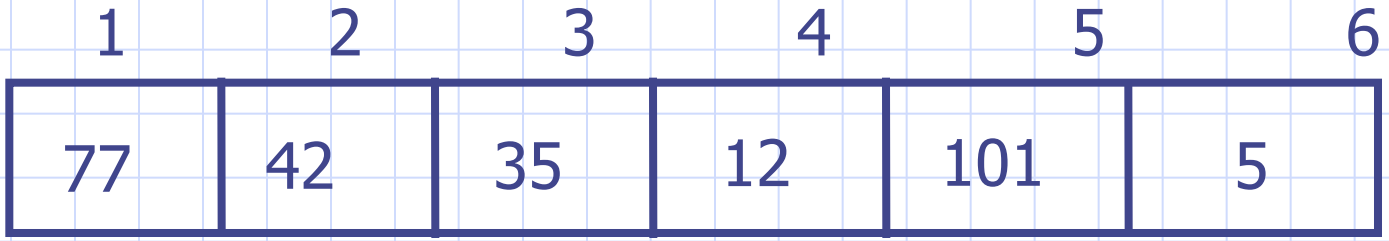
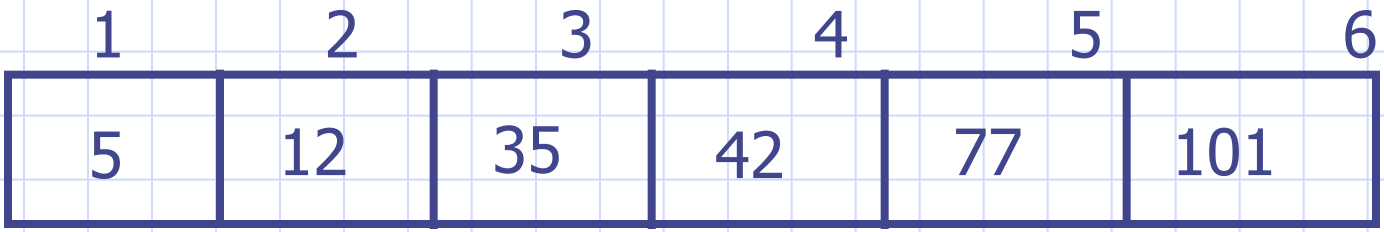
Sorting

Sorting

- Sorting takes an unordered collection and makes it an ordered one.





Internal and External Sorting

- External Sorting : If some of the records that it is sorting are in secondary memory.

  Eg: Telephone directory.

- Internal Sorting : If the records that it is sorting are in main memory.

  Eg: bubble sort, selection sort, Insertion sort, quick sort, merge sort, heap sort, radix sort.

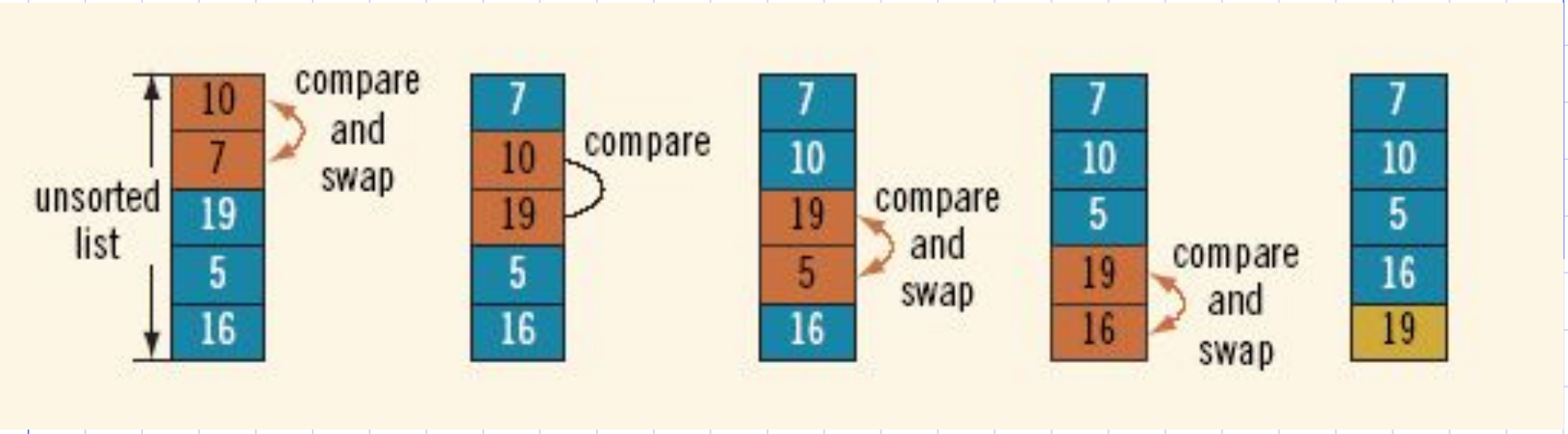
Bubble sort

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- Compare each element (except the last *two*) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places
- Compare each element (except the last *three*) with its neighbor to the right
  - Continue as above until you have no unsorted elements on the left

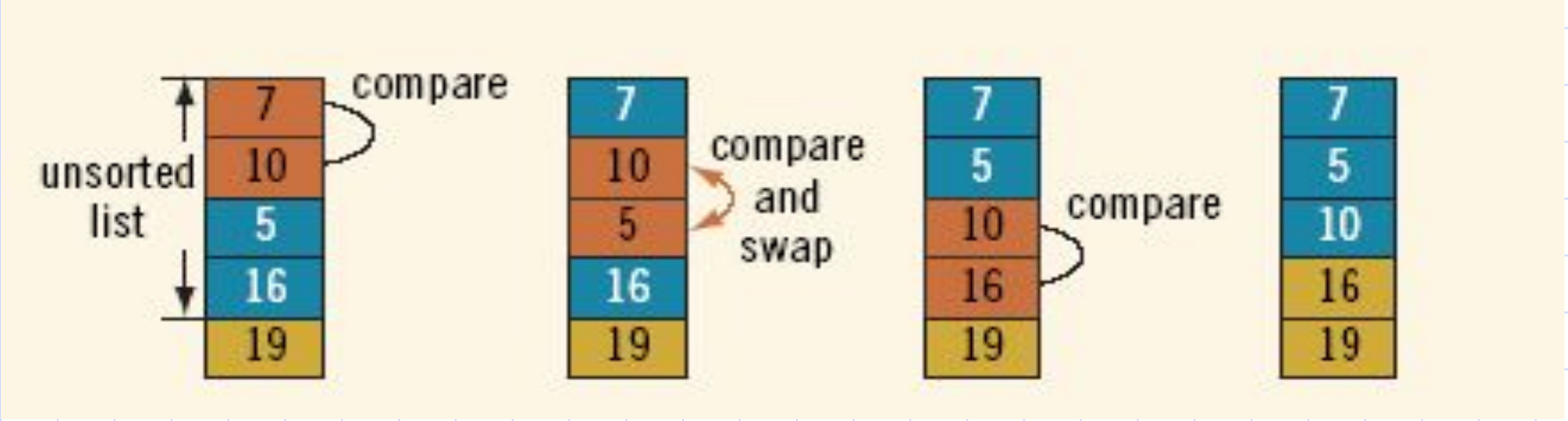
Bubble Sort working

- Bubble sort algorithm:
  - Suppose `list[0...N - 1]` is a list of n elements, indexed `0` to `N - 1`
  - We want to rearrange; that is, sort, the elements of list in increasing order
  - The bubble sort algorithm works as follows:
    - In a series of `N - 1` iterations, the successive elements, `list[index]` and `list[index + 1]` of list are compared
    - If `list[index]` is greater than `list[index + 1]`, then the elements `list[index]` and `list[index + 1]` are swapped, that is, interchanged

Bubble Sort working (Cont'd)



Elements of array *list* during the first iteration



Elements of array *list* during the second iteration

Bubble Sort working (Cont'd)

Elements of array *list* during the third iteration

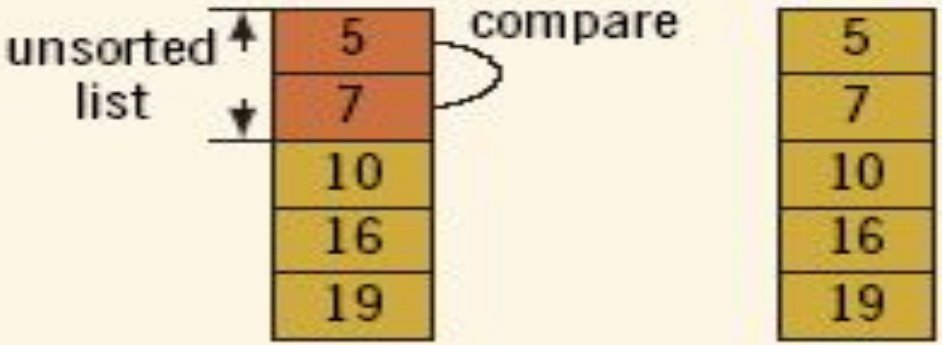


Figure 16: Elements of array *list* during the fourth iteration

FIGURE 10.1
One Pass of Bubble Sort

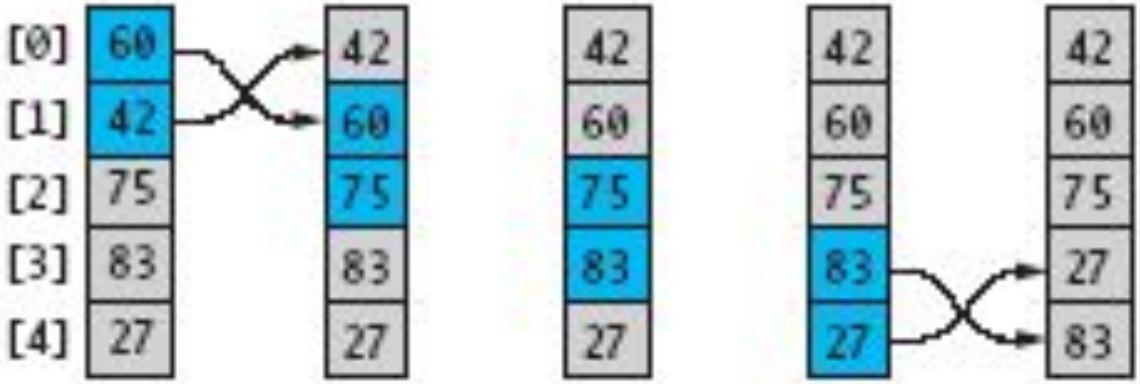
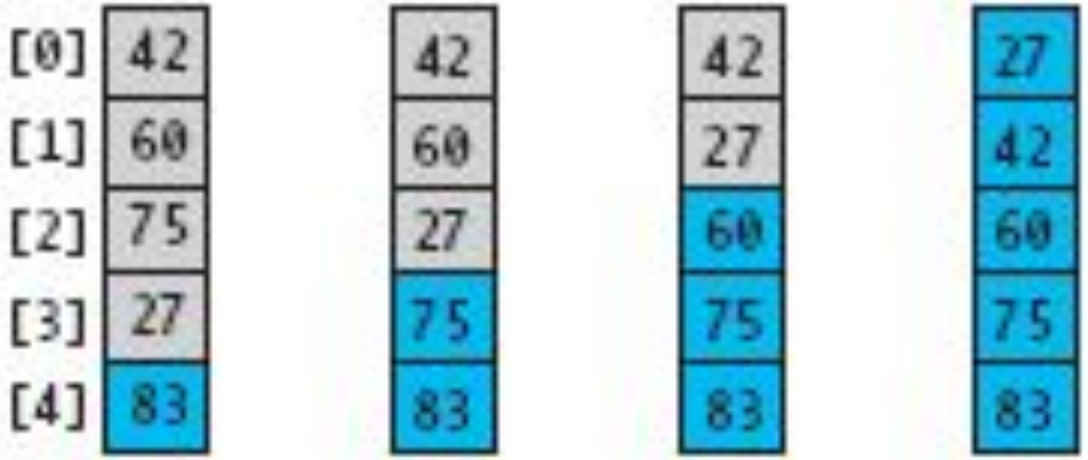


FIGURE 10.2
Array After Completion of Each Pass



Algorithm of Bubble sort

Bubble_sort(n,list)

Where list = Represents the list of elements

n = Represents the number of elements in the list

Step 1: [Initialize]

i=1

Step 2: Repeat through step 5 while (i<=n-1)

Step 3: j=1

Step 4: Repeat through step 5 while (j<=n-i-1)

Step 5: if list[j] > list[j+1]

(i) temp=list[j]

(ii) list[j] = list[j+1]

(iii) list[j+1]=temp

Step 6: Exit

bubblesort

Bubble Sort Algorithm (Cont'd)

- At every iteration largest element in its proper position.
- i.e all element in position >=N- I are place in proper position after iteration i.
- It is known that for a list of length *N*,
- bubble sort makes $N(N-1)/2$ key comparisons.

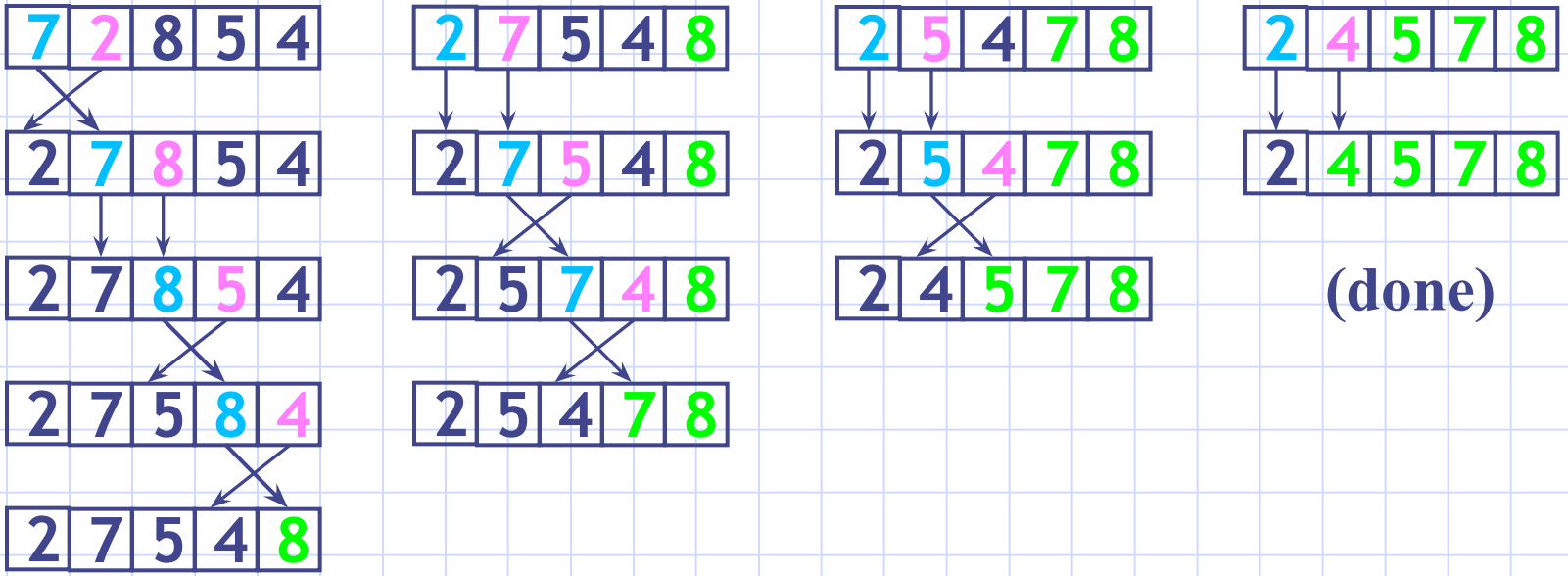
  Total number of comparisons is given by

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

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

- Therefore, if $N = 100$, then to sort the list bubble sort makes about 5,000 key comparisons.

Example of bubble sort



Analysis of Bubble Sort

- Excellent performance *in some cases*
  - But *very poor* performance in others!
- Works ***best*** when array is nearly sorted to begin with
- *Best case* occurs when the array is already sorted:
  - O(n) comparisons : and that means that no swap occurred and only 1 iteration of n elements
  - O(1) exchanges (none actually)
-

- The worst case
  - if the array is already sorted but in descending order. This means that in the first iteration it would have to look at n elements,
  - then second iteration t would look n - 1
  - so on and so forth till 1 comparison
  - Big-O = n + n - 1 + n - 2 ... + 1 = (n*(n -1))/2 = $O(n^2)$
  - so Worst case number of comparisons: $O(n^2)$
  - and Worst case number of exchanges: $O(n^2)$

Selection sort

- Given an array of length n,
  - Search elements 0 through n-1 and select the smallest
    - Swap it with the element in location 0
  - Search elements 1 through n-1 and select the smallest
    - Swap it with the element in location 1
  - Search elements 2 through n-1 and select the smallest
    - Swap it with the element in location 2
  - Search elements 3 through n-1 and select the smallest
    - Swap it with the element in location 3
  - Continue in this fashion until there's nothing left to search

Selection Sort

- List is sorted by selecting list element and moving it to its proper position.
- Algorithm finds position of smallest element and moves it to top of unsorted portion of list.
- Repeats process above until entire list is sorted
- First interation make n-1 comparison and second iteration makes n-2 comparison ..so on.. So it s time complexity is O(n2) .. So its inefficient to e used on large list.
- It is 60% more efficient than bubble sort.

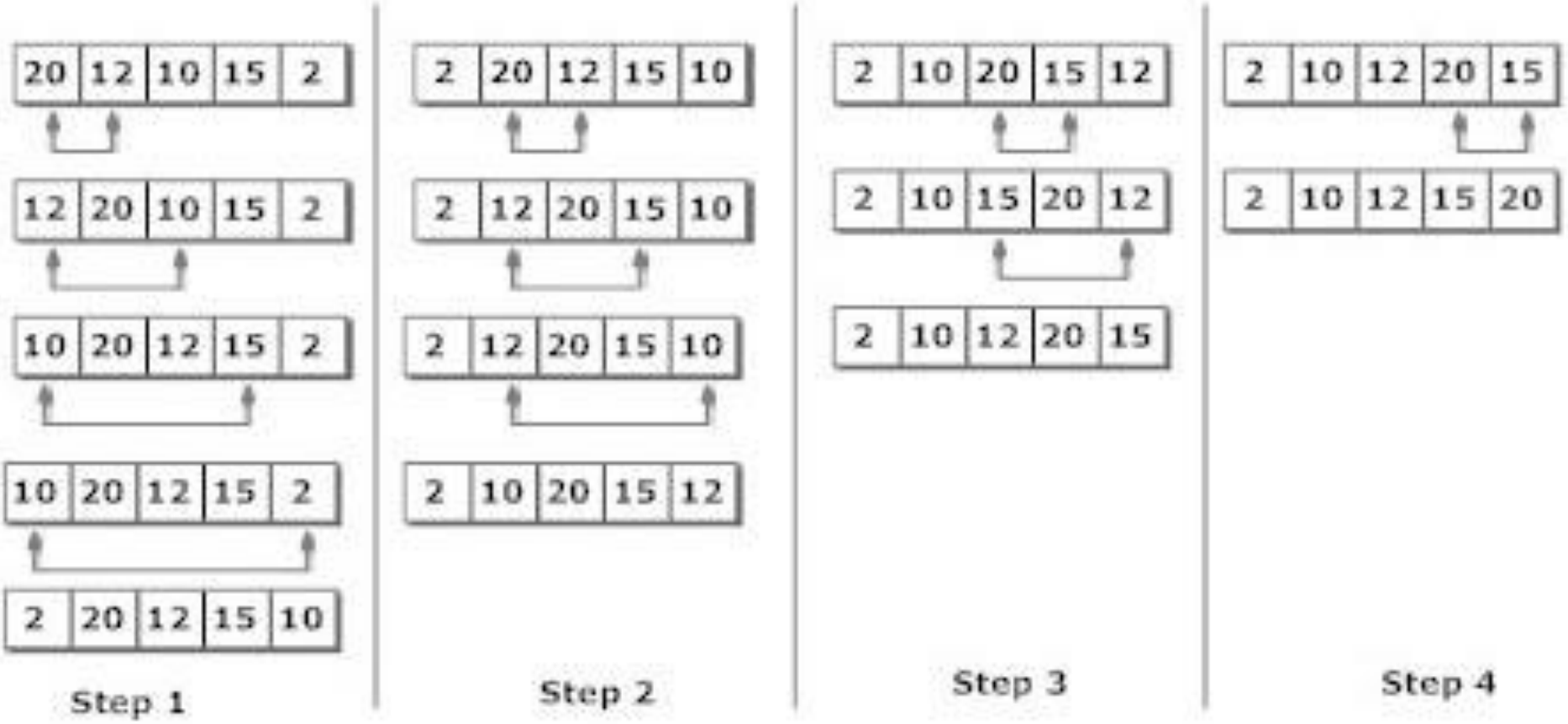


Figure: Selection Sort

Selection Sort Algorithm (Cont'd)

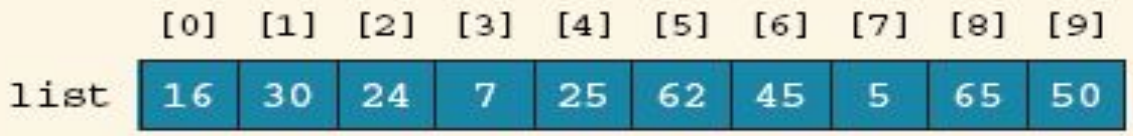


Figure 1: An array of 10 elements

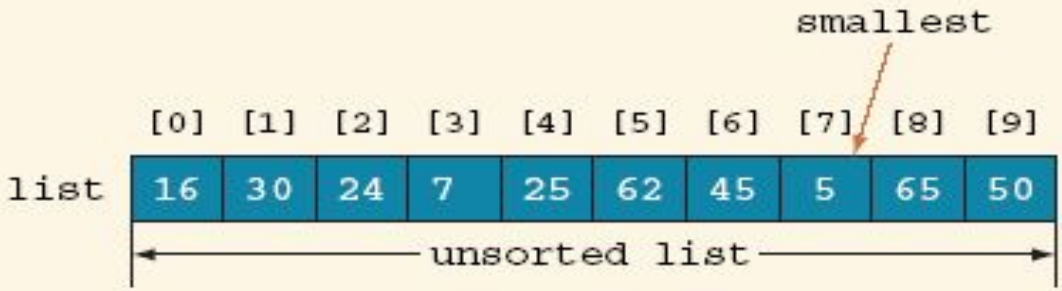


Figure 2: Smallest element of unsorted array

Selection Sort Algorithm (Cont'd)

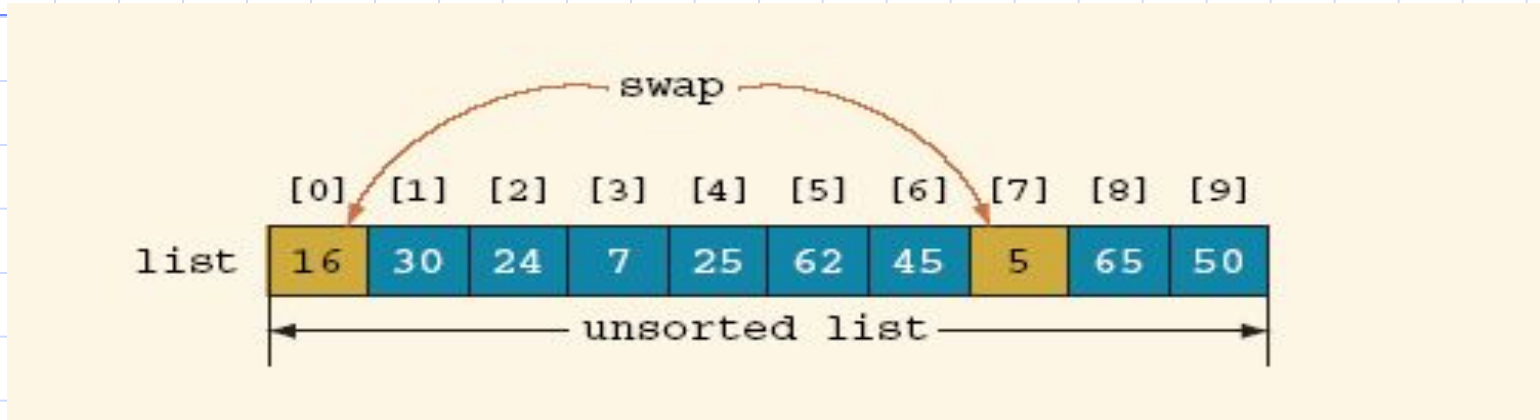
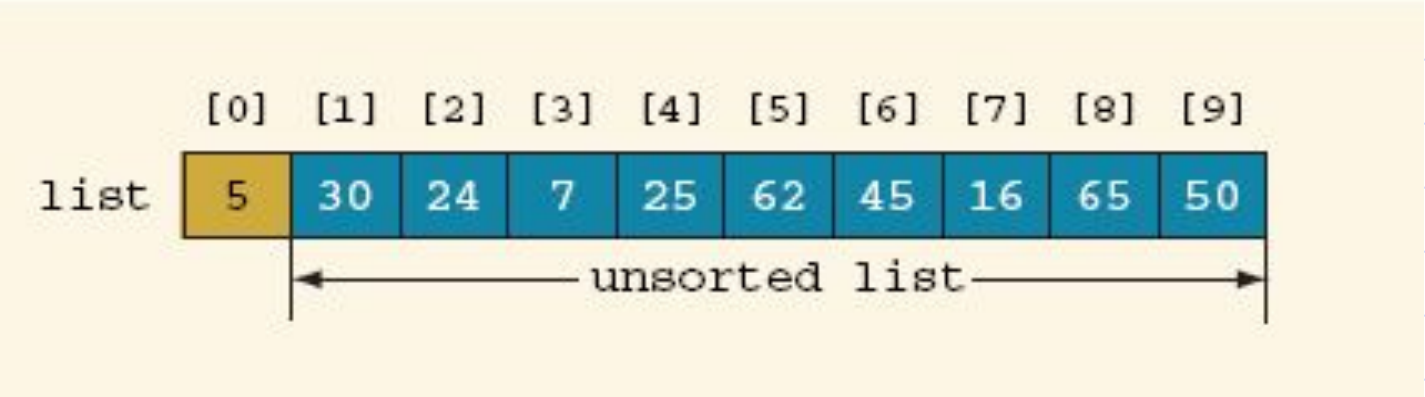


Figure 3: Swap elements *list[0]* and *list[7]*



Array after swapping *list[0]* and *list[7]*

Selection Sort Algorithm (Cont'd)

- It is known that for a list of length n, on an average selection sort makes $n(n-1)/2$ key comparisons.
  Total number of comparisons is given by
  $(n-1)+(n-2)+(n-3)+\ldots+3+2+1$
  $= \sum(n-1) = n(n-1)/2$

- Therefore, if $n = 1000$, then to sort the list selection sort makes about 500,000 key comparisons.

Algorithm of Selection sort

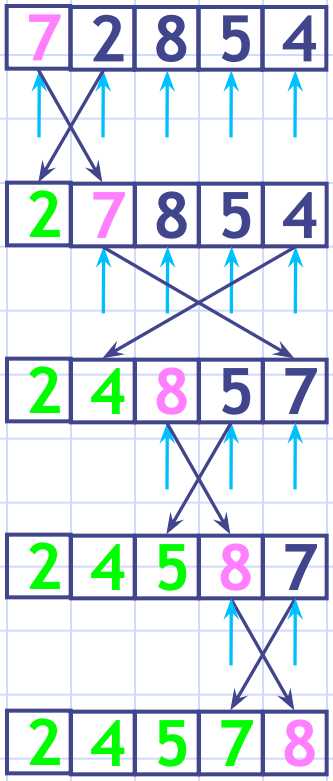
Selection_sort(array,size)

Where array = Represents the list of elements
size = Represents the size of the list

Step 1: [Initialize]
current = 0
Step 2: Repeat through step 7 while (current < size-1)
Step 3: j=current +1
Step 4: Repeat through step 6 while (j<size)
Step 5: if (array[current] > array[j]
(i) temp=array[current]
(ii) array[current] = array[j]
(iii) array[j]=temp
Step 6: j=j+1
Step 7: current=current+1
Step 8: Exit

selectionsort

Example and analysis of selection sort



- The selection sort might swap an array element with itself--this is harmless, and not worth checking for
- Analysis:
  - The outer loop executes n-1 times
  - The inner loop executes about n/2 times on average (from n to 2 times)
  - Work done in the inner loop is constant (swap two array elements)
  - Time required is roughly (n-1)*(n/2)
  - You should recognize this as $O(n^2)$

Insertion sort

- Finding the element's proper place
- Making room for the inserted element (by shifting over other elements)
- Inserting the element

- Techniques for insertion sort
- Step 1: The second element of an array is compared with the elements that appears before it (only first element in this case).
- If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.

Insertion sort Technique

- Step 2: The third element of an array is compared with the elements that appears before it (first and second element).
- If third element is smaller than first element, it is inserted in the position of first element.
- If third element is larger than first element but, smaller than second element, it is inserted in the position of second element.
- If third element is larger than both the elements, it is kept in the position as it is.
- After second step, first three elements of an array will be sorted.

