DATA STRUCTURES

Sorting

Lecture No. 11,12,13

SORTING

- A process that organizes a collection of data into either ascending or descending order.
- Can be used as a first step for searching the data.
- Binary Search required a sorted array.

SORTING ALGORITHMS

- Selection Sort
- Insertion Sort
- Bubble Sort
- Quick Sort
- Merge Sort
- Heap Sort

SELECTION SORT

- It is simple and easy to implement
- It is inefficient for large list, usually used to sort lists of no more than 1000 items
- In array of n elements, n-1 iterations are required to sort the array

SELECTION SORT

- Select the smallest value from the list.
- Bring it to the first location of the list.
- Find the next value and repeat the process by swapping the locations.

SELECTION SORT

- Suppose the name of the array is A and it has four elements with the following values:

4	19	1	3

- To sort this array in ascending order, n-1, i.e. three iterations will be required.

SELECTION SORT

4	19	1	3

- **Iteration-1**

  The array is scanned starting from the first to the last element and the element that has the smallest value is selected. The smallest value is 1 at location 3. The address of element that has the smallest value is noted and the selected value is interchanged with the first element i.e.

  A[1] and A[3] are swapped

1	19	4	3

SELECTION SORT

1	19	4	3

- **Iteration-2**

  The array is scanned starting from the second to the last element and the element that has the smallest value is selected. The smallest value is 3 at location 4. The address of element that has the smallest value is noted. The selected value is interchanged with the second element i.e.

  A[2] and A[4] are swapped

1	3	4	19

SELECTION SORT

1	3	4	19

- **Iteration-3**

  The array is scanned starting from the third to the last element and the element that has the smallest value is selected. The smallest value is 4 at location 3. The address of element that has the smallest value is noted. The selected value is interchanged with the third element i.e.

  A[3] and A[3] are swapped

1	3	4	19

ANOTHER EXAMPLE: SELECTION SORT

- 26 33 43 100 46 88 52 17 53 77
- 17 | 33 43 100 46 88 52 26 53 77
- 17 26 | 43 100 46 88 52 33 53 77
- 17 26 33 | 100 46 88 52 43 53 77
- 17 26 33 43 | 46 88 52 100 53 77
- 17 26 33 43 46 | 88 52 100 53 77
- 17 26 33 43 46 52 | 88 100 53 77
- 17 26 33 43 46 52 53 | 100 88 77
- 17 26 33 43 46 52 53 77 | 88 100
- 17 26 33 43 46 52 53 77 88 | 100

SELECTION SORT

```
void selectionSort(int numbers[ ], int array_size)
{
    int i, j;
    int min, temp;
    for (i = 0; i < array_size-1; i++)
    {
        min = i;
        for (j = i+1; j < array_size; j++)
        {
            if (numbers[j] < numbers[min])
                min = j;
        }
        temp = numbers[i];
        numbers[i] = numbers[min];
        numbers[min] = temp;
    }
}
```

INSERTION SORT

- It is simple as the bubble sort but it is almost twice as efficient as the bubble sort
- It is relatively simple and easy to implement
- It is inefficient for large lists

INSERTION SORT

- In insertion sorting, the list or array is scanned from the beginning to the end
- In each iteration, one element is inserted into its correct position relative to the previously sorted elements of the list
- The array elements are not swapped or interchanged
- They are shifted towards the right of the list to make room for the new element to be inserted

INSERTION SORT

- Given an unsorted list.
- Partition the list into two regions: sorted & unsorted.
- At each step, take the first item from unsorted and place it into its correct position.
- Also requires to shift the remaining items to make a room for the inserted item.

INSERTION SORT

- Suppose the name of the array is A and it has six elements with the following values:

16	17	2	8	18	1

- To sort this array in ascending order, six iterations will be required.

INSERTION SORT

16	17	2	8	18	1

- **Iteration-1**

A[1] is compared with itself and it is not shifted. The array A remains the same

16	17	2	8	18	1

INSERTION SORT

16	17	2	8	18	1

- **Iteration-2**

  All data of elements on left of A[2] that are greater than A[2] are shifted one position to the right to make room for A[2] to insert its data into the correct location.

  There is only one element with value 16 to the left of A[2]. Thus no shifting takes place because 16 is less than 17. So A[1] and A[2] are in correct position relative to each other. The array A remains same

16	17	2	8	18	1

INSERTION SORT

16	17	2	8	18	1

- **Iteration-3**

  All data of elements on left of A[3] that are greater than A[3] are shifted one position to the right to make room for A[3] to insert its data into the correct location.

  There is two elements of left side of A[3] and both are greater than A[3]. Thus shift data A[1] & A[2] one position to right and insert the value of A[3] at A[1]. The array A after shifting and inserting value is:

2	16	17	8	18	1

INSERTION SORT

2	16	17	8	18	1

- **Iteration-4**

  All data of elements on left of A[4] that are greater than A[4] are shifted one position to the right to make room for A[4] to insert its data into the correct location.

  There is three elements of left side of A[4] and A[2] & A[3] are greater than A[4]. Thus shift data A[2] & A[3] one position to right and insert the value of A[4] at A[2]. The array A after shifting and inserting value is:

2	8	16	17	18	1

INSERTION SORT

2	8	16	17	18	1

- **Iteration-5**

  All data of elements on left of A[5] that are greater than A[5] are shifted one position to the right to make room for A[5] to insert its data into the correct location.

  There is four elements of left side of A[5] and all are less than A[5]. Thus no shifting & insertion takes place. The array A remains same:

2	8	16	17	18	1

INSERTION SORT

2	8	16	17	18	1

- **Iteration-6**

  All data of elements on left of A[6] that are greater than A[6] are shifted one position to the right to make room for A[6] to insert its data into the correct location.

  There is five elements of left side of A[6] and all are greater than A[6]. Thus shift data of each element from A[1] to A[5] one position to right and insert the value of A[6] at A[1]. The array A after shifting and inserting value is:

1	2	8	16	17	18

ALGORITHM – INSERTION SORT

InsertionSort()

Algorithm to sort an array A consisting of N elements in ascending order

1. Start
2. Repeat step 3 to 8 For C = 2 to N
3. Set Temp = A[C]
4. Set L = C
5. Repeat Step 6 to 7 While (L>1 and Temp<=A[L-1])
6. Set A[L] = A[L-1]
7. L = L – 1
8. Set A[L] = Temp
9. Exit

INSERTION SORT CONT.....

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

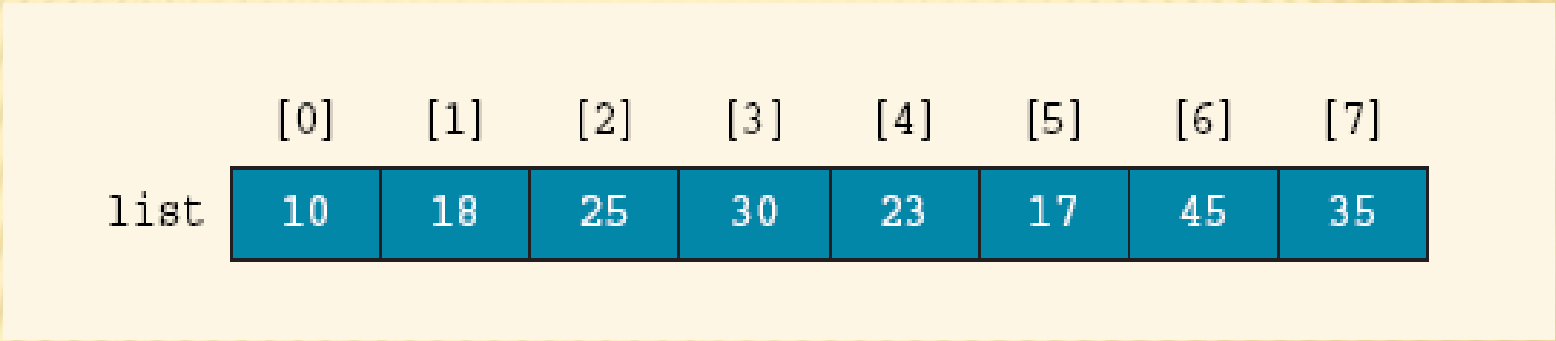


Figure 6: Array *list* to be sorted

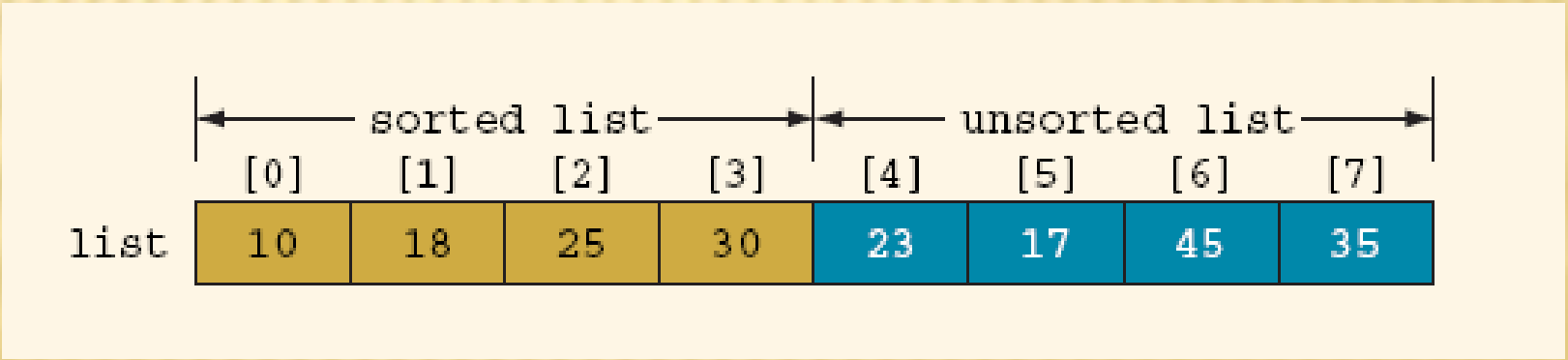


Figure 7: Sorted and unsorted portions of the array *list*

INSERTION SORT ALGORITHM (CONT'D)

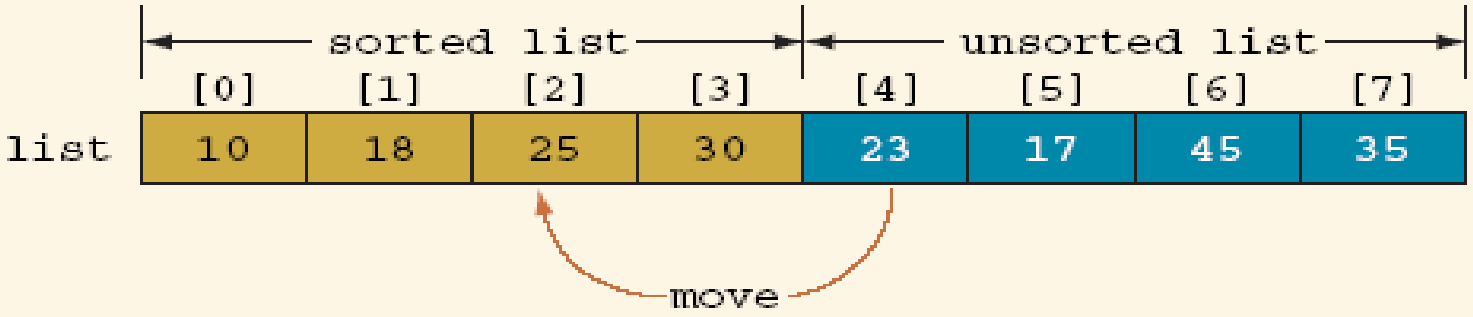


Figure 8: Move *list[4]* into *list[2]*

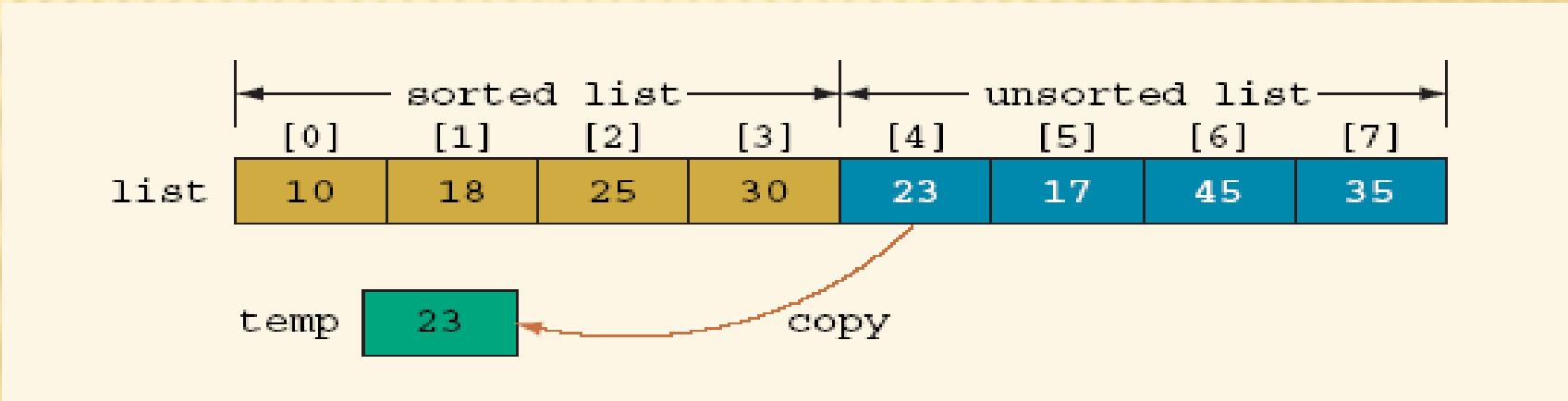


Figure 9: Copy *list[4]* into *temp*

INSERTION SORT ALGORITHM (CONT'D)

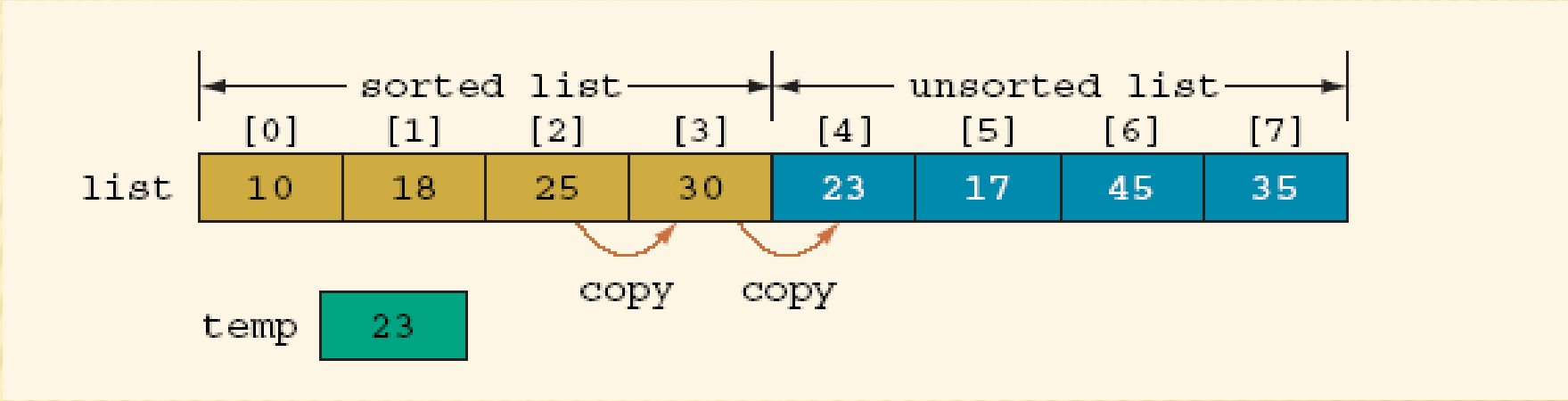


Figure 10: Array *list* before copying *list[3]* into *list[4]*, then *list[2]* into *list[3]*

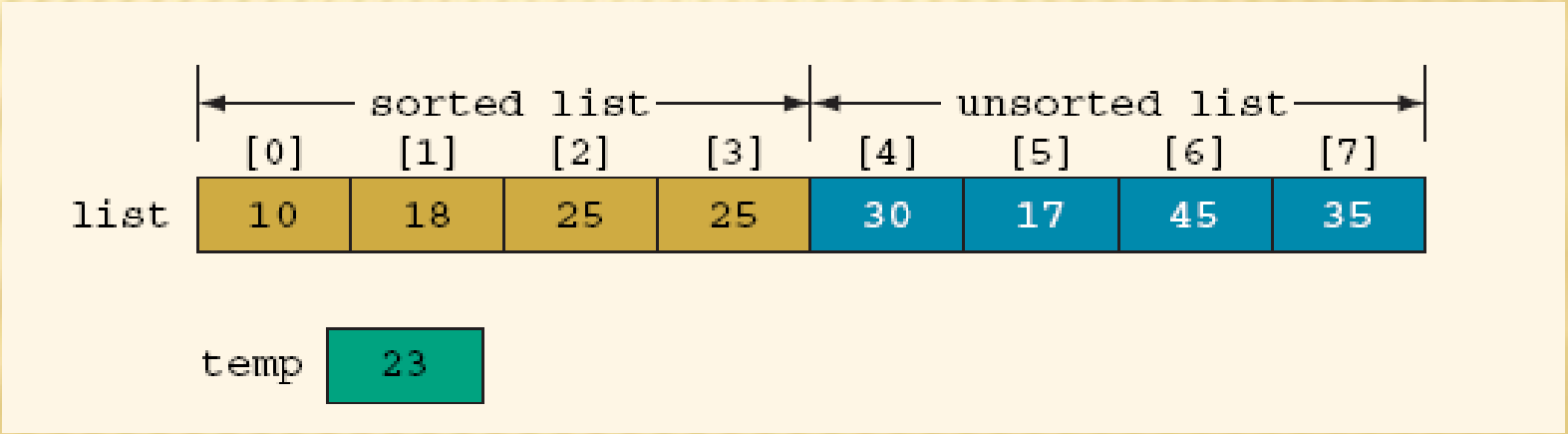


Figure 11: Array *list* after copying *list[3]* into *list[4]*, and then *list[2]* into *list[3]*

INSERTION SORT ALGORITHM (CONT'D)

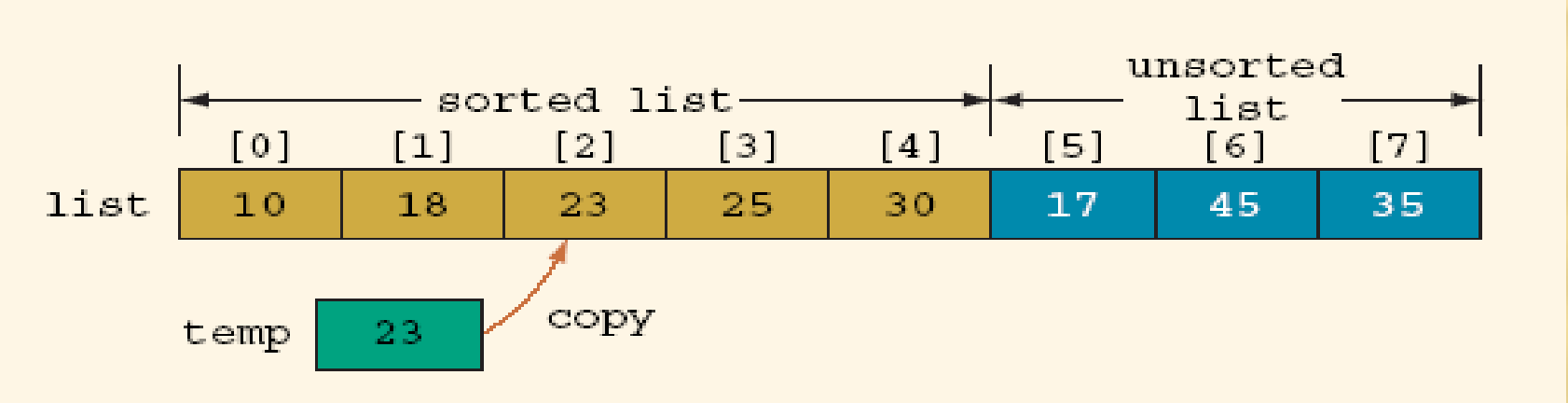
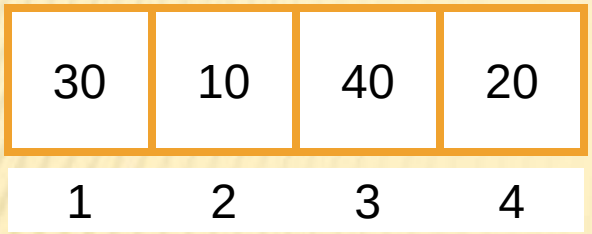


Figure 12: Array *list* after copying *temp* into *list[2]*

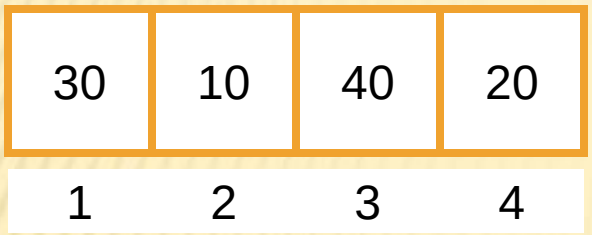
AN EXAMPLE: INSERTION SORT



$i = \varnothing \quad j = \varnothing \quad key = \varnothing$

$A[j] = \varnothing \qquad A[j+1] = \varnothing$

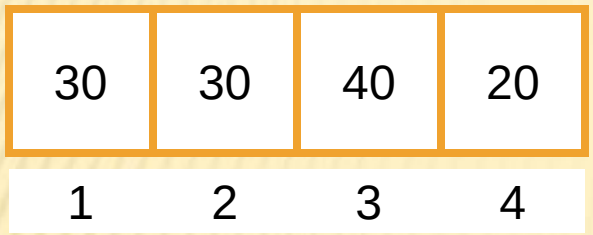
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 2 j = 1 key = 10

A[j] = 30 A[j+1] = 10

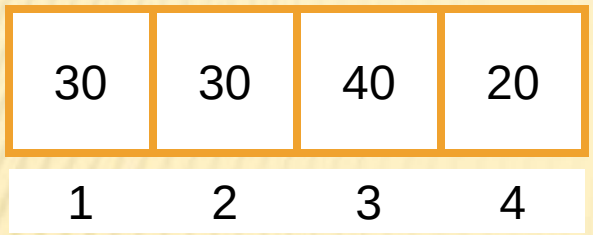
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 2 j = 1 key = 10

A[j] = 30 A[j+1] = 30

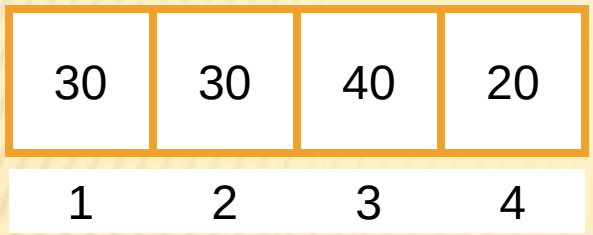
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 2 j = 1 key = 10

A[j] = 30 A[j+1] = 30

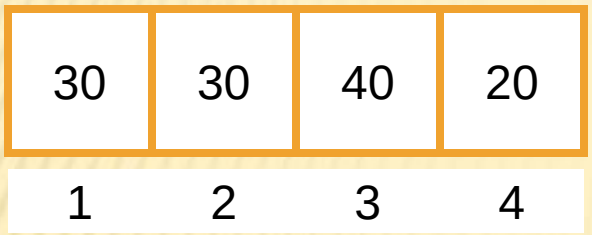
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 2 j = 0 key = 10

A[j] = ∅ A[j+1] = 30

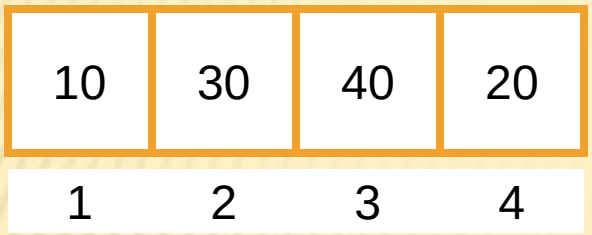
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 2 j = 0 key = 10

A[j] = ∅ A[j+1] = 30

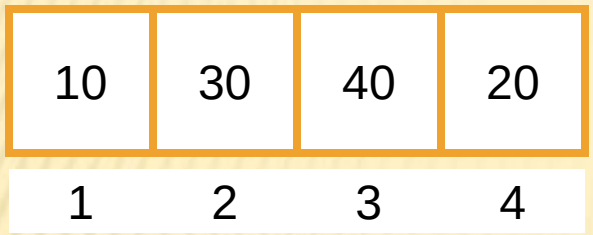
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 2 j = 0 key = 10

A[j] = ∅ A[j+1] = 10

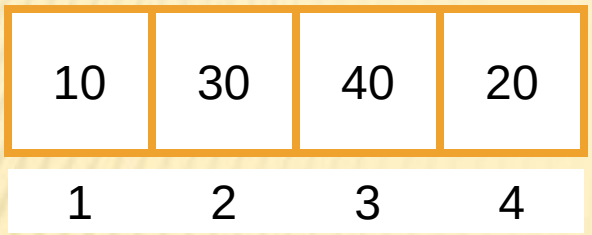
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 3 j = 0 key = 10

