Sorting Algorithms

Bubble Sort
Selection Sort
Insertion Sort
Merge Sort
Quick Sort
Radix Sort

(Heap Sort will be discussed after trees)

Sorting

- The term sorting means arranging the elements of the array so that they are placed in some relevant order which may either be ascending order or descending order. That is, if A is an array then the elements of A are arranged in sorted order (ascending order) in such a way that, A[0] < A[1] < A[2] < …… < A[N]

- For example, if we have an array that is declared and initialized as,

  int A[] = {21, 34, 11, 9, 1, 0, 22};

- Then the sorted array (ascending order) can be given as, A[] = {0, 1, 9, 11, 21, 22, 34}

Sorting

- A sorting algorithm is defined as an algorithm that puts elements of a list in a certain order (that can either be numerical order, lexicographical order or any user-defined order). Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly. There are two types of sorting:

- **Internal sorting** which deals with sorting the data stored in computer's memory

- **External sorting** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in computer's memory.

Bubble Sort

- Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array (in case of arranging elements in ascending order).

- In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at lower index is greater than the element at the higher index, the two elements are interchanged so that the smaller element is placed before the bigger one. This process is continued till the list of unsorted elements exhaust.

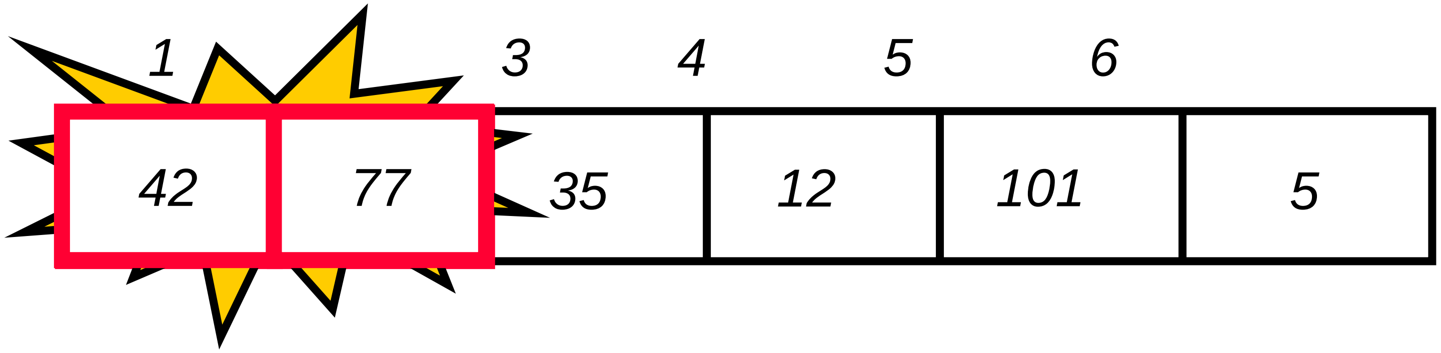
"Bubbling Up" the Largest Element

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

1	2	3	4	5	6
77	*42*	*35*	*12*	*101*	*5*

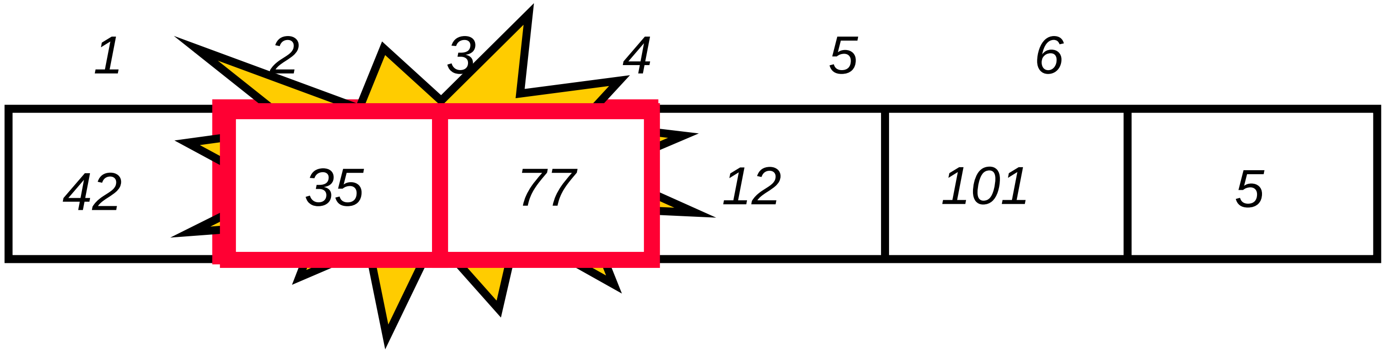
"Bubbling Up" the Largest Element

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



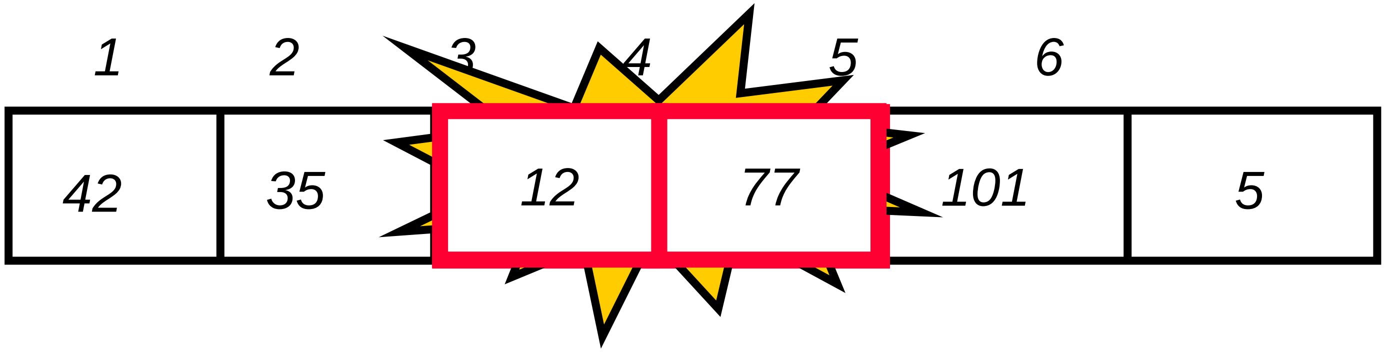
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



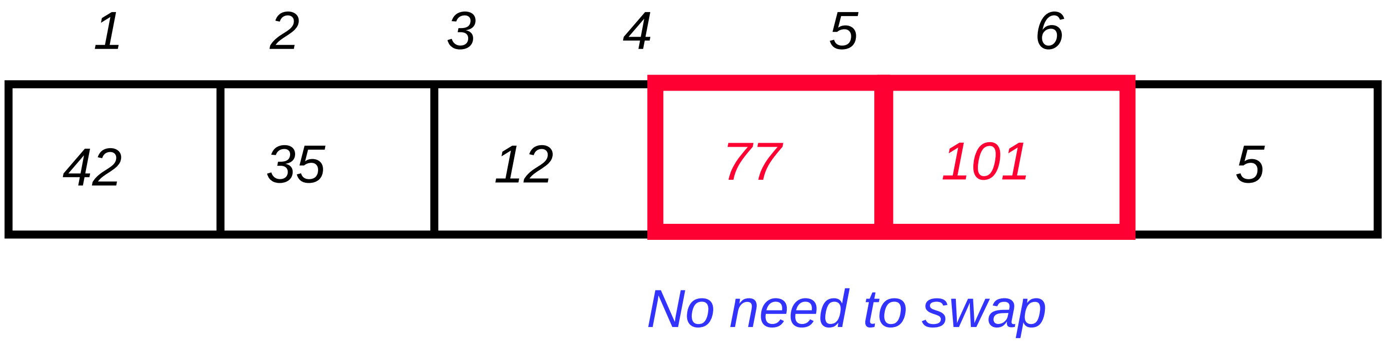
"Bubbling Up" the Largest Element

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



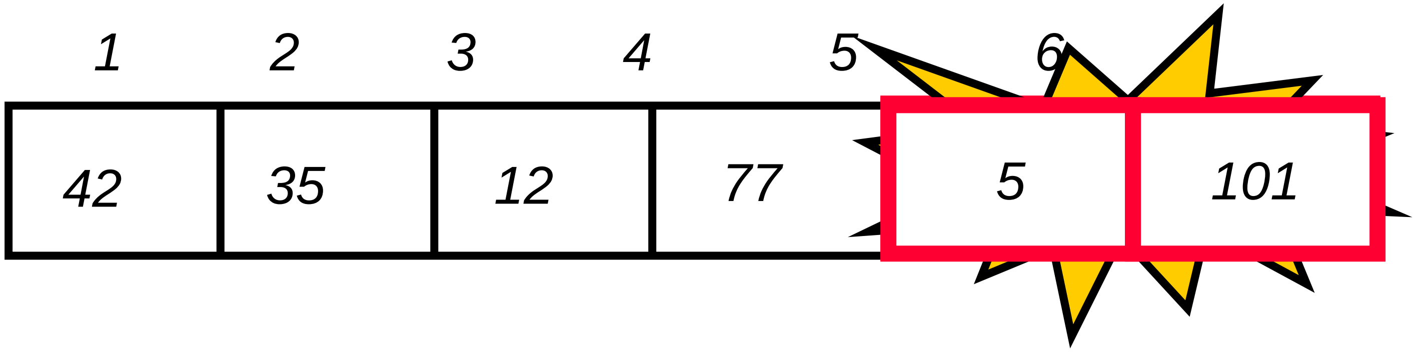
"Bubbling Up" the Largest Element

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



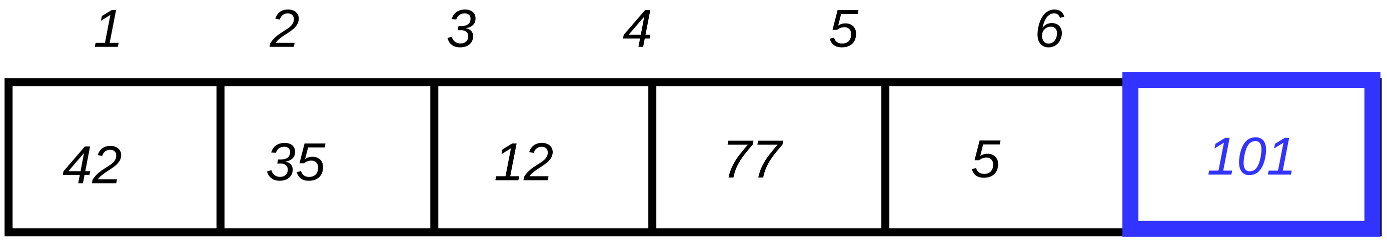
"Bubbling Up" the Largest Element

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



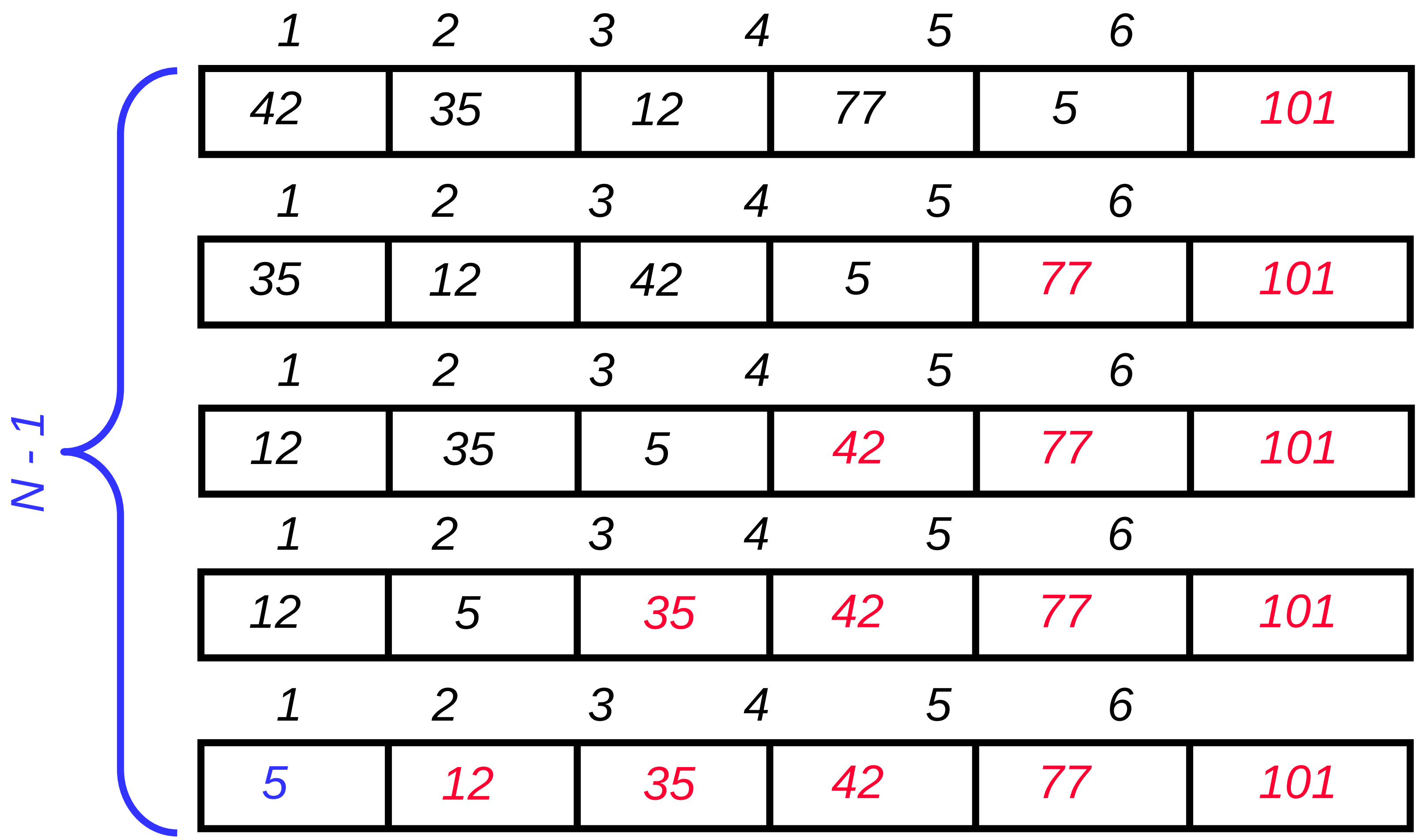
"Bubbling Up" the Largest Element

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



Largest value correctly placed

"Bubbling" All the Elements



1	2	3	4	5	6
42	35	12	77	5	101

1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

1	2	3	4	5	6
12	5	35	42	77	101

1	2	3	4	5	6
5	12	35	42	77	101

Reducing the Number of Comparisons

1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
42	35	12	77	5	101

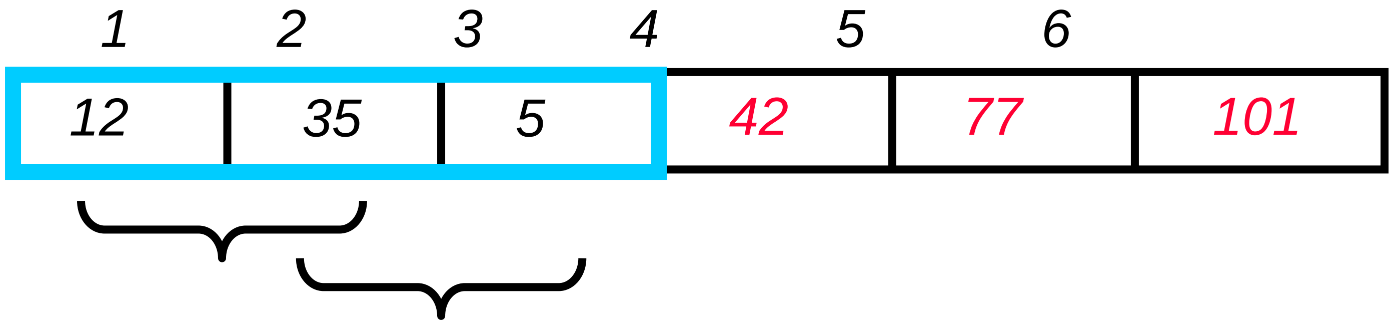
1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

1	2	3	4	5	6
12	5	35	42	77	101

Reducing the Number of Comparisons

- **On the Nth "bubble up", we only need to do MAX-N comparisons.**
- **For example:**
  - **This is the 4th "bubble up"**
  - **MAX is 6**
  - **Thus we have 2 comparisons to do**



Bubble Sort Example

A[] = {30, 52, 29, 87, 63, 27, 18, 54}

Pass 1:

a) Compare 30 and 52. Since30<52, then no swapping is done

b) Compare 52 and 29. Since 52>29, swapping is done

30, **29, 52**, 87, 63, 27, 19, 54

c) Compare 52 and 87. Since 52<87, no swapping is done

d) Compare, 87 and 63. Since, 87>83, swapping is done

30, 29, 52, **63, 87**, 27, 19, 54

e) Compare87 and 27. Since 87>27, swapping is done

30, 29, 52, 63, **27, 87**, 19, 54

f) Compare 87 and 19. Since 87>19, swapping is done

30, 29, 52, 63, 27, 19, 87, 54

g) Compare 87 and 54. Since 87>54, swapping is done

30, 29, 52, 63, 27, 19, **54, 87**

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Bubble Sort Example

Pass 2:

a) Compare 30 and 29. Since 30>29, swapping is done

29, 30, 52, 63, 27, 19, 54, 87

b) Compare 30 and 52. Since 30<52, no swapping is done

c) Compare 52 and 63. Since 52<63, no swapping is done

d) Compare 63 and 27. Since 63>27, swapping is done

29, 30, 52, **27, 63**, 19, 54, 87

e) Compare 63 and 19. Since 63>19, swapping is done

29, 30, 52, 27, **19, 63**, 54, 87

f) Compare 63 and 54. Since 63>54, swapping is done

29, 30, 52, 27, 19, **54, 63**, 87

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

Bubble Sort Example

Pass 3:

a) Compare 29 and 30. Since 29<30, no swapping is done

b) Compare 30 and 52. Since 30<52, no swapping is done

c) Compare 52 and 27. Since 52>27, swapping is done

29, 30, **27, 52**, 19, 54, 63, 87

d) Compare 52 and 19. Since 52>19, swapping is done

29, 30, 27**, 19, 52**, 54, 63, 87

e) Compare 52 and 54. Since 52<54, no swapping is done

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

Bubble Sort Example

Pass 4:

a) Compare 29 and 30. Since 29<30, no swapping is done

b) Compare 30 and 27. Since 30>27, swapping is done

29, **27, 30**, 19, 52, 54, 63, 87

c) Compare 30 and 19. Since 30>19, swapping is done

29, 27, **19, 30**, 52, 54, 63, 87

d) Compare 30 and 52. Since 30<52, no swapping is done

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

This process continues in pass 5, 6 and 7

Bubble Sort Algorithm

```
BubbleSort(A, n)
{ i = 1
while(i < n)
    {
          for (j = 1 to n-i)
                    if (A[j] < A[j+1])
                              Swap A[j] & A[j+1]
    i = i+1
}
}
```

Time Complexity of Bubble Sort

- The complexity of any sorting algorithm depends upon the number of comparisons that are made. In bubble sort, we have seen that there are total N-1 passes. In the first pass, N-1 comparisons are made to place the highest element in its correct position. Then in Pass 2, there are N-2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of the bubble sort, we need to calculate the total number of comparisons made. For this purpose, the number f(n) of comparisons made can be given as,

$$f(n) = (n - 1) + (n - 2) + (n - 3) + \ldots\ldots + 3 + 2 + 1 = n\,(n - 1)/2 = O(n^2)$$

Already Sorted Collections?

- What if the collection was already sorted?
- What if only a few elements were out of place and after a couple of "bubble ups," the collection was sorted?

1	*2*	*3*	*4*	*5*	*6*
5	*12*	*35*	*42*	*77*	*101*

- We want to be able to detect this and "stop early"!

Using a Boolean “Flag”

- We can use a boolean variable to determine if any swapping occurred during the “bubble up.”

- If no swapping occurred, then we know that the collection is already sorted!

- This boolean “flag” needs to be reset after each “bubble up.”

Bubble Sort Algorithm

```
BubbleSort(A, n)
{ i=1
flag = true
while (i < n AND flag = true)
    {       flag = false
            for (j = 1 to n-i)
                    if (A[j] < A[j+1])
                            {
                            Swap A[j] & A[j+1]
                            flag = true
                            }
            i = i+1
    }
}
```

Selection Sort

- Consider an array ARR with N elements. The selection sort takes N-1 passes to sort the entire array and works as follows. First find the smallest value in the array and place it in the first position. Then find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,
- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
- In pass 2, find the position POS of the smallest value in sub-array of N-1 elements. Swap ARR[POS] with ARR[1]. Now, A[0] and A[1] is sorted
- In pass 3, find the position POS of the smallest value in sub-array of N-2 elements. Swap ARR[POS] with ARR[2]. Now, ARR[0], ARR[1] and ARR[2] is sorted
- In pass N-1, find the position POS of the smaller of the elements ARR[N-2] and ARR[N-1}. Swap ARR[POS] and ARR[N-2] so that ARR[0], ARR[1], … , ARR[N-1] is sorted.

Selection Sort

Example: Sort the array given below using selection sort

39	9	81	45	90	27	72	18

PASS	LOC	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90

Selection Sort Algorithm

```
SelectSort(A, n)
{
for (i = 1 to n )
        {       min = A[i]
                pos=i;
                for (j = i+1 to n)
                        if (A[j] < min)
                                {min=A[j]
                                pos=j
                                }
                A[pos]=A[i]
                A[i]=min
        }
}
```

Complexity of Selection Sort Algorithm

- Selection sort is a sorting algorithm that is independent of the original order of the elements in the array. In pass 1, selecting the element with smallest value calls for scanning all n elements; thus, n-1 comparisons are required in the first pass.

