Insertion Sort

Introduction

The Idea of the insertion sort is similer to the Idea of sorting the Playing cards .

© Prof. Pramod Nath

Insertion Sorting

•It is a simple Sorting algorithm which sorts the array by shifting elements One by one. Following are some of the important characteristics of Insertion Sort.

- It has one of the simplest implementation
- It is efficient for smaller data sets, but very inefficient for larger lists.
- Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
- It is better than Selection Sort and Bubble Sort algorithms.
- It is Stable, as it does not change the relative order of elements with equal keys

© Prof. Pramod Nath

5	1	6	2	4	3

Lets take this Array.

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

(Always we start with the second element as key.)

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

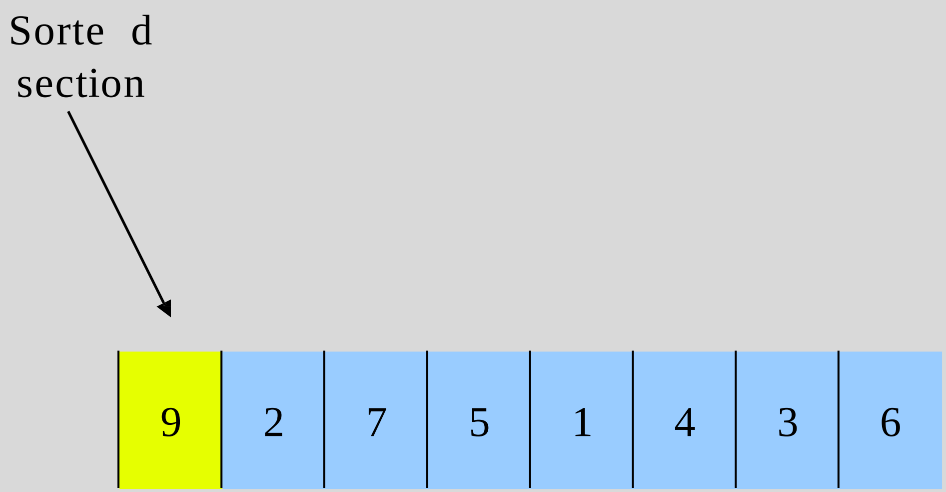
© Prof. Pramod Nath

Insertion Sort

Example :

9	2	7	5	1	4	3	6

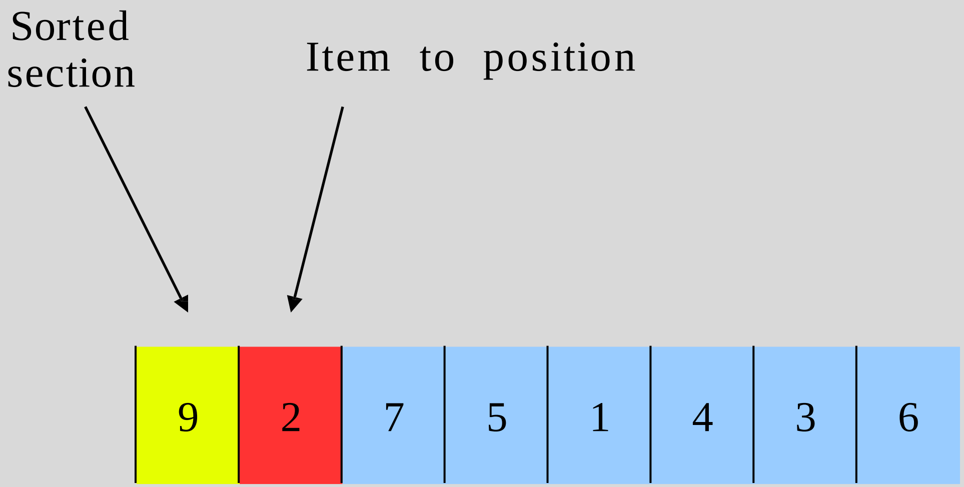
Insertion Sort



We start by dividing the array in a sorted section and an unsorted section. We put the first element as the only element in the sorted section, and the rest of the array is the unsorted section.

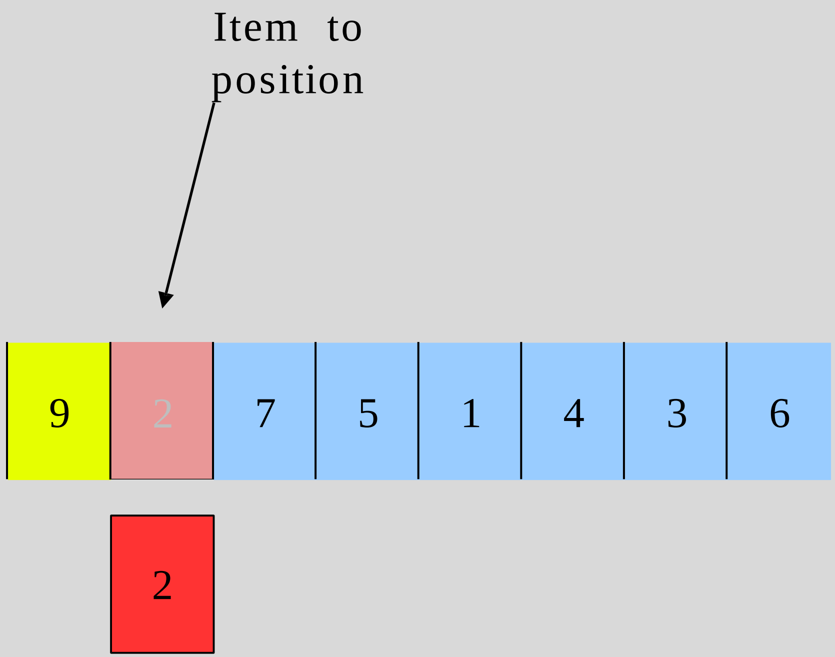
© Prof. Pramod Nath

Insertion Sort



The first element in the unsorted section is the next element to be put into the correct position.

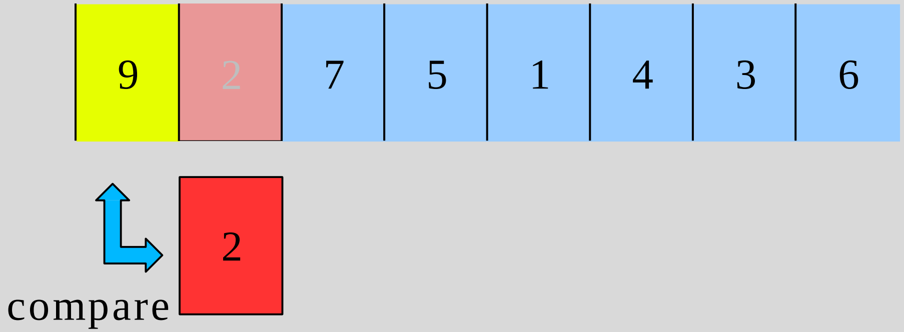
Insertion Sort



We copy the element to be placed into another variable so it doesn't get overwritten.

