Sorting algorithms

Insertion Sort, Bubble sort, Selection sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, and Bucket Sort, Radix sort, Inversions, External sorting.

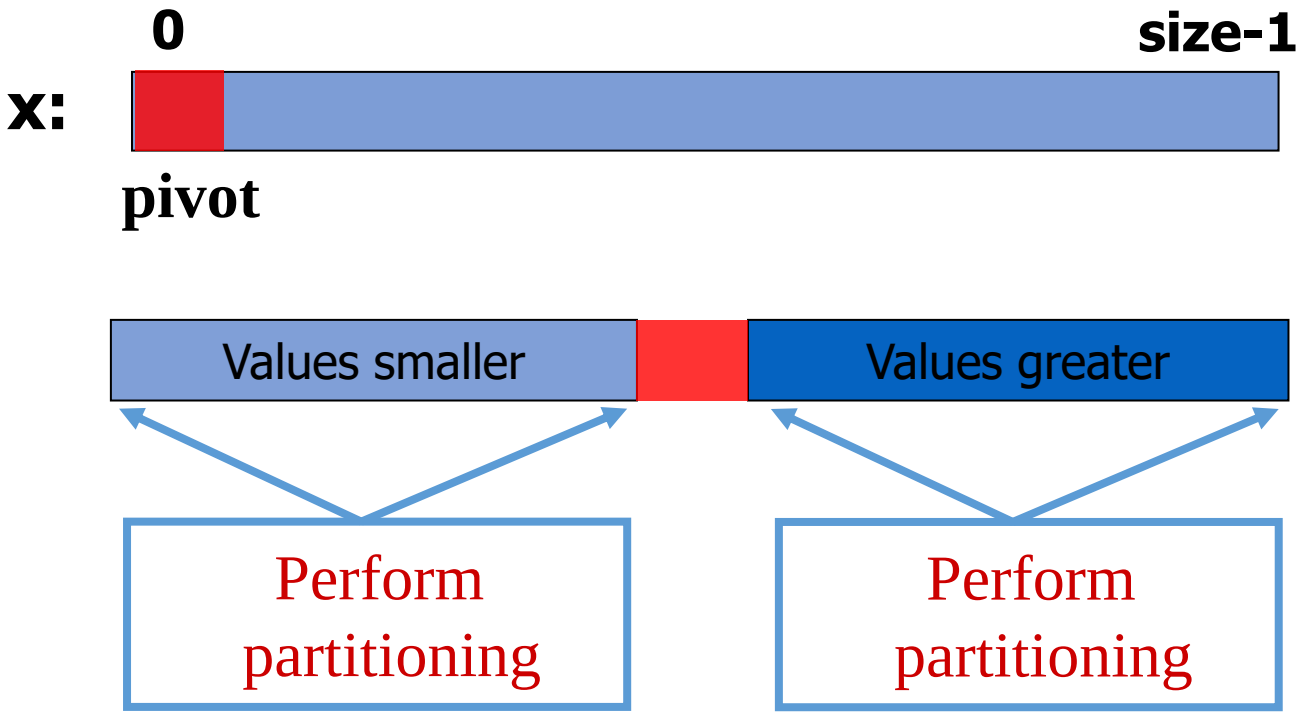
Quick Sort

Quick Sort – How it Works?

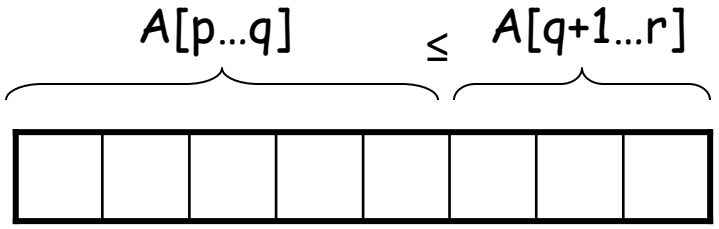
At every step, we select a *pivot element* in the list (usually the first element).

- We put the pivot element in the *final position* of the sorted list.
- All the elements less than or equal to the pivot element are to the left.
- All the elements greater than the pivot element are to the right.

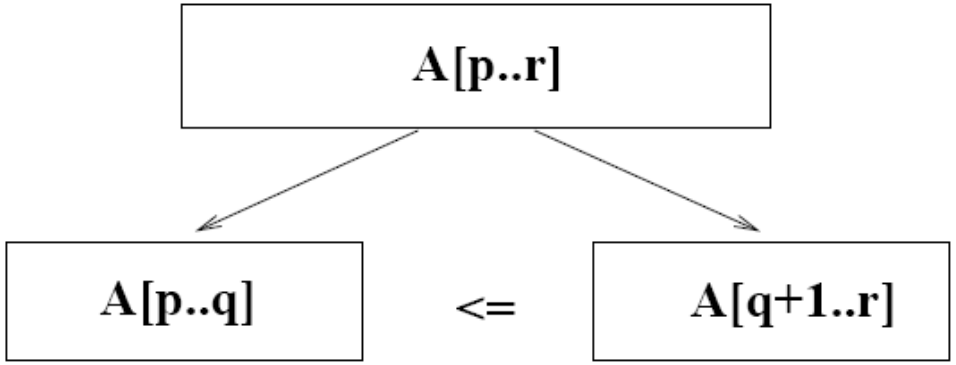
Quick Sort Partitioning



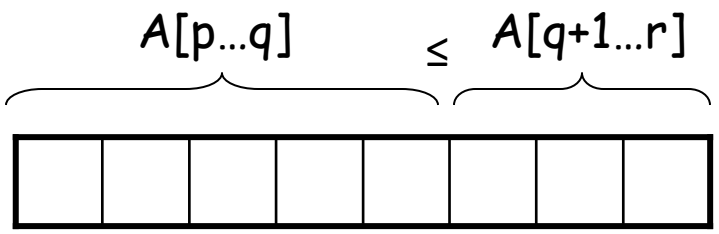
Quicksort



- Sort an array $A[p…r]$
- **Divide**
  - Partition the array A into 2 subarrays $A[p..q]$ and $A[q+1..r]$, such that each element of $A[p..q]$ is smaller than or equal to each element in $A[q+1..r]$
  - Need to find index q to partition the array



Quicksort



- **Conquer**
  - Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

Algorithm/Procedure / Flowchart

```
// Sorts a (portion of an) array, divides it into partitions, then sorts those
algorithm quicksort(A, lo, hi)
  // Ensure indices are in correct order
  if lo >= hi || lo < 0 then
    return

  // Partition array and get the pivot index
  p := partition(A, lo, hi)

  // Sort the two partitions
  quicksort(A, lo, p - 1) // Left side of pivot
  quicksort(A, p + 1, hi) // Right side of pivot
```

```
algorithm partition(A, lo, hi)
// Pivot value
  pivot := A[ floor((hi + lo) / 2) ] // The value in the middle of the array

  // Left index
  i := lo - 1

  // Right index
  j := hi + 1

  loop forever
    // Move the left index to the right at least once and while the element at
    // the left index is less than the pivot
    do i := i + 1 while A[i] < pivot and i<=j

    // Move the right index to the left at least once and while the element at
    // the right index is greater than the pivot
    do j := j - 1 while A[j] > pivot

    // If the indices crossed, return
    if i >= j then return j

    // Swap the elements at the left and right indices
    swap A[i] with A[j]
```

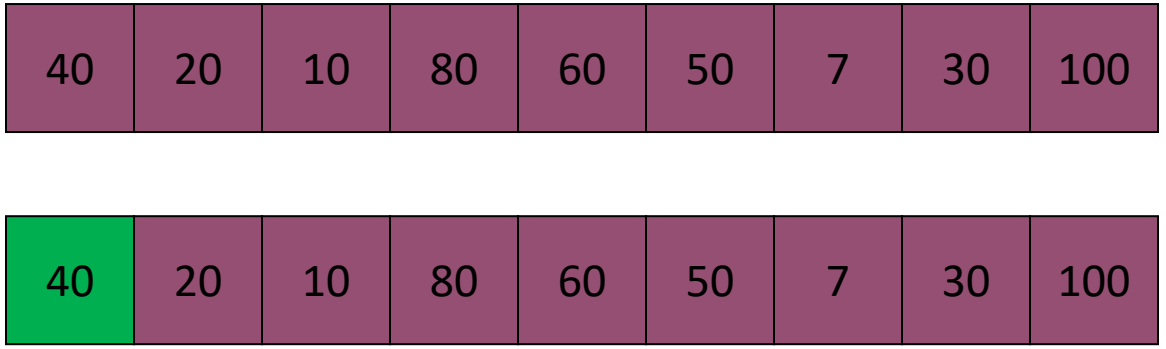
Quick Sort

```
#include <stdio.h>
void quickSort( int[], int, int);
int partition( int[], int, int);
void main()
{
   int i,a[] = { 7, 12, 1, -2, 0, 15, 4, 11, 9};
   printf("\n\nUnsorted array is: ");
   for(i = 0; i < 9; ++i)
      printf(" %d ", a[i]);
   quickSort( a, 0, 8);
   printf("\n\nSorted array is: ");
   for(i = 0; i < 9; ++i)
      printf(" %d ", a[i]);
}
void quickSort( int a[], int l, int r)
{
   int j;
   if( l < r ) {  // divide and conquer
   j = partition( a, l, r);
   quickSort( a, l, j-1);
   quickSort( a, j+1, r);
   }
}
```

Quick Sort

```
int partition( int a[], int l, int r)
{
   int pivot, i, j, t;
   pivot = a[l];
   i = l;
   j = r+1;
   while( 1) {
          do {
              ++i;
             } while(a[i]<=pivot && i<=r);
          do {
              --j;
             } while( a[j] > pivot );
          if( i >= j ) break;
          t = a[i];
          a[i] = a[j];
          a[j] = t;
            }
   t = a[l];
   a[l] = a[j];
   a[j] = t;
   return j;
}
```

```
int partition( int a[], int l, int r)
{
   int pivot, i, j, t;
   pivot = a[l];
   i = l;
   j = r+1;
   while( 1) {
          do {
              ++i;
             } while(a[i]<=pivot && i<=r);
          do {
              --j;
             } while( a[j] > pivot );
          if( i >= j ) break;
          t = a[i];
          a[i] = a[j];
          a[j] = t;
            }
   t = a[l];
   a[l] = a[j];
   a[j] = t;
   return j;
}
```

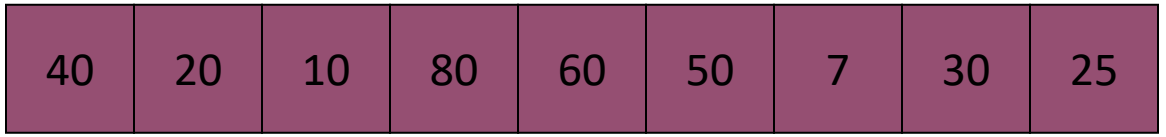


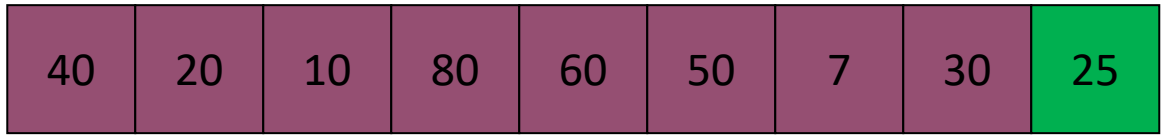
Quick Sort

```python
# function to perform quicksort

def quickSort(array, low, high):
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)

        # Recursive call on the left of pivot
        quickSort(array, low, pi - 1)

        # Recursive call on the right of pivot
        quickSort(array, pi + 1, high)

data = [1, 7, 4, 1, 10, 9, -2]
print("Unsorted Array")
print(data)

size = len(data)

quickSort(data, 0, size - 1)

print('Sorted Array in Ascending Order:')
print(data)
```

```python
# Function to find the partition position
def partition(array, low, high):

    # choose the rightmost element as pivot
    pivot = array[high]

    # pointer for greater element
    i = low - 1

    # traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:

            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])

    # Return the position from where partition is done
    return i + 1
```

Quick Sort






```
# Function to find the partition position
def partition(array, low, high):


    # choose the rightmost element as pivot
    pivot = array[high]


    # pointer for greater element
    i = low - 1


    # traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:


            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1


            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])


    # Swap the pivot element with the greater element
specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])


    # Return the position from where partition is done
    return i + 1
```

Quick Sort - Example

Input: 45 -56 78 90 -3 -6 123 0 -3 45 69 68

45 -56 78 90 -3 -6 123 0 -3 45 69 68

0 -56 -45 -3 -3 -6 45 123 90 78 45 69 68

68 90 78 45 69 123

45 68 78 90 69

69 78 90

Output: -56 -6 -3 -3 0 45 45 68 69 78 90 123

```
int partition( int a[], int l, int r)
{
   int pivot, i, j, t;
   pivot = a[l];
   i = l;
   j = r+1;
   while( 1) {
          do {
              ++i;
             } while(a[i]<=pivot && i<=r);
          do {
              --j;
             } while( a[j] > pivot );
          if( i >= j ) break;
          t = a[i];
          a[i] = a[j];
          a[j] = t;
           }
   t = a[l];
   a[l] = a[j];
   a[j] = t;
   return j;
}
```

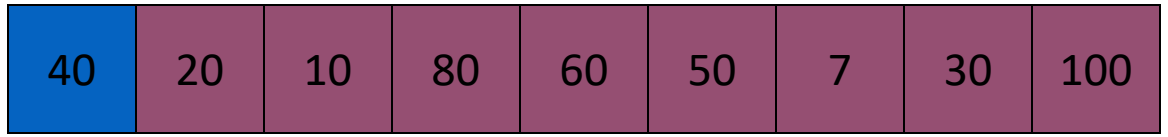
Step by step Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:



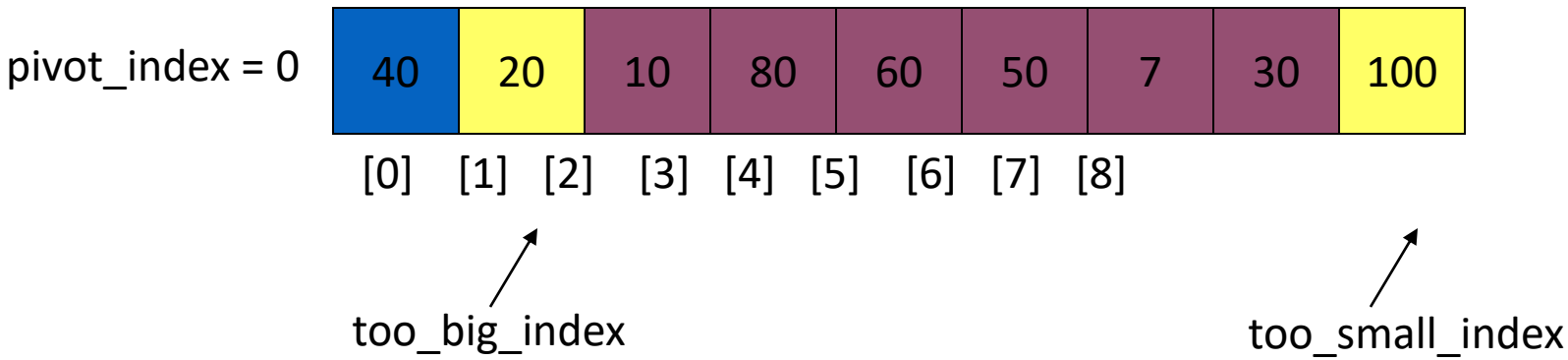
Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

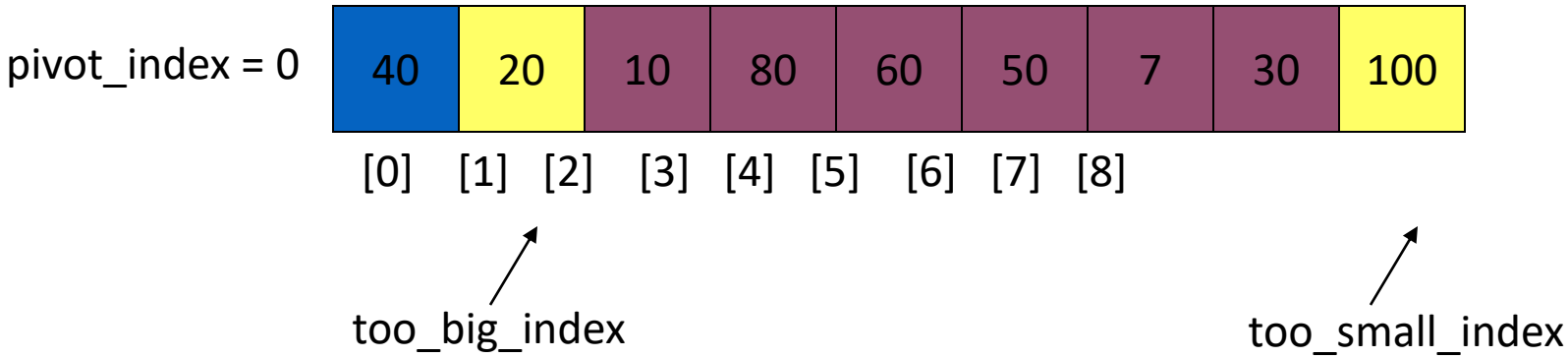
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

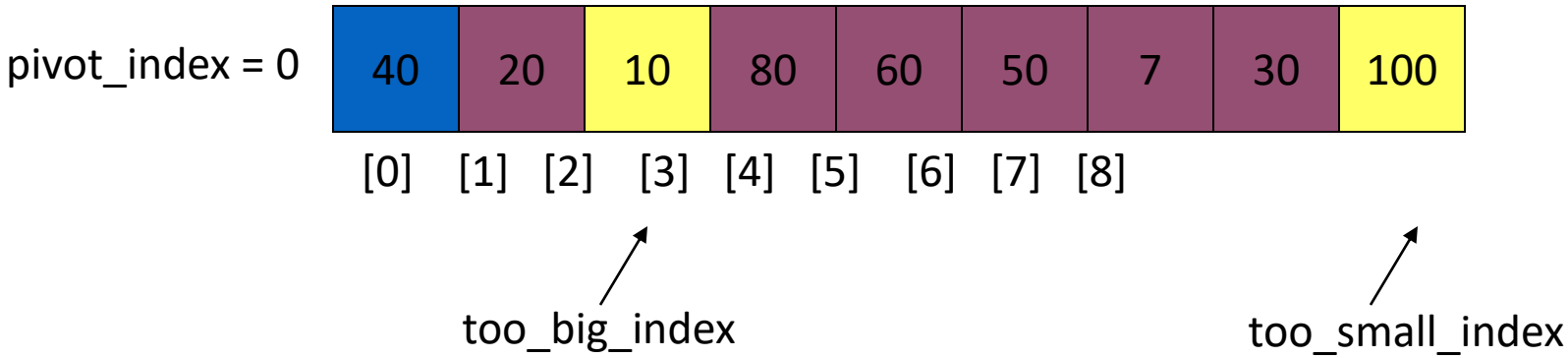
Partitioning loops through, swapping elements below/above pivot.


pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

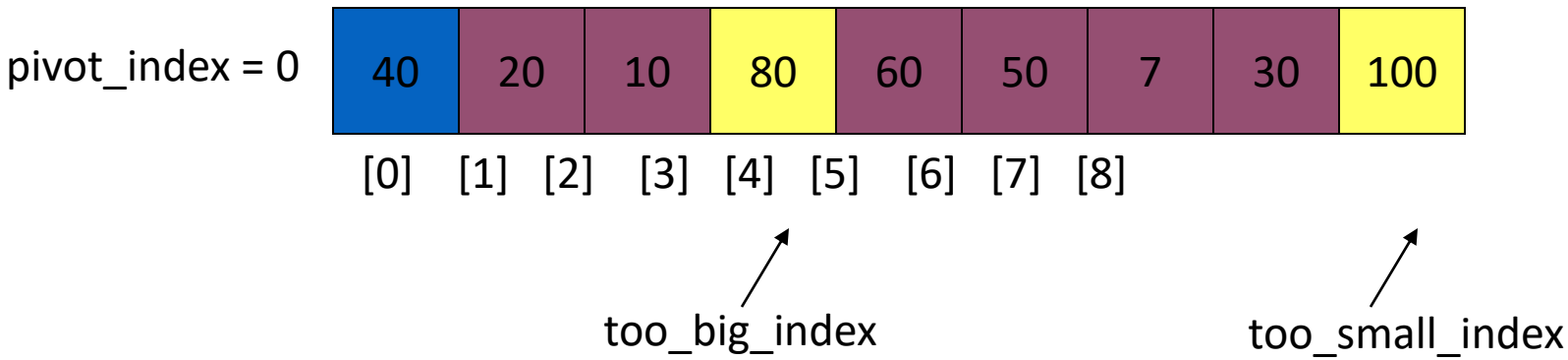
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



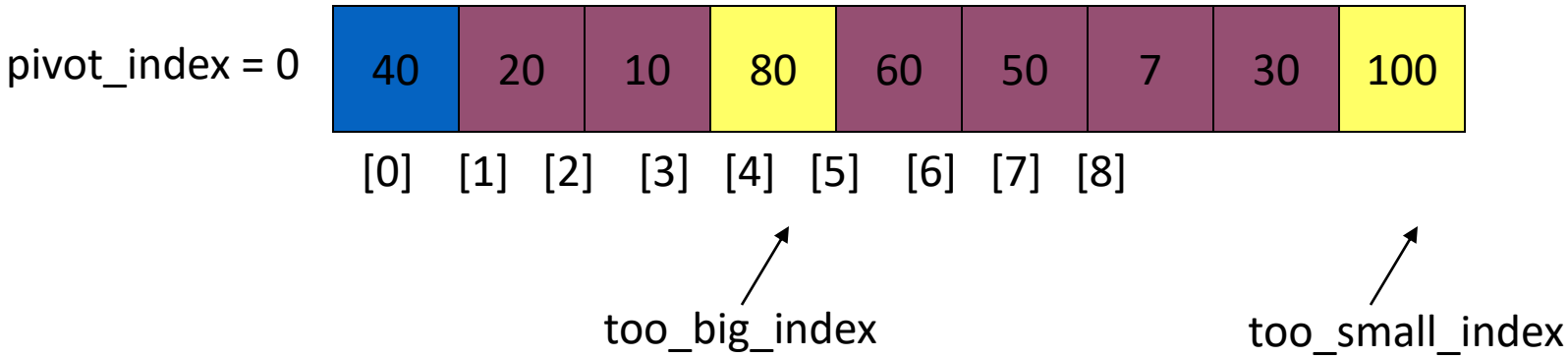
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



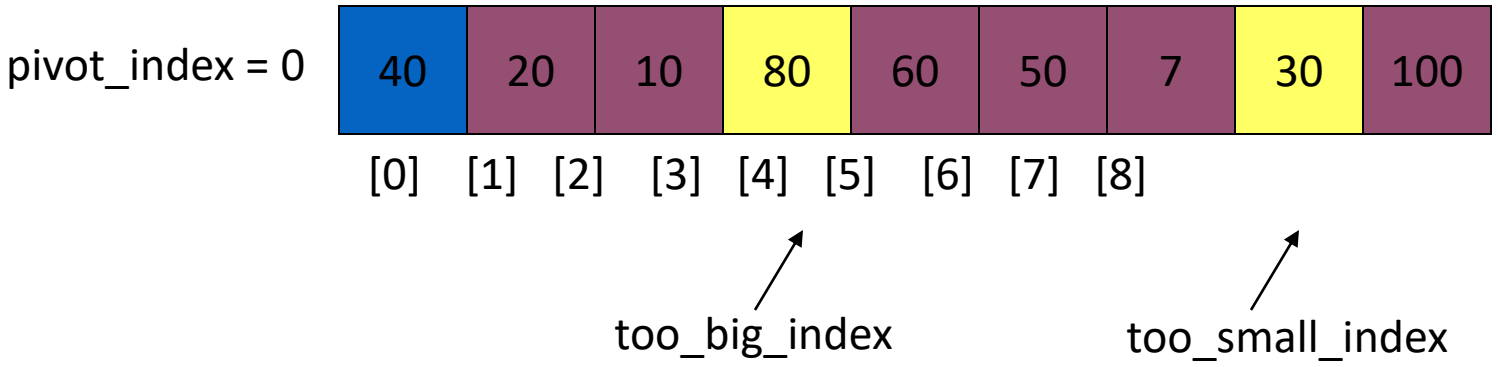
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



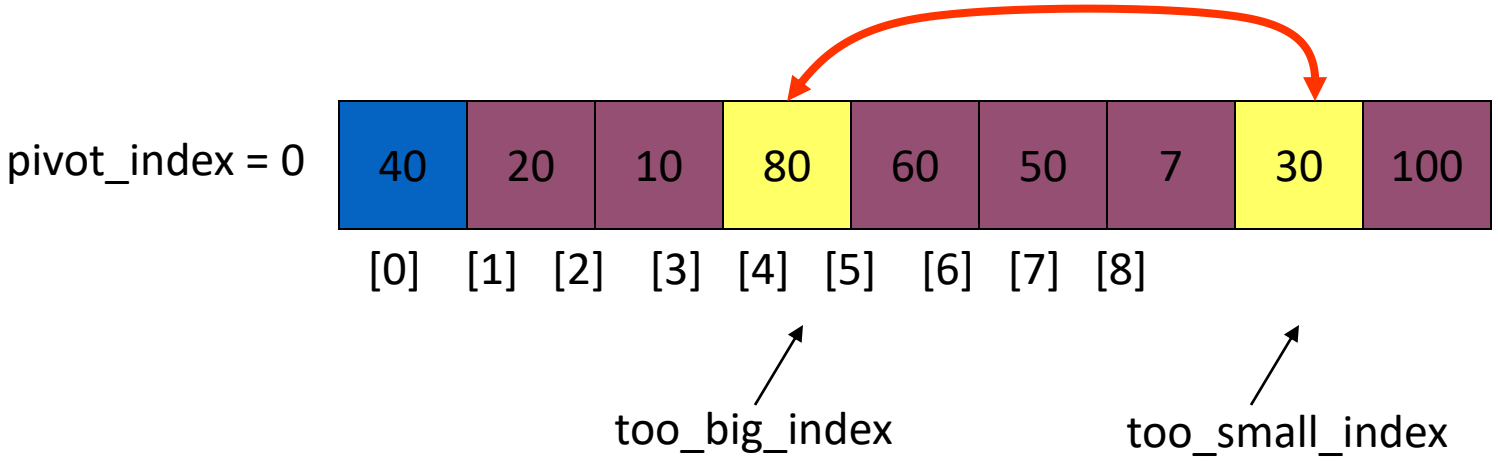
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



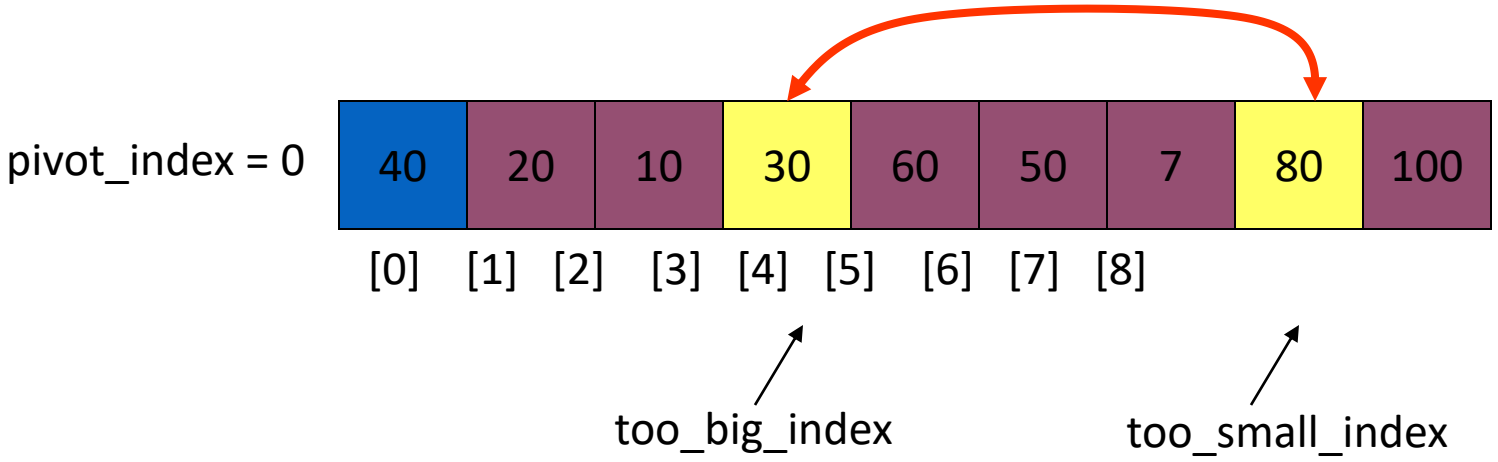
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



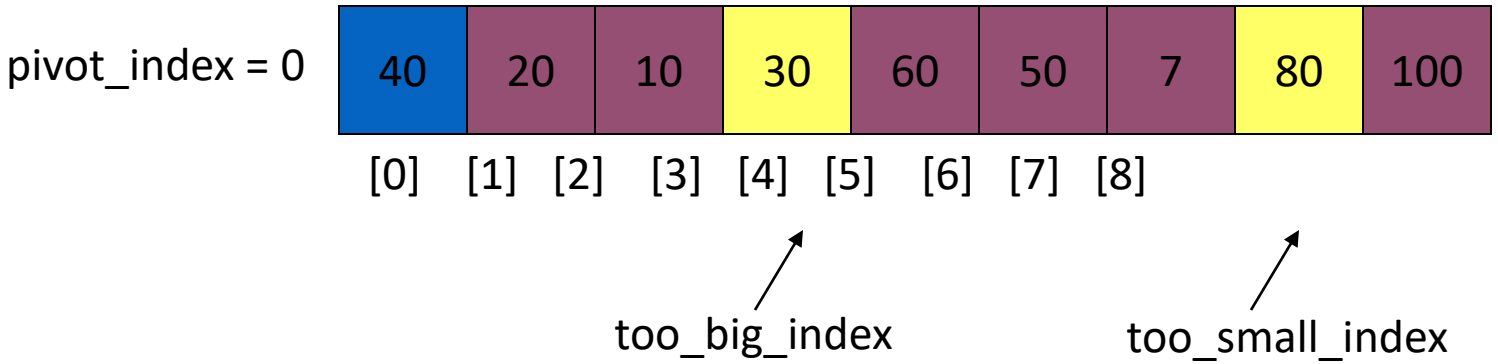
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



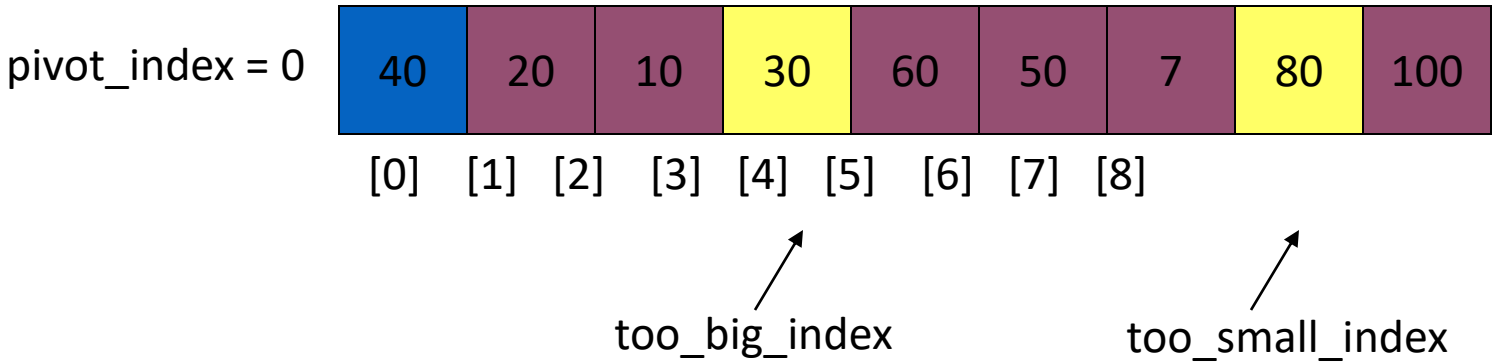
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



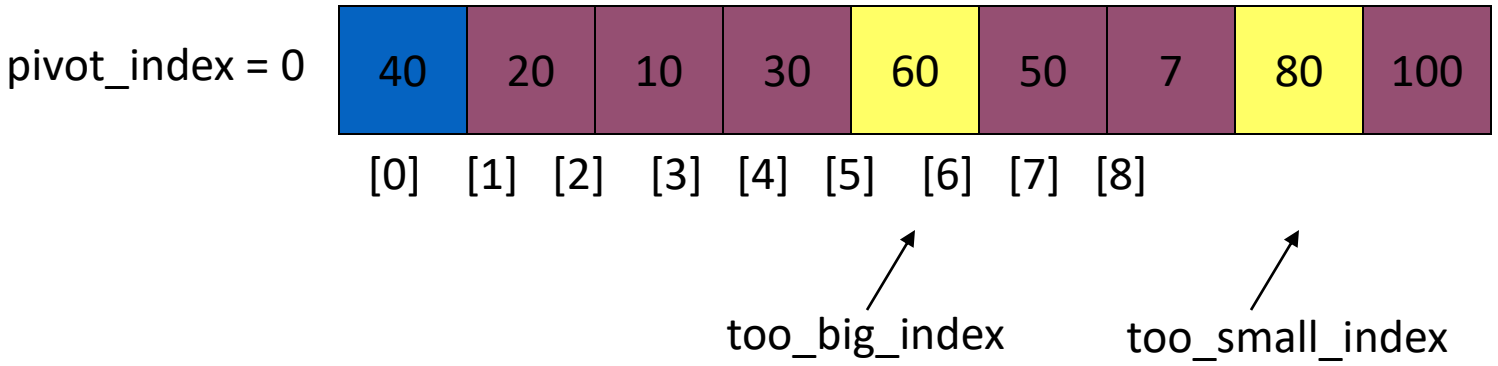
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



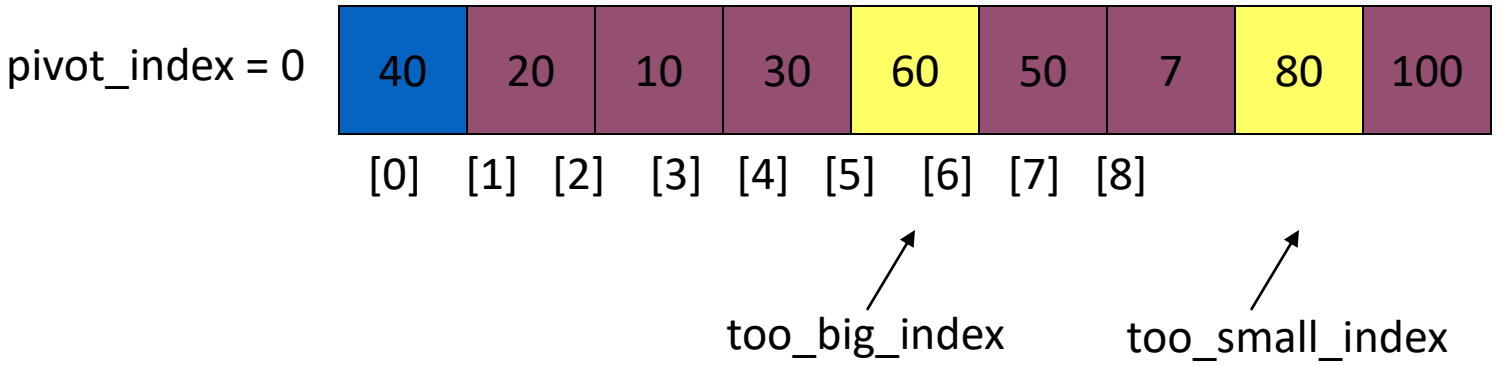
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



