Data Structure

WF

http://smartlead.hallym.ac.kr

Instructor: Jin Kim

010-6267-8189(033-248-2318)

jinkim@hallym.ac.kr

Office Hours:

$O(n)$정렬 알고리즘

Counting sort(계수정렬)

Radix sort(기수정렬)

Bucket sort(버킷정렬)

기타정렬 알고리즘

Shell sort(쉘정렬)

Beating the lower bound(하한값)

- We can beat the lower bound if we don't base our sort on comparisons: 비교하지 않으면하한값을 낮출수있다(더빠를수있다)
  - Counting sort for keys in [0..k], k=O(n)
  - Radix sort for keys with a fixed number of "digits"
  - Bucket sort for random keys (uniformly distributed)

Linear Time Sorting(선형시간정렬)

- Algorithms exist for sorting n items in Θ(n) time IF we can make some assumptions about the input data(입력데이터가어떤 특징이 있다고 가정하면, 실행시간은 Θ(n)이 될 수 있다)
- These algorithms do not sort solely by comparisons, thus avoiding the Ω (n log n) lower bound on comparison-based sorting algorithms (키를 비교하지 않으므로 하한값이 Ω (n log n)보다 작아질수있음)
- Stable, additional space required
  - Counting sort(계수 정렬) //안정적, 추가적인 공간 필요
  - Radix Sort(기수 정렬) // 안정적, 추가적인 공간 필요
  - Bucket Sort(버킷 정렬) // 안정적, 추가적인 공간필요

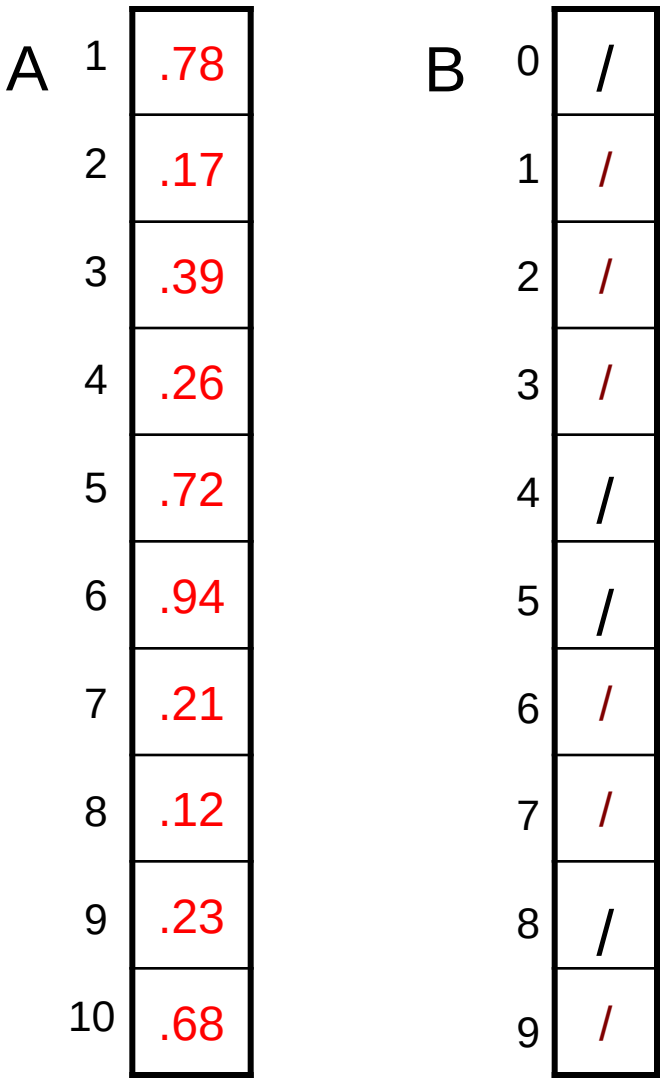
Bucket sort(버킷 정렬)

Bucket Sort(버킷 정렬)

안정적인 정렬 알고리즘

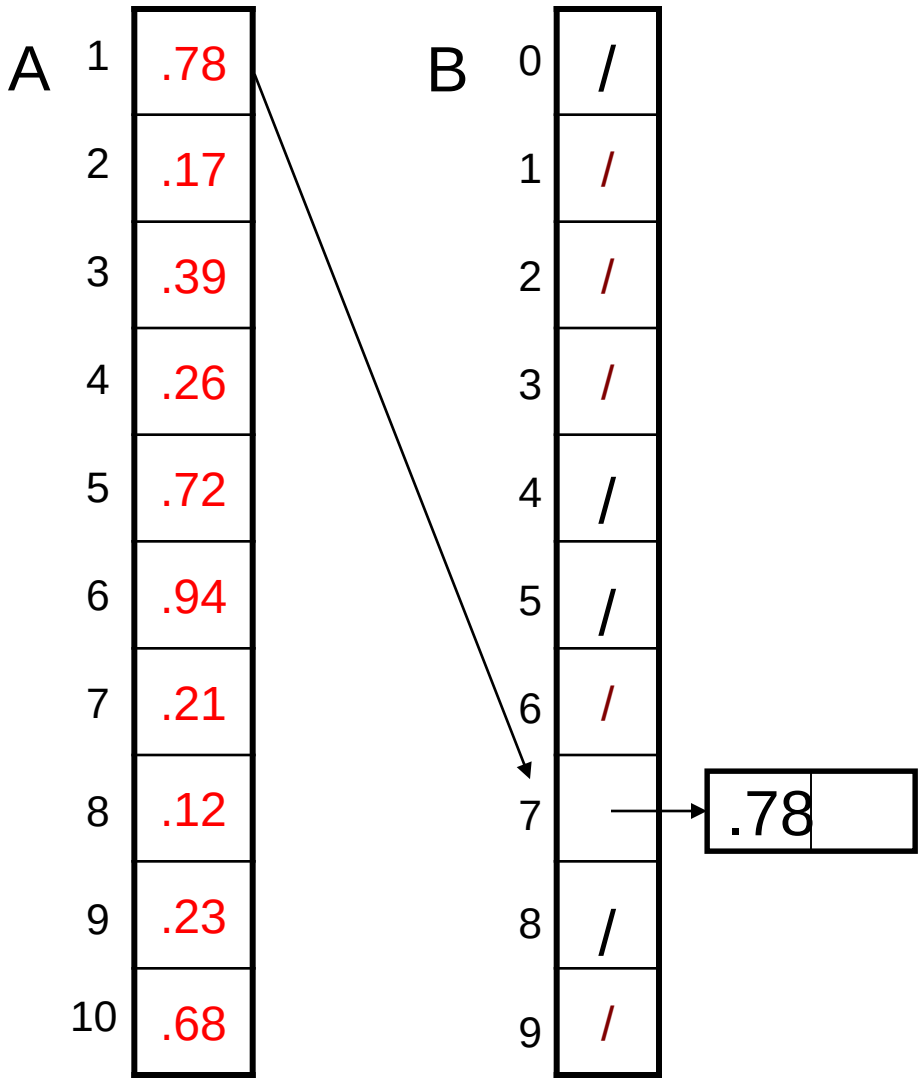
- Assumes input is generated by a random process that distributes the elements uniformly over [0, 1).
- 원소가 [0,1)사이에 균일하게 분포되어 있다 가정
- **Idea:**
  - Divide [0, 1) into *n* equal-sized buckets.
  - Distribute the *n* input values into the buckets.
  - Sort each bucket.
  - Then go through the buckets in order, listing elements in each one.

