# Data Structures – LECTURE 5

## Linear-time sorting

- Can we do better than comparison sorting?
- Linear-time sorting algorithms:
  – Counting-Sort
  – Radix-Sort
  – Bucket-sort



## Linear time sorting

- With more information (or assumptions) about the input, we can do better than comparison sorting. Consider sorting integers.
- Additional information/assumption:
  – Integer numbers in the range [0..*k*] where *k* = *O*(*n*).
  – Real numbers in the range [0,1) distributed <u>uniformly</u>
- Three algorithms:
  – Counting-Sort
  – Radix-Sort
  – Bucket-Sort



## Counting sort

<u>Input</u>: *n* integer numbers in the range [0..*k*] where *k* is an integer and *k* = *O*(*n*).

<u>The idea</u>: determine for each input element *x* its *rank:* the *number of elements less* than *x.*

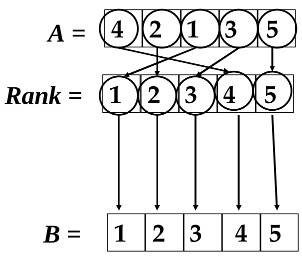
Once we know the rank *r* of *x*, we can place it in position *r*+1

<u>Example</u>: if there are 6 elements smaller than 17, then we can place 17 in the 7th position.

<u>Repetitions</u>: when there are several elements with the same value, locate them one after the other in the order in which they appear in the input → this is called **stable sorting**,



## Counting sort: intuition (1)

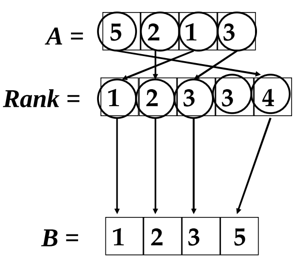



For each *A*[*i*], count the number of elements ≤ to it. This rank of A[*i*] is the index indicating where it goes

When there are no repetitions and *n* = *k*, *Rank*[*A*[*i*]] = *A*[*i*] and *B*[*Rank*[*A*[*i*]] ← *A*[*i*]



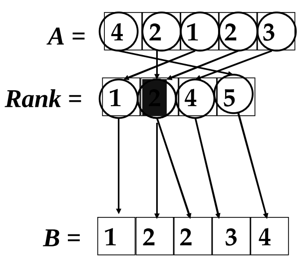
## Counting sort: intuition (2)




When there are no repetitions and *n* < *k*, some cells are unused, but the indexing still works.



## Counting sort: intuition (3)




When *n* > *k* or there are repetitions, place them one after the other in the order in which they appear in the input and adjust the index by one → this is called **stable sorting**

# Counting sort

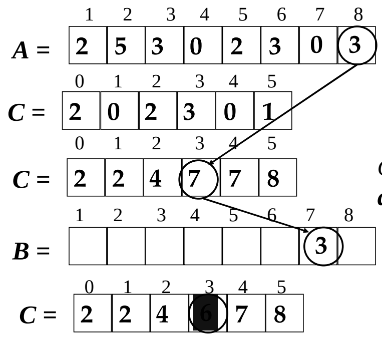
**Counting-Sort**(*A*, *B*, *k*)

*A*[*1..n*] *is the input array*
*B* [*1..n*] *is the output array*
*C* [*0..k*] *is a counting array*

```
1.  for i ← 0 to k
2.     do C[i] ← 0
3.  for j ← 1 to length[A]
4.     do C[A[j]] ← C[A[j]] +1
5.  /* now C contains the number of elements equal to i
6.  for i ← 1 to k
7.     do C[i] ← C[i] + C[i –1]
8.  /* now C contains the number of elements ≤ to i
9.  for j ← length[A] downto 1
10.    do B[C[A[j]]] ← A[j]        /* place element
11.       C[A[j]] ← C[A[j]] – 1    /* reduce by one
```



