Unit-II

UNIT II: Divide and conquer: General method, applications-Binary search, Quick sort, Merge sort, Strassen's matrix multiplication. Applications: PNR number Search, sorting the google search results.

Divide and Conquer Technique

General Method:

- The Divide and Conquer Technique splits n inputs into k subsets , $1 < k \leq n$, yielding k subproblems.

- These subproblems will be solved and then combined by using a separate method to get a solution to the whole problem.

- If the subproblems are large, then the Divide and Conquer Technique will be reapplied.

- Often subproblems resulting from a Divide and Conquer Technique are of the same type as the original problem.

- 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 subproblems that are small enough to solve without splitting further.

Control Abstraction / General Method for Divide and Conquer Technique

```
Algorithm DAndC(p)
{
    if Small(p) then return s(p);
    else
    {
        divide p into smaller problems p1,p2,…….,pk, k≥1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(p1), DAndC(p2),……,DAndC(pk));
    }
}
```

- If the size of p is n and the sizes of the k subproblems are $n_1, n_2, \ldots, n_k$, then the computing time of DAndC is described by the recurrence relation

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

- 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 of subproblems.

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

$$T(n)= \begin{cases} c & n \text{ small} \\ aT(n/b)+f(n) & \text{Otherwise} \end{cases}$$

- Where a , b and c are known constants, and n is a power of b (i.e $n=b^k$)

Applications

1.Binary search Algorithm

Iterative Method

```
Algorithm BinSearch(a, n, x)
// a is an array of size n, x is the key element to be searched.
{       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, low, high, x)
//Given an array a [ low : high ]of elements in increasing
//order,1≤low≤high,determine whether x is present, and
//if so, return j such that x=a[j]; else return 0.
{
    if( low = high ) then              // If small(P)
    {
        if( x=a[low] ) then return low;
        else return 0;
    }
    else
```

```
{
  //Reduce p into a smaller subproblem.
        mid:=  (low+high)/2
        if( x = a[mid] ) then return mid;
        else if ( x<a[mid] ) then
                return BinSrch(a, low, mid-1, x);
        else
        return BinSrch(a, mid+1, high, x);
 }
}
```

Time complexity of Binary Seaych

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

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

Assume $n=2^k$, then

$$
\begin{aligned}
T(n) &= T(n/2) + c2 \\
&= T(n/4)+c2+c2 \\
&= T(n/8) +c2+c2+c2 \\
&\ \ \ldots\ldots \\
&\ \ \ldots\ldots \\
&= T(n / 2^k) + c2+c2+c2+ \ldots\ldots\ldots k \text{ times} \\
&= T(1)+ kc2
\end{aligned}
$$

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

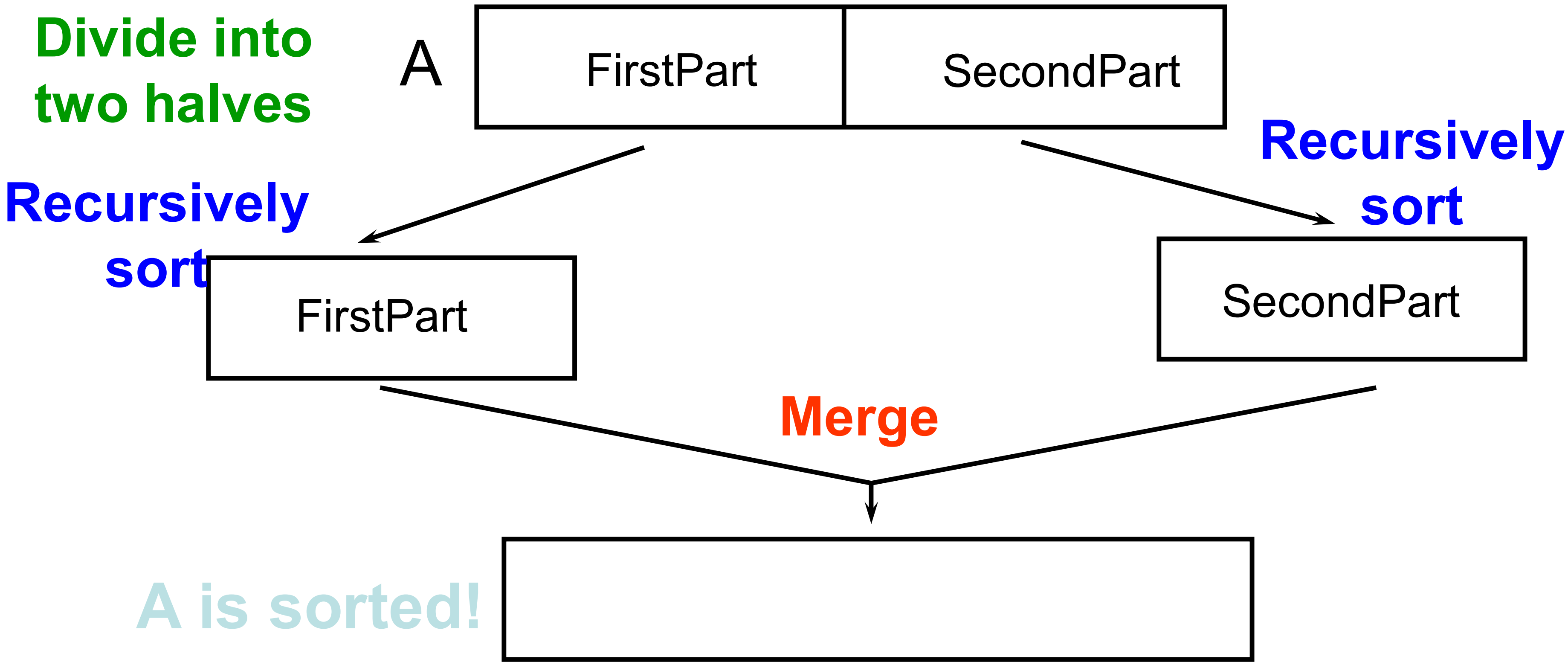
1. **Base Case**, solve the problem **directly** if it is small enough(only one element).

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



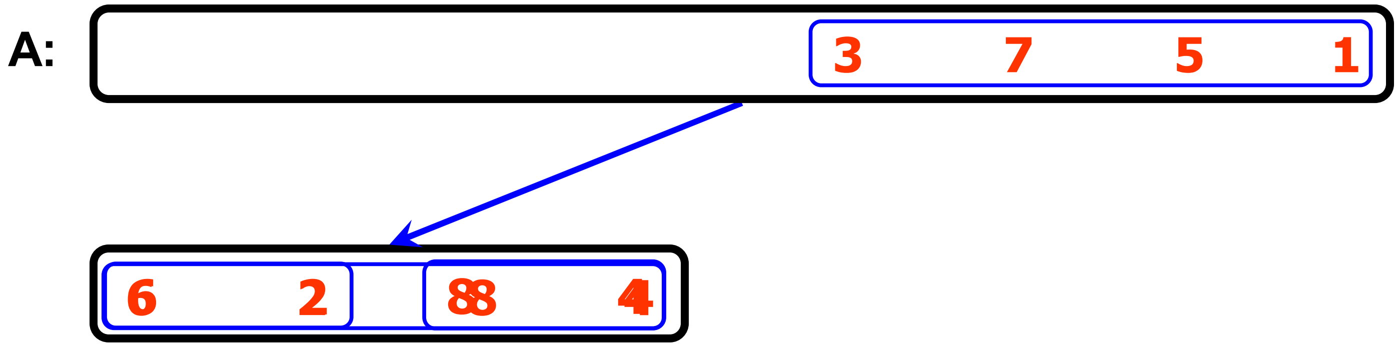
Merge-Sort(A, 0, 7)

Divide

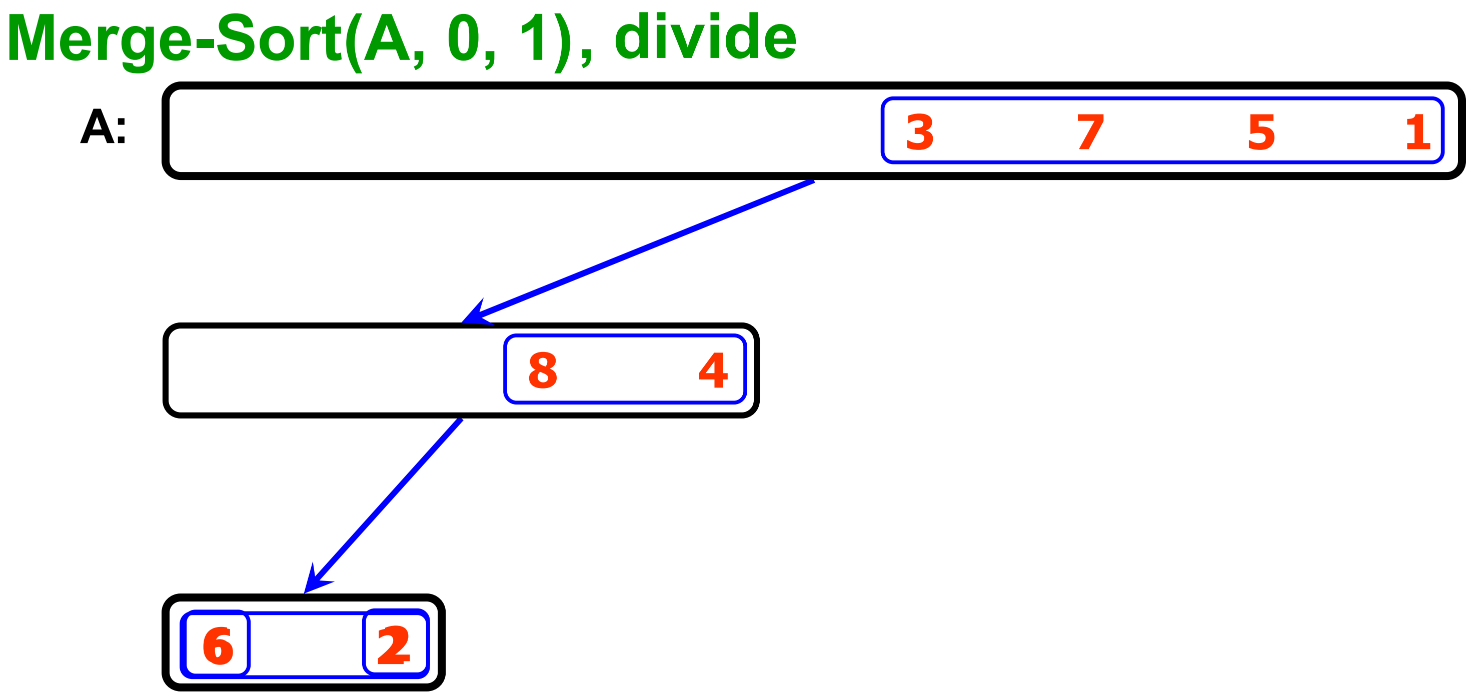
A: 6 2 8 4 3 3 7 7 5 5 1 1

Merge-Sort(A, 0, 7)

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

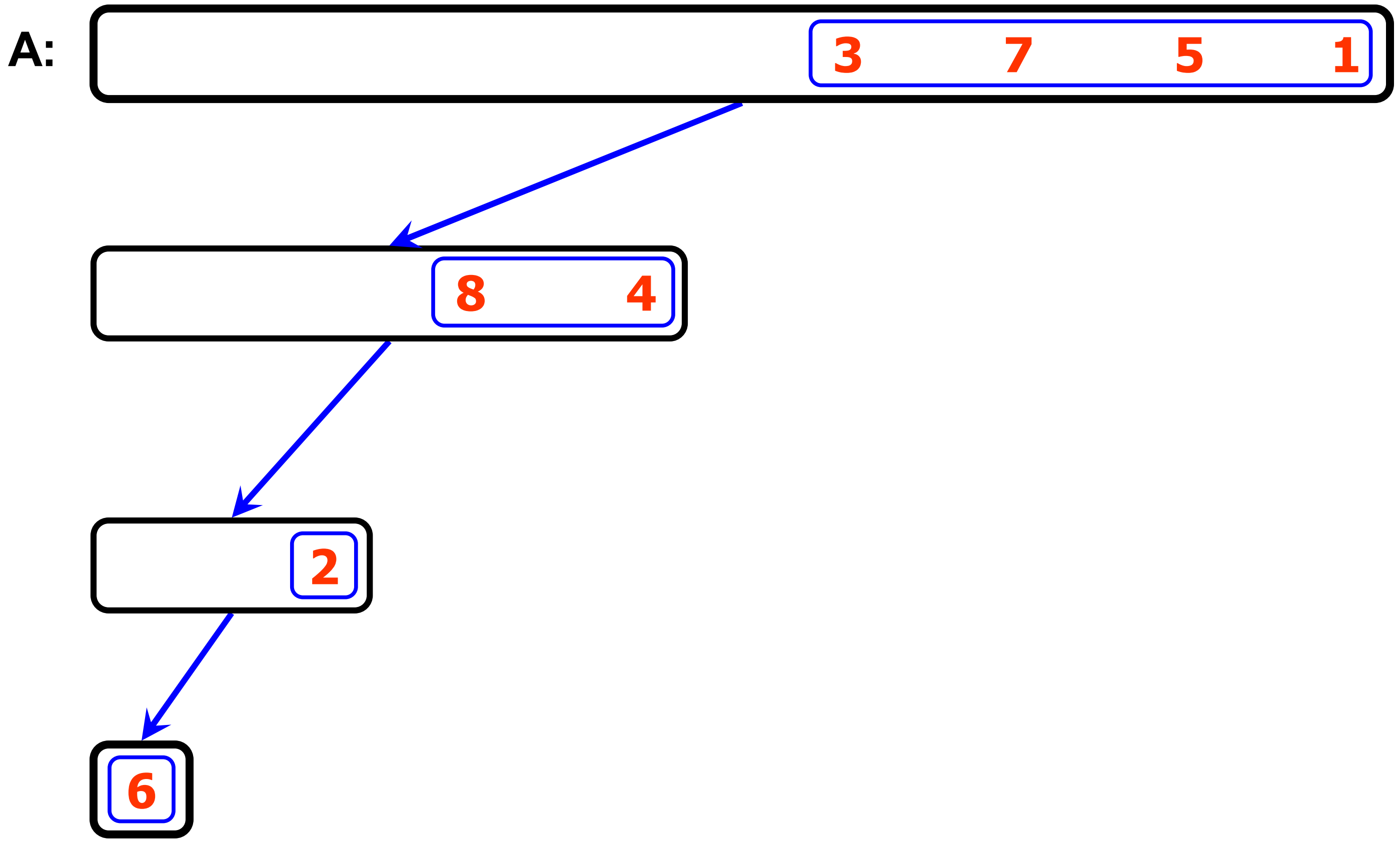


Merge-Sort(A, 0, 7)



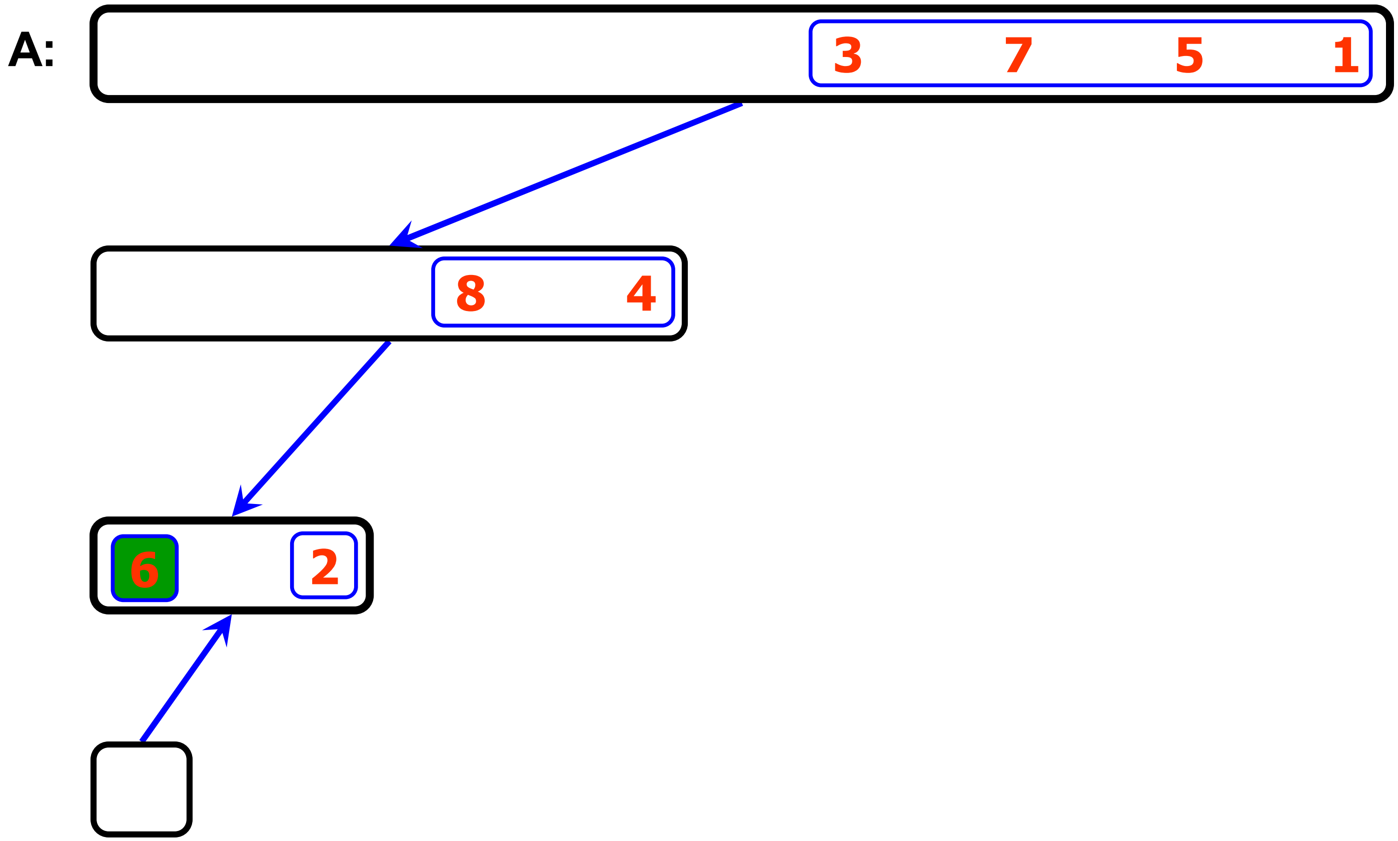
Merge-Sort(A, 0, 7)