A[j] = ∅ A[j+1] = 10

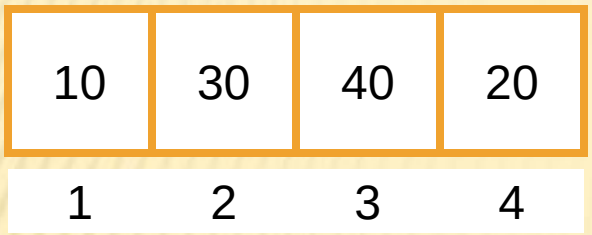
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 3 j = 0 key = 40

A[j] = ∅ A[j+1] = 10

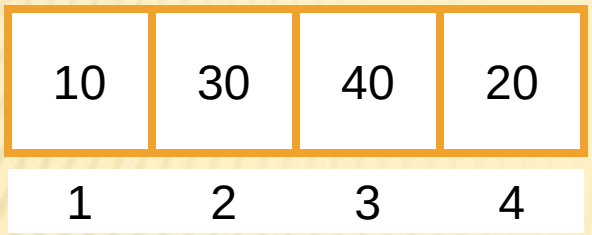
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 3 j = 0 key = 40

A[j] = ∅ A[j+1] = 10

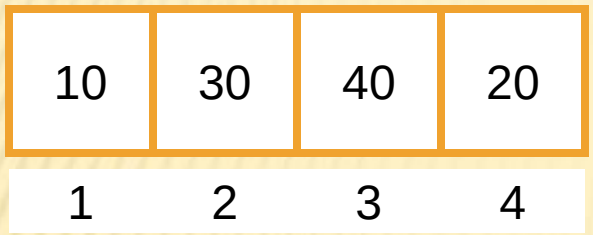
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 3 j = 2 key = 40

A[j] = 30 A[j+1] = 40

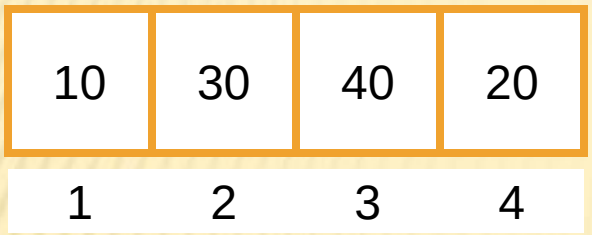
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 3 j = 2 key = 40

A[j] = 30 A[j+1] = 40

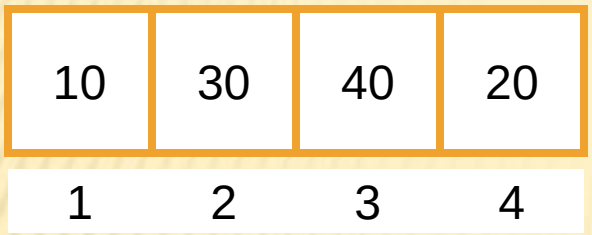
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 3 j = 2 key = 40

A[j] = 30 A[j+1] = 40

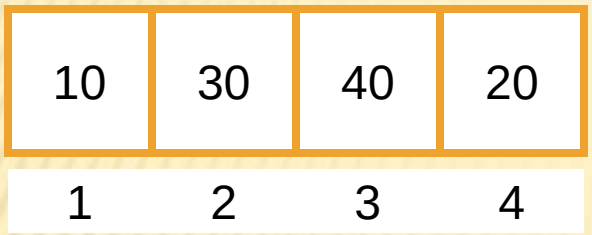
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 2 key = 40

A[j] = 30 A[j+1] = 40

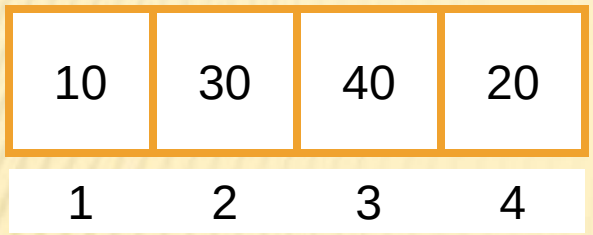
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 40

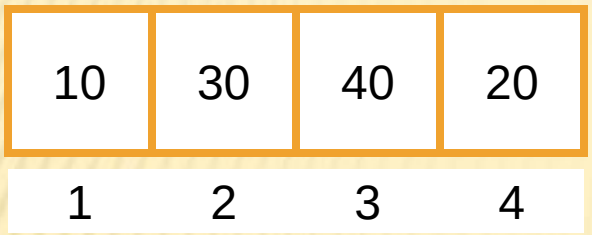
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 40

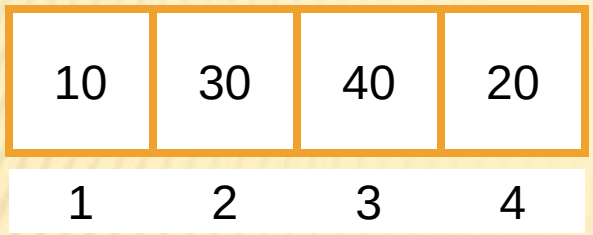
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 3 key = 20

A[j] = 40 A[j+1] = 20

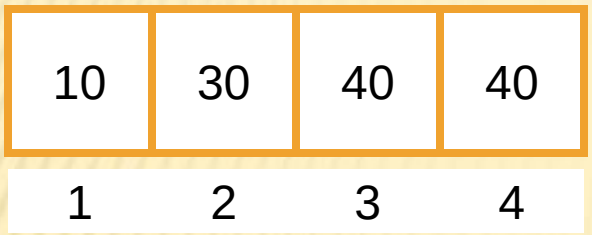
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 3 key = 20

A[j] = 40 A[j+1] = 20

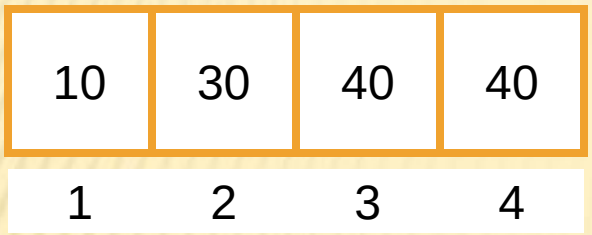
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 3 key = 20

A[j] = 40 A[j+1] = 40

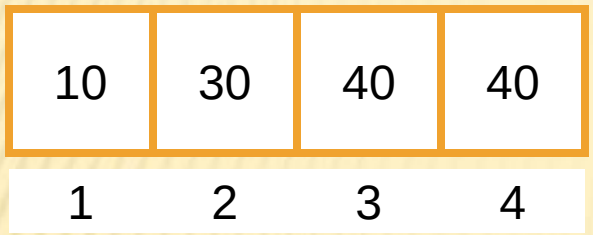
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 3 key = 20

A[j] = 40 A[j+1] = 40

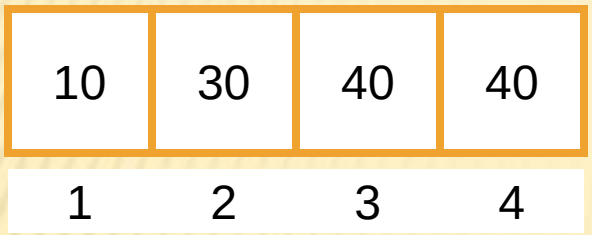
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 3 key = 20

A[j] = 40 A[j+1] = 40

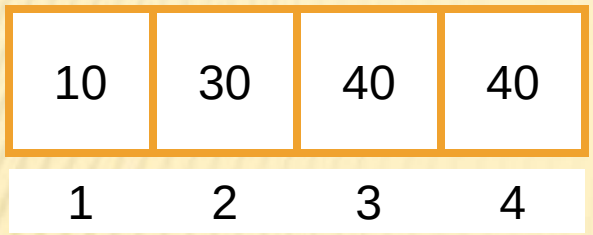
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 40

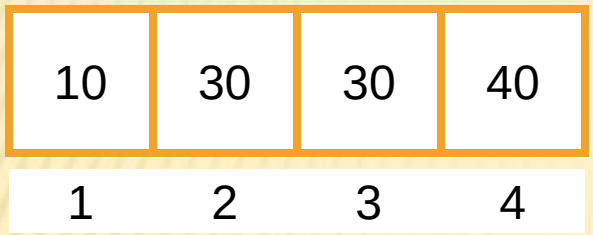
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 40

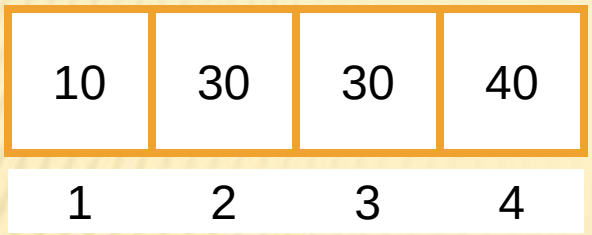
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 30

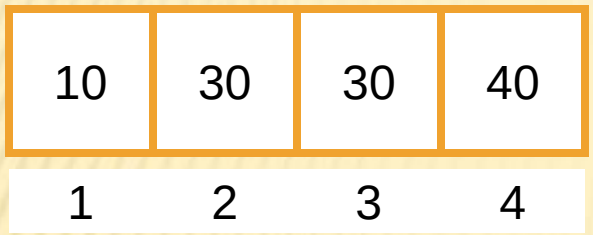
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 30

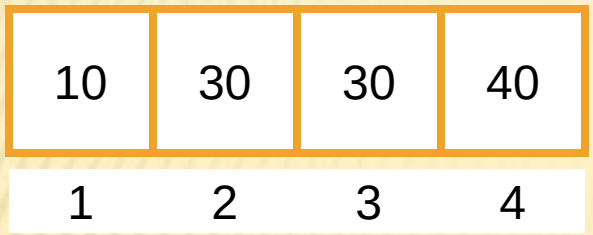
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 1 key = 20

A[j] = 10 A[j+1] = 30

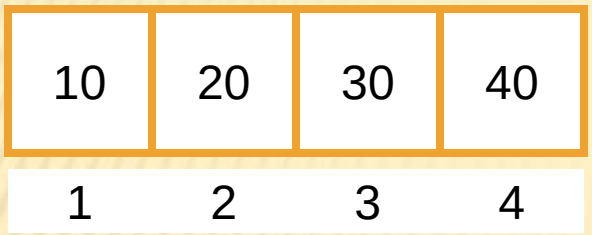
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 1 key = 20

A[j] = 10 A[j+1] = 30

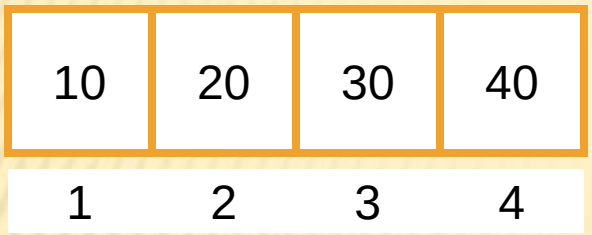
```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 1 key = 20

A[j] = 10 A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```



i = 4 j = 1 key = 20

A[j] = 10 A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

Done!

EXAMPLE: 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 ALGORITHM

```
void insertionSort(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

- It is the oldest and simplest method and can be easily implemented
- It is also the slowest and considered to be the most inefficient sorting algorithm
- It works by comparing each item in the list with the item next to it, and swapping them if required
- It is used only a small amount of data

BUBBLE SORT

- To sort data in an array of n elements, n-1 iterations are required
- Following steps explain sorting of data in an array in acceding order
  - In first iteration, the largest value moves to the last position in the array
  - In second iteration, the above process is repeated and the second largest value moves to the second last position in the array and so on
  - In n-1 iteration, the data is arranged in ascending order

BUBBLE SORT

- Suppose the name of the array is A and it has four elements with the following values:

4	19	1	3

- To sort this array in ascending order, n-1, i.e. three iterations will be required.

BUBBLE SORT

- Iteration-1
  - A[1] is compared with element A[2]. Since 4 is not greater than 19. there will be no change in the list.

4	19	1	3

  - A[2] is compared with element A[3]. Since 19 is greater than 1, the value are interchanged

4	1	19	3

  - A[3] is compared with element A[4]. Since 19 is grater than 3, the value are interchanged

4	1	3	19

- Thus at the end of the first iteration, the largest value moves to the last position in the array

BUBBLE SORT

- Iteration-2
  - A[1] is compared with element A[2]. Since 4 is greater than 1, the value are interchanged

1	4	3	19

  - A[2] is compared with element A[3]. Since 4 is grater than 3, the value are interchanged

1	3	4	19

- Thus at the end of the second iteration, the second largest value moves to the second last position in the array

BUBBLE SORT

- Iteration-3
  - A[1] is compared with element A[2]. Since 1 is not greater than 3, the value are not interchanged

1	3	4	19

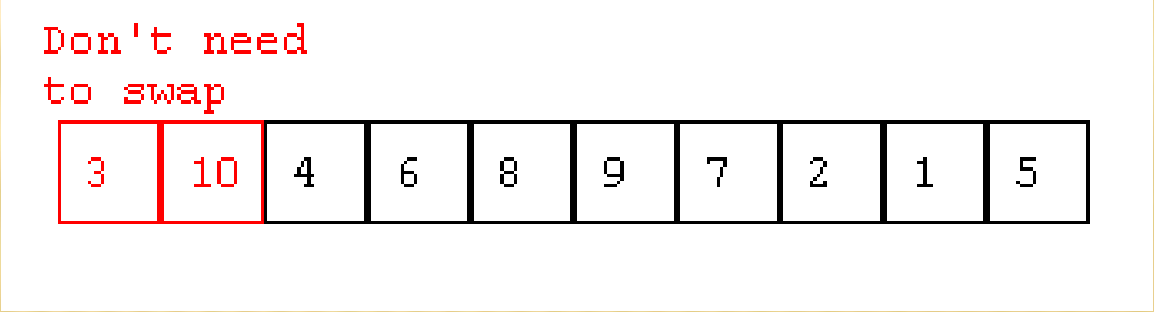
- So array is sorted in ascending order

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.

BUBBLE SORT

- 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

```
void bubbleSort (int a[ ], int n)
{
    int i, j, temp, flag;
    for(i=n-1; i>0; i- -)
    {
        flag = 1;
        for(j=0; i>j; j++)
        {
            if(a[j]>a[j+1])
            {
                flag = 0;
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        } //out this block when flag is true, i.e. inner loop performed no swaps, so the list is already sorted
        if(flag)
            break;
    }
}
```

BUBBLE SORT EXAMPLE

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

6, 2, 9, 11, 9, 12, 3, 7

6, 2, 9, 11, 9, 3, 12, 7

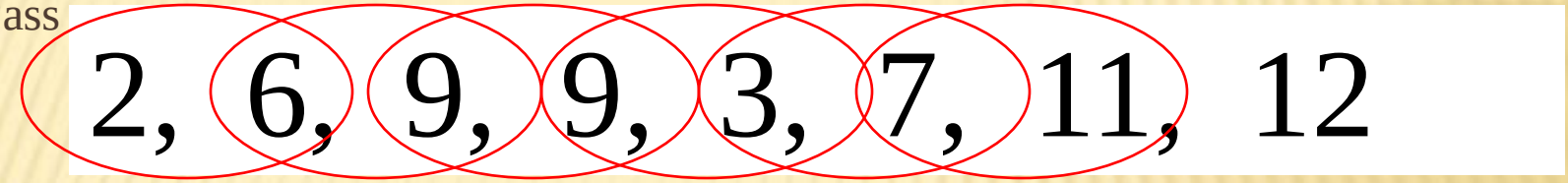
6, 2, 9, 11, 9, 3, 7, 12

BUBBLE SORT EXAMPLE

First Pass

6, 2, 9, 11, 9, 3, 7, 12

Second Pass



Notice that this time we do not have to compare the last two numbers as we know the 12 is in position. This pass therefore only requires 6 comparisons.

BUBBLE SORT EXAMPLE

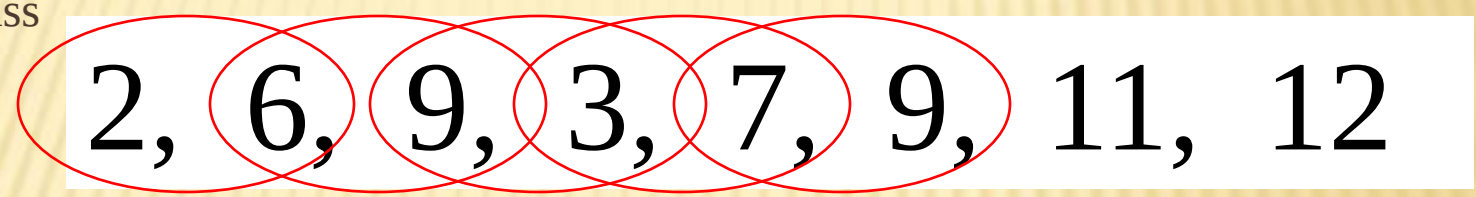
First Pass

6, 2, 9, 11, 9, 3, 7, 12

Second Pass

2, 6, 9, 9, 3, 7, 11, 12

Third Pass



This time the 11 and 12 are in position. This pass therefore only requires 5 comparisons.

BUBBLE SORT EXAMPLE

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Third Pass 2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass 2, 6, 3, 7, 9, 9, 11, 12

Each pass requires fewer comparisons. This time only 4 are needed.

BUBBLE SORT EXAMPLE

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Third Pass 2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass 2, 6, 3, 7, 9, 9, 11, 12

Fifth Pass 2, 3, 6, 7, 9, 9, 11, 12

The list is now sorted but the algorithm does not know this until it completes a pass with no exchanges.

MERGE SORT

- **Merge sort** is a sorting algorithm for rearranging lists (or any other data structure that can only be accessed sequentially) into a specified order.
- It is a particularly good example of the divide and conquer algorithmic paradigm.

MERGE SORT

- Conceptually, merge sort works as follows:
  - Divide the unsorted list into two sub-lists of about half the size.
  - Sort each of the two sub-lists.
  - Merge the two sorted sub-lists back into one sorted list.

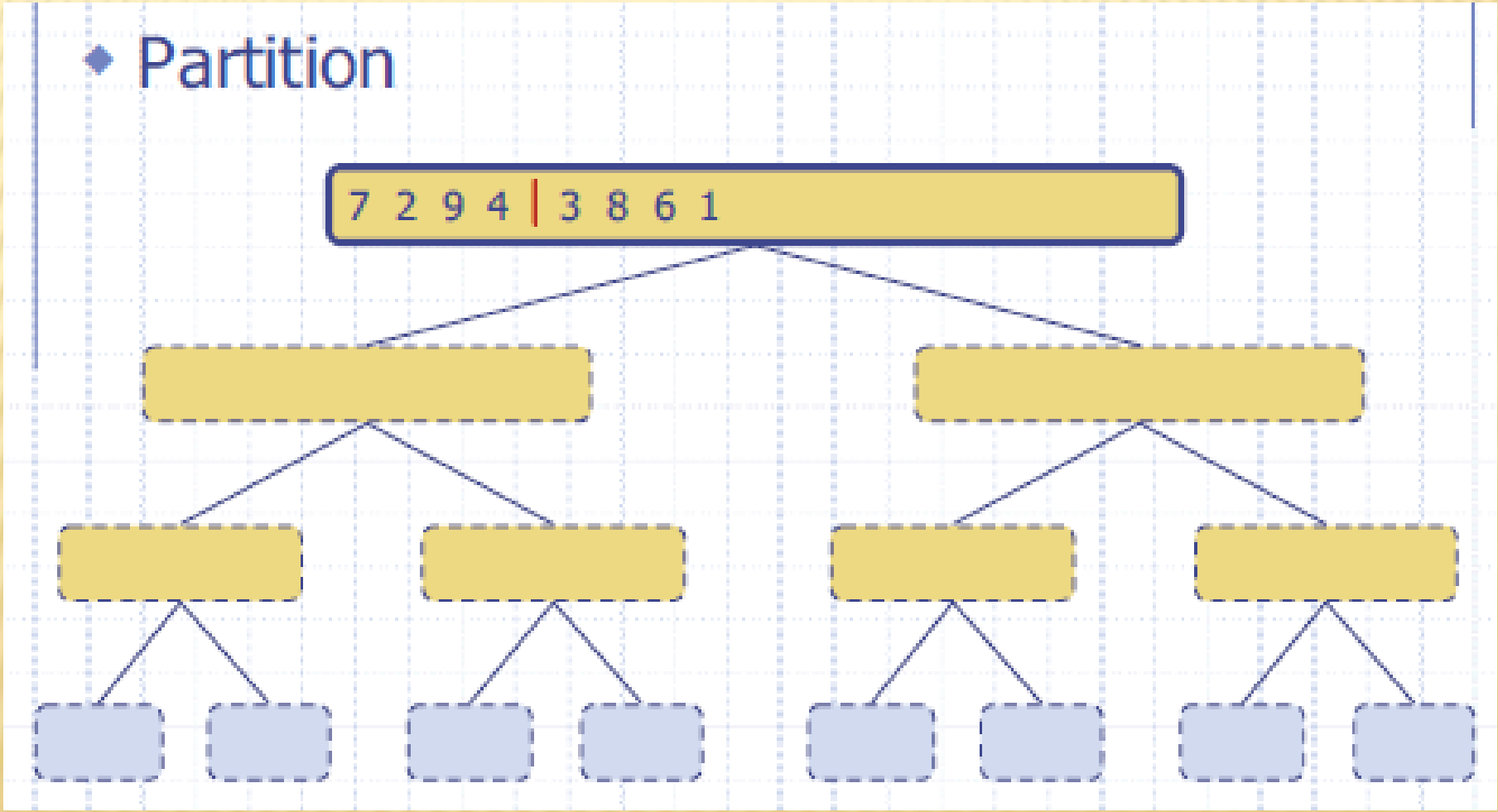
MERGE SORT

```
Array mergeSort(Array m)
   Array left, right.
    if length(m) ≤ 1
        return m
    else
        middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = mergesort(left)
        right = mergesort(right)
        result = merge(left, right)
        return result
```

MERGE SORT

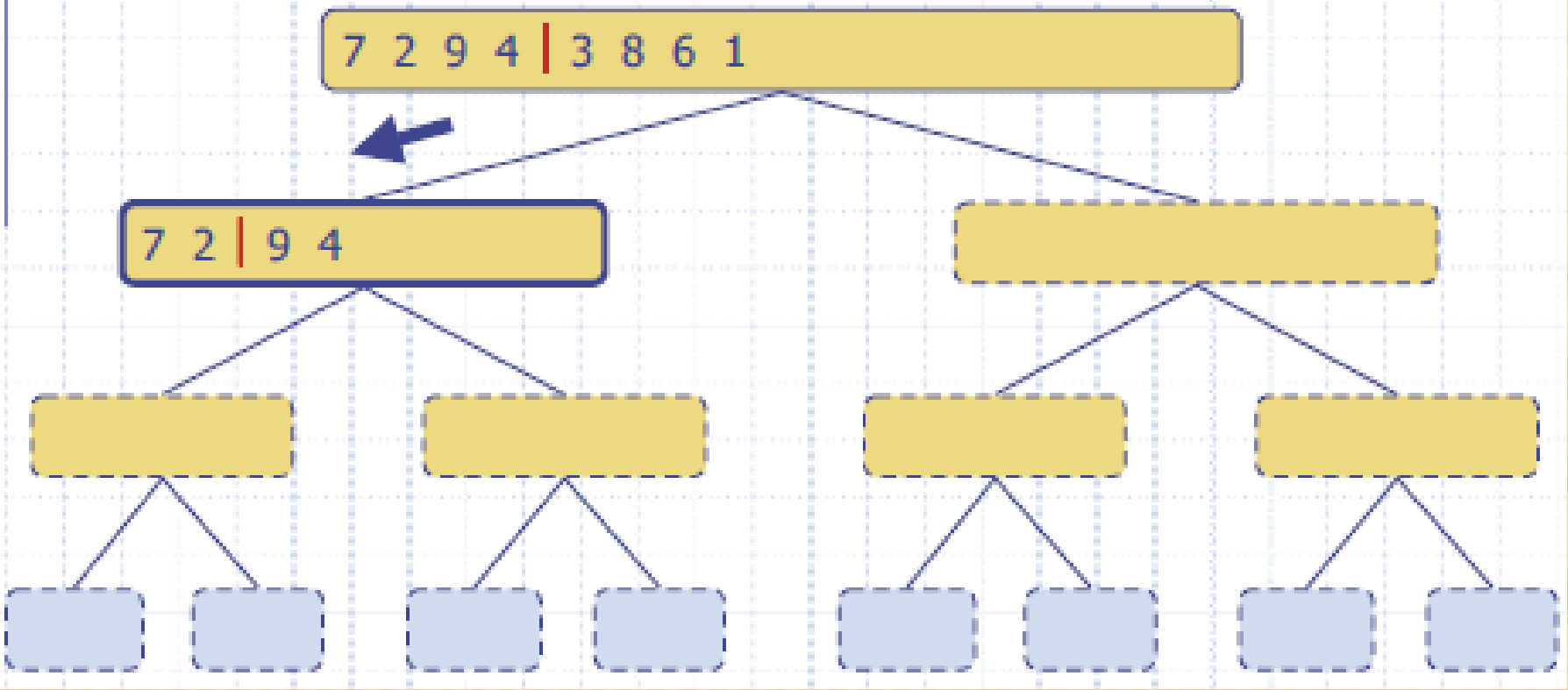
```
Array merge(left,right)
    Array result
    while length(left) > 0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    if length(left) > 0
        append left to result
    if length(right) > 0
        append right to result
    return result
```

