A Long Journey of Changing std::sort Implementation at Scale

DANILA KUTENIN

WHO AM I?

- Senior Software Engineer at Google
- DC efficiency

AGENDA FOR TODAY

- History of sorting
- Why have we decided to change anything?
- Bugs, bugs, bugs
- Implementation
- What can you do?

REMINDERS

- Sorting is the ordering of elements
- std::sort, std::stable_sort, ranges::sort, etc

```
std::sort(begin, end);
std::ranges::sort(cont);
```

REMINDERS

- Sorting is the ordering of elements
- std::sort, std::stable_sort, ranges::sort, etc

```
std::sort(begin, end, comp);
std::ranges::sort(cont, comp);
```

QUICKSORT

QUICKSORT

- Quick sort

QUICKSORT

- Quick sort
- Take any element

QUICKSORT

- Quick sort
- Take any element
- Less to the left

QUICKSORT

- Quick sort
- Take any element
- Less to the left
- More to the right

QUICKSORT

- Quick sort
- Take any element
- Less to the left
- More to the right
- Recurse on both parts

- Quick sort (STL version)

```
1 template <class RandomAccessIterator>
2 void sort(RandomAccessIterator first, RandomAccessIterator l
3     __quick_sort_loop(first, last);
4     __final_insertion_sort(first, last);
5 }
```

- Quick sort (STL version)

```
1 template <class RandomAccessIterator>
2 void sort(RandomAccessIterator first, RandomAccessIterator l
3     __quick_sort_loop(first, last);
4     __final_insertion_sort(first, last);
5 }
```

- Quick sort (STL version)

```
 1 template <class RandomAccessIterator, class T, class Compar
 2 void __quick_sort_loop(RandomAccessIterator first,
 3                         RandomAccessIterator last, Compare comp) {
 4   while (last - first > __stl_threshold) {
 5     RandomAccessIterator cut = __unguarded_partition
 6     (first, last, T(__median(*first,
 7                              *(first + (last - first)/2),
 8                              *(last - 1),
 9                              comp)),
10      comp);
11     if (cut - first >= last - cut) {
12       __quick_sort_loop(cut, last, comp);
13       last = cut;
14     } else {
15       __quick_sort_loop(first, cut, comp);
```

- Quick sort (STL version)

```
template <class RandomAccessIterator, class T, class Compar
void __quick_sort_loop(RandomAccessIterator first,
                       RandomAccessIterator last, Compare comp) {
  while (last - first > __stl_threshold) {
    RandomAccessIterator cut = __unguarded_partition
    (first, last, T(__median(*first,
                             *(first + (last - first)/2),
                             *(last - 1),
                             comp)),
     comp);
    if (cut - first >= last - cut) {
      __quick_sort_loop(cut, last, comp);
      last = cut;
    } else {
      __quick_sort_loop(first, cut, comp);
```

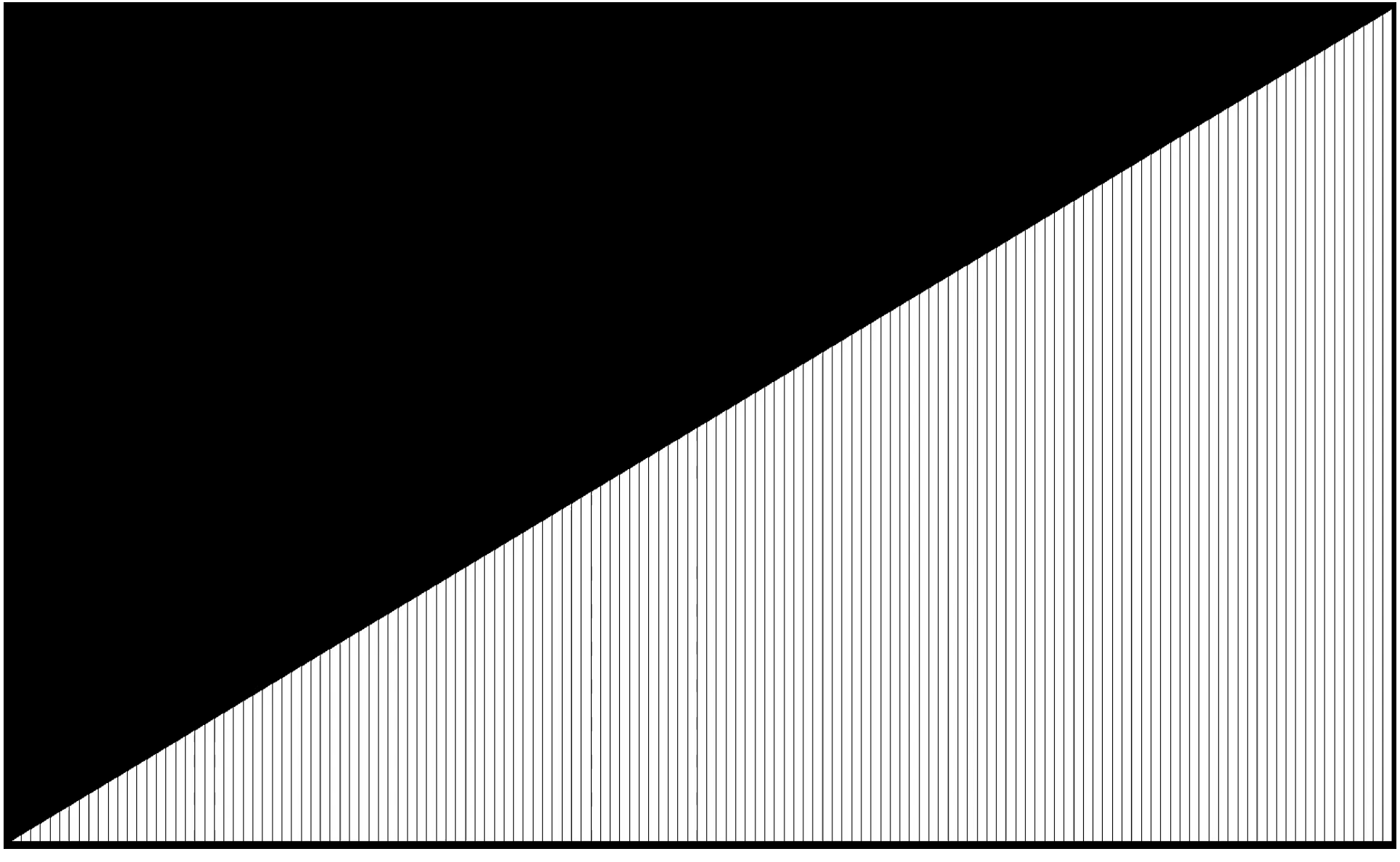
- Quick sort (STL version)

```
template <class RandomAccessIterator, class T, class Compar
void __quick_sort_loop(RandomAccessIterator first,
                       RandomAccessIterator last, Compare comp) {
  while (last - first > __stl_threshold) {
    RandomAccessIterator cut = __unguarded_partition
    (first, last, T(__median(*first,
                             *(first + (last - first)/2),
                             *(last - 1),
                             comp)),
     comp);
    if (cut - first >= last - cut) {
      __quick_sort_loop(cut, last, comp);
      last = cut;
    } else {
      __quick_sort_loop(first, cut, comp);
```

- Quick sort (STL version)

```
1 template <class RandomAccessIterator>
2 void sort(RandomAccessIterator first, RandomAccessIterator l
3     __quick_sort_loop(first, last);
4     __final_insertion_sort(first, last);
5 }
```

- Quick sort (STL version)



STANDARD LIBRARY EVOLVED FROM STL

STD::SORT

std::sort

Defined in header <algorithm>

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );                                   (1)  (until C++20)
template< class RandomIt >
constexpr void sort( RandomIt first, RandomIt last );                              (since C++20)

template< class ExecutionPolicy, class RandomIt >
void sort( ExecutionPolicy&& policy,                                          (2)  (since C++17)
           RandomIt first, RandomIt last );

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );                     (3)  (until C++20)
template< class RandomIt, class Compare >
constexpr void sort( RandomIt first, RandomIt last, Compare comp );                (since C++20)

template< class ExecutionPolicy, class RandomIt, class Compare >
void sort( ExecutionPolicy&& policy,                                          (4)  (since C++17)
           RandomIt first, RandomIt last, Compare comp );
```

Complexity

$O(N \cdot log(N))$ comparisons, where N is `std::distance(first, last)`.

STD::SORT

std::sort

Defined in header <algorithm>

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );                                   (1)  (until C++20)
template< class RandomIt >
constexpr void sort( RandomIt first, RandomIt last );                              (since C++20)

template< class ExecutionPolicy, class RandomIt >
void sort( ExecutionPolicy&& policy,                                          (2)  (since C++17)
           RandomIt first, RandomIt last );

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );                     (3)  (until C++20)
template< class RandomIt, class Compare >
constexpr void sort( RandomIt first, RandomIt last, Compare comp );                (since C++20)

template< class ExecutionPolicy, class RandomIt, class Compare >
void sort( ExecutionPolicy&& policy,                                          (4)  (since C++17)
           RandomIt first, RandomIt last, Compare comp );
```

Proposed resolution:

In 27.8.2.1 [sort], change the complexity to "O(N log N)", and remove footnote 266:

Complexity: ~~Approximately~~ O(*N* log(*N*)) (where *N* == *last* - *first*) comparisons~~ on the average~~.~~266)~~

~~266) If the worst case behavior is important `stable_sort()` (25.3.1.2) or `partial_sort()` (25.3.1.3) should be used.~~

Quicksort's worst case is

Just pick the worst element

GCC libstdc++

```
 1 template<typename _RandomAccessIterator, typename _Compare>
 2   inline void
 3   __sort(_RandomAccessIterator __first, _RandomAccessIterat
 4   {
 5   if (__first != __last)
 6   {
 7     std::__introsort_loop(__first, __last,
 8         std::__lg(__last - __first) * 2,
 9         __comp);
10     std::__final_insertion_sort(__first, __last, __comp);
11   }
12 }
```

GCC libstdc++

```
template<typename _RandomAccessIterator, typename _Compare>
  inline void
  __sort(_RandomAccessIterator __first, _RandomAccessIterat
  {
  if (__first != __last)
  {
    std::__introsort_loop(__first, __last,
        std::__lg(__last - __first) * 2,
        __comp);
    std::__final_insertion_sort(__first, __last, __comp);
  }
}
```