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

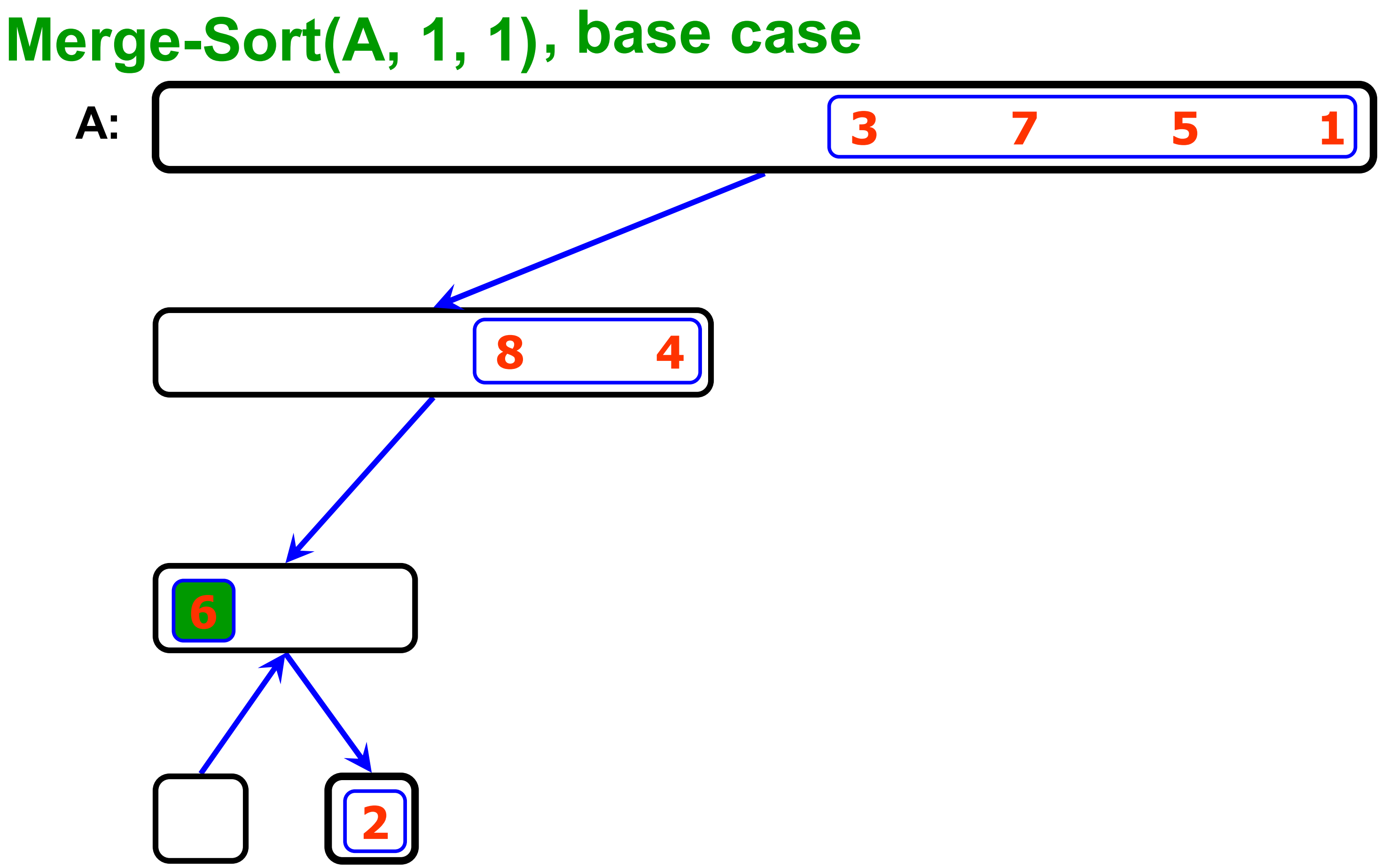


Merge-Sort(A, 0, 7)

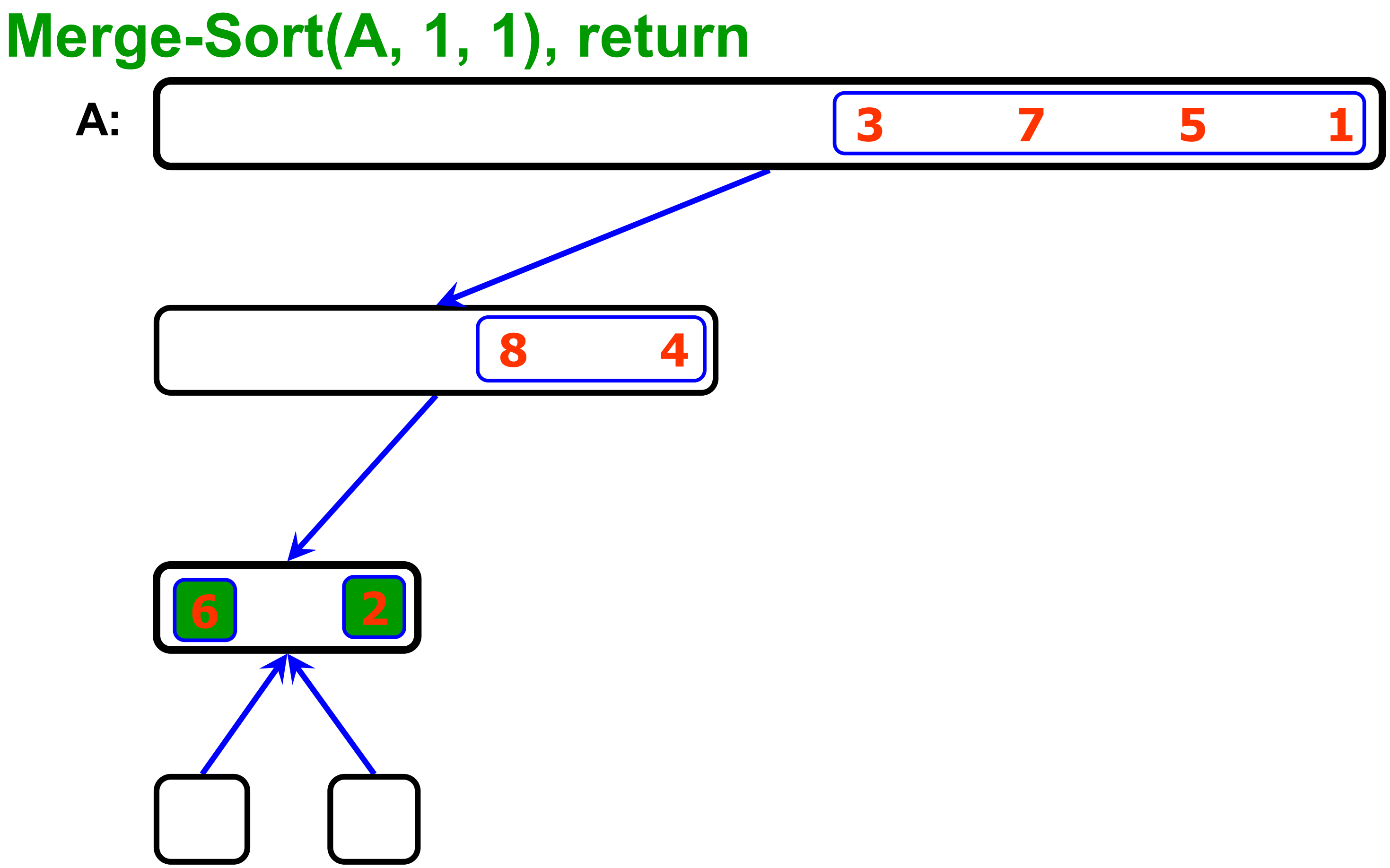
Merge-Sort(A, 0, 0), return



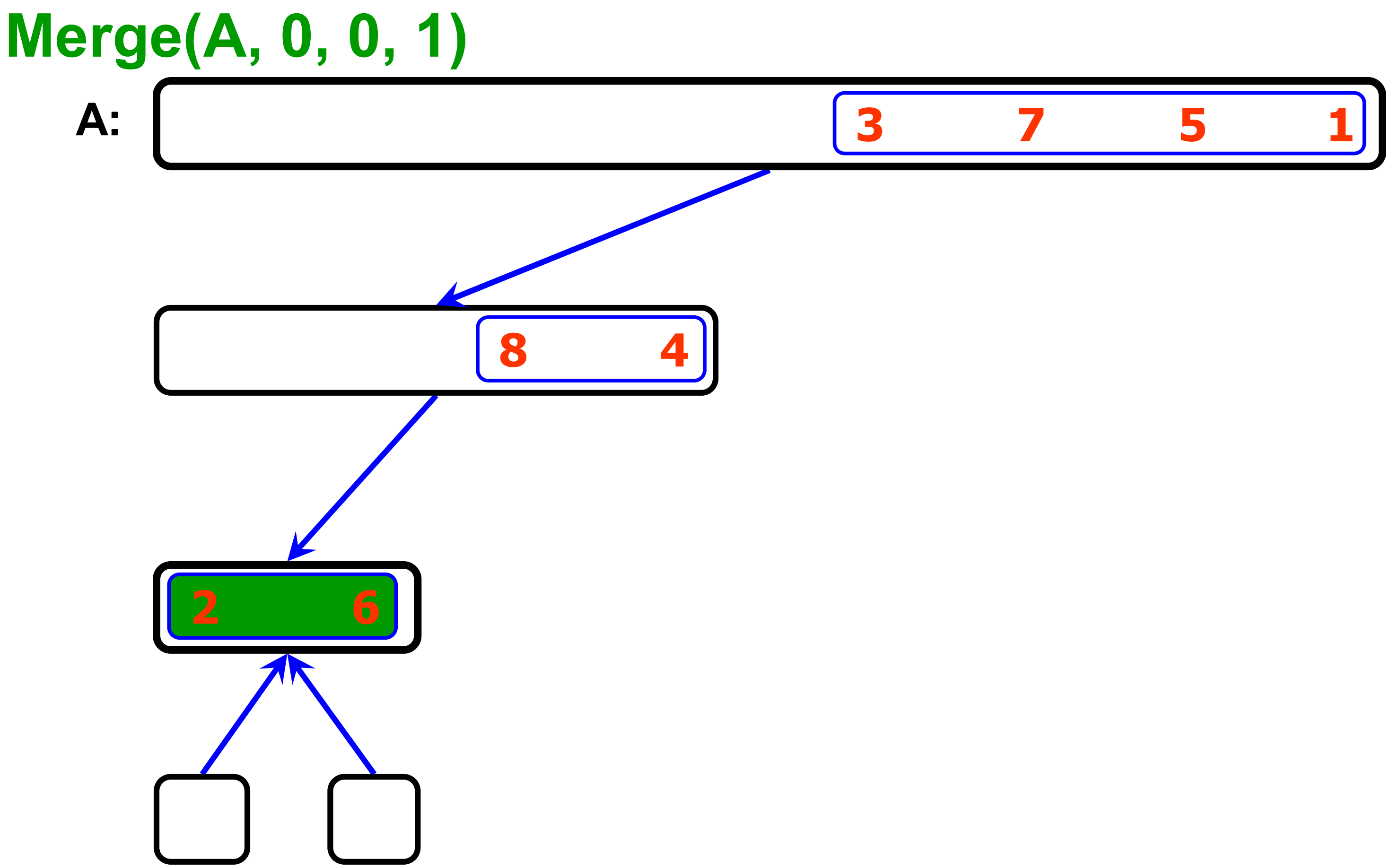
Merge-Sort(A, 0, 7)



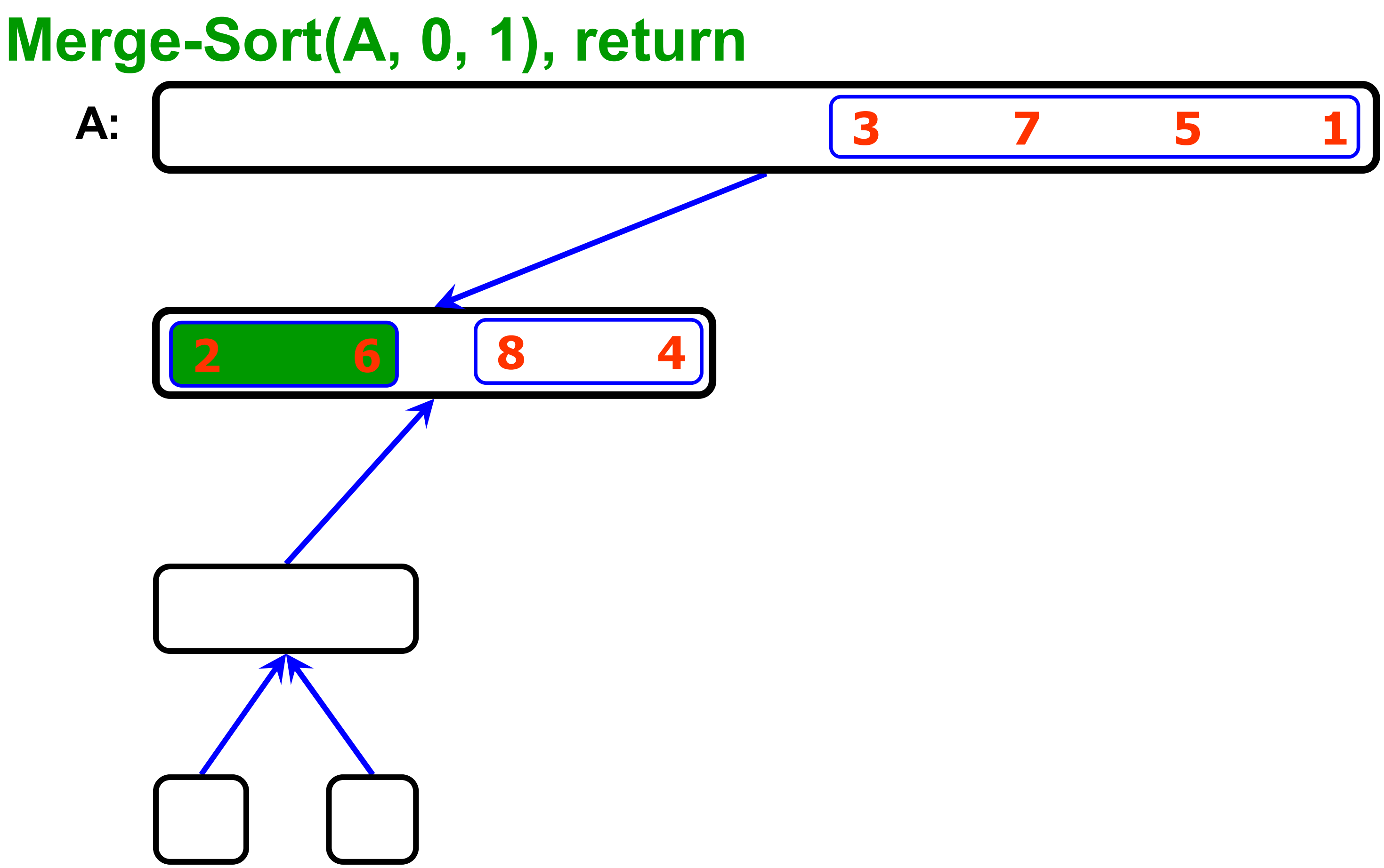
Merge-Sort(A, 0, 7)



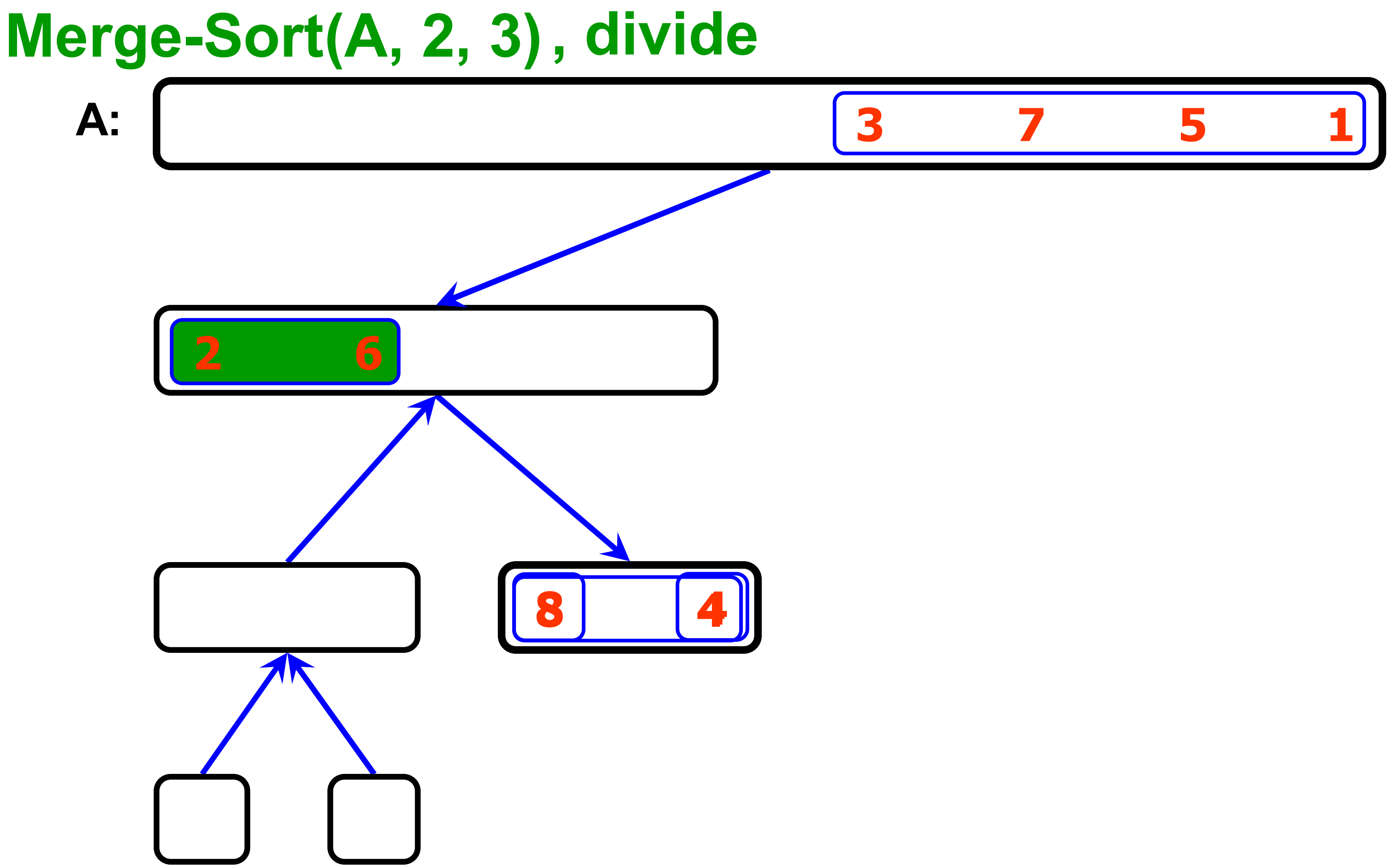
Merge-Sort(A, 0, 7)



