BM267 - Introduction to Data Structures

6. Elementary Sorting Methods

Ankara University

Computer Engineering Department

Bulent Tugrul

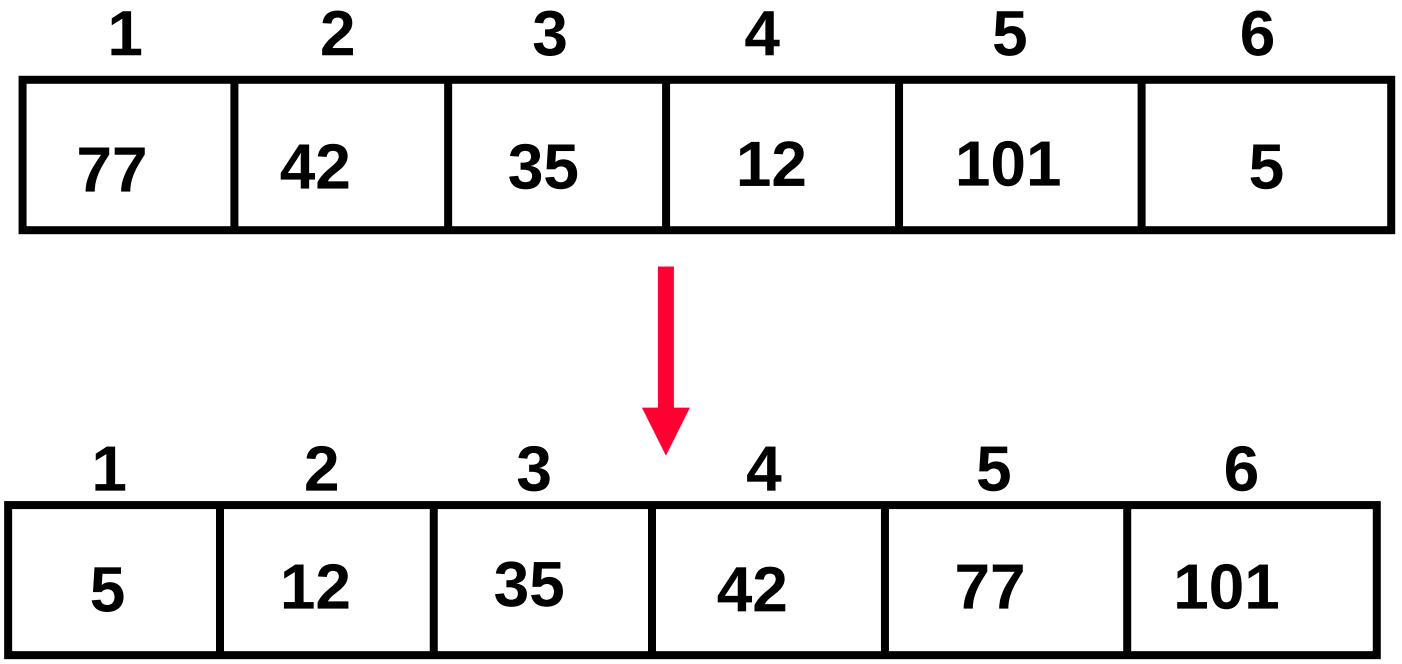
Objectives

Learn about

- Elementary sorting algorithms
  - Selection sort
  - Insertion sort
  - Bubble sort
  - Analysis of each sorting method

Sorting

- Sorting takes an unordered collection and makes it an ordered one.



Choosing a sorting algorithm

- Elementary sorting algorithms are usually slower, but easy to implement.
- The more complex algorithms are not always the preferred ones. (Needs more attention)
- Elementary algorithms are generally more appropriate in the following situations:
  - Less than a few hundred values to be sorted
  - The values will be sorted just once
  - Special cases such as:
    - the input data are "almost sorted"
    - there are many equal keys

Internal / external sorting

- If the file to be sorted can fit into computer's memory, then sorting method is called **internal sorting**.

- Sorting files from tape or disk is called **external sorting.**

- Partially sorted blocks need to be combined or merged in some manner to eventually sort the entire list

Element stability

- A sorting method is said to be stable if it preserves the relative order of items with duplicated keys in the file. Items with identical keys should appear in the same order as in the original input.

- For instance, consider sorting a list of student records alphabetically by name, and then sorting the list again, but this time by letter grade in a particular course. If the sorting algorithm is stable, then all the students who got "A" will be listed alphabetically.

