A Little Order!

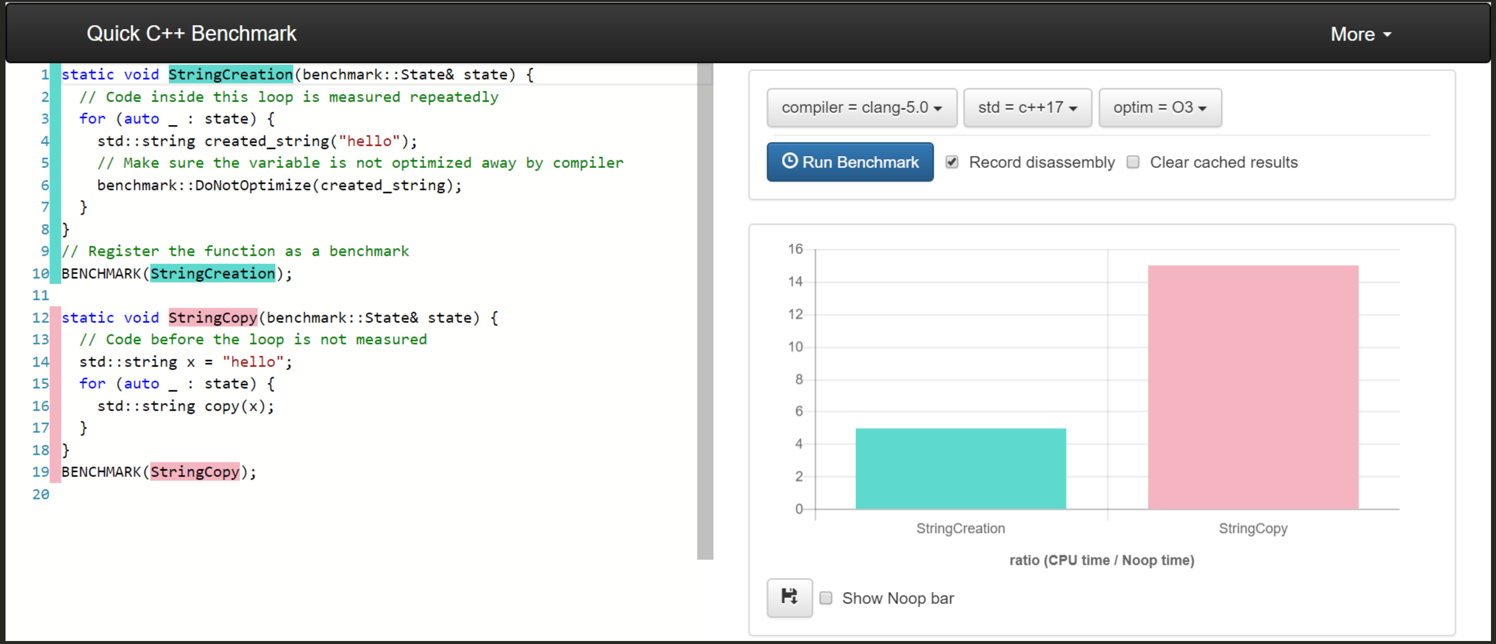
Delving into the STL sorting algorithms

Fred Tingaud

@FredTingaudDev

quick-bench.com

quick-bench.com



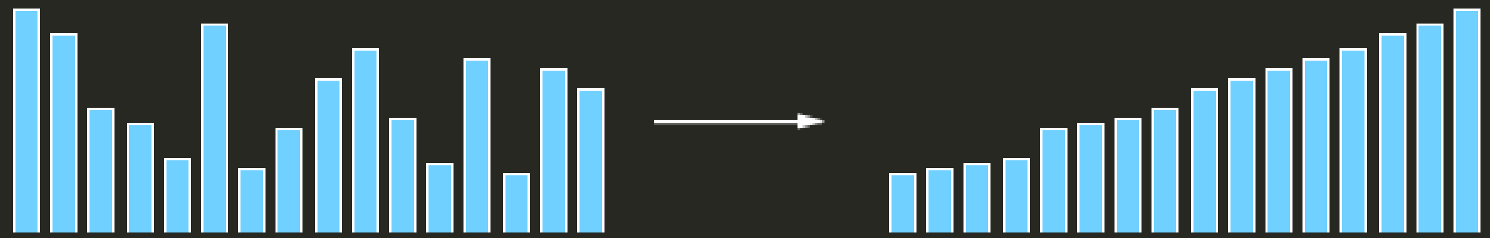
A use case

Sorting a 1,000,000 elements vector to get the median.

