DESIGN AND ANALYSIS OF ALGORITHMS

UNIT-I

Module 2 Divide and Conquer

Dr B.Kumar Babu

Assistant Professor,

Dept. of CSE,

School Of Technology,

GITAM(Deemed to be) University,

Hyderabad.

Agenda

- Algorithm specification
- Performance Analysis
- **Divide and Conquer:**
  - The general method
  - Binary search,
  - finding maximum and minimum,
  - Merge sort
  - Quick sort
  - Selection (Quick-Select)
  - Strassen's Matrix multiplication.

Divide and Conquer Technique

General Method

- The Divide and Conquer Technique splits n inputs into k subsets , 1< k ≤ n, yielding k sub-problems.
- These sub-problems will be solved and then combined by using a separate method to get a solution to a whole problem.
- If the sub-problems are large, then the Divide and Conquer Technique will be reapplied.
- Often sub-problems resulting from a Divide and Conquer Technique are of the same type as the original problem.
- For those cases the reapplication of the Divide and Conquer Technique is naturally expressed by a recursive algorithm.
- Now smaller and smaller problems of the same kind are generated until sub-problems that are small enough to be solved without splitting are produced.

```
Algorithm DAndC(p)
{
   if Small(p) then
          return s(p);
   else
   {
          Divide p into smaller instances p1,p2,…….,pk, k≥1;

          Apply DAndC to each of these sub-problems;

          return Combine(DAndC(p1), DAndC(p2),…….,DAndC(pk));
   }
}
```

If the size of p is n and the sizes of the k sub-problems are n1,n2,….,nk, then the computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1)+T(n_2)+\ldots\ldots+T(n_k)+f(n) & \text{Otherwise} \end{cases}$$

- Where T(n) is the time for DAndC on any input of size n and g(n) is the time to compute the answer directly for small inputs.
- The function f(n) is the time for dividing p and combining the solutions to sub-problems.

The Complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n=1 \\ a\,T(n/b) + f(n) & n>1 \end{cases}$$

Where a and b are known constants.

We assume that T(1) is known and n is a power of b (i.e $n=b^k$)

Applications of DandC

1. Binary search Algorithm

Iterative algorithm

```
Algorithm BinSearch(a,n,x)
{
    low:=1;   high:=n;
    while(low≤high)
    {
        mid:=(low+high)/2;
        if( x<a[mid] ) then high:=mid-1;
        else  if( x> a[mid] ) then low:=mid+1;
              else    return mid;
    }
  return 0;
}
```

Recursive Algorithm (Divide and Conquer Technique)

```
Algorithm BinSrch (a,i,l,x)
//given an array a [i:l]of elements in nondecreasing
//order,1≤i≤l,determine whether x is present,and
//if so, return j such that x=a[j]; else return 0.
{
    if(l=i) then                    // If small(P)
    {
            if(x=a[i]) then return i;
            else return 0;
    }
    else
    {  //Reduce p into a smaller subproblem.
        mid:=  (i+l)/2
        if(x=a[mid]) then return mid;
        else if (x<a[mid]) then
                return BinSrch (a,i,mid-1,x);
        else    return BinSrch(a,mid+1,l,x);
            }
    }
```

Time complexity of Binary Search

If the time for diving the list is a constant, then the computing time for binary search is described by the recurrence relation

$$T(n) = \begin{cases} c_1 & n=1,\ c1 \text{ is a constant} \\ T(n/2) + c_2 & n>1,\ c2 \text{ is a constant} \end{cases}$$

$$\begin{aligned} T(n) &= T(n/2) + c2 \\ &= T(n/4)+c2+c2 \\ &= T(n/8) + c2+c2+c2 \\ &= T(n/2^3) + 3c2 \\ &\ldots\ldots \\ &\ldots\ldots \\ &= T(1)+ kc2 \\ &= c1+kc2 = c1+ \log n * c2 = O(\log n) \end{aligned}$$

Assume $n=2^k$, then

Time Complexity of Binary Search

Successful searches:

best	average	worst
O(1)	O(log n)	O(log n)

Unsuccessful searches :

best	average	worst
O(log n)	O(log n)	O(log n)

2. Merge Sort

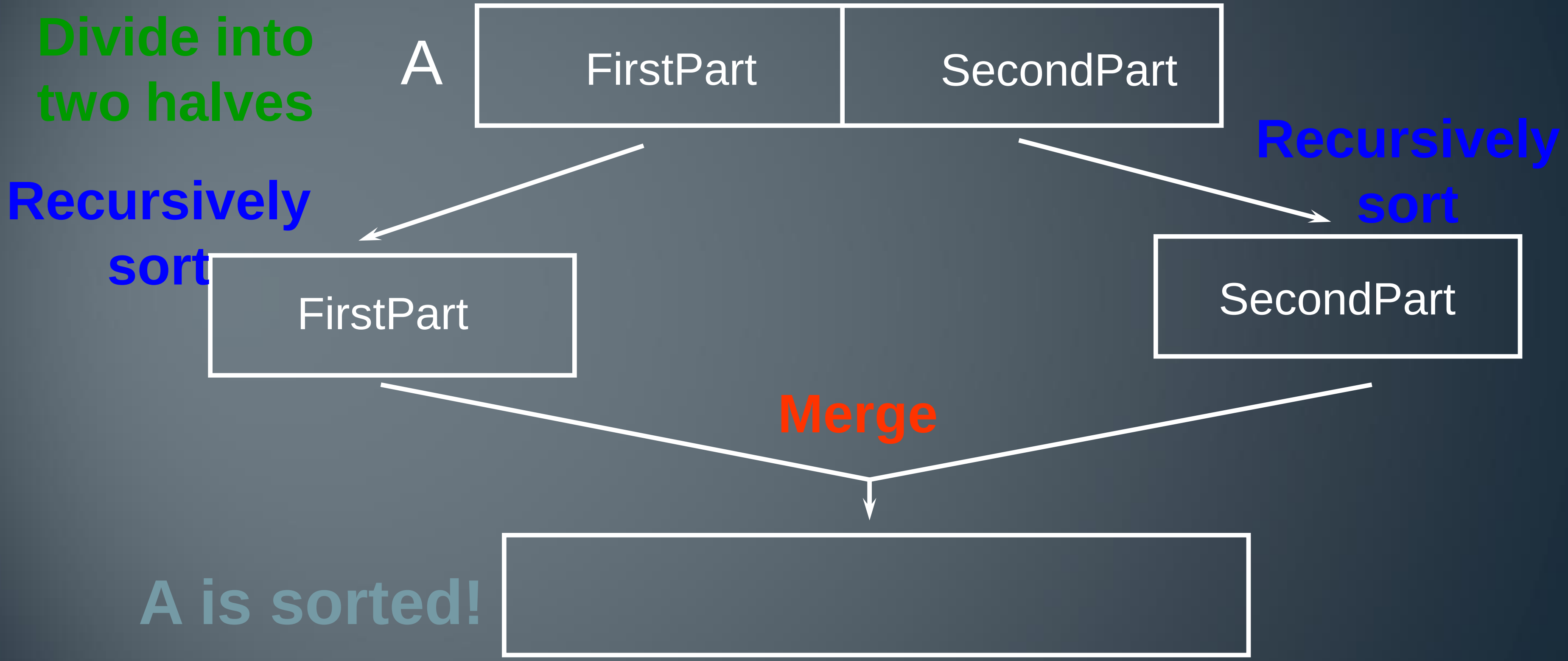
1. **Base Case**, solve the problem **directly** if it is small enough(only one element).

2. **Divide** the problem into two or more **similar and smaller** subproblems.

3. **Recursively** solve the subproblems.

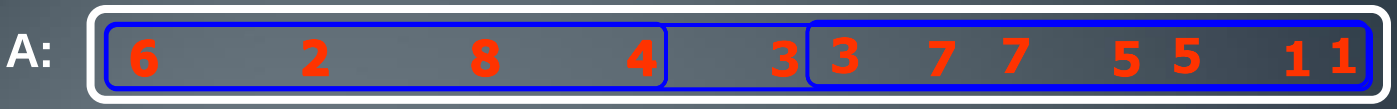
4. **Combine** solutions to the subproblems.

Merge Sort: Idea



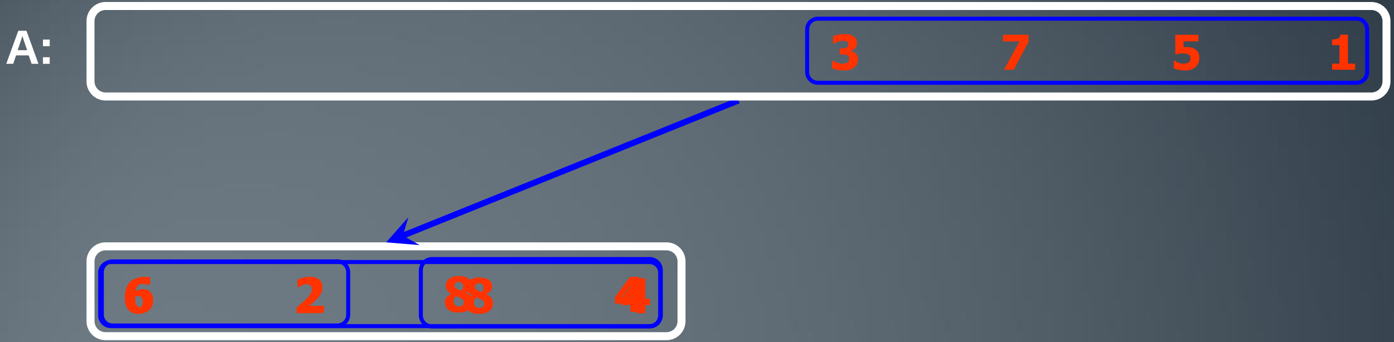
Merge-Sort(A, 0, 7)

Divide



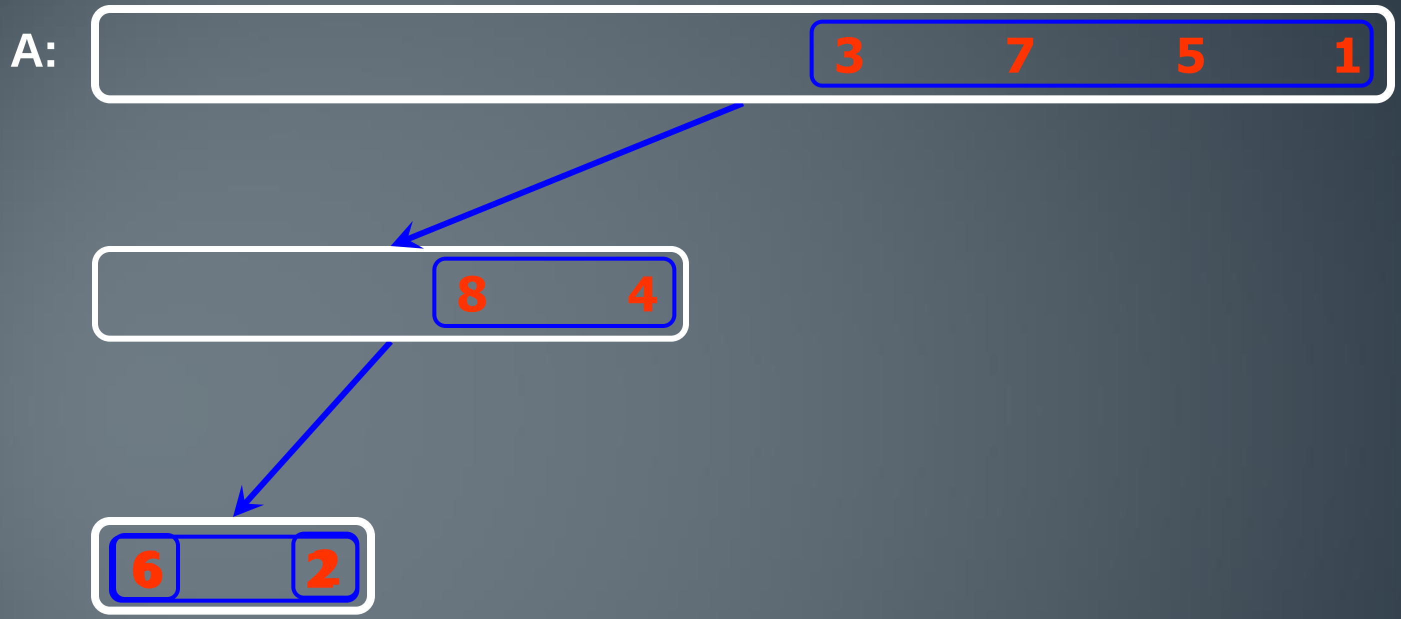
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 3) , divide



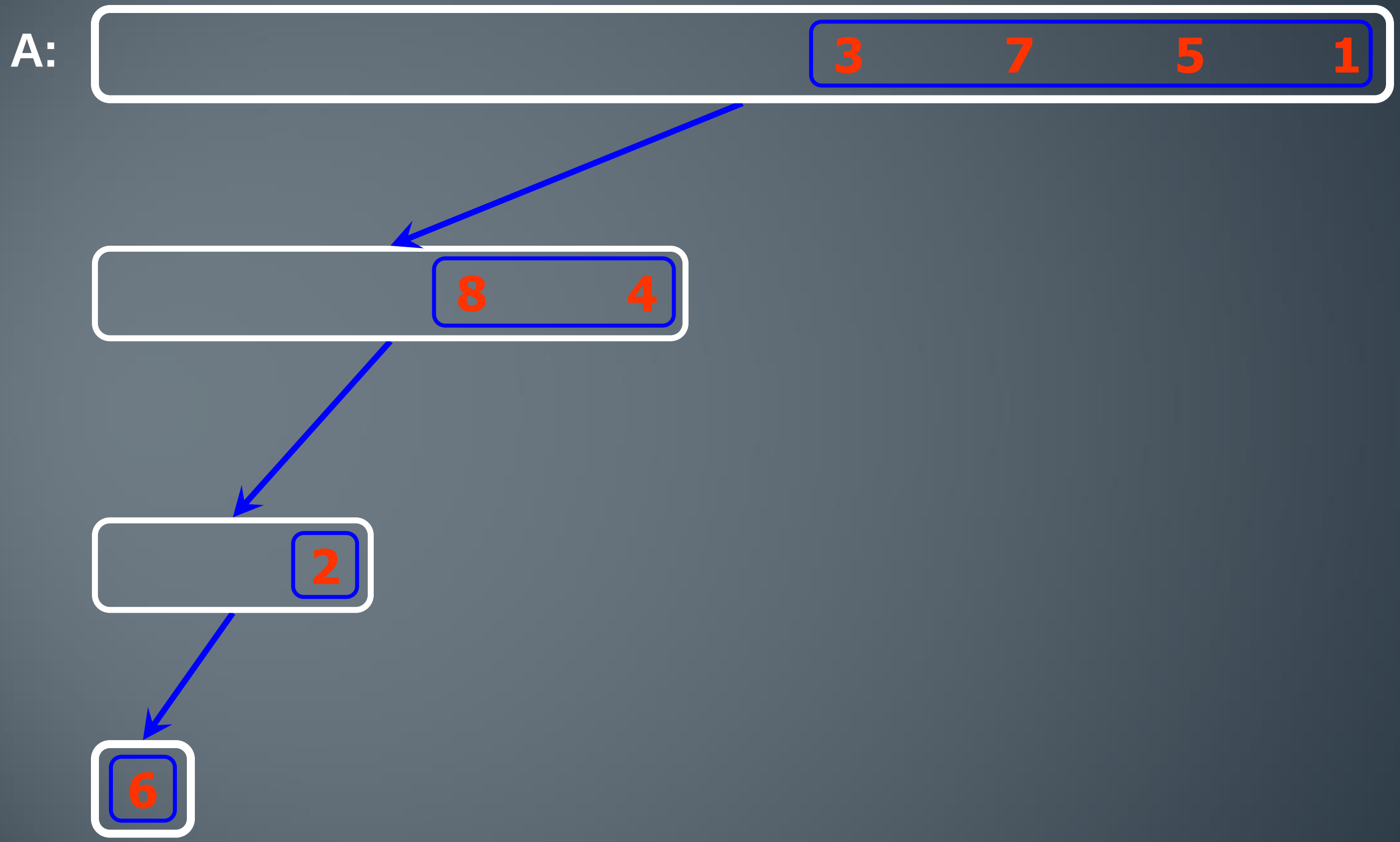
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 1), divide



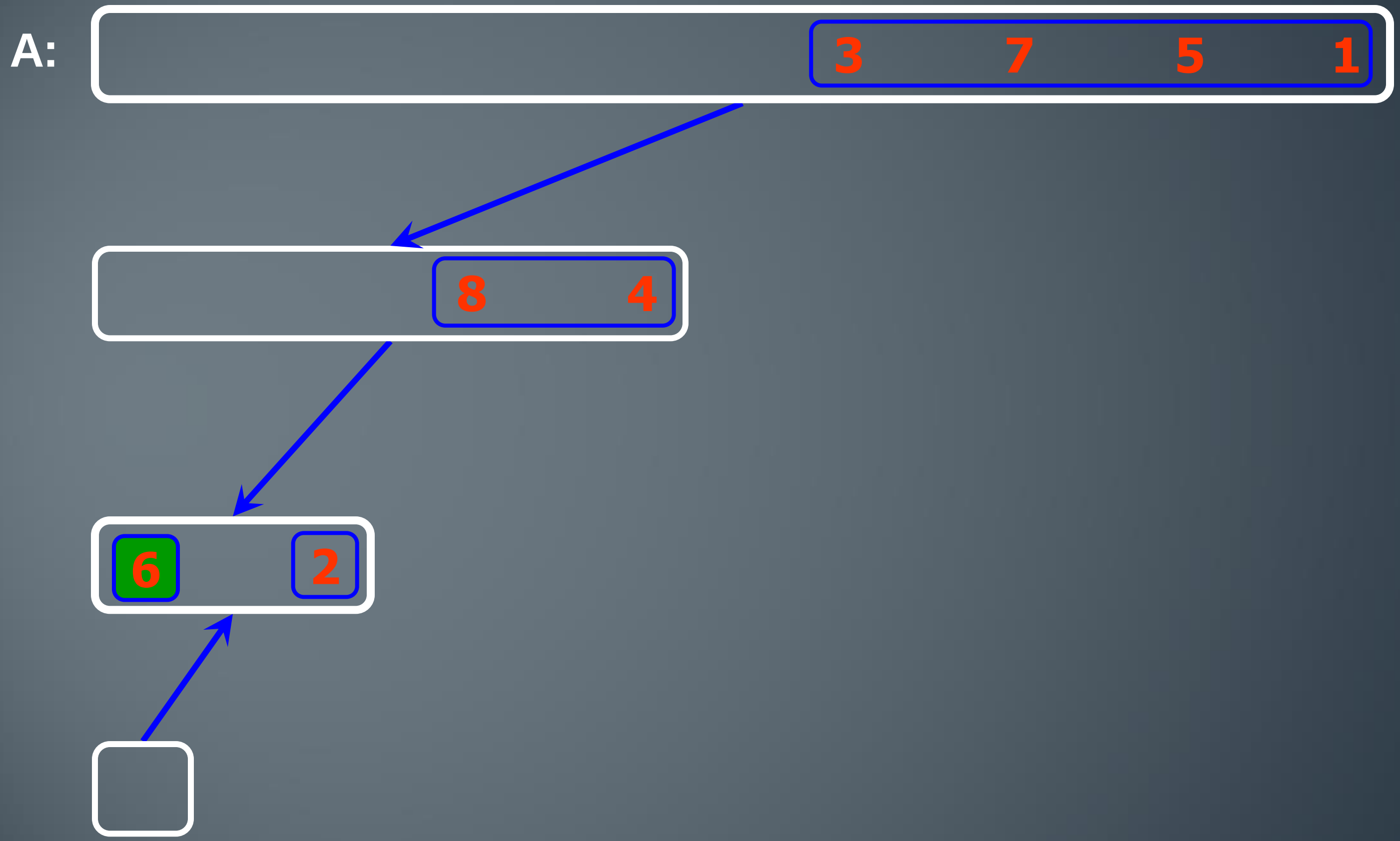
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 0) , base case



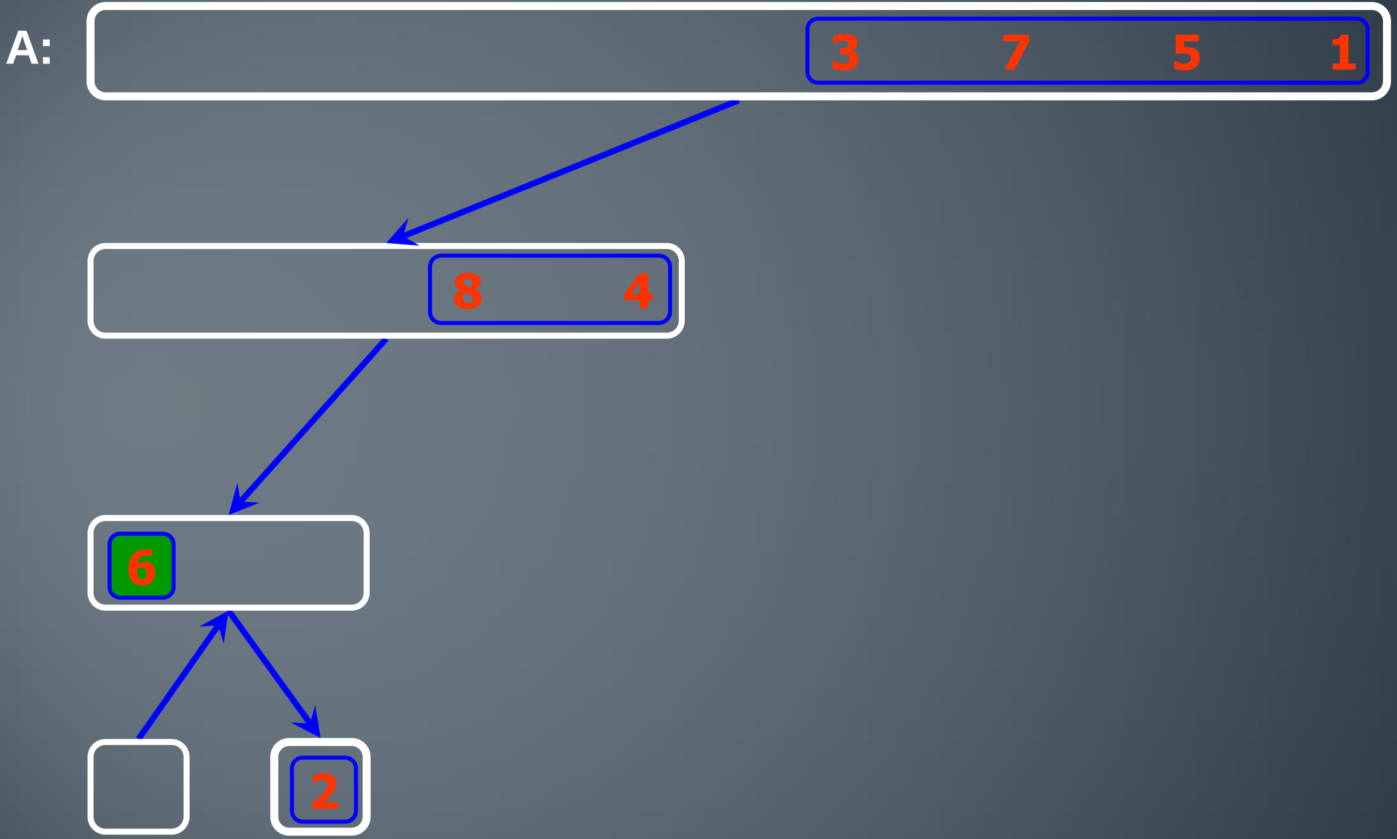
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 0), return



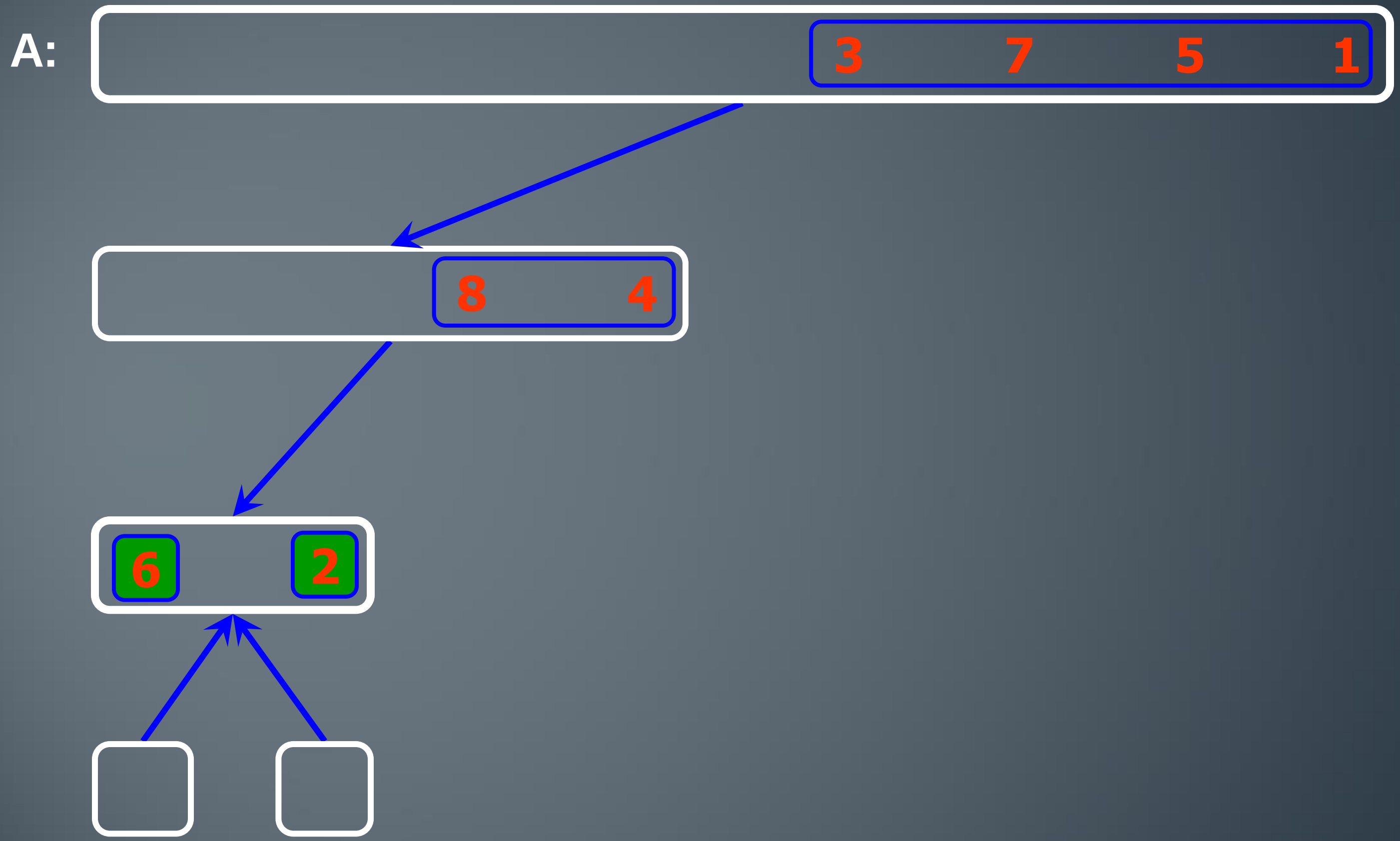
Merge-Sort(A, 0, 7)

