Comparison sort

All the sorting algorithms we have seen so far are comparison sorts: only use comparisons to determine the relative order of elements.

• E.g., insertion sort, merge sort, quicksort, heapsort.

The best worst-case running time that we’ve seen for comparison sorting is $O(n \lg n)$.

*Is $O(n \lg n)$ the best we can do?*

Decision trees can help us answer this question.

```
Decision-tree example
```

```
Sort $\langle a_1, a_2, ..., a_n \rangle$
```

```
Each internal node is labeled $ij$ for $i, j \in \{1, 2, ..., n\}$.

• The left subtree shows subsequent comparisons if $a_i \leq a_j$.

• The right subtree shows subsequent comparisons if $a_i \geq a_j$.

```
Sort $\langle a_1, a_2, a_3 \rangle$
```

```
(9, 4, 6)
```

```
Decision-tree example
```

```
Each internal node is labeled $ij$ for $i, j \in \{1, 2, ..., n\}$.

• The left subtree shows subsequent comparisons if $a_i \leq a_j$.

• The right subtree shows subsequent comparisons if $a_i \geq a_j$.

```
Sort $\langle a_1, a_2, a_3 \rangle$
```

```
(9, 4, 6)
```

```
Decision-tree example
```
**Decision-tree example**

Sort \(\langle a_1, a_2, a_3 \rangle\) = \(\langle 9, 4, 6 \rangle\):

Each internal node is labeled \(ij\) for \(i, j \in \{1, 2, \ldots, n\}\).
- The left subtree shows subsequent comparisons if \(a_i \leq a_j\).
- The right subtree shows subsequent comparisons if \(a_i \geq a_j\).

**Decision-tree model**

A decision tree can model the execution of any comparison sort:
- One tree for each input size \(n\).
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

**Lower bound for decision-tree sorting**

**Theorem.** Any decision tree that can sort \(n\) elements must have height \(\Omega(n \lg n)\).

**Proof.** The tree must contain \(\geq n\) leaves, since there are \(n!\) possible permutations. A height-\(h\) binary tree has \(\leq 2^h\) leaves. Thus, \(n! \leq 2^h\).

\[
\therefore h \geq \lg(n!)
\]

\(\lg\) is mono. increasing

\[
\geq \lg ((n/e)^n)
\]

Stirling’s formula

\[
= n \lg n - n \lg e
\]

\[
= \Omega(n \lg n).
\]

**Sorting in linear time**

**Counting sort:** No comparisons between elements.
- \(\textbf{Input:} A[1 \ldots n]\), where \(A[j] \in \{1, 2, \ldots, k\}\).
- \(\textbf{Key assumption} about input distribution: elements come from fixed range 1..k\)
- \(\textbf{Output:} B[1 \ldots n]\), sorted
- \(\textbf{Auxiliary storage:} C[1 \ldots k]\).
- \(\textbf{Key idea:} use ‘addressing’ array C to move elements from A directly into final B position.\)
The key idea: ‘addressing’ array

- First pass: Count number of elements of each value

**Input A:**

```
4 1 3 4 3
```

**Output B:**

```
\[\text{Space for } 1's\] \[\text{Space for } 3's\] \[\text{Space for } 4's\]
```

```
'1' 1
'2' 0
'3' 2
'4' 2
```

**Auxiliary**

- **Input A:**

```
4 1 3 4 3
```

- **Output B:**

```
\[\text{Space for } 1's\] \[\text{Space for } 3's\] \[\text{Space for } 4's\]
```

```
'1' 1
'2' 0
'3' 2
'4' 2
```

**Q:** Why bother ‘moving’ elements?

- **Question:** if I know the number of 1s, 2s, etc, why not fill out the output array directly?

- **Answer:** Each element may have lots of additional associated information: 1, 2, ... are just keys to compare elements, but entire data structures must move!

- **E.g.:** sorting students by dormitory distance to class. Remember more than just distance or dorm name!

