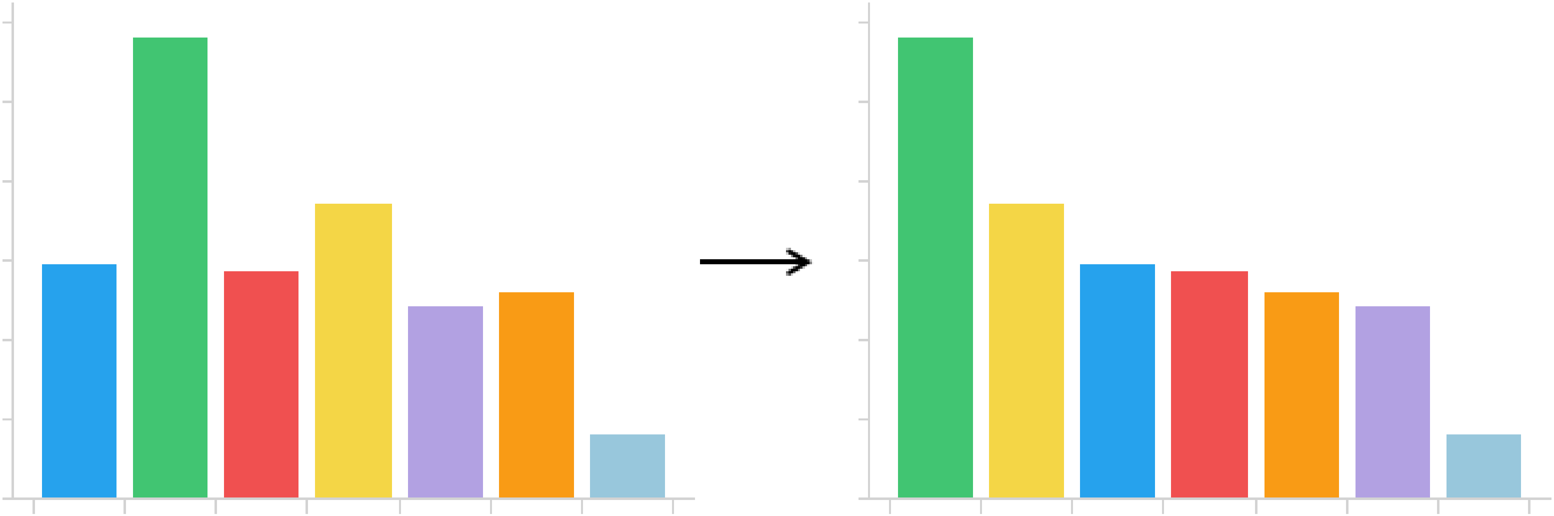
Sorting Algorithms

What is the Sorting Algorithm?

- Definition
  - Given a set of records, the output is to the non-decreasing order (**numerical order** or **lexicographical order**) of records.
    - The output is a permutation of the input.
    - https://www.toptal.com/developers/sorting-algorithms



Why is Sorting Important?

- Sorting has been commonly used as the pre-processed method for searching and matching.
  - Sorting is also used as the basic solution for many other complex problem.
  - In most organization, more than 25% of computing time is spent on sorting.

- No best algorithm for any situation: initial ordering and size of list.
  - We need to know several techniques.
  - The analysis of lower bound is good for understanding basic skill for algorithm analysis.

Categories of Sorting Algorithms

- Comparison sort vs. non-comparison sort
  - Comparative sorting algorithm determines **the order of two element** through a **comparison operator**.
  - Comparison sorting algorithms:
    - Selection sort, Bubble sort, Insertion sort, Quick sort, …
  - Non-comparison sorting algorithms:
    - Radix sort, Bucket sort, Counting sort

- Note: non-comparison sort is also called a linear sorting method.

Categories of Sorting Algorithms

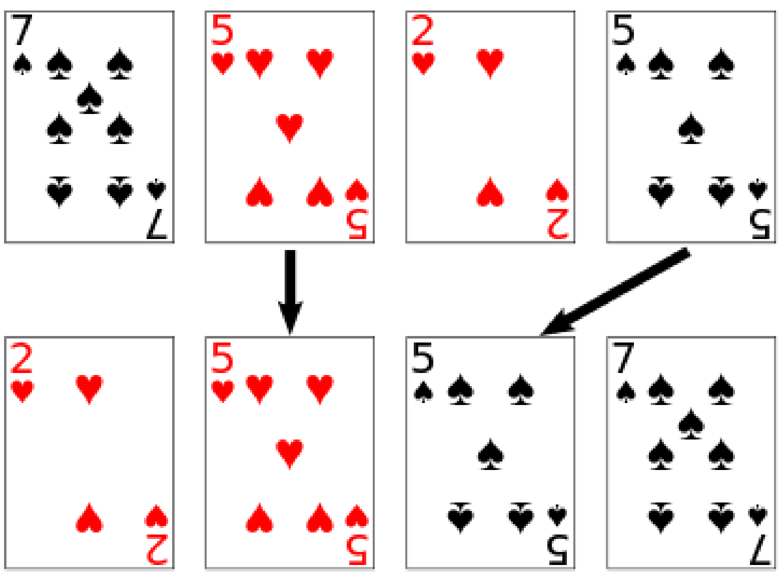
- Internal sort vs. external sort
  - Internal sorting technique is for the case where the list is small enough to sort entirely in main memory
    - Minimizing **the number of comparisons**
  - External sorting technique is used when the list is too big to fit into main memory, (e.g., disk and SSD).
    - Minimizing **the number of I/O accesses**



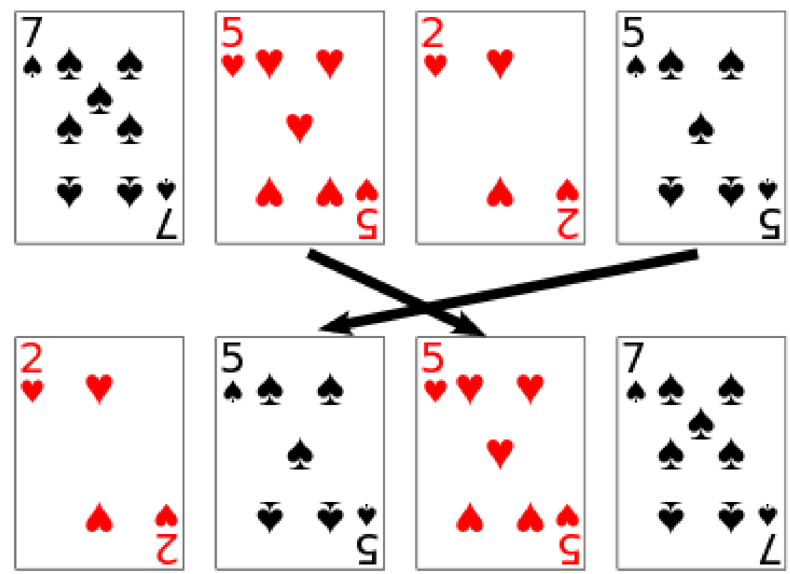
Stability of Sorting Algorithms

- Definition
  - Stable sorting algorithms maintain **the relative order of records with equal keys**.
    - The initial order of records with equal keys **does not changed**.

Stable algorithm



unstable algorithm



What is Selection Sort?

- Description: At the i-th iteration ($\mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{n} - \mathbf{1}$)
  - Given a list L, there are two parts: $\boldsymbol{L}[\mathbf{0}, \boldsymbol{i} - \mathbf{1}]$ *and* $\boldsymbol{L}[\boldsymbol{i}, \boldsymbol{n} - \mathbf{1}]$.
    - $\boldsymbol{L}[\mathbf{0}, \boldsymbol{i} - \mathbf{1}]$: a sublist of items to be **already sorted (blue)**
    - $\boldsymbol{L}[\boldsymbol{i}, \boldsymbol{n} - \mathbf{1}]$: a sublist of items **remaining to be sorted (yellow)**
  - Select the minimum (red) from the unsorted part.
  - Exchange the minimum with the i-th element in the list.

0th	10	21	5	8	1	12	➡	10	21	5	8	1	12
1st	1	21	5	8	10	12	➡	1	21	5	8	10	12
2nd	1	5	21	8	10	12	➡	1	5	21	8	10	12
3rd	1	5	8	21	10	12	➡	1	5	8	21	10	12
4th	1	5	8	10	21	12	➡	1	5	8	10	21	12

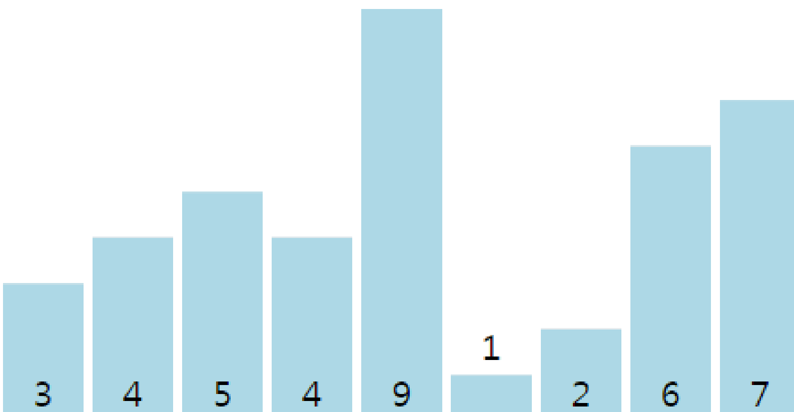
Implementation of Selection Sort

- Implementation

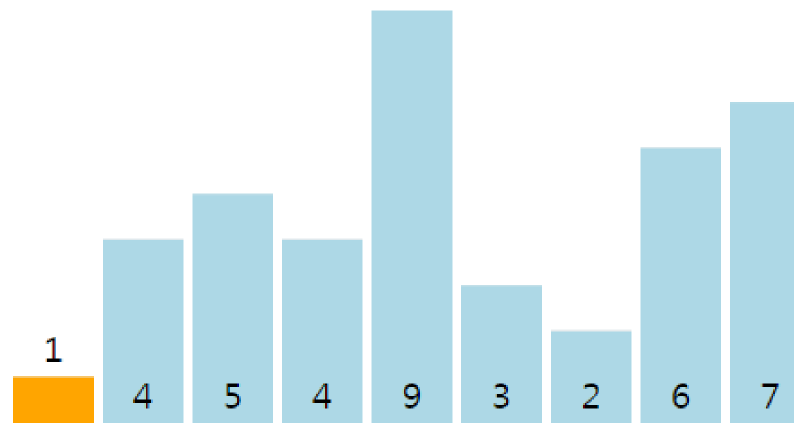
```
void SelectionSort(Data* list, int n)
{
    int min, temp;
    for (int i = 0; i < n - 1; i++)
    {
        min = i;
        for (int j = i + 1; j < n; j++)
        {
            // Find an index with the minimum element.
            if (list[j] < list[min])
                min = j;
        }
        // Exchange the minimum element and the i-th element.
        SWAP(list[i], list[min], temp); /* macro */
    }
}
```

Exercise: Selection Sort

- Animation: sorting 3, 4, 5, 4, 9, 1, 2, 6, 7
  - Draw the step-by-step procedure of selection sort.
  - https://visualgo.net/en/sorting



Initial list



After sorting the first element

- Is it stable?

