Data Structures (15B11CI311)

Odd Semester 2020

3rd Semester , Computer Science and Engineering

Jaypee Institute Of Information Technology (JIIT), Noida

Sorting Techniques in Data Structure

Outlines:

- Divide and Conquer
- Merge Sort
- Quick Sort
- Radix Sort
- Count Sort
- Bucket Sort

Divide-and-Conquer

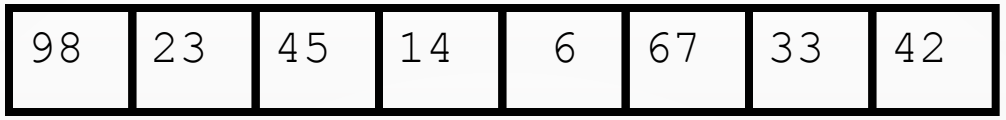
Divide and conquer is a strategy to solving a large problem by:

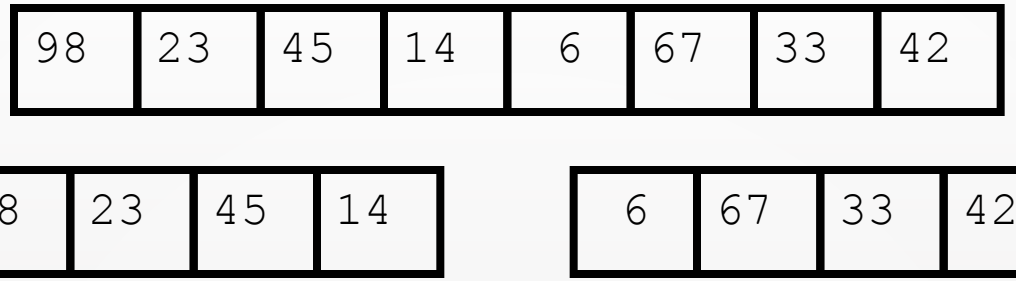
- Divide the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- Conquer the sub-problems
  - Solve the sub-problems recursively
  - Sub-problem size small enough solve the problems in straight forward manner
- Combine the solutions of the sub-problems
  - Obtain the solution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

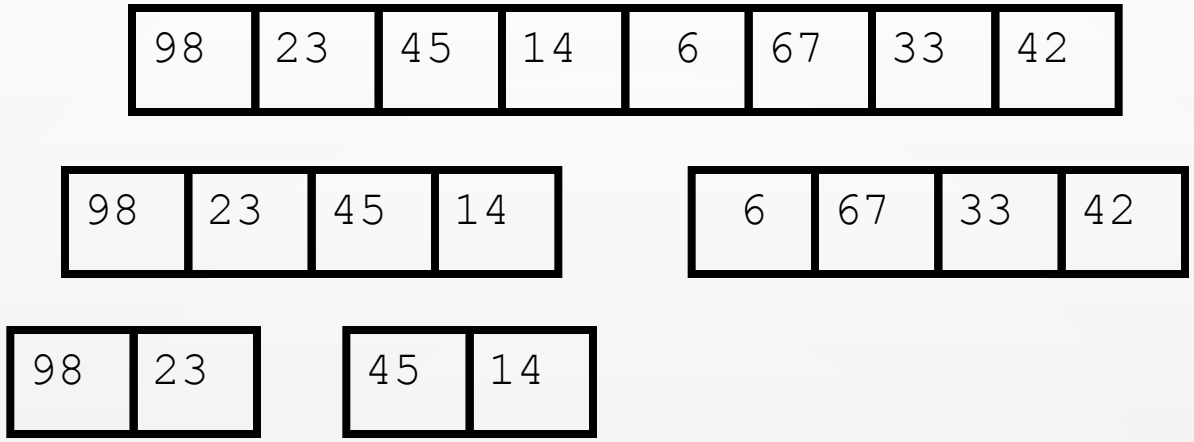
Merge Sort

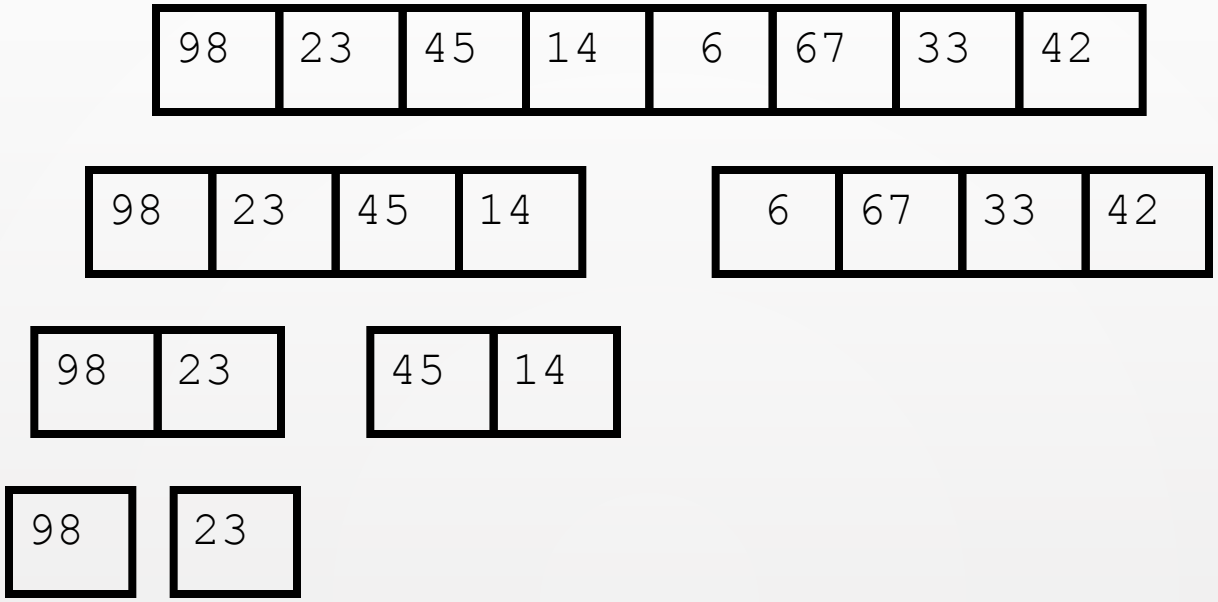
- Merge sort is a one of the efficient sorting algorithm.
- Merge sort work on a principle of divide-and-conquer algorithm.
- Merge sort repeatedly breaks down an array into several sub array until each sub array consists of a single element
- Merge the sub-problem solutions together:
  - Compare the 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

