Lecture-07

Divide and Conquer Strategy

Sharad Hasan

Lecturer

Department of Computer Science and Engineering

Sheikh Hasina University, Netrokona.

D&C Introduction

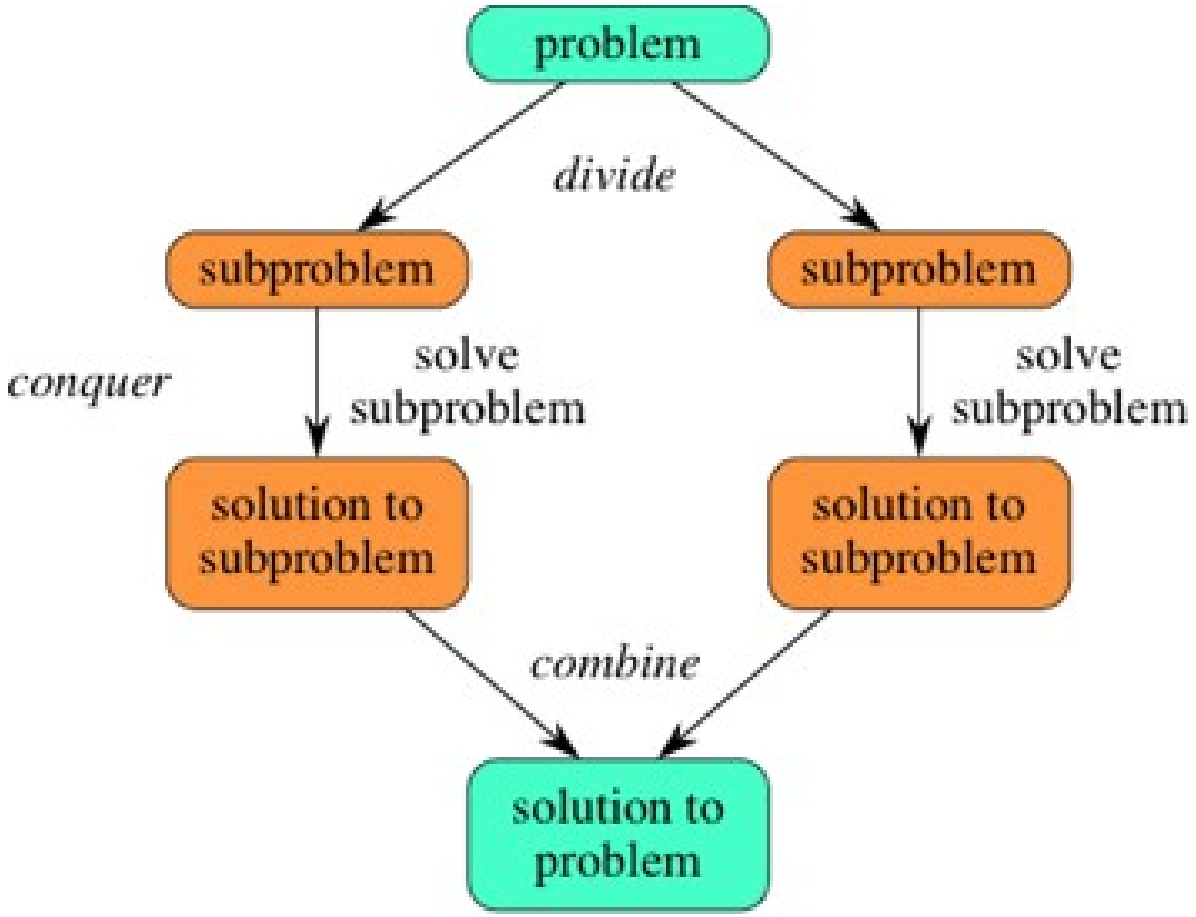
- This paradigm, ***divide-and-conquer***, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem.
- Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems.
- This method usually allows us to reduce the time complexity by a large extent. For example, Bubble Sort uses a complexity of $O(n^2)$, whereas quicksort (an application Of Divide And Conquer) reduces the time complexity to $O(n \log(n))$. Linear Search has time complexity $O(n)$, whereas Binary Search (an application Of Divide And Conquer) reduces time complexity to $O(\log(n))$.

Steps of D&C

You should think of a divide-and-conquer algorithm as having three parts:

- ***Divide*** the problem into a number of subproblems that are smaller instances of the same problem.
- ***Conquer*** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
- ***Combine*** the solutions to the subproblems into the solution for the original problem.

You can easily remember the steps of a divide-and-conquer algorithm as divide, conquer, combine. Here's how to view one step, assuming that each divide step creates two subproblems (though some divide-and-conquer algorithms create more than two):



If we expand out two more recursive steps, it looks like this:

Because divide-and-conquer creates at least two subproblems, a divide-and-conquer algorithm makes multiple recursive calls.

Divide And Conquer algorithm

```
DAC(a, i, j)
{
    if(small(a, i, j))
        return(Solution(a, i, j))
    else
        mid = divide(a, i, j)          // f1(n)
        b = DAC(a, i, mid)             // T(n/2)
        c = DAC(a, mid+1, j)           // T(n/2)
        d = combine(b, c)              // f2(n)
    return(d)
}
```

Recurrence Relation for DAC algorithm :

$$T(n)=\begin{cases} O(1), if\ n\ is\ small \\ f1(n)+2T(n/2)+f2(n) \end{cases}$$

Time Complexity

The complexity of the divide and conquer algorithm is calculated using the master theorem.

$$T(n) = aT(n/b) + f(n),$$

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions.

Application of D&C

The following are some standard algorithms that follow Divide and Conquer algorithm.

1. Binary Search is a searching algorithm. In each step, the algorithm compares the input element (x) with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs to the left side of the middle element, else it recurs to the right side of the middle element.

2. Quicksort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

Application of D&C

3. Merge Sort is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves. The time complexity of this algorithm is ***O(n log (n))***, be it best case, average case or worst case. It's time complexity can be easily understood from the recurrence equates to:

$$T(n) = 2T(n/2) + n2.$$

4. Closest Pair of Points The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in $O(n^2)$ time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in $O(n \log(n))$ time.

Application of D&C

5. Finding k-th largest elements in an array: Given an array arr[ ] of size N, the task is to printing K largest elements in an array. Elements in output array can be in any order. he Kth largest element can be found using binary search by defining a search range based on the minimum and maximum values in the input array. In each iteration of binary search, count the larger than the midpoint and update the search range accordingly. This process continues until the range collapses to a single element, which is the kth largest element.

6. Strassen's Algorithm is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3). Strassen's algorithm multiplies two matrices in O(n^2.8974) time.

Application of D&C

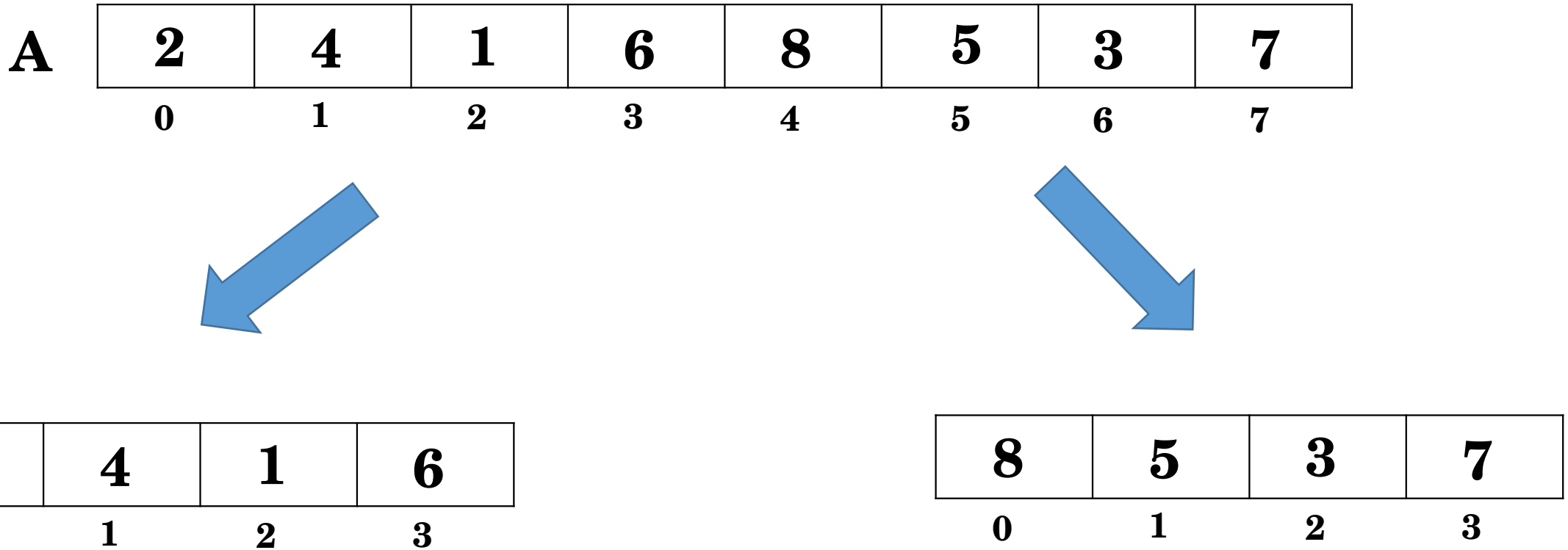
7. Cooley–Tukey Fast Fourier Transform (FFT) algorithm is the most common algorithm for FFT. It is a divide and conquer algorithm which works in ***O(n log(n))*** time.

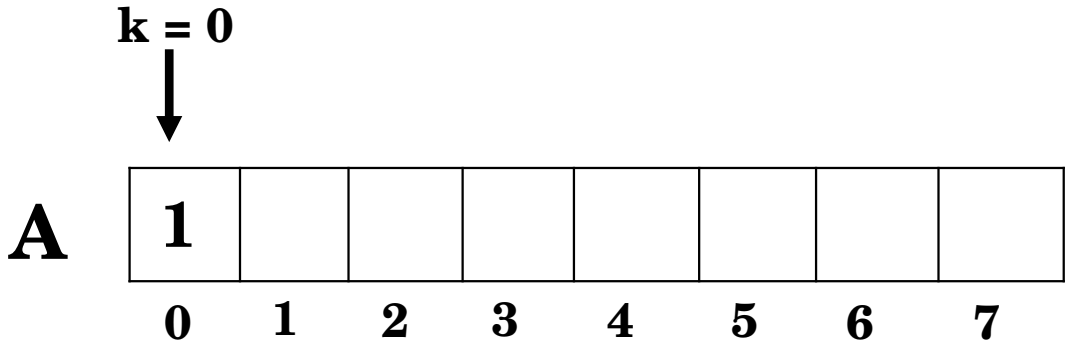
8. The Karatsuba algorithm was the first multiplication algorithm asymptotically faster than the quadratic "grade school" algorithm. It reduces the multiplication of two n-digit numbers to at most to $n^{1.585}$ (which is approximation of log of 3 in base 2) single digit products. It is therefore faster than the classical algorithm, which requires n^2 single-digit products.