Clang 3.8

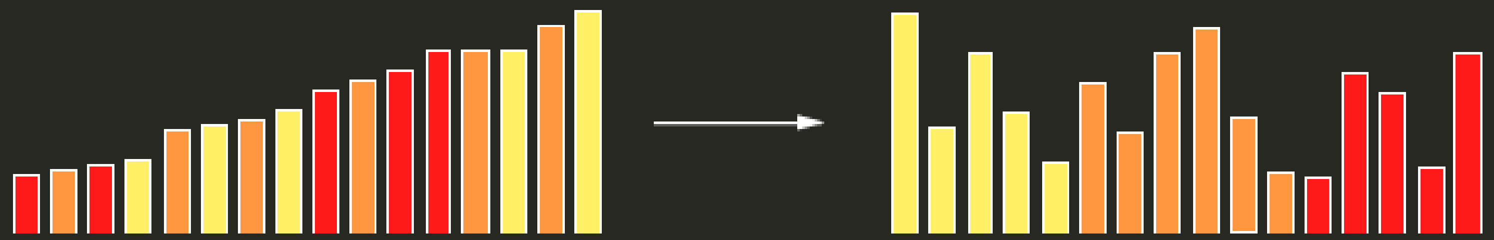
GNU's libstdcxx

std::sort

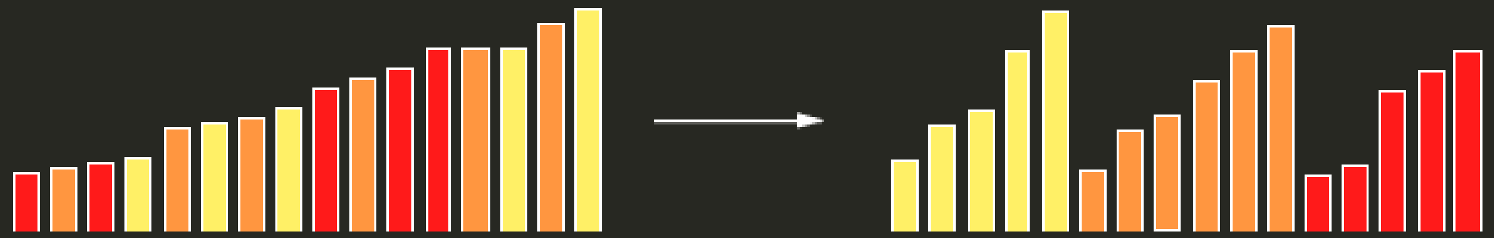


std::stable_sort

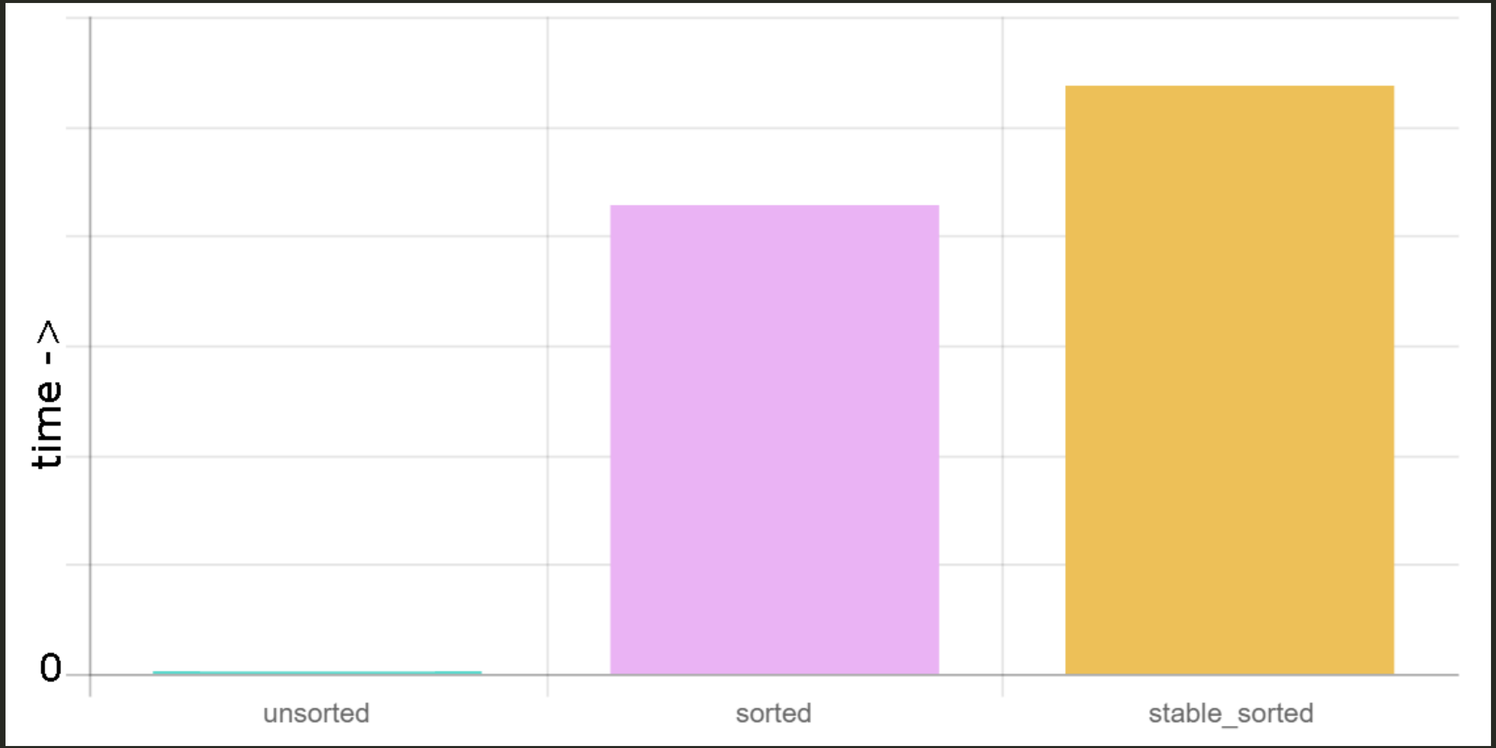
`std::sort`



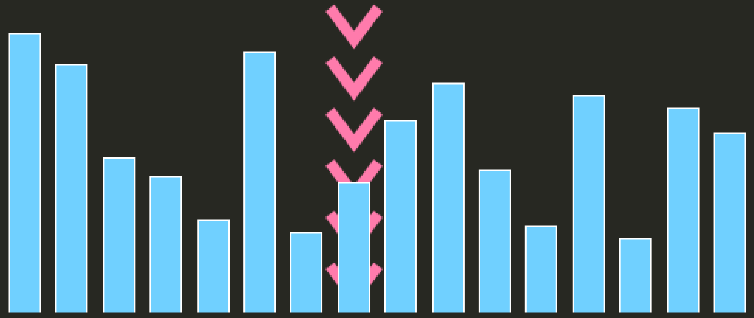
`std::stable_sort`



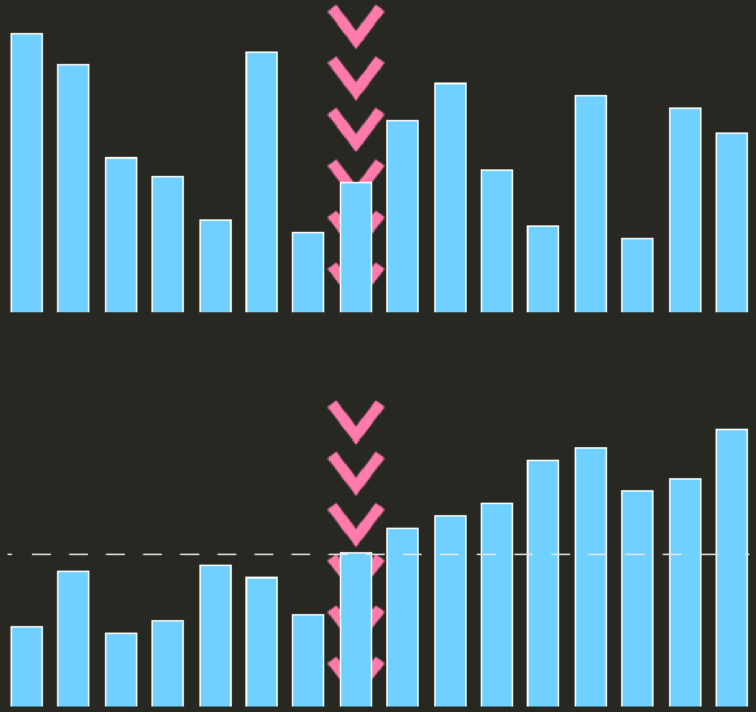
std::stable_sort



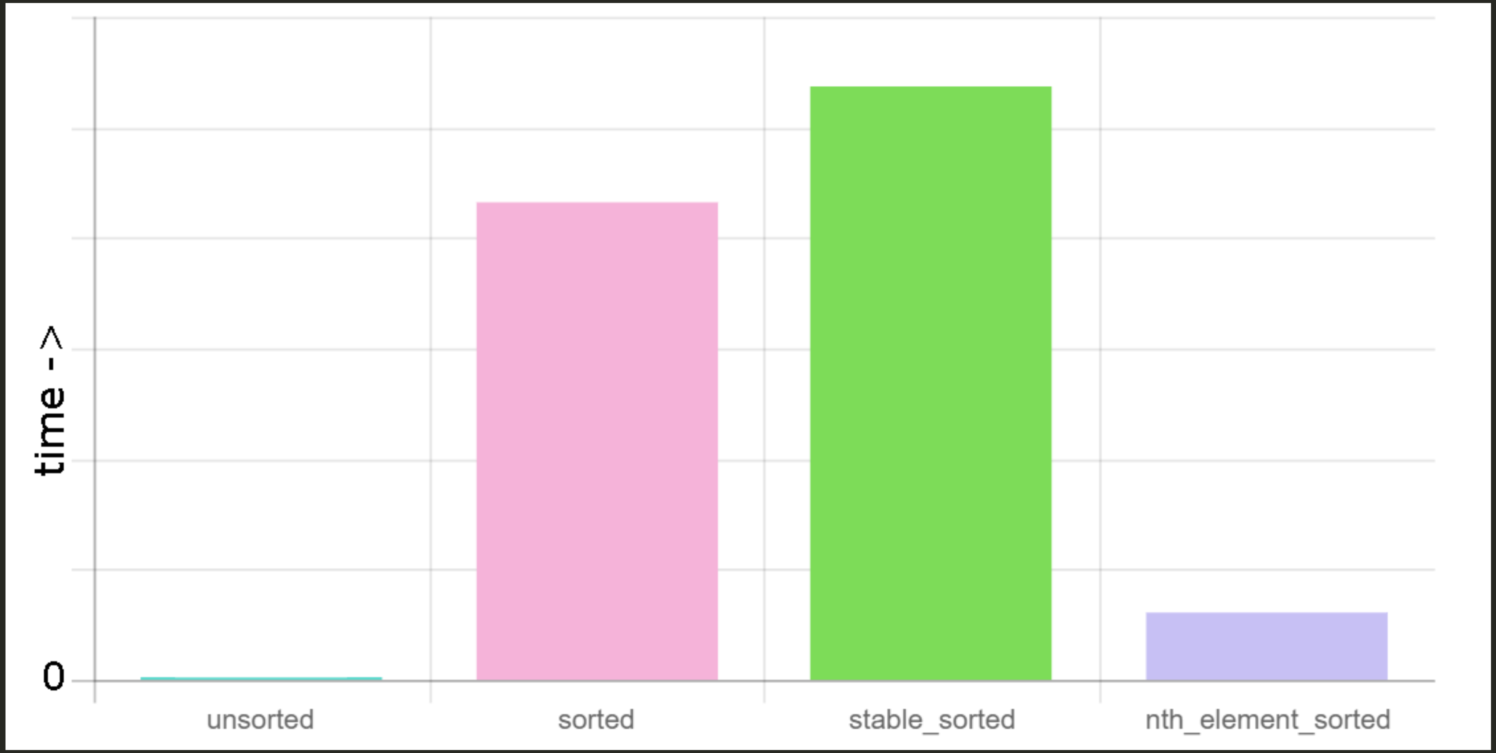
std::nth_element



std::nth_element