Insertion Sort

- Based on technique of card players to arrange a hand
  - Player keeps cards picked up so far in *sorted order*
  - When the player picks up a new card
    - Makes room for the new card
    - Then inserts it in its proper place

FIGURE 10.3
Picking Up a Hand of Cards

FIGURE 10.5
Inserting the Fourth Array Element

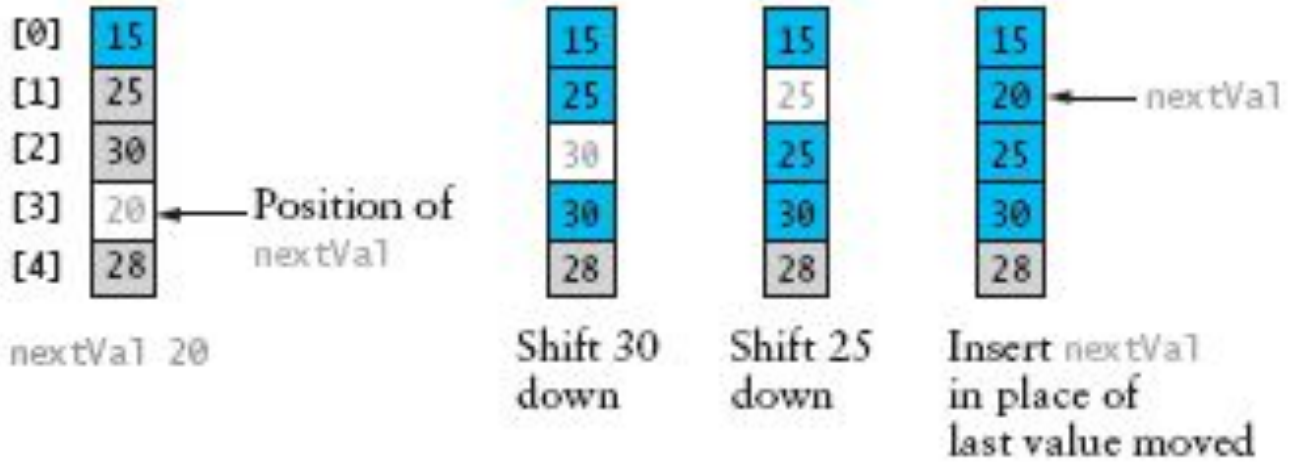
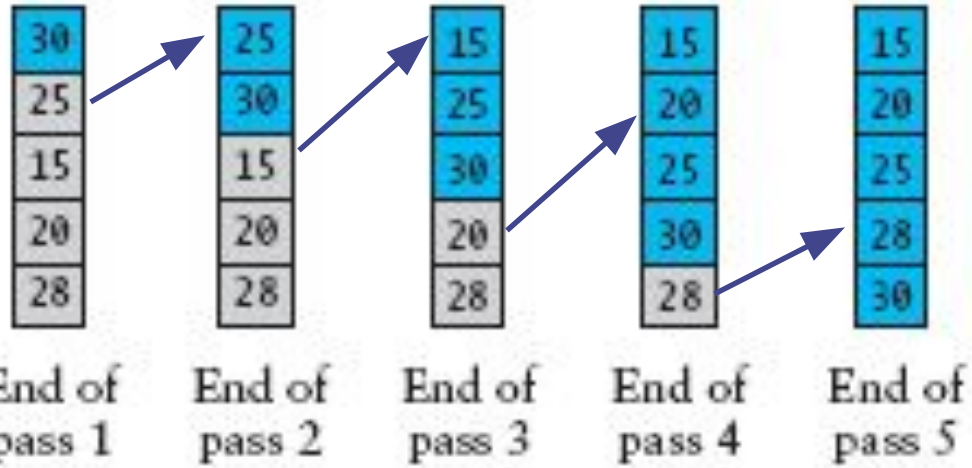
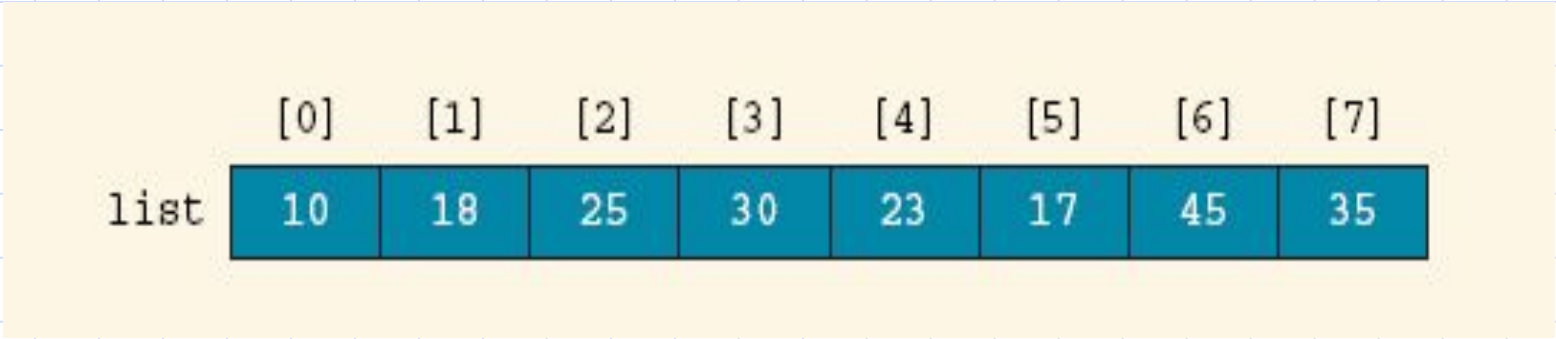


FIGURE 10.4
An Insertion Sort



Insertion Sort

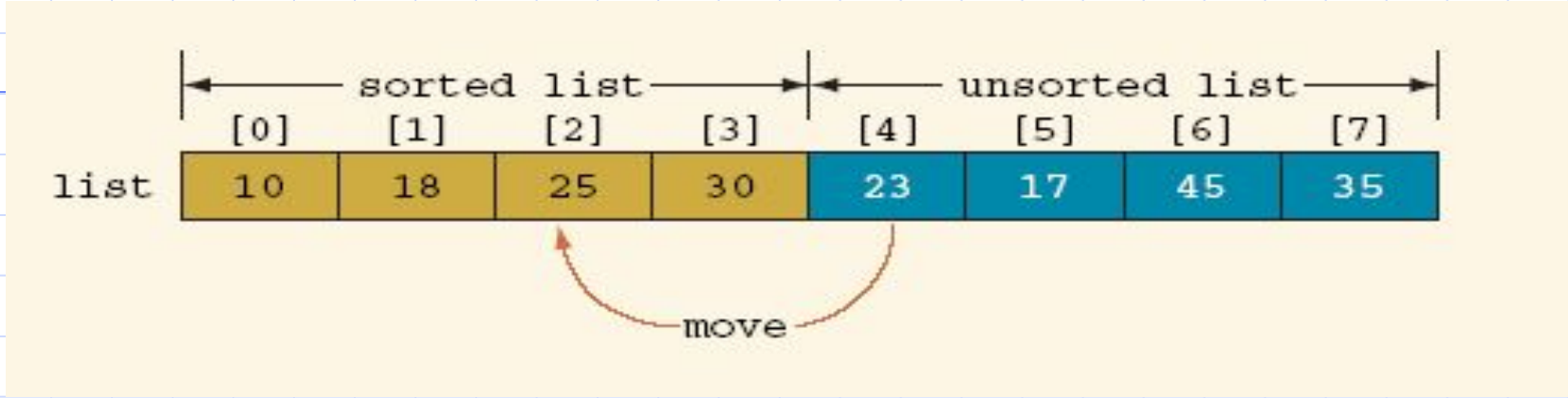
The insertion sort algorithm sorts the list by moving each element to its proper place



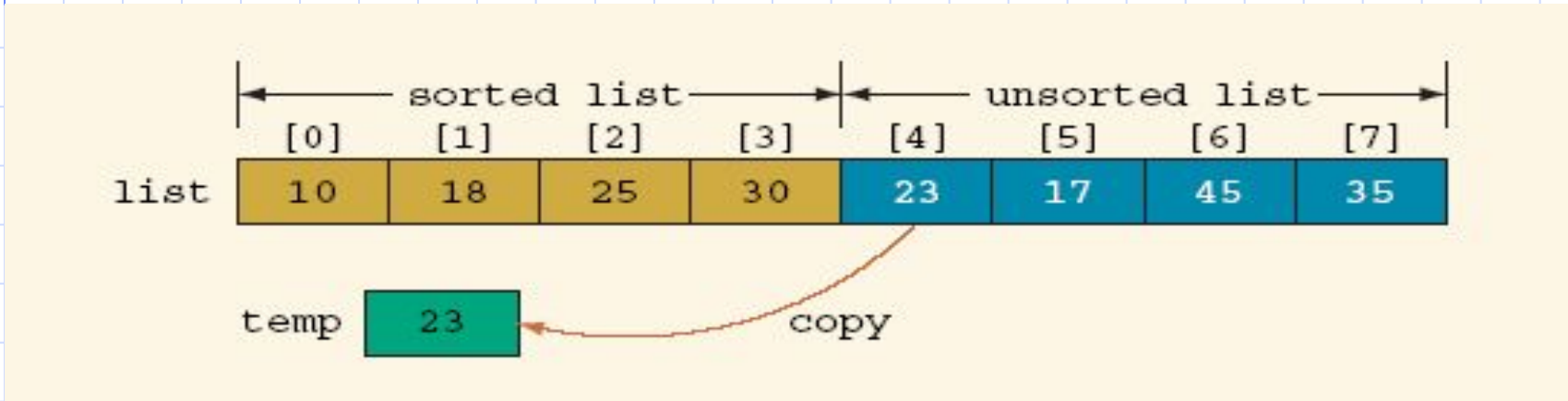
Array *list* to be sorted

Sorted and unsorted portions of the array *list*

Insertion Sort

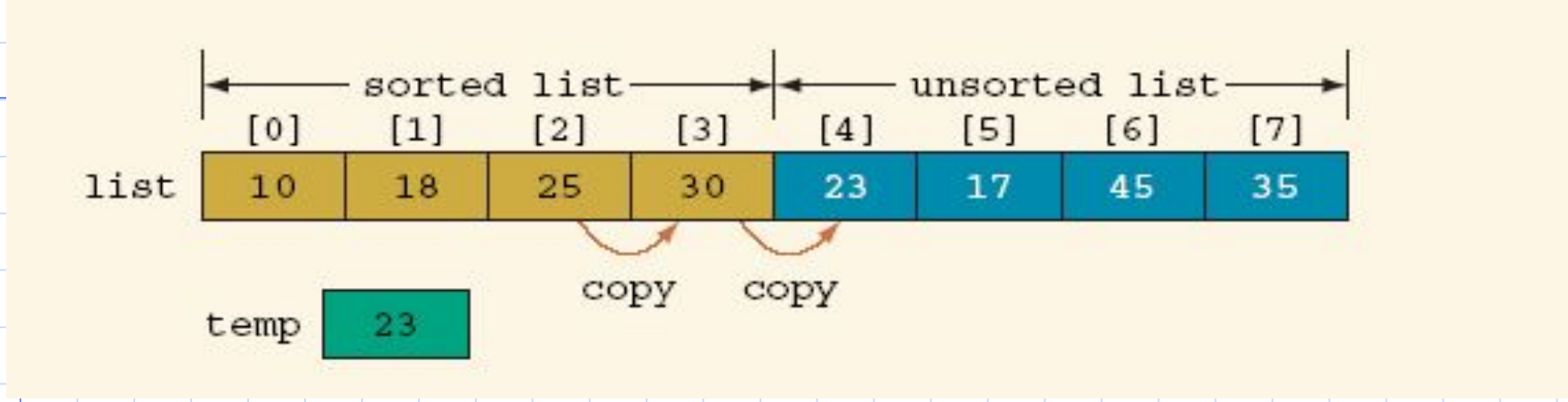


Move *list[4]* into *list[2]*



Copy *list[4]* into *temp*

Insertion Sort



Array *list* before copying *list[3]* into *list[4]*, then *list[2]* into *list[3]*

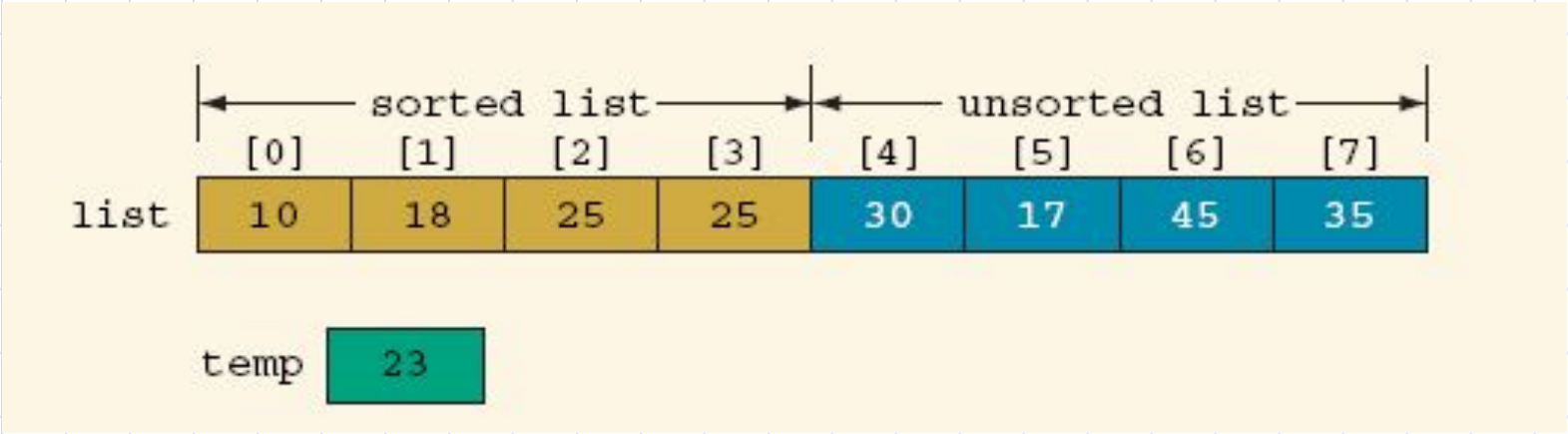
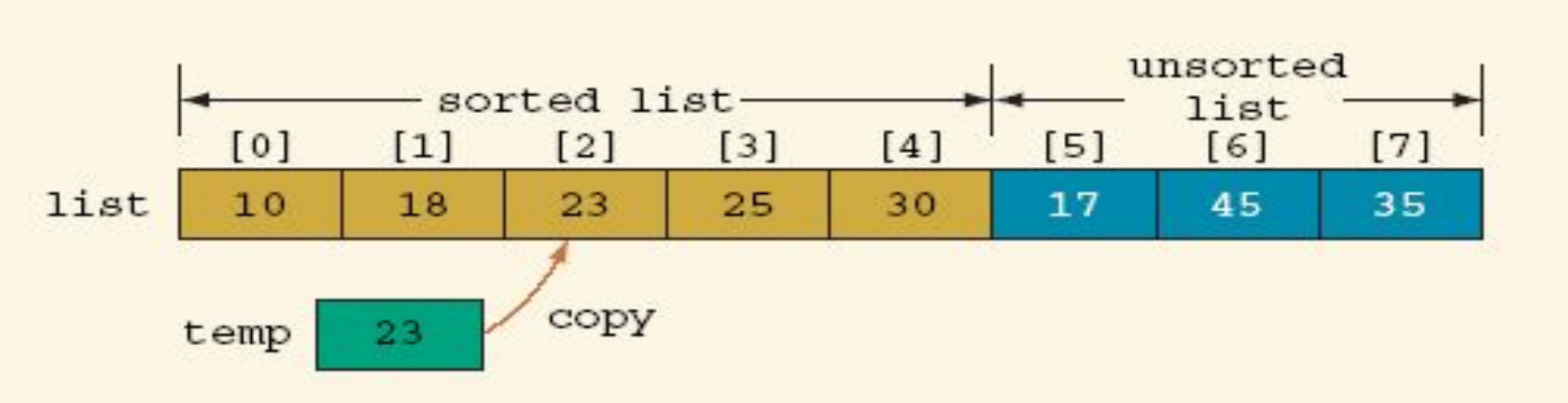


Figure 11: Array *list* after copying *list[3]* into *list[4]*, and then *list[2]* into *list[3]*

Insertion Sort



Array *list* after copying *temp* into *list[2]*

Algorithm of Insertion sort

Insertion_sort(x, n)

Where x= Represents the list of elements
n = Represents the number of elements in the list

Step 1: [Initially]
i=1
Step 2: Repeat through step 3 through 5 for i<n
Step 3: (i) key=x[i]
(ii) j=i-1

- /* * step 4 : Move elements of arr[0..i-1], that are
- greater than key, to one position ahead
- of their current position */

Step 4: Repeat step through 4 for (j>=0 && x[i] > key)
(i) x[j+1] = x[j]
(ii) j=j- 1

Analysis of insertion sort

- We run once through the outer loop, inserting each of n elements; this is a factor of n
- On average, there are n/2 elements already sorted
- Hence, the time required for an insertion sort of an array of n elements is proportional to $n^2/4$
- Worst case : array is sorted in reverse order. So complexity is $O(n^2)$: $O(n^2)$ comparisions.
- Best case : array is already sorted so complexity is O(n) : O(n) comparisons.

Summary

- Bubble sort, selection sort, and insertion sort are all $O(n^2)$
- Within $O(n^2)$,
  - Bubble sort is very slow, and should probably never be used for anything

Summary

- Selection sort is intermediate in speed
- Insertion sort is usually the fastest of the three--in fact, for small arrays (say, 10 or 15 elements), insertion sort is faster than more complicated sorting algorithms

- Selection sort and insertion sort are “good enough” for small arrays

Divide and Conquer

- Divide and Conquer cuts the problem in half each time, but uses the result of both halves:
  - cut the problem in half until the problem is trivial
  - solve for both halves
  - combine the solutions

Divide and Conquer - Sort

- Base case

  single element (n=1), return

- Divide A into two subarrays: FirstPart, SecondPart

  Two Subproblems:

  - sort the FirstPart
  - sort the SecondPart

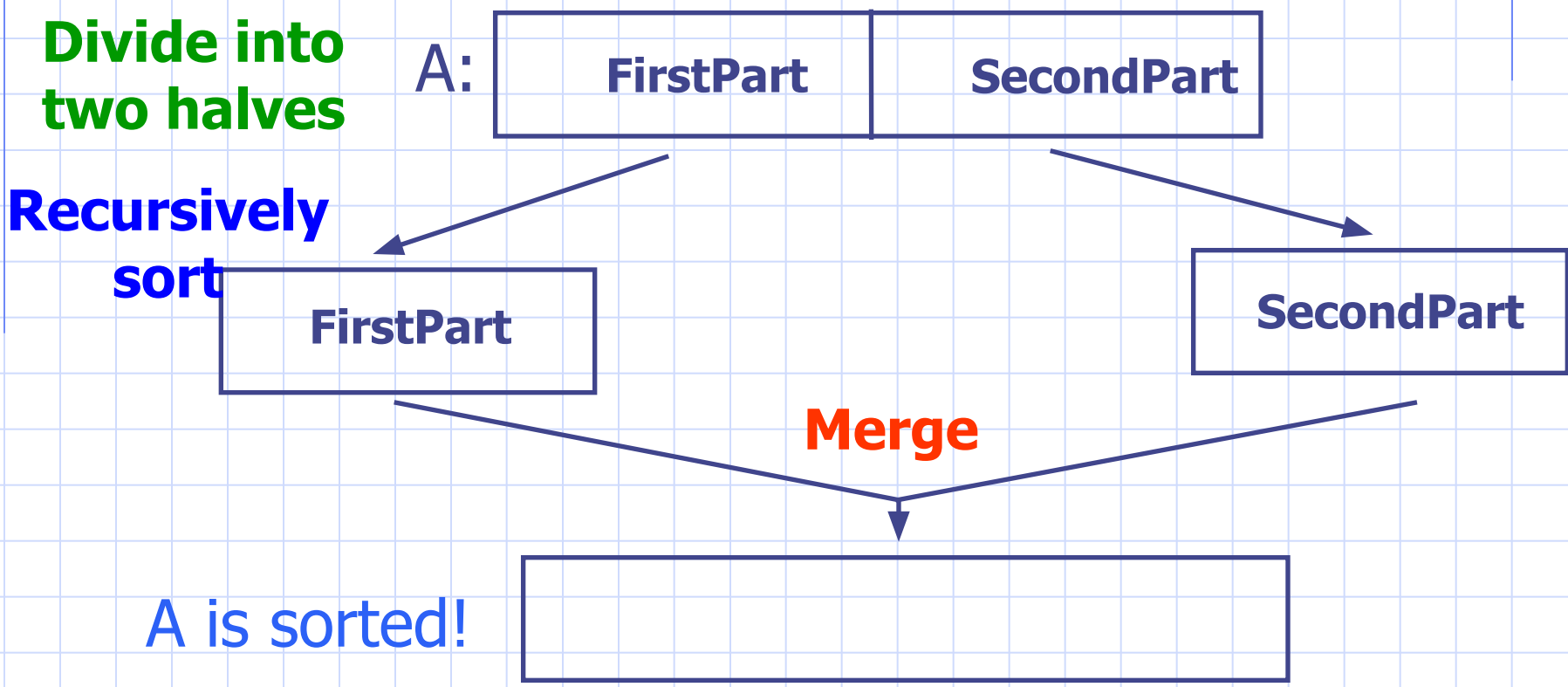
- Recursively

  - sort FirstPart
  - sort SecondPart

- Combine sorted FirstPart and sorted second part

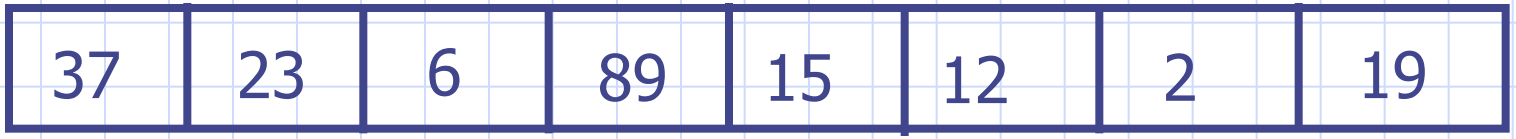
Merge Sort: Idea

Divide into two halves



Mergesort

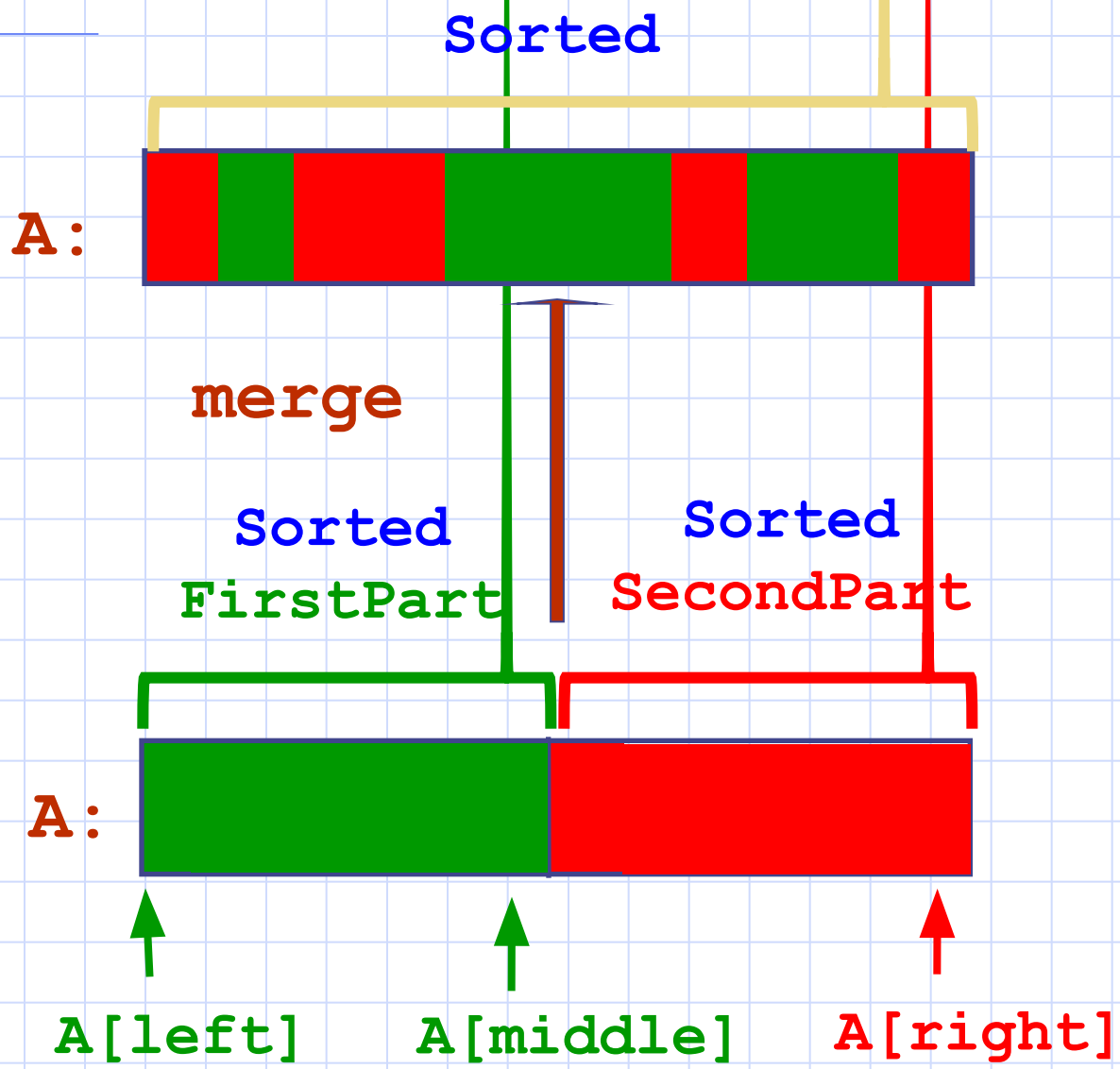
- A divide-and-conquer algorithm:
- Divide the unsorted array into 2 halves until the sub-arrays only contain one element
- Merge the sub-problem solutions together:
  - Compare the sub-array's first elements
  - Remove the smallest element and put it into the result array
  - Continue the process until all elements have been put into the result array



Merge working

- Merge working
  - Access the first item from both sequences
  - While not finished with either sequence
    - Compare the current items from the two sequences, copy the smaller current item to the output sequence, and access the next item from the input sequence whose item was copied
  - Copy any remaining items from the first sequence to the output sequence
  - Copy any remaining items from the second sequence to the output sequence

Merge-Sort: Merge



```
Mergesort(Passed an array)
   if array size > 1
      Divide array in half
      Call Mergesort on first half.
      Call Mergesort on second half.
      Merge two halves.

Merge(Passed two arrays)
   Compare leading element in each array
   Select lower and place in new array.
      (If one input array is empty then place
       remainder of other array in output array)
```

Recursive merge sort Algorithm

merge_sort (a, start, end)

Step 1:

if start<end

(i) mid = (start+end)/2;

(ii) merge_sort(a,start,mid);

(iii) merge_sort(a,mid+1,end);

(iv) merge(a,start,mid,end);

Step 2: Return

- Merge algorithm(a,beg,mid,end)

Resut[] is for final sorted array

K is index for result array

Step 1 :Step1: [Initialize]

i=start j=mid+1 k=0

Step 2: Repeat setp 4 while ((i<=mid) and (j<=end)

Step 3: if (list_a[i] < list_a[j])

(i) result[k]=list_a[i]

(ii) i=i+1

else if (list_a[i] > list_a[j])

(iv) result[k]=list_a[j]

(v) j=j+1

else // elements of both the list are equal and insert only one of them in result[]

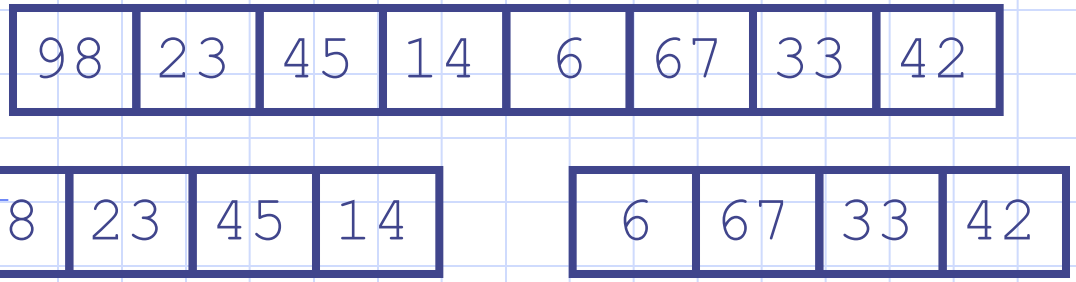
(vii) result[k]=list_a[i] (viii) i = i+1