GCC

```
template<typename _RandomAccessIterator, typename _Compare>
  inline void
  __sort(_RandomAccessIterator __first, _RandomAccessIterat
  {
  if (__first != __last)
  {
    std::__introsort_loop(__first, __last,
        std::__lg(__last - __first) * 2,
        __comp);
    std::__final_insertion_sort(__first, __last, __comp);
  }
}
```

Original

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator l
    __quick_sort_loop(first, last);
    __final_insertion_sort(first, last);
}
```

GCC

```
template<typename _RandomAccessIterator, typename _Compare>
  inline void
  __sort(_RandomAccessIterator __first, _RandomAccessIterat
  {
  if (__first != __last)
  {
    std::__introsort_loop(__first, __last,
        std::__lg(__last - __first) * 2,
        __comp);
    std::__final_insertion_sort(__first, __last, __comp);
  }
}
```

Original

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator l
    __quick_sort_loop(first, last);
    __final_insertion_sort(first, last);
}
```

INTROSORT

INTROSORT

- Almost like a quick sort

INTROSORT

- Almost like a quick sort
- If we suspect a lot of recursion calls, use heap sort

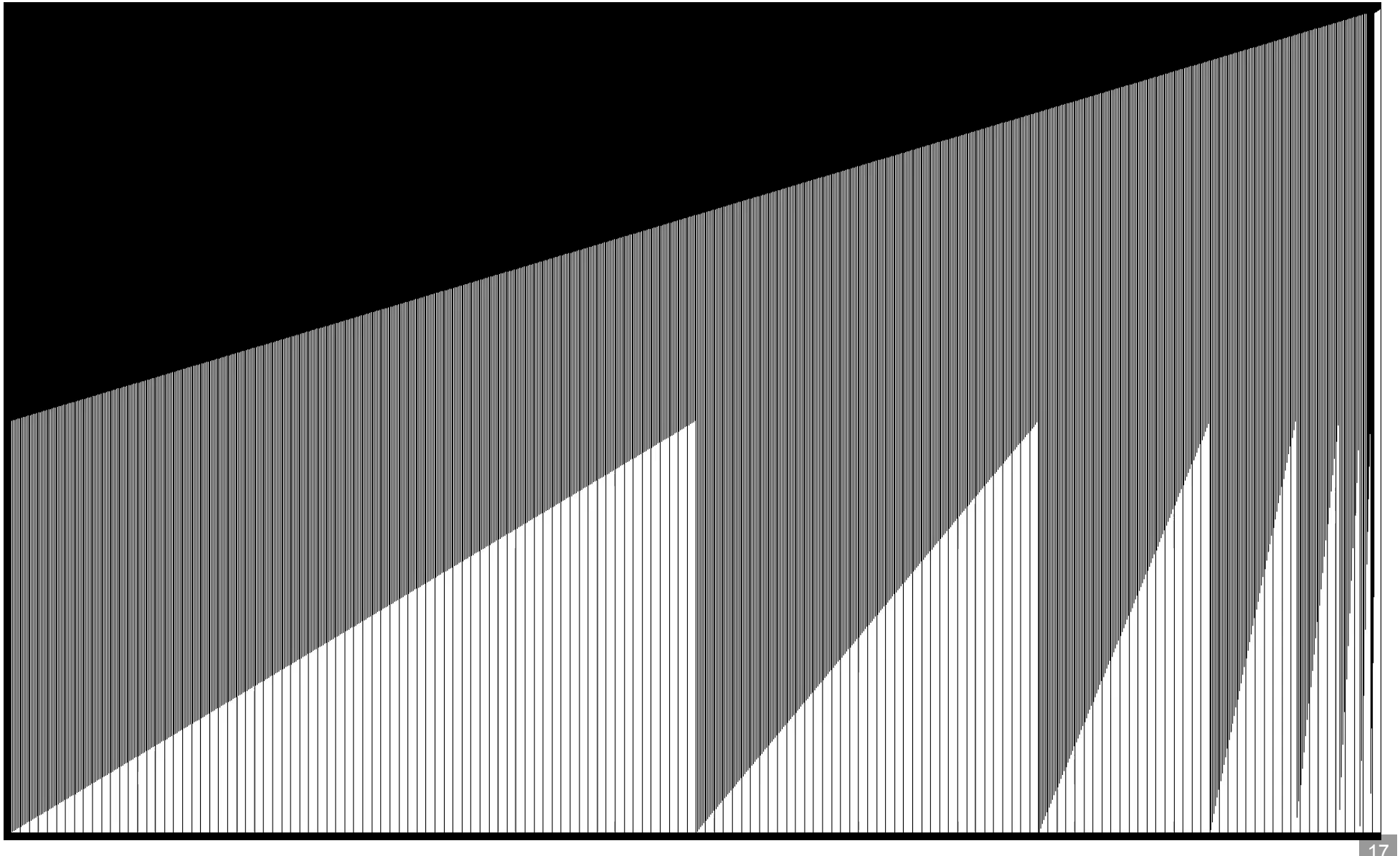
INTROSORT

- Almost like a quick sort
- If we suspect a lot of recursion calls, use heap sort
- GCC libstdc++ uses $2\log_2 n$ depth limit

INTROSORT

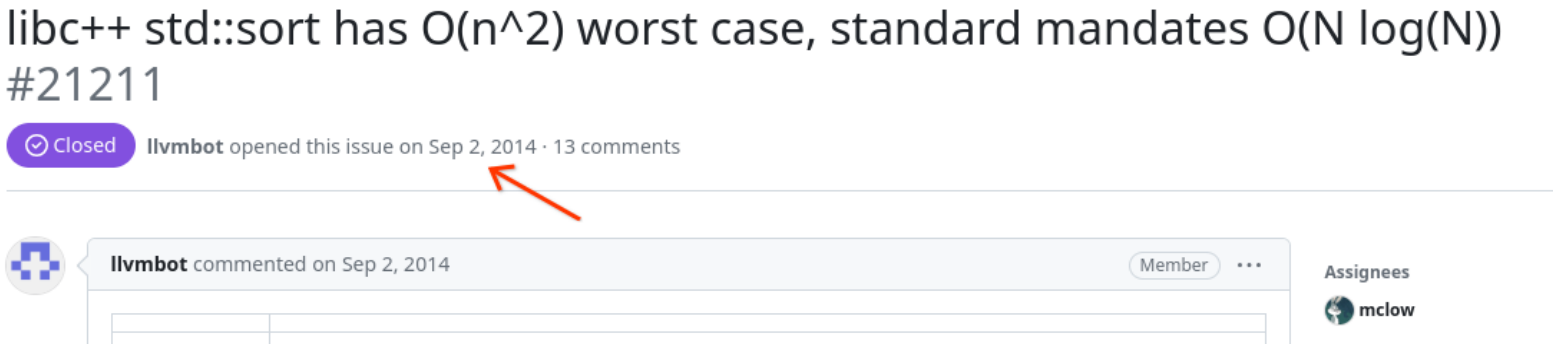
- Almost like a quick sort
- If we suspect a lot of recursion calls, use heap sort
- GCC libstdc++ uses $2\log_2 n$ depth limit
- Microsoft standard library does the same

INTROSORT (WORST CASE)



LLVM libc++ did not do anything at all for a long time

LLVM libc++ did not do anything at all for a long time



LLVM libc++ did not do anything at all for a long time

7 years

Only 0.01% of all calls got into the heap sort fallback in production

LLVM (HISTORY)

What is a good sort?

What is a good sort?

- $O(n \log n)$ comparisons

What is a good sort?

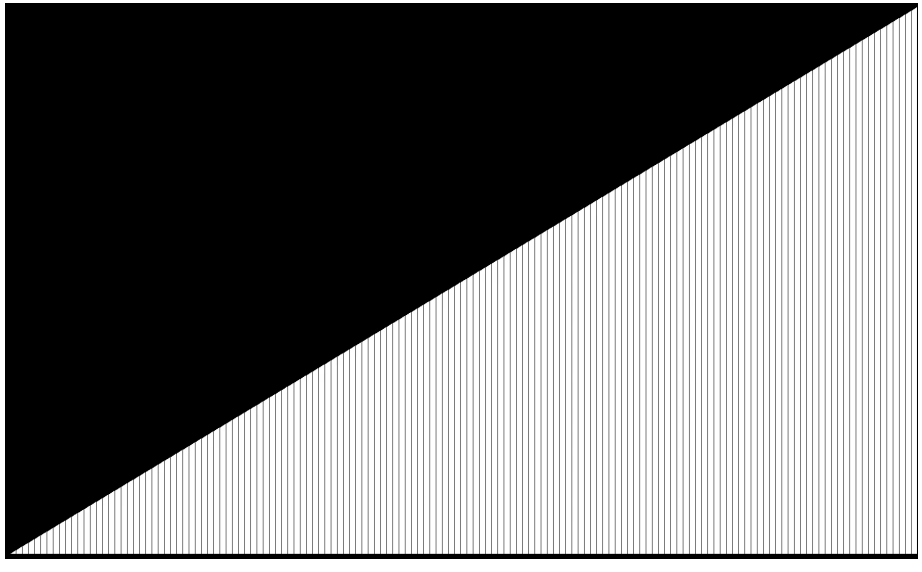
- $O(n \log n)$ comparisons
- Recognizes almost sorted patterns

What is a good sort?

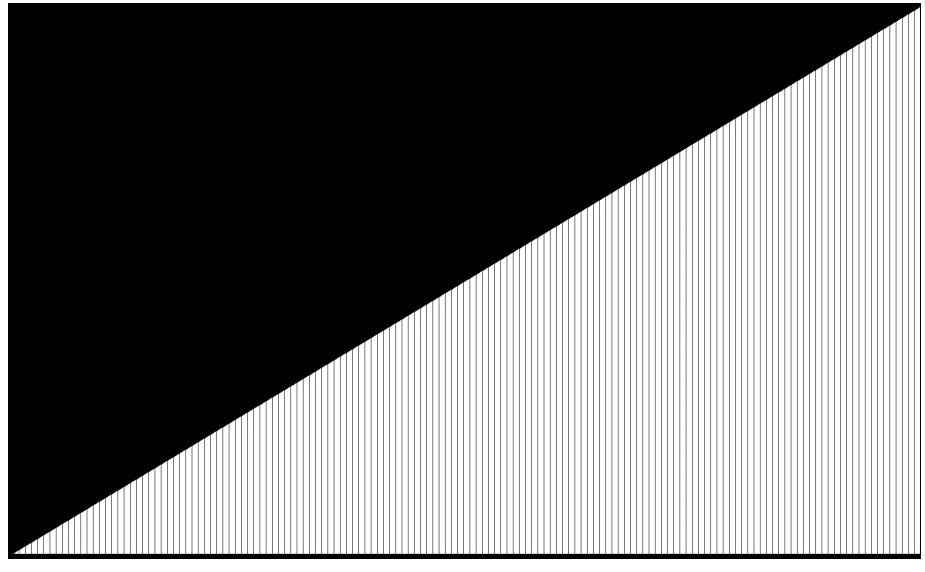
- $O(n \log n)$ comparisons
- Recognizes almost sorted patterns
- Fast for modern hardware

What is a good sort?

