CSE 332:
Sorting Bound, and Radix Sort

Richard Anderson, Steve Seitz
Winter 2014
Announcements (2/7/14)

• Midterm next Monday
  – Review session: Saturday 12-4, EEB 105
How fast can we sort?

Heapsort, Mergesort, Heapsort, AVL sort all have $O(N \log N)$ worst case running time.

These algorithms, along with Quicksort, also have $O(N \log N)$ average case running time.

Can we do any better?
Permutations

• Suppose you are given \( N \) elements
  – Assume no duplicates

• How many possible orderings can you get?
  – Example: \( a, b, c \) (\( N = 3 \))

\[
3 \cdot 2 \cdot 1 = 6 \quad N! = (N) (N-1) (N-2) \ldots
\]
Permutations

• How many possible orderings can you get?
  – Example: a, b, c \((N = 3)\)
  – (a b c), (a c b), (b a c), (b c a), (c a b), (c b a)
  – 6 orderings = \(3 \cdot 2 \cdot 1 = 3!\) (i.e., “3 factorial”)

• For \(N\) elements
  – \(N\) choices for the first position, \((N-1)\) choices for the second position, …, (2) choices, 1 choice
  – \(N(N-1)(N-2)\cdots(2)(1) = N!\) possible orderings
Sorting Model

Recall our basic sorting assumption:

We can only compare two elements at a time.

These comparisons prune the space of possible orderings.

We can represent these concepts in a…
Decision Tree

The leaves contain all the possible orderings of a, b, c.
Decision Trees

• A Decision Tree is a Binary Tree such that:
  – Each node = a set of orderings
    • i.e., the remaining solution space
  – Each edge = 1 comparison
  – Each leaf = 1 unique ordering
  – How many leaves for N distinct elements?
    $N!$

• Only 1 leaf has the ordering that is the desired correctly sorted arrangement
Decision Tree Example

Possible orders:
- $a < b < c$, $b < c < a$, $c < a < b$
- $a < c < b$, $b < a < c$, $c < b < a$

Actual order:
- $a < b < c$
- $a < c < b$
Decision Trees and Sorting

• Every sorting algorithm corresponds to a decision tree
  – Finds correct leaf by choosing edges to follow
    • i.e., by making comparisons
• We will focus on worst case run time
• Observations:
  – Worst case run time $\geq$ max number of comparisons
  – Max number of comparisons
    = length of the longest path in the decision tree
    = tree height
How many leaves on a tree?

Suppose you have a binary tree of height $h$. How many leaves in a perfect tree?

We can prune a perfect tree to make any binary tree of same height. Can # of leaves increase?
Lower bound on Height

• A binary tree of height $h$ has at most $2^h$ leaves
  – Can prove by induction
• A decision tree has $N!$ leaves. What is its minimum height?

\[ N! = \# \text{leaves} \leq 2^h \]

\[ h \geq \log_2 N! \]
An Alternative Explanation

At each decision point, one branch has \( \leq \frac{1}{2} \) of the options remaining, the other has \( \geq \frac{1}{2} \) remaining.

Worst case: we always end up with \( \geq \frac{1}{2} \) remaining.

Best algorithm, in the worst case: we always end up with exactly \( \frac{1}{2} \) remaining.

Thus, in the worst case, the best we can hope for is halving the space \( d \) times (with \( d \) comparisons), until we have an answer, i.e., until the space is reduced to size = 1.

The space starts at \( N! \) in size, and halving \( d \) times means multiplying by \( 1/2^d \), giving us a lower bound on the worst case:

\[
\frac{N!}{2^d} = 1 \quad \Rightarrow \quad N! = 2^d \quad \Rightarrow \quad d = \log_2(N!)
\]
Lower Bound on $\log(N!)$

$$
\log N! = \log \left[ N (N-1)(N-2) \ldots \right] \\
= \log N + \log (N-1) + \log (N-2) + \ldots \\
\geq \frac{N}{2} \log \frac{N}{2} \\
\Omega (N \log N)
$$
Worst case run time of any comparison-based sorting algorithm is $\Omega(N \log N)$.

Can also show that average case run time is also $\Omega(N \log N)$.

Can we do better if we don’t use comparisons? (Huh?)
Can we sort in $O(n)$?

• Suppose keys are integers between 0 and 1000
BucketSort (aka BinSort)

If all values to be sorted are integers between 1 and $B$, create an array `count` of size $B$, increment counts while traversing the input, and finally output the result.

