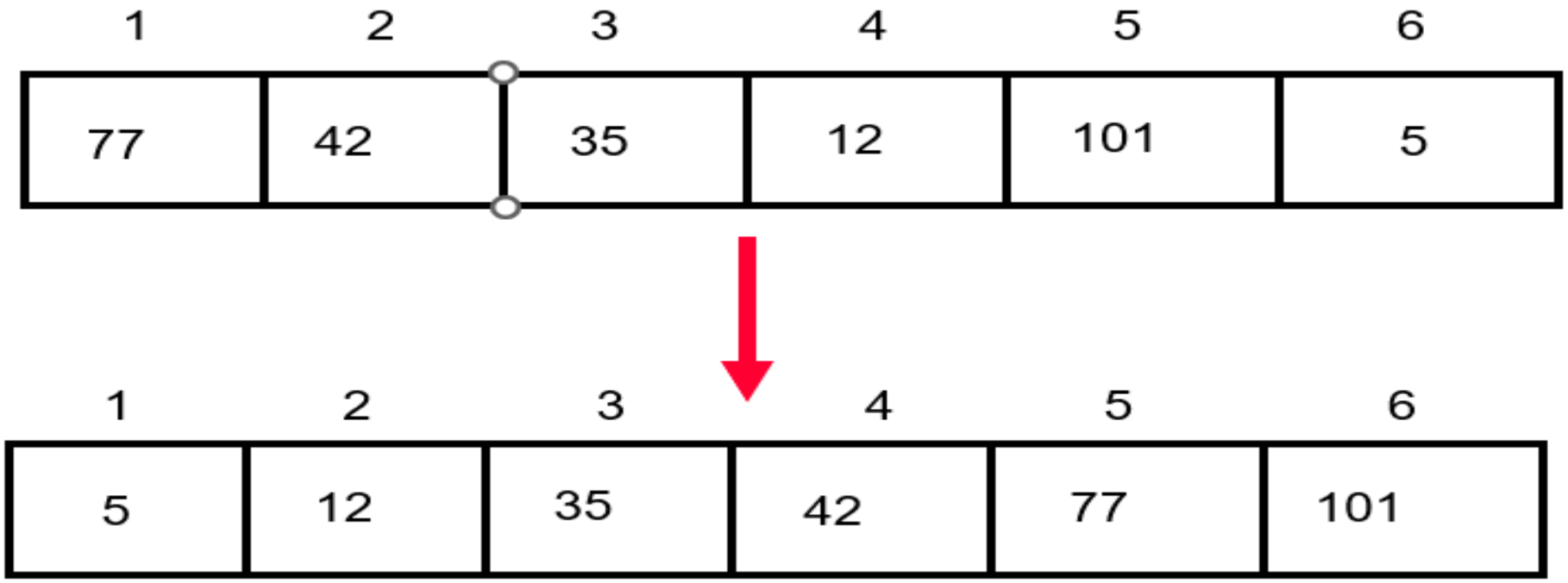
Data Structure and Algorithm chapter 07

Presented by: Er. Aruna Chhatkuli

Nepal College of Information Technology,
Balkumari, Lalitpur

Sorting

- Sorting is a process in which records are arranged in ascending or descending order.



Sorting

Sorting Categories

There are two different categories in sorting:

Internal sorting:

- If the input data is such that it can be adjusted in the main memory at once, it is called internal sorting.

External sorting:

- If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device. This is called external sorting.

Sorting

Types of Sort:

i. Insertion sort and selection sort
ii. Exchange sort
iii. Bubble and quick sort
iv. Merge and Radix sort
v. Shell sort
vi. Heap sort as priority Queue.

Selection sort

- The list is divided into two subsists, sorted and unsorted, which are divided by an imaginary wall.
- We find the smallest element from the unsorted sub list and swap it with the element at the beginning of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sub lists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time we move one element from the unsorted sub list to the sorted sub list, we say that we have completed a sort pass.
- A list of n elements requires n-1 passes to completely rearrange the data.

Selection sort

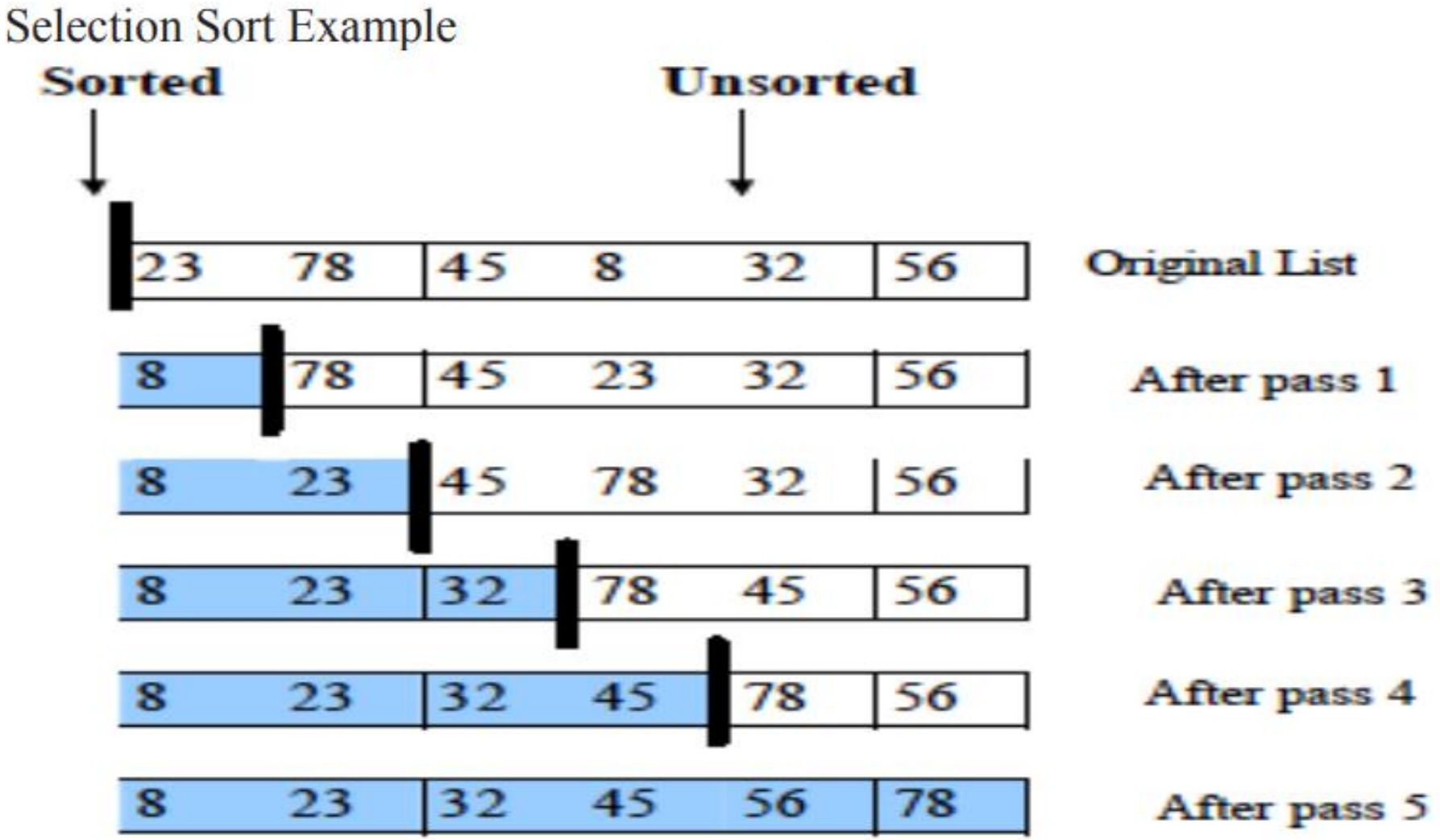
- **Selection sort** is a sorting algorithm which works as follows:

i. Find the minimum value in the list.

ii. Swap it with the value in the first position

iii. Repeat the steps above for remainder of the list (starting at the second position).

Selection sort



Selection sort

- **Algorithm Let A be a linear array of n numbers. tmp be a temporary variable for swapping (or interchanging). min is the variable to store the location of smallest number.**

i. **Input n numbers of an array A**

ii. **Initialize i = 0 and repeat through step5 if (i < n – 1)**

   a. **min = i**

iii. **Initialize j = i + 1 and repeat through step 4 if (j < n)**

iv. **if (min > a[j])**

   i. **min = j**

v. **i. tmp = a[min]**

   ii. a[min] = a[i]

   iii. a[i] = tmp

vi. **Display "the sorted numbers of array A"**

vii. **Exit**

Selection sort

Source Code:

```
void selectionsort(int arr[],int n){
        int i,j,temp;
        int min;
 for(i=0;i<n-1;i++){
        min = i;
 for(j=i+1;j<n;j++){
        if(arr[min] > arr[j])
        min = j;
 }
 temp= arr[min];
 arr[min]=arr[i];
 arr[i]=temp;
 }
}
```

Time Complexity of Selection Sort

The total number of comparisons is:

- (N-1)+(N-2)+(N-3)+…… +1= N(N-1)/2
- We can ignore the constant 1/2
- We can express N(N-1) as $N^2 – N$
- We can ignore N as well since N^2 grows more rapidly than N, making our algorithm $O(N^2)$

Insertion Sort

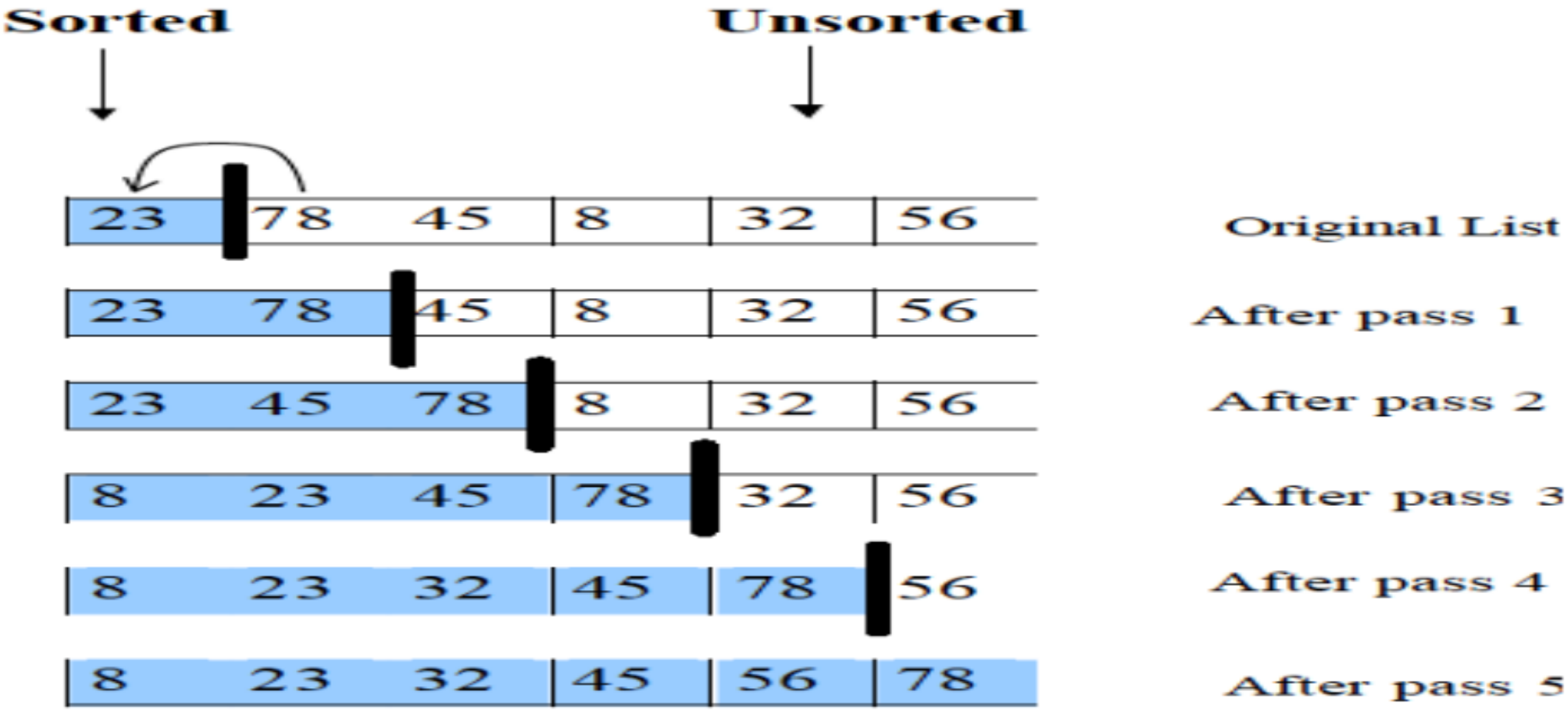
Insertion Sort

- In insertion sort, each successive element in the array to be sorted is inserted into its proper place with respect to the other, already sorted elements.
- We divide our array in a sorted and an unsorted array.
- Initially the sorted portion contains only one element: the first element in the array.
- We take the second element in the array, and put it into its correct place.

Insertion Sort

1. **Step 1:** As the single element A [1] by itself is sorted array.
2. **Step 2:** A [2] is inserted either before or after A [1] by comparing it so that A[1], A[2] is sorted array.
3. **Step 3:** A [3] is inserted into the proper place in A [1], A [2], that is A [3] will be compared with A [1] and A [2] and placed before A [1], between A [1] and A [2], or after A [2] so that A [1], A [2], A [3] is a sorted array.
4. **Step 4:** A [4] is inserted in to a proper place in A [1], A [2], A [3] by comparing it; so that A [1], A [2], A [3], A [4] is a sorted array.
5. **Step 5:** Repeat the process by inserting the element in the proper place in array.
6. **Step n :**A [n] is inserted into its proper place in an array A [1], A [2], A [3], ... A [n –1] so that A [1], A [2], A [3], ... ,A [n] is a sorted array.

Insertion Sort



Insertion Sort

99 | 55 4 66 28 31 36 52 38 72

55 99 | 4 66 28 31 36 52 38 72

4 55 99 | 66 28 31 36 52 38 72

4 55 66 99 | 28 31 36 52 38 72

4 28 55 66 99 | 31 36 52 38 72

4 28 31 55 66 99 | 36 52 38 72

4 28 31 36 55 66 99 | 52 38 72

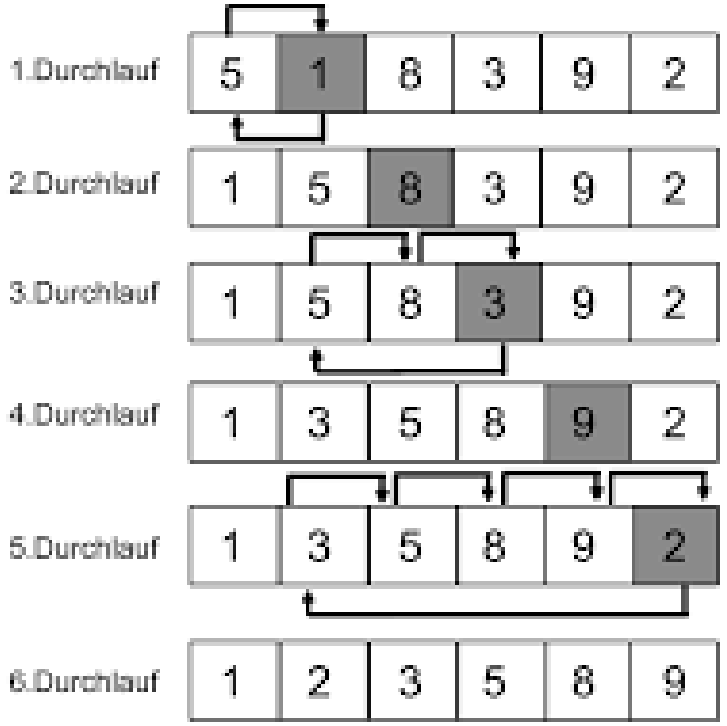
4 28 31 36 52 55 66 99 | 38 72

4 28 31 36 38 52 55 66 99 | 72

4 28 31 36 38 52 55 66 72 99 |

Insertion Sort

- That is, array[0] and array[1] are in order with respect to each other.
- Then the value in array[2] is put into its proper place, so array [0]…. array[2] is sorted and so on.



Insertion Sort

Let A be a linear array of n numbers A [1], A [2], A [3], ... ,A [n].....Swap be a temporary

variable to interchange the two values. Pos is the control variable to hold the position of each

pass.

1. Input an array A of n numbers.
2. Initialize i = 1 and repeat through steps 4 by incrementing i by one.
   i. If (i < = n – 1)
   ii. tmp = A [i],
   iii. Pos = i – 1
3. Repeat the step 3 if (tmp < A[Pos] and (Pos >= 0))
   i. A [Pos+1] = A [Pos]
   ii. Pos = Pos-1
4. A [Pos +1] = tmp
5. Exit

Insertion Sort

```
void insertion_Sort(int array[], int length)
{
  int i, j, value;
  for(i = 1; i < length; i++)
        {
      value = a[i];
      for (j = i - 1; j >= 0 && a[ j ] > value; j--)
            {
         a[j + 1] = a[ j ];
      }
      a[j + 1] = value;
   }
}
```

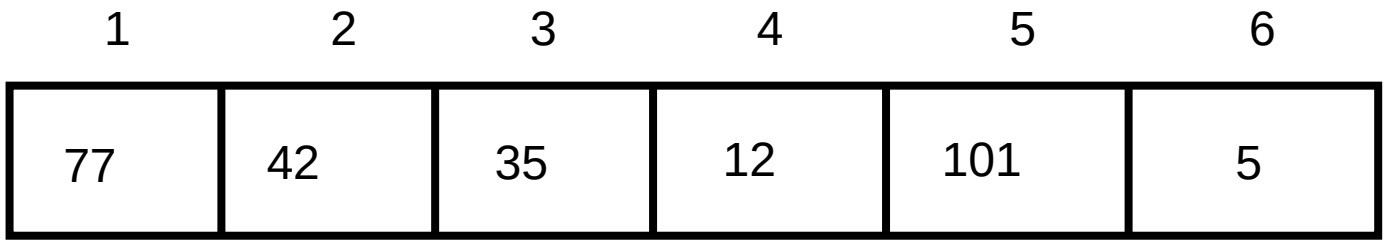
Bubble Sort

- Bubble sort is similar to selection sort in the sense that it repeatedly finds the largest/smallest value in the unprocessed portion of the array and puts it back.
- However, finding the largest value is not done by selection this time.
- We "bubble" up the largest value instead.

- Compares adjacent items and exchanges them if they are out of order.
- Comprises of several passes.
- In one pass, the largest value has been “bubbled” to its proper position.
- In second pass, the last value does not need to be compared.

