Radix Sort

Comparison sort runtime of O(n*log(n)) is optimal

- The *problem* of sorting cannot be solved using comparisons with less than n*log(n) time complexity
- See Proposition I in Chapter 2.2 of the text

How can we sort without comparison?

- Consider the following approach:
  - Look at the least-significant digit
  - Group numbers with the same digit
    - Maintain relative order
  - Place groups back in array together
    - I.e., all the 0' s, all the 1' s, all the 2' s, etc.
  - Repeat for increasingly significant digits

The characteristics of Radix sort

- Least significant digit (LSD) Radix sort
  - a fast stable sorting algorithm
  - begins at the least significant digit (e.g. the rightmost digit)
  - proceeds to the most significant digit (e.g. the leftmost digit)
  - lexicographic orderings

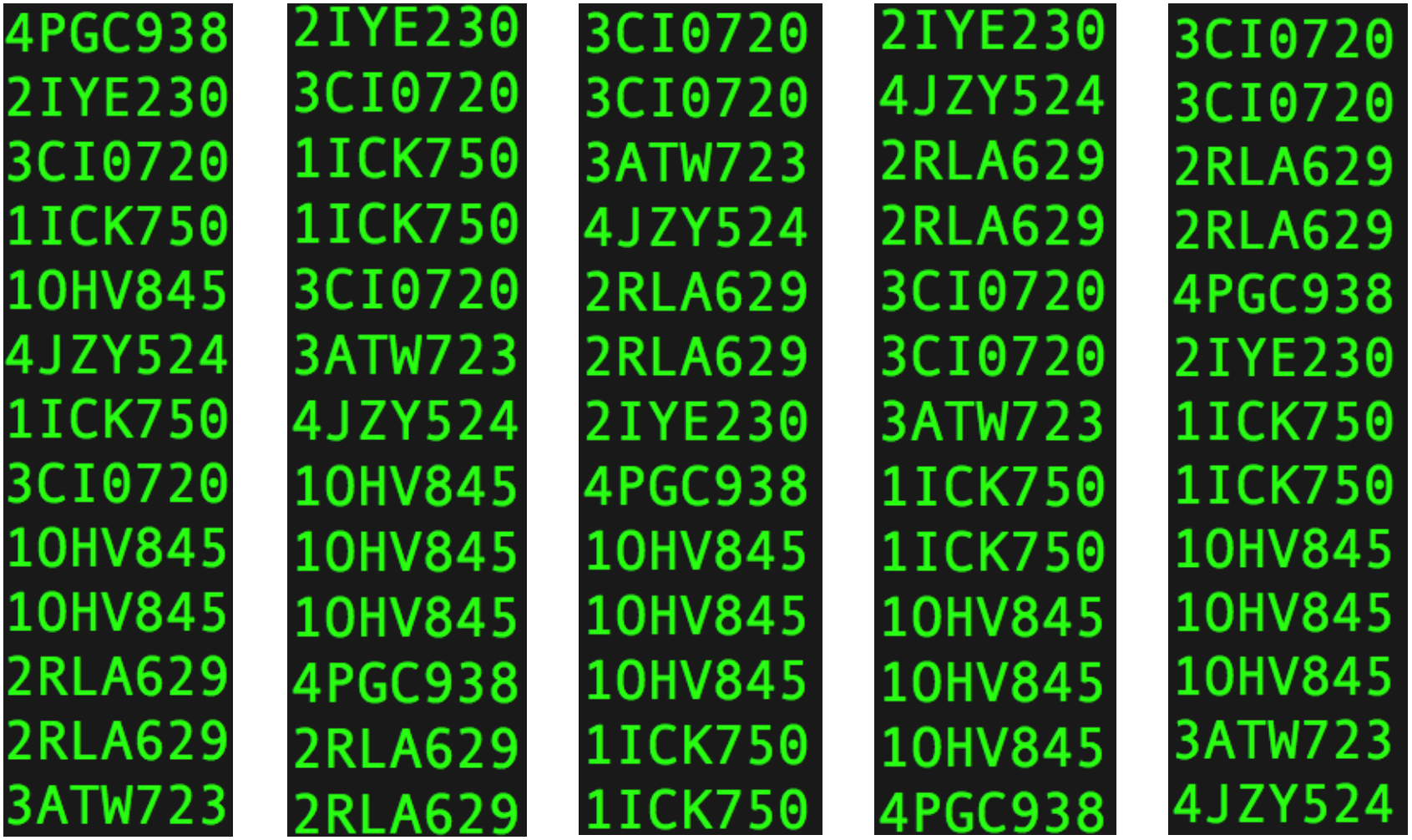
Generally Speaking

- 1. Take the least significant digit (or group of bits) of each key
- 2. Group the keys based on that digit, but otherwise keep the original order of keys.
  - This is what makes the LSD radix sort a stable sort.
- 3. Repeat the grouping process with each more significant digit

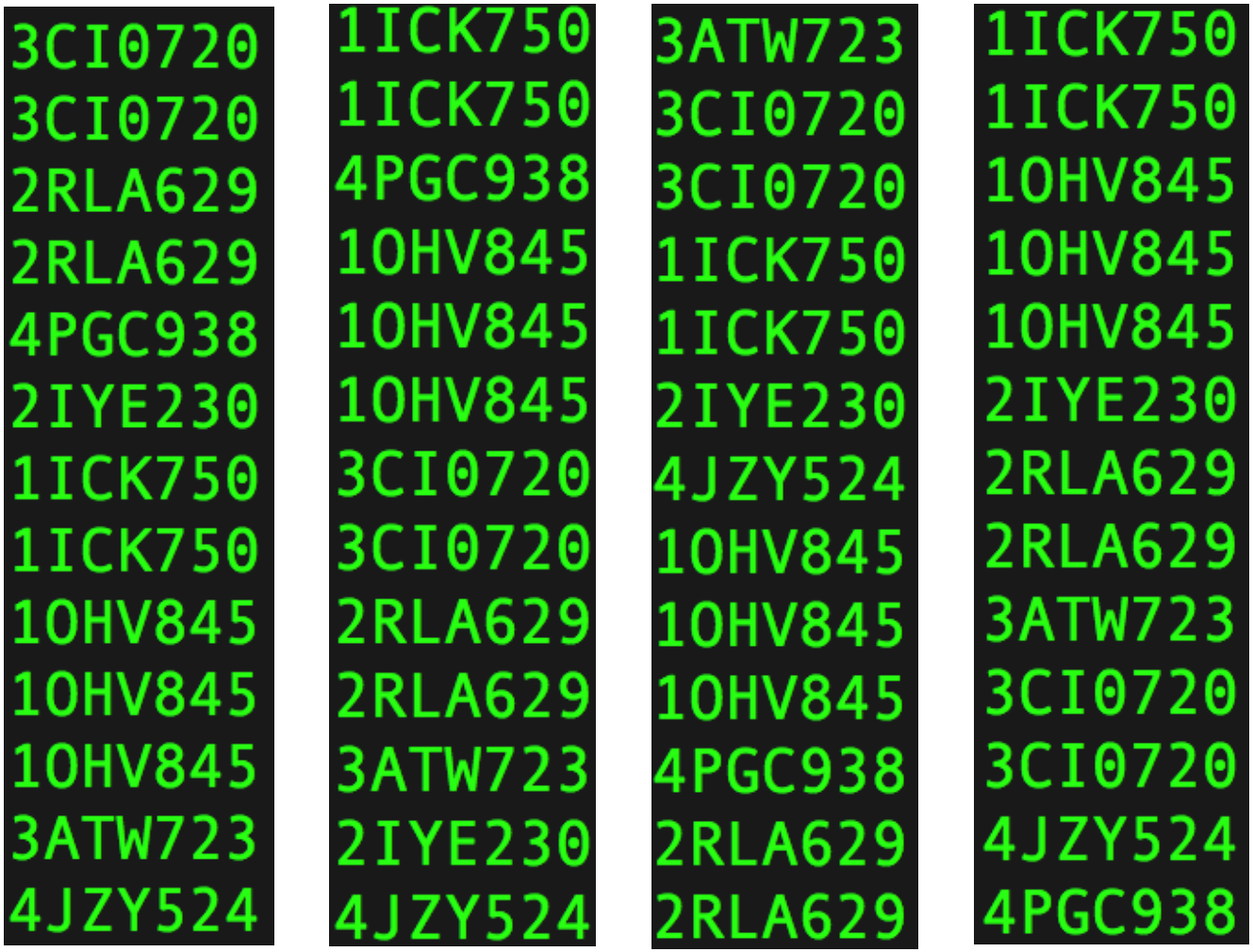
Generally Speaking

```
public static void sort(String [] a, int W) {
        int N = a.length;
        int R = 256;
        String [] aux = new String[N];
        for (int d = W - 1; d >= 0; --d) {
            aux = sorted array a by the d^th character
            a = aux
        }
}
```

