Sorting Algorithms

Sorting

- It’s a way to arrange the unordered collection in some order like ascending or descending

Types of sorting algorithms –

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort
7. Radix Sort
8. Shell Sort etc.

Internal and External Sorting

➢**Internal sort** : Sorting method in which sorting process that takes place entirely within the **main memory** of a computer.

•This is possible whenever the data to be sorted is small enough to all be held in the main memory.

•**Examples:** bubble sort, selection sort etc.

➢**External sorting** is a term for a class of sorting algorithms that can handle massive amounts of data.

•External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive).

External merge sort

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together.

For example, for sorting 900 megabytes of data using only 100 megabytes of RAM

Stable Sorting and in-place Sorting

- **Stable Sorting**: A stable sort is one which preserves the original order of the identical elements in the input set.
- **Example:** Suppose two elements x and y has keys with value 1 for each and suppose the order of arrangement of x and y in the input is x, y. Since x and y have equal keys, i.e. 1, the order of arrangement in the output would be x, y as well.

- **In-place Sorting:** Sorting algorithm is said to be *in-place* if it requires very little additional space besides the initial array holding the elements that are to be sorted.

Bubble Sort

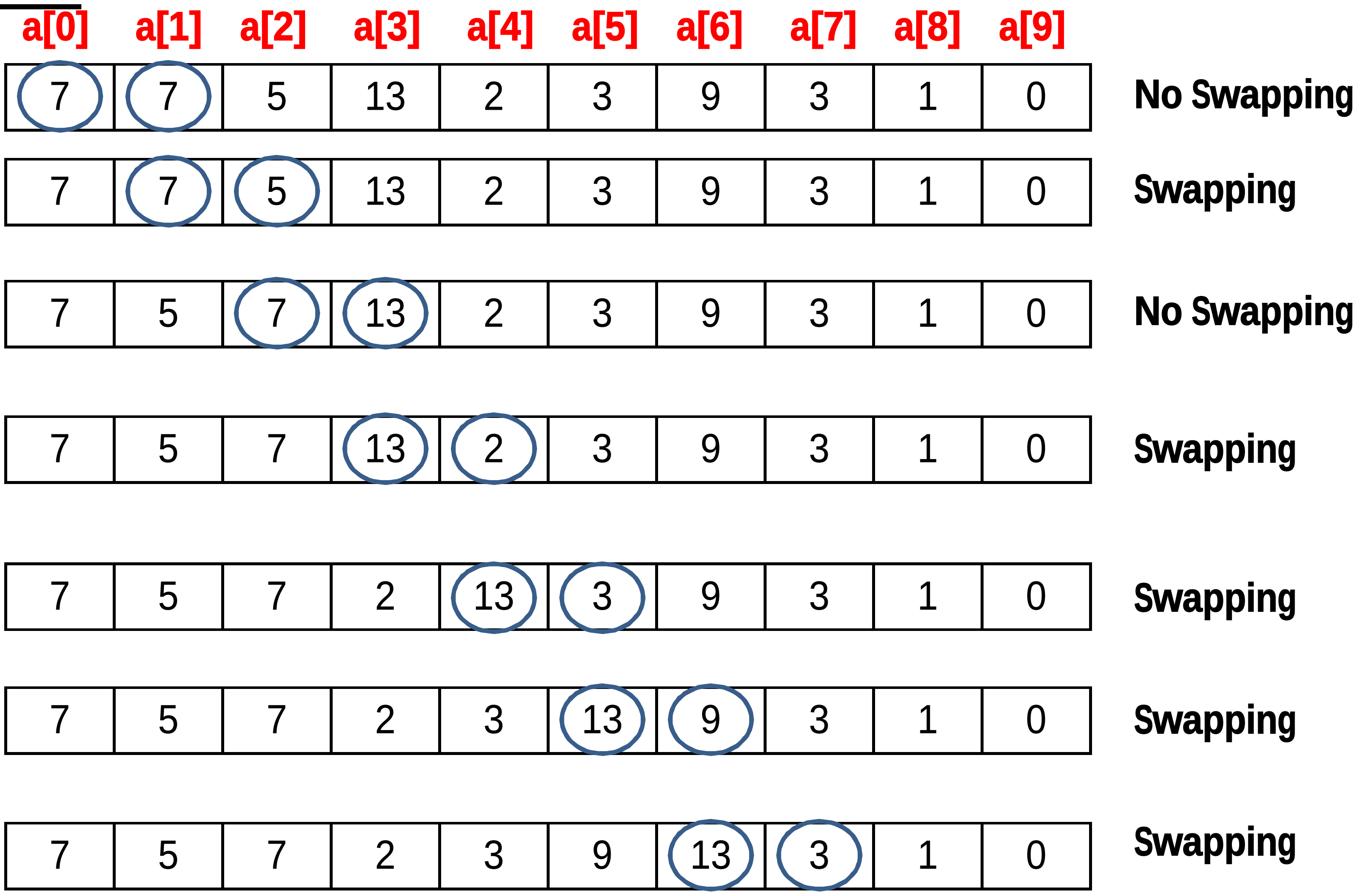
➢Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Basic Steps:

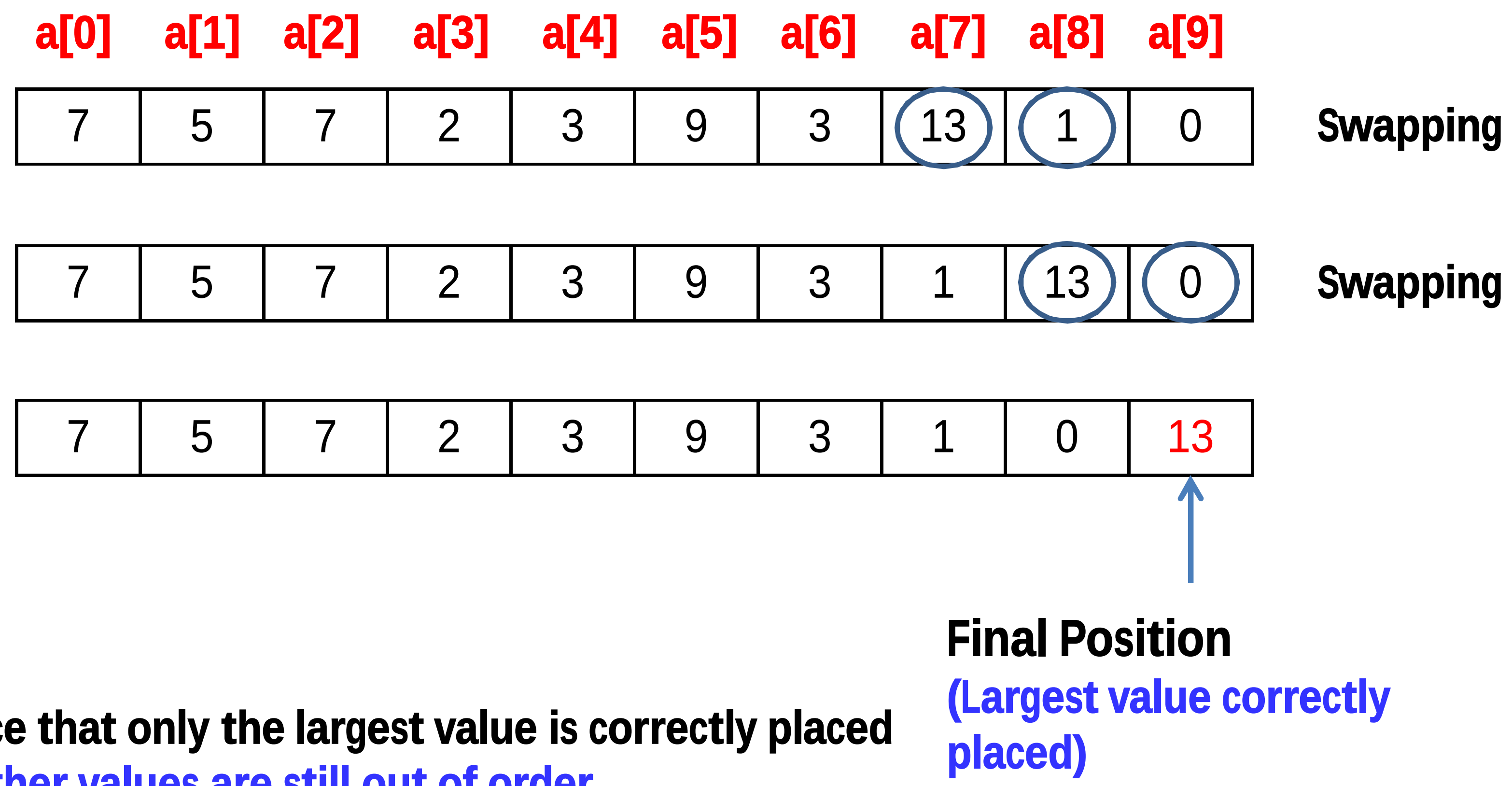
1. Move from left to right end
2. Compare the two adjacent elements and swap them if needed

Bubble Sort Illustration

Iteration 1:



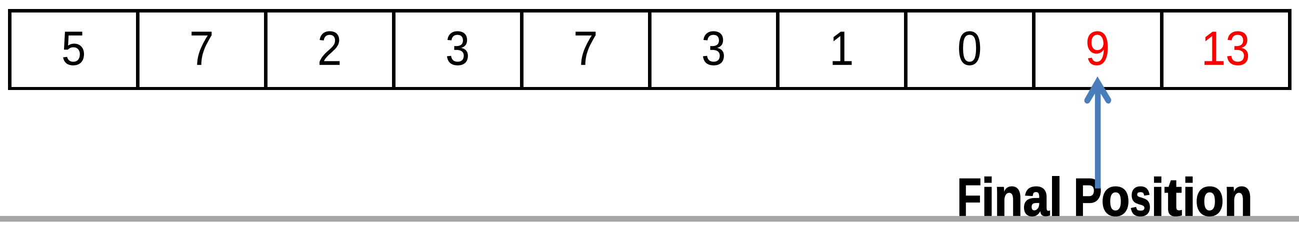
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	
7	7	5	13	2	3	9	3	1	0	**No Swapping**
7	7	5	13	2	3	9	3	1	0	**Swapping**
7	5	7	13	2	3	9	3	1	0	**No Swapping**
7	5	7	13	2	3	9	3	1	0	**Swapping**
7	5	7	2	13	3	9	3	1	0	**Swapping**
7	5	7	2	3	13	9	3	1	0	**Swapping**
7	5	7	2	3	9	13	3	1	0	**Swapping**



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	
7	5	7	2	3	9	3	13	1	0	Swapping
7	5	7	2	3	9	3	1	13	0	Swapping
7	5	7	2	3	9	3	1	0	13	

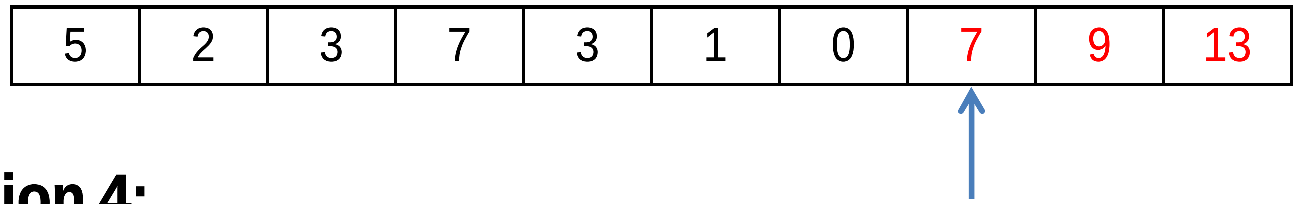
- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process

<u>Iteration 2:</u>

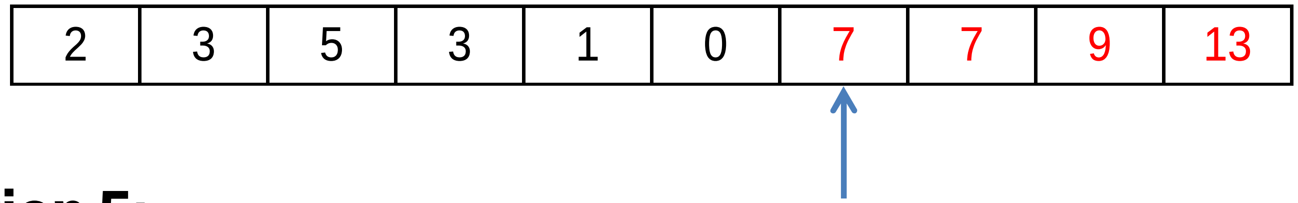


5	7	2	3	7	3	1	0	9	13

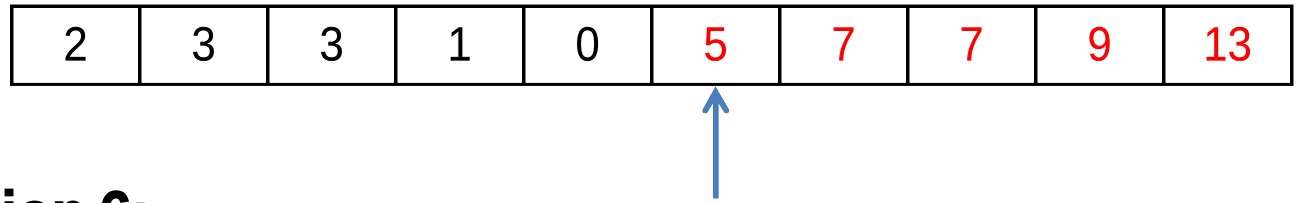
Iteration 3:



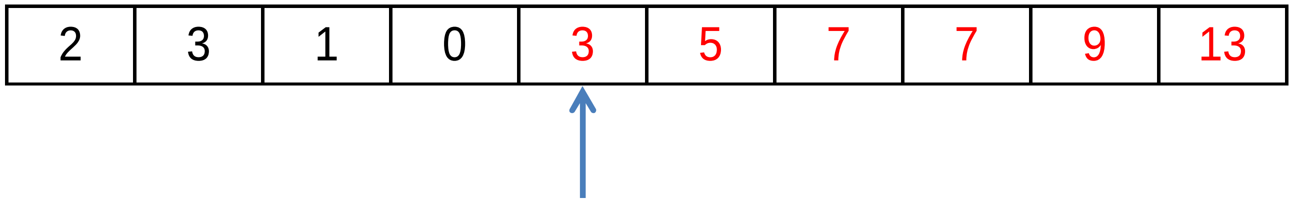
Iteration 4:



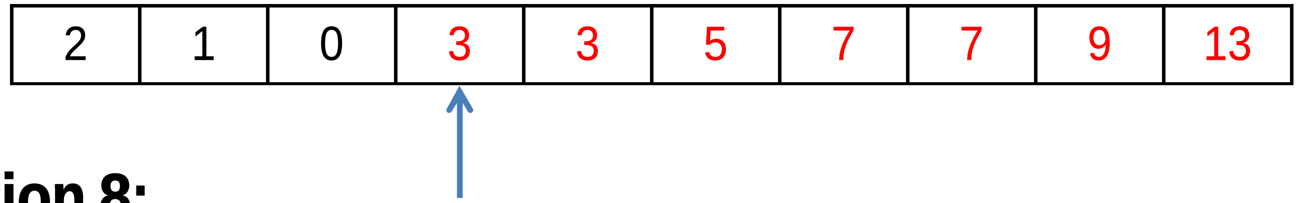
Iteration 5:



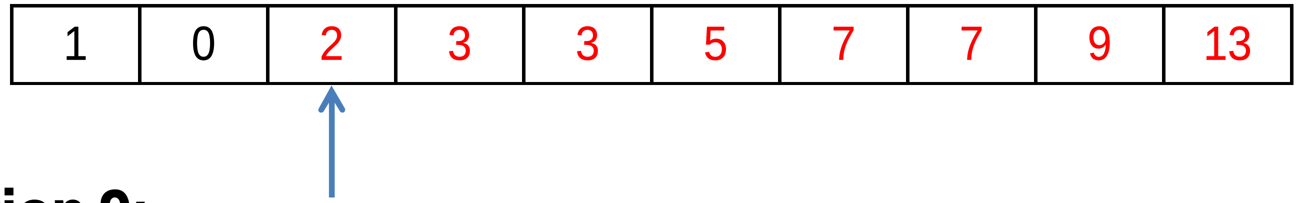
Iteration 6:



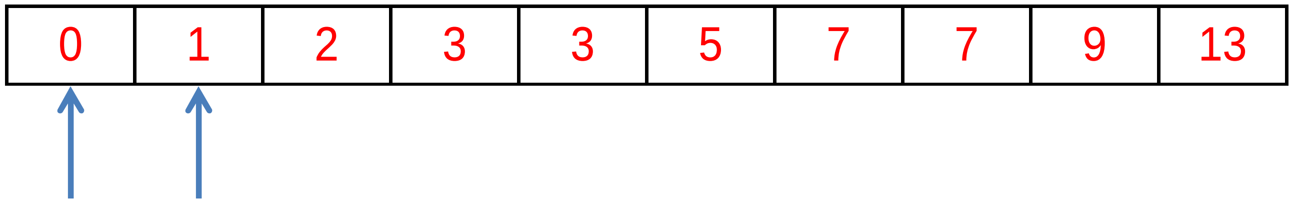
Iteration 7:



Iteration 8:



Iteration 9:



Bubble sort

Algorithm Bubble_sort(a,n): This algorithm sort the elements in ascending order. a is linear array which contains n elements. Variable temp is used to facilitate the swapping of two values. I and J are used as loop control variables.

```
1.  For I = 1 to n-1                                C1
2.          For J = 0 to (n-I-1)                    C2
3.                  If a[J] > a[J+1] then,          C3
4.                          Set temp = a[J]         C4
5.                          Set a[J] = a[J+1]       C5
6.                          Set a[J+1] = temp       C6
```

Analysis of Bubble Sort (version 1)

Worst Case analysis:

For Step 2 :

I =1 J= 0 to (n-2) i.e. total (n-1) and 1 for false. Hence n times

I =2 J =0 to (n-3) i.e. total (n-2) and 1 for false. Hence n-1 times

…….

I=n-1 J=0 to (n-1-n+1) i.e. total 1 and 1 for false. Hence 2 times.

Time Complexity = $n(n+1)/2 - 1$

= $O(n^2)$

Analysis of Bubble Sort (version 2)

Worst Case analysis:

From the above illustration, we observe following points –

In (n-1) iterations or passes array will become sorted.

Iteration 1:	no. of comparisons (n-1)
Iteration 2:	no. of comparisons (n-2)
Iteration 3:	no. of comparisons (n-3)
……………….	
Iteration k:	no. of comparisons (n-k)
……………….	
Iteration last:	no. of comparisons 1

Time Complexity = Total Comparisons

= (n-1) + (n-2) + (n-3) + ….+ (n-k) + …. + 3 + 2 + 1

= n(n-1)/2

= $O(n^2)$

What is the best case time complexity of bubble sort?

Optimized Version of Bubble Sort

This algorithm of Bubble Sort always runs $O(n^2)$time even if the array is sorted.

It can be optimized by stopping the algorithm if there is no swapping in particular pass.

Why?

