Chapter - 1

Algorithm Basic

An algorithm is an unambiguous specification of how to solve a class of problems. The performance of an algorithm is measured on the basis of following properties:

- Time Complexity: The total amount of time required by an algorithm to complete its execution.
- Space Complexity: Space complexity is an amount of memory used by the algorithm, to execute it completely.

Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n.

There are three different notations:

- big O - for the worst case running time.
- big Theta (Θ) - the running time is the same for all cases (average case)
- big Omega (Ω) - for the best case running time.

Following is a list of some common asymptotic notations -

Constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
quadratic	$O(n^2)$
Cubic	$O(n^3)$
polynominal	$nO(1)$
exponential	$2^{O(n)}$

How to write an Algorithm ?

→ There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else) etc. These common constructs can be used to write an algorithm.

Problem:

Design an algorithm to add two numbers and display the result. Writing step numbers, is optional.

→

step 1 : start ADD

step 2 : get values of a & b

step 3 : c ← a+b

a+b এর মান c তে রাখব

step 4 : display c

step 5 : stop

```
{ int a, b, c ;
scanf ("%d %d", &a , &b ) ;
c = a+b ;
print c ;
}
```

Algorithm Analysis:

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following-

- **A priori Analysis** - This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- **A Posterior Analysis**: This is an empirical analysis of an algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis actual statistics like running time and space required, are collected.

Chapter- 2

Sorting algorithm

A sorting algorithm is used to arrange elements of an array/list in a specific order. There are many type of sorting algorithms.

* Bubble sort
* selection sort
* Insertion sort
* Merge sort
* Quick sort
* Counting sort
* Radix Sort.

Quick sort:

→ Quick sort is based on the divide-and-conquer approach based on the idea of choosing one element as a pivot element and partitioning the array around it such that: Left side of pivot contains all the elements that are less than the pivot element Right side contains all elements greater than the pivot.

Example: Sort {1, 12, 5, 26, 7, 14, 3, 7, 2} using quick sort.

[1] [12] [5] [26] [7] [14] [3] [7] [2]

Pivot

[1] [12] [5] [26] [7] [14] [3] [7] [2] ; 12 ≥ 7 ≥ 2 ; swap 12 & 2

i j

[1] [2] [5] [26] [7] [14] [3] [7] [12] ; 26 ≥ 7 ≥ 7 swap 26 & 7

i j

q−1 q+1

[1] [2] [5] [7] [7] [14] [3] [26] [12] ; 7 ≥ 7 ≥ 3 ; swap 7 & 3

i j

q

[1] [2] [5] [7] [3] [14] [7] [26] [12] ; i > j , stop partition

j i

[1] [2] [5] [7] [3] [14] [7] [26] [12] ; run quick sort recursively

. . . . . .

[1] [2] [3] [5] [7] [7] [12] [14] [26] ; sorted

"Code Python"

P = 1st value

r = last value

(2) q = pivot

(1) if p < r

(2) q = partition (A, P, r) → pivot

(3) Quick sort (A, P, q − 1)

(4) Quick sort (A, q + 1, r)

p ——— q-1 q ——— q+1 ——— r

Ques

① Quick sort working process

② Algorithm (if partition needed)

(3) Pseudocode.

~~main code~~

	Best	average	worst
Time complexity	O(nlogn)	O(nlog n) O(nlogn)	O(n²)
space "	O(1)		

Quick sort:

p = 1st value

r = last val

[a.] Sort {24, 9, 29, 14, 19, 27} using quicksort where pivot = 24.

→ [24] [9] [29] [14] [19] [27]
p↑i ↑j

[24] [9] [29] [14] [19] [27]
p↑i ↑j

[19] [9] [29] [14] [24] [27]
↑i ↑p/j

[19] [9] [24] [14] [29] [27]
p/↑i ↑j

[19] [9] [14] [24] [29] [27]
↑i ↑j

. . . . . .