Merge-Sort(A, 0, 7)

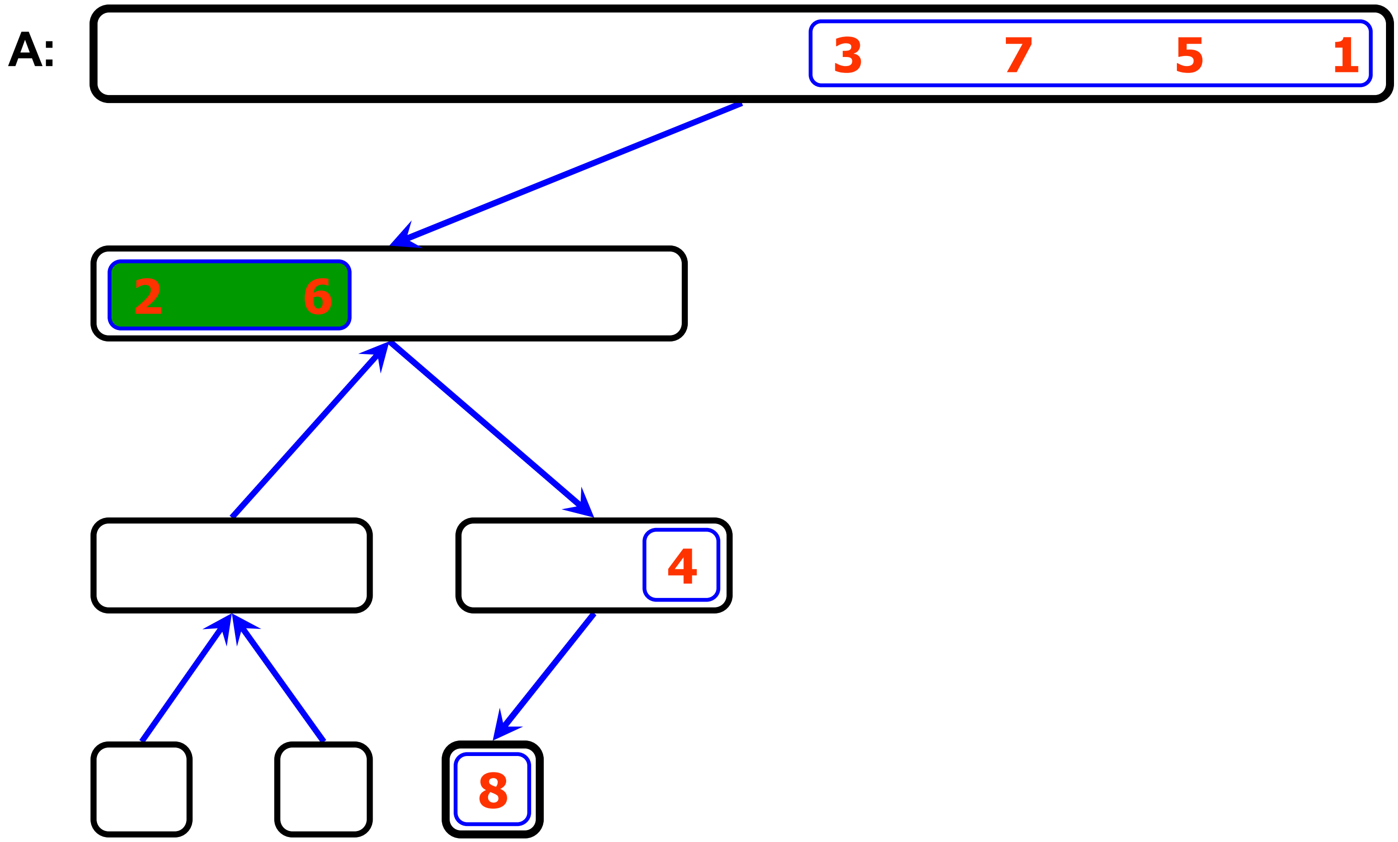


Merge-Sort(A, 0, 7)

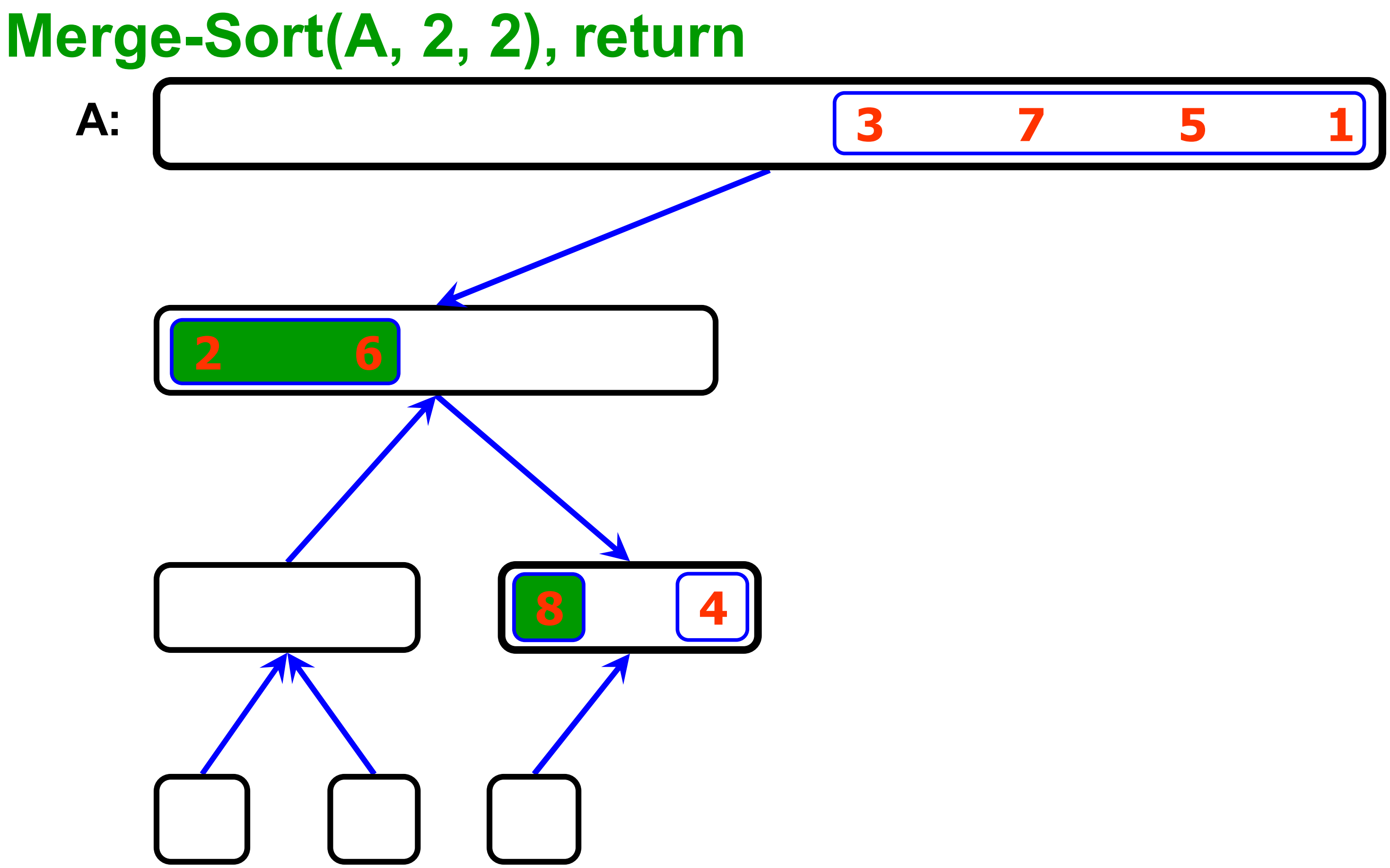


Merge-Sort(A, 0, 7)

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

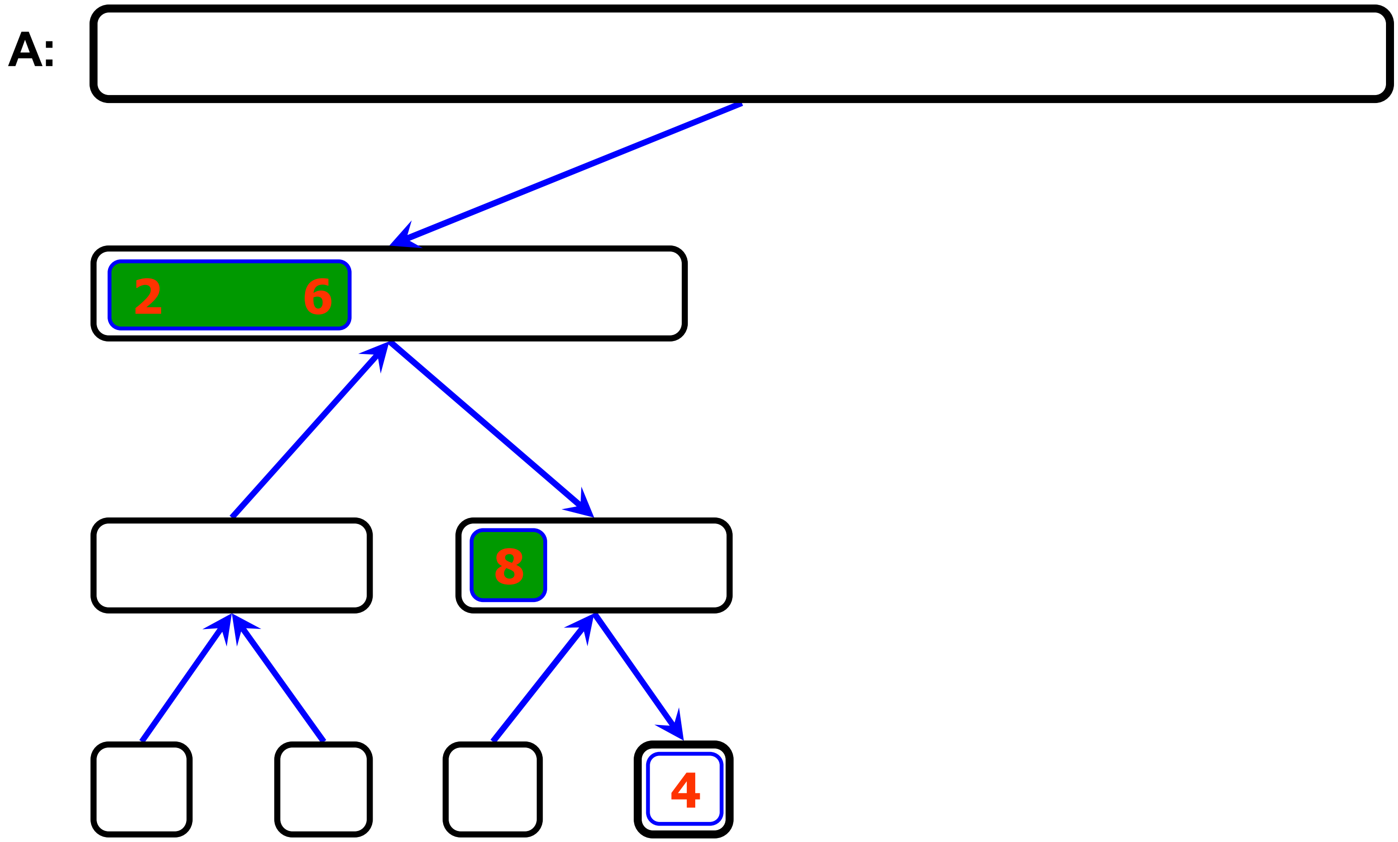


Merge-Sort(A, 0, 7)

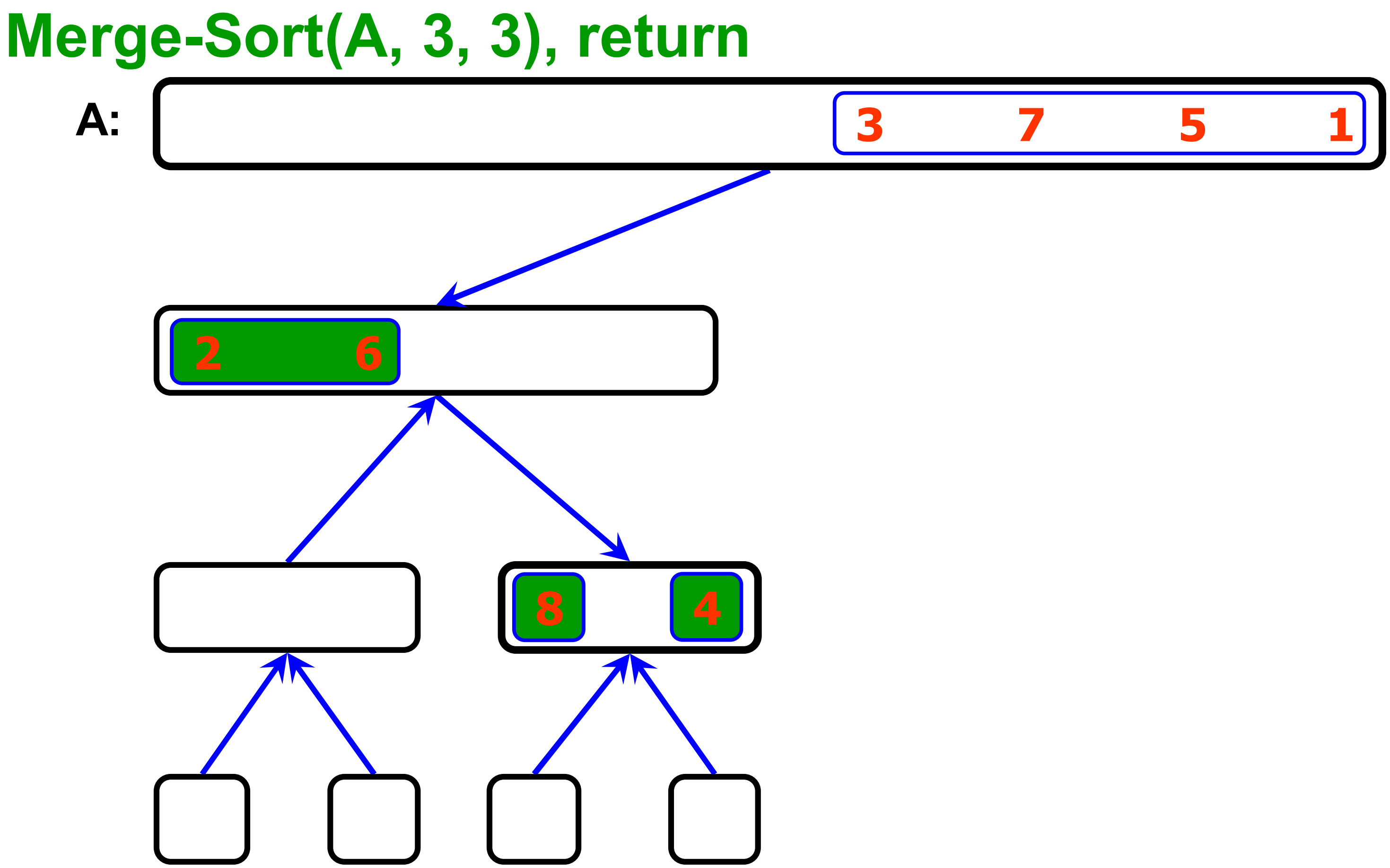


Merge-Sort(A, 0, 7)

