Week 9

Sorting

Sorting

Given a collection of data elements, the task of sorting is to arrange the given data in a particular order – *either ascending or descending*

Given data: 5 8 1 3 7 9 6

Ascending Order: 1 3 5 6 7 8 9

Descending Order: 9 8 7 6 5 3 1

Sorting- Terminologies

Internal Memory Sorting Algorithm: Entire data need to be in RAM

Sorting- Terminologies

Internal Memory Sorting Algorithm: Entire data need to be in RAM

External Memory Sorting Algorithm: Entire data not necessarily need to be in RAM

Sorting- Terminologies

Internal Memory Sorting Algorithm: Entire data need to be in RAM

External Memory Sorting Algorithm: Entire data not necessarily need to be in RAM

Stable Sorting Algorithm: A sorting algorithm is said to be stable, if two data elements with the same value appear in the same order in the sorted output, as they appear in the unsorted input.

Sorting- Terminologies

Internal Memory Sorting Algorithm: Entire data need to be in RAM

External Memory Sorting Algorithm: Entire data not necessarily need to be in RAM

Stable Sorting Algorithm: A sorting algorithm is said to be stable, if two data elements with the same value appear in the same order in the sorted output, as they appear in the unsorted input.

In-placed Sorting Algorithm: Sorting algorithms which do not use auxiliary data structure for storing data elements other than the original input storage are known as in-placed algorithms.

Sorting- Terminologies

Adaptive and Non-adaptive:

- If the time taken for sorting depends on the initial ordering of the elements in the input collection, the algorithm is known as adaptive.
- If initial ordering does not affect the sorting time, the algorithm is called non-adaptive.

Sorting- Terminologies

Adaptive and Non-adaptive:

- If the time taken for sorting depends on the initial ordering of the elements in the input collection, the algorithm is known as adaptive.
- If initial ordering does not affect the sorting time, the algorithm is called non-adaptive.

Inversion Count:

- Given an array, inversion count denotes how far the given array is from its sorted order.
- If the elements in the input array is already in the target ordering, then its inversion count is 0.
- If the elements in the input array is in the reverse order of the target ordering, its inversion count is maximum.

Sorting- Terminologies

Adaptive and Non-adaptive:

- If the time taken for sorting depends on the initial ordering of the elements in the input collection, the algorithm is known as adaptive.
- If initial ordering does not affect the sorting time, the algorithm is called non-adaptive.

Inversion Count:

- Given an array, inversion count denotes how far the given array is from its sorted order.
- If the elements in the input array is already in the target ordering, then its inversion count is 0.
- If the elements in the input array is in the reverse order of the target ordering, its inversion count is maximum.

Lower Bound Theory: It defines the minimum time complexity of a class of algorithms.

Sorting- Terminologies

Adaptive and Non-adaptive:

- If the time taken for sorting depends on the initial ordering of the elements in the input collection, the algorithm is known as adaptive.
- If initial ordering does not affect the sorting time, the algorithm is called non-adaptive.

Inversion Count:

- Given an array, inversion count denotes how far the given array is from its sorted order.
- If the elements in the input array is already in the target ordering, then its inversion count is 0.
- If the elements in the input array is in the reverse order of the target ordering, its inversion count is maximum.

Lower Bound Theory: It defines the minimum time complexity of a class of algorithms.

Comparison and non-comparison: Sorting algorithms which perform sorting by comparing the values of the data elements are referred to as comparison sorting algorithms, otherwise, non-comparison.

Bubble Sort

Bubble Sort

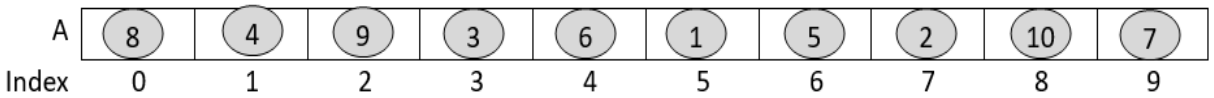
- Bubble sort scans the array multiple times (referred to as ***Pass***) from the lower index to the highest applicable index.

- In each pass, depending on whether the algorithm attempts to arrange the data in ascending or descending order, the algorithm will bubble up the largest or smallest element.

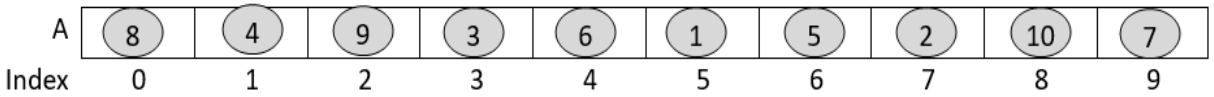
Bubble Sort

- Bubble sort scans the array multiple times (referred to as *pass*) from the lower index to the highest applicable index

- In each pass, depending on whether the algorithm attempts to arrange the data in ascending or descending order, the algorithm will bubble up the largest or smallest element.

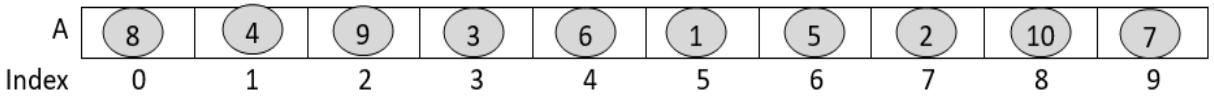


Bubble Sort

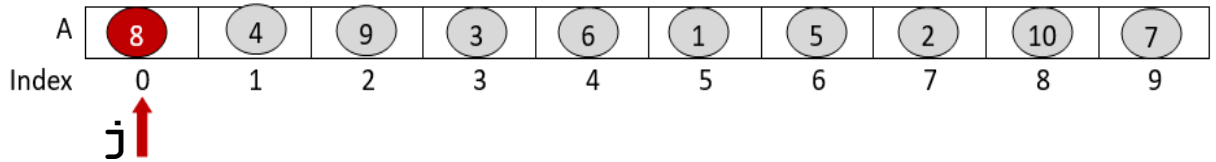


Pass 1

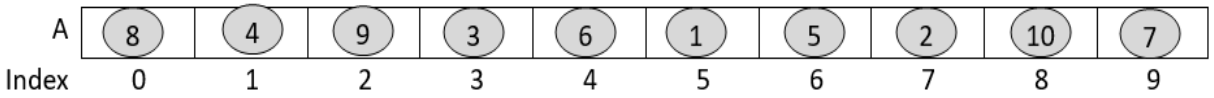
Bubble Sort



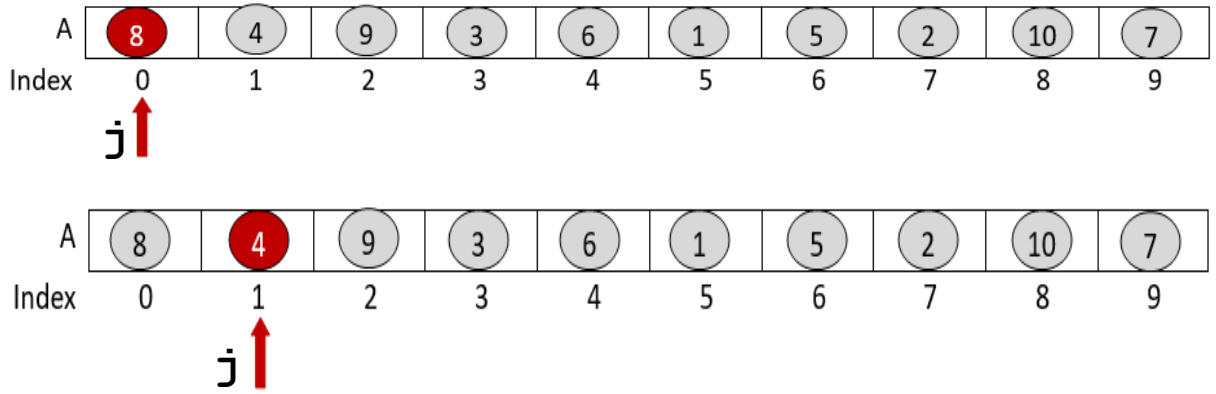
Pass 1: i = 0



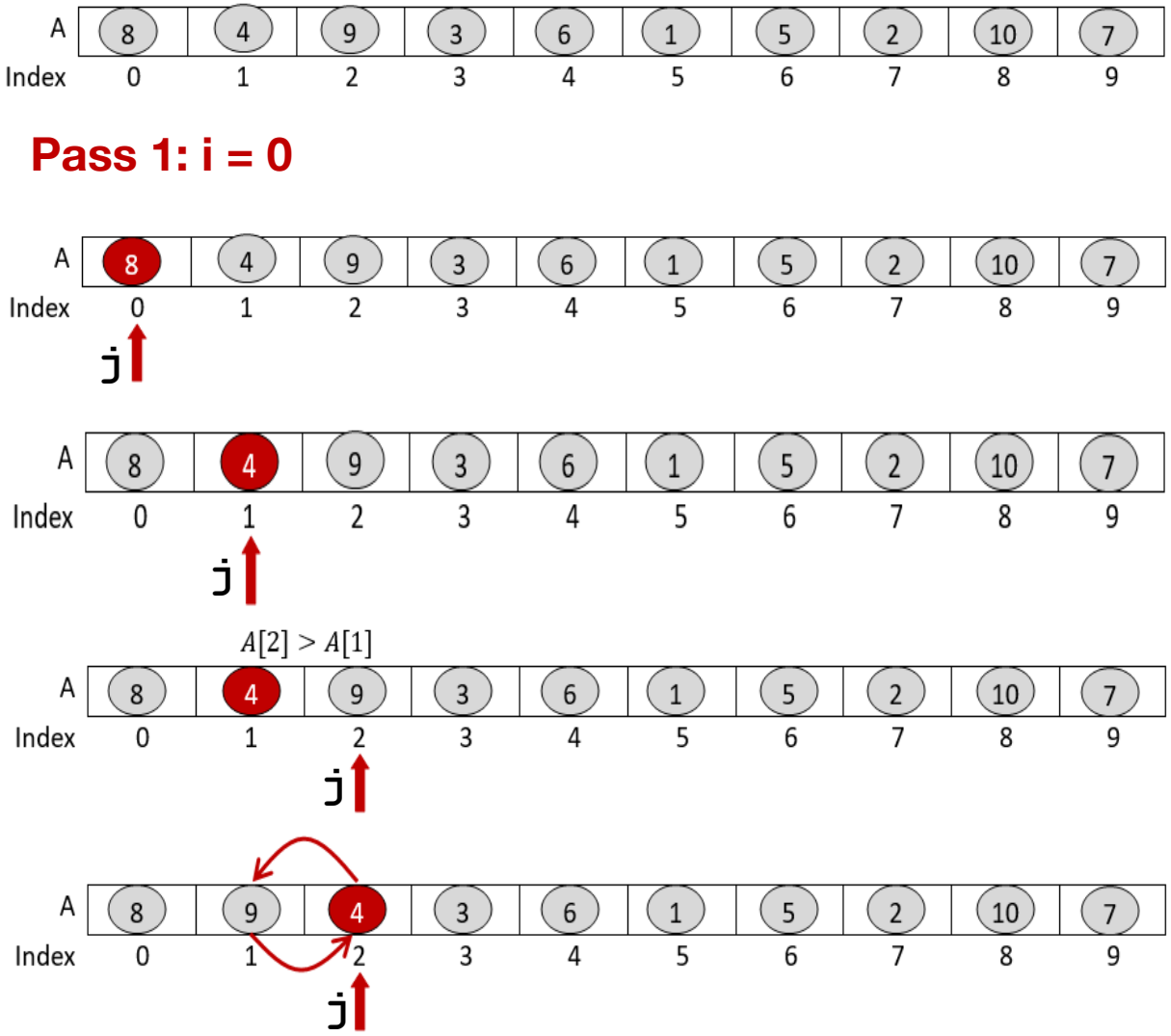
Bubble Sort



Pass 1: i = 0

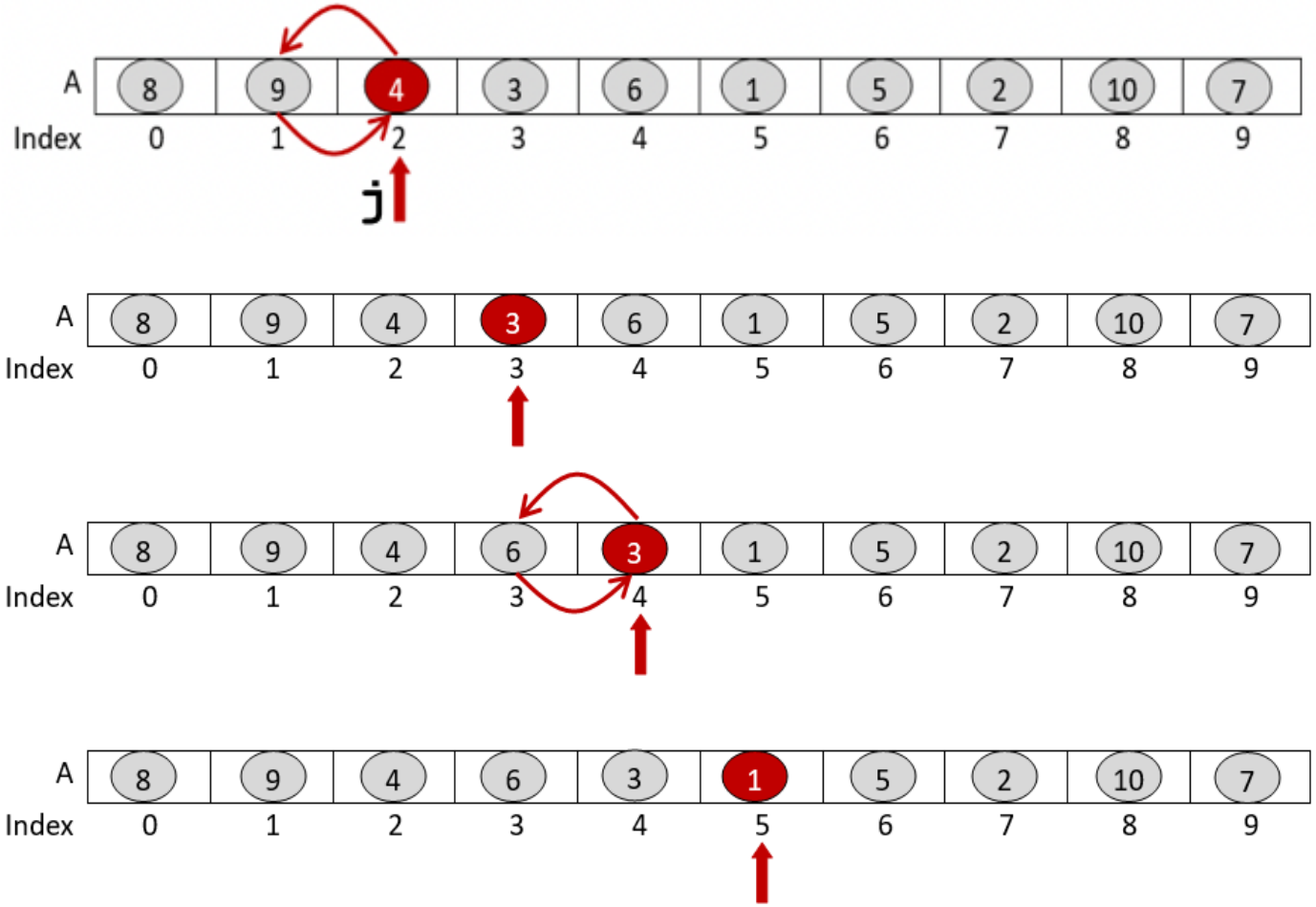


Bubble Sort


A
8 4 9 3 6 1 5 2 10 7
Index
0 1 2 3 4 5 6 7 8 9
Pass 1: i = 0
A
8 4 9 3 6 1 5 2 10 7
Index
0 1 2 3 4 5 6 7 8 9
j
A
8 4 9 3 6 1 5 2 10 7
Index
0 1 2 3 4 5 6 7 8 9
j
A[2] > A[1]
A
8 4 9 3 6 1 5 2 10 7
Index
0 1 2 3 4 5 6 7 8 9
j
A
8 9 4 3 6 1 5 2 10 7
Index
0 1 2 3 4 5 6 7 8 9
j

