

Searching and Sorting (Selection & Bubble)

- **Searching is the process of determining whether or not a given value exists in a data structure or a storage media.**
- **Two searching algorithms**
 - **Linear Search**
 - **Binary Search**

The linear (or sequential) search algorithm on an array is:

- **Start from beginning of an array/list and continues until the item is found or the entire array/list has been searched.**
 - **Sequentially scan the array, comparing each array item with the searched value.**
 - **Linear search algorithm has complexity of $O(n)$.**
 - **Linear search can be applied to both sorted and unsorted arrays.**

- The elements of the array need not be in sorted order.
- It can be applied on any linear data structures even if elements of data structures don't occupy the contiguous memory locations.
- Start from beginning and compare with each element and continue until element is found or reach the end of the array of elements
- The complexity of linear search is Big $O(n)$.
- Example : consider a array

11	42	13	24	15
----	----	----	----	----



Linear Search Algorithm



Searching the position of given element “Data” in an array ‘S’ having ‘n’ elements.

Input : n, S, Data

1. Set $i = 1$

**2. If $S[i] = \text{Data}$
then**

Print “ Element is found at position “: i

Exit

3. Set $i = i+1$

4. If $i \leq n$ go back to step 2

5. Print: “ Desired element Data is NOT found in the array”

6. Exit

Binary Search



- Binary search algorithm can be used when the input is in sorted order and it gives an efficient searching mechanism.
- The binary search starts by testing the data in the list at the middle of the array to determine if the target is in the first or second half of the list.
- If it is in the first half, no need to check the second half.
- If it is in the second half, no need to test the first half.
- In turn eliminate half the list from further consideration.
- Repeat this process until the target is found or until the list has elements to be divided.
- If there are no elements to be divided, conclude that the item does not exist in the list.
- To find the middle of the list, we need three variables,
 - one to identify the beginning of the list,
 - one to identify the middle of the list, and
 - one to identify the end of the list.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	7	8	10	14	21	22	36	62	77	81	91

- Search for 22 in the above sorted list
- The three indexes are **first**, **mid** and **last**.
- Given **first** as 0 and **last** as 11, **mid** is calculated as follows:

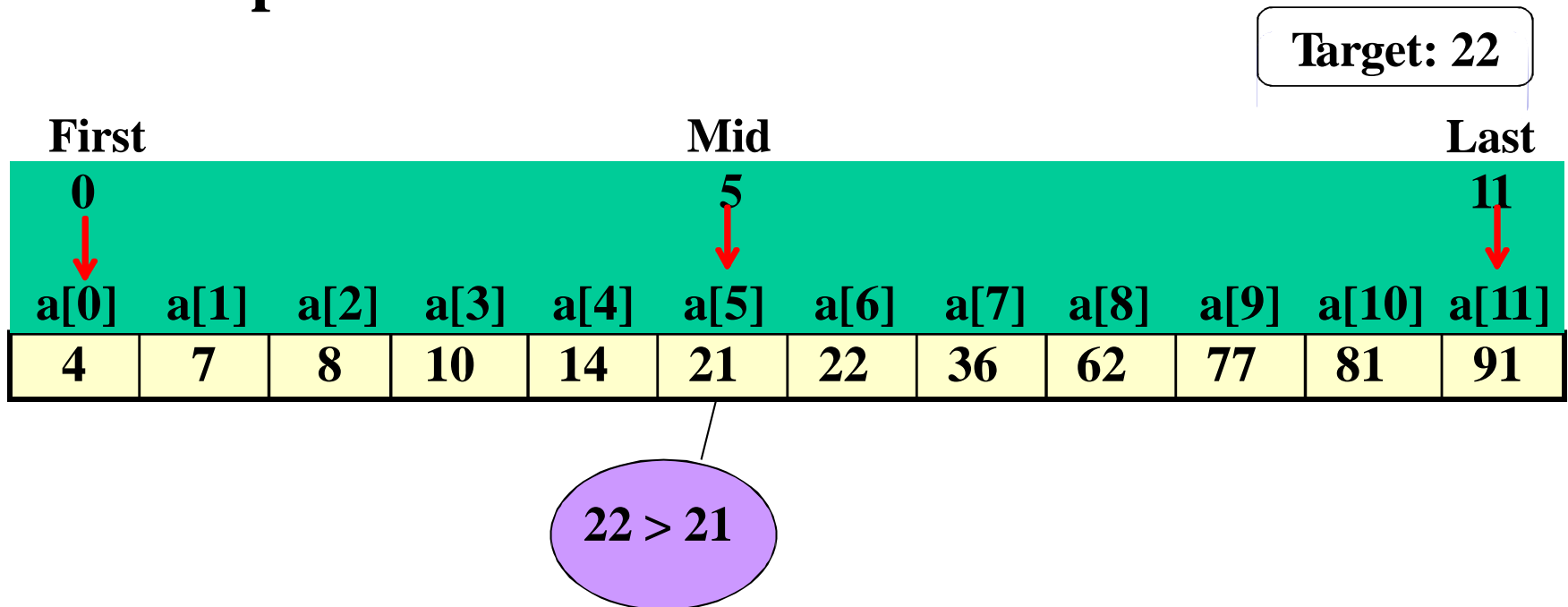
$$\text{mid} = (\text{first} + \text{last}) / 2$$

$$\text{mid} = (0 + 11) / 2 = 11 / 2 = 5$$

Binary Search



- At index location 5, the target is greater than the list value ($22 > 21$).
- Therefore, discard the array locations 0 through 5 (mid is automatically eliminated).
- To narrow down the search, assign $\text{mid} + 1$ to first and repeat the search.






Binary Search



- The next loop calculates mid with the new value for first and determines that the midpoint is now 8 as follows:
$$\text{mid} = (6 + 11) / 2 = 17 / 2 = 8$$
- Now the target is less than the value at mid ($22 < 62$).

Target: 22

						First	Mid		Last		
						6	8		11		
											
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	7	8	10	14	21	22	36	62	77	81	91

$22 < 62$

Binary Search



- This time adjust the end of the list by setting Last to mid – 1 and recalculate mid.
 - $\text{Last} = \text{Mid} - 1 = 8 - 1 = 7$
 - $\text{Mid} = (\text{First} + \text{Last}) / 2 = 6 + 7 / 2 = 6$
- This step effectively eliminates elements 8 through 11 from consideration.
- We have now arrived at index location 6, whose value matches our target. This stops the search.

Mid											
First						Last					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	7	8	10	14	21	22	36	62	77	81	91

22 = 22

Target is found
function terminates

BINARY(DATA, LB, UB, ITEM, LOC)

1. [Initialize segment variables.]

Set $BEG = LB$, $END = UB$ and $MID = \text{INT}(BEG + END)/2$.

2. Repeat Steps 3 and 4 while $BEG \leq END$ and $DATA[MID] \neq ITEM$

**3. If $ITEM < DATA[MID]$, then: Set $END = MID - 1$
else Set $BEG = MID + 1$.**

4. Set $MID = \text{INT}((BEG + END)/2)$. (End of step 2 loop.)

**5. If $DATA[MID] = ITEM$
then: Set $LOC = MID$.
Else: Set $LOC = \text{NULL}$.**

6.Exit.

