

Radix Sort

Sorting in the C++ STL

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, October 19, 2009

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2009 Glenn G. Chappell

Unit Overview

Algorithmic Efficiency & Sorting

Major Topics

- ✓ ■ Introduction to Analysis of Algorithms
- ✓ ■ Introduction to Sorting
- ✓ ■ Comparison Sorts I
- ✓ ■ More on Big- O
- ✓ ■ The Limits of Sorting
- ✓ ■ Divide-and-Conquer
- ✓ ■ Comparison Sorts II
- ✓ ■ Comparison Sorts III
 - Radix Sort
 - Sorting in the C++ STL

Review

Introduction to Analysis of Algorithms

Efficiency

- General: using few resources (time, space, bandwidth, etc.).
- Specific: fast (time).

Analyzing Efficiency

- Determine how the **size of the input** affects running time, measured in **steps**, in the **worst case**.

Scalable: works well with large problems.

	Using Big-O	In Words	
	$O(1)$	Constant time	
	$O(\log n)$	Logarithmic time	
	$O(n)$	Linear time	
	$O(n \log n)$	Log-linear time	
..... Probably not scalable ↓	$O(n^2)$	Quadratic time	
	$O(b^n)$, for some $b > 1$	Exponential time	

Cannot read all of input ↑

Faster ↑

Slower ↓

Review

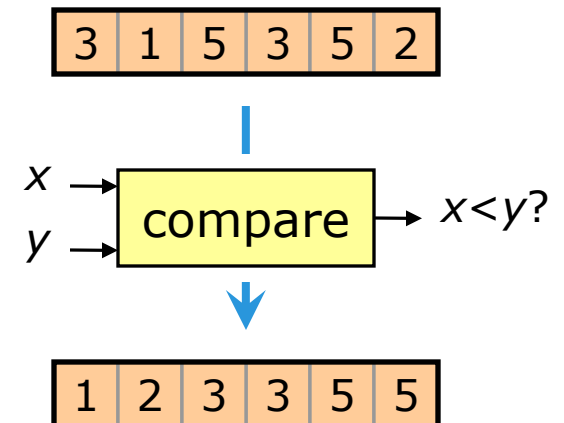
Introduction to Sorting — Basics

Sort: Place a collection of data in order.

Key: The part of the data item used to sort.

Comparison sort: A sorting algorithm that gets its information by comparing items in pairs.

A **general-purpose comparison sort** places no restrictions on the size of the list or the values in it.



Review

Introduction to Sorting — Overview of Algorithms

There is no *known* sorting algorithm that has all the properties we would like one to have.

We will examine a number of sorting algorithms. Most of these fall into two categories: $O(n^2)$ and $O(n \log n)$.

- Quadratic-Time [$O(n^2)$] Algorithms
 - ✓ ▪ Bubble Sort
 - ✓ ▪ Insertion Sort
 - ✓ ▪ Quicksort
 - Treesort (later in semester)
- Log-Linear-Time [$O(n \log n)$] Algorithms
 - ✓ ▪ Merge Sort
 - (some) ▪ Heap Sort (mostly later in semester)
 - ✓ ▪ Introsort (not in the text)
- Special Purpose — Not Comparison Sorts
 - Pigeonhole Sort
 - Radix Sort

Review

Comparison Sorts II — Merge Sort

Merge Sort splits the data in half, recursively sorts each half, and then merges the two.