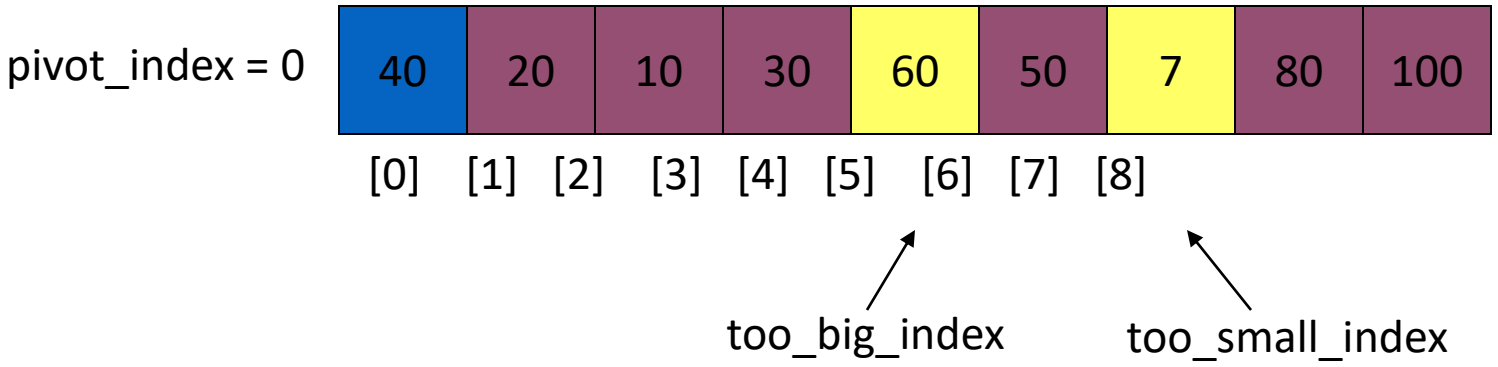
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



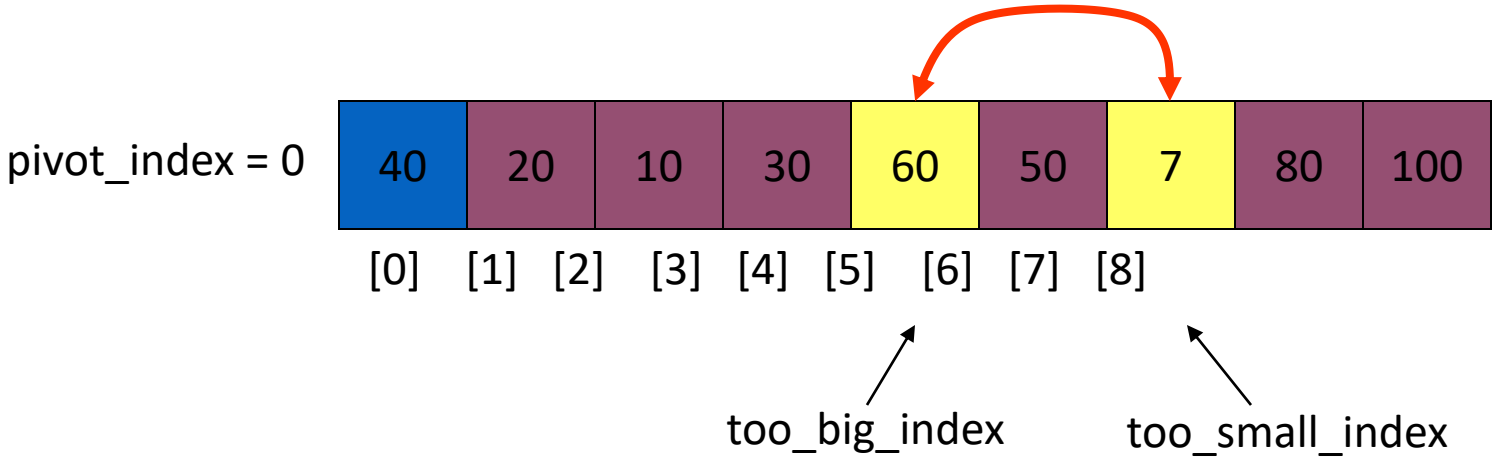
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



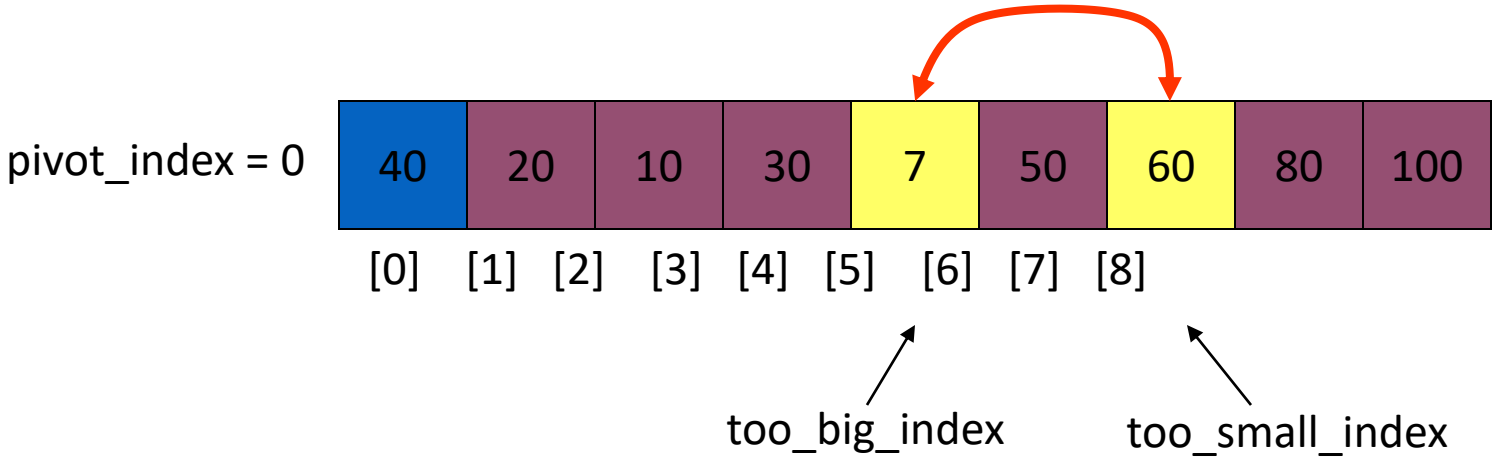
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



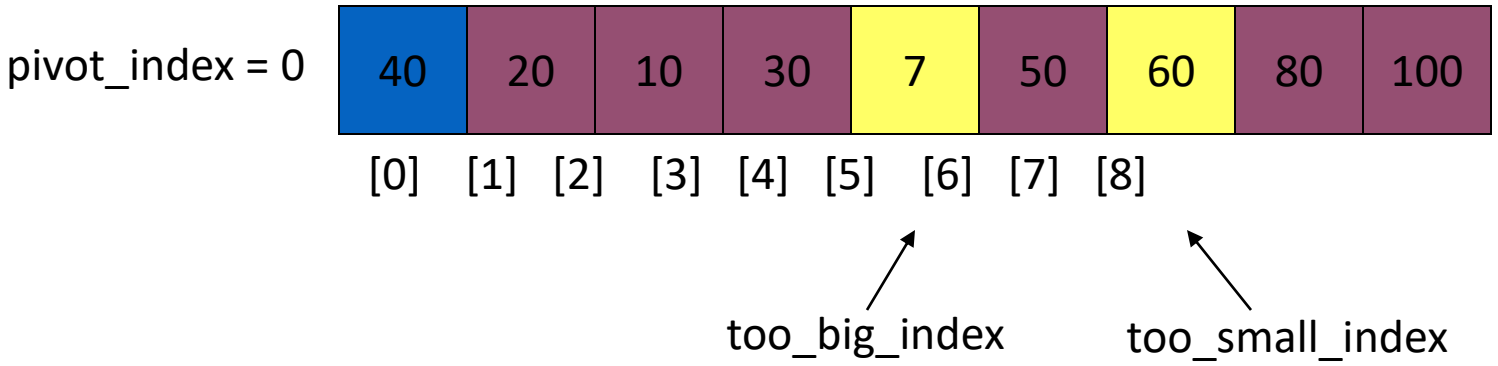
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



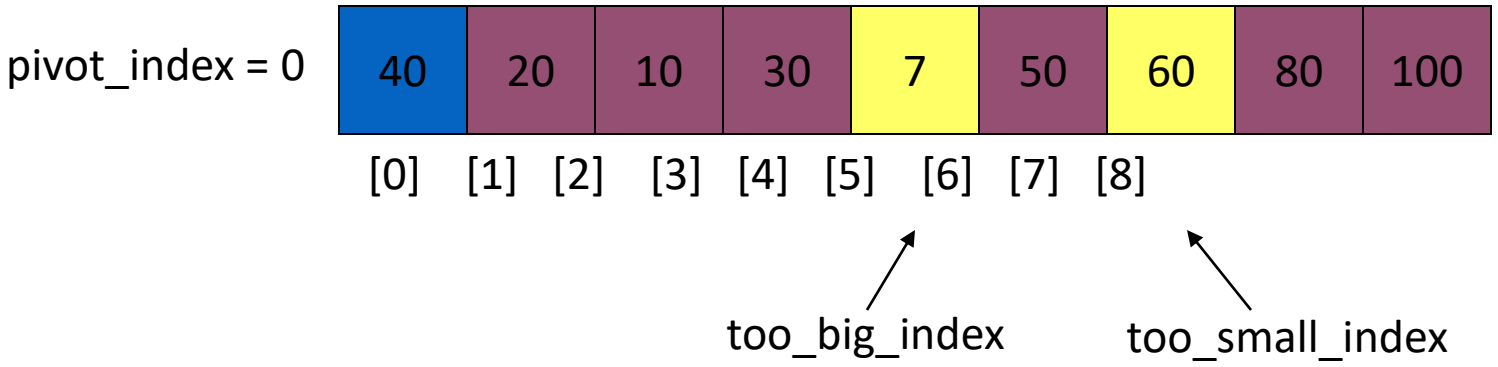
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



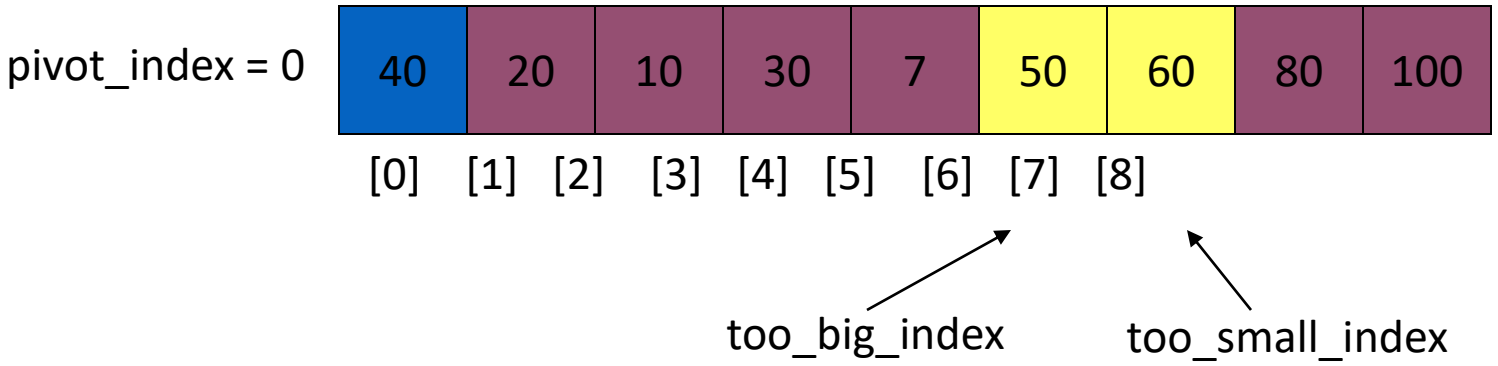
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



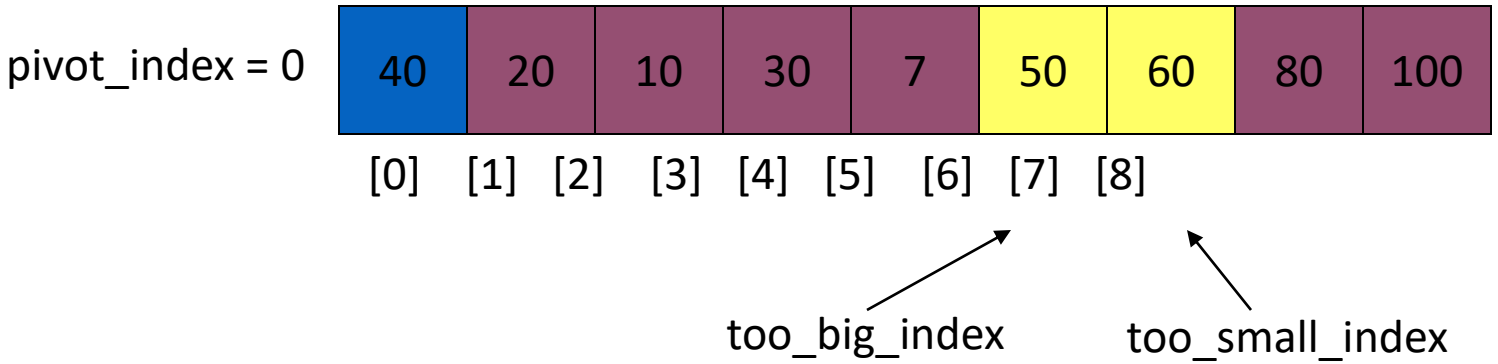
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



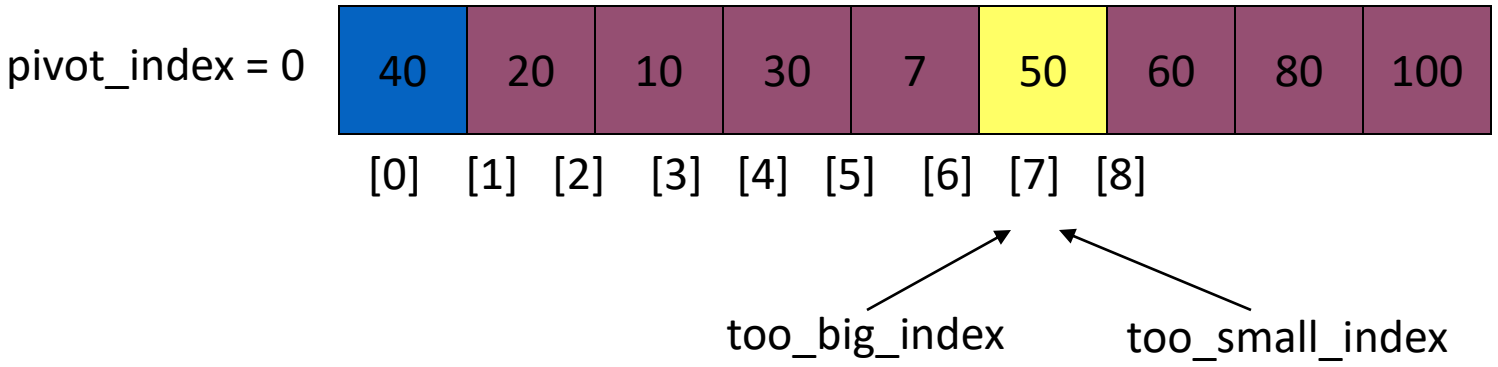
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



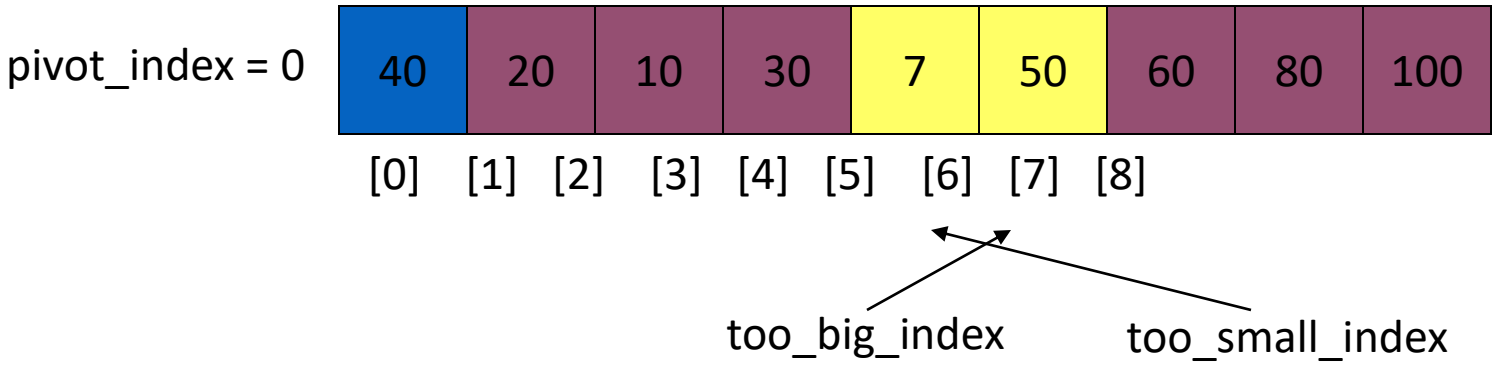
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



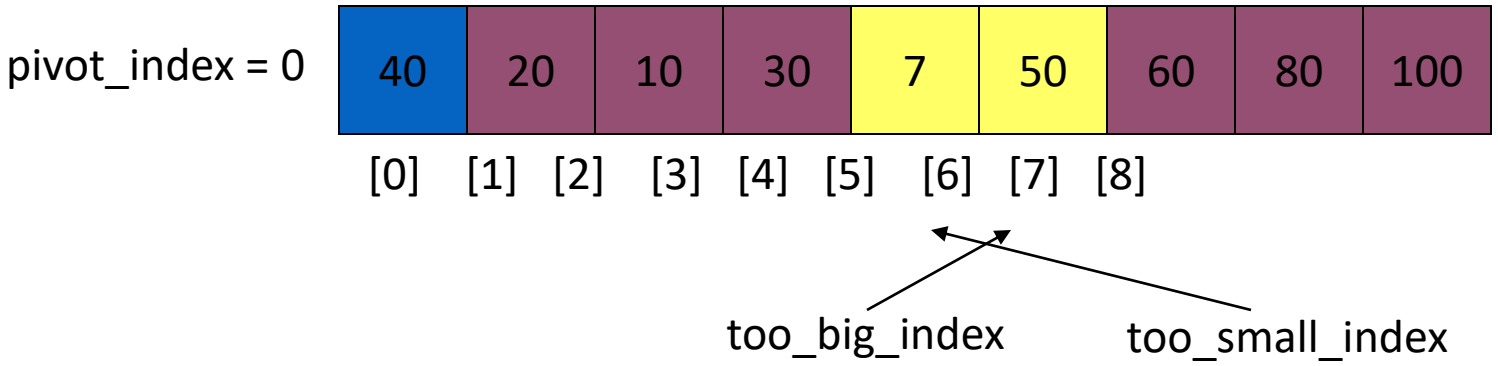
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



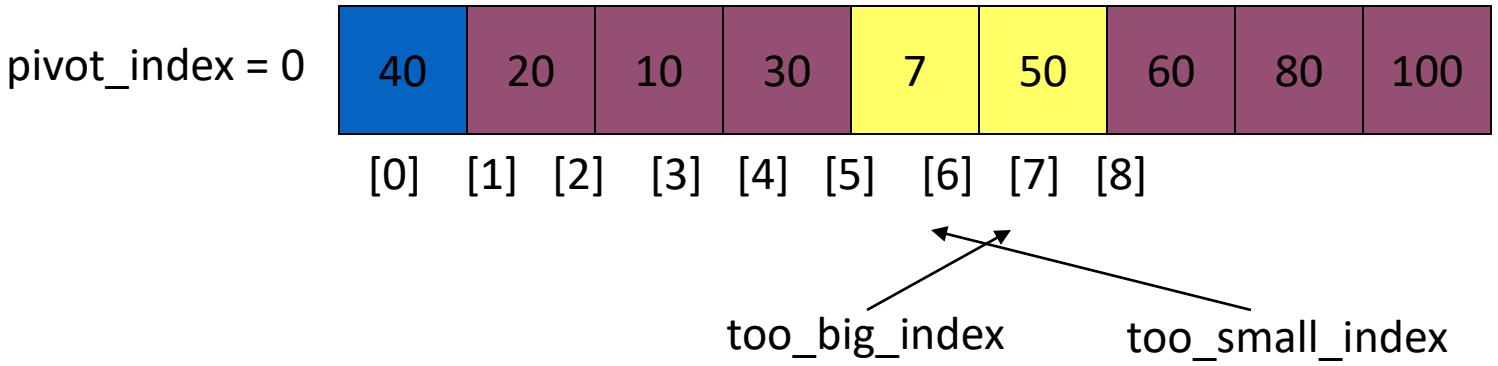
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



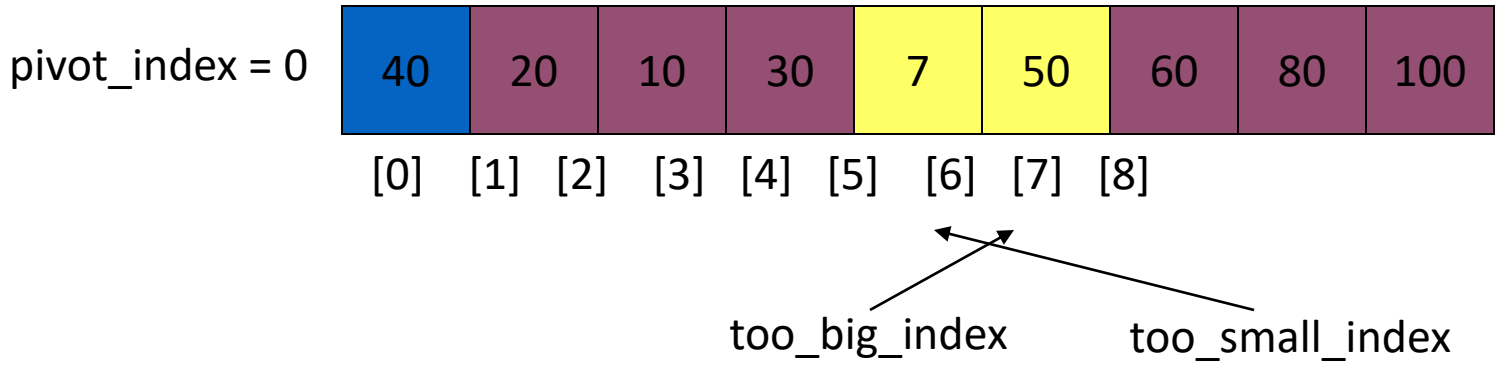
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



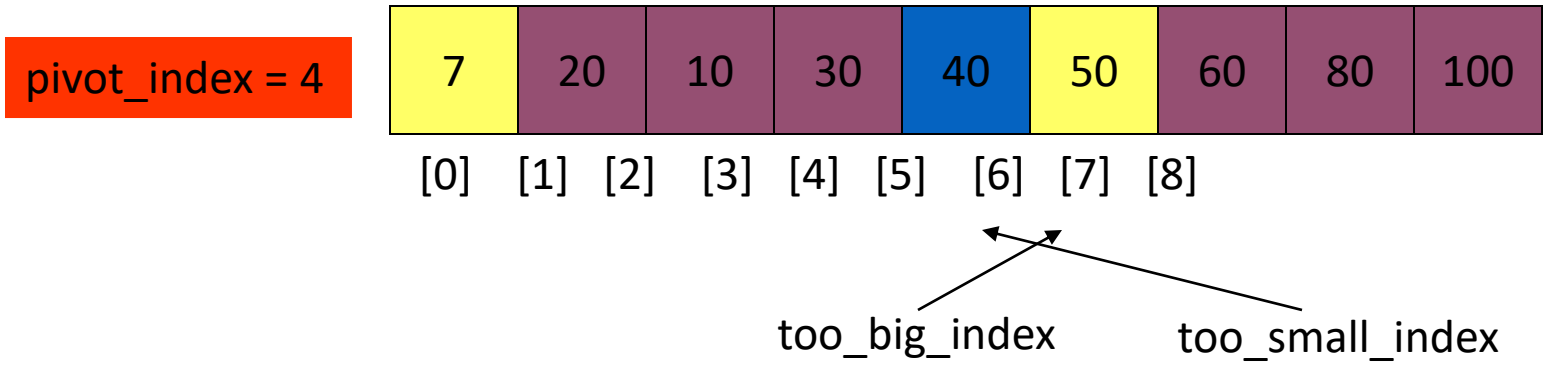
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



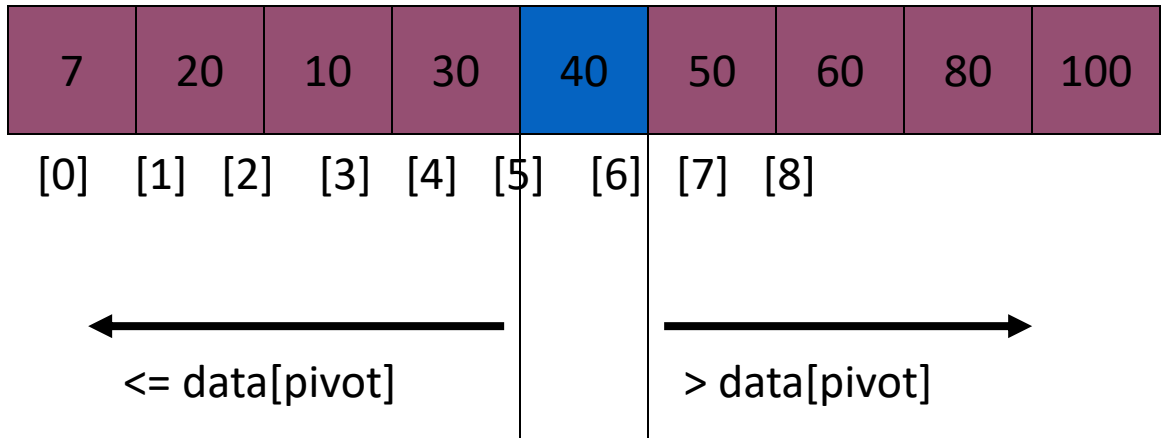
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



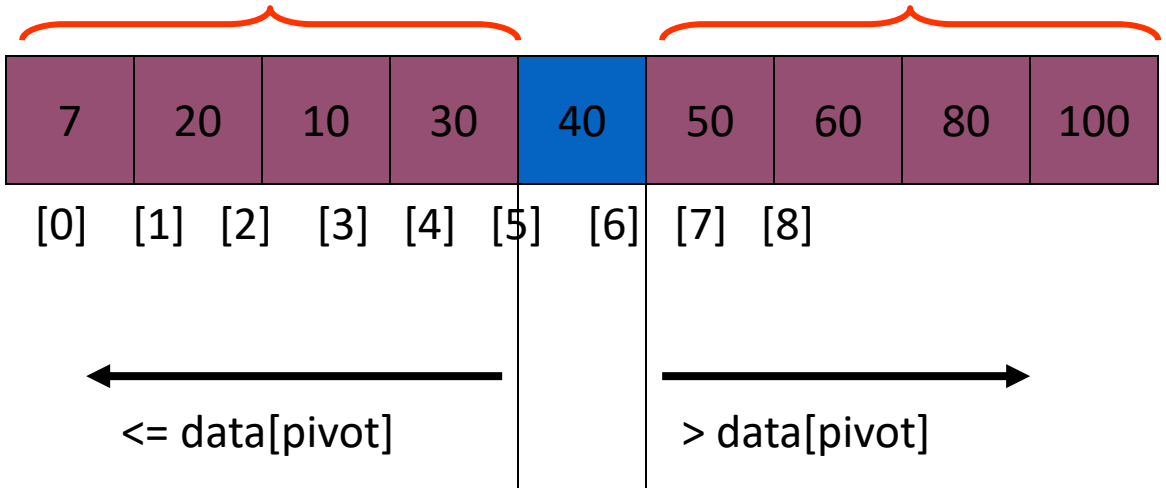
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



Partition Result



Recursion: Quicksort Sub-arrays



QUICKSORT

Alg.: QUICKSORT(*A*, p, r)

Initially: p=1, r=n

if p < r

then *q* ← PARTITION(*A*, p, r)

QUICKSORT (*A*, p, *q*)

QUICKSORT (*A*, *q*+1, r)

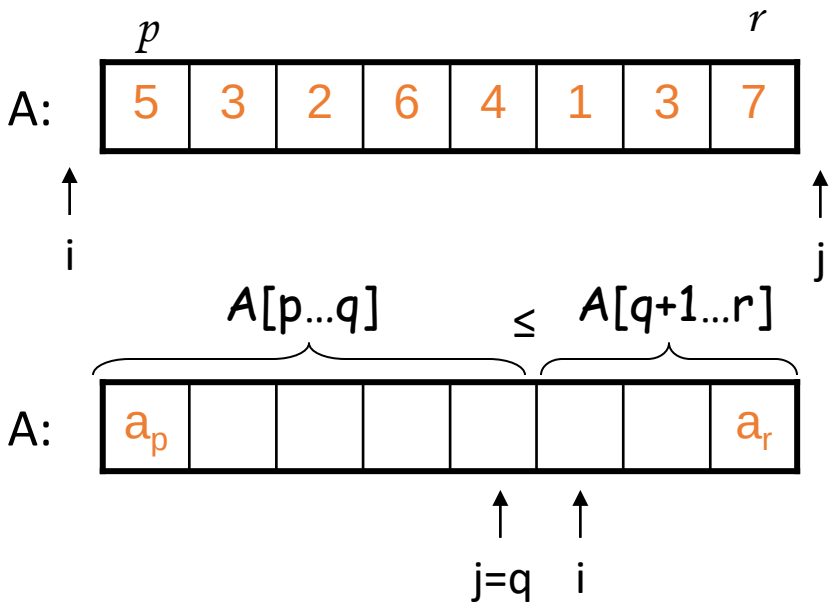
Recurrence: T(n) = T(q) + T(n – q) + f(n)

(*f*(*n*) depends on PARTITION())

Partitioning the Array

Alg. PARTITION (A, p, r)

1. $x \leftarrow A[p]$
2. $i \leftarrow p - 1$
3. $j \leftarrow r + 1$
4. **while** TRUE
5. **do repeat** $j \leftarrow j - 1$
6. **until** $A[j] \leq x$
7. **do repeat** $i \leftarrow i + 1$
8. **until** $A[i] \geq x$
9. **if** $i < j$
10. **then** exchange $A[i] \leftrightarrow A[j]$
11. **else return** j



Each element is visited once!

