ÇUKUROVA UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

**Lecture 5: Sorting and Searching Algorithms**

## Sorting Algorithms

Two important classification criteria of sorting algorithms:

- Stable vs. unstable sorting
- In-place vs. out-of-place sorting

**Sorting Stability:**

- Stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Stable and unstable sorting procedures behave in the same way if there are no duplicates among the keys.
- A multitude of unique elements, for example, can be sorted with a stable or unstable sorting process; the result is always the same.
- Any sorting algorithm can be made stable by considering indexes as comparison parameter.

### Stable sorting process by numbers:

```
1 Anton         1 Anton
4 Karl          1 Paul
3 Otto          3 Otto
5 Bernd    →    3 Herbert
3 Herbert       4 Karl
8 Alfred        5 Bernd
1 Paul          8 Alfred
```

### Unstable sorting process by numbers:

```
1 Anton         1 Paul          1 Anton         1 Paul              1 Anton
4 Karl          1 Anton         1 Paul          1 Anton             1 Paul
3 Otto          3 Otto          3 Herbert       3 Herbert           3 Otto
5 Bernd    →    3 Herbert or    3 Otto     or   3 Otto         or   3 Herbert
3 Herbert       4 Karl          4 Karl          4 Karl              4 Karl
8 Alfred        5 Bernd         5 Bernd         5 Bernd             5 Bernd
1 Paul          8 Alfred        8 Alfred        8 Alfred            8 Alfred
```

If the sorting is unstable, Paul can stand in front of Anton or Herbert in front of Otto.

**In-Place Sorting Algorithm:**

An algorithm works **in-place** if, in addition to the memory required for storing the data to be processed, it only requires a constant amount of memory, i.e. independent of the amount of data to be processed. The algorithm overwrites the input data with the output data. Otherwise, the algorithm is called **out-of-place**, i.e. it requires additional storage space.

**<u>Bubble Sort:</u>**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

**Example:** Sort the following set of elements **-** (5 1 4 2 8)

**First Pass:**

( **5 1** 4 2 8 ) –> ( **1 5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.

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

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

( 1 4 2 **5 8** ) –> ( 1 4 2 **5 8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**

( **1 4** 2 5 8 ) –> ( **1 4** 2 5 8 )

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

( 1 2 **4 5** 8 ) –> ( 1 2 **4 5** 8 )

( 1 2 4 **5 8** ) –> ( 1 2 4 **5 8** )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**

( **1 2** 4 5 8 ) –> ( **1 2** 4 5 8 )

( 1 **2 4** 5 8 ) –> ( 1 **2 4** 5 8 )

( 1 2 **4 5** 8 ) –> ( 1 2 **4 5** 8 )

( 1 2 4 **5 8** ) –> ( 1 2 4 **5 8** )

**Insertion Sort:**

- A graphical example of insertion sort.
- The partial sorted list (black) initially contains only the first element in the list.
- With each iteration one element (red) is removed from the "not yet checked for order" input data and inserted in-place into the sorted list.

6 5 3 1 8 7 2 4

**Selection Sort:**

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

Let the elements of array are:

| | | | | | | |
|---|---|---|---|---|---|---|
| 12 | 29 | 25 | 8 | 32 | 17 | 40 |

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

| | | | | | | |
|---|---|---|---|---|---|---|
| 12 | 29 | 25 | 8 | 32 | 17 | 40 |

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

| | | | | | | |
|---|---|---|---|---|---|---|
| 8 | 29 | 25 | 12 | 32 | 17 | 40 |

For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

| | | | | | | |
|---|---|---|---|---|---|---|
| 8 | 29 | 25 | 12 | 32 | 17 | 40 |

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

| | | | | | | |
|---|---|---|---|---|---|---|
| 8 | 12 | 25 | 29 | 32 | 17 | 40 |

The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.

| | | | | | | |
|---|---|---|---|---|---|---|
| 8 | 12 | 25 | 29 | 32 | 17 | 40 |
| 8 | 12 | 25 | 29 | 32 | 17 | 40 |
| 8 | 12 | 17 | 29 | 32 | 25 | 40 |
| 8 | 12 | 17 | 29 | 32 | 25 | 40 |
| 8 | 12 | 17 | 29 | 32 | 25 | 40 |
| 8 | 12 | 17 | 25 | 32 | 29 | 40 |
| 8 | 12 | 17 | 25 | 32 | 29 | 40 |
| 8 | 12 | 17 | 25 | 32 | 29 | 40 |
| 8 | 12 | 17 | 25 | 29 | 32 | 40 |
| 8 | 12 | 17 | 25 | 29 | 32 | 40 |

Now, the array is completely sorted.

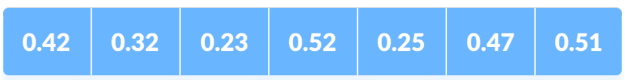
## Bucket Sort:

- Bucket Sort is a sorting technique that sorts the elements by first dividing the elements into several groups called **buckets**.
- Several buckets are created. Each bucket is filled with a specific range of elements.
- The elements inside each **bucket** are sorted using any of the suitable sorting algorithms or recursively calling the same algorithm.
- Finally, the elements of the bucket are gathered to get the sorted array.

The process of bucket sort can be understood as **a scatter-gather** approach. The elements are first scattered into buckets then the elements of buckets are sorted. Finally, the elements are gathered in order.


### How Bucket Sort Works?

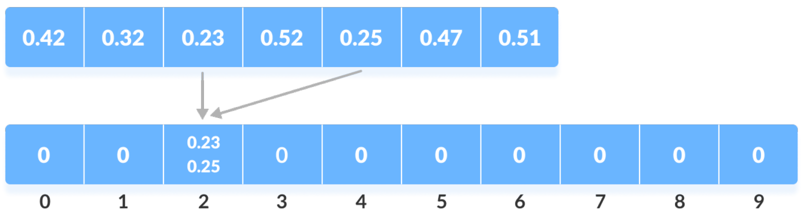
1. Suppose the input array is:




Create an array of size 10. Each slot of this array is used as a bucket for storing elements. (Array in which each position is a bucket)


**2.** Insert elements into the buckets from the array. The elements are inserted according to the range of the bucket.

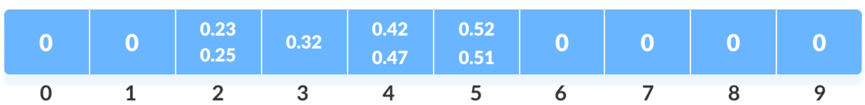
- In our example code, we have buckets each of ranges from 0 to 1, 1 to 2, 2 to 3,...... (n-1) to n.
- Suppose an input element is 0.23 is taken.
- It is multiplied by size = 10 (i.e. 0.23*10=2.3).
- Then, it is converted into an integer (i.e. 2.3 ≈ 2).
- Finally, 0.23 is inserted into **bucket-2**.