- $O(n \log n)$ comparisons
- Recognizes almost sorted patterns
- Fast for modern hardware
- Fewer comparisons for heavy comparison sorting



libcxx on ascending



libstdcxx on ascending

How libcxx has achieved that? Insertion sort on every step up to 8 insertions.

```
 1 const unsigned __limit = 8;
 2 unsigned __count = 0;
 3 for (_RandomAccessIterator __i = __j+1; __i != __last; ++__
 4     if (__comp(*__i, *__j)) {
 5         value_type __t(_VSTD::move(*__i));
 6         _RandomAccessIterator __k = __j;
 7         __j = __i;
 8         // Insert and copy
 9         do {
10             *__j = _VSTD::move(*__k);
11             __j = __k;
12         } while (__j != __first && __comp(__t, *--__k));
13         *__j = _VSTD::move(__t);
14         // Exit after 8 trials
15         if (++__count == __limit)
```

How libcxx has achieved that? Insertion sort on every step up to 8 insertions.

```
 3 for (_RandomAccessIterator __i = __j+1; __i != __last; ++__
 4     if (__comp(*__i, *__j)) {
 5         value_type __t(_VSTD::move(*__i));
 6         _RandomAccessIterator __k = __j;
 7         __j = __i;
 8         // Insert and copy
 9         do {
10             *__j = _VSTD::move(*__k);
11             __j = __k;
12         } while (__j != __first && __comp(__t, *--__k));
13         *__j = _VSTD::move(__t);
14         // Exit after 8 trials
15         if (++__count == __limit)
16             return ++__i == __last;
17     }
```

How libcxx has achieved that? Insertion sort on every step up to 8 insertions.

```
 5          value_type __t(_VSTD::move(*__i));
 6          _RandomAccessIterator __k = __j;
 7          __j = __i;
 8          // Insert and copy
 9          do {
10              *__j = _VSTD::move(*__k);
11              __j = __k;
12          } while (__j != __first && __comp(__t, *--__k));
13          *__j = _VSTD::move(__t);
14          // Exit after 8 trials
15          if (++__count == __limit)
16              return ++__i == __last;
17      }
18      __j = __i;
19  }
```

Overall every now and then we have a better sorting

Let's change and save some cycles

Let's change and save some cycles

We tried

id_1	id_2
0	3
0	**3**
0	**2**
0	2
0	3
1	1
1	3

PostgreSQL output

```
1  create table example (id_1 integer, id_2 integer);
2
3  -- Insert lots of equal id_1
4  insert into example (id_1, id_2) values (1, 1);
5  insert into example (id_1, id_2) values (0, 3);
6  insert into example (id_1, id_2) values (0, 2);
7  insert into example (id_1, id_2) values (0, 3);
8  insert into example (id_1, id_2) values (0, 2);
9  insert into example (id_1, id_2) values (0, 3);
10 insert into example (id_1, id_2) values (1, 3);
11
12 -- Order only by the first element, second
13 -- is undefined for equal first elements.
14 select * from example order by id_1;
```

diff.sql hosted with ❤ by **GitHub** **view raw**

id_1	id_2
0	3
0	**2**
0	**3**
0	2
0	3
1	1
1	3

MySQL output

PROBLEM: TIES

id_1	id_2
0	3
0	**3**
0	**2**
0	2
0	3
1	1
1	3

PostgreSQL output

```sql
create table example (id_1 integer, id_2 integer);

-- Insert lots of equal id_1
insert into example (id_1, id_2) values (1, 1);
insert into example (id_1, id_2) values (0, 3);
insert into example (id_1, id_2) values (0, 2);
insert into example (id_1, id_2) values (0, 3);
insert into example (id_1, id_2) values (0, 2);
insert into example (id_1, id_2) values (0, 3);
insert into example (id_1, id_2) values (1, 3);

-- Order only by the first element, second
-- is undefined for equal first elements.
select * from example order by id_1;
```

diff.sql hosted with ❤ by **GitHub** **view raw**

id_1	id_2
0	3
0	**2**
0	**3**
0	2
0	3
1	1
1	3

MySQL output

PROBLEM: TIES

```
std::vector<std::pair<int, int>> first_elements_equal{
  {1, 1}, {1, 2}
};
std::sort(first_elements_equal.begin(),
          first_elements_equal.end(),
          [](const auto& lhs, const auto& rhs) {
            // Compare only by a part of sorted data.
            return lhs.first < rhs.first;
          });
// Serialize or make assumptions about all data.
// Wrong, might be either 1 or 2.
assert(first_elements_equal[0].second == 1);
```

PROBLEM: TIES

It took a year to fix all golden tests

PROBLEM: TIES. HOW

```
1 template <class _AlgPolicy, class _RandomAccessIterator, cla
2 inline _LIBCPP_HIDE_FROM_ABI _LIBCPP_CONSTEXPR_SINCE_CXX20
3 void __sort_impl(_RandomAccessIterator __first, _RandomAcces
4   std::__debug_randomize_range<_AlgPolicy>(__first, __last);
5   // ...
```

PROBLEM: TIES. HOW

```
1 template <class _AlgPolicy, class _RandomAccessIterator, cla
2 inline _LIBCPP_HIDE_FROM_ABI _LIBCPP_CONSTEXPR_SINCE_CXX20
3 void __sort_impl(_RandomAccessIterator __first, _RandomAcces
4   std::__debug_randomize_range<_AlgPolicy>(__first, __last);
5   // ...
```

Shuffle the range before

PROBLEM: TIES. HOW

```
_LIBCPP_HIDE_FROM_ABI static uint_fast64_t __seed() {
  static char __x;
  return reinterpret_cast<uintptr_t>(&__x);
}
```

PROBLEM: TIES. HOW

```
_LIBCPP_HIDE_FROM_ABI static uint_fast64_t __seed() {
  static char __x;
  return reinterpret_cast<uintptr_t>(&__x);
}
```

Address space layout randomization

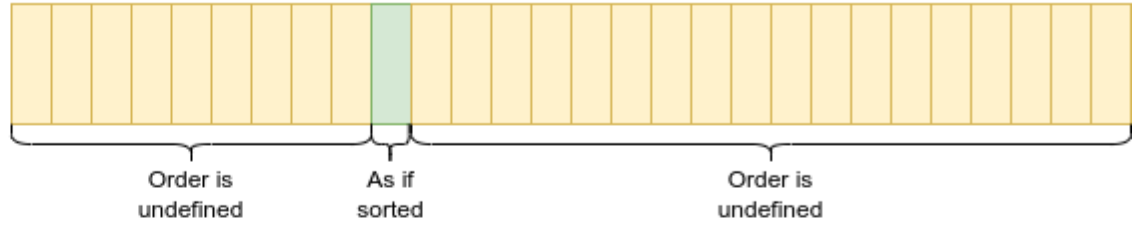
Randomization went beyond std::sort

Randomization went beyond std::sort

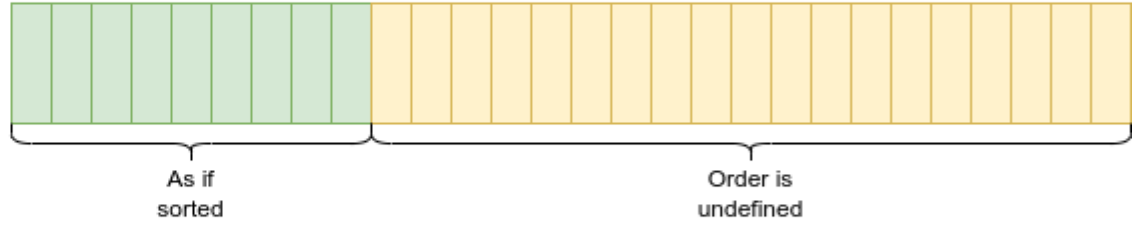
Also to std::nth_element, std::partial_sort

std::nth_element, std::partial_sort

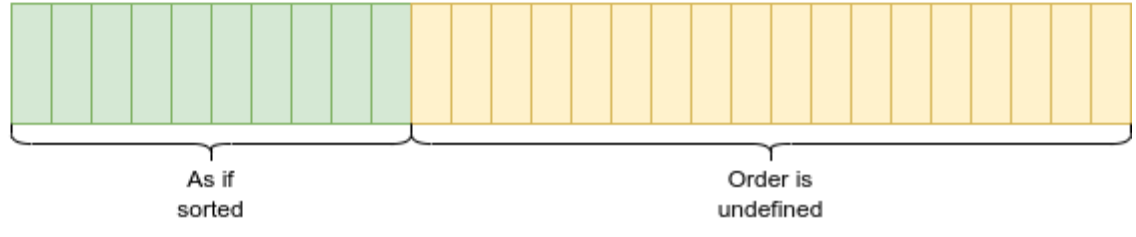
std::nth_element(begin, begin + **9**, end);



std::partial_sort(begin, begin + **9**, end);



std::partial_sort(begin, begin + **10**, end);