**Example**  $B=5$.  Input = (5,1,3,4,3,2,1,1,5,4,5)

<table>
<thead>
<tr>
<th>count array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Running time to sort $n$ items?

$O(N + B)$
What about our $\Omega(n \log n)$ bound?

not comparison based
Dependence on $B$

What if $B$ is very large (e.g., $2^{64}$)?
Fixing impracticality: RadixSort

• RadixSort: generalization of BucketSort for large integer keys

• Origins go back to the 1890 census.

• Radix = “The base of a number system”
  – We’ll use 10 for convenience, but could be anything

• Idea:
  – BucketSort on one digit at a time
  – After k\textsuperscript{th} sort, the last k digits are sorted
  – Set number of buckets: \( B = \text{radix.} \)
# Radix Sort Example

Input: 478, 537, 9, 721, 3, 38, 123, 67

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BucketSort</strong> on 1’s</td>
<td></td>
<td>721</td>
<td></td>
<td>3</td>
<td>123</td>
<td></td>
<td>537</td>
<td>478</td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BucketSort</strong> on 10’s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BucketSort</strong> on 100’s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output:
## Radix Sort Example (1\textsuperscript{st} pass)

Each bucket is sorted using a different bucket sort algorithm (bucket sort). In this example, the buckets are sorted by 1's digit:

### Input data

| 478 | 537 | 9  | 721 | 3  | 38 | 123 | 67  |

### After 1\textsuperscript{st} pass

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>3</td>
<td>123</td>
<td>537</td>
<td>478</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.
Radix Sort Example (2\textsuperscript{nd} pass)

<table>
<thead>
<tr>
<th>After 1\textsuperscript{st} pass</th>
<th>Bucket sort by 10’s digit</th>
<th>After 2\textsuperscript{nd} pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>03 09</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>721 123</td>
<td>9</td>
</tr>
<tr>
<td>123</td>
<td>537 38</td>
<td>721</td>
</tr>
<tr>
<td>537</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td>67</td>
<td></td>
<td>537</td>
</tr>
<tr>
<td>478</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>67</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>478</td>
</tr>
</tbody>
</table>
Radix Sort Example (3\textsuperscript{rd} pass)

<table>
<thead>
<tr>
<th>After 2\textsuperscript{nd} pass</th>
<th>Bucket sort by 100’s digit</th>
<th>After 3\textsuperscript{rd} pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>721</td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>537</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>478</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>003</td>
<td>123</td>
<td></td>
<td>478</td>
<td>537</td>
<td></td>
<td>721</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>038</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>067</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Invariant**: after k passes the low order k digits are sorted.
Radixsort: Complexity

In our examples, we had:

- Input size, \( N \)
- Number of buckets, \( B = 10 \)
- Maximum value, \( M < 10^3 \)
- Number of passes, \( P = \)

How much work per pass?

Total time?
Choosing the Radix

Run time is roughly proportional to:

\[ P(B+N) = \log_B M(B+N) \]

Can show that this is minimized when:

\[ B \log_e B \approx N \]

In theory, then, the best base (radix) depends only on \( N \).

For fast computation, prefer \( B = 2^b \). Then best \( b \) is:

\[ b + \log_2 b \approx \log_2 N \]

Example:

- \( N = 1 \) million (i.e., \( \sim 2^{20} \)) 64 bit numbers, \( M = 2^{64} \)
- \( \log_2 N \approx 20 \rightarrow b = 16 \)
- \( B = 2^{16} = 65,536 \) and \( P = \log_{(2^{16})} 2^{64} = 4 \).

In practice, memory word sizes, space, other architectural considerations, are important in choosing the radix.
Big Data: External Sorting

Goal: **minimize disk/tape access** time:
- Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
- Mergesort scans linearly through arrays, leading to (relatively) efficient sequential disk access

Basic Idea:
- Load chunk of data into Memory, sort, store this “run” on disk/tape
- Use the Merge routine from Mergesort to merge runs
- Repeat until you have only one run (one sorted chunk)
- Mergesort can leverage multiple disks
- Weiss gives some examples
Sorting Summary

\(O(N^2)\) average, worst case:
- Selection Sort, Bubblesort, Insertion Sort

\(O(N \log N)\) average case:
- Heapsort: In-place, not stable.
- AVL Sort: \(O(N)\) extra space (including tree pointers, possibly poor memory locality), stable.
- Mergesort: \(O(N)\) extra space, stable.

\(\Omega(N \log N)\) worst and average case:
- Any comparison-based sorting algorithm

\(O(N)\)