[9] [14] [19] [24] [27] [29] ; sorted

b Sort {4, 2, 6, 5, 3, 9} using quick sort where pivot = 5.

→ [4] [2] [6] [5] [3] [9]
i p j

[4] [2] [6] [5] [3] [9]
i p j

[4] [2] [6] [5] [3] [9]
i p j

[4] [2] [3] [5] [6] [9] ; 6 & 3 swapped
p

[4] [2] [3] [5] [6] [9]

. . . . . .

[2] [3] [4] [5] [6] [9] ; sorted.

{15, 5, 8, 3, 2}

→ [15] [5] [8] [3] [2]
p i j

[2] [5] [8] [3] [15]
i j p

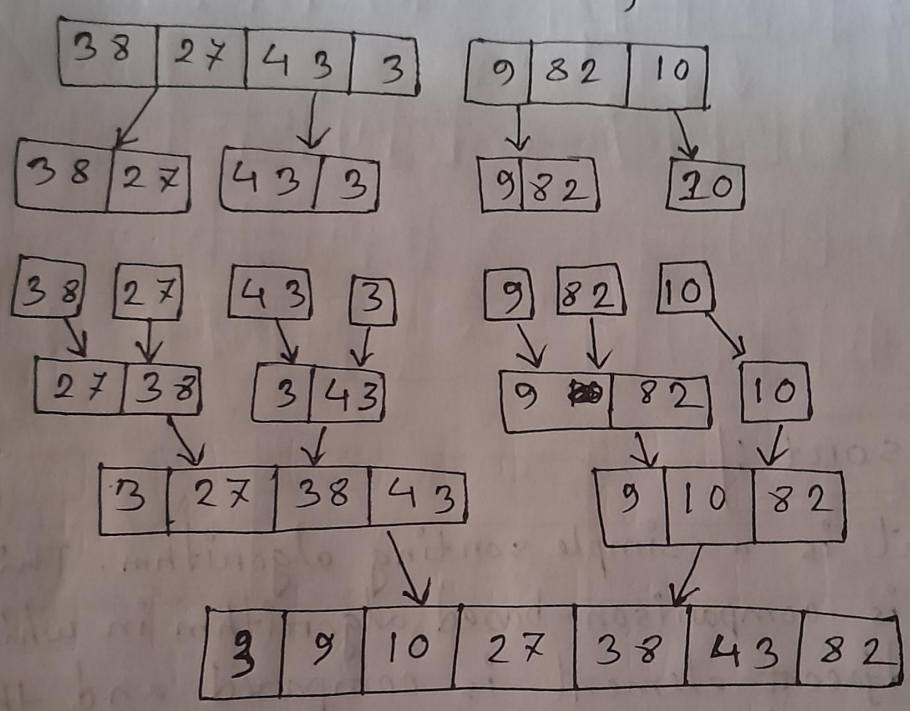
[2] [3] [8] [5] [15]
i j p

[2] [3] [5] [8] [15]

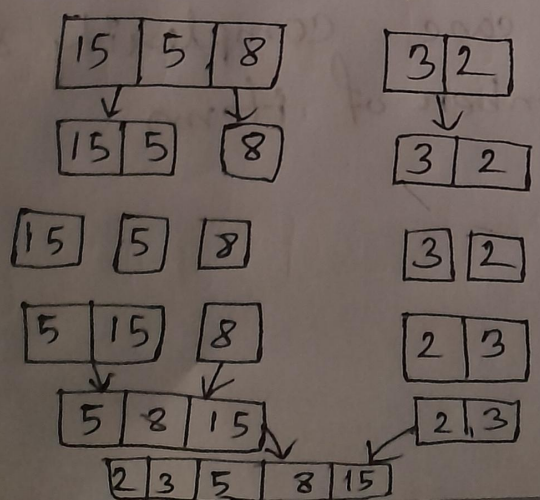
Merge sort:

→ Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublists consists of a single element and merging those sublists in a manner that results into a sorted list.