Similarly, 0.25 is also inserted into the same bucket. Every time, the floor value of the floating point number is taken.

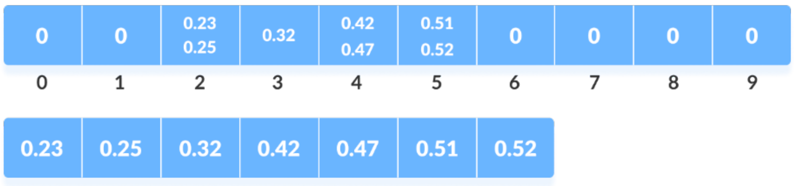
**If we take integer numbers as input, we have to divide it by the interval (10 here) to get the floor value.**

Similarly, other elements are inserted into their respective buckets.



3. The elements of each bucket are sorted using any of the stable sorting algorithms.
4. The elements from each bucket are gathered.

   It is done by iterating through the bucket and inserting an individual element into the original array in each cycle. The element from the bucket is erased once it is copied into the original array.




Bucket sort is used when:

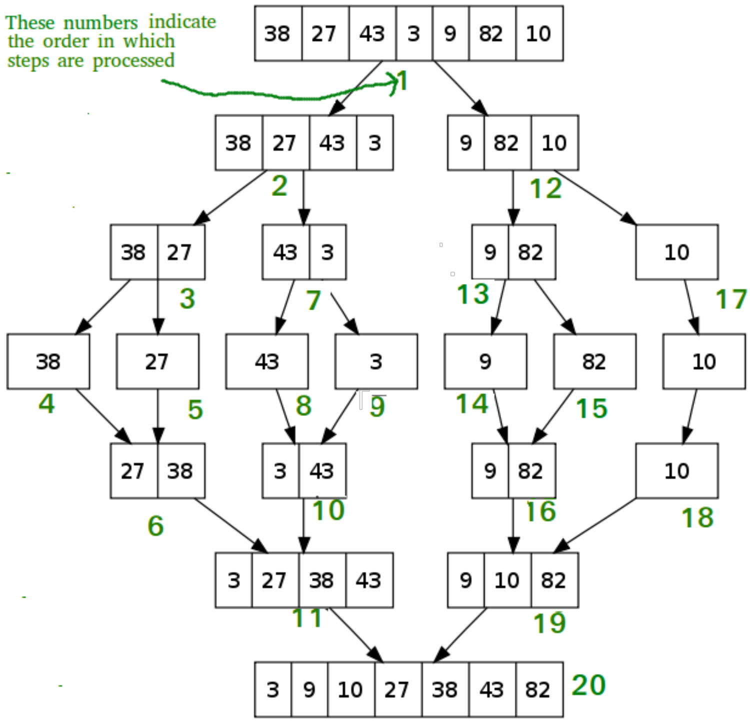
- input is uniformly distributed over a range.
- there are floating point values

**Merge Sort:**

- Merge Sort is a **Divide and Conquer** algorithm.
- It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

**Example:**

- The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.
- If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1.
- Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.




**Quick Sort:**

- Quick Sort is also based on the concept of **Divide and Conquer**, just like merge sort.
- But in quick sort all the heavy lifting (major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays.
- In case of quick sort, the combine step does absolutely nothing.

It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element (Central element)
3. Elements greater than the pivot element

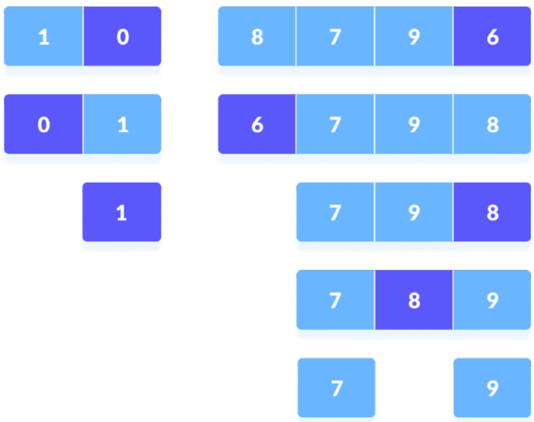
**Pivot** element can be any element from the array, it can be **the first element, the middle element, the last element or any random element.**

**How Quick Sort Works?**

1. A pivot element is chosen from the array. You can choose any element from the array as the pivot element. Here, we have taken the rightmost (i.e., the last element) of the array as the pivot element.


2. The elements smaller than the pivot element are put on the left and the elements greater than the pivot element are put on the right


3. Pivot elements are again chosen for the left and the right sub-parts separately. Within these sub-parts, the pivot elements are placed at their right position. Then, step 2 is repeated.




Select pivot element of in each half and put at correct place using recursion

4. The sub-parts are again divided into smaller sub-parts until each subpart is formed of a single element.

5. At this point, the array is already sorted.

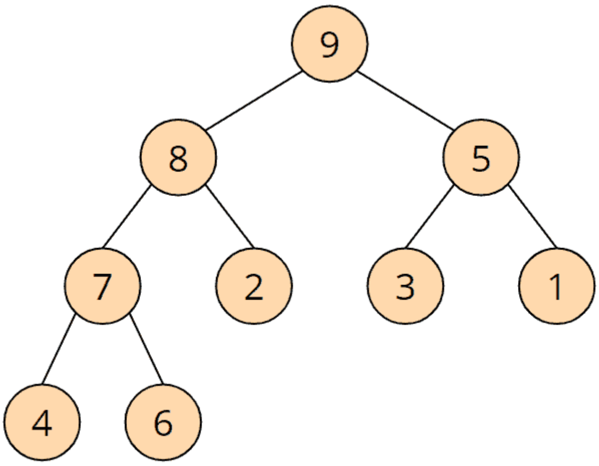
**Note:** The worst case is when the pivot element is the largest or smallest, or when all of the components have the same size. The performance of the quicksort is significantly impacted by these worst-case scenarios.

**Heap Sort**

The heap sort algorithm consists of two phases: In the first phase, the array to be sorted is converted into a max heap. In the second phase, the largest element (i.e. the one at the tree root) is taken and a max Heap is created again from the remaining elements.