**Input A:**

```
4 1 3 4 3
```

**Output B:**

```
\[\text{Space for } 1's\] \[\text{Space for } 3's\] \[\text{Space for } 4's\]
```

```
'1' 1
'2' 0
'3' 2
'4' 2
```

**Auxiliary**

- **Input A:**

```
4 1 3 4 3
```

- **Output B:**

```
\[\text{Space for } 1's\] \[\text{Space for } 3's\] \[\text{Space for } 4's\]
```

```
'1' 1
'2' 0
'3' 2
'4' 2
```

**Auxiliary**
The key idea: ‘addressing’ array

• First pass: Count number of elements of each value
• Post-process: Turn counts into pointers to final destination
• Second pass: Use pointers to move elements from A to B

Input A: 4 1 3 4 3
Output B: 1 3 3 4 4

The key idea: ‘addressing’ array

• First pass: Count number of elements of each value
• Post-process: Turn counts into pointers to final destination
• Second pass: Use pointers to move elements from A to B

Input A: 1 2 3 4 5
Output B: 1 2 3 4 5

Introduction to Algorithms
2/26/2008 L5.19
© Leiserson, Indyk, Kellis

Counting sort pseudocode

• Initialize C array
  ▶ For each value i
    ▶ First pass: count
      ▶ C[i] = |{key = i}|
    ▶ Post-process: pointers
      ▶ C[i] = |{key ≤ i}|
    ▶ Second pass: moving
      ▶ copy A[j] to B[j]
      ▶ update pointer

for i ← 1 to k
  do C[i] ← 0
for j ← 1 to n
  do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
  do C[i] ← C[i] + C[i-1]
for j ← n downto 1
  do B[C[A[j]]] ← A[j]
  do C[A[j]] ← C[A[j]] - 1

‘Running’ the pseudocode

A: 4 1 3 4 3
C: 1 2 3 4
B: 1 2 3 4 5

© Leiserson, Indyk, Kellis
Introduction to Algorithms 2/26/2008 1.5.19
Loop 1

\[ A: \begin{bmatrix} 4 & 1 & 3 & 4 & 3 \end{bmatrix} \]
\[ C: \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \]

\[ B: \begin{bmatrix} \text{empty} \end{bmatrix} \]

\[ \text{for } i \leftarrow 1 \text{ to } k \]
\[ \text{do } C[i] \leftarrow 0 \]

\[ \text{Loop 2} \]

\[ A: \begin{bmatrix} 4 & 1 & 3 & 4 & 3 \end{bmatrix} \]
\[ C: \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \]

\[ B: \begin{bmatrix} \text{empty} \end{bmatrix} \]

\[ \text{for } j \leftarrow 1 \text{ to } n \]
\[ \text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{\text{key} = i\}| \]
for \( i \leftarrow 2 \) to \( k \)
\[
do C[i] \leftarrow C[i] + C[i-1] \quad \triangleright C[i] = |\{ \text{key} \leq i \}|
\]
Loop 4

A: 4 1 3 4 3
B: 1 3 3 4
C: 1 1 1 4

for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] − 1

Analysis

Θ(k)
    \{ for i ← 1 to k
       do C[i] ← 0
    \}
Θ(n)
    \{ for j ← 1 to n
       do C[A[j]] ← C[A[j]] + 1
          \}
Θ(k)
    \{ for i ← 1 to k
       do C[i] ← C[i] + C[i−1]
    \}
Θ(n)
    \{ for j ← n downto 1
       do B[C[A[j]]] ← A[j]
          C[A[j]] ← C[A[j]] − 1
    \}
Θ(n + k)

Running time

If k = O(n), then counting sort takes Θ(n) time.
• But, sorting takes Ω(n lg n) time!
• Where’s the fallacy?
  • Ω(n lg n) time is for comparison sorting.
  • Counting sort is not a comparison sort.
  • (not a single comparison between elmts occurs)
• Why not always use counting sort?
  • Needs additional assumptions about input
  • In general case, 1..k may be >> n (eg. 32-bit ints)

