CS101 Algorithms and Data Structures

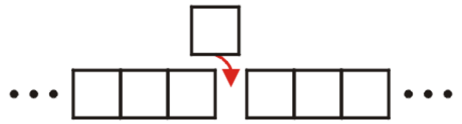
Merge Sort

Textbook Ch 1.4, 7

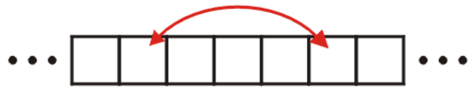
Classifications

The operations of a sorting algorithm are based on the actions performed:

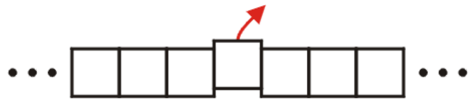
- Insertion



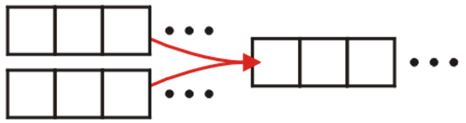
- Exchanging



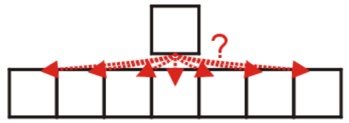
- Selection



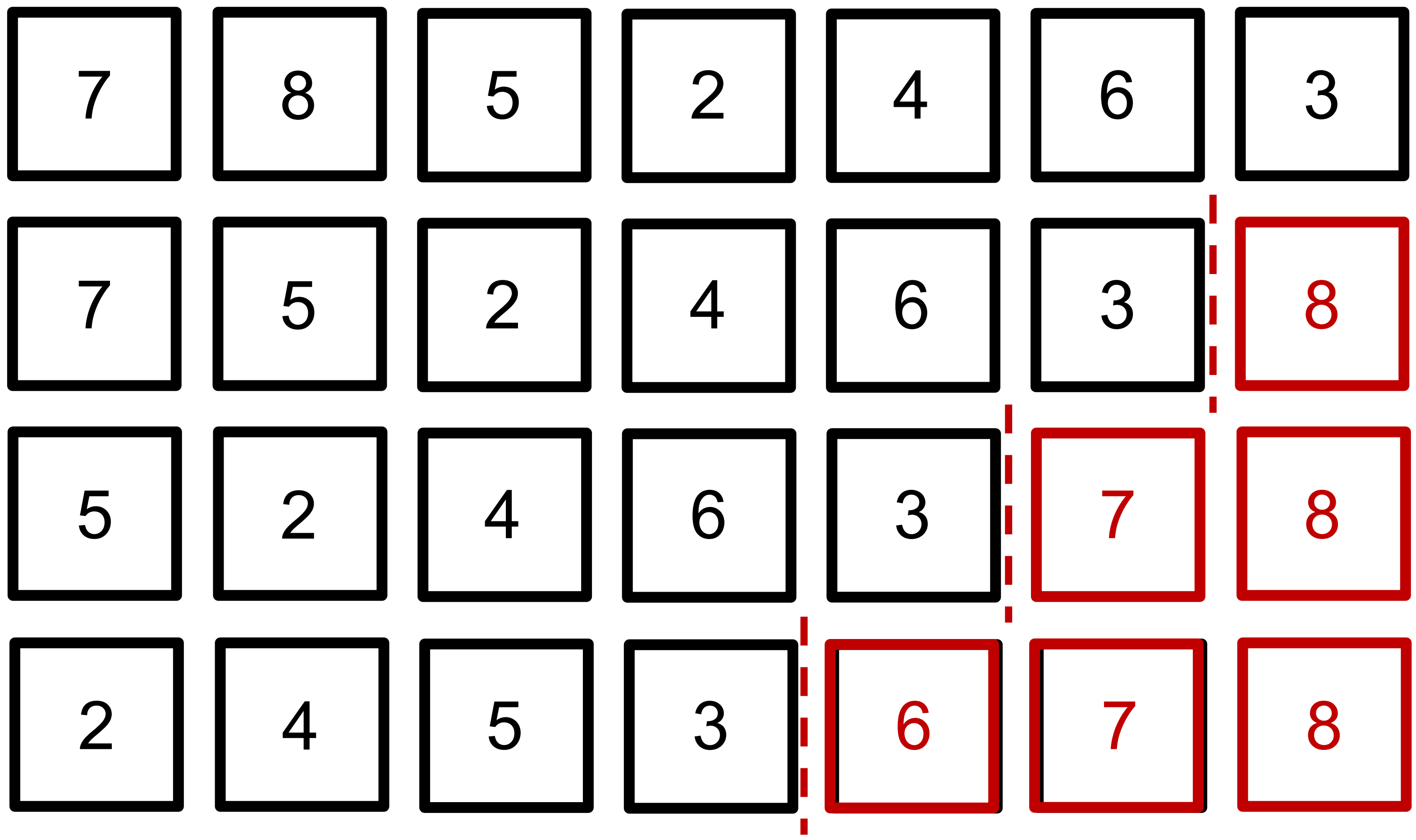
- Merging



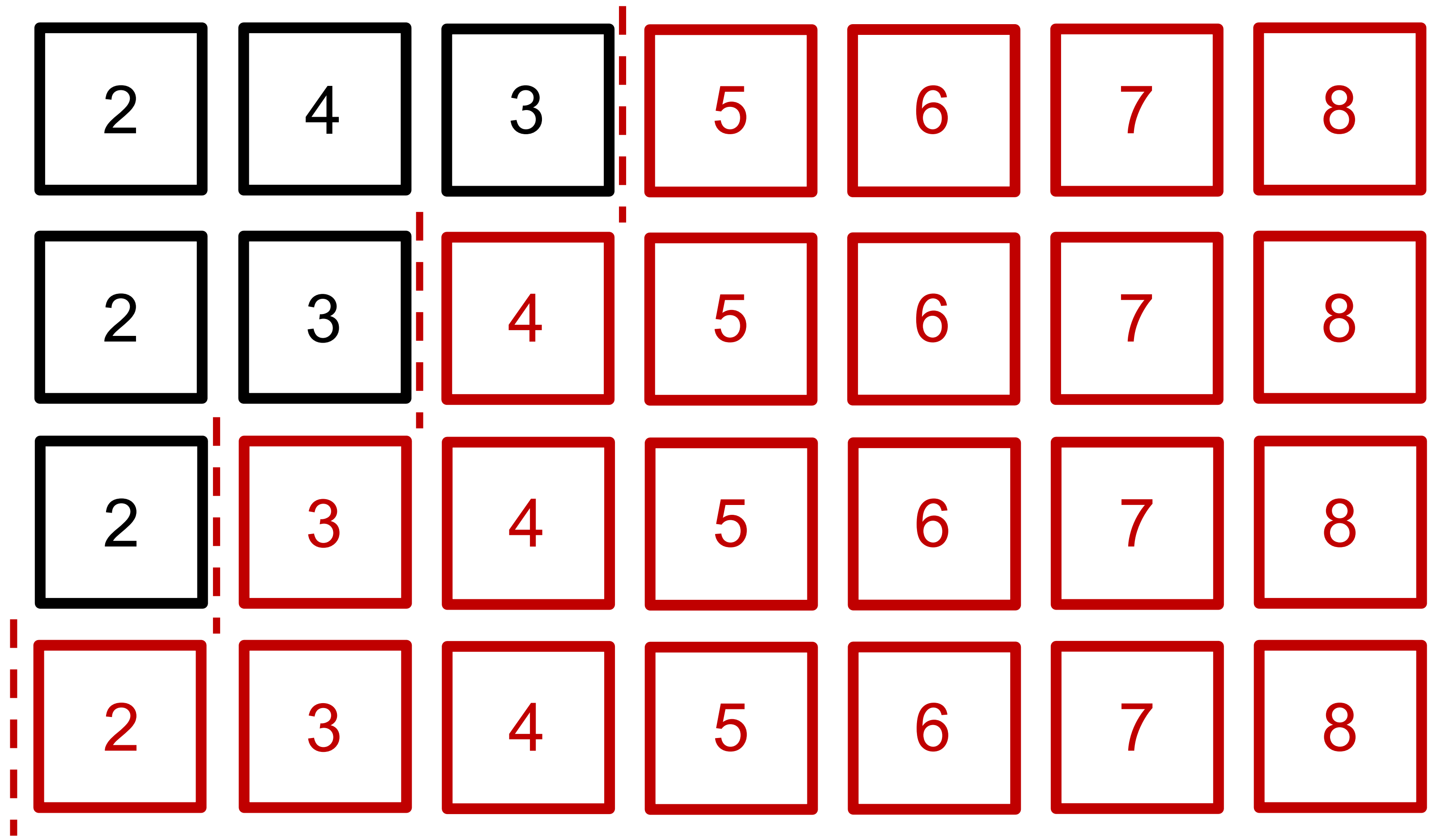
- Distribution



Which sort algorithm?