# Counting sort example (1)




***n* = 8**
***k* = 6**

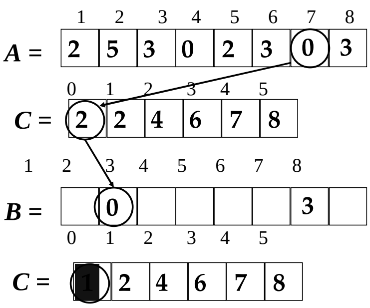
*C*[*A*[*j*]] ← *C*[*A*[*j*]] +1
***after line 4***

*C*[*i*] ← *C*[*i*] + *C*[*i* –1]
***after line 7***

*B*[*C*[*A*[*j*]]] ← *A*[*j*]
*C*[*A*[*j*]] ← *C*[*A*[*j*]] – 1
***after line 11***

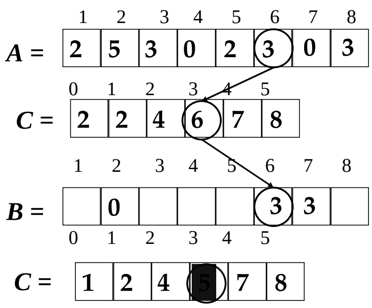


# Counting sort example (2)






# Counting sort example (3)






# Counting sort: complexity analysis

- **for** loop in lines 1—2 takes Θ(*k*)
- **for** loop in lines 3—4 takes Θ(*n*)
- **for** loop in lines 6—7 takes Θ(*k*)
- **for** loop in lines 9 –11 takes Θ(*n*)
- Total time is thus Θ(*n*+*k*)
- Since *k* = *O*(*n*), *T*(n) = Θ(*n*) and *S*(n) = Θ(*n*) → the algorithm is optimal!!
- This <u>does not work</u> if we do not assume *k* = *O*(*n*). Wasteful if *k* >> *n* and is not sorting in place.



# Radix sort

<u>Input</u>: *n* integer numbers with *d* digits

<u>The idea</u>: look at <u>one digit at a time</u> and sort the numbers according to this digit only. Start from the *least* significant digit, working up to the *most* significant one. Since there are only 10 different digits 0..9, there are only 10 places used for each column.

For example, we can use Counting-Sort for each call, with *k* = 9. In general, *k* << *n*, so *k* = O(*n*).

At the end, the numbers will be sorted!!

# Radix sort: example

| | | | |
|---|---|---|---|
| 329 | 720 | 720 | 329 |
| 457 | 355 | 329 | 355 |
| 657 | 436 | 436 | 436 |
| 839 | 457 | 839 | 457 |
| 436 | 657 | 355 | 657 |
| 720 | 329 | 457 | 720 |
| 355 | 839 | 657 | 839 |



# Radix-Sort

**Radix-Sort**(*A*, *d*)

1. **for** $i \leftarrow 1$ to $d$
2. **do** use a stable sort to sort array *A* on digit *d*

Notes:

- Complexity: $T(n) = \Theta(d(n+k)) \rightarrow \Theta(n)$ for constant $d$ and $k = O(1)$
- Every digit is in the range $[0..k-1]$ and $k = O(1)$
- The sorting MUST be a stable sort, otherwise it fails!
- This algorithm was invented to sort computer punched cards!



# Proof of correctness of Radix-Sort (1)

*We want to prove that Radix-Sort is a correct stable sorting algorithm*

Proof: by induction on the number of digits *d*.

Let $x$ be a $d$-digit number. Define $x_l$ as the number formed by the last $l$ digits of $x$, for $l \leq d$.
For example, $x = 2345$ then $x_1 = 5$, $x_2 = 45$, $x_3 = 345$…

Base: for $d = 1$, Radix-Sort uses a stable sorting algorithm to sort $n$ numbers in the range [0..9]. So if $x_1 < y_1$, $x$ will appear before $y$. When $x_1 = y_1$, the positions of $x$ and $y$ will not be changed since stable sorting was used.



