TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Data structures and Algorithms

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Course outline

Chapter 1. Fundamentals

Chapter 2. Algorithmic paradigms

Chapter 3. Basic data structures

Chapter 4. Tree

Chapter 5. Sorting

Chapter 6. Searching

Chapter 7. Graph

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 5. Sorting

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Sorting problem

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

- The data need to be sorted could be:
  - integers/float/…
  - Character strings
  - …
- Sort key
  - Is a field of a record that determines the sorted order of the record in a set of records.
  - We need to order records in the order of key.
  - Example: key is `total = math + physics + chemist`

```
struct thisinh{
     char *ID;
     struct grade{
         float math, physics, chemist, total;
     };
};

struct node{
    thisinh data;
    node* next;
};
```

Sorting problem

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

- Different types of sorting algorithms:

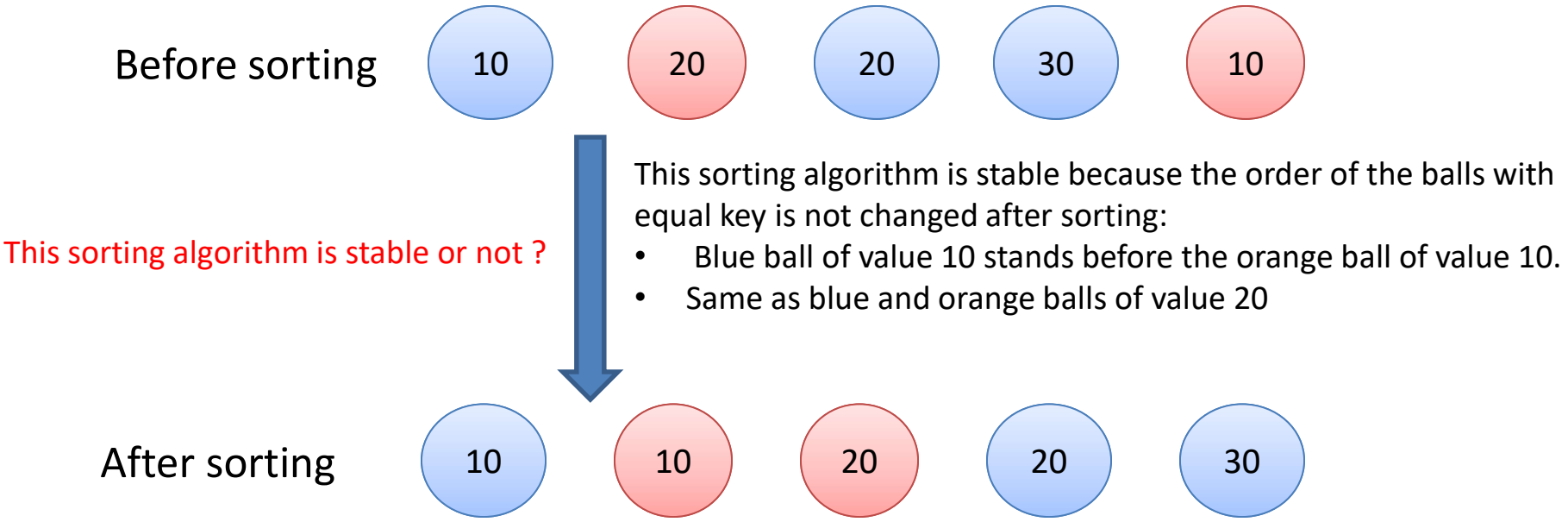
1) **I*nternal sort*** requires that the collection of data fit entirely in the computer's main memory.

2) We can use an ***external sort*** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.

3) ***Parallel sort*** : uses multiple processors to sort, thus reduces the computation time

Properties of Sorting problem

- In place
  - Sorting of a data structure does not require any external data structure for storing the intermediate steps
- 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 problem

- There are 2 basic operations that sorting algorithms often use:
  - **Exchange positions of 2 elements** (Swap): running time $O(1)$

```
void swap( datatype *a, datatype *b){
   datatype *temp = *a;  //datatype- data type of element
   *a = *b;
   *b = *temp;
}
```

```
void swap(int *a, int *b){
   int *temp = *a;
   *a = *b;
   *b = *temp;
}
```

  - **Compare:** Compare(a, b) returns true if a is put before b in the sorted order, false otherwise

- **Sorting analysis**: In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).

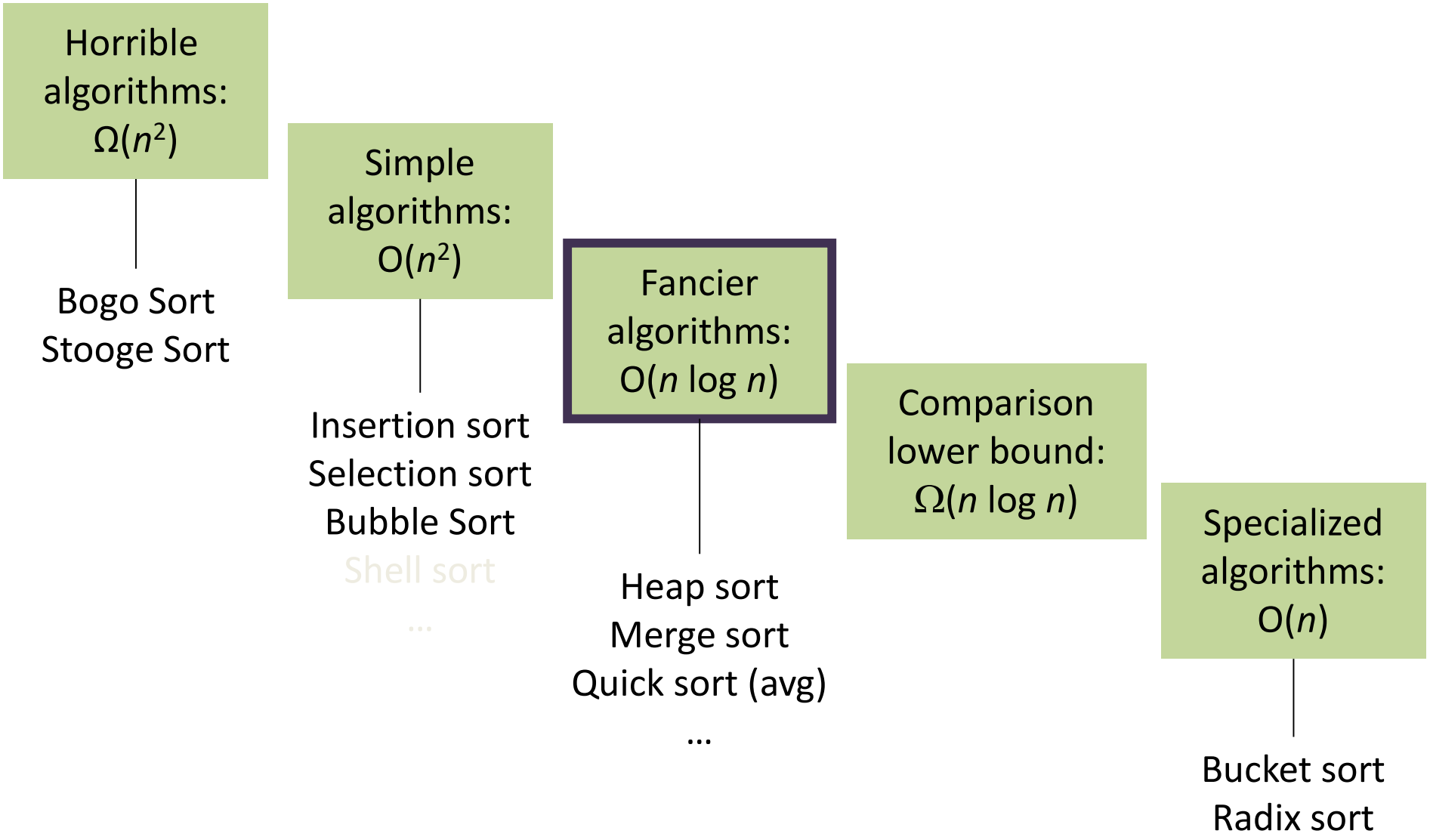
➔ So, to analyze a sorting algorithm we should count the number of data comparisons and the number of moves.

(Ignoring other operations does not affect our final result).

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Contents

Sorting: The Big Picture



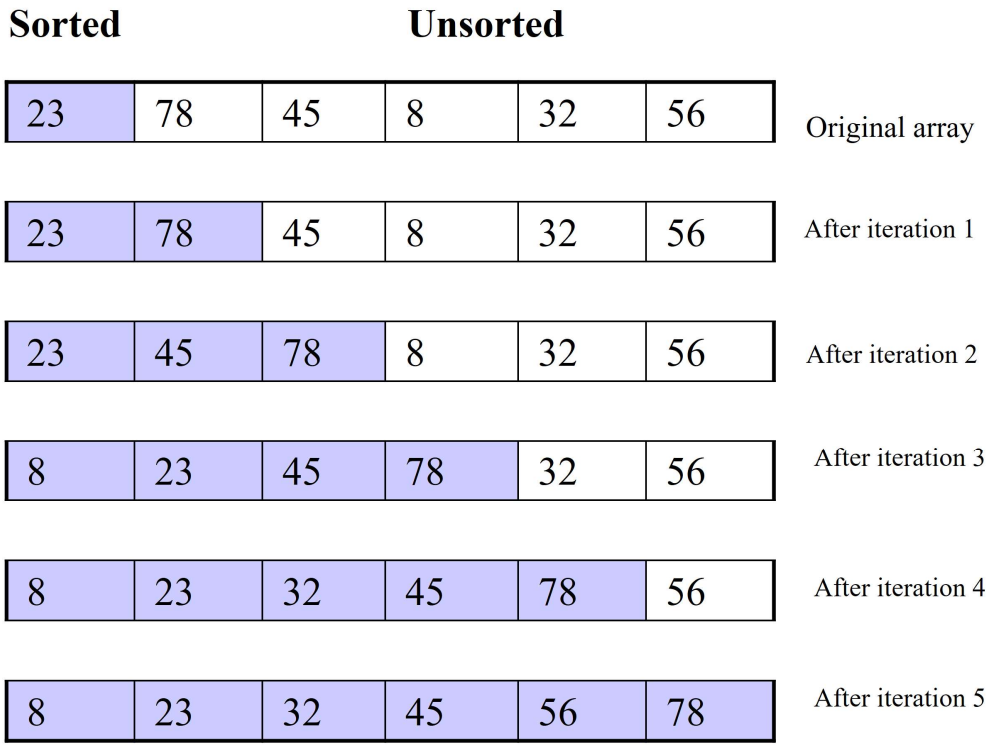
Contents

5.1. Insertion sort

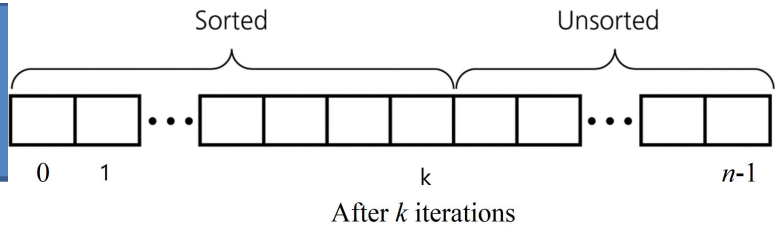
Algorithm:

- The array is divided into two subarrays, *sorted* and *unsorted*
  - Elements in sorted part: is already ordered
  - Elements in unsorted part: is not ordered.
- Each iteration: the first element of the unsorted subarray is picked up, transferred to the sorted subarray, and inserted at the appropriate place.

➔ An array of n elements requires n-1 iterations to completely sort the array.



5.1. Insertion sort



Algorithm:

- The array is divided into two subarrays, *sorted* and *unsorted*.
- Each iteration: the first element of the unsorted subarray is picked up, transferred to the sorted subarray, and inserted at the appropriate place.

➔An array of n elements requires n-1 iterations to completely sort the array.

- Iteration $k = 1, 2, ..., n-1$:

  move the element $A[k]$ to place such that after moving: the first $(k+1)$ elements of array A is ordered

At each iteration *k*, it might need more than one time to swap elements to put element *A*[*k*] to place such that the first *k* elements of array A is ordered.

Iteration *k*: Repeatedly swap element k^{th} with the one to its left if smaller.

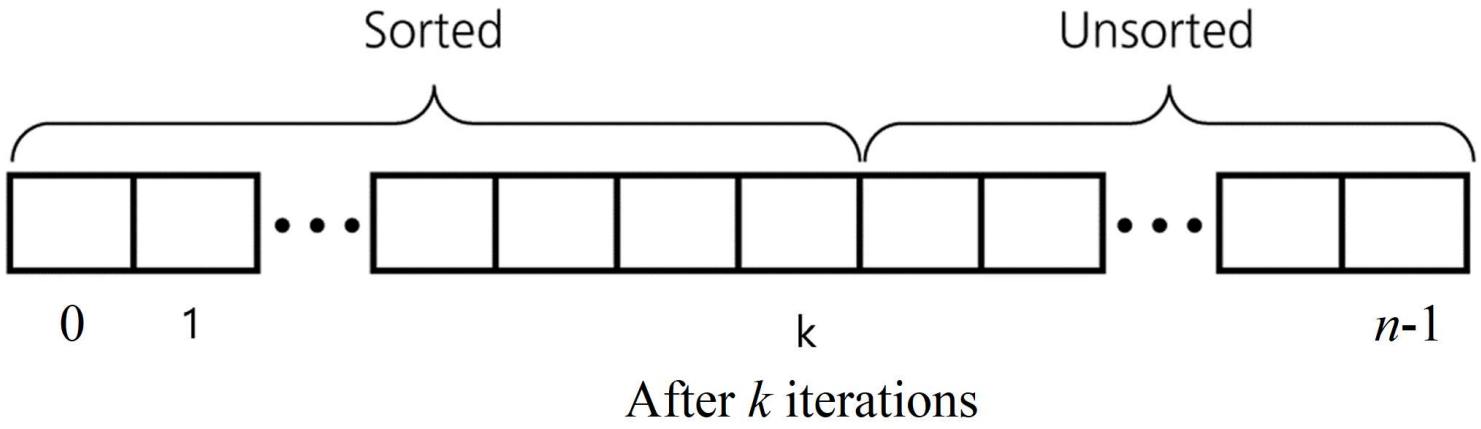
5.1. Insertion sort

- Iteration $k = 1, 2, ..., n-1$:

  move the element $A[k]$ to place such that after moving: the first $(k+1)$ elements of array A is ordered

At each iteration k, it might need more than one time to swap elements to put element $A[k]$ to place such that the first k elements of array A is ordered.

Iteration k: Repeatedly swap element k^{th} with the one to its left if smaller.



After k iterations

Properties: Ater k^{th} iteration , $A[0]$ through $A[k]$ contain first $k+1$ elements in ascending order

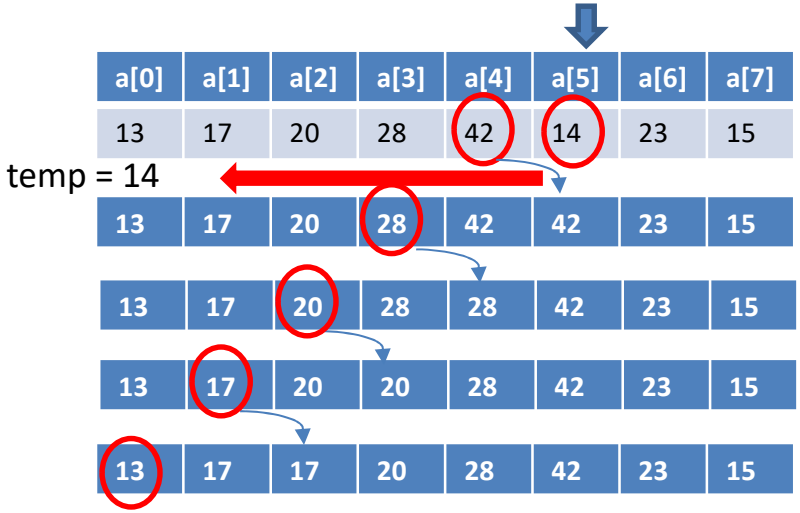
Implementation: Insertion Sort Algorithm

```
void insertionSort(int a[], int size);
```

- **Algorithm:**
  - Iteration $k = 1, 2, ..., n-1$:
    move the element $A[k]$ to place such that after moving: the first $(k+1)$ elements of array A is ordered

At each iteration *k*, it might need more than one time to swap elements to put element *A*[*k*] to place such that the first *k* elements of array A is ordered.

Iteration *k*: Repeatedly swap element k^{th} with the one to its left if smaller.




```
for (int k = 1; k < size; k++) {
        int temp = a[k];
        int pos = k;
        /* Iteration k: repeatedly swap element kth with the one to its left
                        if smaller*/
        while (pos > 0 && a[pos-1] > temp) {
             a[pos] = a[pos-1];
             pos--;
        } // end while


}
```

Implementation: Insertion Sort Algorithm

k=5: find place to put a[5]=14 so that elements from a[0] to a[5] are sorted

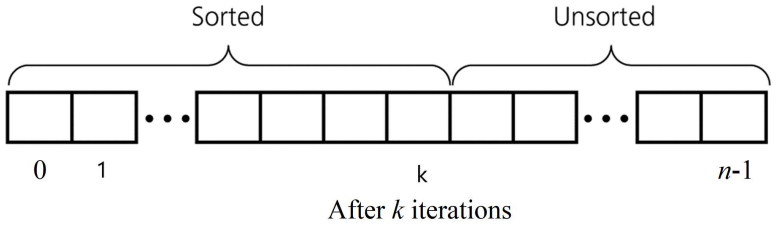
```
void insertionSort(int a[], int size);
```

- **Algorithm:**
  - Iteration $k = 1, 2, ..., n-1$:

    move the element $A[k]$ to place such that after moving: the first $(k+1)$ elements of array A is ordered

At each iteration *k*, it might need more than one time to swap elements to put element *A*[*k*] to place such that the first *k* elements of array A is ordered.

Iteration *k*: Repeatedly swap element k^{th} with the one to its left if smaller.



After *k* iterations

After iteration *k* = 5: a[0]…a[5] are sorted unsorted

```
for (int k = 1; k < size; k++) {
        int temp = a[k];
        int pos = k;
        /* Iteration k: repeatedly swap element kth with the one to its left
                        if smaller*/
        while (pos > 0 && a[pos-1] > temp) {
             a[pos] = a[pos-1];
             pos--;
        } // end while
        // Insert the value temp (=a[k]) into correct place
        a[pos] = temp;
}
```

Implementation: Insertion Sort Algorithm

```
void insertionSort(int a[], int size) {
  int k, pos, temp;
  for (k=1; k < size; k++)  {
      temp = a[k];
      pos = k;
      while ((pos > 0) && (a[pos-1] > temp))   {
            a[pos] = a[pos-1];
            pos = pos - 1;  }
      a[pos] = temp;
  }
}


void main()
{
    int a[5] = {8,4,3,2,1};
    insertionSort(a,5);
    for (int i = 0; i<5; i++)
       printf("%d \n",a[i]);
}
```

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

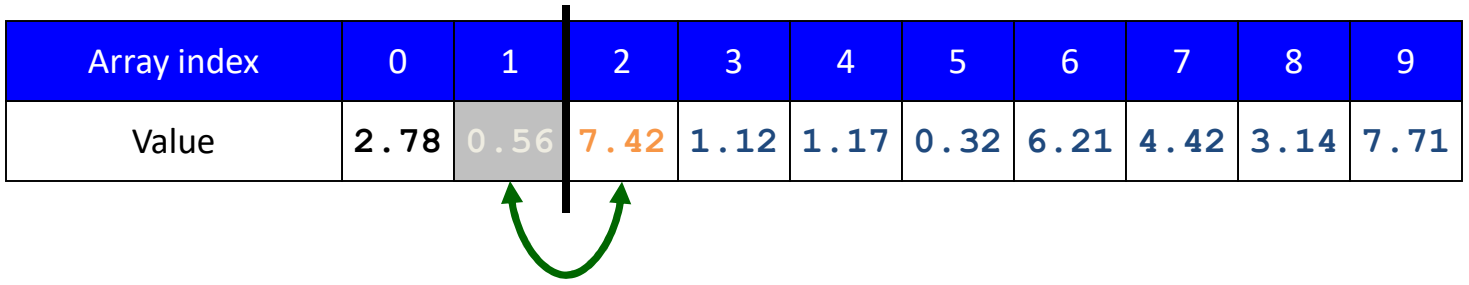
Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 1: step 0.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

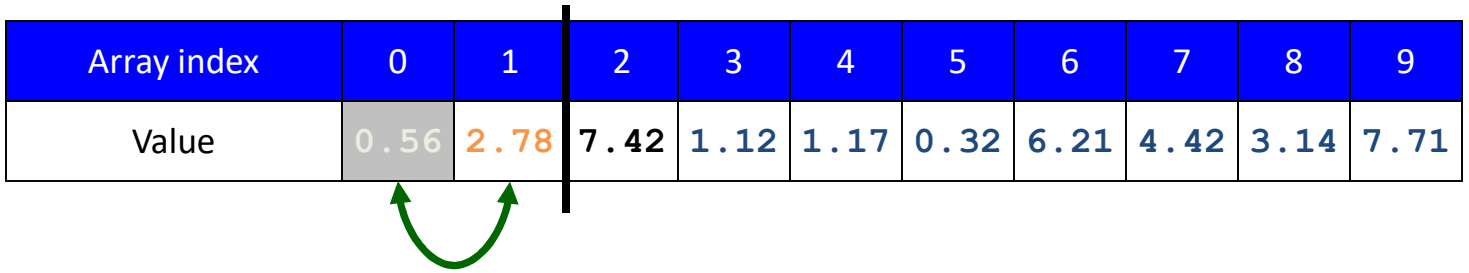


Iteration 2: step 0.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order



Iteration 2: step 1.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

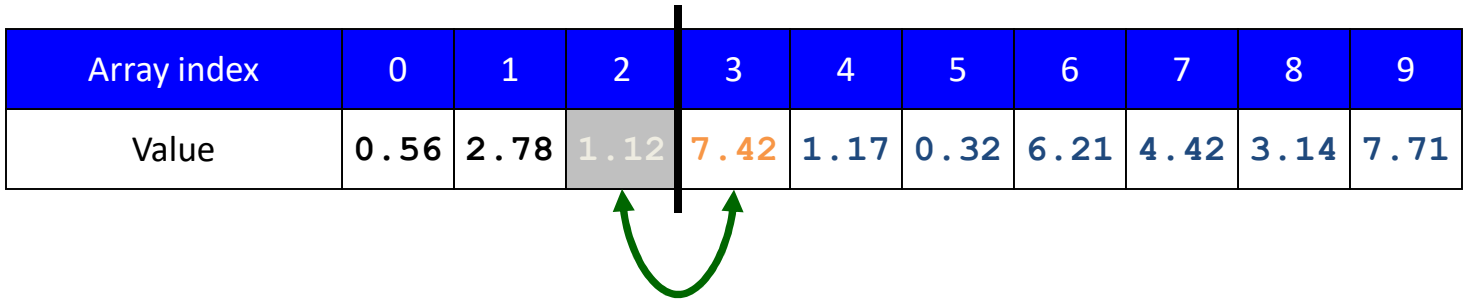
Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 2.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

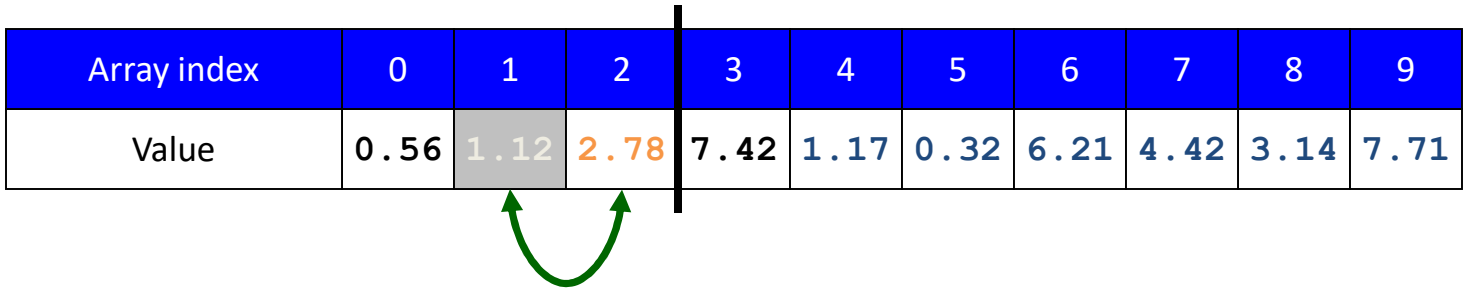


Iteration 3: step 0.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order



Iteration 3: step 1.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

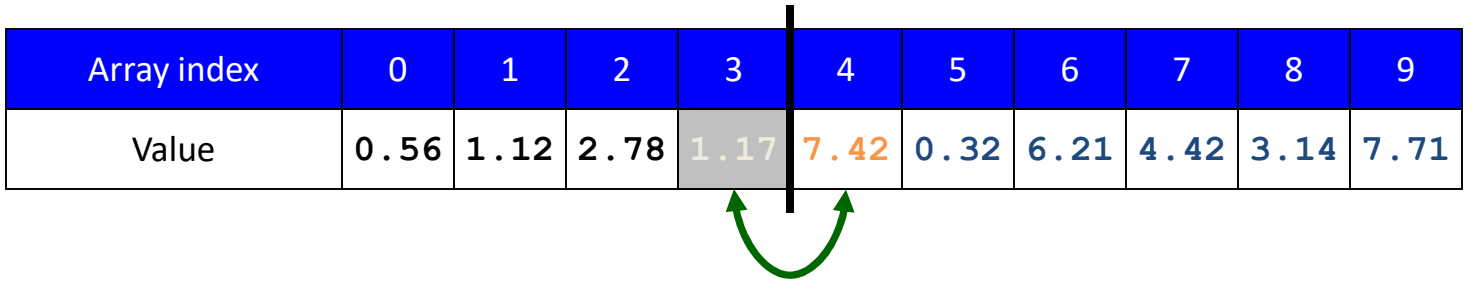
Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 2.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

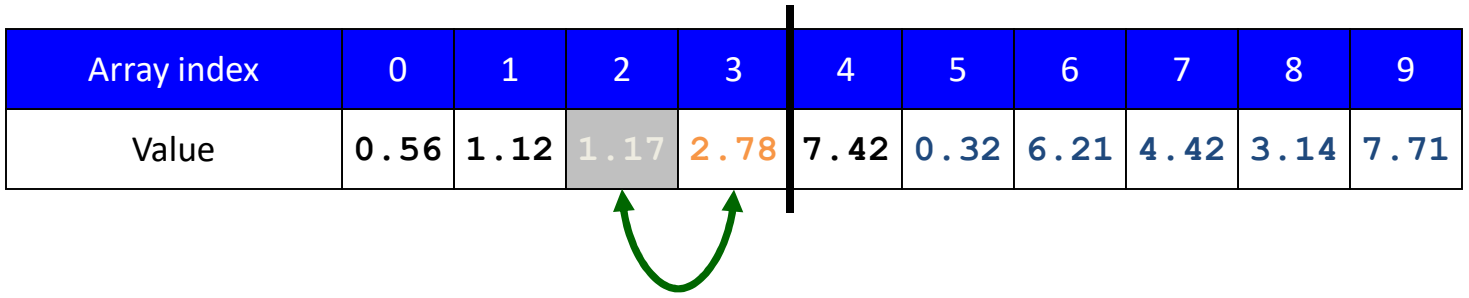


Iteration 4: step 0.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order



Iteration 4: step 1.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

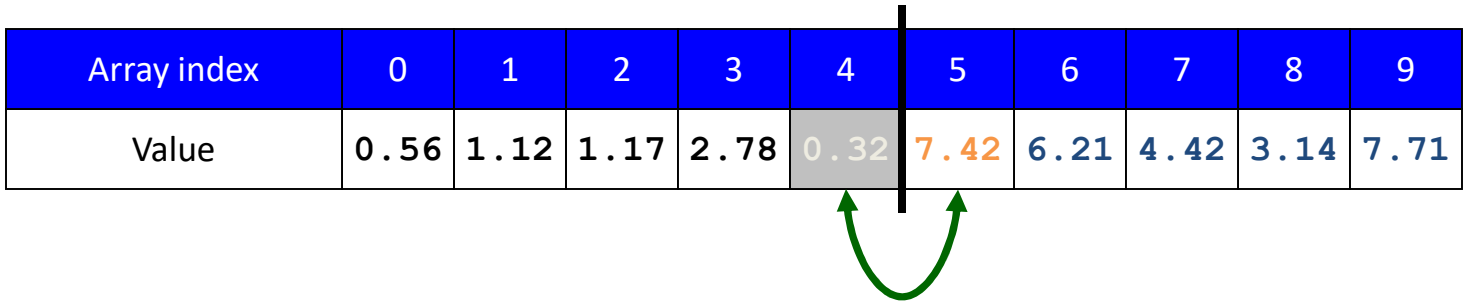
Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	7.42	0.32	6.21	4.42	3.14	7.71

Iteration 4: step 2.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

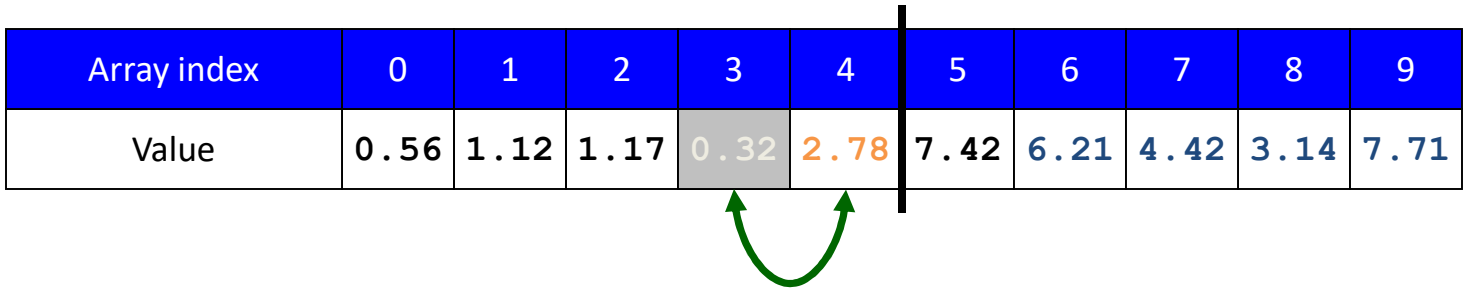


Iteration 5: step 0.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

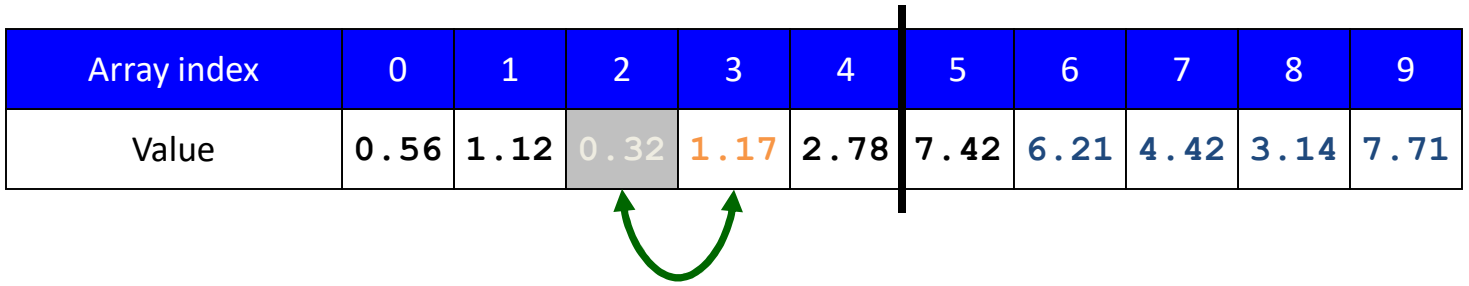


Iteration 5: step 1.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

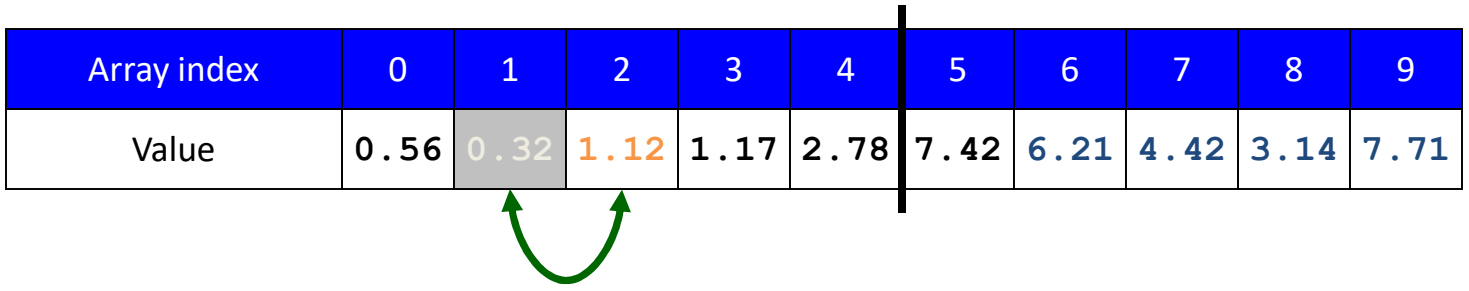


Iteration 5: step 2.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

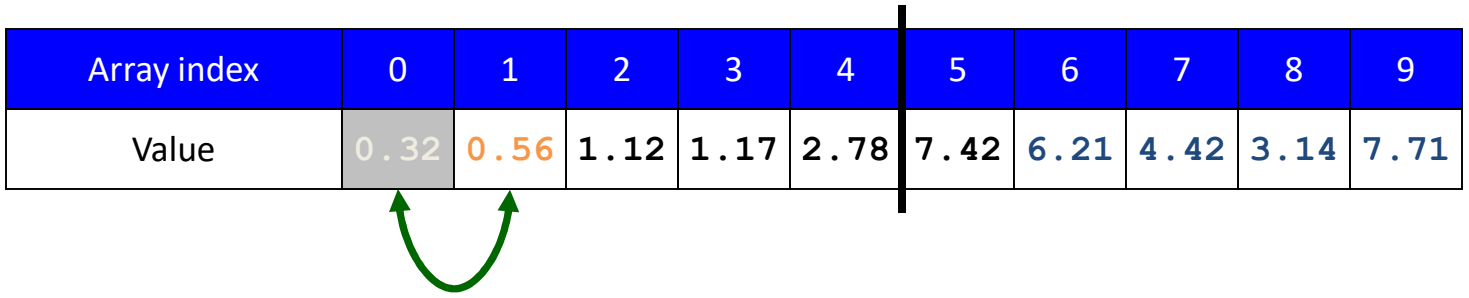


Iteration 5: step 3.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order



Iteration 5: step 4.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

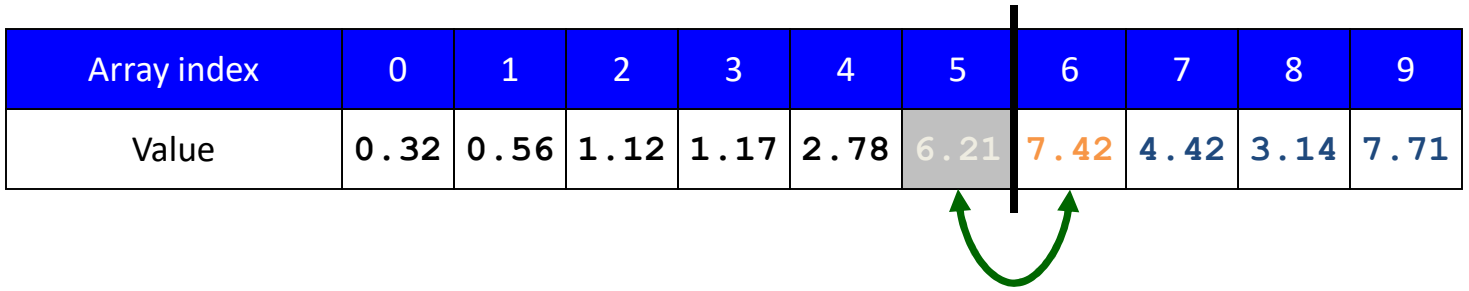
Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 5.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order