Example - Bucket Sort



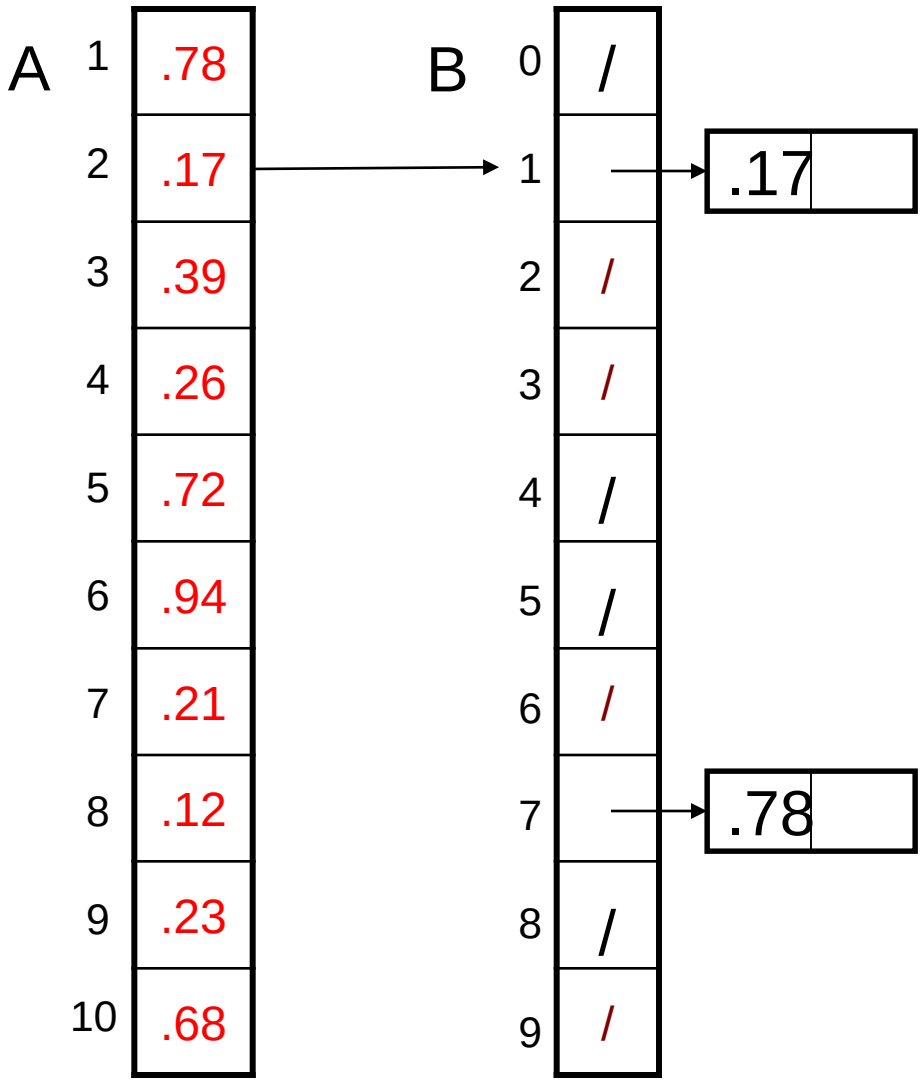
Distribute
Into buckets

Example - Bucket Sort



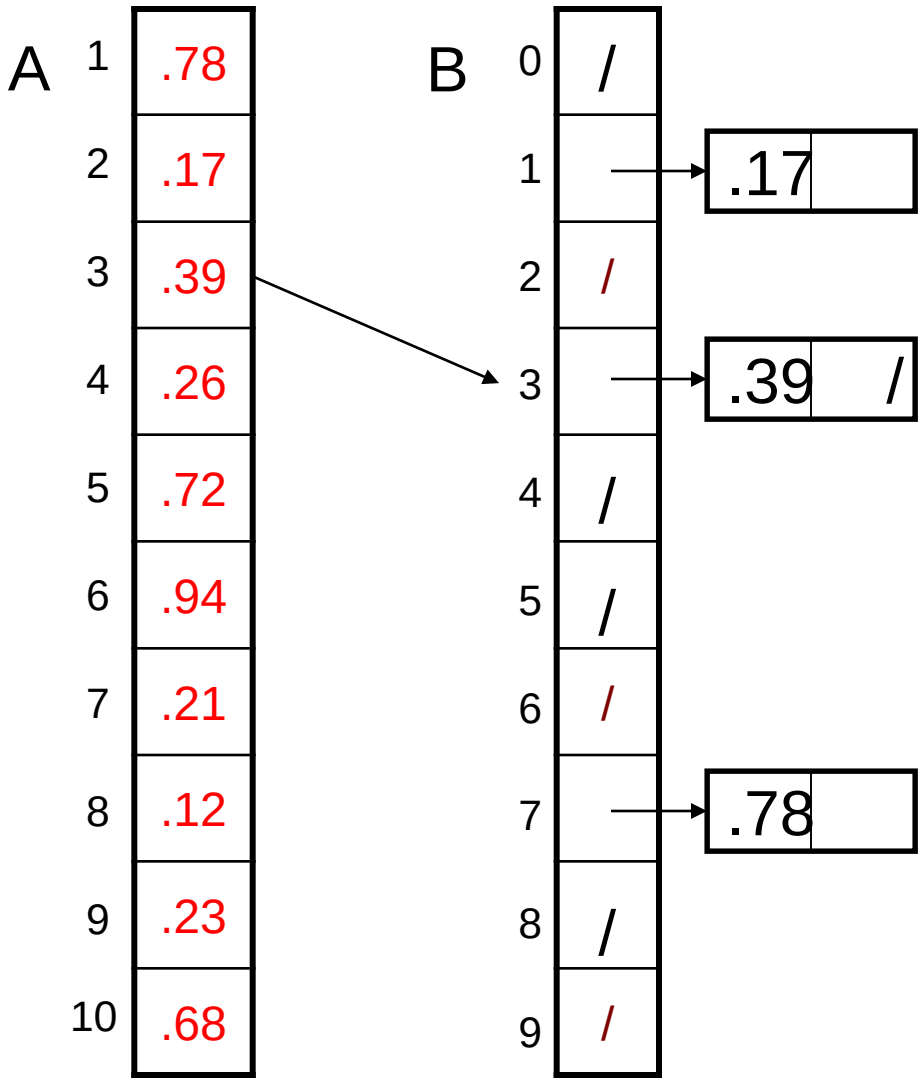
Distribute
Into buckets

Example - Bucket Sort



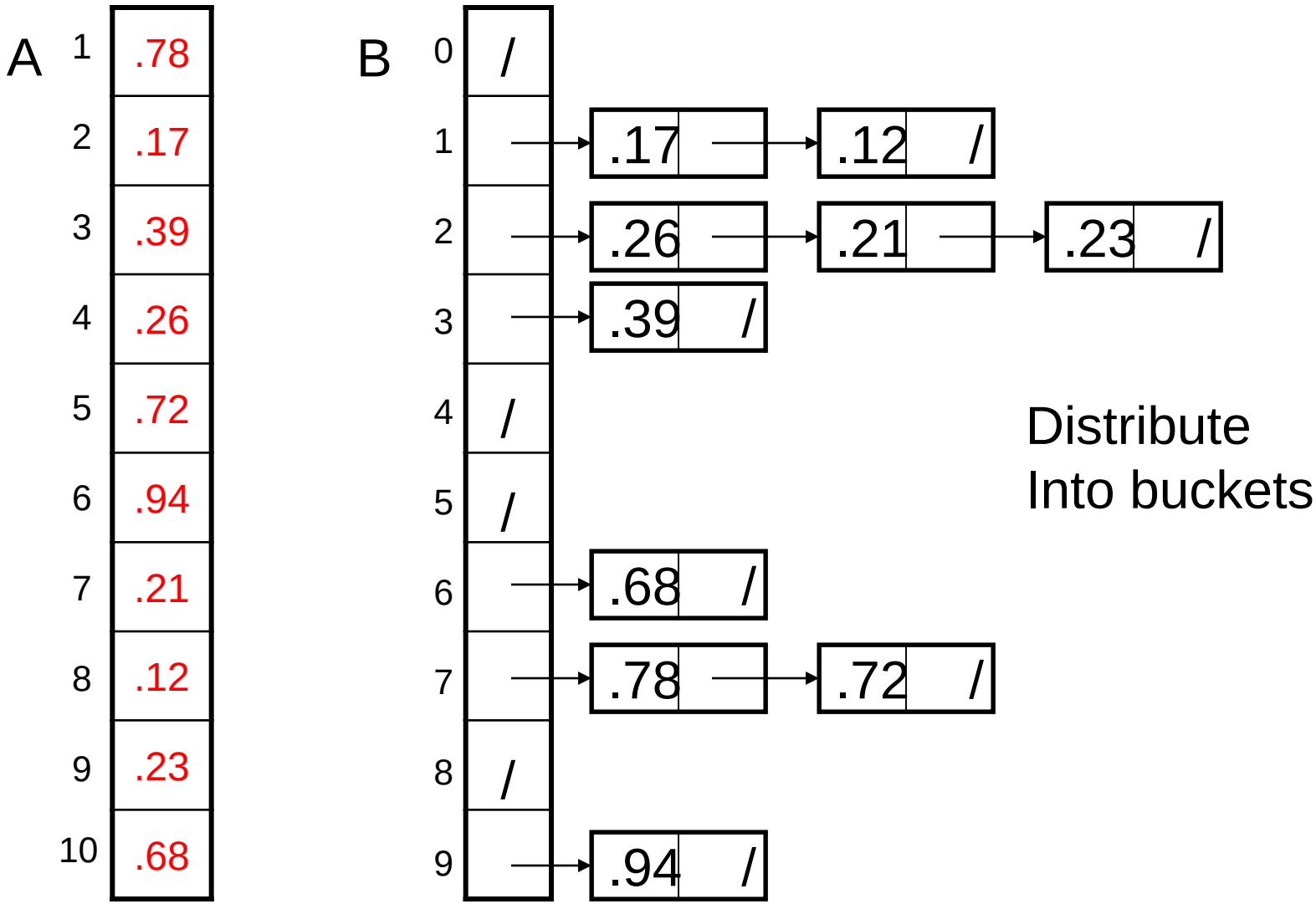
Distribute
Into buckets

Example - Bucket Sort

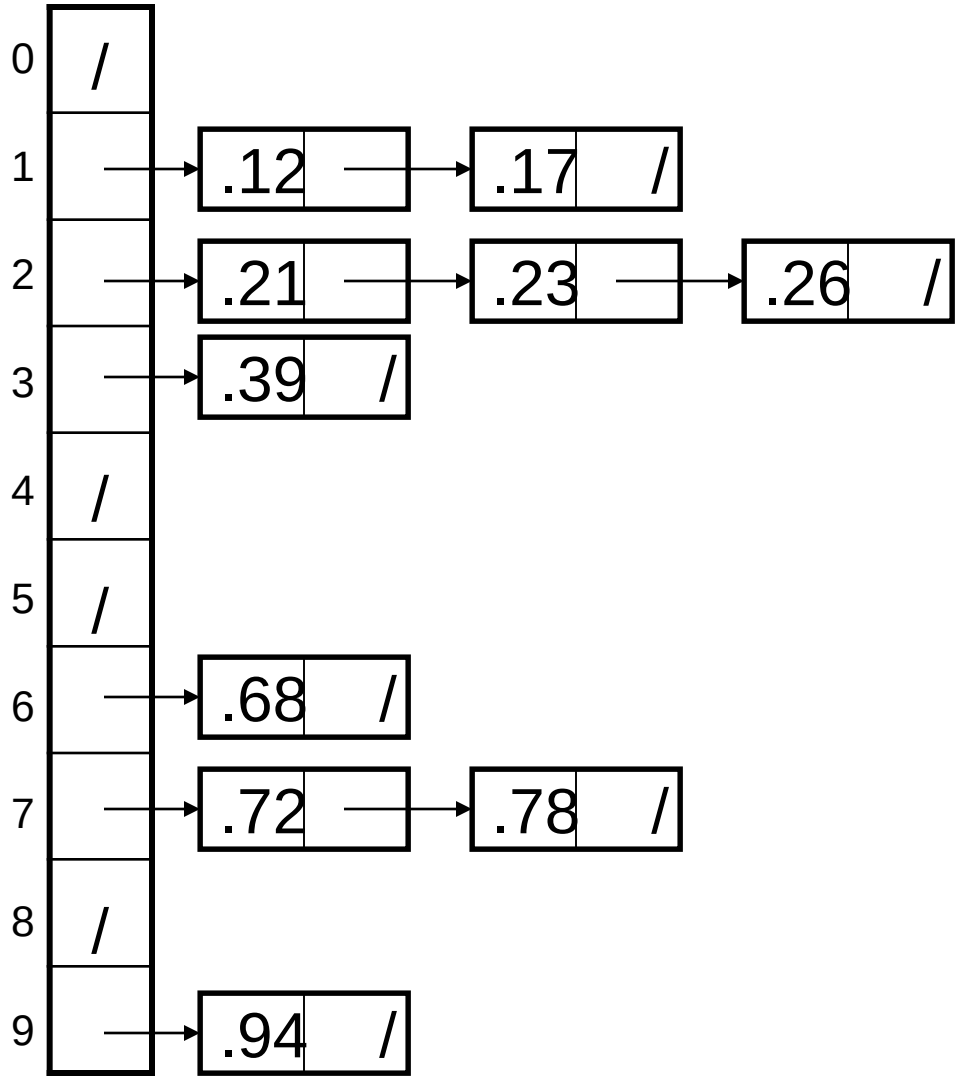


Distribute
Into buckets

Example - Bucket Sort

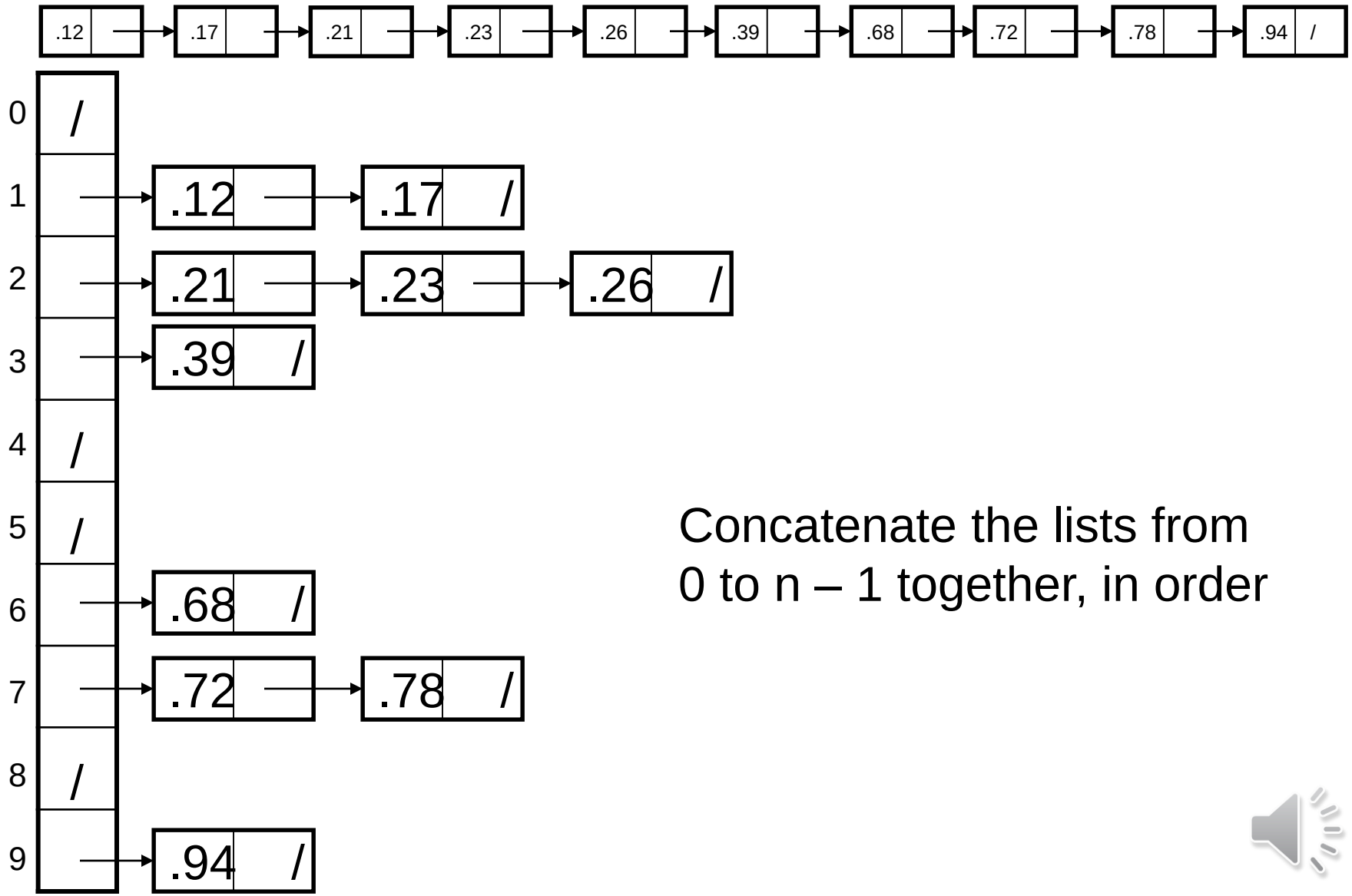


Example - Bucket Sort



Sort within each bucket

Example - Bucket Sort

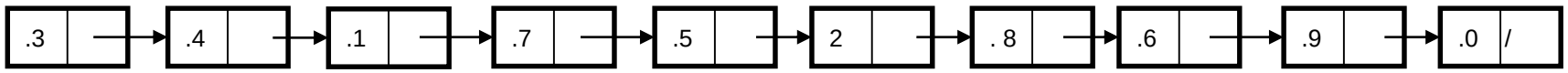


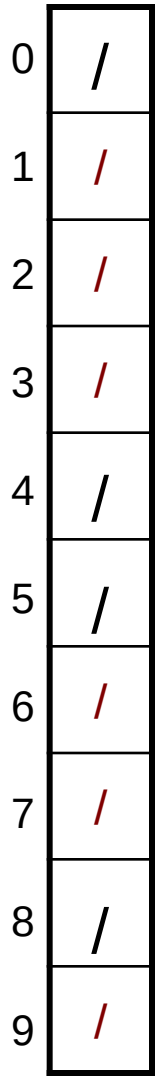
Concatenate the lists from
0 to n – 1 together, in order

만일 버킷하나에 하나의 원소만 입력가능하도록 버킷을 만들면 버킷내에서 정렬할 필요가 없다.

(If we put only one element into bucket(number of bucket is large enough), we do not need sort the element within the bucket.

Example - Bucket Sort





One bucket for one element.
No need to sort in each bucket

Analysis of Bucket Sort

BUCKET-SORT(A, n)

for i ← 1 **to** n

    do insert A[i] into list B[⌊nA[i]⌋] — O(n)

for i ← 0 **to** k - 1

        do sort list B[i] with quicksort sort — k O(n/k log(n/k)) =O(nlog(n/k)

concatenate lists B[0], B[1], . . . , B[n -1] together in order

return the concatenated lists — O(k)

O(n) (if k=Θ(n))

Example bucket sort

```
public class Main
{
  public static int[] bucket_sort(int[] arr, int max_value)
   {
      int[] bucket = new int[max_value + 1];
      int[] sorted_arr = new int[arr.length];

      for (int i= 0; i <arr.length; i++)
         bucket[arr[i]]++;

      int pos = 0;
      for (int i = 0; i < bucket.length; i++)
         for (int j = 0; j < bucket[i]; j++)
            sorted_arr[pos++] = i;

      return sorted_arr;
   }
```

```
static int maxValue(int[] arr)
{
   int max_value = 0;
   for (int i = 0; i < arr.length; i++)
      if (arr[i] > max_value)
         max_value = arr[i];
   return max_value;
}

public static void main(String args[])
{
   int[] arr ={80, 50, 30, 10, 90, 60, 0, 70, 40, 20, 50};
   int max_value = maxValue(arr);

   System.out.print("\nOriginal : ");
   System.out.println(Arrays.toString(arr));

   System.out.print("\nSorted : ");
   System.out.println(Arrays.toString(bucket_sort(arr,max_value)));

}
```

At most one element in one bucket
(한 버킷에는 최대 하나의 원소저장)

When is bucket sort most suitable?

언제 버킷정렬을 사용?

When the input is uniformly distributed (입력데이터가 균일하게 분포되어있을때 최적)

stable sort안정적인 정렬 알고리즘이다.

not in-place sort algorithm(제자리정렬 알고리즘이 아니다. 원소의 개수에 비례하는 추가적인 공간필요)

Counting sort(계수정렬)

Counting Sort(계수 정렬)

- Assumption: Keys to be sorted have a limited finite range, say [0 .. k-1]
- Method:
  - Count number of items with value exactly i
  - Compute number of items with value at most i
  - Use counts for placement of each item in final array
    - Full details in book
- Running time: $\Theta(n+k)$
- Key observation: Counting sort is stable (안정적)
- 원소의 개수에 비례하는 추가적인 공간

Counting Sort

- Counting sort assumes that each of the n input elements is an integer in the **range** 0 to k

- When k=O(n), the sort runs in O(n) time

Counting Sort

- Counting sort assumes that each of the n input elements is an integer in the **range** 0 to k

- When k=O(n), the sort runs in O(n) time

Counting Sort

- **Approach**
  - Sorts keys with values over range 0..k
  - Count number of occurrences of each key
  - Calculate # of keys < each key
  - Place keys in sorted location using # keys counted

Review: Counting Sort

- Counting sort:
  - Assumption: input is in the range 1..k
  - Basic idea:
    - Count number of elements $k \leq$ each element i
    - Use that number to place i in position k of sorted array
  - No comparisons! Runs in time O(n + k)
  - Stable sort
  - Does not sort in place: //제자리정렬아님
    - O(n) array to hold sorted output
    - O(k) array for scratch storage //k만큼 추가적인 공간

Review: Counting Sort

```
1       CountingSort(A, B, k)
2             for i=1 to k
3                   C[i]= 0;
4             for j=1 to n
5                   C[A[j]] += 1;
6             for i=2 to k
7                   C[i] = C[i] + C[i-1];
8             for j=n downto 1
9                   B[C[A[j]]] = A[j];
10                  C[A[j]] -= 1;
```

