

Introduction to Problem Solving and Programming

Lecture 6 – Sorting and Searching



Sorting

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

1	2	3	4	5	6
77	42	35	12	101	5



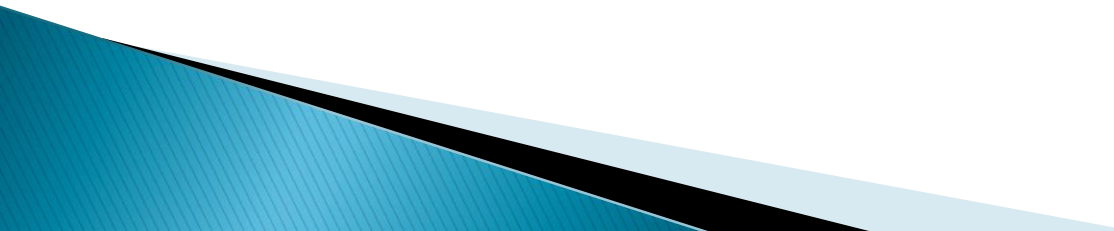
1	2	3	4	5	6
5	12	35	42	77	101

Sorting

There are many algorithms for sorting a list of items

- Bubble sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort
- Heap Sort
- Shell Sort
- Radix Sort
- Bucket Sort
- Sort



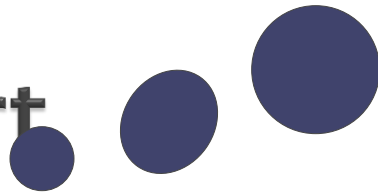


Bubble Sort

- During first pass, compare first two items in the array, exchange them if they are out of order.
- Compare the second item with third one, and swap them if they are out of order.
- You proceed, comparing and exchanging items until you reach the end of the array.
- During the second pass, you perform the same actions of comparing and exchanging every two adjacent items, until you finish comparing the third last item with the second last.

Repeat the same process for $n-1$ passes

Bubble Sort



for pass = 1 .. n-1

 for position = 1 .. n-pass

 if element at position > element at position + 1

 Swap elements

 end if

 next position

next pass

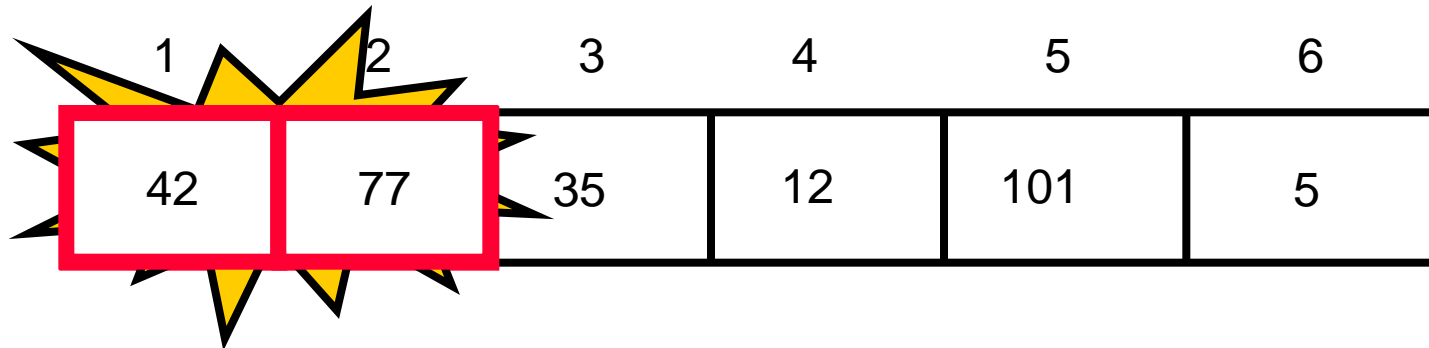
"Bubbling" the Largest Element

- ▶ Traverse a collection of elements
 - Move from the front to the end
 - "Bubble" the **largest value** to the end using **pair-wise comparisons and swapping**