Bubble Sort

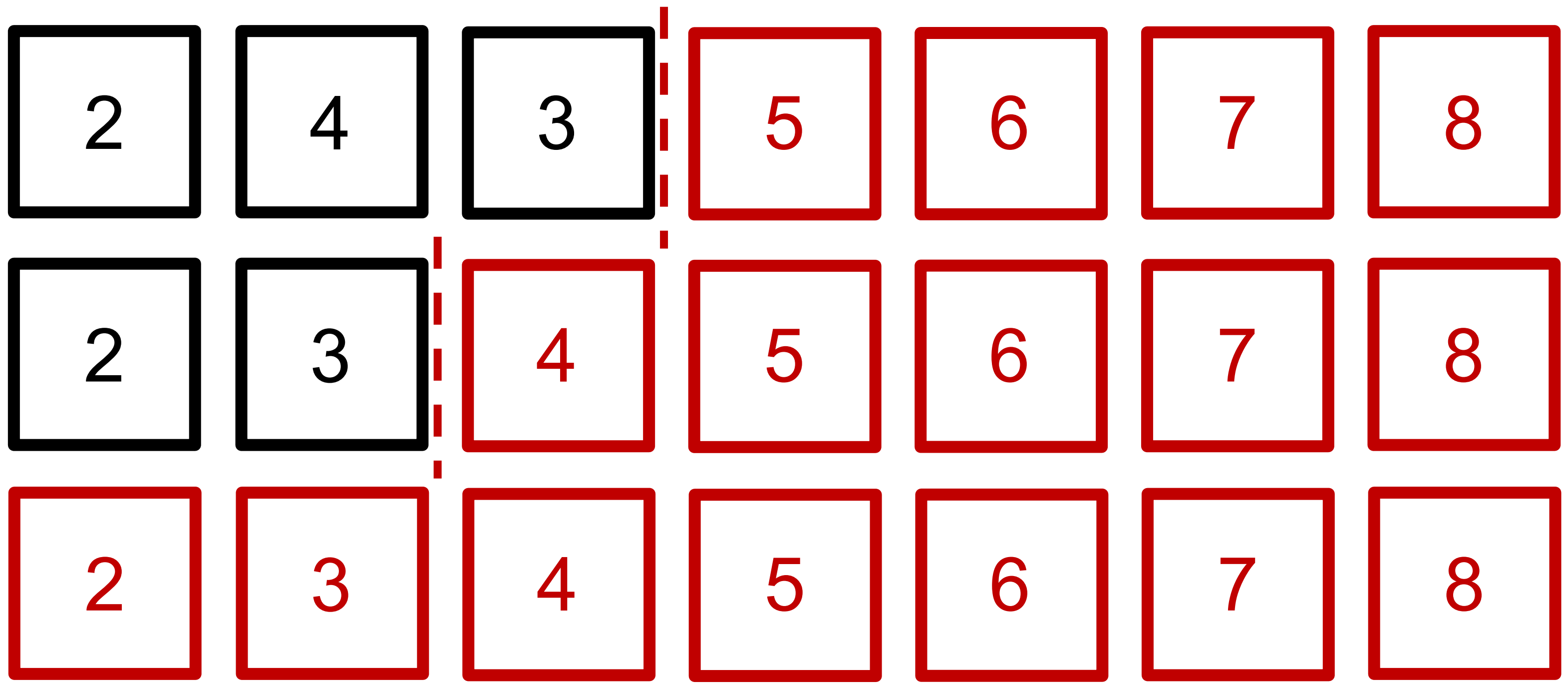


Flagged Bubble Sort

Check if the list is sorted (no swaps)

```
template <typename Type>
void bubble( Type *const array, int const n ) {
   for ( int i = n - 1; i > 0; --i ) {
         Type max = array[0];
         bool sorted = true;
         for ( int j = 1; j <= i; ++j ) {
               if ( array[j] < max ) {
                     array[j - 1] = array[j];
                     sorted = false;
               } else {
                     array[j – 1] = max;
                     max = array[j];
               }
         }
         array[i] = max;
         if ( sorted ) {
               break;
         }
   }
}
```

Flagged Bubble Sort

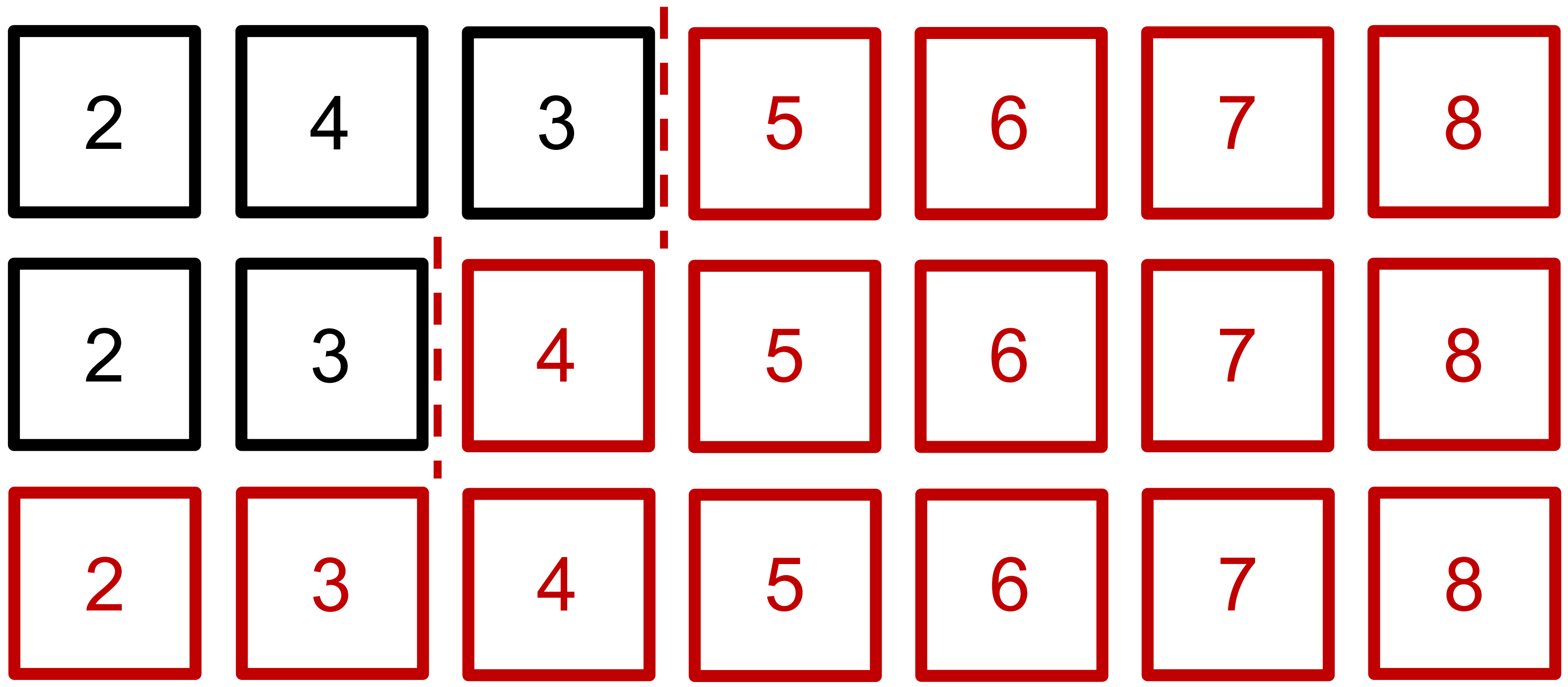


Range-limiting Bubble Sort

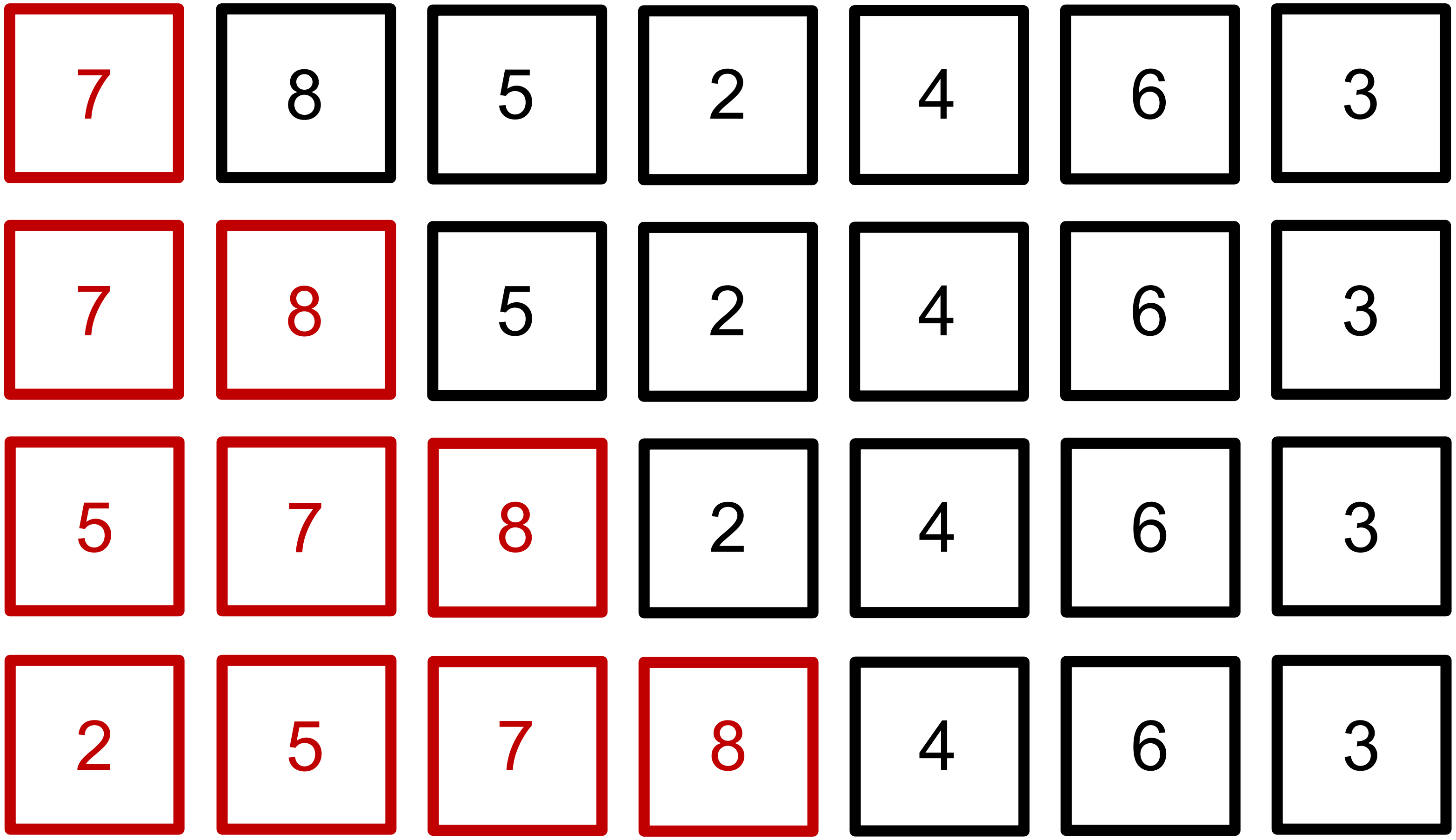
Update **`i`** to at the place of the last swap

```
template <typename Type>
void bubble( Type *const array, int const n ) {
  for ( int i = n - 1; i > 0; ) {
        Type max = array[0];
        int ii = 0;
        for ( int j = 1; j <= i; ++j ) {
              if ( array[j] < max ) {
                    array[j - 1] = array[j];
                    ii = j - 1;
              } else {
                    array[j – 1] = max;
                    max = array[j];
              }
        }
        array[i] = max;
        i = ii;
  }
}
```

