

# Array

An array is a finite , ordered set of homogenous elements.

Finite: There are specific number of elements.

Ordered: The elements are arranged so that there is a first element , second & son on.

Homogenous: All the elements are of the same type.

Primitive Operations on array

Create(array, type, size)

Store(array, index, value)

Retrieve(array, index)

# Array Representation

`int a[10]; //create 1D array`

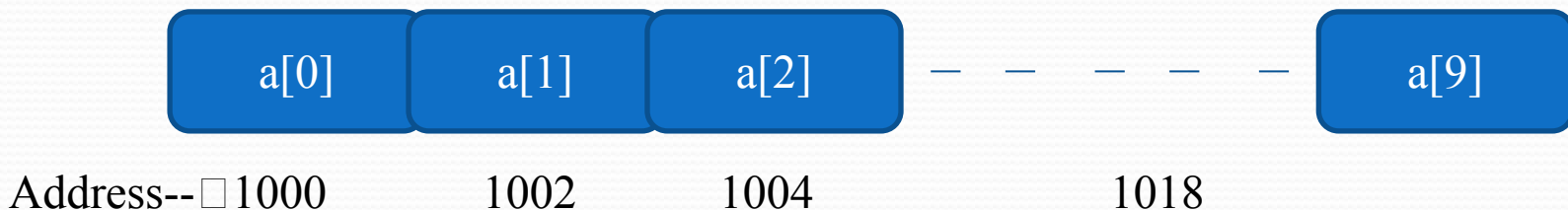
`int a[4][5]; //create 2D array having 4 rows & 5 columns`

The elements are stored in consecutive memory locations.

The address of first element is called as **base address**.

The lower bound of the array starts at 0 & upper bound is size-1.

e. g. `a[2]=15`



# Address Calculation of 1D Array

- To access any element  $a[i]$ , the pointer notation is used as  $a[i]=*(a+i)$
- $a$ - is a base address
- $i$ - is index value
- The address of an element is calculated as  
Address of  $a[i]=\text{base address} + i * \text{element size}$   
e.g. Address of  $a[2]=1000+2*\text{sizeof(int)}$   
 $=1000+2*2$   
 $=1004$

# Representation of Multidimensional Arrays

- `int a[3][5];`

a	0	1	2	3	4
0					
1					
2					

```
int m[2][3][4];
```

A diagram of a 2D array  $m$  with 4 columns and 3 rows. The first row is dark blue, the second row is medium blue, and the third row is light blue. The label  $m[0]$  is at the bottom center, and  $m[1]$  is at the bottom right.



**Logical View:** shows elements according to their relationship.

**Physical View:** elements of multidimensional array are stored linearly in memory.

Array can be physically represented in 2 ways:

**Row-Major Representation:** Elements are arranged row-wise. Elements of row 0 stored first, Elements of row 1 stored next & so on.

**Column-Major Representation:** Elements are arranged column-wise. Elements of column 0 stored first, Elements of column 1 stored next & so on.

Columns->    0    1    2

Rows



0

1

2

1	2	3
4	5	6
7	8	9

Row Major Representation

□----Row0-----□			□----Row1----□			□----Row2----□		
1	2	3	4	5	6	7	8	9

Column Major Representation

□---Column0----□			□--Column1---□			□---Column2----□		
1	4	7	2	5	8	3	6	9

For an array `int a[r][c]`; where `r`=no. of rows, `c`=no of columns, location of an element `a[i][j]` in array can be calculated as:

### **Row Major Representation**

$$\begin{aligned}\text{Address } a[i][j] &= \text{Base-address} + i * c * \text{elementsize} + j * \text{elementsize} \\ &= \text{Base-address} + (i * c + j) * \text{elementsize}\end{aligned}$$

Example

$$\begin{aligned}\text{Address } a[1][2] &= 1000 + (1 * 3 + 2) * \text{sizeof(int)} \\ &= 1000 + 10 \\ &= 1010\end{aligned}$$

### **Column Major Representation**

$$\begin{aligned}\text{Address } a[i][j] &= \text{Base-address} + j * r * \text{elementsize} + i * \text{elementsize} \\ &= \text{Base-address} + (j * r + i) * \text{elementsize}\end{aligned}$$

Example

$$\begin{aligned}\text{Address } a[1][2] &= 1000 + (2 * 4 + 1) * \text{sizeof(int)} \\ &= 1000 + 18 \\ &= 1018\end{aligned}$$

# Generalize Formula

- For n dimensional array stored in row major order int **a[s1][s2].....[sn];**
- To access element **a[i1][i2]....[in]** we have to pass through i1 array of size  $s2*s3*...*sn$  then through i2 arrays of size  $s3*s4*...*sn$  & so on.
- Finally we have to pass in element.
- Addr of **a[i1][i2]...[in]**  
$$= \text{baseaddr} + i1 * s2 * ... * sn * \text{esize} + i2 * s3 * ... * sn * \text{esize} + in * \text{esize}$$
$$= \text{baseaddr} + [i1 * s2 * ... * sn + i2 * s3 * ... * sn + ... + in] * \text{esize}$$



- Given an integer array `int a[5][4][3]` whose base address is 1000, calculate address of element `a[2][3][1]`

Sol- $\rightarrow$  Given  $s_1=5$   $s_2=4$   $s_3=3$   $i_1=2$   $i_2=3$   $i_3=1$

Address of `a[2][3][1]`

$$= 1000 + (i_1 * s_2 * s_3 + i_2 * s_3 + i_3) * \text{sizeof}(\text{int})$$

$$= 1000 + (2 * 4 * 3 + 3 * 3 + 1) * 2$$

$$= 1000 + (24 + 10) * 2$$

$$= 1000 + 34 * 2$$

$$= 1068$$

# Searching

- Searching is a process of finding a particular element from set of several given elements.
- The search is successful if the required element is found otherwise the search is unsuccessful.

There are different types of Searching as follows:

1. Linear Search
2. Binary Search

# Linear Search

- Linear Search is the simplest searching algorithm.
- It traverses the array sequentially to locate the required element.
- It searches for an element by comparing it with each element of the array one by one. so it is also called as **Sequential Search**.
- It can be applied to files , arrays or linked list.

Linear Search Algorithm is applied when-

- No information is given about the array.
- The given array is unsorted or the elements are unordered.
- The list of data items is smaller.

# Linear Search Algorithm

Consider-

- There is a linear array 'a' of size 'n'. Algorithm searches for the required key & returns the position i of the record where key is found
- 1. Start
- 2.  $i=0$
- 3. Read value of key to be searched
- 4. If  $k(i) == \text{key}$   
Display "Record found at position i"  
goto step 8
- 5. Increment i
- 6. If  $i < n$   
goto step 4
- 7. Display "No match found"
- 8. Stop

# Time Complexity Analysis

Linear Search time complexity analysis is done below-

## Best case-

- In the best possible case the element being searched may be found at the first position. Only one comparison is performed
- Thus in best case linear search algorithm takes  **$O(1)$**  operations.