Merge-Sort(A, 1, 1), base case

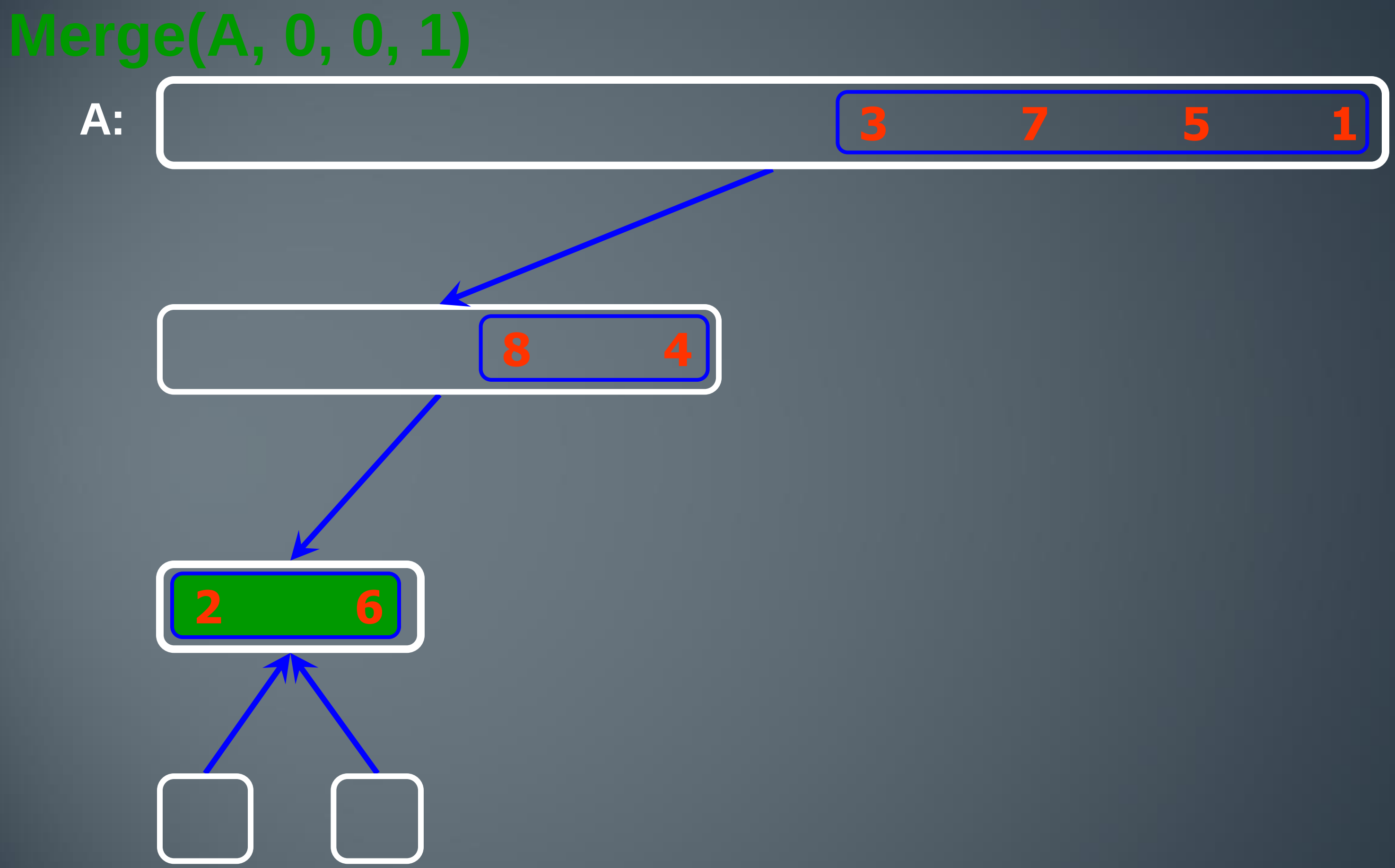


Merge-Sort(A, 0, 7)

Merge-Sort(A, 1, 1), return

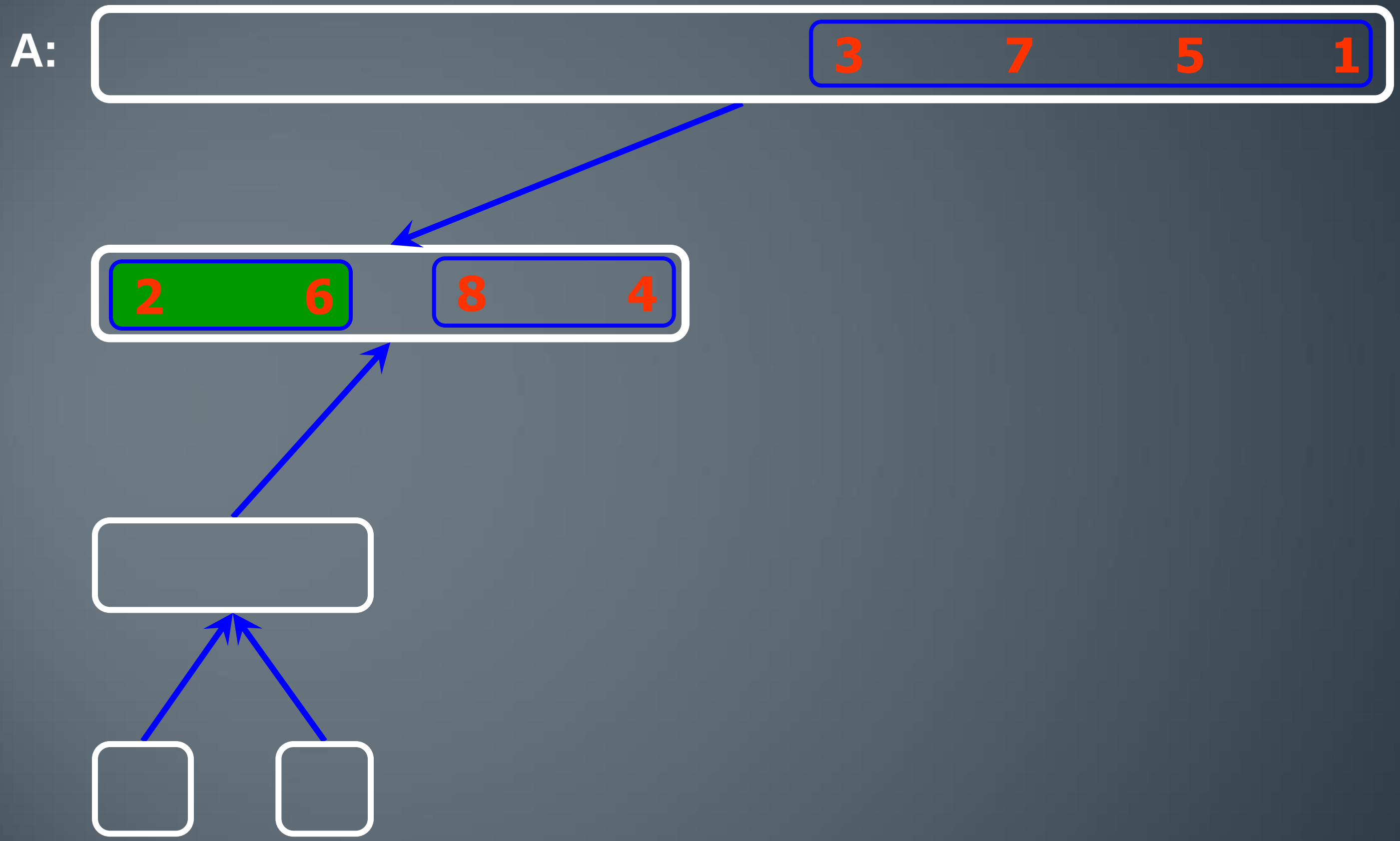


Merge-Sort(A, 0, 7)



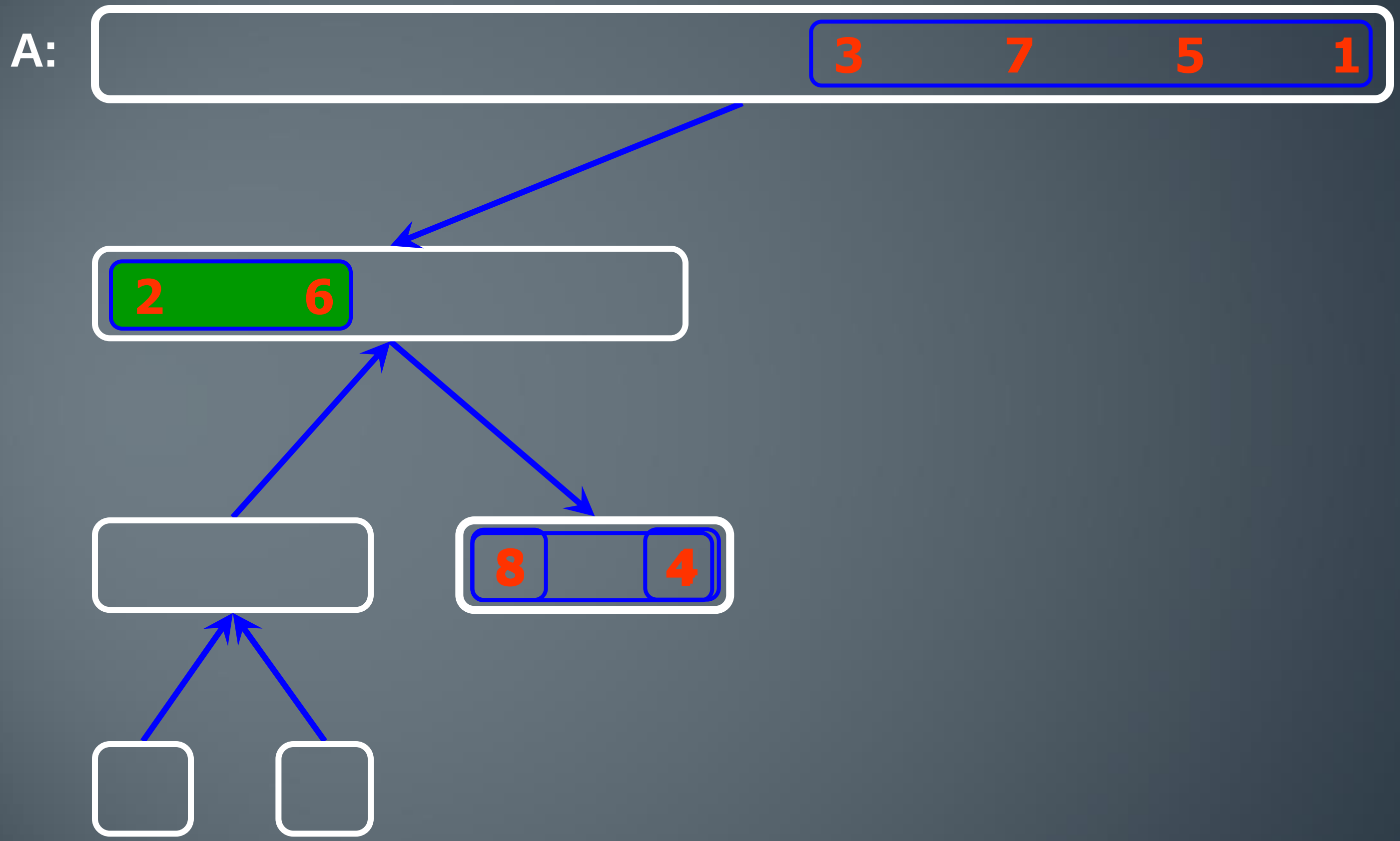
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 1), return



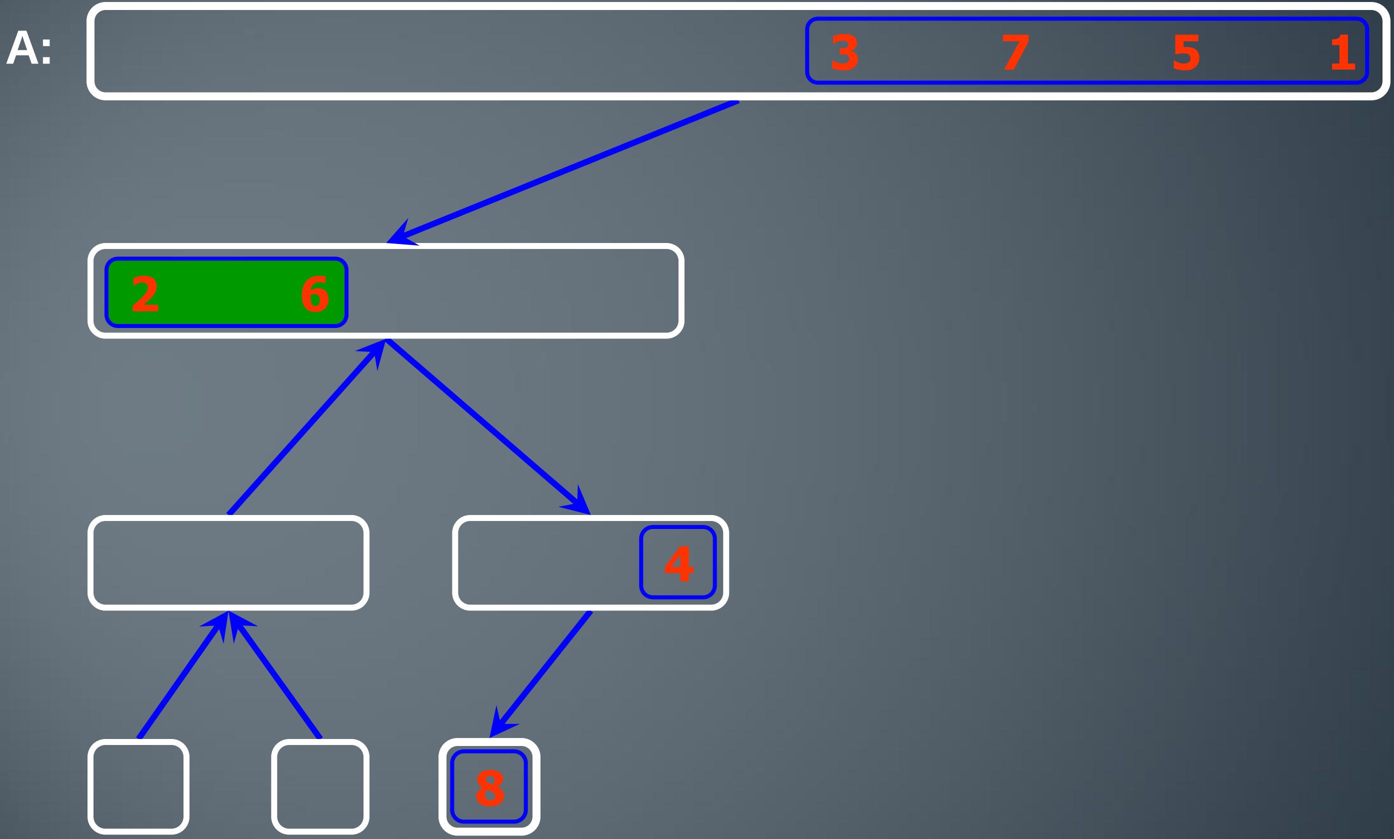
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 3) , divide



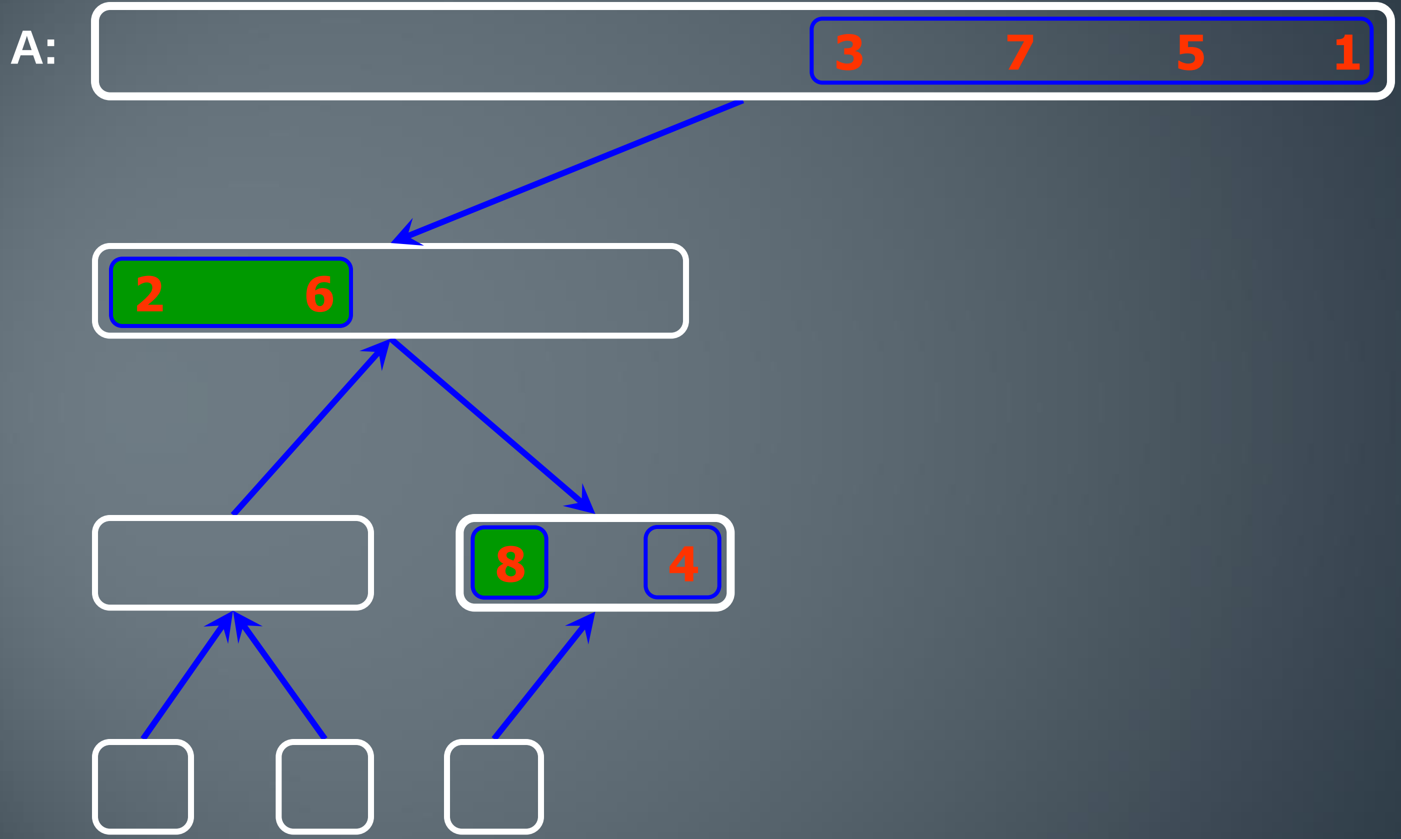
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 2), base case



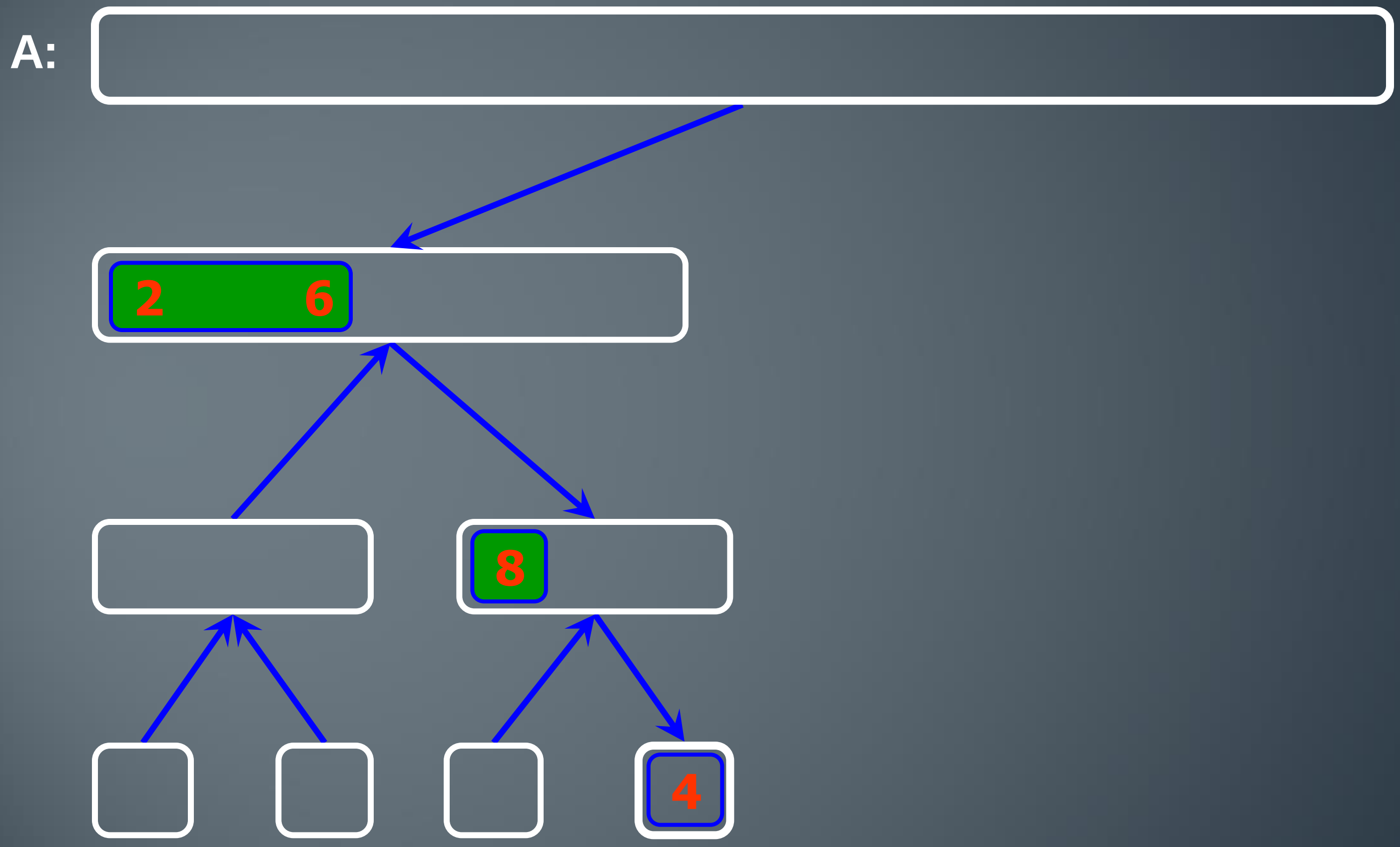
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 2), return



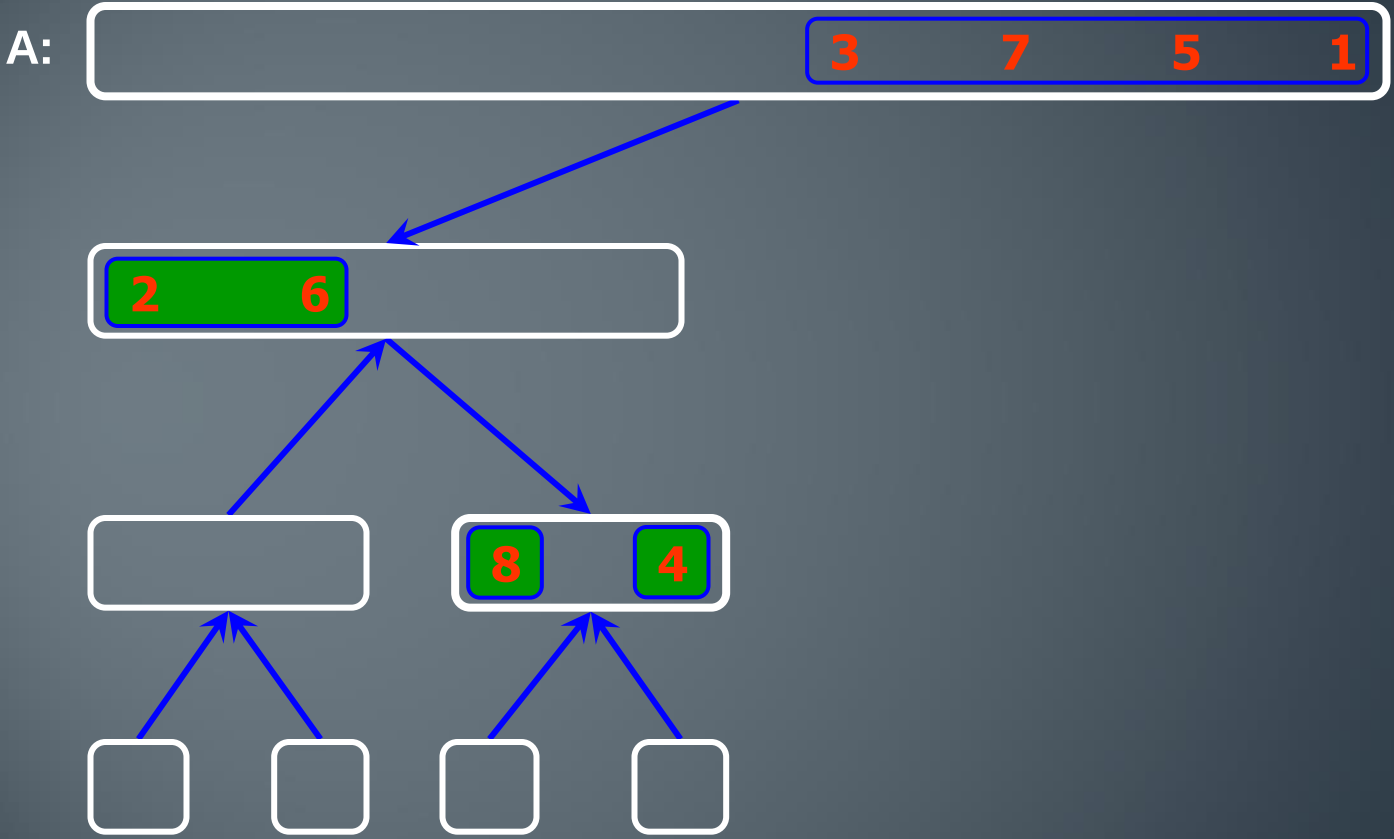
Merge-Sort(A, 0, 7)

Merge-Sort(A, 3, 3), base case

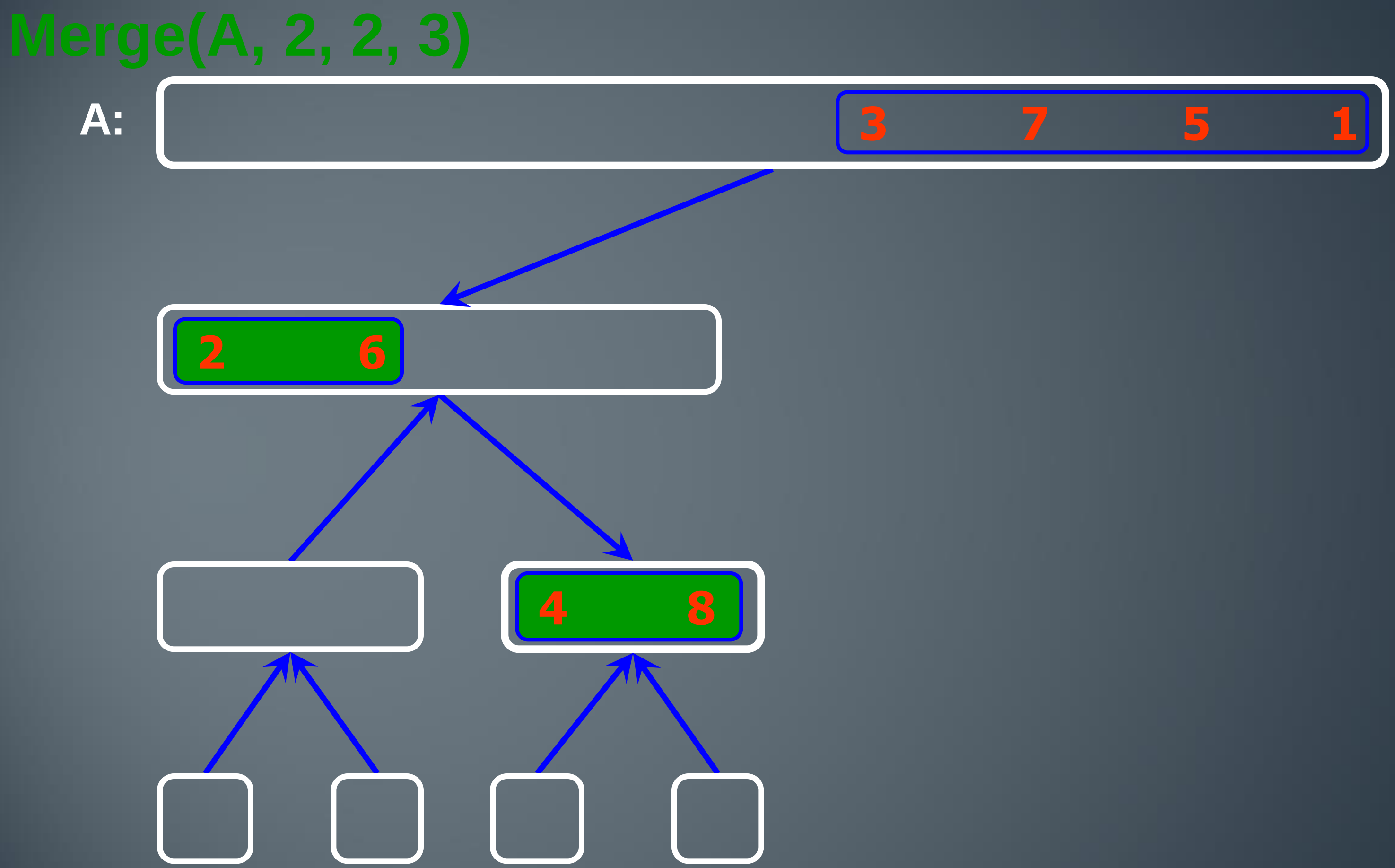


Merge-Sort(A, 0, 7)