Randomize before and randomize after the undefined ranges

std::nth_element

```
 1 template <class _AlgPolicy, class _RandomAccessIterator, cl
 2 inline _LIBCPP_HIDE_FROM_ABI _LIBCPP_CONSTEXPR_SINCE_CXX20
 3 void __nth_element_impl(_RandomAccessIterator __first, _Ran
 4                          _Compare& __comp) {
 5   if (__nth == __last)
 6     return;
 7   std::__debug_randomize_range<_AlgPolicy>(__first,
 8                                            __last);
 9   std::__nth_element<_AlgPolicy, __comp_ref_type<_Compare>
10   std::__debug_randomize_range<_AlgPolicy>(__first,
11                                            __nth);
12   if (__nth != __last) {
13     std::__debug_randomize_range<_AlgPolicy>(++__nth,
14                                              __last);
15   }
```

std::partial_sort

```
 1 template <class _AlgPolicy, class _Compare, class _RandomAc
 2 _LIBCPP_HIDE_FROM_ABI _LIBCPP_CONSTEXPR_SINCE_CXX20
 3 _RandomAccessIterator __partial_sort(_RandomAccessIterator
 4                                       _Compare& __comp) {
 5   if (__first == __middle)
 6       return _IterOps<_AlgPolicy>::next(__middle, __last);
 7   std::__debug_randomize_range<_AlgPolicy>(__first, __last)
 8   auto __last_iter =
 9       std::__partial_sort_impl<_AlgPolicy>(__first, __middl
10   std::__debug_randomize_range<_AlgPolicy>(__middle, __last
11   return __last_iter;
12 }
```

BUGS

DETERMINISM

DETERMINISM

DETERMINISM

- Assume you sort some data

DETERMINISM

- Assume you sort some data
- Cache it

DETERMINISM

- Assume you sort some data
- Cache it
- Now ties are handled randomly

DETERMINISM

- Assume you sort some data
- Cache it
- Now ties are handled randomly
- Lower hit rate

DETERMINISM

Use std::stable_sort

```
@@ -113,7 +112,8 @@ class ArrayAUCImpl
113                 sorted_labels[i].label = label;
114             }
115
116 -           ::sort(sorted_labels.begin(), sorted_labels.end(), [](const auto & lhs, const auto & rhs) { return lhs.score > rhs.score; });
117
```

```
112                 sorted_labels[i].label = label;
113             }
114
115 +           /// Stable sort is required for for labels to apply in same order if score is equal
116 +           std::stable_sort(sorted_labels.begin(), sorted_labels.end(), [](const auto & lhs, const auto & rhs) { return lhs.score > rhs.score; });
117
```

Let's write a median function

```
 1 int64_t median(const std::vector<int64_t>& v) {
 2   int64_t med = v.size() / 2;
 3   std::nth_element(v.begin(), v.begin() + med, v.end());
 4   int64_t result = v[med];
 5   if (v.size() % 2 == 0) {
 6     std::nth_element(v.begin(), v.begin() + med - 1, v.end(
 7     result = (v[med] + v[med - 1]) / 2;
 8   }
 9   return result;
10 }
```

Let's write a median function

```cpp
int64_t median(const std::vector<int64_t>& v) {
  int64_t med = v.size() / 2;
  std::nth_element(v.begin(), v.begin() + med, v.end());
  int64_t result = v[med];
  if (v.size() % 2 == 0) {
    std::nth_element(v.begin(), v.begin() + med - 1, v.end(
    result = (v[med] + v[med - 1]) / 2;
  }
  return result;
}
```

Let's write a median function

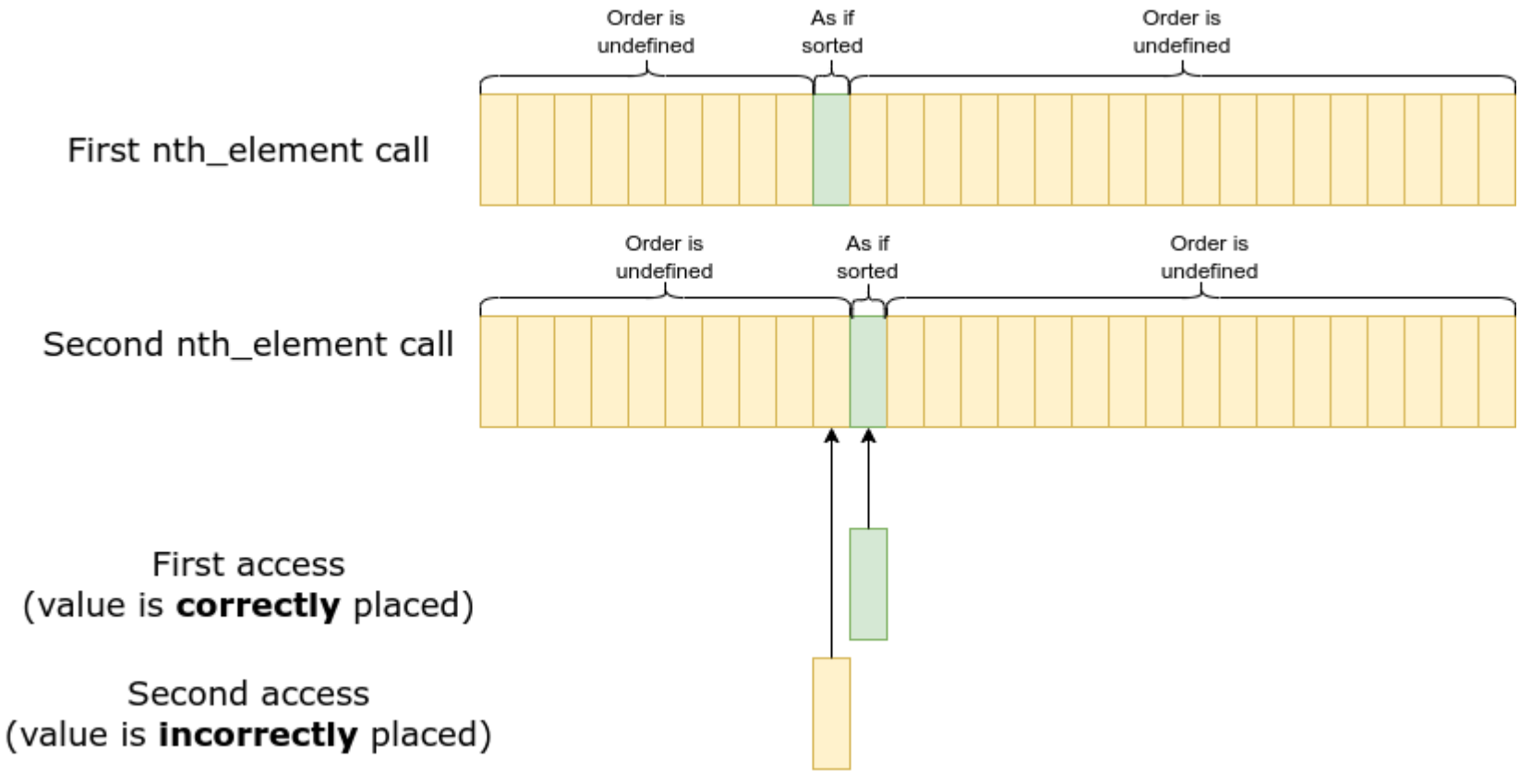
```cpp
int64_t median(const std::vector<int64_t>& v) {
  int64_t med = v.size() / 2;
  std::nth_element(v.begin(), v.begin() + med, v.end());
  int64_t result = v[med];
  if (v.size() % 2 == 0) {
    std::nth_element(v.begin(), v.begin() + med - 1, v.end(
    result = (v[med] + v[med - 1]) / 2;
  }
  return result;
}
```

Let's write a median function

```cpp
int64_t median(const std::vector<int64_t>& v) {
  int64_t med = v.size() / 2;
  std::nth_element(v.begin(), v.begin() + med, v.end());
  int64_t result = v[med];
  if (v.size() % 2 == 0) {
    std::nth_element(v.begin(), v.begin() + med - 1, v.end(
    result = (v[med] + v[med - 1]) / 2;
  }
  return result;
}
```

Let's write a median function

```cpp
int64_t median(const std::vector<int64_t>& v) {
  int64_t med = v.size() / 2;
  std::nth_element(v.begin(), v.begin() + med, v.end());
  int64_t result = v[med];
  if (v.size() % 2 == 0) {
    std::nth_element(v.begin(), v.begin() + med - 1, v.end(
    result = (v[med] + v[med - 1]) / 2;
  }
  return result;
}
```



Before that you might get lucky because nth_element puts elements around nth close to total order

We found so many of them!

We found so many of them!

Repo: shogun-toolbox/shogun
Stars: 3k
Commit: 9b8d85
File: src/shogun/mathematics/linalg/LinalgNamespace.h
Snippet:

```
1819                    int64_t n = a.size() / 2;
1820                    std::nth_element(a_copy.begin(), a_copy.begin() + n, a_copy.end());
1821                    if (a_copy.size() % 2 == 0)
1822                    {
1823                            std::nth_element(
1824                                a_copy.begin(), a_copy.begin() + n - 1, a_copy.end());
1825                            result = (a_copy[n] + a_copy[n - 1]) / 2;
1826                    }
```

We found so many of them!

Repo: alandefreitas/matplotplusplus
Stars: 3.5k
Commit: a40344
File: source/matplot/axes_objects/histogram.cpp
Snippet:

```
384         if (iqr_not_too_small) {
385             size_t q1_index = static_cast<size_t>(n * 0.25);
386             size_t q3_index = n - static_cast<size_t>(n * 0.25);
387             auto x_copy = x;
388             std::nth_element(x_copy.begin(), x_copy.begin() + q1_index,
389                              x_copy.end());
390             std::nth_element(x_copy.begin(), x_copy.begin() + q3_index,
391                              x_copy.end());
392             double interquartile_range = x_copy[q3_index] - x_copy[q1_index];
393             double iq = std::max(interquartile_range, xrange / 10.);
394             bin_width = 2 * iq * pow(n, -1. / 3.);
395         }
```

We found so many of them!

Repo: azerothcore/azerothcore-wotlk
Stars: 4.5k
Commit: 1004da
File: src/common/Utilities/MathUtil.h
Snippet:

```
66          // Applying nth_element
67          // on n/2th index
68          std::nth_element(a.begin(),
69              a.begin() + n / 2,
70              a.end());
71
72          // Applying nth_element
73          // on (n-1)/2 th index
74          std::nth_element(a.begin(),
75              a.begin() + (n - 1) / 2,
76              a.end());
77
78          // Find the average of value at
79          // index N/2 and (N-1)/2
80          return (T)(a[(n - 1) / 2]
81              + a[n / 2])
82              / 2.0;
```

We found so many of them!

Repo: jmzkChain/jmzk
Stars: 1.2k
Commit: 3fbc2d
File: libraries/chain/include/evt/chain/contracts/evt_contract_utils.hpp
Snippet:

```
404             // find median
405             if(values.size() % 2 == 0) {
406                 auto it1 = values.begin() + values.size() / 2 - 1;
407                 auto it2 = values.begin() + values.size() / 2;
408
409                 std::nth_element(values.begin(), it1 , values.end());
410                 std::nth_element(values.begin(), it2 , values.end());
411
412                 nv = ::floor((*it1 + *it2) / 2);
413             }
```

We found so many of them!