Analysis of Selection Sort

- Time complexity
  - Best case: $O(n^2)$
    - The number of comparisons: $(n-1)+(n-2)+\ ...\ +2+1$
  - Worst case: $O(n^2)$

- Q) Is it stable?

- A) No, the movement of elements are not adjacent.
  - The selection sort is unstable.
  - E.g., $4_{(1)}$ 2 $4_{(2)}$ 1 5
    - exchange $4_{(1)\ \&}$ 1
    - done

What is Bubble Sort?

- Description: At the $\boldsymbol{i}$ th iteration ($\mathbf{0} \le \boldsymbol{i} \le \boldsymbol{n} - \mathbf{1}$)
  - There are two parts: $\boldsymbol{L}[\mathbf{0}, \boldsymbol{n} - \boldsymbol{i} - \mathbf{1}]$ *and* $\boldsymbol{L}[\boldsymbol{n} - \boldsymbol{i} - \mathbf{1},\ \boldsymbol{n} - \mathbf{1}]$.
    - $\boldsymbol{L}[\mathbf{0}, \boldsymbol{n} - \boldsymbol{i} - \mathbf{1}]$: a sublist of items to be **already sorted** (blue)
    - $\boldsymbol{L}[\boldsymbol{n} - \boldsymbol{i} - \mathbf{1}, \boldsymbol{n} - \mathbf{1}]$: a sublist of items **to be sorted** (yellow)
  - Compare each pair of **adjacent items** (red) and **swap** them if they are in the **wrong** order from the unsorted part.

10	21	5	8	1	12

Check if swap 10 and 21.

10	21	5	8	1	12

Check if swap 21 and 5.

10	5	21	8	1	12

Check if swap 21 and 8.

10	5	8	21	1	12

Check if swap 21 and 1.

10	5	8	1	21	12

Check if swap 21 and 12.

10	5	8	1	12	21

The 0-th iteration is ended.

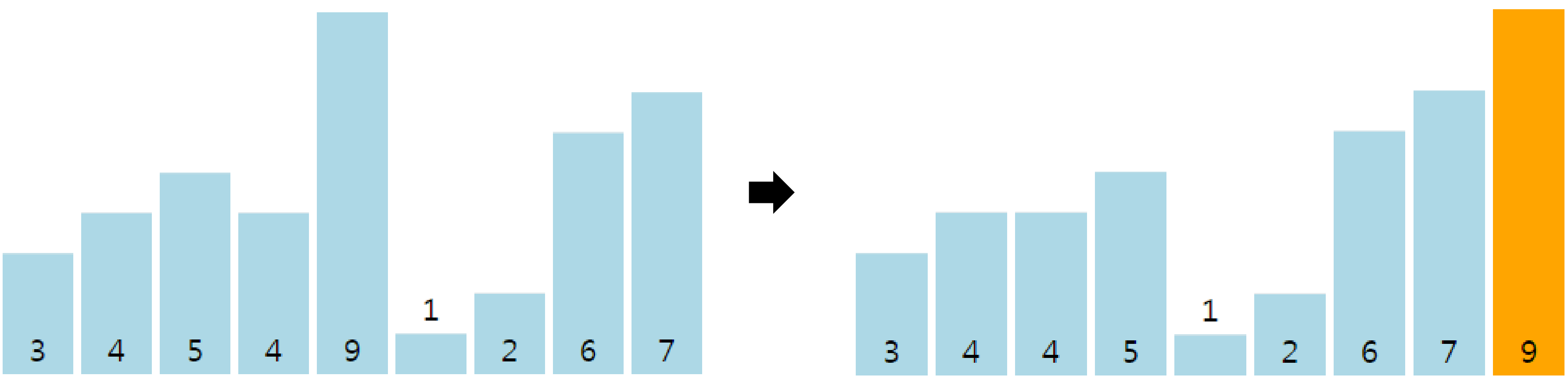
Implementation of Bubble Sort

- Implementation

```
void BubbleSort(Data* list, int n)
{
	int temp;
	for (int i = n - 1; i > 0; i--)
	{
		for (int j = 0; j < i; j++)
		{
			// Compare each pair of adjacent items.
			if (list[j] > list[j + 1])
			{
				// Swap if they are in the wrong order.
				SWAP(list[j], list[j + 1], temp);
			}
		}
	}
}
```

Exercise: Bubble Sort

- Animation: sorting 3, 4, 5, 4, 9, 1, 2, 6, 7
  - Draw the step-by-step procedure of bubble sort.
  - https://visualgo.net/en/sorting



Initial list

After sorting the last element

- Is it stable?

Analysis of Bubble Sort

- Time complexity
  - Best case: $O(n^2)$
  - Worst case: $O(n^2)$

- Q) Is it stable?

- A) Yes, it is based on the exchanges between two adjacent items.
  - Consider $4_{(1)}$ $4_{(2)}$ 1 2 5.

What is Insertion Sort?

- Description: At the $\boldsymbol{i}$ th iteration $(\mathbf{0} \le \boldsymbol{i} \le \boldsymbol{n} - \mathbf{1})$
  - Given a list L, there are two parts: $\boldsymbol{L}[\mathbf{0}, \boldsymbol{i} - \mathbf{1}]$ *and* $\boldsymbol{L}[\boldsymbol{i}, \boldsymbol{n} - \mathbf{1}]$.
    - $\boldsymbol{L}[\mathbf{0}, \boldsymbol{i} - \mathbf{1}]$: a sublist of items that is partially sorted (purple)
    - $\boldsymbol{L}[\boldsymbol{i}, \boldsymbol{n} - \mathbf{1}]$: a sublist of items to be sorted (yellow)
  - Insert $L[i]$ to the correct position in $L[0, i]$ to be sorted.

1th	10	21	5	8	1	12	→	10	21	5	8	1	12
2st	10	21	5	8	1	12	→	10	5	21	8	1	12
3nd	5	10	21	8	1	12	→	5	8	10	21	1	12
4rd	5	8	10	21	1	12	→	1	5	8	10	21	12
5th	1	5	8	10	21	12	→	1	5	8	10	12	21

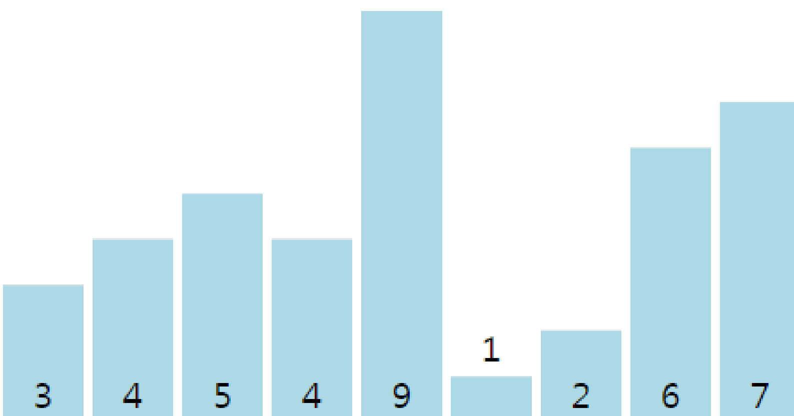
Implementation of Insertion Sort

- Implementation

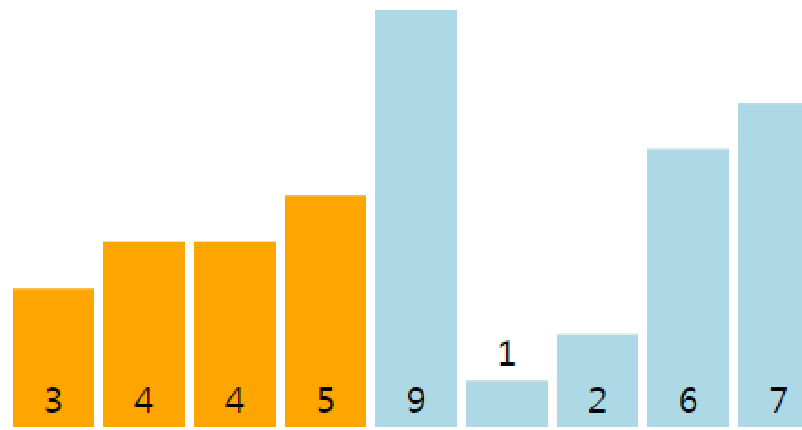
```
void InsertionSort(Data* list, int n)
{
    int j, key;
    for (int i = 1; i < n; i++)
    {
        key = list[i];// Choose the i-th element.
        for (j = i - 1; j >= 0; j--) {
            // If the j-th element is greater than key,
            // move to the next position.
            if (key < list[j])
                list[j + 1] = list[j];
            else
                break; /* key <=
        }
        // list[j] <= key and list[j+1] is empty
        // move the key to the (j+1)-th element.
        list[j + 1] = key;
    }
}
```

Exercise: Insertion Sort

- Animation: sorting 3, 4, 5, 4, 9, 1, 2, 6, 7
  - Draw the step-by-step procedure of insertion sort.
  - https://visualgo.net/en/sorting



Initial list



After sorting four elements

- Is it stable?

Analysis of Insertion Sort

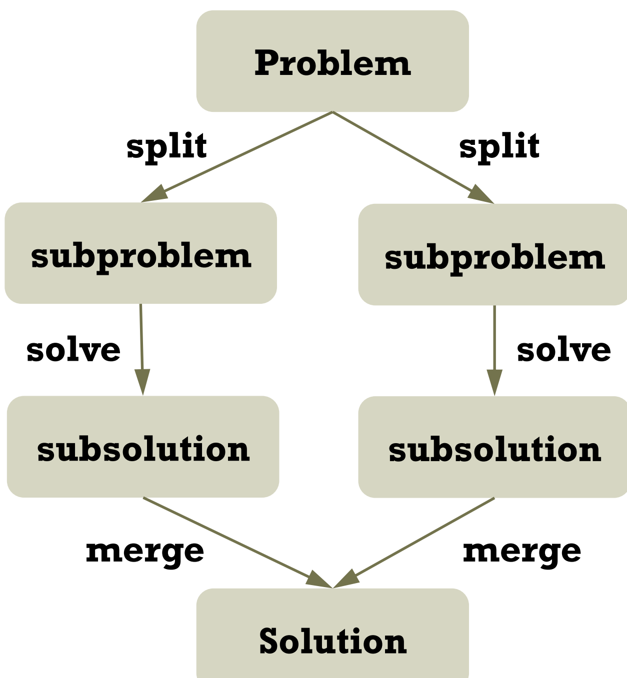
- Time complexity
  - Best case: $O(n)$
  - Worst case: $O(n^2)$

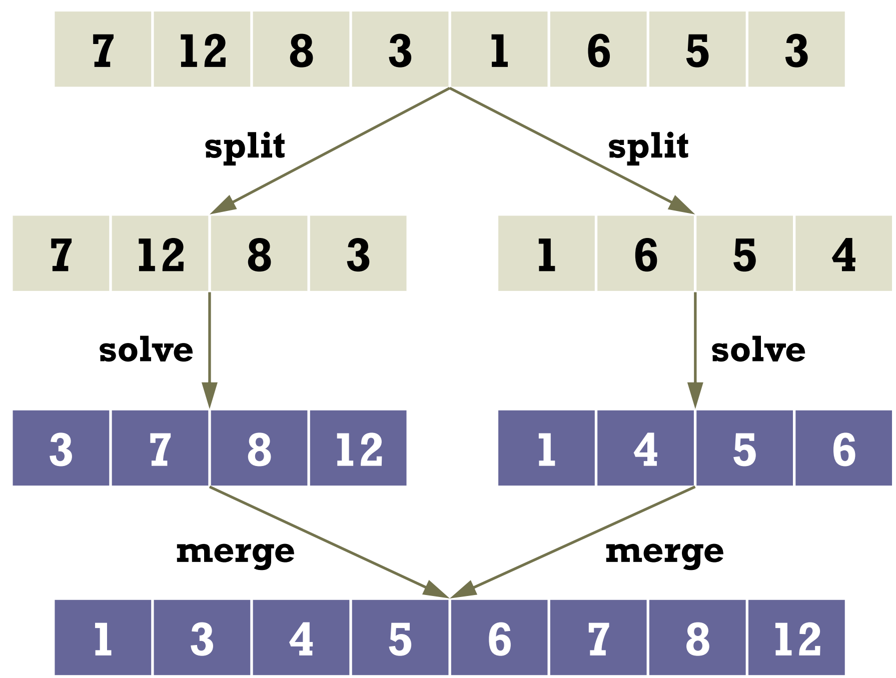
- Q) Is it stable?