© Prof. Pramod Nath

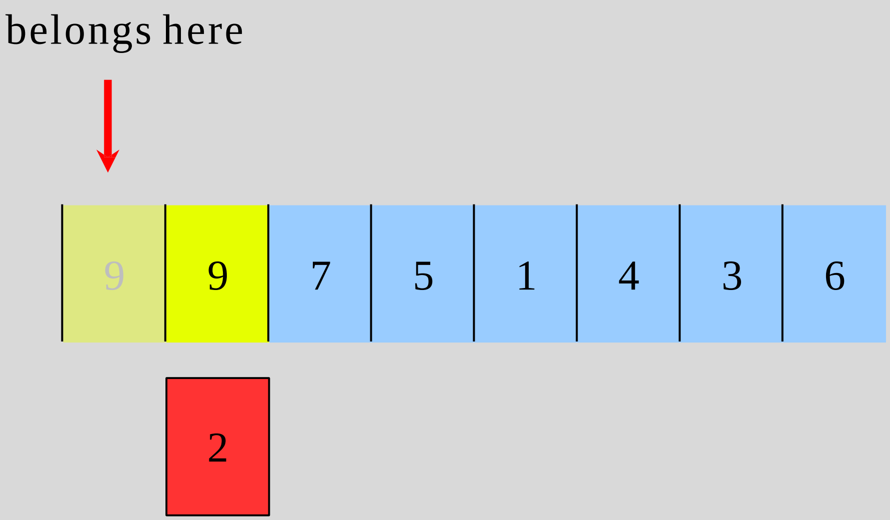
Insertion Sort



If the previous position is more than the item being placed, copy the value into the next position

© Prof. Pramod Nath

Insertion Sort



If there are no more items in the sorted section to compare with, the item to be placed must go at the front.

© Prof. Pramod Nath

Insertion Sort

2	9	7	5	1	4	3	6

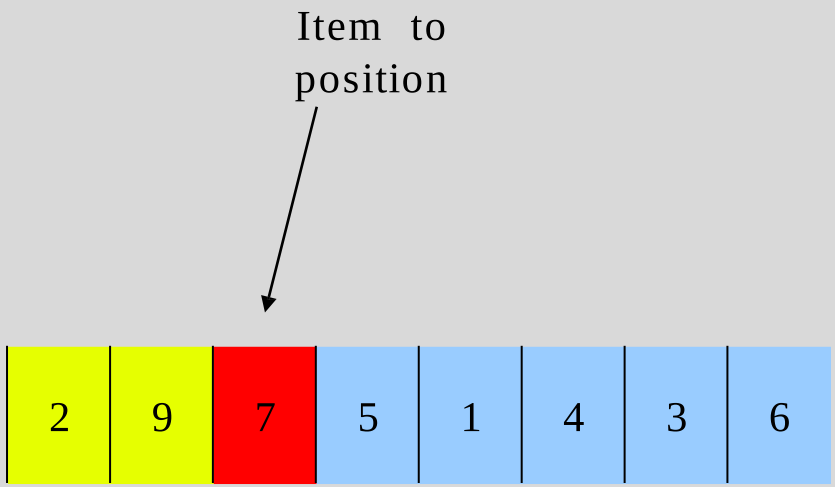
© Prof. Pramod Nath

Insertion Sort

2	9	7	5	1	4	3	6

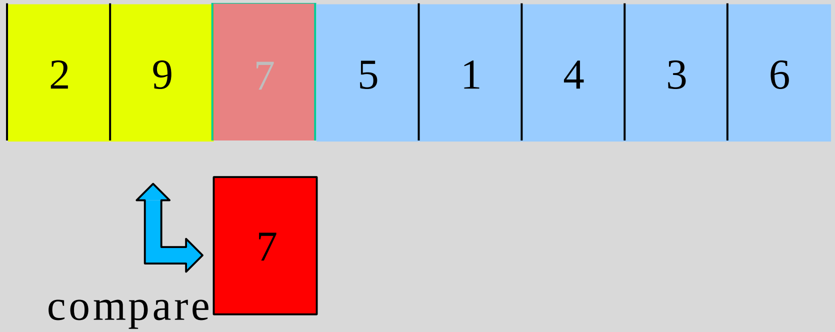
© Prof. Pramod Nath

Insertion Sort



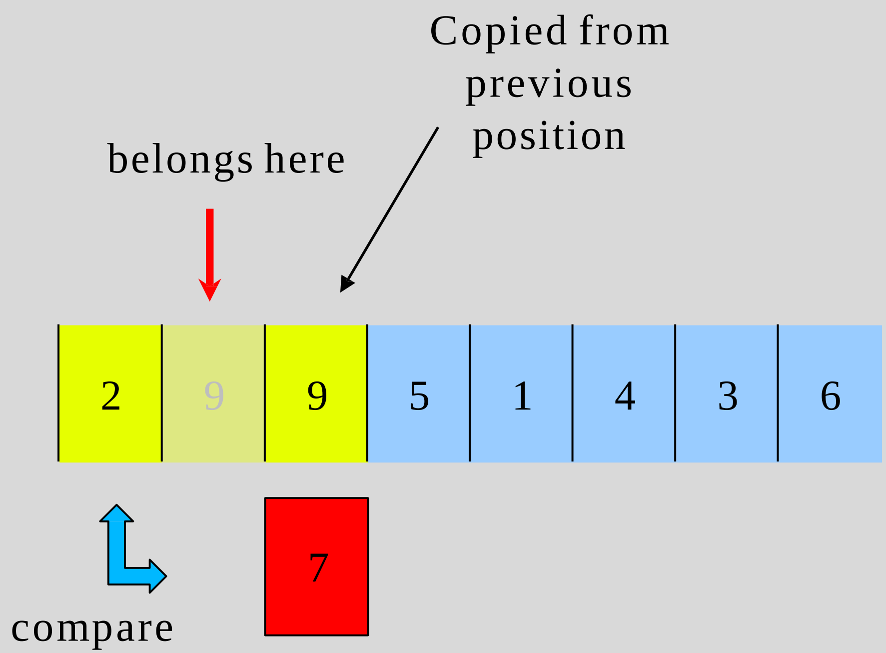
© Prof. Pramod Nath

Insertion Sort



© Prof. Pramod Nath

Insertion Sort



If the item in the sorted section is less than the item to place, the item to place goes *after* it in the array.