What does not qualifies as Divide and Conquer

Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in the array. If the values match, return the index of the middle. Otherwise, if x is less than the middle element, then the algorithm recurs for the left side of the middle element, else recurs for the right side of the middle element. Contrary to popular belief, this is not an example of Divide and Conquer because there is only one sub-problem in each step (Divide and conquer requires that there must be two or more sub-problems) and hence this is a case of ***Decrease and Conquer.***

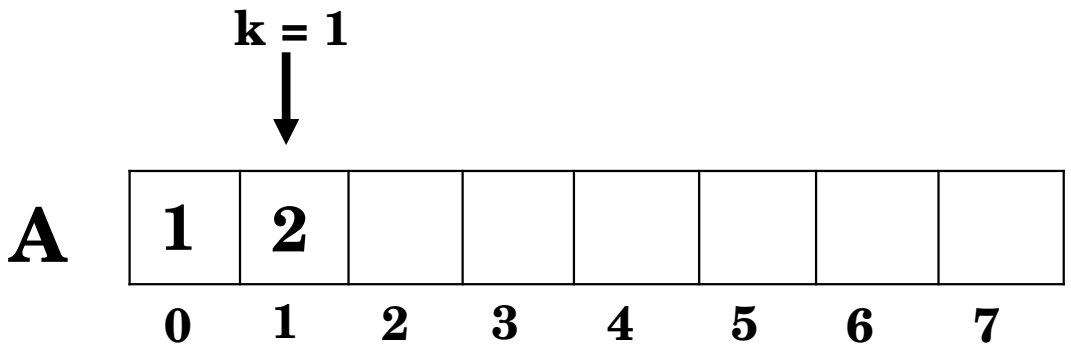
Merge Sort


A
2 4 1 6 8 5 3 7
0 1 2 3 4 5 6 7
2 4 1 6
0 1 2 3
8 5 3 7
0 1 2 3



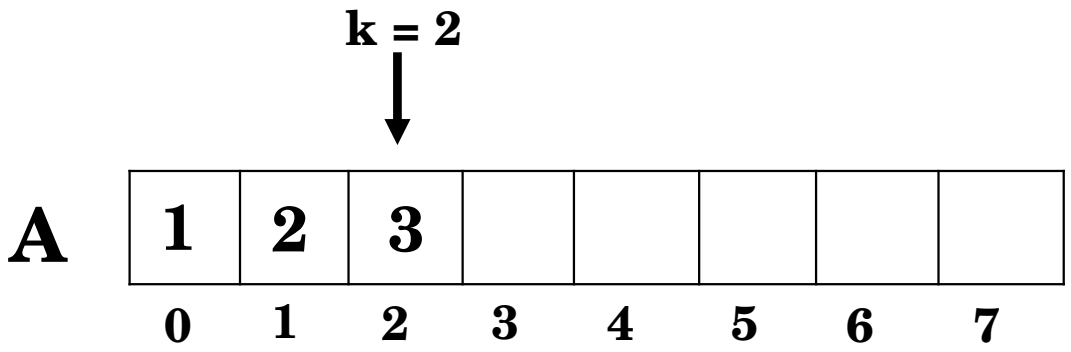

```
Merge (L, R, A){
     nL = length(L)
     nR = length(R)
     i = 0;
     j = 0;
     k = 0;
     while(i<nL && j<nR){
          if(L[i]<= R[j]){
               A[k]= L[i];
               k = k+1;
               i = i+1;
          }
          else{
               A[k]= R[j];
               k = k+1;
               j = j+1;
          }
     }
}
```

Merging 2 sorted arrays



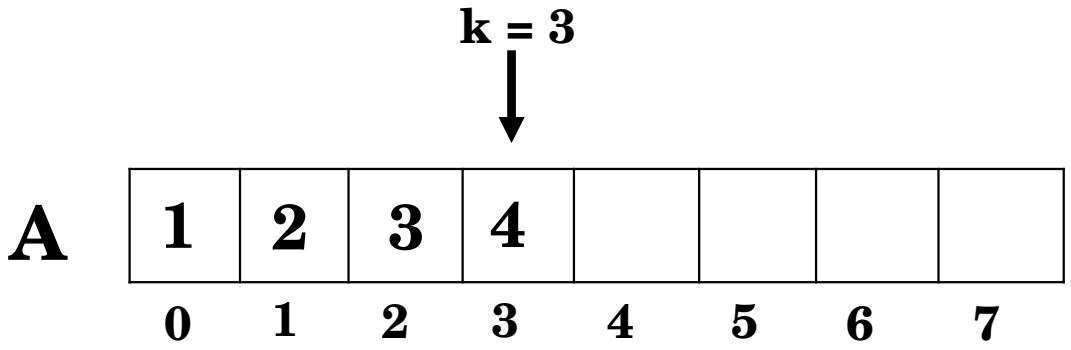

```
Merge (L, R, A){
    nL = length(L)
    nR = length(R)
    i = 0;
    j = 0;
    k = 0;
    while(i<nL && j<nR){
        if(L[i]<= R[j]){
            A[k]= L[i];
            k = k+1;
            i = i+1;
        }
        else{
            A[k]= R[j];
            k = k+1;
            j = j+1;
        }
    }
}
```

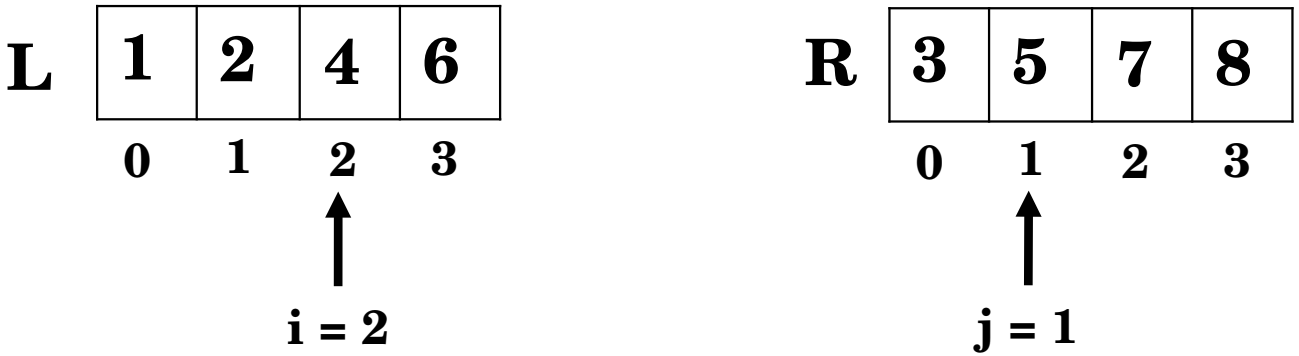
Merging 2 sorted arrays




```
Merge (L, R, A){
    nL = length(L)
    nR = length(R)
    i = 0;
    j = 0;
    k = 0;
    while(i<nL && j<nR){
        if(L[i]<= R[j]){
            A[k]= L[i];
            k = k+1;
            i = i+1;
        }
        else{
            A[k]= R[j];
            k = k+1;
            j = j+1;
        }
    }
}
```

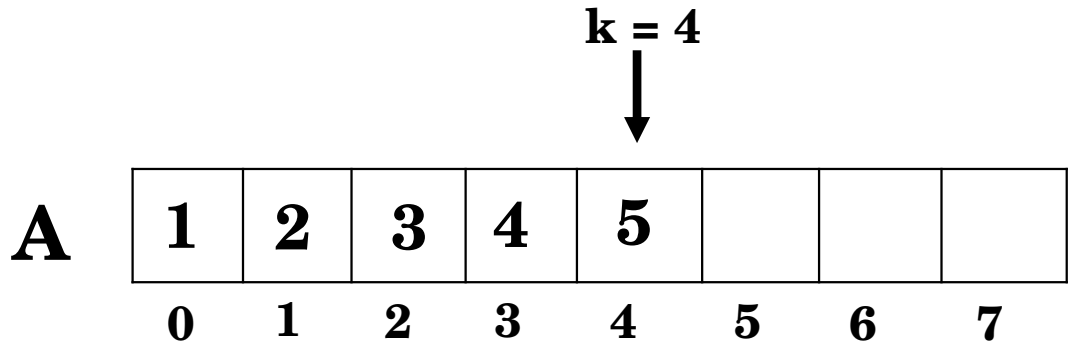
Merging 2 sorted arrays

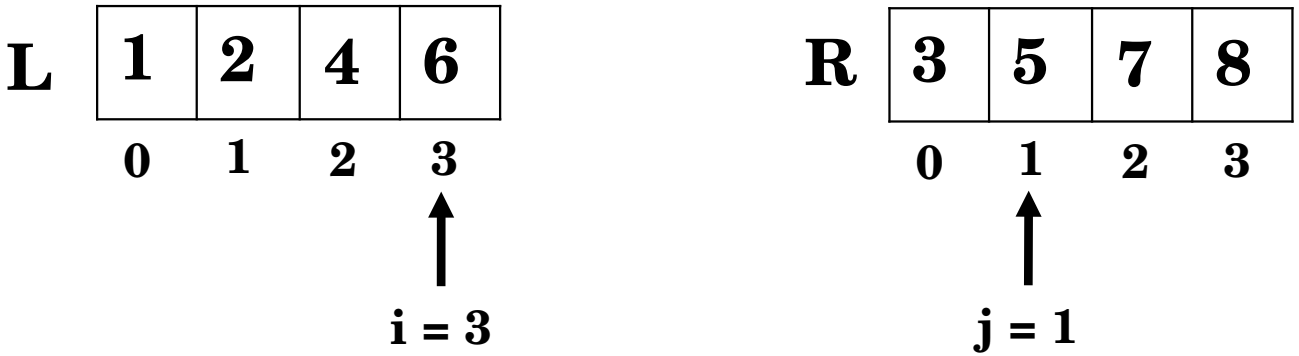





```
Merge (L, R, A){
    nL = length(L)
    nR = length(R)
    i = 0;
    j = 0;
    k = 0;
    while(i<nL && j<nR){
        if(L[i]<= R[j]){
            A[k]= L[i];
            k = k+1;
            i = i+1;
        }
        else{
            A[k]= R[j];
            k = k+1;
            j = j+1;
        }
    }
}
```

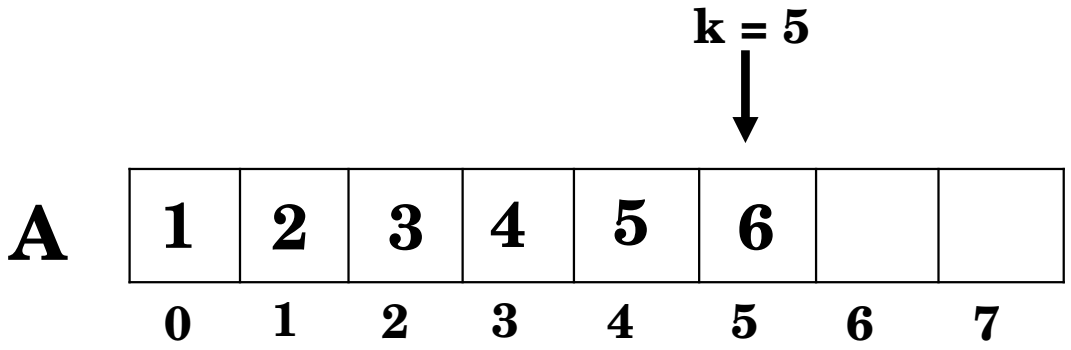
Merging 2 sorted arrays

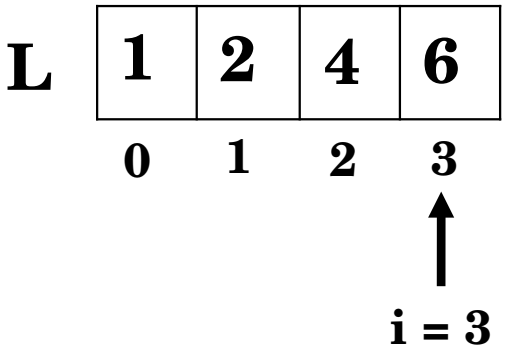


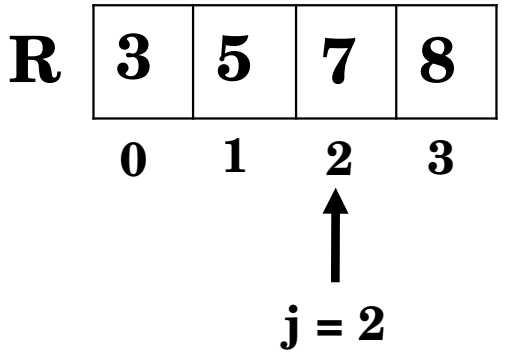



```
Merge (L, R, A){
     nL = length(L)
     nR = length(R)
     i = 0;
     j = 0;
     k = 0;
     while(i<nL && j<nR){
          if(L[i]<= R[j]){
               A[k]= L[i];
               k = k+1;
               i = i+1;
          }
          else{
               A[k]= R[j];
               k = k+1;
               j = j+1;
          }
     }
}
```

