# Non-Comparison Sorts
# Sorting in the C++ STL

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, October 7, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu



Some material contributed by Chris Hartman

# Unit Overview
# Algorithmic Efficiency & Sorting

## Major Topics

- ✓ Analysis of Algorithms
- ✓ Introduction to Sorting
- ✓ Comparison Sorts I
- ✓ Asymptotic Notation
- ✓ Divide and Conquer
- ✓ Comparison Sorts II
- ✓ The Limits of Sorting
- ✓ Comparison Sorts III
- Non-Comparison Sorts
- Sorting in the C++ STL

# Review

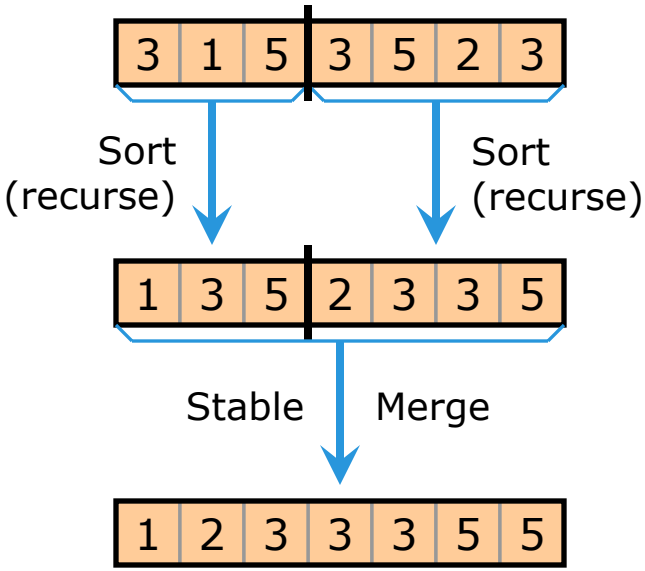
# Review
# Introduction to Sorting

## Sorting Algorithms Covered

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

# Comparison Sorts II — Merge Sort

**Merge Sort** splits the data in half, recursively sorts both, merges.




## Analysis

- Efficiency: $\Theta(n \log n)$. Avg same. ☺
- Requirements on Data: Works for Linked Lists, etc. ☺
- Space Efficiency: $\Theta(\log n)$ space for recursion. Iterative version is in-place for Linked List. $\Theta(n)$ space for array. 😐/☺/☹
- Stable: Yes. ☺
- Performance on Nearly Sorted Data: Not better or worse. 😐

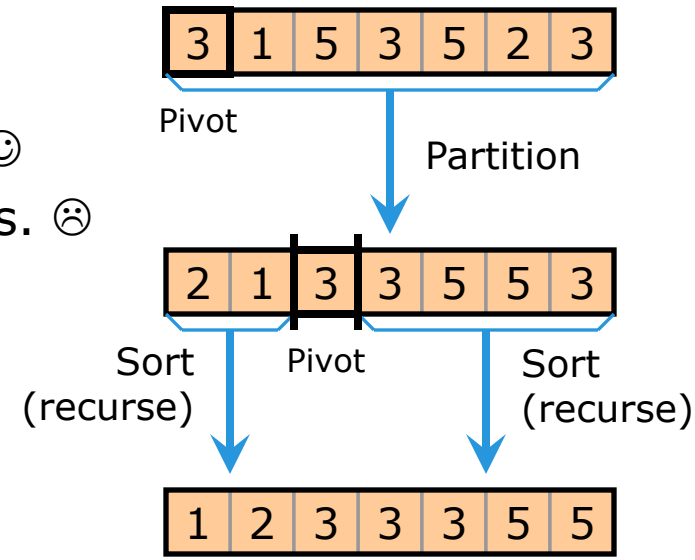
*See* `merge_sort.cpp`.

## Notes

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

# Review
# Comparison Sorts III — Quicksort, Better Quicksort

**Quicksort** chooses **pivot**, does **Partition**, recursively sorts parts.




Analysis

- Efficiency: $\Theta(n^2)$. ☹ Avg $\Theta(n \log n)$. ☺☺
- Requirements on Data: Random-access. ☹
- Space: $\Theta(\log n)$ with tail-recursion elimination. 😐
- Stable: No. ☹
- On Nearly Sorted Data: *mostly* $\Theta(n \log n)$ with Median-of-3. 😐

*See* `quicksort1.cpp`, `quicksort2.cpp`.

