POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

Iterative Linear Sorting Algorithms

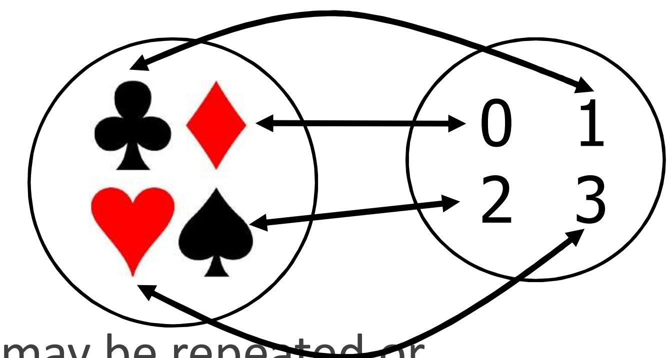
Paolo Camurati

General Features

- Find the position of an item not by comparison, rather by computation
- The worst-case asymptotic lower bound $T(N) = \Omega(N\log N)$ is no more true
- Complexity is linear $T(N) = O(N)$
- There are restriction on use
- Algorithms:
  - Counting sort
  - Radix sort
  - Bin/bucket sort: requires lists, topic dealt with in second year Course

Counting sort

- Goal: to sort an array of N integers in the range 0 …k-1
- Each finite set of k items may be matched with the integers in the range 0 … k-1



- Input data may be repeated or
- Input data may not contains some values in the range 0 … k-1

Approach

- Sorting by computation and not by comparison
- For each item x compute how many items precede it in the sorted array:
  - First compute simple occurrences of x, i.e. how many instances of x appear in the input
  - Staring from simple occurrences, compute multiple occurrences, i.e. how many items are ≤ x
- Walking the array from right to left, put item x in its final correct position

Data structures

Use 3 arrays:

- Input array: A[0..N-1] of N integers
- Result array: B [0..N-1] of N integers
- Simple/multiple occurrences array C of k integers if data belong to the range [0..k-1]

