Performance Choices

Dietmar Kühl
Bloomberg LP
ACCU 2014

Overview

- motivate a bit why it is worth caring
- comparisons of a couple of examples
- what’s not, yet (?), done

Performance Matters

- there is no program which runs too fast
- you’ll never know how components are used
- more can be achieved with the same resources
- less energy being consumed is good
    - whether it’s batteries, desktops, data centers

Premature Optimisation

- the focus is on small choices having an impact
- choice of data structure is not premature
  - often hard to change later
- goal is not to have excuses for distorted code
  - … or spent a lot of time optimising things

Problem with Measures

- hard to measure actual applications
- not using actual applications distorts the results:
  - removes effects of caches
  - branch prediction is more likely to work
  - used data may have different properties

Micro Benchmarks

- focus on a specific aspect
- assume that there are no memory effects
- profile what never happens in the wild
- can still be instructive
- should not be overrated

Repeating Benchmarks

- results on different systems can be different
  - differences in CPUs
  - compiler and its options
  - operating system
- ⇒ benchmarks need to be repeatable/repeated

Evolving Benchmarks

- benchmarks may measure something different
- it is needed to improve/extend them
- the intend is to inform choices
  - the closer to the real application the better
  - benchmarks may become more specific

C++ vs. Optimisations

- compiling without optimisation is a bad idea!
  - optimisers remove many constructs
  - these things can be layered
- compilers may optimise by default (e.g. icc)
- however, optimisations do affect ability to debug

Used Compilers

- those I have readily available:
  - a recent version of the gcc branch
  - a recent version of the clang branch
  - the latest version of Intel’s compiler
- result from my Intel-based notebook

How to Write a Loop

- simple loop:
```
for (T* it = begin, end = it + size; it != end; ++it)
    rc += *it;
```
  - use indices or pointers?
  - use < or !=
  - go from from to back or from back to front
  - use pre/post increment/decrement

Iteration Oddballs

- std::accumulate(begin, end, T(0));

