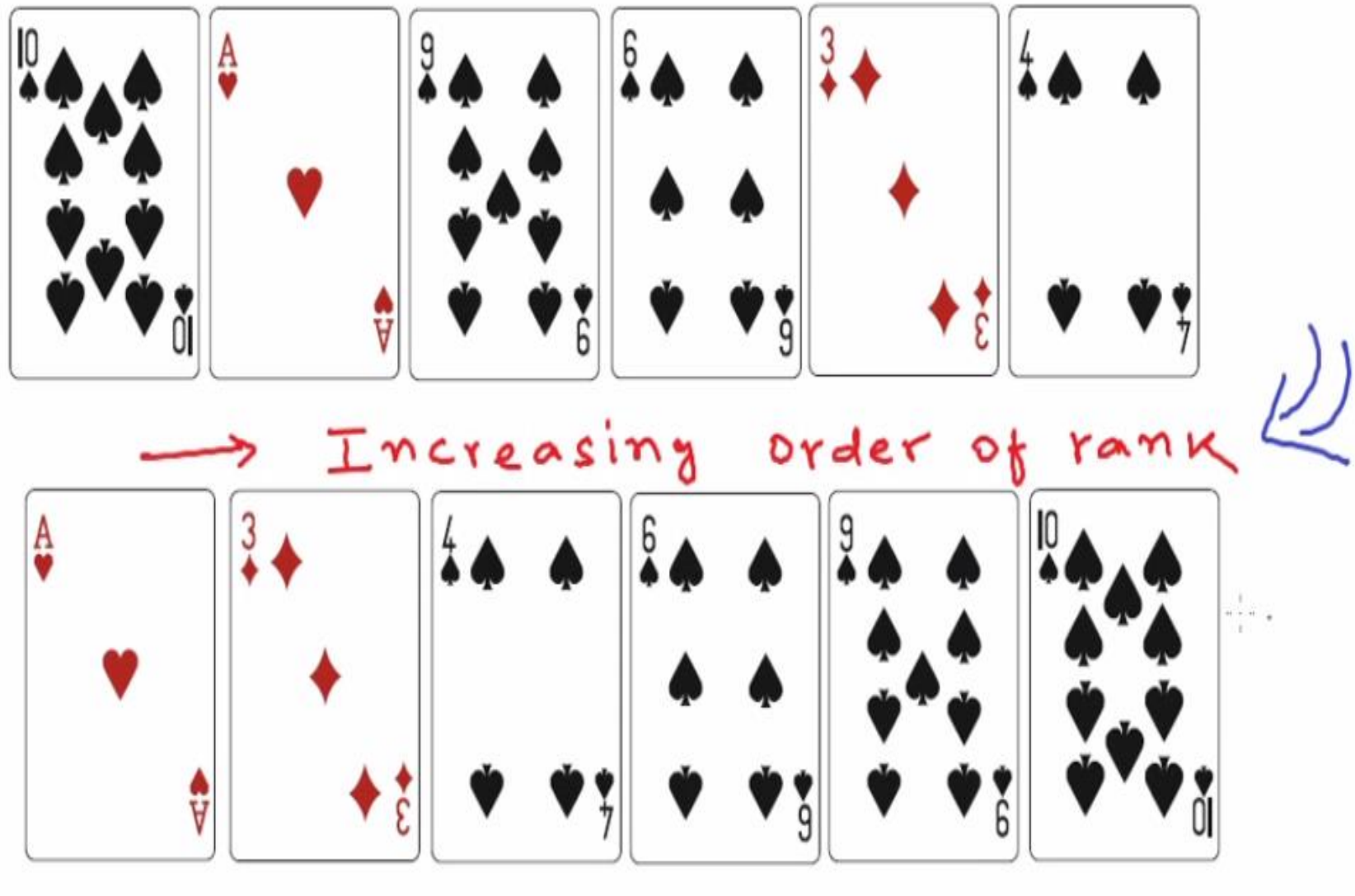
Data structures and Algorithms

Sorting

Sorting



Sorting

- **Sorting** is a process that organizes or rearranges a collection of data into either ascending or descending order.

- Sorting sometimes is done to improve on the readability of that data, at times to be able to search or extract some information quickly out of that data.

- The list should be homogeneous, that is the elements in the list should be the same type.

- To study sorting algorithms, most of the times we use a list of integers and typically we sort the list of integers in an increasing order of value.
  - E.g. 5,9, 8, 1, 3, 2, 4 → 1,2,3,4,5,8,9 increasing order
    → 9,8,5,4,3, 2,1 decreasing order
  - 'fork', 'knife', 'mouse', 'screen', 'key' → 'fork', 'key', 'knife', 'mouse', 'screen'

Sorting

- There are a number of areas where we would like to keep our data sorted
  - Words in the dictionary
  - Telephone numbers
  - Students records in class for easy access of any information required
  - A model tally where we are able to see which team is at the top and which team at the bottom.
- In this lecture we are going to study, analyze and compare different sorting algorithms

Sorting

- Some of the sorting algorithms that we will be talking about, that we will be analyzing about are Bubble sort, Selection sort, Insertion sort, Merge Sort, Quick sort etc.
- We have so many algorithms for sorting that have been designed over a long period of time.
- We often classify sorting algorithms based on some parameters as follows:
  - **Time complexity** – which is the measure of rate of growth of time taken by an algorithm with respect to input size. Some algorithms will be relatively faster than the others.