1	2	3	4	5	6
77	42	35	12	101	5

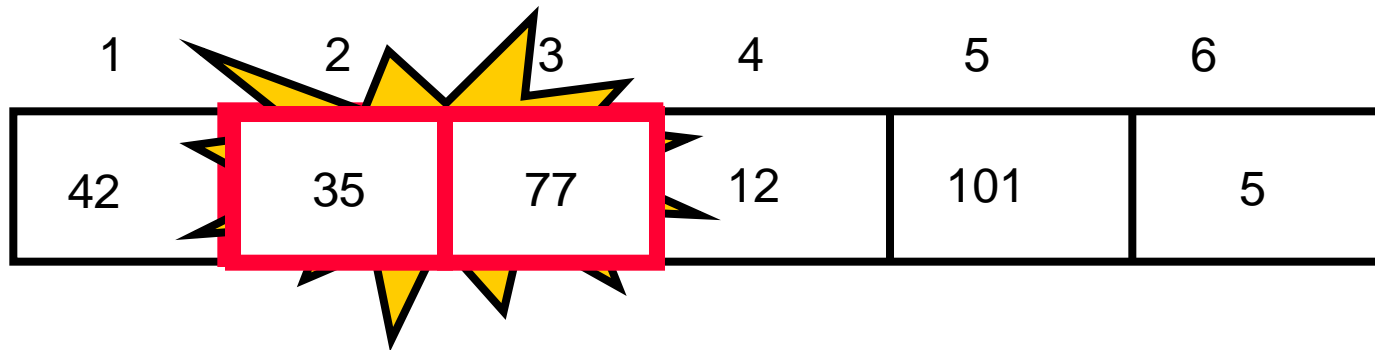
"Bubbling" the Largest Element

- ▶ Traverse a collection of elements
 - Move from the front to the end
 - "Bubble" the largest value to the end using pairwise comparisons and swapping



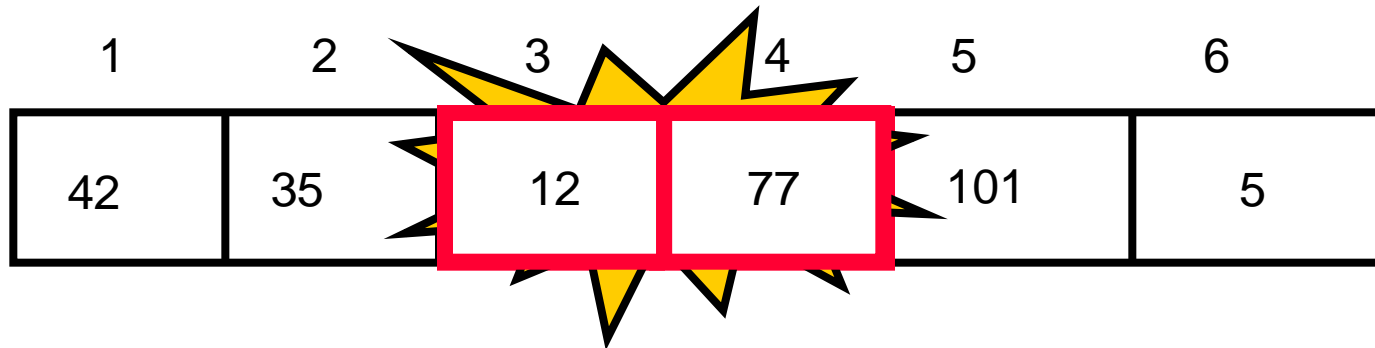
"Bubbling" the Largest Element

- ▶ Traverse a collection of elements
 - Move from the front to the end
 - "Bubble" the largest value to the end using pairwise comparisons and swapping



"Bubbling" the Largest Element

- ▶ Traverse a collection of elements
 - Move from the front to the end
 - "Bubble" the largest value to the end using pairwise comparisons and swapping