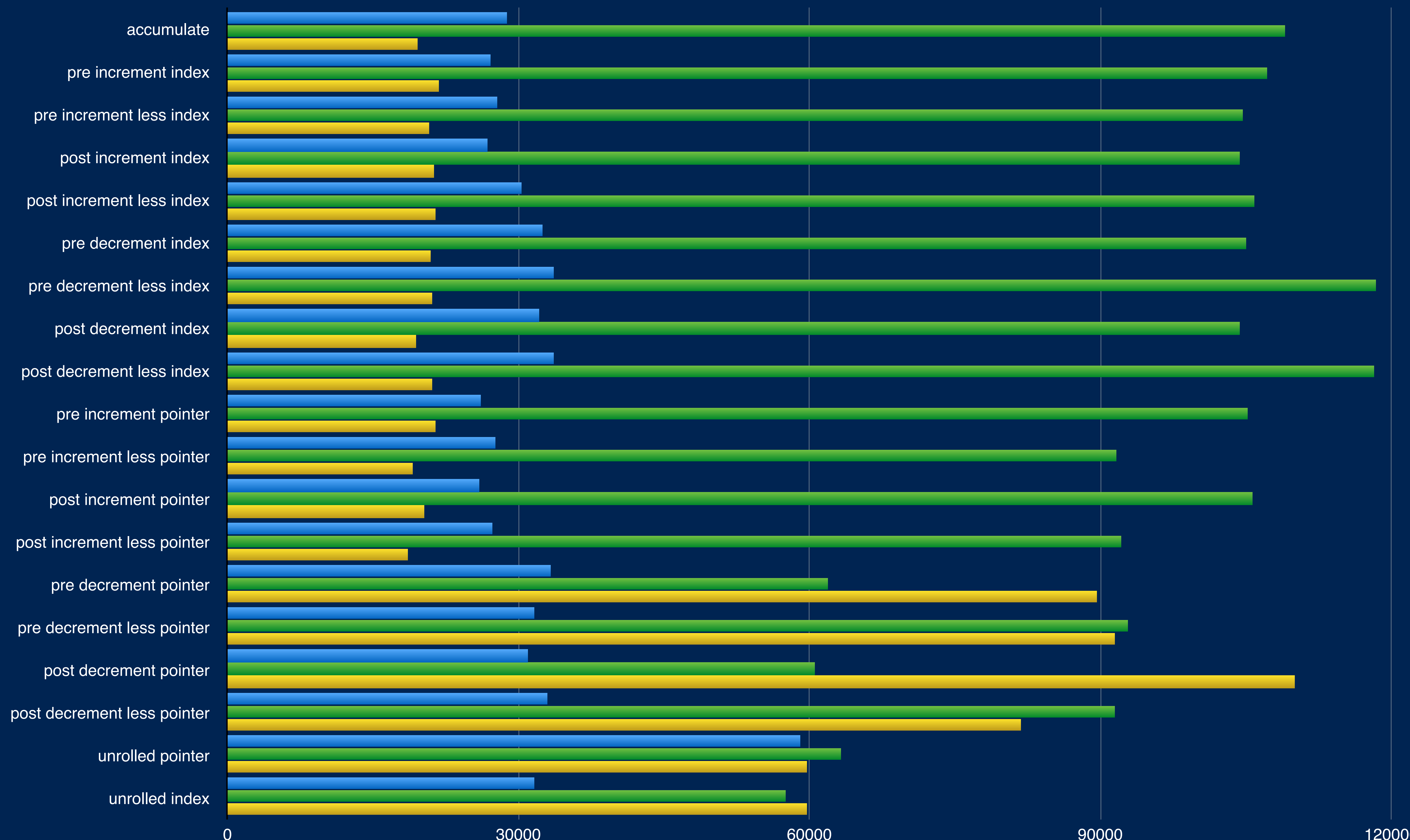
- for (auto&& value: range(begin, end))
    result += value;

- for (; 4 <= end - it; it += 4)
    result += it[0] + it[1] + it[2] + it[3];
  for (; it != end; ++it)
    result += *it;

Iteration Results

	gcc	clang	intel
accumulate	266699	1068923	195670
range for	280863	1044852	193191
pre increment index	261703	1052151	203710
pre increment less index	267635	1048948	203561
post increment index	262833	1046289	202888
post increment less index	263438	1042100	212036
pre decrement index	308393	1040370	202841
pre decrement less index	327695	1171479	206200
post decrement index	311160	1038735	204801
post decrement less index	325170	1177170	186879
pre increment pointer	261964	1046492	190310
pre increment less pointer	278903	910828	209799
post increment pointer	260388	1035324	199766
post increment less pointer	279638	907193	213683
pre decrement pointer	326372	607429	949547
pre decrement less pointer	311962	935578	928107
post decrement pointer	310611	612302	1122938
post decrement less pointer	327901	926136	829393
unrolled pointer	584705	622339	600723
unrolled index	320379	574113	603819

Iteration Results



Iteration Discussion

- gcc and Intel implement vectorisation
- what's good for some compiler is bad for others
  - compiler can't determine what's going on
- ideally, std::accumulate() would be the best
  - sadly, that isn't the case

Different Sequences

- what container type to use?
- containers have different properties
  - different stability guarantees
  - choice influences how containers can be used
  - memory structures tend to be quite different