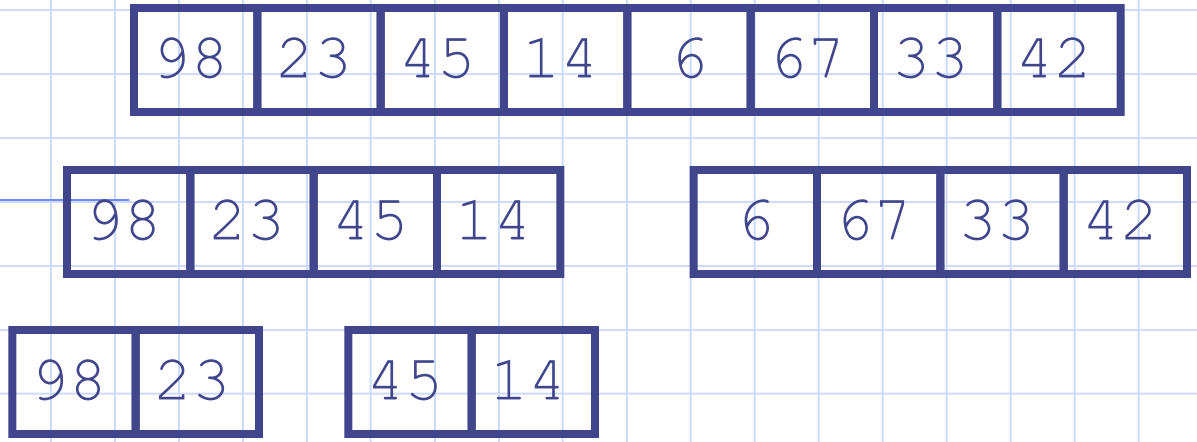
(ix) j= j+1

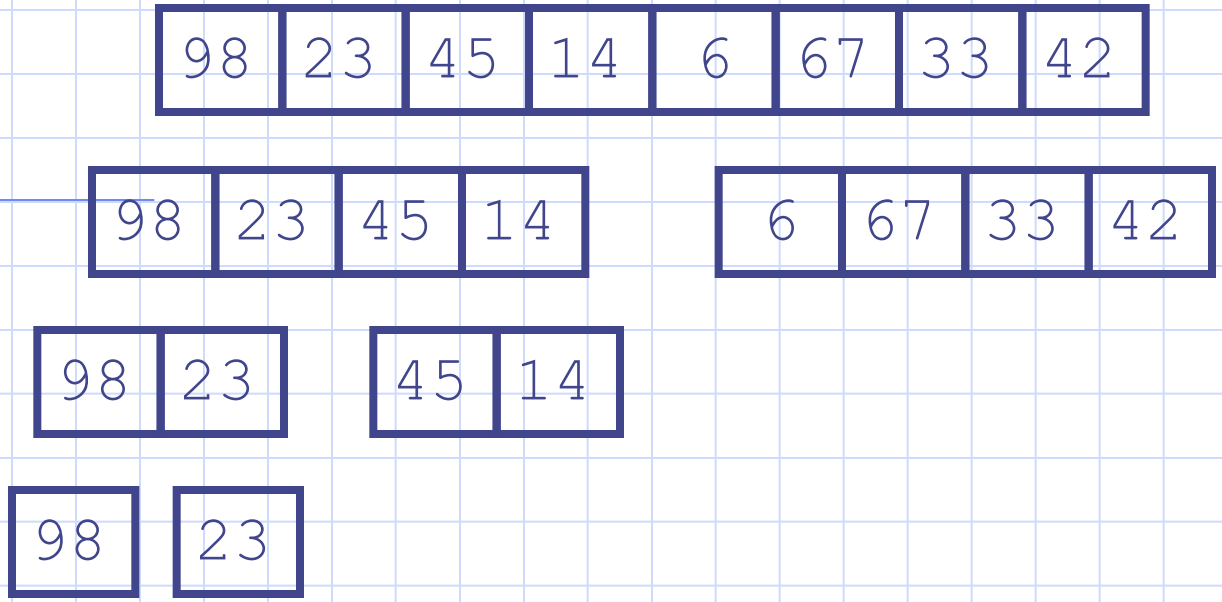
End of IF-ELSE

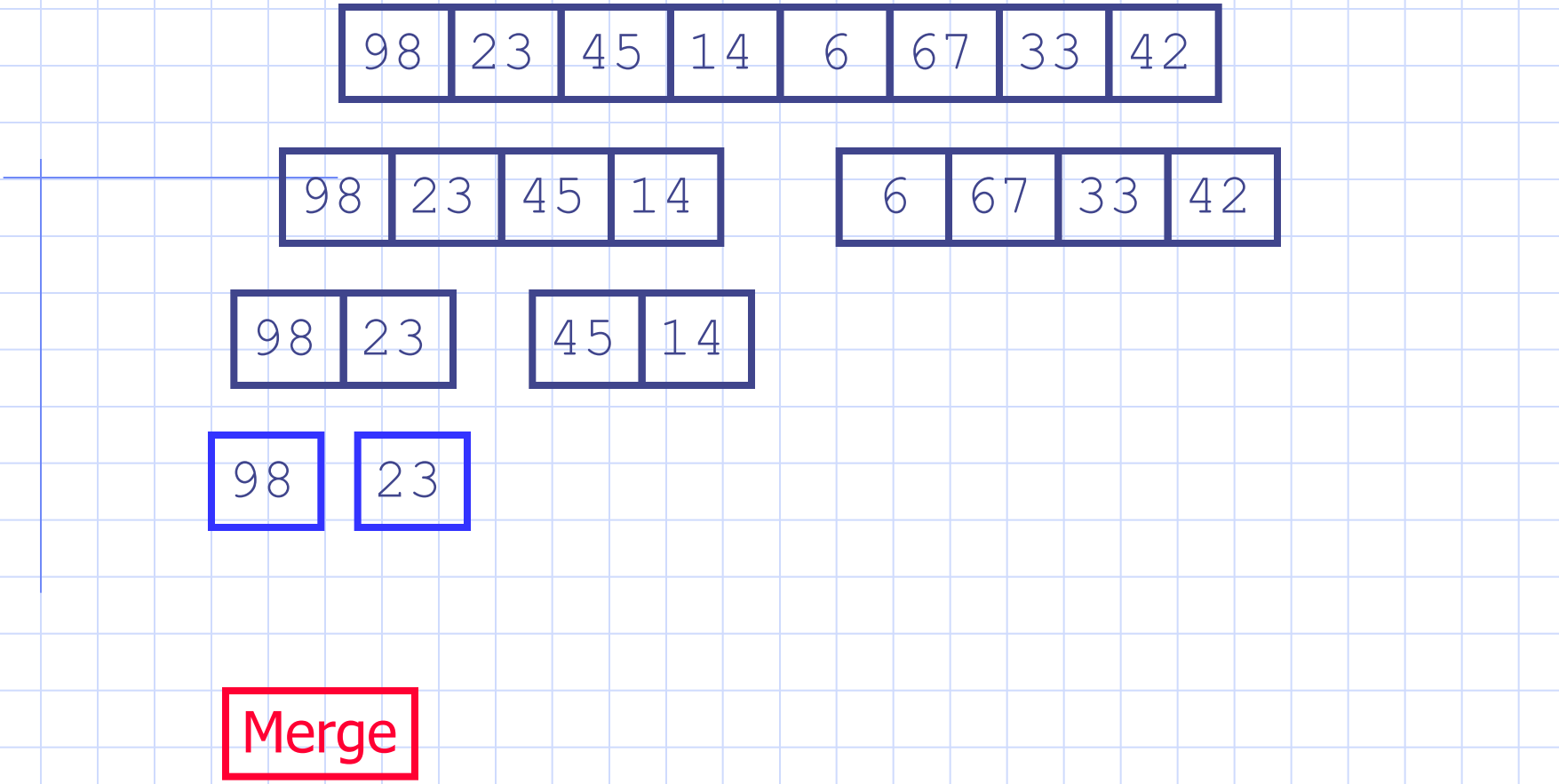
Step 5: if(i > mid) [copy the element of right su-array]
Step 6: Repeat through step 6 for j<=End
    (i) result[k]=list_a[j]
    (ii) j=j+1
    (iii) k=k+1
Step 7: [copy of element of left subarray if left subarray are still reamining]
    else
Step 8: Repeat through step 8 for i<=mid
    (i) result[k]=list_a[i]
    (ii) i=i+1
    (iii) k=k+1
Step 9: Return