- Then, the smallest value is swapped with the element in the first position. In pass 2, selecting the second smallest value requires scanning the remaining $n - 1$ elements and so on. Therefore,

- $T(n) = (n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1) / 2 = O(n^2)$

Insertion Sort

- Insertion sort is a very simple sorting algorithm, in which the sorted array (or list) is built one element at a time.
- Insertion sort works as follows.
- The array of values to be sorted is divided into two sets. One that stores sorted values and the other contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially the element with index 0 (assuming LB, Lower Bound = 0) is in the sorted set, rest all the elements are in the unsorted set
- The first element of the unsorted partition has array index 1 (if LB = 0)
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Insertion Sort

Example: Consider an array of integers given below. Sort the values in the array using insertion sort.

39	9	45	63	18	81	108	54	72	36

39	9	45	63	18	81	108	54	72	36

Insertion Sort

9	39	45	63	18	81	108	54	72	36
9	39	45	63	18	81	108	54	72	36
9	39	45	63	18	81	108	54	72	36
9	18	39	45	63	81	108	54	72	36
9	18	39	45	63	81	108	54	72	36
9	18	39	45	63	81	108	54	72	36
9	18	39	45	54	63	81	108	72	36
9	18	39	45	54	63	72	81	108	36
9	18	36	39	45	54	63	72	81	108

Insertion Sort

Note : In Pass 1, A[0] is the only element in the sorted set.

In Pass 2, A[1] will be placed either before or after A[0], so that the array A is sorted

In Pass 3, A[2] will be placed either before A[0], in-between A[0] and A[1] or after A[1], so that the array is sorted.

In Pass 4, A[4] will be placed in its proper place so that the array A is sorted.

In Pass N, A[N-1] will be placed in its proper place so that the array A is sorted.

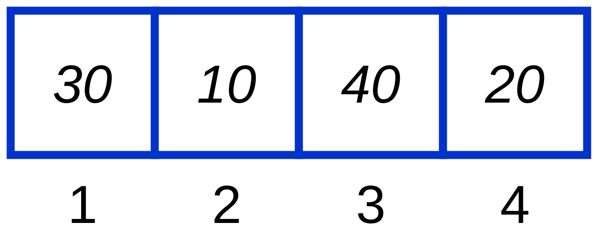
Therefore, we conclude to insert the element A[K] is in the sorted list A[0], A[1], …. A[K-1], we need to compare A[K] with A[K-1], then with A[K-2], then with A[K-3] until we meet an element A[J] such that A[J] <= A[K].

In order to insert A[K] in its correct position, we need to move each element A[K-1], A[K-2], …., A[J] by one position and then A[K] is inserted at the (J+1)th location.

Insertion Sort

```
InsertionSort(A, n) {
  for ( i = 2 to n) {
     key = A[i]
     j = i - 1
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```

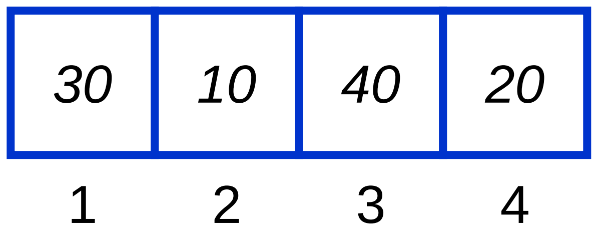
An Example: Insertion Sort



$i = \emptyset$ $j = \emptyset$ $key = \emptyset$
$A[j] = \emptyset$ $A[j+1] = \emptyset$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
               A[j+1] = A[j]
               j = j - 1
        }
        A[j+1] = key
   }
}
```

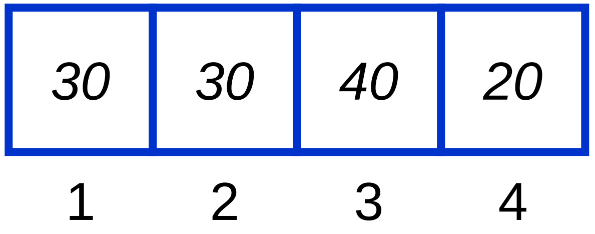
An Example: Insertion Sort



i = 2 j = 1 key = 10
A[j] = 30 A[j+1] = 10

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

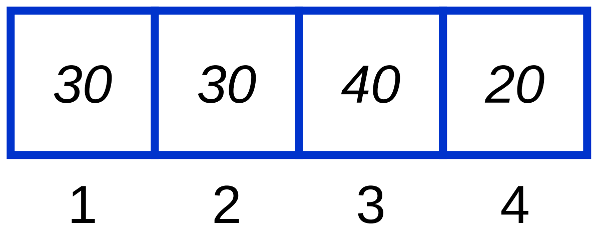
An Example: Insertion Sort



$i = 2$ $j = 1$	$key = 10$
$A[j] = 30$	$A[j+1] = 30$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort



$i = 2$ $j = 1$ $key = 10$

$A[j] = 30$ $A[j+1] = 30$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

30	*30*	*40*	*20*
1	2	3	4

$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 30$

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
       }
       A[j+1] = key
   }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2	j = 0	key = 10
A[j] = ∅		A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
➡       A[j+1] = key
   }
}
```

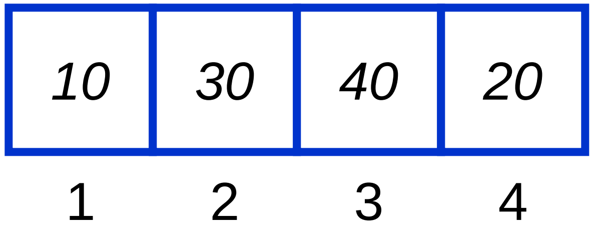
An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

$i = 2$ $j = 0$ $key = 10$
$A[j] = \varnothing$ $A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

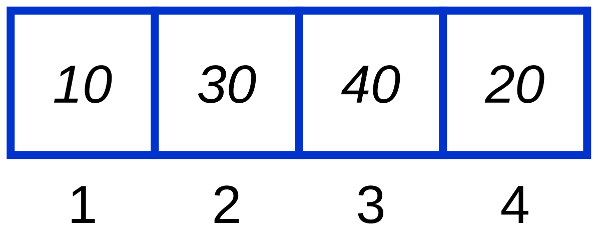


i = 3 j = 0 key = 10

A[j] = ∅ A[j+1] = 10

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort



i = 3   j = 0   key = 40
A[j] = ∅   A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

i = 3 j = 0 key = 40

A[j] = ∅ A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

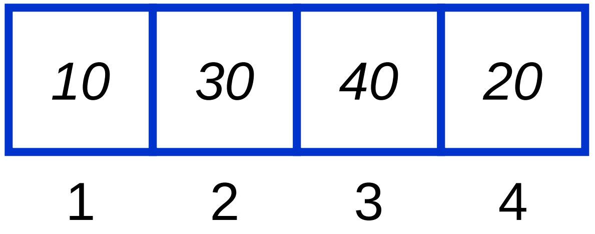
10	*30*	*40*	*20*
1	2	3	4

i = 3   j = 2   key = 40

A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort



i = 3 j = 2 key = 40
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

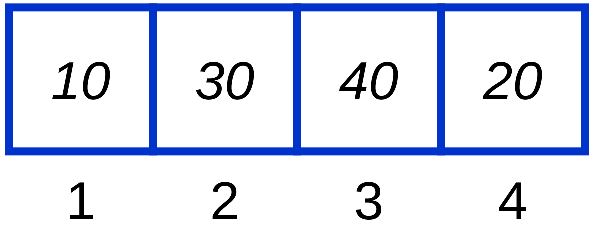
An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

i = 3 j = 2 key = 40
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

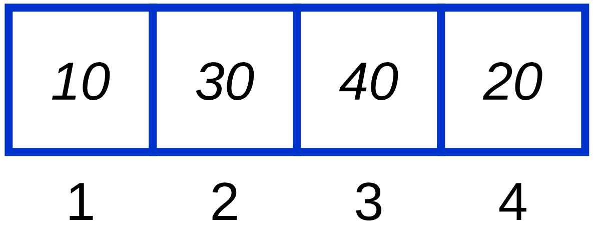


i = 4 j = 2 key = 40

A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

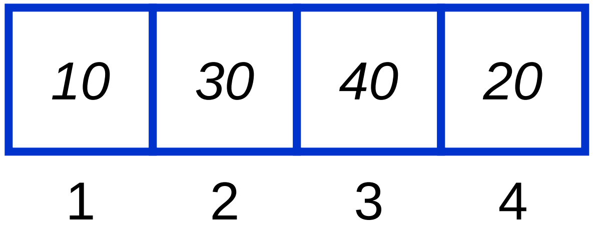


i = 4   j = 2   key = 20

A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
⇨       key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

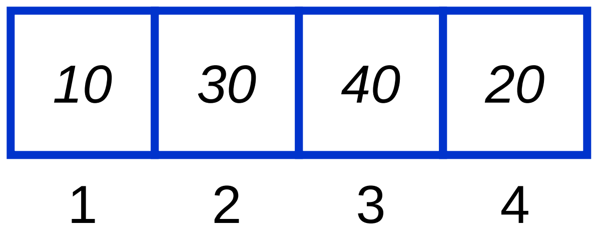
An Example: Insertion Sort



i = 4   j = 2   key = 20
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

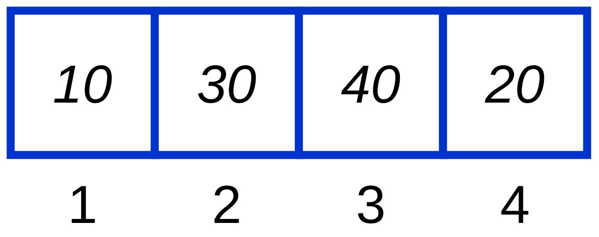
An Example: Insertion Sort



i = 4   j = 3   key = 20
A[j] = 40   A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
       key = A[i]
⇨      j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
  }
}
```

An Example: Insertion Sort



i = 4   j = 3   key = 20
A[j] = 40   A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

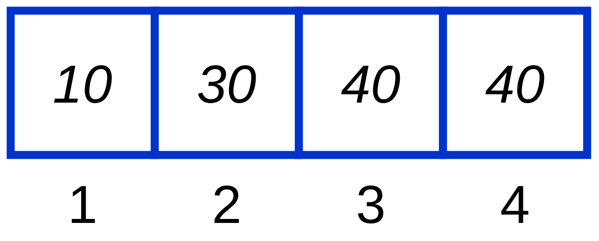
An Example: Insertion Sort

10	*30*	*40*	*40*
1	2	3	4

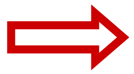
i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
         A[j+1] = A[j]
         j = j - 1
      }
      A[j+1] = key
   }
}
```

An Example: Insertion Sort

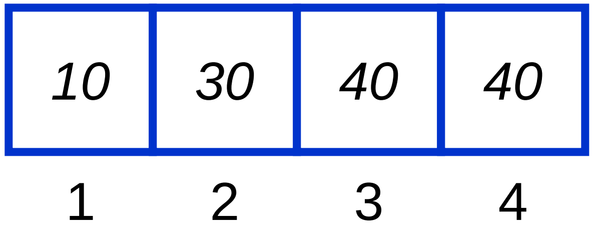


i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40



```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

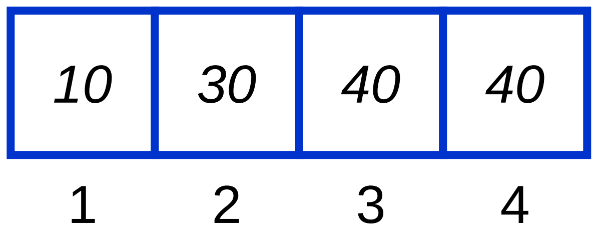
An Example: Insertion Sort



i = 4 j = 3	key = 20
A[j] = 40	A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort



$i = 4$ $j = 2$	$key = 20$
$A[j] = 30$	$A[j+1] = 40$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
⇨               j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