Running time: $\Theta(n)$
$n = r - p + 1$

* Can use some exchange in red marked

Recurrence

Alg.: QUICKSORT(*A*, p, r)

Initially: p=1, r=n

if p < r

then *q* ← PARTITION(*A*, p, r)

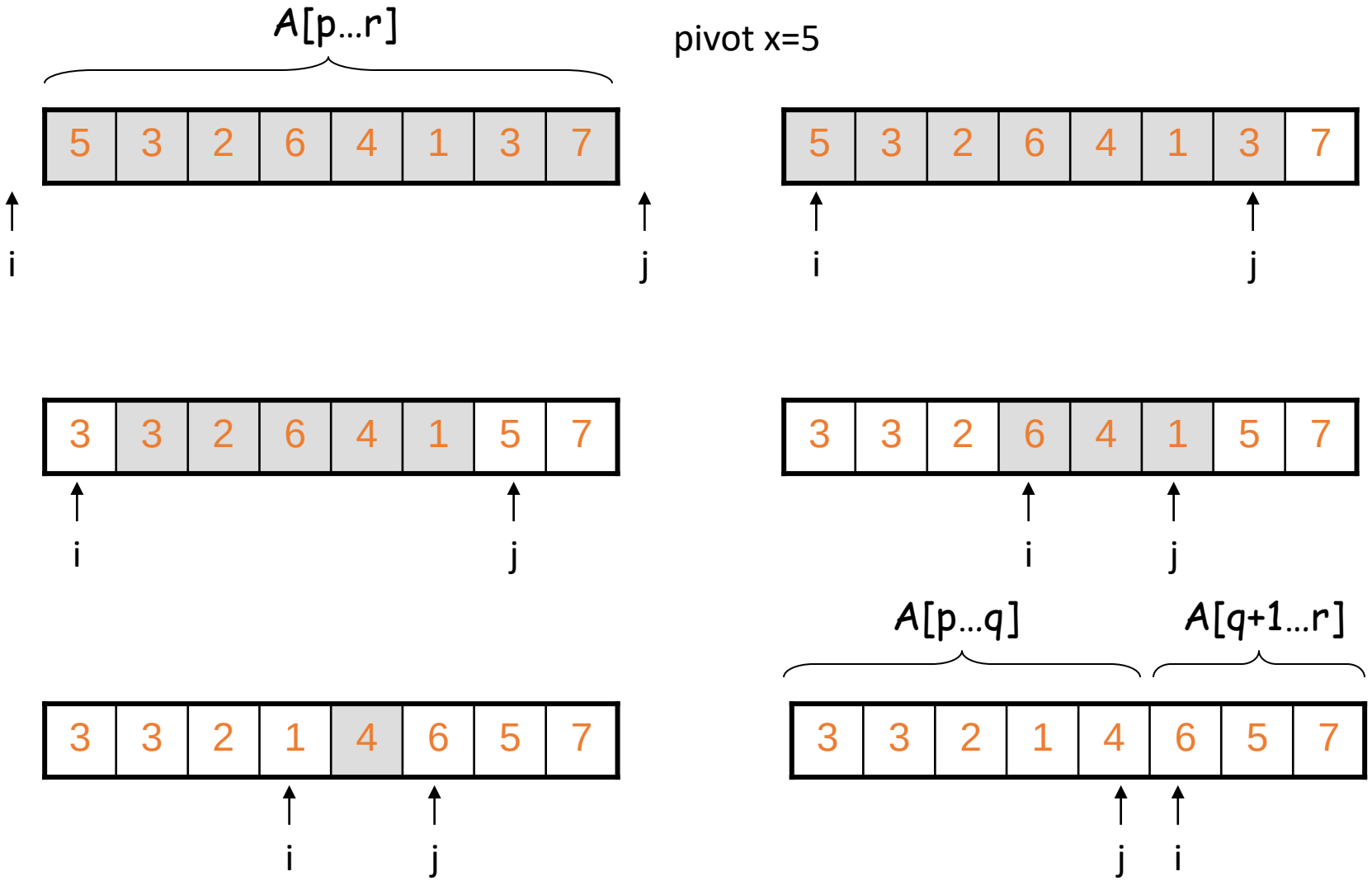
QUICKSORT (*A*, p, *q*-1)

* Also q-1

QUICKSORT (*A*, *q*+1, r)

Recurrence: $T(n) = T(q) + T(n - q) + n$

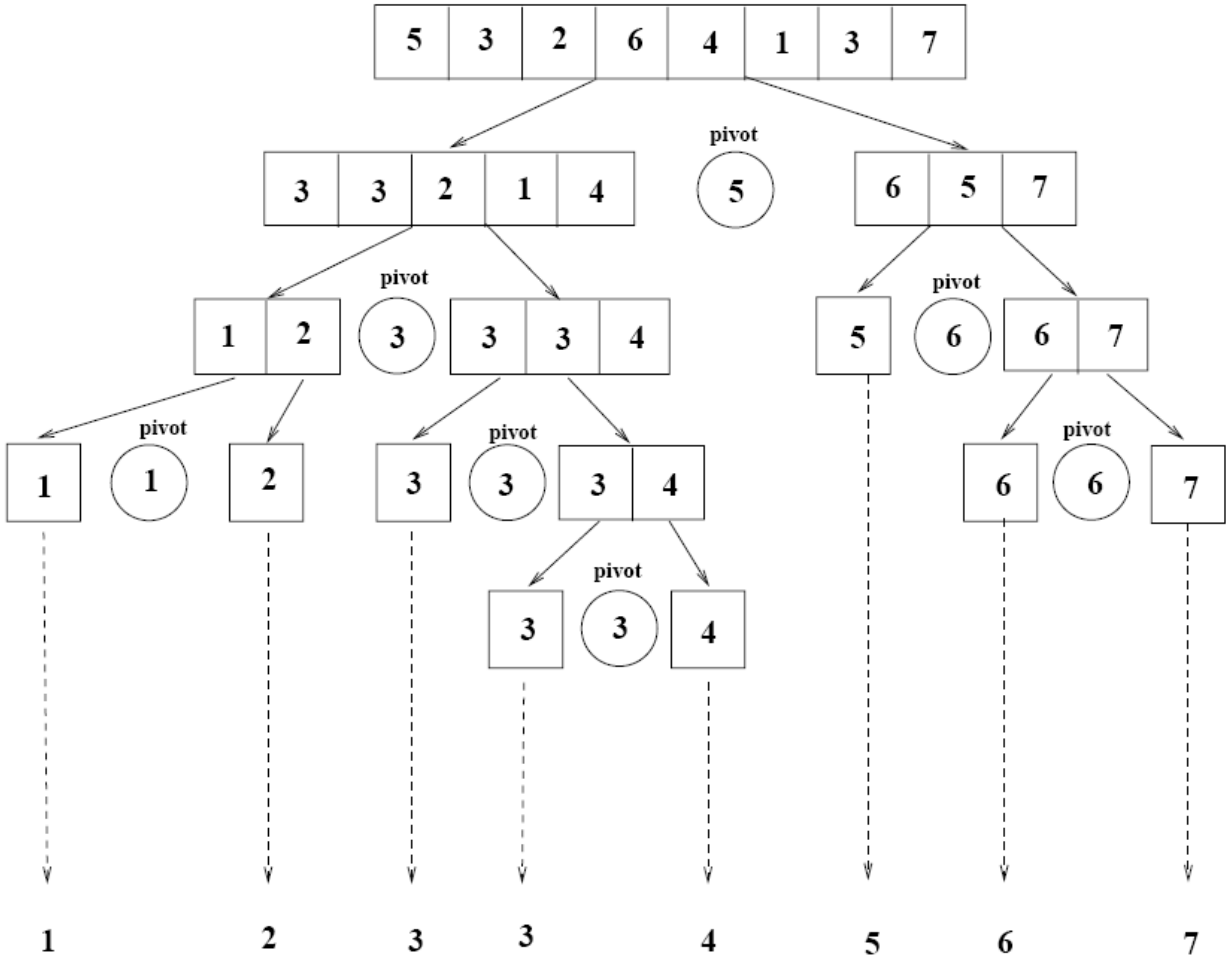
Example


A[p…r]
pivot x=5
5 3 2 6 4 1 3 7
i
j
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 6 4 1 5 7
i
j
A[p…q]
A[q+1…r]
3 3 2 1 4 6 5 7
i
j
3 3 2 1 4 6 5 7
j
i

Another Example of partitioning

- choose pivot: **4 3 6 9 2 4 3 1 2 1 8 9 3 5 6**
- search: **4 3 6 9 2 4 3 1 2 1 8 9 3 5 6**
- swap: **4 3 3 9 2 4 3 1 2 1 8 9 6 5 6**
- search: **4 3 3 9 2 4 3 1 2 1 8 9 6 5 6**
- swap: **4 3 3 1 2 4 3 1 2 9 8 9 6 5 6**
- search: **4 3 3 1 2 4 3 1 2 9 8 9 6 5 6**
- swap: **4 3 3 1 2 2 3 1 4 9 8 9 6 5 6**
- search: **4 3 3 1 2 2 3 1 4 9 8 9 6 5 6** (left > right)
- swap with pivot: **1 3 3 1 2 2 3 4 4 9 8 9 6 5 6**

Example


5 3 2 6 4 1 3 7
pivot
5
3 3 2 1 4
6 5 7
pivot
3
1 2
3 3 4
pivot
6
5
6 7
pivot
1
1
2
3
3
3 4
6
6
7
pivot
3
3
4
1 2 3 3 4 5 6 7

Quick Sort: Complexity analysis

Memory requirement:

Size of the stack is:

$$S(n) = \lceil log_2 n \rceil + 1$$

Number of comparisons:

- Let, $T(n)$ represents total time to sort n elements and $P(n)$ represents the time for perform a partition of a list of n elements.

$$T(n) = P(n) + T(n_l) + T(n_r), \quad \text{with } T(1) = T(0) = 0$$

where, n_l = number of elements in the left sub list

n_r = number of elements in the right sub list and $0 \leq n_l, n_r < n$

Worst Case Partitioning

- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

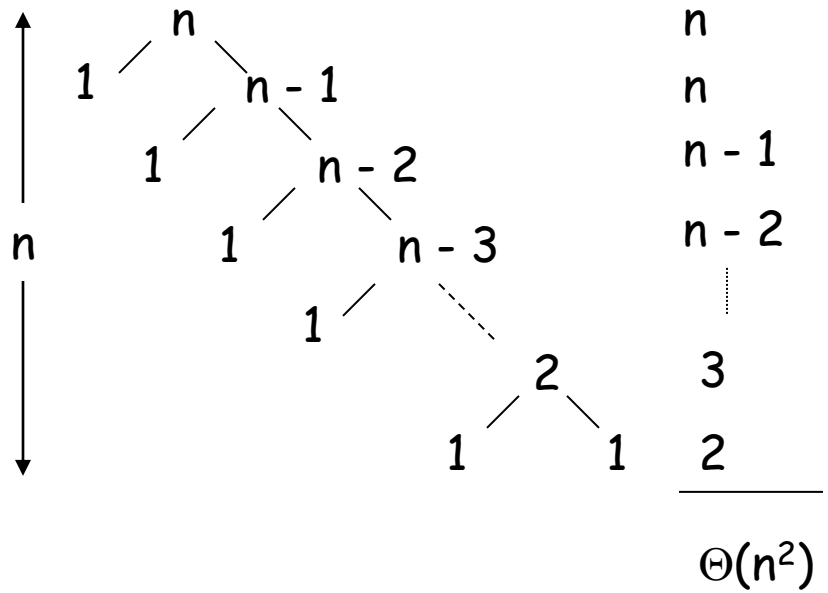
  T(n) = T(1) + T(n – 1) + n,

  T(1) = Θ(1)

  T(n) = T(n – 1) + n

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

  $=\frac{n(n-1)}{2}$



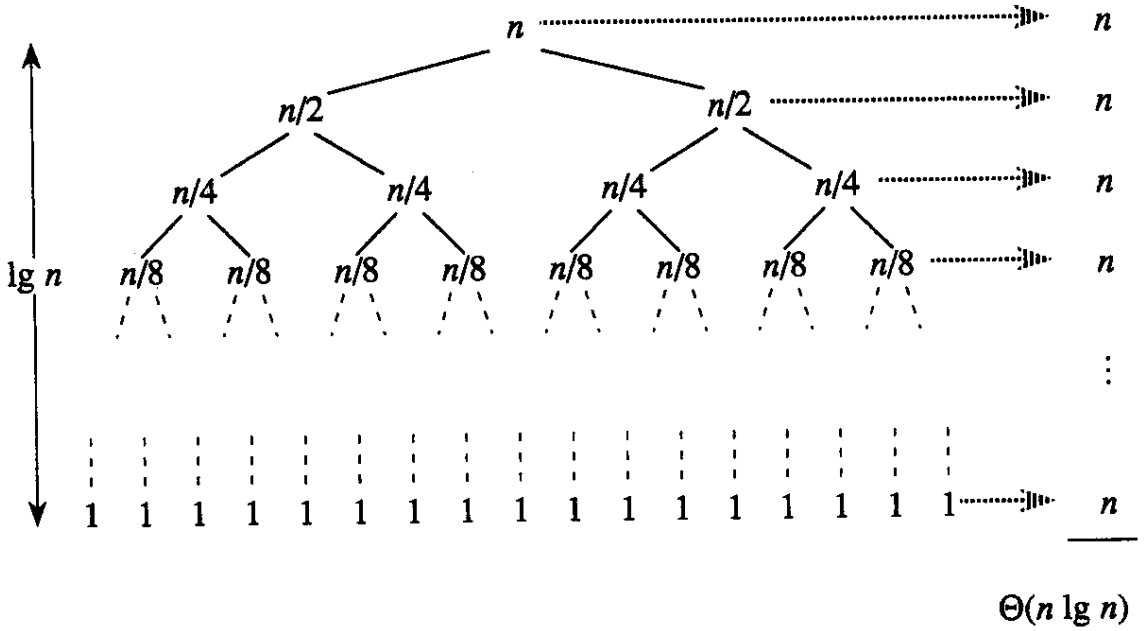
When does the worst case happen?

Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size n/2
- Recurrence: q=n/2

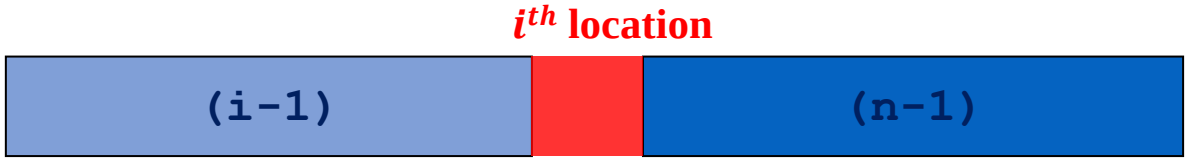
  $T(n) = 2T(n/2) + \Theta(n)$ $2^k = n$ Take both side log

  $T(n) = \Theta(n \lg n)$ (Master theorem)



Quick Sort: Complexity analysis

Case 3: Elements in the list are in random order



Number of comparisons:

$$C(n) = (n-1) + \sum_{i=1}^{n-1} \frac{1}{n}[C(i-1) + C(n-i-1)] \quad with\ C(1) = C(0) = 0$$

For simplicity ……

Analysing Quicksort: The Average Case

For any pivot position i; $i \in \{0, \ldots, n-1\}$:

- Time for partitioning an array : cn
- The head and tail subarrays contain i and $n-1-i$ items, respectively: $T(n) = cn + T(i) + T(n-1-i)$

Average running time for sorting (a more complex recurrence):

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-1-i) + cn)$$

$$= \frac{2}{n} (T(0) + T(1) + \ldots + T(n-2) + T(n-1)) + cn, \text{ or}$$

$$nT(n) = 2 (T(0) + T(1) + \ldots + T(n-2) + T(n-1)) + cn^2$$

$$(n-1)T(n-1) = 2 (T(0) + T(1) + \ldots + T(n-2)) + c(n-1)^2$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c \approx 2T(n-1) + 2cn$$

Thus, $nT(n) \approx (n+1)T(n-1) + 2cn$, or $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$

Analysing Quicksort: The Average Case (continued)

"Telescoping" $\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2c}{n+1}$ to get the explicit form:

$$\begin{aligned}
&\frac{T(n)}{n+1} + \frac{T(n-1)}{n} + \frac{T(n-2)}{n-1} + \ldots + \frac{T(2)}{3} + \frac{T(1)}{2} \\
&\quad - \frac{T(n-1)}{n} - \frac{T(n-2)}{n-1} - \ldots - \frac{T(2)}{3} - \frac{T(1)}{2} - \frac{T(0)}{1} \\
&\quad = \frac{2c}{n+1} + \frac{2c}{n} + \ldots + \frac{2c}{3} + \frac{2c}{2}, \text{ or}
\end{aligned}$$

$$\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2c\left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} + \frac{1}{n+1}\right) \approx 2c(H_{n+1} - 1) \approx c' \log n$$

($H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n + 0.577$ is the n^{th} harmonic number).

Therefore, $T(n) \approx c'(n+1)\log n \in O(n \log n)$.

Quicksort is our first example of dramatically different worst-case and average-case performances!

Quick Sort: Summary of Complexity analysis

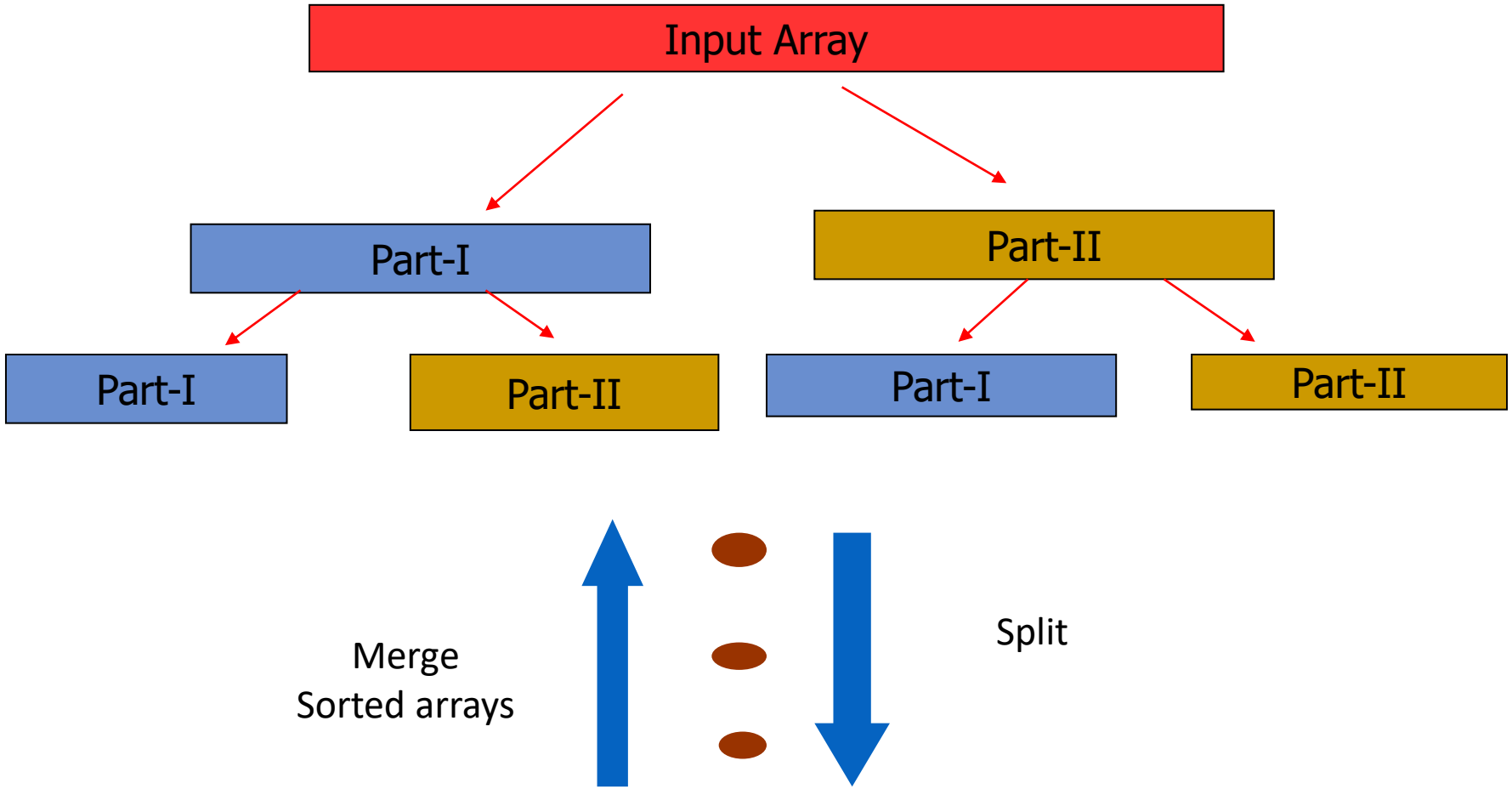
Case	Run time, $T(n)$	Complexity	Remarks
Case 1	$T(n) = c\frac{n(n-1)}{2}$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) = 4c(n+1)(log_e n + 0.577) - 8cn$ $T(n) = 2c[nlog_2 n - n + 1]$	$T(n) = O(nlog_2 n)$	Best / Average case

Is this sort Stable ?

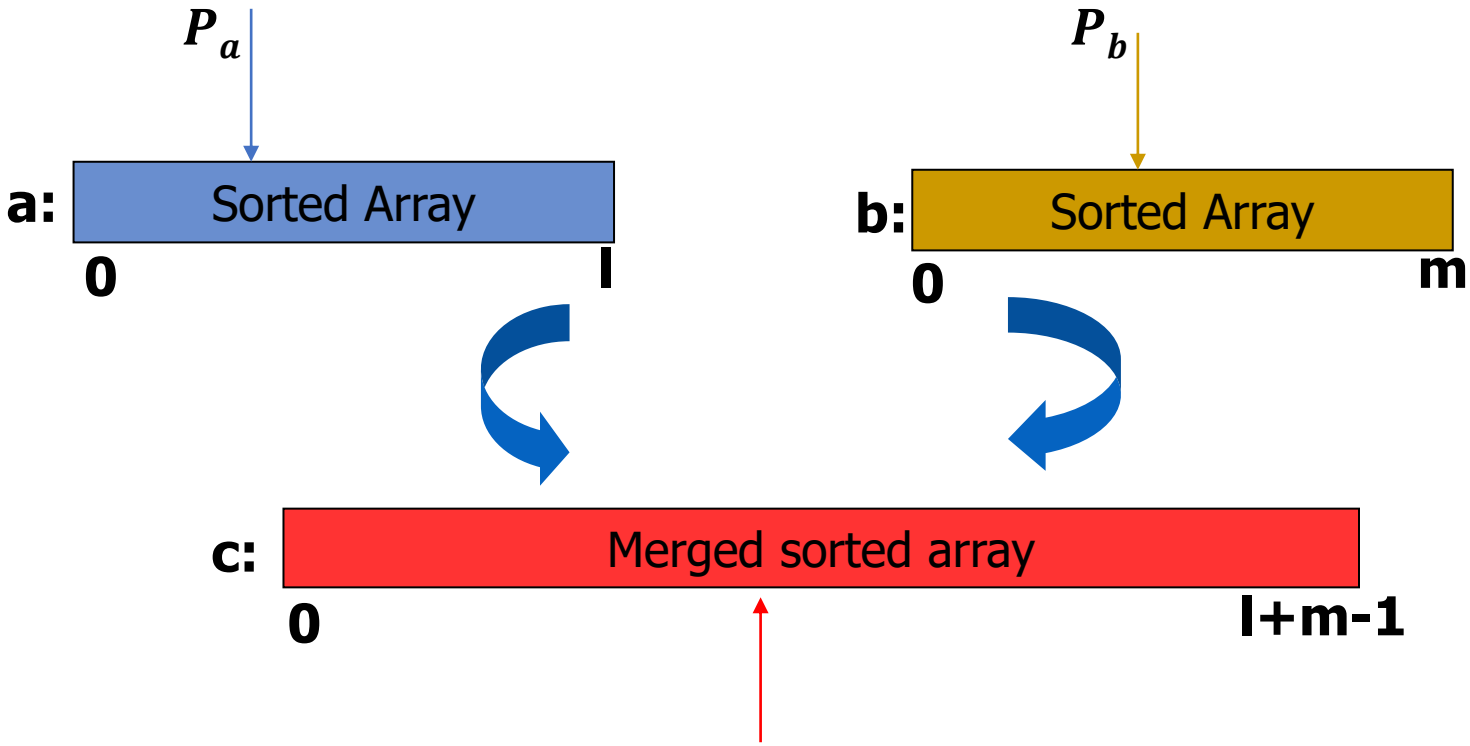
NO

Merge Sort

Merge Sort – How it Works?

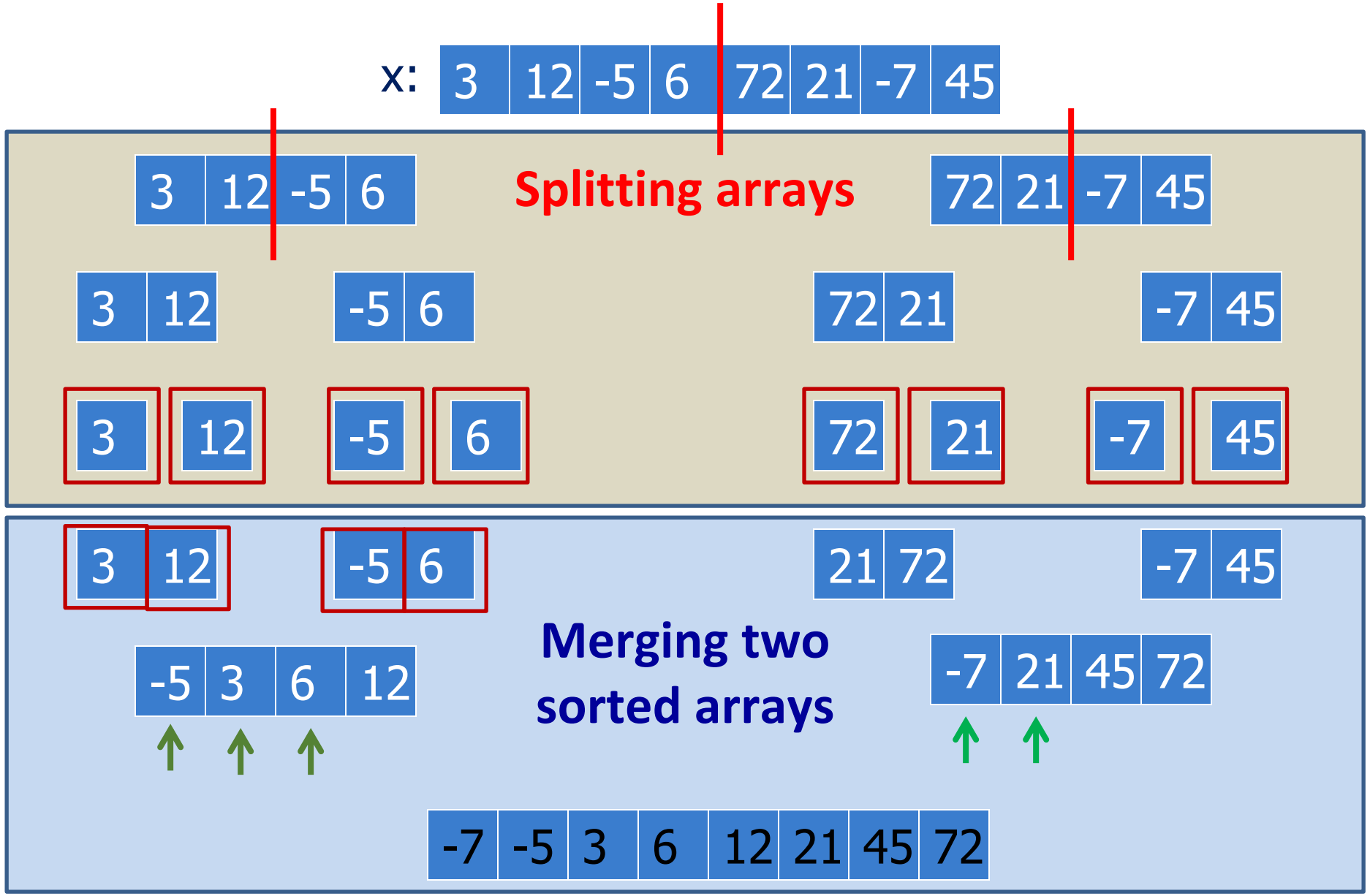


Merging two Sorted arrays



Move and copy elements pointed by P_a if its value is smaller than the element pointed by P_b in $(l + m - 1)$ operations and otherwise.

Merge Sort – Example



Merge Sort Program

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

int main()
{
    int a[30],n,i;
    printf("Enter no of elements:");
    scanf("%d",&n);
    printf("Enter array elements:");

    for(i=0;i<n;i++)
      scanf("%d",&a[i]);

    mergesort(a,0,n-1);

    printf("\nSorted array is :");
    for(i=0;i<n;i++)
      printf("%d ",a[i]);

    return 0;
}
```

Merge Sort Program

```
void mergesort(int a[],int i,int j)
{
     int mid;

     if(i<j) {
     mid=(i+j)/2;
    /* left recursion */
     mergesort(a,i,mid);
    /* right recursion */
     mergesort(a,mid+1,j);
  /* merging of two sorted sub-arrays */
     merge(a,i,mid,mid+1,j);
     }
}
```

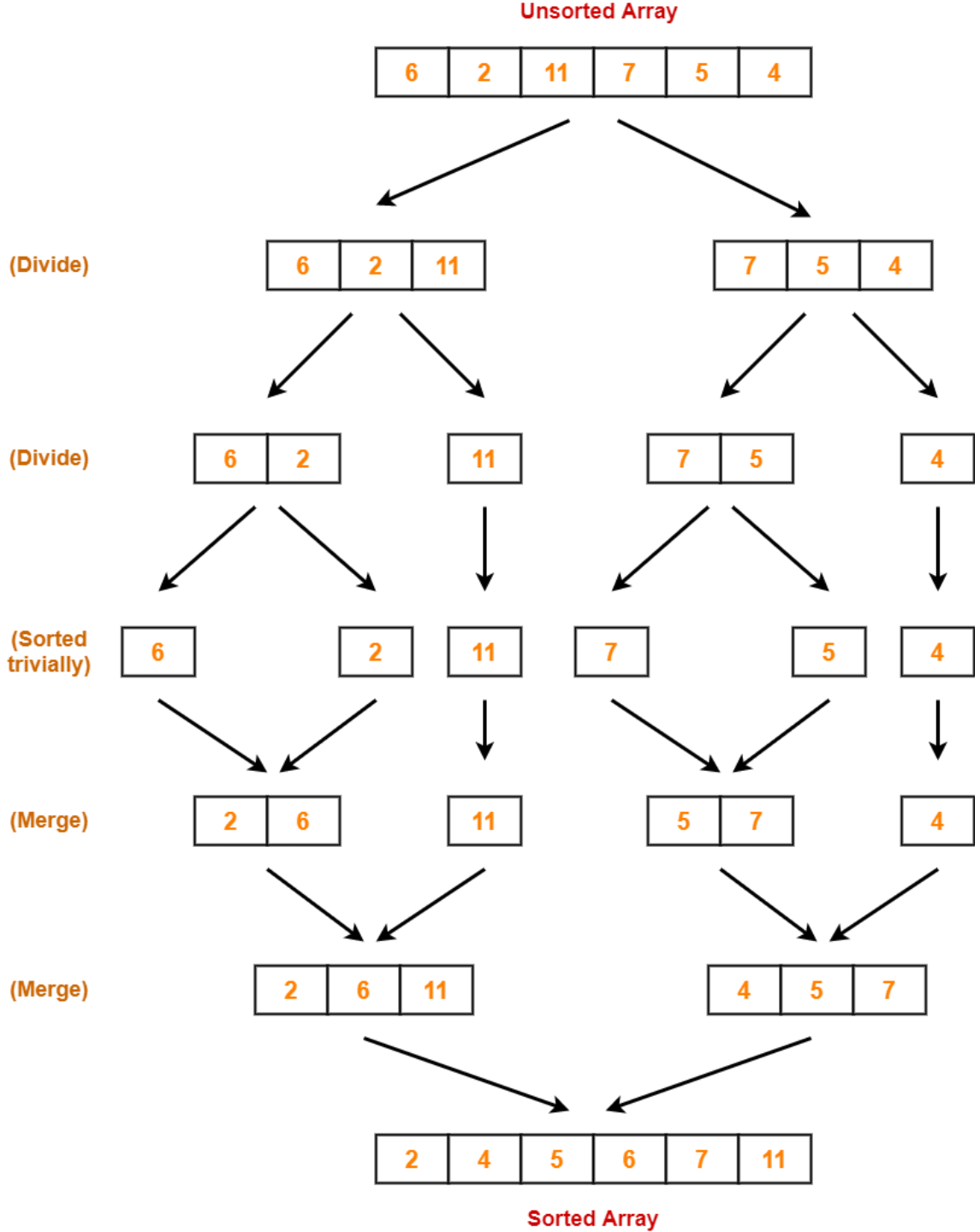
Merge Sort Program

```
void merge(int a[],int i1,int i2,int j1,int j2)
{
    int temp[50]; //array used for merging
    int i=i1,j=j1,k=0;

    while(i<=i2 && j<=j2) //while elements in both lists
    {
      if(a[i]<a[j])
         temp[k++]=a[i++];
      else
         temp[k++]=a[j++];
     }

    while(i<=i2) //copy remaining elements of the first list
       temp[k++]=a[i++];

    while(j<=j2) //copy remaining elements of the second list
       temp[k++]=a[j++];

    for(i=i1,j=0;i<=j2;i++,j++)
    a[i]=temp[j]; //Transfer elements from temp[] back to a[]
}
```

Sample code

```
def mergeSort(arr, l, r):
    if l < r:


        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)//2


        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)



# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print("Given array is")
for i in range(n):
    print("%d" % arr[i],end=" ")


mergeSort(arr, 0, n-1)
print("\n\nSorted array is")
for i in range(n):
    print("%d" % arr[i],end=" ")
```

```
# Python program for implementation of MergeSort # Merges two
subarrays of arr[]. # First subarray is arr[l..m] # Second subarray is
arr[m+1..r]
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

     # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

     # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]

     for j in range(0, n2):
        R[j] = arr[m + 1 + j]

     # Merge the temp arrays back into arr[l..r]
    i = 0  # Initial index of first subarray
    j = 0  # Initial index of second subarray
    k = l      # Initial index of merged subarray

     while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

     # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

 # l is for left index and r is right index of the # sub-array of arr
```


Unsorted Array
6 2 11 7 5 4
(Divide)
6 2 11
7 5 4
(Divide)
6 2
11
7 5
4
(Sorted trivially)
6
2
11
7
5
4
(Merge)
2 6
11
5 7
4
(Merge)
2 6 11
4 5 7
2 4 5 6 7 11
Sorted Array

Divide-and-Conquer

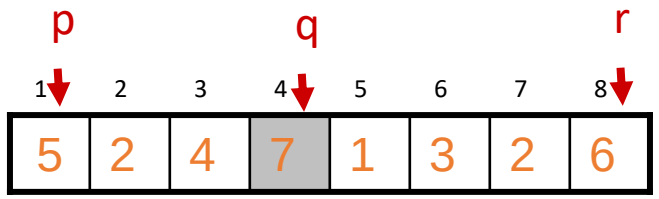
- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Conquer** the sub-problems
  - Solve the sub-problems recursively
  - Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem

Merge Sort Approach

- To sort an array $A[p \,..\, r]$:
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

Merge Sort

Alg.: MERGE-SORT(A, p, r)



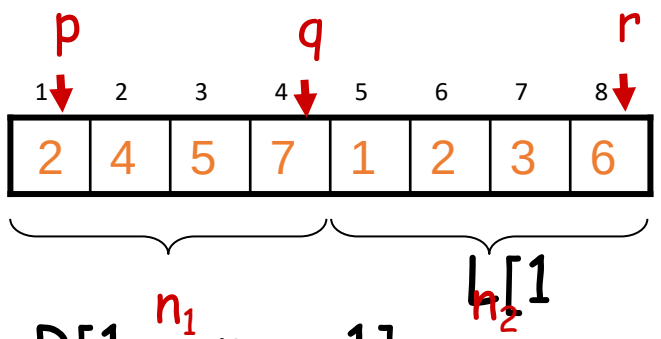