"Bubbling" the Largest Element

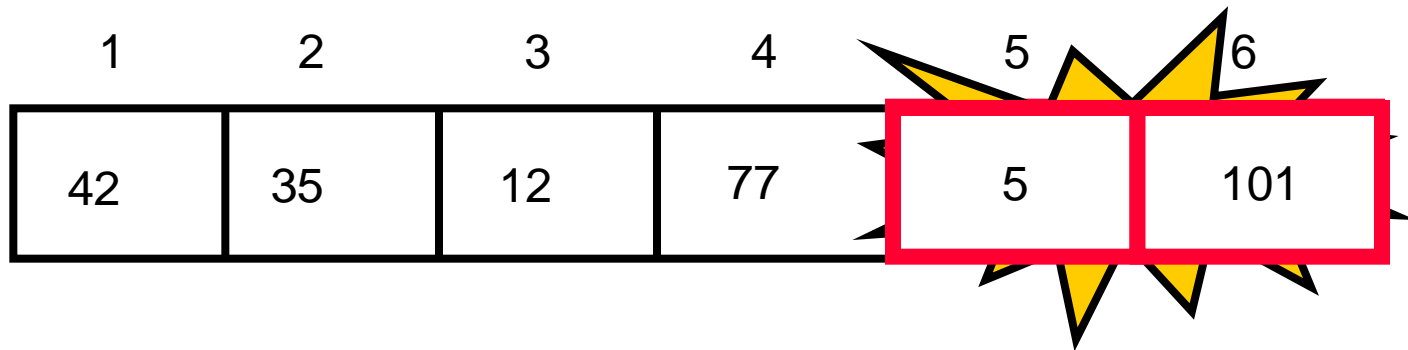
- ▶ Traverse a collection of elements
 - Move from the front to the end
 - "Bubble" the largest value to the end using pairwise comparisons and swapping

1	2	3	4	5	6
42	35	12	77	101	5

No need to swap

"Bubbling" the Largest Element

- ▶ Traverse a collection of elements
 - Move from the front to the end
 - "Bubble" the largest value to the end using pairwise comparisons and swapping



"Bubbling" the Largest Element

- ▶ Traverse a collection of elements
 - Move from the front to the end
 - "Bubble" the largest value to the end using pairwise comparisons and swapping

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

Bubble Sort Example: 1 5 9 6 2 3

FIRST PASS

1	5	9	6	2	3
1	5	9	6	2	3
1	5	9	6	2	3
1	5	6	9	2	3
1	5	6	2	9	3
1	5	6	2	3	9

SECOND PASS

1	5	6	2	3	9
1	5	6	2	3	9
1	5	6	2	3	9
1	5	2	6	3	9
1	5	2	3	6	9

THIRD PASS

1	5	2	3	6	9
1	5	2	3	6	9
1	2	5	3	6	9
1	2	3	5	6	9

FOURTH PASS

1	2	3	5	6	9
1	2	3	5	6	9
1	2	3	5	6	9

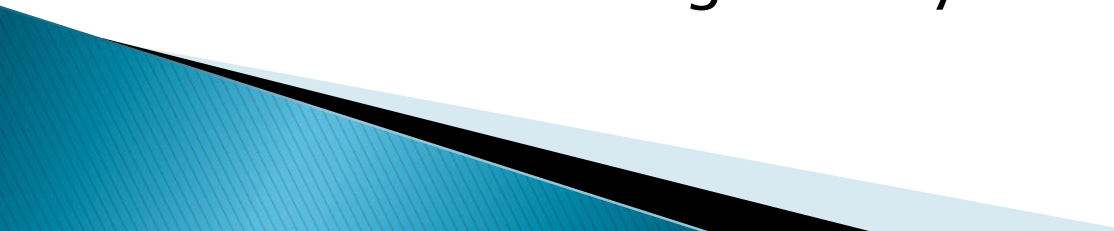
FIFTH PASS

1	2	3	5	6	9
1	2	3	5	6	9

Bubble Sort


```
for (int pass = 1; pass < 100; pass++)  
{  
    for (int j = 0; j < 100 - pass; j++)  
    {  
        /* bubble the larger number to the right */  
        if (A[j] > A[j+1])  
        {  
            int temp = A[j];  
            A[j] = A[j+1];  
            A[j+1] = temp;  
        } /* end if */  
    } /* end for */  
} /* end for */
```

Bubble Sort with Flag

- ▶ Observe if no swapping is performed during a pass, then it implies list of items is sorted.
 - ▶ we can improve efficiency of bubble sort algorithm by introducing flag that is initialized to 0 at beginning of every pass and is set to 1 when a swap occurs.
 - ▶ At the end of each pass, check if flag is still 0. If so, no swap has occurred at this pass, implying that the list is sorted. Thus, the bubble sort algorithm can terminate right away
- 

Modified Bubble Sort

```
for pass = 1 .. n-1
  exchange = false
  for position = 1 .. n-pass
    if element at position < element
      at position + 1
      exchange elements
      exchange = true
    end if
  next position
  if exchange = false BREAK
next pass
```