- A) Yes, the exchange of elements is adjacent.
  - The exchanges of elements are similar to bubble sort.

Divide & Conquer (D&C) Paradigm

- Definition
  - **Breaking down a problem into two or more subproblems** of the same or related type.
  - **The solutions to the subproblems are combined to be a solution** to the original problem.





What is Quick Sort?

- Description
  - Invented by Tony Hoare in 1959
  - Based on the divide and conquer (D&C) paradigm.

- Overall procedure
  - **Pivot selection**: Pick an element, called a pivot, from the list.
  - **Partitioning**: reorder the list with the pivot.
    - The elements less than the pivot come before the pivot.
    - The element greater than the pivot come after the pivot.
  - Recursively apply the above steps to the sublists.

7	4	8	1	12	11	3	9

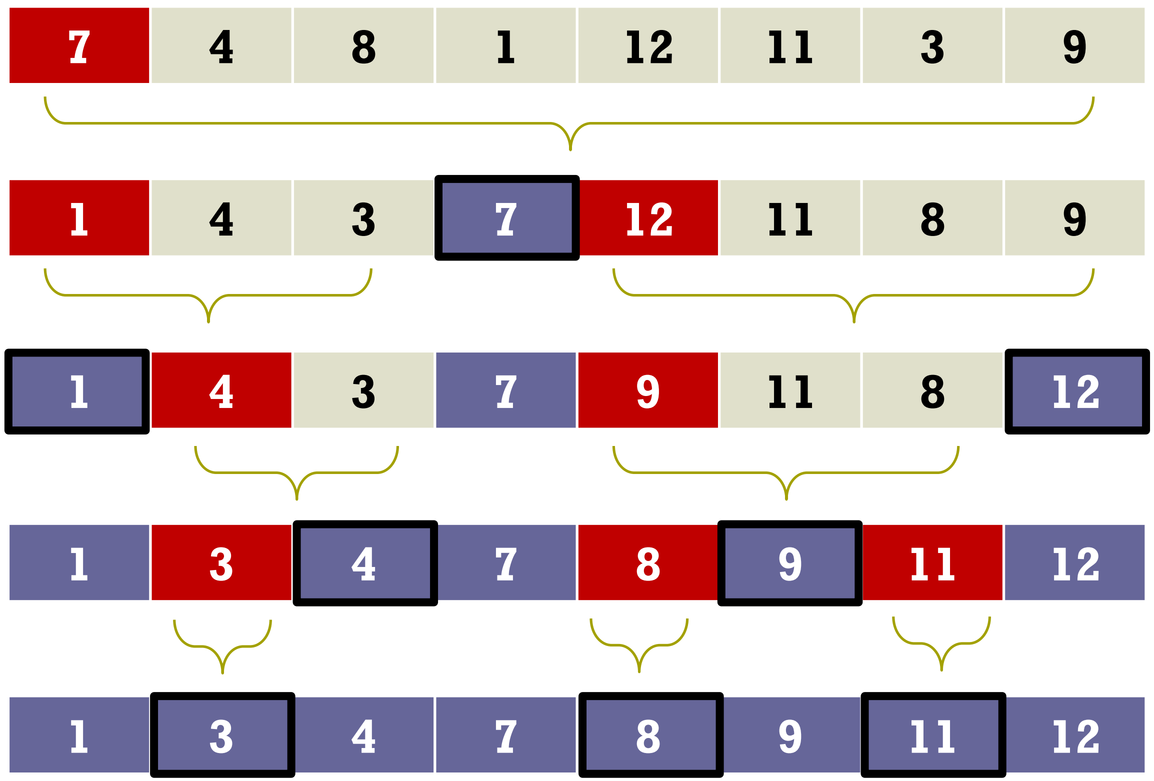
Pivot

Partitioning

1	4	3	7	12	11	8	9

Example of Quick Sort

- Overall procedure
  - For each list, select a pivot as the left-most element.
  - Partition the list into two sublists.



Quick Sort: Partitioning

- Select a pivot from the list.
  - In general, the left-most element is chosen as the pivot.

- Use two variables: *low* and *high*
  - ***low***: if *L*[*low*] is less than a pivot, move to the right element.
  - ***high***: if *L*[*high*] is greater than a pivot, move to the left element.
  - Swap two elements *L*[*low*] and *L*[*high*].

- If *low* and *high* are crossed, stop partitioning.
  - Swap two elements *L*[*left*] and *L*[*high*].

Quick Sort: Partitioning

- Assume that the left-most element is the pivot.
  - ***left***: an starting index for a sublist that is less than a pivot
  - ***right***: an ending index for a sublist that is greater than a pivot

Select a *pivot*	**7**	**4**	**8**	**1**	**12**	**11**	**3**	**9**
	left	***low***						***right*, *high***
Move *low* until *pivot* < *L*[*low*].	**7**	**4**	**8**	**1**	**12**	**11**	**3**	**9**
	left		***low***					***right*, *high***
Move *high* until *pivot* >= *L*[*high*].	**7**	**4**	**8**	**1**	**12**	**11**	**3**	**9**
	left		***low***				***high***	***right***
Swap *L*[*low*] and *L*[*high*].	**7**	**4**	**3**	**1**	**12**	**11**	**8**	**9**
	left		***low***				***high***	***right***

Quick Sort: Partitioning

- Assume that the left-most element is the pivot.
  - ***left***: an starting index for a sublist that is less than a pivot
  - ***right***: an ending index for a sublist that is greater than a pivot

Move *low* until *pivot* < *L*[*low*].

7	4	3	1	12	11	8	9
left				*low*		*high*	*right*

Move *high* until *pivot* >= *L*[*high*].

7	4	3	1	12	11	8	9
left			*high*	*low*			*right*

Swap *L*[*left*] and *L*[*high*].

1	4	3	7	12	11	8	9
left			*high*	*low*			*right*

All elements in the left sublist are less than the pivot.

All elements in the right sublist are greater than the pivot.

Quick Sort: Partitioning

- Partitioning one list into two sublists
  - All elements in the left sublist are less than the pivot.
  - All elements in the right sublist are greater than the pivot.

```
int Partition(Data* list, int left, int right)
{
     int pivot = list[left], temp;
     int low = left + 1, high = right;

     while (1) {
          while (low < right && list[low] < pivot)
               low++;    // Move low until pivot < L[low]
          while (high > left && list[high] >= pivot)
               high--;   // Move high until pivot >= L[low]

          if (low < high)
               // Swap list[low] and list[high].
               SWAP(list[low], list[high], temp);
          else break;
     }
     SWAP(list[left], list[high], temp);
     return high;// return the pivot position.
}
```

Implementation of Quick Sort

- Overall procedure
  - **Pivot selection**: Pick an element, called a pivot, from the list.
  - **Partitioning**: reorder the list with the pivot.
  - Recursively apply the above steps to the sublists.

```
void QuickSort(Data* list, int left, int right)
{
    if (left < right) {
        // The mid refers to the pivot position.
        int mid = Partition(list, left, right);

        // All elements are less than the pivot.
        QuickSort(list, left, mid - 1);

        // All elements are greater than the pivot.
        QuickSort(list, mid + 1, right);
    }
}
```

Analysis of Quick Sort

- We expect that the list will be split into two halves in an average case
  - $\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + cn}$, where splitting time is cn.
  - The time complexity of quick sort is $\boldsymbol{O(nlog\,n)}$.

- However, the worse case is that the list will be split into $\mathbf{1}$ and $\boldsymbol{n-1}$.
  - $\boldsymbol{T(n) = T(n-1) + cn} = T(n-2) + 2cn = \cdots$

    $= T(1) + cn(n-1) = O(n^2)$
  - The time complexity of quick sort is $\boldsymbol{O(n^2)}$.

Analysis of Quick Sort

- The worse case occurs if the pivot is selected as an extremely skewed case.
  - The time complexity of quick sort mainly depends on pivot selection.

