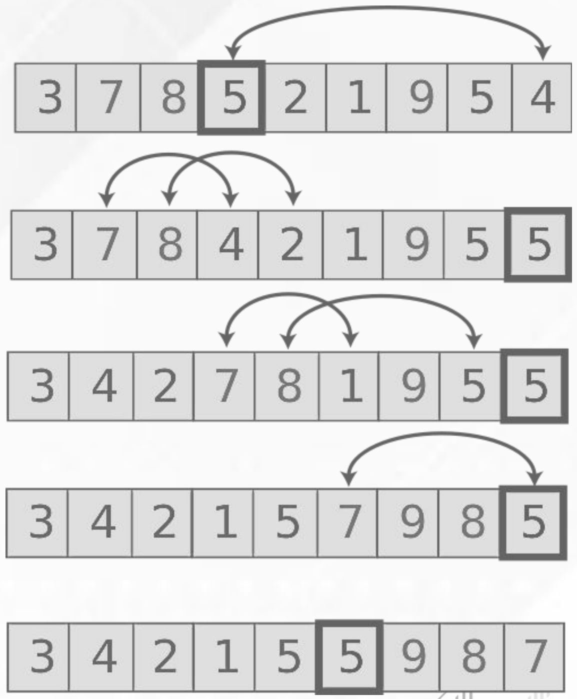
Unit-5

Searching & Sorting



Sorting & Searching

- Searching
  - Linear/Sequential Search
  - Binary Search
- Sorting
  - Bubble sort
  - Selection Sort
  - Insertion Sort
  - Quick Sort
  - Merge Sort

Linear/Sequential Search

- In computer science, **linear search** or **sequential search** is a method for finding a particular value in a list that consists of **checking every** one of its **elements**, **one at a time** and in sequence, **until the desired one is found**.
- Linear search is the simplest search algorithm.
- It is a special case of brute-force search.
- Its **worst case cost** is proportional to the **number of elements in the list**.

Sequential Search – Algorithm & Example

```
# Input: Array A, integer key
# Output: first index of key in A
# or -1 if not found

Algorithm: Linear_Search
for i = 0 to last index of A:
     if A[i] equals key:
          return i

return -1
```

Search for 1 in given array

2	9	3	1	8

Comparing value of i^{th} index with element to be search one by one until we get searched element or end of the array

Step 1: i=0

2	9	3	1	8

i

Step 1: i=1

2	9	3	1	8

i

Step 1: i=2

2	9	3	1	8

i

Step 1: i=3

2	9	3	1	8

i

Element found at i^{th} index, i=3

Binary Search

- If we have an **array** that is **sorted**, we can use a much more **efficient algorithm** called a **Binary Search**.
- In binary search **each time** we **divide array** into **two equal half** and **compare middle element** with **search element**.
- Searching Logic
  - If **middle element** is **equal to search element** then we got that element and **return that index**
  - if **middle element** is **less than search element** we **look right part** of array
  - if **middle element** is **greater than search element** we look **left part** of array.

Binary Search - Algorithm

```
# Input: Sorted Array A, integer key
# Output: first index of key in A,
# or -1 if not found
Algorithm: Binary_Search (A, left, right)
left = 0, right = n-1
while left < right
   middle = index halfway between left, right
   if A[middle] matches key
      return middle
   else if key less than A[middle]
      right = middle -1
   else
      left = middle + 1
return -1
```

Binary Search - Algorithm

Search for **6** in given array

	-1	5	6	18	19	25	46	78	102	114
Index	0	1	2	3	4	5	6	7	8	9
	left									right

Key=6, No of Elements = 10, so left = 0, right=9

Step 1:

middle index = (left + right) /2 = (0+9)/2 **= 4**

middle element value = a[4] = **19**

Key=6 is **less than** middle element = **19**, so **right** = middle − 1 = 4 − 1 = **3, left = 0**

	-1	5	6	18	19	25	46	78	102	114
Index	0	1	2	3	4	5	6	7	8	9
	left			right						

Binary Search - Algorithm

Step 2:

middle index = (left + right) /2 = (0+3)/2 **= 1**

middle element value = a[1] = **5**

Key=6 is **greater than** middle element = **5**, so **left** = middle + 1 =1 + 1 = **2, right = 3**

	-1	5	6	18	19	25	46	78	102	114
Index	0	1	2	3	4	5	6	7	8	9
			left	right						

Step 3:

middle index = (left + right) /2 = (2+3)/2 **= 2**

middle element value = a[2] = **6**

Key=6 is **equals to** middle element = **6**, so **element found**

	-1	5	6	18	19	25	46	78	102	114
Index	0	1	2	3	4	5	6	7	8	9
			Element Found							

Selection Sort

- Selection sort is a simple sorting algorithm.
- The **list** is **divided into two parts**,
  - The **sorted part** at the **left end** and
  - The **unsorted part** at the **right end**.
  - Initially, the sorted part is empty and the unsorted part is the entire list.
- The **smallest element** is **selected** from the **unsorted array** and **swapped** with the **leftmost element**, and that element becomes a part of the sorted array.
- This process continues moving unsorted array boundary by one element to the right.
- This algorithm is **not suitable** for **large data sets** as its average and worst case complexities are of **$O(n^2)$**, where n is the number of items.

Selection Sort

Unsorted Array

5	1	12	-5	16	2	12	14

Step 1 :

Unsorted Array

5	1	12	-5	16	2	12	14
0	1	2	3	4	5	6	7

Step 2 :

Unsorted Array (elements 0 to 7)

	1	12	5	16	2	12	14
0	1	2	3	4	5	6	7

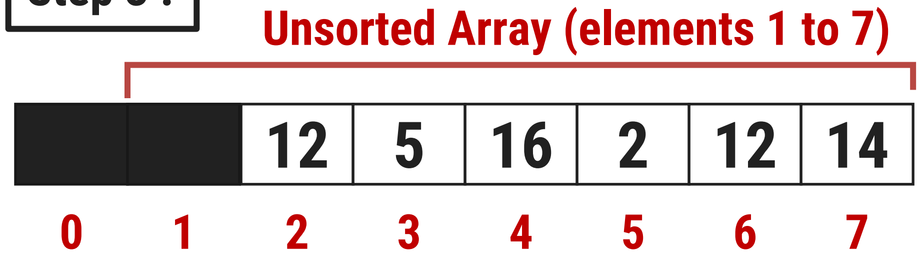
Swap

Min index = 0, value = 5

Find min value from
Unsorted array
Index = 3, value = -5

Selection Sort

Step 3 :