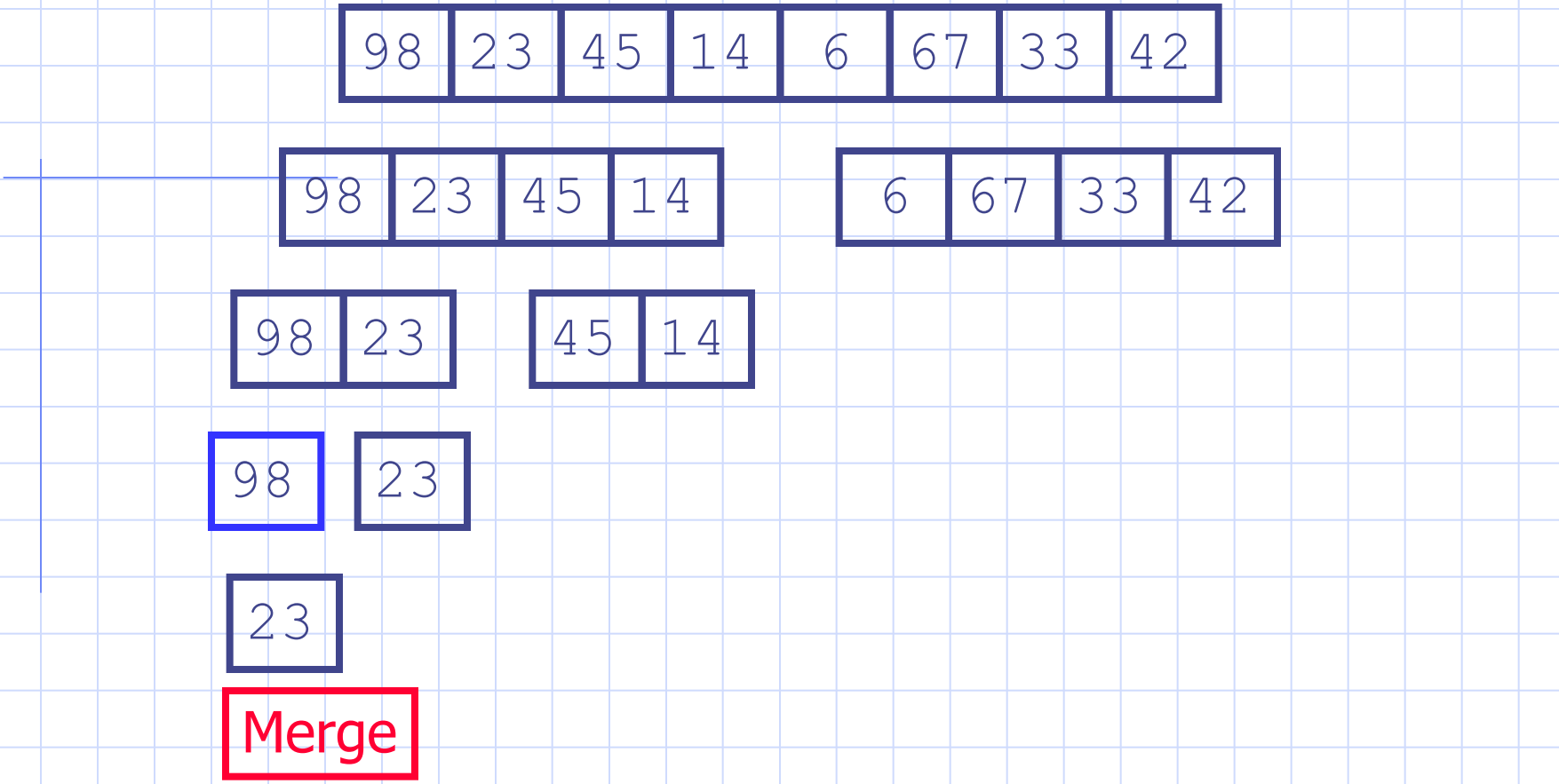

98
23
45
14
6
67
33
42

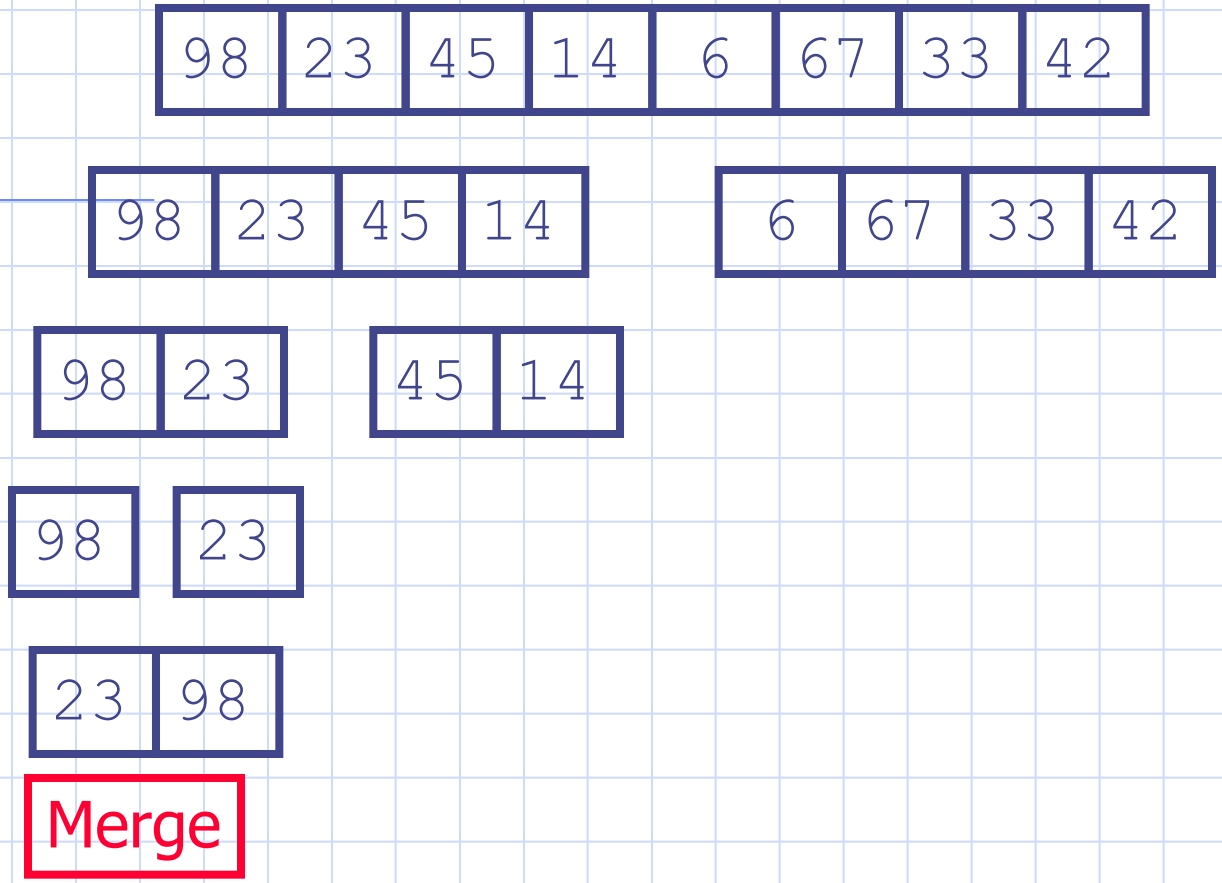

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42

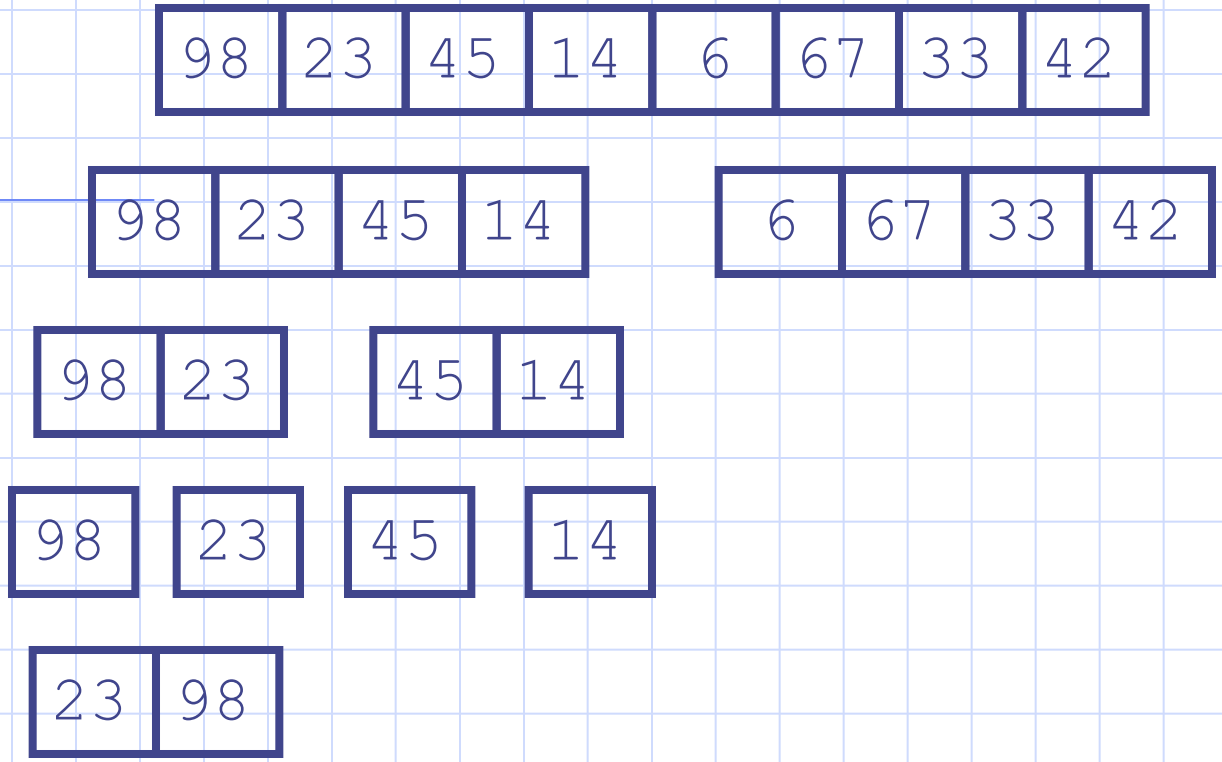

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

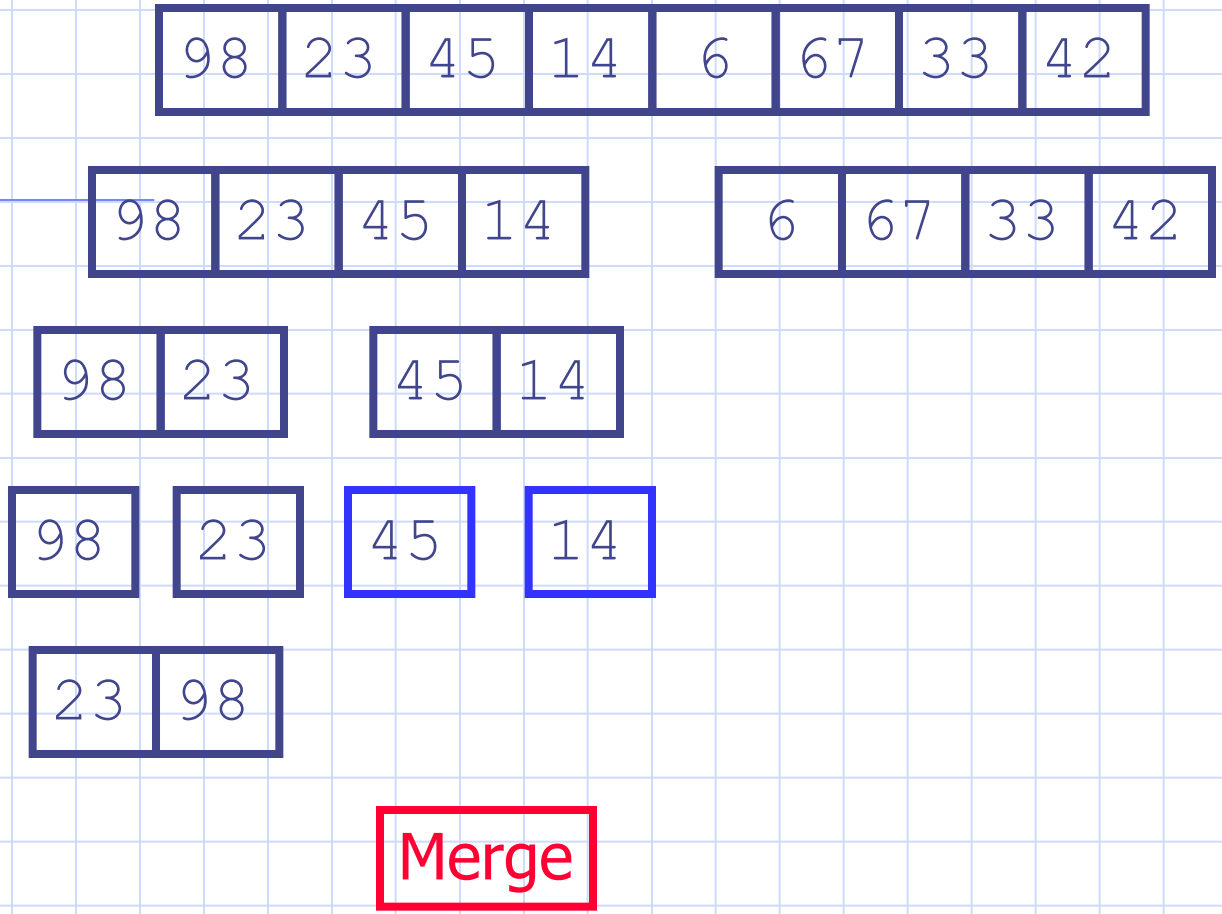

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

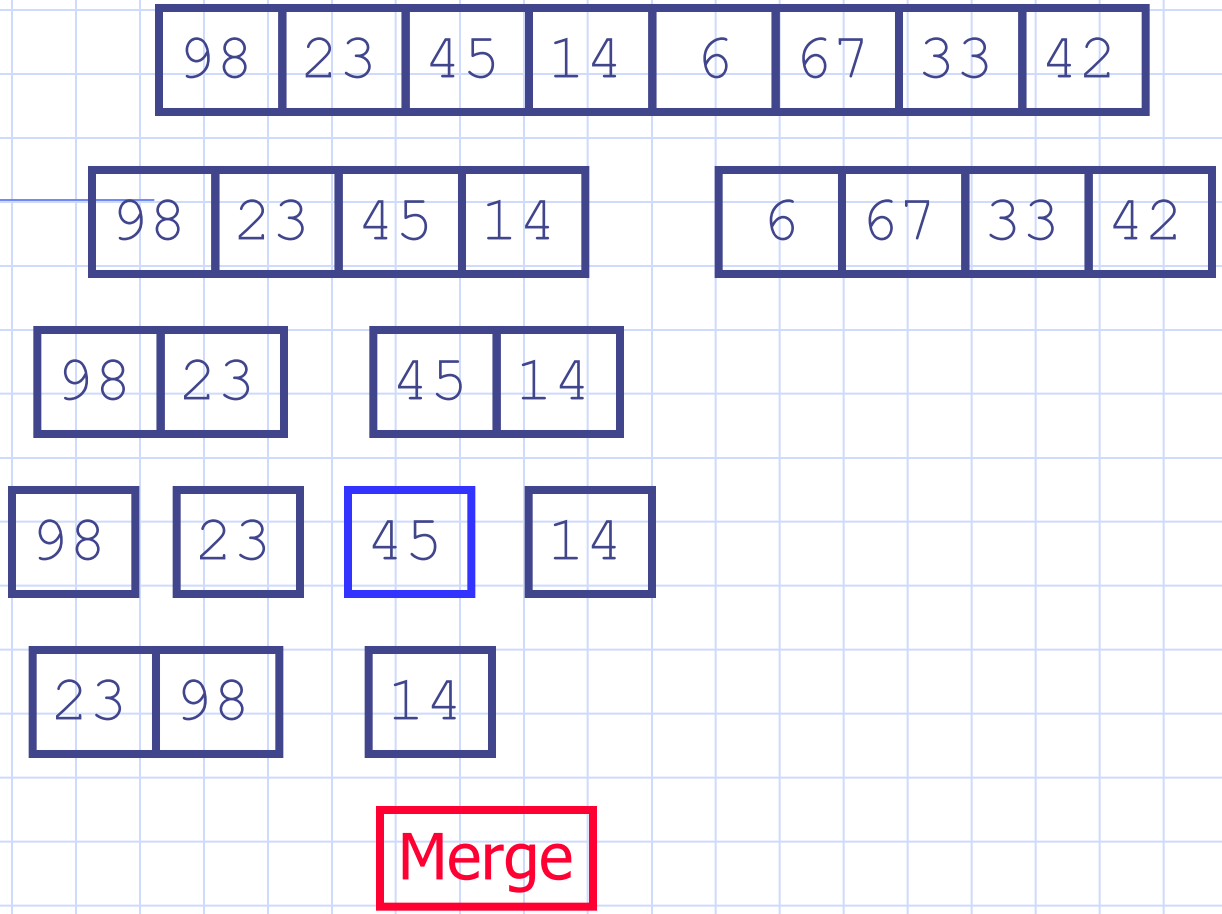

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

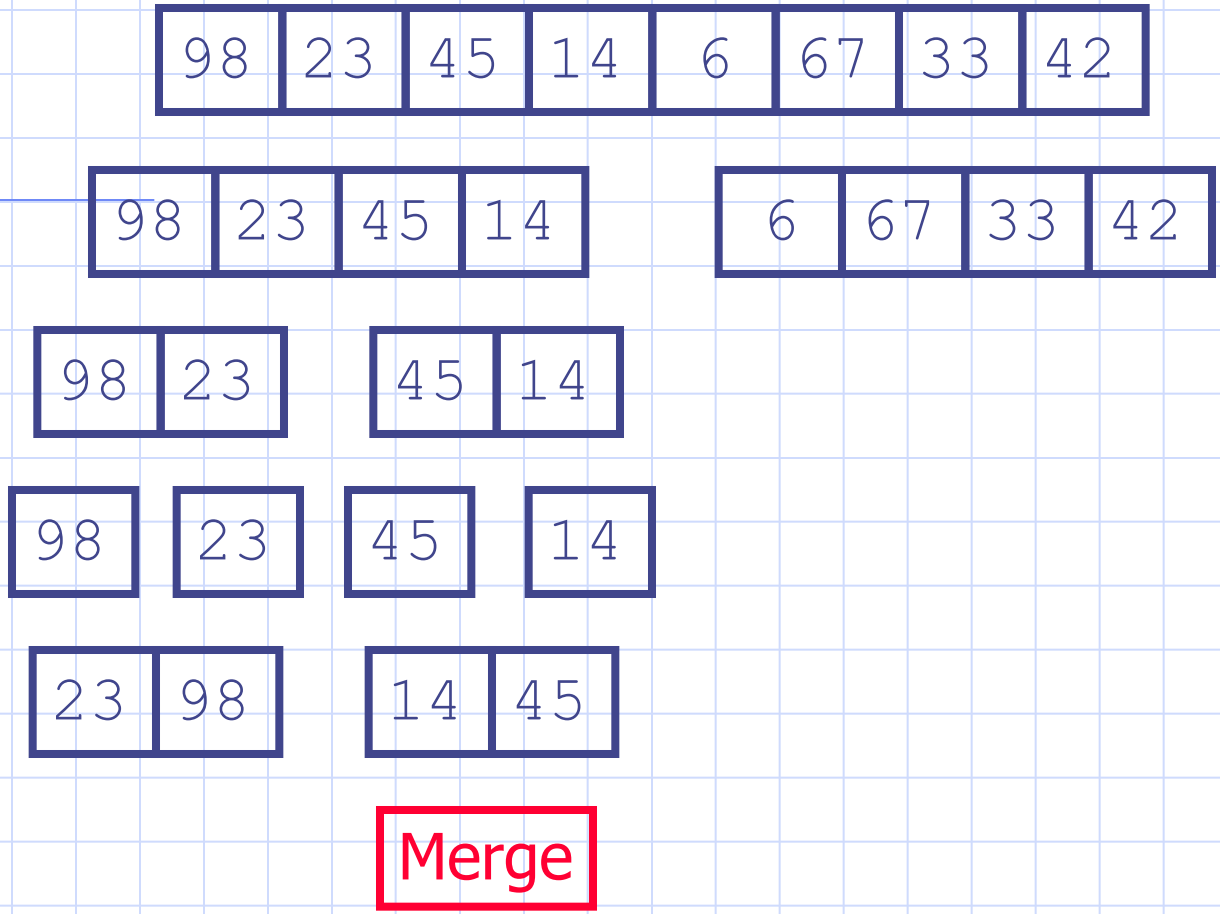

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

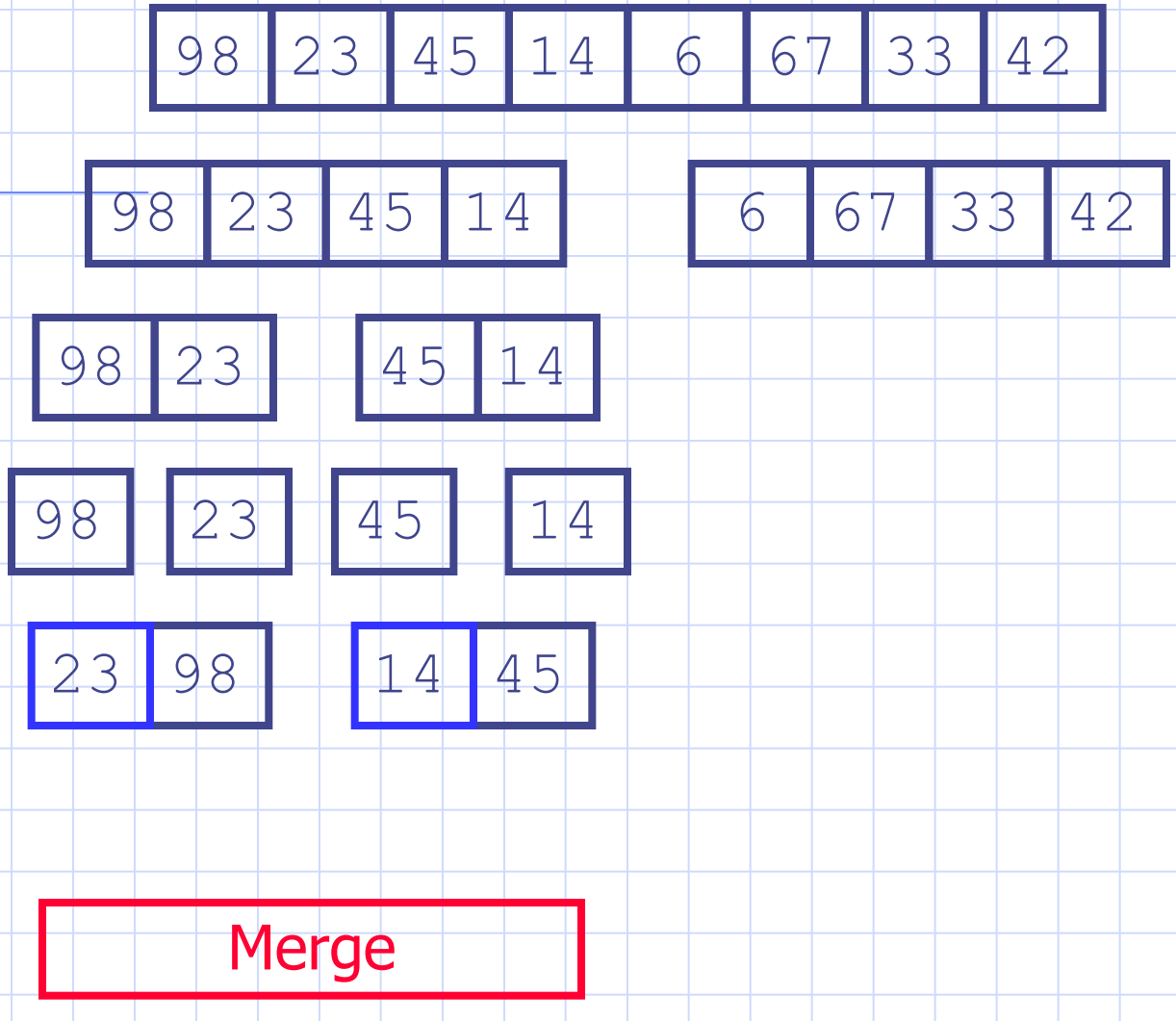

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge

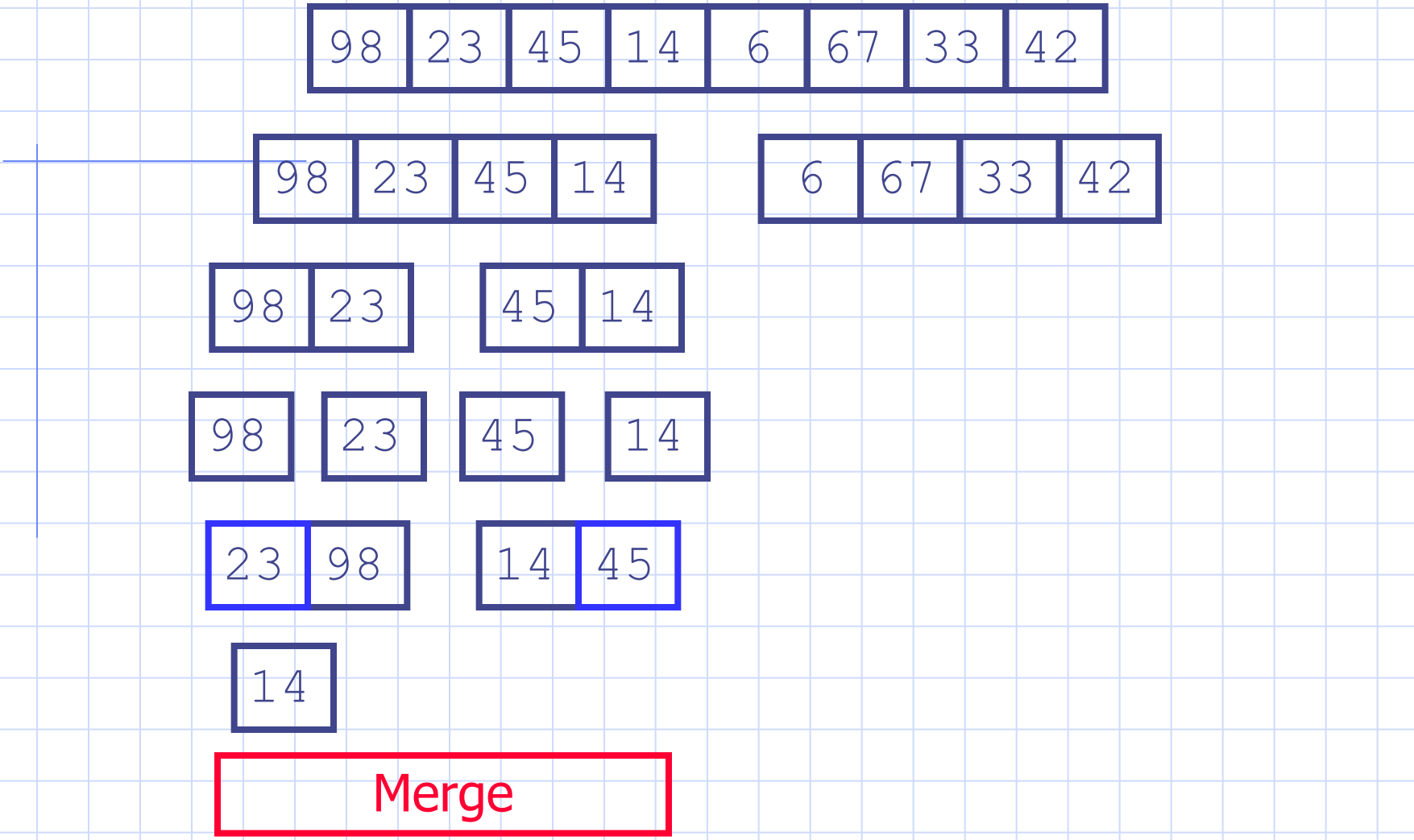

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98

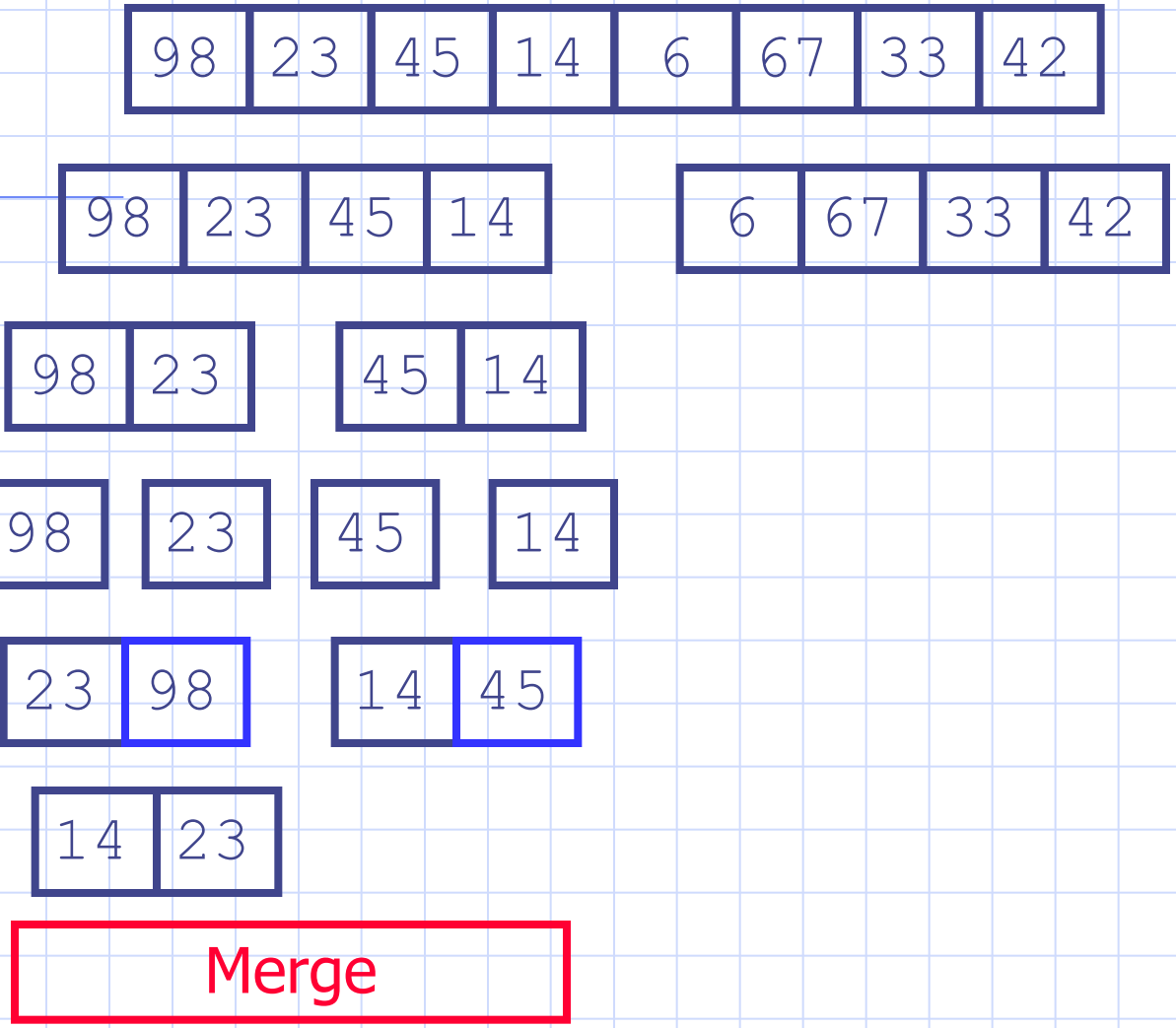

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
Merge

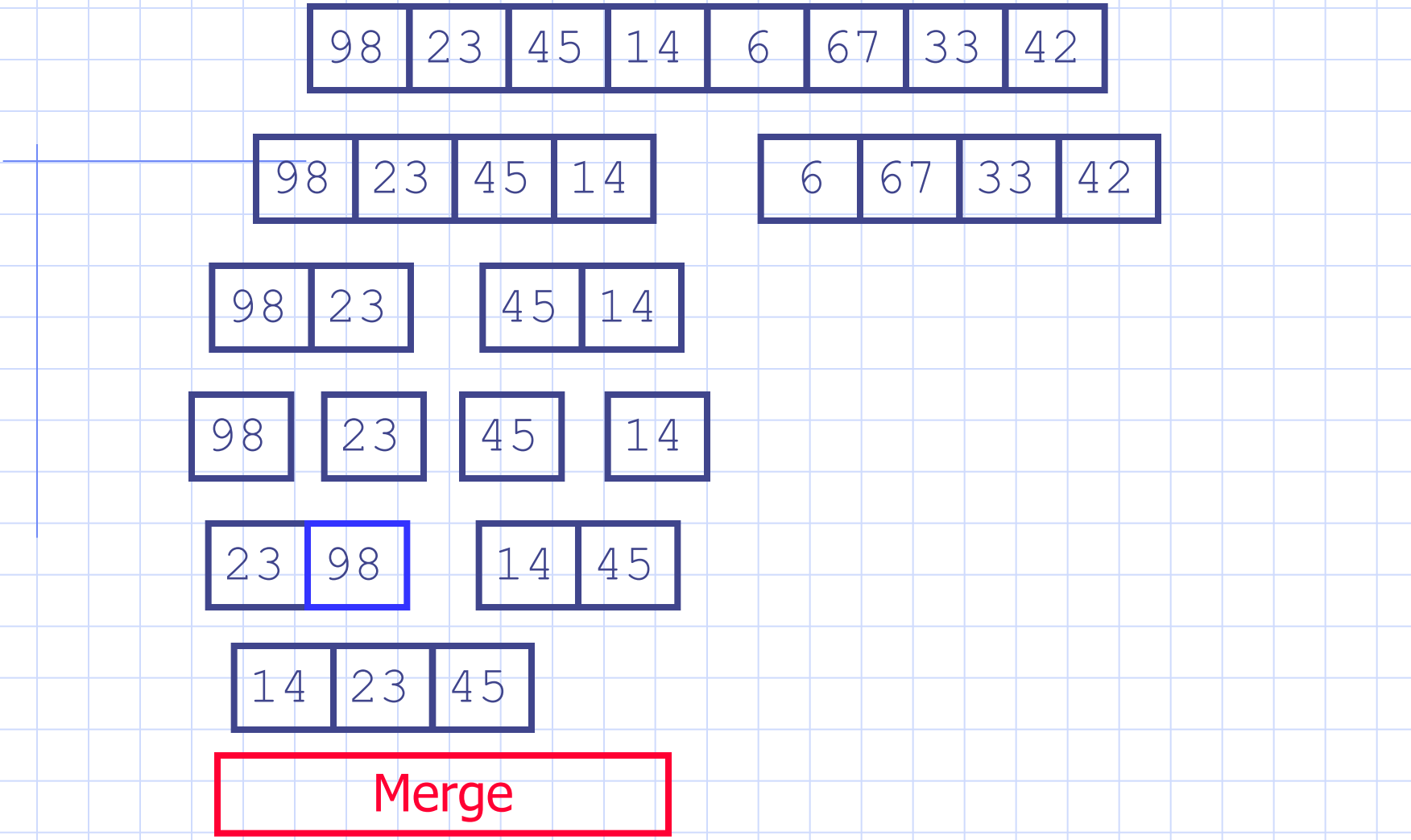

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14
Merge

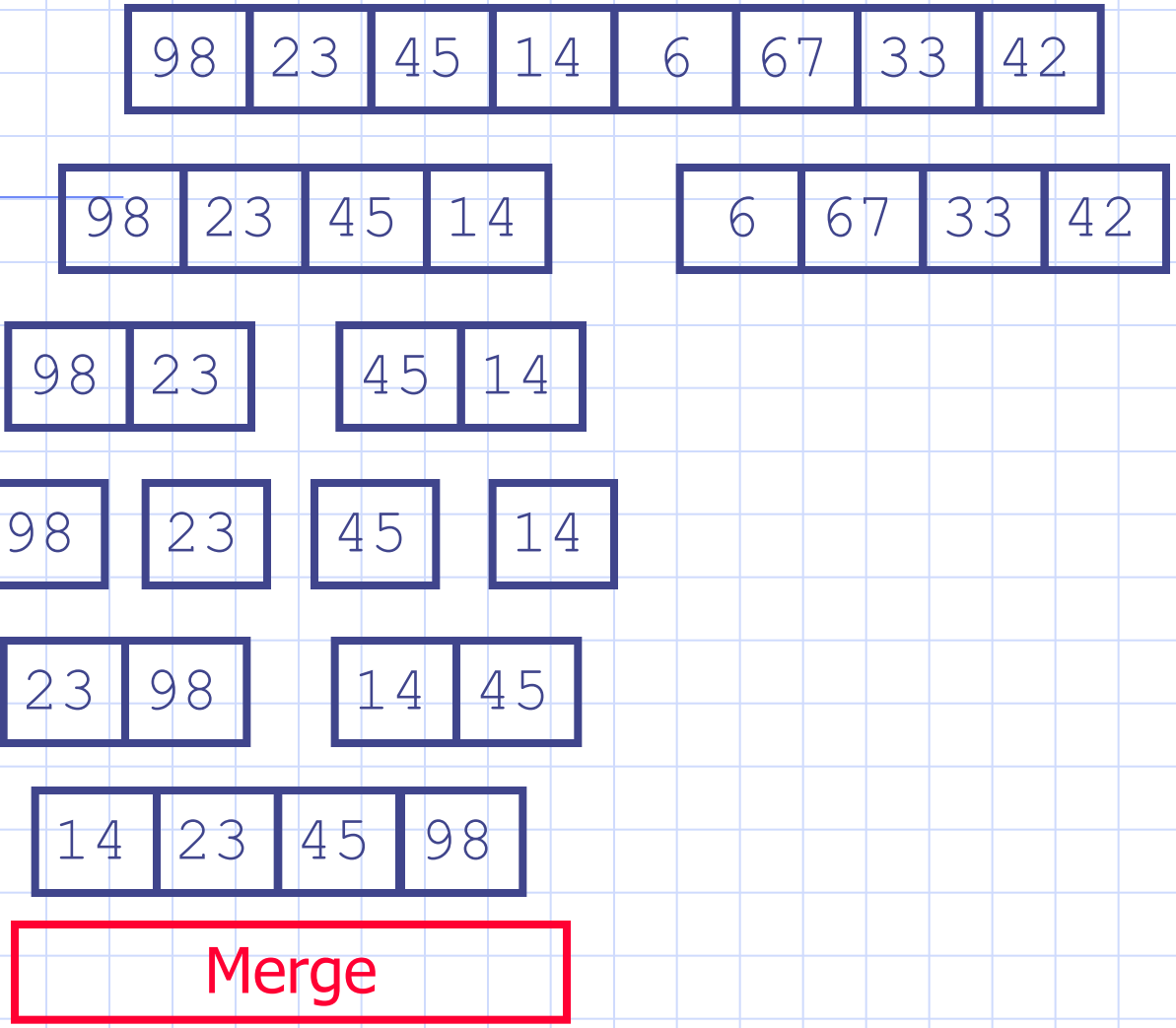

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

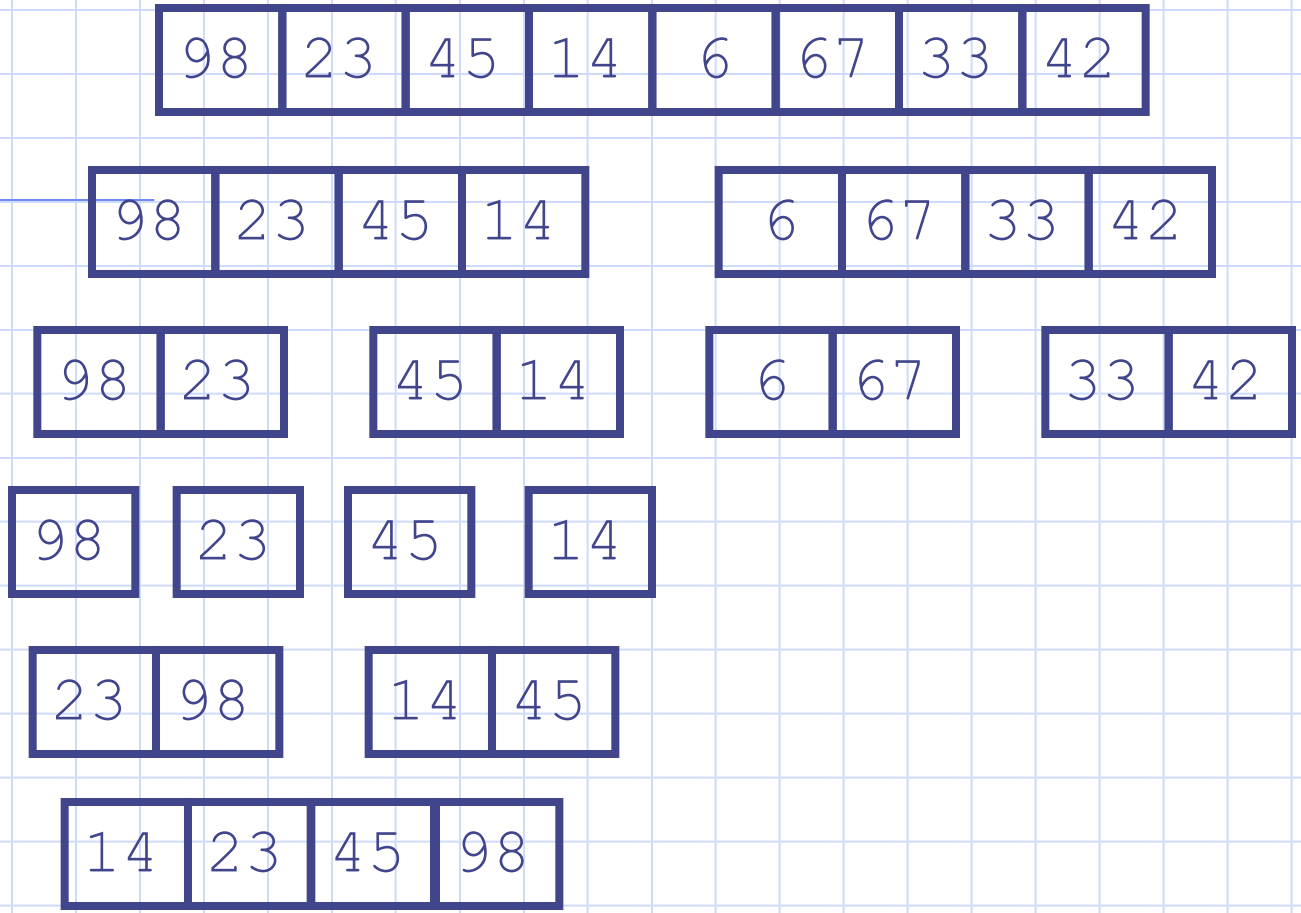

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

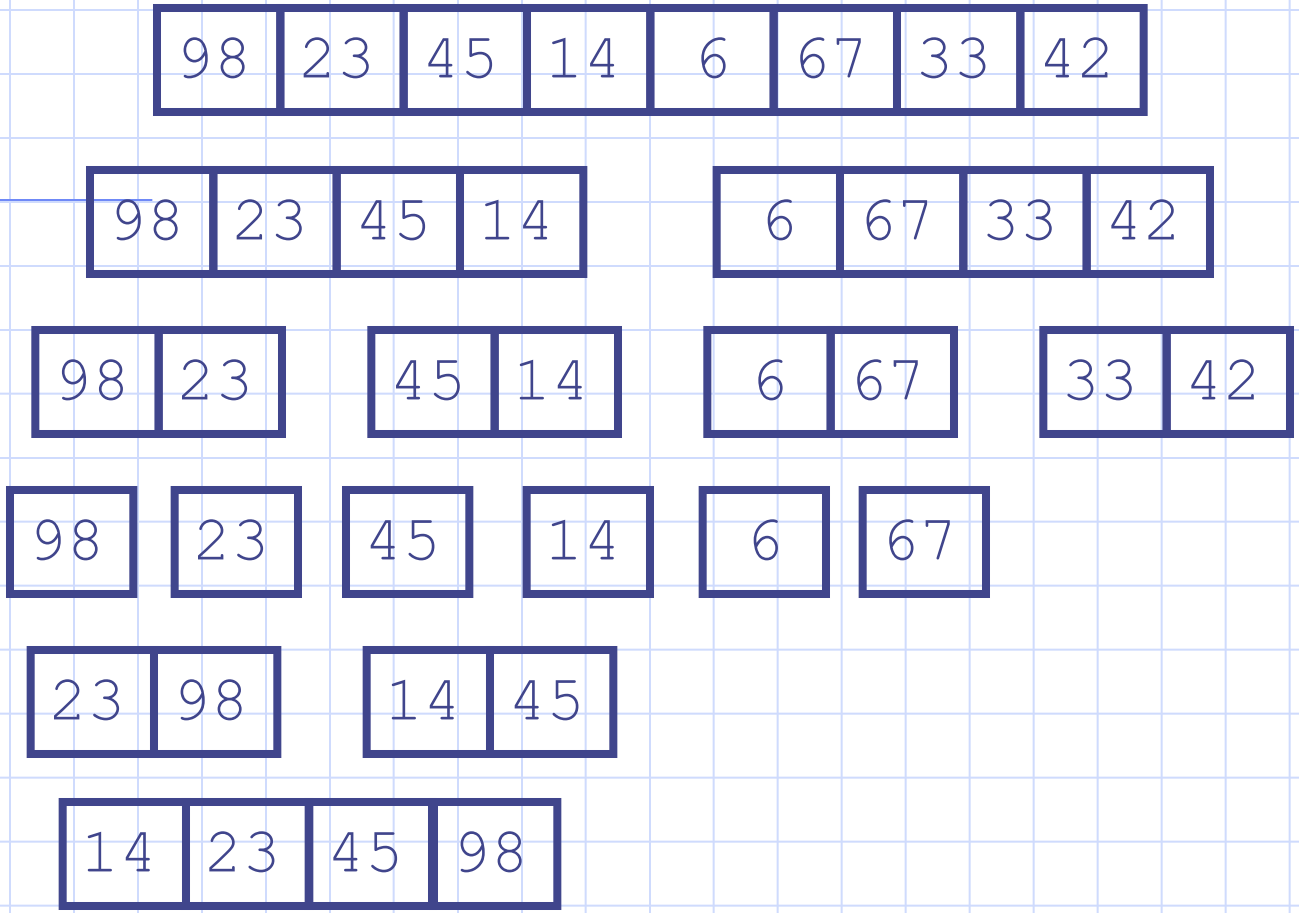

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

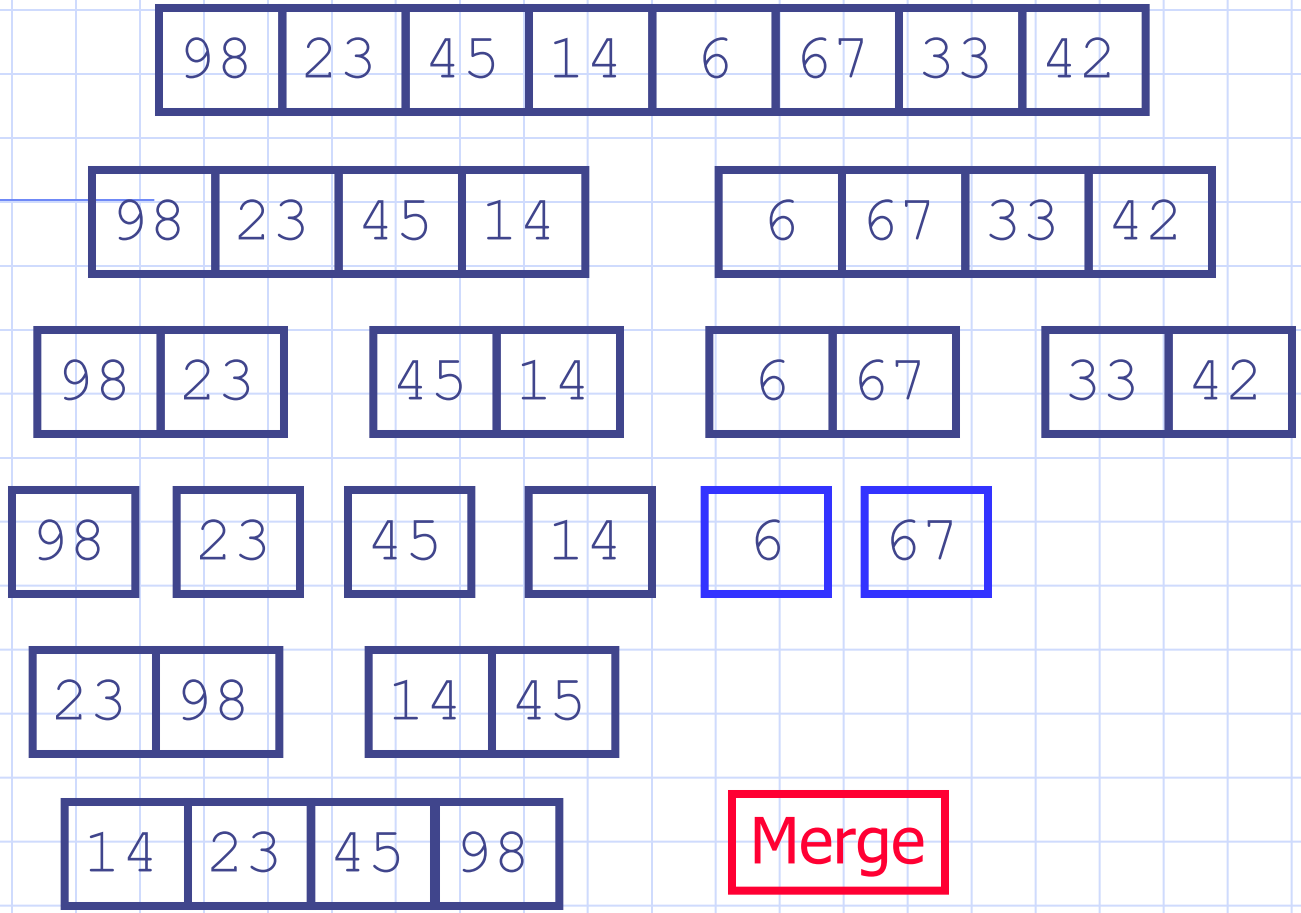

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

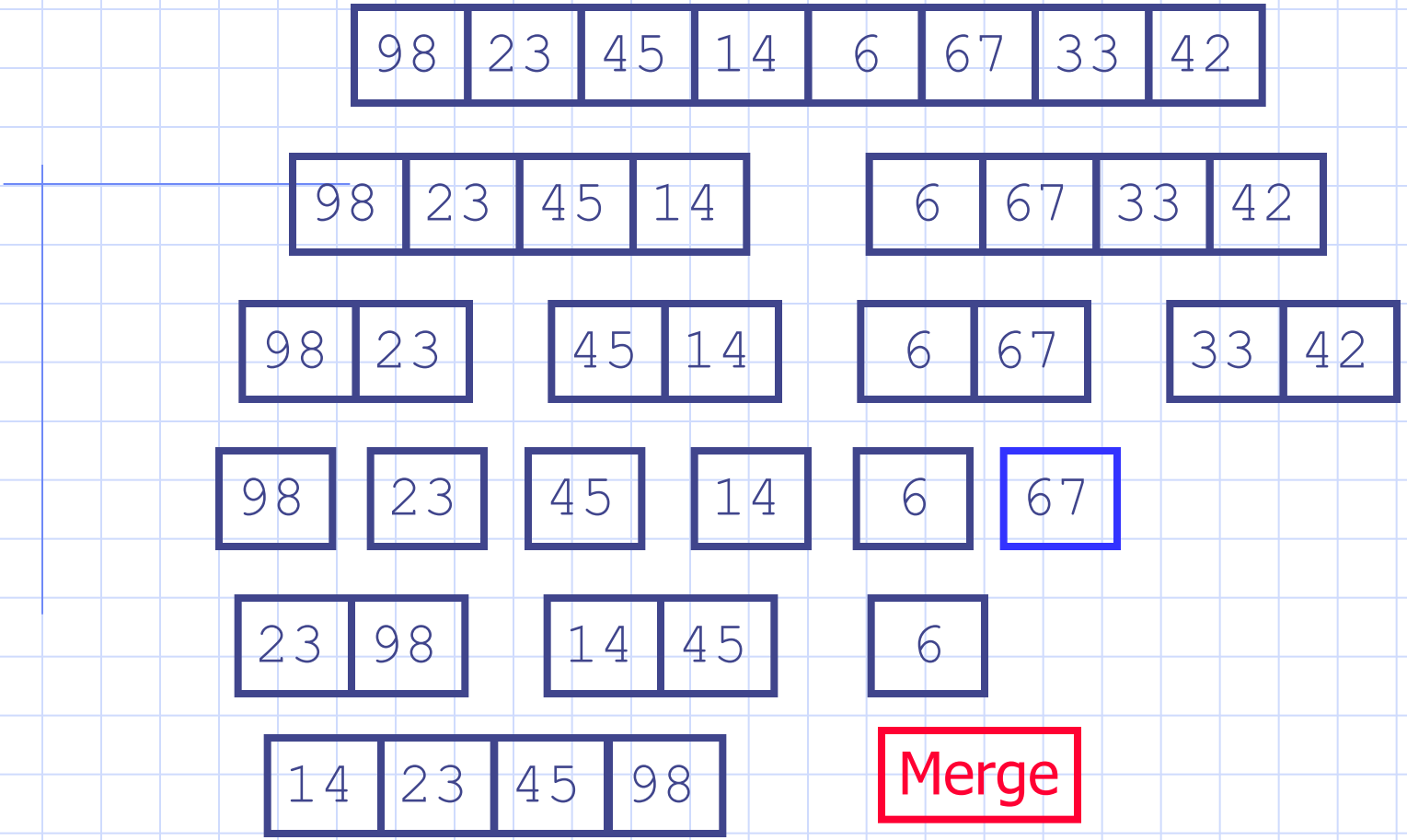

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

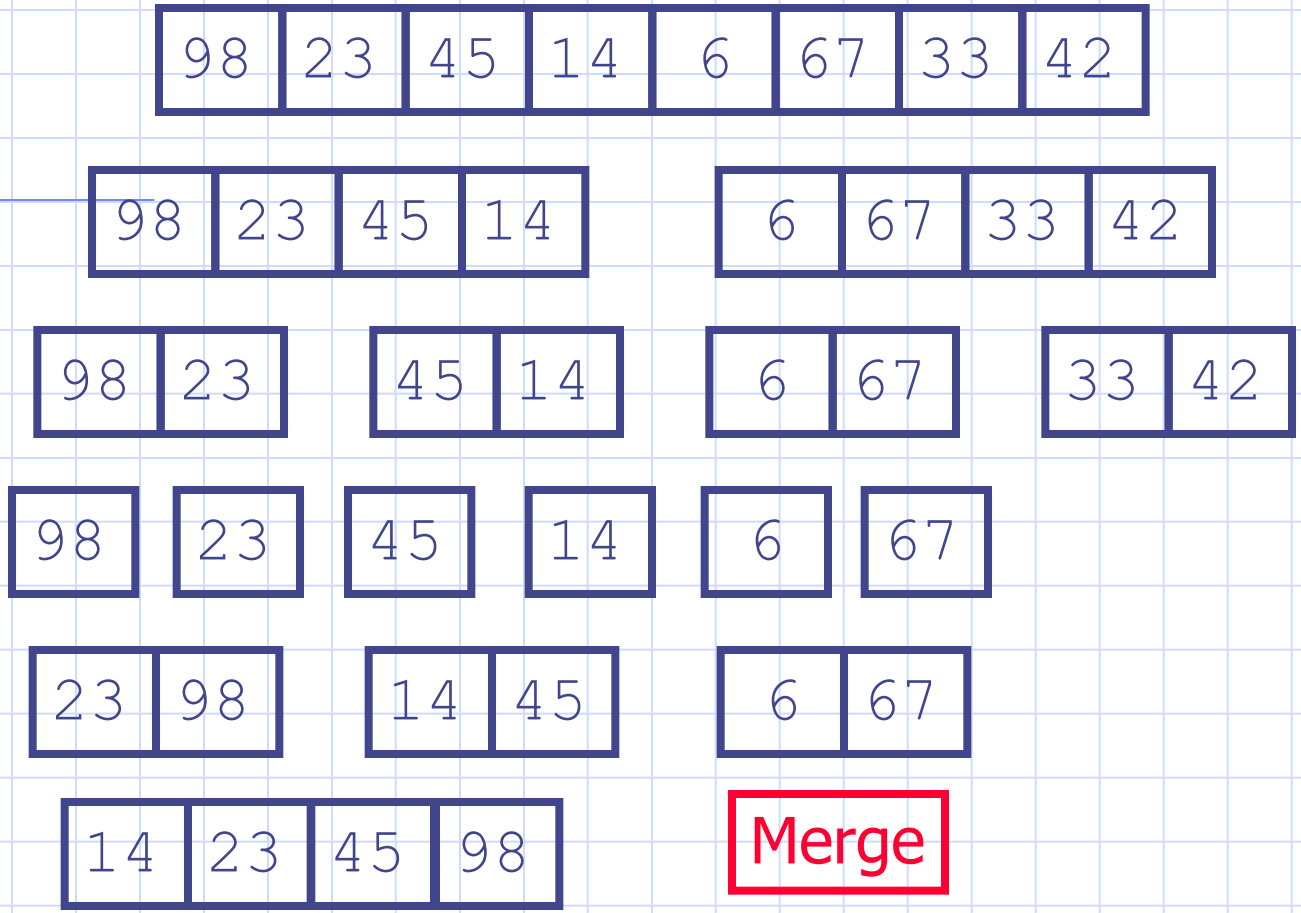

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

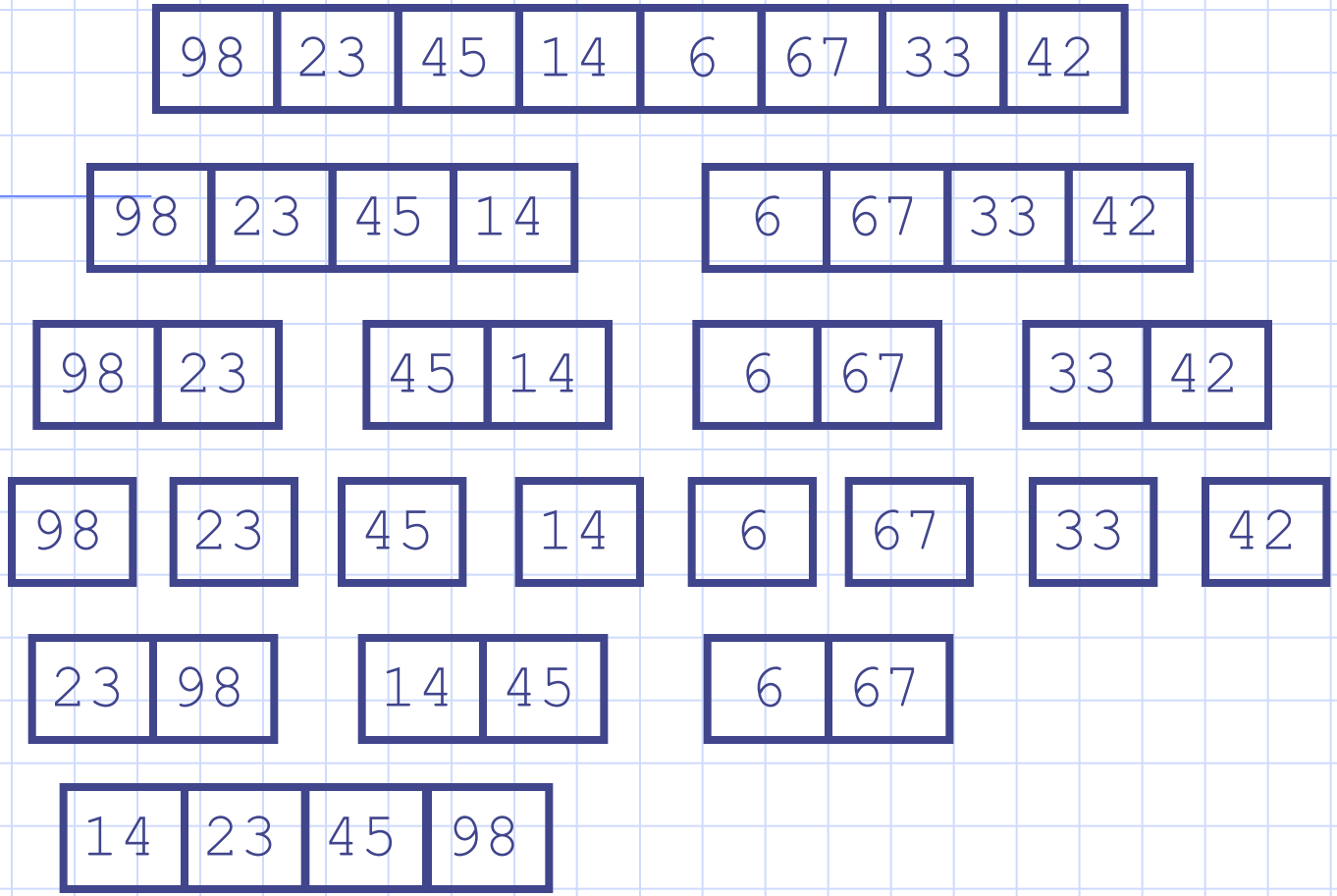

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