Repo: Ableton/link
Stars: 1k
Commit: adbbb5
File: include/ableton/link/Median.hpp
Snippet:

```
36    if (n % 2 == 0)
37    {
38      std::nth_element(begin, begin + n / 2, end);
39      std::nth_element(begin, begin + (n - 1) / 2, end);
40      return (*(begin + (n / 2)) + *(begin + (n - 1) / 2)) / 2.0;
41    }
```

We found so many of them!

Repo: http://mia.sourceforge.net/
Stars: N/A. medical image analysis
Commit: a081c3
File: mia/core/fullstats.cc
Snippet:

```
45          } else {
46                  Vector::iterator i1 = tmp.begin() + n / 2 - 1;
47                  Vector::iterator i2 = tmp.begin() + n / 2;
48                  std::nth_element(tmp.begin(), i1, tmp.end());
49                  std::nth_element(tmp.begin(), i2, tmp.end());
50                  m_median = (*i1 + *i2) / 2.0;
51          }
```

We found so many of them!

Repo: zenustech/zeno
Stars: 471
Commit: 4fd158
File: projects/FastFLIP/FLIP_vdb.cpp
Snippet:

```
3644    int nleaf = max_v_per_leaf.size();
3645    int top90 = nleaf * 99 / 100;
3646    std::nth_element(max_v_per_leaf.begin(), max_v_per_leaf.begin() + nleaf - 1,
3647                     max_v_per_leaf.end());
3648    std::nth_element(max_v_per_leaf.begin(), max_v_per_leaf.begin() + top90,
3649                     max_v_per_leaf.end());
3650
3651    /*for (const auto& v : max_v_per_leaf) {
3652            if (v > max_v) {
3653                    max_v = v;
3654            }
3655    }*/
3656    max_v = max_v_per_leaf[nleaf - 1];
```

We found so many of them!

Repo: orfeotoolbox/OTB
Stars: 287
Commit: d422bb
File: Modules/Registration/DisparityMap/include/otbDisparityMapMedianFilter.hxx
Snippet:

```
380           if ((p & 0x1) == 0)
381           {
382             const unsigned int                                medianPosition_low  = p / 2 - 1;
383             const unsigned int                                medianPosition_high = p / 2;
384             const typename std::vector<InputPixelType>::iterator medianIterator_low  = pixels.begin() + medianPosition_low;
385             const typename std::vector<InputPixelType>::iterator medianIterator_high = pixels.begin() + medianPosition_high;
386             std::nth_element(pixels.begin(), medianIterator_low, pixels.end());
387             std::nth_element(pixels.begin(), medianIterator_high, pixels.end());
388             outputIt.Set(static_cast<typename OutputImageType::PixelType>((*medianIterator_low + *medianIterator_high) / 2));
389           }
```

Use

_LIBCPP_DEBUG_RANDOMIZE_UNSPECIFIED_STABILIT

with libcxx

Strict weak ordering

- Irreflexivity: $x < x$ is false
- Asymmetry: $x < y$ and $y < x$ cannot be both true
- Transitivity: $x < y$ and $y < z$ imply $x < z$
- Transitivity of incomparability: $x = y$ and $y = z$ imply $x = z$, where $x = y$ means $x < y$ and $y < x$ are both false.

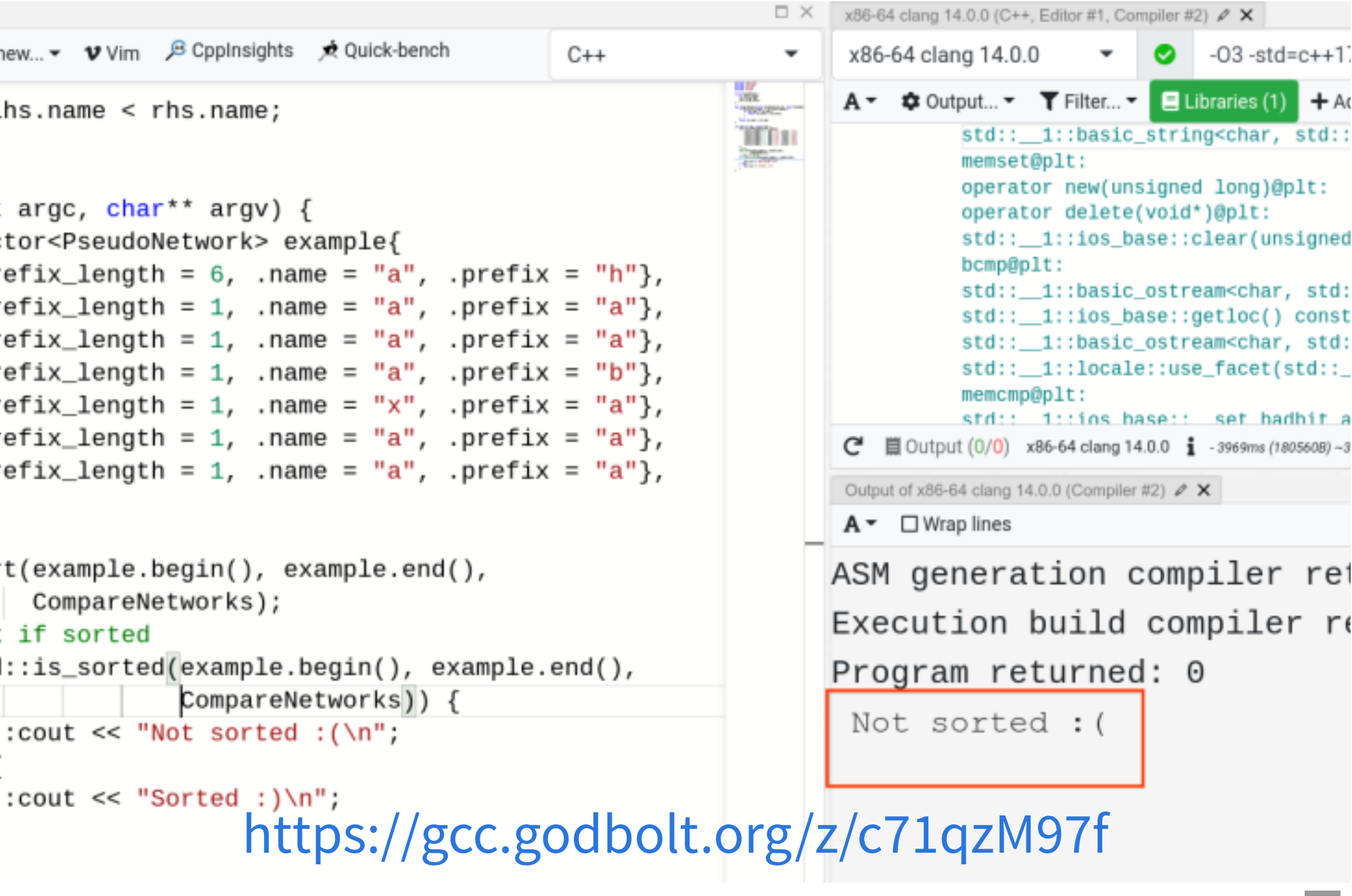
To check this we need to check all triples of elements, it's

$$O(n^3)$$

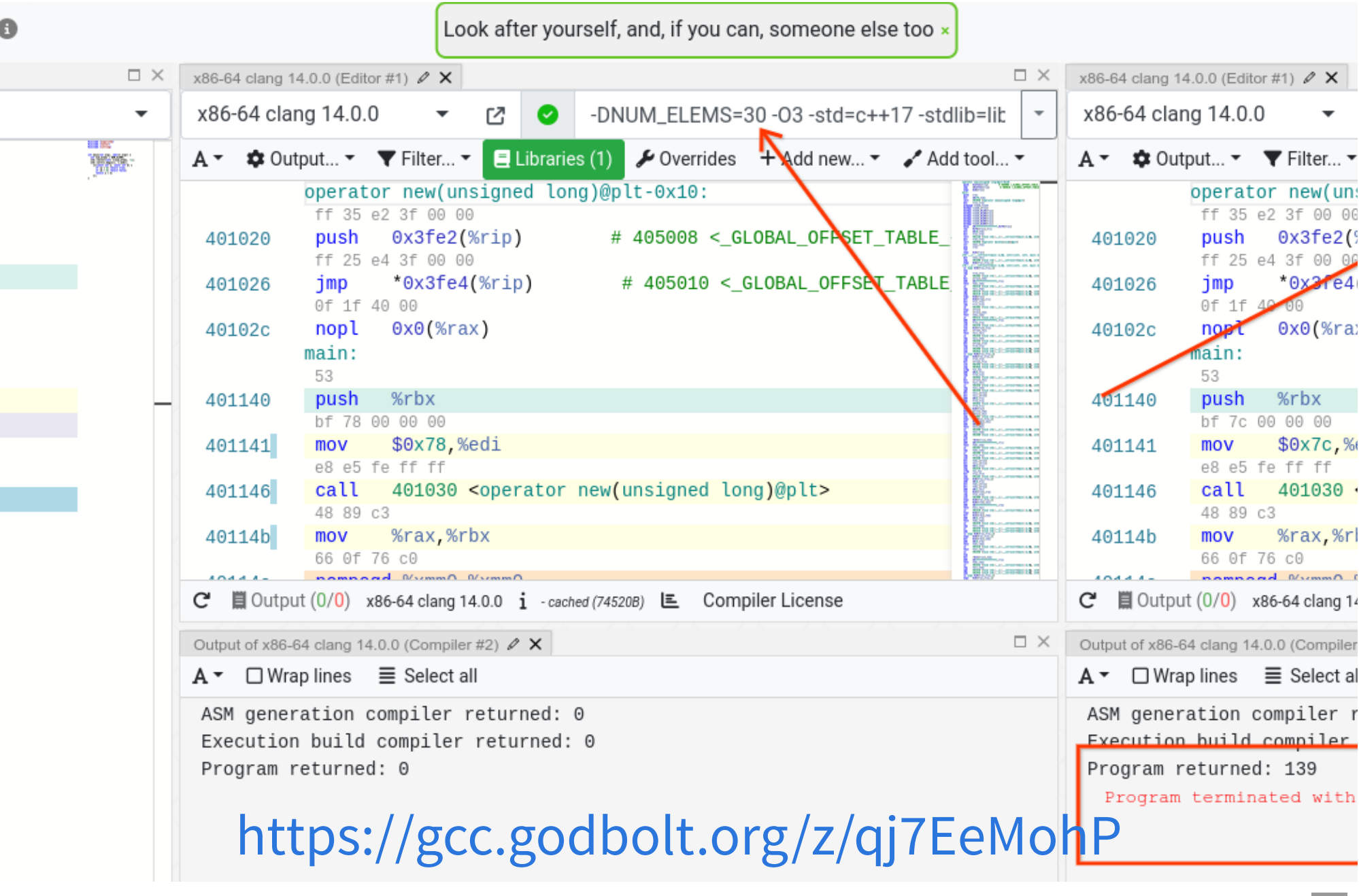
To check this we need to check all triples of elements, it's $O(n^3)$

But what are the risks?