Complexity Analysis of Binary Search



- The **best case** for binary search occurs when the element being searched for is exactly at the middle of the sorted array. In this case only comparison is required to find that desired element giving a best case runtime of **Big O(1)**.
- **Worst case** for the binary search occurs when the element is not found in the array. since binary search halves the sorted array in each step until there are no values that can be halved.
- The efficiency of binary search in this case can be expressed as logarithmic function. For calculating the worst complexity, the number of elements in the array as power of two (i.e. $n=2^x$).

After 1st comparison, number of elements remains= $n/2^1 = n/2$

After 2nd comparison, number of elements remains= $n/2^2$

= $n/2$ After 3rd comparison, number of elements

remains= $n/2^3 = n/2$

:

After xth comparison, number of elements remains= $n/2^x=1$

Total number of maximum comparisons = **$x = \log_2 n$**

What is sorting?



- **Sorting is the process of arranging data into meaningful order to analyze it more effectively.**
- **Order sales data by calendar month to produce a graph of sales performance.**
- **Sort text data into alphabetical order.**
- **Sort numeric data into numerical order.**
- **Group sort data to many levels.**
- **Sort on City within Month within Year**

- **Sorting is the process of rearranging data elements/Items in ascending or descending order**
- **Unsorted Data**

• **Sorte**

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

- **Sorted Data (Descending)**

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

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

- **Sorting can be numerical or lexicographical order.**
- **Seldom we sort isolated values**
- **Usually we sort a number of records containing a number of fields based on a key.**

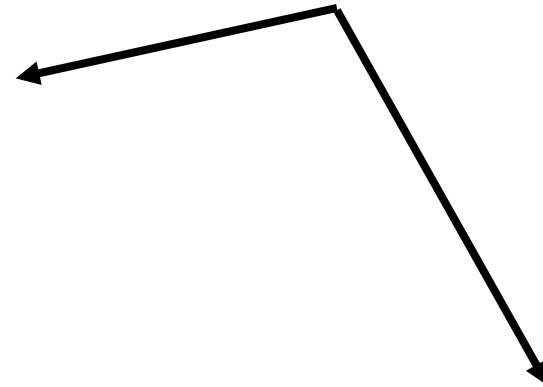
Sorting



19991532	Suresh	Mohan	VV Puram Bangalore
19990253	Robert	D'Souza	Ghatkoper Mumbai
19985832	Kalim	Pasha	M G Road Mangalore
20003541	Ganesh	Karker	Sardar Road Hubli
19981932	Carol	Mathew	Avenue Road Mulki
20003287	Robinson	David	Kalsanka Udupi

Numerically by ID Number

19981932	Carol	Mathew	Avenue Road Mulki
19985832	Kalim	Pasha	M G Road Mangalore
19990253	Robert	D'Souza	Ghatkoper Mumbai
19991532	Suresh	Mohan	VV Puram Bangalore
20003287	Robinson	David	Kalsanka Udupi
20003541	Ganesh	Karker	Sardar Road Hubli



Lexicographically by surname, then given name

19981932	Carol	Mathew	Avenue Road Mulki
20003541	Ganesh	Karker	Sardar Road Hubli
19985832	Kalim	Pasha	M G Road Mangalore
20003287	Robinson	David	Kalsanka Udupi
19990253	Robert	D'Souza	Ghatkoper Mumbai
19991532	Suresh	Mohan	VV Puram Bangalore



- **Bogo Sort**: shuffle and pray
- **Selection Sort**: look for the smallest element, move to front
- **Bubble Sort**: swap adjacent pairs that are out of order
- **Insertion Sort**: build an increasingly large sorted front portion
- **Merge Sort**: recursively divide the array in half and sort it
- **Heap Sort**: place the values into a sorted tree structure
- **Quick Sort**: recursively partition array based on a middle value
- other specialized sorting algorithms:
 - **Bucket Sort**: cluster elements into smaller groups, sort them
 - **Radix Sort**: sort integers by last digit, then 2nd to last, then ...
 - ...

