Other Sorting Algorithms

SWE2016-44

Heap Sort

- **Heap Sort:** a comparison based sorting technique based on Binary Heap data structure
  - First find the maximum element and place the maximum element at the end. Repeat the same process for remaining element.
  - **Binary Heap**: a **Complete Binary Tree** where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. → Max Heap (Min Heap)

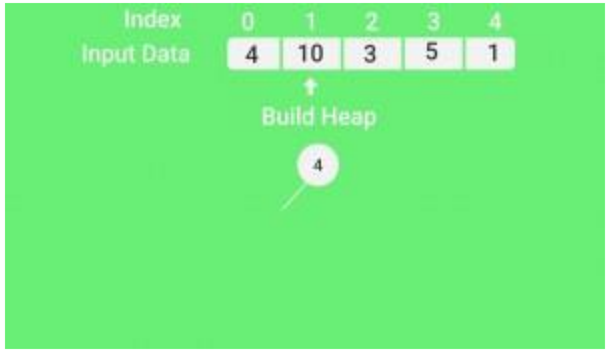
```
         10                     10
       /    \                 /      \
     20      100            15        30
    /                      /  \      /  \
   30                     40   50   100  40
```

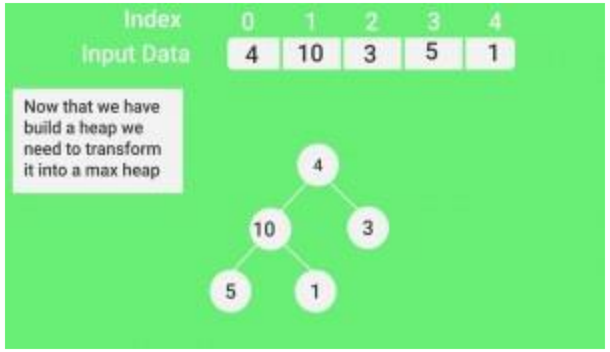
→ Min Heap

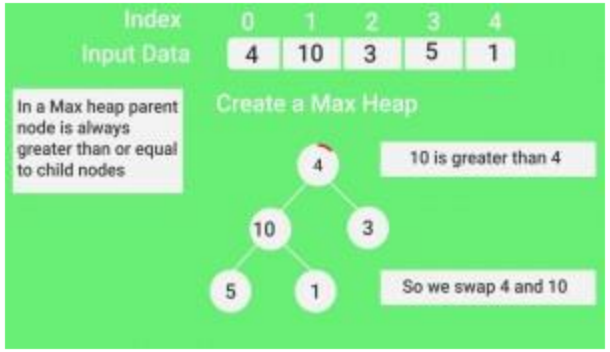
  - If the parent node is stored at index i, the left child can be calculated by 2i+1 and right child by 2i+2 (assuming the indexing starts at 0).

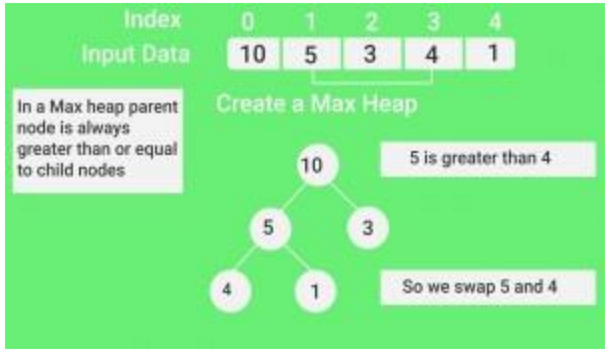
Heap Sort

- Heap Sort Algorithm for sorting in increasing order
  1. Build a max heap from the input data.
  2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
  3. Repeat above steps while size of heap is greater than 1.