A Radix sort example

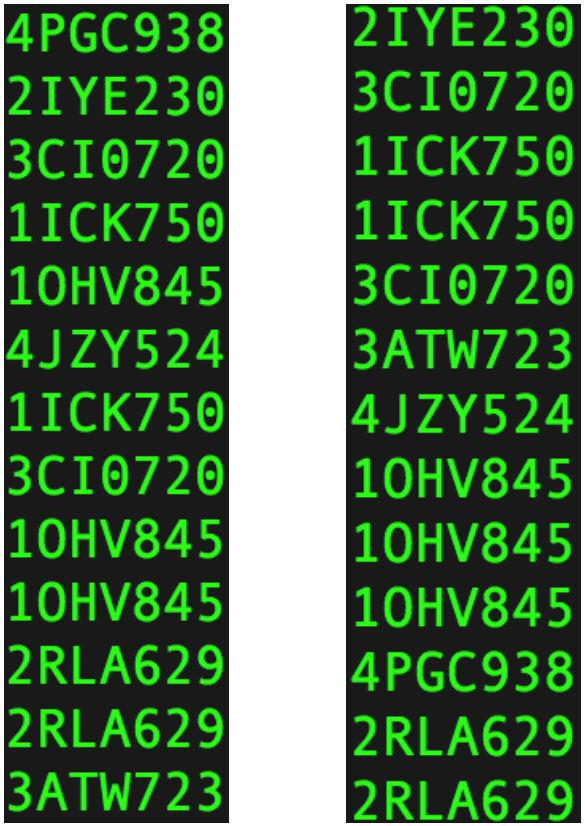


A Radix sort example



A Radix sort example

- Problem: How to ?
  - Group the keys based on that digit,
  - but otherwise keep the original order of keys.



Key-indexed counting

- 1. Take the least significant digit (or group of bits) of each key
- 2. Group the keys based on that digit, but otherwise keep the original order of keys.
  - This is what makes the LSD radix sort a stable sort.
- 3. Repeat the grouping process with each more significant digit

- The sort in step 2 is usually done using **bucket sort or counting sort**, which are efficient in this case

Key-indexed counting

```
public static void sort(String [] a, int W) {
        int N = a.length;
        int R = 256;
        String [] aux = new String[N];
        for (int d = W - 1; d >= 0; --d) {
            aux = sorted array a by the d^th character (stable
sort)
            a = aux
        }
}
```

Key-indexed counting

```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
        count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
        count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal
keys:
for (int i = 0; i < N; ++i) {
        aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
        a[i] = aux[i];
}
```

Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 1: count the frequencies***

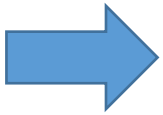
```
 0:  0
 1:  5
 2:  0
 3:  0
 4:  1
 5:  1
 6:  3
 7:  0
 8:  0
 9:  1
10:  2
```

```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```

```
4PGC938
2IYE230
3CI0720
1ICK750
1OHV845
4JZY524
1ICK750
3CI0720
1OHV845
1OHV845
2RLA629
2RLA629
3ATW723
```

Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 2: accumulate the frequencies***
  - ***What is count[] mean?***

0:	0		0: 0
1:	5		1: 5
2:	0		2: 5
3:	0		3: 5
4:	1		4: 6
5:	1		5: 7
6:	3		6: 10
7:	0		7: 10
8:	0		8: 10
9:	1		9: 11
10:	2		10: 13

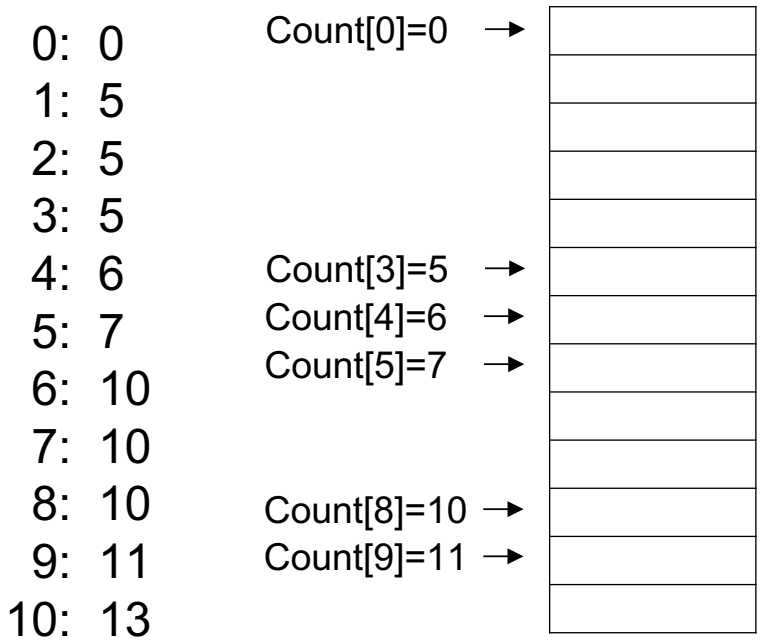
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```

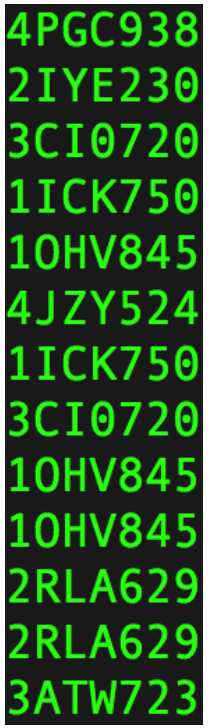
```
4PGC938
2IYE230
3CI0720
1ICK750
1OHV845
4JZY524
1ICK750
3CI0720
1OHV845
1OHV845
2RLA629
2RLA629
3ATW723
```

Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 2: accumulate the frequencies***
  - ***What is count[] mean?***