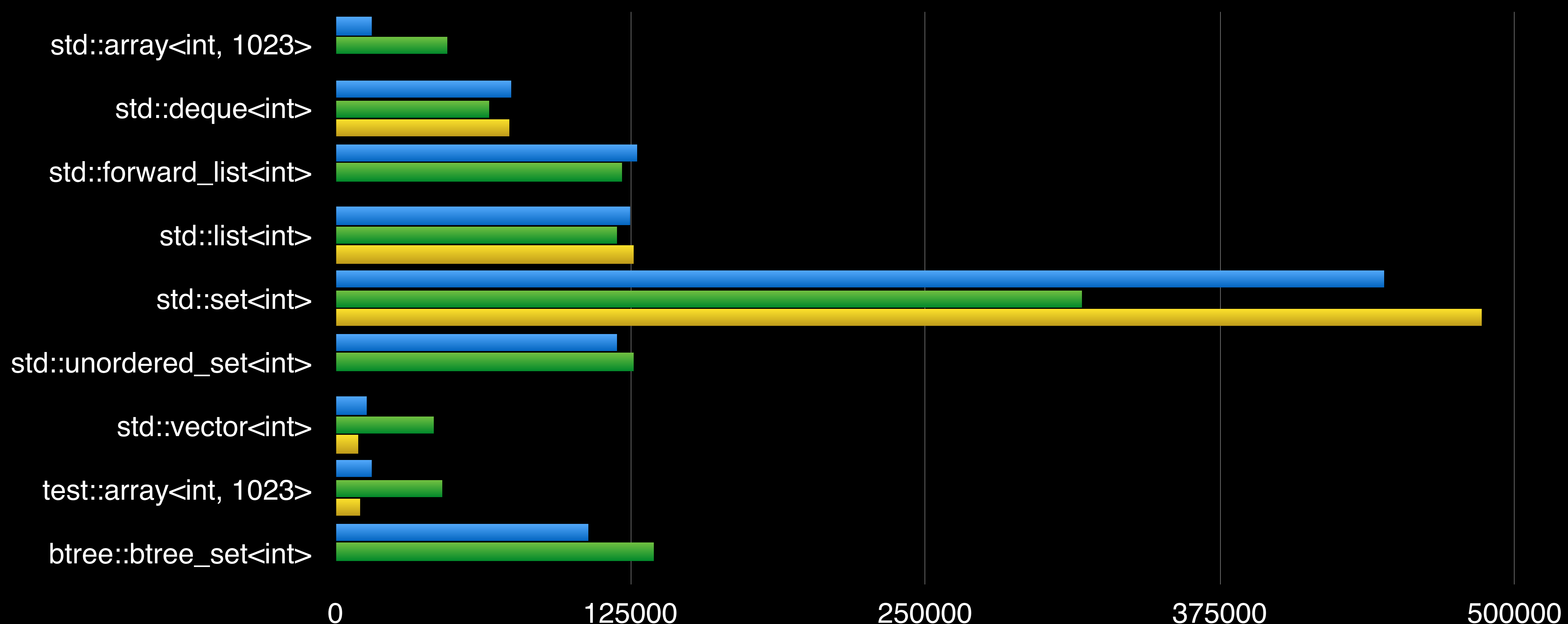
Iterate Over Sequence

- std::accumulate(begin, end, 0)
- for various containers:
  std::array<T, 1023>
  std::deque<T>
  std::forward_list<T>
  std::list<T>
  std::set<T>
  std::unordered_set<T>
  std::vector<T>
  test::array<T, 1023> (custom, inline iterators)
  btree::btree_set<T>

Accumulate int

	gcc	clang	Intel
std::array<int, 1023>	15270	47248	
std::deque<int>	74545	65098	74023
std::forward_list<int>	127617	121489	
std::list<int>	124806	119030	126538
std::set<int>	444655	316318	486050
std::unordered_set<int>	119051	126472	
std::vector<int>	13310	41536	9664
test::array<int, 1023>	15161	45587	10517
btree::btree_set<int>	106891	135088	