## Worst Case-

- In the worst possible case the element being searched may be present at the last position or not present in the array at all.
- If search terminates with success or failure we have to perform  $n$  comparisons.
- Thus in worst case, linear search algorithm takes  **$O(n)$**  operations.

## Linear Search Efficiency-

- Linear Search is less efficient when compared with other algorithms like Binary Search.

# Variations of linear search

- Sentinel Search
- Probability Search
- Ordered List Search

# Sentinel Linear Search

- It is a type of linear search where the number of comparisons are reduced as compared to a traditional linear search.
- When a linear search is performed on an array of size  $N$  then in the worst case a total of  $N$  comparisons are made when the element to be searched is compared to all the elements of the array and  $(N + 1)$  comparisons are made for the index of the element to be compared so that the index is not out of bounds of the array which can be reduced in a Sentinel Linear Search.
- In this search, the last element of the array is replaced with the element to be searched and then the linear search is performed on the array without checking whether the current index is inside the index range of the array or not because the element to be searched will definitely be found inside the array even if it was not present in the original array since the last element got replaced with it. So, the index to be checked will never be out of bounds of the array.
- The number of comparisons in the worst case here will be  $(N + 2)$ .
- **Input:**  $arr[] = \{10, 20, 180, 30, 60, 50, 110, 100, 70\}$ ,  $x = 180$   
**Output:** *180 is present at index 2*
- **Input:**  $arr[] = \{10, 20, 180, 30, 60, 50, 110, 100, 70, 90\}$ ,  $x = 90$   
**Output:** *Not found*

# Probability Linear Search

- It arrange the elements by their frequency of access.
- It will reduce the time taken to access a value that is needed frequently.
- The values in the array are arranged with the most probable search elements at the beginning of the array & the least probable at the end.
- It is useful when some elements are searched more frequently than others.
- Method: After each successful search , we exchange the searched element with the element immediately before it in the array.
- Searched elements are moved up the list so that they occupy positions at the beginning of the array.



# Ordered List Search

- This method works on sorted data to improve search efficiency.
- If the elements are in sorted order then search continues till  $\text{key} < A[i]$  otherwise search terminates.
- If the key is not in the list, all elements need not be compared.
- Method: We sort the elements in the set. Store key at the end as the sentinel search. If  $\text{key} < \text{last element}$ , start search from position 0 & continue searching as long as  $\text{last element} < \text{key}$  & make sure that list has not ended.

# Binary Search


- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.
- It can be applied to files , arrays or linked list.

Binary Search Algorithm can be applied only on **Sorted arrays**.

- In this method the set of elements is partitioned into two equal parts & required key is compared with the key of the middle record. If match is found , the search terminates.
- If the key is less than the middle key , the search proceeds in the first (top) half of an array.
- If the key is greater than the middle key , the search proceeds in the second(lower) half of an array.
- The process continues till no more partitions are possible.
- Thus, every time a match is not found , the remaining array size to be searched reduces to half.

# Binary Search Algorithm

1. Start
2. Accept key to be searched
3.  $\text{begin}=0$  &  $\text{end}=\text{n}-1$
4. Repeat steps 5-7 while  $\text{begin}\leq\text{end}$
5.  $\text{mid}=(\text{begin}+\text{end})/2$
6. If  $\text{a}[\text{mid}]==\text{key}$   
Display "Record found at position mid"  
goto 9
7. If  $\text{key}<\text{a}[\text{mid}]$   
 $\text{end}=\text{mid}-1$   
else  
 $\text{begin}=\text{mid}+1$
8. Display "Key not found"
9. Stop



Binary Search Algorithm searches an element by comparing it with the middle most element of the array. Then, following three cases are possible-

**Case-01**

If the element being searched is found to be the middle most element, its index is returned.

**Case-02**

If the element being searched is found to be greater than the middle most element, then its search is further continued in the right sub array of the middle most element.

**Case-03**

If the element being searched is found to be smaller than the middle most element, then its search is further continued in the left sub array of the middle most element.

This iteration keeps on repeating on the sub arrays until the desired element is found or size of the sub array reduces to zero.

# Time Complexity Analysis-

- Binary Search time complexity analysis is done below-
- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for  $n$  elements in the array, there are  $\log_2 n$  iterations or recursive calls.
- **Time Complexity of Binary Search Algorithm is  $O(\log_2 n)$ .**
- Here,  $n$  is the number of elements in the sorted linear array.
- This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

# Examples on Binary Search

Q: Show the steps of searching using binary search method

A=5,9,15,22,34,47,68,75

Key1=9, Key2=34, Key3=25.

Sol-> A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]



Key1=9

begin	end	mid	A[mid]	Key Found?	Action
0	7	3	22	No	Search into left partition
0	2	1	9	Yes	Search Terminates

Key2=34

begin	end	mid	A[mid]	Key Found?	Action
0	7	3	22	No	Search into right partition
4	7	5	47	No	Search into left partition
4	4	4	34	Yes	Search Terminates

Key3=25

begin	end	mid	A[mid]	Key Found?	Action
0	7	3	22	No	Search into right partition
4	7	5	47	No	Search into left partition
4	4	4	34	No	Search into left partition
4	3	-	-	No	Search Terminates

# Sorting

Process of arranging or ordering elements in an ascending or descending order of the key values.

## **Definitions & Terminology**

- **Internal Sorting**

It is done on data which is stored in main memory.

- **External Sorting**

It is done on data which is stored in auxiliary storage devices.

- **Stable Sorting Technique**

A sorting method is said to be stable if the order of keys having same values in the unsorted set is retained in the sorted set.