Sorting

- **Space Complexity or memory usage-** some sorting algorithms are in place, they use constant amount of extra memory to rearrange the elements in the list, while some sorting algorithms like merge sort, use extra memory to temporarily store data and the memory usage grows with input size

- **Stability**- sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

Sorting

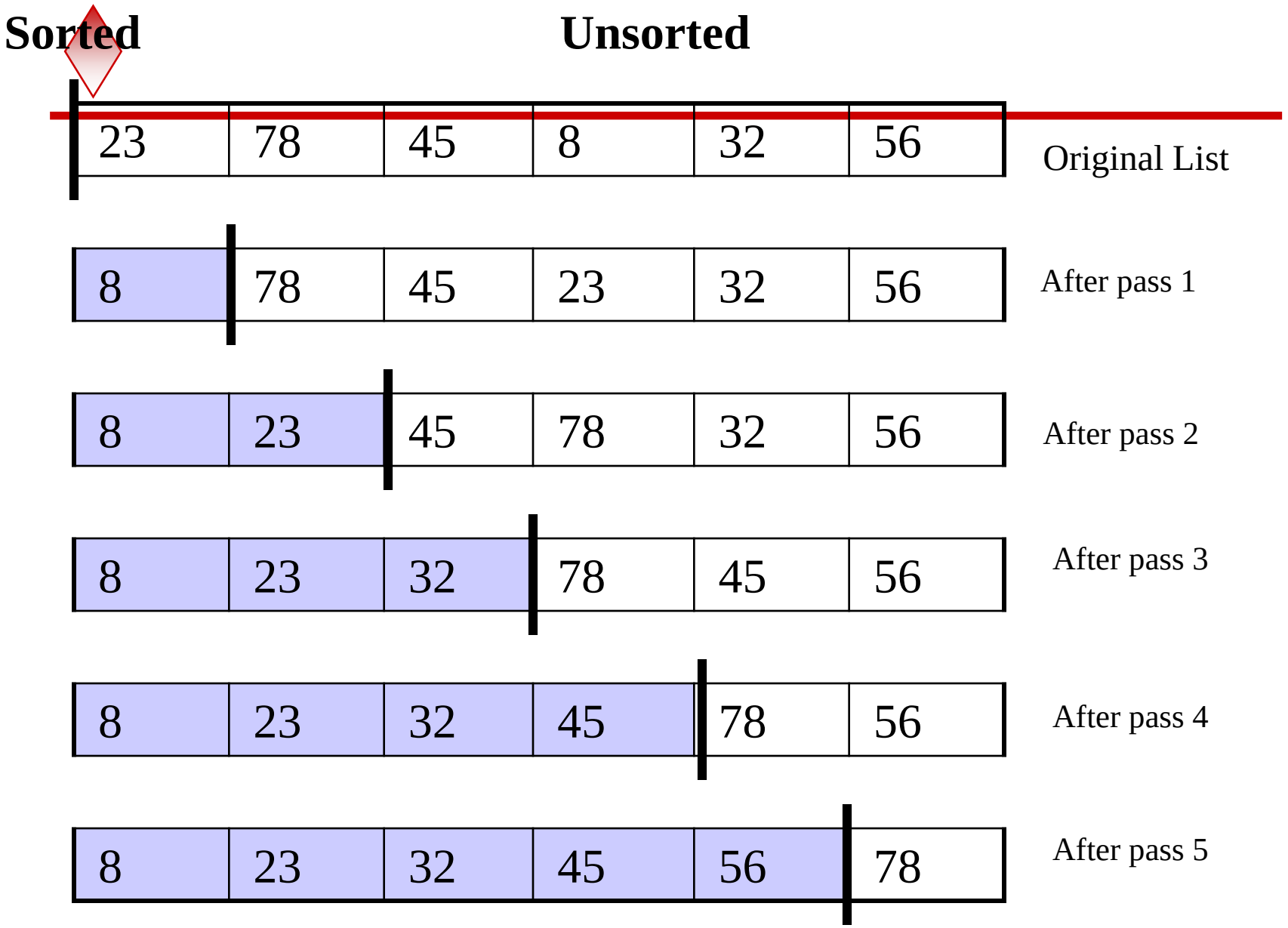
- **Internal sort or External sort** –
  - when all the records that need to be sorted are in the memory or RAM, such sort is internal sort
  - If the records are on auxiliary storage like disk and tapes, quite often it is not possible to get all of them in the main memory in one go, then we call such a sort external sort.
- **Recursive or non-recursive**
  - Some sorting algorithms like quick sort and merge sort are recursive while other like insertion sort and selection sort are non-recursive

Sorting

- Any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.
- Sorting also has **indirect uses**. An initial sort of the data can significantly enhance the performance of an algorithm.
- Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.

Selection Sort

- The list is divided into two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.
- We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.
- A list of *n* elements requires *n-1* passes to completely rearrange the data.