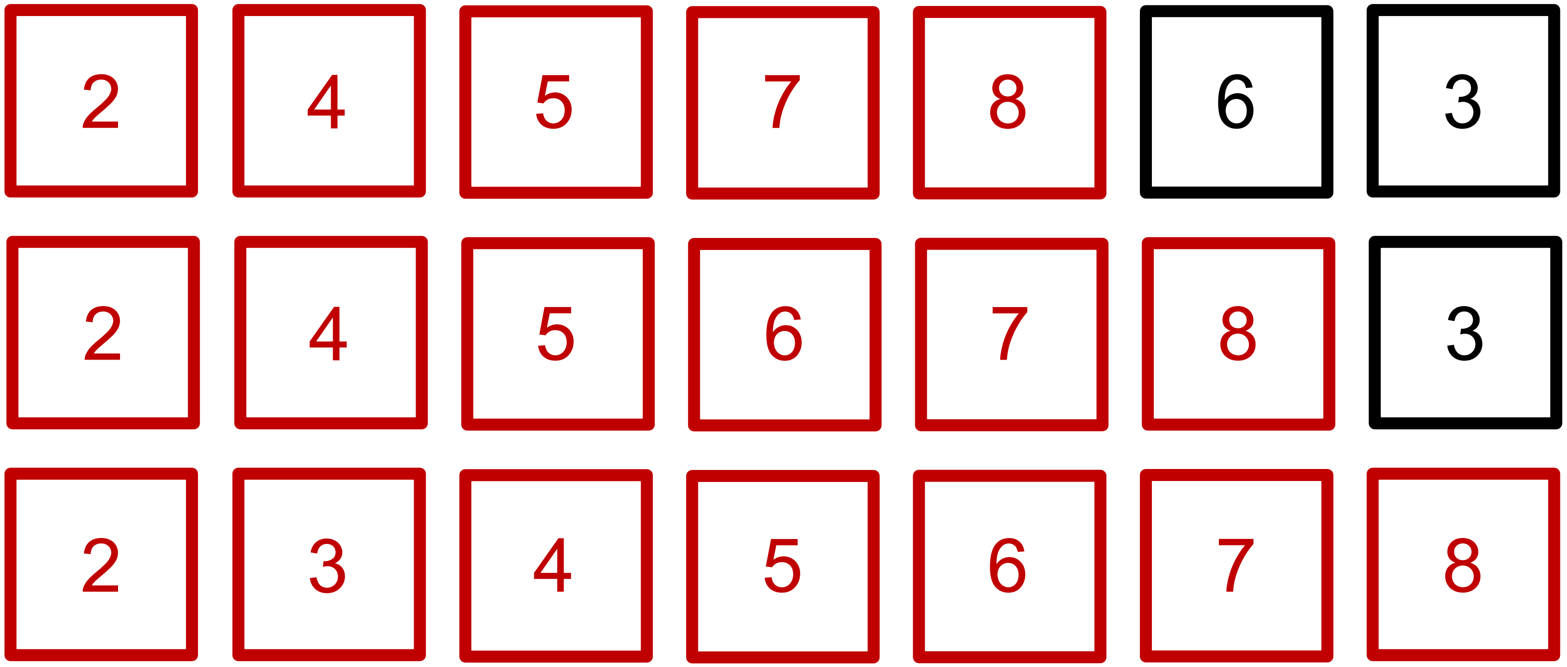
Range-limiting Bubble Sort



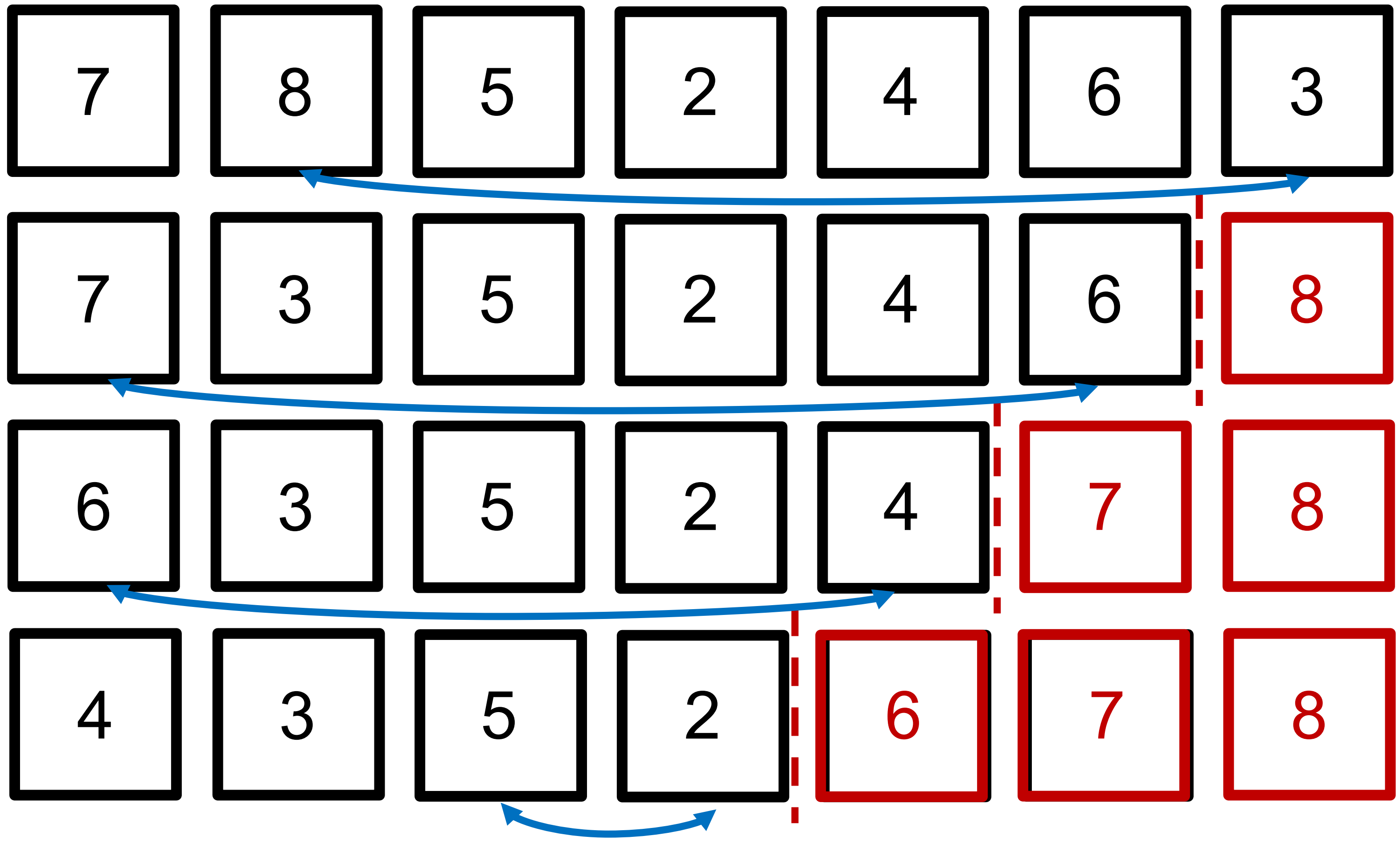
Which sort algorithm?



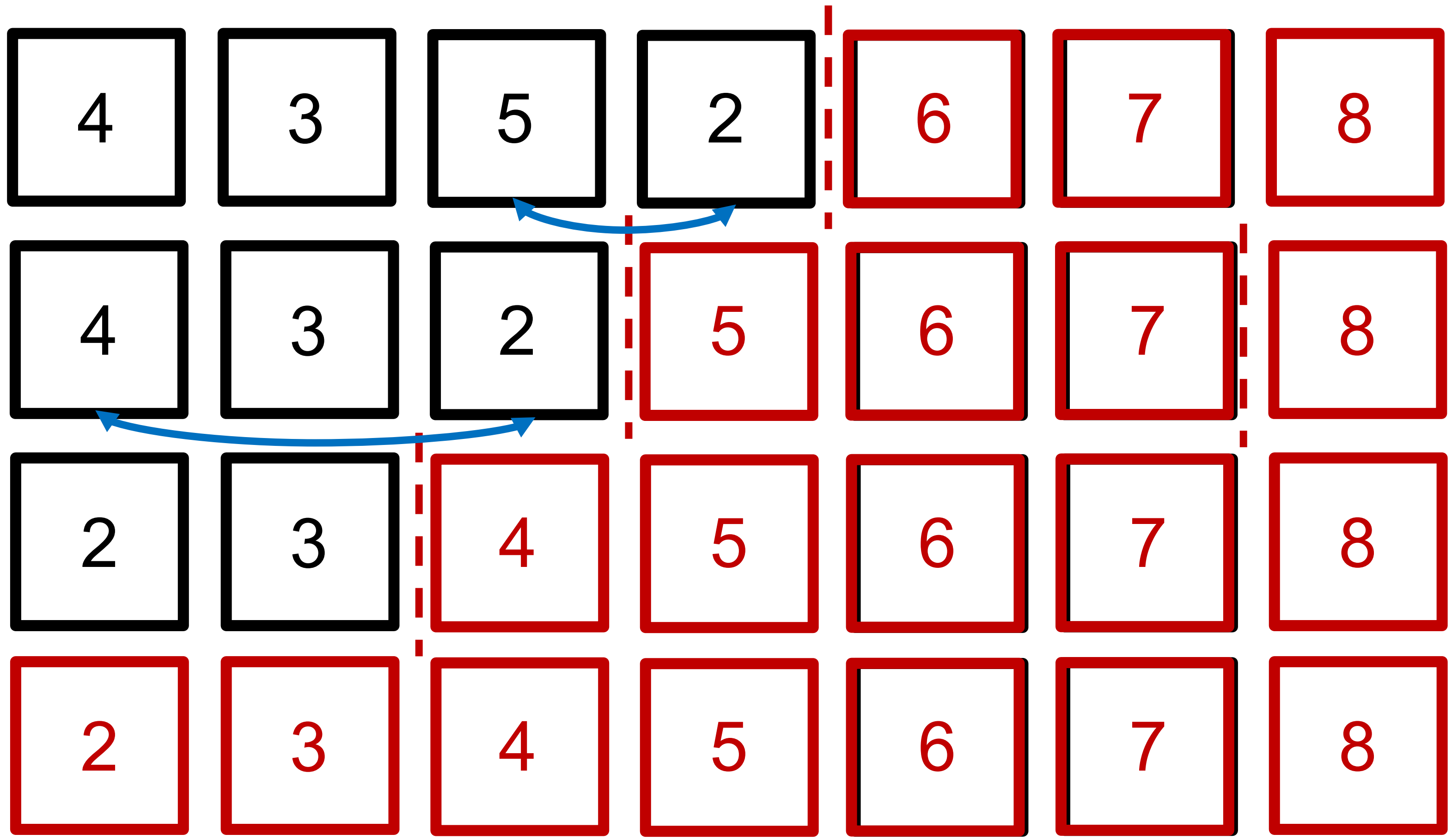
Insertion Sort



Which sort algorithm?