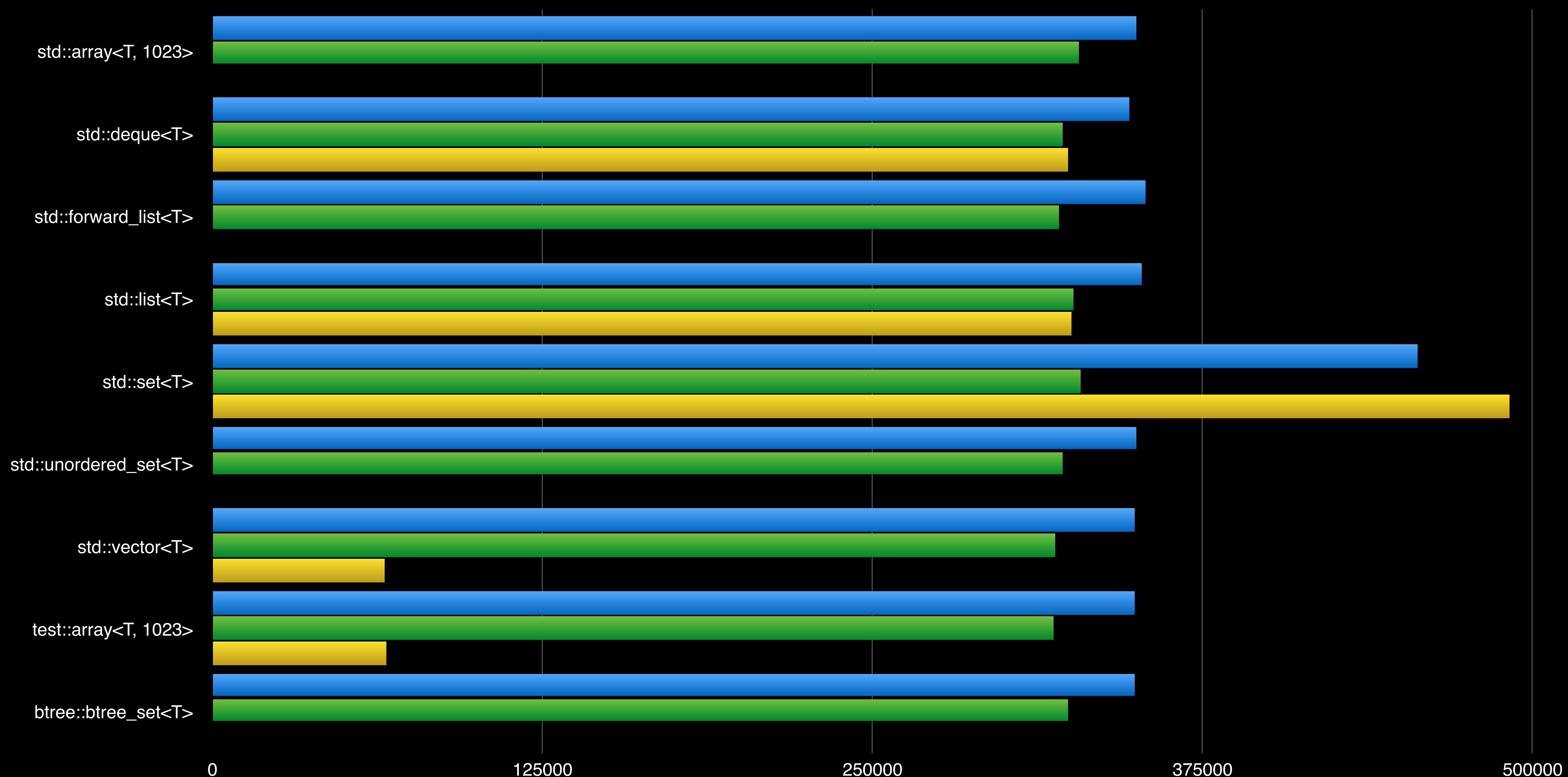
Accumulate int



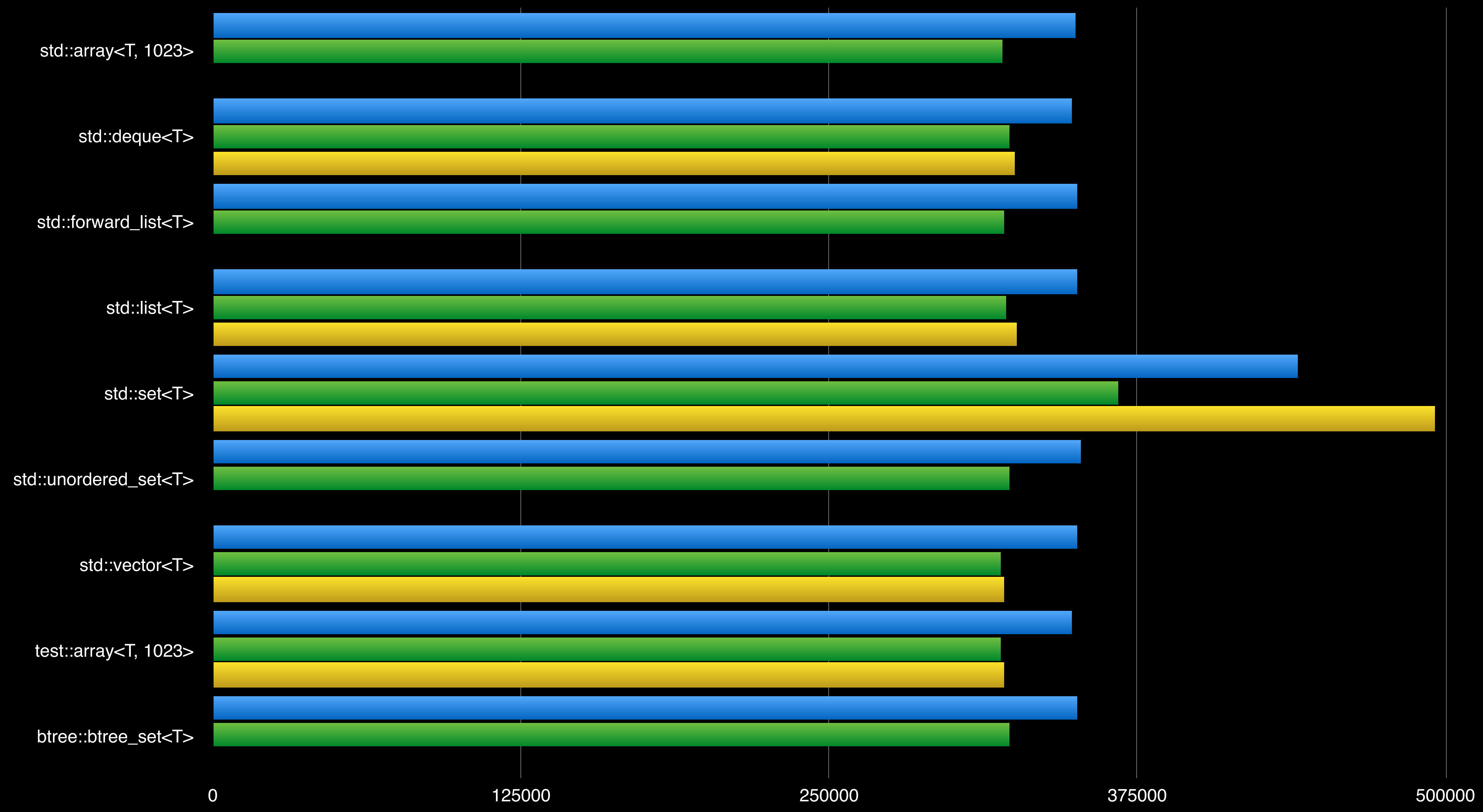
Accumulate Discussion

- if you want to iterate a sequence use vector!
- even if vectorisation can’t be used
- compiler can see through trivial iterators