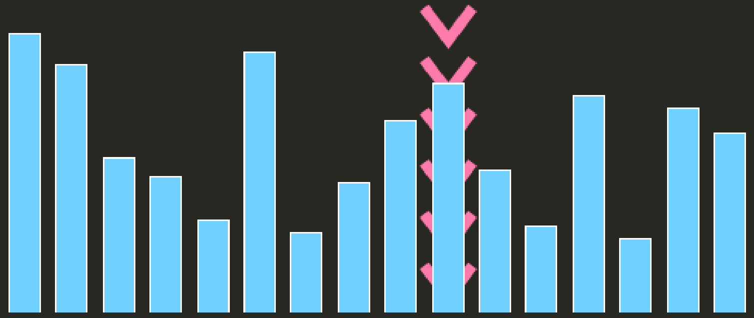


std::nth_element

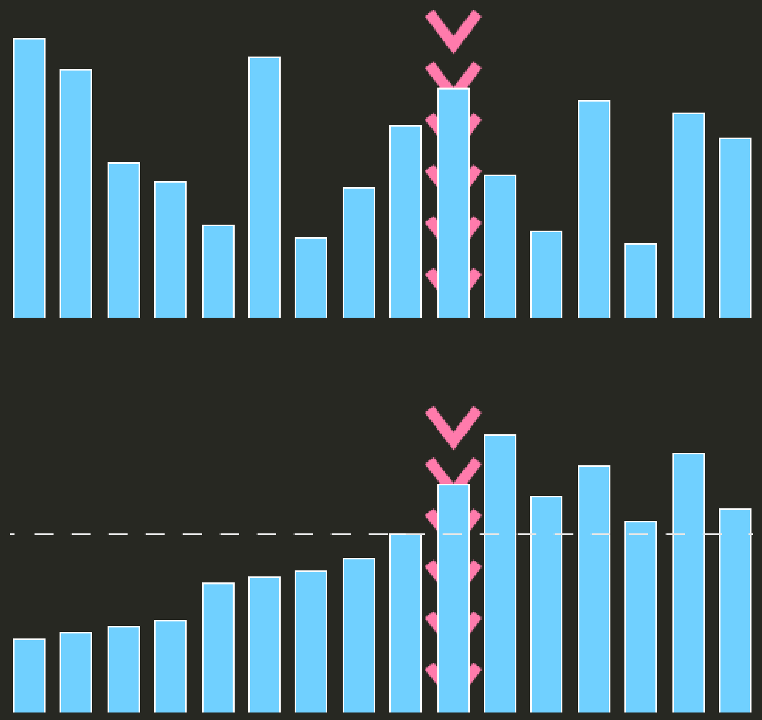


nth_element is 10x faster than sort

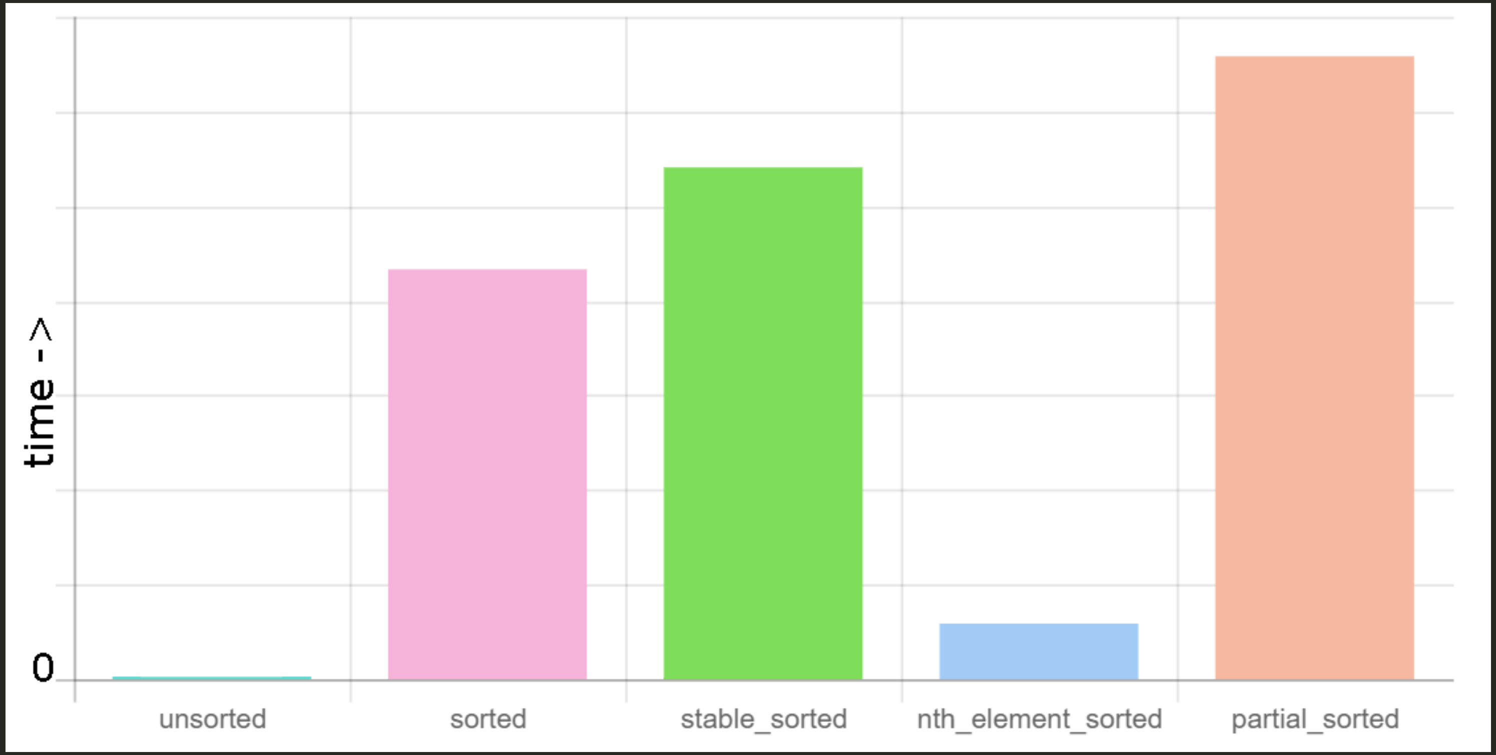
std::partial_sort



std::partial_sort



std::partial_sort

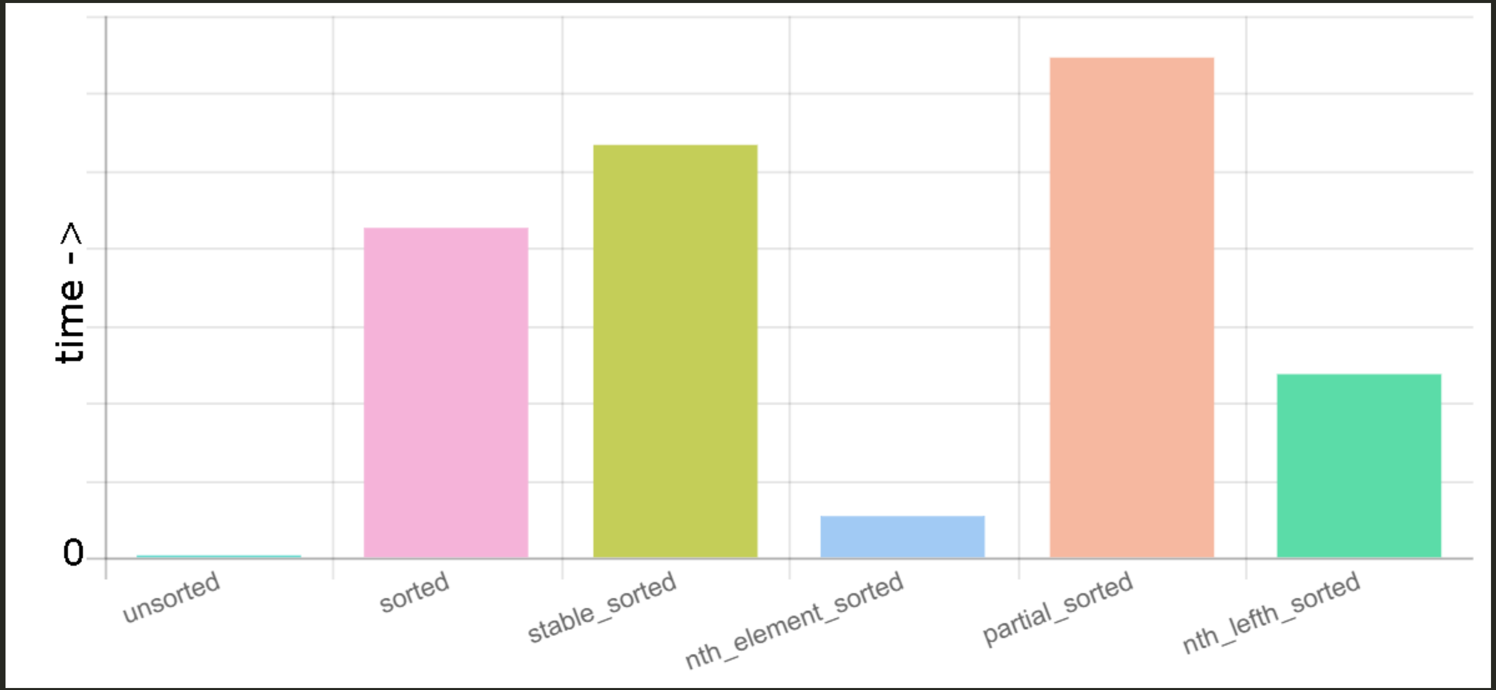


Home-cooked partial sort

What if we just call `std::sort` on the result of `std::nth_element`?