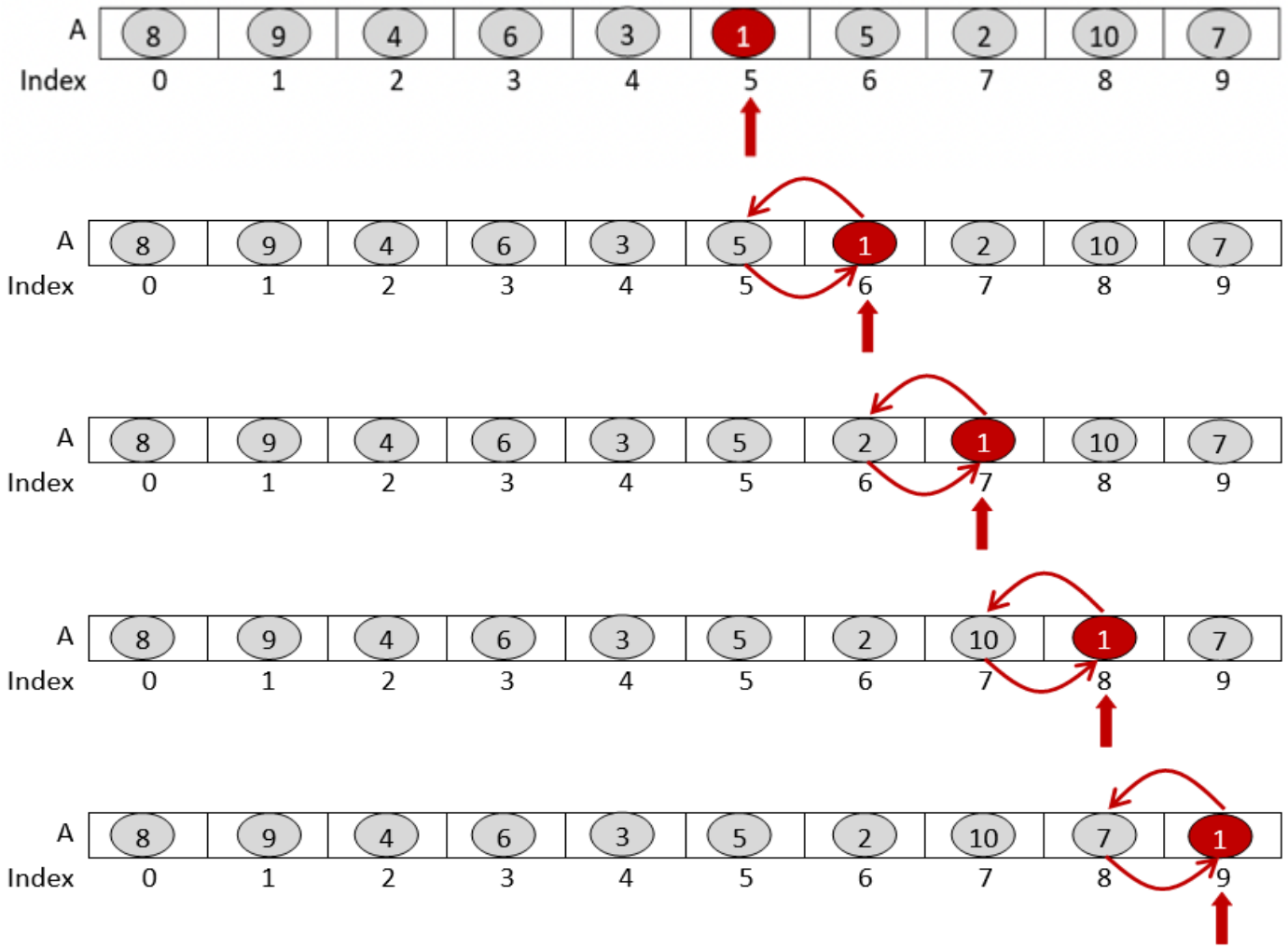
Bubble Sort

Pass 1: i = 0



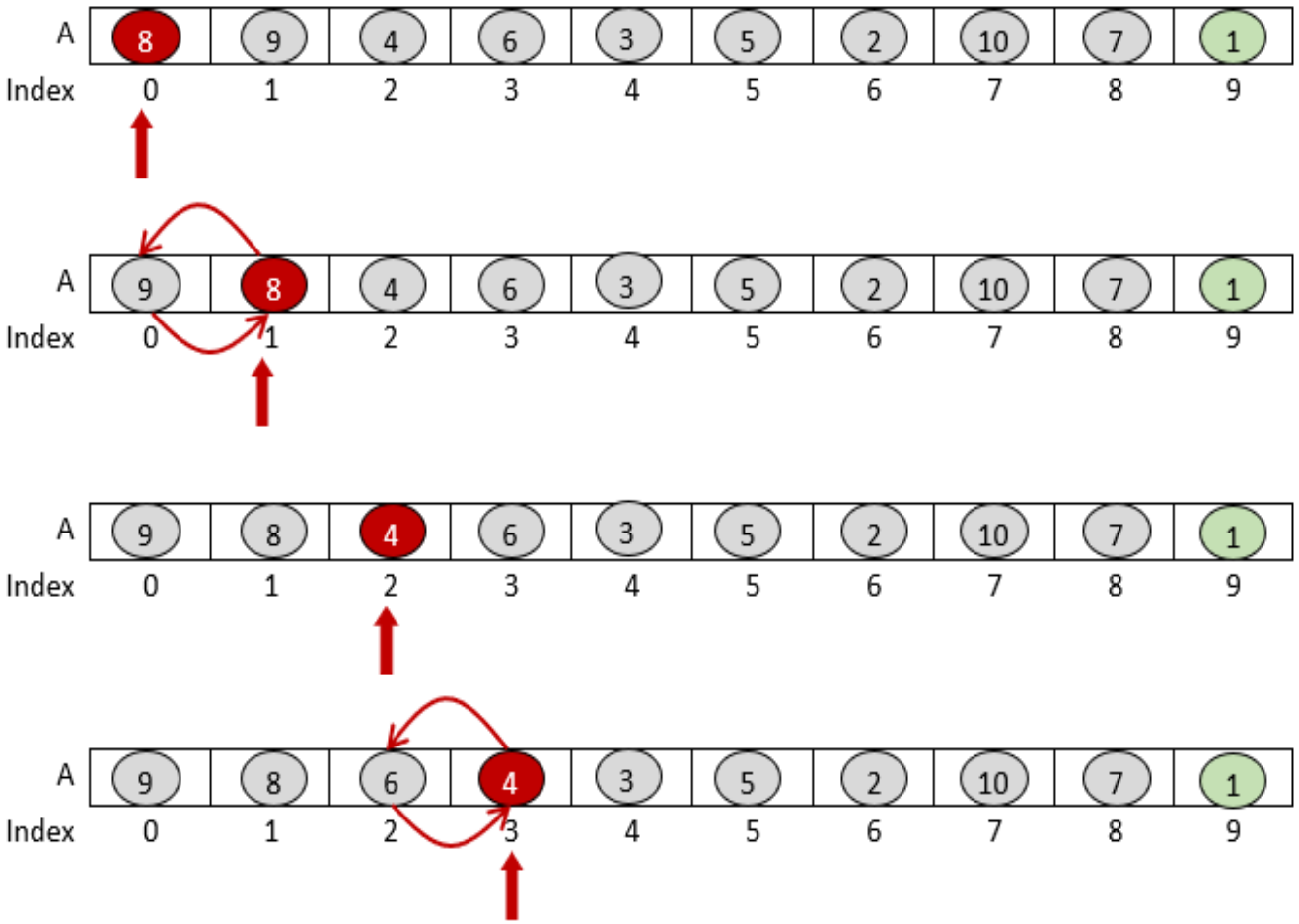
Bubble Sort

Pass 1: i = 0



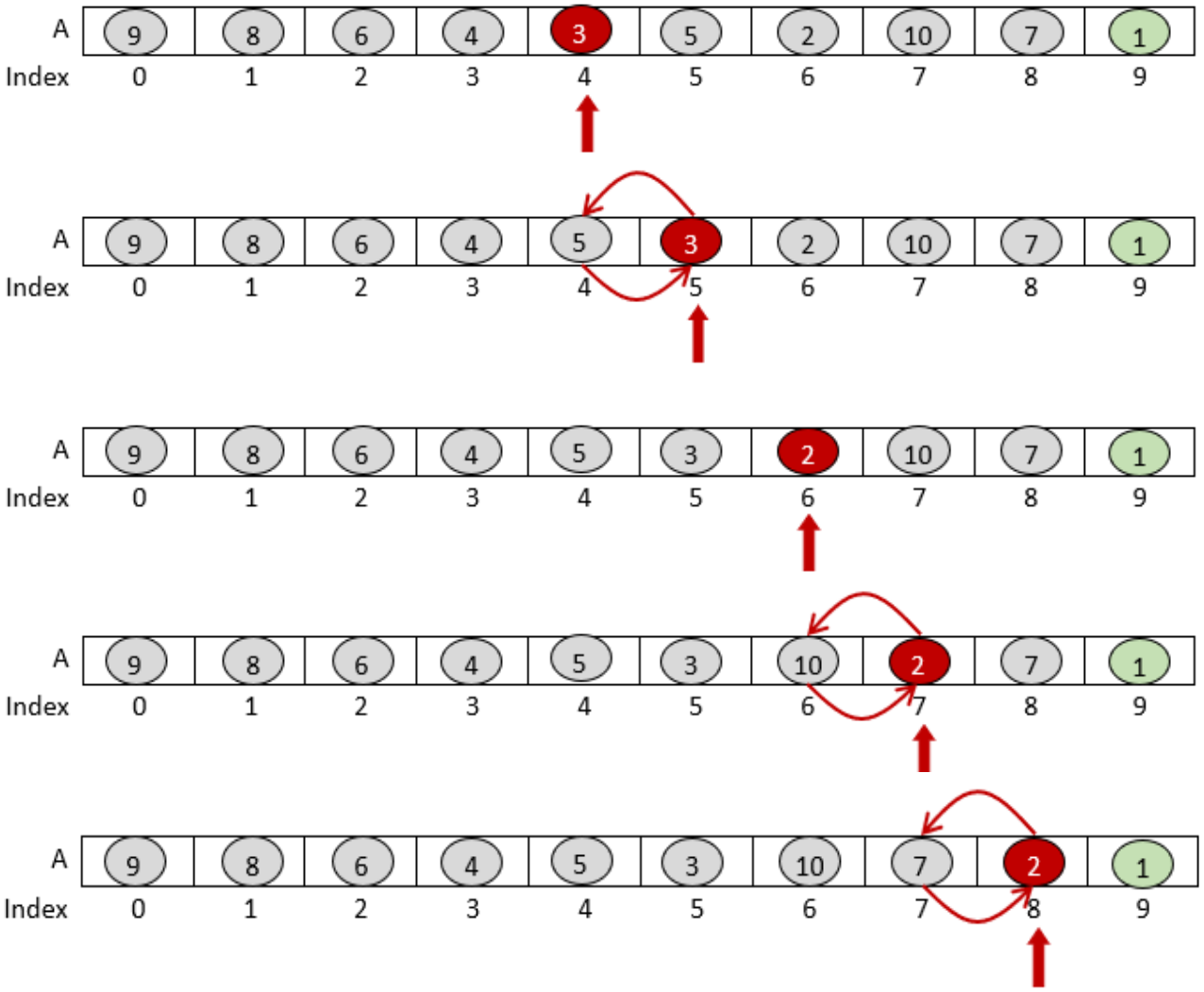
Bubble Sort

Pass 2: i = 2

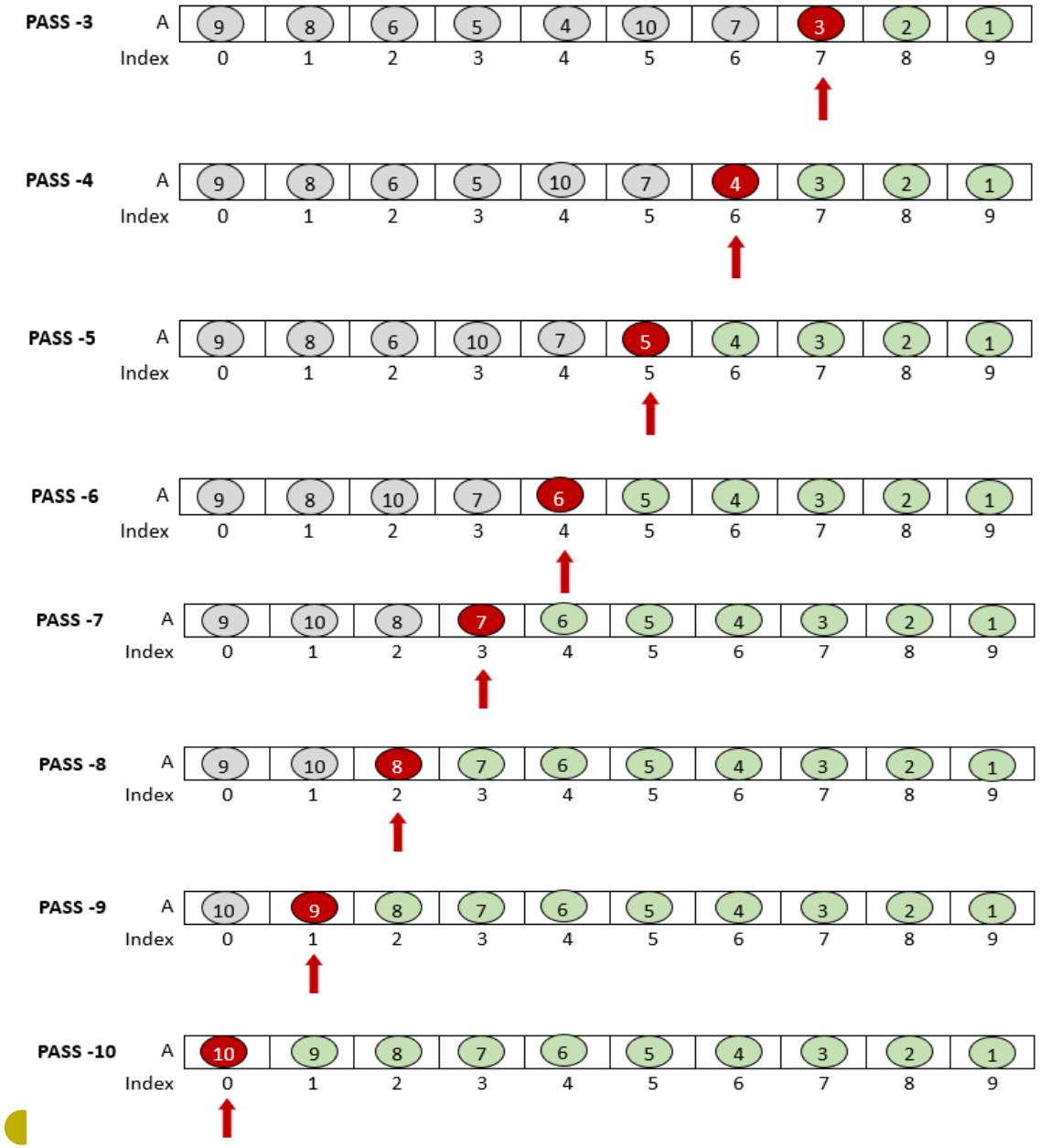


Bubble Sort

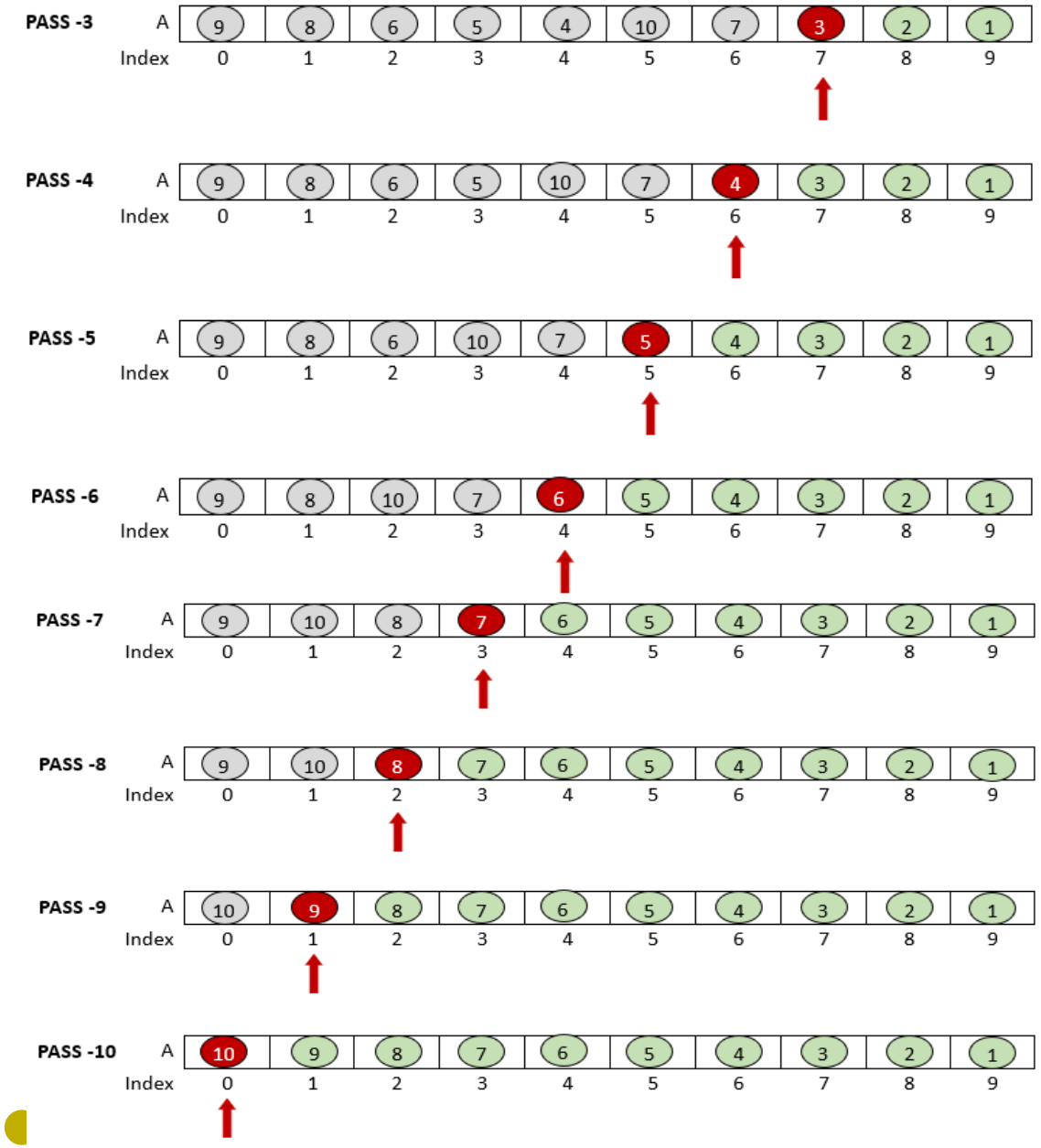
Pass 2: i = 2



Bubble Sort

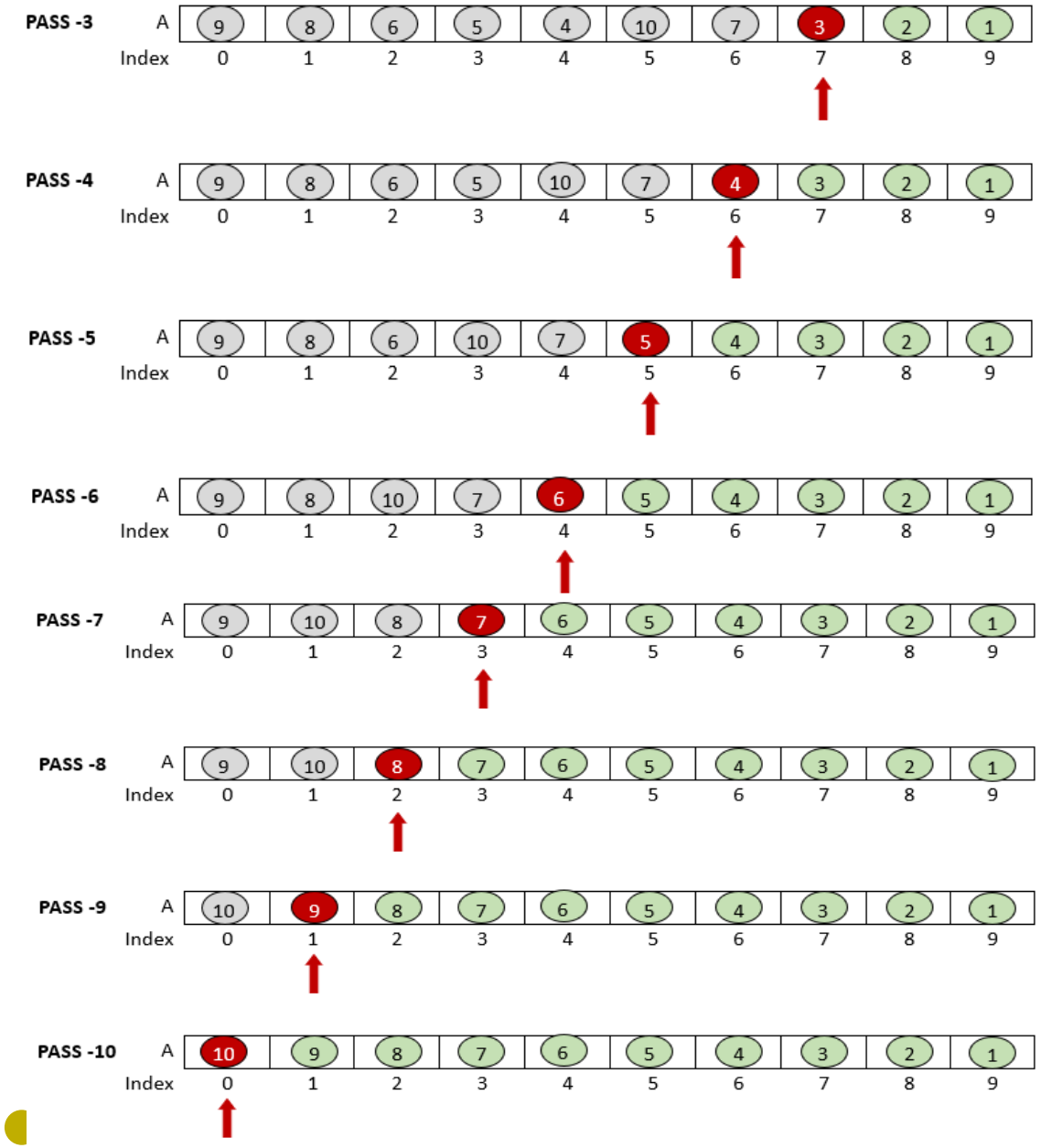

PASS -3
A
9 8 6 5 4 10 7 3 2 1
Index
0 1 2 3 4 5 6 7 8 9
PASS -4
A
9 8 6 5 10 7 4 3 2 1
Index
0 1 2 3 4 5 6 7 8 9
PASS -5
A
9 8 6 10 7 5 4 3 2 1
Index
0 1 2 3 4 5 6 7 8 9
PASS -6
A
9 8 10 7 6 5 4 3 2 1
Index
0 1 2 3 4 5 6 7 8 9
PASS -7
A
9 10 8 7 6 5 4 3 2 1
Index
0 1 2 3 4 5 6 7 8 9
PASS -8
A
9 10 8 7 6 5 4 3 2 1
Index
0 1 2 3 4 5 6 7 8 9
PASS -9
A
10 9 8 7 6 5 4 3 2 1
Index
0 1 2 3 4 5 6 7 8 9
PASS -10
A
10 9 8 7 6 5 4 3 2 1
Index
0 1 2 3 4 5 6 7 8 9

Bubble Sort




```
void bubbleSort(int A[], int n){
    int i, j, temp;
    for (i = 0; i < n - 1; i++){
        for (j = 0; j < n - i - 1; j++){
            if (A[j] < A[j + 1]){
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

Bubble Sort



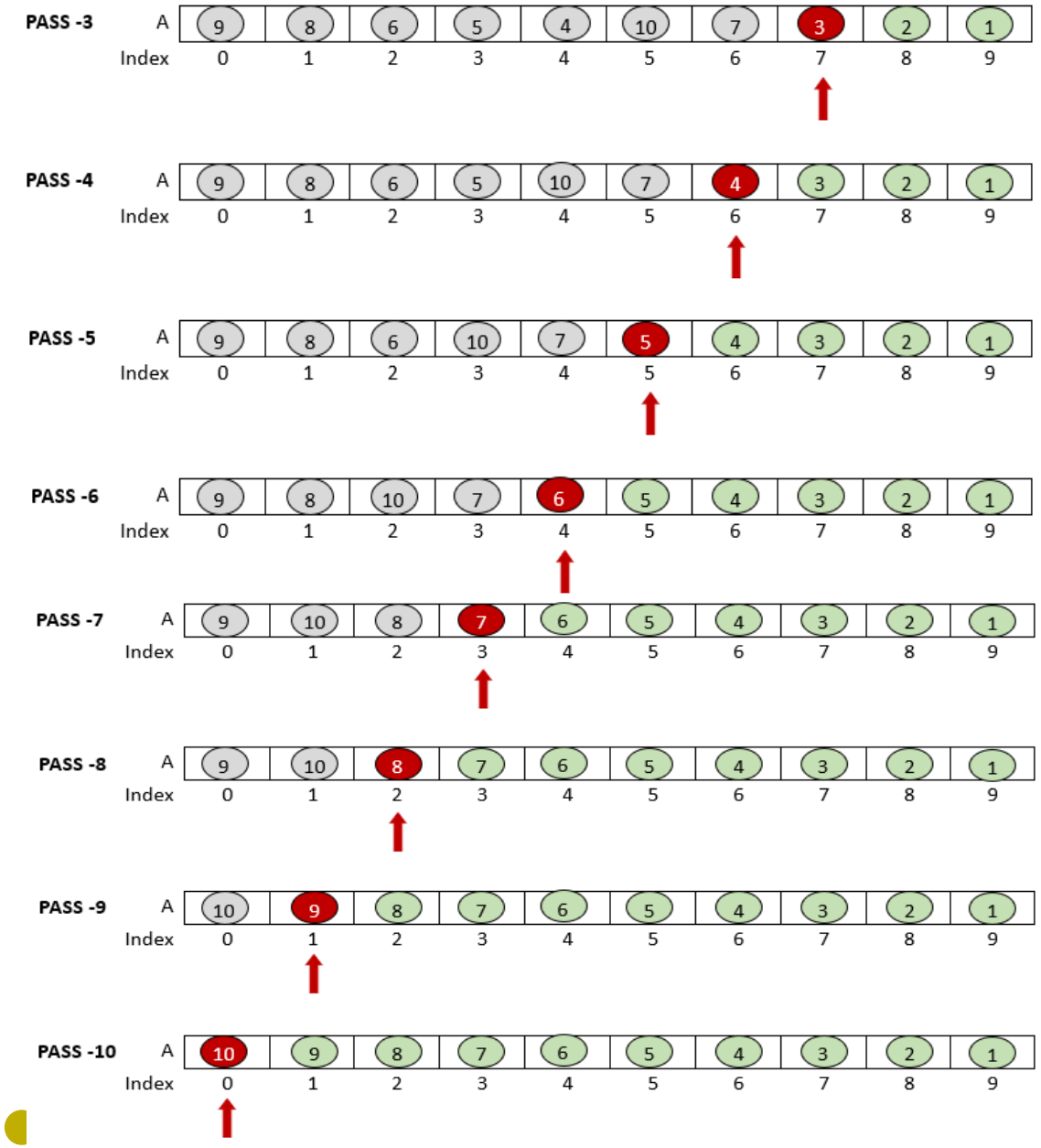

```
