CS473 - Algorithms I

Lecture 9 Sorting in Linear Time

View in slide-show mode

CS 473 – Lecture 9

How Fast Can We Sort?

- □ The algorithms we have seen so far:
 - Based on comparison of elements
 - We only care about the relative ordering between the elements (not the actual values)
 - □ The smallest worst-case runtime we have seen so far: O(nlgn)
 - $\Box \text{ Is } O(nlgn) \text{ the best we can do?}$
- Comparison sorts: Only use comparisons to determine the relative order of elements.

Decision Trees for Comparison Sorts

- Represent a sorting algorithm abstractly in terms of a decision tree
 - A binary tree that represents the comparisons between elements in the sorting algorithm
 - Control, data movement, and other aspects are ignored
- One decision tree corresponds to one sorting algorithm and one value of n (input size)

Reminder: Insertion Sort (from Lecture 1)

Insertion-Sort (A)

- for j ← 2 to n do
 key ← A[j];
 i ← j 1;
 while i > 0 and A[i] > key
 - 5. $A[i+1] \leftarrow A[i];$
 - $6. \qquad i \leftarrow i 1;$
 - endwhile
- 7. $A[i+1] \leftarrow key;$ endfor

Loop invariant: The subarray A[1..j-1] is always sorted



Reminder: Insertion Sort (from Lecture 1)

Insertion-Sort (A)

- 1. for $j \leftarrow 2$ to n do
- 2. key $\leftarrow A[j];$
- 3. $i \leftarrow j 1;$
- 4. while i > 0 and A[i] > key
 do
- 5. $A[i+1] \leftarrow A[i];$
- $6. \qquad i \leftarrow i 1;$
 - endwhile
- 7. $A[i+1] \leftarrow key;$

endfor

CS 473 – Lecture 9

Cevdet Aykanat and Mustafa Ozdal Computer Engineering Department, Bilkent University

Shift right the entries in A[1..j-1] that are > key already sorted j < key > key j < key > key

Reminder: Insertion Sort (from Lecture 1)

Insertion-Sort (A)

- 1. for $j \leftarrow 2$ to n do
- 2. key $\leftarrow A[j];$
- 3. i ← j 1;
- 4. while i > 0 and A[i] > key do
- 5. $A[i+1] \leftarrow A[i];$
- $6. \qquad i \leftarrow i 1;$

endwhile

7. $A[i+1] \leftarrow key;$ endfor



now sorted

 \leq key

key

>key

CS 473 – Lecture 9

Different Outcomes for Insertion Sort and n=3 Input: $\langle a_1, a_2, a_3 \rangle$



CS 473 – Lecture 9

Decision Tree for Insertion Sort and n=3



Decision Tree Model for Comparison Sorts

- \square Internal node (i:j): Comparison between elements a_i and a_j
- □ <u>Leaf node</u>: An output of the sorting algorithm
- Path from root to a leaf: The execution of the sorting algorithm for a given input
- All possible executions are captured by the decision tree
- All possible outcomes (permutations) are in the leaf nodes

CS 473 – Lecture 9

Decision Tree for Insertion Sort and n=3 Input: <9, 4, 6>



CS 473 – Lecture 9

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

<u>The running time of the algorithm</u> = the length of the path taken <u>Worst case running time</u> = height of the tree

Lower Bound for Comparison Sorts

- \square Let **n** be the number of elements in the input array.
- What is the min number of leaves in the decision tree?
 <u>n!</u> (because there are n! permutations of the input array, and all possible outputs must be captured in the leaves)
- What is the max number of leaves in a binary tree of height h?

2^h

□ So, we must have:

$$2^h \ge n!$$

CS 473 – Lecture 9

Lower Bound for Decision Tree Sorting

<u>**Theorem</u>**: Any comparison sort algorithm requires $\Omega(nlgn)$ comparisons in the worst case.</u>

Proof: We'll prove that any decision tree corresponding to a comparison sort algorithm must have height $\Omega(nlgn)$ $2^{h} \ge n!$ (from previous slide)

- $h \ge lg(n!)$
 - $\geq lg((n/e)^n)$ (Stirling's approximation)
 - = nlgn n lge
 - $= \Omega(nlgn)$

Lower Bound for Decision Tree Sorting

<u>Corollary</u>: Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof: The O(nlgn) upper bounds on the runtimes for heapsort and merge sort match the $\Omega(nlgn)$ worst-case lower bound from the previous theorem.