It should do the same as `std::partial_sort`…

Home-cooked partial sort



Home-cooked partial sort

Did we do better than the STL implementers in 5 minutes and 2 lines of code?

Home-cooked partial sort

Did we do better than the STL implementers in 5 minutes and 2 lines of code?

Probably not…

Home-cooked partial sort

Did we do better than the STL implementers in 5 minutes and 2 lines of code?

Probably not…

Let's try to understand why!

Check the contract!

Read the Standards

- Draft: https://github.com/cplusplus/draft
- C++ Reference: https://cppreference.com

Read the Standards - 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, RandomIt first, RandomIt last );         (2) (since C++17)
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, RandomIt first, RandomIt last, Compare comp ); (4) (since C++17)
```

Sorts the elements in the range `[first, last)` in ascending order. The order of equal elements is not guaranteed to be preserved.

1) Elements are compared using `operator<`.

3) Elements are compared using the given binary comparison function `comp`.

2,4) Same as (1,3), but executed according to `policy`. These overloads do not participate in overload resolution unless `std::is_execution_policy_v<std::decay_t<ExecutionPolicy>>` is true

Parameters

first, last - the range of elements to sort

policy - the execution policy to use. See execution policy for details.

comp - comparison function object (i.e. an object that satisfies the requirements of *Compare*) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of *ValueSwappable* and *RandomAccessIterator*.
- The type of dereferenced `RandomIt` must meet the requirements of *MoveAssignable* and *MoveConstructible*.
- `Compare` must meet the requirements of *Compare*.