© Prof. Pramod Nath

Insertion Sort

2	7	9	5	1	4	3	6

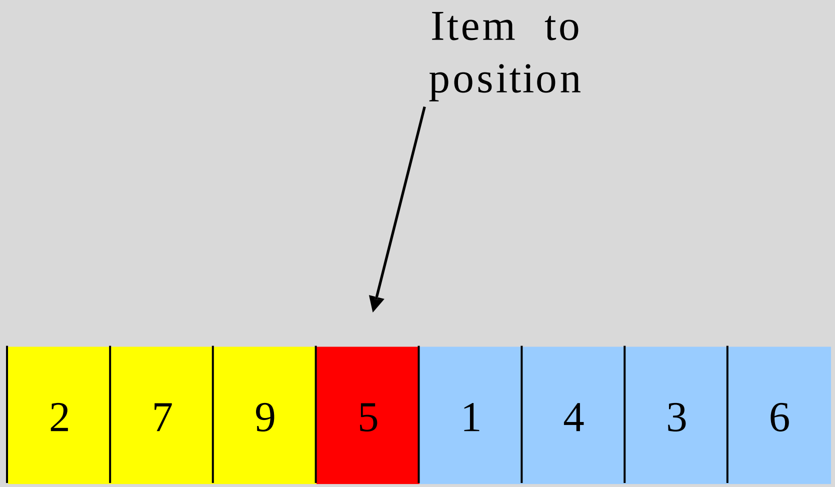
© Prof. Pramod Nath

Insertion Sort

2	7	9	5	1	4	3	6

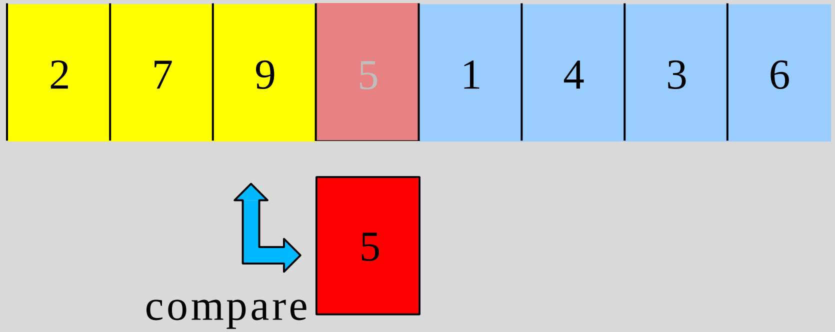
© Prof. Pramod Nath

Insertion Sort



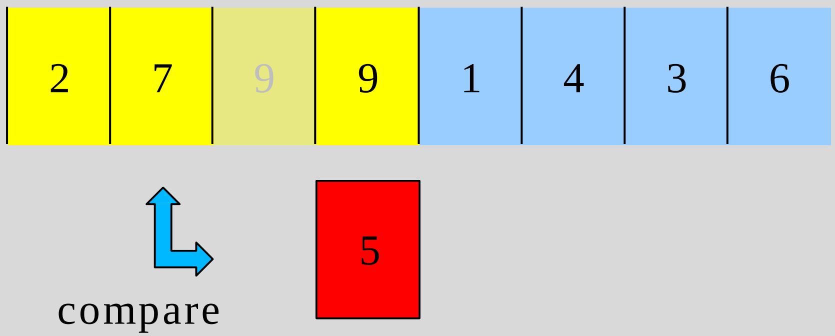
© Prof. Pramod Nath

Insertion Sort



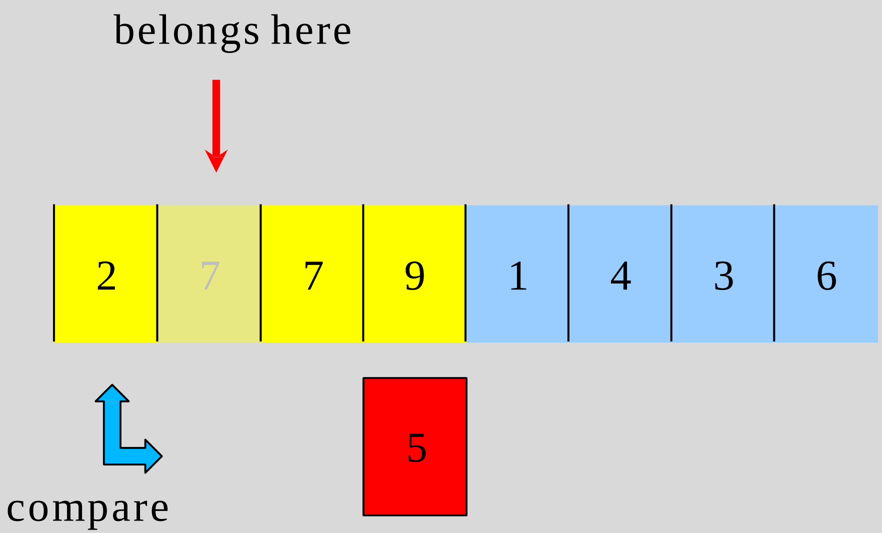
© Prof. Pramod Nath

Insertion Sort



© Prof. Pramod Nath

Insertion Sort



© Prof. Pramod Nath

Insertion Sort

2	5	7	9	1	4	3	6

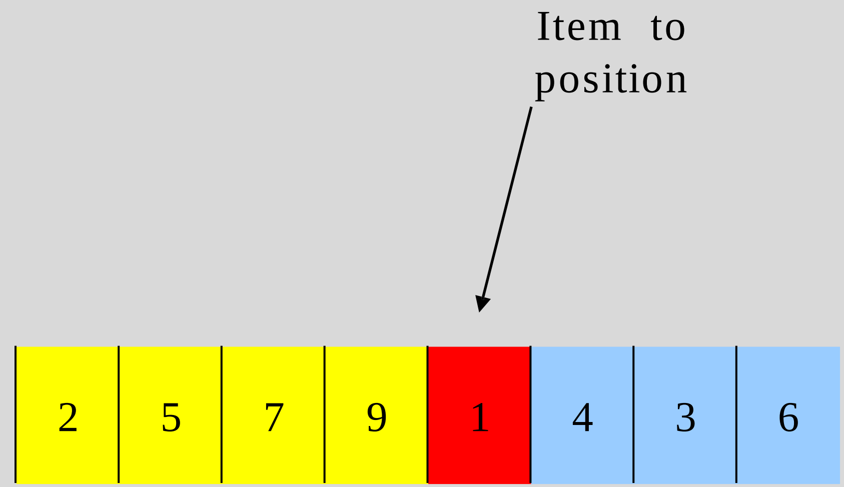
© Prof. Pramod Nath

Insertion Sort

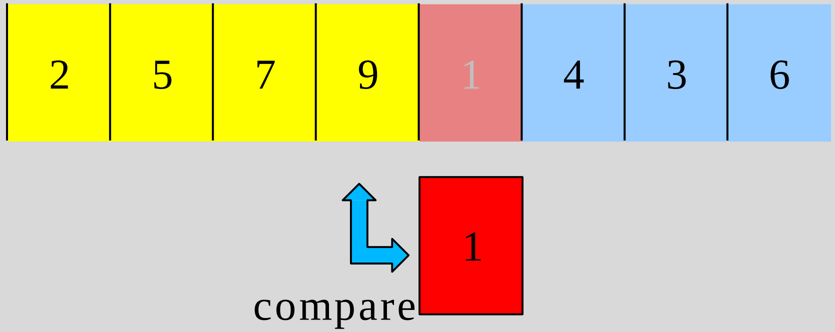
2	5	7	9	1	4	3	6

© Prof. Pramod Nath

Insertion Sort

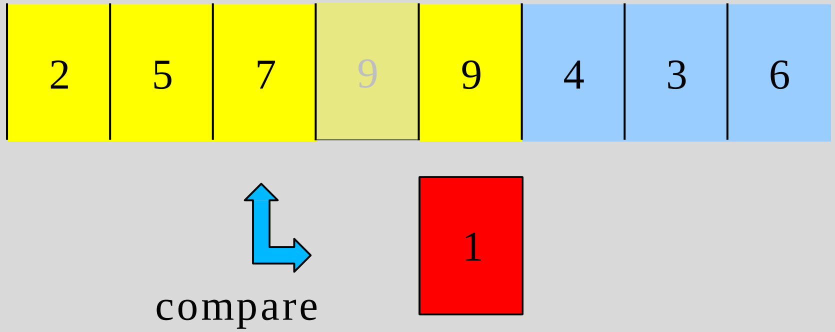


Insertion Sort