Sorting in Linear Time

Counting sort: No comparisons between elements

Input: A[1 .. n], where A[j] \in {1, 2, ..., k} Output: B[1 .. n], sorted Auxiliary storage: C[1 .. k]

for i ← 1 to k do
 C[i] ← 0
for j ← 1 to n do
 C[A[j]] ← C[A[j]] + 1
// C[i] = |{key = i}|

for i ← 2 **to** k **do** C[i] ← C[i] + C[i-1] // C[i] = |{key ≤ i}|

for $j \leftarrow n$ downto 1 do $B[C[A[j]]] \leftarrow A[j]$ $C[A[j]] \leftarrow C[A[j]] - 1$



CS 473 – Lecture 9

for i ← 1 to k do
 C[i] ← 0
for j ← 1 to n do
 C[A[j]] ← C[A[j]] + 1
// C[i] = |{key = i}|

for i ← 2 **to** k **do** C[i] ← C[i] + C[i-1] // C[i] = |{key ≤ i}|

for $j \leftarrow n$ downto 1 do $B[C[A[j]]] \leftarrow A[j]$ $C[A[j]] \leftarrow C[A[j]] - 1$ <u>Step 1</u>: Initialize all counts to 0



CS 473 – Lecture 9

for i ← 1 to k do
 C[i] ← 0
for j ← 1 to n do
 C[A[j]] ← C[A[j]] + 1
// C[i] = |{key = i}|

for $i \leftarrow 2$ to k do $C[i] \leftarrow C[i] + C[i-1]$ $// C[i] = |\{key \le i\}|$

for $j \leftarrow n$ downto 1 do $B[C[A[j]]] \leftarrow A[j]$ $C[A[j]] \leftarrow C[A[j]] - 1$ <u>Step 2</u>: Count the number of occurrences of each value in the input array



CS 473 – Lecture 9

for i ← 1 to k do
 C[i] ← 0
for j ← 1 to n do
 C[A[j]] ← C[A[j]] + 1
// C[i] = |{key = i}|

for i ← 2 **to** k **do** C[i] ← C[i] + C[i-1] // C[i] = |{key ≤ i}|

for $j \leftarrow n$ downto 1 do $B[C[A[j]]] \leftarrow A[j]$ $C[A[j]] \leftarrow C[A[j]] - 1$ <u>Step 3</u>: Compute the number of elements less than or equal to each value



CS 473 – Lecture 9

for i ← 1 to k do
 C[i] ← 0
for j ← 1 to n do
 C[A[j]] ← C[A[j]] + 1
// C[i] = |{key = i}|

for i ← 2 **to** k **do** C[i] ← C[i] + C[i-1] // C[i] = |{key ≤ i}|

for $j \leftarrow n$ downto 1 do $B[C[A[j]]] \leftarrow A[j]$ $C[A[j]] \leftarrow C[A[j]] - 1$ <u>Step 4</u>: Populate the output array

There are C[3] = 3 *elts that are* ≤ 3



CS 473 – Lecture 9

for i ← 1 to k do
 C[i] ← 0
for j ← 1 to n do
 C[A[j]] ← C[A[j]] + 1
// C[i] = |{key = i}|

for i ← 2 **to** k **do** C[i] ← C[i] + C[i-1] // C[i] = |{key ≤ i}|

for j ← n **downto** 1 **do** B[C[A[j]]] ← A[j] C[A[j]] ← C[A[j]] - 1 <u>Step 4</u>: Populate the output array

There are C[4] = 5 *elts that are* ≤ 4



CS 473 – Lecture 9

for $i \leftarrow 1$ to k do $C[i] \leftarrow 0$ for $j \leftarrow 1$ to n do $C[A[j]] \leftarrow C[A[j]] + 1$ $// C[i] = |\{key = i\}|$

for $i \leftarrow 2$ to k do $C[i] \leftarrow C[i] + C[i-1]$ $// C[i] = |\{key \le i\}|$

for $j \leftarrow n$ downto 1 do $B[C[A[i]]] \leftarrow A[i]$ $C[A[j]] \leftarrow C[A[j]] - 1$ <u>Step 4</u>: Populate the output array *There are* C[3] = 2 *elts that are* ≤ 3 4 3 A: 1 3 4 3 4 5 2 3 4 **B**: 1 2 3 4 1