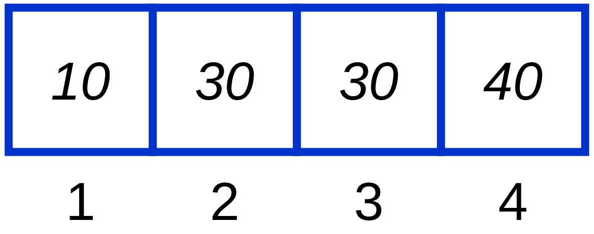
10	*30*	*40*	*40*
1	2	3	4

i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

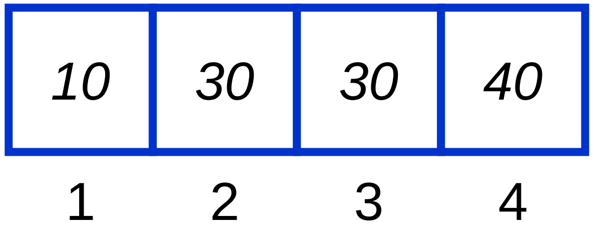


i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
   }
}
```

An Example: Insertion Sort

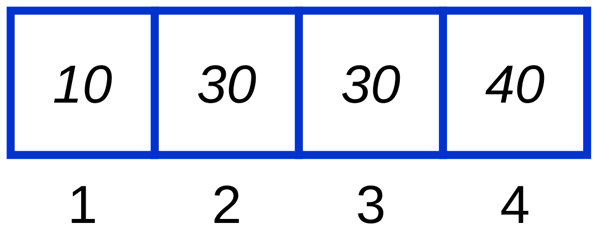


i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

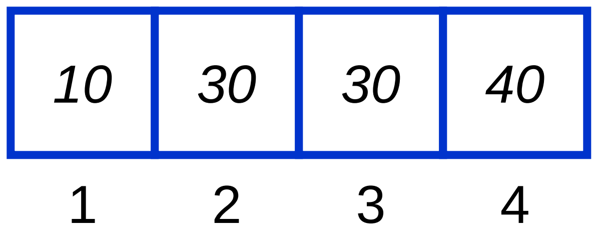
An Example: Insertion Sort



i = 4	j = 1	key = 20
A[j] = 10		A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
⇨               j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort



i = 4   j = 1   key = 20

A[j] = 10   A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

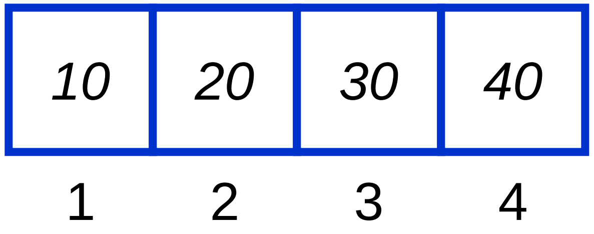
An Example: Insertion Sort

10	*20*	*30*	*40*
1	2	3	4

i = 4 j = 1 key = 20
A[j] = 10 A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort



i = 4   j = 1   key = 20
A[j] = 10   A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

Done!

Complexity of Insertion Sort Algorithm

- For an insertion sort, the best case occurs when the array is already sorted. In this case the running time of the algorithm has a linear running time (i.e., O(n)). This is because, during each iteration, the first element from unsorted set is compared only with the last element of the sorted set of the array.

- Similarly, the worst case of the insertion sort algorithm occurs when the array is sorted in reverse order. In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. Therefore, in the worst case, insertion sort has a quadratic running time (i.e., O(n^2)).

- Even in the average case, the insertion sort algorithm will have to make at least (K-1)/2 comparisons. Thus, the average case also has a quadratic running time.