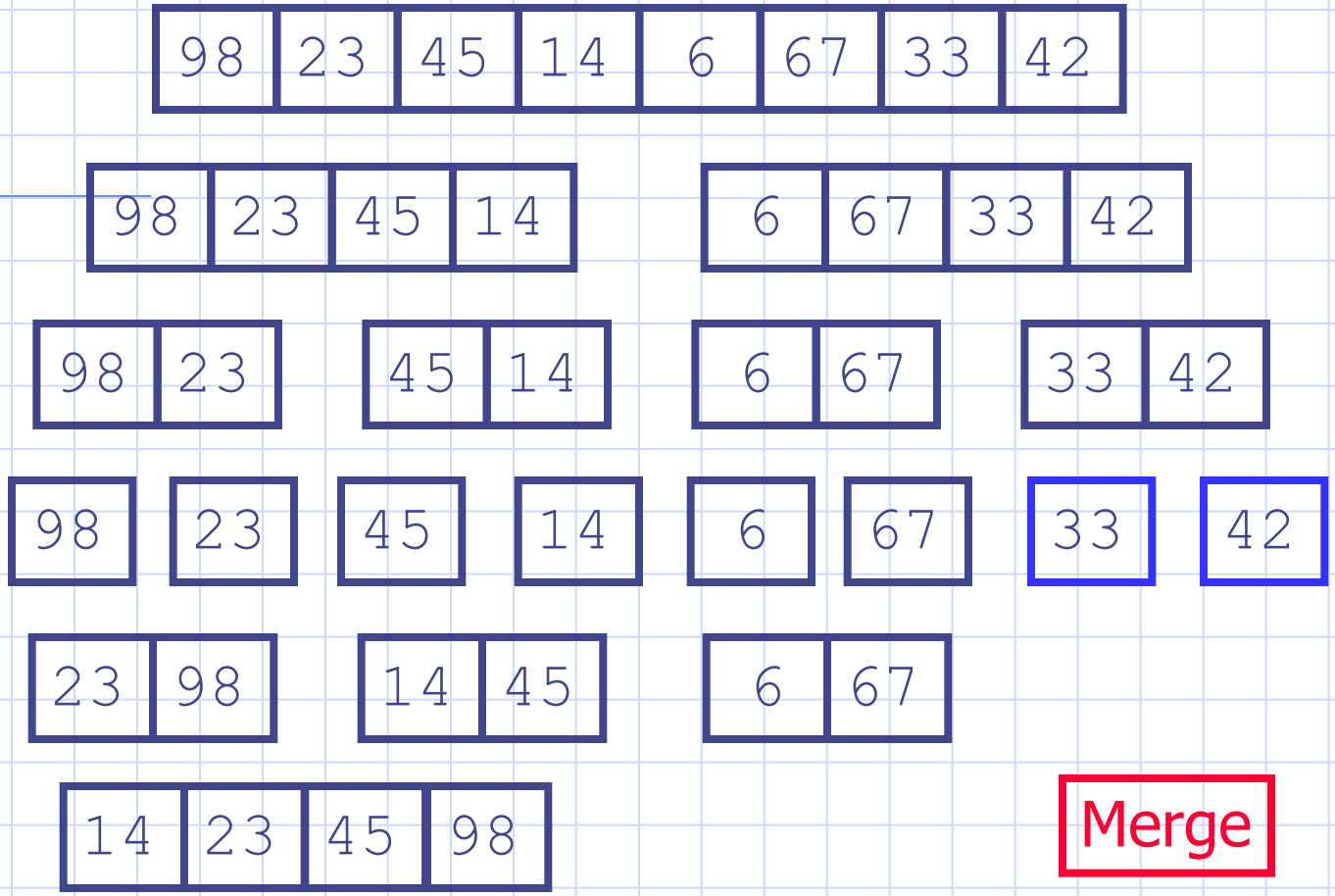

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

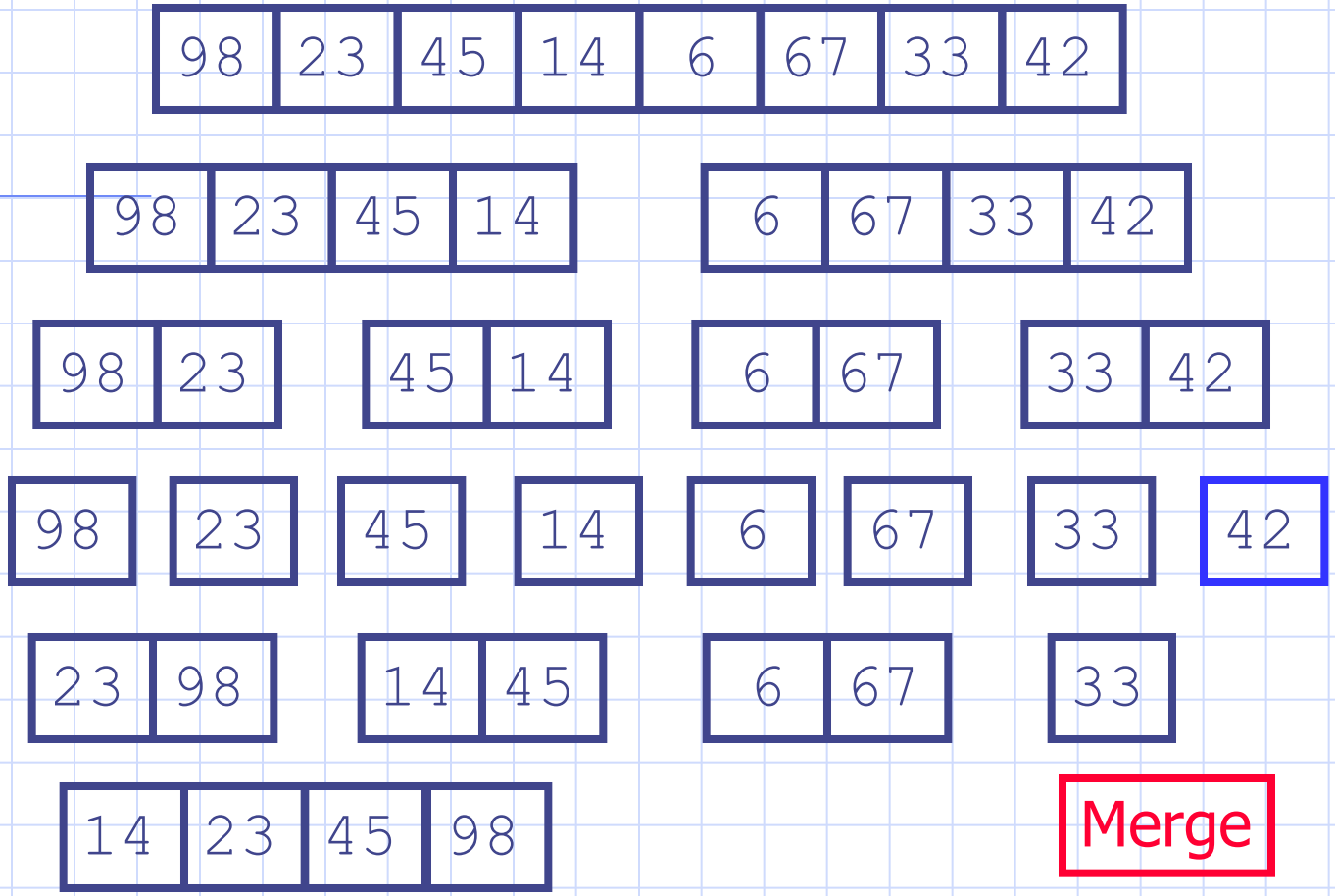

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

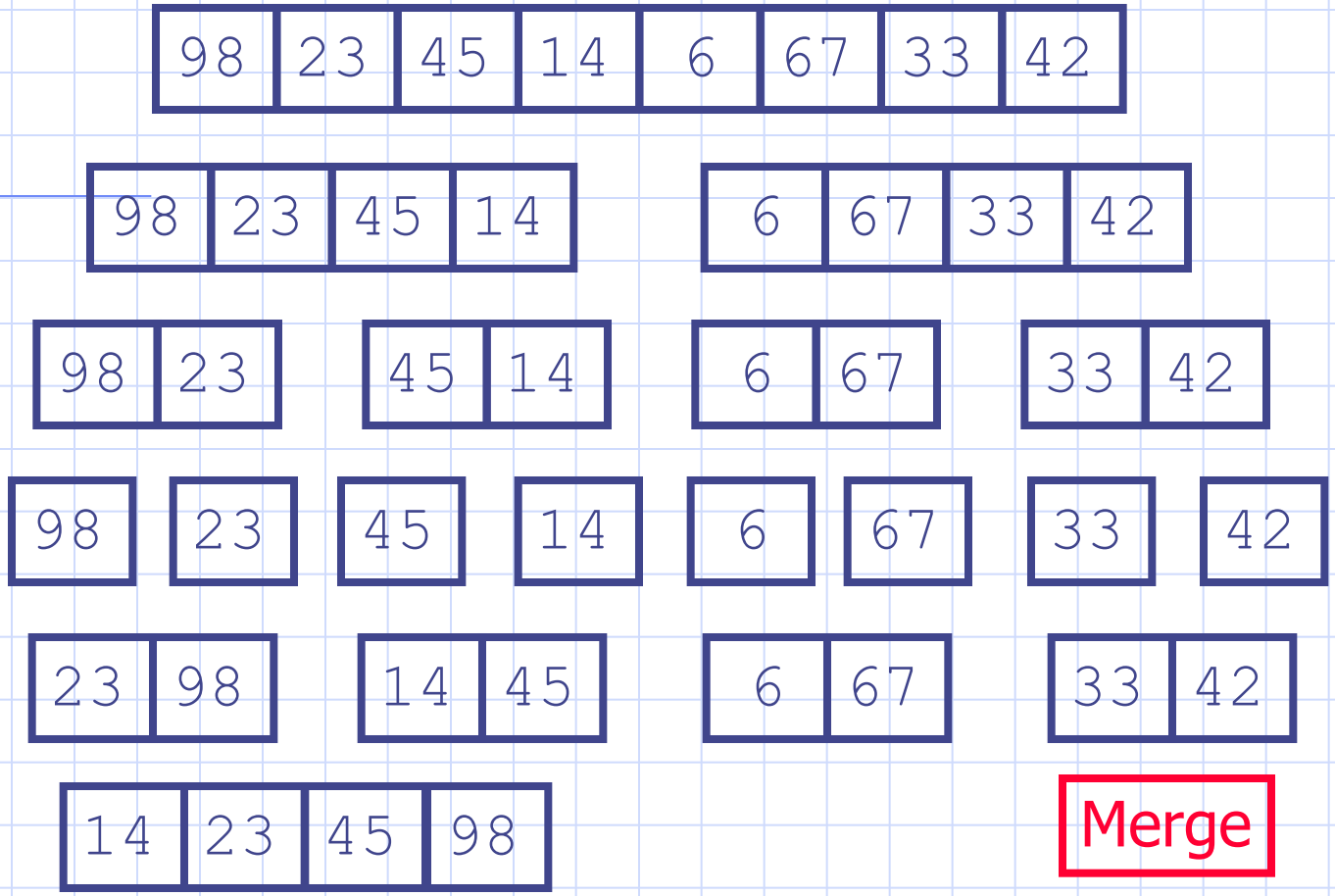

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge

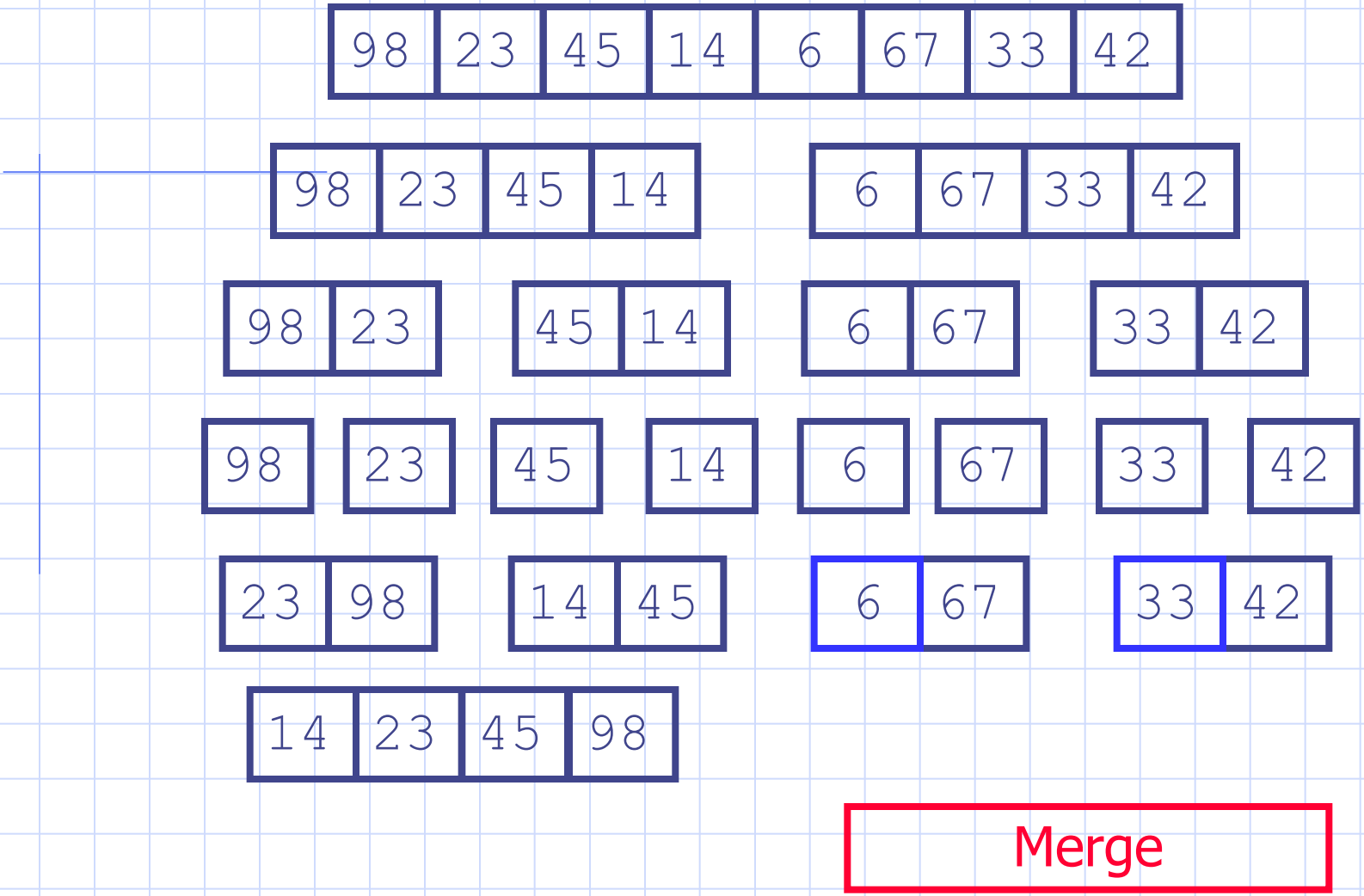

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6 67
14 23 45 98
Merge

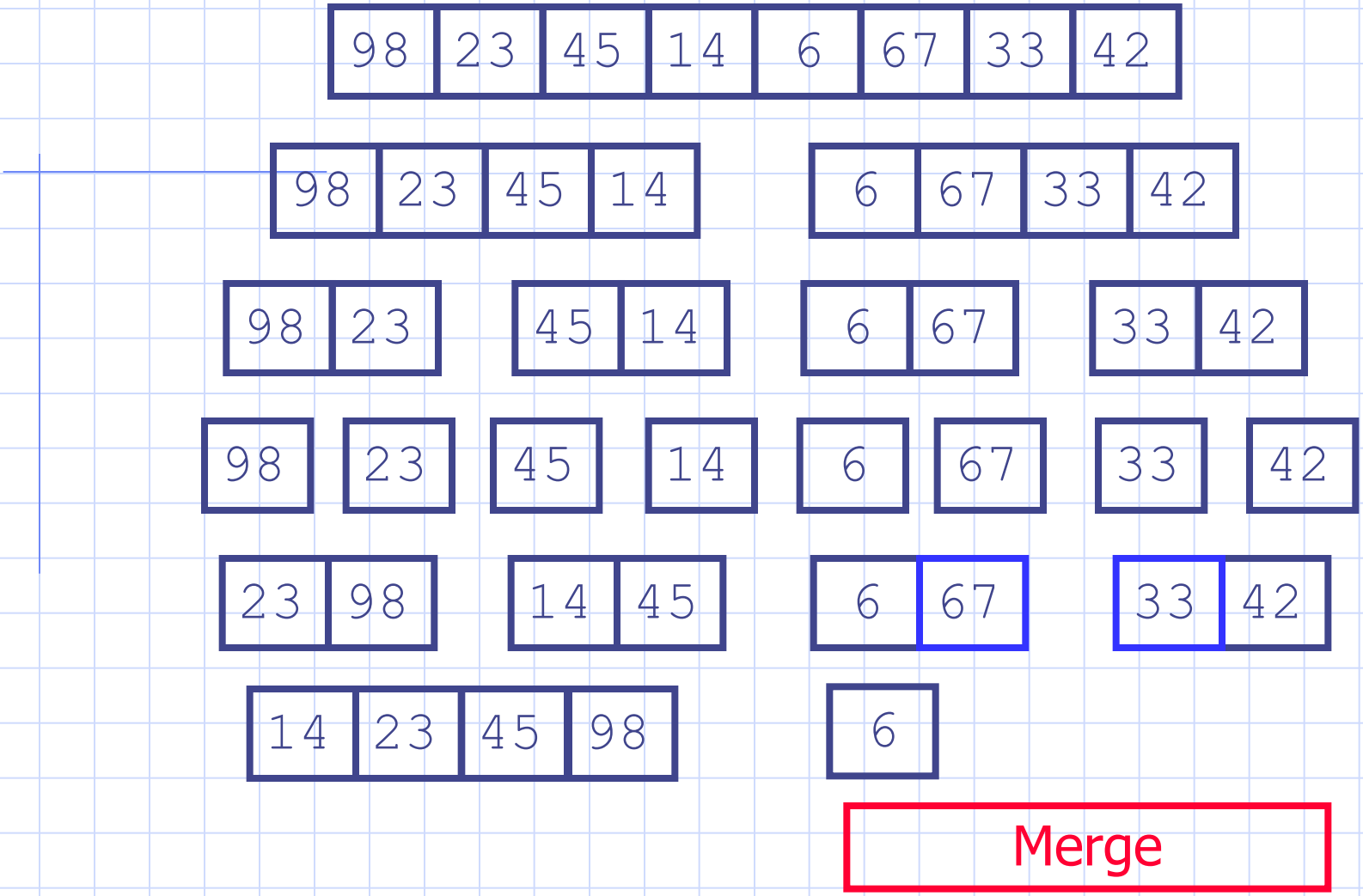

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98

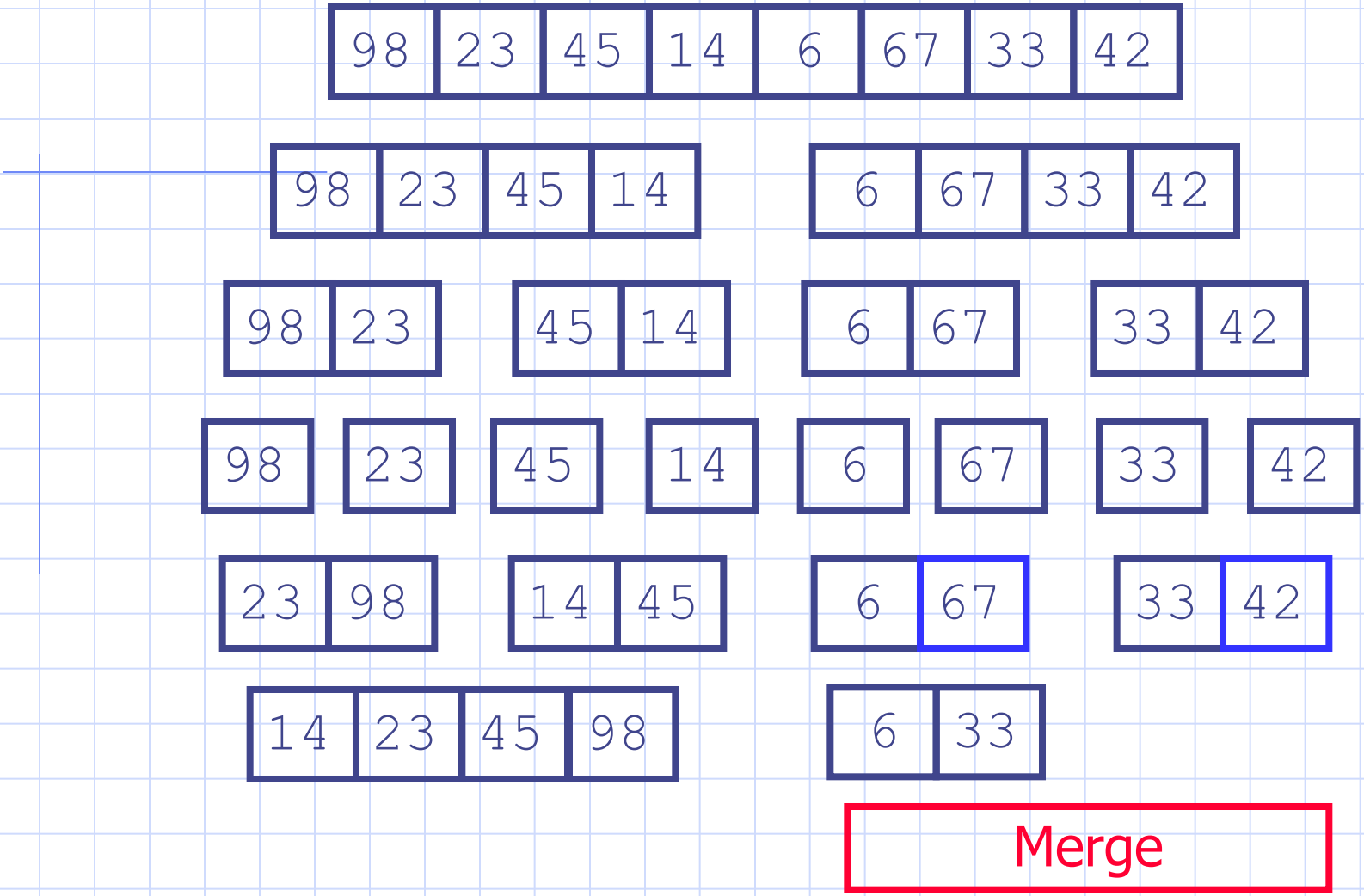

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98
Merge

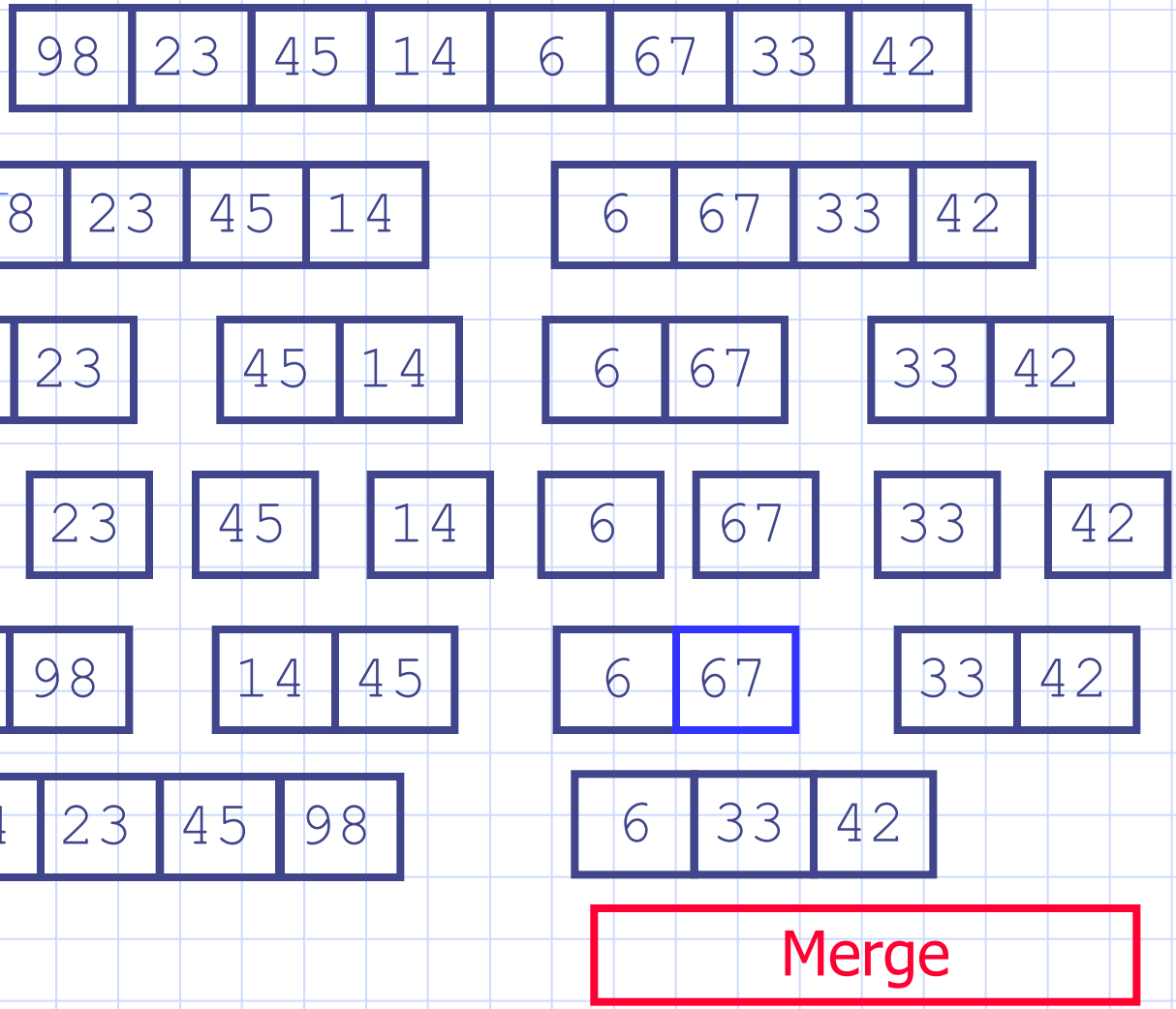

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge

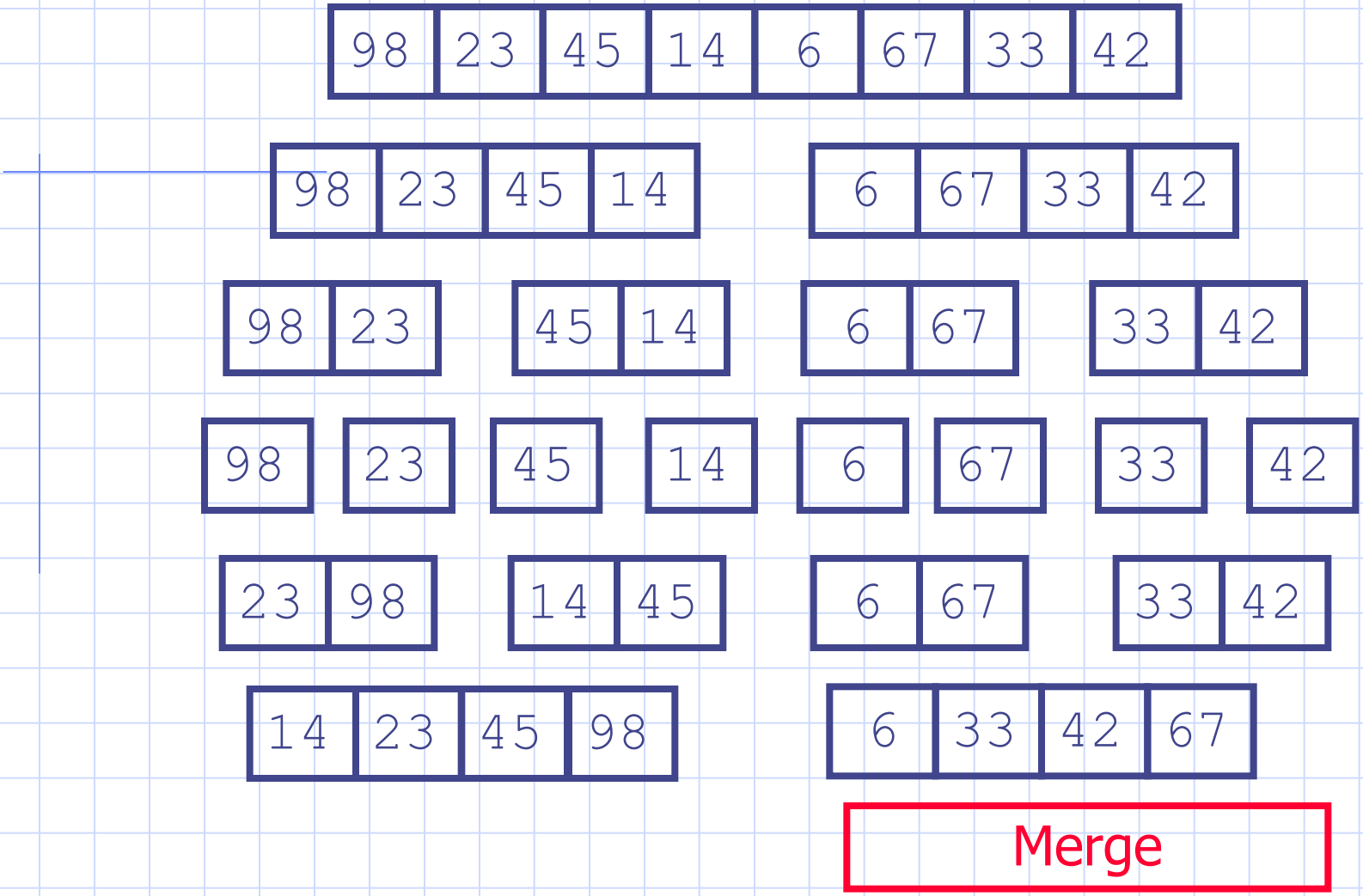

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

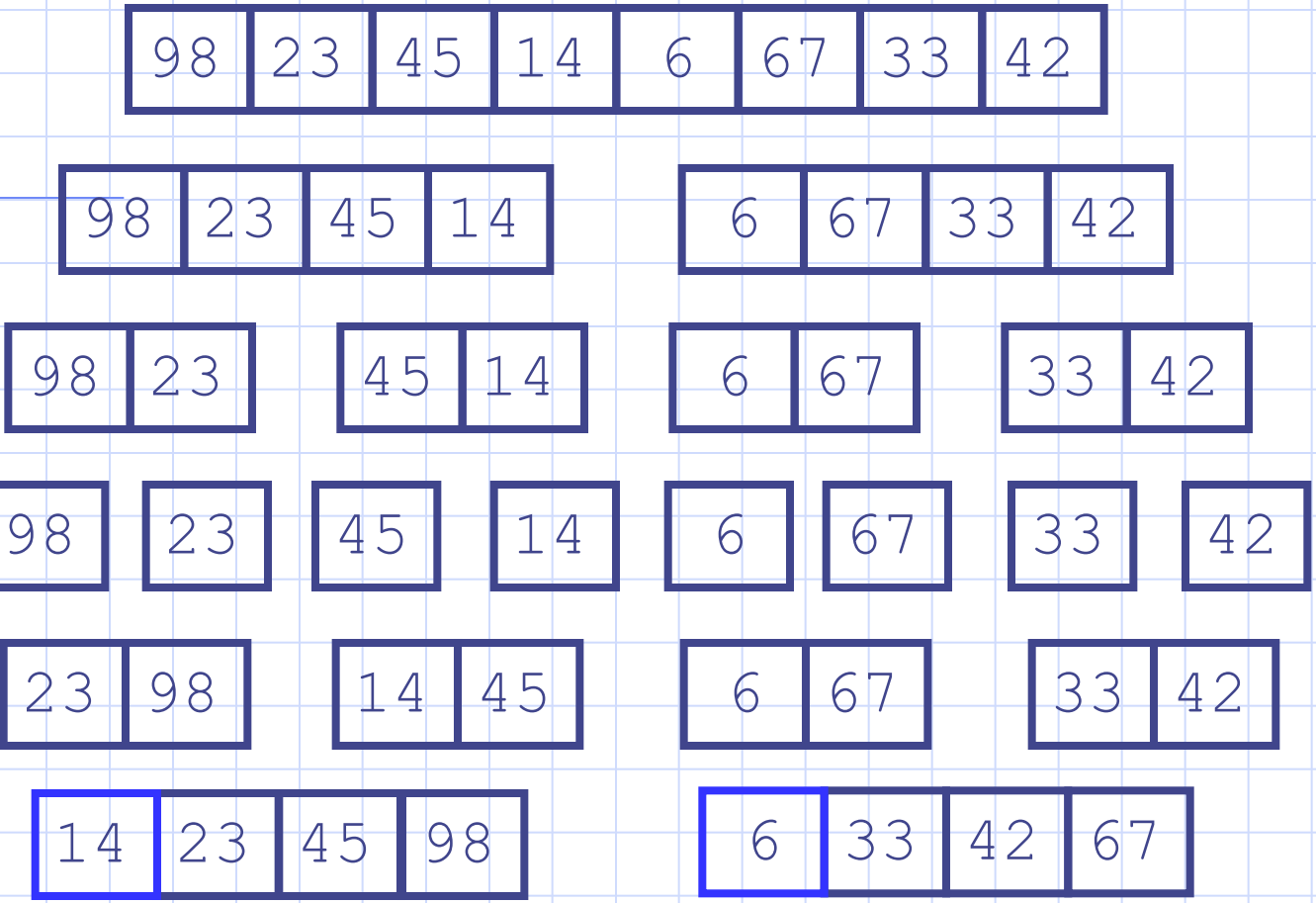

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

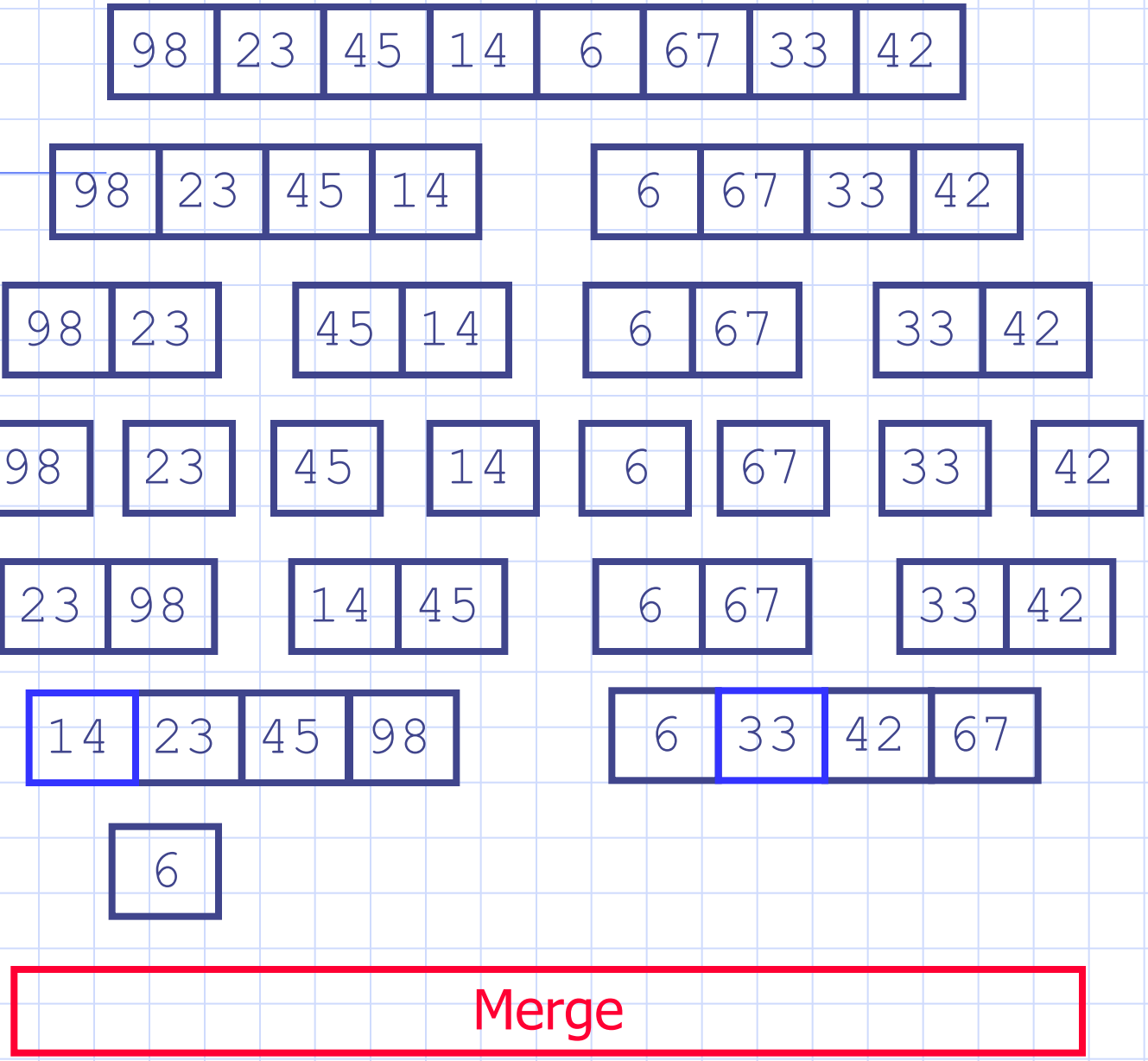

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6
Merge

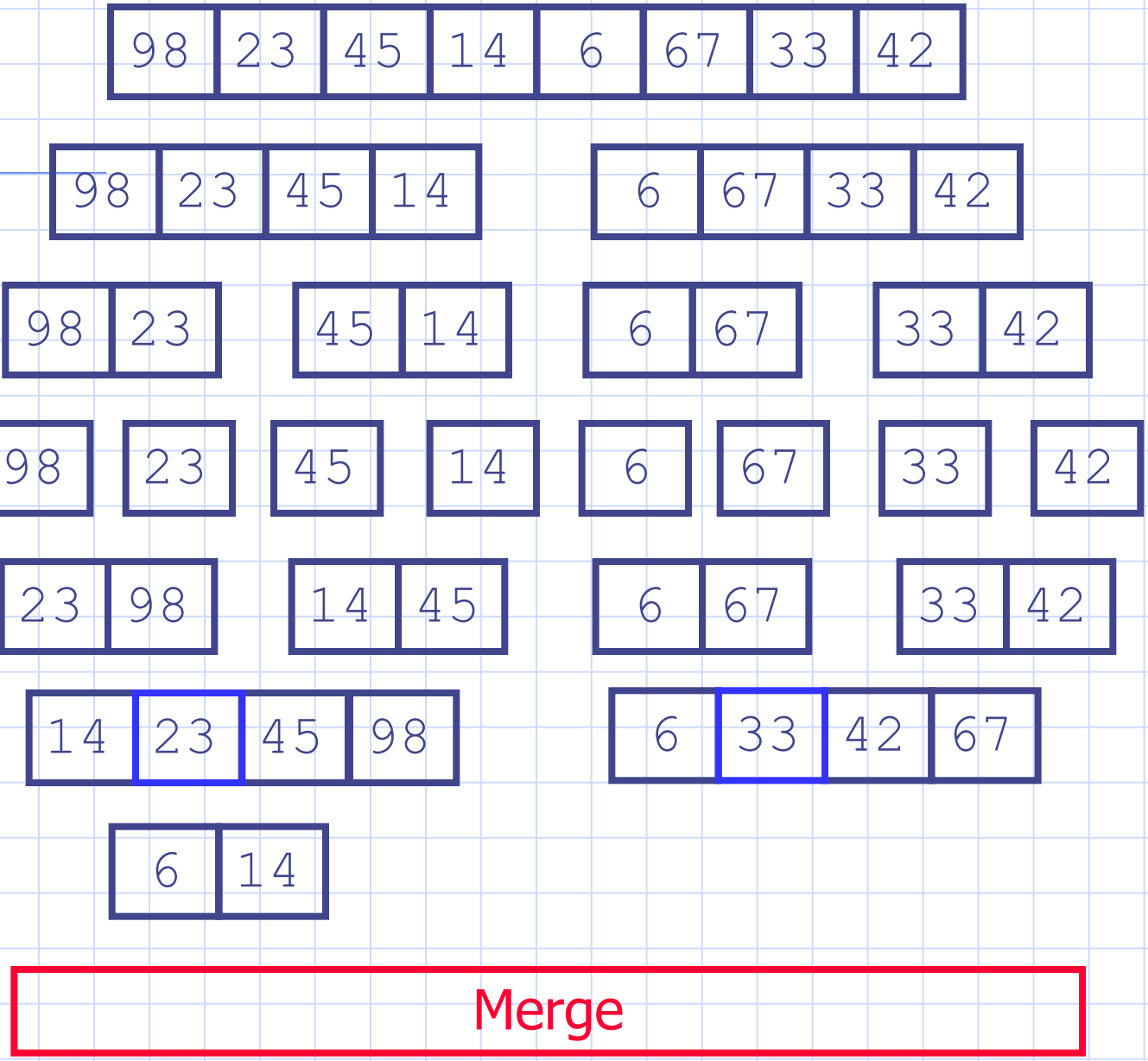

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33
Merge

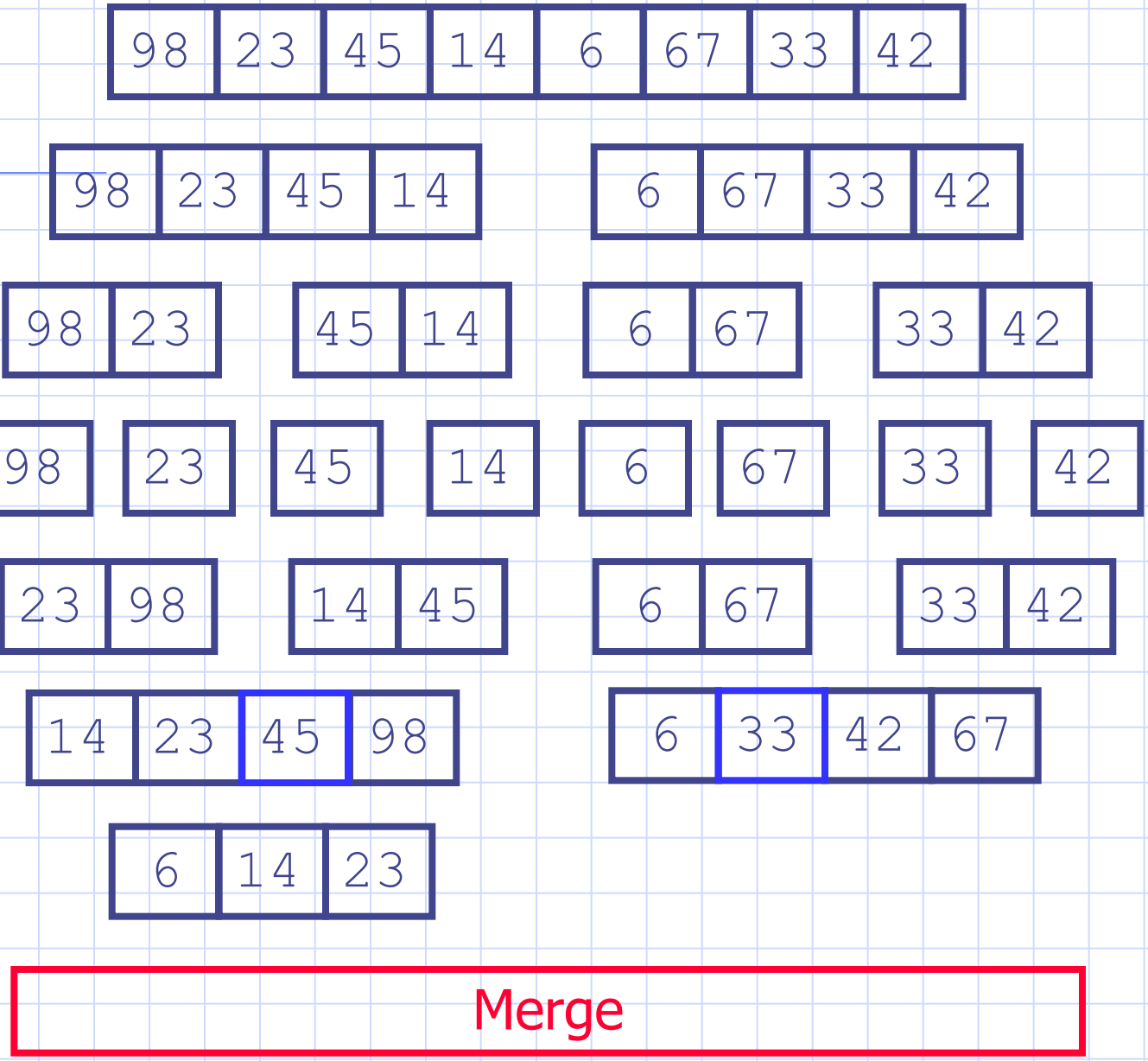

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42
Merge

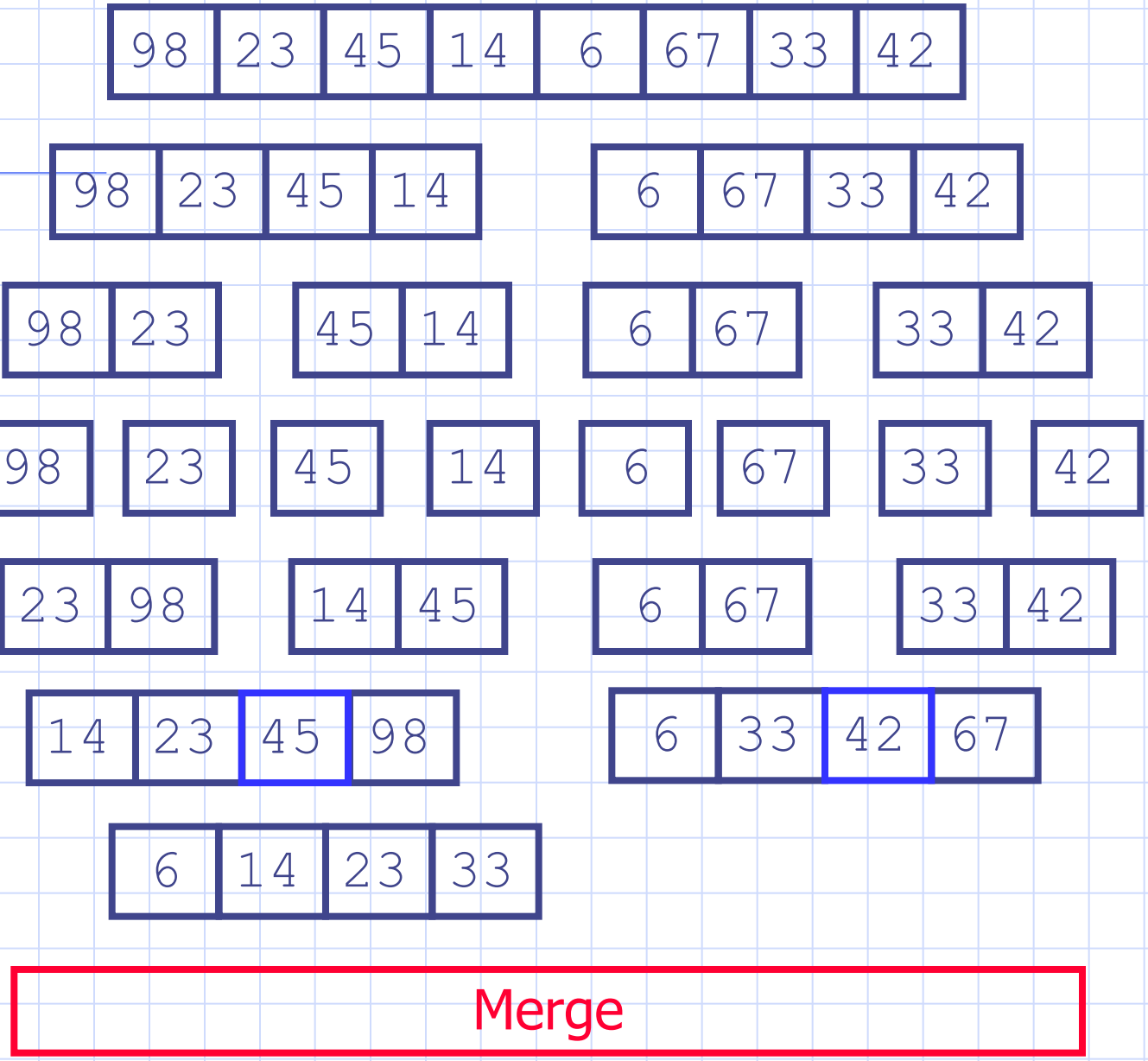

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

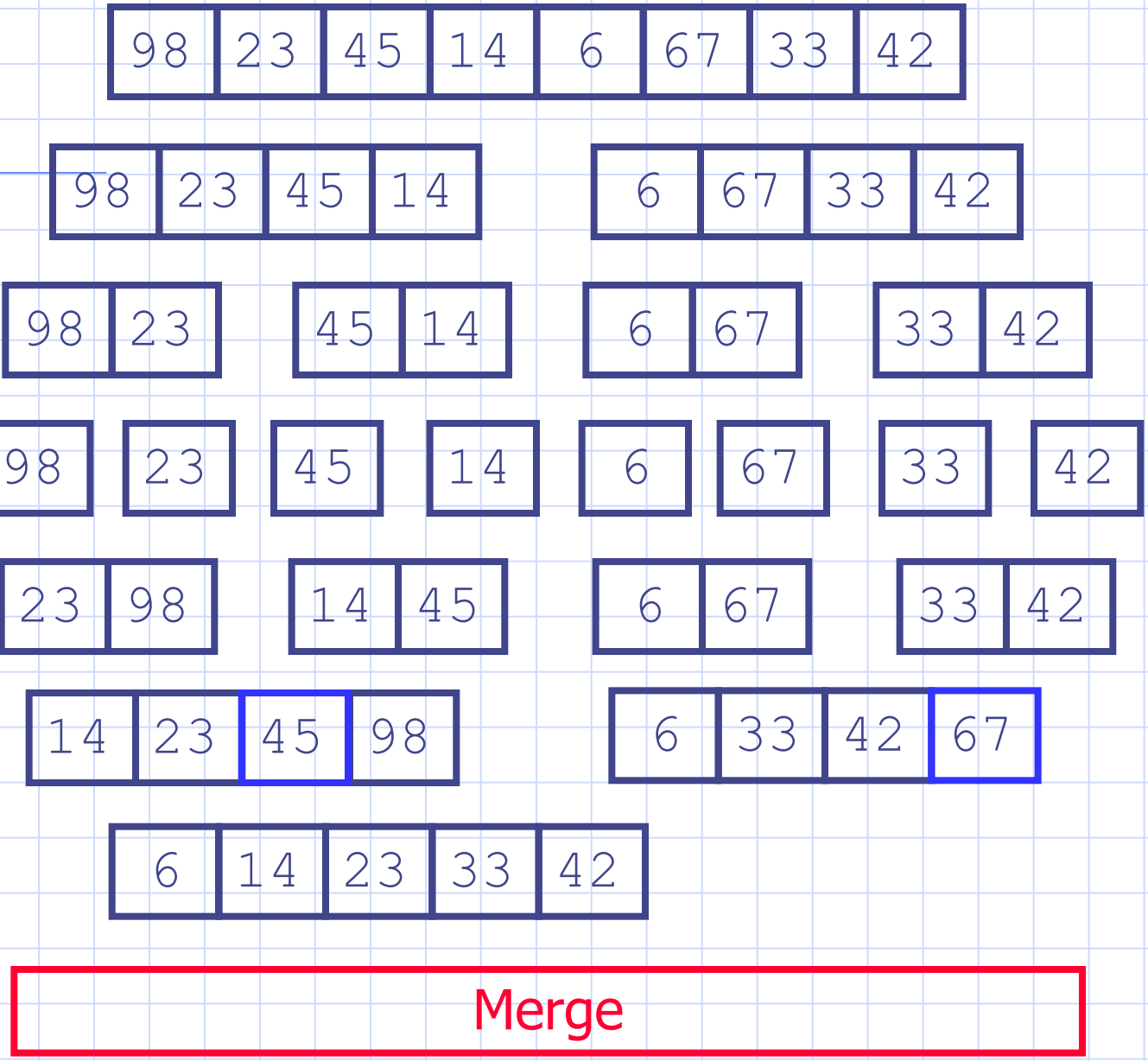

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

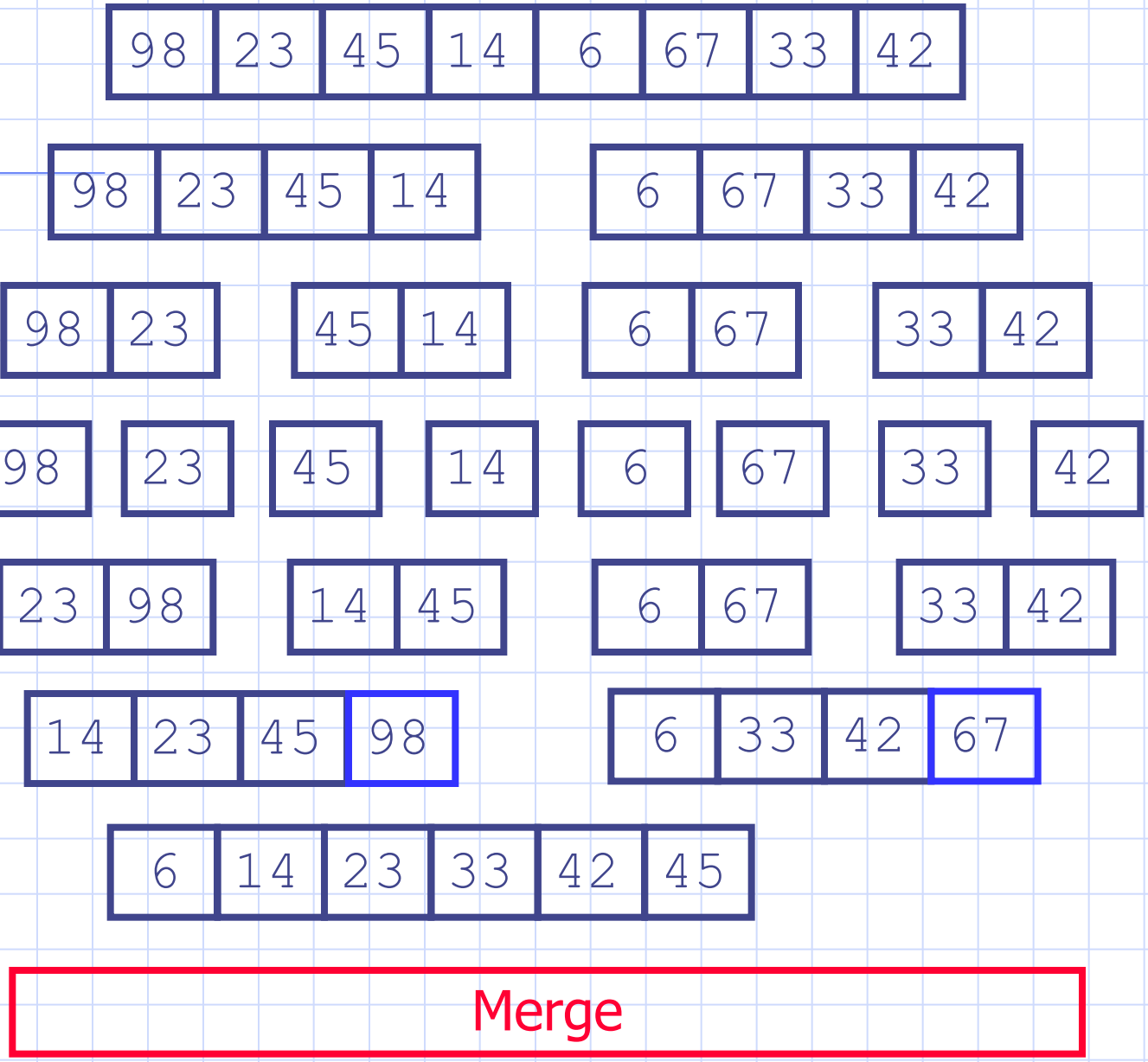

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6
Merge

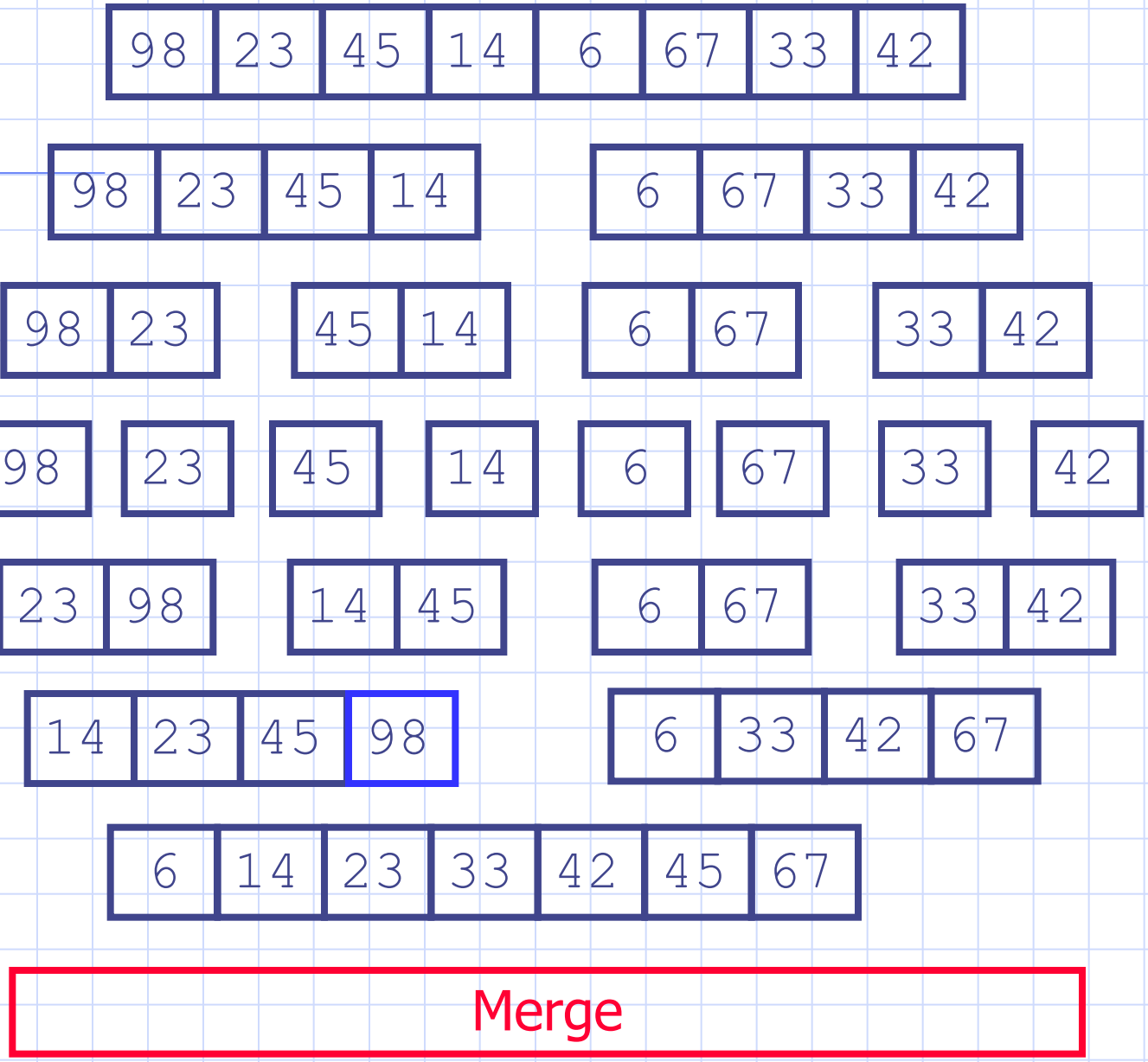

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14
Merge

