Sorting in Linear Time

Verónica Costa
veronica.costa@upf.edu

Esteban Maestre
maestre@eel.upc.es
**Background**

So far, the running time for sorting algorithms is

Comparison-based sorting algorithms

\[
\begin{align*}
\text{Merge Sort} & \rightarrow O(n \log n) \\
\text{Heap Sort} & \rightarrow O(n \log n) \\
\text{Insertion Sort} & \rightarrow O(n^2) \\
\text{Quick Sort} & \rightarrow O(n^2)
\end{align*}
\]

The only operation used in these algorithms is *comparison*, so a lower bound can be given...
Binary decision tree (1/2)

• Within any comparison sort, all comparisons have the form $a_i \geq a_j$
• *Binary Decision Trees* provide an abstraction of comparison sorts

In a full binary tree with $n!$ leaves:
• The root represents the first comparison
• Any internal node represents a comparison:
  – If the answer is YES continue with the left child
  – If the answer is NO continue with the right child
• A leaf represents a permutation
• Leaf height: is the length of the path from the leaf to the root
• Tree height: is the maximum height of one of its leaves
Binary decision tree (2/2)

- Tree paths are all possible execution traces, the longest path is the tree’s *heigth*
- The heigth of the tree is the *worst case* number of comparisons for a comparison-based sorting algorithm

The asymptotic heigth of any decision tree for sorting n elements is $\Omega(n \lg n)$

Not linear in time!!!
Let’s beat it...

**MAIN IDEA**
Sort without comparisons using memory locations (space is not at premium)
Counting sort (1/4)

- There is **no comparisons** between elements!
- **However**... it depends on assumption about the numbers being sorted
  - We assume numbers are in the range 0..k
- The algorithm:
  - Input: unsorted array $A[1..n]$, where $A[j] \in \{0, 1, 2, 3, ..., k\}$
  - Output: sorted array $B[1..n]$ (notice: not sorting in place)
  - Also: Array $C[0..k]$ for auxiliary storage
- Complexity: $O(n + k)$ operations
Counting sort (2/4)

ALGORITHM:

k is the greater integer in the input array
n is the input elements, range from the smallest integer from the input array to k
A [ 1...n ] is the input array
length [ A ] = n
B [ 1...n ] is the SORTED output array
C [ 0 ... k ]

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. C[i] contains numbers of elements equal to i

6. for i <-- 1 to k
7. do C[i] <-- C[i] + C[i - 1]

8. C[i] contains number of elements less than or equal to i

9. for j <-- length[A] downto 1

Takes time O ( k )

Takes time O ( n )
### Counting sort (3/4)

**EXAMPLE**

What we know:

- \( A[] = [2 5 3 0 2 3 0 3] \)
- Length \([A]\) = \( n = 8 \)
- \( k = 5 \)

1. for \( i \) <- 0 to \( k \)
2. \( \text{do } C[i] <- 0 \)

3. for \( j \) <- 1 to 8
4. \( \text{do } C[A[j]] <- C[A[j]] + 1 \)
5. \( C[i] \) contains # of elements equal to \( i \)

6. for \( i \) <- 1 to 5
7. \( \text{do } C[i] <- C[i] + C[i-1] \)
8. \( C[i] \) contains # of elements \( \leq \) than \( i \)

9. for \( j \) <- length \([A]\) downto 1
11. \( C[A[j]] <- C[A[j]] - 1 \)

**EXAMPLE 1.**

```plaintext
for i <- 0 to k
  do C[i] <- 0
for j <- 1 to 8
  do C[A[j]] <- C[A[j]] + 1
for i <- 1 to 5
  do C[i] <- C[i] + C[i-1]
for j <- length[A] downto 1
  do C[A[j]] <- C[A[j]] - 1
```
Counting sort (4/4)

- **RUNNING TIME:** $O(n + k)$
  - Usually, $k = O(n)$
  - Thus counting sort runs in $O(n)$ time
- This algorithm is *stable*
- *Counting sort* is not always used because it depends on the range $k$ of elements: $k$ could be too large to sort

Eg. 32 bit integers ($2^{32} = 4,294,967,296$)
Radix sort (1/6)

- Used for sorting $d$-digits numbers, sorting on each digit
- Fast when sorting integer numbers in a big range $0 \leq n_i < k$
- Intuitively, one might want to sort on the most significant digit, then the second msd, and so on...

**Problem:** lots of intermediate steps when some digits are the same

- Key IDEA: sort on the least significant digit first

**ALGORITHM:**
(sort n d-digits numbers)

Radix-Sort($A, d$)

1. for $i \leftarrow 1$ to $d$
2. do use a stable sort to sort array $A$ on digit $i$
Radix sort (2/6)

Will radix sort work?

