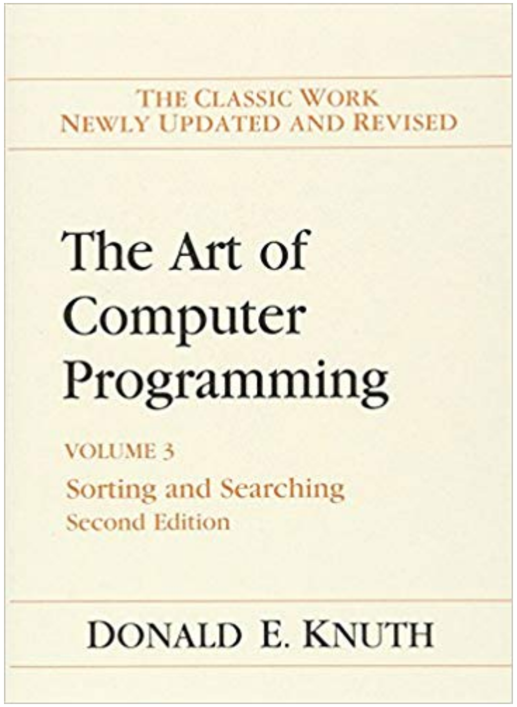
# Algoritmizace

## Třídění v lineárním čase



# Osnova

- Třídění počítáním (CountingSort)
- Přihrádkové třídění (BucketSort)
- Radixové třídění (RadixSort)

# Třídění v lineárním čase

Předpoklad: tříděné hodnoty (klíče) jsou přirozená čísla z množiny $\{0,1,\ldots,k\}$.

Třídění počítáním CountingSort

- pro každý prvek $x$ spočítej # prvků $\le x$
- zapiš $x$ na správnou pozici v setříděné posloupnosti na výstup

Vstup: pole `a` splňující předpoklad výše

Výstup: pole `b` obsahující prvky pole `a` uspořádané vzestupně

Pomocná datová struktura: pole `c[0..k]`

- pro uložení čítačů výskytů tříděných hodnot

# Třídění počítáním CountingSort

```
def countingSort0(a,k):
    c = [0] * (k+1)
    for x in a:
        c[x] += 1  # četnosti
    i = 0
    for x in range(k+1):
        for cetnost in range(c[x]):
            a[i] = x
            i += 1
    return a
```

✖ Nefunguje, jsou-li tříděné prvky strukturované

- klíč + satelitní data

# Třídění počítáním CountingSort

```
def countingSort(a,k,klic):
  c = [0]*(k+1)
  for x in a:
    c[klic(x)] += 1     # četnosti
  sum = 0
  for i in range(k+1): # kumulované č.
    c[i],sum = sum,c[i]+sum
  b = [0]*len(a)        # výstup
  for x in a:
    b[c[klic(x)]] = x
    c[klic(x)] += 1
  return b
```

Čas i prostor $\Theta(n+k)$

stabilní třídění
pořadí prvků
se stejným klíčem
se nemění

# Přihrádkové třídění BucketSort

Třídění rozdělováním do přihrádek

CountingSort

- `c[0..k]` pole počítadel
- `c[i]` četnost prvku s klíčem `i`

BucketSort

- `c[0..k]` pole přihrádek
- `c[i]` seznam prvků s klíčem `i`

# Přihrádkové třídění BucketSort

```
def bucketSort(a,k,klic):
    # prázdné přihrádky
    c = [[] for i in range(k+1)]
    for x in a:
        # vlož x do přihrádky
        c[klic(x)].append(x)
    b = []
    for prihradka in c:
        b += prihradka
    return b
```