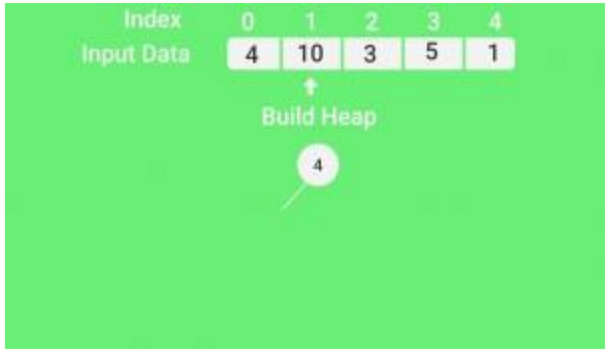


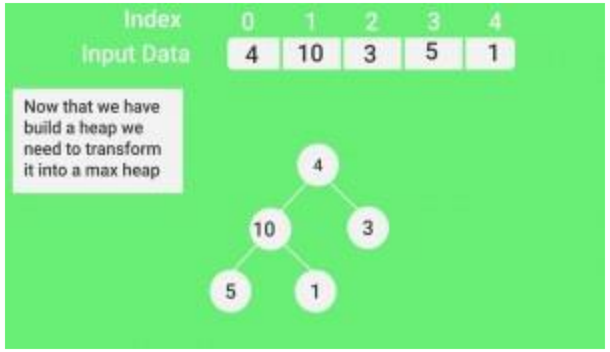


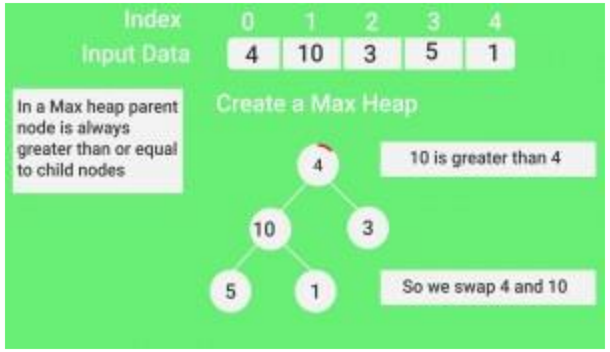


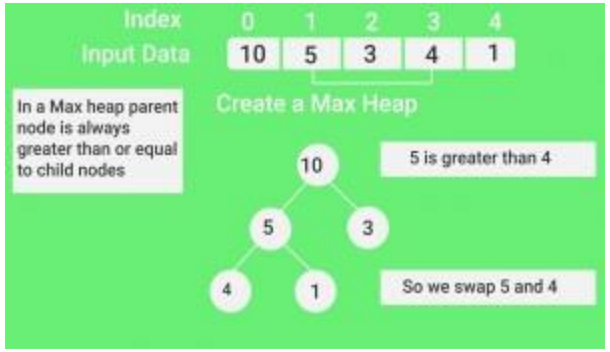
Heap Sort

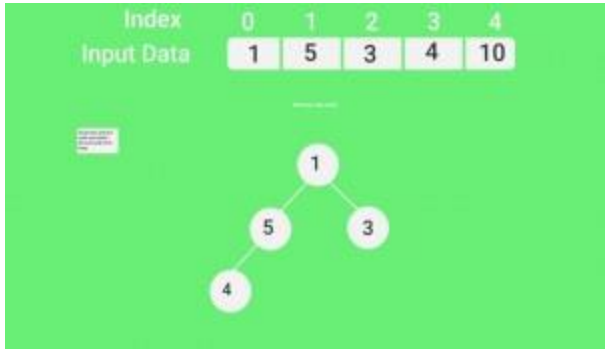
- Heap Sort Algorithm for sorting in increasing order
    1. Build a max heap from the input data.
    2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
    3. Repeat above steps while size of heap is greater than 1.