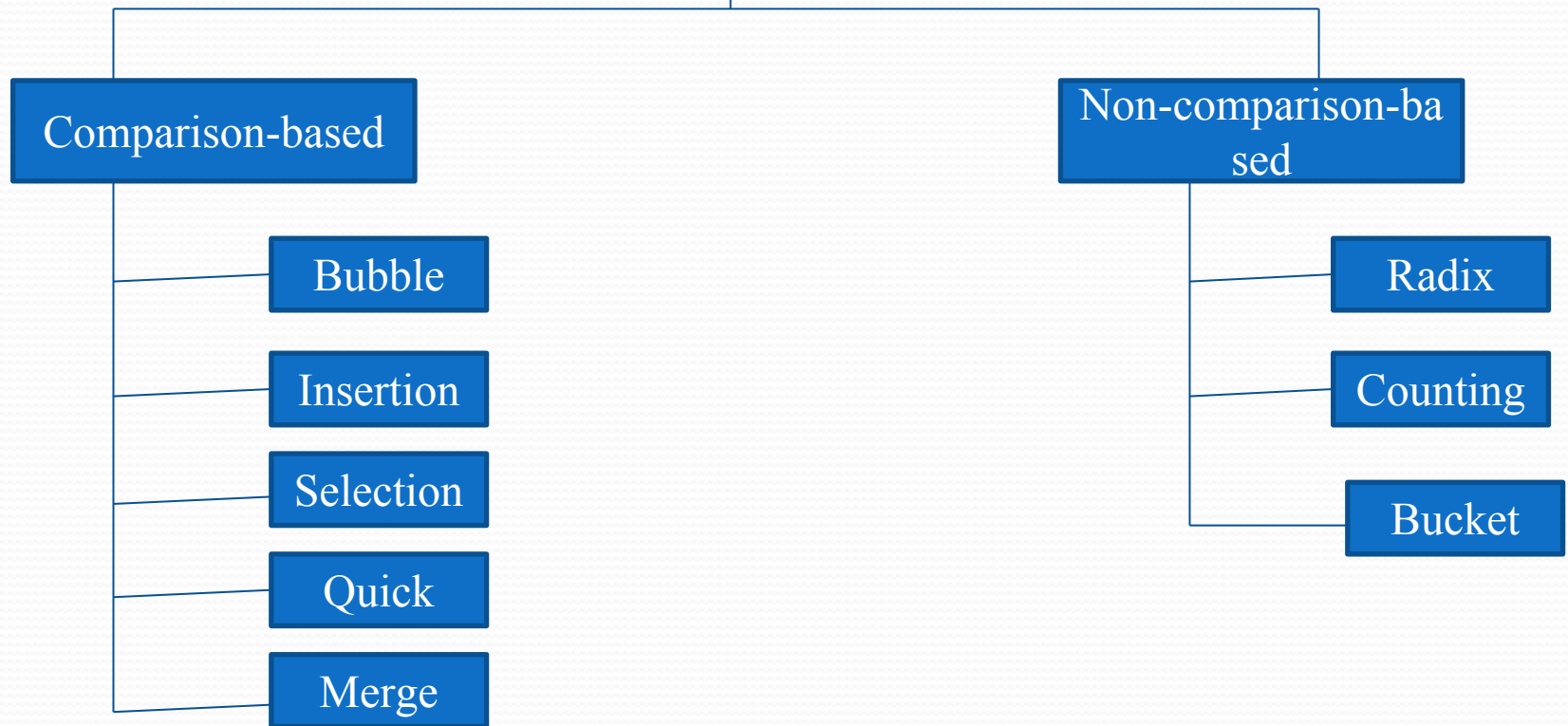
- **In-place Sorting**

A sorting algorithm is said to be in-place if it does not use a lot of additional memory for sorting.

e.g. Bubble sort, Insertion sort



# Types of Sorting Methods



# Bubble Sort

- The **bubble sort** makes multiple passes through a list.
- It compares adjacent items and exchanges those that are out of order.
- Each pass through the list places the next largest value in its proper place.
- Each item “bubbles” up to the location where it belongs.

Q: 25, 37, 12, 48, 57, 33

→ Pass - I

25	25	25	25	25	25
37	37	12	12	12	12
12	12	37	37	37	37
48	48	48	48	48	48
57	57	57	57	57	33
33	33	33	33	33	57

Pass - II

25	12	12	12	12
12	25	25	25	25
37	37	37	37	37
48	48	48	48	33
33	33	33	33	48
57	57	57	57	57

Pass - III

12	12	12	12
25	25	25	25
37	37	37	33
33	33	33	37
48	48	48	48
57	57	57	57

Pass IV

12	12	12
25	25	25
33	33	33
37	37	37
48	48	48
57	57	57



# Pass-V

12	12
25	25
33	33
37	37
48	48
57	57



Q: sort the foll<sup>y</sup> data using bubble sort 14, 33, 27, 35, 10

→ Pass - I

14	14	14	14	14
33	33	27	27	27
27	27	33	33	33
35	35	35	35	10
10	10	10	10	<u>35</u>

Pass II

14	14	14	14
27	27	27	27
33	33	33	10
10	10	10	<u>33</u>
<u>35</u>	<u>35</u>	<u>35</u>	<u>35</u>

Pass III

14	14	14
27	27	10
10	10	27
<u>33</u>	<u>33</u>	<u>33</u>
<u>35</u>	<u>35</u>	<u>35</u>

Pass IV

14	10
10	14
27	27
33	33
35	35

May, Sep, Oct, Jan, Apr, Nov  
 → Pass - I

Ma	Ma	Ma	Ma	Ma	Ma
Se	Se	Oc	Oc	Oc	Oc
Oc	Loc	Se	Ja	Ja	Ja
Ja	Ja	Ja	Se	Ap	Ap
Ap	Ap	Ap	Ap	Se	No
No	No	No	No	No	<span style="border: 1px solid black;">Se</span>

Pass - II

Ma	Ma	Ma	Ma	Ma
Oc	Oc	Ja	Ja	Ja
Ja	Ja	Oc	Ap	Ap
Ap	Ap	Ap	Oc	No
No	No	No	No	<span style="border: 1px solid black;">Oc</span>
<span style="border: 1px solid black;">Se</span>	<span style="border: 1px solid black;">Se</span>	<span style="border: 1px solid black;">Se</span>	<span style="border: 1px solid black;">Se</span>	<span style="border: 1px solid black;">Se</span>

Pass - III

Diagram illustrating the structure of a 4x4 grid with handwritten labels and a table structure:

Ma	Ja	Ja	Ja
Ja	Ma	Ap	Ap
Ap	Ap	Ma	Ma
No	No	No	No

Below the grid, four tables are shown, each with two rows and two columns:

OC	OC
Se	Se

OC	OC
Se	Se

OC	OC
Se	Se

OC	OC
Se	Se

### Pass -IV

Diagram illustrating the structure of a 3D array (3D array structure) with dimensions 3x3x3. The array is represented as a stack of three 2D planes (No, Oc, Se).

Plane 1 (Left):

- Top row: Jq, Ap, Ap
- Middle row: Ap, Jq, Jq
- Bottom row: Ma, Ma, Ma

Plane 2 (Middle):

- Top row: No, No, No
- Middle row: Oc, Oc, Oc
- Bottom row: Se, Se, Se

Plane 3 (Right):

- Top row: No, No, No
- Middle row: Oc, Oc, Oc
- Bottom row: Se, Se, Se

Pass - IV

AP	AP
Jq	Jq
Mq	Mq
No	No
OC	OC
Se	Se



# Efficiency of Bubble Sort

- There are  $n-1$  comparisons in the first pass,  $n-2$  in second pass & 1 comparison in the  $n-1^{\text{th}}$  pass i. e. last pass
- Total no of comparisons  $= (n-1) + (n-2) + \dots + 1 = n(n-1)/2$
- Same no of passes are required in best case & worst case.
- Time complexity in best case & worst case is  $= O(n^2)$