void bubbleSort(int A[], int n){
    int i, j, temp;
    for (i = 0; i < n - 1; i++){
        for (j = 0; j < n - i - 1; j++){
            if (A[j] < A[j + 1]){
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

Time Complexity: (No. of Comparisons)

$$(n-1)+(n-2)+(n-3)+\cdots+2+1 = \frac{n(n-1)}{2}$$

$$= O(n^2)$$

Bubble Sort




```
void bubbleSort(int A[], int n){
    int i, j, temp;
    for (i = 0; i < n - 1; i++){
        for (j = 0; j < n - i - 1; j++){
            if (A[j] < A[j + 1]){
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

Time Complexity: (No. of Comparisons)

$$(n-1)+(n-2)+(n-3)+\cdots+2+1 = \frac{n(n-1)}{2}$$

$= O(n^2)$

Best case, Average Case, Worst Case

Bubble Sort - modified

```
void bubbleSort(int A[], int n){
    int i, j, temp, flag;
    for (i = 0; i < n - 1; i++){
       flag=0;
       for (j = 0; j < n - i - 1; j++){
            if (A[j] < A[j + 1]){
                 temp = A[j];
                 A[j] = A[j+1];
                 A[j+1] = temp;
                 flag = 1;
             }
       }
       if(flag==0) break;
     }
}
```

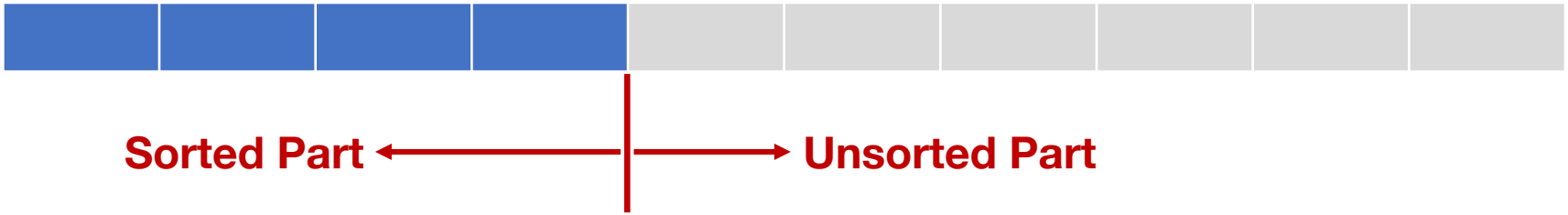
Bubble Sort - modified