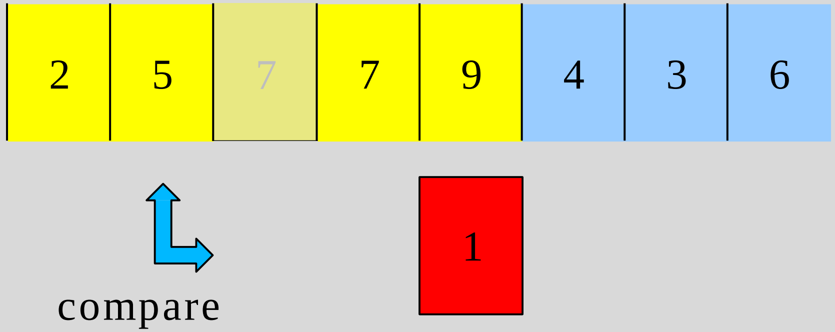
© Prof. Pramod Nath

Insertion Sort



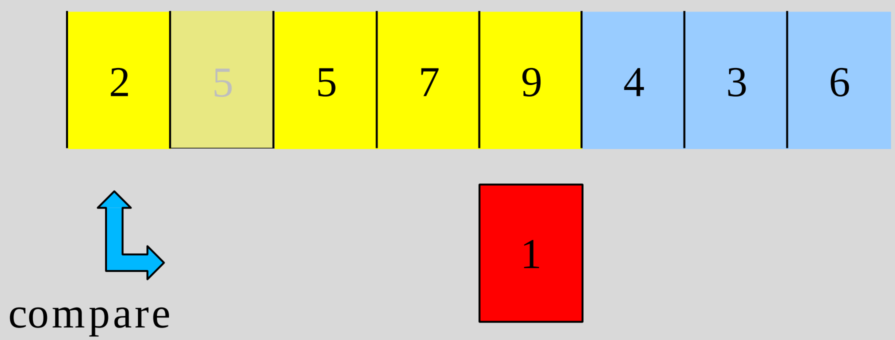
© Prof. Pramod Nath

Insertion Sort



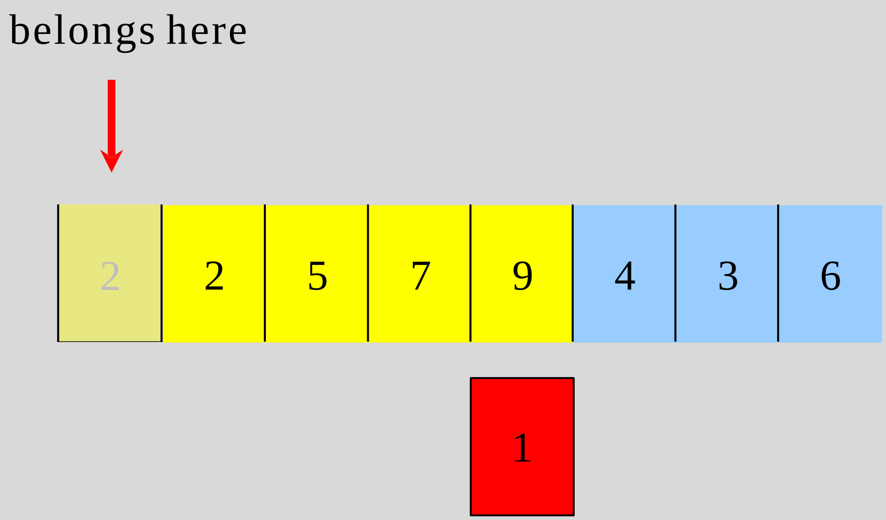
© Prof. Pramod Nath

Insertion Sort



© Prof. Pramod Nath

Insertion Sort



© Prof. Pramod Nath

Insertion Sort

1	2	5	7	9	4	3	6

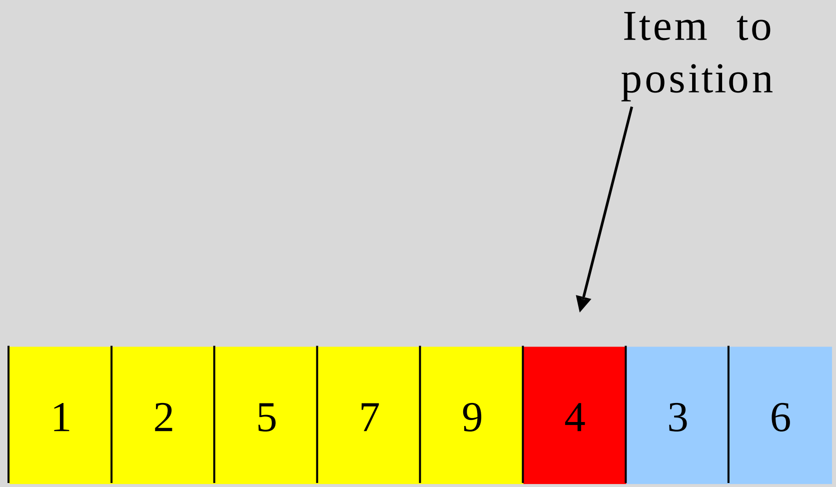
© Prof. Pramod Nath

Insertion Sort

1	2	5	7	9	4	3	6

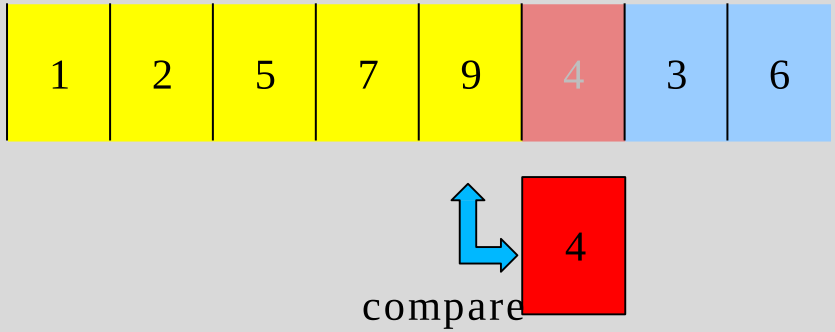
© Prof. Pramod Nath

Insertion Sort



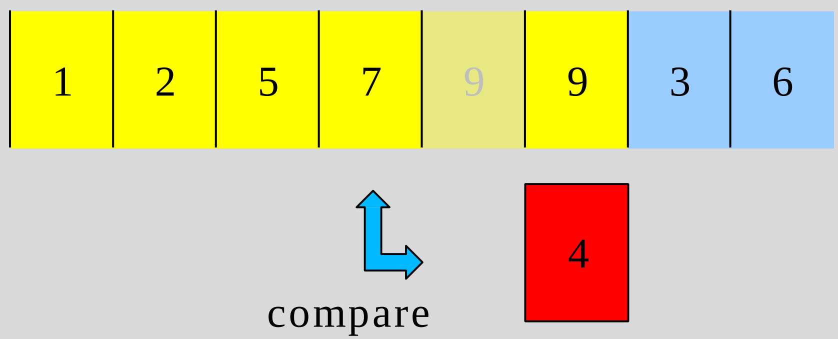
© Prof. Pramod Nath

Insertion Sort



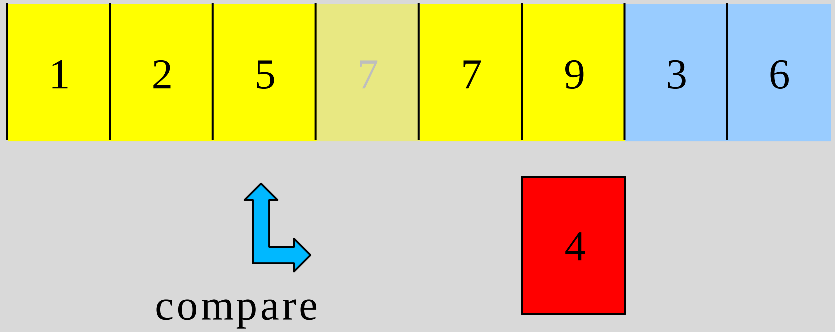
© Prof. Pramod Nath

Insertion Sort

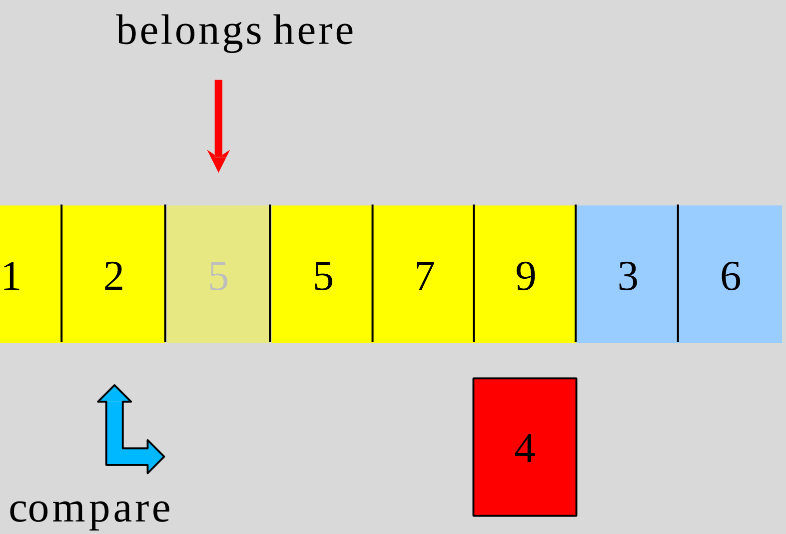


