ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Sorts

Algorithms and data structures

Exercise 6

Sorting

- A big number of operations are simplified by sorting: to find given element in array, to determine whether given element is unique in array, to determine the frequency of given element in array, to find the *k*-th biggest element in array etc.
- The simplest sorting algorithm is selection sort (in each iteration the minimal element of the unsorted part is found and it is set as last element in the sorted part). The complexity of this algorithm is $O(n^2)$, and takes a lot of time to sort array with more than 10000 elements.
- Most of the sorting algorithms are based on elements value comparison. The lower bound on complexity for these types of algorithms is $O(n \log_2 n)$.
- Faster algorithms exist, but usually they can be applied only under specific conditions.

Sorting

- Sorting by comparison
  - Insertion sort
  - Bubble sort
  - Merge sort
  - Quick sort
- Sorting without comparison
  - Counting sort
  - Bucket sort
  - Radix sort

Insertion sort

- The sorting is performed in the following way:

  we observe the elements of the array in in two subsets: $a_0,\ldots,a_{i-1}$ as sorted part of the array and $a_i,\ldots,a_n$ as unsorted part of the array.

  At the beginning only the first element of the array a_0 belongs to the sorted part, while the rest elements belong to the unsorted part.

  In each step starting with i=1 and increasing i by 1, i-th element of the array is considered and it is inserted in the “appropriate" place in the sorted part of the array.

Insertion sort - Illustration

i = 1: **8 2 4 9 3 6**

i = 2: **2 8 4 9 3 6**

i = 3: **2 4 8 9 3 6**

i = 4: **2 4 8 9 3 6**

i = 5: **2 3 4 8 9 6**

2 3 4 6 8 9

Insertion sort - Java

```
public static <T extends Comparable<? super T>> void
   insertionSort(T[] array)
{
   T key;
   int begin = 0;
   int end = array.length - 1;
   int i;

   for (int index = begin + 1; index <= end; index++)
  {
        key = array[index];
        i = index - 1;
        while (i >= begin && array[i].compareTo(key) > 0) {
               array[i + 1] = array[i];
               i = i - 1;
        }
        array[i + 1] = key;
   }
}
```

Insertion sort – Complexity

- $O(n^2)$
- Best case complexity:
  - If the array is already sorted, the inner loop won't be executed at all, so the complexity is : $T(n) = \Omega(n)$
- Worst – case complexity:
  - If the array is sorted in reverse order. In this case in the first iteration the inner loop will be executed once, in the second iteration the inner loop will be execute twice etc... in the last iteration the inner loop will be executed *n* times.

$$T(n) = \sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + n = O(n^2)$$

- Average case complexity:
  - In average case, the inner loop of the *i*-th iteration will be executed *i*/2 time, so average time would be:

$$T(n) = \sum_{i=1}^{n-1} i/2 = \Theta(n^2)$$

Bubble sort

- This algorithm starts from the last element and in the first iteration the smallest element is „pushed“ at the first position of the array, like bubble coming to the surface.

  In the second iteration the second smallest element of the array (i.e. the smallest from the rest of the array elements) is „pushed“ at the second position in the array... The algorithm is repeated *n* times/iterations (i.e. all *n* elements are set in appropriate positions).

  In order to decrease complexity in situation when the array is almost sorted and after only few iteration it is sorted, we can introduce variable that will check whether the array is sorted and will stop execution.

Bubble sort - Illustration

i = 1: **8 2 4 9 3 6** (6 is pushed to the left until it touches smaller element: 3)
8 2 4 9 3 6 (3 is pushed to the left until it touches smaller element : 2)
8 2 3 4 9 6 (2 is pushed to the left and is set to its place)
2 8 3 4 9 6

i = 2: **2 8 3 4 9 6**
2 8 3 4 6 9
2 8 3 4 6 9
2 3 8 4 6 9

i = 3: **2 3 8 4 6 9**
2 3 8 4 6 9
2 3 8 4 6 9
2 3 4 8 6 9

i = 4: **2 3 4 8 6 9**
2 3 4 8 6 9
2 3 4 6 8 9 (the array is already sorted, no need for new iteration)