Stable Merge

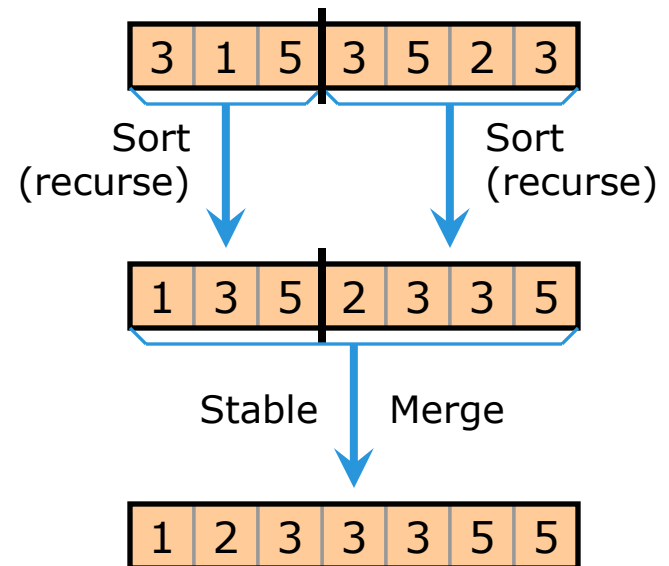
- Linear time, stable.
- In-place for Linked List. Uses buffer [$O(n)$ space] for array.

Analysis

- Efficiency: $O(n \log n)$. Average same. 😊
- Requirements on data: Works for Linked Lists, etc. 😊
- Space Efficiency: $O(\log n)$ space for Linked List. Can eliminate recursion to make this in-place. $O(n)$ space for array. 😊/😊/😞
- Stable: Yes. 😊
- Performance on Nearly Sorted Data: Not better or worse. 😊

Notes

- Practical & often used.
- Fastest known for (1) stable sort, (2) sorting a Linked List.
- Good standard for judging sorting algorithms



Review

Comparison Sorts III — Quicksort: Introduction, Partition

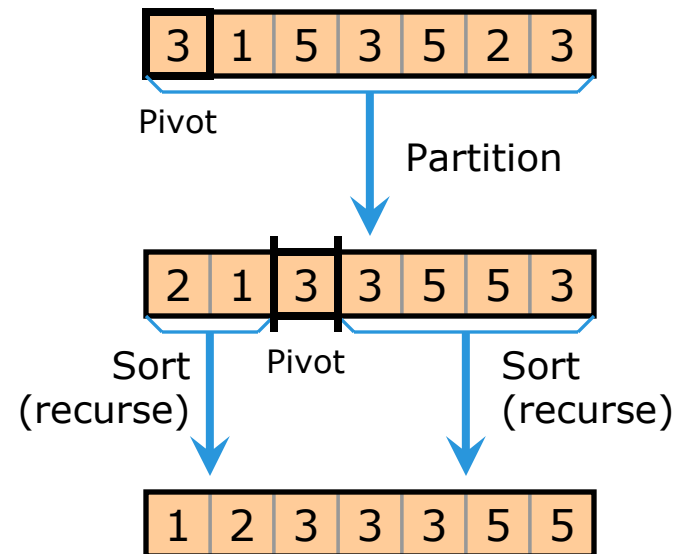
Quicksort is another divide-and-conquer algorithm. Procedure:

- Choose a list item (the **pivot**).
- Do a **Partition**: put items less than the pivot before it, and items greater than the pivot after it.
- Recursively sort two sublists: items before pivot, items after pivot.

We did a simple pivot choice: the first item. Later, we improve this.

Fast Partition algorithms are in-place, but not stable.

- Note: In-place Partition does not give us an in-place Quicksort. Quicksort uses memory for recursion.



Review

Comparison Sorts III — Better Quicksort: Optimizations

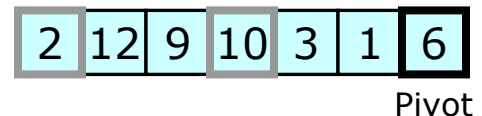
Unoptimized Quicksort is slow (quadratic time) on nearly sorted data and uses a lot of space (linear) for recursion.

We discussed three optimizations:

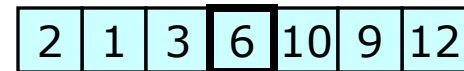
- Median-of-three pivot selection.
 - Improves performance on most nearly sorted data.
 - Requires random-access data.
- Tail-recursion elimination on the larger recursive call.
 - Reduces space usage to logarithmic.
- Do not sort small sublists; finish with Insertion Sort.
 - General speed up.
 - May adversely affect cache hits.