Accumulate float



Accumulate double



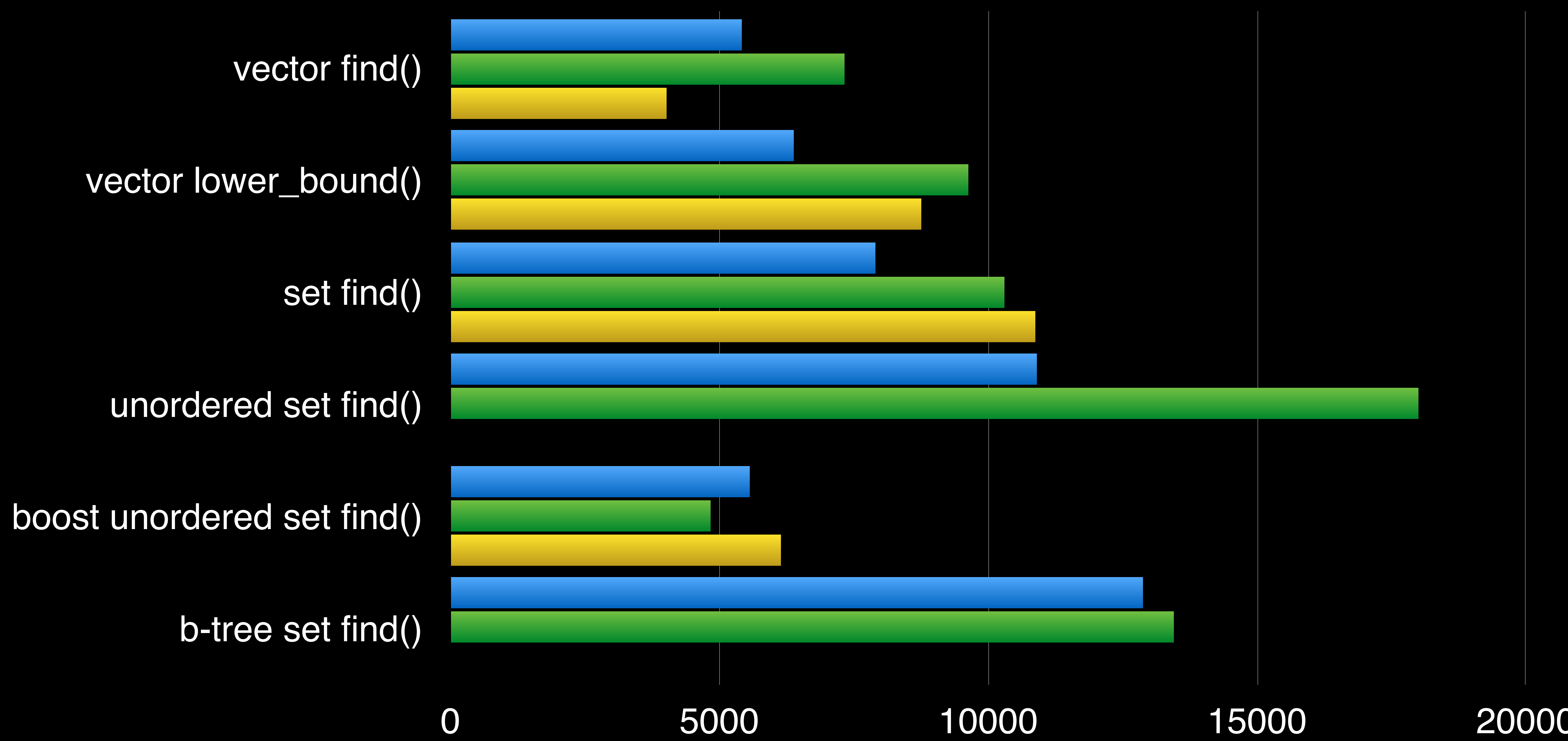
Searching a Value

- operation: find if value needs to be processed: value $\in$ set
- look-up in an immutable set
  - see which data structures are reasonable
  - mutable sets may need a different approach
- use different sizes of sets and different types