Counting Sorting

Original list

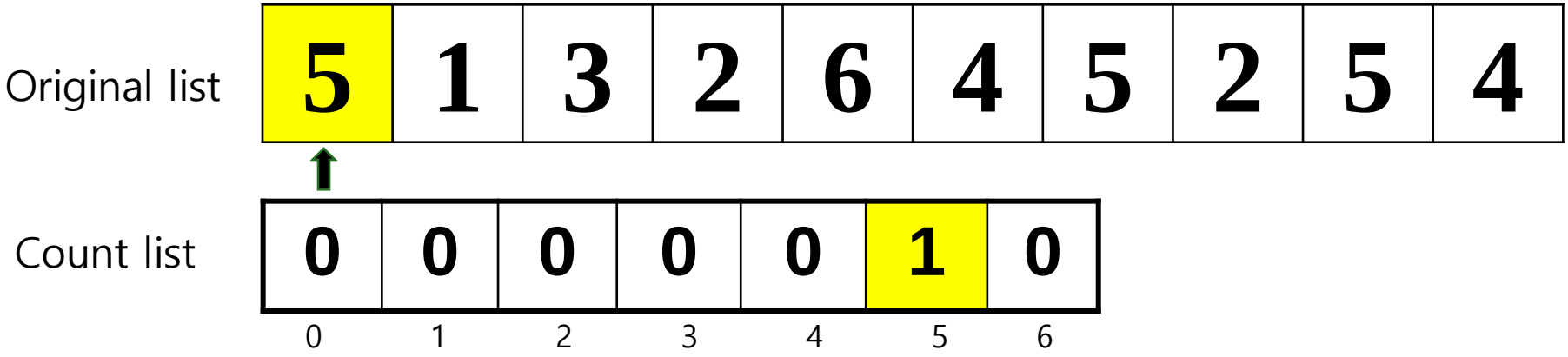
5	1	3	2	6	4	5	2	5	4

Count list

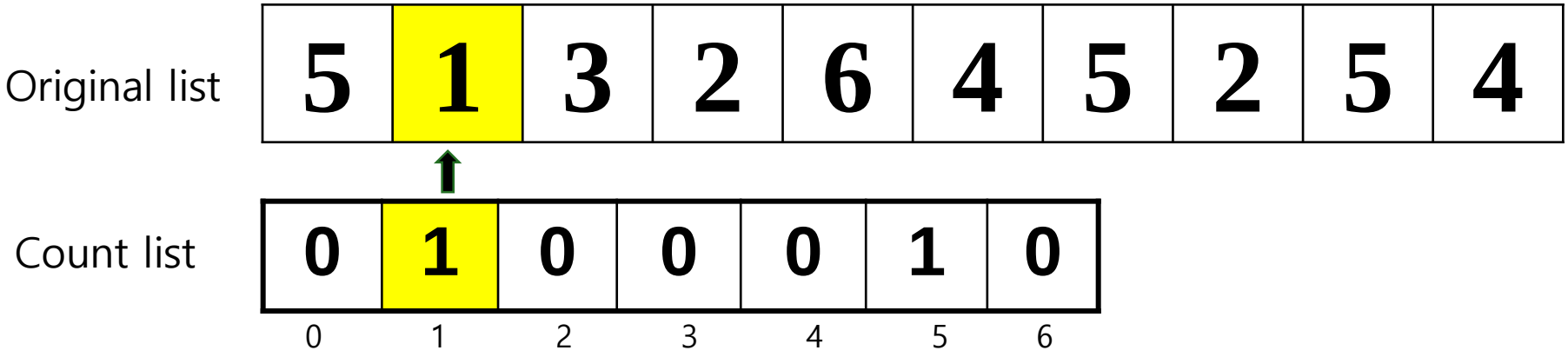
0	0	0	0	0	0	0
0	1	2	3	4	5	6

K(=6, 이경우)만큼 추가적인 공간

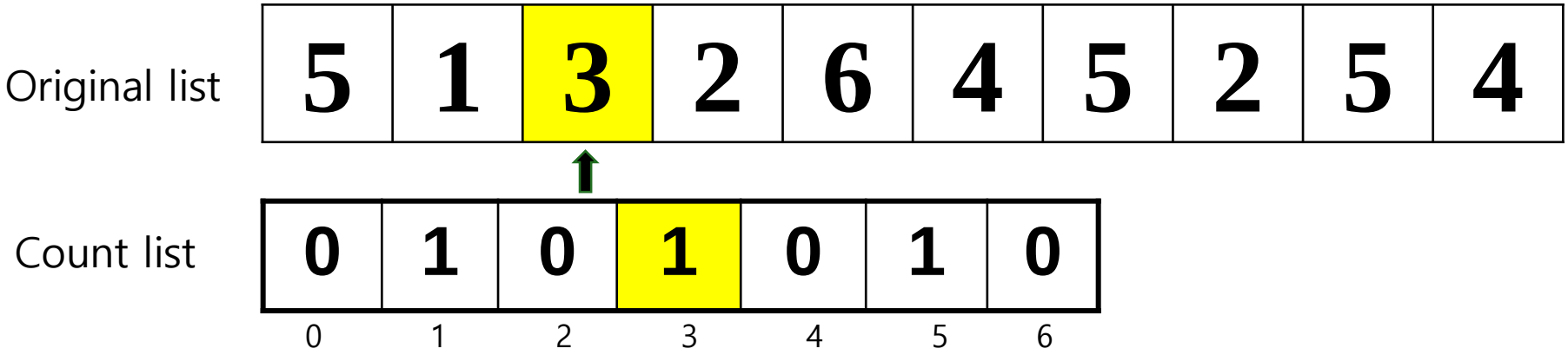
Counting Sorting



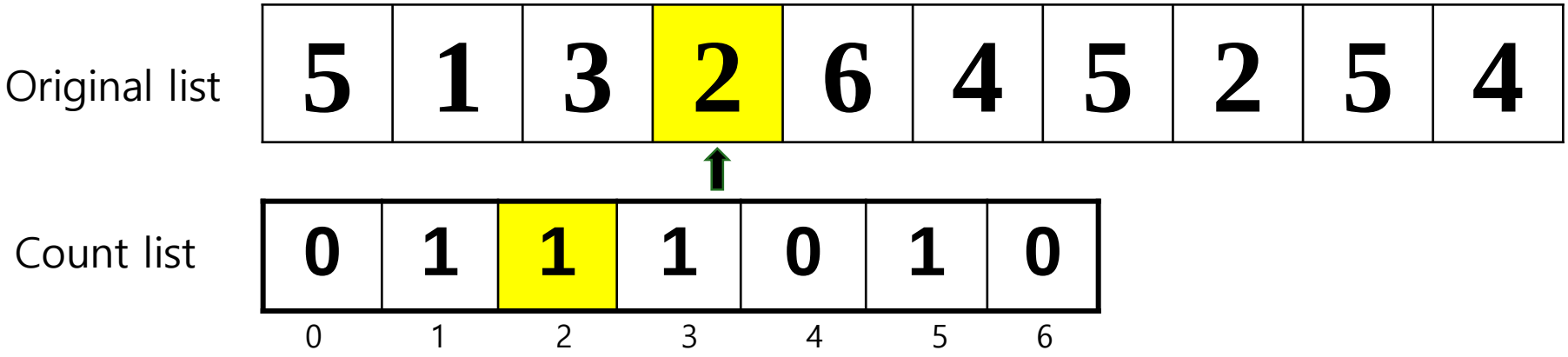
Counting Sorting



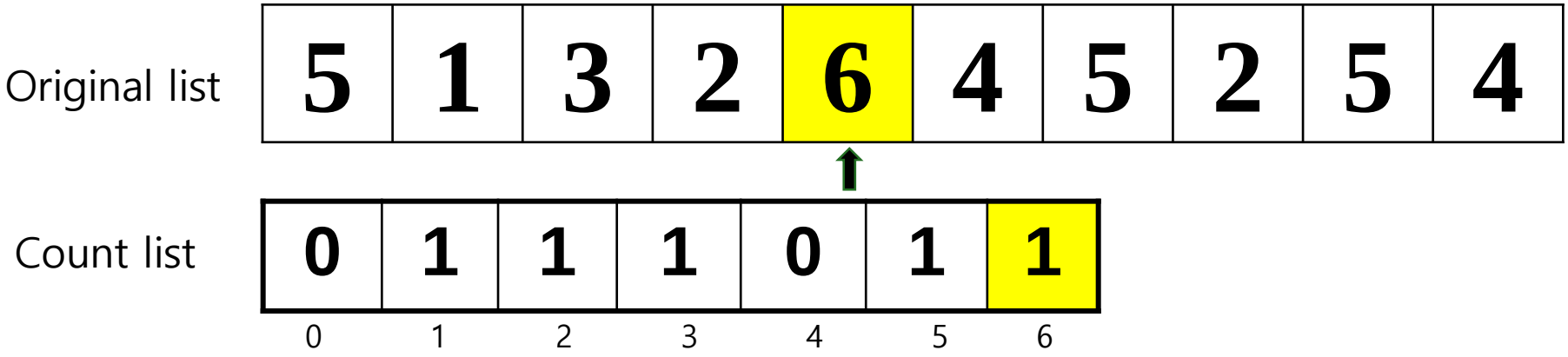
Counting Sorting



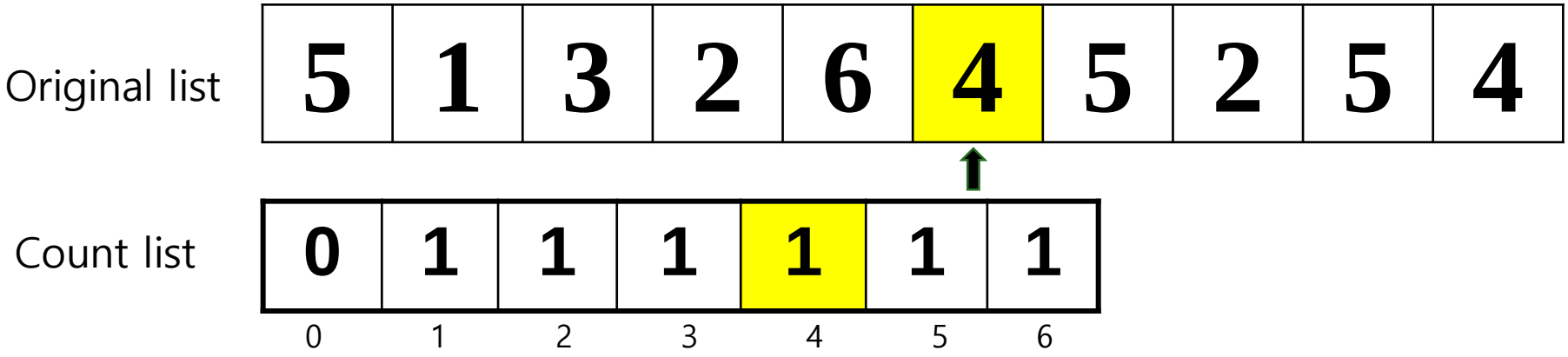
Counting Sorting



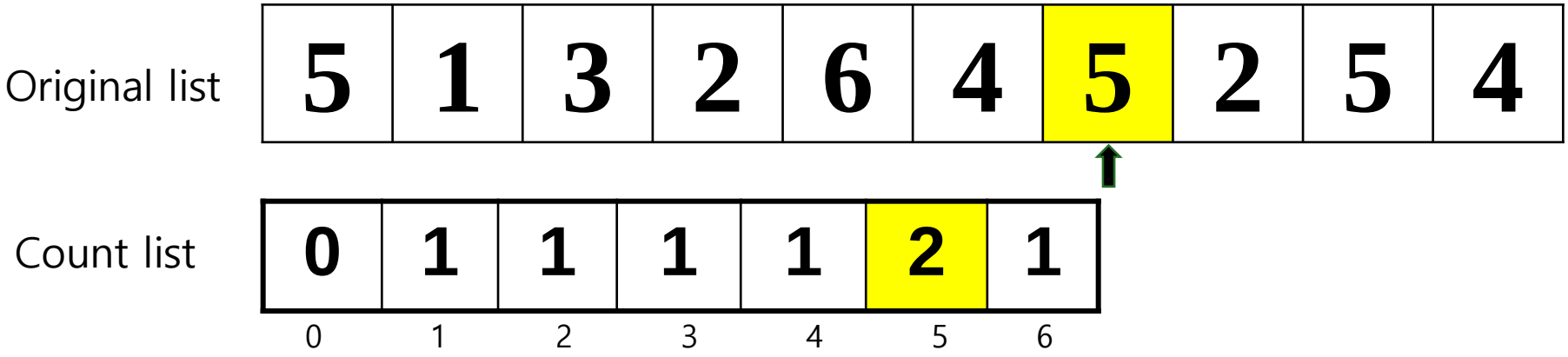
Counting Sorting



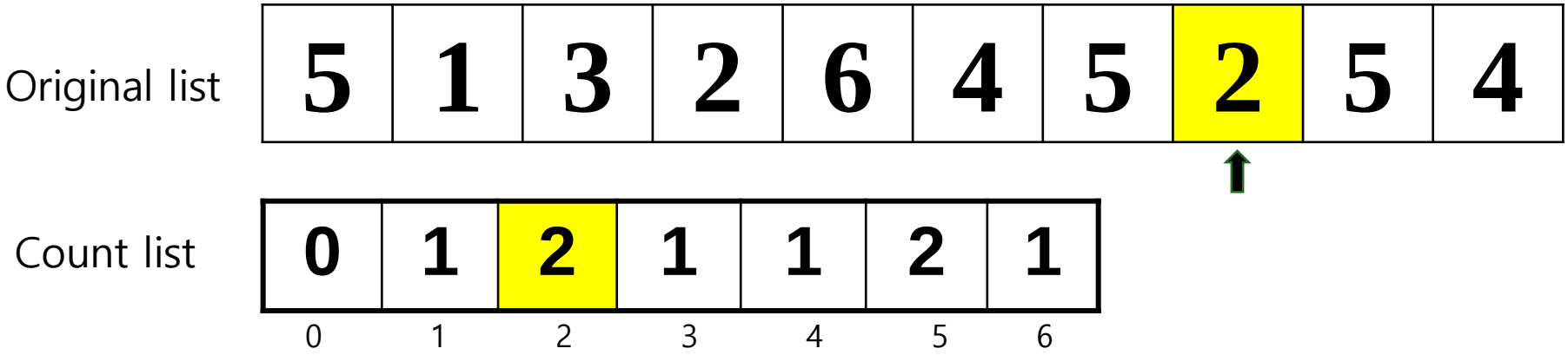
Counting Sorting



Counting Sorting



Counting Sorting



Counting Sorting

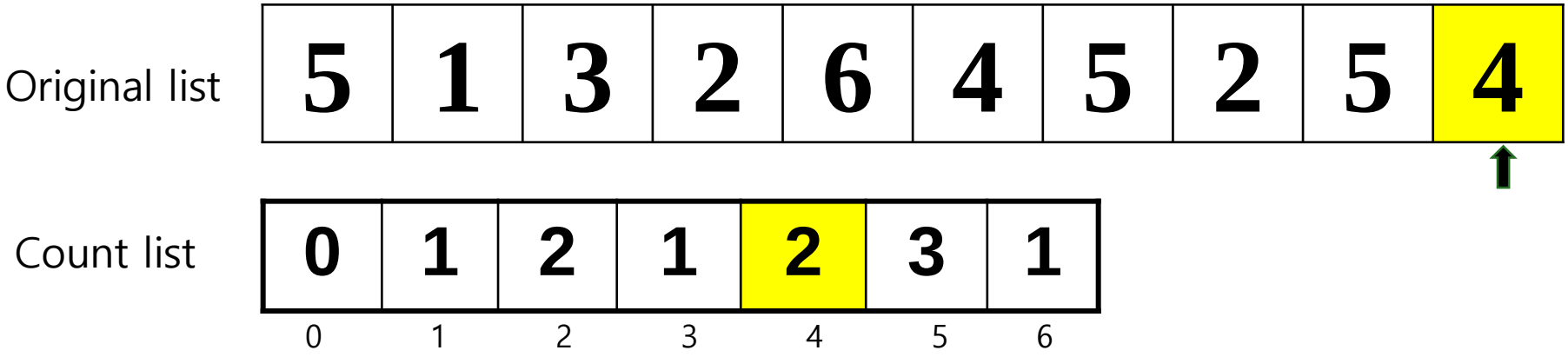
Original list

5	1	3	2	6	4	5	2	5	4

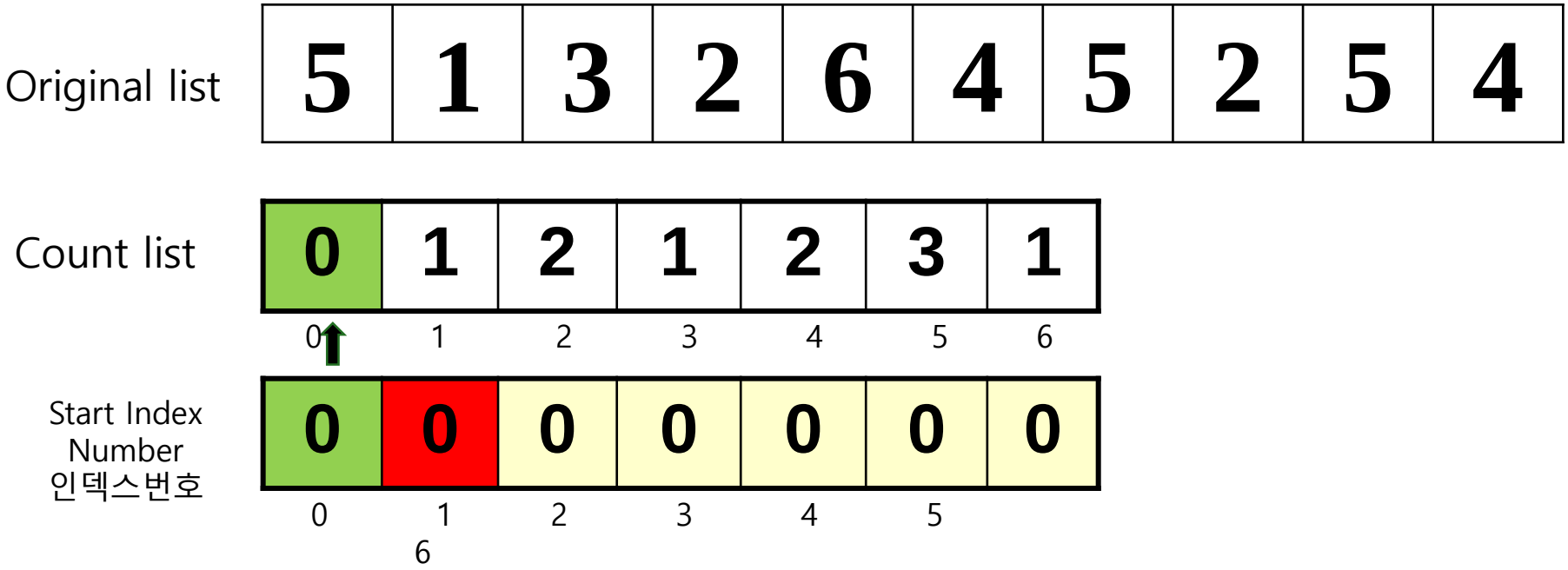
Count list

0	1	2	1	1	3	1
0	1	2	3	4	5	6

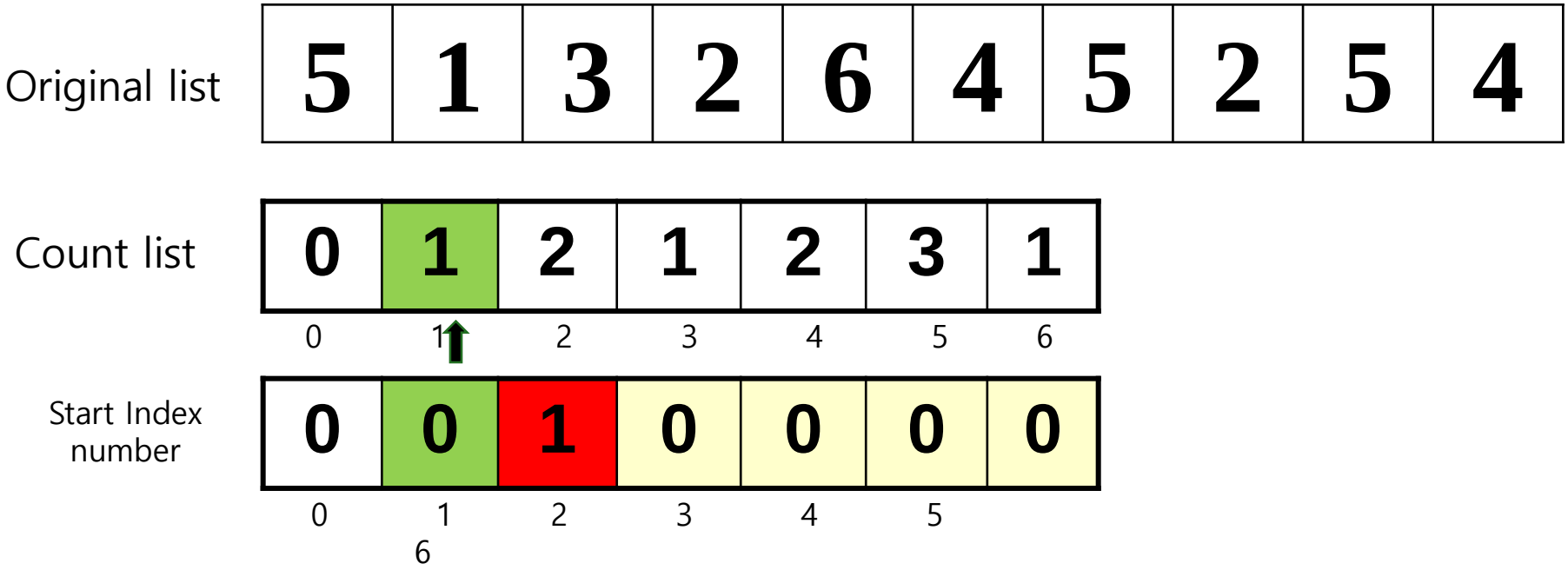
Counting Sorting



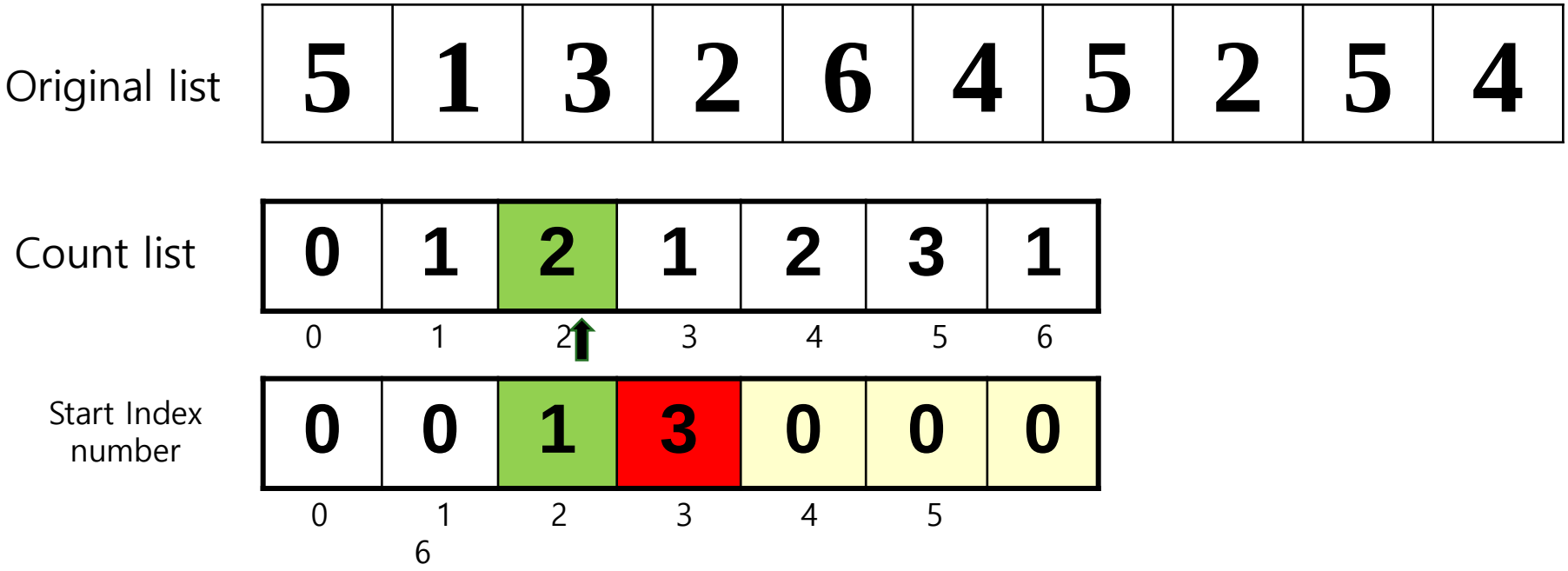
Counting Sorting



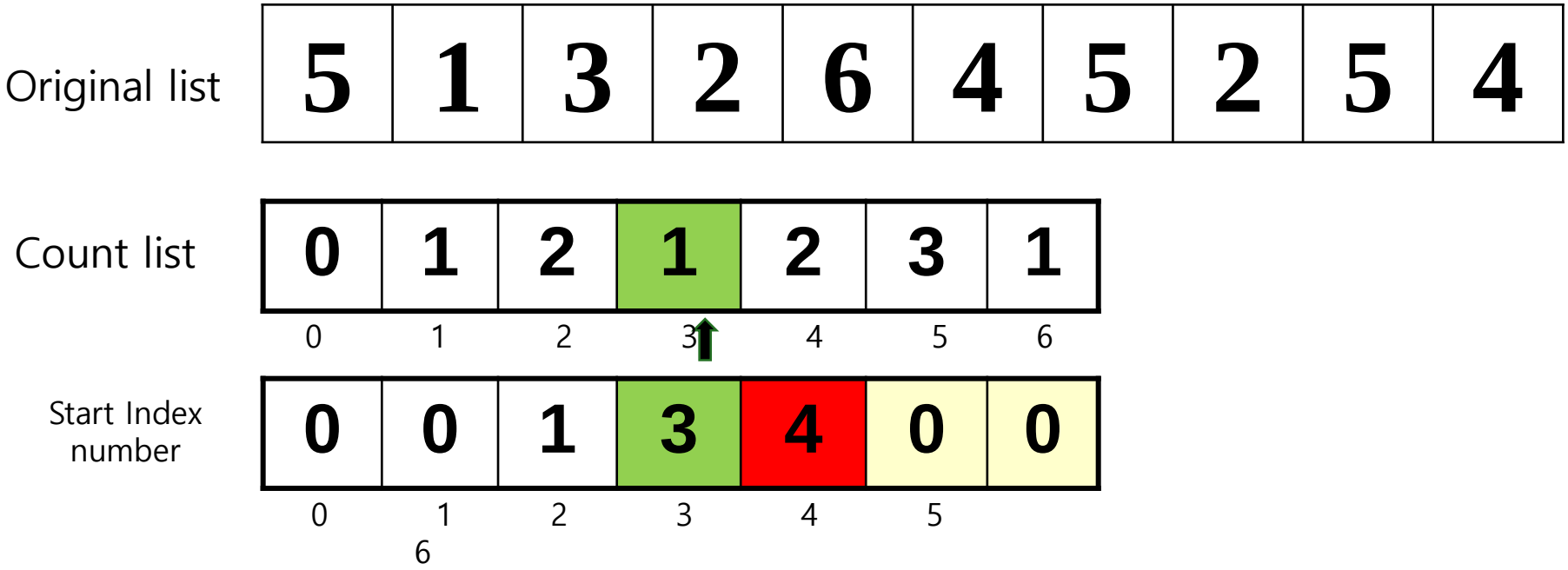
Counting Sorting



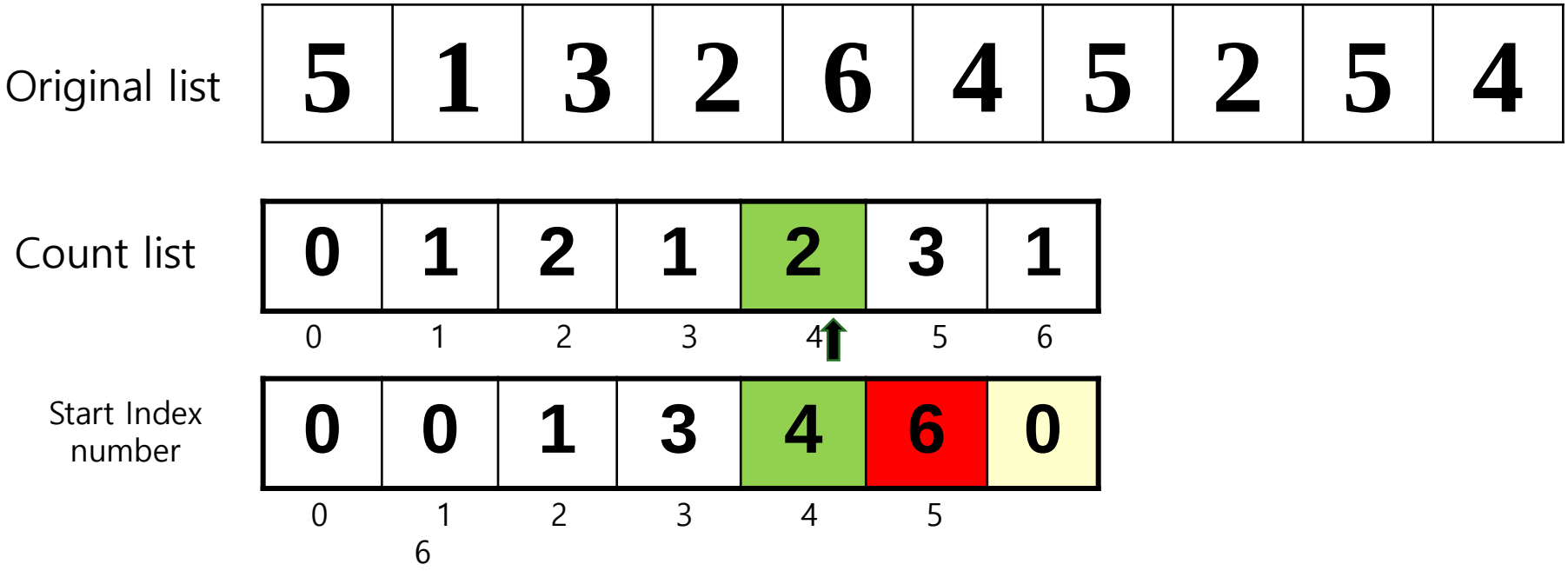
Counting Sorting



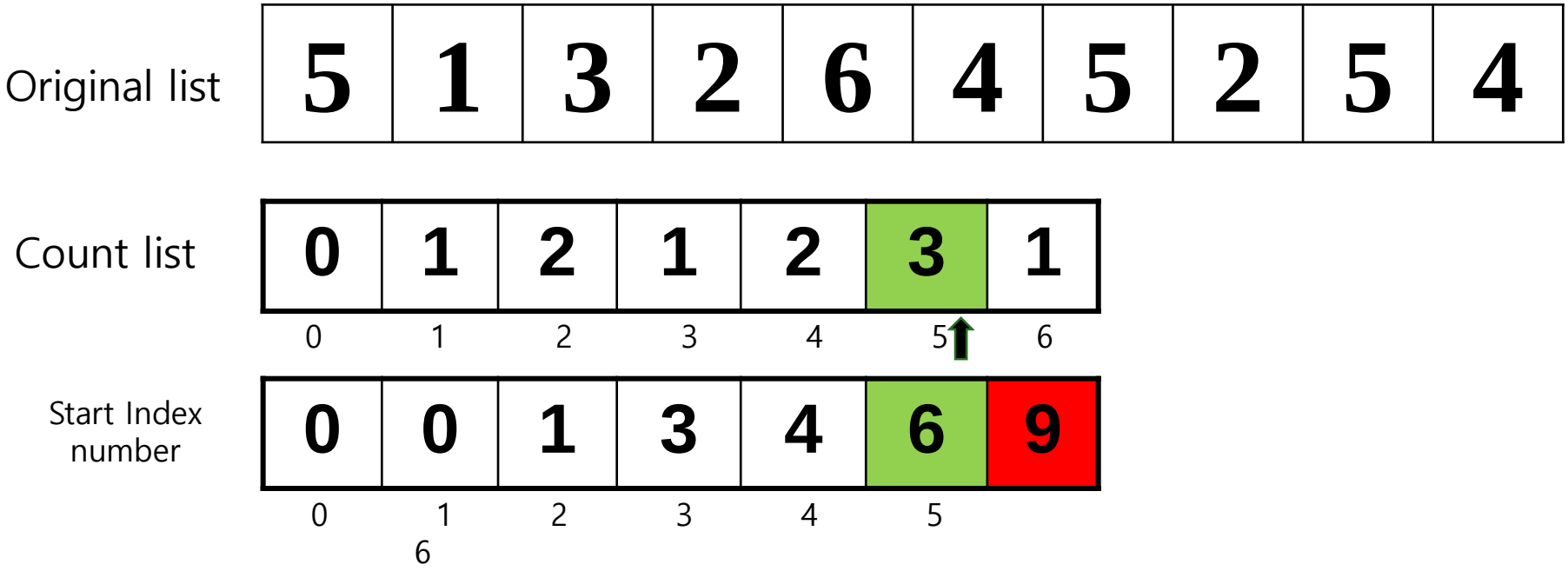
Counting Sorting



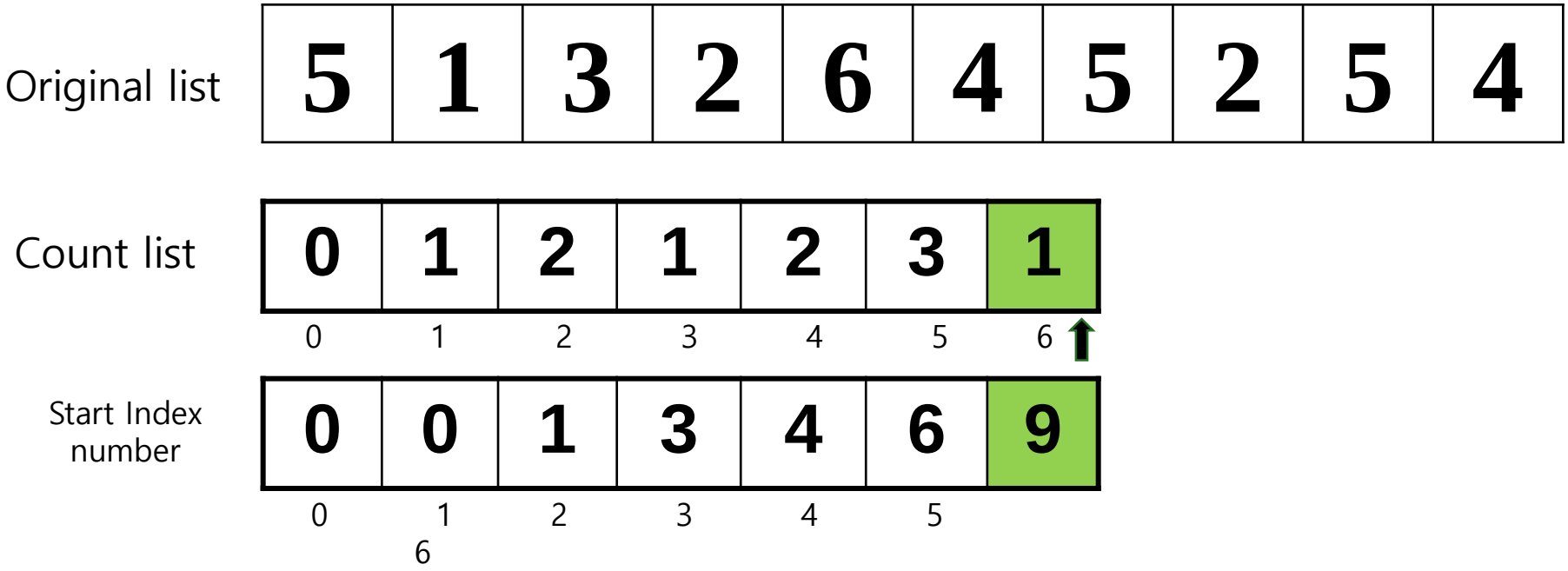
Counting Sorting