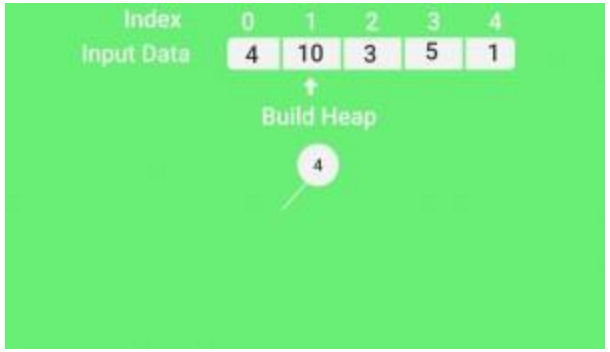


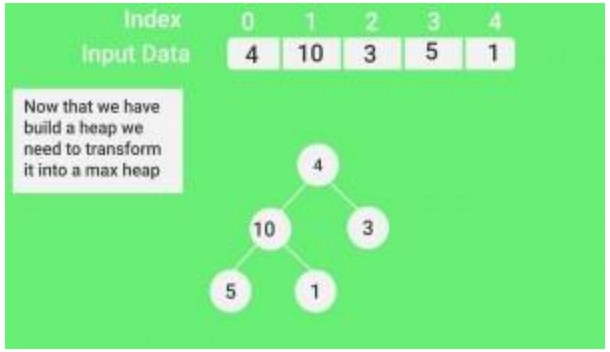


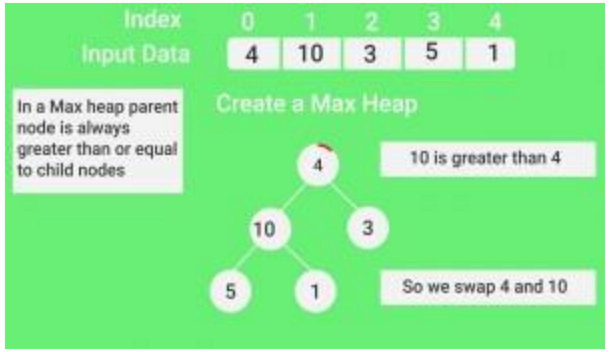


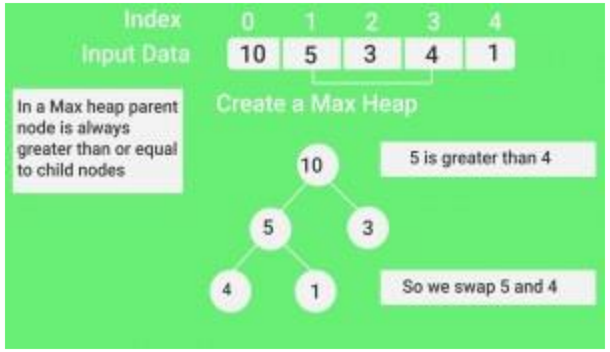
Heap Sort

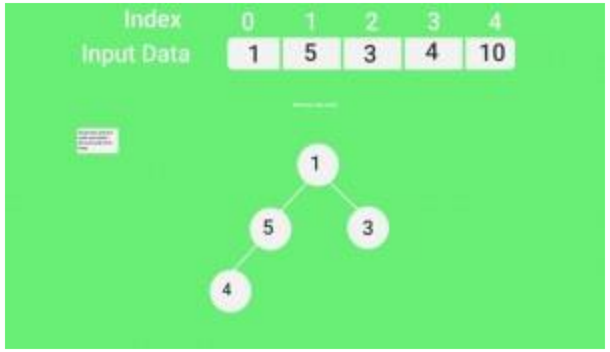
- Heap Sort Algorithm for sorting in increasing order
    1. Build a max heap from the input data.
    2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
    3. Repeat above steps while size of heap is greater than 1.