Selection Sort



Stability in sorting algorithms

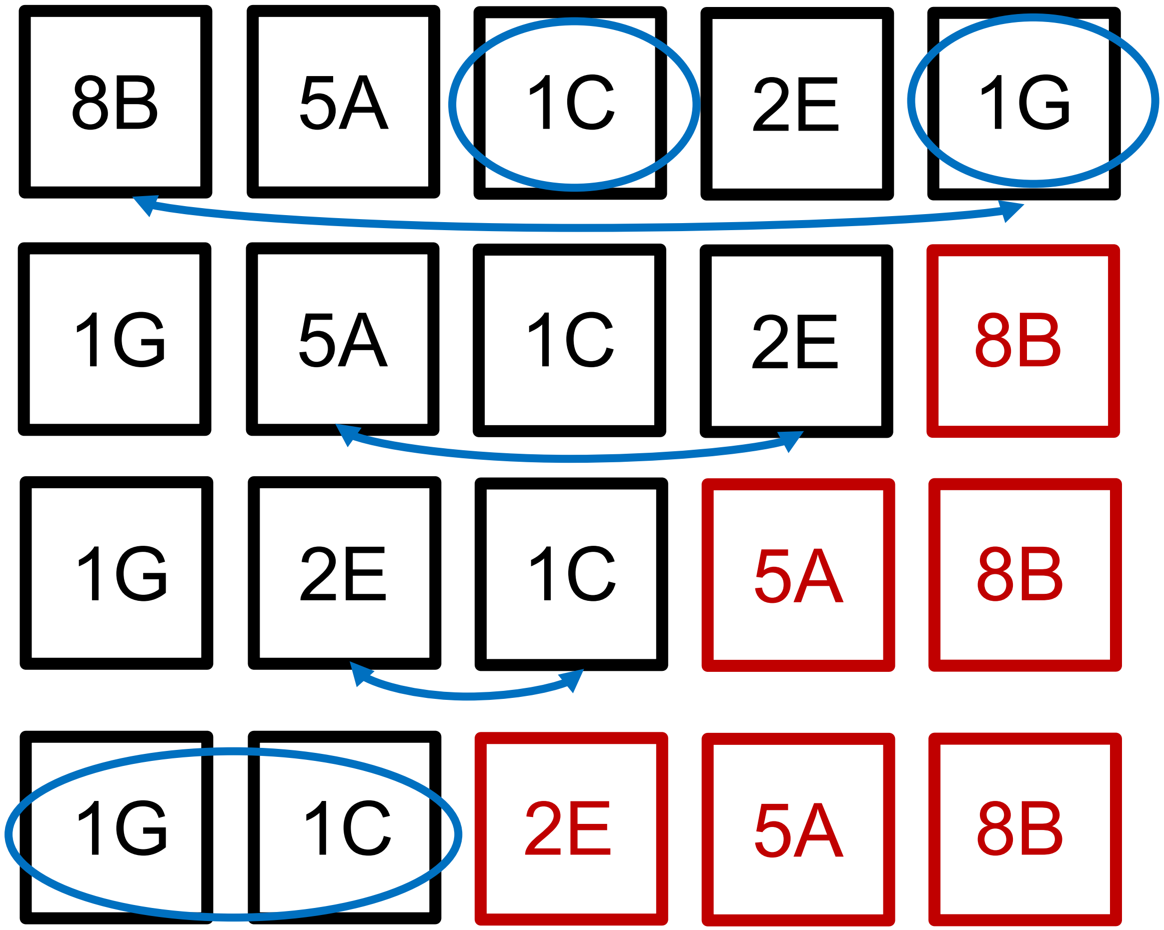
- The stability of a sorting algorithm is concerned with **how the algorithm treats equal (or repeated) elements**. Stable sorting algorithms preserve the relative order of equal elements, while unstable sorting algorithms don’t. In other words, stable sorting maintains the position of two equals elements relative to one another.

Stability in sorting algorithms

- Let A be a collection of elements and < be a strict weak ordering on the elements. Further, let B be the collection of elements in A in the sorted order. Let's consider two equal elements in A at indices i and j, i.e., $A[i]$ and $A[j]$, that end up at indices m and n respectively in B. We can classify the sorting as stable if:

$$i < j, \qquad A[i] = A[j], \qquad \text{and } \mathrm{m} < \mathrm{n}$$

Selection Sort



Outline

- Insertion sort
- Bubble sort
- Merge sort

Outline

This topic covers merge sort

- A recursive divide-and-conquer algorithm
- Merging two lists
- The merge sort algorithm
- A run-time analysis

Merge Sort

The merge sort algorithm is defined recursively:

- If the list is of size 1, it is sorted—we are done;
- Otherwise:
  - Divide an unsorted list into two sub-lists,
  - Sort each sub-list recursively using merge sort, and
  - Merge the two sorted sub-lists into a single sorted list

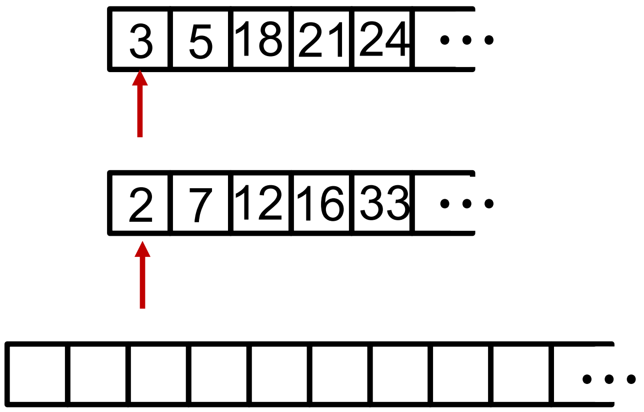
This strategy is called *divide-and-conquer*

Question: How can we merge two sorted sub-lists into a single sorted list?

Merging Example

Consider the two sorted arrays and an empty array

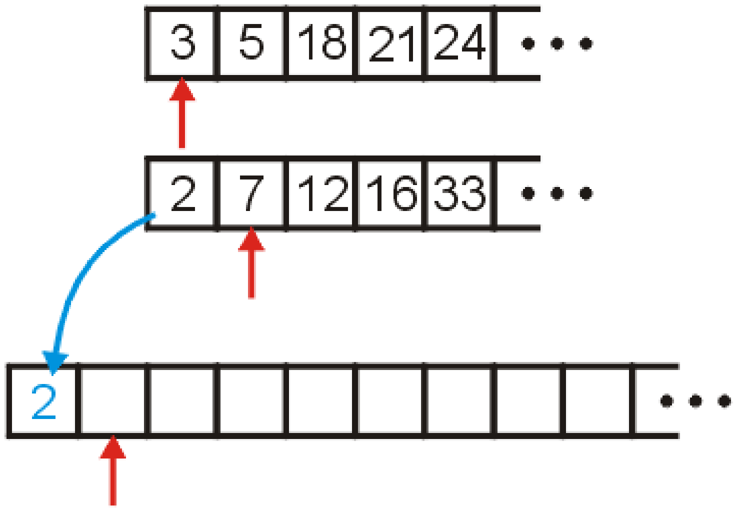
Define three indices at the start of each array



Merging Example

We compare 2 and 3: 2 < 3

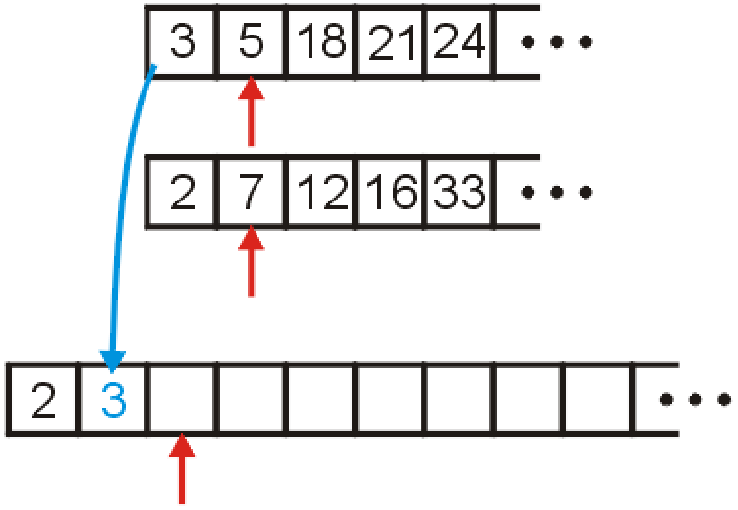
- Copy 2 down
- Increment the corresponding indices



Merging Example

We compare 3 and 7

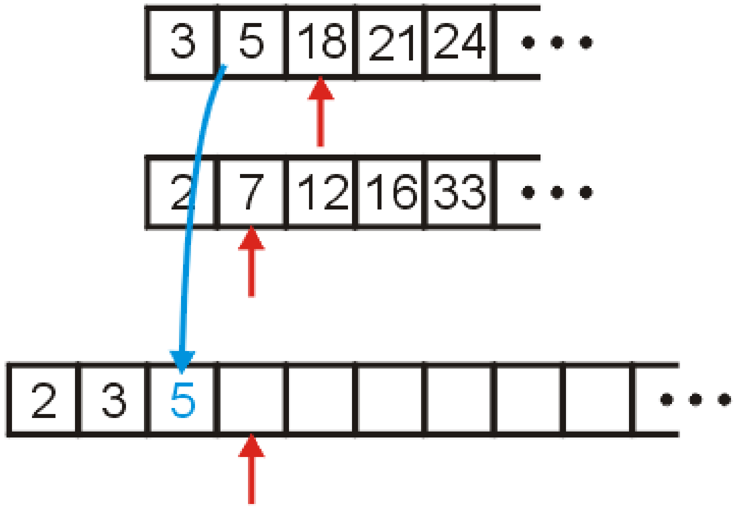
- Copy 3 down
- Increment the corresponding indices



Merging Example

We compare 5 and 7

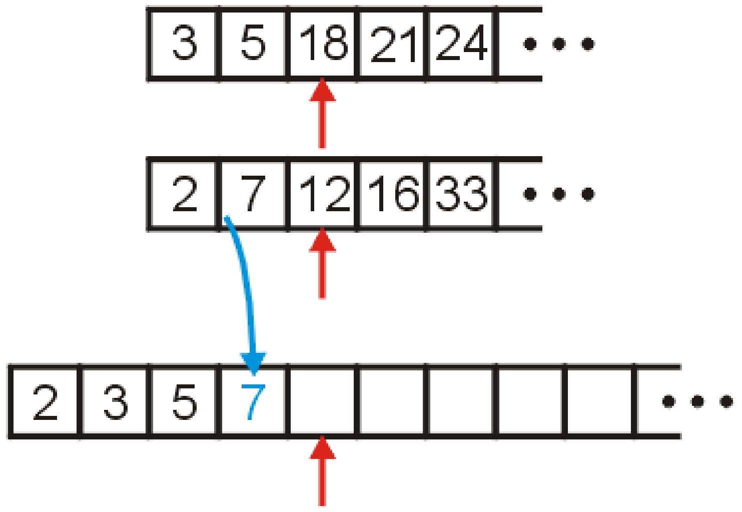
- Copy 5 down
- Increment the appropriate indices



Merging Example

We compare 18 and 7

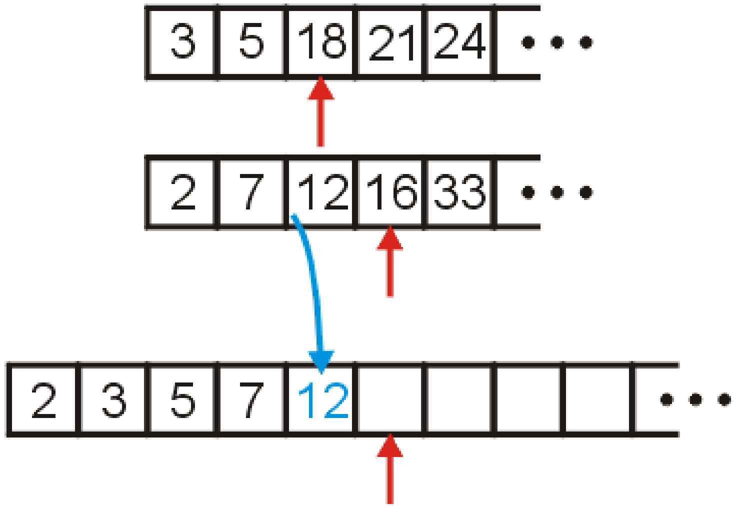
- Copy 7 down
- Increment...