Merge-Sort(A, 3, 3), return

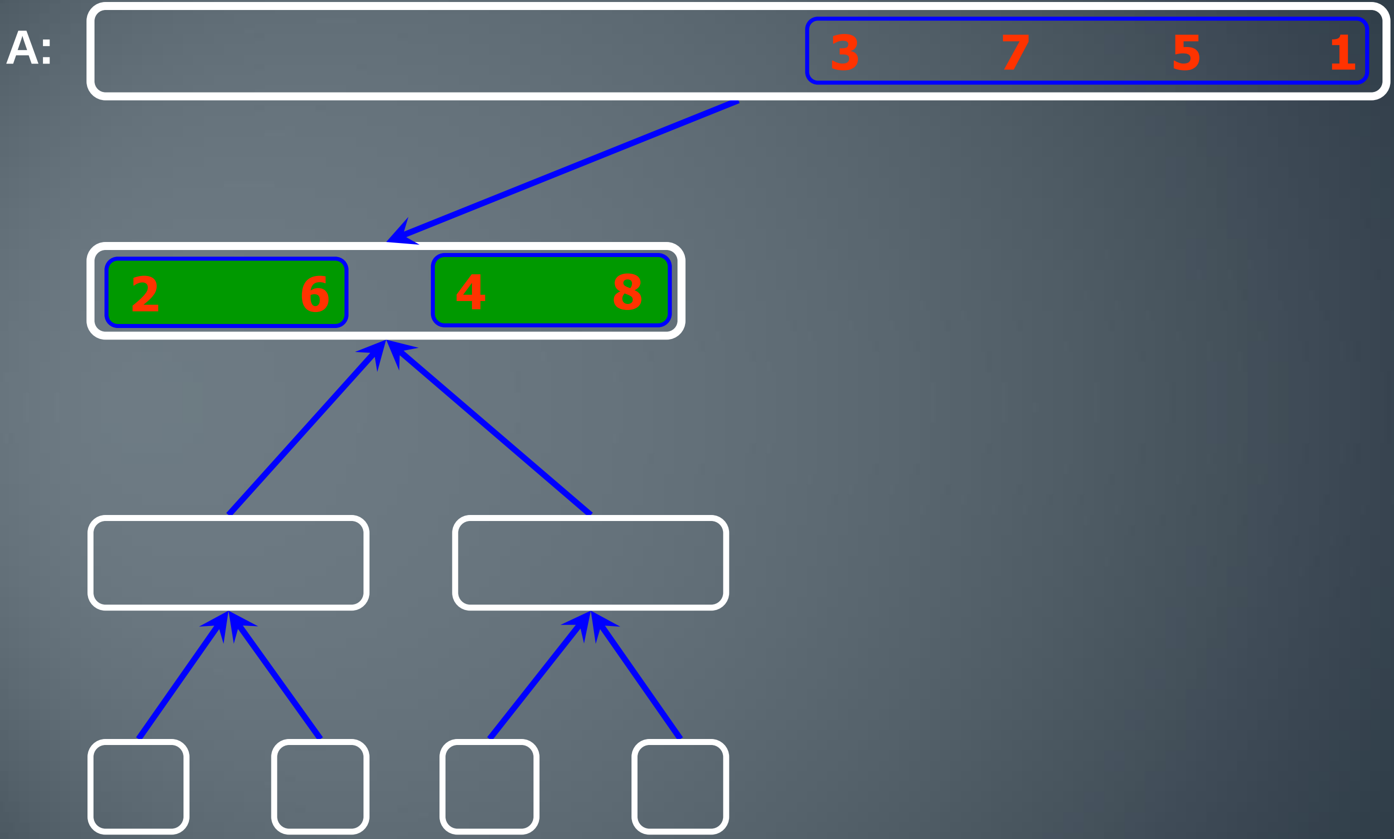


Merge-Sort(A, 0, 7)



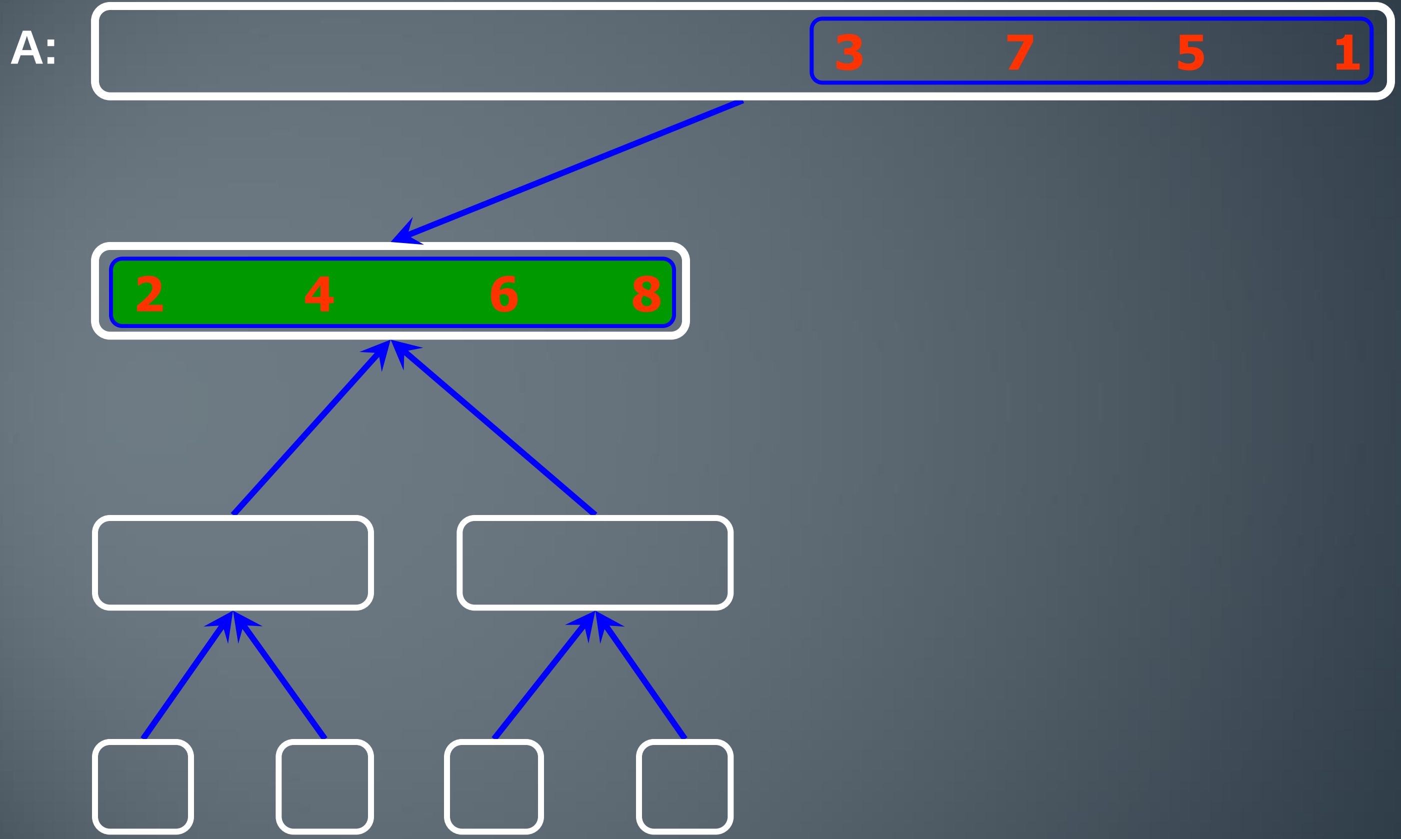
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 3), return



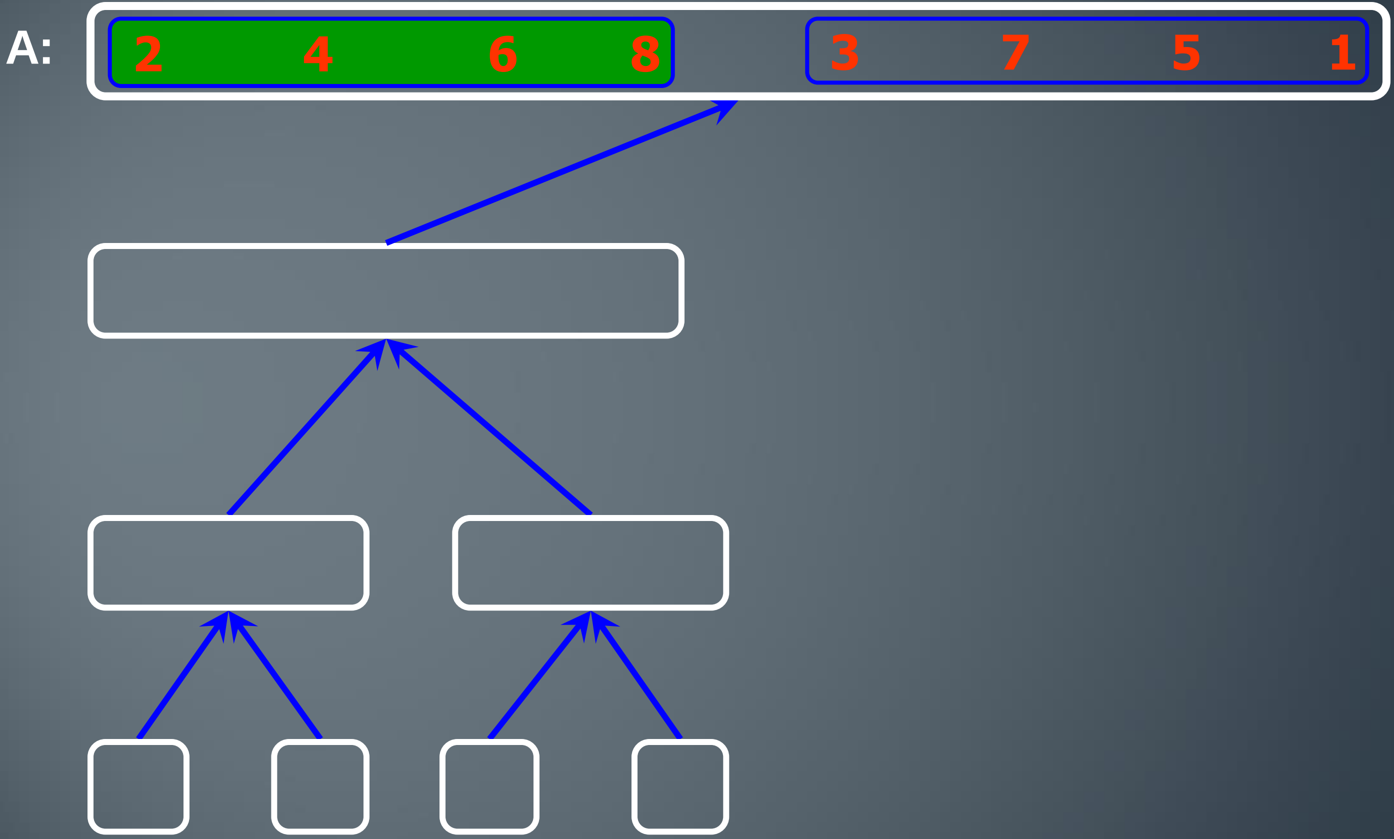
Merge-Sort(A, 0, 7)

Merge(A, 0, 1, 3)

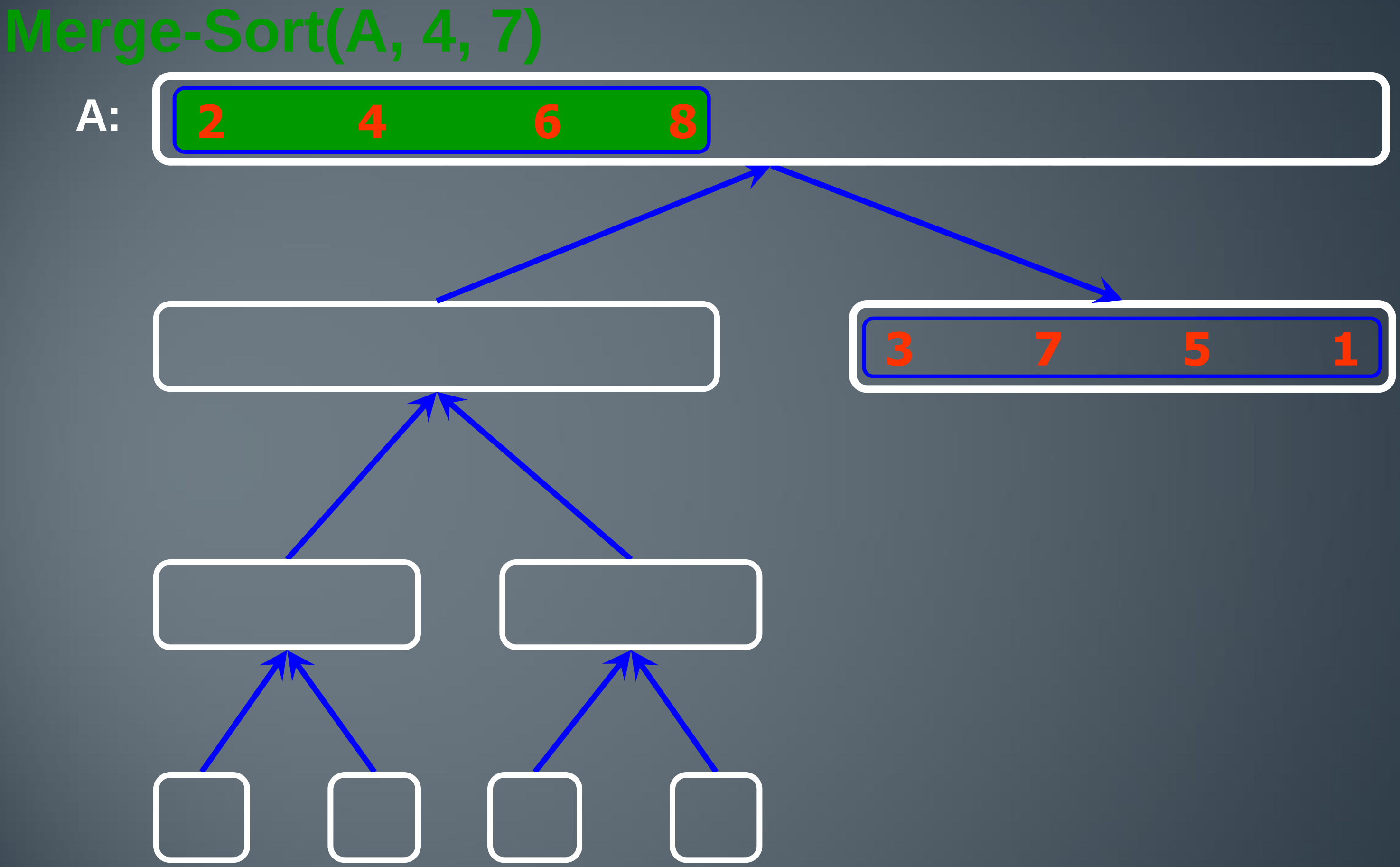


Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 3), return

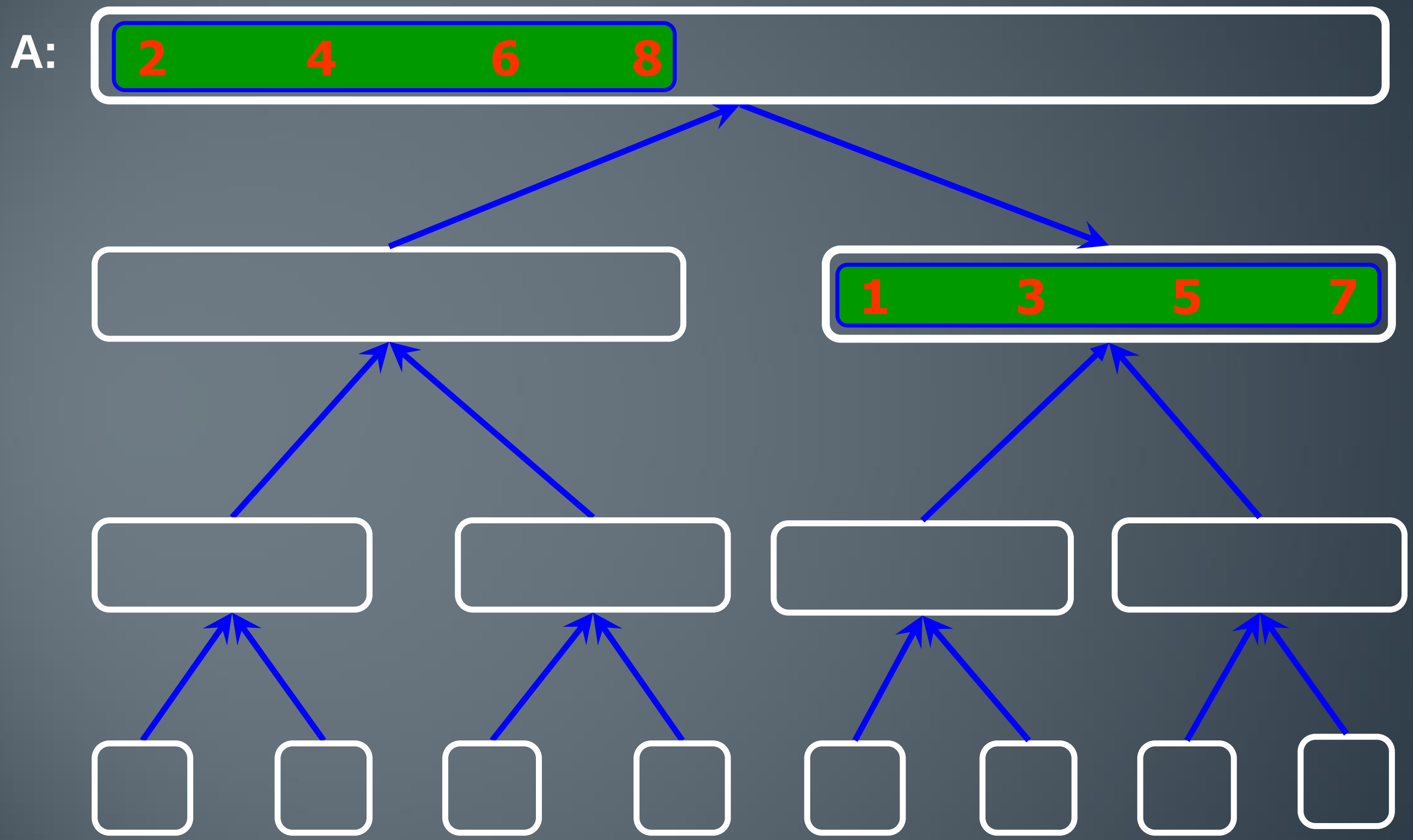


Merge-Sort(A, 0, 7)



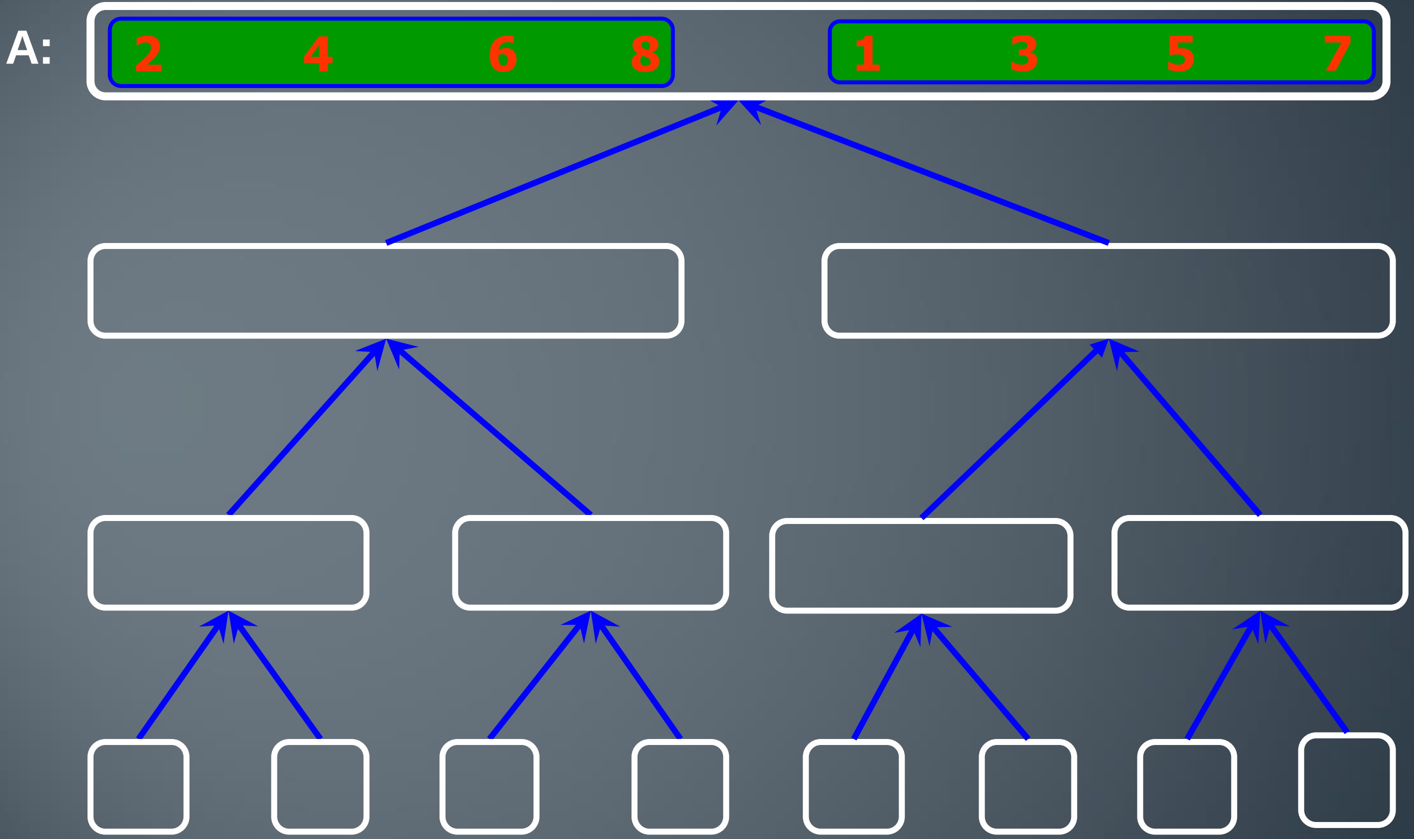
Merge-Sort(A, 0, 7)

Merge (A, 4, 5, 7)



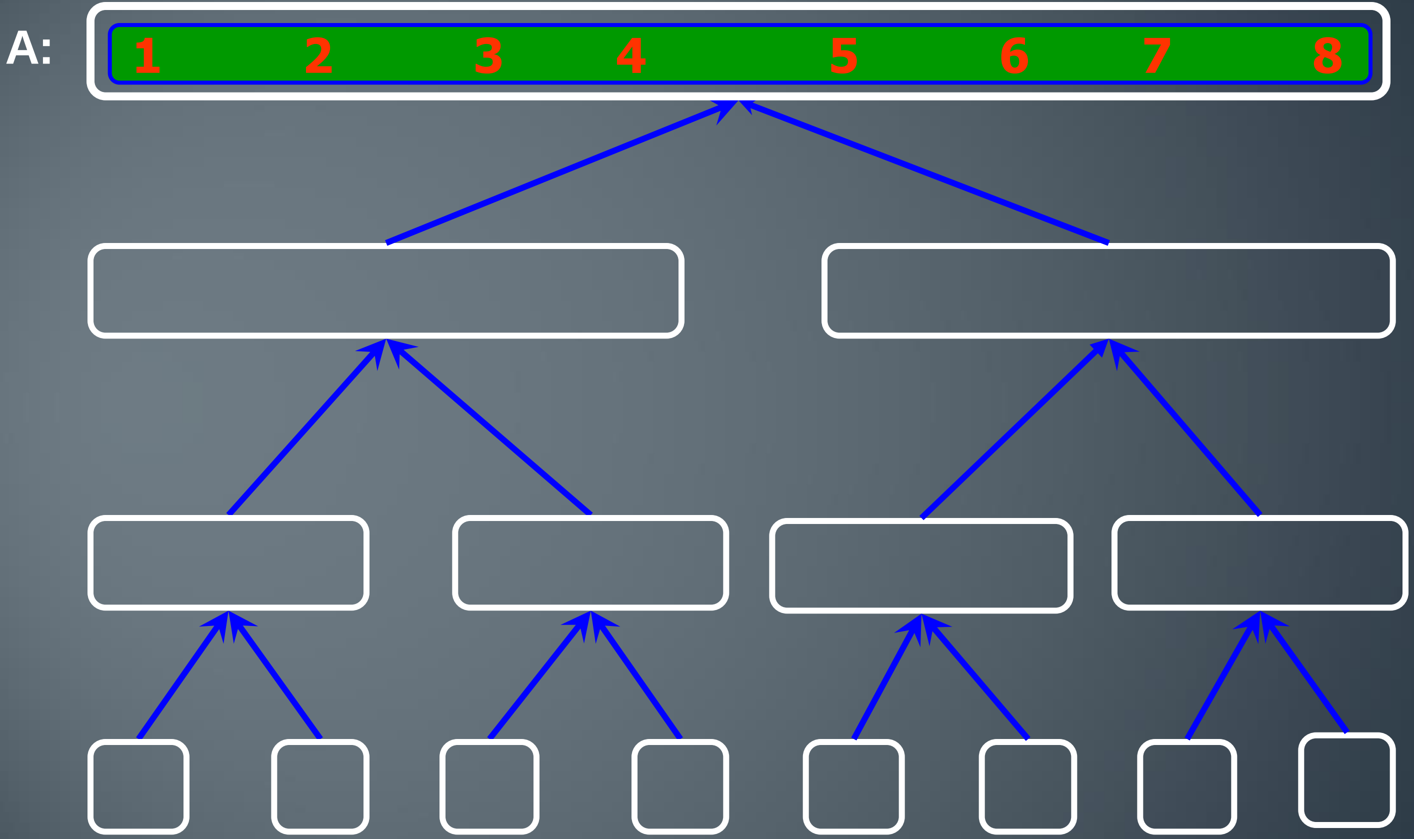
Merge-Sort(A, 0, 7)

Merge-Sort(A, 4, 7), return



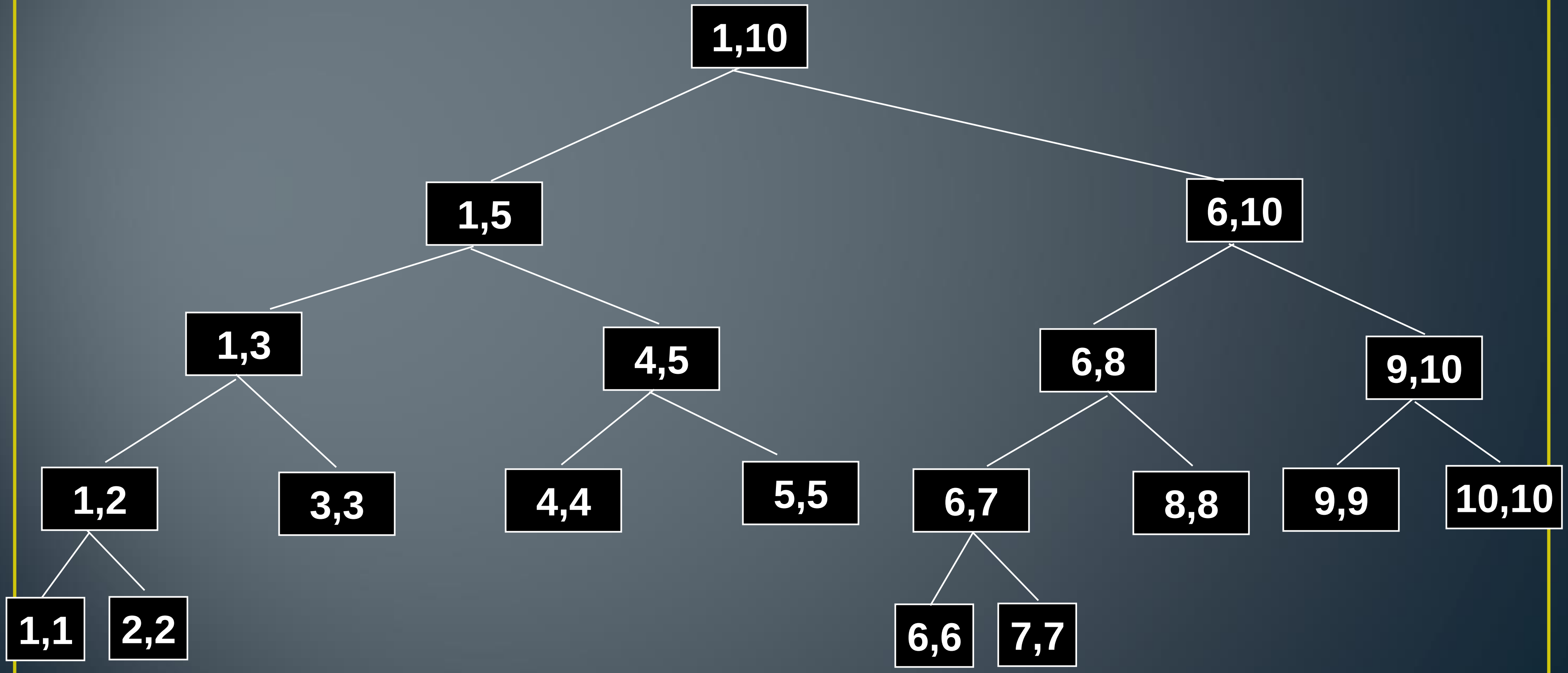
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 7), done!