Example: N=8 k=6

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

Array to sort

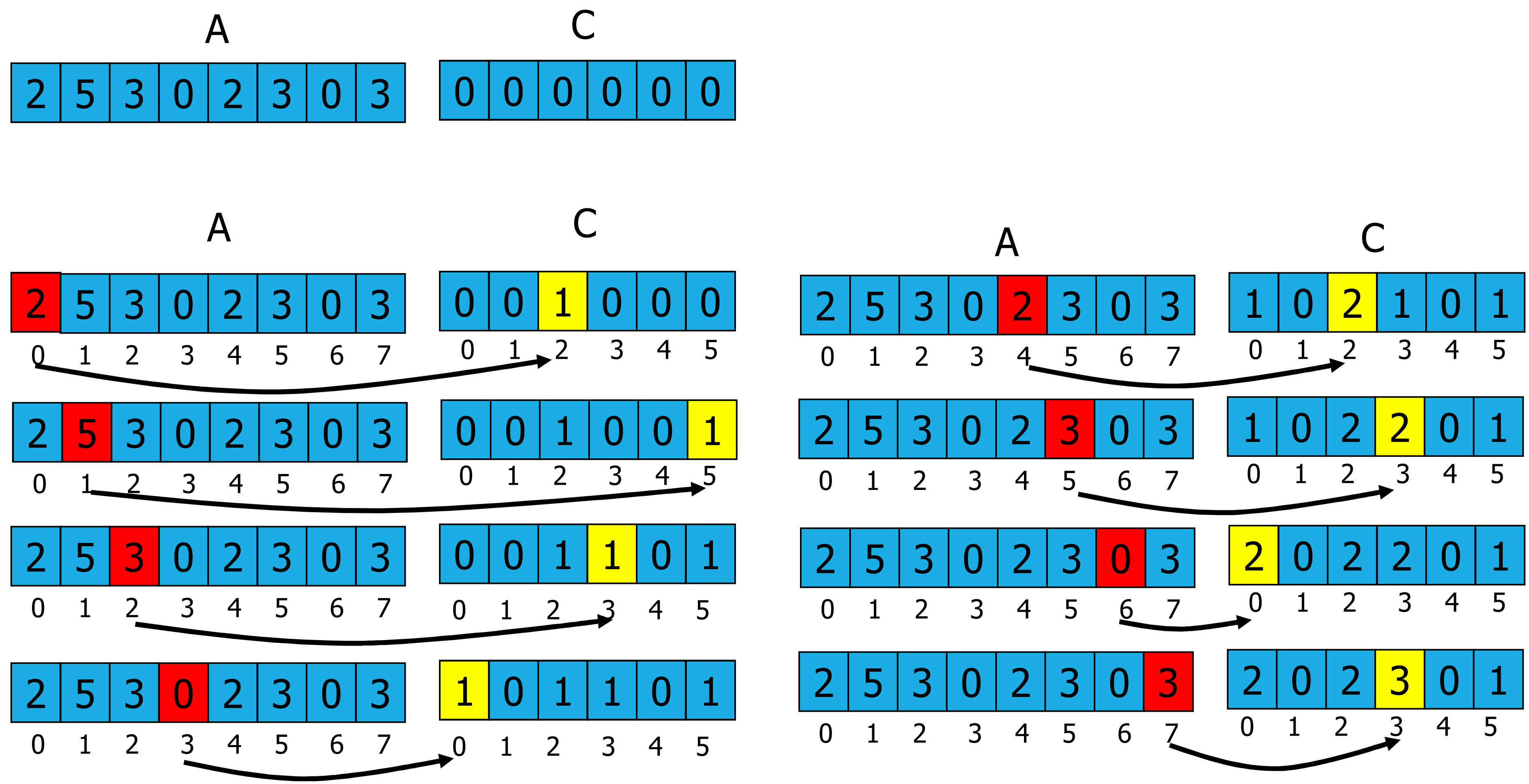
C						
	0	1	2	3	4	5

Simple/multiple occurrences array

Computing Simple Occurrences

- Initialize C to 0
- Scan input array A
  - A[i] is an occurrence of that value that belongs to the range 0… k-1
  - A[i] serves as index in C to increment by 1 the value of that cell

```
for (i = l; i <= r; i++)
  C[A[i]]++;
```


A
C
2 5 3 0 2 3 0 3
0 0 0 0 0 0
A
C
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
0 0 1 0 0 0
0 1 2 3 4 5
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
0 0 1 0 0 1
0 1 2 3 4 5
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
0 0 1 1 0 1
0 1 2 3 4 5
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
1 0 1 1 0 1
0 1 2 3 4 5
A
C
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
1 0 2 1 0 1
0 1 2 3 4 5
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
1 0 2 2 0 1
0 1 2 3 4 5
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
2 0 2 2 0 1
0 1 2 3 4 5
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
2 0 2 3 0 1
0 1 2 3 4 5

Computing Multiple Occurrences

- Scan simple occurrences array C
  - C[0] stores the number of occurrences of 0 and of all values that precede it (none by definition!)
  - C[i-1] stores the occurrences of the data that precede i ($1 \leq i < k$)
  - The occurrences of data that either precede or are equal to i are computed as C[i] = C[i-1] + C[i]

```
for (i = 1; i < k; i++)
  C[i] += C[i-1];
```