Median-of-three

Initial State:



After Partition:



With these optimizations, Quicksort is still $O(n^2)$ time.

Review

Comparison Sorts III — Better Quicksort: Analysis of Quicksort

Efficiency ☹️

- Quicksort is $O(n^2)$.
- Quicksort has a **very** good $O(n \log n)$ average-case time. 😊😊

Requirements on Data ☹️

- Non-trivial pivot-selection algorithms (median-of-3 and others) are only efficient for random-access data.

Space Usage ☹️

- Quicksort uses space for recursion.
 - Additional space: $O(\log n)$, if clever tail-recursion elimination is done.
 - Even if **all** recursion is eliminated, $O(\log n)$ additional space is still used.
 - This additional space need not hold any data items.

Unlike
Merge Sort



Stability ☹️

- Efficient versions of Quicksort are not stable.

Performance on Nearly Sorted Data ☹️

- An unoptimized Quicksort is **slow** on nearly sorted data: $O(n^2)$.
- Quicksort + median-of-3 is $O(n \log n)$ on most nearly sorted data.

Review

Comparison Sorts III — Introsort: Description

In 1997, David Musser found out how to make Quicksort log-linear time.

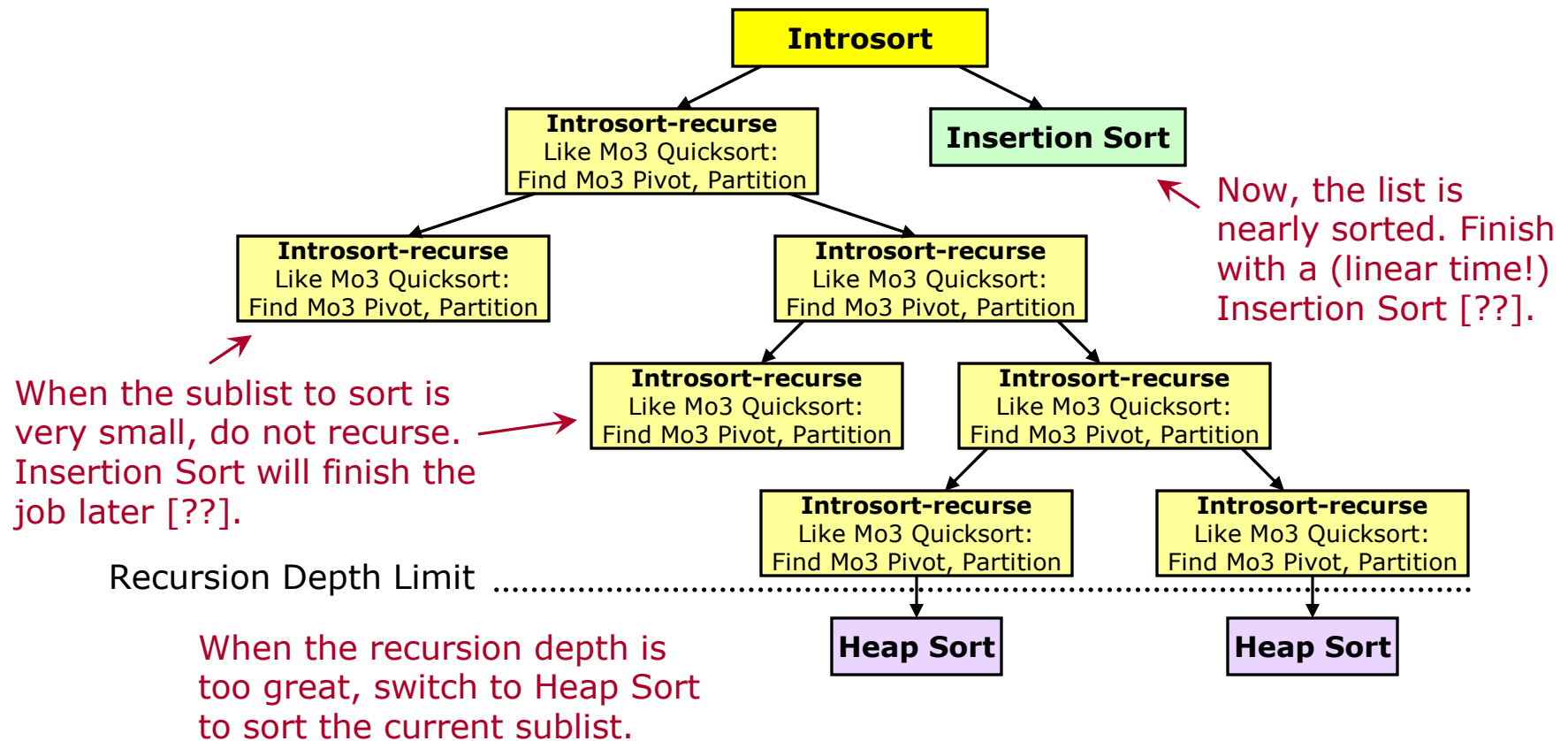
- Keep track of the recursion depth.
- If this exceeds some bound (recommended: $2 \log_2 n$), then switch to Heap Sort for the current sublist.
 - Heap Sort is a general-purpose comparison sort that is log-linear time and in-place. We will discuss it in detail later in the semester.
- Musser calls this technique “introspection”. Thus, introspective sorting, or **Introsort**.