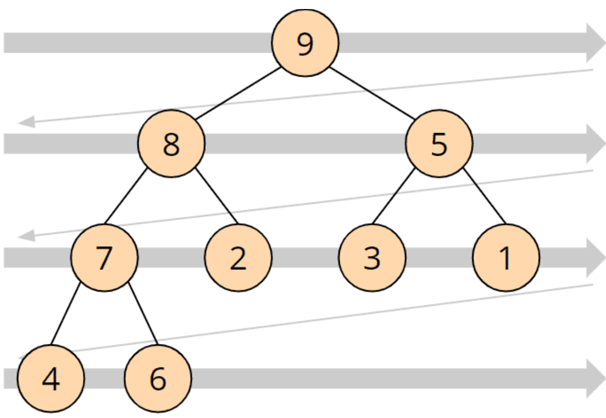
A "heap" denotes a binary tree in which each node is either greater than or equal to its children ("max heap") - or less than or equal to its children ("min heap").

Here is a simple example of a "max heap":




The 9 is greater than the 8 and the 5; the 8 is greater than the 7 and the 2; etc.

A heap is projected onto an array by transferring its elements from the top left to the bottom right of the array, line by line:



The heap shown above looks like this as an array:

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

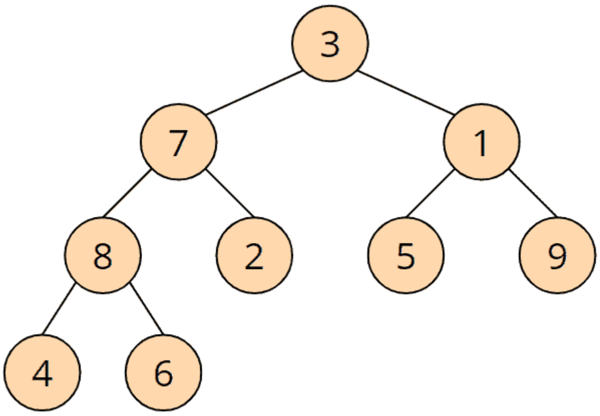
With a "max heap" the largest element is always at the top - in the array form it is therefore on the far left.

**Phase 1: Creating Heap**

The array to be sorted must first be converted into a heap. No new data structure is created for this, but the numbers are resorted within the array in such a way that the heap structure described above is created.

Example: Given the sequence of numbers [3, 7, 1, 8, 2, 5, 9, 4, 6]

We "project" this onto a binary tree as described above. The binary tree is not a separate data structure, but merely a thought construct - in memory the elements are exclusively in the array.



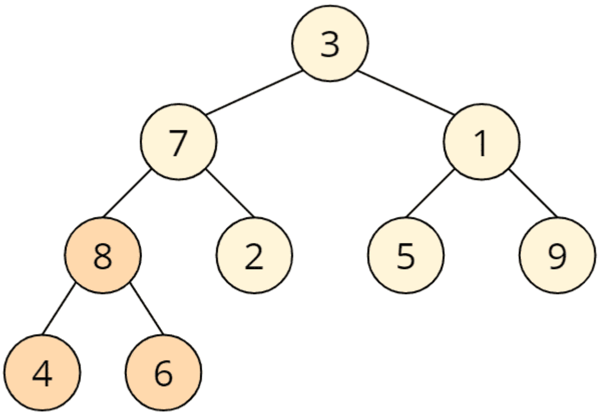

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 1 | 8 | 2 | 5 | 9 | 4 | 6 |

This tree does not yet correspond to a max heap. Its definition is that parents are always greater than or equal to their children.

To create a max heap, we now visit all of the parent nodes—backwards from last to first—and make sure that the heap constraint is satisfied for that node and those below it. We do this with the so-called heapify() function.

**Call #1 of the heapify function**

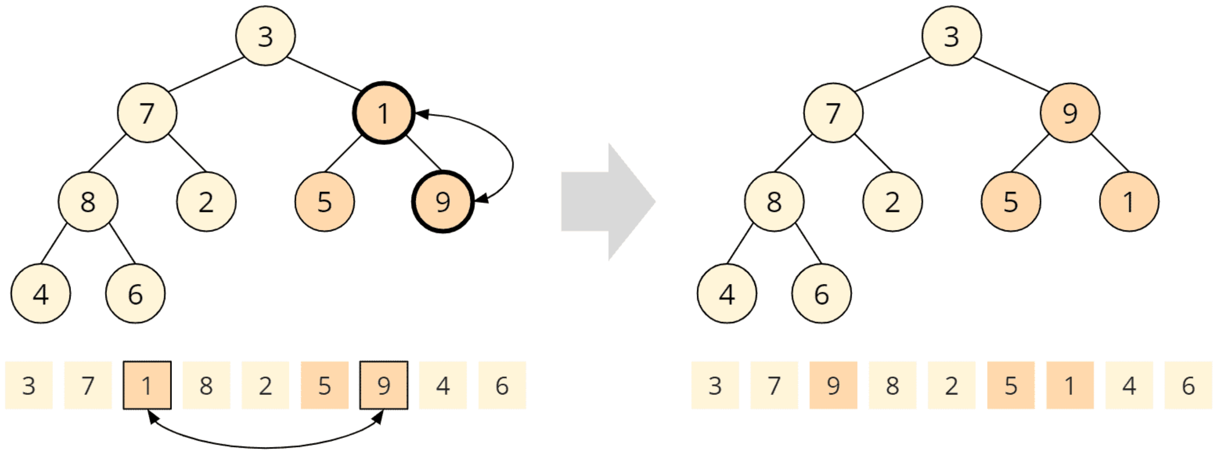
The heapify() function is called first for the last parent node. Parent nodes are 3, 7, 1 and 8. The last parent node is 8. The heapify() function checks if the children are smaller than the parent node. 4 and 6 are less than 8. So, on this parent node, the heap condition is satisfied and the heapify() function is finished.