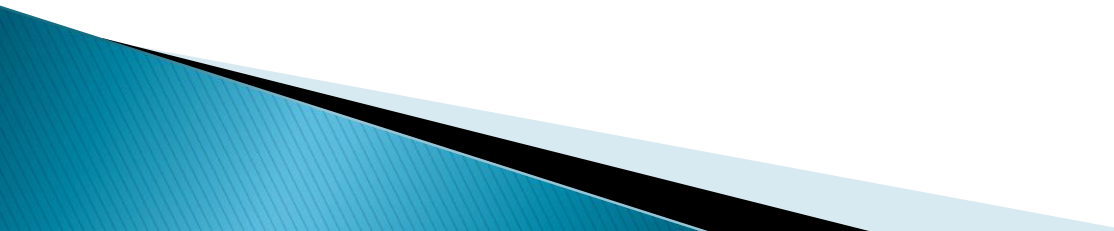


Bubble Sort with Flag

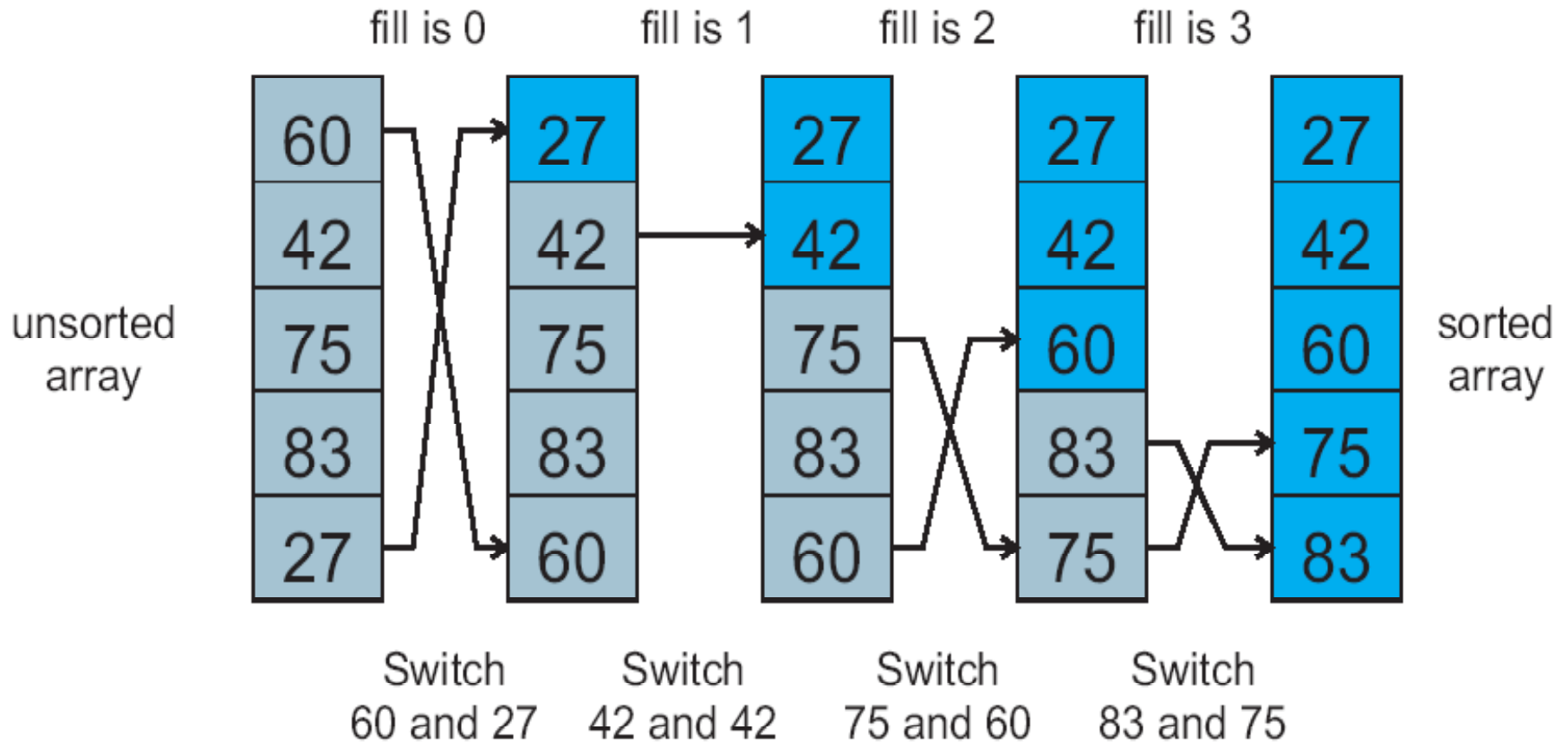
```
int swap = 1 ; /* Nothing swapped yet */
for (int pass = 1; pass < 100 && swap == 1; pass++)
{
    swap = 0; /* assume sorted */

    for (int j = 0; j < 100 - pass; j++) {
        /* bubble the larger number to the right */
        if ( A[j] > A[j+1]) {
            int temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
            swap = 1; /* swap implies not yet sorted */
        } /* end if */
    } /* end for */
} /* end for */
```