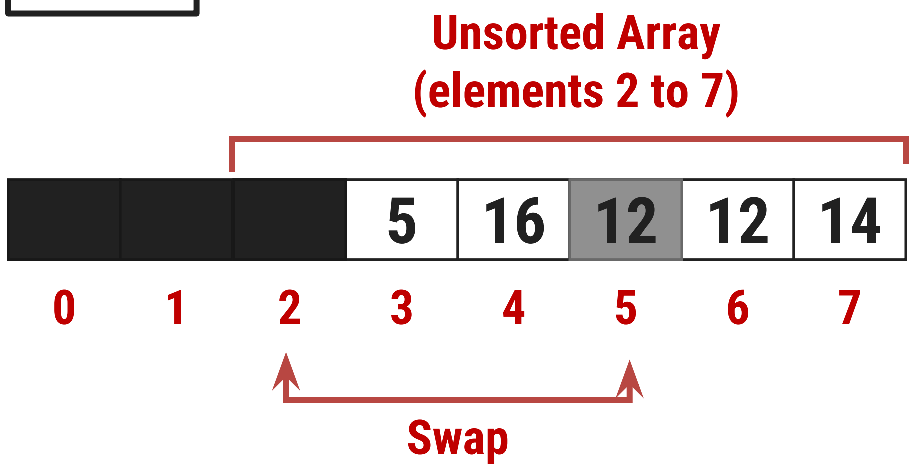


Min index = 1, value = 1

Find min value from
Unsorted array
Index = 1, value = 1

No Swapping as min value is already at right place

Step 4 :

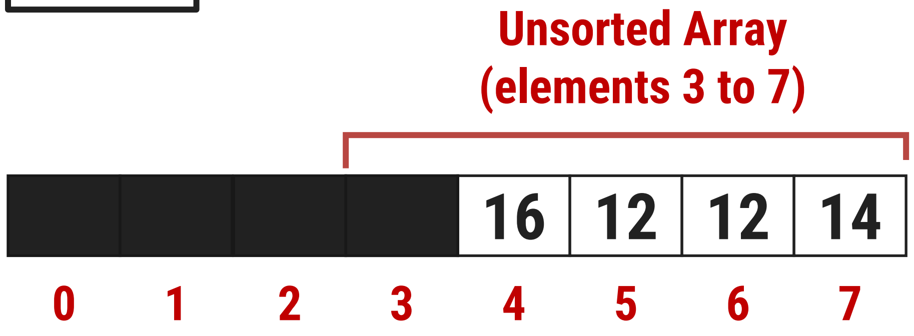


Min index = 2, value = 12

Find min value from
Unsorted array
Index = 5, value = 2

Selection Sort

Step 5 :

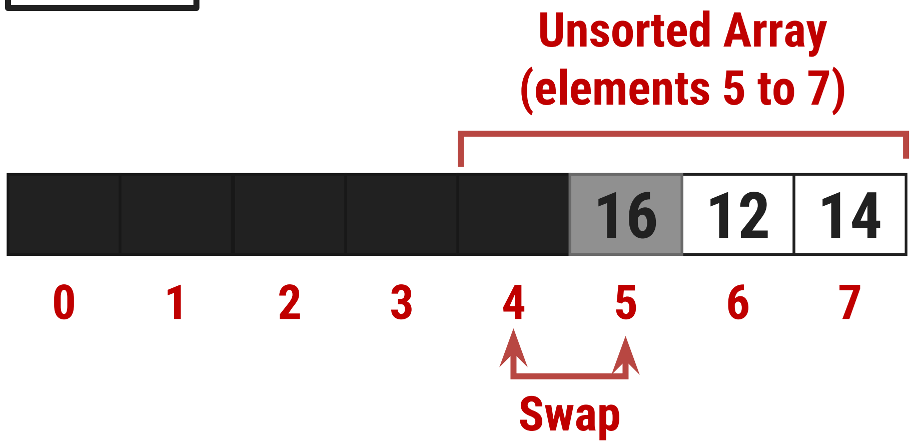


Min index = 3, value = 5

Find min value from
Unsorted array
Index = 3, value = 5

No Swapping as min value is already at right place

Step 6 :

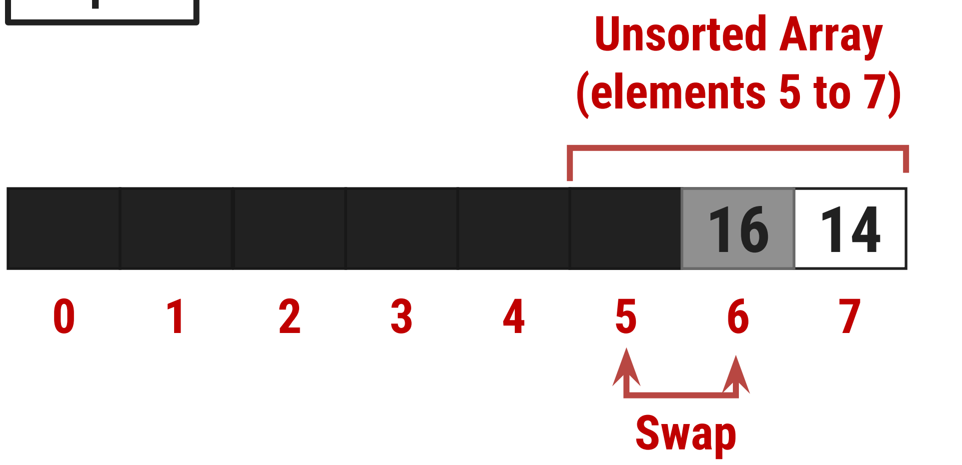


Min index = 4, value = 16

Find min value from
Unsorted array
Index = 5, value = 12

Selection Sort

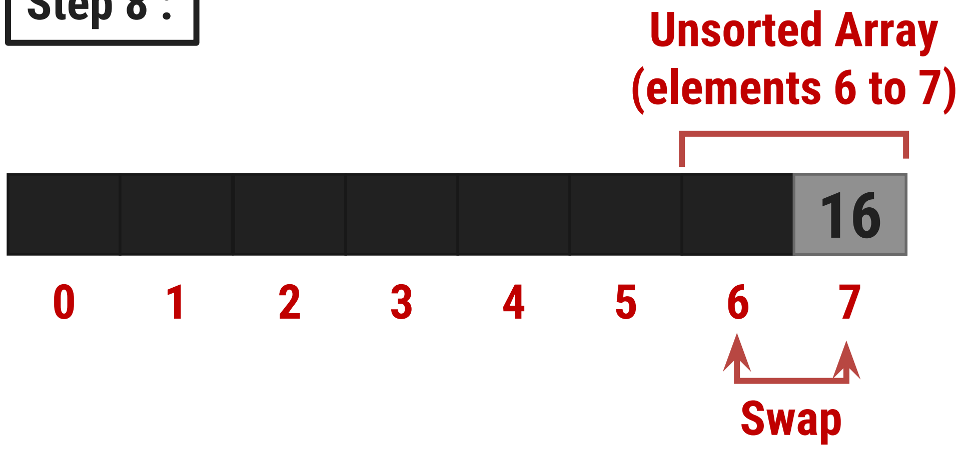
Step 7 :



Min index = 5, value = 16

Find min value from
Unsorted array
Index = 6, value = 12

Step 8 :



Min index = 6, value = 16

Find min value from
Unsorted array
Index = 7, value = 14

SELECTION_SORT(K,N)

- Given a **vector** **K** of **N** elements
- This procedure **rearrange** the **vector** in **ascending order** using **Selection Sort**
- The variable **PASS** denotes the **pass index** and position of the first element in the vector
- The variable **MIN_INDEX** denotes the **position of** the **smallest element** encountered
- The variable **I** is used to index elements

