

CMPE 1666-Intermediate Programming

Lecture 7 - Recursive Searching and Sorting
Algorithms

Binary Search Algorithm

- ▶ Previously we had seen that we can perform a linear search (sequential search) on an array or a list.
- ▶ We have also seen that If our list is sorted, we can have an improved algorithm in that for non-existent values, we don't need to search the whole array/list.
- ▶ However, if our array/list is sorted we can do much better than the improved sequential search, by using the **Binary Search Algorithm**.

Binary Search Algorithm

- ▶ In this algorithm, we compare our search value with the middle element (element at the middle index) in the list.
- ▶ If we find a match we stop there.
- ▶ If we don't find a match then our search value will be either smaller than the middle element or bigger than it.
- ▶ If our element is smaller than the middle element, we only need to search the 1st half of the list.
- ▶ If our search element is bigger we only need to compare our element with the second half.

Binary Search Algorithm

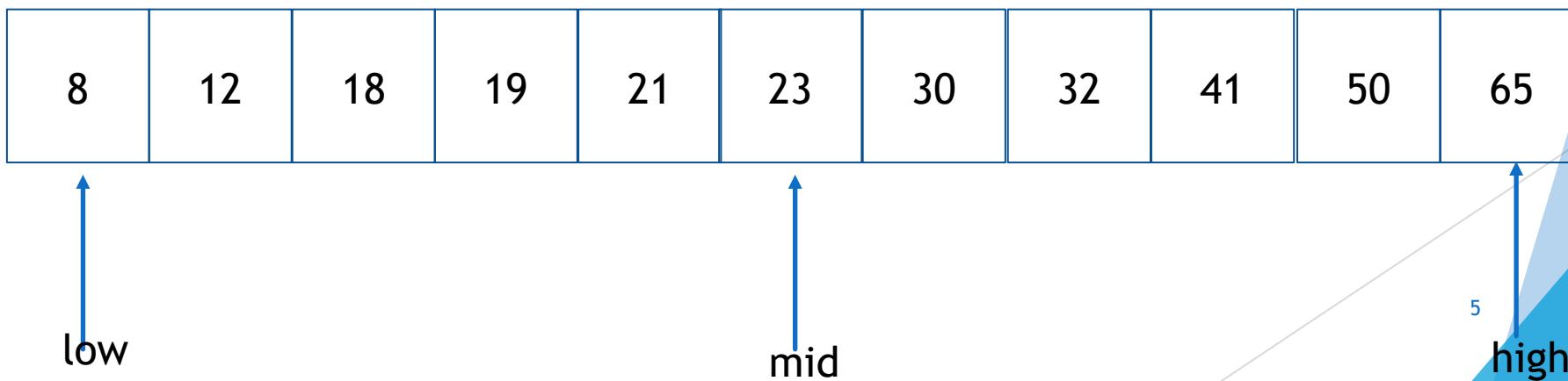
Important Note

Binary Search works on **sorted** Arrays or Lists.

To use Binary Search, always ensure that your list/array is sorted.

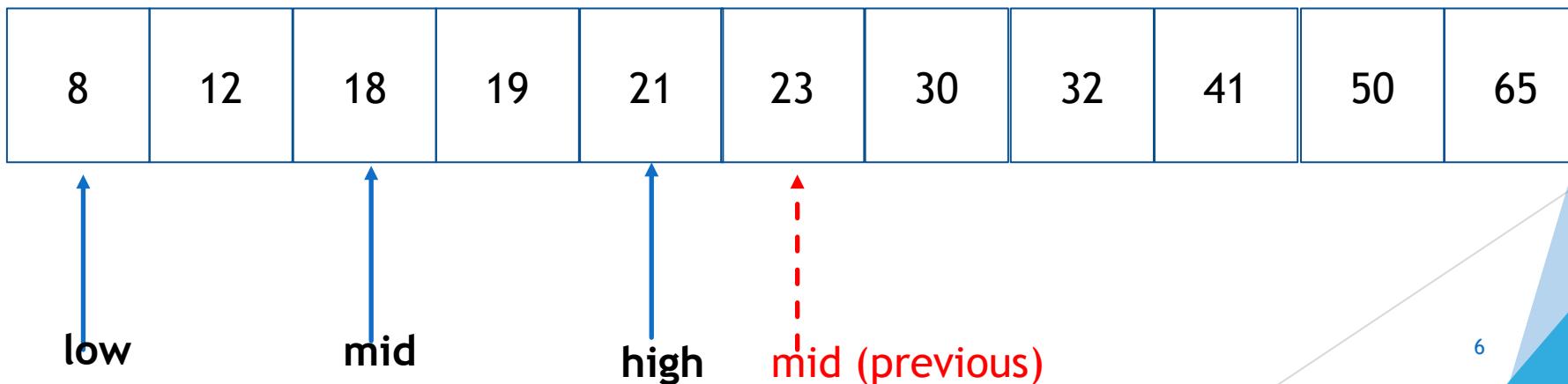
Binary Search algorithm- tracing

- ▶ Consider Searching value 18 in the list below
 - ▶ We initially keep an index, **low**, as the first index of the list and an index **high** as the last index
 - ▶ For the given list, Initially **low=0, high=10** (since we have 11 values)
 - ▶ Then we find the middle index **mid**
 - ▶ **mid=(low+ high)/2 =5**
 - ▶ We compare our search value with **L[mid]**, i.e **L[5]**



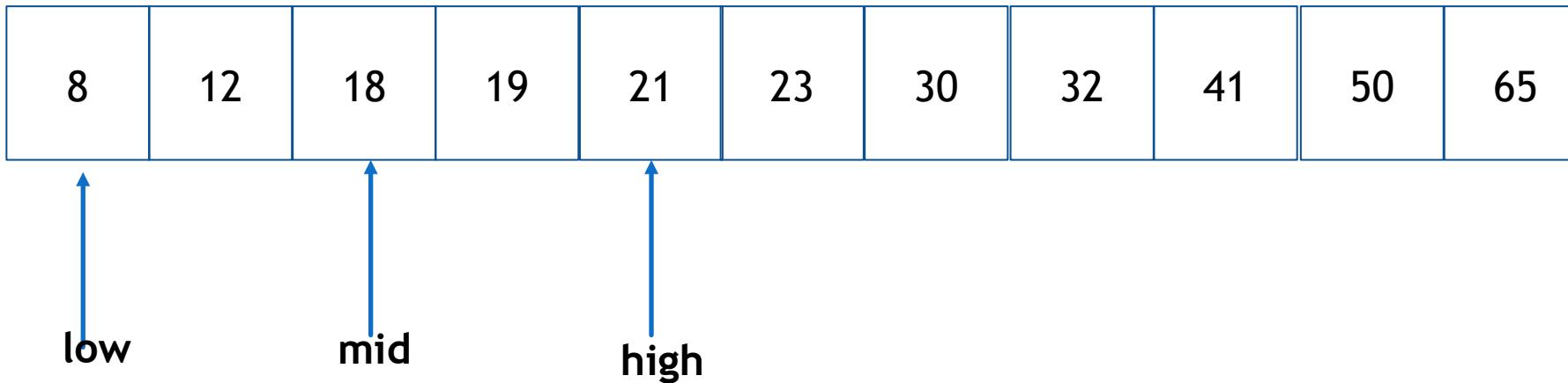
Binary Search algorithm - tracing

- ▶ Our first consideration is whether our search value is equal to L[mid]
- ▶ If the values match, we have found the value and we can return mid
- ▶ In our case, it doesn't match.
- ▶ Since our search value (18) is less than the mid value (23), we'll search the 1st half of the list.
 - We do this by moving **high** to **mid - 1**
 - Then, we recalculate **mid** $((\text{low}+\text{high})/2)= (0+4)/2=2$



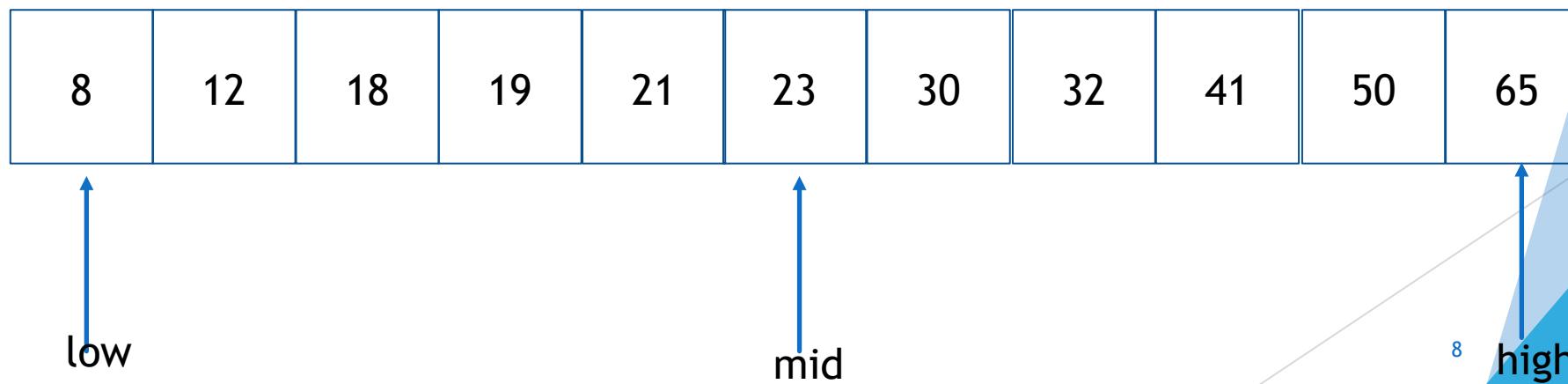
Binary Search algorithm - tracing

- At this point we see a match between $L[mid]$ and our search value (18), thus we return the position **mid** and the algorithm terminates.