- How to choose a good pivot in quick sort?
  - random
  - median of 1st, middle and last elements

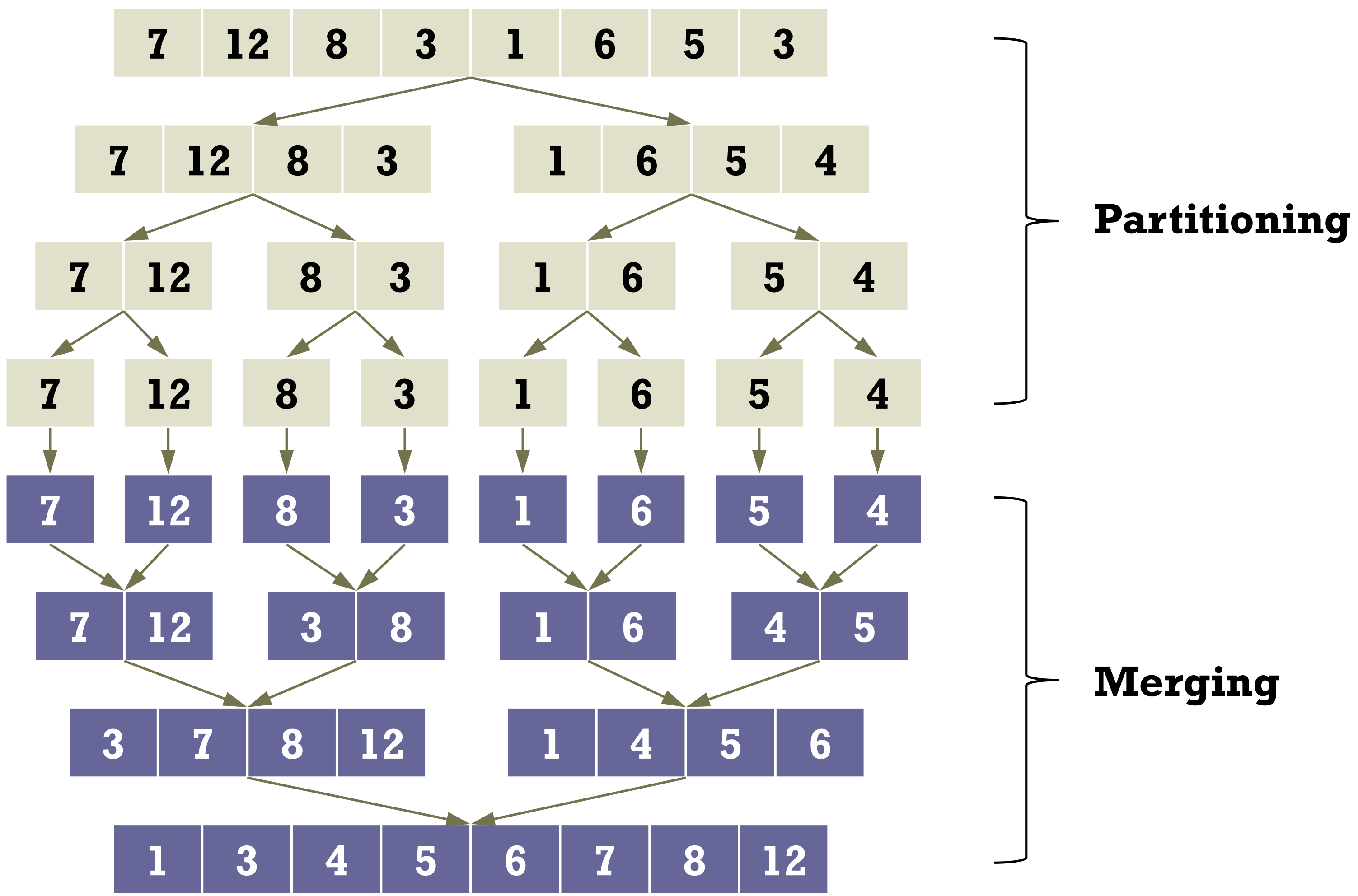
What is Merge Sort?

- Why does quick sort have $\boldsymbol{O}(\boldsymbol{n}^2)$ in the worse case?
  - When the sizes of partitioned sublists are extremely skewed.
  - **Let us split the list into exactly half and half**.

- Description: Use the D&C paradigm.
  - **Divide**: split the list into two halves.
  - **Conquer**: Sort two sublists.
  - **Combine**: Merge two sorted sublits into one list.
  - Recursively apply the above steps to the sublists.

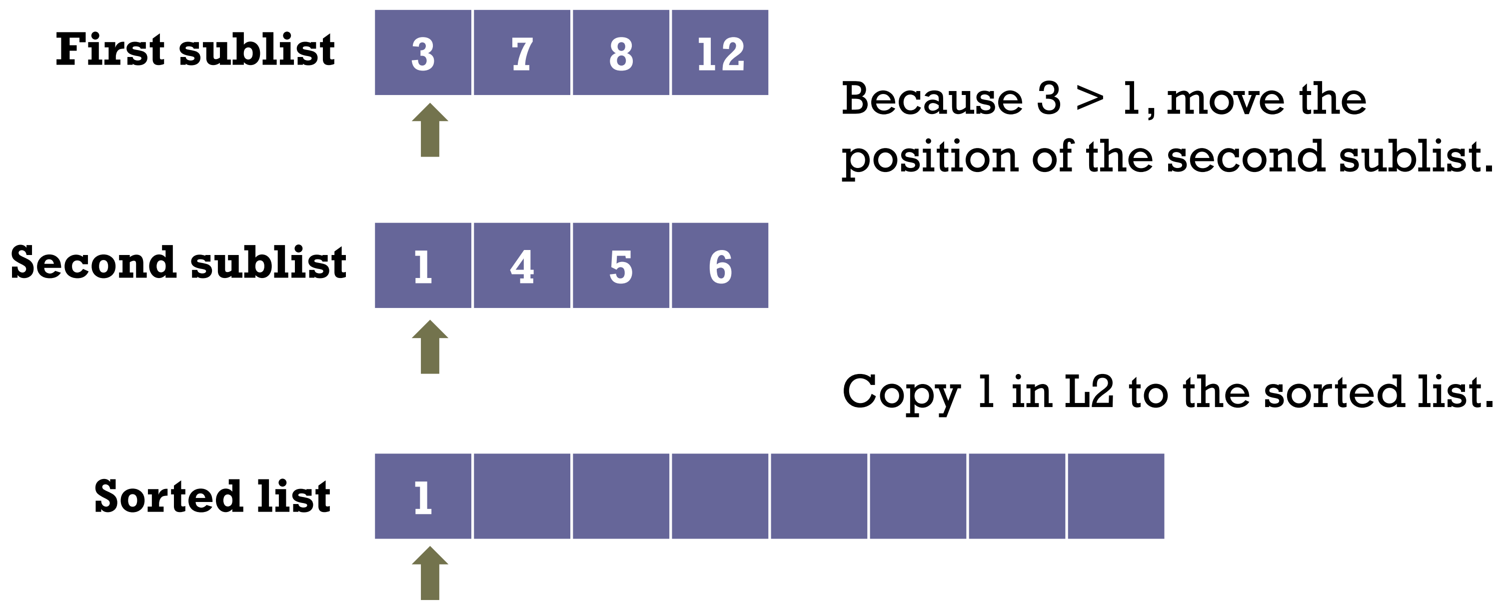
What is Merge Sort?

- Partitioning and merging in a recursive manner



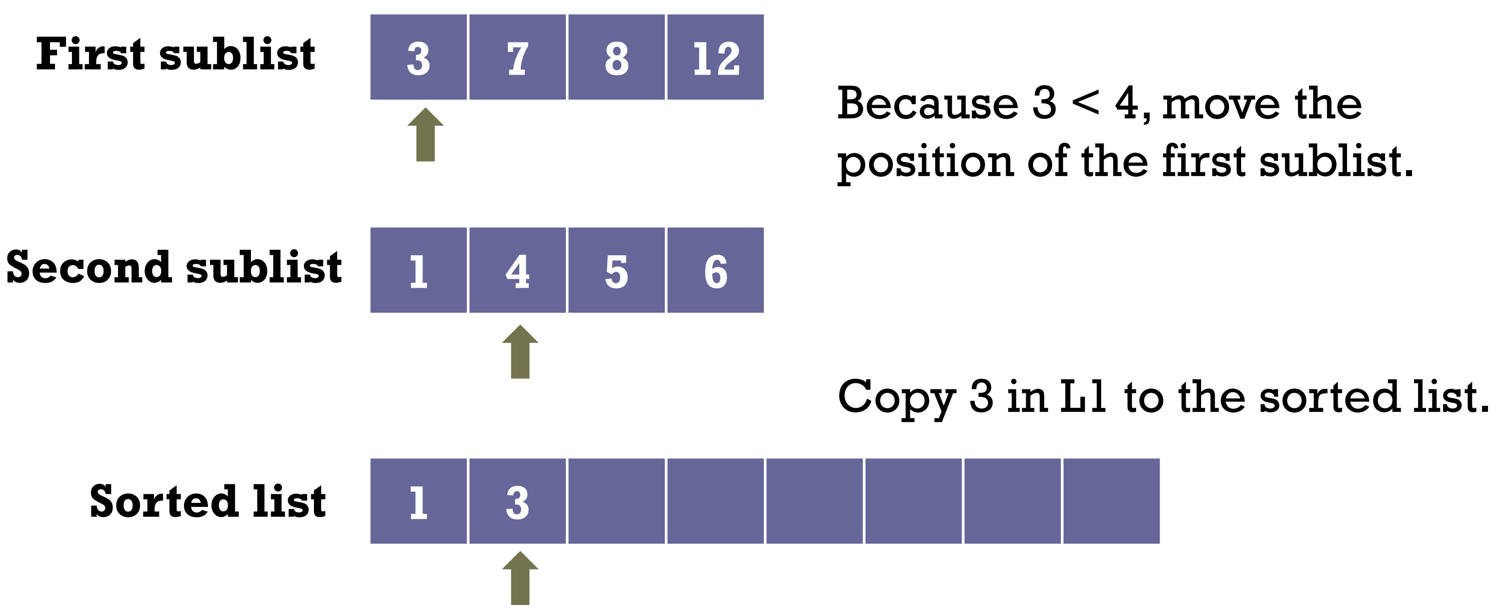
Merge Sort: Merging

- How to merge two sublists into one list?
  - Compare two elements in L1 and L2 in sequence.
    - If the element in L1 is less than or equal to that in L2, move to the next position in L1.
    - If the element in L1 is greater than that in L2, move to the next position in L2.



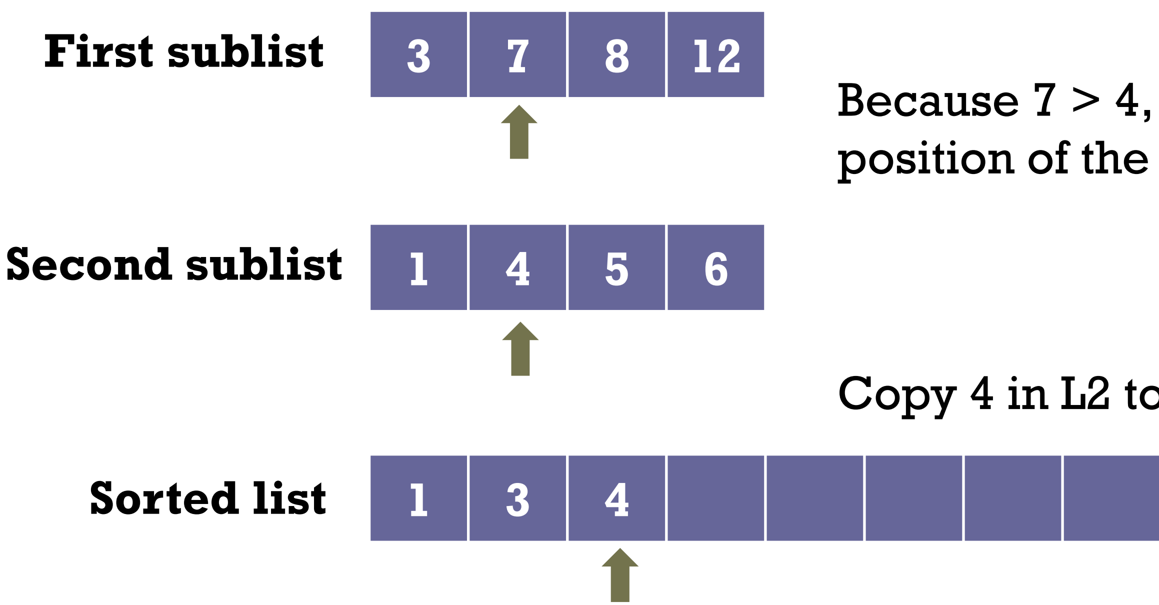
Merge Sort: Merging

- How to merge two sublists into one list?
  - Compare two elements in L1 and L2 in sequence.
    - If the element in L1 is less than or equal to that in L2, move to the next position in L1.
    - If the element in L1 is greater than that in L2, move to the next position in L2.