## **Advantages:**

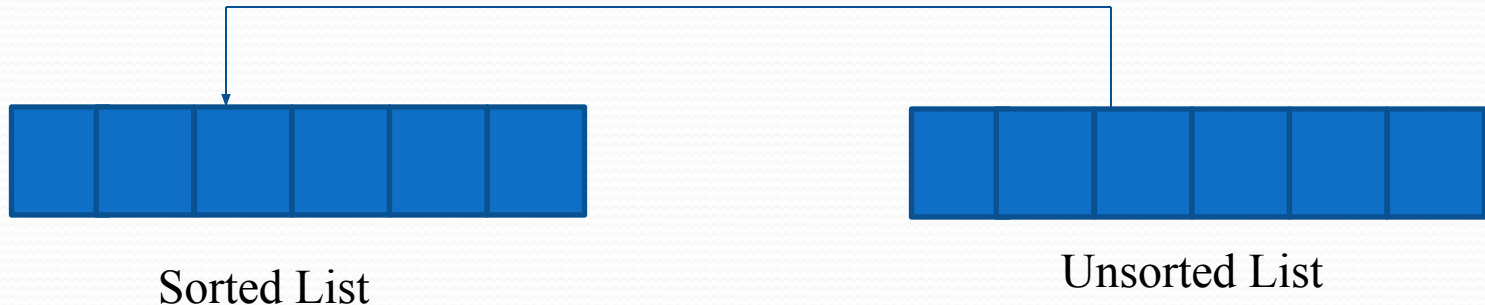
- Simple sorting method
- No additional data structure is required.
- It is in-place & stable sorting method

## **Disadvantages:**

Even if the elements are in sorted order , all  $n-1$  passes will be done.

# Insertion Sort

- In this method, we insert an unsorted element in its correct position in a sorted set of elements.
- We select one element from the unsorted set at a time & insert it into its correct position in the sorted set. This process continues till the last element.



Unsorted Data: 15 ,7,22,3,14,2

Step	Sorted List	Unsorted List	Action
0	empty	15 7 22 3 14 2	-----
1	15	7 22 3 14 2	Select 15,insert it in sorted set
2	7 15	22 3 14 2	Select 7,compare with 15
3	7 15 22	3 14 2	Select 22, compare with 15
4	3 7 15 22	14 2	Select 3,compare with 22,15,7
5	3 7 14 15 22	2	Select 14,compare with 22,15,7
6	2 3 7 14 15 22	empty	Select 2, compare with 22,15,14,7,3

# Unsorted Data: 21,3,5,12,11,17,26

Step	Sorted List	Unsorted List	Action
0	empty	21 3 5 12 11 17 26	-----
1	21	3 5 12 11 17 26	Select 21,insert it in sorted set
2	3 21	5 12 11 17 26	Select 3,compare with 21
3	3 5 21	12 11 17 26	Select 5, compare with 3, 21
4	3 5 12 21	11 17 26	Select 12,compare with 3,5,21
5	3 5 11 12 21	17 26	Select 11,compare with 3,5,12,21
6	3 5 11 12 17 21	26	Select 17, compare with 3,5,11,12,21
7	3 5 11 12 17 21 26	empty	Select 26, compare with 3,5,11,12,17,21

# Complexity Analysis of Insertion Sort

- **Best Case**-If initially data is sorted only one comparison is made in each pass i.e. each element will be compared only with last element.
- Total of  $n-1$  comparisons will be made. The best case time complexity is  $\Omega(n)$
- **Worst Case**-If data is in reverse order each element will be compared with each element in sorted list.
- Total no of comparisons for the  $n-1$  passes will be  $1+2+\dots+n-1=n(n-1)/2$  which is  $O(n^2)$

# Selection sort

- The list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst case complexities are of  $O(n^2)$ , where **n** is the number of items.

Consider this example



we search the whole list to find min value from array so 10 is the lowest value.



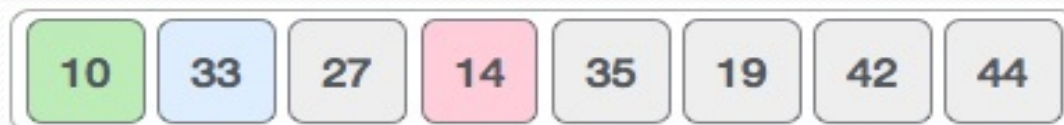
we replace 14 with 10. It appears in the first position of the sorted list.



For the second position i.e. 33, we start scanning the rest of the list in a linear manner.



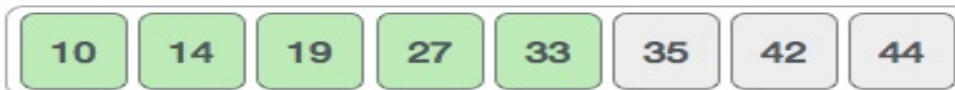
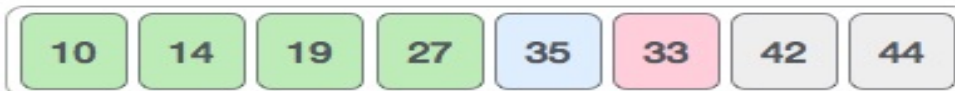
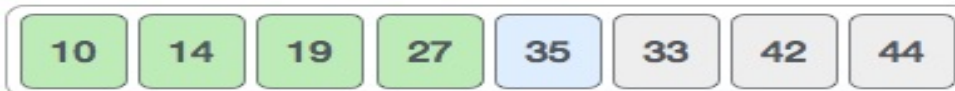
14 is the second lowest value in the list and it should appear at the second place.



Swap 14 & 33.



same process is applied to the rest of the items in the array.





(a)

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

(b)

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

(c)

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

(d)

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

(e)

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

(f)

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

(g)