Because if there is no swapping in first pass, this ensures that array is already sorted

```
1. For I = 1 to n-1
2.     Flag=0
3.     For J = 0 to (n-I-1)
4.             If a[J] > a[J+1] then,   // comparison
5.                  Swap(a[j],a[j+1])    // Swap or exchange
6.                  Flag=1
7.     If Flag==0
8.         Break
```

Important facts about Bubble Sort

Worst and Average Case Time Complexity: O(n^2). Worst case occurs when array is reverse sorted.
Best Case Time Complexity: O(n). Best case occurs when array is already sorted.
Auxiliary Space: O(1)
Sorting In Place: Yes
Stable: Yes

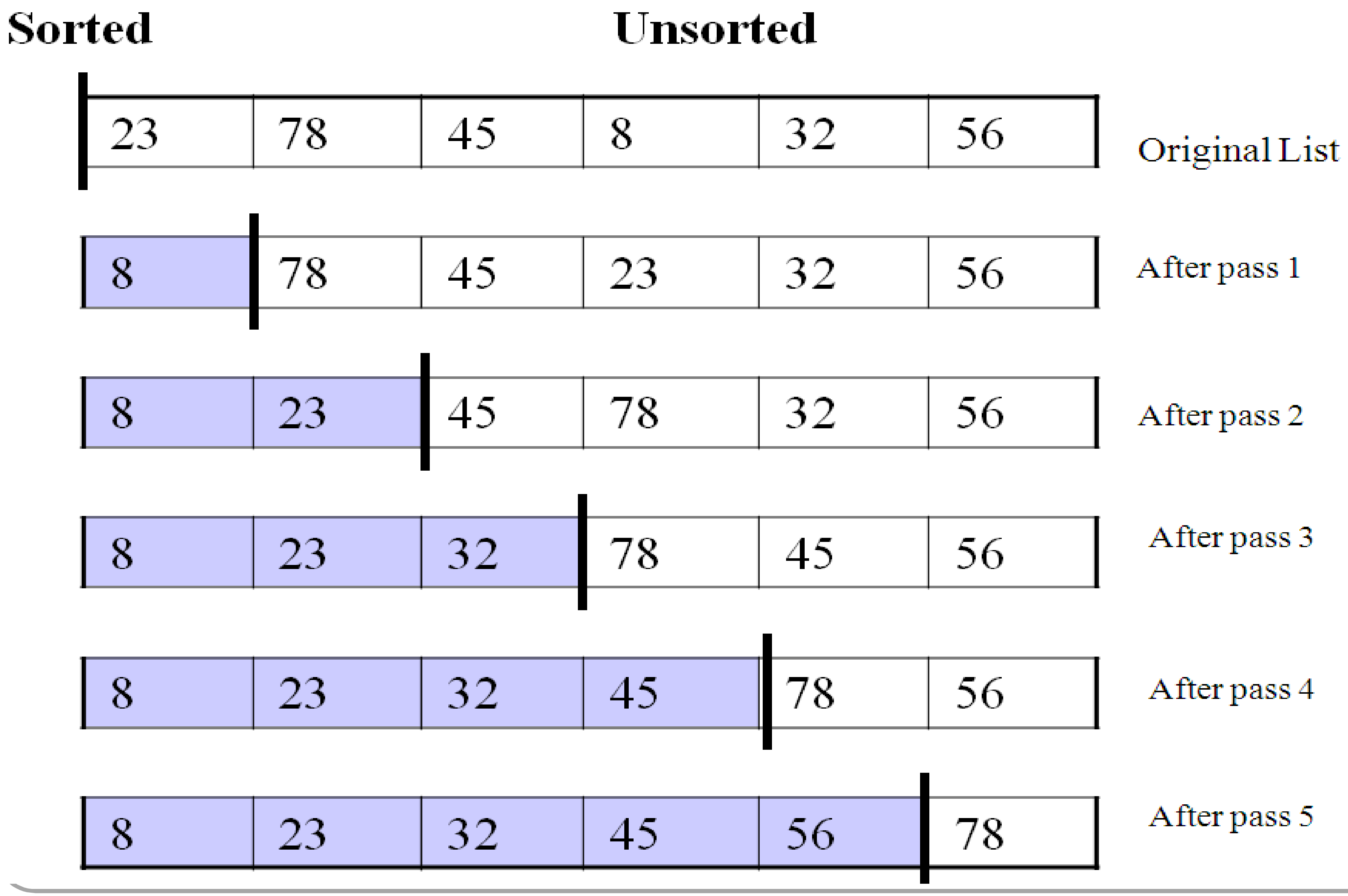
Selection Sort

➢The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.

➢**Basic Steps:**

1. Move from left to right end
2. Each time least element gets its final position i.e. we select least element and put it at it’s final position

Selection Sort



Selection Sort

Algorithm Select_sort(a,n): This algorithm sort the elements in ascending order. a is linear array which contains n elements. Variable temp is used to facilitate the swapping of two values. I and J are used as loop control variables.

```
1. For I = 0 to n-2
2.     min=I
3.     For J = I+1 to (n-1)
4.         If a[min] > a[J] then, //comparison
5.             min=J
6.     Set temp = a[min]
7.     Set a[min] = a[I]
8.     Set a[I] = temp
```

Analysis of Selection Sort

From the above illustration, we observe following points –

In (n-1) iterations or passes array will become sorted.

Iteration 1:	no. of comparisons (n-1)
Iteration 2:	no. of comparisons (n-2)
Iteration 3:	no. of comparisons (n-3)
……………….	
Iteration k:	no. of comparisons (n-k)
……………….	
Iteration last:	no. of comparisons 1

Time Complexity = Total Comparisons

$= (n-1) + (n-2) + (n-3) + \ldots + (n-k) + \ldots + 3 + 2 + 1$

$= n(n-1)/2$

$= O(n^2)$

Important facts about Selection Sort

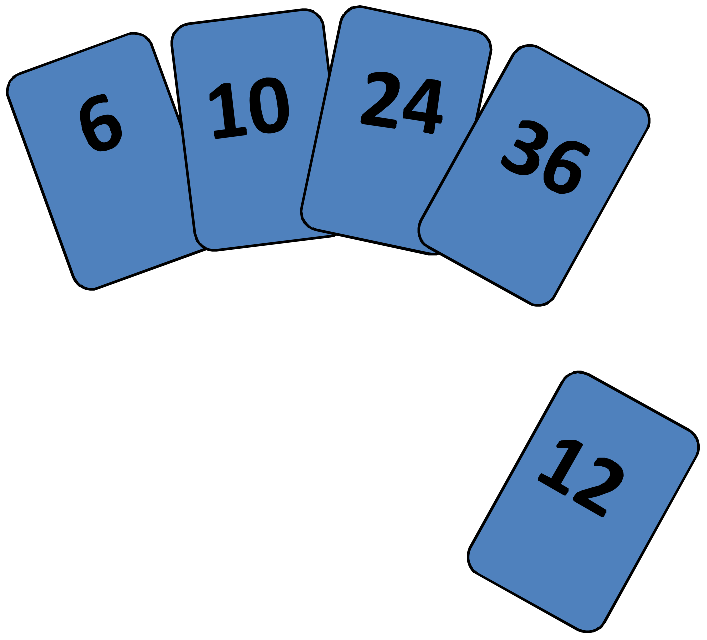
Worst and Average Case Time Complexity: $O(n^2)$.
Best Case Time Complexity: $O(n^2)$.
Auxiliary Space: O(1)
Sorting In Place: Yes
Stable: No

Note : The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Insertion Sort

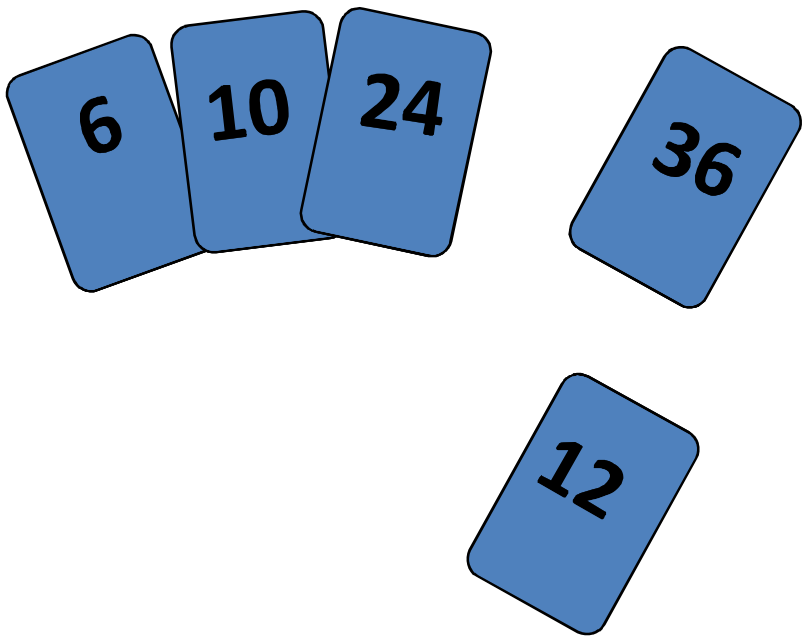
- ➢Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.
- ➢ Pick one element from the unsorted part.
- ➢ compare it with each of the elements in the sorted part, from right to left
- ➢ Create a space by shifting or moving the elements to next position
- ➢ Insert the element at empty space

Insertion Sort

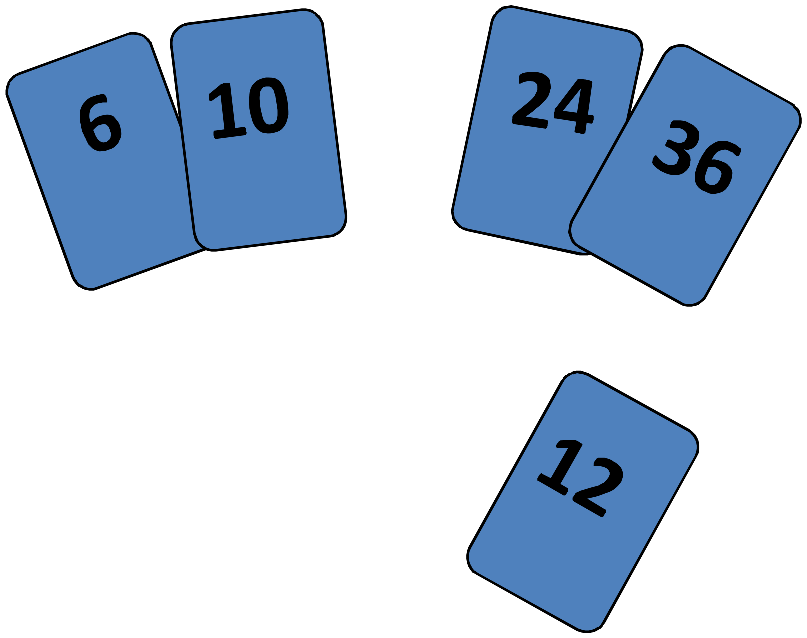


To insert 12, we need to make room for it by moving first 36 and then 24.

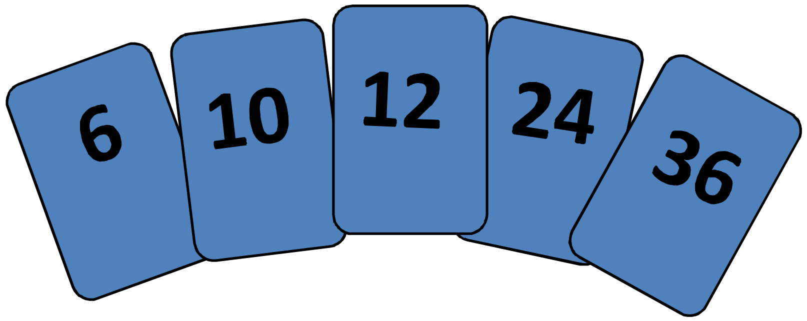
Insertion Sort



Insertion Sort

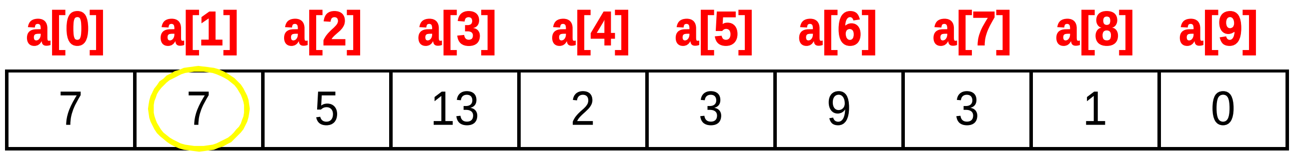


Insertion Sort

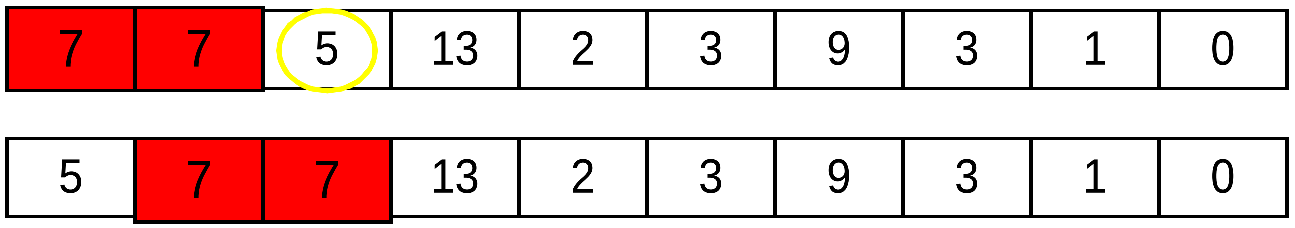


Insertion Sort Illustration

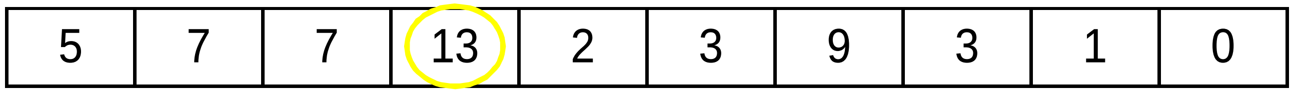
Pass 1:



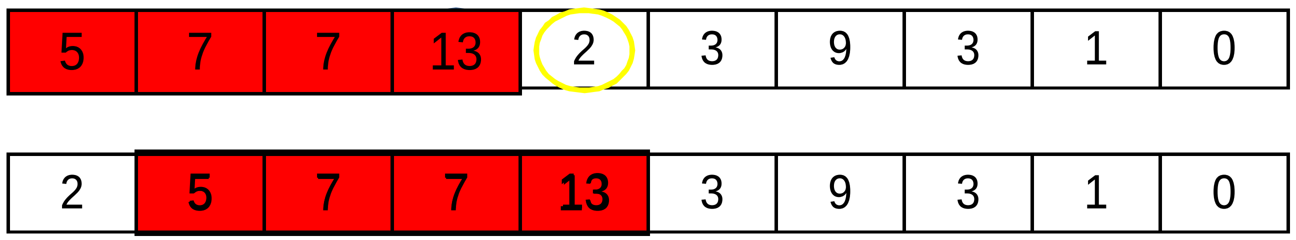
Pass 2:



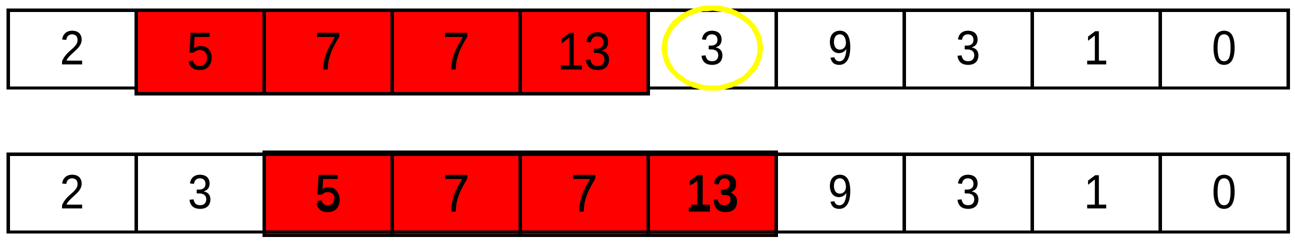
Pass 3:



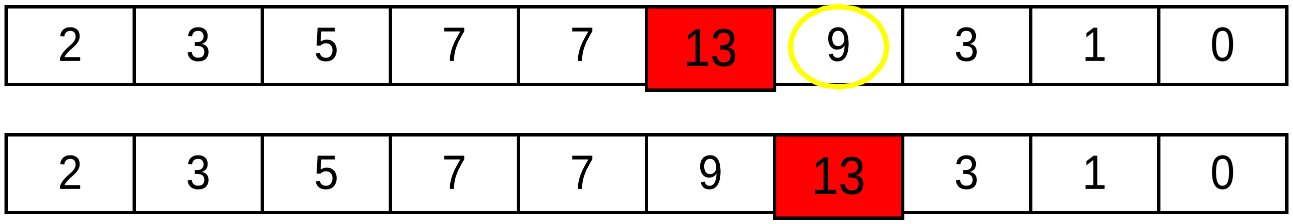
Pass 4:



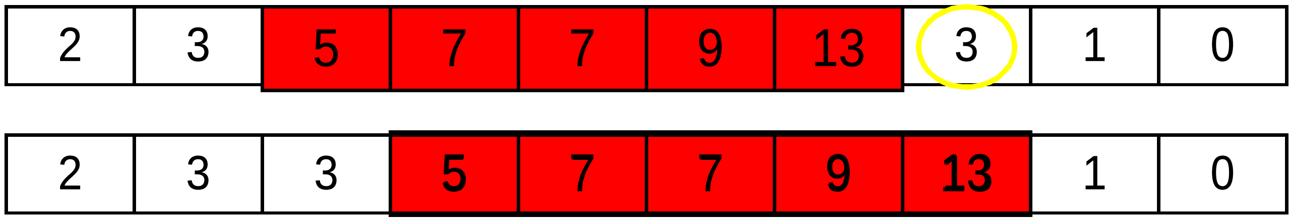
Pass 5:



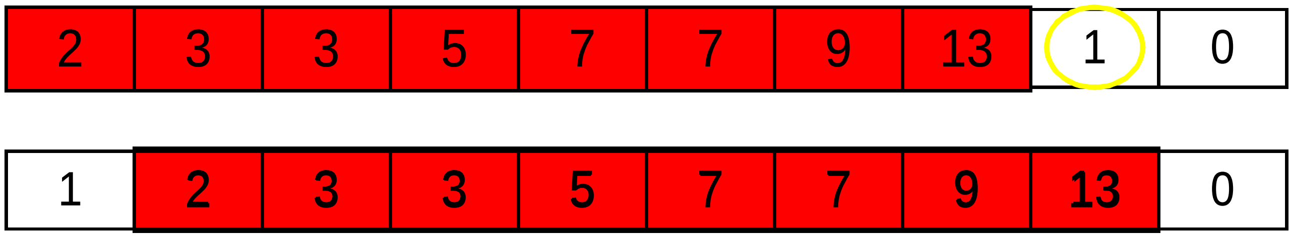
Pass 6:



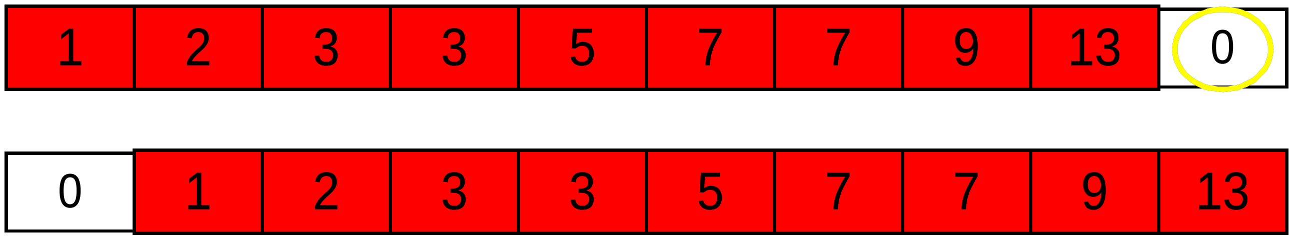
Pass 7:



Pass 8:



Pass 9:



Insertion sort

Algorithm Insert_sort(a,n): This algorithm sort the elements in ascending order. a is linear array which contains n elements. I and J are used as loop control variables.

```
1.  For I = 1 to n-1
2.          Key = a[I]
3.          J = I - 1
4.          While J>=0 and a[J] > Key
5.                  Set a[J+1] = a[J]
6.                  J = J - 1
7.          a[J+1] = Key
```

Analysis of Insertion Sort

From the above illustration, we observe following points –

In (n-1) iterations or passes array will become sorted.

Iteration 1:	no. of comparisons 1 and 1 movement
Iteration 2:	no. of comparisons 2 and 2 movements
Iteration 3:	no. of comparisons 3 and 3 movements
…………….	
Iteration last:	no. of comparisons n-1 and N-1 movements.

Time Complexity = Total Comparisons

$$= 1 + 2 + 3 + \ldots\ldots + (n-2) + (n-1)$$

$$= n(n-1)/2$$

$$= O(n^2)$$

What will be the Worst Case for insertion sort?

The worst-case will be when Array consists of distinct items in reverse sorted order:
i.e. $a[0] > a[1] > \ldots > a[n-1]$.