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

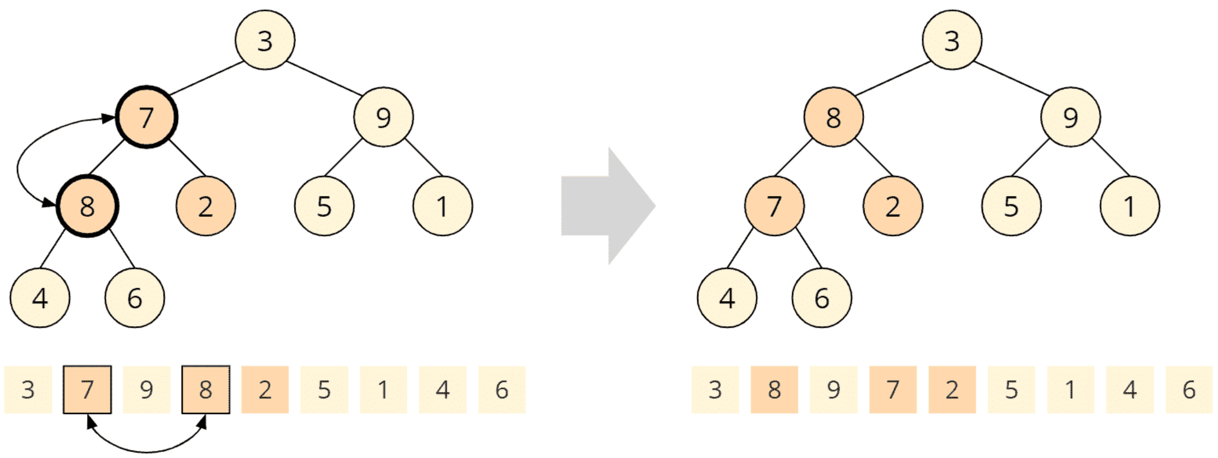
**Call #2 of the heapify function**

Second, heapify() is called on the penultimate node: 1. Children 5 and 9 are both greater than 1, so the heap constraint is violated. In order to restore the heap condition, we now exchange the larger child with the parent node, i.e. the 9 with the 1. The heapify() function is thus completed again.

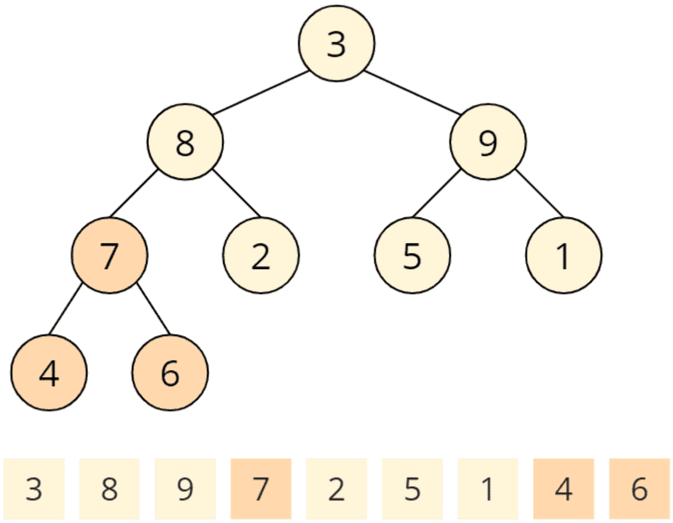



**Call #3 of the heapify function**

Now heapify() is called on node 7. Child nodes are 8 and 2, only the 8 is larger than the parent node. So we swap the 7 with the 8:

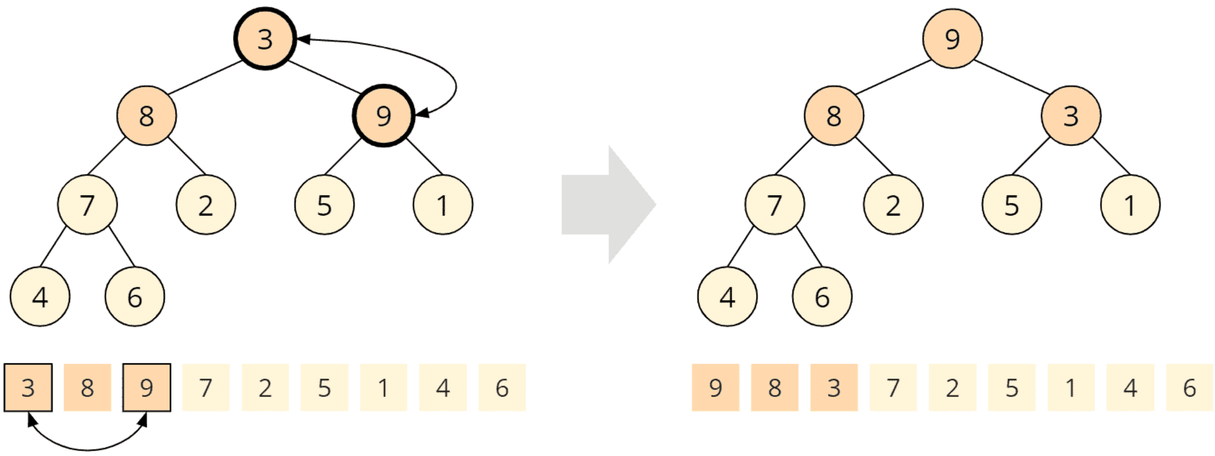



Since the child node we just swapped has two children itself, the heapify() function must now check whether the heap condition for this child node is still satisfied. In this case, the 7 is greater than 4 and 6, so the heap constraint is satisfied; this completes the heapify() function.