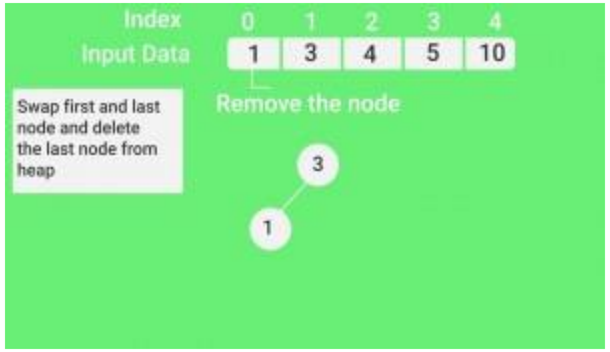






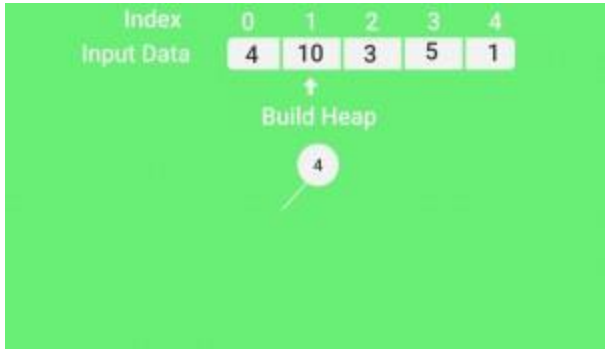


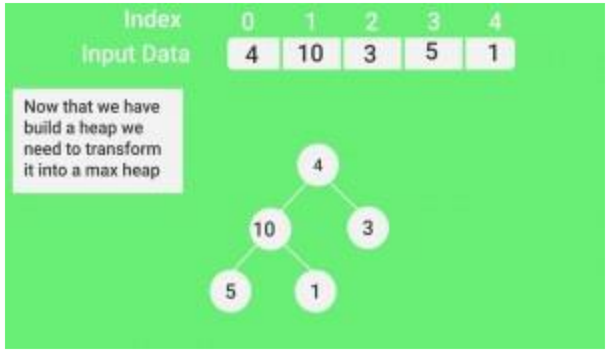


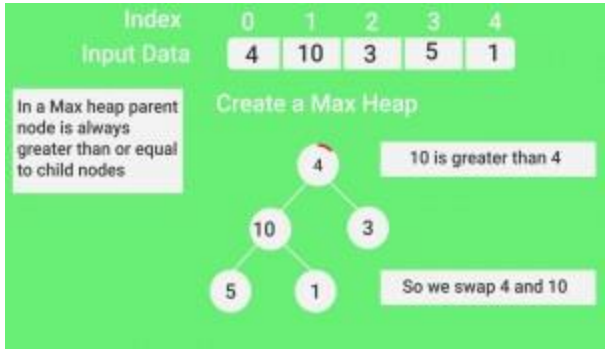


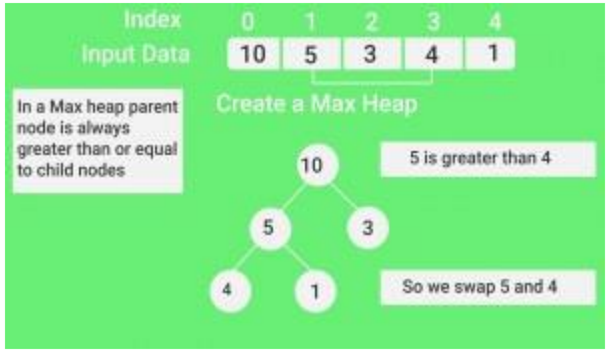
Heap Sort

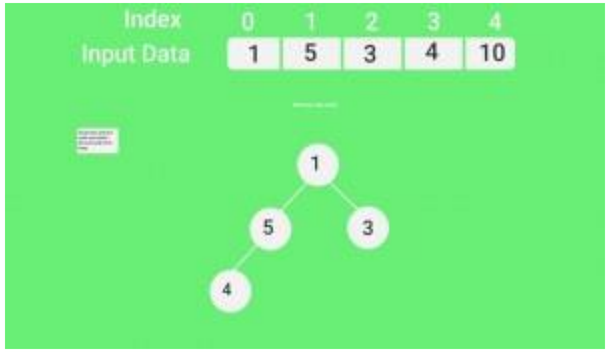
- Heap Sort Algorithm for sorting in increasing order
  1. Build a max heap from the input data.
  2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
  3. Repeat above steps while size of heap is greater than 1.

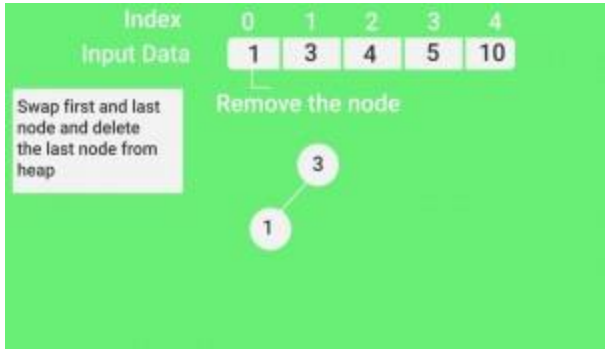












- Time complexity of heapify: O(Logn)
- Time complexity of createAndBuildHeap(): O(n)

→ **Overall time complexity of Heap Sort: O(nLogn)**

Selection Sort

- Sort an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
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
```

→ **Time Complexity:** $O(n^2)$ as there are two nested loops

Counting Sort

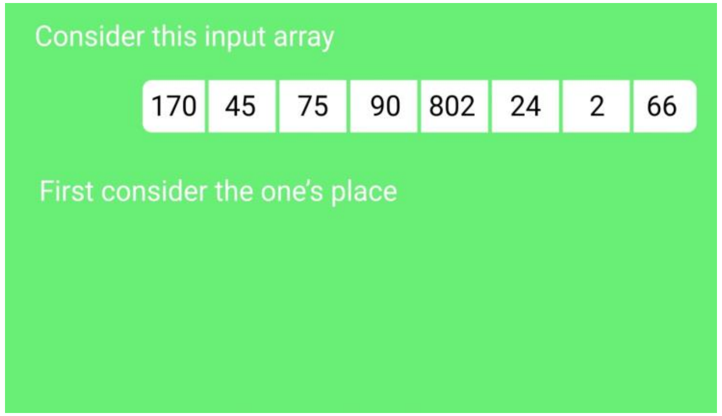
- **Counting Sort**: A sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing).

```
For simplicity, consider the data in the range 0 to 9.
Input data: 1, 4, 1, 2, 7, 5, 2
  1) Take a count array to store the count of each unique object.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  2  0   1  1  0  1  0  0

  2) Modify the count array such that each element at each index
  stores the sum of previous counts.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  4  4  5  6  6  7  7  7

The modified count array indicates the position of each object in
the output sequence.