Selection Sort

- Find the smallest value in the list
 - Switch it with the value in the first position
 - Find the next smallest value in the list
 - Switch it with the value in the second position
 - Repeat until all values are in their proper places
- 

Selection Sort



Selection Sort

► An example:

3 9 6 1 2

smallest is 1:

1 9 6 3 2

smallest is 2:

1 2 6 3 9

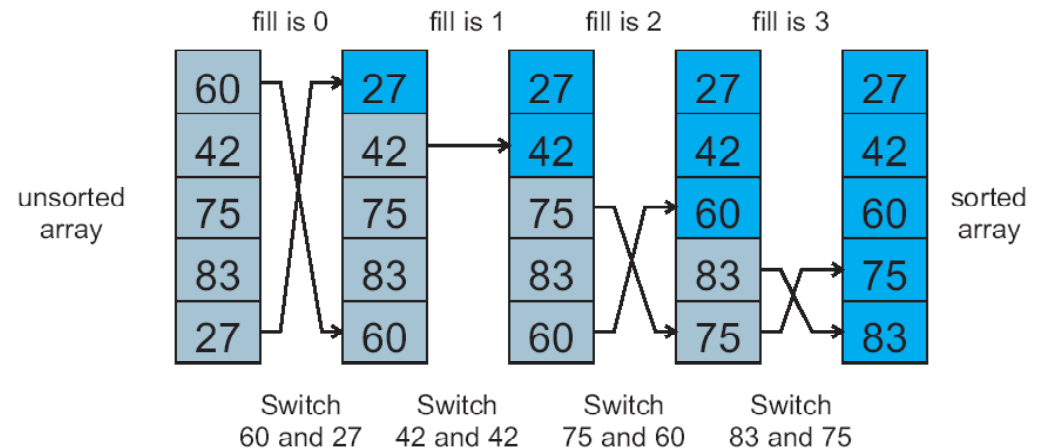
smallest is 3:

1 2 3 6 9

smallest is 6:

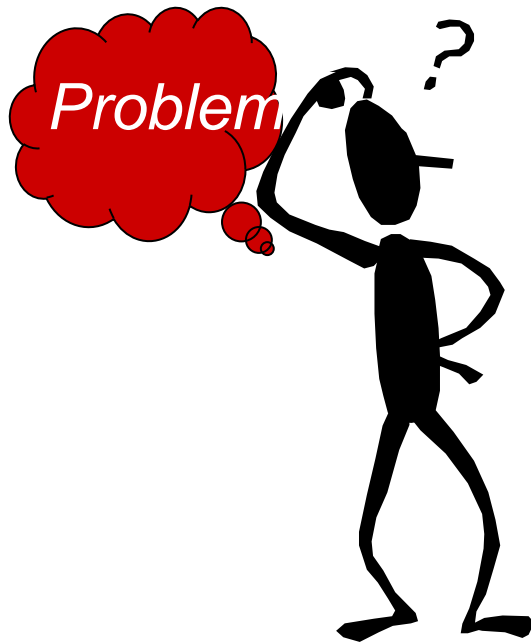
1 2 3 6 9

Selection Sort



- Solution Steps:
- For ($i = 0$; $i < \text{Last}$; $i++$)
- find smallest element **M** in subarray ($i .. \text{Last}-1$)
- Swap (**M** , first of this subarray)