Merging 2 sorted arrays

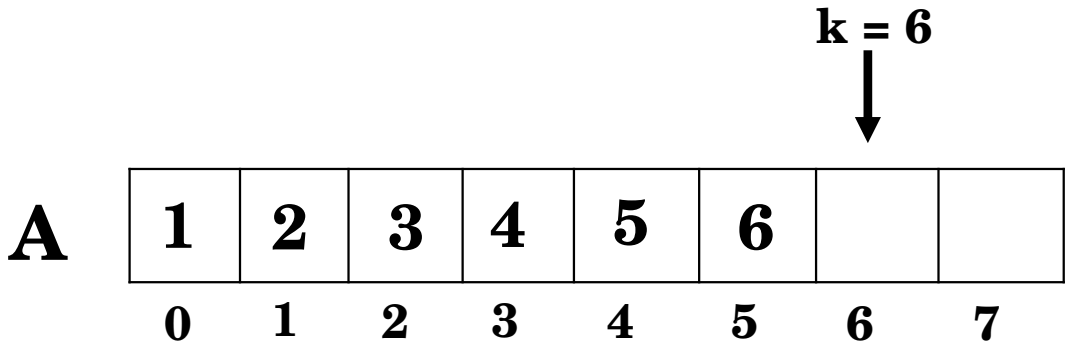


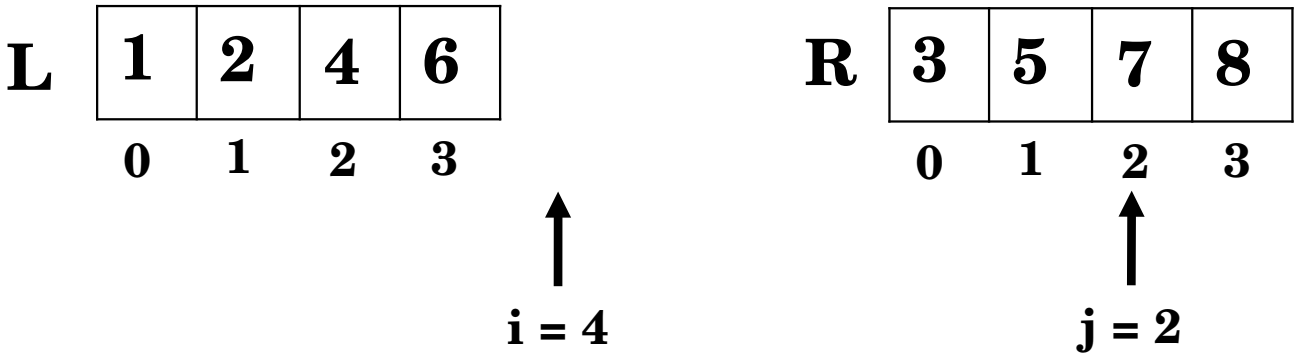





```
Merge (L, R, A){
     nL = length(L)
     nR = length(R)
     i = 0;
     j = 0;
     k = 0;
     while(i<nL && j<nR){
          if(L[i]<= R[j]){
               A[k]= L[i];
               k = k+1;
               i = i+1;
          }
          else{
               A[k]= R[j];
               k = k+1;
               j = j+1;
          }
     }
}
```

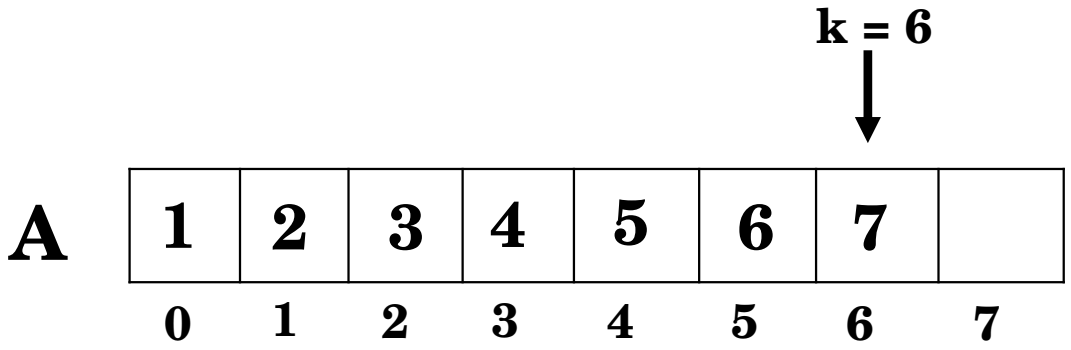
Merging 2 sorted arrays

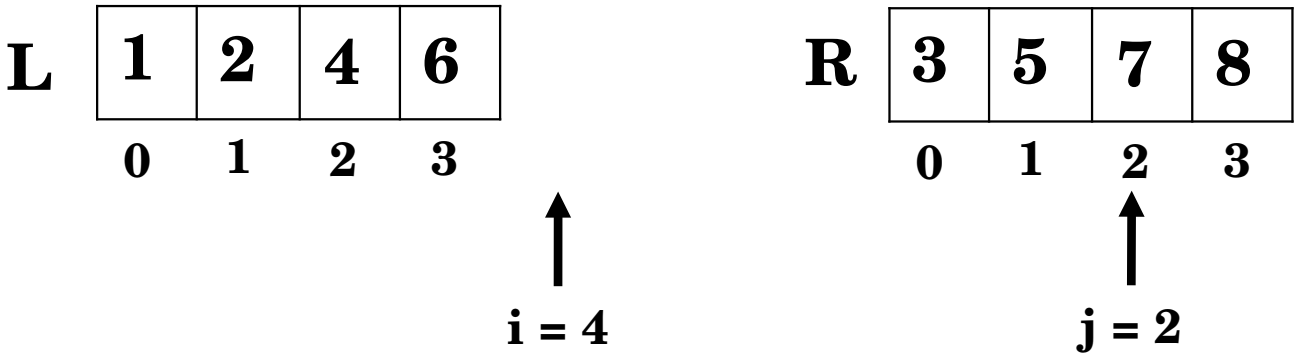





```
Merge (L, R, A){
     nL = length(L)
     nR = length(R)
     i = 0;
     j = 0;
     k = 0;
     while(i<nL && j<nR){
          if(L[i]<= R[j]){
               A[k]= L[i];
               k = k+1;
               i = i+1;
          }
          else{
               A[k]= R[j];
               k = k+1;
               j = j+1;
          }
     }
}
```

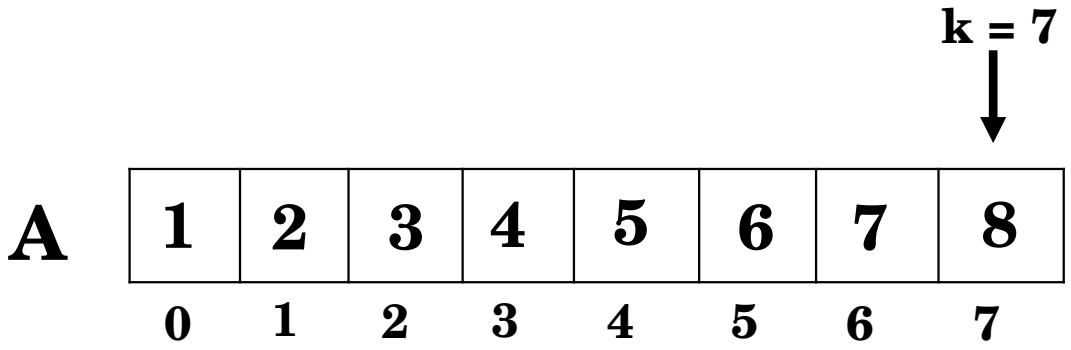
Merging 2 sorted arrays

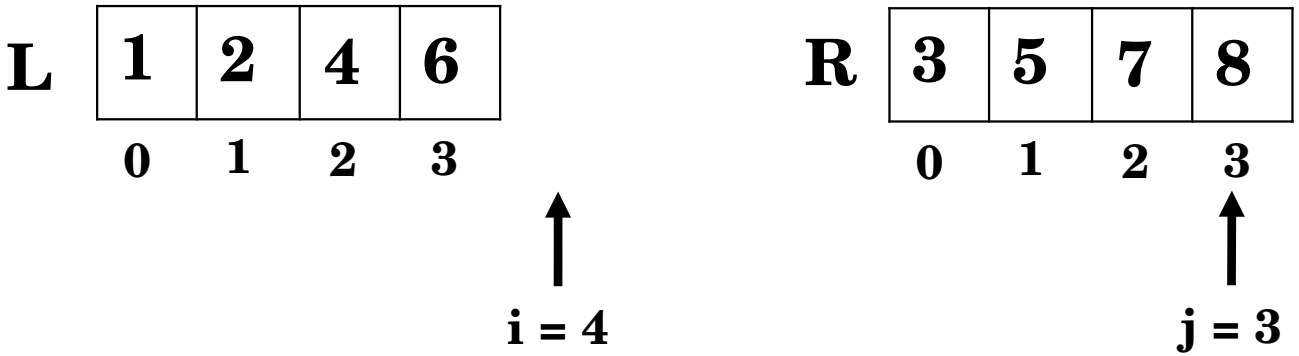





```
Merge (L, R, A){
     nL = length(L)
     nR = length(R)
     i = 0;
     j = 0;
     k = 0;
     while(i<nL && j<nR){
          if(L[i]<= R[j]){
               A[k]= L[i];
               k = k+1;
               i = i+1;
          }
          else{
               A[k]= R[j];
               k = k+1;
               j = j+1;
          }
     }
     while (i<nL){
          A[k]= L[i];
          k = k+1;
          i = i+1;
     }
     while (j<nR){
          A[k]= R[j];
          k = k+1;
          j = j+1;
     }
}
```

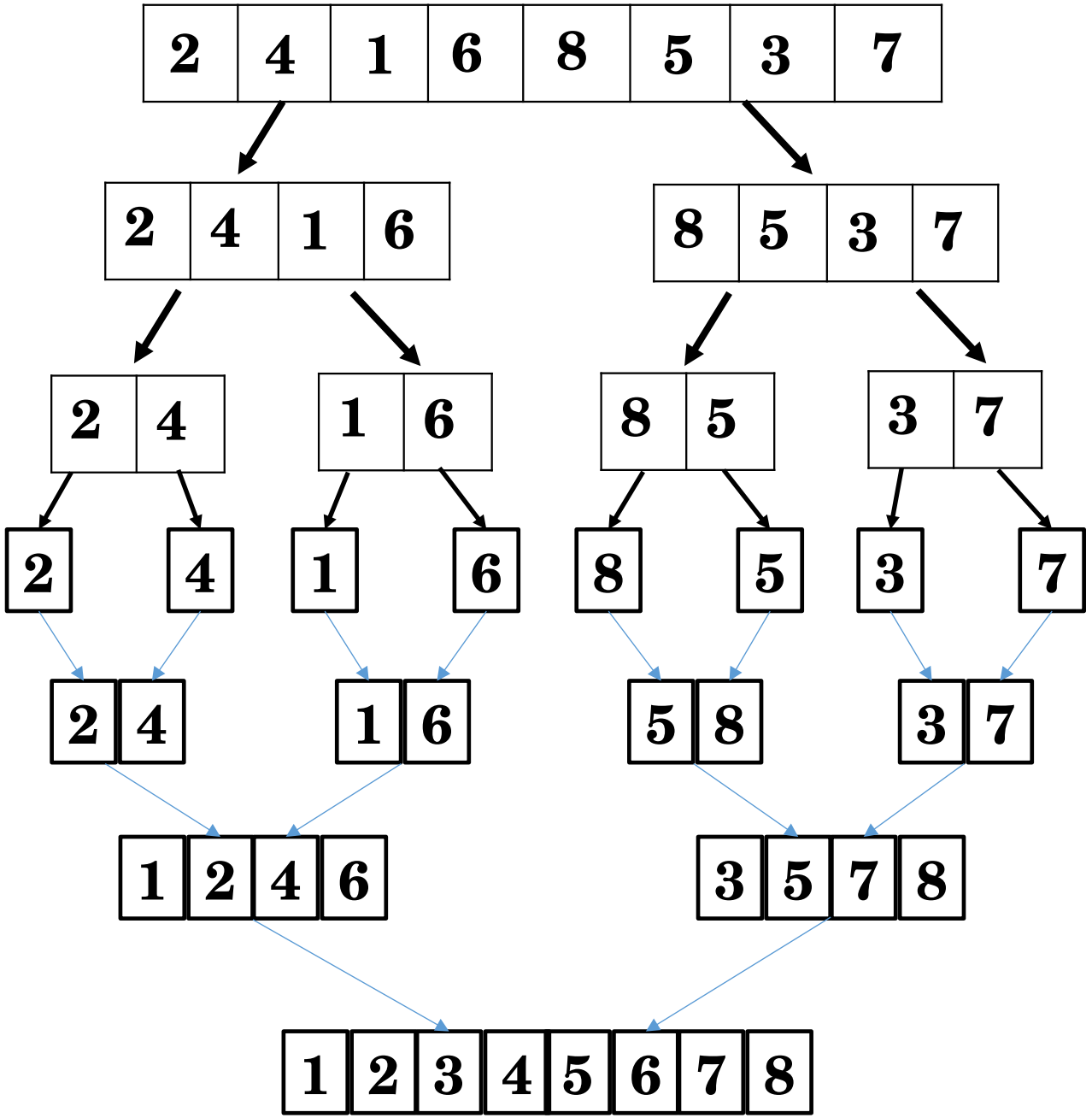
Merging 2 sorted arrays





Merging 2 sorted arrays

```
Merge (L, R, A){
    nL = length(L)
    nR = length(R)
    i = 0;
    j = 0;
    k = 0;
    while(i<nL && j<nR){
        if(L[i]<= R[j]){
            A[k]= L[i];
            k = k+1;
            i = i+1;
        }
        else{
            A[k]= R[j];
            k = k+1;
            j = j+1;
        }
    }
    while (i<nL){
        A[k]= L[i];
        k = k+1;
        i = i+1;
    }
    while (j<nR){
        A[k]= R[j];
        k = k+1;
        j = j+1;
    }
}
```




```
MergeSort (A){
    n = length(A)
    if(n<2)
        return
    mid = n/2
    left = array of size (mid)
    right = array if size (n-mid)
    for i=0 to (mid-1)
        left[i]= A[i]
     for i=mid to (n-1)
        right[i-mid]= A[i]
    MergeSort(left)
    MergeSort(right)
    Merge(left, right, A)
}
```

Finding k-th Smallest Element from an Array

Let's understand the problem

Given an array A [ ] and a positive integer k, write a program to find the kth smallest element in the array.