```
if p < r                                  ▷ Check for base case
  then q ← ⌊(p + r)/2⌋                    ▷ Divide
       MERGE-SORT(A, p, q)                ▷ Conquer
       MERGE-SORT(A, q + 1, r)            ▷ Conquer
       MERGE(A, p, q, r)                  ▷ Combine
```

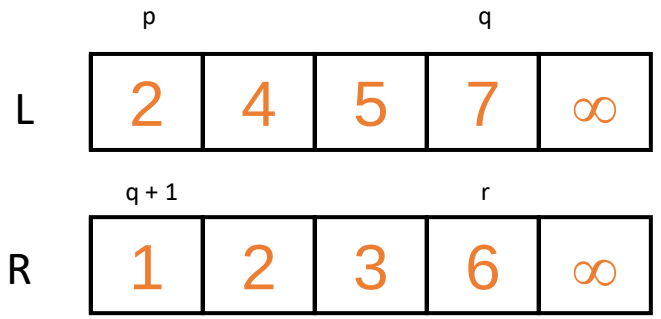
- Initial call: MERGE-SORT(A, 1, n)

Merge - Pseudocode

Alg.: MERGE(A, p, q, r)

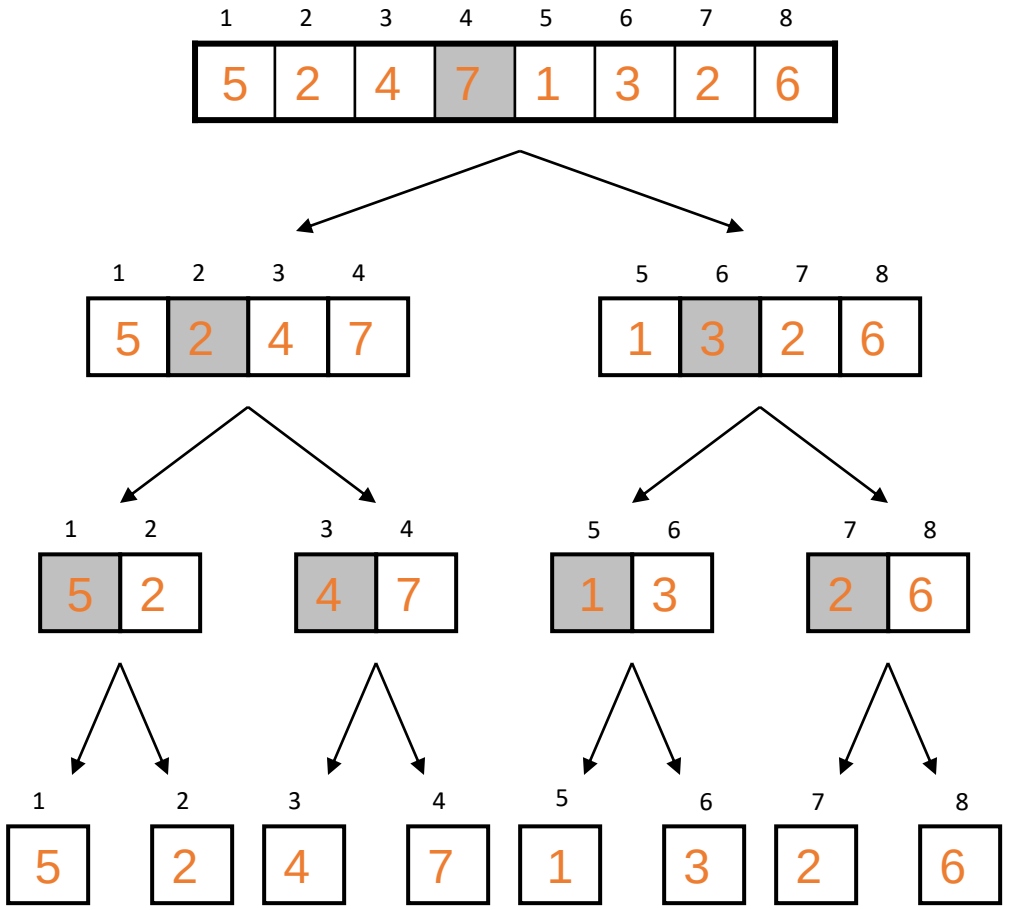
1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1 \ldots n_1 + 1]$ and the next n_2 elements into $R[1 \ldots n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\, i \,] \leq R[\, j \,]$
7. **then** $A[k] \leftarrow L[\, i \,]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\, j \,]$
10. $j \leftarrow j + 1$





Example – **n** Power of 2

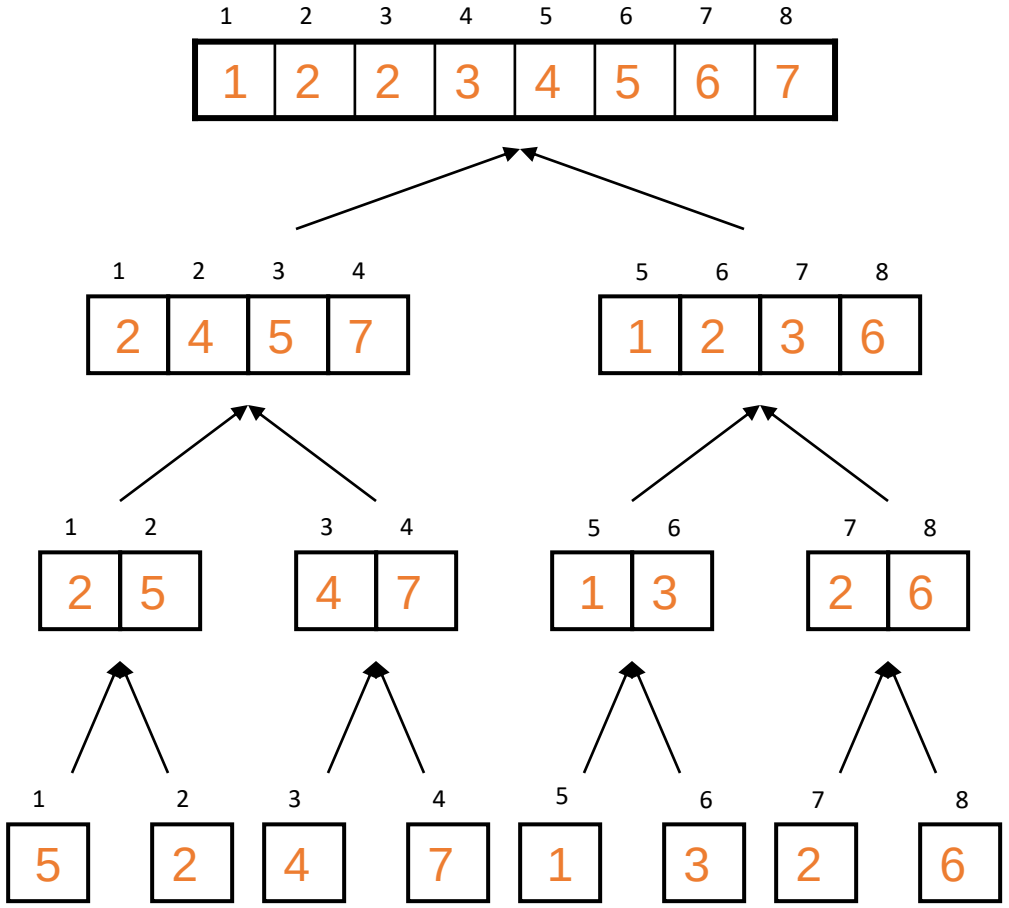
Divide



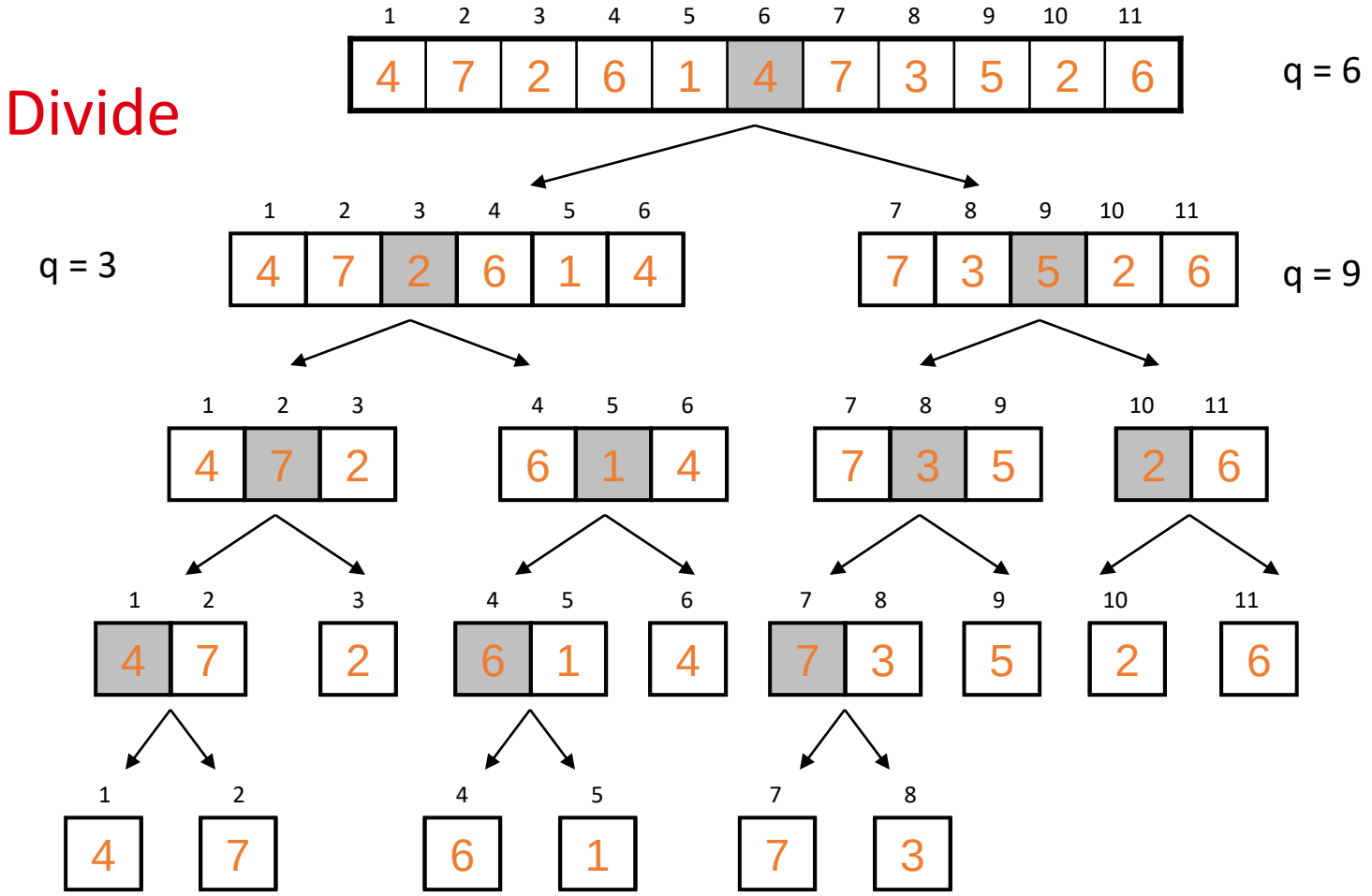
q = 4

Example – **n** Power of 2

Conquer and Merge

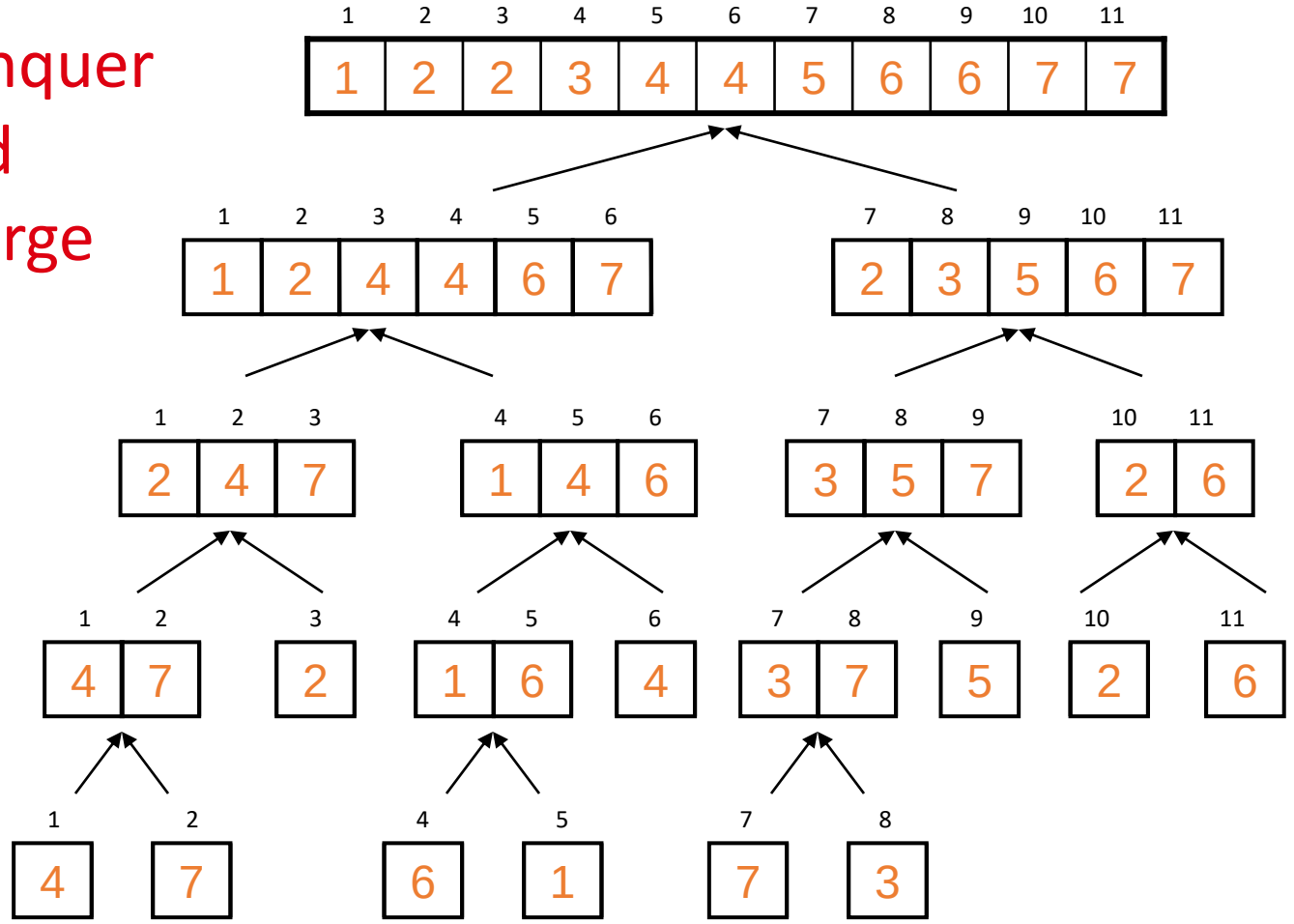


Example – **n** Not a Power of 2

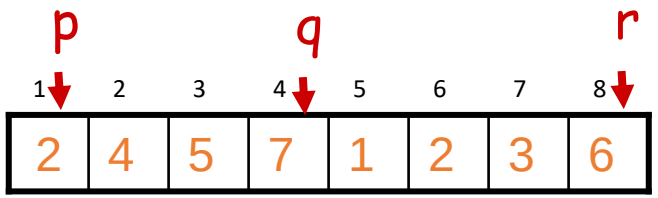


Example – **n** Not a Power of 2

Conquer and Merge



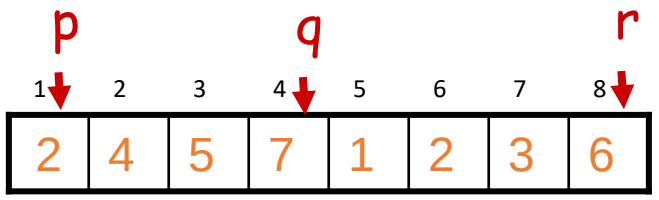
Merging

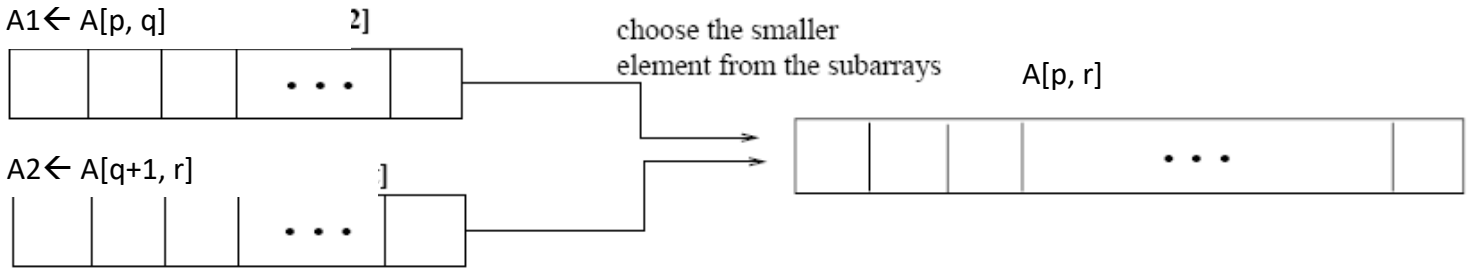


- **Input:** Array A and indices p, q, r such that $p \leq q < r$
  - Subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ are sorted
- **Output:** One single sorted subarray $A[p \,..\, r]$

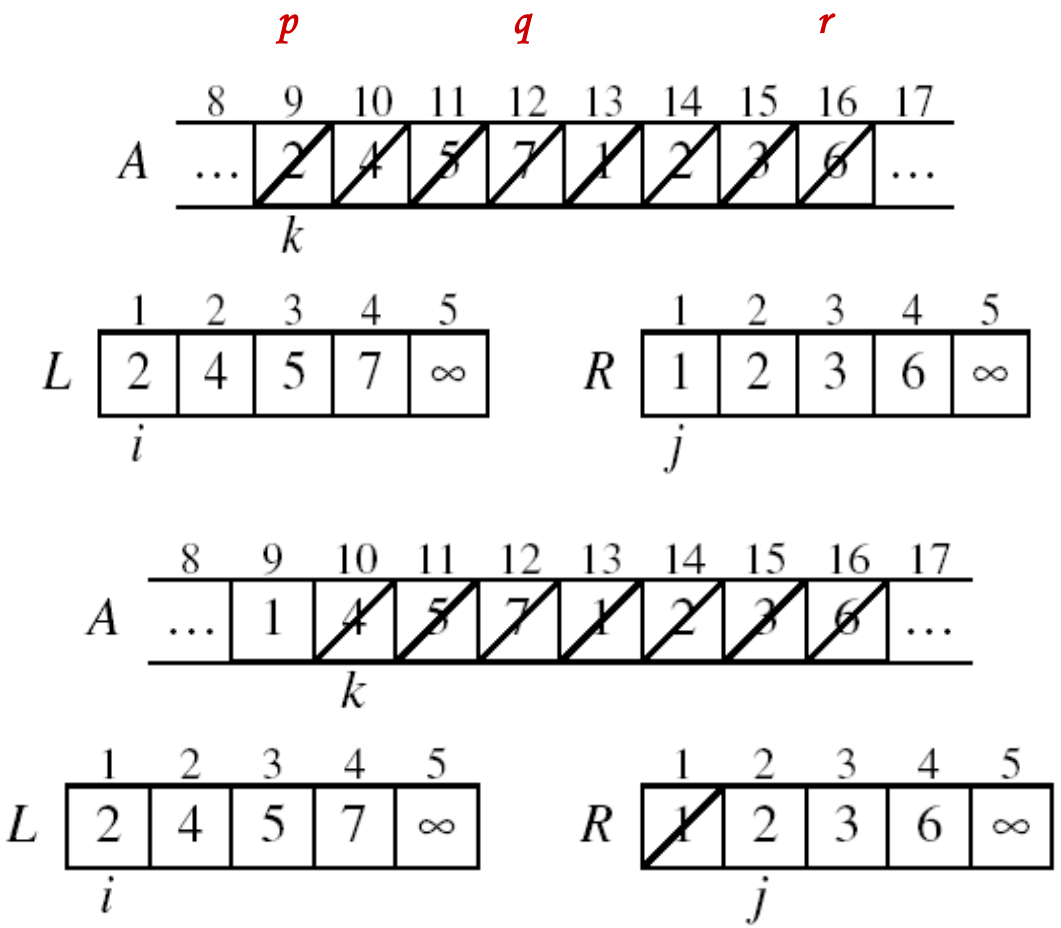
Merging

- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile

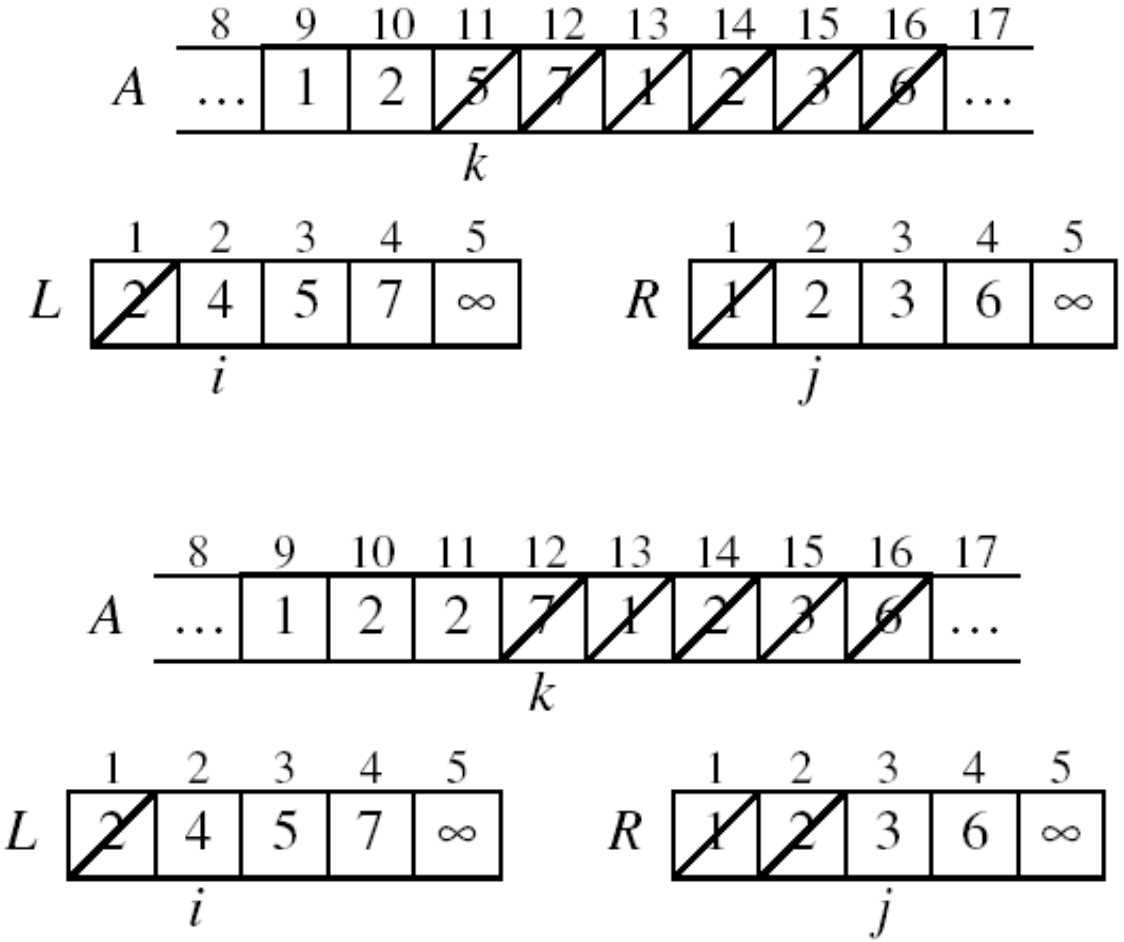




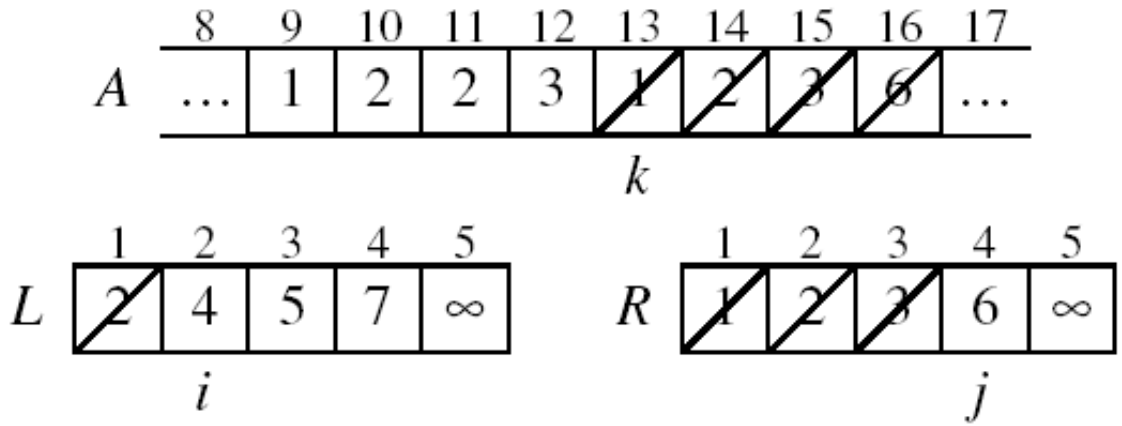
Example: MERGE(A, 9, 12, 16)

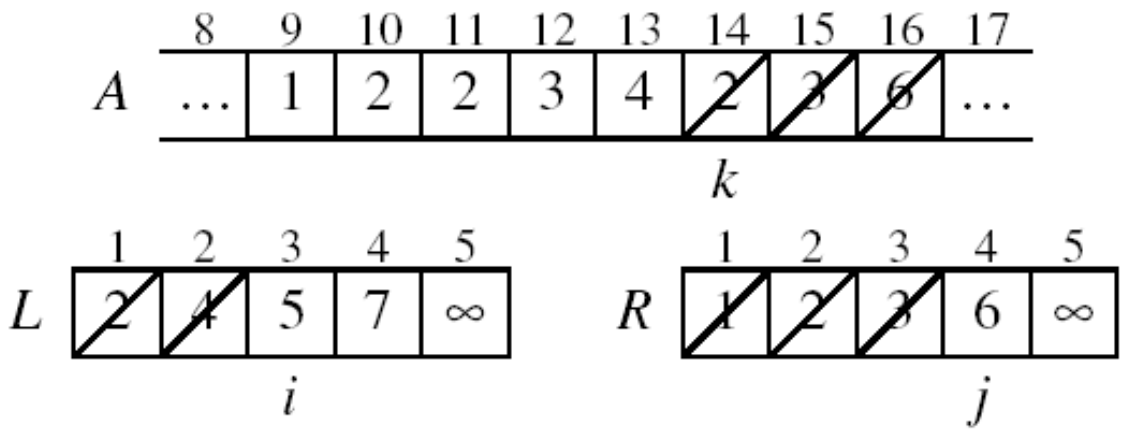


Example: MERGE(A, 9, 12, 16)

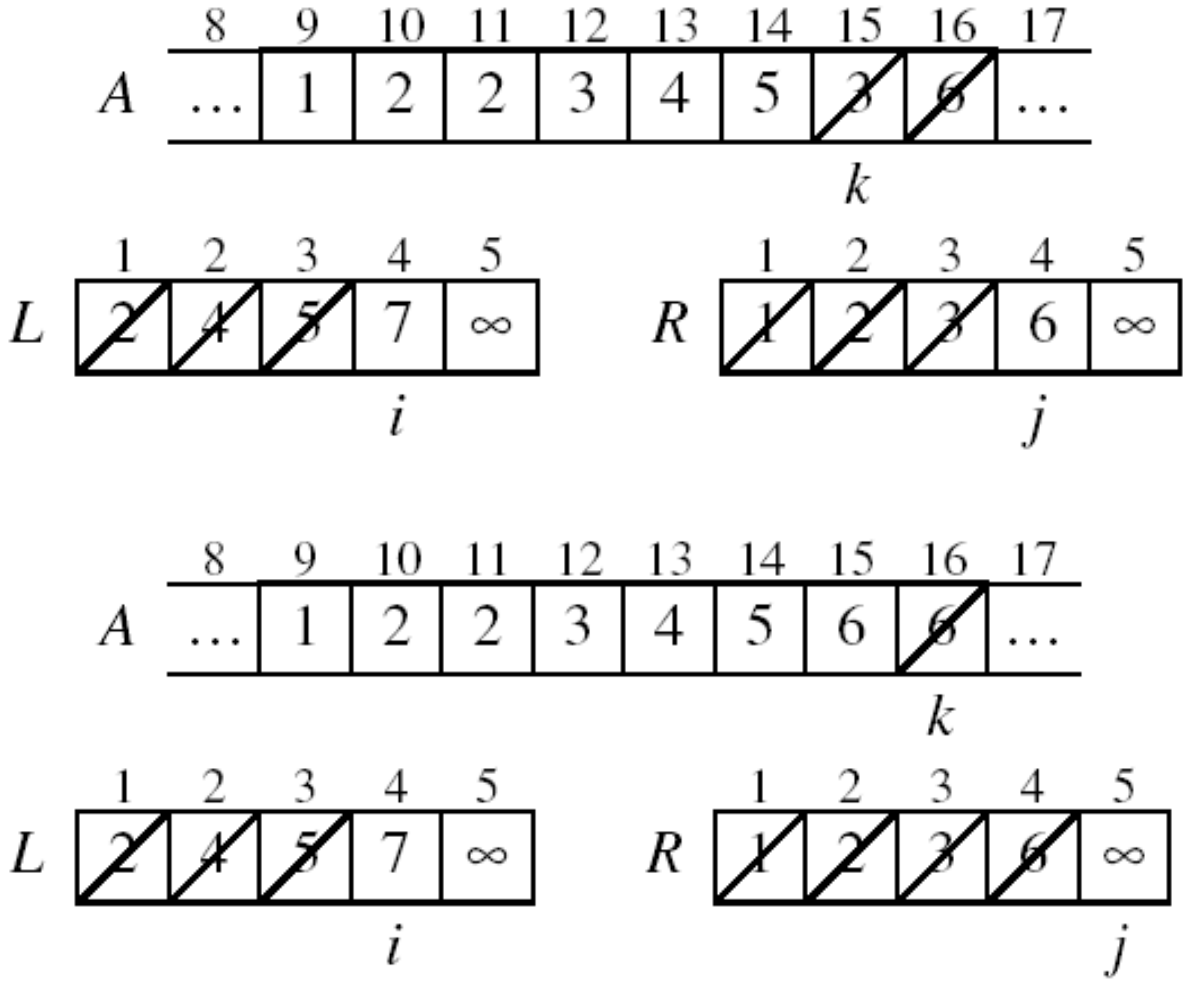


Example (cont.)

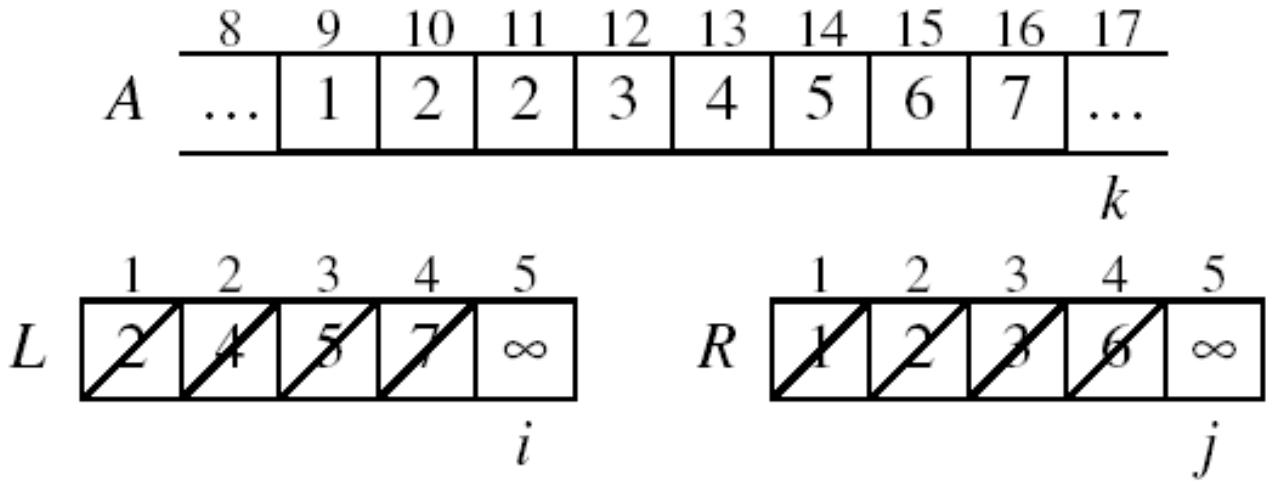