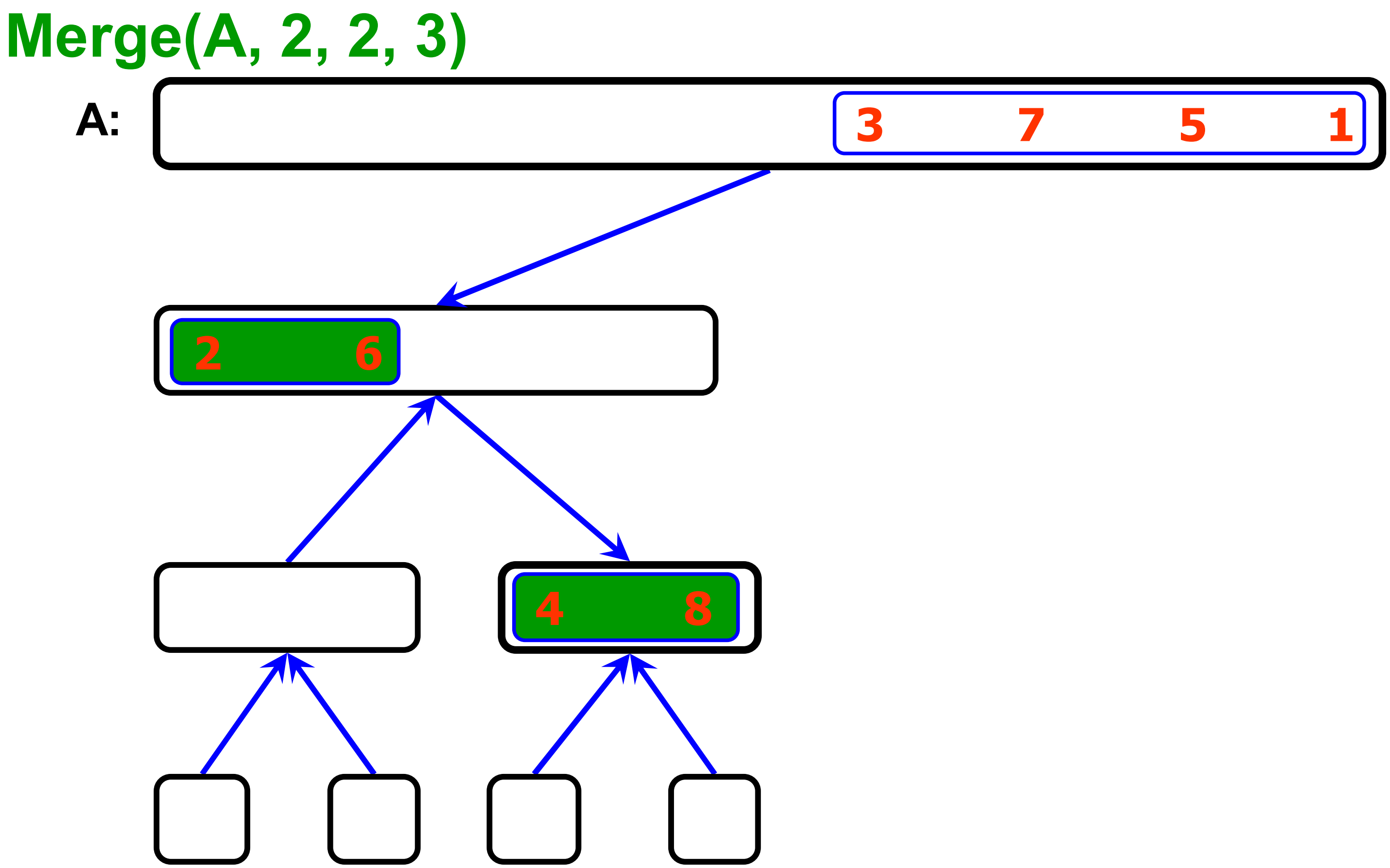
Merge-Sort(A, 3, 3), base case



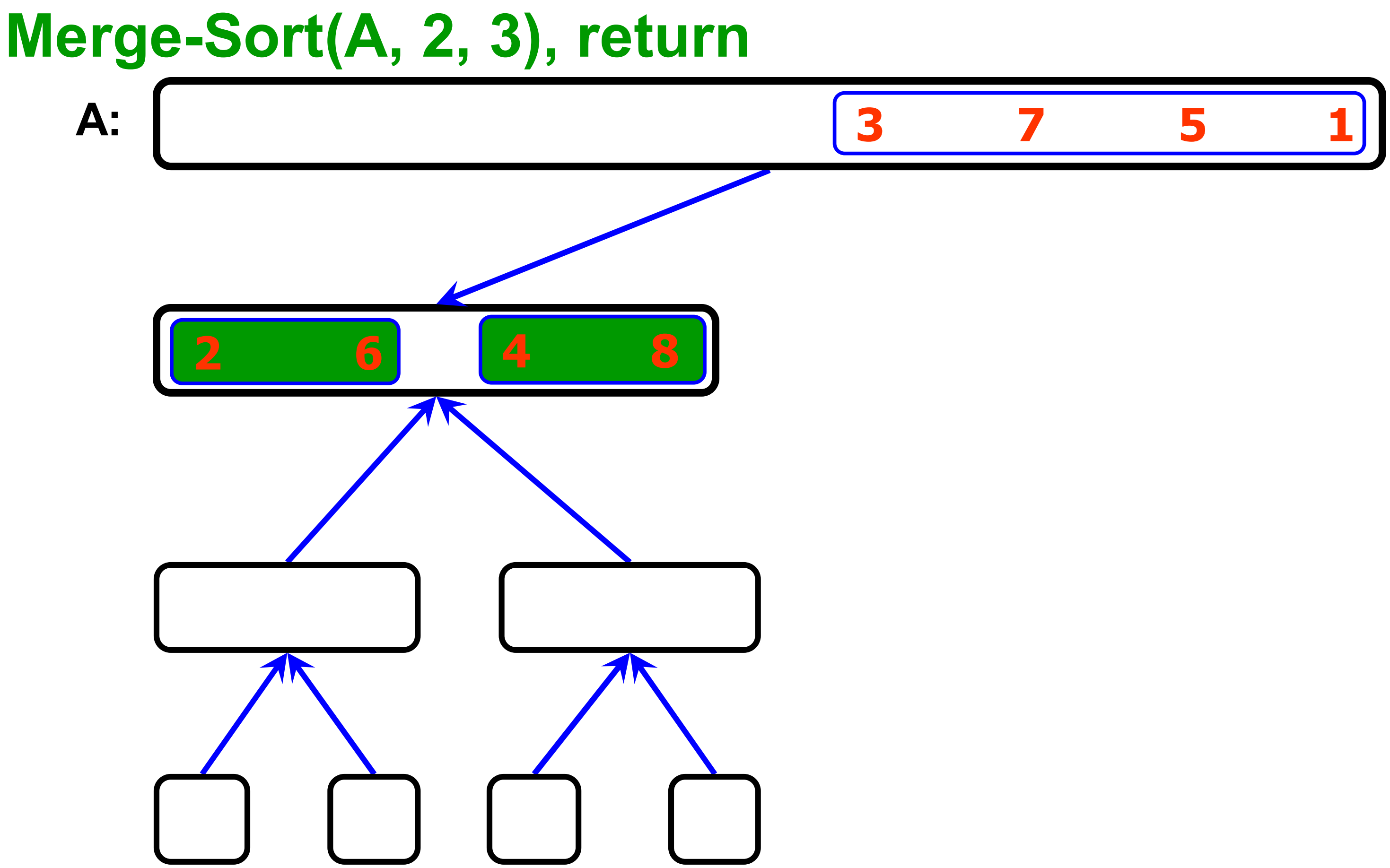
Merge-Sort(A, 0, 7)



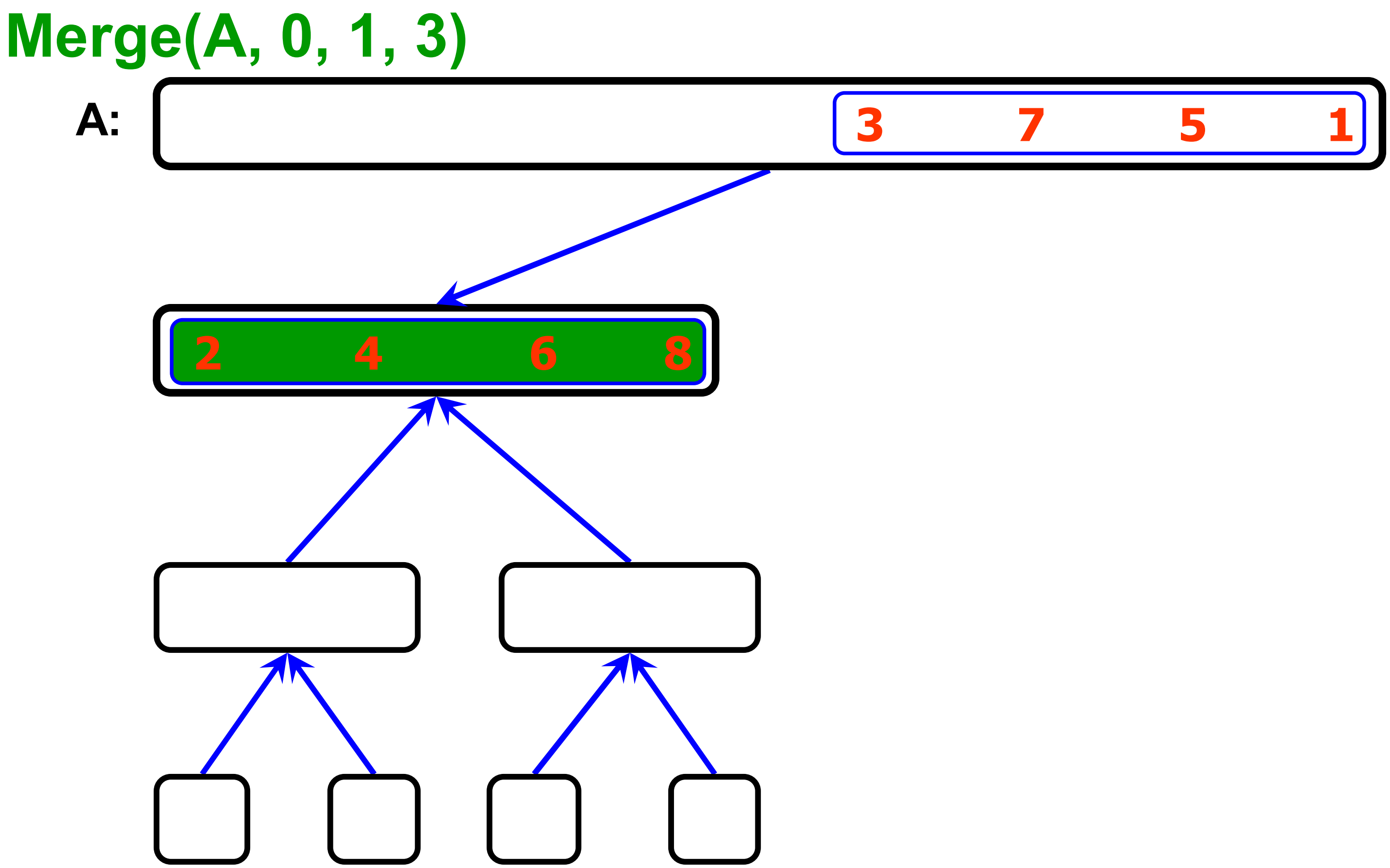
Merge-Sort(A, 0, 7)



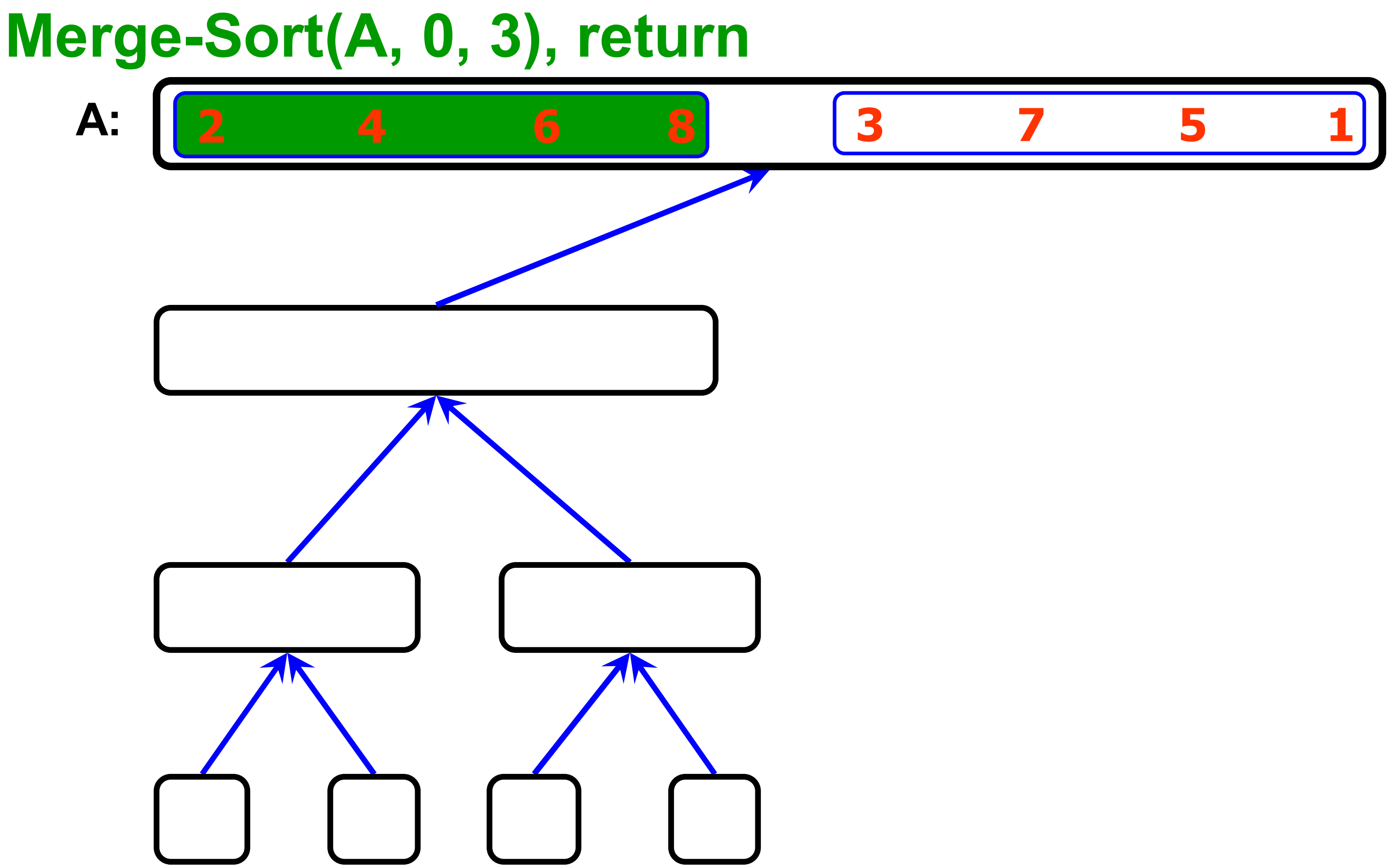
Merge-Sort(A, 0, 7)



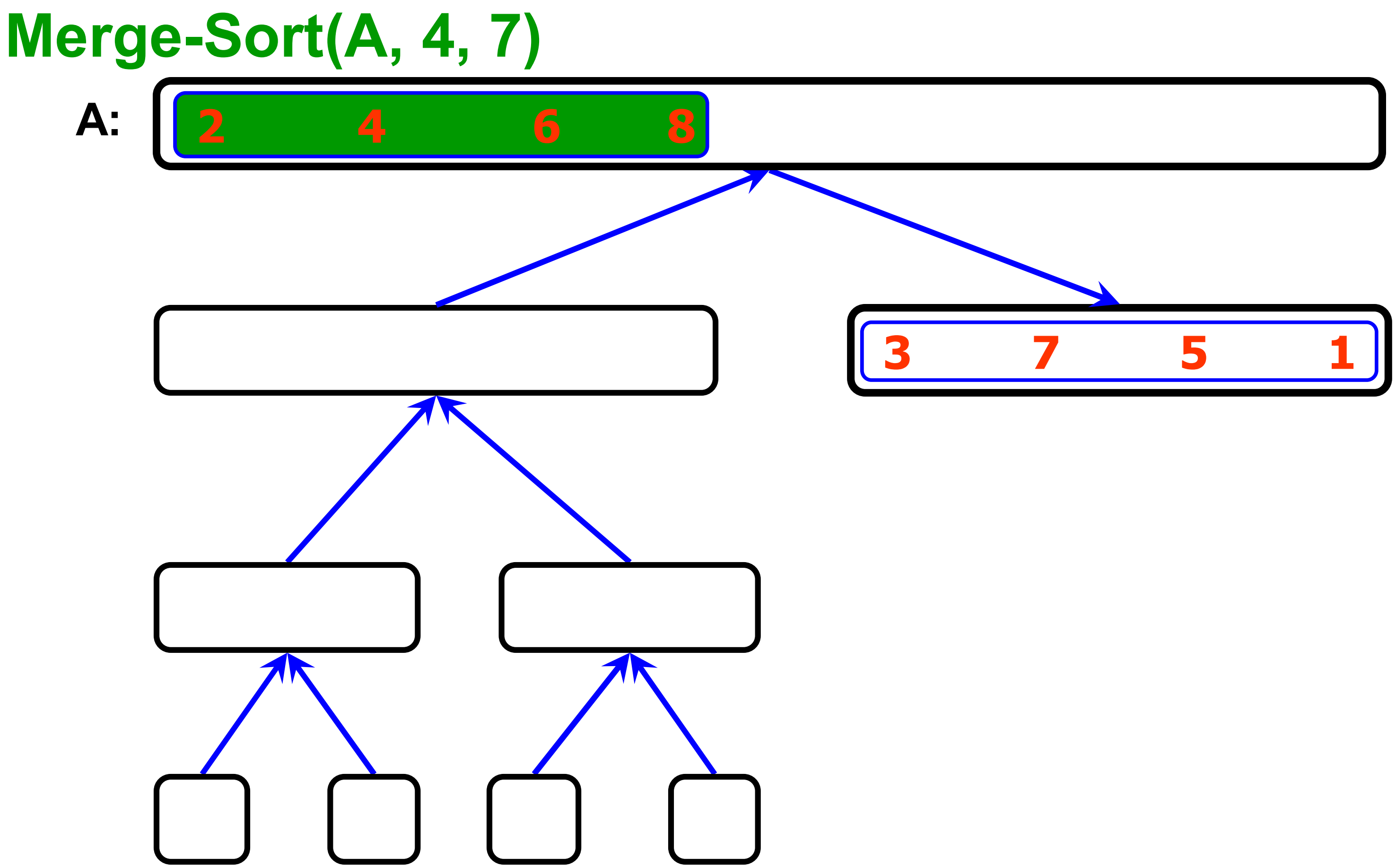
Merge-Sort(A, 0, 7)



Merge-Sort(A, 0, 7)

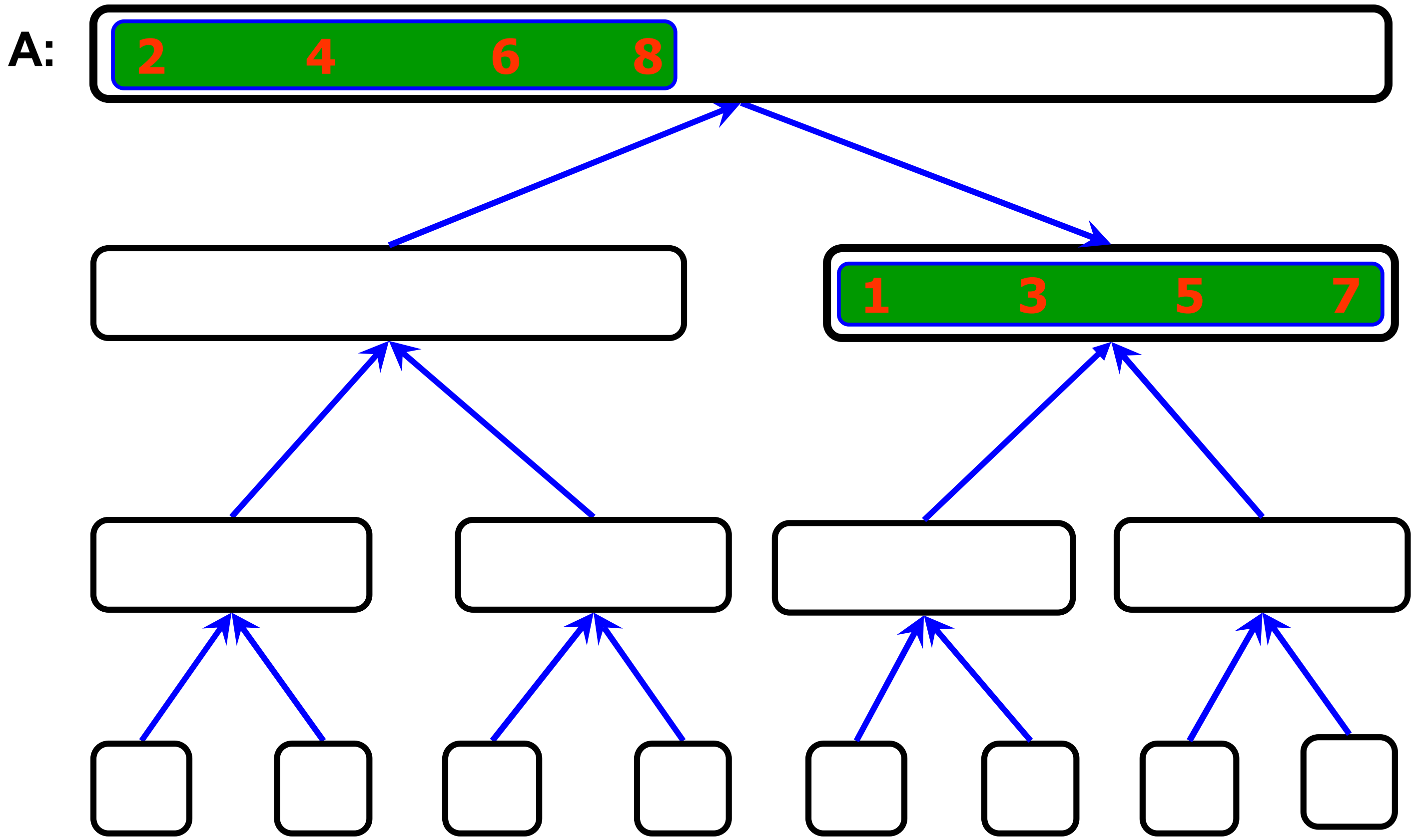


Merge-Sort(A, 0, 7)