```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```

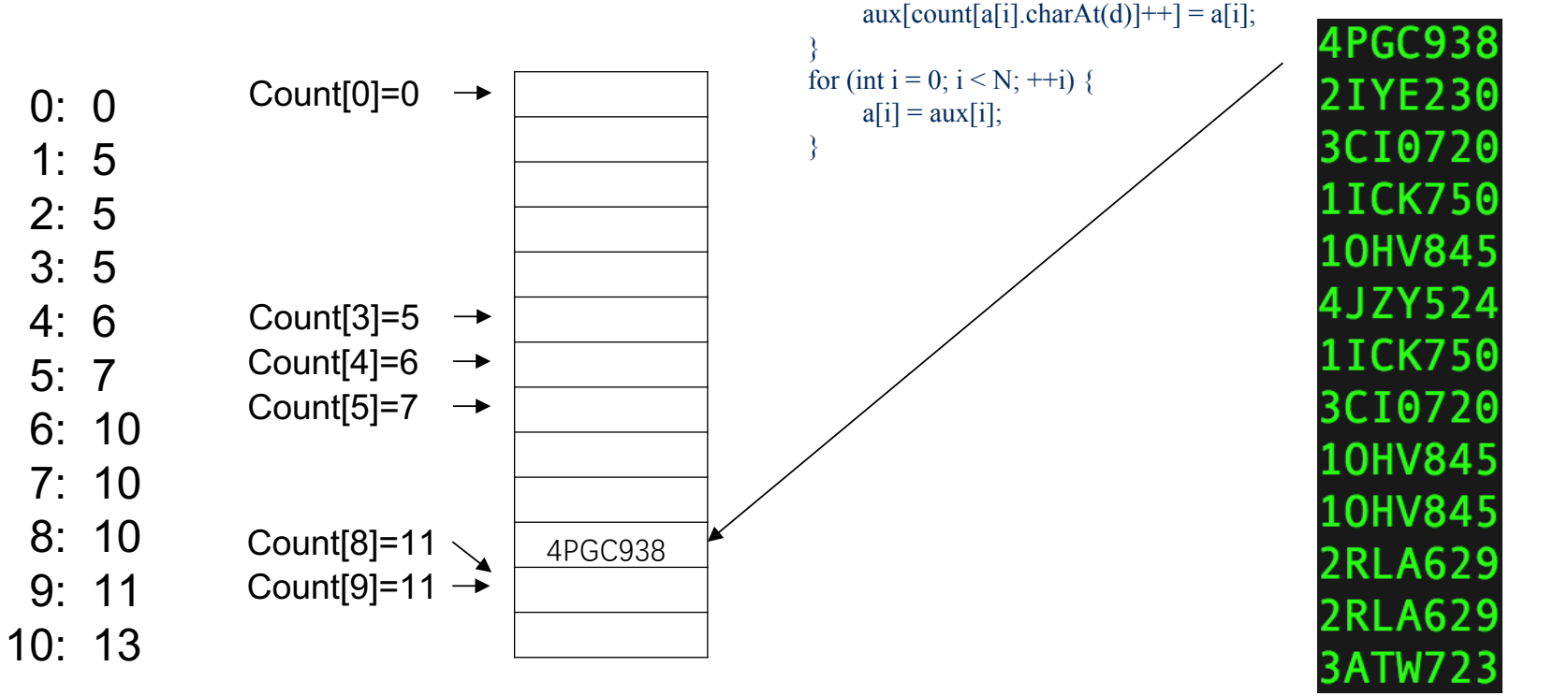




Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

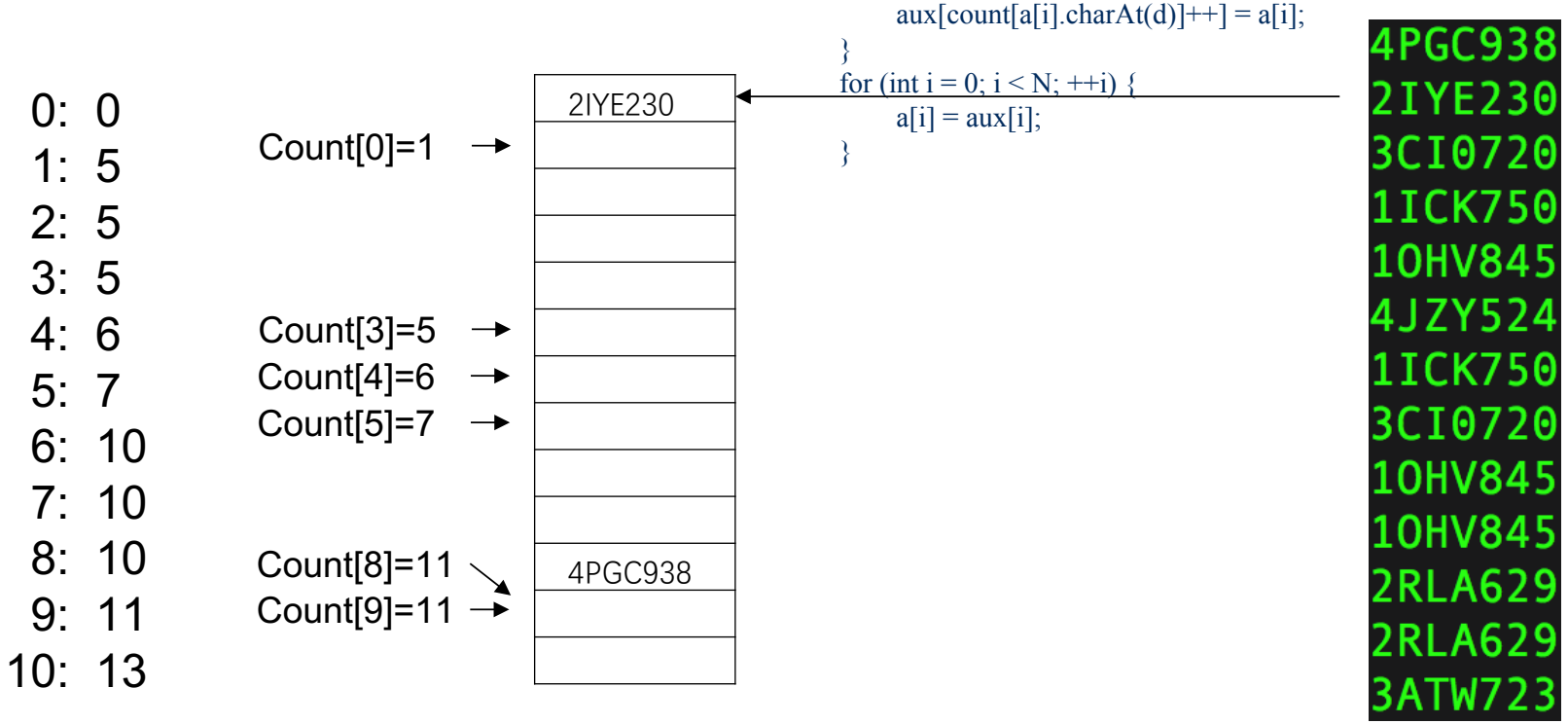
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

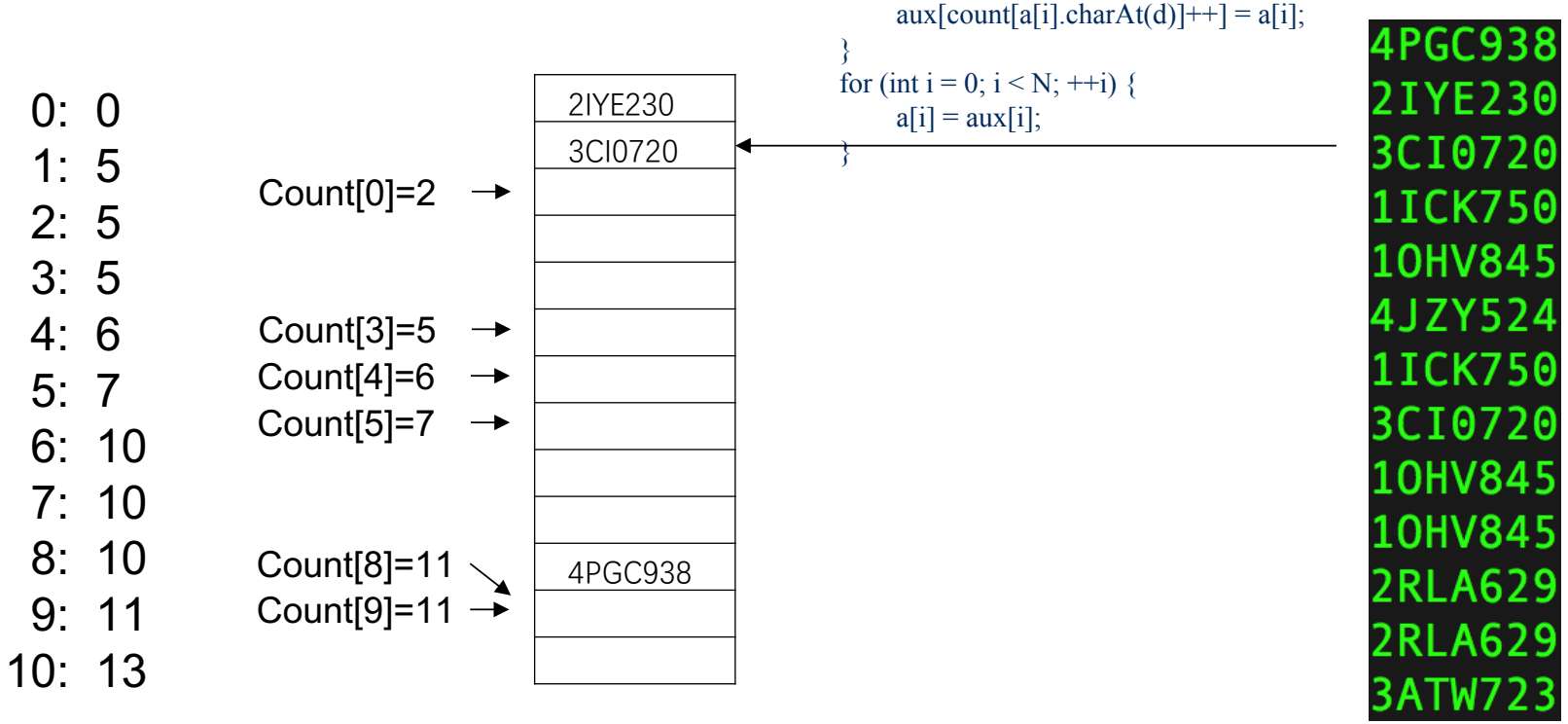
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