4

CS 473 – Lecture 9

Cevdet Aykanat and Mustafa Ozdal Computer Engineering Department, Bilkent University

C:

1

for i ← 1 to k do
 C[i] ← 0
for j ← 1 to n do
 C[A[j]] ← C[A[j]] + 1
// C[i] = |{key = i}|

for $i \leftarrow 2$ to k do $C[i] \leftarrow C[i] + C[i-1]$ $// C[i] = |\{key \le i\}|$

for j ← n **downto** 1 **do** B[C[A[j]]] ← A[j] C[A[j]] ← C[A[j]] - 1 <u>Step 4</u>: Populate the output array

There are C[1] = 1 *elts that are* ≤ 1



CS 473 – Lecture 9

for i ← 1 to k do
 C[i] ← 0
for j ← 1 to n do
 C[A[j]] ← C[A[j]] + 1
// C[i] = |{key = i}|

for i ← 2 **to** k **do** C[i] ← C[i] + C[i-1] // C[i] = |{key ≤ i}|

for j ← n **downto** 1 **do** B[C[A[j]]] ← A[j] C[A[j]] ← C[A[j]] - 1 <u>Step 4</u>: Populate the output array

There are C[4] = 4 *elts that are* ≤ 4



CS 473 – Lecture 9

Counting Sort: Runtime Analysis



CS 473 – Lecture 9

Counting Sort: Runtime

- Runtime is $\Theta(n+k)$
- If k = O(n), then counting sort takes $\Theta(n)$
- <u>*Question*</u>: We proved a lower bound of Θ(nlgn) before! Where is the fallacy?
- □ <u>Answer</u>:
 - \Box $\Theta(nlgn)$ lower bound is for comparison-based sorting
 - Counting sort is not a comparison sort
 - □ In fact, not a single comparison between elements occurs!

Stable Sorting

- Counting sort is a <u>stable sort</u>: It preserves the input order among equal elements.
 - i.e. The numbers with the same value appear in the output array in the same order as they do in the input array.



Exercise: Which other sorting algorithms have this property?

CS 473 – Lecture 9	Cevdet Aykanat and Mustafa Ozdal
	Computer Engineering Department, Bilkent University

Radix Sort

- Origin: Herman Hollerith's card-sorting machine for the 1890 US Census.
- Basic idea: Digit-by-digit sorting
- Two variations:
 - □ Sort from MSD to LSD (bad idea)
 - □ Sort from LSD to MSD (good idea)
 - ISD/MSD: Least/most significant digit

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 (IBM).

Hollerith Punched Card



<u>*Punched card*</u>: A piece of stiff paper that contains digital information represented by the presence or absence of holes.

"Modern" IBM card

One character per column



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

CS 473 – Lecture 9

Hollerith Tabulating Machine and Sorter



- Mechanically sorts the cards based on the hole locations.
- Sorting performed for one column at a time
- Human operator needed to load/retrieve/move cards at each stage

- Sort starting from the most significant digit (MSD)
- Then, sort each of the resulting bins recursively
- At the end, combine the decks in order



□ To sort a subset of cards recursively:

- All the other cards need to be removed from the machine, because the machine can handle only one sorting problem at a time.
- The human operator needs to keep track of the intermediate card piles



CS 473 – Lecture 9

MSD-first sorting may require:

- -- very large number of sorting passes
- -- very large number of intermediate card piles to maintain
- S(d): # of passes needed to sort d-digit numbers (worst-case)
 Recurrence:

S(d) = 10 S(d-1) + 1 with S(1) = 1

<u>Reminder</u>: Recursive call made to each subset with the same most significant digit (MSD)

CS 473 – Lecture 9

Hollerith's MSD-First Radix Sort Recurrence: S(d) = 10S(d-1) + 1

$$\begin{split} S(d) &= 10 \ S(d-1) + 1 \\ &= 10 \ (10 \ S(d-2) + 1) + 1 \\ &= 10 \ (10 \ (10 \ S(d-3) + 1) + 1) + 1 \\ &= 10^i \ S(d-i) + 10^{i-1} + 10^{i-2} + \ldots + \ 10^1 + 10^0 \end{split}$$