Iteration 6: step 0.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

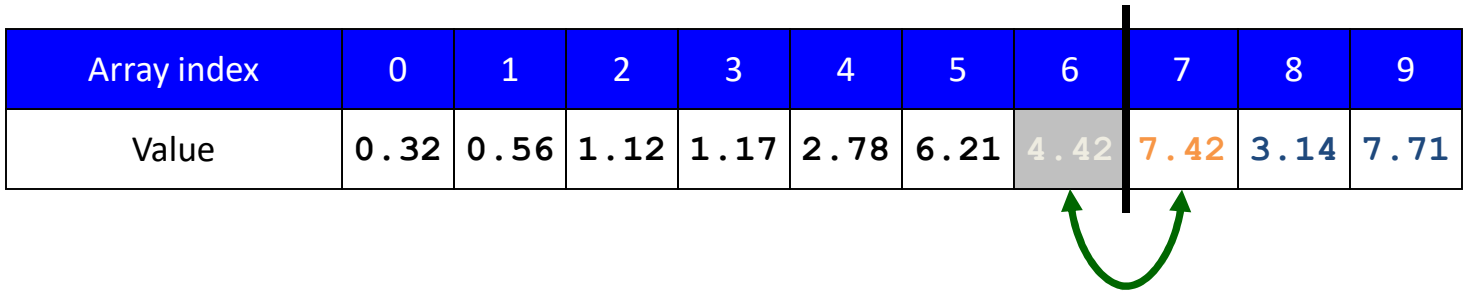
Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	7.42	4.42	3.14	7.71

Iteration 6: step 1.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

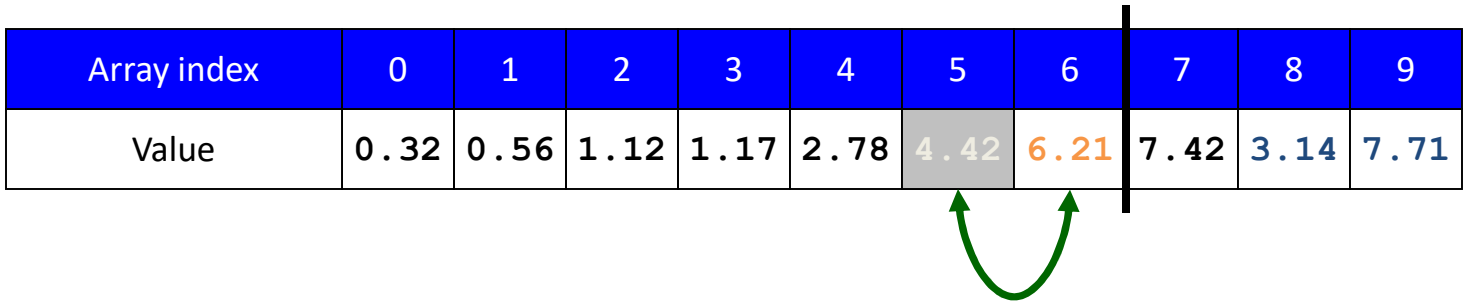


Iteration 7: step 0.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order



Iteration 7: step 1.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

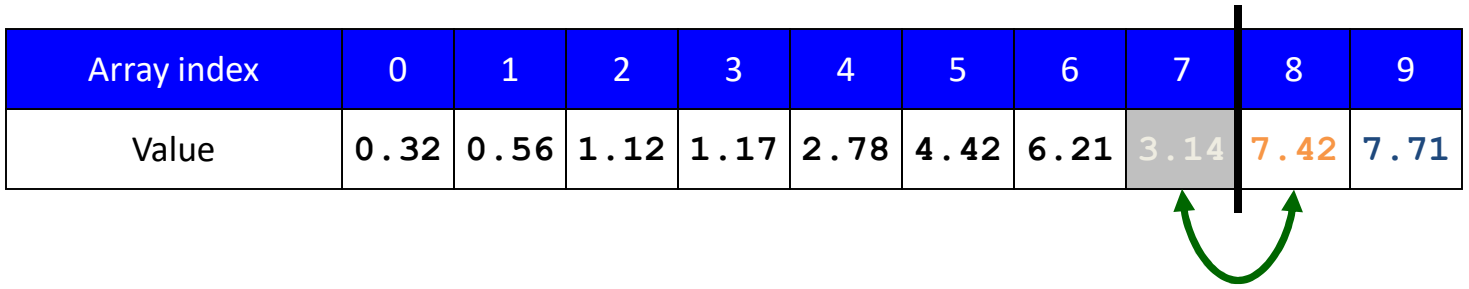
Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71

Iteration 7: step 2.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

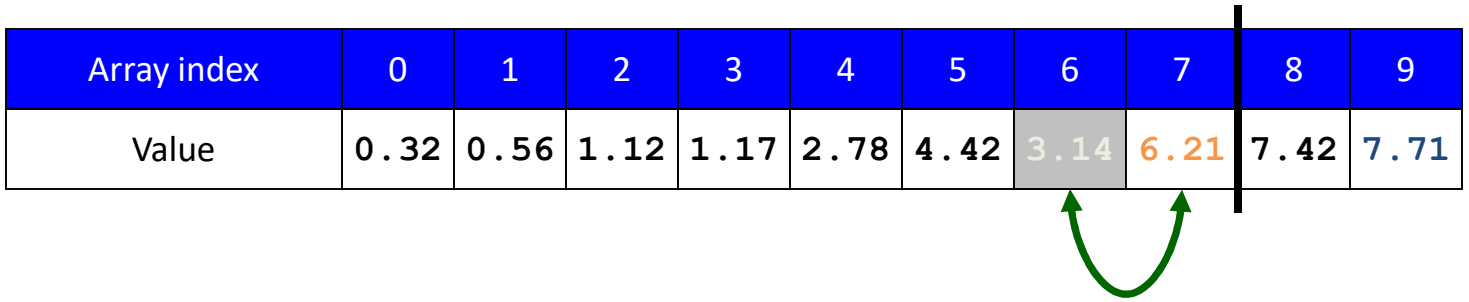


Iteration 8: step 0.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

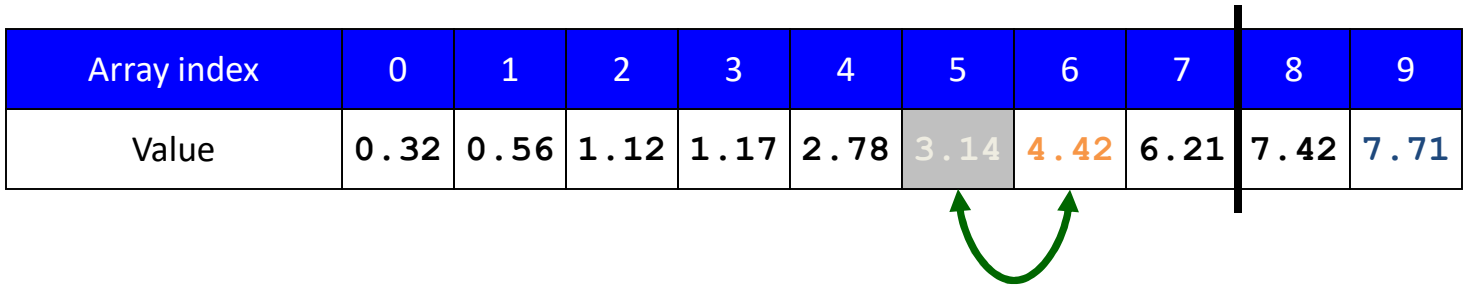


Iteration 8: step 1.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order



Iteration 8: step 2.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 8: step 3.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 9: step 0.

Example 1: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller

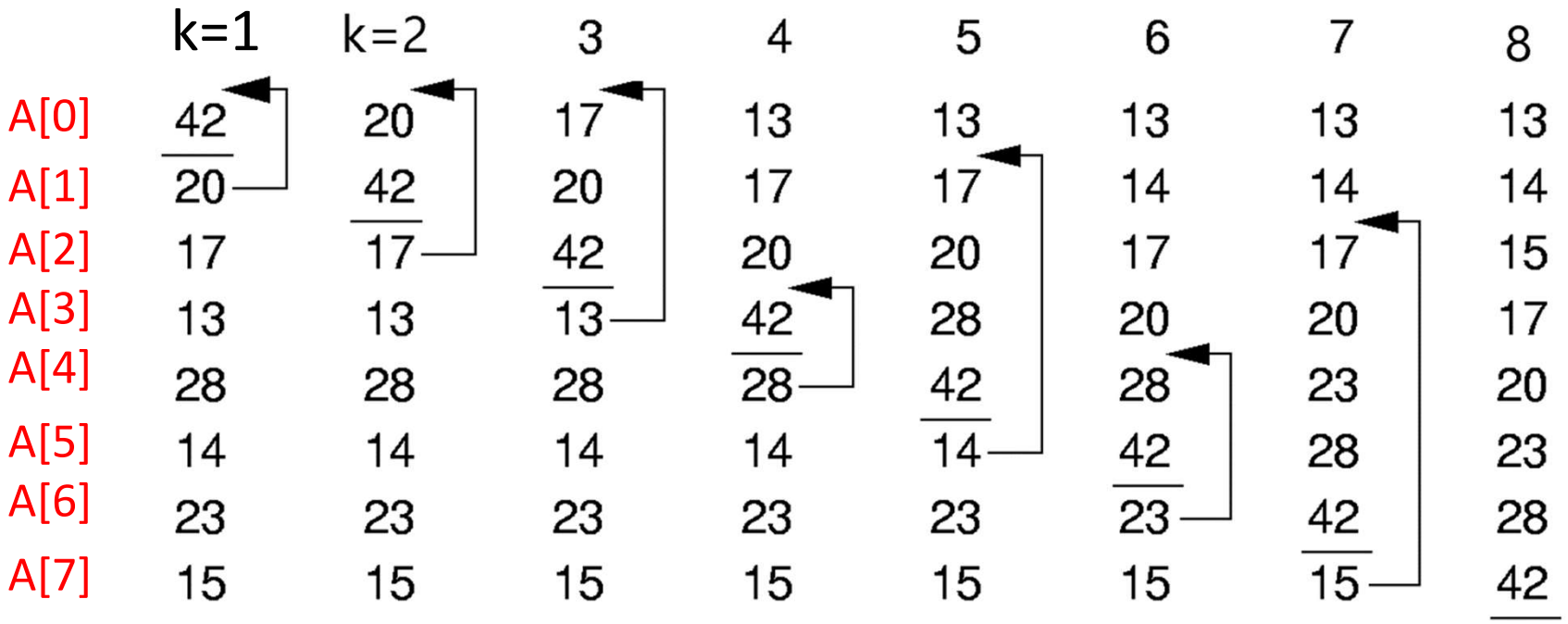
- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 10: DONE.

Example 2: Insertion Sort

- Iteration k: repeatedly swap element k^{th} with the one to its left if smaller
- Properties. After k^{th} iteration, $A[0]$ through $A[k]$ contain first k+1 elements in ascending order



Running time of Insertion sort: $O(n^2)$

- Properties: In place and Stable
- Running time:
  - Best Case: 0 swaps, n-1 comparisons
    - $O(n)$ when the input array is already sorted, or O(1) if only 1 element)
  - Worst Case: $n^2/2$ swaps and comparisons
    - $O(n^2)$, when the input array is in reverse order.
  - Average Case: $n^2/4$ swaps and comparisons
    - $O(n^2)$

# of Sorted elements	Best case	Worst case
0	0	0
1	1	1
2	1	2
…	…	…
n-1	1	n-1
Number of comparisons	**n-1**	**$n(n-1)/2$**

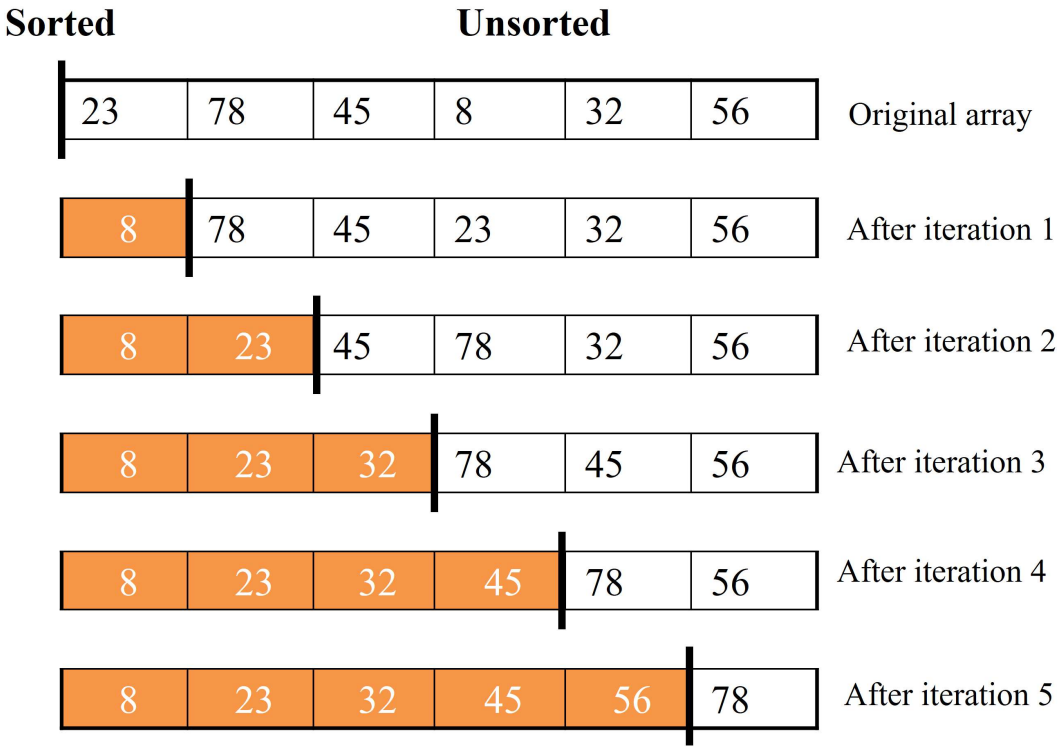
Running time of Insertion sort: $O(n^2)$

- When we should use insertion sort ?:
  - When the data sets are relatively small.
    - Moderately efficient.
  - When you want a quick easy implementation.
    - Not hard to code Insertion sort.
  - When data sets are mostly sorted already (each element is almost put very near its correct place in ordered array)
    - Example: (1, 2, 4, 6, 3, 2)

Contents

5.2. Selection sort

- The array is divided into two subarrays, *sorted* and *unsorted*, which are divided by an imaginary wall.
- Each iteration: We find the smallest element from the unsorted subarray and swap it with the element at the beginning of the unsorted subarray ➔ After each iteration (after each selection and swapping): the imaginary wall between the two subarrays move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- An array of n elements requires n-1 iterations to completely sort the array.



At iteration i: find the smallest element among the unsorted elements a[i+1]…a[n-1] and put it at position i^{th}

5.2. Selection sort

Algorithm:

- The array is divided into two subarrays, *sorted* and *unsorted*, which are divided by an imaginary wall.
- Each iteration: We find the smallest element from the unsorted subarray and swap it with the element at the beginning of the unsorted subarray ➔ After each iteration (after each selection and swapping), the imaginary wall between the two subarrays move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- An array of n elements requires n-1 iterations to completely sort the array.

- Find the smallest element and put it to 1st place of the array
- Find the next smallest element, put it 2nd
- Find the next smallest element, put it 3rd
- …

5.2. Selection sort

```
void swap(int *a,int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
void selectionSort(int a[], int n){
  int i, j, index_min;
  for (i = 0; i < n-1; i++) {
    index_min = i;
     //Find the smallest element in the unsorted subarray: a[i+1] … a[n-1]
    for (j = i+1; j < n; j++)
       if (a[j] < a[index_min]) index_min = j;
    //put a[index_min] to the ith position
    swap(&a[i], &a[index_min]);
  }
}
```

At iteration *i*: find the smallest element among the unsorted elements a[*i*+1]…a[*n*-1] and put it at position i^{th}

Running time analysis

```
void swap(int *a,int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
void selectionSort(int a[], int n){
  int i, j, index_min;
  for (i = 0; i < n-1; i++) {
    index_min = i;
     //Find the smallest element in the unsorted subarray: a[i+1] … a[n-1]
    for (j = i+1; j < n; j++)
       if (a[j] < a[index_min]) index_min = j;
    //put a[index_min] to the ith position
    swap(&a[i], &a[index_min]);
  }
}
```

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) since each swap, there are 3 moves

The inner for loop executes the size of the unsorted part minus 1 (from n-1 downto 1), and in each iteration we make one key comparison; therefore in total:

- ➔ # of key comparisons = (n-1) + (n-2) + …. + 2 + 1 = n*(n-1)/2
- ➔ **So, Selection sort is O(n^2)**

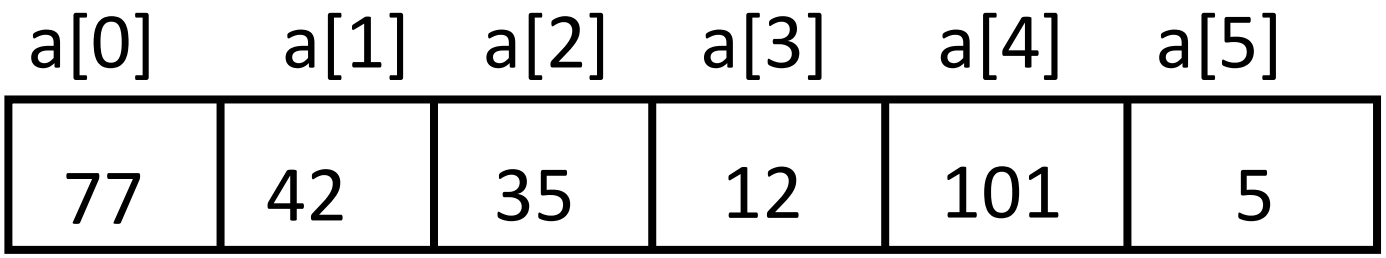
Running time of Selection sort: $O(n^2)$

- Properties: In place and Stable
- Running time: $O(n^2)$ for best/worse/average case. This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.
- 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).

Contents

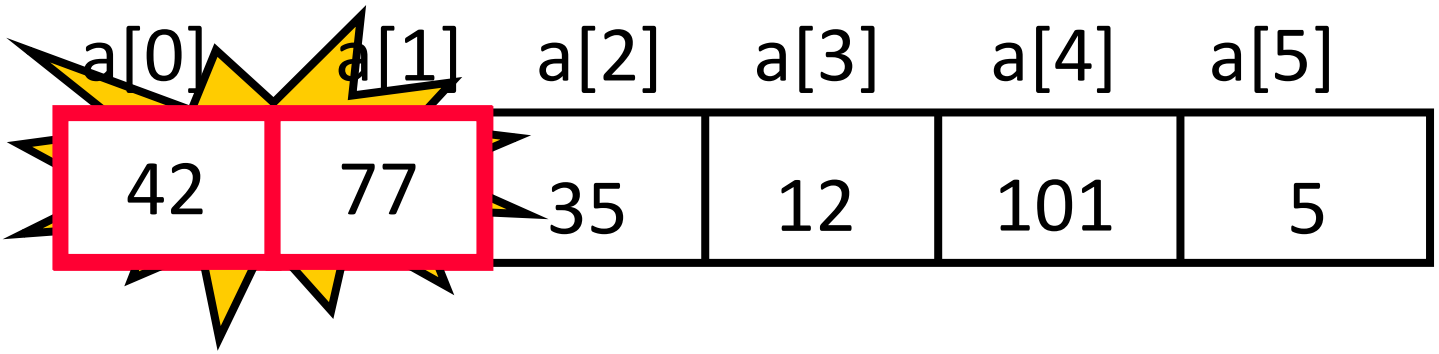
"Bubbling Up" the Largest Element

- Traverse elements of array:
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



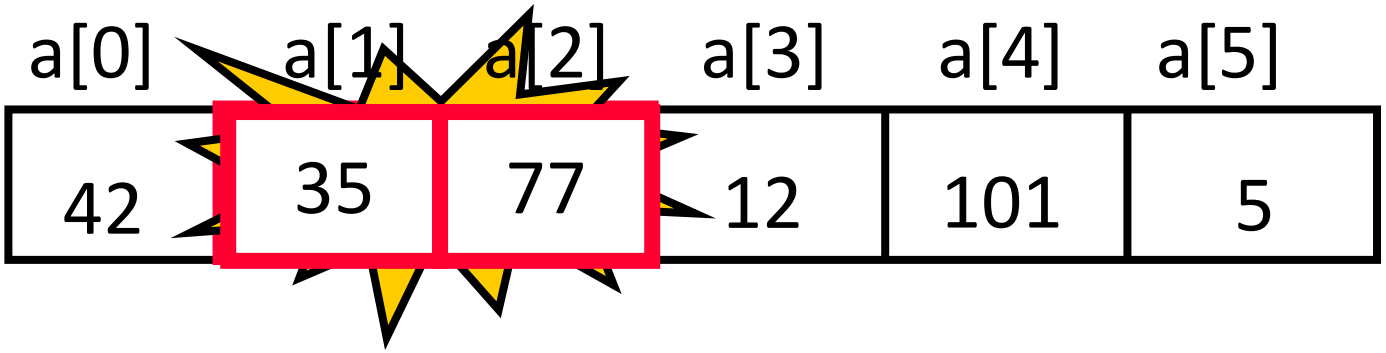
"Bubbling Up" the Largest Element

- Traverse elements of array:
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



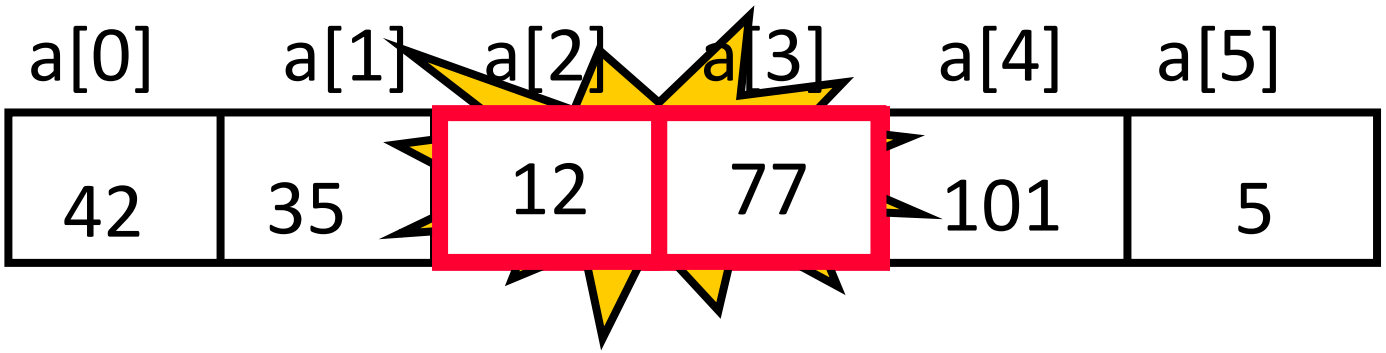
"Bubbling Up" the Largest Element

- Traverse elements of array:
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



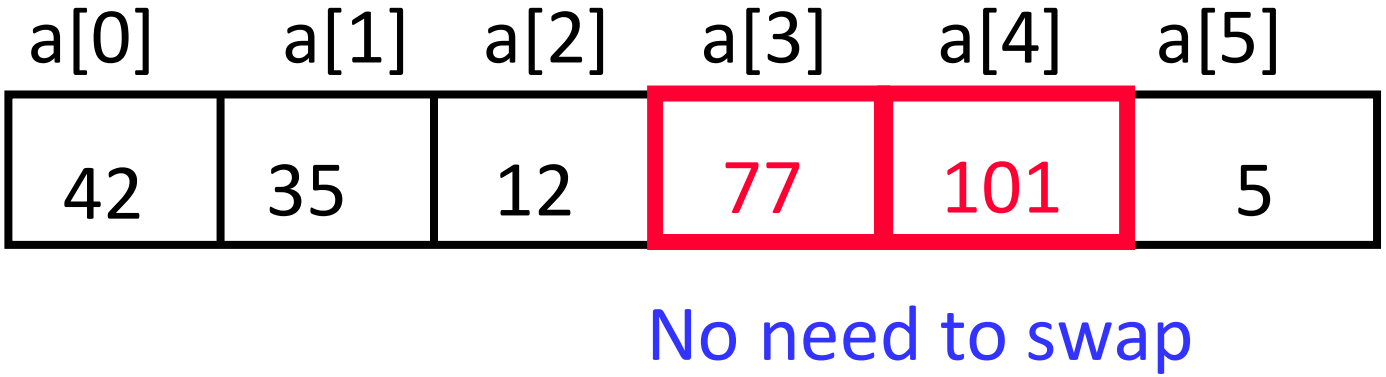
"Bubbling Up" the Largest Element

- Traverse elements of array:
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



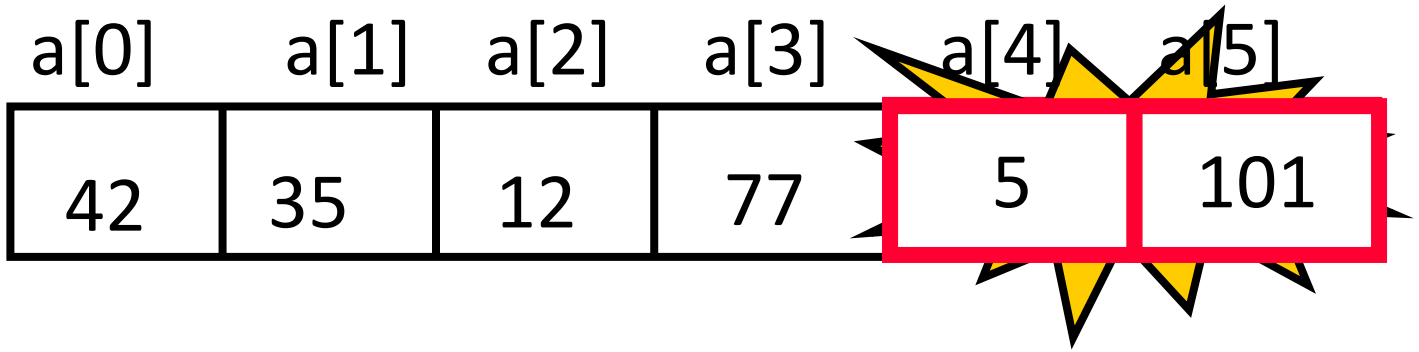
"Bubbling Up" the Largest Element

- Traverse elements of array:
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



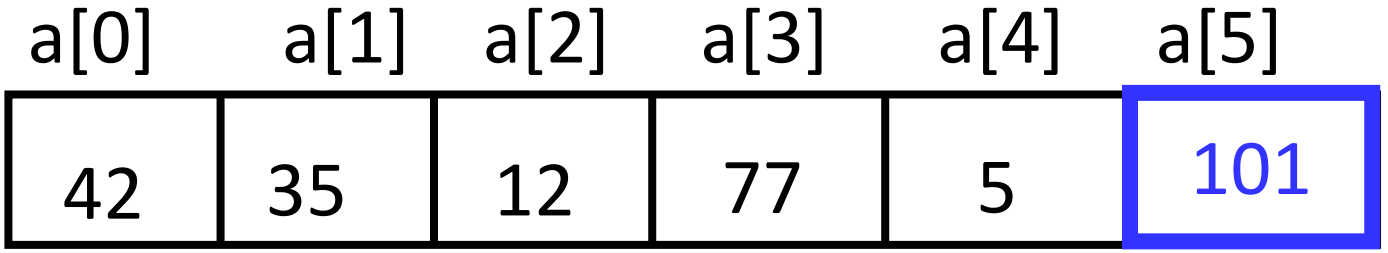
"Bubbling Up" the Largest Element

- Traverse elements of array:
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

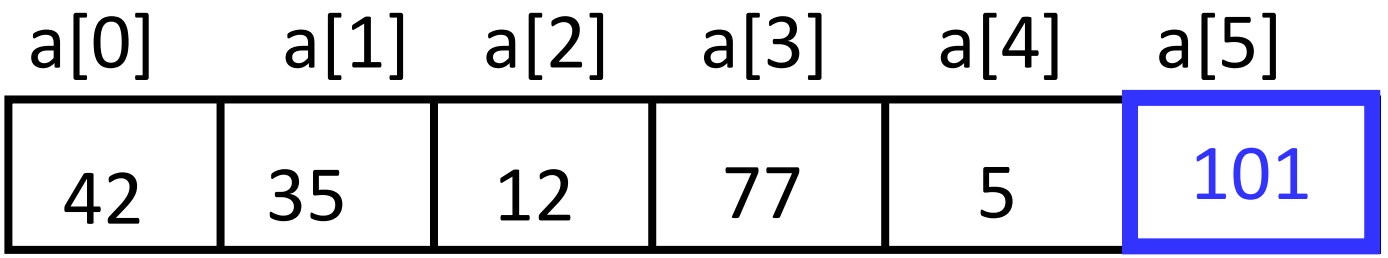
- Traverse elements of array:
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



Largest value correctly placed

Items of Interest

- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process

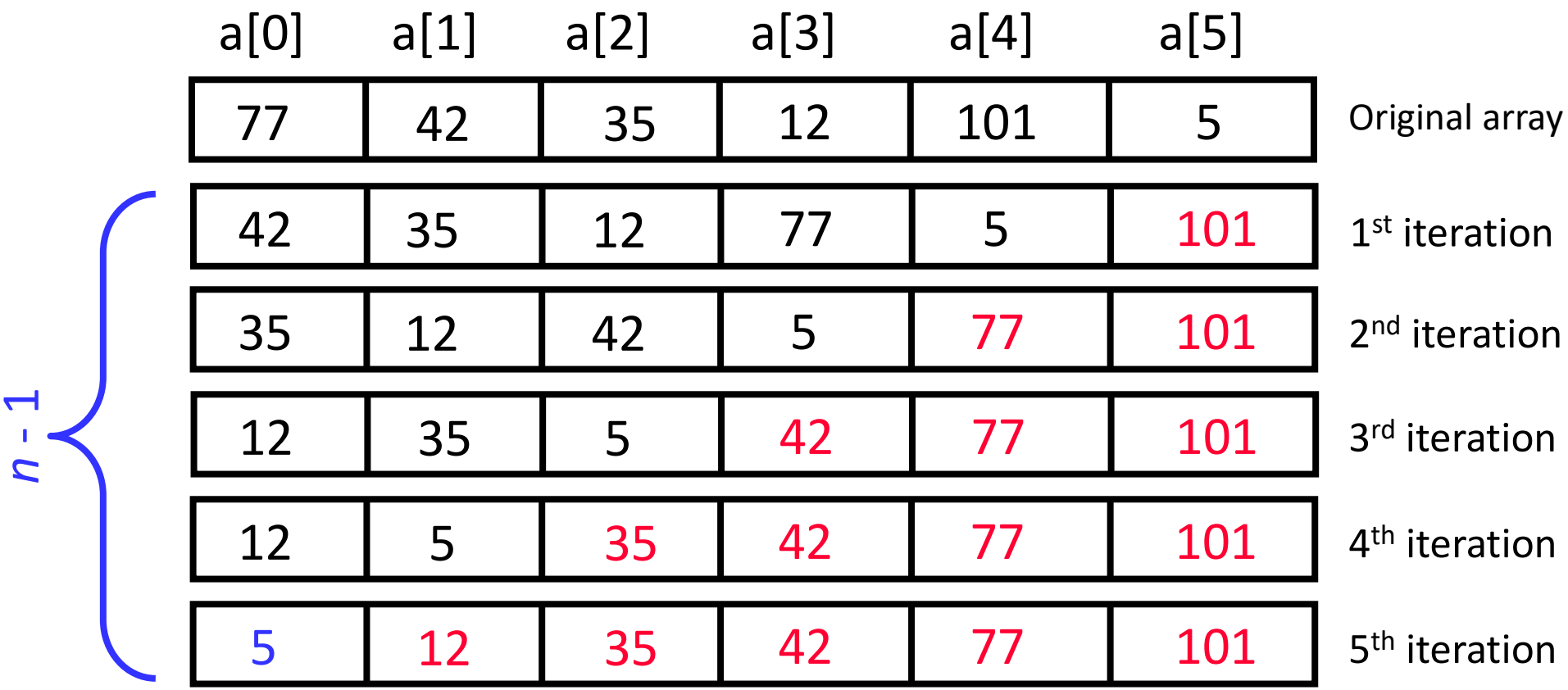


Largest value correctly placed

Repeat "Bubble Up" How Many Times?