Sorted
Unsorted
23 78 45 8 32 56
Original List
8 78 45 23 32 56
After pass 1
8 23 45 78 32 56
After pass 2
8 23 32 78 45 56
After pass 3
8 23 32 45 78 56
After pass 4
8 23 32 45 56 78
After pass 5

Selection Sort Algorithm

- Step 1 – Set MIN to location 0
- Step 2 – Search the minimum element in the list
- Step 3 – Swap with value at location MIN
- Step 4 – Increment MIN to point to next element
- Step 5 – Repeat until list is sorted

Selection Sort

```
SelectionSort ( A, n){
  for (i ← 0   to n - 1) {
        imin ← i
        for(j ←i+1 to n-1 )
            {
             if(A[j]<A[imin]
               imin ←j
         swap(A[i], A[imin]);
              }
            temp ← A[i]
            A[i] ← A[imin]
            A[imin] ← temp
       }
     }
```

Selection Sort -- Analysis

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).
  - So, to analyze a sorting algorithm we should count the number of key comparisons and the number of moves.
  - Ignoring other operations does not affect our final result.
- In selectionSort function, the outer for loop executes n-1 times.
- We invoke swap function once at each iteration.
  - Total Swaps: n-1
  - Total Moves: 3*(n-1)(Each swap has three moves)

Selection Sort – Analysis (cont.)

- The inner for loop executes the size of the unsorted part minus 1 (from 1 to n-1), and in each iteration we make one key comparison.
  - ➔ # of key comparisons = 1+2+...+n-1 = n*(n-1)/2
  - ➔ **So, Selection sort is $O(n^2)$**
- The best case, the worst case, and the average case of the selection sort algorithm are same. ➔ all of them are **$O(n^2)$**
  - This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.
  - Since $O(n^2)$ grows so rapidly, the selection sort algorithm is appropriate only for small n.
  - Although the selection sort algorithm requires $O(n^2)$ key comparisons, it only requires $O(n)$ moves.
  - A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).

Comparison of *N*, *logN* and N^2

N	O(LogN)	$O(N^2)$
16	4	256
64		6 4K
256	8	64K
1,024	10	1M
16,384	14	256M
131,072	17	16G
262,144	18	6.87E+10
524,288	19	2.74E+11
1,048,576	20	1.09E+12
1,073,741,824		30 1.15E+18

Insertion Sort

- Insertion sort is slower than quick sort, but not as slow as bubble sort, and it is easy to understand.

- Insertion sort works the same way as arranging your hand when playing cards.
  - Out of the pile of unsorted cards that were dealt to you, you pick up a card and place it in your hand in the correct position relative to the cards you're already holding.

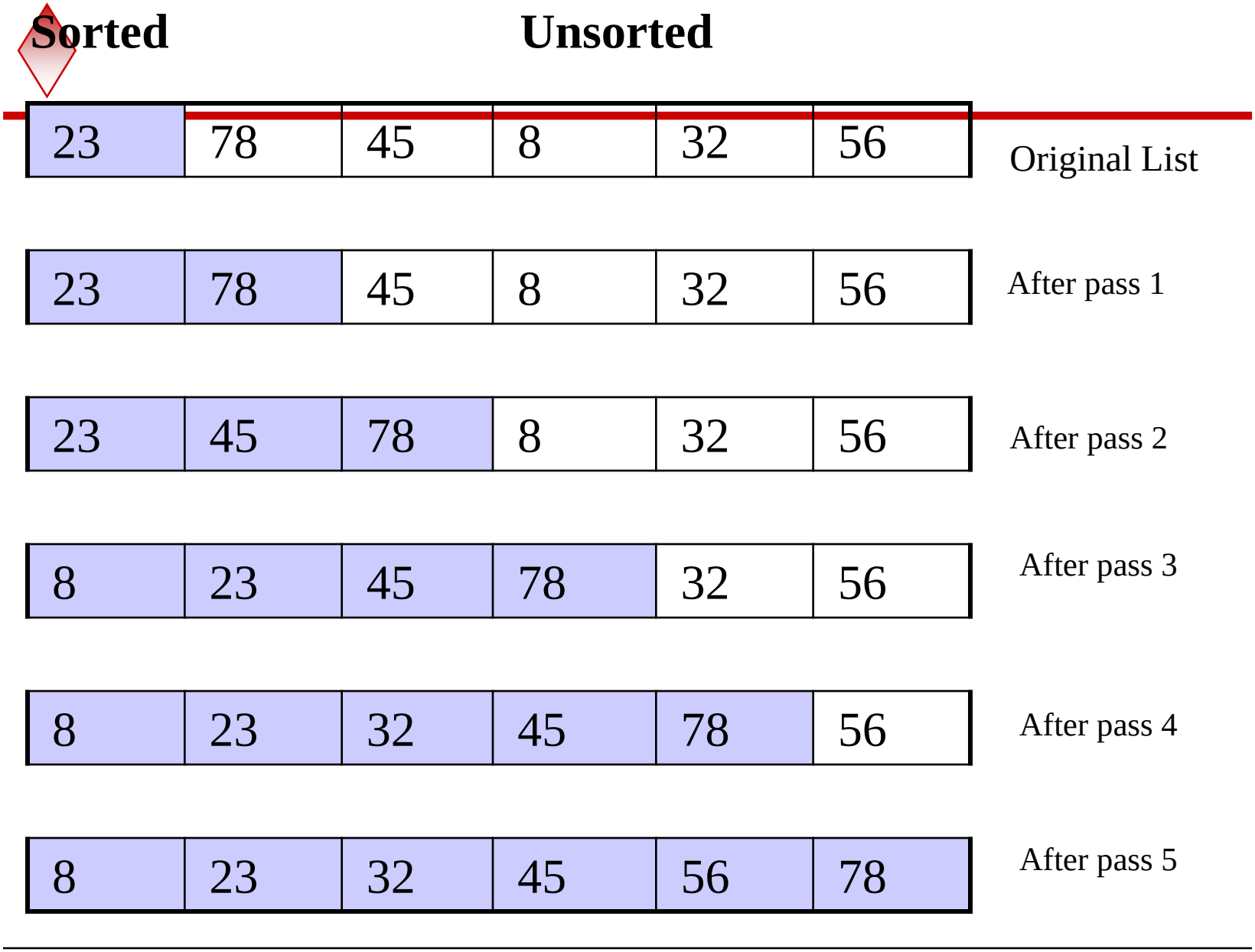
Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
  - Most common sorting technique used by card players.
- The list is divided into two parts: sorted and unsorted.