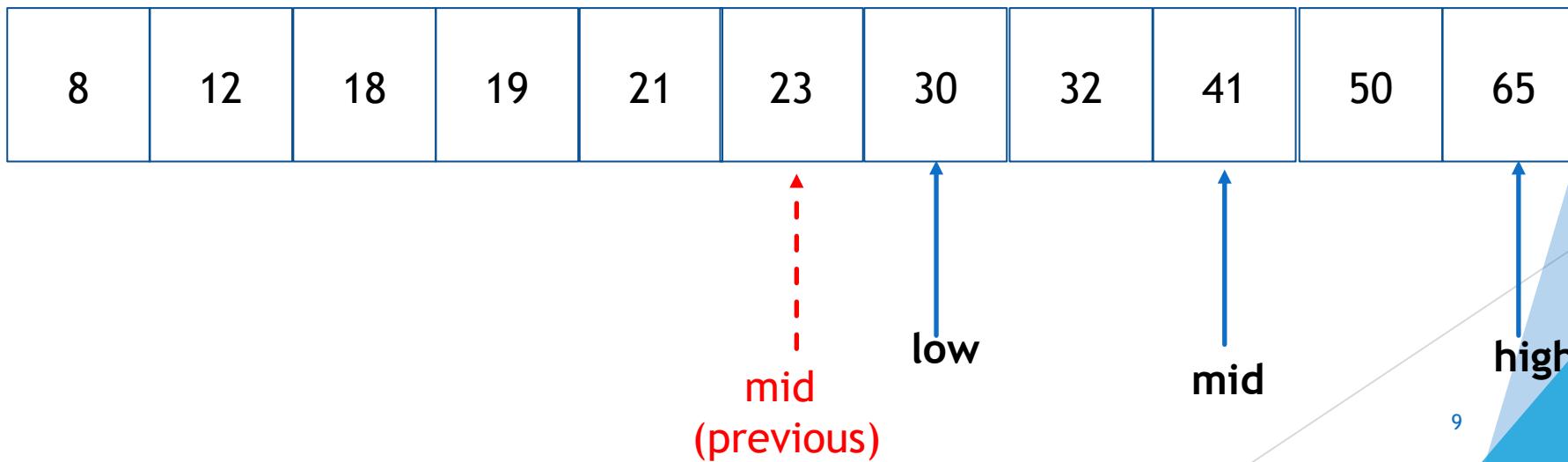
Binary Search algorithm - tracing

- ▶ Consider that we want to search the value 50
 - ▶ Again, initially low=0, high=10
 - ▶ $\text{mid} = (\text{low} + \text{high}) / 2 = 5$
 - ▶ We compare our search value with $L[\text{mid}]$, i.e $L[5]$
 - ▶ In this case our search value (50) is higher than the mid value (23)
 - ▶ We thus need to search the second half of the list



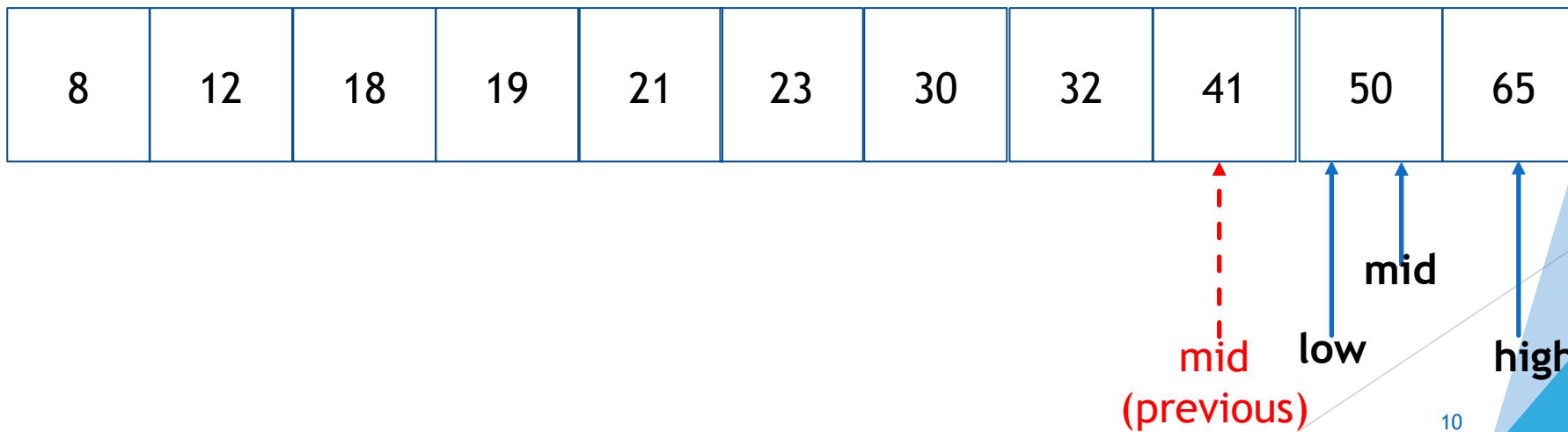
Binary Search algorithm - tracing

- ▶ This time we move **low** to **mid+1** (i.e 6)
- ▶ And we recalculate $\text{mid} = (\text{low} + \text{high})/2 = (6+10)/2 = 8$
- ▶ Again, we compare our search value (50) with $L[\text{mid}]$ (41).
- ▶ Our search value is greater, so we consider the upper half of this new search range.



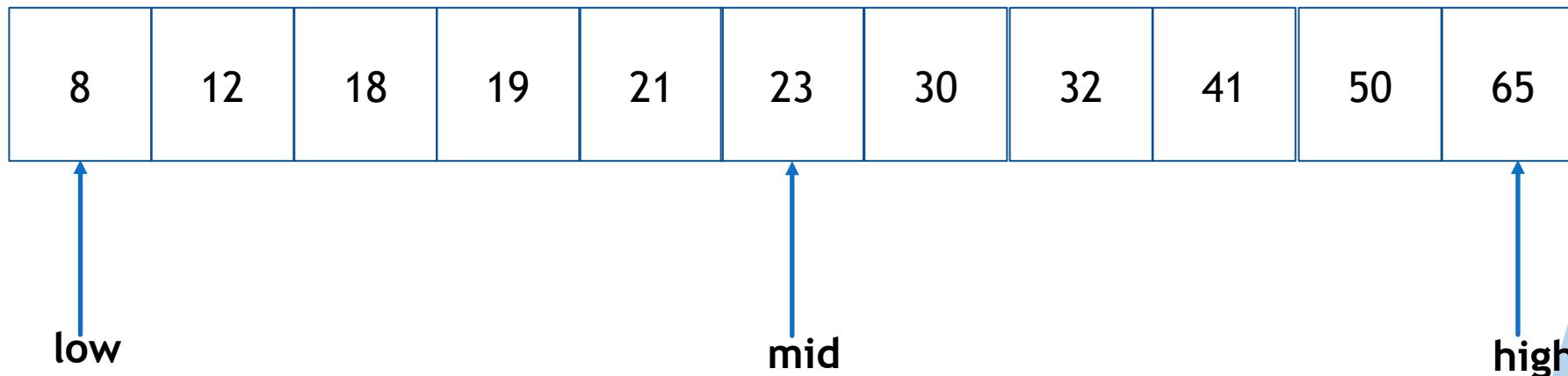
Binary Search algorithm - tracing

- ▶ New value of **low** is now $\text{mid}+1=9$
- ▶ We recalculate **mid** as $(\text{low}+\text{high})/2 = (9 + 10)/2 = 9$
- ▶ This time there is a match and the position (9) is returned.