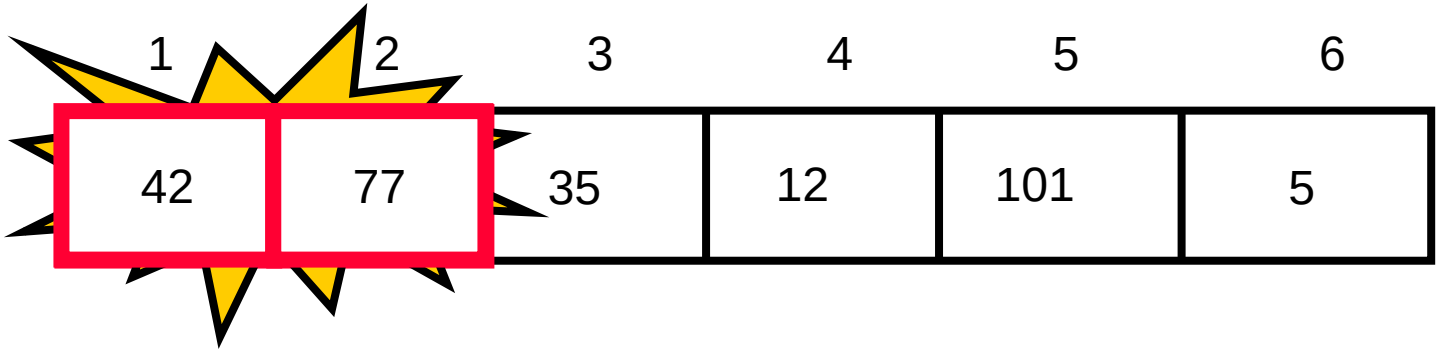
Bubble Sort

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



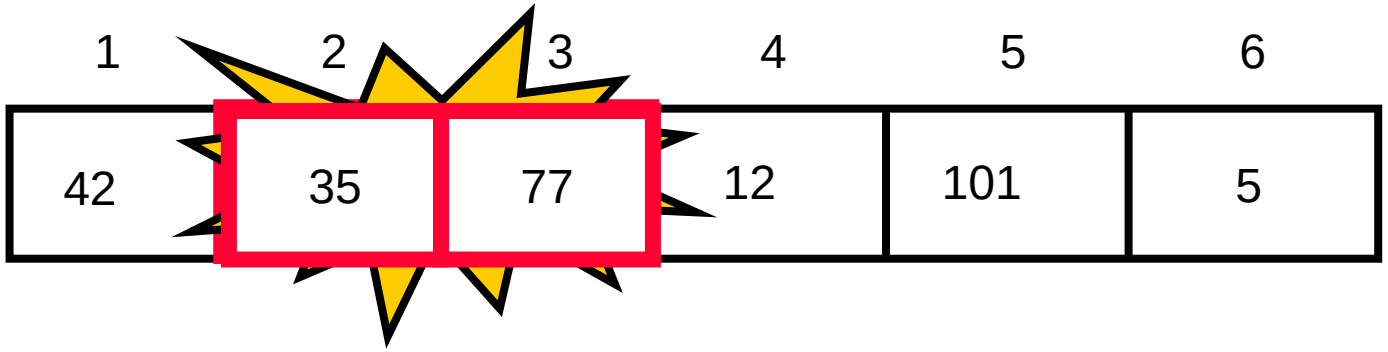
Bubble Sort

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



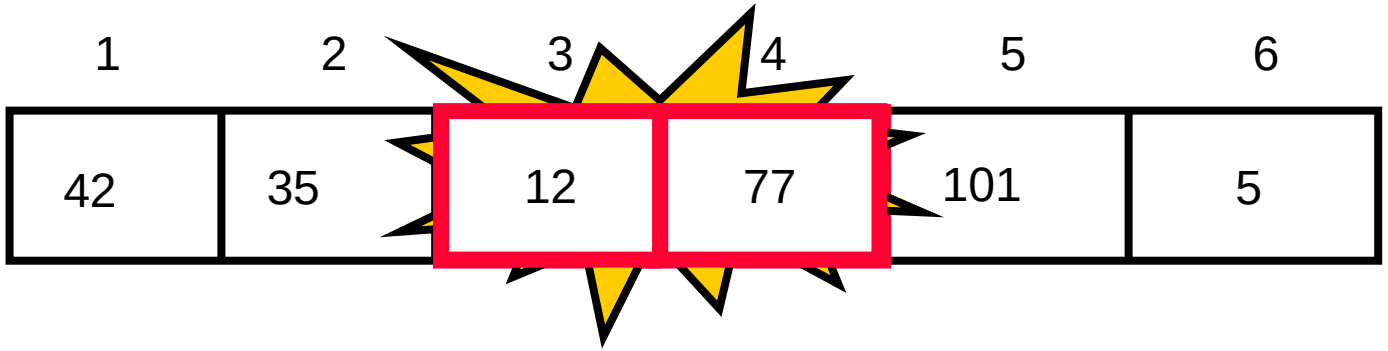
Bubble Sort

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



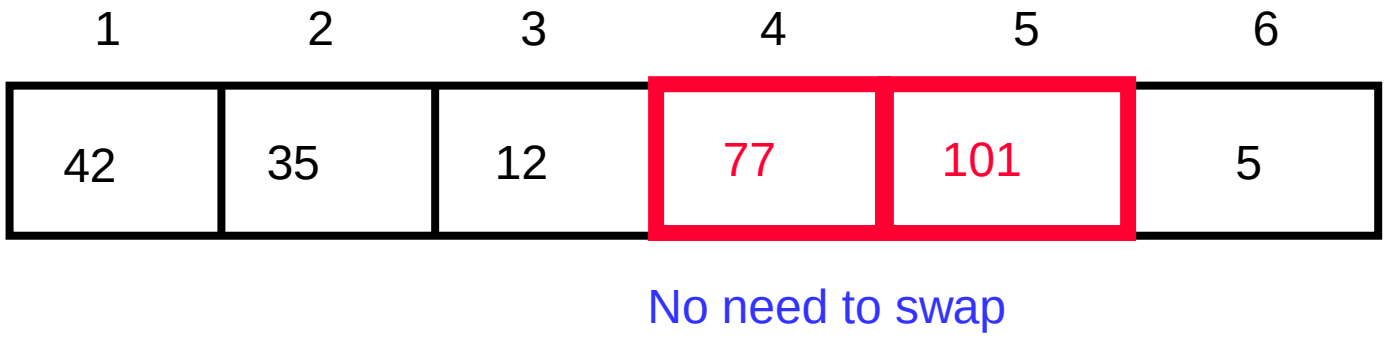
Bubble Sort

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



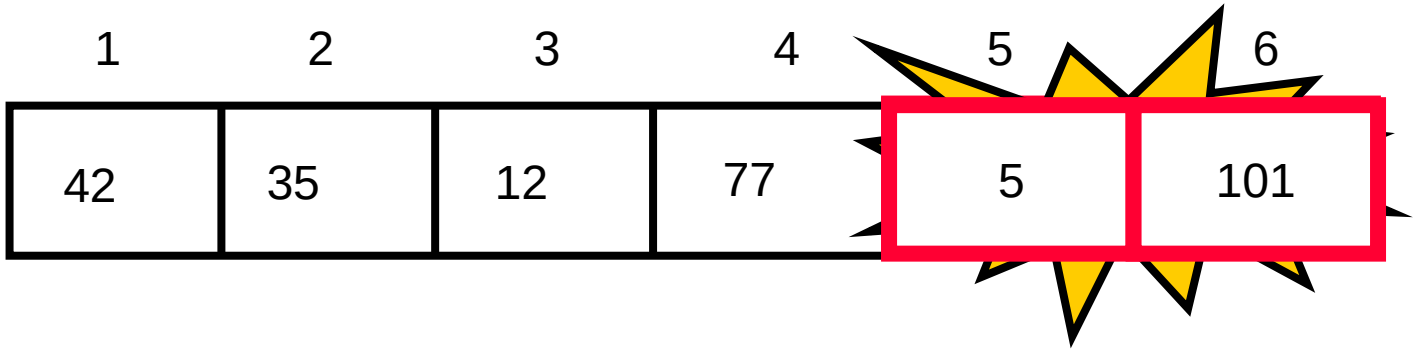
Bubble Sort

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



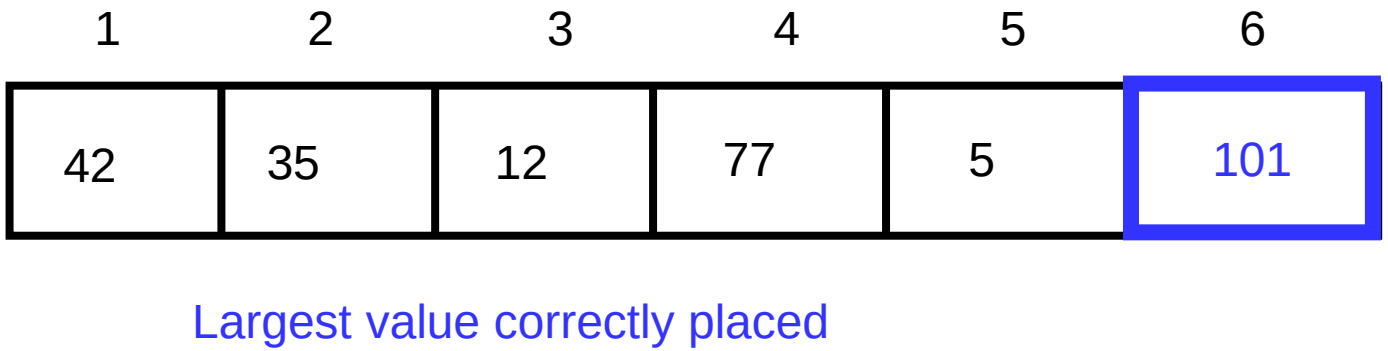
Bubble Sort

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

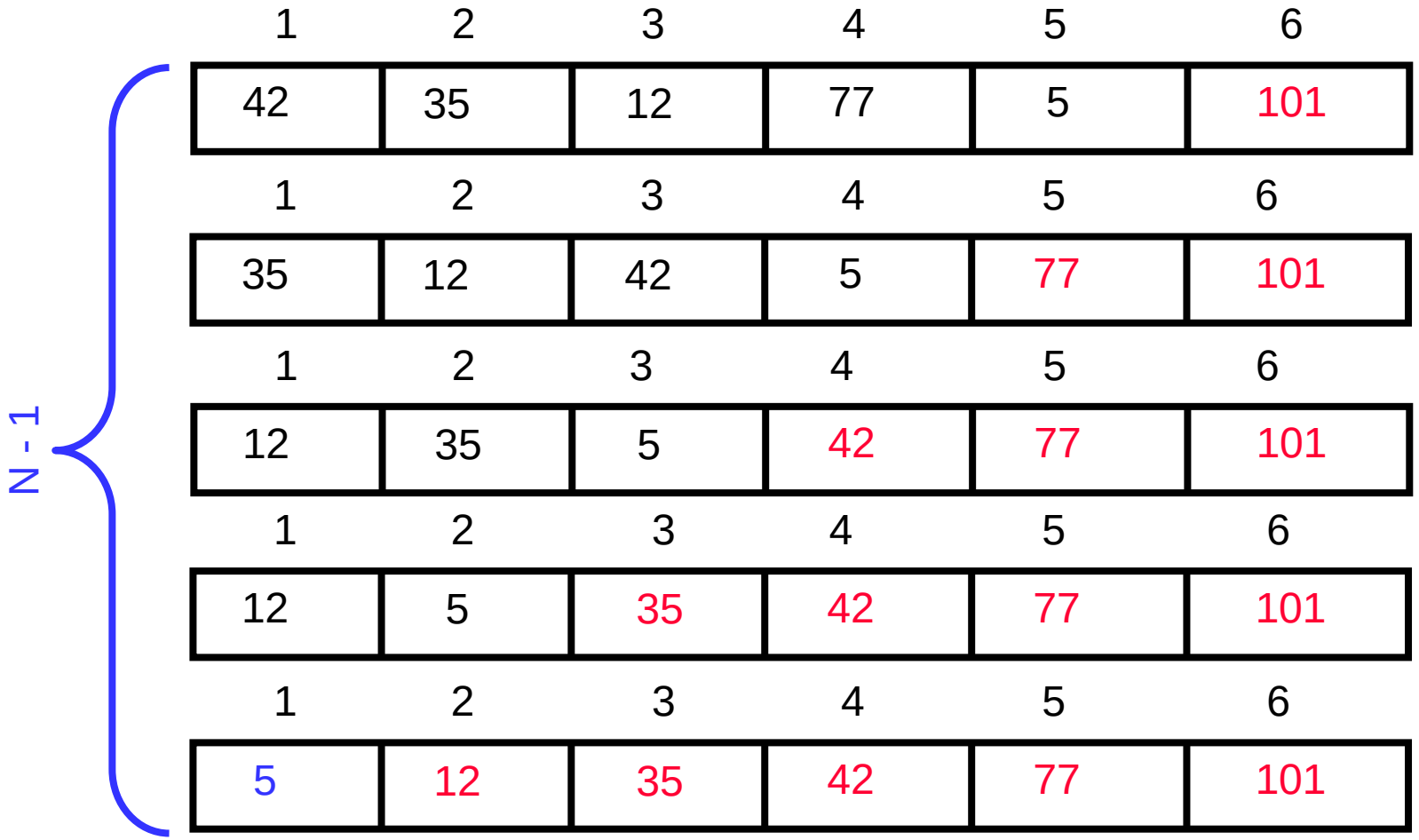


Bubble Sort

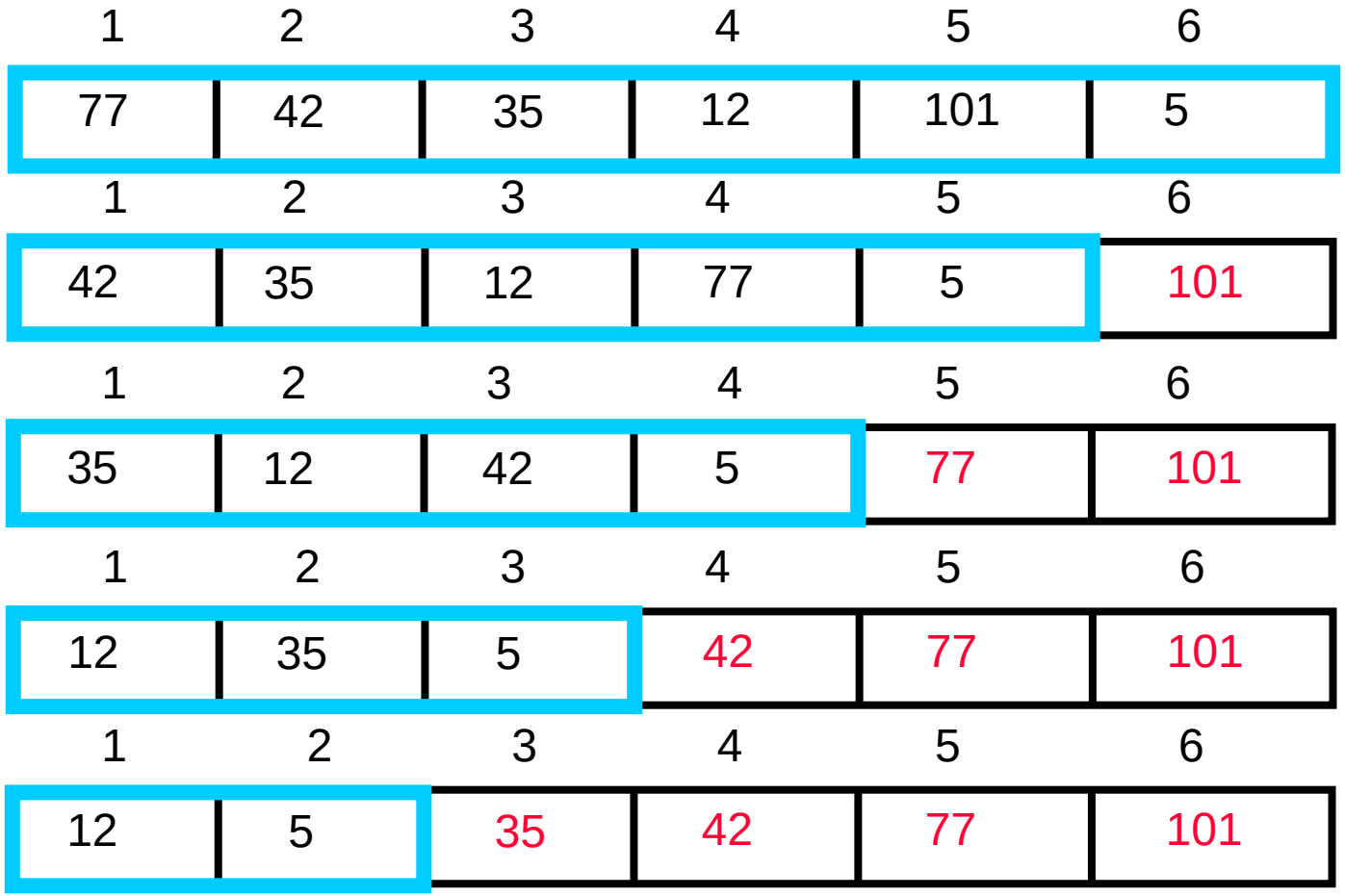
- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



“Bubbling” All the Elements



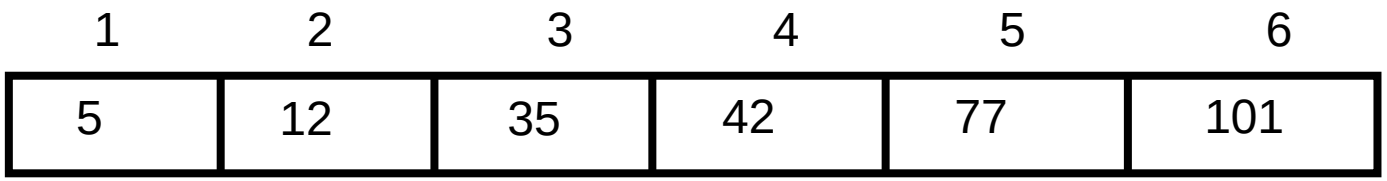
Reducing the Number of Comparisons



Already Sorted Collections?

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

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



Bubble sort example:

The elements of an array A to be sorted are: 42, 33, 23, 74, 44

First Pass

33 swapped	33	33	33
42	23 swapped	23	23
23	42	42 no swapping	42
74	74	74	44 swapped
44	44	44	74

Second Pass

23 swapped	23	23
33	33 no swapping	33
42	42	42 no swapping
44	44	44
74	74	74

Third Pass

23 no swapping	23
33	33 no swapping
42	42
44	44
74	74

Fourth Pass

23 no swapping
33
42
44
74

Thus the sorted array is 23, 33, 42, 44, 74.

Algorithm for bubble sort

Algorithm Let A be a linear array of n numbers. tmp is a temporary variable for swapping (or interchange) the position of the numbers.

1. Input n numbers of an array A
2. Initialize i = 0 and repeat through step 4 if (i < n)
3. Initialize j = 0 and repeat through step 4 if (j < n – i – 1)
4. If (A[j] > A[j + 1])
   i. tmp = A[j]
   ii. A[j] = A[j + 1]
   iii. A[j + 1] = tmp
5. Display the sorted numbers of array A
6. Exit.

Bubble Sort Algorithm

```
/* Bubble sort code */
#include <stdio.h>
int main(){
  int array[100], n, c, d, swap;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 0 ; c < n - 1; c++) {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (array[d] > array[d+1]) /* For
     decreasing order use '<' instead of '>' */
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      )}}
  printf("Sorted list in ascending
    order:\n");
  for (c = 0; c < n; c++)
    printf("%d\n", array[c]);
  return 0;
}
```

Summary

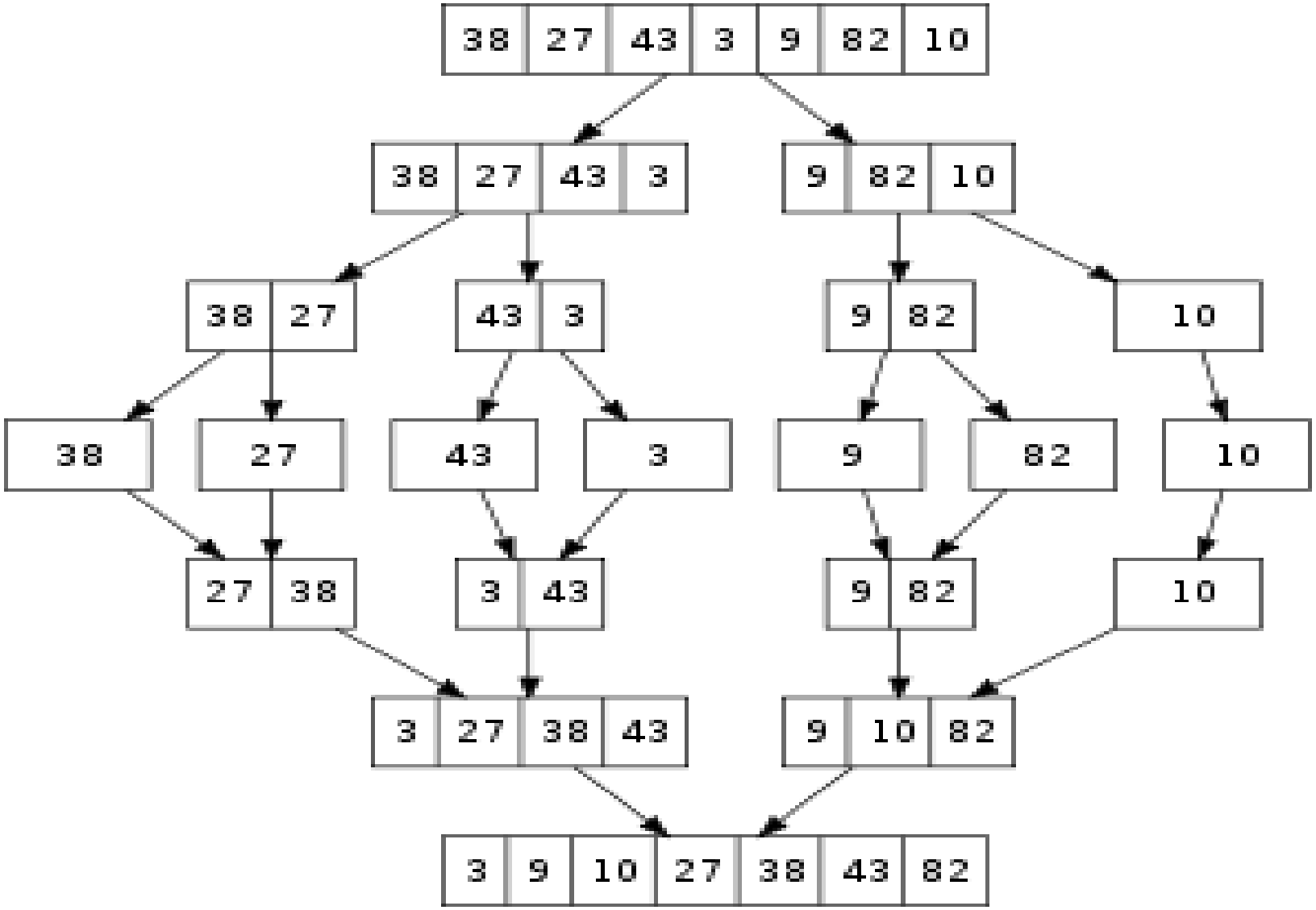
- The insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less.
- The insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort.
- The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items.

Merge Sort

Divide and Conquer cuts the problem in half each time, but uses the result of both halves:

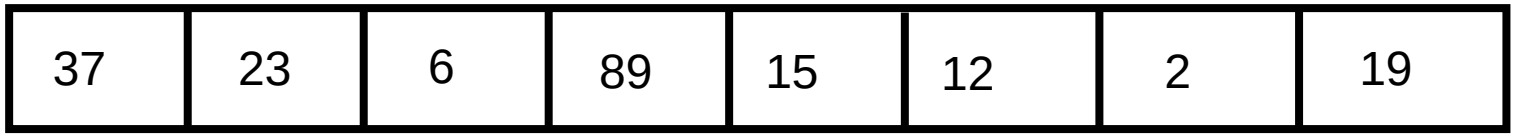
1. Cut the problem in half until the problem is trivial
2. Solve for both halves
3. Combine the solutions

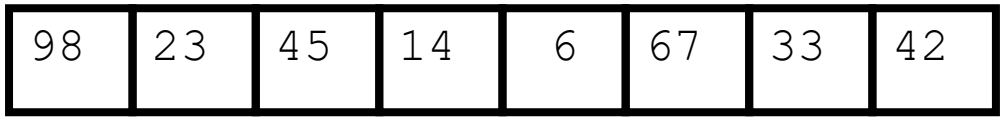
Merge Sort

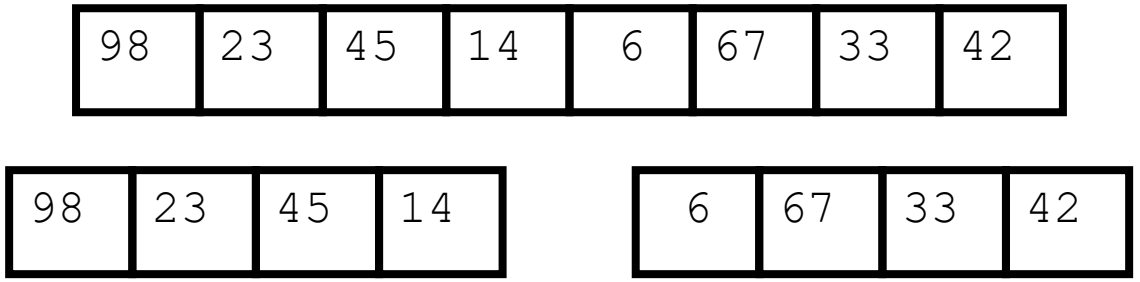

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

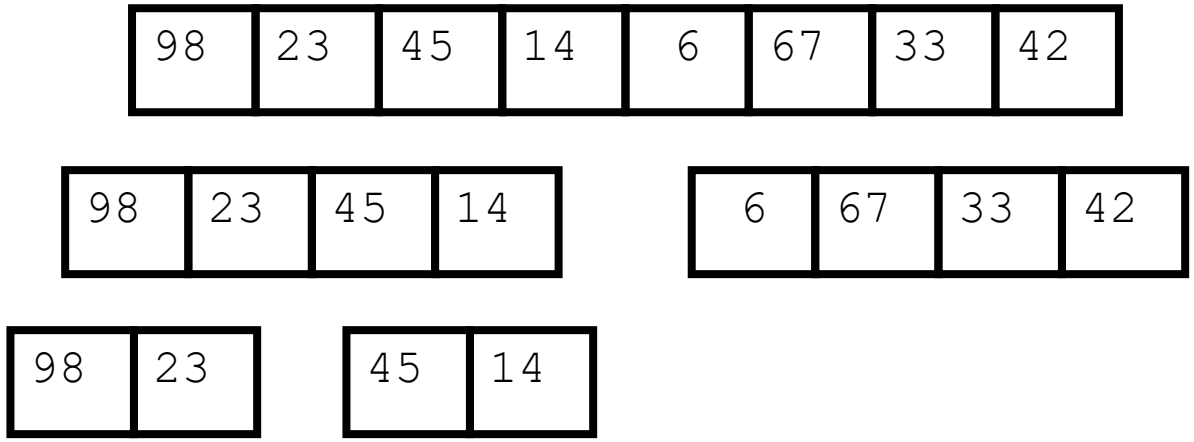
Mergesort

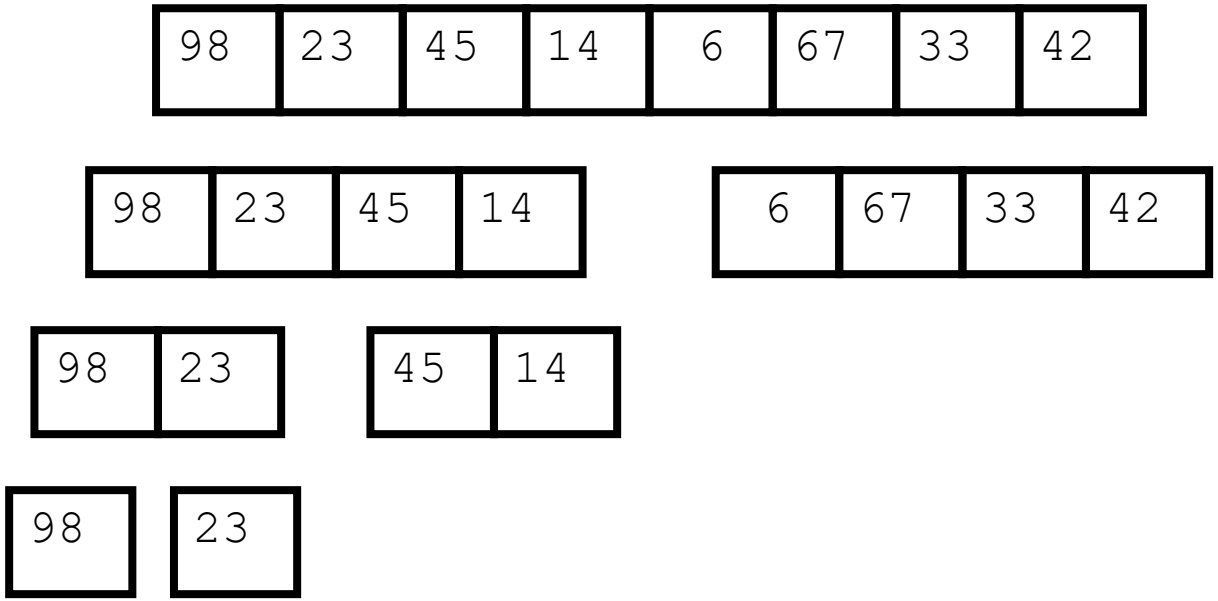
- **A divide-and-conquer algorithm:**
- **Divide the unsorted array into 2 halves until the sub-arrays only contain one element**
- **Merge the sub-problem solutions together:**
  - **Compare the sub-array's first elements**
  - **Remove the smallest element and put it into the result array**
  - **Continue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