Example (cont.)



Example (cont.)

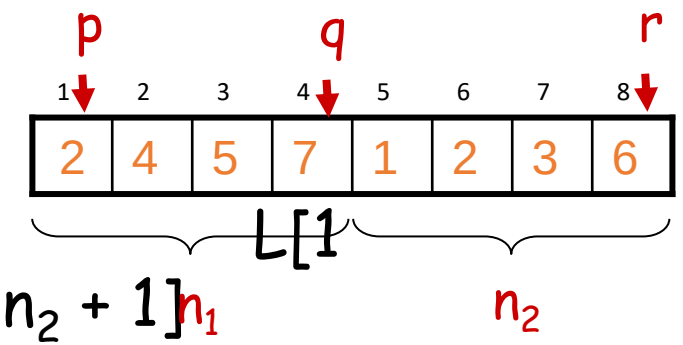


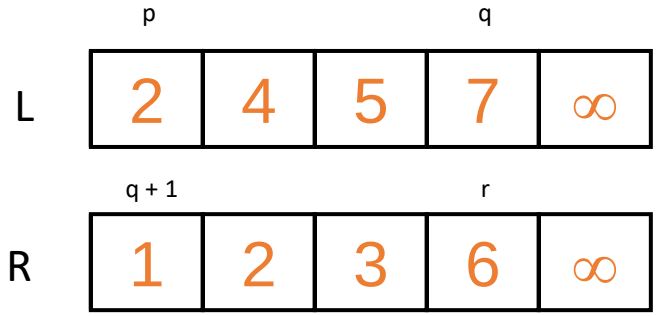
Done!

Merge - Pseudocode

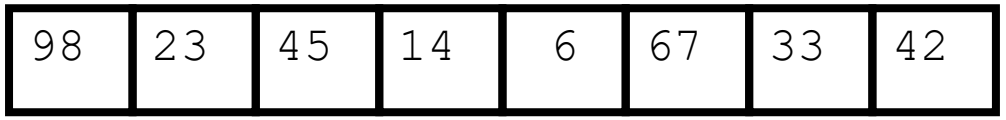
Alg.: MERGE(A, p, q, r)

1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1 \ldots n_1 + 1]$ and the next n_2 elements into $R[1 \ldots n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\, i \,] \leq R[\, j \,]$
7. **then** $A[k] \leftarrow L[\, i \,]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\, j \,]$
10. $j \leftarrow j + 1$

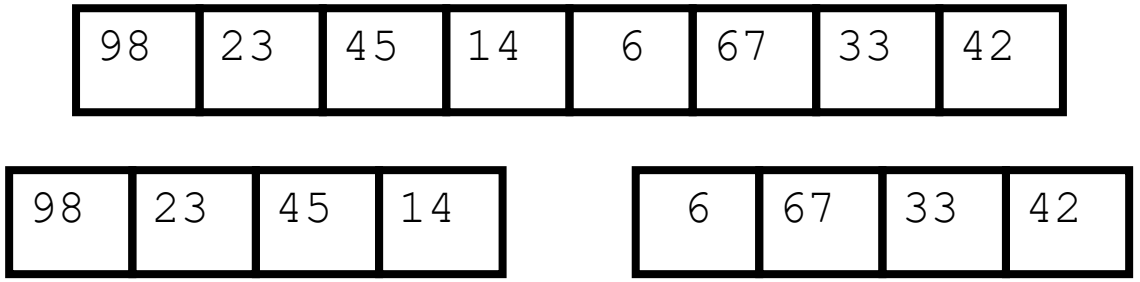




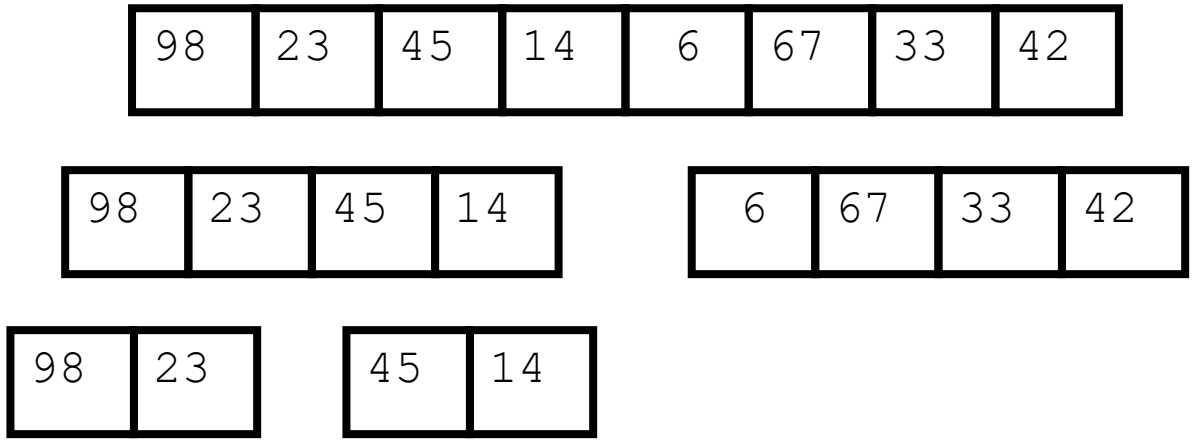
Another example



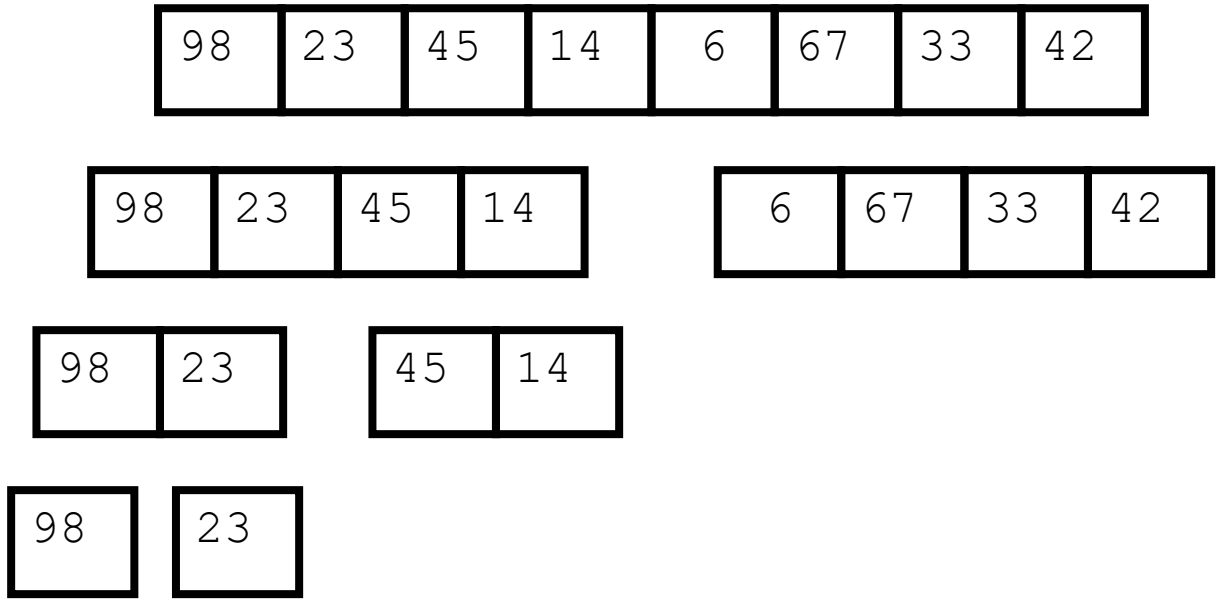
Another example



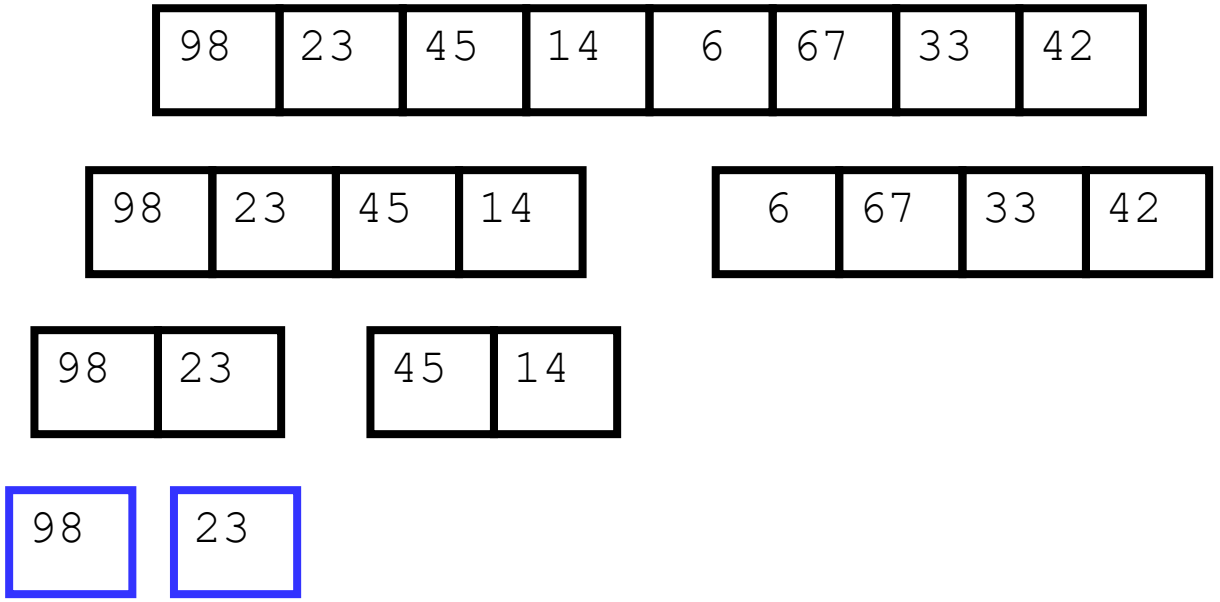
Another example



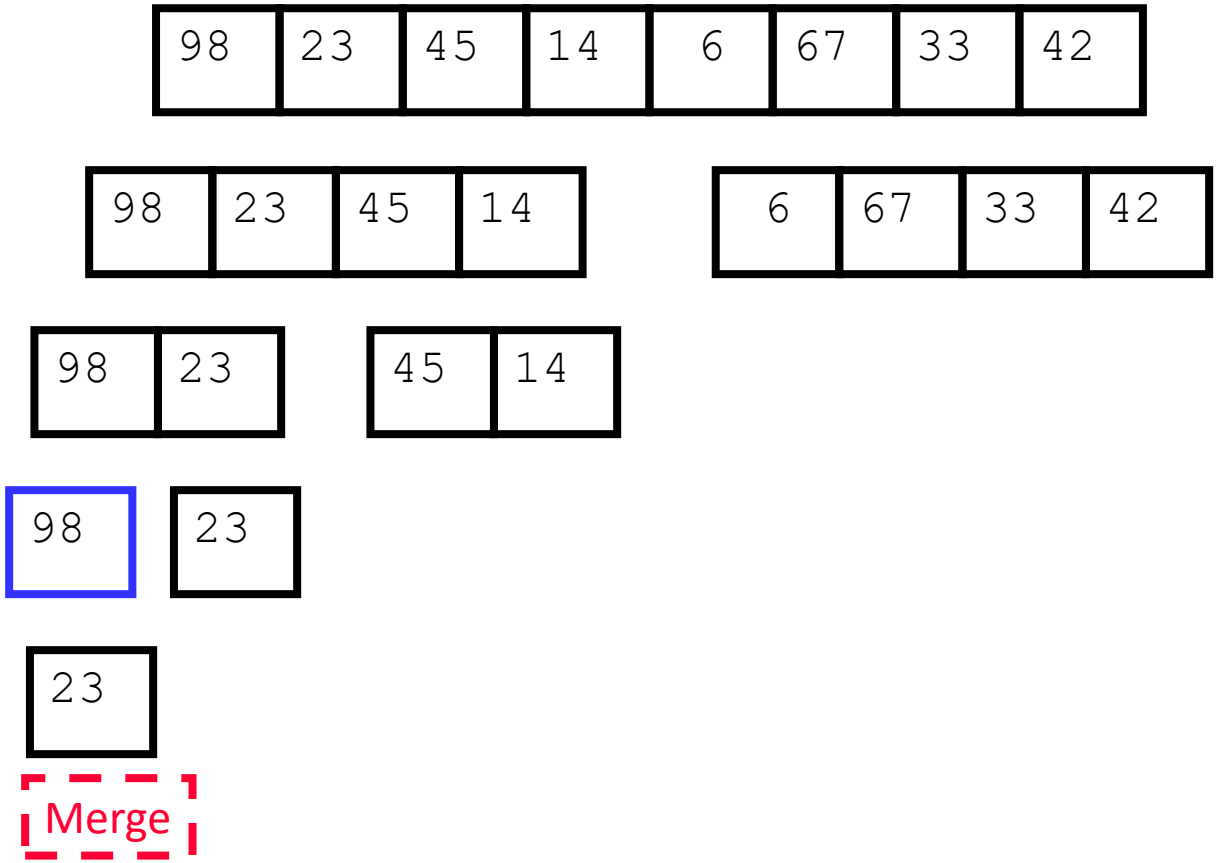
Another example



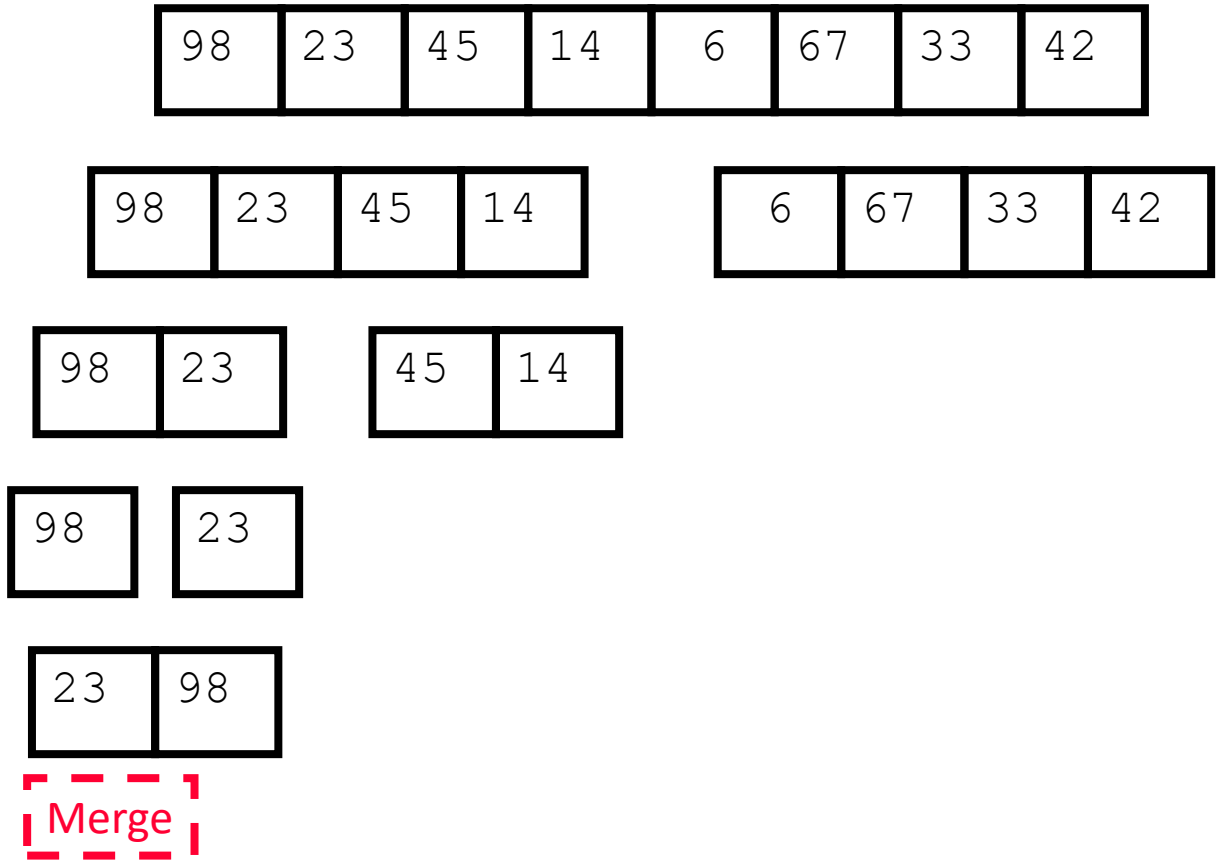
Another example



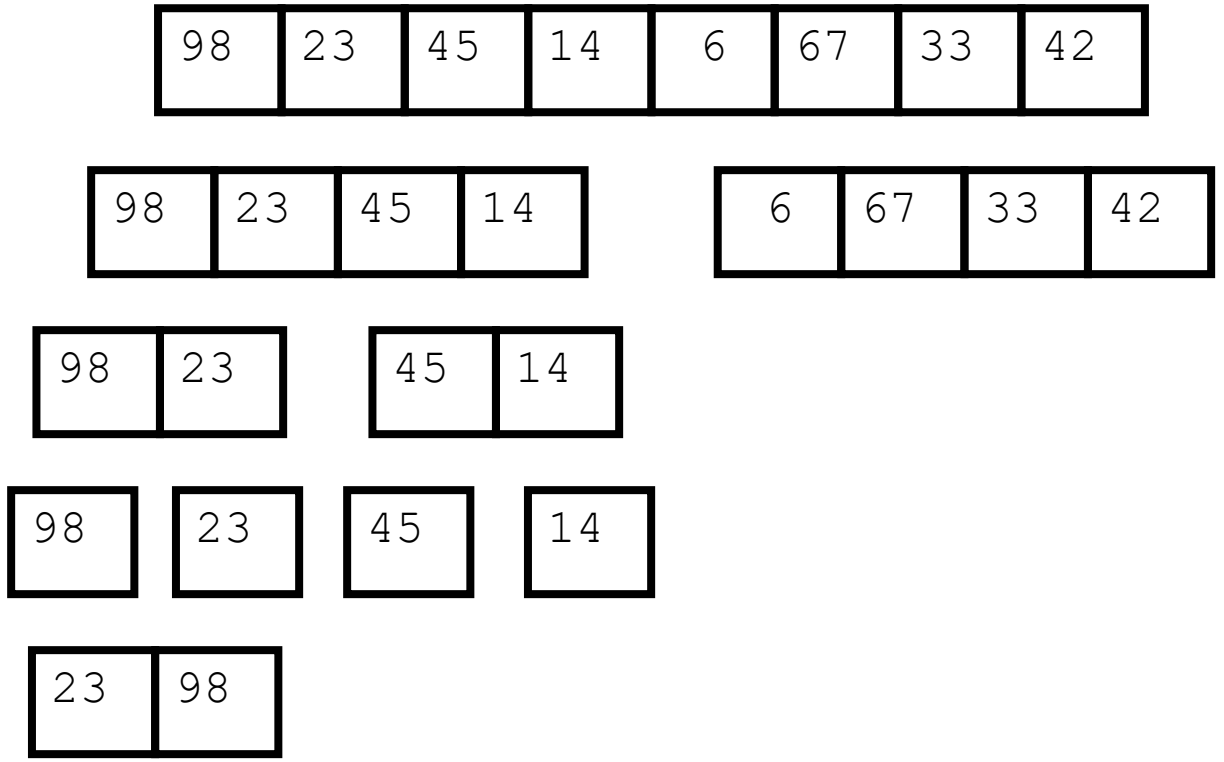
Another example



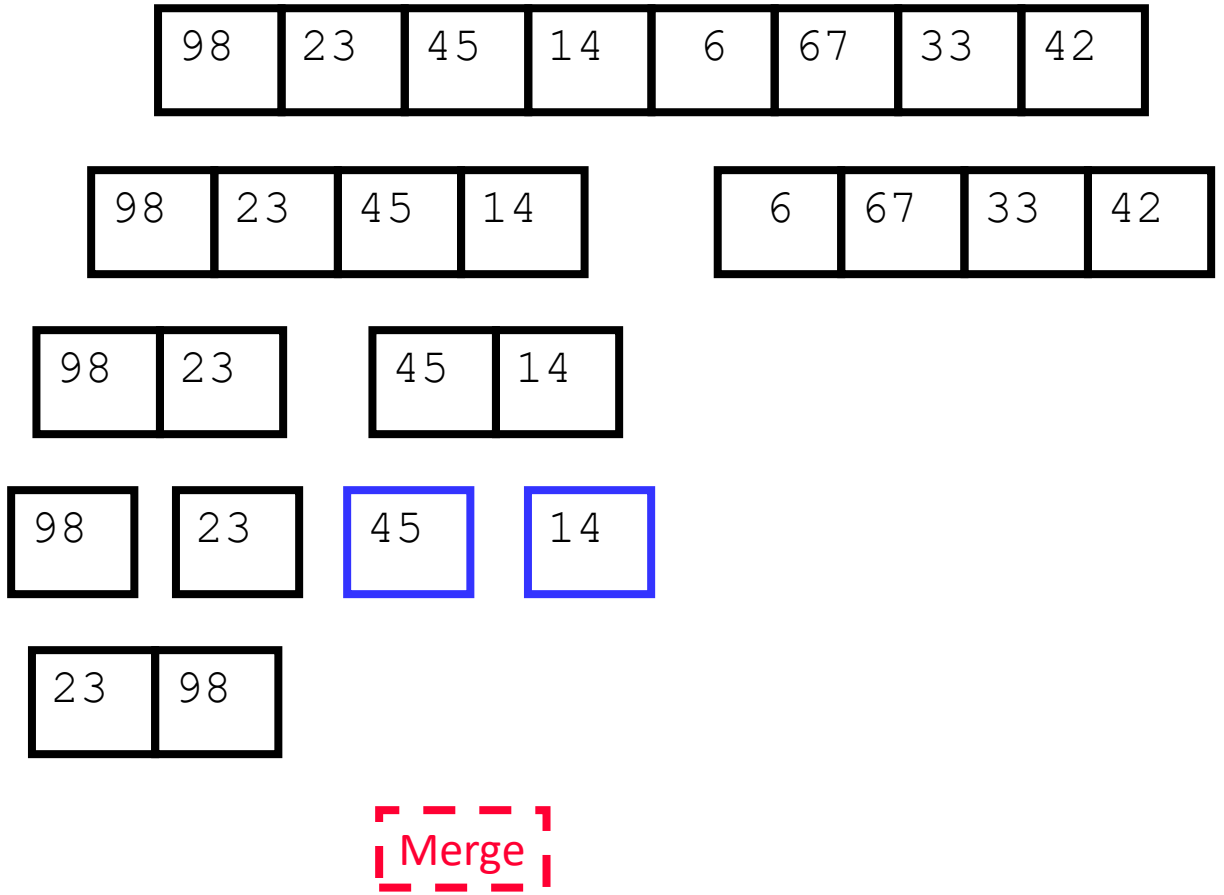
Another example



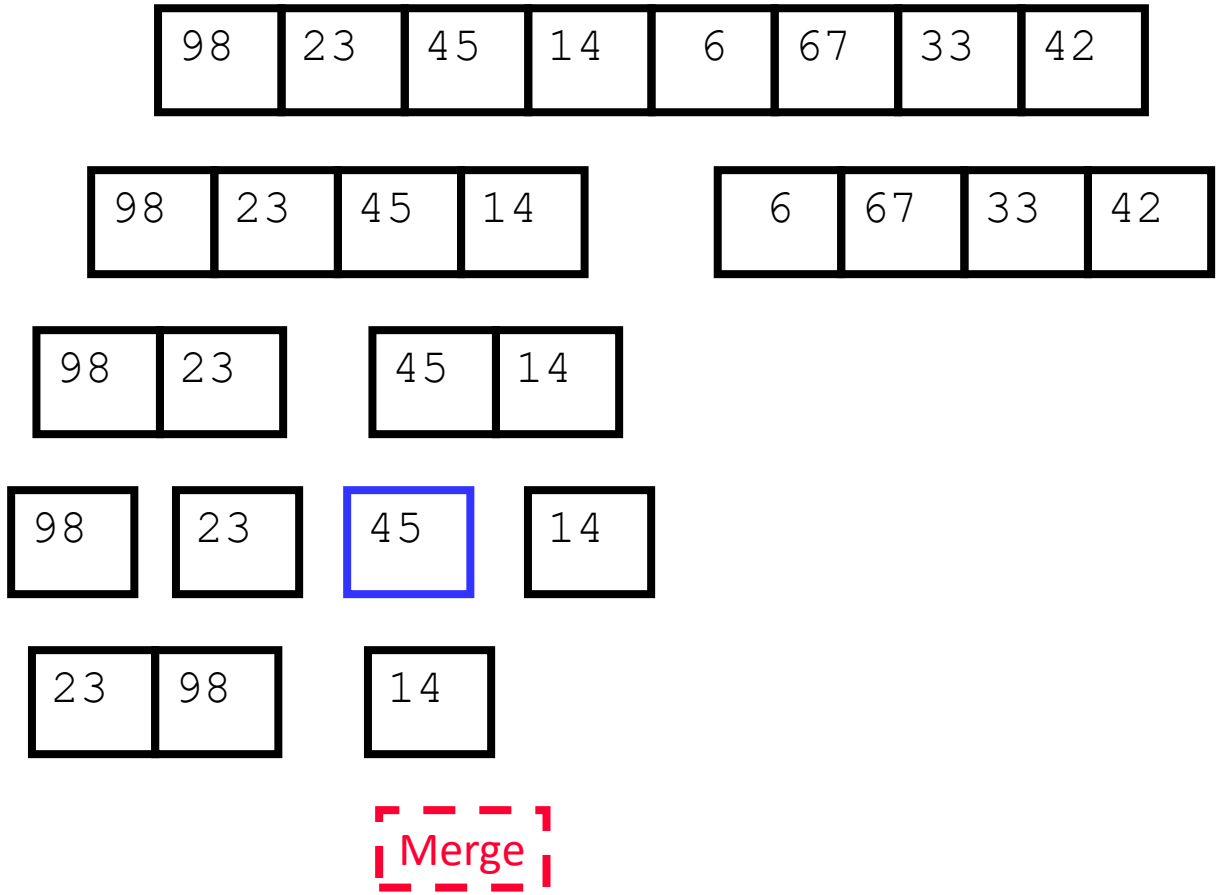
Another example



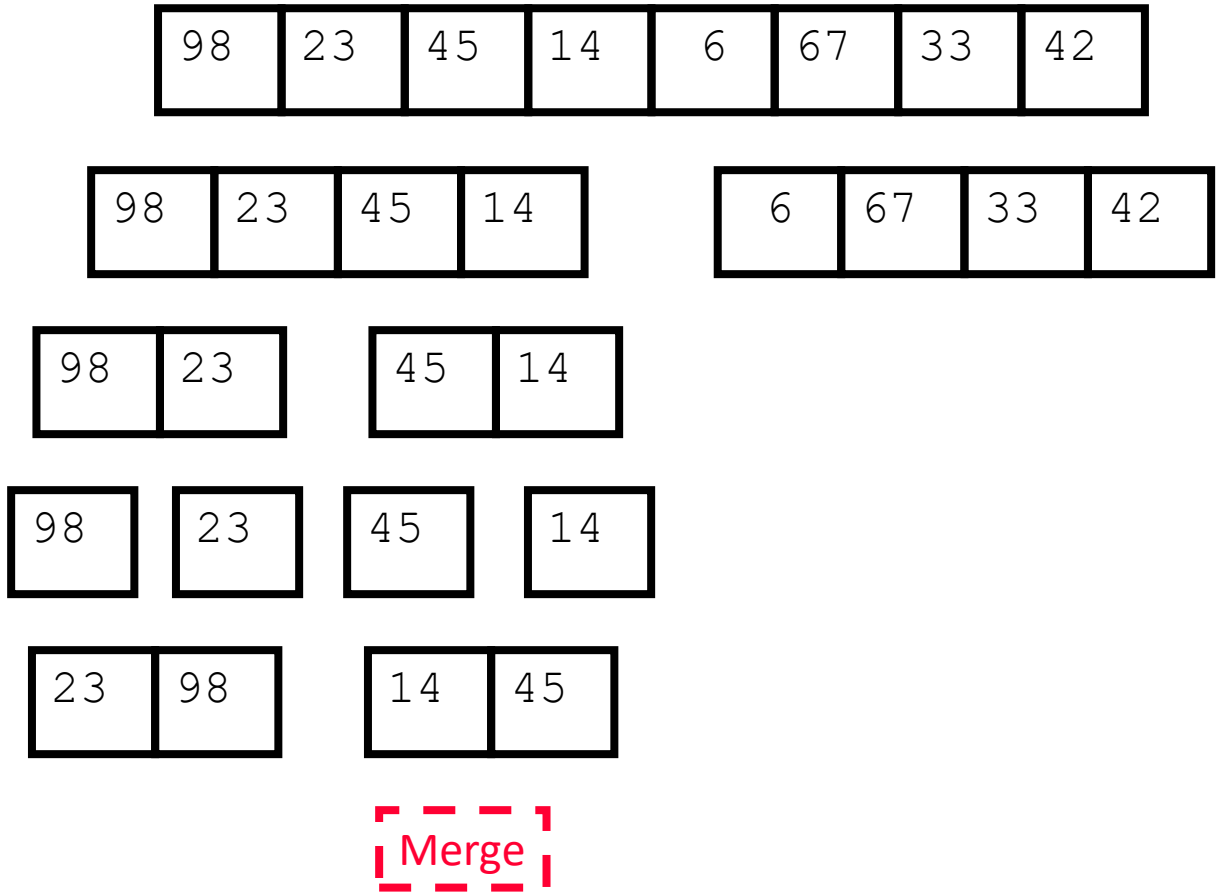
Another example



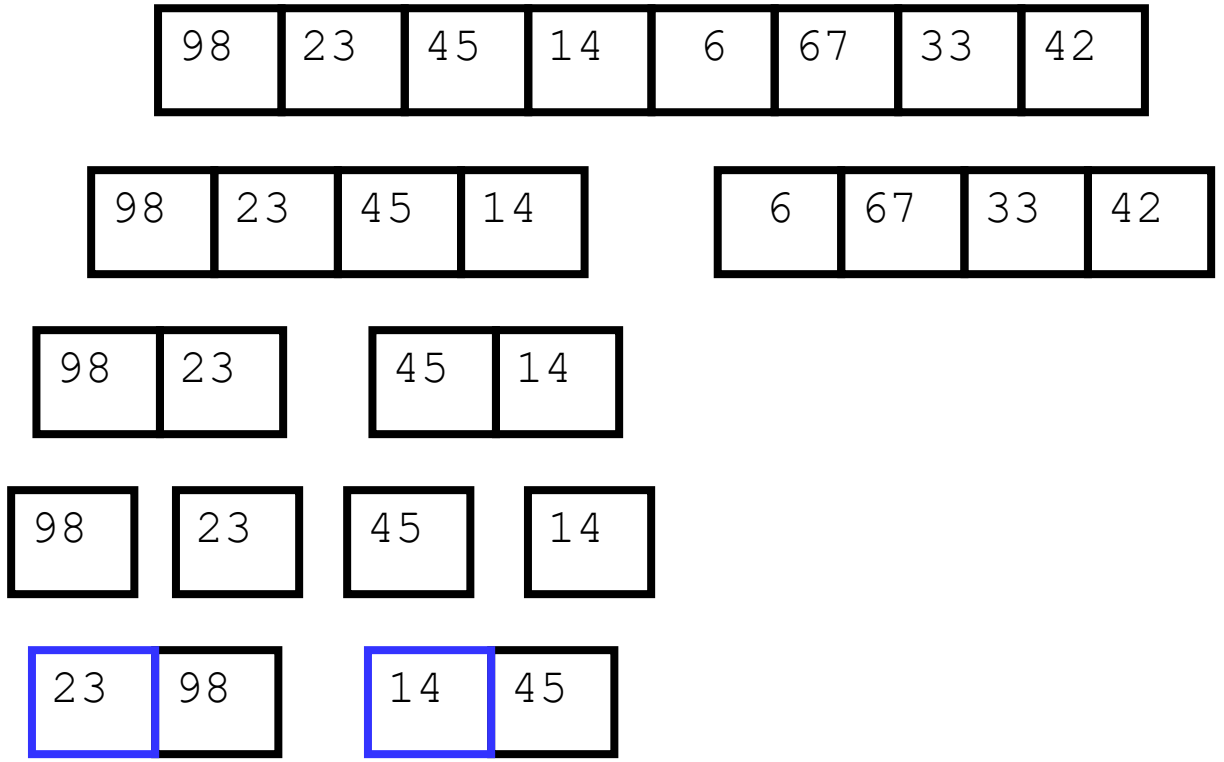
Another example



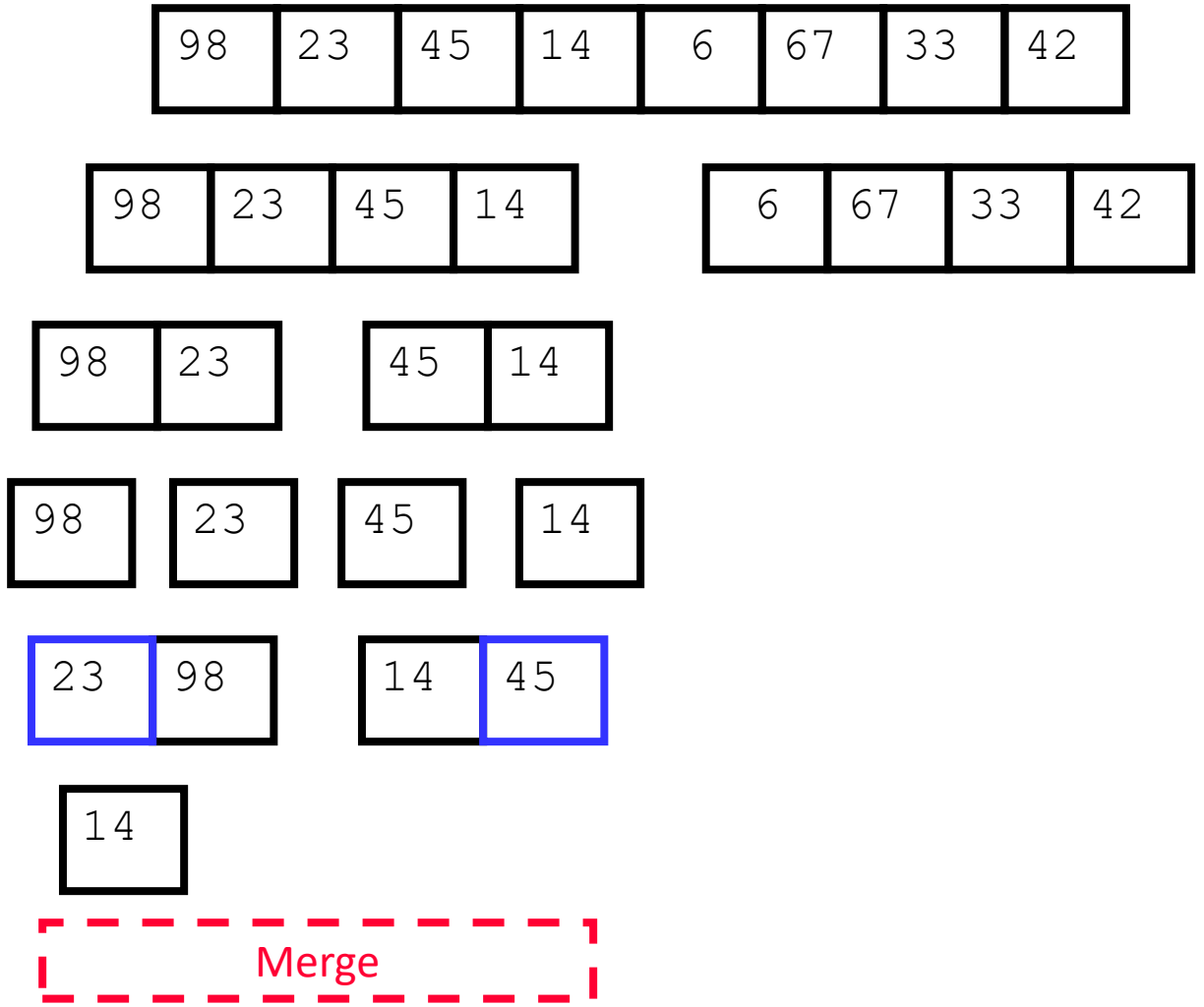
Another example



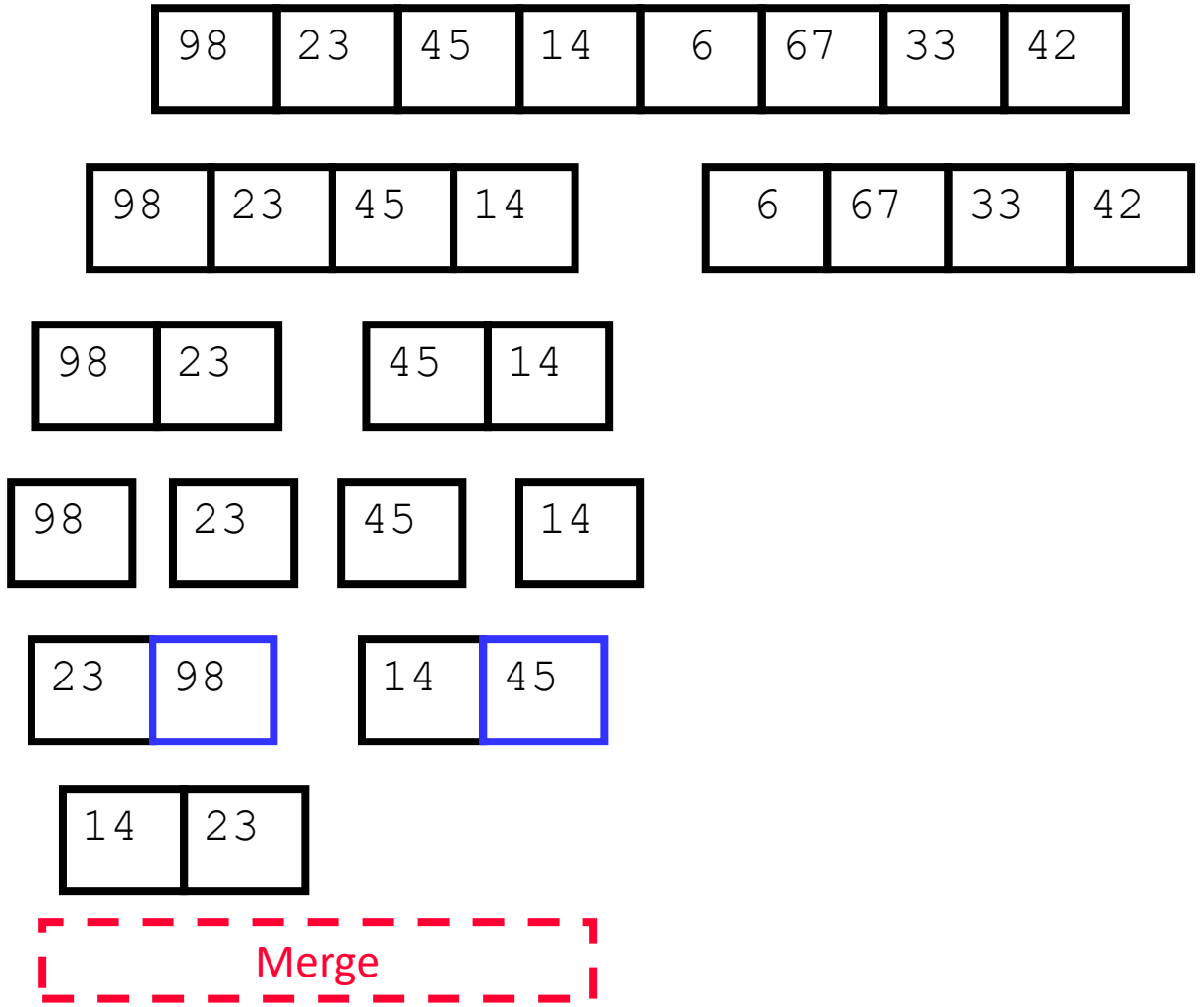
Another example



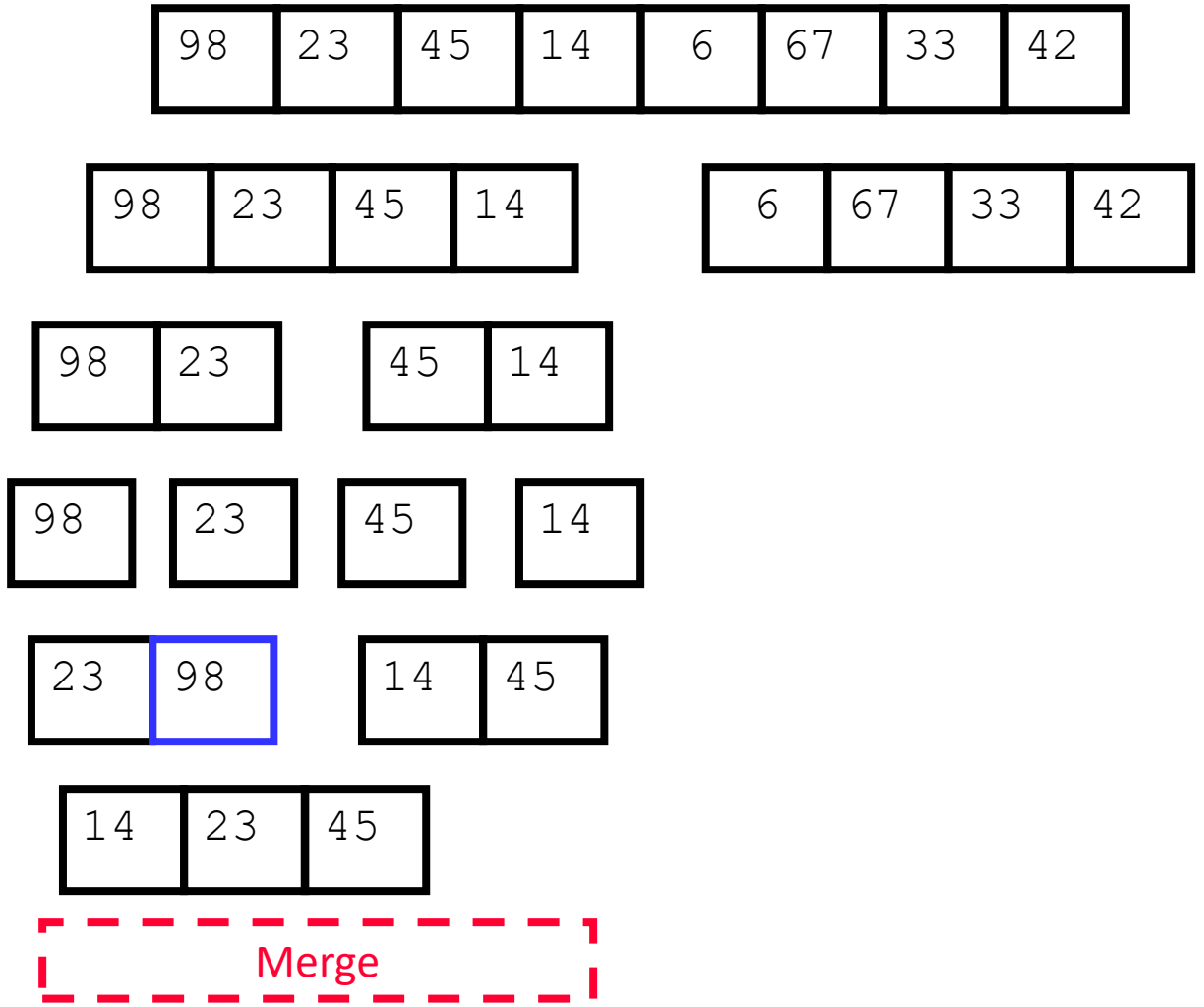
Another example



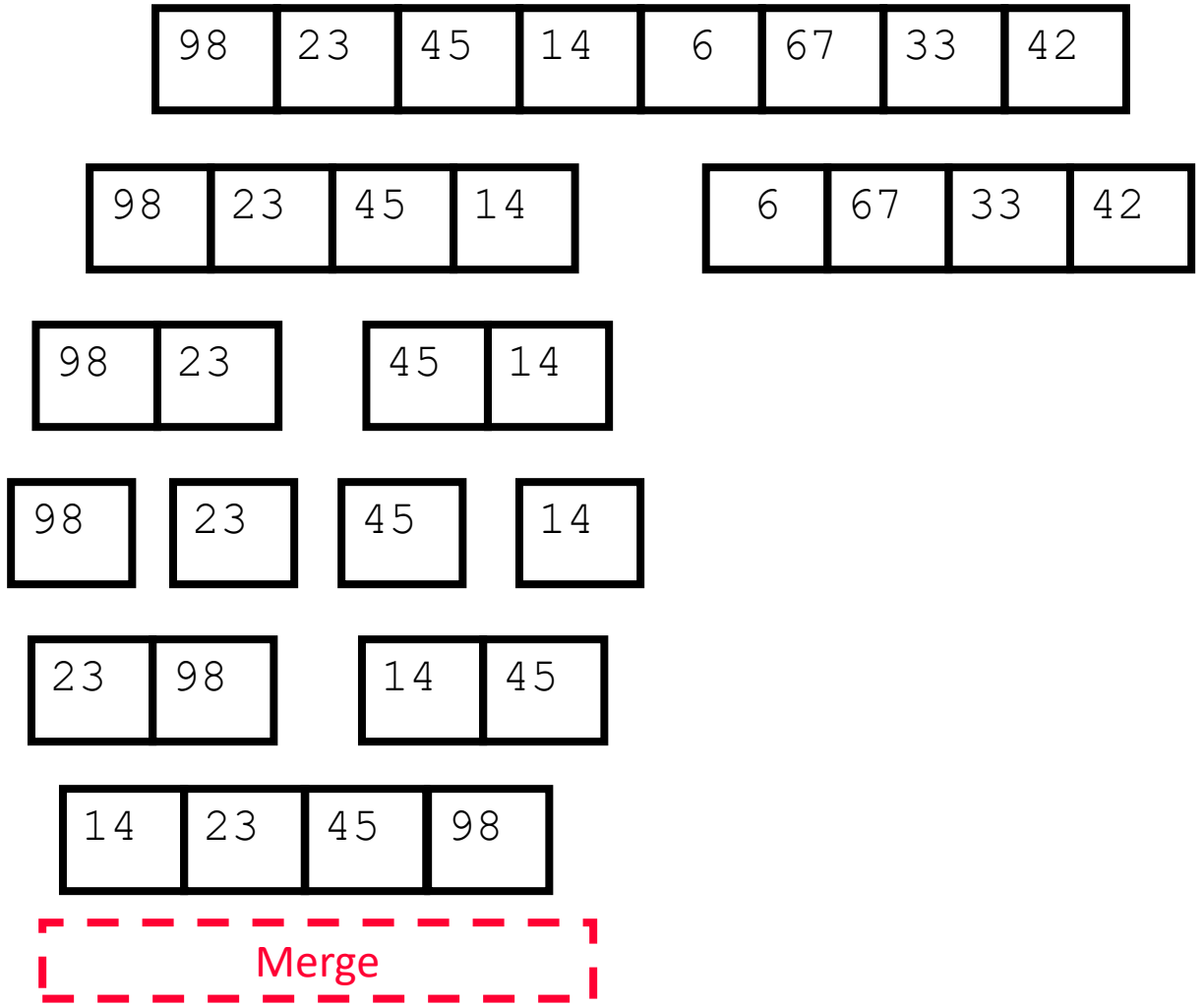
Another example



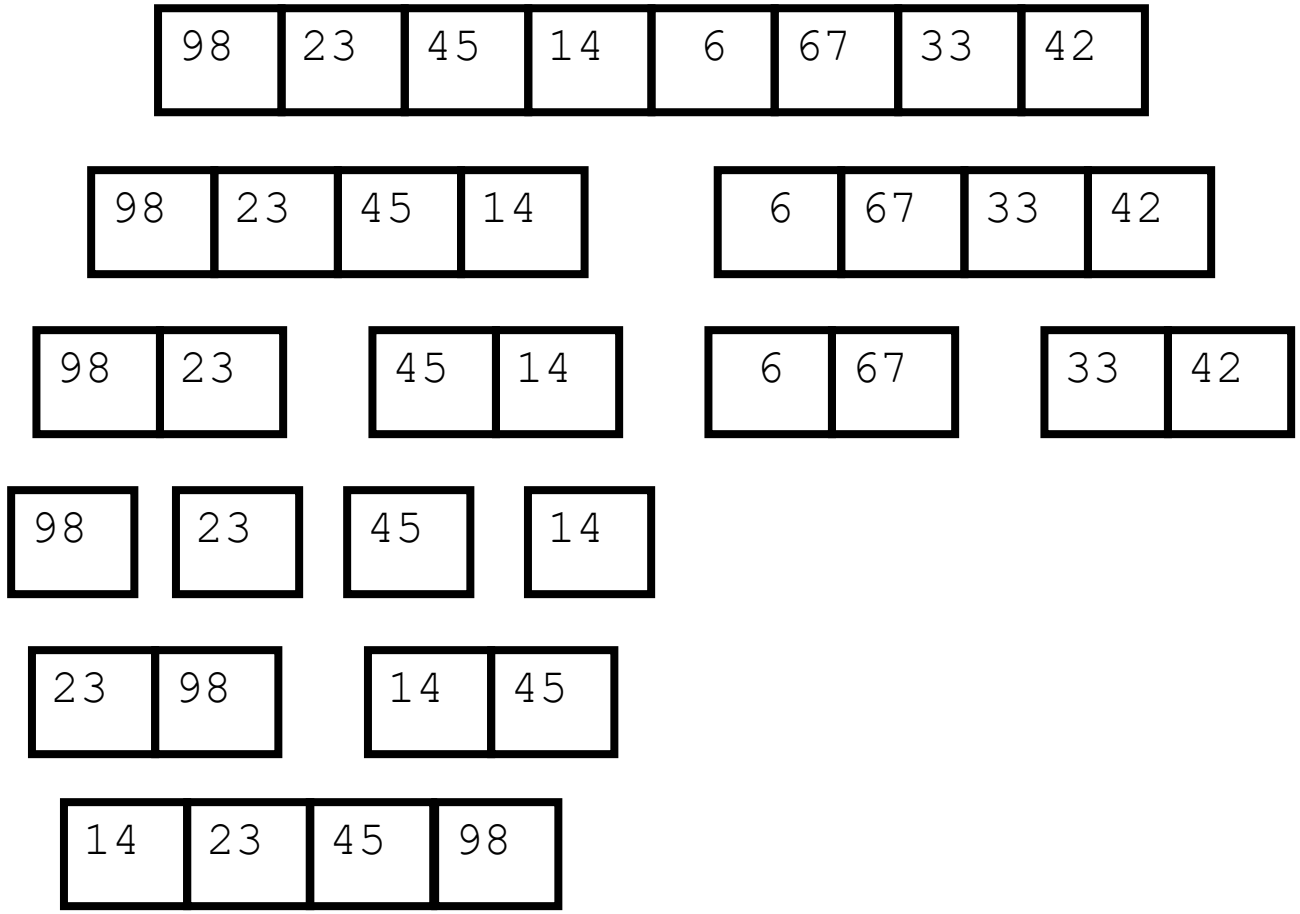
Another example



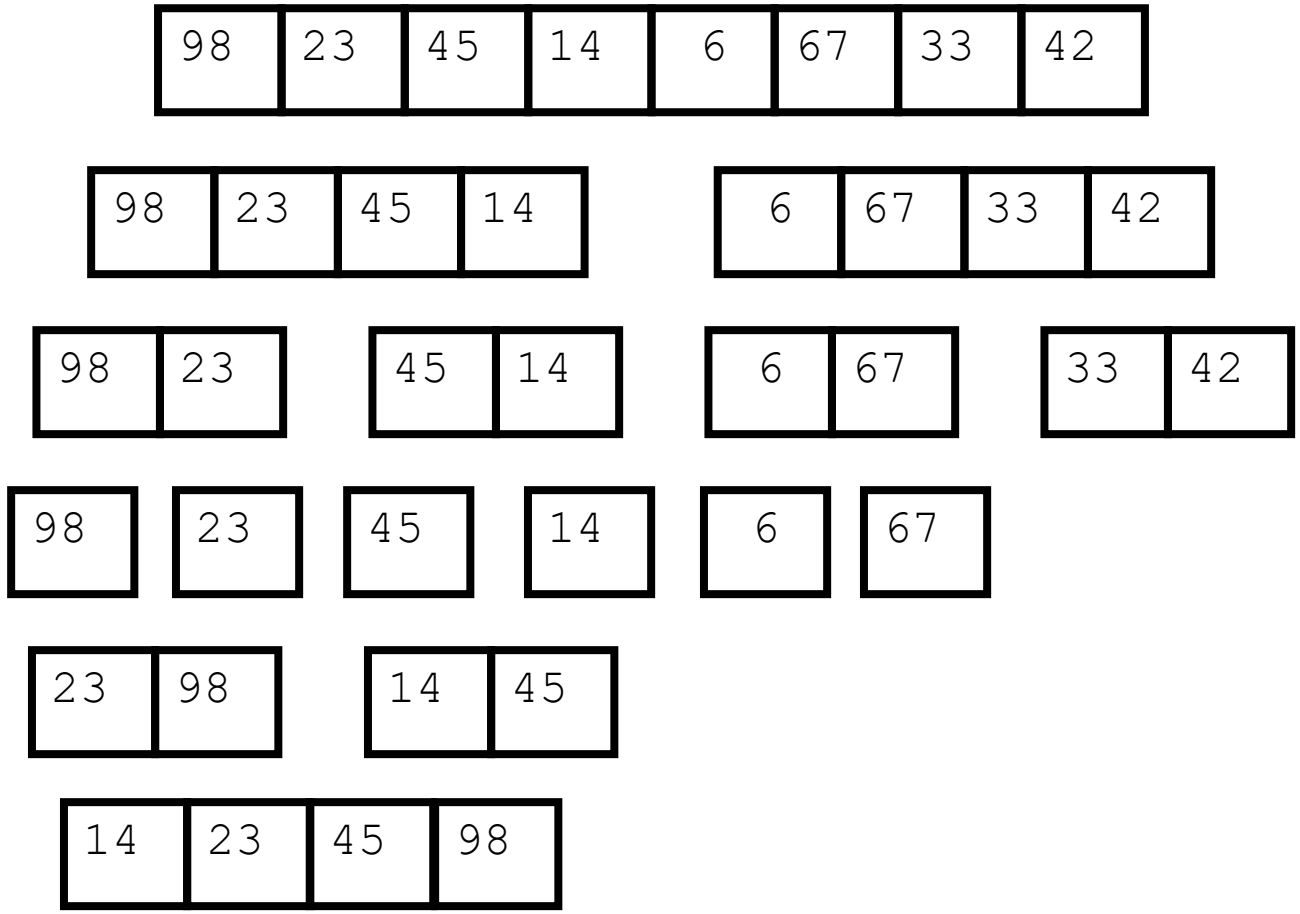
Another example



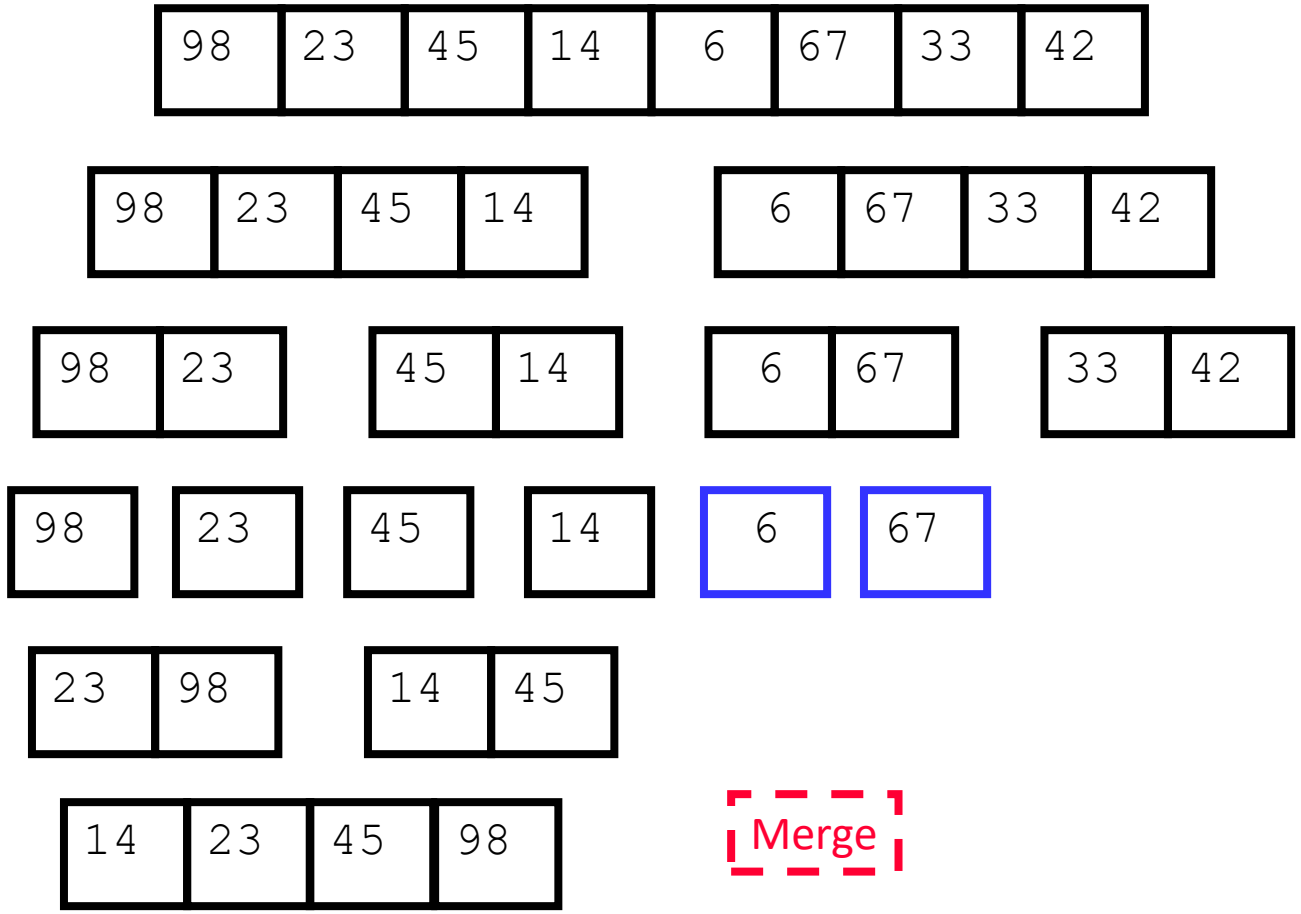
Another example



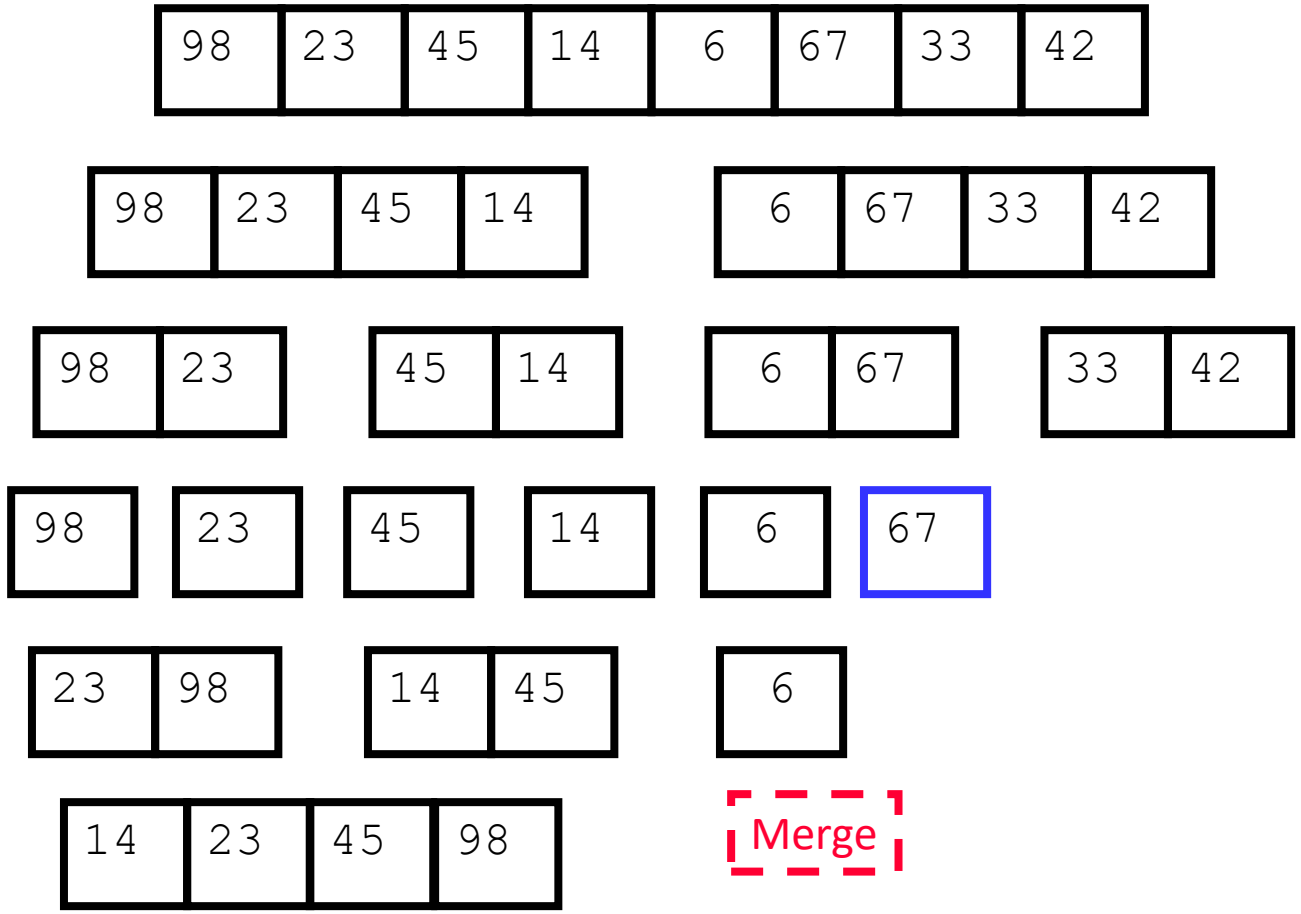
Another example



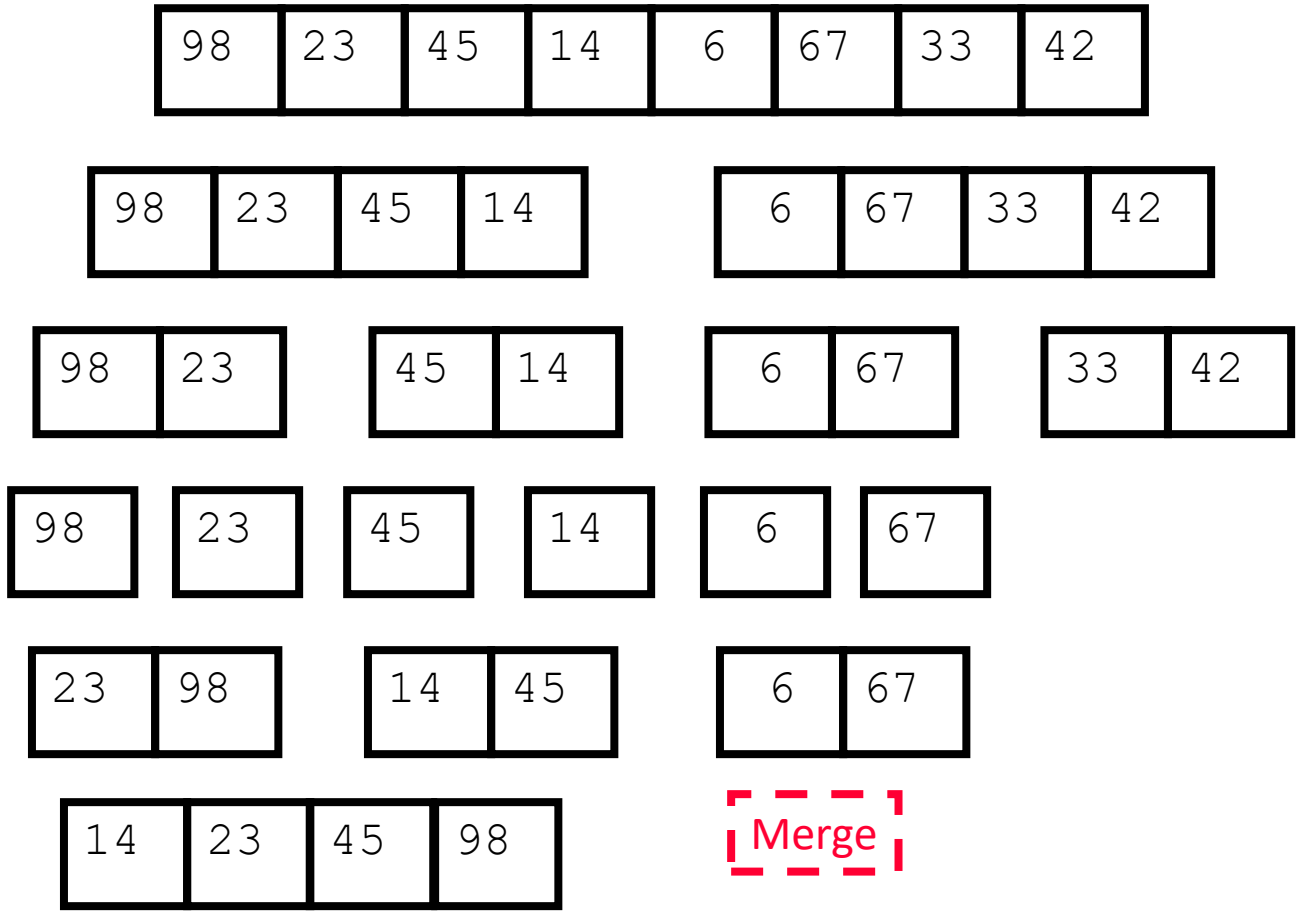
Another example



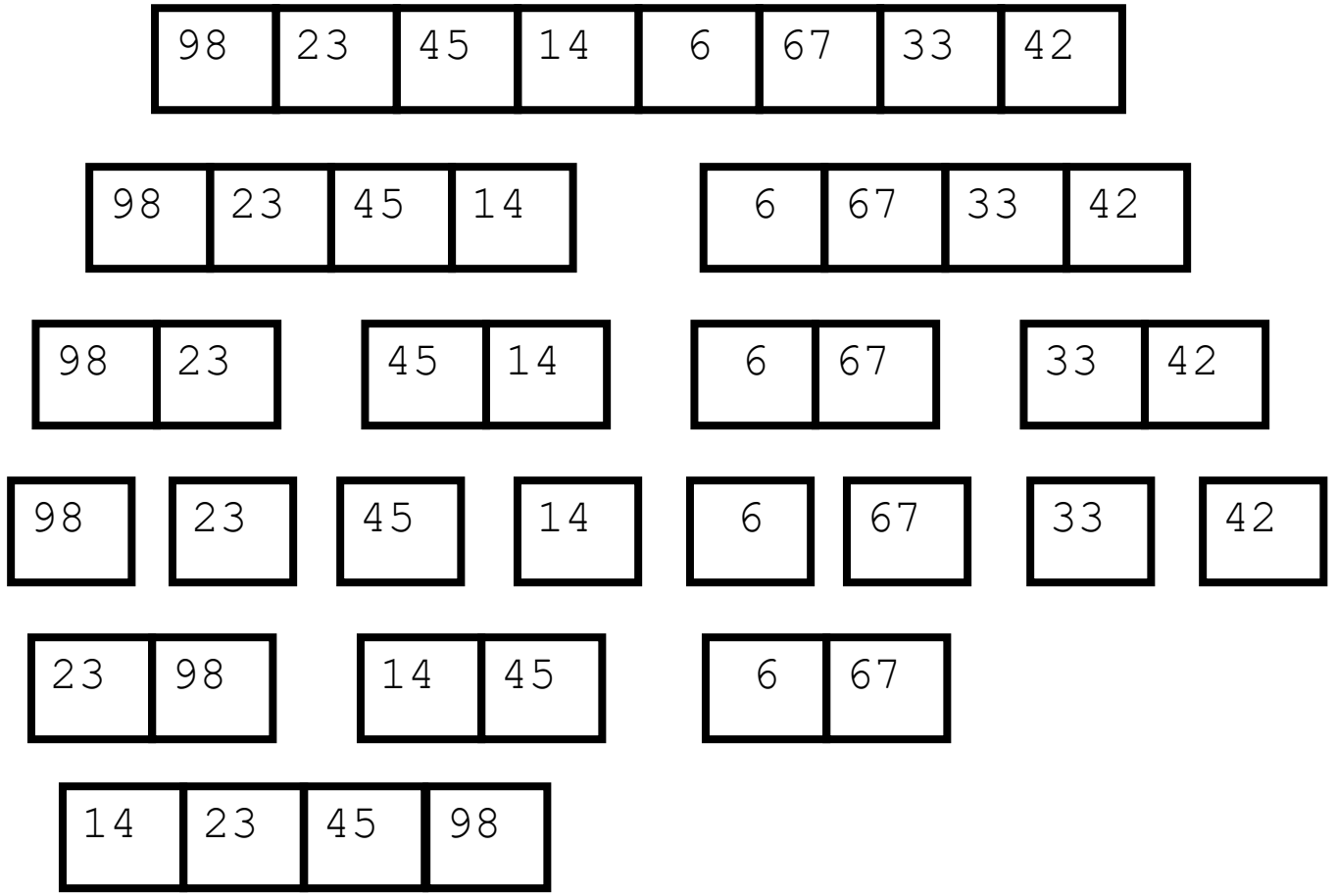
Another example