- ❖It is given that all array elements are distinct.
- ❖We can assume that k is always valid, $1 \leq k \leq n$.

Examples:

Input: X [ ] = [4, 3, 13, 2, 12, 7, 23], k = 4

Output: 7, i.e., 7 is the 4th smallest element in the array.

Input: A [ ] = [-12, -8, 16, 23], k = 2

Output: -8, i.e., -8 is the 2nd smallest element in the array.

Brute force approach using sorting

Solution idea

As we know, all elements in the array are distinct. So one basic idea would be to sort the array in increasing order and directly return the kth number from the start, i.e., return A [k - 1].

Solution pseudocode

```
int KthSmallestElement(int A [ ], int n, int k) {
      sort (A, n)
      return A[k - 1]
}
```

Solution analysis

Suppose we are using heap sort, which is an efficient O(n log n) sorting algorithm. So time complexity is equal to the time complexity of the heap sort + the time complexity of accessing the kth smallest element, which is O(n log n) + 0(1) = O(n log n).

The space complexity is 0(1) because heap sort is an in-place sorting algorithm.

Quick-select approach (Divide and conquer idea similar to quick-sort)

Solution idea

Now, we will discuss an interesting quick-select approach that solves the problem efficiently by using a divide-and- conquer idea similar to the quick-sort algorithm.

The solution intuition comes from the quick-sort partition process: dividing the array into two parts around a pivot and returning the sorted array's pivot index. Elements in the array will look like this after the partition: ***A [l...pos-1] < pivot < A [pos+1...r].*** Here the pivot element is present at index ***pos***, and ***(pos – 1)*** elements are smaller than the pivot.

So the pivot element is the ***(pos - l + 1)th*** smallest element in the array.

Quick-select approach (Divide and conquer idea similar to quick-sort)

- if ***((pos – l + 1) == k)***: the pivot is the k-th smallest, and we return A[pos].
- if ***((pos - l + 1) > k)***: the k-th smallest must be present in the left subarray.
- if ***((pos - l + 1) < k)***: the k-th smallest must be present on the right subarray.

So from the above insight, we can develop an approach to use the partition process and find the kth smallest element recursively. But unlike the quick-sort, which processes both subarrays recursively, we process only one subarray. We recur for either the left or right side based on comparing k and the pivot position.

Quick-select approach (Divide and conquer idea similar to quick-sort)

Solution Steps:

1. We define a function ***kth_smallest_element (int A [ ], int l, int r, int k)***, where the left and right ends of the array are provided in the input. Initially, ***l = 0*** and ***r = n - 1***.
2. The base case is the scenario of a single-element array, i.e., ***if l == r, return A [l]***.
3. Now we partition the array ***A [l ... r]*** into two subarrays ***(A [l ... pos-l] and A [pos + 1 ... r])*** using the partition process. Here pos is the index of the pivot returned by the partition.
4. After the partition, we calculate the pivot order in the array, i.e., ***count = pos - l + 1***. In other words, the pivot is the ***(pos -l + l)th*** smallest element.

Quick-select approach (Divide and conquer idea similar to quick-sort)

Solution Steps:

5. If ***count == k***, we return ***A[pos]***. Otherwise, we determine in which of the two subarrays ***(A [l….. pos-1]) and A [pos+1……r])*** the kth smallest is present.
6. If ***count > k***, the desired element must lie in the left subarray. We call the same function recursively for the left part, i.e ***kth_smallest_element(A, l, pos-l, k)***.
7. If ***count < k***, the desired element must lie in the right subarray. We call the same function recursively for the right part, i.e., ***kth_smallest_element(A, pos+1, r, k-count)***.

Note: The desired element is the ***(k - count)th*** smallest element of the right side because we already know ***count*** number of values is smaller than the ***k-th*** smallest element (left subarray and a pivot).

Example

A	**1**	**17**	**7**	**5**	**3**	**2**	**26**	**19**	**12**
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

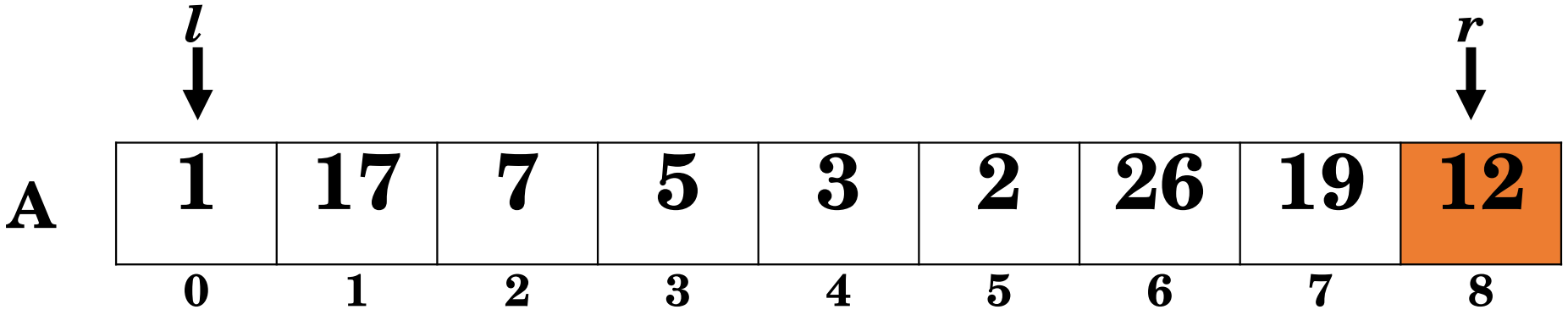
Find the 4^{th} smallest element. That means, ***k = 4***

Example

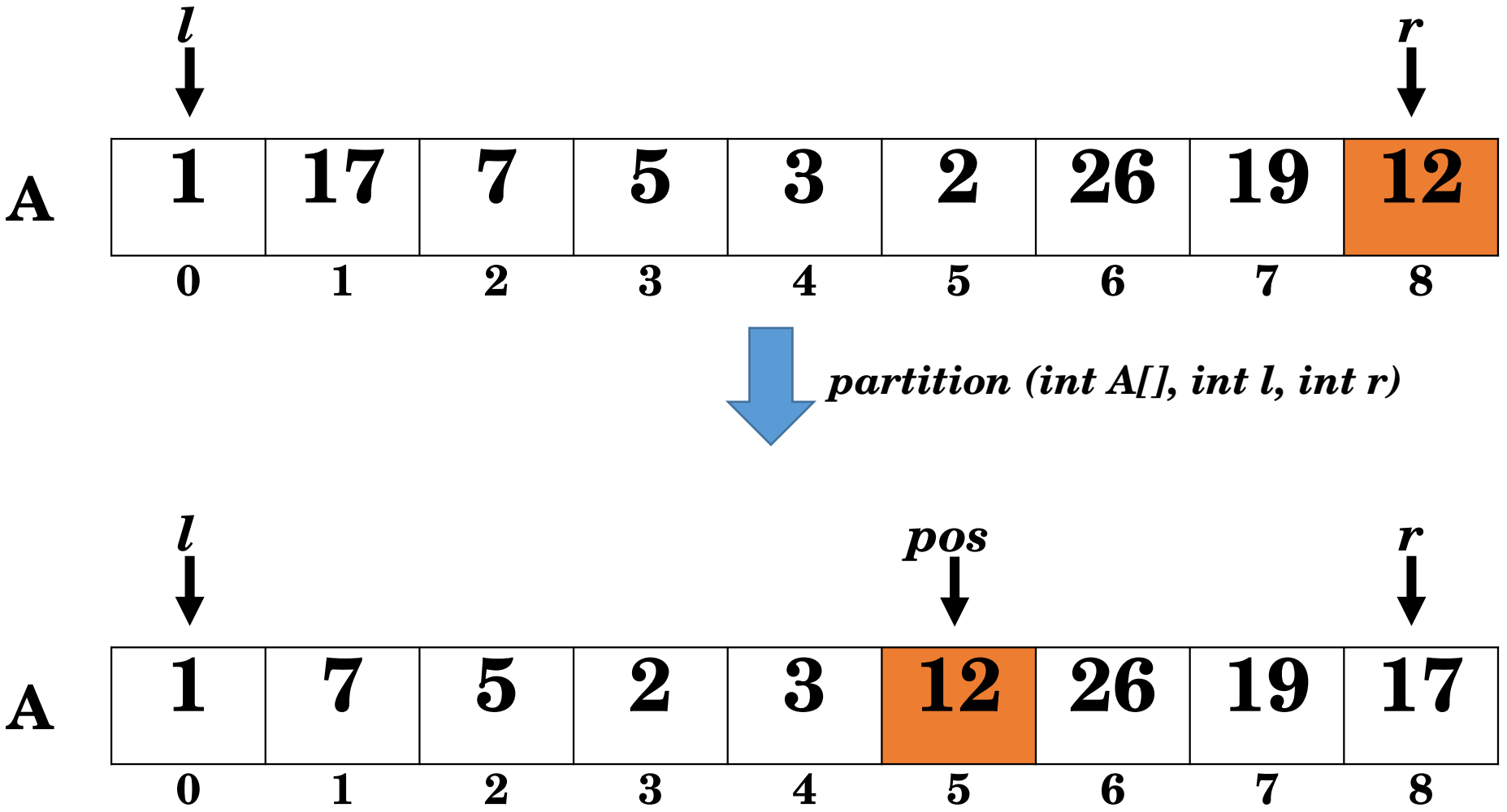
1. Select any element of the array as ***pivot.*** Generally, the last element is selected as ***pivot.***

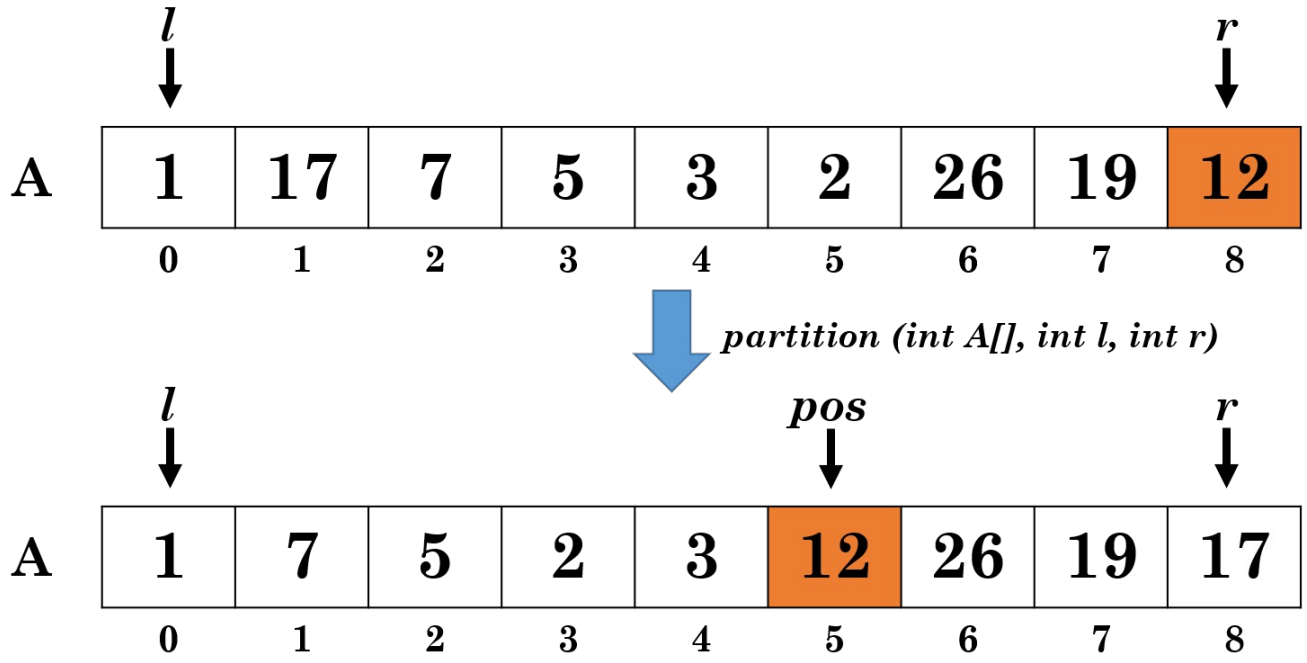
Initially, ***l = 0*** and ***r = 8***.