  3) Output each object from the input sequence followed by
  decreasing its count by 1.
  Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.
  Put data 1 at index 2 in output. Decrease count by 1 to place
  next data 1 at an index 1 smaller than this index.
```

→ **Time Complexity:** O(n+k) where n is the number of elements in input array and k is the range of input.

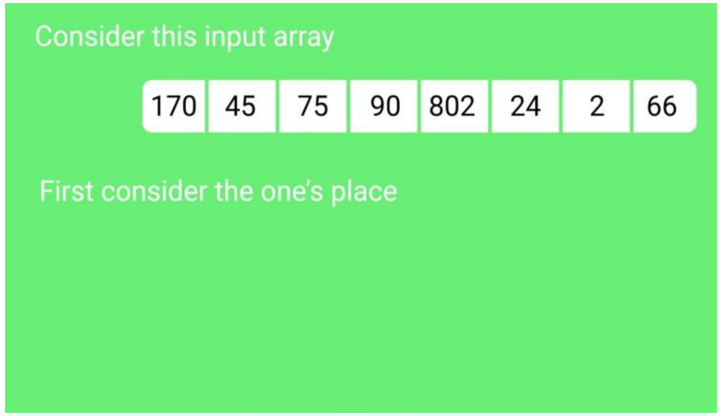
Radix Sort

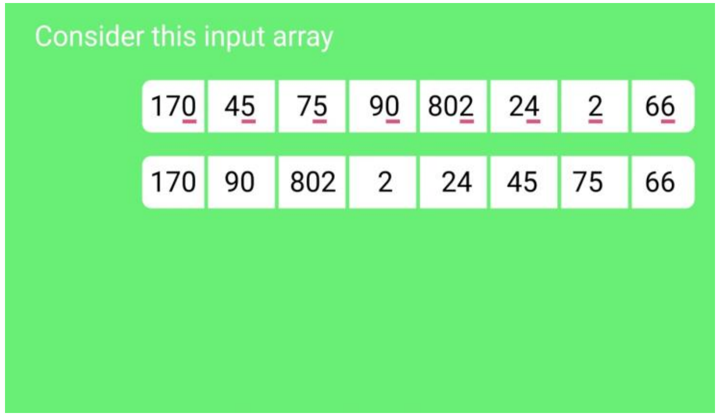
- The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.
  - Sort input array using counting sort (or any stable sort) according to the i'th digit.



Radix Sort

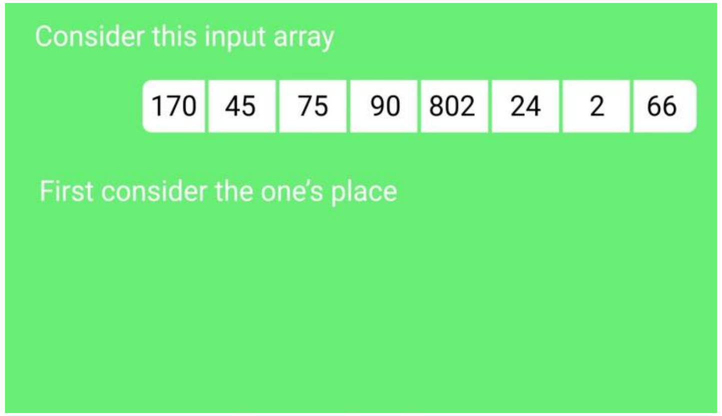
- The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.
  - Sort input array using counting sort (or any stable sort) according to the i’th digit.

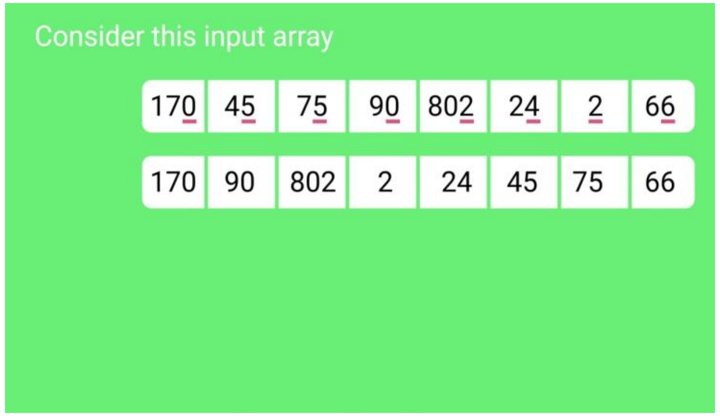


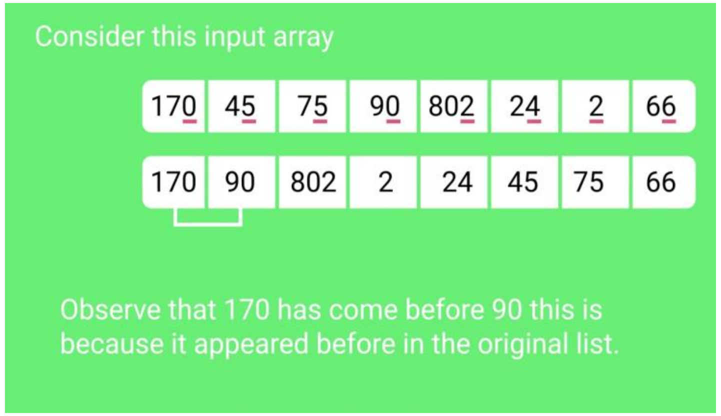


Radix Sort