Bubble sort - Java

```
public static <T extends Comparable<? super T>> void
   bubbleSort(T[] array)
{
   int begin = 0;
   int end = array.length - 1;
   T temp; boolean flipped = false;

   for (int i = end; i >= begin; i--) {
        flipped = false;
        for (int j = 1; j <= i; j++) {
               if (array[j - 1].compareTo(array[j]) > 0) {
                     temp = array[j - 1];
                     array[j - 1] = array[j];
                     array[j] = temp;
                     flipped = true;
               }
        }
        if(!flipped) break;
   }
} // another implementation – the biggest element is pushed right
```

Bubble sort – Complexity

- The outer loop is executed *n*-1 times (until all *n* elements are set in appropriate positions). In first iterations *n*-1 comparisons and possible swaps are executed, in second iteration *n*-2... The complexity is:

$$T(n) = n-1 + n-2 + \ldots + 1 = \frac{n*(n-1)}{2} = O(n^2)$$

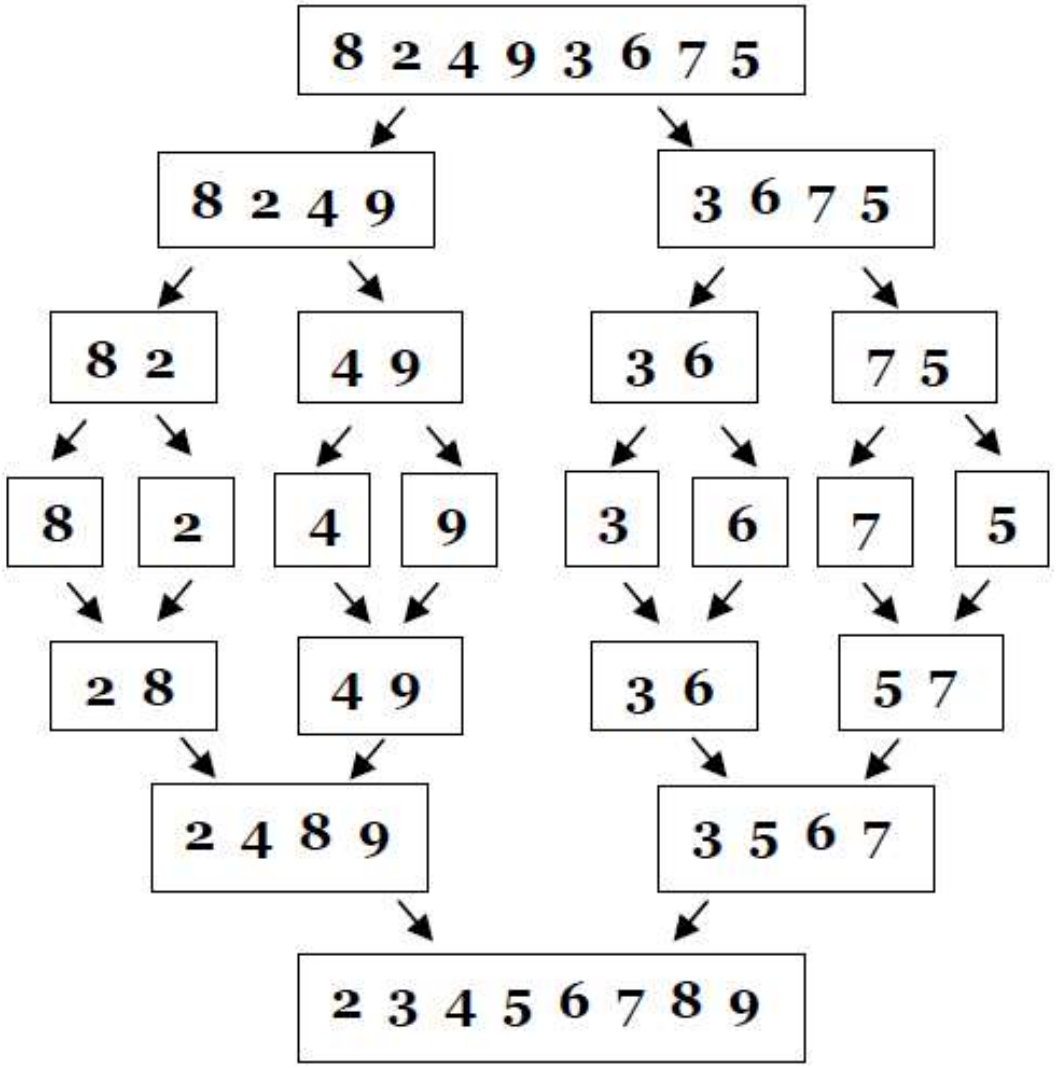
- Best case complexity:
  - If the array is already sorted only the first iteration will be executed after which the variable *flipped* will determine that the array is sorted, so the complexity is the number of comparisons in first iteration:

$$T(n) = \Omega(n)$$

Merge sort

- "Divide and conquer" algorithm in 3 steps for mergesort of an array [0 … n]:
  - **Divide**: the array a[0..n] is halved into two sub arrays
  - **Conquer**: each half a[0 .. (n/2)] and a[(n/2) +1 .. n] is sorted recursively
  - **Merge**: both sorted halves are merged into a sorted array

Merge sort - Illustration



Merge sort - Java

```
public static  <T extends Comparable<? super T>> void mergeSort(T a[],
   int l, int r)
{
   if (l == r) {
        return;
   }

   int mid = (l + r) / 2;
   mergeSort(a, l, mid);
   mergeSort(a, mid + 1, r);
   merge(a, l, mid, r);
}
```

Merge sort - Java

```
public static  <T extends Comparable<? super T>> void merge(T a[], int
   l, int mid, int r)
 {
   int numel = r - l + 1;
   T temp[] = (T[]) new Comparable[a.length];

   int i = l, j = mid+1, k = 0;

   while ((i <= mid) && (j <= r)) {
        if (a[i].compareTo(a[j]) < 0) {
                temp[k] = a[i];
                i++;
        } else {
                temp[k] = a[j];
                j++;
        }
        k++;
   }
```

Merge sort - Java

```
    while (i <= mid) {
          temp[k] = a[i];
          i++;
          k++;
    }

    while (j <= r) {
          temp[k] = a[j];
          j++;
          k++;
    }

    for (k = 0; k < numel; k++) {
          a[l + k] = temp[k];
    }
}
```

Merge sort - Complexity

- Algorithm complexity:

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

$2T(n/2) = 2(2(T(n/4))+n/2) = 4T(n/4)+n \rightarrow$

$T(n) = 4T(n/4)+2n$

$4T(n/4) = 4(2T(n/8))+(n/4) = 8T(n/8)+n \rightarrow$

$T(n) = 8T(n/8)+3n$

…

$T(n) = 2^k T(n/ 2^k)+kn$ and with k=log(n):

$T(n) = nT(1)+n \log(n) = n \log(n)+n$

Hence, the complexity is $\mathbf{O(}\boldsymbol{n}\mathbf{log}_2\boldsymbol{n}\mathbf{)}$.

Quick sort

- Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are:

1. Pick an element, called a pivot, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Quick sort - Illustration

6	10	13	(5)	8	3	2	11	5 is pivot
6	10	13	(5)	8	3	2	11	2 and 6 swap places
2	10	13	(5)	8	3	6	11	10 and 3 swap places
2	3	13	(5)	8	10	6	11	5 and 13 swap places
2	3	5	13	8	10	6	11	5 is where it should be
(2)	3	5	13	8	10	6	11	2 is pivot of the left sub array
2	3	5	13	8	(10)	6	11	2 and 3 are where they should be, 10 is pivot of the right sub array
2	3	5	13	8	10	6	11	13 and 6 swap places
2	3	5	6	8	10	13	11	10 is where it should be
2	3	5	(6)	8	10	13	11	6 is pivot of the left sub array
2	3	5	6	8	10	(13)	11	6 and 8 are where they should be, 13 is pivot of the right sub array, 11 and 13 swap places
2	3	5	6	8	10	11	13	11 and 13 are where they should be

Quick sort - Java

```
public static <T extends Comparable<? super T>> void swap(T A[], int
   x, int y) {
   T temp = A[x];
   A[x] = A[y];
   A[y] = temp;
}

public static <T extends Comparable<? super T>> void quickSort(T A[],
   int left, int right)
{
   int pivot_indeks = partition(A,left,right);
   if(left < pivot_indeks-1)
        quickSort(A, left, pivot_indeks-1);
   if(pivot_indeks < right)
        quickSort(A, pivot_indeks, right);
}
```