© Prof. Pramod Nath

Insertion Sort



Insertion Sort



© Prof. Pramod Nath

Insertion Sort

1	2	4	5	7	9	3	6

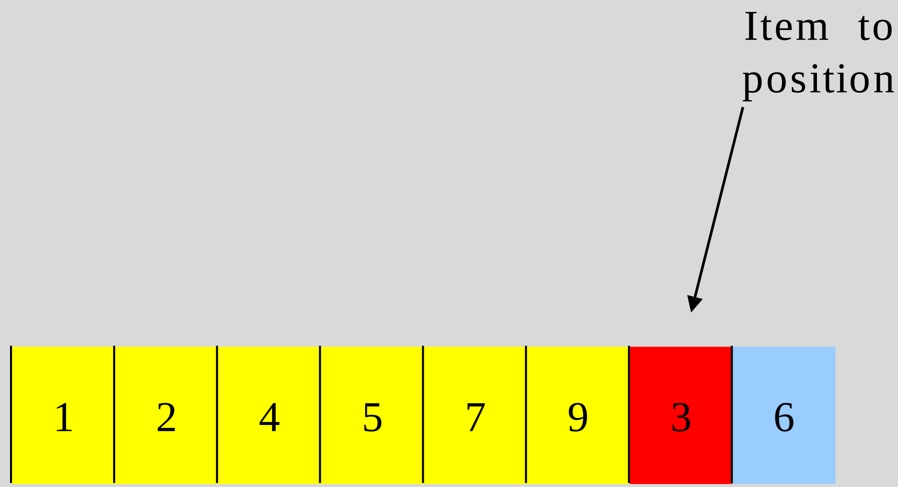
© Prof. Pramod Nath

Insertion Sort

1	2	4	5	7	9	3	6

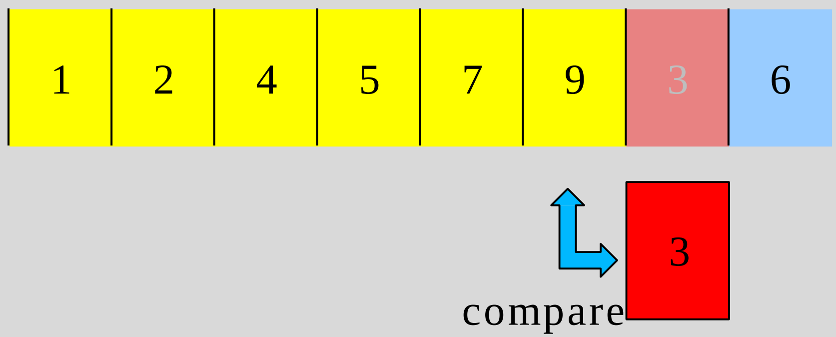
© Prof. Pramod Nath

Insertion Sort



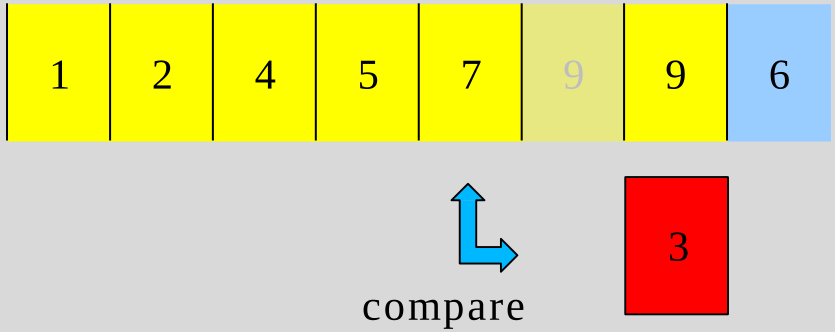
© Prof. Pramod Nath

Insertion Sort



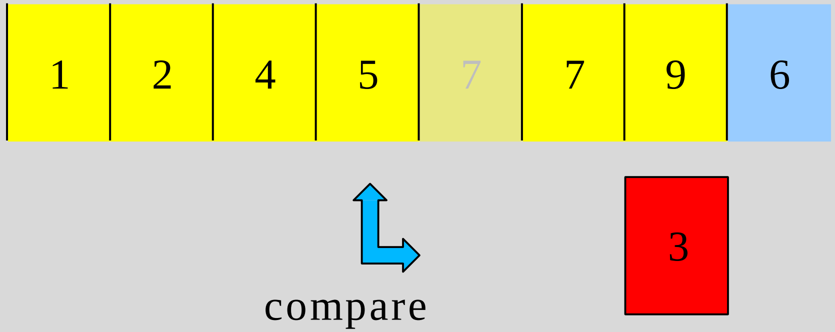
© Prof. Pramod Nath

Insertion Sort



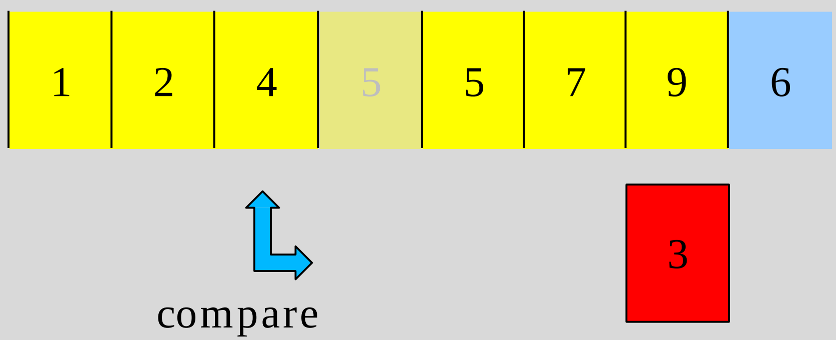
© Prof. Pramod Nath

Insertion Sort



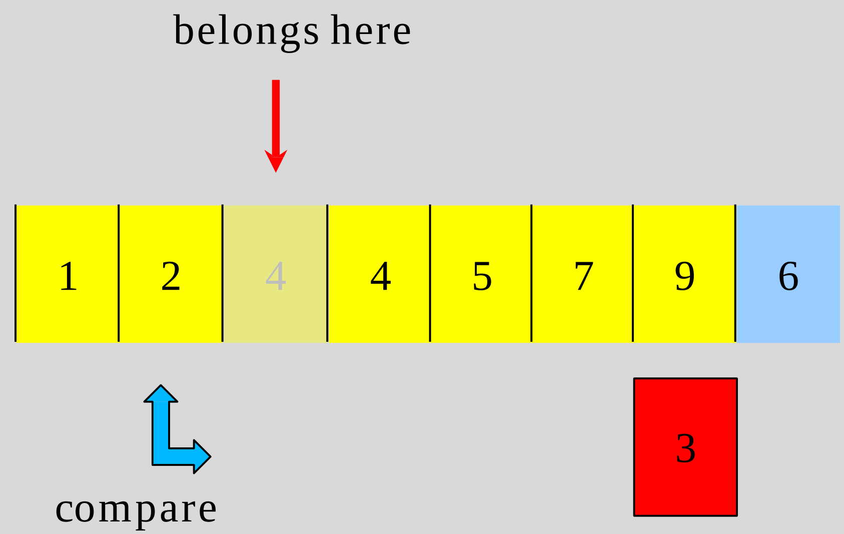
© Prof. Pramod Nath

Insertion Sort



© Prof. Pramod Nath

Insertion Sort



© Prof. Pramod Nath

Insertion Sort

1	2	3	4	5	7	9	6

© Prof. Pramod Nath