Common optimizations:

- Choose pivot with **Median-of-3** or similar.
- Make larger recursive call last, do tail-recursion elimination.
- Do not sort small sublists; finish with Insertion Sort (maybe).

# Review
# Comparison Sorts III — Introsort

The best optimization of all turns Quicksort into *Introsort*.

**Introspection**. An algorithm tracks its own performance. If this becomes poor, switch to an algorithm with a faster worst case.

**Heap Sort**

- Log-linear-time sort. In place.
- To be discussed in detail later in the semester.

Apply introspection to Quicksort to get **Introsort**.

- Track recursion depth; eliminated tail calls still count! If depth exceeds $2 \log_2 n$, then switch to Heap Sort for the current sublist.
- Worst case: $\Theta(n \log n)$. Average-case time as good as Quicksort.
- Other properties essentially the same as Quicksort.

# Non-Comparison Sorts

# Non-Comparison Sorts
## Pigeonhole Sort — Description

Suppose we are given a list to sort, and:

- Keys lie in a small fixed set of values.
- Keys can be used to index an array.

Keys might be small-ish nonnegative integers, characters, etc.

Procedure

- Make an array of lists—called **buckets**—one for each possible key. Each bucket holds items of the same type as those in the given list; it must be expandable to the size of the given list. Initialize each bucket to an empty list.
- Iterate through the given list; insert each item at the end of the bucket corresponding to its key.
- Copy items in each bucket, in order, back to the original list.

This algorithm has many names. One of them is **Pigeonhole Sort**.

# Non-Comparison Sorts
## Pigeonhole Sort — Data Structure

How should we store each bucket? We need to be able to:

- Insert a new item at the end of a bucket.
- **Traverse** (look at each item in) a bucket in forward order.

If each bucket is a (smart, resizable) array, then insert-at-end may require a slow **reallocate-and-copy**.

However, if we pre-allocate enough memory, then insert-at-end is constant time.

For `std::vector`, pre-allocate with member function `reserve`.

```
vector<Foo> vv;
vv.reserve(BIGSIZE);  // Does not change size of vv
```

This is a **space-time trade-off**.
For others, see CS 411.

# Non-Comparison Sorts
# Pigeonhole Sort — CODE

TO DO

- Look at an implementation of Pigeonhole Sort for small-ish positive integers.

*See* `pigeonhole_sort.cpp`.

Pigeonhole Sort is not very useful. But we can design a more useful sort based on it: *Radix Sort*.

First we analyze Pigeonhole Sort.

# Non-Comparison Sorts
# Pigeonhole Sort — Analysis

Efficiency ☺☺☺

- Pigeonhole Sort is $\Theta(n)$.

How can this be true? Didn't we prove it was impossible? *More on this coming up.*

- Pigeonhole Sort also has an average-case time of $\Theta(n)$ [obviously].

Requirements on Data ☹☹☹

- Pigeonhole Sort does not require random-access data.
- Pigeonhole Sort places *very* strong requirements on keys:
  - Keys must belong to a small, fixed set of values.
  - We must be able to index an array using keys.

Space Usage ☹

- Pigeonhole Sort requires an array of buckets: $\Theta(n)$ additional space.

Stability ☺

- Pigeonhole Sort is stable.

Performance on Nearly Sorted Data 😐

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

# Non-Comparison Sorts
## Radix Sort — Description [1/2]

Suppose we want to sort a list of short sequences of some kind:

- A list of short strings.
- A list of numbers, each number considered as a sequence of digits.
- A list of short-ish sequences of some other kind.

Requirements

- Each sequence must be no longer than some fixed length.
- The items in each sequence must be valid keys for Pigeonhole Sort.

We refer to a short sequence as a **string**. Entries are **characters**.

Our 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—with leading zeroes—this is numerical order.

# Non-Comparison Sorts
## Radix Sort — Description [2/2]

**Radix Sort** sorts a list of short sequences called *strings*. Each item in a string is called a *character*. The list is sorted in lexicographic order. The strings should all be the same length. If they are not, then **pad** the shorter strings with extra characters—or *treat them as if they are padded*.