→ {38, 27, 43, 3, 9, 82, 10}



→ {15, 5, 8, 3, 2}



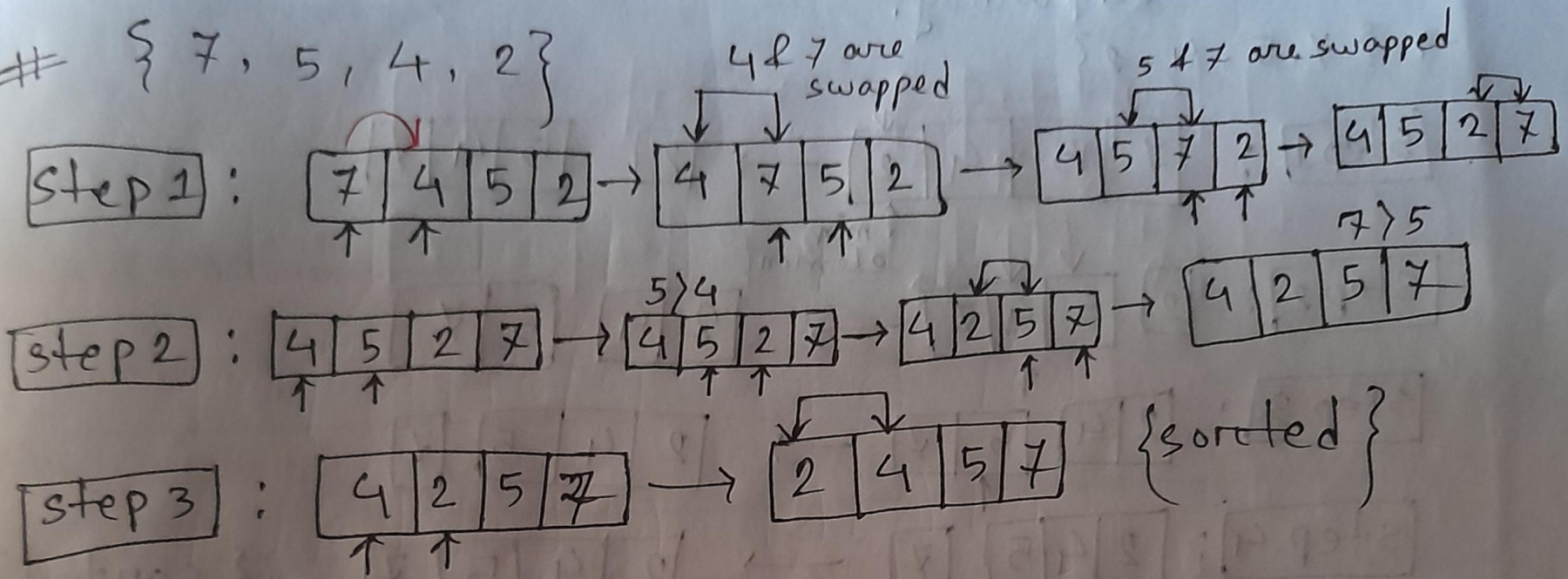
sinat

Complexity	Best case	Average case	Worst case
Time Complexity	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Space Complexity	$O(n)$		

Bubble sort :

→ Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent element is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