```
void bubbleSort(int A[], int n){
    int i, j, temp, flag;
    for (i = 0; i < n - 1; i++){
       flag=0;
       for (j = 0; j < n - i - 1; j++){
            if (A[j] < A[j + 1]){
                 temp = A[j];
                 A[j] = A[j+1];
                 A[j+1] = temp;
                 flag = 1;
             }
       }
       if(flag==0) break;
     }
}
```

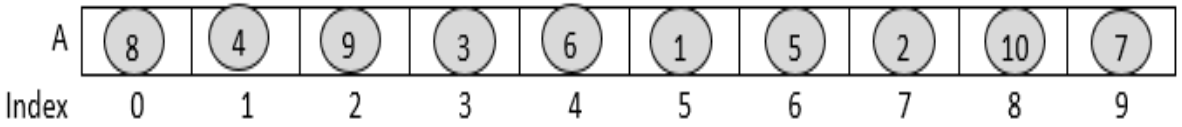
Best case: O(n)

Insertion Sort

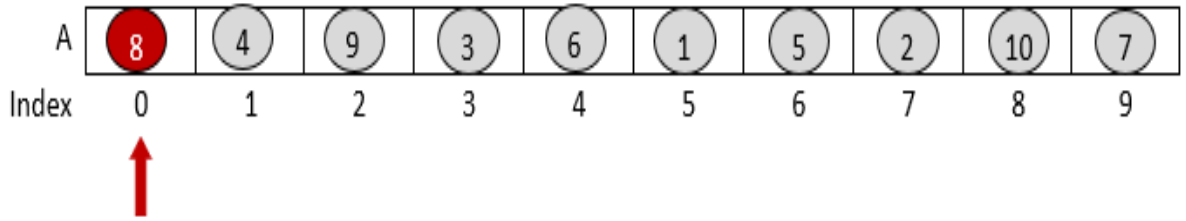
Insertion Sort



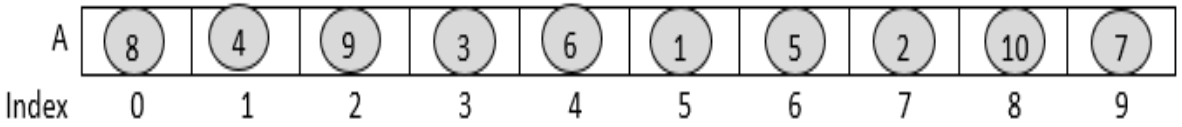
Insertion Sort



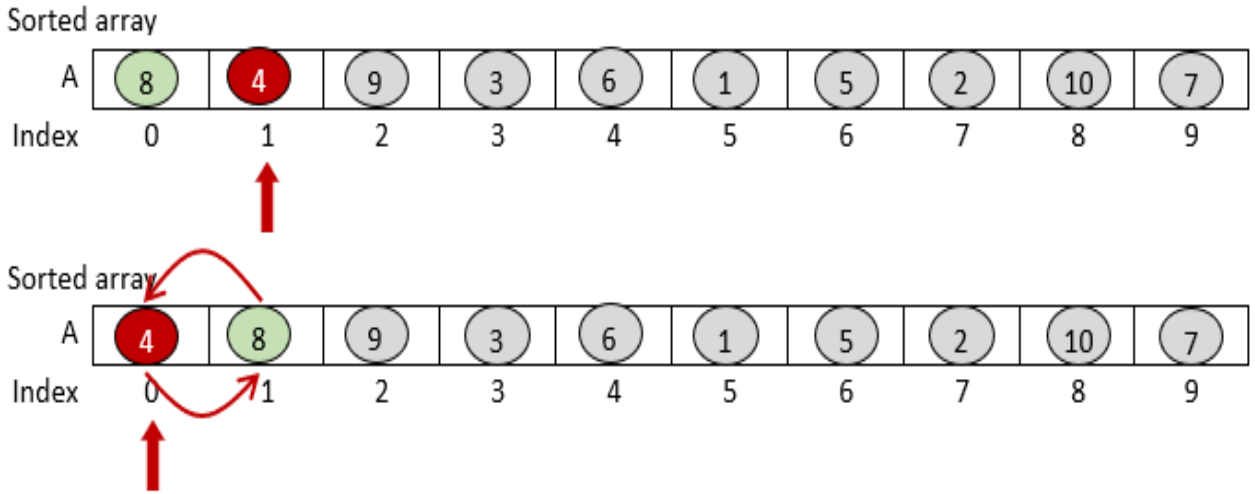
Pass 1



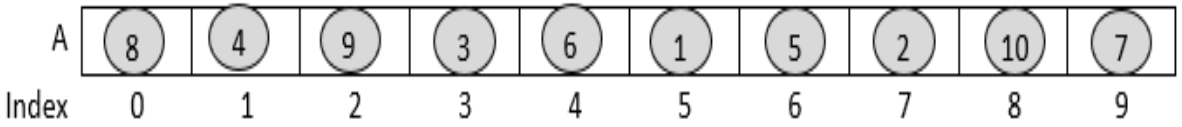
Insertion Sort



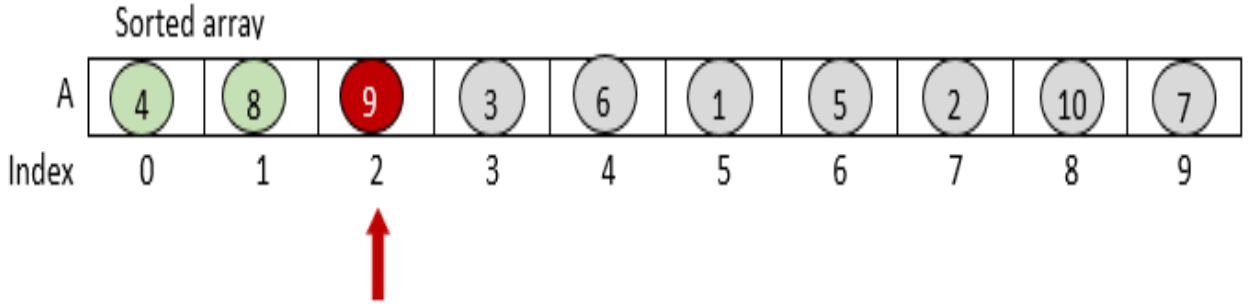
Pass 2



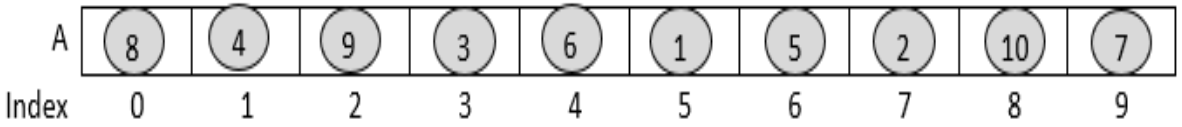
Insertion Sort



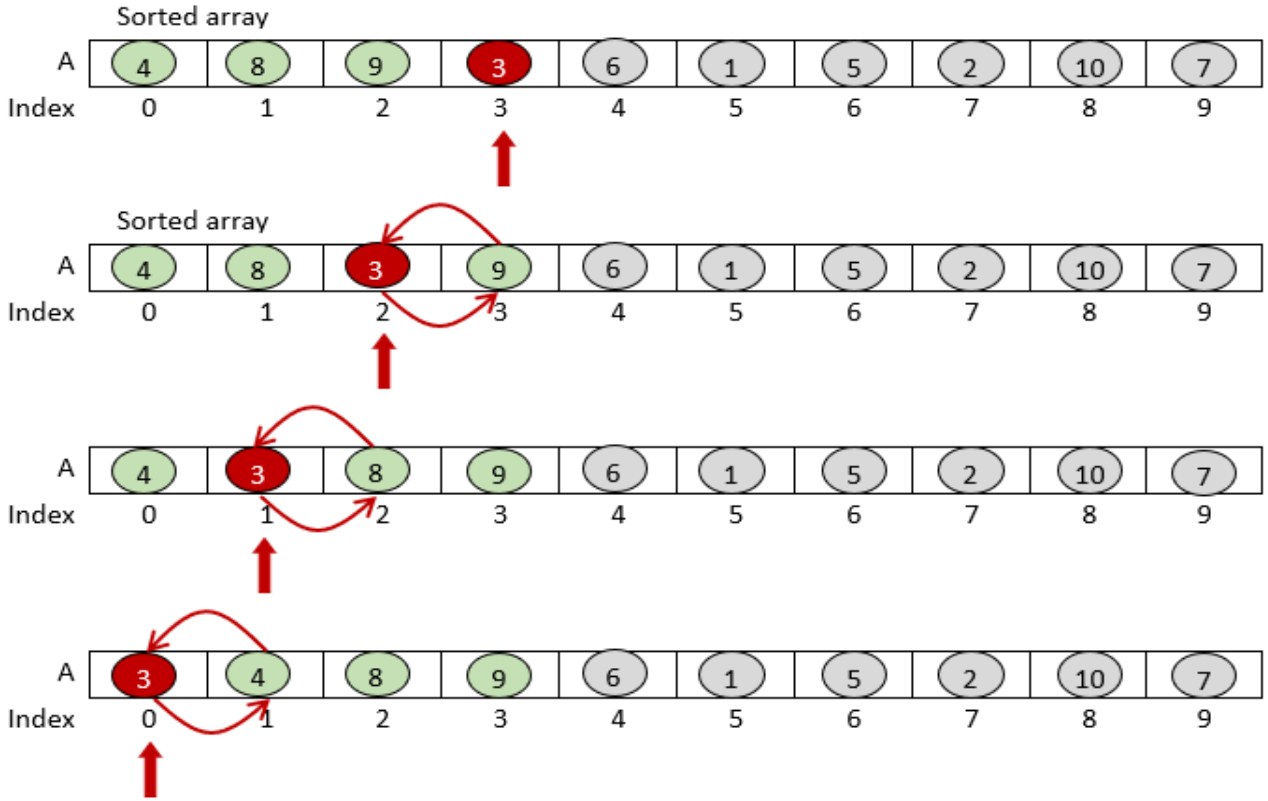
Pass 3



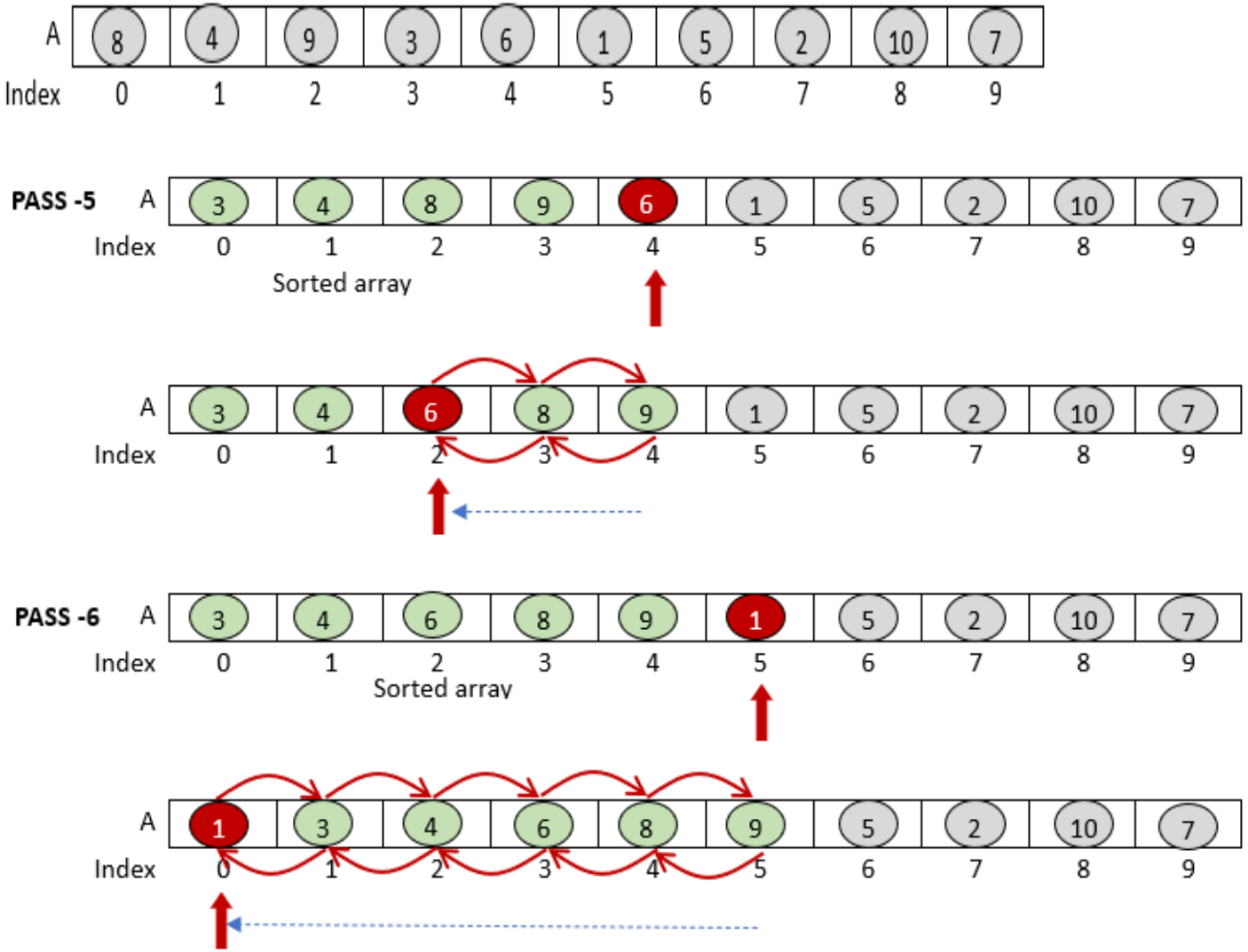
Insertion Sort



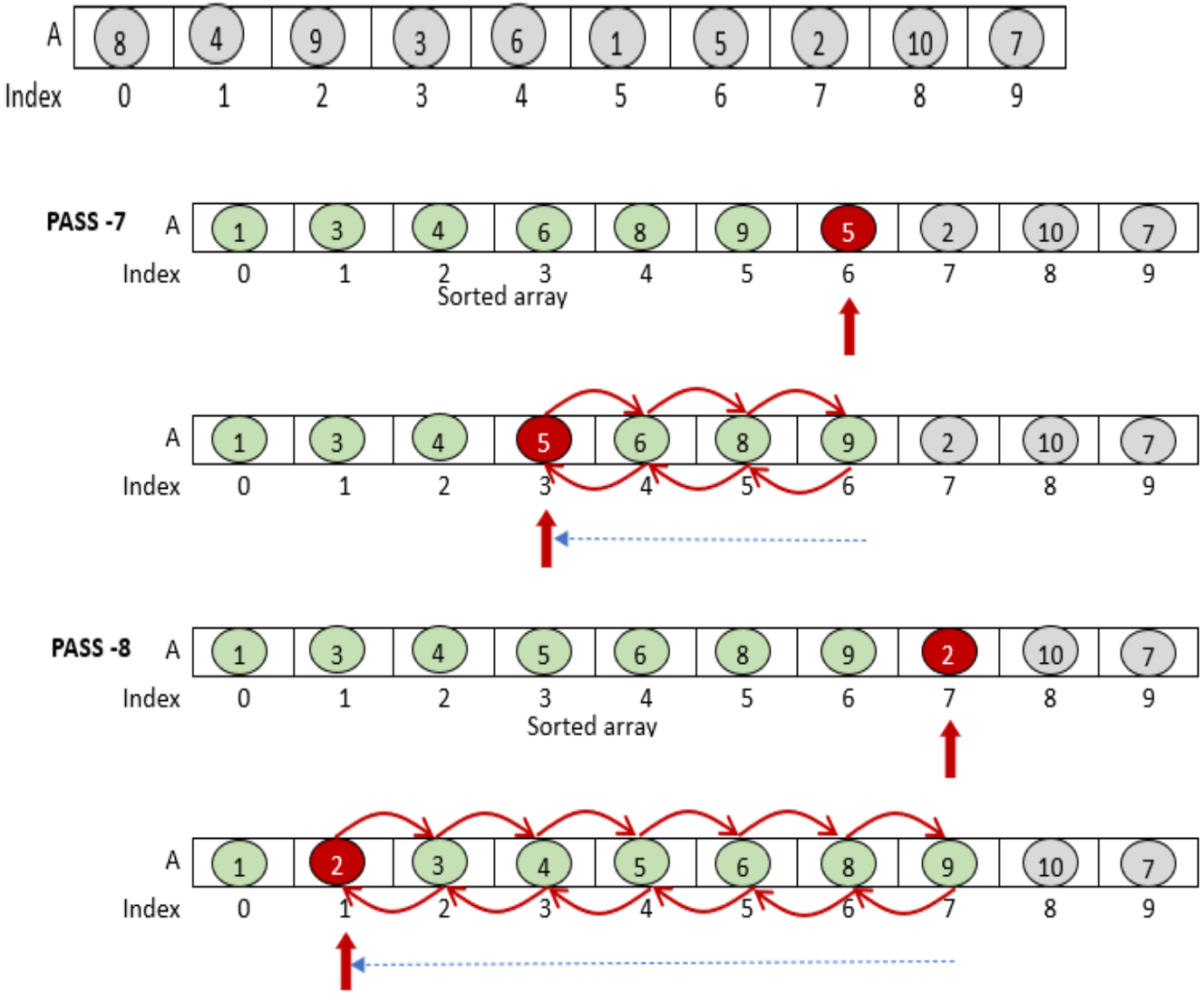
Pass 4



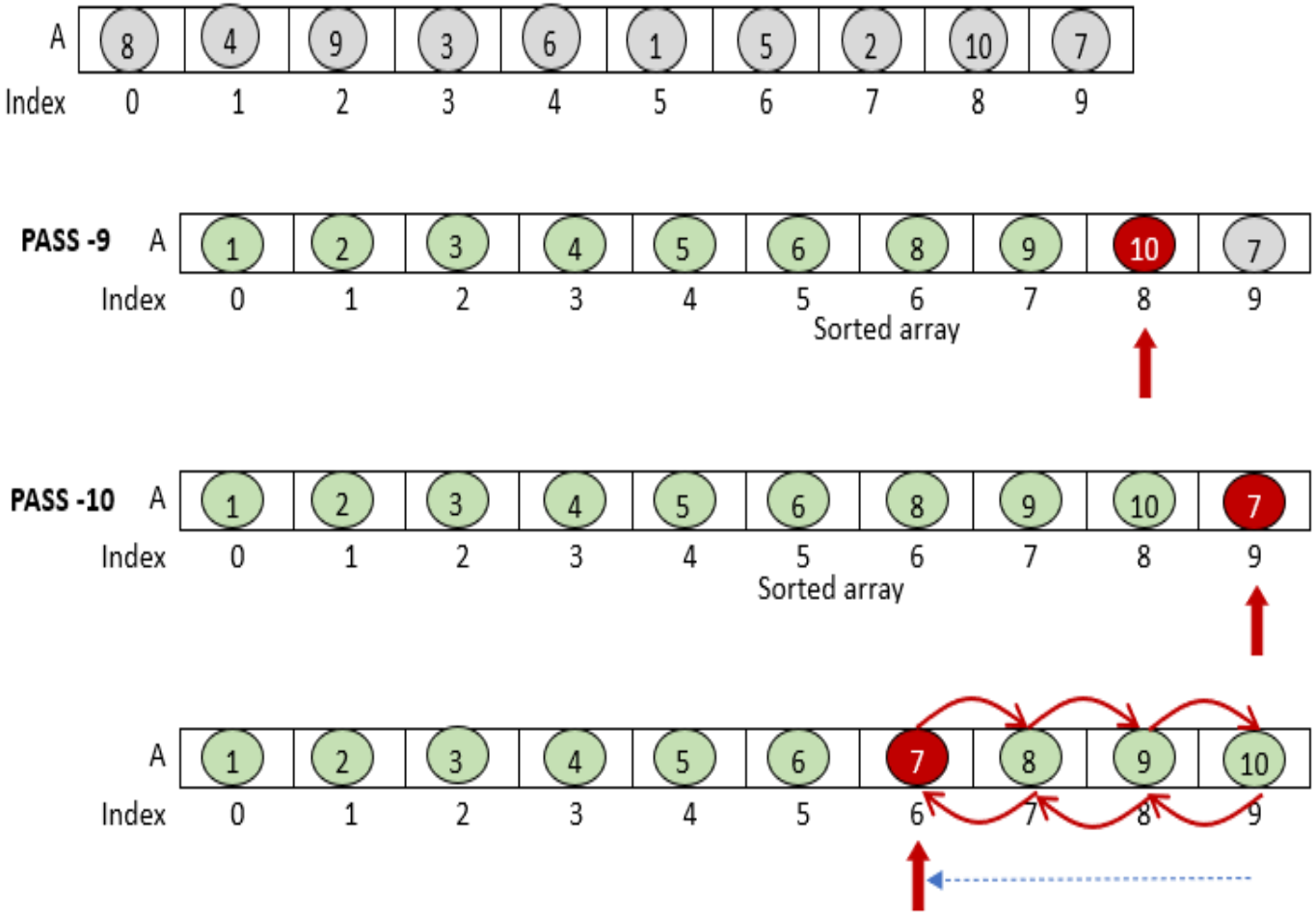
Insertion Sort



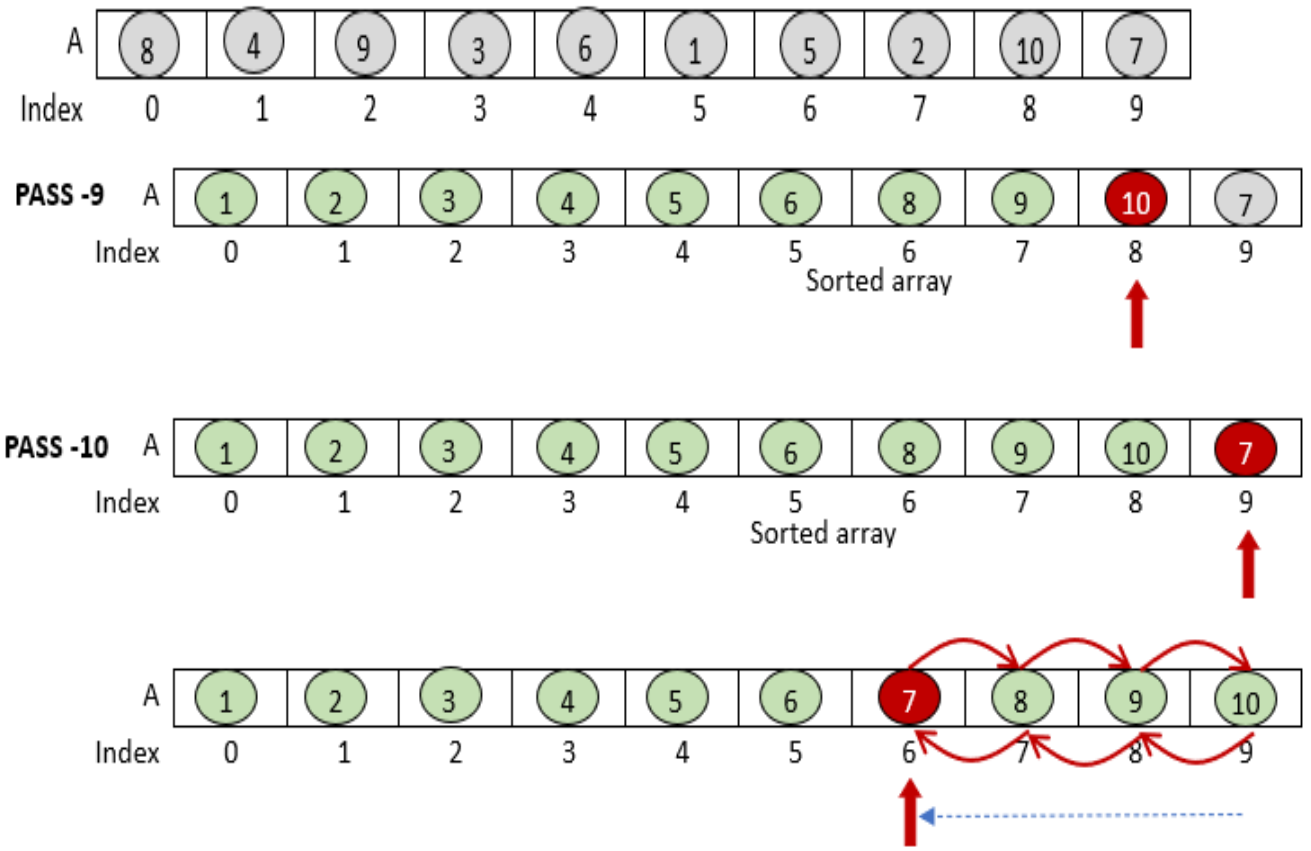
Insertion Sort



Insertion Sort



Insertion Sort