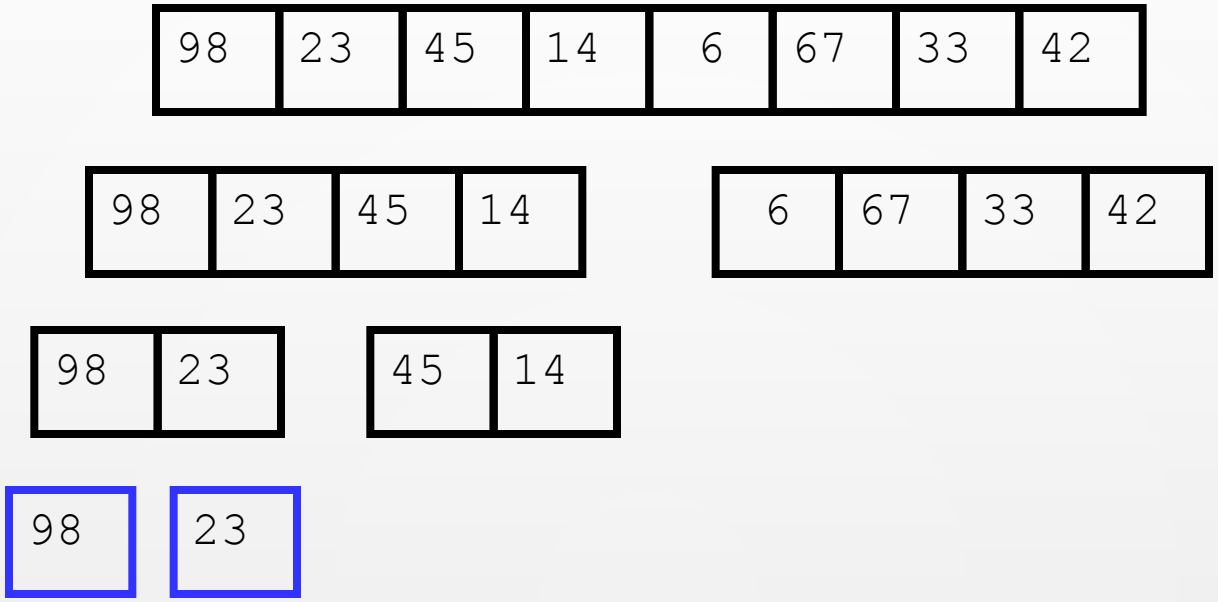
Illustration of Merge Sort

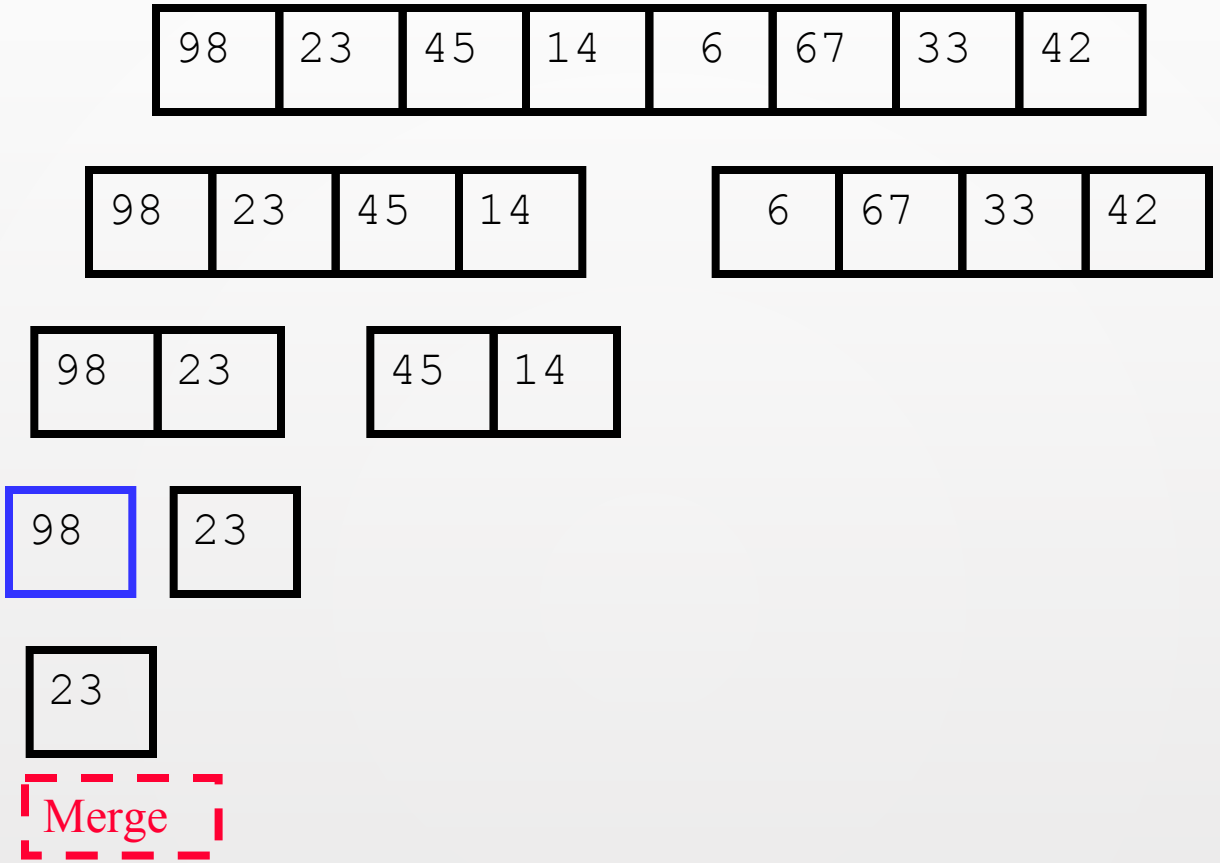


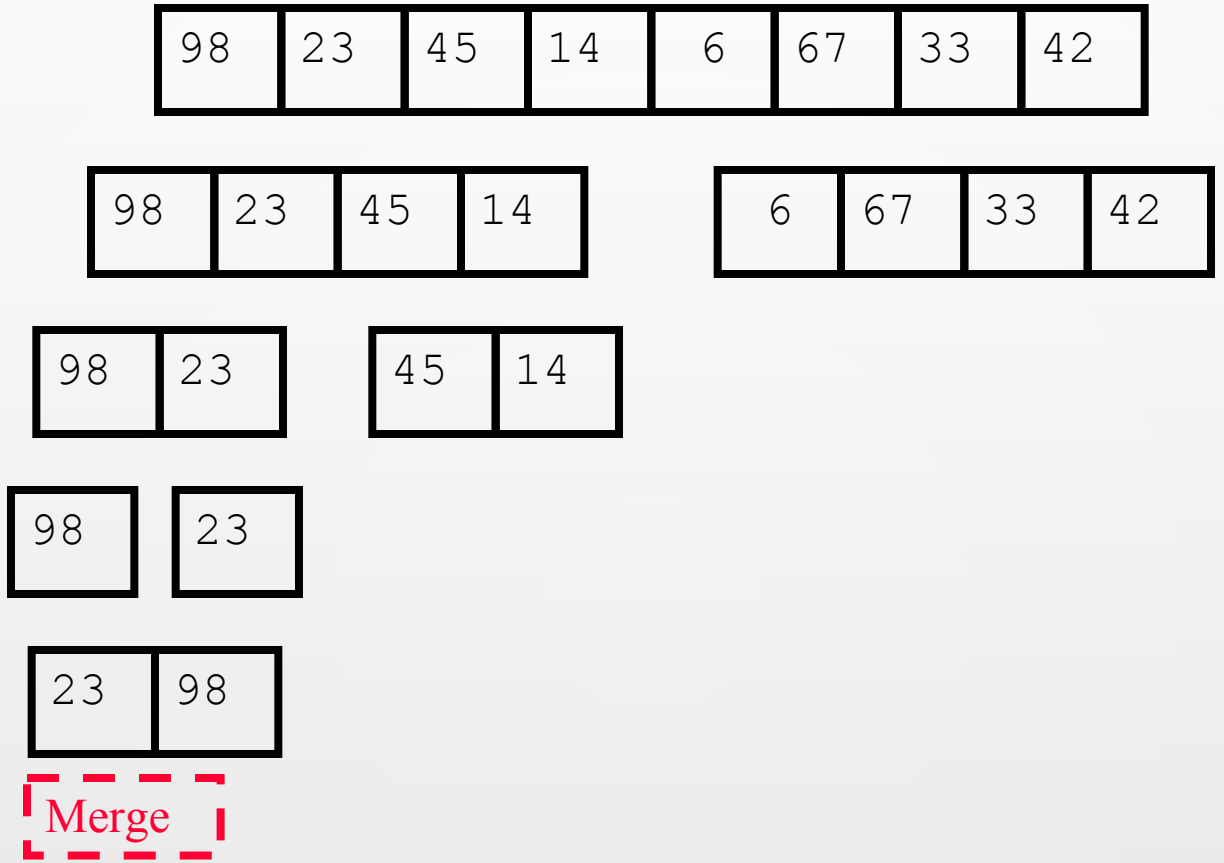

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42

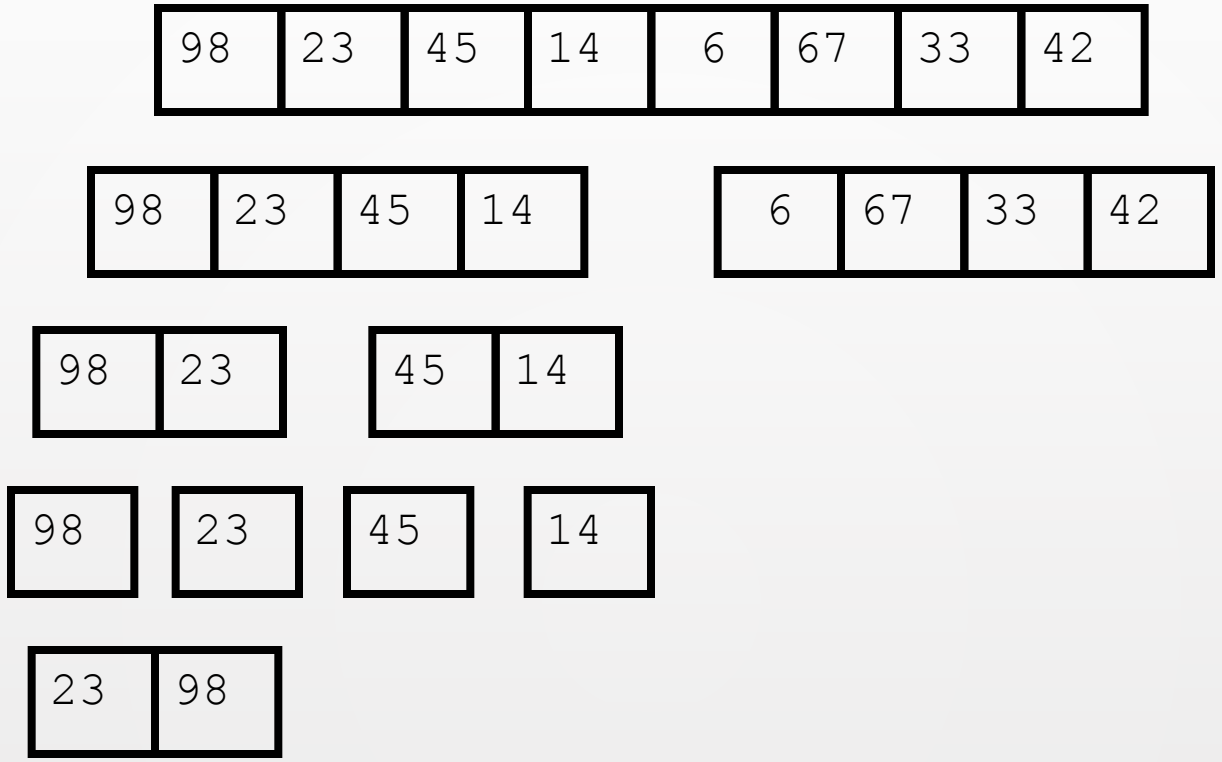

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

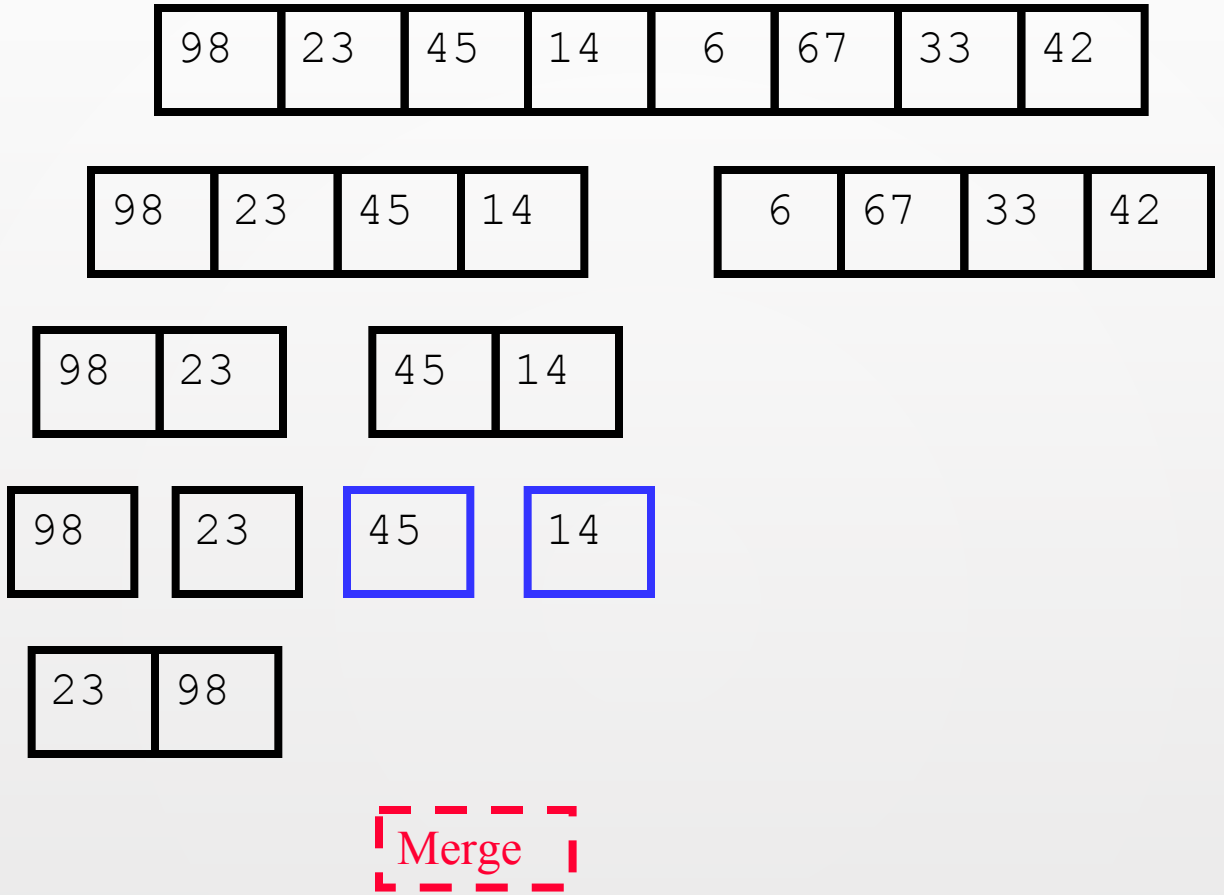

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

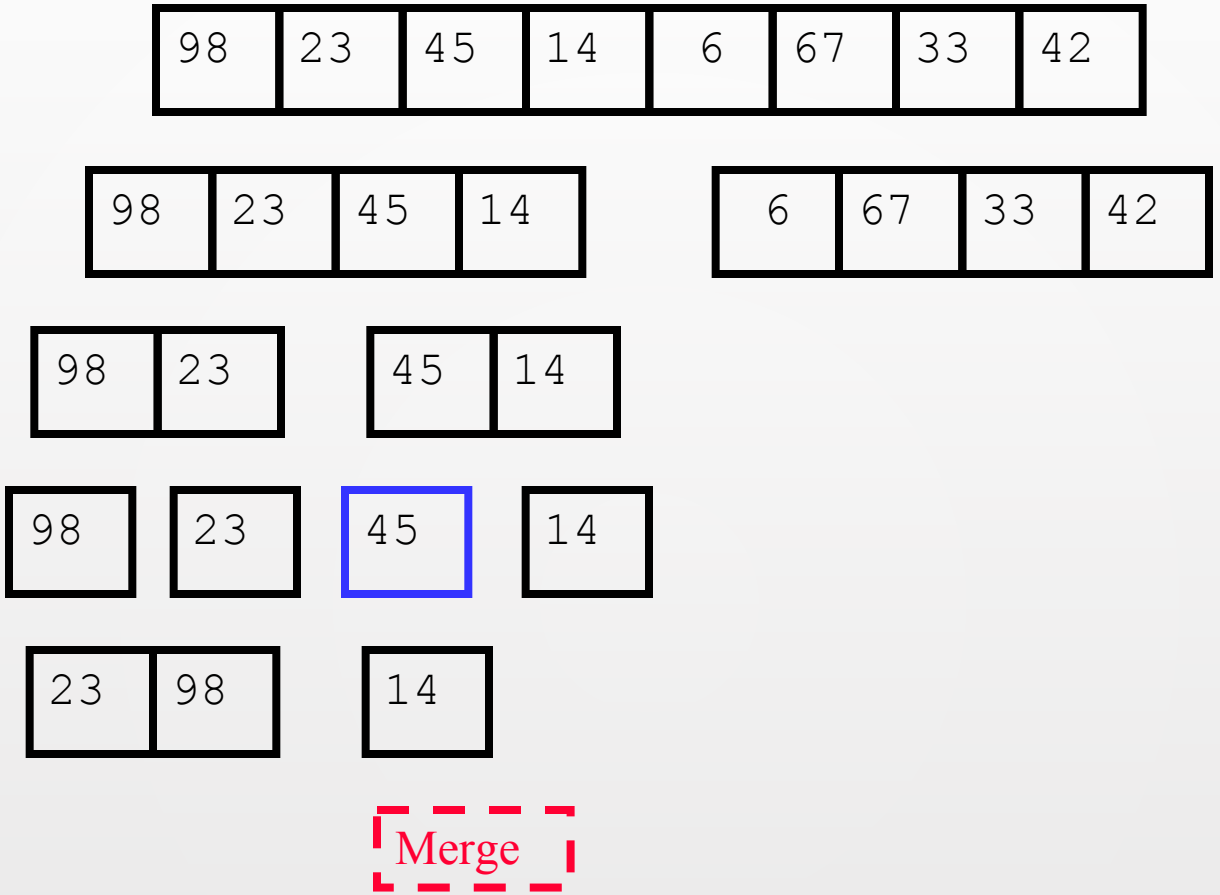

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

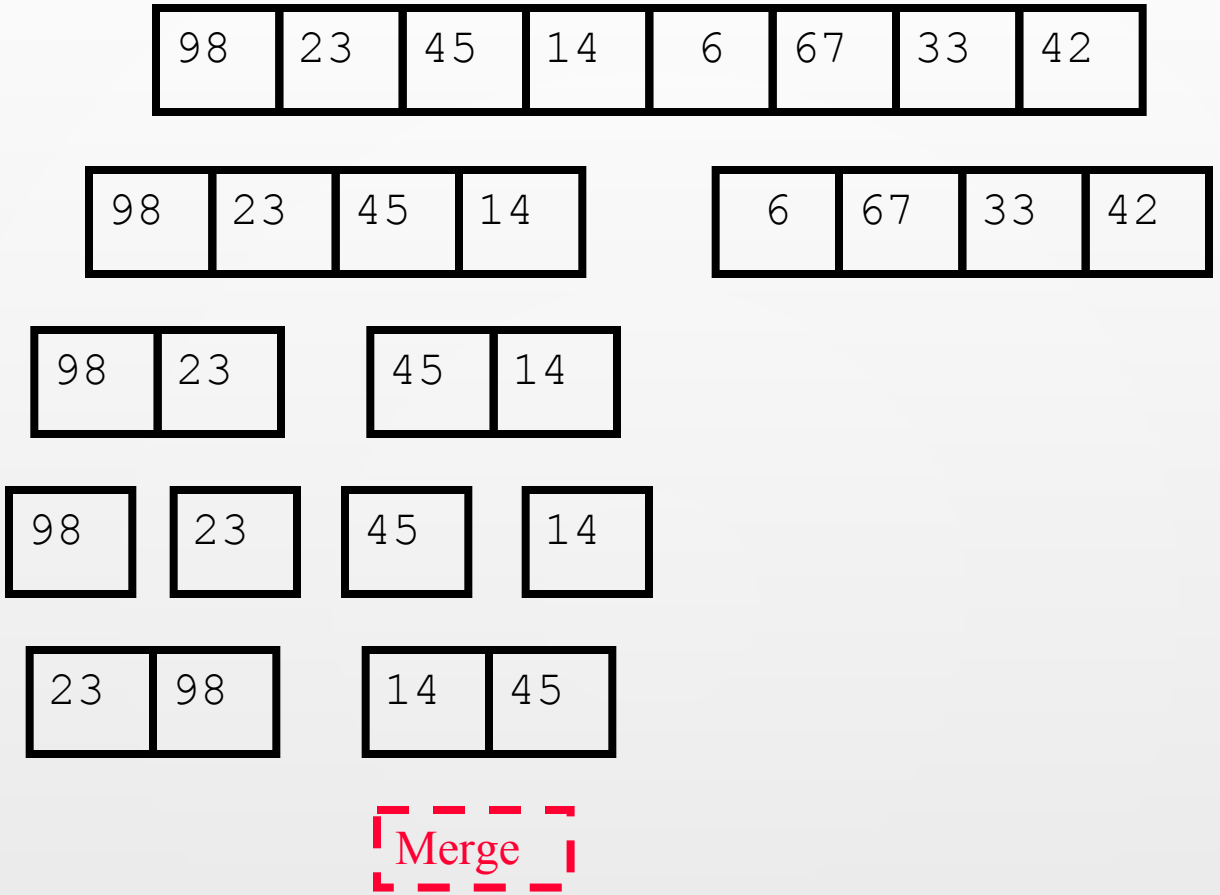

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

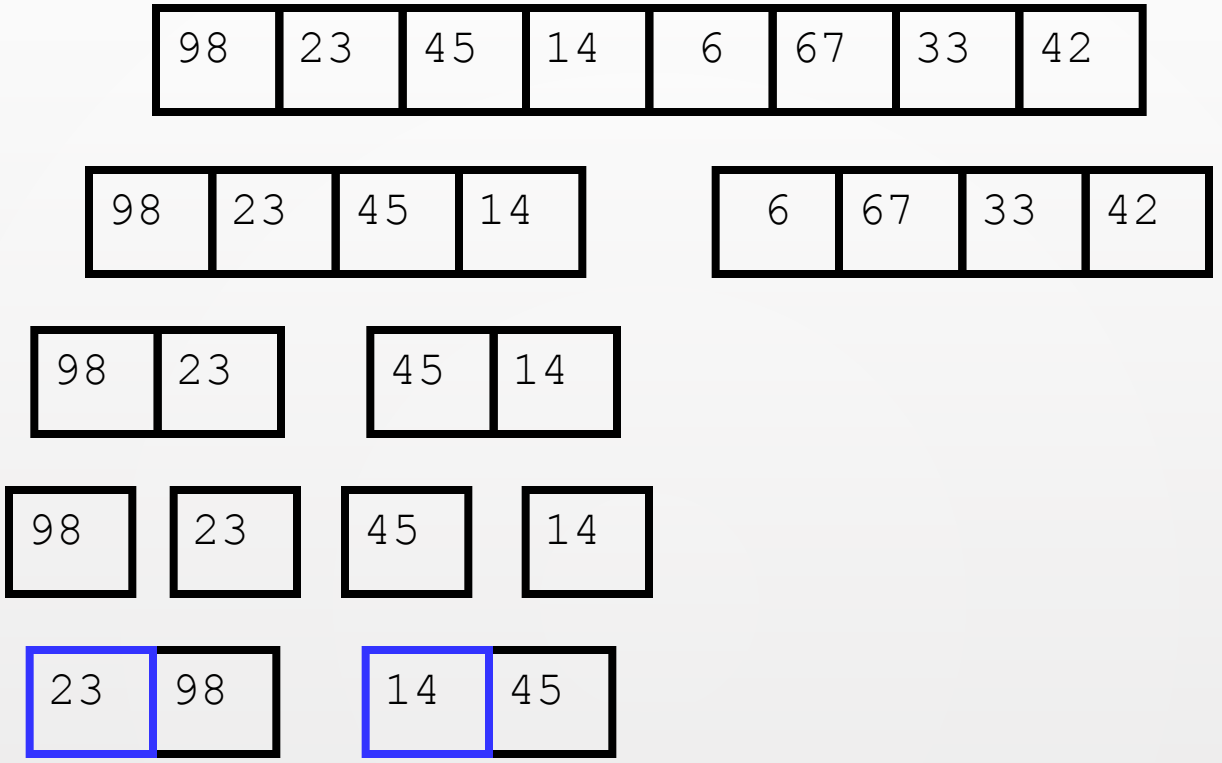

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge


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

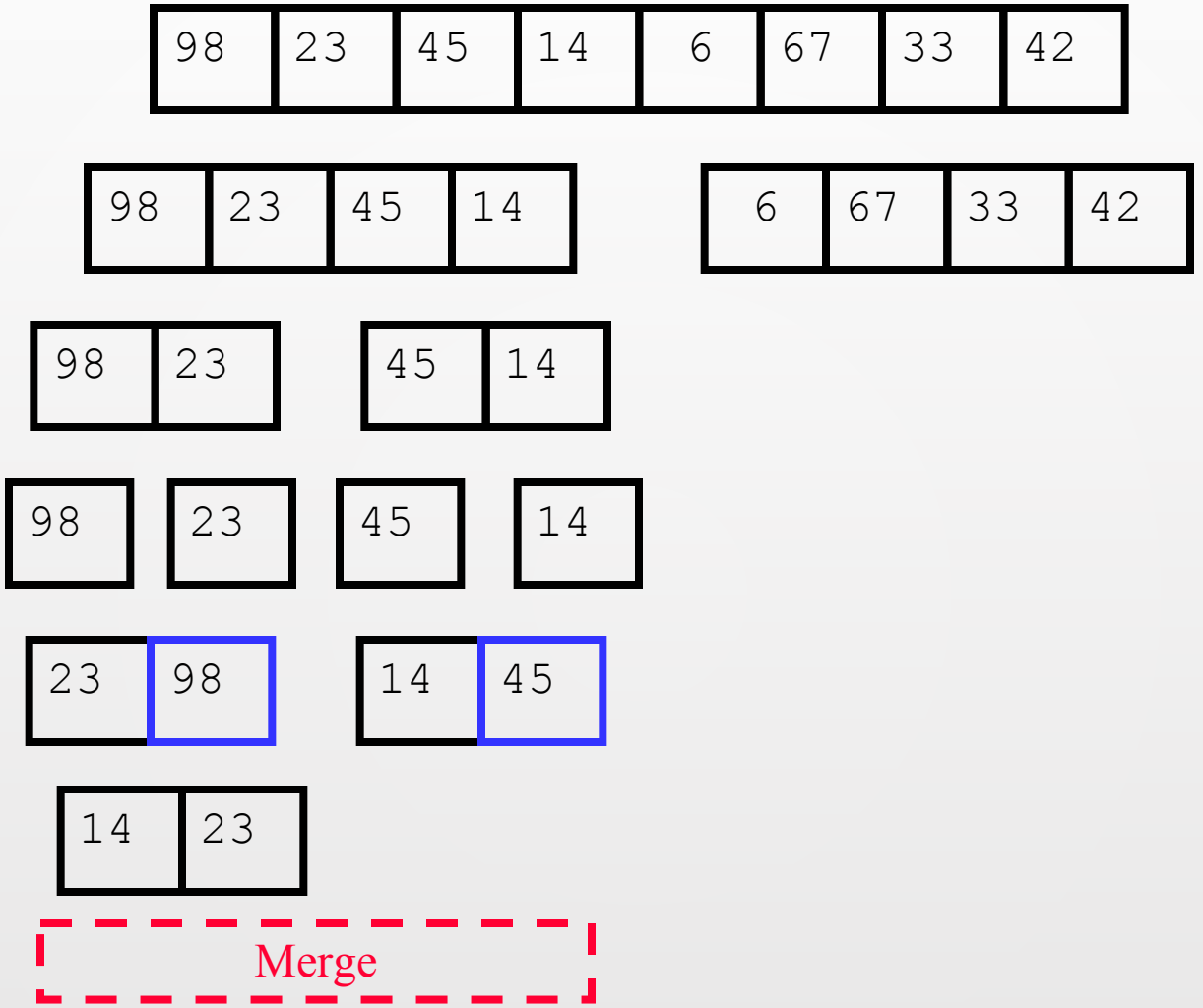

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