{ 7, 5, 4, 2 }

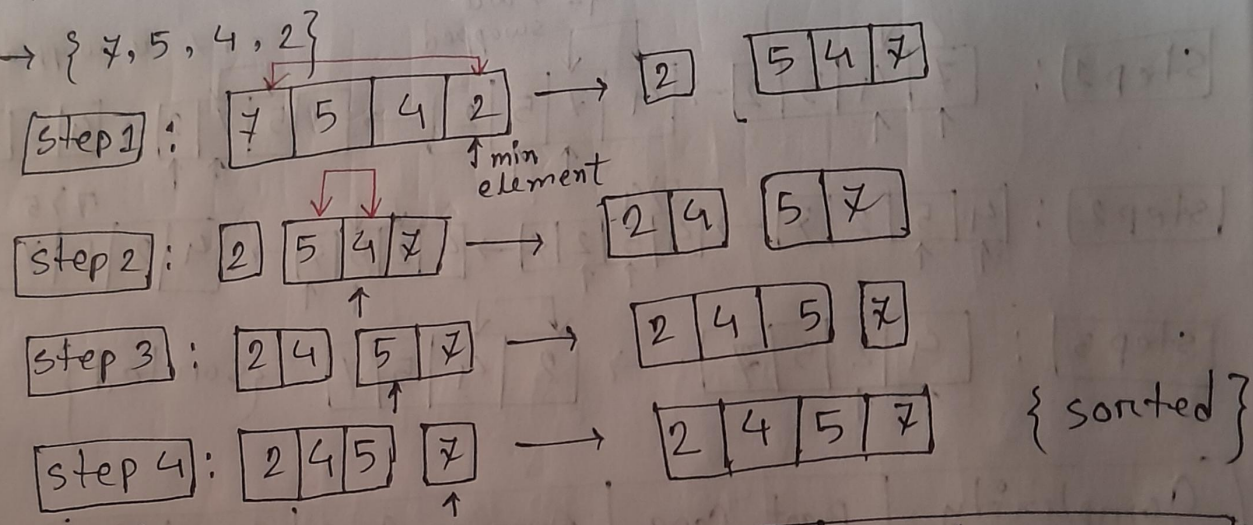


Complexity	Best Case	Average Case	Worst Case
Time Complexity	$O(n)$	$O(n^2)$	$O(n^2)$
Space Complexity	$O(1)$		

Selection Sort:

→ The selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting in its correct position in a sorted array.

→ {7, 5, 4, 2}



Complexity	Best Case	Average Case	Worst case
Time Complexity	$O(n^2)$	$O(n^2)$	$O(n^2)$
Space Complexity	$O(1)$		

Insertion sort:

This is an in-place comparison-based sorting algorithm. Insertion sort is based on the ~~inp~~ idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in sorted array.

~~The algorithm is~~

Complexity	Best Case	Average Case	Worst case
Time Complexity	$O(n)$	$O(n^2)$	$O(n^2)$
Space Complexity	$O(1)$		

{ 7, 4, 5, 2 }

Step 1:

7	4	5	2

→ No element on left side of 7, so no change in its position

Step 2:

7	4	5	2

→

4	7	5	2

Step 3:

4	7	5	2

→

4	5	7	2

Step 4:

4	5	7	2

→

2	4	5	7

Counting Sort

→ Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurences 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.

Sort {4, 2, 2, 8, 3, 3, 1} using counting sort.

→ Step 1: Find max element from the given array.

4	2	2	8	3	3	1

Here, max = 8

Step 2: Initialize count array of length max + 1 with all elements 0.

0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8

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

0	1	2	3	4	5	6	7	8
0	1	2	2	1	0	0	0	1

Step-4: Store cumulative sum of element at their of the array.

0	1	2	3	4	5	6	7	8
0	1	3	5	6	6	6	6	7

Step-5: Find the index of each element of the original array in the count array. This gives the cumulative count. After placing each element at its correct position, decrease it count by one.

array

4	2	2	8	3	3	1

count

0	1	2	3	4	5	6	7	8
0	1	3	5	6	6	6	6	7

6-1=5

output

0	1	2	3	4	5	6
1	2	2	3	3	4	8

Time Complexity:

worst case : $O(n+k)$

Best Case : $O(n+k)$

Average Case : $O(n+k)$

where n is the number of elements in the array and k is the range of the elements.

Sort {2,5,0,3,2,3} using counting sort.

Step 1: Find max element from the given array

2	5	0	3	2	3

Here, max = 5

Step 2: Initialize count array of length max+1 with all elements 0.

5+1 = 6

0	0	0	0	0	0	~~0~~
0	1	2	3	4	5	~~6~~

Step 3: Store the count of each elements at their respective index in count array.