Return value

(none)

Complexity

$O(N \cdot log(N))$, where `N = std::distance(first, last)` comparisons on average.	(until C++11)
$O(N \cdot log(N))$, where `N = std::distance(first, last)` comparisons.	(since C++11)

Exceptions

Standards - `std::sort`

Complexity

`O(N·log(N))`, on average (until C++11).

`O(N·log(N))`(since C++11).

Standards - `std::partial_sort`

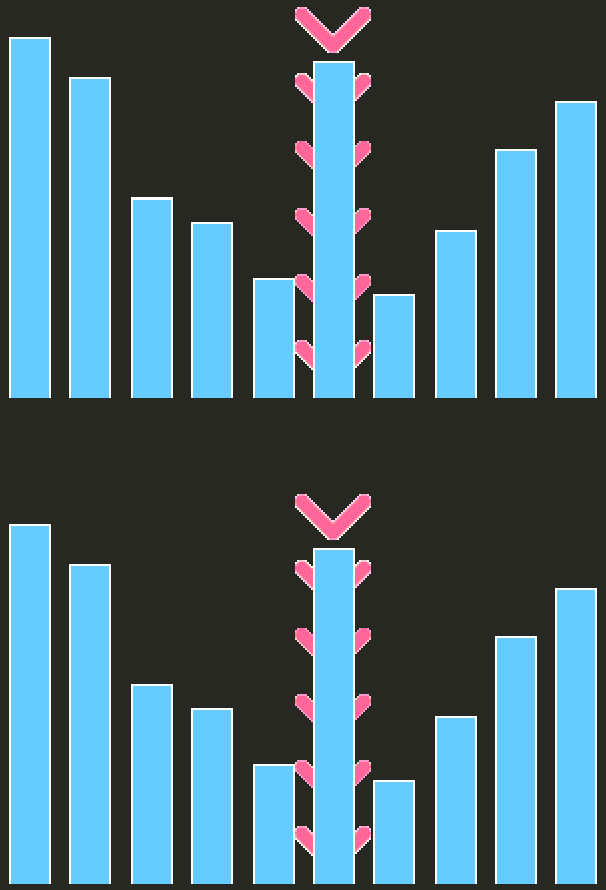
Complexity

Approximately (last-first)log(middle-first)) applications of cmp

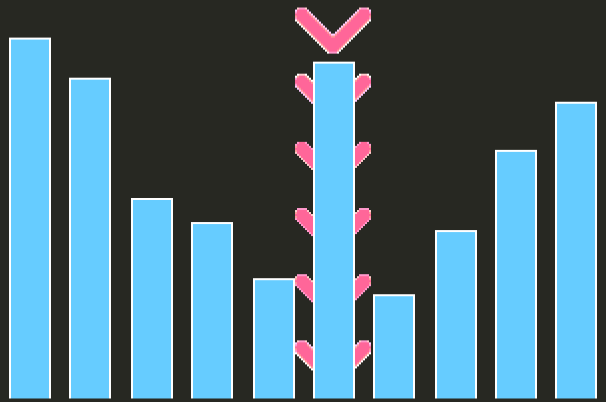
`O(N·log(k))`

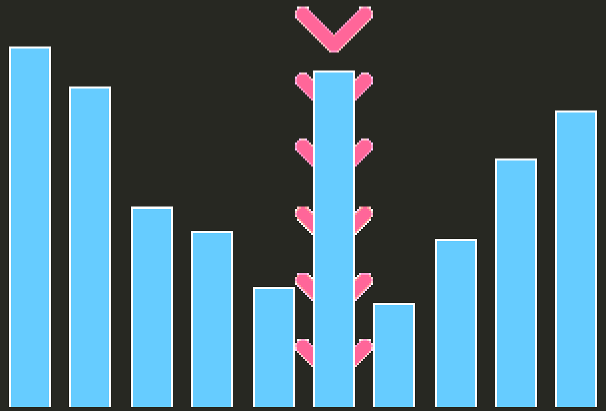
- N = full container size
- k = subset size

Two complexities



Two complexities





Read the Standards - Partial Sort

For a median, k = N / 2

`O(N·log(k))` <=> `O(N·log(N / 2))`

Read the Standards - Partial Sort

For a median, k = N / 2

`O(N·log(N))`

Read the Standards - Partial Sort

For a median, k = N / 2

`O(N·log(N))`

Just like `std::sort`

Read the Standards - Partial Sort

If we want to sort the 10 best scores in a MMO scoreboard.

k = 10

`O(N·log(10))`

Read the Standards - Partial Sort

If we want to sort the 10 best scores in a MMO scoreboard.

k = 10

`O(N)`

Read the Standards - Nth Element + Sort

The contract of Nth Element:

`O(N)` on average

So Nth Element + Sort on the result:

`O(N + k·log(k))`

Read the Standards - Nth Element + Sort

When k = N / 2

`O(N + N/2·log(N/2))` <=> `O(N·log(N))`

When k = 10

`O(N + 10·log(10))` <=> `O(N)`

Read the Standards

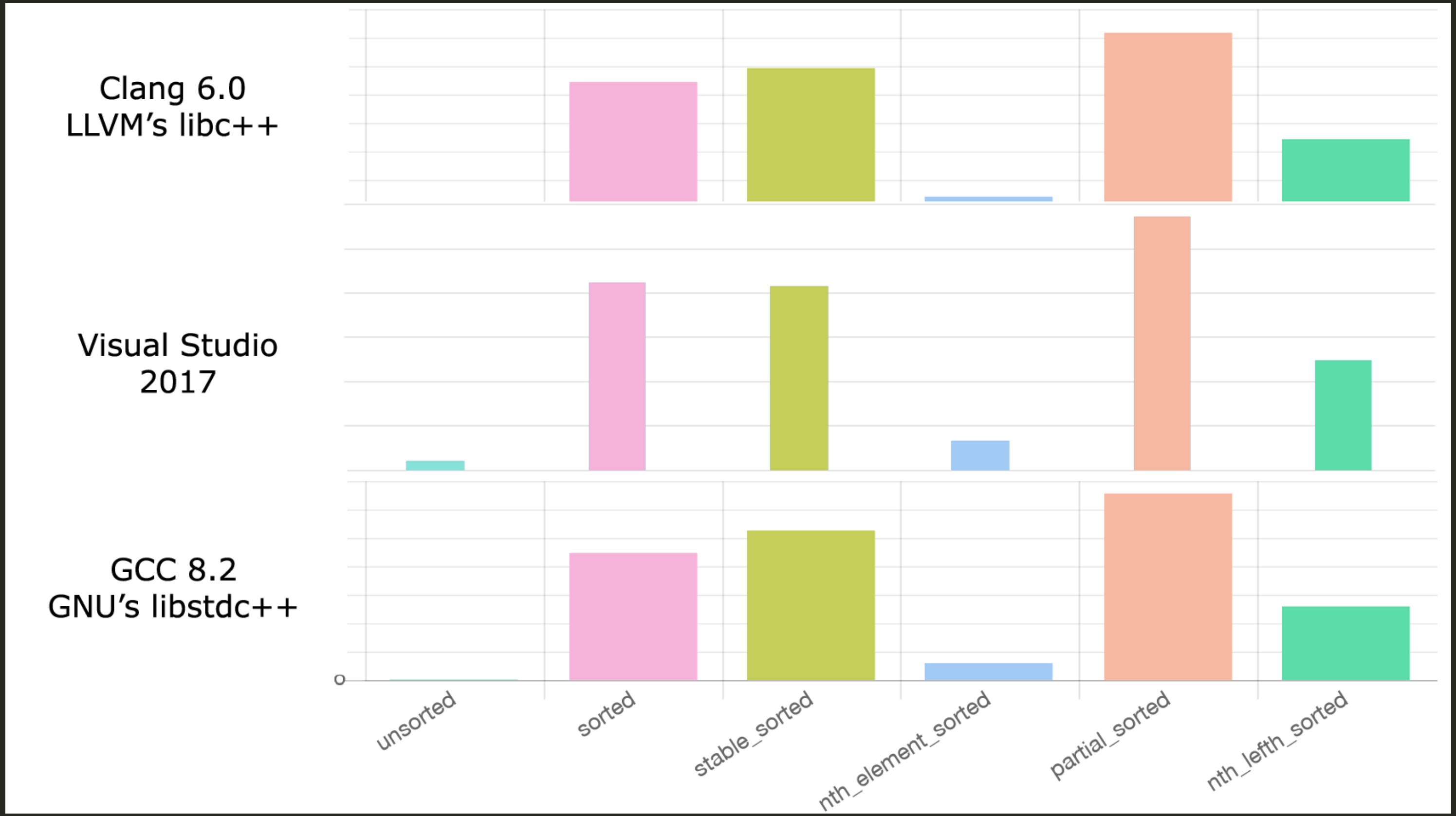
Same complexities for `std::partial_sort` and our algorithm.

Compare implementations / compilers

Comparing implementations

- Compiler Explorer: https://godbolt.org
- Wandbox: https://wandbox.org

Comparing implementations