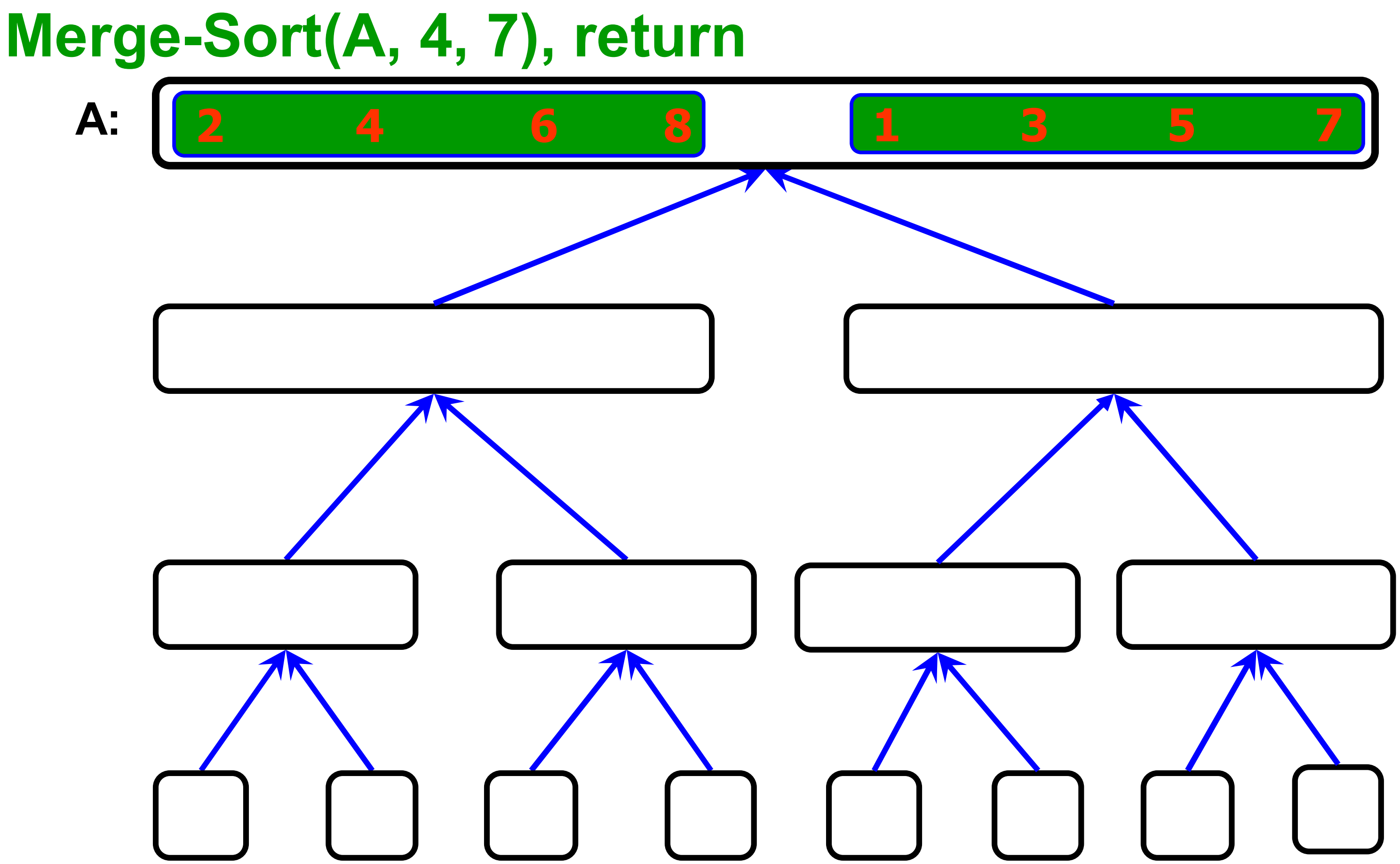


Merge-Sort(A, 0, 7)

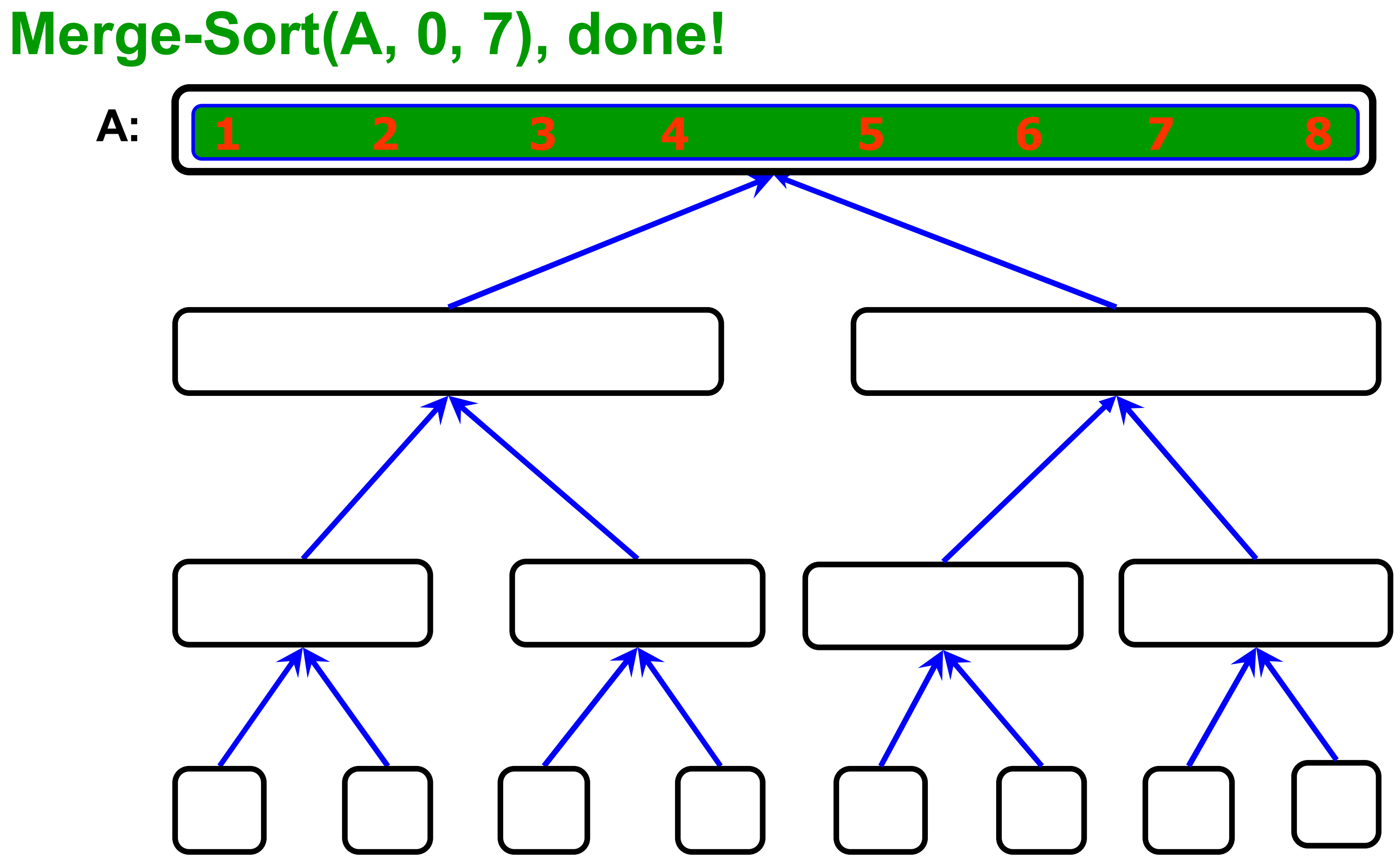
Merge (A, 4, 5, 7)



Merge-Sort(A, 0, 7)

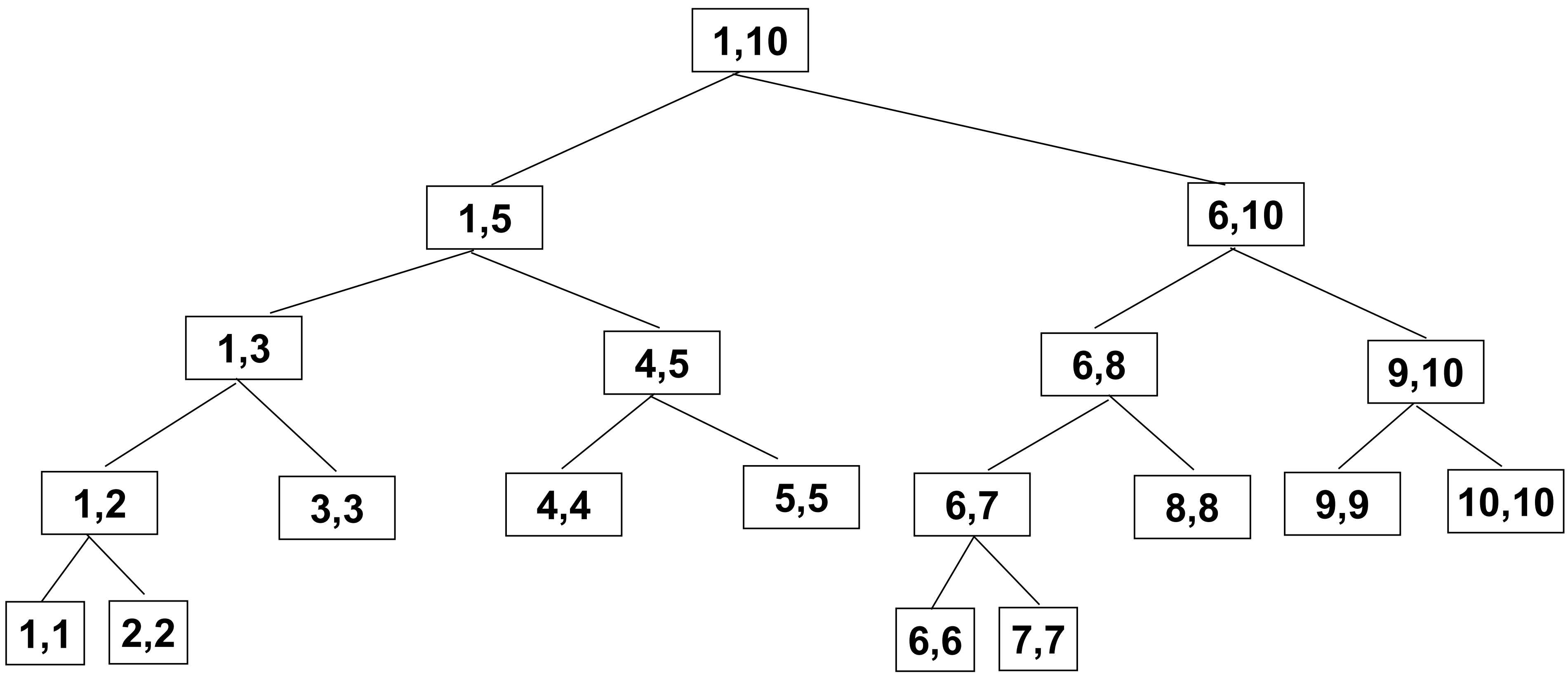


Merge-Sort(A, 0, 7)