Insertion Sort

- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.

- *A* list of *n* elements will take at most *n-1* passes to sort the data.


Sorted
Unsorted
23 78 45 8 32 56
Original List
23 78 45 8 32 56
After pass 1
23 45 78 8 32 56
After pass 2
8 23 45 78 32 56
After pass 3
8 23 32 45 78 56
After pass 4
8 23 32 45 56 78
After pass 5

Insertion Sort Algorithm

Step 1 - If it is the first element, it is already sorted. return 1;

Step 2 - Pick next element

Step 3 - Compare with all elements in the sorted sub-list

Step 4 - Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 - Insert the value

Step 6 - Repeat until list is sorted

Insertion Sort Pseudocode

```
InsertionSort ( A, n){
  for (i ← 1   to n – 1) {
        value ← A[i]
        hole ← i
        while ( hole > 0 && A[hole-1] < value   )
              {
              A[hole] ← A[hole-1]
               hole←hole-1
              }
              A[hole] ← value
          }
      }
```

Insertion Sort Algorithm

```
template <class Item>
void insertionSort(Item a[], int n)
{
   for (int i = 1; i < n; i++)
   {
      Item tmp = a[i];

      for (int j=i; j>0 && tmp < a[j-1]; j--)
         a[j] = a[j-1];
      a[j] = tmp;
   }
}
```

Insertion Sort – Analysis

- Running time depends on not only the size of the array but also the contents of the array.
- ***Best-case:*** ➔ **$O(n)$**
  - Array is already sorted in ascending order.
  - Inner loop will not be executed.
  - The number of key comparisons: (n-1) ➔ $O(n)$
- ***Worst-case:*** ➔ **$O(n^2)$**
  - Array is in reverse order:
  - Inner loop is executed i-1 times, for i = 2,3, …, n
  - The number of key comparisons: (1+2+…+n-1)= n*(n-1)/2 ➔ $O(n^2)$
- Average-case: ➔ **$O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Insertion Sort is $O(n^2)$**

Bubble Sort

- The **bubble sort** makes multiple passes through a list.
- It compares adjacent items and exchanges those that are out of order.
- Each pass through the list places the next largest value in its proper place.
- In essence, each item "bubbles" up to the location where it belongs.

Bubble Sort

23	78	45	8	32	56	Original List
23	78	45	8	32	56	Compare 1st and 2nd (Do notswap)
23	45	78	8	32	56	Compare 2nd and 3rd (swap)
23	45	8	78	32	56	Compare 3rd and 4th (swap)
23	45	8	32	78	56	Compare 4th and 5th (swap)
23	45	8	32	56	78	Compare 5th and 6th (swap)

Bubble Sort Algorithm

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Bubble Sort Algorithm

```
BubbleSort ( A, n){
   for (k ← 1   to n - 1) {
          for ( i ← 0   to n - 2) {
               if (A[i] > A[i + 1]) {
                    swap(A[i], A[i + 1])
                     }
             }
        }
}
```