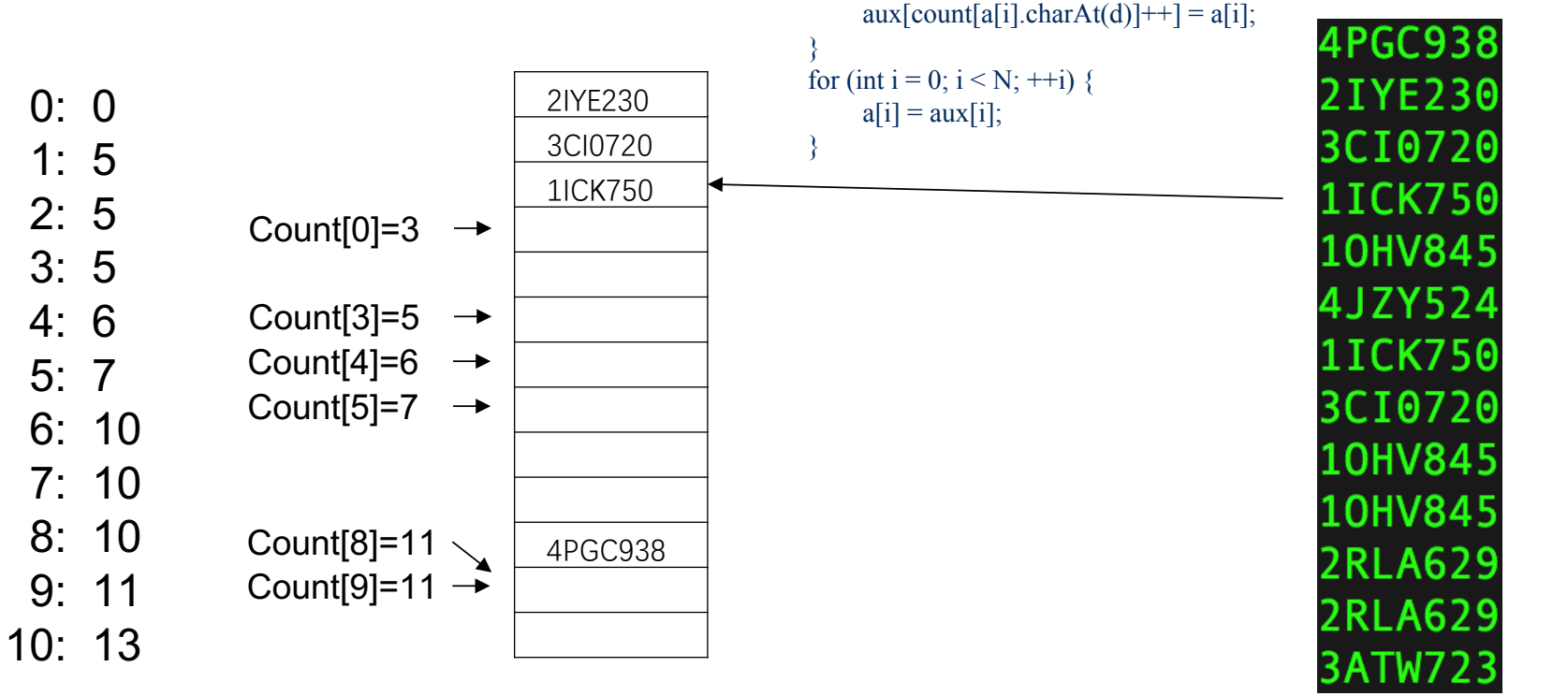
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

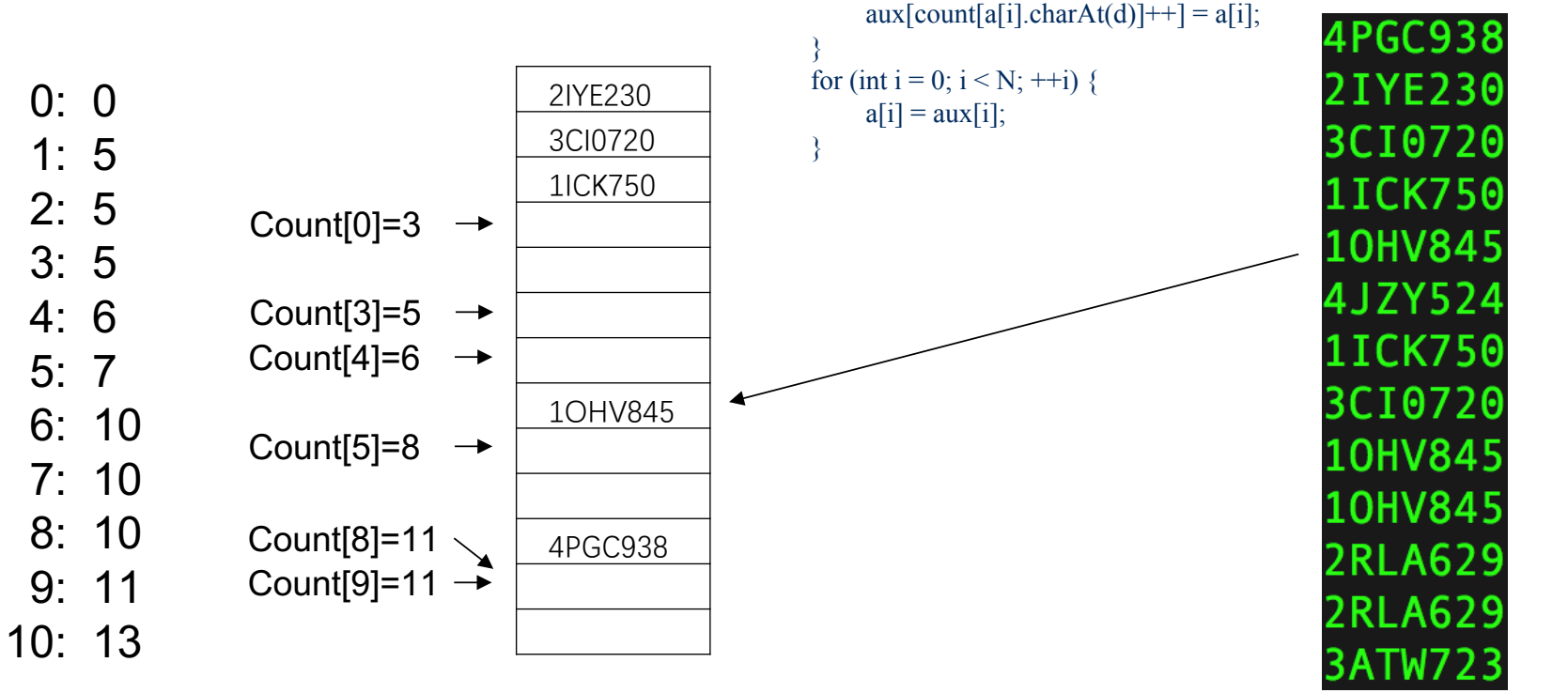
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