On paper, UB. But in practice?



https://gcc.godbolt.org/z/c71qzM97f


Look after yourself, and, if you can, someone else too ×
x86-64 clang 14.0.0 (Editor #1)
x86-64 clang 14.0.0
-DNUM_ELEMS=30 -O3 -std=c++17 -stdlib=lib
Output... Filter... Libraries (1) Overrides Add new... Add tool...
operator new(unsigned long)@plt-0x10:
ff 35 e2 3f 00 00
401020 push 0x3fe2(%rip) # 405008 <_GLOBAL_OFFSET_TABLE_
ff 25 e4 3f 00 00
401026 jmp *0x3fe4(%rip) # 405010 <_GLOBAL_OFFSET_TABLE
0f 1f 40 00
40102c nopl 0x0(%rax)
main:
53
401140 push %rbx
bf 78 00 00 00
401141 mov $0x78,%edi
e8 e5 fe ff ff
401146 call 401030 <operator new(unsigned long)@plt>
48 89 c3
40114b mov %rax,%rbx
66 0f 76 c0
Output (0/0) x86-64 clang 14.0.0 - cached (74520B) Compiler License
Output of x86-64 clang 14.0.0 (Compiler #2)
Wrap lines Select all
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
Program returned: 139
Program terminated with
https://gcc.godbolt.org/z/qj7EeMohP

RISKS

- Incorrect comparator for ≤ 30 elements, all good
- Incorrect comparator for > 30 elements, can fail with OOB

RISKS

- Incorrect comparator for ≤ 30 elements, all good
- Incorrect comparator for > 30 elements, can fail with OOB
- Have you written a well thought test for 30+ elements?

At Google we consistently (once in months) have outage somewhere because of that

OOB read

```
// The search going up is known to be guarded but the search c
// Prime the downward search with a guard.
// __m still guards upward moving __i
while (__comp(*__i, *__m))
    ++__i;
```

To check this we need to check all triples of elements, it's $O(n^3)$, sorting is $O(n \log n)$.

There is a $O(n^2)$ algorithm to check for strict weak ordering

There is a $O(n^2)$ algorithm to check for strict weak ordering

- Sort the range, if not std::is_sorted, return false

There is a $O(n^2)$ algorithm to check for strict weak ordering

- Sort the range, if not std::is_sorted, return false
- Find prefix of all equivalent elements

There is a $O(n^2)$ algorithm to check for strict weak ordering

- Sort the range, if not std::is_sorted, return false
- Find prefix of all equivalent elements
- Check they are equal

There is a $O(n^2)$ algorithm to check for strict weak ordering

- Sort the range, if not std::is_sorted, return false
- Find prefix of all equivalent elements
- Check they are equal
- Check they are less than all next

There is a $O(n^2)$ algorithm to check for strict weak ordering

- Sort the range, if not std::is_sorted, return false
- Find prefix of all equivalent elements
- Check they are equal
- Check they are less than all next
- Remove them

There is a $O(n^2)$ algorithm to check for strict weak ordering

- Sort the range, if not std::is_sorted, return false
- Find prefix of all equivalent elements
- Check they are equal
- Check they are less than all next
- Remove them
- Proof can be read at https://github.com/danlark1/quadratic_strict_weak_o

is still a lot but we can use it for, say, 100 first elements

$O(n^2)$

Most sorts are small!

CAVEATS

CAVEATS

- Sort should not fail

CAVEATS

- Sort should not fail
- It only tells if strict weak ordering is met

CAVEATS

- Sort should not fail
- It only tells if strict weak ordering is met
- Arbitrary comparator can do anything

CAVEATS

- Sort should not fail
- It only tells if strict weak ordering is met
- Arbitrary comparator can do anything
- We can check we don't read OOB in std::sort itself

[RFC] Strict weak ordering checks in the debug libc++

Runtimes C++

danlark1 1 Apr 26

Summary

The proposal aims to fix strict weak ordering problems in comparison-based algorithms to prevent security issues and uncover bugs in the user code. The proposal includes adding debug checks for strict weak ordering within `_LIBCPP_ENABLE_DEBUG_MODE`.

Background

Not so long ago we proposed and changed std::sort algorithm 25. However, it was rolled back in 16.0.1 because of failures within the broken comparators. That was even true for the previous implementation 4, however, the new one exposed the problems more often.

https://discourse.llvm.org/t/rfc-strict-weak-ordering-checks-in-the-debug-libc/70217

OOB check in 36d8b4

```
uarded since we know the last element is greater than the pivot.
__comp(__pivot, *++__first)) {


rst;
P_ASSERT(__first != __end, "Would read out of bounds, is your comparator a va
(!__comp(__pivot, *__first));

+  first <   last && !  comp(  pivot, *  first)) {
```

Submitted SWO checks for std::{stable_,partial_}sort and rolled out in LLVM 17 in 7e1ee1

```
25
26    template <class _RandomAccessIterator, class _Comp>
27    _LIBCPP_HIDE_FROM_ABI _LIBCPP_CONSTEXPR_SINCE_CXX14 void
28    __check_strict_weak_ordering_sorted(_RandomAccessIterator __
29    #ifdef _LIBCPP_DEBUG_STRICT_WEAK_ORDERING_CHECK
30      using __diff_t  = __iter_diff_t<_RandomAccessIterator>;
31      using _Comp_ref = __comp_ref_type<_Comp>;
32      if (!__libcpp_is_constant_evaluated()) {
33        // Check if the range is actually sorted.
```

-D_LIBCPP_DEBUG_STRICT_WEAK_ORDERING_CHECK
-D_LIBCPP_ENABLE_DEBUG_MODE=1

It took us around 6-7 months to fix everything

BUGS

We found fewer of them but more dangerous.

```
absl::c_sort(vector, [](const auto& lhs, const auto& rhs) {
  return lhs >= rhs; // BUG, comp(a, a) is true
});
```

We found fewer of them but more dangerous.

```
const bool descending = absl::GetFlag(FLAGS_sort_descending);
absl::c_sort(vector, [descending](const auto& lhs,
                  const auto& rhs) {
  if (lhs < rhs) {
    return descending;
  }
  // BUG, if descending is false, comp(a, a) is true.
  return !descending;
});
```

We found fewer of them but more dangerous.

```
absl::c_sort(vector, [](const auto& lhs, const auto& rhs) {
  return lhs->value() < rhs->value();
});

// In test

TEST(TestSort, NoValue) {
  std::vector<T> vector = {NoValueElement()};
  // Technically passes but incorrect.
  Sort(vector);
}
```

We found fewer of them but more dangerous.

```
absl::c_sort(vector, [](const auto& lhs, const auto& rhs) {
  if (!lhs.has_value()) {
    return true;   // BUG, comp(a, a) can be true.
  }
  if (!rhs.has_value()) {
    return false;
  }
  return lhs.value() < rhs.value();
});
```

We found fewer of them but more dangerous.

```
absl::c_sort(*response->mutable_elems(),
    [&](const Element& a,
        const Element& b) {
    int day_diff_a = ...;
    int day_diff_b = ...;
    if (day_diff_a != day_diff_b) {
      // BUG, a = diff, b = -diff, c = diff can be
      // comp(a, b) is true because of CompareClosely
      // comp(b, c) is true because of CompareClosely
      // but comp(a, c) is false because diff is same.
      return std::abs(day_diff_a) < std::abs(day_diff_b);
    }
    return CompareClosely(a, b)
});
```

We found fewer of them but more dangerous.

```
absl::c_sort(vector, [](const auto& lhs, const auto& rhs) {
  // Lots of logic.
  // ...
  // ...
  return true; // BUG, comp(a, a) is true.
});
```

We found fewer of them but more dangerous.

```
absl::c_sort(vector, [](const auto& lhs, const auto& rhs) {
  absl::StatusOr<bool> cmp = Compare(lhs, rhs);
  if (!cmp.ok()) {
    // BUG, a, b, c can be like comp(a, b) is true,
    // comp(b, c) is true, comp(a, c) has !cmp.ok().
    return false;
  }
  // ...
});
```

We found fewer of them but more dangerous.

```
std::sort(points.begin(), points.end(),
              [](const DataPoint& point_1,
                 const DataPoint& point_2) {
  if (MathUtil::AlmostEquals(point_1.coordinates[0],
                             point_2.coordinates[0])) {
    return point_1.id < point_2.id;
  }
  return point_1.coordinates[0] < point_2.coordinates[0];
});
```