- Stability is a difficult property to achieve if we also want our sorting algorithm to be efficient.

Element stability

Adams	A
Black	B
Brown	D
Jackson	B
Jones	D
Smith	A
Thompson	D
Washington	B
White	C
Wilson	C

Adams	A
Smith	A
Washington	B
Jackson	B
Black	B
White	C
Wilson	C
Thompson	D
Brown	D
Jones	D

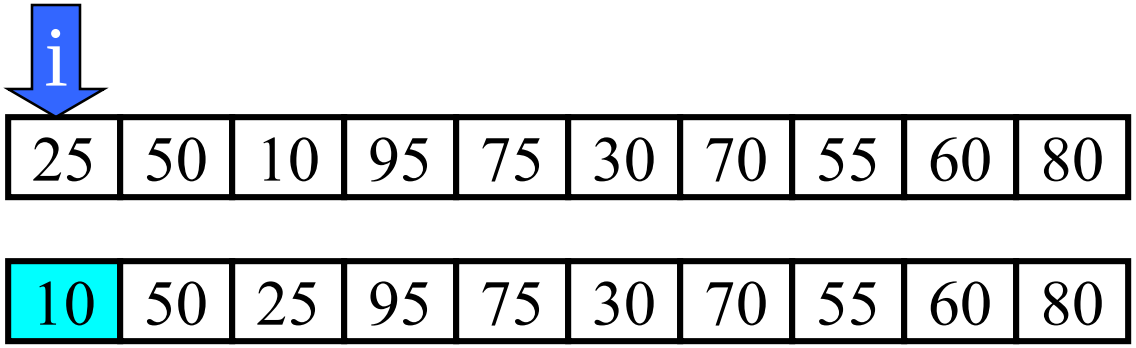
Adams	A
Smith	A
Black	B
Jackson	B
Washington	B
White	C
Wilson	C
Brown	D
Jones	D
Thompson	D

Indirect sorting

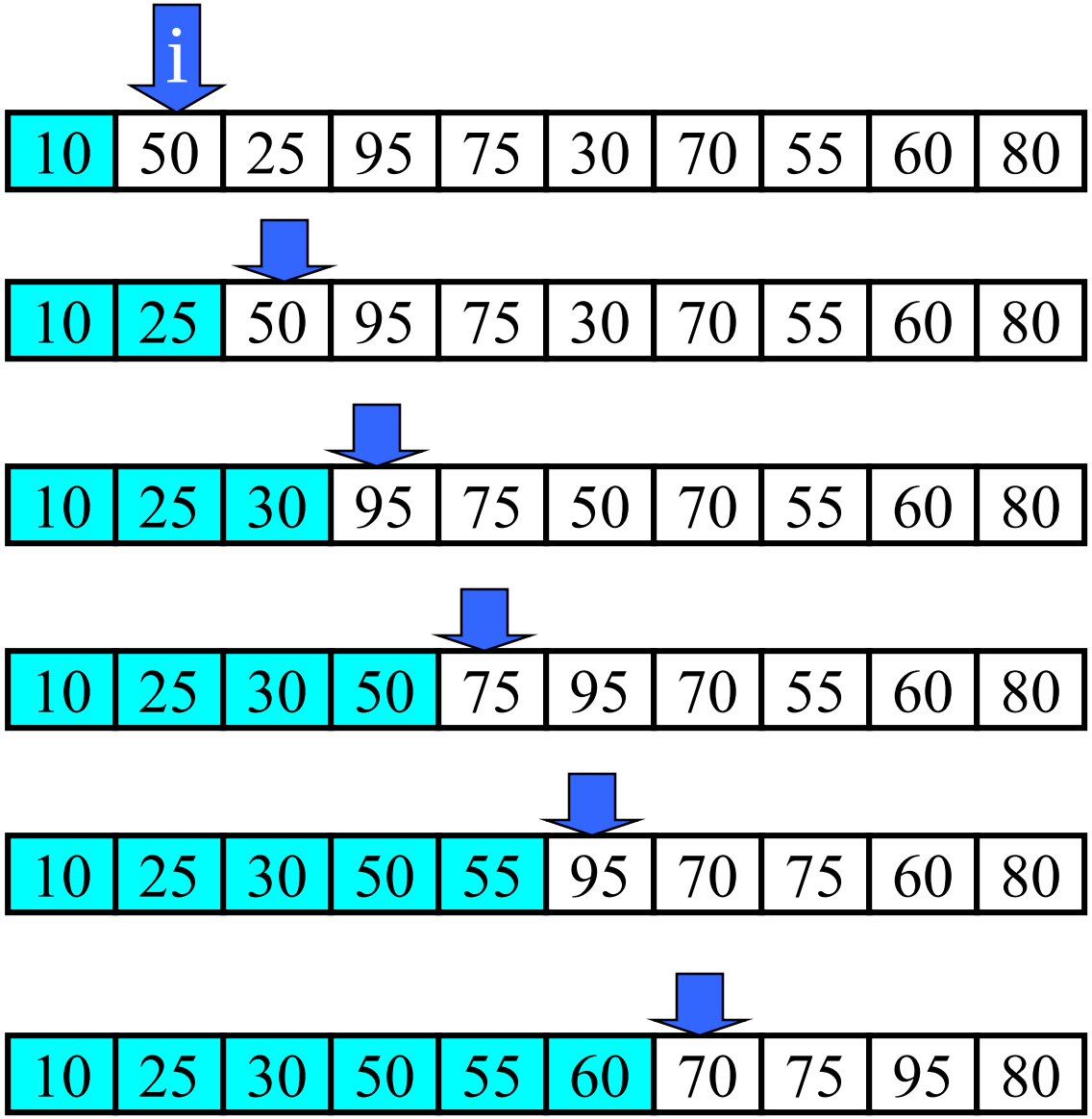
- Many sorting algorithms move and interchange records in memory several times during the process of sorting.
- For large records, or when the data set is large, this is inefficient.
- With "indirect sorting“, the indices (or pointers) of the records are sorted, rather than the records themselves.

