Analysis of Algorithms

COMP.4040, Summer 2019

Chapter 8: Sorting in Linear Time

By: Sirong Lin, PhD

University of Massachusetts Lowell

UMASS LOWELL
Learning with Purpose

Announcements

- a current total score was calculated on Blackboard
  - homework 30% (4 hw)
  - quizzes: 70% (3)
- Please revisit your class goal and adjust your learning strategy if needed
- Withdrawal deadline is June 14

Announcement

Homework 7: Due June 20 (Th)

Homework 10: Due June 24 (M)

Blackboard, e-version

in class, paper-version

Outline

- Lower bounds for (comparison-based) sorting
- Sorting in Linear Time (non-comparison based)
  - Counting Sort
  - Radix Sort
  - Bucket Sort

Sorting Introduction

Sorting Algorithms Comparison

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n\lg n)$	$\Theta(n\lg n)$
Heapsort	$O(n\lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n\lg n)$ (expected)

What is a common property of these sorting algorithm?

Can we do better?

Sorting Algorithms Comparison

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n\lg n)$	$\Theta(n\lg n)$
Heapsort	$O(n\lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n\lg n)$ (expected)
Counting sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Lower bounds for (comparison-based) sorting

Two models of Sorting & Searching

Comparison Model and their lower bound

- Sorting: insertion, selection, merge, quick, and heap sort
  - lower bound: $\Omega(n\lg n)$
- Searching: binary search
  - lower bound: $\Omega(\lg n)$

Non-comparison model

- we can do linear time, sometimes

Comparison Model

restriction in this model: only operations allowed are *comparisons*

gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$ by comparisons between elements

input: all input items are ADT (abstract data type) (it contains a key that comparable, and it may contain some values and operations)

time cost: # of comparisons

Decision Tree

any comparison algorithm can be viewed as a tree of (1) all possible comparisons and their outcomes and (2) resulting answer (for any particular of n inputs)

abstracts away everything else, e.g., control, data movement

example 1: binary search, for n = 3 (notes)

Decision Tree & an algorithm

see notes

- draw 1 tree for each n
- the algorithm splits in two at each node
- the tree models all possible execution traces

Binary Search Lower Bound

Theorem: n (preprocessed) items, finding a given item in comparison model requires $\Omega(\lg n)$

Proof:

decision tree is binary

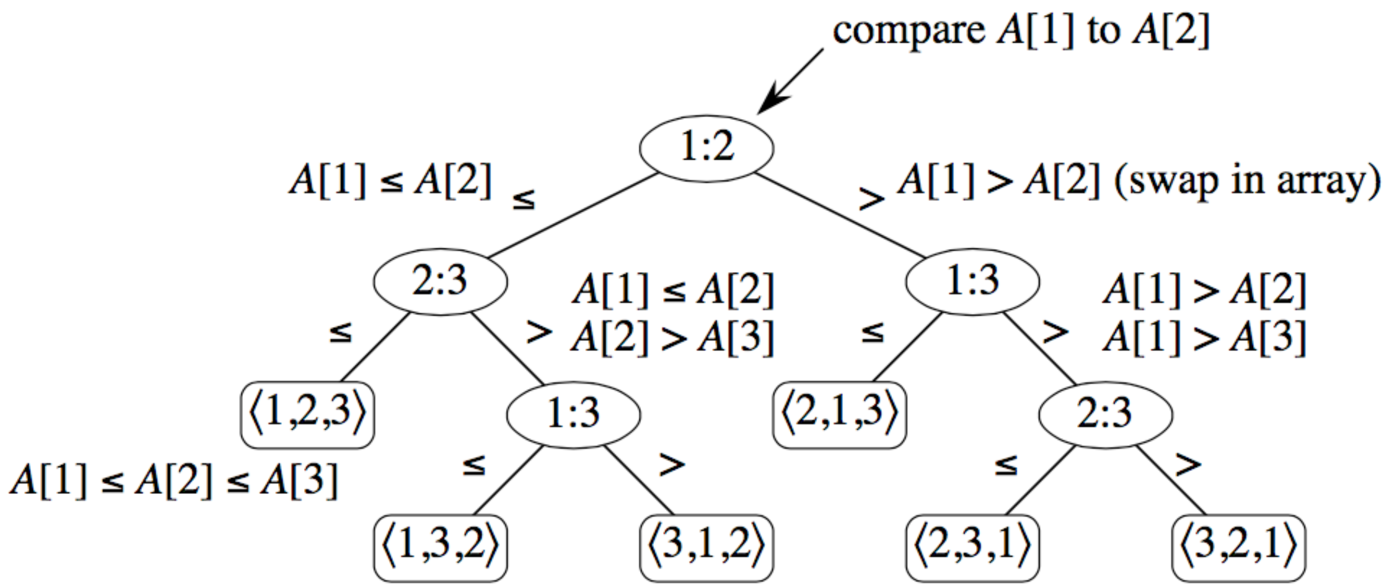
have at least *n* leaves, one for each result