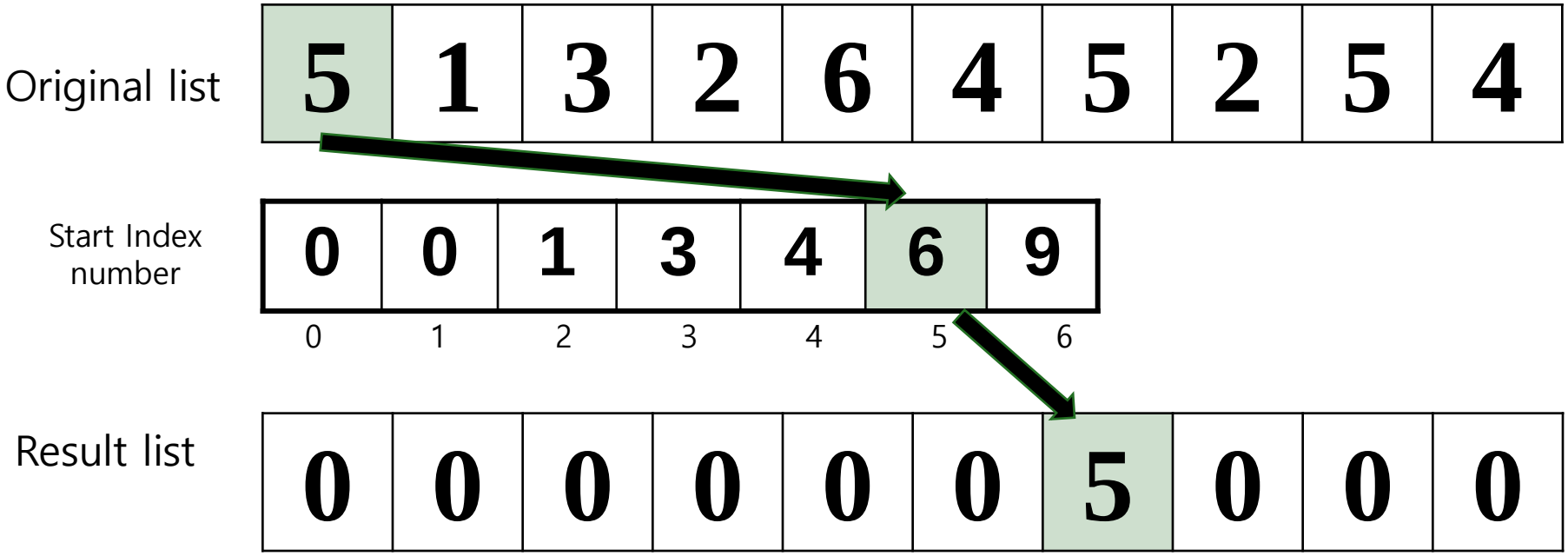
Counting Sorting



Counting Sorting



Counting Sorting



Counting Sorting

Original list

5	1	3	2	6	4	5	2	5	4

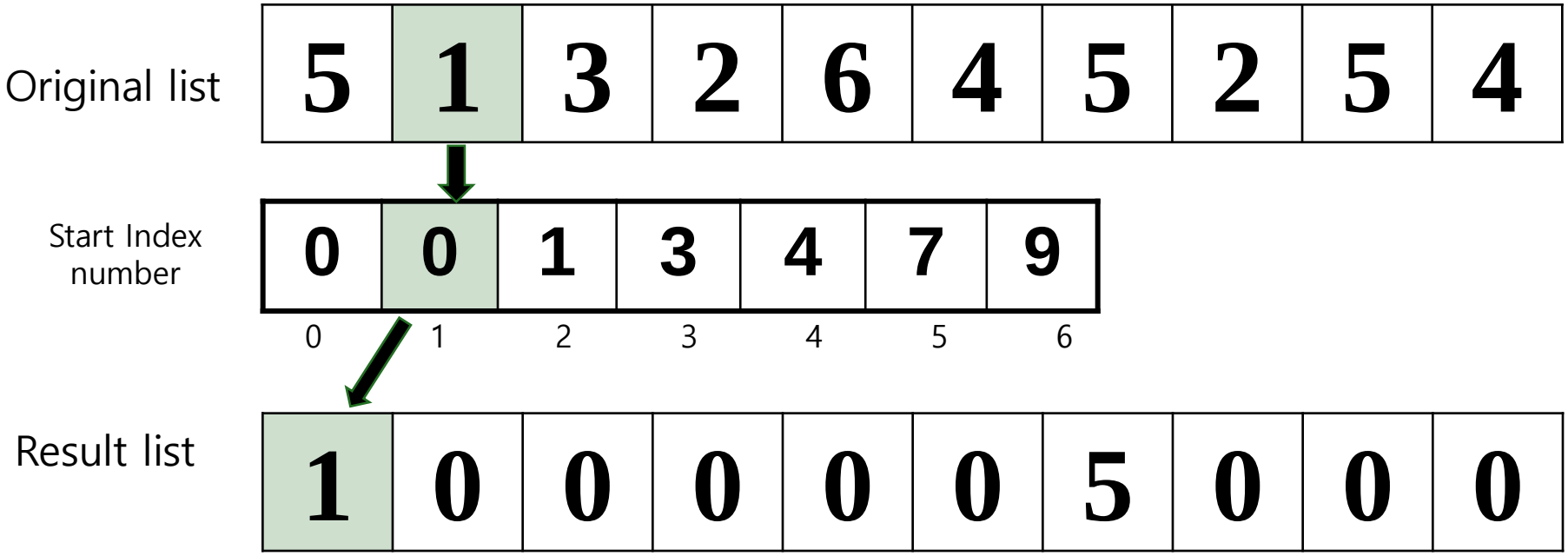
Start Index number

0	0	1	3	4	7	9
0	1	2	3	4	5	6

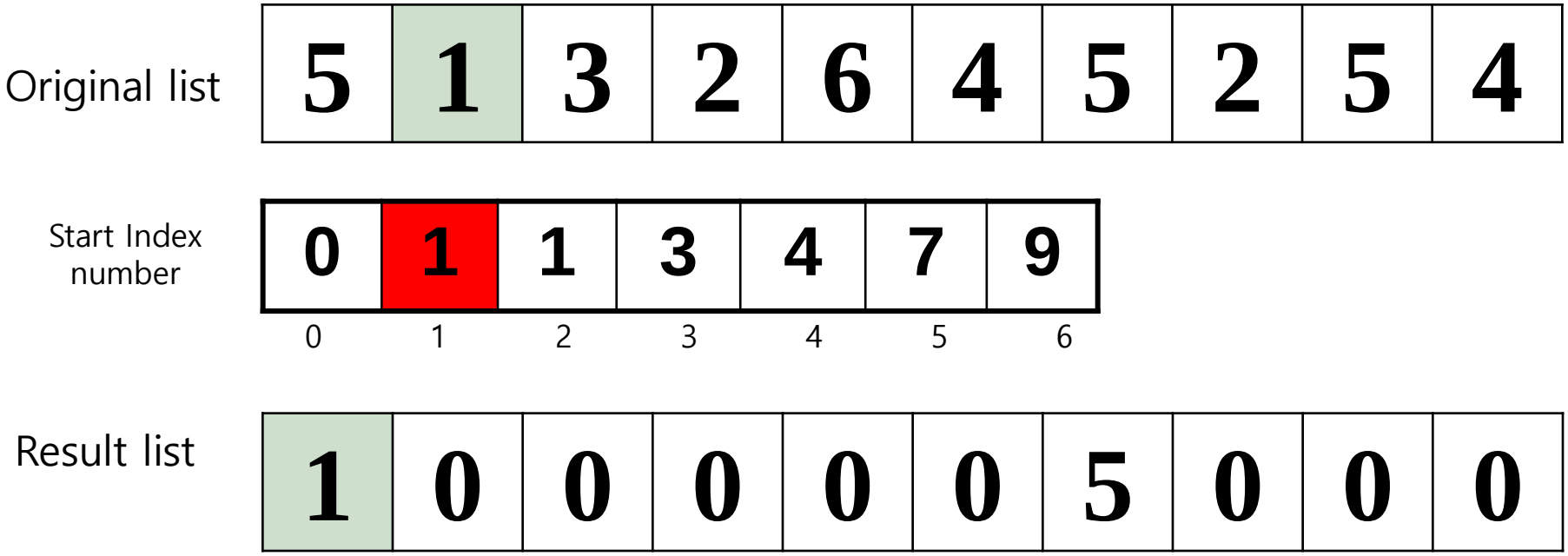
Result list

0	0	0	0	0	0	5	0	0	0

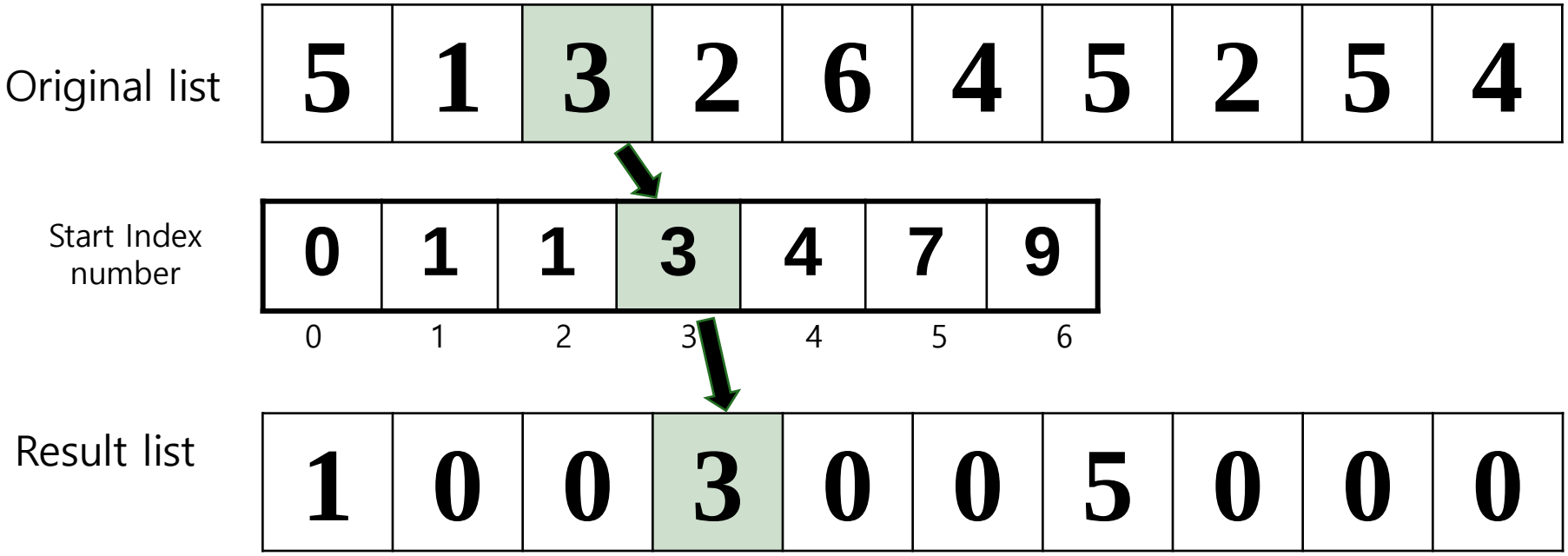
Counting Sorting



Counting Sorting



Counting Sorting



Counting Sorting

Original list

5	1	3	2	6	4	5	2	5	4

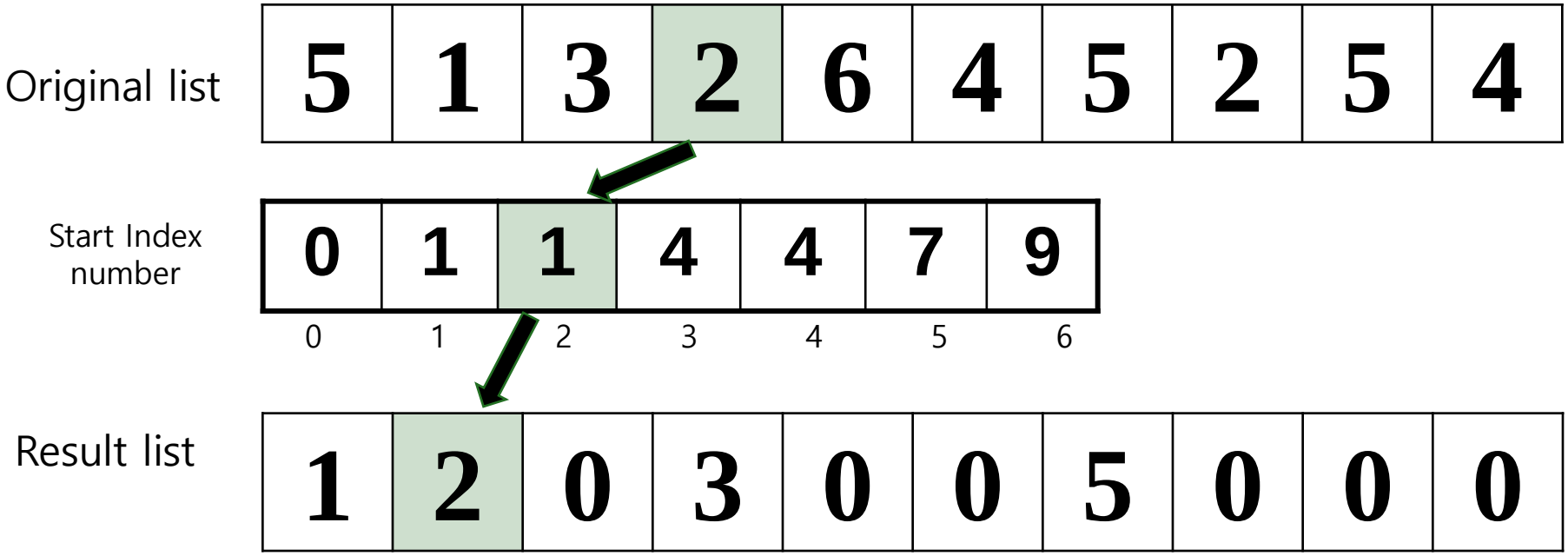
Start Index number

0	1	1	4	4	7	9
0	1	2	3	4	5	6

Result list

1	0	0	3	0	0	5	0	0	0

Counting Sorting



Counting Sorting

Original list

5	1	3	2	6	4	5	2	5	4

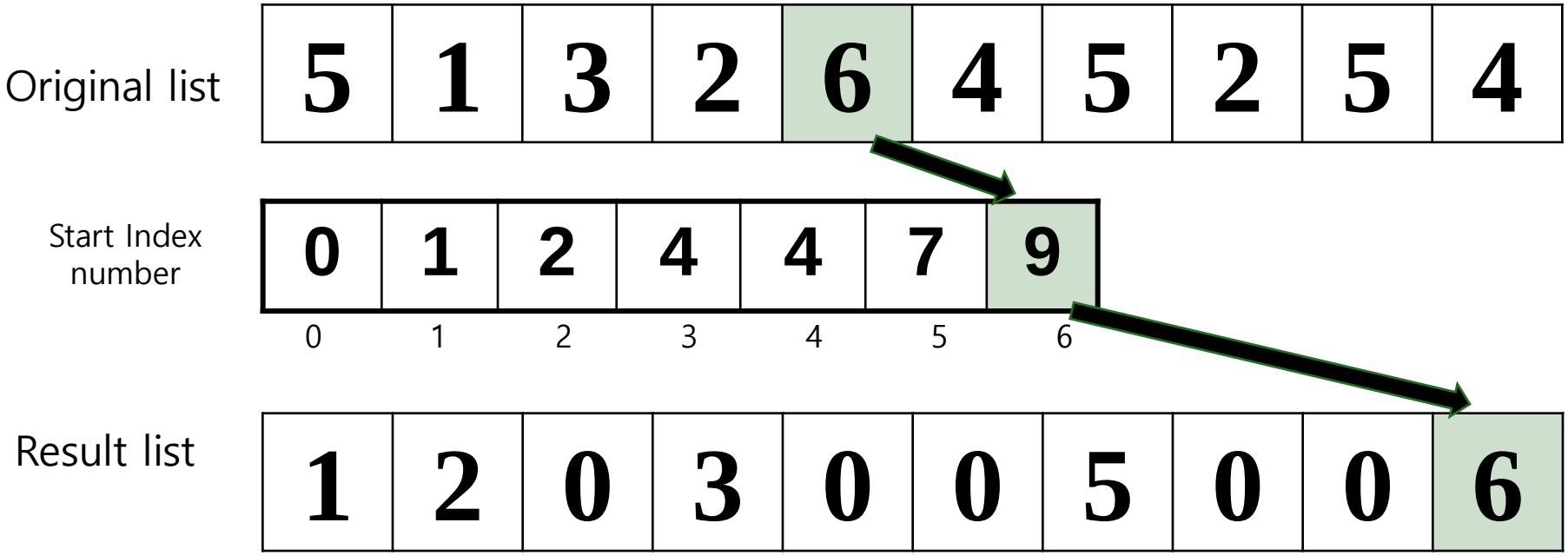
Start Index number

0	1	2	4	4	7	9
0	1	2	3	4	5	6

Result list

1	2	0	3	0	0	5	0	0	0

Counting Sorting



Counting Sorting

Original list

5	1	3	2	6	4	5	2	5	4

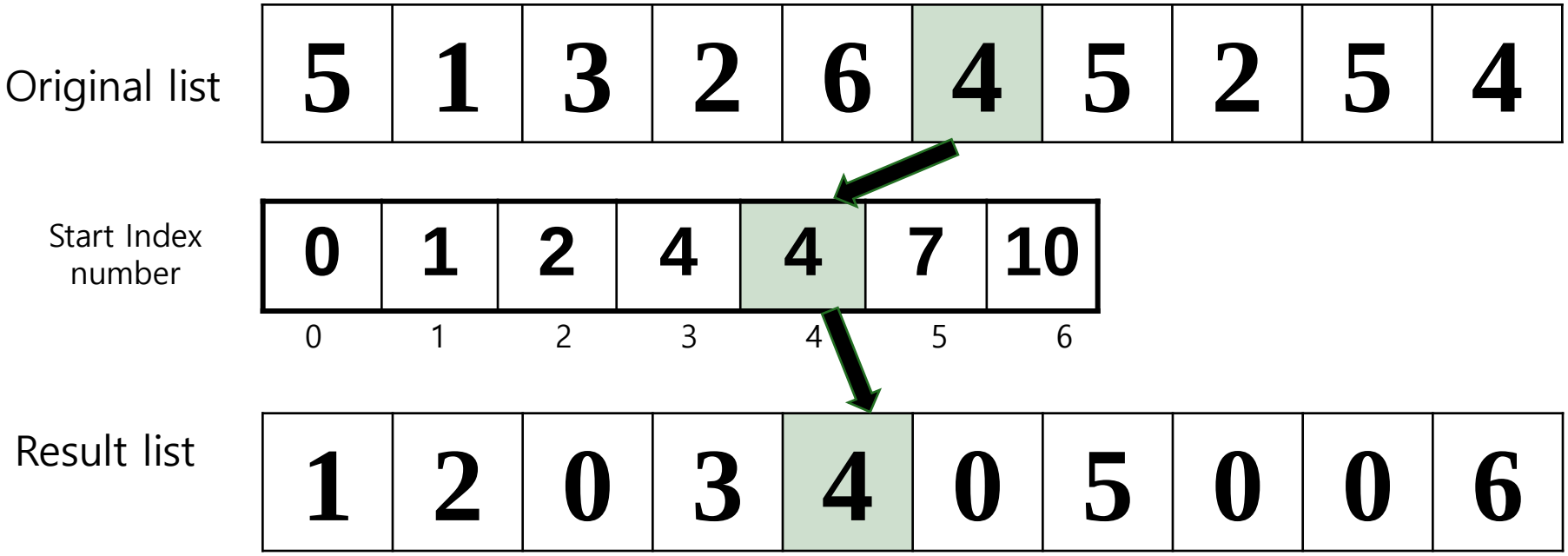
Start Index number

0	1	2	4	4	7	10
0	1	2	3	4	5	6

Result list

1	2	0	3	0	0	5	0	0	6

Counting Sorting



Counting Sorting

Original list

5	1	3	2	6	4	5	2	5	4

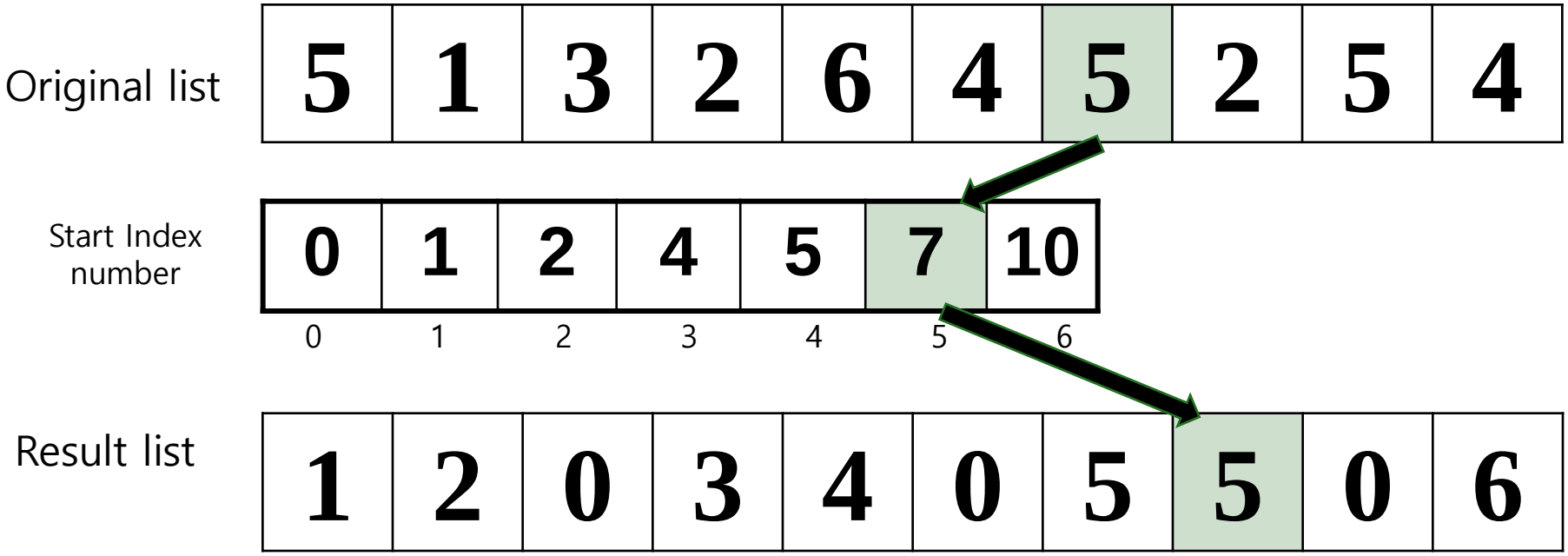
Start Index number

0	1	2	4	5	7	10
0	1	2	3	4	5	6