Selection Sort



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



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison



Data Movement



Sorted

Selection Sort



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

↑
Largest



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison

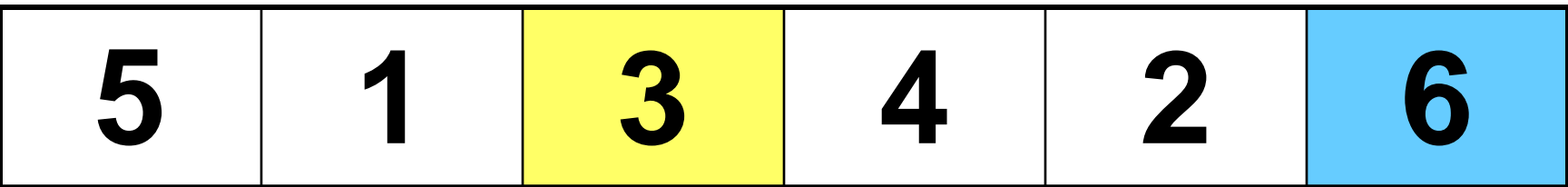


Data Movement



Sorted

Selection Sort



Comparison

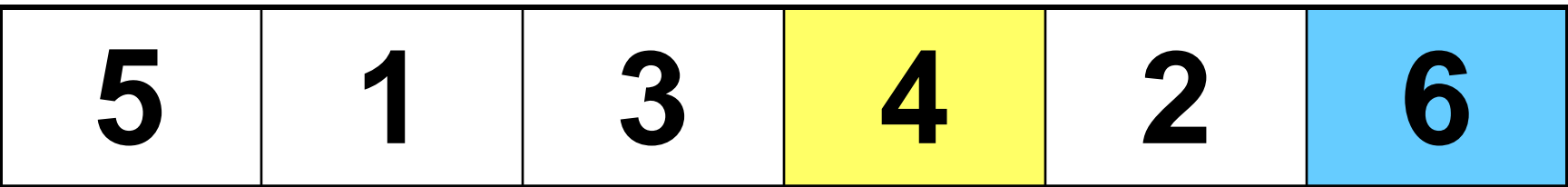


Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison

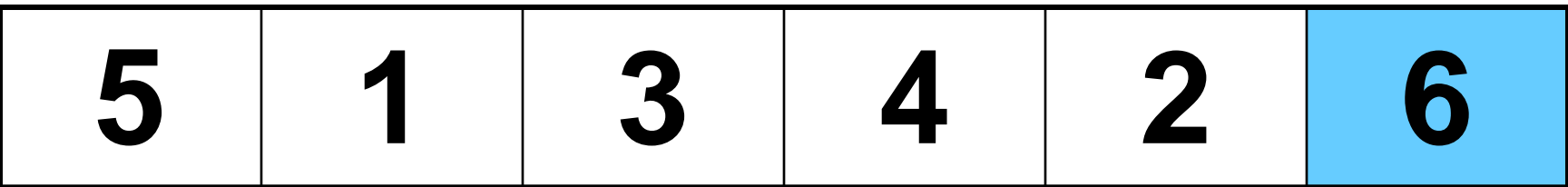


Data Movement



Sorted

Selection Sort



↑
Largest



Comparison

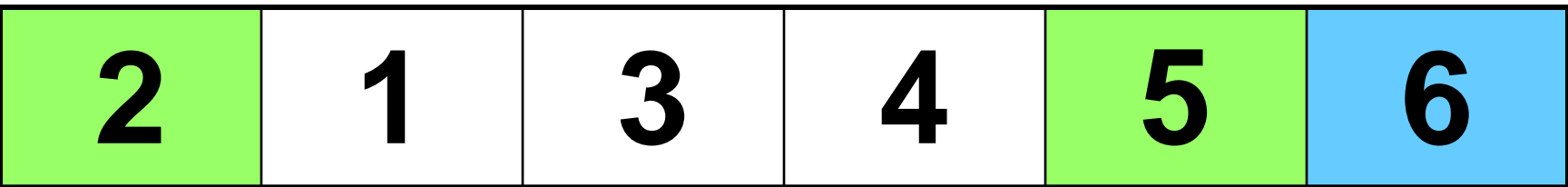


Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison

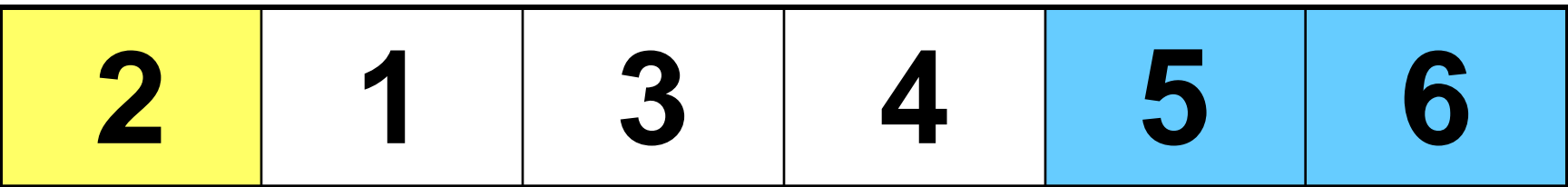


Data Movement



Sorted

Selection Sort



Comparison

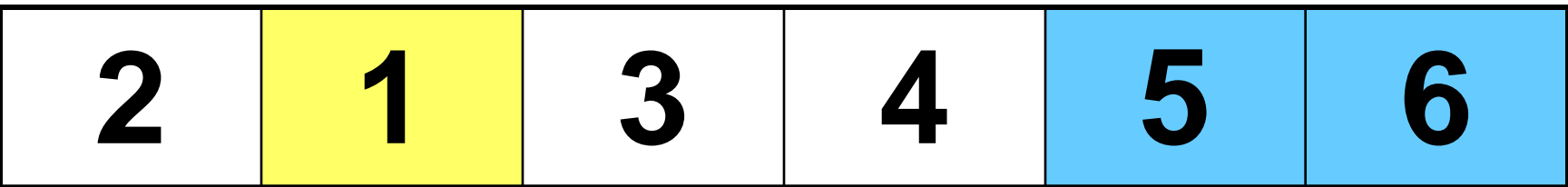


Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort



Comparison

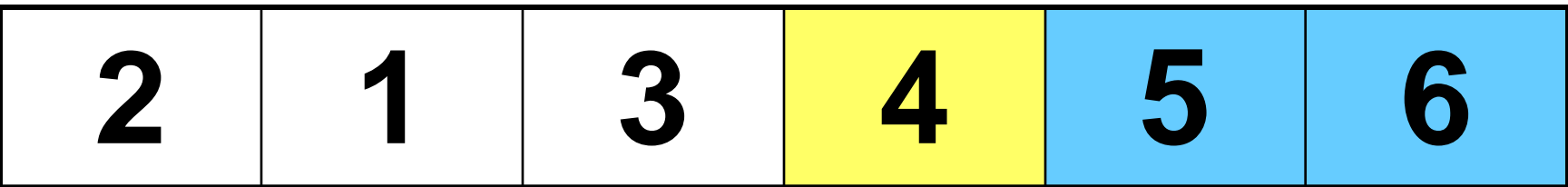


Data Movement



Sorted

Selection Sort



Comparison

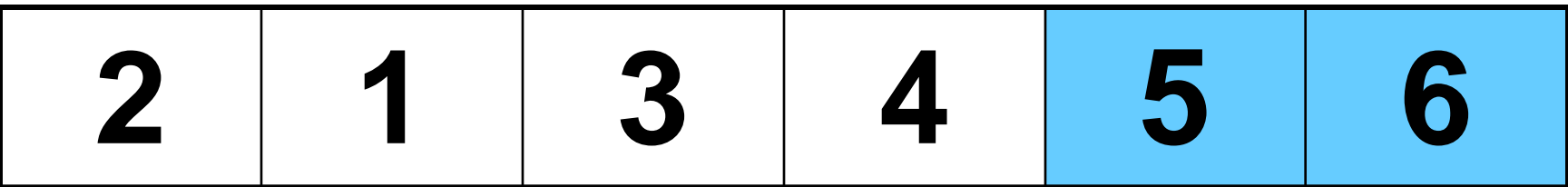