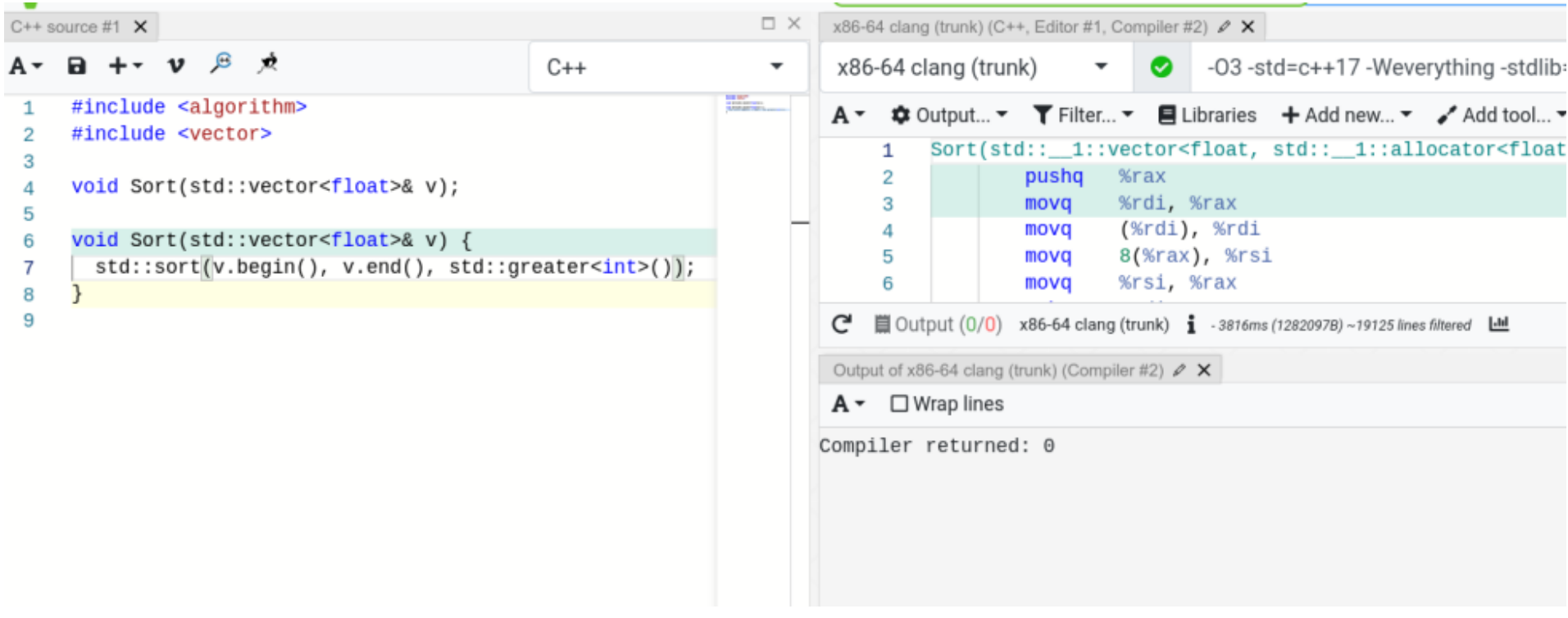
We found fewer of them but more dangerous.

(a is equal to $a + \epsilon$), ($a + \epsilon$ equals to $a + 2\epsilon$) but (a is no
equal to $a + 2\epsilon$). Transitivity of incomparability is lost

We found fewer of them but more dangerous.

```
std::vector<float> vector_with_possibly_nans = Fill();
absl::c_sort(vector_with_possibly_nans);
// Can be bug:
// As doubles/floats can be NaN which means x < NaN and NaN < x
// are both false and that means x is equivalent to NaN
// thus for every finite x we have x == NaN but clearly
// x == NaN and y == NaN does not imply x == y.
// If you are sure your sort does not have NaNs, no action needed.
```

We found fewer of them but more dangerous.



We found fewer of them but more dangerous.

Repo: android/platform/system/unwinding
Stars: N/A
Commit: d96368
File: libunwindstack/tools/unwind_reg_info.cpp
Snippet:

```
85    std::sort(loc_regs.begin(), loc_regs.end(), [](auto a, auto b) {
86      if (a.first == CFA_REG) {
87        return true;
88      } else if (b.first == CFA_REG) {
89        return false;
90      }
91      return a.first < b.first;
92    });
```

We found fewer of them but more dangerous.

Repo: iovisor/bpftrace
Stars: 5.2k
Commit: 384e4d
File: src/utils.cpp
Snippet:

```
465     // Sort paths lexically by name (with the exception of unified, which always
466     // comes first)
467     std::sort(result.begin(), result.end(), [](auto &pair1, auto &pair2) {
468       if (pair1.first == "unified")
469         return true;
470       else if (pair2.first == "unified")
471         return false;
472       else
473         return pair1.first < pair2.first;
474     });
```

We found fewer of them but more dangerous.

Repo: ROCmSoftwarePlatform/rocSPARSE
Stars: 2.4k
Commit: 584393
File: clients/common/rocsparse_host.cpp
Snippet:

```
3535
3536        rocsparse_int* col_entry = &col[row_begin];
3537        T*             val_entry = &val[row_begin];
3538
3539        std::sort(perm.begin(), perm.end(), [&](const rocsparse_int& a, const rocsparse_int& b) {
3540            return col_entry[a] <= col_entry[b];
3541        });
3542
3543        for(rocsparse_int j = 0; j < row_nnz; ++j)
3544        {
3545            csr_col_ind_C[row_begin + j] = col_entry[perm[j]];
3546            csr_val_C[row_begin + j]     = val_entry[perm[j]];
3547        }
3548    }
```

We found fewer of them but more dangerous.

Repo: chromium/webrtc
Stars: N/A
Commit: a7b919
File: rtc_base/network.cc
Snippet:

```
65  bool CompareNetworks(const Network* a, const Network* b) {
66    if (a->prefix_length() == b->prefix_length()) {
67      if (a->name() == b->name()) {
68        return a->prefix() < b->prefix();
69      }
70    }
71    return a->name() < b->name();
72  }
```

We found fewer of them but more dangerous.

```
// Self modifying comparator can lead to O(n^2) worst case
std::nth_element(tmp.begin(), tmp.end() - 2, tmp.end(), [&](int x,
  if (v[x] == gas && v[y] == gas) {
    if (x == candidate) v[x] = num_solid++;
    else v[y] = num_solid++;
  }
  if (v[x] == gas) candidate = x;
  else if (v[y] == gas) candidate = y;
  return v[x] < v[y];
});
```

We found fewer of them but more dangerous.

libc++ std::nth_element is quadratic, should be linear o
average #52747

SORTING TRIVIA

Can we sort **{1.02, 1.01, 1.0}**?

Can we sort **{1.02, 1.01, 1.0}**?

Yes

Can we sort **{3.0, NaN, 4.0}**?

Can we sort **{3.0, NaN, 4.0}**?

No

Can we sort **{3.0, NaN, 4.0}**?

No

NaN thinks it is equal to 3.0 and 4.0

Can we sort **{NaN, 3.0, NaN}**?

Can we sort **{NaN, 3.0, NaN}**?

Yes

Can we sort **{NaN, 3.0, NaN}**?

Yes

All elements are equal

WHAT ARE WE CHANGING?

WHAT ARE WE CHANGING?

- After these cleanups we can change to any sort

WHAT ARE WE CHANGING?

- After these cleanups we can change to any sort
- We chose a mix of BlockQuickSort and pdqsort

```
// *__m is median. partition [__first, __m) < *__m and
// *__m <= [__m, __last)
// Special handling for almost sorted targets
while (true) {
  while (__comp(*++__i, *__m));
  while (!__comp(*--__j, *__m));
  if (__i > __j) break;
  swap(*__i, *__j);
}
swap(*__i, *__m);
```

Data dependent branches are hard to predict

```
1  // *__m is median. partition [__first, __m) < *__m and
2  // *__m <= [__m, __last)
3  // Special handling for almost sorted targets
4  while (true) {
5    while (__comp(*++__i, *__m));
6    while (!__comp(*--__j, *__m));
7    if (__i > __j) break;
8    swap(*__i, *__j);
9  }
10 swap(*__i, *__m);
```

Data dependent branches are hard to predict

```
1  // *__m is median. partition [__first, __m) < *__m and
2  // *__m <= [__m, __last)
3  // Special handling for almost sorted targets
4  while (true) {
5    while (__comp(*++__i, *__m));
6    while (!__comp(*--__j, *__m));
7    if (__i > __j) break;
8    swap(*__i, *__j);
9  }
10 swap(*__i, *__m);
```

Pack results into blocks

```
1 uint64_t __left_bitset = 0;
2 _RandomAccessIterator __iter = __first;
3 for (int __j = 0; __j < __block_size;) {
4   bool __comp_result = !__comp(*__iter, __pivot);
5   __left_bitset |= (__comp_result << __j);
6   __j++;
7   ++__iter;
8 }
```

Pack results into blocks

```
1 uint64_t __left_bitset = 0;
2 _RandomAccessIterator __iter = __first;
3 for (int __j = 0; __j < __block_size;) {
4   bool __comp_result = !__comp(*__iter, __pivot);
5   __left_bitset |= (__comp_result << __j);
6   __j++;
7   ++__iter;
8 }
```

CTZ - Count Trailing Zeros, BLSR - Reset Lowest Set Bit

```
1 void __swap_bitset_pos(__first, __last, __left_bitset, __rig
2   while (__left_bitset != 0 && __right_bitset != 0) {
3     difference_type tz_left = __ctz(__left_bitset);
4     __left_bitset = __blsr(__left_bitset);
5     difference_type tz_right = __ctz(__right_bitset);
6     __right_bitset = __blsr(__right_bitset);
7     _VSTD::iter_swap(__first + tz_left, __last - tz_right);
8   }
9 }
```

CTZ - Count Trailing Zeros, BLSR - Reset Lowest Set Bit

```
1 void __swap_bitset_pos(__first, __last, __left_bitset, __rig
2   while (__left_bitset != 0 && __right_bitset != 0) {
3     difference_type tz_left = __ctz(__left_bitset);
4     __left_bitset = __blsr(__left_bitset);
5     difference_type tz_right = __ctz(__right_bitset);
6     __right_bitset = __blsr(__right_bitset);
7     _VSTD::iter_swap(__first + tz_left, __last - tz_right);
8   }
9 }
```

CTZ - Count Trailing Zeros, BLSR - Reset Lowest Set Bit

```
1 void __swap_bitset_pos(__first, __last, __left_bitset, __rig
2   while (__left_bitset != 0 && __right_bitset != 0) {
3     difference_type tz_left = __ctz(__left_bitset);
4     __left_bitset = __blsr(__left_bitset);
5     difference_type tz_right = __ctz(__right_bitset);
6     __right_bitset = __blsr(__right_bitset);
7     _VSTD::iter_swap(__first + tz_left, __last - tz_right);
8   }
9 }
```

CTZ - Count Trailing Zeros, BLSR - Reset Lowest Set Bit

```
1 void __swap_bitset_pos(__first, __last, __left_bitset, __rig
2   while (__left_bitset != 0 && __right_bitset != 0) {
3     difference_type tz_left = __ctz(__left_bitset);
4     __left_bitset = __blsr(__left_bitset);
5     difference_type tz_right = __ctz(__right_bitset);
6     __right_bitset = __blsr(__right_bitset);
7     _VSTD::iter_swap(__first + tz_left, __last - tz_right);
8   }
9 }
```