- If we have *n* elements…
- And if each time we bubble an element, we place it in its correct location…
- Then we repeat the "bubble up" process $n - 1$ times.
- This guarantees we'll correctly place all *n* elements.

“Bubbling” All the Elements



Example: Sort array: 9, 6, 2, 12, 11, 9, 3, 7

Bubble sort compares the numbers in pairs from left to right, and exchanging when necessary. Here the first number is compared to the second and as it is larger they are exchanged.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

Now the next pair of numbers are compared. Again the 9 is the larger than the 2, and so this pair is also exchanged.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

In the third comparison, the 9 is not larger than the 12 so no exchange is made. We move on to compare the next pair without any change to the array.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

The 12 is larger than the 11 so they are exchanged.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

The 12 is greater than the 9 so they are exchanged

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

6, 2, 9, 11, 9, 12, 3, 7

The 12 is greater than the 3 so they are exchanged.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

6, 2, 9, 11, 9, 12, 3, 7

6, 2, 9, 11, 9, 3, 12, 7

The 12 is greater than the 7 so they are exchanged.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

The end of the array has been reached so this is the end of the first iteration. The twelve at the end of the array must be largest number in the array and so is now in the correct position. We now start a new iteration from left to right.

6, 2, 9, 11, 9, 12, 3, 7

6, 2, 9, 11, 9, 3, 12, 7

6, 2, 9, 11, 9, 3, 7, 12

Example: Sort array: 9, 6, 2, 12, 11, 9, 3, 7

1st iteration: 6, 2, 9, 11, 9, 3, 7, 12

2nd iteration: 2, 6, 9, 9, 3, 7, 11, 12

Notice that this time we do not have to compare the last two numbers as we know the 12 is in position. This iteration therefore only requires 6 comparisons.

Bubble Sort Example

1st iteration: 6, 2, 9, 11, 9, 3, 7, 12

2nd iteration: 2, 6, 9, 9, 3, 7, 11, 12

3rd iteration: 2, 6, 9, 3, 7, 9, 11, 12

This time the 11 and 12 are in position. This iteration therefore only requires 5 comparisons.

Bubble Sort Example

1st iteration: 6, 2, 9, 11, 9, 3, 7, 12

2nd iteration: 2, 6, 9, 9, 3, 7, 11, 12

3rd iteration: 2, 6, 9, 3, 7, 9, 11, 12

4th iteration: 2, 6, 3, 7, 9, 9, 11, 12

Each iteration requires fewer comparisons. This time only 4 are needed.

Bubble Sort Example

1st iteration: 6, 2, 9, 11, 9, 3, 7, 12

2nd iteration: 2, 6, 9, 9, 3, 7, 11, 12

3rd iteration: 2, 6, 9, 3, 7, 9, 11, 12

4th iteration: 2, 6, 3, 7, 9, 9, 11, 12

5th iteration: 2, 3, 6, 7, 9, 9, 11, 12

The array is now sorted but the algorithm does not know this until it completes an iteration with no exchanges.

Bubble Sort Example

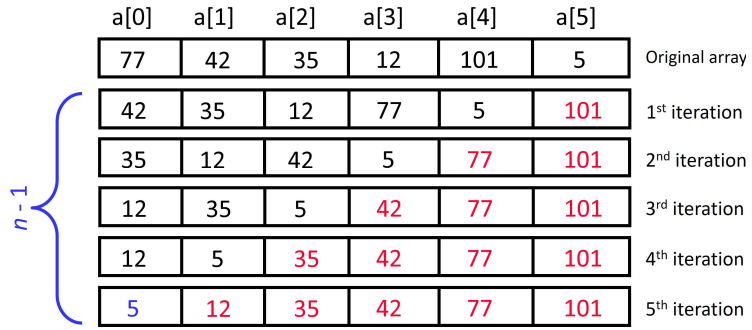
1st iteration:	6, 2, 9, 11, 9, 3, 7, 12
2nd iteration:	2, 6, 9, 9, 3, 7, 11, 12
3rd iteration:	2, 6, 9, 3, 7, 9, 11, 12
4th iteration:	2, 6, 3, 7, 9, 9, 11, 12
5th iteration:	2, 3, 6, 7, 9, 9, 11, 12
6th iteration:	2, 3, 6, 7, 9, 9, 11, 12

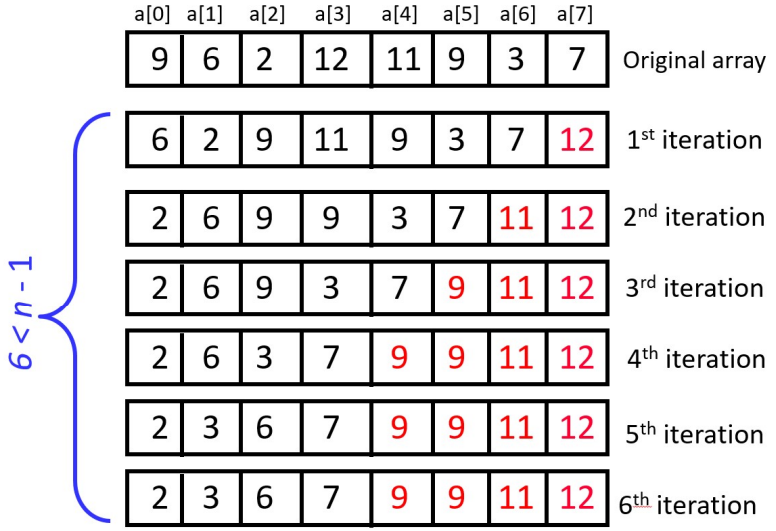
This iteration no exchanges are made so the algorithm knows the array is sorted. It can therefore save time by not doing the final iteration. With other arrays this check could save much more work.

5.3. Bubble Sort: implementation

Apply bubble sort to sort this array of n numbers: a[0], a[1], …a[n-1]:

- The algorithm stops when it identifies that the array is already sorted (@ iteration in which there does not occur any exchange)
- The maximum number of iterations = $n - 1$
  - Each iteration i (i=1,2,..,n-1): bubble the largest number among elements a[0]…a[n-i] to the last position (n-i)





Stop @ iteration 6th because the algorithm identifies that there does not occur any exchange

```
void bubbleSort(int a[], int n)
{
   bool sorted = false; /*use to identify the iteration having no
                 exchanges, it means the array is sorted already */
   for (int i = 1; i <= n-1; i++)
      if (sorted == false)
      {
          sorted = true;
          for (int j= 0; j <= n-i-1; j++)
            if (a[j] < a[j+1])
            {
                swap(&a[j],&a[j+1]);
                sorted = false; // signal exchange
            }
       }
}
```

5.3. Bubble Sort: implementation

```
void bubbleSort(int a[], int n)
{
    bool sorted = false; /*use to identify the iteration having no
                          exchanges, it means the array is sorted already */
    for (int i = 1; i <= n-1; i++)
        if (sorted == false)
        {
            sorted = true;
            for (int j= 0; j <= n-i-1; j++)
              if (a[j] < a[j+1])
              {
                  swap(&a[j],&a[j+1]);
                  sorted = false; // signal exchange
              }
         }
}
```

5.3. Bubble Sort – Analysis

- ***Best-case:*** **O(n)**
  - Array is already sorted in ascending order.
    - The number of moves: 0 ➔ O(1)
    - 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 moves: 3*(1+2+...+n-1) = 3 * n*(n-1)/2 ➔ O(n^2)
    - 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, Bubble Sort is O(n^2)

```
void bubbleSort(int a[], int n)
{
   bool sorted = false; /*use to identify the iteration having no
                exchanges, it means the array is sorted already */
   for (int i = 1; i <= n-1; i++)
      if (sorted == false)
      {
          sorted = true;
          for (int j= 0; j <= n-i-1; j++)
            if (a[j] < a[j+1])
            {
                swap(&a[j],&a[j+1]);
                sorted = false; // signal exchange
            }
       }
}
```

Contents

$O(n^2)$

Divide and conquer

Divide and Conquer

3 steps:

- Divide: Decomposing a given problem into a problem of the same type with a smaller size (called a subproblem)
- Conquer: Solve subproblems independently
- Combine: Synthesize the solution of subproblems to obtain the solution of the original problem

Idea 1: Divide a given array into halves, recursively arrange left and right halves, then merge the two halves to get the complete sorted array ➔ Merge sort

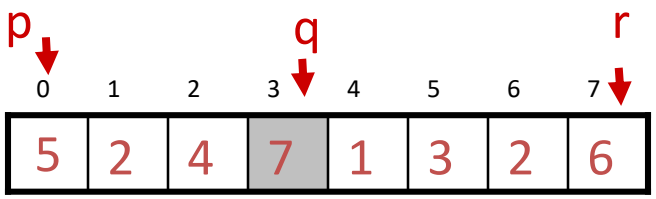
Idea 2: Divide a given array into two sets: a set containing "small" elements and a set containing "large" elements, then recursively sort these two sets → Quick sort

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

5.4. Merge Sort

- Problem: Sorting n elements of an array A[0]…A[n-1]
- Merge-sort on an input sequence $\boldsymbol{A}$ with $\boldsymbol{n}$ elements consists of three steps:
  - Divide: partition $\boldsymbol{A}$ into two sequences $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ of about $\boldsymbol{n}/2$ elements each
  - Conquer: recursively sort $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ by using merge sort
  - Combine: merge $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ into a unique sorted sequence

5.4.Merge Sort



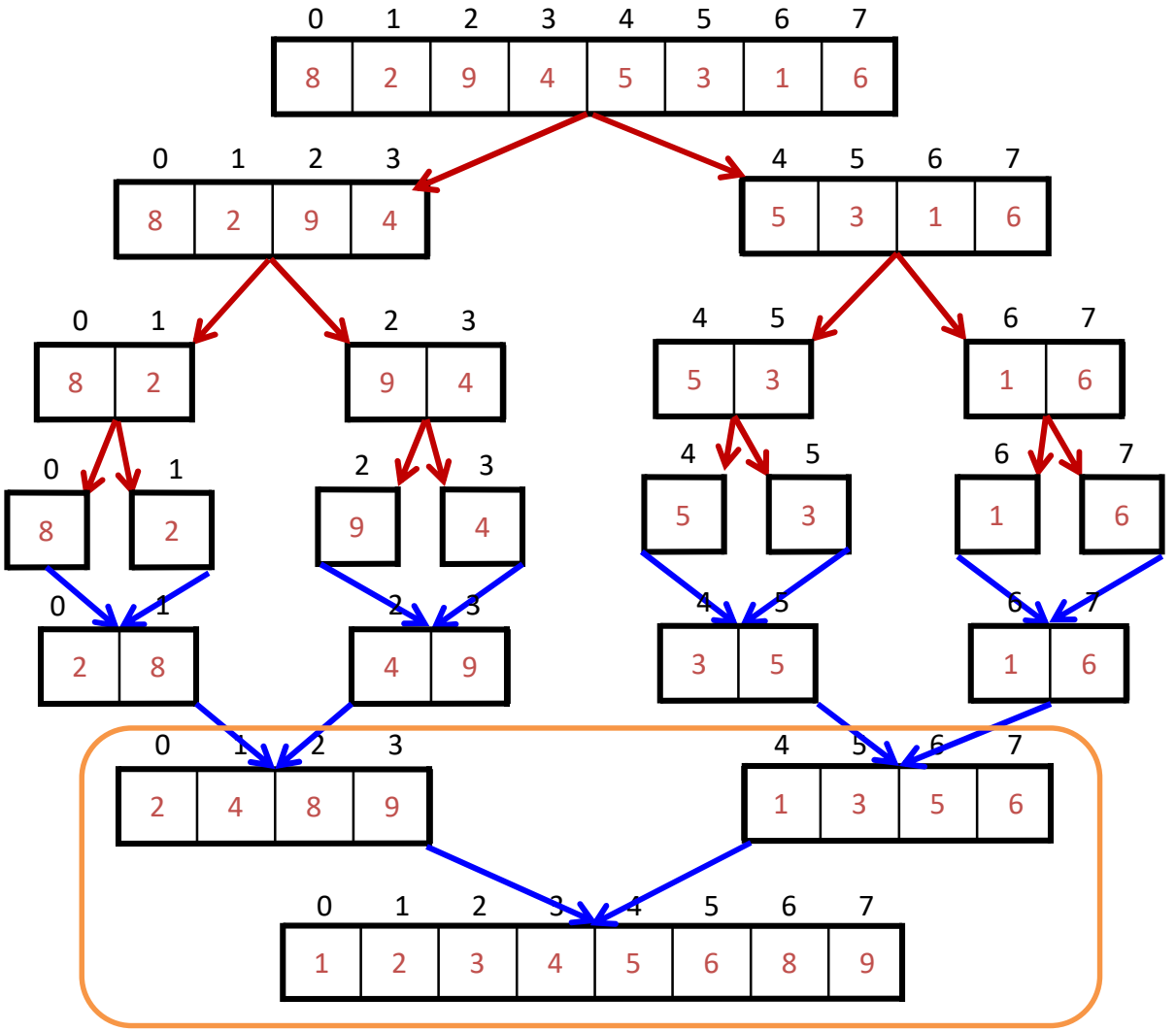
MS(*A*, p, r)

```
if p < r  then                          ▷ Check the base case
    q ← ⌊(p + r)/2⌋                     ▷ Divide
    MS(A, p, q)                         ▷ Conquer
    MS(A, q + 1, r)                     ▷ Conquer
    MERGE(A, p, q, r)                   ▷ Combine
endif
```

- The statement to sort array A of n elements: MS(A, 0, n-1);

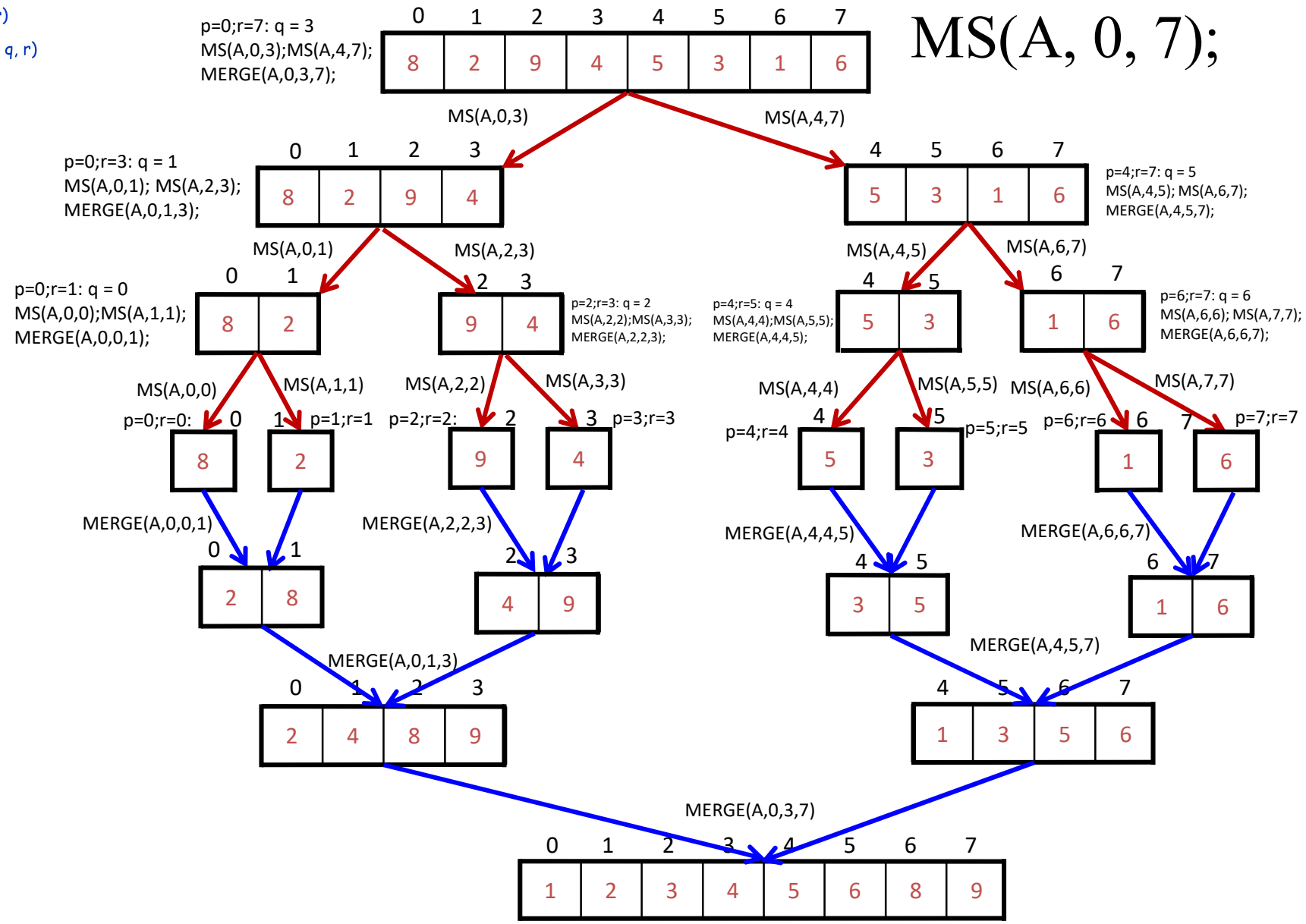
Example: Merge sort

```
MS(A, p, r)
if p < r then
    q ← ⌊(p + r)/2⌋
    MS(A, p, q)
    MS(A, q + 1, r)
    MERGE(A, p, q, r)
endif
```

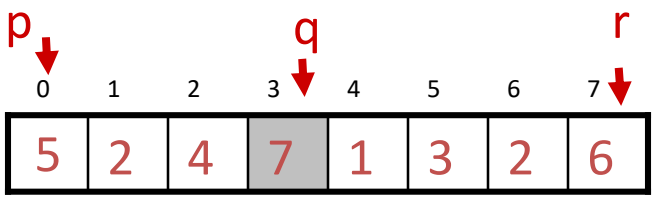


```
MS(A, p, r)
if p < r then
    q ← ⌊(p + r)/2⌋
    MS(A, p, q)
    MS(A, q + 1, r)
    MERGE(A, p, q, r)
endif
```

Example: Merge sort



5.4.Merge Sort



MS(*A*, p, r)

if p < r **then** ▷ Check the base case

$q \leftarrow \lfloor (p + r)/2 \rfloor$ ▷ Divide

MS(*A*, p, *q*) ▷ Conquer

MS(*A*, *q* + 1, r) ▷ Conquer

MERGE(*A*, p, *q*, r) ▷ Combine

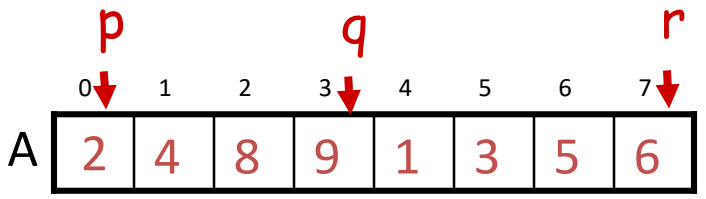
endif

Merge 2 ordered subarrays A[p]...A[q] and A[q+1]...A[r] into one array that is ordered in ascending order

Merge 2 ordered subarrays A[p]...A[q] and A[q+1]...A[r] into one array that is ordered in ascending order

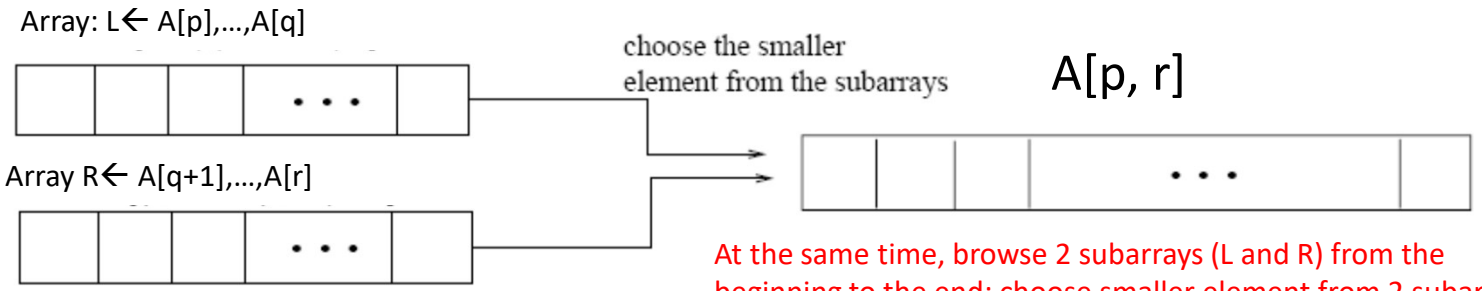
Array A consists of 2 ordered halves:

- Subarray 1: consists of elements A[p]… A[q]
- Subarray 2: consists of elements A[q+1]… A[r]

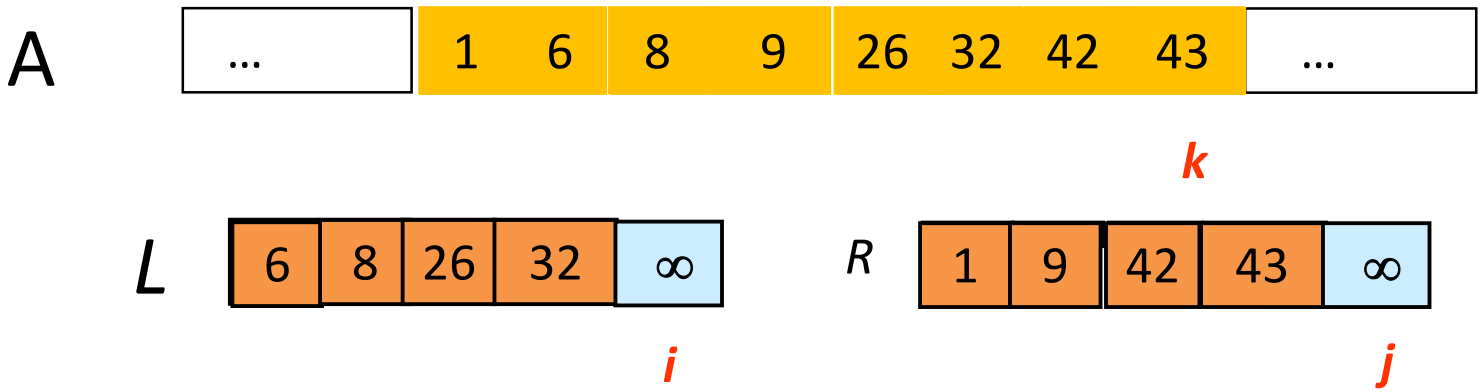


We need to merge these 2 subarrays to get an array that is ordered in ascending order:

Idea 1: we use 2 extra subarrays L, R with size = ½ size of array A



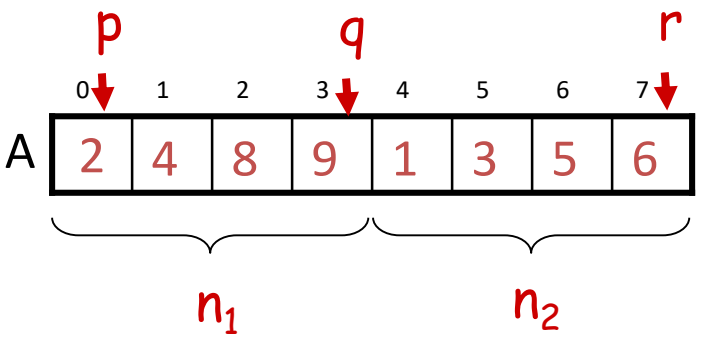
At the same time, browse 2 subarrays (L and R) from the beginning to the end: choose smaller element from 2 subarrays to insert into array A.

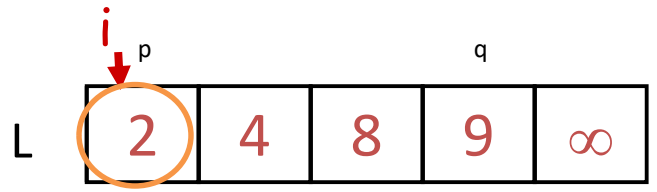


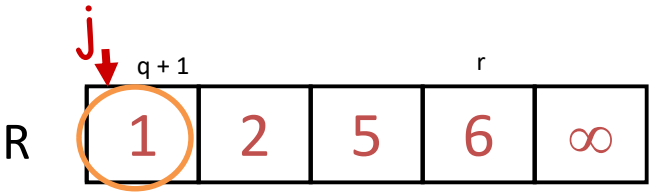
Merge - Pseudocode

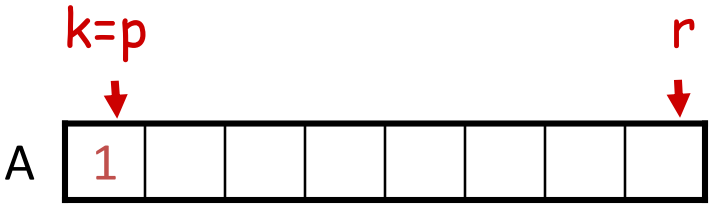
MERGE(A, p, q, r)

1. Calculate n_1 and n_2
2. Copy the first n_1 elements to $L[1 \ldots n_1]$ and remain n_2 elements to $R[1 \ldots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \leq R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$





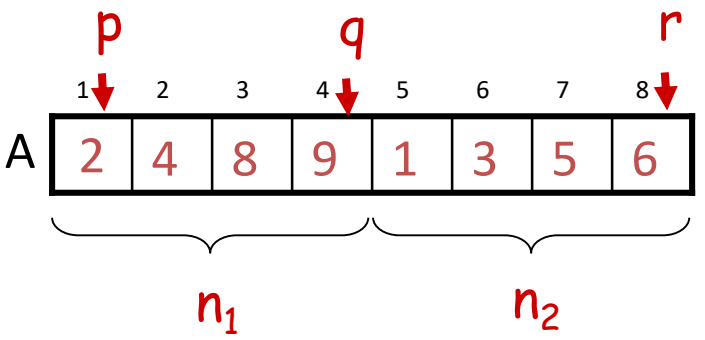


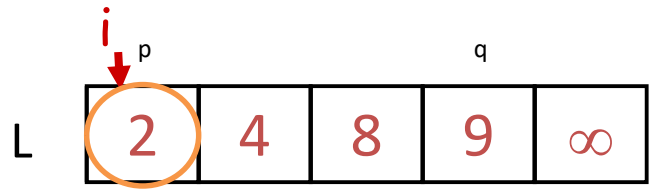


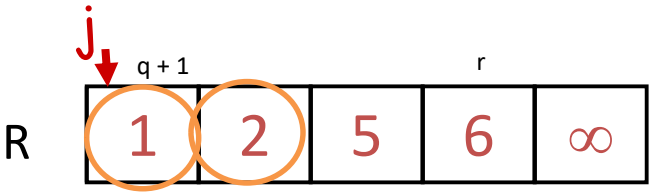
Merge - Pseudocode

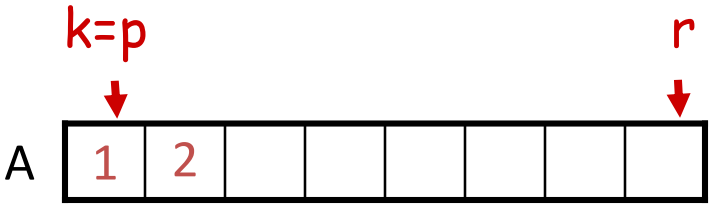
MERGE(A, p, q, r)

1. Calculate n_1 and n_2
2. Copy the first n_1 elements to $L[1 \ldots n_1]$ and remain n_2 elements to $R[1 \ldots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \leq R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$





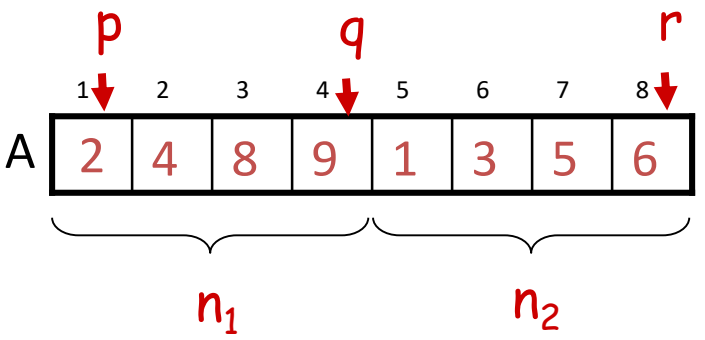


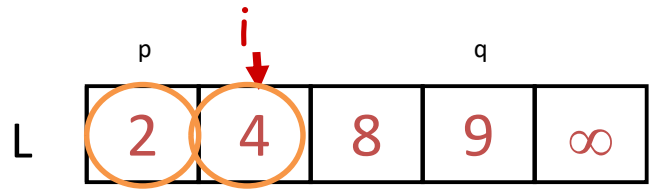


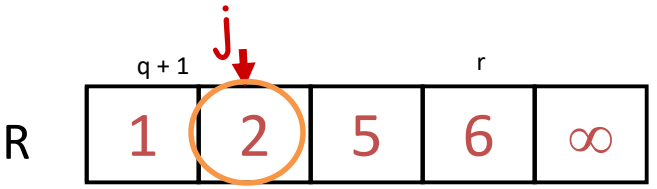
Merge - Pseudocode

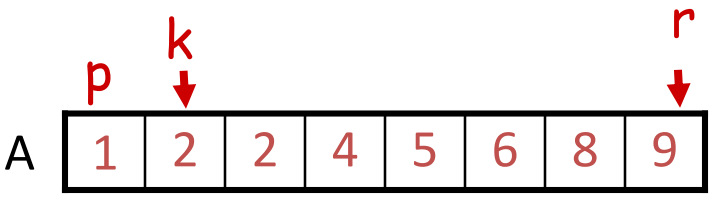
MERGE(A, p, q, r)

1. Calculate n_1 and n_2
2. Copy the first n_1 elements to $L[1 \ldots n_1]$ and remain n_2 elements to $R[1 \ldots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \leq R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$









Computation time analysis Merge sort: O(*n*log*n*)

Running time of Merge procedure:

- Initialize (create 2 subarrays L and R):
  - $O(n_1 + n_2) = O(n)$
- Insert elements into array S (the loop **for**):
  - n loops, each loop requires constant time $\Rightarrow O(n)$
- Total running time of Merge procedure: $O(n)$

Running time of Merge Sort:

- **Divide:** calculate q as the average of p and r: O(1)
- **Conquer:** solve 2 sub problems with size $n/2$ each $\Rightarrow 2T(n/2)$
- **Combine:** MERGE 2 sub arrays of n elements requires $O(n)$

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

Therefore: $T(n) = O(n \log n)$

MERGE(A, p, q, r)

1. Calculate n_1 and n_2
2. Copy the first n_1 elements to $L[1 \ldots n_1]$ and remain n_2 elements to $R[1 \ldots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \le R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$

MS(A, p, r)

if p < r **then**

q ← ⌊(p + r)/2⌋

MS(A, p, q)

MS(A, q + 1, r)

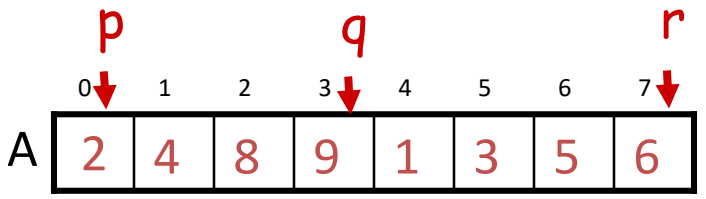
MERGE(A, p, q, r)

endif

Merge 2 ordered subarrays A[p]...A[q] and A[q+1]...A[r] into one array that is ordered in ascending order

Array A consists of 2 ordered halves:

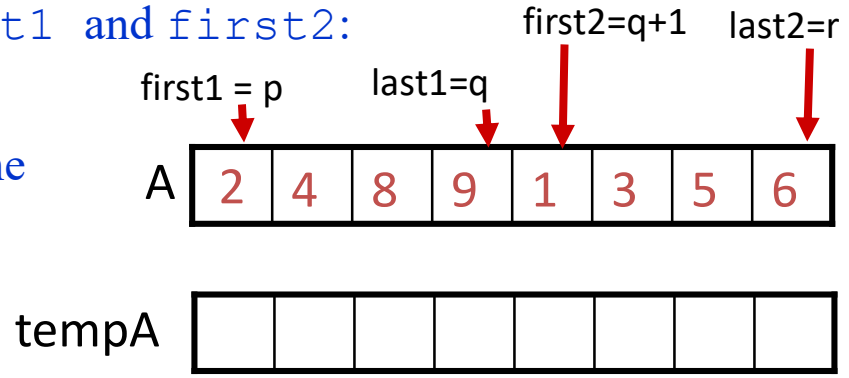
- Subarray 1: consists of elements A[p]… A[q]
- Subarray 2: consists of elements A[q+1]… A[r]



We need to merge these 2 subarrays to get an array that is ordered in ascending order:

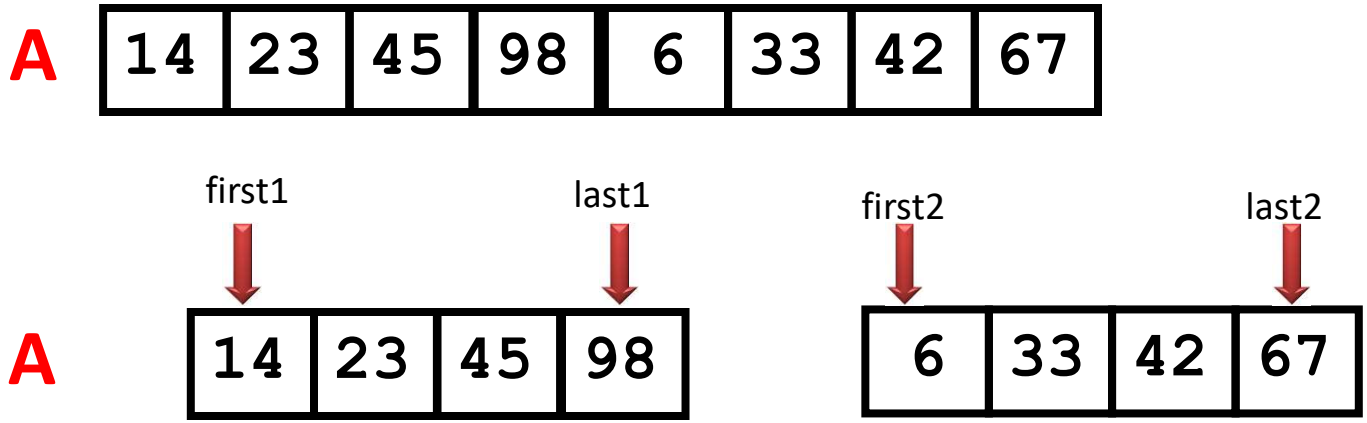
Idea 2: use a temporary array tempA with the same size of array A

At the same time scan through each element of two sub-sequences A [p..q] and A [q + 1… r] by using the variable `first1` and `first2`: compare each pair of corresponding 2 elements of 2 sub-sequences, selects smaller element to copy to the sub array tempA. At the end of the loop, all elements of the two sub-sequences have been scanned; then the tempA array contains all the elements of the two sub-sequences, but they are are sorted. Do A = tempA



Idea 2: Merge

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```



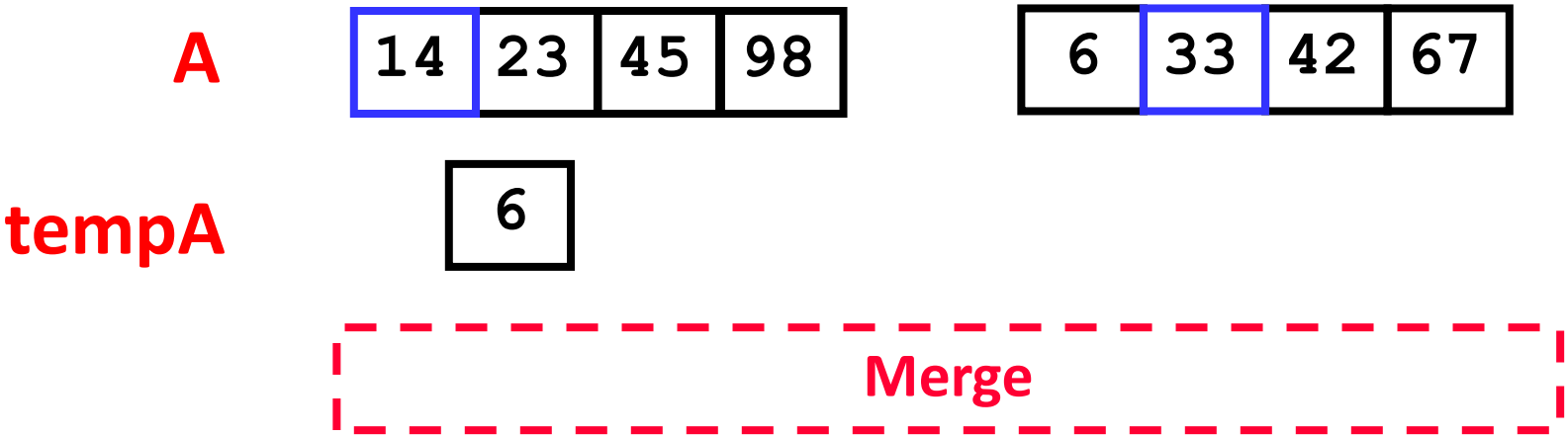
Idea 2: Merge

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```

Merge

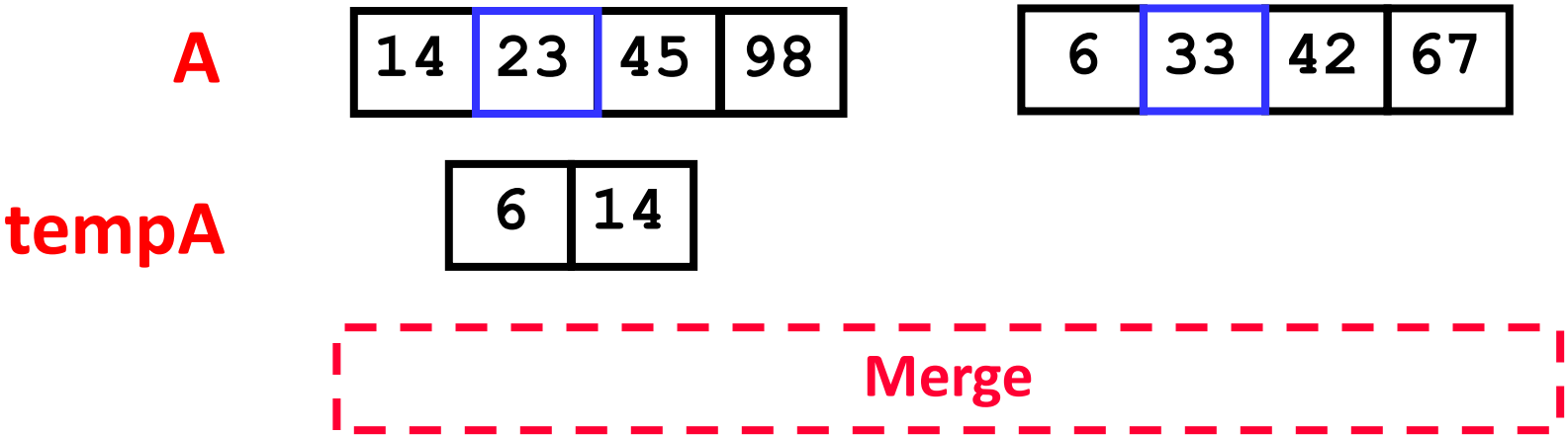
Idea 2: Merge

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```



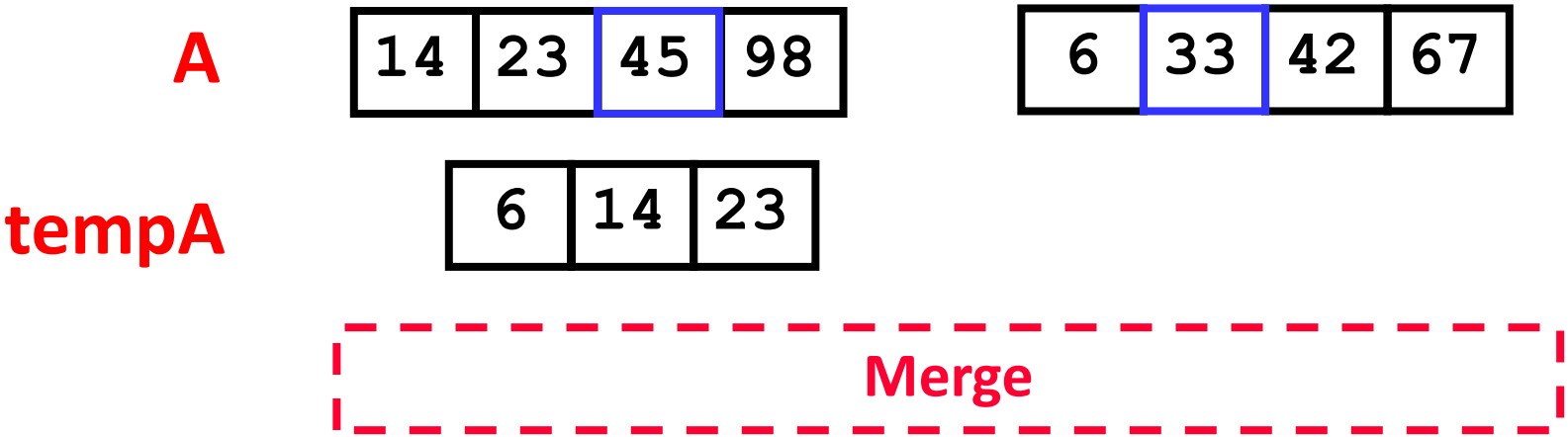
Idea 2: Merge

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```



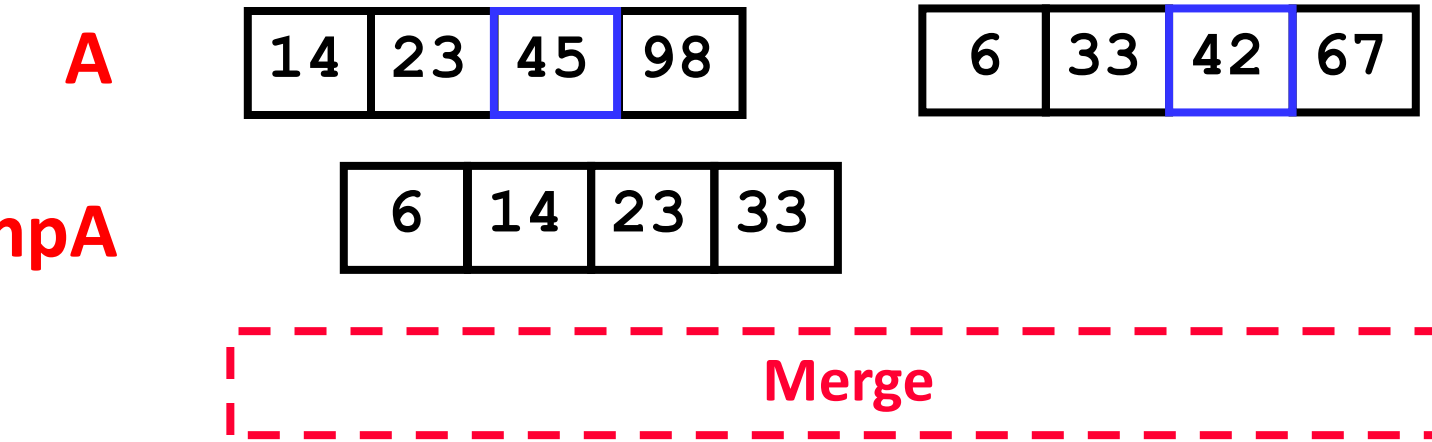
Idea 2: Merge

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```



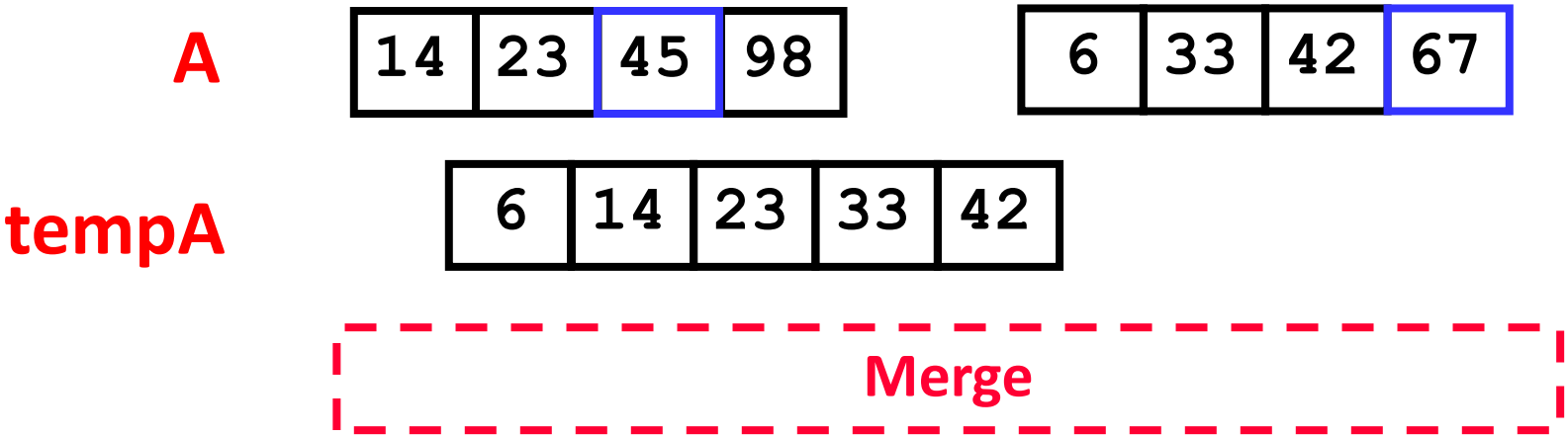
Idea 2: Merge

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```



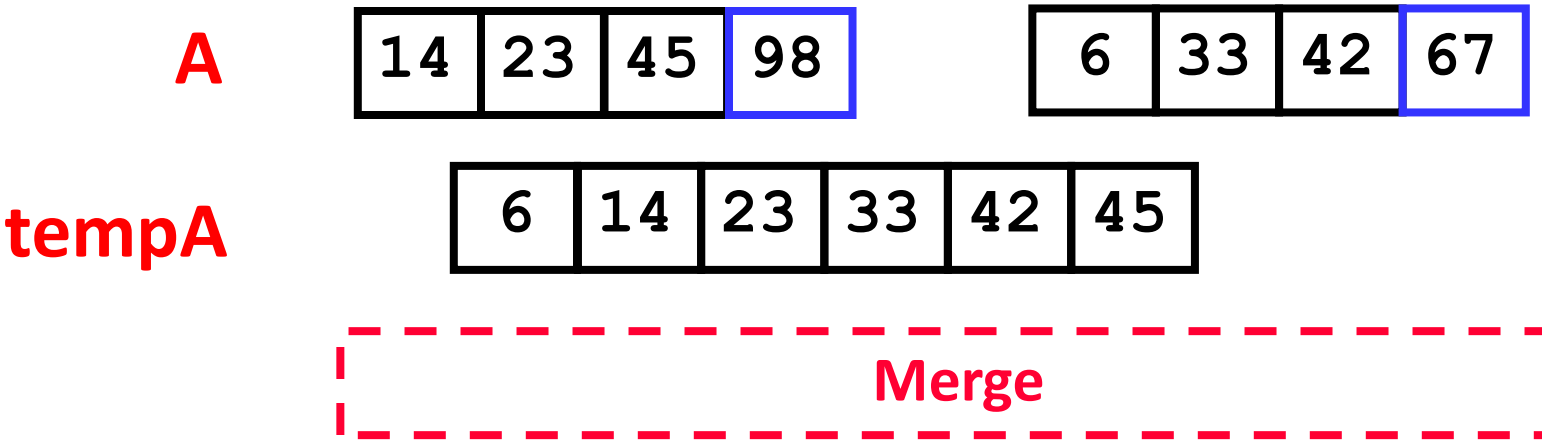
Idea 2: Merge

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```



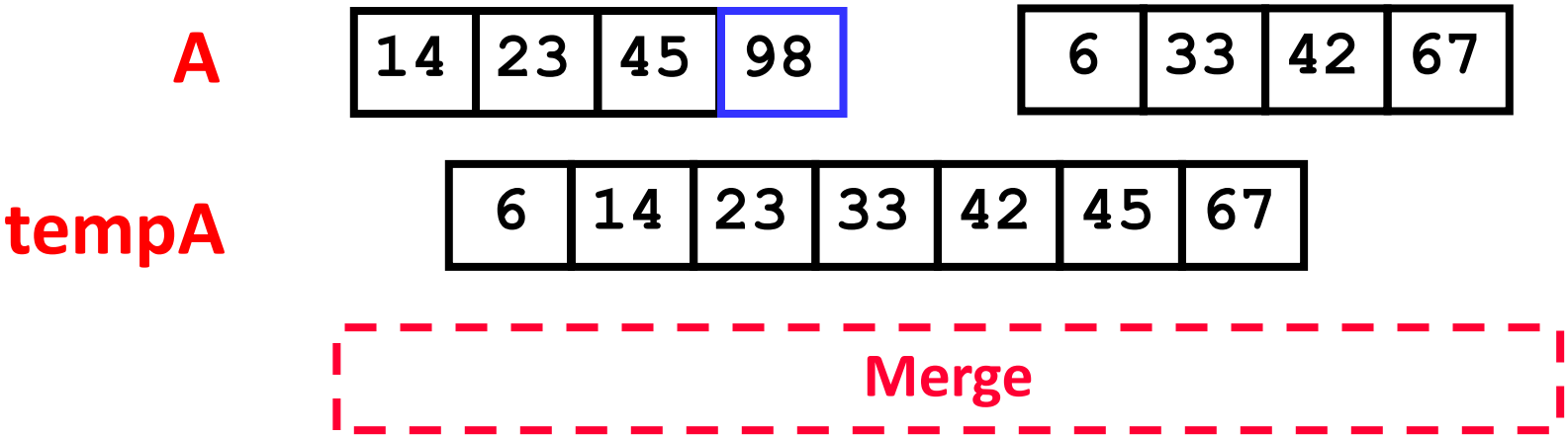
Idea 2: Merge

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```



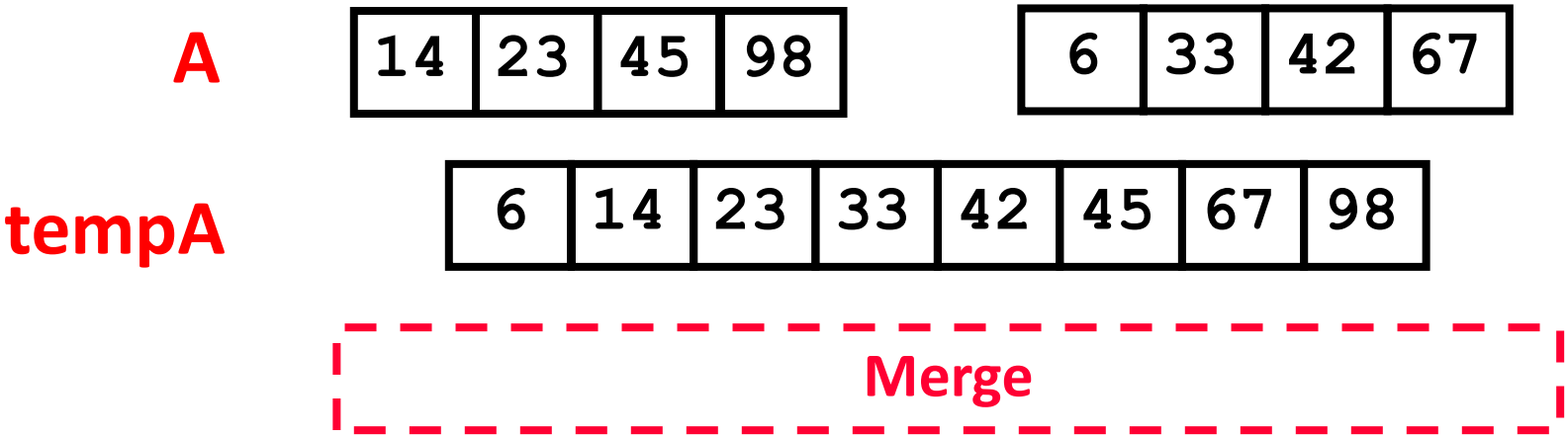
Idea 2: Merge

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```



Idea 2: Merge

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```



```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];     // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        {tempA[index] = A[first1]; ++first1;}
        else
        {  tempA[index] = A[first2]; ++first2;}
    }

     for (; first1 <= last1; ++first1, ++index)
       tempA[index] = A[first1]; // sao not dãy con 1
     for (; first2 <= last2; ++first2, ++index)
       tempA[index] = A[first2]; // sao not dãy con 2
     for (index = first; index <= last; ++index)
       A[index] = tempA[index]; // sao tra mang ket qua
}  // end merge
```

```
void mergeSort(int A[], int first, int last)
{
    if (first < last)
    {  // chia thành hai dãy con
        int mid = (first + last)/2;     // chi so diem giua
        // sap xep dãy con trái A[first..mid]
        mergeSort(A, first, mid);
        // sap xep dãy con phai A[mid+1..last]
        mergeSort(A, mid+1, last);
       // Tron hai dãy con
        merge(A, first, mid, last);
   }  // end if
}  // end mergesort


void main()
{
    int a[5] = {8,4,3,2,1};
    mergeSort(a,0,4);
    for (int i = 0; i<5; i++)
        printf("%d \n",a[i]);
}
```

Contents

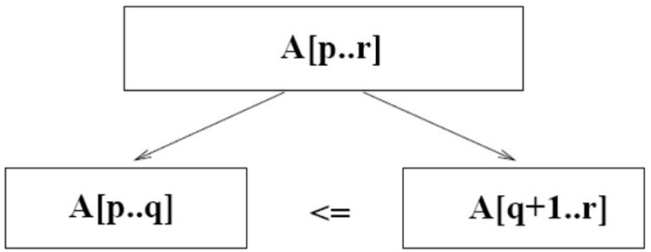
$O(n^2)$

Divide and conquer

5.5. Quick sort

The quick sort algorithm is described recursively as following (similar to merge sort):

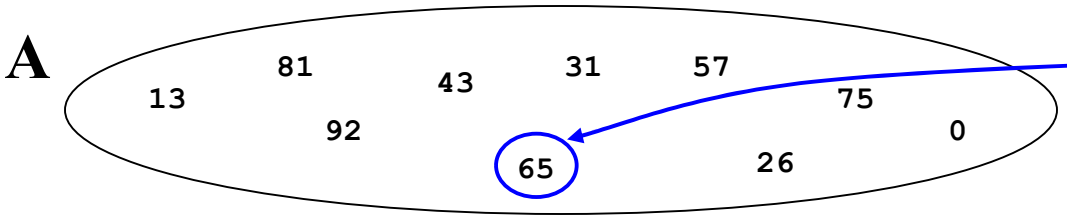
1. **Base case**. If the array has only one element, then the array is sorted already, return it without doing anything.
2. **Divide**:
   - Select an element in the array, and call it as the pivot ***p***.
   - Divide the array into 2 subarrays: Left subarray (L) consists of elements $\leq$ the pivot, right subarray (R) consists of elements $\geq$ the pivot. This operation is called "Partition".



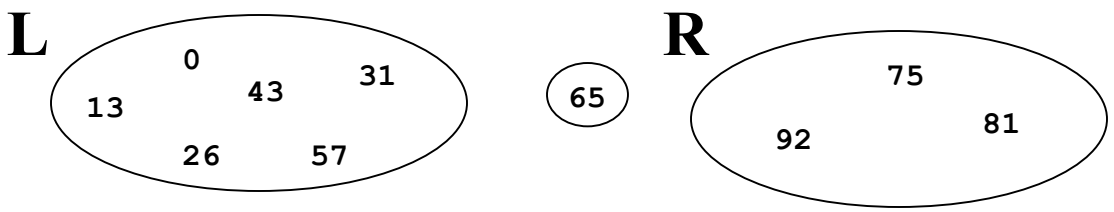
3. **Conquer**: recursively call QuickSort for 2 subarrays $L = A[p \ldots q]$ and $R = A[q+1 \ldots r]$.
4. **Combine**: The sorted array is $L\ p\ R$.

In contrast to Merge Sort, in Quick Sort: division operation is complicate, but the Partition operation is simple.

5.5.Quick sort: Example

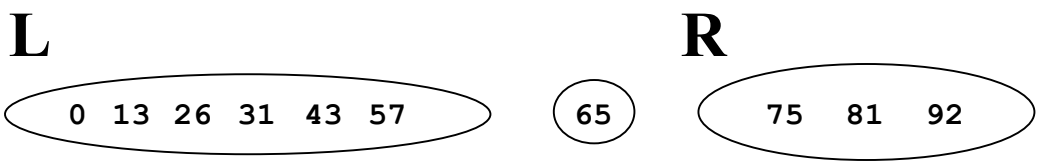


Select pivot

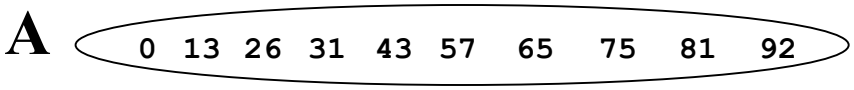


Divide array S into 2 subarrays:

- Left half: consists of elements <=pivot
- Right half: consists of elements >= pivot



QuickSort(L) and
QuickSort(R)



OK! **A** is sorted

5.5. Quick sort diagram

Quick-Sort(*A*, *Left*, *Right*)

```
1.    if (Left < Right ) {
2.          idPivot = Partition(A, Left, Right);
3.          Quick-Sort(A, Left, idPivot –1);
4.          Quick-Sort(A, idPivot +1, Right);
5.    }
```

Function Partition(*A*, *Left*, *Right*) divides *A*[*Left*..*Right*] into 2 subarrays *A*[*Left*..*idPivot* –1] and *A*[id*Pivot*+1..*Right*] such that:

- Elements of *A*[*Left*..*idPivot* –1] is less than or equal to *A*[id*Pivot*]
- Elements of *A*[id*Pivot*+1..*Right*] is greater than or equal to *A*[id*Pivot*].

The statement to sort array A consists of *n* elements: Quick-Sort(*A*, 0, *n*-1)

Quick sort

Professor Knuth thinks that when the sub-array has only a small number of elements (according to him, it is no more than 9 elements), we should use simple algorithms to sort this sequence, not continue to subdivide. The algorithm in such a situation can be described as follows:

```
Quick-Sort(A, Left, Right)
1.    if (Right - Left < n0 )
2.        InsertionSort(A, Left, Right);
3.    else {
4.            idPivot = Partition(A, Left, Right);
5.            Quick-Sort(A, Left, idPivot –1);
6.            Quick-Sort(A, idPivot +1, Right);
7.    }
```

Quick sort

The selection of the pivot plays a decisive role in the efficiency of the algorithm. It is best if the selected pivot is the one with the average value in the array (we call such an element median). Then, after $\log_2 n$ times of partition, we reach an array of size 1. However, that is very difficult to do. It is common to use the following ways to select the pivot element:

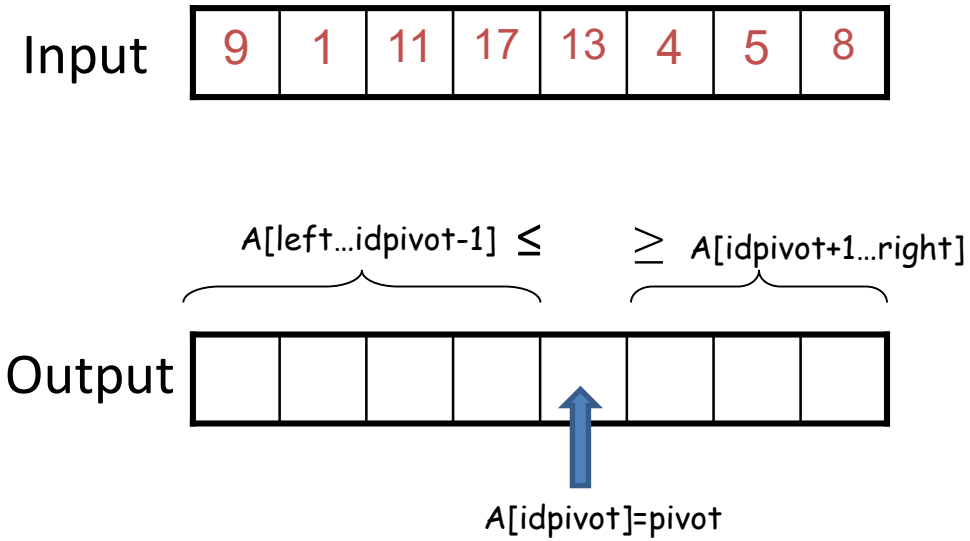
- Select the left most (first) element as the pivot.
- Select the rightmost (last) element as the pivot.
- Select the element in the middle of the array as the pivot.
- Select the median element in the 3 elements: top, middle and the last as the pivot element (Knuth).
- Randomly selects an element as the pivot.