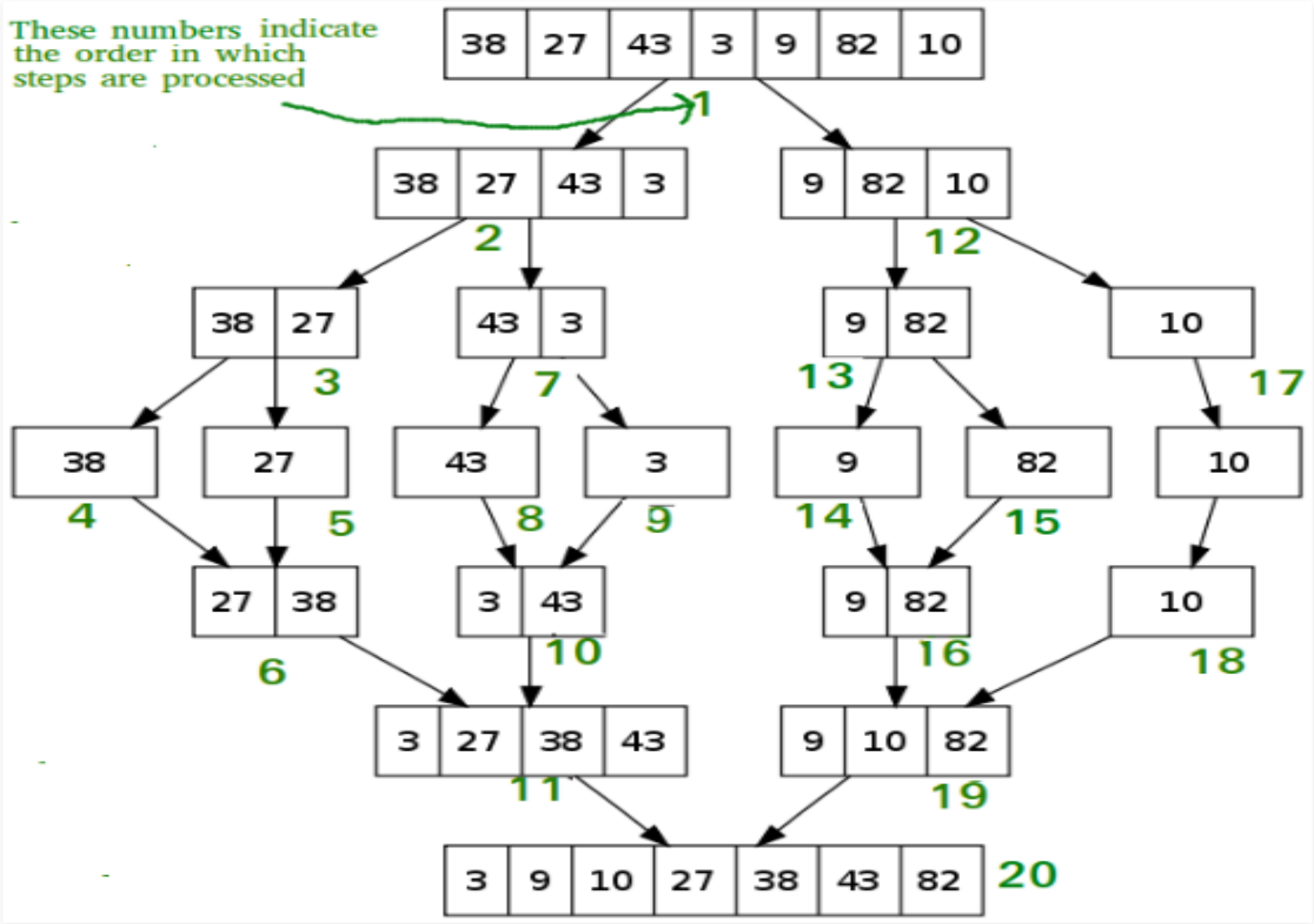
Bubble Sort – Analysis

- ***Best-case:*** ➔ ***O(n)***
  - Array is already sorted in ascending order.
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** ➔ ***O(n^2)***
  - Array is in reverse order:
  - Outer loop is executed n-1 times,
  - The number of key comparisons: (n-1)+(n-2)+(n-3)+.....+3+2+1
  - Sum = n(n-1)/2 i.e. O(n^2) ➔ O(n^2)
- Average-case: ➔ ***O(n^2)***
  - We have to look at all possible initial data organizations.
- **So, Bubble Sort is O(n^2)**