```
void insertionSort(int A[], int n){
    int i, j, ele;
    for (i = 1; i < n; i++){
        j = i - 1;
        ele = A[i];
        while ((j >= 0) && (A[j] > ele)){
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j + 1] = ele;
    }
}
```

Insertion Sort

10	22	34	41	52	67	74	82	91	104

```
void insertionSort(int A[], int n){
    int i, j, ele;
    for (i = 1; i < n; i++){
        j = i - 1;
        ele = A[i];
        while ((j >= 0) && (A[j] > ele)){
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j + 1] = ele;
    }
}
```

Time Complexity:

Best Case: (the array is already sorted)

- **No exchange (only one comparison per Pass)**
- $O(n)$

Insertion Sort

32	31	27	23	18	15	9	8	7	2

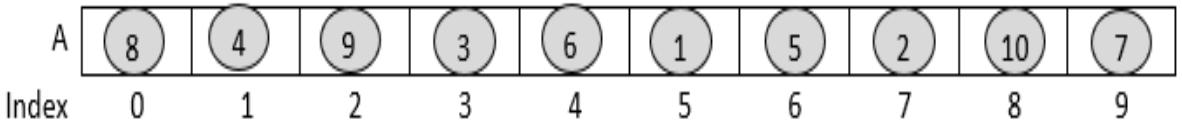
```
void insertionSort(int A[], int n){
    int i, j, ele;
    for (i = 1; i < n; i++){
        j = i - 1;
        ele = A[i];
        while ((j >= 0) && (A[j] > ele)){
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j + 1] = ele;
    }
}
```

Time Complexity:

Worst Case: (the array is already sorted, but in reverse order)

- $1 + 2 + 3 + \cdots . + (n-2) + (n-1) = \frac{n(n-1)}{2}$ **number of comparisons**
- **Order of** $O(n^2)$

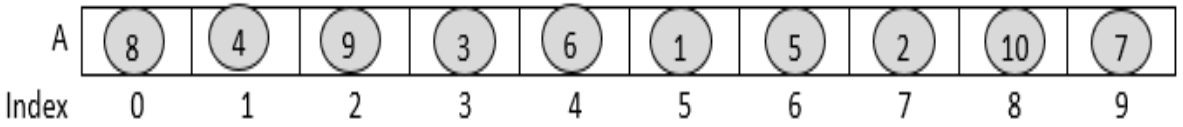
Insertion Sort



For an arbitrary input array, at an arbitrary pass i, there will be $\frac{i}{2}$ number of comparisons on an average.

```
void insertionSort(int A[], int n){
    int i, j, ele;
    for (i = 1; i < n; i++){
        j = i - 1;
        ele = A[i];
        while ((j >= 0) && (A[j] > ele)){
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j + 1] = ele;
    }
}
```

Insertion Sort