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

- is to be swapped
  - Sorted

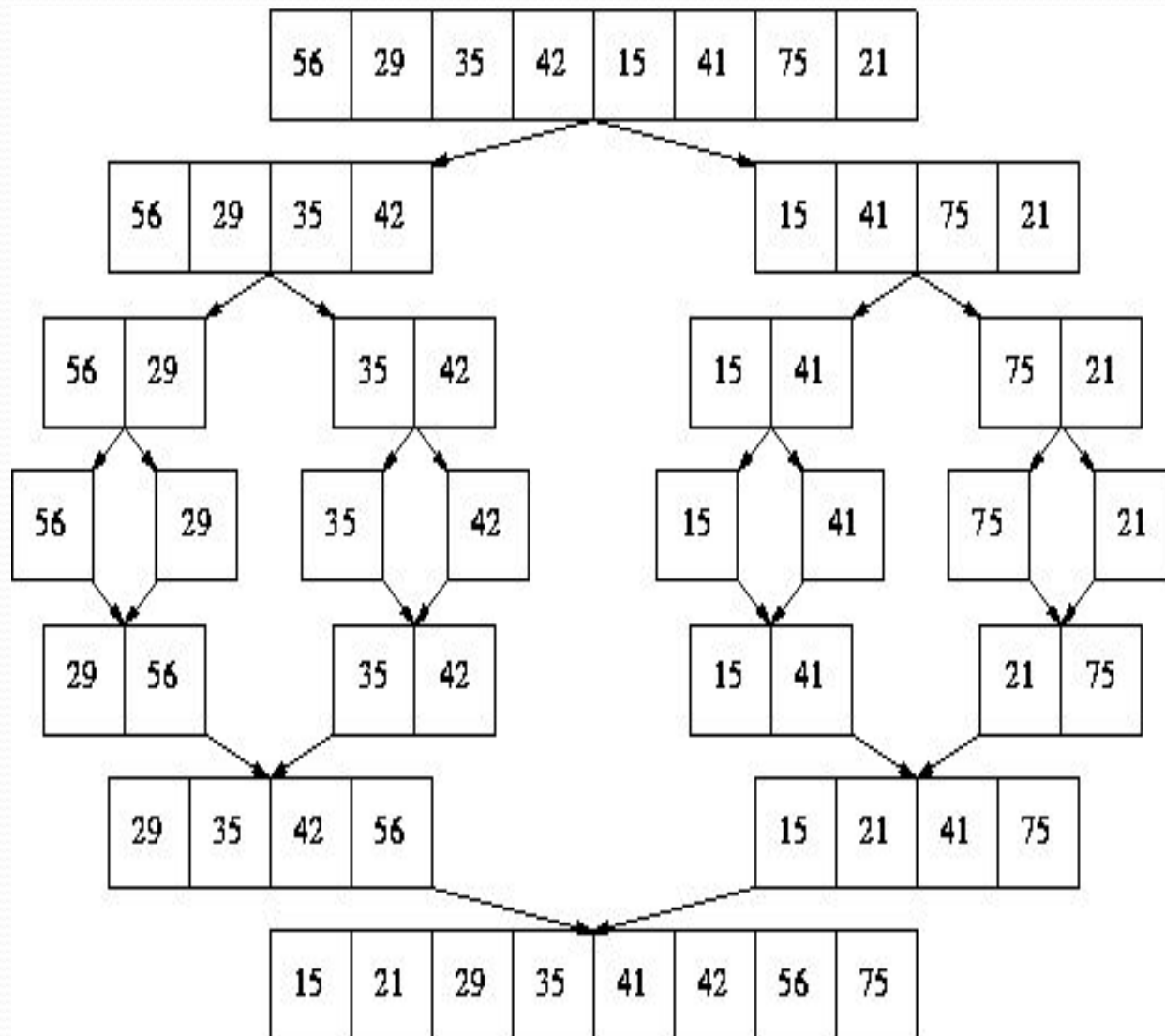
## Selection Sort

# Complexity Analysis of Selection Sort

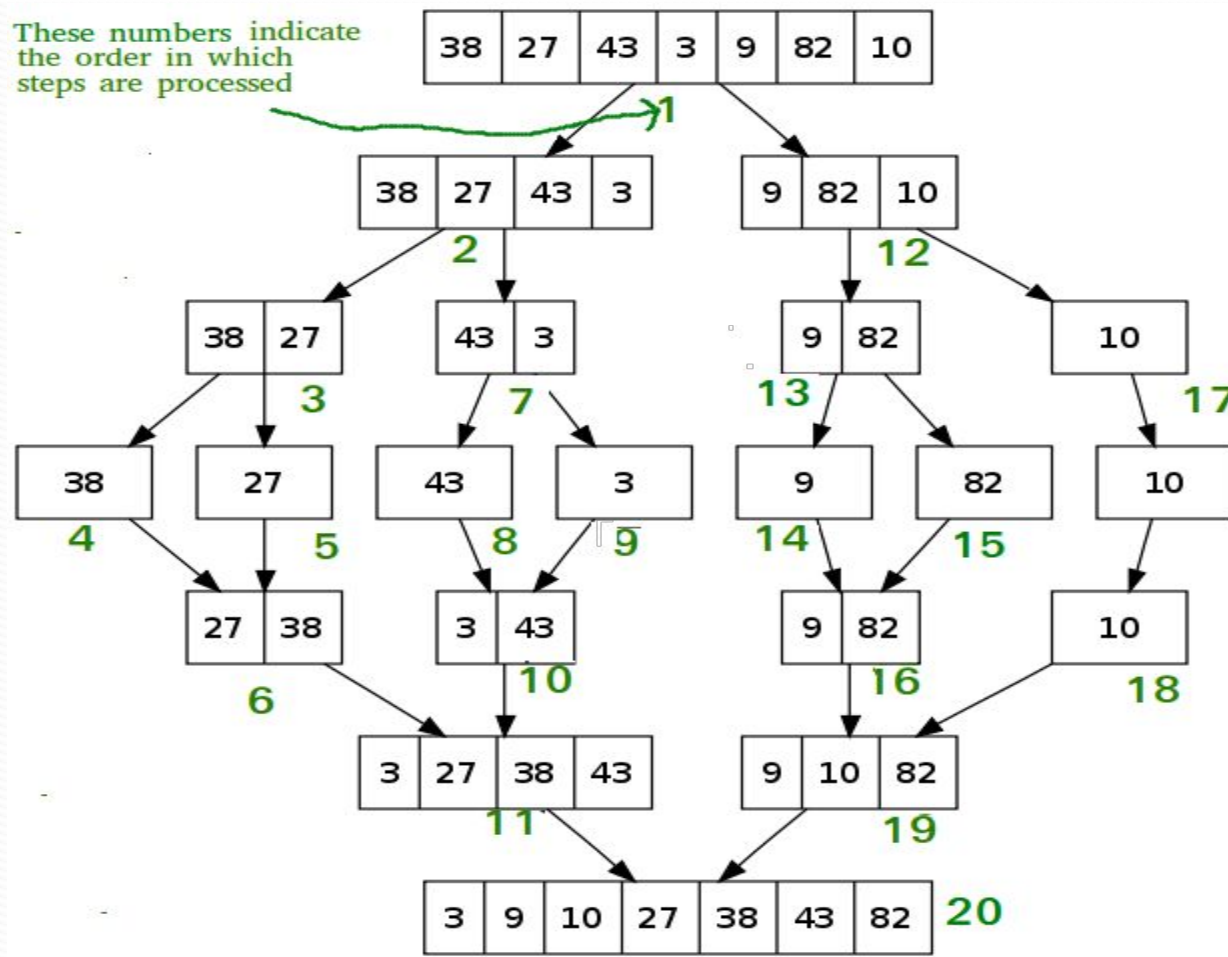
- The selection of smallest from  $n$  elements require  $n$  comparisons.
- To select second smallest element, it requires  $n-1$  comparisons.
- Total comparisons= $n+n-1+n-2+\dots\dots+1$   
 $=n(n-1)/2$   
 $=O(n^2)$

# Merge Sort

- Merge Sort is based on **divide-conquer** algorithm.
- **Divide**: Break the given problem into sub problems of same type.
- **Conquer**: Recursively solve these sub problems
- **Combine**: Appropriately combine the answers
- It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.
- **The merge() function** is used for merging two halves. The merge(a, l, m) is key process that assumes that arr[l..m] and arr[m+1..h] are sorted and merges the two sorted sub-arrays into one.