SELECTION_SORT(K,N)

```
1. [Loop on the Pass index]
   Repeat thru step 4 for PASS = 1,2,……..., N-1
2. [Initialize minimum index]
    MIN_INDEX ⍰ PASS
3. [Make a pass and obtain element with smallest value]
   Repeat for I = PASS + 1, PASS + 2, ……………..., N
   If    K[I] < K[MIN_INDEX]
   Then  MIN_INDEX ⍰ I
4. [Exchange elements]
   IF   MIN_INDEX <> PASS
   Then  K[PASS] ⍰⍰    K[MIN_INDEX]
5. [Finished]
   Return
```

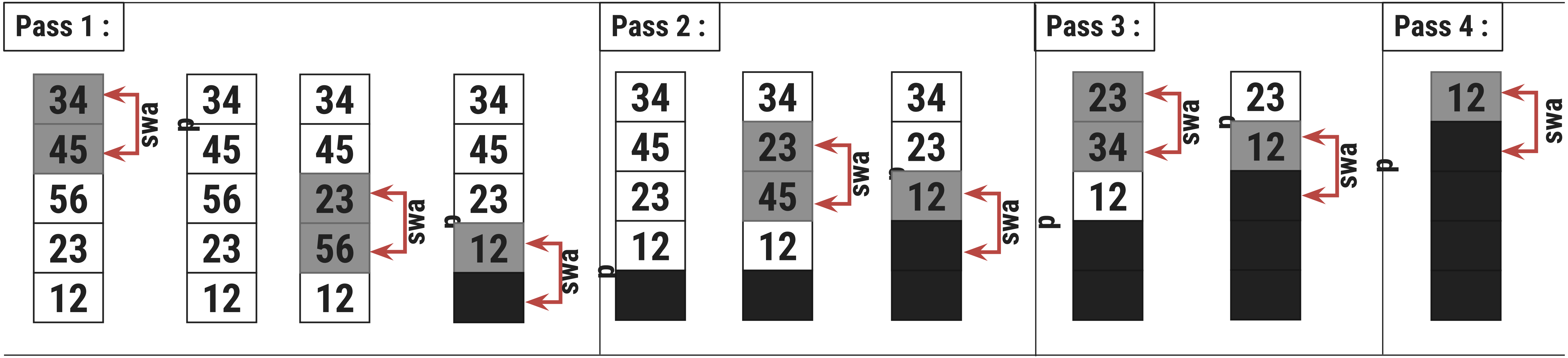
Bubble Sort

- Unlike **selection sort**, **instead of finding the smallest record** and performing the interchange, two records are **interchanged immediately** upon discovering that they are out of order
- During the **first pass R_1 and R_2 are compared** and **interchanged in case of our of order**, this process is repeated for records R_2 and R_3, and so on.
- This method will cause records with **small key to move "bubble up",**
- **After** the **first pass**, the record with **largest key** will be in the n^{th} position.
- On each successive pass, the records with the next largest key will be placed in position n-1, n-2 ….., 2 respectively
- This approached required at most n−1 passes, The **complexity** of bubble sort is $O(n^2)$

Bubble Sort

Unsorted Array

45	34	56	23	12



BUBBLE_SORT(K,N)

- Given a **vector** **K** of **N** elements
- This procedure **rearrange** the **vector** in **ascending order** using **Bubble Sort**
- The variable **PASS & LAST** denotes the **pass index** and position of the first element in the vector
- The variable **EXCHS** is used to count number of exchanges made on any pass
- The variable **I** is used to index elements

Procedure: BUBBLE_SORT (K, N)

```
1. [Initialize]
   LAST  N
2. [Loop on pass index]
   Repeat thru step 5 for PASS = 1, 2, 3, ….  , N-1
3. [Initialize exchange counter for this pass]
   EXCHS  0
4. [Perform pairwise comparisons on unsorted elements]
   Repeat for I = 1, 2, ……….., LAST – 1
      IF  K[I] > K [I+1]
      Then   K[I]  K[I+1]
       EXCHS  EXCHS + 1
5. [Any exchange made in this pass?]
   IF    EXCHS = 0
   Then  Return (Vector is sorted, early return)
   ELSE  LAST  LAST - 1
6. [Finished]
   Return
```

Quick Sort

- **Quick sort** is a highly efficient sorting algorithm and is based on **partitioning of array** of data into **smaller arrays**.
- Quick Sort is **divide and conquer** algorithm.
- At each step of the method, the goal is to place a particular record in its final position within the table,
- In doing so all the records which precedes this record will have smaller keys, while all records that follows it have larger keys.
- This particular records is termed **pivot element**.
- The same process can then be applied to each of these subtables and repeated until all records are placed in their positions

Quick Sort

- There are many **different versions** of Quick Sort **that pick pivot** in different ways.
  - Always pick **first element as pivot**. (in our case we have consider this version).
  - Always pick **last element as pivot** (implemented below)
  - Pick a **random element as pivot**.
  - Pick **median as pivot**.
- Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays.
- This algorithm is quite **efficient for large-sized data sets**
- Its average and **worst case complexity** are of **$O(n^2)$**, where n is the number of items.

Quick Sort

Sort Following Array using Quick Sort Algorithm

We are considering **first element as pivot element**, so **Lower bound** is **First Index** and **Upper bound** is **Last Index**

We need to find our proper position of Pivot element in sorted array and perform same operations recursively for two sub array

Pivot Element

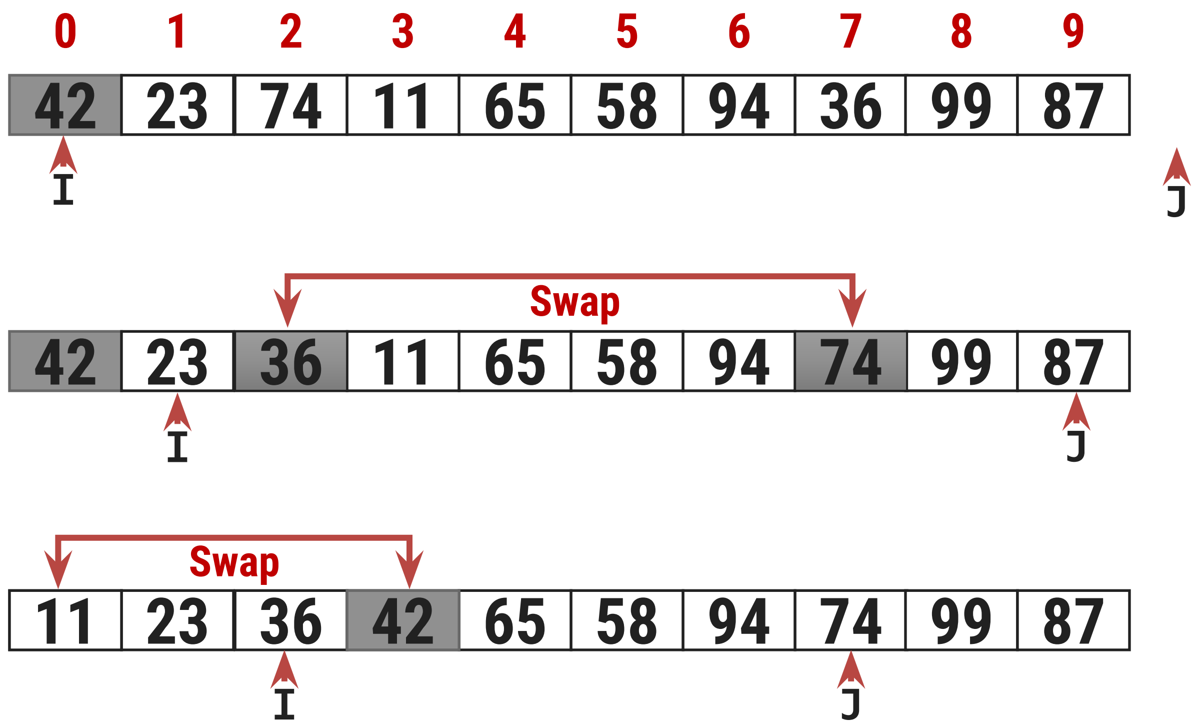
0	1	2	3	4	5	6	7	8	9
42	23	74	11	65	58	94	36	99	87