Result list

1	2	0	3	4	0	5	0	0	6

Counting Sorting



Counting Sorting

Original list

5	1	3	2	6	4	5	2	5	4

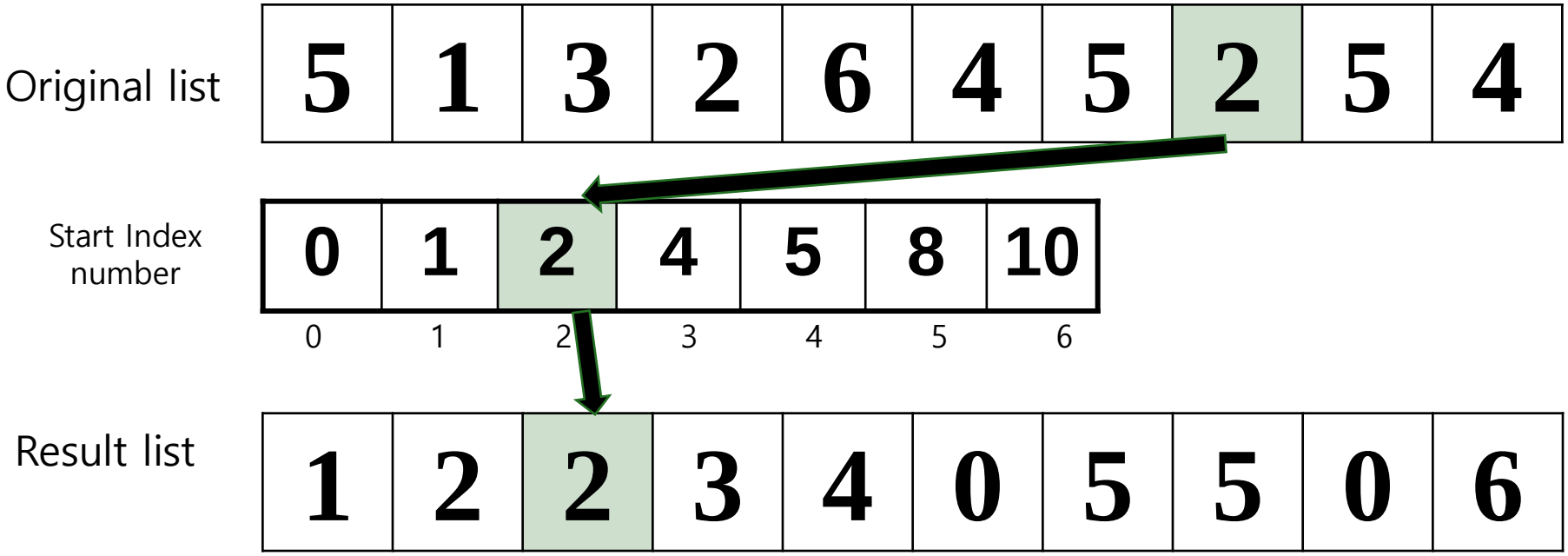
Start Index number

0	1	2	4	5	8	10
0	1	2	3	4	5	6

Result list

1	2	0	3	4	0	5	5	0	6

Counting Sorting



Counting Sorting

Original list

5	1	3	2	6	4	5	2	5	4

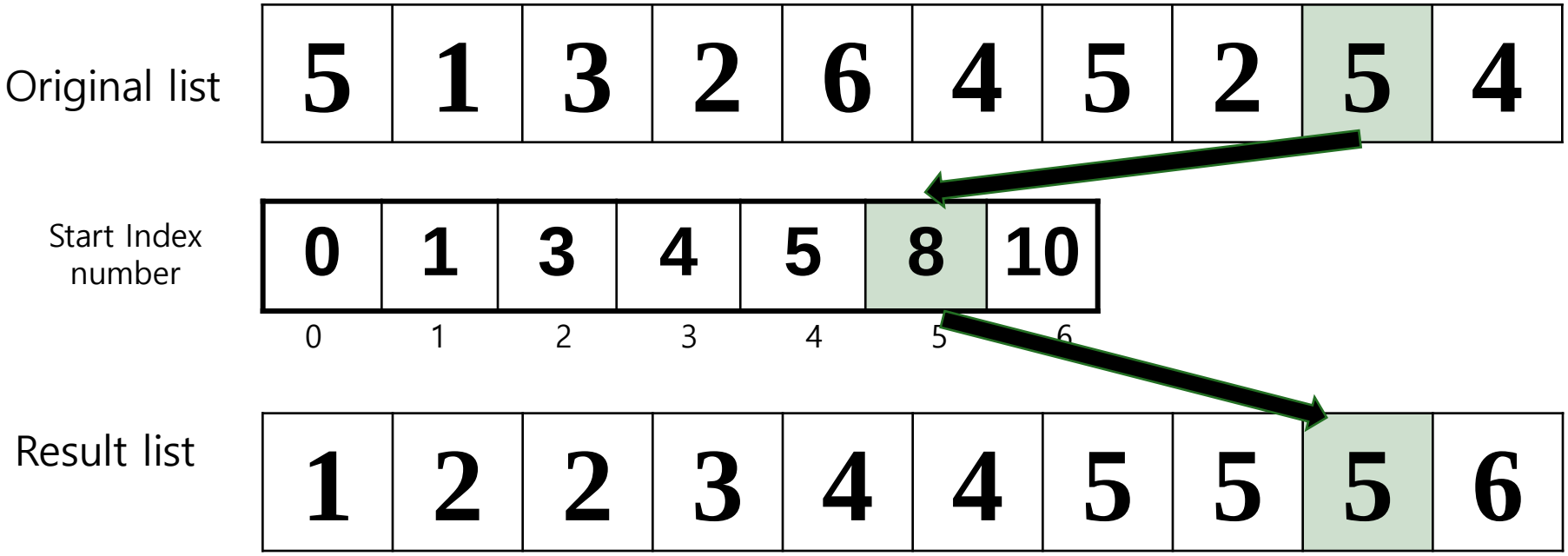
Start Index number

0	1	3	4	5	8	10
0	1	2	3	4	5	6

Result list

1	2	2	3	4	0	5	5	0	6

Counting Sorting



Counting Sorting

Original list

5	1	3	2	6	4	5	2	5	4

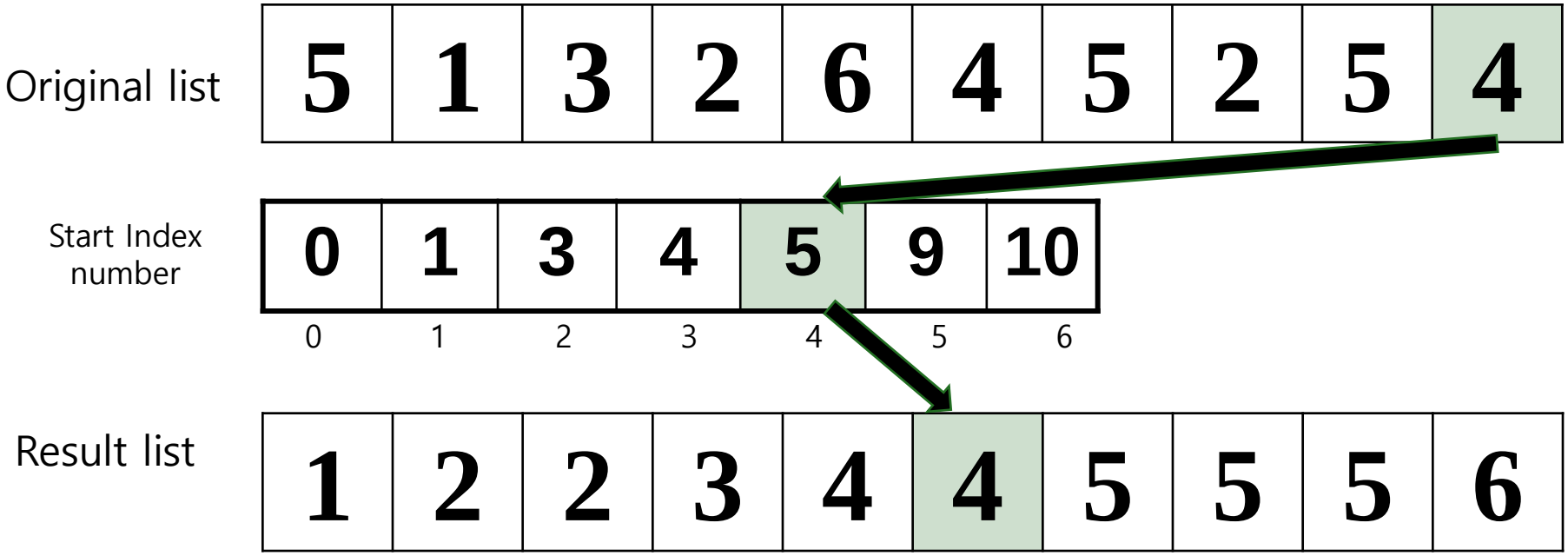
Start Index number

0	1	3	4	5	9	10
0	1	2	3	4	5	6

Result list

1	2	2	3	4	4	5	5	5	6

Counting Sorting



Counting Sorting

Original list

5	1	3	2	6	4	5	2	5	4

Start Index number

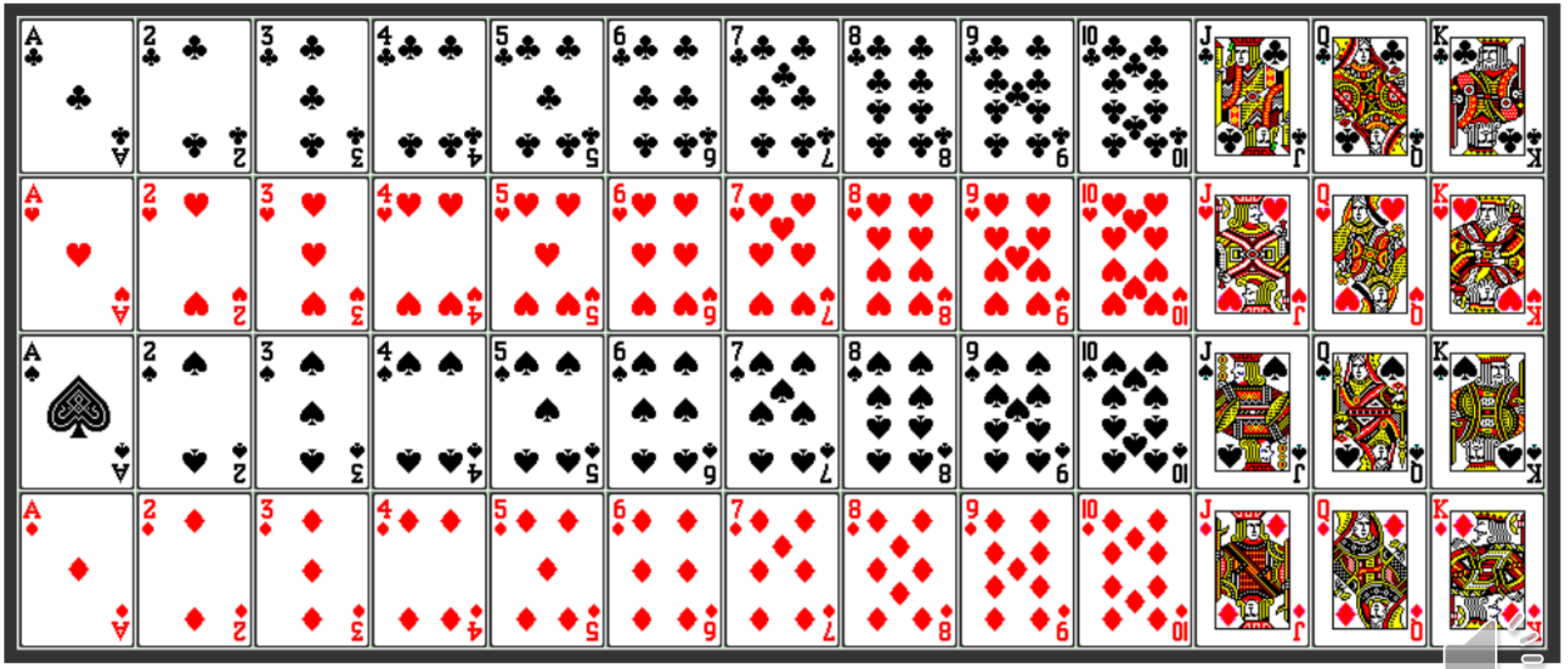
0	1	3	4	6	9	10
0	1	2	3	4	5	6

Result list

1	2	2	3	4	4	5	5	5	6

E N D

Radix sort(기수정렬)
레이딕스, 근원,뿌리,어근



Radix Sort Example

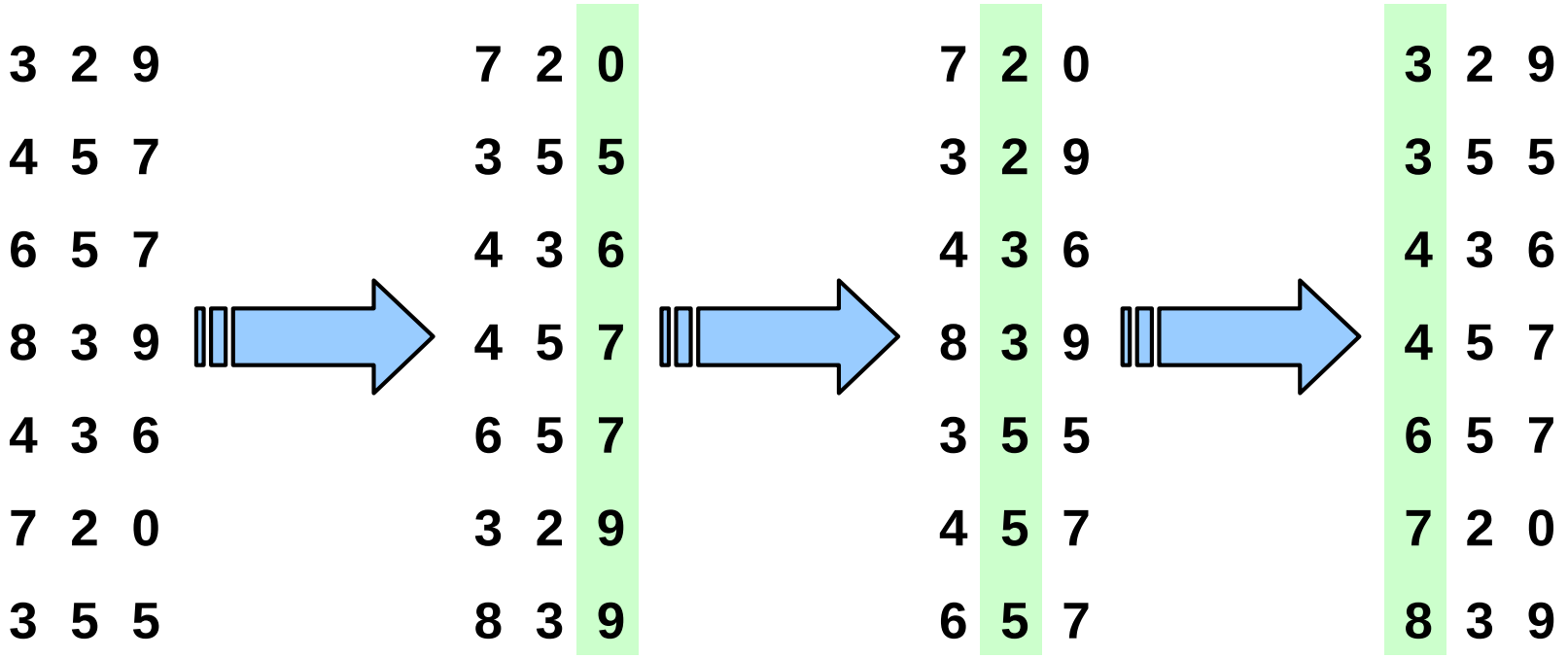
1. Original list
   - 623, 192, 144, 253, 152, 752, 552, 231
2. Sort on 3rd digit (counting sort from 0-9)
   - 231, 192, 152, 752, 552, 623, 253, 144
3. Sort on 2nd digit (counting sort from 0-9)
   - 623, 231, 144, 152, 752, 552, 253, 192