Analysis of Insertion Sort

Best Case :

Let Array is already Sorted.

Iteration 1:	no. of comparisons 1
Iteration 2:	no. of comparisons 1
Iteration 3:	no. of comparisons 1
…………….	
Iteration last:	no. of comparisons 1

Time Complexity = Total Comparisons
= 1 + 1+1+1+…….+1 upto n)
= n
= O(n)

Analysis of Insertion Sort

Time Complexity: O(n^2)
Auxiliary Space: O(1)
Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (O(n)) when elements are already sorted.
Sorting In Place: Yes
Stable: Yes

Solve Recurrence Relation

- **T(n) = T(n-1) + *O(n)* // [the O(n) is for Combine]**
- **T(1) = O(1) // Base Case**

$O(n^2)$

Quick Sort

- This algorithm follows divide and conquer strategy
- One of the fastest sorting algorithm
- Divide the list of numbers into sub-list until no more division possible.
- It picks an element as pivot and partitions the given array around the picked pivot.
- There are many different versions of quickSort that pick pivot in different ways:

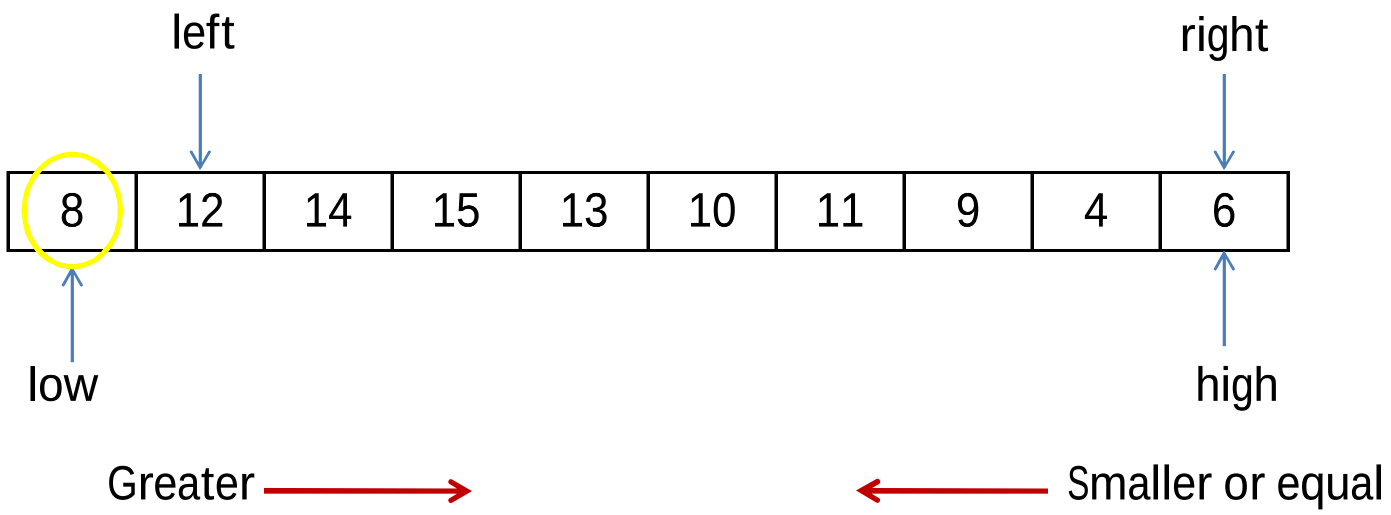
1. **Always pick first element as pivot.**
2. Always pick last element as pivot
3. Pick a random element as pivot.
4. Pick median as pivot.

Quick Sort

- The key process in Quick sort is partition algorithm.
- Purpose of partition algorithm is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.
- But choosing a right pivot is very important and affect the complexity of algorithm.
- Choose pivot – a random element initially (generally 0th index element)
- Once pivot gets final position then,

<=Pivot	Pivot	>Pivot

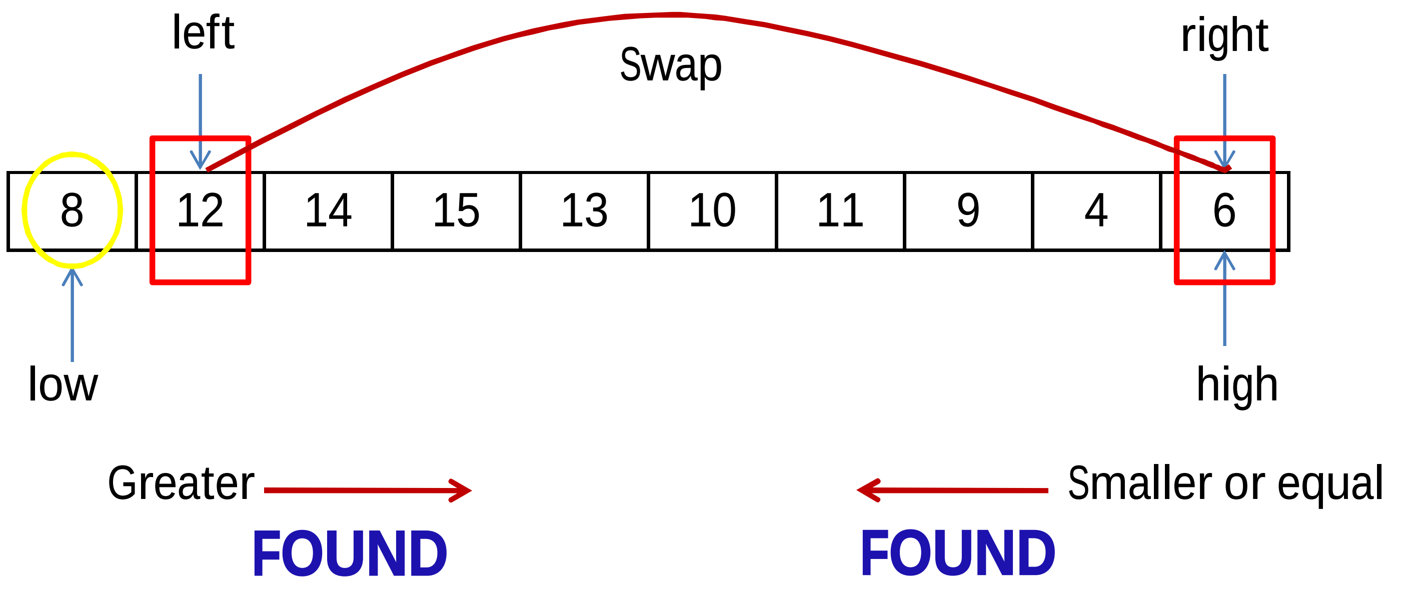
Quick Sort Illustration



pivot = A[low]

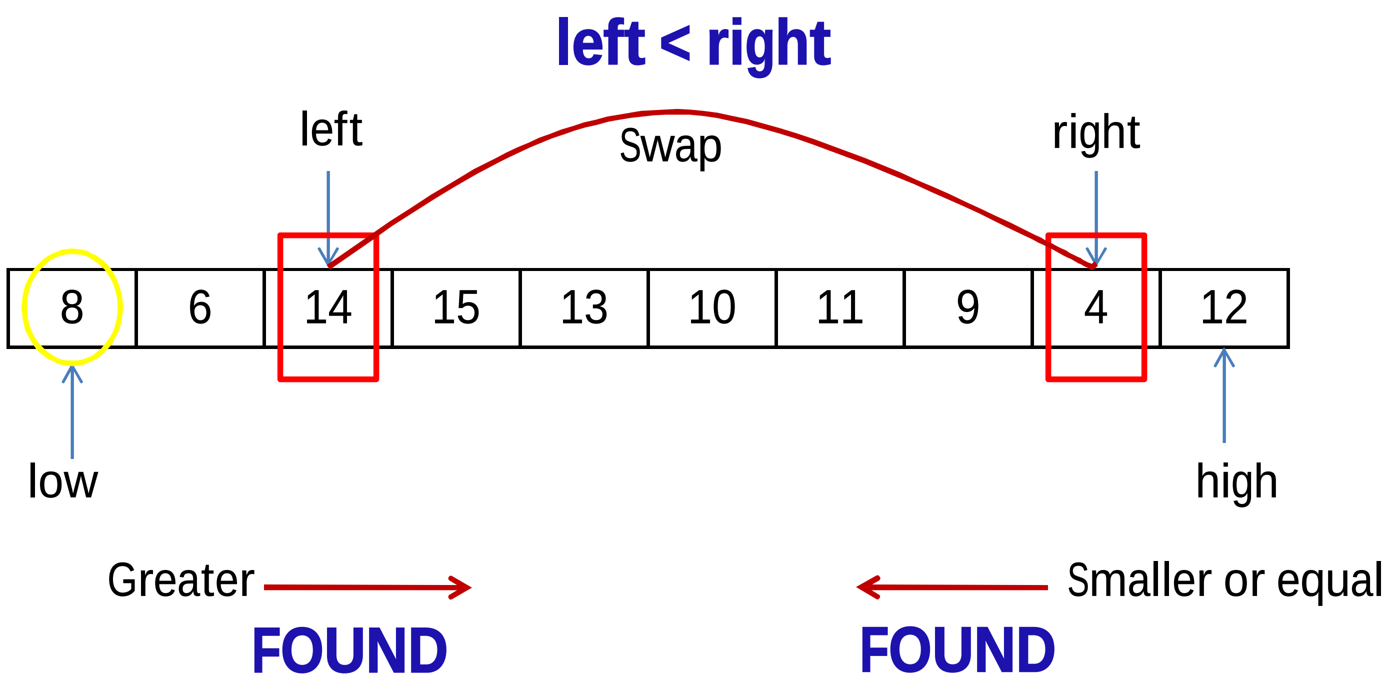
Quick Sort Illustration

left < right



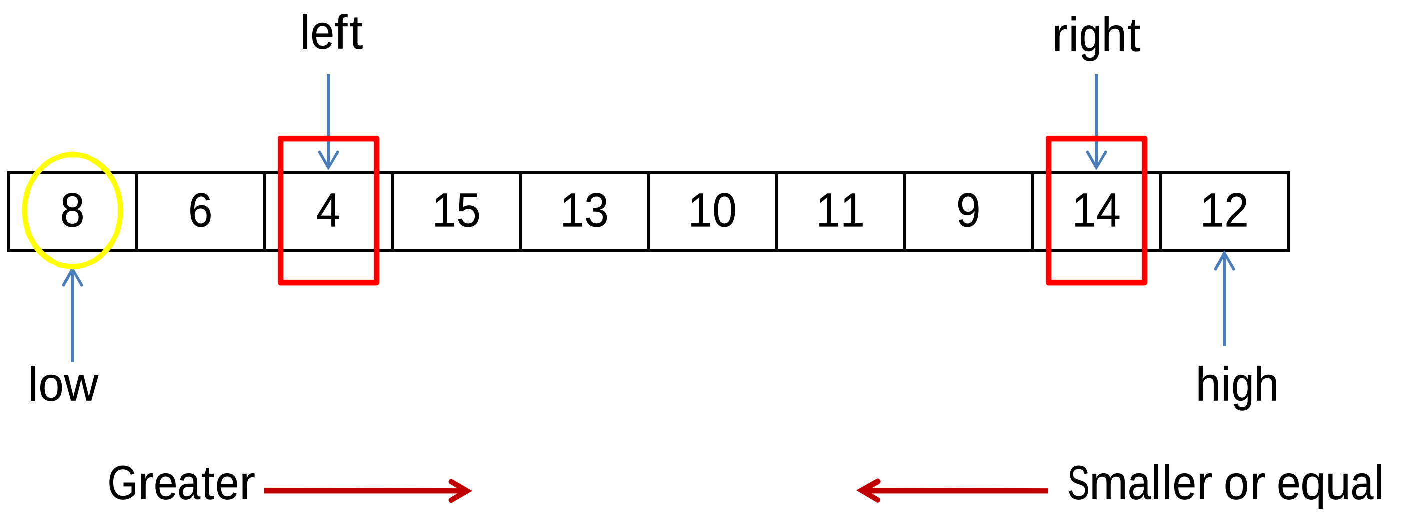
pivot = A[low]

Quick Sort Illustration



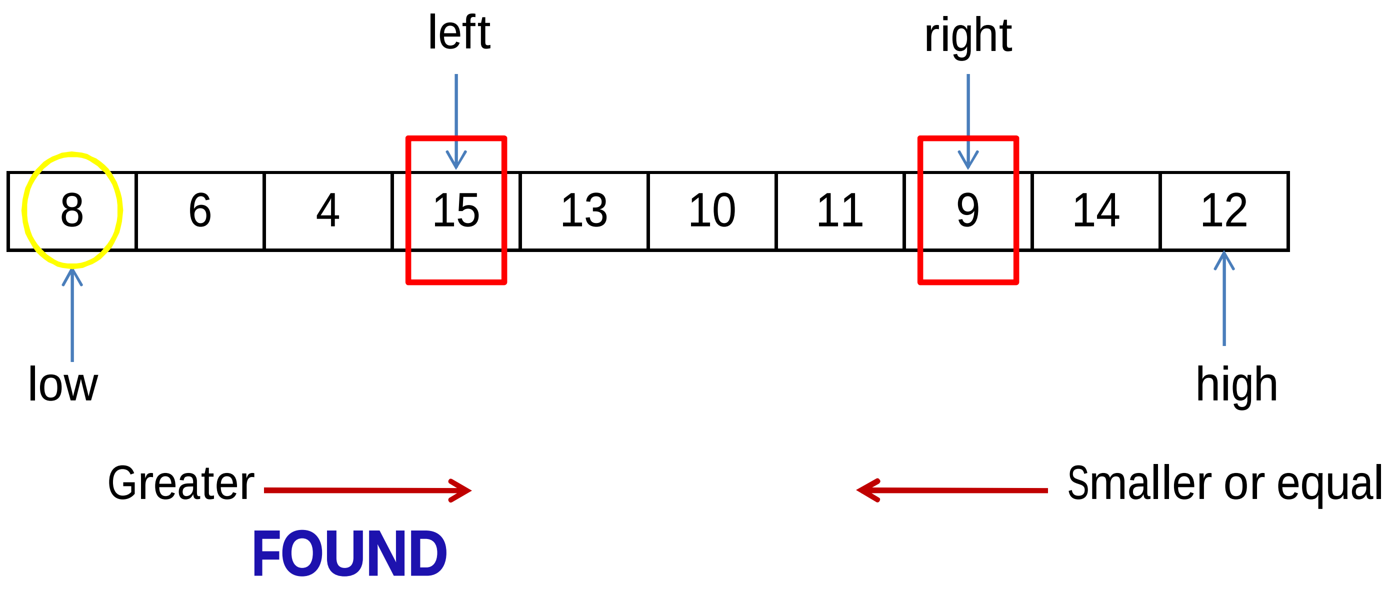
pivot = A[low]

Quick Sort Illustration



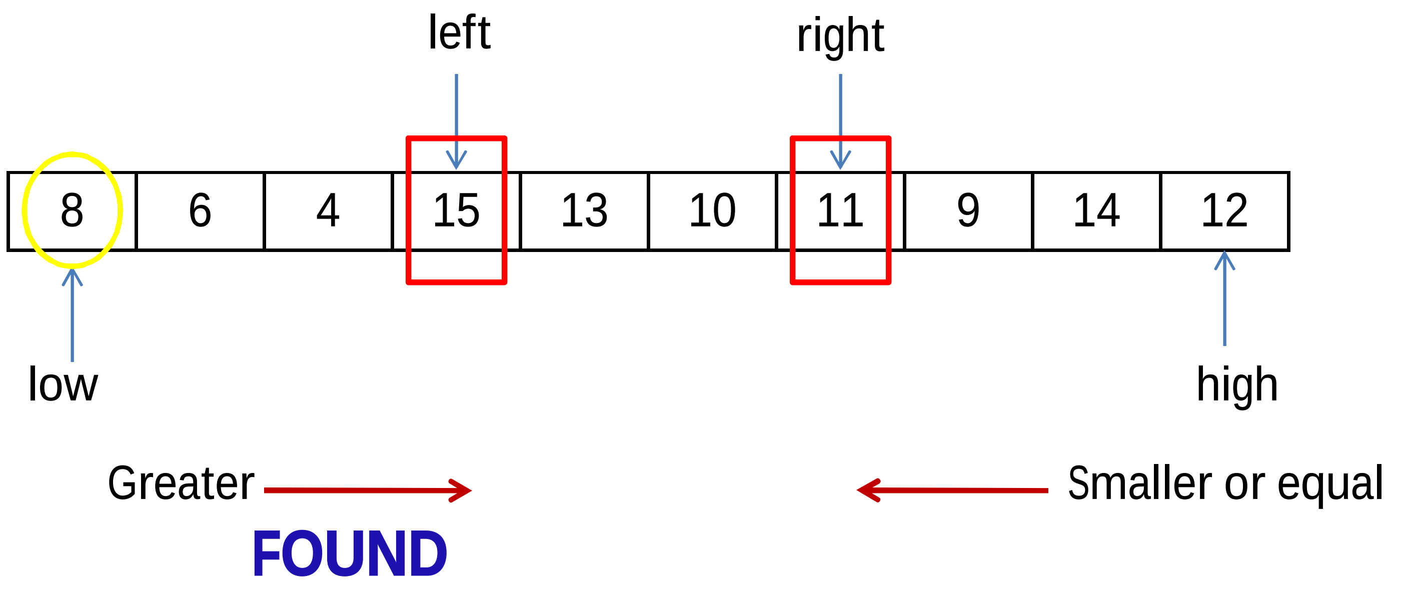
pivot = A[low]

Quick Sort Illustration



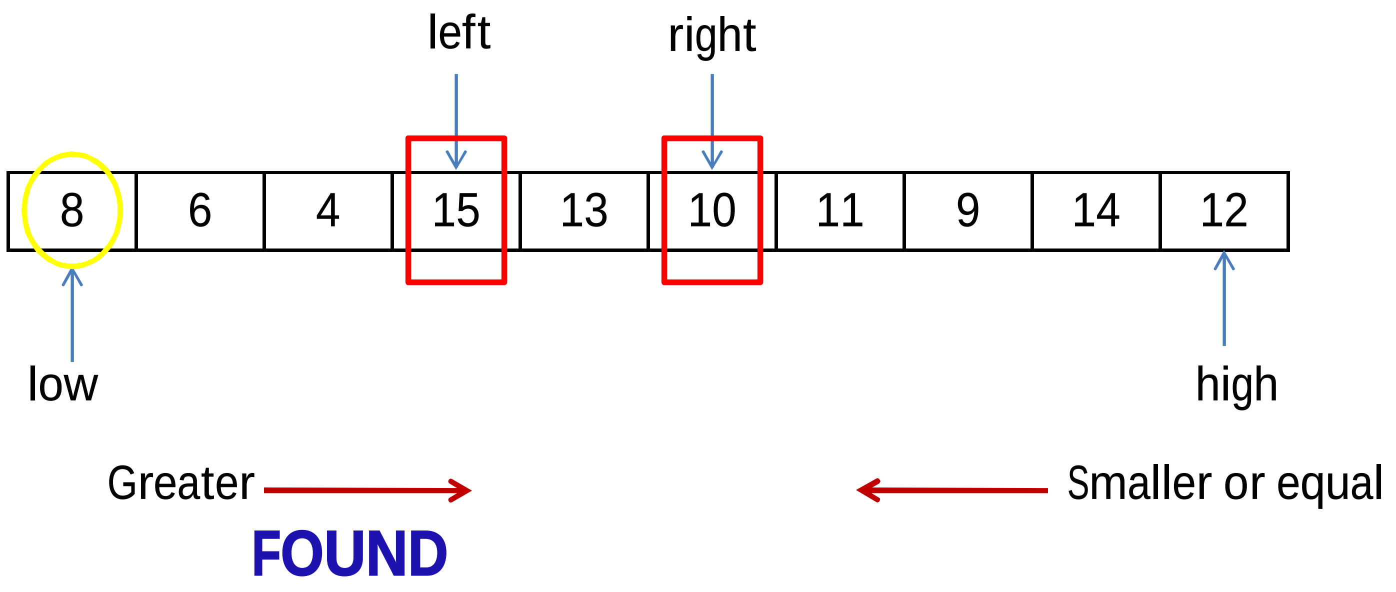
pivot = A[low]