```
void insertionSort(int A[], int n){
    int i, j, ele;
    for (i = 1; i < n; i++){
        j = i - 1;
        ele = A[i];
        while ((j >= 0) && (A[j] > ele)){
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j + 1] = ele;
    }
}
```

For an arbitrary input array, at an arbitrary pass i, there will be $\frac{i}{2}$ number of comparisons on an average.

Time Complexity:

- ***On an average,*** $\frac{1}{2}+\frac{2}{2}+\frac{3}{2}+\cdots+\frac{n-2}{2}+\frac{n-1}{2}=\frac{n(n-1)}{4}$ ***number of comparisons***
- **Order of $O(n^2)$.**

Selection Sort

Selection Sort

In bubble sort and insertion sort algorithms, whenever two successive elements are out of order in the inner loop, they are exchanged.

Selection Sort

In bubble sort and insertion sort algorithms, whenever two successive elements are out of order in the inner loop, they are exchanged.

Can we reduce the number of swap operations while sorting an array?

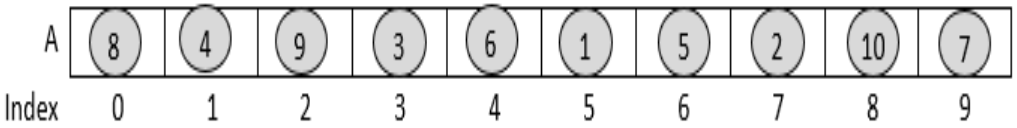
Selection Sort

In bubble sort and insertion sort algorithms, whenever two successive elements are out of order in the inner loop, they are exchanged.

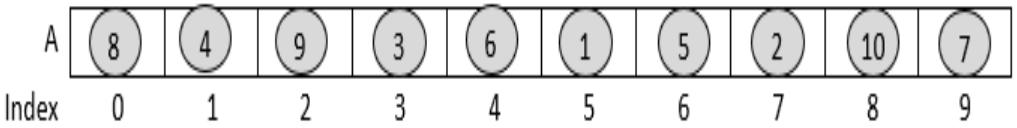
Can we reduce the number of swap operations while sorting an array?

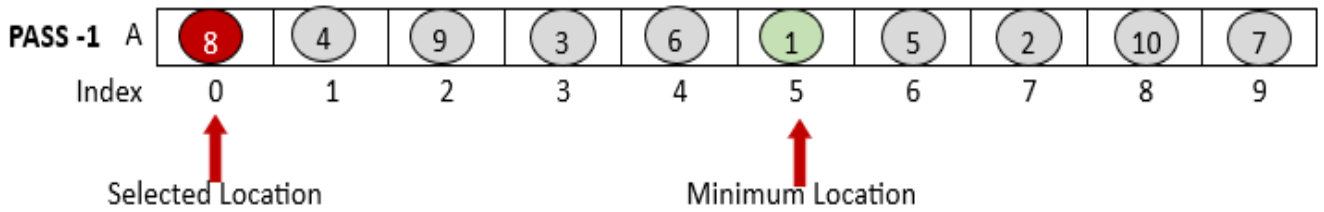
In selection sort, select the location, and identify the element that is to be stored in the selected location

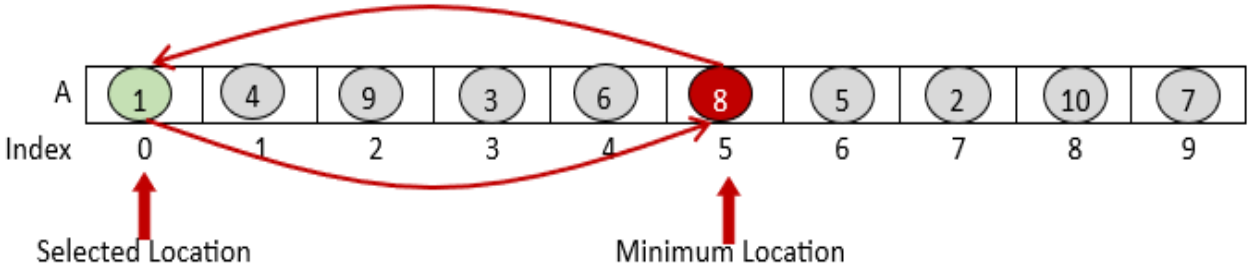
Selection Sort



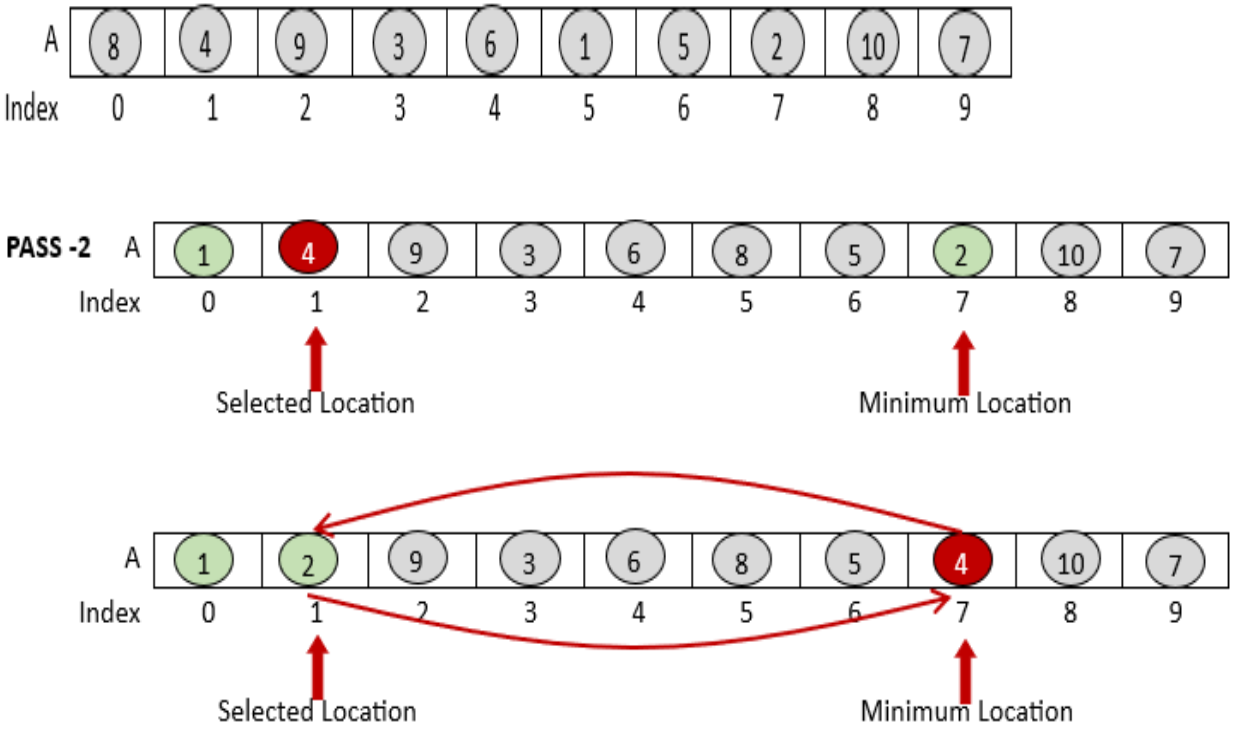
Selection Sort



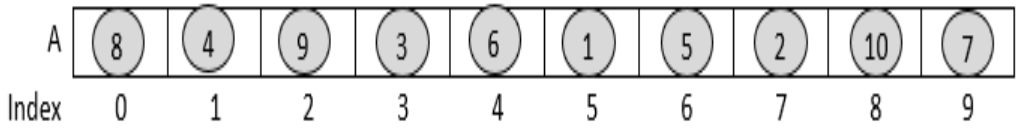


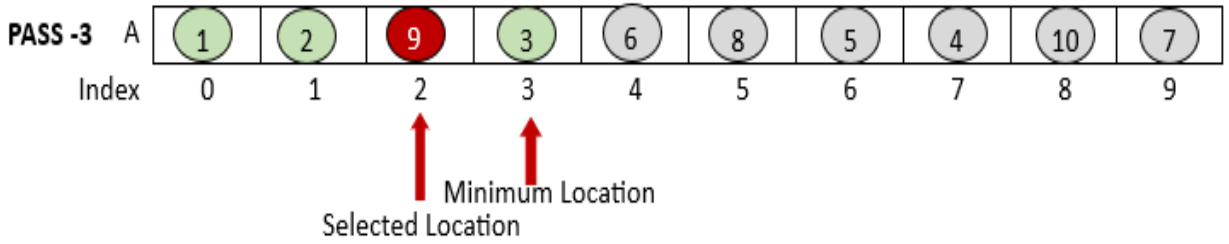


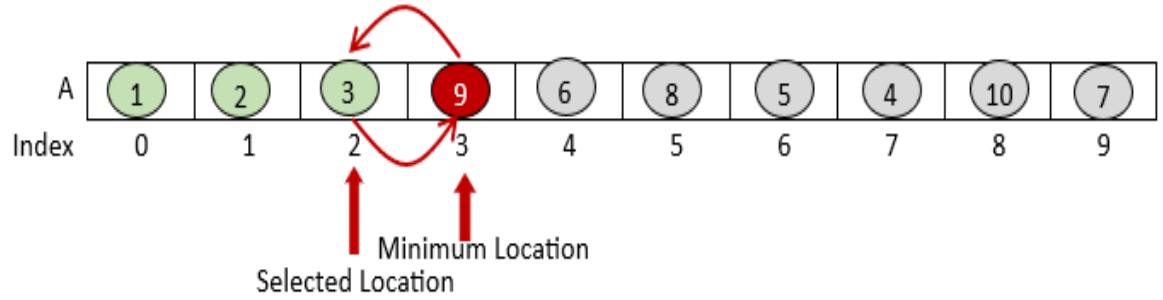
Selection Sort



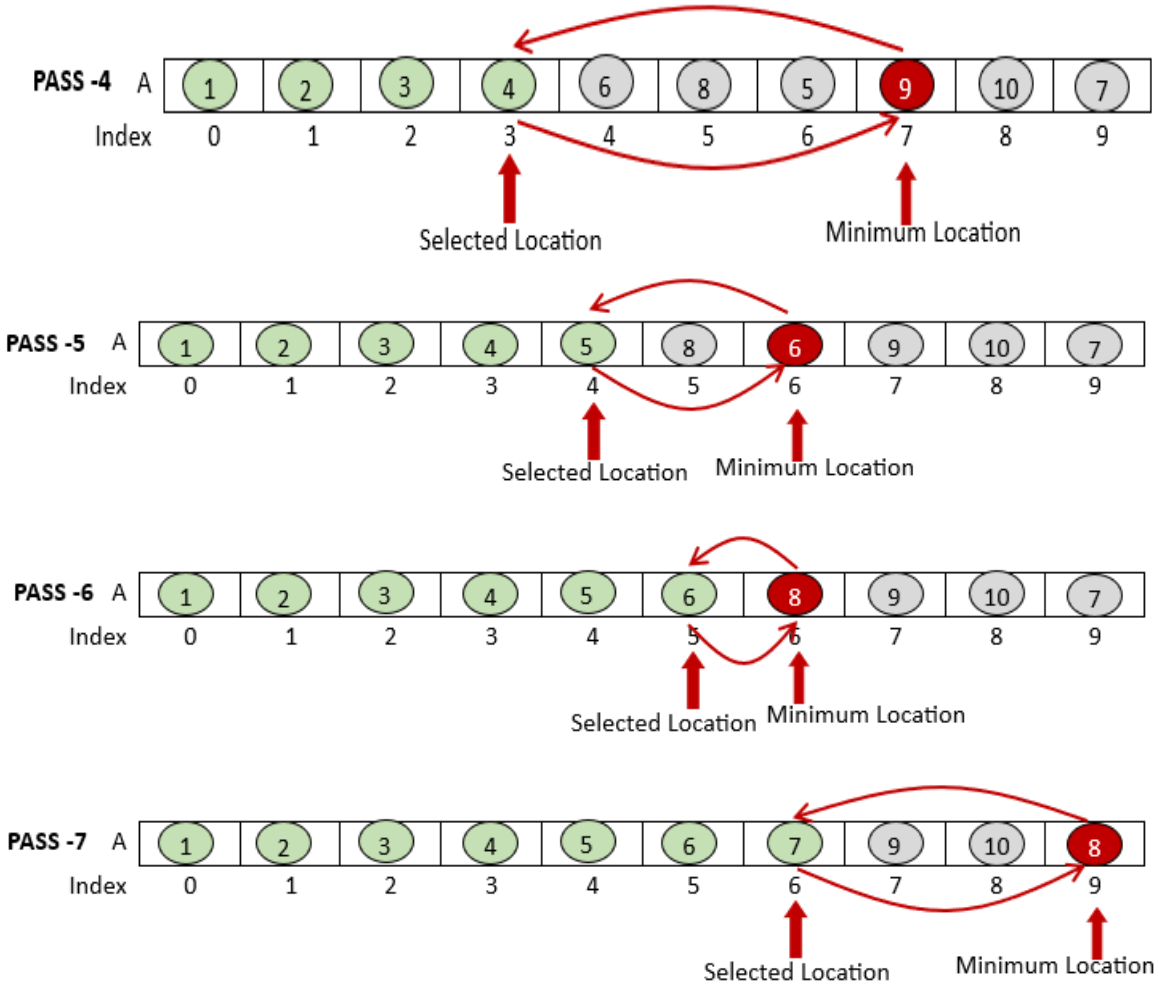
Selection Sort



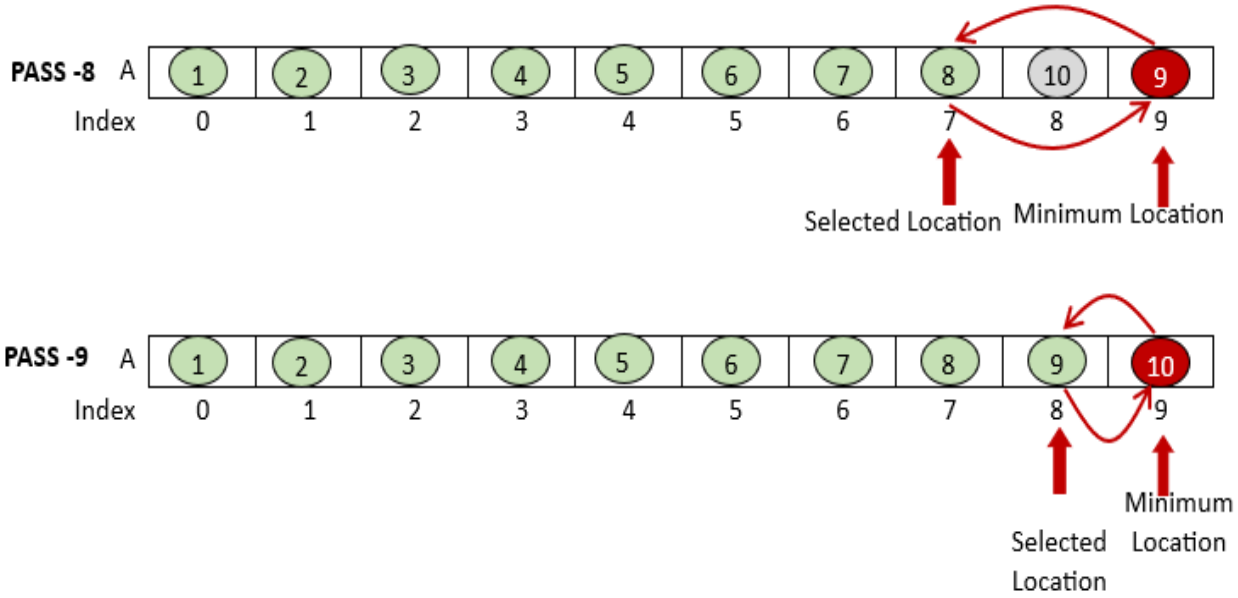




Selection Sort

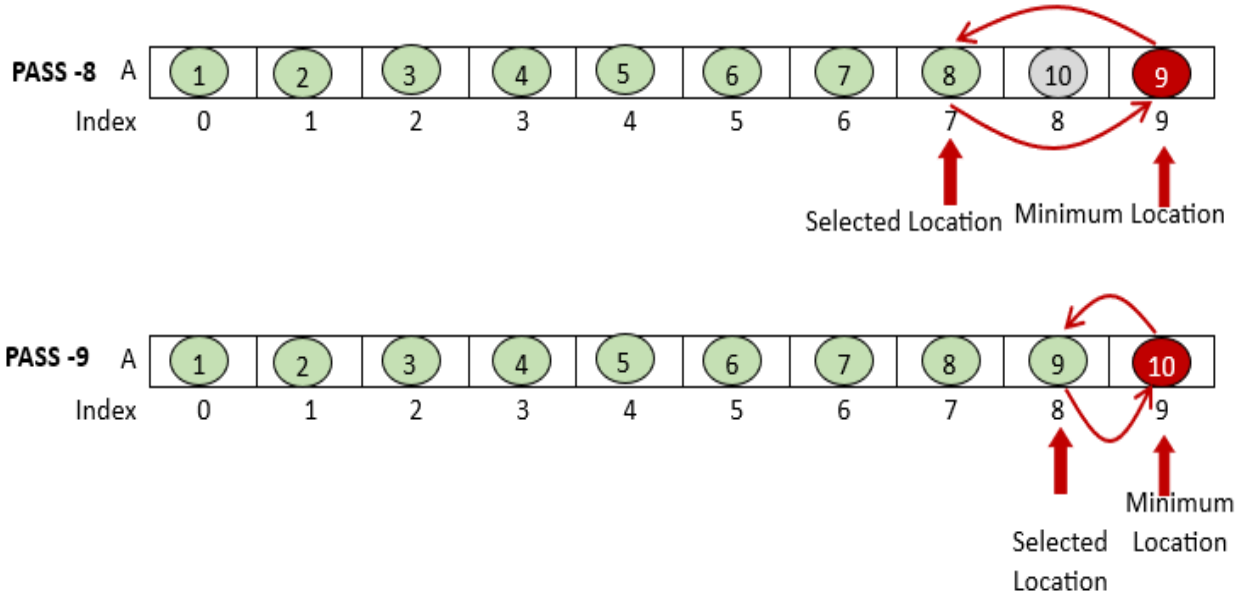


Selection Sort




```
void selectionSort(int A[], int n)
{
    int i, j, min, temp;
    for (i = 0; i < n-1; i++){
        min = i;
        for (j = i+1; j < n; j++){
            if (A[j] < A[min])
                min = j;
        }
        if(min != i){
            temp = A[min];
            A[min] = A[i];
            A[i] = temp;
        }
    }
}
```

Selection Sort