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

Tree of calls of merge sort

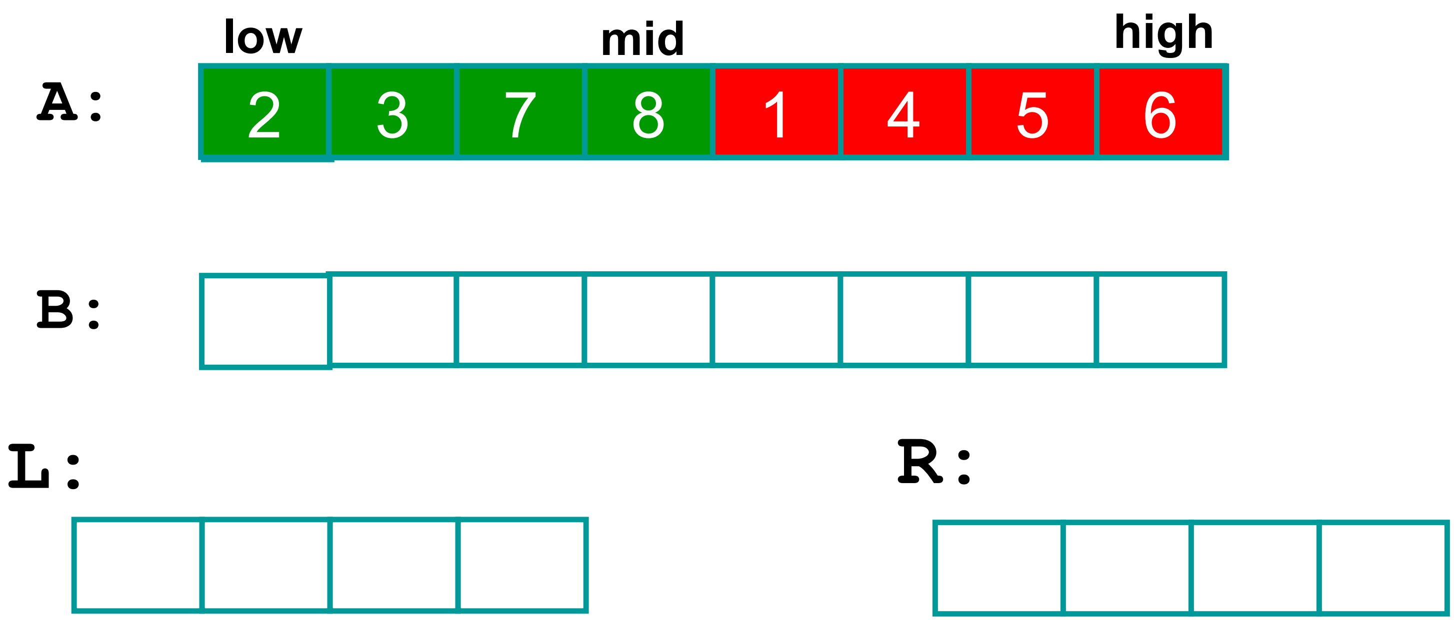


Merge Sort: Algorithm

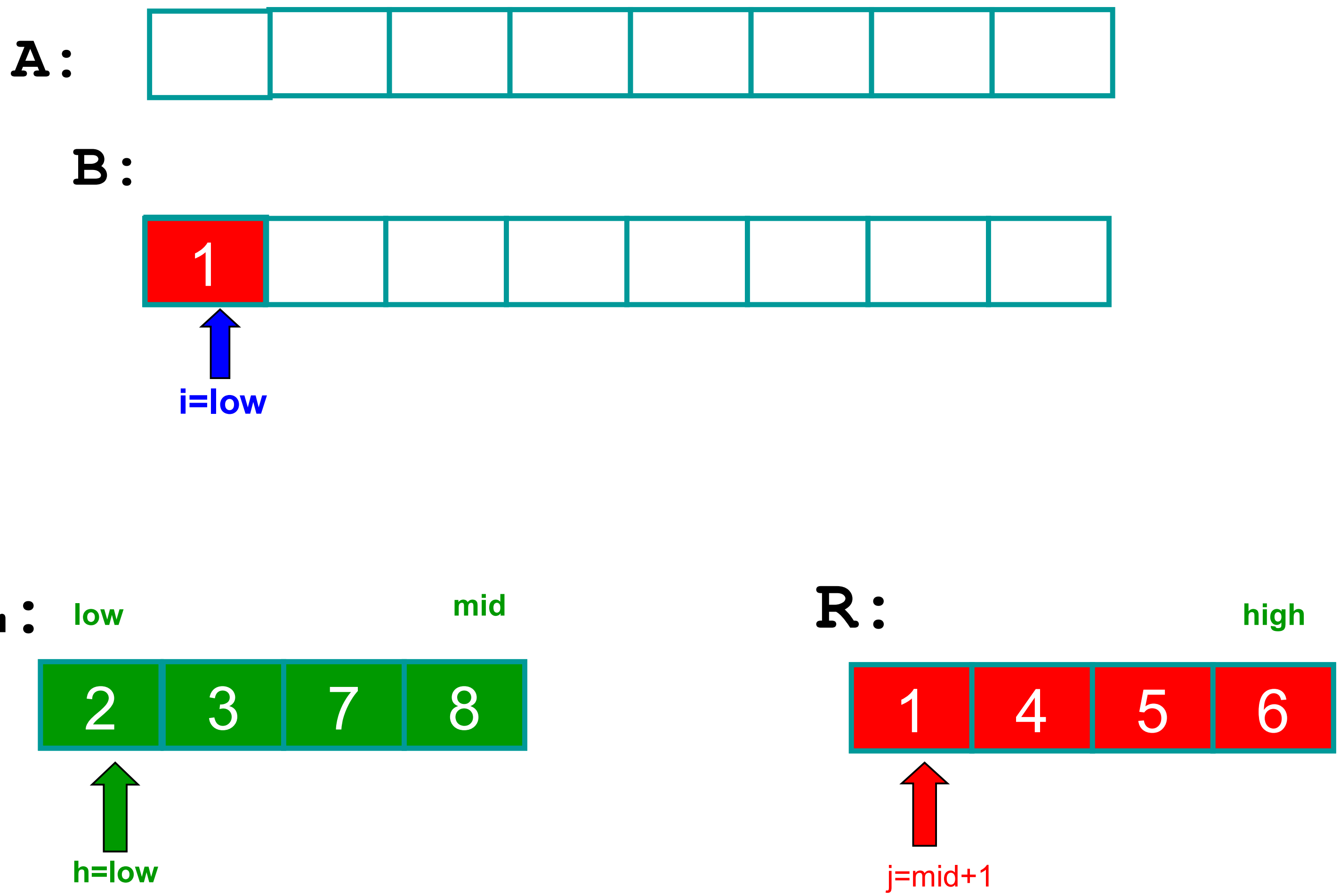
```
MergeSort ( low,high)
// sorts the elements a[low],…,a[high]  which are in the global array
//a[1:n] into ascending order ( increasing order ).
// Small(p) is true if there is only one element  to sort. In this case the list is
// already sorted.
{   if  (  low<high  ) then  //  if there are more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

Recursive Calls

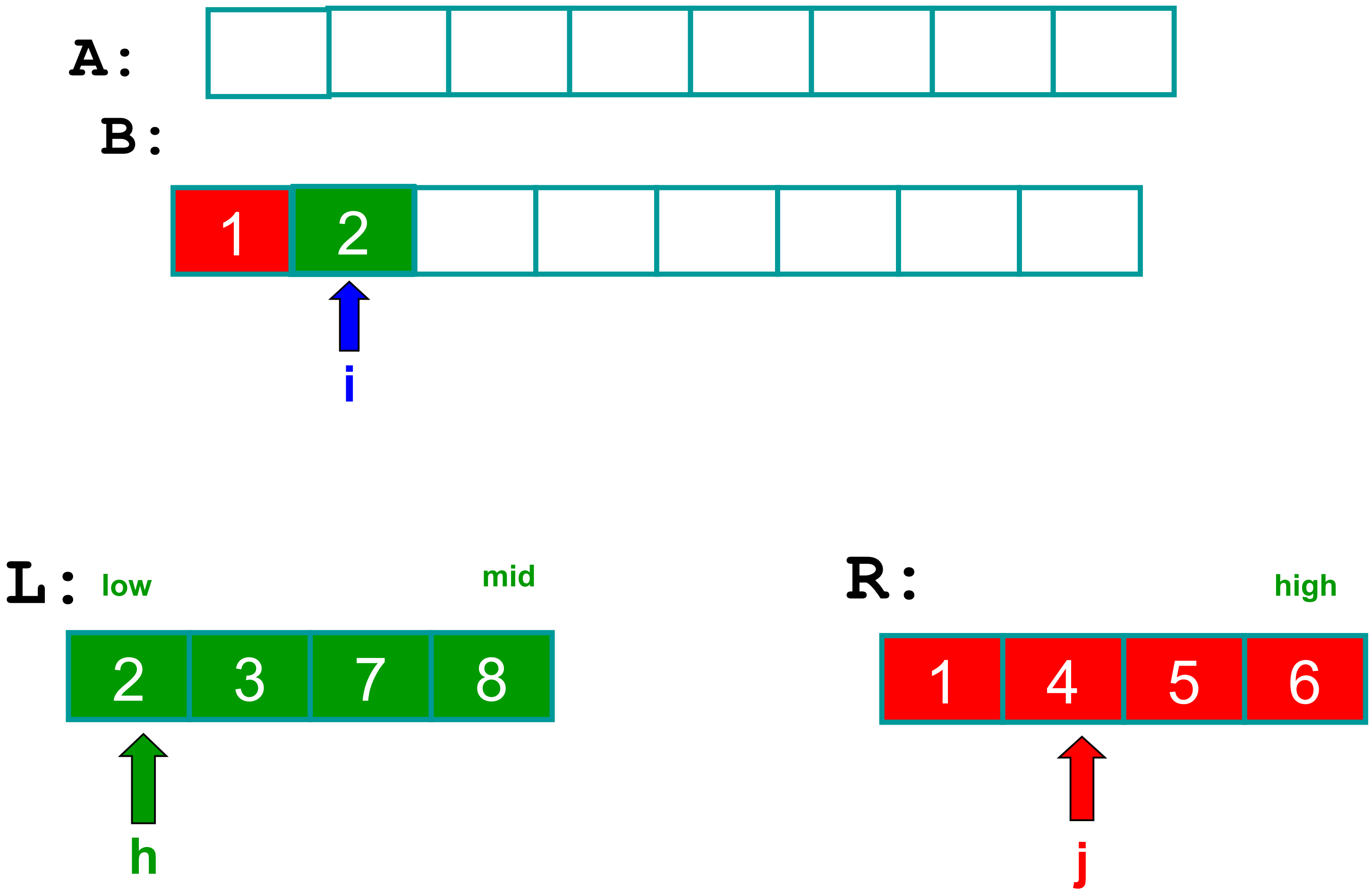
Merge-Sort: Merge Example