LB

UB

Quick Sort

```
FLAG ← true
IF    LB < UB
Then
   I ← LB
   J ← UB + 1
   KEY ← K[LB]
   Repeat While FLAG = true
       I ← I+1
       Repeat While K[I] < KEY
          I ← I + 1
       J ← J – 1
       Repeat While K[J] > KEY
          J ← J – 1
       IF    I<J
       Then K[I] ←---→ K[J]
       Else FLAG ← FALSE

  K[LB] ←---→ K[J]
```



LB = 0, UB = 9   KEY = 42

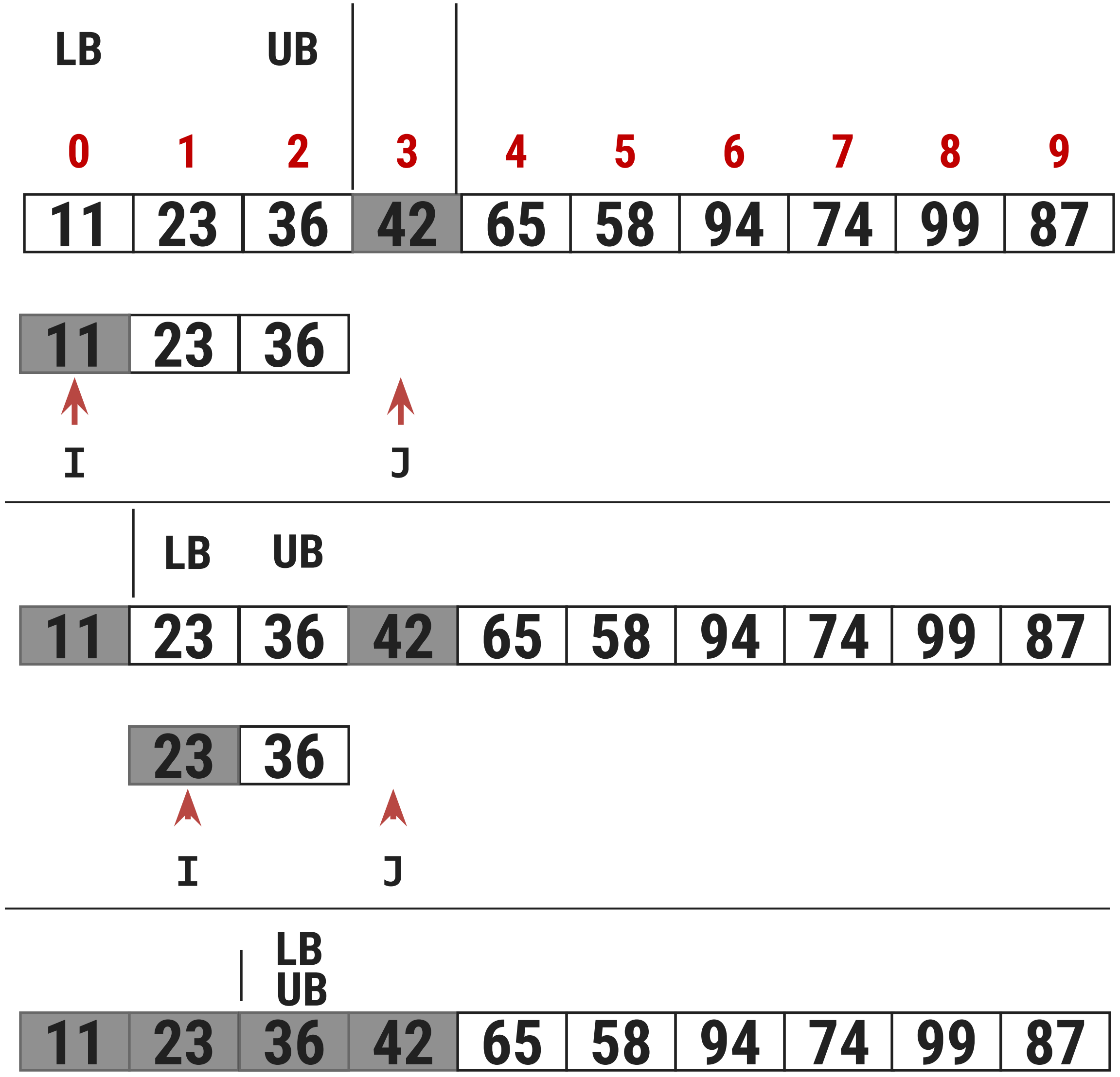
I= 0

J= 10

FLAG= true

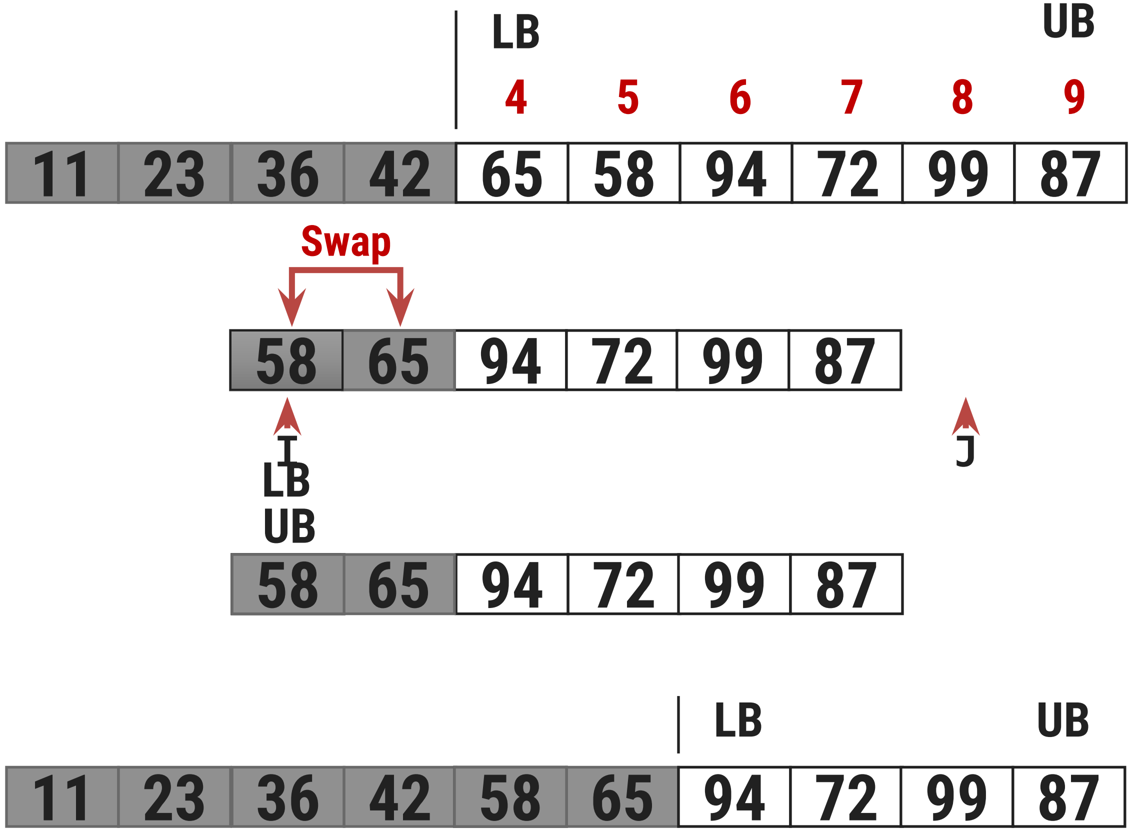
Quick Sort

```
FLAG ⍰ true
IF   LB < UB
Then
  I ⍰ LB
  J ⍰ UB + 1
  KEY ⍰ K[LB]
  Repeat While FLAG = true
     I ⍰ I+1
     Repeat While K[I] < KEY
       I ⍰ I + 1
     J ⍰ J - 1
     Repeat While K[J] > KEY
       J ⍰ J - 1
     IF   I<J
     Then K[I] ⍰---⍰ K[J]