Quick Sort Illustration



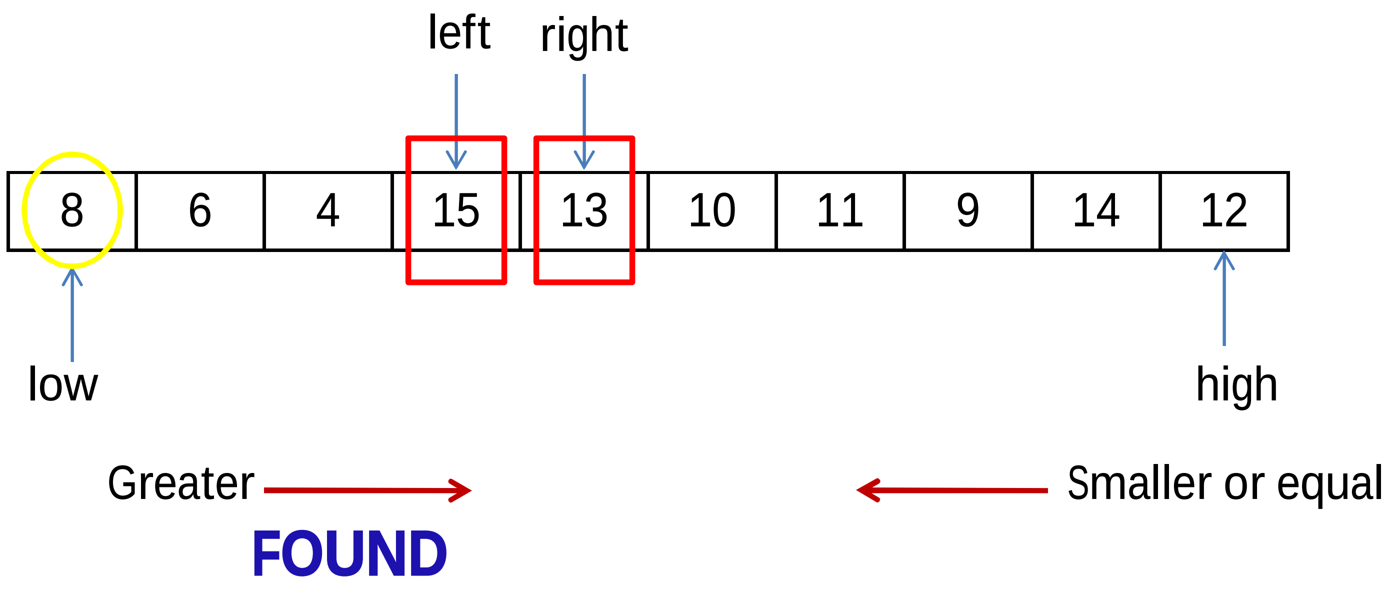
pivot = A[low]

Quick Sort Illustration



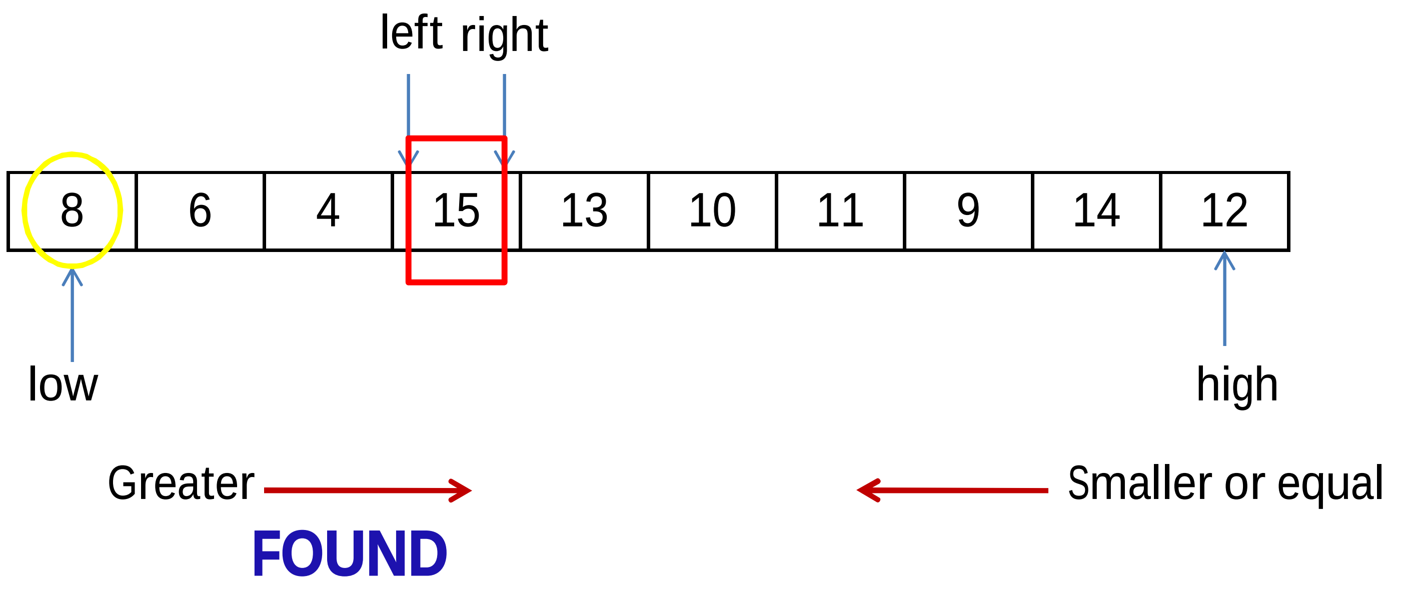
pivot = A[low]

Quick Sort Illustration



pivot = A[low]

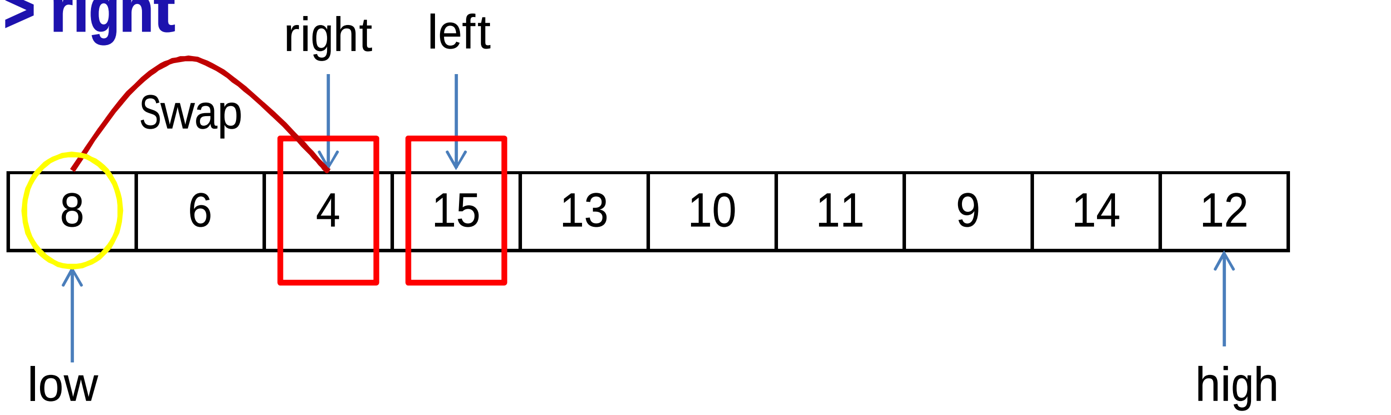
Quick Sort Illustration



pivot = A[low]

Quick Sort Illustration

left > right

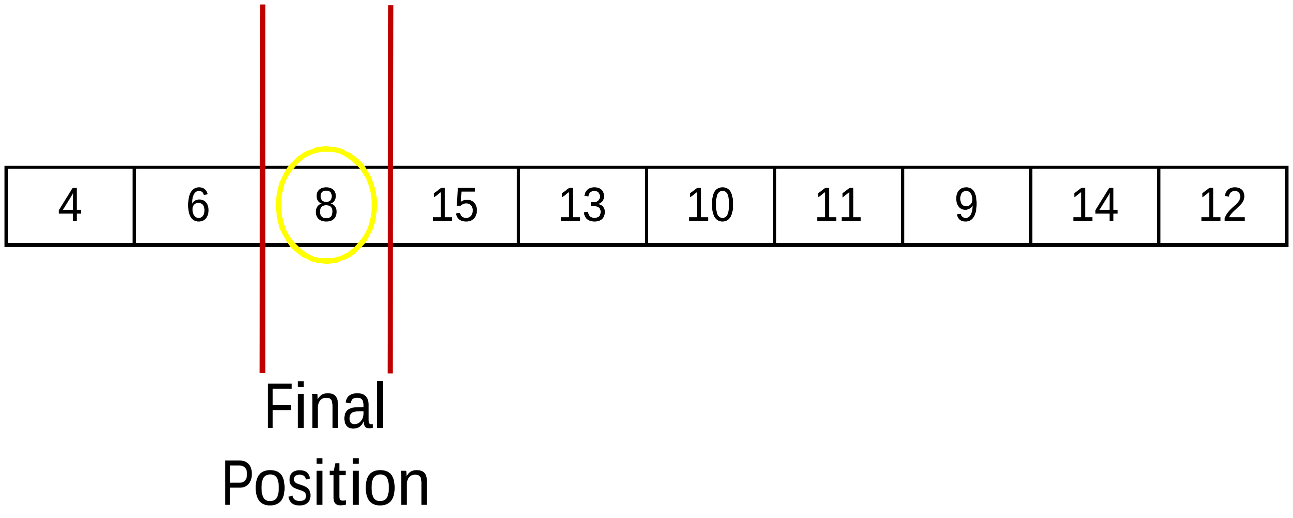


Greater → **FOUND**

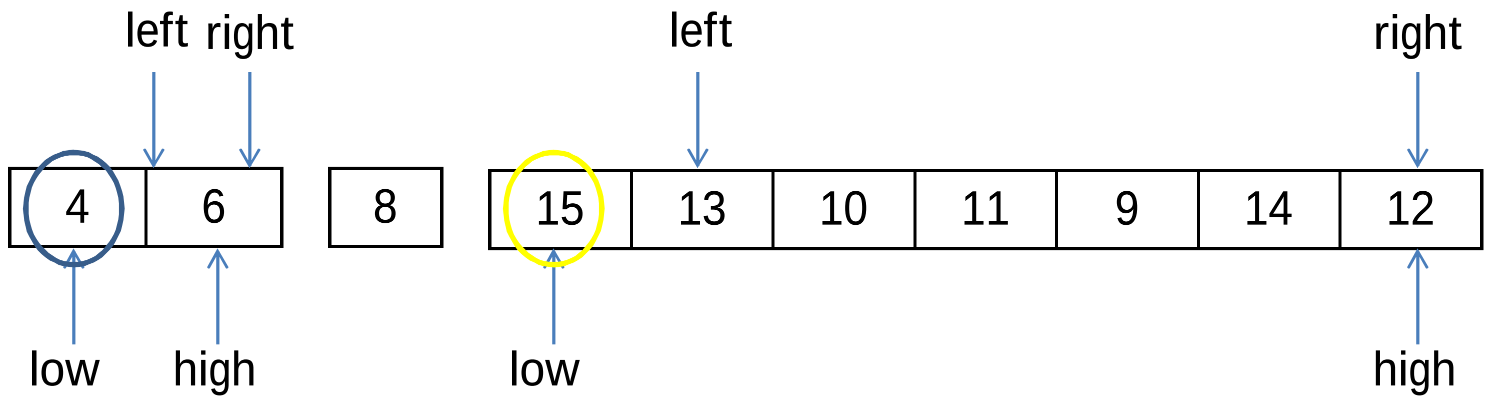
← Smaller or equal **FOUND**

pivot = A[low]

Quick Sort Illustration

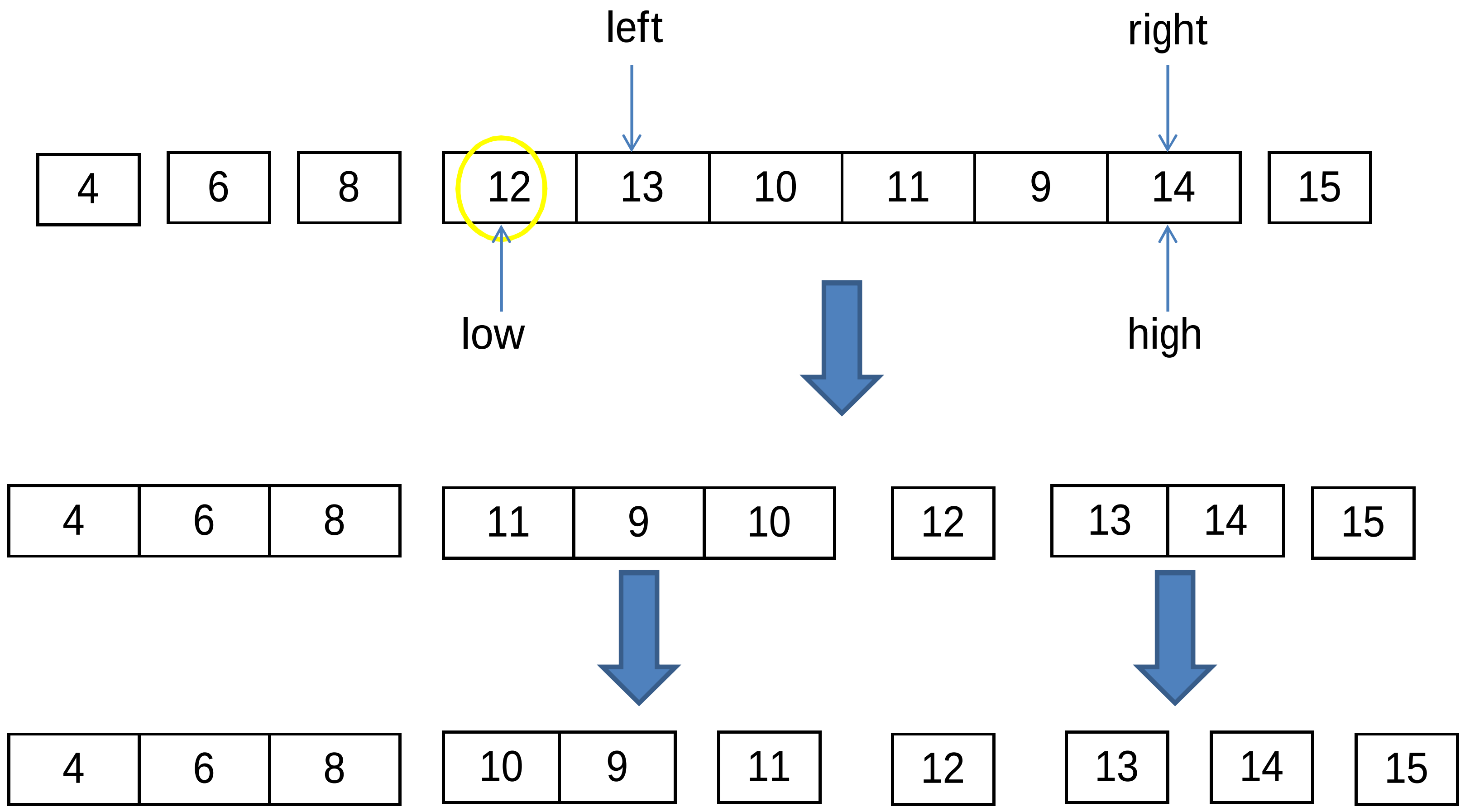


Quick Sort Illustration



pivot = A[low]

Quick Sort Illustration



Quick Sort Illustration

Final Output:

4	6	8	9	10	11	12	13	14	15

Quick sort

Algorithm Quick_sort(A,low,high): This algorithm sorts the elements in ascending order. A is linear array where low is equal to 0 and high is highest index i.e. n-1. A sub algorithm partition is used to split a array into two parts and it returns final position of pivot element.

Algorithm Quick_sort(A,low,high)

```
1.  If low < high  // Stop recursion when problem size is 1 or zero
2.      Loc_pivot = Partition(A, low, high)
3.      Quick_sort(A, low, Loc_pivot-1)
4.      Quick_sort(A, Loc_pivot+1, high)
```

Quick Sort Partition

Partition(A, low, high) – This algorithm splits the array into two parts and fixed the position of pivot element. Function **swap** is used to exchange the element in array.

```
1.  pivot = A[low]
2.  left = low+1
3.  right = high
4.  While ( left <= right )
5.      While(left<=high && A[left] <= pivot ) //move left while A[left]<=pivot
            left=left+1
6.      While( A[right] > pivot )   //move right while A[right]>pivot
7.          right=right-1
8.      If ( left < right )
9.          swap(A[left],A[right])
11. swap(A[right], A[low])  //Swap pivot with A[right]
12. Return right      //right is final position of pivot, so return right
```

Analysis of Quick Sort

Worst case: **When the list is already sorted.**

This happens when the pivot is the smallest (or the largest) element.

Then one of the partitions is empty, and we repeat recursively the procedure for N-1 elements.

Recurrence relation –

$$T(n) = \begin{cases} 1 & n=1 \text{ or } 0 \\ T(n-1) + n & n>1 \end{cases}$$

$$
\begin{aligned}
T(n) &= T(n-1) + n \\
&= T(n-2) + n-1 + n && //T(n-1) = T(n-2) + n-1 \\
&= T(n-2)+2n-1 \\
&= T(n-3) + (n-2)+2n-1 && //T(n-2) = T(n-3) + (n-2) \\
&= T(n-3)+3n-1-2 \\
&= T(n-4) + (n-3)+3n-1-2 && //T(n-3) = T(n-4) + (n-3) \\
&= T(n-4) + 4n-1-2-3
\end{aligned}
$$

....

$$
\begin{aligned}
&= T(n-k)+kn-1-2-3-\text{............}-(k-1) \\
&= T(n-k) + kn - k(k-1)/2
\end{aligned}
$$

Let $n-k=0 \Rightarrow k=n$ and $T(0)=1$

$T(n) = T(0) + n^2 - n(n-1)/2$

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

Analysis of Quick Sort (Cont…)

Best case:

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

Recurrence relation –

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

Time Complexity = $O(n\log_2 n)$

Analysis of Quick Sort (Cont…)

$T(n) = 2\,T(n/2) + n$

$= 2\,[\mathbf{2\,T(n/4) + n/2}] + n$

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

$= 4\,[\mathbf{2\,T(n/8) + n/4}] + 2n$

$= 8\,T(n/8) + 3n$

$= \mathit{16\,T(n/16)} + \mathit{4n}$

……………

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

$2^k=n$

$\Rightarrow k=\log n$

$=nT(1)+n\log n$

$=n+n\log n$

Time Complexity = $O(n\log_2 n)$

Analysis of Quick Sort

Worst case Time Complexity: O(n^2)
Average case time complexity : O(nlogn)
Best case time complexity : O(nlong)
Auxiliary Space: O(1)
Boundary Cases: Quick sort takes maximum time to sort if elements are already sorted or reverse sorted and pivot is first or last element. And it takes minimum time (nlogn) when pivot is middle element.
Algorithmic Paradigm: Divide and Conquer Approach
Sorting In Place: Yes
Stable: No

Merge Sort

- It follows the strategy of divide and conquer.
- Divide the list of numbers into sub-list until no more division possible.
- Then merge them into one list by comparing them.
- It is not a in **place sort** i.e. extra memory of n elements is required.

Now suppose, low and high are the indexes of first and last element respectively then,

Logically,

```
If (low < high) then,
        Divide array into two half size array
        Perform merge sort for left half array
        Perform merge sort for right half array
        Merge two sorted array into one final sorted array.
End
```

Merge Sort

Algorithm Merge_sort(A,low,high): This algorithm sort the elements in ascending order. A is linear array where low is equal to 0 and high is highest index i.e. n-1. Its uses a variable mid to represent the index of the element at which array will be divided into two half arrays. A sub algorithm Merge is used to combine two half arrays into final sorted array.