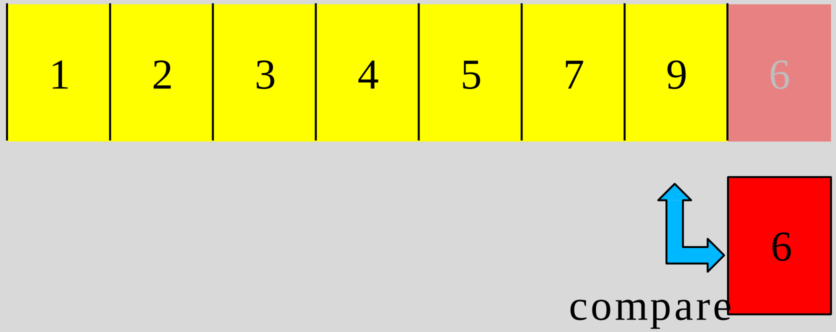
Insertion Sort

1	2	3	4	5	7	9	6

© Prof. Pramod Nath

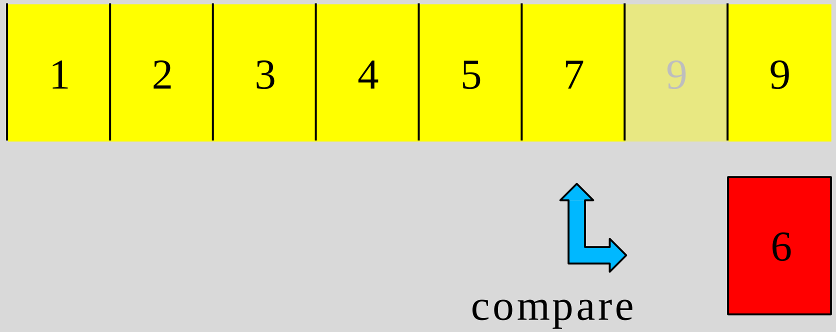
Insertion Sort

Insertion Sort



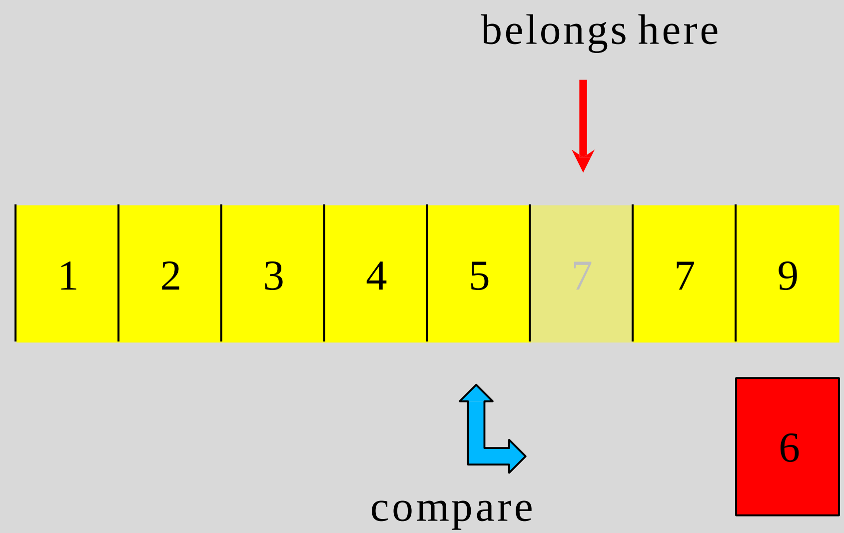
© Prof. Pramod Nath

Insertion Sort



© Prof. Pramod Nath

Insertion Sort



© Prof. Pramod Nath

Insertion Sort

1	2	3	4	5	6	7	9

© Prof. Pramod Nath

Insertion Sort

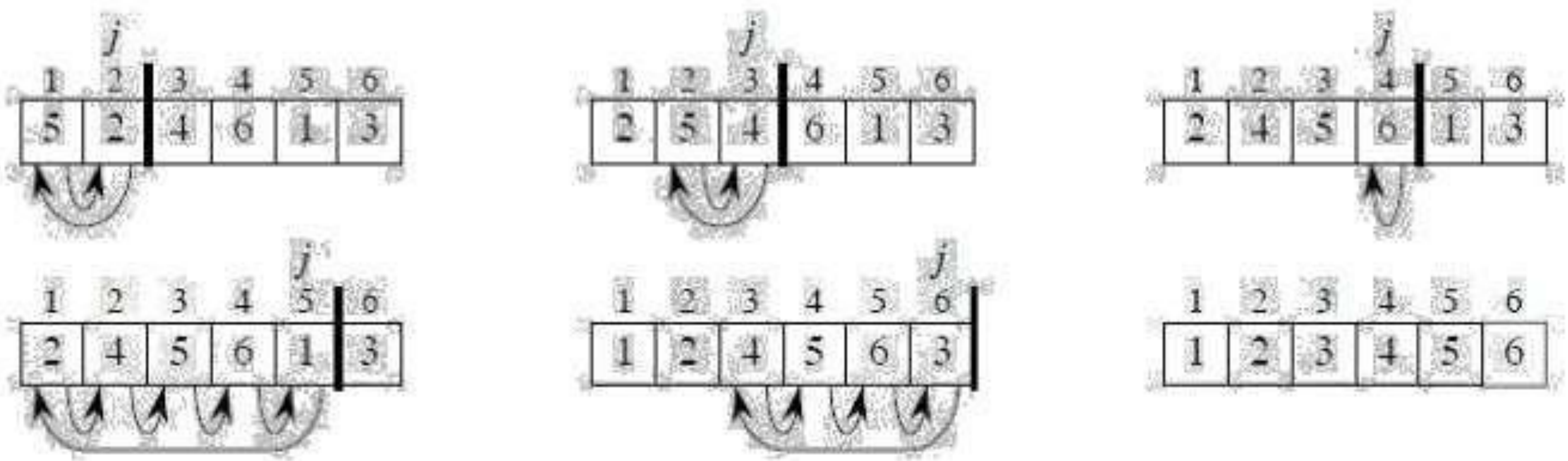
1	2	3	4	5	6	7	9

SORTED!

© Prof. Pramod Nath

--* (Way of working) *-*-*

Select – Compare – Shift - Insert



© Prof. Pramod Nath

Pseudo Code

- **for i = 0 to n – 1**

```
j = 1
while j > 0 and A[j] < A[j – 1]
        swap(A[j], A[j-1])
        j = j - 1
```

© Prof. Pramod Nath

CODE OF INSERTION SORT

```
void insertion_sort (int arr[ ], int length)
{
        int i, j, temp;

        for (i = 0; i < length; i++)
        {
                j = i;

          while (j > 0 && arr[j] < arr[j-1])
            {
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
                j--;
            }
        }
}
```

© Prof. Pramod Nath

Best Times to Use Insertion Sort

- When the data sets are relatively small.
  - ➢Moderately efficient
- When you want a quick easy implementation.
  - ➢Not hard to code Insertion sort.
- When data sets are mostly sorted already.
  - ➢(1,2,4,6,3,2)

© Prof. Pramod Nath

Worst Times to Use Insertion Sort

- When the data sets are relatively large.