Selection sort

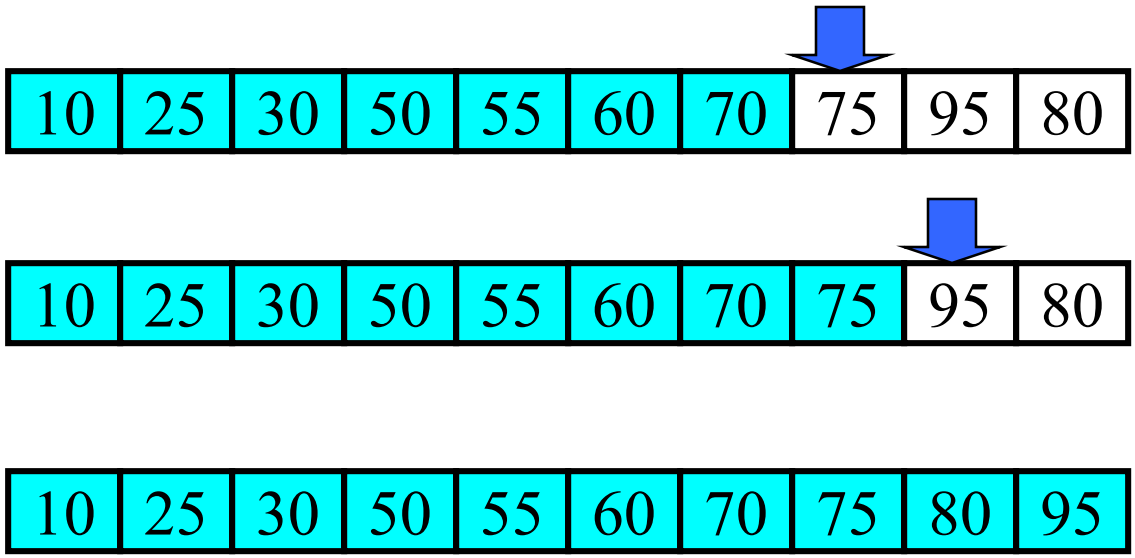
- Selection sort works as follows:
  - Find the smallest element in the array, and exchange it with the element in the first position.
  - Then, find second smallest element and exchange it with the element in the second position.
  - Continue in this way until the array is sorted.



