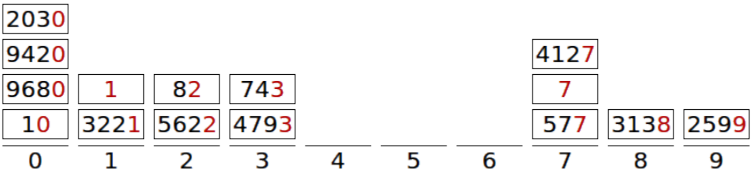
Sorting in Linear Time

Pedro Ribeiro

DCC/FCUP

2018/2019



Sorting with and without comparisons

- We have already talked about several algorithms for **sorting**
- They were based on **comparisons**: questions of the form "$a < b$?"
  - Quicksort
  - Mergesort
  - There are others such as Heapsort, ...
- We have shown that for these comparison-based algorithms we have a **lower bound** on the number of comparisons needed: $\Theta(n \log n)$
- Today we are going to talk about algorithms **not based on comparisons** (and thus not restricted to the previous lower bound)

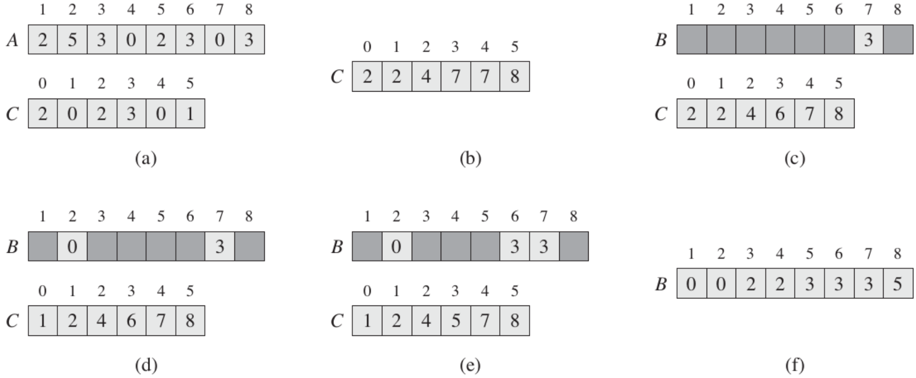
Counting Sort

- Assumption: integers to be sorted are in the range 0 to k

Counting Sort - Sort elements in A, with sorted output in B

```
CountingSort(A,B,k):
   C[0..k] ← new array
   for i = 0 to k
      C[i] = 0
   for i = 1 to A.length
      C[A[i]] = C[A[i]] + 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 i = A.length downto 1
      B[C[A[i]]] = A[i]
      C[A[i]] = C[A[i]] -1
```

Counting Sort



(a) A and C after first counting loop
(b) C after third loop
(c), **(d)** and **(e)** first iterations of last loop
(f) final state of array B
(figure from CLRS)

Counting Sort

What is the **time complexity**?

Let $n = A.length$

- First loop: $\Theta(k)$
- Second loop: $\Theta(n)$
- Third loop: $\Theta(k)$
- Fourth loop: $\Theta(n)$
- **Total:** $\Theta(n + k)$

If k is $O(n)$ then this sort runs in **linear time**! $\Theta(n)$

Stable sorting

Stable sorting

A sorting algorithm is said to be **stable** if in case of a tie it keeps the order of the original array

- This property may be important, for instance, when "satellite" data is carried around the elements being sorted
  - Ex: sorting persons - tuples (name, age) - only by age
- **Counting Sort is stable!** (because of the order of the last loop)
  - This is why it may be used as a subroutine in Radix Sort, as we'll see

Radix Sort

- Imagine you are sorting **"by columns"**, i.e., by considering at a time each possible digit position
- Intuitively, you would probably imagine starting with the **most significant digit**.
- **RadixSort** solves the sorting problem somehow "counter-intuitively", by starting with the **least significant digit**.

Radix Sort

- Assumption: each element in the array A has d digits, where 1 is the lowest order digit, and d the highest-order digit

Radix Sort

RadixSort(A,d):
  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	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

(figure from CLRS)

Radix Sort

- Radix Sort is akin to **sorting by different fields**
  - Ex: sort dates by year, then by month, then by day
    We could do a comparison-based sort: compare year, if tie, compare month, if tie compare day
    or
    do 3 consecutive stable sorts: first by day, then by month, and finally by year

- **Complexity** of Radix Sort depends on the used sorting algorithm

- An usual choice is... Counting Sort! In this case we have $\Theta(d(k+n))$

  - We can sort n b-bit numbers in $\Theta((b/r)(n+2^r))$ with counting sort
    - Each key has $\lceil b/r \rceil$ digits of r bits each (our d)
    - Each digit is an integer in the range 0 to $2^r - 1$ (our k)

Sorting Keys that are not numerical

- Counting and Radix sort **do not depend** on having numerical keys
- They depend on having the possibility to **"break" the key into "digits"** (parts) and to **map each part into a numerical value**
- For instance, if we have **strings**, we can break by chars, and map each char into a number ($a = 0, b = 1, \ldots, z = 25$)

```
EAR        SEA        EAR        BOX
COW        DOG        SEA        COW
DOG        BIG        DIG        DIG
NOW   ->   EAR   ->   DOG   ->   DOG
SEA        COW        COW        EAR
BOX        NOW        NOW        FOX
BIG        BOX        BOX        NOW
FOX        FOX        FOX        SEA
```

Bucket Sort

- Assumption: keys follow an uniform distribution over a certain range
- Idea: make n **equal sized "buckets"**, distribute input over buckets, sort each bucket and then concatenate results
- Rationale: If the input is uniformly distributed, than **there will not be many numbers in each bucket**!