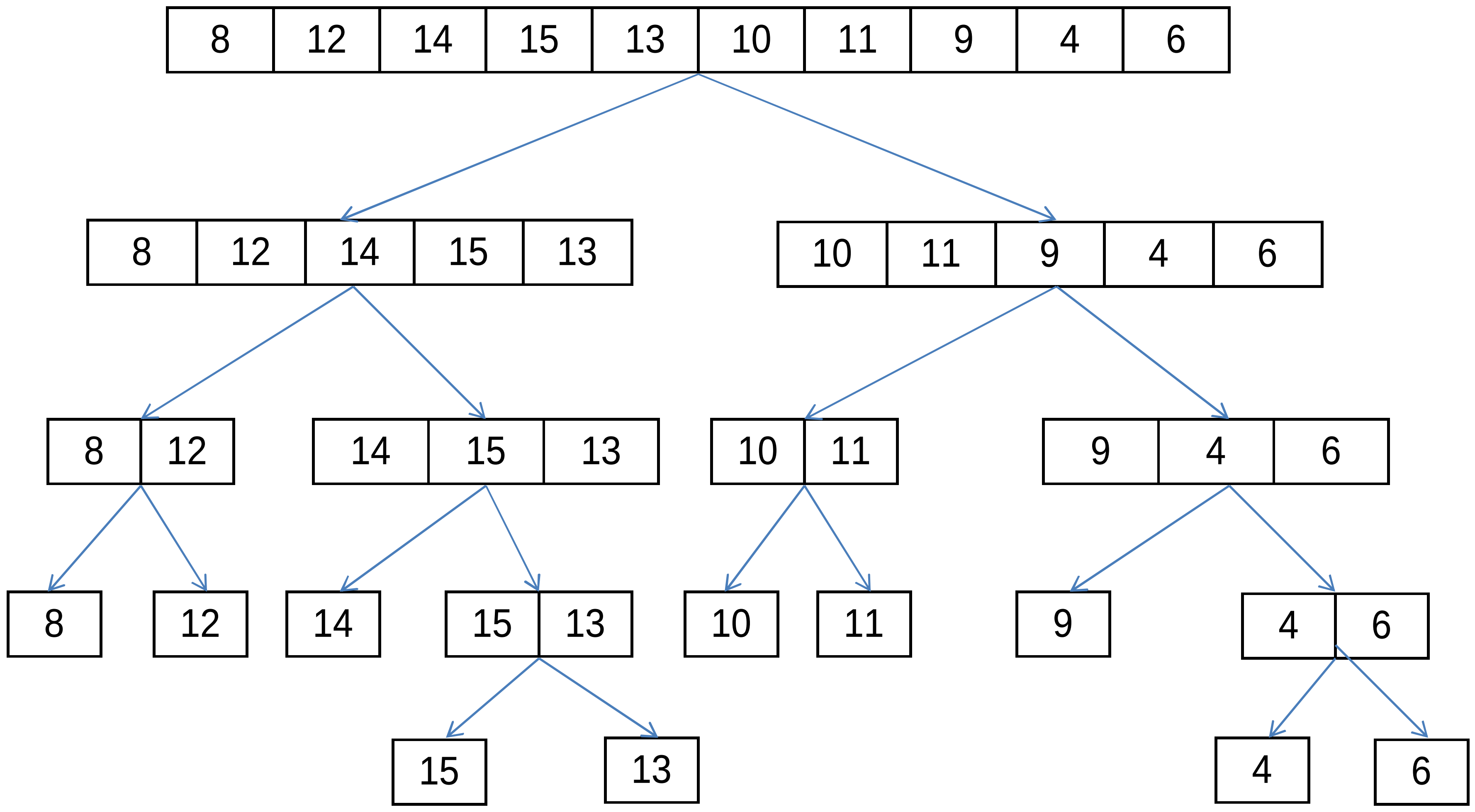
```
1.  If  low < high // Stop dividing array when array size is 1 or zero
2.          mid = Ceiling [(low + high)/2]
3.          Merge_sort(A, low, mid-1) // apply merge sort on first half
4.          Merge_sort(A, mid, high)  //apply merge sort on second half
5.          Merge(A, low, mid-1, mid ,high)
```

Merge Sort Merge

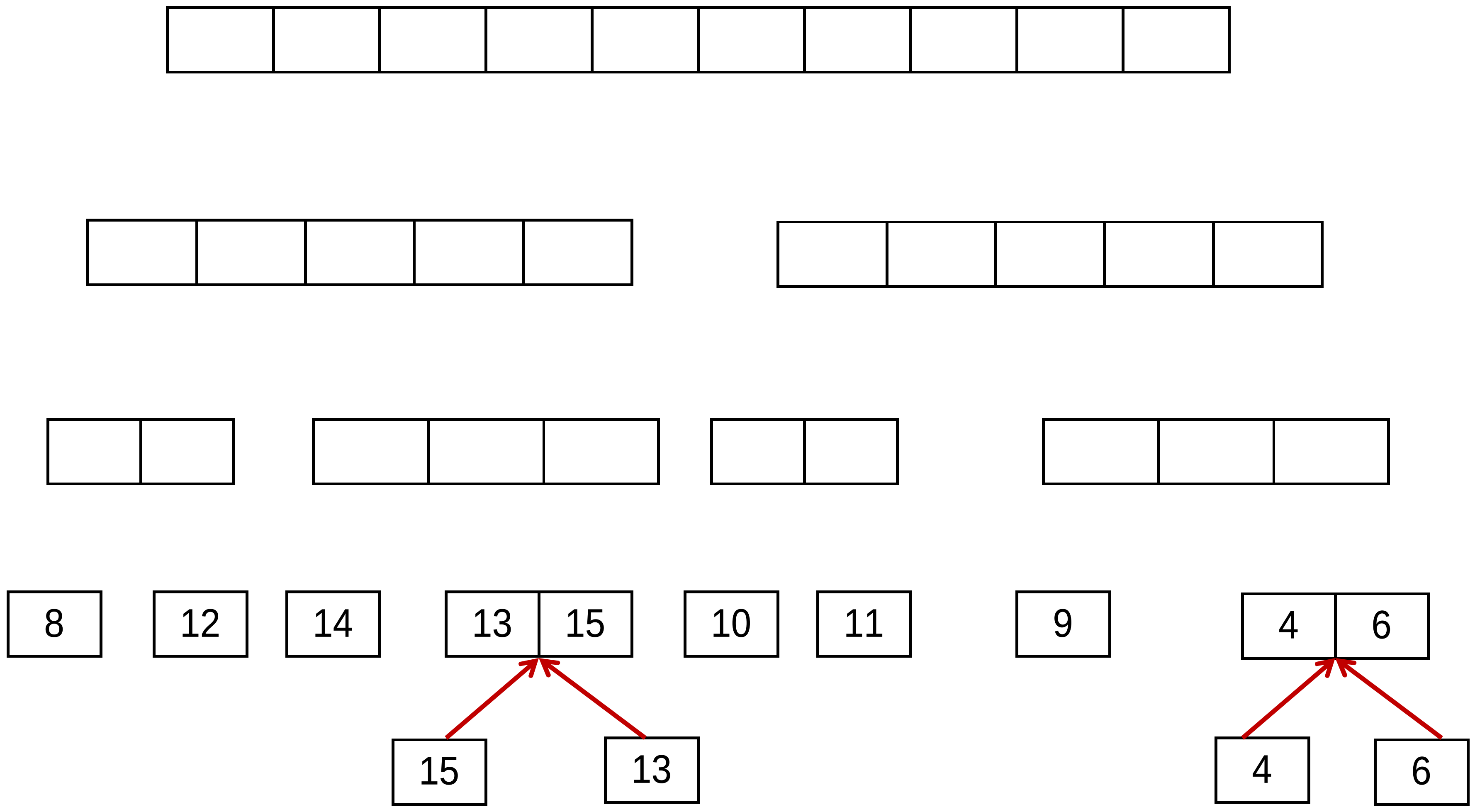
Merge(A, low, mid1, mid2, high) – This algorithm merge two half arrays into sorted array. It uses some local variable left, right to keep track of indexes in sub arrays. Temp is a temporary array of size n which is used to keep the sorted elements. **C** represents the counter which is used to keep track of the indexes of Temp array.

```
1.  C = low
2.  left = low
3.  right = mid2
4.  Temp[high-low+1] // Temp array of size high-low+1
5.  While ( left <= mid1) and (right <= high) //while elements in both sub-arrays
6.      If( A[left] <= A[right])
7.              Temp[C]= A[left]
8.              left=left+1
9.      Else
10.             Temp[C]= A[right]
11.             right=right+1
12.     C =C+1
13. While(left < = mid1) //copy remaining elements of the left sub-array
14.             Temp[C]= A[left]
15.             left=left+1
16.             C = C+1
17. While(right < = high)  //copy remaining elements of the right sub-array
18.             Temp[C]= A[right]
19.             right=right+1
20.             C = C+1
21. For (i=low, j=0; i<=high ; i++,j++)
22.     A[i] = Temp [j]
```