4. Sort on 1st digit (counting sort from 0-9)
   - 144, 152, 192, 231, 253, 552, 623, 752

Compare with: counting sort from 192-752

Running Time of Radix Sort

- use counting sort as the invoked stable sort, if the range of digits is not large
- if digit range is 1..k, then each pass takes $\Theta(n+k)$ time
- there are *d* passes, for a total of $\Theta(d(n+k))$
- if $k = O(n)$, time is $\Theta(dn)$
- when *d* is const, we have $\Theta(n)$, linear!
- 원소의 개수에 비례하는 추가적인 공간

Radix Sort (cont)




```
Radix-Sort(A, d) // d=자리수
    for i ← 1 to d
        use a stable sort to sort array A on digit d
```

안정적인 정렬을 자릿수만큼 하는 셈이 된다

안정적인 정렬을 사용하지 않으면 정렬되지 않을 수 있다

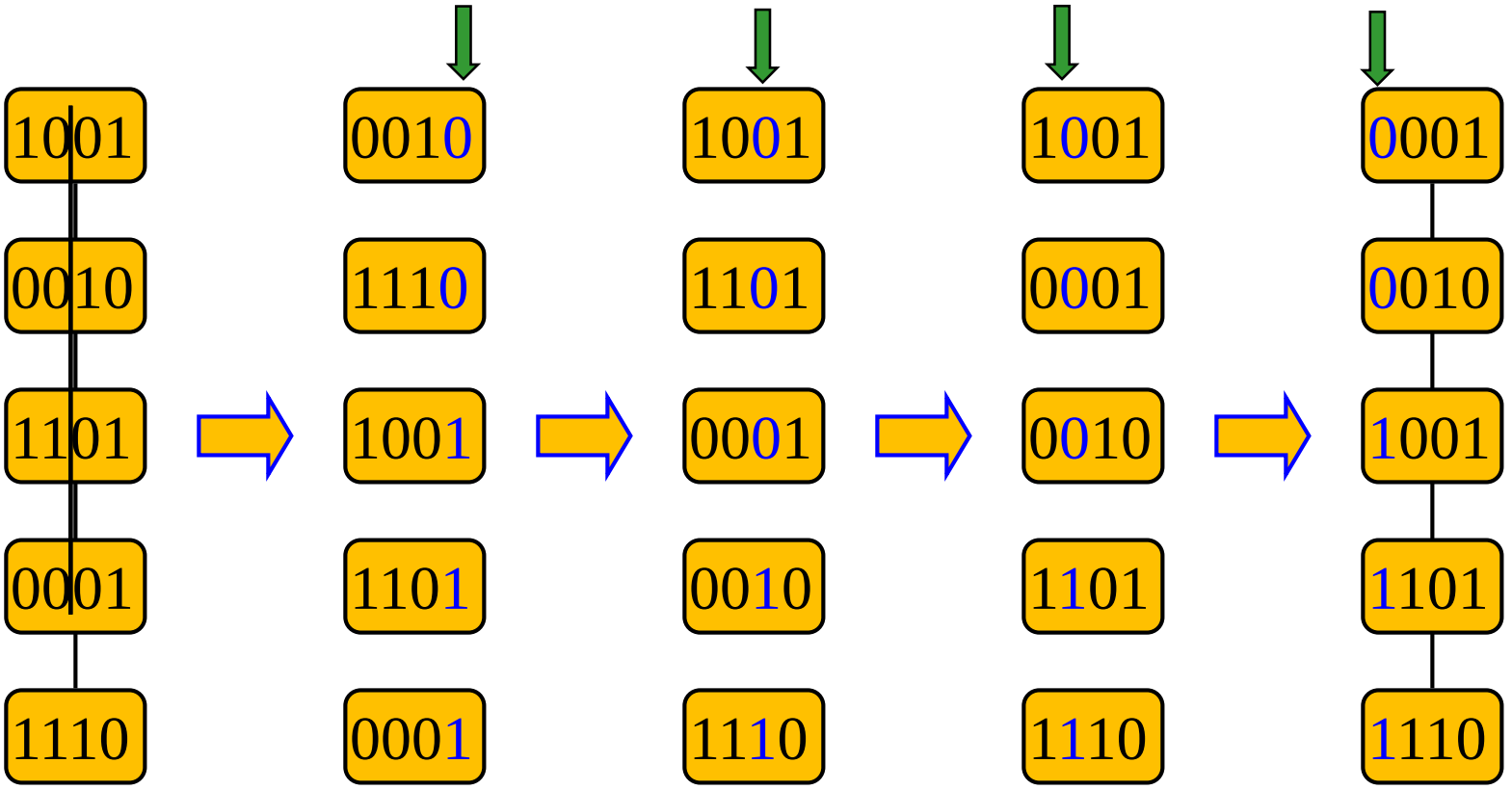
위의 경우 버킷10개인 안정적(버킷)정렬을 자릿수만큼하는셈

Radix Sort Example

A						
492		031		102		031
299		492		204		102
102		102		031		204
031	→	204	→	835	→	299
996		835		492		492
204		996		996		835
835		299		299		996

Example

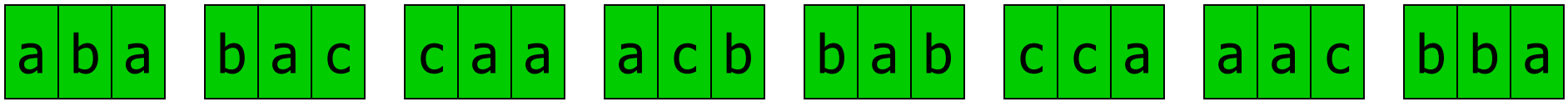
- Sorting a sequence of 4-bit integers



추가적인 공간필요

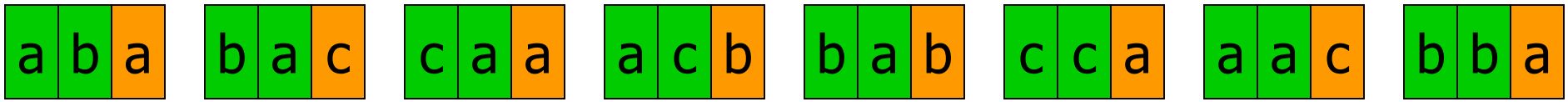
Radix Sort Example

Input data:



Radix Sort Example

Pass 1: Looking at rightmost position.