Searching



0.32861447
0.16199208
0.60571415
0.11655935
0.50451736
0.64783047
0.78676896
0.22110392
0.25150021
0.25818766
0.97566364
0.25786005
0.29415618
0.93306238
0.86822721
0.21293442
0.32099153
0.02347026
0.83202107
0.30187124
0.45002303
0.93928335

Searching

- ▶ Searching is the process of determining if a *target* item is present in a list of items, and locating it
- ▶ A typical searching algorithm over an array returns the array index where the target item is found, or -1 if it is not found
- ▶ We will examine two specific algorithms:
 - Linear search
 - Binary search

Linear Search

- ▶ Linear search is also called sequential search
- ▶ The approach of linear search:
 - look at each item in turn, beginning with the first one, until either you find the target item or you reach the end of the array

Linear Search

```
loop
    if ((i > MAX) OR (my_array[i] = target))
        exit
    else
        i <- i + 1
Endloop
```

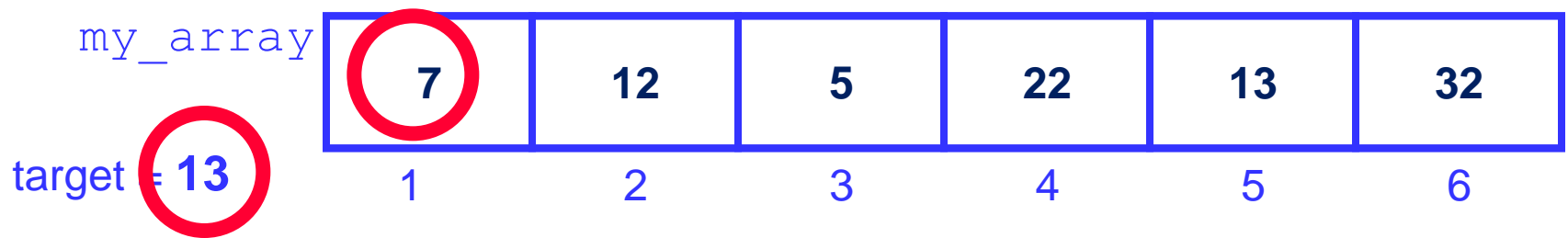
my_array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

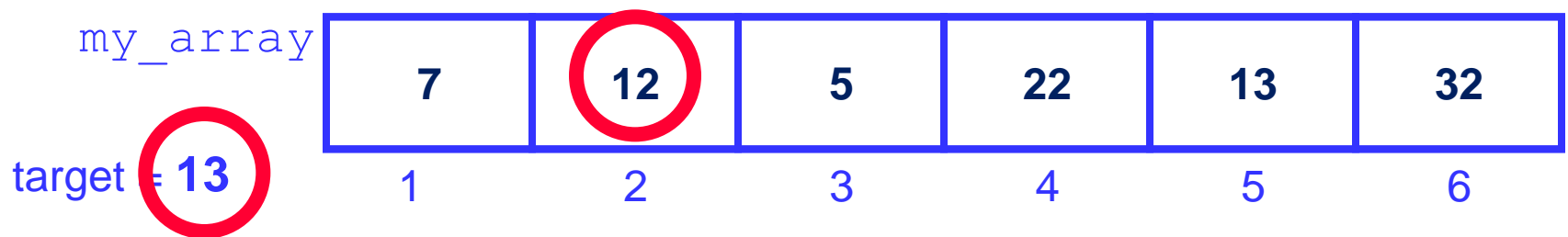
Linear Search

```
loop
    if ((i > MAX) OR (my_array[i] = target))
        exit
    else
        i <- i + 1
Endloop
```



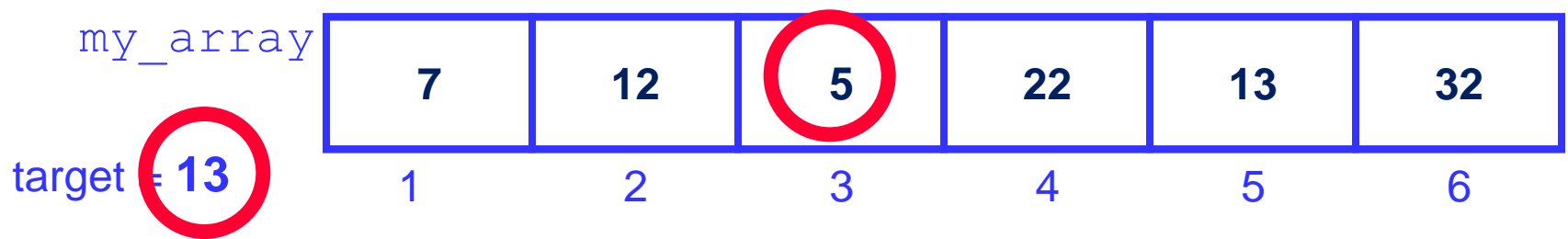
Linear Search

```
loop
    if ((i > MAX) OR (my_array[i] = target))
        exit
    else
        i <- i + 1
Endloop
```