Quick sort - Java

```
public static <T extends Comparable<? super T>> int partition(T A[],
        int left, int right)
{

   int i = left, j = right;
   T pivot = A[(left + right) / 2];
   while (i <= j) {
        while (A[i].compareTo(pivot) < 0)
                i++;
        while (A[j].compareTo(pivot) > 0)
                j--;
        if (i <= j) {
                swap(A, i, j);
                i++;
                j--;
        }
   }
   return i;
}
```

Quick sort - Complexity

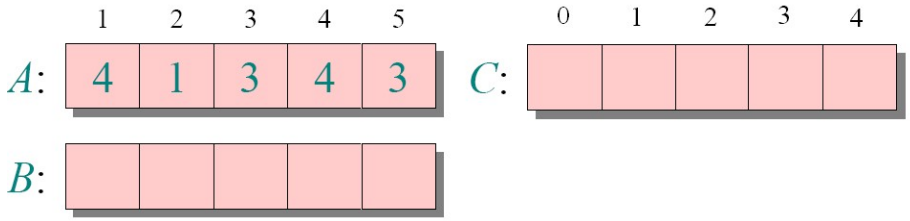
- In every iteration after picking a pivot and elements reordering the algorithm executes *n* comparisons and swaps. Next the problem is divided in two sub problems where again *n* comparisons and swaps are executed. Partitioning the array in two sub arrays, as in merge sort, requires $\log_2 n$ partitions. This means that the complexity is $O(n \log_2 n)$. Yet, for this to be true, the pivot selection needs to split the array in two similar in size sub arrays.
- The complexity of quicksort significantly depends upon pivot selection. If the pivot in each iteration is the smallest or the biggest element it splits the array in one sub array with only one element while the rest of the elements are the other sub array, then n-1 iteration will be executed. This is the worst case complexity $O(n^2)$.
- The pivot selection problem can be solved by choosing a random index for the pivot

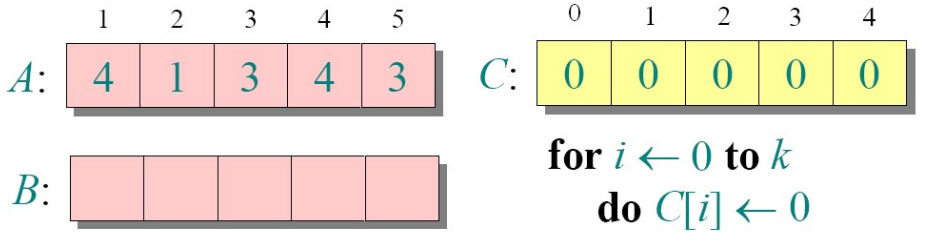
Counting sort

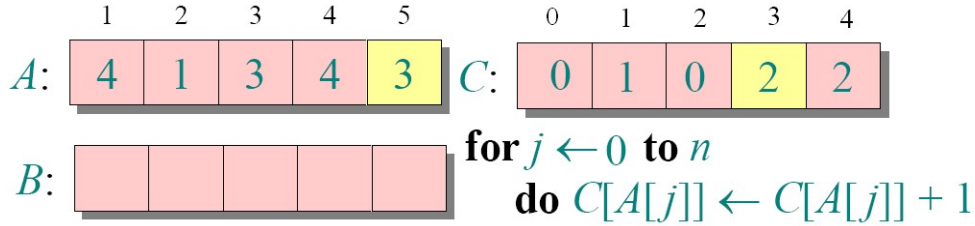
- The counting sort runs in linear time. It doesn't perform comparisons. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. It can be applied when the arrays elements are values from final set, i.e. the values are in range 0 - *k*:

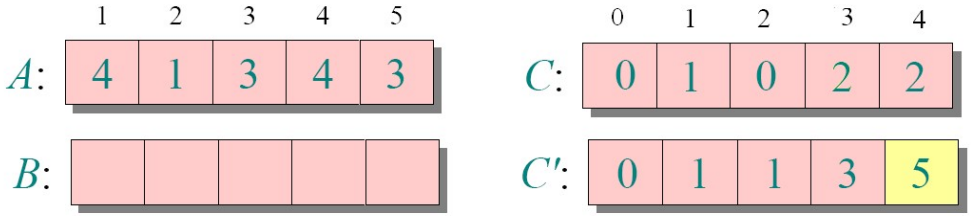
$$a[j] \in \{0,1,2,\ldots,k\}$$

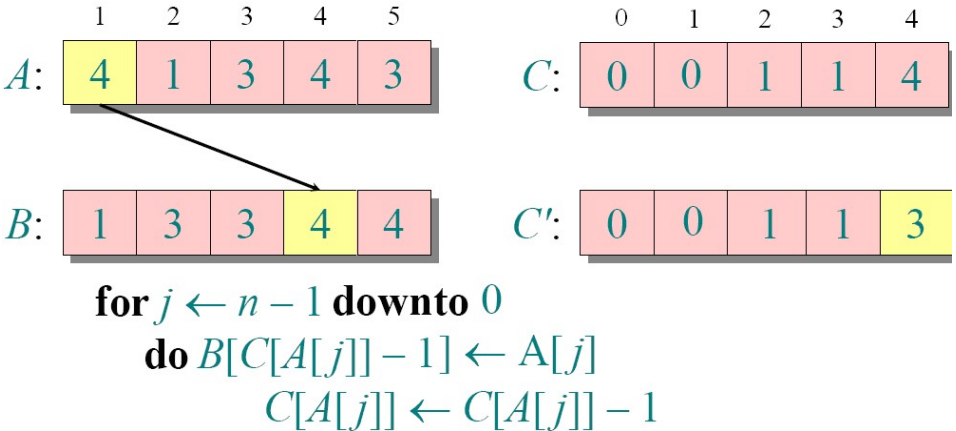
Counting sort - Illustration











Counting sort - Java

```
public static void countingSort(Integer A[], int k)
{
   int i, j;
   Integer c[] = new Integer[k+1];
   Integer b[] = new Integer[A.length];

   for (i=0; i <= k; i++)
      c[i] = 0;
   for (j=0; j < A.length; j++)
      c[A[j]] = c[A[j]] + 1;
   for (i = 1;  i <= k;  i++)
      c[i] = c[i] + c[i-1];
   for (j = A.length-1 ; j >= 0; j--)
   {
        b[c[A[j]] - 1] = A[j];
        c[A[j]] = c[A[j]] - 1;
   }

   for (j=0; j < A.length; j++)
          A[j] = b[j];
}
```

Counting sort – Complexity

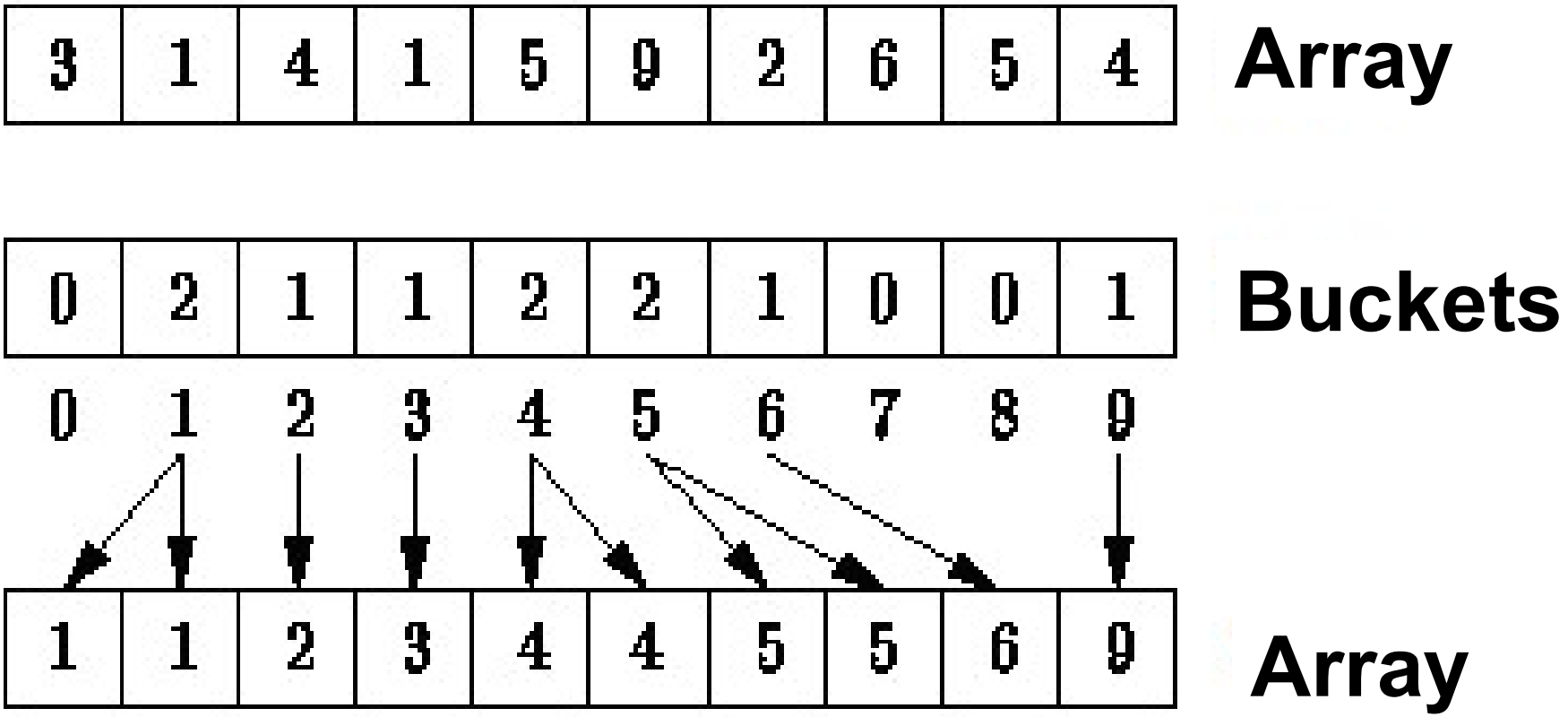
- The counting sort algorithm for array with n elements with k different values executes the following operations:
  - Two loops with k operations (elements occurrence initialization and total sum) - O(*k*) complexity
  - Two loops with n operation - O(*n*) complexity
  - Total complexity is *O(n+k)*
- This is the worst and average performance