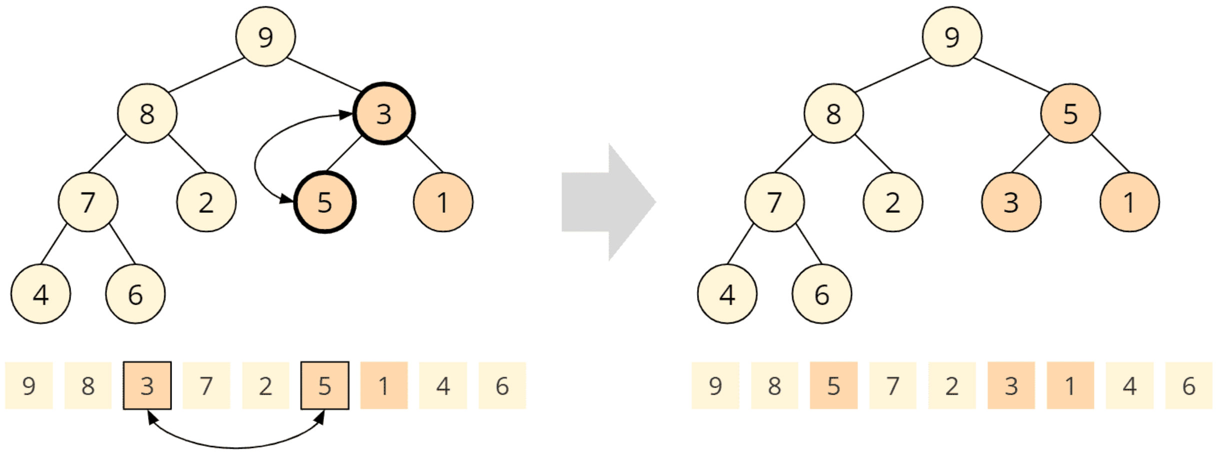



**Call #4 of the heapify function**

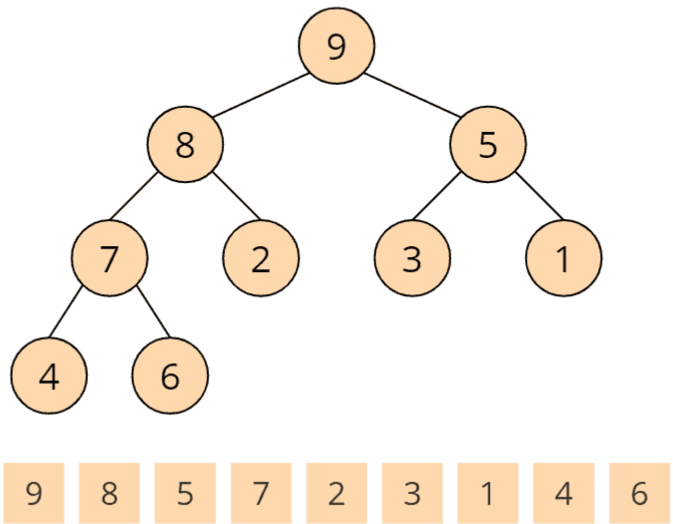
Now we have already arrived at the root node with element 3. Both child nodes, 8 and 9 are larger, with 9 being the largest child and therefore swapped with the parent node:




Again, the swapped child node has children of its own, so we need to check the heap constraint on that child node. The 5 is greater than the 3, i.e. the heap constraint is not met and must be restored by swapping the 5 and the 3:




This also completes the fourth and final call to the heapify() function. A max heap is created:



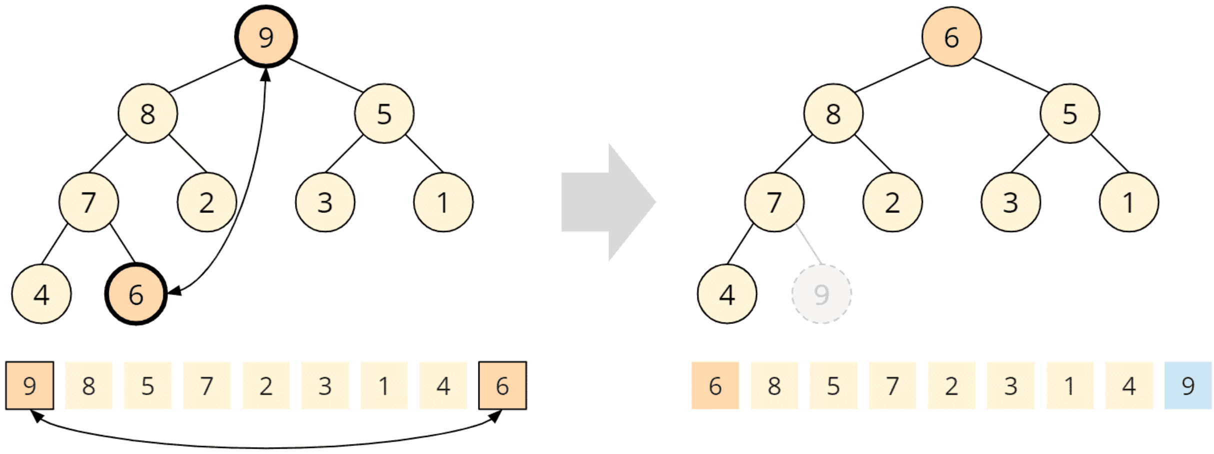

This brings us to phase two of the heapsort algorithm.

**Phase 2: Sorting the array**

In phase 2 we take advantage of the fact that the largest element of the max heap is always at its root (or at the far left in the array).

**Phase 2, Step 1: Swap root and last element**

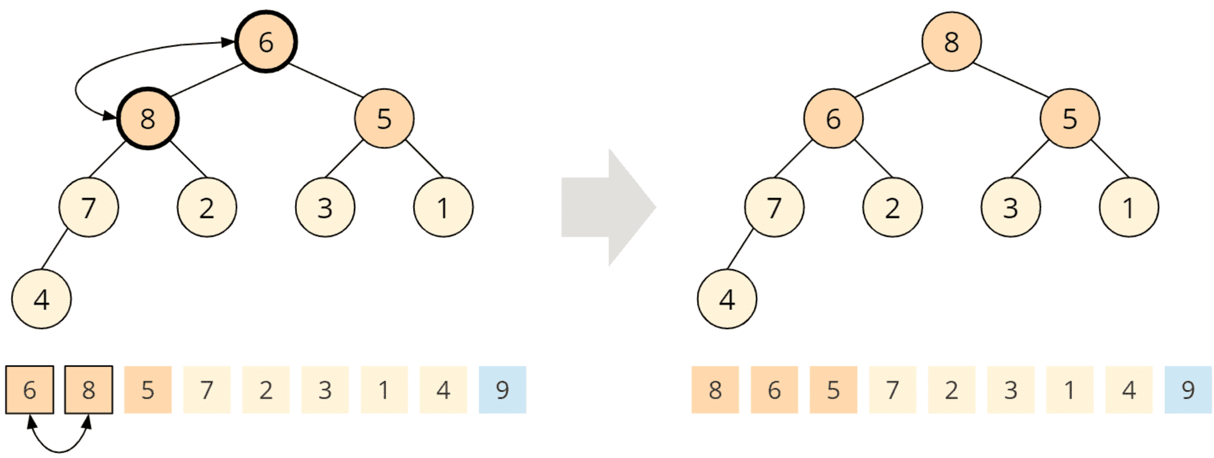
We now exchange the root element (the 9) with the last element (the 6), so that the 9 is in its final position at the end of the array (marked in blue in the array). We also mentally remove this element from the tree (shown in gray in the tree):