Merge Sort

- Merge sort is a sorting algorithm that uses the divide, conquer and combine algorithmic paradigm. Where,

- **Divide** means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements in each sub-array. (If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A).

- **Conquer** means sorting the two sub-arrays recursively using merge sort.

- **Combine** means merging the two sorted sub-arrays of size n/2 each to produce the sorted array of n elements.

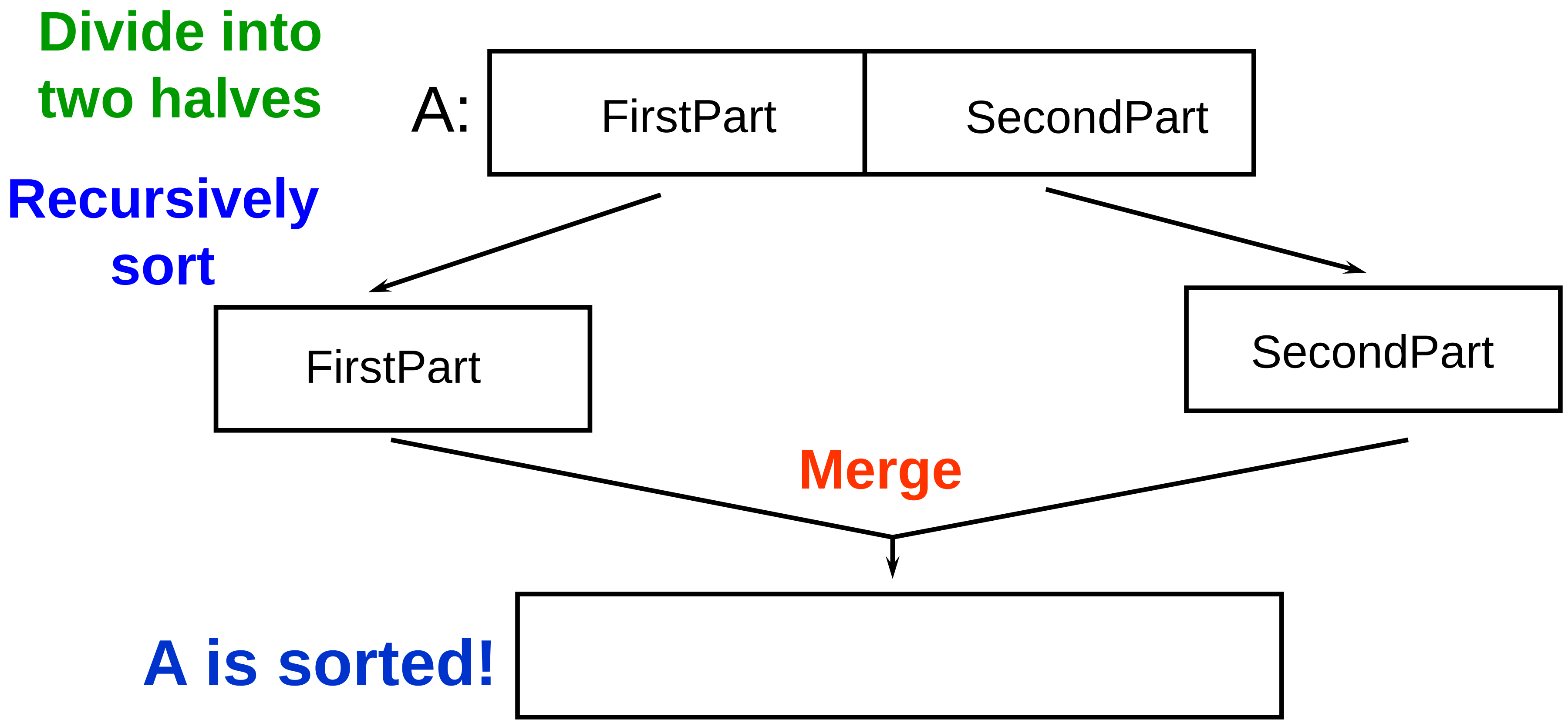
Merge Sort

- Merge sort algorithms focuses on two main concepts to improve its performance (running time):
- A smaller list takes few steps and thus less time to sort than a large list.
- Less steps, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted. Otherwise:
- (Conceptually) divide the unsorted array into two sub- arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array
- Merge the two sub-arrays to form a single sorted list

Merge Sort: Idea

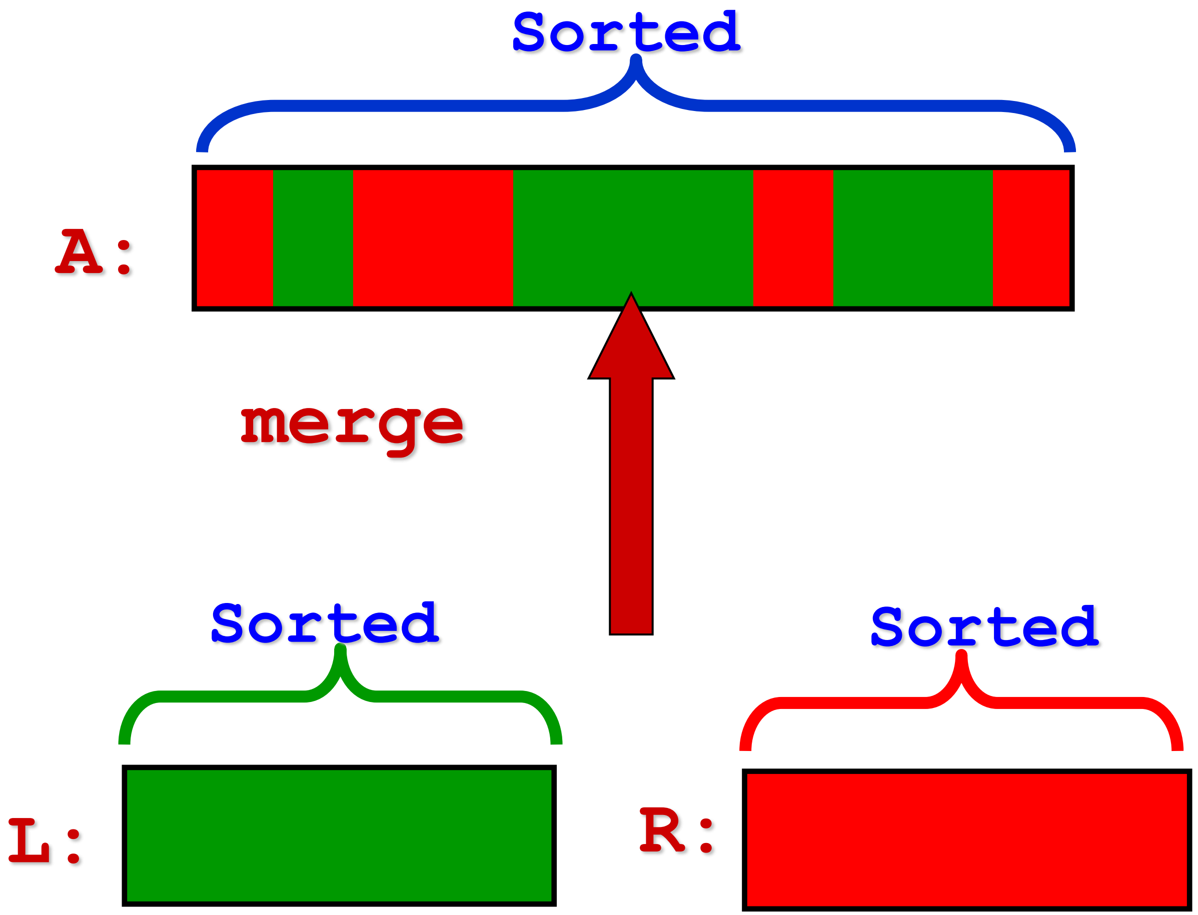


Merge Sort Algorithm

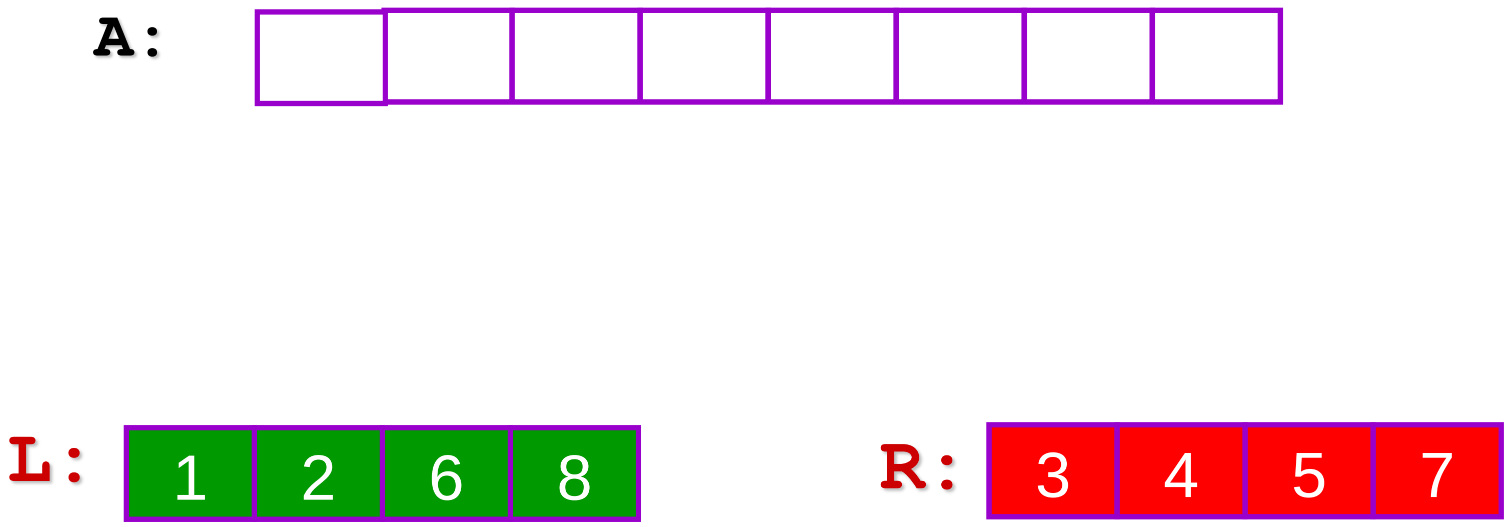
```
MergeSort(A, left, right) {
   if (left < right) {
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);
   }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
//     (how long should this take?)
```

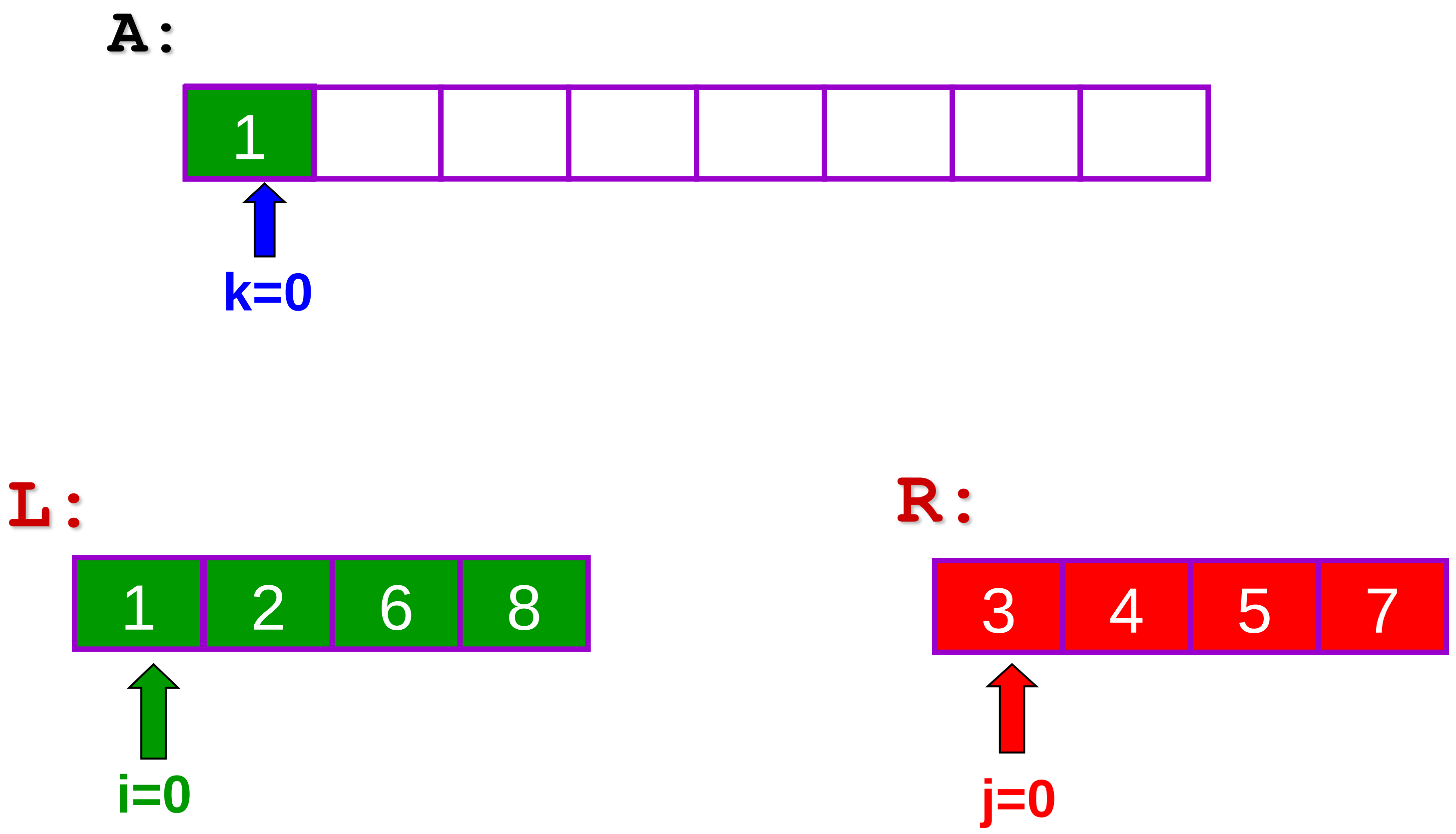
Merge-Sort: Merge



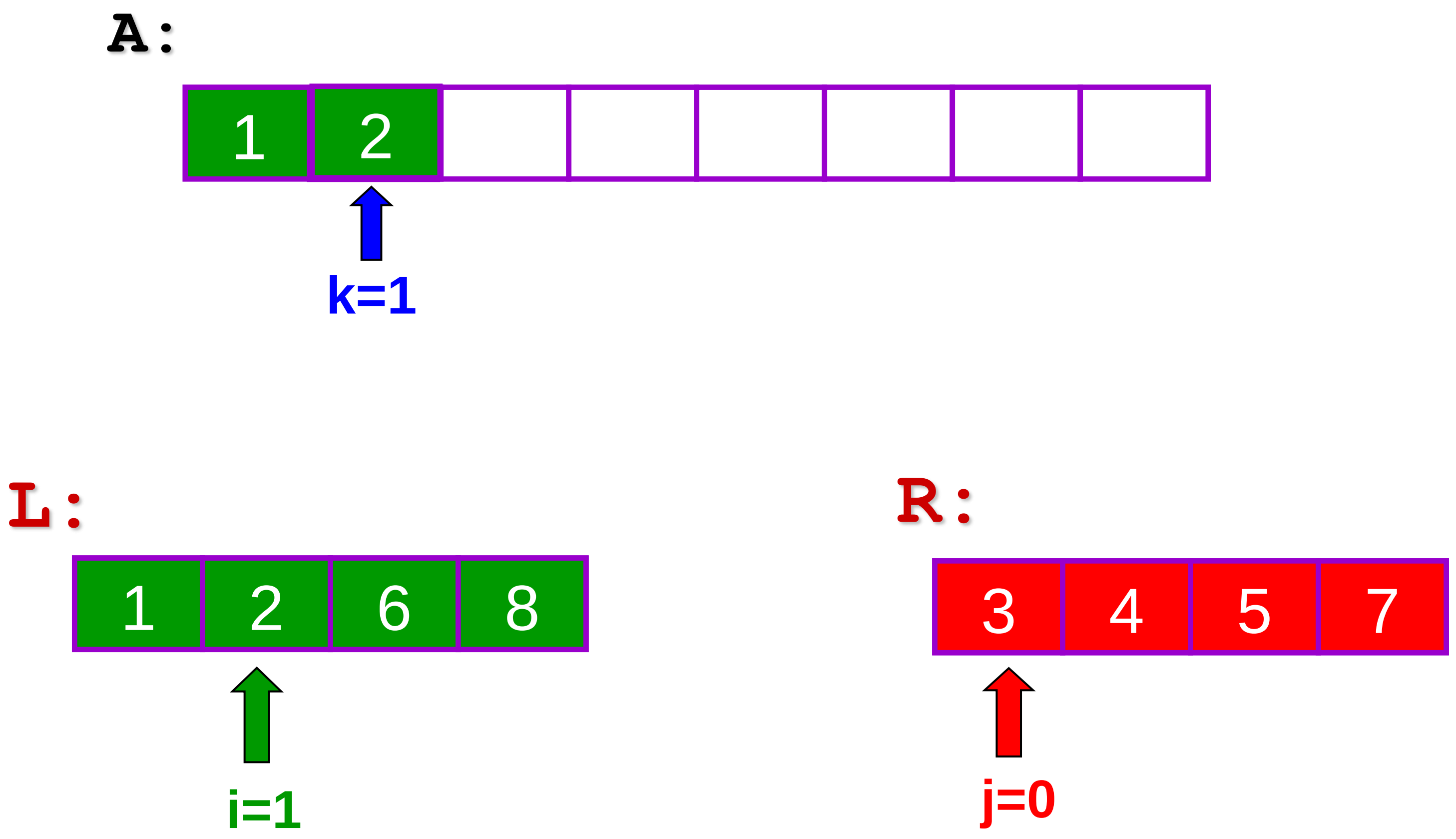
Merge-Sort: Merge Example



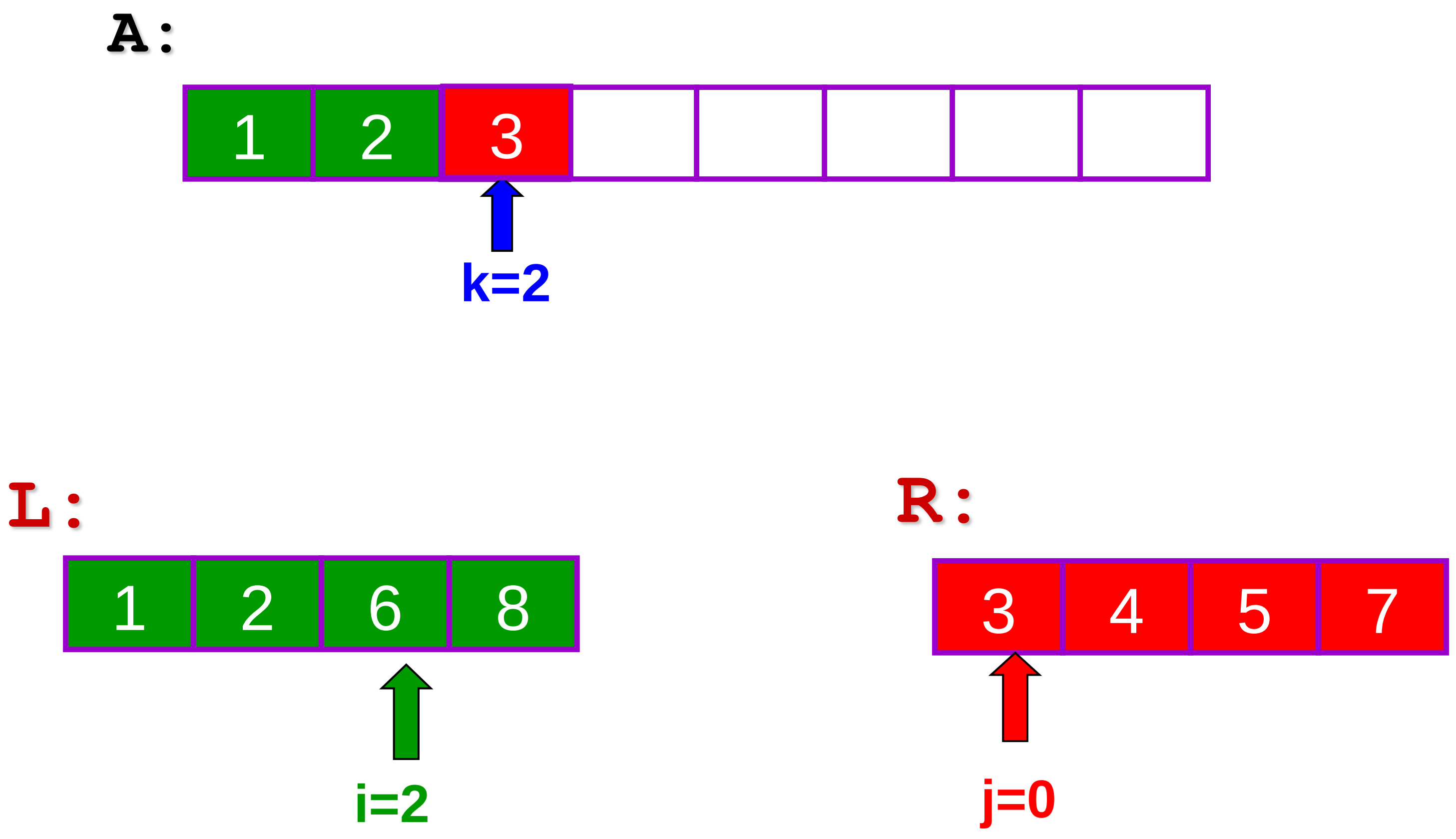
Merge-Sort: Merge Example



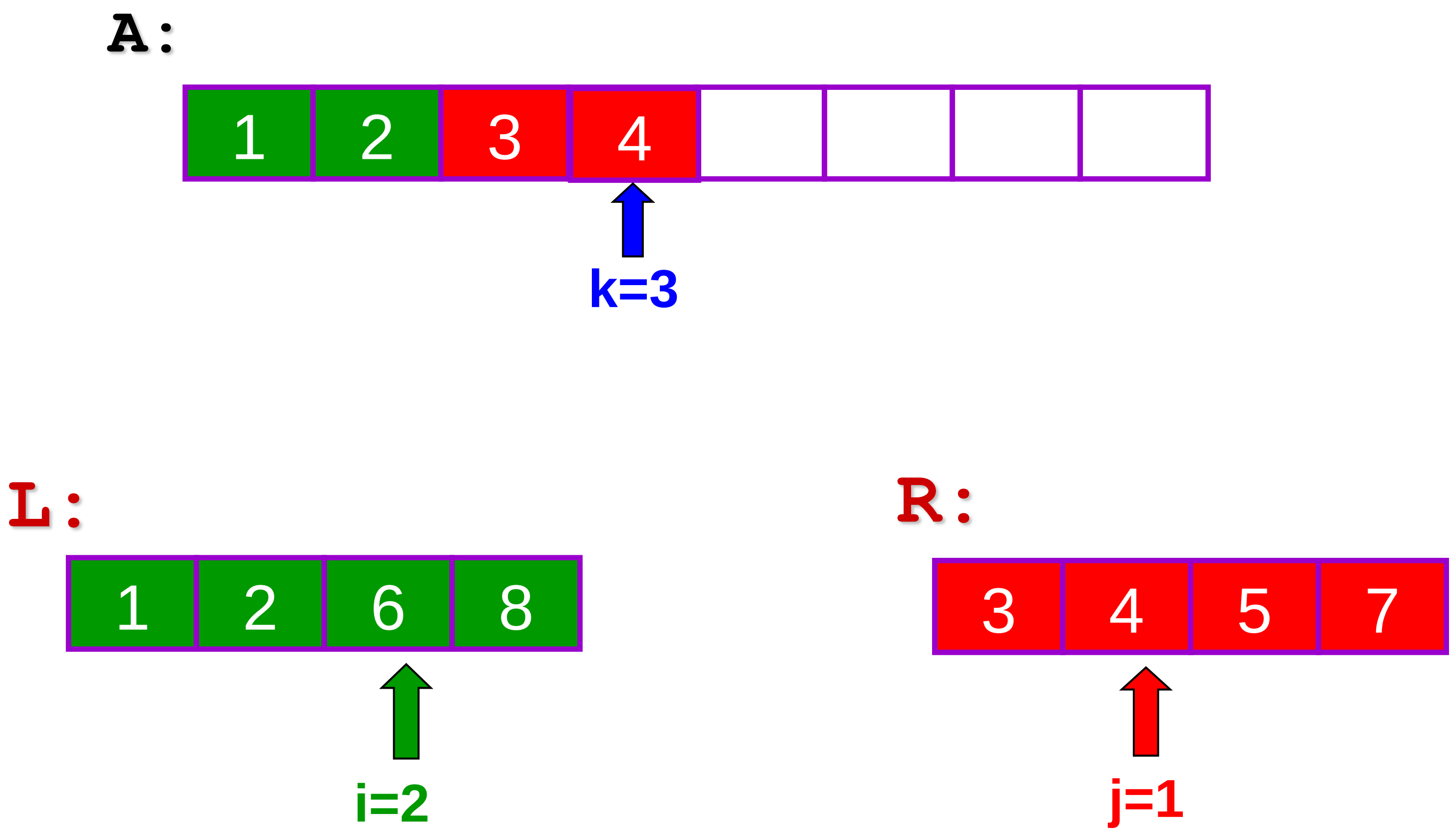
Merge-Sort: Merge Example



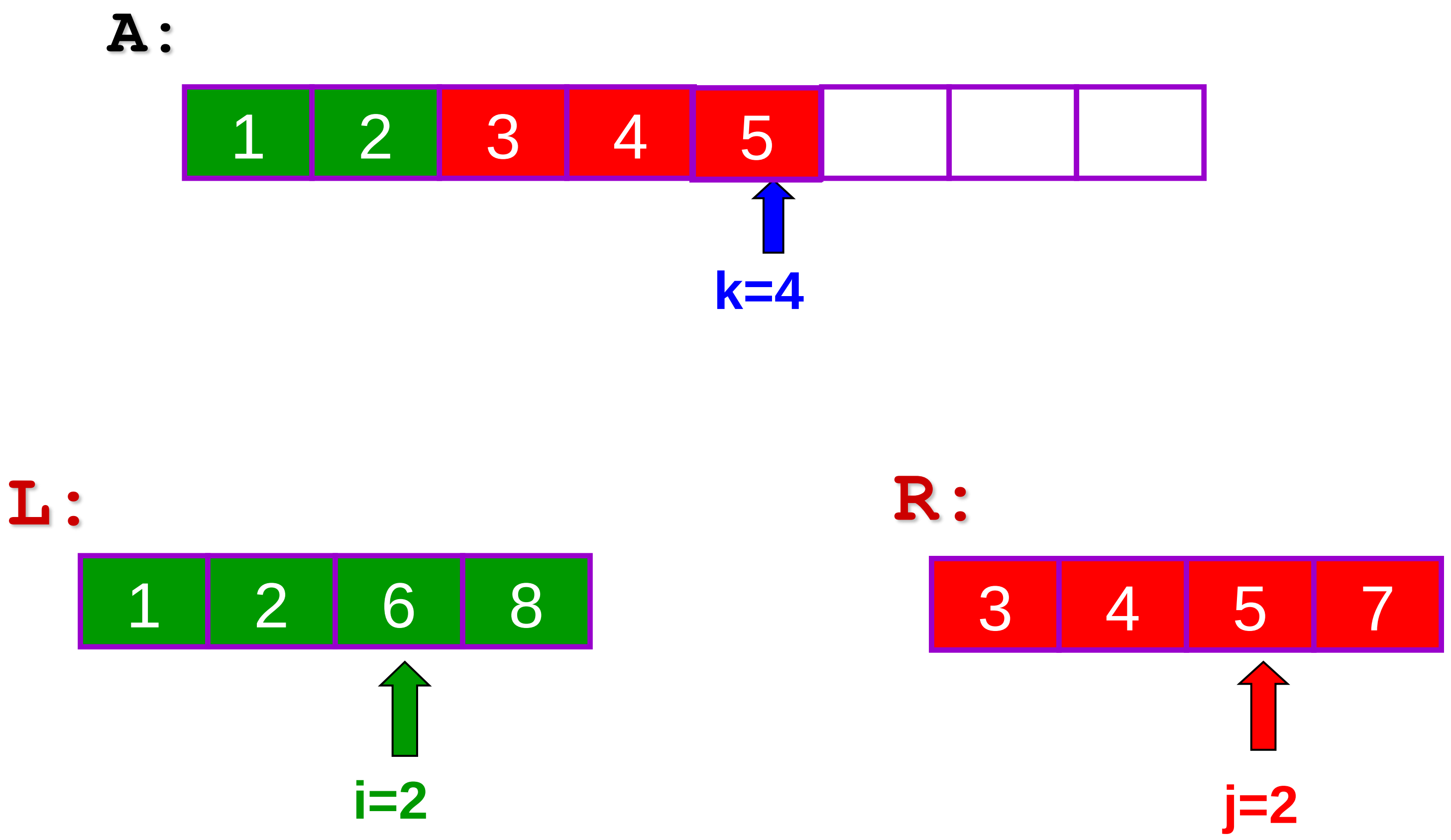
Merge-Sort: Merge Example



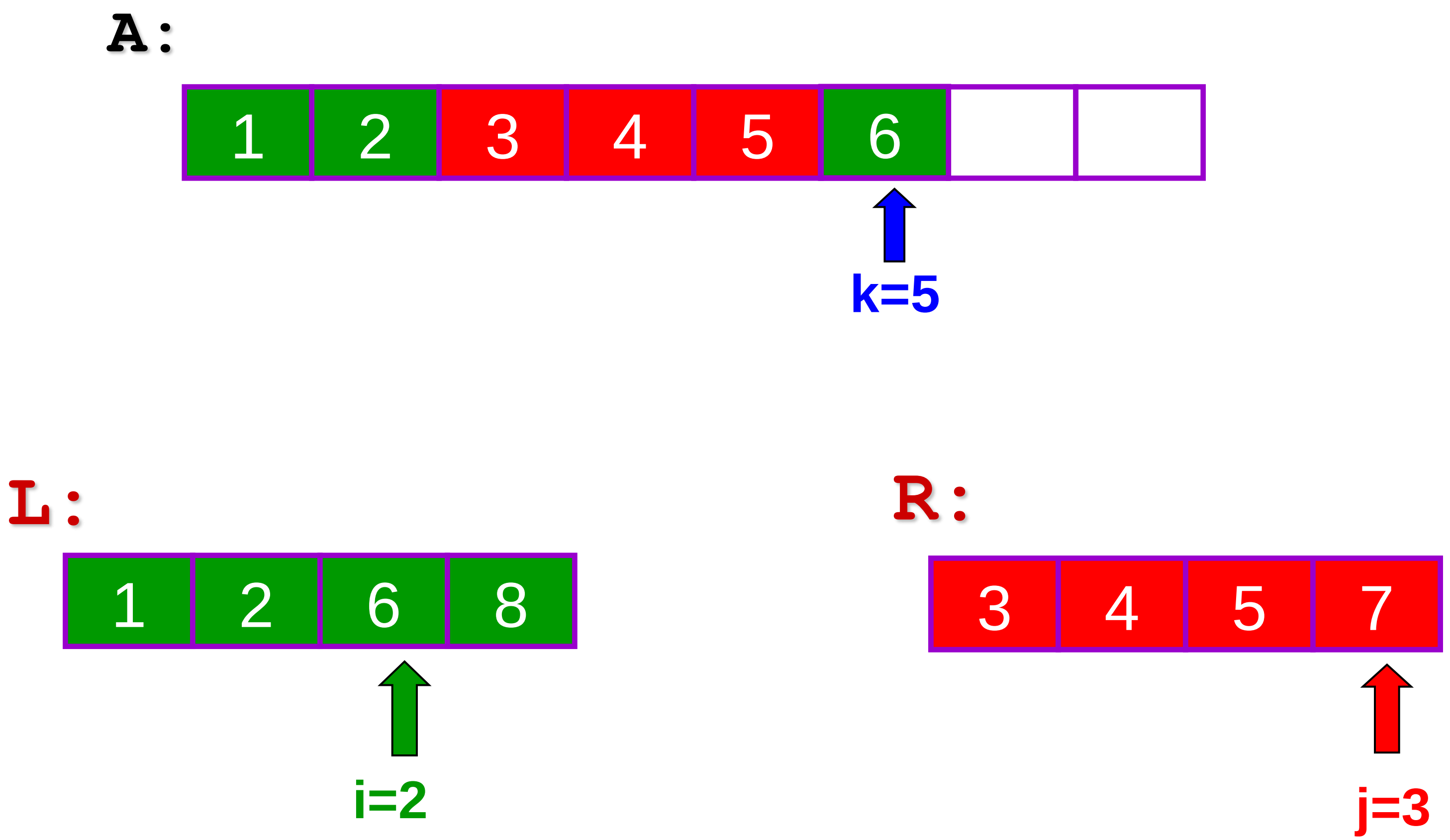