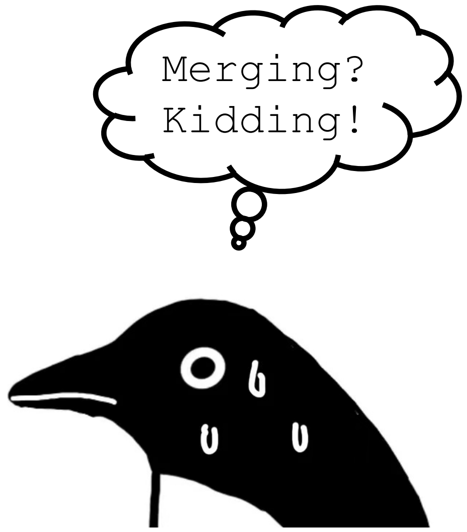


Merging Example

We compare 18 and 12

- Copy 12 down
- Increment...

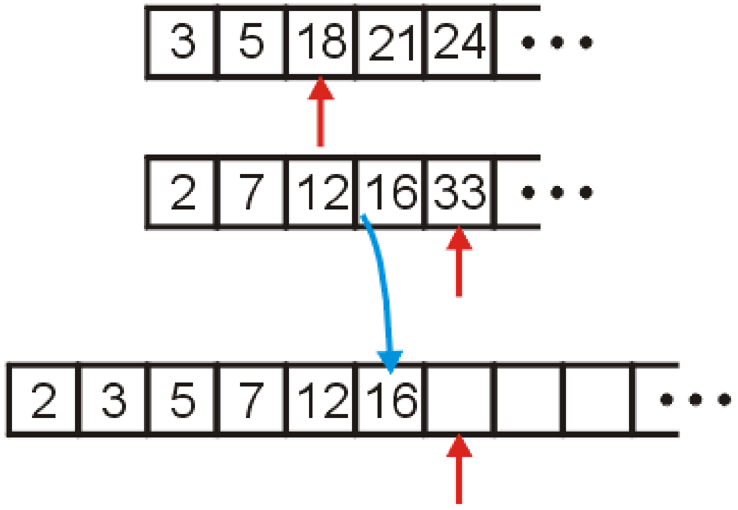




Merging Example

We compare 18 and 16

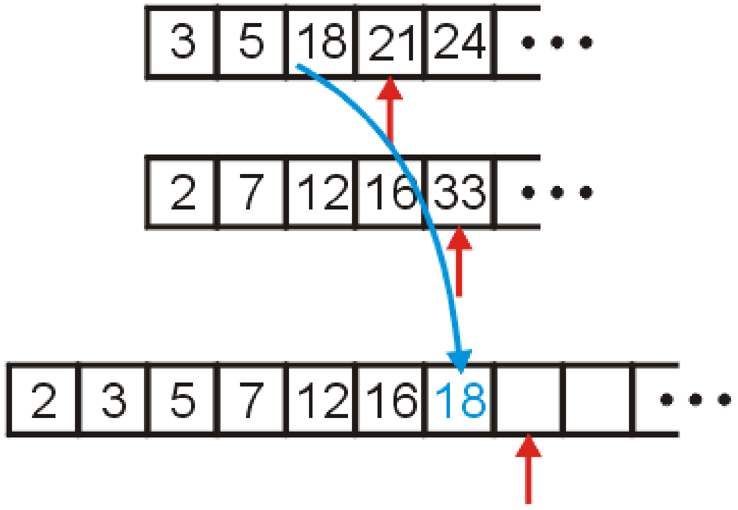
- Copy 16 down
- Increment...



Merging Example

We compare 18 and 33

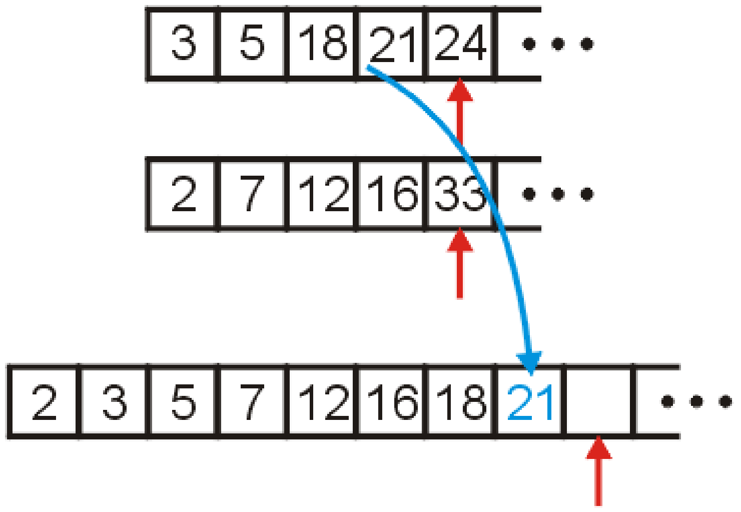
- Copy 18 down
- Increment...



Merging Example

We compare 21 and 33

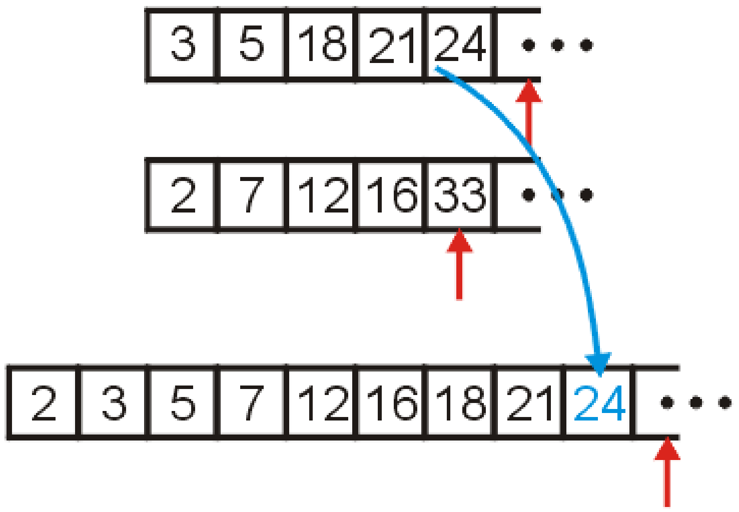
- Copy 21 down
- Increment...

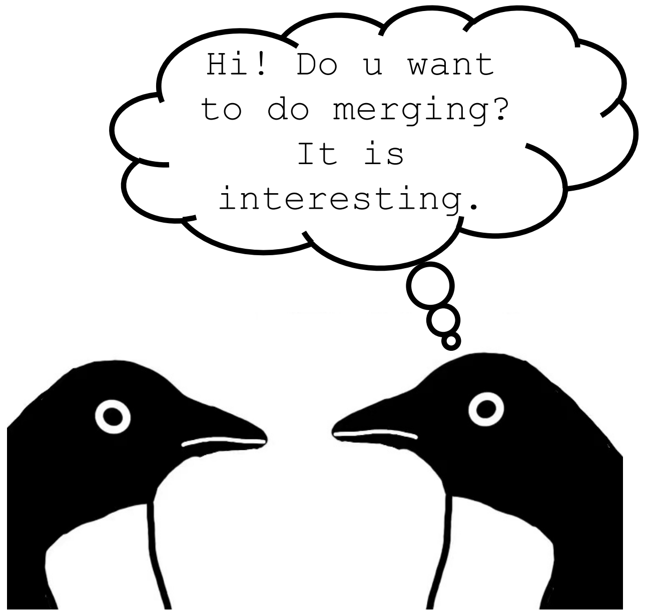


Merging Example

We compare 24 and 33

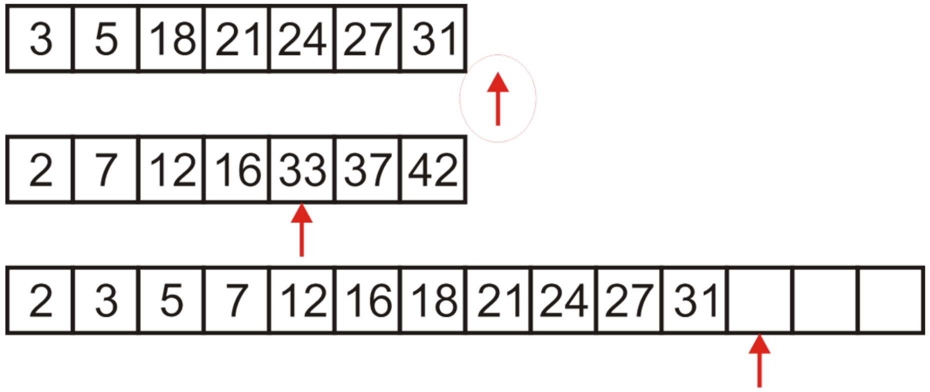
- Copy 24 down
- Increment...



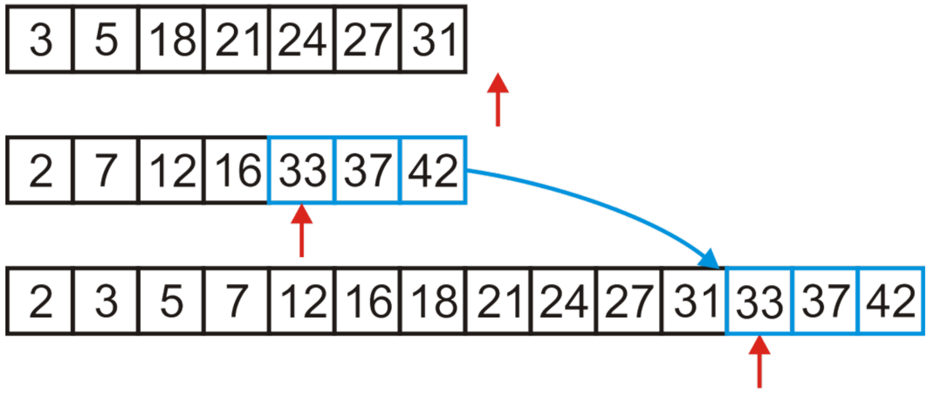


Merging Example

We would continue until we have passed beyond the limit of one of the two arrays



After this, we simply copy over all remaining entries in the non-empty array



Merging Two Lists

Programming a merge is straight-forward:

- the sorted arrays, `array1` and `array2`, are of size `n1` and `n2`, respectively, and
- we have an empty array, `arrayout`, of size `n1 + n2`

Define three variables

```
    int i1 = 0, i2 = 0, k = 0;
```

which index into these three arrays

Merging Two Lists

We can then run the following loop:

```
#include <cassert>
//...
int i1 = 0, i2 = 0, k = 0;

while ( i1 < n1 && i2 < n2 ) {
    if ( array1[i1] < array2[i2] ) {
        arrayout[k] = array1[i1];
        ++i1;
    } else {
        assert( array1[i1] >= array2[i2] );
        arrayout[k] = array2[i2];
        ++i2;
    }
    ++k;
}
```