Bucket sort

- This is variation of counting sort
- The algorithm works as follows:

1. Given an array of values to be sorted, set up an auxiliary array of initially empty “buckets," one bucket for each key through the range of the original array.
2. Going over the original array, put each value into the bucket corresponding to its key, such that each bucket eventually contains a list of all values with that key.
3. Iterate over the bucket array in order, and put elements from non-empty buckets back into the original array.

Bucket sort - Illustration



Bucket sort - Java

```
public static void bucketSort(Integer A[], int maxVal){
   Integer [] bucket=new Integer[maxVal+1];

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

   for (int i=0; i<A.length; i++){
        bucket[A[i]]++;
   }

   int outPos=0;
   for (int i=0; i<bucket.length; i++){
        for (int j=0; j<bucket[i]; j++){
               A[outPos++]=i;
        }
   }
}
```

Bucket sort - Complexity

- The bucket sort algorithm for array with n elements with k different values executes the following operations:
  - One loop with k operations (buckets initialization) - O(*k*) complexity
  - One loops with n operations (elements counting) - O(*n*) complexity
  - Two nested loops where the outer loop iterates k buckets and gets elements from each bucket and the inner loops is executed $c_1+c_2+..+c_n$ times, where c_i is number of elements in i-th bucket (i.e. n times in total) - O(*n+k*) complexity
  - Total complexity is O(*n+k*)
- This is the worst and average performance