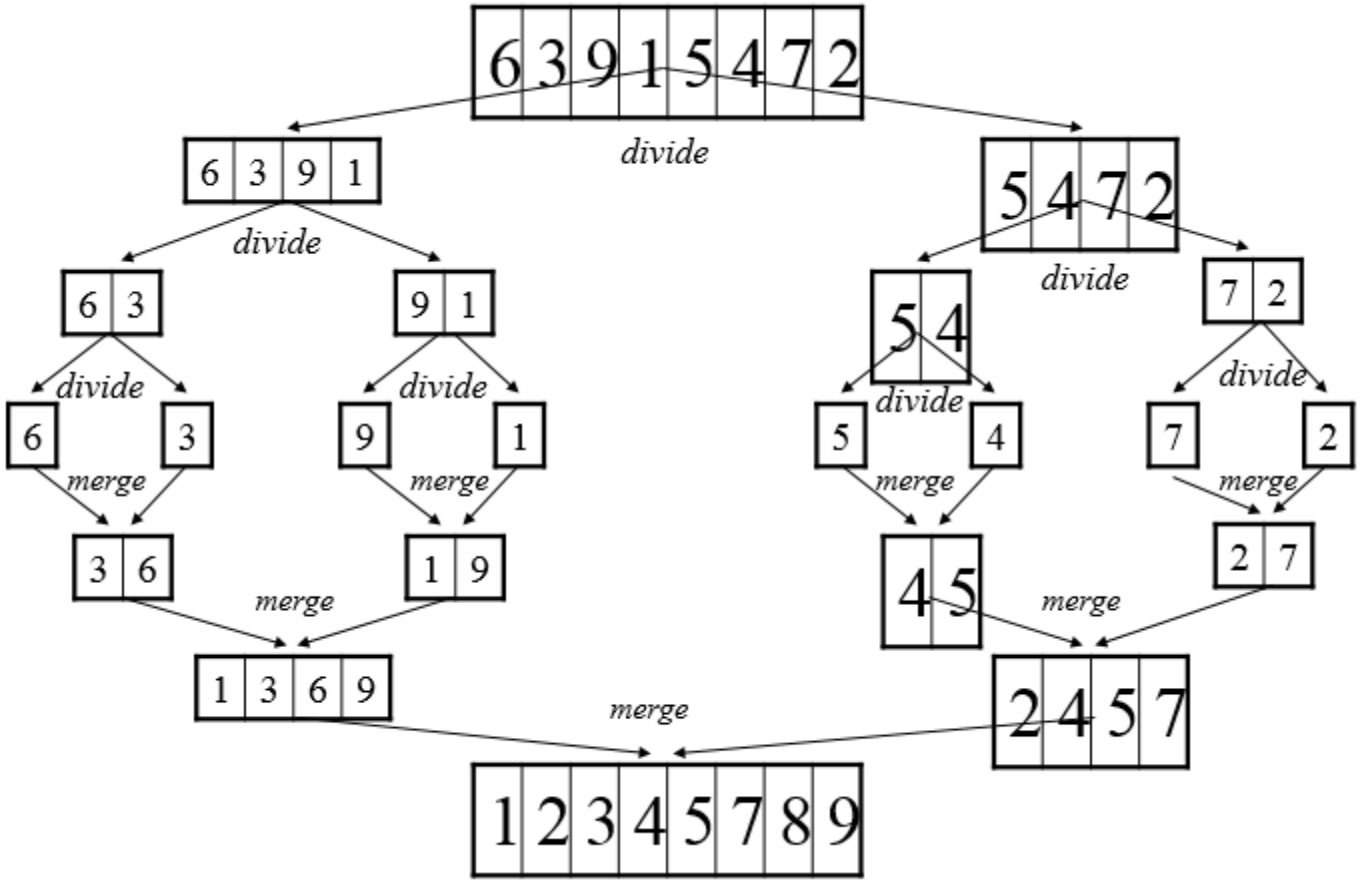
Mergesort

- Mergesort algorithm is one of two important divide-and-conquer sorting algorithms (the other one is quicksort).
- It is a recursive algorithm.
  - Divides the list into halves,
  - Sort each halve separately, and
  - Then merge the sorted halves into one sorted array.

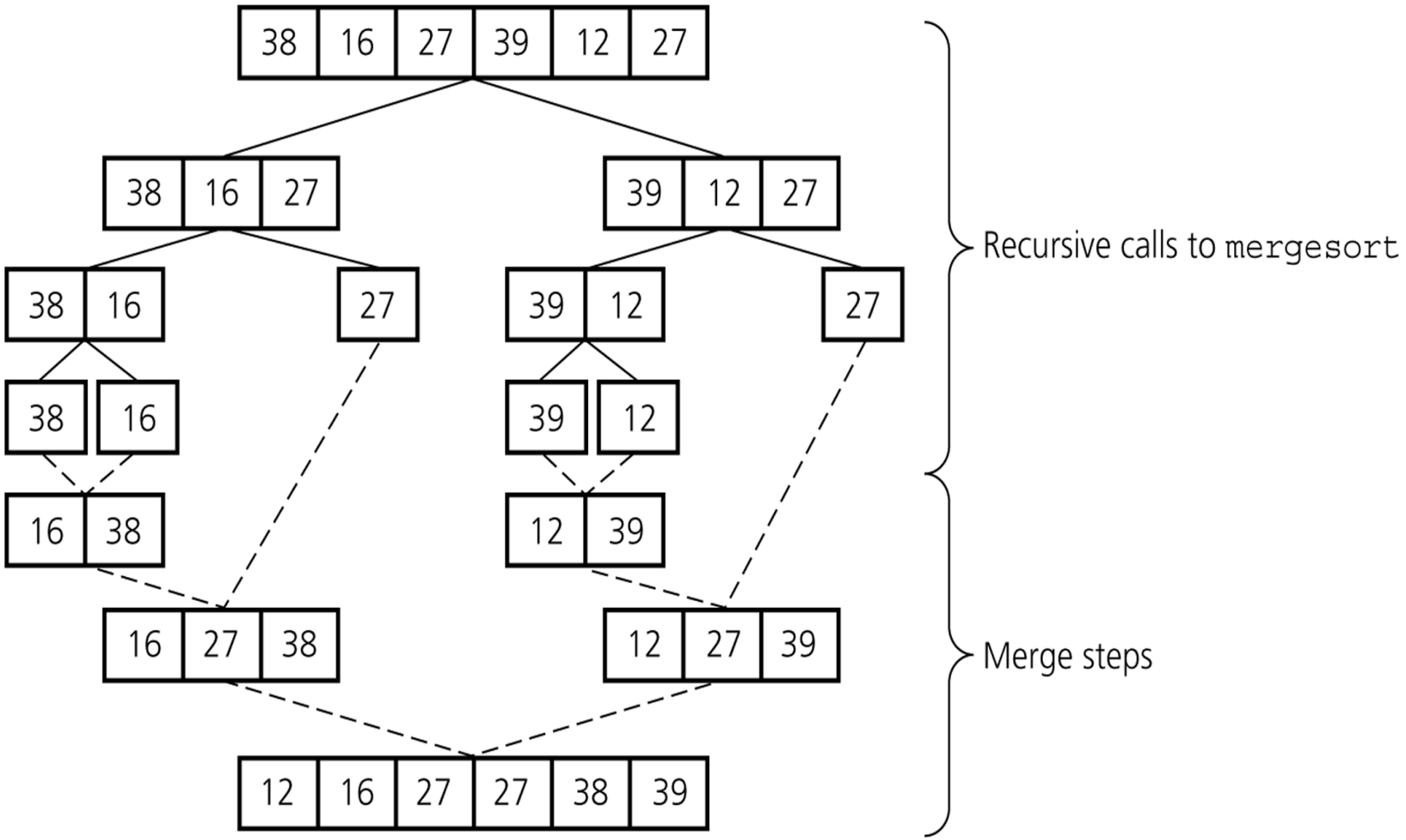
Merge sort – Example 1



Merge Sort Example 2



Merge Sort Example 3



Merge Sort Algorithm

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

   middle m = (l+r)/2

2. Call mergeSort for first half:

   Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

   Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

   Call merge(arr, l, m, r)