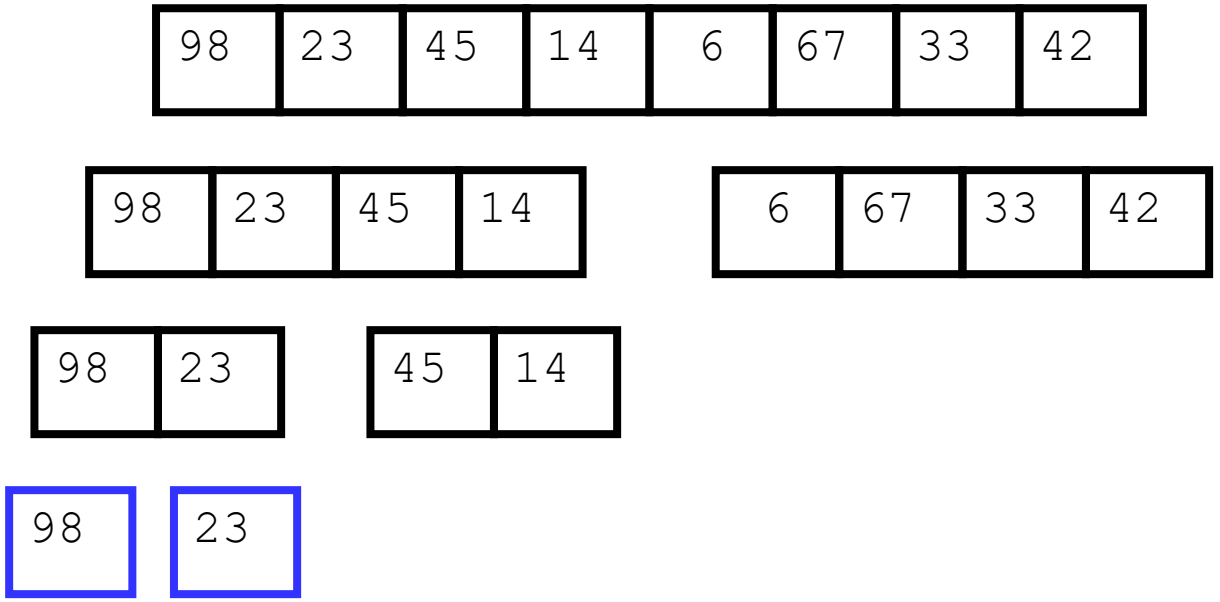


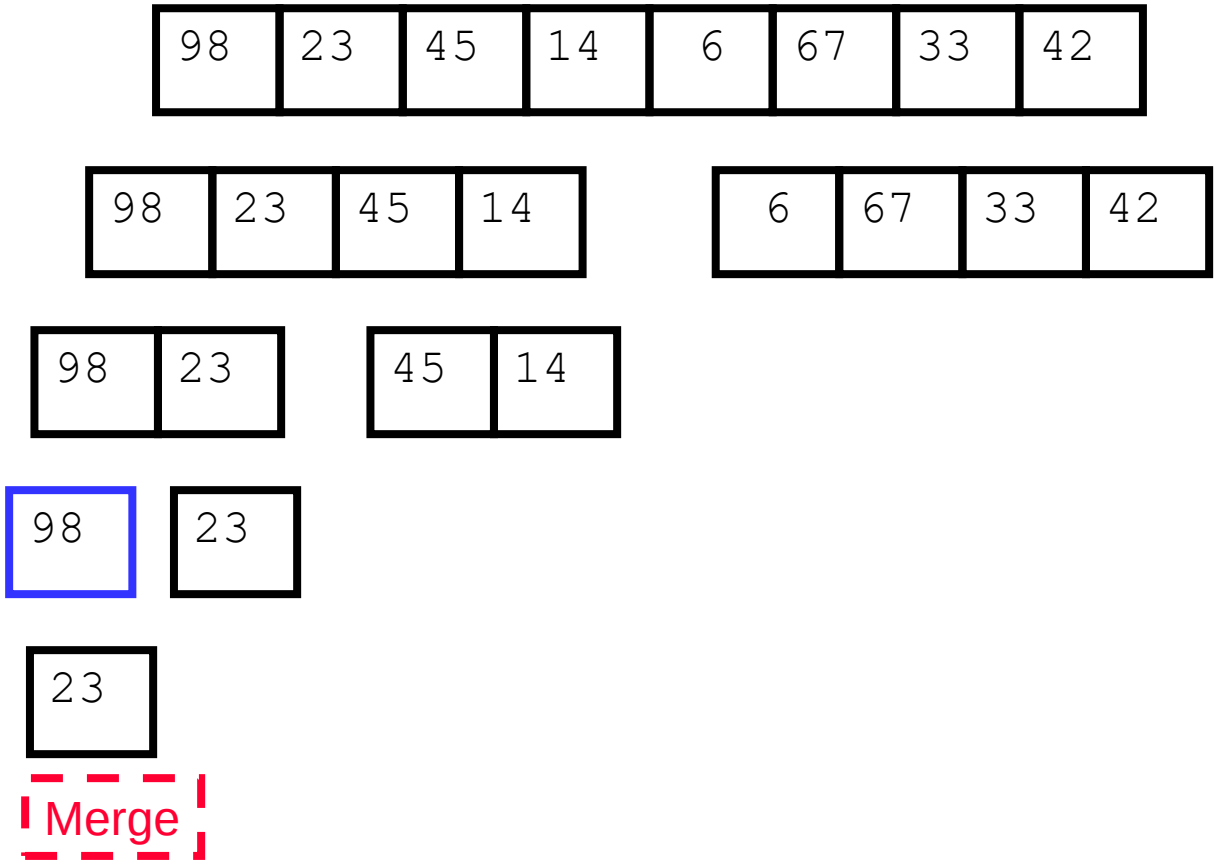

98
23
45
14
6
67
33
42

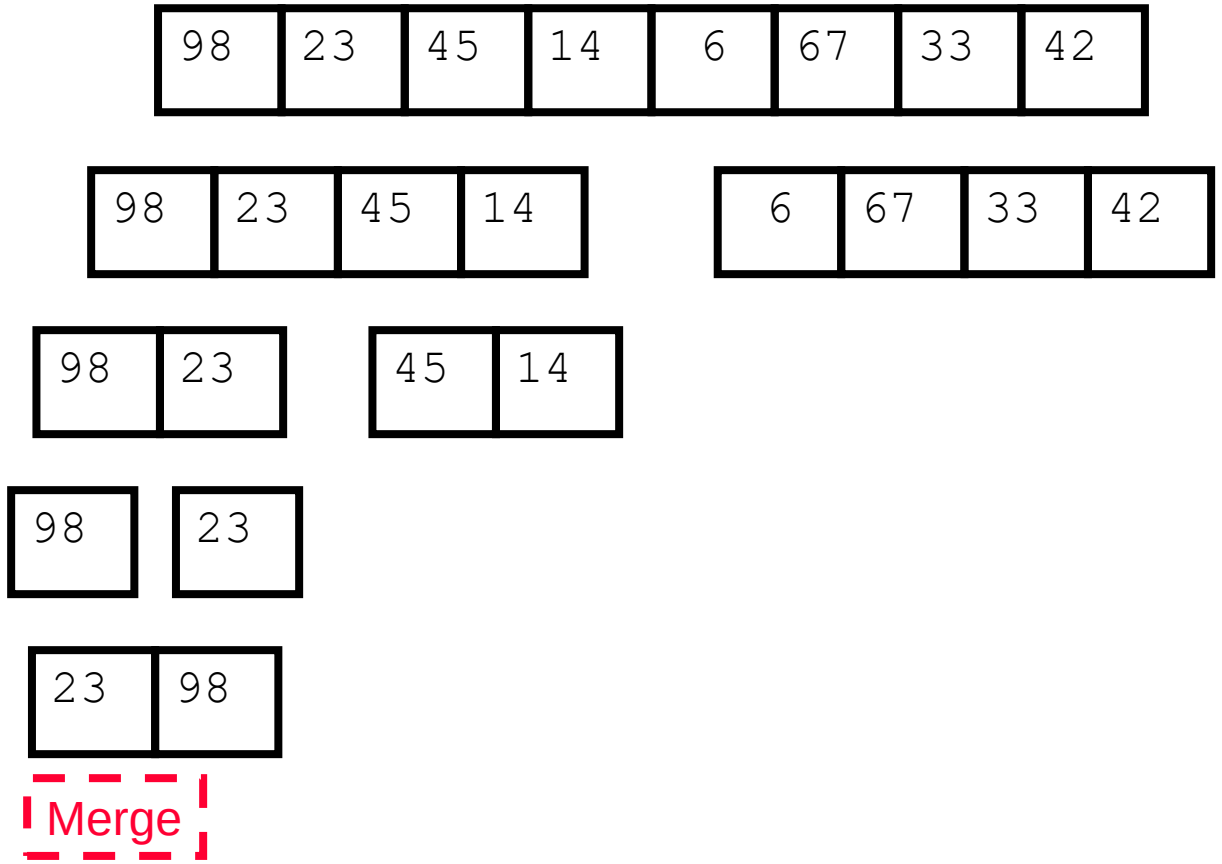

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42

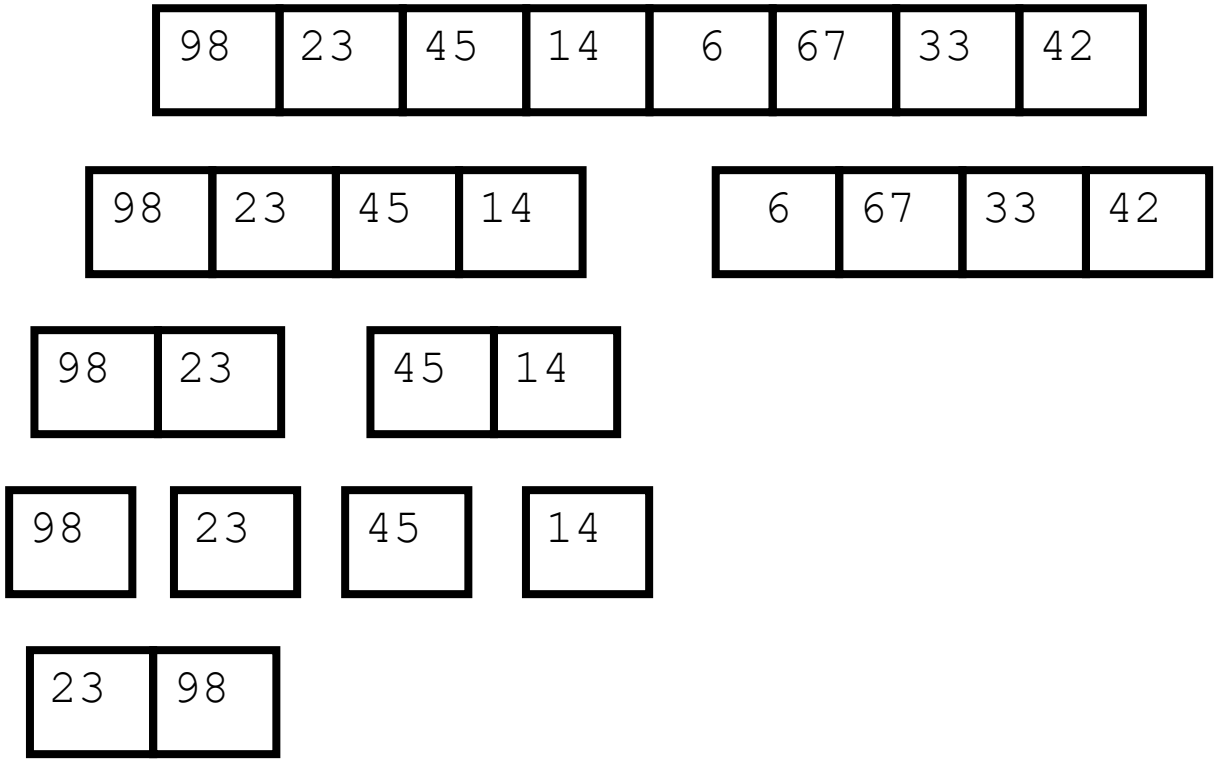

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

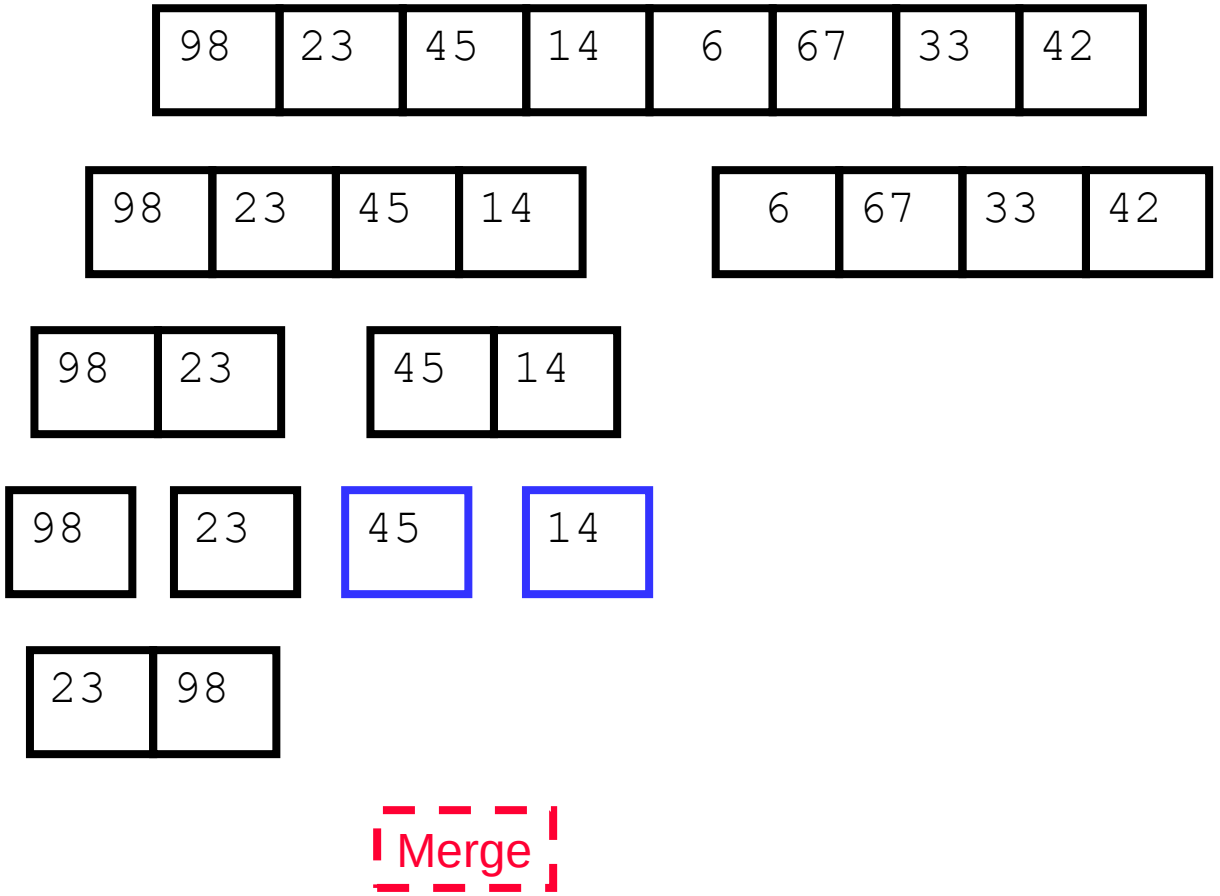

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

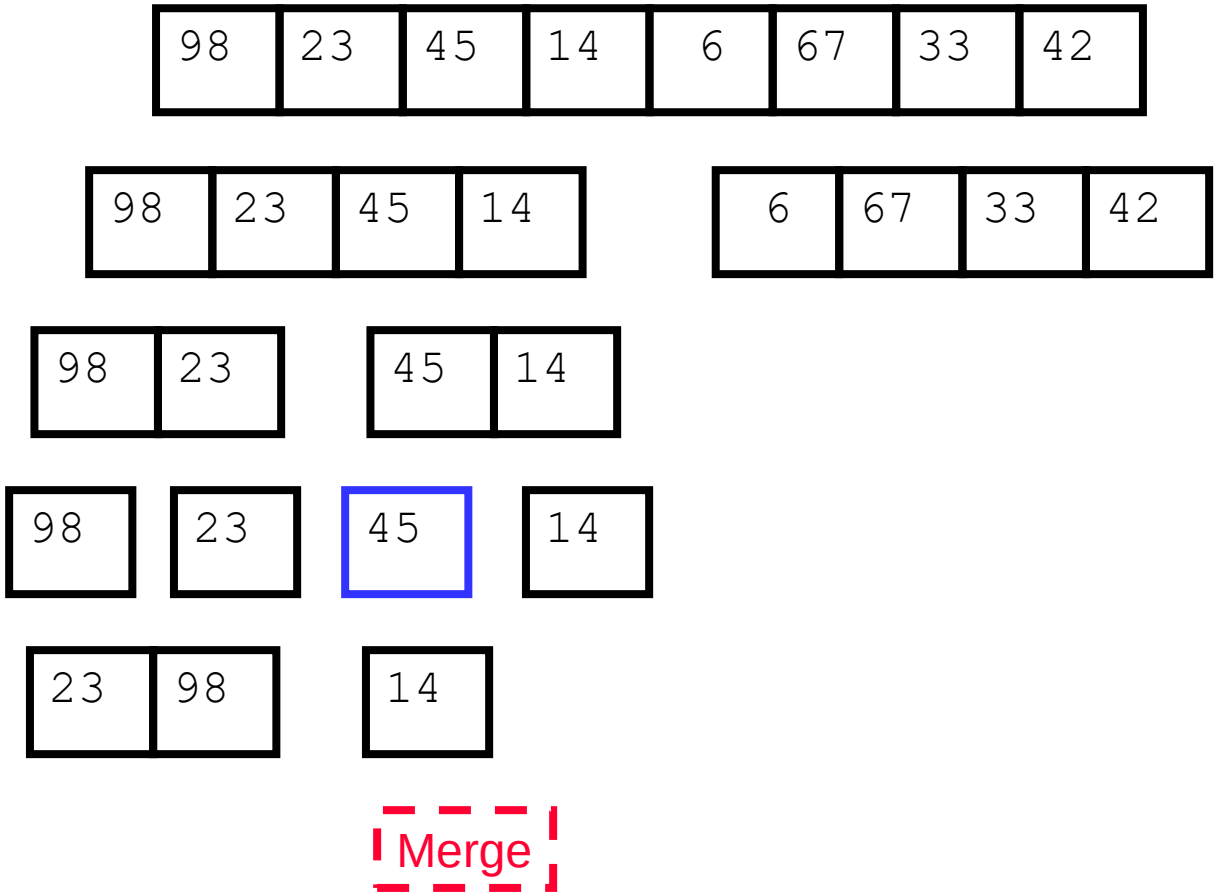

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

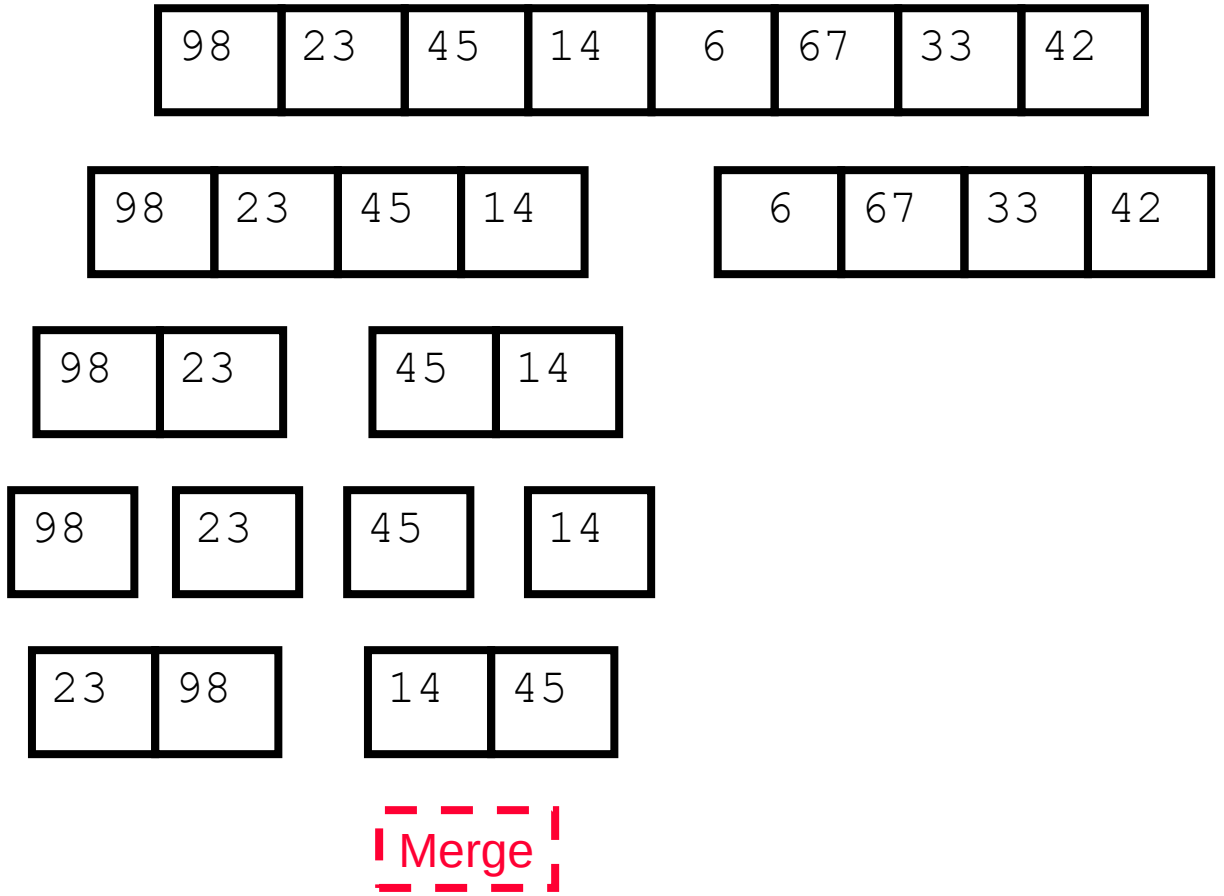

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

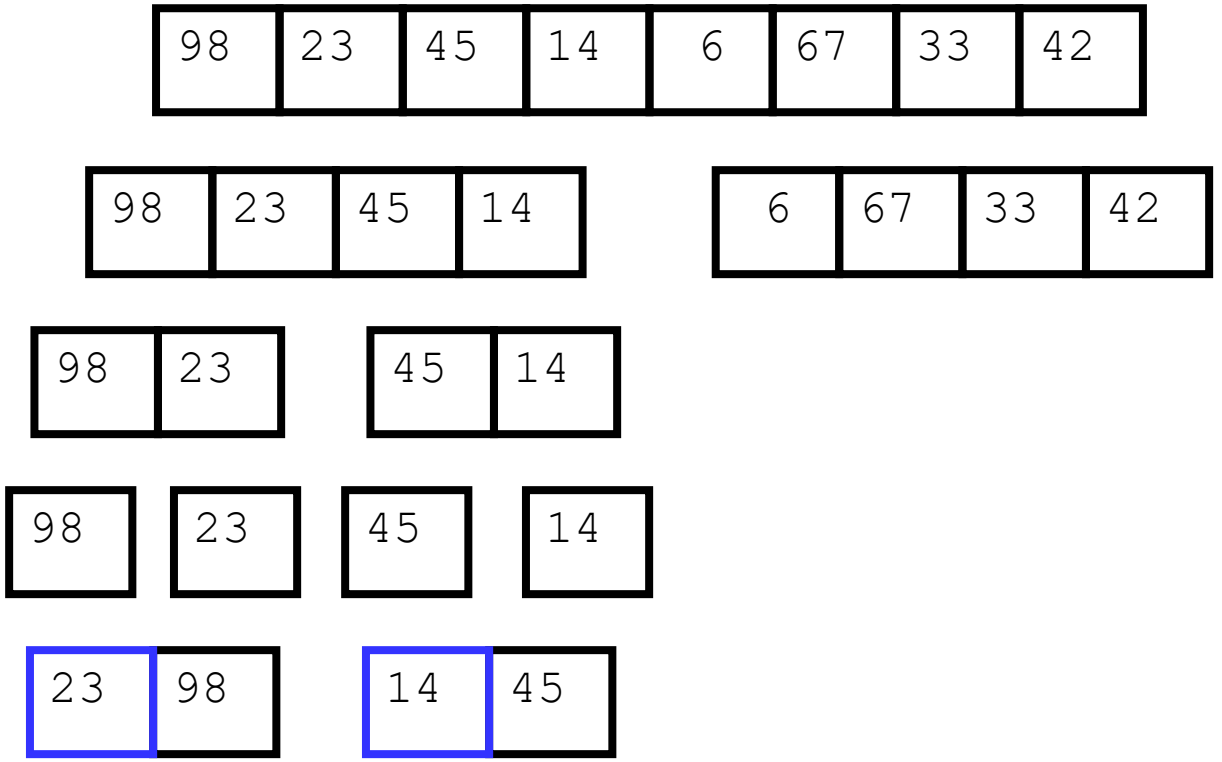

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge

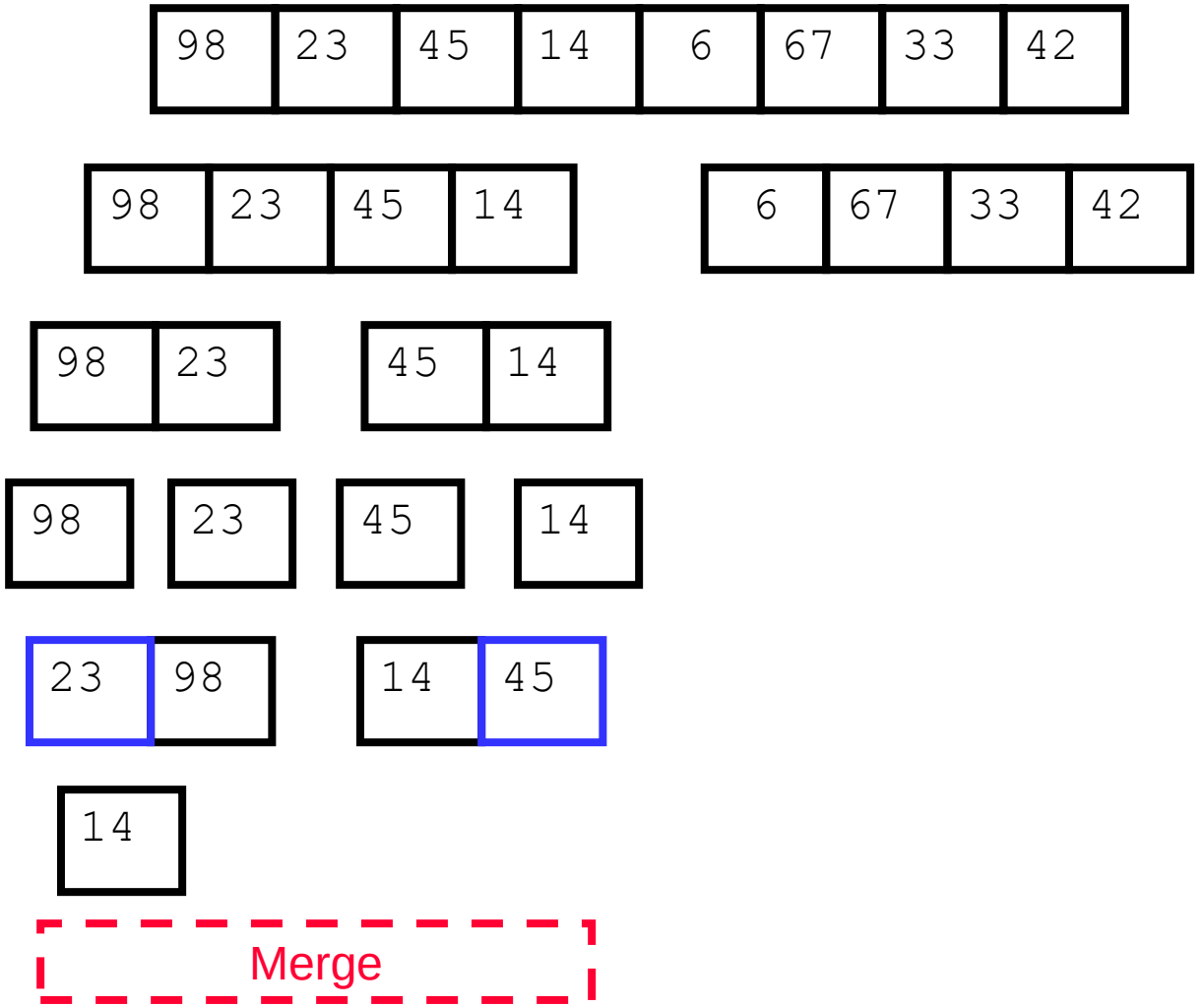

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98

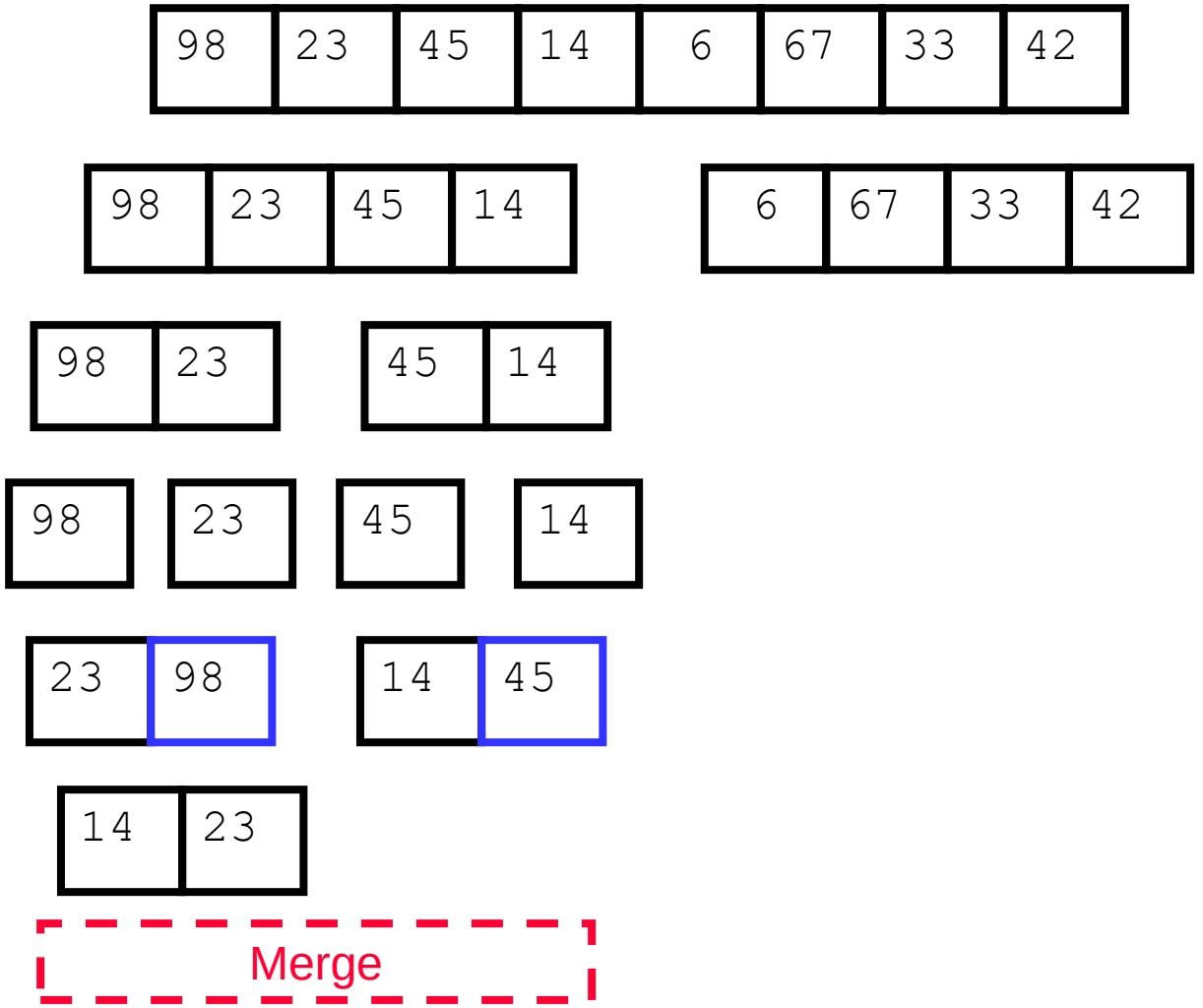

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
Merge

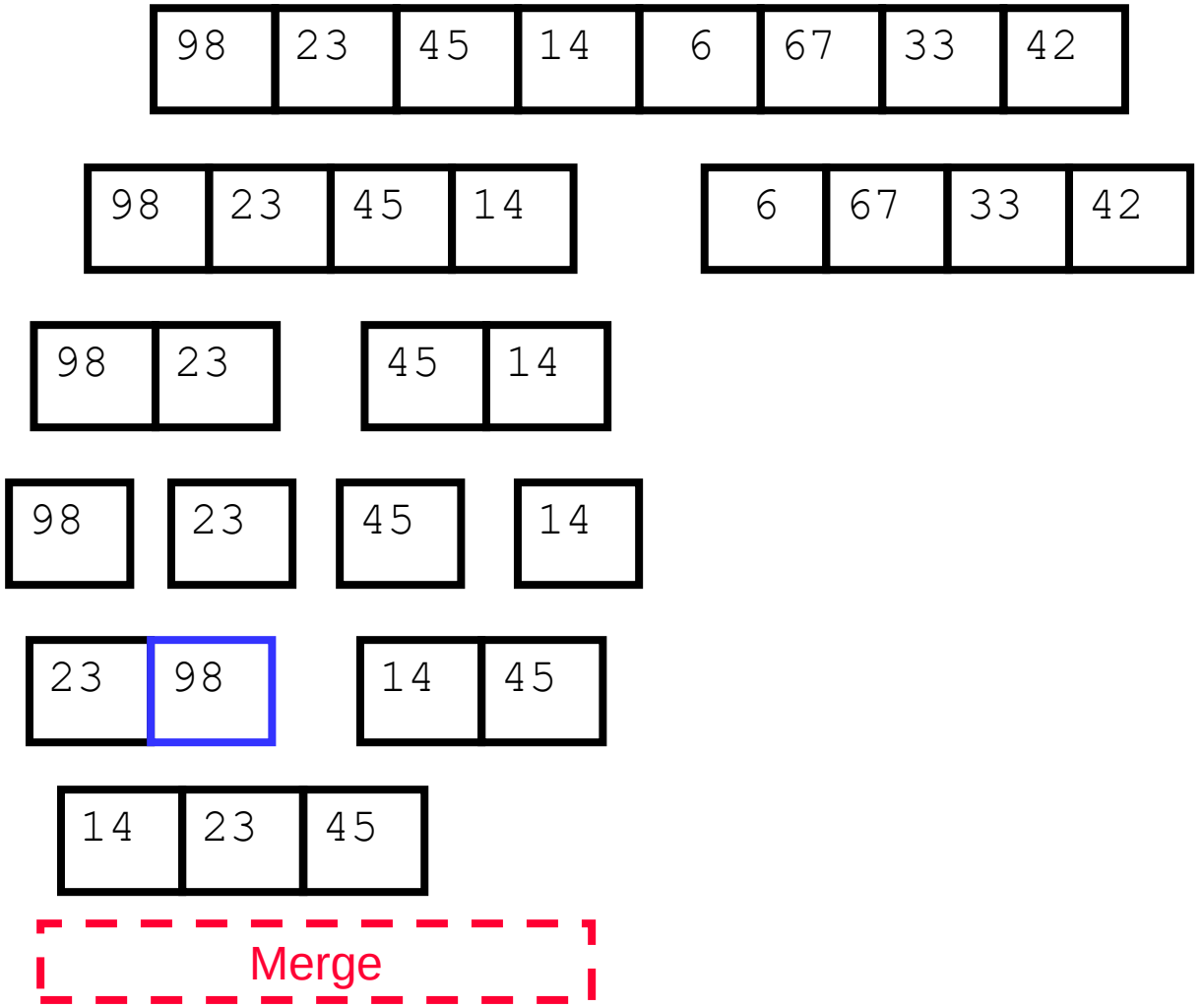

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14
Merge