Therefore, ***pivot = A [r] = 12***



2. Then, we use ***partition ( )*** function to rearrange the array ***A*** and place ***pivot*** at such an index that, all the elements to the right of ***pivot*** are less than the ***pivot*** and all the elements to the right of the ***pivot*** are greater than the ***pivot***.





The ***partiition ( )*** function returns the current index of the ***pivot*** element, which we are calling ***pos***. in this case,

pos = 5

Remember, the ***pivot*** element is the ***(pos – l + 1)th*** smallest element. We call it ***count***.

Therefor, ***count = (pos – l + 1) = (5-0+1) = 6***

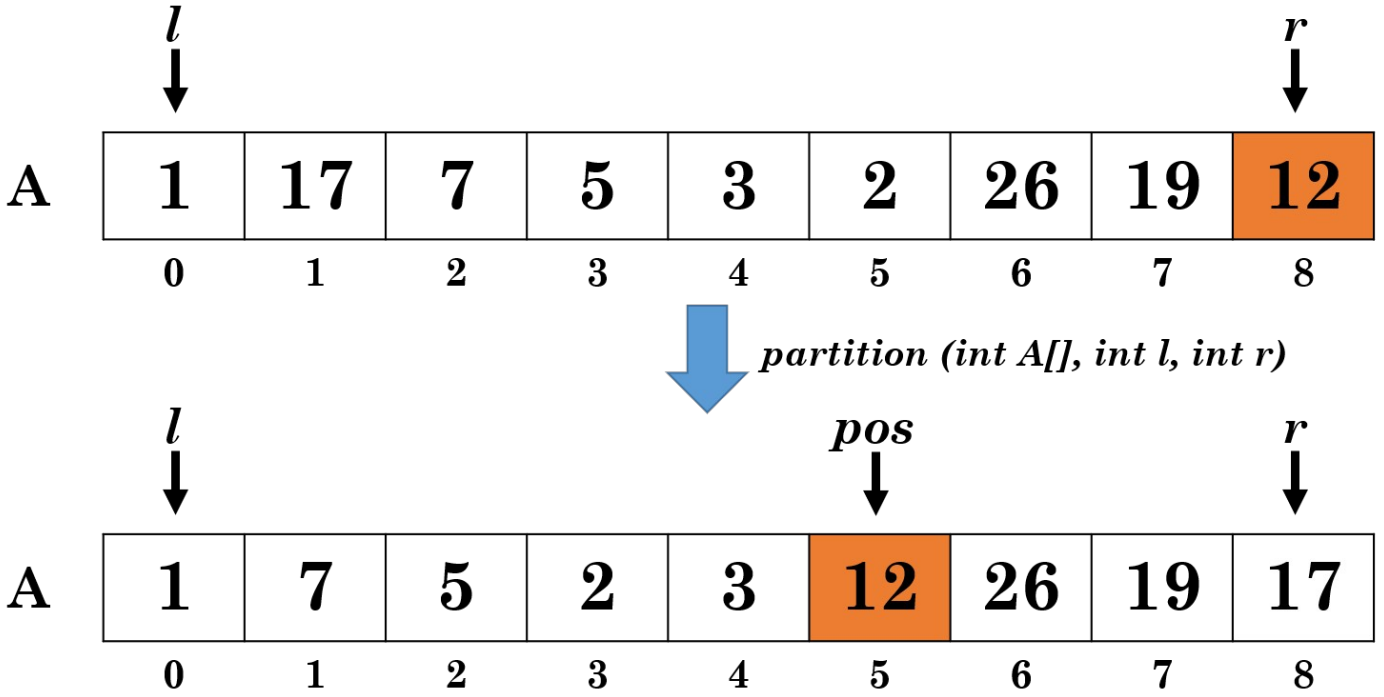
∴ ***pivot*** is the 6^{th} smallest element in this array.

Three cases can happen.

Case 1:

if ***(count == k)***

return ***A [pos]***



Case 2:

if ***(count > k)***

return ***kth_smallest_element(A, l, pos-1, k)***

Case 3:

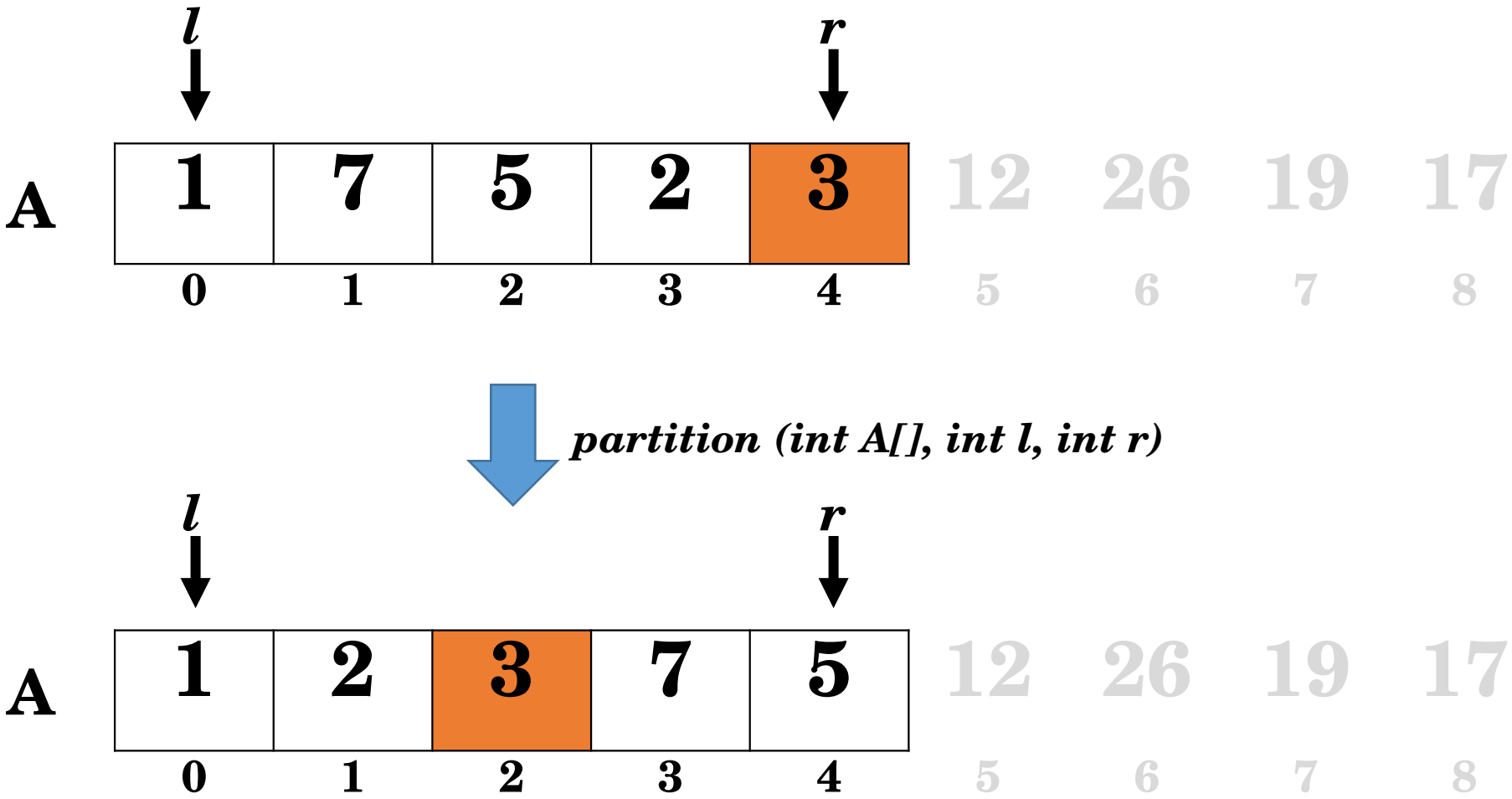
if ***(count < k)***

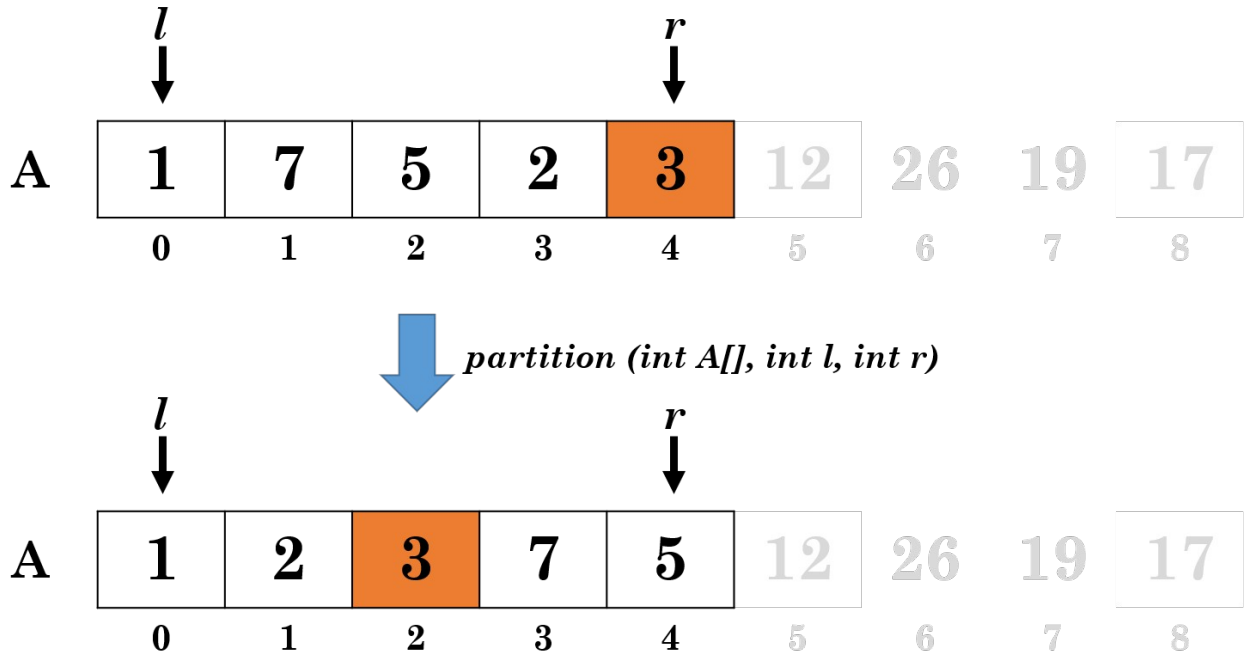
return ***kth_smallest_element(A, pos+1, r, k-count)***

3. Case 2 happens in our example. So we recursively call the function

kth_smallest_element(A, l, pos-1, k)

or, kth_smallest_element(A, 0, 4, 2)





The ***partiition ( )*** function returns the current index of the ***pivot*** element, which we are calling ***pos***. in this case,

pos = 2

Remember, the ***pivot*** element is the ***(pos – l + 1)th*** smallest element. We call it ***count***.

Therefor, ***count = (pos – l + 1) = (2-0+1) = 3***

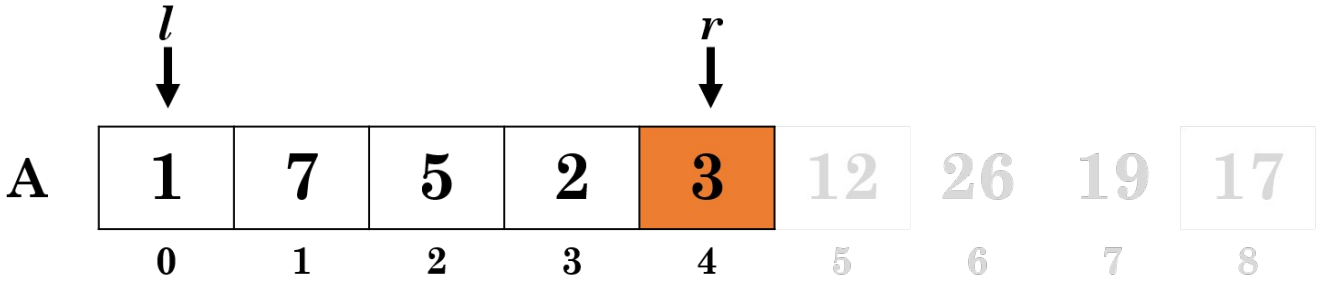
∴ ***pivot*** is the 3^{rd} smallest element in this array.