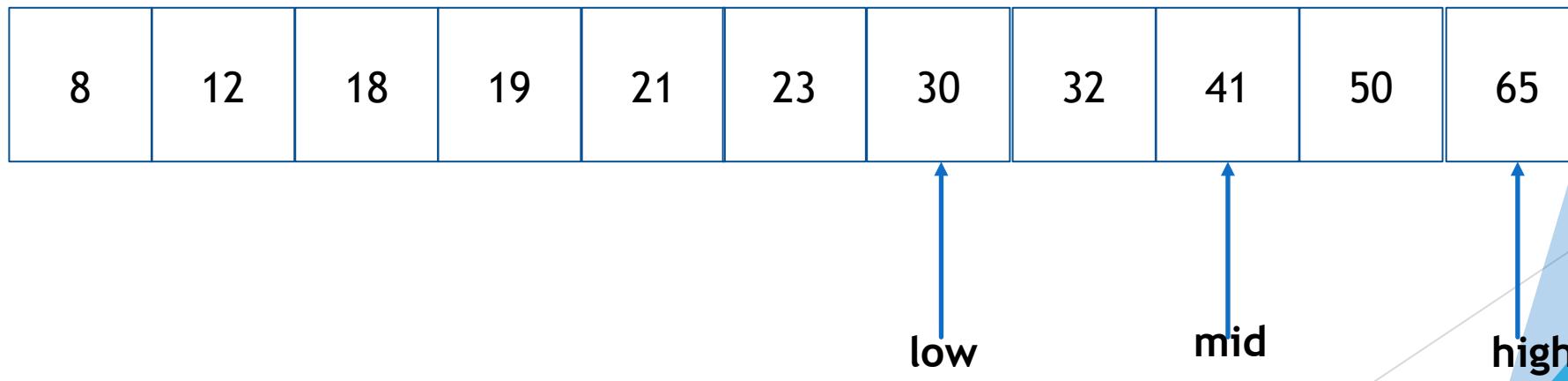
Binary Search algorithm - tracing

- ▶ Let's now search value 35, which is not present in the list.
- ▶ Again we start with **low=0**, **high=10** and **mid=(low+high)/2= 5**
- ▶ We compare our search value with $L[mid]$



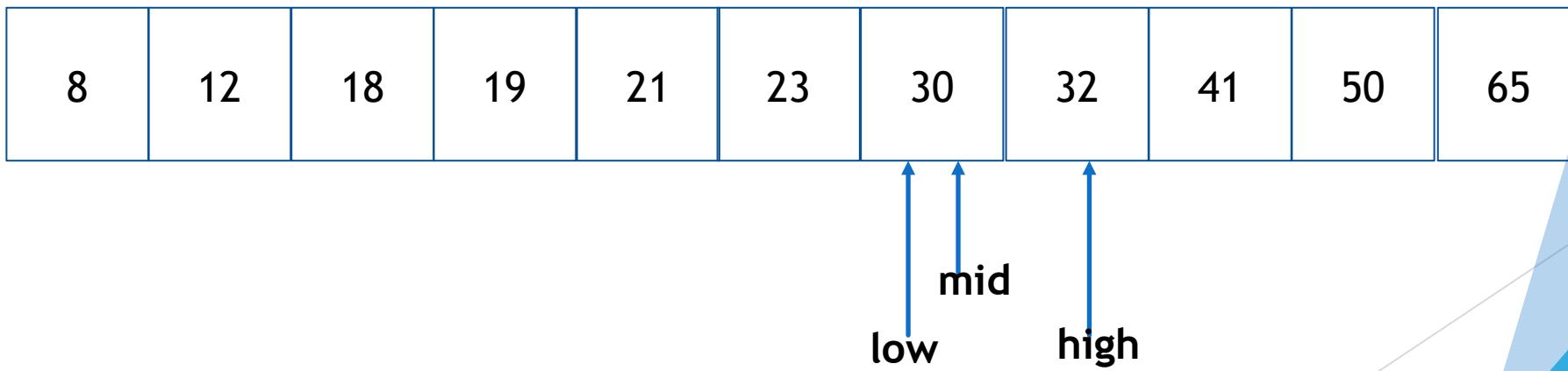
Binary Search algorithm - tracing

- ▶ Our search value is greater, so we consider the upper half of this new search range.
- ▶ Our **low** moves to position 6 and **mid** to position 8
- ▶ We again compare our search value (35) with $L[\text{mid}]$
- ▶ It's now lower than $L[\text{mid}]$



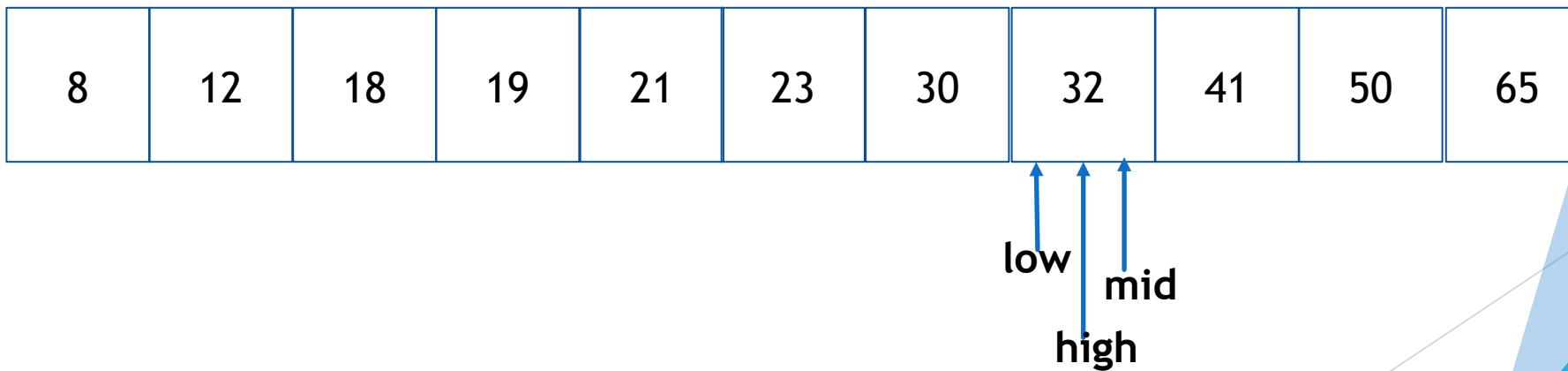
Binary Search algorithm - tracing

- ▶ High moves to mid-1 (i.e position 7)
- ▶ Mid moves to $(\text{low}+\text{high})/2 = (6+7)/2=6$
- ▶ Our search value > L[mid]



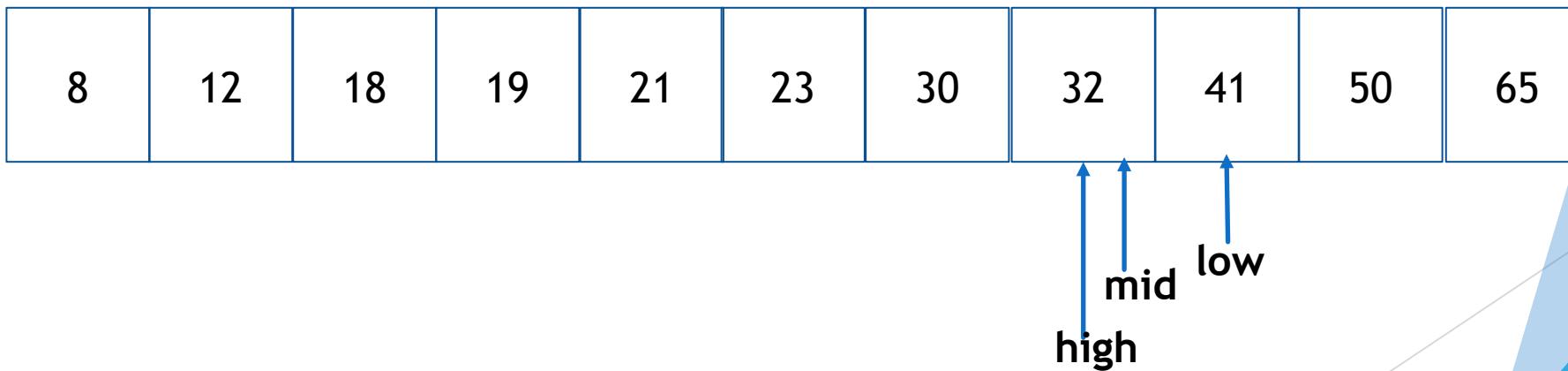
Binary Search algorithm - tracing

- ▶ low moves to mid+1 (i.e position 7)
- ▶ mid moves to $(\text{low}+\text{high})/2 = (7+7)/2 = 7$
- ▶ Our search value > L[mid]



Binary Search algorithm - tracing

- ▶ Since our search value > L[mid]
- ▶ low moves to mid+1 ($7+1 = 8$)
- ▶ Since low > high, the algorithm stops and returns -1