Comparing implementations

Same results for all implementations of the libc++.

Measure

Measure

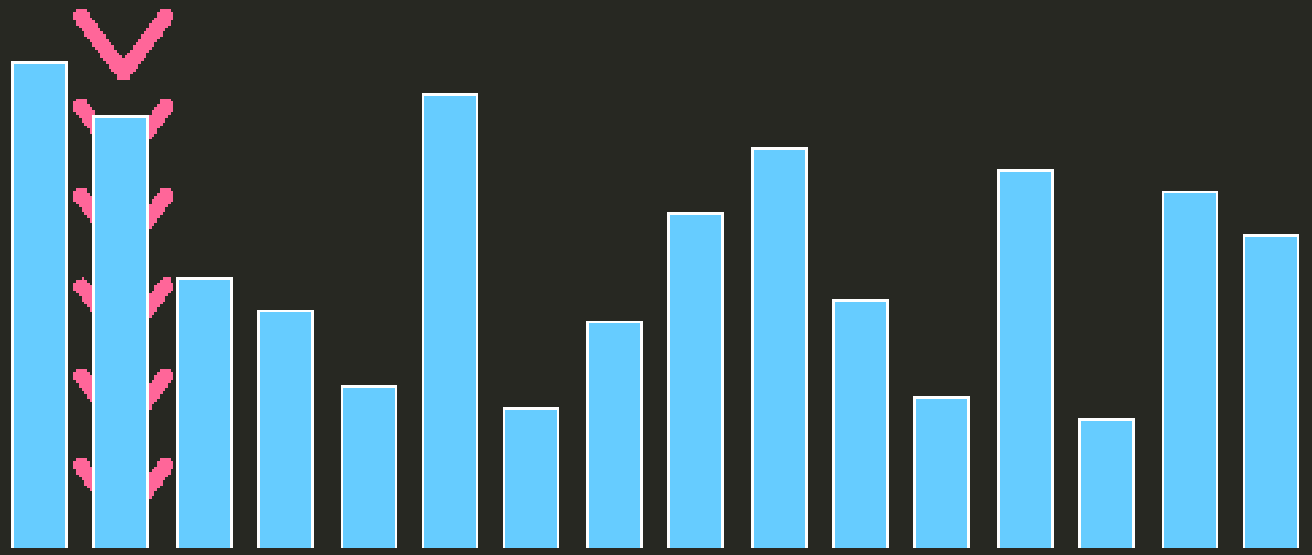
- Google Benchmark: https://github.com/google/benchmark
- Quick Bench: http://quick-bench.com

Measure Size Variation

We are sorting a subset inside a container.

- Subset size may vary.
- Container size may vary.
- Both may vary.

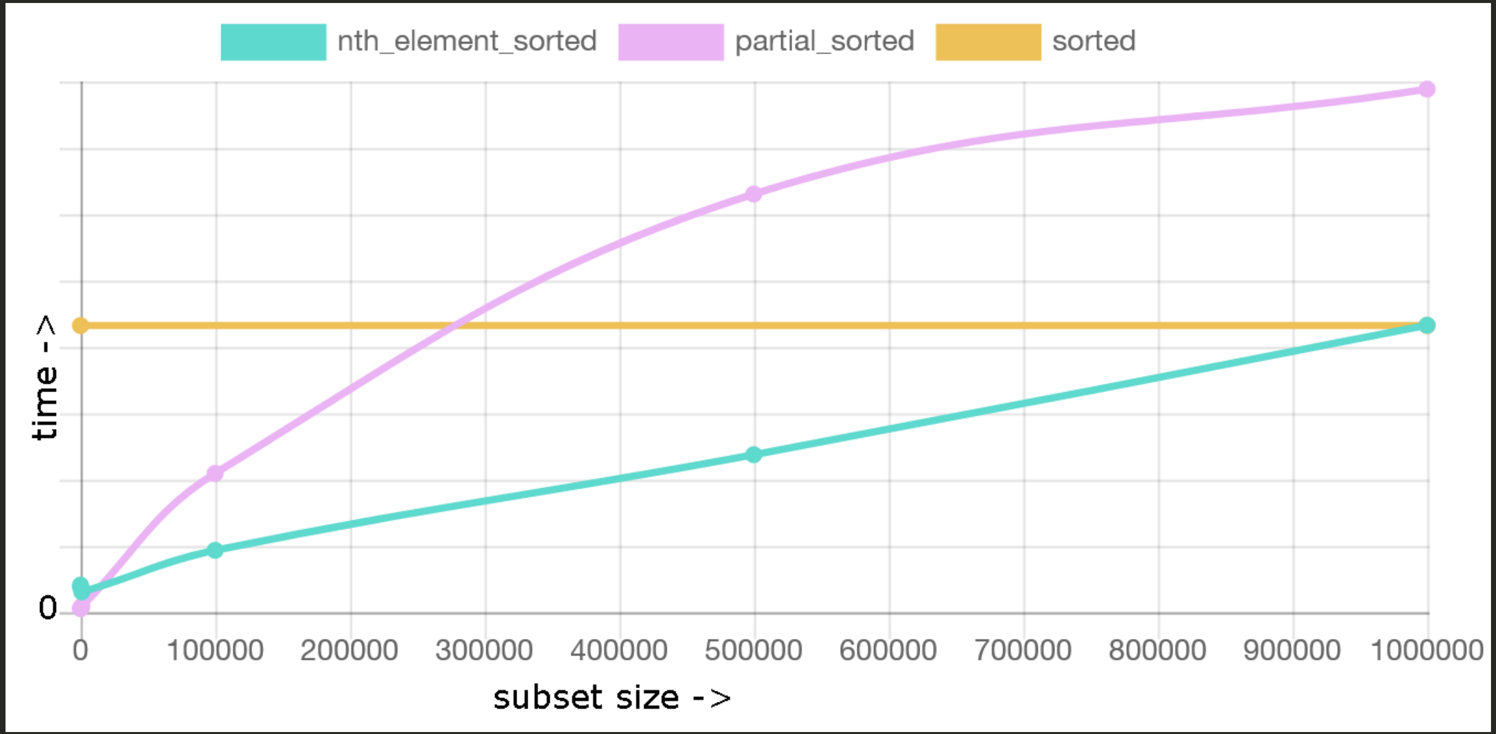
Measure Size Variation - Subset



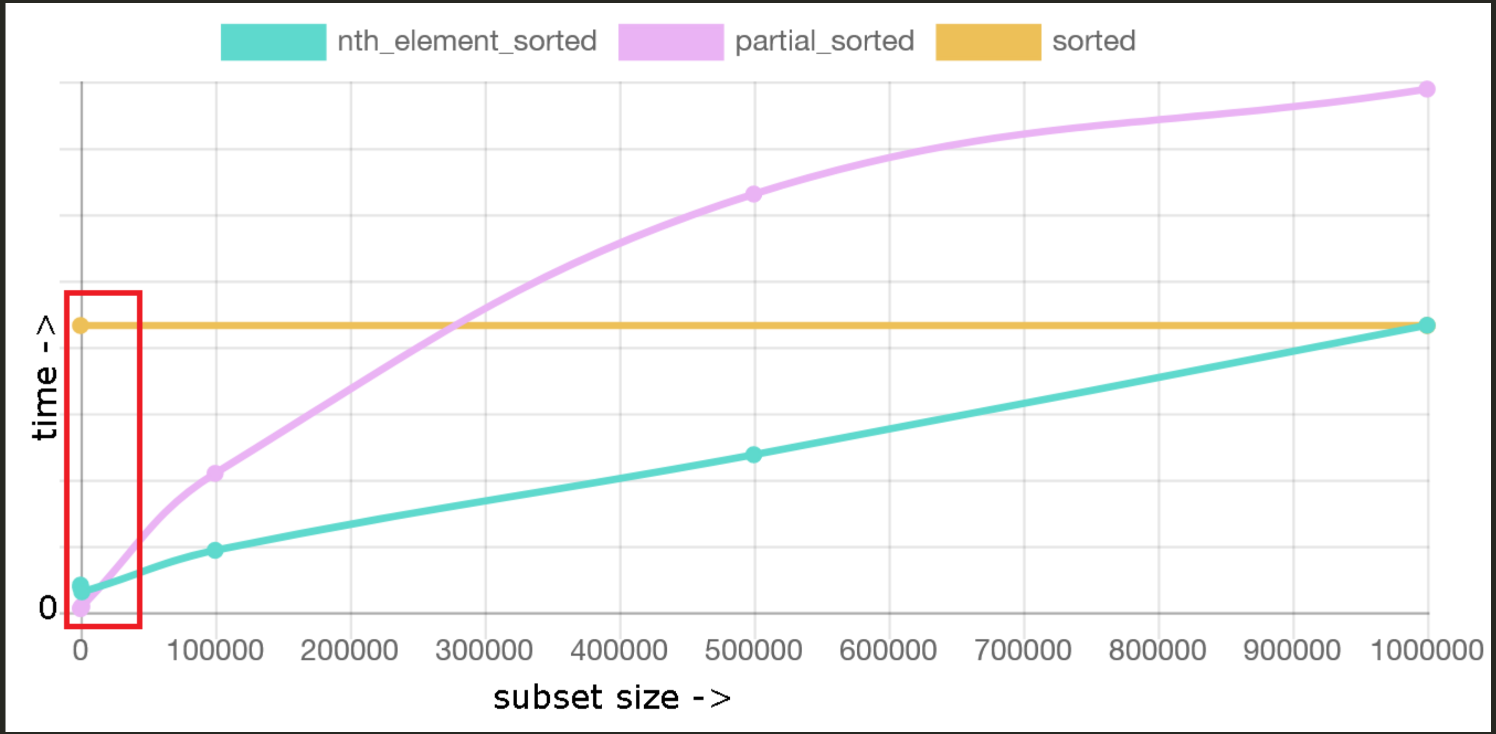
Measure Size Variation - Subset

- 1,000,000 elements
- We vary the number of elements we sort in it

Measure Size Variation - Subset

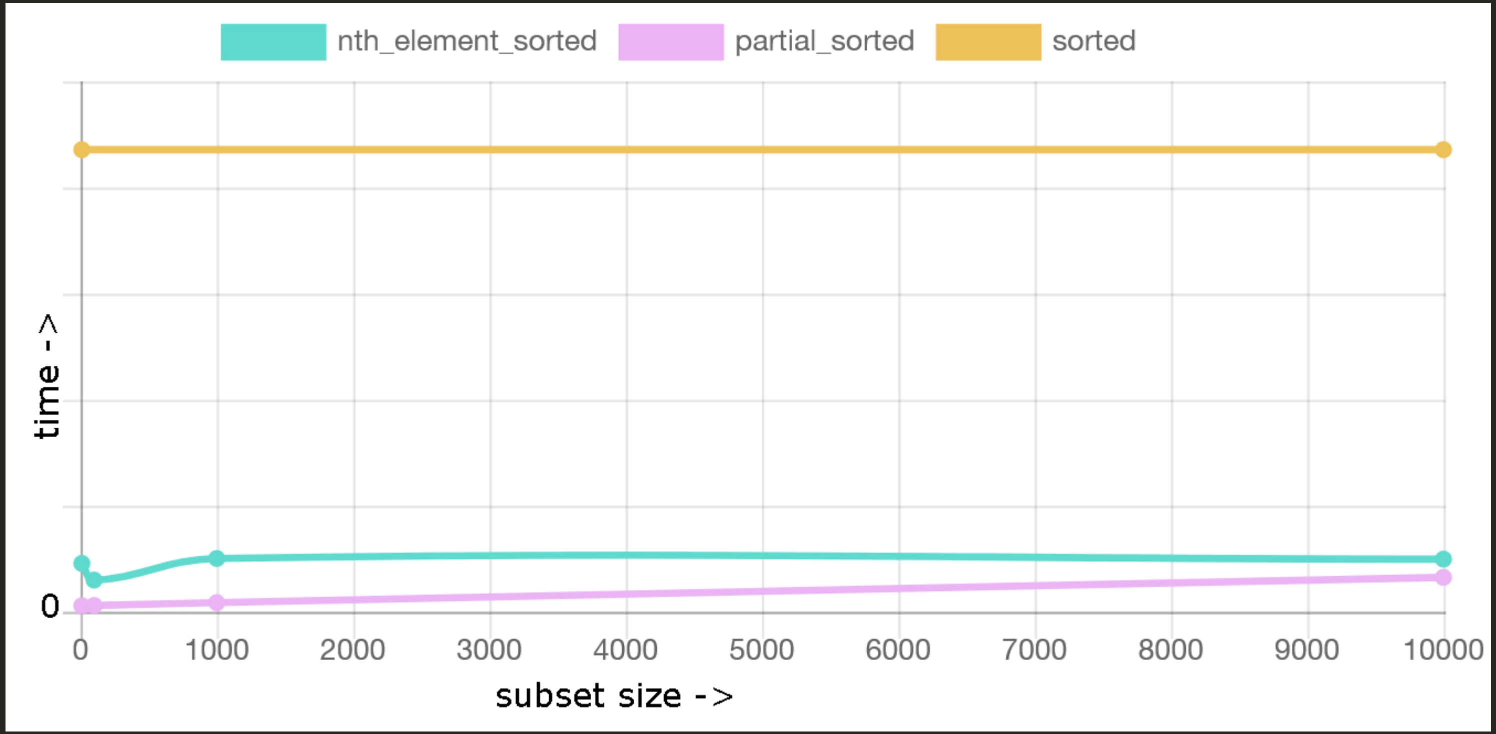


Measure Size Variation - Subset

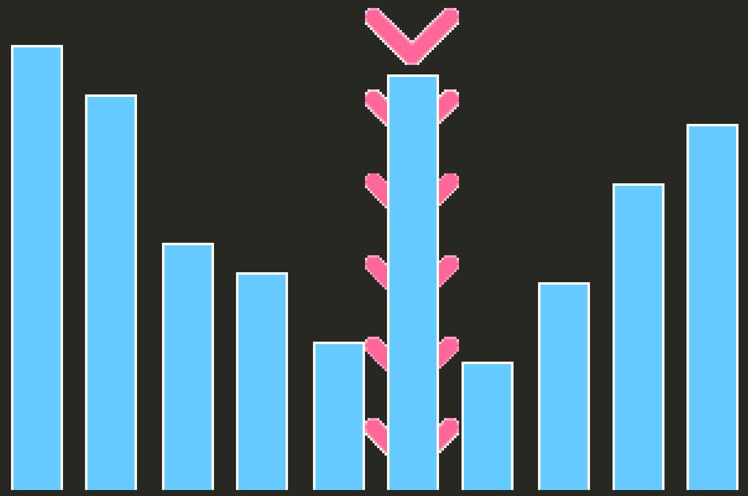


Measure Size Variation - Subset

Zoomed



Measure Size Variation - Container

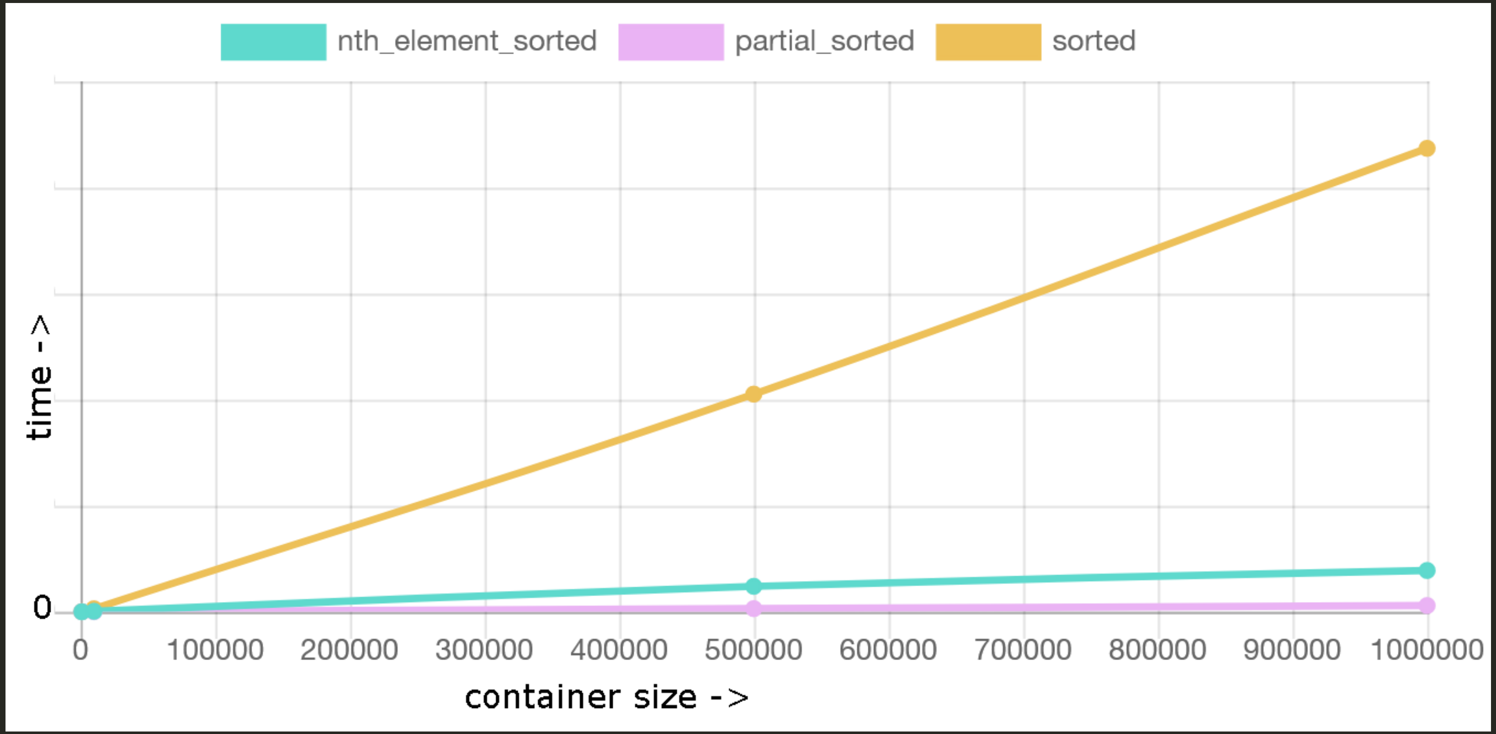


Measure Size Variation - Container

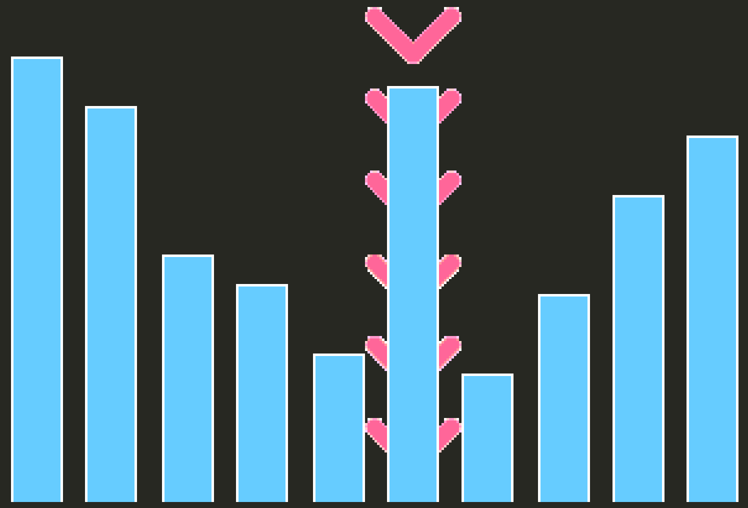
- We sort 100 elements
- We vary the size of the full container

Measure Size Variation - Container

k = 100



Measure Size Variation - Both

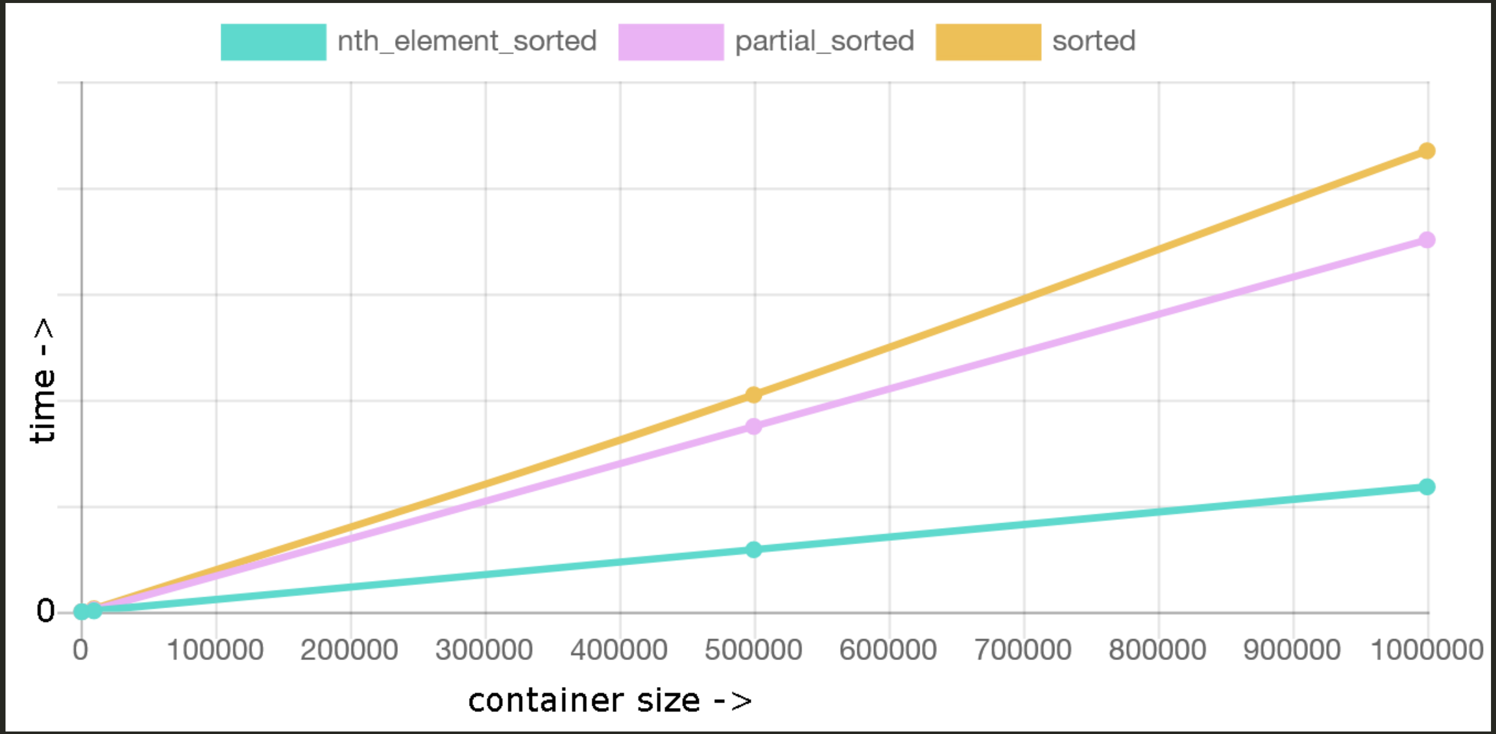


Measure Size Variation - Both

- We sort 1/5th of the elements
- We vary the size of the container

Measure Size Variation - Both

k = N / 5



Measure Size Variation

We have an actual difference!

When to use `std::partial_sort`

Use `partial_sort` when sorting a subset that is considerably smaller than the whole container.

Otherwise, use `nth_element + sort`.

Why?

How is partial_sort faster than nth_element for lower values of k?

Why is it slower for higher values?

Read The Code

Read The Code

- GCC - https://github.com/gcc-mirror/gcc
- LLVM's libcxx - https://github.com/llvm-mirror/libcxx
- Visual Studio - `F12`

`std::sort`