# Proof of correctness of Radix-Sort (2)

General case: assume Radix sorting is correct after $i-1 < d$ passes, the numbers $x_{i-1}$ are sorted in stable sort order

Assume $x_i < y_i$. There are two cases:

1. The $i^{th}$ digit of $x$ < $i^{th}$ digit of $y$
   Radix-Sort will put $x$ before $y$, so it is OK.
2. The $i^{th}$ digit of $x$ = $i^{th}$ digit of $y$
   By the induction hypothesis, $x_{i-1} < y_{i-1}$, so $x$ appears before $y$ before the iteration and since the $i^{th}$ digits are the same, their order will not change in the new iteration, so they will remain in the same order.



# Proof of correctness of Radix-Sort (3)

Assume now $x_i = y_i$.

All the digits that have been sorted are the same.

By induction, x and y remain in the same order they appeared before the ith iteration, and snde the ith iteration is stable, they will remain so after the additional iteration.

This completes the proof!



# Properties of Radix-Sort

- Given $n$ $b$-bit numbers and a number $r \leq b$. Radix-Sort will take $\Theta((b/r)(n+2^r))$
- Take $d = b/r$ digits of $r$ bits each in the range $[0..2^r-1]$, so we can use Counting-Sort with $k = 2^r-1$. Each pass of Counting-Sort takes $\Theta(n+k)$ so we get $\Theta(n+2^r)$ and there are $d$ passes, so the total running time is $\Theta(d(n+2^r))$, or $\Theta((b/r)(n+2^r))$.
- For given values of $n$ and $b$, we can choose $r \leq b$ to be optimum $\rightarrow$ minimize $\Theta((b/r)(n+2^r))$.
- Choose $r = \lg n$ to get $\Theta(n)$.

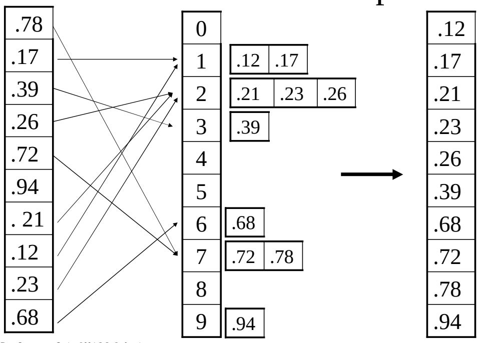
# Bucket sort

Input: *n* real numbers in the interval [0..1) uniformly distributed (numbers have equal probability)

The idea: Divide the interval [0..1) into *n* buckets 0, 1/*n*, 2/*n*. … (*n*–1)/*n*. Put each element $a_i$ into its matching bucket $1/i \le a_i \le 1/(i+1)$. Since the numbers are uniformly distributed, not too many elements will be placed in each bucket. If we insert them in order (using Insertion-Sort), the buckets and the elements in them will always be in sorted order.



# Bucket sort: example






# Bucket-Sort

**Bucket-Sort**(A)

1. *n* ← length(*A*)
2. **for** *i* ← 0 to *n*
3. **do** insert *A*[*i*] into list *B*[floor(*nA*[*i*])]
4. **for** *i* ← 0 to *n* –1
5. **do** Insertion-Sort(*B*[*i*])
6. Concatenate lists *B*[*0*], *B*[*1*], … *B*[*n* –*1*] in order

*A*[*i*] *is the input array*
*B*[*0*], *B*[*1*], … *B*[*n –1*]
*are the bucket lists*



# Bucket-Sort: complexity analysis

- All lines except line 5 (Insertion-Sort) take *O*(*n*) in the worst case.
- In the worst case, *O*(*n*) numbers will end up in the same bucket, so in the worst case, it will take $O(n^2)$ time.
- However, in the *average case*, only a constant number of elements will fall in each bucket, so it will take *O*(*n*) (see proof in book).
- Extensions: use a different indexing scheme to distribute the numbers (hashing – later in the course!)



# Summary

With additional assumptions, we can sort *n* elements in optimal time and space Ω(*n*).