Review

Comparison Sorts III — Introsort: Diagram

Here is an illustration of how Introsort works.

- In practice, the recursion will be deeper than this.
- The Insertion-Sort call *might* not be done, due to its effect on cache hits.



Review

Comparison Sorts III — Introsort: Analysis

Efficiency 😊😊

- Introsort is $O(n \log n)$.
- Introsort also has an average-case time of $O(n \log n)$ [of course].
 - Its average-case time is just as good as Quicksort. 😊😊

Requirements on Data 😞

- Introsort requires random-access data.

Space Usage 😞

- Introsort uses space for recursion (or simulated recursion).
 - Additional space: $O(\log n)$ — even if all recursion is eliminated.
 - This additional space need not hold any data items.

Stability 😞

- Introsort is not stable.

Performance on Nearly Sorted Data 😞

- Introsort is not significantly faster or slower on nearly sorted data.

Radix Sort Background

We have looked in detail at five general-purpose comparison sorts. Now we look at two sorting algorithms that do not use a comparison function:

- Pigeonhole Sort.
- Radix Sort.

Later in the semester, we will look closer at Heap Sort, which *is* a general-purpose comparison sort, but which can also be conveniently modified to handle other situations.

Radix Sort

Preliminaries: Pigeonhole Sort — Description

Suppose we have a list to sort, and:

- Keys lie in a **small fixed set of values.**
- Keys can be used to **index an array.**
 - E.g., they might be small-ish nonnegative integers.

← **Not** general-purpose

← Not even a comparison sort

Procedure

- Make an array of empty lists (**buckets**), one for each possible key.
- Iterate through the given list; insert each item at the end of the bucket corresponding to its value.
- Copy items in each bucket, in order, back to the original list.

Time efficiency: **linear time**, if written properly.

- How is this possible? Answer: We are not doing general-purpose comparison sorting. Our $\Omega(n \log n)$ bound does not apply.

This algorithm is often called **Pigeonhole Sort**.

- Not applicable to many situations; requires a limited set of keys.
- Pigeonhole Sort is stable, and uses linear additional space.

Radix Sort

Preliminaries: Pigeonhole Sort — Write It

TO DO

- Write a function to do Pigeonhole Sort.

*Done. See `pigeonhole_sort.cpp`,
on the web page.*

Radix Sort Description

Based on Pigeonhole Sort, we can design a useful algorithm: **Radix Sort**.
Suppose we want to sort a list of **strings** (in some sense):

- Character strings.
- Numbers, considered as strings of digits.
- Short-ish sequences of some other kind.