Iteration terminates when i = d-1 with S(d-(d-1)) = S(1) = 1

CS 473 – Lecture 9

P(d): # of intermediate card piles maintained (worst-case) <u>Reminder</u>: Each routing pass generates 9 intermediate piles except the sorting passes on least significant digits (LSDs) There are 10^{d-1} sorting calls to LSDs P(d) = 9 (S(d) - 10^{d-1}) = 9 (($10^d - 1$)/9 - 10^{d-1}) = ($10^d - 1 - 9 \cdot 10^{d-1}$) = $10^{d-1} - 1$ P(d) = $10^{d-1} - 1$

Alternative solution: Solve the recurrence:

P(d) = 10P(d-1) + 9P(1) = 0

- Example: To sort 3 digit numbers, in the worst case:
 S(d) = (1/9) (10³-1) = 111 sorting passes needed
 P(d) = 10^{d-1}-1 = 99 intermediate card piles generated
- MSD-first approach has more recursive calls and intermediate storage requirement
 - Expensive for a "tabulating machine" to sort punched cards
 - Overhead of recursive calls in a modern computer

LSD-First Radix Sort

- Least significant digit (LSD)-first radix sort seems to be a folk invention originated by machine operators.
- It is the counter-intuitive, but the better algorithm.
- Basic algorithm:

Sort numbers on their LSD first

Stable sorting needed!!!

Combine the cards into a single deck in order

Continue this sorting process for the other digits

from the LSD to MSD

Requires only d sorting passesNo intermediate card pile generated

CS 473 – Lecture 9

LSD-first Radix Sort: Example



Correctness of Radix Sort (LSD-first)

Proof by induction:**Base case**: d=1 is correct (trivial)Inductive hyp:Assume the first d-1 digits are sorted correctlyProve that all d digits are sorted correctly after sorting digit d



Radix Sort: Runtime

Use counting-sort to sort each digit
 <u>Reminder</u>: Counting sort complexity: Θ(n+k)

 n: size of input array
 k: the range of the values

- Radix sort runtime: \O(d(n+k))
 d: # of digits
- How to choose the d and k?

Radix Sort: Runtime – Example 1

- We have flexibility in choosing d and k
- Assume we are trying to sort 32-bit words
 - We can define each digit to be 4 bits
 - □ Then, the range for each digit $k = 2^4 = 16$ So, counting sort will take $\Theta(n+16)$
 - The number of digits d = 32/4 = 8
 - □ Radix sort runtime: $\Theta(8(n+16)) = \Theta(n)$

Radix Sort: Runtime – Example 2

- \square We have flexibility in choosing d and k
- Assume we are trying to sort 32-bit words
 - Or, we can define each digit to be 8 bits
 - □ Then, the range for each digit $k = 2^8 = 256$ So, counting sort will take $\Theta(n+256)$
 - The number of digits d = 32/8 = 4
 - □ Radix sort runtime: $\Theta(4(n+256)) = \Theta(n)$



Radix Sort: Runtime

- Assume we are trying to sort **b**-bit words
 - Define each digit to be r bits
 - □ Then, the range for each digit $k = 2^r$ So, counting sort will take $\Theta(n+2^r)$
 - The number of digits d = b/r
 Radix sort runtime:

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

b/r digits

 r bits
 r bits
 r bits

 b-bit

Radix Sort: Runtime Analysis

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

Minimize T(n, b) by differentiating and setting to 0 Or, intuitively:

We want to balance the terms (b/r) and $(n + 2^r)$

<u>Choose $r \approx lgn$ </u>

If we choose $r \ll \lg n \square (n + 2^r)$ term doesn't improve If we choose $r \gg \lg n \square (n + 2^r)$ increases exponentially

Radix Sort: Runtime Analysis

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

Choose
$$r = lgn$$
 \square $T(n, b) = \Theta(bn/lgn)$

For numbers in the range from 0 to n^d − 1, we have: The number of bits b = lg(n^d) = d lgn
Radix sort runs in Θ(dn)

CS 473 – Lecture 9

Radix Sort: Conclusions

Choose
$$r = lgn$$
 \square $T(n, b) = \Theta(bn/lgn)$

Example: Compare radix sort with merge sort/heapsort 1 million (2^{20}) 32-bit numbers ($n = 2^{20}$, b = 32)

<u>Radix sort</u>: [32/20] = 2 passes

<u>Merge sort/heap sort</u>: lgn = 20 passes

Downsides:

Radix sort has little locality of reference (more cache misses) The version that uses counting sort is not in-place

 On modern processors, a well-tuned quicksort implementation typically runs faster.