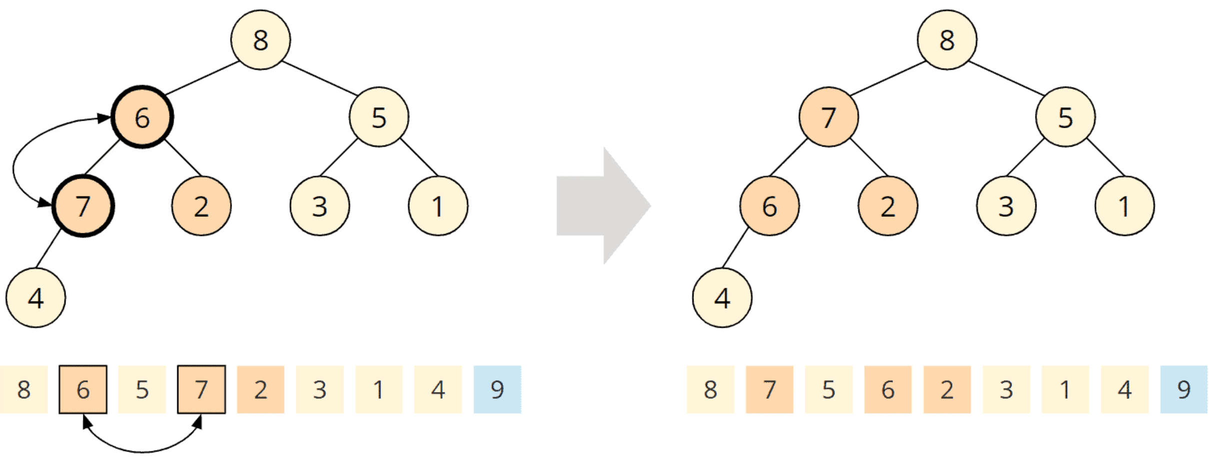
After we put the 6 at the root of the tree, it's no longer a max heap. In the next step we "fix" the heap.

***Phase 2, Step 2: Restore heap condition***

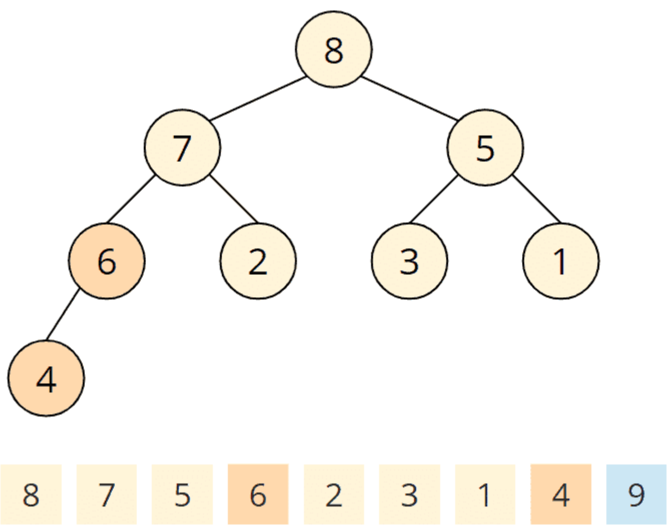
To restore the heap condition, we call the heapify() function known from phase 1 on the root node. So we compare the 6 with its children, 8 and 5. The 8 is bigger, so we swap it with the 6:



The swapped child node in turn has two children, the 7 and the 2. The 7 is larger than the 6, so we swap these two elements as well:

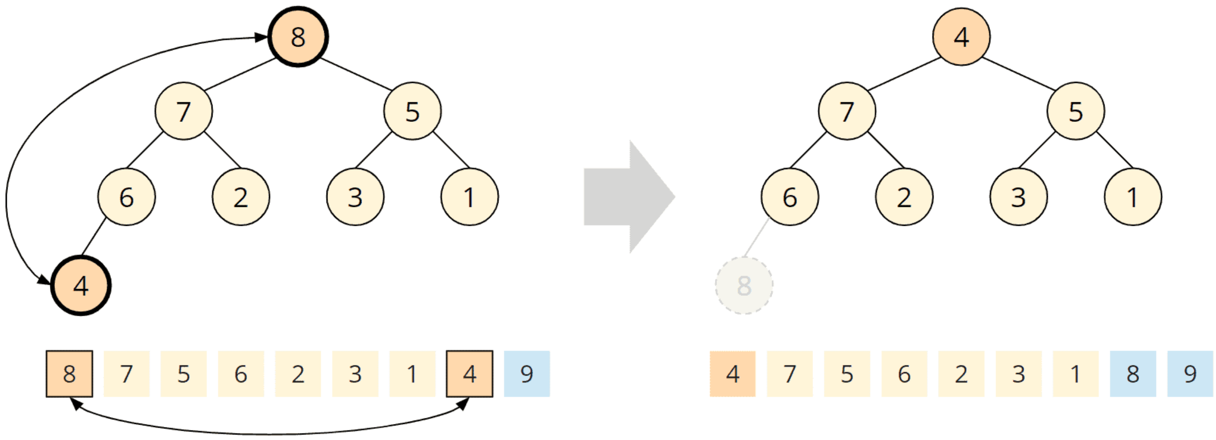



The exchanged child node also has another child, the 4. The 6 is larger than the 4, so the heap condition is fulfilled at this node, so the heapify() function is finished and we have a max heap again:



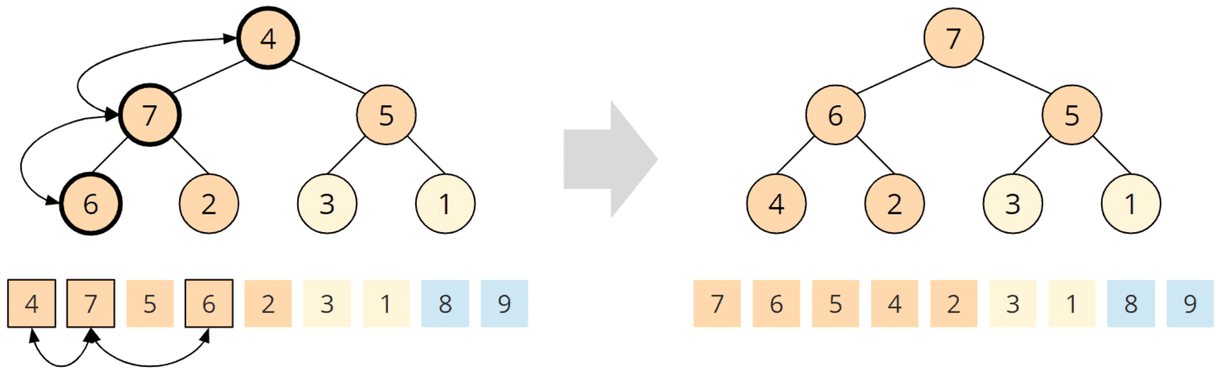

**Repeating the steps**

This puts the largest number in the remaining array, 8, in first place. This is swapped again with the last element of the tree. Since we truncated the tree by one element, the last element of the tree is on the penultimate field of the array:

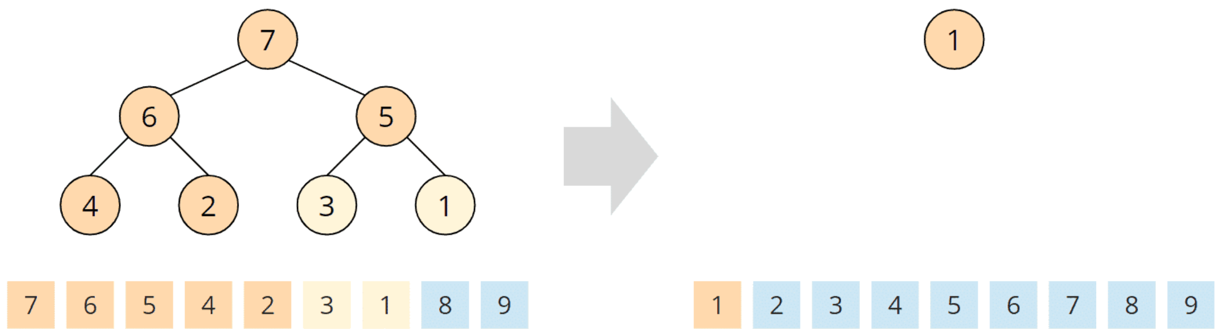



This sorts the last two fields of the array.

The heap constraint is now violated again at the root, and we repair the tree by calling heapify() on the root element (the following image shows all heapify steps at once).




We repeat the process until the tree contains only one element:

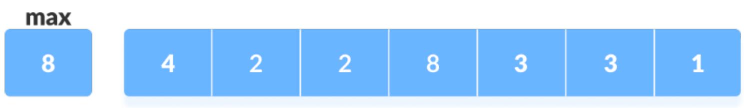


This is the smallest and stays at the beginning of the array. The algorithm is finished, the array is sorted:


**Counting Sort**

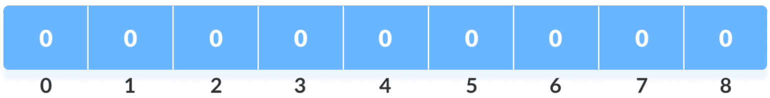
Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

1. Find out the maximum element (let it be max) from the given array.




Given array

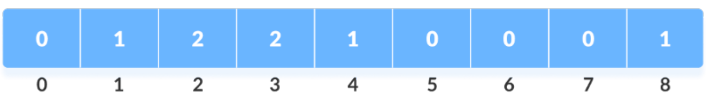
2. Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.




Count array

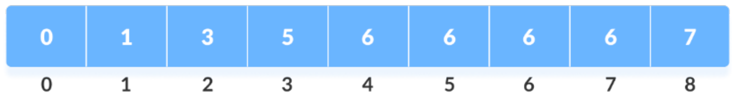
3. Store the count of each element at their respective index in count array

   For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of countarray. If element "5" is not present in the array, then 0 is stored in 5th position.



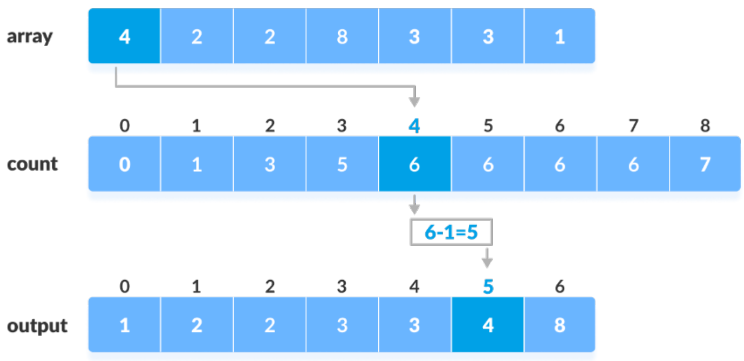

Count of each element stored

4. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.




Cumulative count

5. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.




Counting sort

6. After placing each element at its correct position, decrease its count by one.

**Note:** Counting sort is most efficient if the range of input values is not greater than the number of values to be sorted. In that scenario, the complexity of counting sort is much closer to O(n), making it a linear sorting algorithm.

## Radix Sort

The lower bound for the comparison based sorting algorithms (Merge Sort, Heap Sort, Quick-Sort etc) is $\Omega(n \log n)$, i.e., they cannot do better than $n \log n$. Counting sort is a linear time sorting algorithm that sort in $O(n+k)$ time when elements are in the range from 1 to k.

**What if the elements are in the range from 1 to $n^2$?**

In such a case, we can't use counting sort because counting sort will take $O(n^2)$ which is worse than comparison-based sorting algorithms. Can we sort such an array in linear time?

Radix Sort is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

The steps used in the sorting of radix sort are listed as follows -

- First, we have to find the largest element (suppose **max**) from the given array. Suppose **'d'** be the number of digits in **max**. The **'d'** is calculated because we need to go through the significant places of all elements.
- After that, go through one by one each significant place. Here, we have to use any stable sorting algorithm to sort the digits of each significant place.

Now let's see the working of radix sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using radix sort. It will make the explanation clearer and easier.

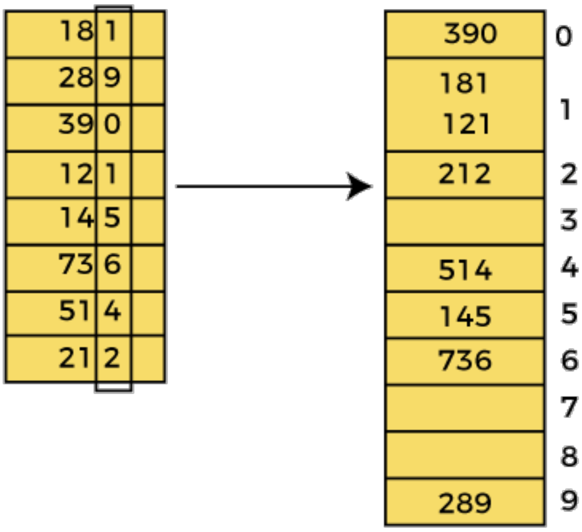
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 181 | 289 | 390 | 121 | 145 | 736 | 514 | 212 |

In the given array, the largest element is **736** that have **d = 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., **d = 0**). Here, we are using the counting sort algorithm to sort the elements.