Merging Two Lists

We’re not finished yet, We have to empty out the remaining array

```
for ( ; i1 < n1; ++i1, ++k ) {
    arrayout[k] = array1[i1];
}

for ( ; i2 < n2; ++i2, ++k ) {
    arrayout[k] = array2[i2];
}
```

Analysis of merging

Time: we have to copy $n_1 + n_2$ elements

- Hence, merging may be performed in $\Theta(n_1 + n_2)$ time
- If the arrays are approximately the same size, $n = n_1 \approx n_2$, we can say that the run time is $\Theta(n)$

Space: we cannot merge two arrays in-place

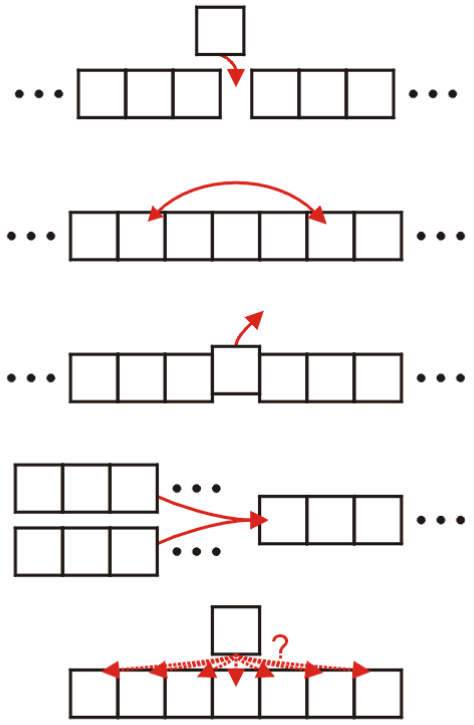
- This algorithm always required the allocation of a new array
- Therefore, the memory requirements are also $\Theta(n)$

The Algorithm

Recall the five sorting techniques:

- Insertion
- Exchange
- Selection
- Merging
- Distribution

Clearly merge sort falls into the fourth category



The Algorithm

The merge sort algorithm is defined recursively:

- If the list is of size 1, it is sorted—we are done;
- Otherwise:
  - Divide an unsorted list into two sub-lists,
  - Sort each sub-list recursively using merge sort, and
  - Merge the two sorted sub-lists into a single sorted list

In practice:

- If the list size is less than a threshold, use an algorithm like insertion sort
- Otherwise:
  - Divide…

Implementation

Suppose we already have a function

```
template <typename Type>
void merge( Type *array, int a, int b, int c );
```

that assumes that the entries

`array[a]` through `array[b - 1]`, and

`array[b]` through `array[c - 1]`

are sorted and merges these two sub-arrays into a single sorted array from index a through index `c - 1`, inclusive

Implementation

For example, given the array,

9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	23	48	73	89	95	17	32	37	57	94	99	28	15	55	7	51	88	97	62

a call to

```
void merge( array, 14, 20, 26 );
```

merges the two sub-lists

9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	23	48	73	89	95	17	32	37	57	94	99	28	15	55	7	51	88	97	62

forming

9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	17	23	32	37	48	57	73	89	94	95	99	28	15	55	7	51	88	97	62

Implementation

We implement a function

```
   template <typename Type>
   void merge_sort( Type *array, int first, int last );
```

that will sort the entries in the positions `first <= i` and `i < last`

- If the number of entries is less than N, call insertion sort
- Otherwise:
  - Find the mid-point,
  - Call merge sort recursively on each of the halves, and
  - Merge the results

Implementation

```
template <typename Type>
void merge_sort( Type *array, int first, int last ) {
    if ( last - first <= N ) {
        insertion_sort( array, first, last );
    } else {
        int midpoint = (first + last)/2;

        merge_sort( array, first, midpoint );
        merge_sort( array, midpoint, last );
        merge( array, first, midpoint, last );
    }
}
```

Implementation

Like merge sort, insertion sort will sort a sub-range of the array:

```
template <typename Type>
void insertion_sort( Type *array, int first, int last ) {
    for ( int k = first + 1; k < last; ++k ) {
        Type tmp = array[k];

        for ( int j = k; k > first; --j ) {
            if ( array[j - 1] > tmp ) {
                array[j] = array[j - 1];
            } else {
                array[j] = tmp;
                goto finished;
            }
        }

        array[first] = tmp;
        finished: ;
    }
}
```

Example

Consider the following is of unsorted array of 25 entries

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We will call insertion sort if the list being sorted of size $N = 6$ or less

Example

We call `merge_sort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
merge_sort( array,  0, 25 )
```

Example

We are calling `merge_sort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

First, $25 - 0 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
```

```
merge_sort( array,  0, 25 )
```

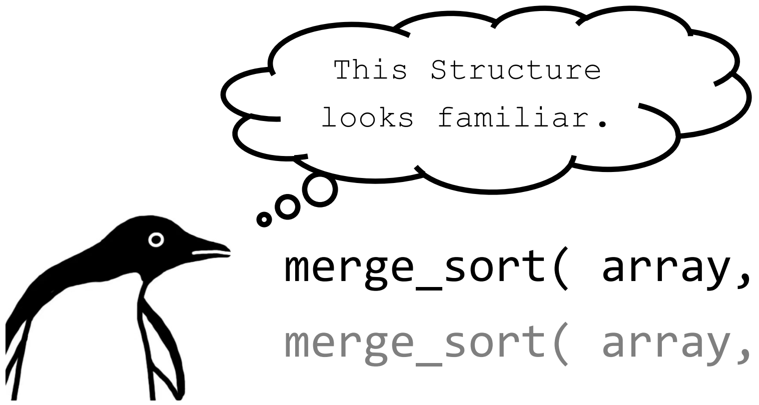
Example

We are now executing `merge_sort( array, 0, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

First, $12 - 0 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
```



```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 0, 6 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Now, $6 - 0 \leq 6$, so find we call insertion sort

```
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from `0` to `5`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
insertion_sort( array, 0, 6 )
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from `0` to `5`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

– This function call completes and so we exit

```
insertion_sort( array, 0, 6 )
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 0, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
merge_sort( array, 6, 12 );
```

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 6,  12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Now, $12 - 6 \le 6$, so find we call insertion sort

```
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 6 to 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
insertion_sort( array, 6, 12 )
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 6 to 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

– This function call completes and so we exit

```
insertion_sort( array, 6, 12 )
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

```
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 0, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
merge_sort( array, 6, 12 );
merge( array, 0, 6, 12 );
```

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 0, 6, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

These two sub-arrays are merged together

```
merge( array, 0, 6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 0, 6, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

These two sub-arrays are merged together
- This function call exists

```
merge( array, 0, 6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 0, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

We are finished calling this function as well

```
    midpoint = (0 + 12)/2; // == 6
    merge_sort( array, 0, 6 );
    merge_sort( array, 6, 12 );
    merge( array, 0, 6, 12 );
```

Consequently, we exit

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
merge_sort( array, 12, 25 );
```

```
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 0, 12, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	7	15	17	28	32	51	55	57	62	88	94	97	99

These two sub-arrays are merged together

```
merge( array, 0, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 0, 12, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

These two sub-arrays are merged together

– This function call exists

```
merge( array, 0, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

We are finished calling this function as well

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
merge_sort( array, 12, 25 );
merge( array, 0, 12, 25 );
```

Consequently, we exit

`merge_sort( array,  0, 25 )`

Run-time Analysis of Merge Sort

The time required to sort an array of size $n > 1$ is:

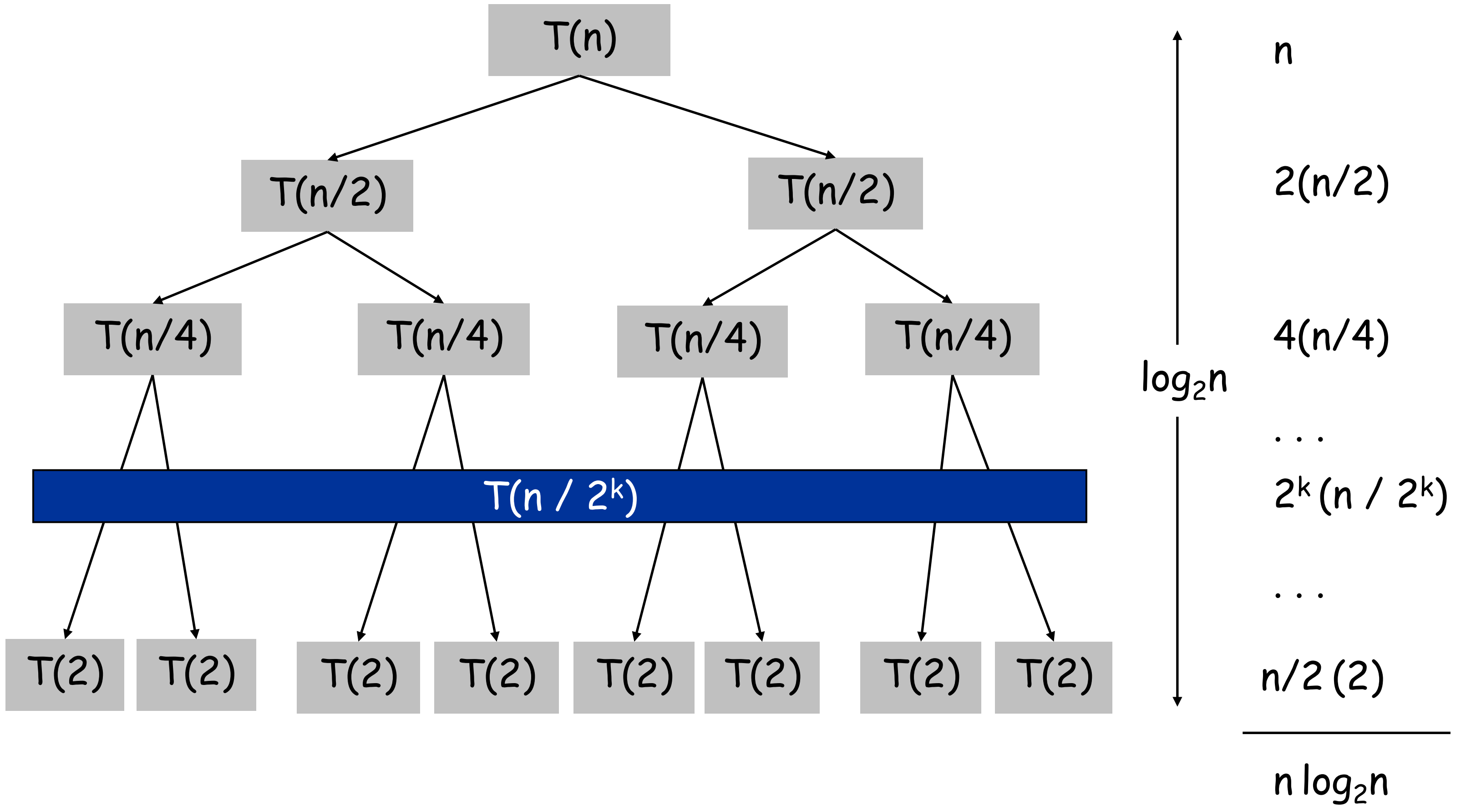
- the time required to sort the first half,
- the time required to sort the second half, and
- the time required to merge the two lists