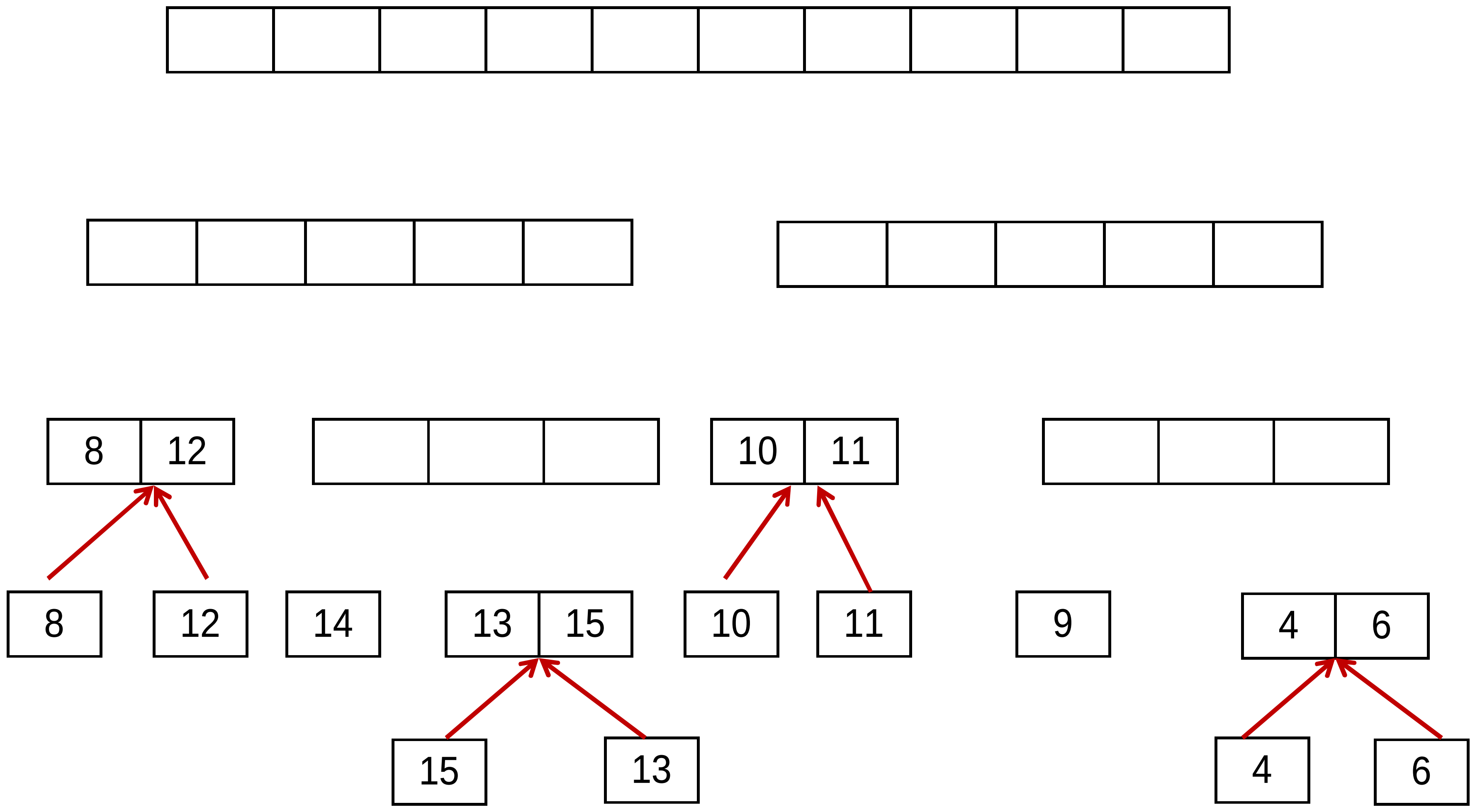
Merge Sort Illustration



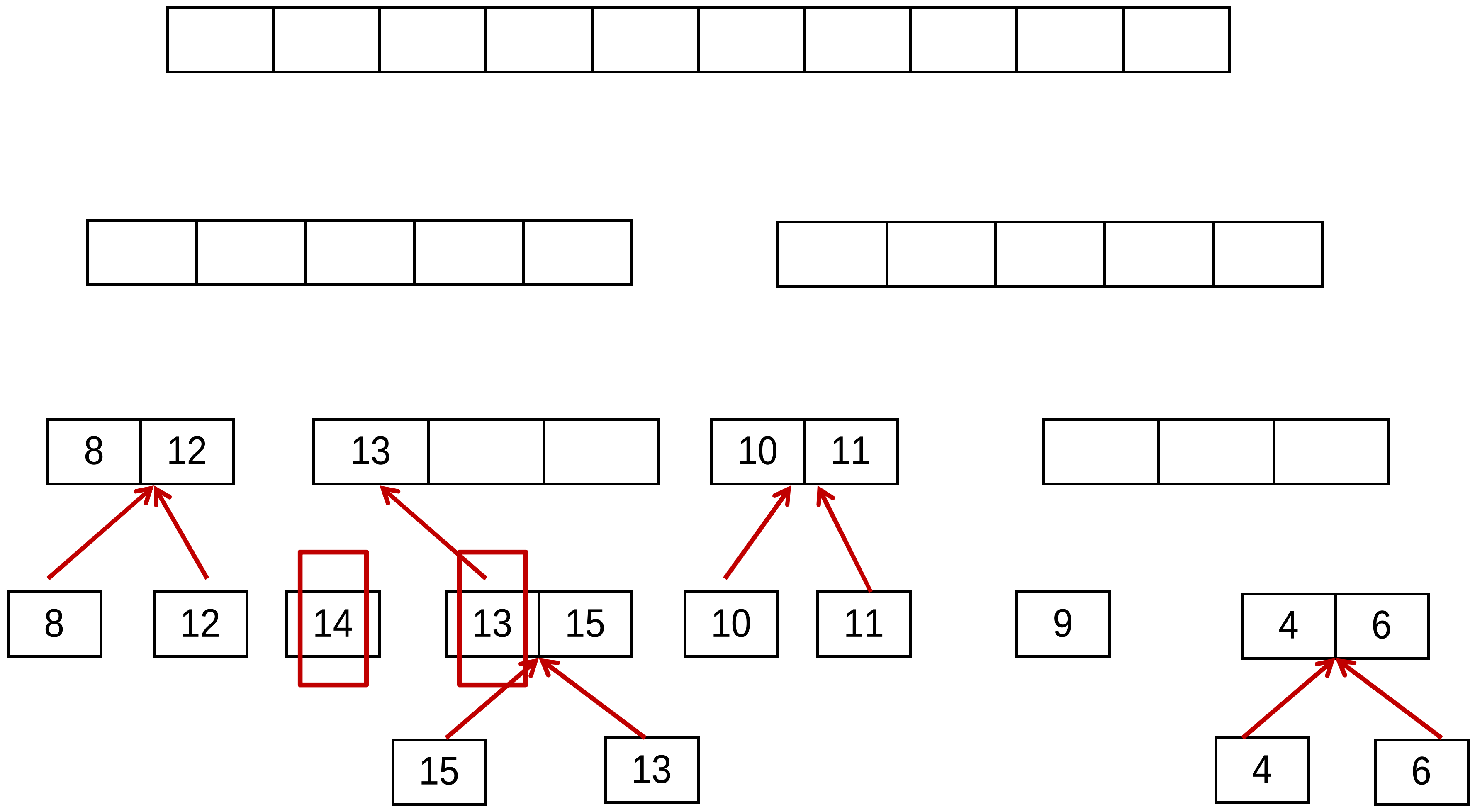
Merge Sort Illustration



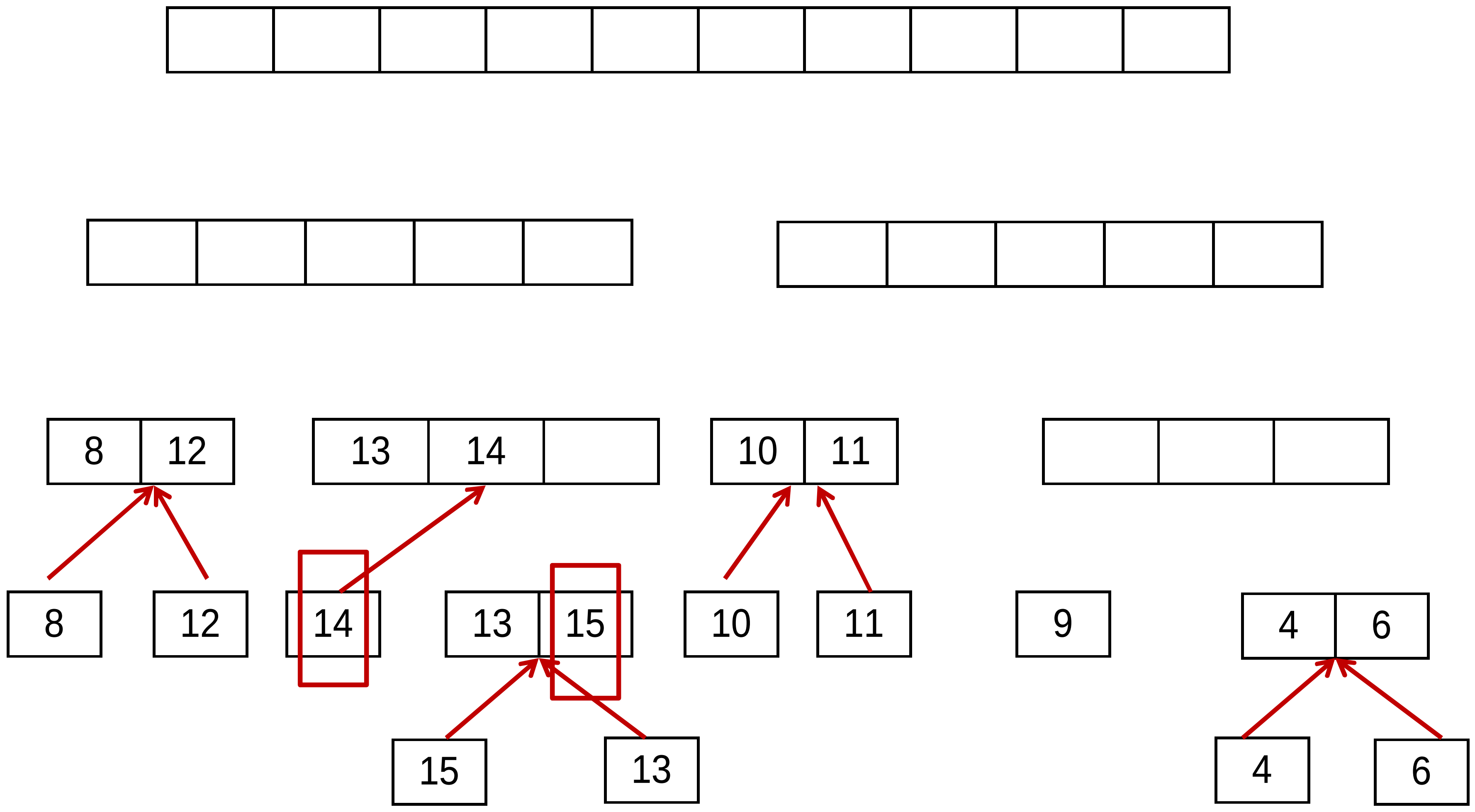
Merge Sort Illustration



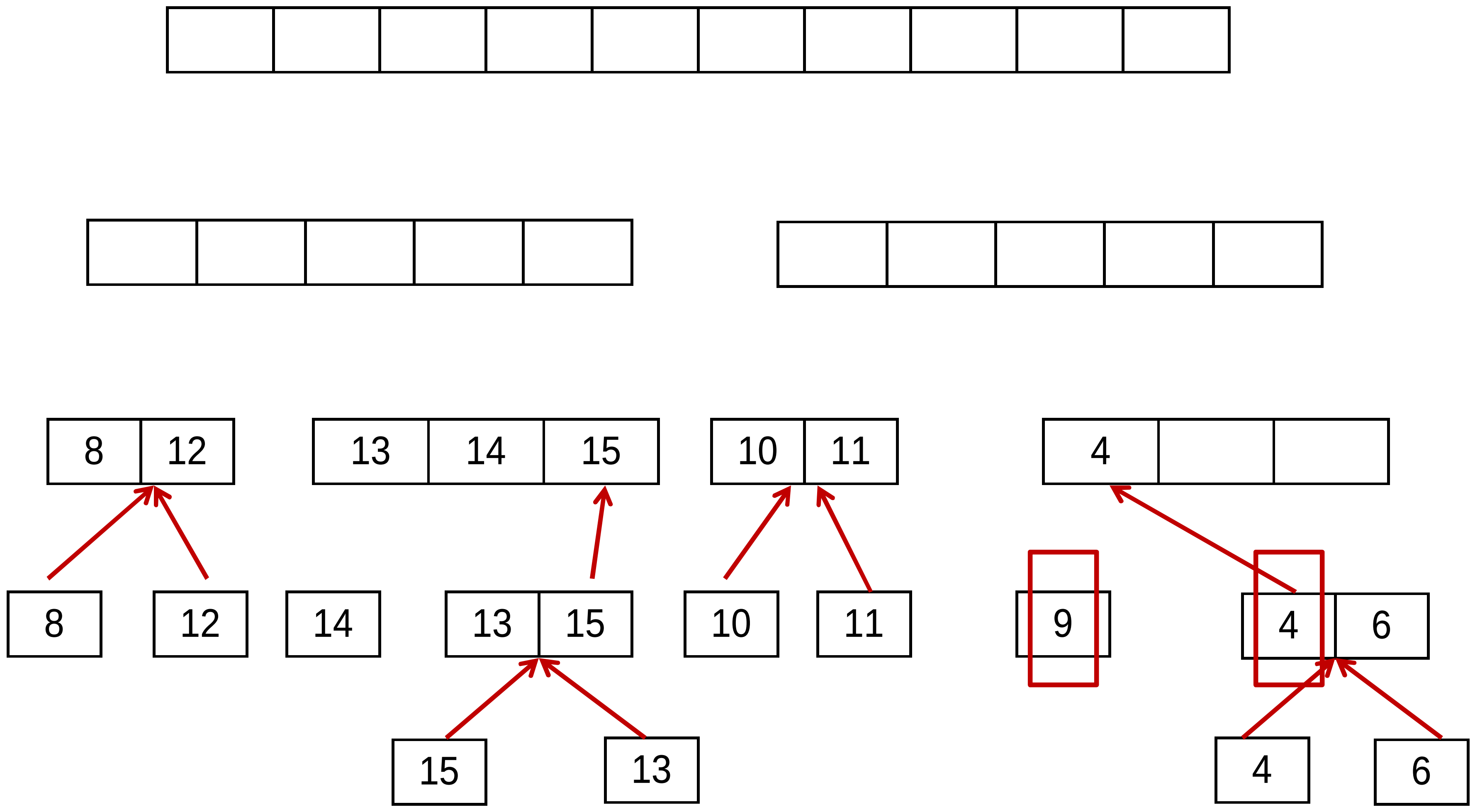
Merge Sort Illustration



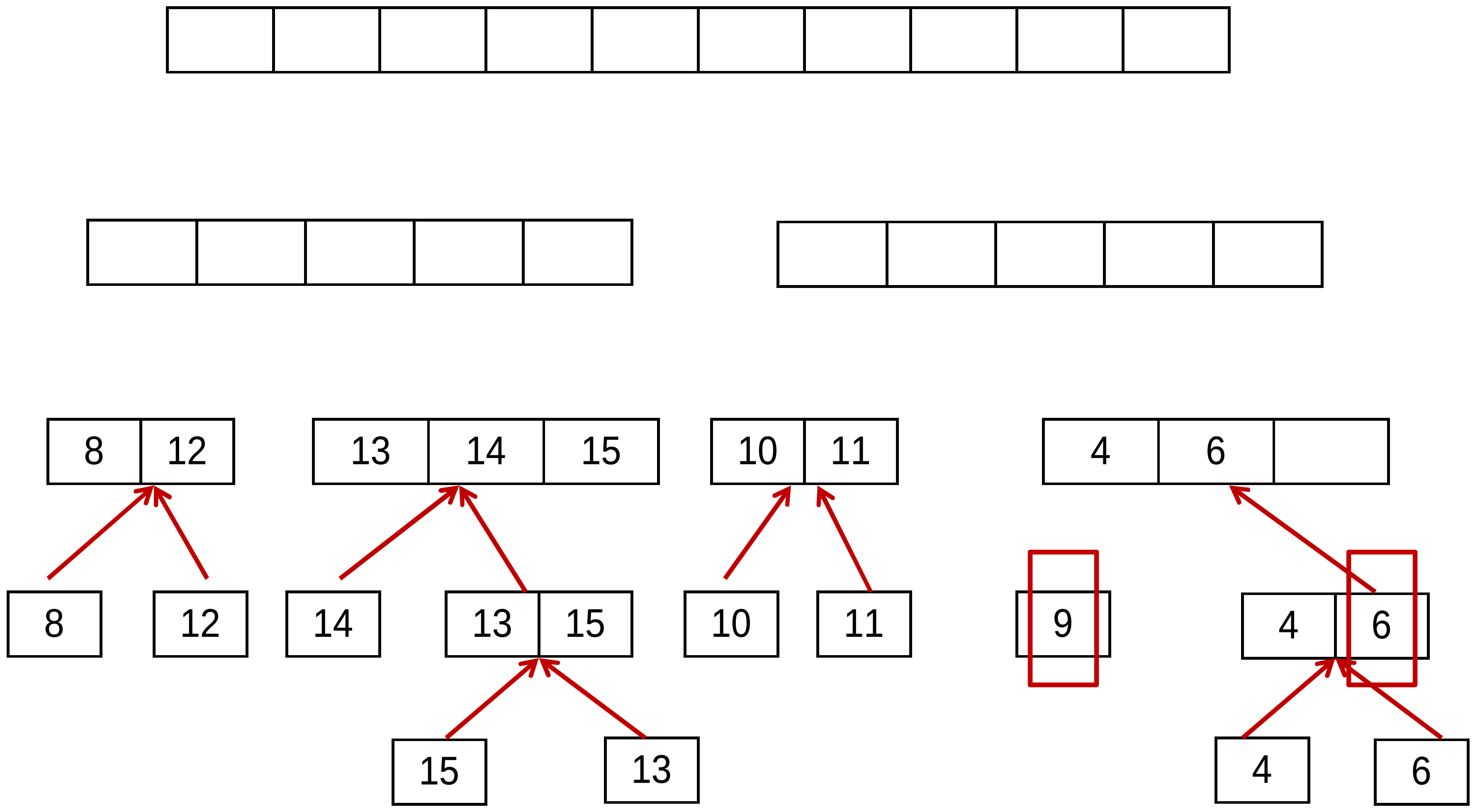
Merge Sort Illustration



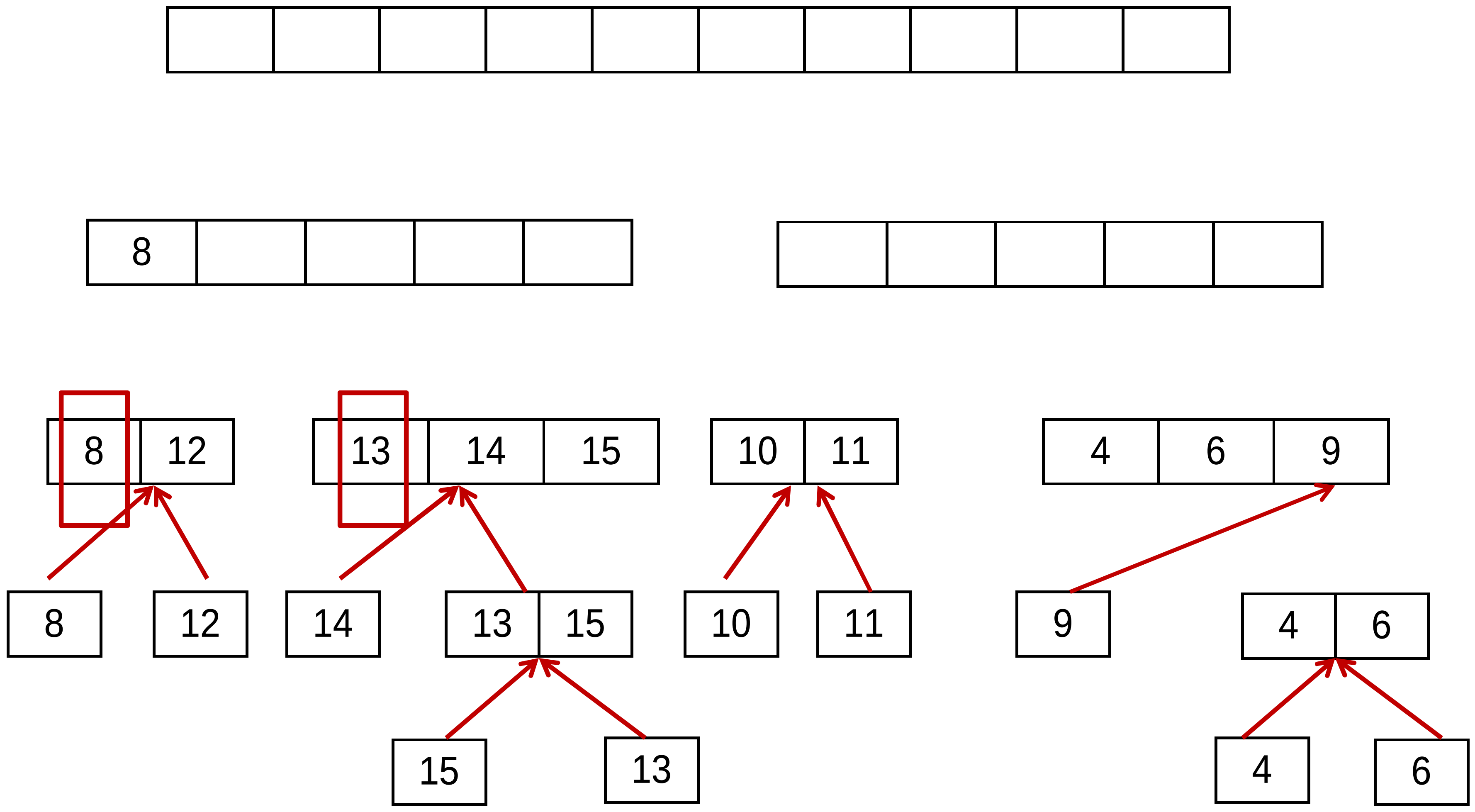
Merge Sort Illustration



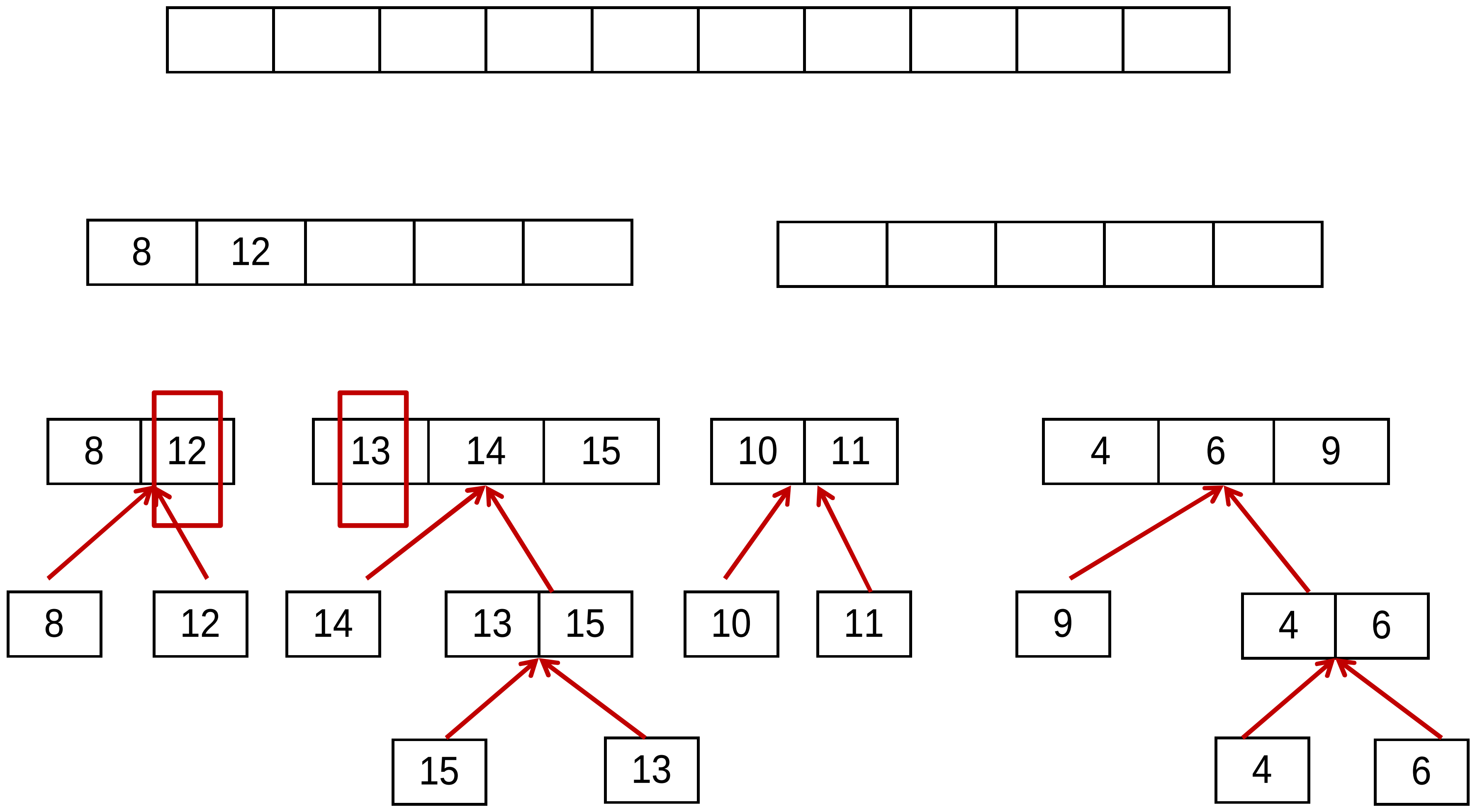
Merge Sort Illustration



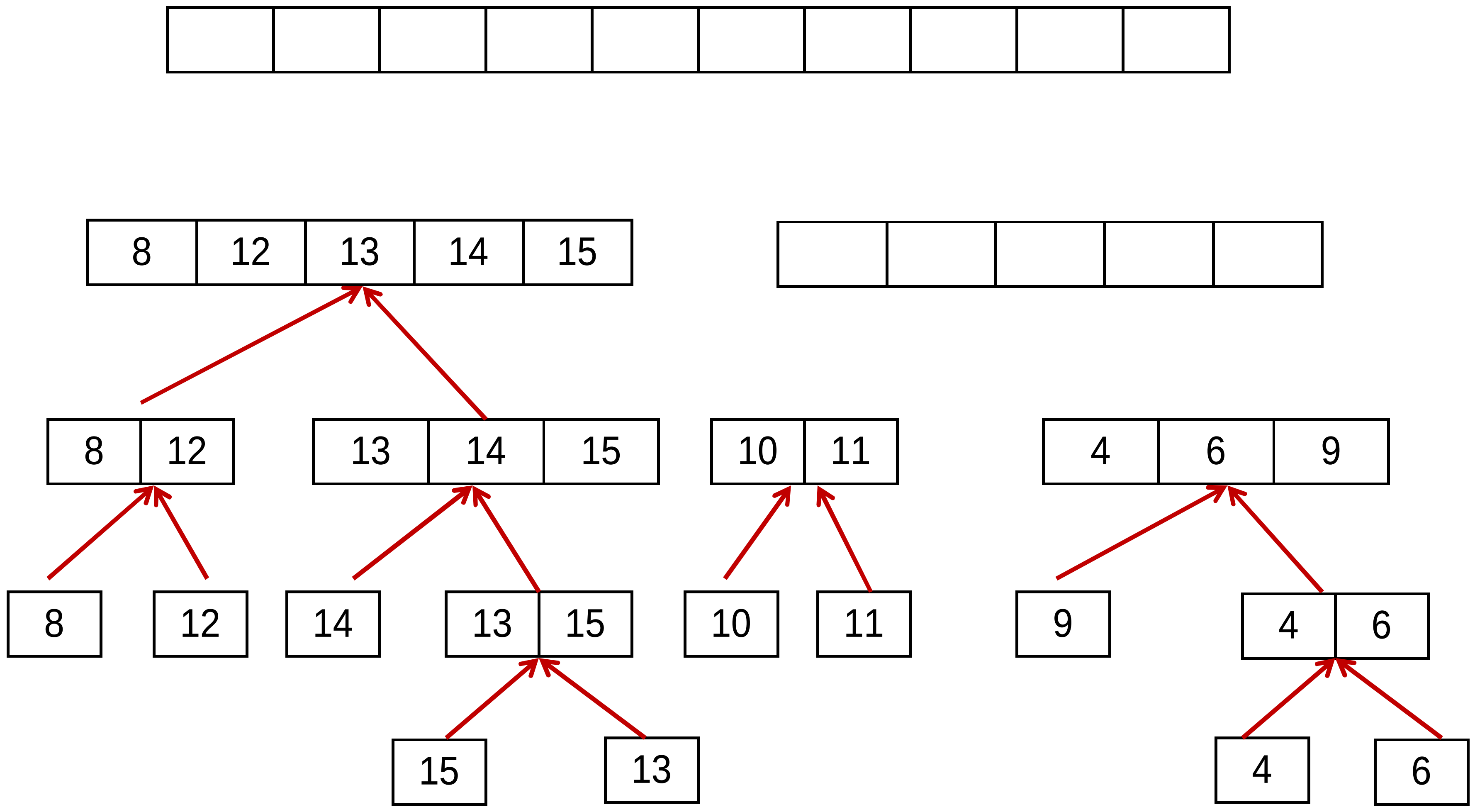
Merge Sort Illustration



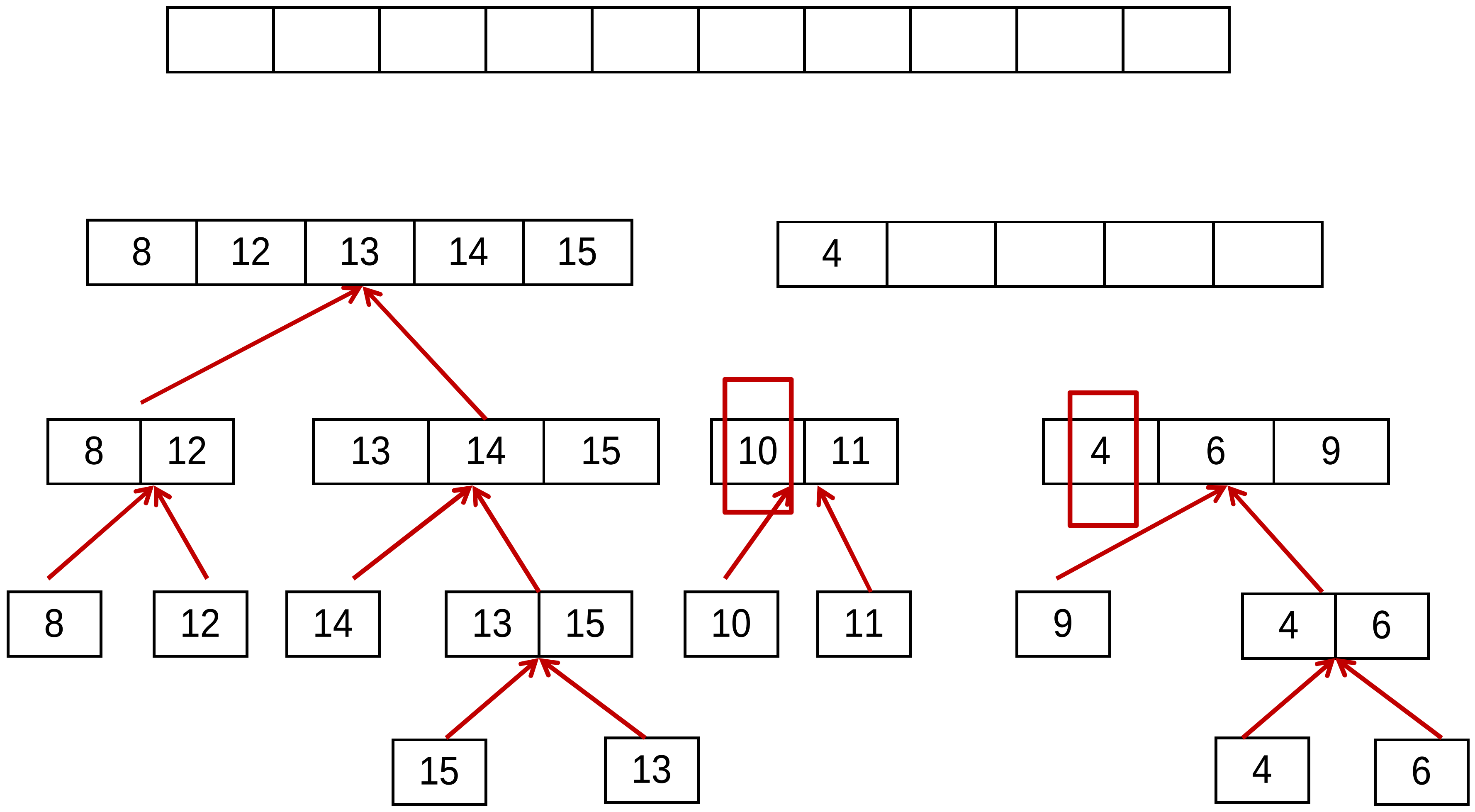
Merge Sort Illustration



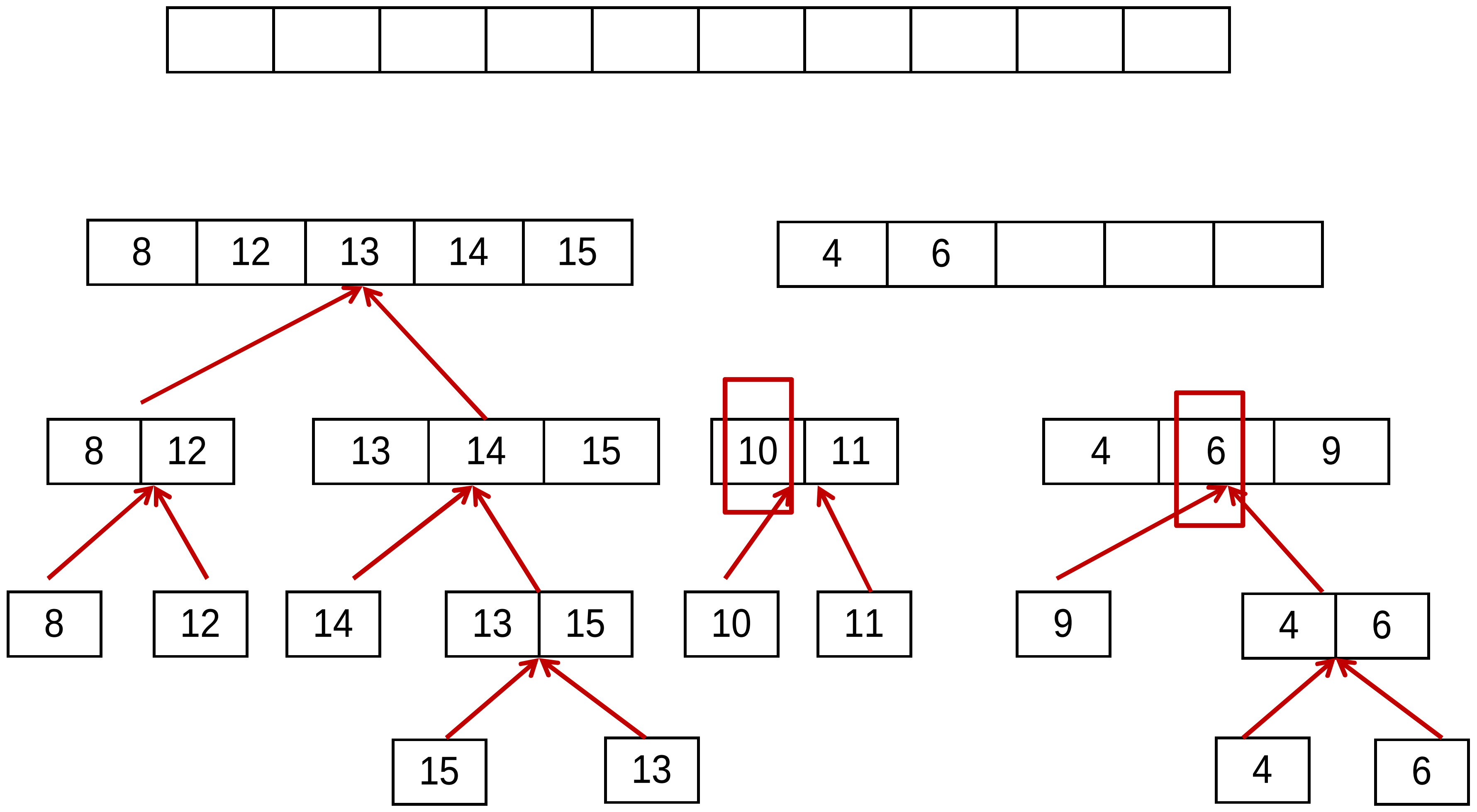
Merge Sort Illustration



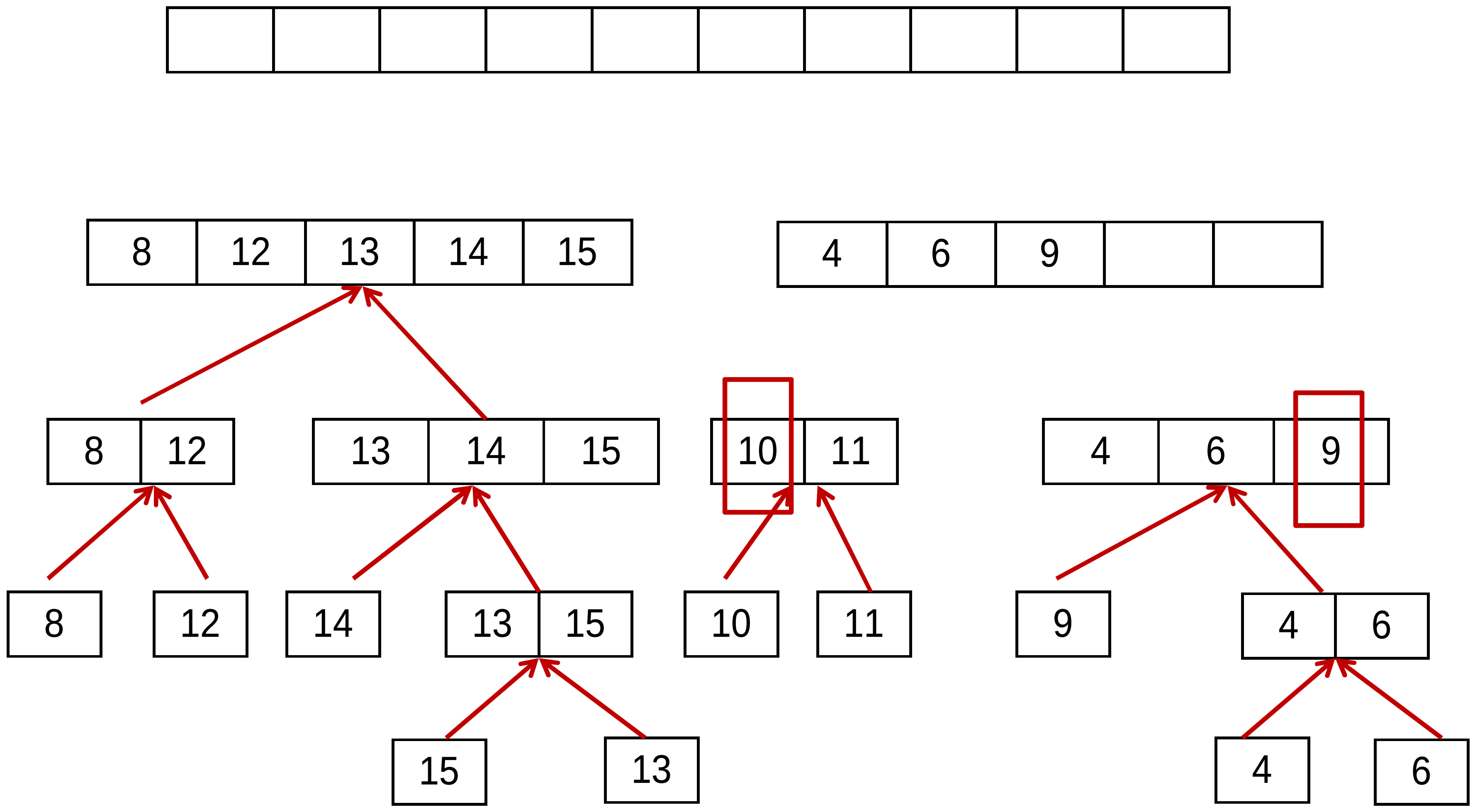
Merge Sort Illustration



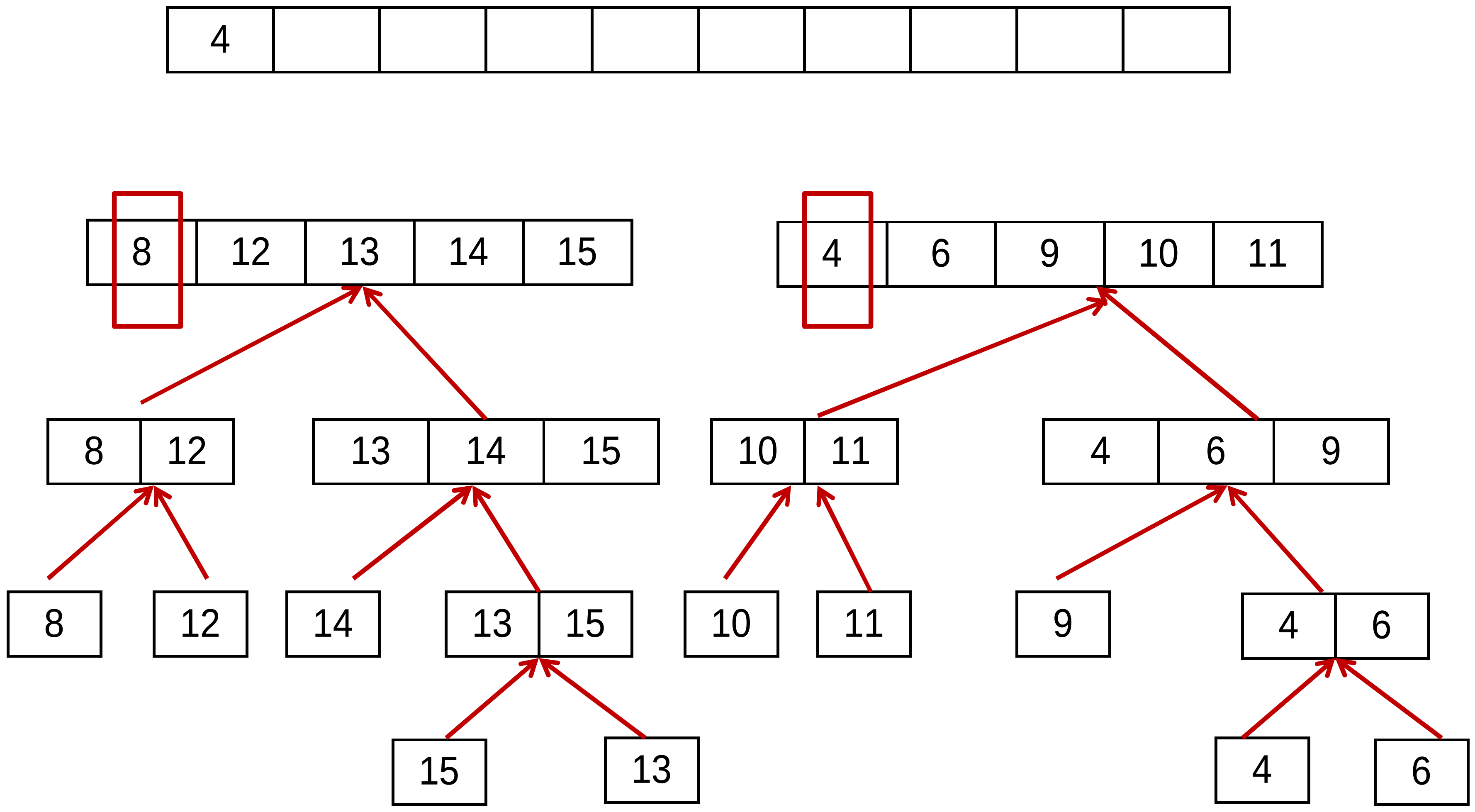
Merge Sort Illustration



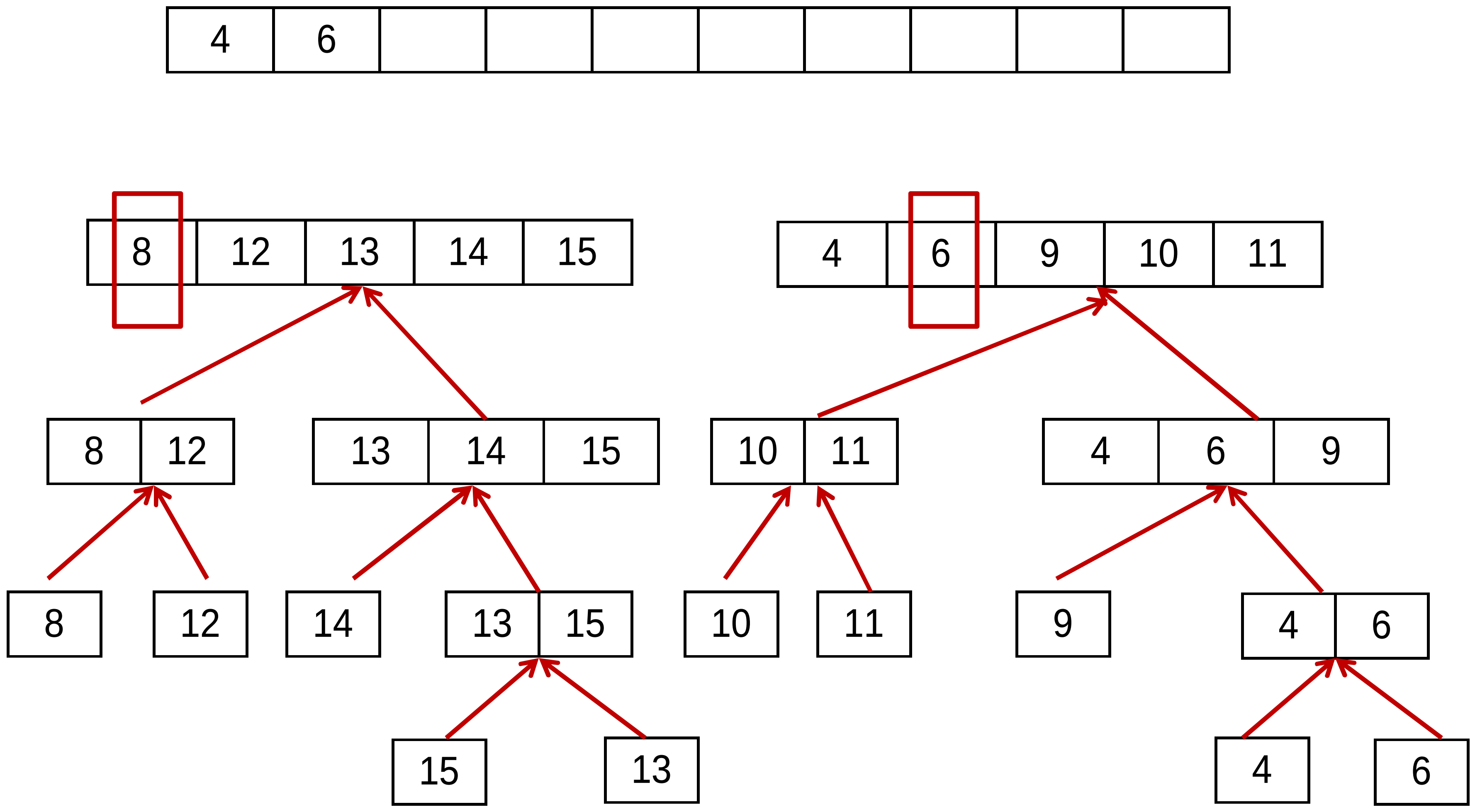
Merge Sort Illustration



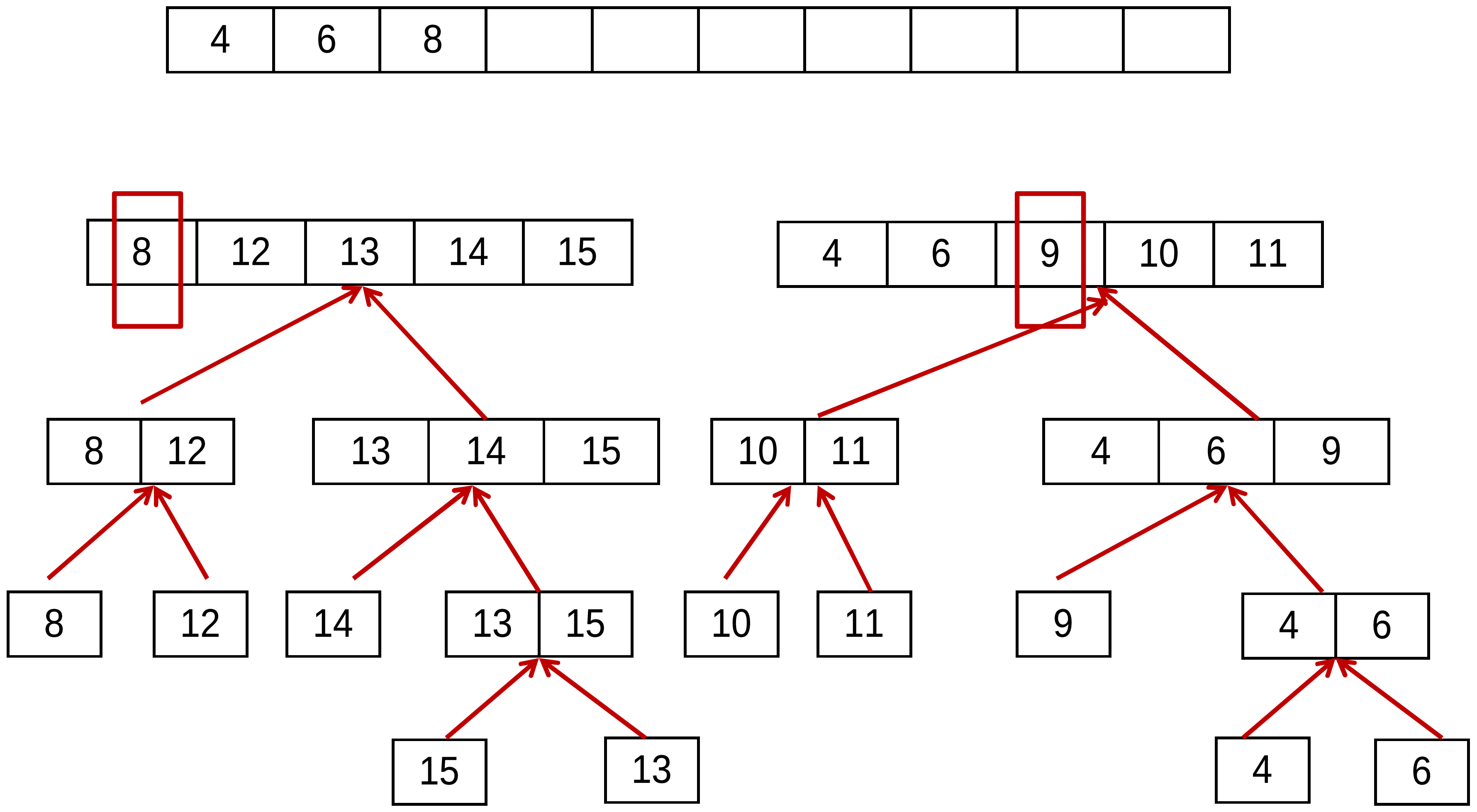
Merge Sort Illustration



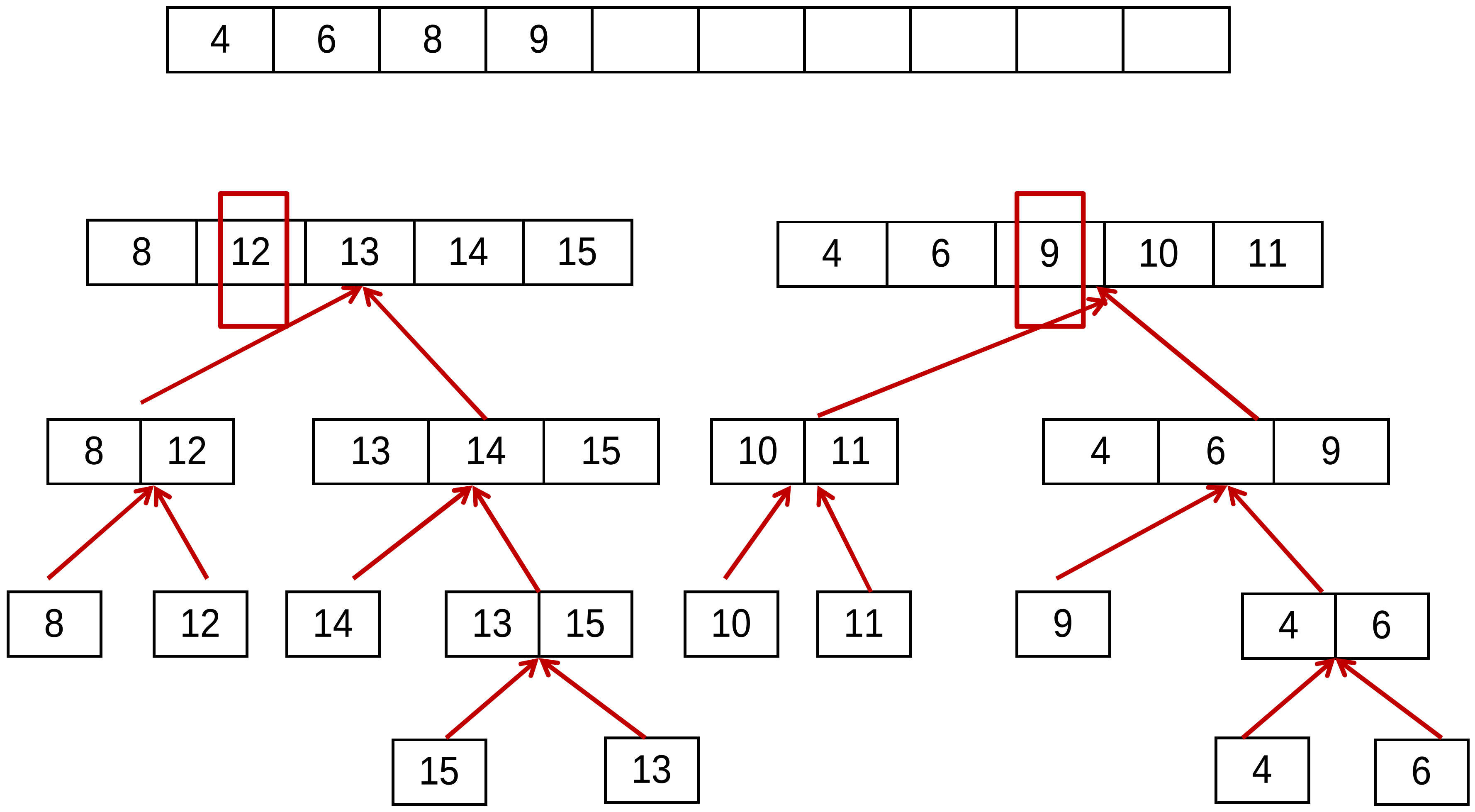
Merge Sort Illustration



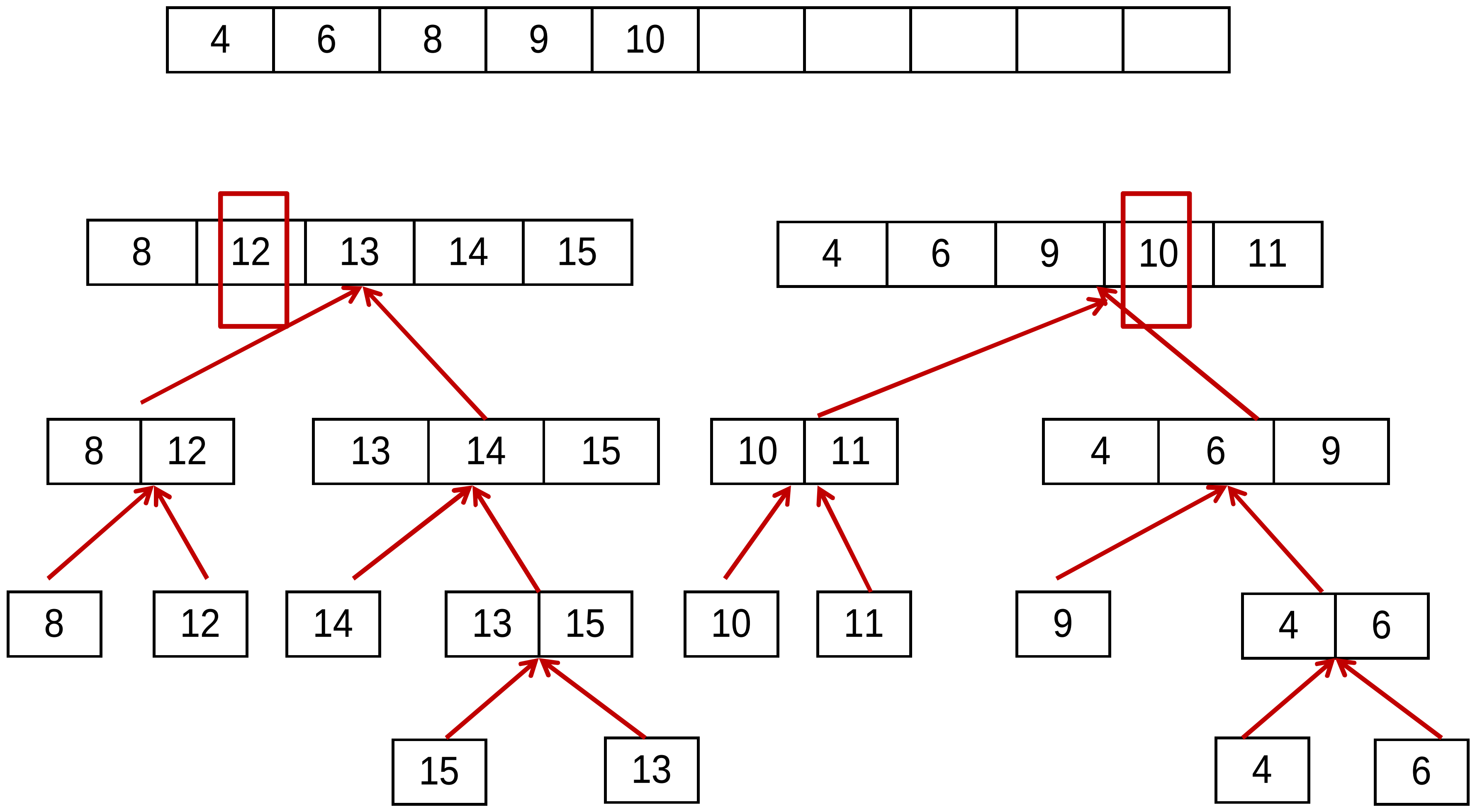
Merge Sort Illustration



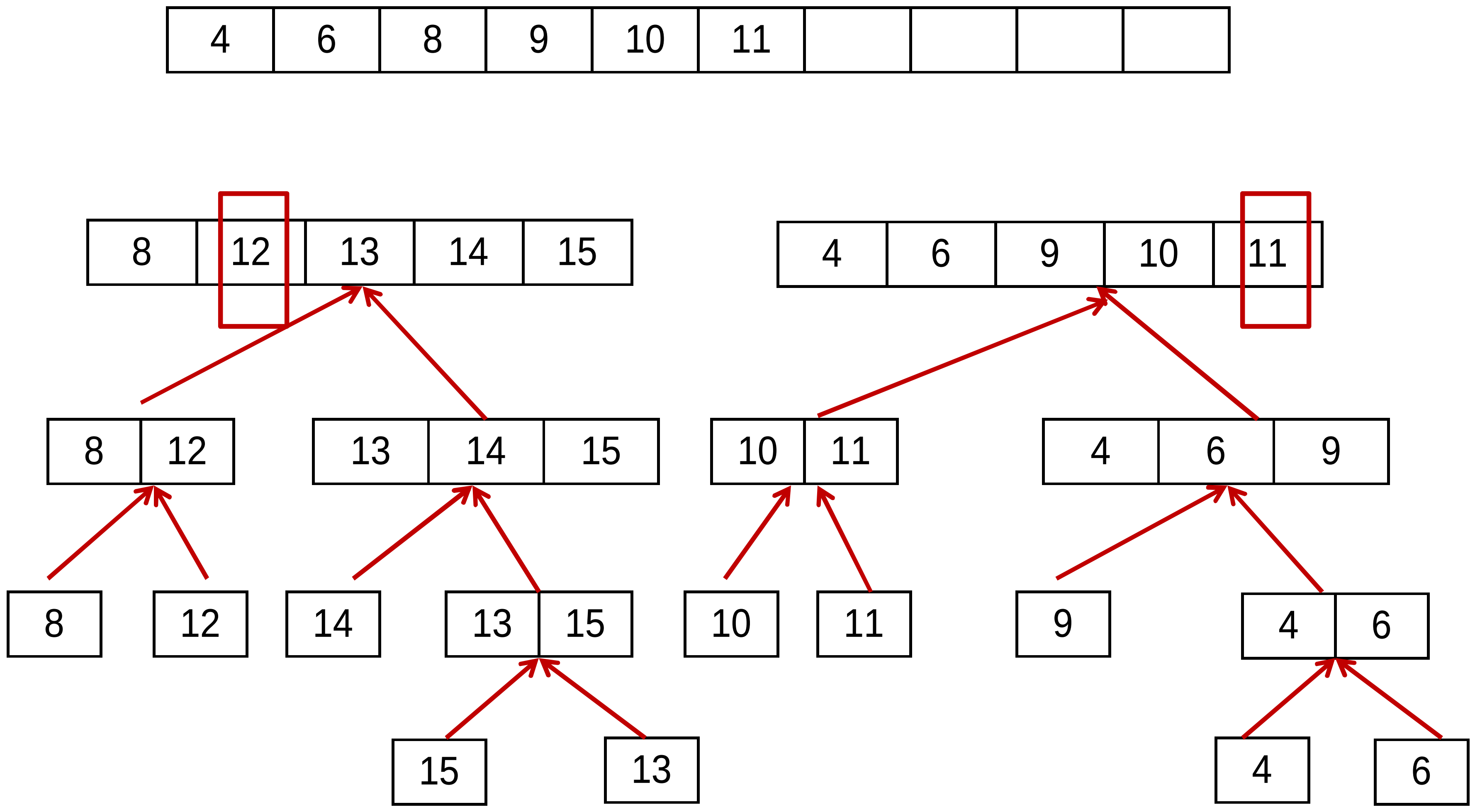
Merge Sort Illustration



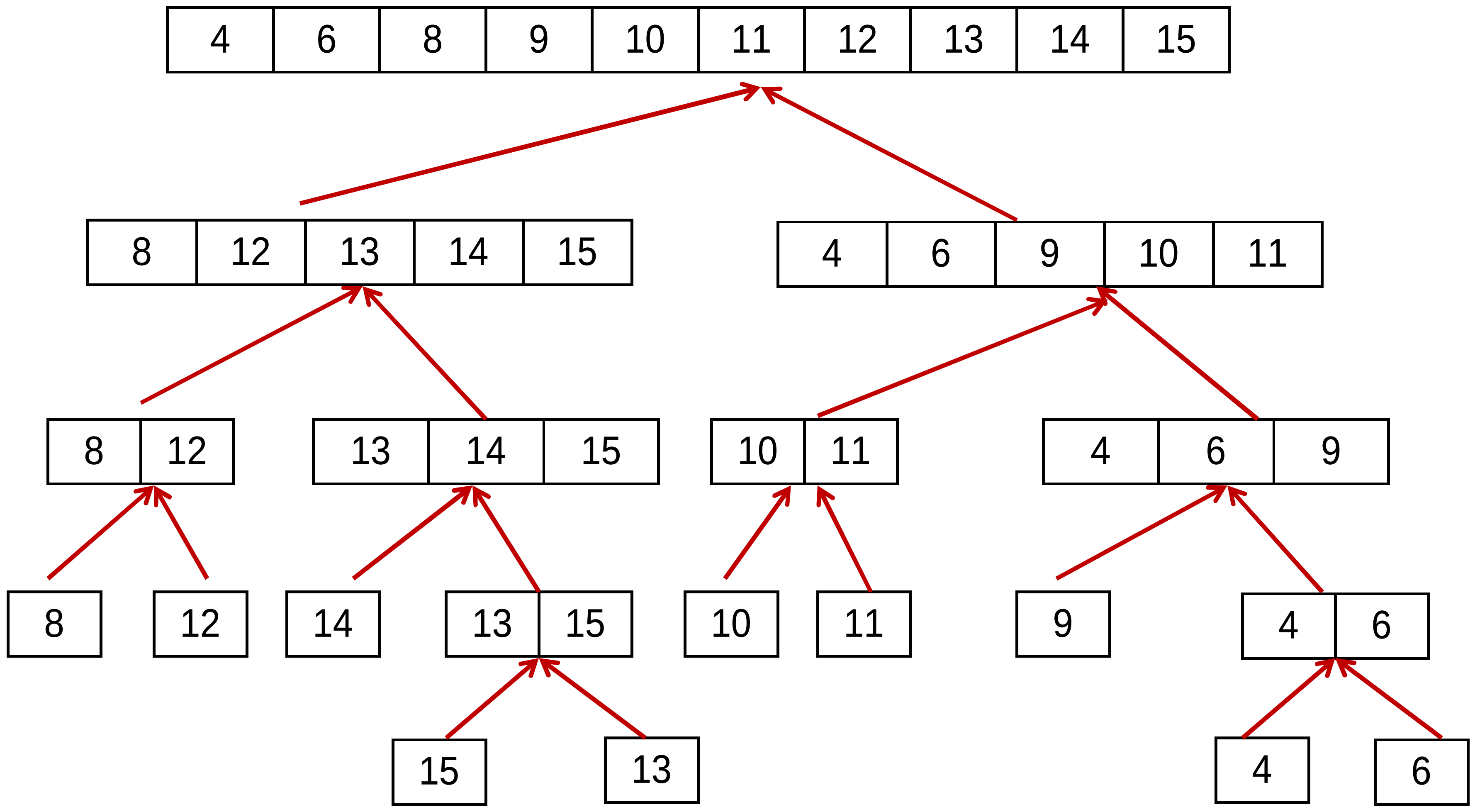
Merge Sort Illustration



Merge Sort Illustration



Merge Sort Illustration



Analysis of Merge Sort

Worst case:
Recurrence relation –

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

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4)+n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8)+n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\ldots\ldots\ldots\ldots \\ &= 2^kT(n/2^k) + kn \end{aligned}$$

Now,

Let $n/2^k = 1$, $n = 2^k$, $\log_2 n = k * \log_2 2$

i.e. $k = \log_2 n$

After substitution,

$T(n) = nT(1) + (\log_2 n)*n = n + n\log_2 n$

$= O(n\log_2 n)$

Analysis of Merge Sort

Worst case Time Complexity: O(nlogn)