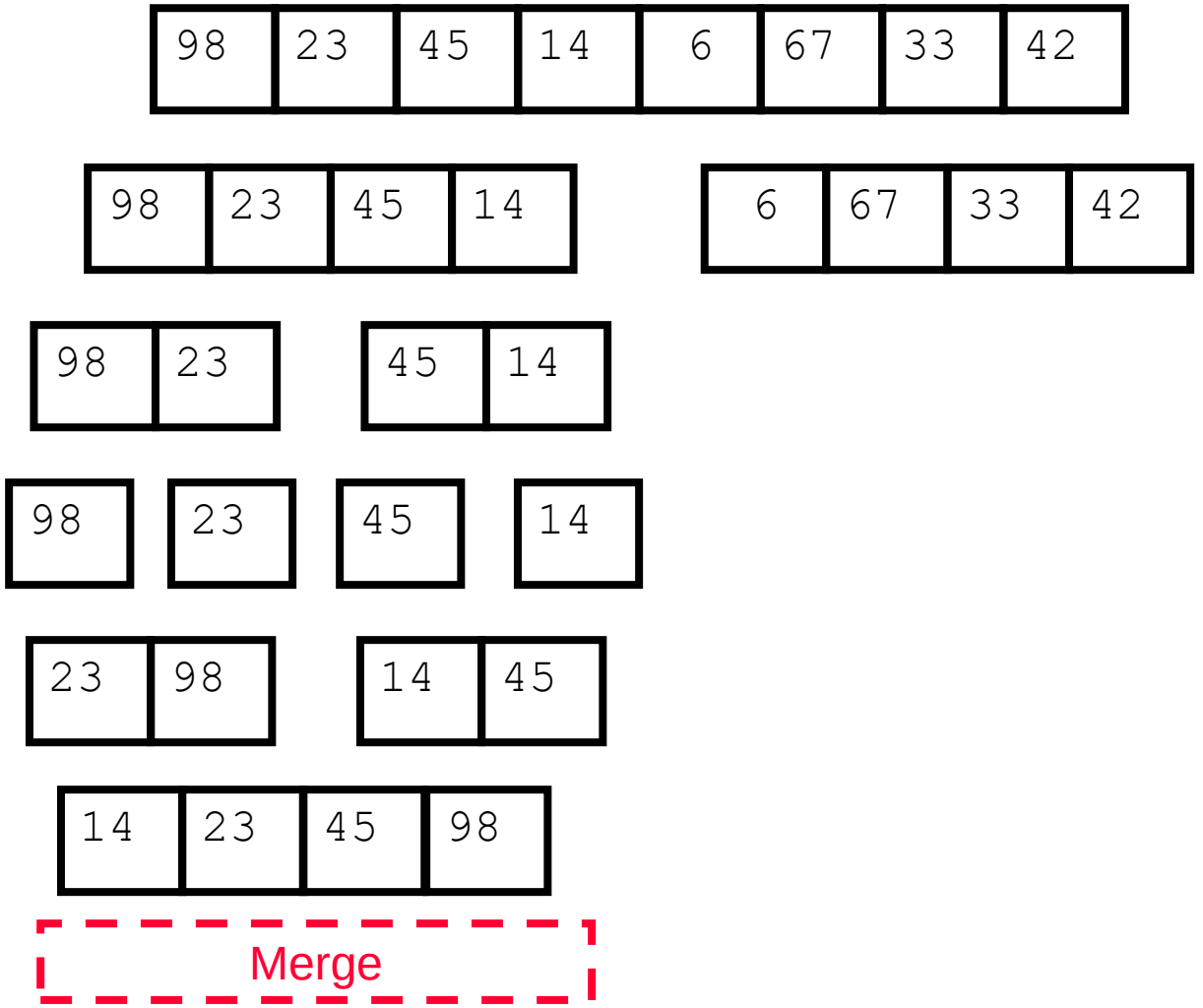

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

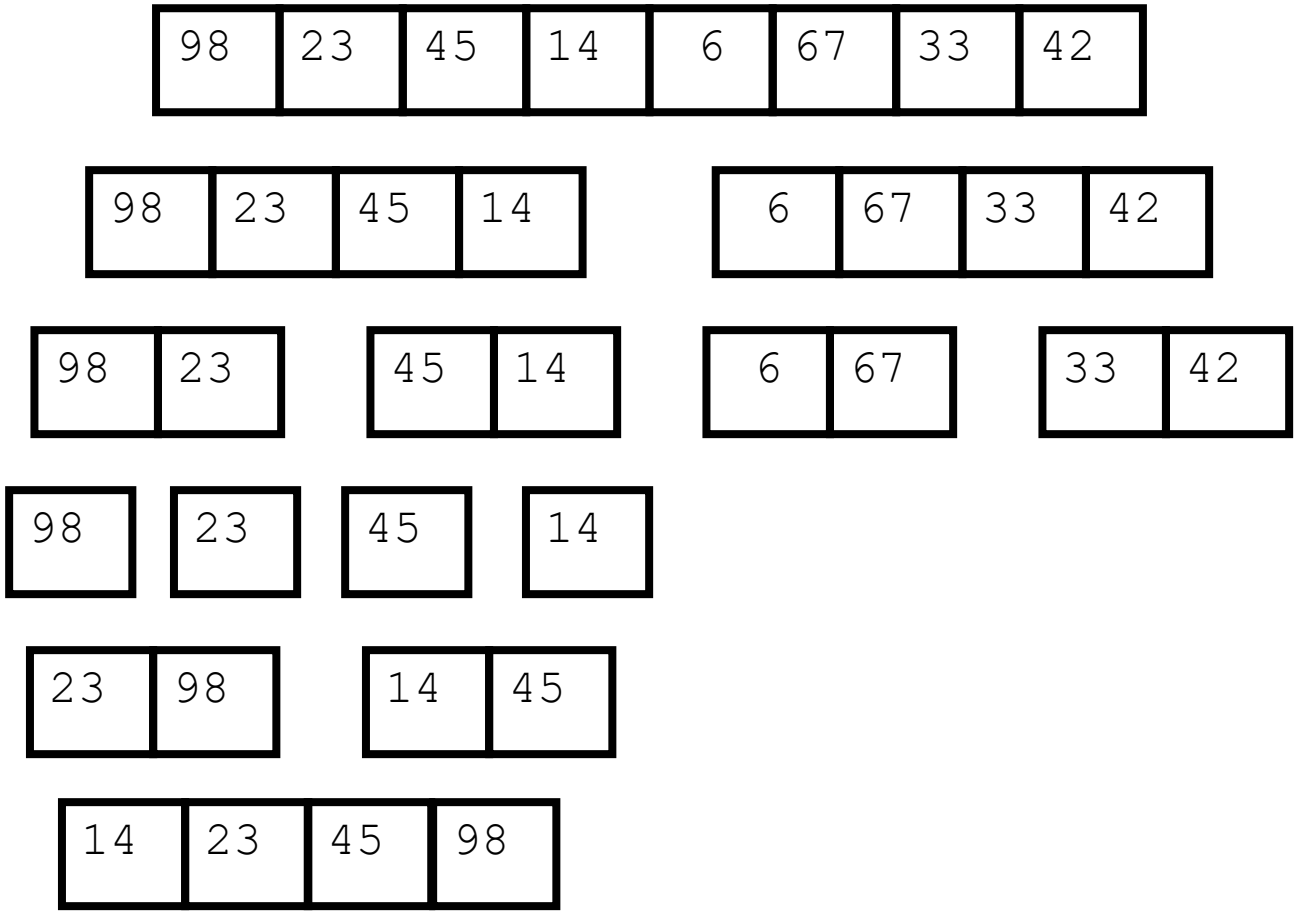

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

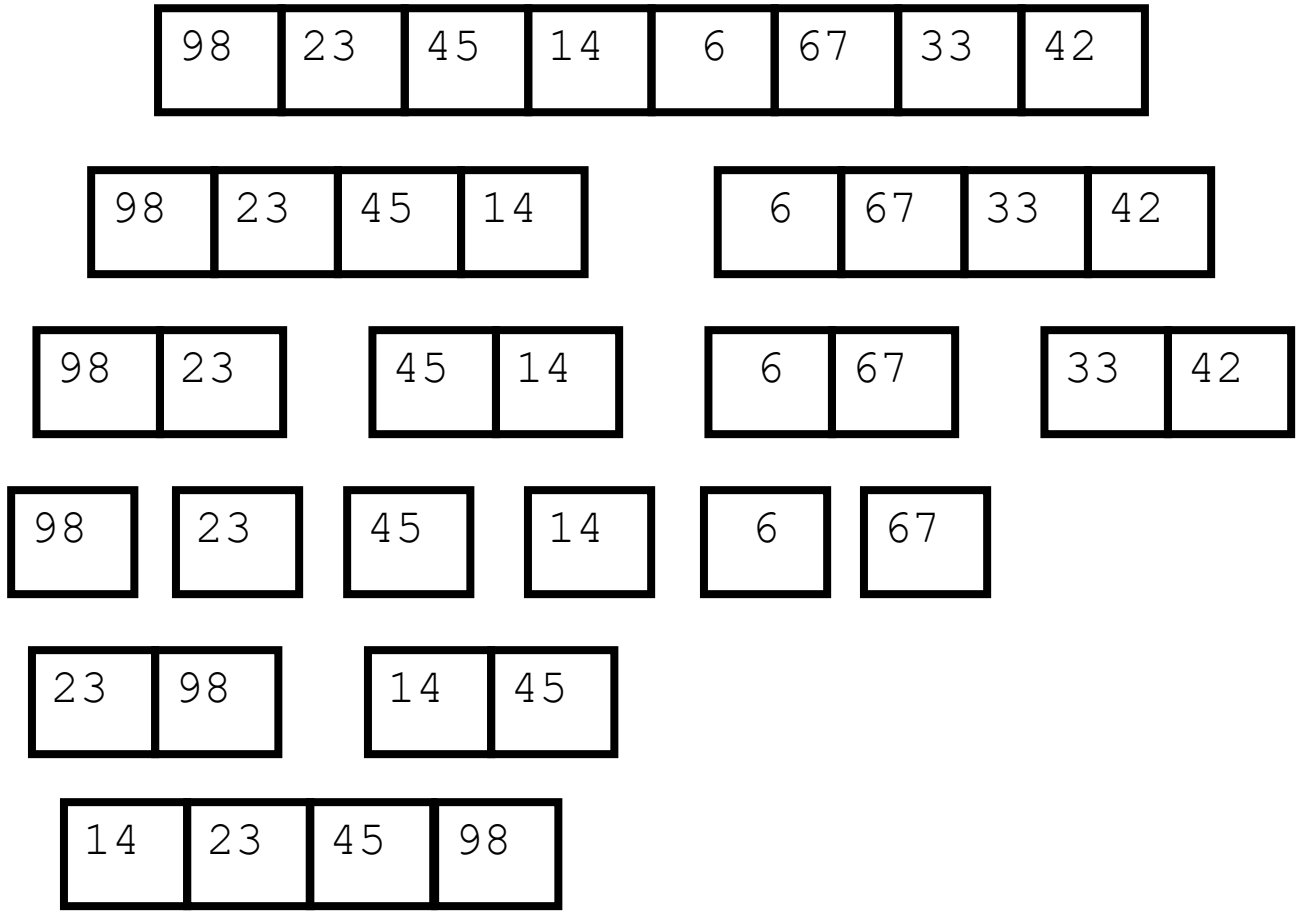

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

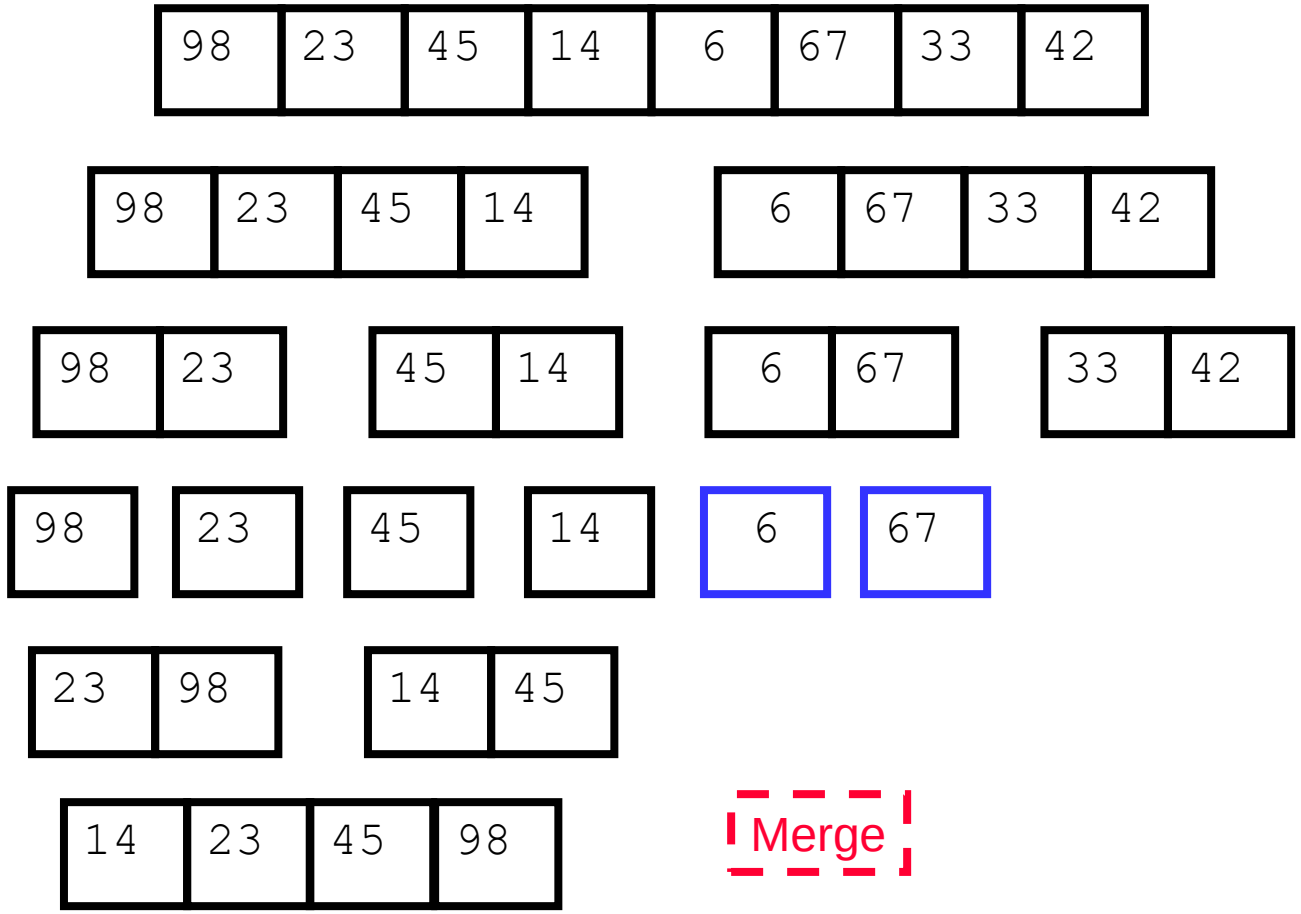

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

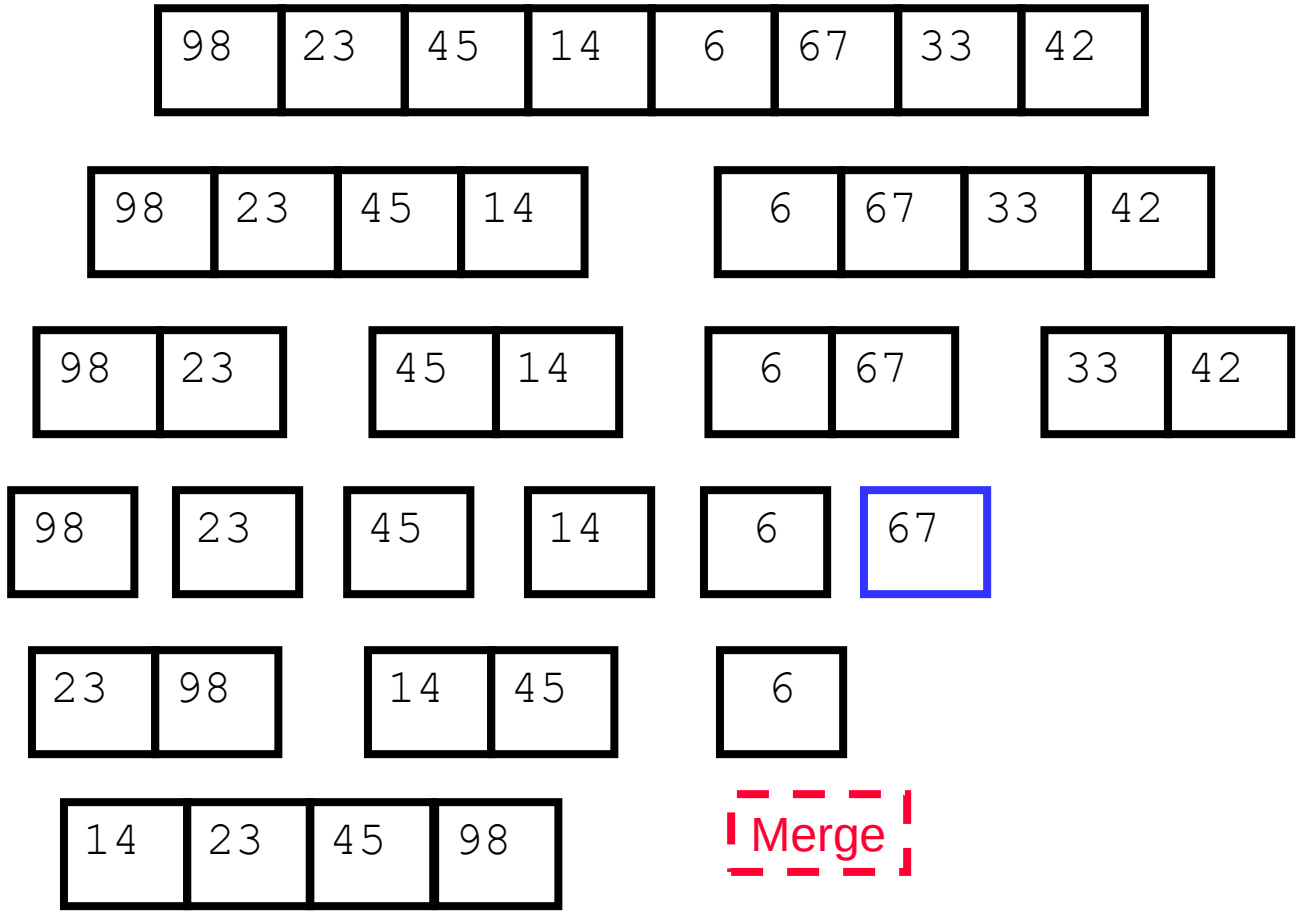

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

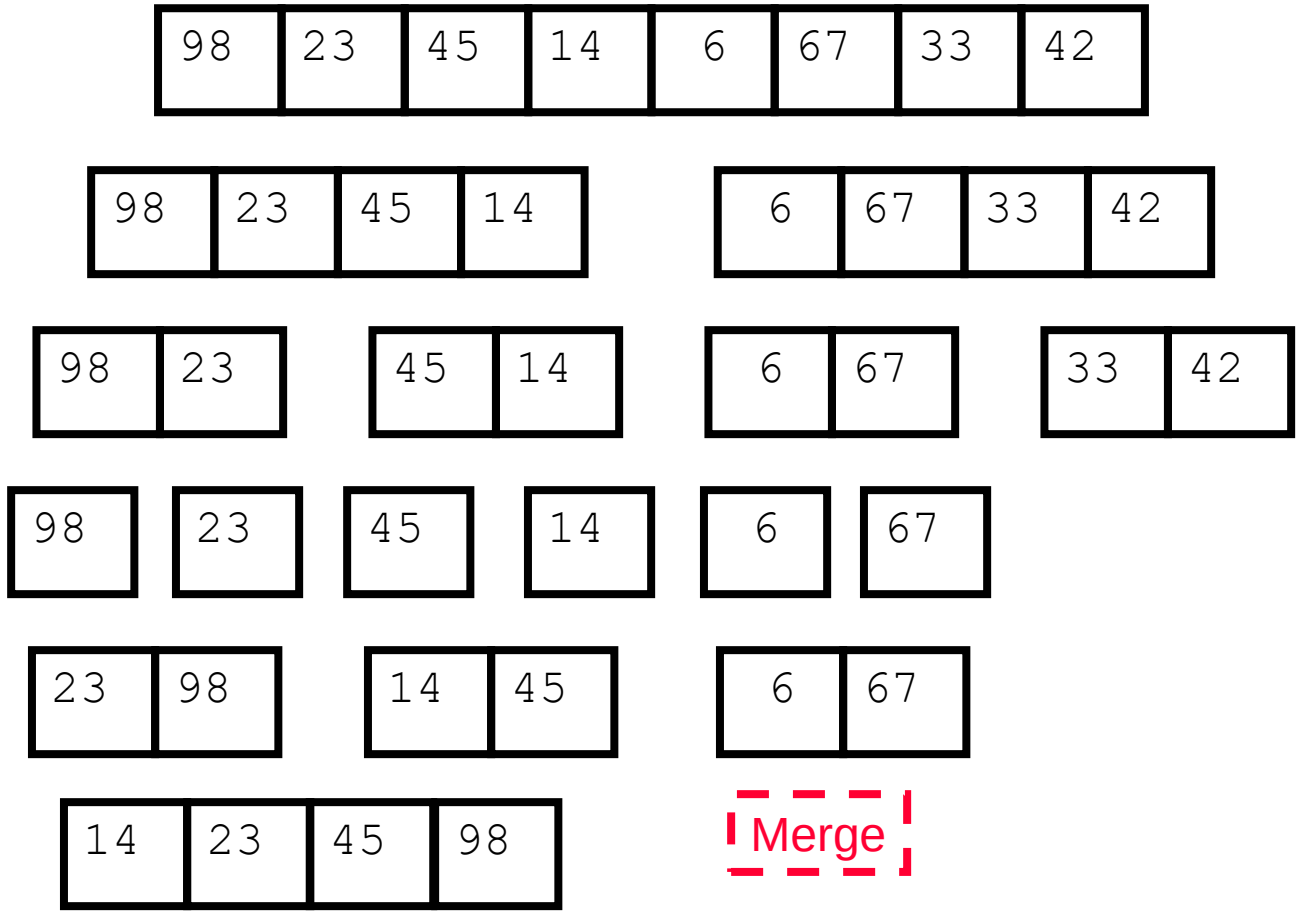

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

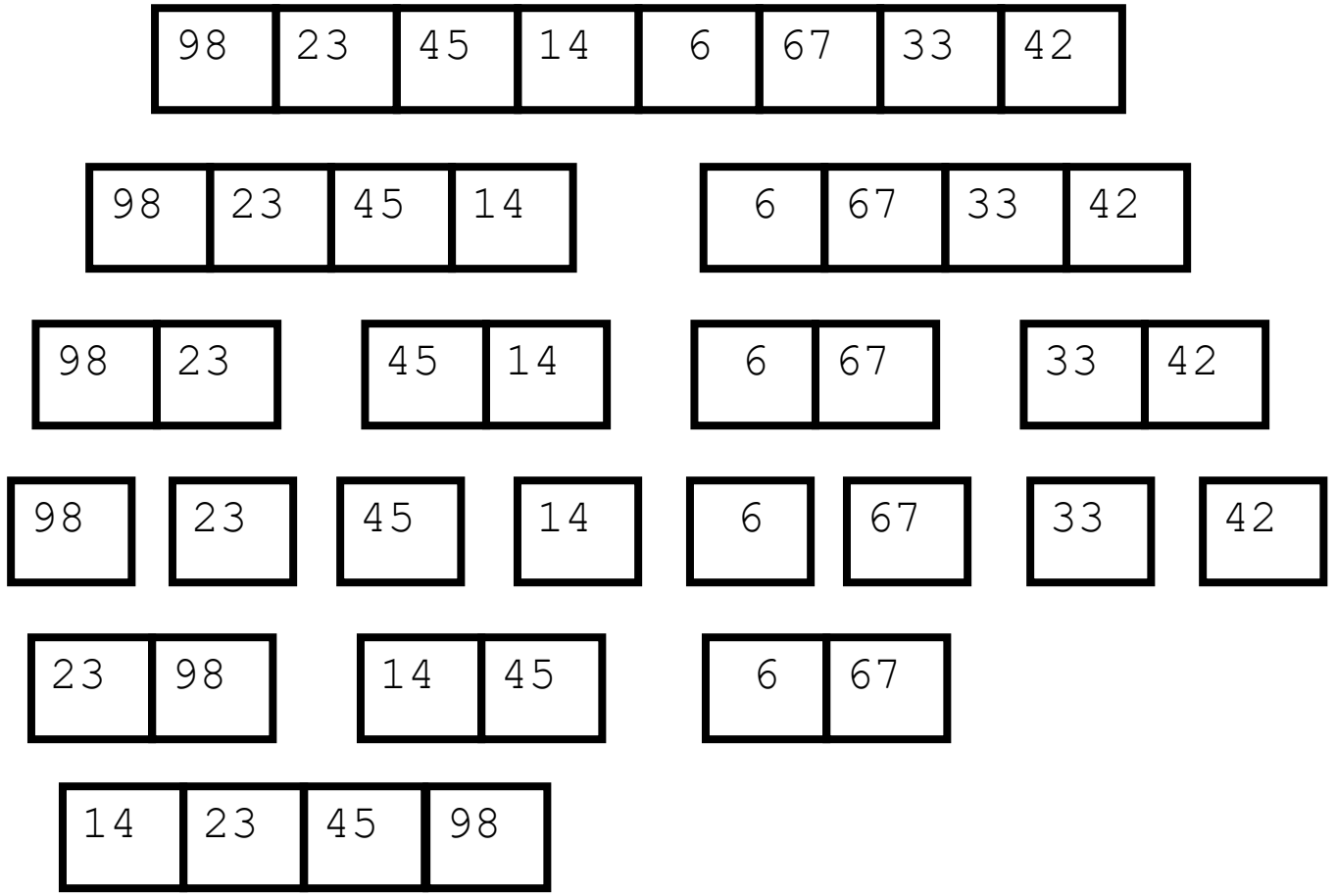

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

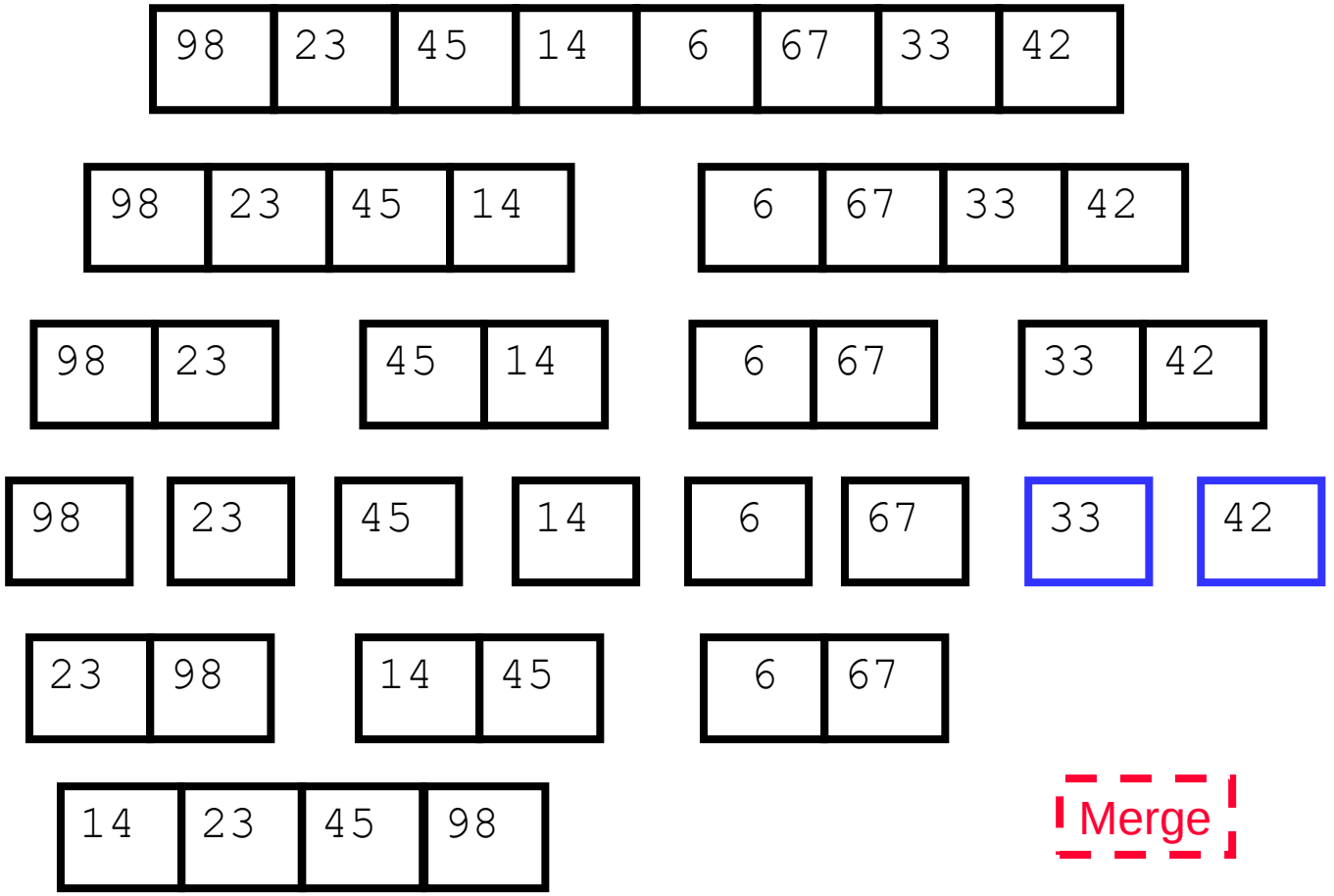

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

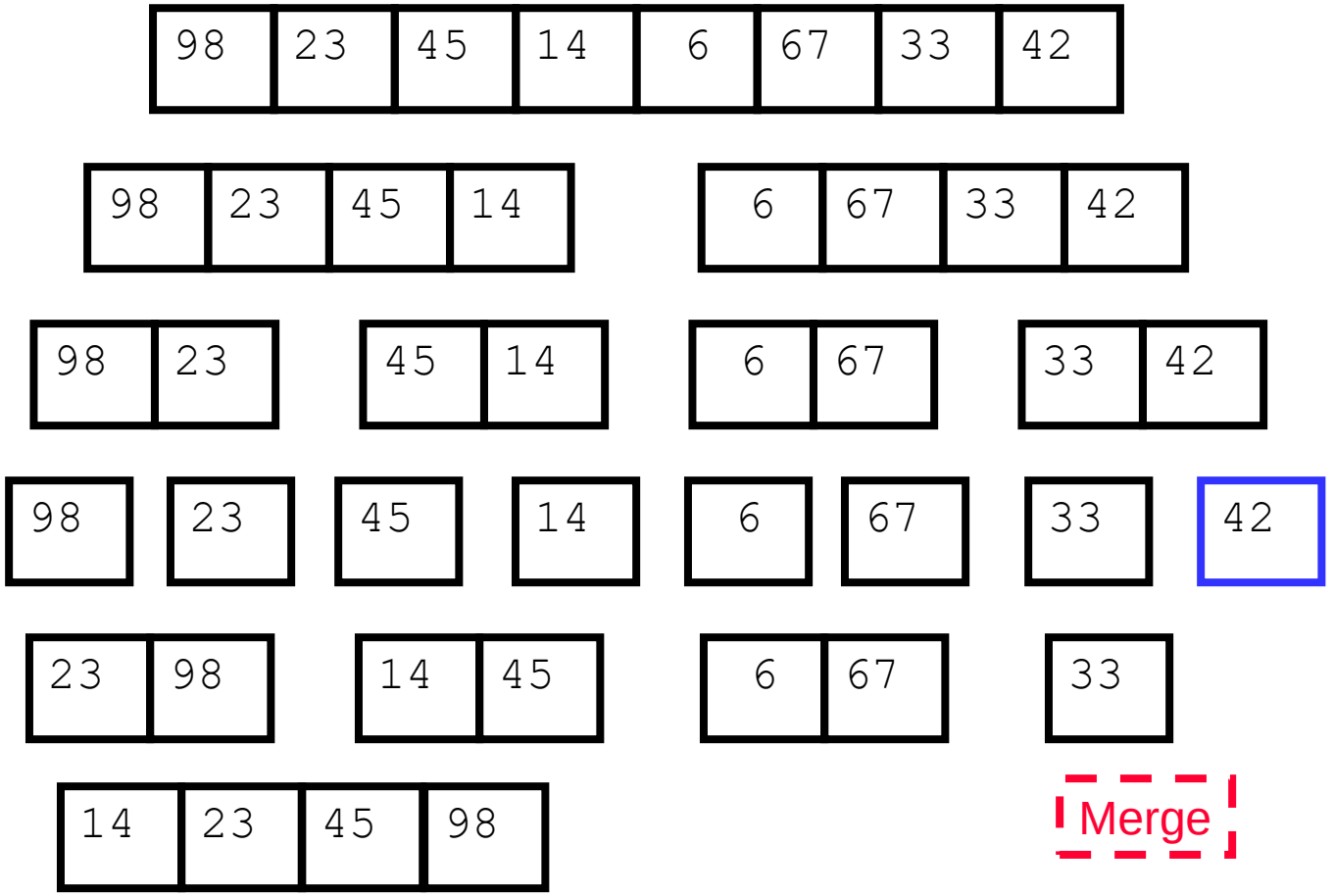

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

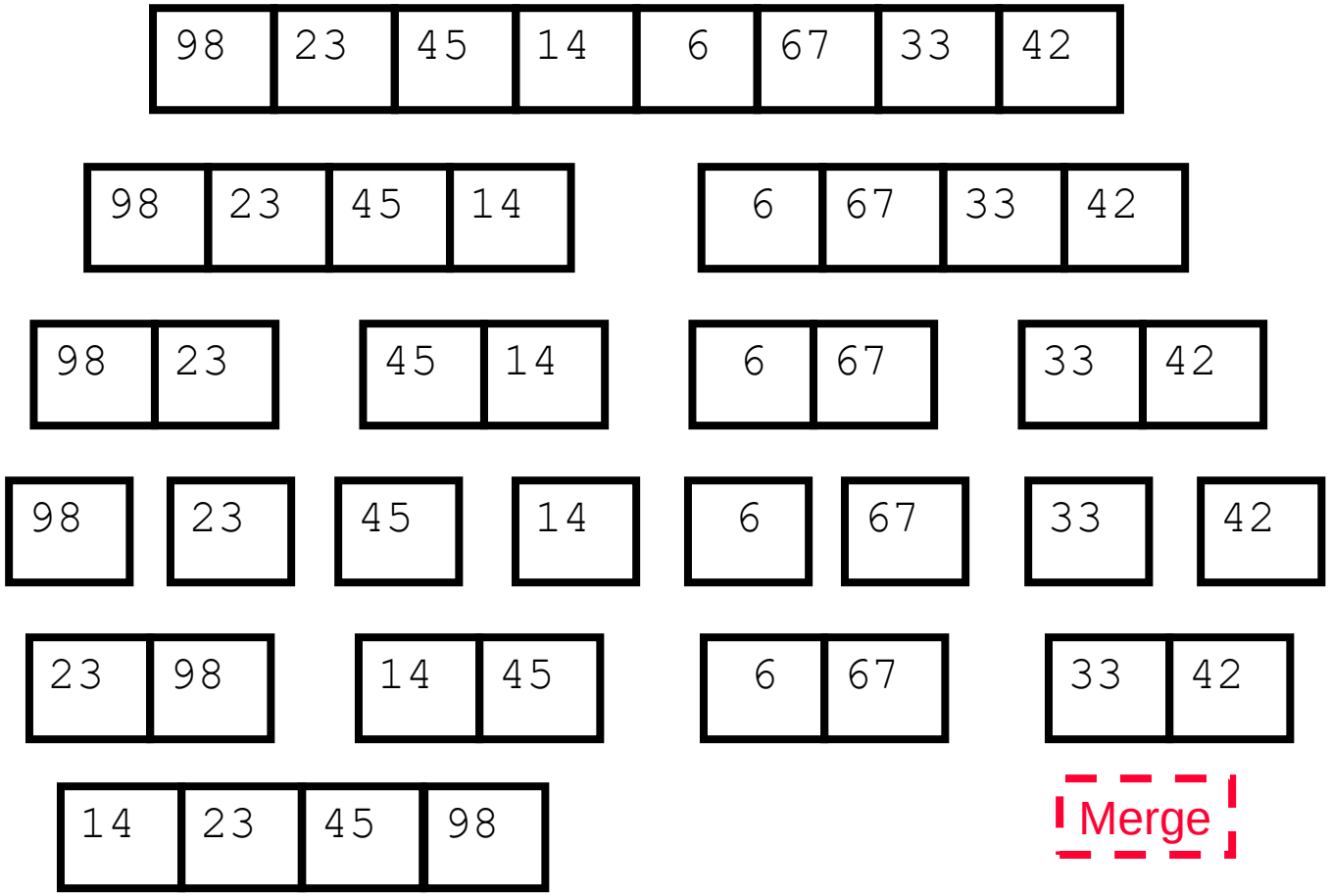

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge

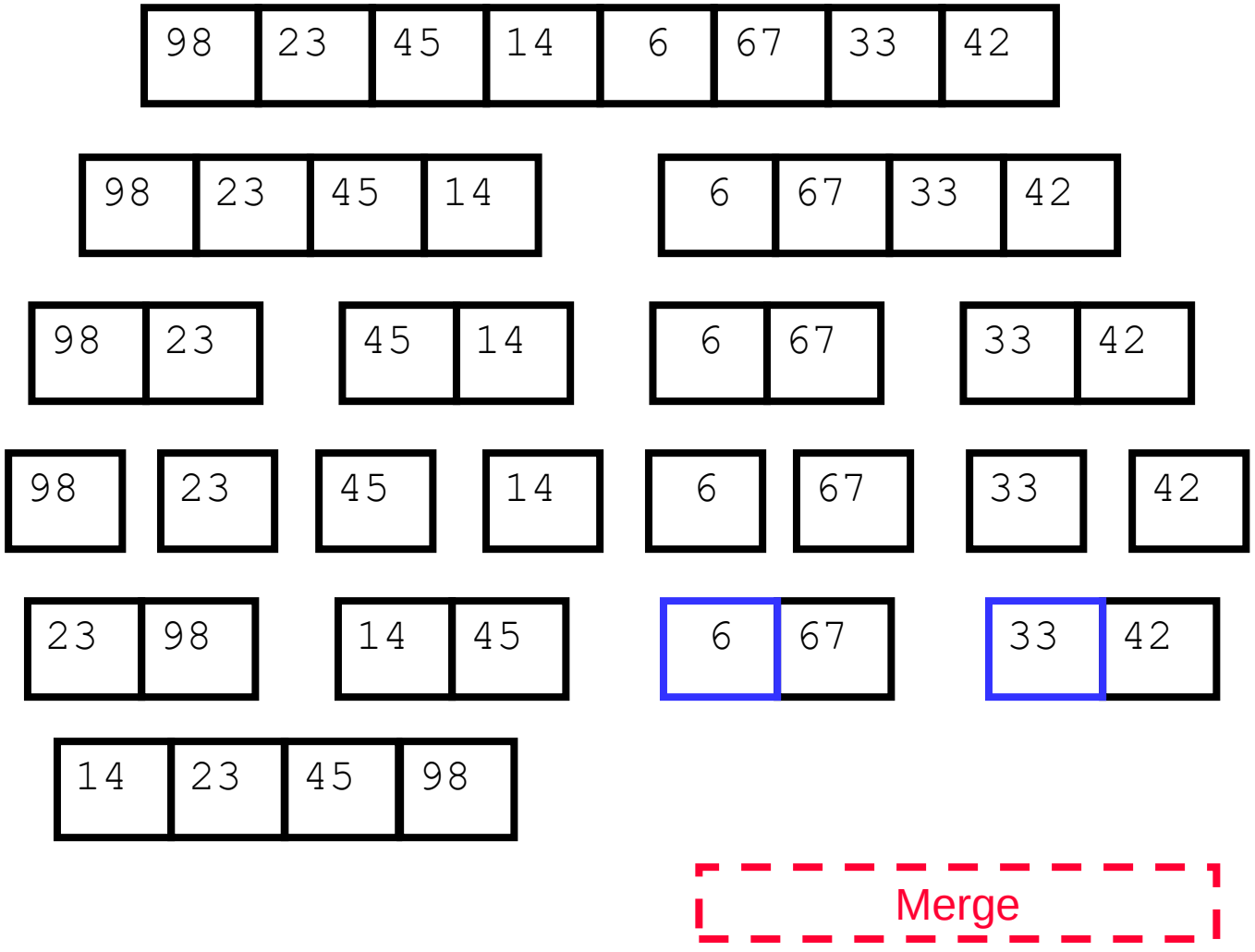