EXECUTION EXAMPLE



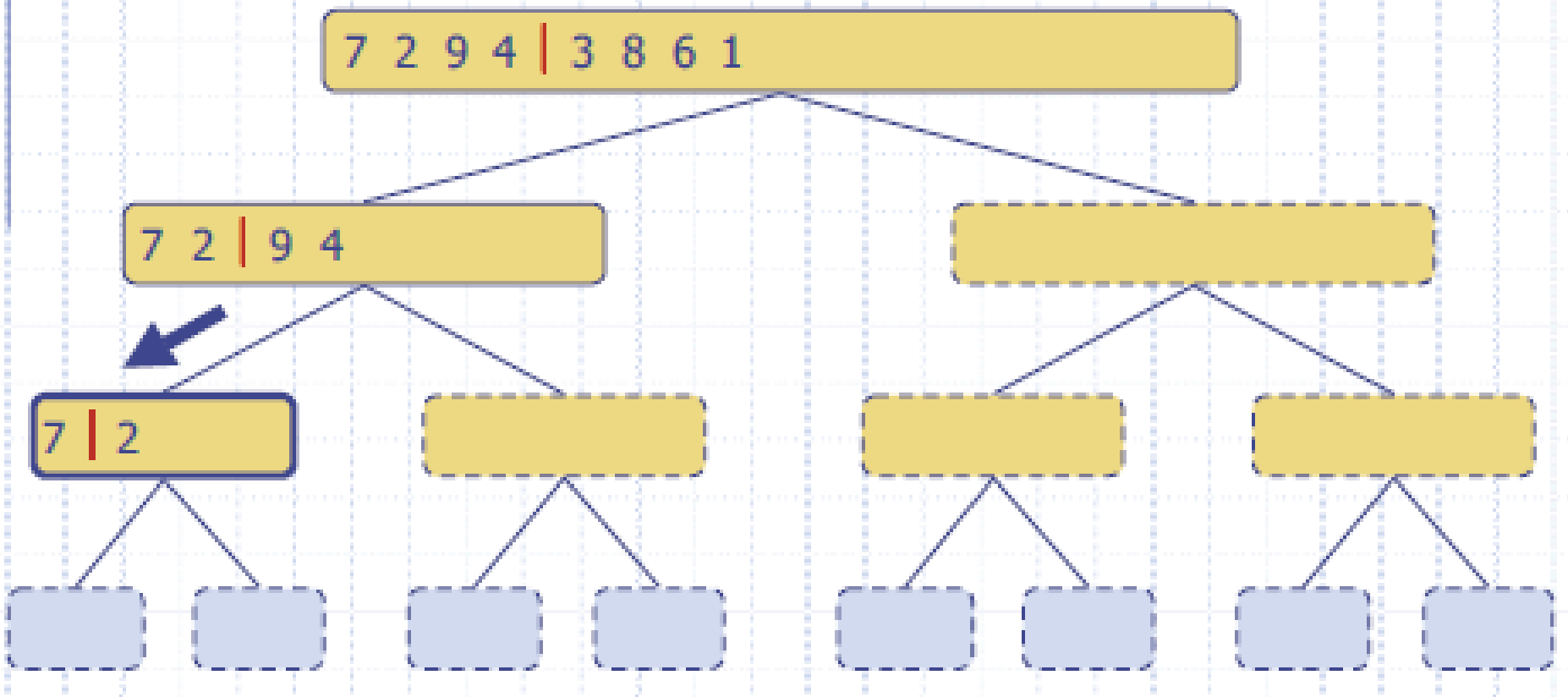
Execution Example

- Recursive call, partition

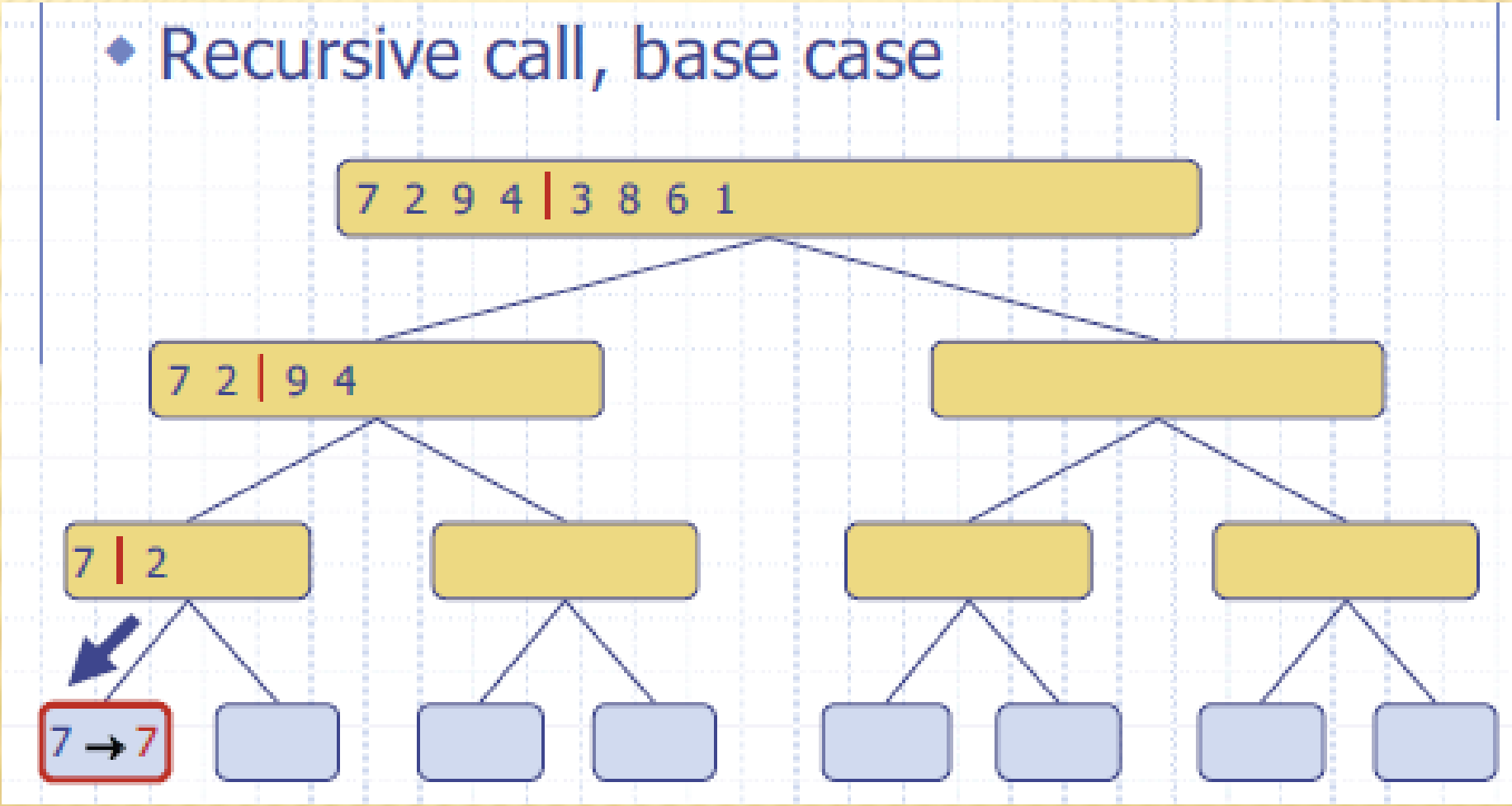


Execution Example

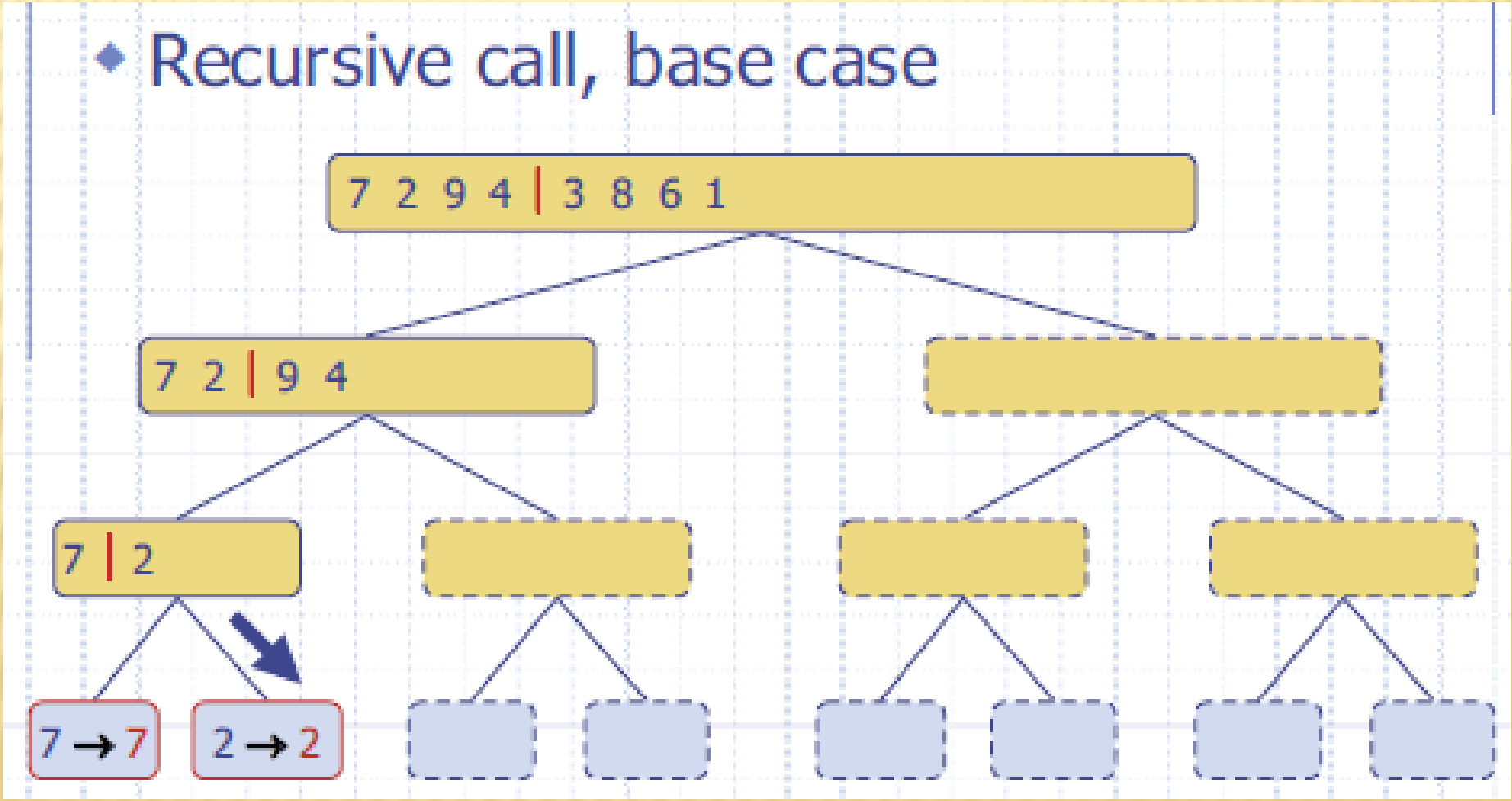
- Recursive call, partition



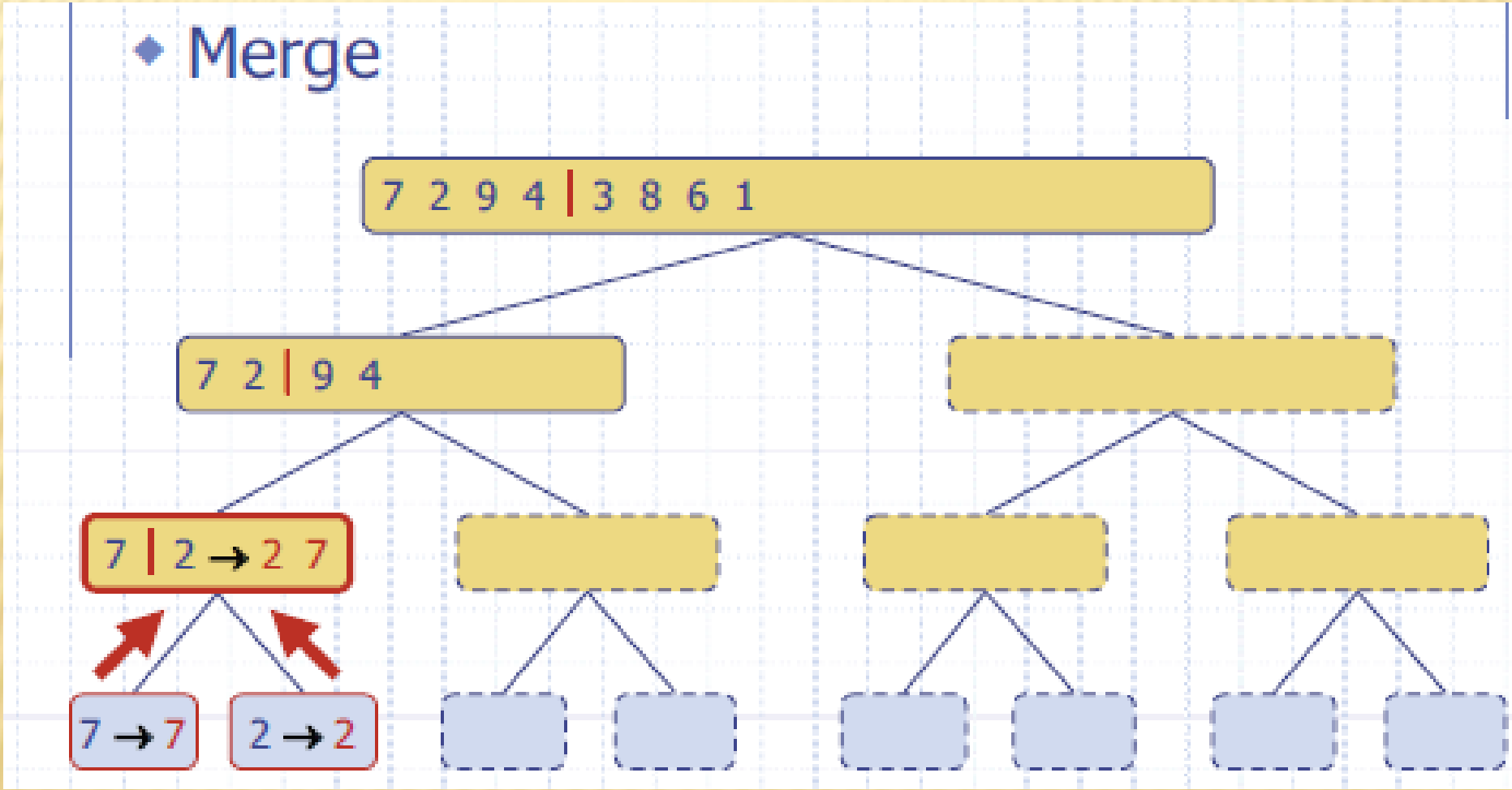
Execution Example



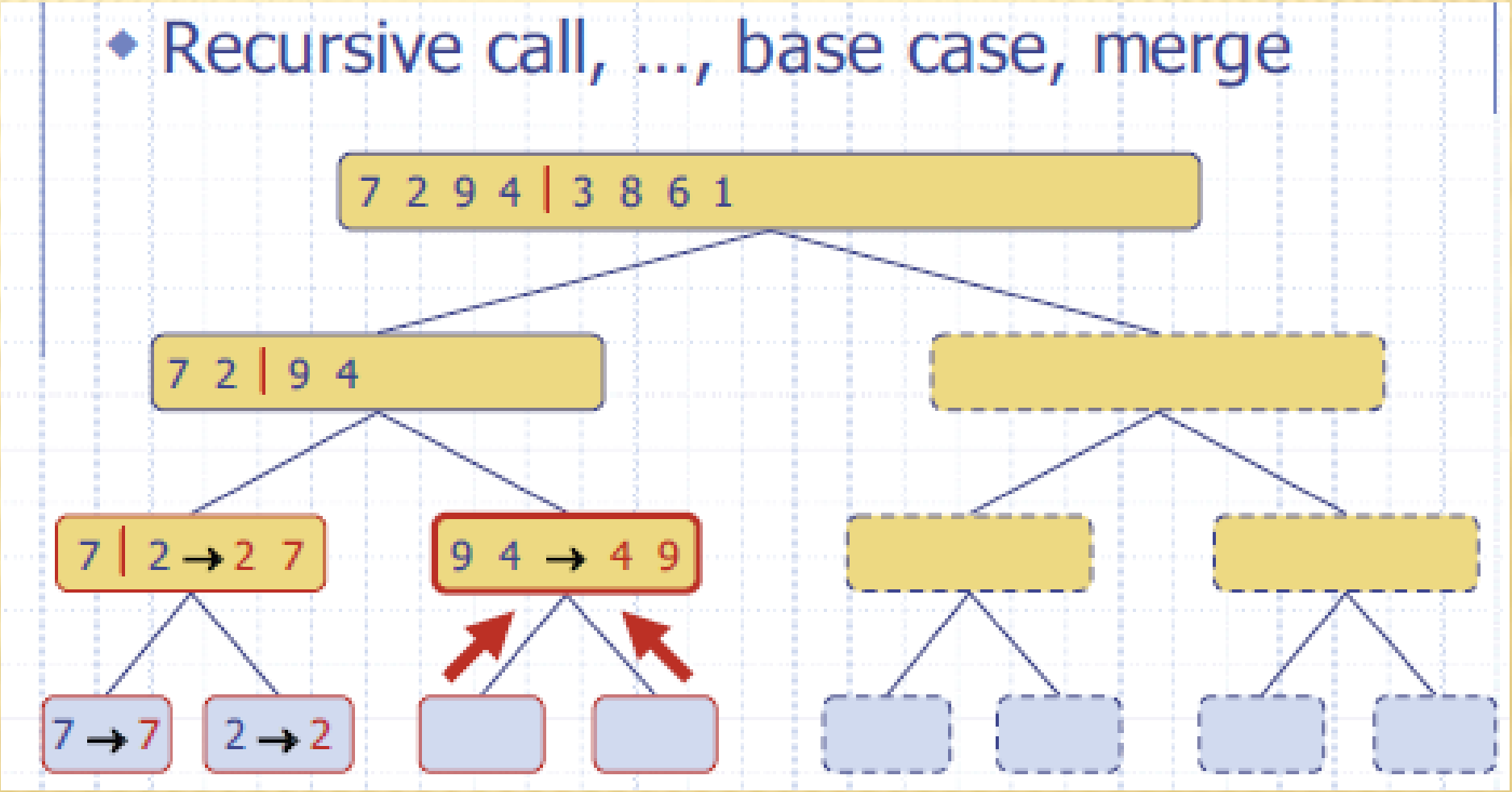
Execution Example



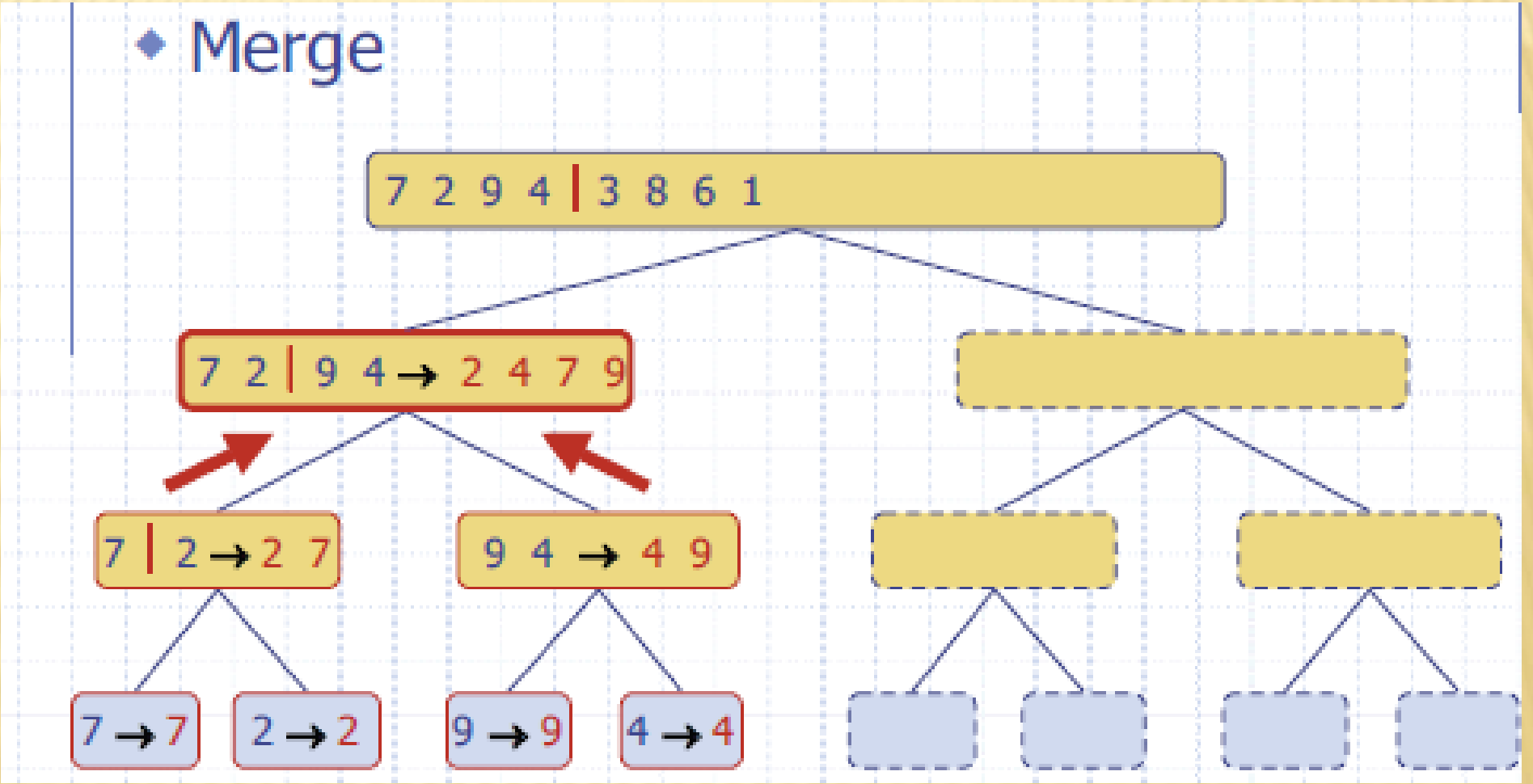
Execution Example



Execution Example

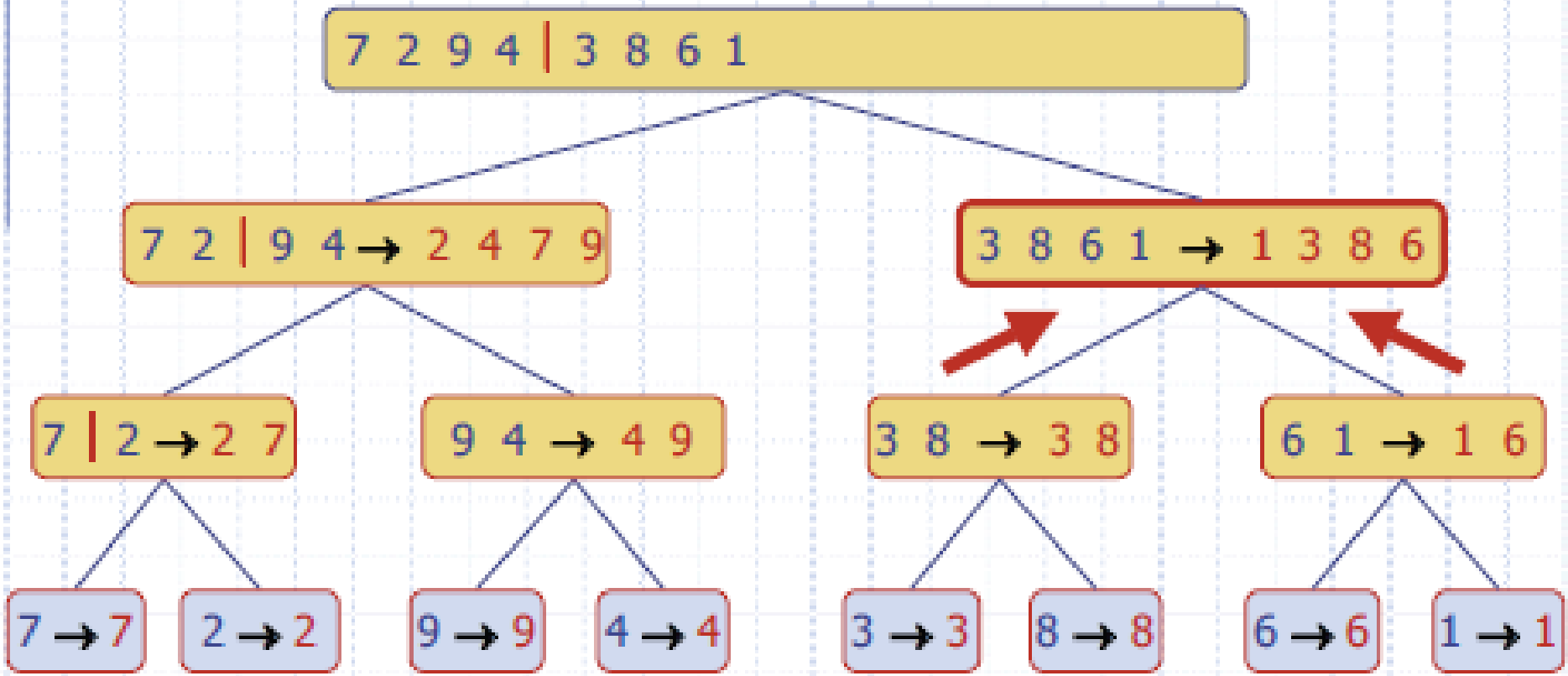


Execution Example

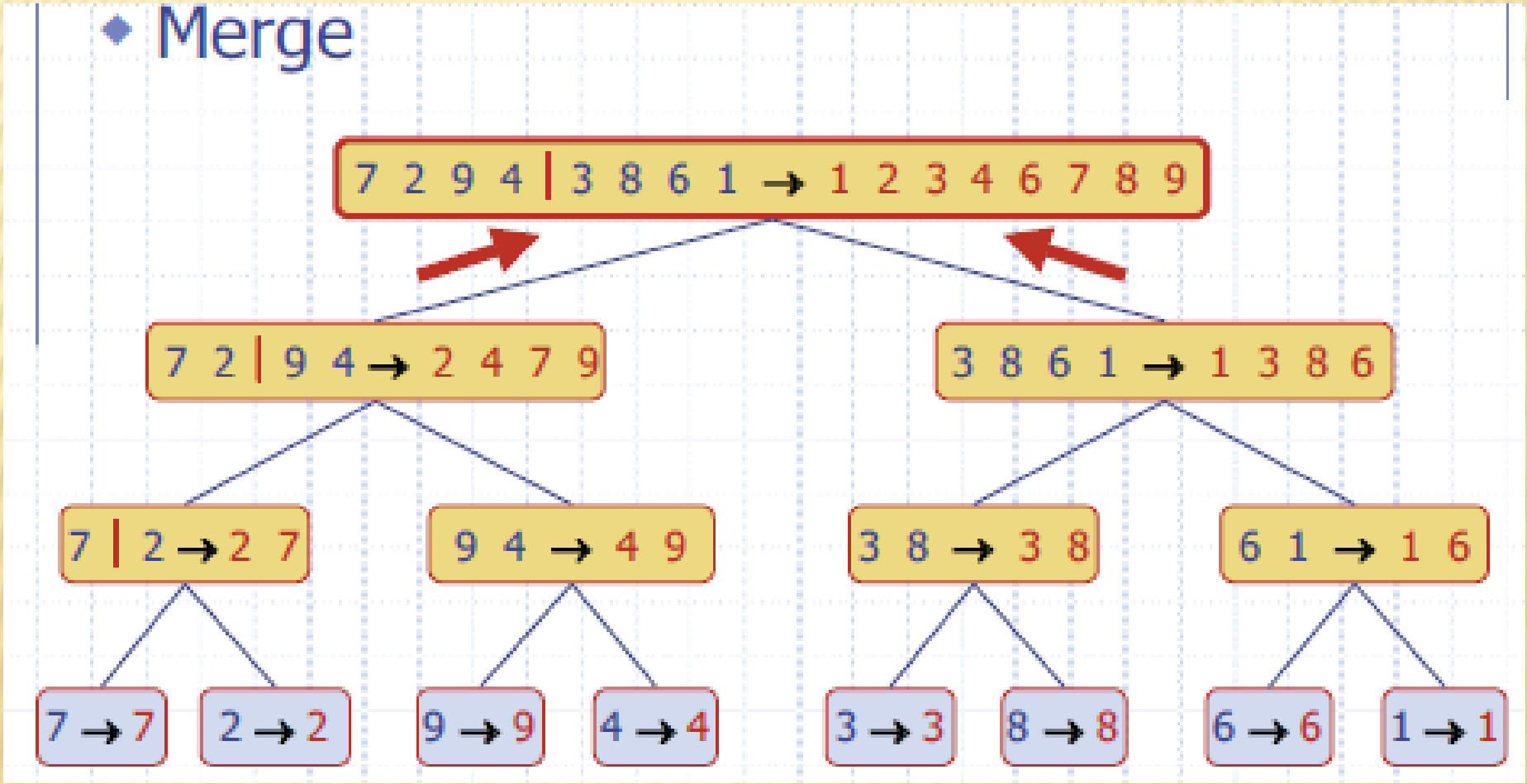


Execution Example

- Recursive call, ..., merge, merge



Execution Example



MERGE SORT ANOTHER EXAMPLE

- Suppose the name of the array is AB and it has six elements with the following values:

16	17	2	8	18	1

- To sort this array in ascending order

MERGE SORT

AB

16	17	2	8	18	1

- Divide array AB into two sub-arrays A & B

A

16	17	2

B

8	18	1

- Sort A & B using Bubble or selection or insertion sort

A

2	16	17

B

1	8	18

MERGE SORT

A		
2	16	17

B		
1	8	18

- Compare A[1] to B[1], so B[1] is less than A[1], the value of B[1] is move to AB[1]

AB

1					

MERGE SORT

A

2	16	17

B

1	8	18

- Compare A[1] to B[2], so A[1] is less than B[2], the value of A[2] is move to AB[2]

AB

1	2				

MERGE SORT

A

2	16	17

B

1	8	18

- Compare A[2] to B[2], so B[2] is less than A[2], the value of B[2] is move to AB[3]

AB

1	2	8			

MERGE SORT

A

2	16	17

B

1	8	18

- Compare A[2] to B[3], so A[2] is less than B[3], the value of A[2] is move to AB[4]

AB

1	2	8	16		

MERGE SORT

A

2	16	17

B

1	8	18

- Compare A[3] to B[3], so A[3] is less than B[3], the value of A[3] is move to AB[5]