Call the entries in a string “**characters**”.

- These need to be valid keys for Pigeonhole Sort.
- In particular, we must be able to use them as array indices.

The algorithm will arrange the list in **lexicographic order**.

- This means sort first by first character, then by second, etc.
- For strings of letters, this is alphabetical order.
- For positive integers (padded with leading zeroes), this is numerical order.

Radix Sort Procedure

- Pigeonhole Sort the list using the **last** character as the key.
- Take the list resulting from the previous step and Pigeonhole Sort it, using the **next-to-last** character as the key.
- Continue ...
- After re-sorting by **first** character, the list is sorted.

Radix Sort Example

Here is the list to be sorted.

- **583 508 183 90 223 236 924 4 426 106 624**

We first sort them by the units digit, using Pigeonhole Sort.

Nonempty buckets
are underlined

- **90 583 183 223 924 4 624 236 426 106 508**

Then Pigeonhole Sort again, based on the tens digit, in a stable manner (note that the tens digit of 4 is 0).

- **4 106 508 223 924 624 426 236 583 183 90**

Again, using the hundreds digit.

- **4 90 106 183 223 236 426 508 583 624 924**

And now the list is sorted.

Radix Sort

Write It, Comments

TO DO

- Write Radix Sort for small-ish positive integers.

*Done. See `radix_sort.cpp`,
on the web page.*

Comments

- Radix Sort makes very strong assumptions about the values in the list to be sorted.
- It requires linear additional space.
- It is stable.
- It does not perform especially well or badly on nearly sorted data.
- Of course, what we really care about is speed. *See the next slide.*

Radix Sort

Efficiency [1/2]

How Fast is Radix Sort?

- Fix the number of characters and the character set.
- Then each sorting pass can be done in linear time.
 - Pigeonhole Sort with one bucket for each possible character.
- And there are a fixed number of passes.
- Thus, Radix Sort is $O(n)$: **linear time**.

How is this possible?

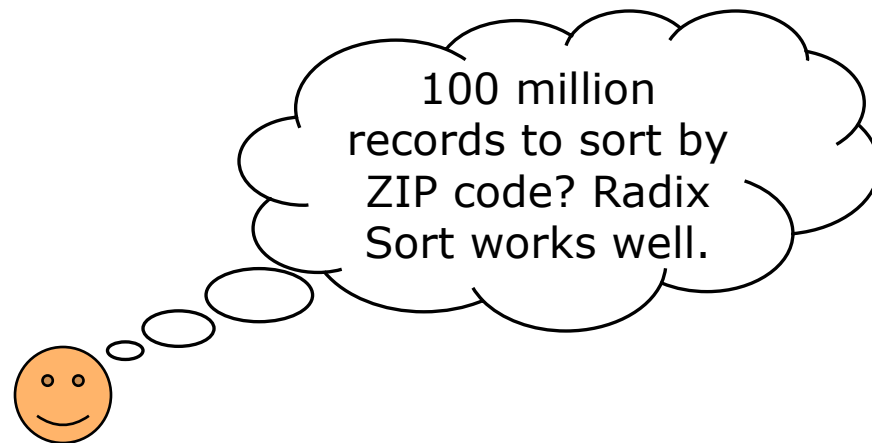
- Radix Sort is a sorting algorithm. However, again, it is neither general-purpose nor a comparison sort.
 - It places restrictions on the values to be sorted: not general-purpose.
 - It gets information about values in ways other than making a comparison: not a comparison sort.
- Thus, our argument showing that $\Omega(n \log n)$ comparisons were required in the worst case, does not apply.

Radix Sort Efficiency [2/2]

In practice, Radix Sort is not really as fast as it might seem.

- There is a hidden logarithm. The number of passes required is equal to the length of a string, which is something like the logarithm of the number of possible values.
- The number of passes is fixed, since we limit the length of a string. This limits the number of possible values in the list.
- However, if we consider Radix Sort applied to a list in which *all the values might be different*, then it is in the same efficiency class as normal sorting algorithms.