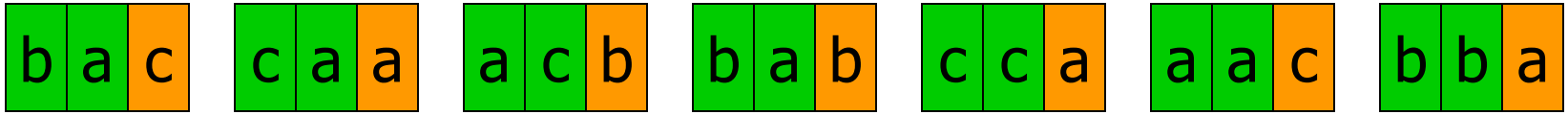


Place into appropriate pile.

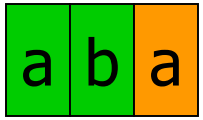
a b c

Radix Sort Example

Pass 1: Looking at rightmost position.



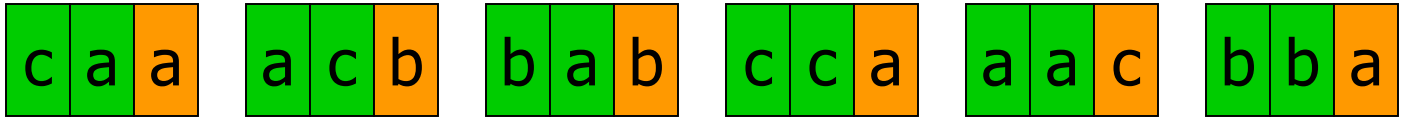
Place into appropriate pile.



a b c

Radix Sort Example

Pass 1: Looking at rightmost position.

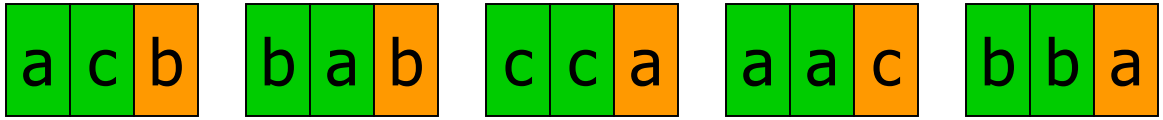


Place into appropriate pile.

a b c

Radix Sort Example

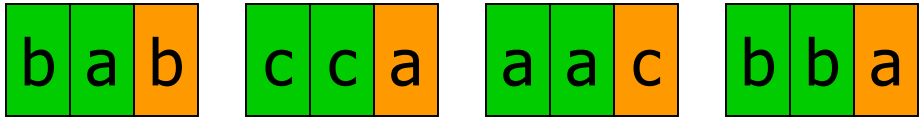
Pass 1: Looking at rightmost position.



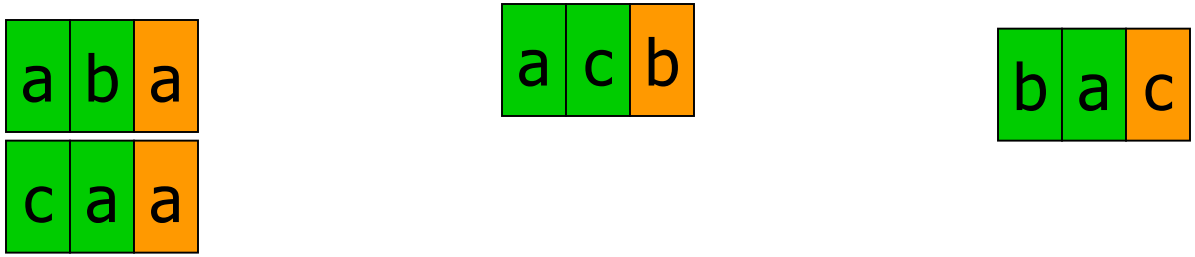
Place into appropriate pile.

Radix Sort Example

Pass 1: Looking at rightmost position.



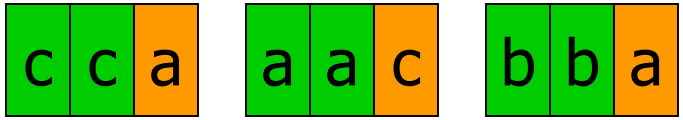
Place into appropriate pile.



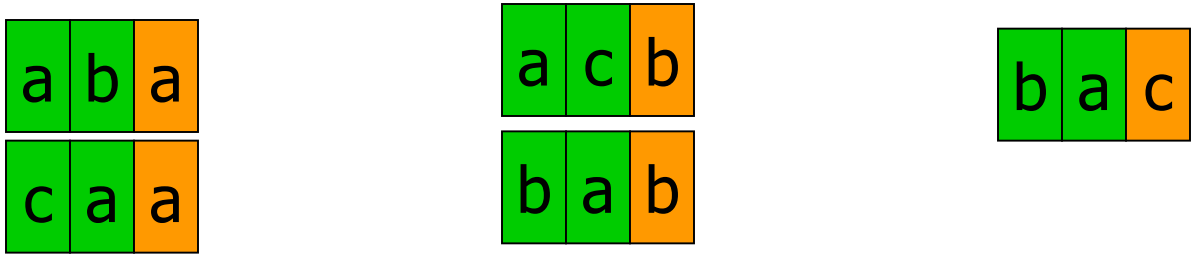
a b c

Radix Sort Example

Pass 1: Looking at rightmost position.



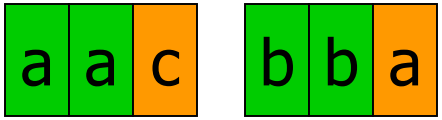
Place into appropriate pile.



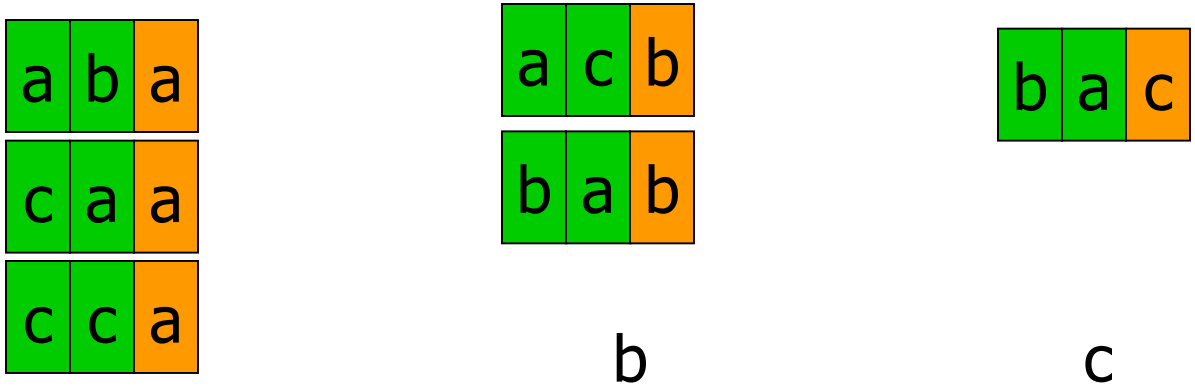
a b c

Radix Sort Example

Pass 1: Looking at rightmost position.

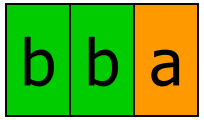


Place into appropriate pile.

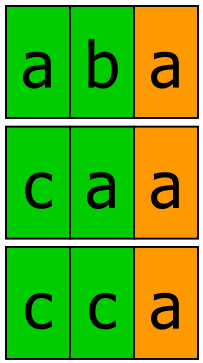


Radix Sort Example

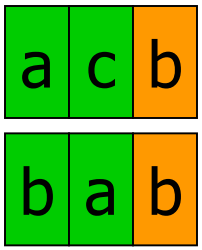
Pass 1: Looking at rightmost position.



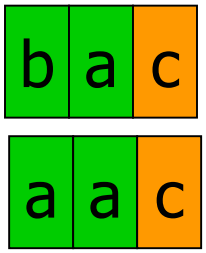
Place into appropriate pile.



a



b

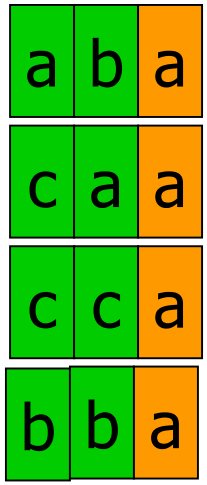


c

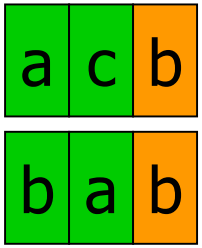
Radix Sort Example

Pass 1: Looking at rightmost position.

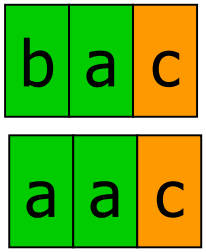
Place into appropriate pile.



a



b

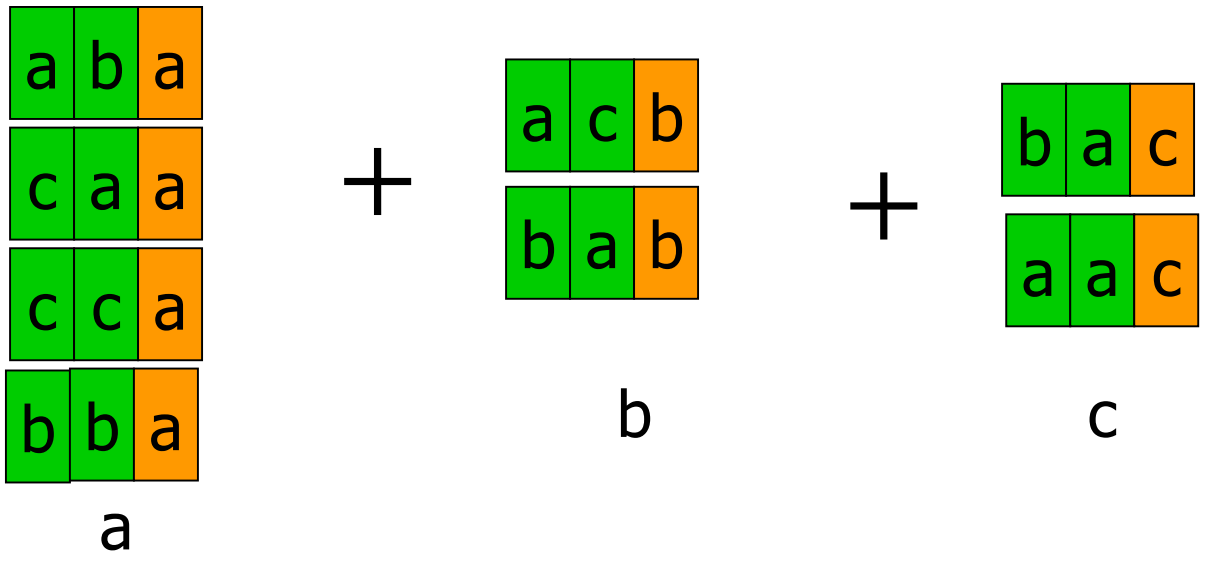


c

Radix Sort Example

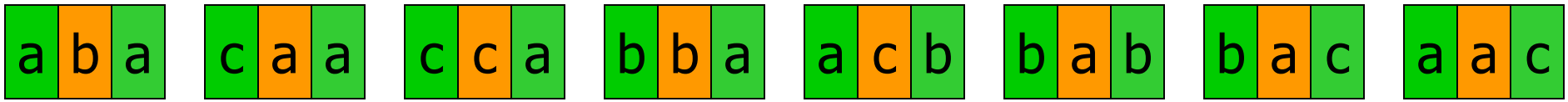
Pass 1: Looking at rightmost position.

Join piles.



Radix Sort Example

Pass 2: Looking at next position.

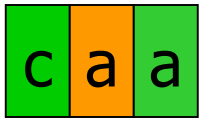


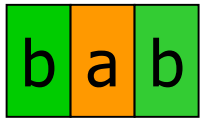
Place into appropriate pile.

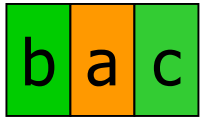
a b c

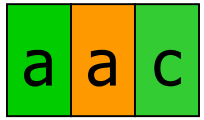
Radix Sort Example

Pass 2: Looking at next position.



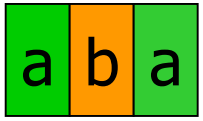


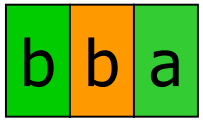




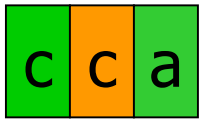
a

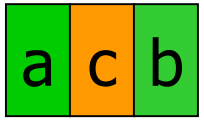
Join piles.





b

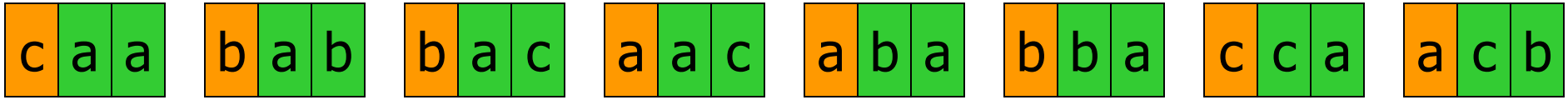




c

Radix Sort Example

Pass 3: Looking at last position.



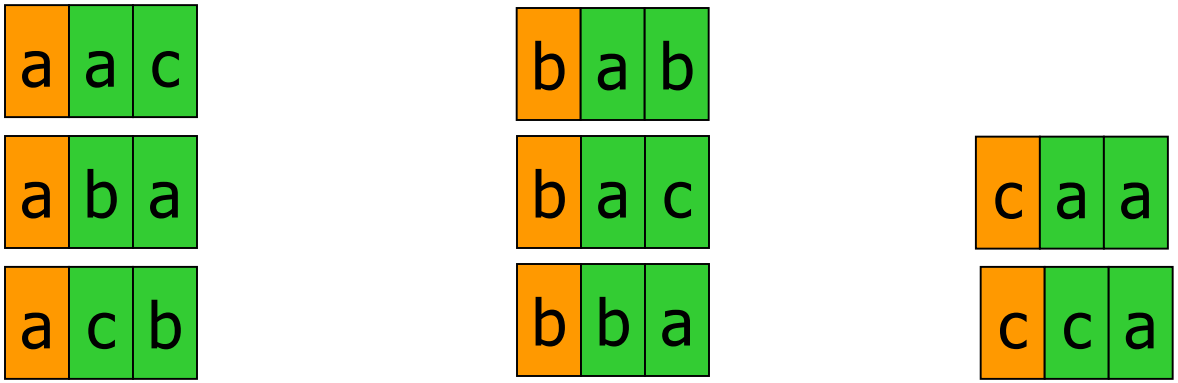
Place into appropriate pile.

a b c

Radix Sort Example

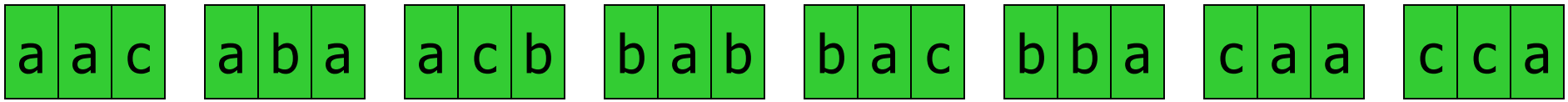
Pass 3: Looking at last position.

Join piles.



Radix Sort Example

Result is sorted.



RadixSorting Strings example

문자열을 기수정렬

	5th pass	4th pass	3rd pass	2nd pass	1st pass
String 1	z	i	p	p	y
String 2	z	a	p		
String 3	a	n	t	s	
String 4	f	l	a	p	s

NULLs are just like fake characters

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(not in-place sort)

- 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

```
import java.util.*;
public class my_radix_sorting {
  static int get_max_val(int my_arr[], int arr_len) {
    int max_val = my_arr[0];
    for (int i = 1; i < arr_len; i++)
      if (my_arr[i] > max_val)
      max_val = my_arr[i];
    return max_val;
  }
  static void countSort(int my_arr[], int arr_len, int exp) {
    int result[] = new int[arr_len];
    int i;
    int count[] = new int[10];
    Arrays.fill(count,0);
    for (i = 0; i < arr_len; i++)
      count[ (my_arr[i]/exp)%10 ]++;
    for (i = 1; i < 10; i++)
      count[i] += count[i - 1];
    for (i = arr_len - 1; i >= 0; i--) {
      result[count[ (my_arr[i]/exp)%10 ] - 1] = my_arr[i];
      count[ (my_arr[i]/exp)%10 ]--;
    }
    for (i = 0; i < arr_len; i++)
      my_arr[i] = result[i];
  }
  static void radix_sort(int my_arr[], int arr_len) {
      int m = get_max_val(my_arr, arr_len);
      for (int exp = 1; m/exp > 0; exp *= 10)
      countSort(my_arr, arr_len, exp);
    }
   public static void main (String[] args) {
      int my_arr[] = {56, 78, 102, 345, 67, 90, 102, 45, 78};
      int arr_len = my_arr.length;
      System.out.println("The array after performing radix sort is ");
      radix_sort(my_arr, arr_len);
      for (int i=0; i<arr_len; i++)
      System.out.print(my_arr[i]+" ");
    }
}
```

Shell sort(계수정렬)

Shellsort(쉘 정렬)

- Simple *extension* of insertion sort.
- Created by Donald Shell in 1959
  - the first sub quadratic sorting algorithm
  - Shell sort is a good choice for moderate amounts of data
  - Start with sub arrays created by looking at data that is far apart and then reduce the gap size
- Gains speed by allowing exchanges with elements that are far apart (thereby fixing multiple inversions).
- *K-sort* the file
  - Divide the file into *k* subsequences.
  - Each subsequence consists of keys that are *k* locations apart in the input file.
  - Sort each *k*-sequence using insertion sort.
  - Will result in *k* *k*-sorted files. Taking every *k*-th key from anywhere results in a sorted sequence.
- *k*-sort the file for decreasing values of increment *k*, with *k*=1 in the last iteration.
- *k*-sorting for large values of *k* in earlier iterations, *reduces* the number of comparisons for smaller values of *h* in later iterations.
- Correctness follows from the fact that the last step is plain insertion sort.

Shellsort

- A *family* of algorithms, characterized by the sequence $\{h_k\}$ of increments that are used in sorting.
- By interleaving, we can fix multiple inversions with each comparison, so that later passes see files that are "nearly sorted." This implies that either there are many keys not too far from their final position, or only a small number of keys are far off.

Shell Sort

Pass	List (K=5)										Notes
1	77	62	14	9	30	21	80	25	70	55	Swap
	21	62	14	9	30	77	80	25	70	55	In order
	21	62	14	9	30	77	80	25	70	55	In order
	21	62	14	9	30	77	80	25	70	55	In order
	21	62	14	9	30	77	80	25	70	55	In order

Just as in the straight insertion sort, we compare 2 values and swap them if they are out of order. However, in the shell sort, we compare values that are a distance *K* apart.