Data Movement



Sorted

Selection Sort



↑
Largest



Comparison

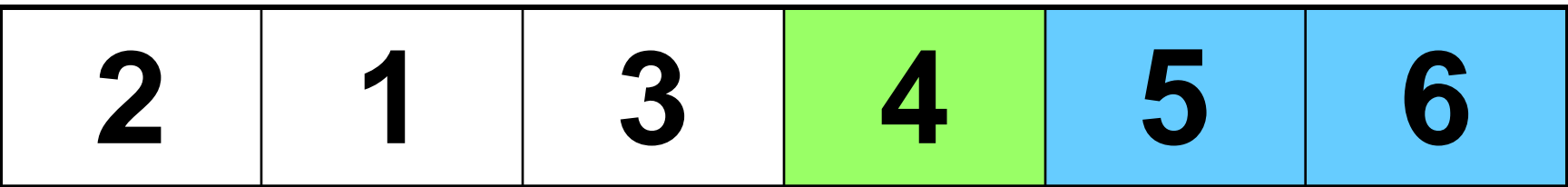


Data Movement



Sorted

Selection Sort



Comparison



Data Movement



Sorted

Selection Sort



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



Comparison

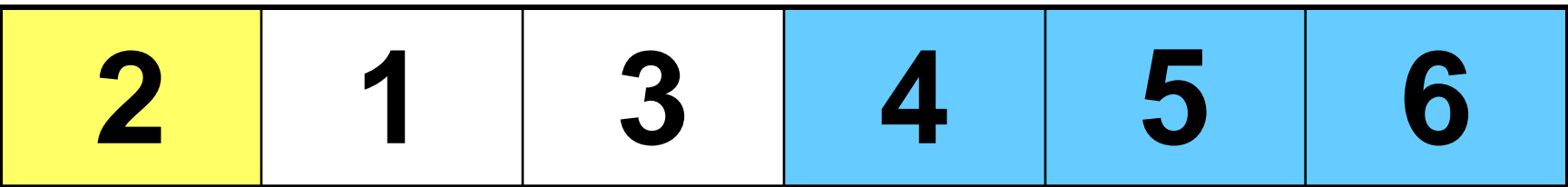


Data Movement



Sorted

Selection Sort



Comparison

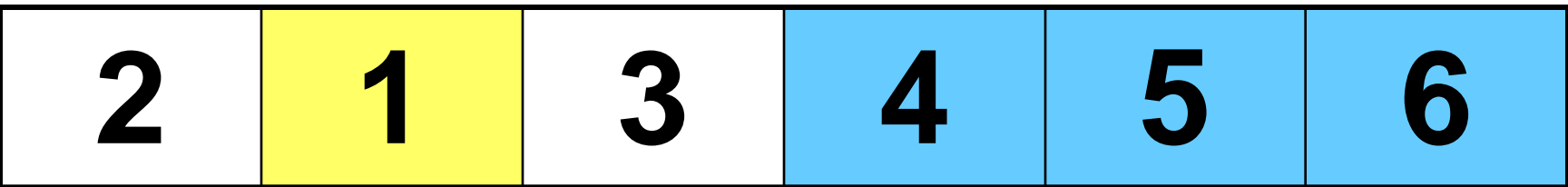


Data Movement



Sorted

Selection Sort



Comparison

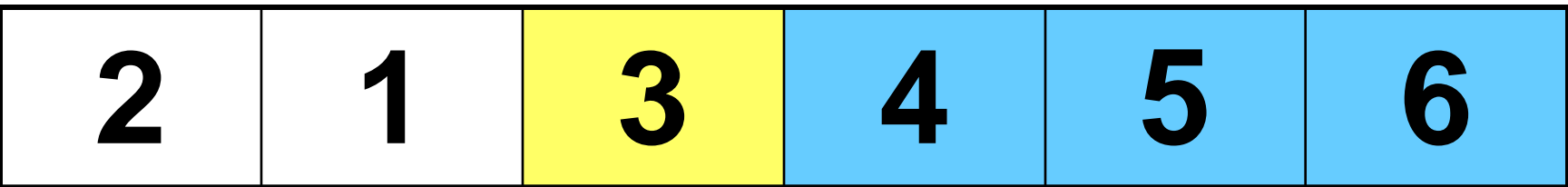


Data Movement



Sorted

Selection Sort



Comparison

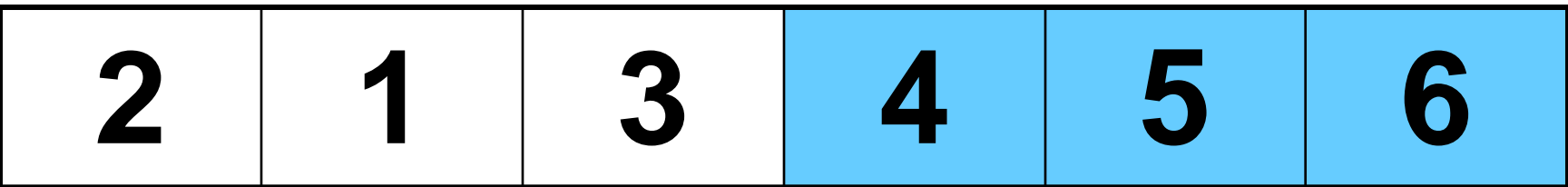


Data Movement



Sorted

Selection Sort



↑
Largest



Comparison

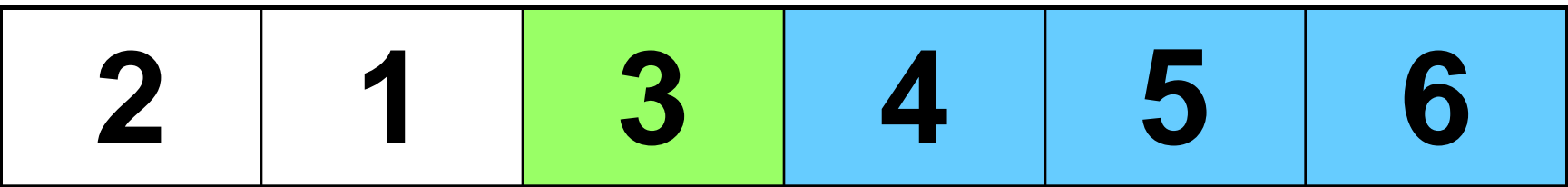


Data Movement



Sorted

Selection Sort



Comparison

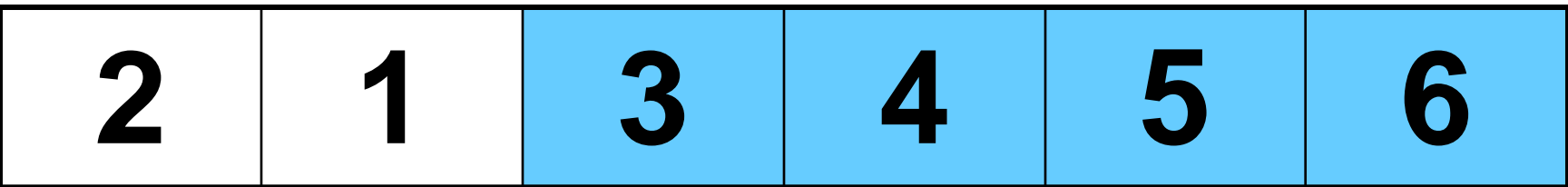


Data Movement



Sorted

Selection Sort



Comparison

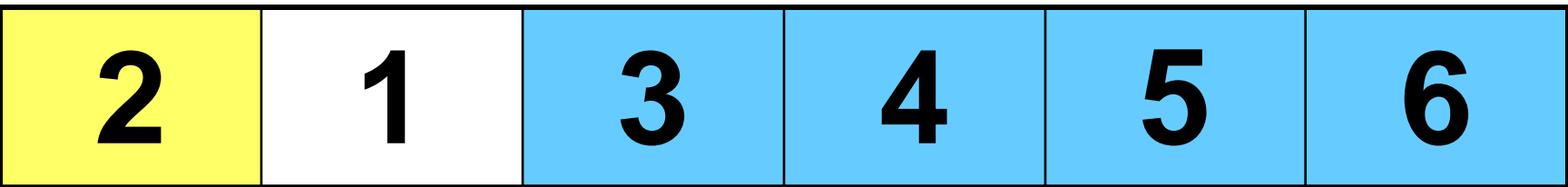


Data Movement



Sorted

Selection Sort



Comparison

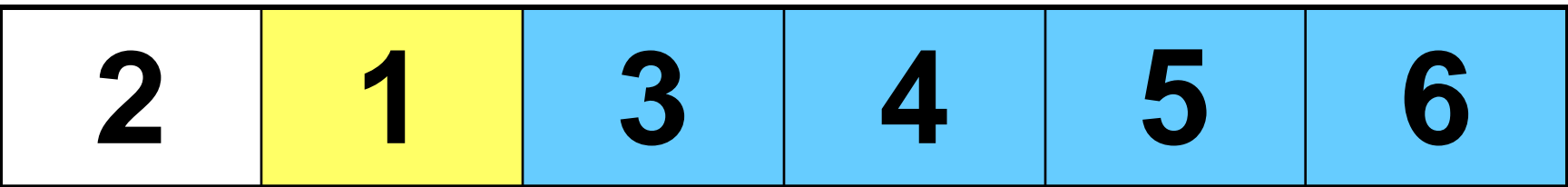


Data Movement