AB

1	2	8	16	17	

- At the end, B[3] is move to AB[6], array is sorted

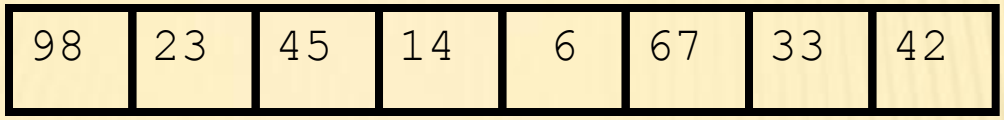
AB

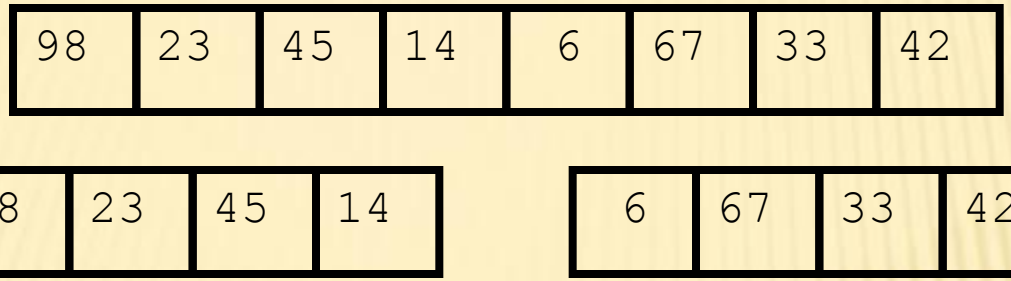
1	2	8	16	17	18

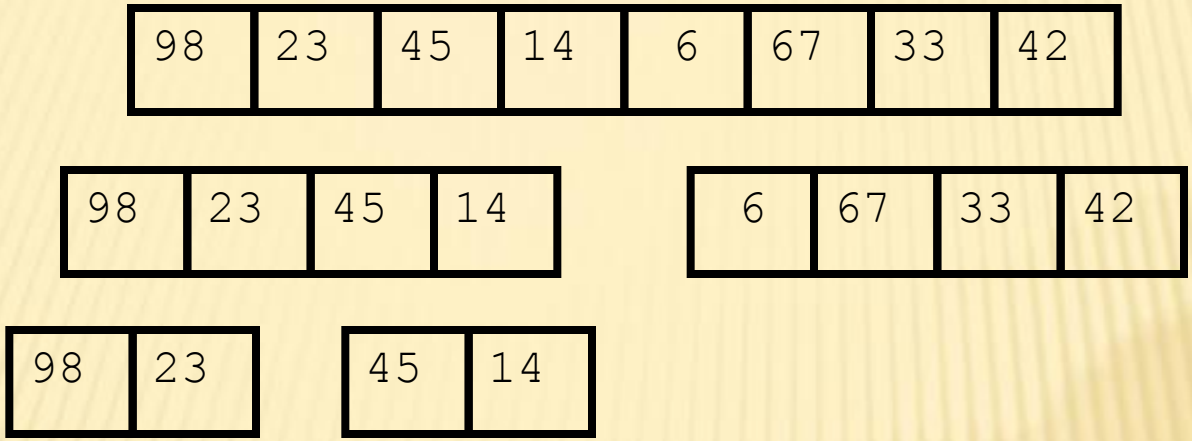
LB

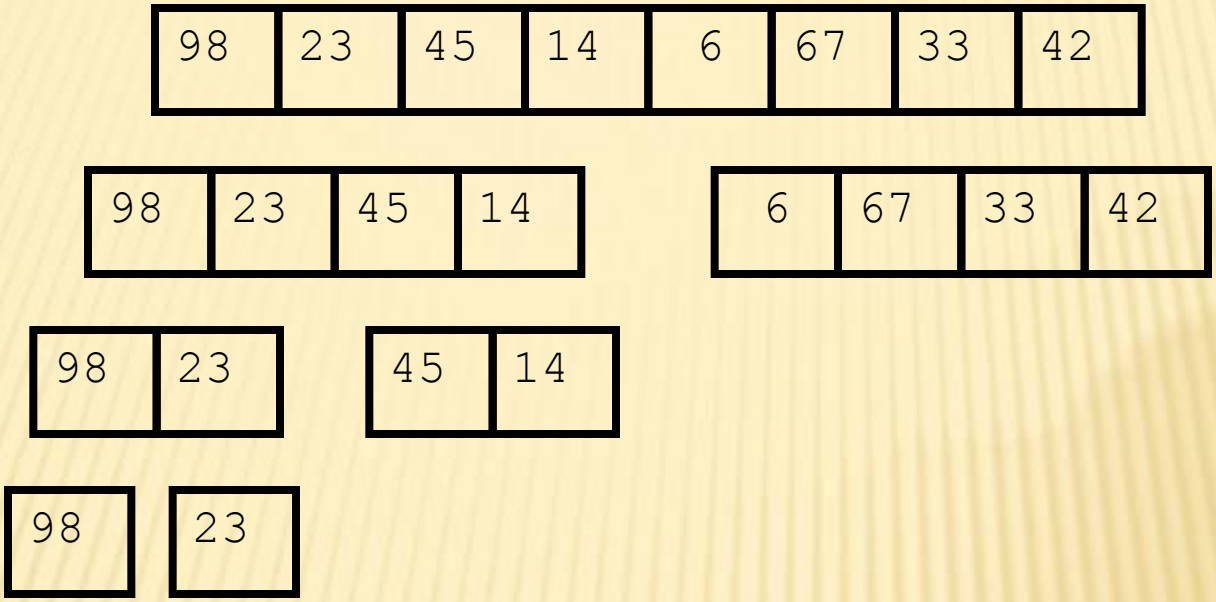
ALGORITHM

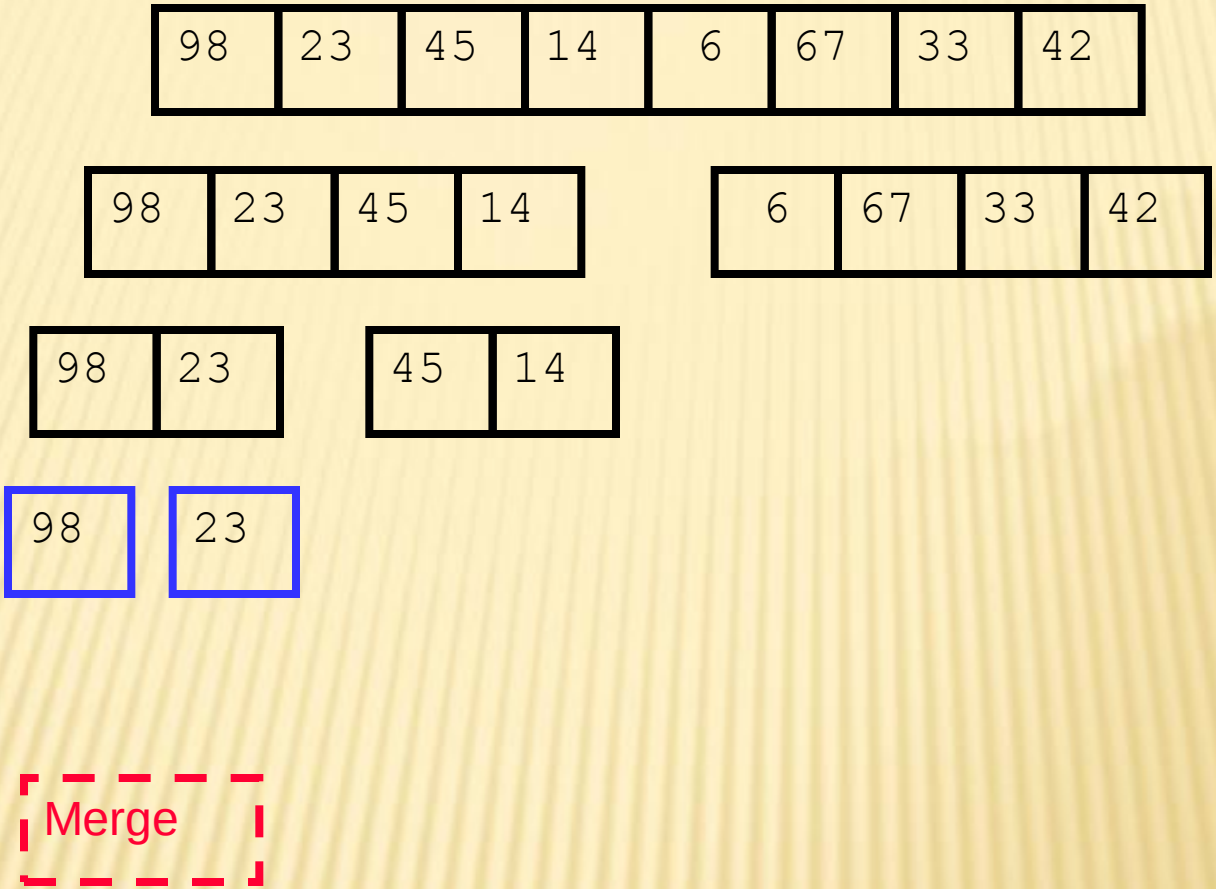
```
Mergesort(Passed an array)
   if array size > 1
      Divide array in half
      Call Mergesort on first half.
      Call Mergesort on second half.
      Merge two halves.

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

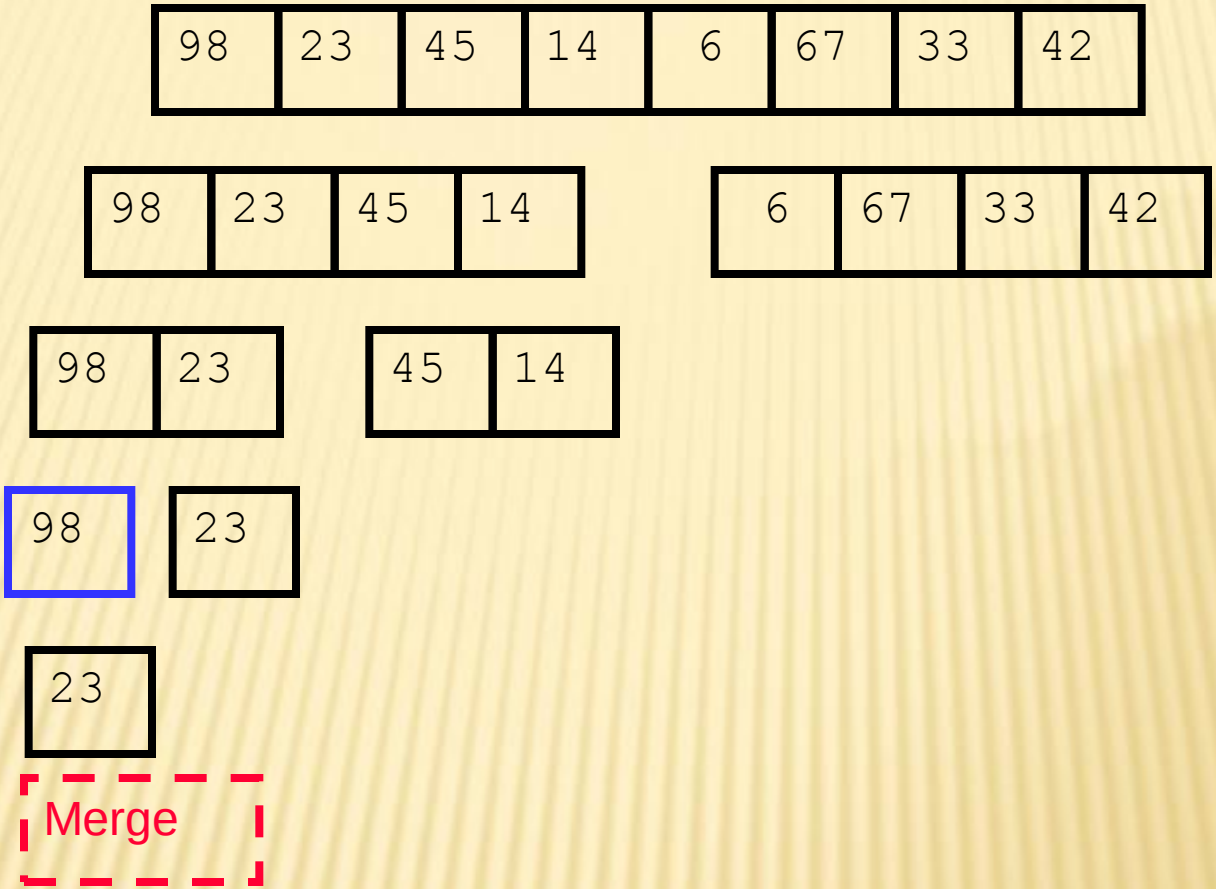

98
23
45
14
6
67
33
42

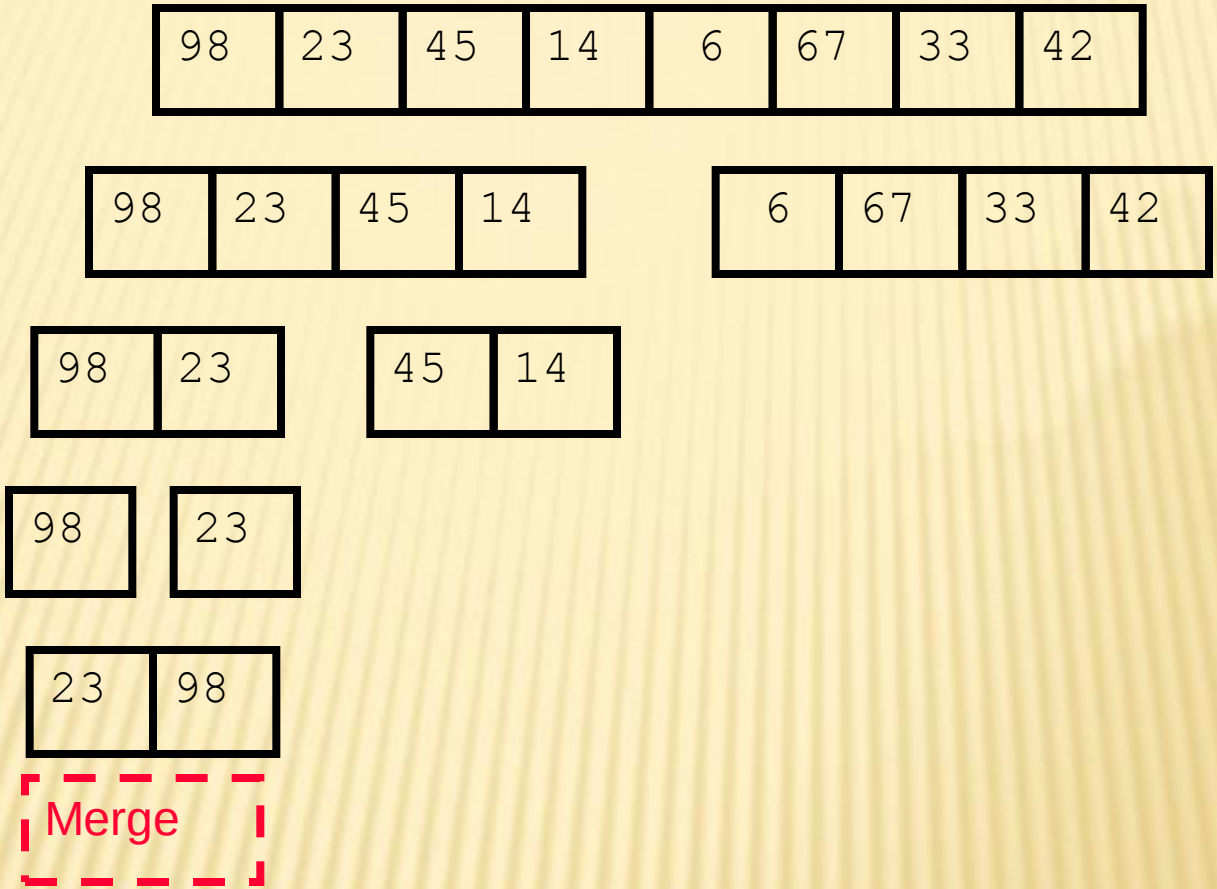

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42

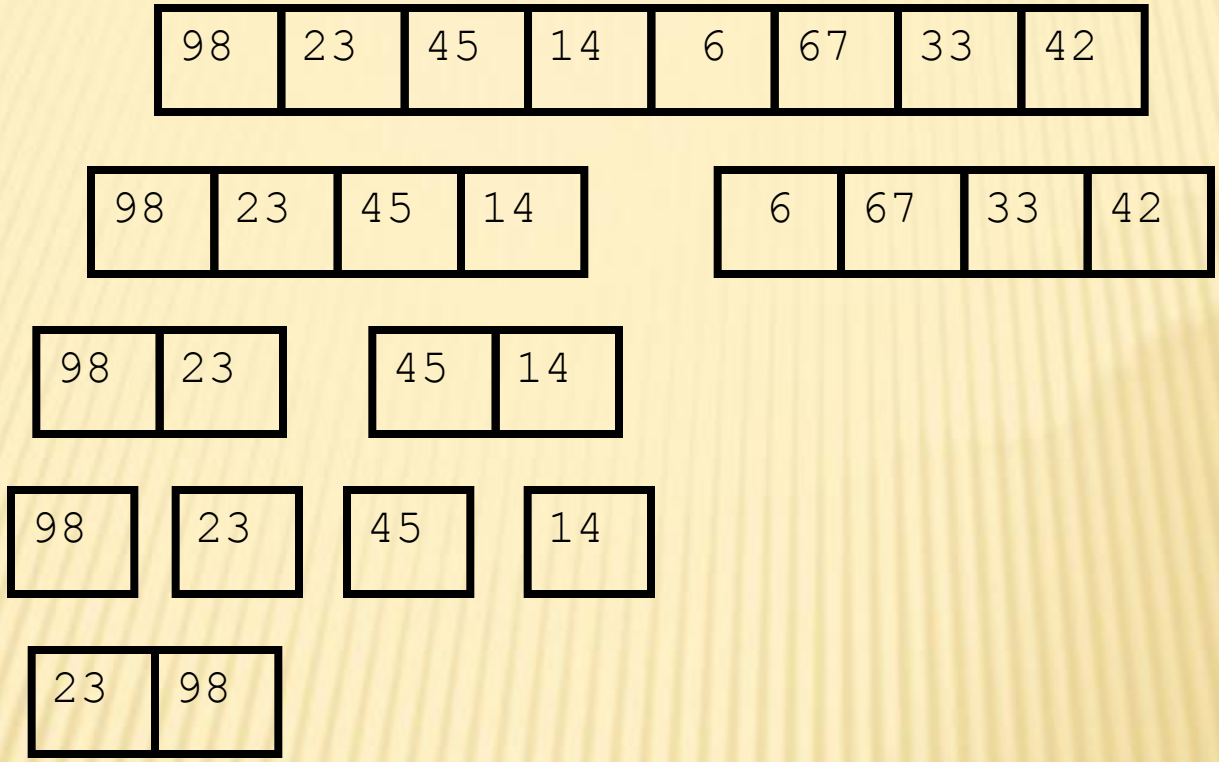

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

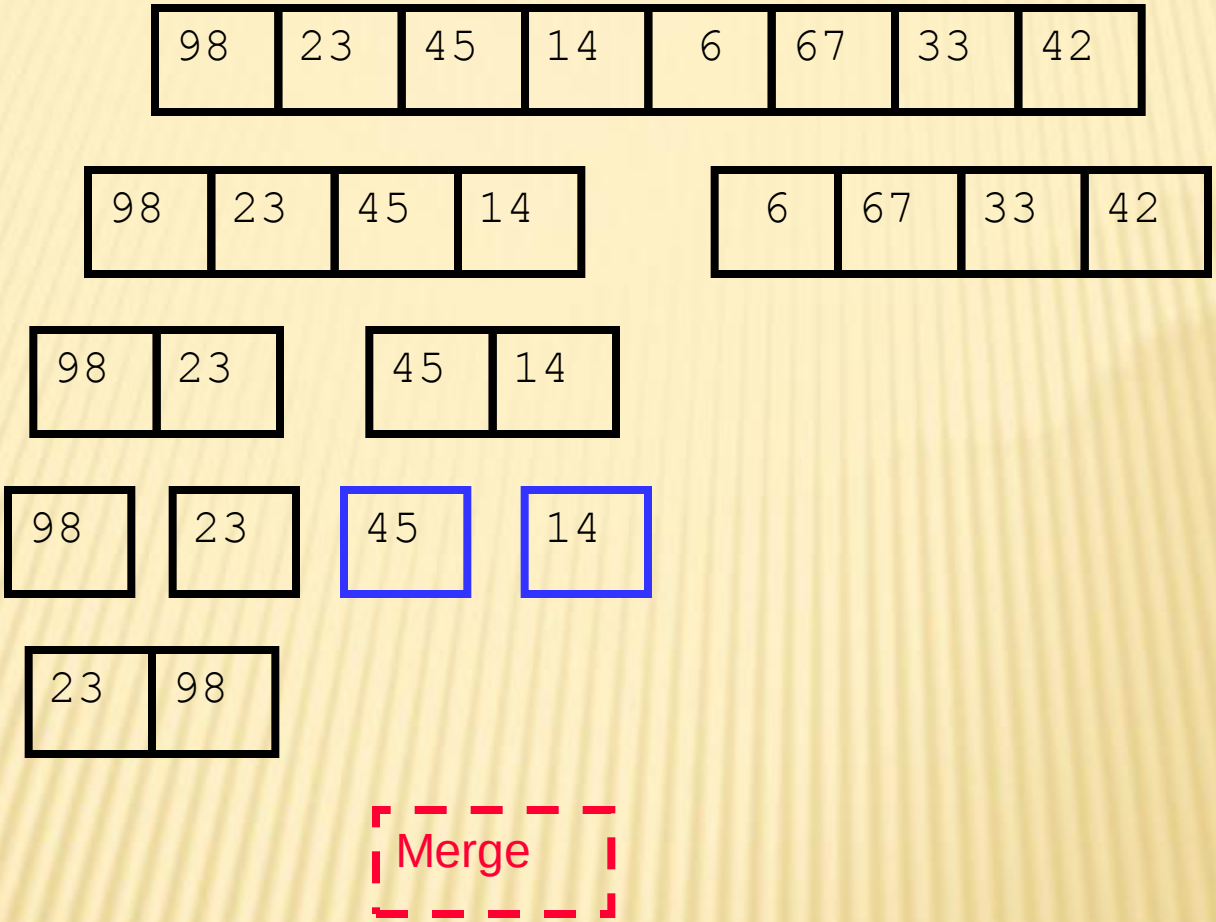

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

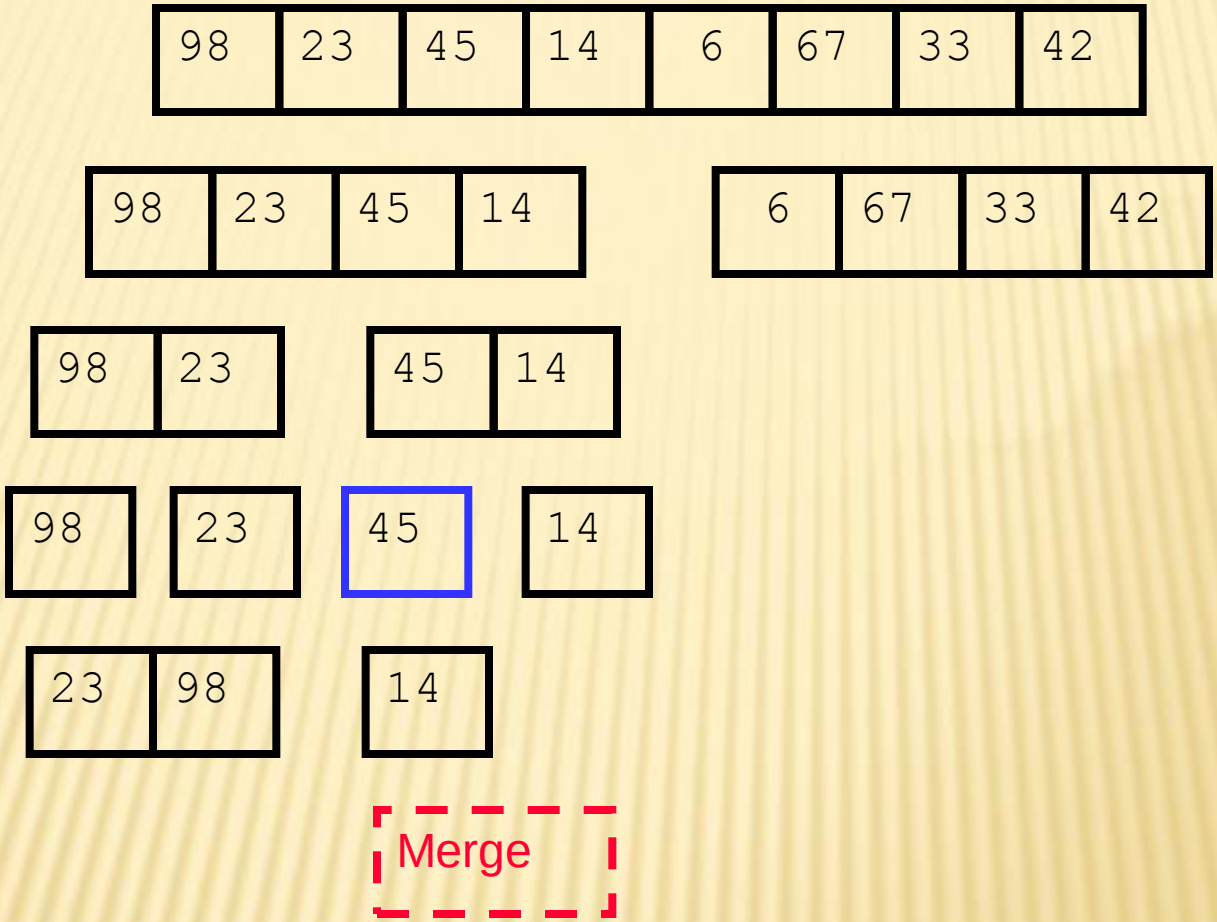

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

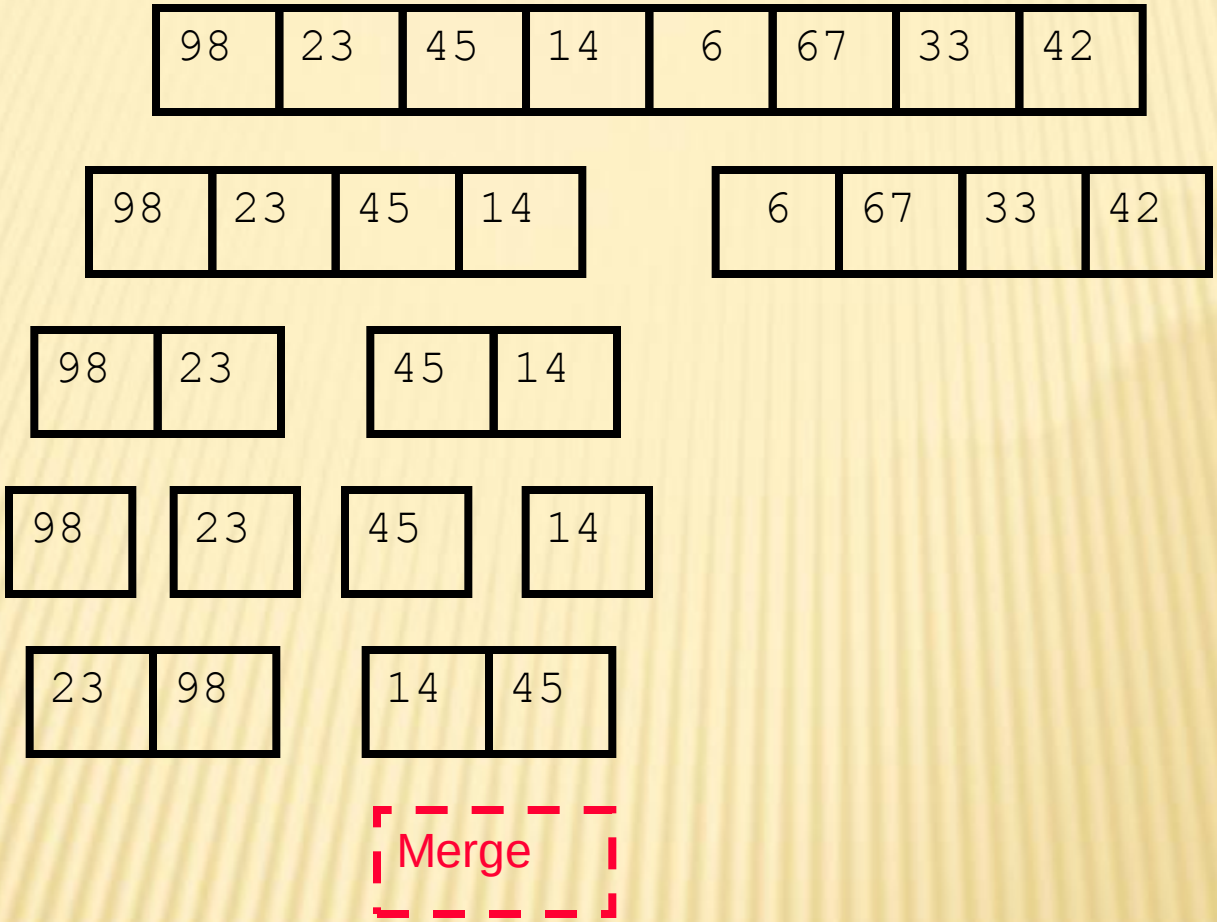