C

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

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

⇒

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

There are 2 occurrences of data ≤1

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

⇒

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

There are 4 occurrences of data ≤2

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

⇒

0	1	2	3	4	5
2	2	4	7	0	1

There are 7 occurrences of data ≤3

0	1	2	3	4	5
2	2	4	7	0	1

⇒

0	1	2	3	4	5
2	2	4	7	7	1

There are 7 occurrences of data ≤4

0	1	2	3	4	5
2	2	4	7	7	1

⇒

0	1	2	3	4	5
2	2	4	7	7	8

There are 8 occurrences of data ≤5

Computing Final Positions

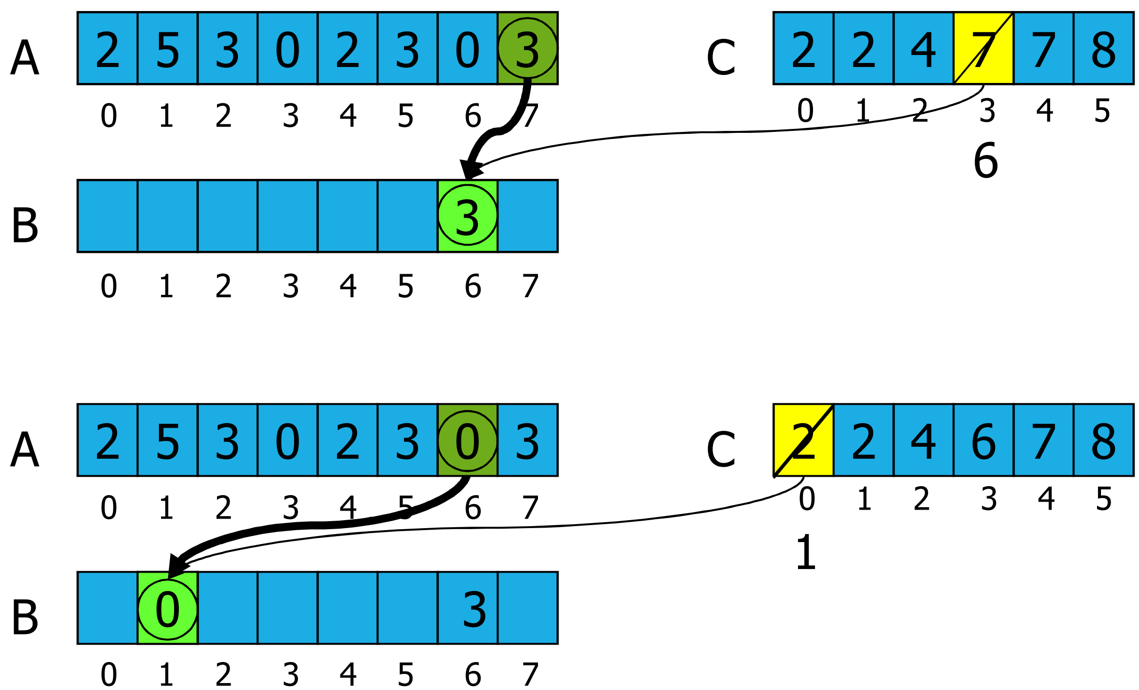
- Scan input array A from right to left
  - C[A[i]] stores the number of multiple occurrences of A[i] and of all the items that precede it
  - The final position in array B of A[i] is at index C[A[i]]-1. Why -1? Don't forget: in the C language array indices start from 0
  - Once A[i] is stored in its final position, update the multiple occurrences array C at index A[i] decrementing it by 1

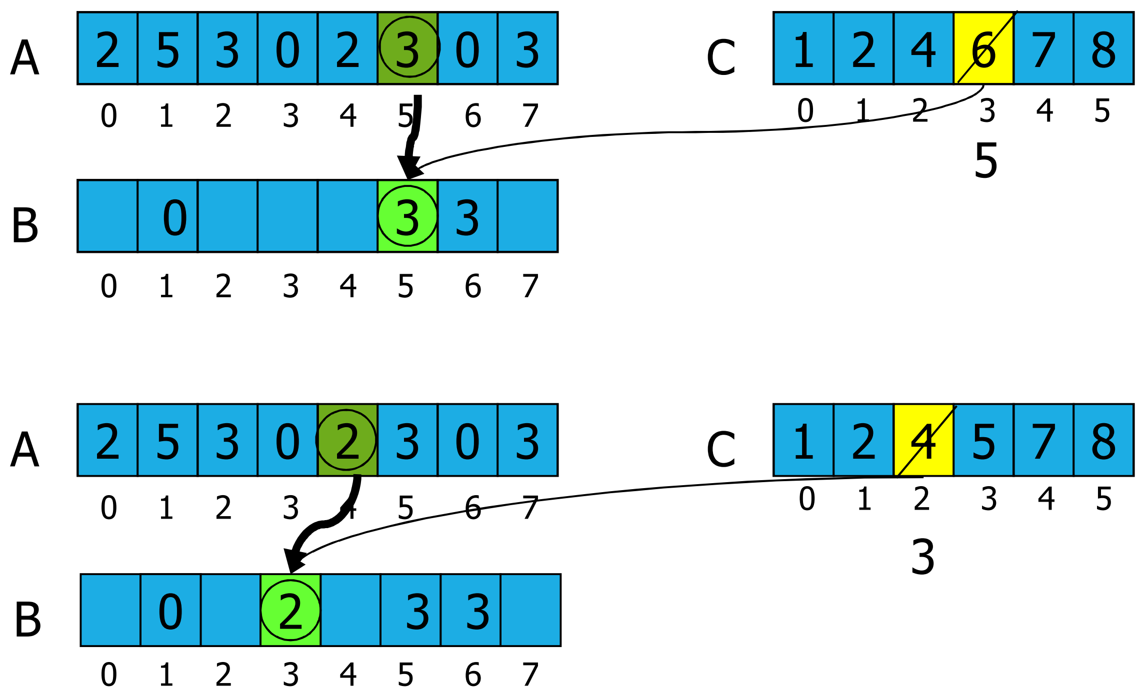
```
for (i = r; i >= l; i--) {
   B[C[A[i]]-1] = A[i];
   C[A[i]]--;
}
```