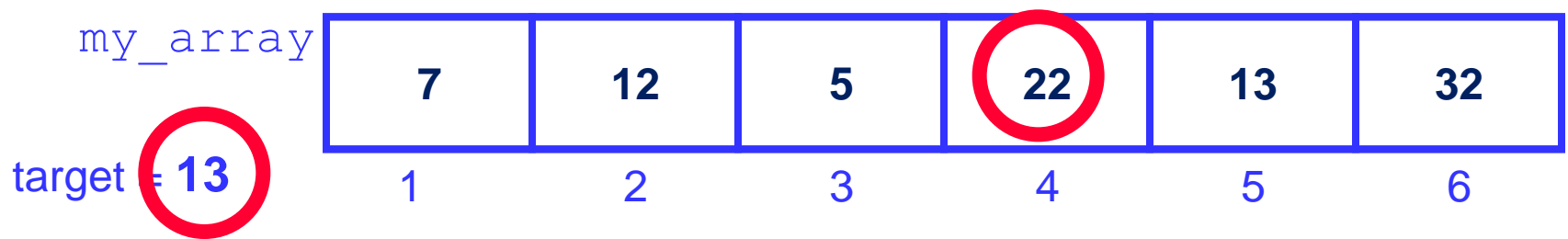
Linear Search

```
loop
    if ((i > MAX) OR (my_array[i] = target))
        exit
    else
        i <- i + 1
Endloop
```



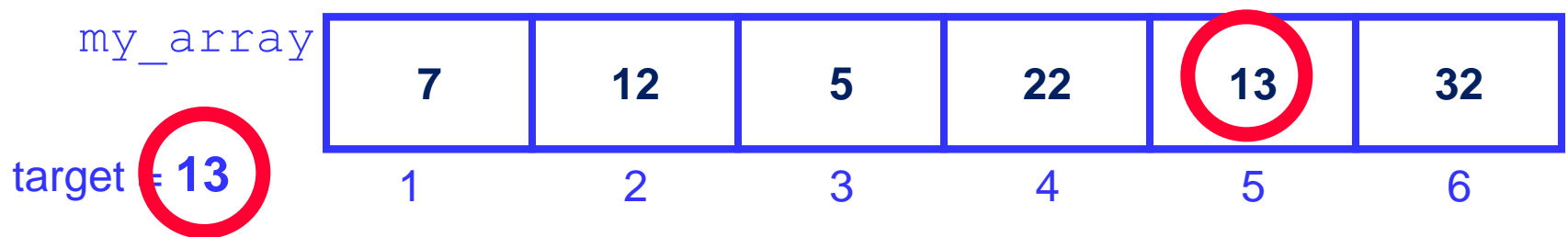
Linear Search

```
loop
    if ((i > MAX) OR (my_array[i] = target))
        exit
    else
        i <- i + 1
Endloop
```



Linear Search

```
loop
    if ((i > MAX) OR (my_array[i] = target))
        exit
    else
        i <- i + 1
Endloop
```



Linear Search

```
int Found = 0;
for ( i=0 ; i < 100 ; i++)
{
    if ( A[i] == target )
    {
        printf(" Found it ! At %d\n" , i);
        found = 1;
        break ;
    }
}
if (found ==0)
    printf(" Looked all over, Not here \n");
```


Linear Search

- ▶ Example:

3 9 6 1 2 23 8

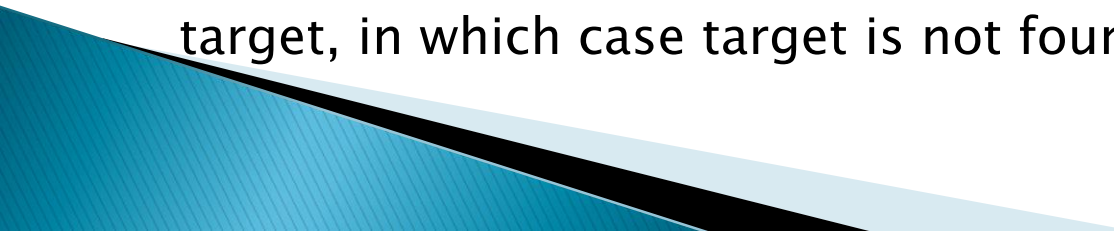
How many array elements are compared with the target value if the target is 3?