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

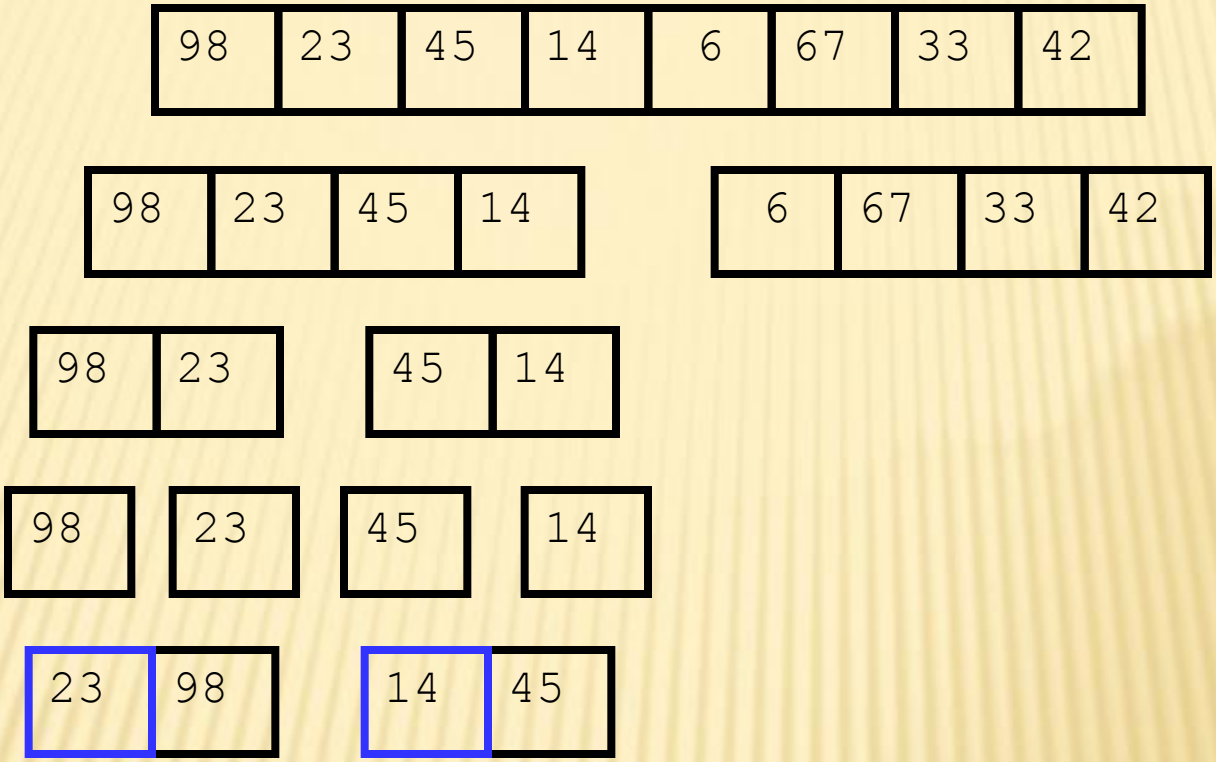

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge

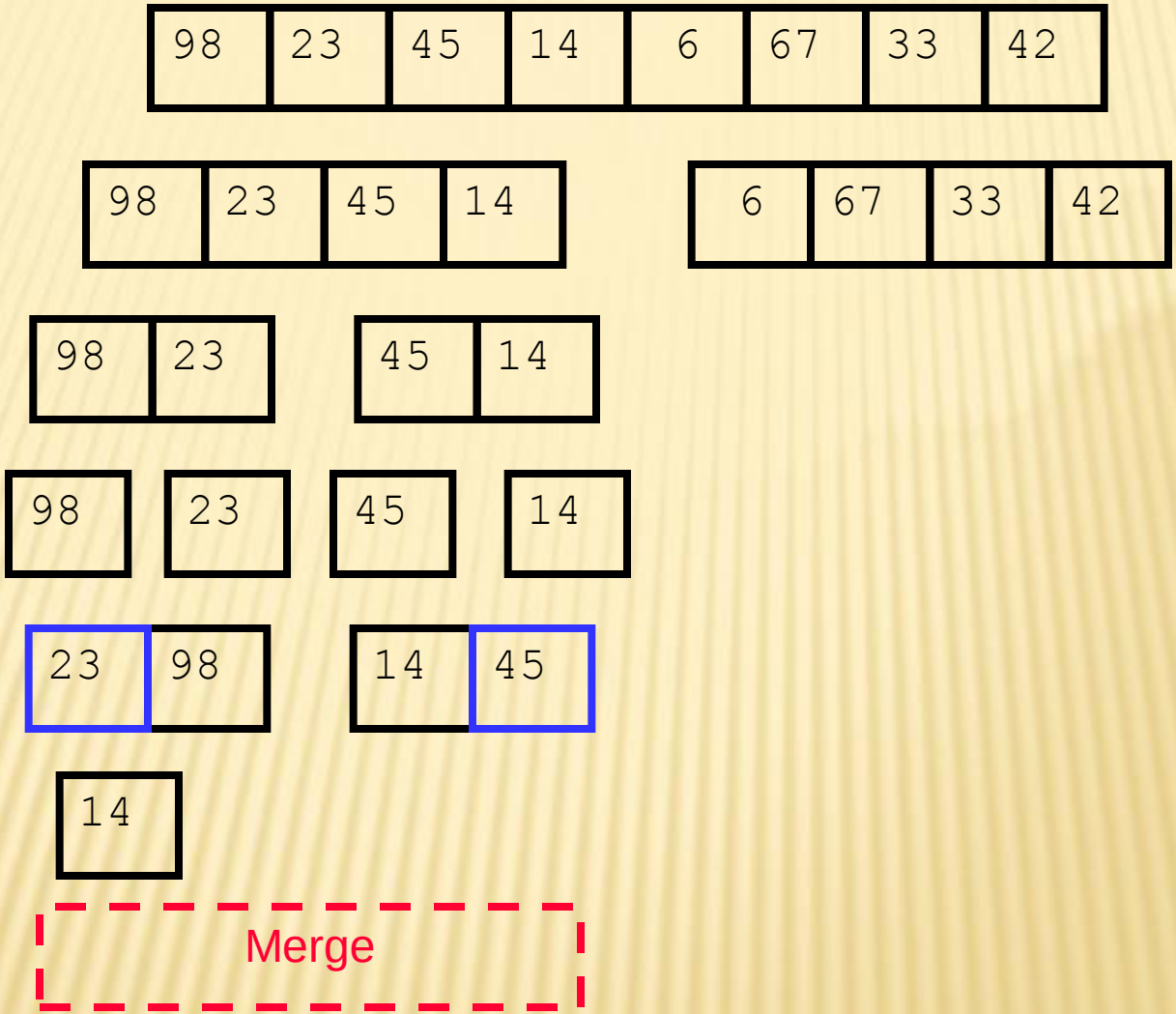

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98


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

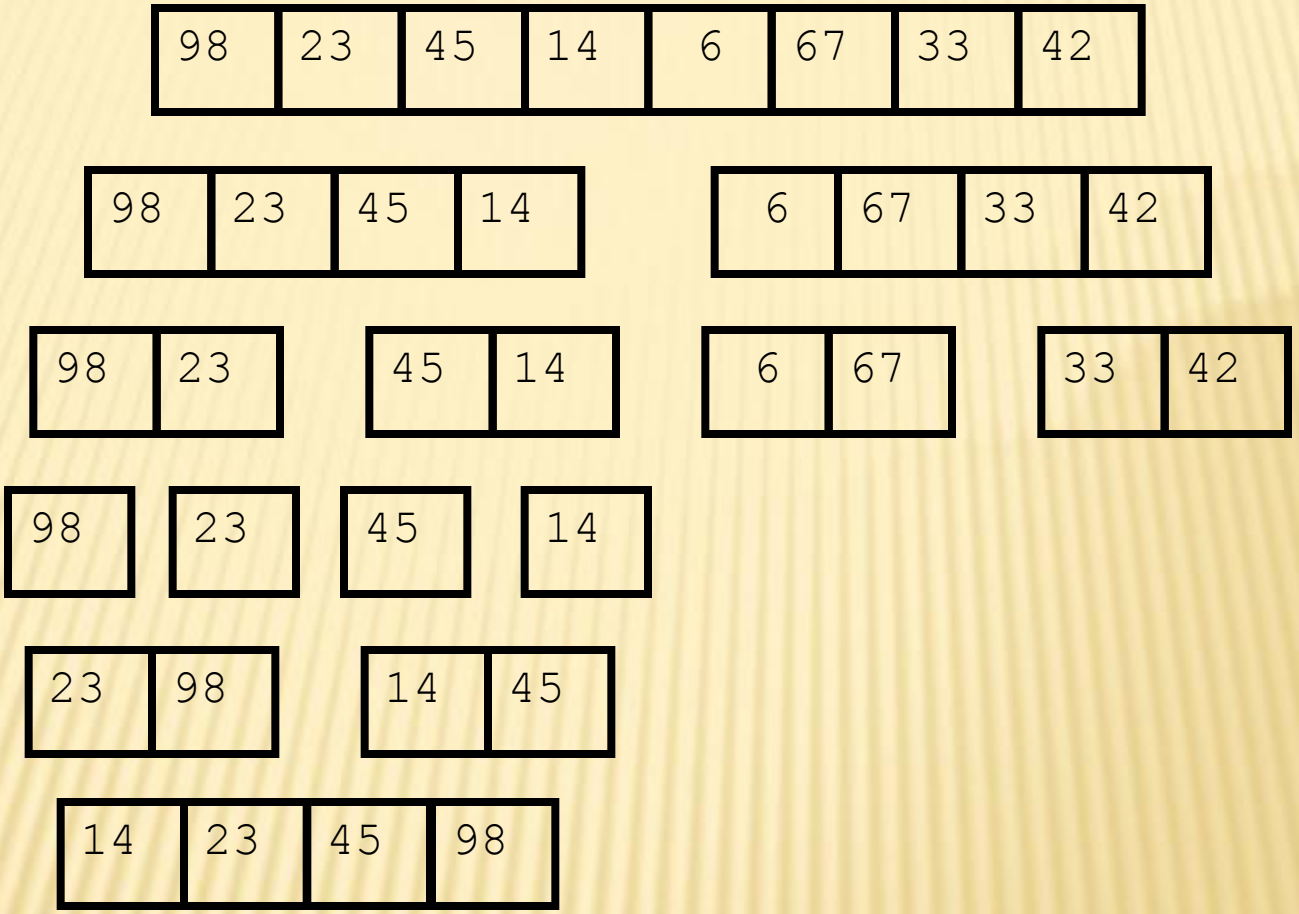

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

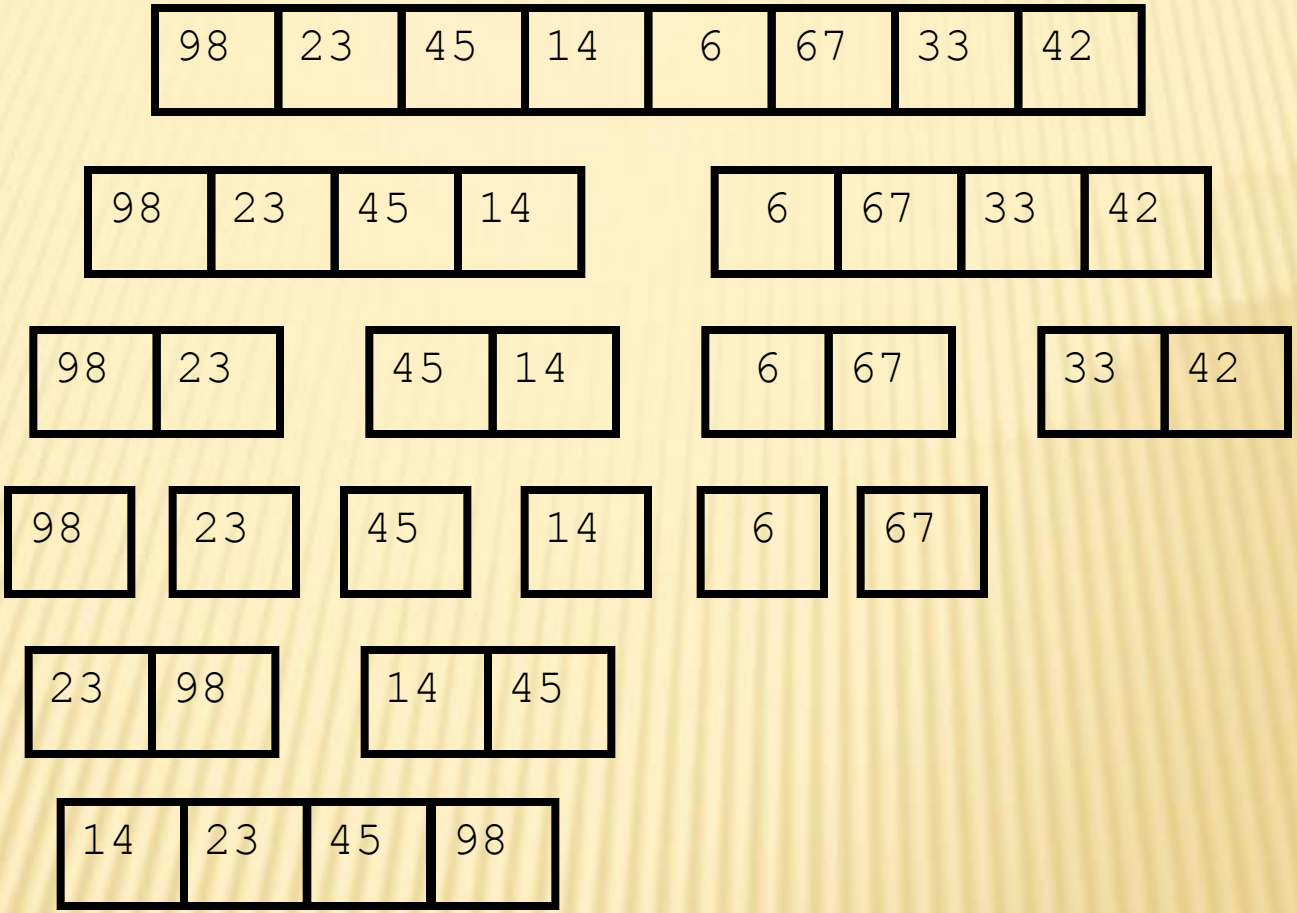

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

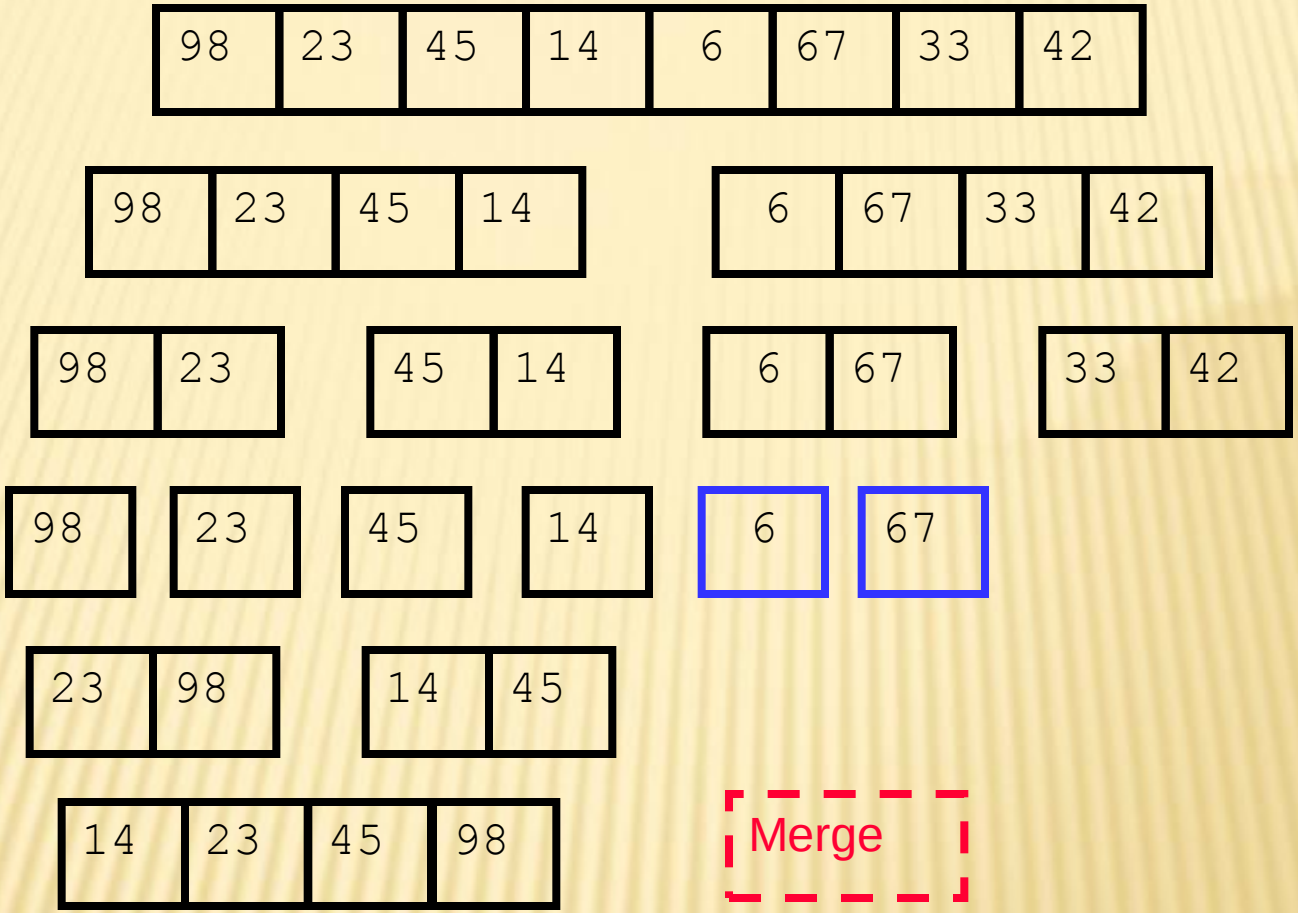

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

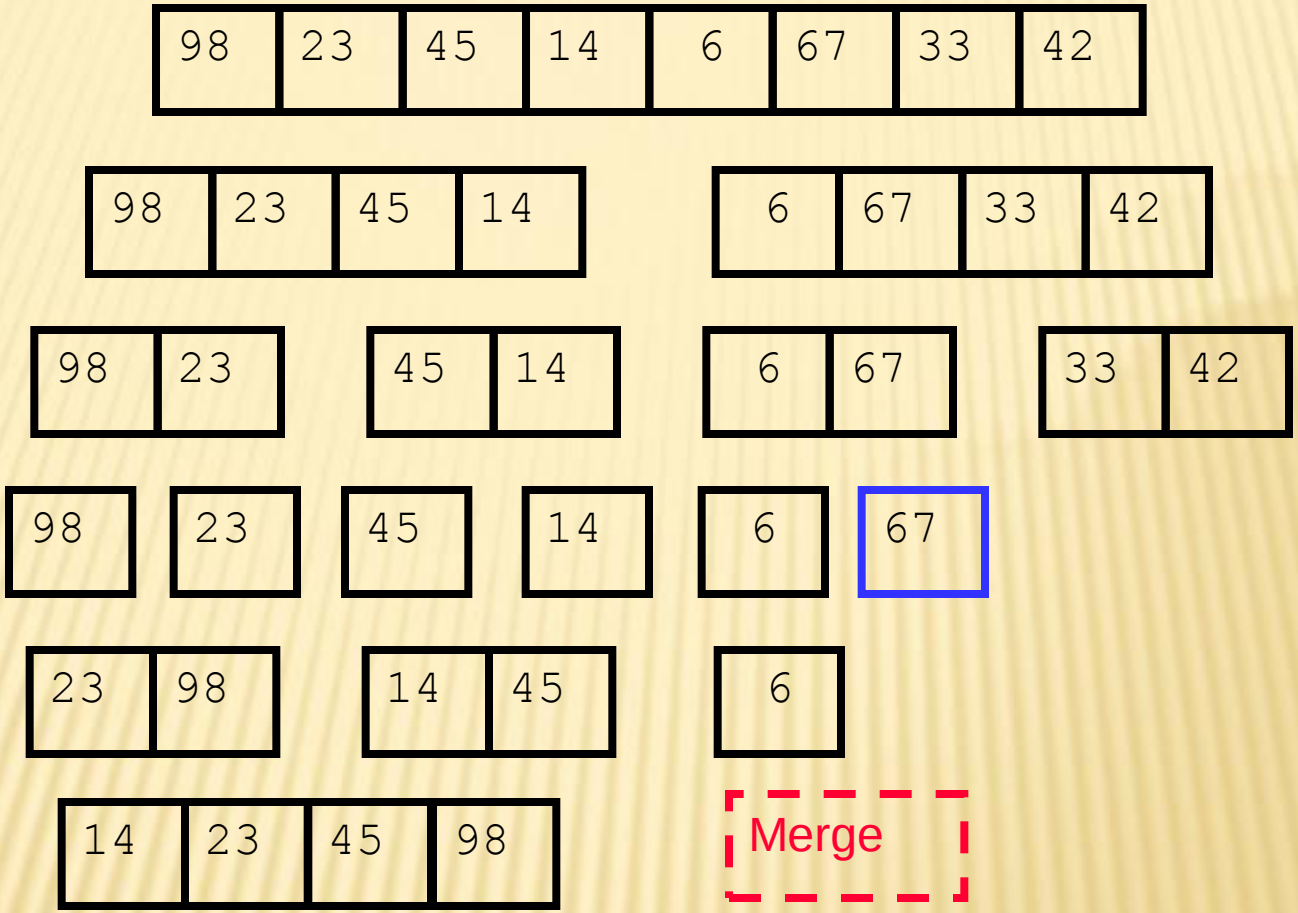

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

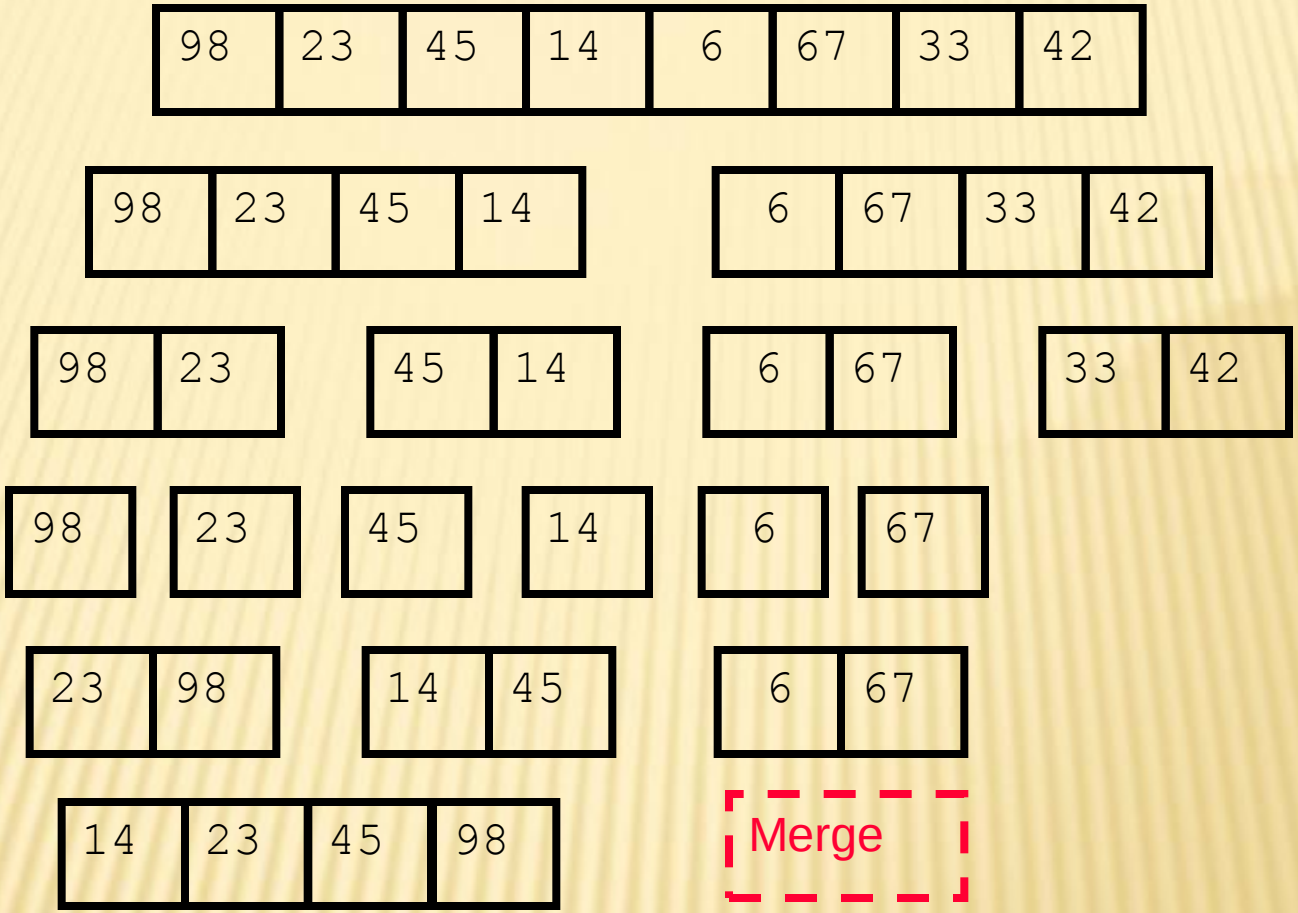

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

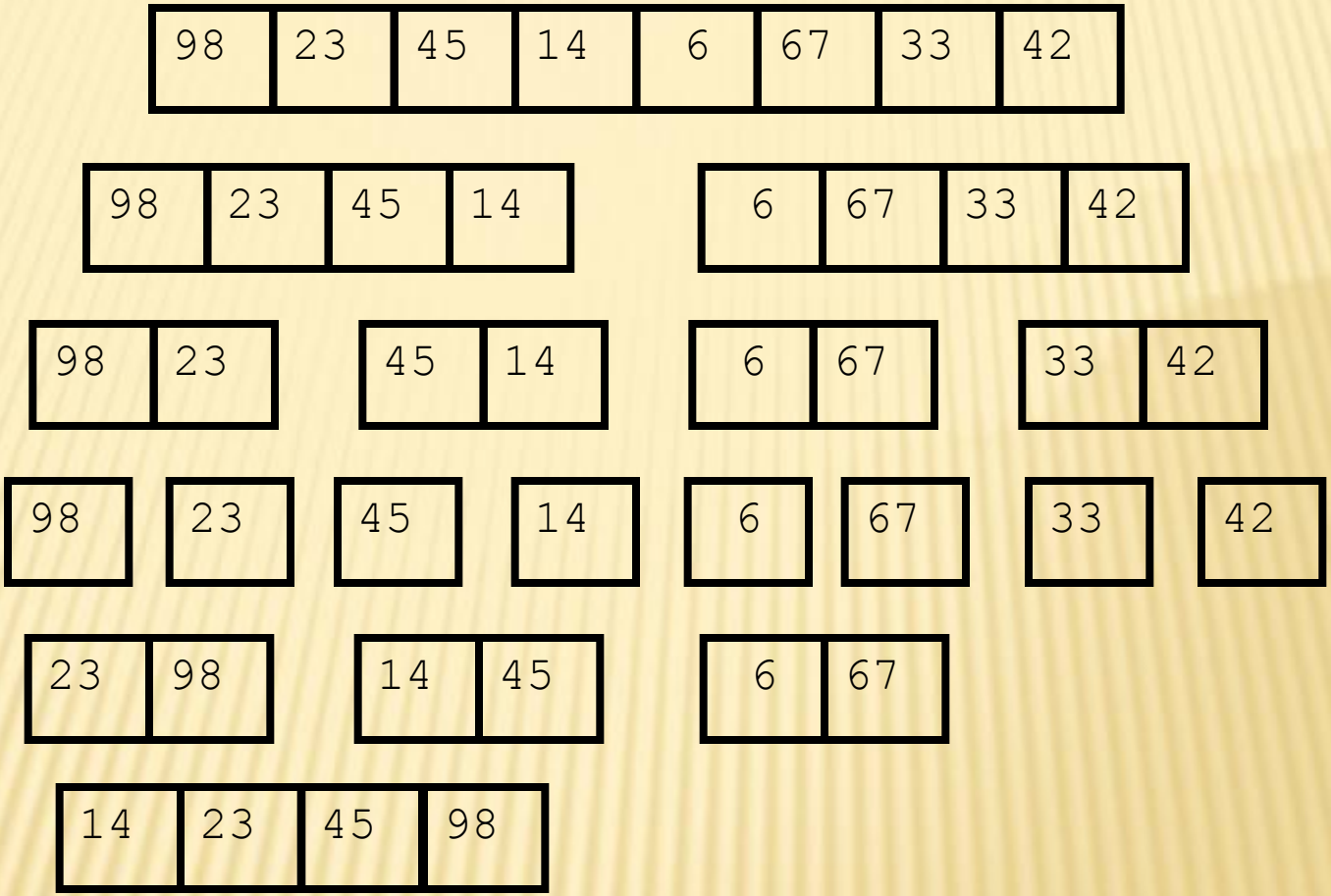

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

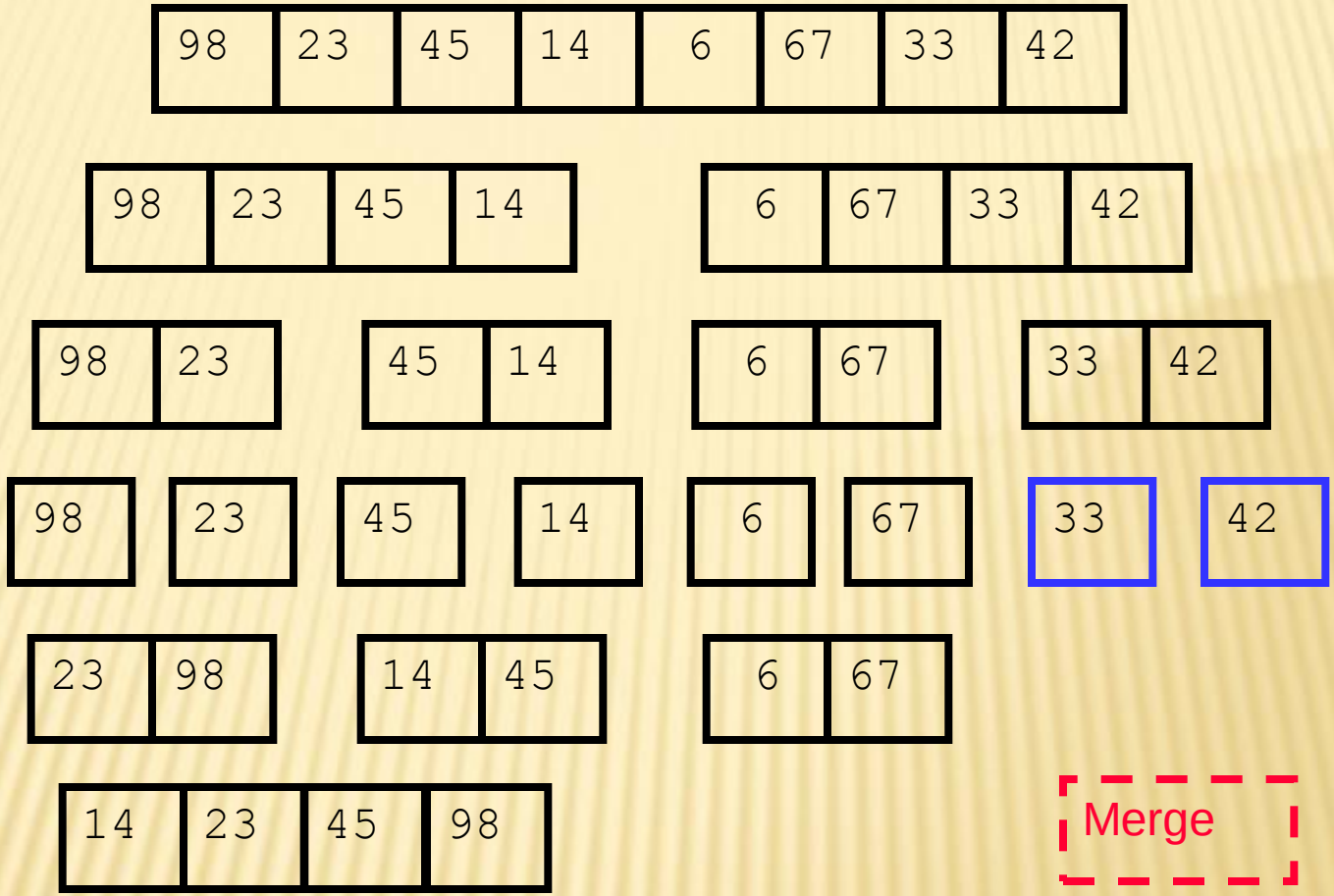

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

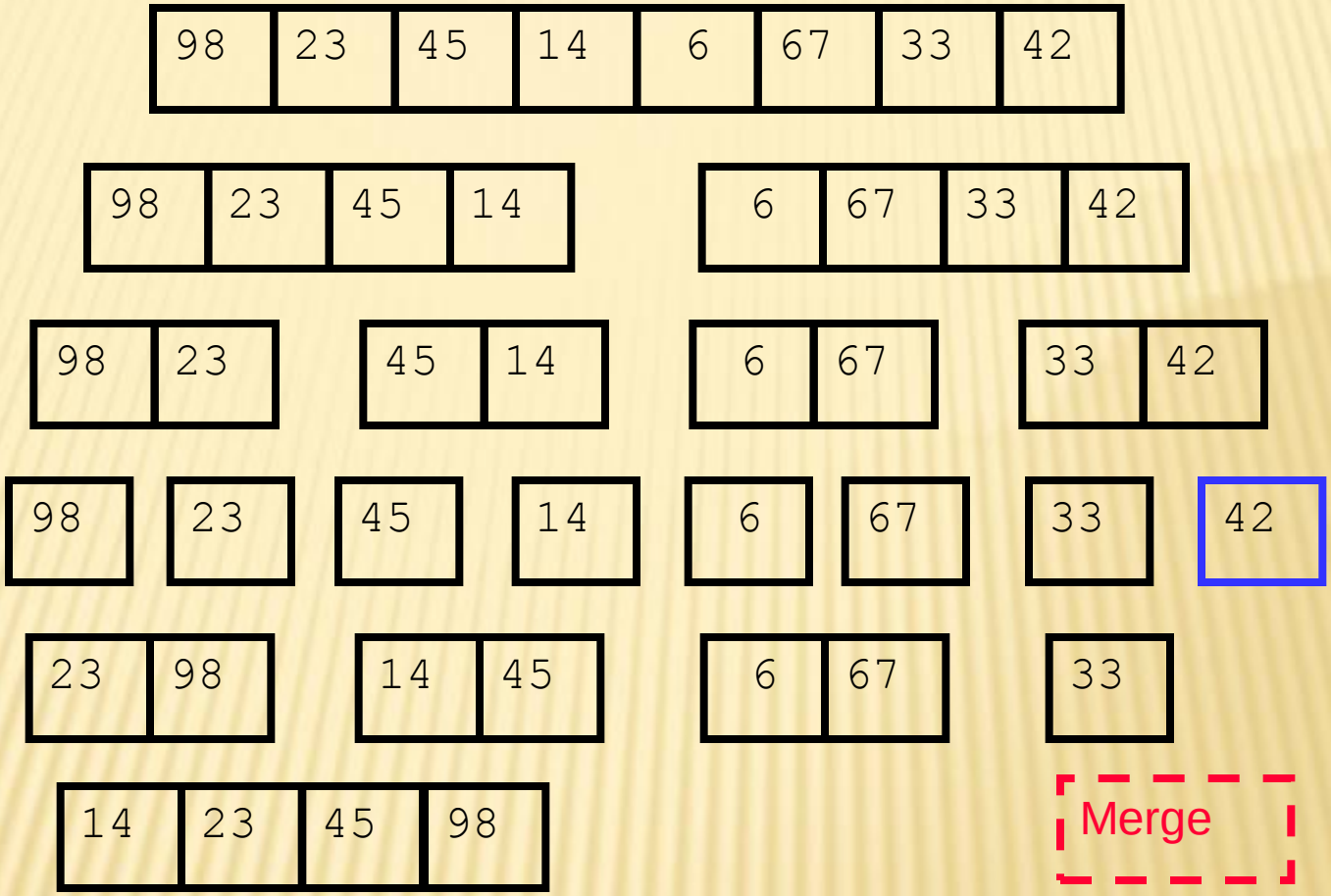