Merge-Sort: Merge Example



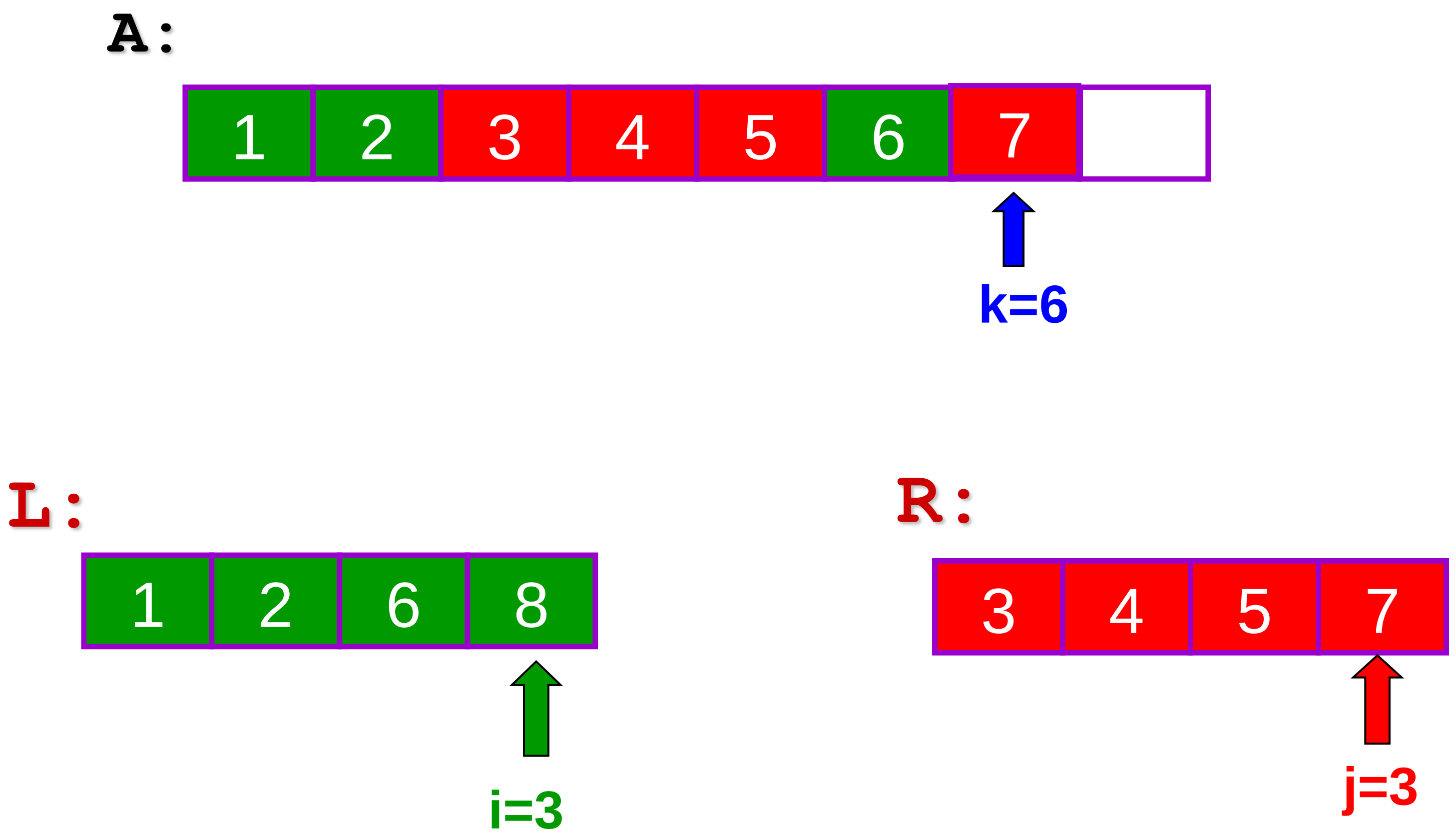
Merge-Sort: Merge Example



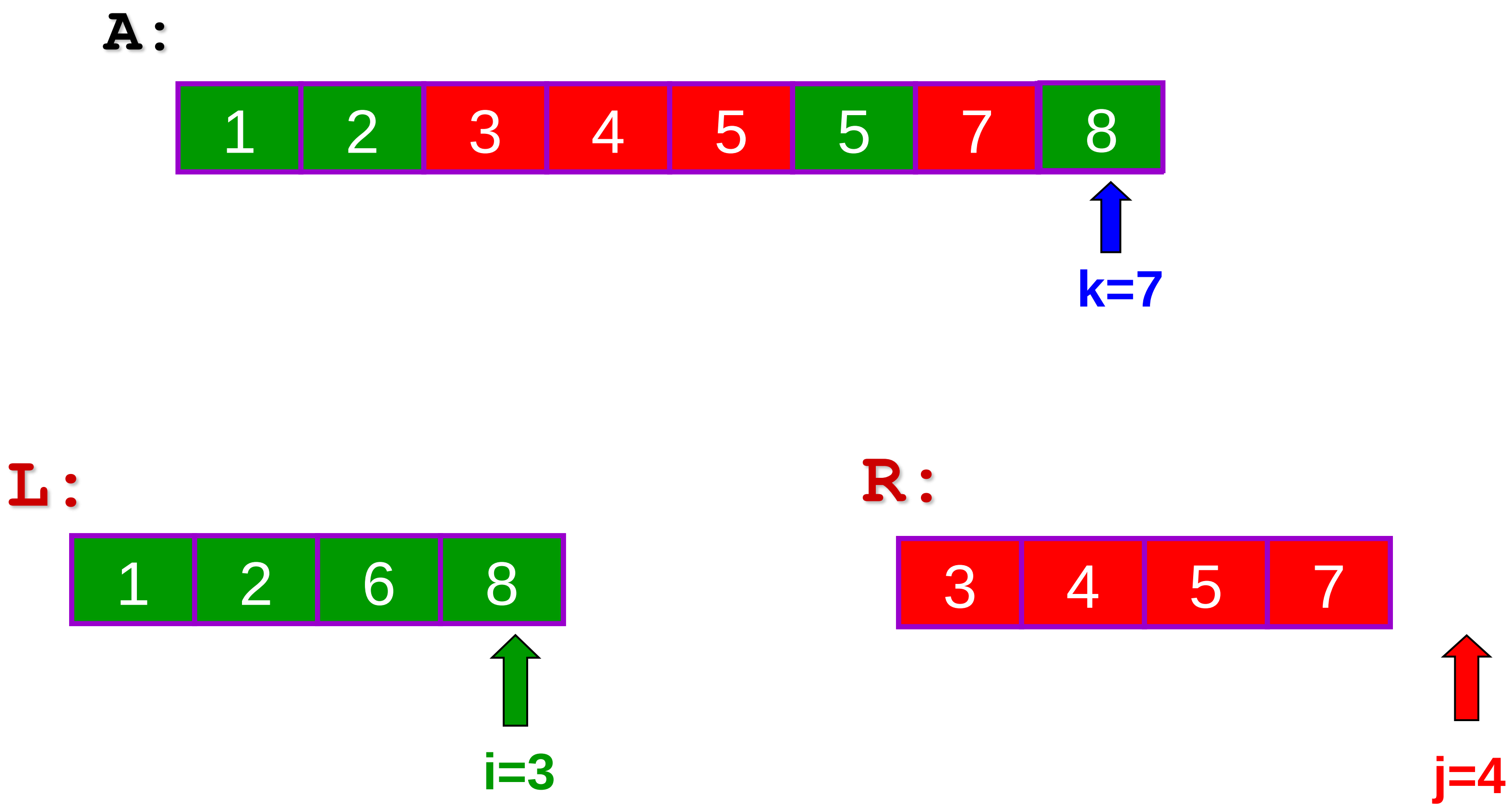
Merge-Sort: Merge Example



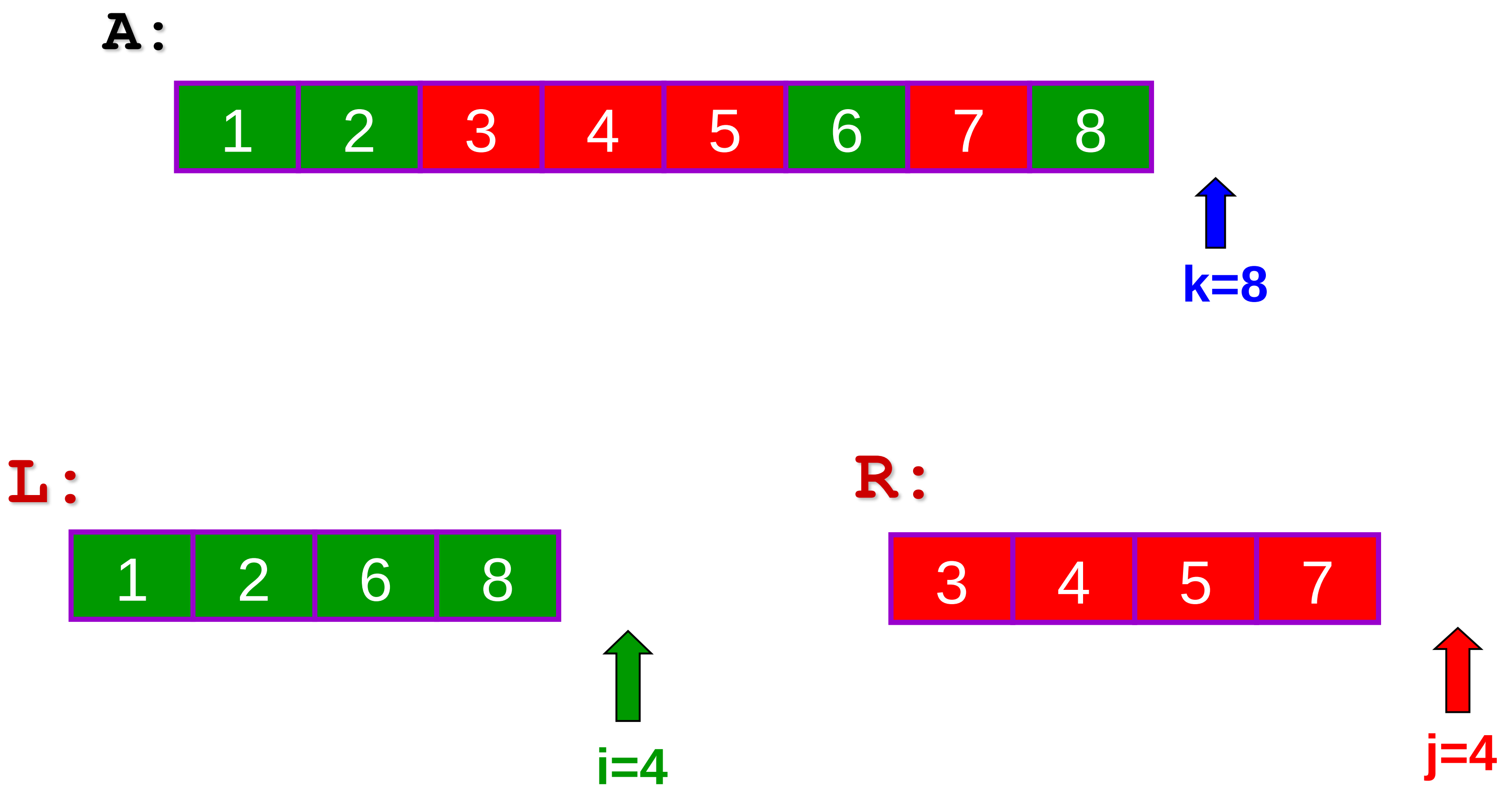
Merge-Sort: Merge Example



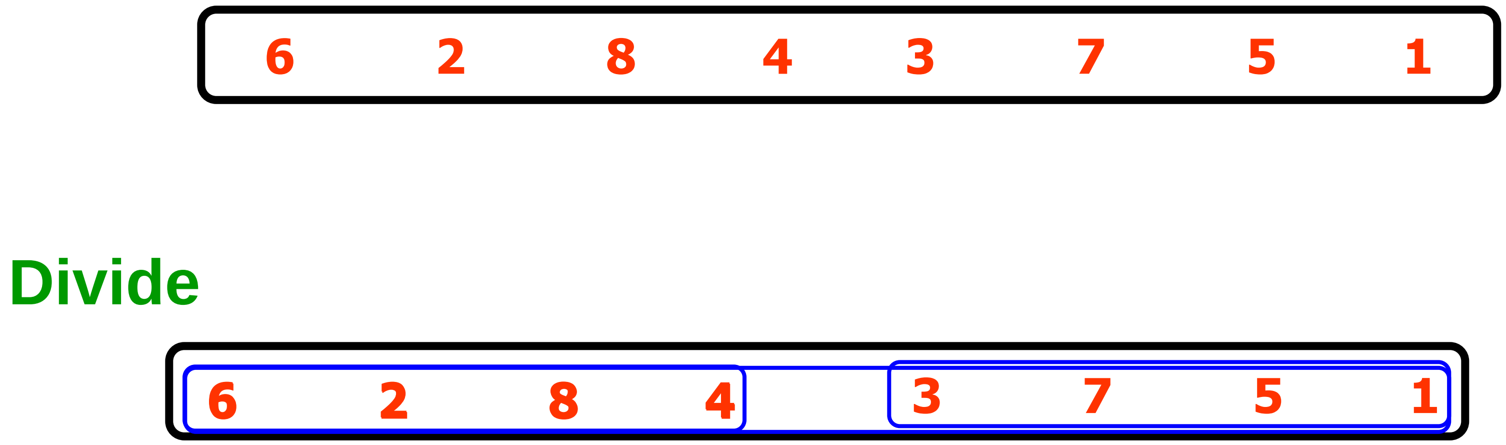
Merge-Sort: Merge Example



Merge-Sort: Merge Example

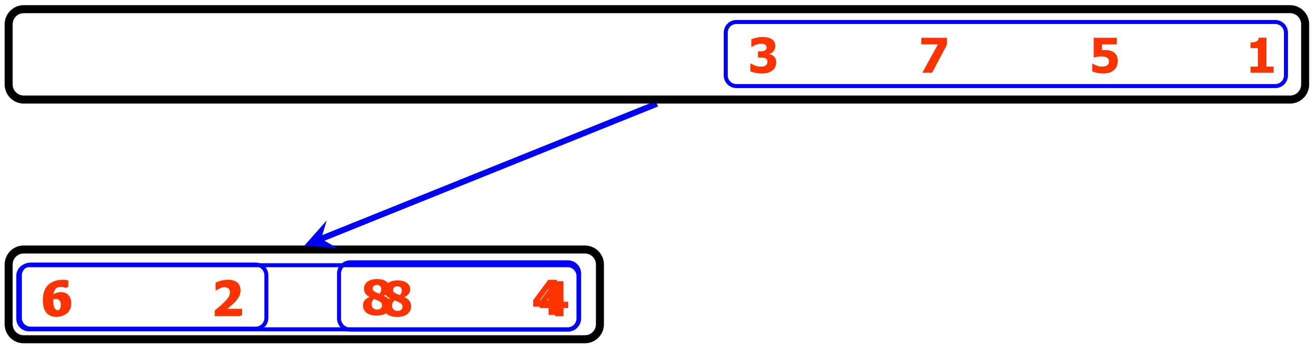


Merge-Sort Execution Example



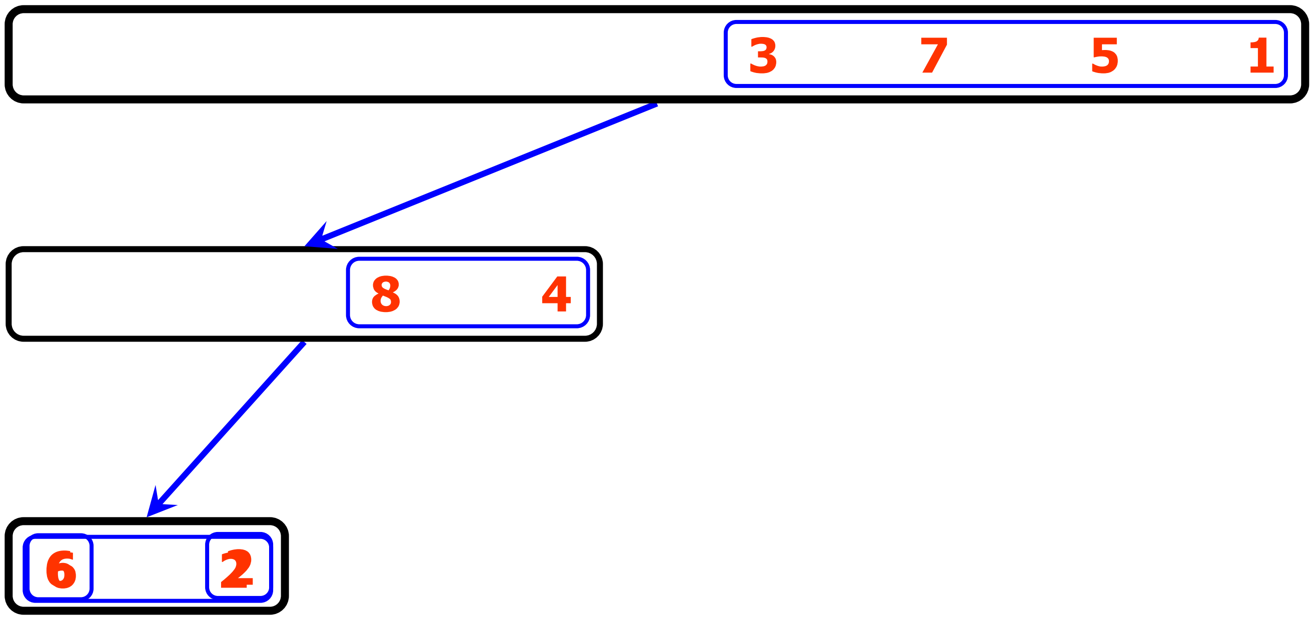
Merge-Sort Execution Example

Recursive call , divide



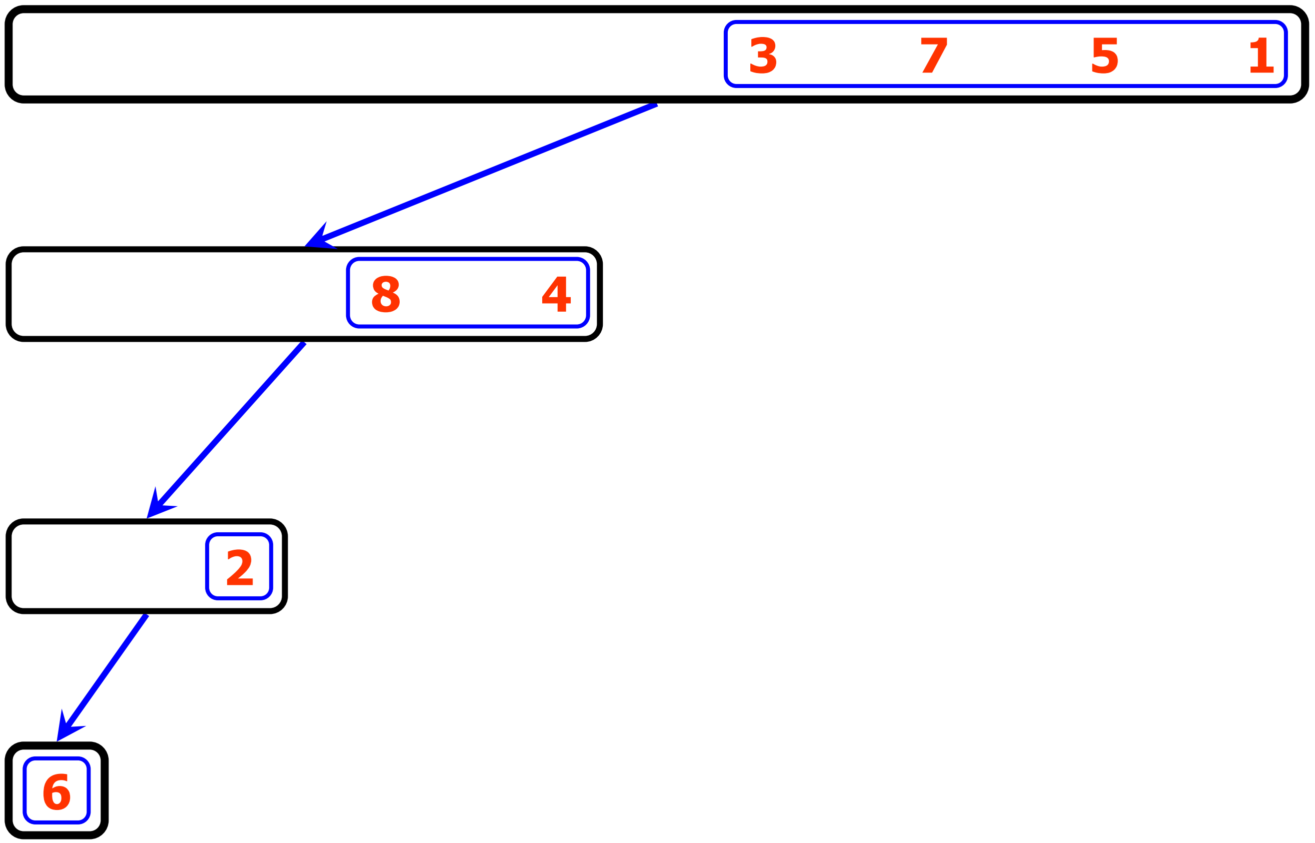
Merge-Sort Execution Example

Recursive call , divide



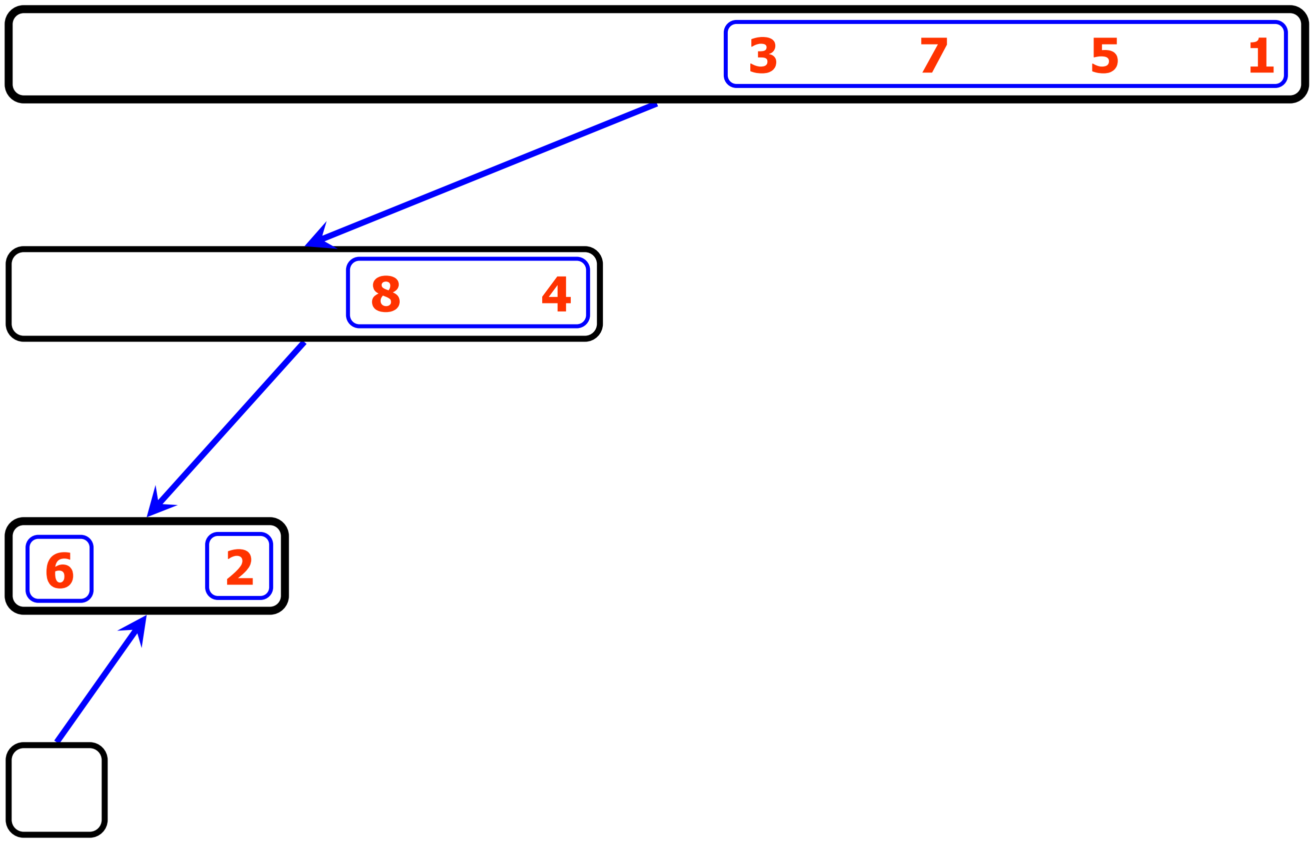
Merge-Sort Execution Example

Recursive call , base case



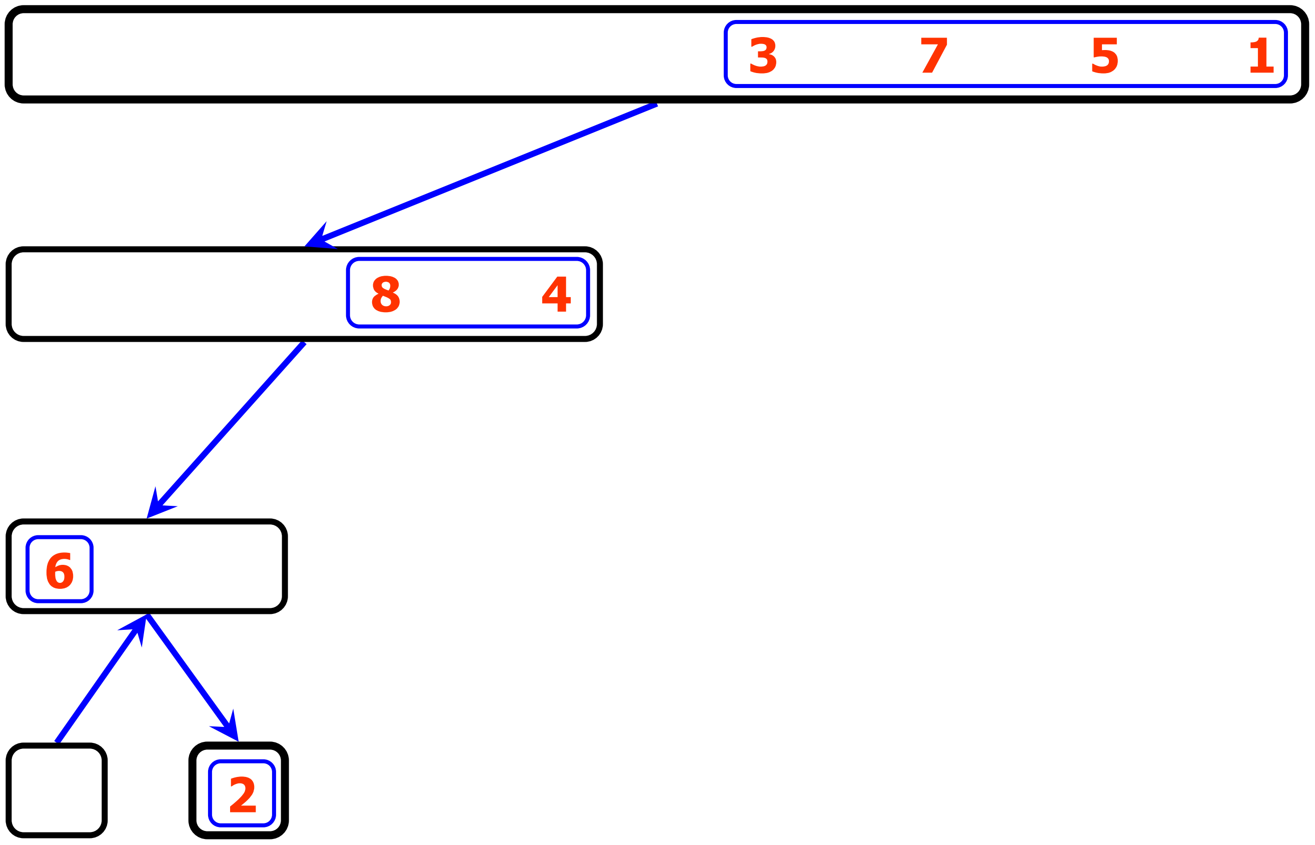
Merge-Sort Execution Example

Recursive call return



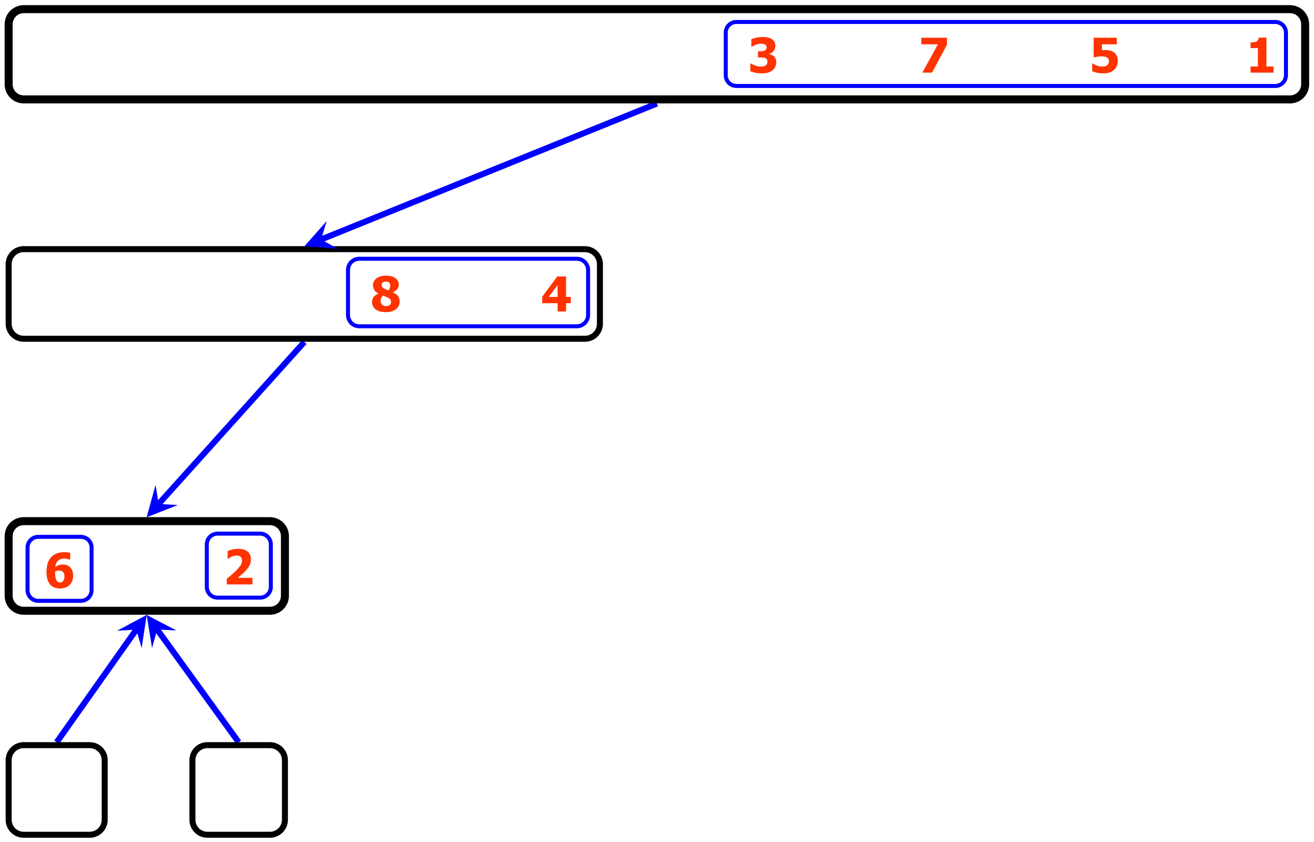
Merge-Sort Execution Example

Recursive call , base case



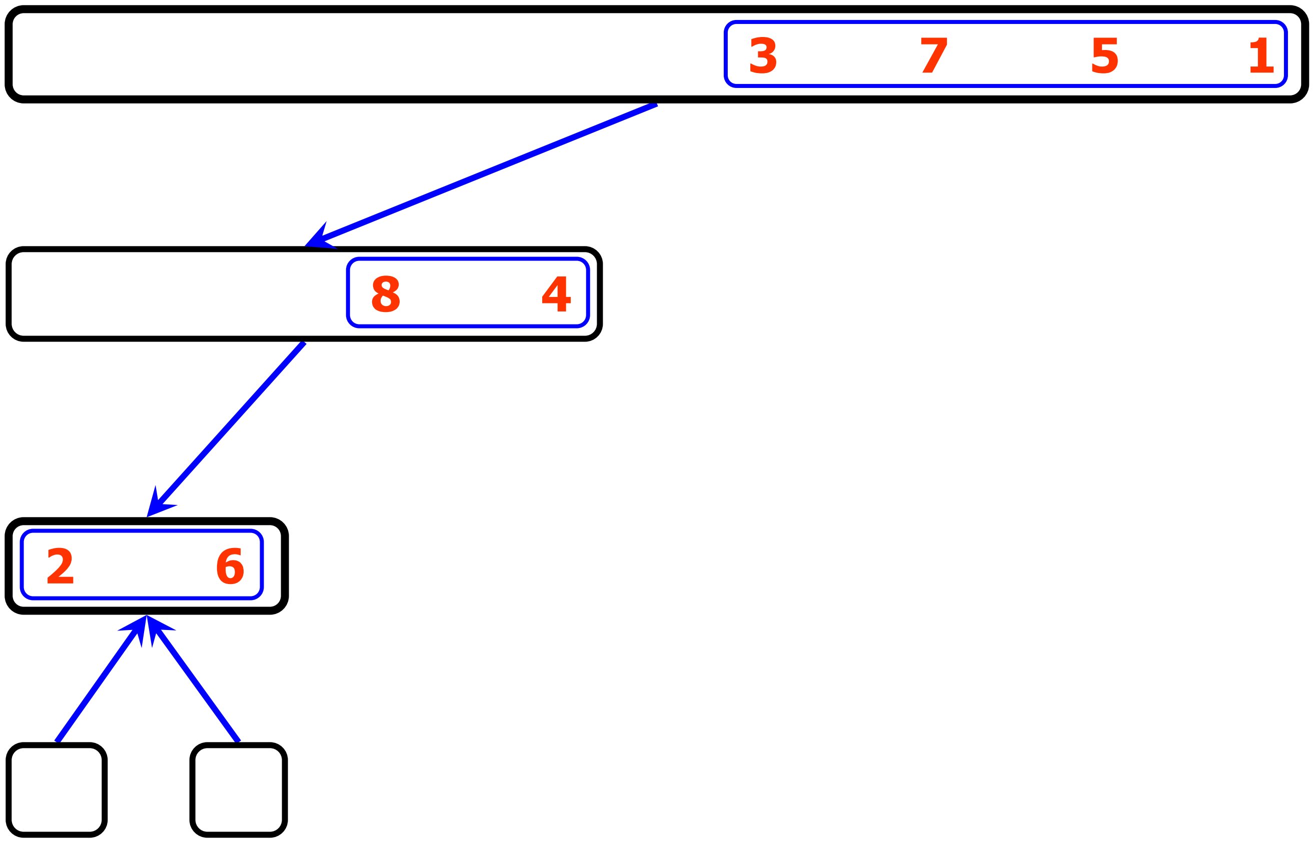
Merge-Sort Execution Example

Recursive call return



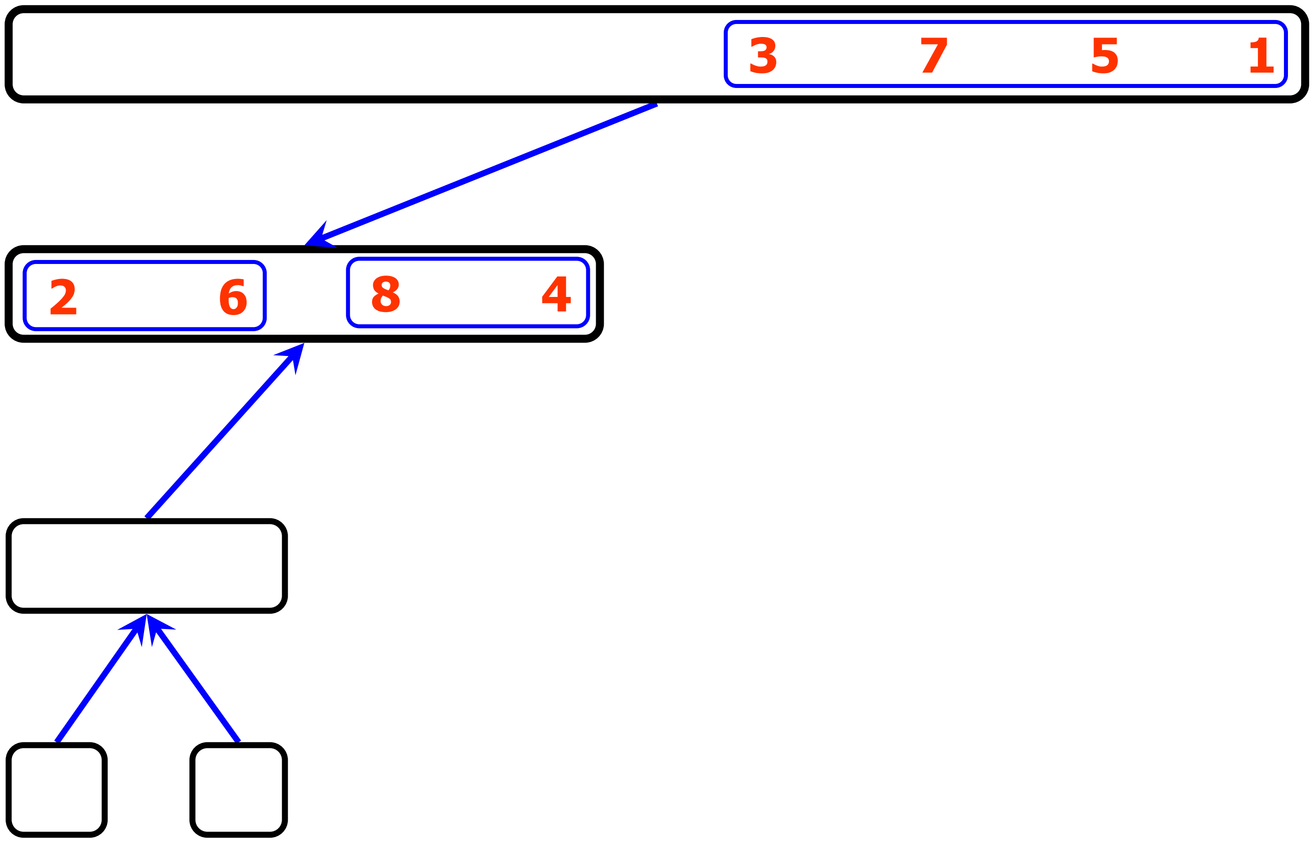
Merge-Sort Execution Example

Merge



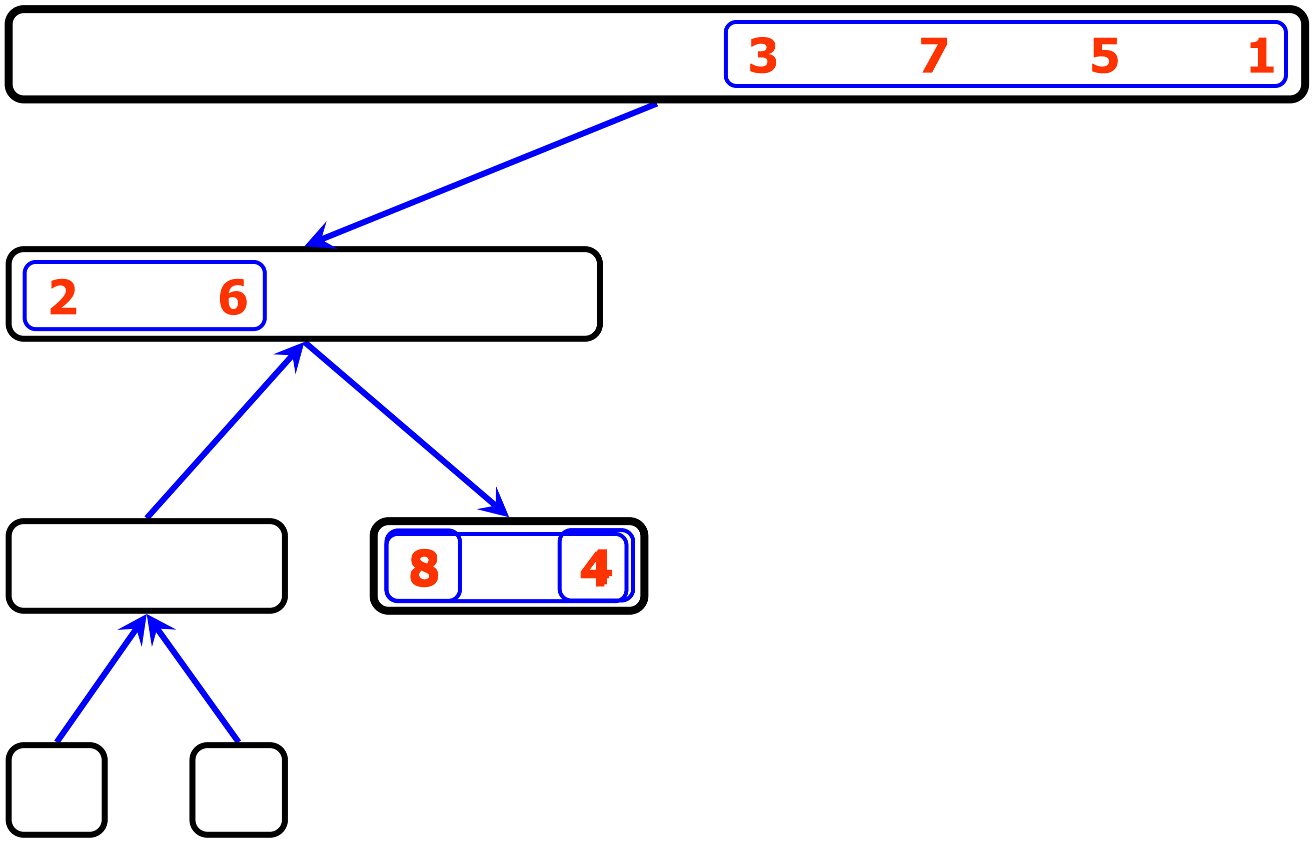
Merge-Sort Execution Example

Recursive call return



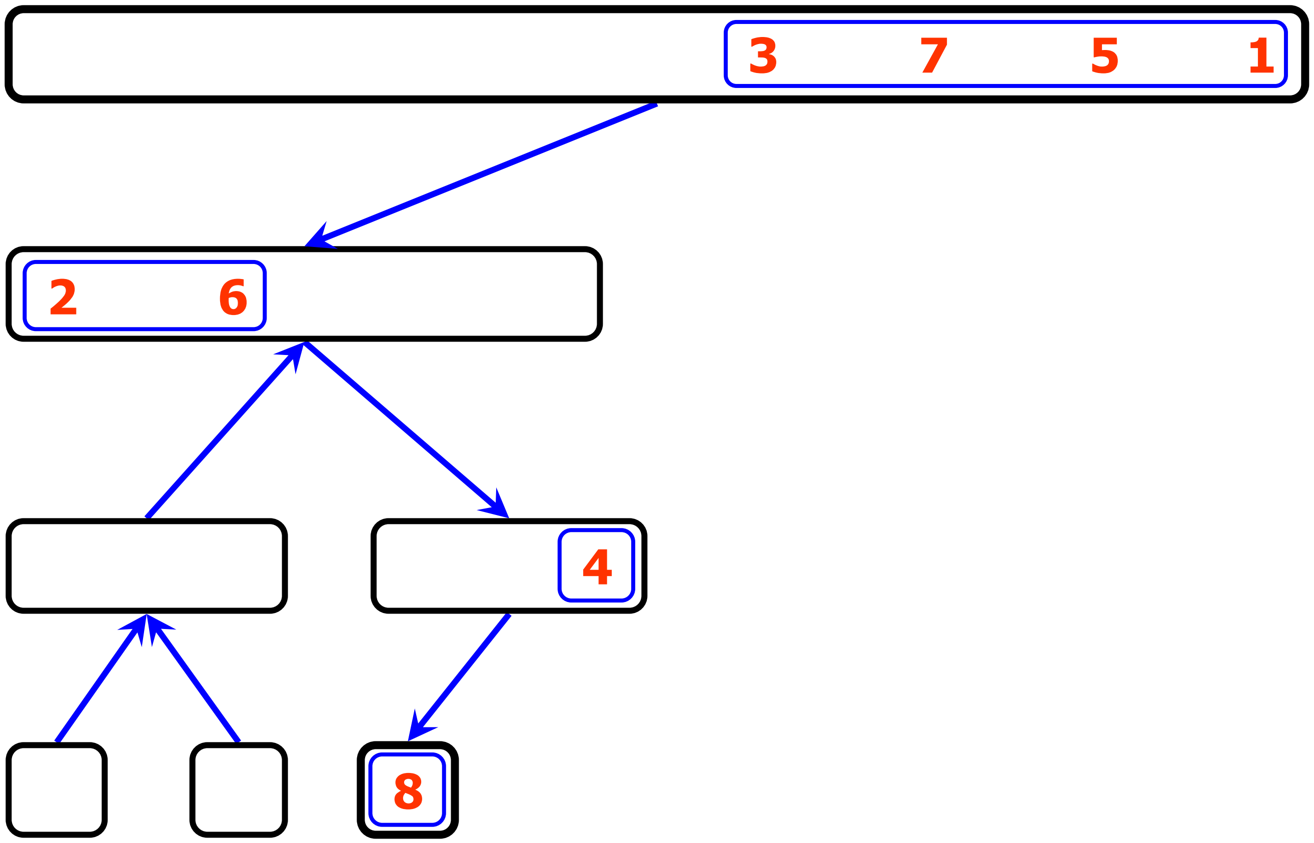
Merge-Sort Execution Example

Recursive call , divide



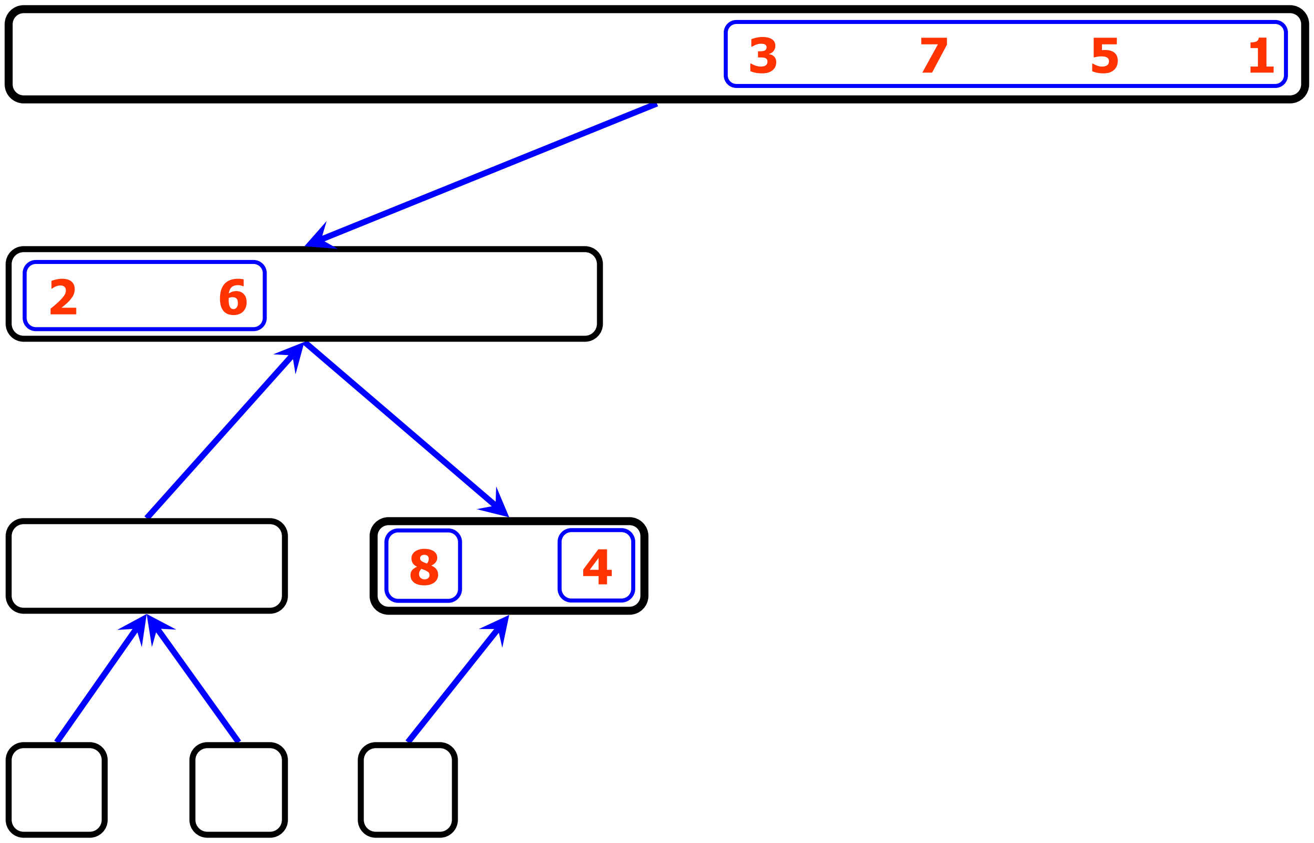
Merge-Sort Execution Example

Recursive call, base case



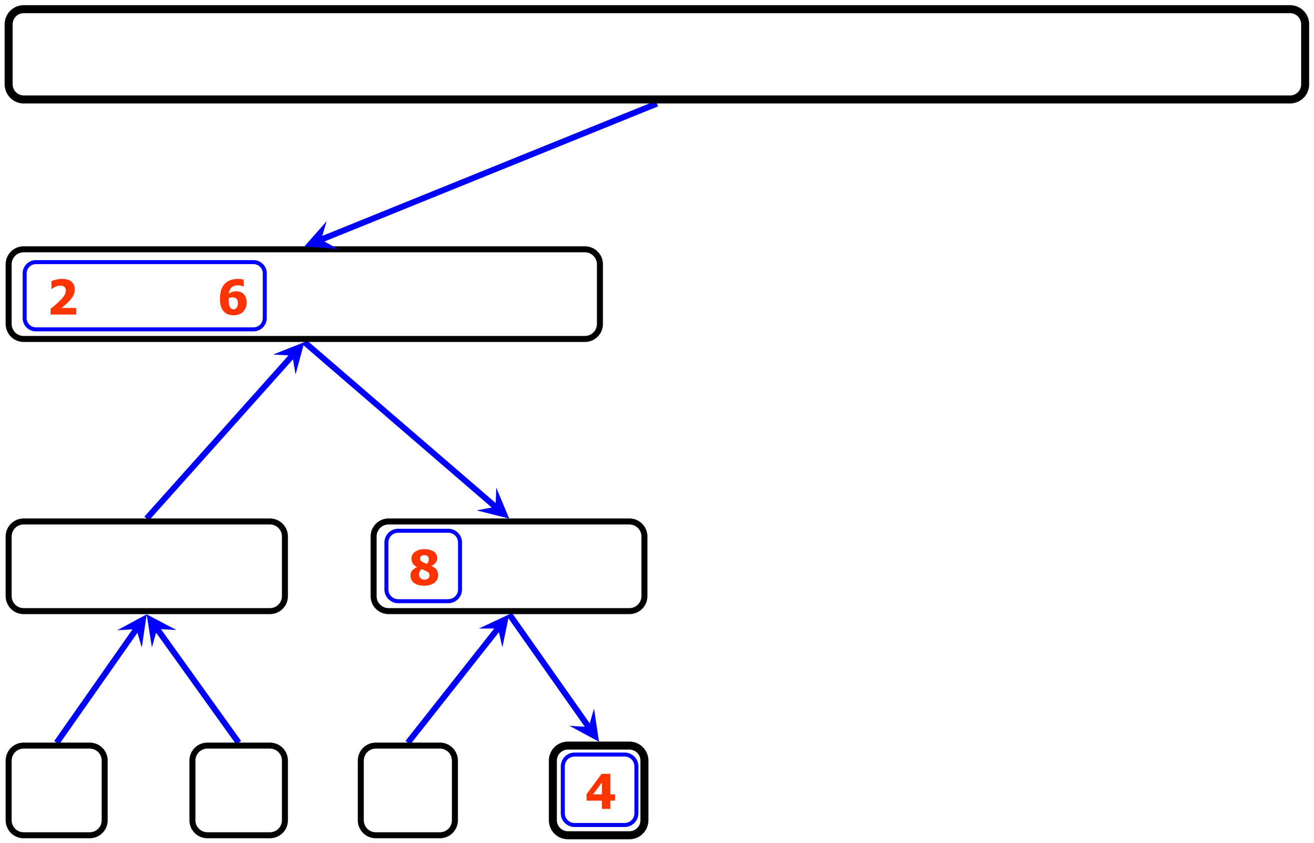
Merge-Sort Execution Example

Recursive call return



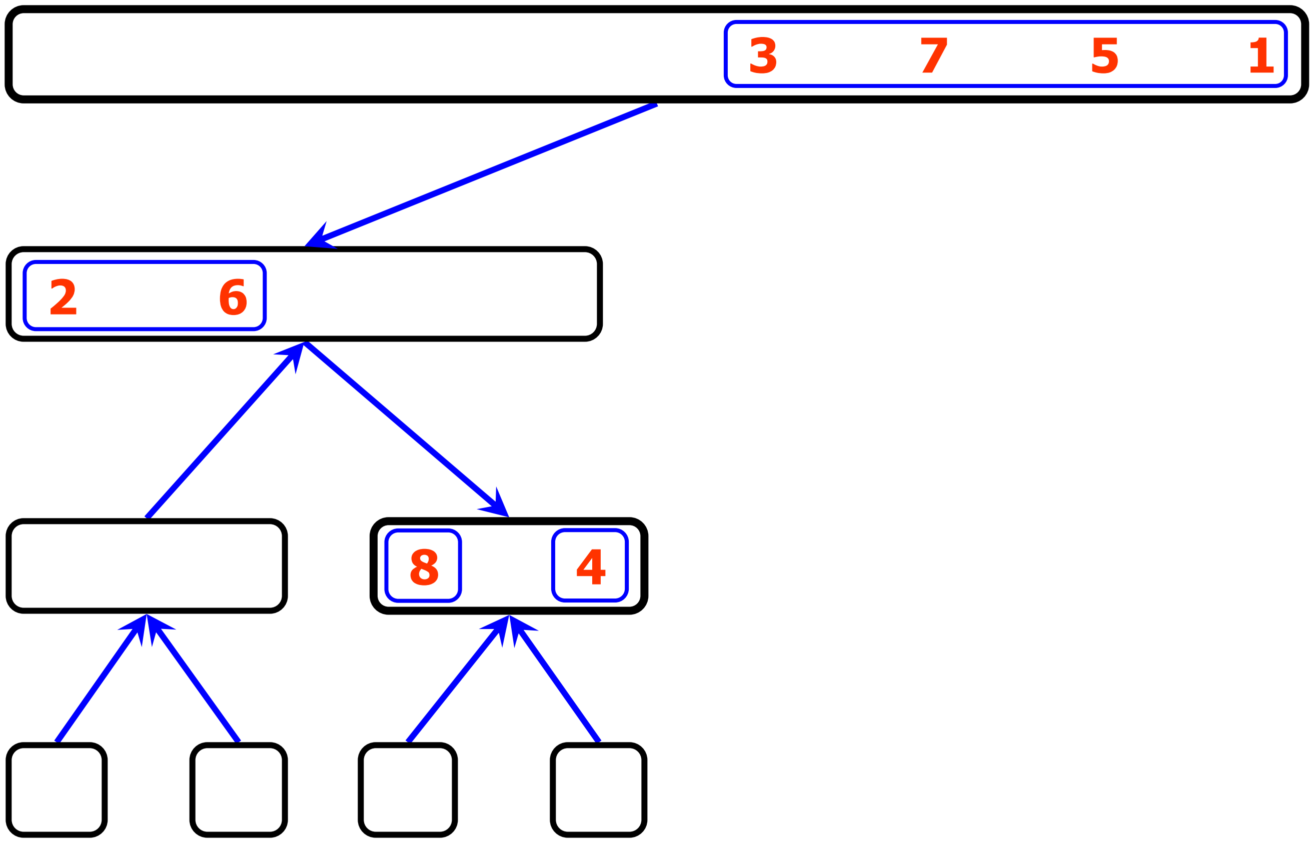
Merge-Sort Execution Example

Recursive call, base case



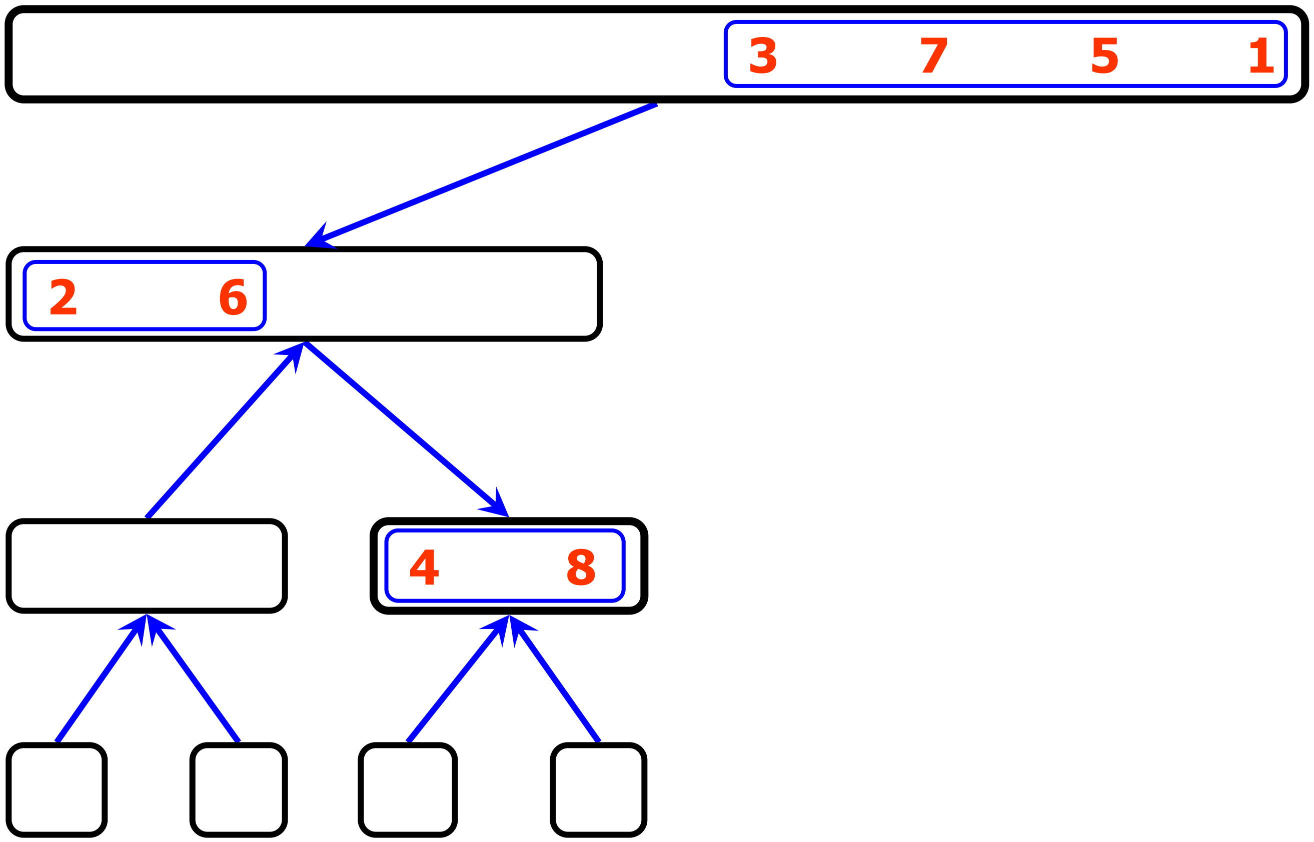
Merge-Sort Execution Example

Recursive call return



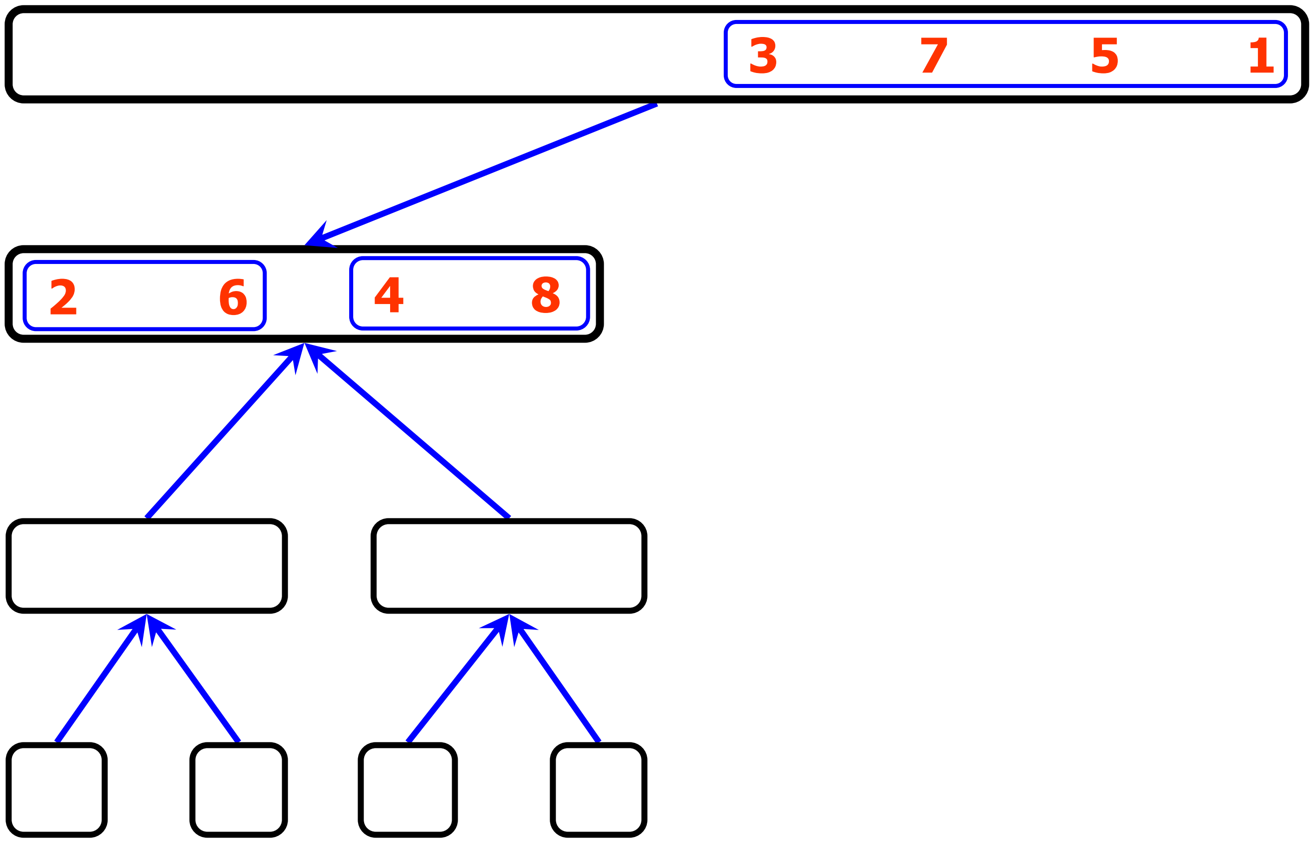