Binary Search (Recursive)

- The List **L**,
- A value **low**, representing the first index to consider
- A value **high** representing the last index to consider
- The search value **val**
- So a call to the method will typically be **binarySearch(L, low, high, val)**
- ▶ Initially **binarySearch()** is called with value 0 for **low** L.Count-1 for **high**
 - The search value is compared with **L[mid]** if there is a match **mid** is returned.
 - If **val < L[mid]**, the method is called recursively for the lower half of the list. Thus the parameters are **L, low, mid-1** and **val**
 - If **val > L[mid]**, the method is called recursively with the upper half of the list. Thus the parameters are **L, mid+1, high** and **val**
 - The recursion stops when either **L[mid]==val** (a match is found) or **high<low** (the search value is not in the list)

Binary Search (recursive) - C# Code

```
// A recursive binary search method. It returns location of val in given //list L, if
present, otherwise -1
static private int RBinarySearch(List<int> L, int low, int high, int val)
{
    if (high >= low)
    {
        int mid = (low + high) / 2;
        // If the element is present at the middle
        if (L[mid] == val)
            return mid;
        // If element is smaller than L[mid], then it can only be present in left sublist
        if (val < L[mid])
            return RBinarySearch(L, low, mid - 1, val);
        // Else the element can only be present in right sublist
        return RBinarySearch(L, mid + 1, high, val);
    }

    // We reach here when element is not present in the list
    return -1;
}
```

Binary Search (recursive)-Test Program

```
static void Main(string[] args)
{
    List<int> myList = new
    List<int>(){8,12,15,19,23,38,45,56};

    int P1;
    int value;

        value = 56;
        P1 = RBinarySearch(myList,0,myList.Count-1,value);
        if (P1 > -1)
            Console.WriteLine($"{value} found at position {P1}");
        else
            Console.WriteLine($"{value} not found ");

        value = 12;
        P1 = RBinarySearch(myList,0,myList.Count-1, value);
        if (P1 > -1)
            Console.WriteLine($"{value} found at position {P1} ");
        else
            Console.WriteLine($"{value} not found");

        value = 40;
        P1 = RBinarySearch(myList,0, myList.Count-1, value);
        if (P1 > -1)
            Console.WriteLine($"{value} found at position {P1} ");
        else
            Console.WriteLine($"{value} not found ");
}
```

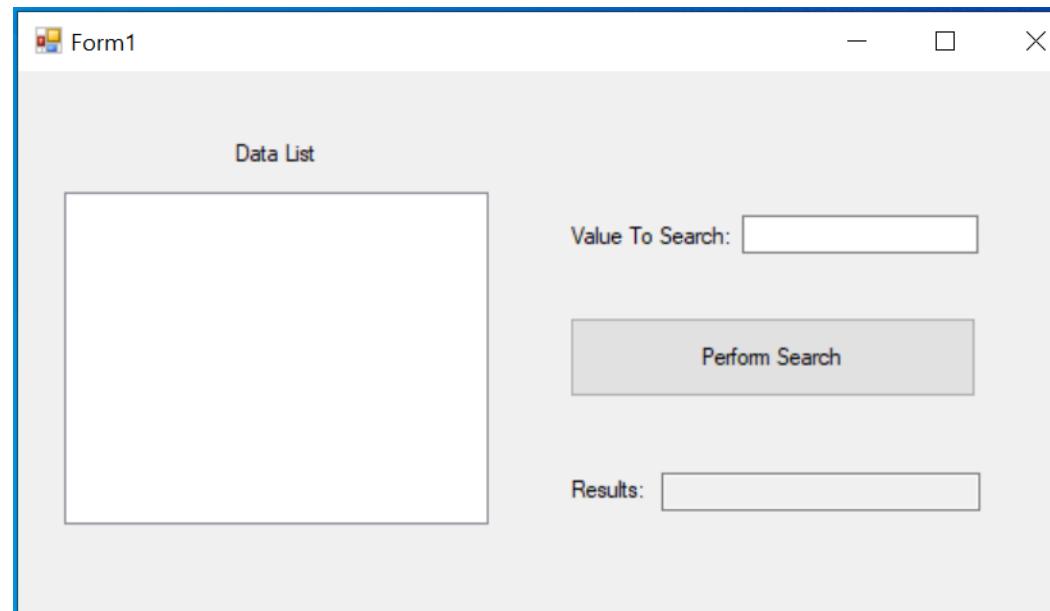
Output

cs Microsoft Visual Studio Debug Console

```
56 found at position 7
12 found at position 1
40 not found
```

Lecture7- Exercise1

- ▶ Download the zip file for Lecture7-Exercise1 and extract the project
- ▶ To the form class, add a method BinarySearch() that uses the recursive binary search technique to search for a given value in a list of strings
- ▶ Add an event handler such that when the form loads, the values from the provided list are loaded into the listbox
- ▶ When the user clicks on the “Perform Search” button, the value in the “Value To Search” textbox must be searched from the list using the BinarySearch() method. If the value is found, the message **Value <value> found at index <index>** must be displayed in the read-only textbox. Otherwise, the message **Value <value> not found** must be displayed
- ▶ Clear the value textbox each time its value has been used



Complexity of Binary Search

- ▶ Consider that we have a list of 16 elements to search.
- ▶ So our search space is 16 elements.
- ▶ In one step we divide it by 2 and we have a search space of only 8 elements
- ▶ In the next step our search space becomes 4 elements
- ▶ In 2 more steps our search space becomes only 1 element and we can decide on whether our value is in the list or not.
- ▶ Thus in a maximum of 4 comparisons our algorithm is over
- ▶ What if we increase our list size to 32?
- ▶ Again in one step our search space becomes 16
- ▶ Thus doubling our list size only adds one more comparison²¹

Quicksort

- ▶ It is a Divide and Conquer algorithm
- ▶ Picks an element as **pivot** and **partitions** the array around the pivot
- ▶ Pivot can be chosen in different ways
 - ▶ Pick a random element
 - ▶ Pick the middle element
 - ▶ Pick the first element
 - ▶ **Pick the last element**

QuickSort

- ▶ An informal specification of QuickSort is as follows:
 1. Pick a pivot
 2. Bring the pivot to its position, such that all elements smaller than the pivot are now before it and all higher elements are after it.
 3. Repeat the same procedure for all elements before the pivot and all elements after the pivot

Quicksort

- ▶ E.g. Consider the following array.

50	21	12	18	30	8	23	65	41	32
----	----	----	----	----	---	----	----	----	----

- ▶ Our first pivot will be 32.
- ▶ After partitioning our array will be as below

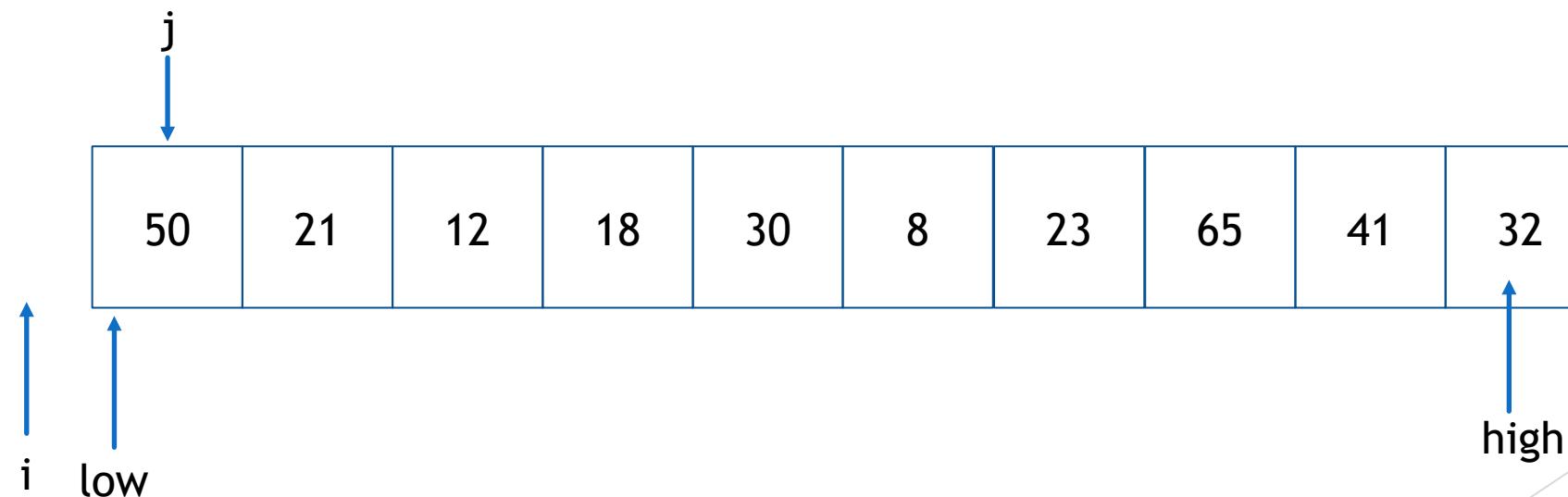
21	12	18	30	8	23	32	65	41	50
----	----	----	----	---	----	----	----	----	----



- ▶ We then repeat the same procedure with the part of the array before the pivot and the part of the array after the pivot.