Merge Sort: Merging

- How to merge two sublists into one list?
  - Compare two elements in L1 and L2 in sequence.
    - If the element in L1 is less than or equal to that in L2, move to the next position in L1.
    - If the element in L1 is greater than that in L2, move to the next position in L2.

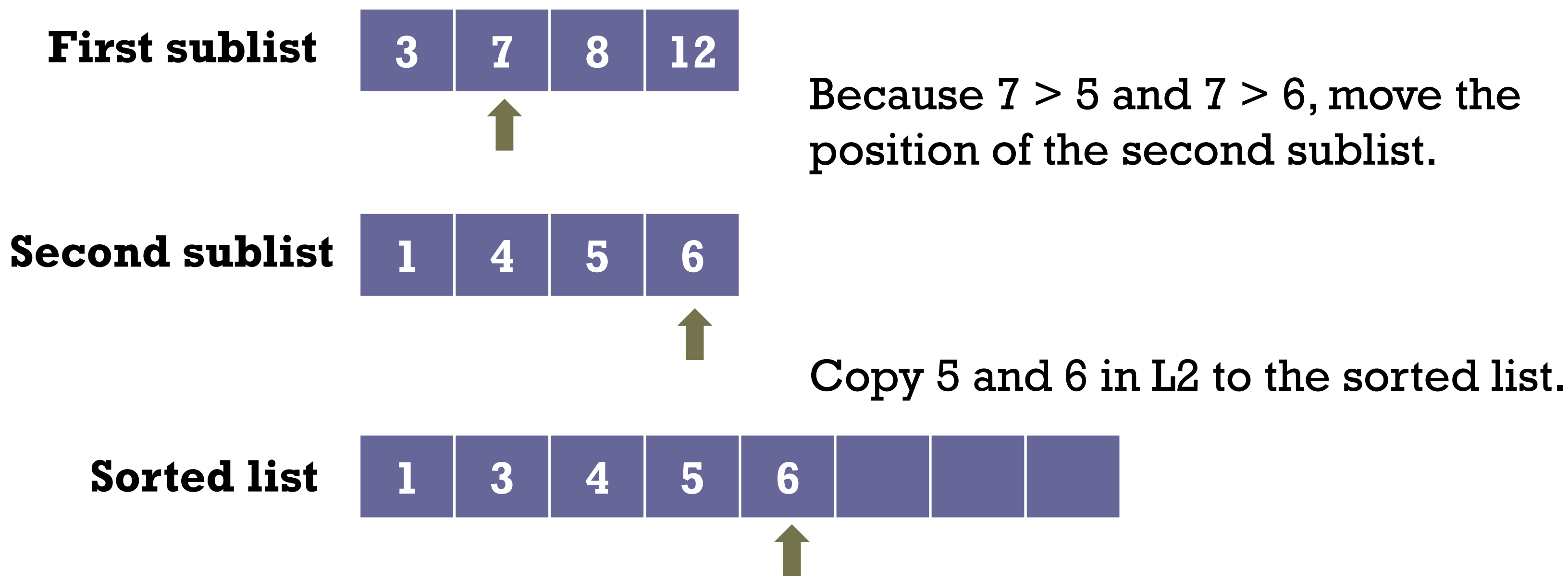


Because 7 > 4, move the position of the second sublist.

Copy 4 in L2 to the sorted list.

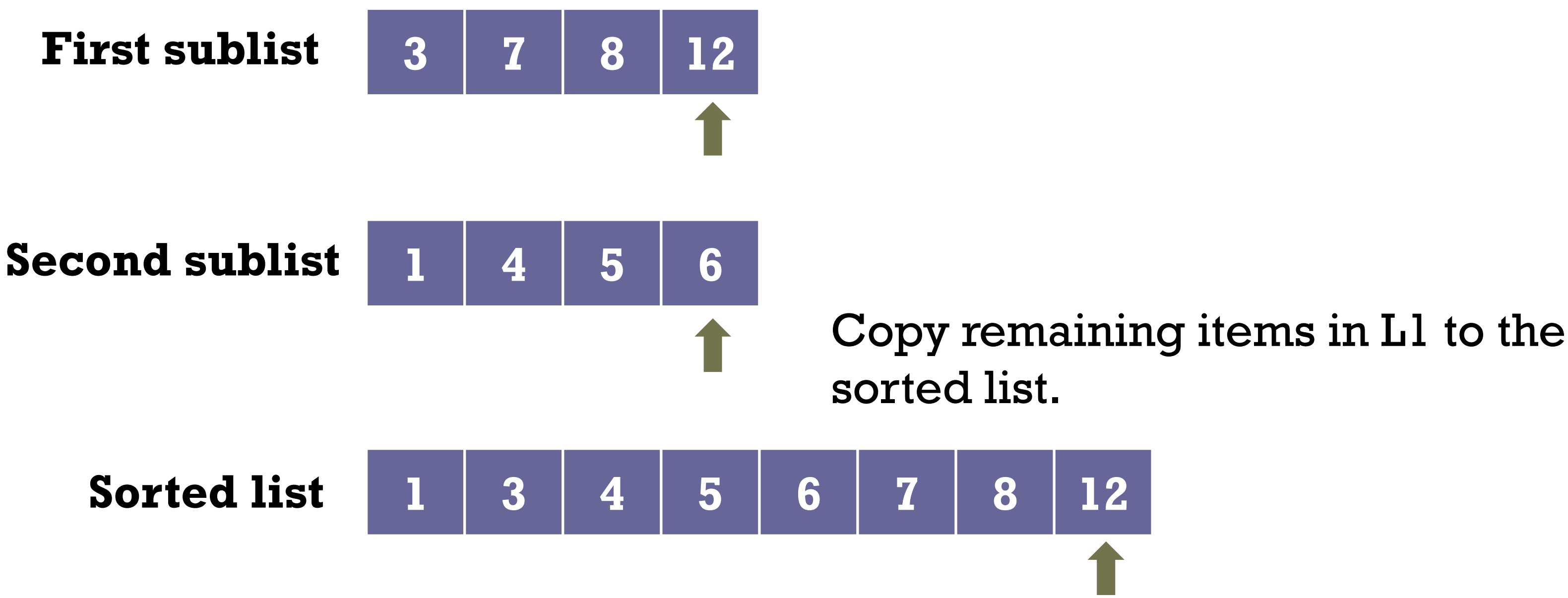
Merge Sort: Merging

- How to merge two sublists into one list?
  - Compare two elements in L1 and L2 in sequence.
    - If the element in L1 is less than or equal to that in L2, move to the next position in L1.
    - If the element in L1 is greater than that in L2, move to the next position in L2.



Merge Sort: Merging

- How to merge two sublists into one list?
  - Compare two elements in L1 and L2 in sequence.
    - If the element in L1 is less than or equal to that in L2, move to the next position in L1.
    - If the element in L1 is greater than that in L2, move to the next position in L2.



Implementation of Merging

```
void Merge(Data* list, int left, int mid, int right)
{
     int sorted[MAX_SIZE];
     int first = left, second = mid + 1, i = left;

     // Merge two lists by comparing elements in sequence.
     while (first <= mid && second <= right) {
          if (list[first] <= list[second])
               sorted[i++] = list[first++];
          else
               sorted[i++] = list[second++];
     }

     // For remaining items, add them in sequence.
     if (first > mid)
          for (int j = second; j <= right; j++)
               sorted[i++] = list[j];
     else
          for (int j = first; j <= mid; j++)
               sorted[i++] = list[j];

     // Copy the sorted list to the list.
     for (int j = left; j <= right; j++)
          list[j] = sorted[j];
}
```

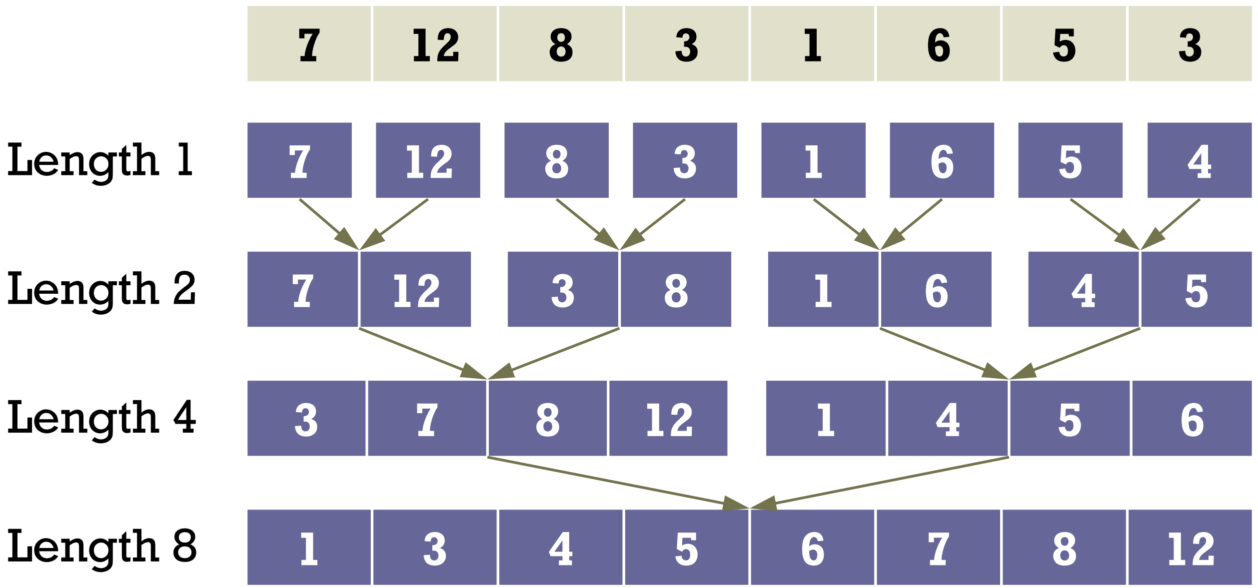
Implementation of Merge Sort

- Overall procedure
  - **Partitioning**: split the list into two halves.
  - **Merge**: Merge two sorted sublits into one list.
  - Recursively apply the above steps to the sublists.

```
void MergeSort(Data* list, int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2;    // Equal partitioning
        MergeSort(list, left, mid);      // Sorting sublists
        MergeSort(list, mid + 1, right); // Sorting sublists
        Merge(list, left, mid, right);   // Merging two sublists
    }
}
```

Iterative Merge Sort

- Iterative merge sort algorithm
  - Increase the length of merging two lists in sequence.