     Else FLAG ⍰ FALSE

   K[LB] ⍰---⍰ K[J]
```



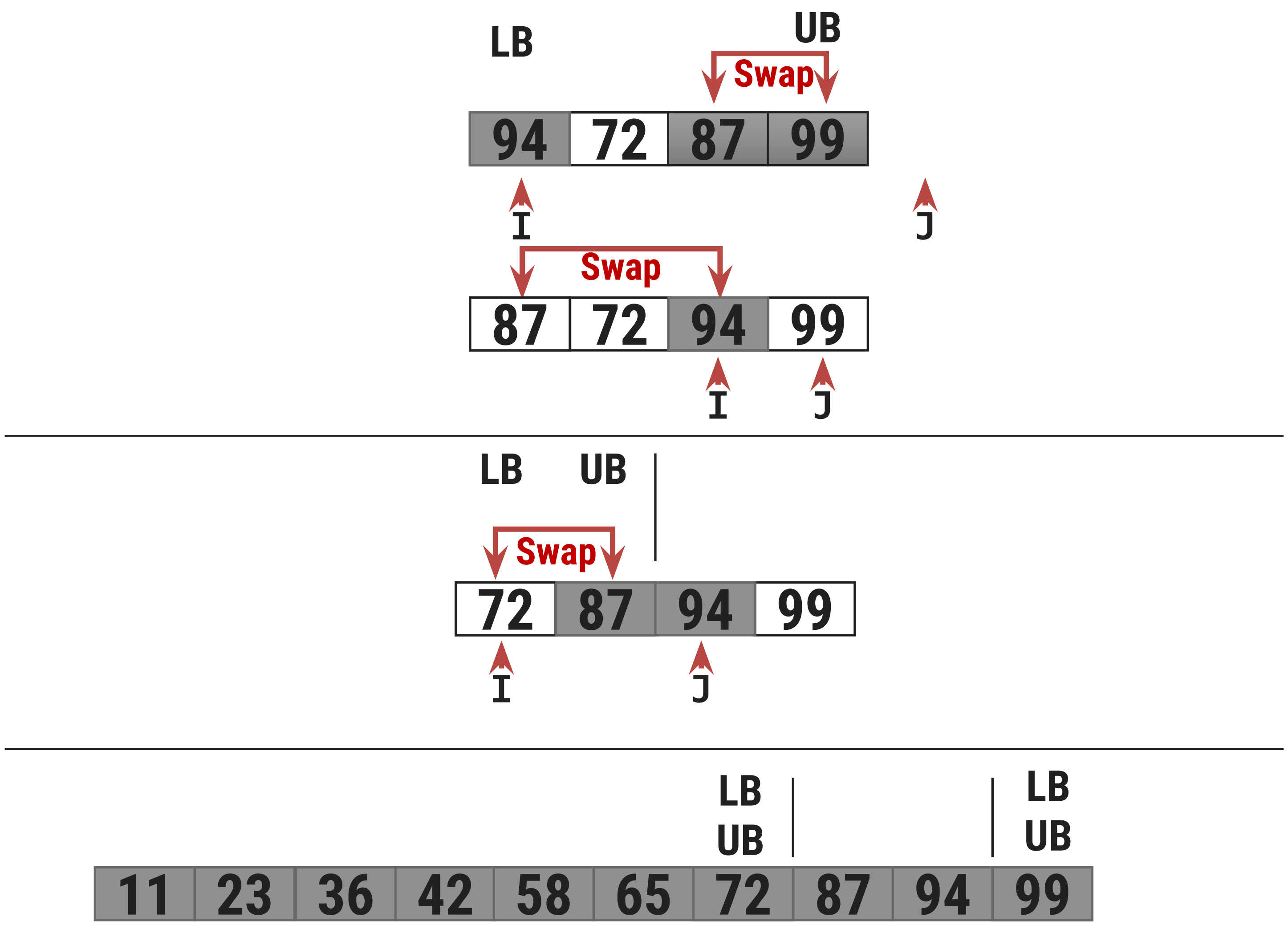
Quick Sort

```
FLAG ⍰ true
IF   LB < UB
Then
  I ⍰ LB
  J ⍰ UB + 1
  KEY ⍰ K[LB]
  Repeat While FLAG = true
     I ⍰ I+1
     Repeat While K[I] < KEY
       I ⍰ I + 1
     J ⍰ J – 1
     Repeat While K[J] > KEY
       J ⍰ J – 1
     IF   I<J
     Then K[I] ⍰---⍰ K[J]
     Else FLAG ⍰ FALSE

   K[LB] ⍰---⍰ K[J]
```



Quick Sort

```
FLAG ← true
IF   LB < UB
Then
  I ← LB
  J ← UB + 1
  KEY ← K[LB]
  Repeat While FLAG = true
     I ← I+1
     Repeat While K[I] < KEY
       I ← I + 1
     J ← J - 1
     Repeat While K[J] > KEY
       J ← J - 1
     IF   I<J
     Then K[I] ←---→ K[J]
     Else FLAG ← FALSE

   K[LB] ←---→ K[J]
```