```
void selectionSort(int A[], int n)
{
    int i, j, min, temp;
    for (i = 0; i < n-1; i++){
        min = i;
        for (j = i+1; j < n; j++){
            if (A[j] < A[min])
                min = j;
        }
        if(min != i){
            temp = A[min];
            A[min] = A[i];
            A[i] = temp;
        }
    }
}
```

Time Complexity: (No. of Comparisons)

$$(n-1)+(n-2)+\cdots+3+2+1 = \frac{n(n-1)}{2}$$

$$= O(n^2)$$

Best case, Average Case, Worst Case

Quick Sort

Quick Sort

- Identify an element called **Pivot**

Quick Sort

- Identify an element called **Pivot**
- Find the **location** of the Pivot - the location where the pivot element should have been placed in the sorted array

Quick Sort

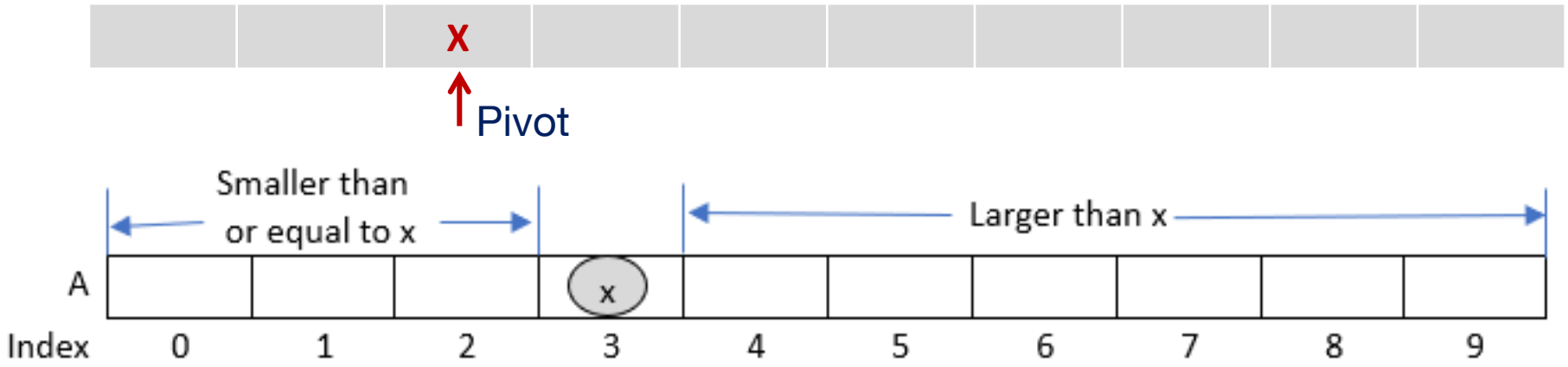
- Identify an element called **Pivot**
- Find the **location** of the Pivot - the location where the pivot element should have been placed in the sorted array
- While finding the location of the Pivot,
  - transfer all the elements smaller than or equal to the pivot before the location of the pivot element
  - transfer all the elements larger than the pivot after the location of the pivot element

Quick Sort

- Identify an element called **Pivot**
- Find the **location** of the Pivot - the location where the pivot element should have been placed in the sorted array
- While finding the location of the Pivot,
  - transfer all the elements smaller than or equal to the pivot before the location of the pivot element
  - transfer all the elements larger than the pivot after the location of the pivot element

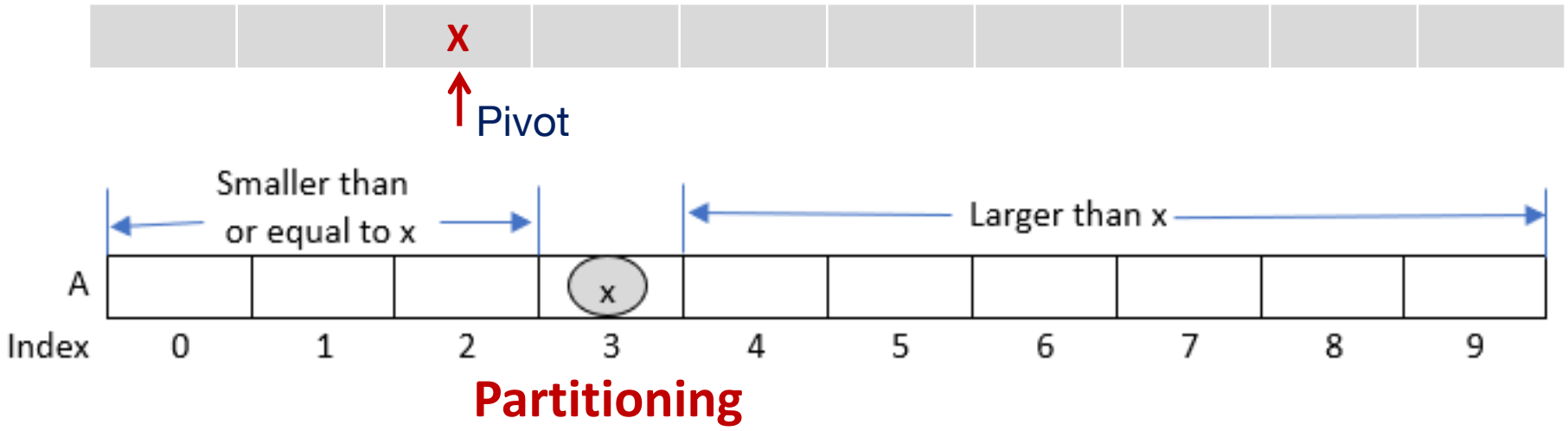
Quick Sort

- Identify an element called **Pivot**
- Find the **location** of the Pivot - the location where the pivot element should have been placed in the sorted array
- While finding the location of the Pivot,
  - transfer all the elements smaller than or equal to the pivot before the location of the pivot element
  - transfer all the elements larger than the pivot after the location of the pivot element



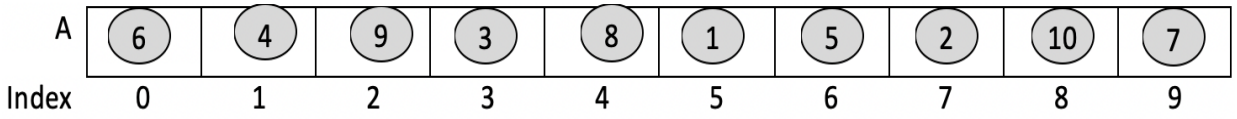
Quick Sort

- Identify an element called **Pivot**
- Find the **location** of the Pivot - the location where the pivot element should have been placed in the sorted array
- While finding the location of the Pivot,
  - transfer all the elements smaller than or equal to the pivot before the location of the pivot element
  - transfer all the elements larger than the pivot after the location of the pivot element

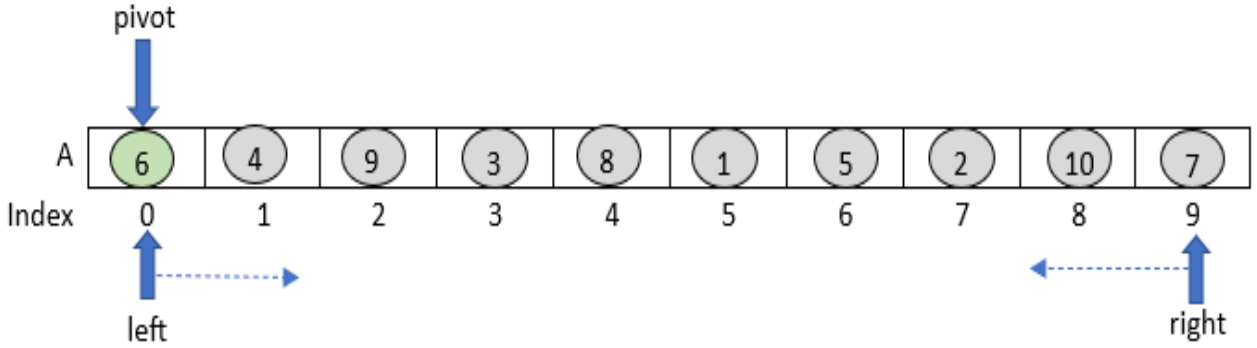


Partitioning

Quick Sort

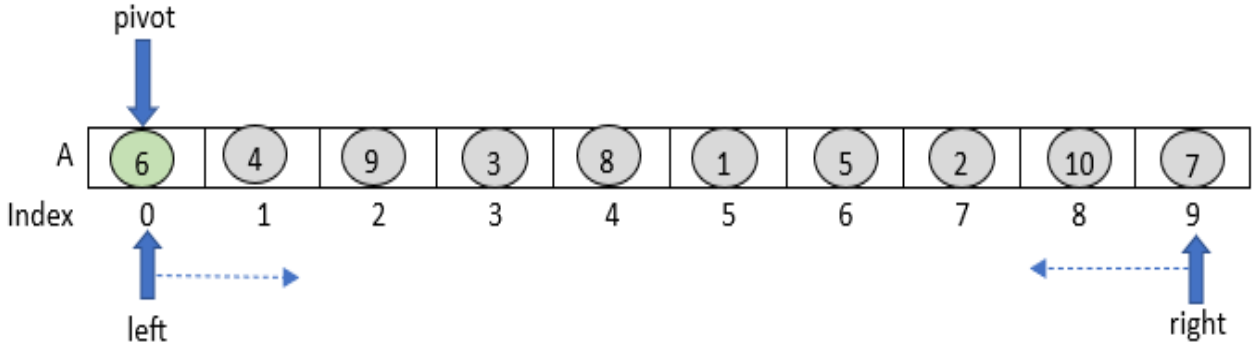


Quick Sort

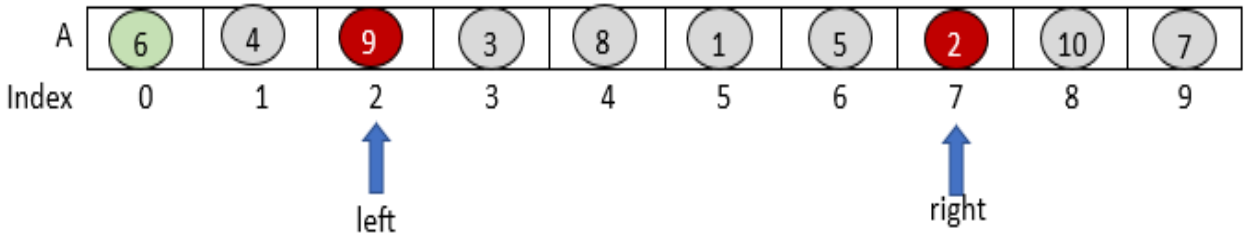


- The left pointer scans the array from left to right direction.
- The right pointer scans the array from right to left direction.
- Whenever the left pointer finds an element larger than the pivot, it stops scanning.
- Whenever the right pointer finds an element smaller than or equal to pivot, it stops scanning.

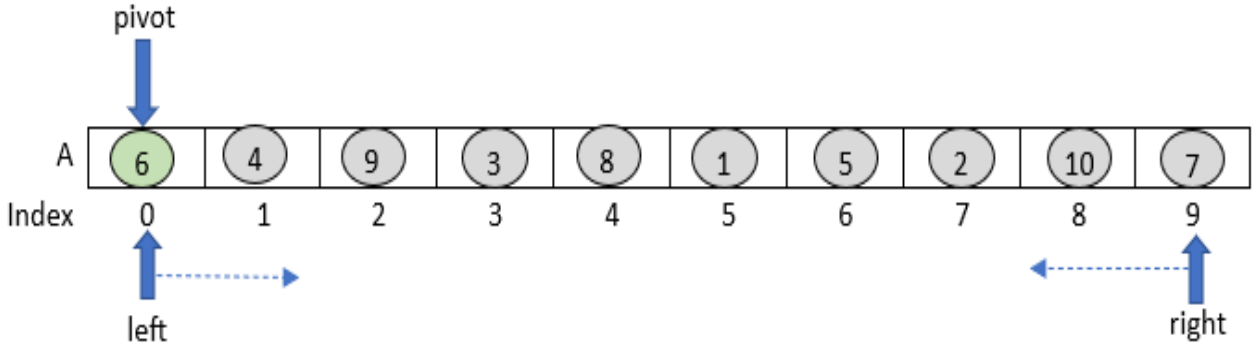
Quick Sort



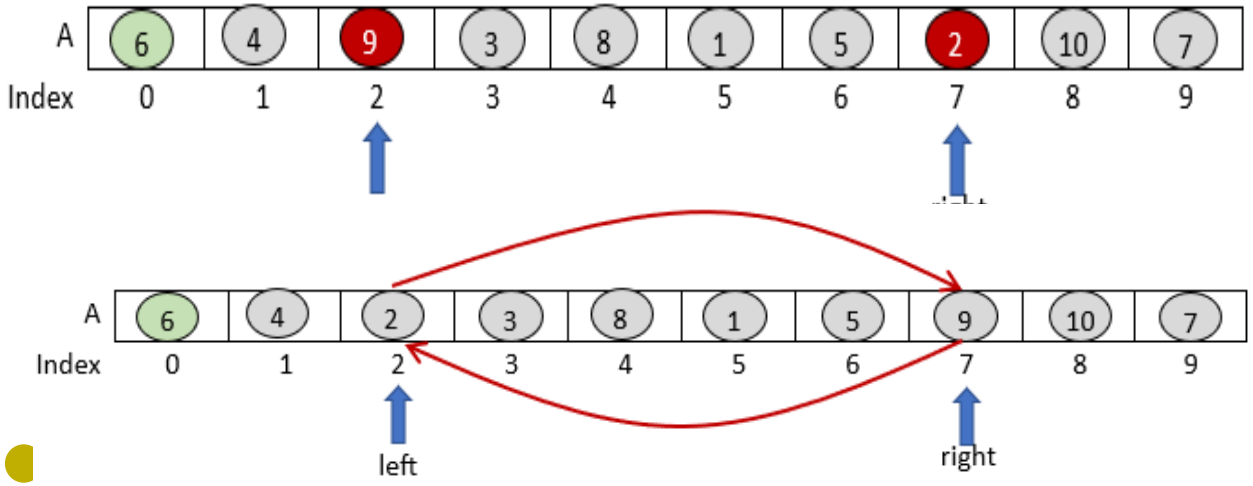
- The left pointer scans the array from left to right direction.
- The right pointer scans the array from right to left direction.
- Whenever the left pointer finds an element larger than the pivot, it stops scanning.
- Whenever the right pointer finds an element smaller than or equal to pivot, it stops scanning.



Quick Sort

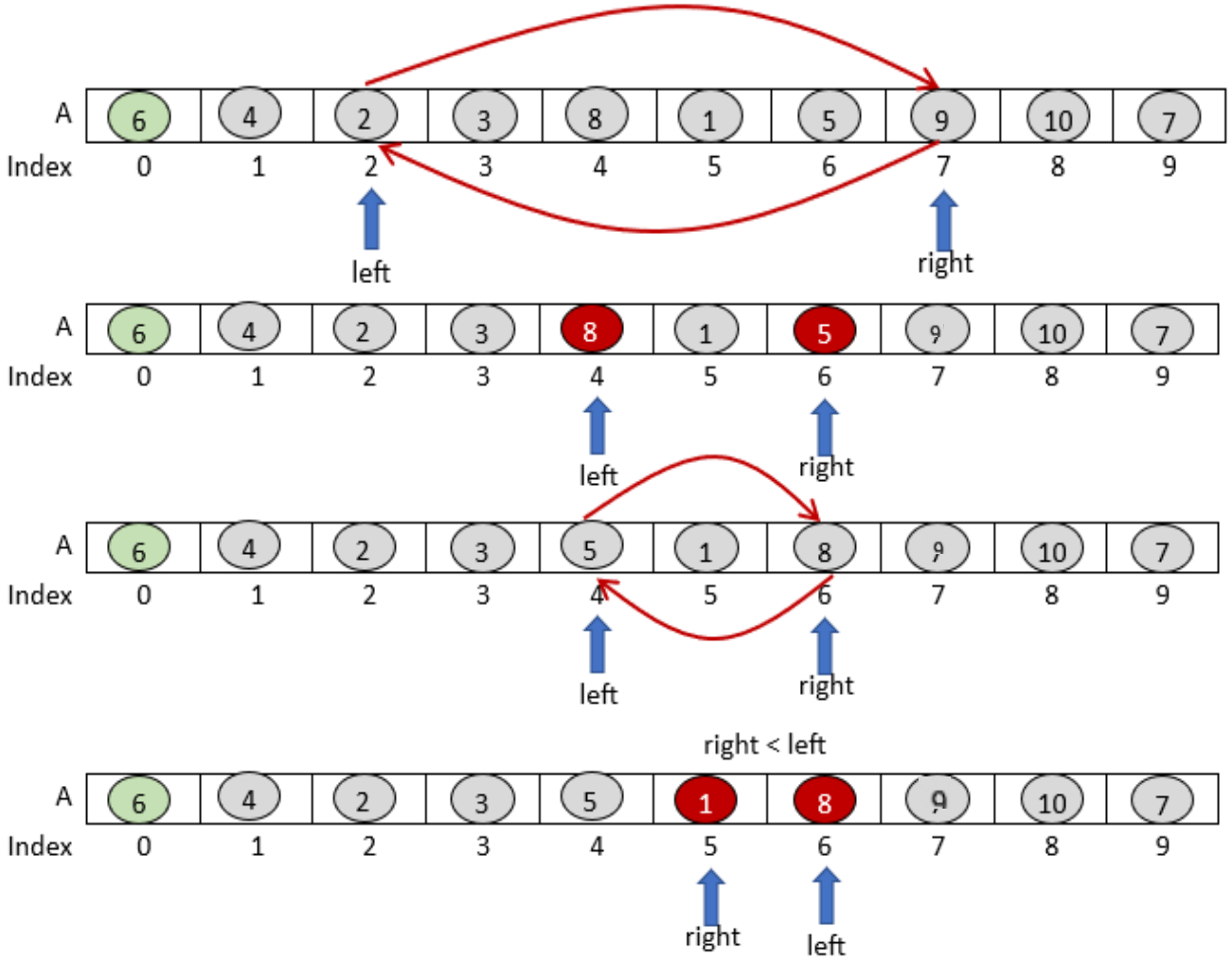


- The left pointer scans the array from left to right direction.
- The right pointer scans the array from right to left direction.
- Whenever the left pointer finds an element larger than the pivot, it stops scanning.
- Whenever the right pointer finds an element smaller than or equal to pivot, it stops scanning.

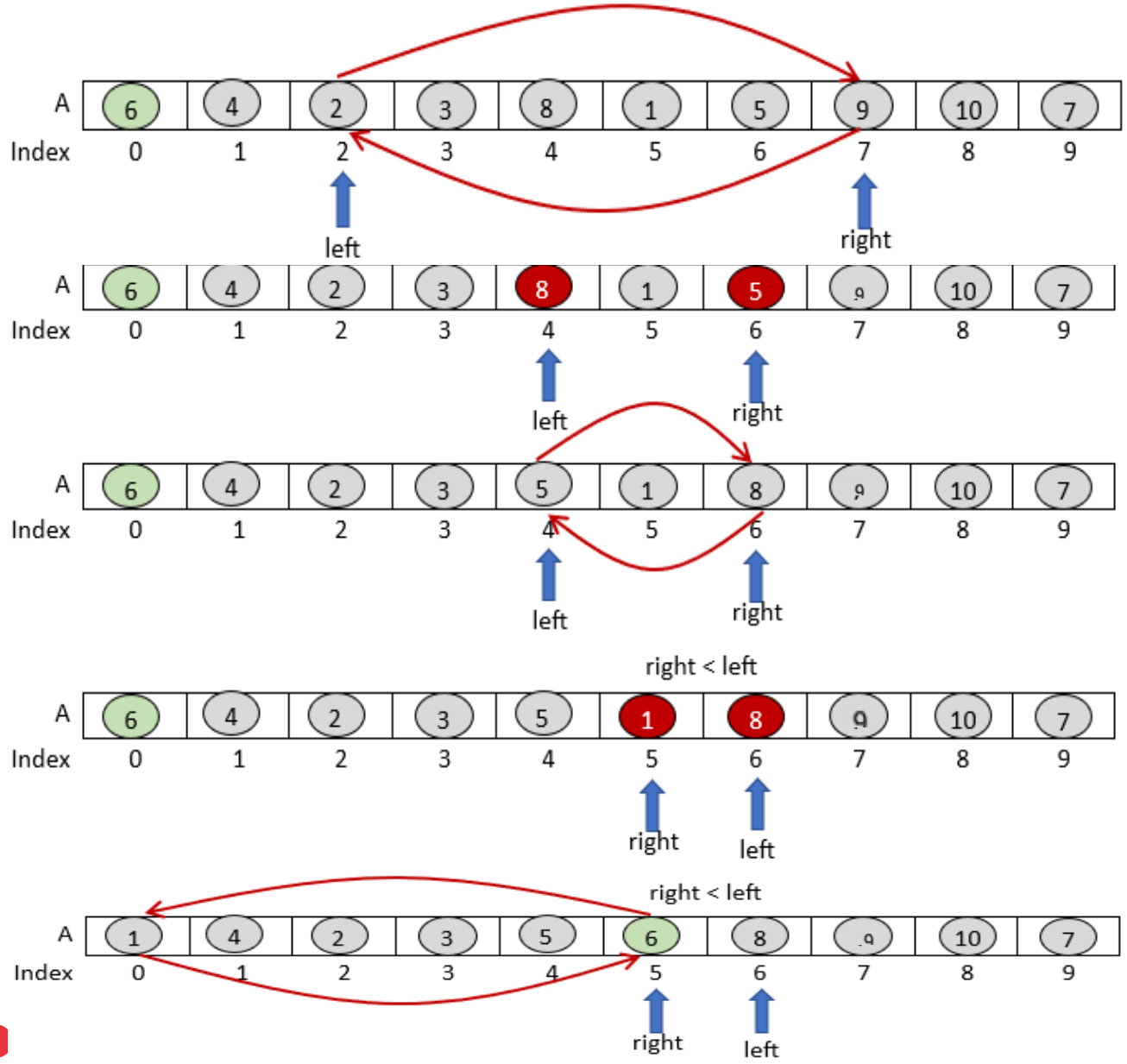


Swap the two elements pointed by ***left*** pointer and ***right*** pointer.

Quick Sort

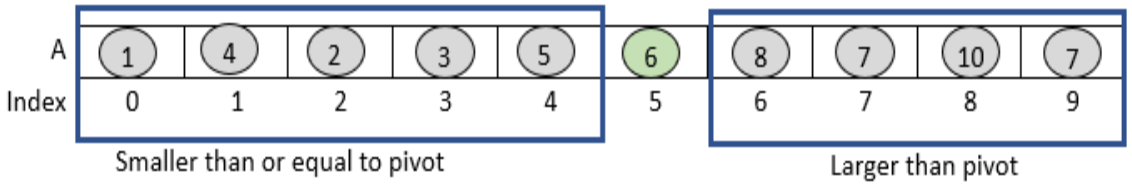

A
6 4 2 3 8 1 5 9 10 7
Index
0 1 2 3 4 5 6 7 8 9
left
right
A
6 4 2 3 8 1 5 9 10 7
Index
0 1 2 3 4 5 6 7 8 9
left
right
A
6 4 2 3 5 1 8 9 10 7
Index
0 1 2 3 4 5 6 7 8 9
left
right
right < left
A
6 4 2 3 5 1 8 9 10 7
Index
0 1 2 3 4 5 6 7 8 9
right
left

Quick Sort



- When the *right* pointer is smaller than the *left* pointer, the **scanning terminates**.
- The pivot element and the element pointed by the *right* pointer are **swapped**

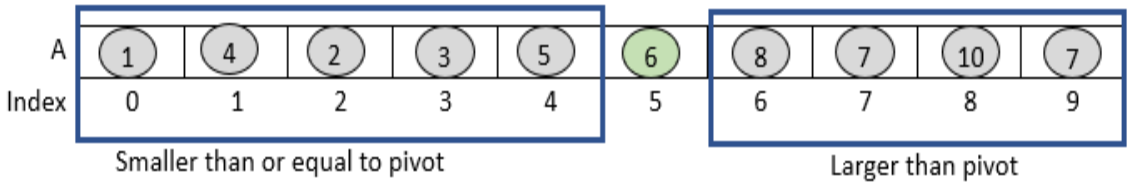
Quick Sort



The array is partitioned into two –

- The left partition holds the elements smaller than or equal to pivot
- The right partition holds the elements larger than the pivot

Quick Sort



The array is partitioned into two –

- The left partition holds the elements smaller than or equal to pivot
- The right partition holds the elements larger than the pivot

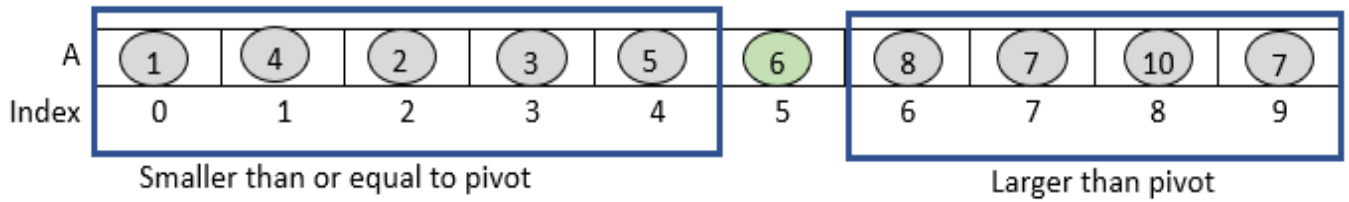
For any partitioning algorithm, as it needs to visit every element in the array at least once, it will have at least $\theta(n)$ time complexity

Quick Sort