Average case time complexity : O(nlogn)
Best case time complexity : O(nlong)
Auxiliary Space: O(n)
Algorithmic Paradigm: Divide and Conquer Approach
Sorting In Place: No
Stable: Yes

Summary

Name of Sorting Technique	Worst Case	Average Case	Best Case	Stable	in-Place
Bubble Sort	N^2	N^2	N	Yes	Yes
Selection Sort	N^2	N^2	N^2	No	Yes
Insertion Sort	N^2	N^2	N	Yes	Yes
Quick Sort	N^2	$N\log_2 N$	$N\log_2 N$	No	Yes
Merge Sort	$N\log_2 N$	$N\log_2 N$	$N\log_2 N$	Yes	No

Complete picture of sorting algorithms

- The ideal sorting algorithm would have the following properties:
  1. Stable: Equal keys should not reordered.
  2. Operates in place, requiring O(1) extra space.
  3. Worst-case O(n·log(n)) key comparisons.
  4. Worst-case O(n) swaps.
  5. Adaptive: Speeds up to O(n) when data is nearly sorted or when there are few unique keys.

- There is no algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application.

Criteria for selecting a sorting algorithm

1.**Size of the input** (for example: for small inputs, insertion sort is empirically faster then more advanced algorithms, though it takes O(n^2)).

2.**Location of the input** (sorting algorithms on disk are different from algorithms on RAM, because disk reads are much less efficient when not sequential. The algorithm which is usually used to sort on disk is a variation of merge-sort).

3.**How is the data distributed**? If the data is likely to be "almost sorted" - maybe a usually terrible bubble-sort can sort it in just 2-3 iterations and be super fast comparing to other algorithms.

Real World Sorting Algorithms

➢**Modern sorting algorithms use hybrid sorts that combine the best qualities of the different basic sorting algorithms.**

1. **Introsort (std::sort in C++) which runs quick sort and switches to heap sort when the recursion gets too deep. This way, you get the fast performance of quick sort in practice while guaranteeing a worst case O(nlogn) run time.**