Procedure

- Pigeonhole Sort the list using the *last* character of each string as the key.
- Take the list resulting from the previous step and Pigeonhole Sort it, using the next-to-last character as the key. This must be done in a stable manner.
- Then Pigeonhole Sort by the character before that, stably.
- And so on …
- After sorting by the *first* character, the list is in lexicographic order.

# Non-Comparison Sorts
## Radix Sort — Example

List to be sorted.

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

Treat each "string" as if it were a 3-digit number. So **4** is treated as **004**.

Nonempty buckets are underlined

First, Pigeonhole Sort by the units digit.

- **<u>90</u> <u>583 183 223</u> <u>924 4 624</u> <u>236 426 106</u> <u>508</u>**

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

- **<u>4 106 508</u> <u>223 924 624 426</u> <u>236</u> <u>583 183</u> <u>90</u>**

Again, based on the hundreds digit.

- **<u>4 90</u> <u>106 183</u> <u>223 236</u> <u>426</u> <u>508 583</u> <u>624</u> <u>924</u>**

And now the list is sorted.

# Non-Comparison Sorts
# Radix Sort — CODE

TO DO

- Look at an implementation of Radix Sort for positive integers with an upper limit on their value.

*See* `radix_sort.cpp`.

# Non-Comparison Sorts
# Radix Sort — Analysis [1/3]

## How Fast is Radix Sort?

Important!

- Fix the set of characters and the length of a string.
- Each sorting pass is a Pigeonhole Sort with one bucket for each possible character: $\Theta(n)$.
- And there are a fixed number of passes.
- Therefore, like Pigeonhole Sort, Radix Sort is $\Theta(n)$: linear time.

## How is this possible?

- Pigeonhole Sort and Radix Sort are sorting algorithms. However, they are *not* general-purpose comparison sorts.
  - Both place restrictions on the values to be sorted: not general-purpose.
  - Both get information about values in ways other than making a comparison: not comparison sorts.
- So our proof that $\Omega(n \log n)$ comparisons were required in the worst case, does not apply.

# Non-Comparison Sorts
# Radix Sort — Analysis [2/3]

Efficiency ☺☺☺

- Radix Sort is $\Theta(n)$—for strings of a fixed size.
- Radix Sort also has an average-case time of $\Theta(n)$ [obviously].

Requirements on Data ☹☹

- Radix Sort does not require random-access data.
- However, Radix Sort places strong requirements on keys:
  - Keys are strings (broadly defined) of at most some small, fixed length.
  - Characters (items in a "string") are legal keys for Pigeonhole Sort.
    - Characters belong to a small, fixed set of values.
    - We must be able to index an array using characters.

Space Usage ☹

- Radix Sort requires an array of buckets: $\Theta(n)$ additional space.

Stability ☺

- Radix Sort is stable.

Performance on Nearly Sorted Data 😐

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

# Non-Comparison Sorts
## Radix Sort — Analysis [3/3]

In practice, Radix Sort is not quite 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 a string can have.

So if we consider Radix Sort applied to a list in which *all the values can be different*, then the length of a string needs to be larger, for larger lists. Thought of in this way, Radix Sort lies in the same efficiency class as Merge Sort and Introsort.

But Radix Sort is still quite fast.

# Non-Comparison Sorts
## Radix Sort — Final Note

Lastly, Radix Sort is *easy to implement well*.

Why have we covered algorithms like Merge Sort and Introsort?

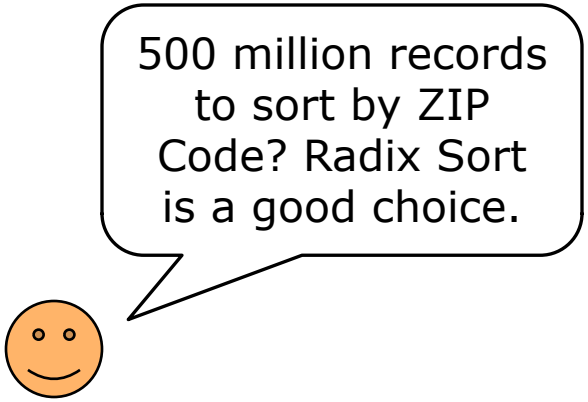
- So you will know how things work "under the hood", and you will be aware of issues like stability, recursion depth, etc.
- As examples of different ways to solve a single problem.
- As practice in analyzing algorithms.