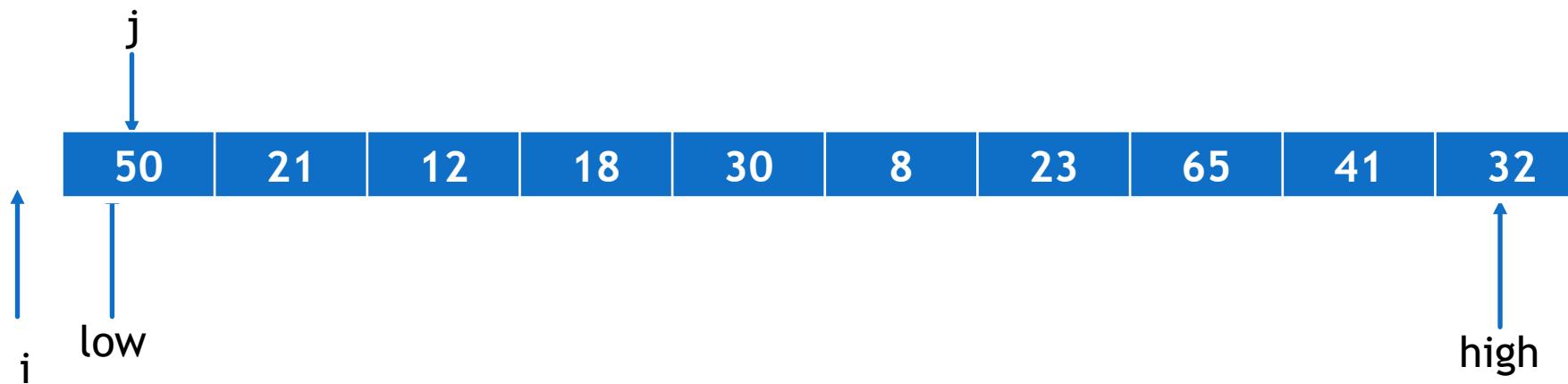
The partitioning procedure

- ▶ We start with the pivot and 3 pointers, **i**, **low**, **high**. The pointers **low** and **high** point respectively to the lowest and highest positions of the part of the array under consideration.
 - ▶ We initialize **i** to **low-1**
 - ▶ We then use an additional pointer **j** that will start with value **low** and incremented in each iteration until it reaches the value of **high**



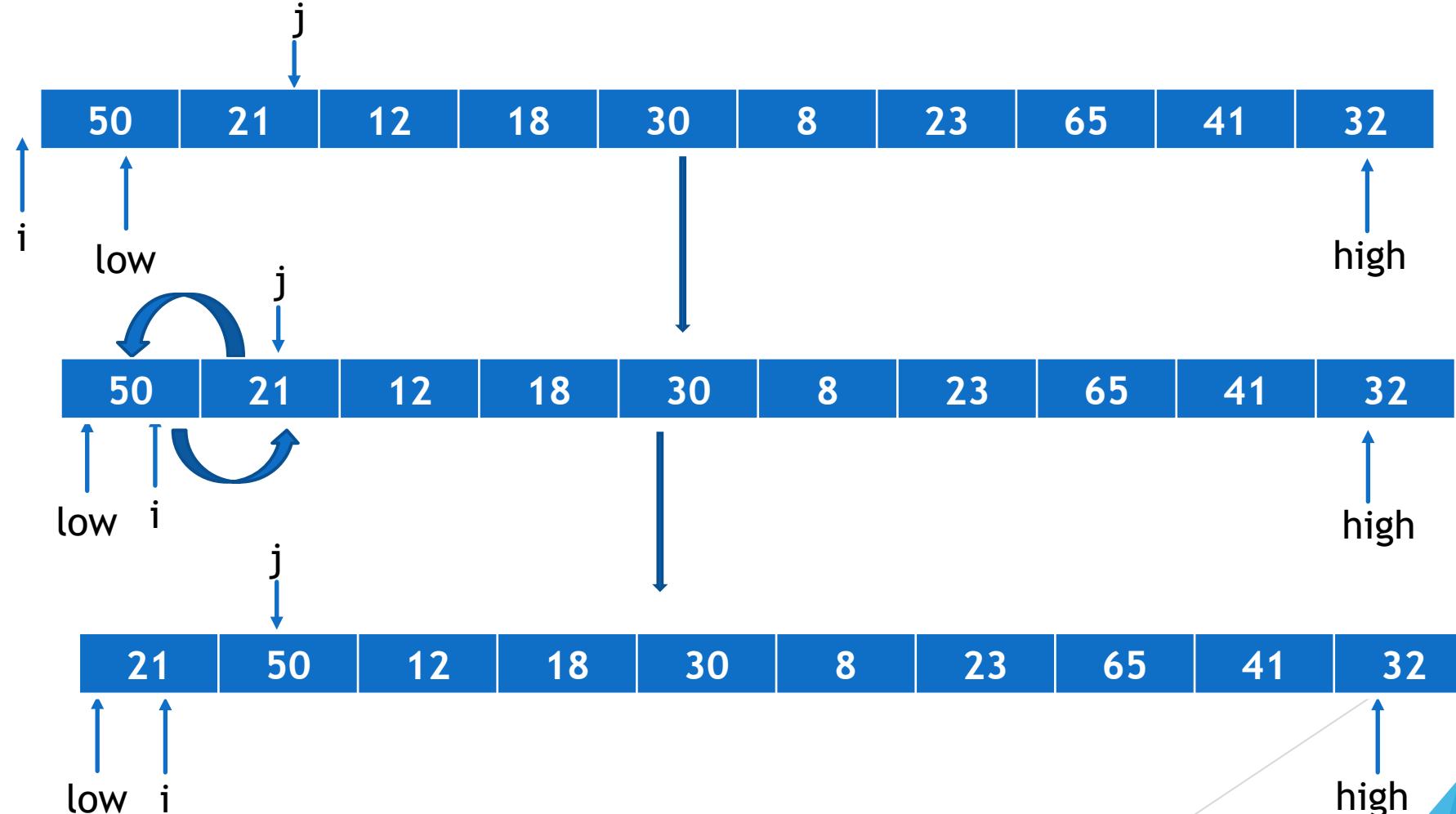
The partitioning procedure

- We'll call the array A.
 - As we move j, each time we obtain an $A[j]$ smaller than the pivot, we increment i and swap $A[j]$ with $A[i]$
 - First value $A[j]$ is greater than the pivot. So we don't do anything.