Iterative Merge Sort

```
void IterMergeSort(Data* list, int n)
{
	// Merge subarrays in bottom up manner.  First merge subarrays of
	// size 1 to create sorted subarrays of size 2, then merge subarrays
	// of size 2 to create sorted subarrays of size 4, and so on.
	for (int size = 1; size <= n - 1; size = 2 * size)
	{
		// Pick starting point of different subarrays of current size
		for (int left_start = 0; left_start < n - 1; left_start += 2 * size)
		{
			// Find ending point of left subarray.
			// mid+1 is starting  point of right
			int mid = left_start + size - 1;
			int right_end = MIN(left_start + 2 * size - 1, n - 1);

			// Merge Subarrays arr[left_start...mid] & arr[mid+1...right_end]
			Merge(list, left_start, mid, right_end);
		}
	}
}
```

Analysis of Merge Sort

- Time complexity
  - Split a list to into two sublists: $O(1)$
  - Sort two sublists: $2T(\frac{n}{2})$
  - Merge two sublists: cn
  - So, the recurrence relation is $T(n) = 2\left(\frac{T}{2}\right) + cn$.
  - The time complexity of merge sort is $\boldsymbol{O(nlog\, n)}$.
    - Average case and worst case are equal.

- Is it stable?
  - Yes, the merging procedure can maintain stability.

Comparison of Sorting Algorithms

Algorithm	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Radix sort	$O(dn)$	$O(dn)$	$O(dn)$

Comparison of Sorting Algorithms

- Comparing the running time (N = 100K)

Algorithm	Running time (sec)
Selection sort	10.842
Bubble sort	22.894
Insertion sort	7.438
Quick sort	0.014
Merge sort	0.026
Heap sort	0.034

Summary of Sorting Algorithms

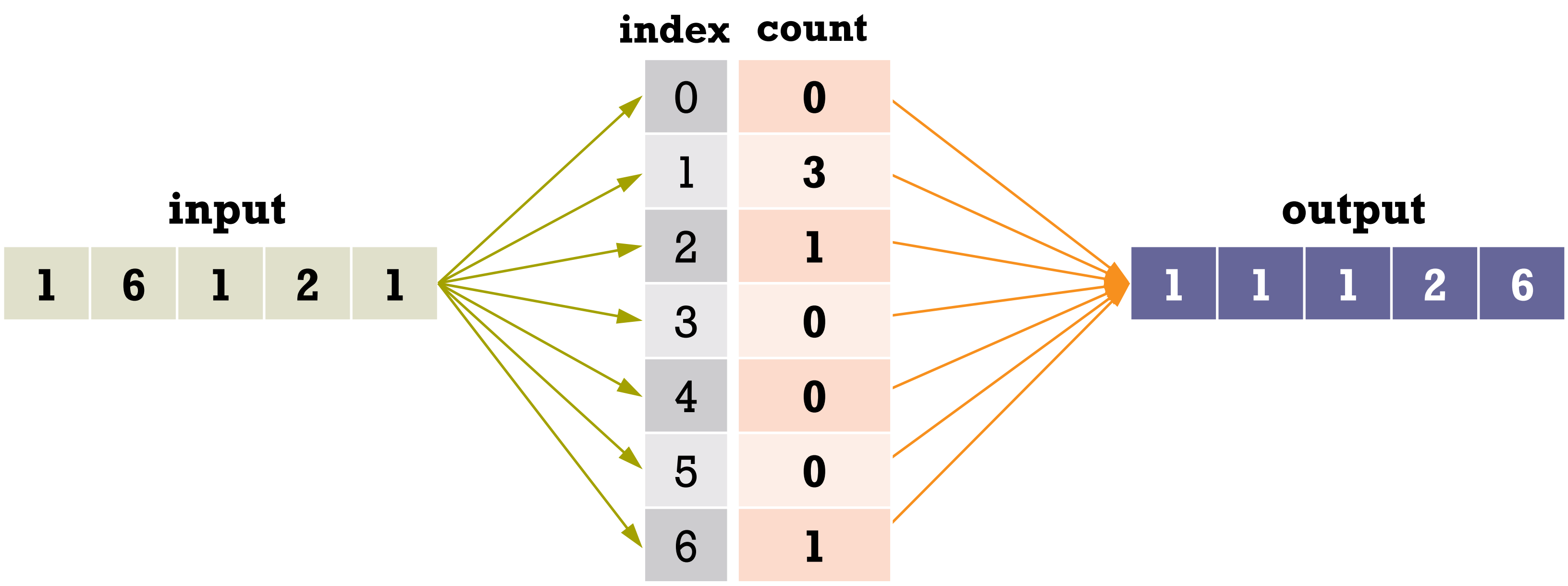
- Pros & cons
  - Insertion sort
    - Best for almost sorted: $\boldsymbol{O(n)}$
    - Best for small # of elements
  - Quick sort
    - Best in average case
    - Worse case: $\boldsymbol{O(n^2)}$
  - Merge sort
    - Best in the worst case: $\boldsymbol{O(n \log n)}$

- **Combination of sorting algorithms**
  - Insertion sorting is the fastest when $n < 20$.
  - Quick sorting is the fastest when $20 < n < 45$.
  - Merge sorting is the fastest when n is large.

Non-comparison sorting algorithms

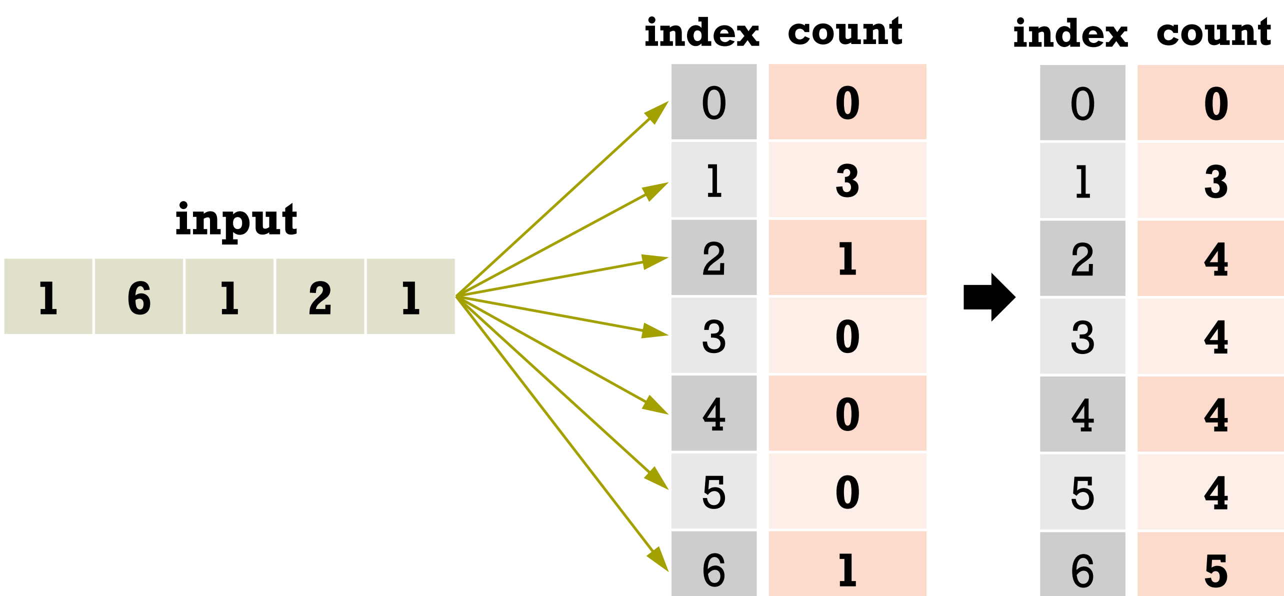
What is Counting Sort?

- Description
  - **Non-comparison** sorting algorithm
  - Count the number of elements with **distinct** key values.
  - Output each element from the input sequence followed by **decreasing its count by one**.



Example of Counting Sort

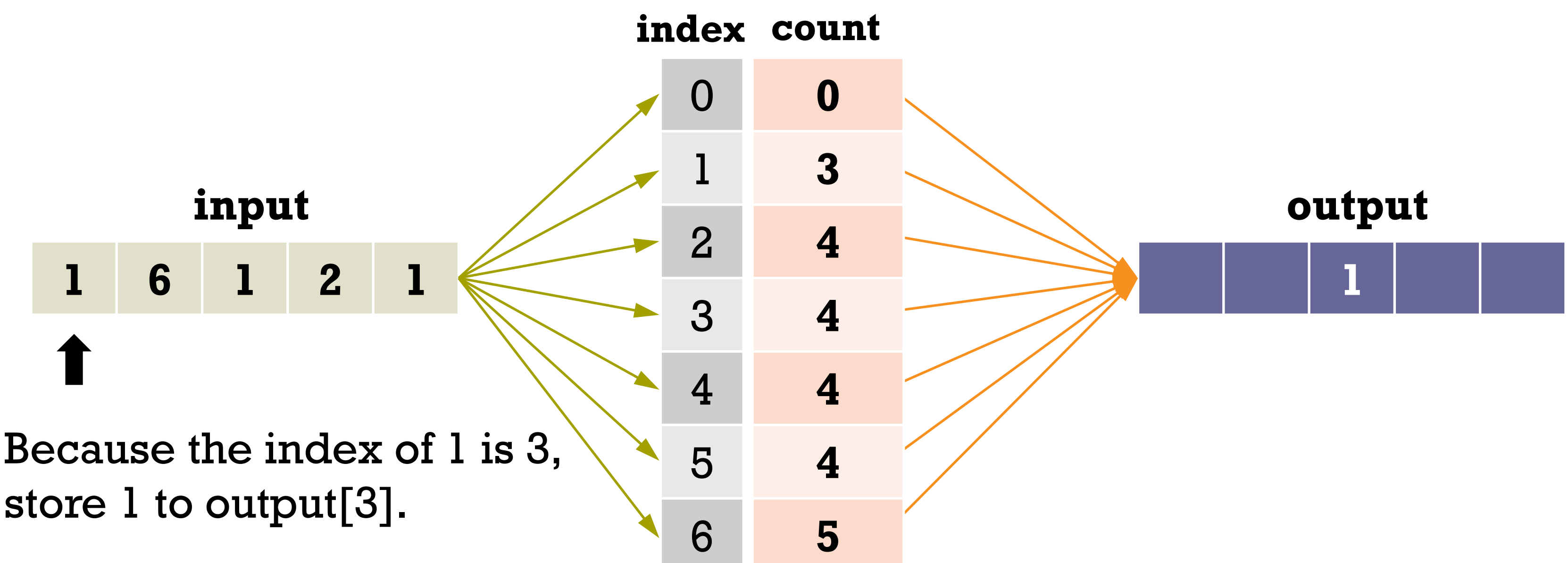
- Description
  - Count the number of elements with **distinct** key values.
    - Determine the **positions** of key values in the output.
  - Output each element from the input sequence followed by **decreasing its count by one**.