Google interview questions for presidential candidates

• During his presidential bid, senator Barack Obama is asked a question key to the election (Nov 14, 2007):
  • Larry Schwimmer asks: “What’s the most efficient way to sort a million 32-bit integers?”
  • Without losing a heartbeat, Obama replies: “I think the bubble sort would be the wrong way to go”
• What Obama really meant was: “Radix Sort, clearly”

Radix sort

• Origin: Herman Hollerith’s card-sorting machine for the 1890 U.S. Census. (See Appendix D.)
  • Digit-by-digit sort.
  • Hollerith’s original (bad) idea: sort on most-significant digit first.
  • Good idea: Sort on least-significant digit first with auxiliary stable sort.
Stable sorting

Counting sort is a **stable** sort: it preserves the input order among equal elements.

A:

```
4 1 3 4 3
```

B:

```
1 3 3 4 4
```

**Exercise:** What other sorts have this property?

Operation of radix sort

<table>
<thead>
<tr>
<th>3 2 9</th>
<th>7 2 0</th>
<th>7 2 0</th>
<th>3 2 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5 7</td>
<td>3 5 5</td>
<td>3 2 9</td>
<td>4 5 7</td>
</tr>
<tr>
<td>6 5 7</td>
<td>4 3 6</td>
<td>4 3 6</td>
<td>6 5 7</td>
</tr>
<tr>
<td>8 3 9</td>
<td>4 5 7</td>
<td>8 3 9</td>
<td>4 3 6</td>
</tr>
<tr>
<td>4 3 6</td>
<td>6 5 7</td>
<td>3 5 5</td>
<td>6 5 7</td>
</tr>
<tr>
<td>7 2 0</td>
<td>3 2 9</td>
<td>4 5 7</td>
<td>7 2 0</td>
</tr>
<tr>
<td>3 5 5</td>
<td>8 3 9</td>
<td>6 5 7</td>
<td>8 3 9</td>
</tr>
</tbody>
</table>

Correctness of radix sort

**Induction on digit position**

- Assume that the numbers are sorted by their low-order \( t - 1 \) digits.
- Sort on digit \( t \)

<table>
<thead>
<tr>
<th>7 2 0</th>
<th>3 2 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 2 9</td>
<td>3 5 5</td>
</tr>
<tr>
<td>4 3 6</td>
<td>4 3 6</td>
</tr>
<tr>
<td>8 3 9</td>
<td>4 5 7</td>
</tr>
<tr>
<td>3 5 5</td>
<td>6 5 7</td>
</tr>
<tr>
<td>4 5 7</td>
<td>7 2 0</td>
</tr>
<tr>
<td>6 5 7</td>
<td>8 3 9</td>
</tr>
</tbody>
</table>

How many passes should we make?

- Sort \( n \) (e.g. 10⁶) integers of \( b \) (e.g. 32) bits each.
- Interpreted as having \( b/r \) ‘digits’, each base-2ⁿ.
- \( b \)-bit integer:
  - \( r \) bits
  - \( r \) bits
  - \( r \) bits

Correctness of radix sort

**Induction on digit position**

- Assume that the numbers are sorted by their low-order \( t - 1 \) digits.
- Sort on digit \( t \)
  - Two numbers that differ in digit \( t \) are correctly sorted.
  - Two numbers equal in digit \( t \) are put in the same order as the input \( \Rightarrow \) correct order().

Analysis of radix sort \( T(n,b) \)

- Assume counting sort is the auxiliary **stable** sort.

**How many passes should we make?**

**Example:** 32-bit word

\( r = 8 \) \( \Rightarrow \) \( b/r = 4 \) passes on base-\( 2^8 \) digits.