1	0	2	2	0	1	~~0~~
0	1	2	3	4	5	~~6~~

Step 4: Store cumulative sum of the elements of the count array.

1	1	3	5	5	6	~~6~~
0	1	2	3	4	5	~~6~~

Step 5: Find the index of each element of the original array in the count array. This gives the cumulative count. After placing each element at its correct position, decrease its count by one.

array

2	5	0	3	2	3

count

1	1	3	5	5	6	~~6~~
0	1	2	3	4	5	~~6~~

output

0	2	2	3	3	5
0	1	2	3	4	5

Radix Sort Algorithm

In this article, we will discuss the Radix sort Algorithm. Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

The process of radix sort works similar to the sorting of students names, according to the alphabetical order. In this case, there are 26 radix formed due to the 26 alphabets in English. In the first pass, the names of students are grouped according to the ascending order of the first letter of their names. After that, in the second pass, their names are grouped according to the ascending order of the second letter of their name. And the process continues until we find the sorted list.

Now, let's see the algorithm of Radix sort.

Algorithm

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

Working of Radix sort Algorithm

Now, let's see the working of Radix sort Algorithm.

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 '**x**' be the number of digits in **max**. The '**x**' 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.

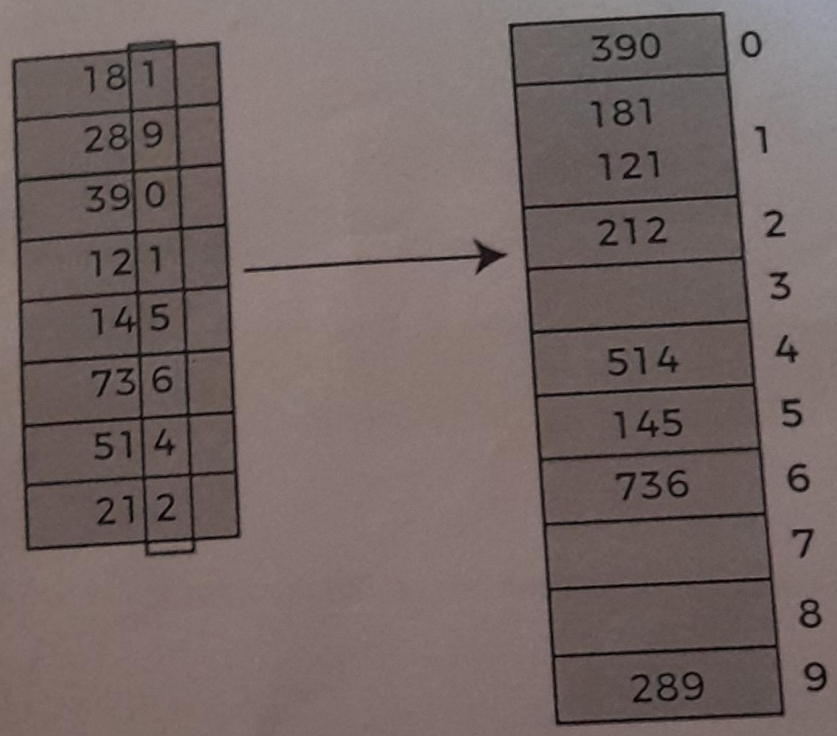
181	289	390	121	145	736	514	212

In the given array, the largest element is **736** that have **3** digits in it. So, the loop will run up t
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 elements.

