Advanced Algorithmics (4AP)
Sorting - revisited

Jaak Vilo
2009 Spring
Sorting

• given a list, arrange values so that
  \[ L[1] \leq L[2] \leq \ldots \leq L[n] \]
• \( n \) elements => \( n! \) possible orderings
• One test \( L[i] \leq L[j] \) can divide \( n! \) to 2
  – Make a binary tree and calculate the depth
• \( \log( n! ) = O( n \log n ) \)
• Hence, lower bound for sorting is \( O( n \log n ) \)
  – using comparisons...
  – (proved using decision tree model)
Decision-tree example

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

$= \langle 9, 4, 6 \rangle$:

Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ has been established.
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, $n! \leq 2^h$.

$$h \geq \lg(n!)$$

$$\geq \lg ((n/e)^n)$$

$$= n \lg n - n \lg e$$

$$= \Omega(n \lg n).$$
• $\log(n!) = \log(n) + \log(n-1) + \ldots + \log(1)$

  a) $\leq n \log(n)$

  b) $\geq \frac{n}{2} \times \log\left(\frac{n}{2}\right) = \frac{n}{2} \log n - \frac{n}{2}$
Merge sort

```
Merge-Sort(A,p,r)
if p<r
    then q = (p+r)/2 // floor
    Merge-Sort( A, p, q )
    Merge-Sort( A, q+1,r)
    Merge( A, p, q, r )
```

It was invented by John von Neumann in 1945.
Run-time Analysis of Merge Sort

• Thus, the time required to sort an array of size $n > 1$ is:
  – the time required to sort the first half,
  – the time required to sort the second half, and
  – the time required to merge the two lists

• That is:
  $$T(n) = \begin{cases} 
    \Theta(1) & n = 1 \\
    2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 
  \end{cases}$$

• $O(n \log n)$ worst case
Divide and conquer

Quicksort an \( n \)-element array:

1. **Divide:** Partition the array into two subarrays around a *pivot* \( x \) such that elements in lower subarray \( \leq x \leq \) elements in upper subarray.

   \[ \leq x \quad x \quad \geq x \]

2. **Conquer:** Recursively sort the two subarrays.

3. **Combine:** Trivial.

**Key:** *Linear-time partitioning subroutine.*
Pseudocode for quicksort

\textbf{QUICKSORT}(A, p, r)

\textbf{if} p < r

\textbf{then} q \leftarrow \textbf{PARTITION}(A, p, r)

\textbf{QUICKSORT}(A, p, q-1)

\textbf{QUICKSORT}(A, q+1, r)

\textbf{Initial call:} \textbf{QUICKSORT}(A, 1, n)
QuickSort in practice

- QuickSort is a great general-purpose sorting algorithm.
- QuickSort is typically over twice as fast as merge sort.
- QuickSort can benefit substantially from code tuning.
- QuickSort behaves well even with caching and virtual memory.
Quicksort

- Average: $O(n \log n)$
- Worst $O(n^2)$
• Minutesort – max amount sorted in 1 minute
  – 116GB in 58.7 sec (Jim Wyllie, IBM Research)
  – 40-node 80-Itanium cluster, SAN array of 2,520 disks

• Performance / Price Sort and PennySort

  • **Sort Benchmark Home Page**
  • We have a new benchmark called new GraySort, new in memory of the father of the sort benchmarks, Jim Gray. It replaces TeraByteSort which is now retired.
  • The submission deadline is new 15 April 2009.
  •

  **New rules for GraySort:**
  • The input file size is now minimum ~100TB or 1T records. Entries with larger input sizes also qualify.
  • The winner will have the fastest SortedRecs/Min.
  • We now provide a new input generator that works in parallel and generates binary data. See below.
  • For the Daytona category, we have two new requirements. (1) The sort must run continuously (repeatedly) for a minimum 1 hour. (This is a minimum reliability requirement). (2) The system cannot overwrite the input file.
## Year 2008 Results

*commentary by Mehul Shah on 2007 winners [here](http://www.hpl.hp.com/hosted/sortbenchmark/)*