That is: $$\mathrm{T}(n) = \begin{cases} \mathbf{\Theta}(1) & n = 1 \\ 2\,\mathrm{T}\left(\frac{n}{2}\right) + \mathbf{\Theta}(n) & n > 1 \end{cases}$$

Solution: $\mathrm{T}(n) = \Theta(n \ln(n))$

Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Run-time Summary

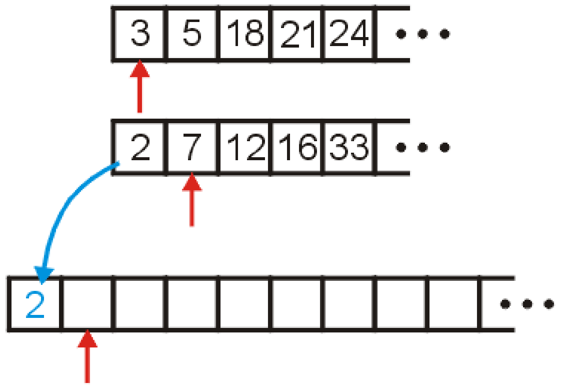
The following table summarizes the run-times of merge sort

Case	**Run Time**	**Comments**
Worst	$\Theta(n \ln(n))$	No worst case
Average	$\Theta(n \ln(n))$	
Best	$\Theta(n \ln(n))$	No best case

Why is it not $O(n^2)$

When we are merging, we are comparing values

- What operation prevents us from performing $O(n^2)$ comparisons?
- During the merging process, if 2 came from the second half, it was only compared to 3 and it was not compared to any other of the other $n - 1$ entries in the first array



- In this case, we remove n inversions with one comparison

Merge Sort

The (likely) first proposal of merge sort was by John von Neumann in 1945

- The creator of the *von Neumann architecture* used by all modern computers:

http://en.wikipedia.org/wiki/Von_Neumann

Divide and Conquer

- Divide-and-conquer.
  - Divide up problem into several subproblems (of the same kind).
  - Solve (conquer) each subproblem recursively.
  - Combine solutions to subproblems into overall solution.
- Most common usage.
  - Divide problem of size n into two subproblems of size $n/2$.
  - Solve (conquer) two subproblems recursively.
  - Combine two solutions into overall solution.
- Consequence.
  - Brute force: $\Theta(n^2)$.
  - Divide-and-conquer: $O(n \log n)$.

Divide and Conquer

- Two typical divide and conquer algorithm we have already known:
  - Merge Sort
  - Binary Search

Binary Search

```
int bfind(int key, int a[], int left, int right)
{   if (left+1 == right) return -1;
    int m = (left + right) / 2;
    if (key == a[m]) return m;
    if (key < a[m]) return bfind(key, a, left, m);
    else return bfind(key, a, m, right);
}
```

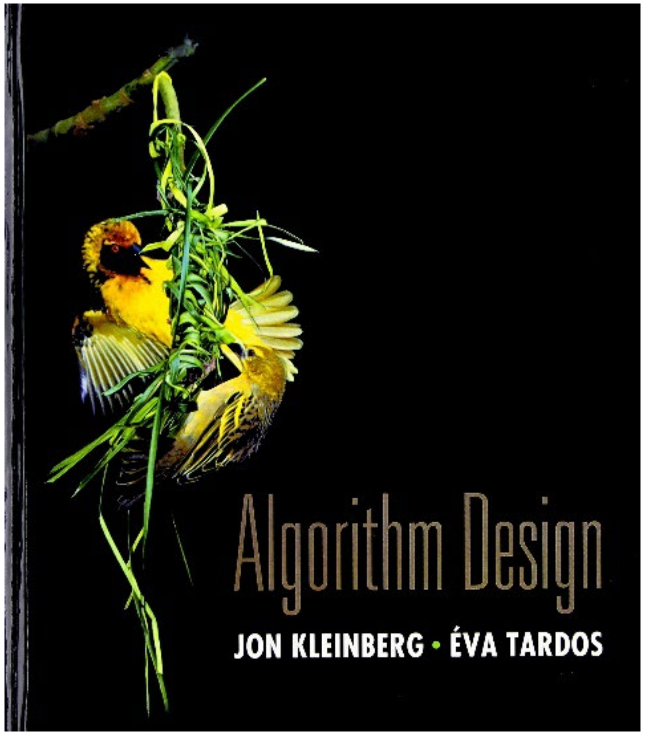
Examples

- Binary Search
  - Key problem can be divide up problem into several sub-problems.
  - Sub-problems are with same type and independent from each other.
  - It is ***not*** necessary to merge Sub-problems to get the key problem solved.

- Merge Sort
  - Key problem can be divide up problem into several sub-problems.
  - Sub-problems are with same type and independent from each other.
  - Sub-problems need to be merged to get the key problem solved.

Count Inversions in an array

- *Inversion Count* for an array indicates – how far (or close) the array is from being sorted. If array is already sorted, then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.



- Section 5.3

METHOD 1 (Simple)

- **Approach :**Traverse through the array and for every index find the number of smaller elements on its right side of the array. This can be done using a nested loop. Sum up the counts for all index in the array and print the sum.
- **Algorithm :**

  - Traverse through the array from start to end

  - For every element find the count of elements smaller than the current - number upto that index using another loop.

  - Sum up the count of inversion for every index.

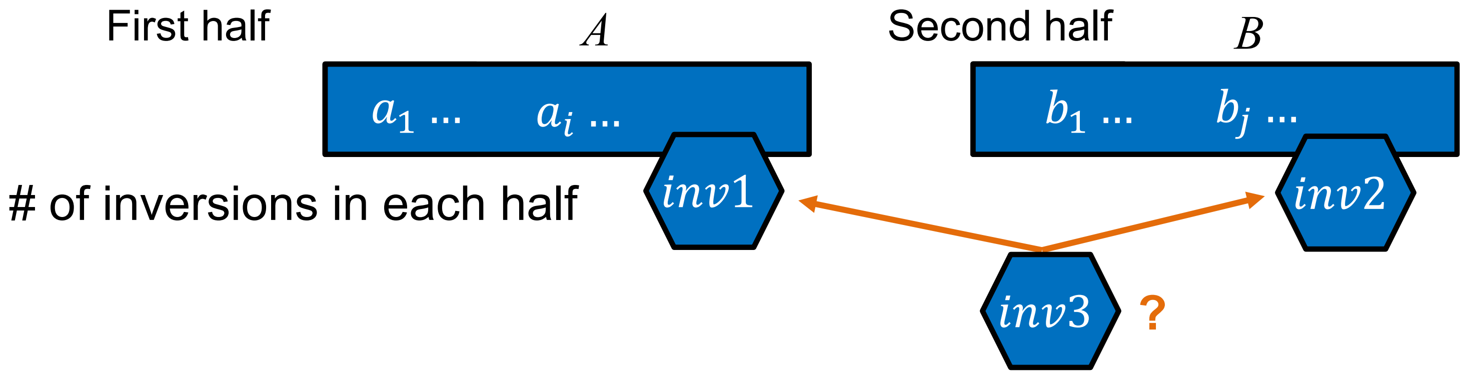
  Print the count of inversions.

METHOD 1 (Simple)

- **Complexity Analysis: Time Complexity:** $O(n^2)$, Two nested loops are needed to traverse the array from start to end so the Time complexity is $O(n^2)$.
- **Space Compelxity:** $O(1)$, No extra space is required.

METHOD 2 (Enhance Merge Sort)

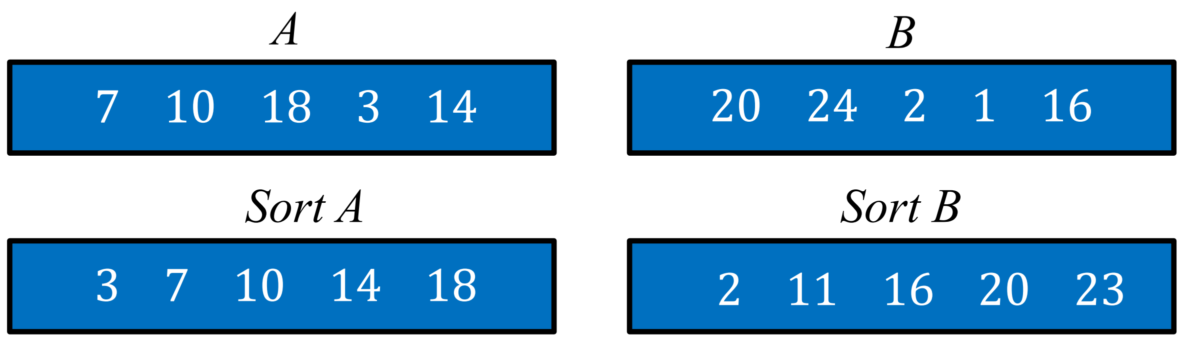
- **Approach:** Suppose the number of inversions in the left half and right half of the array (let be $inv1$ and $inv2$), what kinds of inversions are not accounted for in $inv1 + inv2$?
- The answer is : the inversions that need to be counted during the merge step. Therefore, to get a number of inversions, that needs to be added a number of inversions in the left subarray, right subarray and $merge()$.



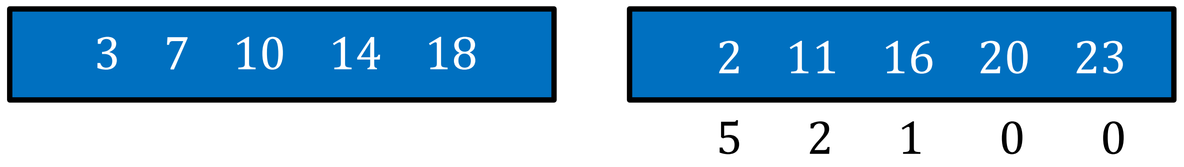
How to count inversions (a, b) with $a \in A$ and $b \in B$?

METHOD 2(Enhance Merge Sort)

- Q. How to count inversions (a, b) with $a \in A$ and $b \in B$?
- A. Easy if A and B are sorted!
- Warmup algorithm.
  - Sort A and B.
  - For each element $b \in B$,
    - binary search in A to find how many elements in A are greater than b.