height of the tree $\geq \lg n$

in comparison model: binary search is optimal

Decision Tree — Sorting

example 2: sort 3 elements



shows all comparisons (internal nodes) and possible results (leaves), but binary branching everywhere, lots of nodes

Decision Tree — Sorting (Cont'd)

the height of the tree represents the worst-case running time, depends on algorithms

insertion sort: $\Theta(n^2)$

merge sort: $\Theta(n \lg n)$

Decision Tree — Sorting (Cont'd)

Theorem: sort n elements in comparison model requires $\Omega(n\lg n)$ in worst case.

Proof:

decision tree is binary

total # of leaves: at least *n!* (because each permutation appears at least once)

height of the tree $\geq \lg(n!) = \Omega(n\lg n)$ (equation 3.19)

can be proved by using Stirling's approximation (equation 3.18) (notes)

or the summation of logarithm

Lower bound of sorting

What does $\Omega(n\lg n)$ means to us?

- this means that for any comparison based sorting, at least *nlgn* in worst-case running time
- heap-sort and merge-sort are asymptotically optimal comparison sort algorithms

Non-comparison Model

Any ideas of sorting without comparisons?

Sorting in Linear time

Counting Sort

Radix Sort

Bucket Sort

Non-comparison Model

RAM (Random Access Machine) model

- memory is in array, can access anything in the array in constant time, e.g., A[**key**] is $\Theta(1)$
- sort in linear time (sometimes) using the power of RAM

Linear-time Sorting

Linear-time sorting (Integer sorting)

- assume sort n keys, and each key are integers in [0, k] (k positive integer)
- perform operations rather than comparisons to integers (and decide the order)
- for k __________ can sort in O(n) time

Counting Sort

Intuition: count all the items and place them into their position directly

Example: sort <2, 5, 3, 0, 2, 3, 0, 3>

There are two 0s, two 2s, three 3s, and one 5

Counting Sort (Cont'd)

Implementation:

- input: array A[1..n], where A[j] ∈ {0, 1, 2, …, k} for j = 1 to n
- Auxiliary storage: C to store the result of counting
- Output: array B [1..n], sorted
  - how to output the items?
  - traverse the array of counters (C) and the array is already written in order by keys

Counting Sort (Cont’d)

an important sorting property — **stable**

numbers with the same value appear in the output array in the same order as they do in the input array

In counting sort, we care about the order of the outputs (because each input item may have some “satellite data” together with the key)

Are Insertion sort, Merge sort, Quick sort, Heap sort stable?

Counting Sort (Cont’d)

Algorithm:

COUNTING-SORT(A, B, n, k)

let $C[0..k]$ be a new array
for $i = 0$ **to** k
    $C[i] = 0$
for $j = 1$ **to** n
    $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k
    $C[i] = C[i] + C[i-1]$
for $j = n$ **downto** 1
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

Counting Sort (Cont’d)

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

Line 4~5 counts the occurrence of each key in array A and saved the counts into array C

	0	1	2	3	4	5
C	2	0	2	3	0	1

Line 6~7 counts how many number of keys are less than or equal to each individual key. the number is also the the initial position for that key in the output array (if the key exists in the input array)

	0	1	2	3	4	5
C	2	2	4	7	7	8

Counting Sort (Cont'd)

Analysis of running time:

```
COUNTING-SORT(A, B, n, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0                          Θ(k)
for j = 1 to n
    C[A[j]] = C[A[j]] + 1             Θ(n)
for i = 1 to k
    C[i] = C[i] + C[i - 1]            Θ(k)
for j = n downto 1
    B[C[A[j]]] = A[j]                 Θ(n)
    C[A[j]] = C[A[j]] - 1
```

Counting Sort (Cont'd)

Counting Sort:

step 1: calculate the distribution of the keys

step 2: put them in order

can use a list implementation

Analysis:

$T(n) = \Theta(k) + \Theta(n) + \Theta(k) + \Theta(n)$

$= \Theta(n+k)$

if $k = O(n)$, $T(n)$ is $\Theta(n)$, beats the lower bound of comparison sort

Counting Sort (Cont'd)

$T(n) = \Theta(n+k)$

then how big *k* is practical (32-bit, 16-bit, 8-bit)?

good for sorting 32-bit values? NO

16-bit? Probably not

8-bit? Maybe, depending on n

4-bit? Probably

Counting sort is usually used in radix sort

Sorting in Linear time

- Counting Sort
- **Radix Sort**
- Bucket Sort

Radix

the **radix** or **base** is the number of unique digits, including the digit zero, used to represent numbers in a positional numeral system

Radix Sort

Key idea: Sort digit-by-digit

sort integers by least significant digit

….

sort integers by most significant digit

Radix Sort (Cont'd)

Algorithm:

RADIX-SORT(A, d)

1 **for** $i = 1$ **to** d

2 use a stable sort to sort array A on digit i

example: see notes

Radix Sort (Cont'd)

Key idea: Sort digit-by-digit, # of digits is d

- sort integers by least significant digit
- ….
- sort integers by most significant digit

extract each digit:

- in constant time using "divide" and "mod"

sort each digit using counting sort

- Θ(n+k'), where k' is the possible largest value of one digit (ranges from 0 to k' inclusive)

Total time: Θ(d(n+k'))

Radix Sort — Analysis

Proof and Analysis: see notes

Total time: $\Theta(d(n+k'))$

$\Theta(bn/\lg n)$ if an integer has *b* bits

It runs in linear time: Θ (cn), when the values of the input ranges in $[0..n^c]$, and c is not a very large positive number

Radix Sort (Cont'd)

Is Radix Sort preferable to a comparison-based sorting algorithm, e.g., Quicksort?

Depends on the characteristics of implementations, hardware, and input data. Here are some disadvantages:

- There is a hidden constant factor in Radix Sort running time, e.g., HW2, compare nlgn vs. 256n
- Radix Sort uses Counting Sort, which is not "in place" sorting and requires extra space
- Radix Sort doesn't work well on cache

Radix Sort (Cont'd)

Linear-time sorting (Integer sorting)

- assume sort n keys sorting, and each key are integers in [0, k] (k positive integer)
- perform operations rather than comparisons to integers (and decide the order)
- for k <u>in the polynomial of n</u> we, can sort in O(n) time

Sorting in Linear time

Counting Sort

Radix Sort

Bucket Sort

Bucket Sort

Key idea:

Assumes the input is generated by a random process that distributes elements uniformly over [0, 1)

- divide [0, 1) into *n* equal-sized buckets
- distribute the n input values into the buckets
  - using a function of key values to index into an array
- sort each bucket (with a linked-list)
- go through buckets in order, listing elements in each one

Bucket Sort (Cont'd)

Example: Sort array A (10 elements)

<0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68>

see notes for details

Bucket Sort (Cont'd)

Algorithm:

```
BUCKET-SORT(A, n)
let B[0 .. n − 1] be a new array
for i = 1 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊n · A[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate lists B[0], B[1], . . . , B[n − 1] together in order
return the concatenated lists
```

Bucket Sort (Cont'd)

Analysis:

relies on no bucket getting too many values

Intuitively, if each bucket gets a constant number of elements, it takes O(1) time to sort each bucket, so O(n) sort time for all buckets

We "expect" each bucket to have few elements, since the average is 1 element per bucket.

Bucket Sort (Cont'd)

Probabilistic Analysis: Insertion Sort runs in quadratic time

n_i = the # of elements placed in bucket B[i]

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

$$\begin{aligned} \mathrm{E}[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\ &= \Theta(n) + O(n) \\ &= \Theta(n) \end{aligned}$$

Bucket Sort (Cont'd)

What is the worst-case running time for bucket sort? When does it happen? (Homework)

Chapter Summary

Lower bound of Comparison-based Searching/ Sorting Algorithm

What do we mean a sorting algorithm is **stable**?

- numbers with the same value appear in the output array in the same order as they do in the input array

Stability of the sorting algorithms we learned. Why or why not?

Chapter Summary (Cont'd)

Three non-comparison based sorting algorithms in linear time (Counting Sort, Radix Sort, Bucket Sort)

- how does each one work, running time
- when can we achieve the linear time, when can't (when to use these?)
- Compare with other comparison-based sorting algorithms. Understand the usage of all sorting algorithms