Partition operation

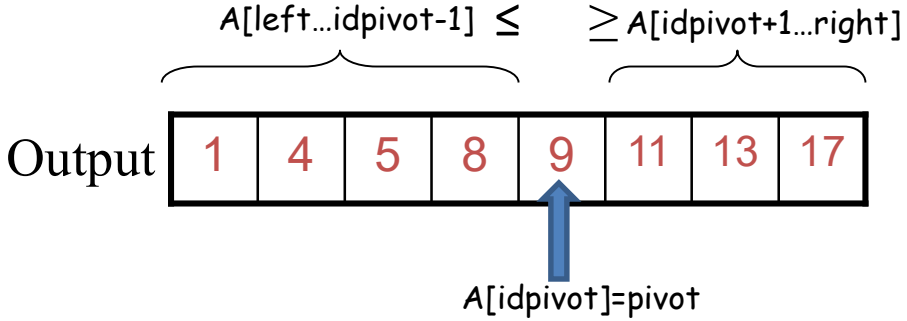
Function **Partition(A, left, right)** :

- **Input:** Array *A*[*left* .. *right*].
- **Output:** Redistribute elements of the array A based on the selected ***pivot*** element and return the ***idpivot*** index to satisfy:
  - *a*[*idpivot*] has the value = *pivot*,
  - *a*[*i*] ≤ *a*[*idpivot*], with *left* ≤ *i* < *idpivot*,
  - *a*[*j*] ≥ *a*[*idpivot*]), with *idpivot* < *j* ≤ *right*.

where *pivot* is the value of selected pivot element



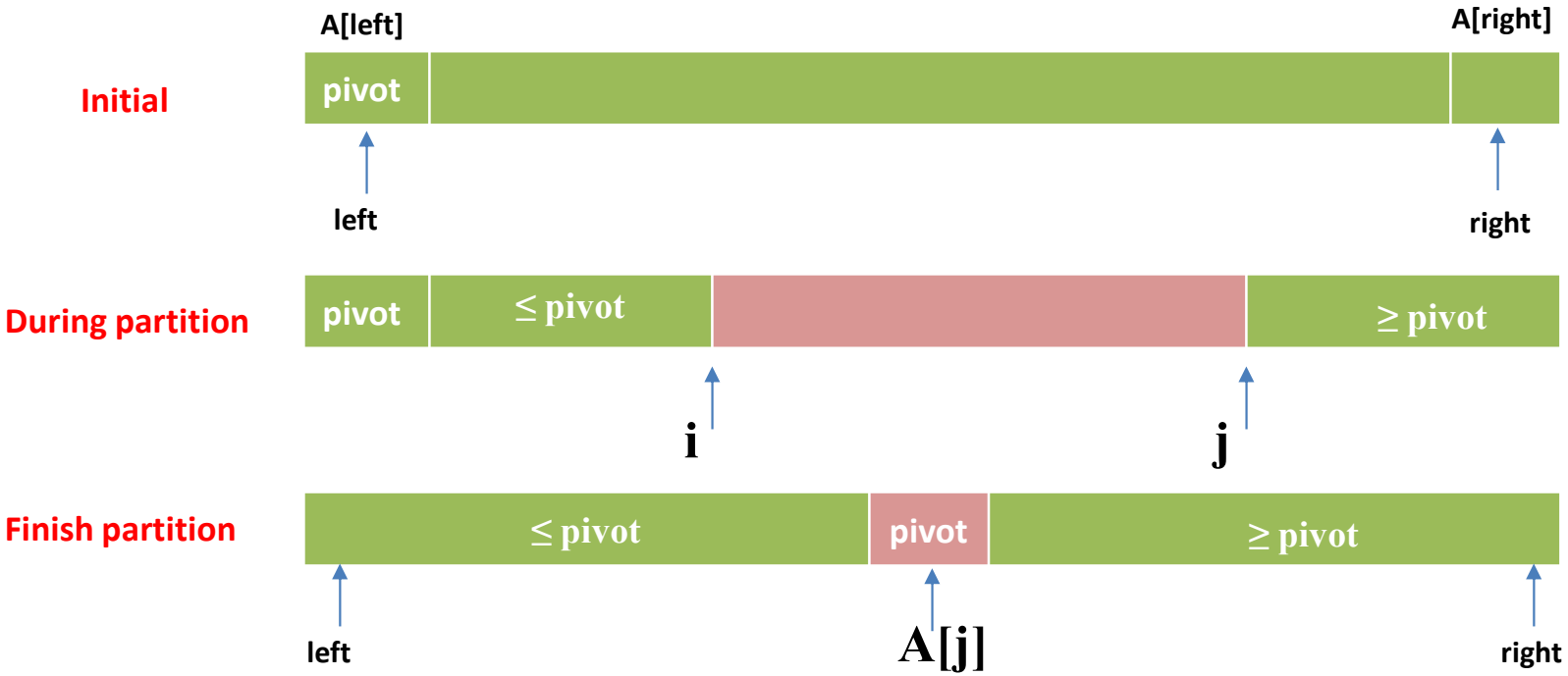
Example: select pivot as the first element of array



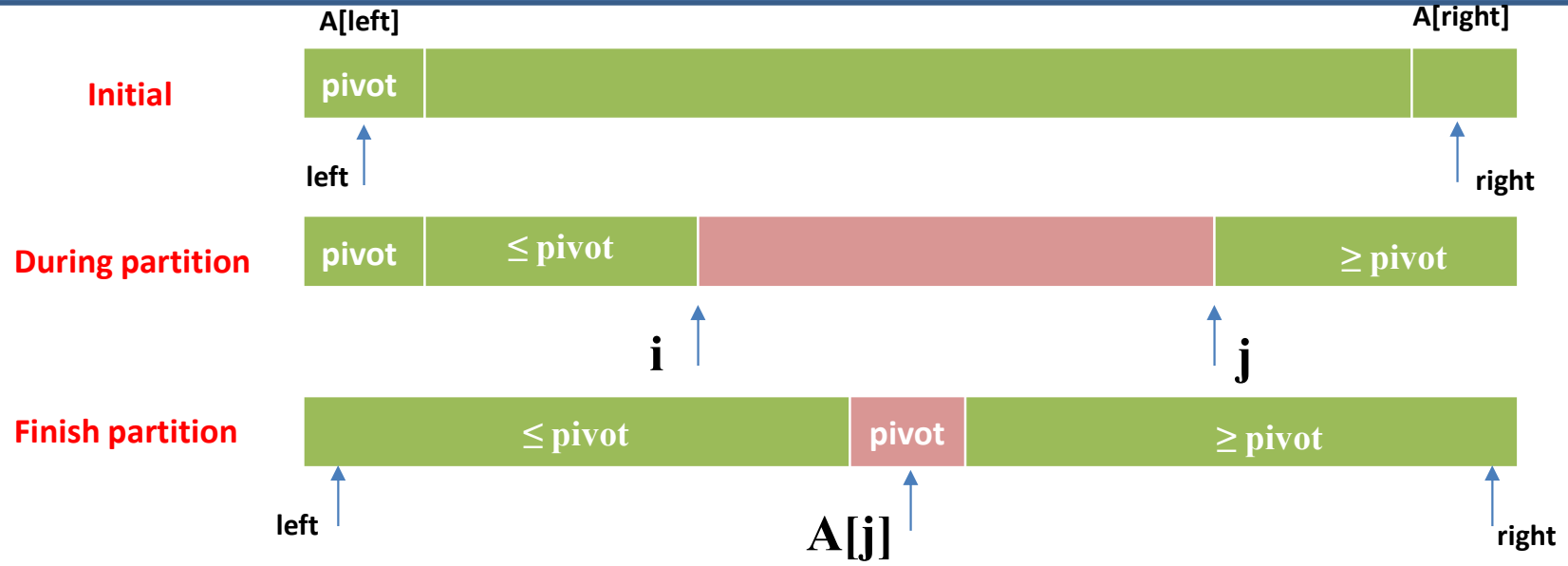
The pivot is the first element of array

The function Partition moves elements of the array A[left].....A[right] to obtain a new array such that:

- The pivot element goes to its final position (denoted as j^{th} position) in the sorted array
- Elements from A [left] ... A [j-1] are <= the pivot
- Elements from A [j + 1] ... A [right] are > = the pivot



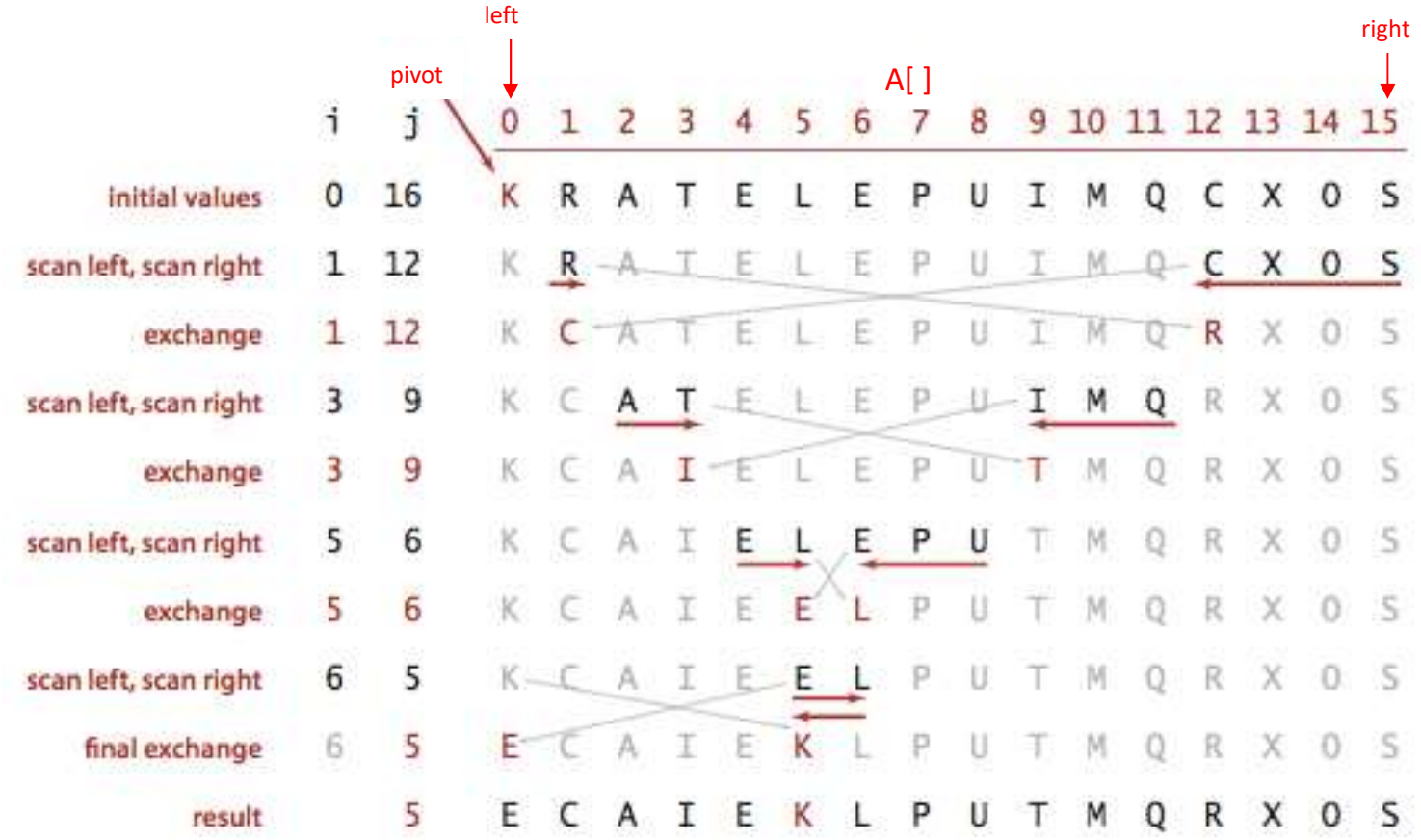
The pivot is the first element of array



- Select the first element of the array as the pivot: pivot = A[left] (when finish the function Partition, the pivot element is set to the position j^{th} which is its final position in the final sorted array A)
- i = left +1; Scan from left: Scan from element A[i] to the end of the array until we find the element FIRST1 >= pivot
- j=right; Scan from right: Scan from element A[j] to the beginning of the array until we find the element FIRST2 <= pivot
- SWAP (FIRST1, FIRST2) because they are out of place in the final sorted array
- Continue scanning from left and right to swap elements if necessary, stop scanning when i >= j
- Finally, SWAP(A[left] = pivot, A[j]) – swap the pivot element with the last element of left subarray
- return j

Example: the process of executing the partition function when selecting the last element of the array as the pivot

- *i* = left +1; Scan left: Scan from element A[*i*] to the end of the array until we find the element FIRST1 >= pivot
- *j*=right; Scan right: Scan from element A[*j*] to the beginning of the array until we find the element FIRST2 <= pivot
- SWAP (FIRST1, FIRST2) because they are out of place in the final sorted array
- Continue scanning from left and right to swap elements if necessary, stop scanning when *i* >= *j*
- Finally, SWAP(A[left] = pivot, A[j]) – swap the pivot element with the last element of left subarray



Partitioning trace (array contents before and after each exchange)

The pivot is the first element of the array

```
Partition(A, left, right)
    i = left; j = right + 1;
    pivot = A[left];
     while (true) do
     {
          i = i + 1;
          //Scan left: find the first element >= pivot:
          while  i ≤ right and A[i] < pivot do  i = i + 1;
          j = j – 1;
          //Scan right: find the first element <= pivot:
          while  j ≥ left and pivot < A[j] do  j = j – 1;
          if (i >= j) break;
          swap(A[i] , A[j]);
     }
    swap(A[j], A[left]);
   return j;
```

pivot is the first element of the array

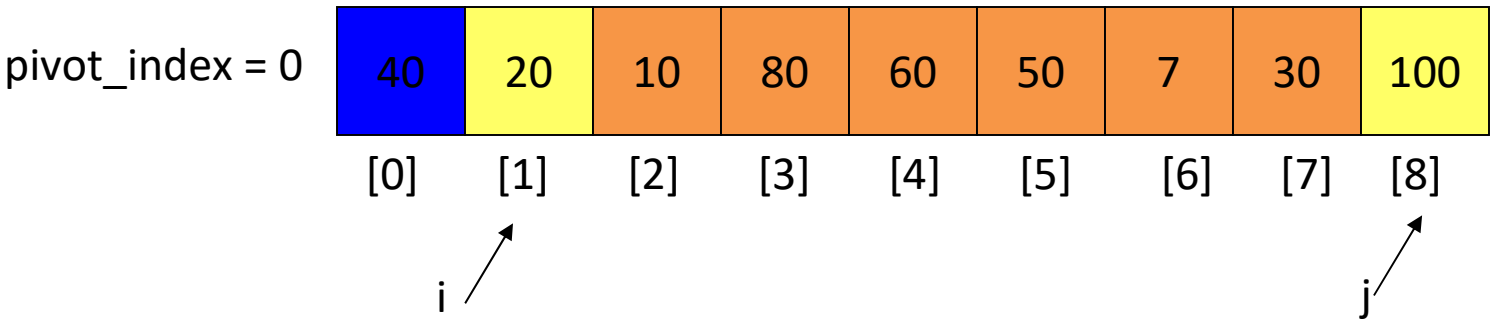
j is the index (jpivot) need to be returned, therefore, need to swap position of a[left] and a[*j*]

i

After selecting the *pivot,* move the pointer *i* and *j* from the beginning and the end of the array, and swap pair of elements *a*[*i*] and *a*[*j*] such that a[*i*] > *pivot* and a[*j*] <= *pivot*

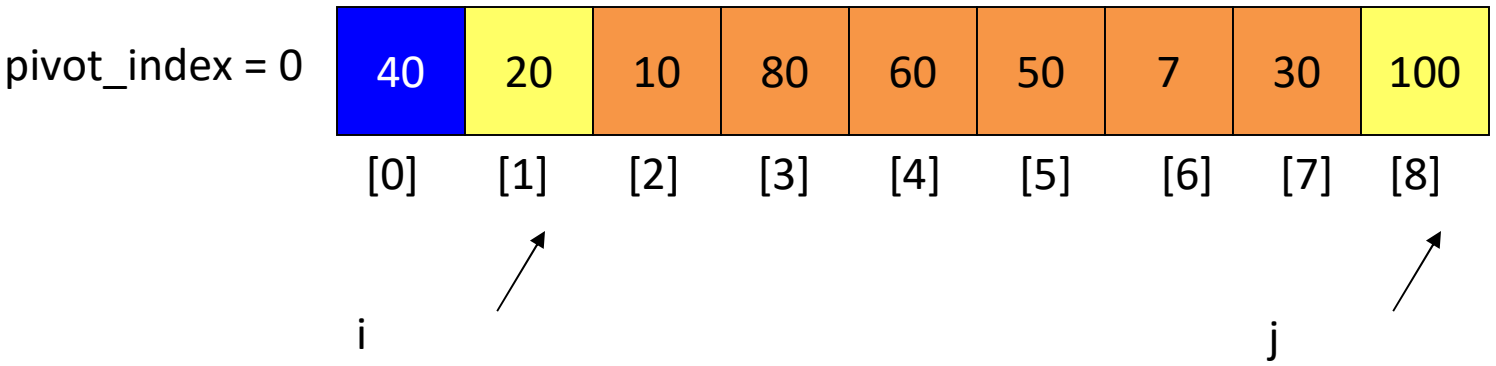
j

The pivot is the first element of the array



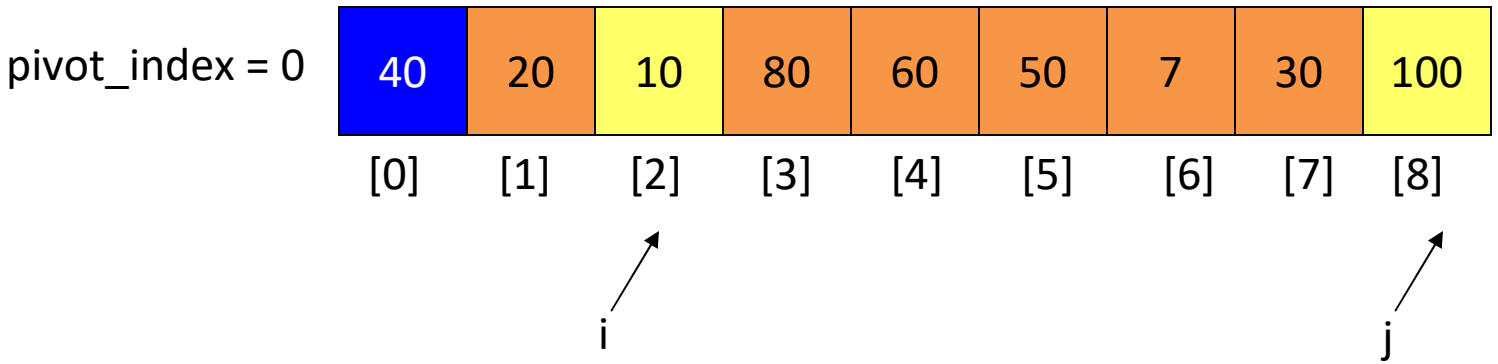
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i



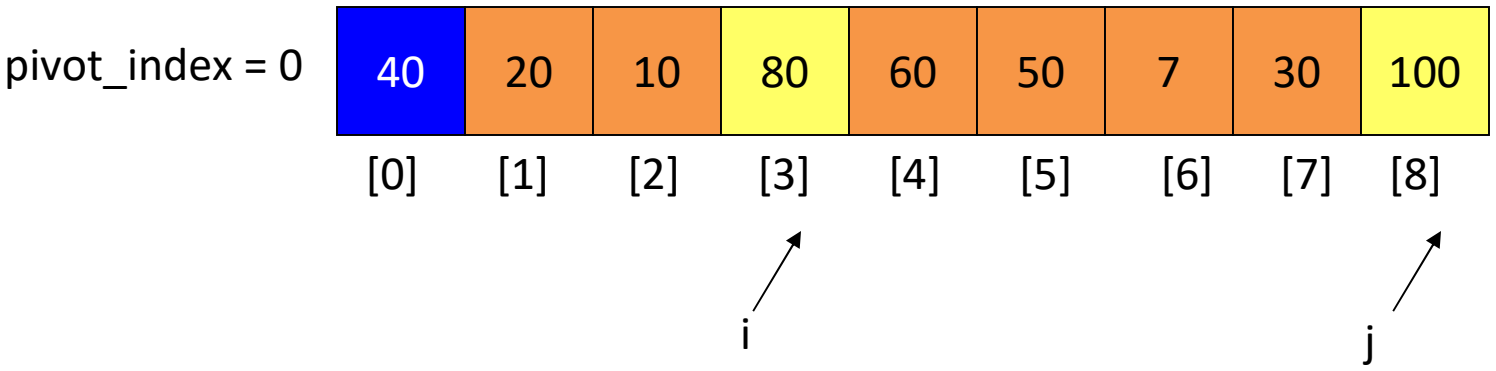
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i



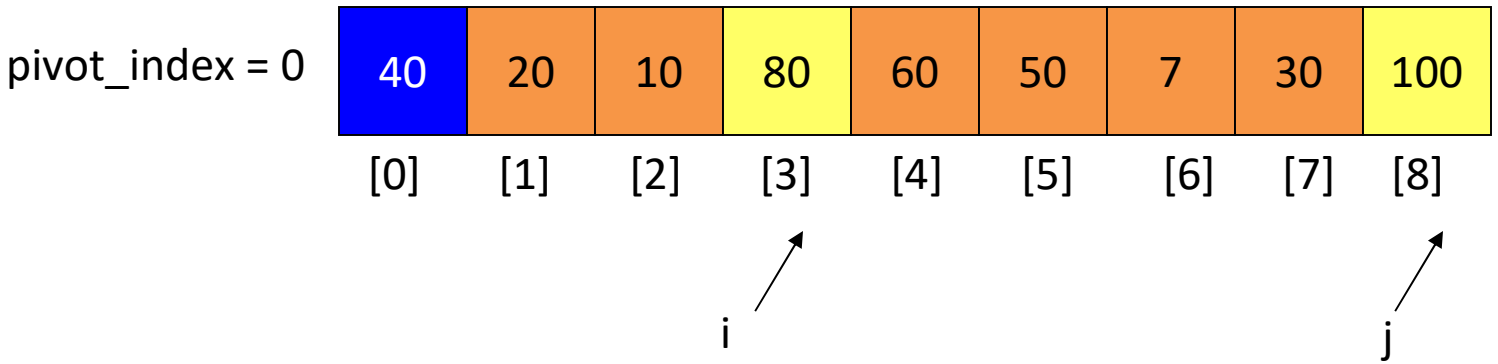
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i



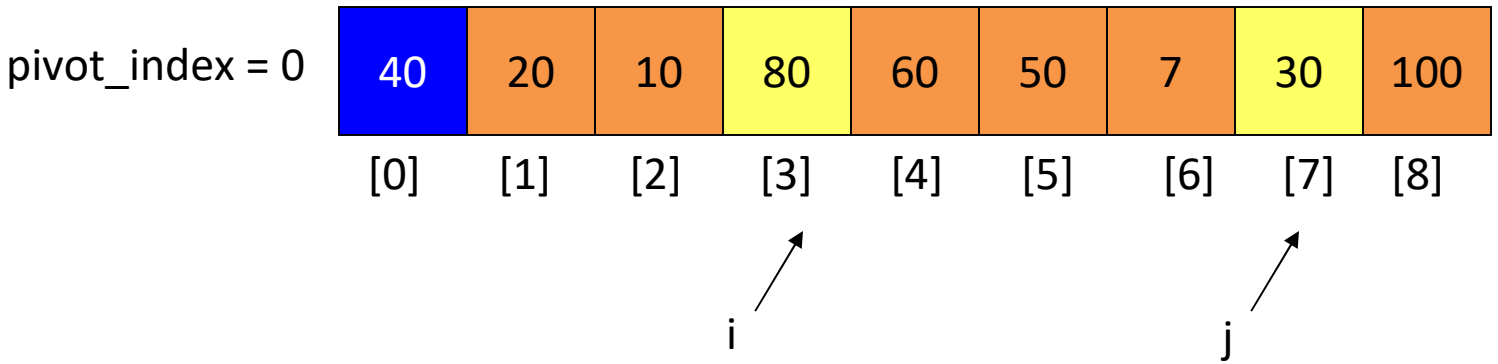
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j



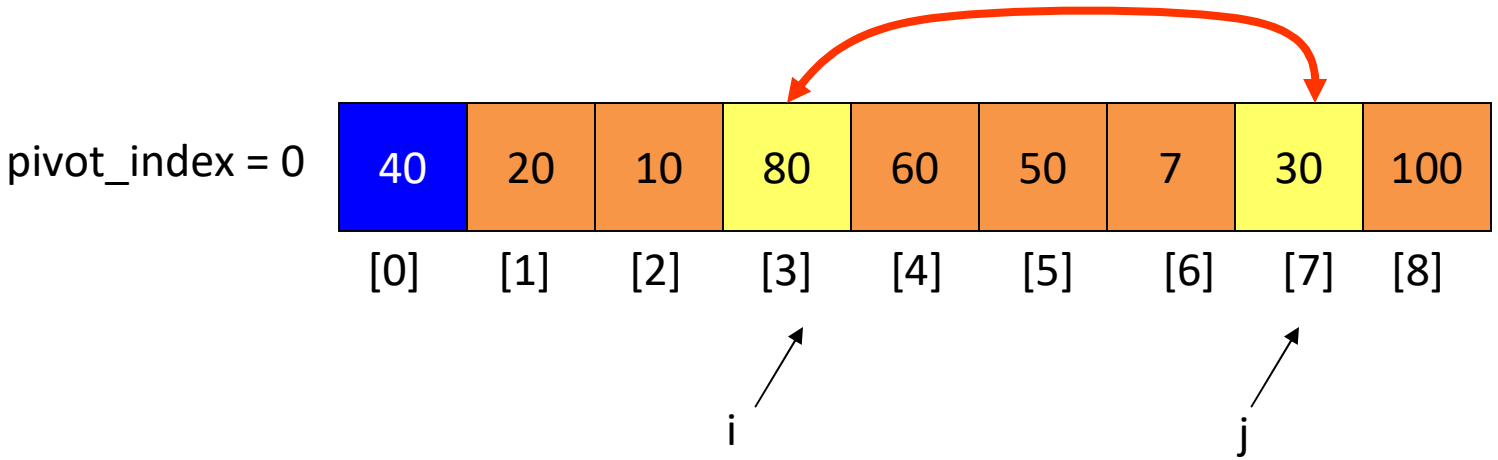
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j



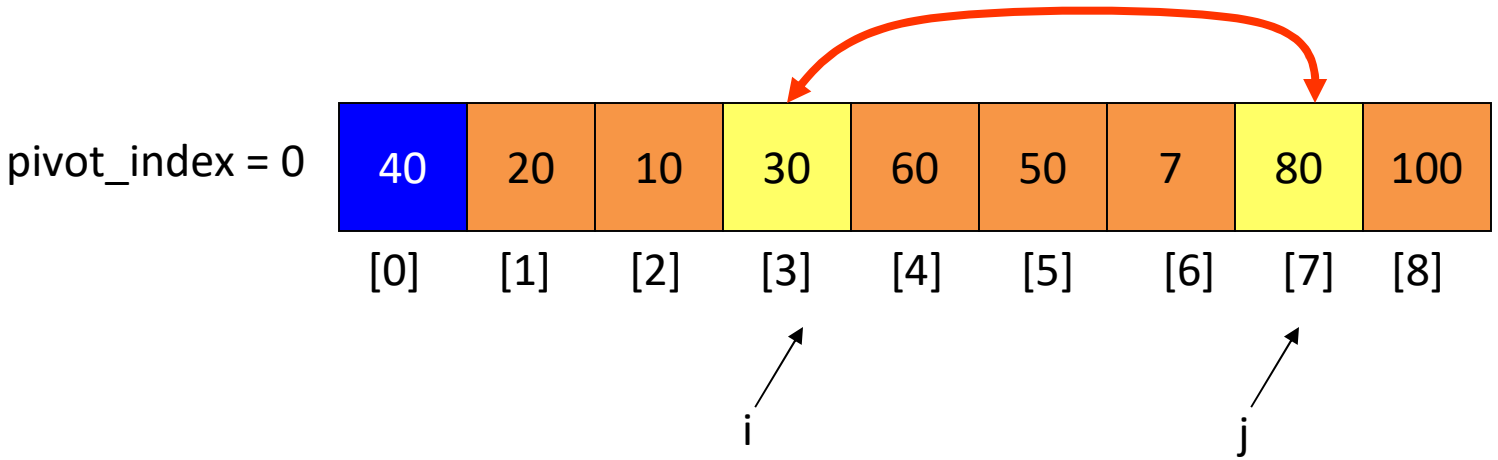
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])



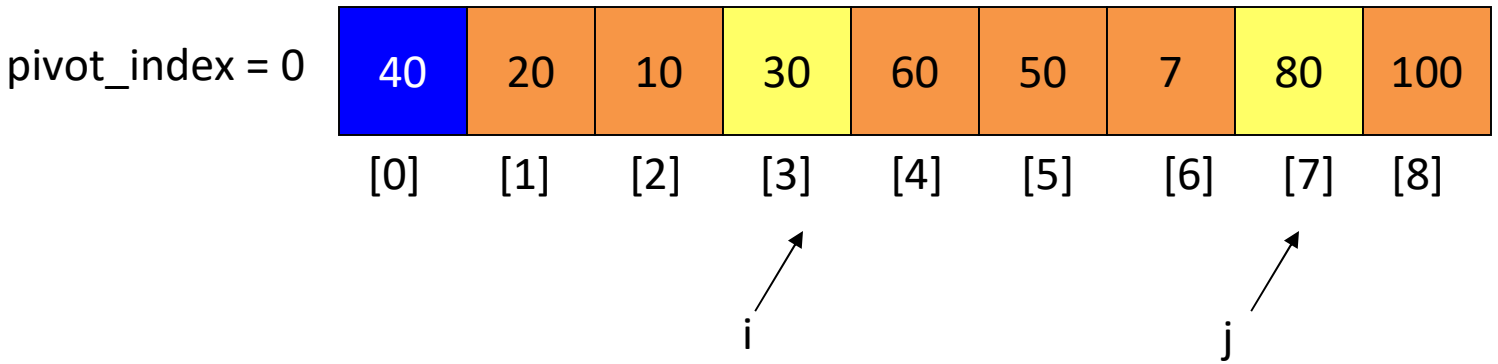
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])



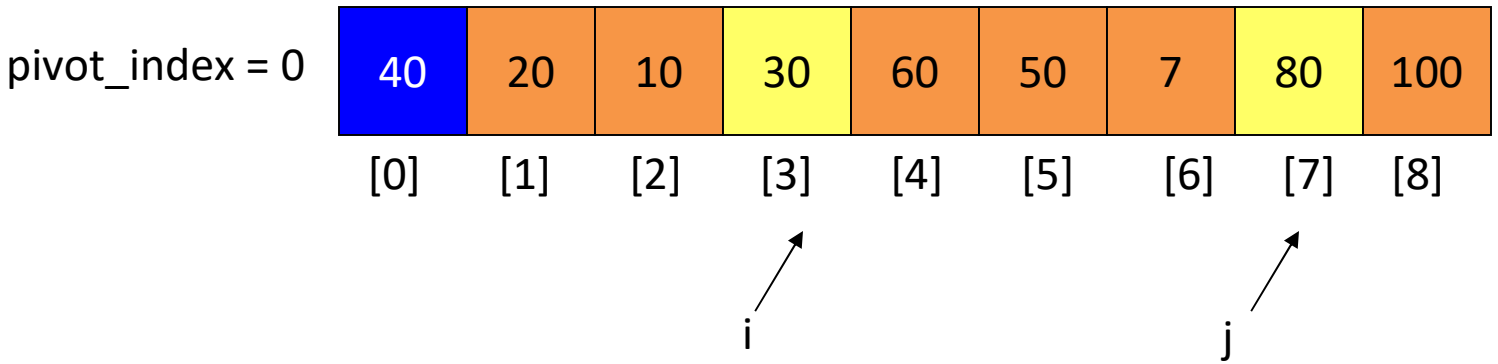
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



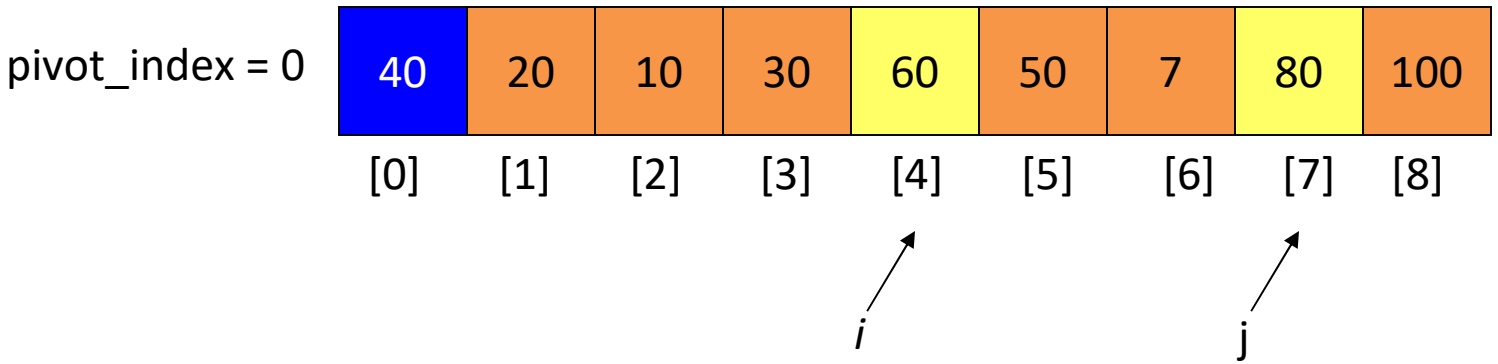
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



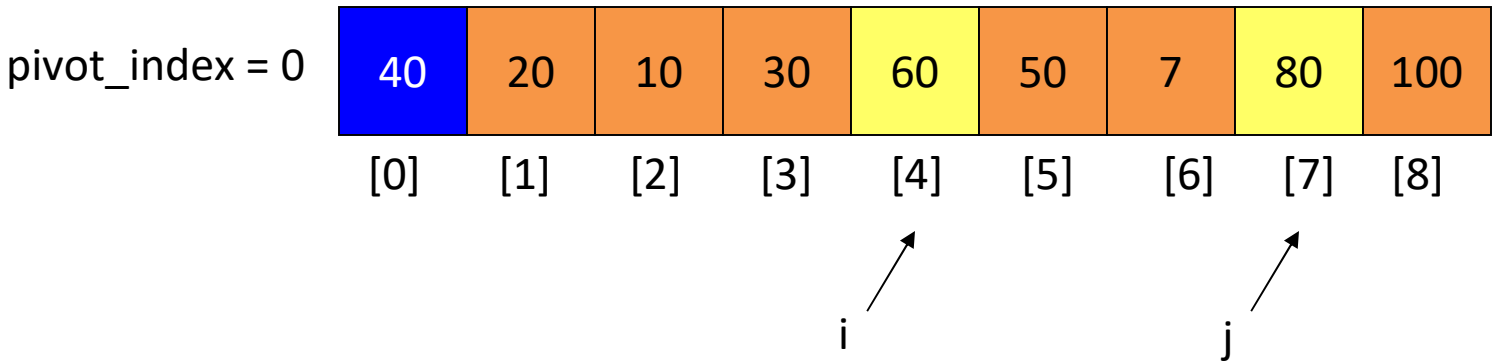
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



