O(n \log^2 n)$

Summary

- **Worst-case time complexity**: $O(n \log^2 n)$
- **Best-case time complexity**: $O(n \log^2 n)$
- **Average-case time complexity**: $O(n \log^2 n)$

Cocktail sort

- Cocktail Sort, also known as Bidirectional Bubble Sort or Shaker Sort, is a variation of Bubble Sort.
- It improves on Bubble Sort by moving in both directions during the sort pass, which helps to bubble the largest unsorted element to its correct position more quickly.

Cocktail sort

Cocktail Sort Algorithm

The algorithm consists of two main phases in each pass:

1. **Forward Pass**: Move the largest unsorted element to the end of the list.
2. **Backward Pass**: Move the smallest unsorted element to the beginning of the list.

This bidirectional movement helps to reduce the number of passes needed compared to traditional Bubble Sort, especially when the list is nearly sorted.

Cocktail sort

- Cocktail Sort is a variation of Bubble sort.
- The Bubble sort algorithm always traverses elements from left and moves the largest element to its correct position in the first iteration and second-largest in the second iteration and so on.
- Cocktail Sort traverses through a given array in both directions alternatively. Cocktail sort does not go through the unnecessary iteration making it efficient for large arrays.

Cocktail sort

Algorithm:

- Each iteration of the algorithm is broken up into 2 stages:

1. The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if the value on the left is greater than the value on the right, then values are swapped. At the end of the first iteration, the largest number will reside at the end of the array.

2. The second stage loops through the array in opposite direction- starting from the item just before the most recently sorted item, and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required.

Cocktail sort

- **Example :**
- Let us consider an example array (5 1 4 2 8 0 2)
- **First Forward Pass:**
  (**5 1** 4 2 8 0 2) ? (**1 5** 4 2 8 0 2), Swap since 5 > 1
  (1 **5 4** 2 8 0 2) ? (1 **4 5** 2 8 0 2), Swap since 5 > 4
  (1 4 **5 2** 8 0 2) ? (1 4 **2 5** 8 0 2), Swap since 5 > 2
  (1 4 2 **5 8** 0 2) ? (1 4 2 **5 8** 0 2)
  (1 4 2 5 **8 0** 2) ? (1 4 2 5 **0 8** 2), Swap since 8 > 0
  (1 4 2 5 0 **8 2**) ? (1 4 2 5 0 **2 8**), Swap since 8 > 2
  After the first forward pass, the greatest element of the array will be present at the last index of the array.

Cocktail sort

First Backward Pass:

(1 4 2 5 **0 2** 8) ? (1 4 2 5 **0 2** 8)

(1 4 2 **5 0** 2 8) ? (1 4 2 **0 5** 2 8), Swap since 5 > 0

(1 4 **2 0** 5 2 8) ? (1 4 **0 2** 5 2 8), Swap since 2 > 0

(1 **4 0** 2 5 2 8) ? (1 **0 4** 2 5 2 8), Swap since 4 > 0

(**1 0** 4 2 5 2 8) ? (**0 1** 4 2 5 2 8), Swap since 1 > 0

After the first backward pass, the smallest element of the array will be present at the first index of the array.

Cocktail sort

Second Forward Pass:

(0 **1 4** 2 5 2 8) ? (0 **1 4** 2 5 2 8)

(0 1 **4 2** 5 2 8) ? (0 1 **2 4** 5 2 8), Swap since 4 > 2

(0 1 2 **4 5** 2 8) ? (0 1 2 **4 5** 2 8)

(0 1 2 4 **5 2** 8) ? (0 1 2 4 **2 5** 8), Swap since 5 > 2

Cocktail sort

Second Backward Pass:

(0 1 2 **4 2** 5 8) ? (0 1 2 **2 4** 5 8), Swap since 4 > 2

Cocktail sort

Now, the array is already sorted, but our algorithm doesn't know if it is completed. The algorithm needs to complete this whole pass without any **swap** to know it is sorted.
(0 1 **2 2** 4 5 8) ? (0 1 **2 2** 4 5 8)
(0 **1 2** 2 4 5 8) ? (0 **1 2** 2 4 5 8)

Cocktail sort

- **The basic idea of cocktail sort is as follows:**

1. Start from the beginning of the array and compare each adjacent pair of elements. If they are in the wrong order, swap them.
2. Continue iterating through the array in this manner until you reach the end of the array.
3. Then, move in the opposite direction from the end of the array to the beginning, comparing each adjacent pair of elements and swapping them if necessary.
4. Continue iterating through the array in this manner until you reach the beginning of the array.
5. Repeat steps 1-4 until the array is fully sorted.

Cocktail sort-properties

- Sorting In Place: Yes
- Stable: Yes