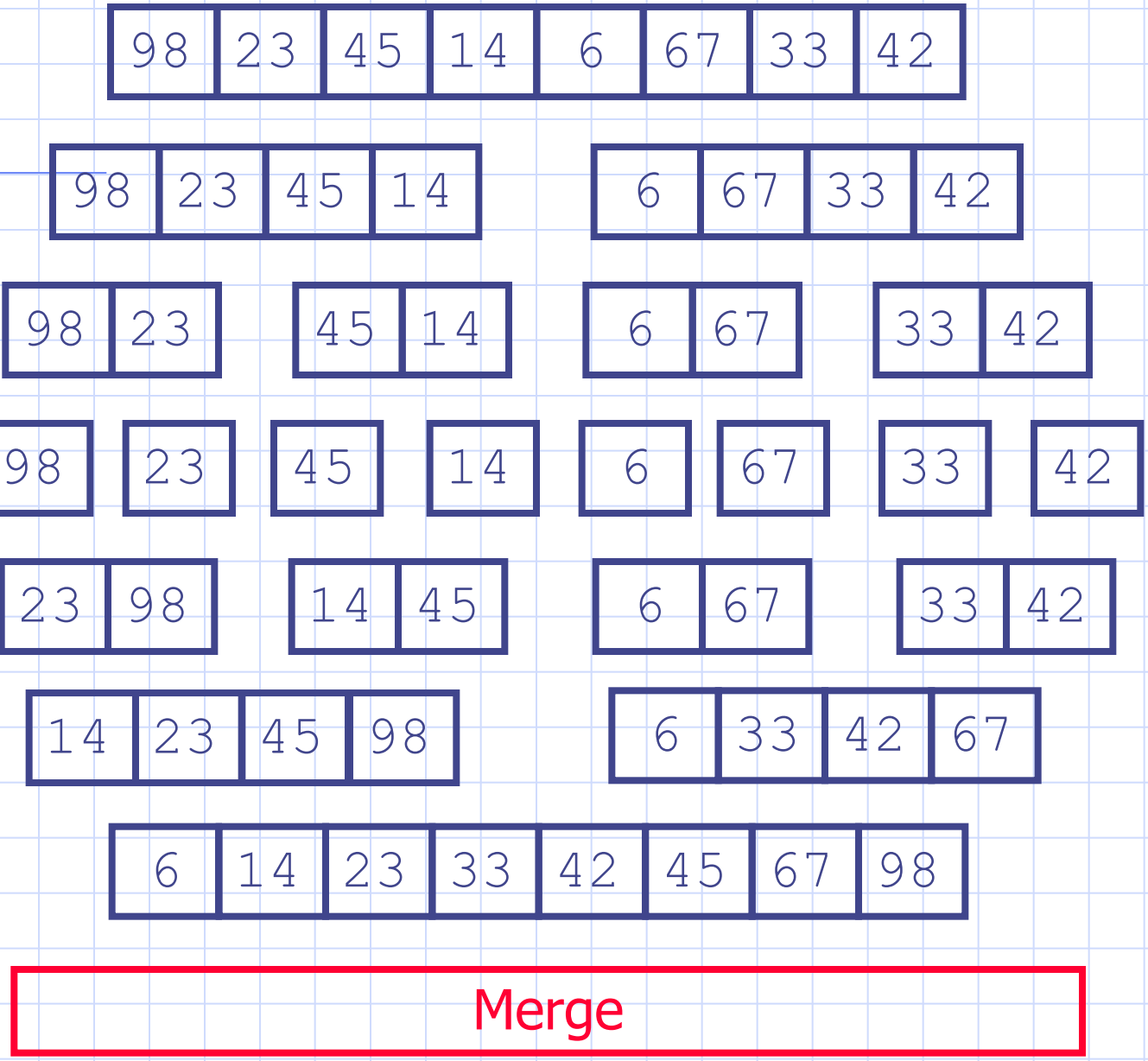

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23
Merge

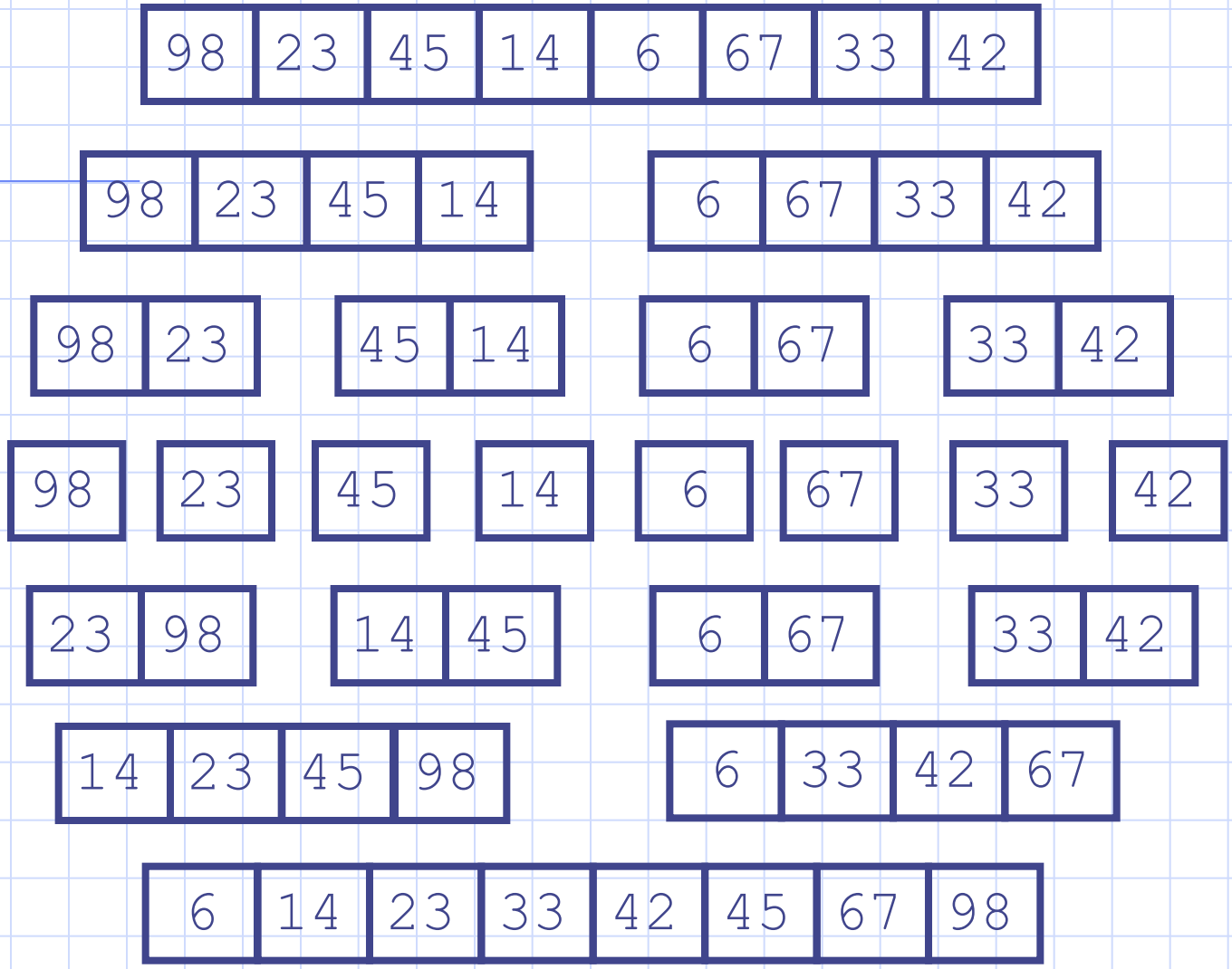

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33
Merge

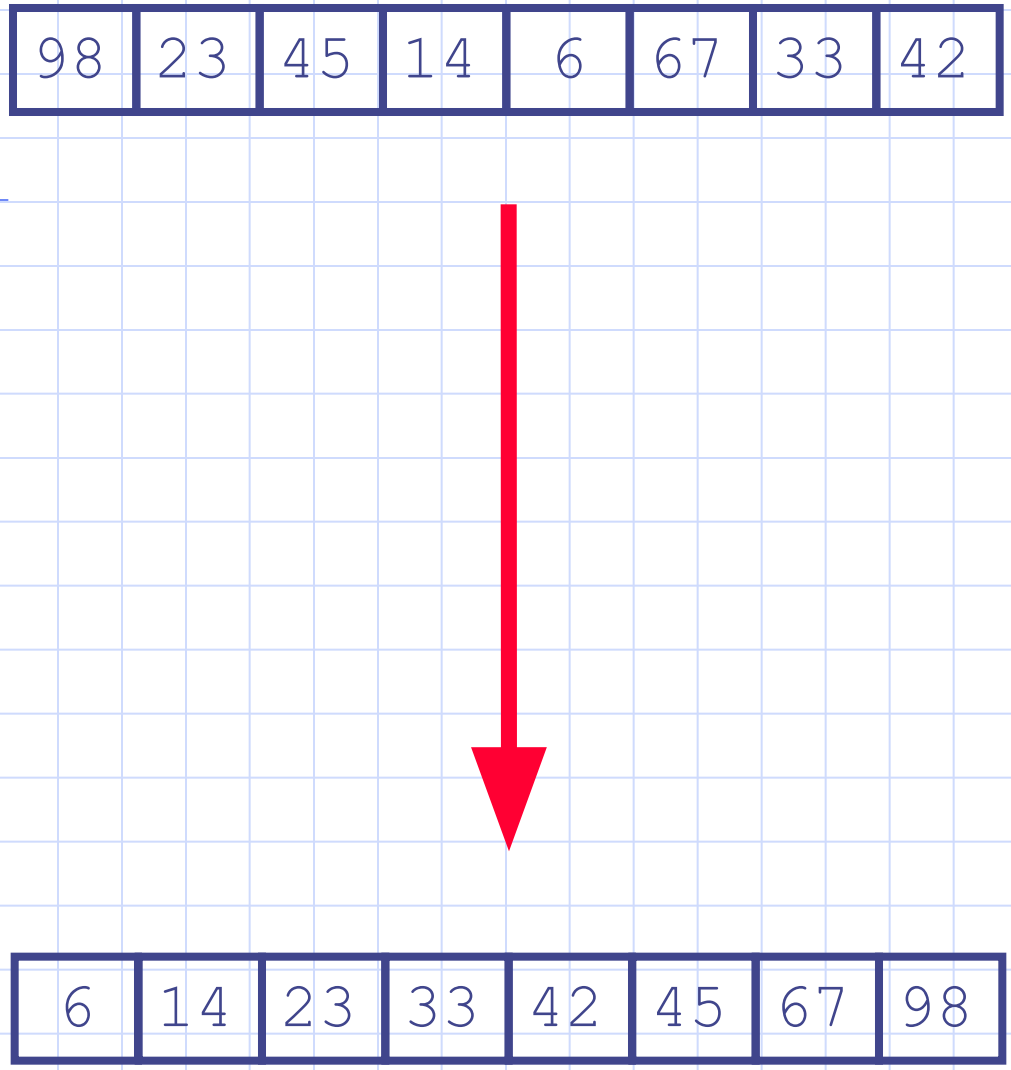

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45
Merge

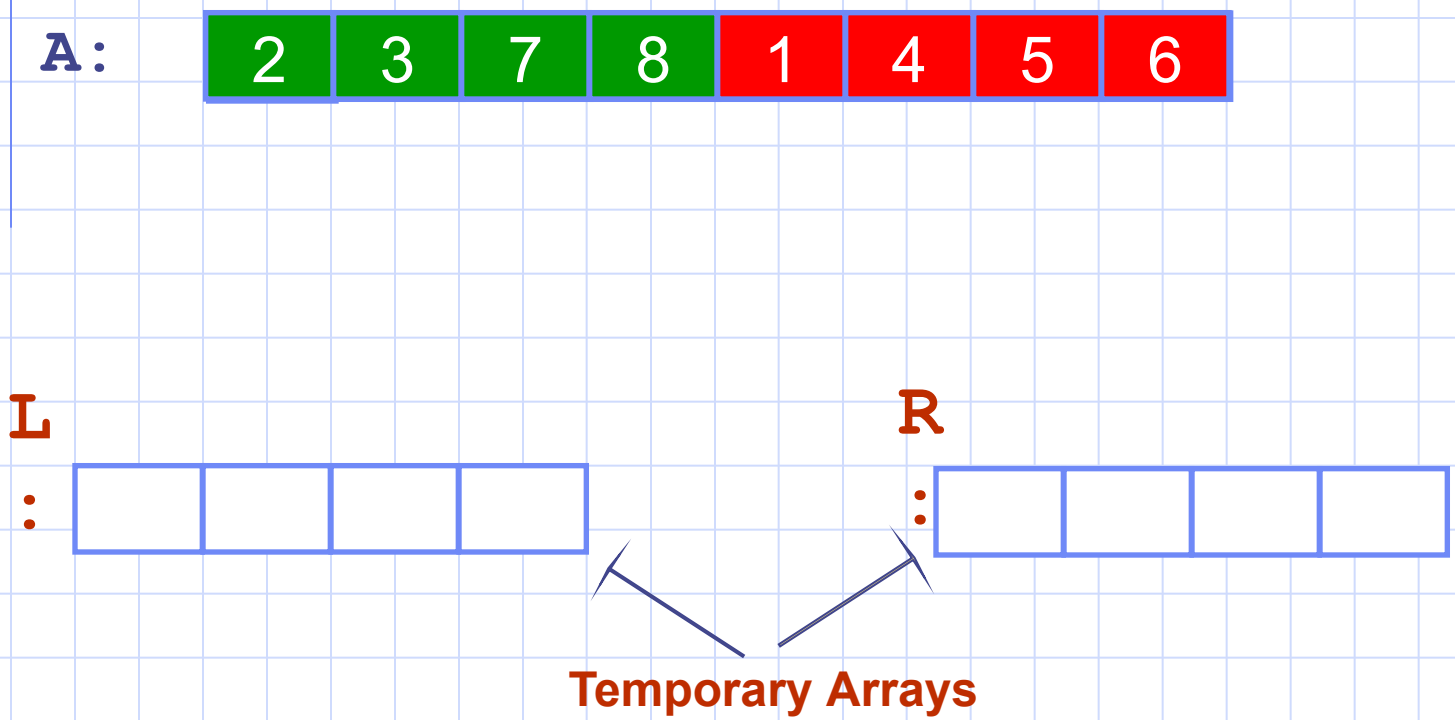

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67 98
Merge

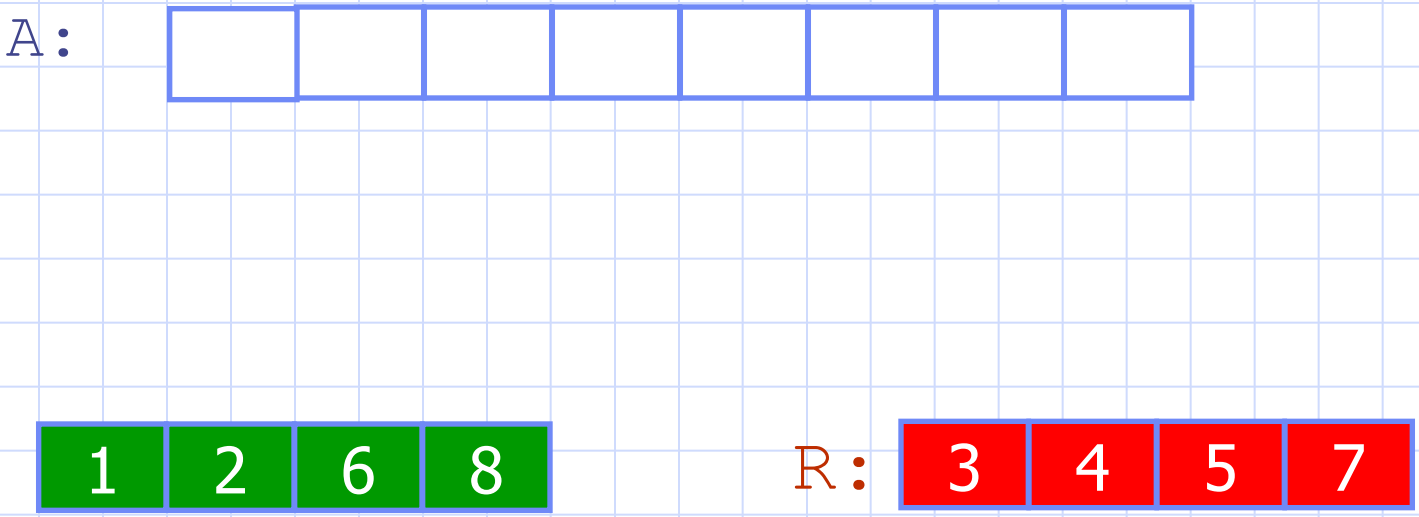

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67 98


98 23 45 14 6 67 33 42
6 14 23 33 42 45 67 98

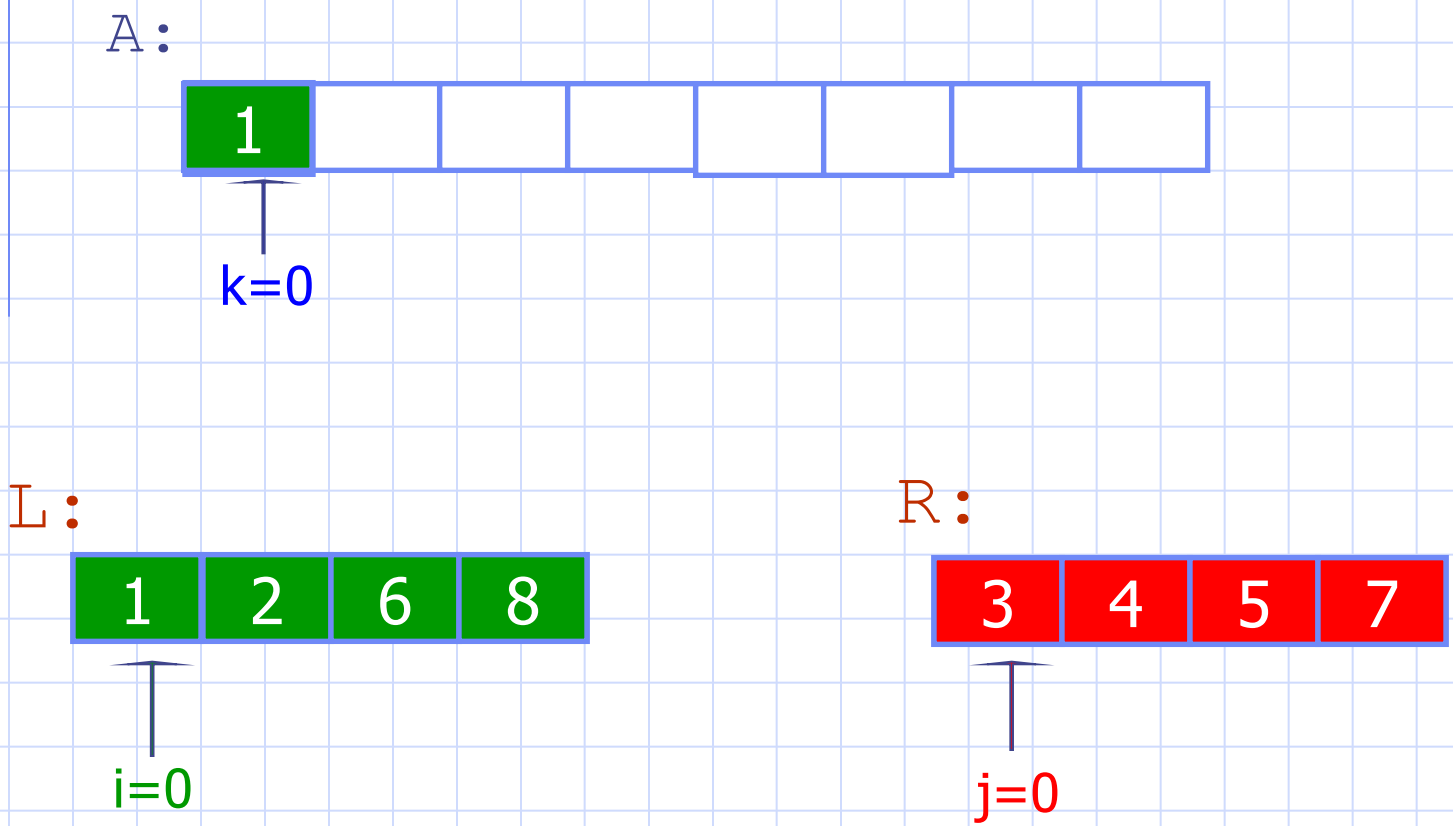
Merge-Sort: Merge Example



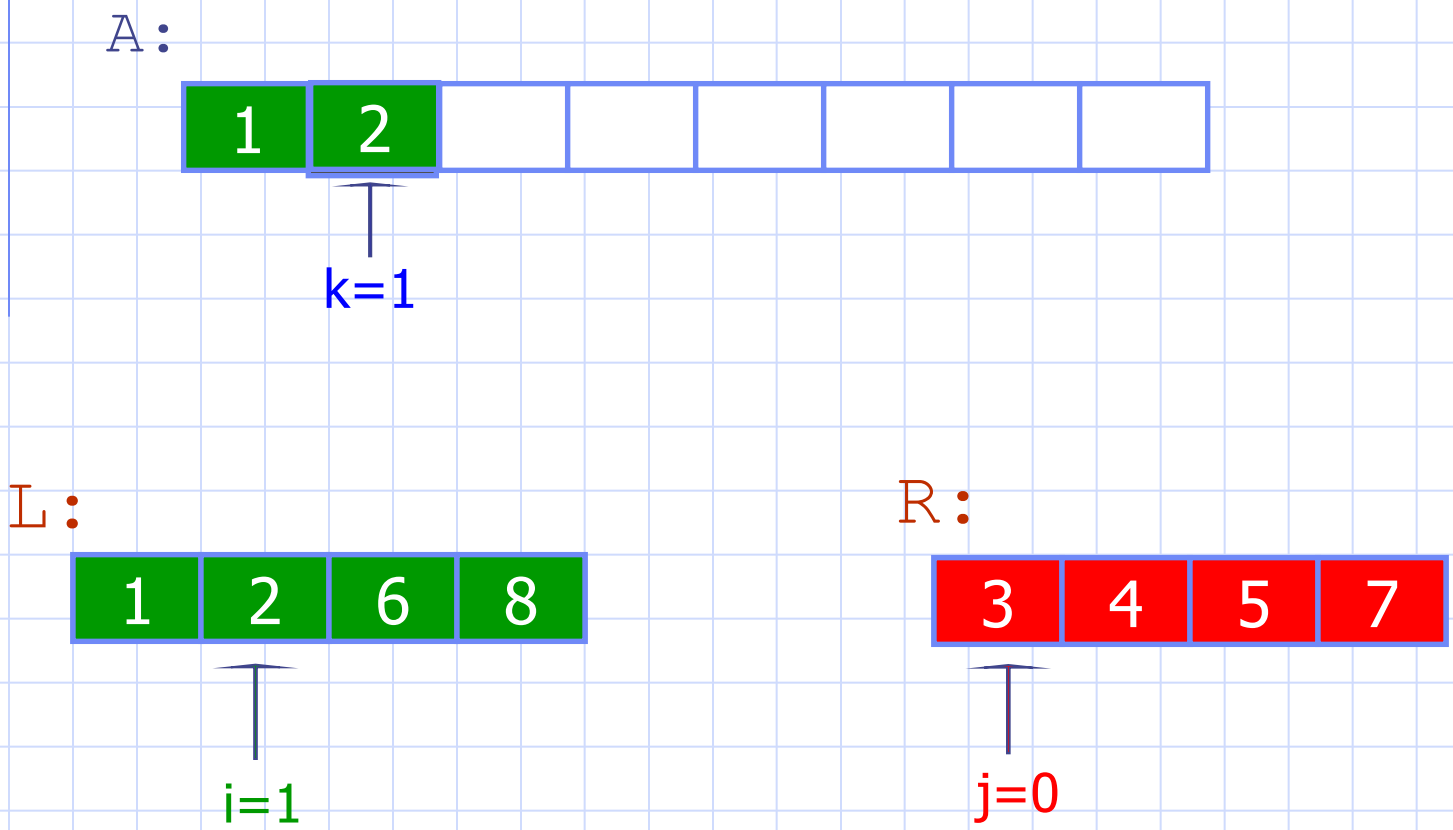
Merge-Sort: Merge Example



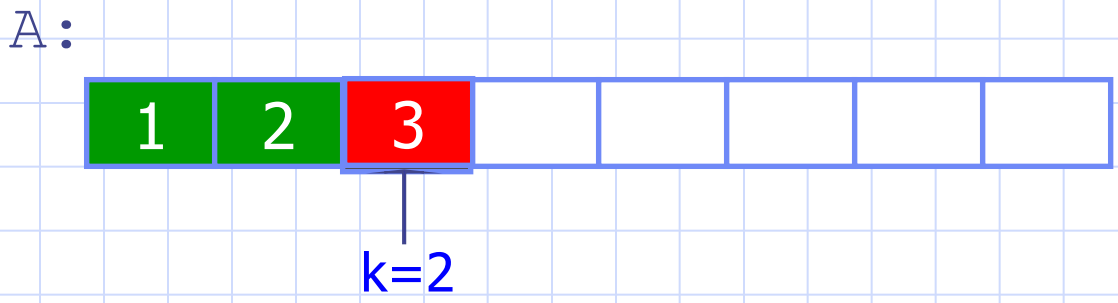
Merge-Sort: Merge Example

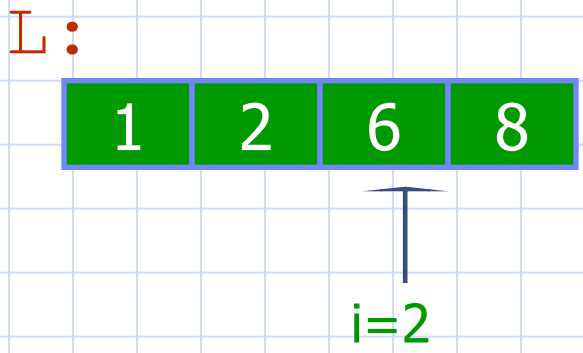


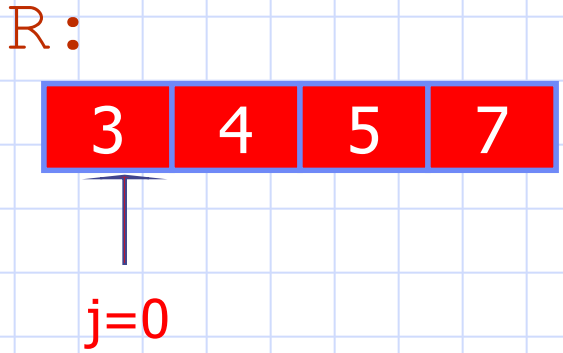
Merge-Sort: Merge Example



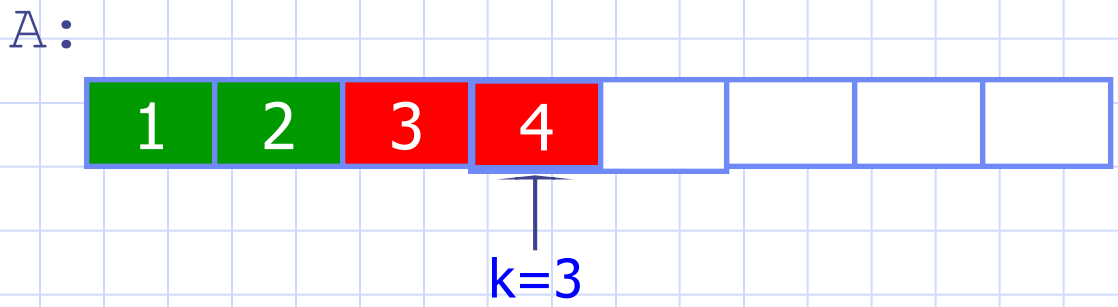
Merge-Sort: Merge Example

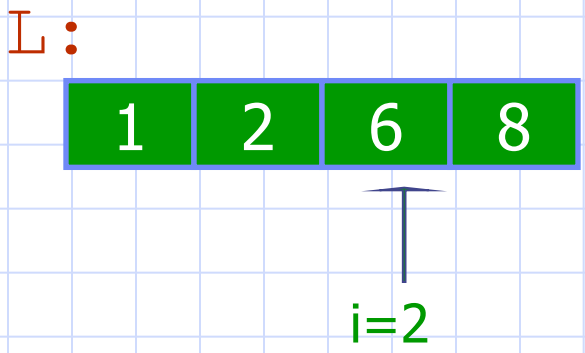


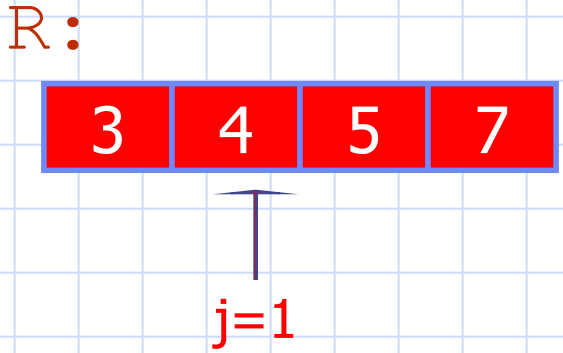




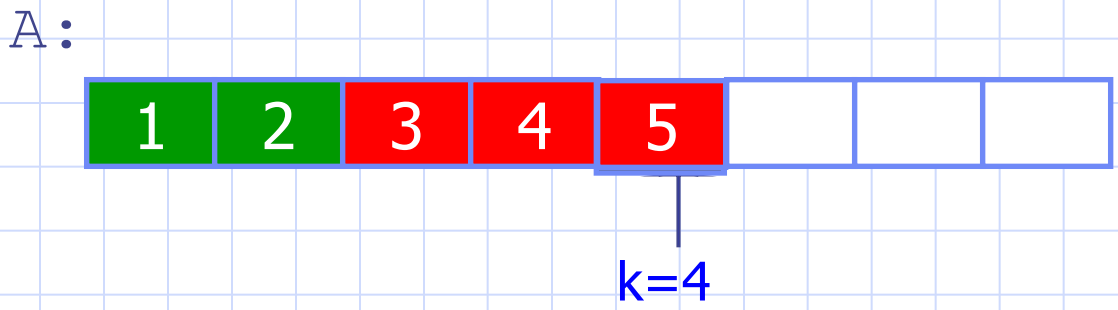
Merge-Sort: Merge Example

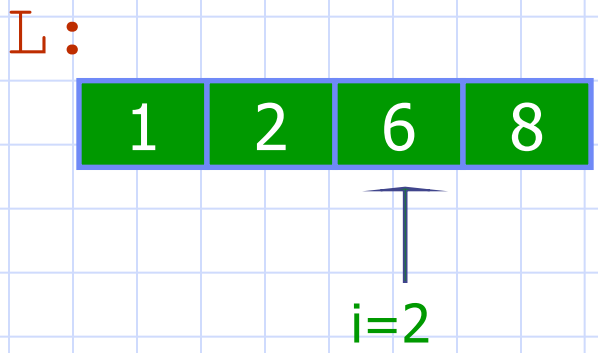


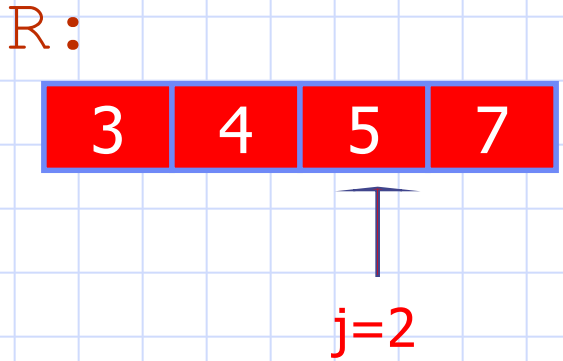




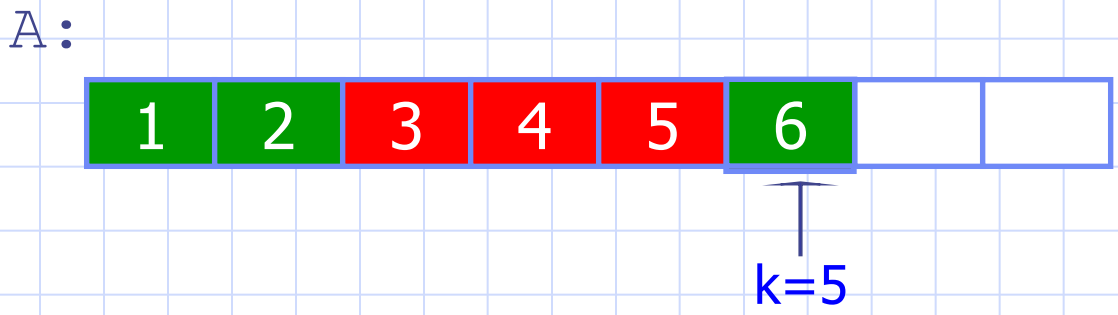
Merge-Sort: Merge Example

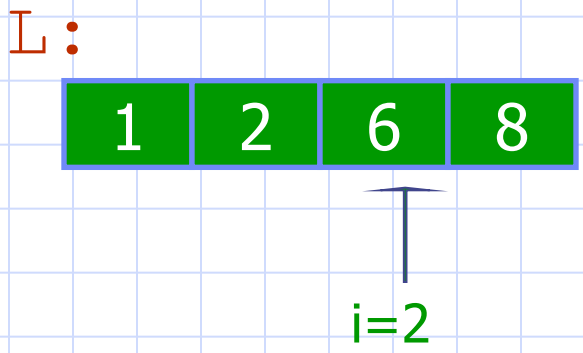


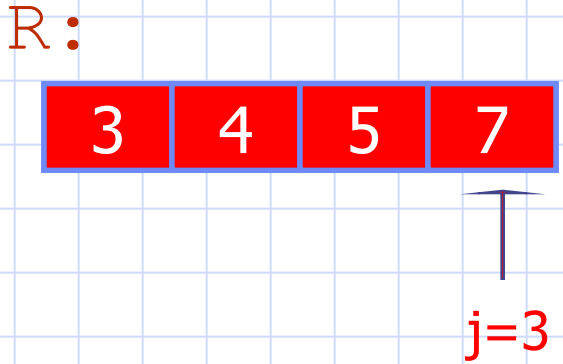




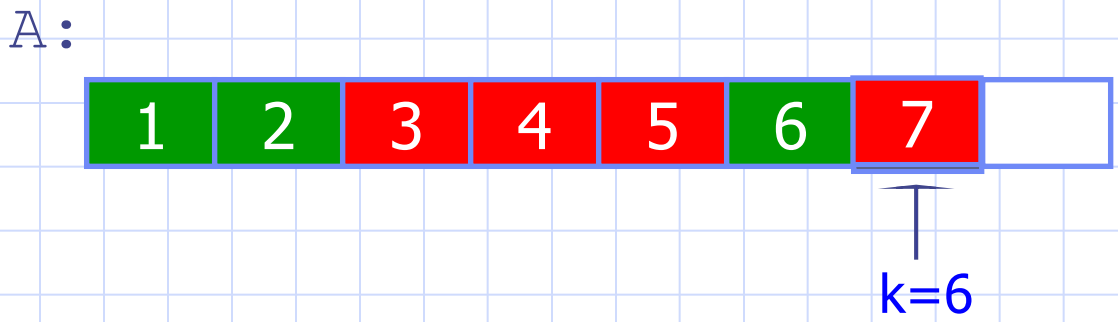
Merge-Sort: Merge Example

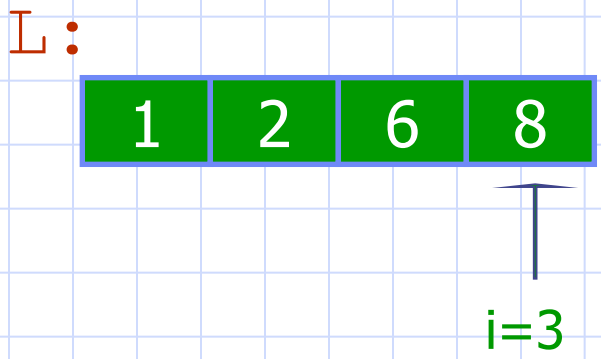


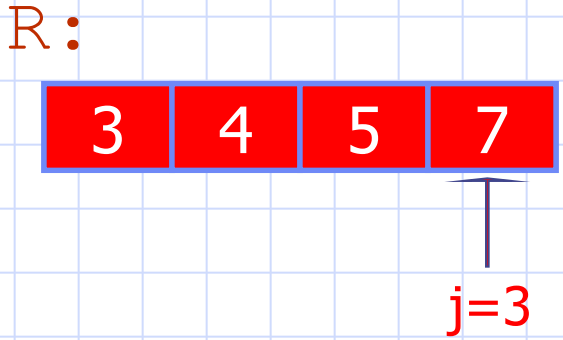




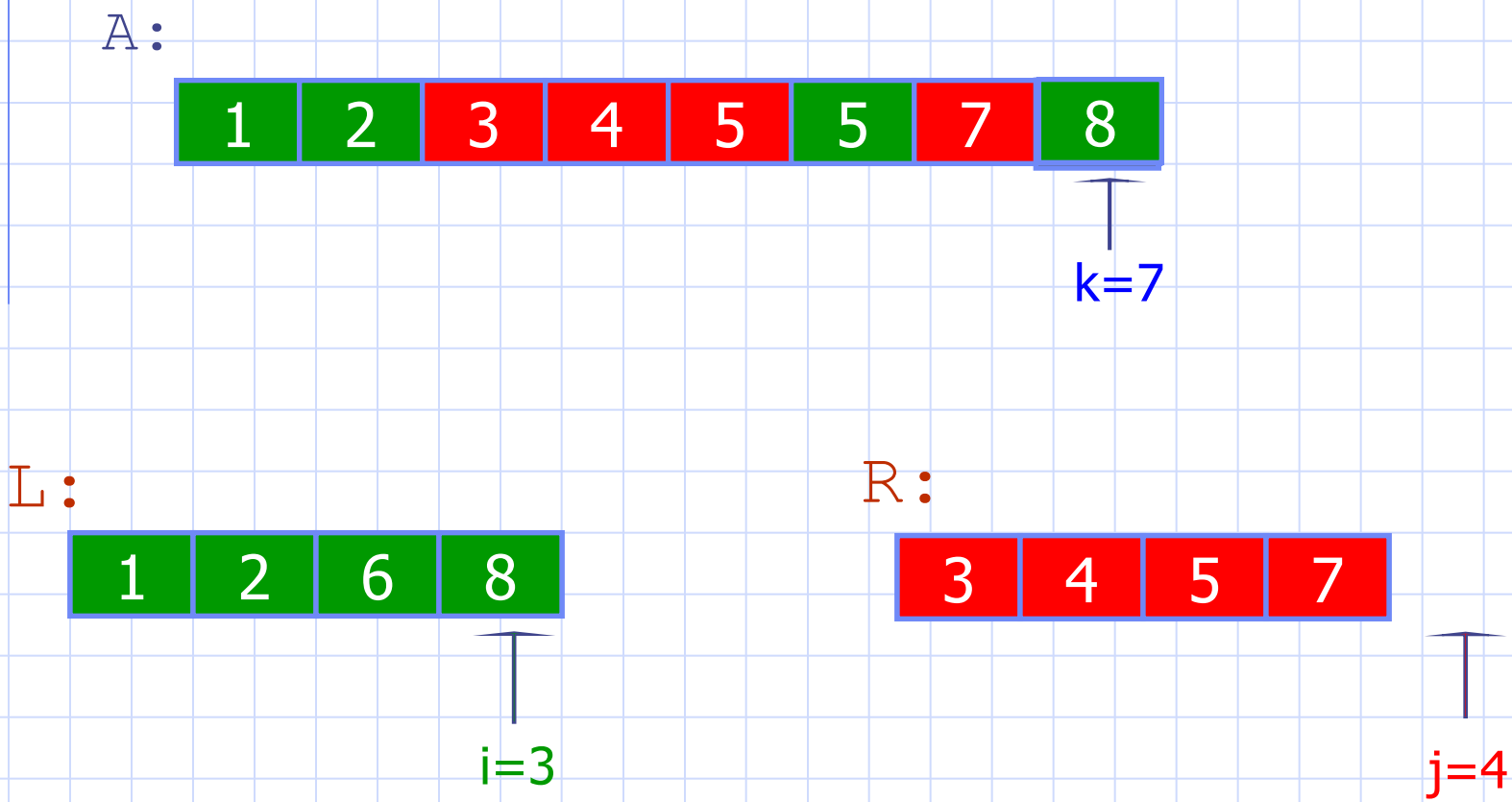
Merge-Sort: Merge Example



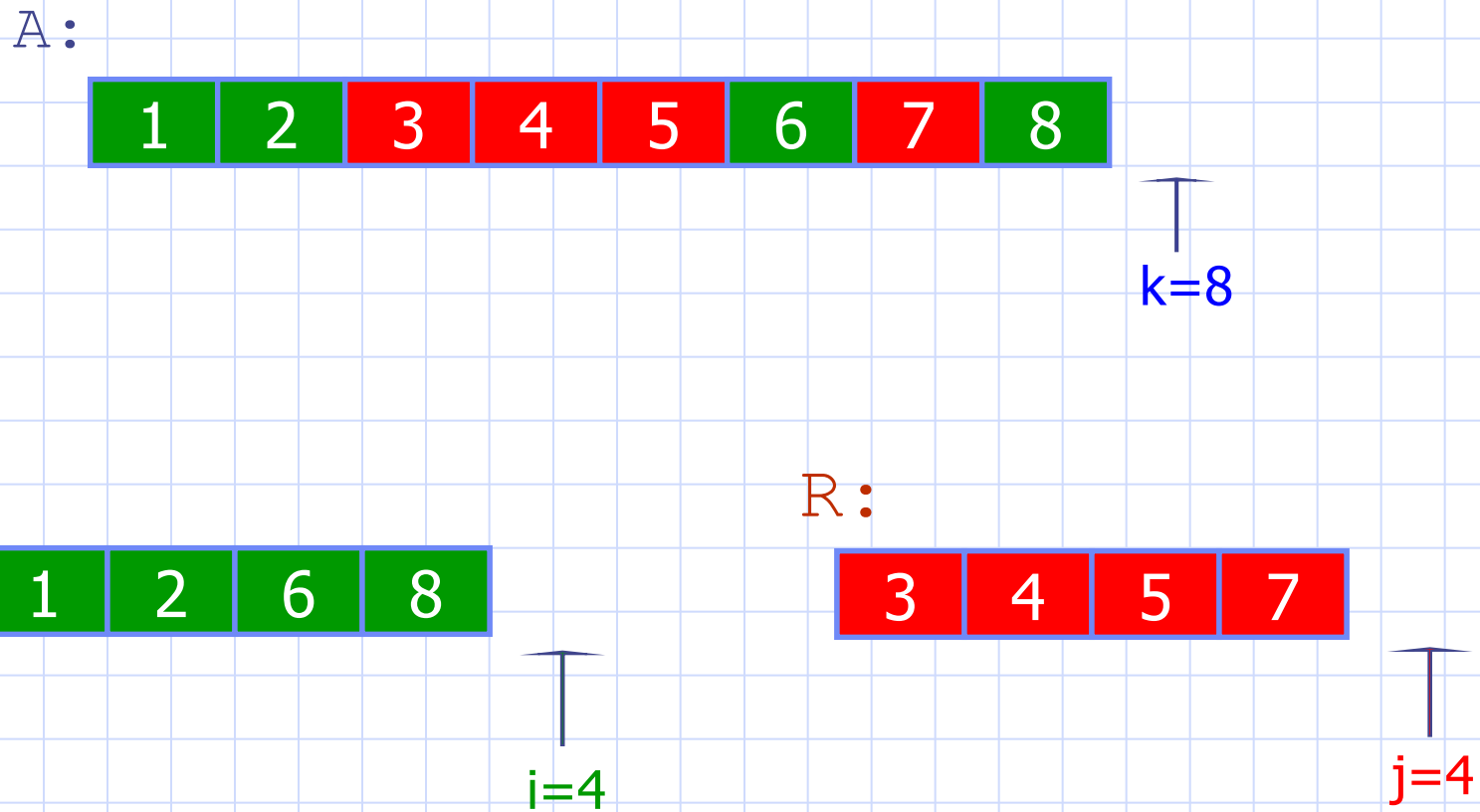