Ex:- [179, 254, 285, 310, 351, 423, 450, 520, 652,861]

Tree of calls of merge sort



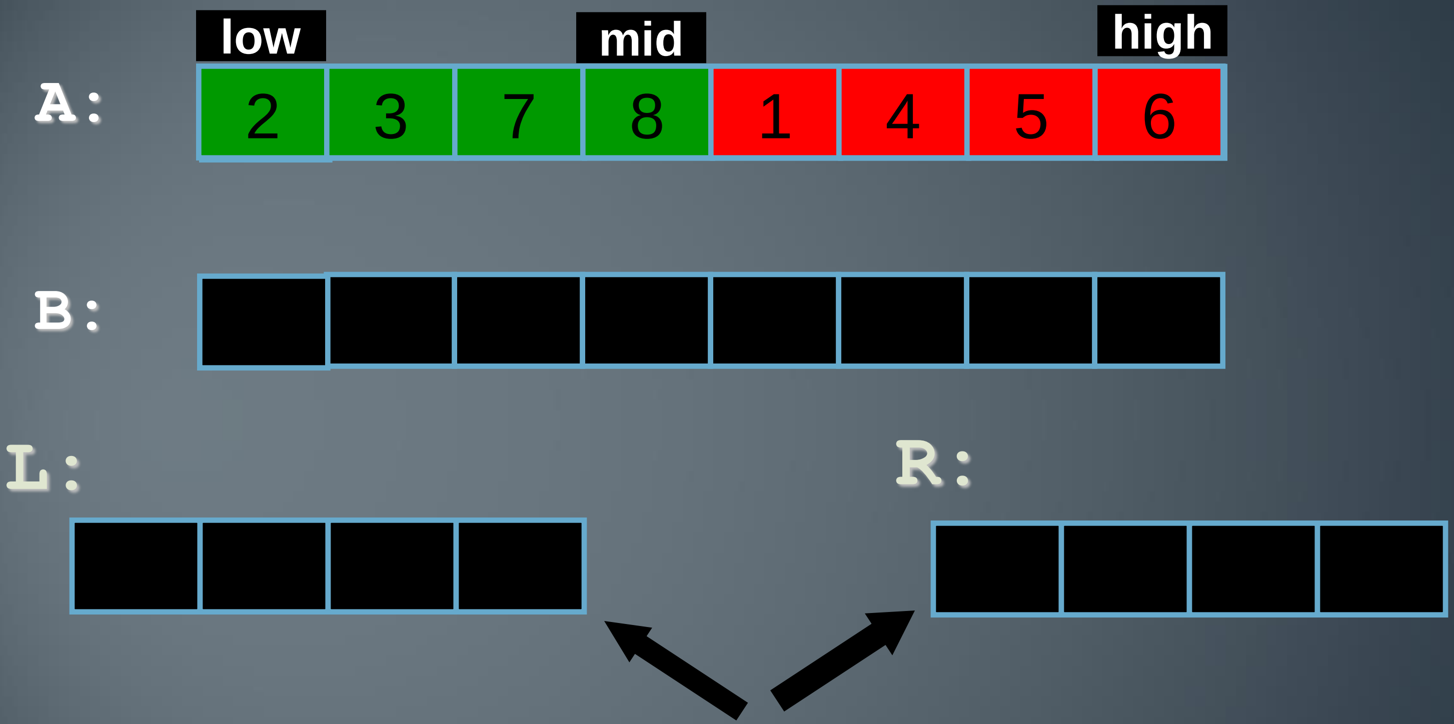
Merge Sort: Algorithm

```
MergeSort ( low,high)
// sorts the elements a[low],…,a[high]  which reside in the global array
//a[1:n] into ascending order.
// Small(p) is true if there is only one element  to sort. In this case the list is
// already sorted.
{   if  (  low<high  ) then  //  if there are more than one element
    {     mid ← (low+high)/2;
          MergeSort(low,mid);
          MergeSort(mid+1, high);
          Merge(low, mid, high);
    }
}
```

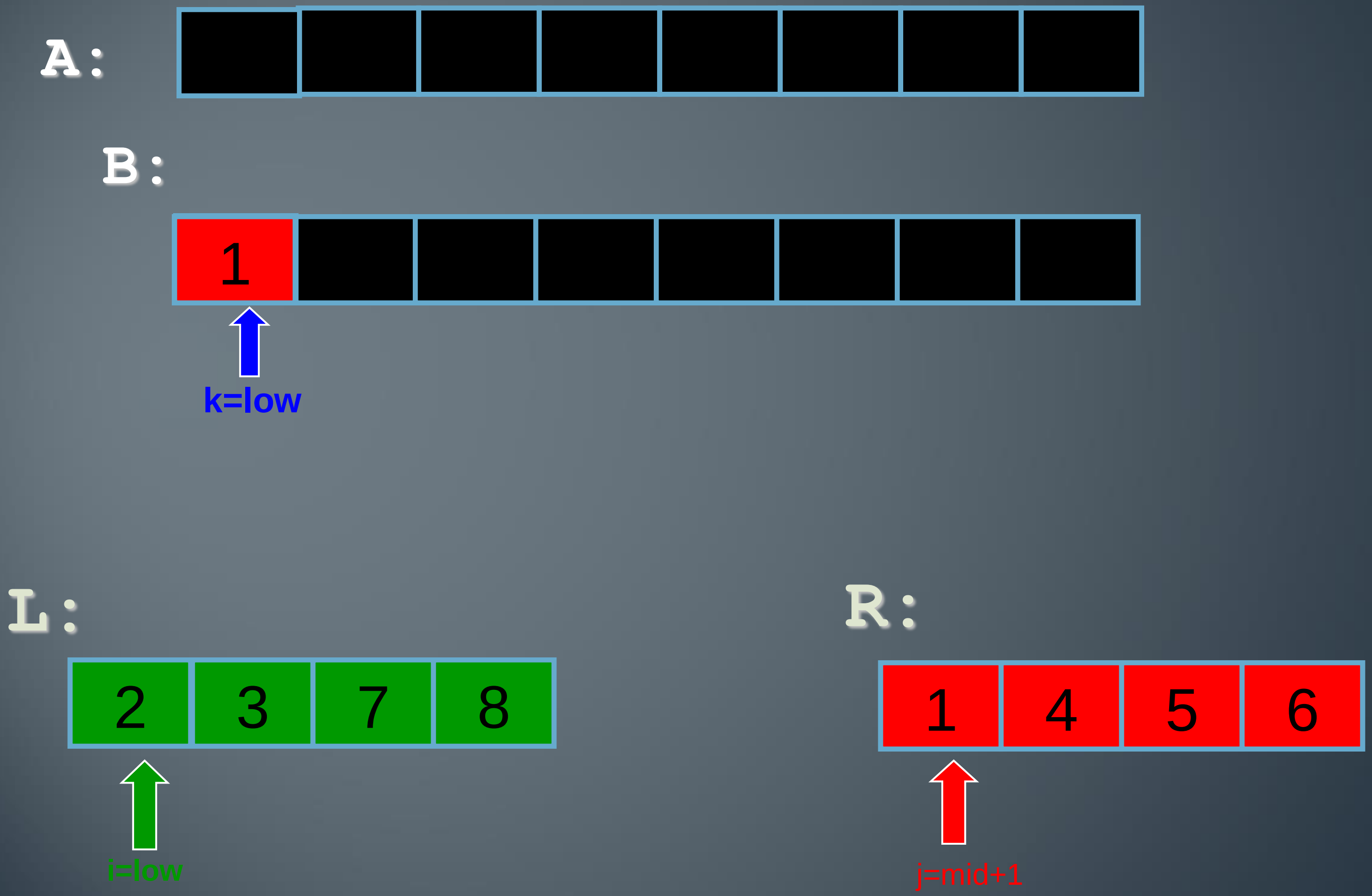
Recursive Calls

```
Algorithm Merge(low,mid,high)
// a[low:high] is a global array containing two sorted subsets in a[low:mid]
// and  in a[mid+1:high]. The goal is to merge these two sets into a single set
//  residing  in a  [low:high]. b[ ] is a temporary global array.
{
    k:=low; i:=low; j:=mid+1;
            while( i ≤ mid ) and ( j ≤ high ) do
            {
                        if( a[i] ≤ a[j] ) then
                        {
                                    b[k]:=a[i]; i:=i+1;
                        }
                        else
                        {
                                    b[k]:=a[j]; j:=j+1;
                        }
                        k:=k+1;
            }
            while(i<=mid)
            {
               b[k] := a[i];
               i := i+1;
               k := k+1
            }
          while(j<=high)
            {
               b[k] := a[j];
                j := j+1;
                k := k+1
            }
       for(i=low; i<=high; i++)
            a[i]=b[i];
}
```