Mergesort Pseudocode

```
void mergesort(DataType theArray[], int first, int
   last) {
    if (first < last) {
       int mid = (first + last)/2;        //   index   of
   midpoint
       mergesort(theArray, first, mid);
       mergesort(theArray, mid+1, last);

       // merge the two halves
       merge(theArray, first, mid, last);
    }
}  // end mergesort
```

Mergesort – Analysis

- Mergesort is extremely efficient algorithm with respect to time.
  - Both worst case and average cases are **O (n * $\log_2 n$)**
- But, mergesort requires an extra array whose size equals to the size of the original array.
- If we use a linked list, we do not need an extra array
  - But, we need space for the links
  - And, it will be difficult to divide the list into half (O(n))

Quicksort

- **QuickSort** is a Divide and Conquer algorithm.
- It picks an element as pivot and partitions the given array around the picked pivot.
- There are many different versions of quickSort that pick pivot in different ways.
  1. Always pick first element as pivot.
  2. Always pick last element as pivot (implemented below)
  3. Pick a random element as pivot.
  4. Pick median as pivot.

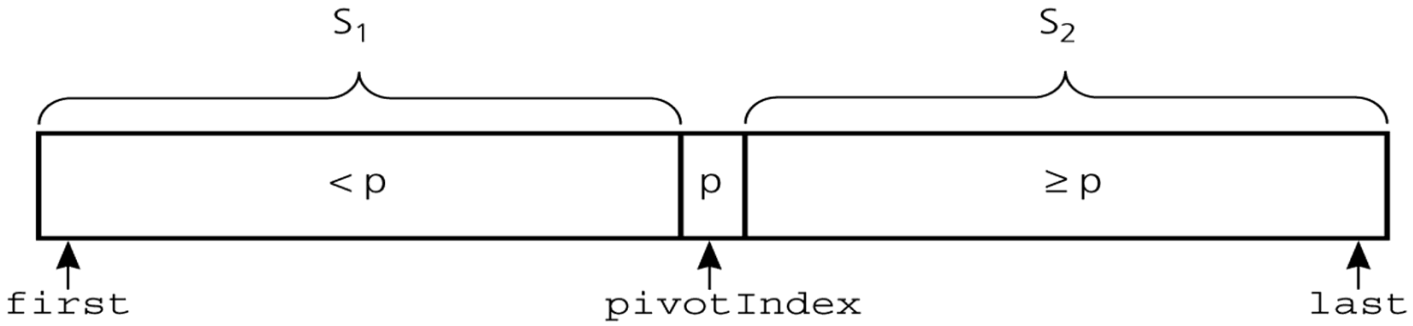
Quicksort (cont.)

The quick-sort algorithm consists of the following three steps:

1. ***Divide***: Partition the list.
   - To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the ***pivot***.
     - **(Always pick last element as pivot).**
   - Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater values come in another.
2. ***Recursion***: Recursively sort the sublists separately.
3. ***Conquer***: Put the sorted sublists together by a simple concatenation.

Partition

- Partitioning places the pivot in its correct place position within the array.