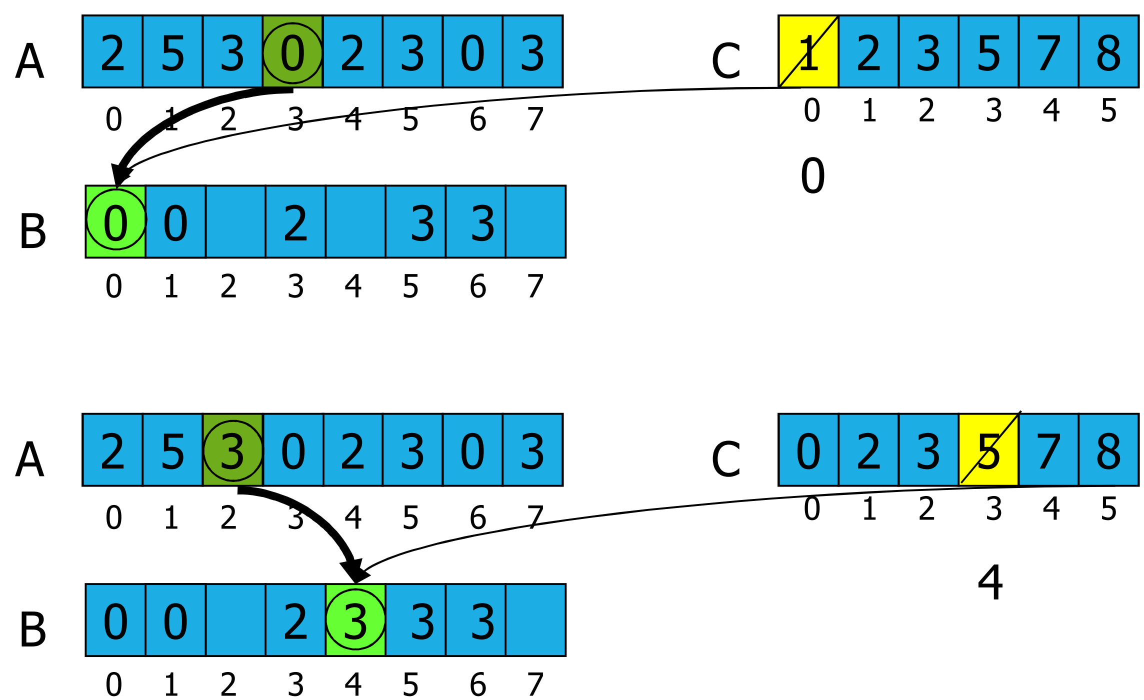
Example

how many values preeceed 3? 3 is the 7th so 6, but indicises start at 0 so 6th pos