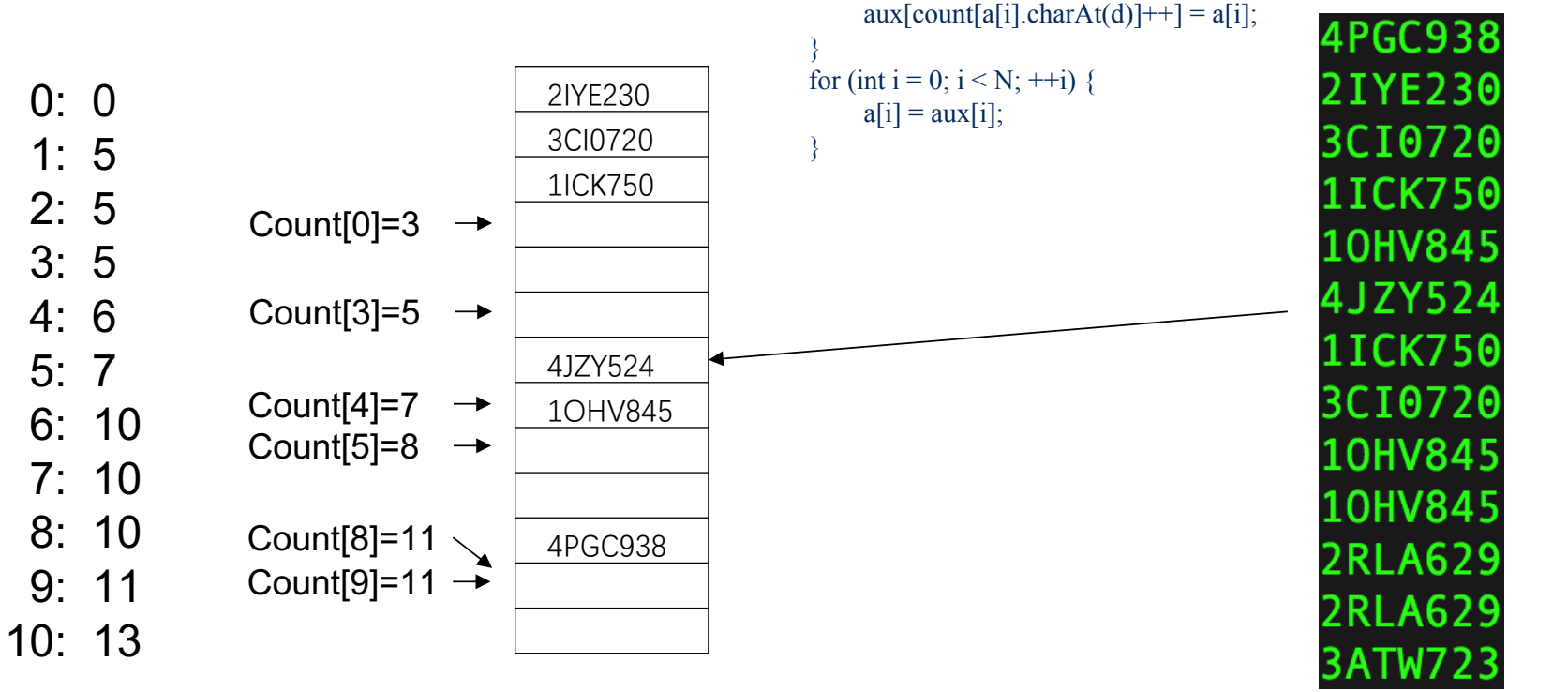
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

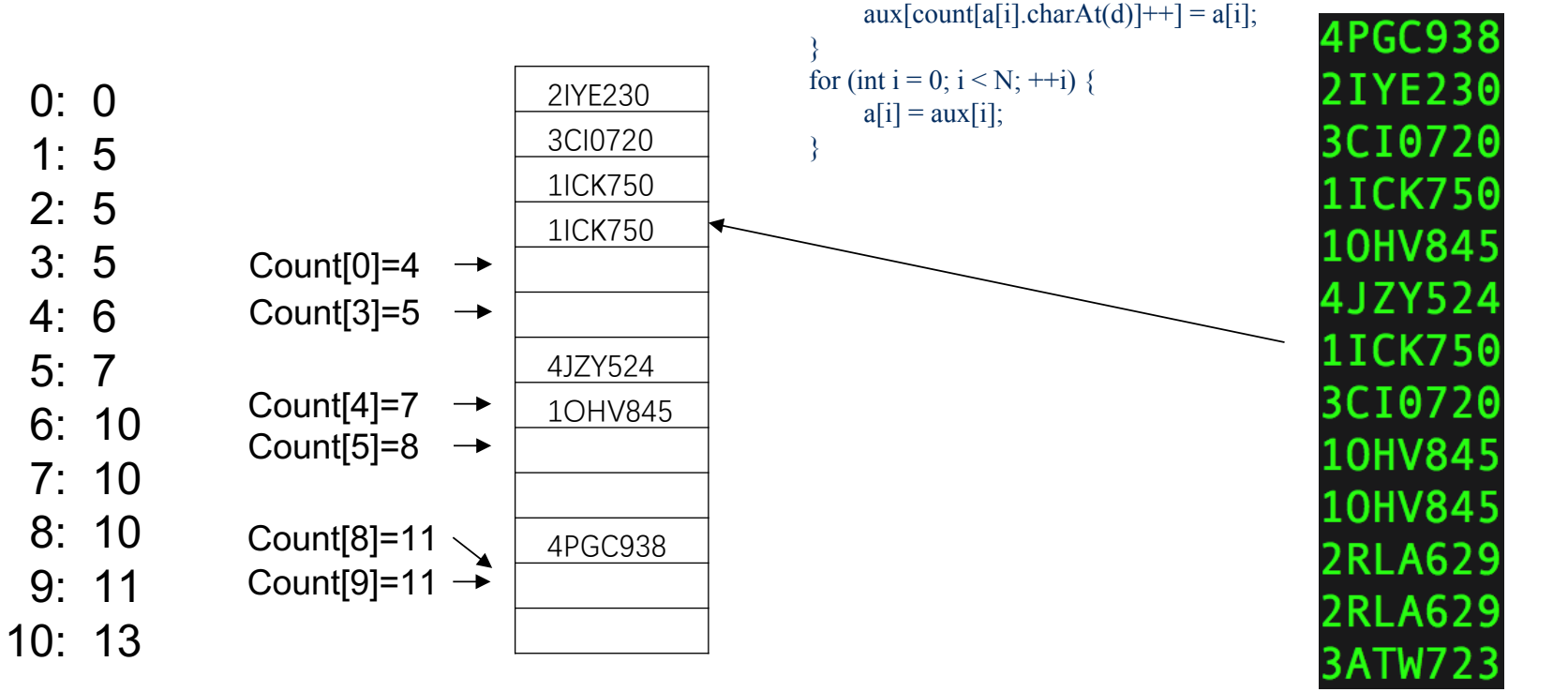
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