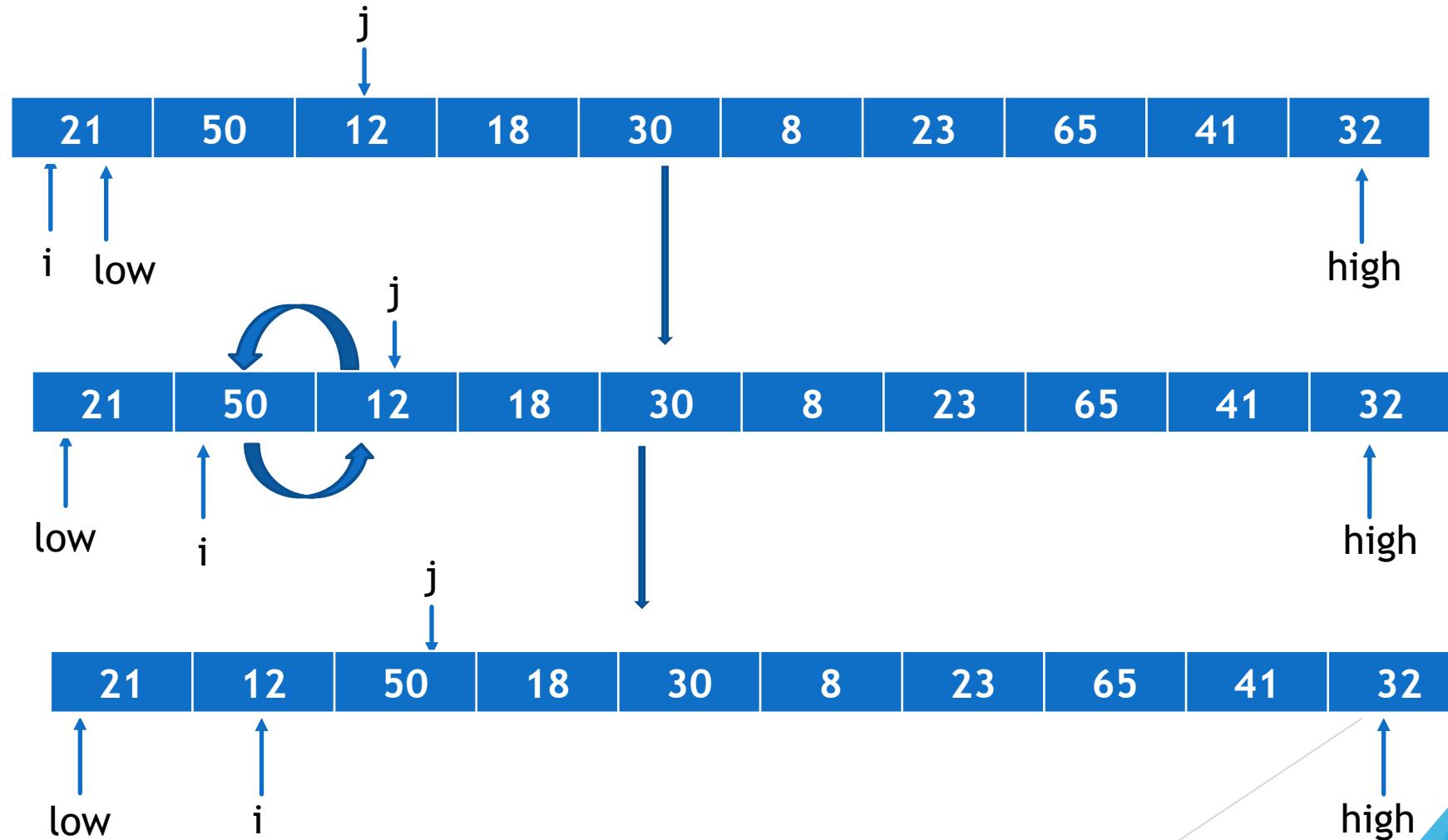
The partitioning procedure

- At the next value of j , $A[j] < \text{pivot}$. So we increment i and swap $A[j]$ and $A[i]$