```
template<typename RandIt, typename _Compare>
  inline void
  __sort(RandIt first, RandIt last,
         _Compare comp) {
  if (first != last){
    std::__introsort_loop(first, last,
                          std::__lg(last-first)*2,
                          comp);
    std::__final_insertion_sort(first, last, comp);
  }
}
```

std::sort

Complexity

`O(N·log(N))`, on average (until C++11).

`O(N·log(N))`(since C++11).

Introsort

- Quicksort is very fast for most scenarios, but it can become `O(N²)` on worst-case scenarios.
- Heapsort is always `O(N·log(N))` but it takes 4-5 times longer to sort typical scenario.

Introsort

- Quicksort is very fast for most scenarios, but it can become `O(N²)` on worst-case scenarios.
- Heapsort is always `O(N·log(N))` but it takes 4-5 times longer to sort typical scenario.

Introsort does BOTH!

Introsort

Quicksort recurses on `2 * log(N)` levels max.

Then Heapsort is called on the rest in case it's still not sorted.

`O(N·log(N))` in all cases.

```cpp
template<typename RandIt, typename _Compare>
  inline void
  sort(RandIt first, RandIt last,
         _Compare comp) {
  if (first != last){
    std::__introsort_loop(first, last,
                          std::__lg(last-first)*2,
                          comp);
    std::__final_insertion_sort(first, last, comp);
  }
}
```

Insertion Sort

`O(N²)` algorithm.

Insertion Sort

`O(N²)` algorithm.

Over small subranges, Insertion Sort performs better than Quicksort.

We sort until the size of subranges are < k.

`std::nth_element`