Merge


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

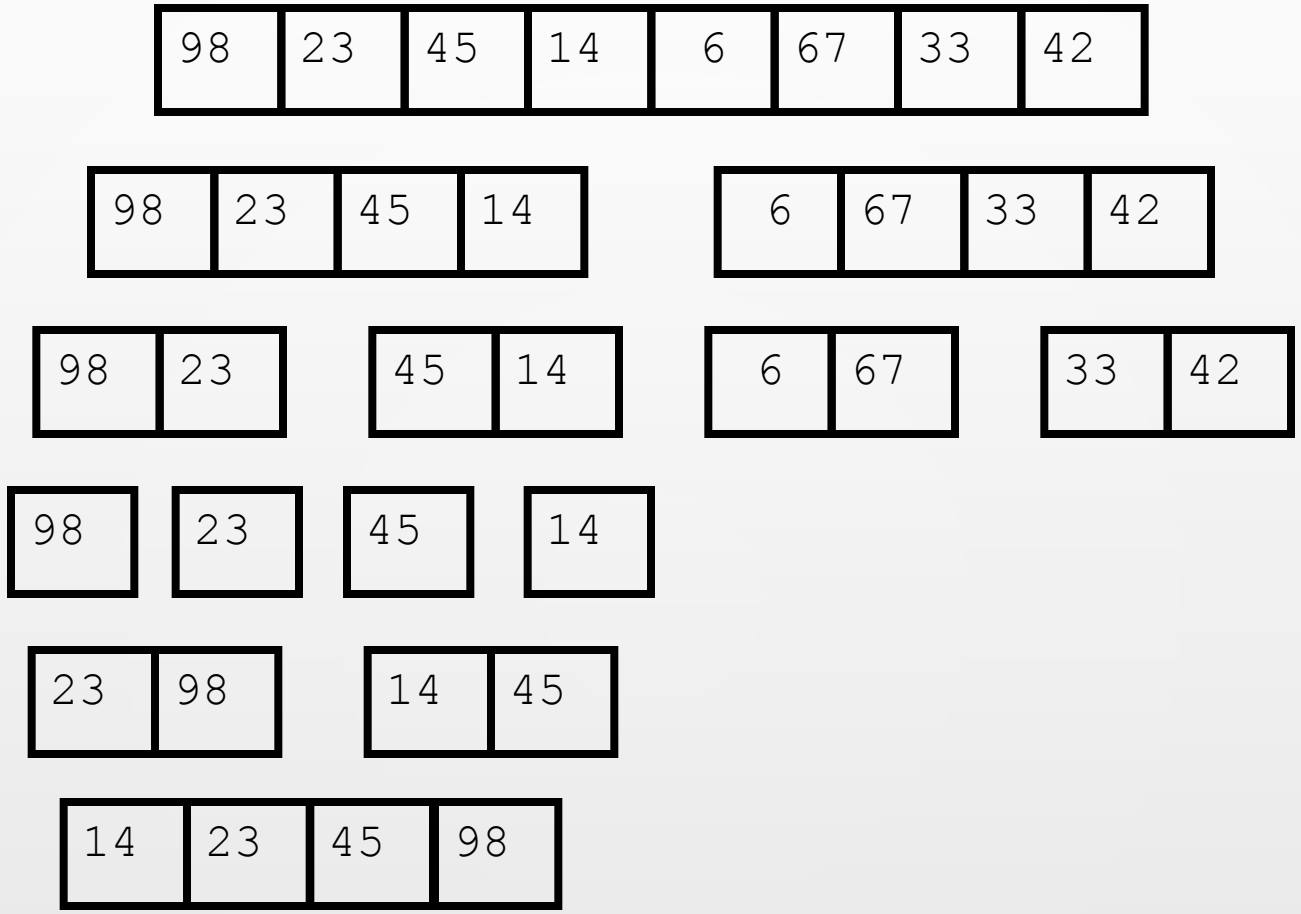

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

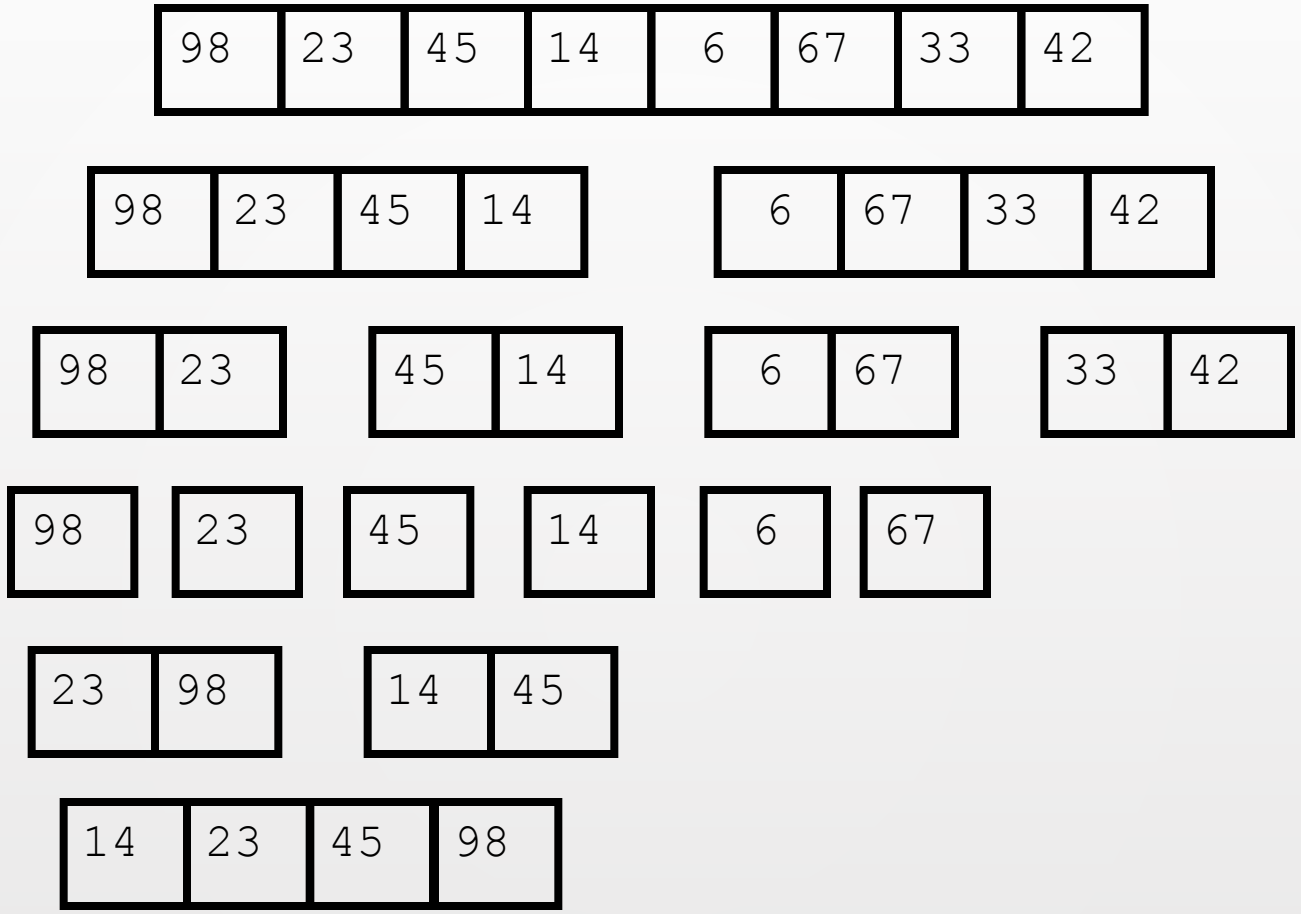

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge


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

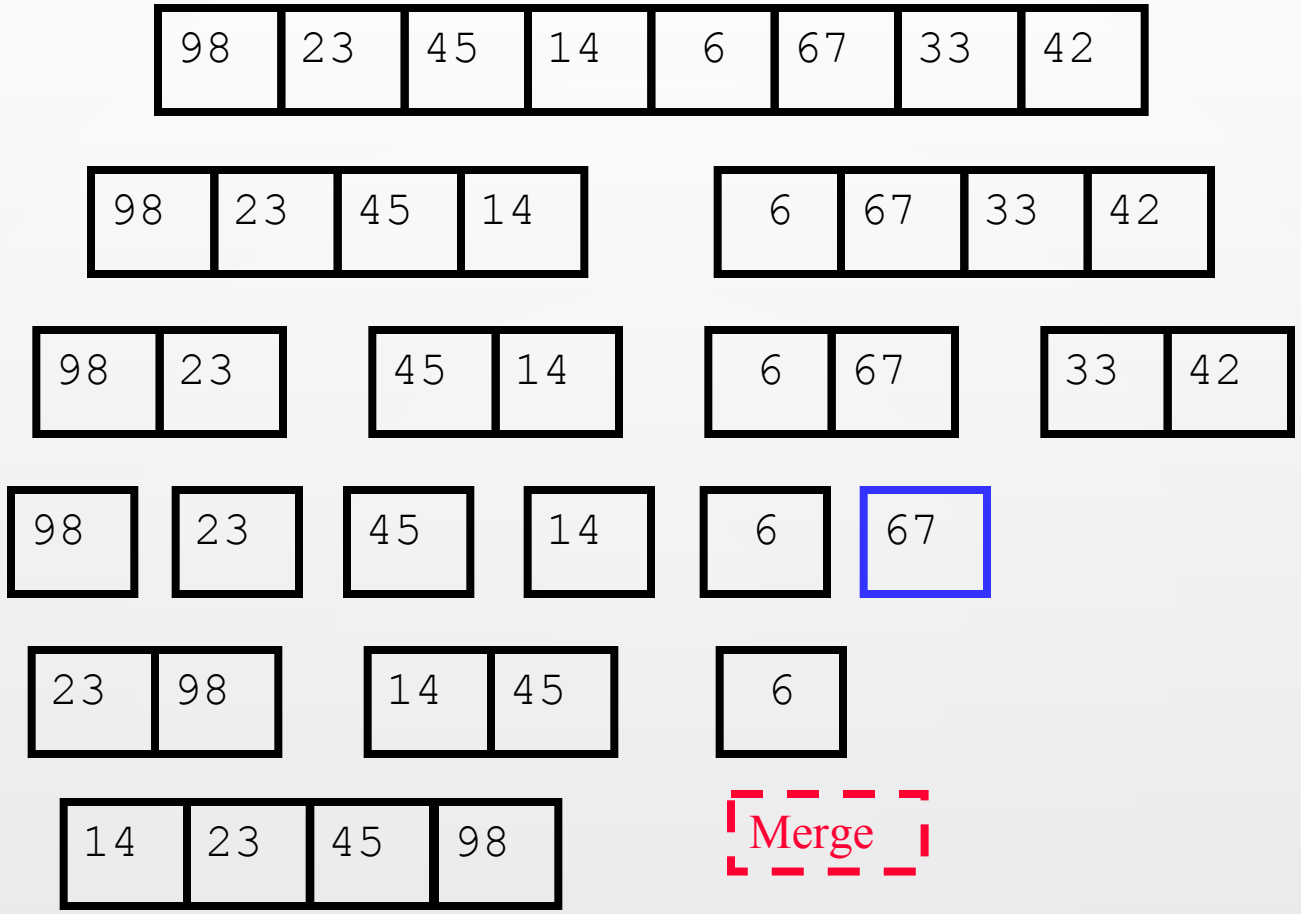

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge


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

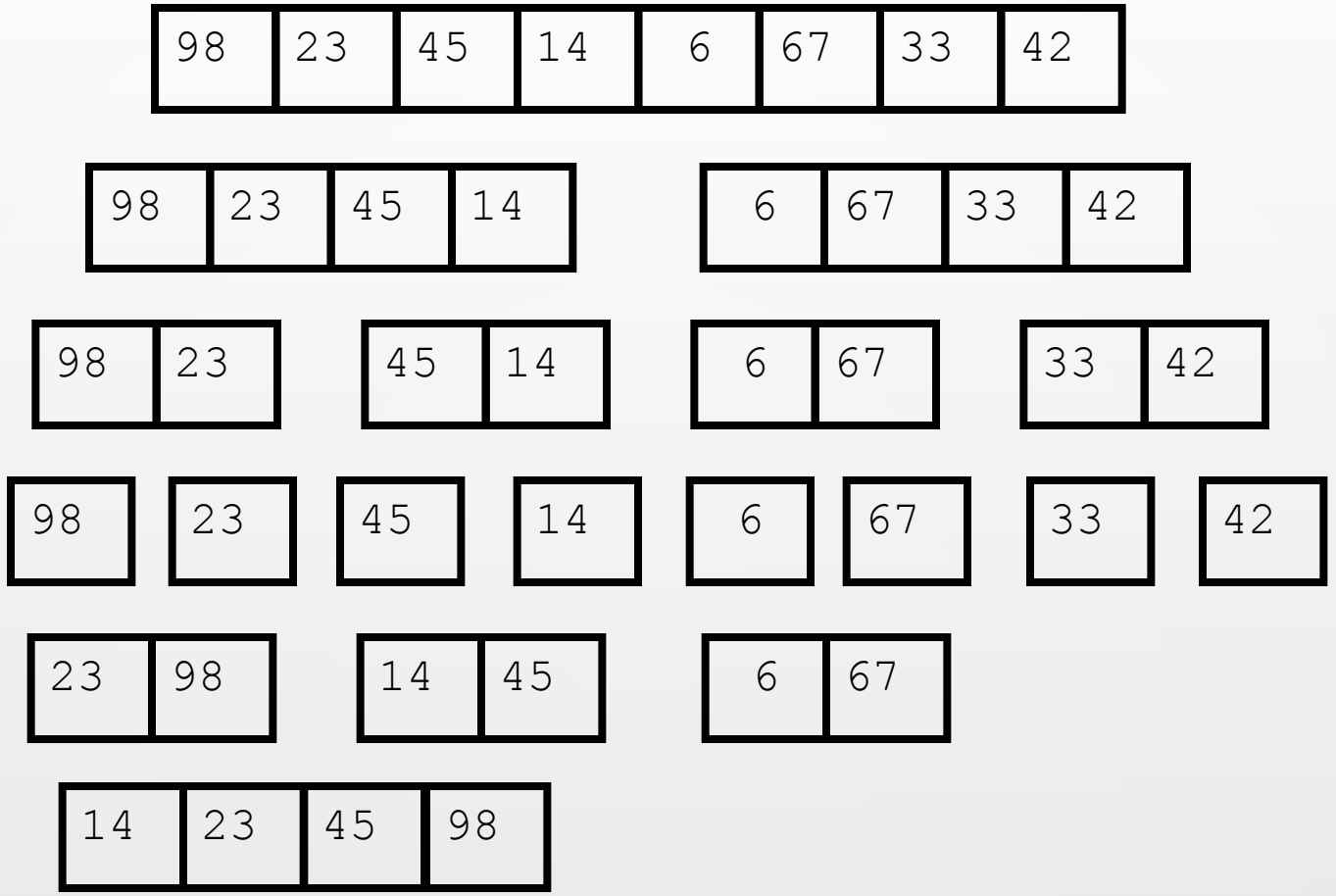

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

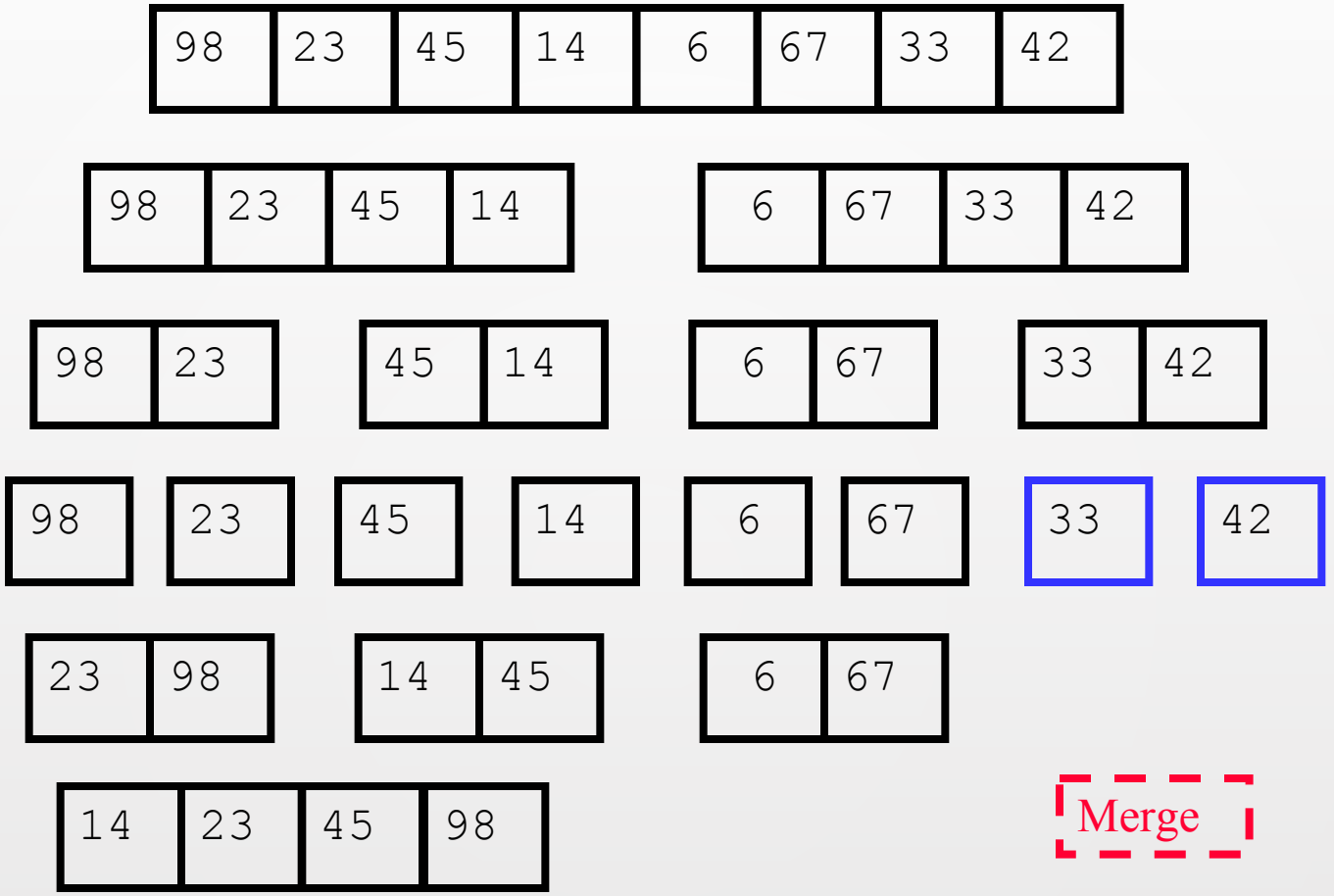

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

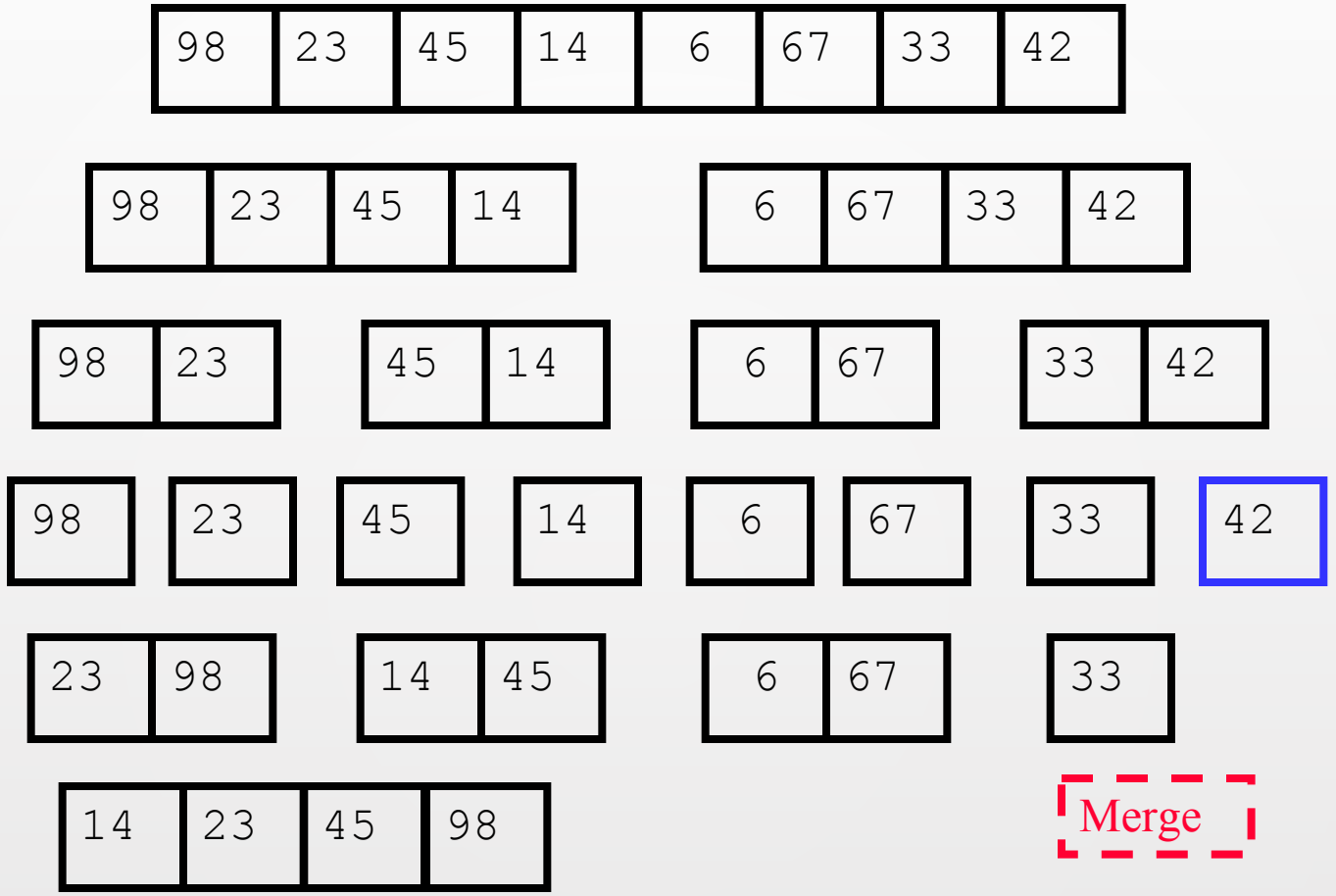

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

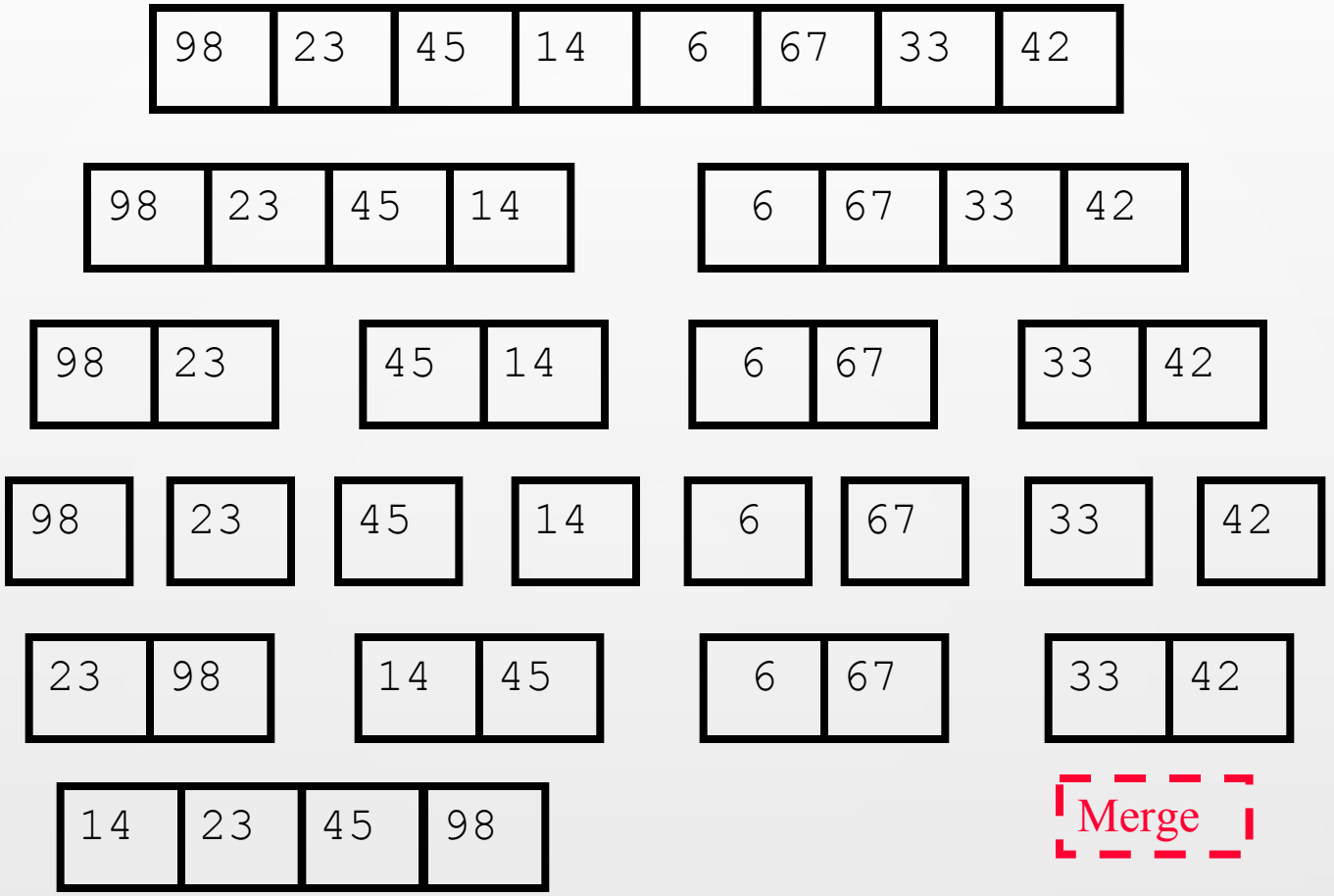