How many array elements are compared with the target value if the target is 2?

How many array elements are compared with the target value if the target is 8?

How many array elements are compared with the target value if the target is 5?

Binary Search

- ▶ If an array is sorted
 - ▶ Check whether the middle element is equal to target. If so, it done.
 - ▶ Otherwise, determine which half of array the target should be in
 - ▶ Perform binary search again on sub-array by comparing its middle element with target
 - ▶ Repeatedly divide array in half, until desired item is found, or you have a sub-array contains only one element not equal to target, in which case target is not found.
- 

Binary Search

- ▶ Example: target value is 25

Array = 1, 3, 4, 7, 9, 12, 17, 20, 21, 25, 34, 39, 41

Middle element = 17 , 25 should be in second half

Sub-array = 20, 21, 25, 34, 39, 41

Middle element = 34 , 25 should be in first half

Sub-array = 20, 21, 25

Middle element = 21 , 25 should be in second half

Sub-array = 25

Middle element = 25, done!

Binary Search

Example: target value is 6

Array = 1, 3, 4, 7, 9, 12, 17, 20, 21, 25, 34, 39, 41

Middle element = 17 , 6 should be in first half

Sub-array = 1, 3, 4, 7, 9, 12

Middle element = 7 , 6 should be in first half

Sub-array = 1, 3, 4

Middle element = 3 , 6 should be in second half

Sub-array = 4

Middle element = 4 != 6, failed!

Binary Search

```
int low = 0;
int high = 99;
int found = 0;
int mid ;

while ((low <= high) && (found==0)) {
    mid = (low + high) / 2;

    if (target == A[mid])
    { printf(" Found it ");
      found = 1 ;
    }
    else if (target < A[mid])
        high = mid - 1; /* consider 1st half of array */
    else
        low = mid + 1;  /* consider 2nd half of array */
}

if (found == 0 )
    printf(" Definitely NOT here ");
```

► Example: tracing binary search

2 3 7 7 9 15 16 18 20 21

target = 9, # comparisons = 1			
Iteration	low	mid	high
1	0	4	9

target = 7, # comparisons = 3			
Iteration	low	mid	high
1	0	4	9
2	0	1	3
3	2	2	3

Example: tracing binary search

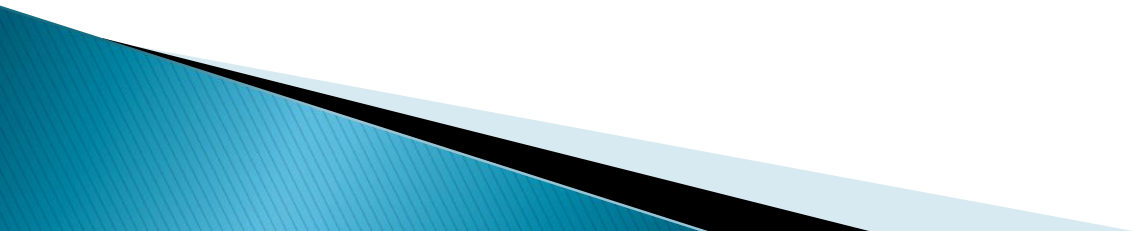
2 3 7 7 9 15 16 18 20 21

target = 21, # comparisons = 4			
Iteration	low	mid	high
1	0	4	9
2	5	7	9
3	8	8	9
4	9	9	9

target = 22, # comparisons = 4			
Iteration	low	mid	high
1	0	4	9
2	5	7	9
3	8	8	9
4	9	9	9
	10		9

Questions?

Searching and Sorting
in
Two Dimensional Arrays?

A decorative graphic in the bottom-left corner consisting of overlapping blue and black geometric shapes, possibly representing a stylized wave or a corner element.