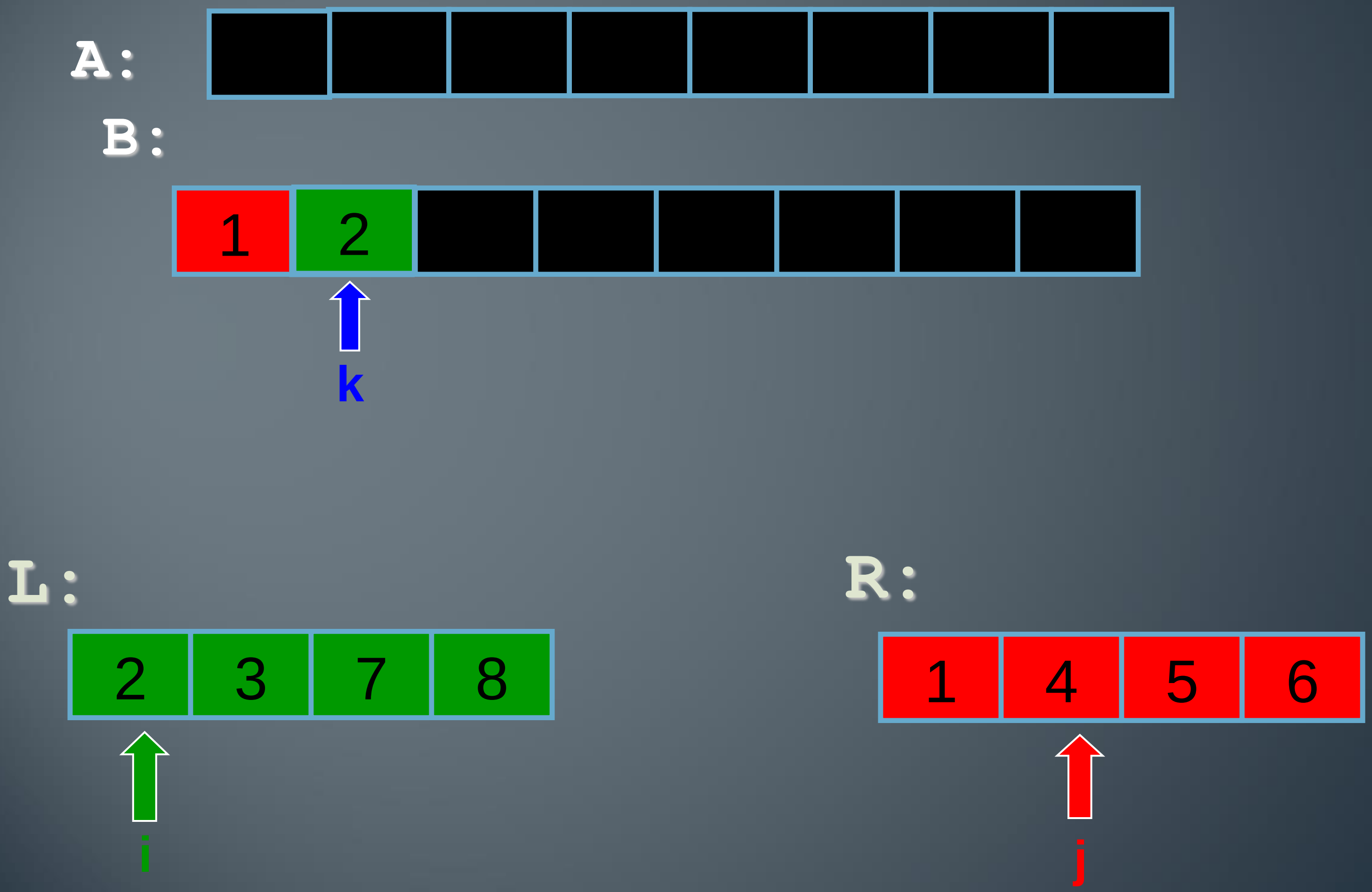
Merge-Sort: Merge Example



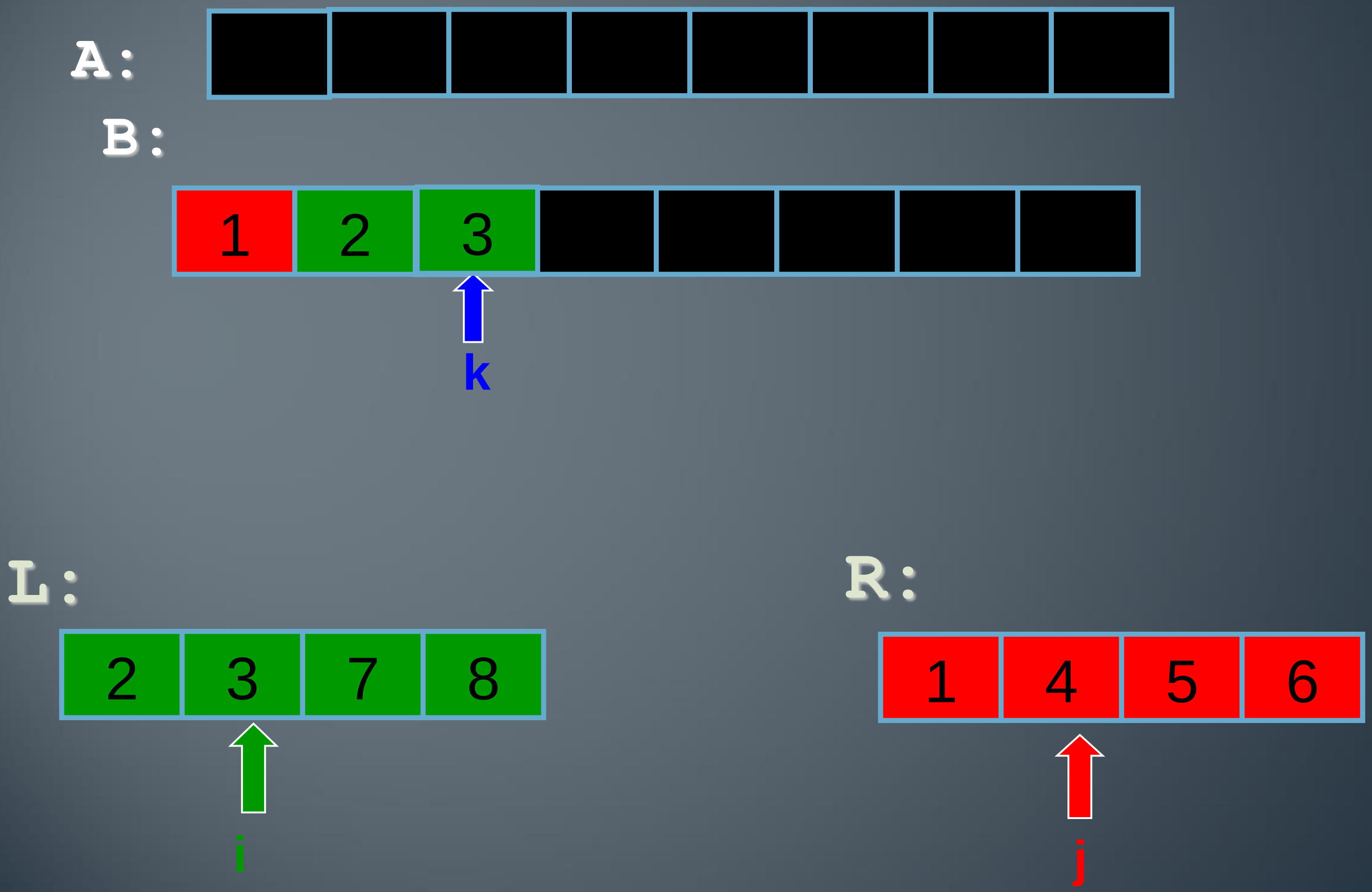
Merge-Sort: Merge Example



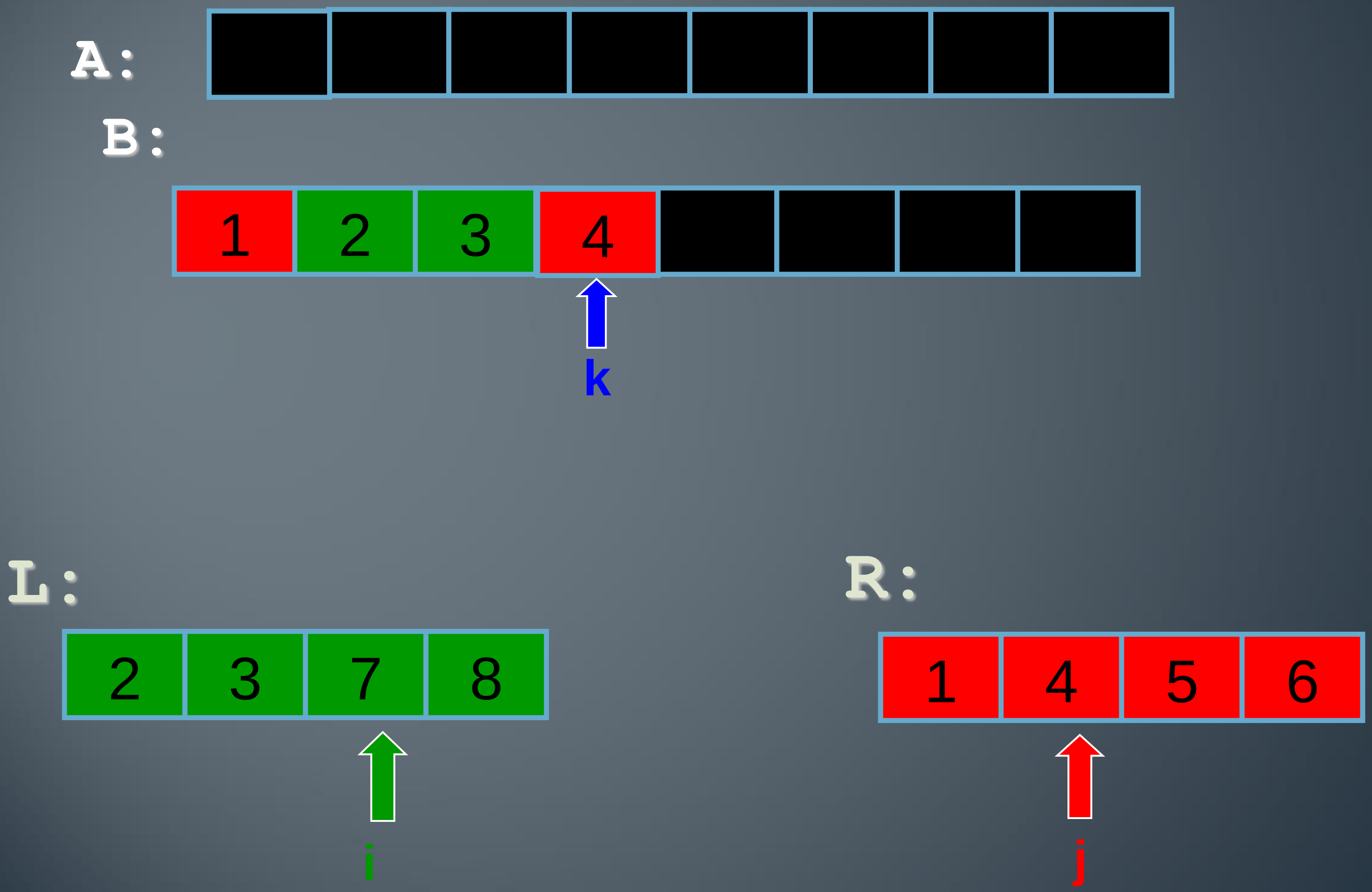
Merge-Sort: Merge Example



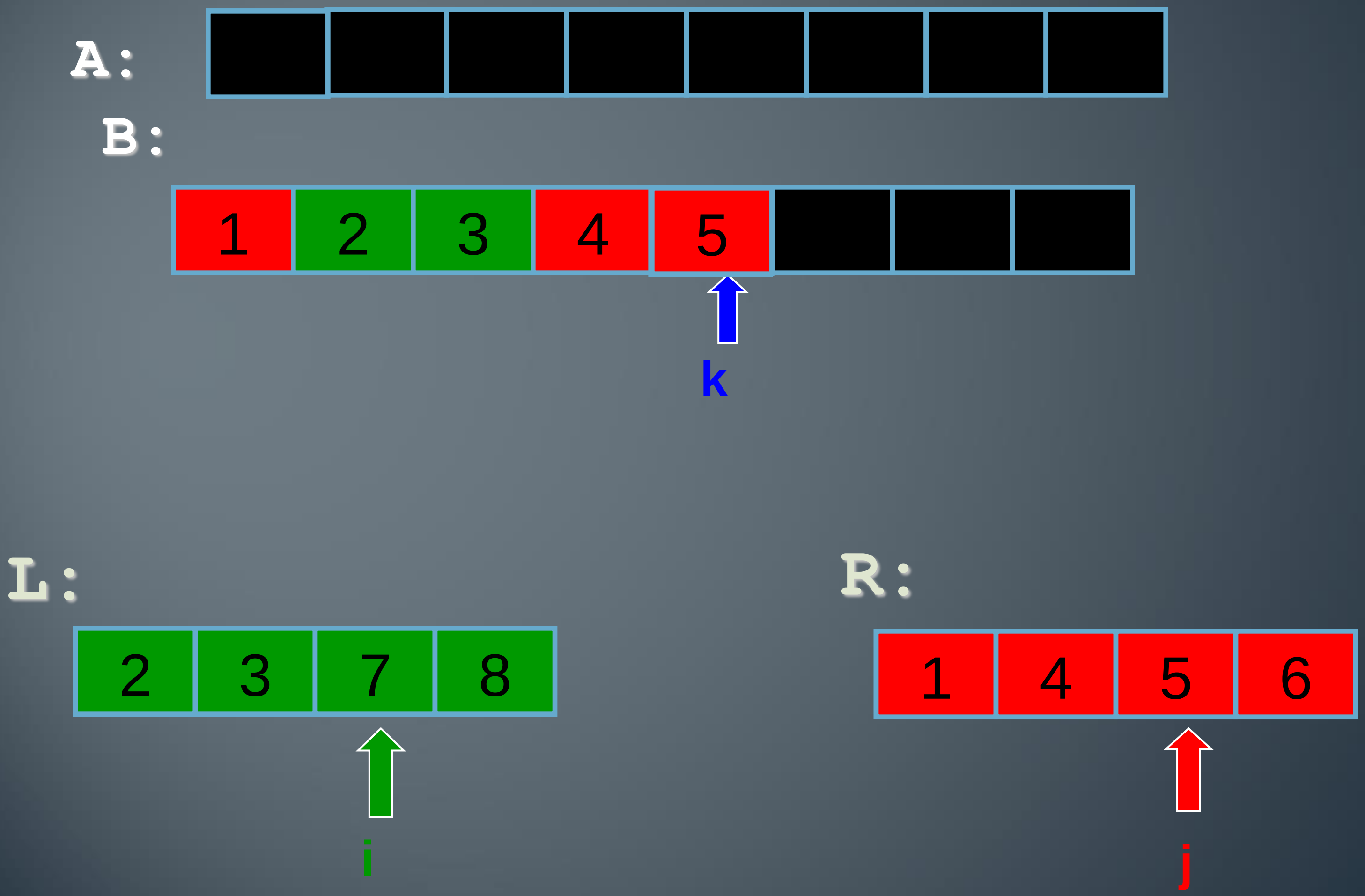
Merge-Sort: Merge Example



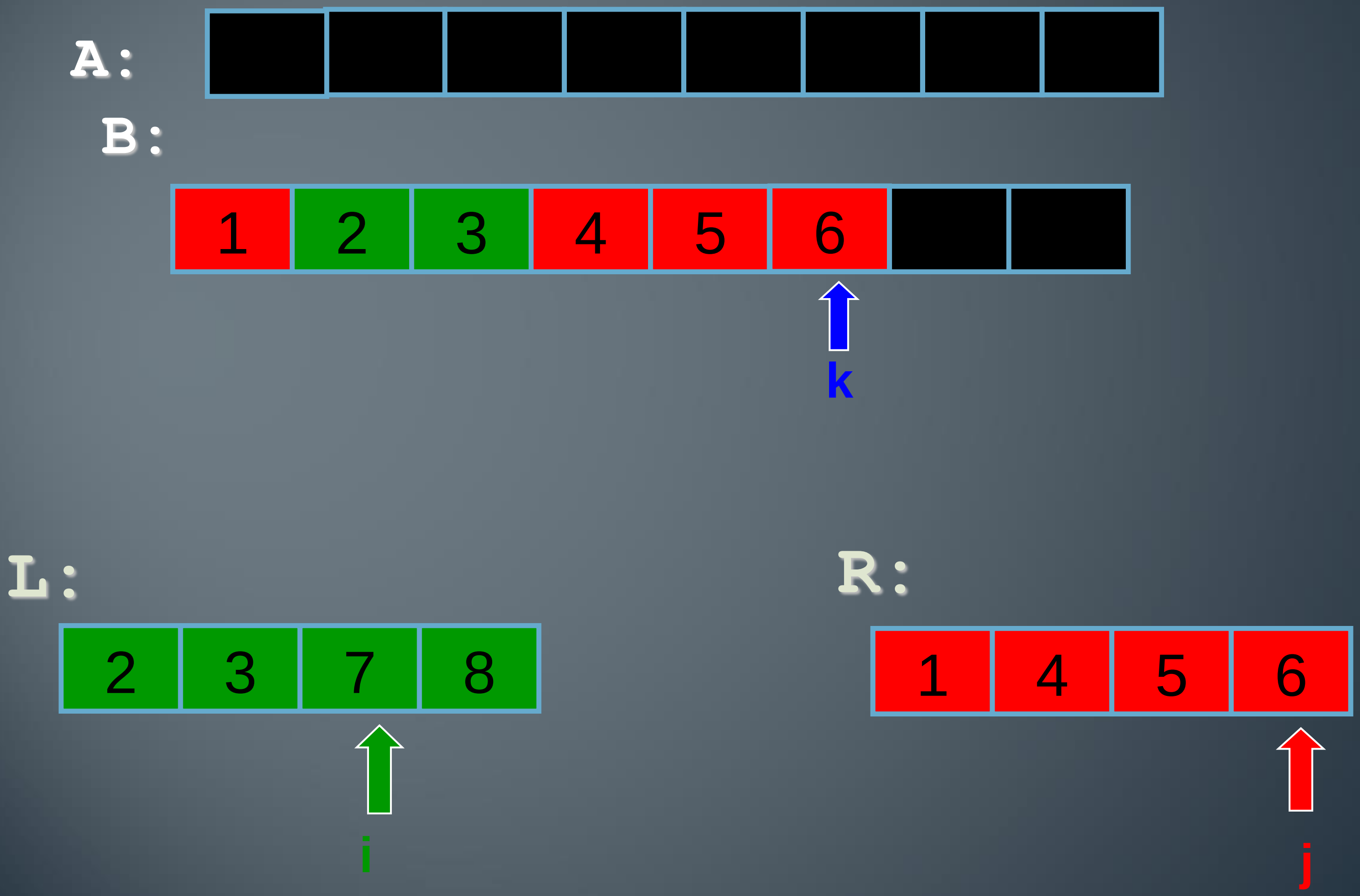
Merge-Sort: Merge Example



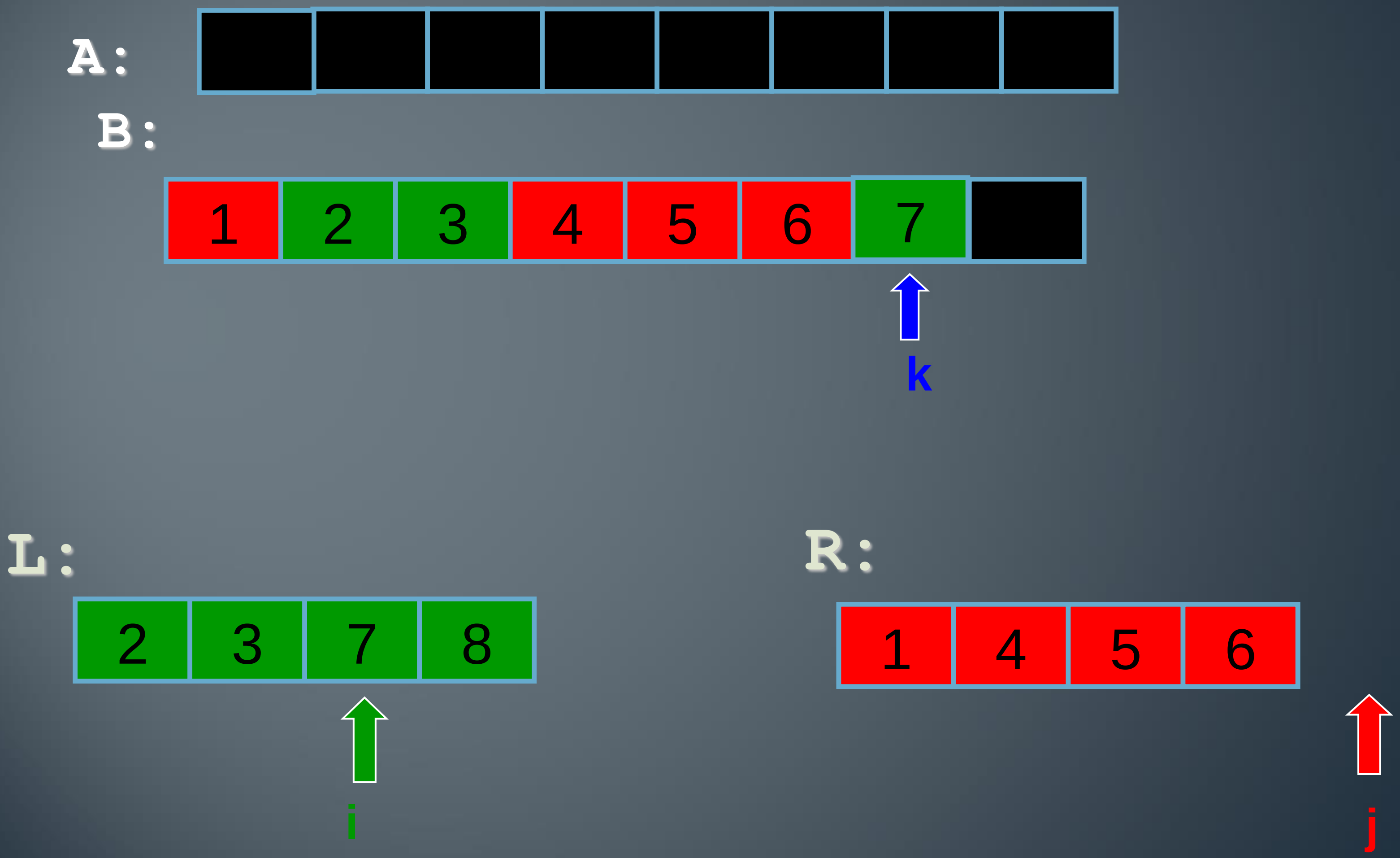
Merge-Sort: Merge Example



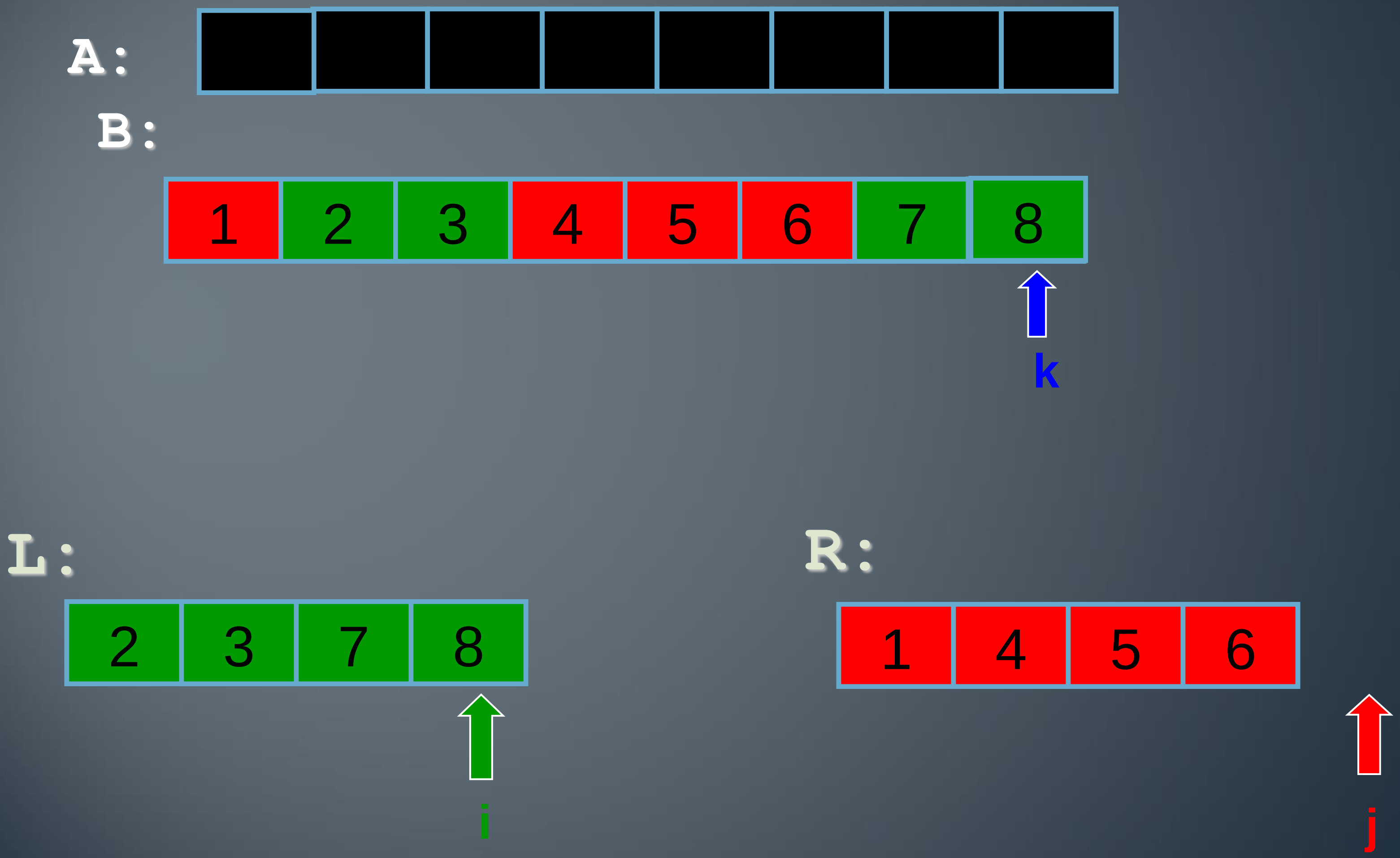
Merge-Sort: Merge Example



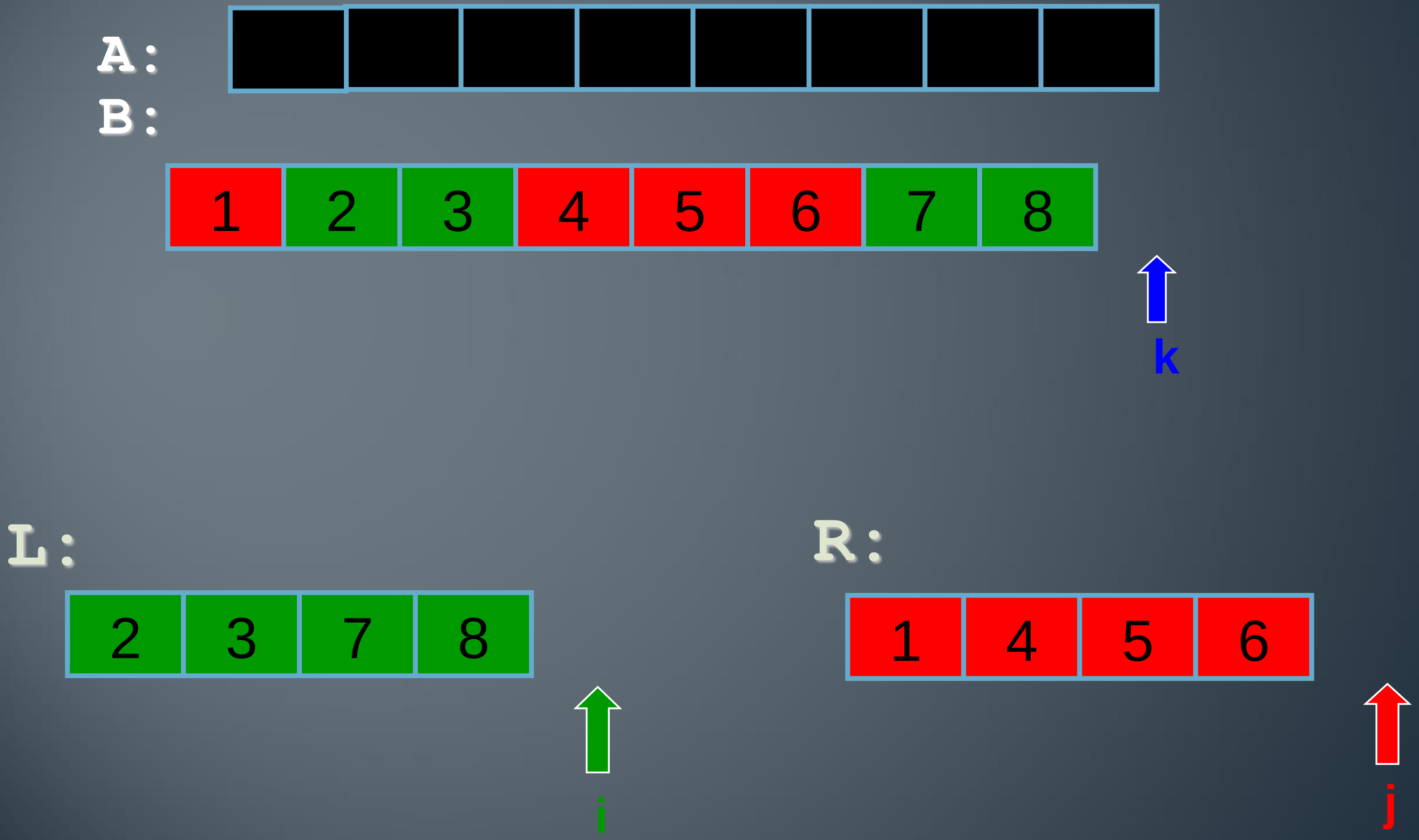
Merge-Sort: Merge Example

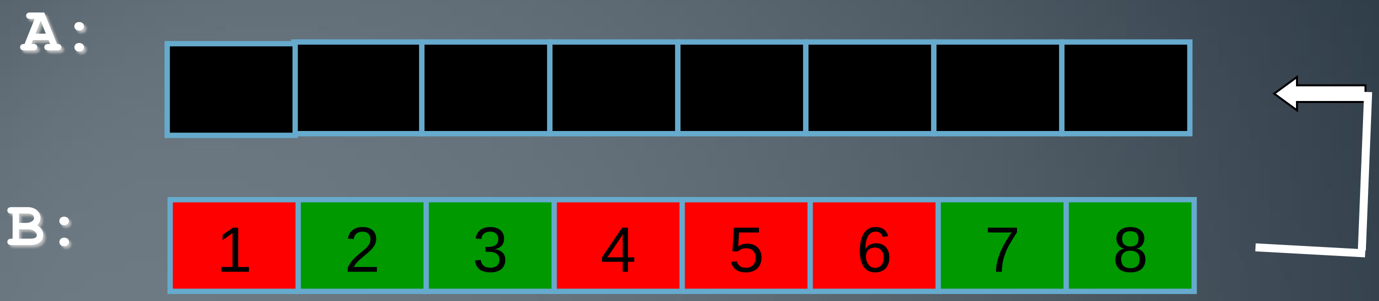


Merge-Sort: Merge Example

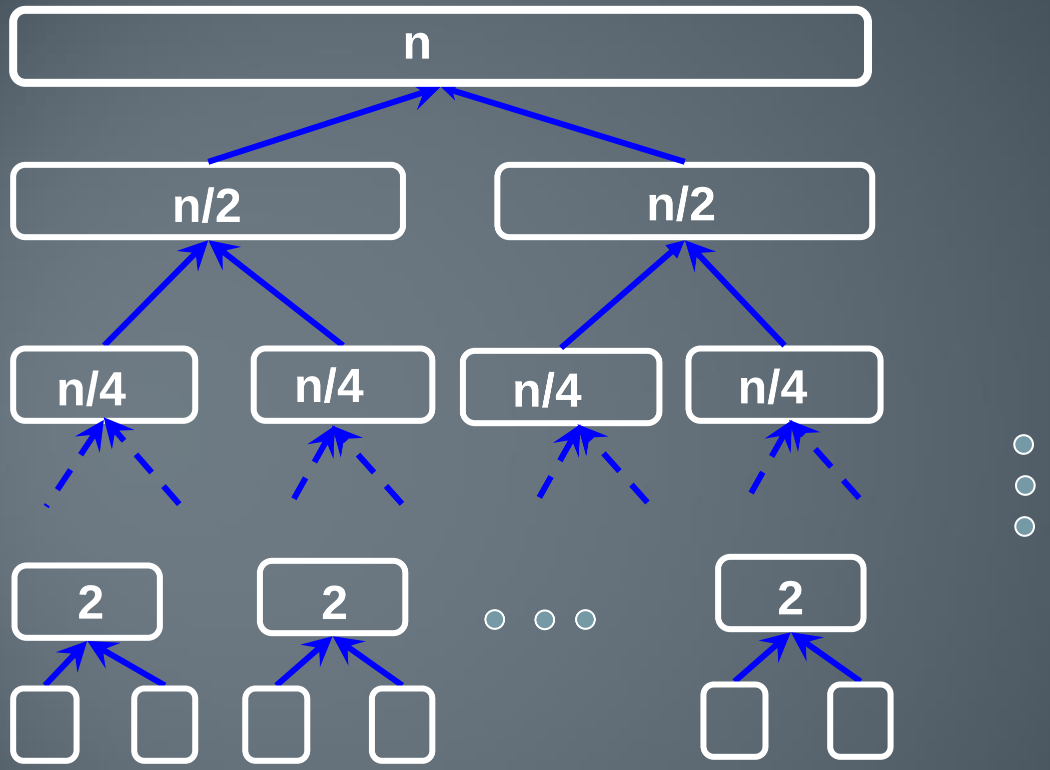


Merge-Sort: Merge Example




A:
B:
1
2
3
4
5
6
7
8

Merge-Sort Analysis using Recursion tree method



Merge-Sort Time Complexity

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} c_1 & n=1,\ c_1 \text{ is a constant} \\ 2T(n/2) + c_2 n & n>1,\ c_2 \text{ is a constant} \end{cases}$$

Assume $n=2^k$, then

$T(n) = 2T(n/2) + c_2 n$

$= 2(2T(n/4)+c2n/2)+c2n$

$= 4T(n/4)+2c_2 n$

.....

.....

$= 2^k\, T(1) + kc_2 n$

$= c_1 n + c_2 n\log n = = O(n\log n)$

Summary

- Merge-Sort
  - Most of the work done in combining the solutions.
  - Best case takes **O(n log(n))** time
  - Average case takes **O(n log(n))** time
  - Worst case takes **O(n log(n))** time

3. QUICK SORT

- Quick sort is a divide and conquer technique of sorting elements.
- Quick sort was invented by Sir Charles Antony Richard Hoare in 1960.
- Quick sort does the following:
  - Select and place the pivot in its place
  - Recursively sort both the sides of the pivot.

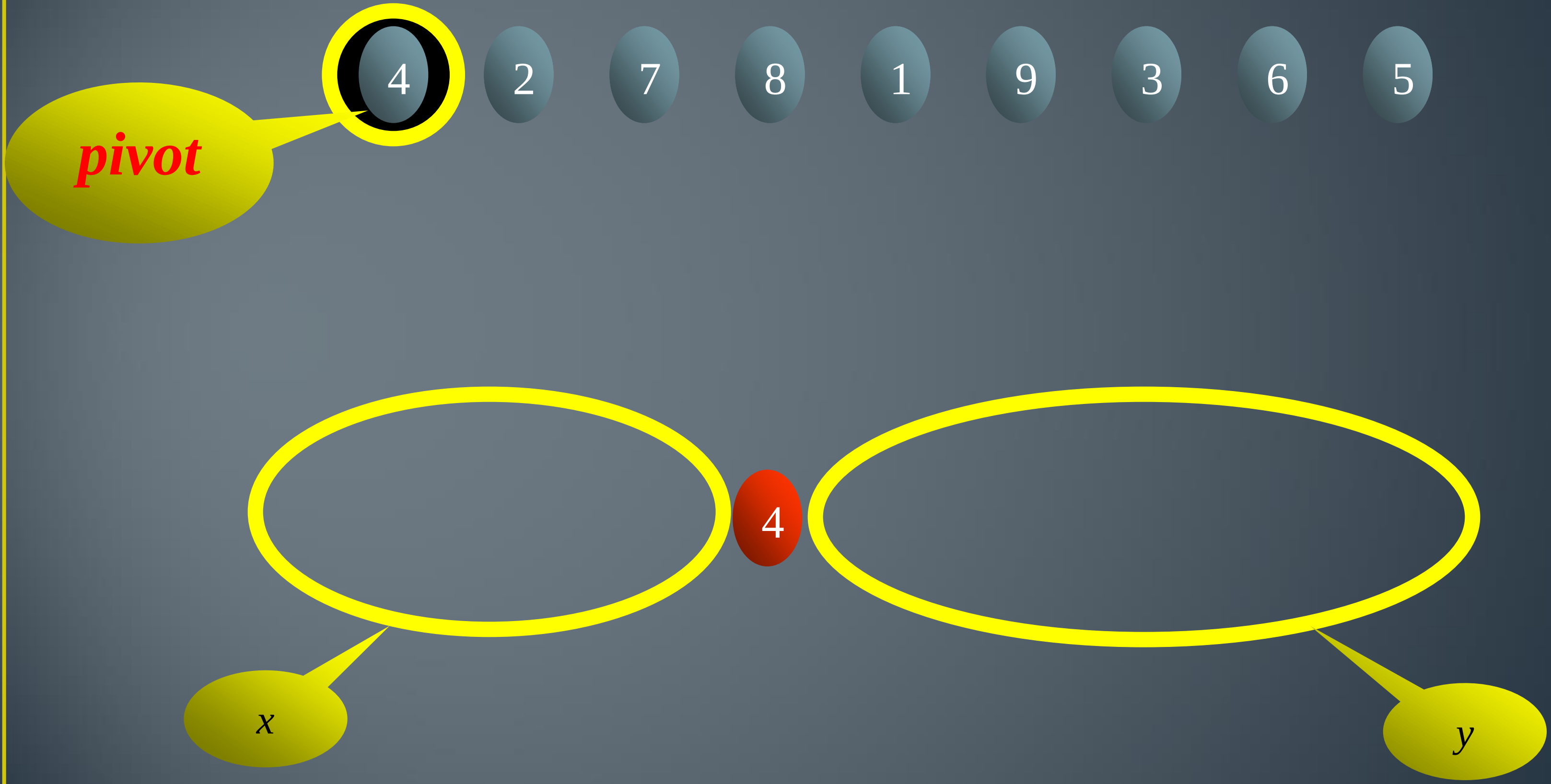
- **Divide:**
    - Pick any element as the **pivot**, e.g, the first element
    - Partition the remaining elements into

      FirstPart, which contains all elements < pivot

      SecondPart, which contains all elements > pivot

- **Recursively sort** FirstPart and SecondPart.
- **Combine:** no work is necessary since sorting is done in place.

pivot divides a into two sublists x and y.

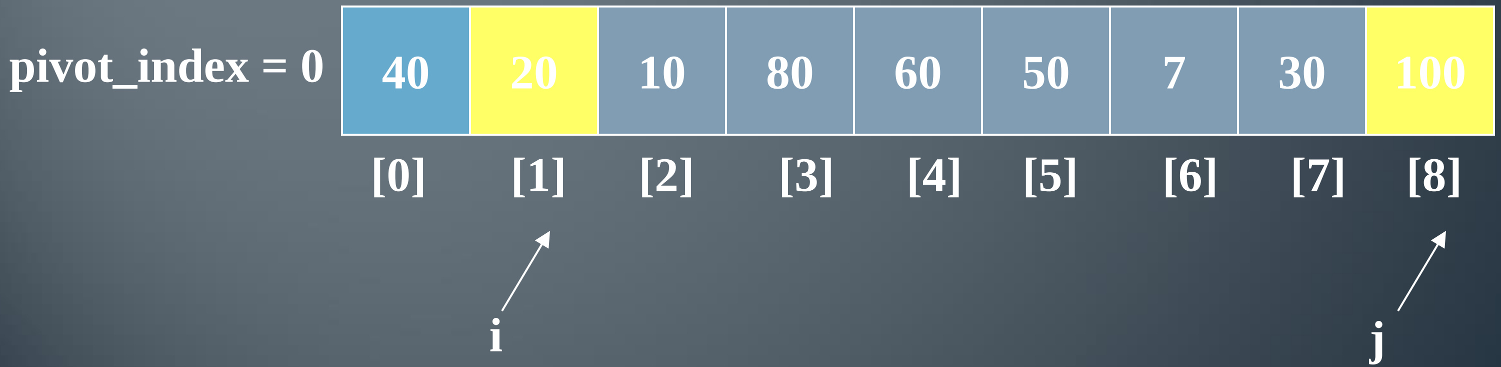


Quicksort

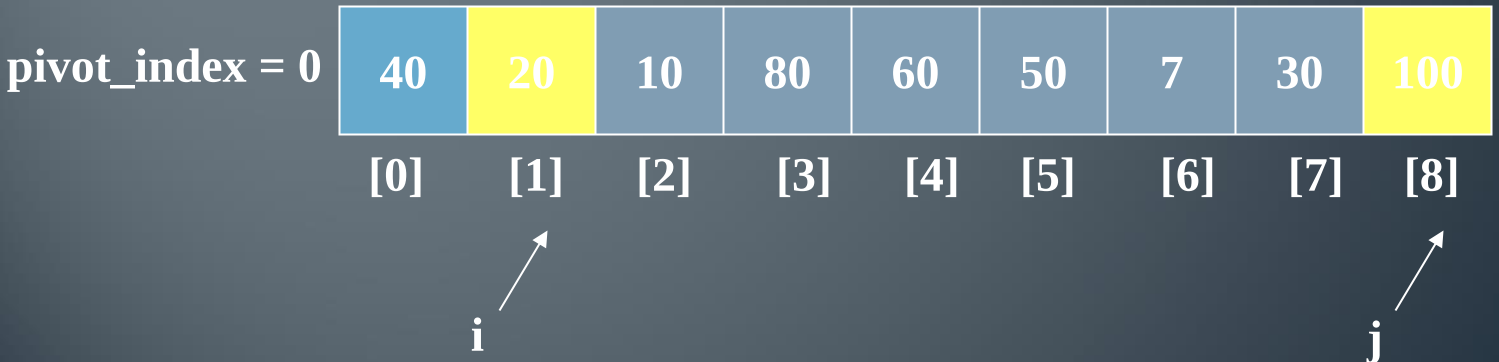
- In this method, an element called pivot is identified and that element is fixed in its place by moving all the elements less than that to its left and all the elements greater than that to its right.

Procedure:

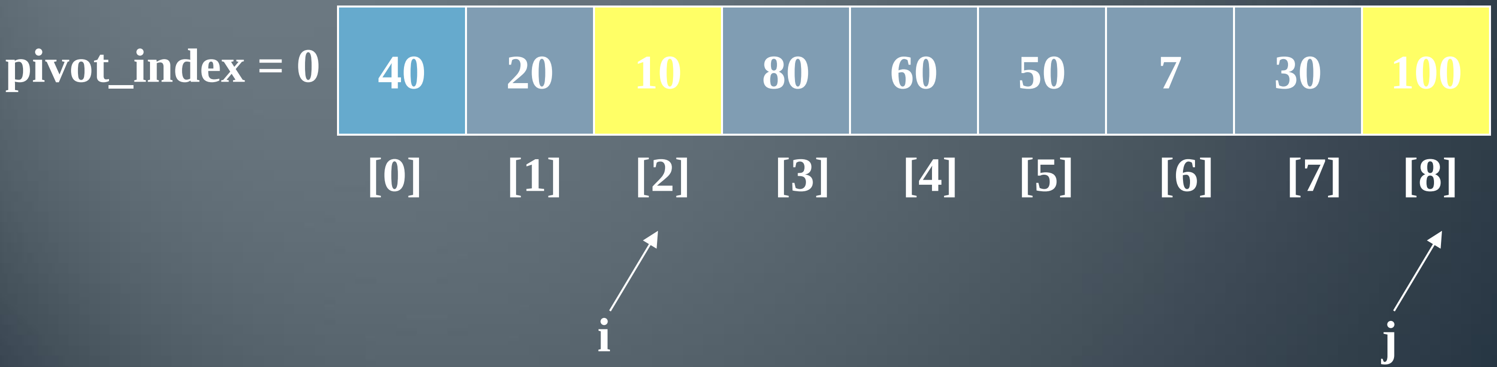
1. If array only contains one element, return it.
2. Else
    1. Pick one element to use **as pivot.**
    2. Partition elements into **two sub-arrays**
        1. Elements **less than or equal** to pivot
        2. Elements **greater than** pivot
    3. **Quicksort two sub-arrays.**
    4. Return results.


pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
i
j

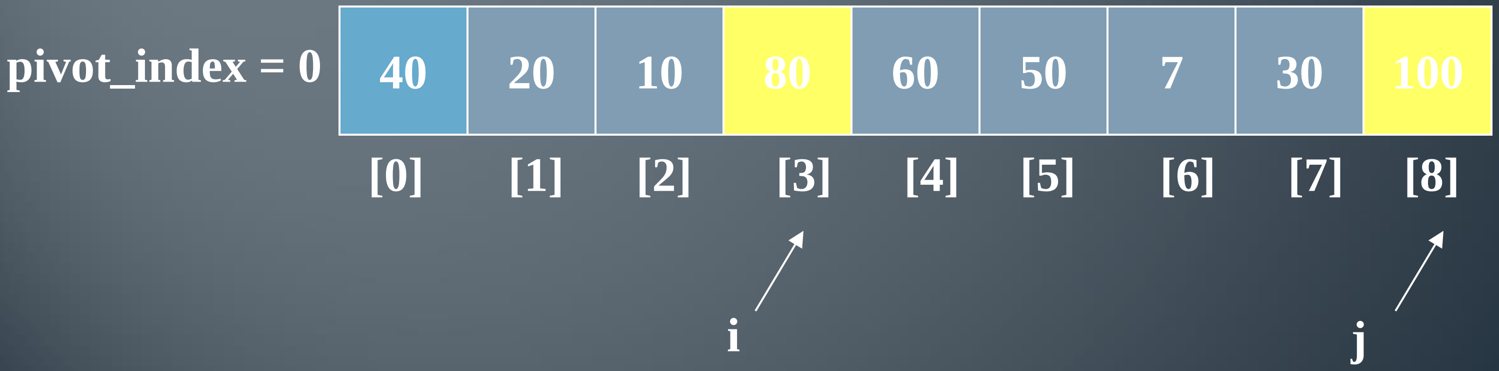
1. **While data[i] <= data[pivot]**
   ++i



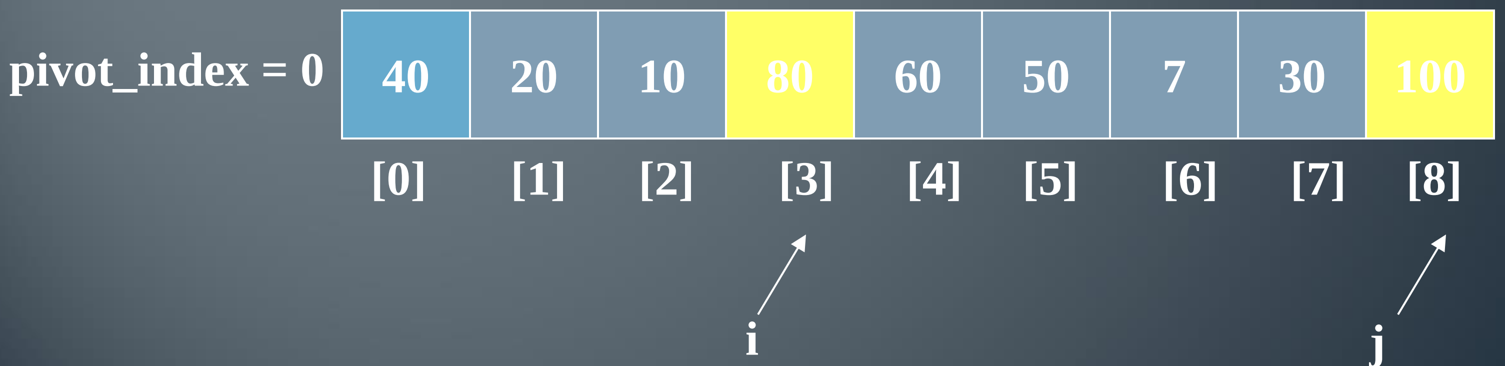
1. **While data[i] <= data[pivot]**
   ++i