Sorted

Selection Sort



Comparison

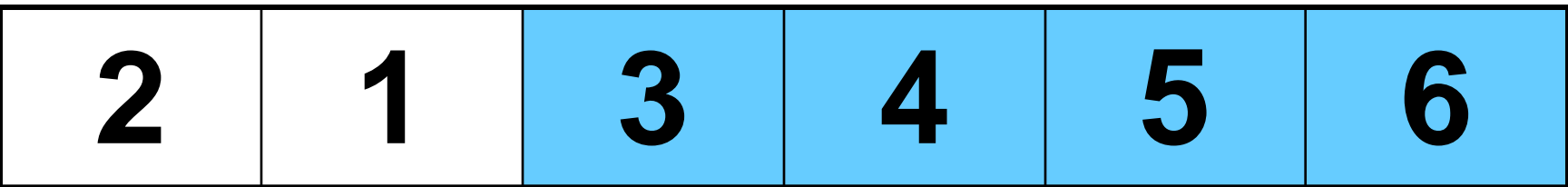


Data Movement



Sorted

Selection Sort



↑
Largest



Comparison

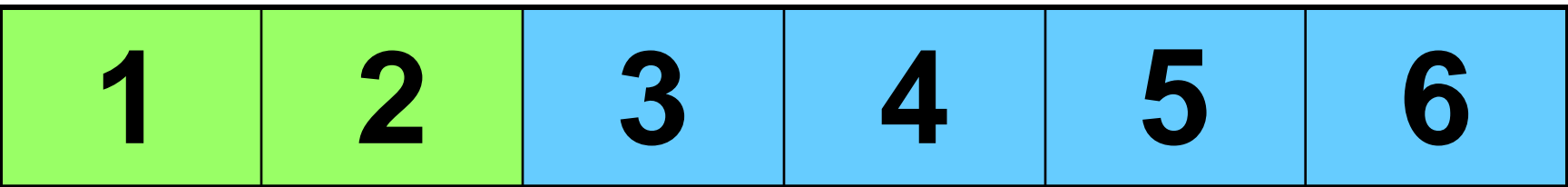


Data Movement



Sorted

Selection Sort



Comparison

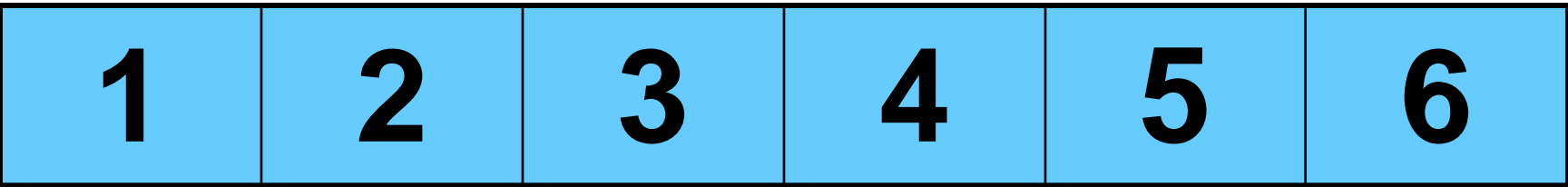


Data Movement



Sorted

Selection Sort



DONE!



Comparison



Data Movement



Sorted

Write the code for Selection Sort



Is it the right code for Selection Sort?



```
for ( i = 0 ; i < n-1 ; i++ )  
{  
  index = i;  
  for ( j = i+1 ; j < n ; j++ )  
  {  
    if(A[j] < A[index])
```

```
    index = j;
```

```
  }  
→
```

```
  temp = A[i];
```

```
  A[i] = A[index];
```

```
  A[index] = temp;
```

```
}
```

// i = variable to traverse the array A

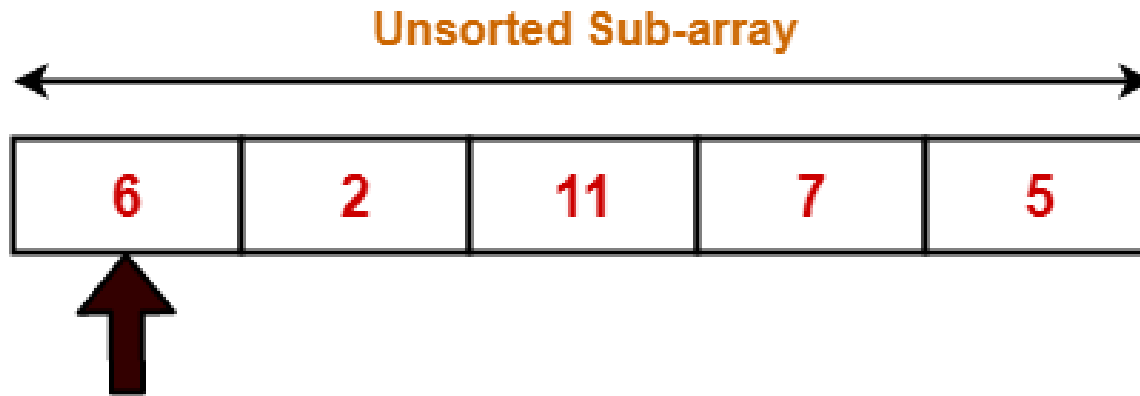
// index = variable to store the index of minimum element

// j = variable to traverse the unsorted sub-array

// temp = temporary variable used for swapping

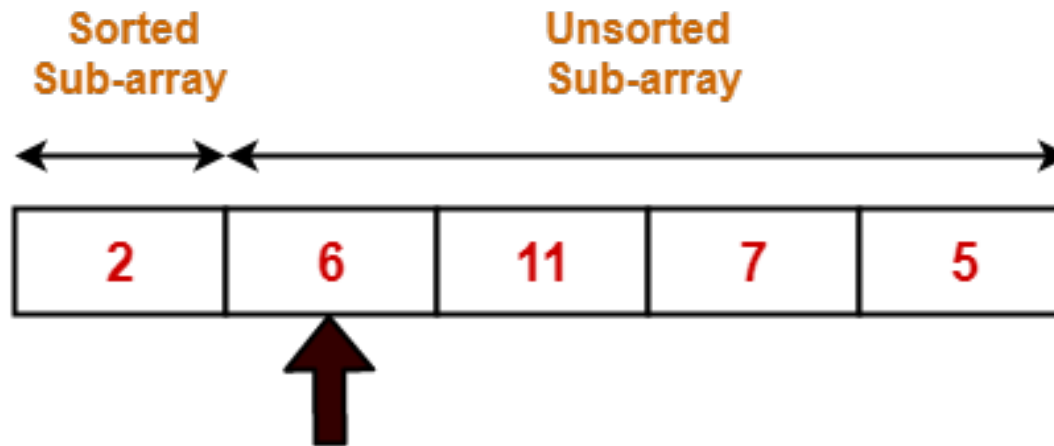
Ex. To sort **6 2 11 7 5** in ascending order