46

Example of Counting Sort

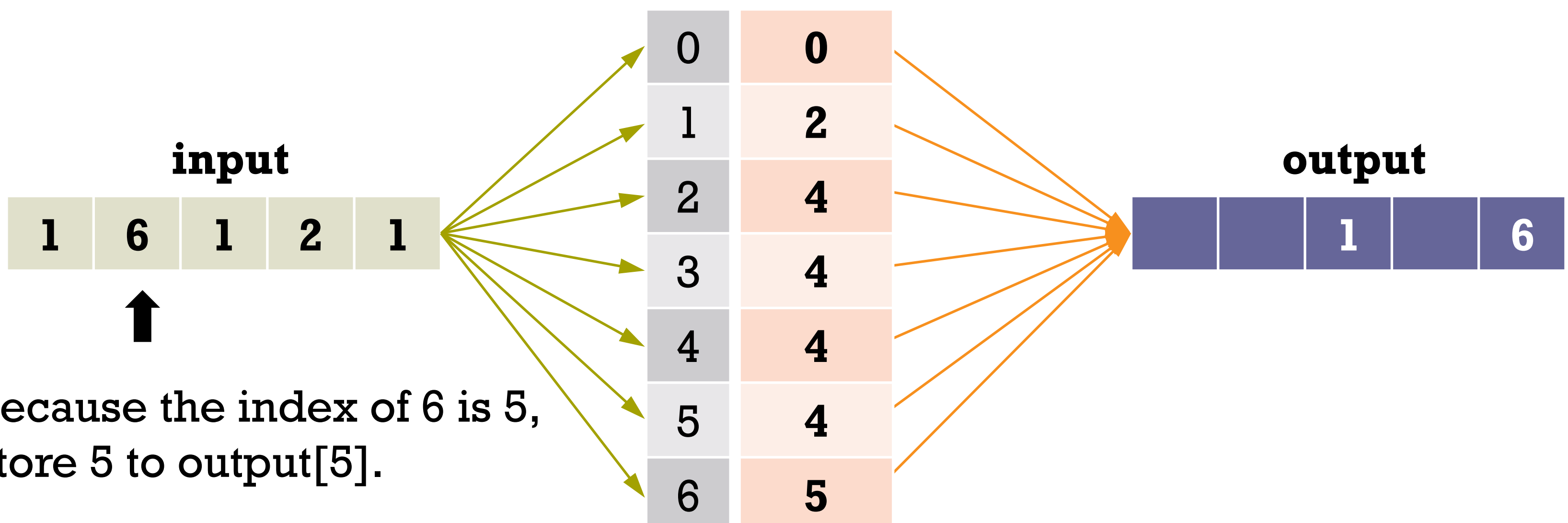
- Description
  - Count the number of elements with **distinct** key values.
    - Determine the **positions** of key values in the output.
  - Output each element from the input sequence followed by **decreasing its count by one**.



47

Example of Counting Sort

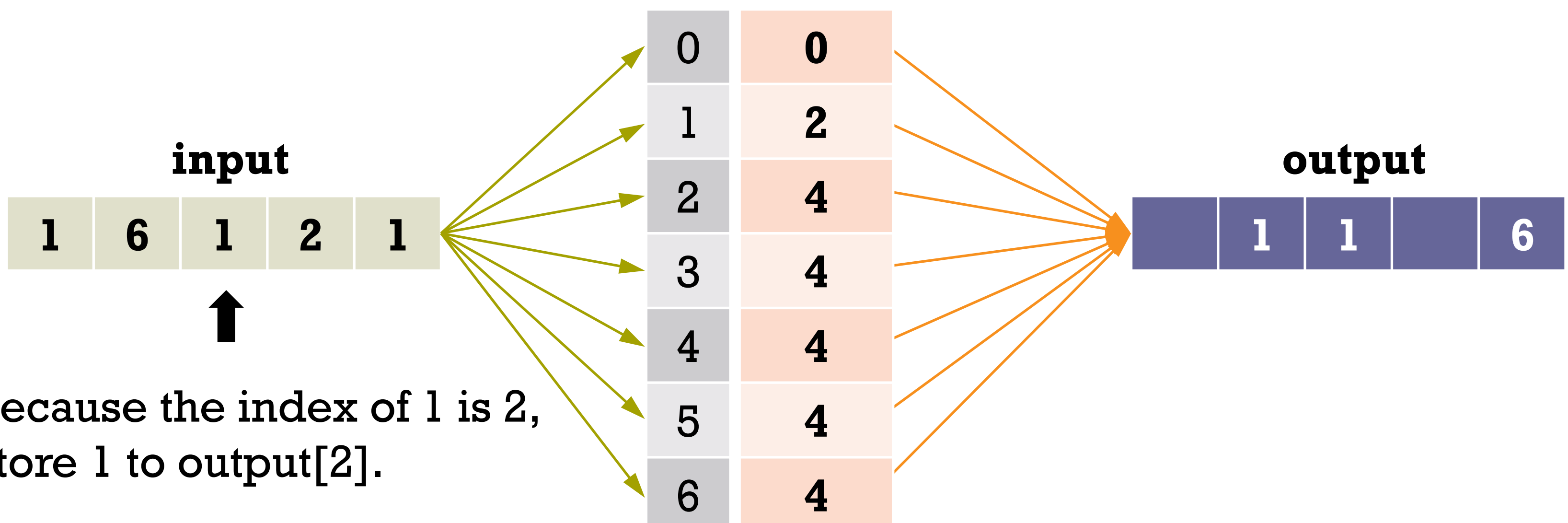
- Description
  - Count the number of elements with **distinct** key values.
    - Determine the **positions** of key values in the output.
  - Output each element from the input sequence followed by **decreasing its count by one**.



Because the index of 6 is 5,
store 5 to output[5].

Example of Counting Sort

- Description
  - Count the number of elements with **distinct** key values.
    - Determine the **positions** of key values in the output.
  - Output each element from the input sequence followed by **decreasing its count by one**.

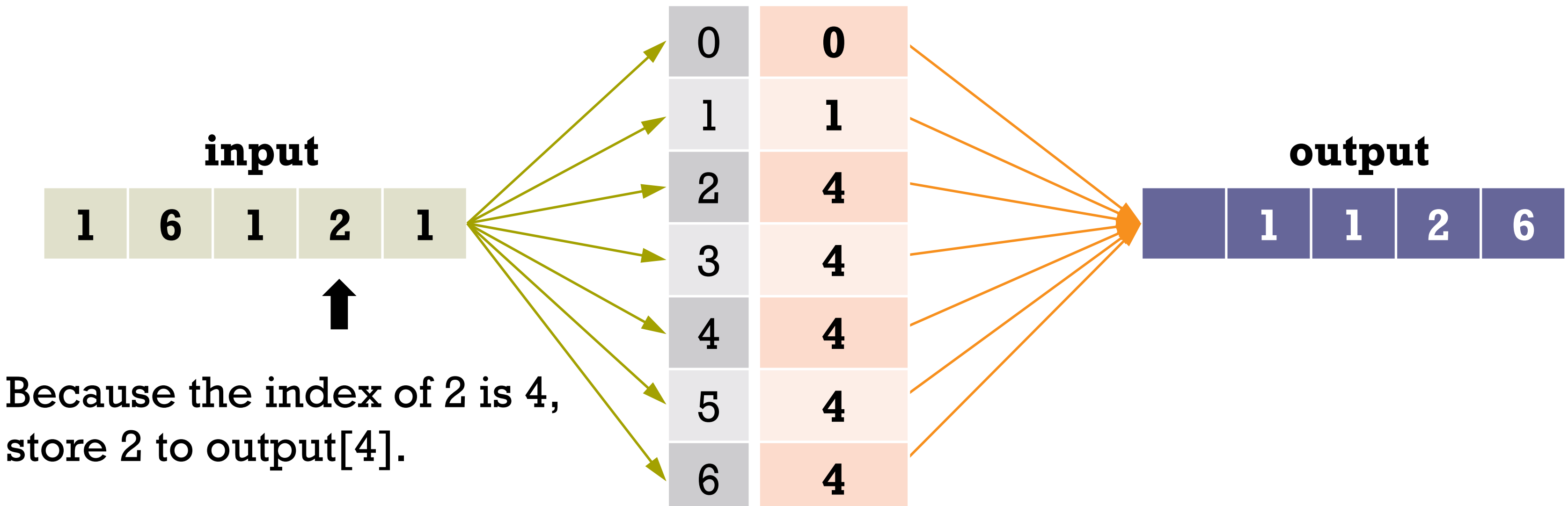


Because the index of 1 is 2, store 1 to output[2].

49

Example of Counting Sort

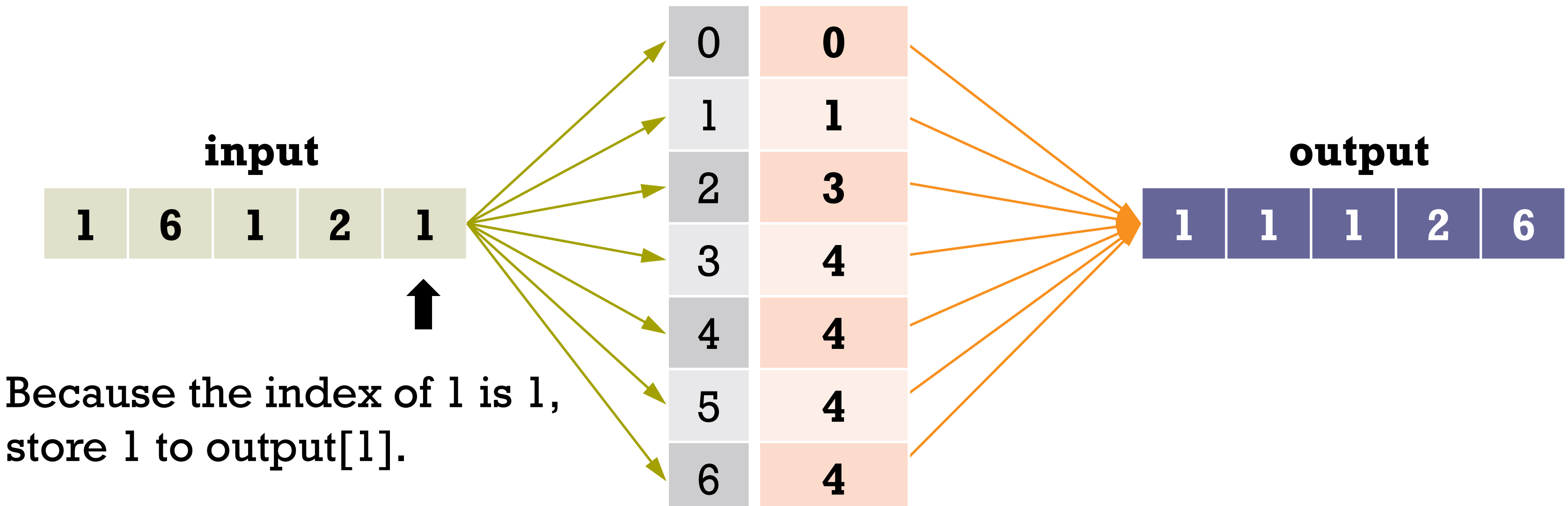
- Description
  - Count the number of elements with **distinct** key values.
    - Determine the **positions** of key values in the output.
  - Output each element from the input sequence followed by **decreasing its count by one**.



50

Example of Counting Sort

- Description
  - Count the number of elements with **distinct** key values.
    - Determine the **positions** of key values in the output.
  - Output each element from the input sequence followed by **decreasing its count by one**.