But *not* because you will need to write them!

A top-notch Merge Sort or Introsort is a serious project. And it is typically already written; use your language's standard library!

But *you* can write a good Radix Sort.

And in some special cases, Radix Sort can be worth writing.



# Sorting in the C++ STL

# Sorting in the C++ STL
## Overview

The C++ STL includes seven sorting algorithms:

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

We briefly cover each of the seven.

Then we look at *lambda functions*, which can be used to specify a custom comparison.

Lastly, we look closer at how the first few algorithms are used.

# Sorting in the C++ STL In Brief [1/4]

All STL sorting algorithms are log-linear time, except where noted. All take an optional comparison as an additional argument.

Global function `std::sort` (`<algorithm>`)

- Takes a range: 2 random-access iterators.
- Not stable.
- Intended algorithm: Introsort.

Global function `std::stable_sort` (`<algorithm>`)

- Takes a range: 2 random-access iterators.
- Additional space: $\Theta(n)$*.
- *If sufficient space for a buffer cannot be allocated, then the time is allowed to be slower: $\Theta(n\ [\log n]^2)$.
- Intended algorithm: Merge Sort, with the general-sequence version of Stable Merge—or a slower in-place version of Stable Merge, if the buffer cannot be allocated.

# Sorting in the C++ STL In Brief [2/4]

Member function `sort` of `std::list<T>`

- Sorts the container it is called on.
- Takes no arguments.
- Stable.
- Intended algorithm: Merge Sort, with the Linked-List version of Stable Merge.

`std::list<T>` is a Doubly Linked List.

`std::forward_list<T>` is a Singly Linked List.

Member function `sort` of `std::forward_list<T>`

- Sorts the container it is called on.
- Takes no arguments.
- Stable.
- Intended algorithm: Merge Sort, with the Linked-List version of Stable Merge.

# Sorting in the C++ STL In Brief [3/4]

Global function `std::partial_sort` (`<algorithm>`)

- Takes 3 random-access iterators (*first*, *middle*, *last*).
- Not stable.
- Is more general than sorting a range. After call:
  - [*first*, *middle*) contains low items, in sorted order.
  - [*middle*, *last*) contains high items, in unspecified order.
- Intended algorithm: variant of Heap Sort.

The remaining STL sorts involve Heap Sort. We discuss these briefly now; we cover Heap Sort later in the semester.

Global function `std::partial_sort_copy` (`<algorithm>`)

- Takes 2 ranges: 4 iterators, last 2 must be random-access.
- Not stable.
- Is more general than sorting a range. After call:
  - Second range contains low items—as many as it can hold—from first range, in sorted order.
- Intended algorithm: variant of Heap Sort.

# Sorting in the C++ STL In Brief [4/4]

Combination of two global functions: `std::make_heap` & `std::sort_heap` (`<algorithm>`)

- Both take a range: 2 random-access iterators. This should be the same range for both function calls.
- Combination is $\Theta(n \log n)$ time. Not stable.
- Algorithm used: Heap Sort.

Again, all STL sorting algorithms take an optional comparison as an additional argument. Those optional comparisons can be specified conveniently using *lambda functions*.

*Next we look at these.*

# Sorting in the C++ STL
## Lambda Functions — Introduction

A **lambda function** is a function with no name.

In C++, create a lambda function as follows:

- Start with a pair of brackets: `[]`
- Then a normal function parameter list and function body.
- The return type is not required.

The term comes from the **Lambda Calculus** [Alonzo Church, 1930s], a mathematical formalism in which a function begins with the Greek letter lambda (λ).

A lambda function that takes two `int`s and returns their sum:

```
[](int a, int b)
{
    return a+b;
}
```

A C++ lambda function is technically an *object*, not a function. Officially, it is called a **lambda expression**. Or we can avoid the issue and just call it a **lambda**.

Lambda functions can be defined inside other functions.