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6 67
14 23 45 98
Merge

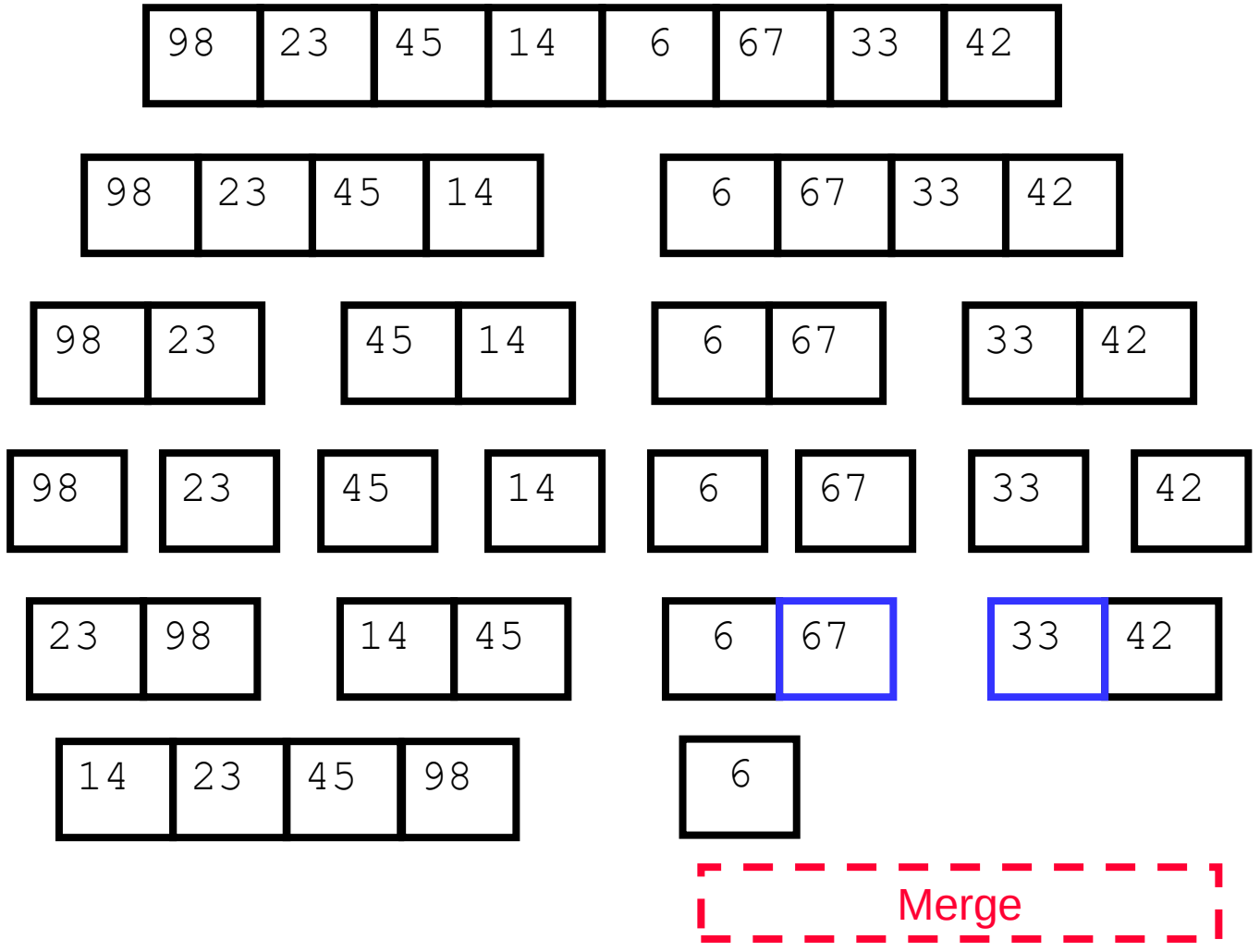

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98

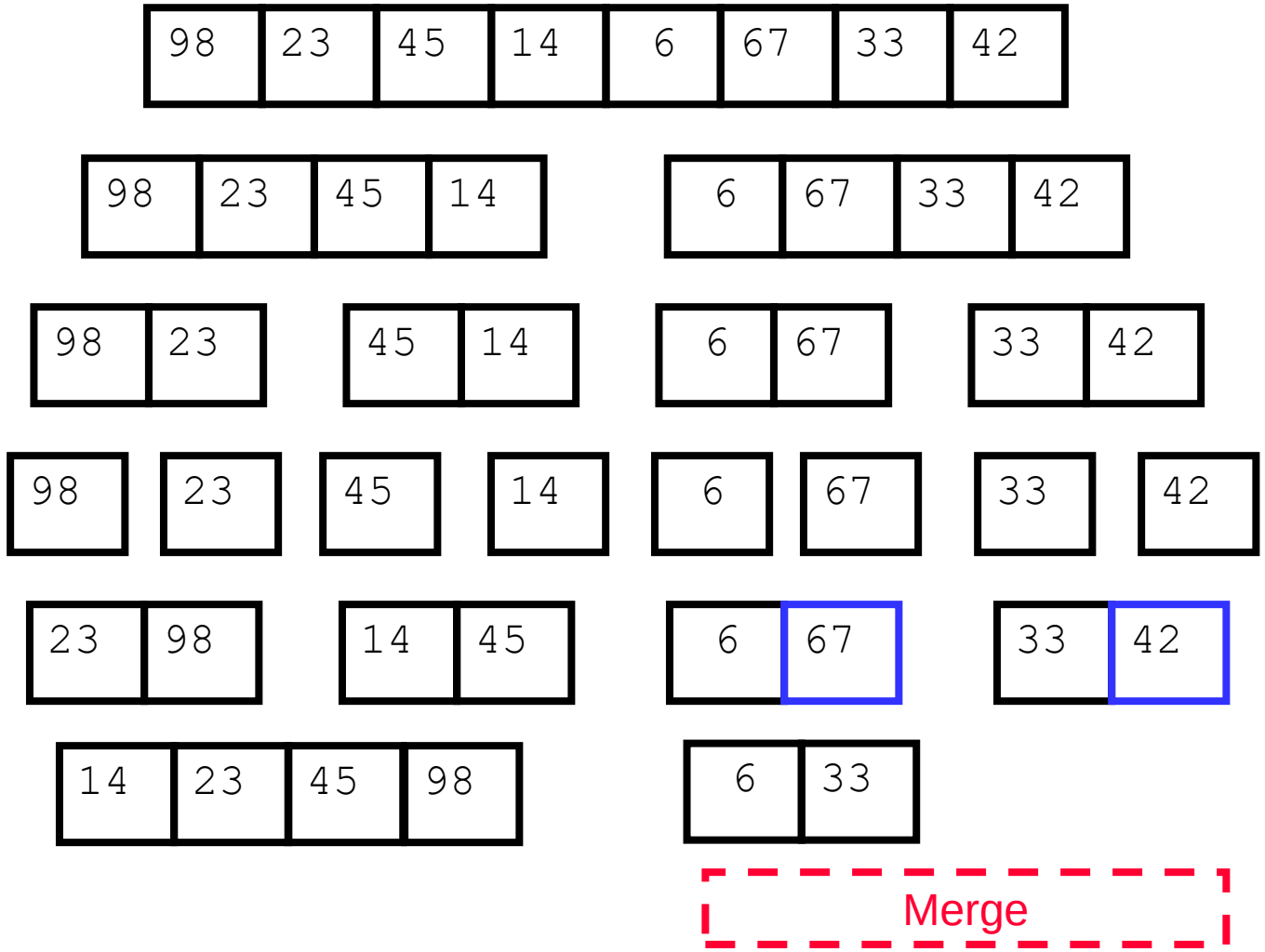

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98
Merge

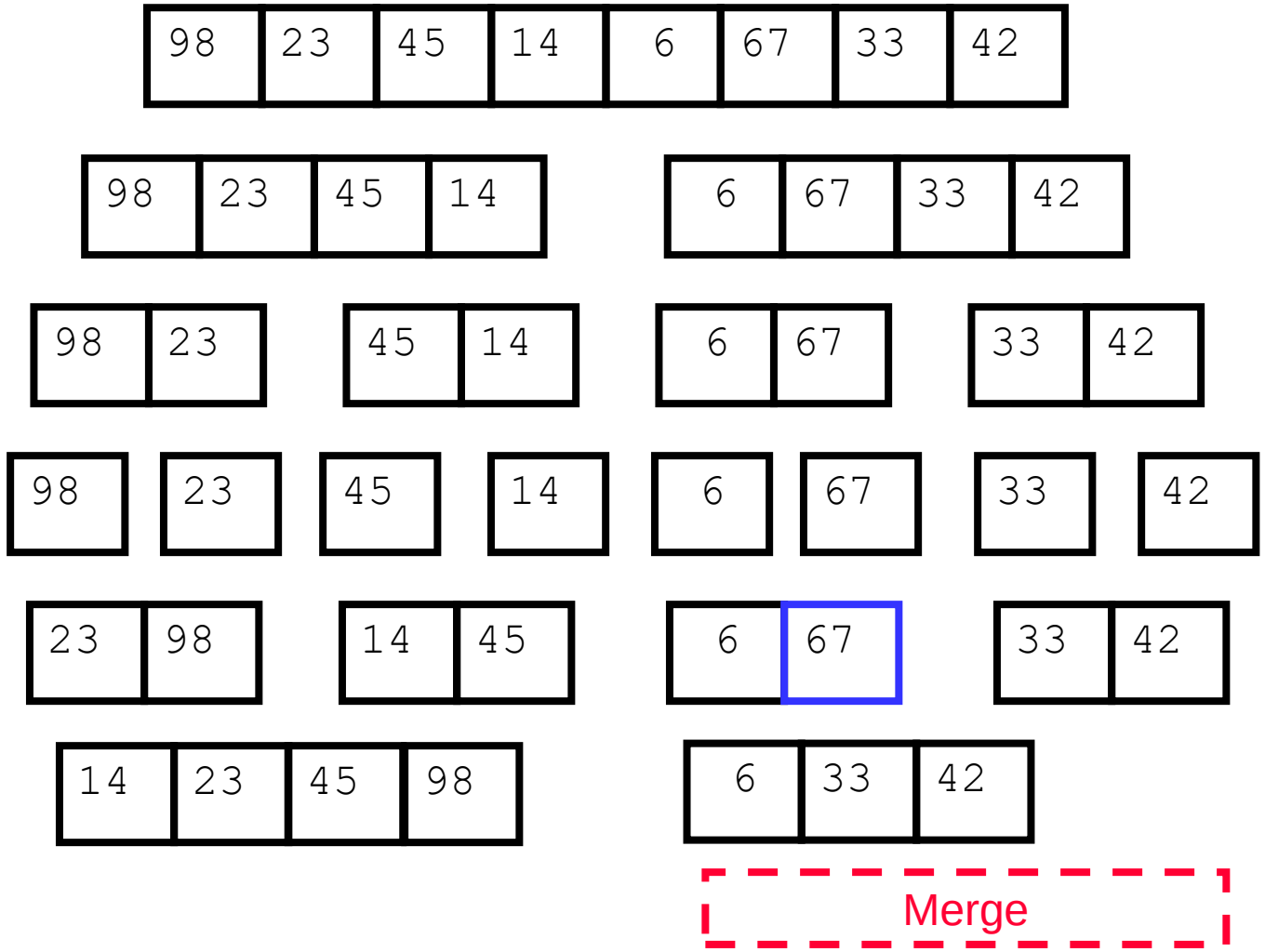

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge

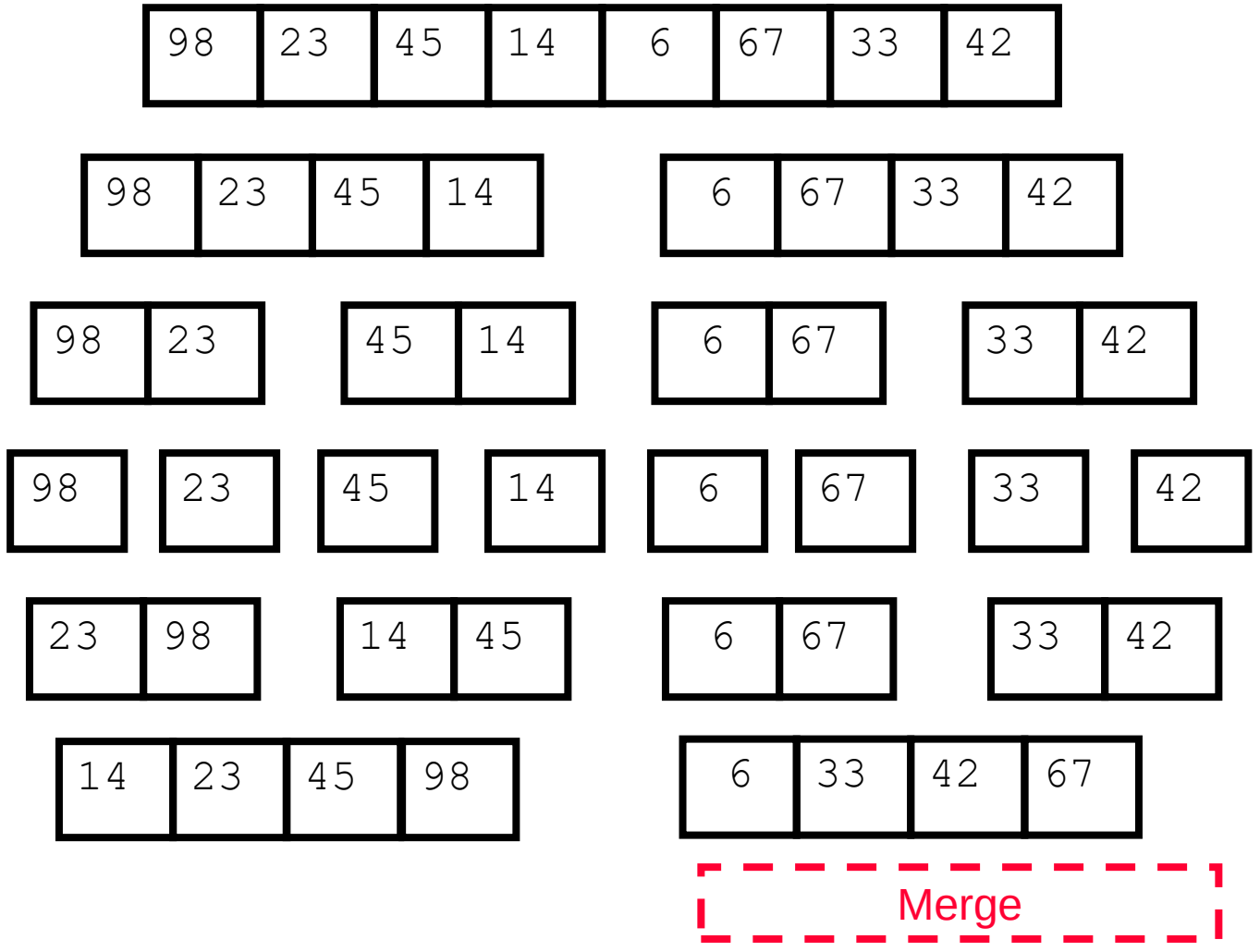

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

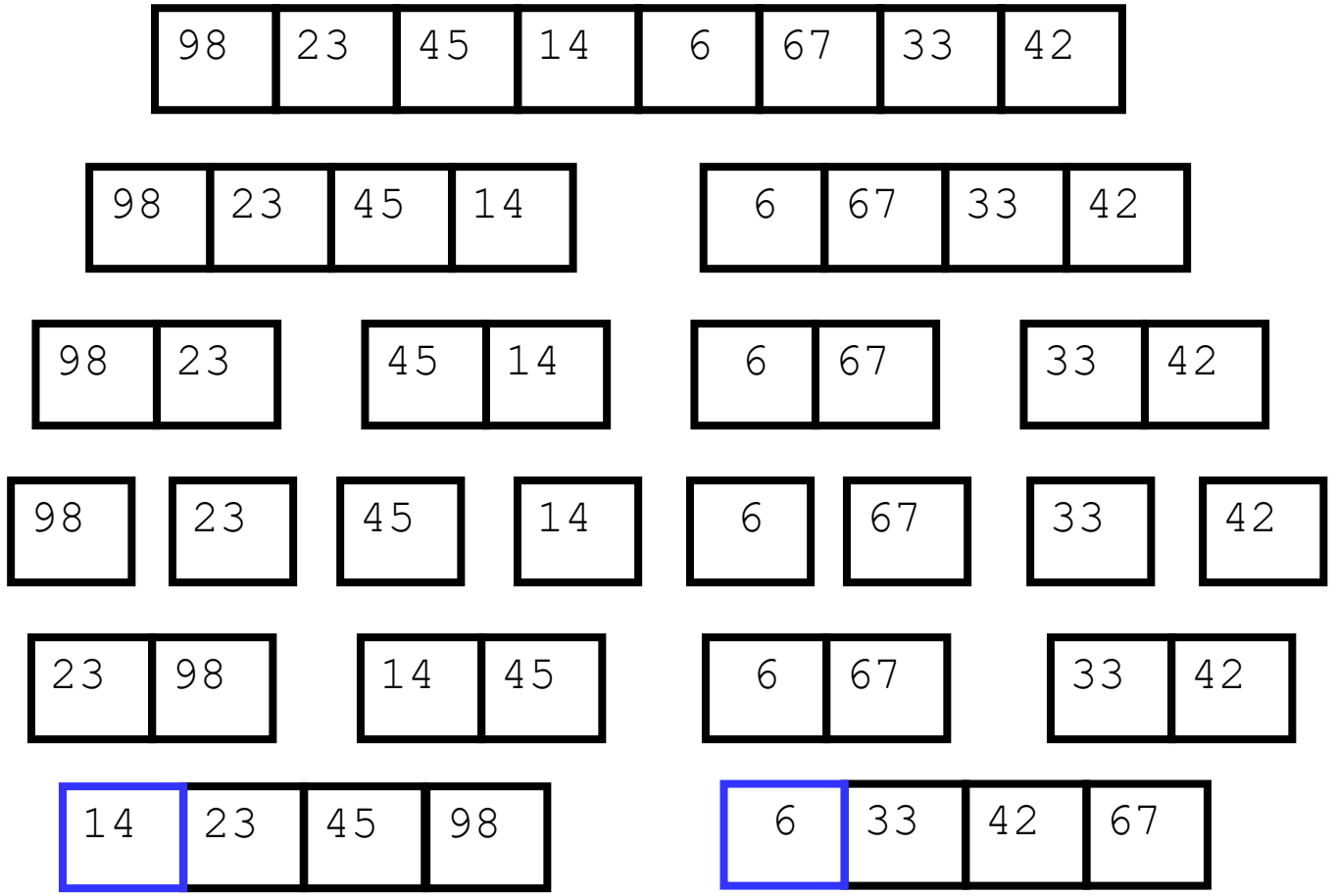

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

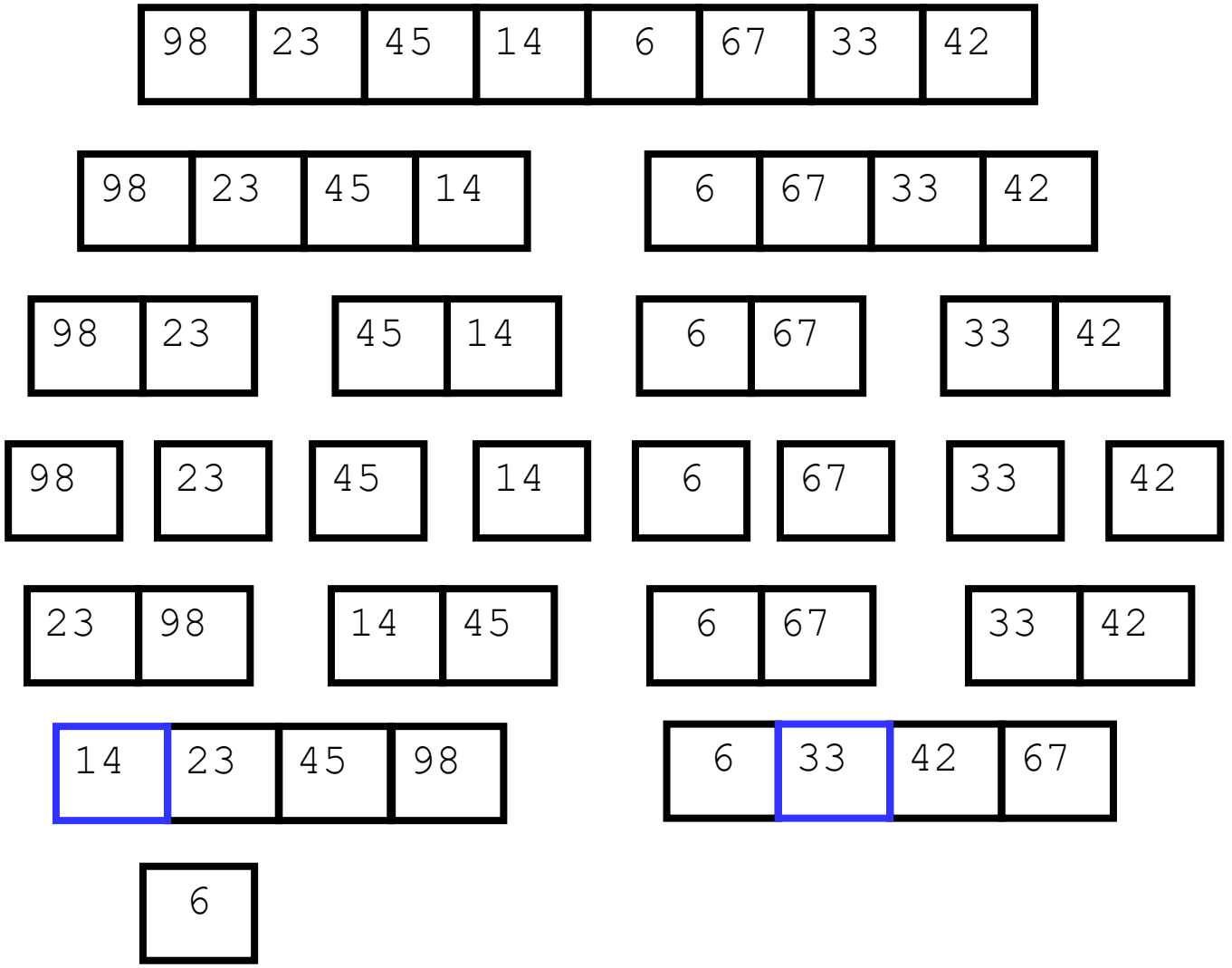

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6
Merge

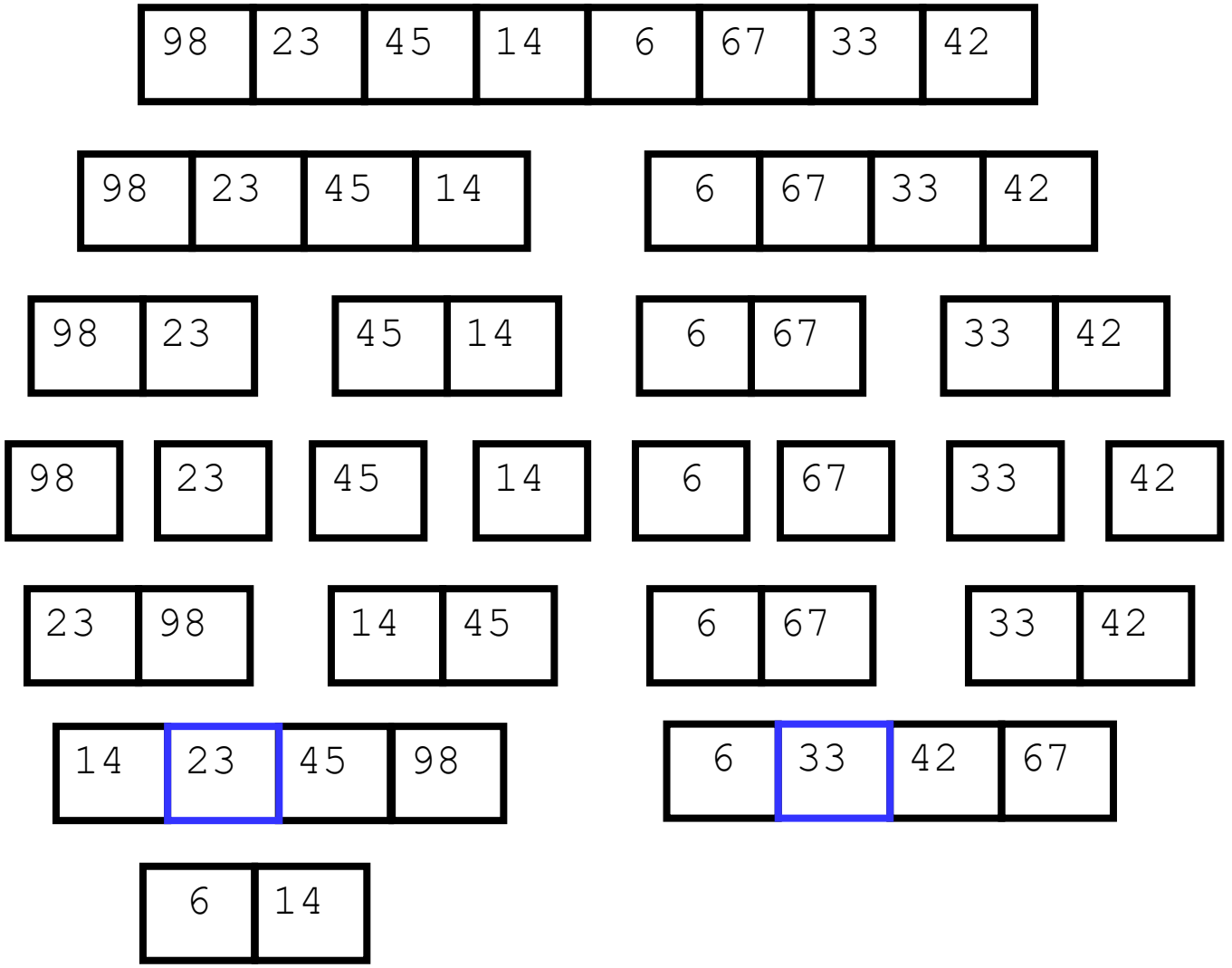

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33
Merge

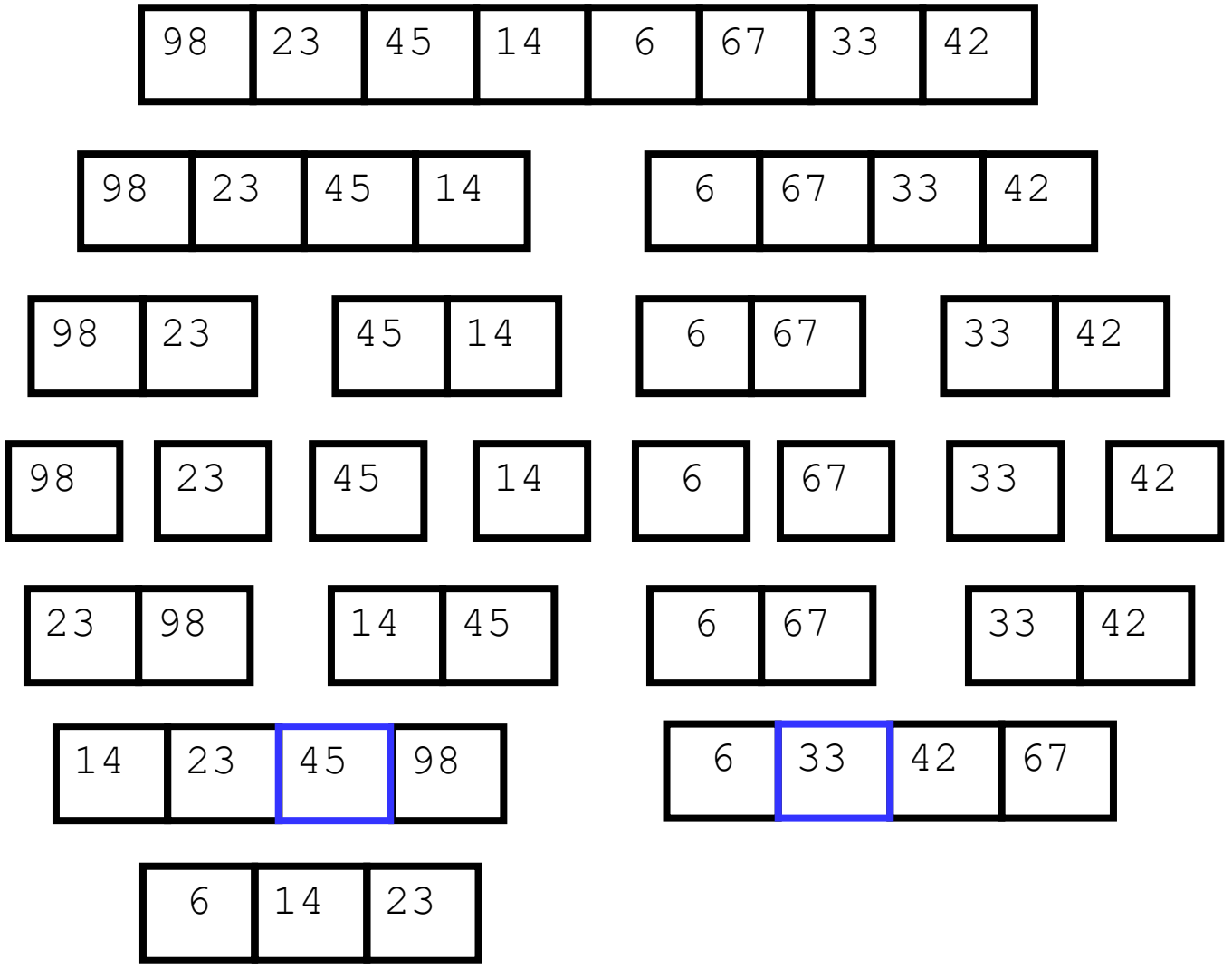

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42
Merge

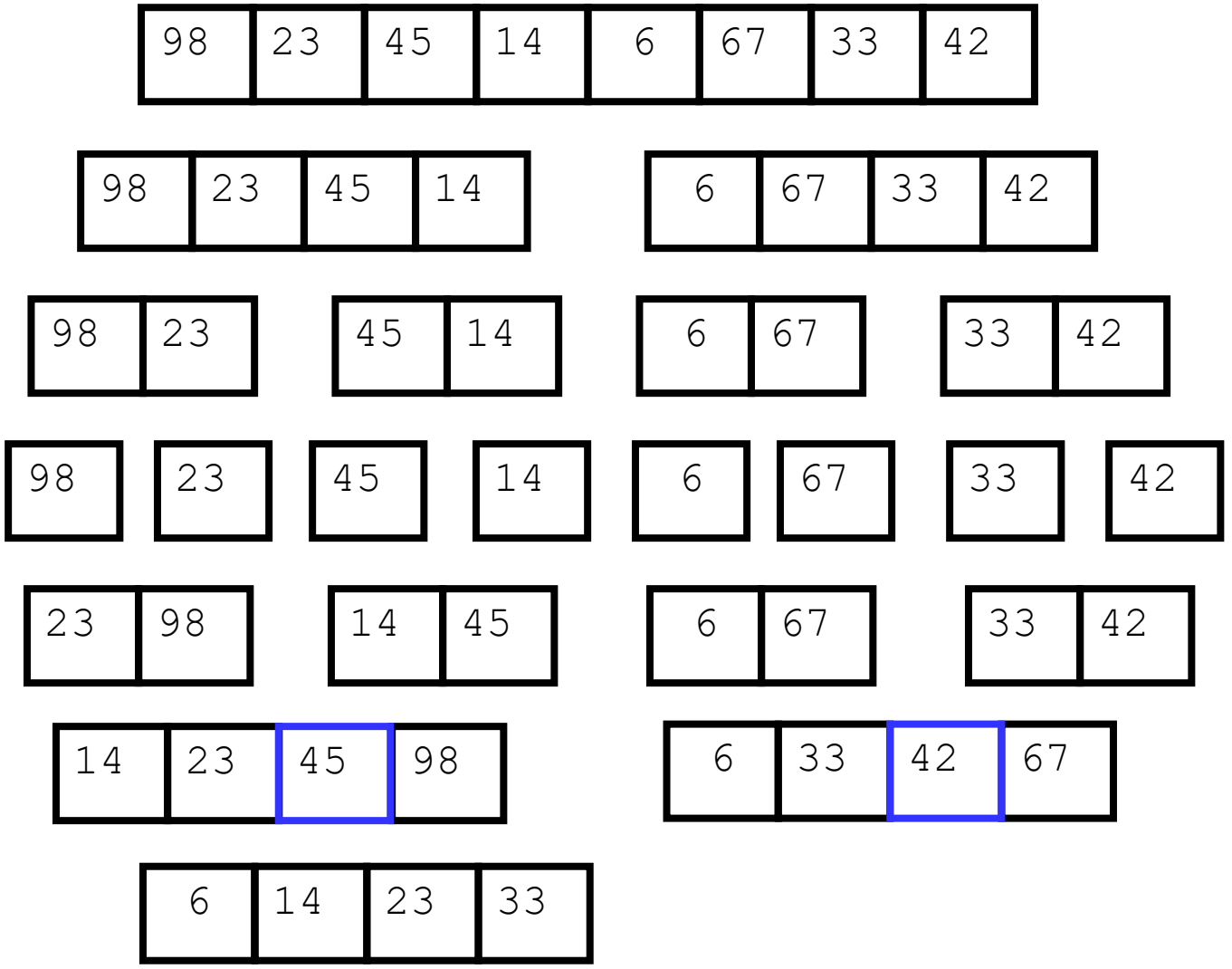

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