Merge-Sort Execution Example

merge



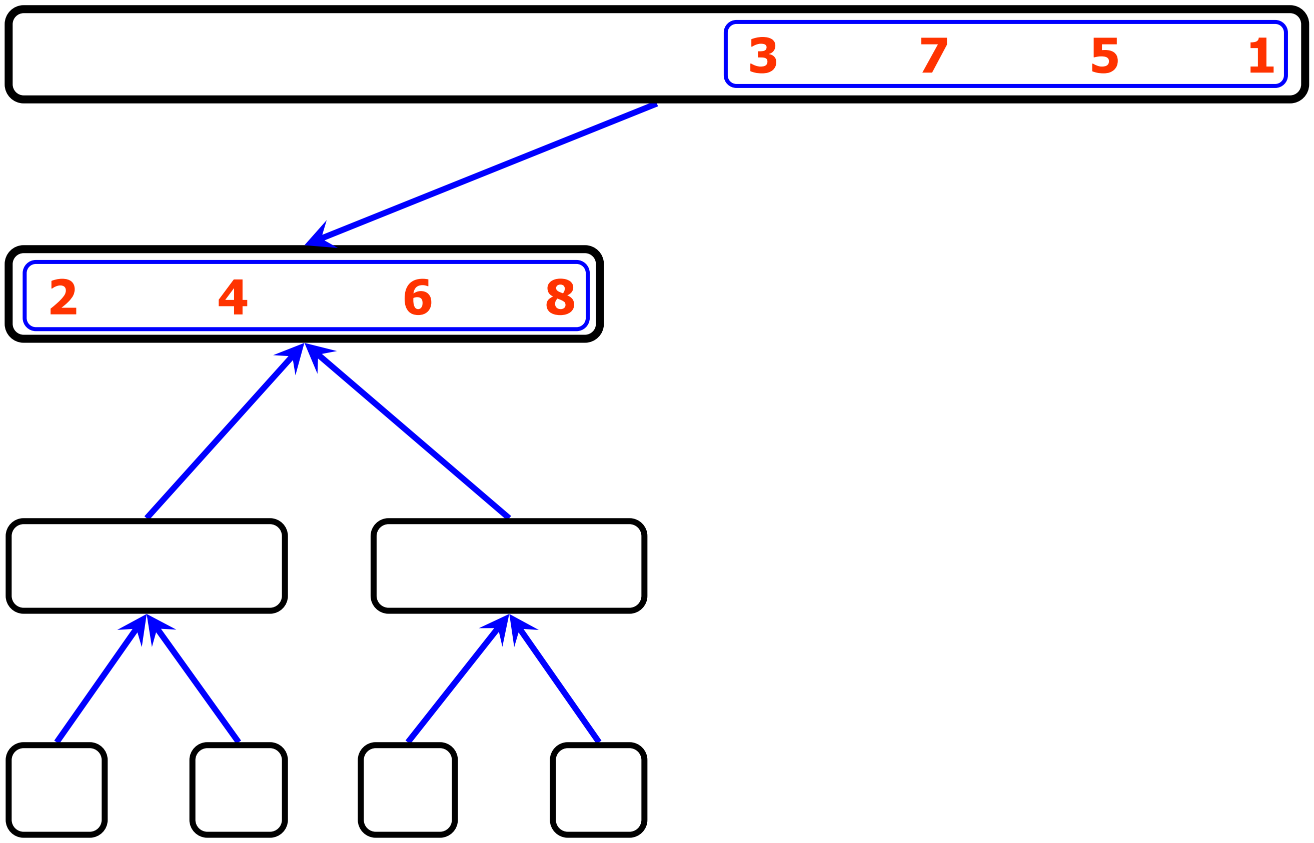
Merge-Sort Execution Example

Recursive call return



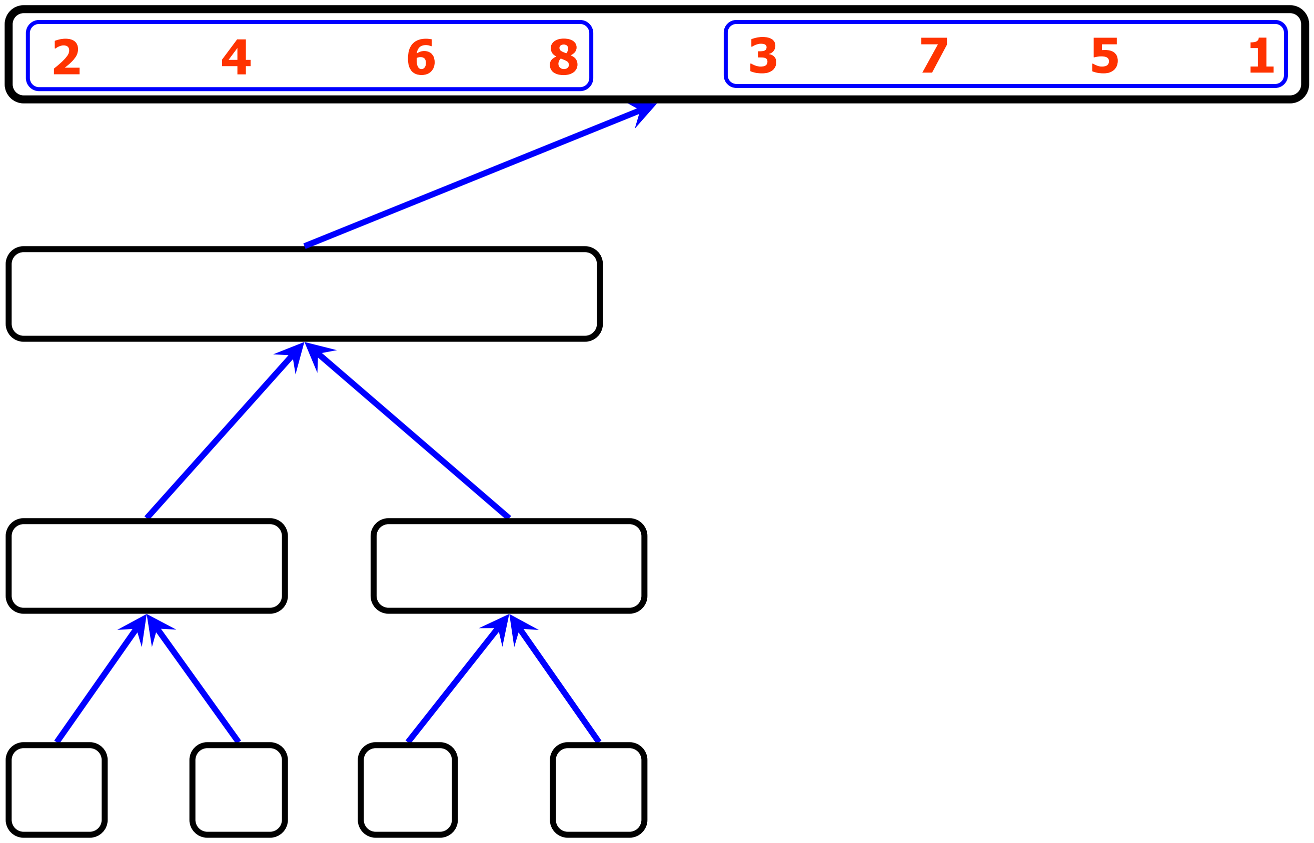
Merge-Sort Execution Example

merge



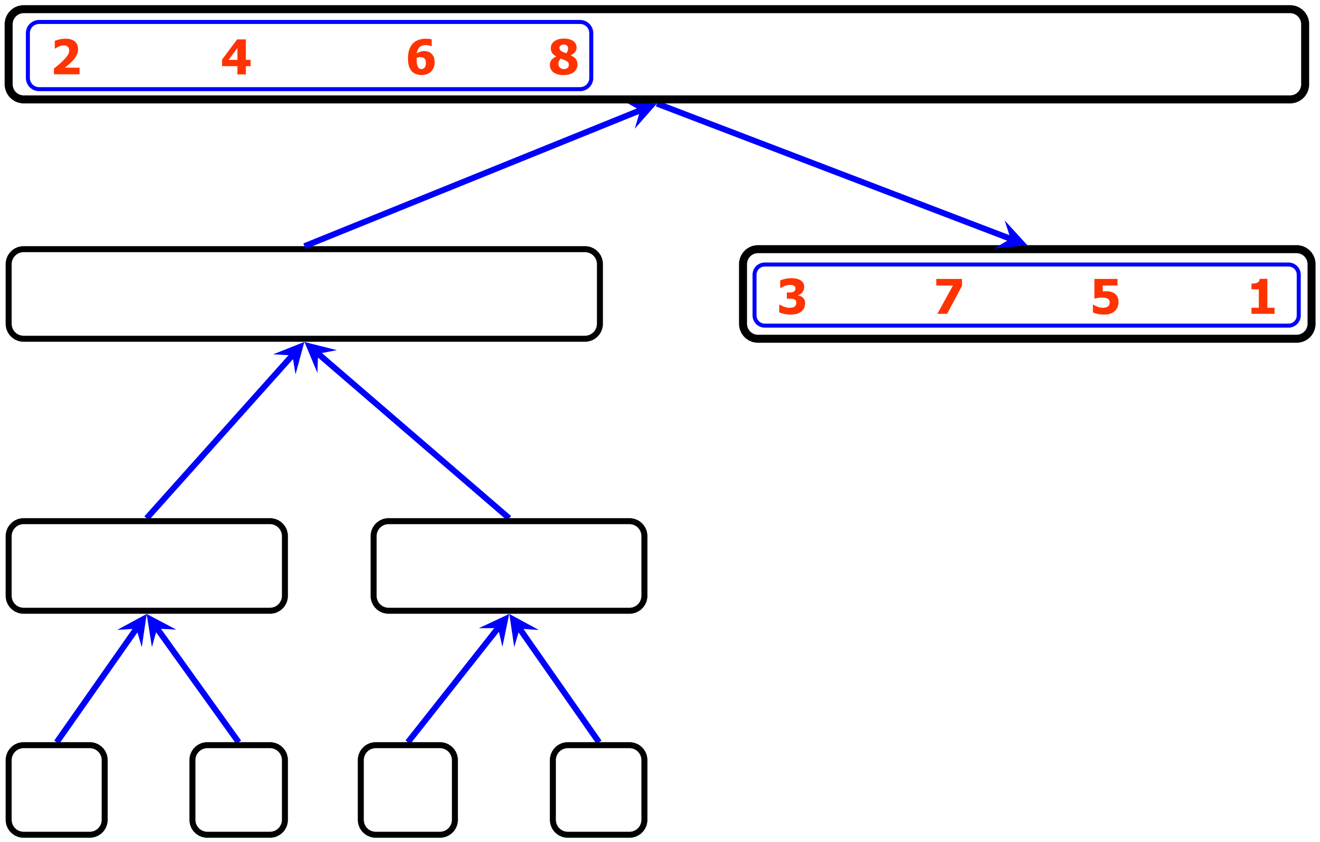
Merge-Sort Execution Example

Recursive call return

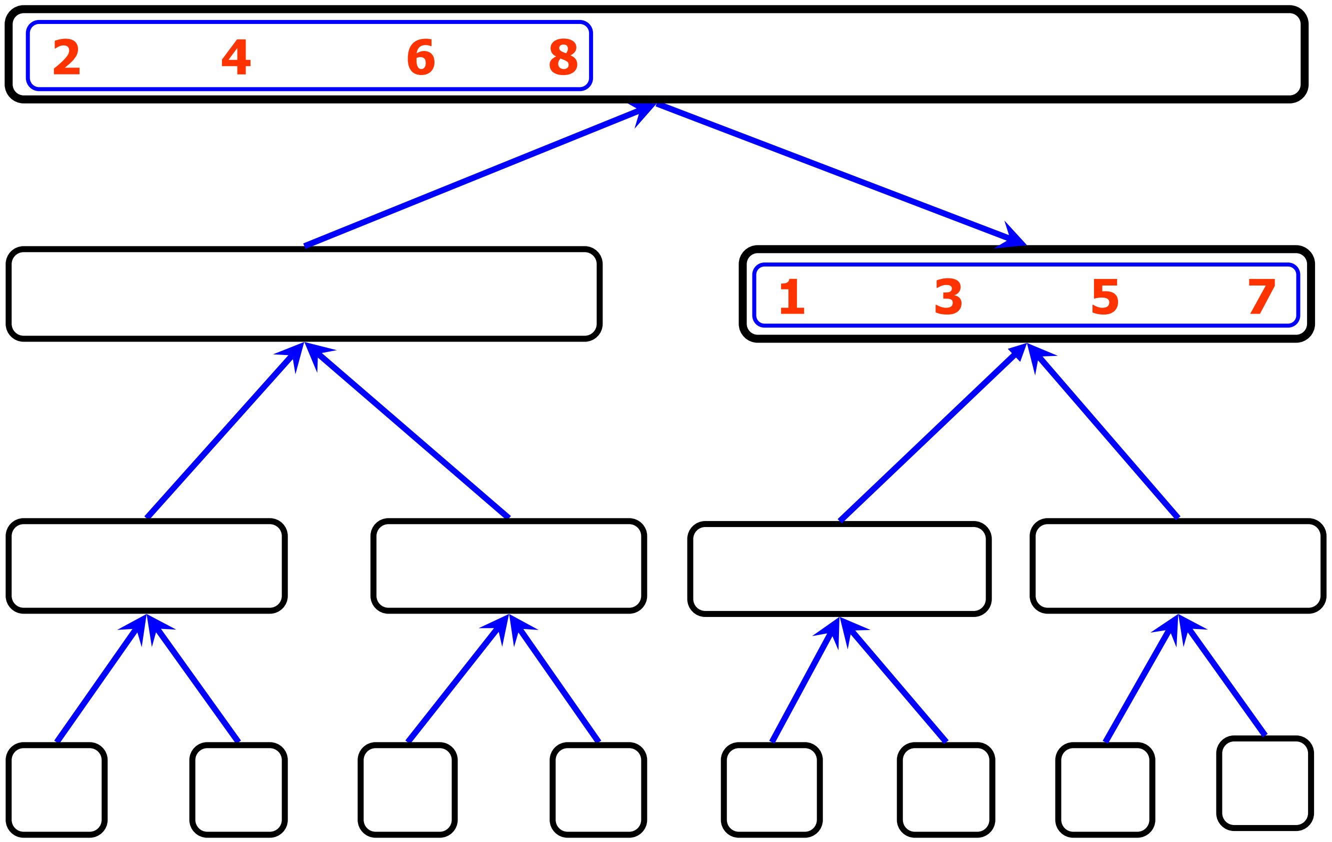


Merge-Sort Execution Example

Recursive call

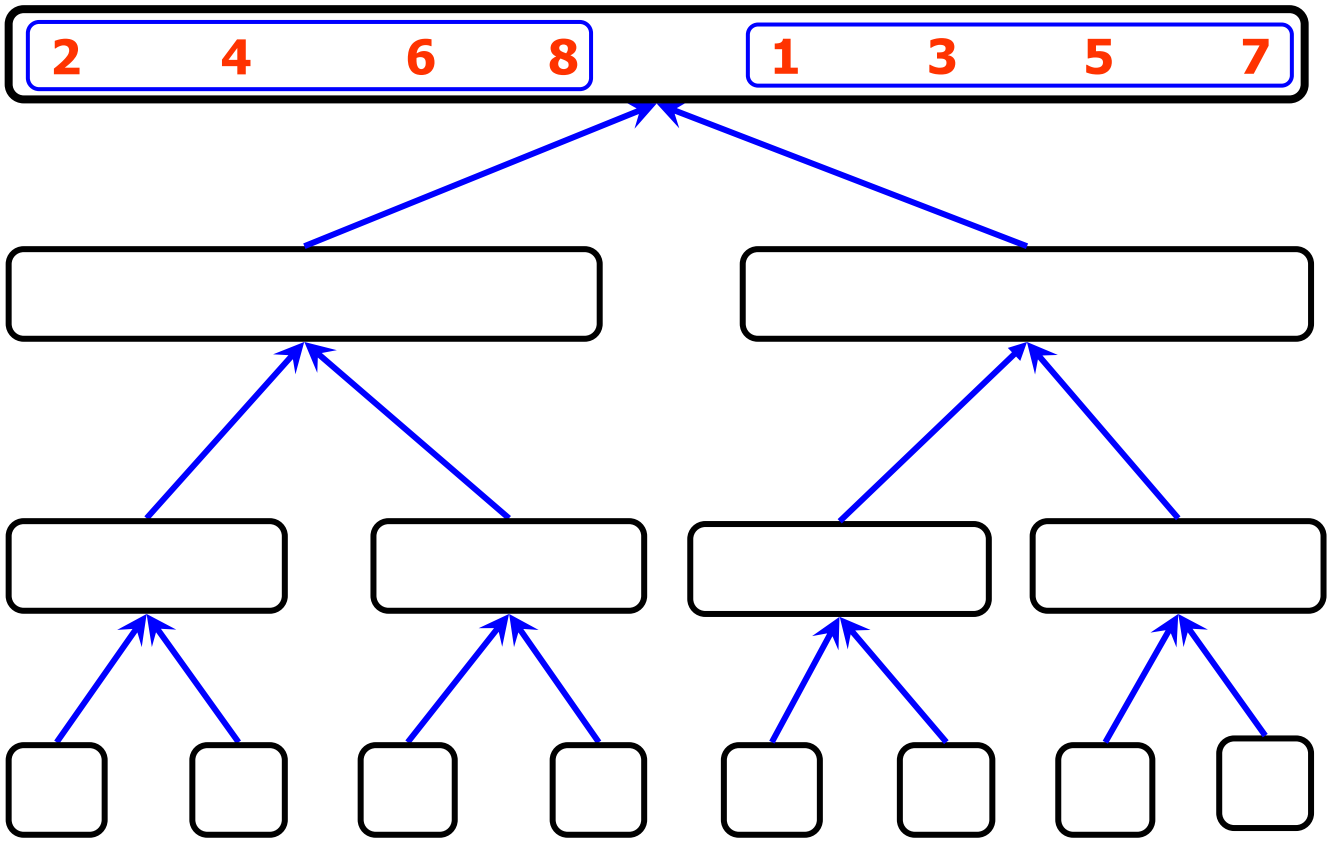


Merge-Sort Execution Example



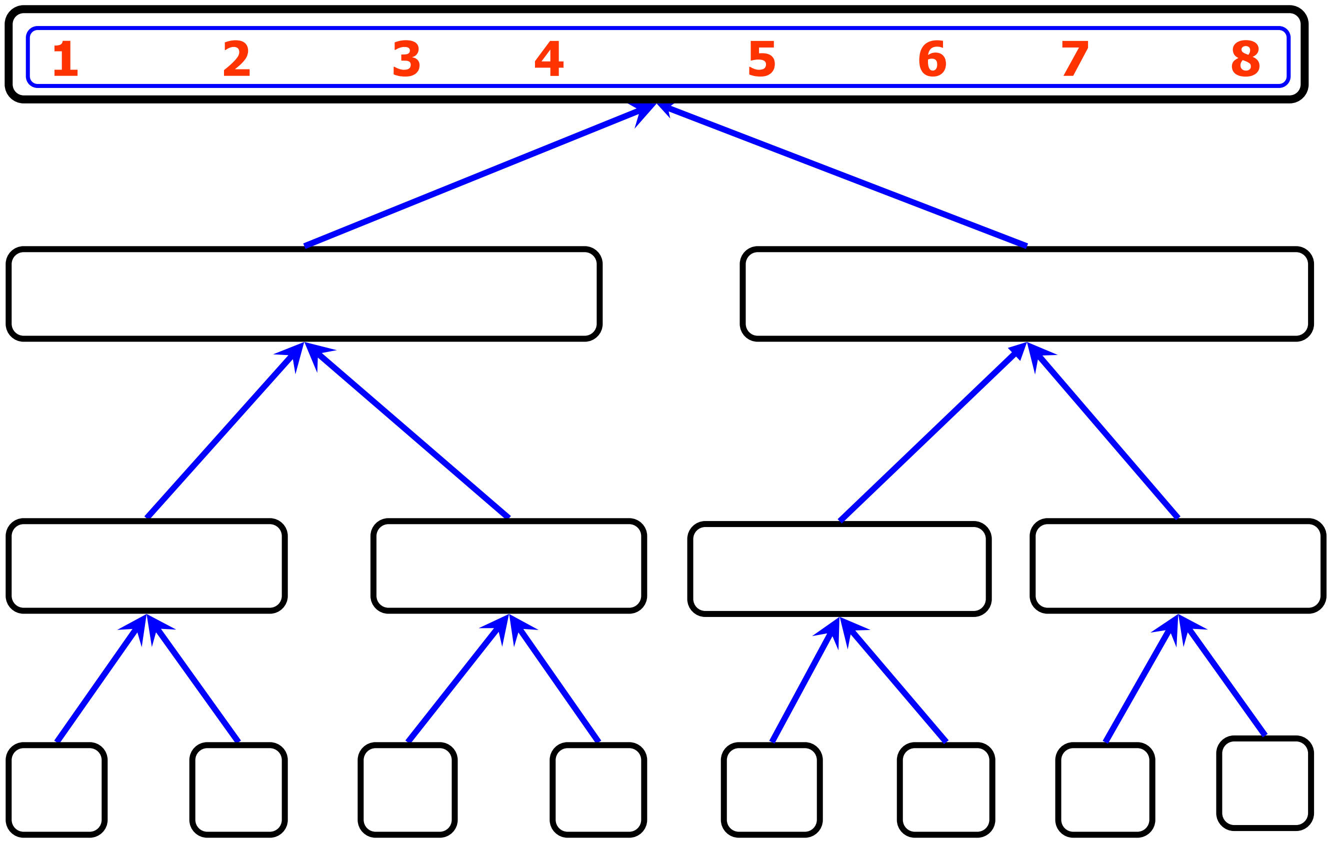
Merge-Sort Execution Example

Recursive call return



Merge-Sort Execution Example

merge

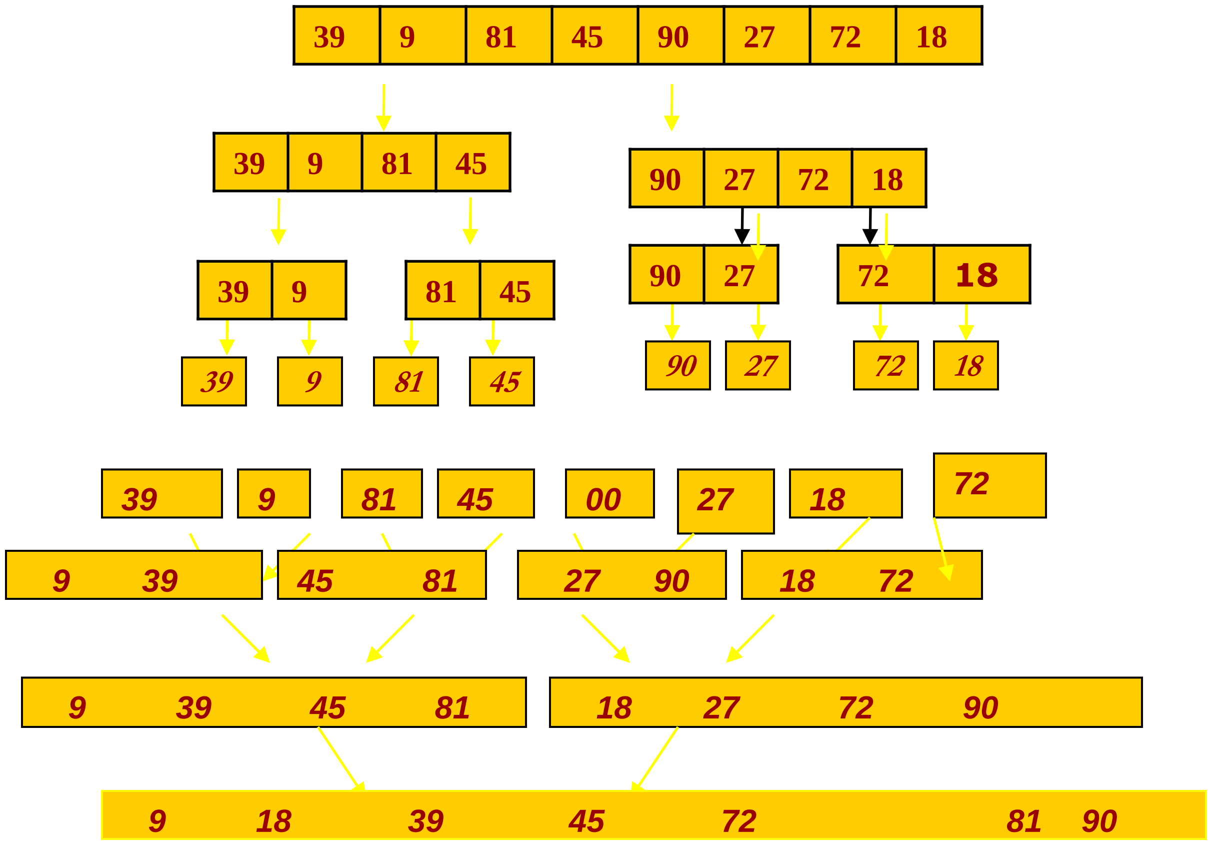


Merge Sort

Example: *Sort the array given below using merge sort*

9	9	81	45	90	27	72	18

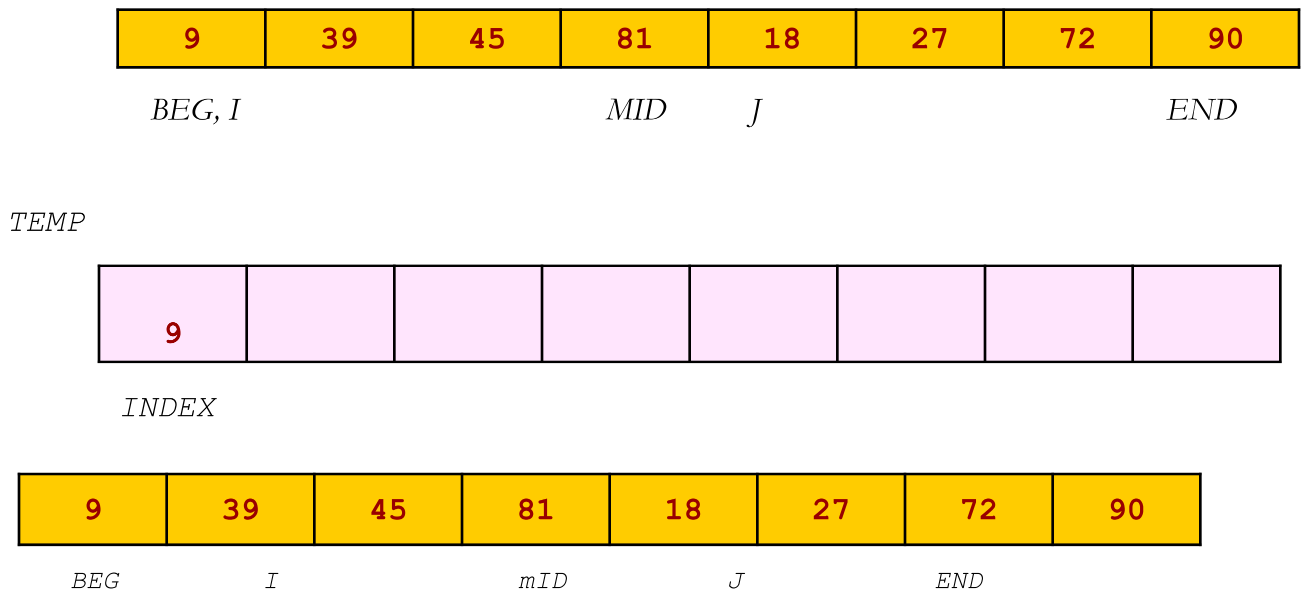
Divide and conquer the array



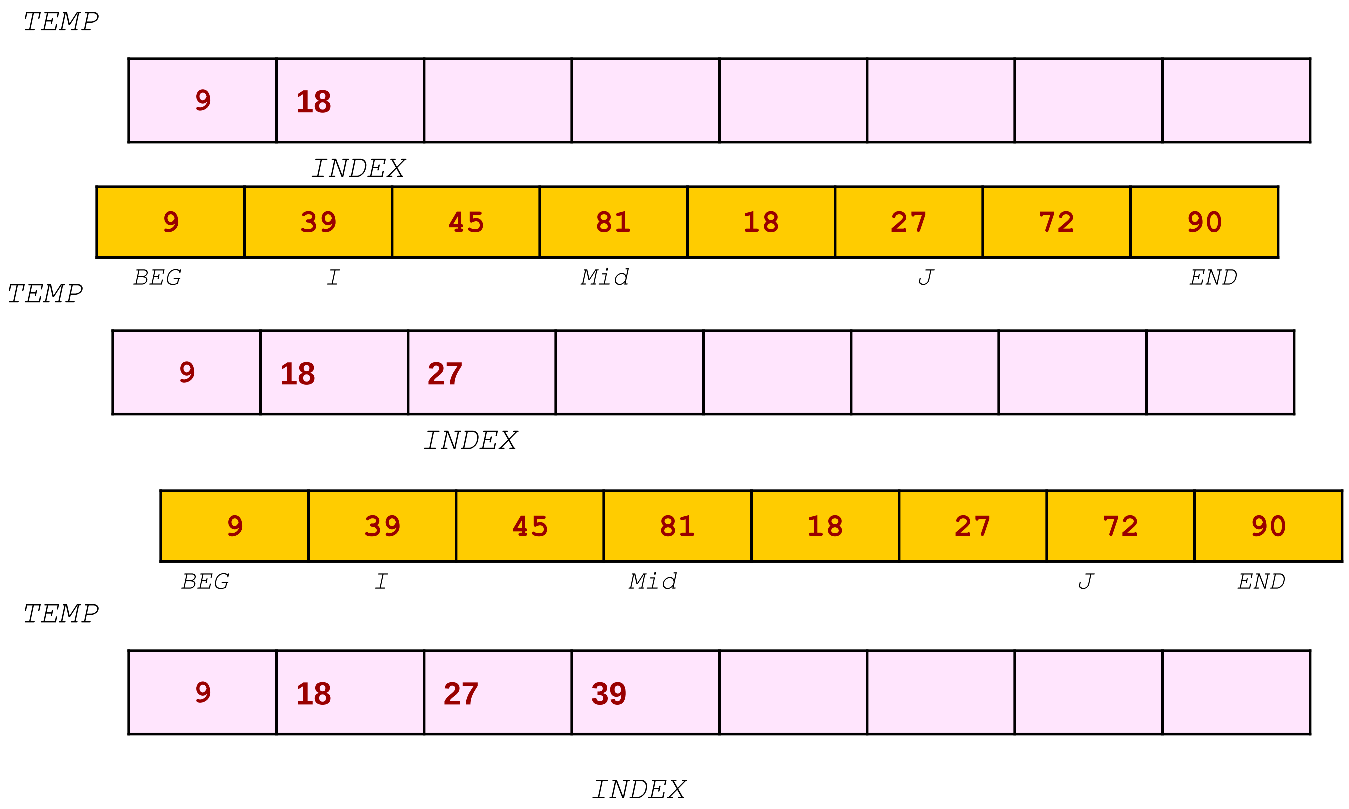
Combine the elements to form a sorted array

Merge Sort

- *To understand the merge algorithm, consider figure which shows how we merge two lists to form one list. For the sake of understanding we have taken two sub-lists each containing four elements. The same concept can be utilized to merge 4 sub-lists containing two elements, and eight sub-lists having just one element.*



Merge Sort


TEMP
9
18
INDEX
9
39
45
81
18
27
72
90
BEG
I
Mid
J
END
TEMP
9
18
27
INDEX
9
39
45
81
18
27
72
90
BEG
I
Mid
J
END
TEMP
9
18
27
39
INDEX

Merge Sort

9	39	45	81	18	27	72	90
BEG		I	Mid			J	END

9	18	27	39	45			
				INDEX			

9	39	45	81	18	27	72	90
BEG			I,Mid			J	END

9	18	27	39	45	72		
					INDEX		

When I is greater than MID copy the remaining elements of the right sub-array in TEMP

9	18	27	39	45	72	72	81 | 90
							INDEX

Merge Algorithm

