CSE 326: Data Structures
Bucket and Radix Sorts

Brian Curless

Spring 2008

Announcements (5/15/08)

- Next homework will be assigned today.
- Graded homework #4 returned at end of class today.
- Reading for this lecture: Chapter 7.

BucketSort (aka BinSort)

If all values to be sorted are *known* to be 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)

`count` array	
1	III
2	I
3	II
4	II
5	III

1,1,1,2,3,3,4,4,5,5,5

Running time to sort n items?

$O(n+B)$

Dependence on *B*

What if…

…*B* is a constant? $O(n)$

…*B* is proportional to n? $O(n)$

…*B* is a very large constant (e.g., 2^{64})

$O(n)$ … but impractical!

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 each **digit**, least significant to most significant (lsd to msd)
  - Set number of buckets: *B* = radix.
  - Don't just count; insert the numbers into the buckets.

Radix Sort Example

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

BucketSort on 1's

0	1	2	3	4	5	6	7	8	9
	721		03 123				537 67	478 38	09

BucketSort on 10's

0	1	2	3	4	5	6	7	8	9
003 009		721 123	537 038			067	478		

BucketSort on 100's

0	1	2	3	4	5	6	7	8	9
3 9 38 67	123			478	537		721		

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

Radix Sort Example (1^{st} pass)

Input data

478
537
9
721
3
38
123
67

Bucket sort by 1's digit

0	1	2	3	4	5	6	7	8	9
	721		3 123				537 67	478 38	9

After 1^{st} pass

721
3
123
537
67
478
38
9

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^{nd} pass)

After 1^{st} pass

721
3
123
537
67
478
38
9

Bucket sort by 10's digit

0	1	2	3	4	5	6	7	8	9
03 09		721 123	537 38			67	478		

After 2^{nd} pass

3
9
721
123
537
38
67
478

Radix Sort Example (3^{rd} pass)

After 2^{nd} pass

3
9
721
123
537
38
67
478

Bucket sort by 100's digit

0	1	2	3	4	5	6	7	8	9
003	123			478	537		721		
009									
038									
067									

After 3^{rd} pass

3
9
38
67
123
478
537
721

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

Your Turn

Another Example

Input: 126, 328, 636, 341, 416, 131, 328

BucketSort on 1's

0	1	2	3	4	5	6	7	8	9

BucketSort on 10's

0	1	2	3	4	5	6	7	8	9

BucketSort on 100's

0	1	2	3	4	5	6	7	8	9

Output:

Radixsort: Complexity

In our examples, we had:
- Input size, N
- Number of buckets, B = 10
- Maximum value, $M \le 10^3$
- Number of passes, P = $\log_{10} 10^3 \approx \log_B M$

How much work per pass? Same as BucketSort!

$O(N+B)$

Total time? $O(P(N+B))$

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.

Summary of sorting

$O(N^2)$ average, worst case:

- **Selection Sort**, **Bubblesort**, **Insertion Sort**

$O(N^{4/3})$ worst case:

- **Shell Sort**

$O(N \log N)$ average case:

- **Heapsort**: In-place, not stable.
- (**Balanced**) **Binary Tree Sort**: $O(N)$ extra space (including tree pointers, possibly poor memory locality), stable.
- **Mergesort**: $O(N)$ extra space, stable.
- **Quicksort**: claimed fastest in practice, but $O(N^2)$ worst case. Recursion/stack requirement. Not stable.

$\Omega(N \log N)$ worst and average case:

- **Any comparison-based sorting algorithm**

$O(N)$ – $O(p(N+k))$

- **Radix Sort**: fast and stable. Not comparison based. Not in-place. Poor memory locality can undercut performance.