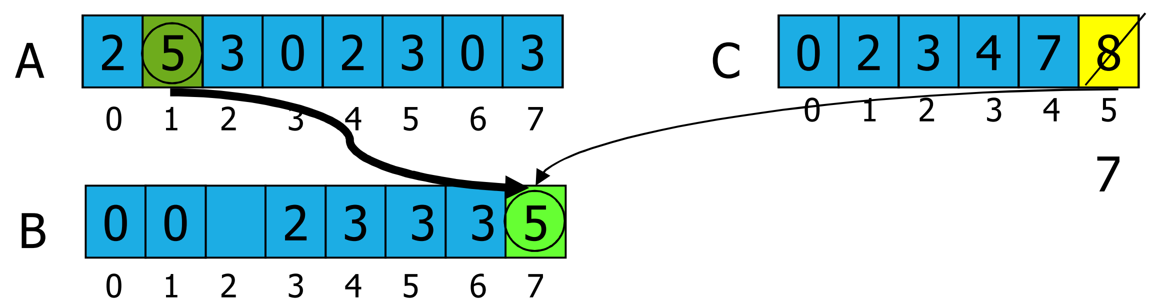
decrement the frequencies

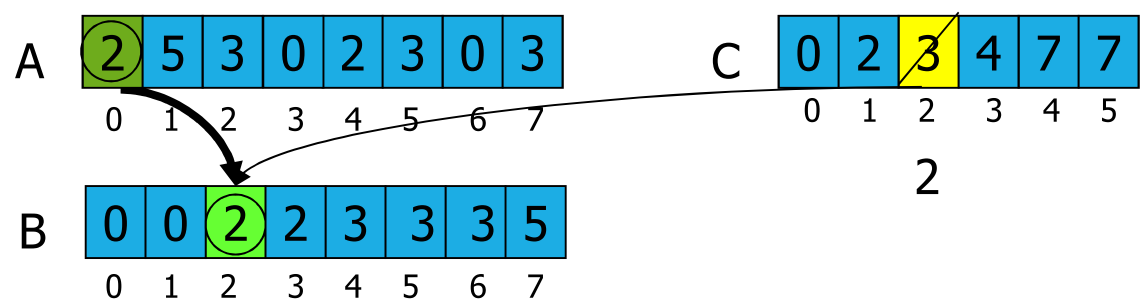



A
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
B
0 3 3
0 1 2 3 4 5 6 7
C
1 2 4 6 7 8
0 1 2 3 4 5
5
A
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
B
0 2 3 3
0 1 2 3 4 5 6 7
C
1 2 4 5 7 8
0 1 2 3 4 5
3



A
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
C
1 2 3 5 7 8
0 1 2 3 4 5
0
B
0 0 2 3 3
0 1 2 3 4 5 6 7
A
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
C
0 2 3 5 7 8
0 1 2 3 4 5
4
B
0 0 2 3 3 3
0 1 2 3 4 5 6 7


A
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
C
0 2 3 4 7 8
0 1 2 3 4 5
7
B
0 0 2 3 3 3 5
0 1 2 3 4 5 6 7


A
2 5 3 0 2 3 0 3
0 1 2 3 4 5 6 7
C
0 2 3 4 7 7
0 1 2 3 4 5
2
B
0 0 2 2 3 3 3 5
0 1 2 3 4 5 6 7

```
void CountingSort(int A[],int B[],int C[],int N,int k){
  int i, l=0, r=N-1;
  for (i = 0; i < k; i++)
    C[i] = 0;
  for (i = l; i <= r; i++)
    C[A[i]]++;
  for (i = 1; i < k; i++)
    C[i] += C[i-1];
  for (i = r; i >= l; i--) {
    B[C[A[i]]-1] = A[i];
    C[A[i]]--;
  }
  for (i = l; i <= r; i++)
    A[i] = B[i];
}
```

Arrays allocated in `main` and passed as parameters

Initialization of C

Simple occurrences

Multiple occurrences

Correct item positioning

Copy of result

Counting sort Features

- Non in place: arrays B and C are required, in addition to A
- Stable: stability is guaranteed by scanning array from right to left when finding the correct positions of the items: if there are duplicate keys, the last one is the first that is stored and its position is as rightmost as possible. No other duplicate key could ever «jump over», since the corresponding cell in the multiple occurrences array C is decremented
- Scanning from left to right doesn't guarantee stability. However it results in a sorted array.

Complexity Analysis of Counting sort

- Loop to initialize C: $\Theta(k)$
- Loop to compute simple occurrences: $\Theta(N)$
- Loop to compute multiple occurrences: $\Theta(k)$
- Loop to position item in B: $\Theta(N)$
- Loop to copy B in A: $\Theta(N)$

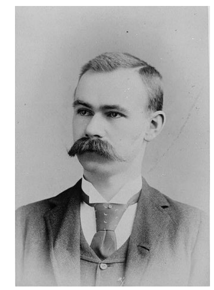
$$T(N) = \Theta(N+k).$$

If $k = \Theta(N)$, $T(N) = \Theta(N)$.

Applicability: k and N must be "reasonably" comparable in size. If $k=10^6$, N=3 and A= 999999, 1, 1000, it makes no sense to allocate an array of size $k=10^6$ to sort 3 items!