```
Merge (A, left, mid, right)
{ create array temp
i = left
 j =mid+1
k=1
while (i<=mid AND j<=right)
{ if( A[i] < =A[j])
            { temp [k] = A[i]
              i=i+1
              k=k+1}
else { temp [k] = A[j]
              j=j+1
              k=k+1}
}
while (i <= mid)
    { temp[k] = A[i]
    k=k+1
    i=i+1
    }
while (j<= right)
    { temp[k] = A[j]
    k=k+1
    j=j+1
    }
//Copy temp array back to A//
}
```

Complexity of Merge Sort Algorithm

- The running time of the merge sort algorithm in average case and worst case can be given as O(n logn). Although algorithm merge sort has an optimal time complexity but a major drawback of this algorithm is that it needs an additional space of O(*n*) for the temporary array TEMP

Quicksort

- Another divide-and-conquer algorithm
  - The array A[p..r] is *partitioned* into two possibly empty subarrays A[p..q-1] and A[q+1..r]
    - All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively sorted by calls to quicksort
  - Unlike merge sort, no combining step: two subarrays form an already-sorted array

Quicksort

Quick Sort Algorithm: Steps on how it works:

1. Find a “pivot” item in the array. This item is the basis for comparison for a single round.
2. Start a pointer (the left pointer) at the first item in the array.
3. Start a pointer (the right pointer) at the last item in the array.
4. While the value at the left pointer in the array is less than the pivot value, move the left pointer to the right (add 1). Continue until the value at the left pointer is greater than or equal to the pivot value.
5. While the value at the right pointer in the array is greater than the pivot value, move the right pointer to the left (subtract 1). Continue until the value at the right pointer is less than or equal to the pivot value.
6. If the left pointer is less than or equal to the right pointer, then swap the values at these locations in the array.
7. Move the left pointer to the right by one and the right pointer to the left by one.
8. If the left pointer and right pointer don’t meet, go to step 1.

Quicksort

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q-1);
        Quicksort(A, q+1, r);
    }
}
```

```
Initial call is quicksort(A, 1, n)
```

Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the index of the "pivot" element separating the two subarrays
- *How do you suppose we implement this?*

Partition

```
PARTITION($A, p, r$)
{
    $x = A[r]$
    $i = p - 1$
    **for** $j = p$ **to** $r - 1$
        { **if** $A[j] \leq x$
                {$i \leftarrow i + 1$
                exchange $A[i] \leftrightarrow A[j]$
                }
        }
    exchange $A[i + 1] \leftrightarrow A[r]$
    **return** $i + 1$
}
```

Partition In Words
(Alternative Approach)

- Partition(A, p, r):
  - Select an element to act as the "pivot" (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] >= pivot
  - Decrement j until A[j] <= pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Return j

Note: slightly different from previous slide

Complexity of Quick Sort Algorithm

- In the average case, the running time of the quick sort algorithm can be given as, O($n$log$n$). The partitioning of the array which simply loops over the elements of the array once, uses O(n) time.
- In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, recursive call processes a sub-array of half the size. Since, at the most only log$n$ nested calls can be made before we reach a sub-array of size 1. This means that the depth of the call tree is O(log$n$). And because at each level there can only be O(n), the resultant time is given as O($n$log$n$) time.
- Practically, the efficiency of the quick sort algorithm very much depends on the element is chosen as the pivot. The worst-case efficiency of the quick sort is given as $O(n^2)$. The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot.
- However, many implementations randomly choose the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of $O(n \log n)$.

Radix/ Bucket Sort

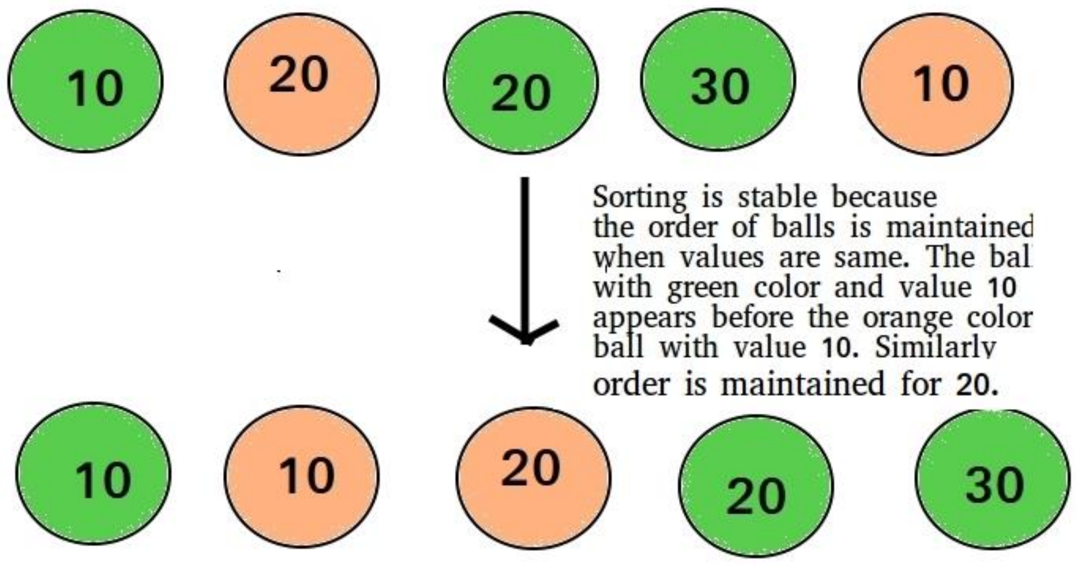
- **Radix sort** is a small method that many people intuitively use when alphabetizing a large list of names.
- Key idea:- sort on the least significant digit, then the second lsd, etc.
- **RadixSort(A, d)**

```
            for i=1 to d
                StableSort(A) on digit I
```

- Counting sort is a linear time algorithm which is stable and suitable

Stability in sorting algorithms

- A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.


10
20
20
30
10
Sorting is stable because the order of balls is maintained when values are same. The bal with green color and value 10 appears before the orange color ball with value 10. Similarly order is maintained for 20.
10
10
20
20
30

Basic Idea

Input	1st Pass	2nd Pass	3rd Pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algorithm (using buckets)

```
RadixSort (A,n)
{ max = findmax (A,n)
m = No of digits in max
k=1
while (k<=m)
{ i =1
Initialize buckets
while (i<=n)
    { d = digit at kth place in A[i]
    Add A[i] to the dth bucket
    i=i+1
    }
Collect the numbers bucket by bucket starting at bucket 0 and store them in A
k=k+1
}
}
```

Radix/Bucket Sort

Example: Sort the numbers given below using radix sort.

345,654, 924, 123, 567, 472, 555, 808, 911

In the first pass the numbers are sorted according to the digit at one's place. The buckets are pictured upside down as shown below.

Radix/Bucket Sort

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

Radix/Bucket Sort

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

Radix/Bucket Sort

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

After this pass, the numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as, ***123, 345, 472, 555, 567, 654, 808, 911, 924.***

Analysis of Radix Sort

- Each key is looked at once for each digit (or letter if the keys are alphabetic) of the longest key. Hence, if the longest key has **m** digits and there are **n** keys, radix sort has order **O(m.n)**.
- However, if we look at these two values, the size of the keys will be relatively small when compared to the number of keys. For example, if we have six-digit keys, we could have a million different records.
- Here, we see that the size of the keys is not significant, and this algorithm is of linear complexity **O(n)**.

Advantages & Disadvantages of various sorting algorithms

- Bubble sort:
  - Easy to code
  - O(n^2) worst case
  - O(n^2) average (equally-likely inputs) case
  - Not suitable for real-life applications.

- Selection sort:
  - Easy to code
  - performs well on a small listFast on nearly-sorted inputs
  - O(n^2) worst case
  - O(n^2) average (equally-likely inputs) case
  - O(n^2) reverse-sorted case

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - Doesn’t sort in place

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children’s keys
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

- Quick sort:
    - Divide-and-conquer:
        - Partition array into two subarrays, recursively sort
        - All of first subarray < all of second subarray
        - No merge step needed!
    - O(n lg n) average case
    - Fast in practice
    - O(n^2) worst case
        - Naïve implementation: worst case on sorted input
        - Address this with randomized quicksort

- Radix sort:
  - Linear time efficiency
  - Fast when the keys are short i.e. when the range of the array elements is less
  - Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. Hence , for every different type of data it needs to be rewritten.
  - Additional Space Requirement