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

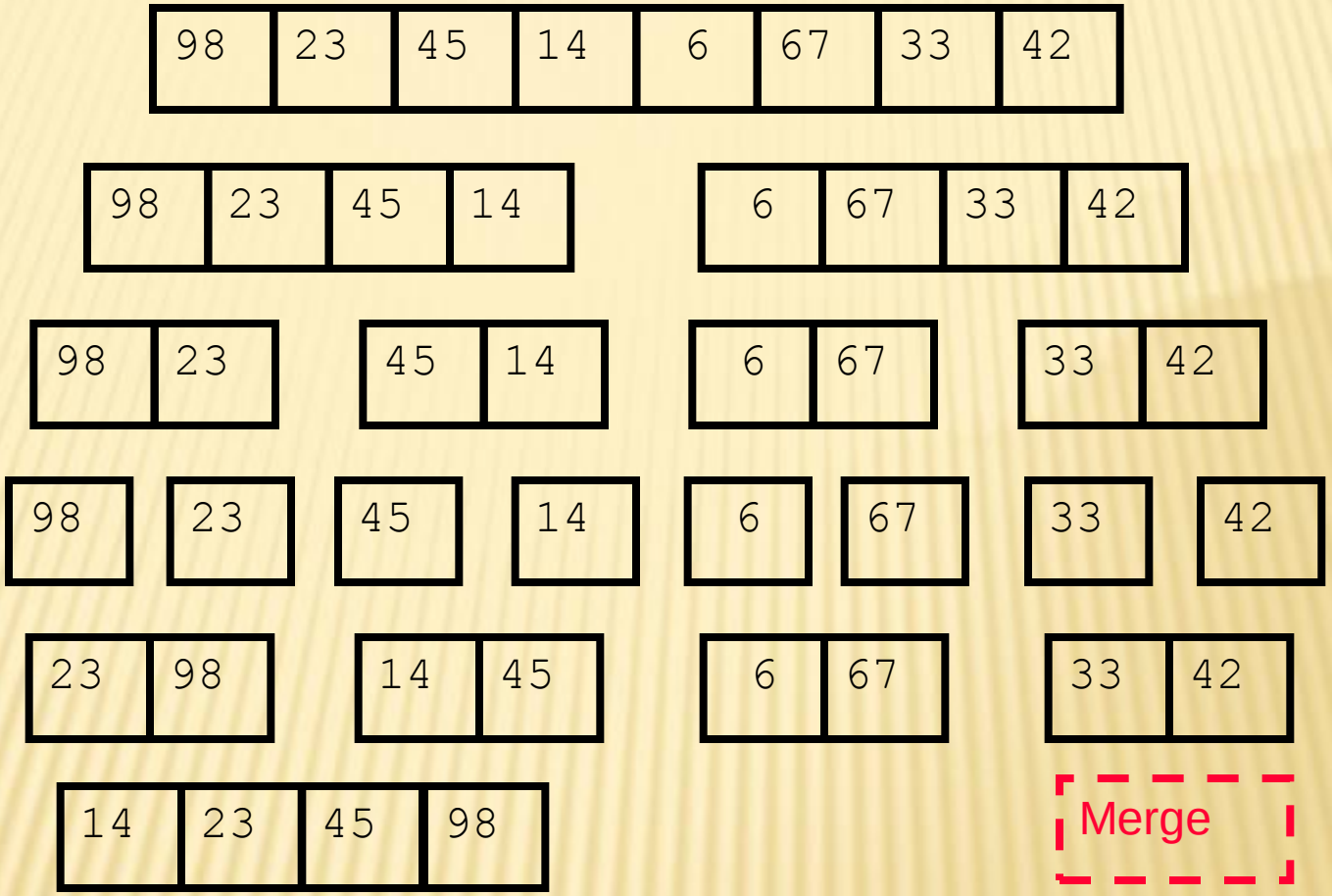

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge


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

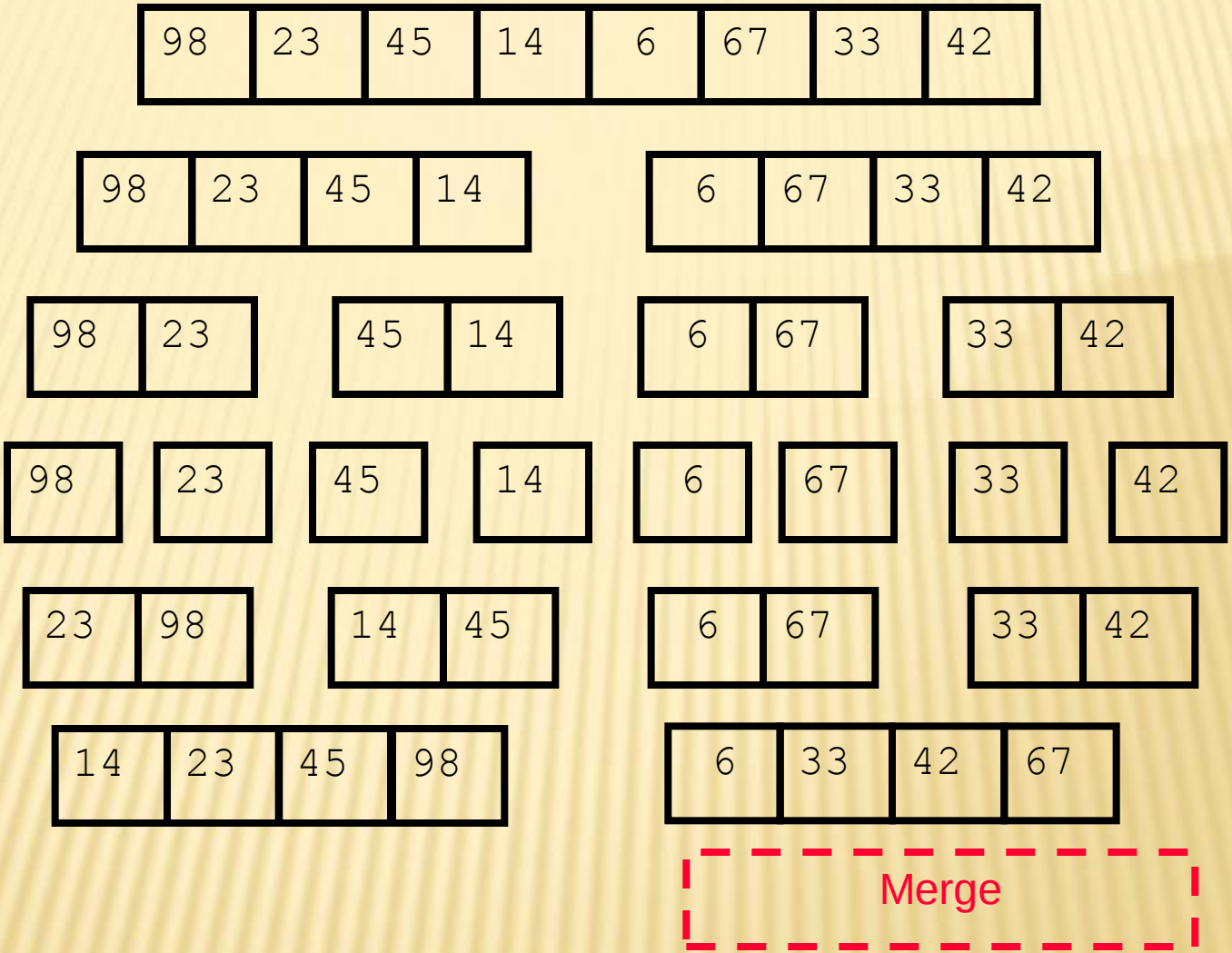

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

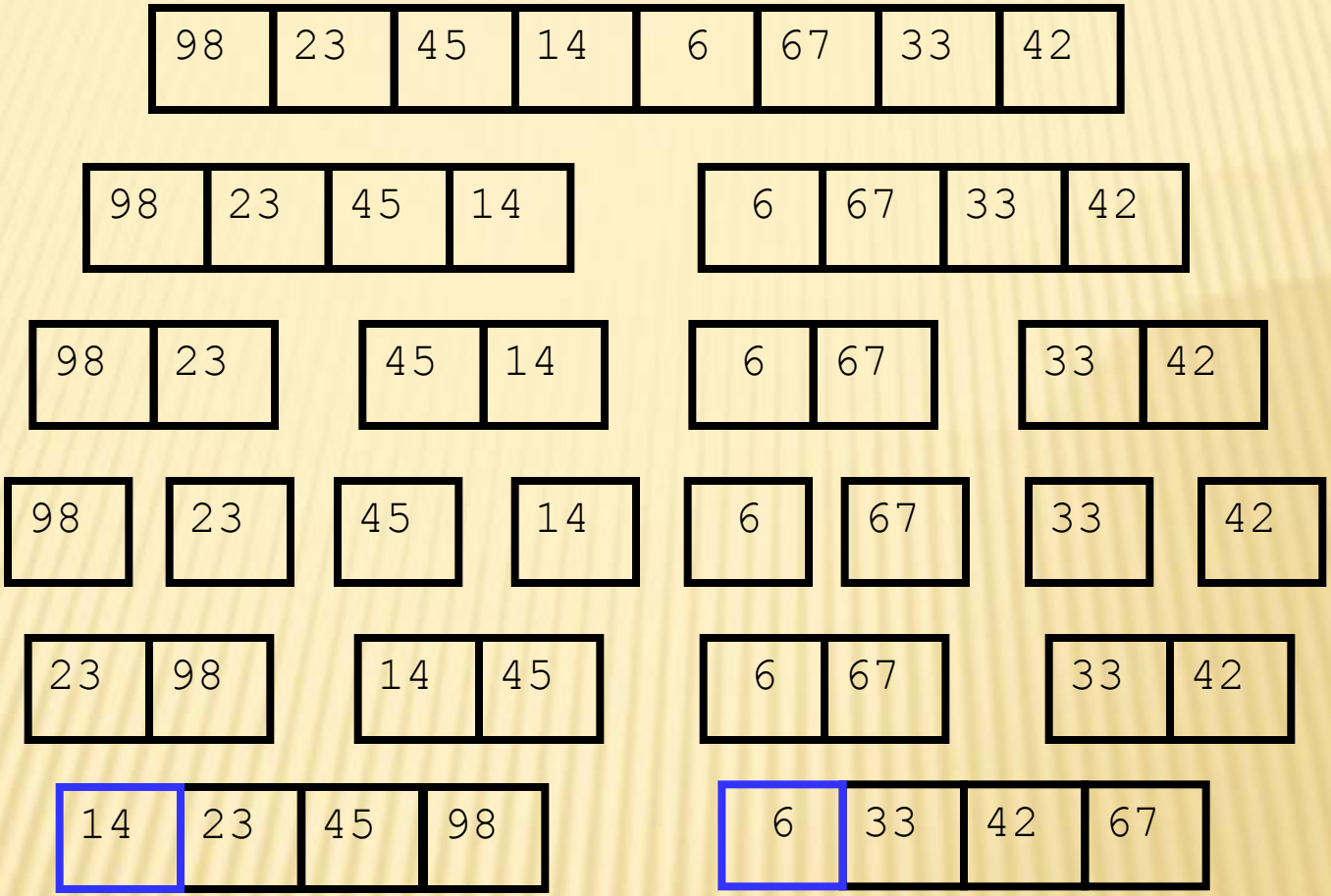

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

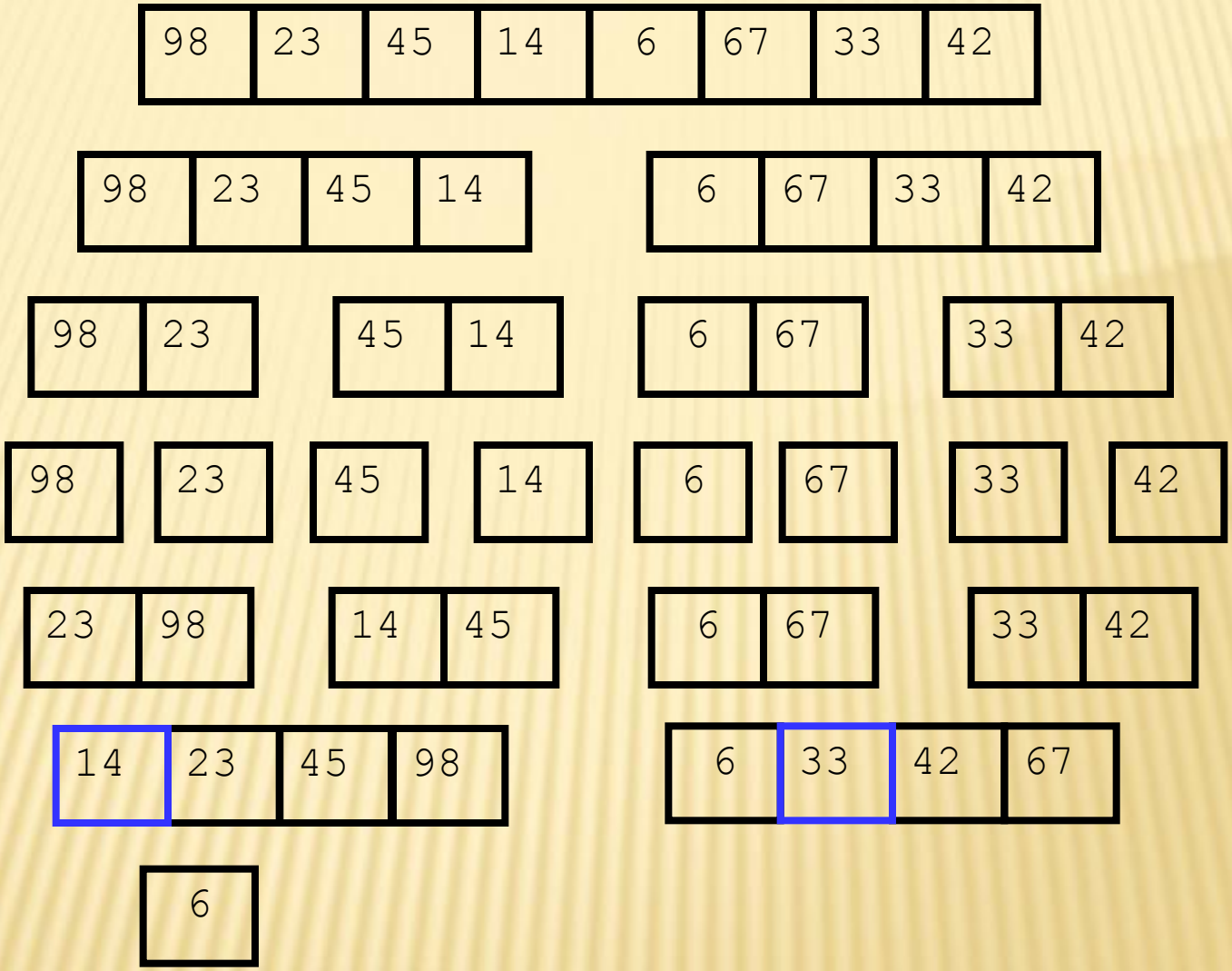

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6
Merge

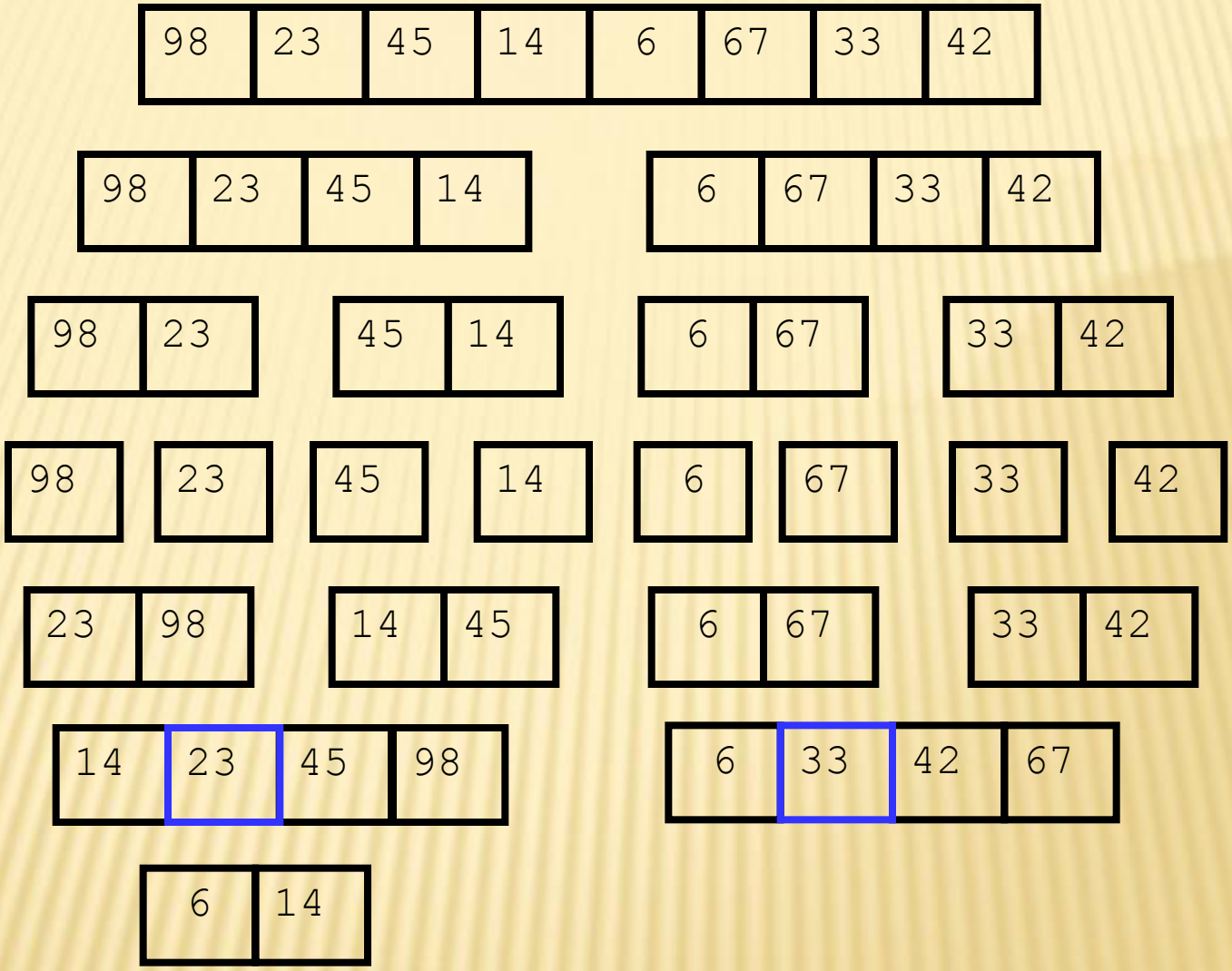

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33
Merge

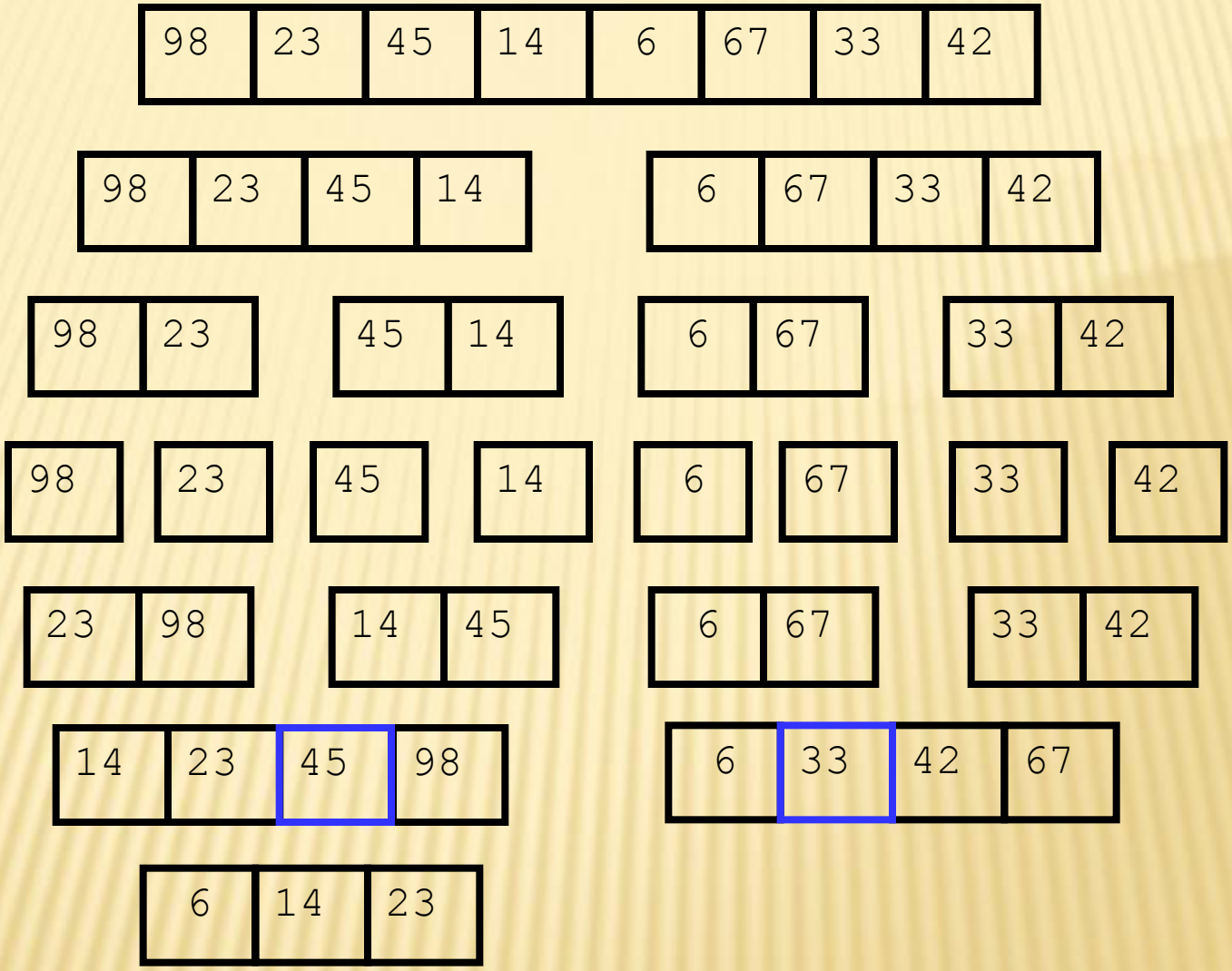

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42
Merge

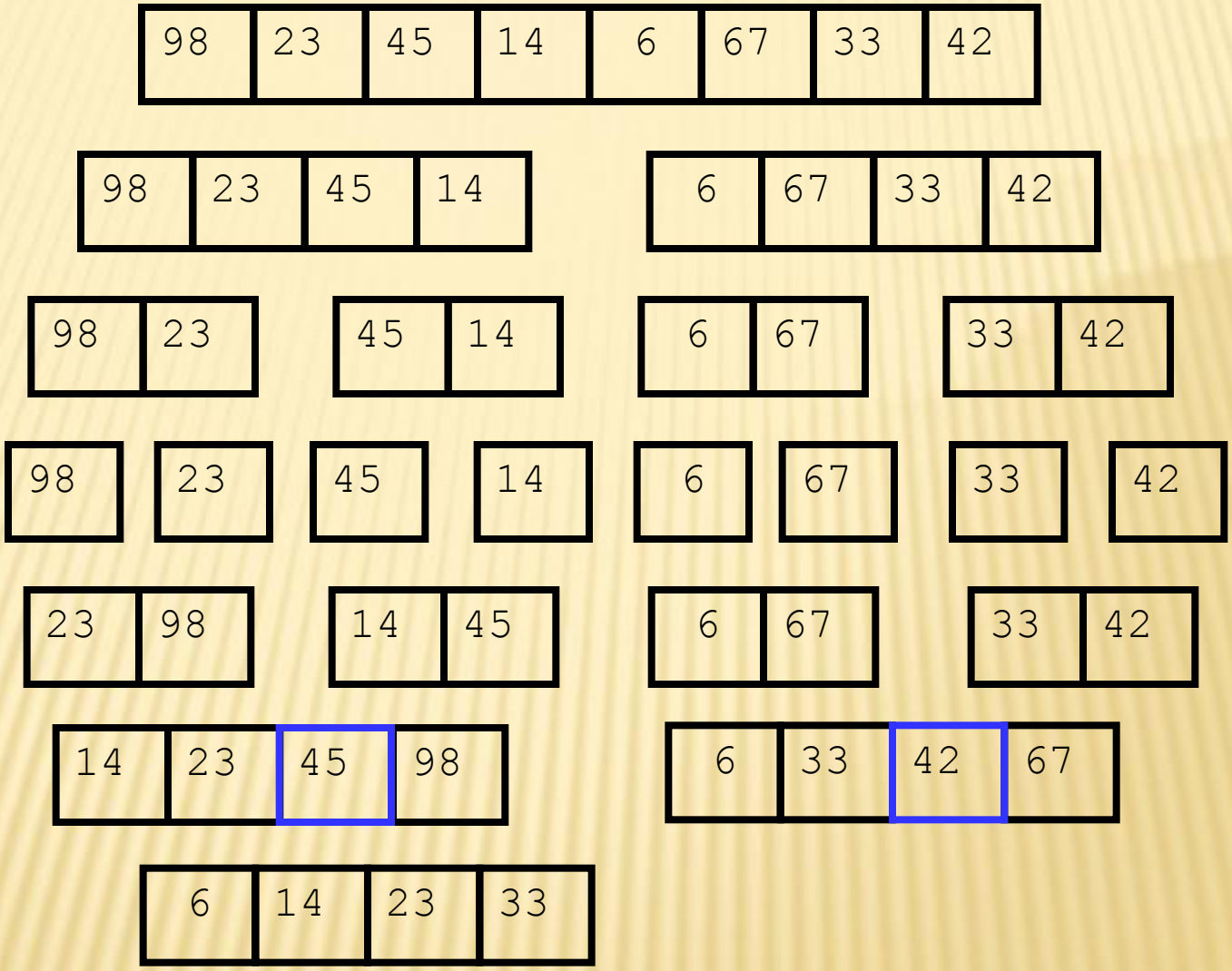

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

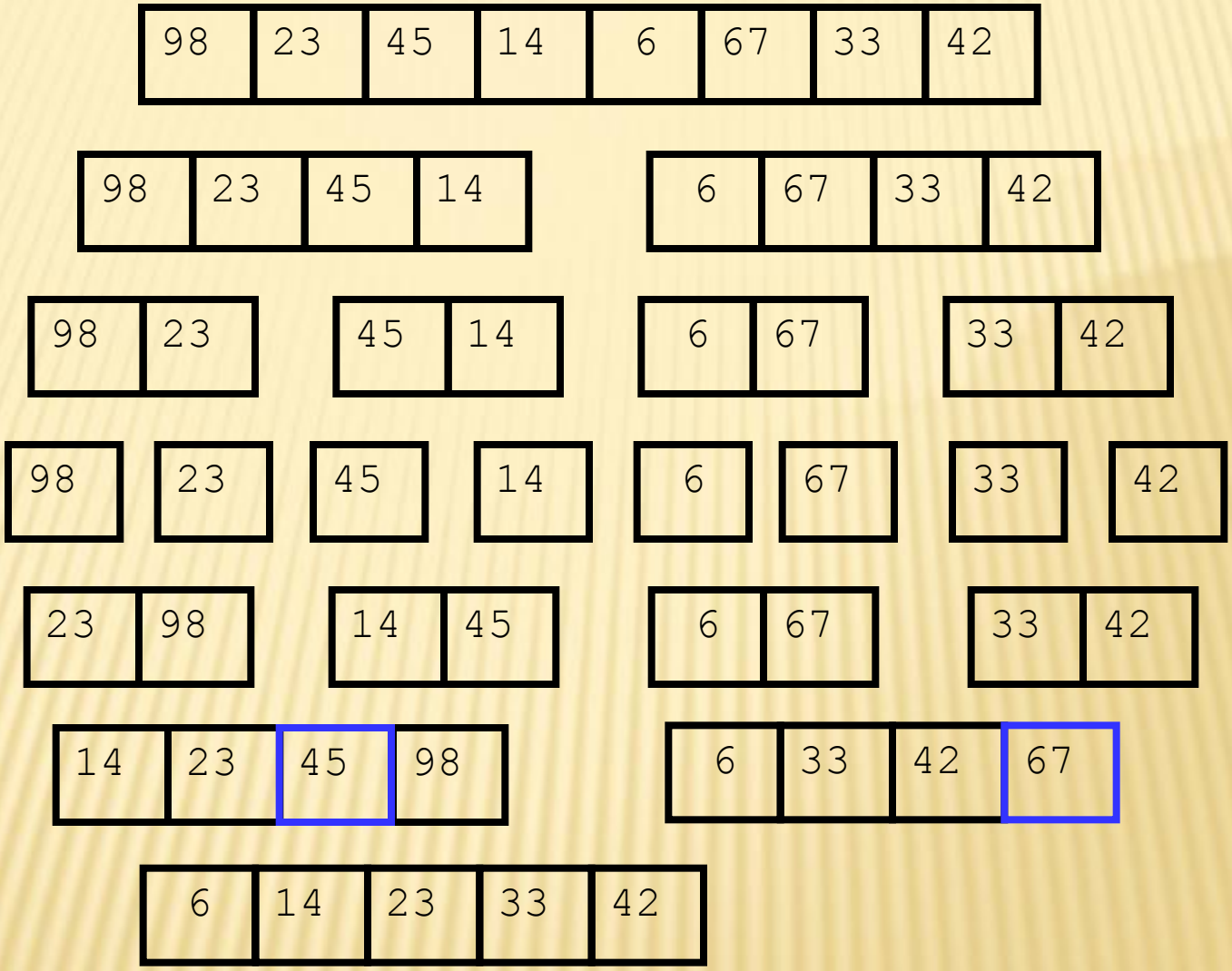

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

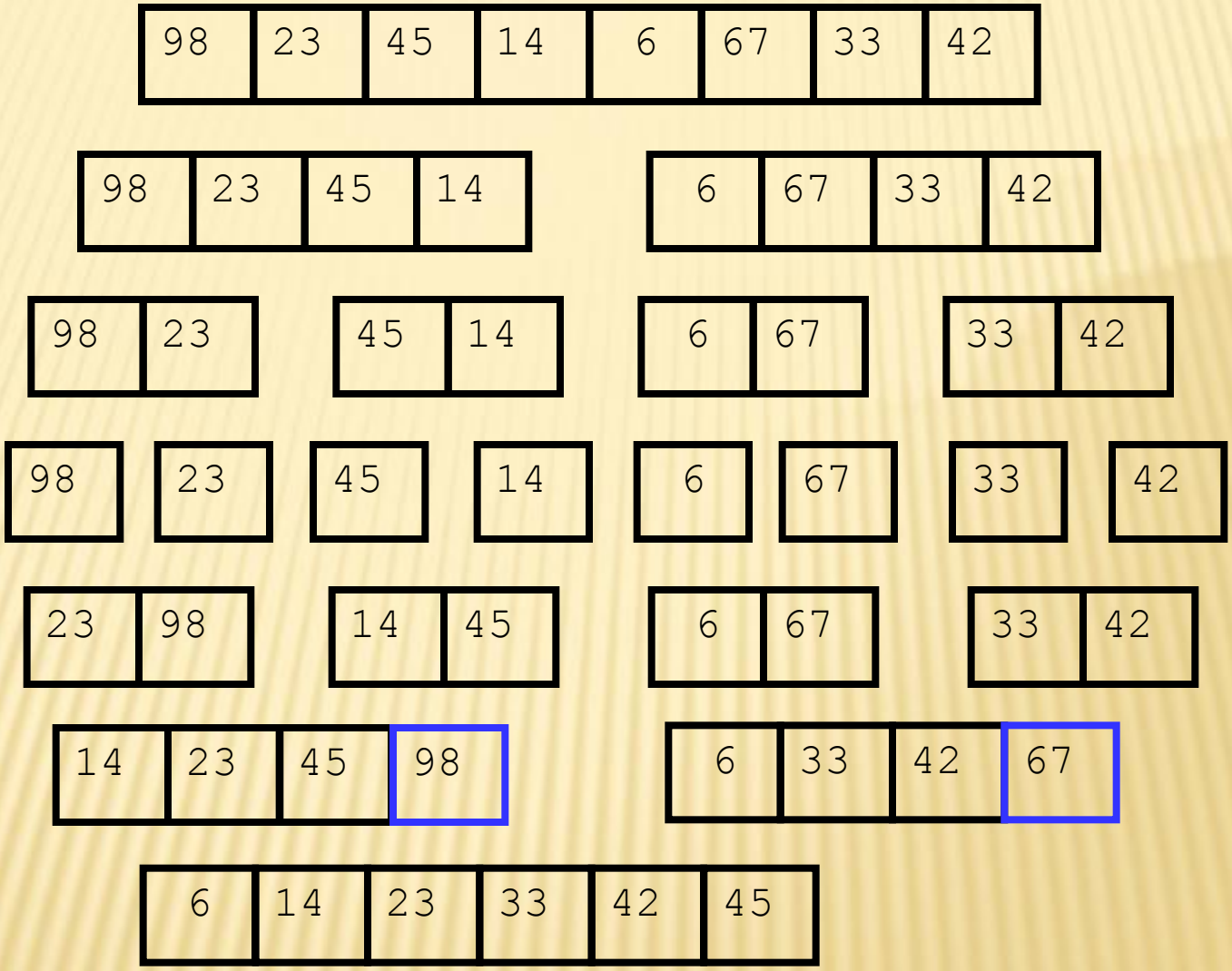

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6