- If two digits at position $i$ are different, ordering numbers by that digit is correct (lower-order digits irrelevant).
- If they are the same, numbers are already sorted on the lower-order digits. Since a stable sort is used, the numbers stay in the right order.

**EXAMPLE:**

(sort 10 4-digits numbers)

```
10 410 4

8567 4486 7534 456 7675 134 8546 1370 4108 5743 56 54

1370 5743 7534 134 54 7675 4486 456 8546 56 8567 4108

4108 7534 134 5743 8546 054 456 056 8567 1370 7675 4486

0054 0056 4108 0134 1370 0456 4486 7534 8546 8567 7675 5743

54 56 134 456 1370 4108 4486 5743 7534 7675 8546 8567
```
Radix sort (3/6)

**RUNNING TIME:**
(it depends on which sorting algorithm is used to sort on digits)

When *counting sort* is used, sorting *n* numbers holding values up to *k* takes time \(O(n + k)\)

- For *d*-digits numbers, the total time taken for radix-sort them is \(O(dn + dk) = O(n)\)

- More generally, given *n* *b*-bit numbers and any positive integer \(r \leq b\), each one can be viewed like a number with \(b/r\) digits of radix \(2^r\)
Radix sort (4/6)

**LEMMA:**

Given $n$ $b$-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\Theta \left( \left( \frac{b}{r} \right) \left( n + 2^r \right) \right)$ time.

- Each pass of counting sort takes time $\Theta \left( n + k \right) = \Theta \left( n + 2^r \right)$
- There are $d$ passes, therefore the time it takes is $\Theta \left( d \left( n + 2^r \right) \right)$
- Now $d = \left( \frac{b}{r} \right)$, then RADIX-SORT takes $\Theta \left( \left( \frac{b}{r} \right) \left( n + 2^r \right) \right)$ time

For given values $n$ and $b$, a value of $r$ must be chosen, with $r \leq b$, that minimizes the expression $\left( \frac{b}{r} \right) \left( n + 2^r \right)$
Radix sort (5/6)

Selection of the value $r$

[Graphs showing running time for different bit sizes and values of n and r.]
Radix sort (6/6)
Bucket sort (1/4)

- No comparisons between elements
- Runs in linear time when the input is drawn from a uniform distribution
- Assumes that the input is generated by a random process distributing the elements uniformly over the interval \([0, 1)\)
- Key IDEA: Divide the interval \([0, 1)\) into \(n\) equal-sized subintervals or \(buckets\)

\[\downarrow\]

Similar amount of input numbers falls into each bucket

\[\downarrow\]

Sort the numbers in each bucket and then go through them in order, listing the elements in each
Bucket sort (2/4)

**ALGORITHM:**

- n-element input array $A$, being satisfied that $0 \leq A[i] < 1$
- auxiliary array $B[1..k]$ of linked lists (buckets)
- assumed that there is a mechanism for maintaining linked lists
- output (concatenation of buckets) not sorted in place

1. $n \leftarrow \text{length}(A)$
2. $k \leftarrow \text{num of buckets}$
3. for $i \leftarrow 1$ to $n$
   3.1. do insert $A[i]$ into list $B[\text{ceil}(kA[i])]$
4. for $i \leftarrow 1$ to $k$
5. do sort list $B[i]$ with insertion sort
Bucket sort (3/4)

**EXAMPLE:** (k=10)

```
1 n ← length(A)
2 k ← 10
3 for i ← 1 to n
   4 do insert A[i] into list B[ceil(kA[i])]
4 for i ← 1 to k
   5 do sort list B[i]
       with insertion sort
6 concatenate lists B[1], ..., B[k] in order
```
**Bucket sort (4/4)**

**RUNNING TIME:**

- All lines except lines 4 and 5 take $O(n)$ time in the worst case
- It remains to balance the total time taken by the $k$ calls to insertion sort

```plaintext
1 $n \leftarrow \text{length}(A)$
2 $k \leftarrow \text{num of buckets}$
3 $\text{for } i \leftarrow 1 \text{ to } n$
4 $\quad \text{do insert } A[i] \text{ into list } B[\text{ceil}(kA[i])]$
5 $\text{for } i \leftarrow 1 \text{ to } k$
6 $\quad \text{do sort list } B[i] \text{ with insertion sort}$
7 $\text{concatenate lists } B[1], B[2], \ldots, B[k] \text{ in order}$
```

Since insertion sort runs in quadratic time, denoting as $n_i$ the number of elements in bucket $B_i$, the total time can be written as

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

This can be proved:

$$T(n) = \Theta(n) + n \cdot O\left(2 - \frac{1}{n}\right) = \Theta(n)$$
Sorting in Linear Time

Is there any QUESTION?