A classic example of Divide and Conquer is Merge Sort demonstrated below.



# Complexity Analysis of Merge Sort

Total number of iterations in Merge Sort are  $\log_2 n$ .

In each iteration,  $n$  elements are merged. Hence the time complexity of Merge Sort is  $O(n \log_2 n)$

# Quick Sort

- Efficient sorting method
- Called as **Partition Exchange Sort**
- It is based on **divide & conquer** technique
- We choose one element at a time as pivot & place in its correct position.
- Select pivot such that all elements to the left are less than pivot & all the elements to the right are greater.
- This divide array into 2 parts as left & right partition respectively.
- Go on repeating the same process till no more partition can be made.



Left Partition

Right Partition

# Algorithm

Partition(A,lb,ub)

Step1: Start

Step2:down=lb

Step3:up=ub

Step4:pivot=A[lb]

Step5:while(A[down]<=pivot && down<ub)  
    down++

Step6:while(A[up]>pivot && up>lb)  
    up--

Step7:if(down<up)  
    interchange A[down] & A[up]  
    goto step 5

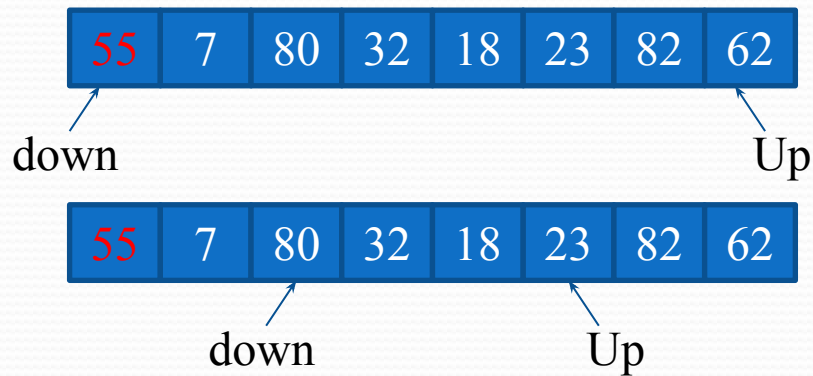
Step8:Interchange A[up] & A[lb] i.e Pivot

Step9: return up

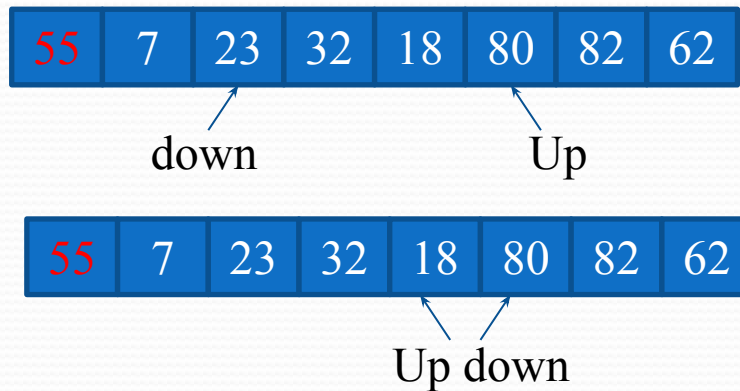
Step10:stop



# Sort using quick sort 55,7,80,32,18,23,82,62

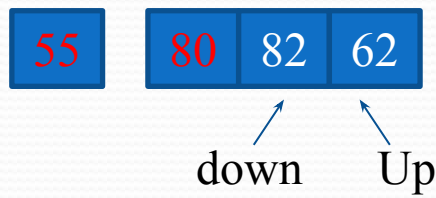
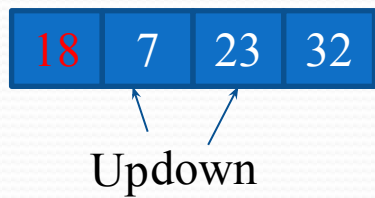
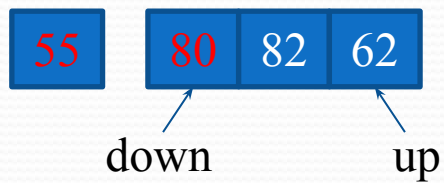
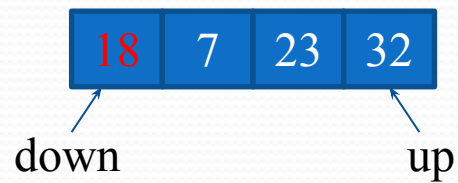


Down < up 2 < 5 Swap A[down] & A[up] i. e 80 & 23



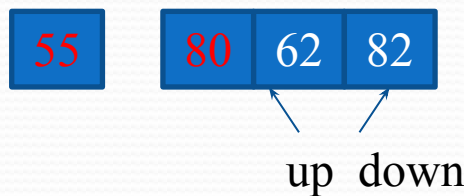
Interchange Pivot with A[up]





Swap Pivot & A[up]

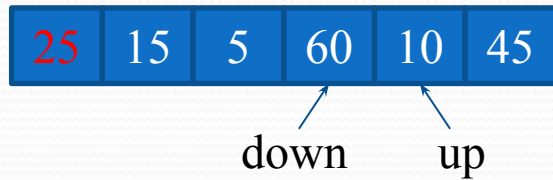
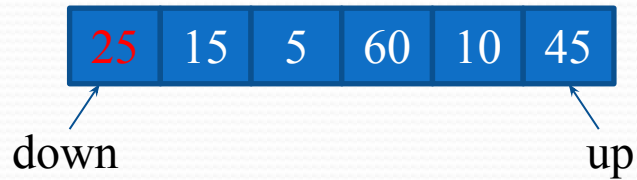
Swap A[down] & A[up]  
82 & 62



Swap Pivot & A[up]  
i. e 80 & 60

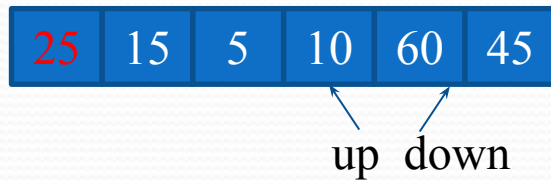


# Sort using quick sort 25, 15, 5, 60, 10, 45



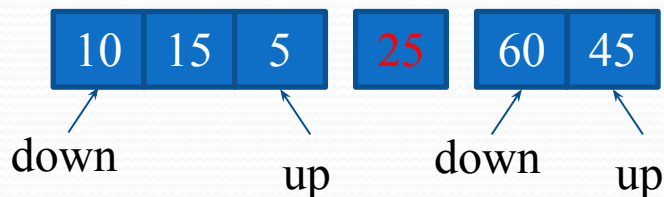
down < up i. e. 3 < 4

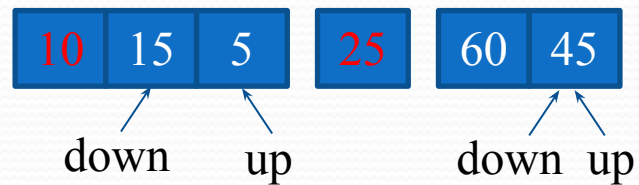
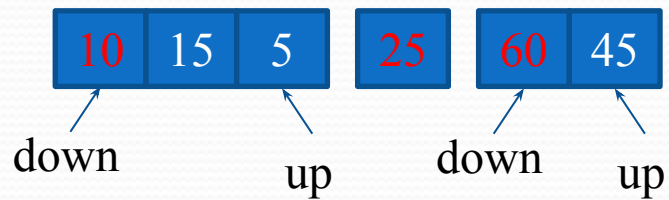
Swap A[down] & A[up] i. e. 60 & 10