2. **Timsort (used in Java and Python) is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data.**

Answer it

- Which of the following is not a stable sorting algorithm in its typical implementation.
  (A) Insertion Sort
  (B) Merge Sort
  (C) Quick Sort
  (D) Bubble Sort
- Which of the following sorting algorithms in its typical implementation gives best performance when applied on an array which is sorted or almost sorted (maximum 1 or two elements are misplaced).
- **(A)** Insertion Sort
  (B) Merge Sort
  (C) Quick Sort
  (D) Selection Sort

Answer it

- Which sorting algorithm will take the least time when all elements of input array are identical? Consider typical implementations of sorting algorithms.
  (A) Insertion Sort
  (B) Heap Sort
  (C) Merge Sort
  (D) Selection Sort

Answer it

- Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order, which of the following are TRUE ?
- I. Quicksort runs in $\Theta(n^2)$ time
- II. Bubblesort runs in $\Theta(n^2)$ time
- III. Mergesort runs in $\Theta(n)$ time
- IV. Insertion sort runs in $\Theta(n)$ time

Home work

Q1-Show all the iteration of insertion sort for the following array:

84,69,76,86,94,91

Q2- Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this:

2, 5, 1, 7, 9, 12, 11, 10

Which statement is correct?

(A) The pivot could be either the 7 or the 9.
(B) The pivot could be the 7, but it is not the 9
(C) The pivot is not the 7, but it could be the 9
(D) Neither the 7 nor the 9 is the pivot.

Q3-Let P be a QuickSort Program to sort numbers in ascending order using the first element as pivot. Let t1 and t2 be the number of comparisons made by P for the inputs {1, 2, 3, 4, 5} and {4, 1, 5, 3, 2} respectively. Which one of the following holds?

(A) t1 = 5 **(B)** t1 < t2

(C) t1 > t2 **(D)** t1 = t2