Once we have completed going through the elements in our list with K=5, we decrease K and continue the process.

Shell Sort

Pass	List (K=2)										Notes
2	21	62	14	9	30	77	80	25	70	55	Swap
	14	62	21	9	30	77	80	25	70	55	Swap
	14	9	21	62	30	77	80	25	70	55	In order
	14	9	21	62	30	77	80	25	70	55	In order
	14	9	21	62	30	77	80	25	70	55	In order
	14	9	21	62	30	77	80	25	70	55	Swap
	14	9	21	62	30	25	80	77	70	55	Swap
	14	9	21	25	30	62	80	77	70	55	In order
	14	9	21	25	30	62	80	77	70	55	Swap
	14	9	21	25	30	62	70	77	80	55	In order
	14	9	21	25	30	62	70	77	80	55	Swap
	14	9	21	25	30	62	70	55	80	77	Swap
	14	9	21	25	30	55	70	62	80	77	In order

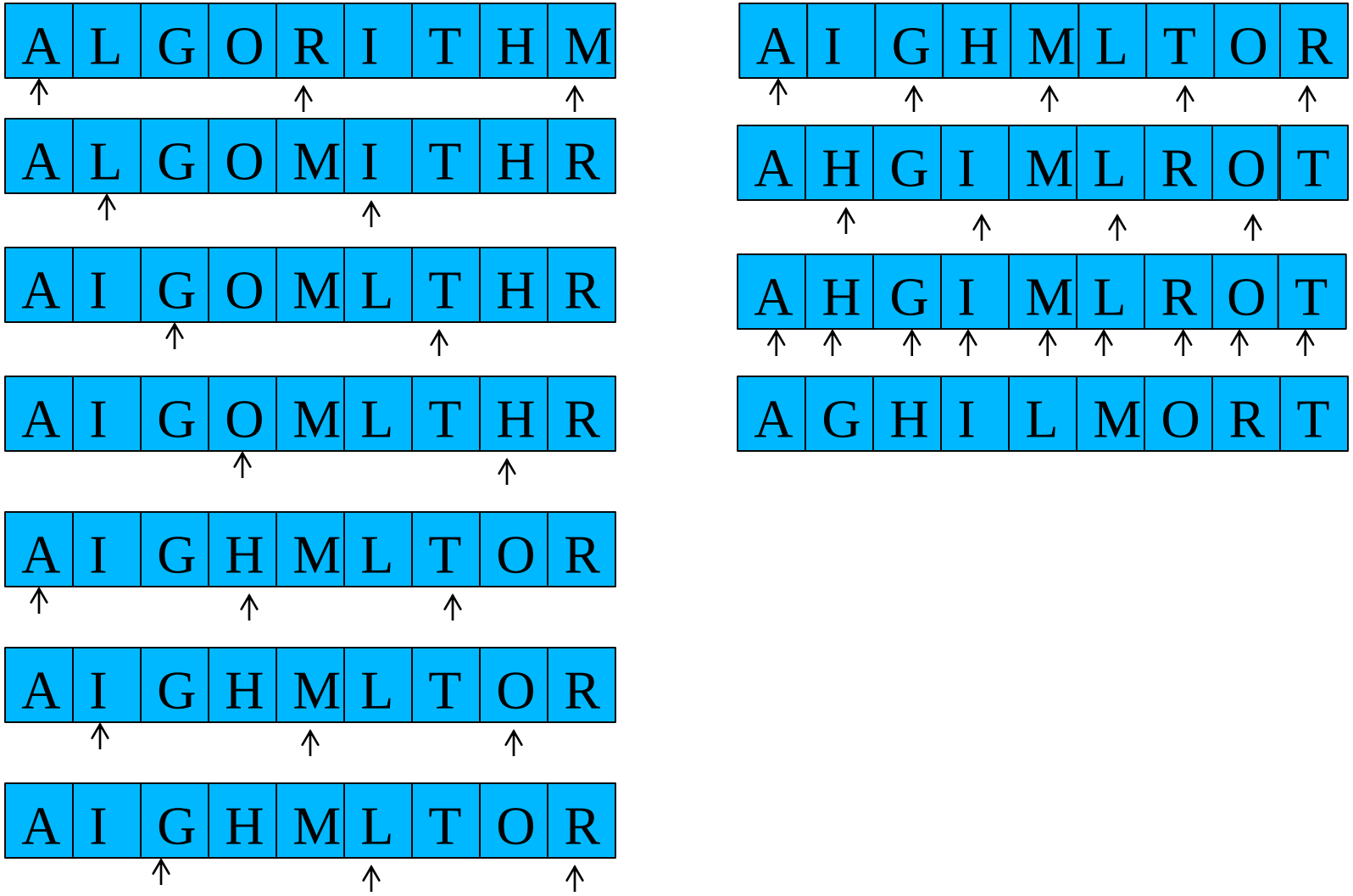
Here we have reduced K to 2. Just as in the insertion sort, if we swap 2 values, we have to go back and compare the previous 2 values to make sure they are still in order.

Shell Sort

Pass	List (K=1)										Notes
3	14	9	21	25	30	55	70	62	80	77	Swap
	9	14	21	25	30	55	70	62	80	77	In order
	9	14	21	25	30	55	70	62	80	77	In order
	9	14	21	25	30	55	70	62	80	77	In order
	9	14	21	25	30	55	70	62	80	77	In order
	9	14	21	25	30	55	70	62	80	77	In order
	9	14	21	25	30	55	70	62	80	77	Swap
	9	14	21	25	30	55	62	70	80	77	In order
	9	14	21	25	30	55	62	70	80	77	In order
	9	14	21	25	30	55	62	70	80	77	Swap
	9	14	21	25	30	55	62	70	77	80	In order

All shell sorts will terminate by running an insertion sort (i.e., K=1). However, using the larger values of K first has helped to sort our list so that the straight insertion sort will run faster.

Shell Sort


A L G O R I T H M
A L G O M I T H R
A I G O M L T H R
A I G O M L T H R
A I G H M L T O R
A I G H M L T O R
A I G H M L T O R
A I G H M L T O R
A H G I M L R O T
A H G I M L R O T
A G H I L M O R T

Sorting in Place – Shell Sort

h = 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
77	62	14	25	30	14	56	98	4	12	88	21	80	**9**	70	55

Sorting in Place – Shell Sort

h = 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
77	62	14	25	30	14	56	98	4	12	88	21	80	**9**	70	55
9	62	14	25	30	14	56	98	4	12	88	21	80	**77**	70	55

Sorting in Place – Shell Sort

h = 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
77	62	14	25	30	14	56	98	4	12	88	21	80	**9**	70	55
9	**62**	14	25	30	14	56	98	4	12	88	21	80	77	**70**	55

Sorting in Place – Shell Sort

h = 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
77	62	14	25	30	14	56	98	4	12	88	21	80	**9**	70	55
9	**62**	14	25	30	14	56	98	4	12	88	21	80	77	**70**	55
9	**62**	14	25	30	14	56	98	4	12	88	21	80	77	**70**	55

Sorting in Place – Shell Sort

h = 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
77	62	14	25	30	14	56	98	4	12	88	21	80	**9**	70	55
9	**62**	14	25	30	14	56	98	4	12	88	21	80	77	**70**	55
9	62	**14**	25	30	14	56	98	4	12	88	21	80	77	70	**55**

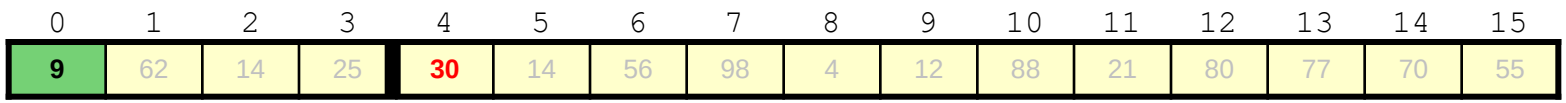
Sorting in Place – Shell Sort

h = 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
77	62	14	25	30	14	56	98	4	12	88	21	80	**9**	70	55
9	**62**	14	25	30	14	56	98	4	12	88	21	80	77	**70**	55
9	62	**14**	25	30	14	56	98	4	12	88	21	80	77	70	**55**
9	62	**14**	25	30	14	56	98	4	12	88	21	80	77	70	**55**

Sorting in Place – Shell Sort

h = 4



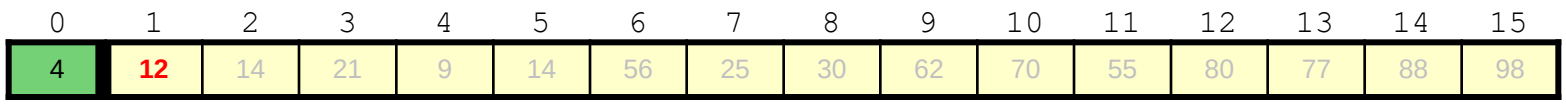
Sorting in Place – Shell Sort

h = 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
9	62	14	25	**30**	14	56	98	4	12	88	21	80	77	70	55
9	**62**	14	25	30	**14**	56	98	4	12	88	21	80	77	70	55
9	14	**14**	25	30	62	**56**	98	4	12	88	21	80	77	70	55
9	14	14	**25**	30	62	56	**98**	4	12	88	21	80	77	70	55
9	14	14	25	**30**	62	56	98	**4**	12	88	21	80	77	70	55
4	**14**	14	25	9	**62**	56	98	30	**12**	88	21	80	77	70	55
4	12	**14**	25	9	14	**56**	98	30	62	**88**	21	80	77	70	55
4	12	14	**25**	9	14	56	**98**	30	62	88	**21**	80	77	70	55
4	12	14	21	**9**	14	56	25	**30**	62	88	98	**80**	77	70	55
4	**12**	14	21	9	**14**	56	25	30	**62**	88	98	80	**77**	70	55
4	12	**14**	21	9	14	**56**	25	30	62	**88**	98	80	77	**70**	55
4	12	14	**21**	9	14	56	**25**	30	62	70	**98**	80	77	88	**55**
4	12	14	21	9	14	56	25	30	62	70	55	80	77	88	98

Sorting in Place – Shell Sort

h = 1



Sorting in Place – Shell Sort

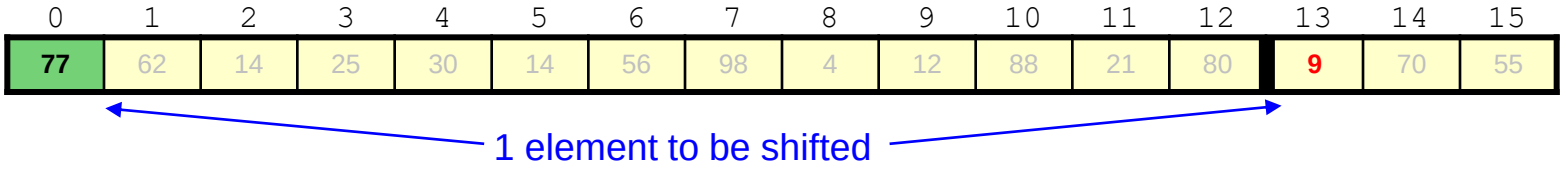
h = 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	**12**	14	21	9	14	56	25	30	62	70	55	80	77	88	98
4	12	**14**	21	9	14	56	25	30	62	70	55	80	77	88	98
4	12	14	**21**	9	14	56	25	30	62	70	55	80	77	88	98
4	12	14	21	**9**	14	56	25	30	62	70	55	80	77	88	98
4	9	12	14	21	**14**	56	25	30	62	70	55	80	77	88	98
4	9	12	14	14	21	**56**	25	30	62	70	55	80	77	88	98
4	9	12	14	14	21	56	**25**	30	62	70	55	80	77	88	98
4	9	12	14	14	21	25	56	**30**	62	70	55	80	77	88	98
4	9	12	14	14	21	25	30	56	**62**	70	55	80	77	88	98
4	9	12	14	14	21	25	30	56	62	**70**	55	80	77	88	98
4	9	12	14	14	21	25	30	56	62	70	**55**	80	77	88	98
4	9	12	14	14	21	25	30	55	56	62	70	**80**	77	88	98
4	9	12	14	14	21	25	30	55	56	62	70	80	**77**	88	98
4	9	12	14	14	21	25	30	55	56	62	70	77	80	**88**	98
4	9	12	14	14	21	25	30	55	56	62	70	77	80	88	**98**

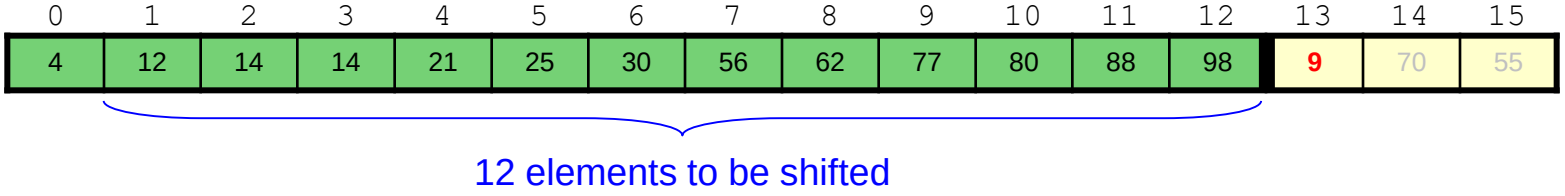
Advantage of Shell Sort

- Faster than the ordinary insertion sort
- More efficient exchange of elements that are far from their proper position

Shell sort



Insertion sort



Efficiency of Shell Sort

- **Efficiency depends of the values of h**
- **Values of h recommended by Knuth in 1969**

  `1 4 13 40 121 364 1093 3280 9841 ...`

  - Start with 1, then multiply by 3 and add 1
  - Less than **$O(n^{3/2})$** comparisons
- **Values of h recommended by Shell in 1959**

  `1 2 4 8 16 32 64 128 256 512 1024 2048 ...`

  - Bad sequence because elements in odd positions are not compared to elements in even positions until the final pass
  - Worst case efficiency of **$O(n^2)$** – smallest values in even positions and largest values in odd positions

How to Choose k

- Common Increments
  - Shell (1, 2, 4, 8, …, N/2)
  - Hibbard (1, 3, 7, 15, …, $2^n - 1$)
  - Knuth (1, 4, 13, 40, …, $3k_{n-1} + 1$)
- All series start with largest number and work towards 1

Shellsort

```
ShellSort(A, n)
1.  h ← 1
2.  while h ≤ n {
3.      h ← 3h + 1
4.   }
5.  repeat
6.      h ← h/3
7.      for i = h to n do {
8.          key ← A[i]
9.          j ← i
10.         while key < A[j - h] {
11.             A[j] ← A[j - h]
12.             j ← j - h
13.             if j < h then break
14.         }
15.         A[j] ← key
16.     }
17. until h ≤ 1
```

Calculate h

When *h*=1, this is insertion sort. Otherwise, performs insertion sort on keys *h* locations apart.

h values are set in the outermost repeat loop. Note that they are *decreasing* and the final value is 1.

쉘 정렬 (4)

- ShellSort 알고리즘

```
shellSort(a[])
    interval ← a.length;
    while (interval > 1) do  {
        interval ← 1 + interval / 3;
        for (i ← 0;  i < interval;  i ← i+1) do  {
                intervalSort(a, i, interval);
        }
    }
end ShellSort()
```

쉘 정렬 (5)

- intervalSort 알고리즘

```
intervalSort(a, i, interval)
// 서브리스트를 삽입 정렬로 정렬하는 ShellSort()의 보조 함수
   j ← i + interval;
   while (j < a.length) do  {
      new ← a[j];    // 서브리스트의 새로운 원소
      k ← j;    // new보다 큰 원소는 interval만큼 오른편으로 이동
      move ← true;
      while (move) do  {
         if (a[k – interval] ≤ new) then  {
               move ← false;
         }
         else  {
               a[k] ← a[k – interval];
               k ← k – interval;
               if (k = i) then
                  move ← false;
         }
      }
      a[k] ← new;    // 이동해서 생긴 자리에 삽입
      j ← j + interval;    // 다음 서브리스트 원소의 인덱스
   }
end intervalSort()
```

쉘 정렬 (6)

- Sorting 클래스의 메소드 멤버 구현

```
public static void shellSort(int[] a)  {
    int interval = a.length;
    • • • • •    // ShellSort 알고리즘의 Java 코드
    intervalSort(a, i, interval);
    • • • • •    // 기타 Java 코드
}

private static void intervalSort(int[] a, int i, int interval)  {
// 서브리스트를 삽입 정렬로 정렬하는 shellSort()의 보조 메소드
    • • • • •    // intervalSort 알고리즘의 Java 코드
}
```

Algorithm Analysis(쉘정렬분석)

- In-place sort(제자리정렬)
- Not stable(불안정)
- The exact behavior of the algorithm depends on the sequence of increments -- difficult & complex to analyze the algorithm.
- For $h_k = 2^k - 1$, $T(n) = \Theta(n^{3/2})$

Empirical Analysis of Shellsort (Advantage)

- Advantage of Shellsort is that its only efficient for medium size lists. For bigger lists, the algorithm is not the best choice. Fastest of all O(N^2) sorting algorithms.
- 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.

Empirical Analysis of Shellsort (Disadvantage)

- Disadvantage of Shellsort is that it is a complex algorithm(복잡한 알고리즘) and its not nearly as efficient as the merge, heap, and quick sorts. (합병,히프,퀵보다 빠르지 않다)
- The shell sort is still significantly slower than the merge, heap, and quick sorts, but its relatively simple algorithm makes it a good choice for sorting lists of less than 5000 items unless speed important. It ' s also an excellent choice for repetitive sorting of smaller lists. (5000개 이하의 키를 정렬하는데 빠르다)

Shellsort Best Case

- Best Case: The best case in the shell sort is when the array is already sorted in the right order. The number of comparisons is less.(거의 정렬된 입력데이터에 최적인 알고리즘)

Shellsort Worst Case

- The running time of Shellsort depends on the choice of increment sequence(*h*).
- The problem with Shell's increments is that pairs of increments are not necessarily relatively prime and smaller increments can have little effect.

Sorting Properties

Name	Compari-son Sort	Avg Case Complexity	Worst Case Complexity	In Place	Can be Stable
Insertion	√	$O(n^2)$	$O(n^2)$	√	√
Bubble	√	$O(n^2)$	$O(n^2)$	√	√
Shell	√	?	$O(n^2)$		
Heap	√	O(n log(n))	O(n log(n))		
Quick	√	O(n log(n))	$O(n^2)$	√	
Merge	√	O(n log(n))	O(n log(n))		√
Counting		O(n)	O(n)		√
Bucket		O(n)	$O(n^2)$		√
Radix		O(n)	O(n)		√

감사합니다.