Selection sort



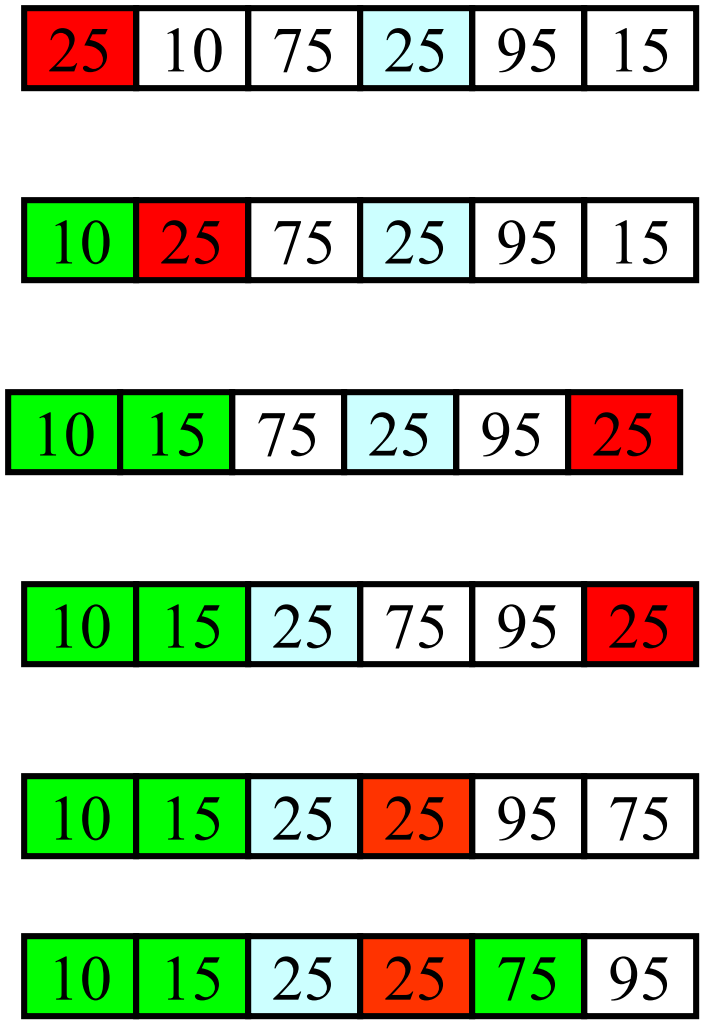
Selection sort



Selection sort

```
void SelectionSort (int A[], int N)
{ int i,j,min;
  for (i=0; i<N-1, i++)
  {
     /* find the the smallest among A[j]...A[n-1] */
     /* place it in A[i] */
     min = i;
     for (j=i+1; j<N; j++)
        if ( A[j] < A[min] )
            min = j;
     swap(A[i], A[min]);
  }
}
```

Selection sort- Is selection sort stable?



Selection sort - analysis

Iteration 1:

- Find smallest value in a list of **n** values: n-1 comparisons
- Exchange values and move marker

Iteration 2:

- Find smallest value in a list of **n-1** numbers: n-2 comparisons
- Exchange values and move marker

…

Iteration n-2:

- Find smallest value in a list of 2 numbers: 1 comparison
- Exchange values and move marker

Total: (n-1) + (n-2) + …. + 2 + 1 = n(n-1)/2

Selection sort - analysis

Space efficiency:

- No extra space used (except for a few variables)

Time efficiency:

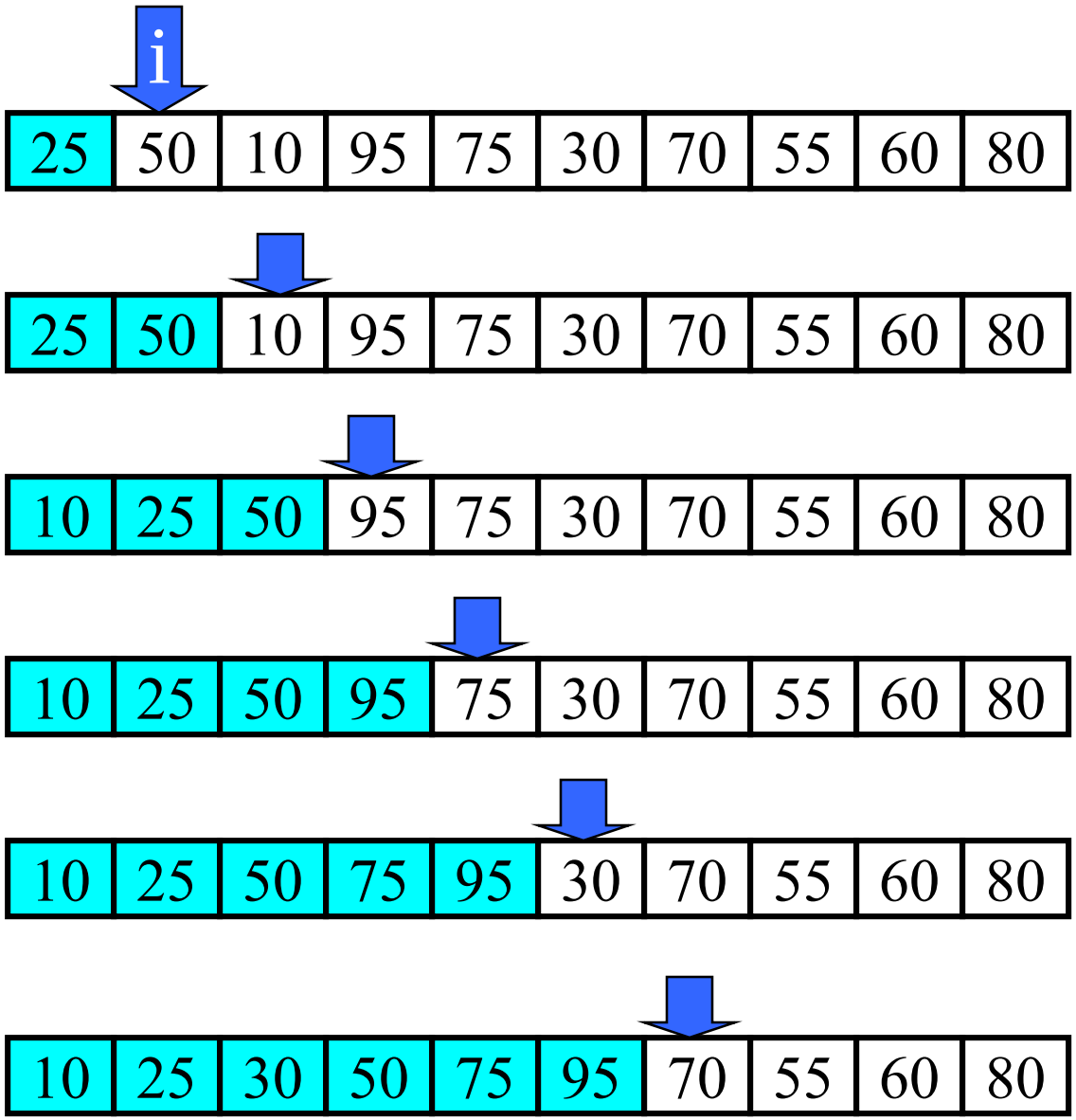
- The best-case and worst-case are same. All input sequences need same number of comparisons.
- the amount of work is the same:

$$T(n) = n(n-1)/2$$

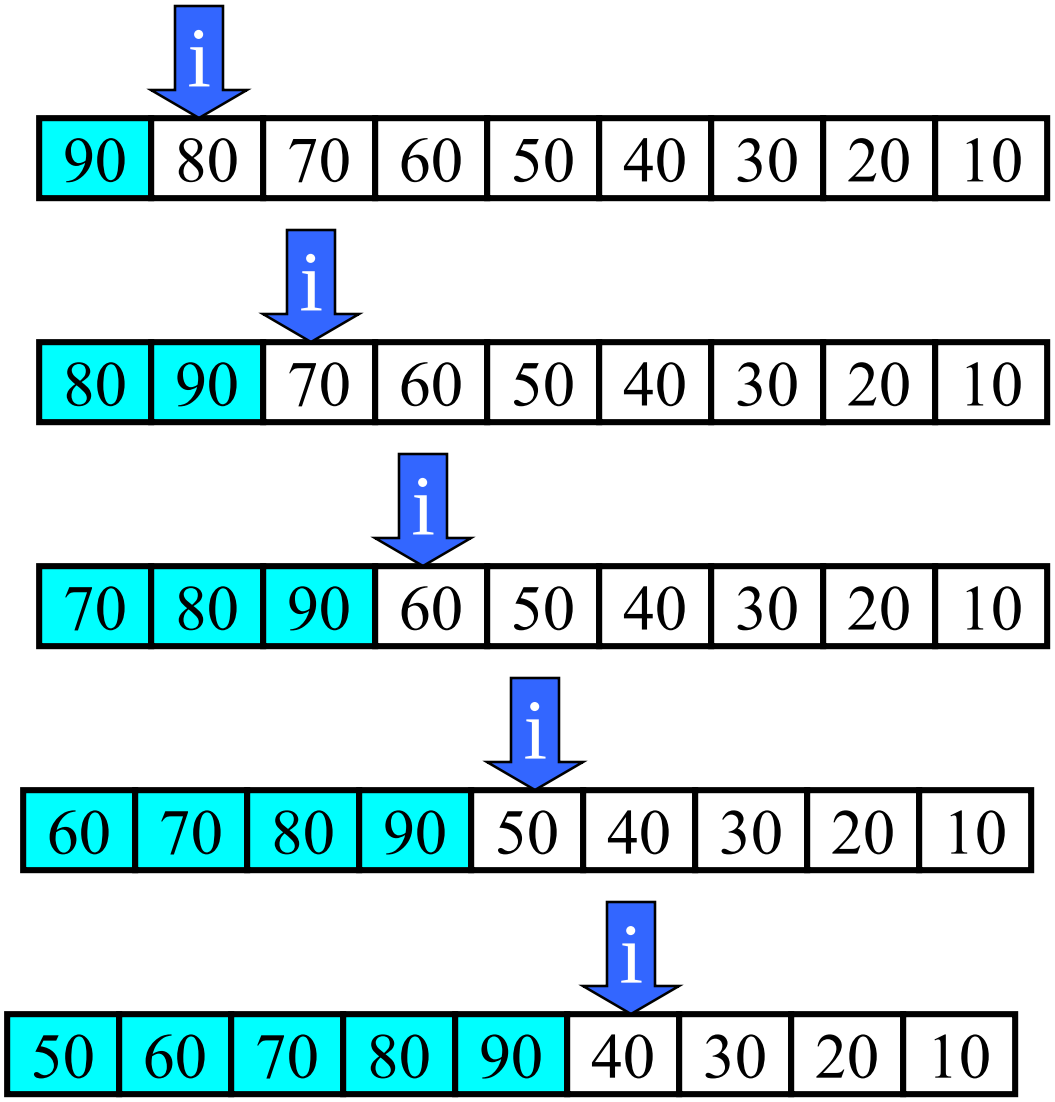
Insertion sort

- Insert the first element of the unsorted array into already sorted portion of the array by shifting all larger elements to the right.
- Initially the sorted portion consists of the first element.
- The sorted portion grows by one after every pass.

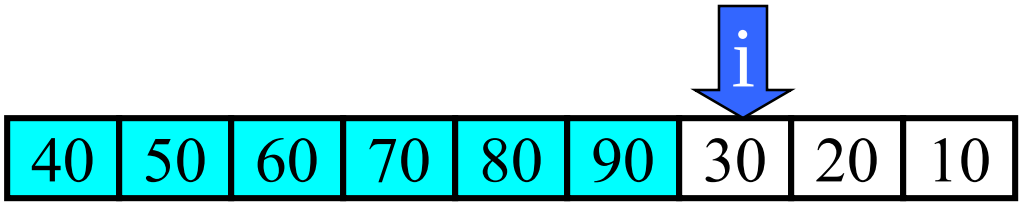
Insertion sort



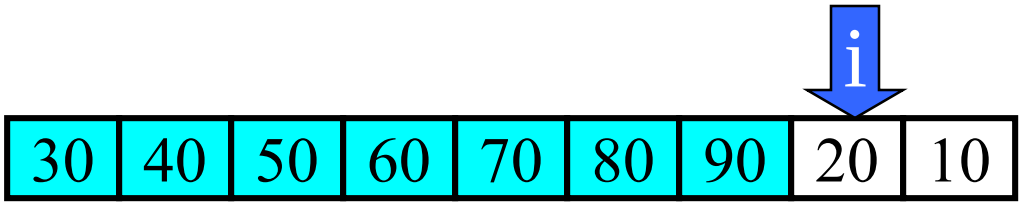
Insertion sort- Worst Case



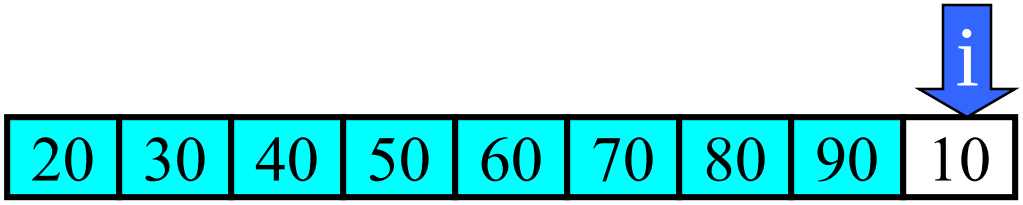
Insertion sort- Worst Case



# of Comparison = 6



# of Comparison = 7



# of Comparison = 8

Insertion sort- Worst Case

- For size n, total # of comparisons:

  $T(n)_{worst} = n-1 + n-2 + n-3+ \ldots + 2 + 1 = (n-1)n / 2$

  $T(n)_{avg} = N^2 / 4$

Insertion sort

```
void insertionSort(int A[], int N)
{  int i, j, next;

   for (i=1; i<N; i++)
   {
      next = A[i];

      for(j=i; j>0 && next < A[j-1]; j--)
         A[j] = A[j-1];
      A[j] = next;
   }
}
```

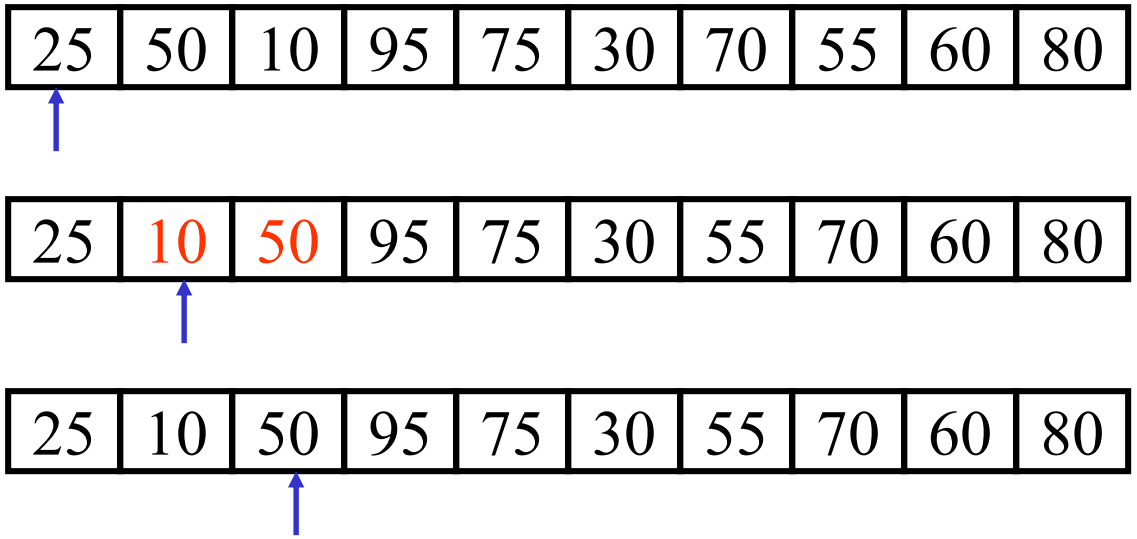
Comparing insertion sort / selection sort

Selection sort	Insertion sort
scan unsorted portion of array	scan sorted portion of array
the amount of work does not depend on the type of input	for already sorted arrays runs faster.
insert each element into its final position	after the insertion every element can be moved later.
minimal amount of element exchanges	a lot of shifts.