Step-01: for $i = 0$



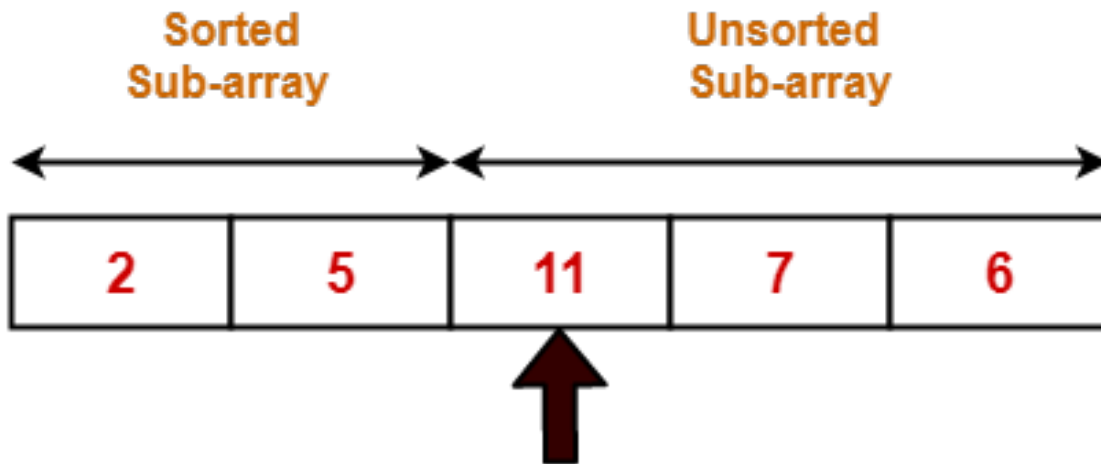
We start here, find the minimum element and swap it with the 1st element of array

Step-02: for $i = 1$



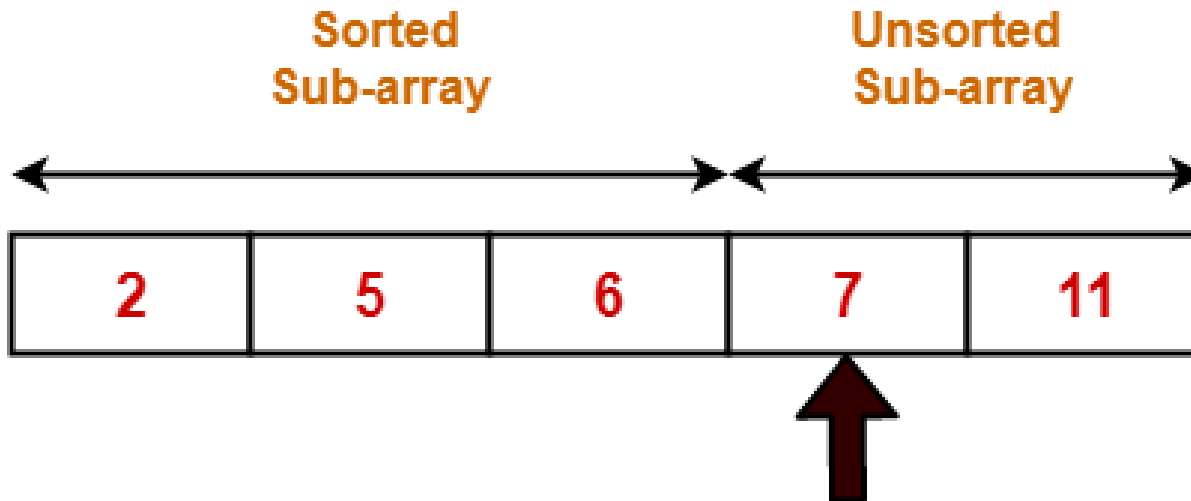
We start here, find the minimum element and swap it with the 2nd element of array

Step-03: for $i = 2$



We start here, find the minimum element and swap it with the 3rd element of array

Step-04: for $i = 3$



We start here, find the minimum element but there is no need to swap
(4th element is itself the minimum)

Step-05: **for $i = 4$**

**Sorted
Sub-array**





- With each loop cycle,
 - The minimum element in unsorted sub-array is selected.
 - It is then placed at the correct location in the sorted sub-array until array A is completely sorted.
- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has $O(n^2)$ time complexity

	Time Complexity
Best Case	n^2
Average Case	n^2
Worst Case	n^2



- Selection sort is an **in-place** algorithm.
- It performs all computation in the original array and no other array is used.
- Hence, the space complexity works out to be $O(1)$.

Note :

- Selection sort is **not a very efficient algorithm** when **data sets are large**.
- This is indicated by the average and worst case complexities.
- **Selection sort uses minimum number of swap operations $O(n)$ among all the sorting algorithms.**

Bubble sort



- It is an **in-place** sorting algorithm; It uses no auxiliary data structures (extra space) while sorting.
- Bubble sort uses multiple passes (scans) through an array.
- In each pass, bubble sort compares the adjacent elements of the array.
- It then swaps the two elements if they are in the wrong order.
- In each pass, bubble sort places the next largest element to its proper position.
- In short, it bubbles down the largest element to its correct position.

Bubble sort

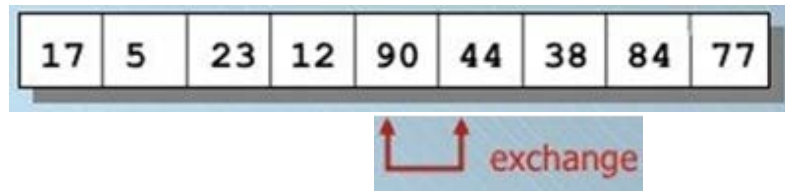
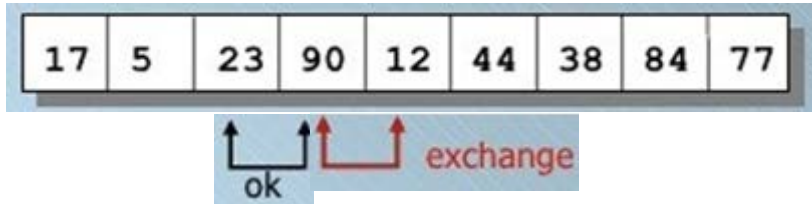
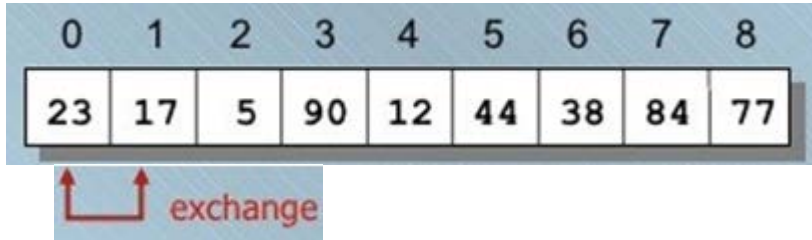


- With the selection sort, we make one exchange at the end of one pass.
- The bubble sort improves the performance by making more than one exchange during its pass.
- By making multiple exchanges, we will be able to move more elements toward their correct positions using the same number of comparisons as the selection sort makes.
- The key idea of the bubble sort is to make pairwise comparisons and exchange the positions of the pair if they are out of order.