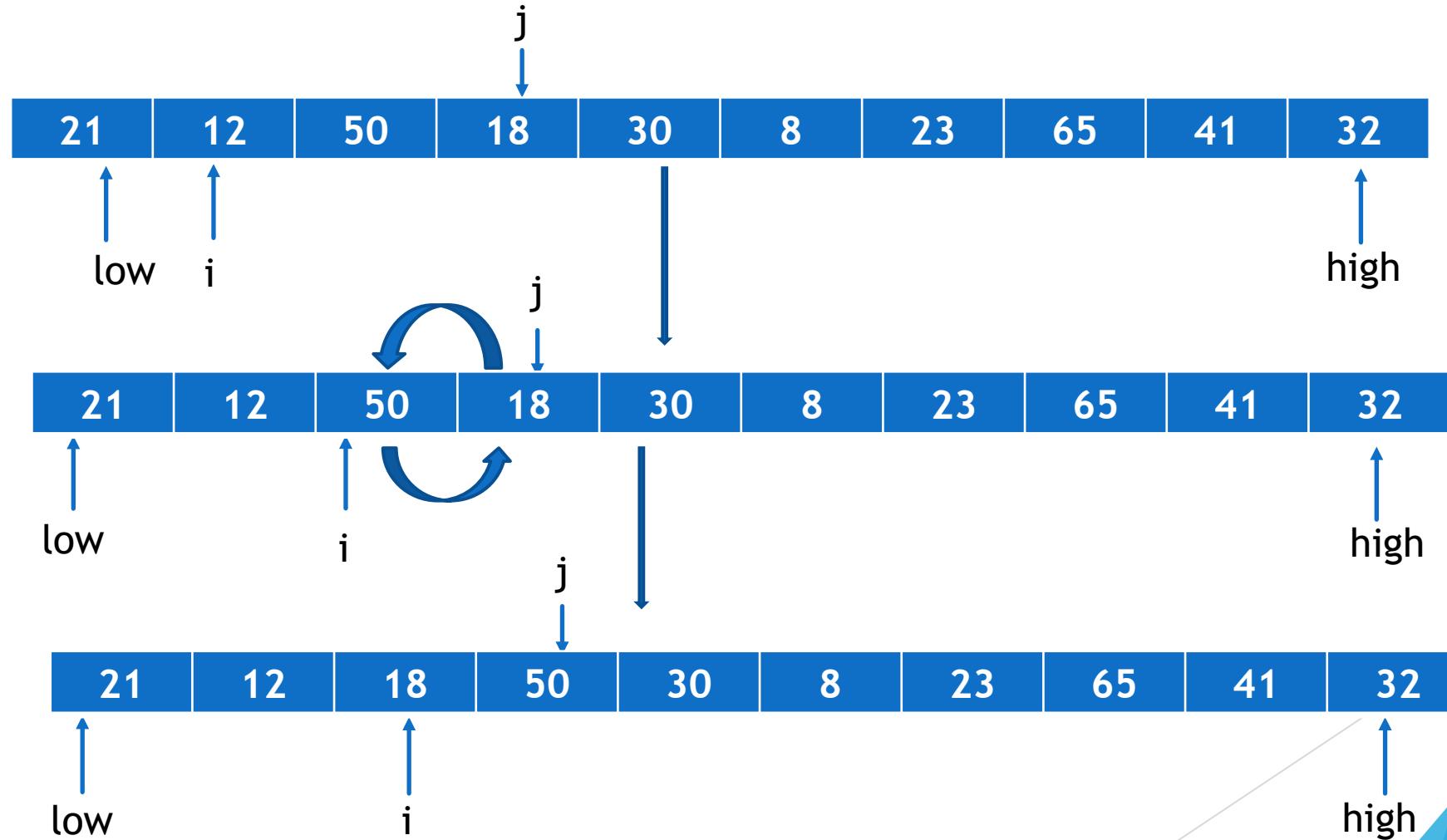


The partitioning procedure

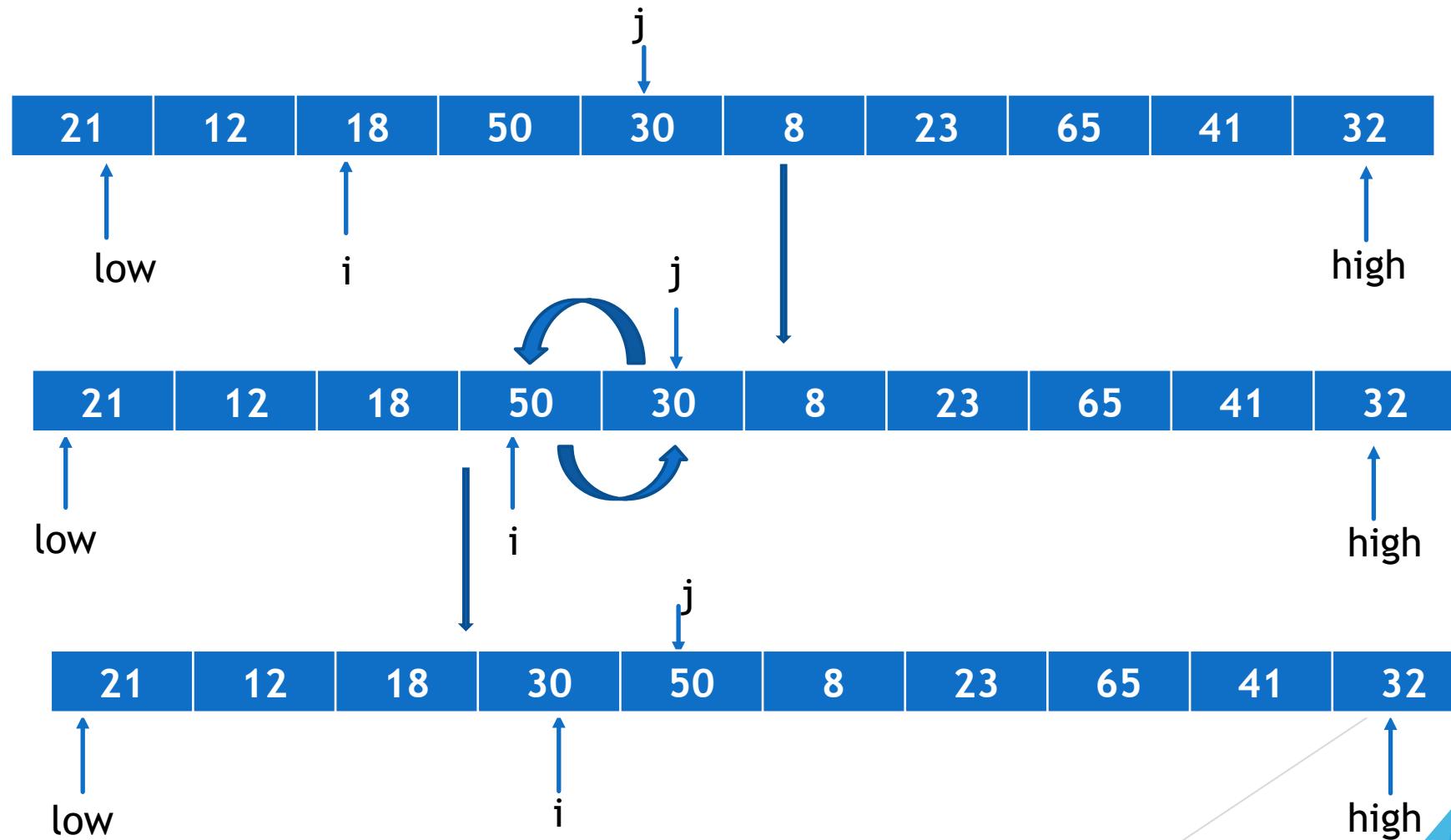
- ▶ Next value of j , we swap again



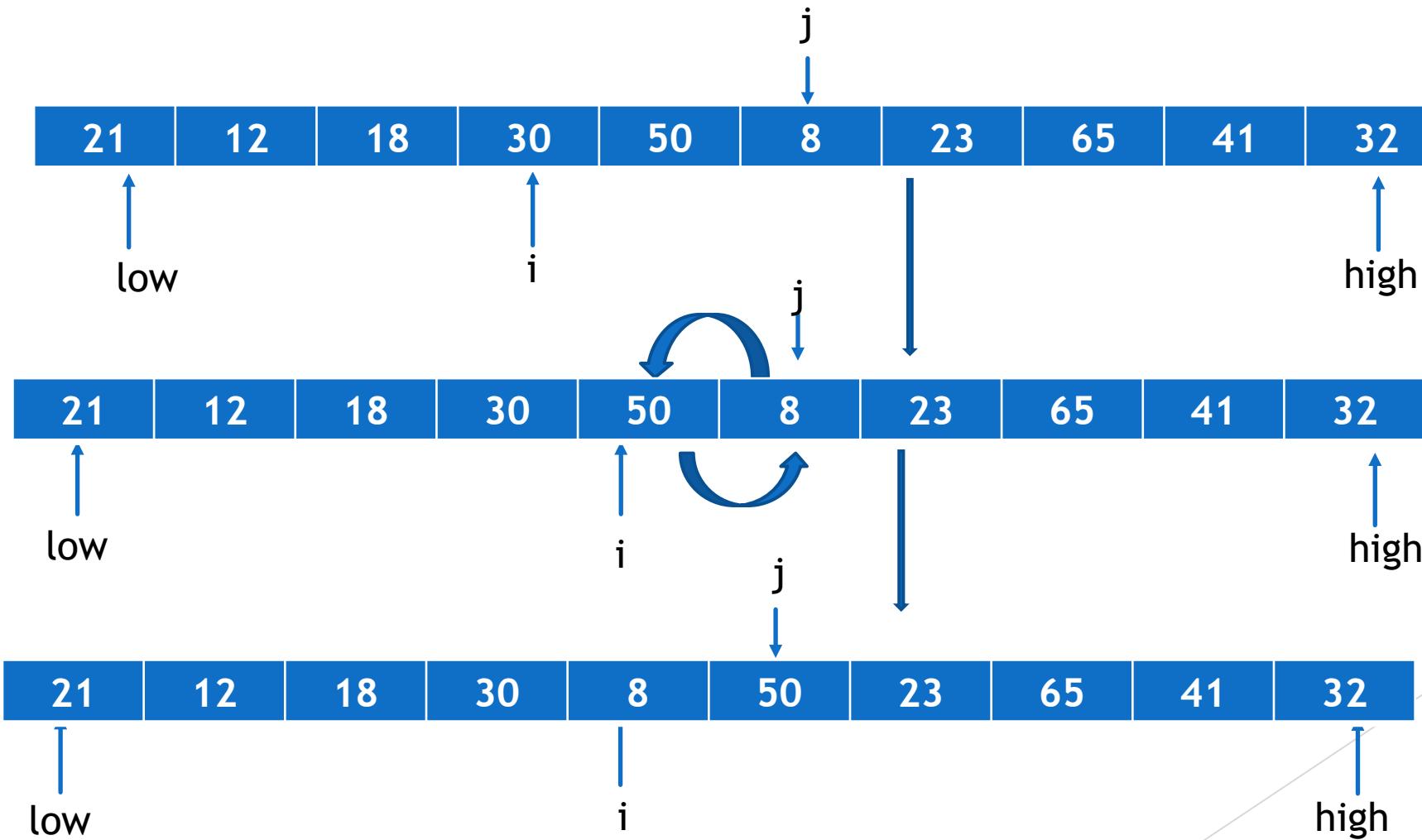
The partitioning procedure



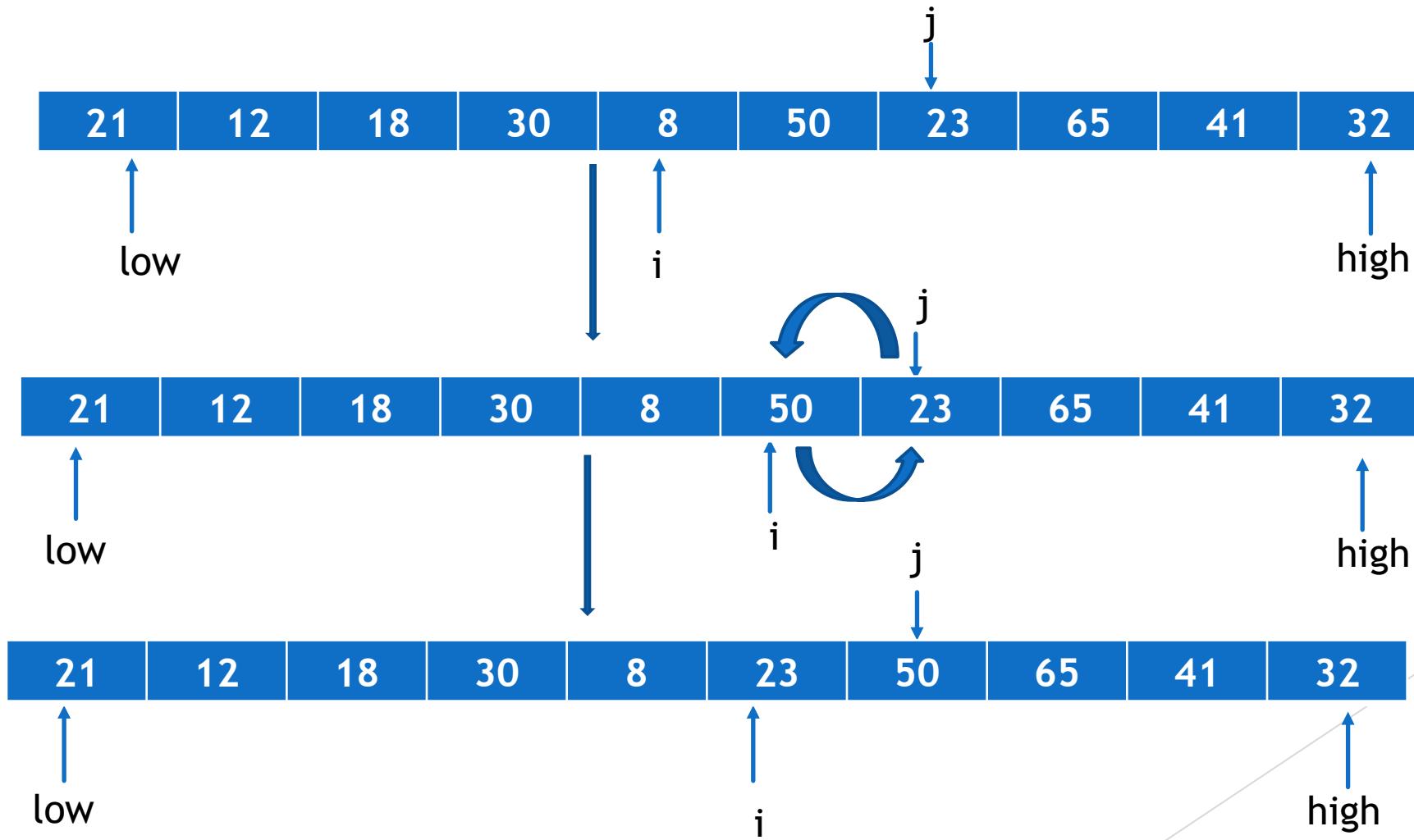
The partitioning procedure



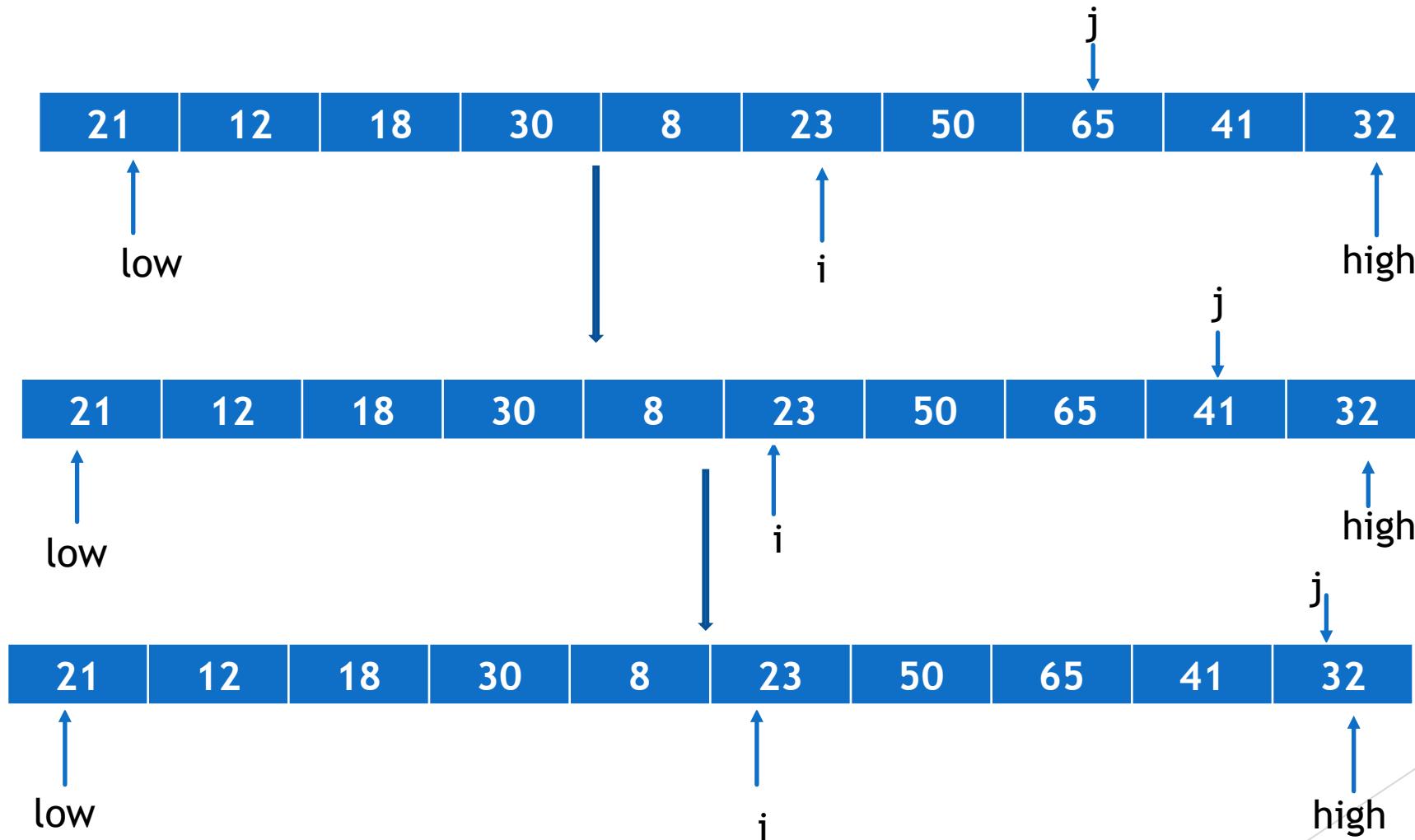
The partitioning procedure



The partitioning procedure

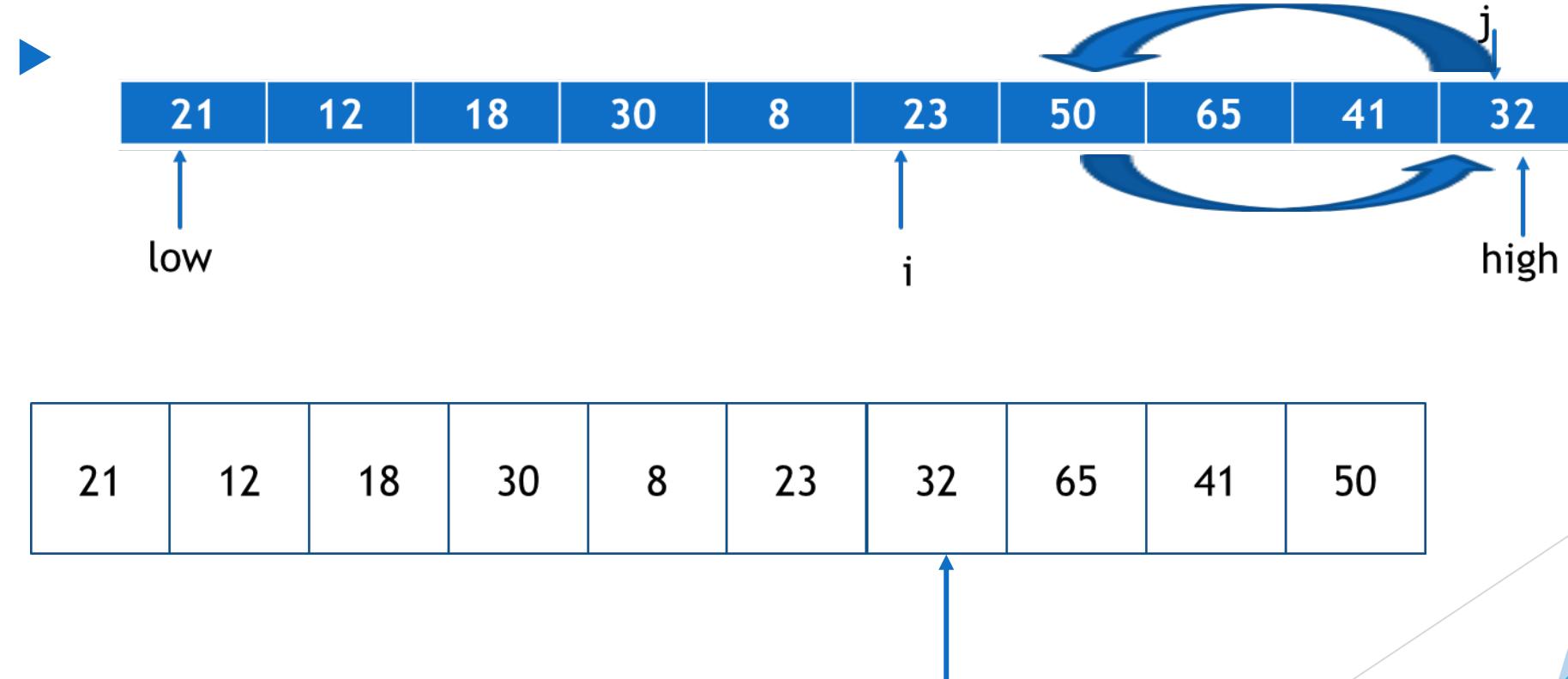


The partitioning procedure



The partitioning procedure

- ▶ Now that j is at high, we swap $A[i+1]$ and $A[high]$



QuickSort

- ▶ Example : Picking the last element as the pivot

[50 21 12 18 30 8 23 65 41 32]

[21 12 18 30 8 23 32 41 65 50]

[21 12 18 30 8 23] 32 [41 65 50]

[21 12 18 30 8 23] 32 [41 65 50]

[21 12 18 8 23 30] 32 [41 50 65]

[21 12 18 8] 23 [30] 32 [41] 50 [65]

QuickSort

[21 12 18 8] 23 [30] 32 [41] 50 [65]

[8 21 12 18] 23 [30] 32 [41] 50 [65]

8 [21 12 18] 23 30 32 41 50 65

8 [21 12 18] 23 30 32 41 50 65

8 [12 18 21] 23 30 32 41 50 65

8 [12] 18 [21] 23 30 32 41 50 65

8 12 18 21 23 30 32 41 50 65

QuickSort

► Pseudocode -recursive

```
/* low --> Starting index, high --> Ending index */  
quickSort(arr[], low, high)  
{  
    if (low < high)  
    {  
        /* pi is partitioning index, arr[pi] is now  
         at right place */  
        pi = partition(arr, low, high);  
  
        quickSort(arr, low, pi - 1); // Before pi  
        quickSort(arr, pi + 1, high); // After pi  
    }  
}
```

QuickSort- The Partition Algorithm

- ▶ /* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */

```
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1) // Index of smaller element and indicates the
                  // right position of pivot found so far

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```