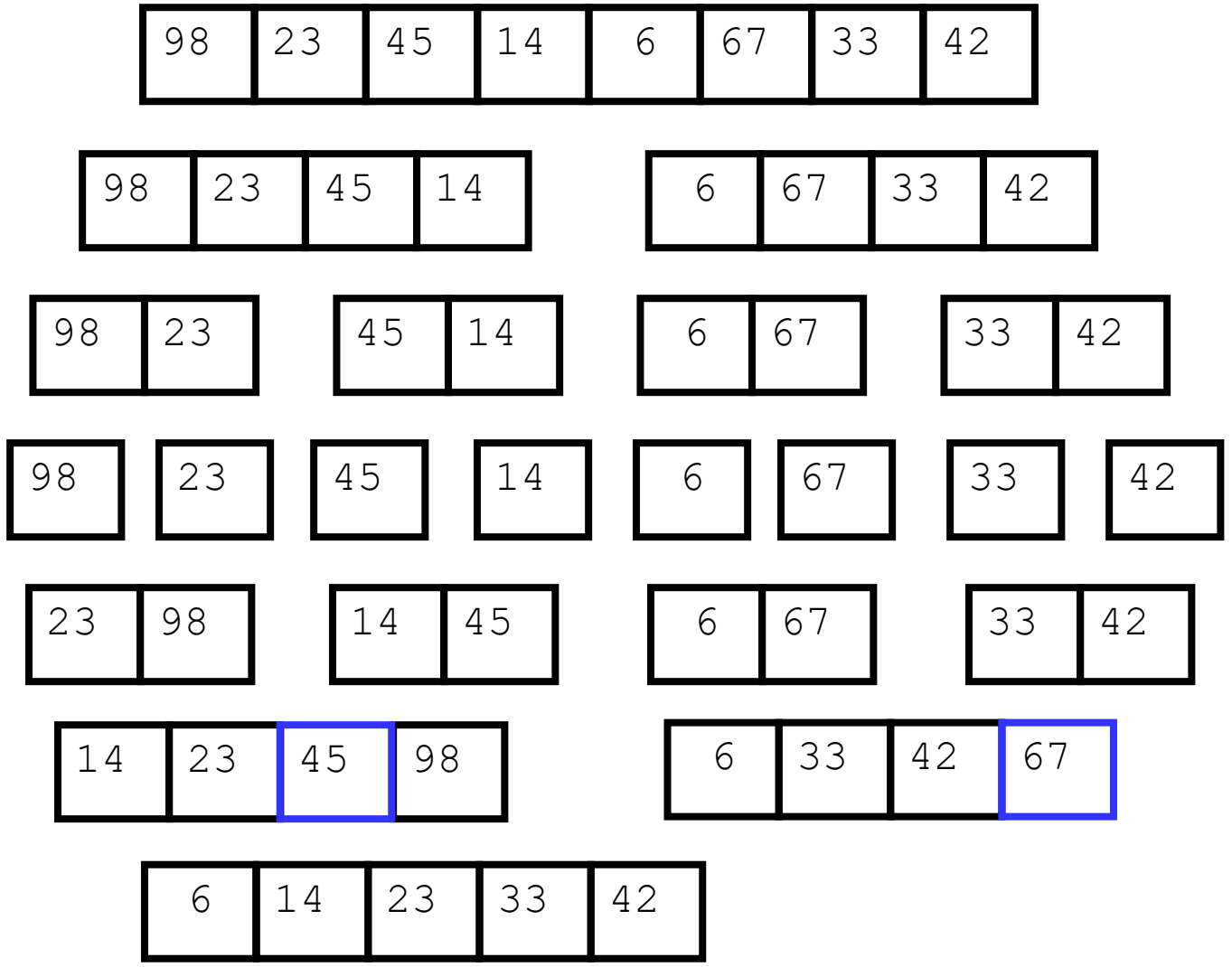

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

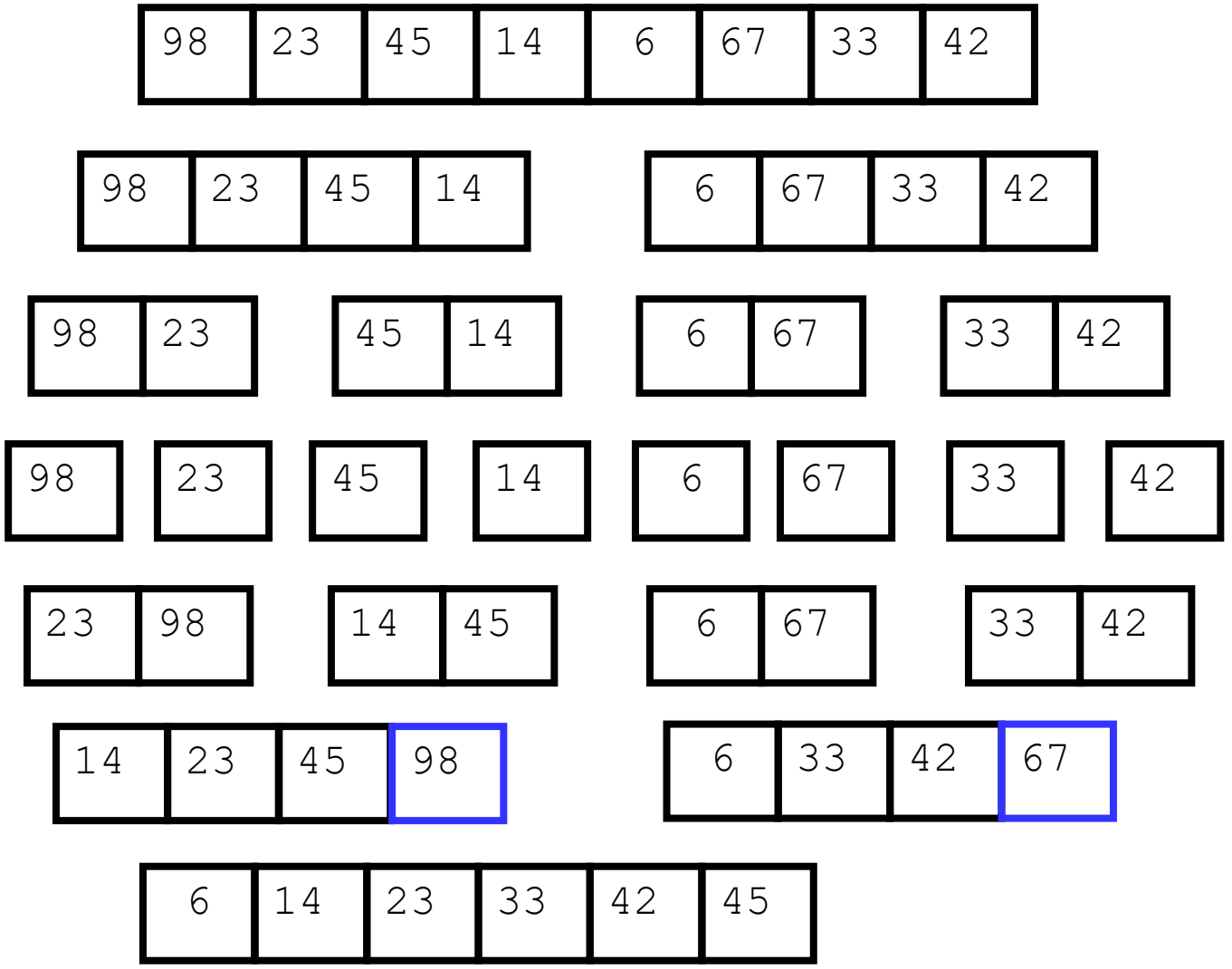

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6
Merge

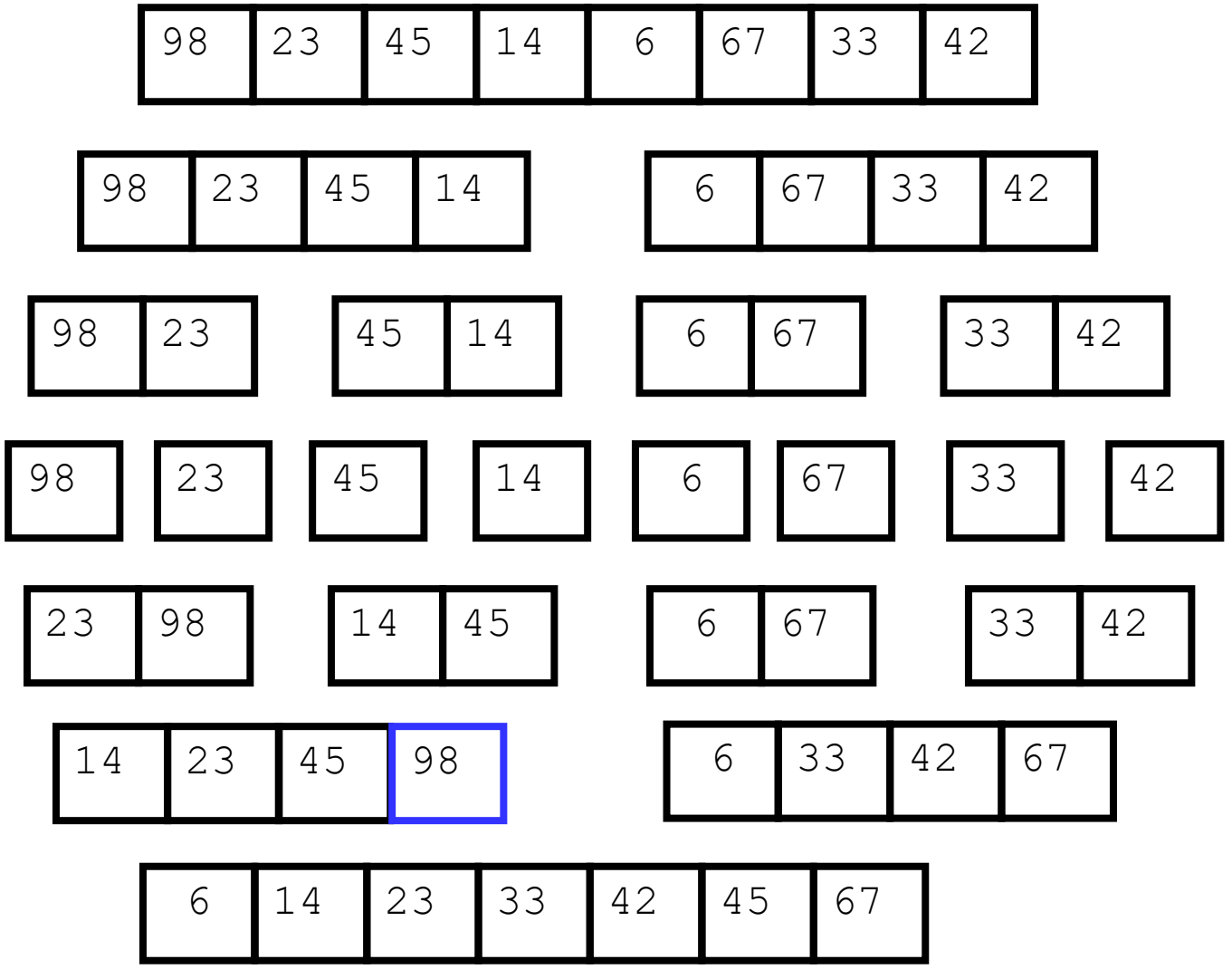

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14
Merge

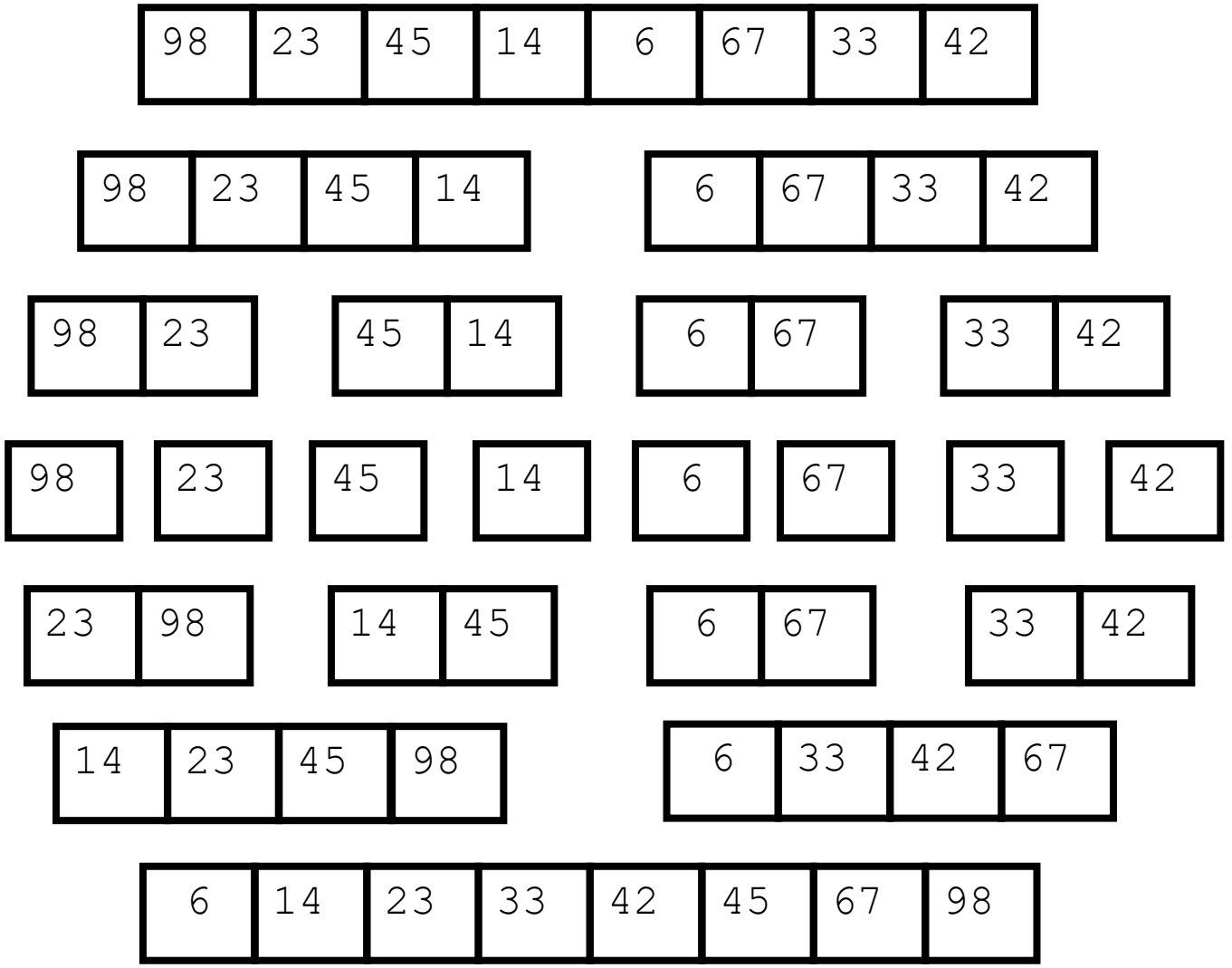

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23
Merge

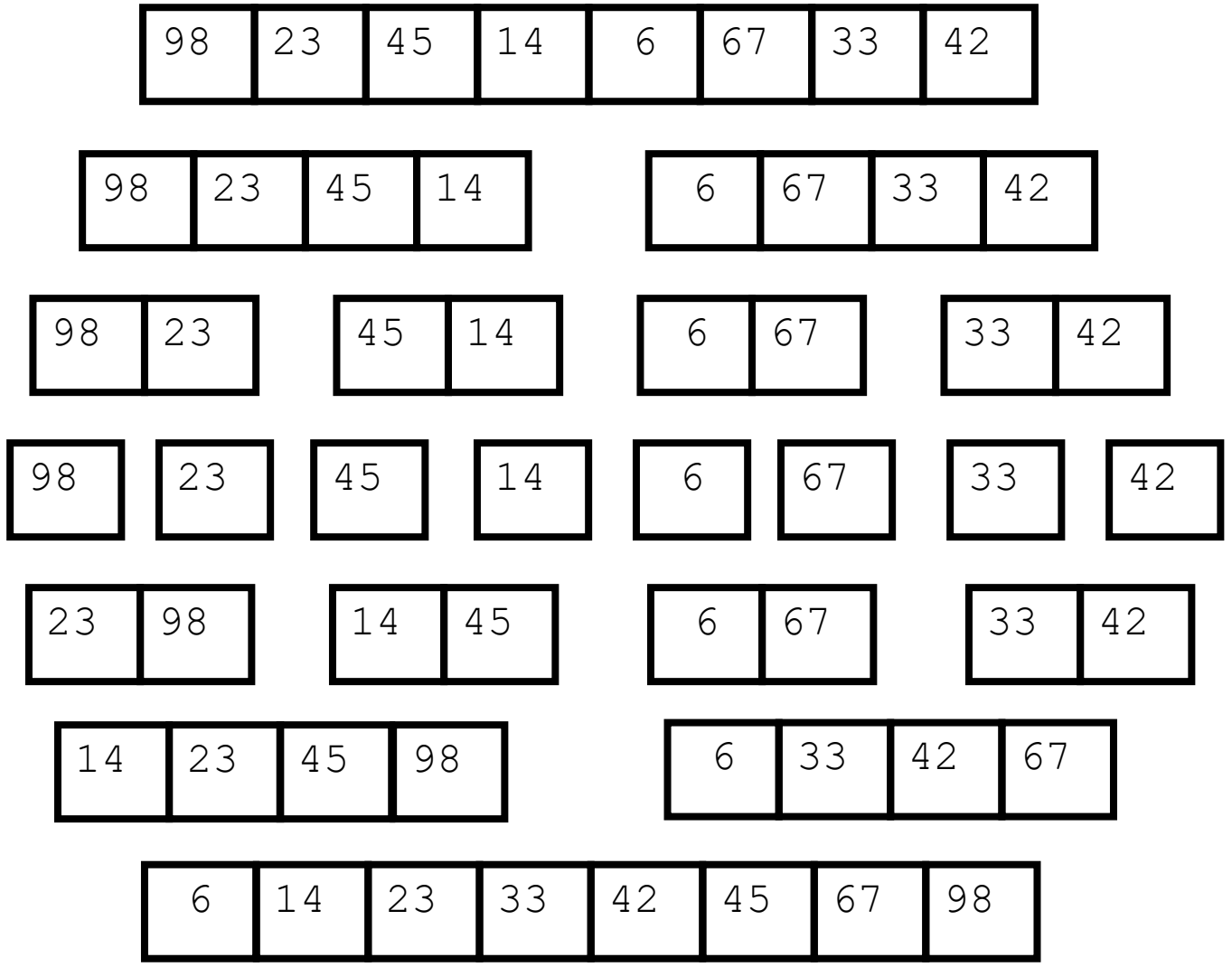

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33
Merge

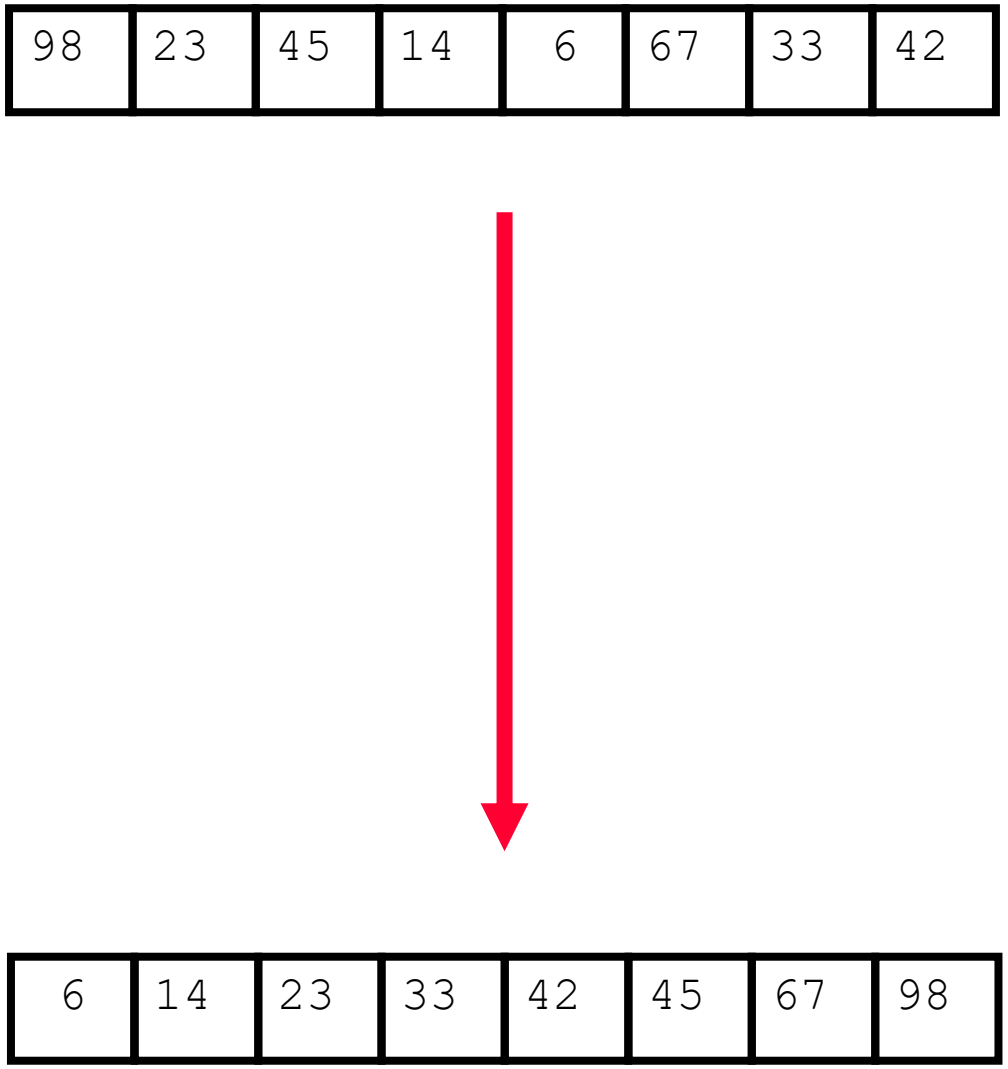

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42
Merge


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

Algorithm

1. *step 1: start*
2. *step 2: declare array and left, right, mid variable*
3. *step 3: perform merge function.*

   if left > right

   return

   mid= (left+right)/2

   mergesort(array, left, mid)

   mergesort(array, mid+1, right)

   merge(array, left, mid, right)
4. *step 4: Stop*

Quick sort

It is a divide and conquer algorithm.