\( r = 16 \) \( \Rightarrow \) \( b/r = 2 \) passes on base-\( 2^{16} \) digits.
Analysis (continued)

\[ T(n, b) = \Theta\left( \frac{b}{r}(n + 2^r) \right) \]

\( b \) bits

\( \frac{b}{r} \) passes

\( r \) bits

Counting sort \( \Theta(n + k) \).
Range is \( (1..k = 1..2^r) \).

Choose \( r \) to minimize \( T(n, b) \):
- Increase \( r \) to have fewer passes = \( \frac{b}{r} \).
- Decrease \( r \) to have faster counting sort = \( 2^r \).

(If \( r \gg \lg n \), the time grows exponentially).

Analysis: choosing optimal \( r \)

\[ T(n, b) = \Theta\left( \frac{b}{r}(n + 2^r) \right) \]

Minimize \( T(n, b) \) by differentiating and setting to 0.
Or, just observe that we don’t want \( 2^r \gg n \), and there’s no harm asymptotically in choosing \( r \) as large as possible subject to this constraint.
Choosing \( r = \lg n \) implies \( T(n, b) = \Theta(bn/\lg n) \).
- For numbers in the range from 0 to \( n^d - 1 \), we have \( b = d \lg n \Rightarrow \) radix sort runs in \( \Theta(dn) \) time.

Conclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers):
- At most 3 passes when sorting \( \geq 2000 \) numbers.
- Merge sort and quicksort do at least \( \lceil \lg 2000 \rceil = 11 \) passes.

Potential downside: Radix sort displays little locality of reference, and thus a well-tuned quicksort may fare better on modern processors, which feature steep memory hierarchies.

Appendix: Punched-card technology

- Herman Hollerith (1860-1929)
- Punched cards
- Hollerith’s tabulating system
- Operation of the sorter
- Origin of radix sort
- “Modern” IBM card
- Web resources on punched-card technology

Herman Hollerith (1860-1929)

- The 1880 U.S. Census took almost 10 years to process.
- While a lecturer at MIT, Hollerith prototyped punched-card technology.
- His machines, including a “card sorter,” allowed the 1890 census total to be reported in 6 weeks.
- He founded the Tabulating Machine Company in 1911, which merged with other companies in 1924 to form International Business Machines.

Punched cards

- Punched card = data record.
- Hole = value.
- Algorithm = machine + human operator.
Hollerith’s tabulating system

- Pantograph card punch
- Hand-press reader
- Dial counters
- Sorting box

Figure from [Howells 2000].

Operation of the sorter

- An operator inserts a card into the press.
- Pins on the press reach through the punched holes to make electrical contact with mercury-filled cups beneath the card.
- Whenever a particular digit value is punched, the lid of the corresponding sorting bin lifts.
- The operator deposits the card into the bin and closes the lid.
- When all cards have been processed, the front panel is opened, and the cards are collected in order, yielding one pass of a stable sort.

Origin of radix sort

Hollerith’s original 1889 patent alludes to a most-significant-digit-first radix sort:

“The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then reassorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards.”

Least-significant-digit-first radix sort seems to be a folk invention originated by machine operators.

“Modern” IBM card

- One character per column.

So, that's why text windows have 80 columns!

Web resources on punched-card technology

- Doug Jones’s punched card index
- Biography of Herman Hollerith
  [http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Hollerith.html](http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Hollerith.html)
- The 1890 U.S. Census
  [http://www.nps.gov/history/education/1890census/index.htm](http://www.nps.gov/history/education/1890census/index.htm)
- Early history of IBM
- Pictures of Hollerith’s inventions
- Hollerith’s patent application (borrowed from Gordon Bell’s CyberMuseum)
- Biographies of Hollerith
  [http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Hollerith.html](http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Hollerith.html)
- The 1890 U.S. Census
  [http://www.cs.uiowa.edu/~jones/cards/1890.html](http://www.cs.uiowa.edu/~jones/cards/1890.html)

- Early history of IBM
- Pictures of Hollerith’s inventions
- Hollerith’s patent application (borrowed from Gordon Bell’s CyberMuseum)