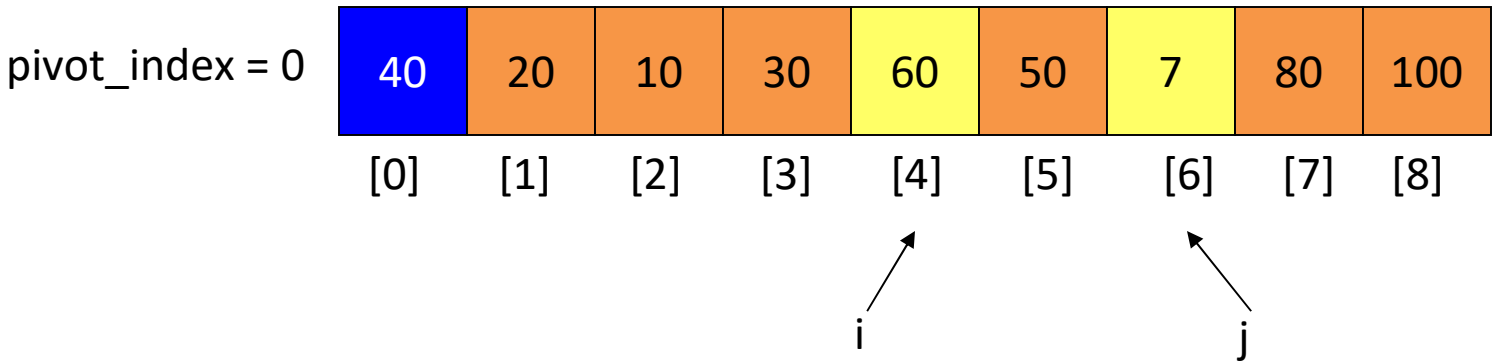
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



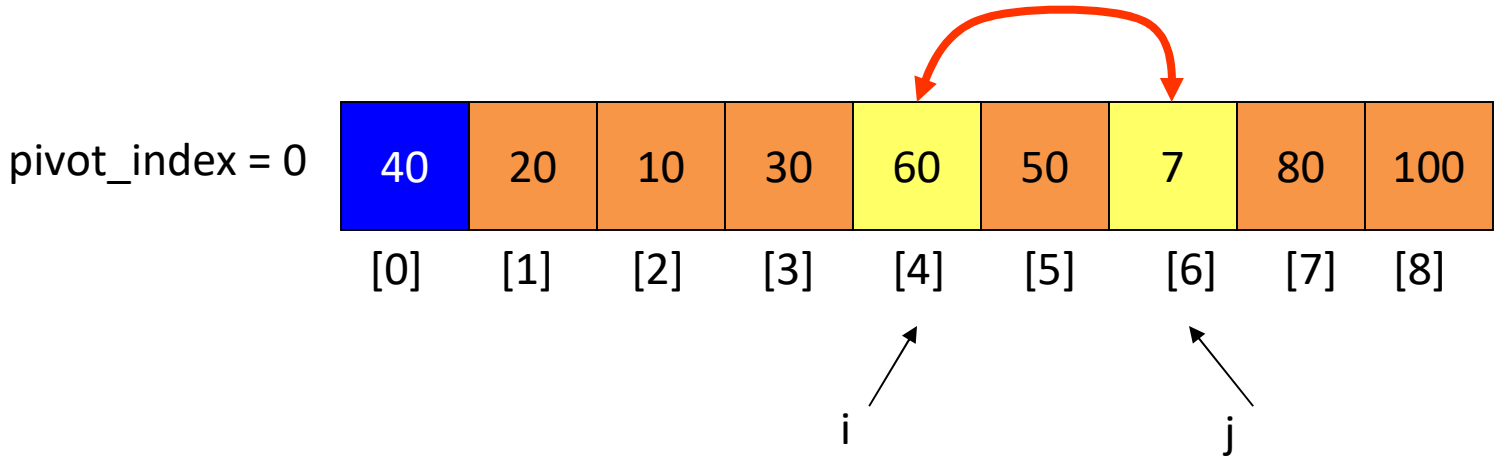
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



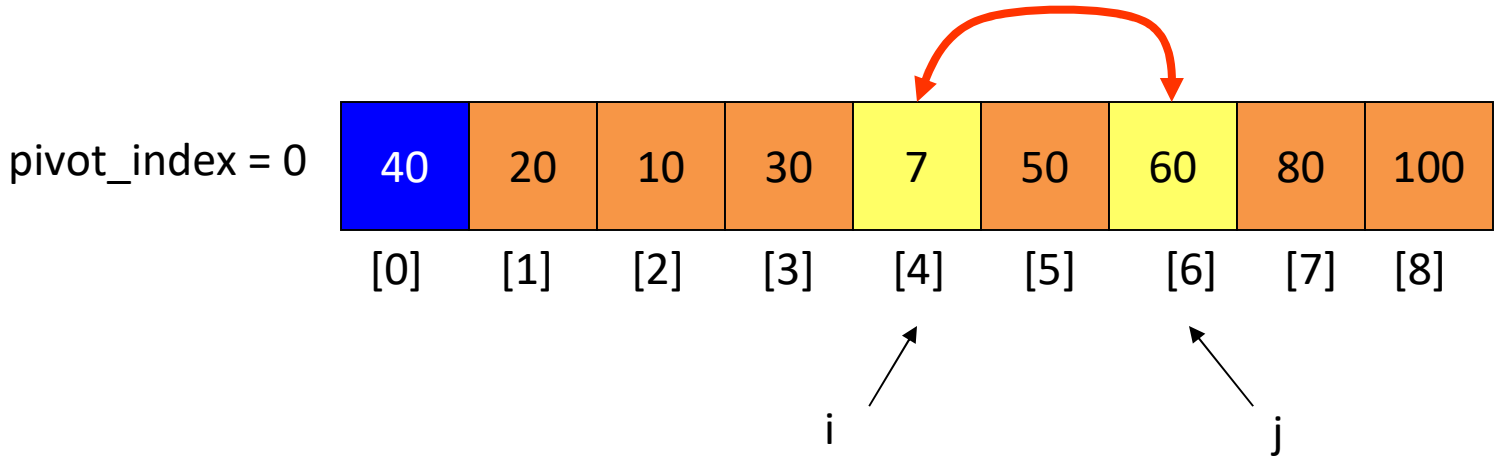
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



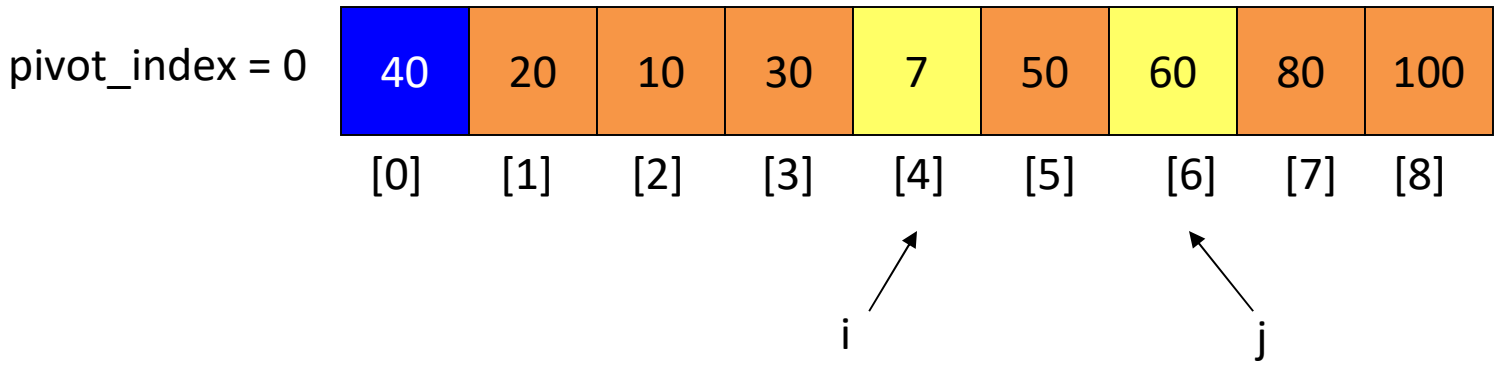
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



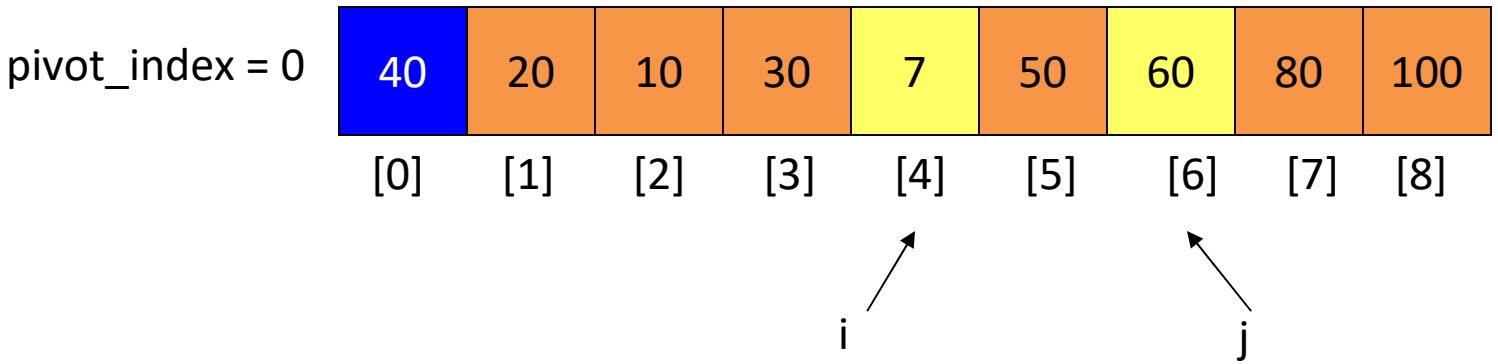
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



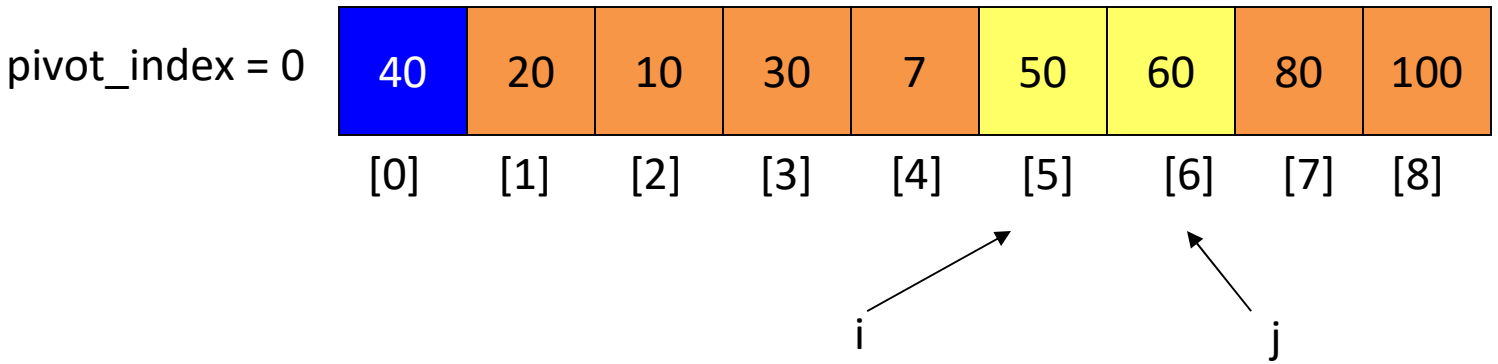
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



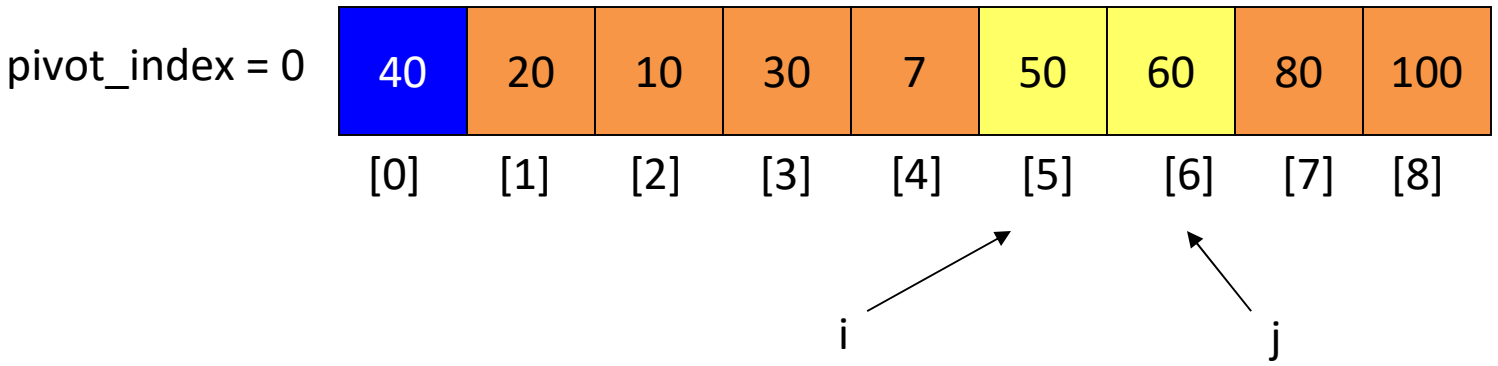
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



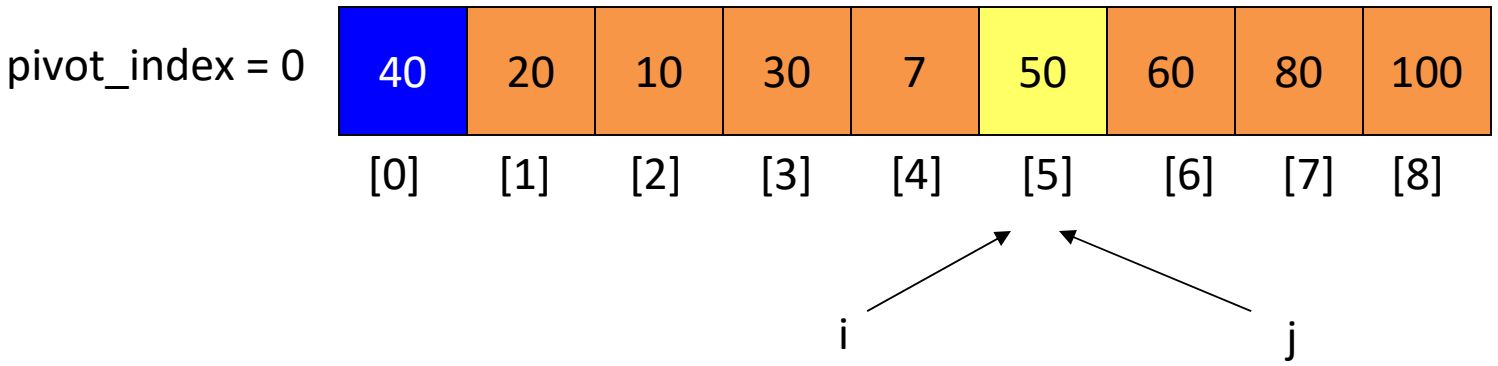
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



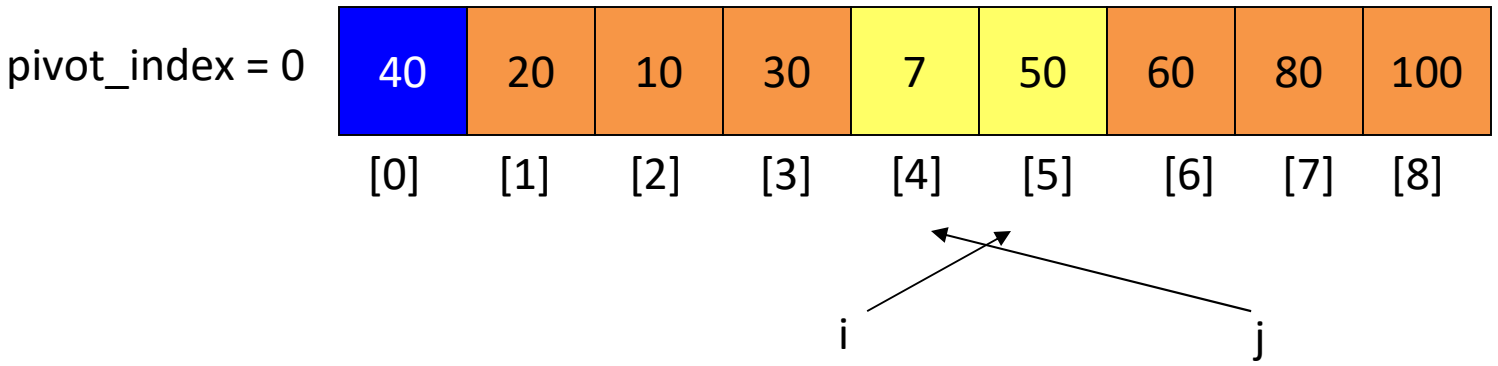
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



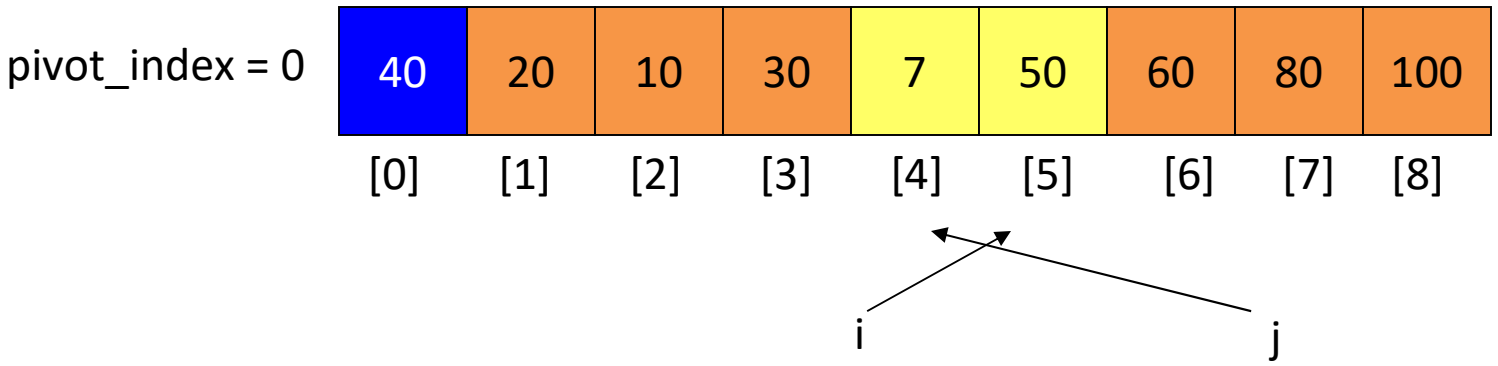
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



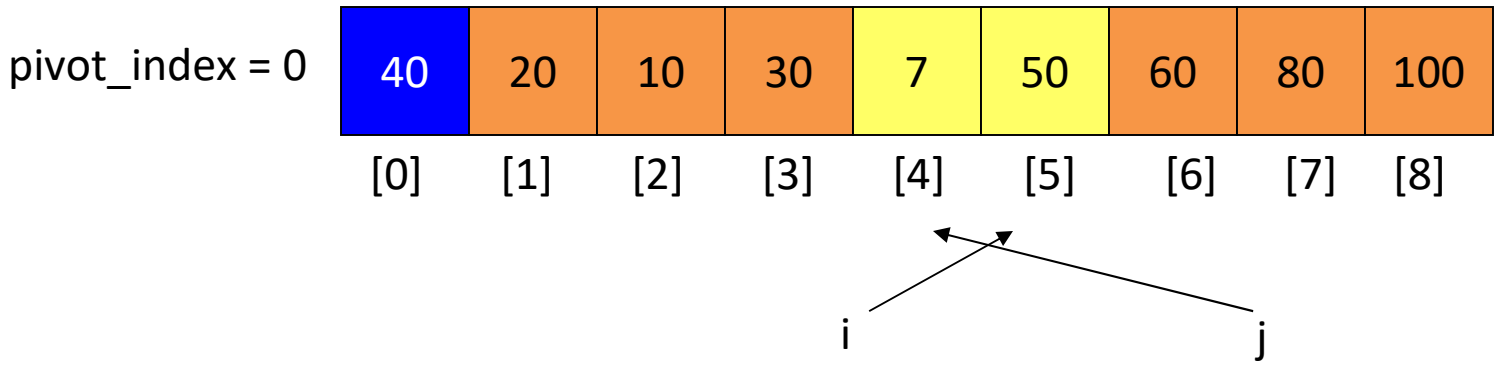
Phần tử chốt là phần tử đứng đầu

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



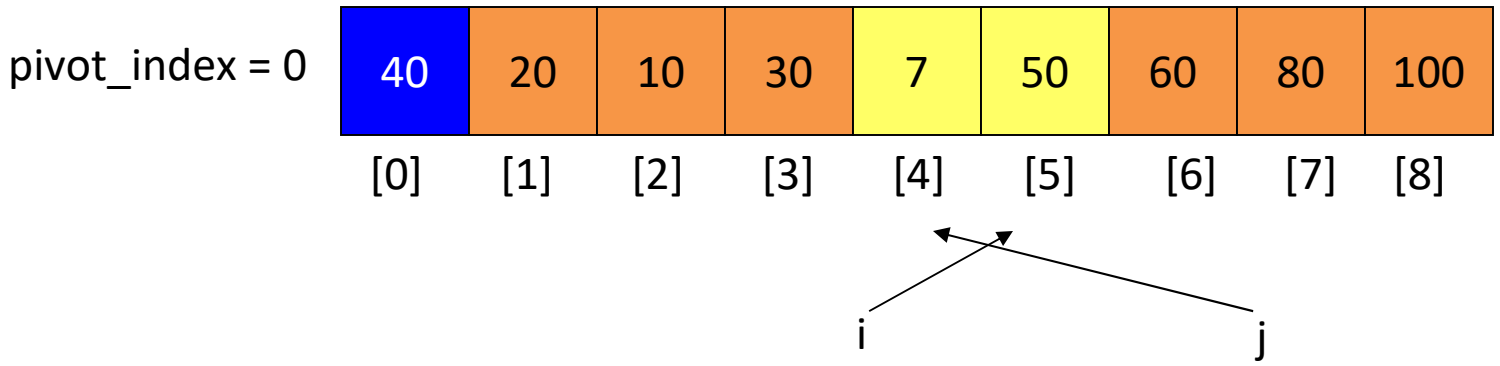
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.



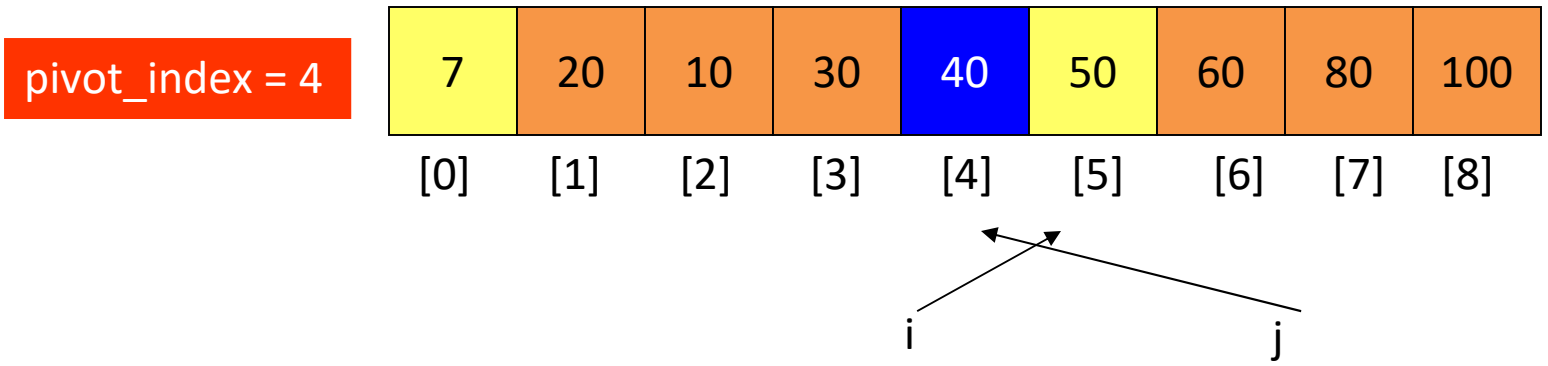
The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.

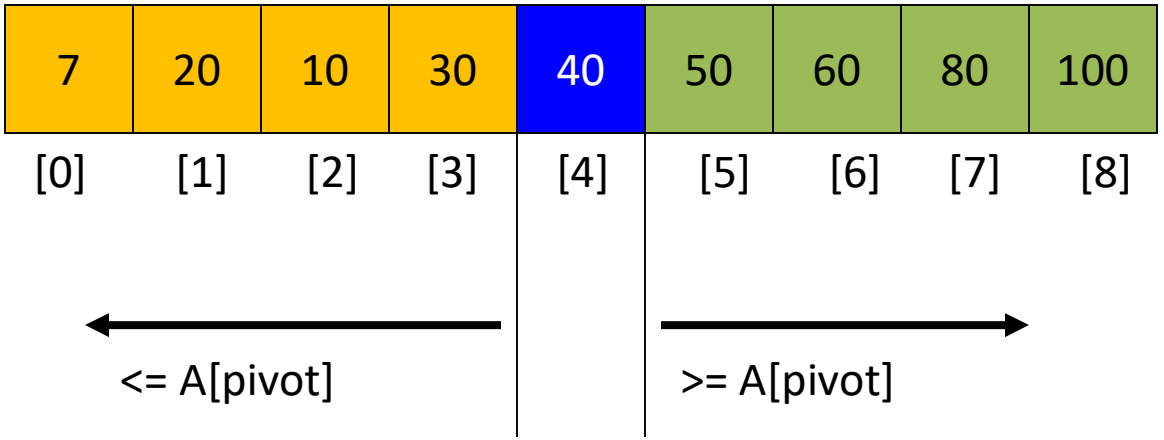


The pivot is the first element of the array

1. While i <= right and A[i] < pivot
   ++i
2. While j >=left and A[j] > pivot
   --j
3. If i < j
   swap(A[i], A[j])
4. While j > i, go to 1.
5. Swap(A[j], A[pivot_index])



The array obtained after calling: Partition(A,0,8);

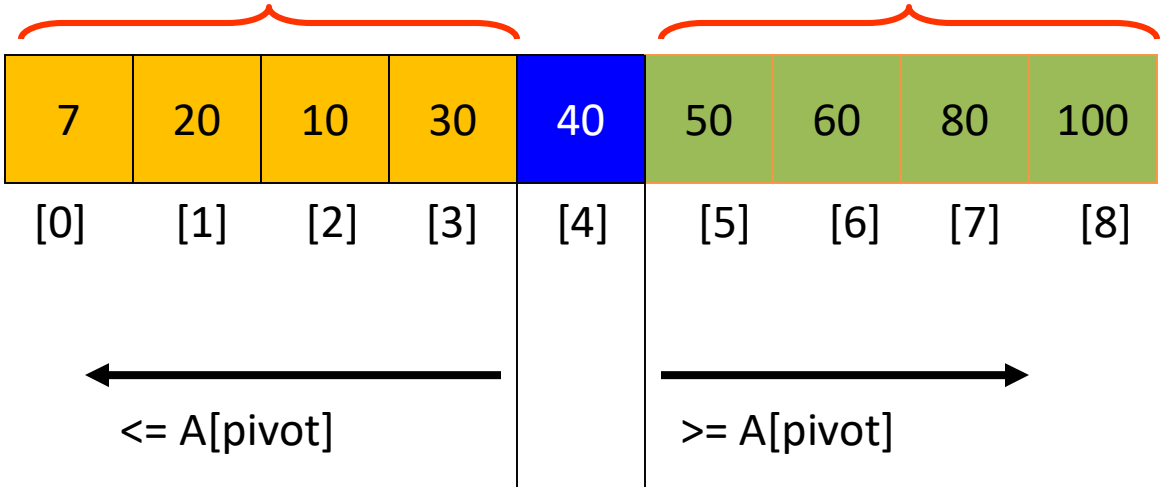



```
Quick-Sort(A, Left, Right)
1.    if  (Left < Right ) {
2.            Pivot = Partition(A, Left, Right);
3.            Quick-Sort(A, Left, Pivot –1);
4.            Quick-Sort(A, Pivot +1, Right);
5.     }
```

Quick-Sort(A,0,8);

Partition(A,0,8);

Recursion: Quicksort Sub-arrays




```
Quick-Sort(A, Left, Right)
1.    if (Left < Right ) {
2.          Pivot = Partition(A, Left, Right);
3.          Quick-Sort(A, Left, Pivot –1);
4.          Quick-Sort(A, Pivot +1, Right);
5.     }
```

```
Quick-Sort(A,0,8);

4 = Partition(A,0,8);
    Quick-Sort(A, 0, 3);  ⟶
    Quick-Sort(A, 5, 8);
```

Source code QuickSort:
The pivot is the first element of the array

```
//QuickSort: Chon phan tu dau lam phan tu chot:
int Partition(int A[], int left, int right)
{
    int i = left; int j = right+1;
    int pivot = A[left];
    while (true)
    {
        //Tim tu trai sang phan tu dau tien >=pivot:
        i = i + 1;
        while (i <= right && A[i] < pivot) i=i+1;
        //Tim tu phai sang phan tu dau tien <=pivot:
        j -=1;
        while (j >= left && pivot < A[j]) j =j-1;
        if (i >= j) break;
        swap(&A[i], &A[j]);
    }
    swap(&A[j], &A[left]);
    return j;
}
```

```
//QuickSort: Chon phan tu dau lam phan tu chot:
int Partition(int A[], int left, int right)
{
    int i = left; int j = right+1;
    int pivot = A[left];
    while (true)
    {
        //Tim tu trai sang phan tu dau tien >pivot:
        while (A[++i] < pivot)
            if (i==right) break;
        //Tim tu phai sang trai phan tu dau tien < pivot
        while (pivot < A[--j]);
        //  if (j==left) break; //khong can thiet vi phan tu chot a[left] acts as sentinel
        if (i >= j) break;
        swap(&A[i], &A[j]);
    }
    swap(&A[j], &A[left]);
    return j;
}
```

```
void swap(int *a, int *b)
{
    int temp =*a;
    *a=*b;
    *b=temp;
}
```

```
void QuickSort(int A[], int Left, int Right)
{
    int index_Pivot;
    if  (Left < Right )
    {
        index_Pivot =Partition(A,Left,Right);
        QuickSort(A,Left,index_Pivot-1);
        QuickSort(A, index_Pivot +1, Right);
    }
}
```

Source code QuickSort:
The pivot is the middle element of the array

```
int PartitionMid(int A[], int left, int right)
{
  int pivot = A[(left + right)/2];
  while (left < right){
    //Tim tu trai sang phai phan tu dau tien >= pivot:
    while (A[left] < pivot) left++;
    //Tim tu phai sang trai phan tu dau tien <= pivot:
    while (A[right] > pivot) right--;
    if (left < right)
    {
        //doi cho 2 phan tu do cho nhau:
        swap(&A[left],&A[right]);
        left++;right--;
    }
  }
  return right;
}

void QuickSort(int A[], int Left, int Right)
{
    int index_Pivot;
    if  (Left < Right )
    {
        index_Pivot =PartitionMid(A,Left,Right);
        QuickSort(A,Left,index_Pivot-1);
        QuickSort(A, index_Pivot +1, Right);
    }
}
```

Source code QuickSort:
The pivot is the last element of the array

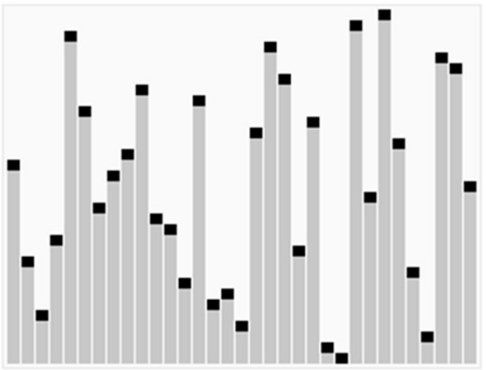
```
int PartitionLast(int A[], int left, int right) {
  int pivot = A[right];
  int j = left - 1;
  for (int i = left; i < right; i++) {
      if (pivot >= A[i])
      {
          j = j + 1;
          swap(&A[i], &A[j]);
      }
  }
  A[right] = A[j + 1];
  A[j + 1] = pivot;
  return (j + 1);
}
```

```
void QuickSort(int A[], int Left, int Right)
{
    int index_Pivot;
    if  (Left < Right )
    {
        index_Pivot =PartitionLast(A,Left,Right);
        QuickSort(A,Left,index_Pivot-1);
        QuickSort(A, index_Pivot +1, Right);
    }
}
```

5.5. Quick sort

- Developed by British computer scientist Tony Hoare in 1959 and published in 1961,
- Quick sort has the average running time $O(n \log n)$, however the worst running time is $O(n^2)$.
- Quick sort is in place, but not stable.
- Quick sort is easy in theory but difficult to implement.