- The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.
  - Sort input array using counting sort (or any stable sort) according to the i’th digit.

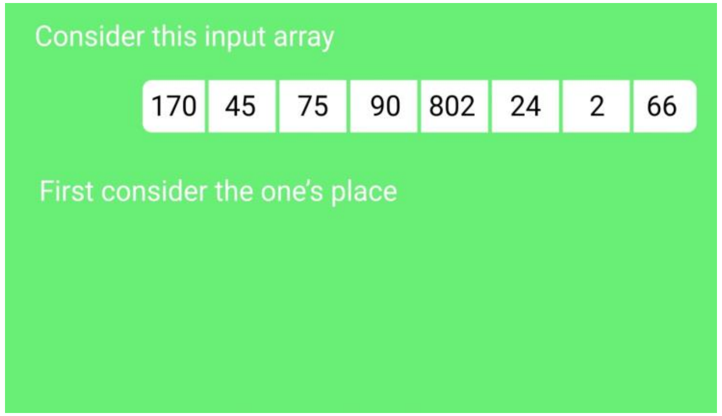


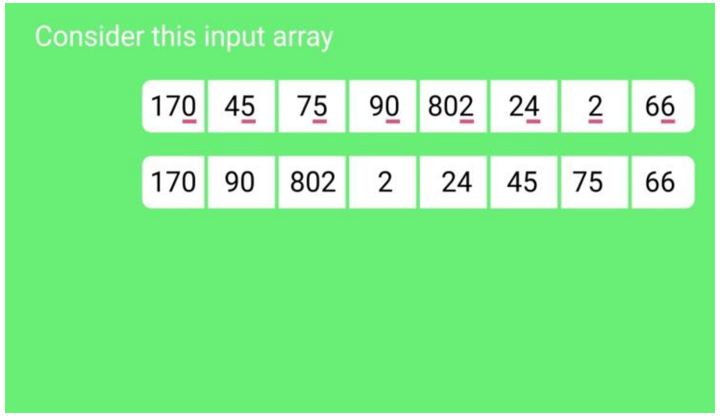


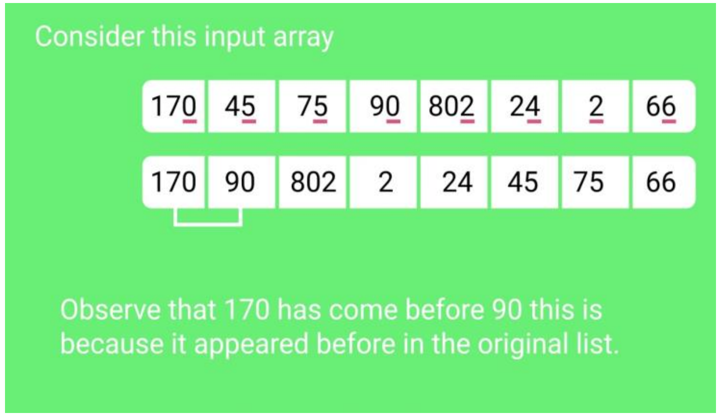


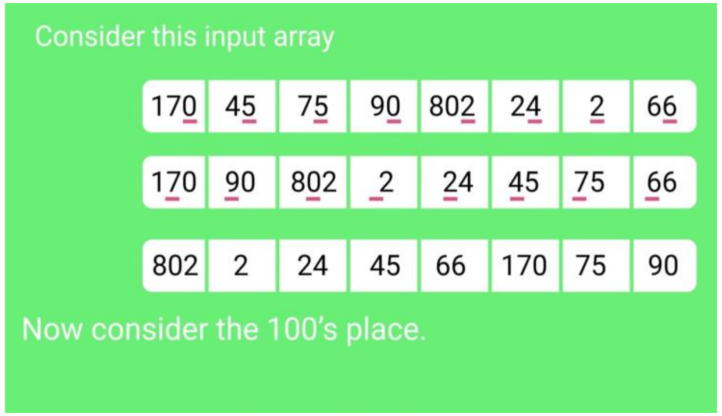
Radix Sort

- The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.
  - Sort input array using counting sort (or any stable sort) according to the i’th digit.



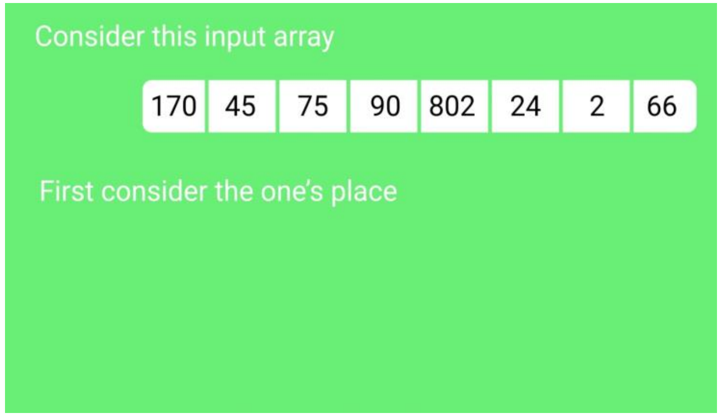


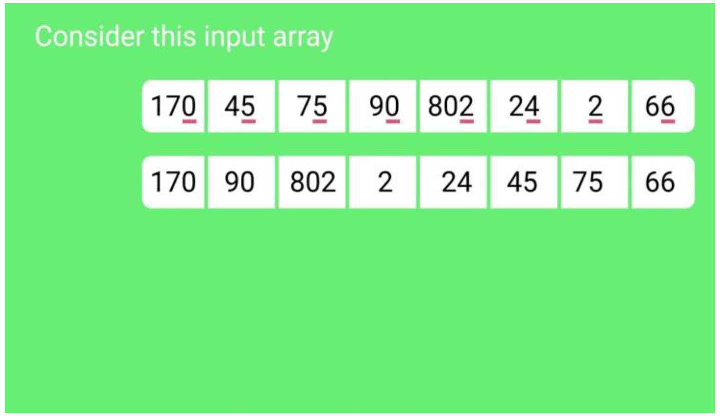


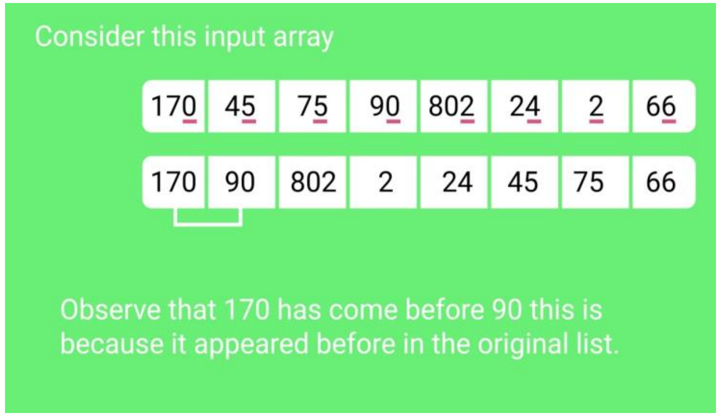


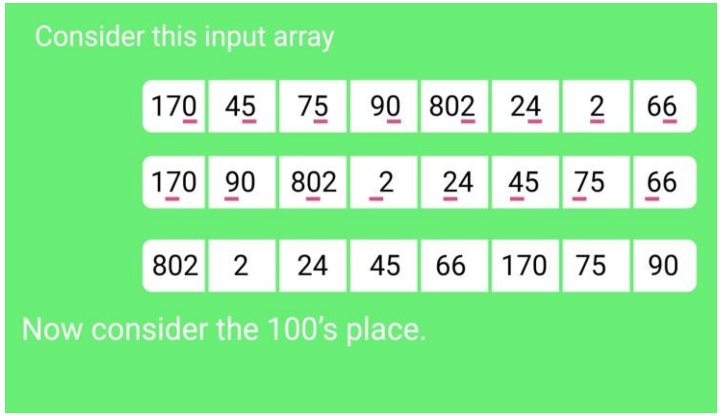
Radix Sort

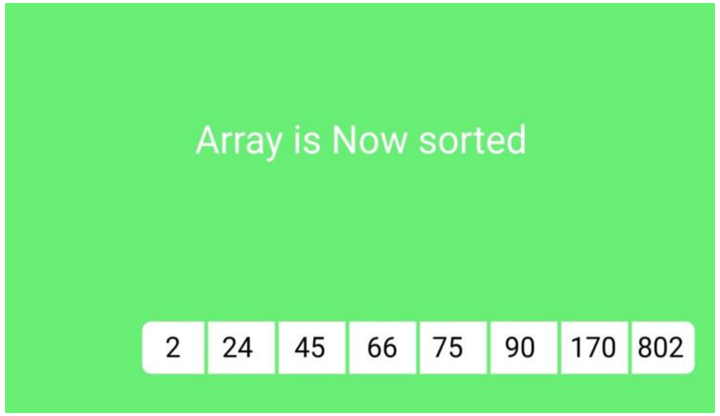
- The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.
  - Sort input array using counting sort (or any stable sort) according to the i’th digit.











Radix Sort

- The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.