Merge-Sort: Merge Example



Merge-Sort: Merge Example



Summary

- Divide the unsorted collection into two
- Until the sub-arrays only contain one element
- Then merge the sub-problem solutions together

Analysis of Merge

- For two input sequences that contain a total of n elements, we need to move each element's input sequence to its output sequence
  - Merge time is O(n)
- We need to be able to store both initial sequences and the output sequence
  - The array cannot be merged in place
  - Additional space usage is O(n)

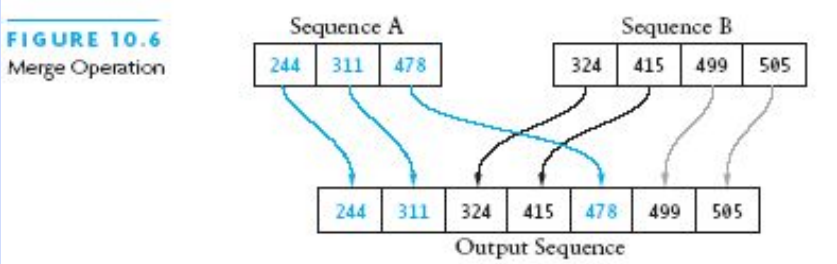
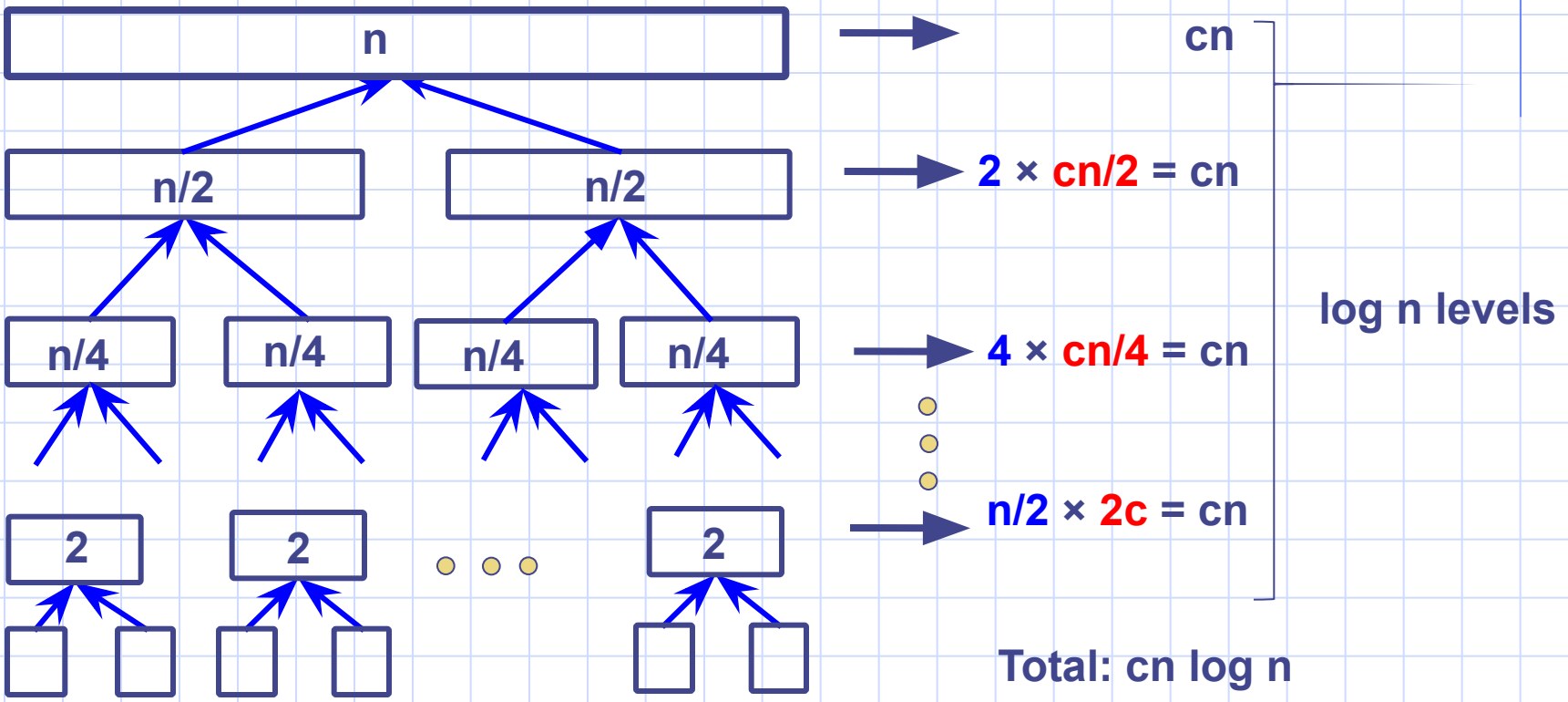


FIGURE 10.6
Merge Operation

Merge-Sort Analysis



- Total running time: $\Theta(n \log n)$
- Total Space: $\Theta(n)$

Quicksort

- Another divide-and-conquer algorithm
  - The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: 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

- Developed in 1962
- Quicksort rearranges an array into two parts so that all the elements in the left subarray are less than or equal to a specified value, called the pivot
- Quicksort ensures that the elements in the right subarray are larger than the pivot.

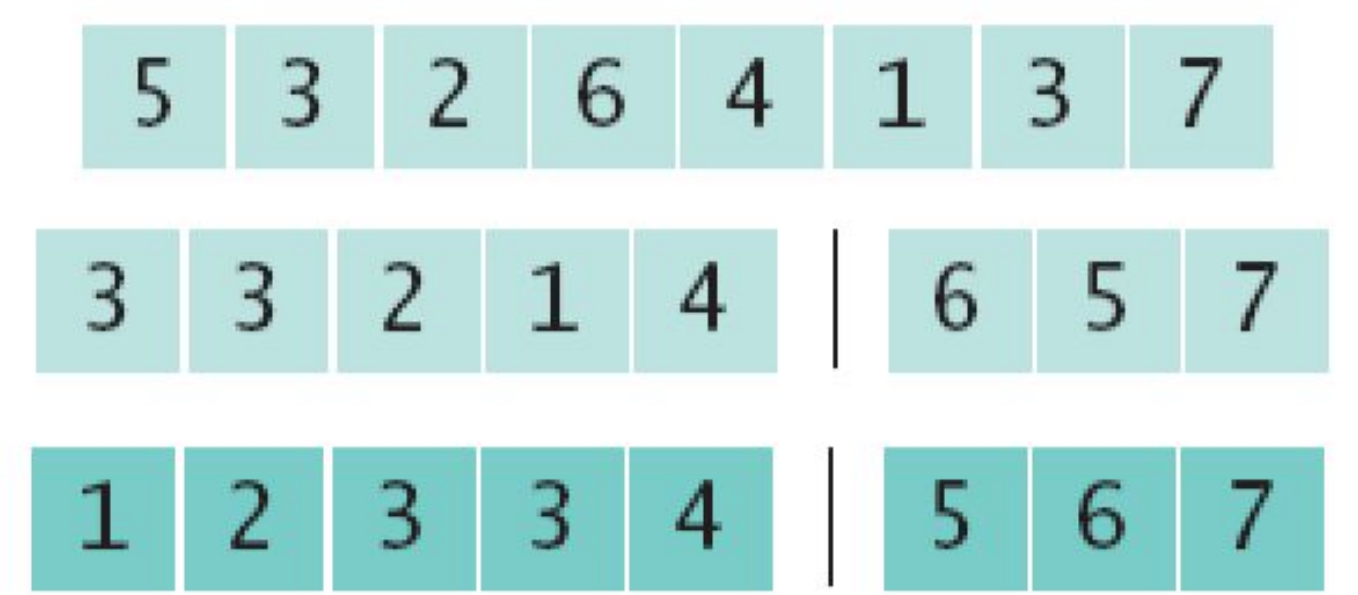
Partition In Words

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] – if(i<=j)
- Repeat until i >= j
- Return j

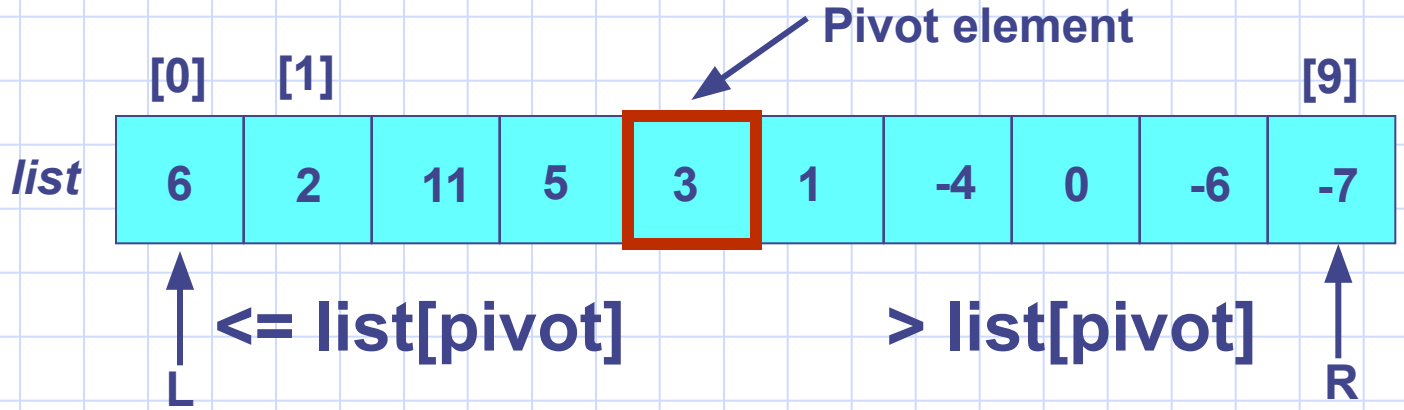
The Quick Sort Algorithm

- Divide and conquer
  1. Partition the range
  2. Sort each partition

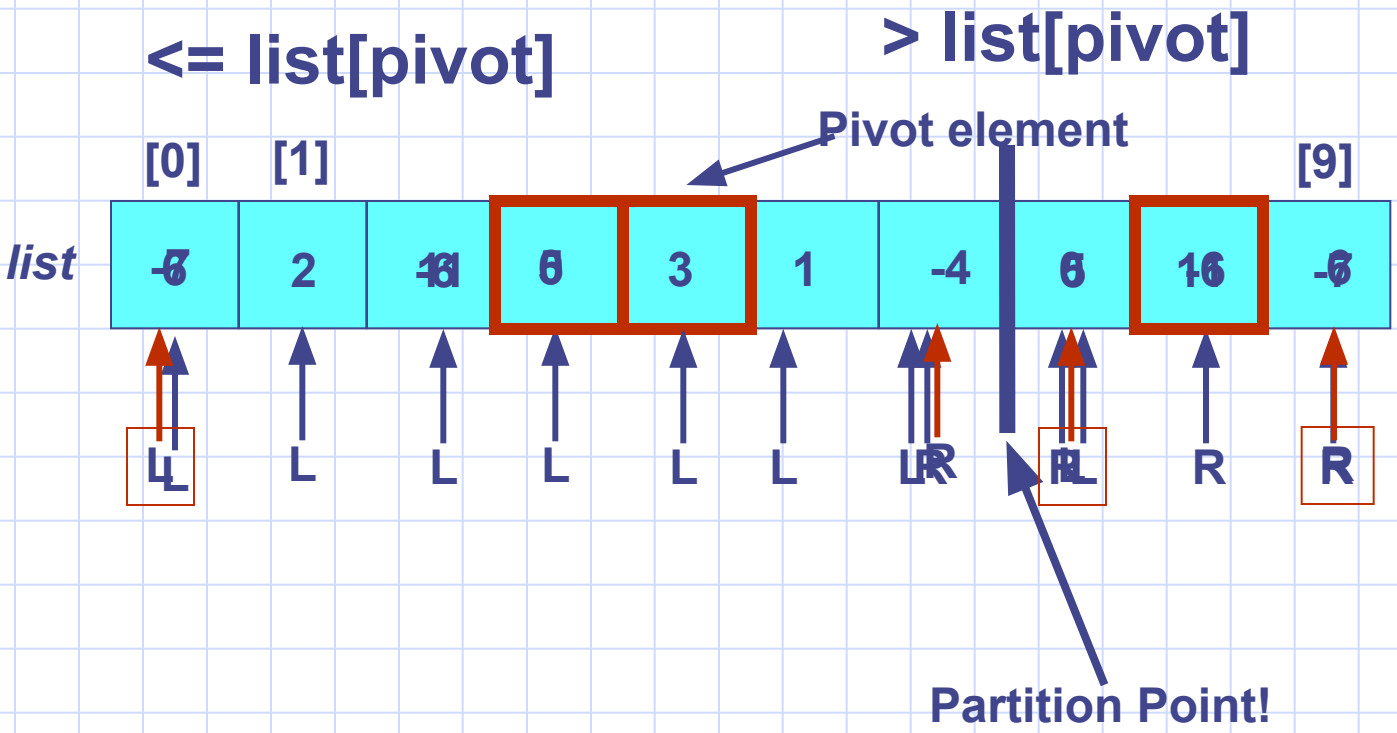


The Quick Sort Algorithm (Cont'd)

- **Quick Sort Steps**
  1. Select the pivot index = (L + R) / 2