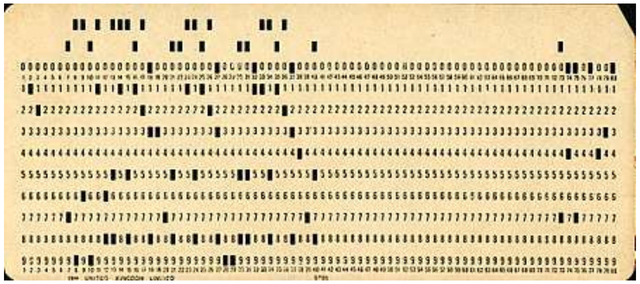
Radix sort

- 1890: first US «modern» census: large amounts of complex data
- Herman Hollerith introduces:
  - Punched cards to store information in binary form
  - «Tabulating machines» to mechanically sort data



Punched cards

- Stiff paper sheet organized in rows and columns
- Holes to indicate for a certain row/column the presence/absence of an information
- Features:
  - Information items in binary form
  - Information items on several fields



The «tabulating machine»

Electromechanical device able to

- «Read» punched cards
- Count information items depending on the presence/absence of a hole in a certain column

Hollerith’s Tabulating Machine Company becomes International Business Machines (IBM) in 1924.

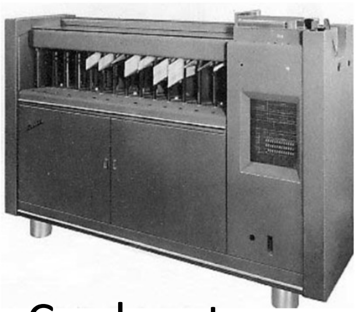
Punched Card Sorting (‘60)

- Starting from the rightmost column, a machine distributed cards into bins depending on the information stored in the column
- Cards in bins were picked up keeping the order (stability)
- Distribution in bins continued on the next column
- Termination: leftmost column processed.


Card puncher


Punched cards


Card reader


Card sorter

Radix sort

- Until now only «monolithic» item considered, like 1234, VCDF, etc.
- In Radix sort items consist of fields, whose values belong to a set of cardinality n

Example:
3-digit decimal numbers
d=3, n=10, values=0,1,2…9
329
457
657
839
436
720
355

Example:
Car plates: 3 fields: 2 letters 3 digits 2 letters
d=3,
letters n=22, values=A,…,Z no I, O, Q e U
digits n=10, values=0,1,2…9
FA 457 AA
GC 657 SD
AB 839 MN
ZZ 000 AA

Field-by-field «Intuitive» Sorting

- Sort according to leftmost column, then according to the next column to the right, until rightmost column is processed
- Intuitive for numbers, as they are represented according to a positional notation

329	329	329	720
457	355	720	355
657	436	436	436
839	457	355	457
436	657	457	657
720	720	657	329
355	859	859	859

Result is not sorted! To get a correct result, recursion is required. Recursion is a topic of the second year Course

Field-by-field «Counter-Intuitive» Sorting

- Sort according to rightmost column, then according to the next column to the left, until leftmost column is processed
- Counterintuitive for numbers, as it doesn't consider their positional notation

329	720	720	329
457	355	329	355
657	436	436	436
839	457	355	457
436	657	457	657
720	329	657	720
355	859	859	859

Result sorted!

- Constraint of the sorting algorithm used for each column: it must be STABLE!
- Counting sort is an excellent choice:
  - It is stable
  - It is applicable: the size k of array C is fixed and depends on the radix of the numbering system (hence the name Radix sort) of the digits that appear in each column. We may sort:
    - Base-10 numbers: k = 10
    - Strings of letters A…Z: C k = 26
    - Strings of ASCII characters: k = 128

Sorting integers (in base 10)

- Given n integers stored in array A and consisting of (non necessarily identical) number of digits
- Find the maximum number of digits d, left padding with 0s shorter numbers

170		0170
45		0045
2375		2375
90	With padding	0090
802		0802
24		0024
2		0002
66		0066

- Apply d steps of Counting sort starting from the rightmost column (weight 10^0) until the leftmost column (weight 10^{d-1}) is processed

```
void radixSort(int A[], int B[], int C[], int D[], int n) {
  int largest, d=1, i;
  largest = getMax(A, n);          Identify maximum of A

  while (largest/10 > 0){
    d++;                           Compute number of digits d
    largest /= 10;
  }
  for (i = 0; i < d; i++)          Iterate d times
    CountingSort(A, B, C, D, n , i);   Counting sort
}
```