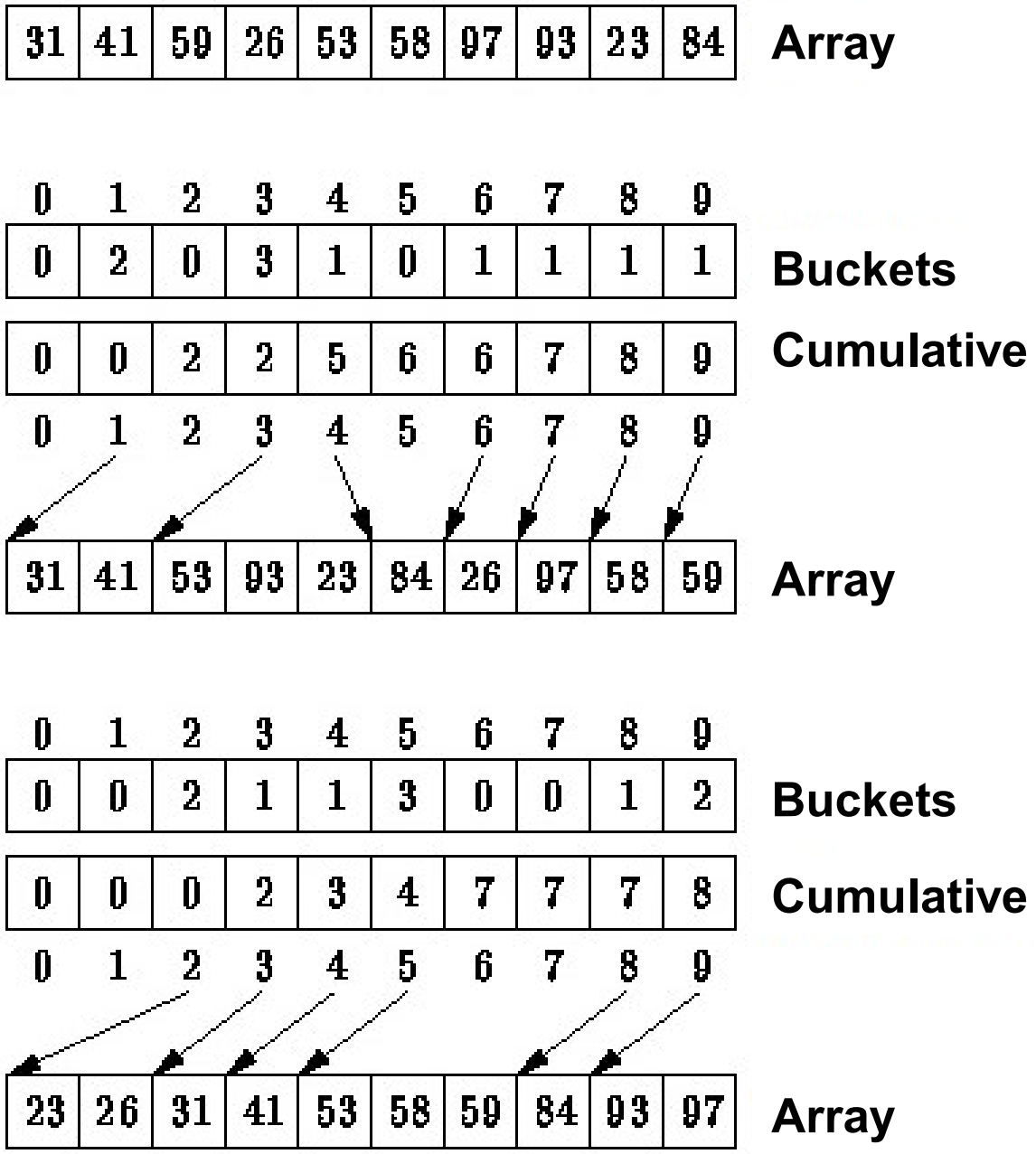
Radix sort

- Radix sort is a non-comparative integer sorting algorithm
- It sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value
  - E.g. for number no grater than 999 first sorts the elements according to the least significant digit (LSD) - ones, then according the to next significant digit - tens and at last according to the most significant digit (MSD) - hundreds
- Based on bucket sort, but more practical in case of bigger keys set
- Each element is considered as an array of digits (or letters or symbols)

Radix sort - Sample

- To sort an array having elements with values between 0 and 99
  - The algorithm will execute two iterations:
    - In first iteration will sort the elements according to the least significant digit - the ones
    - In second iteration will sort the elements according to the next significant digit –the tens
  - Every iteration is based on bucket sort

Radix sort - Illustration



Radix sort - Java

```
public static void radixSort(Integer A[]){
    Integer[] pom = new Integer[A.length];
    int i, max = A[0], exp = 1;

    for (i = 0; i < A.length; i++)
    {
      if (A[i] > max)
         max = A[i];
    }

    while (max / exp > 0)
    {
      int bucket[] =  new int[10];
      for (i = 0; i < A.length; i++)
         bucket[A[i] / exp % 10]++;
      for (i = 1; i < 10; i++)
         bucket[i] += bucket[i - 1];
      for (i = A.length - 1; i >= 0; i--)
         pom[--bucket[A[i] / exp % 10]] = A[i];
      for (i = 0; i < A.length; i++)
         A[i] = pom[i];
      exp *= 10;
    }
}
```

Radix sort - Complexity

- Fixed number of values of each position (10 in the illustration)
- If each element has maximum *k* positions the complexity is *O(nk)*
- Since the number of values of each position is fixed the complexity is *O(n)*