Bucket Sort

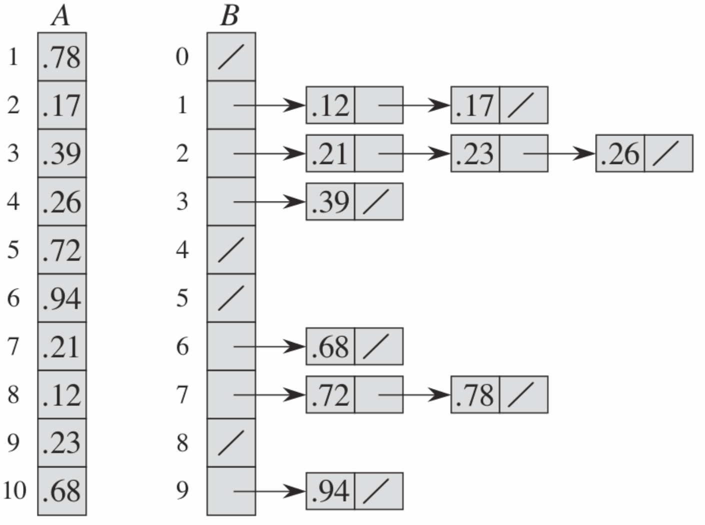
- Without loss of generality, let's assume for now that the **range is** 0..1

Bucket Sort - each bucket is a list

BucketSort(A):
  n = A.length
  B[0..n.1] ← new array of lists
  for i = 0 **to** n-1
    B[i] = empty list
  for i = 1 **to** n
    insert A[i] into B[$\lfloor nA[i] \rfloor$]
  for i = 0 **to** n-1
    sort list B[i]
  concatenate lists B[0], B[1], ..., B[n-1] together, in order

Bucket Sort

An example



(figure from CLRS)

Bucket Sort

What is the **execution time**?

- Everything except the cycle with sorts takes $\boldsymbol{\Theta(n)}$
- We need to analyse the **time taken by the sorts**
- Let's assume we use **Insertion Sort** (good for lists) - each is $O(n^2)$

Execution Time of Bucket Sort (with Insertion Sort)

n_i: random variable denoting the number of elements in bucket i

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Bucket Sort

We want to analyse the average-case running time of bucket sort: calculate the expected value of running time.

$$
\begin{aligned}
\mathbf{E}[T(n)] &= \mathbf{E}[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathbf{E}[O(n_i^2)] \quad \text{(by linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathbf{E}[n_i^2])
\end{aligned}
$$

What is the value of $\mathbf{E}[n_i^2]$? (expected size of each bucket i)

Bucket Sort

Claim: $\mathbf{E}[n_i^2] = 2 - 1/n$

Let's define an indicator random variable $\mathbf{X_{ij}}$ with values:

- **1** if $A[j]$ falls on bucket i, **0** otherwise

With this, $n_i = \sum_{j=1}^{n} X_{ij}$

$$\begin{aligned}
\mathbf{E}[n_i^2] &= \mathbf{E}[(\sum_{j=1}^{n} X_{ij})^2] \\
&= \mathbf{E}[\sum_{j=1}^{n} \sum_{k=1}^{n} X_{ij} X_{ik}] \\
&= \mathbf{E}[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n, k \neq j} X_{ij} X_{ik}] \\
&= \sum_{j=1}^{n} \mathbf{E}[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n, k \neq j} \mathbf{E}[X_{ij} X_{ik}]
\end{aligned}$$

Bucket Sort

Let's first solve $\mathbf{E}[X_{ij}^2]$. The probability of j being in bucket i is $1/n$, since the keys are uniformly distributed.

$\mathbf{E}[X_{ij}^2] = 1^2 \times \frac{1}{n} + 0^2 \times (1 - \frac{1}{n}) = \frac{1}{n}$

Now, when $k \neq j$, variables X_{ij} and X_{ik} are independent:

$\mathbf{E}[X_{ij} X_{ik}] = \mathbf{E}[X_{ij}]\,\mathbf{E}[X_{ik}] = \frac{1}{n} \times \frac{1}{n} = \frac{1}{n^2}$

Finally:

$$\begin{aligned}
\mathbf{E}[n_i^2] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \leq j \leq n} \; \sum_{1 \leq k \leq n, k \neq j} \frac{1}{n^2} \\
&= n \times \frac{1}{n} + n(n-1) \times \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n}
\end{aligned}$$

Bucket Sort

We want to analyse the average-case running time of bucket sort: calculate the expected value of running time.

$$\begin{aligned} \mathbf{E}[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathbf{E}[n_i^2]) \\ &= \Theta(n) + n \times O(2 - 1/n) \\ &= \Theta(n) \end{aligned}$$

Hence, Bucket Sort will have linear complexity on the average-case!

Even if the input does not follow a uniform distribution, bucket sort may still run in linear time, as long as the **sum of the squares of the bucket sizes is linear** in the total number of elements.