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge


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

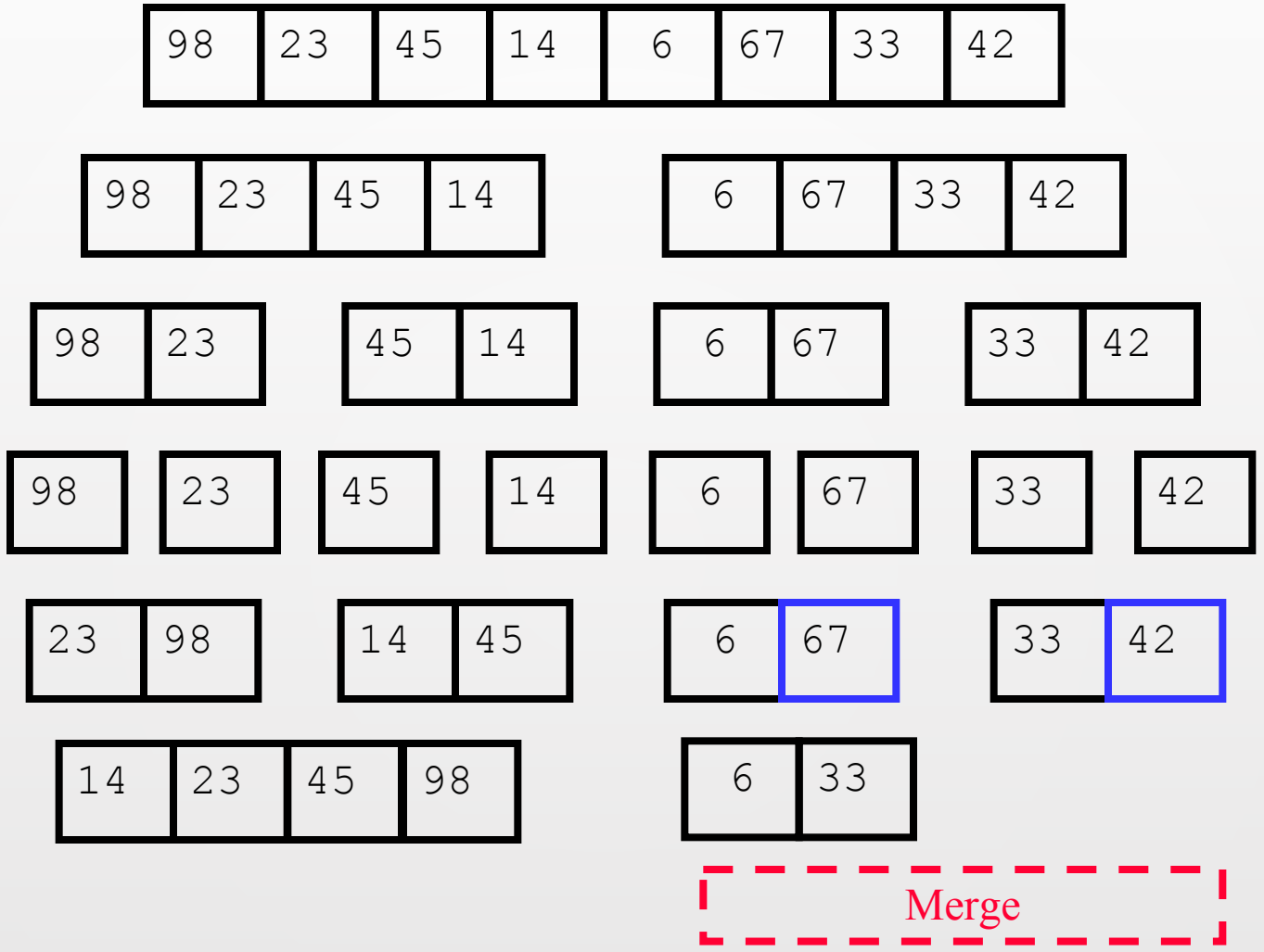

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98
Merge

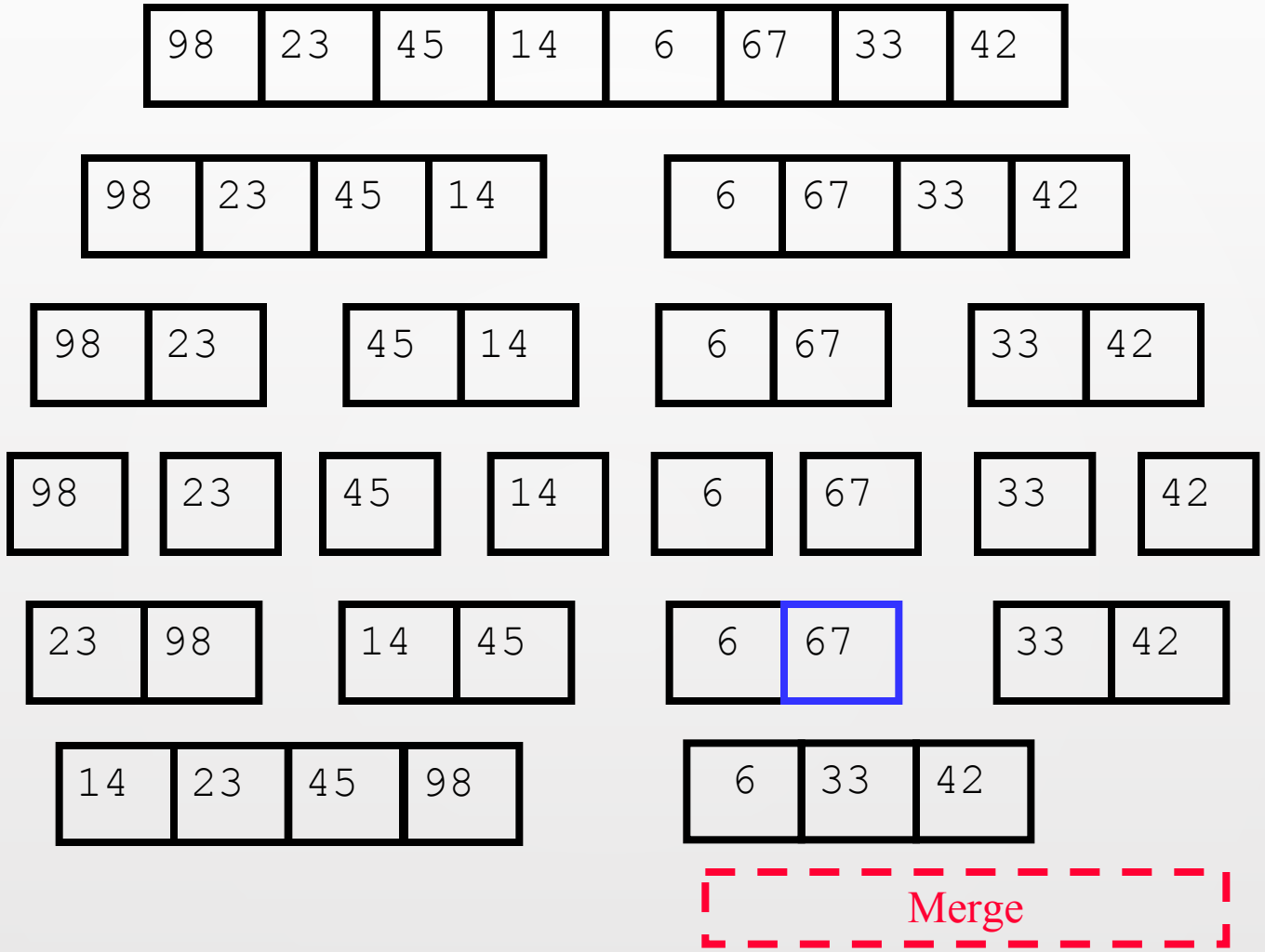

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge


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

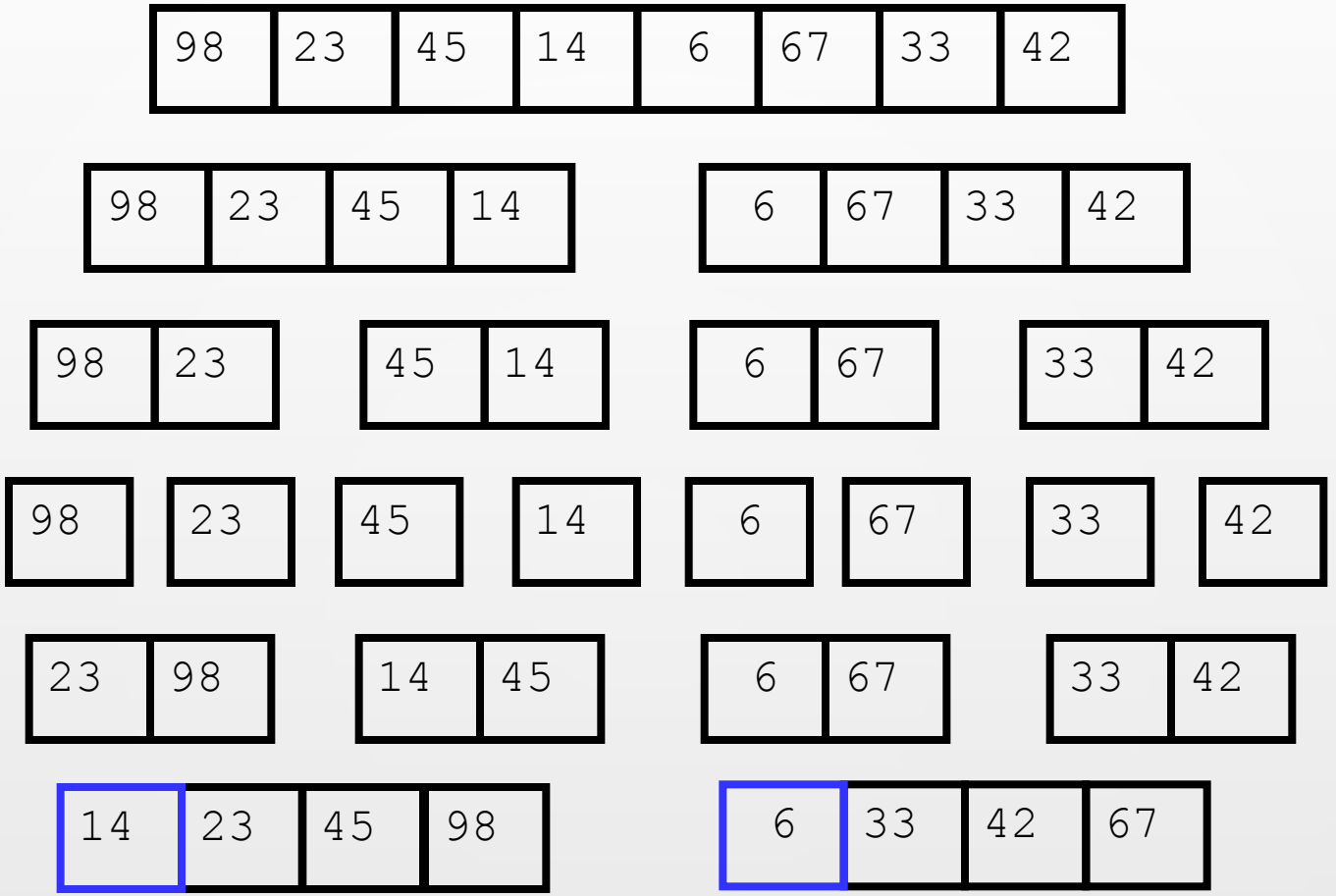

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge


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

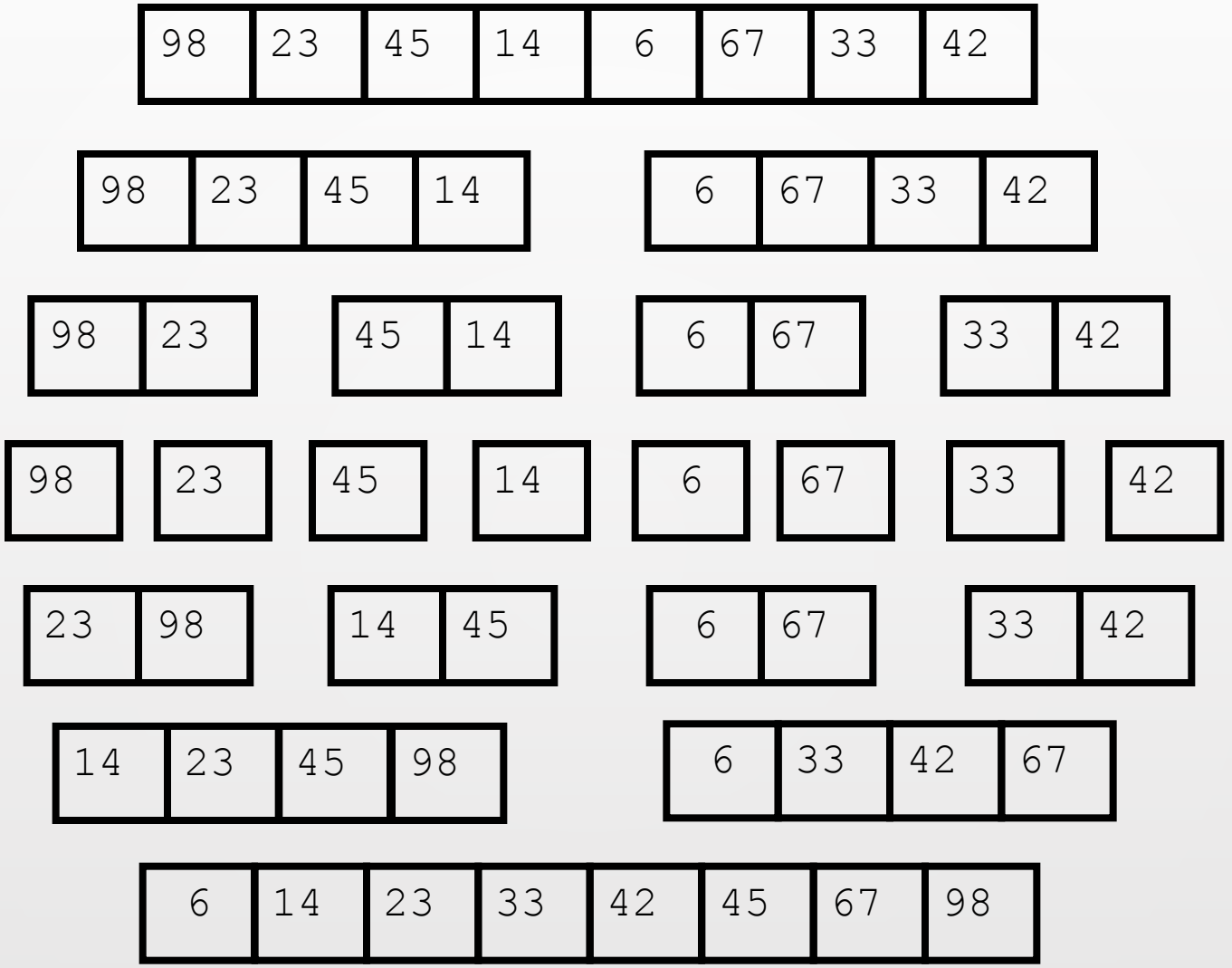

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23
Merge

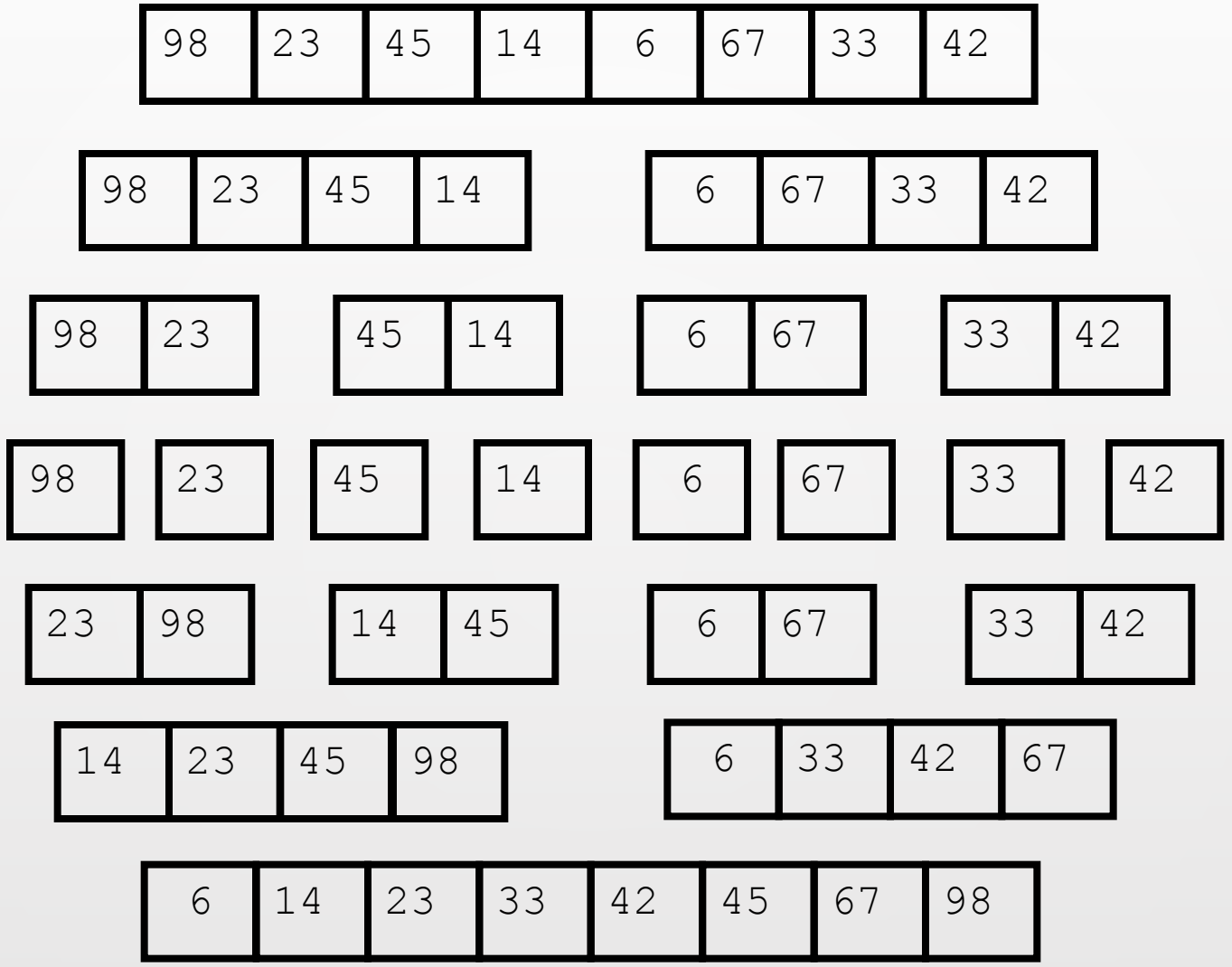

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33
Merge

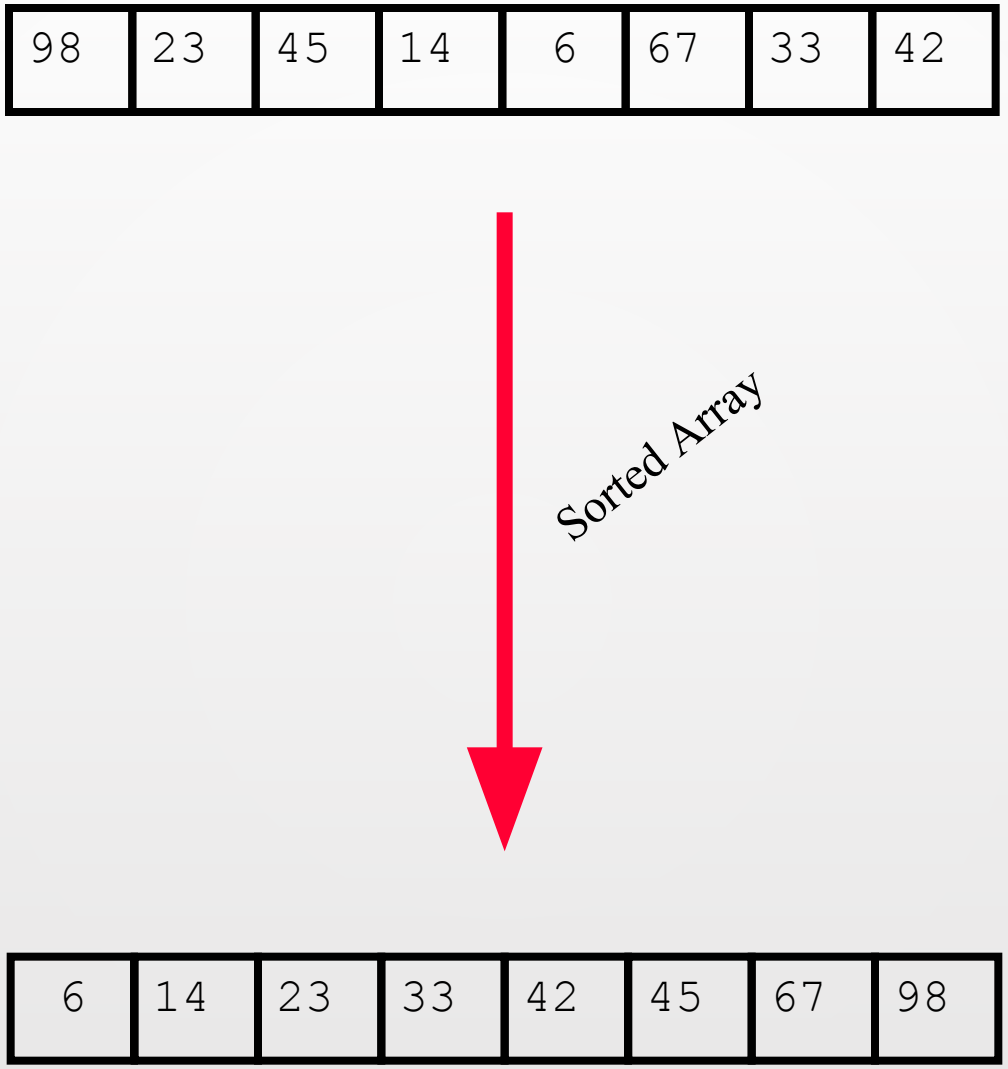

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42
Merge


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
Sorted Array
6 14 23 33 42 45 67 98

Merge Sort Algorithm

INPUT: a sequence of n numbers stored in array A

OUTPUT: an sorted sequence of n numbers

l points first element index of array and r points the index of last element

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
if p< r                                          ❑ Check for base case
   then q  ⌊(p+r)/2⌋                             ❑ Divide
      MergeSort (A, p, q)                        ❑ Conquer
      MergeSort (A, q+1, r)                      ❑ Conquer
      Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
                                                 ❑ Combine
```

Initial Call: MergeSort(A, 1, n)

Procedure Merge

Merge(A, p, q, r)

```
n1 ← q – p + 1
n2 ← r – q
for i ← 1 to n1
    do L[i] ← A[p + i – 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1+1] ← ∞
R[n2+1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
             i ← i + 1
        else A[k] ← R[j]
             j ← j + 1
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

O(n)

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Source: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4. 1320 pp

- Merge-sort is out-place sorting technique. During merge procedure it takes O(n) extra space. Therefore merge sort is not in-place sorting algorithm.
- Merge sort is not better approach for smaller size array.
- For smaller size array use insertion sort only.
- Merge sort is a stable sorting algorithm. After sorting if repeated element order not changes is called stable sorting technique.