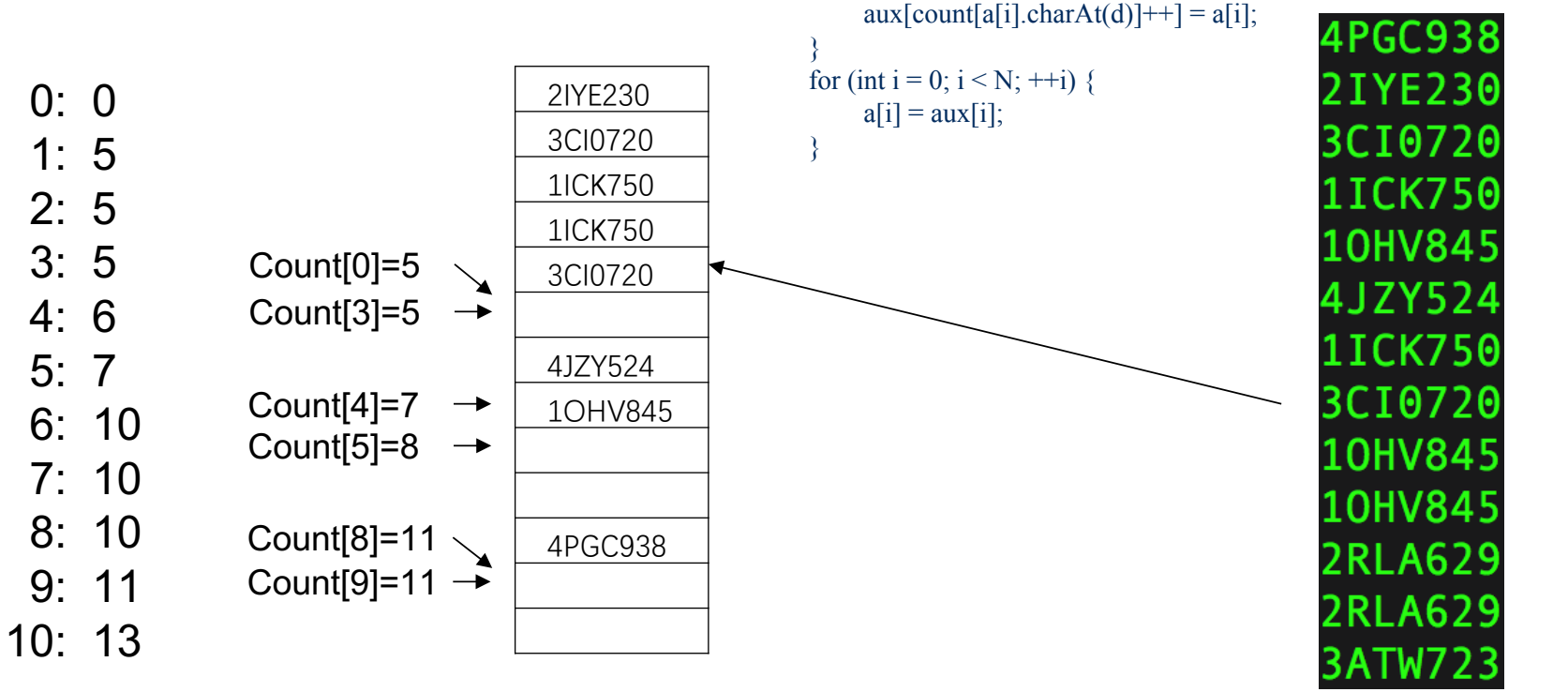
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

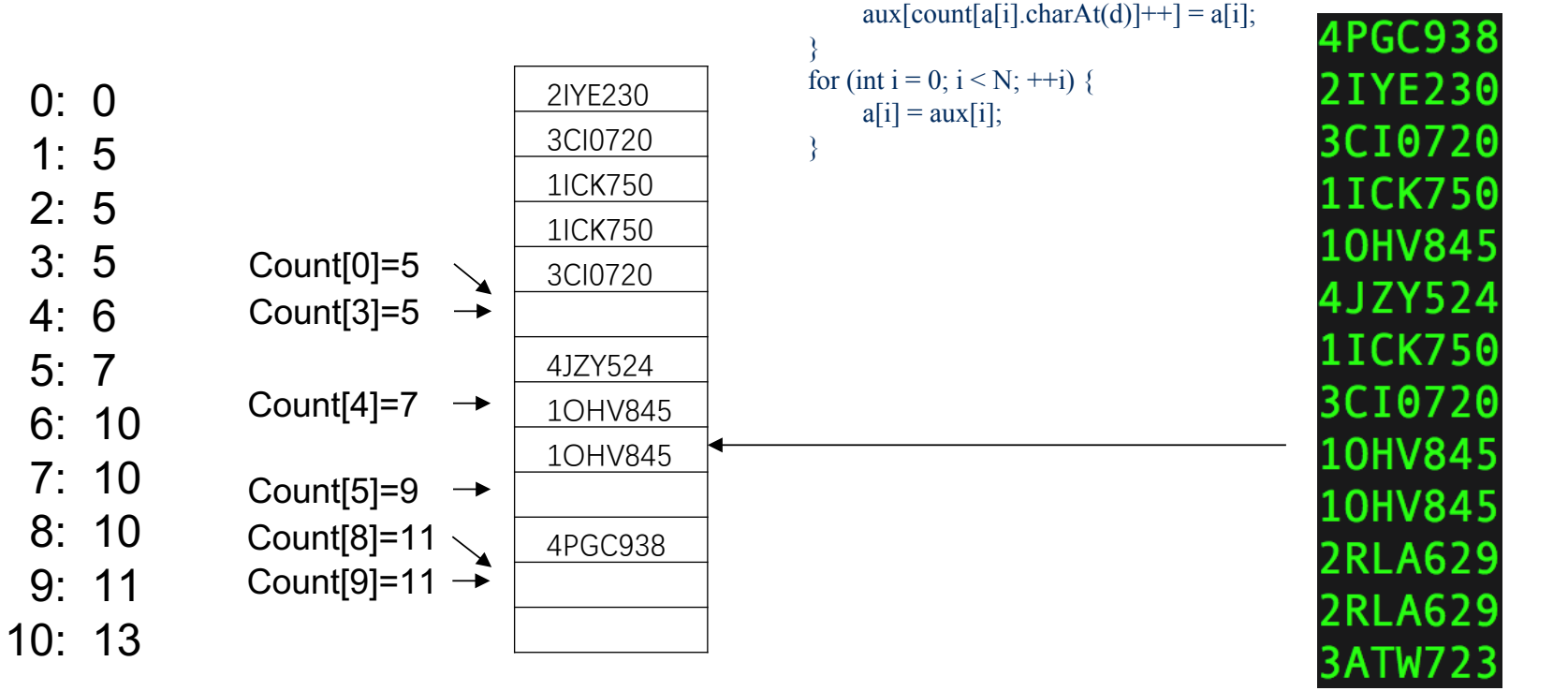
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