<table>
<thead>
<tr>
<th></th>
<th>Daytona</th>
<th>Indy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Penny</strong></td>
<td><em>(new 2008)</em> 1,812 M records (181 GB) in 2,408 seconds</td>
<td><em>(new 2008)</em> 1,896 M records (190 GB) in 2,408 seconds</td>
</tr>
<tr>
<td></td>
<td><em>psort</em></td>
<td><em>psort</em></td>
</tr>
<tr>
<td></td>
<td>2.4 GHz AMD Athlon 64, 2 GB RAM, 4x160GB SATA disks, Linux</td>
<td>2.4 GHz AMD Athlon 64, 2 GB RAM, 4x160GB SATA disks, Linux</td>
</tr>
<tr>
<td></td>
<td>Paolo Bertasi, Marco Bressan and Enrico Pescero</td>
<td>Paolo Bertasi, Marco Bressan and Enrico Pescero</td>
</tr>
<tr>
<td></td>
<td>Univ Pavia, Italy</td>
<td>Univ Padova, Italy</td>
</tr>
<tr>
<td><strong>Minute</strong></td>
<td>214 GB (2140 million records)</td>
<td>264 GB (2640 M records)</td>
</tr>
<tr>
<td></td>
<td><em>TokuSampleSort</em></td>
<td><em>TokuSampleSort</em></td>
</tr>
<tr>
<td></td>
<td>tx2500 disk cluster, 400 nodes x (2 processors, 6-disk RAID, 8 GB memory)</td>
<td>tx2500 disk cluster, 400 nodes x (2 processors, 6-disk RAID, 8 GB memory)</td>
</tr>
<tr>
<td></td>
<td>Bradley C. Kuszma, MIT</td>
<td>Bradley C. Kuszma, MIT</td>
</tr>
<tr>
<td><strong>TeraByte</strong></td>
<td><em>(new 2008)</em> 209 seconds (3.48 minutes)</td>
<td><em>(new 2007)</em> 197 seconds (3.28 minutes)</td>
</tr>
<tr>
<td></td>
<td><em>Hadoop</em></td>
<td><em>TokuSampleSort</em></td>
</tr>
<tr>
<td></td>
<td>910 nodes x (4 dual-core processors, 4 disks, 8 GB memory)</td>
<td>tx2500 disk cluster, 400 nodes x (2 processors, 6-disk RAID, 8 GB memory)</td>
</tr>
<tr>
<td></td>
<td>Owen O’Malley, Yahoo</td>
<td>Bradley C. Kuszma, MIT</td>
</tr>
<tr>
<td><strong>Joule</strong></td>
<td><em>(2007)</em> 10 GB sorted using 8.6 kJoules</td>
<td><em>(2007)</em> 100 GB sorted using 88 kJoules</td>
</tr>
<tr>
<td></td>
<td>11,600 records sorted / joule</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>CoolSort</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mobile Core 2 Duo, 13 SATA laptop disks, <em>Nsort</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Suzanne Rivoire (Stanford), Mehul A. Shah (HP Labs), Partha Ranganathan (HP Labs), Christos Korvakis (Stanford)</td>
<td></td>
</tr>
<tr>
<td><strong>Joule</strong></td>
<td><em>(2009)</em> 309</td>
<td><em>(2007)</em> 100 GB sorted using 88 kJoules</td>
</tr>
</tbody>
</table>
Counting sort: No comparisons between elements.

• **Input**: $A[1 \ldots n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
• **Output**: $B[1 \ldots n]$, sorted.
• **Auxiliary storage**: $C[1 \ldots k]$. 
Counting sort

for $i \leftarrow 1$ to $k$
    do $C[i] \leftarrow 0$
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$ \> $C[i] = |\{\text{key} = i\}|$
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$ \> $C[i] = |\{\text{key} \leq i\}|$
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
Loop 1

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

\[ B: \begin{bmatrix} & & & & \end{bmatrix} \]

\[ C: \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \]

\[ \text{for } i \leftarrow 1 \text{ to } k \]
\[ \text{do } C[i] \leftarrow 0 \]
Loop 2

\[
\begin{array}{c}
A: & 4 & 1 & 3 & 4 & 3 \\
B: & & & & & \\
C: & 1 & 0 & 2 & 2 \\
\end{array}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \\
\text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright C[i] = |\{\text{key} = i\}|\]
Loop 3