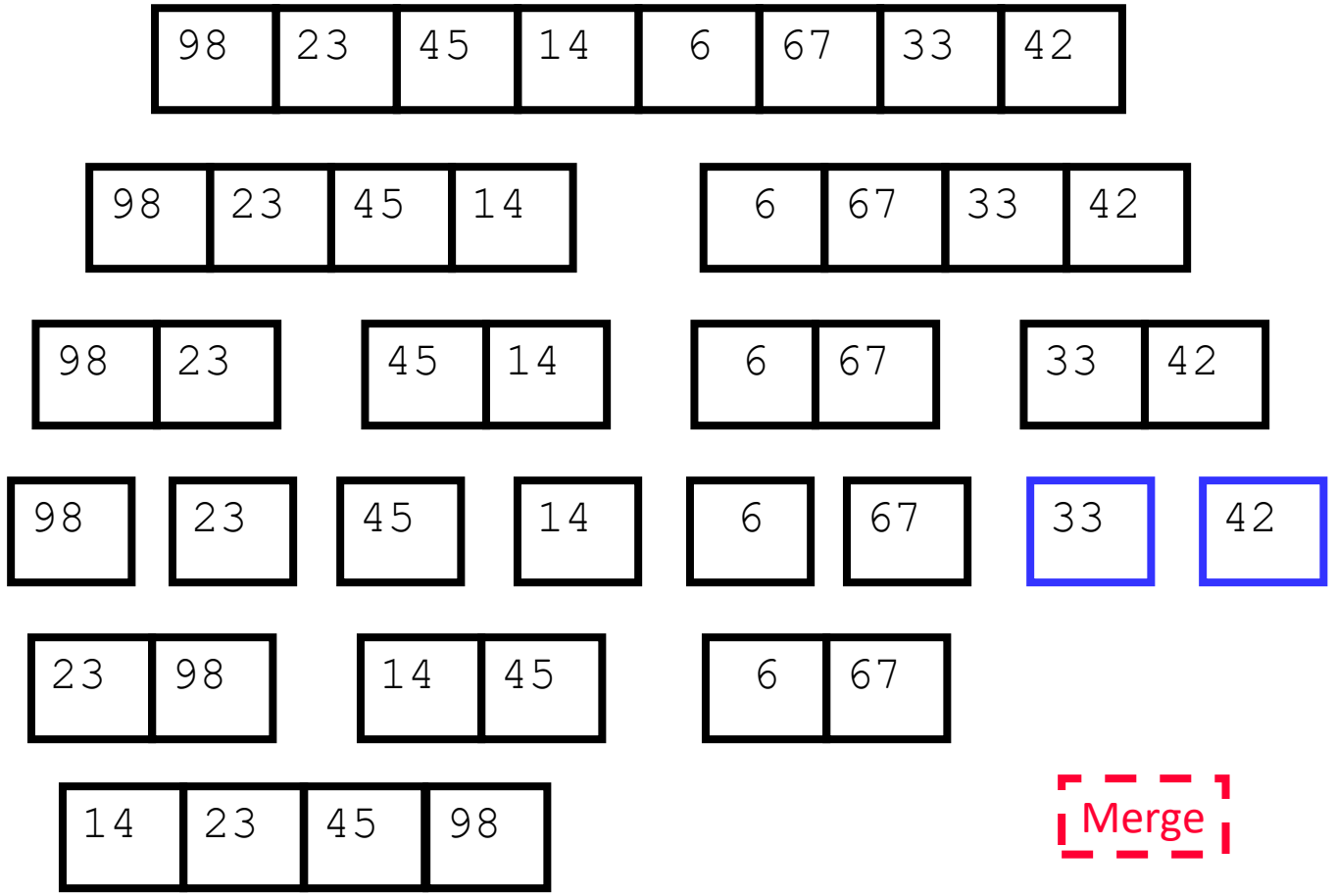
Another example



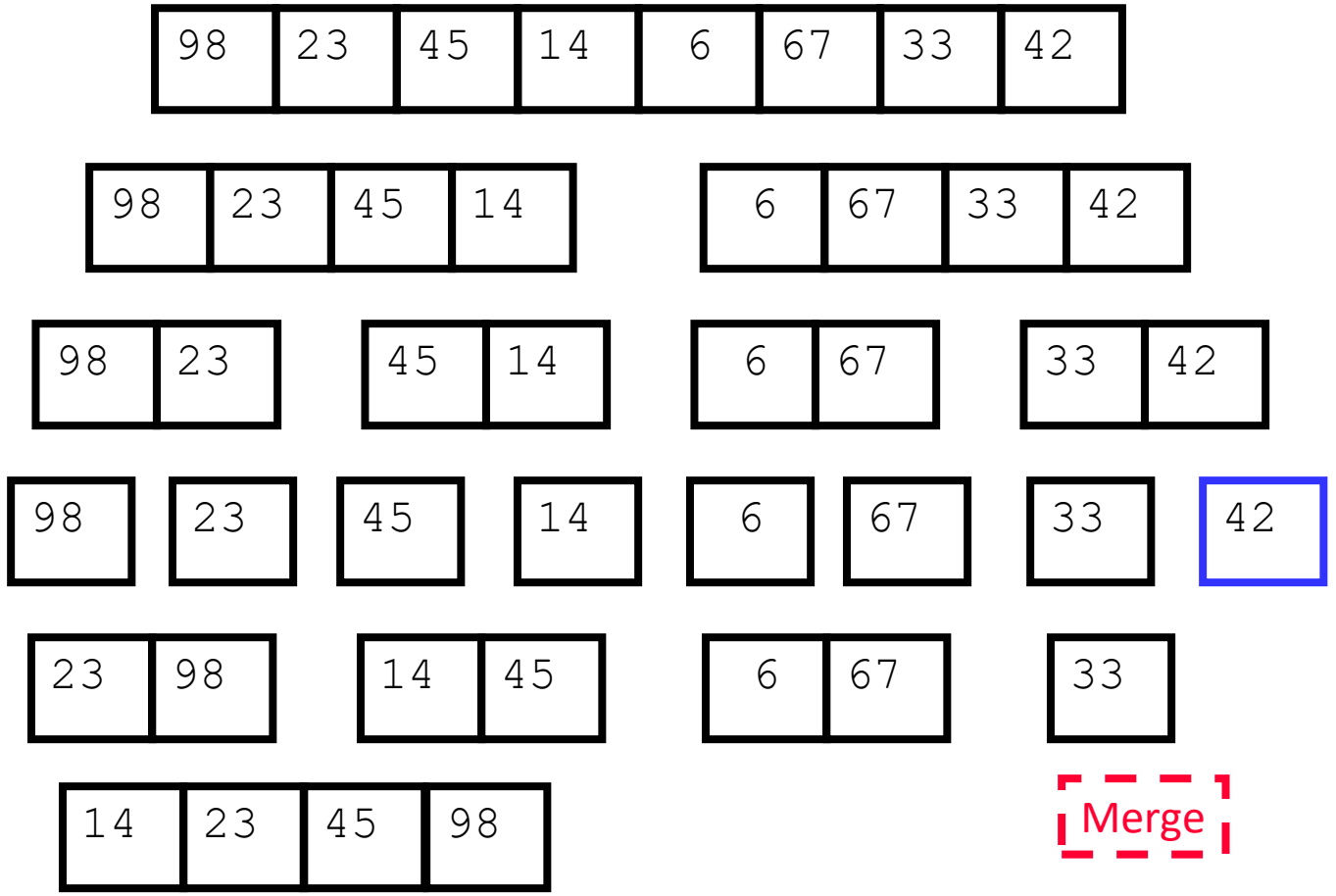
Another example



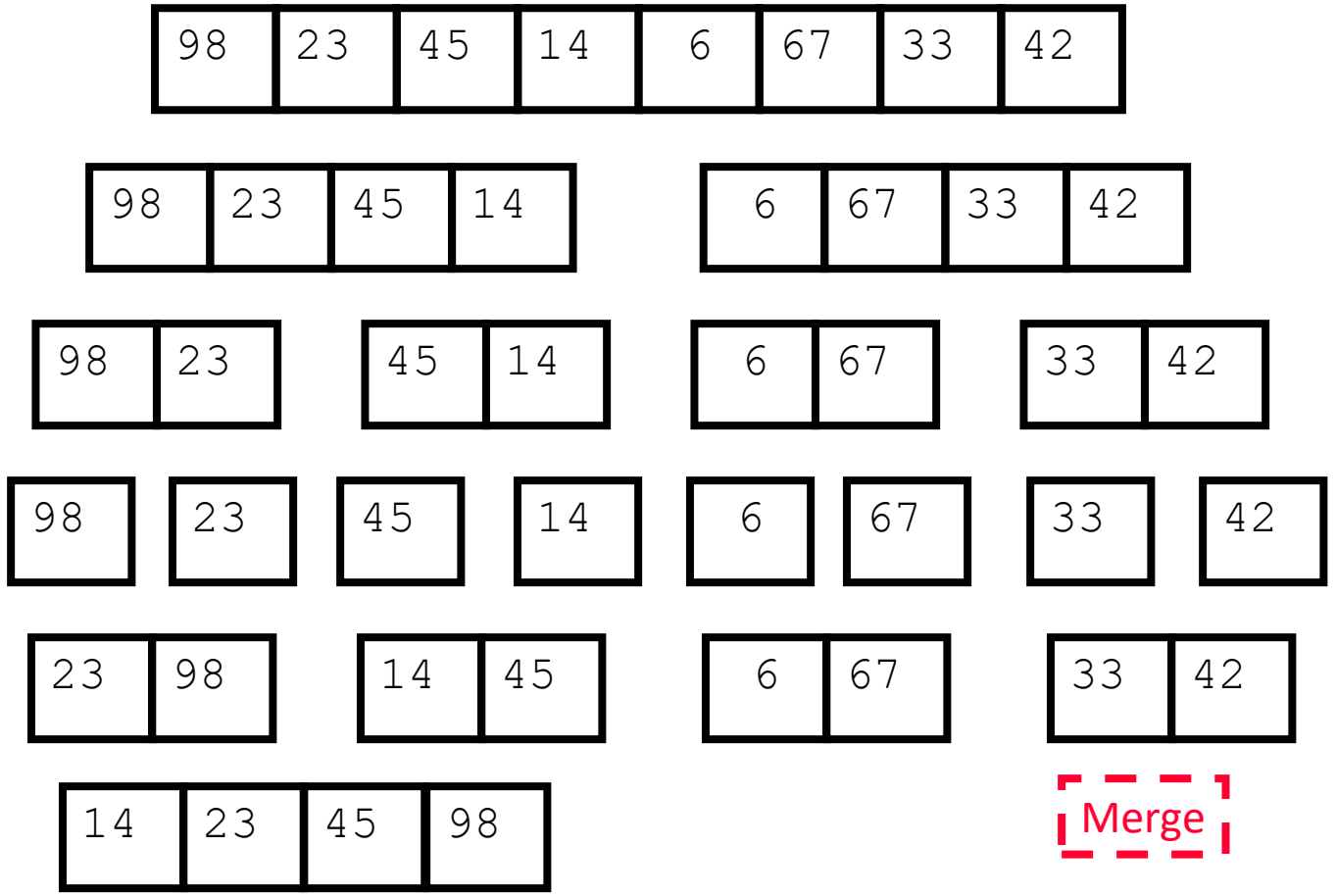
Another example



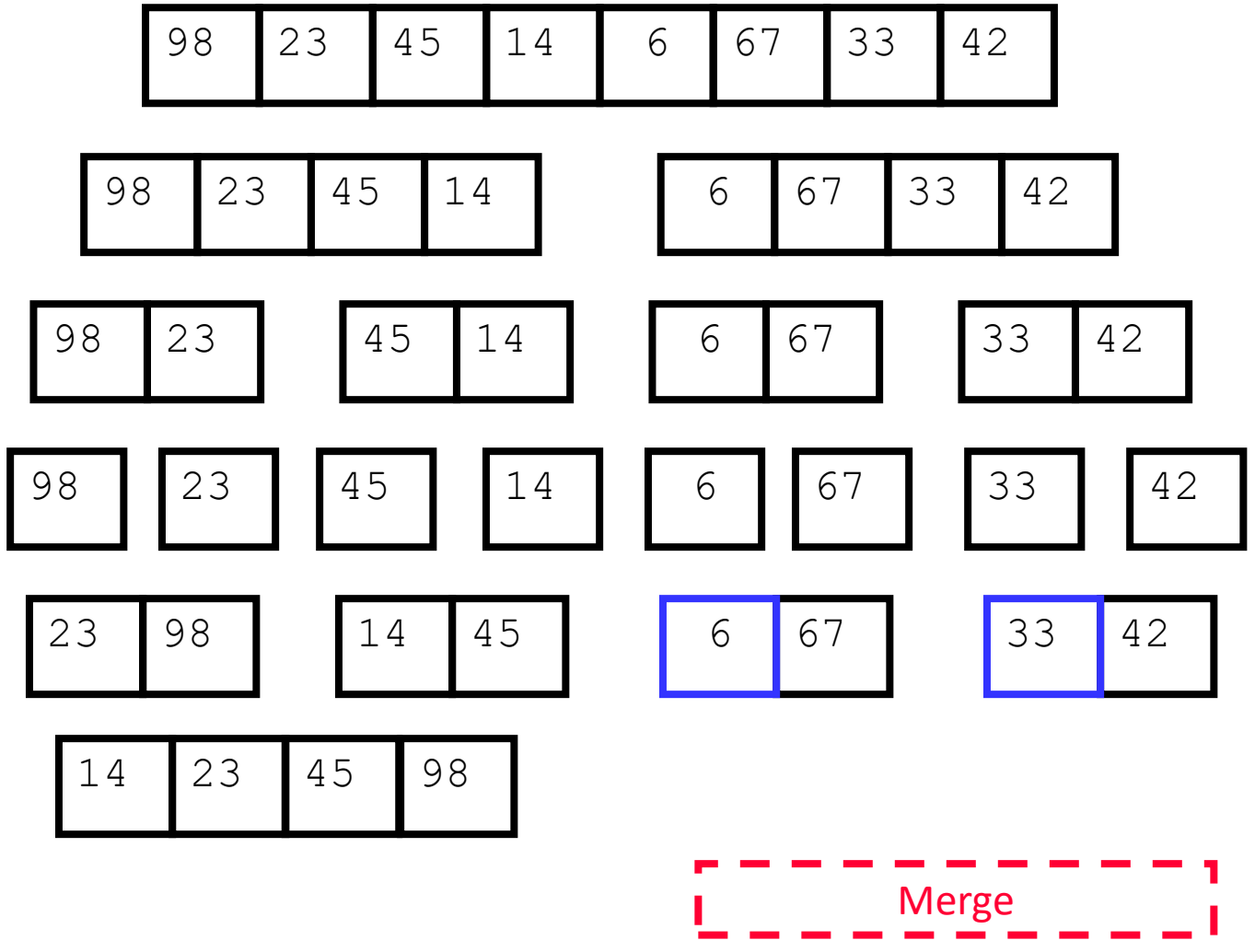
Another example



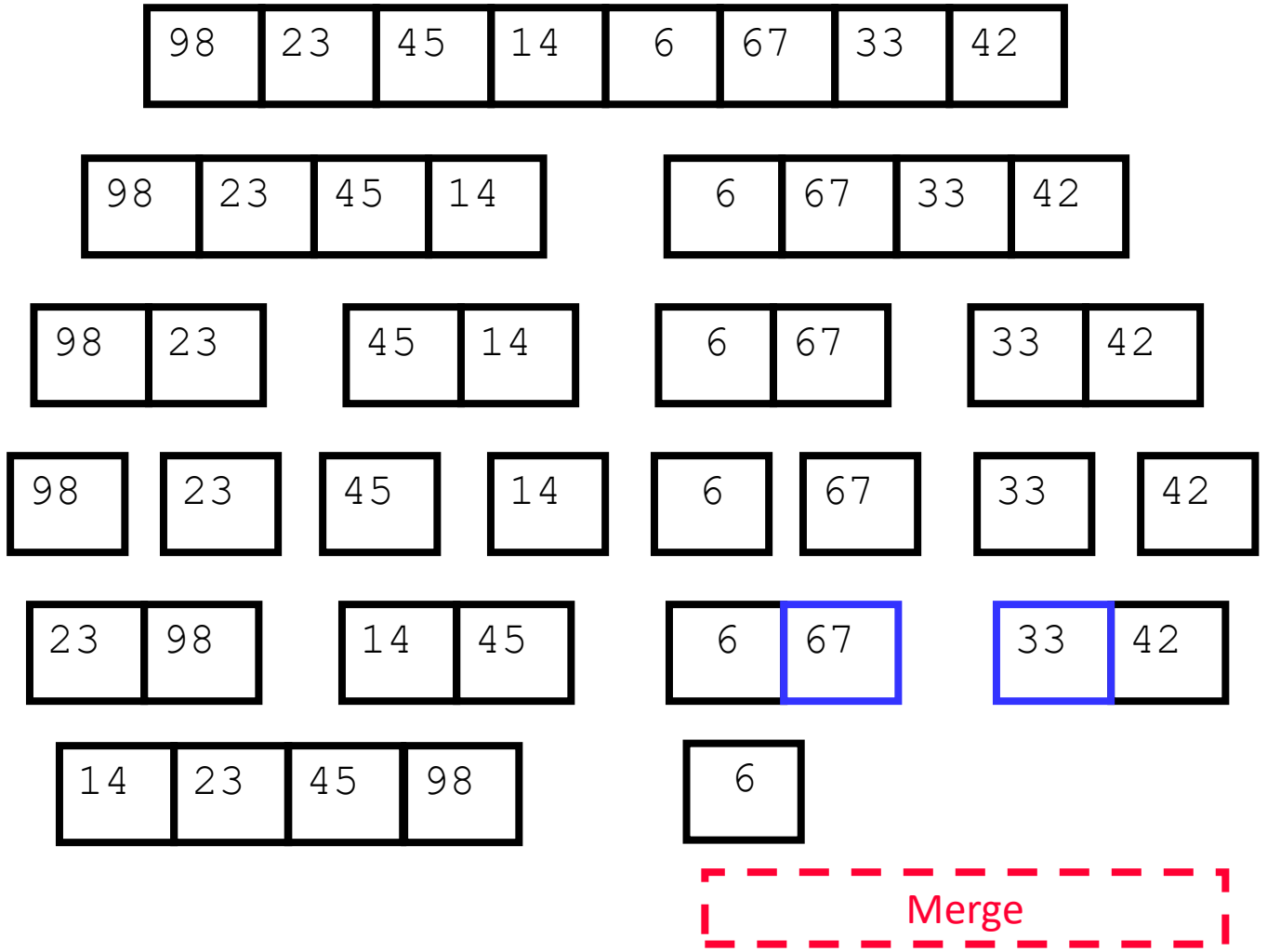
Another example



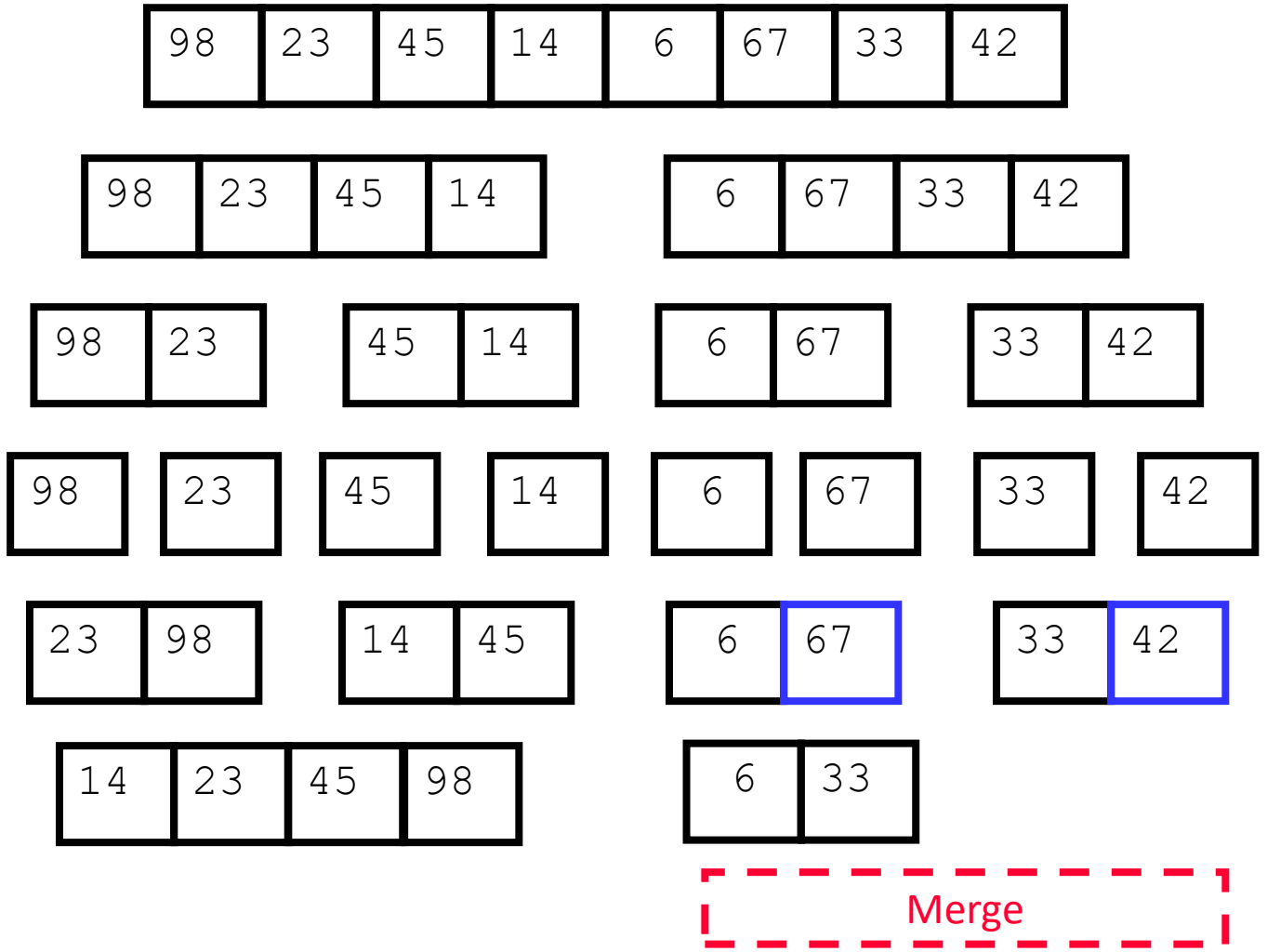
Another example



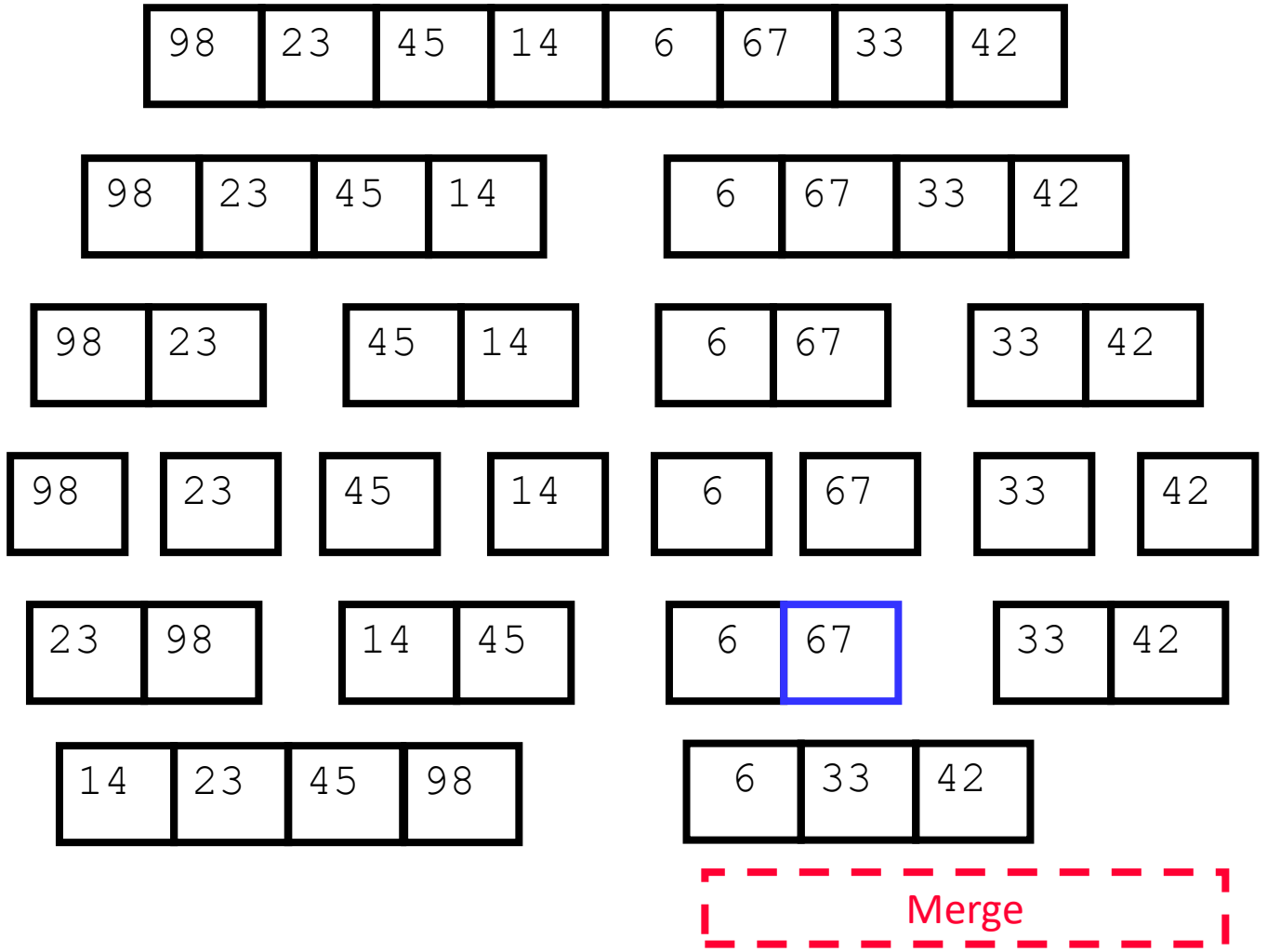
Another example



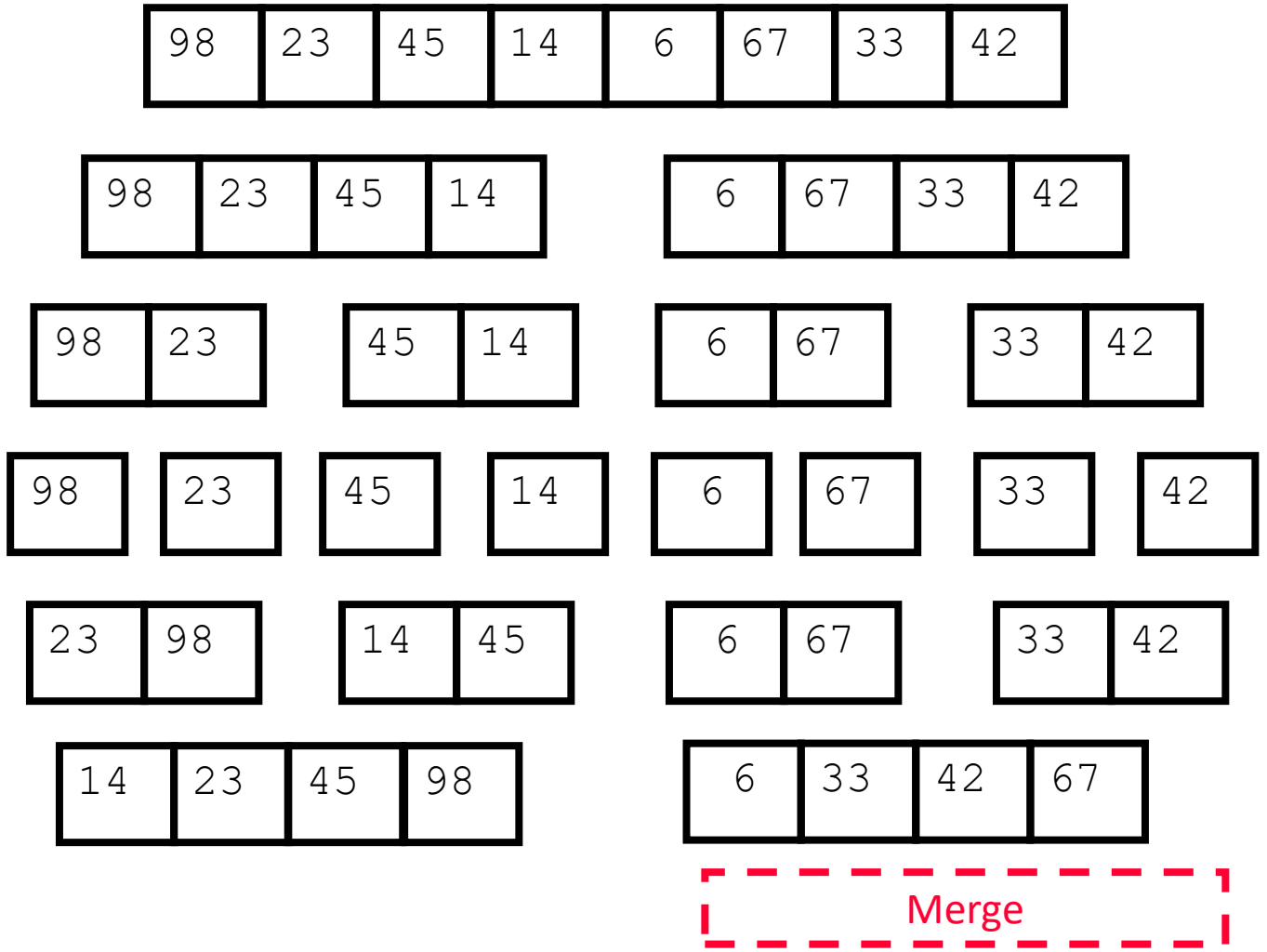
Another example



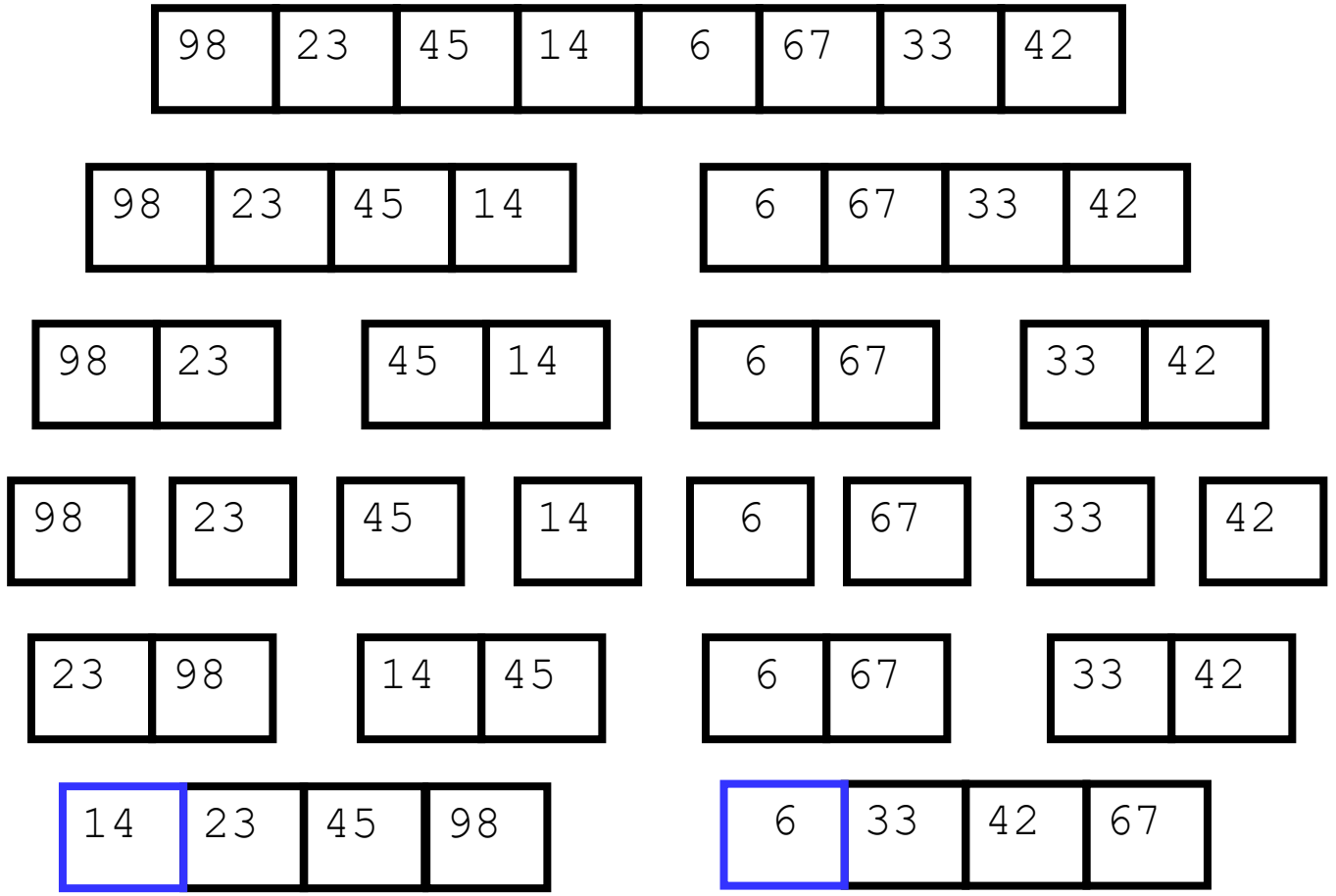
Another example



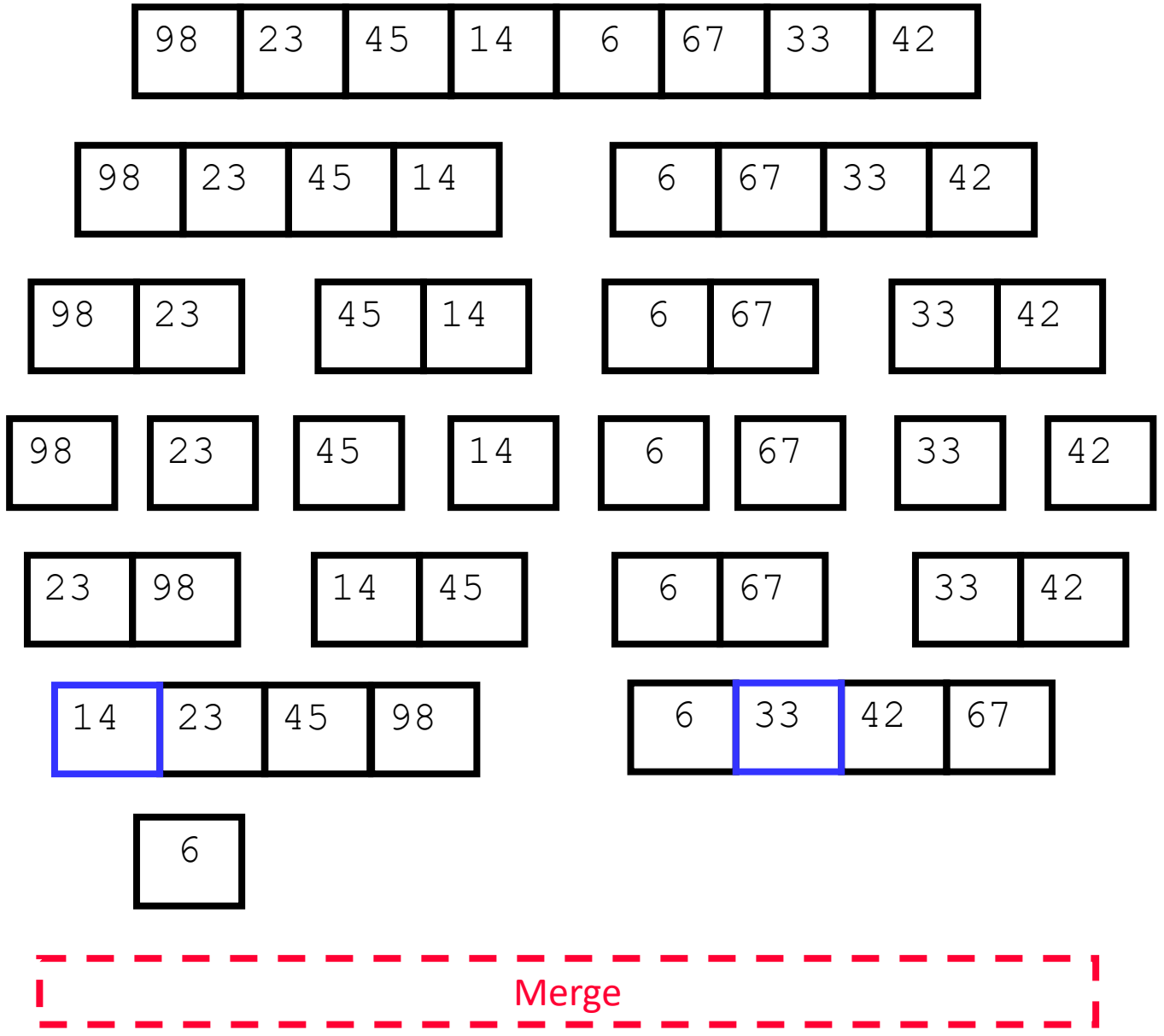
Another example



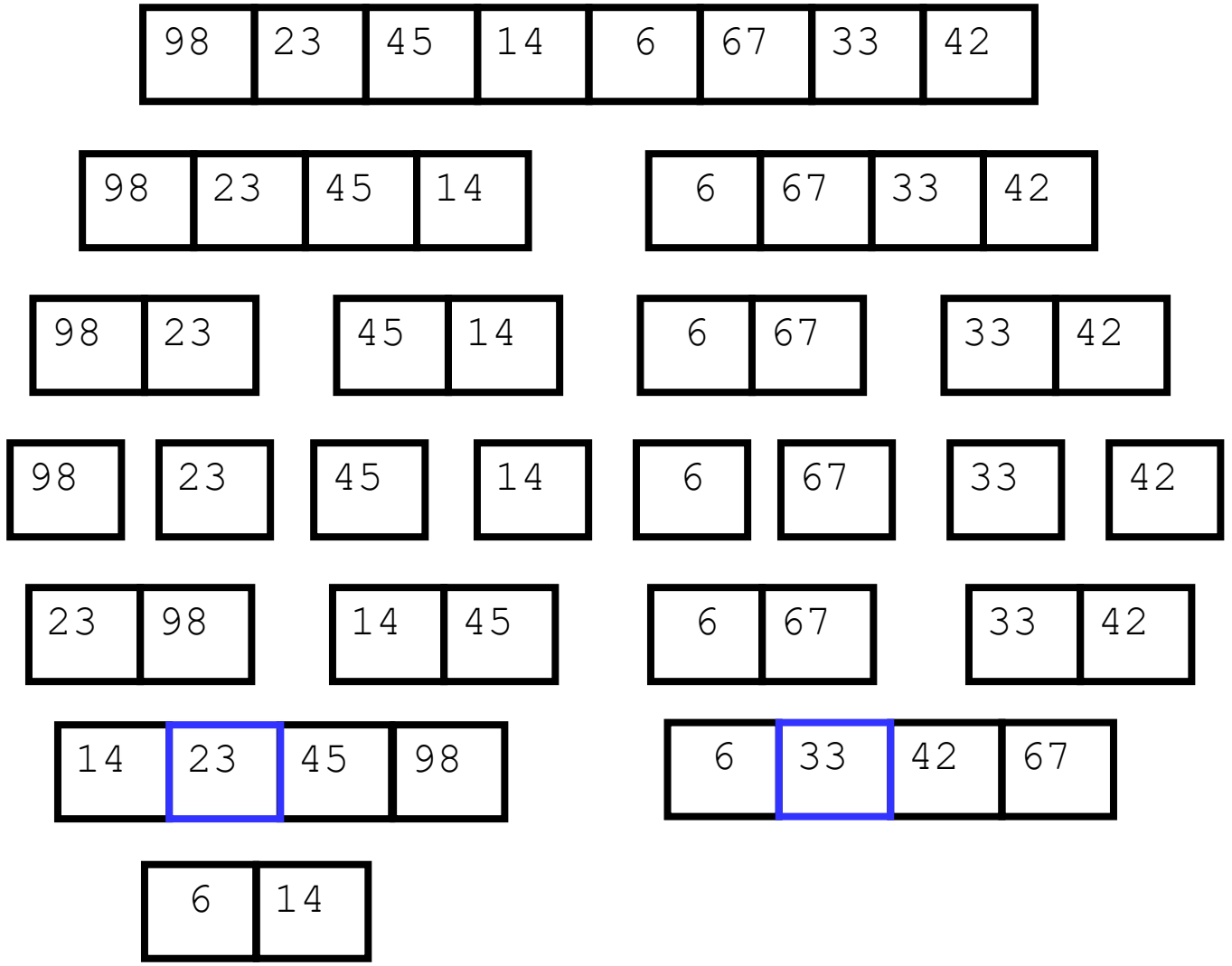
Another example



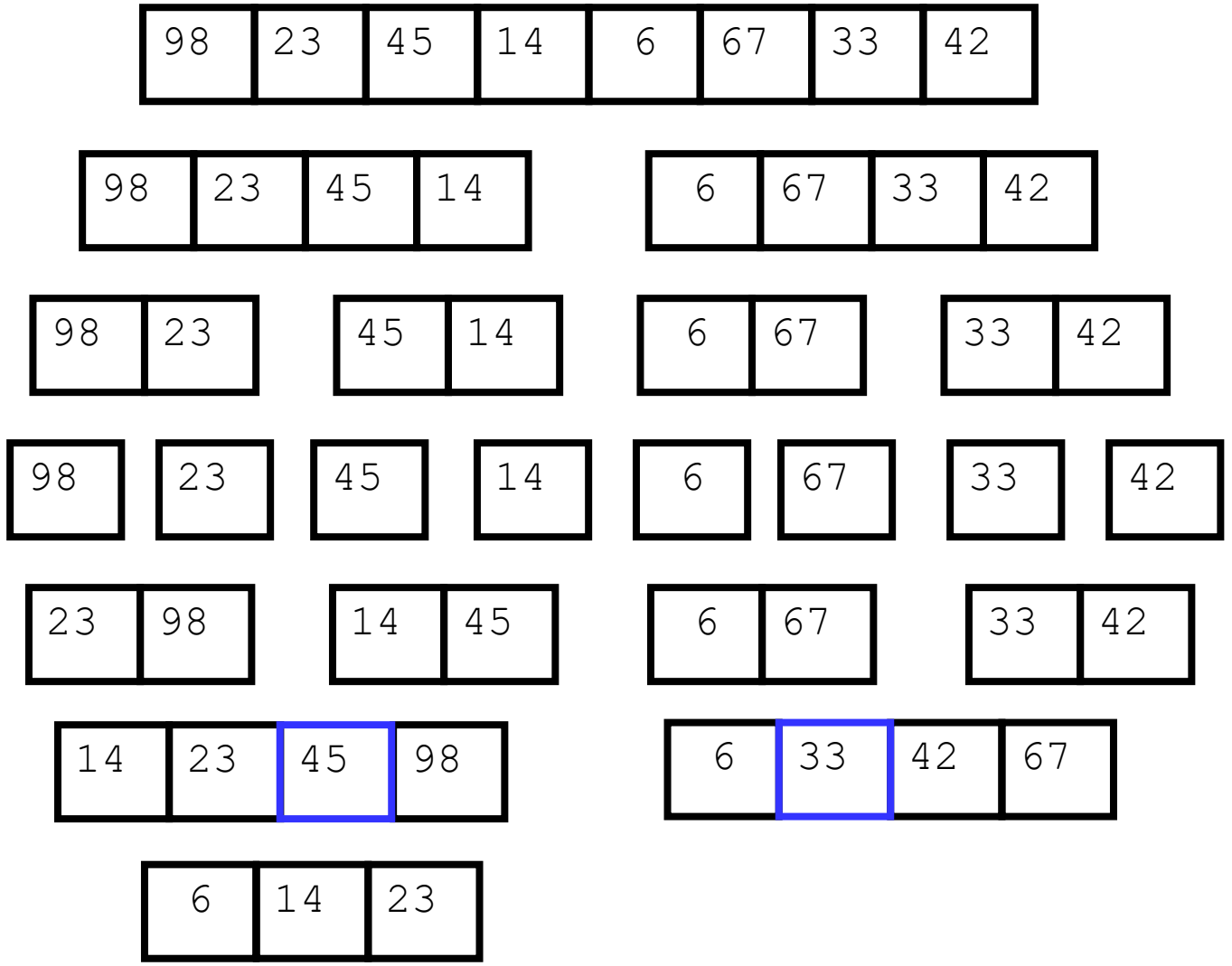
Another example



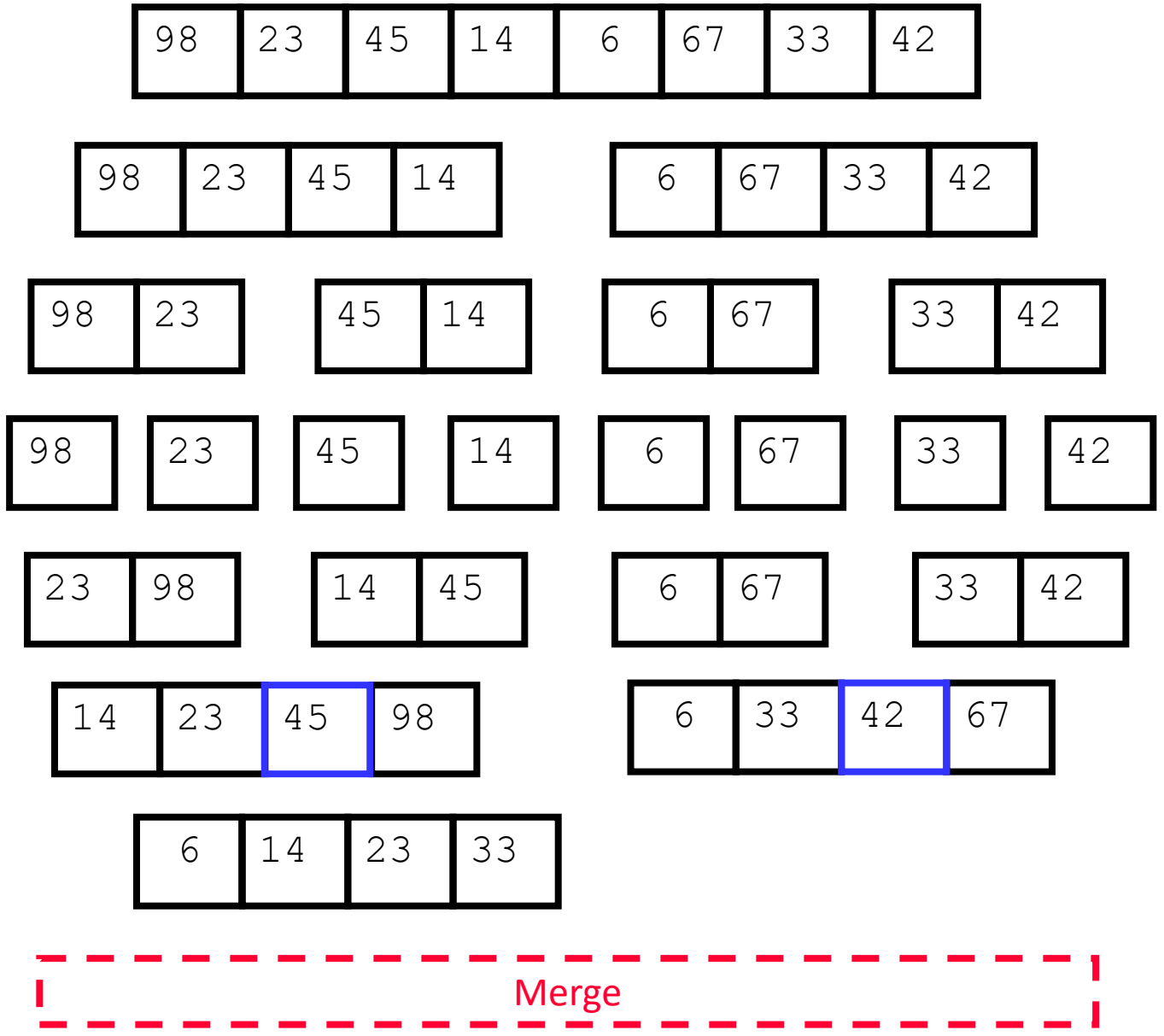
Another example



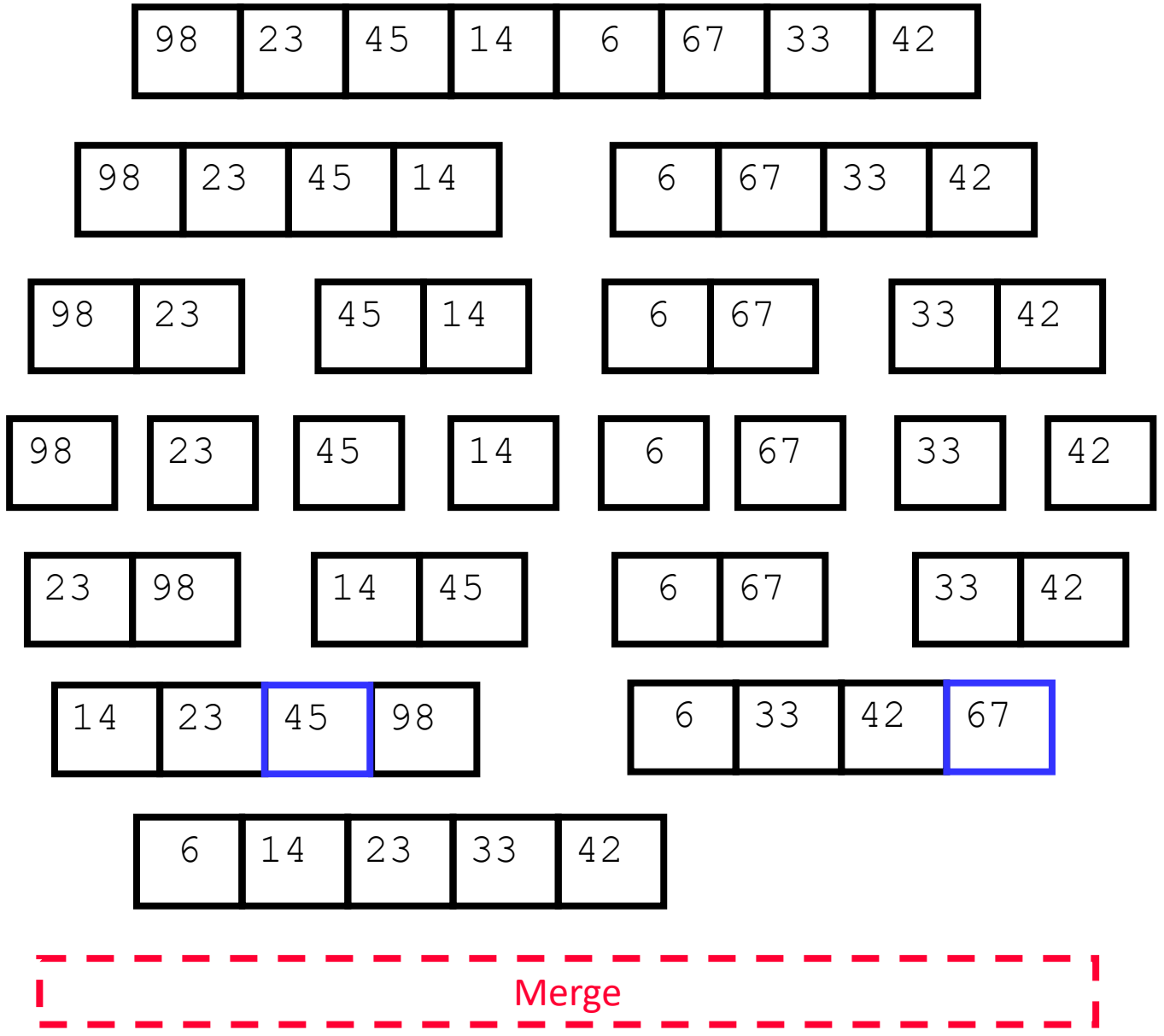
Another example



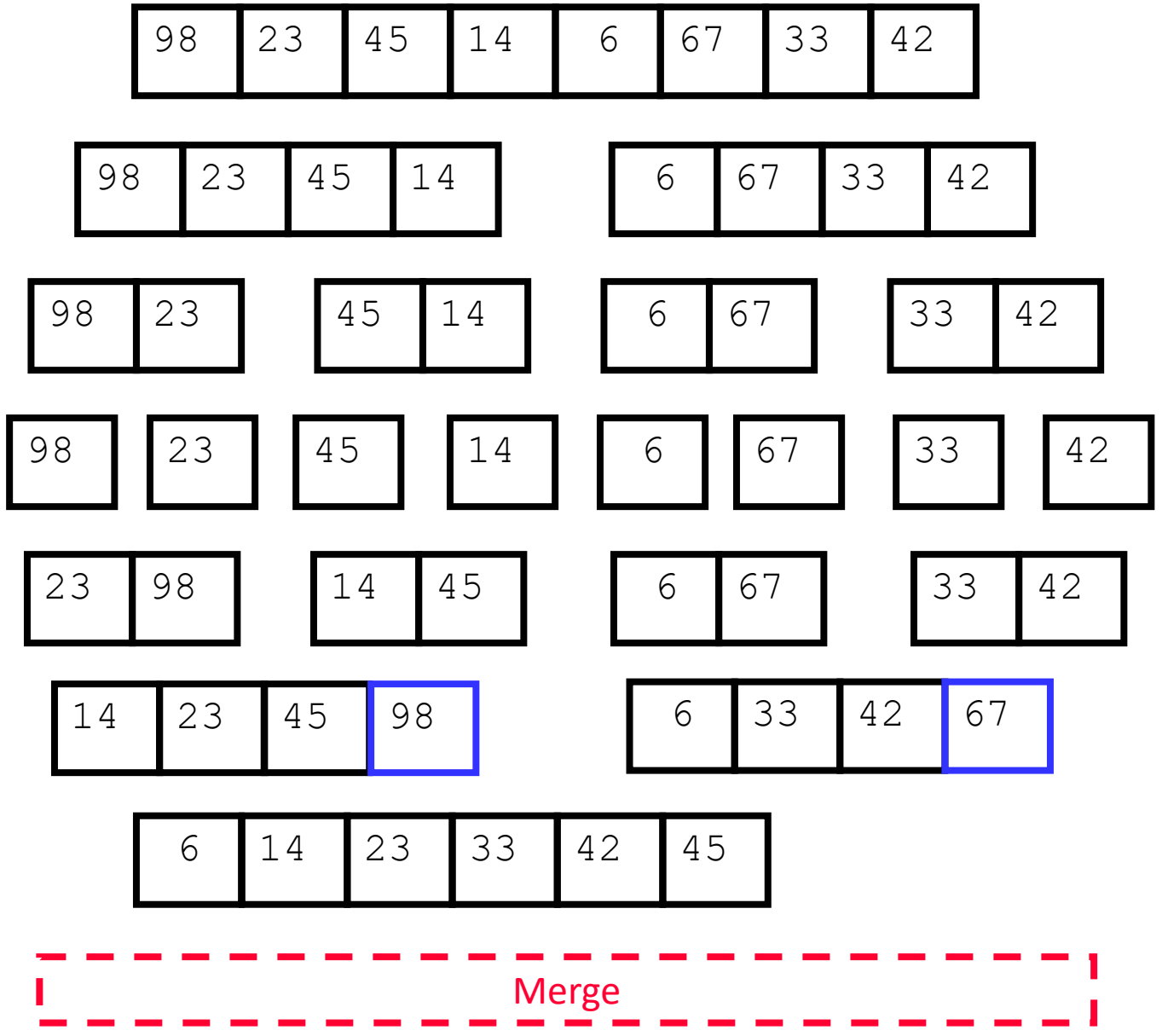
Another example



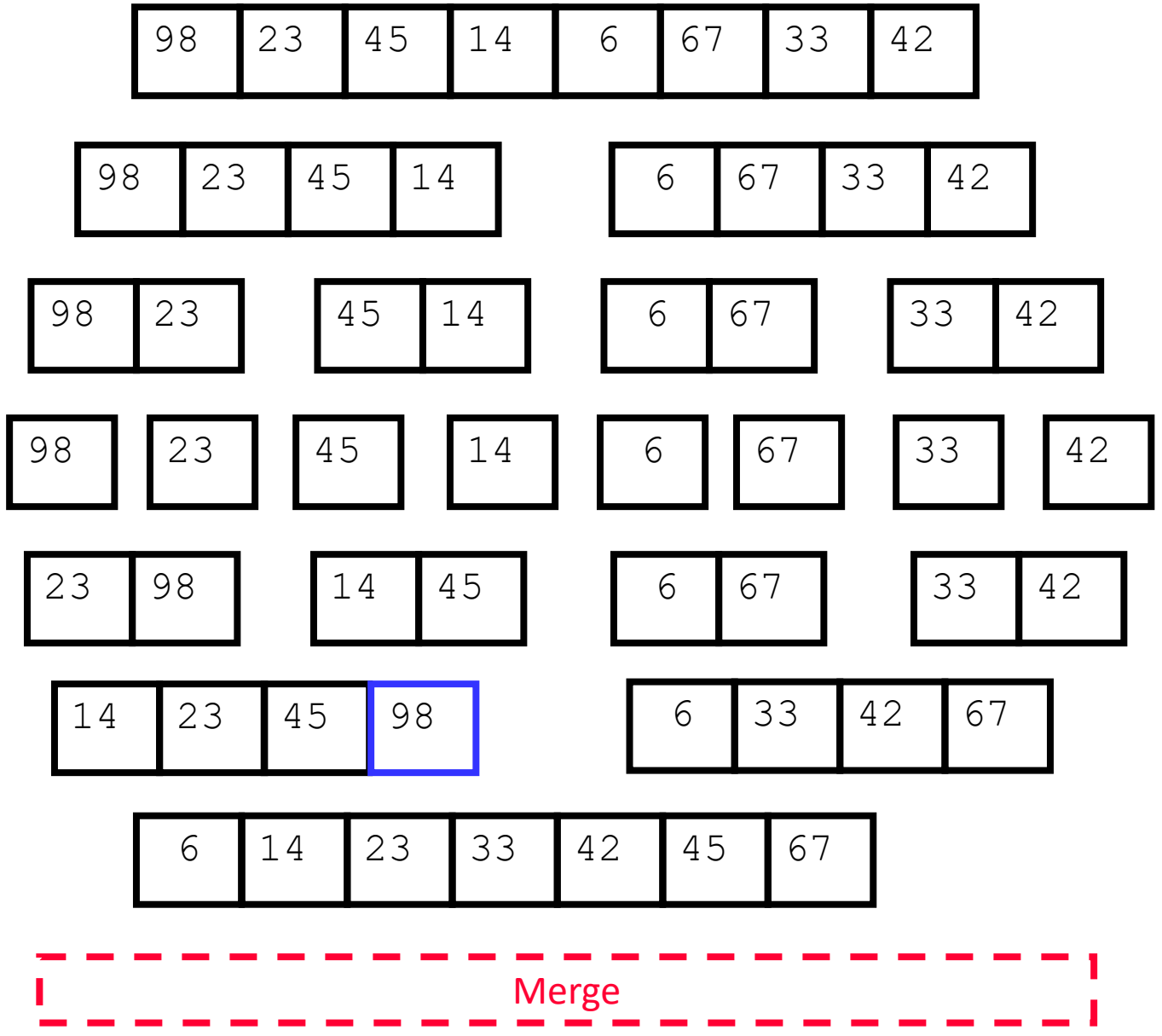
Another example



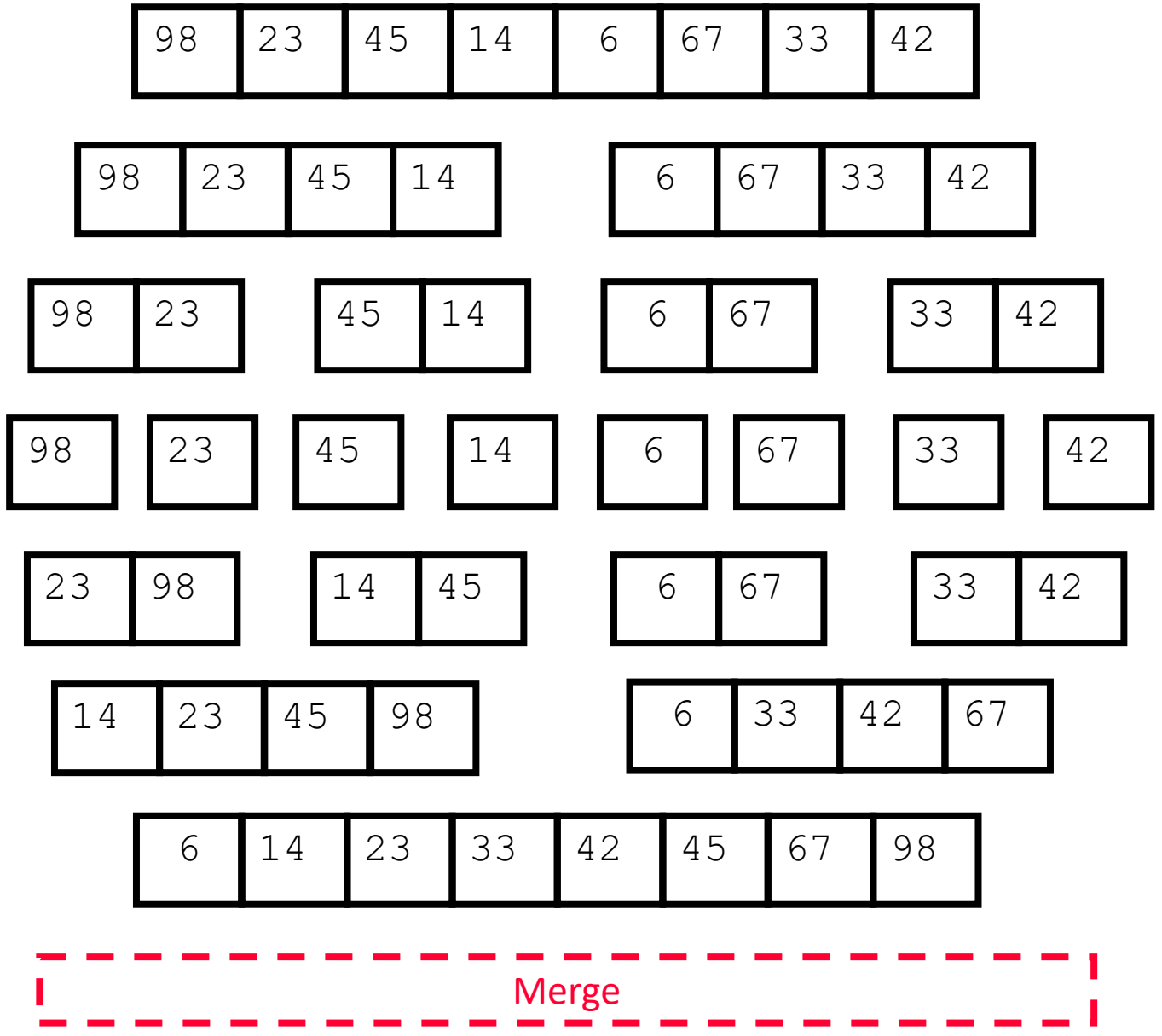
Another example



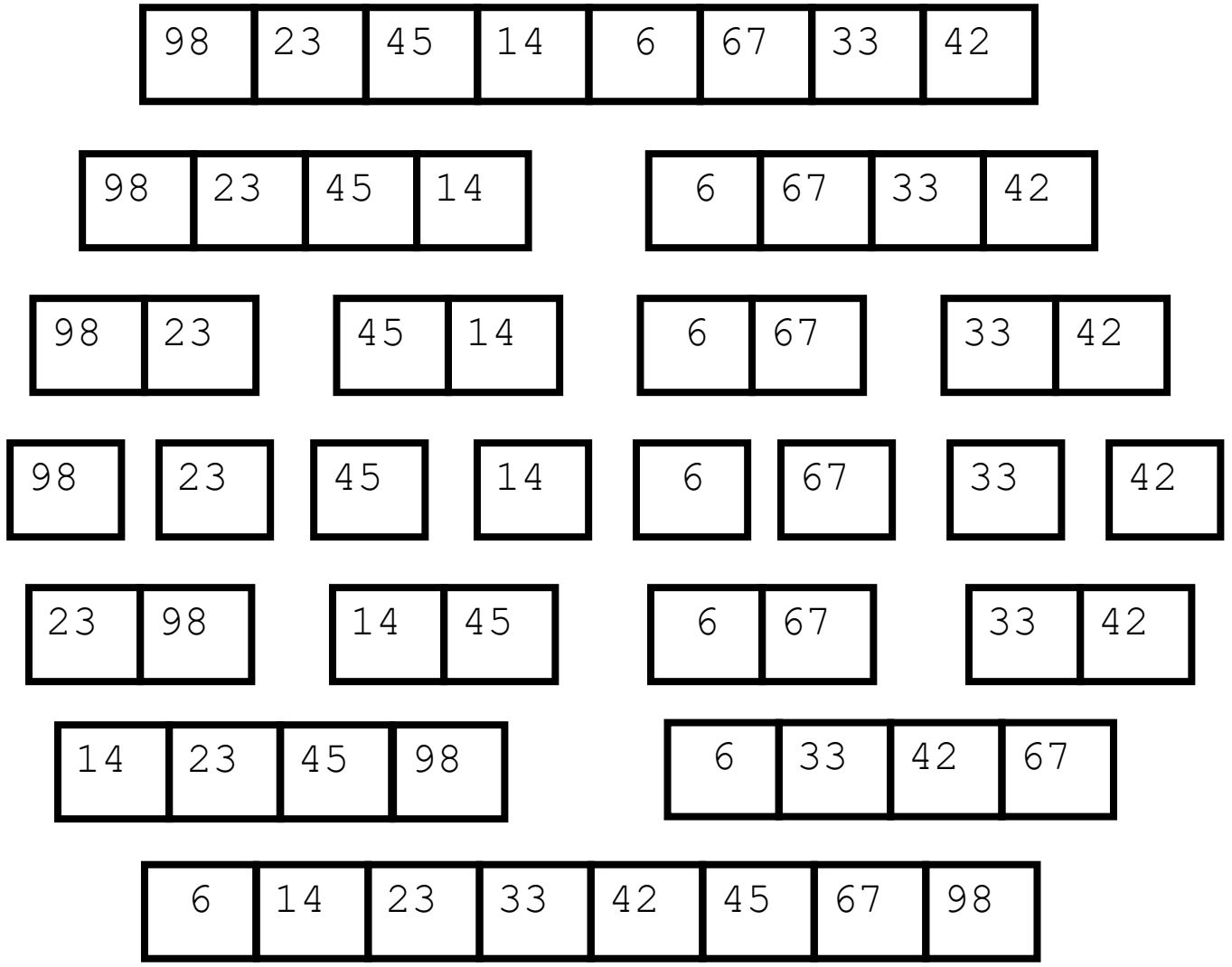
Another example



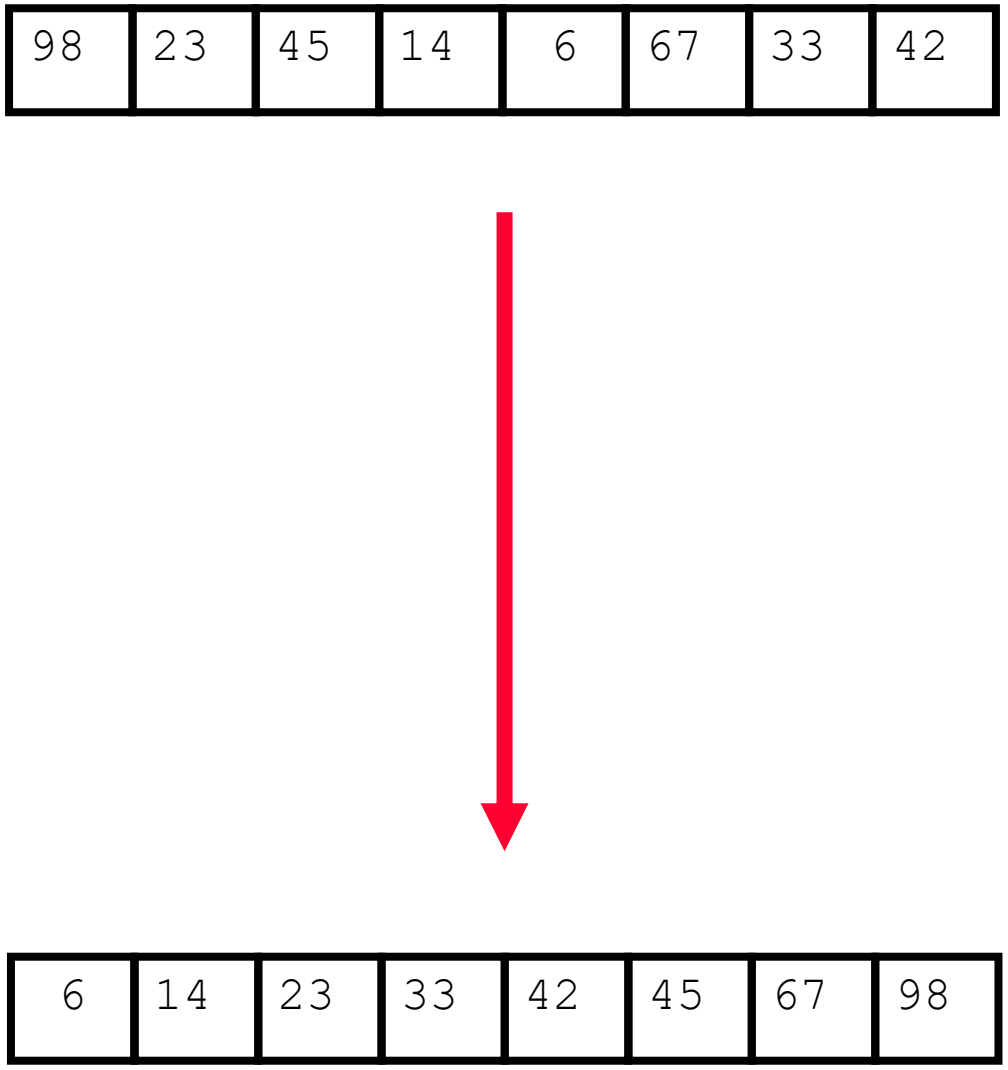
Another example



Another example



Another example



Merge Sort: Complexity analysis

Time Complexity:

$$T(n) = D(n) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + C(n) \qquad if\ n > 1$$

$$T(n) = c_1 \qquad if\ n = 1$$

- For simplicity of calculation

$$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) = T\left(\frac{n}{2}\right)$$

$$T(n) = c_1 \qquad if\ n = 1$$

$$T(n) = c_2 + 2T\left(\frac{n}{2}\right) + (n-1) \qquad if\ n > 1$$

$$T(n) = n \cdot c_1 + nlog_2 n + (c_2 - 1)(n-1) \qquad Assuming\ n = 2^k$$

Simple recurrence relation $T(n)=2T(n/2) + n$

Analysis of Merge Sort Time Complexity

- **Best Case Time Complexity of Merge Sort**