Algorithm: QUICK_SORT(K,LB,UB)

```
1. [Initialize]
   FLAG ? true
2. [Perform Sort]
   IF   LB < UB
   Then I ? LB
        J ? UB + 1
        KEY ? K[LB]
        Repeat While FLAG = true
          I ? I+1
          Repeat While K[I] < KEY
              I ? I + 1
          J ? J – 1
          Repeat While K[J] > KEY
              J ? J – 1
          IF   I<J
          Then K[I] ?---? K[J]
          Else FLAG ? FALSE

        K[LB] ?---? K[J]
```

```
 CALL QUICK_SORT(K,LB, J-1)
 CALL QUICK_SORT(K,J+1, UB)

 CALL QUICK_SORT(K,LB, J-1)

3. [Finished]
 Return
```

Merge Sort

- The **operation of sorting** is closely related to **process of merging**
- Merge Sort is a **divide and conquer algorithm**
- It is based on the **idea of breaking down a list into several sub-lists** until each sub list consists of a **single element**
- **Merging those sub lists** in a manner that results into a sorted list
- **Procedure**
  - **Divide** the unsorted **list into N sub** lists, **each containing 1 element**
  - Take **adjacent pairs** of two singleton lists and **merge them** to form a **list of 2 elements**. N will now convert into N/2 lists of size 2
  - Repeat the process till a single sorted list of obtained
- Time complexity is **O(n log n)**

Merge Sort

Unsorted Array

724	521	2	98	529	31	189	451
0	1	2	3	4	5	6	7

Step 1: Split the selected array (as evenly as possible)

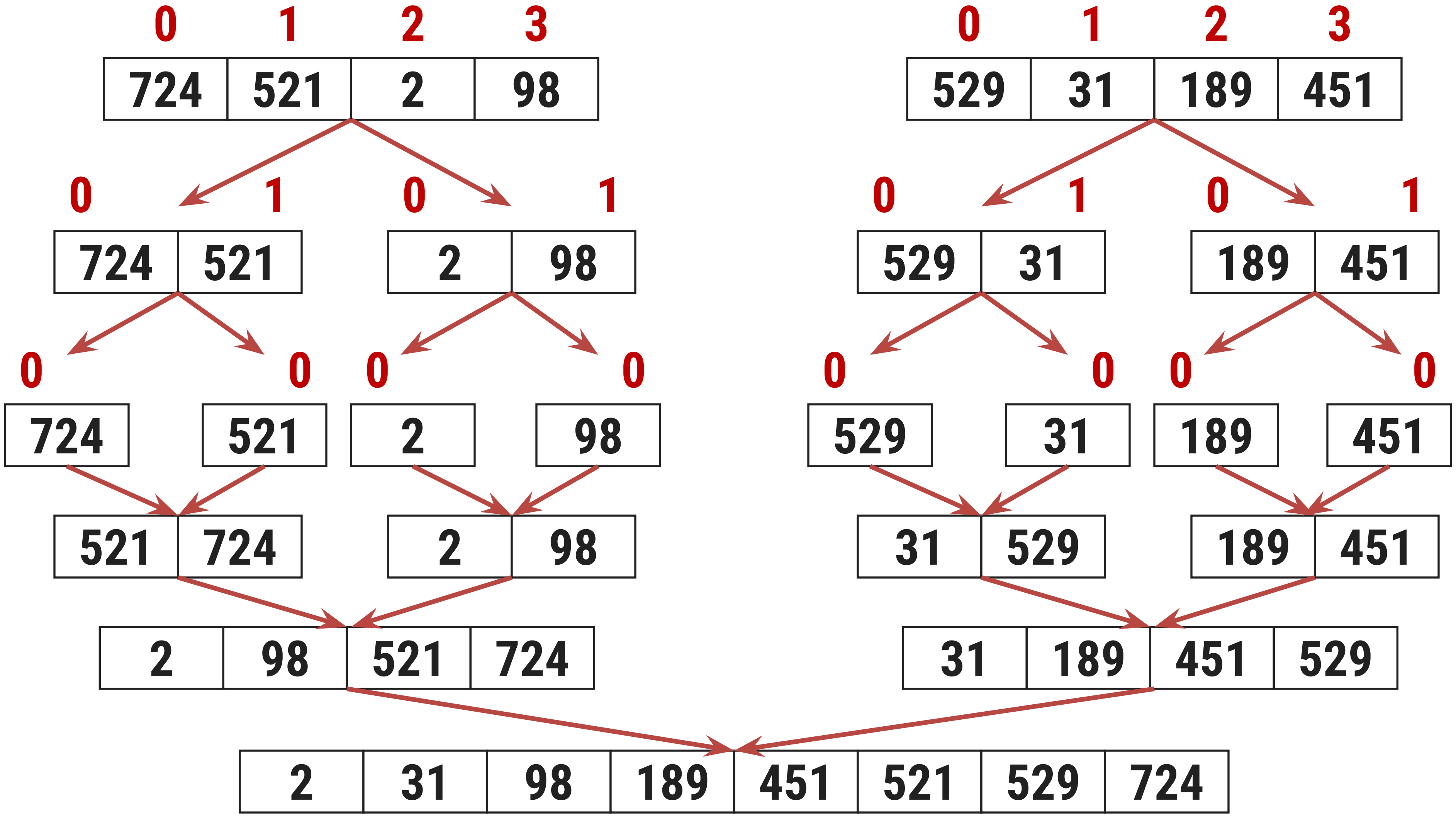
0	1	2	3	4	5	6	7
724	521	2	98	529	31	189	451