Time Analysis of Merge Sort

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
1    if p < r
2        then q ← ⌊(p+r)/2⌋          O(1)
3             MergeSort (A, p, q)    T(n/2)
4             MergeSort (A, q+1, r)  T(n/2)
5             Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]   O(n)
```

So, we can write the overall running time of MERGE-SORT function in the form of recurrence relation as:

$$T(n)=2T(n/2)+\Theta(n)+\Theta(1)$$

So Solving above equation: **Θ(nlgn)**

Quicksort Algorithm

Given an array of n elements (e.g., integers):

- If array contains only one element, return
- Else
  - pick one element to use as pivot.
  - Partition elements into two sub-arrays:
    - Left sub-array contains elements less than or equal to pivot
    - Right sub-array contains elements greater than pivot
  - Then repeatedly apply above two steps on the two sub-arrays until array become sorted
  - Return results

Partitioning Array

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

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

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

Partitioning loops through, swapping elements below/above pivot.

Partition – Choosing the pivot

- First, we have to select a pivot element among the elements of the given array.
- Which array item should be selected as pivot?
  - Somehow we have to select a pivot, and we hope that we will get a good partitioning.
  - If the items in the array arranged randomly, we choose a pivot randomly.
  - We can choose the first or last element as a pivot (it may not give a good partitioning).
  - We can use different techniques to select the pivot.

Quick Sort Algorithm

As the name implies, it is quick, and it is the algorithm generally preferred for sorting.

Basic Ideas

(Another divide-and-conquer algorithm)

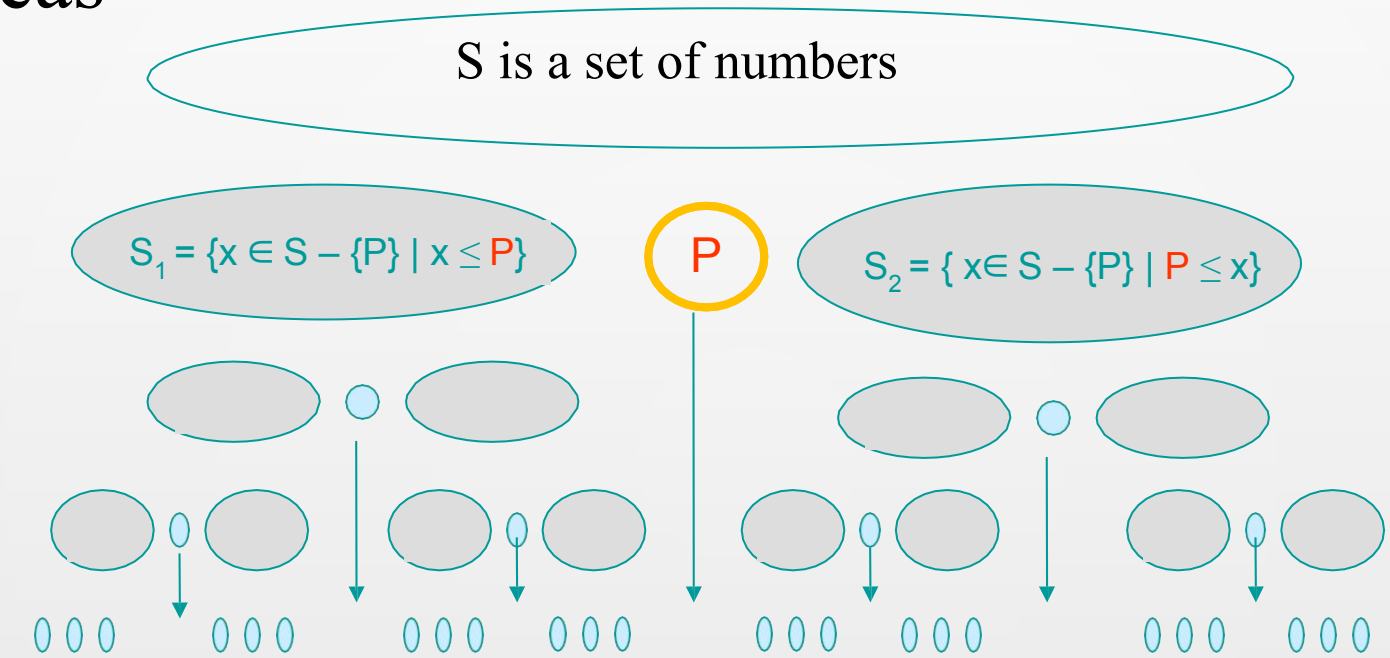
- Pick an element, say **P**(the pivot)
- Re-arrange the elements into 3 sub-blocks,

  1.those less than or equal to (≤) **P**(the **left-**block **S1**)

  2. **P**(the only element in the **middle**-block)

  3. those greater than or equal to (≥) **P**(the **right-**block **S2**)
- Repeat the process **recursively** for the **left-**and **right-**sub-blocks. Return {quicksort(**S1**), **P**, quicksort(**S2**)}.(That is the results of quicksort(**S1**), followed by **P**, followed by the results of quicksort(**S2**))

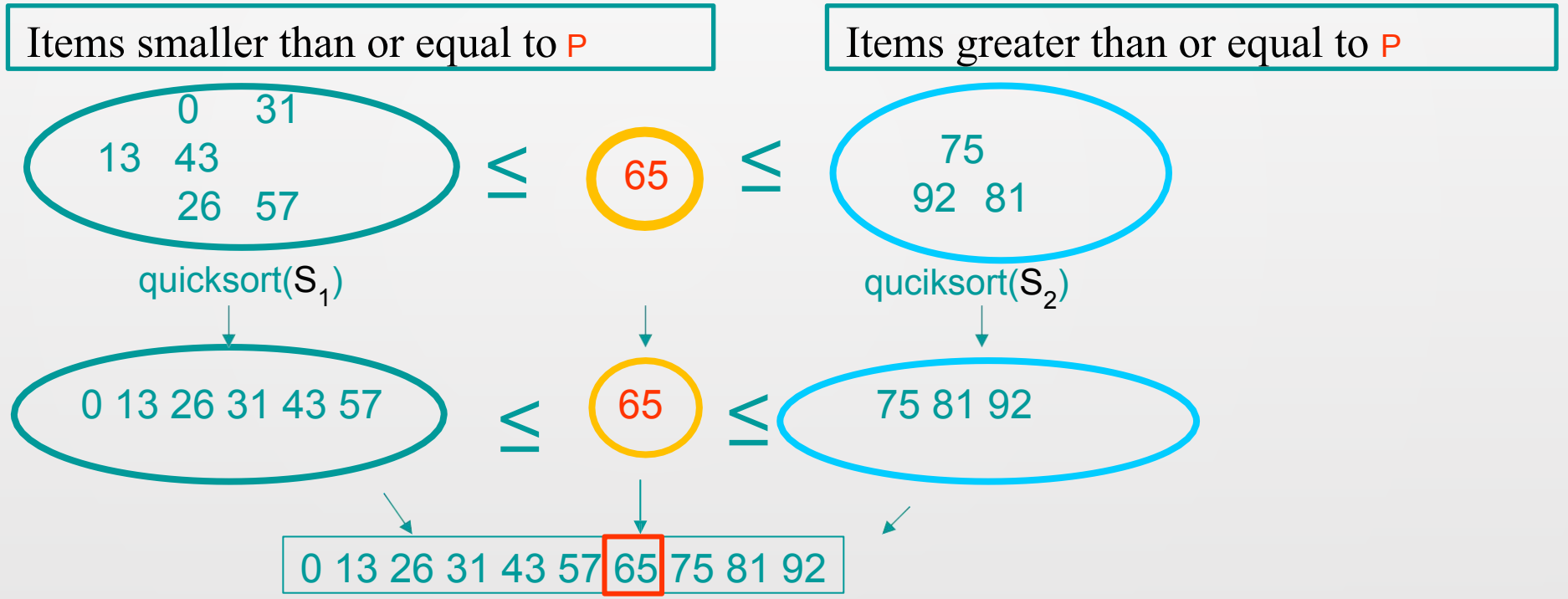
Basic Ideas



Source:https://home.cse.ust.hk/~dekai/271/notes/L01a/quickSort.pdf

Basic Ideas

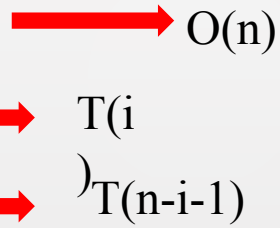
Pick a "Pivot" value, P Create 2 new sets without P



Source: https://home.cse.ust.hk/~dekai/271/notes/L01a/quickSort.pdf

Quicksort Code:

```
void quicksort(int a[], int p, int r)
{
  if(p < r)
    { int q;
        q = partition(a, p, r);
quicksort(a, p, q-1);
quicksort(a, q+1, r);
 }
 }
int partition (int arr[], int low, int high)
  {
```



Source: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4. 1320 pp

```
int pivot = arr[high]; // selecting last element as pivot
          int i = (low - 1); // index of smaller element
         for (int j = low; j <= high- 1; j++)
 {
  // If the current element is smaller than or equal to pivot
        if (arr[j] <= pivot)
          { i++; // increment index of smaller element
           swap(&arr[i], &arr[j]);
        }
  }
 swap(&arr[i + 1], &arr[high]);
   Return (i + 1);
}
```

Source: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4. 1320 pp

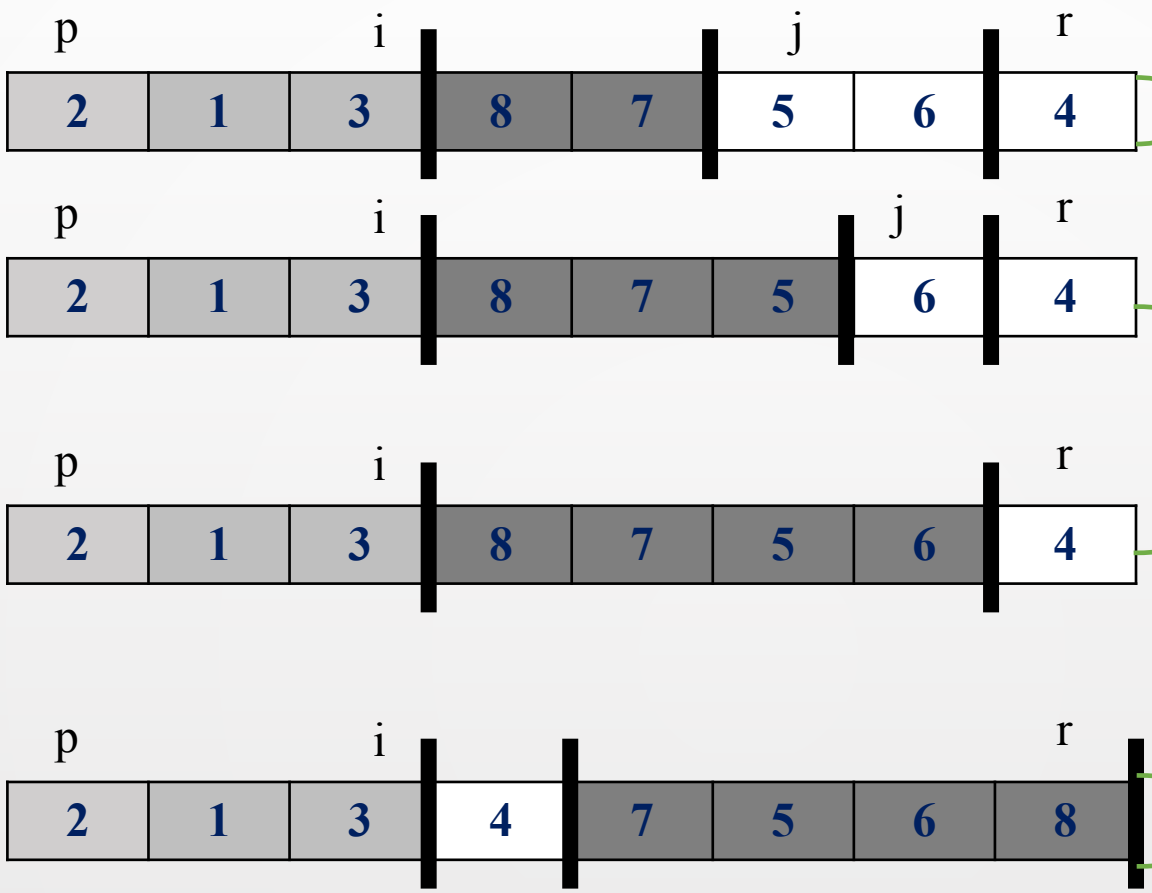
Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x

The initial array and variable settings. None of the elements have been placed in either of the first two partitions.

The value 2 is "swapped with itself" and put in the partition of smaller values

The values 8 and 7 are added to the partition of larger values

The values 1 and 8 are swapped, and the smaller partition grows.



The values 3 and 7 are swapped, and the smaller partition grows.

The larger partition grows to include 5 and 6, and the loop terminates.

In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

Running time analysis

- The advantage of this quicksort is that we can sort "in-place", i.e., *without* the need for a temporary buffer depending on the size of the inputs.
- Partitioning Step: Time Complexity is **O(n).**

Recall that quicksort involves *partitioning, and 2 recursive calls.*

Thus, giving the basic quicksort relation:

$$T(n) = O(n) + T(i) + T(n-i-1) = cn + T(i) + T(n-i-1)$$

where ***i*** is the size of the first sub-block after partitioning.

We shall take $T(0) = T(1) = 1$ as the initial conditions.

To find the solution for this relation, we'll consider three cases:

1. The Worst-case
2. The Best-case
3. The Average-case

All above cases depends on the value of the pivot!!

Running time analysis

Worst-Case (Data is sorted already)

- When the pivot is the smallest (or largest) element at partitioning on a block of size ***n***, the result
  - yields one empty sub-block, one element (pivot) in the "correct" place and one sub-block of size ***(n-1)***
  - takes O(**n**) times.
- Recurrence Equation:

  $T(1) = 1$

  $T(n) = T(n-1) + cn$

Solution: $\mathbf{O(n^2)}$

Worse than Mergesort!!!

Running time analysis

Best case:

- The pivot is in the middle (median) (at each partition step), i.e. after each partitioning, on a block of size ***n,*** the result
  - yields two sub-blocks of approximately equal size and the pivot element in the "middle" position
  - takes ***n*** data comparisons.
- Recurrence Equation becomes

  $T(1) = 1$

  $T(n) = 2T(n/2) + cn$

Solution: **O*(n logn)***

Comparable to Mergesort!!

Running time analysis

Average case:

The average case input would be any randomly generated arrays and the pivot is at the random position every step.

It turns out the average case running time also is ***O(n logn).***

More in detail will discuss offline.

So the trick is to select a good pivot

Different ways to select a good pivot.

- First element
- Last element
- Median-of-three elements
  - Pick three elements, and find the median *x of these elements*. Use that median as the pivot.
- Random element
  - Randomly pick a element as a pivot.

Selection problem : a ‘faster’ answer.

We can borrow the idea from the partition algorithm. Suppose we want to find a element of rank i in A[1..n]. After the 1st partition call (use **a random element** as pivot):

1. If the return index ‘q’ = i, then A[q] is the element we want. (Since there is exactly i-1 elements smaller than or equal to A[q]).
2. If the return index ‘q’ > i, then the target element can NOT be in A[q .. right].The target element is rank i in A[1.. q-1] Recursive call with ~~para~~meters (A, 1, q-1, i).
3. If the return index ‘q’ < i, then the target element can NOT be in A[1 .. q]. The target element is rank i-q in A[q+1 ..n] Recursive call with parameters (A, q+1,n, i-q).

A ‘faster’ selection algorithm : Codes

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p = r
2     then return A[p]
3  q ← RANDOMIZED-PARTITION(A, p, r)
4  k ← q − p + 1
5  if i = k              ▷ the pivot value is the answer
6     then return A[q]
7  elseif i < k
8     then return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

return a element of rank i in A[p..r]

$A[p..q-1] \leq A[q] \leq A[q+1..r]$

size of A[p..q]= k

Source: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4. 1320 pp

Different sorting algorithms

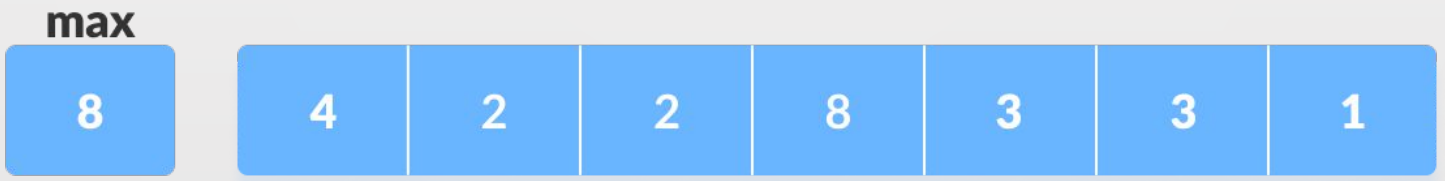
Sorting Algorithm	Worst-case time	Average-case time	Space overhead
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(1)$

Counting Sort

- Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array.
- The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

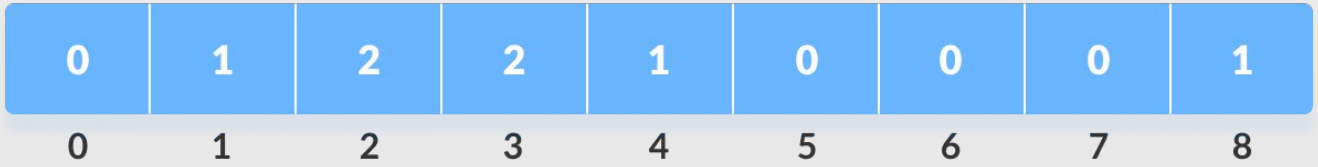
How Counting Sort Works?

Find out the maximum element (let it be max) from the given array:



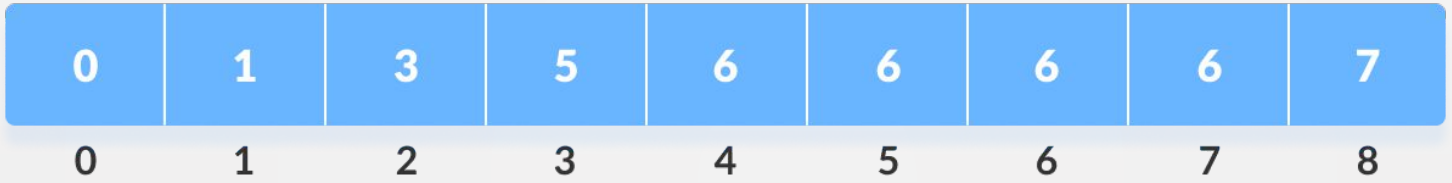
2. Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.

3. Store the count of each element at their respective index in count array. For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of count array. If element "5" is not present in the array, then 0 is stored in 5th position.



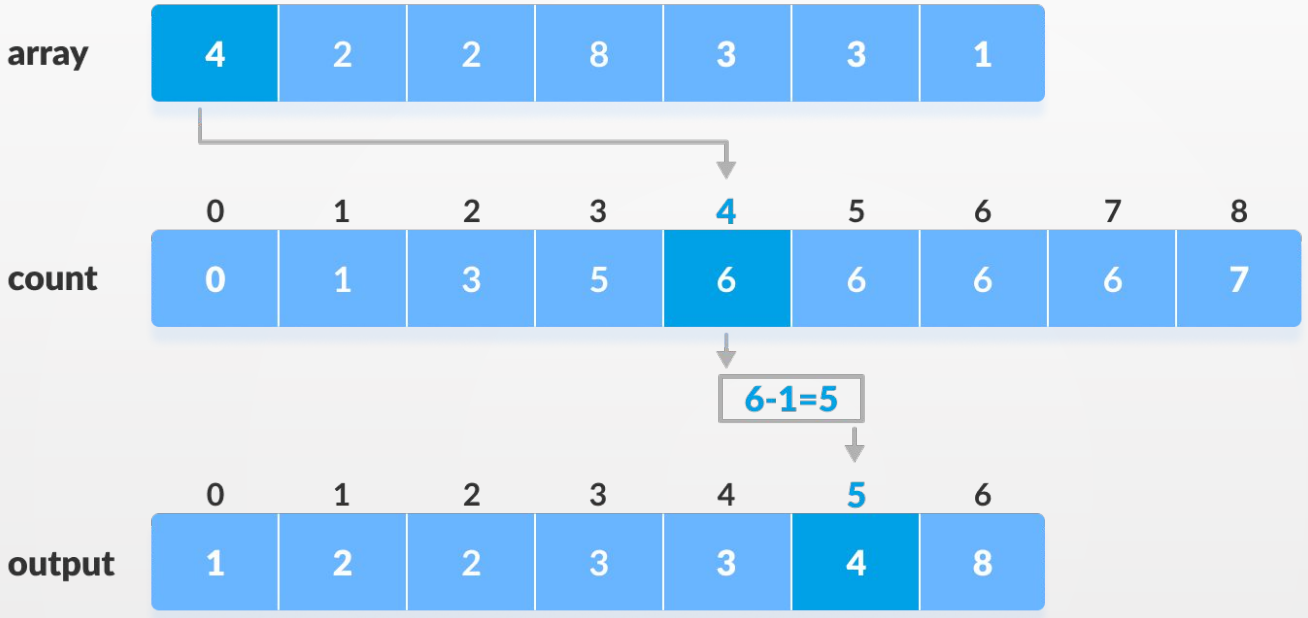
https://www.programiz.com/dsa/counting-sort

4. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.



5. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below

https://www.programiz.com/dsa/counting-sort



6. After placing each element at its correct position, decrease its count by one.

https://www.programiz.com/dsa/counting-sort

Counting Sort Algorithm