  - The best-case scenario occurs when the elements are already sorted in **ascending order**. If two sorted arrays of size n need to be merged, the minimum number of comparisons will be n. This happens when all elements of the first array are less than the elements of the second array. The best-case time complexity of merge sort is $O(n*logn)$.
- **Average Case Time Complexity of Merge Sort**
  - The average case scenario occurs when the elements are **jumbled** (neither in ascending nor descending order). This depends on the number of comparisons. The average case time complexity of merge sort is $O(n*logn)$.
- **Worst Case Time Complexity of Merge Sort**
  - The worst-case scenario occurs when the given array is sorted in **descending order** leading to the maximum number of comparisons. In this case, for two sorted arrays of size n, the minimum number of comparisons will be 2n. The worst-case time complexity of merge sort is $O(n*logn)$.

Quick Sort vs. Merge Sort

- **Quick sort**
  - **hard division, easy combination**
  - partition in the divide step of the divide-and-conquer framework
  - hence combine step does nothing
- **Merge sort**
  - **easy division, hard combination**
  - merge in the combine step
  - the divide step in this framework does one simple calculation only

Quick Sort vs. Merge Sort

Both the algorithms divide the problem into two sub problems.

- **Merge sort:**
  - two sub problems are of almost equal size always.
- **Quick sort:**
  - an equal sub division is not guaranteed.
- This difference between the two sorting methods appears as the deciding factor of their run time performances.

Any question?

Problems to Ponder…

Quick Sort vs Merge Sort