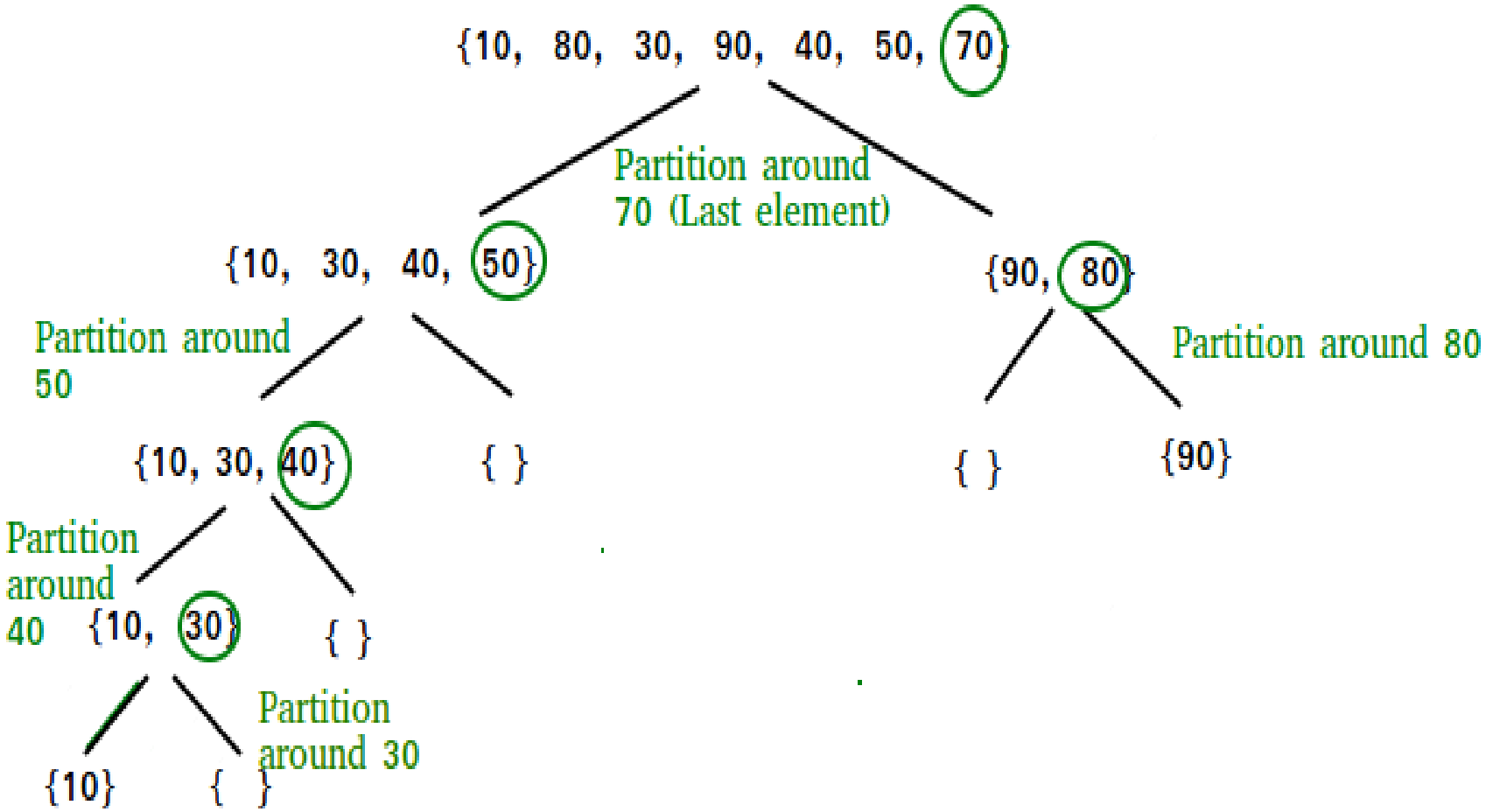


- Arranging the array elements around the pivot p generates two smaller sorting problems.
  - sort the left section of the array, and sort the right section of the array.
  - when these two smaller sorting problems are solved recursively, our bigger sorting problem is solved.

Partition – Choosing the pivot

- First, we have to select a pivot element among the elements of the given array, and we put this pivot into the first location of the array before partitioning.
- Which array item should be selected as pivot?
  - Somehow we have to select a pivot, and we hope that we will get a good partitioning.
  - If the items in the array arranged randomly, we choose a pivot randomly.
  - We can choose the first or last element as a pivot (it may not give a good partitioning).
  - We can use different techniques to select the pivot.

Partition Function (cont.)



Partition Function

```
Partition (A, start, end)
   {
      pivot ← A[end]
      pindex ← start
      for i ← start to end – 1
          {
           if (A[i] ≤ pivot)
             {
               swap( A[i], A[pindex])
               pindex ← pindex + 1
             }
           } swap (A[pindex], A[end])
             return pindex;
   }
```

Quick Sort Algorithm

- Step 1 – Choose the highest index value as pivot
- Step 2 – Take two variables to point left and right of the list excluding pivot
- Step 3 – left points to the low index
- Step 4 – right points to the high
- Step 5 – while value at left is less than pivot move right
- Step 6 – while value at right is greater than pivot move left
- Step 7 – if both step 5 and step 6 does not match swap left and right
- Step 8 – if left ≥ right, the point where they met is new pivot

Quicksort Function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] 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 – Analysis

Worst Case: (assume that we are selecting the first element as pivot)

- The pivot divides the list of size n into two sublists of sizes 0 and n-1.
- The number of key comparisons
  = n-1 + n-2 + ... + 1
  = $\mathbf{n^2/2 - n/2}$ ➔ $\mathbf{O(n^2)}$
- The number of swaps =
  = n-1 + n-1 + n-2 + ... + 1

  swaps outside of the for loop swaps inside of the for loop
  = $\mathbf{n^2/2 + n/2 - 1}$ ➔ $\mathbf{O(n^2)}$

❑ So, Quicksort is $\mathbf{O(n^2)}$ in worst case

Quicksort – Analysis

- Quicksort is **$O(n*\log_2 n)$** in the best case and average case.
- Quicksort is slow when the array is sorted and we choose the first element as the pivot.
- Although the worst case behavior is not so good, its average case behavior is much better than its worst case.
  - So, Quicksort is one of best sorting algorithms using key comparisons.

Comparison of Sorting Algorithms

Algorithms	Best Case	Average Case	Worst Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$

Comparison of Sorting Algorithms

Algorithms	Auxiliary Space	Sorting In Place	Stability	
Selection Sort	O(1)	Yes	No	
Bubble Sort	O(1)	Yes	Yes	
Insertion Sort	O(1)	Yes	Yes	
Merge Sort	O(n)	No	Yes	
Quick Sort	O(1)	Yes	Yes	