In certain special cases (e.g., big lists of small numbers) Radix Sort can be useful.



Sorting in the C++ STL

Specifying the Interface

Iterator-based sorting functions can be specified two ways:

- Given a range
 - "last" is actually just past the end, as usual.

```
template<typename Iterator>  
void sortIt(Iterator first, Iterator last);
```

- Given a range and a comparison.

```
template<typename Iterator, typename Ordering>  
void sortIt(Iterator first, Iterator last, Ordering compare);
```

"compare", above, should be something you can use to compare two values.

- "compare(va11, va12)" should be a legal expression, and should return a `bool`: true if `va11` comes before `va12` (think "less-than").
- So `compare` can be a function (passed as a function pointer).
- It can also be an object with `operator()` defined: a **function object**.

Sorting in the C++ STL

Overview of the Algorithms [1/4]

The C++ Standard Template Library has six sorting algorithms:

- Global function `std::sort`
- Global function `std::stable_sort`
- Member function `std::list<T>::sort`
- Global functions `std::partial_sort` and `partial_sort_copy`.
- Combination of two global functions: `std::make_heap` & `std::sort_heap`

We now look briefly at each of these.

Sorting in the C++ STL

Overview of the Algorithms [2/4]

Function `std::sort`, in `<algorithm>`

- Global function.
- Takes two random-access iterators and an optional comparison.
- $O(n^2)$, but has $O(n \log n)$ average-case.
 - This should become $O(n \log n)$ in the forthcoming revised C++ standard.
 - It is currently $O(n \log n)$ in good STL implementations.
- Not stable. $O(\log n)$ additional space used.
- Algorithm used:
 - Quicksort is what the standards committee was thinking.
 - Introsort is what good implementations now use.
 - Other algorithms (Heap Sort?) are possible, but unlikely.

Function `std::stable_sort`, in `<algorithm>`

- Global function.
- Takes two random-access iterators and an optional comparison.
- $O(n \log n)$.
- Stable. $O(n)$ additional space used.
- Algorithm used: probably Merge Sort, general sequence version.

Sorting in the C++ STL

Overview of the Algorithms [3/4]

Function `std::list<T>::sort`, in `<list>`

- Member function. Sorts only objects of type `std::list<T>`.
- Takes either no parameters or a comparison.
- $O(n \log n)$. Stable.
- Algorithm used: probably Merge Sort, Linked-List version.

Sorting in the C++ STL

Overview of the Algorithms [4/4]

We will look at the last two STL algorithms in more detail later in the semester, when we cover Priority Queues and Heaps:


- Functions `std::partial_sort` and `std::partial_sort_copy`, in `<algorithm>`
 - Global functions.
 - Take three random-access iterators and an optional comparison.
 - $O(n \log n)$. Not stable.
 - Solve a more general problem than comparison sorting.
 - Algorithm used: probably Heap Sort.
- Combination: `std::make_heap` & `std::sort_heap`, in `<algorithm>`
 - Both Global functions.
 - Both take two random-access iterators and an optional comparison.
 - Combination is $O(n \log n)$. Not stable.
 - Solves a more general problem than comparison sorting.
 - Algorithm used: Heap Sort.

Sorting in the C++ STL Using the Algorithms [1/2]

Algorithm `std::sort` is declared in the header `<algorithm>`.
Call it with two iterators:

```
vector<int> v;  
std::sort(v.begin(), v.end());  
    // Ascending order
```

Default constructor call.
We can only pass an
object, not a **type**.



Or use two iterators and a comparison:

```
std::sort(v.begin(), v.end(), std::greater<int>());  
    // Descending order
```

- Class template `std::greater` is defined in `<functional>`.
Use `std::stable_sort` similarly to `std::sort`.

Sorting in the C++ STL Using the Algorithms [2/2]

When sorting a `std::list`, use the `sort` member function:

```
#include <list>

std::list<int> myList;
myList.sort();           // Ascending order
myList.sort(std::greater<int>()); // Descending order
```