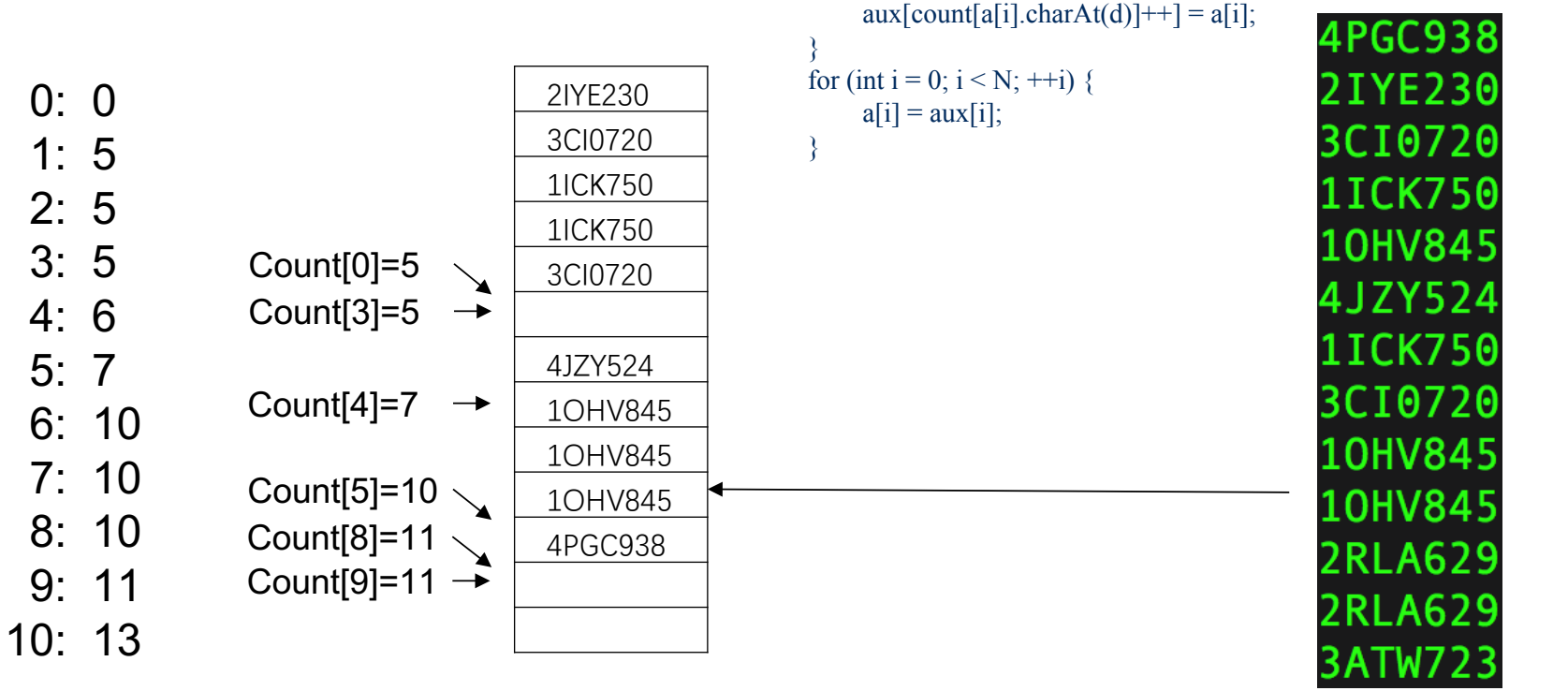
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

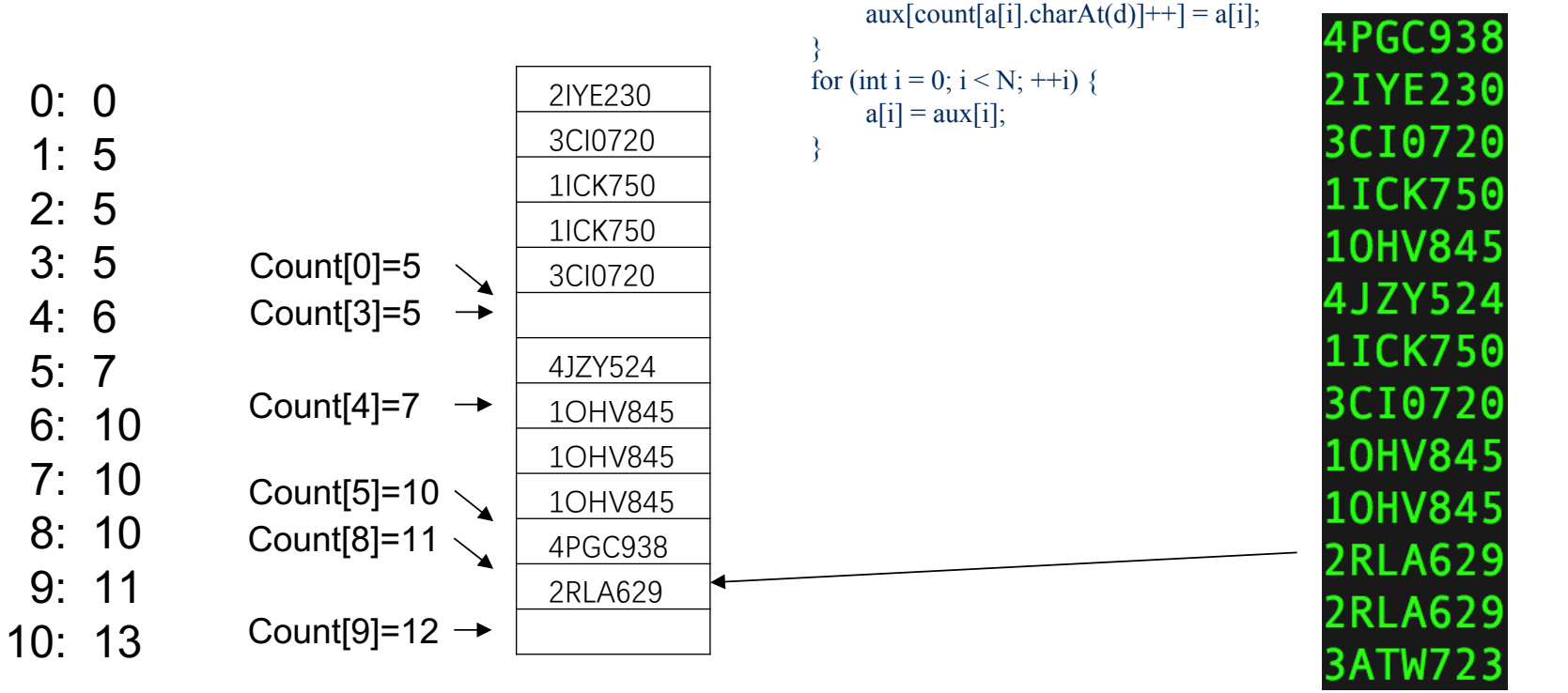
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

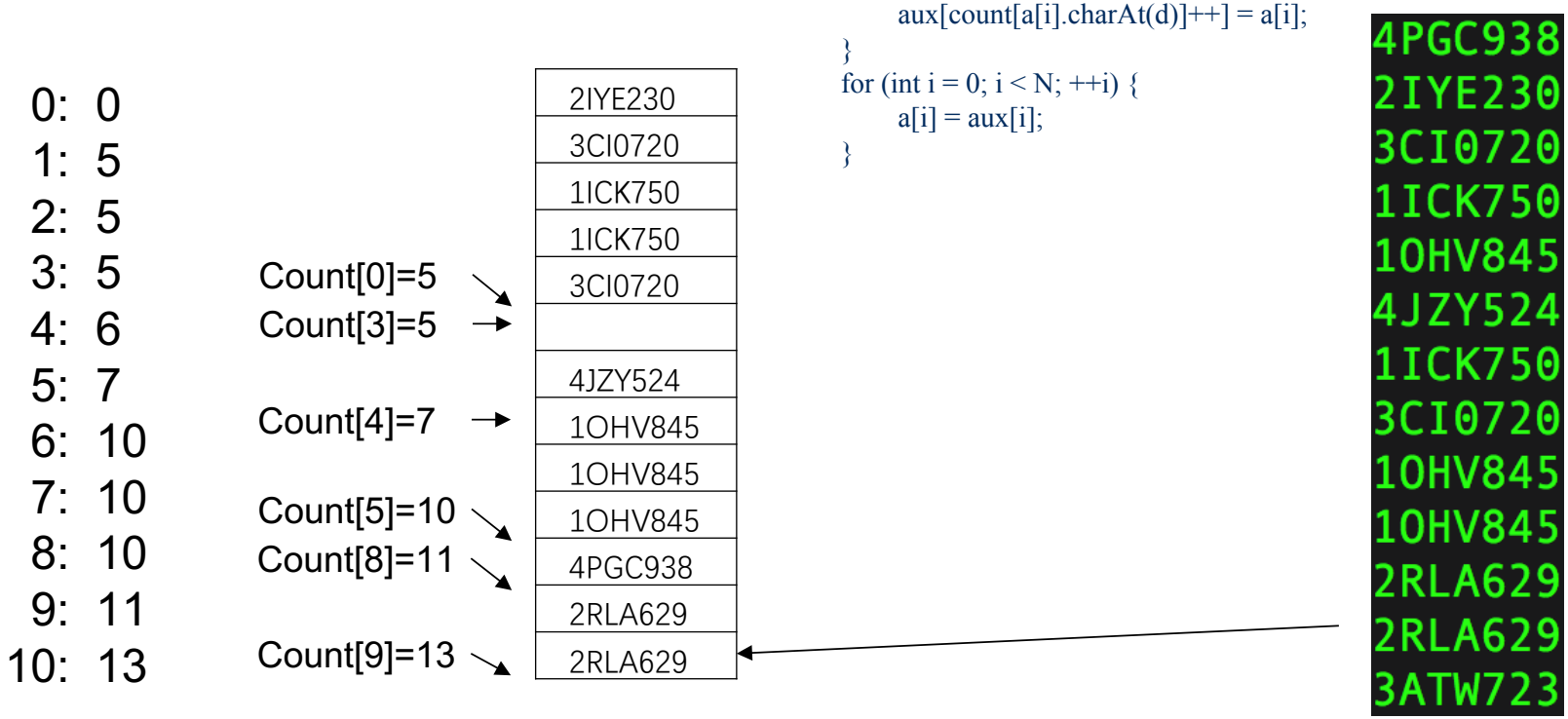
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