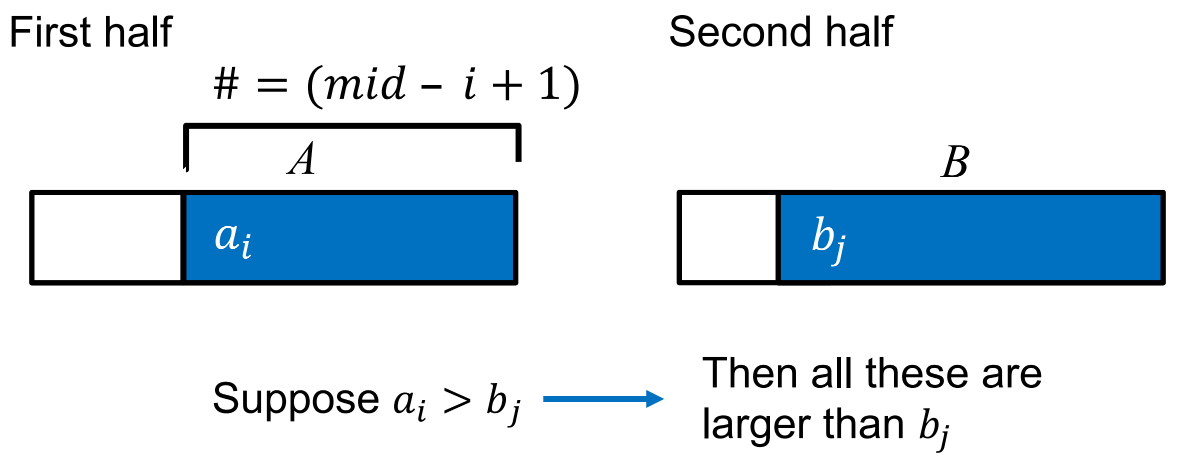
Binary search to count inversions (a, b) with $a \in A$ and $b \in B$



METHOD 2(Enhance Merge Sort)

- **How to get number of inversions in merge()?**

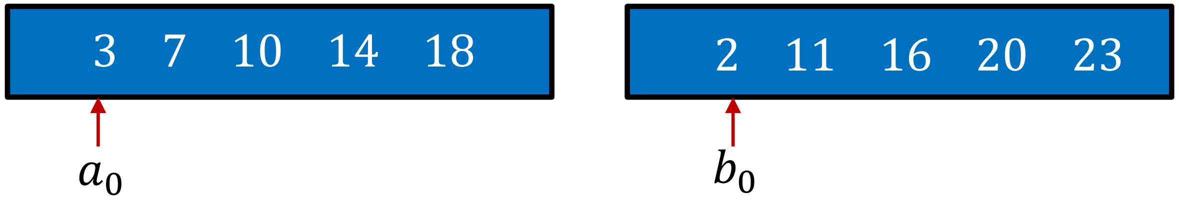
In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in $merge()$, if $a[i]$ is greater than $b[j]$, then there are $(mid - i + 1)$ inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray $(a[i+1], a[i+2]$ … $a[mid])$ will be greater than $b[j]$



METHOD 2(Enhance Merge Sort)

- **The complete picture:**

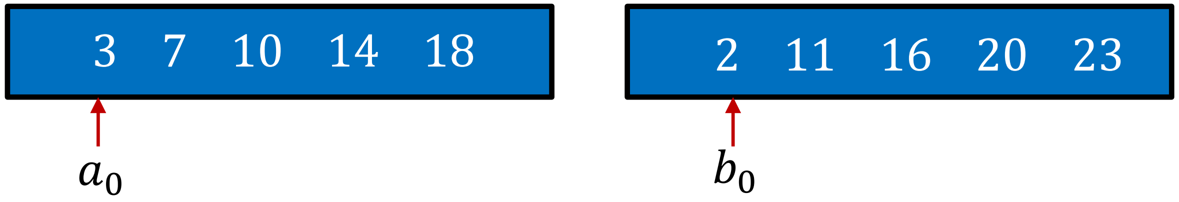
  Count inversions (a, b) with $a \in A$ and $b \in B$



METHOD 2(Enhance Merge Sort)

- **The complete picture:**

Count inversions (a, b) with $a_i \in A$ and $b_j \in B$



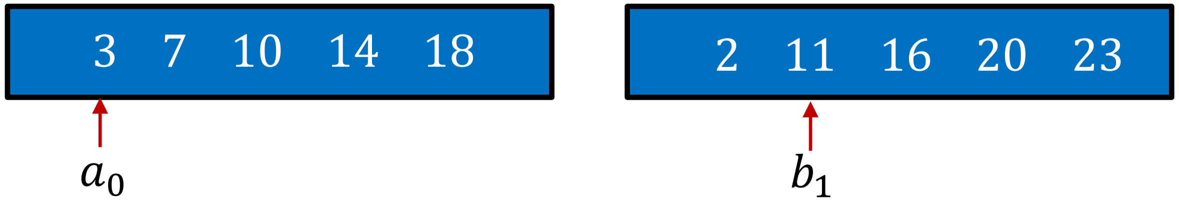
$a_0 > b_0$

```
Inv3 = Inv3+ mid-i+1 = 0+ 4-0+1 = 5
j++;
```

METHOD 2(Enhance Merge Sort)

- **The complete picture:**

Count inversions (a, b) with $a \in A$ and $b \in B$

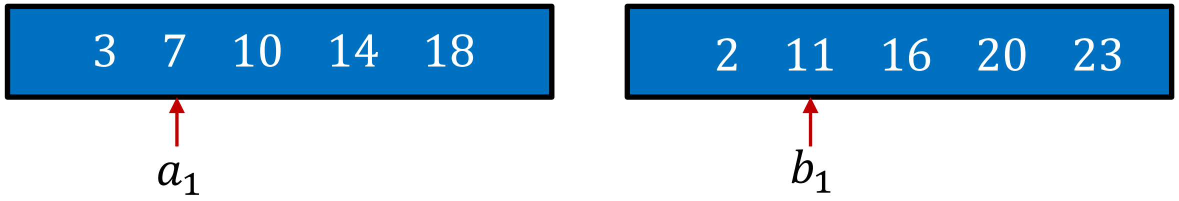


$a_0 < b_1$ `i++`

METHOD 2(Enhance Merge Sort)

- **The complete picture:**

Count inversions (a, b) with $a \in A$ and $b \in B$

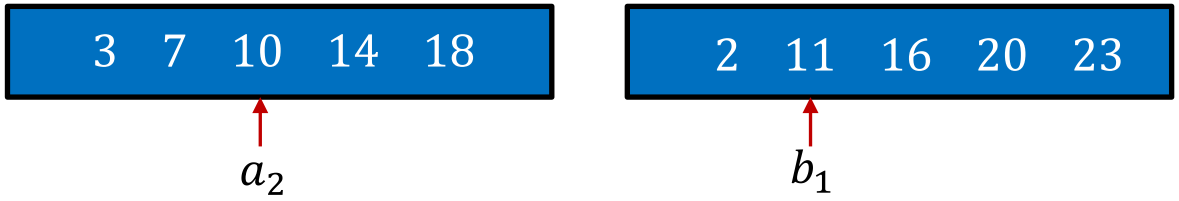


$a_1 < b_1$ `i++`

METHOD 2(Enhance Merge Sort)

- **The complete picture:**

Count inversions (a, b) with $a \in A$ and $b \in B$

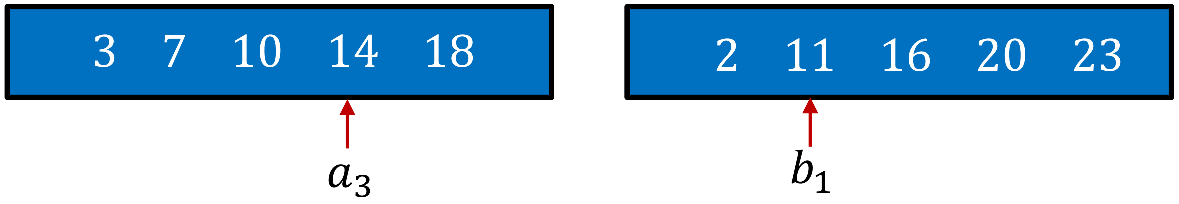


$a_2 < b_1$ `i++`

METHOD 2(Enhance Merge Sort)

- **The complete picture:**

Count inversions (a, b) with $a \in A$ and $b \in B$



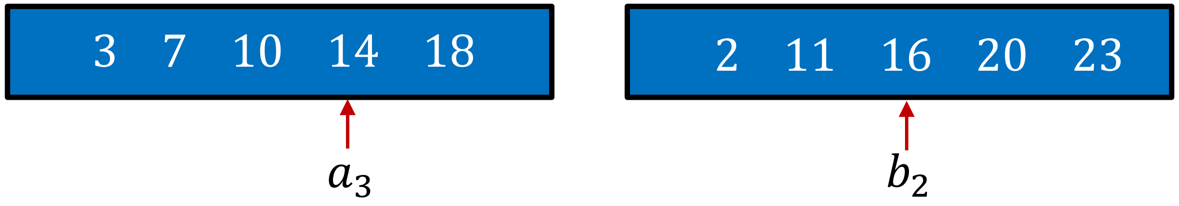
$a_3 > b_1$

```
Inv3 = Inv3+ mid-i+1 = 5+ 4-3+1 = 7
j++;
```

METHOD 2(Enhance Merge Sort)

- **The complete picture:**

Count inversions (a, b) with $a \in A$ and $b \in B$

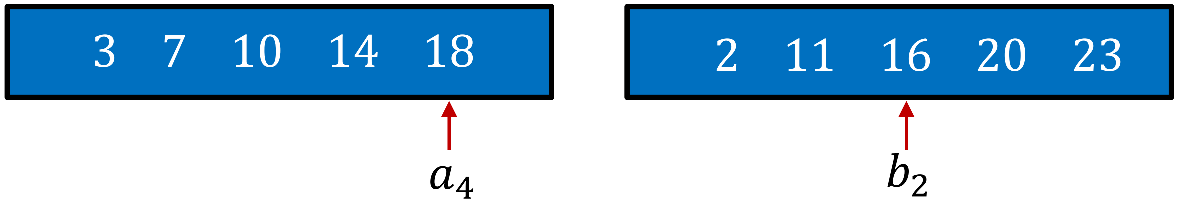


$a_3 < b_2$ `i++`

METHOD 2(Enhance Merge Sort)

- **The complete picture:**

Count inversions (a, b) with $a \in A$ and $b \in B$



$a_4 > b_2$

```
Inv3 = Inv3+ mid-i+1 = 7 + 4 – 4 + 1 = 8

j++;
```

METHOD 2(Enhance Merge Sort)

- **Algorithm:**

  - The idea is similar to merge sort, divide the array into two equal or almost equal halves in each step until the base case is reached.

  - Create a function merge that counts the number of inversions when two halves of the array are merged, create two indices i and j, i is the index for first half and j is an index of the second half. if $a[i]$ is greater than $b[j]$, then there are $(mid - i + 1)$ inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray $(a[i + 1], a[i + 2]$ ... $a[mid])$ will be greater than $b[j]$.

  - Create a recursive function to divide the array into halves and find the answer by summing the number of inversions is the first half, number of inversion in the second half and the number of inversions by merging the two.

  - The base case of recursion is when there is only one element in the given half.

Summary

This topic covered merge sort:

- Divide an unsorted list into two equal or nearly equal sub lists,
- Sorts each of the sub lists by calling itself recursively, and then
- Merges the two sub lists together to form a sorted list