# Sorting in the C++ STL
# Lambda Functions — Storing [1/2]

We can store a lambda function in an `auto` variable.

```
auto add = [](int a, int b) { return a+b; };
cout << add(2, 3);  // Call like a normal function
```

Semicolon at the end of a variable declaration

To give it a definite type, use `std::function` (`<functional>`), a wrapper for functions and function-like objects.

```
#include <functional>
using std::function;

function<int(int,int)> add =
    [](int a, int b){ return a+b; };
cout << add(2, 3);  // Call like a normal function
```

Return type & passing method

Parameter types & passing methods

# Sorting in the C++ STL
## Lambda Functions — Storing [2/2]

Passing a lambda to a function:

```
template<typename Func>
void foo(Func f)
{
    cout << f(2, 3);
}

auto add = [](int a, int b){ return a+b };
foo(add);
```

We can rewrite the last two lines to avoid using a variable:

```
foo([](int a, int b){ return a+b });
```

# Sorting in the C++ STL
## Lambda Functions — Capture [1/2]

By default, a lambda function is prohibited from accessing most variables other than its own.

Inaccessible variable

```
int k = 3;
auto mult = [](int n){ return k*n; };  // COMPILE ERROR!
```

Give a lambda access to outside variables, *as they are at the point the lambda is defined*, by **capturing** them.

**Capture by copy**: (think "by value") the lambda gets a *copy* of `k`.

```
auto mult_cp  = [k](int n) { return k*n; };
auto mult_ref = [&k](int n) { return k*n; };
```

**Capture by reference**: the lambda's `k` is an *alias*. If the outside `k` changes, then the lambda knows about it. If the outside `k` goes away, then the lambda has a problem.

# Sorting in the C++ STL
## Lambda Functions — Capture [2/2]

Here are some fancier capture lists.

```
[a,b,&c,&d](int n){ … // Capture a, b by copy,
                      //  c, d by reference
```

I mostly use these two.

```
[=](int n){ …         // Capture any needed by copy

[&](int n){ …         // Capture any needed by reference

[=,&c,&d](int n){ …   // Capture c, d by reference,
                      //  any other needed by copy

[&,a,b](int n){ …     // Capture a, b by copy,
                      //  any other needed by reference
```

# Sorting in the C++ STL
## Using the Algorithms — Ordinary Usage

Call algorithm `std::sort` with two random-access iterators:

```
vector<int> vv;
sort(begin(vv), end(vv));  // Ascending order
```

… or two random-access iterators and a comparison. For descending order, use `std::greater` (`<functional>`).

```
sort(begin(vv), end(vv),
     greater<int>());      // Descending order
```

Default constructor call. `std::greater<int>` is a *type*, but we are only allowed to pass an *object*.

`std::stable_sort` is used the same way.

# Sorting in the C++ STL
## Using the Algorithms — Custom Comparison

A custom comparison can be written using a lambda function.

- This should take two parameters of the type of the items being sorted. Pass these by value or reference-to-const, as appropriate.
- It should return `bool`: `true` if the value of the first parameter must come before the value of the second (think `operator<`).

```
vector<pair<int, string>> data;

stable_sort(begin(data), end(data),  // Custom order
    [](const pair<int, string> & a,
       const pair<int, string> & b)
    {
        return a.first < b.first;    // Sort by int part
    }
);
```

Closing parenthesis and semicolon for the `std::stable_sort` call.

A lambda definition is always inside a statement. Such a statement needs to end the way any statement ends.

# Sorting in the C++ STL
## Using the Algorithms — Sorting Linked Lists

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

```
#include <list>
using std::list;
#include <functional>
using std::greater;

list<double> myList;
myList.sort();                          // Ascending order
myList.sort(greater<double>());         // Descending order
myList.sort([](double a, double b)      // Custom order
    { …
```

Sorting a `std::forward_list` works the same way.

# Sorting in the C++ STL
# Using the Algorithms — CODE

TO DO

- Look at some code that uses STL sorting algorithms with custom comparison functions.

*See* `comparison.cpp`.

A number of other STL algorithms take optional comparisons. These are specified in the same way.

- Binary Search (`std::binary_search`, `std::lower_bound`, …)
- Sort testing (`std::is_sorted`, …)
- Stable Merge (`std::merge`, …)
- Maximum/minimum (`std::max`, `std::min`, `std::max_element`, …)
- Etc.