Merge

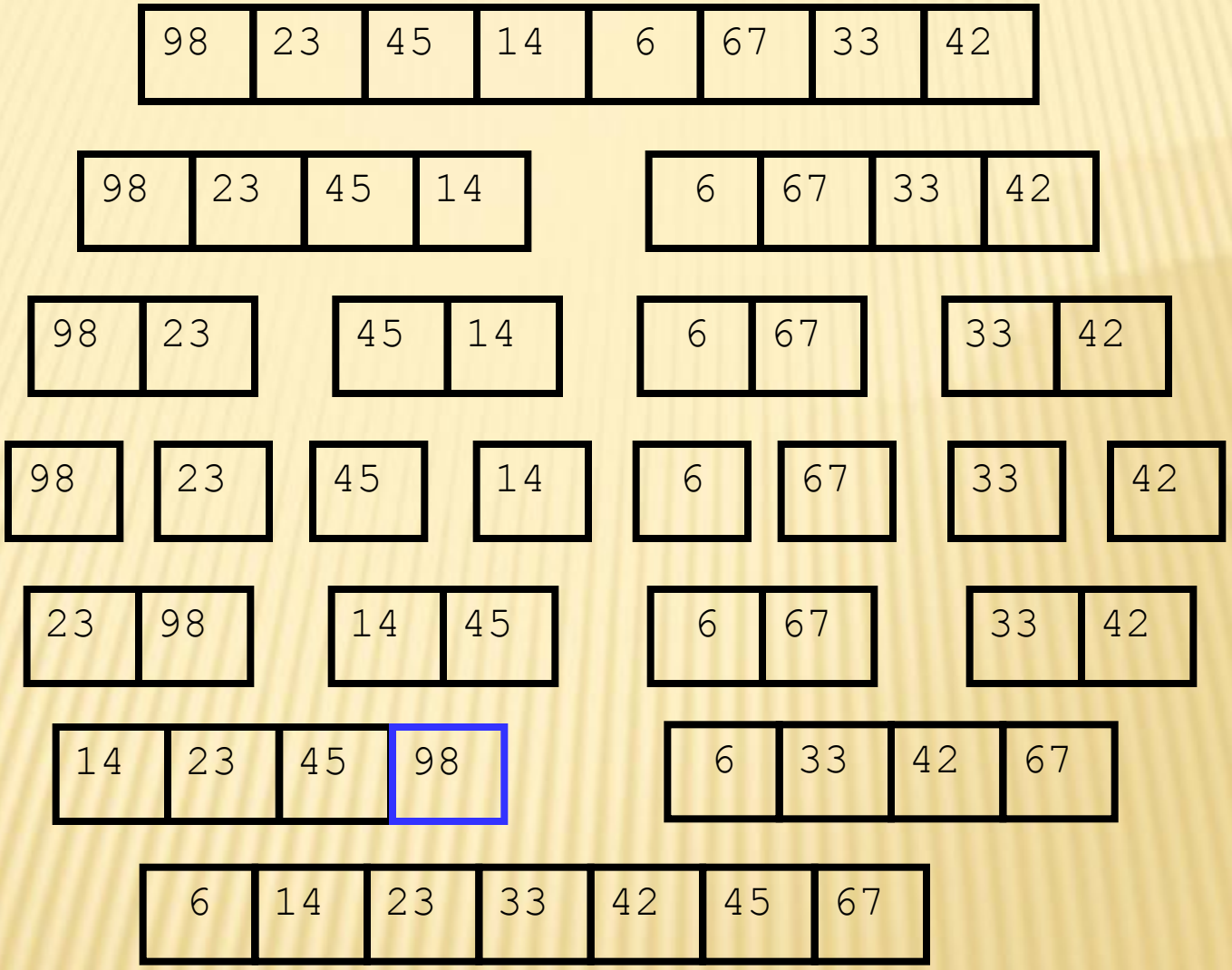

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14
Merge

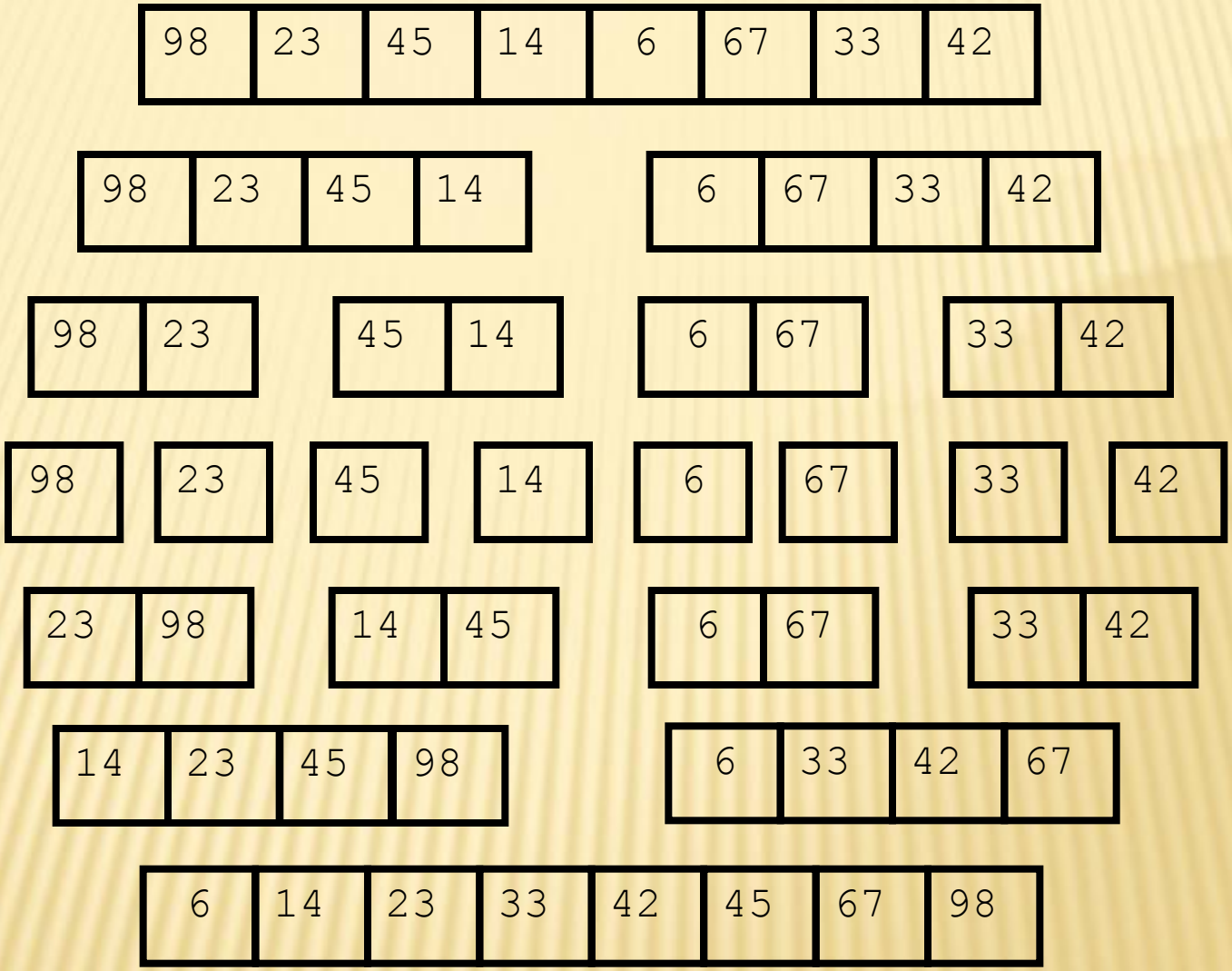

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23
Merge

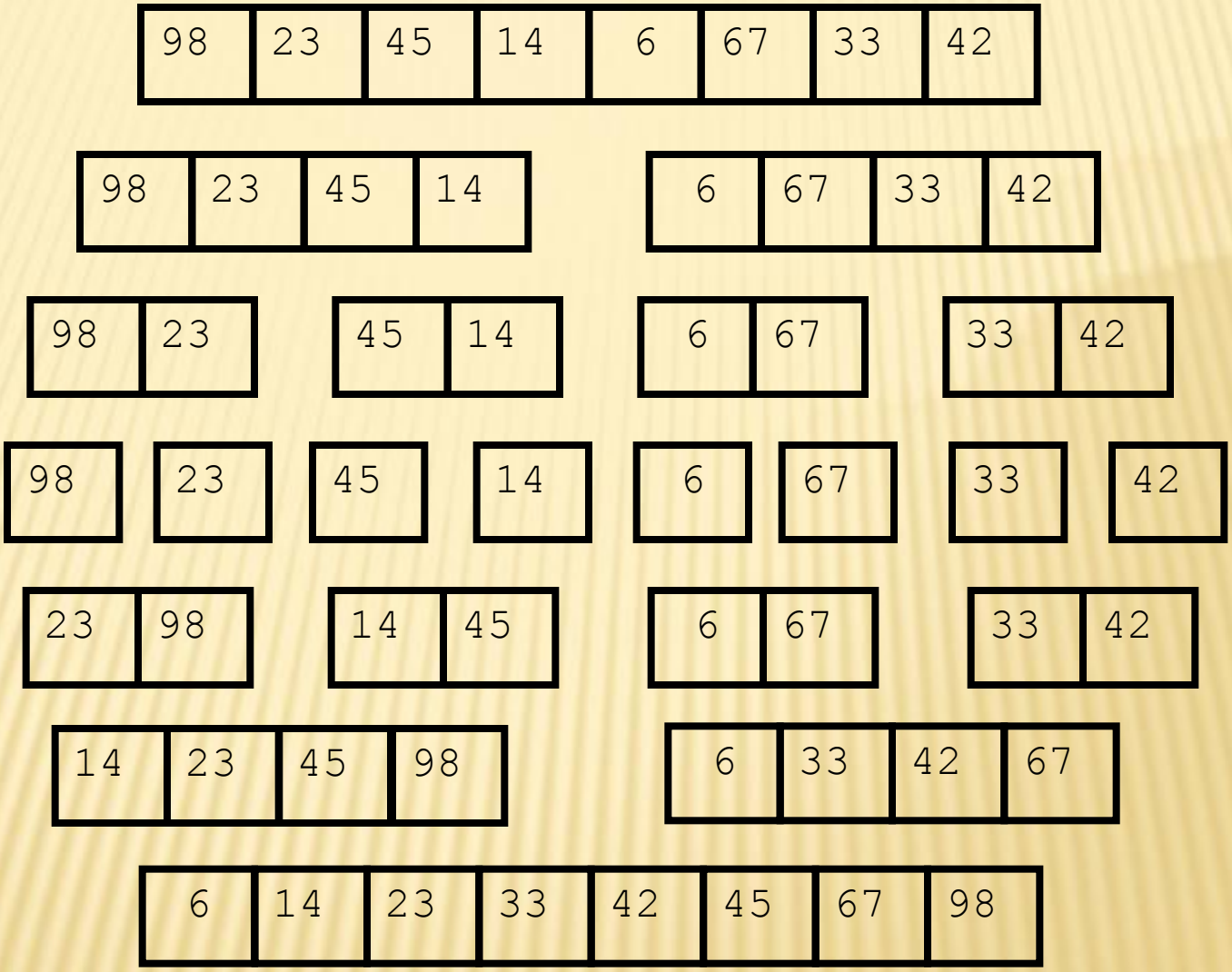

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33
Merge

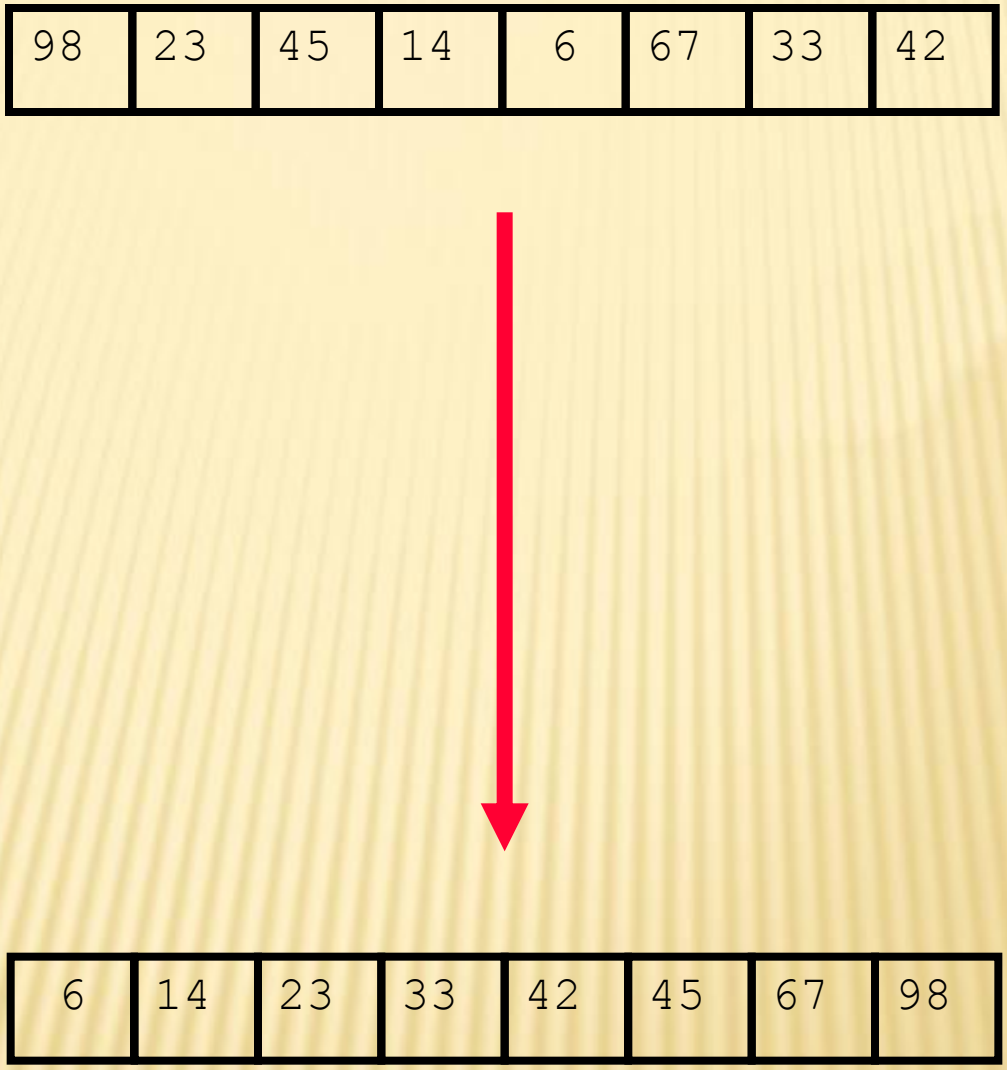

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42
Merge


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

QUICK SORT

- Quick sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.
- The steps are:
  - Pick an element, called a *pivot*, from the list.
  - Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way).
  - After this partitioning, the pivot is in its final position. This is called the partition operation.
  - Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

QUICK SORT

- Choose the appropriate pivot, either randomly or near the median of the array elements.
- Avoid a pivot which makes either of the two halves empty.

QUICK SORT