down > up i. e. 4 > 3

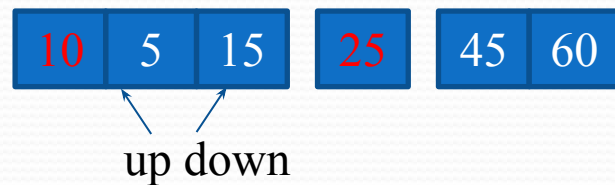
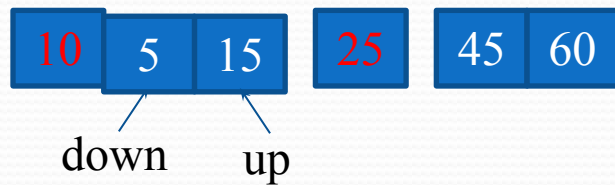
Swap pivot & A[up] i. e. 25 & 10



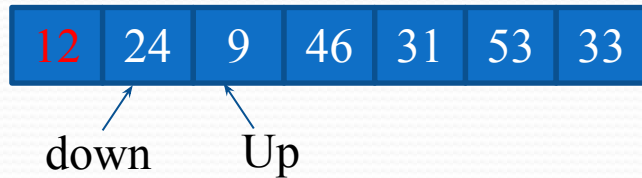
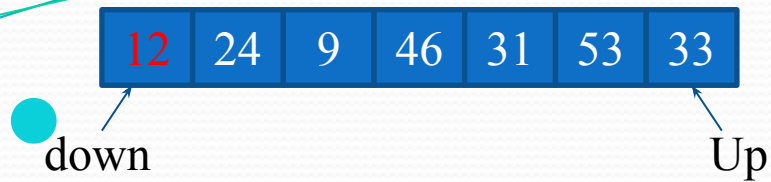


down < up i. e.  $1 < 2$

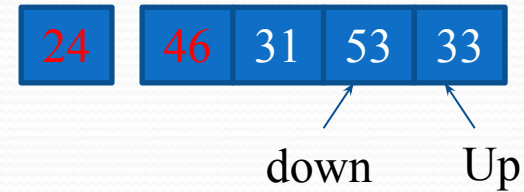
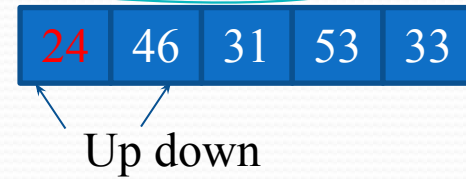
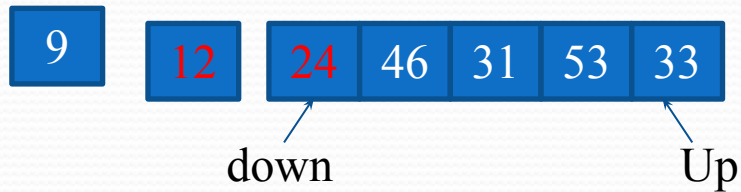
Swap A[down] & A[up] i. e. 15 & 5



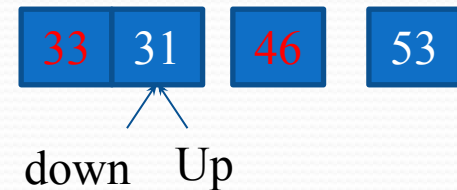
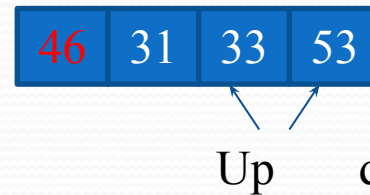
12, 24, 9, 46, 31, 53, 33



Swap A[up] & pivot i. e. 9 & 12



Swap 53 & 33



# Non Comparison Based Sorting

- All comparison based methods have a pre-condition.
- Data to be sorted must be satisfy some pre-determined requirements.
- It is done using some information about some internal characteristics of data elements , so it is not always possible to apply these methods.

Example of this are:

- Counting Sort
- Radix Sort

# Counting Sort

- Non-Comparison based sorting technique.
- Counting sort is a sorting technique based on keys between a specific range.
- It is based on observation that position of an element in the sorted list depends upon number of elements less than it.
- E. G If there are  $x$  elements  $< \text{key}$ , the proper position of key will be  $x+1$  in the sorted list.

## Steps of Counting Sort:

1. The range of elements is 0 to  $k-1$
2. Calculate the number of times each unique element  $X_i$  (each  $X_i$  is in range 0 to  $k-1$ ) occurs in the set.  $K$  number of counters are needed.
3. From the occurrences, calculate the number of elements preceding each element i.e for each element  $X_i$  we find  $\text{count}[i] = \text{total number of elements} < X_i$
4. Read each  $A[i]$  & use the counters to 'construct' the sorted array

## Arrays

$A[0.....n-1]$  :- Original unsorted array

$B[0.....n-1]$  :- Final sorted array

$C[0.....k-1]$  :- array to store counts

# Algorithm of counting sort

Step 1: Start

Step 2: for  $i=0$  to  $k-1$  -> Initialize counts

$C[i]=0$

Step 3: for  $i=0$  to  $n-1$  -> Count occurrences of each element  $A[i]$

$C[A[i]]+=1$

Step 4: for  $i=1$  to  $k-1$  ->  $C[i]$  contains number of elements  $\leq$  each unique element  
 $C[i]=C[i]+C[i-1]$

Step 5: for  $i=n-1$  to  $0$  -> construct sorted array

$B[C[A[i]]-1]=A[i]$

$C[A[i]]-=1$

Step 6: Stop



Consider  $n=6$  elements in the range 0 to 4

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

Calculate frequency of each element 0 to 4

C	0	1	2	3	4
	2	1	0	2	1

Now calculate ending positions of each element 1 to 4

$$C[i] = C[i] + C[i-1]$$

$$C[1] = C[1] + C[0] = 1 + 2 = 3$$

$$C[2] = C[2] + C[1] = 0 + 3 = 3$$

$$C[3] = C[3] + C[2] = 2 + 3 = 5$$

$$C[4] = C[4] + C[3] = 1 + 5 = 6$$

C	0	1	2	3	4
	2	3	3	5	6

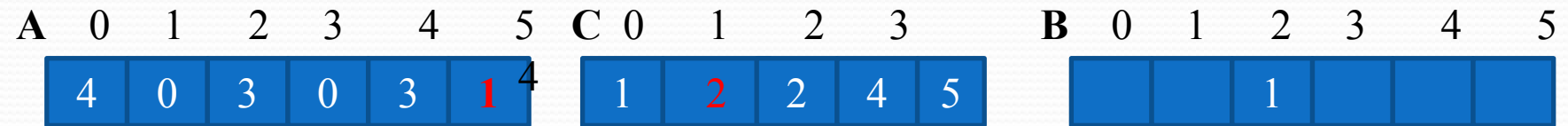
Reduce all positions by 1

C	0	1	2	3	4
	1	2	2	4	5

Each element  $C[i]$  indicates ending position of element  $i$ .