1. **Partition of elements in the array** : In the merge sort, the array is parted into just 2 halves (i.e. n/2). whereas In case of quick sort, the array is parted into any ratio. There is no compulsion of dividing the array of elements into equal parts in quick sort.
2. **Worst case complexity** : The worst case complexity of quick sort is O(n^2) as there is need of lot of comparisons in the worst condition. whereas In merge sort, worst case and average case has same complexities O(n log n).
3. **Usage with datasets** : Merge sort can work well on any type of data sets irrespective of its size (either large or small). whereas The quick sort cannot work well with large datasets.
4. **Additional storage space requirement** : Merge sort is not in place because it requires additional memory space to store the auxiliary arrays. whereas The quick sort is in place as it doesn't require any additional storage.
5. **Efficiency** : Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets. whereas Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.
6. **Sorting method** : The quick sort is internal sorting method where the data is sorted in main memory. whereas The merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.
7. **Stability** : Merge sort is stable as two elements with equal value appear in the same order in sorted output as they were in the input unsorted array. whereas Quick sort is unstable in this scenario. But it can be made stable using some changes in code.
8. **Preferred for** : Quick sort is preferred for arrays. whereas Merge sort is preferred for linked lists.
9. **Locality of reference** : Quicksort exhibits good cache locality, and this makes quicksort faster than merge sort (in many cases like in virtual memory environment).

If you try to solve problems yourself, then you will learn many things automatically.

Spend few minutes and then enjoy the study.