Bubble Sort



One pass of Bubble Sort



The largest value 90 is at the end of the list.

Bubble sort




Example : We have an unsorted array list 'S' having 6 elements as shown below:

1	2	3	4	5	6
8	7	5	11	15	2

An Unsorted Array 'S' with 6 elements. Various Passes takes place to sort the list .

Pass 1 : Number of steps = $n-1 = 6-1 = 5$.

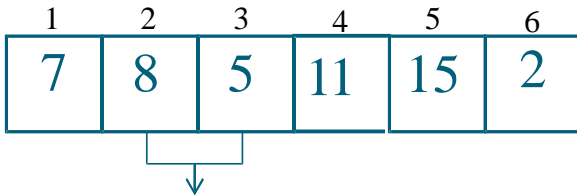
1	2	3	4	5	6
8	7	5	11	15	2



If($8 > 7$) then
interchange

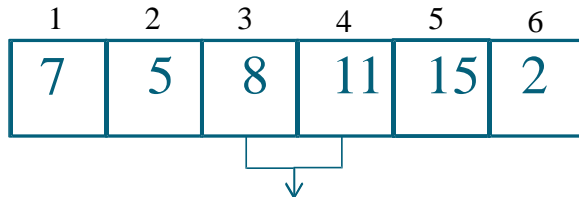
$S[1]$ is compared with $S[2]$; as $S[1] > S[2]$ then exchange takes place.

Bubble sort



If($8 > 5$) then
interchange

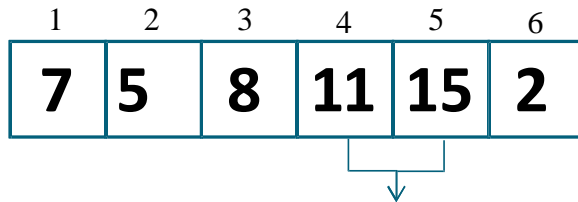
$S[2]$ is compared with $S[3]$; as $S[2] > S[3]$, then exchange



If($8 > 11$) then no interchange

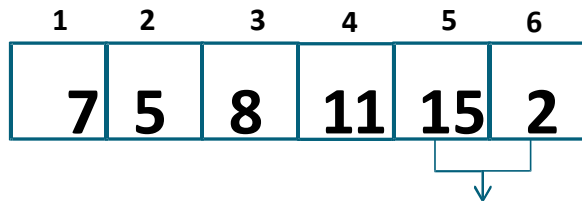
$S[3]$ is compared with $S[4]$; as $S[3] < S[4]$, then no
exchange takes place.

Bubble Sort



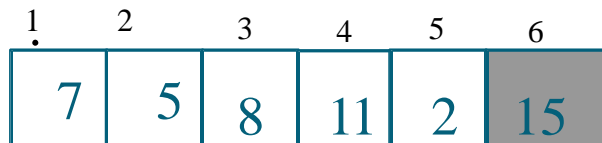
If(11>15) then no interchange

S[4] is compared with S[5]; as $S[4] < S[5]$, then no exchange takes place.



If(15>2) then interchange

S[5] is compared with S[6]; as $S[5] > S[6]$, then exchange



The 1st largest element 15
position S[6] in 5 comparisons

obtained its proper

Bubble Sort



Pass 2: $n-2 = 6-2 = 4$

1	2	3	4	5	6
7	5	8	11	2	15

If($7 > 5$) then
interchange

$S[1]$ is compared with $S[2]$; as $S[1] > S[2]$, then exchange takes place.

1	2	3	4	5	6
5	7	8	11	2	15

If($7 > 8$) then no
interchange

$S[2]$ is compared with $S[3]$; as $S[2] < S[3]$, then no exchange takes place.

Bubble Sort



1	2	3	4	5	6
5	7	8	11	2	15

↓
If(8>11) then no
interchange

$S[3]$ is compared with $S[4]$; as $S[3] < S[4]$, then no exchange

1	2	3	4	5	6
5	7	8	11	2	15

↓
If(11>2) then
interchange

$S[4]$ is compared with $S[5]$; as $S[4] > S[5]$, then exchange

Bubble Sort



1	2	3	4	5	6
5	7	8	2	11	15

The 2nd largest element 11 has obtained its proper position S[5] in 4 comparisons

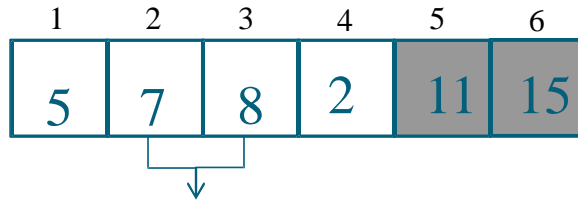
Pass 3: $n-3 = 6-3 = 3$

1	2	3	4	5	6
5	7	8	2	11	15

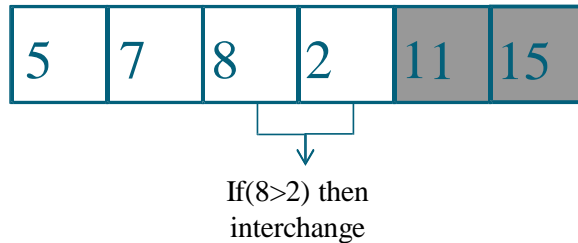
↓
If(5<7) then no
interchange

S[1] is compared with S[2]; as $S[1] < S[2]$, then no exchange takes place.

Bubble Sort

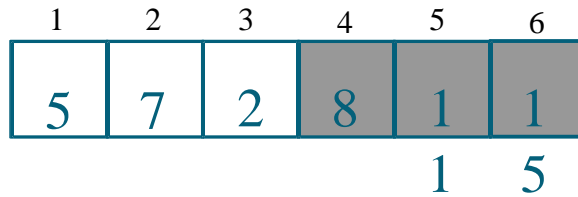


$S[2]$ is compared with $S[3]$; as $S[2] < S[3]$, then no exchange



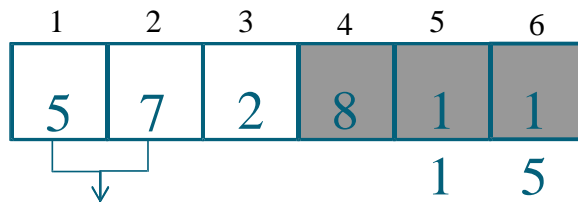
$S[3]$ is compared with $S[4]$; as $S[3] > S[4]$, then exchange takes place.

Bubble Sort



The 3rd largest element 8 has obtained its proper position $S[4]$ in 3 comparisons.