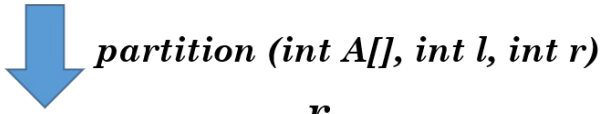
Three cases can happen.

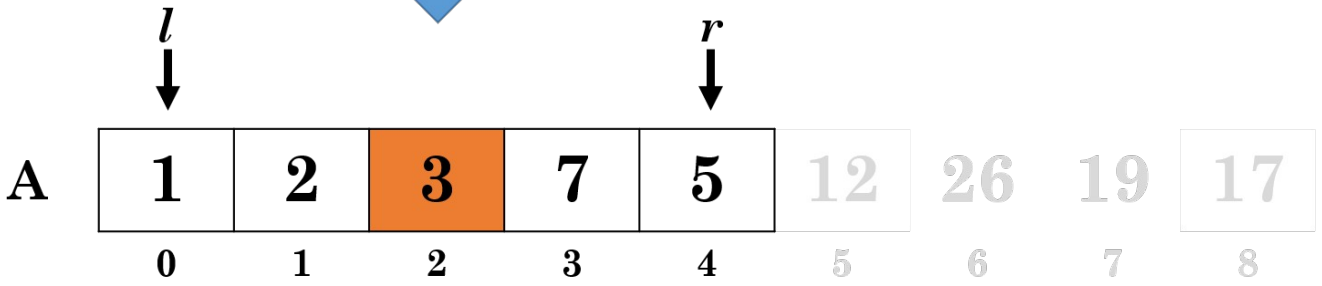
Case 1:

if ***(count == k)***

return ***A [pos]***







Case 2:

if ***(count > k)***

return ***kth_smallest_element(A, l, pos-1, k)***

Case 3:

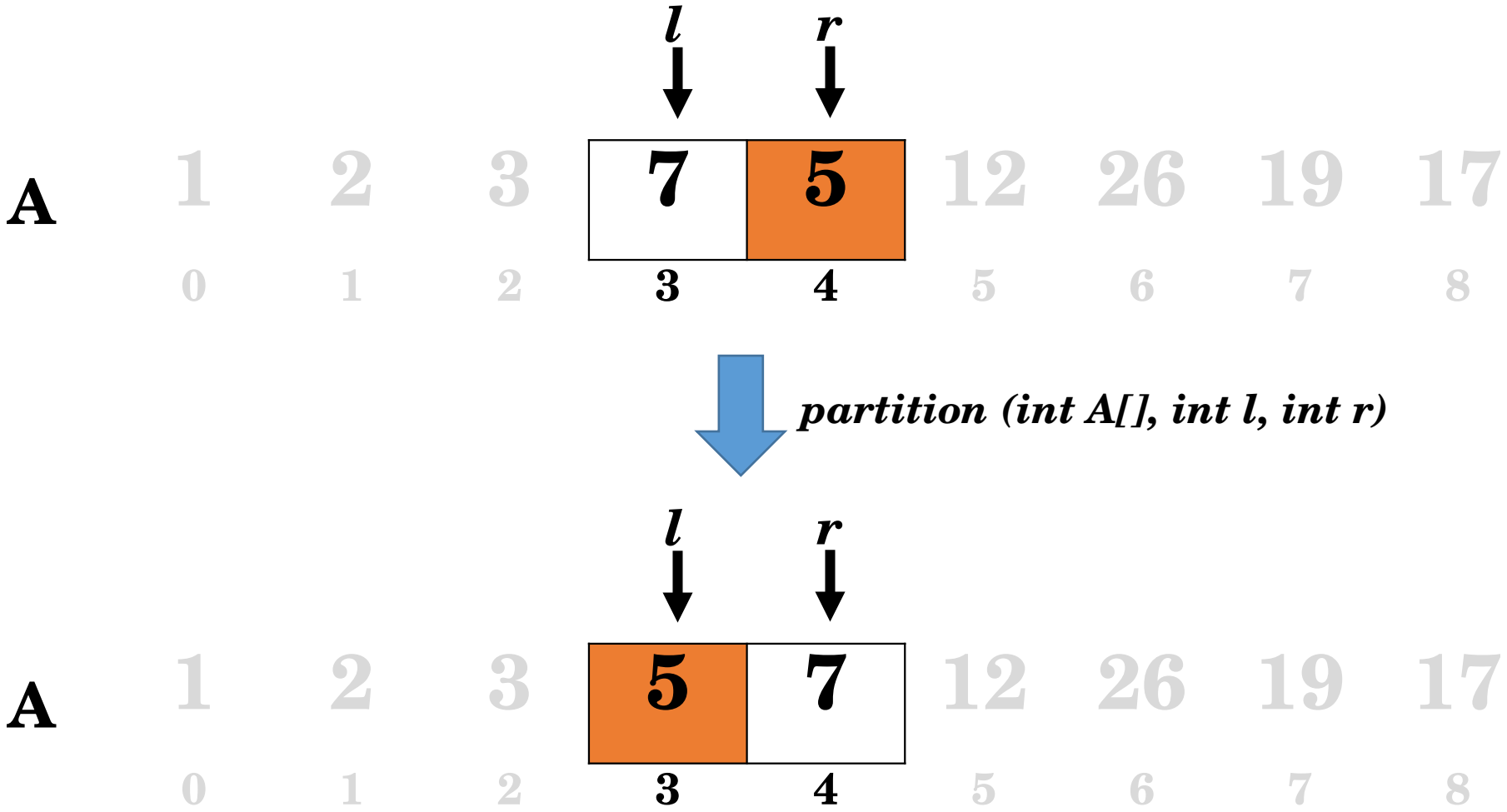
if ***(count < k)***

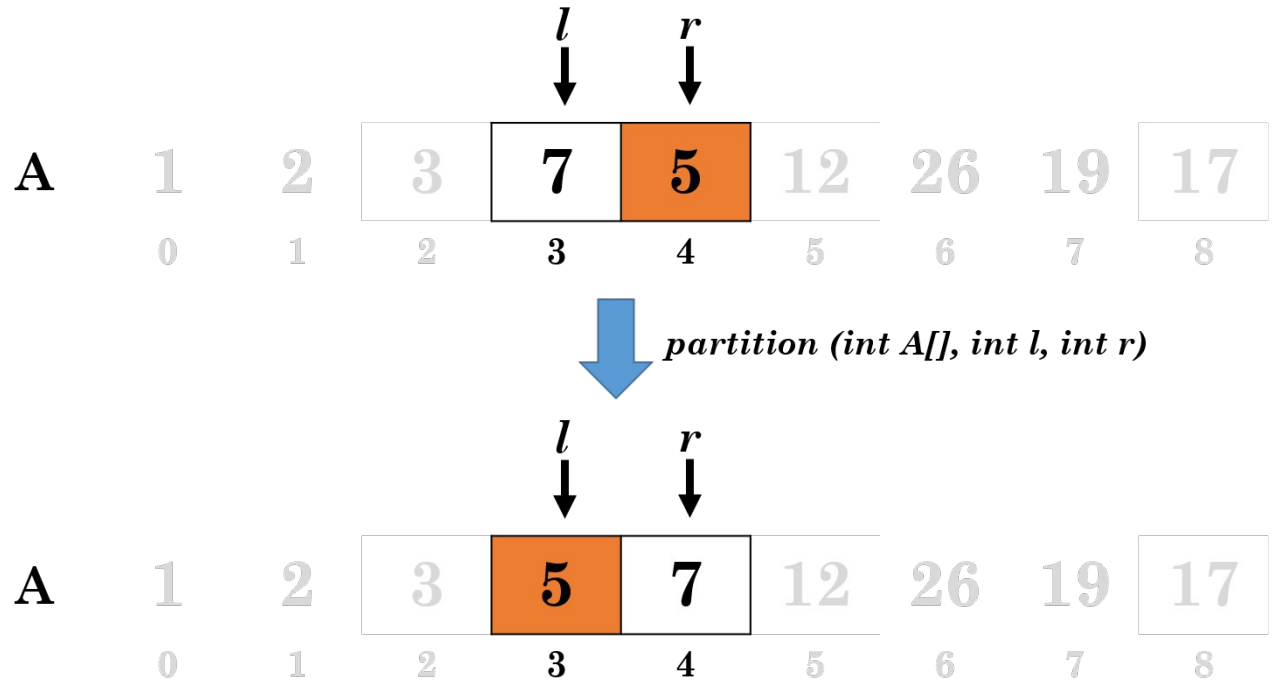
return ***kth_smallest_element(A, pos+1, r, k-count)***

4. Case 3 happens this time. So we recursively call the function

kth_smallest_element(A, pos+1, r, k-count)

or, kth_smallest_element(A, 3, 4, 1)





The ***partiition ( )*** function returns the current index of the ***pivot*** element, which we are calling ***pos***. in this case,

pos = 3

Remember, the ***pivot*** element is the ***(pos – l + 1)th*** smallest element. We call it ***count***.

Therefor, ***count = (pos – l + 1) = (3-0+1) = 4***

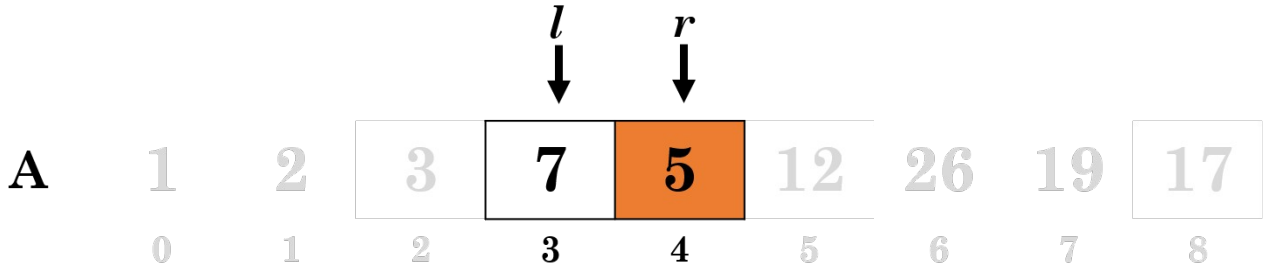
∴ ***pivot*** is the 4^{th} smallest element in this array.

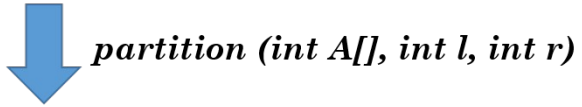
Three cases can happen.

Case 1:

if ***(count == k)***

    return ***A [pos]***

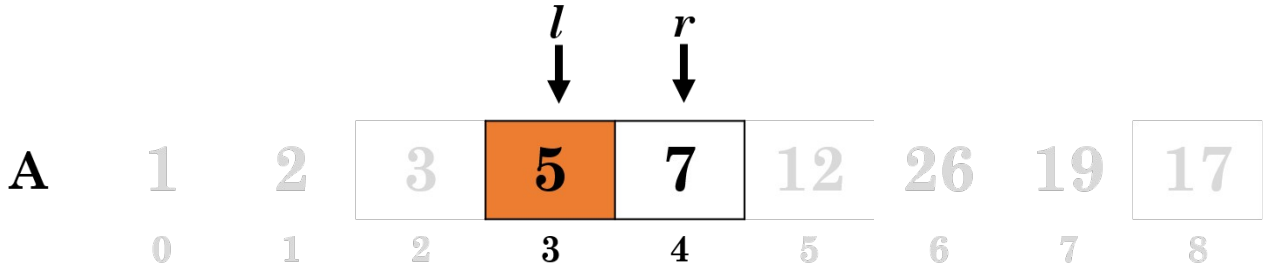




Case 2:

if ***(count > k)***

    return ***kth_smallest_element(A, l, pos-1, k)***



Case 3:

if ***(count < k)***

    return ***kth_smallest_element(A, pos+1, r, k-count)***

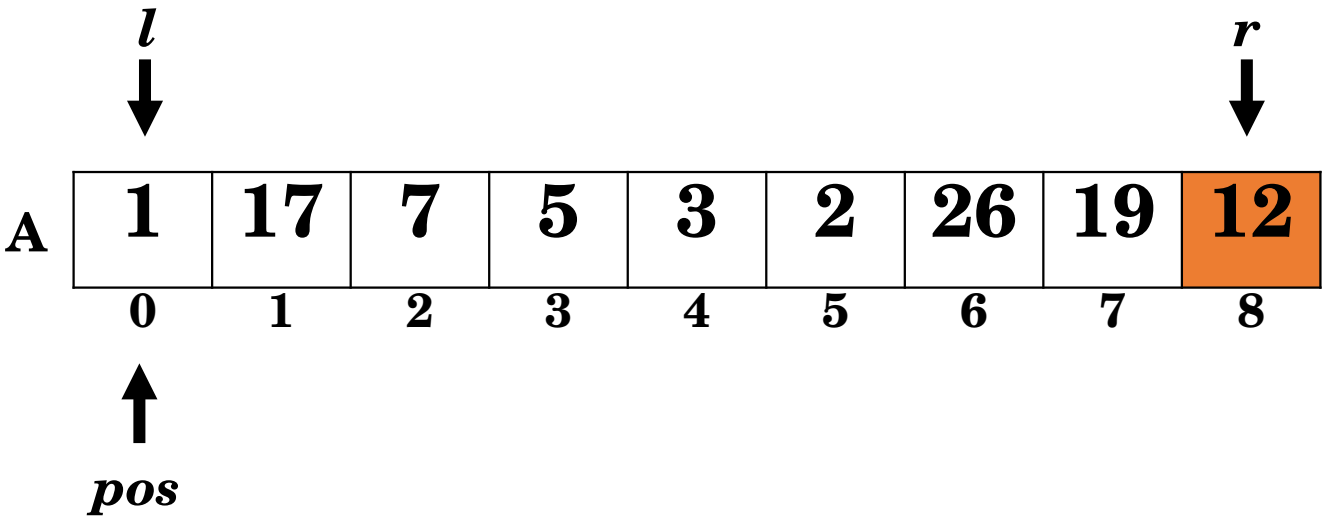
5. Case 1 happens this time. So we recursively call the function. So we

return ***A [pos]***

A [3] = 5

That means, 5 is our 4^{th} smallest element.