Čas i prostor $\Theta(n+k)$

# Přihrádkové třídění – variace

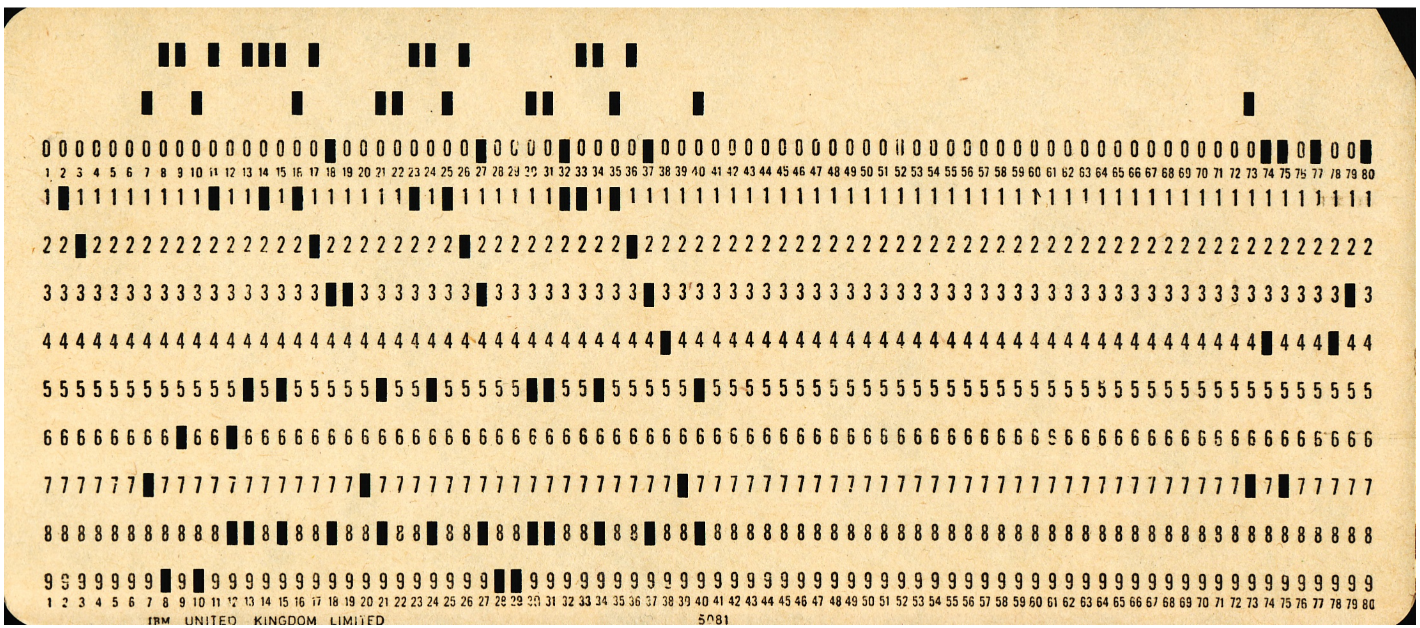
- Přihrádky `c[i]` lze implementovat jako spojové seznamy
- Co když jsou klíči desetinná čísla?
  - `klic(x)` $\in \langle 0, 1)$
  - pole `c[0..n-1]` kde `n` je délka vstupního pole `a`
  - vlož `x` do přihrádky (= na konec seznamu)
    `c[` $\lfloor$ `n`$\cdot$`klic(x)` $\rfloor$ `]`
  - každou přihrádku setřiď algoritmem InsertionSort
  - klíče rozmístěny rovnoměrně po $\langle 0, 1)$ $\Rightarrow$ čas $O(n)$

# Radixové třídění RadixSort

Vicecestné příhrádkové třídění

Číslicové třídění

Herman Hollerith (1887)




3 CLT
2 BKS
1 A-J
0 S-Z
11 J-R
12 A-I
R
IBM

# Radixové třídění RadixSort

| | | | |
|---|---|---|---|
| 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 |

```
def radixSort(a,k,d,klic):
    # klic(a[i]) je d-tice ∈{0,1,…,k}^d
    for i in reversed(range(d)):
        stabilním algoritmem setřiď
        pole a dle i-té položky klíče
```

# Radixové třídění – analýza

☝ **Invariant cyklu:** Po $i$-tém průchodu jsou prvky uspořádány dle posledních $i$ souřadnic klíče.

☞ korektnost algoritmu

☝ **Složitost:** RadixSort pracuje v čase $O(d(n+k))$, pokud využívá stabilní třídění pracující v čase $O(n+k)$.

- $d = O(1)$ a $k = O(n) \Rightarrow$ třídění v čase $O(n)$

# Radixové třídění – analýza

- Jak převést klíče na $d$-tice ?
  - klíč uložen v $b$ bitech
  - pro lib. $r \le b$ lze klíč rozdělit na $\lceil b/r \rceil$ $r$-bitových "cifer"
  - každá "cifra" $\in \{0,1,\ldots,2^r\text{-}1\}$ $\Rightarrow$ CountingSort pro $k = 2^r\text{-}1$
  - jedna iterace v čase $O(n+k) = O(n+2^r)$, celkem $d$ iterací
  - čas $O(d(n+2^r)) = O((b/r)(n+2^r))$
- Je-li dáno $n$ a $b$, jak zvolit $r$, $r \le b$ ?

$b < \lfloor \log_2 n \rfloor$

- $n+2^r = O(n)$
- zvolme $r = b$ $\Rightarrow$ časová složitost $O(n)$

$b \ge \lfloor \log_2 n \rfloor$

- zvolme $r = \lfloor \log_2 n \rfloor$ $\Rightarrow$ časová složitost $O(bn / \log n)$