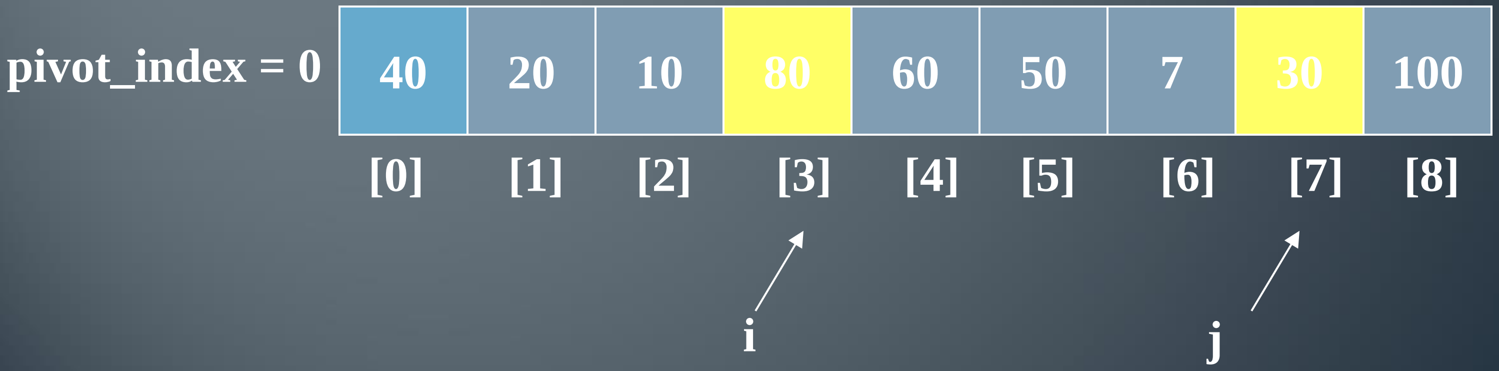
1. While data[i] <= data[pivot]
      ++i



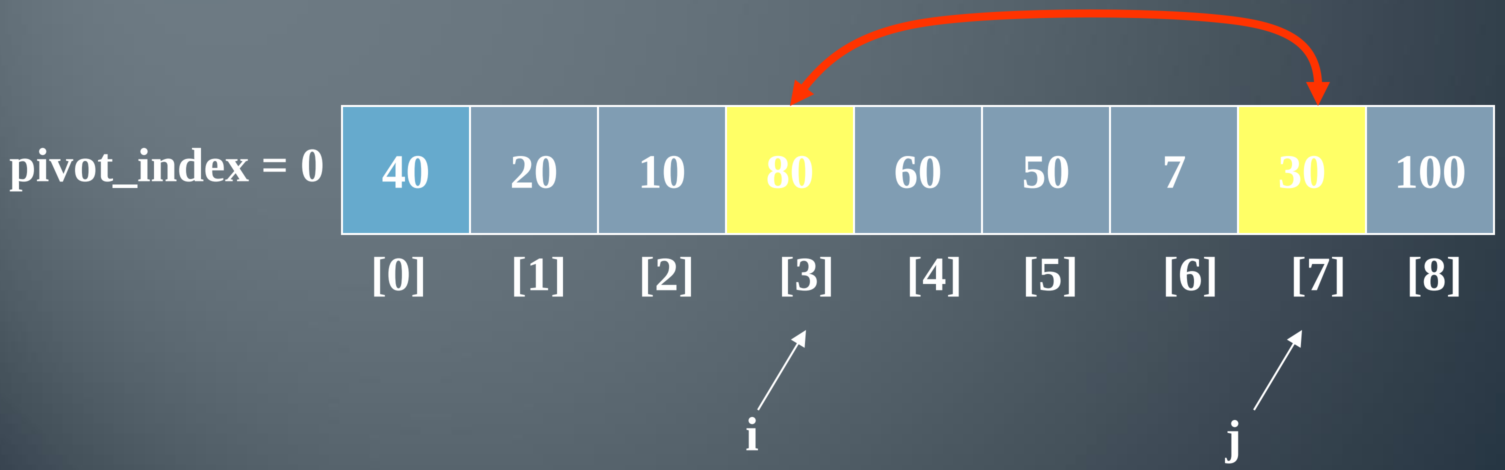
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
```



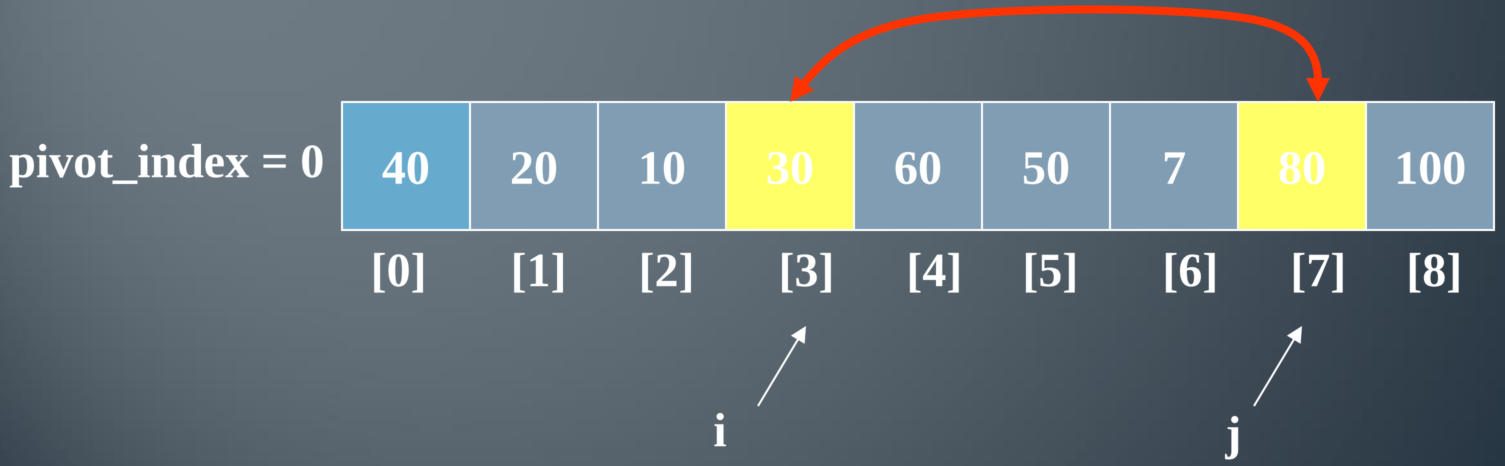
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
```



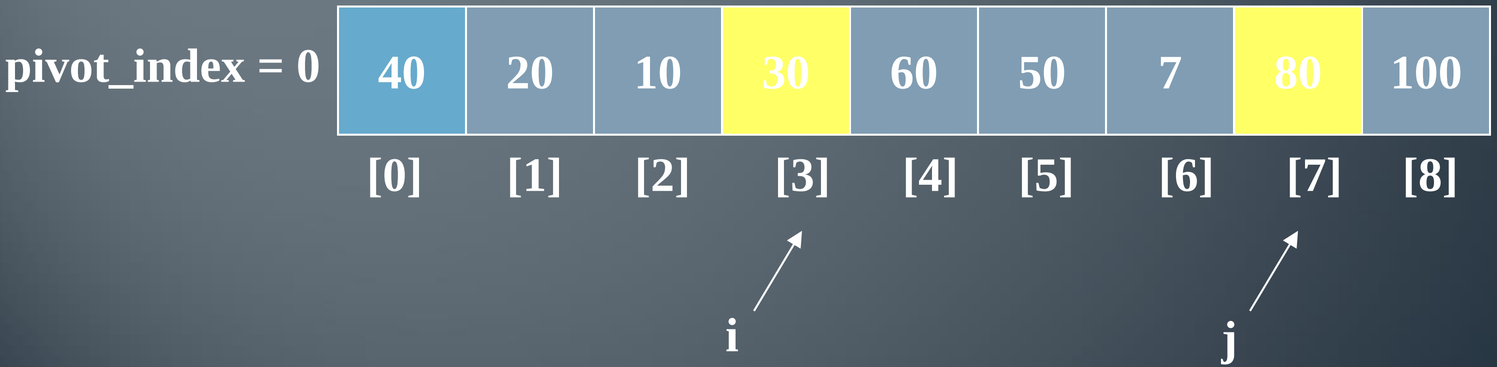
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
```



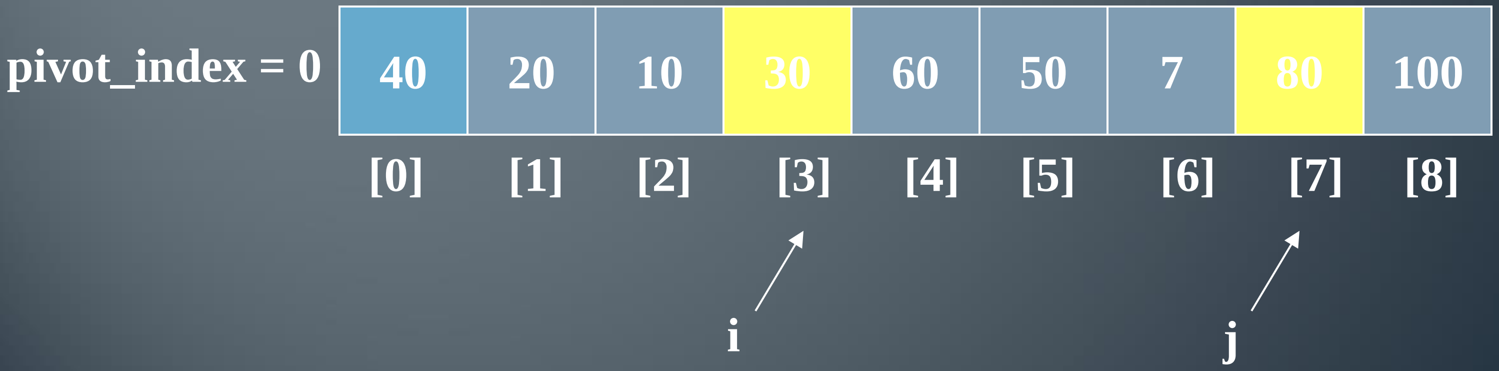
```
1. While data[i] <= data[pivot]
           ++i
2. While data[j] > data[pivot]
           --j
3. If i < j
           swap data[i] and data[j]
```



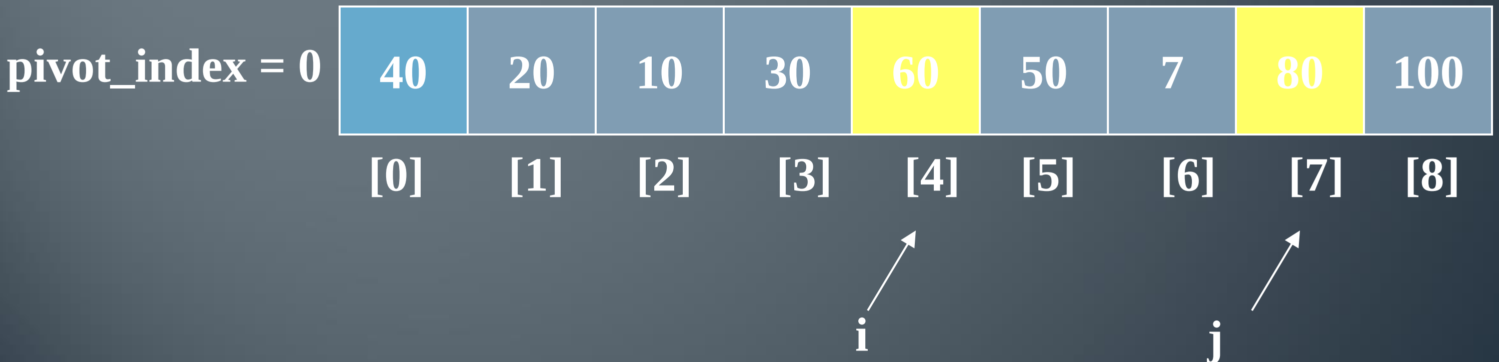
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



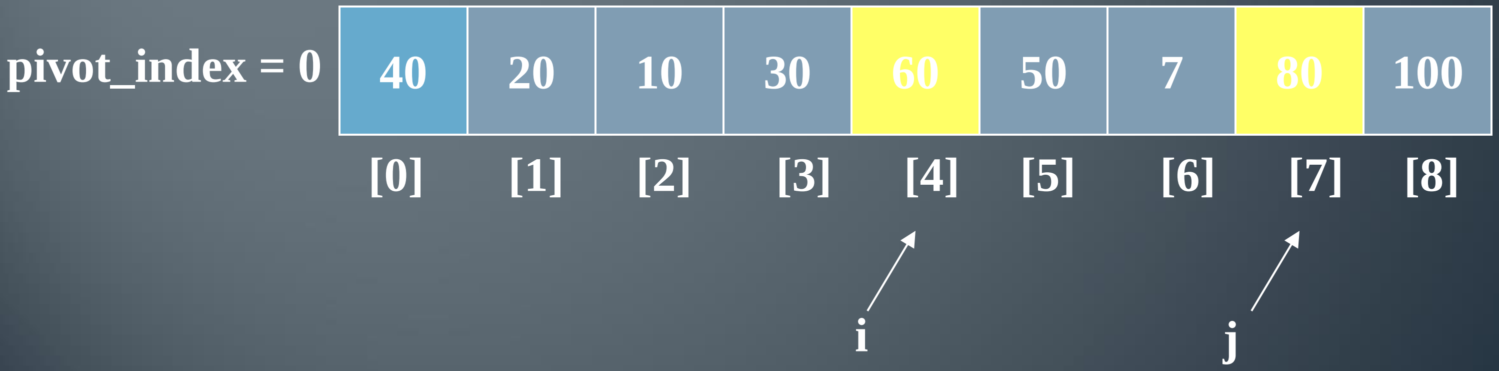
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



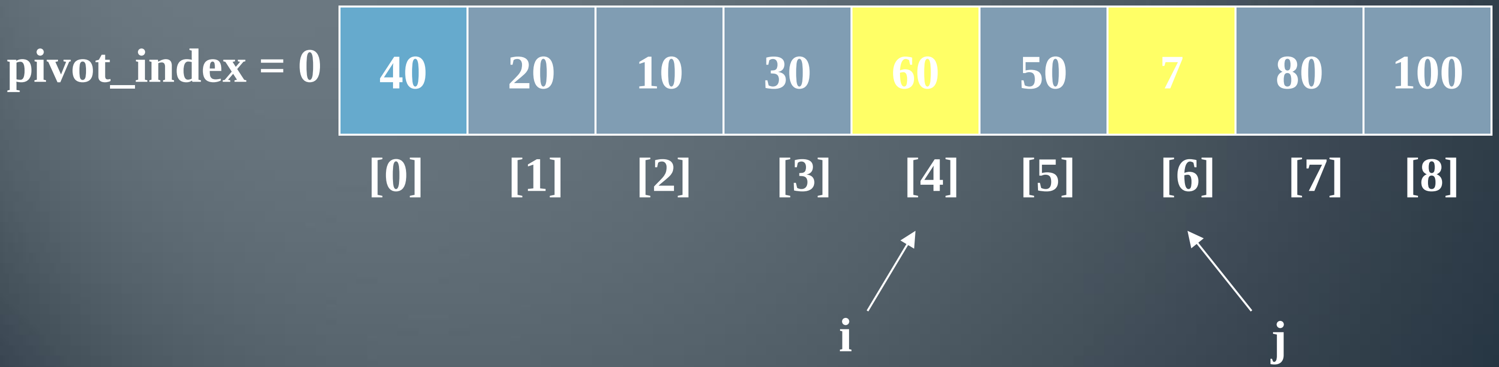
```
1.   While data[i] <= data[pivot]
         ++i
2.   While data[j] > data[pivot]
         --j
3.   If i < j
         swap data[i] and data[j]
4.   While j > i, go to 1.
```



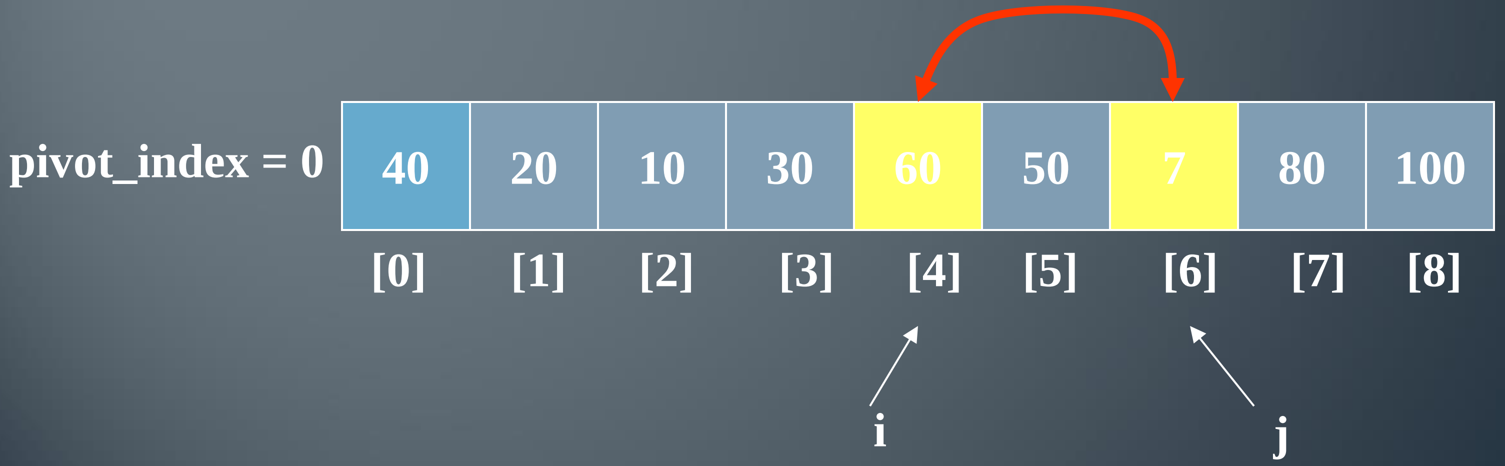
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



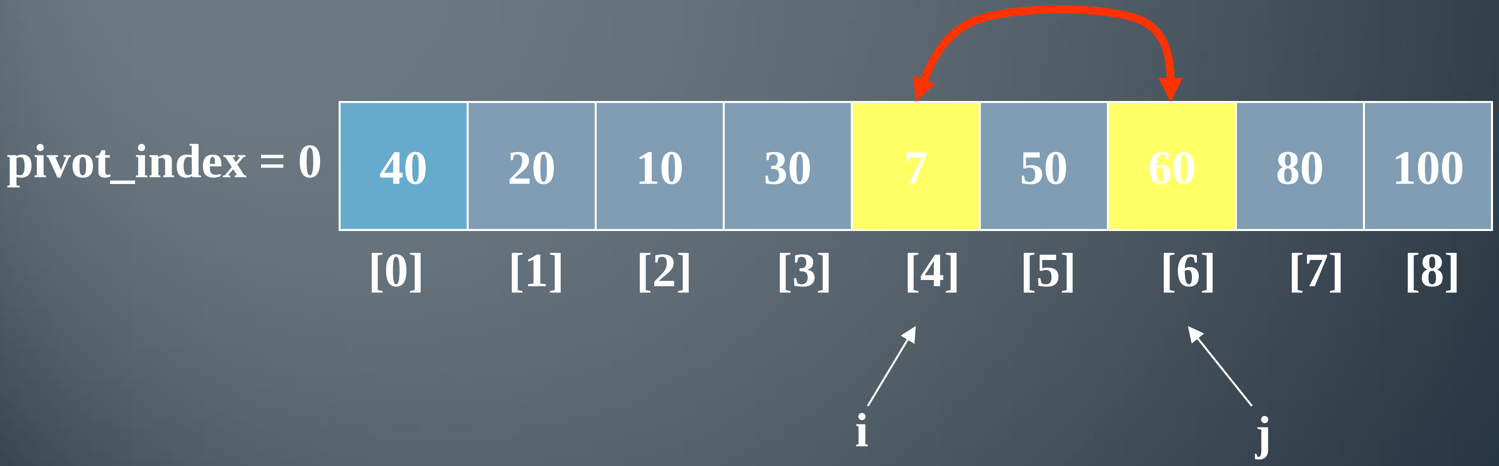
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



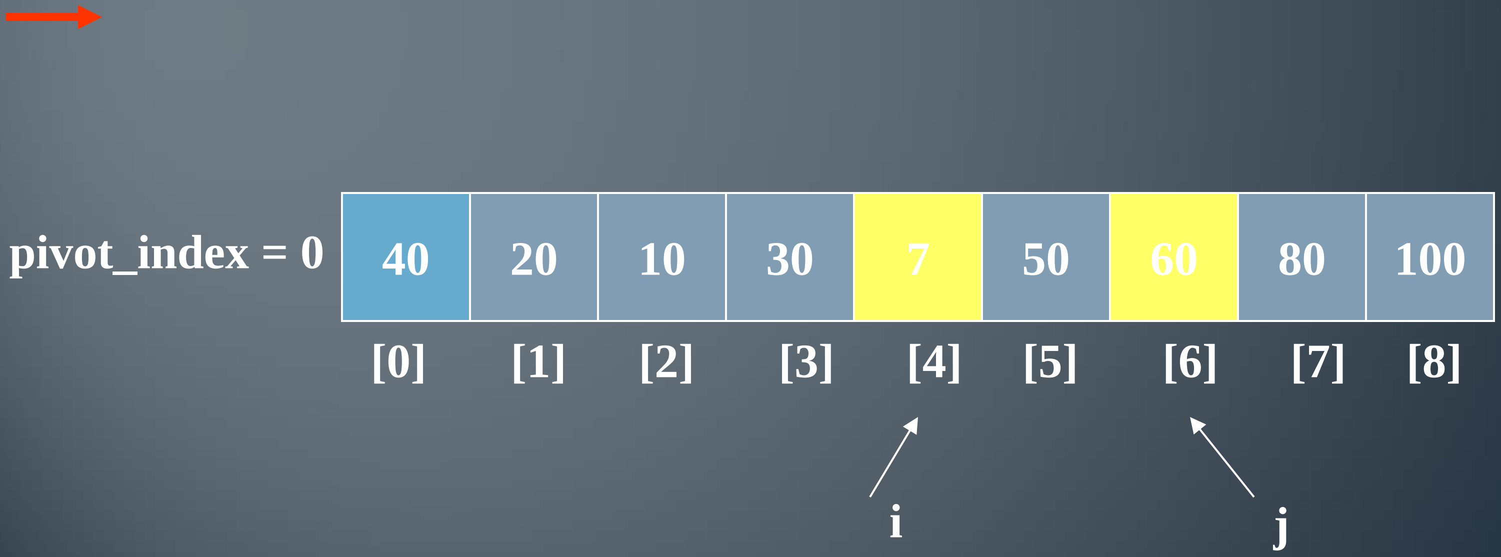
```
1.    While data[i] <= data[pivot]
              ++i
2.    While data[j] > data[pivot]
              --j
3.    If i < j
              swap data[i] and data[j]
4.    While j > i, go to 1.
```



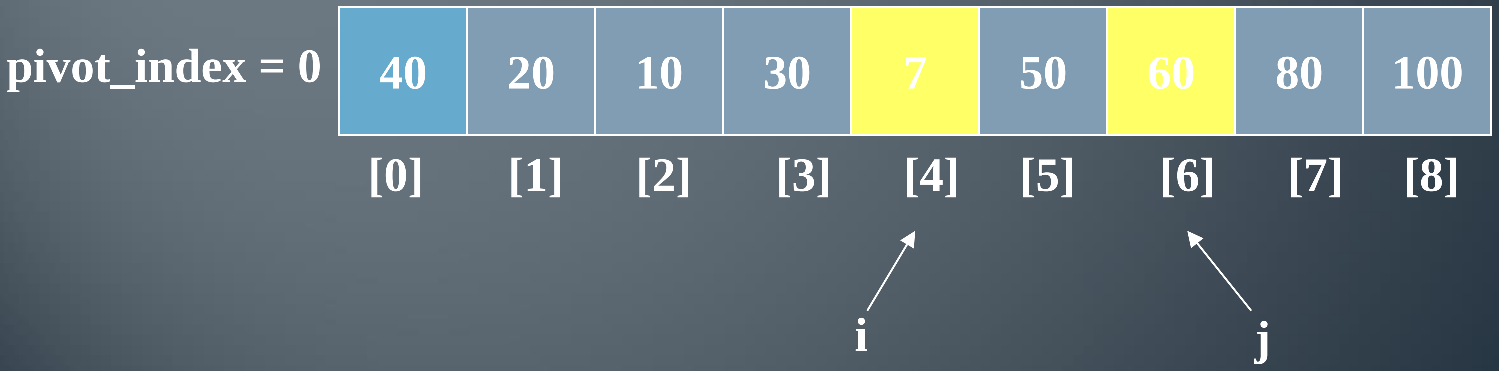
```
1. While data[i] <= data[pivot]
        ++i
2. While data[j] > data[pivot]
        --j
3. If i < j
        swap data[i] and data[j]
4. While j > i, go to 1.
```



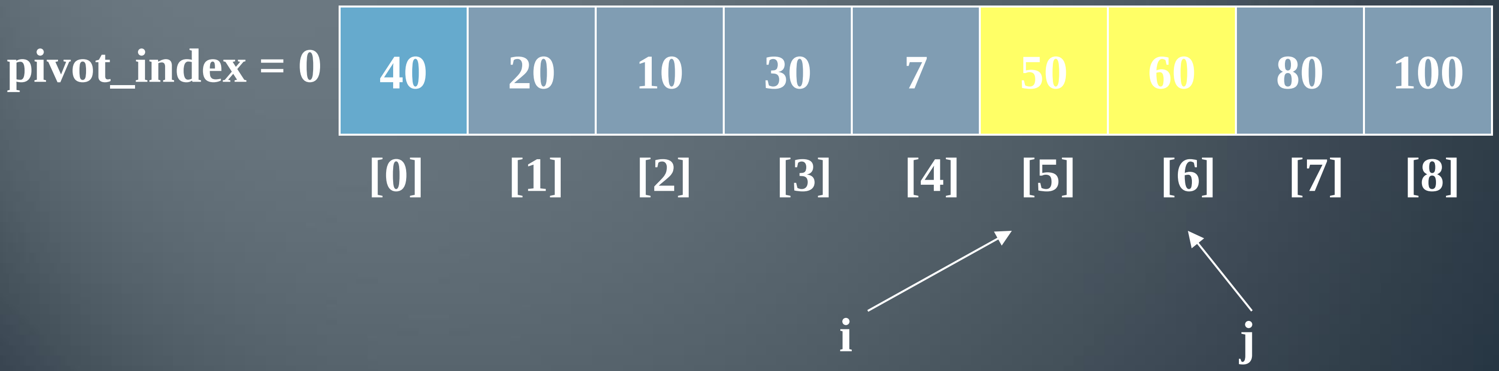
```
1.  While data[i] <= data[pivot]
            ++i
2.  While data[j] > data[pivot]
            --j
3.  If i < j
            swap data[i] and data[j]
4.  While j > i, go to 1.
```



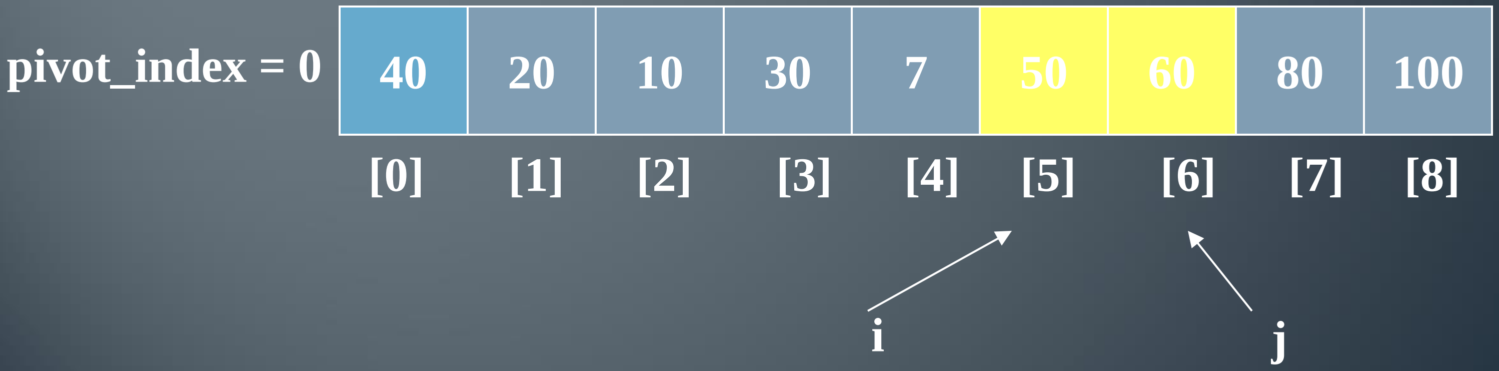
```
1.  While data[i] <= data[pivot]
            ++i
2.  While data[j] > data[pivot]
            --j
3.  If i < j
            swap data[i] and data[j]
4.  While j > i, go to 1.
```



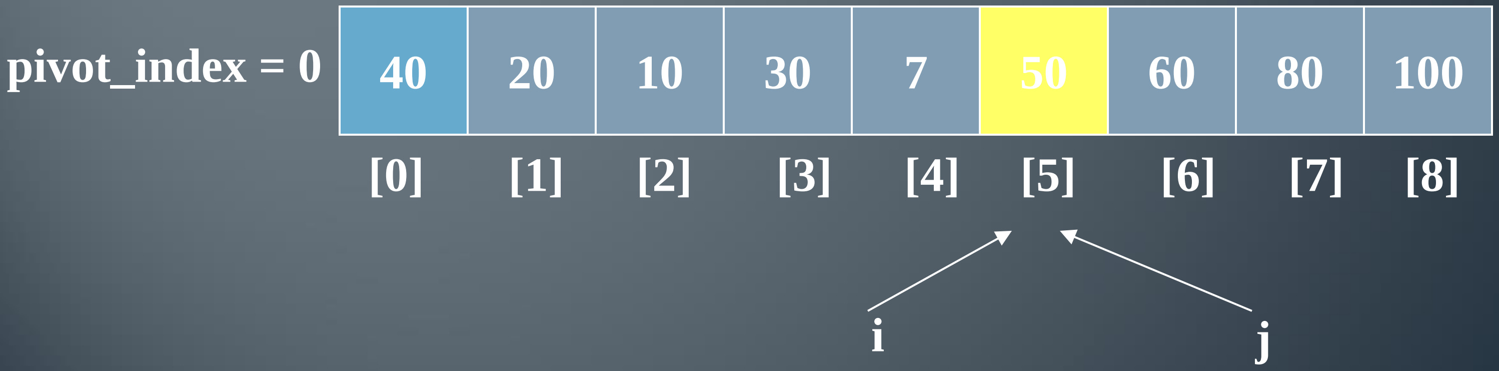
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```

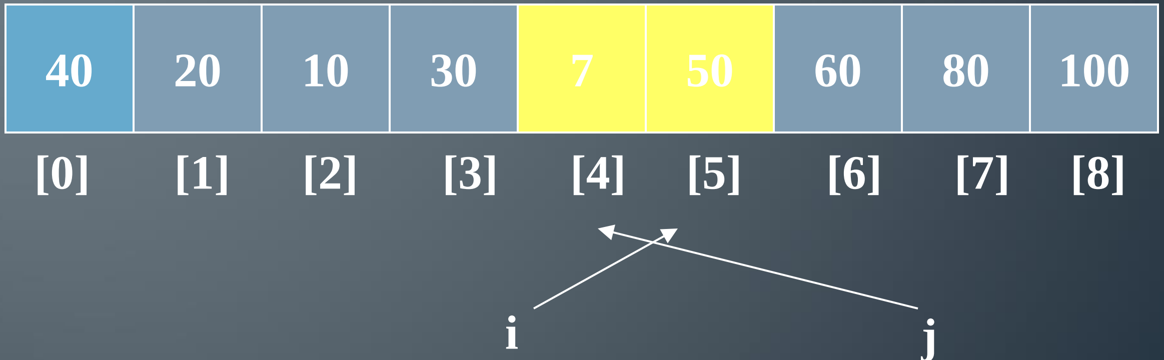