How does *partition ( )* works



Initially,

$l = 0$

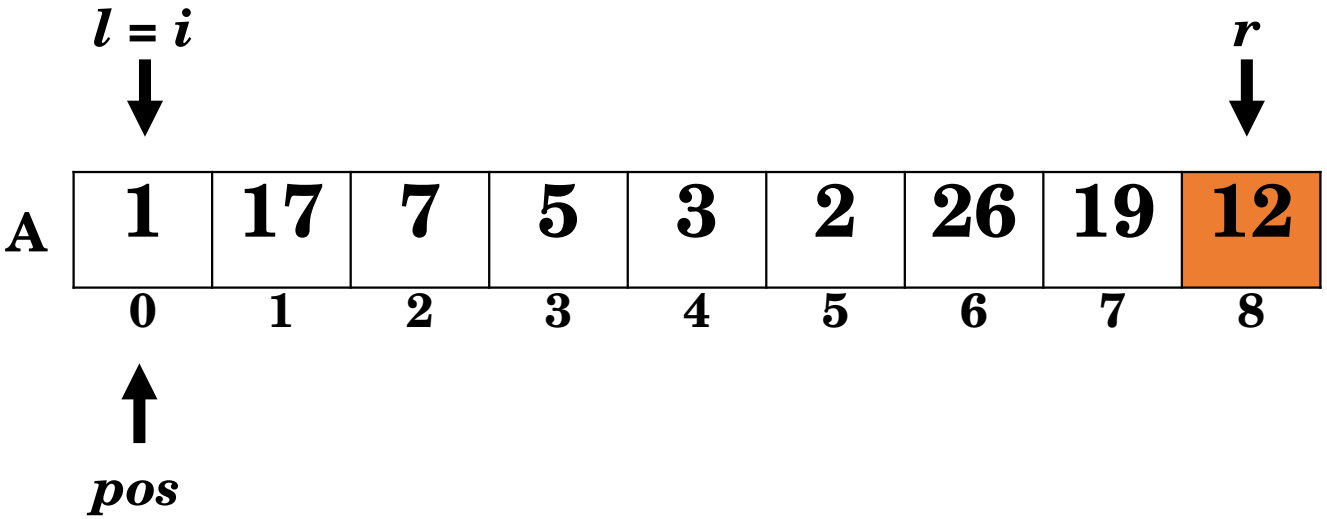
$r = 8$

$pivot = A[r] = A[8] = 12$

$pos = l = 0$

```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

How does *partition ( )* works



At *i == 0*

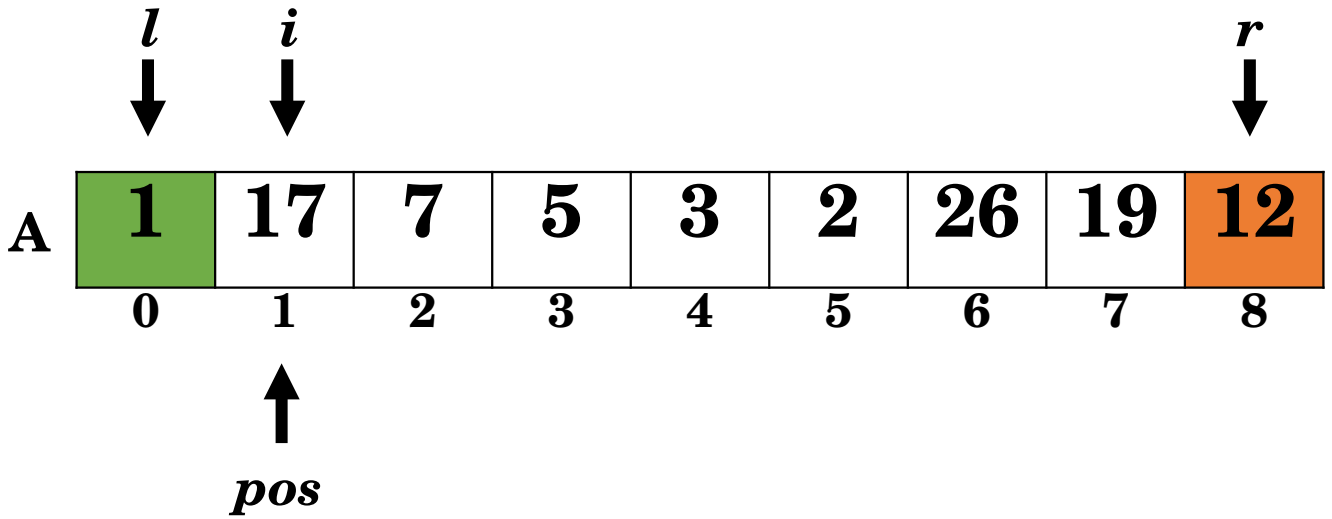
1 < 12

swap(1,1)

pos = 0 + 1 = 1

```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

How does *partition ( )* works



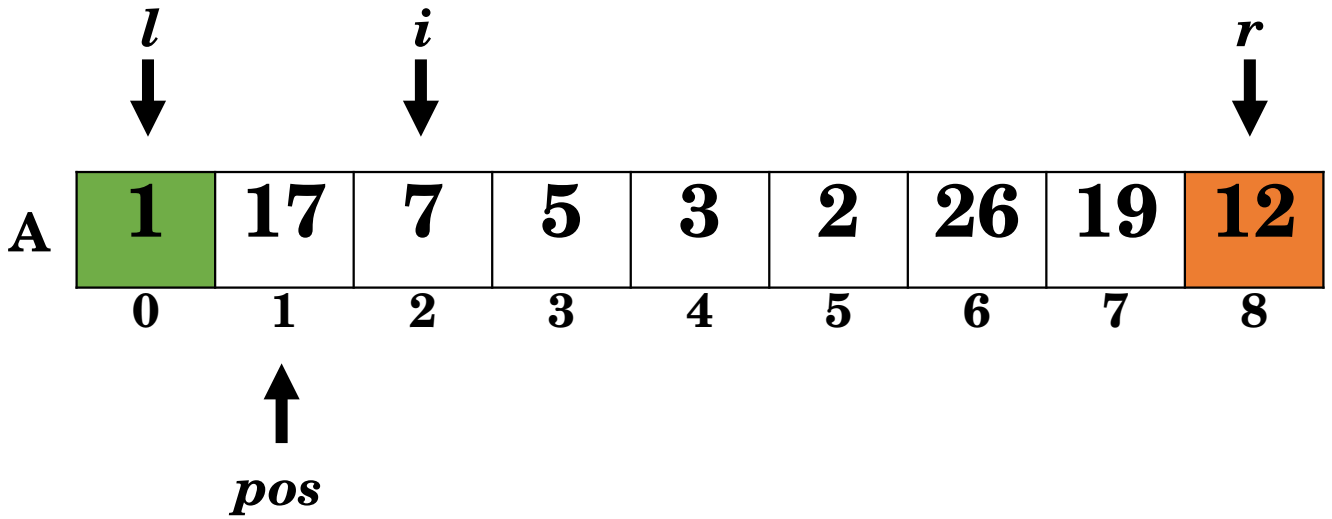
At *i == 1*

17 > 12

no swaping

```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

How does *partition ( )* works



At $i == 2$

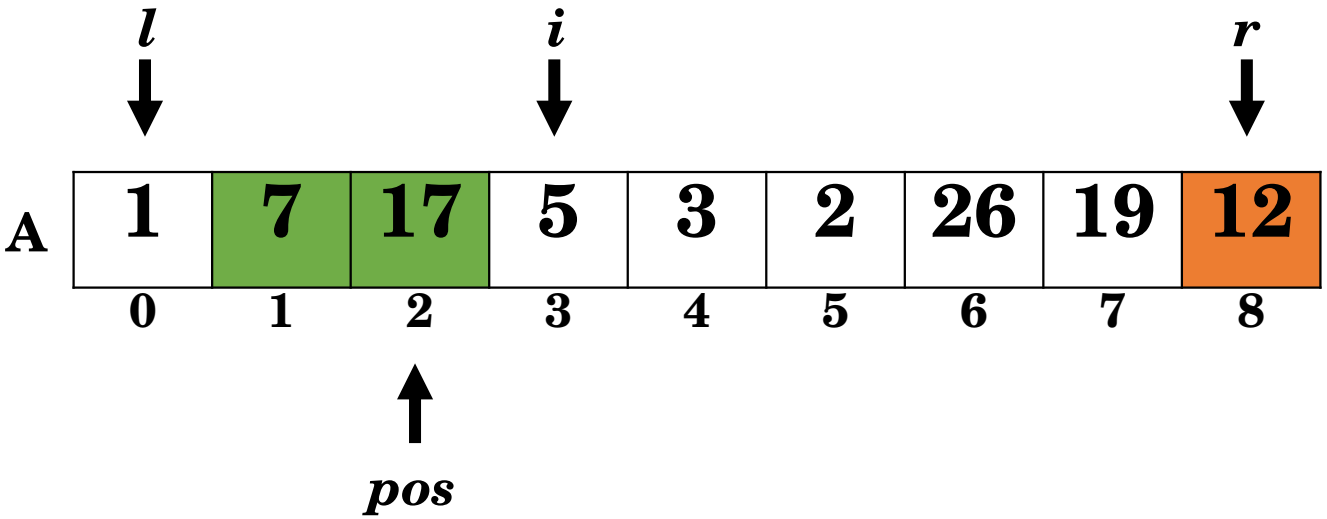
$7 < 12$

swap(7, 17)

pos = 1 + 1 = 2