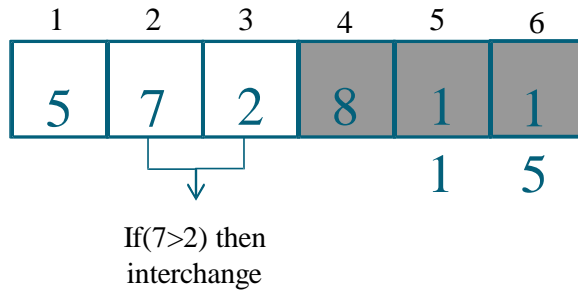
Pass 4: $n-4 = 6-4 = 2$



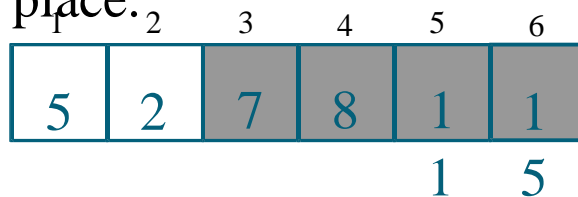
If $(5 < 7)$ then no
interchange

$S[1]$ is compared with $S[2]$; as $S[1] < S[2]$, then no exchange takes place.

Bubble Sort



$S[2]$ is compared with $S[3]$; as $S[2] > S[3]$, then exchange takes place.



The 4th largest element 7 has obtained its proper position $S[3]$ in 2 comparisons.

Bubble Sort



Pass 5: $n-5 = 6-5 = 1$

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

If($5 > 2$) then
interchange

$S[1]$ is compared with $S[2]$; as $S[1] > S[2]$, then exchange takes place.

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

The $S[1]$ and $S[2]$ have obtained its proper position in just a single comparison.

**Write the code for
Bubble Sort**

Is it the right code for Bubble Sort?



```
for( int pass=1 ; pass<=n-1 ; ++pass)    // passes through array
{ for( int i=0 ; i<=n-2 ; ++i)
    { if(A[i] > A[i+1])    // If adjacent elements are in wrong order
        { int temp = A[i];    // Swap them
          A[i] = A[i+1];
          A[i+1] = temp;
        }
    }
}
```

// C program for Bubble Sort (Correct ???)

#include <stdio.h>

main()

{ int i,n=5,j,key, A[12] = { 907,556,43,-5,6,404,55,3,22,-122,4,55};

for(i=0;i<n;i++)

printf(" %d ",A[i]);

for(int pass=1 ; pass<=n-1 ; ++pass) // passes through array

{ for(i=0 ; i<=n-2 ; ++i)

{ printf("\n Pass No: %d, i: %d, Compare A[%d] & A[%d] i.e.%d & %d",pass,i,i+1,A[i],A[i+1]);

if(A[i] > A[i+1]) // If adjacent elements are in wrong order

{ printf(" Swapping %d & %d",A[i],A[i+1]);

int temp = A[i]; // Swap them

A[i] = A[i+1];

A[i+1] = temp;

}

}

}

```
printf("\n");  
for( i=0;i<n;i++)  
    printf(" %d ",A[i]);  
}
```

907 556 43 -5 6

Pass No: 1, i: 0, Compare A[0] & A[1] i.e. 907 & 556 Swapping 907 & 556

Pass No: 1, i: 1, Compare A[1] & A[2] i.e. 907 & 43 Swapping 907 & 43

Pass No: 1, i: 2, Compare A[2] & A[3] i.e. 907 & -5 Swapping 907 & -5

Pass No: 1, i: 3, Compare A[3] & A[4] i.e. 907 & 6 Swapping 907 & 6

Pass No: 2, i: 0, Compare A[0] & A[1] i.e. 556 & 43 Swapping 556 & 43

Pass No: 2, i: 1, Compare A[1] & A[2] i.e. 556 & -5 Swapping 556 & -5

Pass No: 2, i: 2, Compare A[2] & A[3] i.e. 556 & 6 Swapping 556 & 6

Pass No: 2, i: 3, Compare A[3] & A[4] i.e. 556 & 907

Pass No: 3, i: 0, Compare A[0] & A[1] i.e. 43 & -5 Swapping 43 & -5

Pass No: 3, i: 1, Compare A[1] & A[2] i.e. 43 & 6 Swapping 43 & 6

Pass No: 3, i: 2, Compare A[2] & A[3] i.e. 43 & 556

Pass No: 3, i: 3, Compare A[3] & A[4] i.e. 556 & 907

Pass No: 4, i: 0, Compare A[0] & A[1] i.e. -5 & 6

Pass No: 4, i: 1, Compare A[1] & A[2] i.e. 6 & 43

Pass No: 4, i: 2, Compare A[2] & A[3] i.e. 43 & 556

Pass No: 4, i: 3, Compare A[3] & A[4] i.e. 556 & 907

-5 6 43 556 907

Is it the right code for Bubble Sort?



```
for( int pass=1 ; pass<=n-1 ; ++pass)    // passes through array
{ for( int i=0 ; i<=n-pass-1 ; ++i)
    { if(A[i] > A[i+1])    // If adjacent elements are in wrong order
        { int temp = A[i];    // Swap them
          A[i] = A[i+1];
          A[i+1] = temp;
        }
    }
}
```

// C program for Bubble Sort

#include <stdio.h>

main()

{ int i,n=5,j,key, A[12] = { 907,556,43,-5,6,404,55,3,22,-122,4,55};

for(i=0;i<n;i++)

printf(" %d ",A[i]);

for(int pass=1 ; pass<=n-1 ; ++pass) // passes through array

{ for(i=0 ; i<=n-pass-1 ; ++i)

{ printf("\n pass %d, i=%d, compare %d & %d",pass,i,A[i] , A[i+1]);

if(A[i] > A[i+1]) // If adjacent elements are in wrong order

{ int temp = A[i]; // Swap them

A[i] = A[i+1];

A[i+1] = temp;

}

}

}

```
printf("\n");  
for( i=0;i<n;i++)  
    printf(" %d ",A[i]);  
}
```

907 556 43 -5 6

pass 1, i =0, comapre 907 & 556

pass 1, i =1, comapre 907 & 43

pass 1, i =2, comapre 907 & -5

pass 1, i =3, comapre 907 & 6

pass 2, i =0, comapre 556 & 43

pass 2, i =1, comapre 556 & -5

pass 2, i =2, comapre 556 & 6

pass 3, i =0, comapre 43 & -5

pass 3, i =1, comapre 43 & 6

pass 4, i =0, comapre -5 & 6

-5 6 43 556 907



- The complexity of any sorting algorithm is analysed through the number of Comparisons required during the sorting procedure.
- In this algorithm an Array of size n gets sorted after $n-1$ passes.
- $n-1$ comparisons take Place during the 1st pass which places the largest element of the array on the last Position
- $n-2$ comparisons take place in the 2nd pass which places the second largest Element of the array on the second last position



- k^{th} pass requires $n-k$ comparisons which places k^{th} largest element at $(n-k+1)^{\text{th}}$ position of the array and
- the last pass requires only one comparison.

- Total number of comparison will be:

$$\begin{aligned} F(n) &= (n-1) + (n-2) + (n-3) + \dots + (n-k) + 2 + 1 \\ &= (n-1) + (n-2) + \dots + 2 + 1 \\ &= ((n-1) * n) / 2 \end{aligned}$$

- The complexity of bubble sort algorithm will be **Big $O(n^2)$**