```
function quicksort(list q)
    list low, pivotList, hi
     if length(q) ≤ 1
          return q
     select a pivot value from q
     for each x in q except the pivot element
          if x < pivot then add x to low
          if x ≥ pivot then add x to high
     add pivot to pivotList
   return concatenate(quicksort(less), pivotList,
    quicksort(greater))
```

EXECUTION

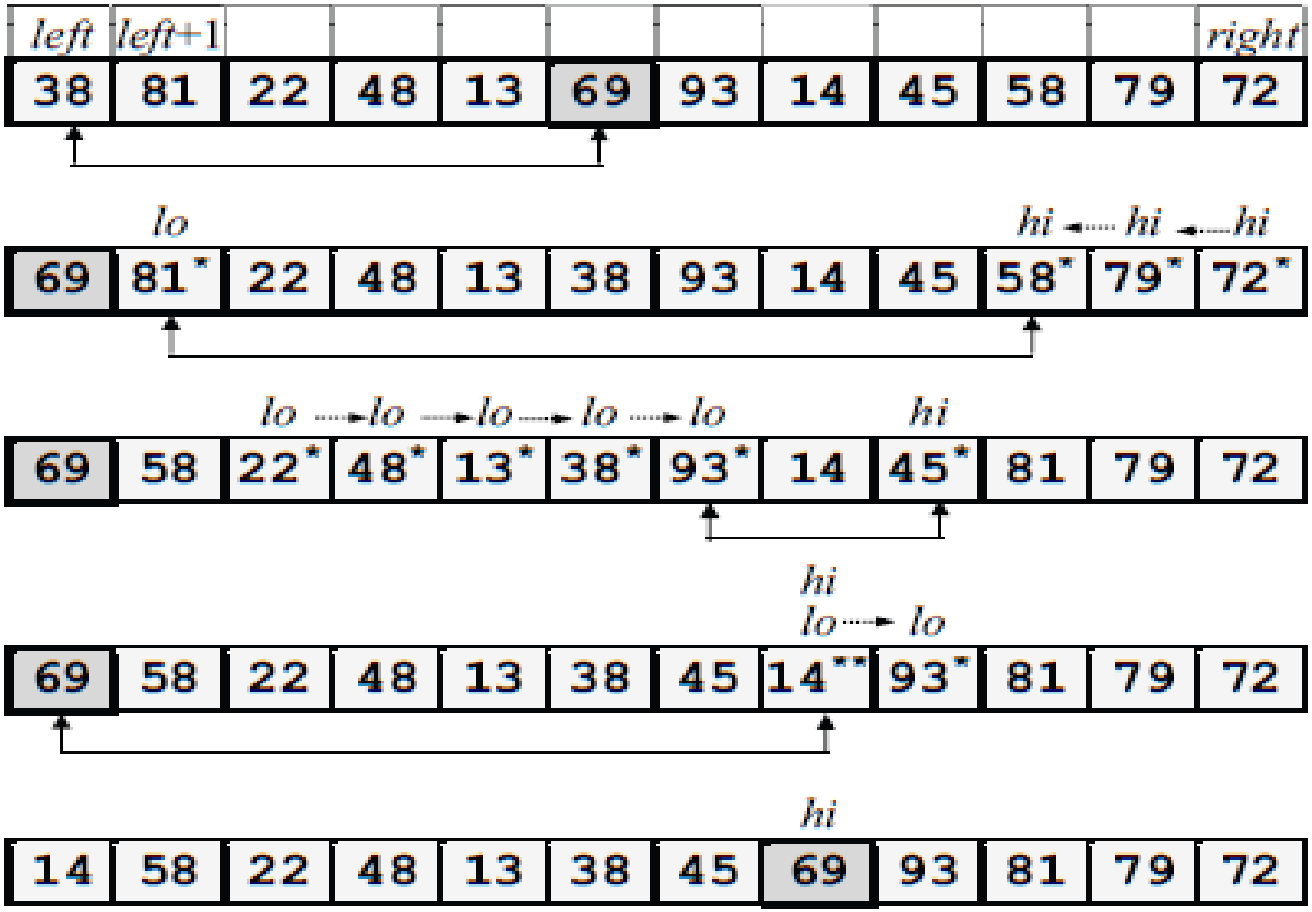
Swap pivot element with leftmost element. *lo*=*left*+1; *hi*=*right*;

Move *hi* left and *lo* right as far as we can; then swap *A*[*lo*] and *A*[*hi*], and move *hi* and *lo* one more position.

Repeat above

Repeat above until *hi* and *lo* cross; then *hi* is the final position of the pivot element, so swap *A*[*hi*] and *A*[*left*].

Partitioning complete; return value of *hi*.



Execution

Here is the tree of recursive calls to quicksort. Calls to sort subarrays of size 0 or 1 are not shown. (They could be omitted.)