1. Step 1 - Pick an element from an array, call it as pivot element.
2. Step 2 - Divide an unsorted array element into two arrays.
3. Step 3 - If the value less than pivot element come under first sub array, the remaining elements with value greater than pivot come in second sub array.

Consider an example given below, wherein

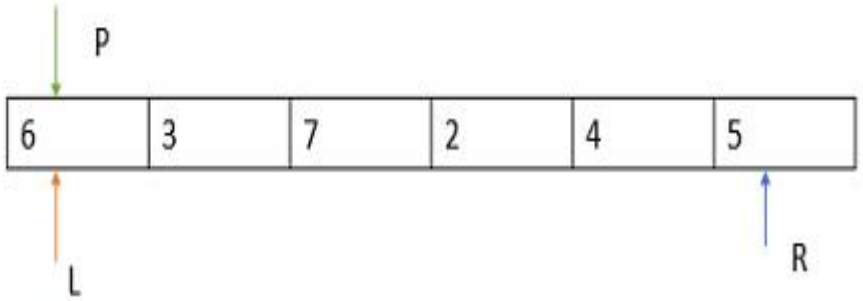
i. P is the pivot element.

ii. L is the left pointer.

iii. R is the right pointer.

The elements are 6, 3, 7, 2, 4, 5.

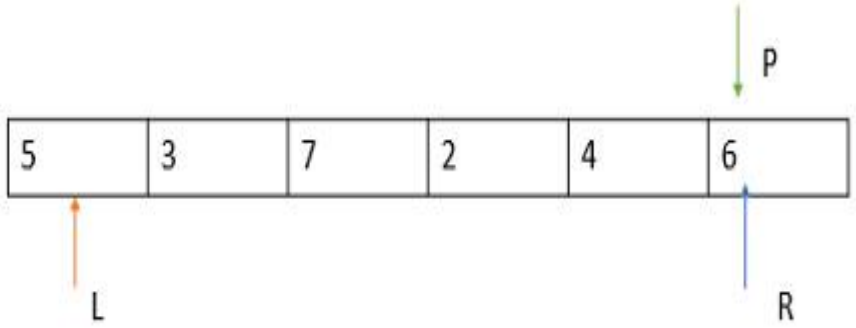
Quick sort



Case 1: P=6 {Right side P is greater and Left side of is Leese}

Is P<R => 6<5 {wrong}

So, swap P with R

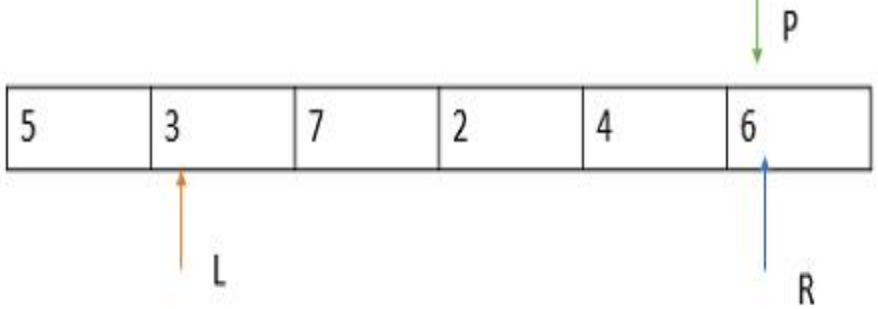


Case 2: P=6 , L=5 {Right side P is greater and Left side of is Lesser}

Is P>L => 6> 5 {right}

Move L towards right

Case 3:

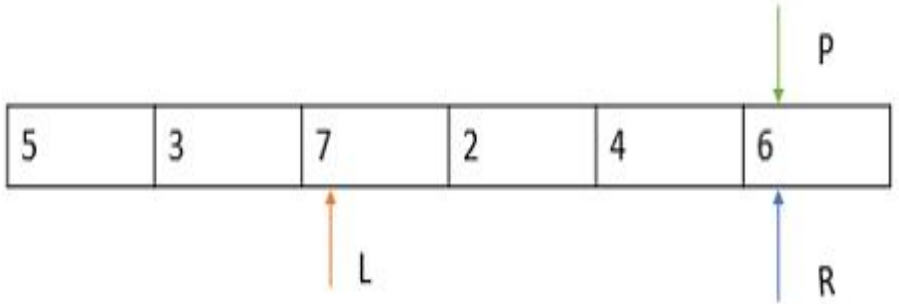


Is P>L, 6>3, Yes

So, move L towards right

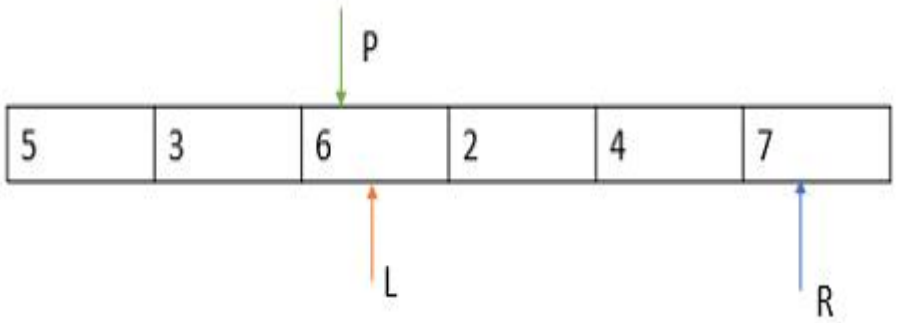
Quick sort

Case 4:



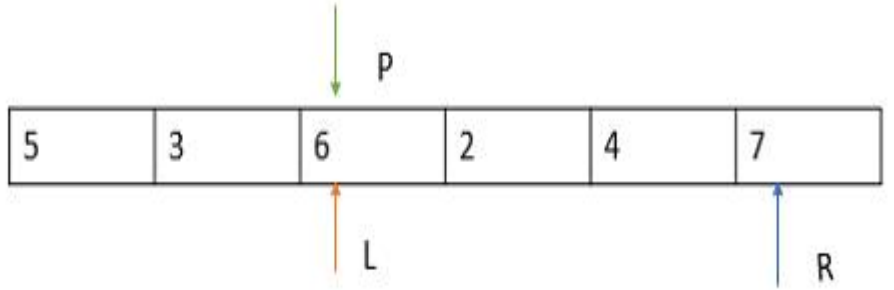
Is P>L => 6> 7 {wrong}
then swap P and L

Case5 :



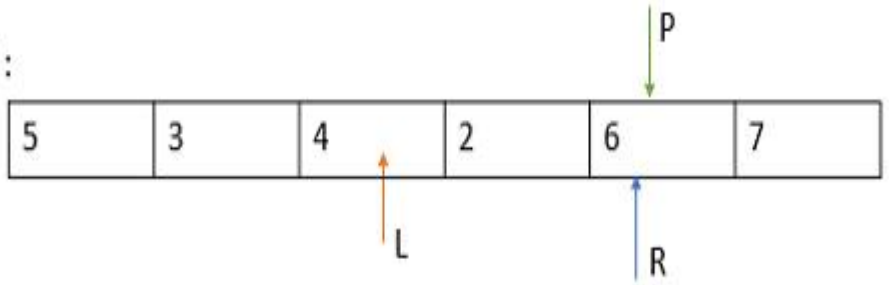
Is P<R => 6<7, => Yes
Decrement R

Case 6:



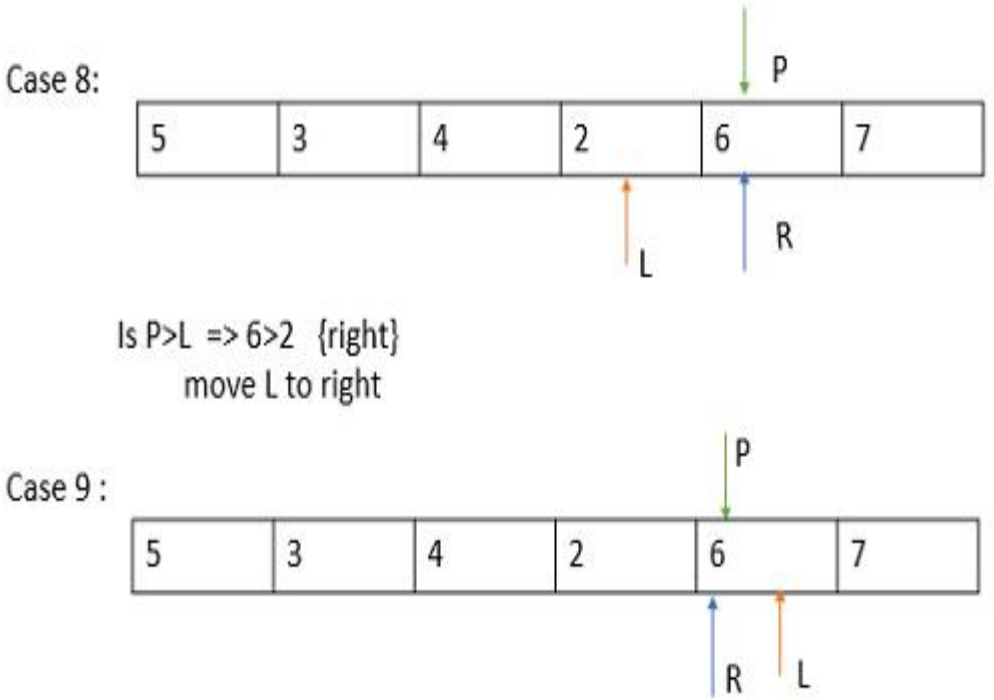
Is P<R => 6<4 {wrong}
then swap

Case 7 :



Is P>L => 6>4, => Yes
Move L to right

Quick sort

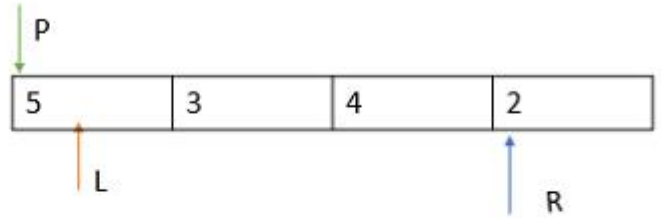


Now,

- The pivot is in fixed position.
- All the left elements are less.
- The right elements are greater than pivot.
- Now, divide the array into 2 sub arrays left part and right part.
- Take left partition apply quick sort.

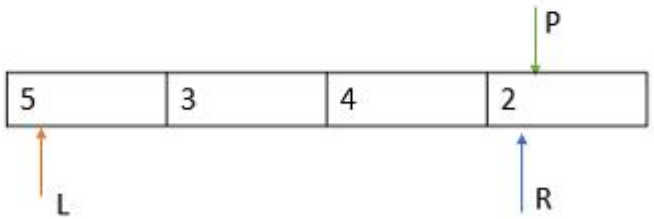
Quick sort

Case 1:

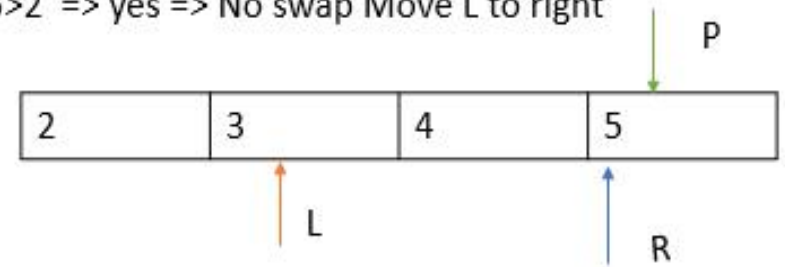


Is P<R => 5<2 {wrong} so, swap

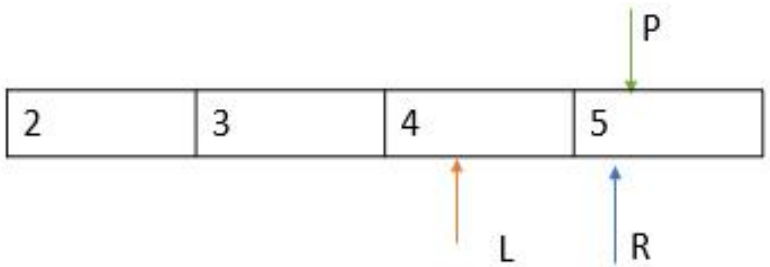
Case 2 :



Case 3: is P>L => 5>2 => yes => No swap Move L to right



Case 4: Is P> L => 5>3 => Yes => No swap => Move L to right



Now,

- The pivot is in fixed position.
- All the left elements are less and sorted
- The right elements are greater and are in sorted order.
- The final sorted list is combining two sub arrays is 2, 3, 4, 5, 6, 7

Quick sort

Time complexity

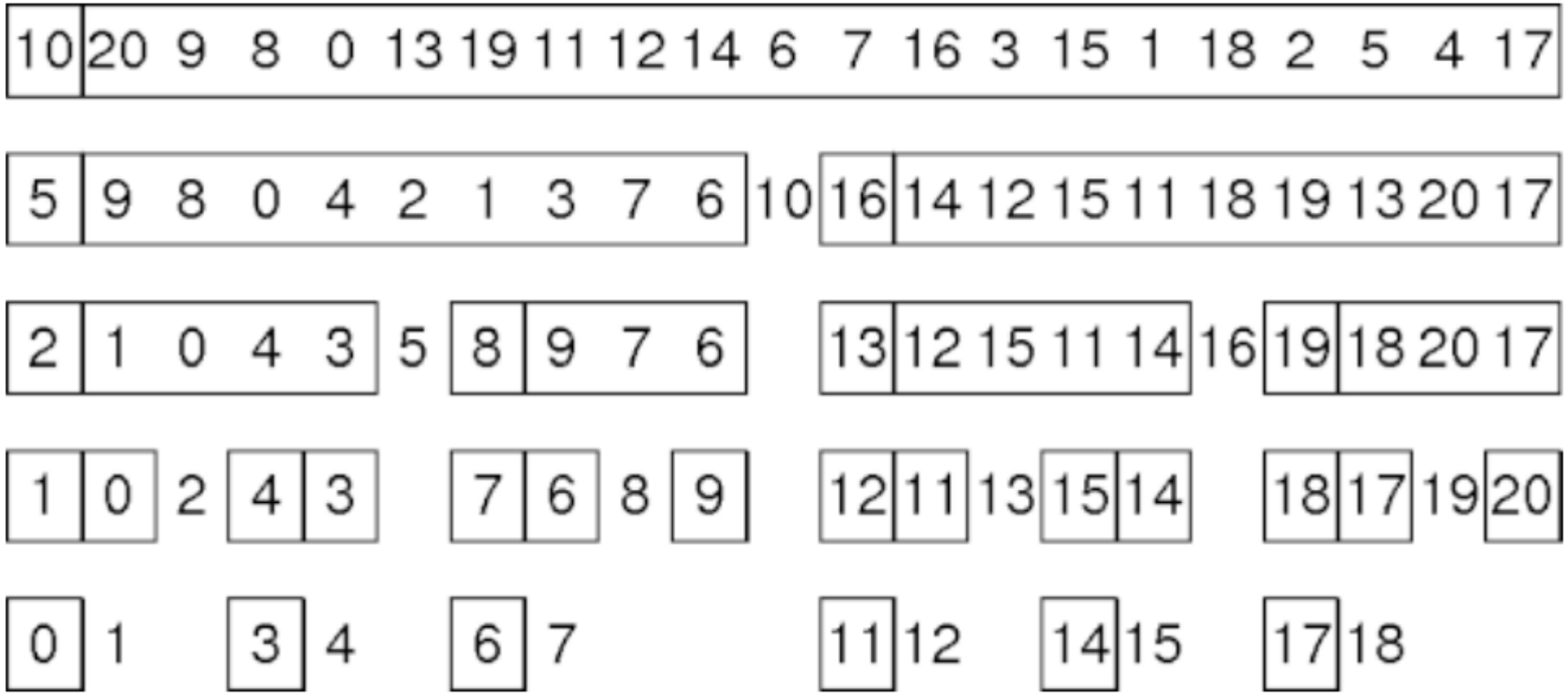
Best case: O(nlogn).

Average case:O(nlogn).

Worst case: O(n^2).

Quick sort

Simple example:



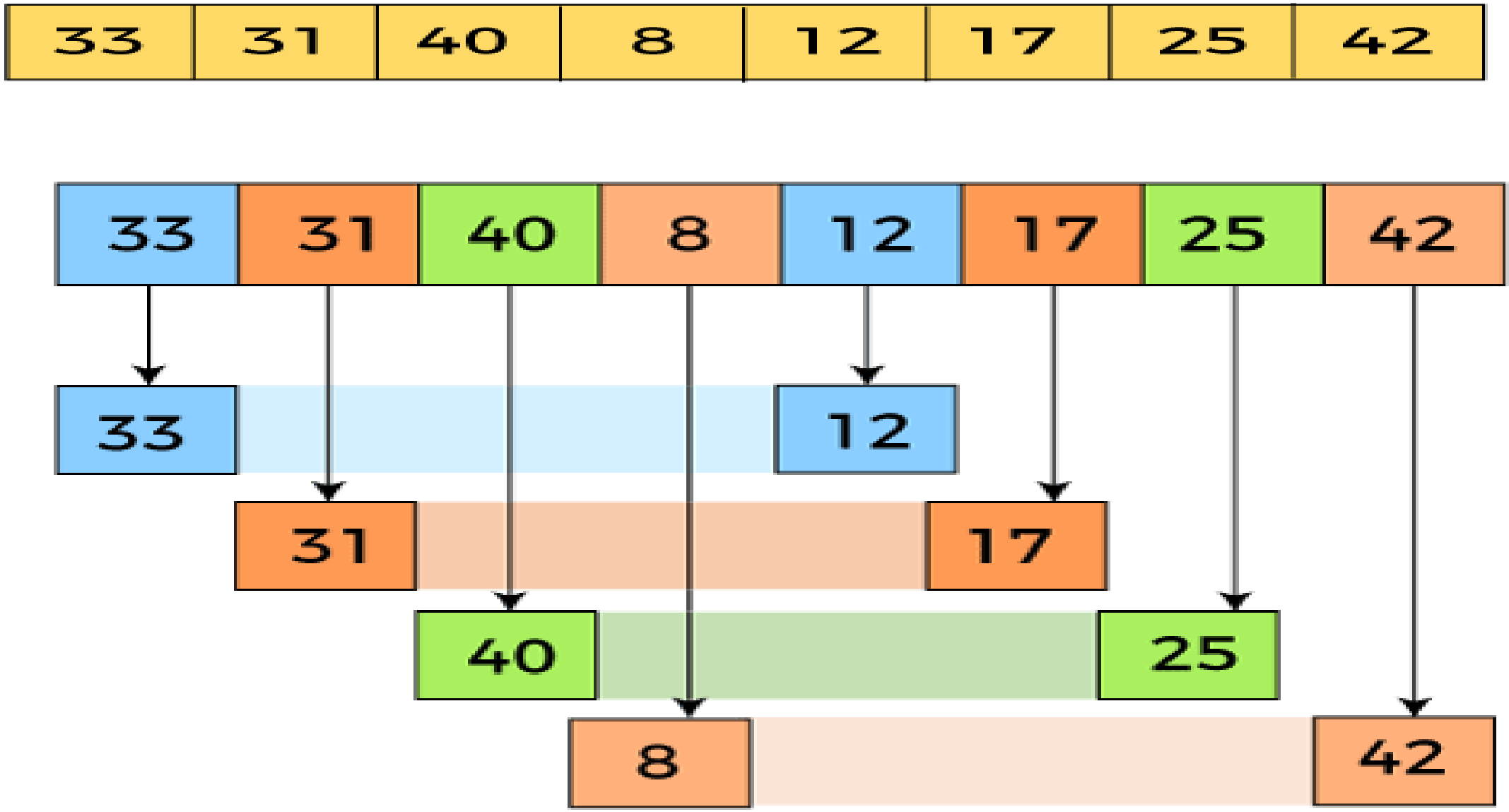
Shell sort

- The shell sort is an improved version of the straight insertion sort in which diminishing particiapating are used to sort the data.
- Compare elements that are distant apart rather than adjacent.
- We start by comparing elements that are at a certain distance aprt. So, if there are N elements then we start a value gap < N.

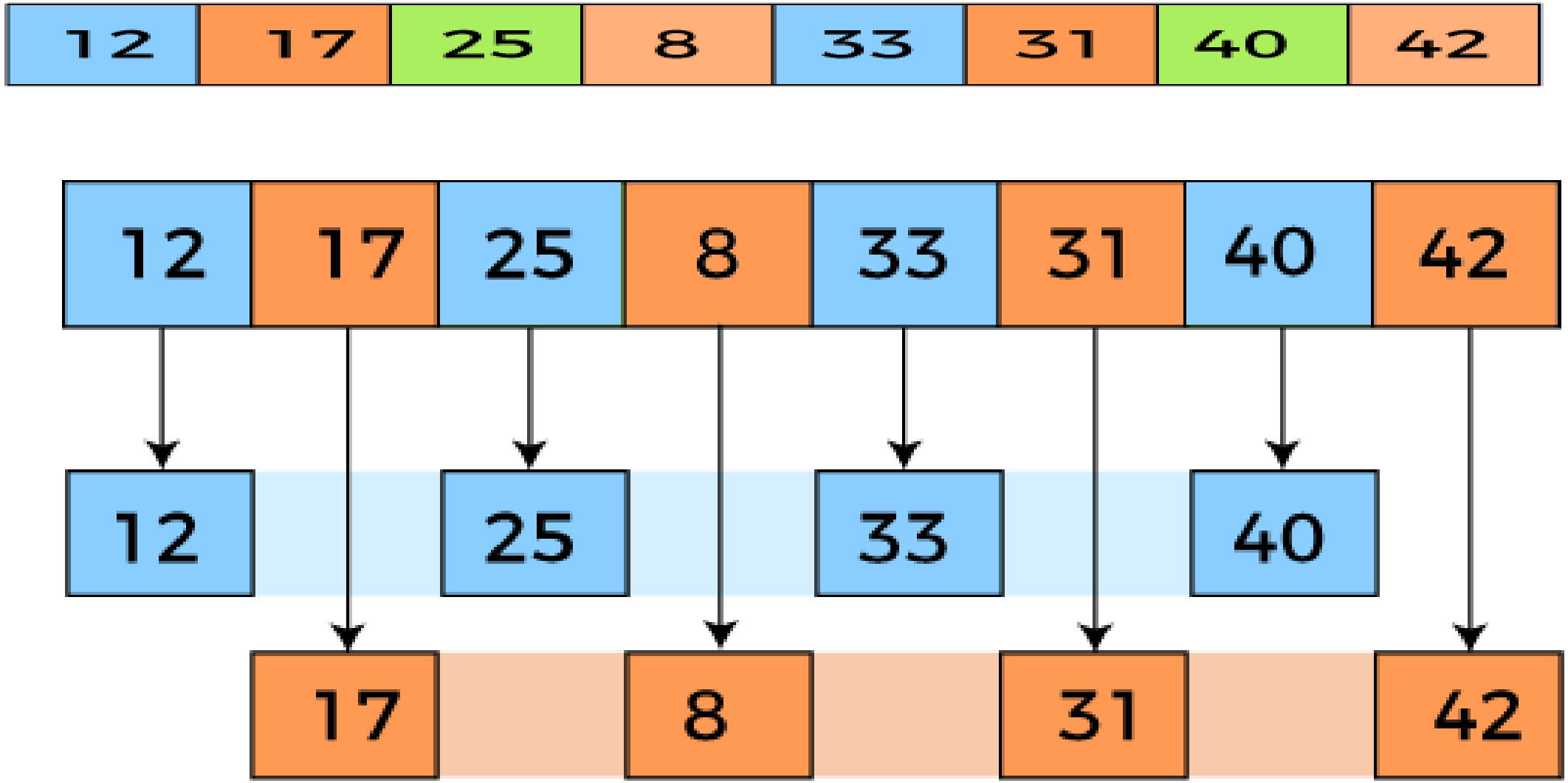
Formulae: gap = floor(N/2), where N = number of element in an array.

- In each pass we keep reducing the value of gap till reach the last pass when gap is 1.
- In the last pass shell sort is like insertion sort. Gap 1 = floor(N/2) and Gap2 = floor(Gap 1 /2).

Shell sort



Shell sort



Shell sort

12	8	25	17	33	31	40	42

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Shell sort

12	8	25	17	33	31	40	42

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Shell sort

- **Best Case Complexity -** It occurs when there is no sorting required, i.e., the array is already sorted. The best-case time complexity of Shell sort is **O(n*logn).**

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Shell sort is **O(n*logn).**

- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Shell sort is **$O(n^2)$.**

Radix Sort

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.

Example

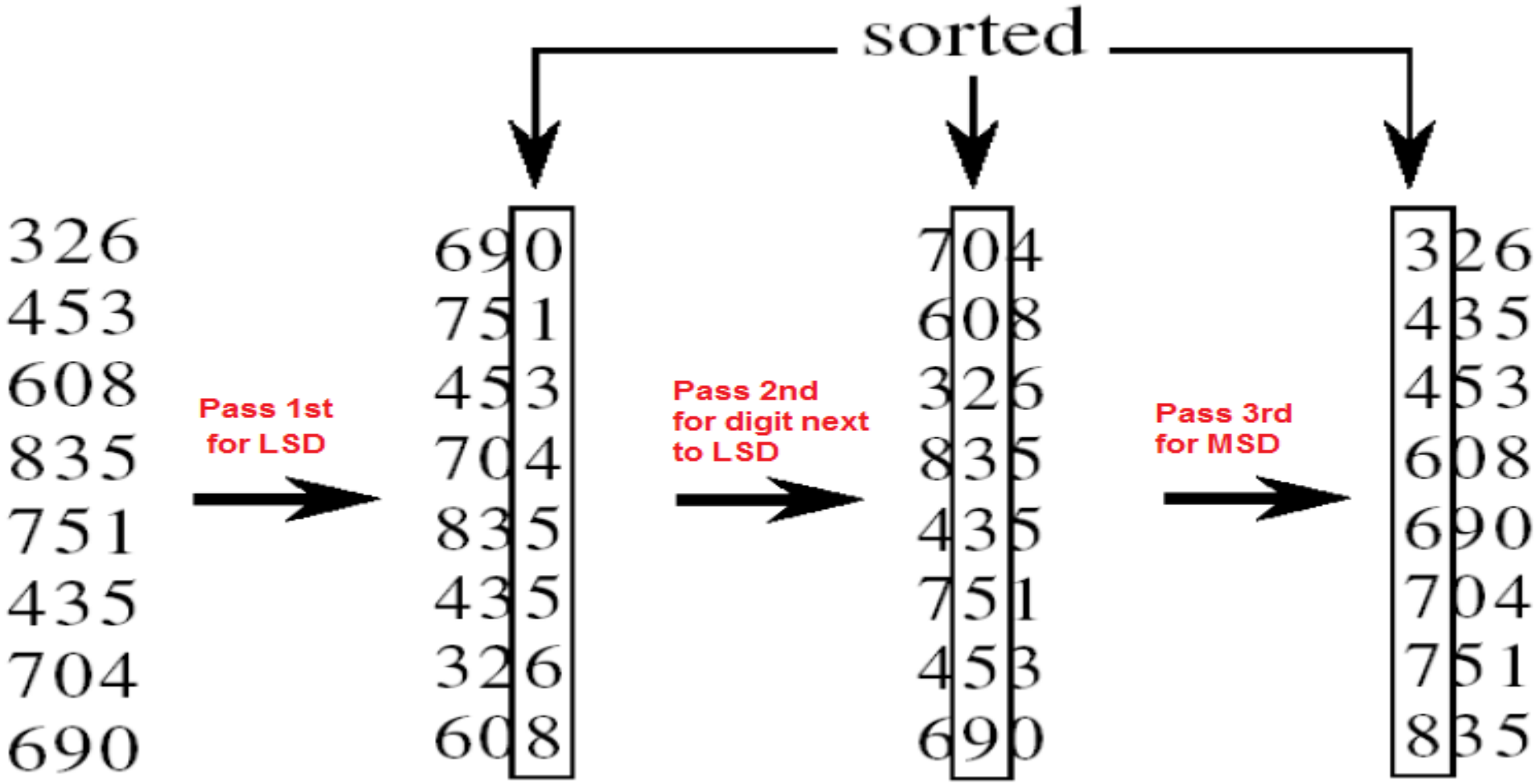
15,1,321,10,802,2,123,90,109,11

Solution:

Making each element of equal digits

015,001,321,010,802,002,123,090,109,011

Radix Sort


sorted
326
453
608
835
751
435
704
690
Pass 1st
for LSD
690
751
453
704
835
435
326
608
Pass 2nd
for digit next
to LSD
704
608
326
835
435
751
453
690
Pass 3rd
for MSD
326
435
453
608
690
704
751
835

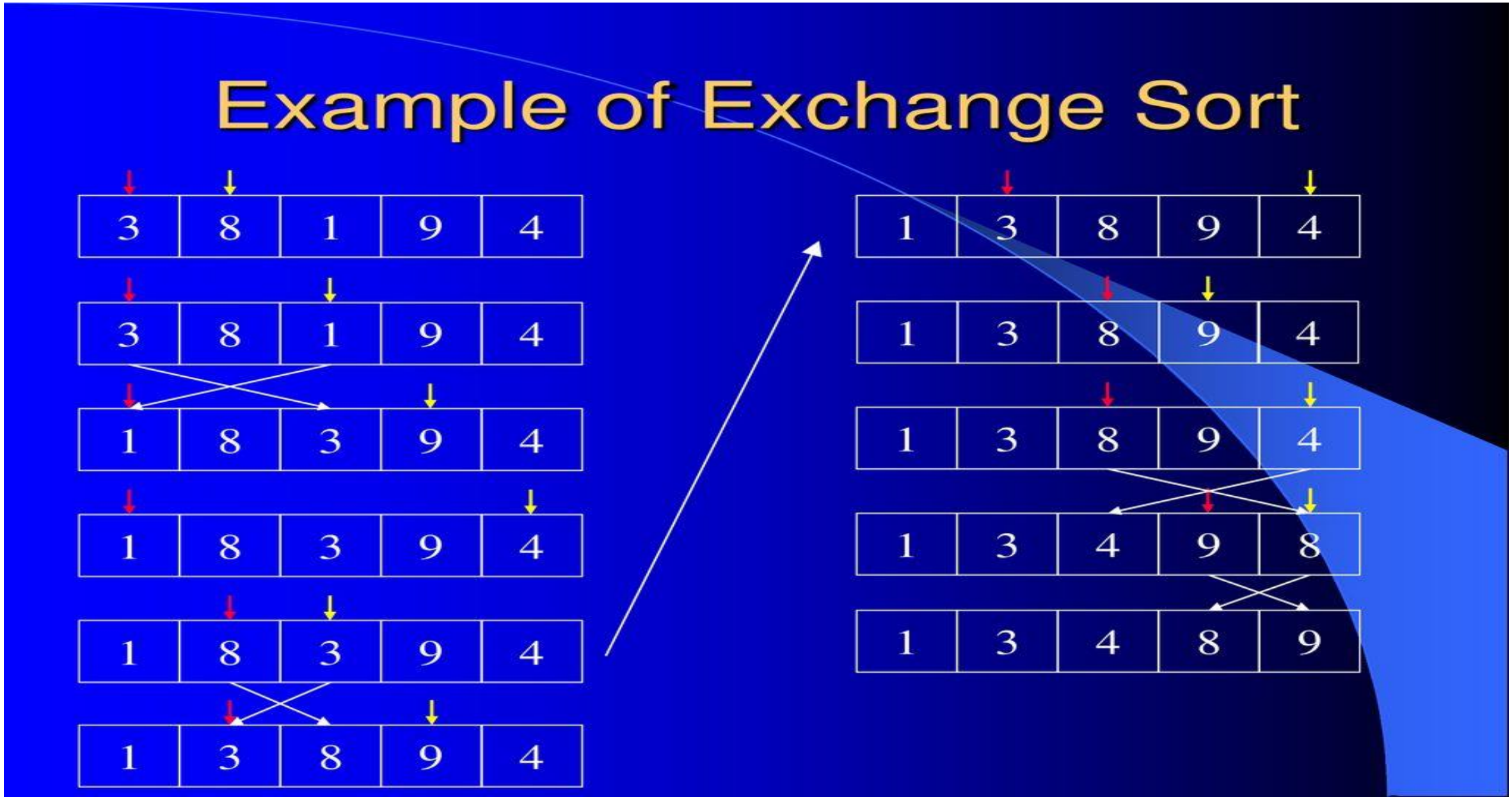
Exchange sort

- Exchange sort, also known as bubble sort, is a simple sorting algorithm that works by repeatedly iterating through the list of items to be sorted, comparing adjacent items and swapping them if they are in the wrong order.