- When data sets are completely unsorted
  - Absolute worst case would be reverse ordered. (9,8,7,6,5,4)

© Prof. Pramod Nath

- Advantages
  - Good running time for "almost sorted" arrays $\Theta(n)$
  - Simple implementation.
  - Stable, i.e. does not change the relative order of elements with equal keys

© Prof. Pramod Nath

Disadvantages

It is less efficient on list containing more number of elements.
As the number of elements increases the performance of the program would be slow.
Insertion sort needs a large number of element shifts.

© Prof. Pramod Nath

Insertion sort analysis

Best Case of Insertion Sort

10, 20, 30, 40,

		Comparison	Swap
Pass 1.	10	0	0
Pass 2.	10 | 20	1	0
Pass 3.	10 | 20 | 30	1	0
Pass 4.	10 | 20 | 30 | 40	1	0
		⋮	
		1	0
		$n-1$	0

$\therefore n-1+0$

$\Rightarrow O(n)$

If array is almost sorted, then insertion sort is best algo.

10	20	30	40	50	60	70	80	90	5
0	1	1	1	1	1	1	1	1	n

$2n$

$O(n)$

© Prof. Pramod Nath

Worst case

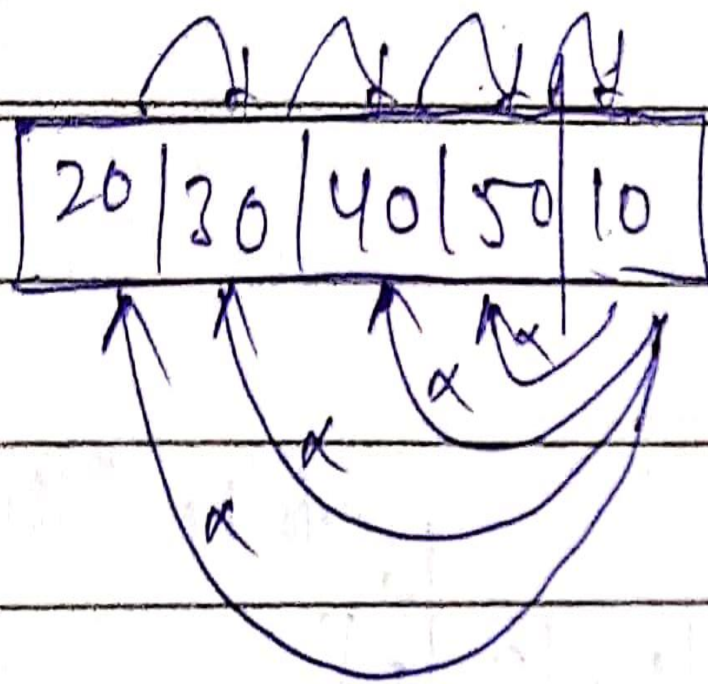
i/p: 50 40 30 20 10

		Linear search: Comparison	Swap
Pass ①	[50]	→ 0	0
Pass ②	[50 | 40] ⇒ [40 | 50]	→ 1	1
Pass ③	[40 | 50 | 30] $x = 30$ 2 Right Shift ⇓ [30 | 40 | 50]	→ 2	2
Pass ④	[30 | 40 | 50 | 20] $x = 20$ 3 Right Shift (∵ 3 failure) ⇓ [20 | 30 | 40 | 50]	→ 3	3

© Prof. Pramod Nath

L.S

Date______



$x = 10$

4 failure

4 right shift

C	S
$n-1$	$n-1$
$O(n^2)$	$O(n^2)$

$$\Downarrow$$

10	20	30	40	50

$2n^2$

$\therefore O(n^2)$

© Prof. Pramod Nath

- **Average case**

the order of growth =0(N^2)

the no of comparison and shift/swap is less than the selection and bubble sort.

- **Hence it is better than selection and bubble sort.**

© Prof. Pramod Nath

Important Points

Date

Note ① Insertion Sort Algo will take Best case (n-1) comparison and 0 swap.

∴ Best Case time Complexity = $O(n)$

② If array is almost sorted, insertion sort is preferable.

③ If array size is very less, insertion sort is preferable (no divide and no conquer, no combine).

Comparisons

© Prof. Pramod Nath

Comparison with Bubble Sort:

- **In it we set the largest element at the end in each iteration**
- **Time Complexity:**
- **Worst case: we have to do n comparisons for 1st iteration, n-1 for next and n-2 for next until we reach at 1.**

Time complexity= $O(n^2)$

•Best Case: when the list is already sorted

Time Complexity= $O(n)$

- **Average Case:**

Time Complexity=$O(n^2)$

© Prof. Pramod Nath

Comparison with Selection Sort:

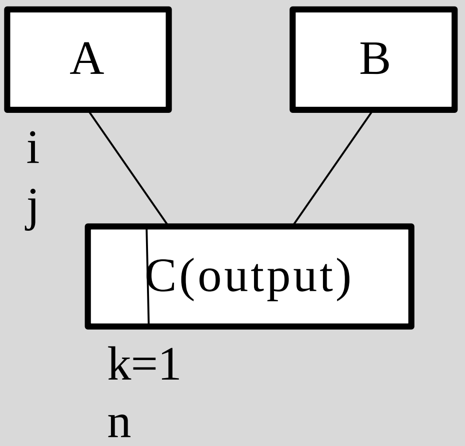
- In selection sort minimum element n the whole list will be placed at rightmost
- Worst case: Time complexity= $O(n^2)$ because we have to traverse the whole list
- Best Case: Time complexity= $O(n^2)$ because swaping will must happen at least one time
- Average Case: Time complexity= $O(n^2)$

© Prof. Pramod Nath

Comparison with Merge Sort:

- Divide and Conquer Rule
- We divide the array recursively until it reach to one element and then it get sorted according to comparisons and then merged again

```
for k=1 to n
     if(A[i]<B[j])
          C[k]=A[i];
           i++;
     else if(A[i]>B[j])
            C[k]=B[j]
             j++
```



Time Complexity:

- Best Case:

  Time Complexity= O(nlogn)
- Worst Case:

  Time Complexity= O(nlogn)
- Average Case:
- Time Complexity= O(nlogn)

Because we are dividing the array in this case, e.g if we have 32 elements in array then we will have to divide 5 times….so it will take logn times for dividing and n times for merging…. Total=nlogn

© Prof. Pramod Nath

Any Questions ?