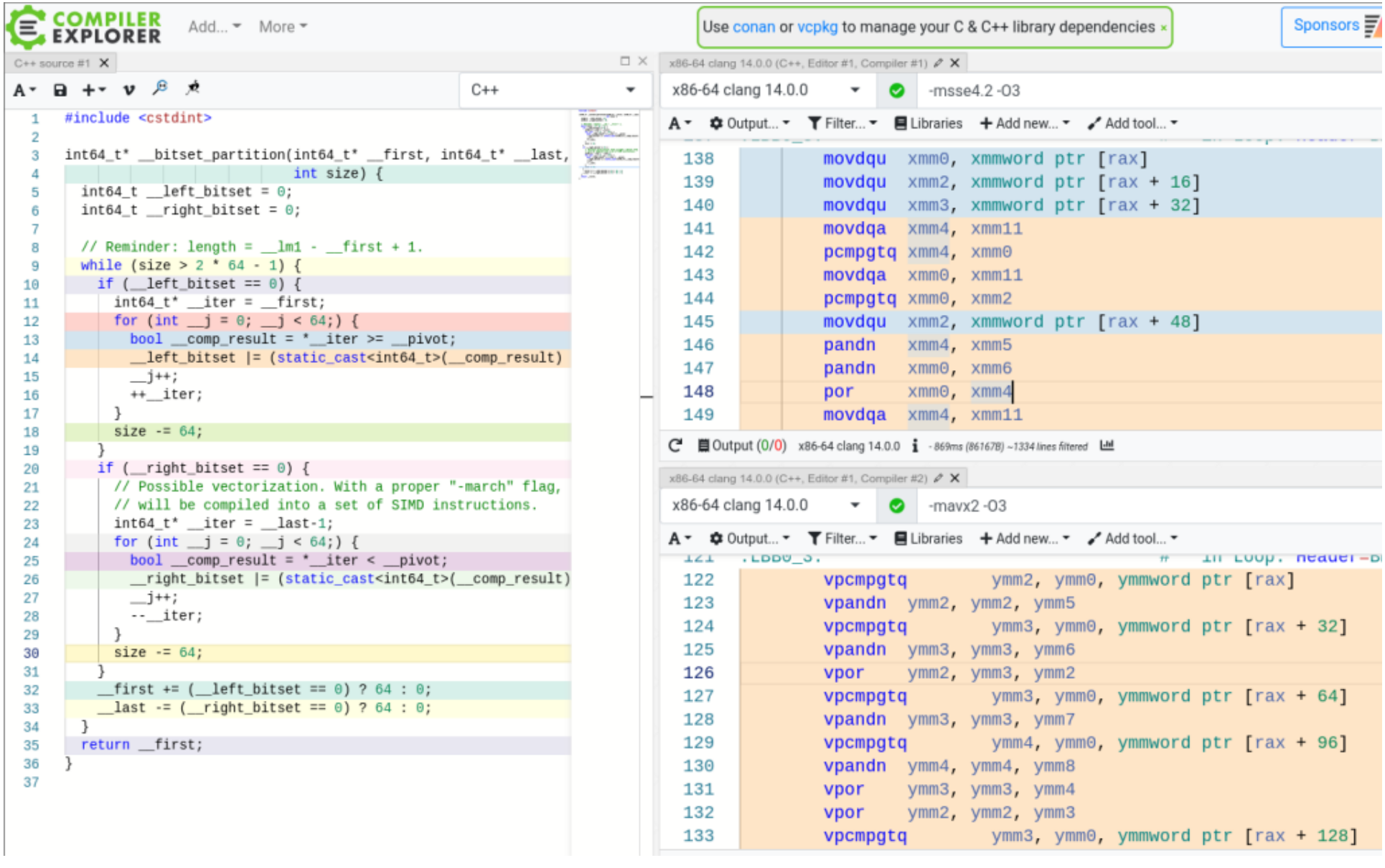
We also improved picking a pivot out of 9 elements. So called "Tukey's ninther"

```
 1 sort3(first,
 2       first + half_len,
 3       last - 1, comp);
 4 sort3(first + 1,
 5       first + (half_len - 1),
 6       last - 2, comp);
 7 sort3(first + 2,
 8       first + (half_len + 1),
 9       last - 3, comp);
10
11 sort3(first + (half_len - 1),
12       first + half_len,
13       first + (half_len + 1), comp);
```

We also improved picking a pivot out of 9 elements. So called "Tukey's ninther"

```
sort3(first,
      first + half_len,
      last - 1, comp);
sort3(first + 1,
      first + (half_len - 1),
      last - 2, comp);
sort3(first + 2,
      first + (half_len + 1),
      last - 3, comp);

sort3(first + (half_len - 1),
      first + half_len,
      first + (half_len + 1), comp);
```



https://godbolt.org/z/nrhT88MsM

```
                                    old           new
BM_Sort_uint64_Rand_1       4.26ns ± 0%  4.41ns ± 1%  3.44%
BM_Sort_uint64_Rand_4       1.96ns ± 0%  1.95ns ± 1%  -0.34%
BM_Sort_uint64_Rand_16      10.5ns ± 0%  11.6ns ± 1%  10.77%
BM_Sort_uint64_Rand_64      18.8ns ± 0%  17.5ns ± 1%  -6.60%
BM_Sort_uint64_Rand_256     26.4ns ± 0%  21.7ns ± 1%  -17.88%
BM_Sort_uint64_Rand_1024    33.8ns ± 1%  24.6ns ± 1%  -27.15%
BM_Sort_uint64_Rand_16384   48.7ns ± 0%  30.1ns ± 1%  -38.10%
BM_Sort_uint64_Rand_262144  63.4ns ± 1%  35.7ns ± 1%  -43.71%
```

On other patterns we sometimes regressed but by a little, hardest cases are random.

```
BM_Sort_string_Random_1       4.66ns ± 6%  4.40ns ± 4%    -5.55%
BM_Sort_string_Random_4       14.9ns ± 3%  15.0ns ± 6%      ~
BM_Sort_string_Random_16      45.5ns ± 6%  35.8ns ± 8%   -21.37%
BM_Sort_string_Random_64      66.6ns ± 4%  58.2ns ± 3%   -12.69%
BM_Sort_string_Random_256     86.0ns ± 5%  77.4ns ± 3%   -10.01%
BM_Sort_string_Random_1024    106ns ± 3%    96ns ± 6%    -9.39%
BM_Sort_string_Random_16384   154ns ± 3%   141ns ± 5%    -8.03%
BM_Sort_string_Random_262144  213ns ± 4%   197ns ± 4%    -7.59%
```

SOME THOUGHTS

SOME THOUGHTS

- Default for standard sorting is unstable

SOME THOUGHTS

- Default for standard sorting is unstable
- Python, Rust, Java have stable default sorts

SOME THOUGHTS

- Default for standard sorting is unstable
- Python, Rust, Java have stable default sorts
- Why do we have std::sort that can read OOB even if comparator is broken?

SOME THOUGHTS

- Default for standard sorting is unstable
- Python, Rust, Java have stable default sorts
- Why do we have std::sort that can read OOB even if comparator is broken?
- std::ranges::sort wanted to have 1 argument, found 0 bugs with different iterators in std::sort

SOME MORE THOUGHTS

- std::weak_ordering in C++20 allows to make fewer bugs

```
struct Weak {
  bool operator==(Weak const&) const;
  std::weak_ordering operator<=>(Weak const&) const;
};

struct Foo {
  Weak w;
  int i;
  auto operator<=>(Foo const&) const = default;
};
```

RESULTS

RESULTS

- We called it a **comparator sanitizer**

RESULTS

- We called it a **comparator sanitizer**
- We saw 7-8% perf improvement in production

RESULTS

- We called it a **comparator sanitizer**
- We saw 7-8% perf improvement in production
- Reduced branch mispredictions a lot

RESULTS

- We called it a **comparator sanitizer**
- We saw 7-8% perf improvement in production
- Reduced branch mispredictions a lot
- Fixed thousands of bugs

RESULTS

- We called it a **comparator sanitizer**
- We saw 7-8% perf improvement in production
- Reduced branch mispredictions a lot
- Fixed thousands of bugs
- Fought the Hyrum's Law

RESULTS

- We called it a **comparator sanitizer**
- We saw 7-8% perf improvement in production
- Reduced branch mispredictions a lot
- Fixed thousands of bugs
- Fought the Hyrum's Law
- Even simplest things are hard to do right

RESULTS

- We called it a **comparator sanitizer**
- We saw 7-8% perf improvement in production
- Reduced branch mispredictions a lot
- Fixed thousands of bugs
- Fought the Hyrum's Law
- Even simplest things are hard to do right
- Things take time (2.5 years)

WHAT CAN YOU DO?

WHAT CAN YOU DO?

- If you use libcxx, enable **-D_LIBCPP_DEBUG_RANDOMIZE_UNSPECIFIED_STAB -D_LIBCPP_DEBUG_STRICT_WEAK_ORDERING_CHEC D_LIBCPP_ENABLE_DEBUG_MODE=1** in your **debug builds**

WHAT CAN YOU DO?

- If you use libcxx, enable **-D_LIBCPP_DEBUG_RANDOMIZE_UNSPECIFIED_STAB -D_LIBCPP_DEBUG_STRICT_WEAK_ORDERING_CHEC D_LIBCPP_ENABLE_DEBUG_MODE=1** in your **debug builds**
- Probably randomization can be applied to std::partitio std::make_heap

WHAT CAN YOU DO?

- If you use libcxx, enable **-D_LIBCPP_DEBUG_RANDOMIZE_UNSPECIFIED_STAB -D_LIBCPP_DEBUG_STRICT_WEAK_ORDERING_CHEC D_LIBCPP_ENABLE_DEBUG_MODE=1** in your **debug builds**
- Probably randomization can be applied to std::partitio std::make_heap
- SWO checks for std::nth_element, etc

WHAT CAN YOU DO?

WHAT CAN YOU DO?

- Remember Bjarne talked about invalidation of vector iterators?

WHAT CAN YOU DO?

- Remember Bjarne talked about invalidation of vector iterators?
- In debug mode reallocate on first 100 push_backs

LINKS AND FURTHER

- danlark.org blog about sorting
- LLVM RFC for strict weak ordering checks
- Sorting change itself with benchmarks, etc

- Personal Email: kutdanila@gmail.com
- Work Email: danilak@google.com
- Github: @Danlark1
- Telegram: @Danlark