**Pass 1:**

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

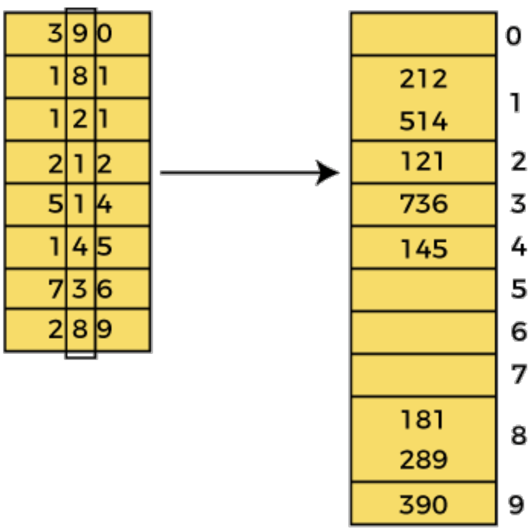



After the first pass, the array elements are -

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 390 | 181 | 121 | 212 | 514 | 145 | 736 | 289 |

**Pass 2:**

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

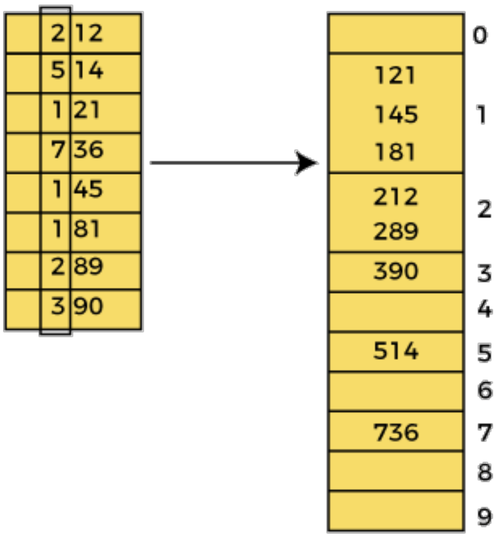



After the second pass, the array elements are -

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 212 | 514 | 121 | 736 | 145 | 181 | 289 | 390 |

**Pass 3:**

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




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 is a non-comparative sorting algorithm that is better than the comparative sorting algorithms. It has linear time complexity that is better than the comparative algorithms with complexity O(n log n).

| Sorting Algorithm | Best Case | Average Case | Worst Case | Worst Space | Advantage | Disadvantage |
|---|---|---|---|---|---|---|
| Bubble Sort | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ | $O(1)$ | 1) Simple<br>2) Stable<br>3) In-place<br>4) When the list is already sorted (best-case), the complexity of bubble sort is only O(n). | Not practical |
| Insertion Sort | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ | $O(1)$ | 1) Simple<br>2) Stable<br>3) In-place<br>4) When the list is already sorted (best-case), the complexity of insertion sort is only O(n). | Not practical |

| | | | | | | |
|---|---|---|---|---|---|---|
| Selection Sort | $\Omega(n^2)$ | $\Theta(n^2)$ | $O(n^2)$ | $O(1)$ | 1) Stable<br>2) In-place | Not practical |
| Bucket Sort | $\Omega(n+k)$<br>($n$: size of array; $k$: number of buckets) | $\Theta(n+k)$<br>($n$: size of array; $k$: number of buckets) | $O(n^2)$<br>($n$: size of array) | $O(n)$ | 1) Fast<br>2) Stable | Out-of-place |
| Merge Sort | $\Omega(n\,log(n))$ | $\Theta(n\,log(n))$ | $O(n\,log(n))$ | $O(n)$ | 1) Fast (also even in worst case)<br>2) Stable | Out-of-place |
| Quick Sort | $\Omega(n\,log(n))$ | $\Theta(n\,log(n))$ | $O(n^2)$ | $O(log(n))$ | 1) Fast in practice<br>2) In-place | 1) Not stable<br>2) Worst case with pivot element |
| Heap Sort | $\Omega(n\,log(n))$ | $\Theta(n\,log(n))$ | $O(n\,log(n))$ | $O(1)$ | 1) In-place<br>2) no additional memory requirement | 1) Not stable |
| Count Sort | $\Omega(n+k)$ | $\Theta(n+k)$ | $O(n+k)$ | $O(k)$ | 1) Stable<br>2) Linear complexity[1] | Out-of-place |
| Radix Sort | $\Omega(n*k)$ | $\Theta(n*k)$ | $O(n*k)$ | $O(n+k)$ | 1) Stable | Out-of-place |

[1] Only if the range k is not an order of n

| | | | | | | |
|---|---|---|---|---|---|---|
| | ($n$: size of array; $k$: number of digits) | ($n$: size of array; $k$: number of digits) | ($n$: size of array; $k$: number of digits) | ($n$: size of array; $k$: number of digits) | 2) Linear complexity | |

| There are three classes of sorting algorithms: | | |
|---|---|---|
| $\Theta(n^2)$<br>• Bubble Sort<br>• Insertion Sort<br>• Selection Sort | $O(n\,log(n))$<br>• Merge Sort<br>• Quick Sort<br>• Heap Sort | $O(n+k)$ or $O(n+k)$<br>• Count Sort (*depends on k)<br>Radix Sort |

| **Comparison based sorting** | **Non-comparison based sorting** |
|---|---|
| • Bubble Sort<br>• Insertion Sort<br>• Selection Sort<br>• Merge Sort<br>• Quick Sort<br>• Heap Sort | • Bucket Sort<br>• Count Sort<br>• Radix Sort |

**Searching Algorithms:**

**Linear Search:**

- Linear search is the simplest searching algorithm that searches for an element in a list in sequential order.
- We start at one end and check every element until the desired element is not found.

➢ Time Complexities
- Best case complexity: $O(1)$
- Average case complexity: $O(n/2)$
- Worst case complexity: $O(n)$

➢ Worst case space complexity: $O(1)$ (no extra space)

**Binary Search:**

- Binary Search is a searching algorithm for finding an element's position in a **sorted** array.
- In this approach, the element is always searched in the middle of a portion of an array.

➢ Time Complexities
- Best case complexity: $O(1)$
- Average case complexity: $O(log(n))$
- Worst case complexity: $O(log(n))$

➢ Worst case space complexity: $O(n)$