  - Sort input array using counting sort (or any stable sort) according to the i'th digit.

Time Complexity Analysis

Let there be $\boldsymbol{d}$ digits in input integers.

→ $\mathbf{O}\big(\boldsymbol{d}(\boldsymbol{n}+\boldsymbol{b})\big)$ time where $\boldsymbol{b}$ is the base for representing numbers (for example, for decimal system, $\boldsymbol{b}=\mathbf{10}$)

→ If $\boldsymbol{k}$ is the maximum possible value, then $\boldsymbol{d} \approx \boldsymbol{O}(\mathbf{log}_{\boldsymbol{b}} \boldsymbol{k})$ → overall time complexity: $\mathbf{O}\big((\boldsymbol{n}+\boldsymbol{b})\,\mathbf{log}_{\boldsymbol{b}} \boldsymbol{k}\big)$.

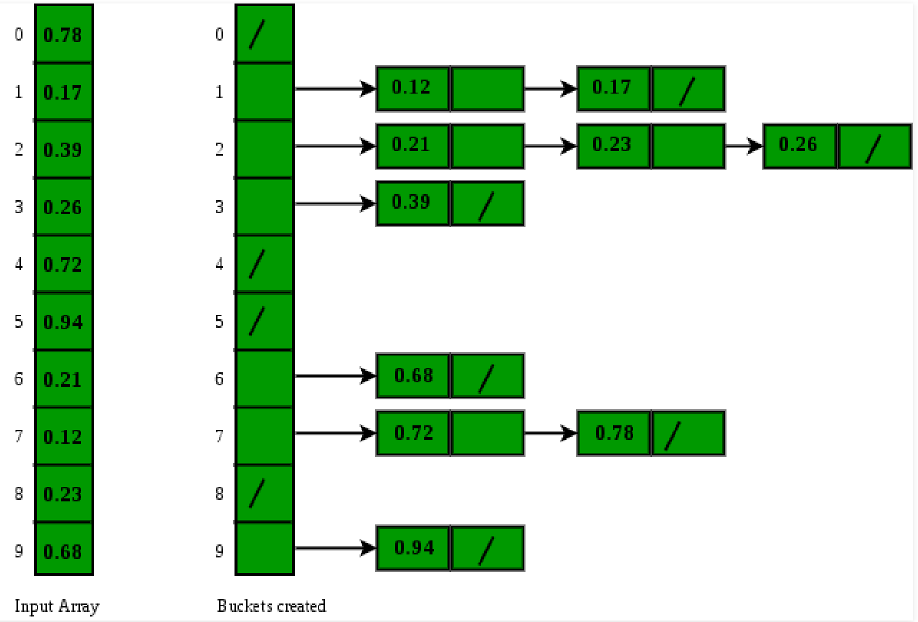
→ Let $\boldsymbol{k} \leq \boldsymbol{n}^{\boldsymbol{c}}$ where $\boldsymbol{c}$ is a constant. → complexity: $\mathbf{O}(\boldsymbol{n}\,\mathbf{log}_{\boldsymbol{b}} \boldsymbol{n})$

→ If we set $\boldsymbol{b}$ as $\boldsymbol{n}$, we get the time complexity as $\mathbf{O}(\boldsymbol{n})$. In other words, we can sort an array of integers with range from 1 to $\boldsymbol{n}^{\boldsymbol{c}}$ if the numbers are represented in base $\boldsymbol{n}$ (or every digit takes $\mathbf{log}_2 \boldsymbol{n}$ bits).

Bucket Sort

- Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem. Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range.

```
bucketSort(arr[], n)
1) Create n empty buckets (Or lists).
2) Do following for every array element arr[i].
.......a) Insert arr[i] into bucket[n*array[i]]
3) Sort individual buckets using insertion sort.
4) Concatenate all sorted buckets.
```



→ Time Complexity:

- If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket.
- Step 3 also takes O(n) time on average if all numbers are uniformly distributed
- Step 4 also takes O(n) time as there will be n items in all buckets.

Time Complexities of all Sorting Algorithms

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	θ(nk)	O(nk)