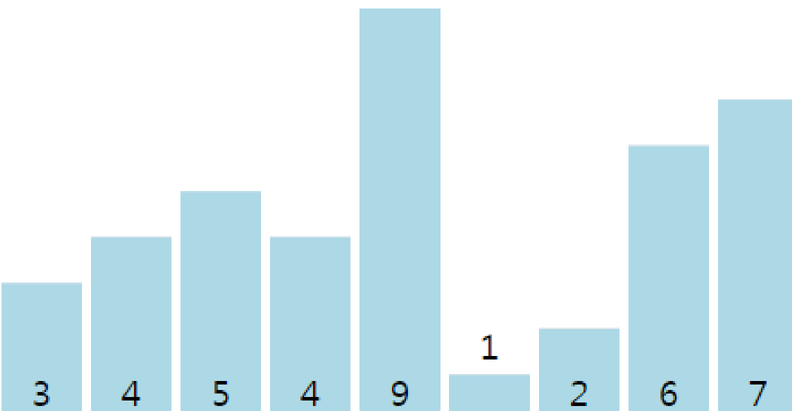


Implementation of Counting Sort

```
void CountingSort(Data* list, int n)
{
     Data count[MAX_SIZE] = { 0 };
     Data output[MAX_SIZE];

     // Counting the redundant elemnts
     for (int i = 0; i < n; i++)
          count[list[i]]++;

     // Cumulate the number of elements.
     for (int i = 1; i < MAX_SIZE; i++)
          count[i] += count[i - 1];

     // Read the elements in the list and copy them to the output list.
     for (int i = 0; i < n; i++) { // this is unstable
          output[count[list[i]] - 1] = list[i];
          count[list[i]]--;
     }

     // Copy the output list to the original list.
     for (int i = 0; i < n; i++)
          list[i] = output[i];
}
```

Exercise: Counting Sort

- Animation: sorting 3, 4, 5, 4, 9, 1, 2, 6, 7
  - Draw the step-by-step procedure of counting sort.
  - https://visualgo.net/en/sorting



Initial list

- Q: Is it stable?

53

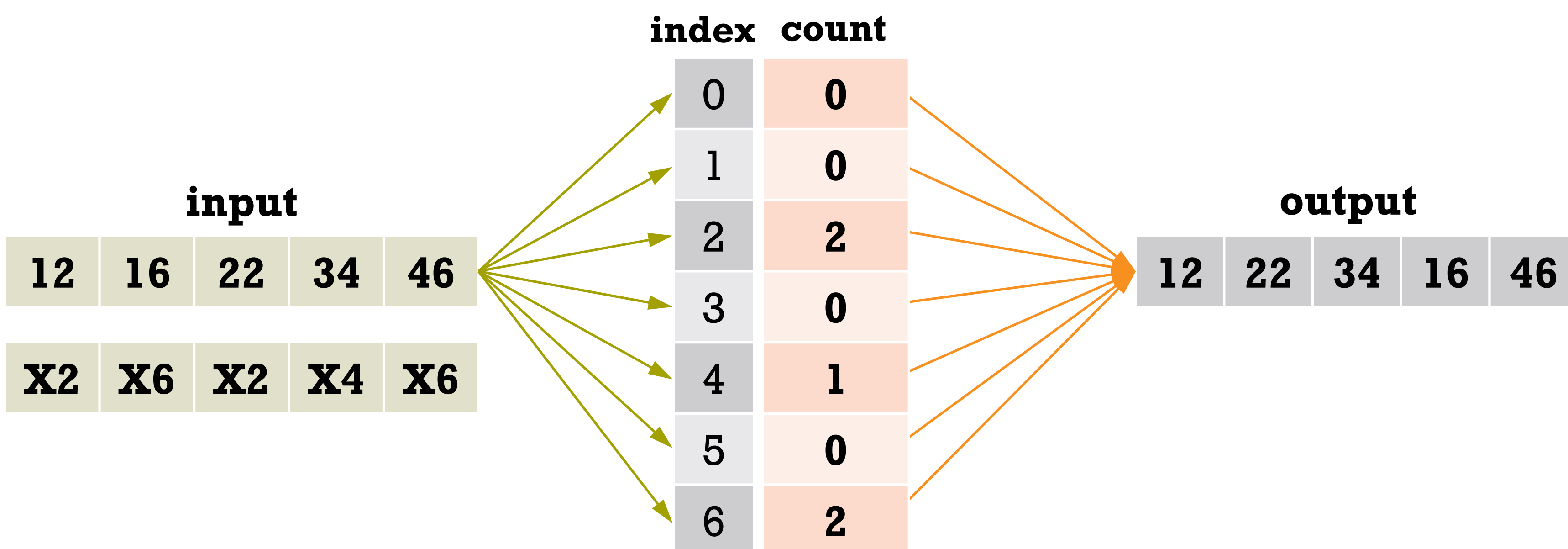
Analysis of Counting Sort

- Characteristics
  - The time complexity is **linear** in the number of items and the difference between the maximum and minimum key values.
    - It is only suitable for the case where **the variation in keys is not significantly greater than the number of items**.

  - It is often used as a subroutine in another sorting algorithm, **radix sort**, that can handle larger keys more efficiently.

What is Radix Sort?

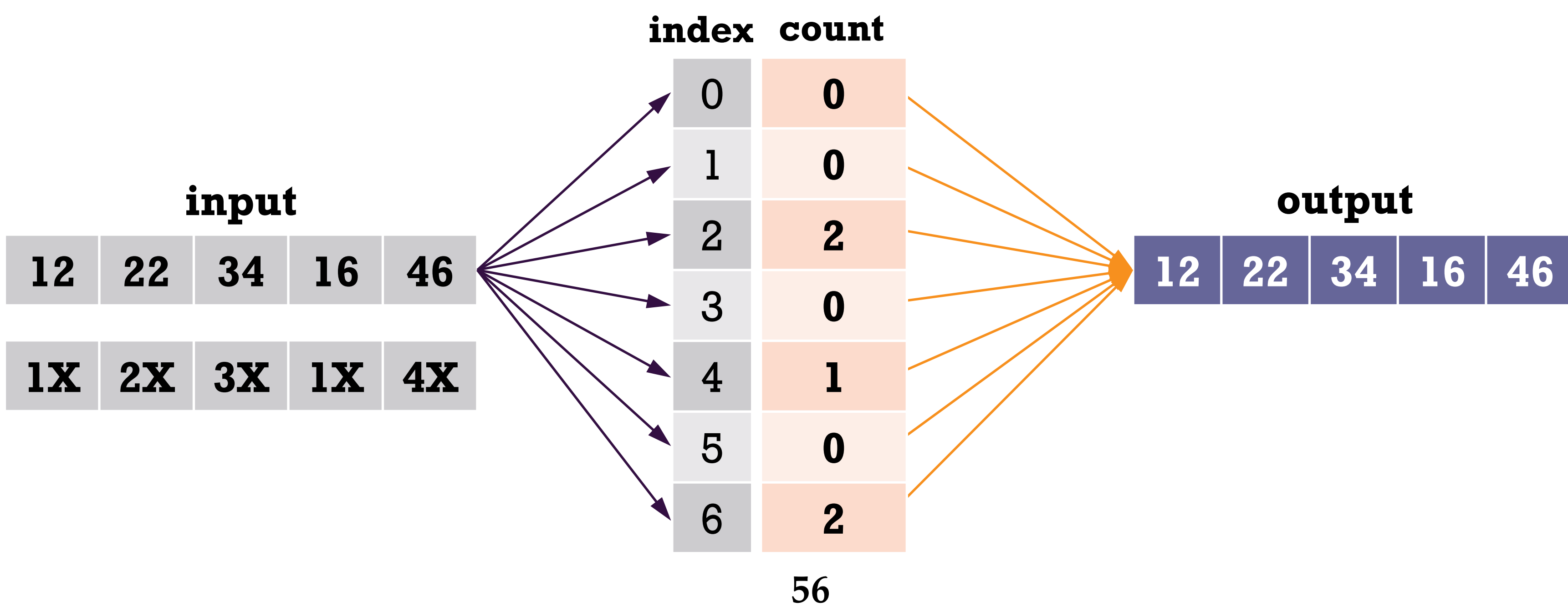
- Description
  - **Non-comparison** sorting algorithm
  - Grouping keys by the individual digits which share the same significant position and value.



55

What is Radix Sort?

- Description
  - **Non-comparison** sorting algorithm
  - Grouping keys by the individual digits which share the same significant position and value.



Implementation of Radix Sort

```
void Counting(int list[], int n, int exp)
{
     int count[10] = { 0 };
     int output[MAX_SIZE];

     // Store count of occurrences in count list.
     for (int i = 0; i < n; i++)
          count[(list[i] / exp) % 10]++;

     // Change count[i] so that count[i] contains actual position of this
digit in output list.
     for (int i = 1; i < 10; i++)
          count[i] += count[i - 1];

     // Build the output list.
     for (int i = n - 1; i >= 0; i--) { // this is stable
          output[count[(list[i] / exp) % 10] - 1] = list[i];
          count[(list[i] / exp) % 10]--;
     }

     // Copy the output list to list[], so that list[] now
     // contains sorted numbers according to current digit
     for (int i = 0; i < n; i++)
          list[i] = output[i];
}
```

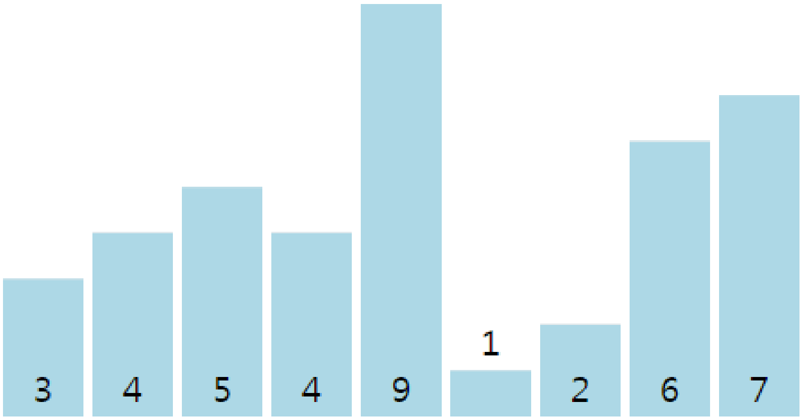
Implementation of Radix Sort

- Implementation

```
void RadixSort(Data* list, int n)
{
    // Find the maximum number to know the number of digits.
    int max = list[0];
    for (int i = 1; i < n; i++) {
        if (list[i] > max)
            max = list[i];
    }

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; max / exp > 0; exp *= 10)
        Counting(list, n, exp);
}
```

Exercise: Radix Sort

- Animation: sorting 3, 4, 5, 4, 9, 1, 2, 6, 7
  - Draw the step-by-step procedure of radix sort.
  - https://visualgo.net/en/sorting



Initial list

- Q: Is it stable?

Analysis of Radix Sort

- Time complexity
  - For each digit, perform counting sort.
  - The time complexity of radix sort is $\boldsymbol{O(dn)}$.
    - d is the maximum number of digits.
    - n is the number of elements.