Pass 1:

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

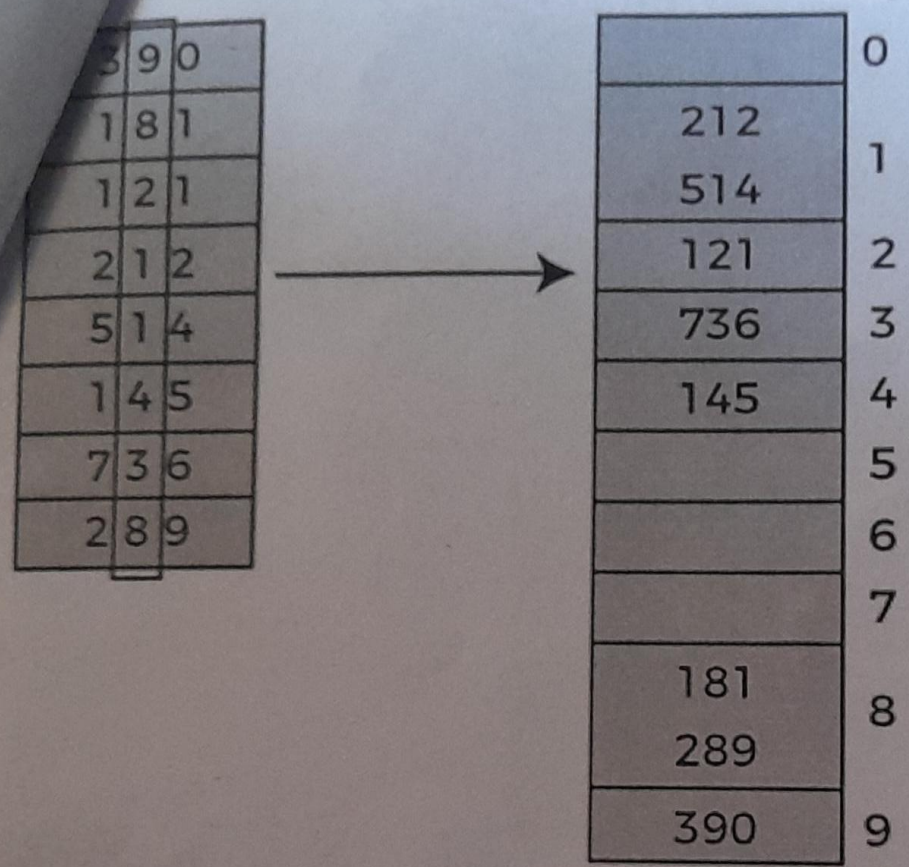


After the first pass, the array elements are -

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

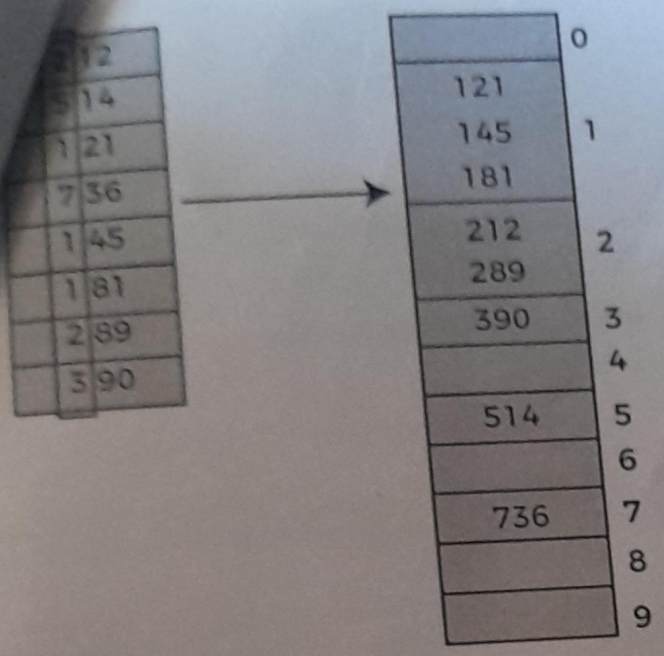


After the second pass, the array elements are –

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 100th pla



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 complexity

Now, let's see the time complexity of Radix sort in best case, average case, and worst cas of Radix sort.

1. Time Complexity

Case	Time Complexity
Best Case	Ω(n+k)
Average Case	Θ(nk)
Worst Case	O(nk)

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. th complexity of Radix sort is **Ω(n+k)**.