```
int partition(int A[], int left, int right){
   int l = left, r = right, temp;
   int pivot =A[left];
   while (l<r){
      while((A[l] <= pivot) &&(l<= right))l++;
      while(A[r] > pivot)r--;
      if (l<r){
            temp = A[l];
            A[l] = A[r];
            A[r] = temp;
      }
   }
   temp = A[left];
   A[left] = A[r];
   A[r] = temp;
   return r;
}
```

Quick Sort

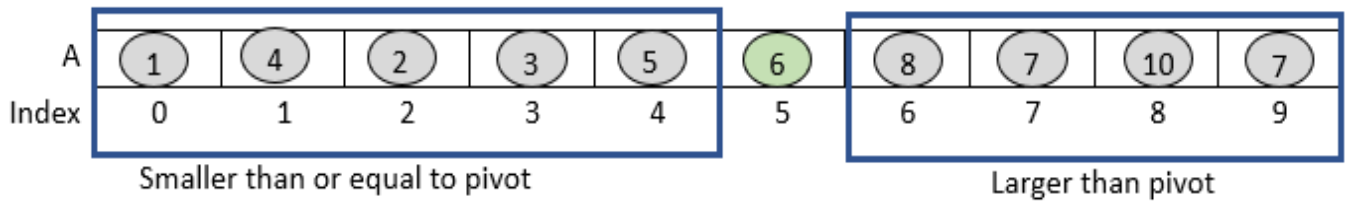
```
int partition(int A[], int left, int right){
   int l = left, r = right, temp;
   int pivot =A[left];
   while (l<r){
      while((A[l] <= pivot) &&(l<= right))l++;
      while(A[r] > pivot)r--;
      if (l<r){
            temp = A[l];
            A[l] = A[r];
            A[r] = temp;
      }
   }
   temp = A[left];
   A[left] = A[r];
   A[r] = temp;
   return r;
}
```



Quick Sort

```
int partition(int A[], int left, int right){
    int l = left, r = right, temp;
    int pivot =A[left];
    while (l<r){
        while((A[l] <= pivot) &&(l<= right))l++;
        while(A[r] > pivot)r--;
        if (l<r){
                temp = A[l];
                A[l] = A[r];
                A[r] = temp;
        }
    }
    temp = A[left];
    A[left] = A[r];
    A[r] = temp;
    return r;
}
```

```
void quickSort(int A[], int left, int right){
   if (left < right){
        int p = partition(A, left, right);
        quickSort(A, left, p - 1);
        quickSort(A, p + 1, right);
   }
}
```



Quick Sort – Time Complexity (Best Case)

$$T(n) = \begin{cases} 2T(n/2) + n, & if\ n > 1 \\ 1 \quad , & if\ n = 1 \end{cases}$$

$$T(n) = 2T(n/2) + n = 2[2T(n/2^2) + n/2] + n = 2^2T(n/2^2) + n + n$$
$$= 2^2[2T(n/2^3) + n/4] + n + n = 2^3T(n/2^2) + n + n + n$$
$$= 2^3[2T(n/2^4) + n/8] + n + n + n = 2^4T(n/2^4) + n + n + n + n$$

.....

.....

$$= 2^k\ T\left(n/2^k\right) + n + \cdots + n + n + n + n$$
$$= 2^k\ T(1) + n + \cdots + n + n + n + n,\ Assume\ that\ n = 2^k$$
$$= n\log_2 n + n = O(n\log_2 n)$$

Quick Sort – Time Complexity (Worst Case)

$$T(n) = \begin{cases} T(n-1) + n, & if\ n > 1 \\ 1 \quad , & if\ n = 1 \end{cases}$$

$$\begin{aligned} & T(n) = T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\ldots\ldots\ldots \\ &= T(1) + 2 + 3 + \cdots + (n-2) + (n-1) + n \\ &= \frac{n(n-1)}{2} = \theta(n^2) \end{aligned}$$

Quick Sort – Time Complexity (Average Case)

Quick Sort – Time Complexity (Average Case)

Quick Sort – Time Complexity (Average Case)

Quick Sort – Time Complexity (Average Case)

Quick Sort – Time Complexity (Average Case)

Quick Sort – Time Complexity (Average Case)

The location of pivot could be at any arbitrary index.

Quick Sort – Time Complexity (Average Case)

The location of pivot could be at any arbitrary index.

Average case is the average of the Time Complexities of the Quick Sort algorithm when pivot divides the array at 0 or 1 or 2, …., or n-1 indices.

Quick Sort – Time Complexity (Average Case)

The location of pivot could be at any arbitrary index.

Average case is the average of the Time Complexities of the Quick Sort algorithm when pivot divides the array at 0 or 1 or 2, …., or n-1 indices.

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n), & if\ n > 1 \\ 1, & if\ n = 1 \\ 0, & if\ n = 0 \end{cases}$$

Where I is the position of the pivot, and n is the size of the array

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n), & if\ n > 1 \\ 1, & if\ n = 1 \\ 0, & if\ n = 0 \end{cases}$$

When i = 0 or i = n-1 in every subsequent partitions , it is the worst case scenario.

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n), & if\ n > 1 \\ 1, & if\ n = 1 \\ 0, & if\ n = 0 \end{cases}$$

When i = 0 or i = n-1 in every subsequent partitions , it is the worst case scenario.

When i = n/2 in every subsequent partitions, it is best case scenario.

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n), & if\ n > 1 \\ 1, & if\ n = 1 \\ 0, & if\ n = 0 \end{cases}$$

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n)$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n), & if\ n > 1 \\ 1, & if\ n = 1 \\ 0, & if\ n = 0 \end{cases}$$

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n)$$

$$T(0) + T(1) + T(2) + \cdots .. + T(n-1)$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n), & if\ n > 1 \\ 1, & if\ n = 1 \\ 0, & if\ n = 0 \end{cases}$$

$$T(n-1) + T(n-2) + \cdots .. + T(2) + T(1) + T(0)$$

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n)$$

$$T(0) + T(1) + T(2) + \cdots .. + T(n-1)$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n), & if\ n > 1 \\ 1, & if\ n = 1 \\ 0, & if\ n = 0 \end{cases}$$

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1) + n)$$

$$\Rightarrow T(n) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}(T(i)+T(n-i-1)+n), & if\ n>1 \\ 1, & if\ n=1 \\ 0, & if\ n=0 \end{cases}$$

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i)+T(n-i-1)+n)$$

$$\Rightarrow T(n) = \frac{2}{n}\sum_{i=0}^{n-1}T(i)+n$$

$$\Rightarrow T(n) = \frac{2}{n}\big(T(0)+T(1)+T(2)+\cdots\ldots+T(n-2)+T(n-1)\big)+n$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i)+T(n-i-1)\big)+n, & if\ n>1 \\ 1 \quad , & if\ n=1 \\ 0 \quad , & if\ n=0 \end{cases}$$

$$T(n) = \frac{2}{n}\big(T(0)+T(1)+T(2)+\cdots\ldots+T(n-2)+T(n-1)\big)+n$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i)+T(n-i-1)\big)+n, & if\ n>1 \\ 1 \quad , & if\ n=1 \\ 0 \quad , & if\ n=0 \end{cases}$$

$$T(n) = \frac{2}{n}\big(T(0)+T(1)+T(2)+\cdots\ldots+T(n-2)+T(n-1)\big)+n$$

$$\Rightarrow nT(n) = 2\big(T(0)+T(1)+T(2)+\cdots\ldots+T(n-2)+T(n-1)\big)+n^2 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\text{(A)}$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i)+T(n-i-1)\big)+n, & if\ n>1 \\ 1 \quad , & if\ n=1 \\ 0 \quad , & if\ n=0 \end{cases}$$

$$T(n) = \frac{2}{n}\big(T(0)+T(1)+T(2)+\cdots\ldots+T(n-2)+T(n-1)\big)+n$$

$$\Rightarrow nT(n) = 2\big(T(0)+T(1)+T(2)+\cdots\ldots+T(n-2)+T(n-1)\big)+n^2 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\text{(A)}$$

$$\Rightarrow (n-1)T(n-1) = 2\big(T(0)+T(1)+T(2)+\cdots\ldots+T(n-3)+T(n-2)\big)+(n-1)^2 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\text{(B)}$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i)+T(n-i-1)\big)+n, & if\ n>1 \\ 1 \quad , & if\ n=1 \\ 0 \quad , & if\ n=0 \end{cases}$$

$$T(n) = \frac{2}{n}\big(T(0)+T(1)+T(2)+\cdots\ldots+T(n-2)+T(n-1)\big)+n$$

$$\Rightarrow nT(n) = 2\big(T(0)+T(1)+T(2)+\cdots\ldots+T(n-2)+T(n-1)\big)+n^2 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\text{(A)}$$

$$\Rightarrow (n-1)T(n-1) = 2\big(T(0)+T(1)+T(2)+\cdots\ldots+T(n-3)+T(n-2)\big)+(n-1)^2 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\text{(B)}$$

Equation A -B

$$nT(n)-(n-1)T(n-1) = 2T(n-1)+2n-1 \approx 2T(n-1)+2n$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i)+T(n-i-1)\big)+n, & if\ n>1 \\ 1 \quad , \quad if\ n=1 \\ 0 \quad , \quad if\ n=0 \end{cases}$$

$$\mathrm{nT(n)} - (\mathrm{n}-1)T(n-1) = 2T(n-1) + 2n - 1 \approx 2T(n-1) + 2n$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i)+T(n-i-1)\big)+n, & if\ n>1 \\ 1 \quad , & if\ n=1 \\ 0 \quad , & if\ n=0 \end{cases}$$

$$\mathrm{nT(n)} - (\mathrm{n}-1)T(n-1) = 2T(n-1) + 2n - 1 \approx 2T(n-1) + 2n$$

$$\Rightarrow \mathrm{nT(n)} \approx (\mathrm{n}+1)T(n-1) + 2n$$

Quick Sort – Time Complexity (Average Case)

$$T(n) = \begin{cases} \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i)+T(n-i-1)\big)+n, & if\ n>1 \\ 1 \quad , & if\ n=1 \\ 0 \quad , & if\ n=0 \end{cases}$$

$$\mathrm{nT(n)} - (\mathrm{n}-1)T(n-1) = 2T(n-1) + 2n - 1 \approx 2T(n-1) + 2n$$

$$\Rightarrow \mathrm{nT(n)} \approx (\mathrm{n}+1)T(n-1) + 2n$$

$$\Rightarrow \frac{T(n)}{n+1} \approx \frac{T(n-1)}{n} + \frac{2}{n+1}$$

Quick Sort – Time Complexity (Average Case)

$$\frac{T(n)}{n+1} \approx \frac{T(n-1)}{n} + \frac{2}{n+1}$$

Quick Sort – Time Complexity (Average Case)

$$\frac{T(n)}{n+1} \approx \frac{T(n-1)}{n} + \frac{2}{n+1}$$

$$\Rightarrow \frac{T(n)}{n+1} \approx \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

Quick Sort – Time Complexity (Average Case)

$$\frac{T(n)}{n+1} \approx \frac{T(n-1)}{n} + \frac{2}{n+1}$$

$$\Rightarrow \frac{T(n)}{n+1} \approx \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$\Rightarrow \frac{T(n)}{n+1} \approx \frac{T(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

Quick Sort – Time Complexity (Average Case)

$$\frac{T(n)}{n+1} \approx \frac{T(n-1)}{n} + \frac{2}{n+1}$$

$$\Rightarrow \frac{T(n)}{n+1} \approx \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$\Rightarrow \frac{T(n)}{n+1} \approx \frac{T(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$\Rightarrow \frac{T(n)}{n+1} \approx \frac{T(0)}{1} + \frac{2}{2} + \frac{2}{3} + \cdots . + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \approx 2\sum_{k=1}^{n} \frac{1}{k} \approx \int_{1}^{n} \frac{1}{x} dx \approx 2\log n$$

Quick Sort – Time Complexity (Average Case)

$$\frac{T(n)}{n+1} \approx 2 \log n$$

$$\Rightarrow \mathrm{T(n)} \approx 2(n+1) \log n$$

Average Time complexity is $\mathbf{O(n \log n)}$

Summary

Algorithm	Internal	Stable	In-place	Adaptive	Comparison
Bubble	Yes	Yes	Yes	No (could be modified to be adaptive)	Yes
Insertion	Yes	Yes	Yes	Yes	Yes
Selection	Yes	No	Yes	No	Yes
Quick	Yes	No	Yes	No (could be modified to be adaptive)	Yes