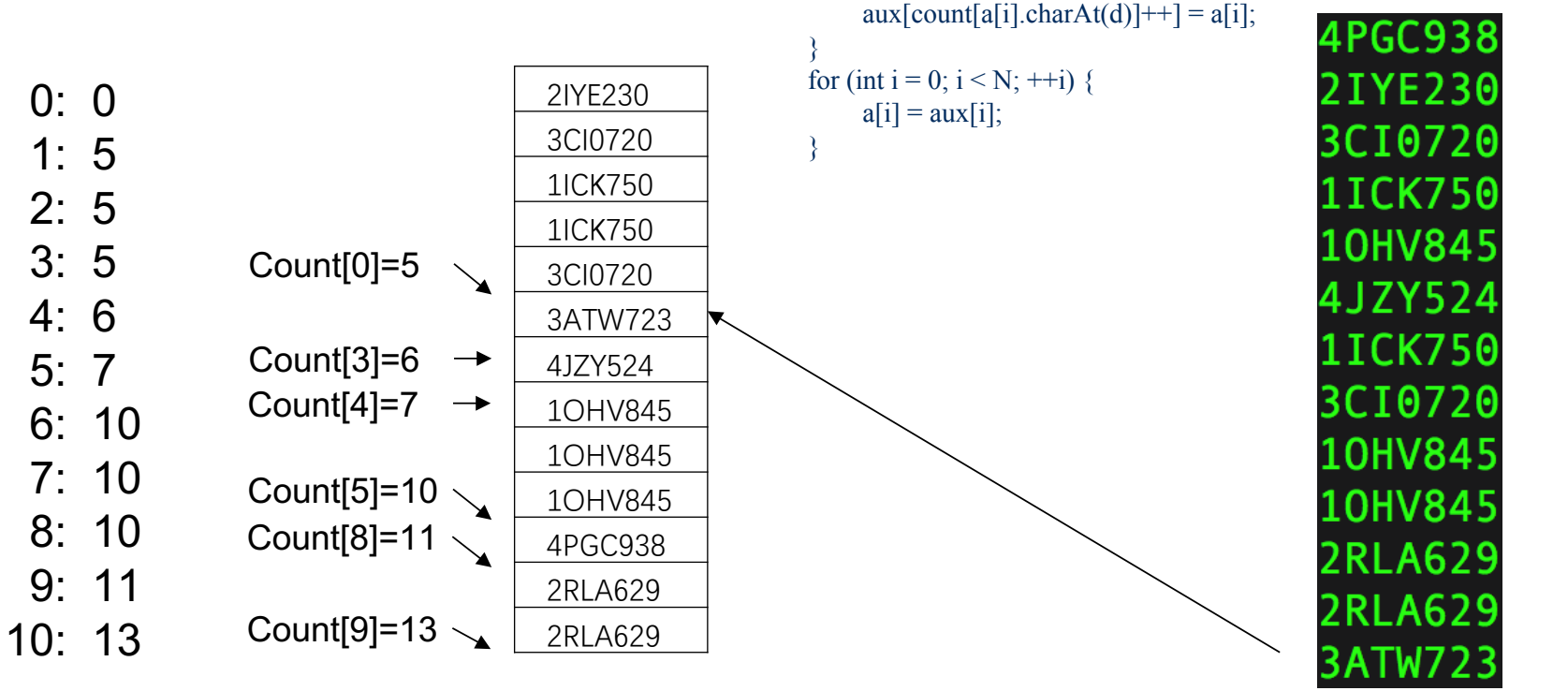
```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

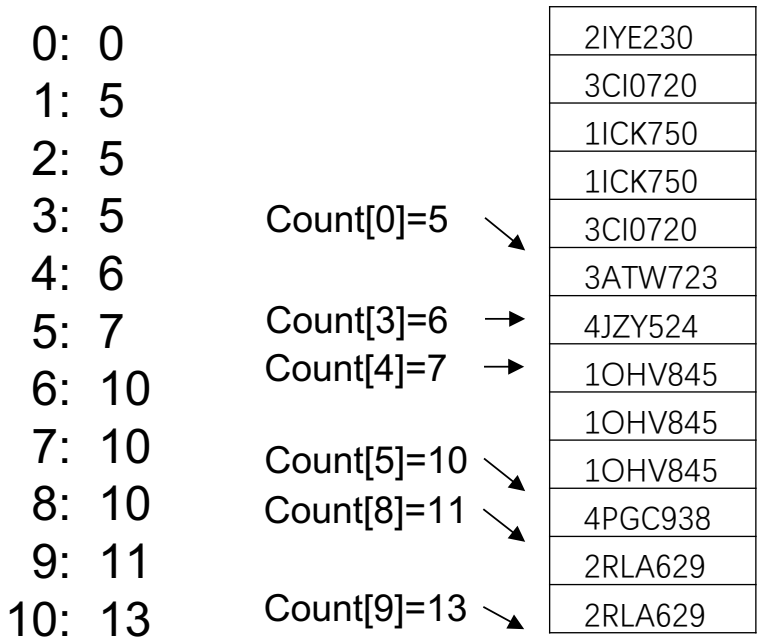
- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***

```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```



Key-indexed counting

- ***Ascii charset: 256 entries***
  - ***Digits charset as an example***
  - ***Sort by the right-most digit***
- ***Step 3: copy to output array***




```
int [] count = new int[R + 1];
// step1: calculate the histogram of key frequencies
for (int i = 0; i < N; ++i) {
      count[a[i].charAt(d) + 1]++;
}
// step 2: calculate the starting index for each key
for (int r = 0; r < R; ++r) {
      count[r + 1] += count[r];
}
// step 3: copy to output array, preserving order of with equal keys:
for (int i = 0; i < N; ++i) {
      aux[count[a[i].charAt(d)]++] = a[i];
}
for (int i = 0; i < N; ++i) {
      a[i] = aux[i];
}
```

```
4PGC938
2IYE230
3CI0720
1ICK750
1OHV845
4JZY524
1ICK750
3CI0720
1OHV845
1OHV845
2RLA629
2RLA629
3ATW723
```

Radix sort

```
public static void sort(String [] a, int W) {
        int N = a.length;
        int R = 256;
        String [] aux = new String[N];
        for (int d = W - 1; d >= 0; --d) {
            aux = sorted array a by the d^th character
            a = aux
        }
}
```

Radix sort

```
public static void sort(String [] a, int W) {
        int N = a.length;
        int R = 256;
        String [] aux = new String[N];

        for (int d = W - 1; d >= 0; --d) {
            int [] count = new int[R + 1];
            for (int i = 0; i < N; ++i) {
                count[a[i].charAt(d) + 1]++;
            }
            for (int r = 0; r < R; ++r) {
                count[r + 1] += count[r];
            }
            for (int i = 0; i < N; ++i) {
                aux[count[a[i].charAt(d)]++] = a[i];
            }
            for (int i = 0; i < N; ++i) {
                a[i] = aux[i];
            }
        }
    }
```

Radix sort analysis

- Runtime:
  - Worst case:
    - O(nk)
      - Where k is the length of the strings
- In-place?
  - No
- Stable?
  - Yes
- Limitations?

Qustions?