# **Sorting in Linear Time**

### **Counting Sort**

- Assumes that each of the n input elements is an integer in the range 0 to k, for some integer k.
- When k = O(n), the sort runs in  $\Theta(n)$  time.
- The input is an array A[1....n], and thus A.length = n
- Two other arrays are required:
  - The array B[1...n] holds the sorted output.
  - the array C[0...k] provides temporary working storage.
- An important property of counting sort is that it is stable: numbers with the same value appear in the same order.

# Pseudocode & Example

let 
$$C[0..k]$$
 be a new array

for  $i = 0$  to  $k$ 
 $C[i] = 0$ 

for  $j = 1$  to  $A.length$ 
 $C[A[j]] = C[A[j]] + 1$ 

//  $C[i]$  now contains the number of elements equal to  $i$ .

for  $i = 1$  to  $k$ 
 $C[i] = C[i] + C[i - 1]$ 

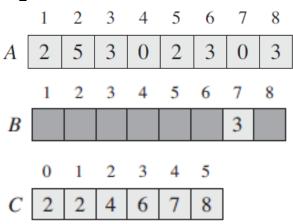
//  $C[i]$  now contains the number of elements less than or equal to  $i$ .

for  $j = A.length$  downto 1

 $C[A[j]] = C[A[j]] - 1$ 
 $C[A[j]] = C[A[j]] - 1$ 

# Pseudocode & Example

```
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A. length
    C[A[j]] = C[A[j]] + 1
```



// C[i] now contains the number of elements equal to i.

for 
$$i = 1$$
 to  $k$   
 $C[i] = C[i] + C[i-1]$ 

// C[i] now contains the number of elements less than or equal to i.

for 
$$j = A$$
.length downto 1  
 $B[C[A[j]]] = A[j]$ 

$$C[A[j]] = C[A[j]] - 1$$



# Pseudocode & Example

```
3
                                                       0
                                                                0
let C[0..k] be a new array
for i = 0 to k
                                                            3
    C[i] = 0
for j = 1 to A. length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i-1]
// C[i] now contains the number of elements less than or equal to i
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

3 4 5 6 7 8

# **Counting Sort Analysis**

- Two **for** loops take  $\Theta(k)$  time.
- And two **for** loops take  $\Theta(n)$  time.
- Thus the overall time is  $\Theta(k + n)$ .
- In practice, we usually use counting sort when we have k = O(n), in which case the running time is  $\Theta(n)$ .

### **Radix Sort**

### Assumption:

input taken from large set of numbers

### Basic idea:

 Sort the input on the basis of digits starting from unit's place.

#### Pro's:

- Fast
- Asymptotically fast O(d(n+k))
- Simple to code

### Con's:

Doesn't sort in place.

### **Radix Sort**

In input array A, each element is a number of d digits.

**for** i = 1 **to** d

use a stable sort to sort array A on digit i

329		720		720		329
457		355		329		355
657		436		436		436
839	ապիտ	457	jjj)	839	ումիթ.	457
436		657		355		657
720		329		457		720
355		839		657		839

# **Radix Sort Analysis**

• **Lemma 8.3:** Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in  $\Theta(d(k + n))$  time if the stable sort it uses takes  $\Theta(k + n)$  time.

#### Proof:

- The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm.
- When each digit is in the range 0 to k-1 and k is not too large, counting sort is the obvious choice.
- Each pass over n d-digit numbers then takes time  $\Theta(k + n)$ .
- There are d passes, and so the total time for radix sort is  $\Theta(d(k + n))$ .