\[
\begin{array}{cccccc}
A: & 1 & 2 & 3 & 4 & 5 \\
& 4 & 1 & 3 & 4 & 3 \\
B: & & & & & \\
C: & 1 & 2 & 3 & 4 \\
& 1 & 0 & 2 & 2 \\
C': & 1 & 1 & 3 & 5 \\
\end{array}
\]

for \( i \leftarrow 2 \) to \( k \)

do \( C[i] \leftarrow C[i] + C[i-1] \) \( \triangleright C[i] = |\{ \text{key} \leq i \}| \)
Loop 4

\[
A: \begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3 \\
\end{array} \quad B: \begin{array}{ccc}
3 & 4 & \\
\end{array} \quad C: \begin{array}{cccc}
1 & 1 & 2 & 5 \\
\end{array} \\
C': \begin{array}{cccc}
1 & 1 & 2 & 4 \\
\end{array}
\]

for \( j \leftarrow n \) downto 1

\[
C[A[j]] < C[A[j]] \quad 1
\]
Analysis

\(\Theta(k)\) \{ 
\begin{align*}
&\text{for } i \leftarrow 1 \text{ to } k \\
&\quad \text{do } C[i] \leftarrow 0
\end{align*}
\}

\(\Theta(n)\) \{ 
\begin{align*}
&\text{for } j \leftarrow 1 \text{ to } n \\
&\quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1
\end{align*}
\}

\(\Theta(k)\) \{ 
\begin{align*}
&\text{for } i \leftarrow 2 \text{ to } k \\
&\quad \text{do } C[i] \leftarrow C[i] + C[i-1]
\end{align*}
\}

\(\Theta(n)\) \{ 
\begin{align*}
&\text{for } j \leftarrow n \text{ downto } 1 \\
&\quad \text{do } B[C[A[j]]] \leftarrow A[j] \\
&\quad \quad C[A[j]] \leftarrow C[A[j]] - 1
\end{align*}
\}

\(\Theta(n + k)\)
Running time

If \( k = O(n) \), then counting sort takes \( \Theta(n) \) time.

• But, sorting takes \( \Omega(n \lg n) \) time!
• Where’s the fallacy?

Answer:

• *Comparison sorting* takes \( \Omega(n \lg n) \) time.
• Counting sort is not a *comparison sort*.
• In fact, not a single comparison between elements occurs!
Stable sorting

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

Exercise: What other sorts have this property?
Radix sort

- **Origin**: Herman Hollerith’s card-sorting machine for the 1890 U.S. Census. (See Appendix.)
- 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.
Radix sort

Radix-Sort(A,d)

1. for i = 1 to d
2. do use a stable sort to sort A on digit i
Operation of radix sort

3 2 9  7 2 0  7 2 0  3 2 9
4 5 7  3 5 5  3 2 9  3 5 5
6 5 7  4 3 6  4 3 6  4 3 6
8 3 9  4 5 7  8 3 9  4 5 7
4 3 6  6 5 7  3 5 5  6 5 7
7 2 0  3 2 9  4 5 7  7 2 0
3 5 5  8 3 9  6 5 7  8 3 9
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.
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

- Assume counting sort is the auxiliary stable sort.
- Sort $n$ computer words of $b$ bits each.
- Each word can be viewed as having $\frac{b}{r}$ base-$2^r$ digits.

**Example:** 32-bit word

$$r = 8 \Rightarrow \frac{b}{r} = 4$$ passes of counting sort on base-$2^8$ digits; or $r = 16 \Rightarrow \frac{b}{r} = 2$ passes of counting sort on base-$2^{16}$ digits.

*How many passes should we make?*
Recall: Counting sort takes $\Theta(n + k)$ time to sort $n$ numbers in the range from 0 to $k - 1$. If each $b$-bit word is broken into $r$-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are $b/r$ passes, we have

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

Choose $r$ to minimize $T(n, b)$:

- Increasing $r$ means fewer passes, but as $r \gg \lg n$, the time grows exponentially.
Choosing $r$

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

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 $\lceil \log 2000 \rceil = 11$ passes.

**Downside:** Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.
Radix sort using lists (stable)
Radix sort using lists (stable)

1. a → bba → aba → cca
   b → bbb → adb → ccb
   c → aac → ccc
   d → aad
Radix sort using lists (stable)

1. a → bba → aba → cca
   b → bbb → adb → ccb
   c → aac → ccc
   d → aad

2. a → aac → aad
   b → bba → aba → bbb
   c → cca → ccb → ccc
   d → adb

3. a → aac → aad → aba → adb
   b → bba → bbb
   c → cca → ccb → ccc
Why not from left to right?