724	521	2	98
0	1	2	3

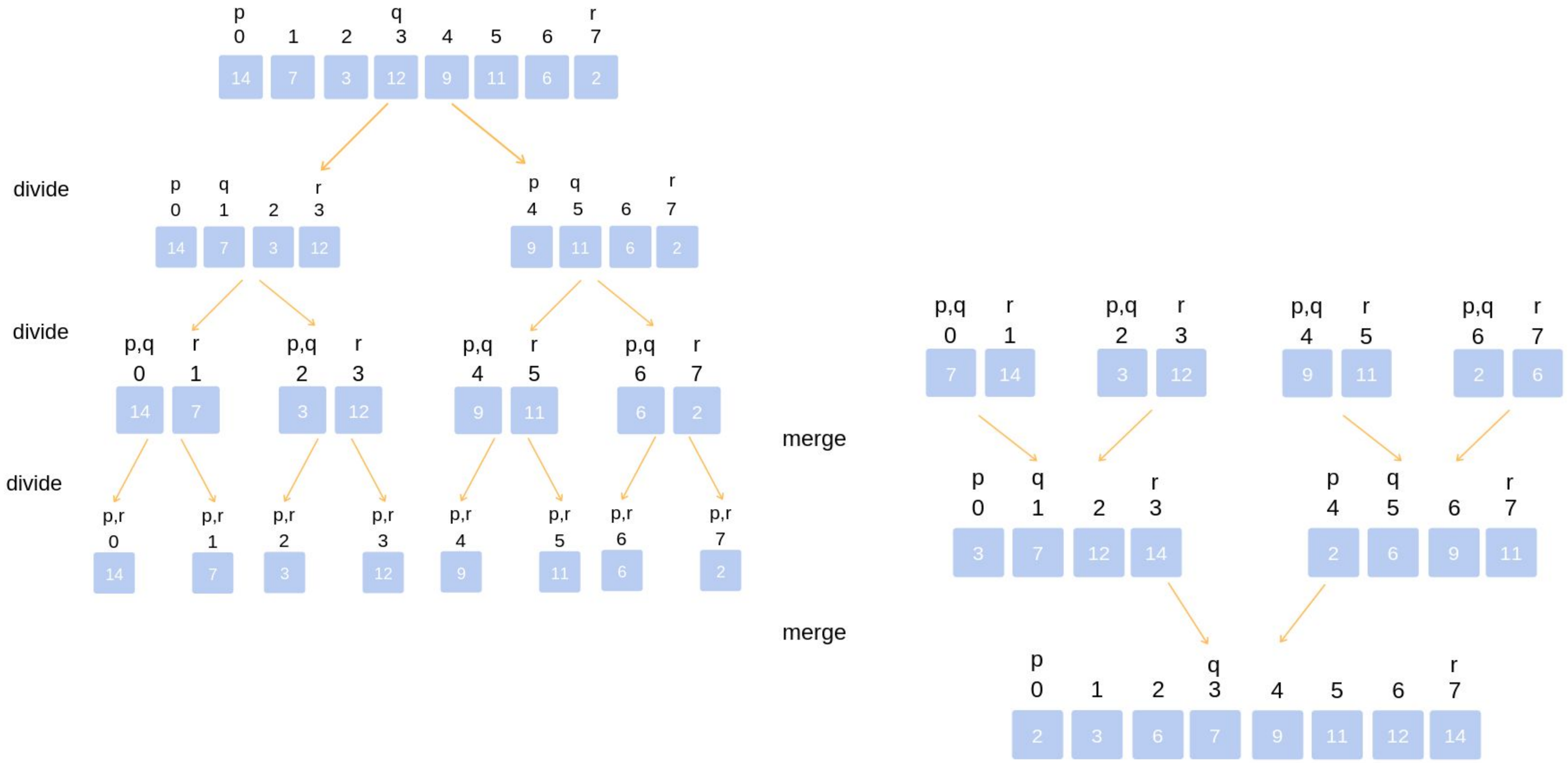
529	31	189	451
0	1	2	3

Merge Sort

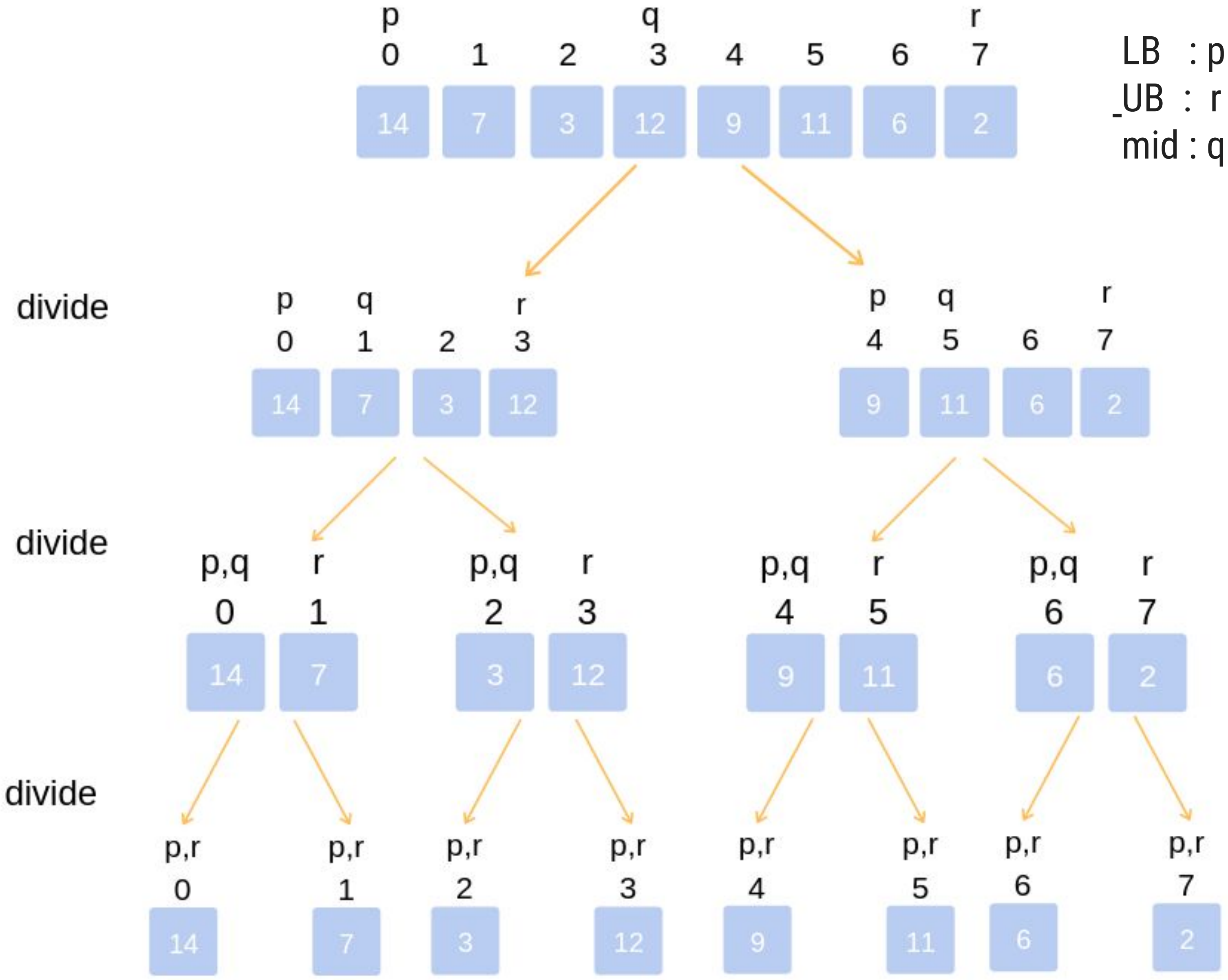
Step: Select the left subarray, Split the selected array (as evenly as possible)



Merge Sort


p
q
r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 2
divide
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
divide
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
divide
p,r 0 14
p,r 1 7
p,r 2 3
p,r 3 12
p,r 4 9
p,r 5 11
p,r 6 6
p,r 7 2
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
merge
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
merge
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