- This process is repeated until no more swaps are needed, which indicates that the list is sorted.

- Exchange sort is called bubble sort because the smaller elements "bubble" to the top of the list as the sorting algorithm is applied.

Exchange sort



Exchange sort

```
#include<iostream>
using namespace std;
int main(void)
{
	int array[5];			// An array of
integers.
	int length = 5;			// Lenght of the array. }
	int i, j;
	int temp;
    //Some input
	for (i = 0; i < 5; i++)
	{
		cout << "Enter a number: ";
		cin >> array[i];
	}

	//Algorithm
	for(i = 0; i < (length -1); i++)
	{
		for (j=(i + 1); j < length; j++)
		{
			if (array[i] > array[j])
			{
				temp = array[i];
				array[i] = array[j];
				array[j] = temp;
			}
		}
}
//Some output
for (i = 0; i < 5; i++)
{
		cout << array[i] << endl;
}
```

Heap Sort

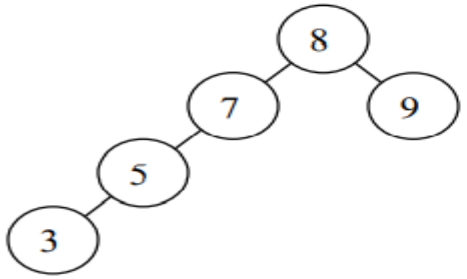
A binary heap or simply a heap is an almost complete binary tree with the key in each node, such that

i. All the leaves of the tree are on two adjacent levels.
ii. All the leaves on the lowest level occur to the left and all levels except possibly the lowest are filled.
iii. The key in the root is at least as large as the keys in its children (if any), and the left and rights subtrees are again heaps.

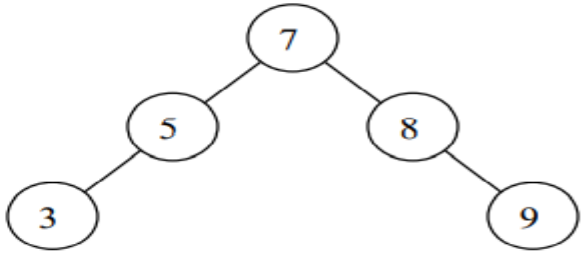
Above definition is of max heap (descending partially ordered tree).
For min heap the root has the smallest key

Heap Sort

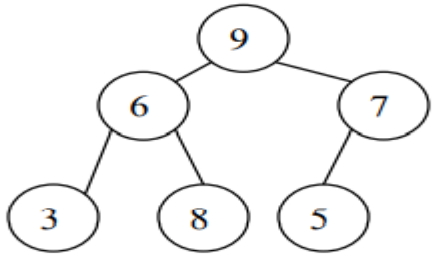
i. All the leaves of the tree are on two adjacent levels.
ii. All the leaves on the lowest level occur to the left and all levels except possibly the lowest are filled.
iii. The key in the root is at least as large as the keys in its children (if any), and the left and rights subtrees are again heaps.



This is not a Heap since it violates rule 1



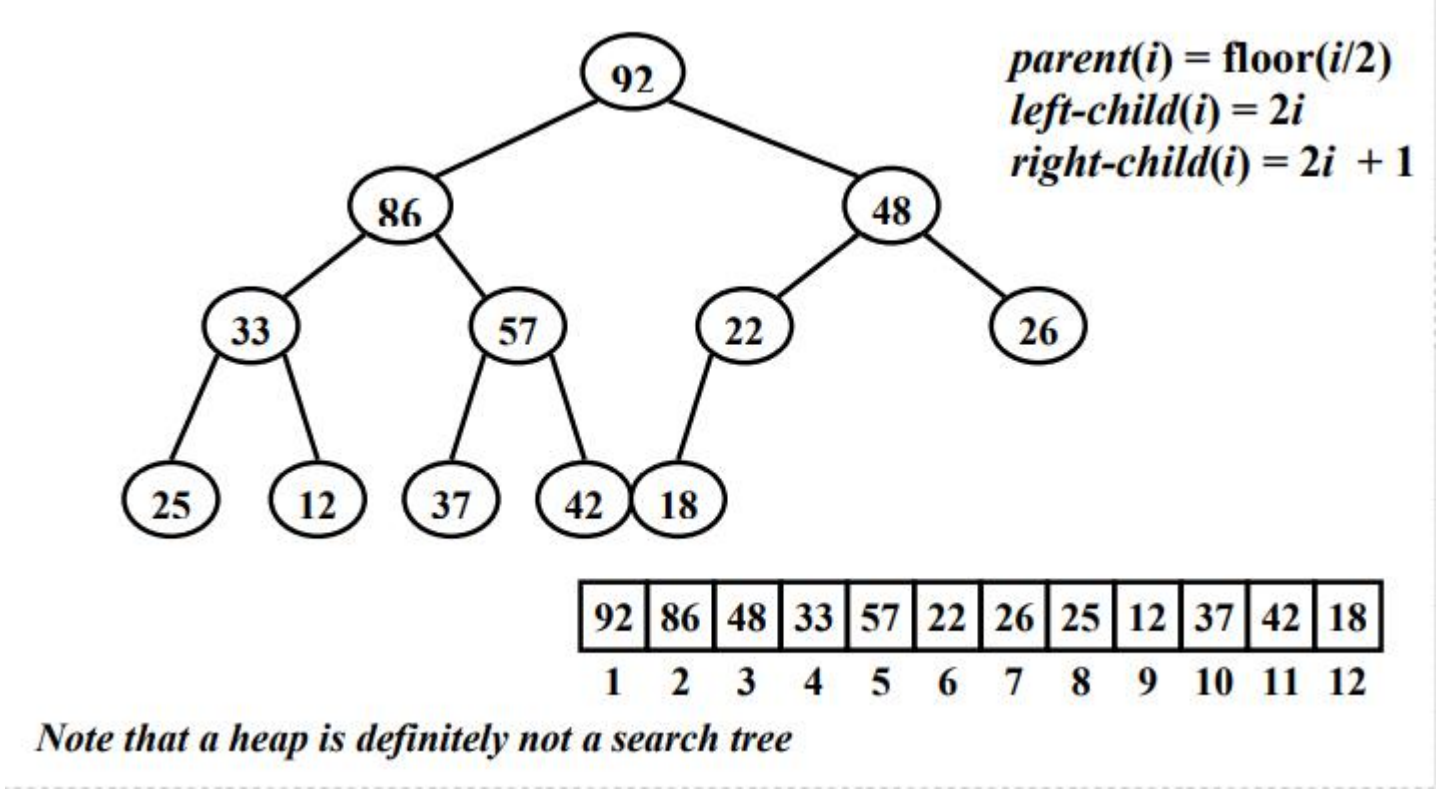
This is not a Heap since it violates rule 2



This is not a Heap since it violates rule 3

Heap sort

- A heap is defined as an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value of the father.
- It can be sequentially represented as

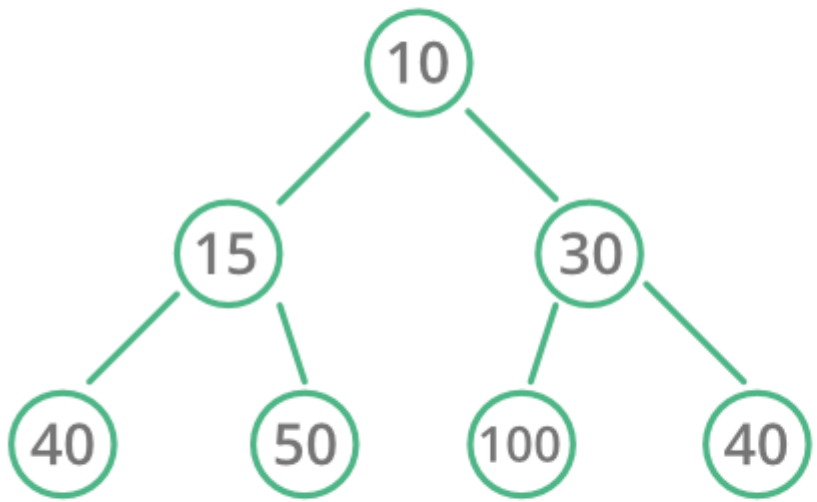


Heap sort

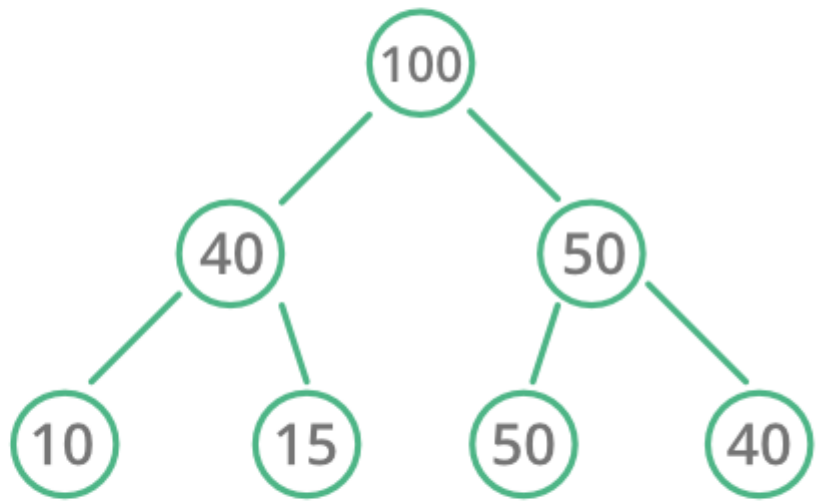
- The root of the binary tree (i.e., the first array element) holds the largest key in the heap. This type of heap is usually called descending heap or **max heap**.

- We can also define an ascending heap as an almost complete binary tree in which the value of each node is greater than or equal to the value of its father. This root node has the smallest element of the heap. This type of heap is also called **min heap**.

Heap Data Structure



Min Heap

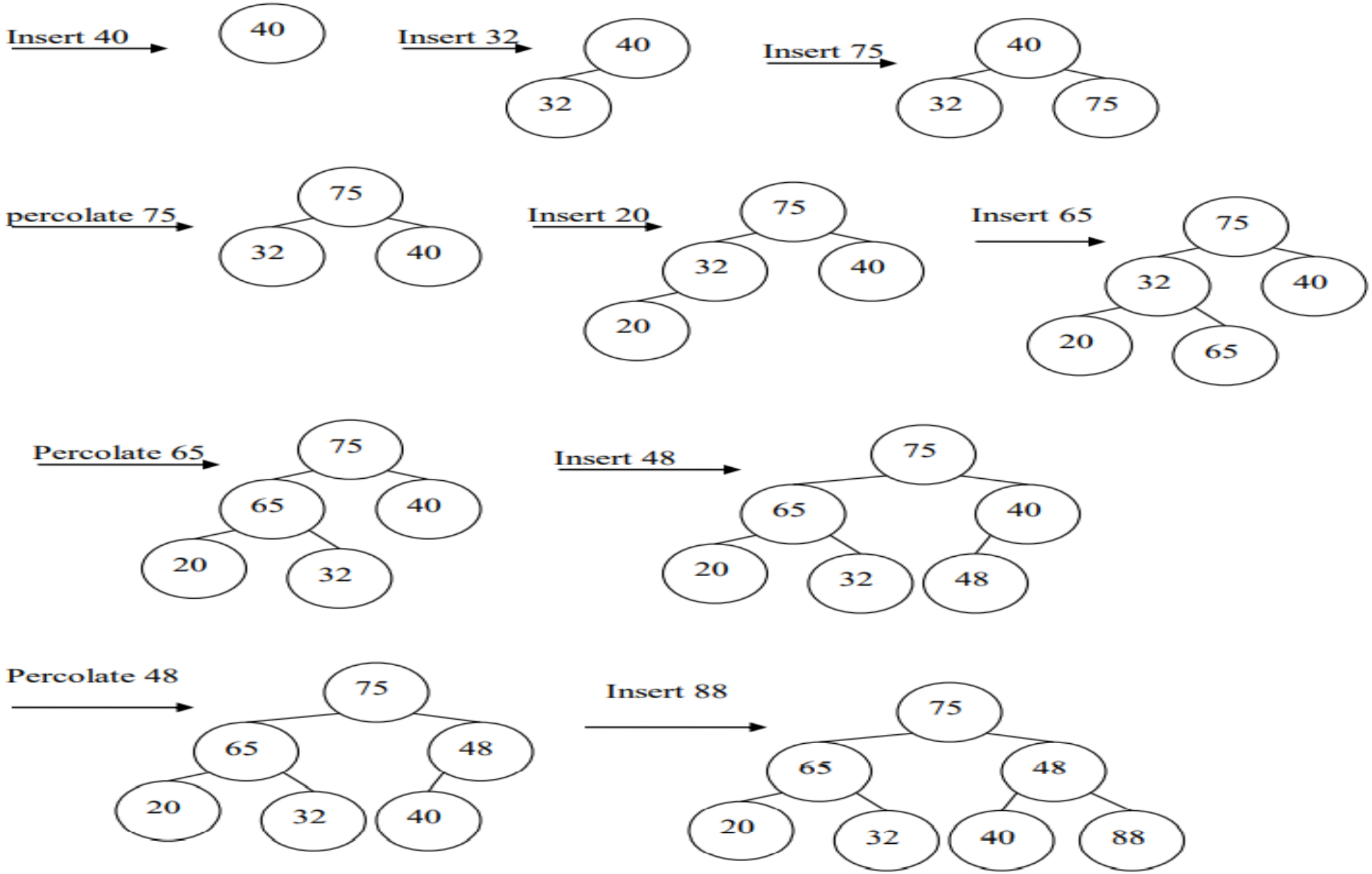


Max Heap

Creating a Heap and Insertion:

- A Binary tree with only one node satisfies the properties of a heap.
- To insert a new element in to the heap, we create a hole in the next available location, since otherwise the tree will not be complete tree.
- If new item can be placed in the hole without violating heap order, then we do insert there. Otherwise, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up towards the root.
- The process is continued until the new item can be placed in the hole.
- This strategy is known as a percolate up.
- The new element is percolated up the heap until the correct location is formed.

Eg: Inserting following numbers in the max Heap: 40, 32, 75, 20, 65, 48, 88.



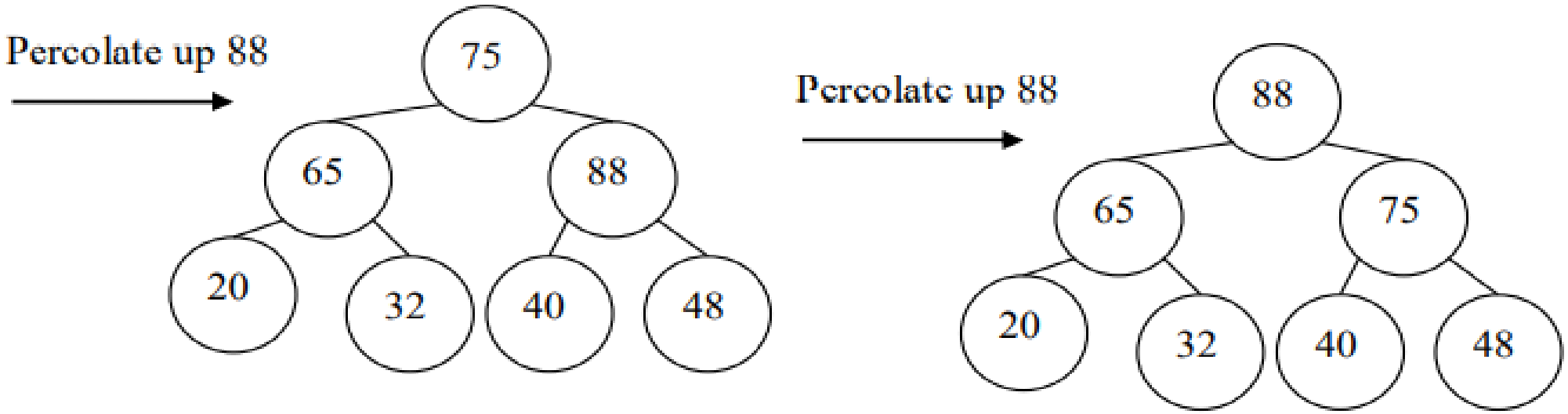
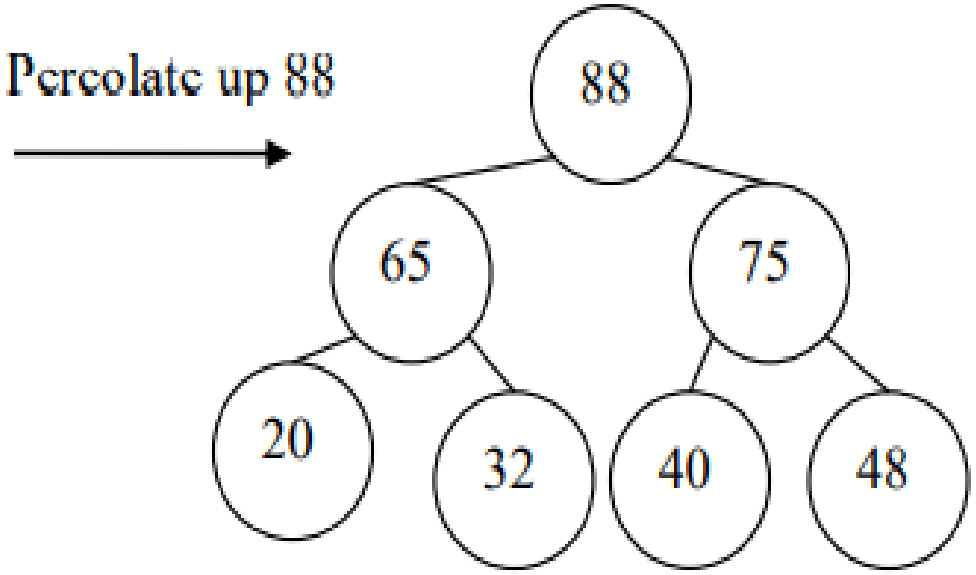


Fig: Creating Max Heap

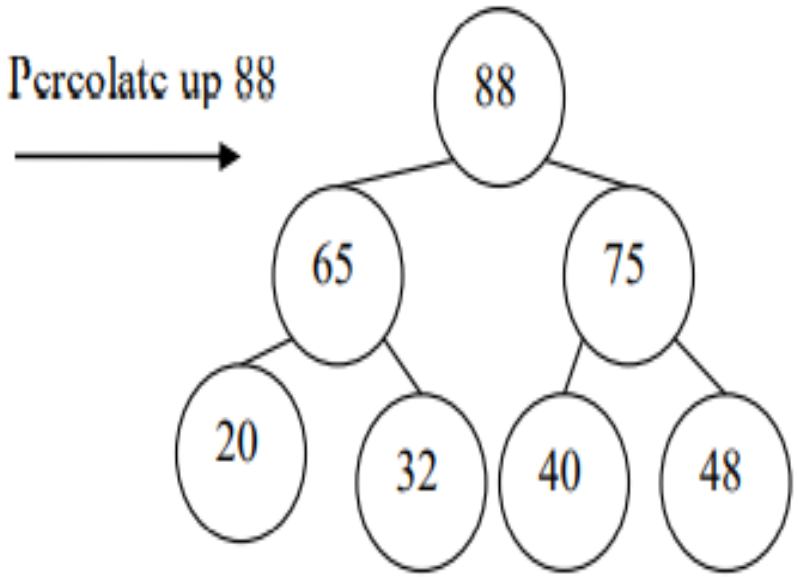
Deletions in Heap

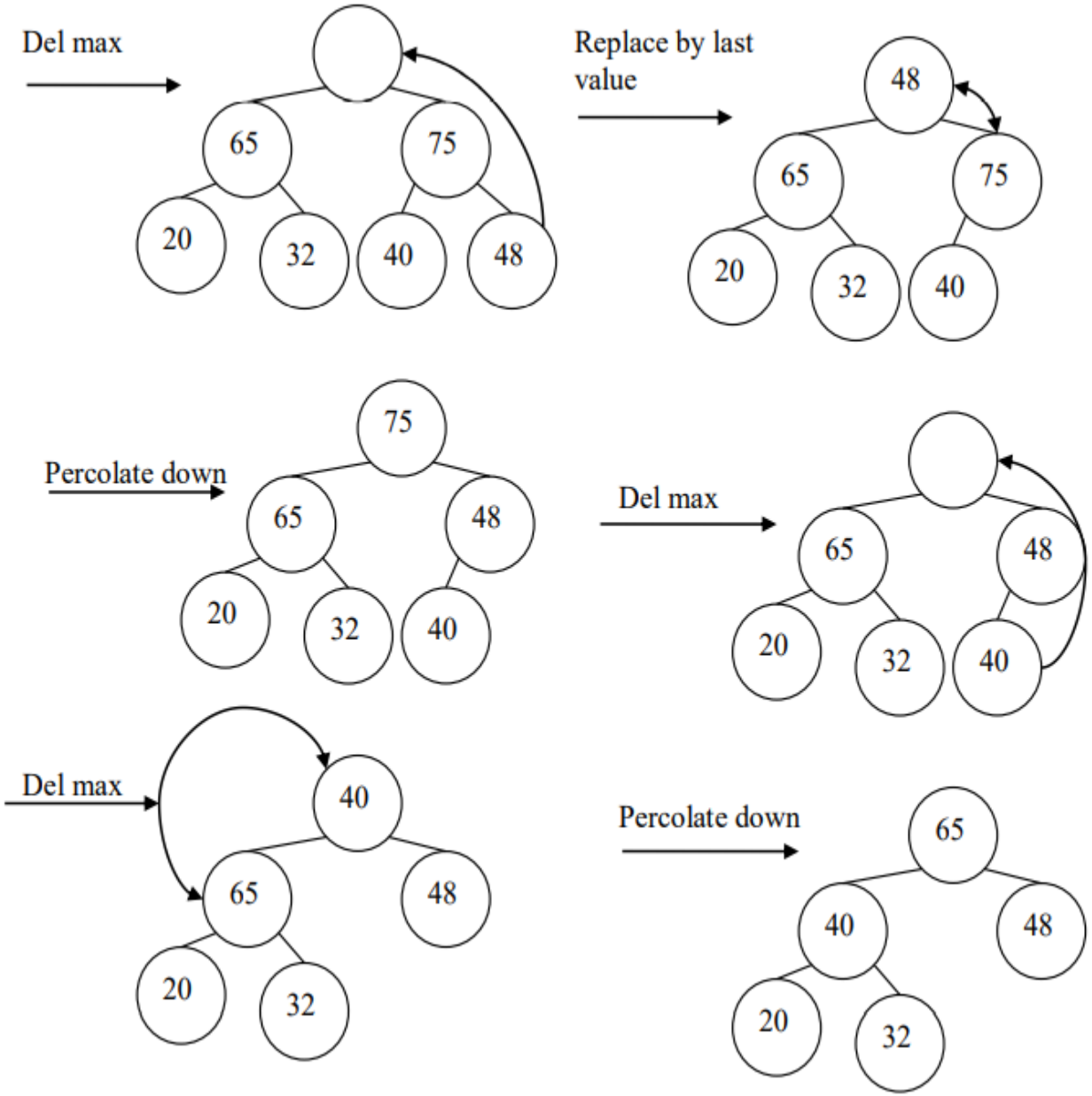
- Deleting in heap is always applied for the item at the root (i.e minimum in min heap and maximum in max heap).
- When the root item is removed, a hole is created at the root.
- Since, the heap now becomes one smaller, it follows that the last element ‘x’ in the heap is placed in the hole.
- If the heap property is violated, we slide the smaller (larger) hole’s children into the hole in the case of min (max) heap.
- This process of pushing up the smaller item is continued until the item ‘x’ finds its proper place.

For deleting in heap lets take above example:



Deletions in Heap





Deletions in Heap

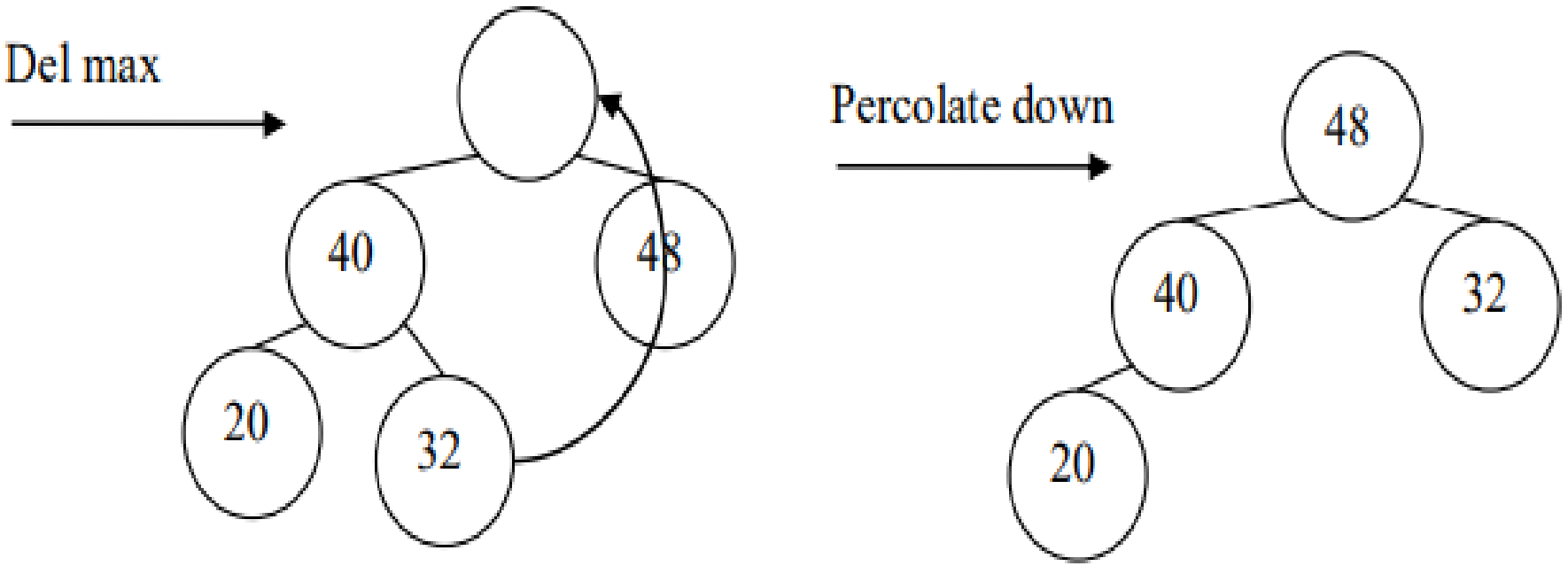


Fig: Deletion in Max Heap

Heap Tree Algorithm

Let H be a heap with n elements stored in the array HA. Data is the item of the node to be removed. Last gives the information about last node of H. The LOC, left, right gives the location of Last and its left and right children as the Last rearranges in the appropriate place in the tree.

1. Input n elements in the heap H
2. Data = HA[0]; last = HA[n-1] and n = n – 1
3. LOC = 0, left = 2*LOC+1 and right = 2*LOC+2
4. Repeat the steps 5, 6 and 7 while (right <= n)
5. If (last >= HA[left]) and (last >= HA[right])
   (a) HA[LOC] = last
   (b) Exit
6. a)If (HA[right] <= HA[left])
   (i) HA[LOC] = HA[left]
   (ii) LOC = left

   (b) Else
   (i) HA[LOC] = HA[right]
   (ii) LOC = right
7. left = 2 * LOC; right = left +1
8. If (left = n) and (last < HA[left])

   (a) LOC = left
9. HA [LOC] = last

Sorting Algorithm	Time Complexity(Best)	Time Complexity (Worst)	Time Complexity (Average)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick Sort	O(nlogn)	O(n^2)	O(nlogn)
Shell Sort	O(nlogn)	O(n^2)	O(nlogn)
Radix Sort	O(n+k)	O(n+k)	O(n+k)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)

Thank you!!!!!