- Swap ‘0’ with first ‘1’
- Idea 1: recursively sort first and second half
  – Exercise?
Bitwise sort left to right

• Idea2:
  – swap elements only if the prefixes match...

  – For all bits from most significant
    • advance when 0
    • when 1 -> look for next 0
      – if prefix matches, swap
      – otherwise keep advancing on 0’s and look for next 1
Bitwise left to right sort

/* Historical sorting – was used in Univ. of Tartu using assembler.... */
/* C implementation – Jaak Vilo, 1989 */

void bitwisesort( SORTTYPE *ARRAY, int size )
{
    int i, j, tmp, nrbit ;

    register SORTTYPE mask, curbit, group ;

    nrbits = sizeof( SORTTYPE ) * 8 ;

    curbit = 1 << (nrbits-1) ;    /* set most significant bit 1 */
    mask = 0;                    /* mask of the already sorted area */

    Jaak Vilo, Univ. of Tartu


```c
do {  /* For each bit */
    i=0;

    new_mask:
        for( ; ( i < size ) && ( ! (ARRAY[i] & curbit) ) ; i++ ) ;  /* Advance while bit == 0 */
    if( i >= size ) goto array_end ;
    group = ARRAY[i] & mask ;  /* Save current prefix snapshot */

    j=i;  /* memorize location of 1 */
    for( ; ; ) {
        if ( ++i >= size ) goto array_end ;  /* reached end of array */
        if ( (ARRAY[i] & mask) != group ) goto new_mask ;  /* new prefix */

        if ( ! (ARRAY[i] & curbit) ) {  /* bit is 0 – need to swap with previous location of 1, A[i] ↔ A[j] */
            tmp = ARRAY[i];  ARRAY[i] = ARRAY[j];  ARRAY[j] = tmp ;  j+= 1 ;  /* swap and increase j to the next possible 1 */
        }
    }

array_end:
    mask = mask | curbit ;  /* area under mask is now sorted */
    curbit >>= 1 ;  /* next bit */
} while( curbit ) ;  /* until all bits have been sorted... */
```

Jaak Vilo, Univ. of Tartu
Bitwise from left to right

0010000
0010010
0101000
0101100
100101\textcolor{red}{0}\ 
1001001
1001001
1111000

• Swap ‘0’ with first ‘1’

Jaak Vilo, Univ. of Tartu
Bucket sort

- Assume uniform distribution
- Allocate $O(n)$ buckets
- Assign each value to pre-assigned bucket
Sort small buckets with insertion sort
Order statistics

• Minimum – the smallest value
• Maximum – the largest value
• In general i’th value.
• Find the median of the values in the array
• Median in sorted array A :
  – n is odd  $A[(n+1)/2]$
  – n is even – $A\lfloor (n+1)/2 \rfloor$ or $A\lceil (n+1)/2 \rceil$
Order statistics

• Input: A set $A$ of $n$ numbers and $i$,  $1 \leq i \leq n$
• Output: $x$ from $A$ that is larger than exactly $i-1$ elements of $A$
Minimum

Minimum(A)

1 \( \text{min} = A[1] \)

2 \( \text{for } i = 2 \text{ to length}(A) \)

3 \( \text{if } \text{min} > A[i] \)

4 \( \text{then } \text{min} = A[i] \)

5 \( \text{return } \text{min} \)

\( n-1 \) comparisons.
Min and max together

• compare every two elements $A[i], A[i+1]$
• Compare larger against current max
• Smaller against current min

• $3 \lceil n / 2 \rceil$
Selection in expected $O(n)$

Randomised-select( A, p, r, i )

if p=r then return A[p]

q = Randomised-Partition(A,p,r)

k = q – p + 1       // nr of elements in subarr

if i<= k

    then return Randomised-Partition(A,p,q,i)

else return Randomised-Partition(A,q+1,r,i-k)
Conclusion

• Sorting in general $O(n \log n)$
• Quicksort is rather good

• Linear time sorting is achievable when one does not assume only direct comparisons

• Find i’th value – expected $O(n)$

• Find i’th value: worst case $O(n)$ – see CLRS