Merge Sort

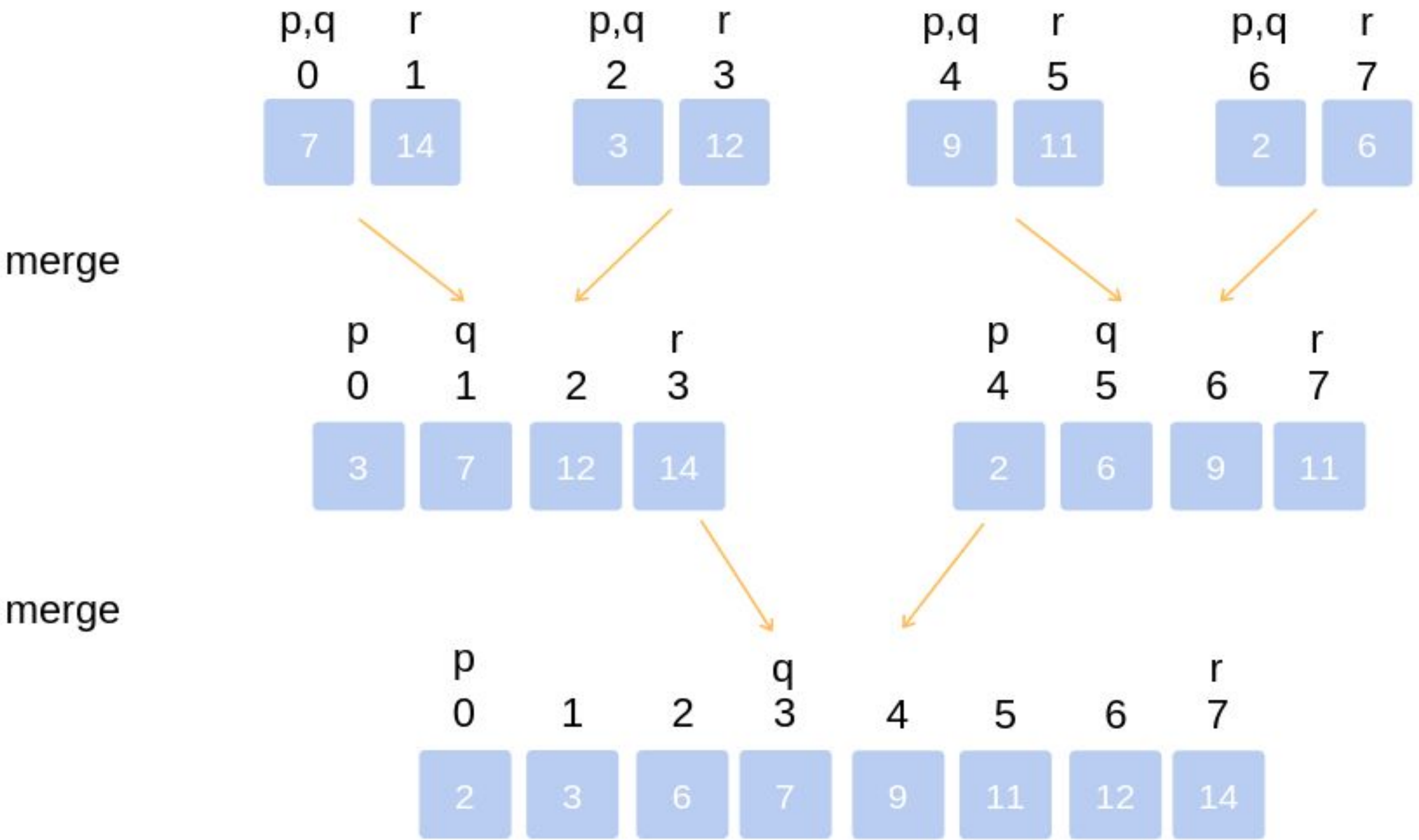


LB : p = 0

UB : r = 7

mid : q = (p + q) / 2 = 7/2 = 3 (int)

Merge Sort



p,q
r
0
1
7
14
p,q
r
2
3
3
12
p,q
r
4
5
9
11
p,q
r
6
7
2
6
merge
p
q
r
0
1
2
3
3
7
12
14
p
q
r
4
5
6
7
2
6
9
11
merge
p
q
r
0
1
2
3
4
5
6
7
2
3
6
7
9
11
12
14

Algorithm:

```
procedure mergesort( var a as array )

1. if ( n == 1 )
    Return a

2. var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

3. l1 = mergesort( l1 )
   l2 = mergesort( l2 )

4. merge( l1, l2 )

5. Return
```

```
procedure merge( var a as array, var b as array )
  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while
  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while
  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while
  return c
end procedure
```

Algorithm: MERGE_SORT(K,LB,UB)

```
1. [Initialize]
   FLAG ← true
2. [Perform Sort]
   IF   LB < UB
   Then I ← LB
        J ← UB + 1
        KEY ← K[LB]
        Repeat While FLAG = true
          I ← I+1
          Repeat While K[I] < KEY
              I ← I + 1
          J ← J – 1
          Repeat While K[J] > KEY
              J ← J – 1
          IF   I<J
          Then K[I] ←---→ K[J]
          Else FLAG ← FALSE

        K[LB] ←---→ K[J]

 CALL QUICK_SORT(K,LB, J-1)
 CALL QUICK_SORT(K,J+1, UB)

 CALL QUICK_SORT(K,LB, J-1)

3. [Finished]
 Return
```

Insertion Sort

In insertion sort, **every iteration moves** an **element** from **unsorted portion** to **sorted portion** until all the elements are sorted in the list.

Steps for Insertion Sort	
1	Assume that **first element** in the list is in **sorted portion** of the list and **remaining all elements** are in **unsorted portion**.
2	Select **first element** from the **unsorted list** and **insert** that element **into the sorted** list in **order specified**.
3	**Repeat** the above process **until all** the **elements** from the **unsorted list** are **moved into** the **sorted list.**

This algorithm is not suitable for large data sets

Insertion Sort cont.

Complexity of the Insertion Sort Algorithm

To sort a unsorted list with **'n'** number of **elements** we **need** to make **(1+2+3+......+n-1)** = **(n (n-1))/2** number of **comparisons** in the worst case.

If the list **already sorted**, then it requires **'n'** number of **comparisons**.

- Worst Case : $\Theta(n^2)$
- Best Case : $\Omega(n)$
- Average Case : $\Theta(n^2)$

Insertion Sort Example

Sort given array using Insertion Sort

6	5	3	1	8	7	2	4

Pass - 1 : Select First Record and considered as Sorter Sub-array

6	5	3	1	8	7	2	4

Sorted | Unsorted

Pass - 2 : Select Second Record and Insert at proper place in sorted array

6	5	3	1	8	7	2	4

5	6	3	1	8	7	2	4

Sorted | Unsorted

Insertion Sort Example Cont.

Pass - 3 : Select Third record and Insert at proper place in sorted array

5	6	3	1	8	7	2	4

3	5	6	1	8	7	2	4

Sorted | Unsorted

Pass - 4 : Select Forth record and Insert at proper place in sorted array

3	5	6	1	8	7	2	4

1	3	5	6	8	7	2	4

Sorted | Unsorted

Insertion Sort Example Cont.

Pass - 5 : Select Fifth record and Insert at proper place in sorted array

1	3	5	6	8	7	2	4

8 is at proper position

1	3	5	6	8	7	2	4

Sorted | Unsorted

Pass - 6 : Select Sixth Record and Insert at proper place in sorted array

1	3	5	6	8	7	2	4

1	3	5	6	7	8	2	4

Sorted | Unsorted

Insertion Sort Example Cont.

Pass - 7 : Select Seventh record and Insert at proper place in sorted array

1	3	5	6	7	8	2	4

1	2	3	5	6	7	8	4

Sorted | Unsorted

Pass - 8 : Select Eighth Record and Insert at proper place in sorted array

1	2	3	5	6	7	8	4

1	2	3	4	5	6	7	8

Sorted | Unsorted

Thank You