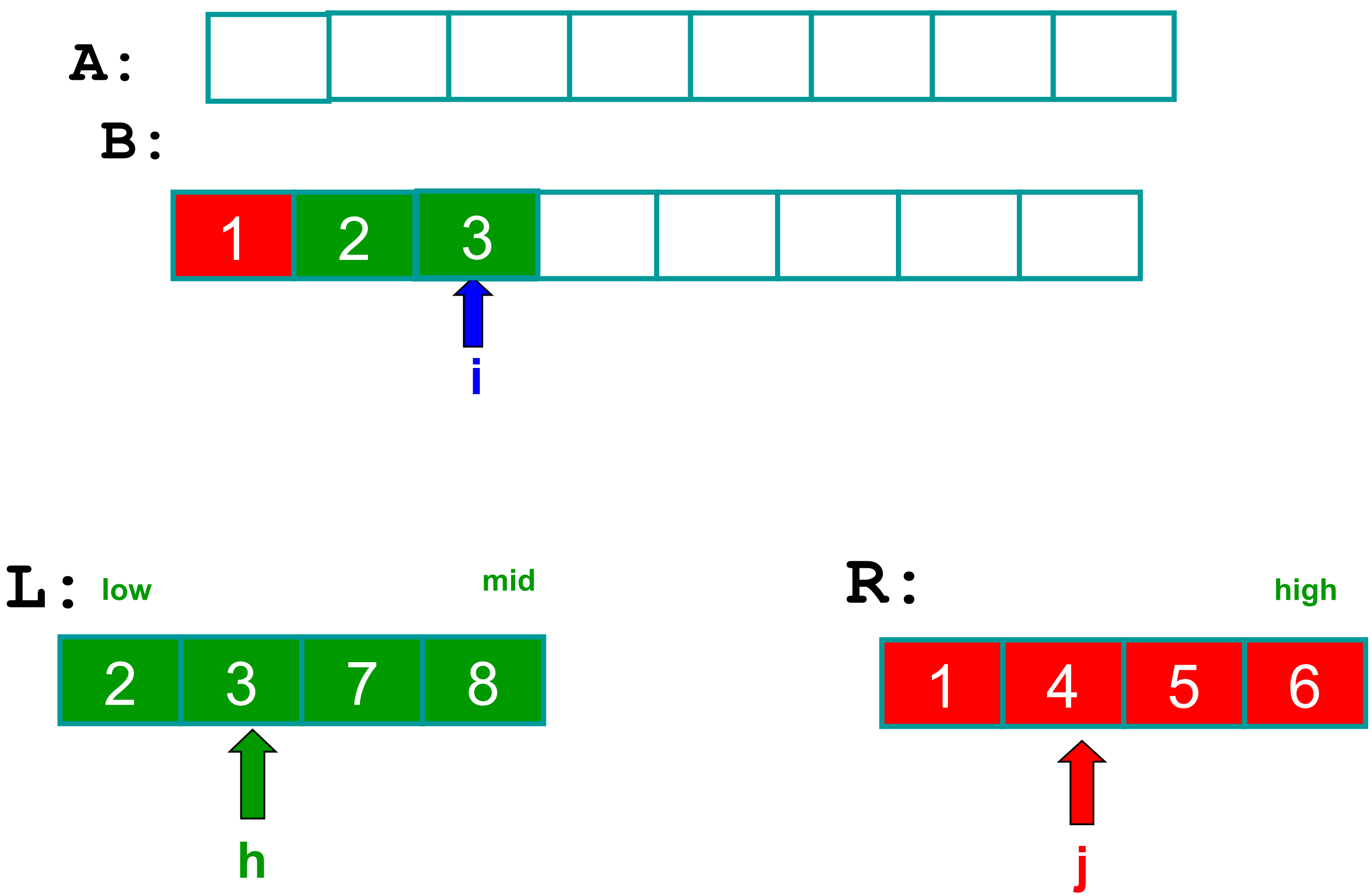
Merge-Sort: Merge Example



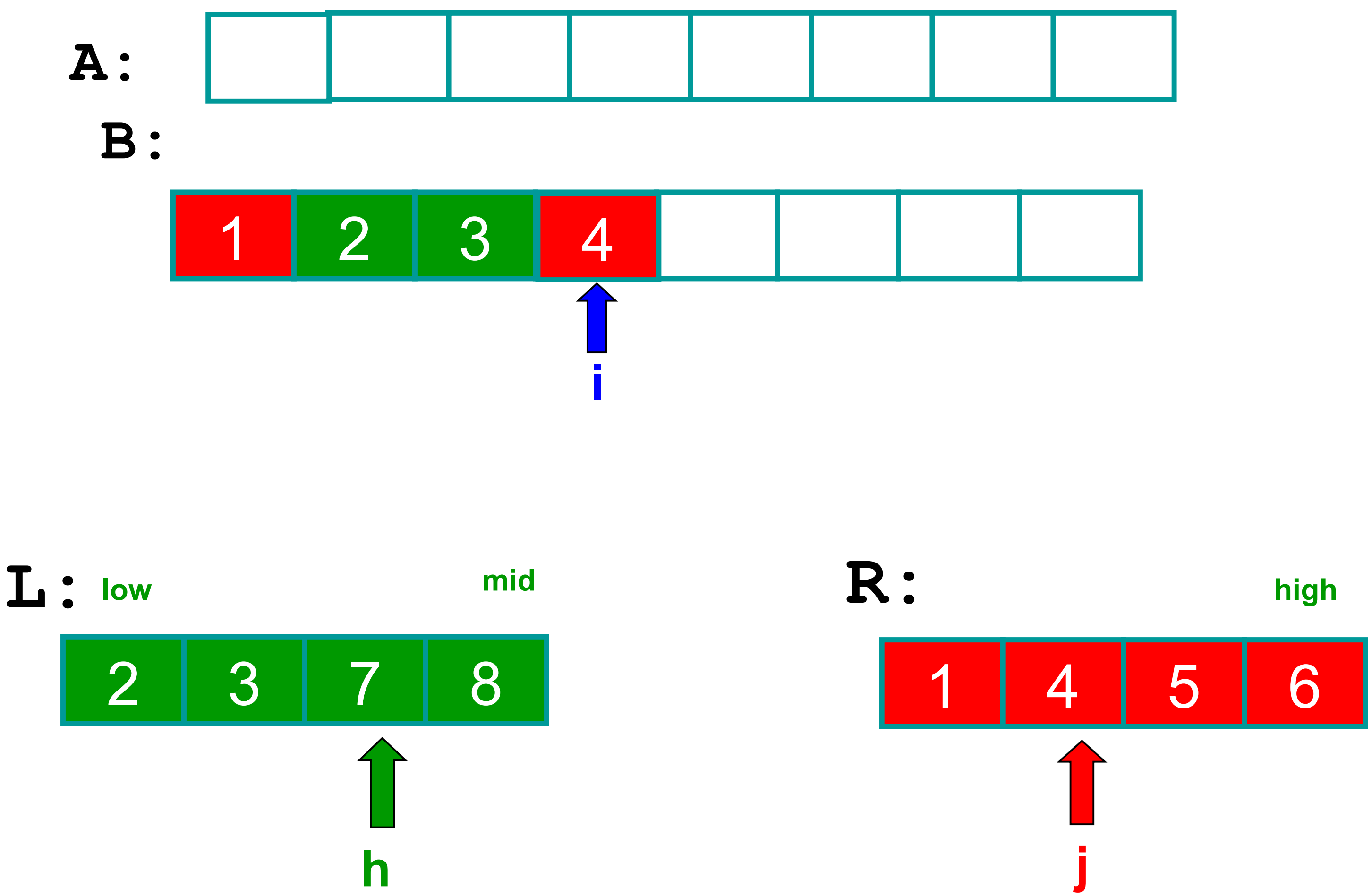
Merge-Sort: Merge Example



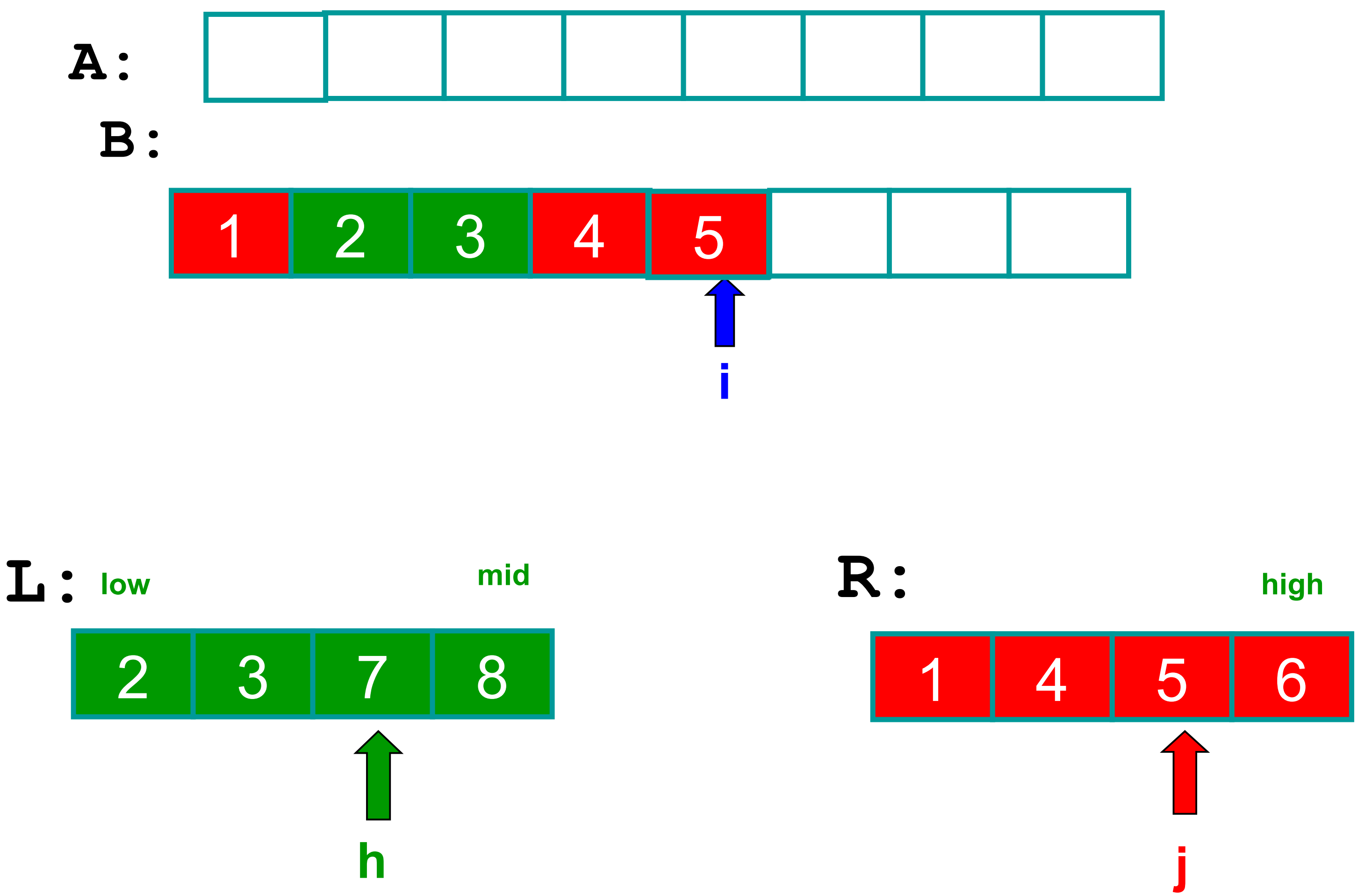
Merge-Sort: Merge Example



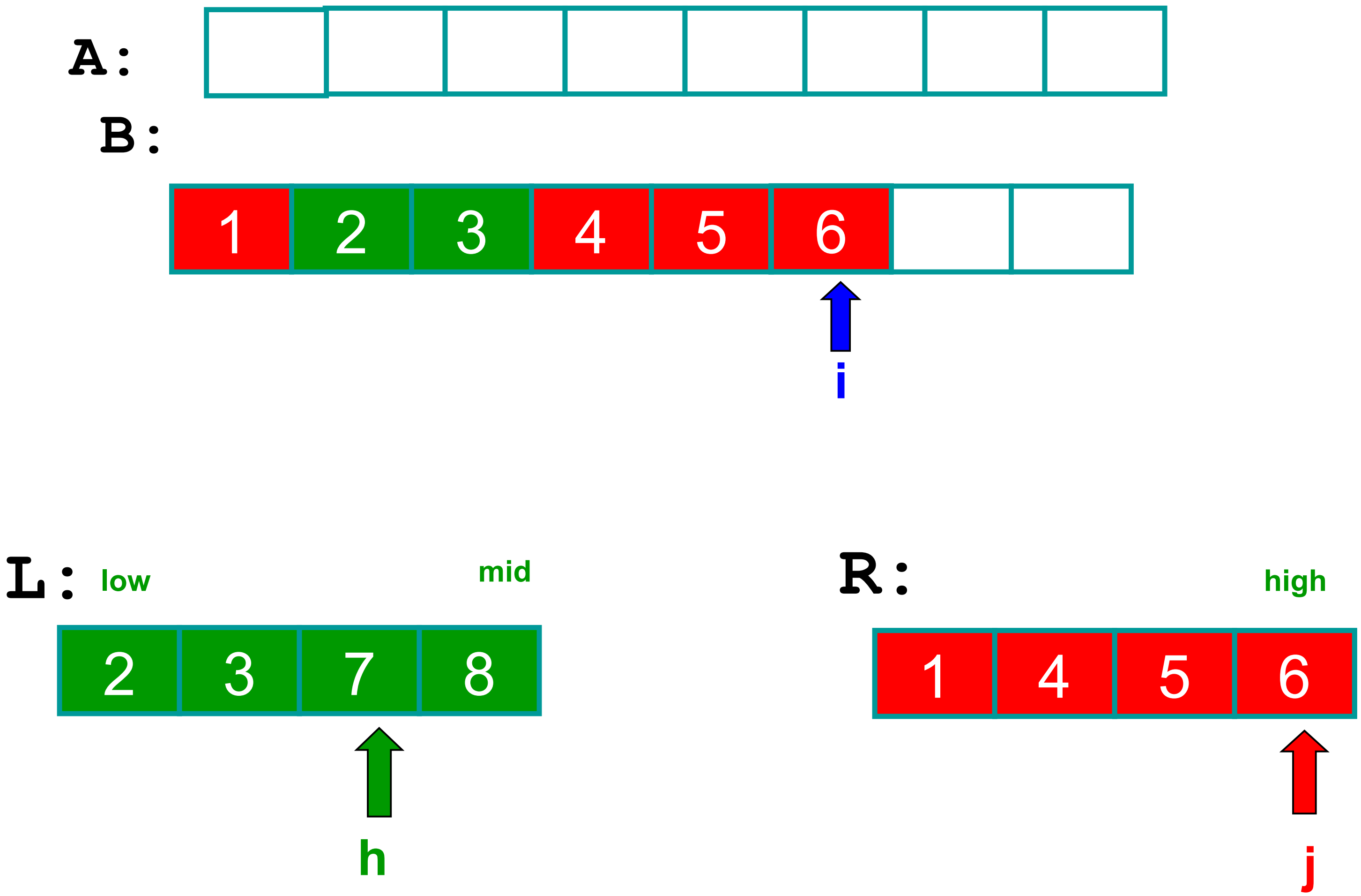
Merge-Sort: Merge Example



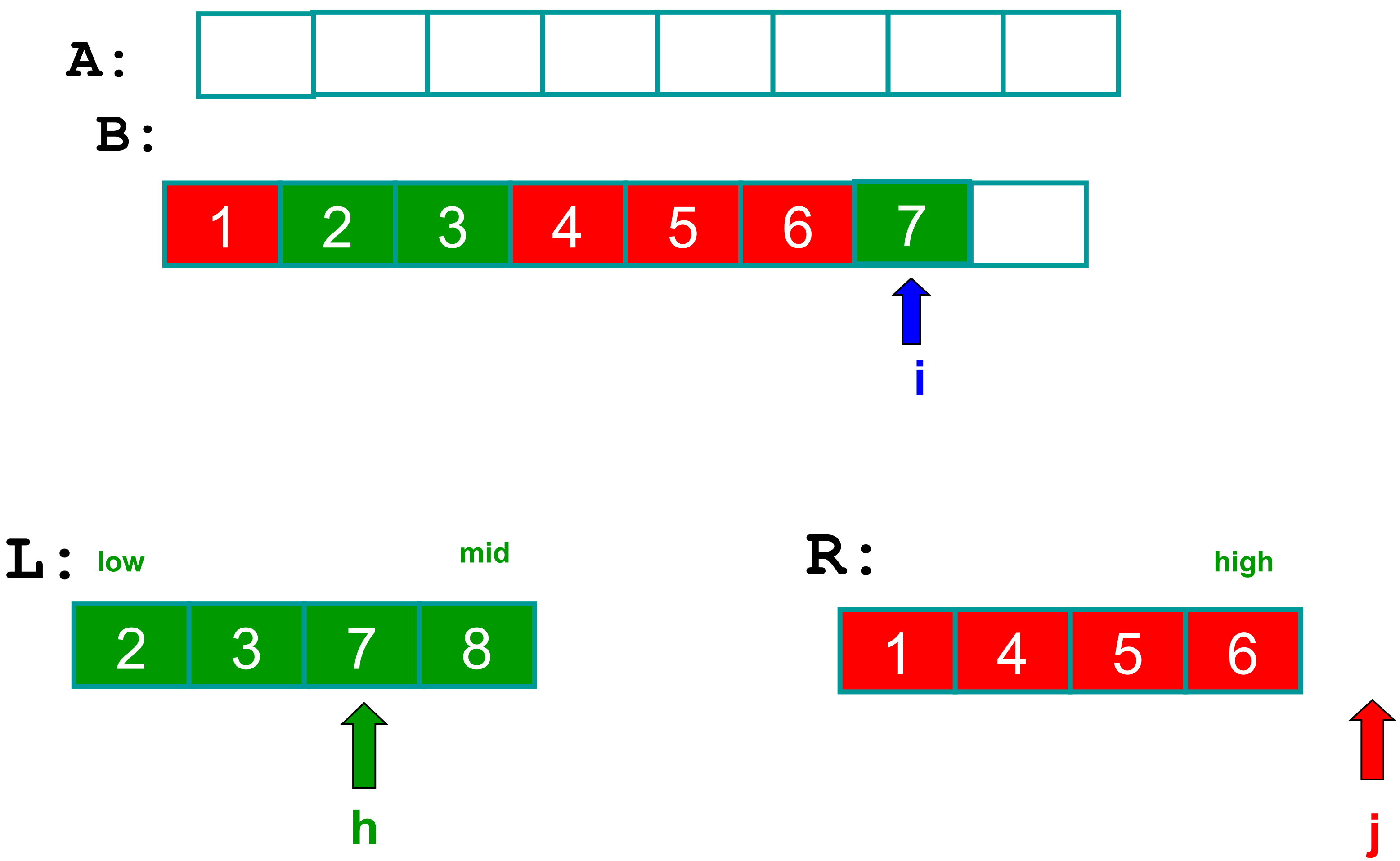
Merge-Sort: Merge Example



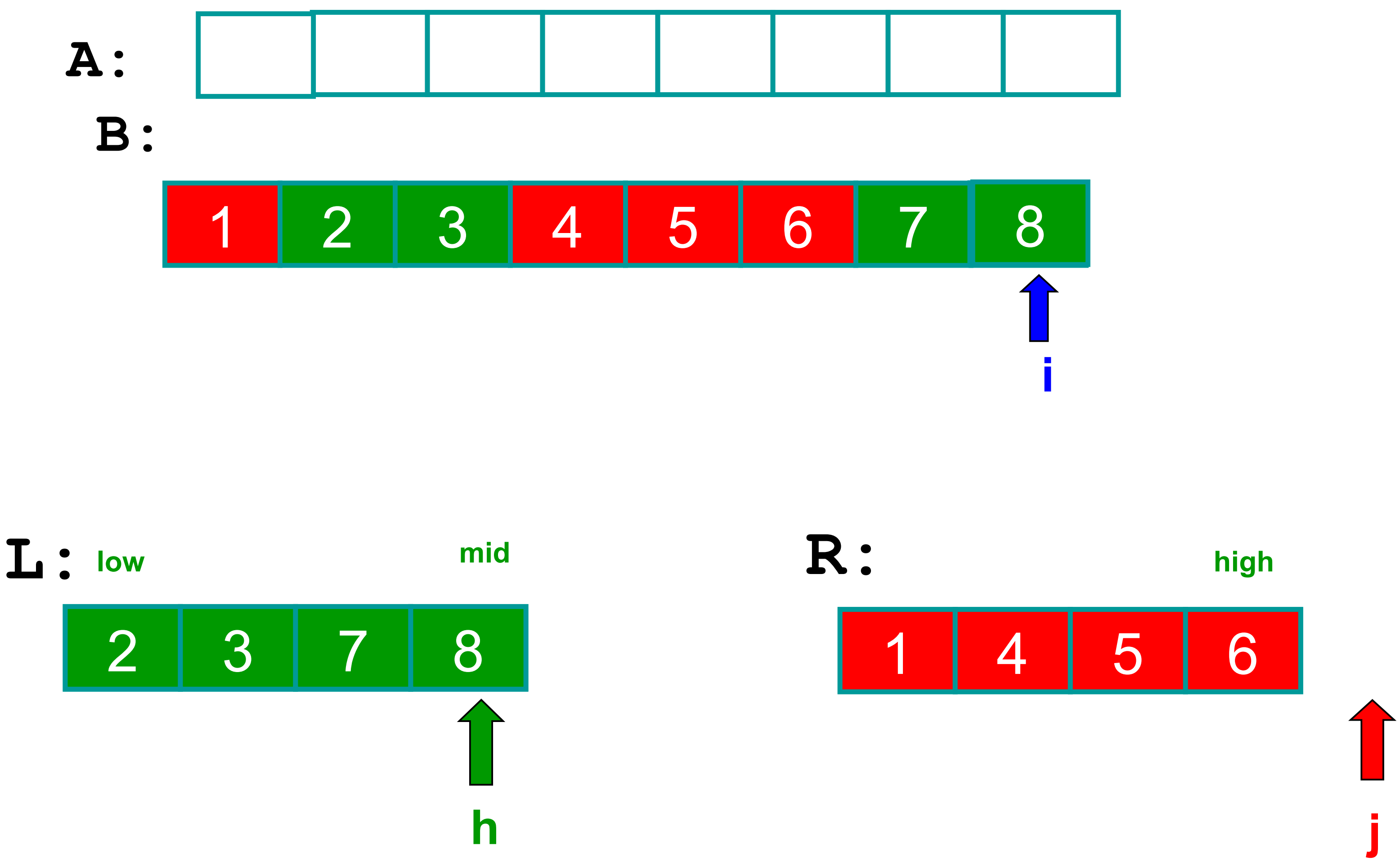
Merge-Sort: Merge Example



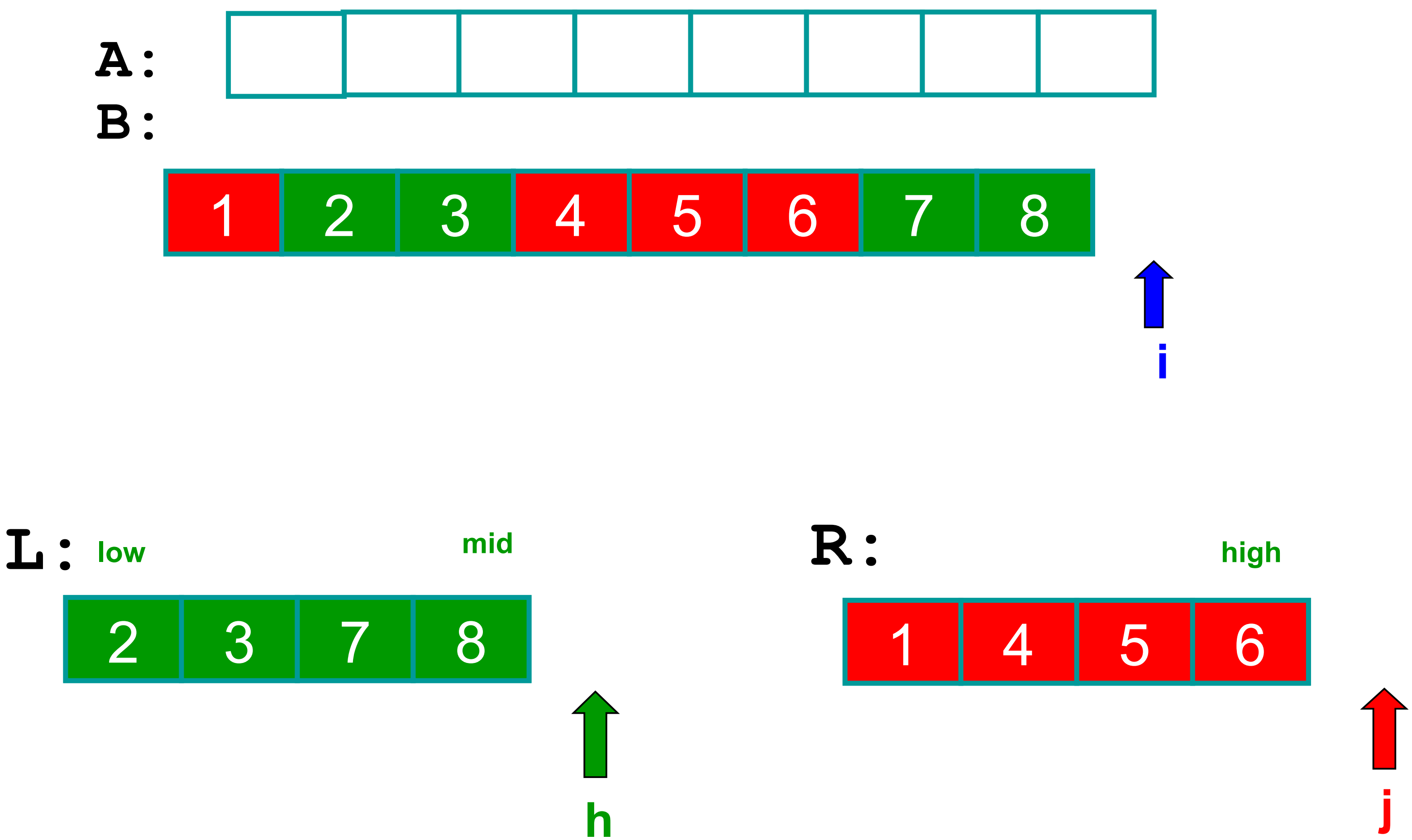
Merge-Sort: Merge Example

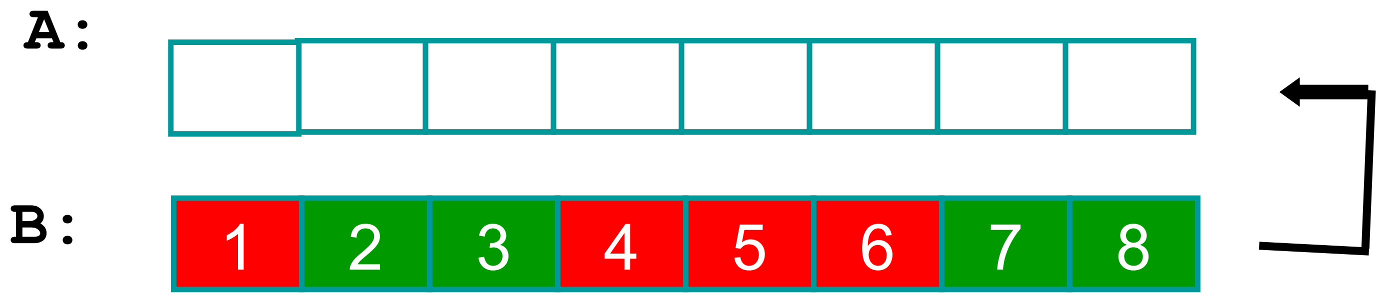


Merge-Sort: Merge Example



Merge-Sort: Merge Example

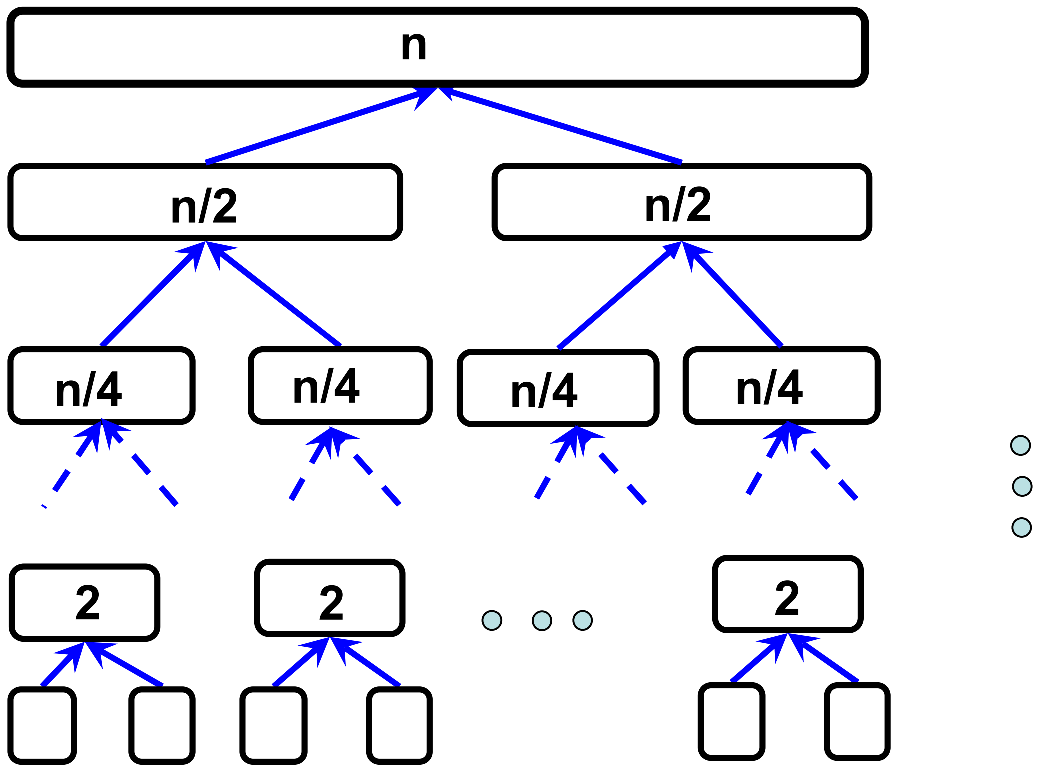



A:
B:
1
2
3
4
5
6
7
8