  2. Keep on the left partition all elements <= list[pivot]
  3. Keep on the right partition all elements > list[pivot]



The Quick Sort Algorithm (Cont'd)



Algorithm of Quick sort

Q_sort(array, first,last)

Where array= Represents the list of elements

first= Represents the position of the first element in the list (Only at the starting point, its value changes during the execution of the the function)

last= Represents the position of the last element in the list (Only at the starting point, its value changes during the execution of the function)

Step 1: [Initially]
low=first
high=last
pivot=array(low+high)/2 [Middle element of the list]
Step 2: Repeat through step 7 while (low<= high)
Step 3: Repeat step 4 while (array[low] < pivot)
Step 4: low=low+1
Step 5: Repeat step 6 while (array[high] > pivot)
Step 6: high=high -1
Step 7: if (low<=high)
(i) temp= array[low]
(ii) array[low] array[high]
(iii) array[high]=temp
(iv) low=low+1
(v) high= high-1
Step 8: if (first<high)
Q_sort(array, first,high)
Step 9: if (low<last)
Q_sort(array, low,last)
Step 10: exit
quicksort

Quicksort

- The performance of quicksort depends critically on the quality of the pivot.
- Sorts in place
- Sorts O(n log n) in the average case
- Sorts $O(n^2)$ in the worst case
  - But in practice, it's quick
  - And the worst case doesn't happen often (but more on this later…)

Analyzing Quicksort

- What will be the worst case for the algorithm?
  - Partition is always unbalanced
- What will be the best case for the algorithm?
  - Partition is perfectly balanced
- Will any particular input elicit the worst case?
  - Yes: Already-sorted input

The End