```cpp
template<typename RandIt>
  inline void
  nth_element(RandIt first, RandIt nth,
              RandIt last)
{
  if (first == last || nth == last)
    return;

  std::__introselect(first, nth, last,
             std::__lg(last - first) * 2,
             __gnu_cxx::__ops::__iter_less_iter());
}
```

Introselect

- Quickselect up to `2 * log(N)` recursions.
- Then Heapselect.

Quickselect

Simplified Quicksort.

We choose a pivot and we partition like Quicksort.

Quickselect

Simplified Quicksort.

We choose a pivot and we partition like Quicksort.

If the pivot ends up at nth position, we are done.

Quickselect

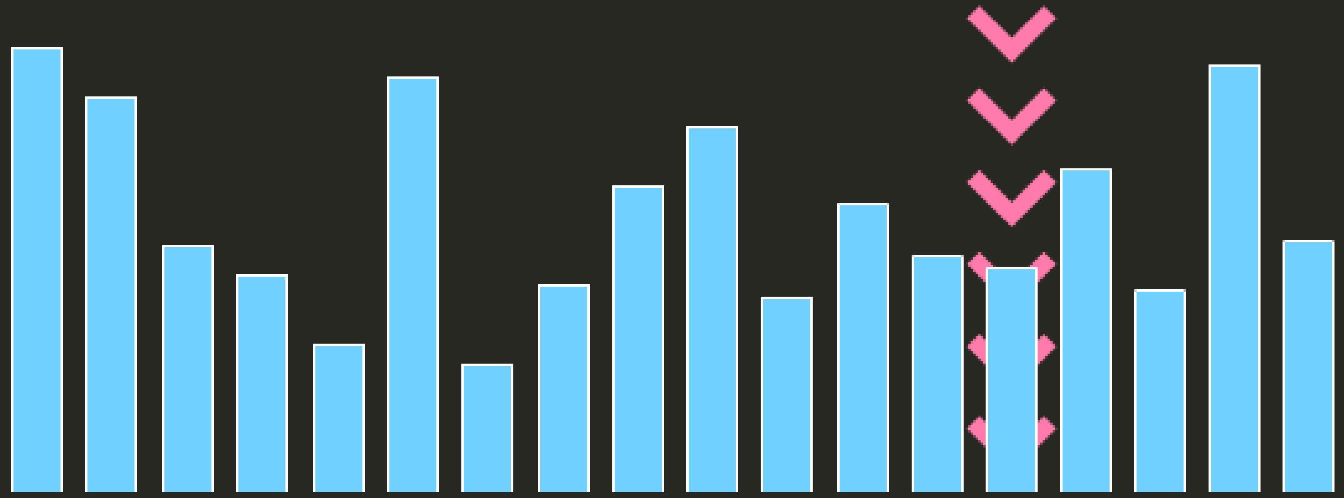
Simplified Quicksort.

We choose a pivot and we partition like Quicksort.

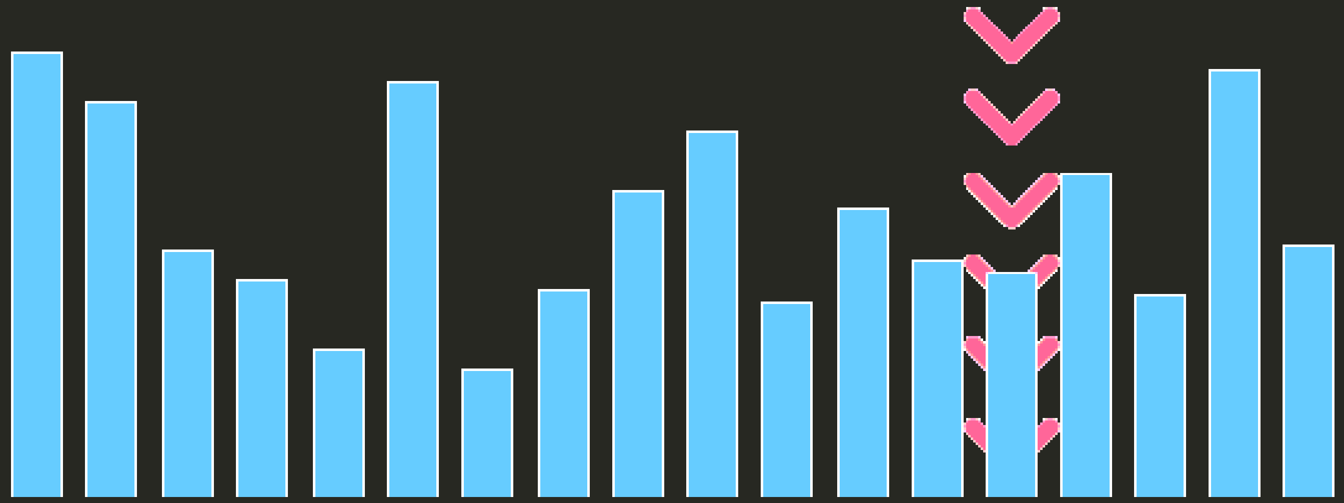
If the pivot ends up at nth position, we are done.

Otherwise, we recurse only on the side that contains nth position.

Quickselect



Quickselect



Heapselect

We create a heap on n elements.

Heapselect

We create a heap on n elements.

We iterate on all other elements.

Heapselect

We create a heap on n elements.

We iterate on all other elements.

If an element is smaller than heap max, we pop the heap and add the new element.

Heapselect

We create a heap on n elements.

We iterate on all other elements.

If an element is smaller than heap max, we pop the heap and add the new element.

Heap max is the nth element at the end.

Heapselect

- `O(N·log(k))`.

`std::partial_sort`

GCC Implementation

```
template<typename RandIt, typename _Compare>
  inline void
  __partial_sort(RandIt first,
                 RandIt middle,
                 RandIt last,
                 _Compare comp)
{
  std::__heap_select(first, middle, last, comp);
  std::__sort_heap(first, middle, comp);
}
```

Partial sort

We said that heapsort was slower than quicksort.

Heapselect is `O(N·log(k))` when quickselect is `O(N)`.

How come partial_sort can sometimes perform better than quickselect + quicksort?!

Heapselect benchmark

- 1.000.000 elements
- We vary the position we are searching

Heapselect benchmark

Partial sort

- Heapselect performs better with low `k` value
- Heapselect has bigger complexity.

Context of usage

Nth element

Nth element has valid use-cases for any value of `k`.

First decile, last decile, median…

The STL implementers chose a `O(N)` algorithm that doesn't depend on `k`.

Partial Sort

The typical usage of `std::partial_sort` is to sort a small subset of elements in a big container.

The STL implementers chose a faster `O(N·log(k))` algorithm that performs well for this typical use-case at the expense of other scenarios.

Despite the fact that the STL is generic, some choices have to be made.

In this case, algorithms are fine-tuned for their typical use cases.

The STL implementers knew what they were doing!

And now we do too!

A Little Order!

Delving into the STL sorting algorithms

Fred Tingaud

@FredTingaudDev