```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

How does *partition ( )* works



At $i == 3$

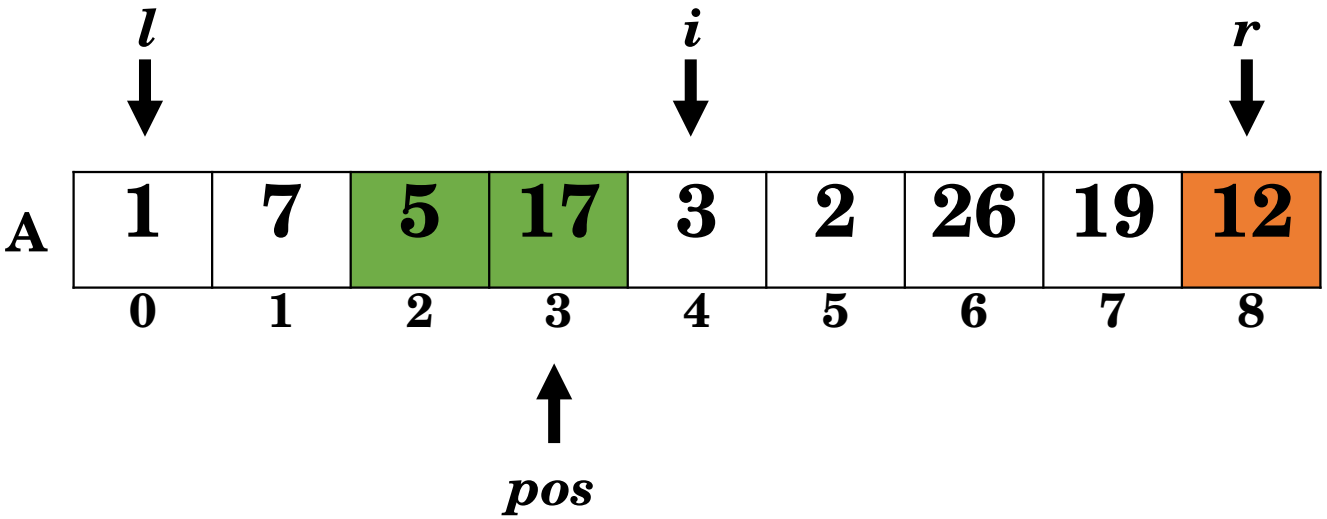
$7 < 12$

swap(5, 17)

pos = 2 + 1 = 3

```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

How does *partition ( )* works



At $i == 4$

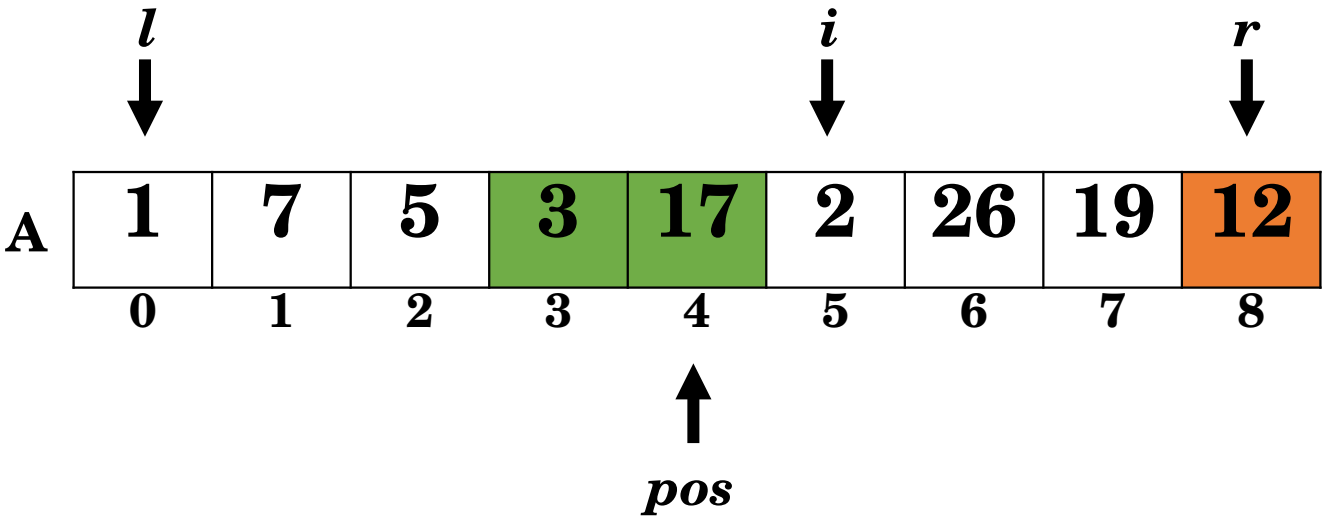
$3 < 12$

swap(3, 17)

$pos = 3 + 1 = 4$

```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

How does *partition ( )* works



At $i == 5$

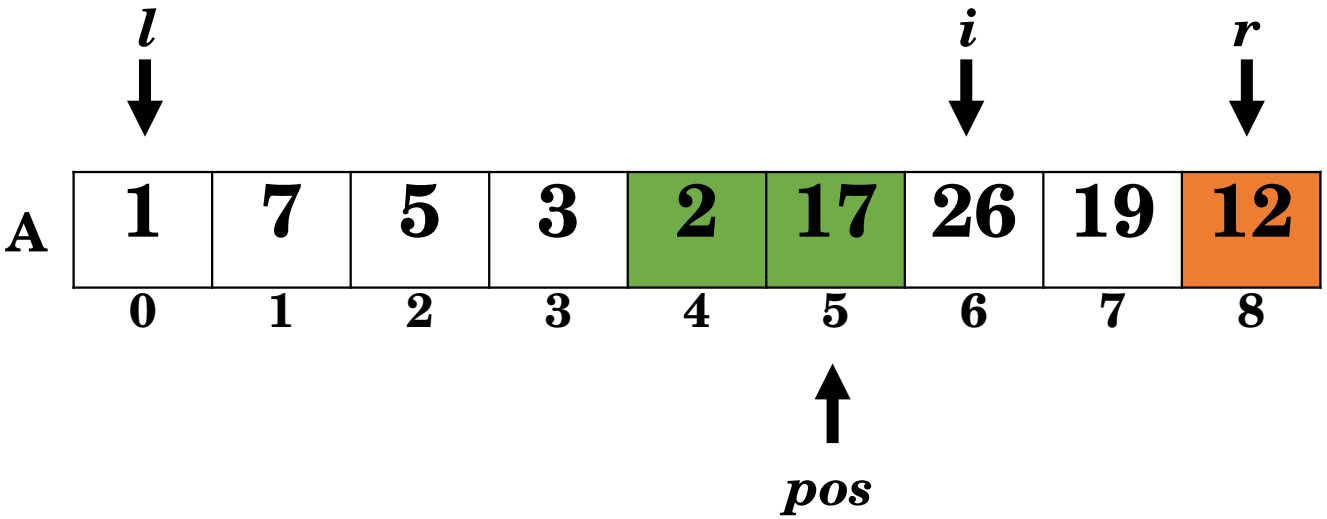
$2 < 12$

swap(2, 17)

pos = 4 + 1 = 5

```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

How does *partition ( )* works



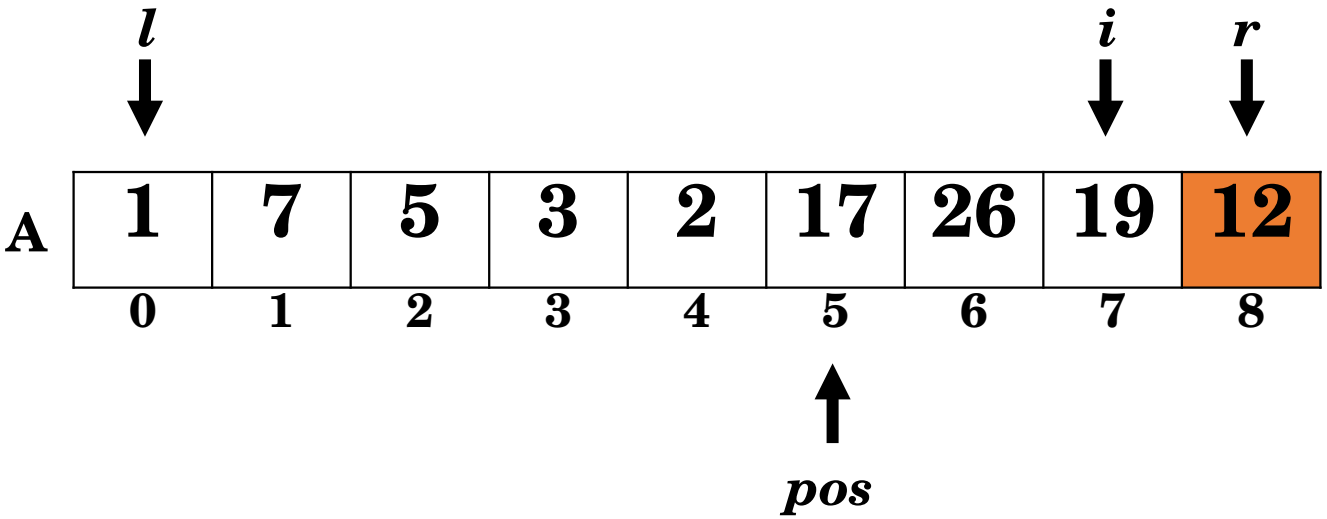
At $i == 6$

26 > 12

no swaping

```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

How does *partition ( )* works



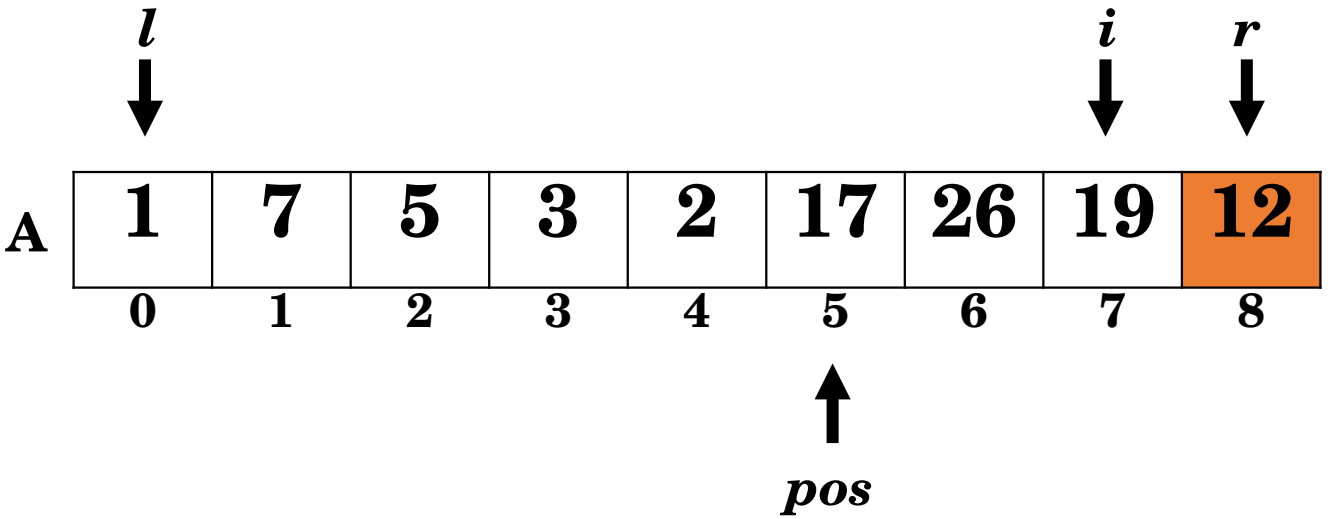
At $i == 7$

19 > 12

no swapping

```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

How does *partition ( )* works

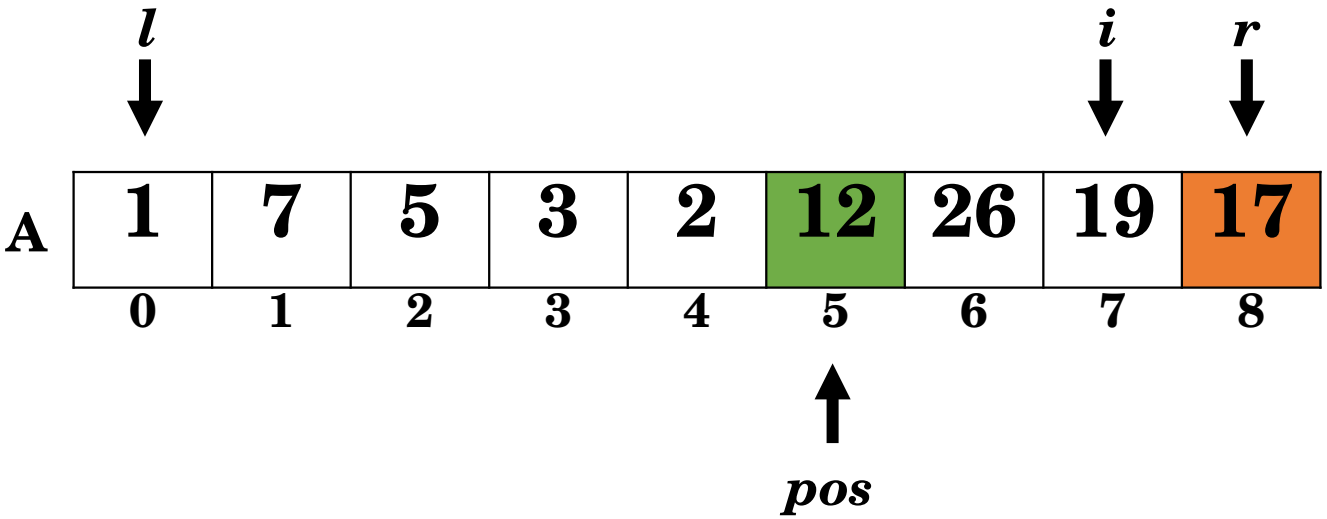


Finally,

swap(12, 17)

```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

How does *partition ( )* works




```
int partition(int A[ ], int l, int r)
{
    int pivot = A[r];
    int pos = l;
    for ( int i = l; i<= r-1; i++ ){
        if (A[i] <= pivot){
            swap(A[i], A[pos]);
            pos = pos + 1;
        }
    }
    swap(A[pos], A[r]);
return pos;
}
```

This algorithm can also be used to

- Find k-th largest element
- Find median of an array of elements