```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```

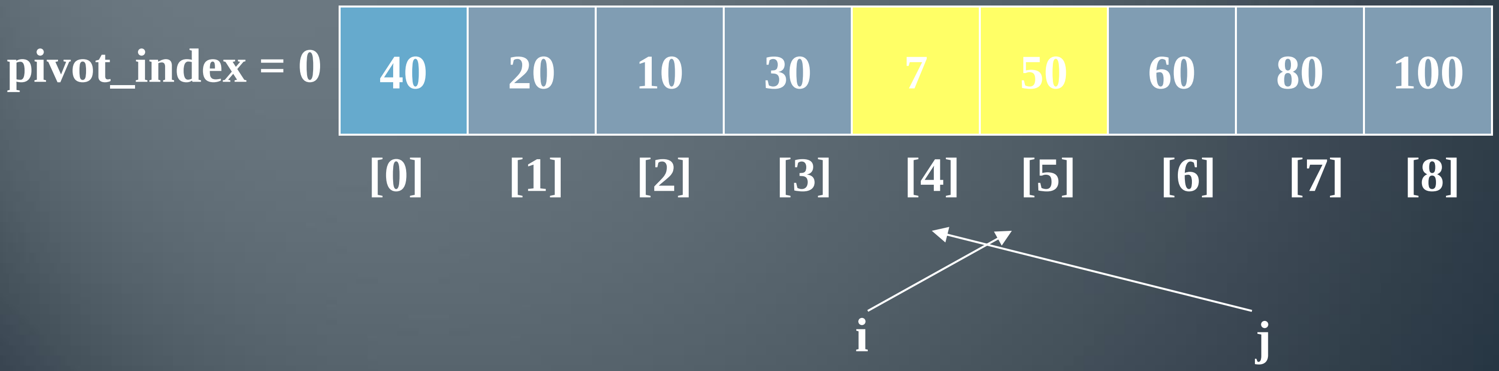


```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```

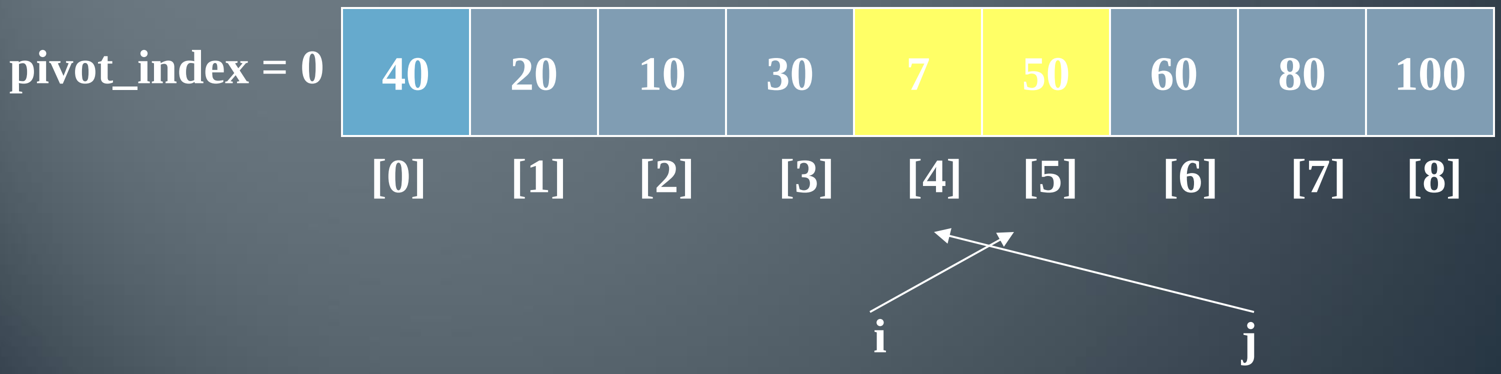
pivot_index = 0



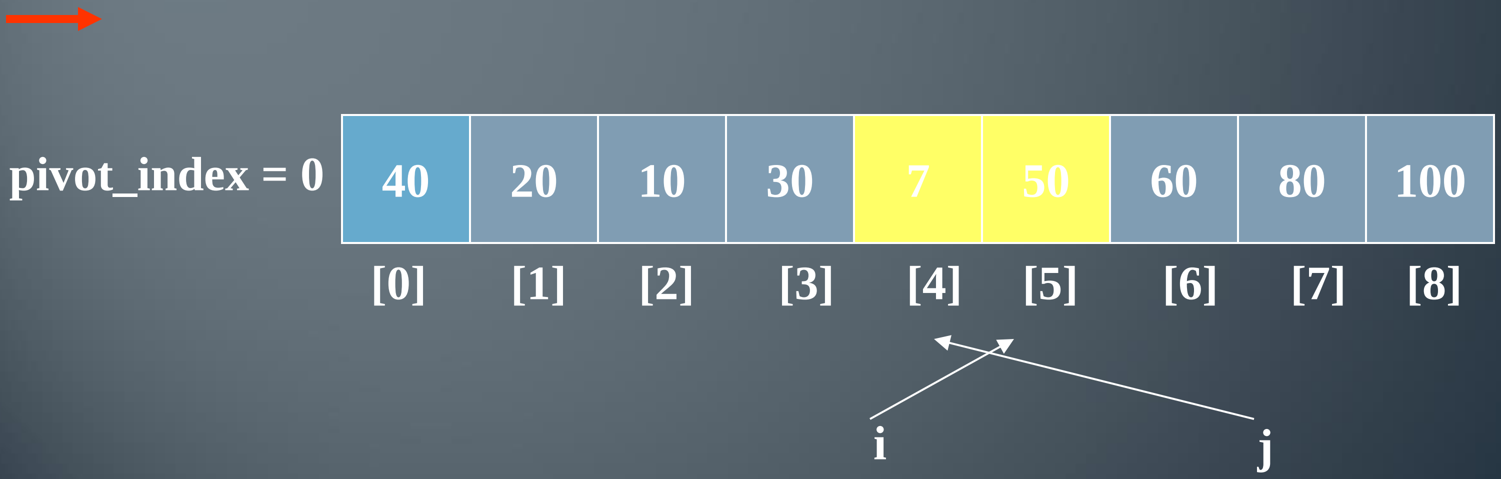
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



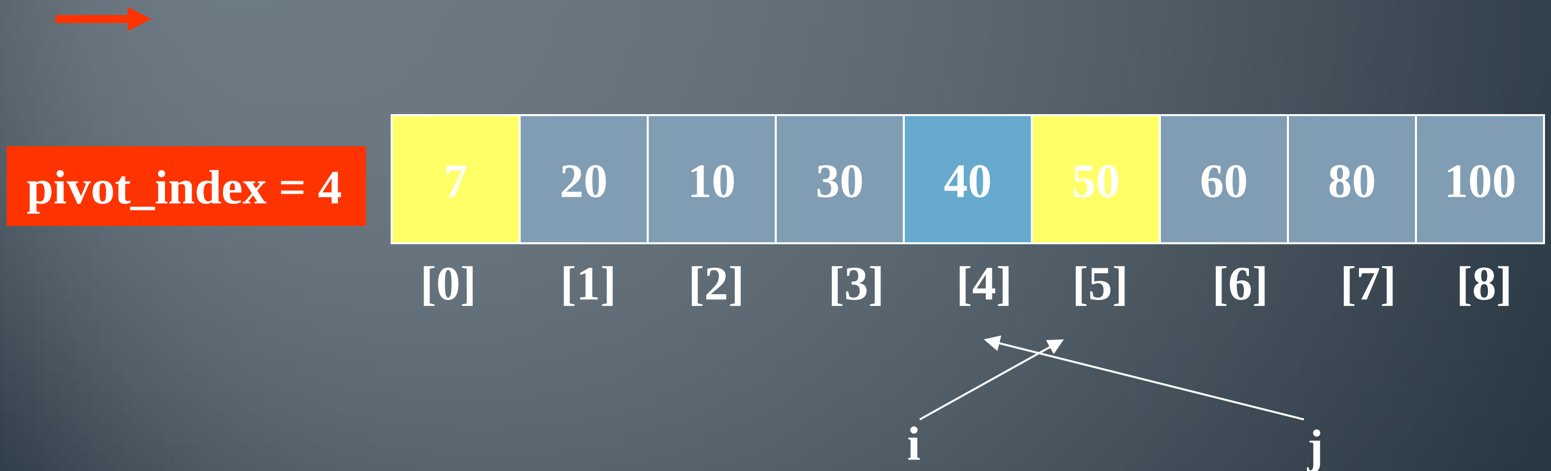
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



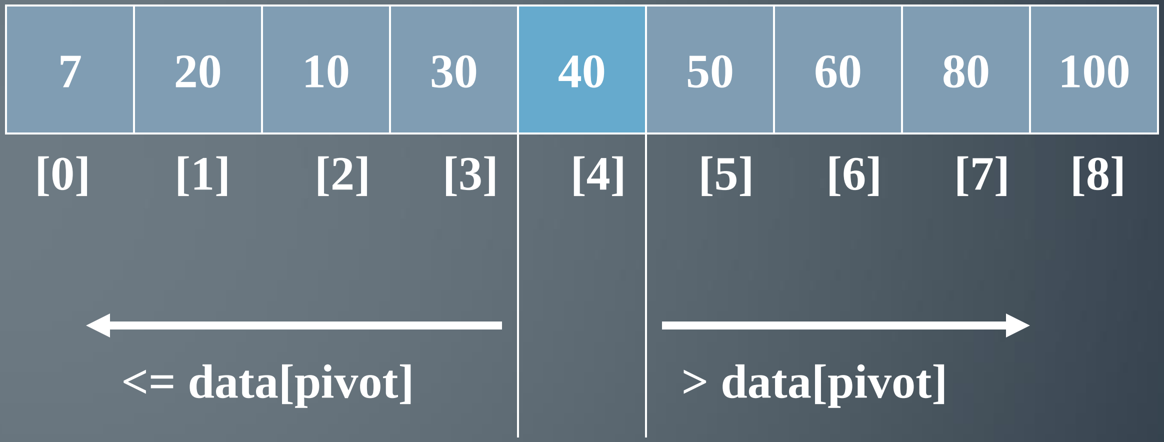
```
1. While data[i] <= data[pivot]
        ++i
2. While data[j] > data[pivot]
        --j
3. If i < j
        swap data[i] and data[j]
4. While j > i, go to 1.
5. Swap data[j] and data[pivot_index]
```



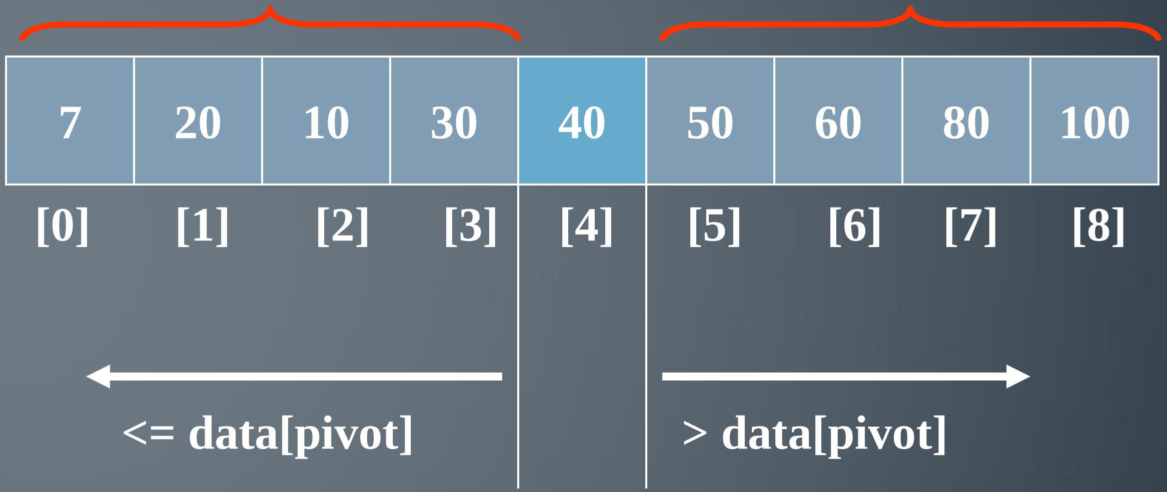
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
5.  Swap data[j] and data[pivot_index]
```



Partition Result



Recursion: Quicksort Sub-arrays



- **Note:** we know that the first element is smaller than the pivot, so the first element to be processed is the element to the right. The last two elements are the pivot and an element greater than the pivot, so they are not processed.

  - While **i** is to the left of **j**, we move **i** right, skipping all the elements less than the pivot. If an element is found greater then the pivot, **i** stops
  - While **j** is to the right of **i**, we move **j** left, skipping all the elements greater than the pivot. If an element is found less then the pivot, **j** stops
  - When both **i** and **j** have stopped, the elements are swapped.
  - When **i** and **j** have crossed, no swap is performed, scanning stops, and the element pointed to by **i** is swapped with the pivot .

Process:

Keep going from left side as long as a[i]<pivot and from the right side as long as a[j]>pivot

pivot →	85	24	63	95	17	31	45	98
		i						j
	85	24	63	95	17	31	45	98
			i					j
	85	24	63	95	17	31	45	98
				i				j
	85	24	63	95	17	31	45	98
				i			j	

If $i<j$ interchange i^{th} and j^{th} elements and then Continue the process.

85	24	63	45	17	31	95	98
			i			j	

85	24	63	45	17	31	95	98
				i		j	

85	24	63	45	17	31	95	98
					i		

85 24 63 45 17 31 95 98

j

i

85 24 63 45 17 31 95 98

j i

If i ≥j interchange jth and pivot elements and then divide the list into two sublists.

31 24 63 45 17 85 95 98

j

Two sublists:

31 24 63 45 17 | 85 | 95 98

Recursively sort

FirstPart and SecondPart

QickSort(low, j-1) QickSort(j+1,high)

Quick Sort Algorithm :

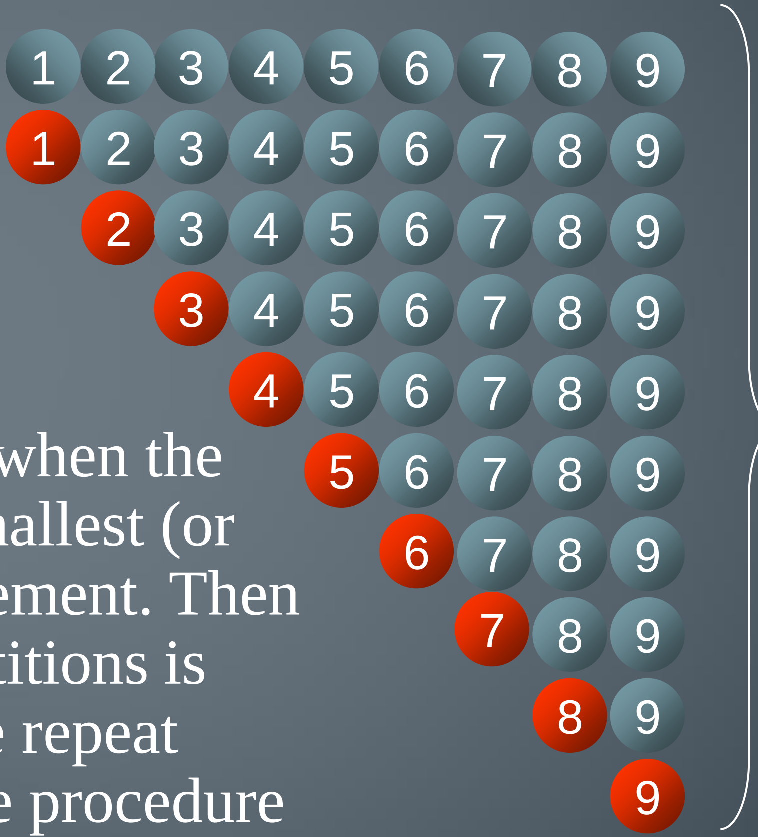
```
Algorithm QuickSort(low,high)
//Sorts the elements a[low],…..,a[high] which resides
//in the global array a[1:n] into ascending order;
// a[n+1] is considered to be defined and must ≥ all the
// elements  in a[1:n].
{
        if( low< high )  // if there are more than one element
        {              // divide p into two subproblems.
               j :=Partition(low,high);
                  // j is the position of  the partitioning element.
               QuickSort(low,j-1);
               QuickSort(j+1,high);
               // There is no need for combining solutions.
        }
}
```

```
Algorithm Partition(l,h)
{
        pivot:= a[l] ; i:=l;  j:= h+1;
        while( i < j ) do
        {
                    i++;
                    while( a[ i ] < pivot ) do
                            i++;
                    j--;
                    while( a[ j ]  > pivot ) do
                            j--;

                    if ( i < j ) then Interchange(i,j ); // interchange ith and
        }                                                 //  jth elements.

        Interchange(l, j ); return j;  // interchange pivot and jth  element.
}
```

```
Algorithm interchange (x,y )
{
        temp=a[x];
        a[x]=a[y];
        a[y]=temp;
}
```

Time complexity analysis

Worst-case:

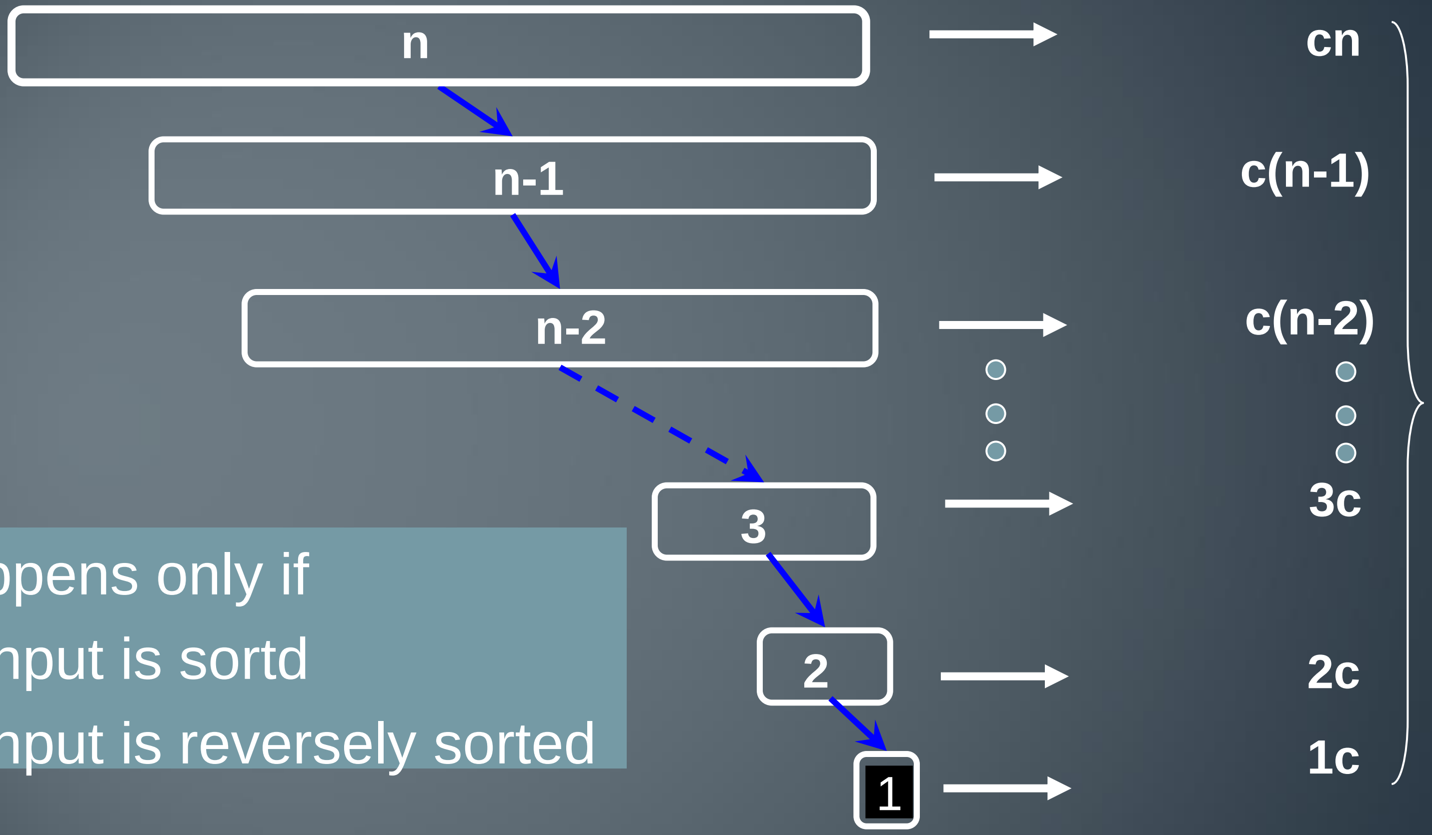


$$T(n) = \begin{cases} T(1) & \text{if } n = 1 \\ T(n-1)+cn & \text{if } n > 1 \end{cases}$$

$O(n^2)$

This happens when the pivot is the smallest (or the largest) element. Then one of the partitions is empty, and we repeat recursively the procedure for N-1 elements.

Worst/bad Case cont.



Happens only if

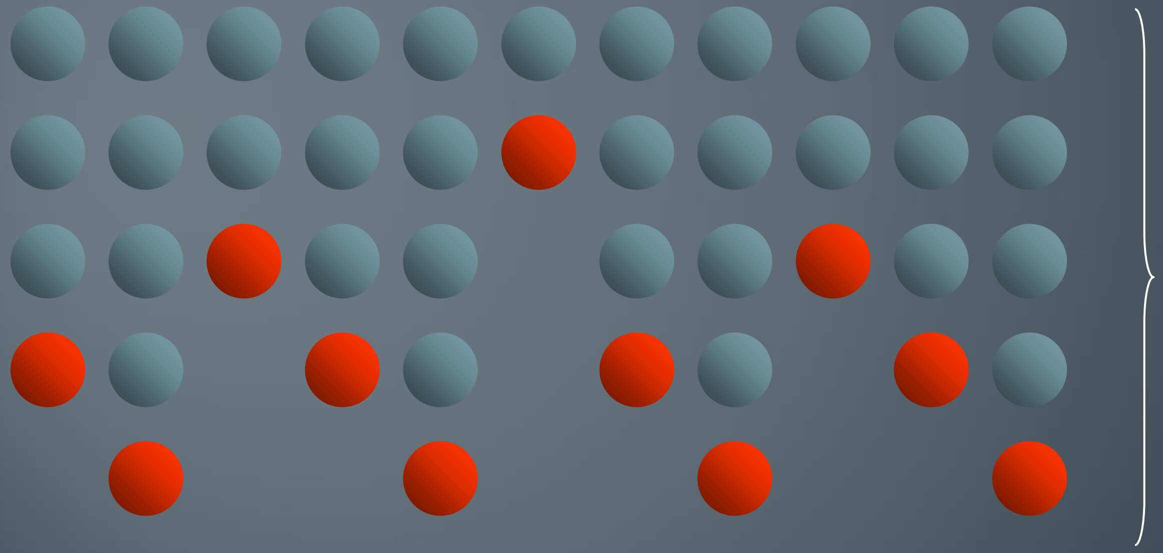
- input is sortd
- input is reversely sorted

Total time:

$O(n^2)$

Best-case

- It occurs only if each partition divides the list into two equal size sublists.



O(n logn)

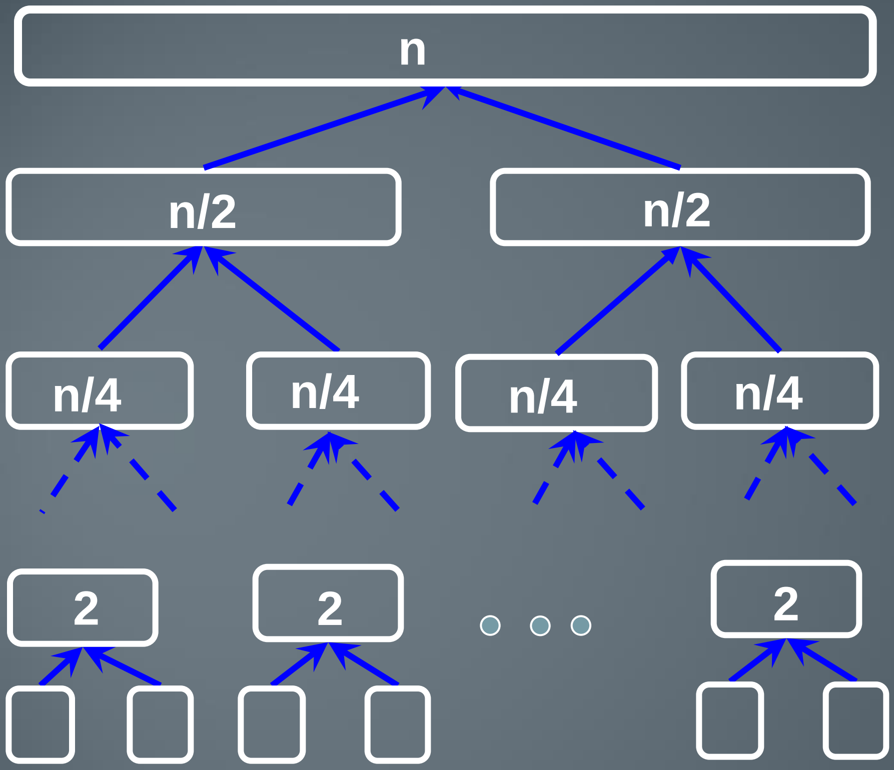
A best/good case cont.