Example

Step 0
Column of units

A after
Counting sort
on column
of units

A	D	A after
170	0	170
45	5	90
2375	5	802
90	0	2
802	2	24
24	4	45
2	2	2375
66	6	66

Step 1
Column of tens

A after
Counting sort
on column
of tens

A	D	A after Counting sort on column of tens
170	7	802
90	9	2
802	0	24
2	0	45
24	2	66
45	4	170
2375	7	2375
66	6	90

Step 2
Column of hundreds

A	D	A after Counting sort on column of hundreds
802	8	2
2	0	24
24	0	45
45	0	66
66	0	90
170	1	170
2375	3	2375
90	0	802

Step 3
Column of thousands

A	D	A after Counting sort on column of thousands
2	0	2
24	0	24
45	0	45
66	0	66
90	0	90
170	0	170
2375	2	802
802	0	2375

Identifying digits

Positional representation of numbers in base b:

- Digits in the range from 0 to b-1
  - in base 2 b=2, digits 0, 1
  - in base 10 b=10, digits 0,1,2,3,4,5,6,7,8,9
  - in base 8 b=8, digits 0,1,2,3,4,5,6,7
  - In base 16 b=16, digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

$x_{\text{(in base b, on d digits)}} = a_{d-1}b^{d-1} + a_{d-2}b^{d-2} + a_{d-3}b^{d-3} + \ldots\ldots\ a_1b^1 + a_0b^0$

Example

$12345_{\text{(in base 10, on 5 digits)}} = 1\cdot10^4 + 2\cdot10^3 + 3\cdot10^2 + 4\cdot10^1 + 5\cdot10^0$

- units: $(x / b^0) \% b$
- tens: $(x / b^1) \% b$
- hundreds: $(x / b^2) \% b$
- ....

()%b: remainder of the integer division of () by b

Example

$x = 12345_{(\text{in base } 10)}$

- units: $(x / b^0) \% b$ (12345/1) % 10 = 5
- tens: $(x / b^1) \% b$ (12345/10) % 10 = 1234 % 10 = 4
- hundreds: $(x / b^2) \% b$ (12345/100) % 10 = 123 % 10 = 3
- thousands: $(x / b^3) \% b$ (12345/1000) % 10 = 12 % 10 = 2
- tens of thousands: $(x / b^4) \% b$ (12345/10000) % 10 = 1 % 10 = 1

The auxiliary array D of n integers stores at each step the corresponding column and is is used by Counting sort to sort A

```
...
int i, ..., weight=1;
for (i=0; i < step; i++)
  weight *= 10;

for (i = l; i <= r; i++)
  D[i] =(A[i]/weight)%10;
...
```

```
void CountingSort(int A[],int B[],int C[],int D[], int N, int step){
  int i, l=0, r=N-1, weight=1;

  for (i=0; i < step; i++) weight *= 10;

  for (i = 0; i < 10; i++) C[i] = 0;

  for (i = l; i <= r; i++) D[i] =(A[i]/weight)%10;

  for (i = l; i <= r; i++) C[D[i]]++;

  for (i = 1; i < 10; i++) C[i] += C[i-1];

  for (i = r; i >= l; i--) {
    B[C[D[i]]-1] = A[i];
    C[D[i]]--;
  }
  for (i = l; i <= r; i++) A[i] = B[i];
}
```

compute 10^{step}

identify column

simple occorrences

multiple occurrences

sorting step

copy

Radix sort Features

- Not in place: arrays B, C and D used. D could be avoided recomputing its current item whenever necessary
- stable: stability is guaranteed by the use at each step of a stable algorithm like Counting sort.

Complexity Analysis of Radix sort

- Worst-case asymptotic complexity of Counting sort is $T(N) = \Theta(N+k)$, where items to sort are integers in the range (0… k-1)
- Run Counting sort d times
- Complexity is è $T(N) = \Theta(d(N+k))$.
- For numbers in base 10, k is fixed and is 10, thus

$$T(N) = \Theta(dN)$$

- If the number of digits d is fixed

$$T(N) = \Theta(N).$$