Read array  $A$  backwards & place  $A[i]$  in its correct position in  $B$ .  $B[C[A[i]]] = A[i]$

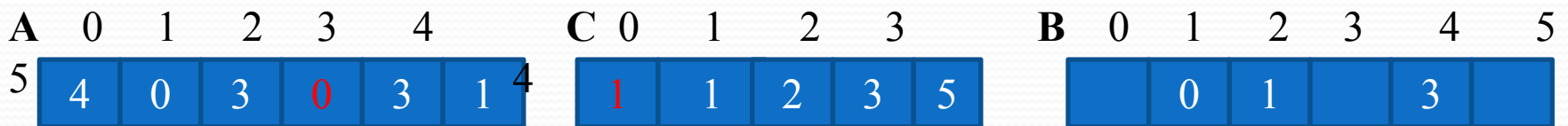
1.  $A[i]=1, C[A[i]]=2$ . Place 1 at position  $B[2]$ . Reduce  $C[1]$  by 1



2.  $A[i]=3, C[A[i]]=4$ . Place 3 at position  $B[4]$ . Reduce  $C[3]$  by 1



3.  $A[i]=0, C[A[i]]=1$ . Place 0 at position  $B[1]$ . Reduce  $C[0]$  by 1



4.  $A[i]=3$ ,  $C[A[i]]=3$ . Place 3 at position  $B[3]$ . Reduce  $C[3]$  by 1

A	0	1	2	3	4		C	0	1	2	3		B	0	1	2	3	4	5
5	4	0	3	0	3	1	4	0	1	2	3	5			0	1	3	3	

5.  $A[i]=0$ ,  $C[A[i]]=0$ . Place 0 at position  $B[0]$ . Reduce  $C[0]$  by 1

A	0	1	2	3	4		C	0	1	2	3		B	0	1	2	3	4	5
5	4	0	3	0	3	1	4	0	1	2	2	5		0	0	1	3	3	

6.  $A[i]=4$ ,  $C[A[i]]=5$ . Place 4 at position  $B[5]$ . Reduce  $C[4]$  by 1

A	0	1	2	3	4		C	0	1	2	3		B	0	1	2	3	4	5
5	4	0	3	0	3	1	4	0	1	2	2	5		0	0	1	3	3	4



## **Advantages:**

- It is only used with integer data
- Linear sorting method

## **Disadvantages:**

- Range of the elements must be known in advance.
- If range of elements is large ,it is highly inefficient.

Time Complexity of counting sort is  $O(n+k)$

# Radix Sort

- Non Comparison based sorting method which sorts in linear time.
- Precondition of radix sort is that all elements must be of the same length.
- It is used to sort numbers , strings etc
- Idea of this sort is to sort numbers digit by digit, sort starting from least significant digit to most significant digit & each digit is in the range 0 to 9
- We can use any non comparison based method to sort the digits
- Only requirement is that sorting method should be stable.
- It uses counting sort to sort digits.

## **Algorithm**

RADIX-SORT(A, d)

For  $i=1$  to  $d$  do

Use stable sort to sort array A on digit  $i$

Consider the following set of  $n=6$  elements. All elements are 3 digit numbers.

439, 372, 816, 602, 314, 149

Sort by Last Digit      Sort by Middle Digit      Sort by First Digit      Sorted

4	3	9		3	7	2		6	0	2		1	4	9		
3	7	2		6	0	2		3	1	4		3	1	4		
8	1	6		3	1	4		8	1	6		3	7	2		
6	0	2		8	1	6		4	3	9		4	3	9		
3	1	4			4	3	9		1	4	9			6	0	2
1	4	9		1	4	9		3	7	2				8	1	6

Show all the steps of sorting the following data using Radix Sort:

BUG, TEA, GUY, ATE, SEA, FOX, COW, DOG, BAG

Sort by Last Digit

B	U	G
T	E	A
G	U	Y
A	T	E
S	E	A
F	O	X
C	O	W
D	O	G
B	A	G

Sort by Middle Digit

T	E	A
S	E	A
A	T	E
B	U	G
D	O	G
B	A	G
C	O	W
F	O	X
G	U	Y

Sort by First Digit

B	A	G
T	E	A
S	E	
A		
D	O	G
C	O	W
F	O	X
A	T	E
B	U	G
G	U	
Y		

Sorted

A	T	E
B	A	G
B	U	G
C	O	W
D	O	G
F	O	X
G	U	Y
S	E	A
T	E	A

# Radix Sort

## **Advantages**

- It sorts in linear time
- It is stable
- Fast when elements are short

## **Disadvantages**

- It requires fixed size elements & some standard way of breaking the elements into smaller parts
- It depends on number of digits
- Requires additional data structure

Time Complexity of counting sort is  $O(d(n+k))$



# Comparison of sorting methods

Method	Type	Best Complexity	Worst Case Complexity	In-place	Stable
Bubble	Comparison-Based	$\Omega(n^2)$	$O(n^2)$	Yes	Yes
Insertion	Comparison-Based	$\Omega(n)$	$O(n^2)$	Yes	Yes
Selection	Comparison-Based			Yes	Yes
Merge	Comparison-Based	$\Omega(n^2)$	$O(n^2)$	No	Yes
Quick	Comparison-Based	$\Omega(n \log_2 n)$	$O(n \log_2 n)$	Yes	No
Counting	Non Comparison-Based	$\Omega(n \log_2 n)$	$O(n^2)$	No	Yes
Radix	Comparison-Based	$\Omega(n)$	$O(n+k)$	No	Yes
		$\Omega(n)$	$O(d(n+k))$		

# Comparison of Searching Methods

Sequential Search	Binary Search
No Precondition : Elements need not be in sorted order.	Precondition: Elements must be in sorted order
Iterative Algorithm	Based on Divide & Conquer
Requires sequential access to elements	Requires random access to elements
Search starts from the first element	Search starts from the middle element
Best case time complexity	Best case time complexity
Worst case time complexity	Worst case time complexity
Inefficient when number of elements are very large	Very efficient
There are different variations like sentinel, Probability & Ordered List search	Variant is ternary search