- The best case is when the pivot is the median of the array, and then the left and the right part will have same size.

- There are **logN** partitions, and to obtain each partitions we do **N** comparisons (and not more than **N/2** swaps).

  Hence the complexity is **O(NlogN)**

Best/good Case



The Pivot is in the middle

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

- Total time: *O*(nlogn)

Average Case

- It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending.

So the average case time complexity of quicksort is O(n*logn).

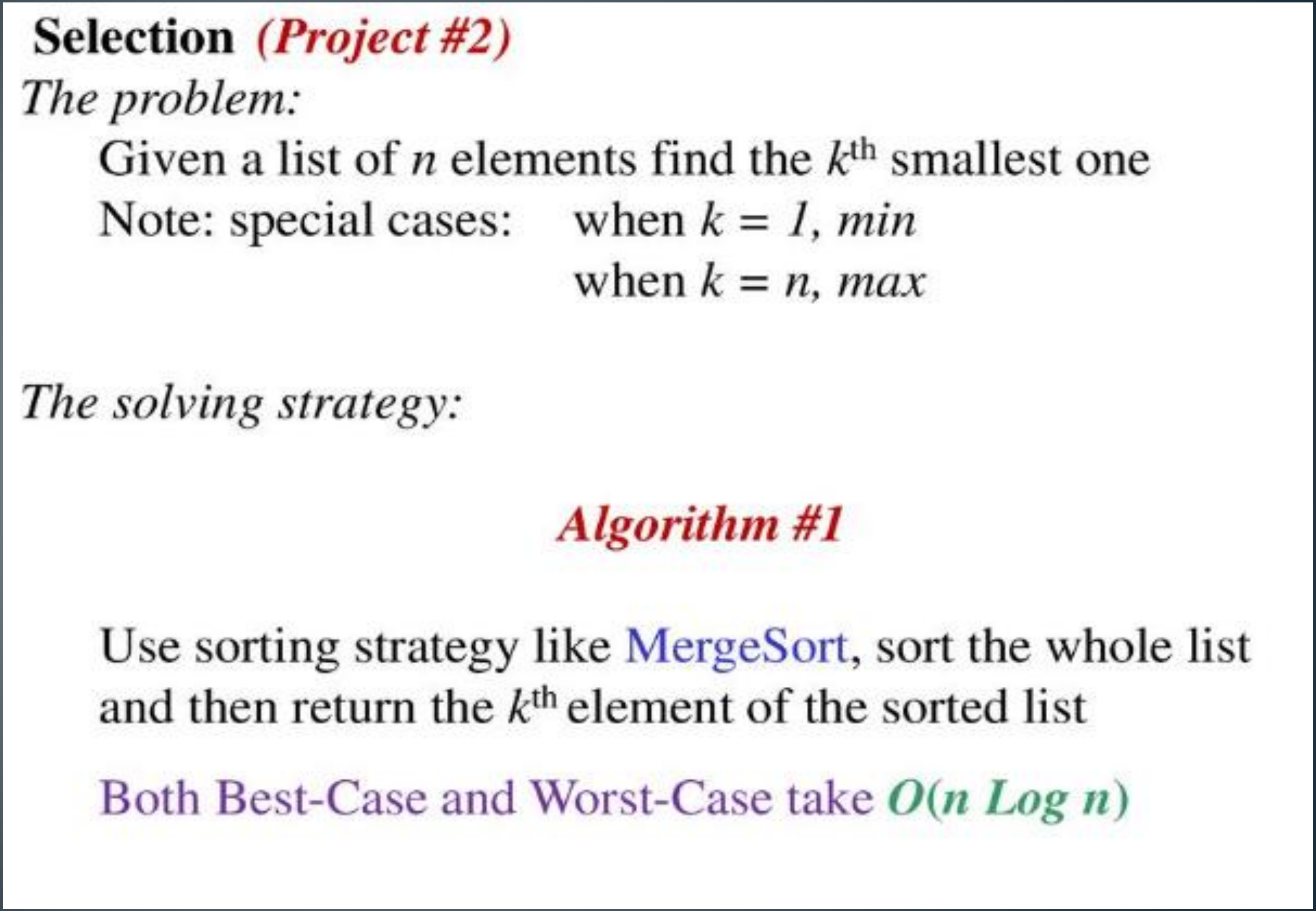
Conclusions

- **Advantages:**
  - One of the fastest algorithms on average.
  - Does not need additional memory (the sorting takes place in the array - this is called **in-place** processing). Compare with merge sort: merge sort needs additional memory for merging.
- **Disadvantages:**
  - The worst-case complexity is O(n^2)

Summary

- Quick-Sort
  - Most of the work done in partitioning
  - Best case takes **O(n log(n))** time
  - Average case takes **O(n log(n))** time
  - Worst case takes **O(n^2)** time

4. Selection Problem(Quick-Select)



Selection ***(Project #2)***

The problem:

Given a list of n elements find the k^{th} smallest one

Note: special cases: when $k = 1$, *min*

when $k = n$, *max*

The solving strategy:

Algorithm #1

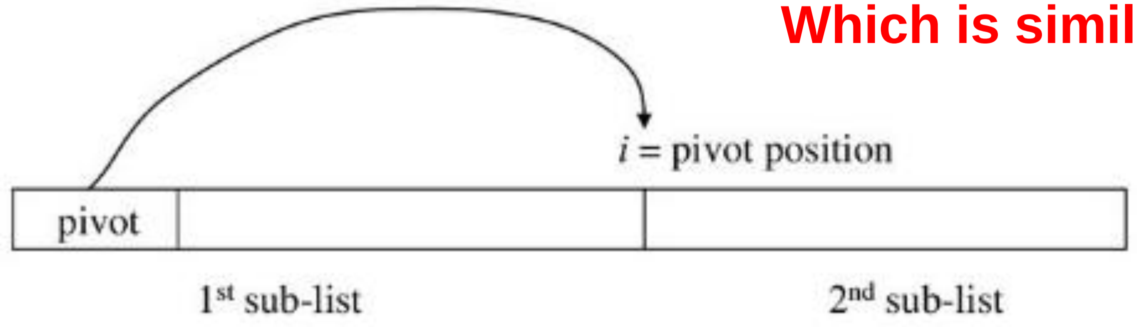
Use sorting strategy like MergeSort, sort the whole list and then return the k^{th} element of the sorted list

Both Best-Case and Worst-Case take $O(n \, Log \, n)$

Algorithm #2

idea: use "partition" (from Quick Sort) procedure repeatedly until we find the k^{th} *smallest* element.

Which is similar to quick sort



For each iteration:

If pivot position $i = k \Rightarrow$ Done!
If pivot position $i < k \Rightarrow$ to select $(k-i)^{th}$ element in the 2nd sub-list.
If pivot position $i > k \Rightarrow$ to select k^{th} element in the 1st sub-list.

Best-Case is $O(n)$ (when pivot is the k^{th} smallest one)
Worst-Case will call partition $O(n)$ times,
each time takes $O(n)$, so Worst-Case is $O(n^2)$.

Quick-Select

- This approach is similar to the quick sort algorithm where we use the partition on the input array recursively. But unlike quicksort, which processes both sides of the array recursively, this algorithm works on only one side of the partition.

- So we recur for either the left or right side according to the position of pivot.

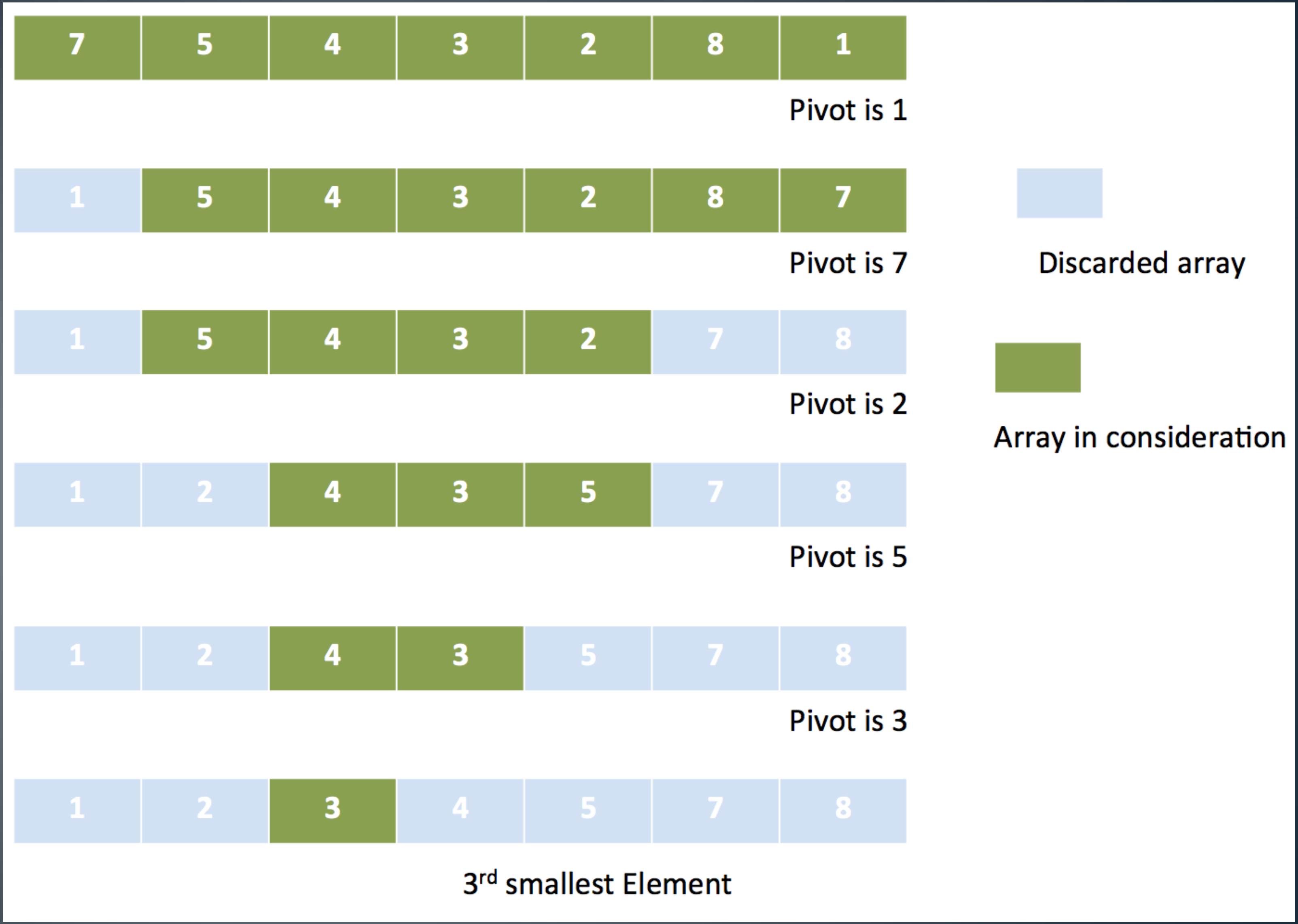
Algorithm

1. Partition the array A[left .. right] into two subarrays A[left .. pos] and A[pos + 1 .. right] such that each element of A[left .. pos] is less than each element of A[pos + 1 .. right].
2. Computes the number of elements in the subarray A[left .. pos] i.e. count = pos - left + 1
3. if (count == K), then A[pos] is the Kth smallest element.
4. Otherwise determines in which of the two subarrays A[left .. pos-1] and A[pos + 1 .. right] the Kth smallest element lies.
    - If (count > K) then the desired element lies on the left side of the partition
    - If (count < K), then the desired element lies on the right side of the partition. Since we already know i values that are smaller than the kth smallest element of A[left .. right], the desired element is the (K - count)th smallest element of A[pos + 1 .. right].
    - Base case is the scenario of single element array i.e left ==right. return A[left] or A[right].

Pseudo code

```
// Original value for left = 0 and right = n-1
int kthSmallest(int A[], int left, int right, int K)
{
    if (left == right)
         return A[left]
    int pos = partition(A, left, right)
    count = pos - left + 1
    if ( count == K )
         return A[pos]
    else if ( count > K )
         return kthSmallest(A, left, pos-1, K)
    else
         return kthSmallest(A, pos+1, right, K-i)
}
```

```
int partition(int A[], int l, int r)
{
    int x = A[r]
    int i = l-1
    for ( j = l to r-1 )
    {
        if (A[j] <= x)
        {
            i = i + 1
            swap(A[i], A[j])
        }
    }
    swap(A[i+1], A[r])
    return i+1
}
```


7 5 4 3 2 8 1
Pivot is 1
1 5 4 3 2 8 7
Pivot is 7
1 5 4 3 2 7 8
Pivot is 2
1 2 4 3 5 7 8
Pivot is 5
1 2 4 3 5 7 8
Pivot is 3
1 2 3 4 5 7 8
3rd smallest Element
Discarded array
Array in consideration

6. Finding maximum and minimum

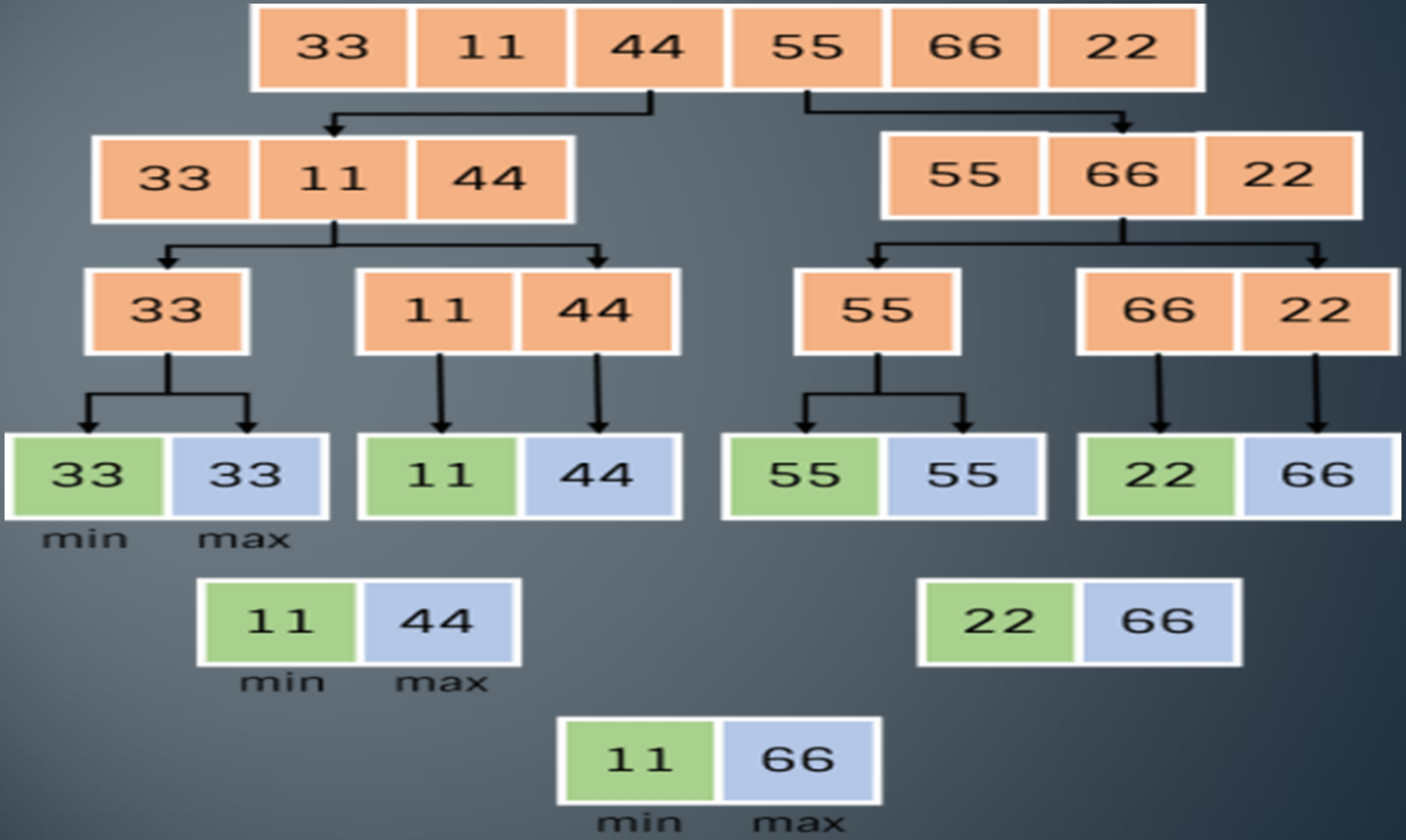
- Divide and conquer approach for Max. & Min problem works in three stages.

  Stage1: If a_1 is the only element in the array, a_1 is the maximum and minimum.

  Satge2: If the array contains only two elements a_1 and a_2, then the single comparison between two elements can decide the minimum and maximum of them.

  Stage3: If there are more than two elements, the algorithm divides the array from the middle and creates two sub-problems. Both sub-problems are treated as an independent problem and the same recursive process is applied to them. This division continues until sub-problem size becomes one or two.

For example, **find max and min from the sequence <33, 11, 44, 55, 66, 22> using divide and conquer approach**



```
Algorithm DC_MAXMIN (A, low, high)
{ // Description : Find minimum and maximum element from array using divide and
conquer approach
// Input : Array A of length n, and indices low = 0 and high = n - 1
// Output : (min, max) variables holding minimum and maximum element of array
 if n == 1 then
  return (A[1], A[1])
else if n == 2 then
  if A[1] < A[2] then
    return (A[1], A[2])
  else
    return (A[2], A[1])
else
 mid ← (low + high) / 2
  [LMin, LMax] = DC_MAXMIN (A, low, mid)
  [RMin, RMax] = DC_MAXMIN (A, mid + 1, high)
  if LMax > RMax then   // Combine solution
    max ← LMax
  else
    max ← RMax
 if LMin < RMin then    // Combine solution
    min ← LMin
  else
    min ← RMin
 return (min, max)
```

Time Complexity

DC_MAXMIN does two comparisons to determine the minimum and maximum element and creates two problems of size n/2, so the recurrence can be formulated as

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ 2T(n/2) + 2 & \text{if } n > 2 \end{cases}$$

Let us solve this equation using interactive approach.

$T(n) = 2T(n/2) + 2$ … **(1)**

By substituting n by (n / 2) in Equation (1)

$T(n/2) = 2T(n/4) + 2$

$\Rightarrow T(n) = 2(2T(n/4) + 2) + 2$

$= 4T(n/4) + 4 + 2$ … **(2)**

By substituting n by n/4 in Equation (1),

$T(n/4) = 2T(n/8) + 2$

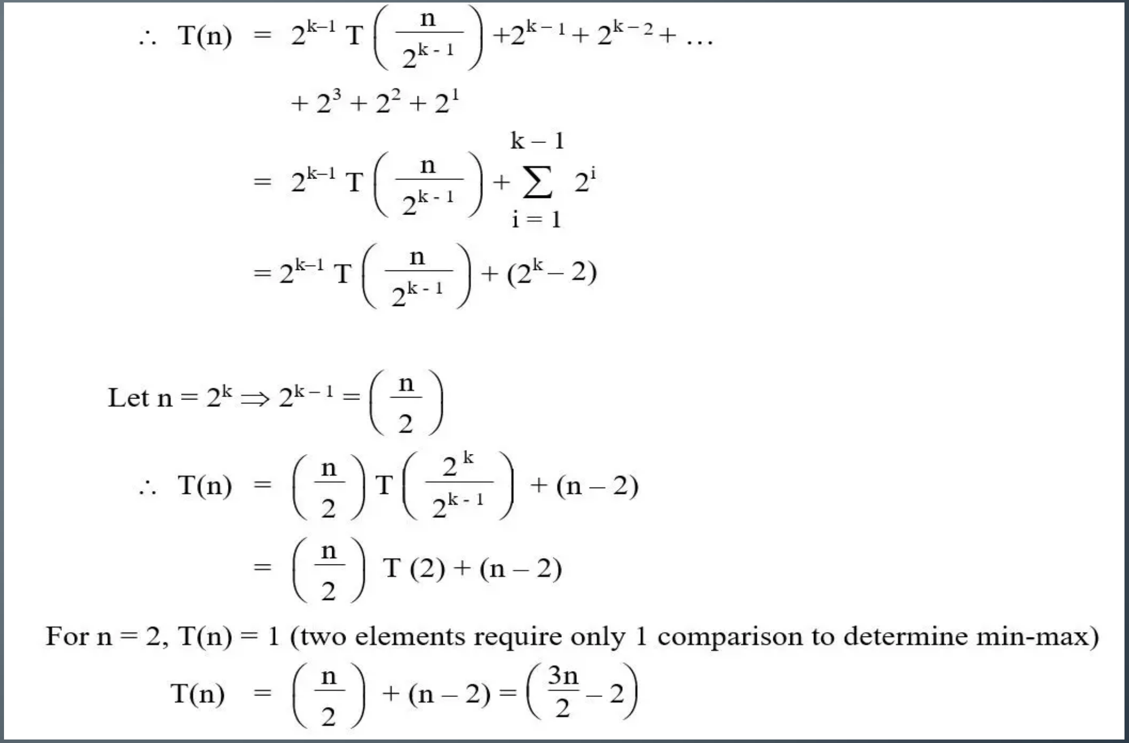
Substitute it in Equation (1),

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

$= 8T(n/8) + 8 + 4 + 2$

$= \; 2^3 T(n/2^3) + 2^3 + 2^2 + 2^1$

After k – 1 iterations



$$\therefore \; T(n) \; = \; 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + 2^{k-1} + 2^{k-2} + \ldots + 2^3 + 2^2 + 2^1$$

$$= \; 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + \sum_{i=1}^{k-1} 2^i$$

$$= 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + (2^k - 2)$$

$$\text{Let } n = 2^k \Rightarrow 2^{k-1} = \left(\frac{n}{2}\right)$$

$$\therefore \; T(n) \; = \; \left(\frac{n}{2}\right) T\left(\frac{2^k}{2^{k-1}}\right) + (n-2)$$

$$= \; \left(\frac{n}{2}\right) T(2) + (n-2)$$

For n = 2, T(n) = 1 (two elements require only 1 comparison to determine min-max)

$$T(n) \; = \; \left(\frac{n}{2}\right) + (n-2) = \left(\frac{3n}{2} - 2\right)$$

O(n)

5.Strassen’s Matrix Multiplication

Basic Matrix Multiplication

Let A an B two n×n matrices. The product C=AB is also an n×n matrix.

```
void matrix_mult (){
  for (i = 1; i <= N; i++) {
        for (j = 1; j <= N; j++) {
              for(k=1; k<=N; k++){
                C[i,j]=C[i,j]+A[i,k] * B[k,j];
                                          }
  }}
```

Time complexity of above algorithm is
T(n)=O(n3)

Divide and Conquer technique

- We want to compute the product C=AB, where each of A,B, and C are n×n matrices.
- Assume n is a power of 2.
- If n is not a power of 2, add enough rows and columns of zeros.
- We divide each of A,B, and C into four n/2×n/2 matrices, rewriting the equation C=AB as follows:

$$\left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) * \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

Then,

$C_{11}=A_{11}B_{11}+A_{12}B_{21}$

$C_{12}=A_{11}B_{12}+A_{12}B_{22}$

$C_{21}=A_{21}B_{11}+A_{22}B_{21}$

$C_{22}=A_{21}B_{12}+A_{22}B_{22}$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{array}{c} A_{11} \quad\quad A_{12} \\ \left[\begin{array}{cc|cc} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ \hline 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{array}\right] \\ A_{21} \quad\quad A_{22} \end{array} \times \begin{array}{c} B_{11} \quad\quad B_{12} \\ \left[\begin{array}{cc|cc} 5 & 5 & 6 & 6 \\ 5 & 5 & 6 & 6 \\ \hline 7 & 7 & 8 & 8 \\ 7 & 7 & 8 & 8 \end{array}\right] \\ B_{21} \quad\quad B_{22} \end{array}$$

- Each of these four equations specifies two multiplications of $n/2\times n/2$ matrices and the addition of their $n/2\times n/2$ products.
- We can derive the following recurrence relation for the time $T(n)$ to multiply two $n\times n$ matrices:

$$T(n)= \begin{cases} c_1 & \text{if } n<=2 \\ 8T(n/2)+ c_2n^2 & \text{if } n>2 \end{cases}$$

T(n) = O(n3)

• This method is no faster than the ordinary method.

$$T(n) = 8T(n/2) + c_2 n^2$$

$$= 8\left[8T(n/4) + c_2 (n/2)^2 \right] + c_2 n^2$$

$$= 8^2\, T(n/4) + c_2 2n^2 + c_2 n^2$$

$$= 8^2 \left[8T(n/8) + c_2 (n/4)^2 \right] + c_2 2n^2 + c_2 n^2$$

$$= 8^3\, T(n/8) + c_2 4n^2 + c_2 2n^2 + c_2 n^2$$

$$\vdots$$

$$= 8^k T(1) + \underbrace{\ldots\ldots\ldots\ldots\ldots\ldots + c_2 4n^2 + c_2 2n^2 + c_2 n^2}$$

$$= 8^{\log_2 n} c_1 + c\, n^2$$

.

$$= n^{\log_2 8} c_1 + c\, n^2 = n^3 c_1 + cn^2 = O(n^3)$$

Strassen's method

- Matrix multiplications are more expensive than matrix additions or subtractions($O(n^3)$ versus $O(n^2)$).
- Strassen has discovered a way to compute the multiplication using only 7 multiplications and 18 additions or subtractions.
- His method involves computing 7 $n \times n$ matrices $M_1, M_2, M_3, M_4, M_5, M_6$, and M_7, then cij's are calculated using these matrices.

Formulas for Strassen's Algorithm

$M_1 = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$M_2 = (A_{21} + A_{22}) * B_{11}$

$M_3 = A_{11} * (B_{12} - B_{22})$

$M_4 = A_{22} * (B_{21} - B_{11})$

$M_5 = (A_{11} + A_{12}) * B_{22}$

$M_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$

$M_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

C_{11}=M1 + M4 - M_5 + M_7
C_{12}= M_3 + M_5
C_{21}= M_2 + M_4
C_{22}=M_1 + M_3 - M_2 + M_6

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

The resulting recurrence relation for T(n) is

$$T(n) = \begin{cases} c_1 & n<=2 \\ 7T(n/2) + c_2 n^2 & n>2 \end{cases}$$

$T(n) = 7T(n/2) + c_2 n^2$

$= 7\left[7T(n/4) + c_2(n/2)^2\right] + c_2 n^2$

$= 7^2\, T(n/4) + 7/4 c_2 n^2 + c_2 n^2$

$= 7^2 \left[7T(n/8) + c_2(n/4)^2\right] + c_2 2 n^2 + c_2 n^2$

$= 7^3\, T(n/8) + (7/4)^2 c_2 n^2 + 7/4 c_2 n^2 + c_2 n^2$

$\vdots$

$= 7^k T(1) + c_2 n^2 \left[1 + 7/4 + (7/4)^2 + (7/4)^3 + \ldots\ldots\ldots\ldots\ldots\ldots + (7/4)^{k-1}\right]$

$= 7^{\log_2 n} c_1 + c_2\, n^2\, (7/4)^{\log_2 n}$

$= c_1\, n^{\log_2 7} + c_2\, n^2\, (7/4)^{\log_2 n}$

$= c_1\, n^{\log_2 7} = O(n^{\log_2 7}) \sim O(n^{2.81})$

$$T(n) = 7^k T(1) + c_2 n^2 \left[1 + 7/4 + (7/4)^2 + (7/4)^3 + \ldots\ldots\ldots\ldots\ldots + (7/4)^{k-1} \right]$$

$$S_n = a + ar + ar^2 + \ldots + ar^{n-1}$$

When $r > 1$, $S_n = a\frac{(r^n - 1)}{(r - 1)}$

$$= 7^{\log_2 n} c_1 \quad + c_2\, n^2\, (7/4)^{\log_2 n}$$

$$= c_1\, n^{\log_2 7} + c_2\, n^2\, (7/4)^{\log_2 n}$$

$$= c_1\, n^{\log_2 7} \quad = O(n^{\log_2 7}) \sim O(n^{2.81})$$

THANK YOU!

KEEP CALM AND SAVE TREES
KeepCalmAndPosters.com