```
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.
{
    h:=low; i:=low; j:=mid+1;
        while( h ≤ mid ) and ( j ≤ high ) do
        {
                if( a[h] ≤ a[j] ) then
                {
                        b[i]:=a[h]; h:=h+1;
                }
                else
                {
                        b[i]:=a[j]; j:=j+1;
                }
                i:=i+1;
        }
```

```
        if( h > mid ) then
                for k:=j to high do
                {
                        b[i] := a[k]; i:= i+1;
                }
        else
                for k:=h to mid do
                {
                        b[i] := a[k]; i:= i+1;
                }
         for k:= low  to high do  a[k]:=b[k];
}
```

Merge-Sort Analysis



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

$$\begin{aligned} T(n) &= 2T(n/2) + c_2 n \\ &= 2(2T(n/4)+c2n/2)+cn \\ &= 4T(n/4)+2c_2 n \\ &\ \ \ldots\ldots \\ &\ \ \ldots\ldots \\ &= 2^k\, T(1)+ kc_2 n \\ &= c_1 n+c_2 n\log n = = O(n\log n) \end{aligned}$$

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

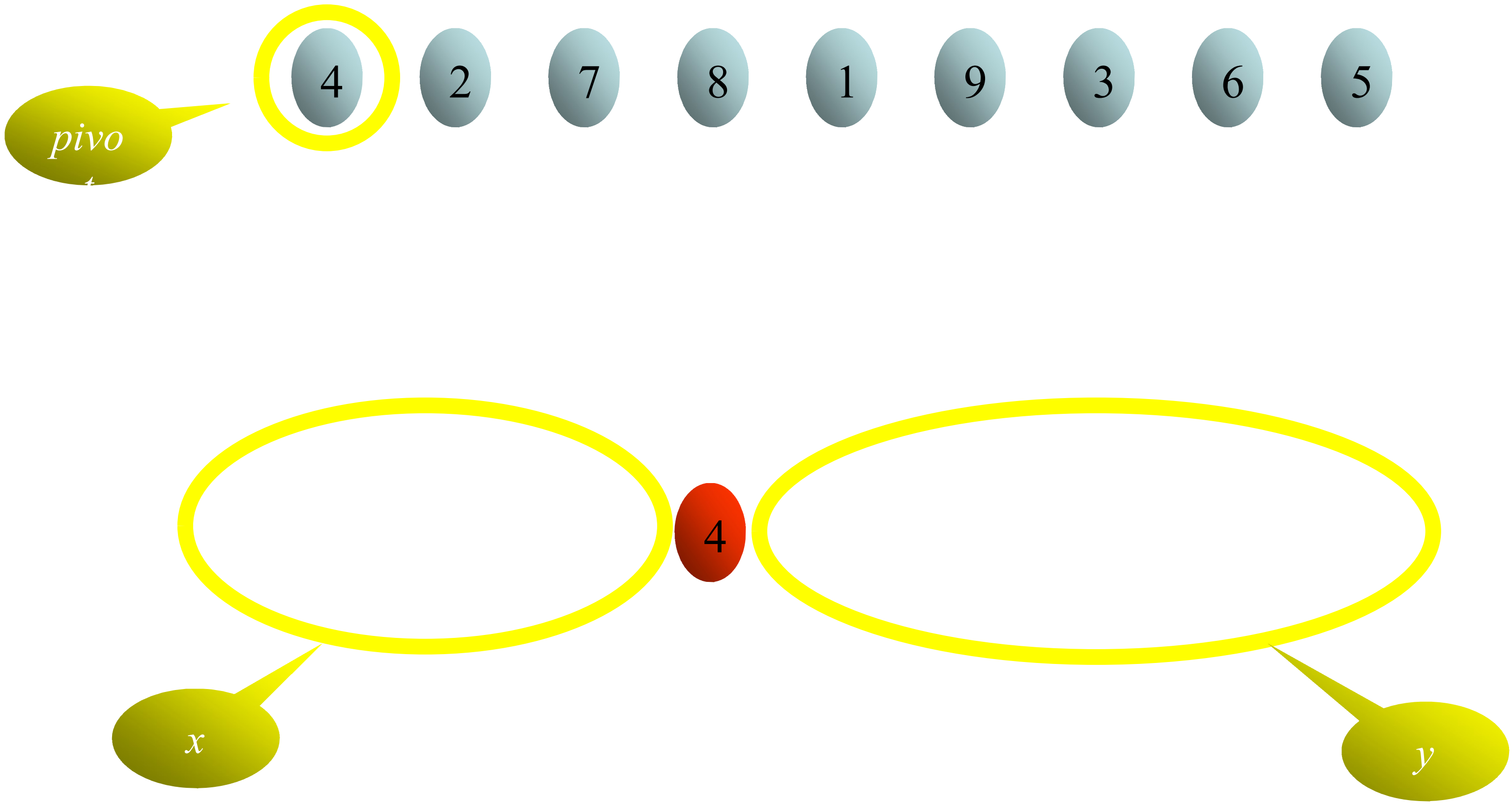
- **Divide**:
  - Pick any element as the **pivot**, e.g, the first element
  - Partition the remaining elements into

    FirstPart, which contains all elements < pivot

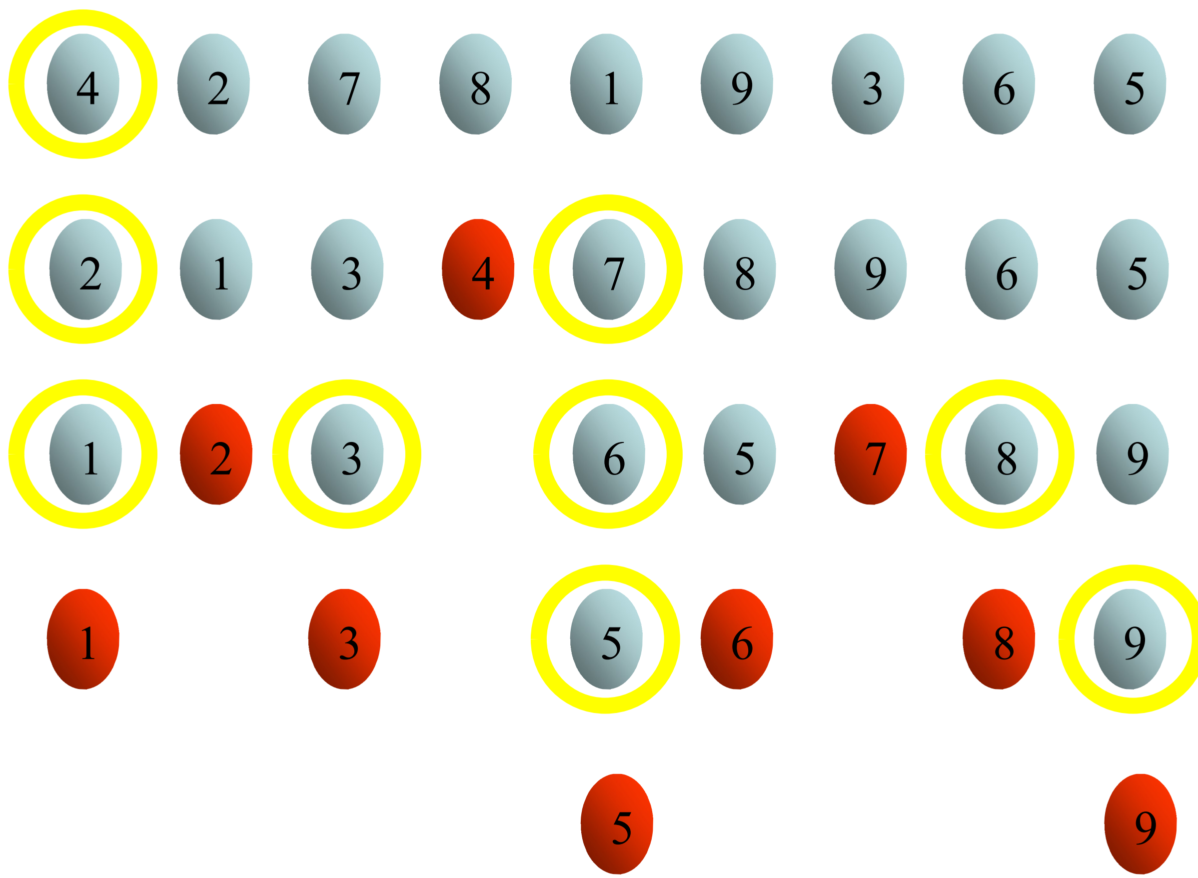
    SecondPart, which contains all elements > pivot

- **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.



The whole process



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 $i \geq j$ interchange j^{th} and pivot elements and then divide the list into two sublists.

35 24 63 45 17 85 95 98

j

Two sublists:

35 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[i] ; 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;
}
```

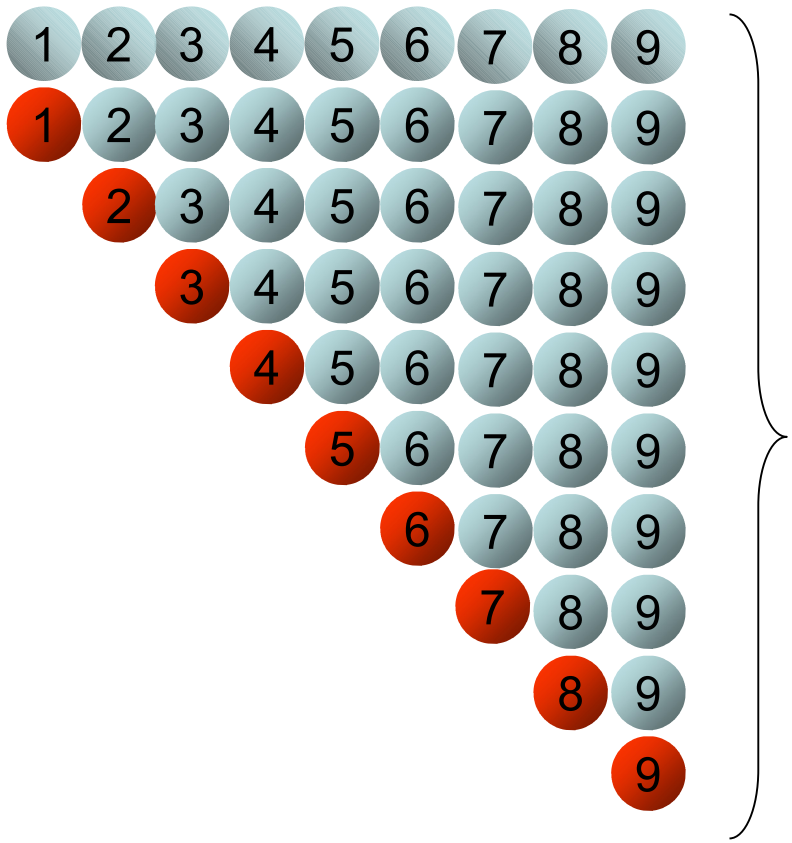
Time complexity analysis

- The time required to sort n elements using quicksort involves 3 components.
  - Time required for partitioning the array, which is *roughly proportional to n*.
  - Time required for sorting lower subarray.
  - Time required for sorting upper subarray.
- Assume that there are k elements in the lower subarray.
- Therefore,

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

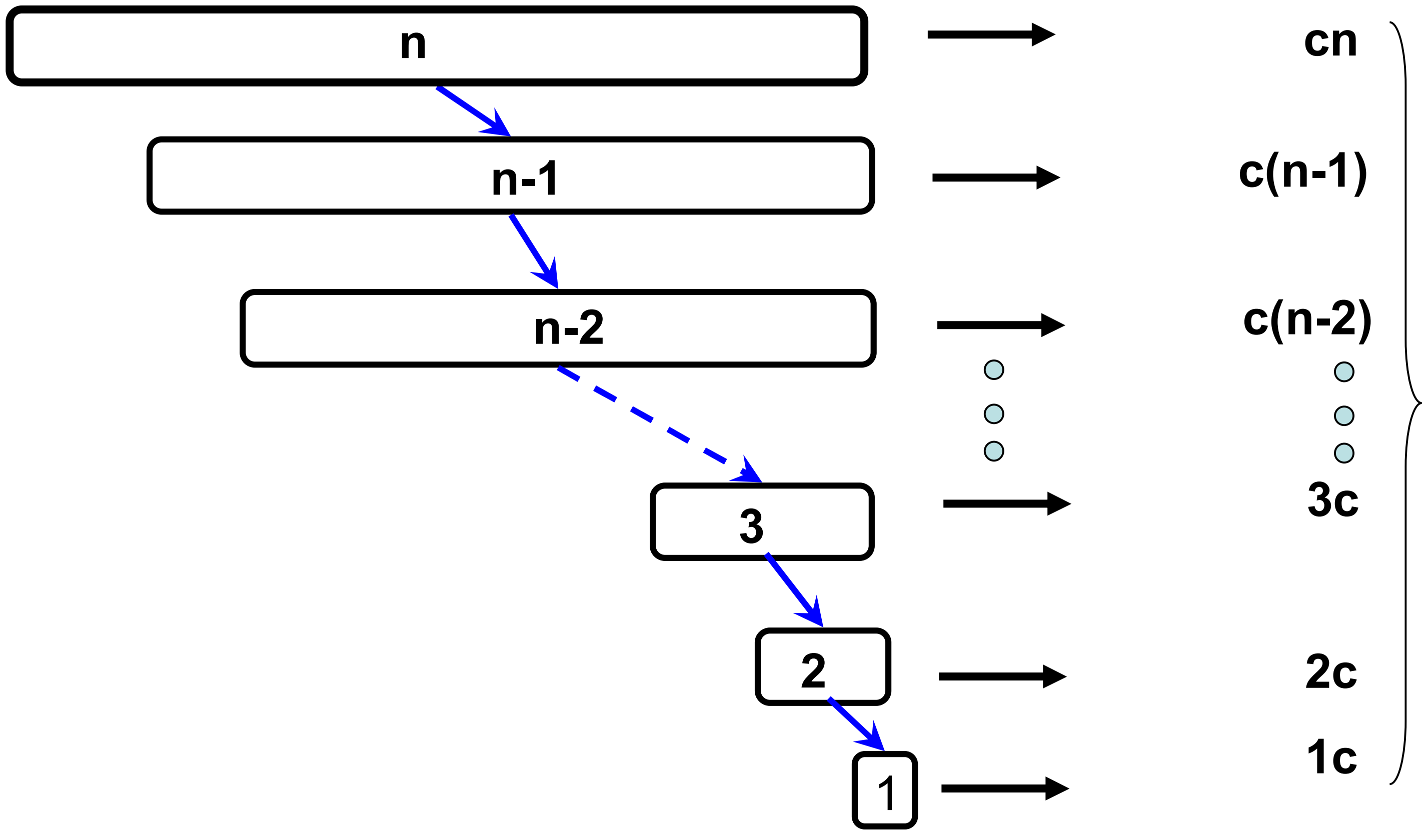
A worst/bad case

It occurs if the list is already in sorted order



$O(n^2)$

Worst/bad Case



contd...

- In the worst case, the array is always partitioned into two subarrays in which one of them is always empty. Thus , for the worst case analysis,

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

$$= T(n-1) + c_2 n$$

$$= T(n-2) + c_2 (n-1) + c_2 n$$

$$= T(n-3) + c_2 (n-2) + c_2 (n-1) + c_2 n$$

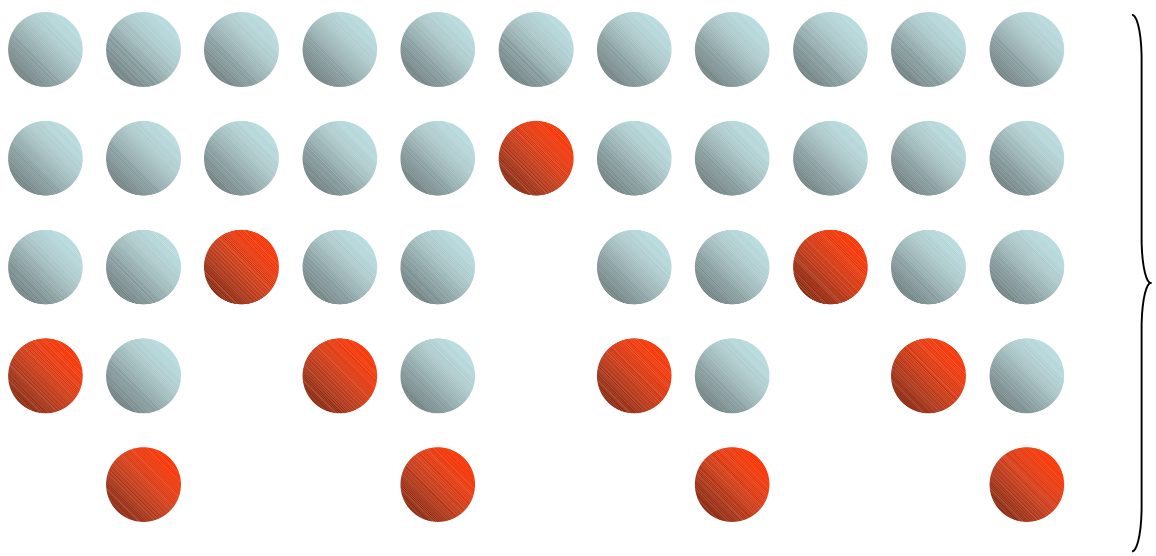
.....

.....

$$= n(n+1)/2 = (n^2+n)/2 = O(n^2)$$

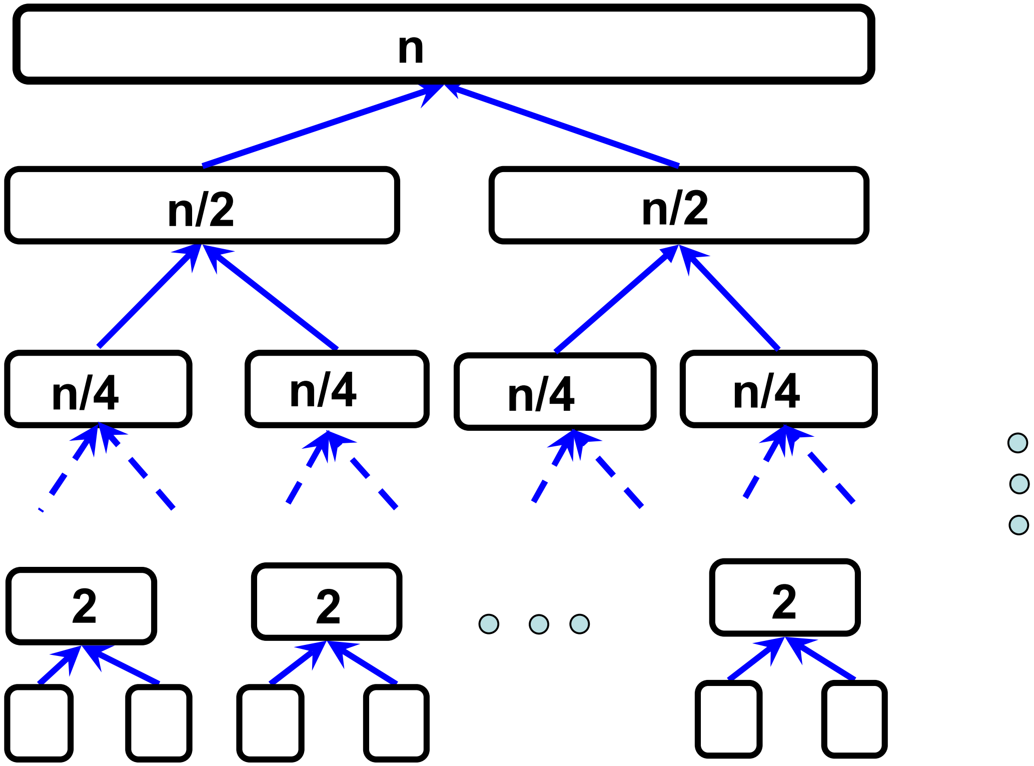
A Best/Good case

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



O(n logn)

Best/Good Case



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

Assume $n=2^k$, then

$$\begin{aligned} T(n) &= 2T(n/2) + c_2 n \\ &= 2(2T(n/4)+c2n/2)+cn \\ &= 4T(n/4)+2c_2 n \\ &\ \ldots\ldots \\ &\ \ldots\ldots \\ &= 2^k\, T(1) + kc_2 n \\ &= c_1 n + c_2 n\log n = = O(n\log n) \end{aligned}$$

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.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:

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

Then,

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

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

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

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

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 5 & 6 & 6 \\ 5 & 5 & 6 & 6 \\ 7 & 7 & 8 & 8 \\ 7 & 7 & 8 & 8 \end{bmatrix}$$

A_{11} A_{12} B_{11} B_{12} A_{21} A_{22} B_{21} B_{22}

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

$M_1 = (A_{11} + A_{22}) * (B_{11} + \mathbf{B_{22}})$
$M_2 = (A_{21} + A_{22}) * B_{11}$
$M_3 = A_{11} * (B_{12} - \mathbf{B_{22}})$
$M_4 = A_{22} * (B_{21} - \mathbf{B_{11}})$
$M_5 = (A_{11} + A_{12}) * \mathbf{B_{22}}$
$M_6 = (A_{21} - A_{11}) * (B_{11} + \mathbf{B_{12}})$
$M_7 = (A_{12} - A_{22}) * (B_{21} + \mathbf{B_{22}})$

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

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

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

The resulting recurrence relation for T(n) is

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

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

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

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

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

$$= c_1 n^{\log_2 7} + c_2 n^{\log_2 4} (n^{\log_2 7 - \log_2 4})$$

$$= c_1 n^{\log_2 7} + c_2 (n^{\log_2 4 + \log_2 7 - \log_2 4}) = c_1 n^{\log_2 7} + c_2 n^{\log_2 7}$$

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