C.A.R. Hoare
January 11, 1934
ACM Turing Award, 1980
Photo: 2006



Estimated running time

- Home computer: assume the computer can perform 10^8 comparisons per second
- Super computer: assume the computer can perform 10^{12} comparisons per second

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Contents

$O(n^2)$

Divide and conquer

$O(n \log_2 n)$

5.6. Heap sort (Sắp xếp vun đống)

5.6.1. Heap data structure (Cấu trúc dữ liệu đống)

5.6.2. Heap sort (Sắp xếp vun đống)

5.6. Heap sort (Sắp xếp vun đống)

5.6.1. Heap data structure (Cấu trúc dữ liệu đống)

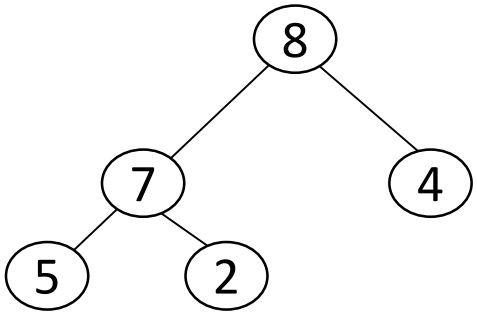
5.6.2. Heap sort (Sắp xếp vun đống)

The Heap Data Structure

- Definition: A **heap** is a binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x

$$\text{Parent}(x) \geq x : \textit{max-heap}$$

$$\text{OR} \quad \text{Parent}(x) \leq x : \textit{min-heap}$$



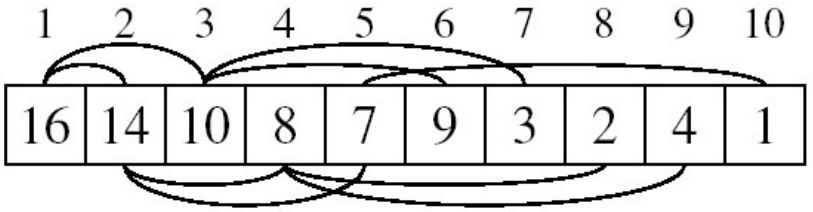
From the heap property, it follows that:
"The root is the maximum element of the max-heap!"

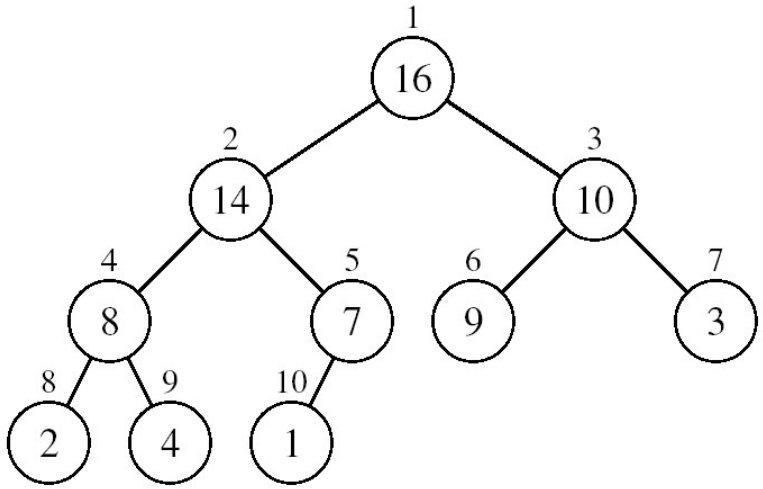
MAX-Heap

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is A[1]
  - Left child of A[i] = A[2*i]
  - Right child of A[i] = A[2*i + 1]
  - Parent of A[i] = A[$\lfloor i/2 \rfloor$]
  - Heapsize[A] ≤ length[A]
- The elements in the subarray A[($\lfloor n/2 \rfloor$+1) .. n] are leaves





Heap Types

- Max Heap
  - Used to sort array in ascending order
  - Has property: for all nodes *i*, excluding the root:

    $A[PARENT(i)] \geq A[i]$
- Min Heap
  - Used to sort array in descending order
  - Has property: : for all nodes *i*, excluding the root:

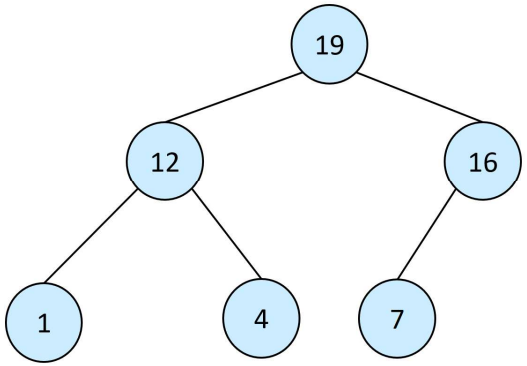
    $A[PARENT(i)] \leq A[i]$

MAX-HEAP EXAMPLE:

Max-heap as an array:

19	12	16	1	4	7

Max-heap as a binary tree:



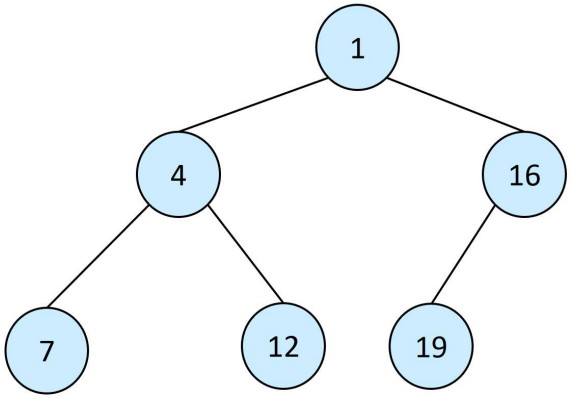
Last row filled from left to right.

MIN-HEAP EXAMPLE:

Min-heap as an array:

1	4	16	7	12	19

Min-heap as a binary tree:



Last row filled from left to right.

Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP

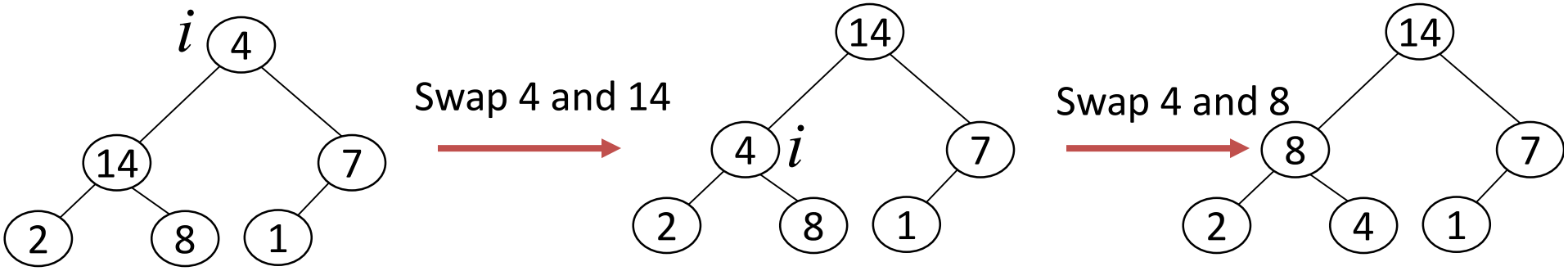
Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP

Max-heap: MAX-HEAPIFY

- Max Heap
  - Has property: for all nodes *i*, excluding the root:
    $A[PARENT(i)] \geq A[i]$

- If there exists a node *i* is smaller than its child (left and right subtree of node *i* are max-heaps), then it is violated the max-heap property, we need to eliminate it by calling MAX-HEAPIFY(A, i. n)
  - Exchange node *i* with its larger child
  - Move down the tree
  - Continue until node is not smaller than its children



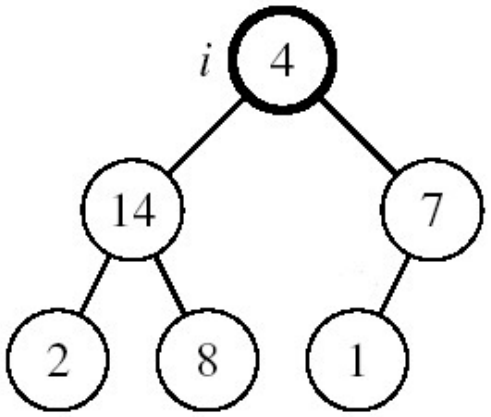
It is not max-heap because: Max-heap property is violated

It is not max-heap because: Max-heap property is violated

Max heap

MAX-HEAPIFY: Maintaining the Max-Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - $A[i]$ may be smaller than its children

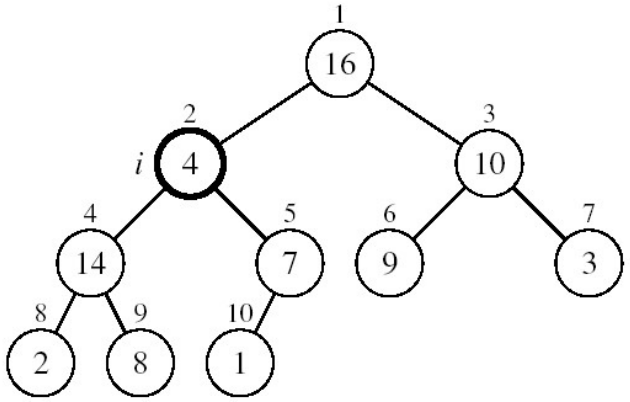


Alg: MAX-HEAPIFY(A, i, n)

1. l ← LEFT(i)
2. r ← RIGHT(i)
3. **if** l ≤ n and A[l] > A[i]
4. **then** largest ← l
5. **else** largest ← i
6. **if** r ≤ n and A[r] > A[largest]
7. **then** largest ← r
8. **if** largest ≠ i
9. **then** exchange A[i] ↔ A[largest]
10. MAX-HEAPIFY(A, largest, n)

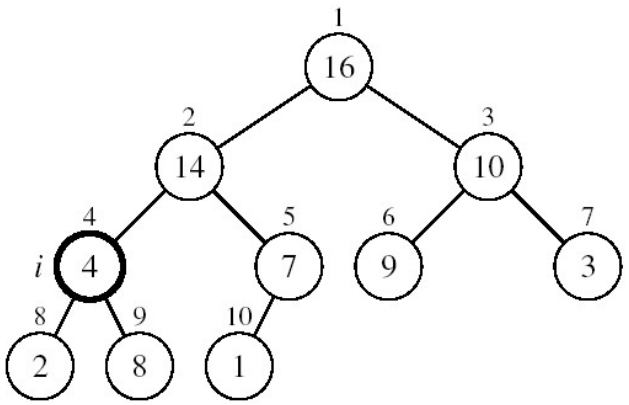
Example

MAX-HEAPIFY
- Exchange node *i* with its larger child
- Move down the tree
- Continue until node is not smaller than its children



A[2] violates the max-heap property

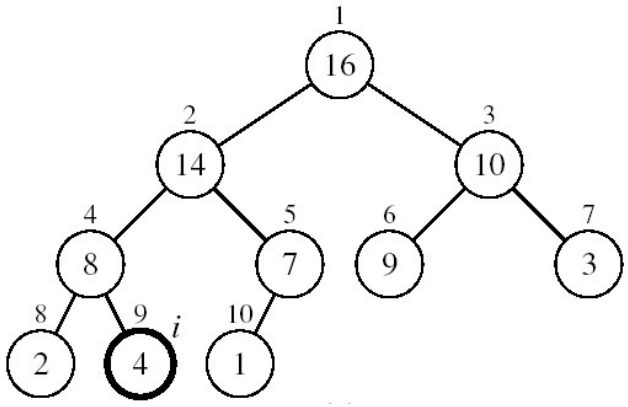
A[2] ↔ A[4]



A[4] violates the max-heap property

➔CALL: **MAX-HEAPIFY(A, 2, 10);**
to restore the max-heap property

A[4] ↔ A[9]



Heap property restored

MAX-HEAPIFY(A,2,10) is terminated; and we get a Max heap

Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- **Create a max-heap from an unordered array**
  - **BUILD-MAX-HEAP**

Create a max-heap from an unordered array: BUILD-MAX-HEAP

We want to convert an array $A[1 \ldots n]$ into a max-heap ($n = length[A]$) in which the elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves:

- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

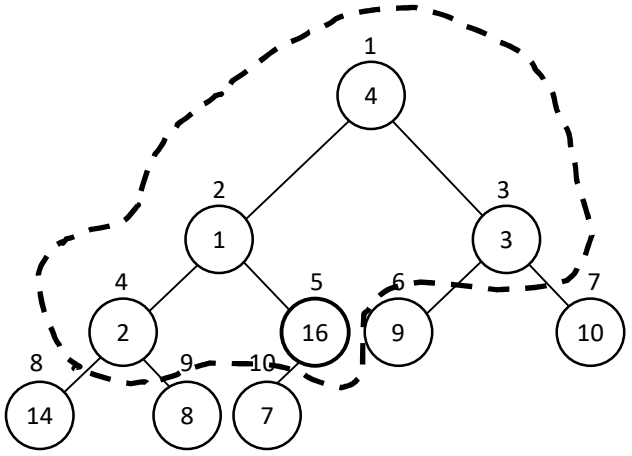
Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** i ← $\lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)

A:

4	1	3	2	16	9	10	14	8	7

Apply MAX-HEAPIFY on internal nodes A[$\lfloor n/2 \rfloor$] … A[1]

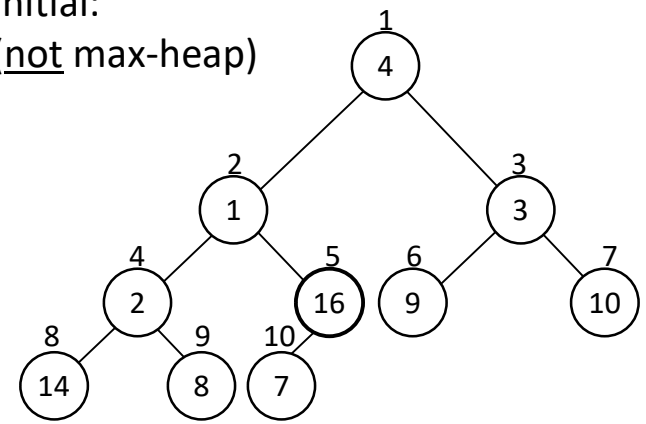


Example: given an array A, build max-heap to represent A

A:

4	1	3	2	16	9	10	14	8	7

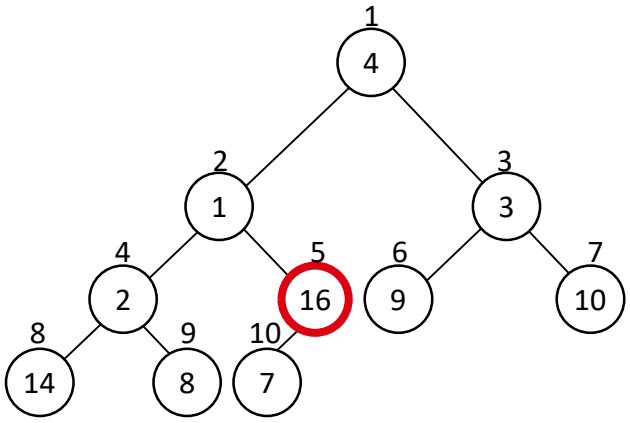
Initial:
(not max-heap)



To make this tree become max-heap: we need to apply MAX-HEAPIFY on all internal nodes:
A[$\lfloor n/2 \rfloor$] …A[1]

$\lfloor 10/2 \rfloor = 5$

$i = 5$: CALL MAX-HEAPIFY(A,5,10)

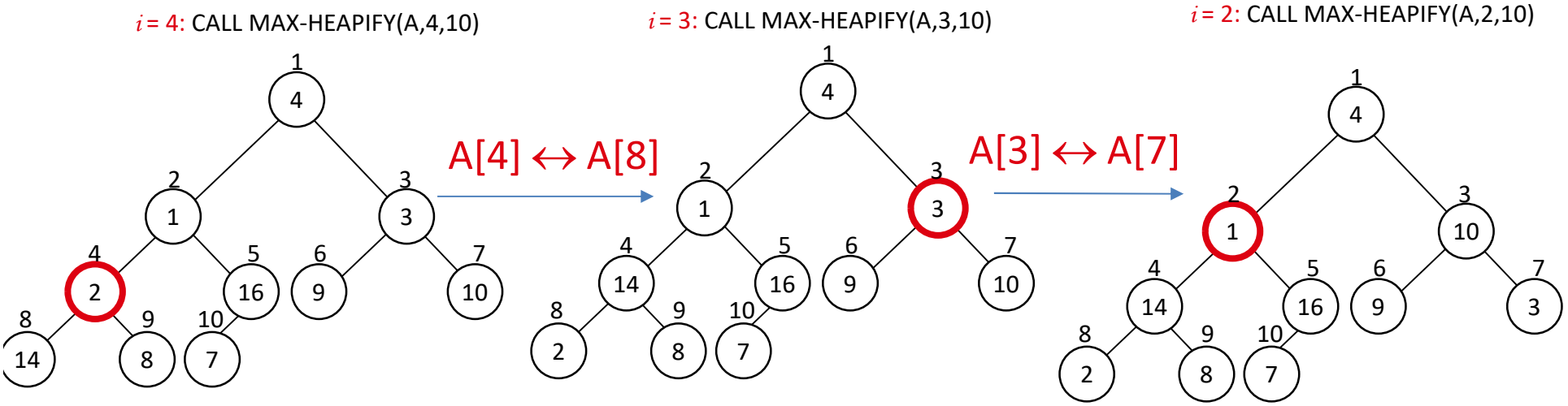


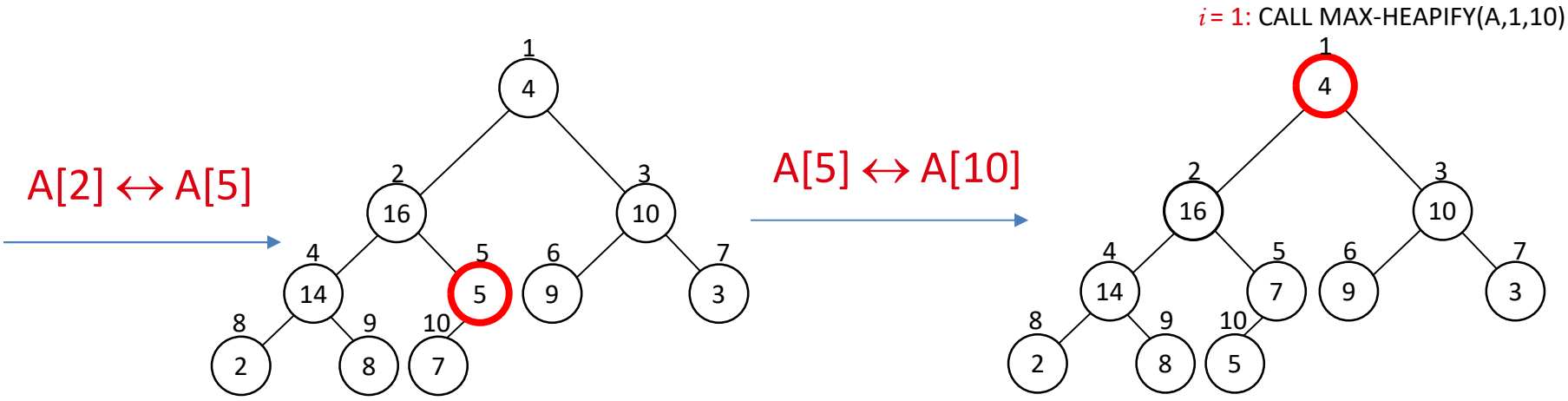
A[5] = 16 is greater than all its children (A[10]=7)
➔ ok; don't need to do anything

Example: given an array A, build max-heap to represent A

A:

4	1	3	2	16	9	10	14	8	7



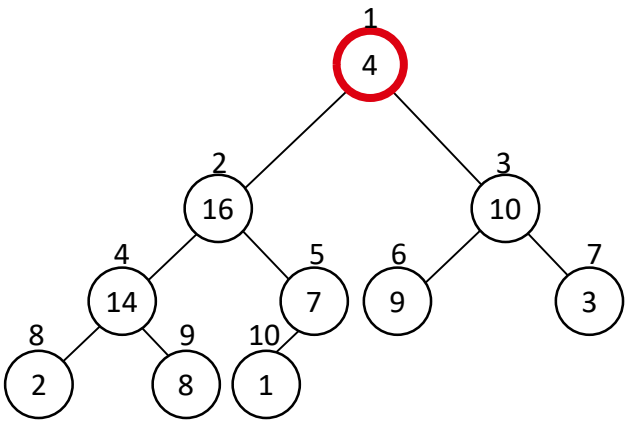


Example: given an array A, build max-heap to represent A

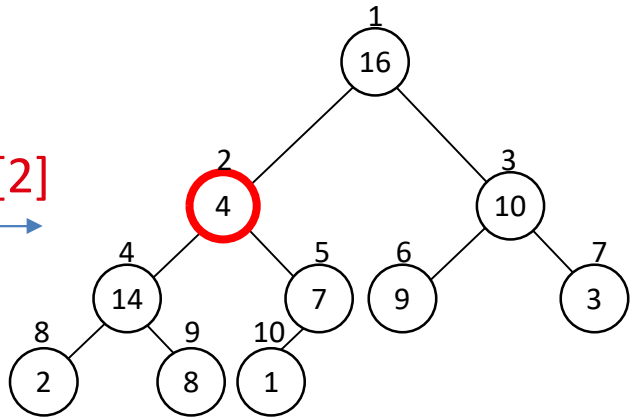
A:

4	1	3	2	16	9	10	14	8	7

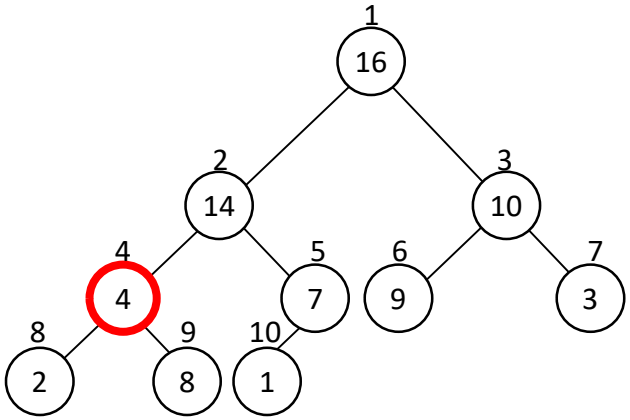
i = 1: CALL MAX-HEAPIFY(A,1,10)



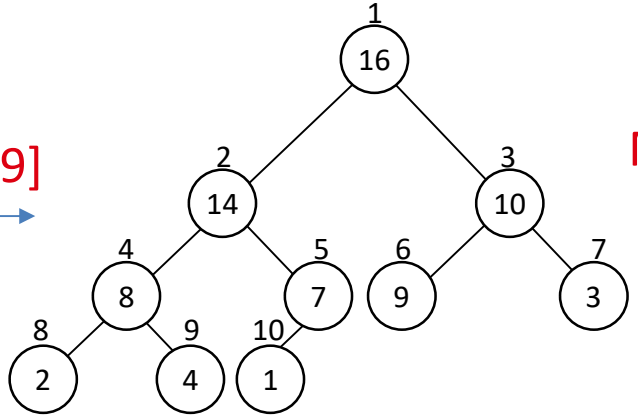
A[1] ↔ A[2]



A[2] ↔ A[4]



A[4] ↔ A[9]



Max heap

BUILD-MAX-HEAP: Complexity

Alg: BUILD-MAX-HEAP(*A*)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(*A*, i, n)

A:

4	1	3	2	16	9	10	14	8	7

Quick evaluation:

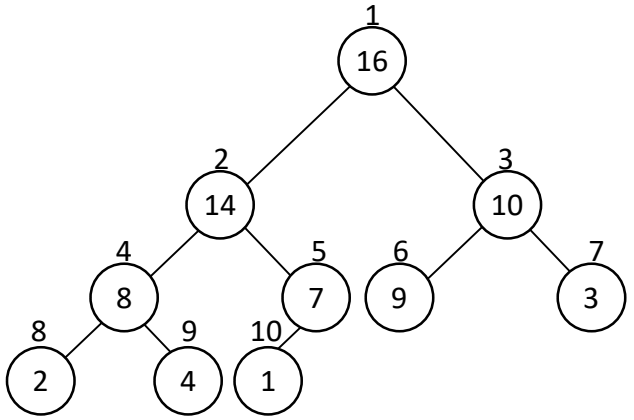
- O(n) times call `MAX_HEAPIFY`
- Each call `MAX_HEAPIFY` takes O($\log_2 n$)

➔ Complexity of `BuildMaxHeap` = O($n \log_2 n$)

BUILD-MAX-HEAP: Evaluate complexity more precisely

Alg: BUILD-MAX-HEAP(*A*)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(*A*, i, n)



- Number of nodes with the height h $\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Height of heap heap = $\log_2 n$
- Complexity of `MAX_HEAPIFY` on a node of height h is `O(h)`
- So, total time complexity = $\sum_{h=0}^{log_2 n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor log_2 n \rfloor} \frac{h}{2^h})$

As $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{\left(1-\frac{1}{2}\right)^2} = 2$ replace x =1/2 into $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ $when\ |x| < 1$

➔ Complexity of `BuildMaxHeap = O(n)`

5.6. Heap sort (Sắp xếp vun đống)

5.6.1. Heap data structure (Cấu trúc dữ liệu đống)

5.6.2. Heap sort (Sắp xếp vun đống)

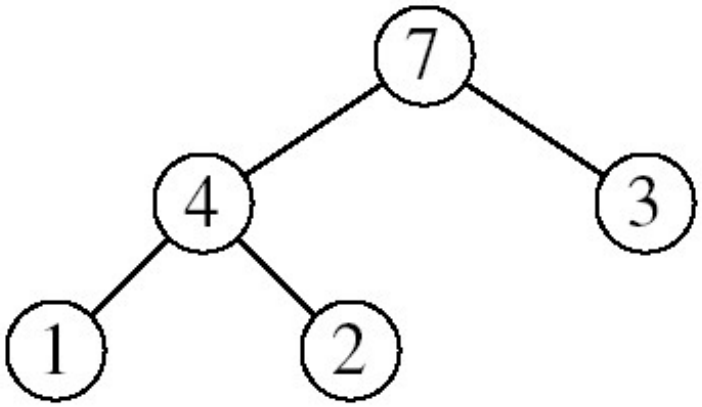
Heapsort algorithm

Goal: Sort an array in ascending order using heap representations

Idea:

1. Use procedure **BUILD-MAX-HEAP** to build a max-heap on the input array A[1 . . *n*].
2. Since the maximum element of the array is now stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*] (the last element in *A*).
3. Discard node *n* from the max-heap by decreasing the max-heap size.
4. The new element at the root may violate the max-heap property. Thus, we need to call **MAX-HEAPIFY** on the new root to restore the max-heap property.

Repeat this process (2-3-4) until only one node remains



Alg: HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*)
2. n = length[A]
3. **for** i ← n **downto** 2 {
4. exchange *A*[1] ↔ *A*[i]
5. MAX-HEAPIFY(*A*, 1, i - 1)
6. }

Alg: HEAPSORT(*A*)

Alg: HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*) $O(n)$
2. n = length[A]
3. **for** i ← n **downto** 2 {
4. exchange *A*[1] ↔ *A*[i]
5. MAX-HEAPIFY(*A*, 1, i - 1) $O(\log_2 n)$
6. }

n-1 times

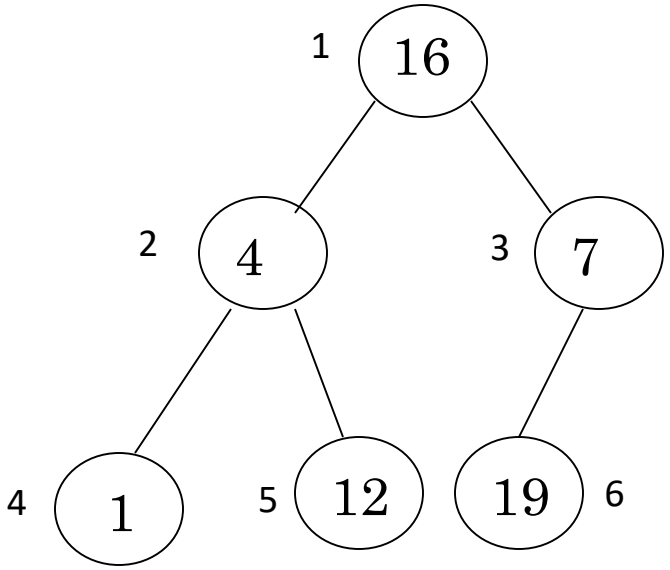
➔ Running time: $O(n \log_2 n)$

Example 1: Sort array A in ascending order

A:

16	4	7	1	12	19
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]

- Step 1: Represent array as a complete binary tree



Alg: HEAPSORT(*A*)

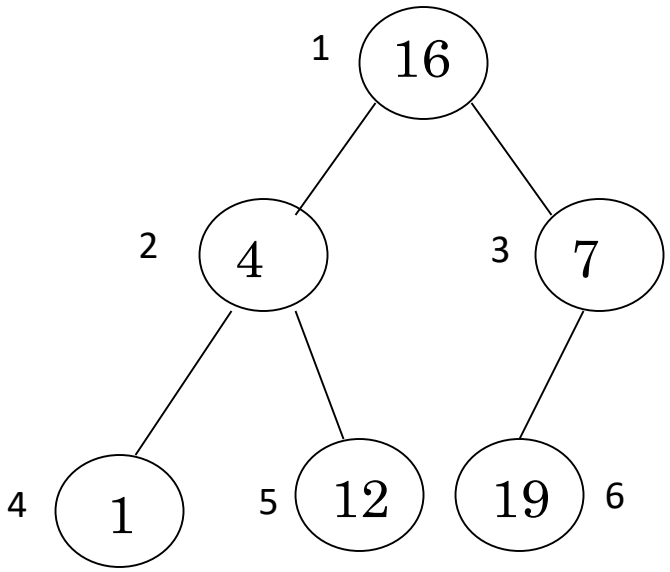
1. BUILD-MAX-HEAP(*A*)
2. n = length[A]
3. **for** i ← n **downto** 2 {
4. exchange *A*[1] ↔ *A*[i]
5. MAX-HEAPIFY(*A*, 1, i - 1)
6. }

- Step 2: Call HEAPSORT(A):

1. Call BUILD-MAX-HEAP(A): to make this tree become max-heap

Example 1: Sort array A in ascending order

- Step 2: Call HEAPSORT(A):
  - BUILD-MAX-HEAP(A)
    - $n = 6$



Alg: BUILD-MAX-HEAP(*A*)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(*A*, i, n)

To make this tree become max-heap: we need to apply MAX-HEAPIFY on all internal nodes:

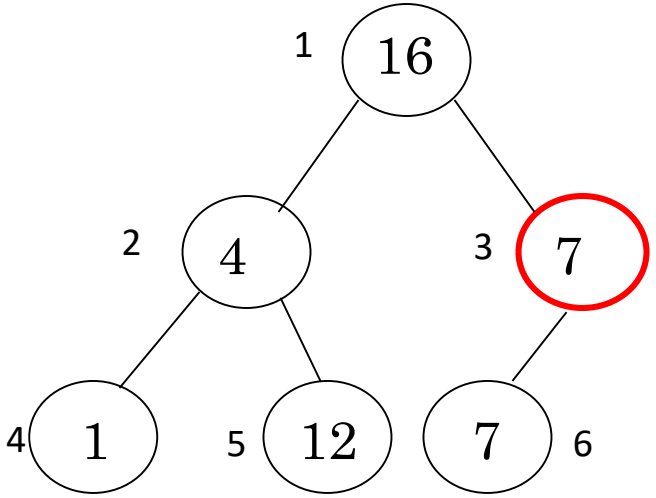
A[⌊n/2⌋] …A[1]