```
countingSort(array, size)
  max <- find largest element in array
   initialize count array with all zeros
   for j <- 0 to size
       find the total count of each unique element and
       store the count at jth index in count array
   for i <- 1 to max
      find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array decrease count of each element restored by 1
```

Radix Sort

- This sort is unusual because it does not directly compare any of the elements
- We instead create a set of buckets and repeatedly separate the elements into the buckets
- On each pass, we look at a different part of the elements.
- Assuming decimal elements and 10 buckets, we would put the elements into the bucket associated with its units digit
- The buckets are actually queues so the elements are added at the end of the bucket
- At the end of the pass, the buckets are combined in increasing

Radix Sort

- On the second pass, we separate the elements based on the “tens” digit, and on the third pass we separate them based on the “hundreds” digit
- Each pass must make sure to process the elements in order and to put the buckets back together in the correct order.
- Radix sort use counting sort in the intermediate steps.

Working of Radix Sort:

1. Find the largest element in the array, i.e. max. Let X be the number of digits n max. X is calculated because we have to go through all the significant places of all elements.

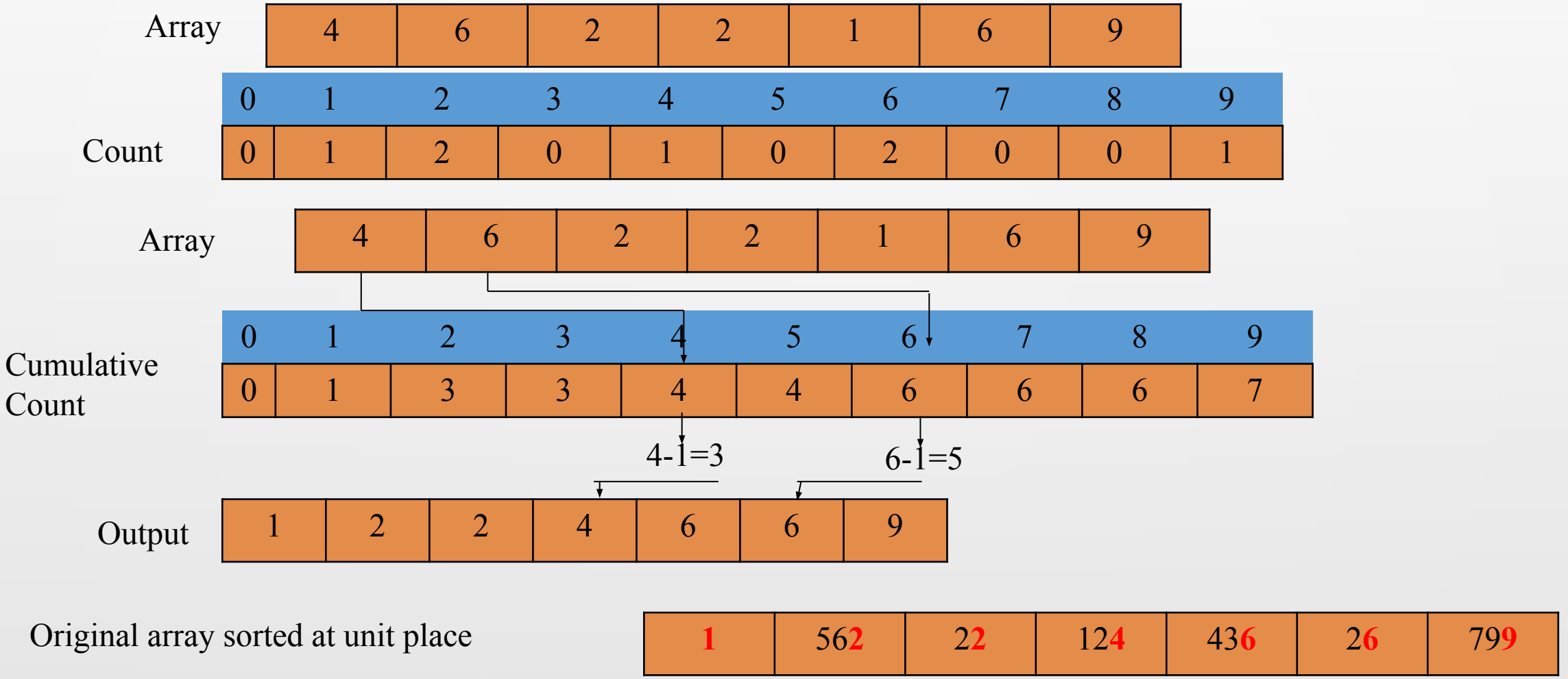
In the below array, we have the largest number 799. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).

2. Now, go through each significant place one by one. Use any stable sorting technique(Counting Sort) to sort the digits at each significant place. We have used counting sort for this

.

124	436	562	22	1	26	799

3. Sort the elements based on the unit place digits (X=0).



Original array sorted at unit place

3. Now, sort the elements based on digits at tens place.

1	22	124	26	436	562	799

Sort elements based on tens place

4. Finally, sort the elements based on the digits at hundreds place

001	022	026	124	436	562	799

Sort elements based on hundreds place

```
void radixsort(int array[], int size)
   { // Get maximum element
        int max = getMax(array, size);
  // Apply counting sort to sort elements based on place value.
    for (int place = 1; max / place > 0; place *= 10)
          countingSort(array, size, place);
 }
void countingSort(int array[], int size, int place)
 {
 const int max = 10;
  int output[size]; int count[max];
 for (int i = 0; i < max; ++i)
```

```
count[i] = 0; // Calculate count of elements
  for (int i = 0; i < size; i++)
  count[(array[i] / place) % 10]++;
for (int i = 1; i < max; i++)
 count[i] += count[i - 1];
for (int i = size - 1; i >= 0; i--)
 { output[count[(array[i] / place) % 10] - 1] = array[i];
 count[(array[i] / place) % 10]--;
 }
 for (int i = 0; i < size; i++)
 array[i] = output[i]; }
```

Radix Sort Analysis

- Each element is examined once for each of the digits it contains, so if the elements have at most M digits and there are N elements this algorithm has order O(M*N)
- This means that sorting is linear based on the number of elements
- Why then isn't this the only sorting algorithm used?

Radix Sort Analysis

- Though this is a very time efficient algorithm it is not space efficient
- If an array is used for the buckets and we have B buckets, we would need N*B extra memory locations because it's possible for all of the elements to wind up in one bucket
- If linked lists are used for the buckets you have the overhead of pointers

Bucket Sort

- Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.
  Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?
- A simple way is to apply a comparison based sorting algorithm. The lower bound for comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nlogn.
- Can we sort the array in linear time? Counting sort can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.

The idea is to use bucket sort. Following is bucket algorithm.

Bucket_Sort(arr[], n)

1)Create n empty buckets (Or lists).

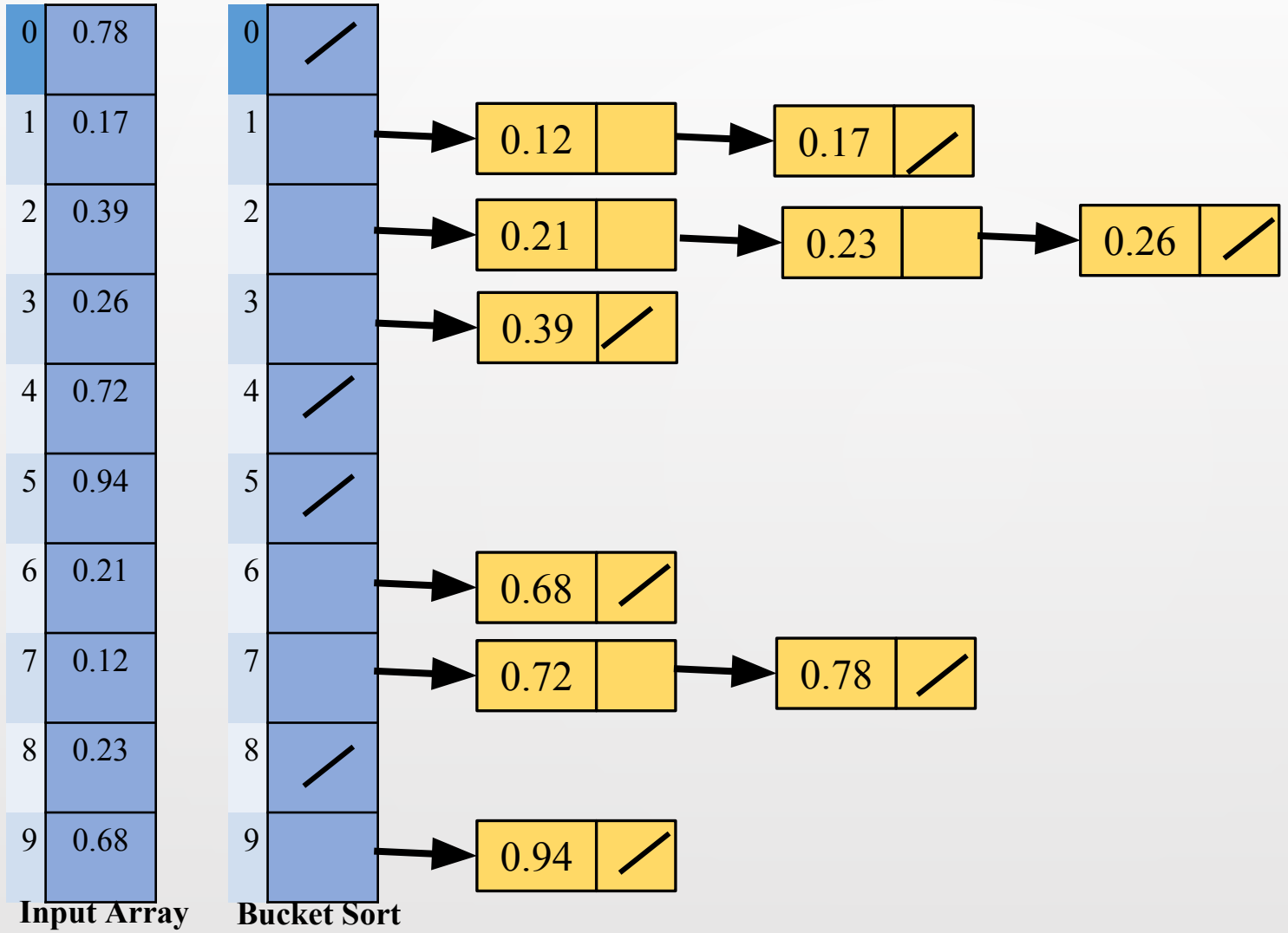
2) Do following for every array element arr[i].

a) Insert arr[i] into bucket[n*array[i]]

3) Sort individual buckets using insertion sort.

4) Concatenate all sorted buckets.

Bucket Sort Example



Complexity of Bucket Sort

Worst Case Complexity: $O(n^2)$

When there are elements of close range in the array, they are likely to be placed in the same bucket. This may result in some buckets having more number of elements than others.

Best Case Complexity: $O(n+k)$
It occurs when the elements are uniformly distributed in the buckets with a nearly equal number of elements in each bucket.

Average Case Complexity: $O(n)$
It occurs when the elements are distributed randomly in the array. Even if the elements are not distributed uniformly, bucket sort runs in linear time

References:

- *Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4. 1320 pp.*
- https://home.cse.ust.hk/~dekai/271/notes/L01a/quickSort.pdf
- https://www.programiz.com/dsa/counting-sort