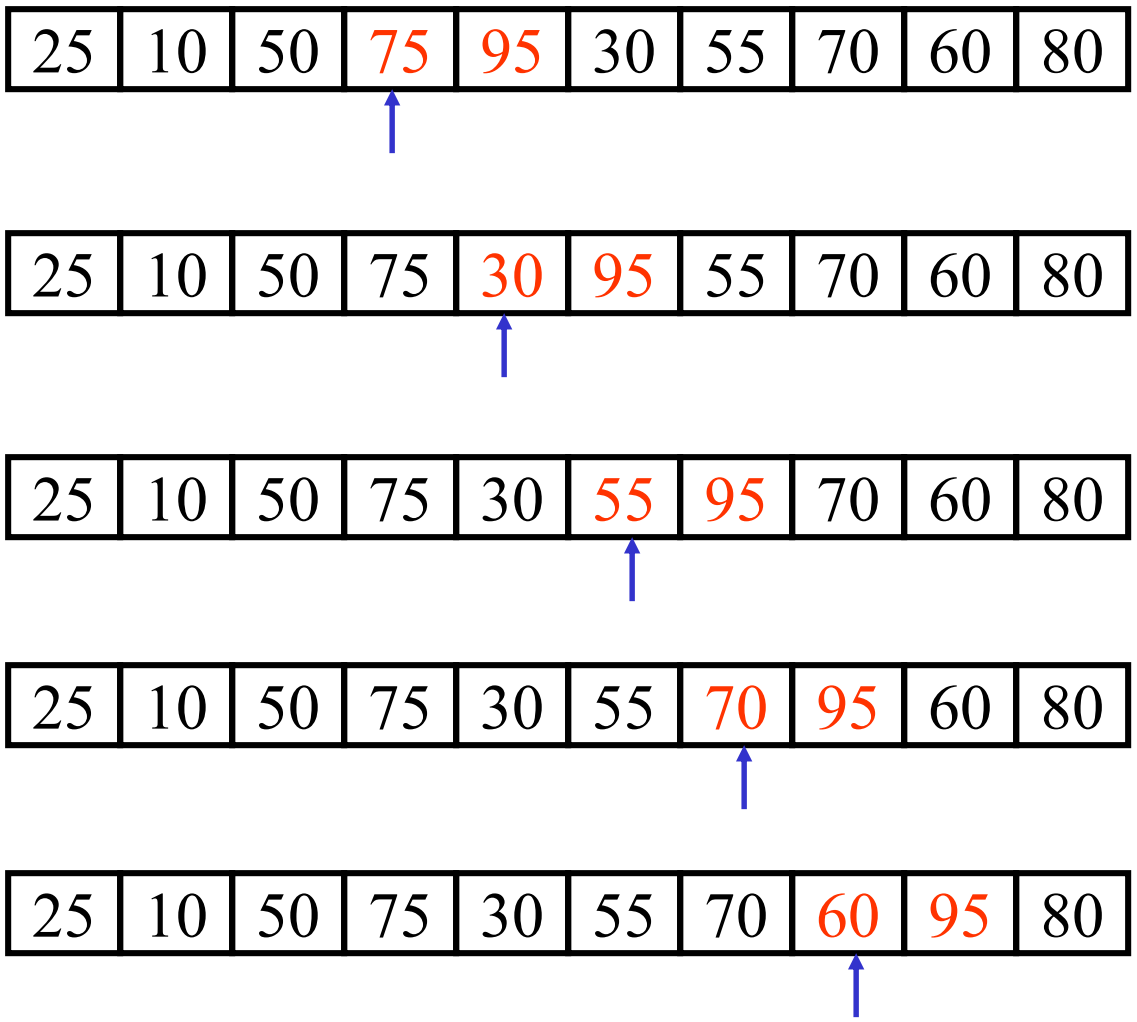
Bubble sort

- Bubble sort compares adjacent elements of the list and exchange them if they are out of order.
  - After the first pass, we put the largest element to the last position in the list.
  - The next pass puts the second largest element in its position(just before the last position).

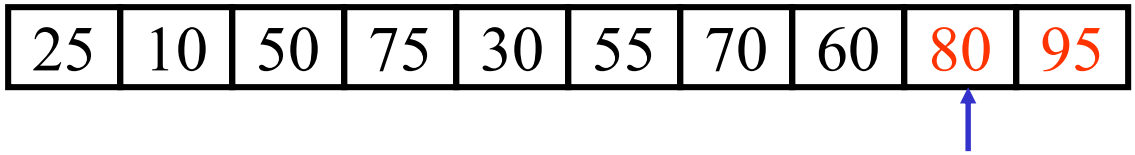
First pass



Bubble sort


25 10 50 75 95 30 55 70 60 80
25 10 50 75 30 95 55 70 60 80
25 10 50 75 30 55 95 70 60 80
25 10 50 75 30 55 70 95 60 80
25 10 50 75 30 55 70 60 95 80

Bubble sort




```
void Bubblesort(int A[], int N)
for (i=0; i<N-1; i++)
   for (j= 0; j<n-2-i; j++)
      if (A[j+1] < A[j])
         Swap(A[j], A[j+1]);
```

- For size n, total # of comparisons:

$$T(n) = n-1 + n-2 + n-3+ \ldots + 2 + 1 = (n-1)n / 2$$