⌊ 6/2 ⌋ = 3

BUILD-MAX-HEAP(A):
call MAX-HEAPIFY on all internal nodes

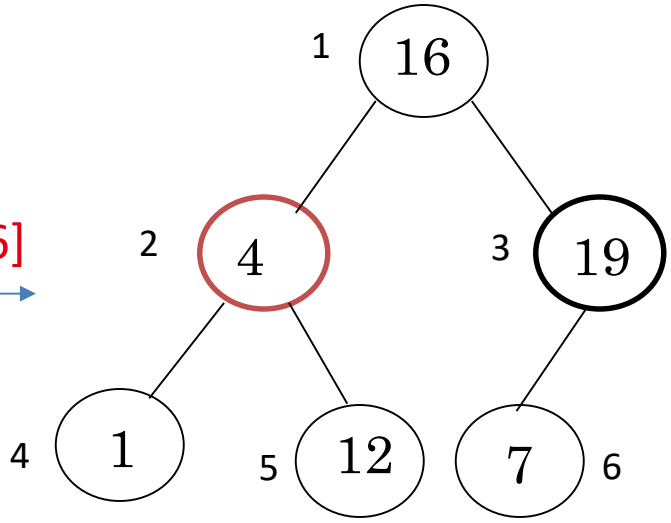
MAX-HEAPIFY
- Exchange node *i* with its larger child
- Move down the tree
- Continue until node is not smaller than its children

i = 3: Call MAX-HEAPIFY(A,3,6)



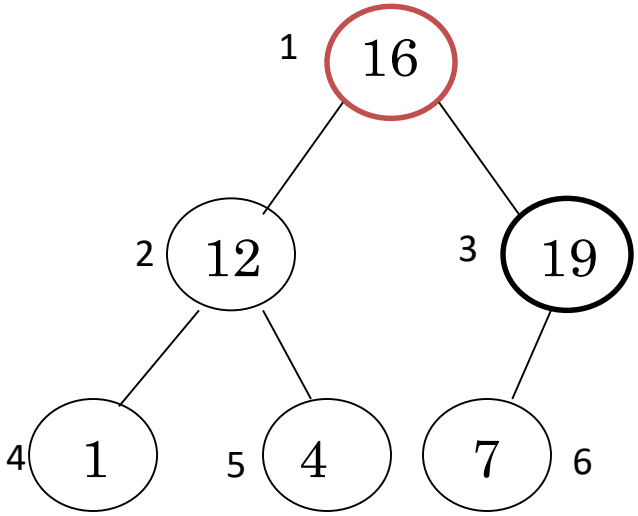
A[3] ↔ A[6]

i = 2: Call MAX-HEAPIFY(A,2,6)

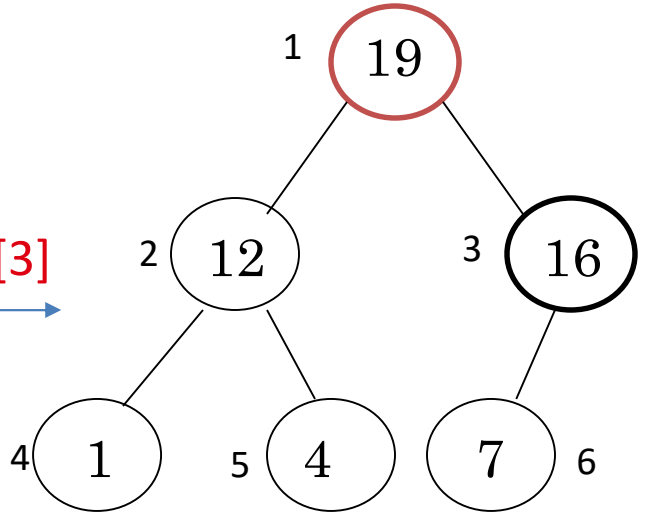


A[2] ↔ A[5]

i = 1: Call MAX-HEAPIFY(A,1,6)



A[1] ↔ A[3]

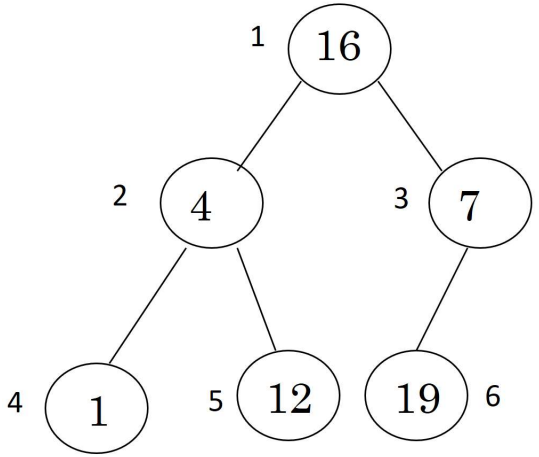


Max heap

Example 1: Sort array A in ascending order

A:

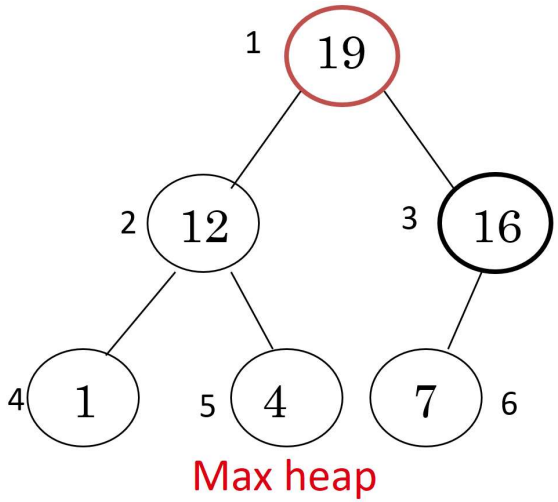
16	4	7	1	12	19
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]



- Step 1: Represent array as a complete binary tree

- Step 2: Call HEAPSORT(A):

  1. Call BUILD-MAX-HEAP(A): to make this tree become max-heap



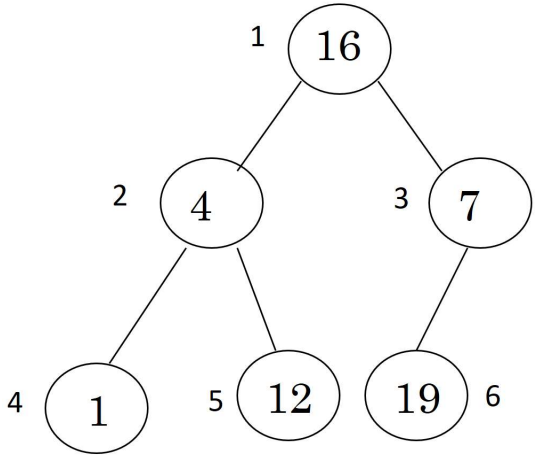
Alg: HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*)
2. n = length[A]
3. **for** i ← n **downto** 2 {
4. exchange *A*[1] ↔ *A*[i]
5. MAX-HEAPIFY(*A*, 1, i - 1)
6. }

Example 1: Sort array A in ascending order

A:

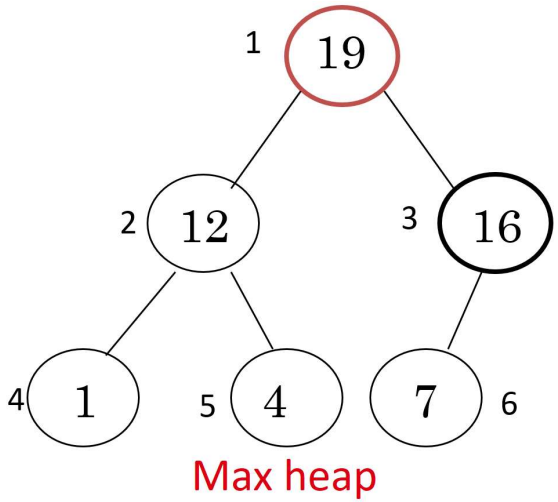
16	4	7	1	12	19
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]



- Step 1: Represent array as a complete binary tree

- Step 2: Call HEAPSORT(A):

1. Call BUILD-MAX-HEAP(A): to make this tree become max-heap



Max heap

2. Do lines 3-6

Alg: HEAPSORT(*A*)

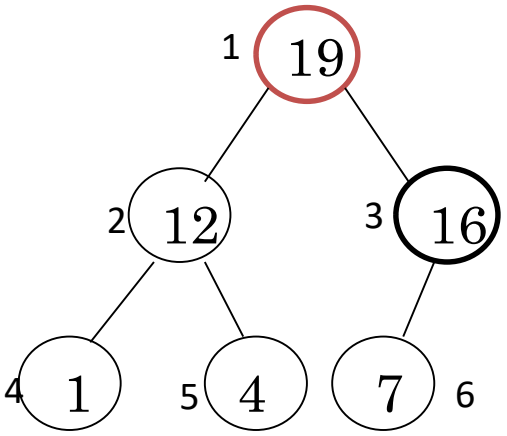
1. BUILD-MAX-HEAP(*A*)
2. n = length[A]
3. **for** i ← n **downto** 2 {
4. exchange *A*[1] ↔ *A*[i]
5. MAX-HEAPIFY(*A*, 1, i - 1)
6. }

Example 1:

Alg: HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*)
2. n = length[A]
3. **for** i ← n **downto** 2 {
4. exchange *A*[1] ↔ *A*[i]
5. MAX-HEAPIFY(*A*, 1, i - 1)
6. }

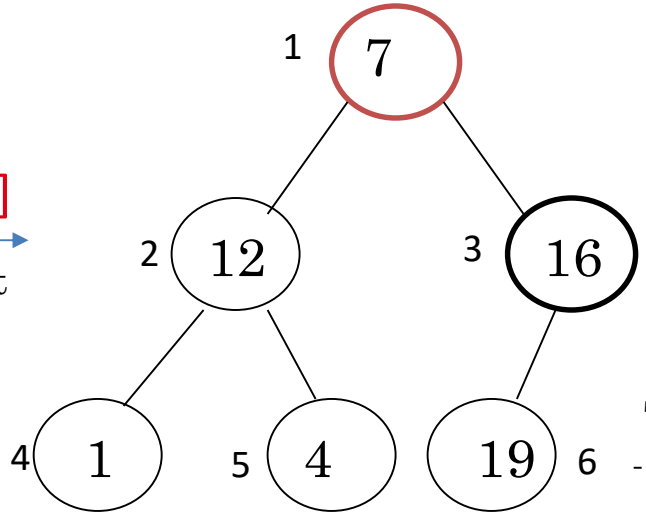
Max-heap:



i=6:

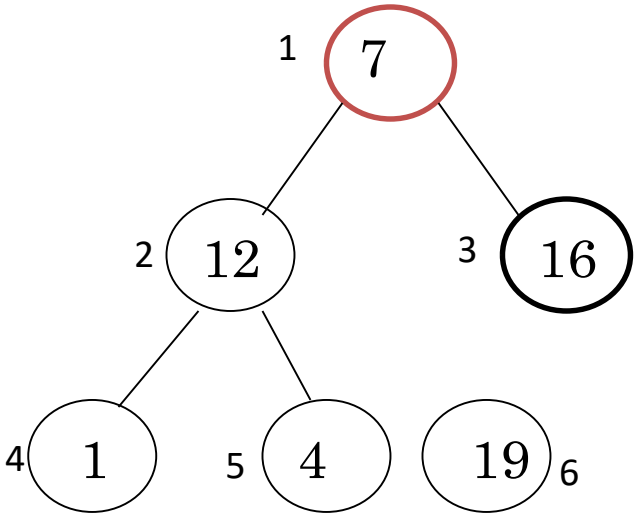
A[1] ↔ A[6]

Exchange last element and the root



Take out biggest ---------- 19

Call MAX-HEAPIFY(A,1,5)



Array A: Sorted:

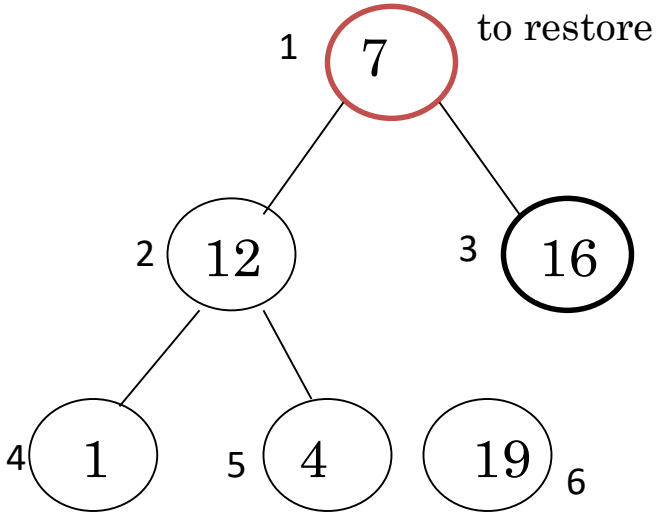
7	12	16	1	4	19

Example 1:

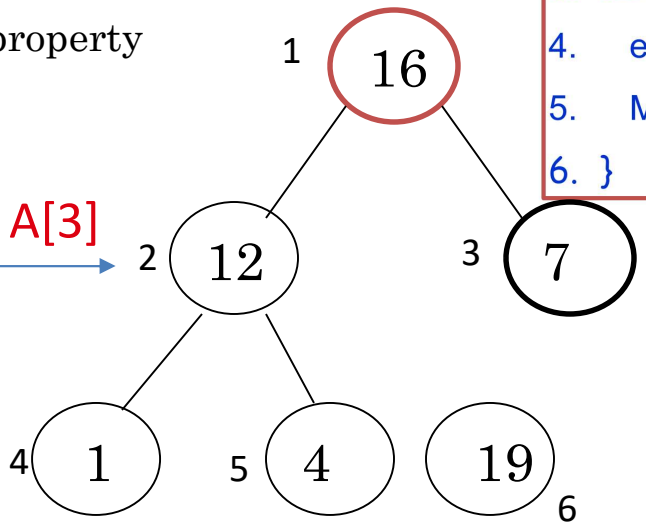
Alg: HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*)
2. n = length[A]
3. **for** i ← n **downto** 2 {
4. exchange *A*[1] ↔ *A*[i]
5. MAX-HEAPIFY(*A*, 1, i - 1)
6. }

Call MAX-HEAPIFY(A,1,5) to restore max-heap property



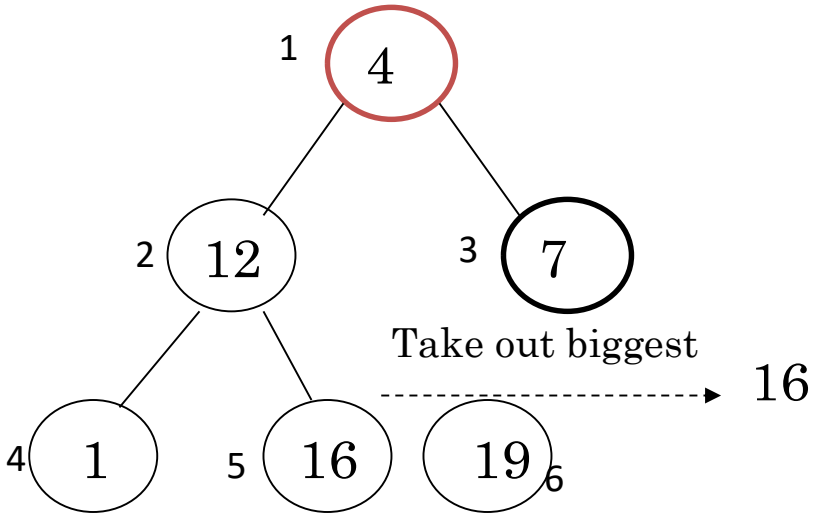
A[1] ↔ A[3]



i=5:
A[1] ↔ A[5]

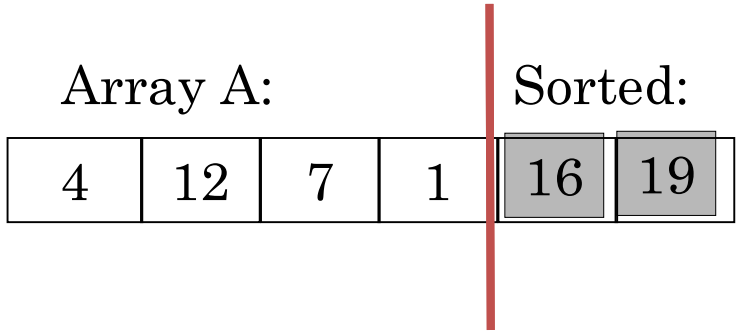
Exchange last element and the root

Call MAX-HEAPIFY(A,1,4)



Take out biggest

16

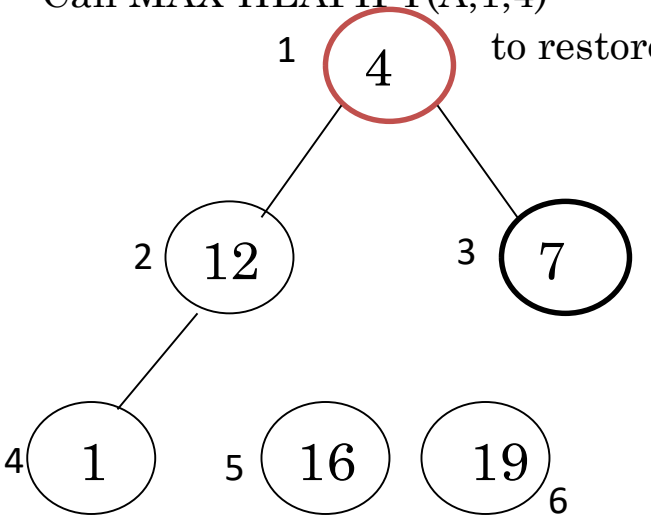


Example 1:

Alg: HEAPSORT(*A*)

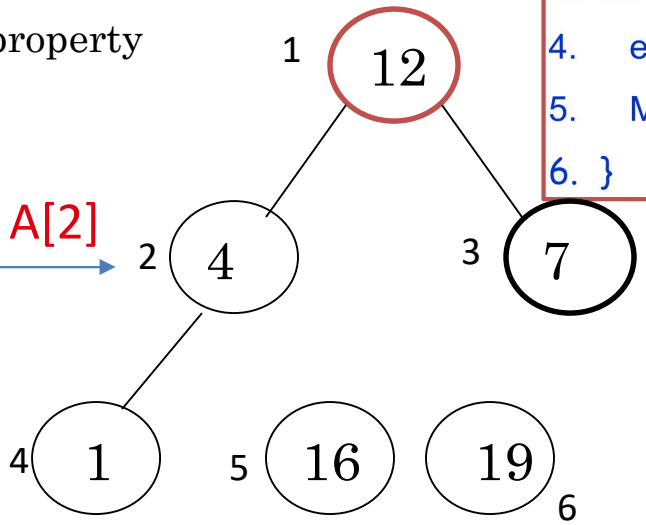
1. BUILD-MAX-HEAP(*A*)
2. n = length[A]
3. **for** i ← n **downto** 2 {
4. exchange *A*[1] ↔ *A*[i]
5. MAX-HEAPIFY(*A*, 1, i - 1)
6. }

Call MAX-HEAPIFY(A,1,4)



to restore max-heap property

A[1] ↔ A[2]

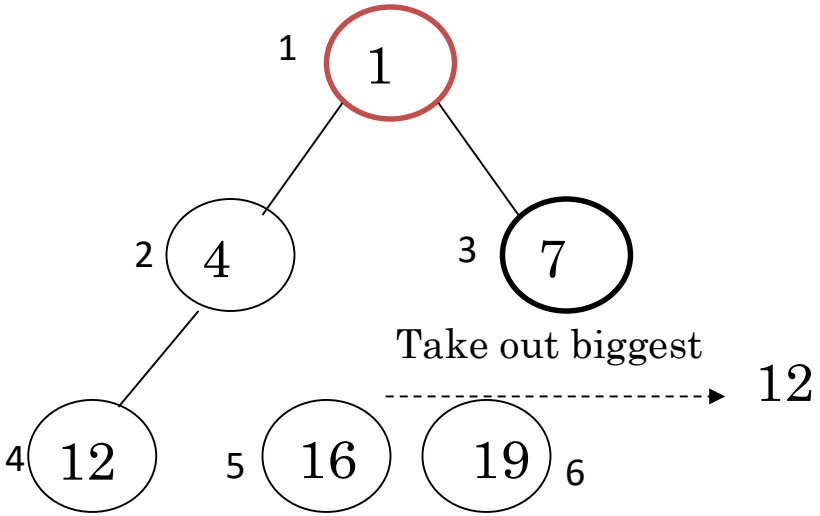


i=4:

A[1] ↔ A[4]

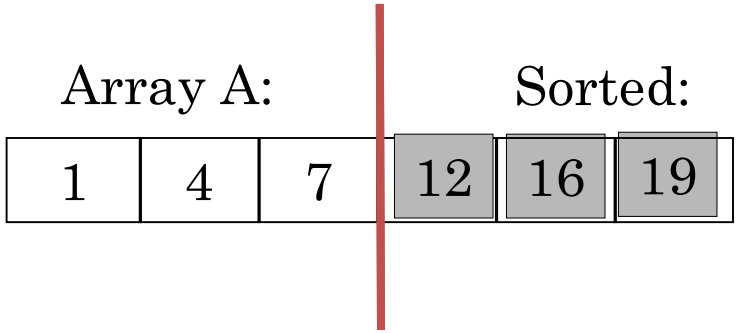
Exchange last element and the root

Call MAX-HEAPIFY(A,1,3)



Take out biggest

12

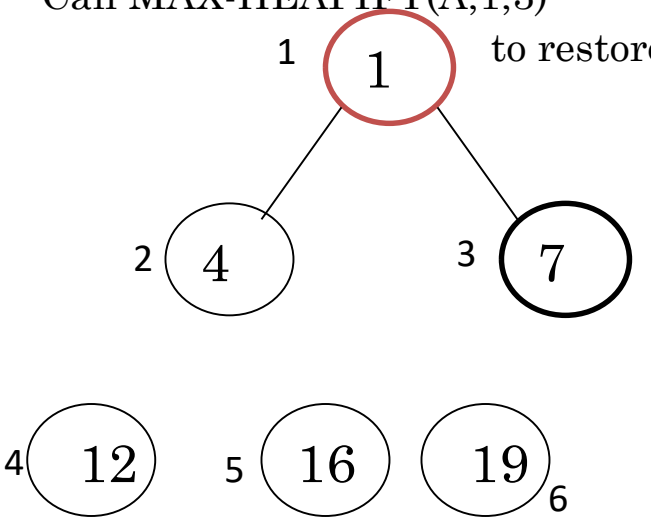


Example 1:

Alg: HEAPSORT(*A*)

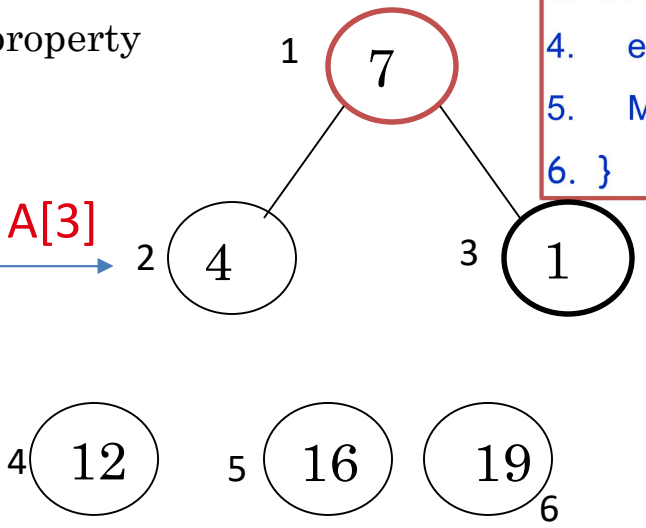
1. BUILD-MAX-HEAP(*A*)
2. n = length[A]
3. **for** i ← n **downto** 2 {
4. exchange *A*[1] ↔ *A*[i]
5. MAX-HEAPIFY(*A*, 1, i - 1)
6. }

Call MAX-HEAPIFY(A,1,3)



to restore max-heap property

A[1] ↔ A[3]

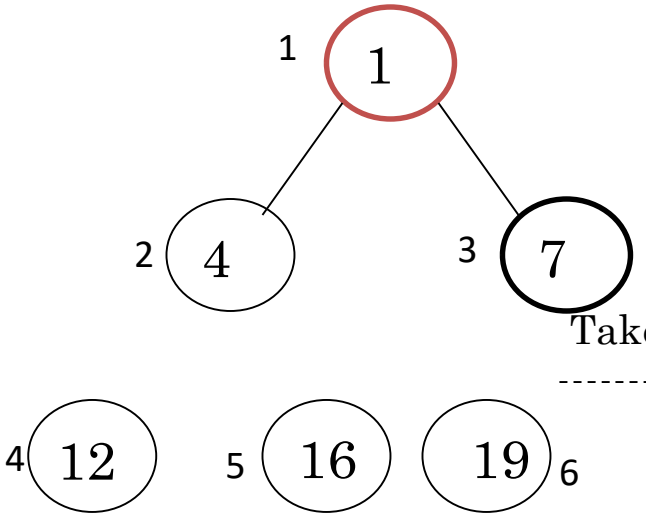


i=3:

A[1] ↔ A[3]

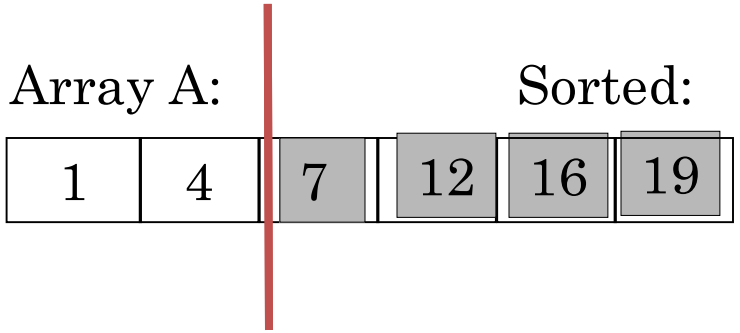
Exchange last element and the root

Call MAX-HEAPIFY(A,1,2)



Take out biggest

7

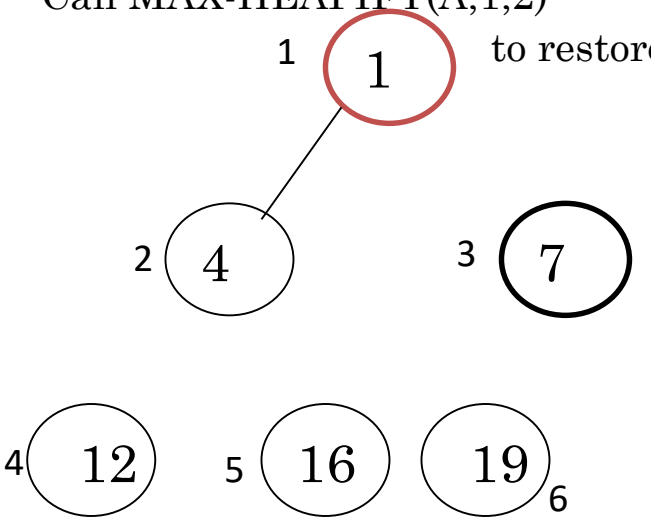


Example 1:

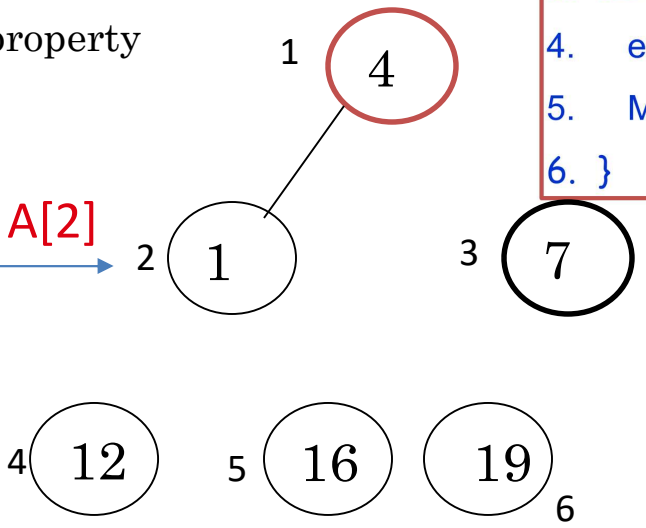
Alg: HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*)
2. n = length[A]
3. **for** i ← n **downto** 2 {
4. exchange *A*[1] ↔ *A*[i]
5. MAX-HEAPIFY(*A*, 1, i - 1)
6. }

Call MAX-HEAPIFY(A,1,2) to restore max-heap property



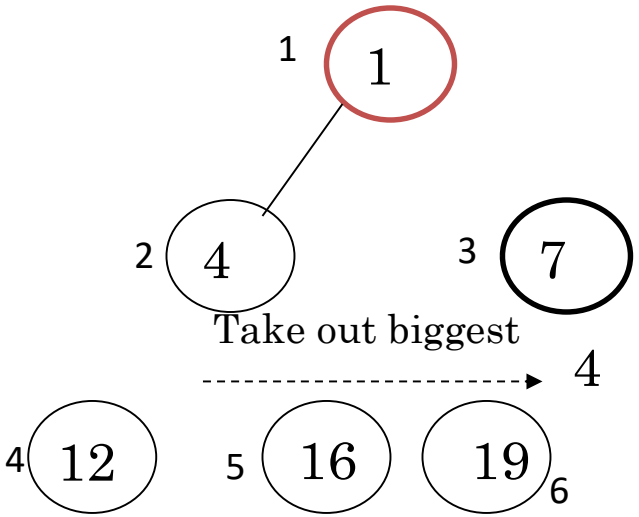
A[1] ↔ A[2]



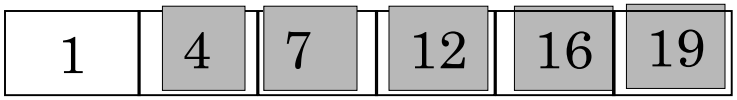
i=2:

A[1] ↔ A[2]

Exchange last element and the root



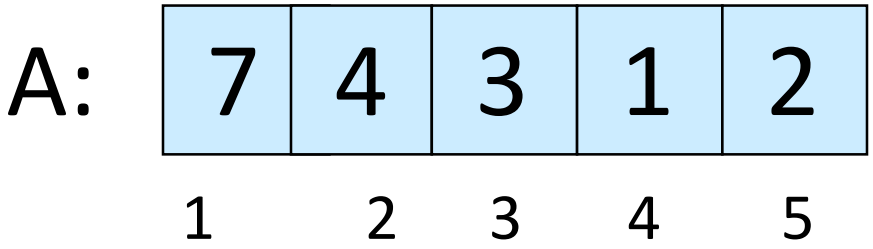
Array A: Sorted:



We get array A sorted in ascending order

Example 2: Use heap sort algorithm

- To sort the array A in ascending order

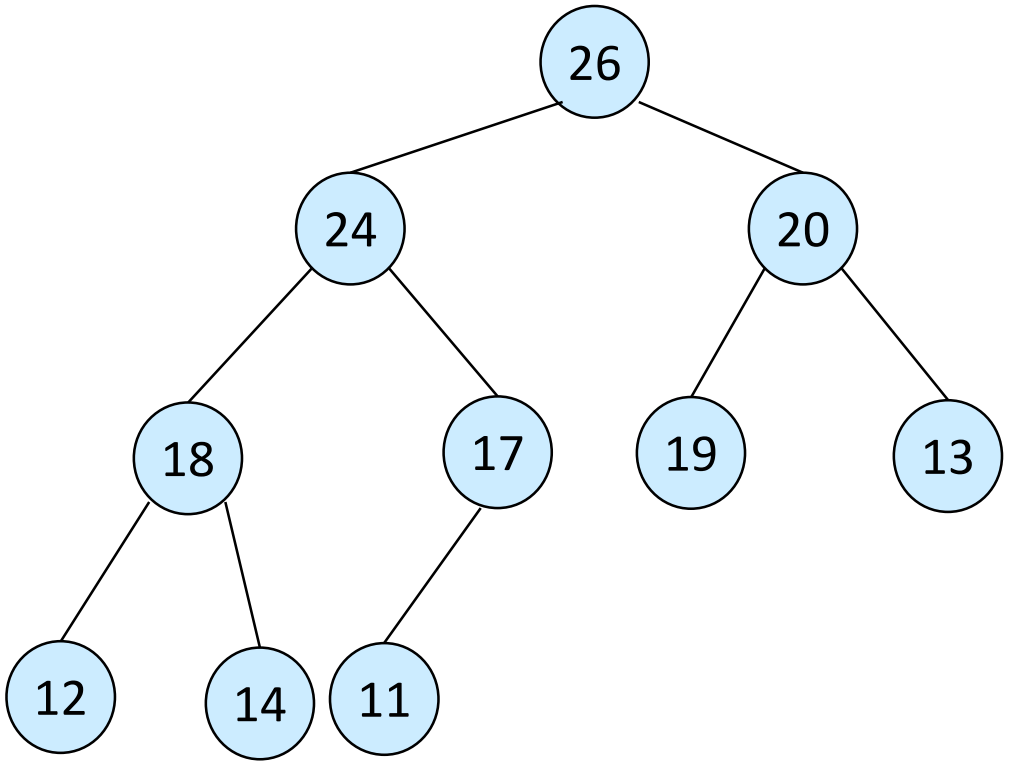


Example 3: Use heap sort algorithm

- To sort the array A in ascending order

A:

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10



Summary

5.1. Insertion Sort

5.2. Selection Sort

5.3. Bubble Sort

$O(n^2)$

5.4. Merge Sort

5.5. Quick Sort

Divide and conquer

$O(n\log_2 n)$

5.6. Heap Sort

$O(n\log_2 n)$