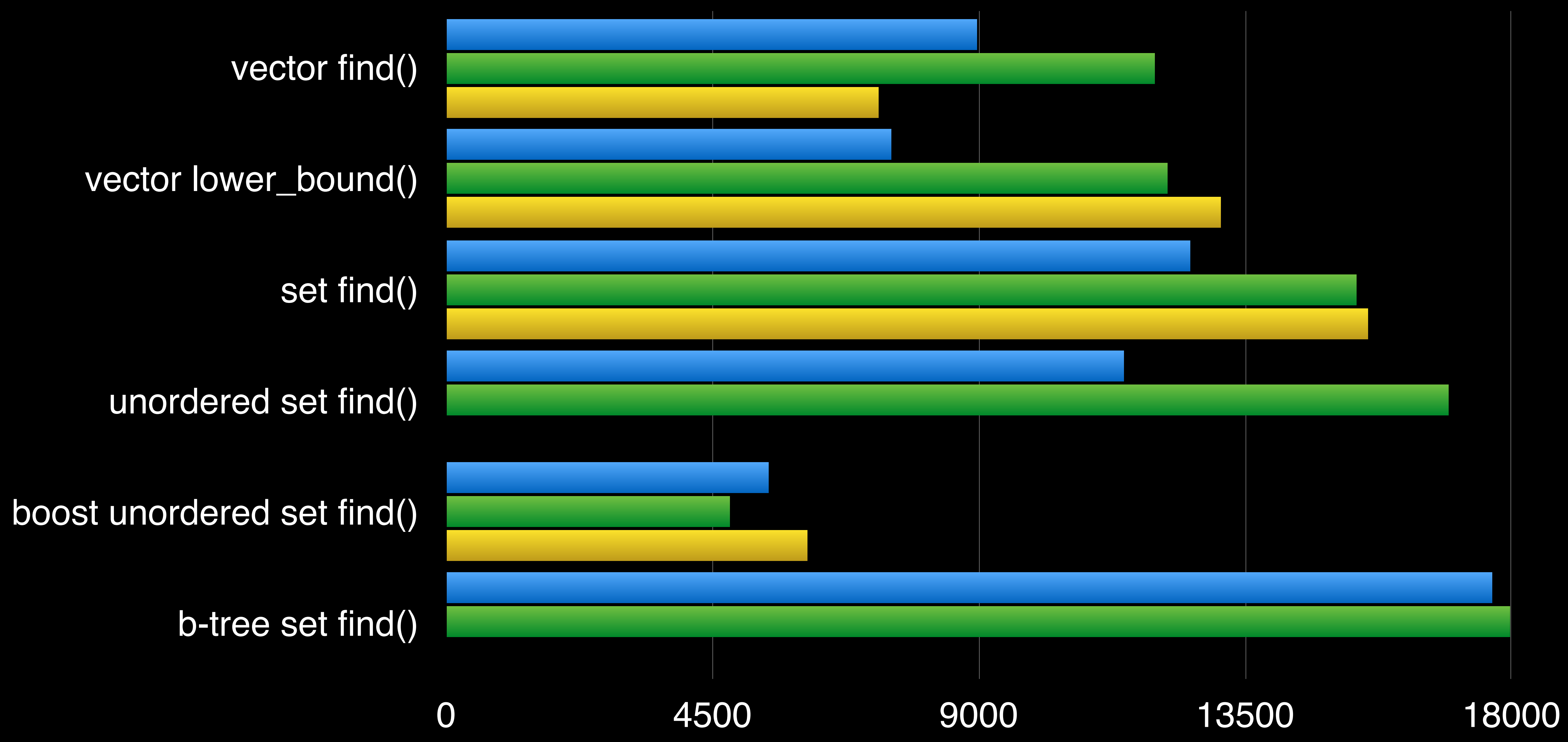
Search Options

- std::vector<T> and std::find(): linear search
- std::vector<T> and std::lower_bound()
- std::set<T>::find()
- std::unordered_mapt<T>::find()
- boost::unordered_map<T>::find()
- btree::btree_set<T>::find

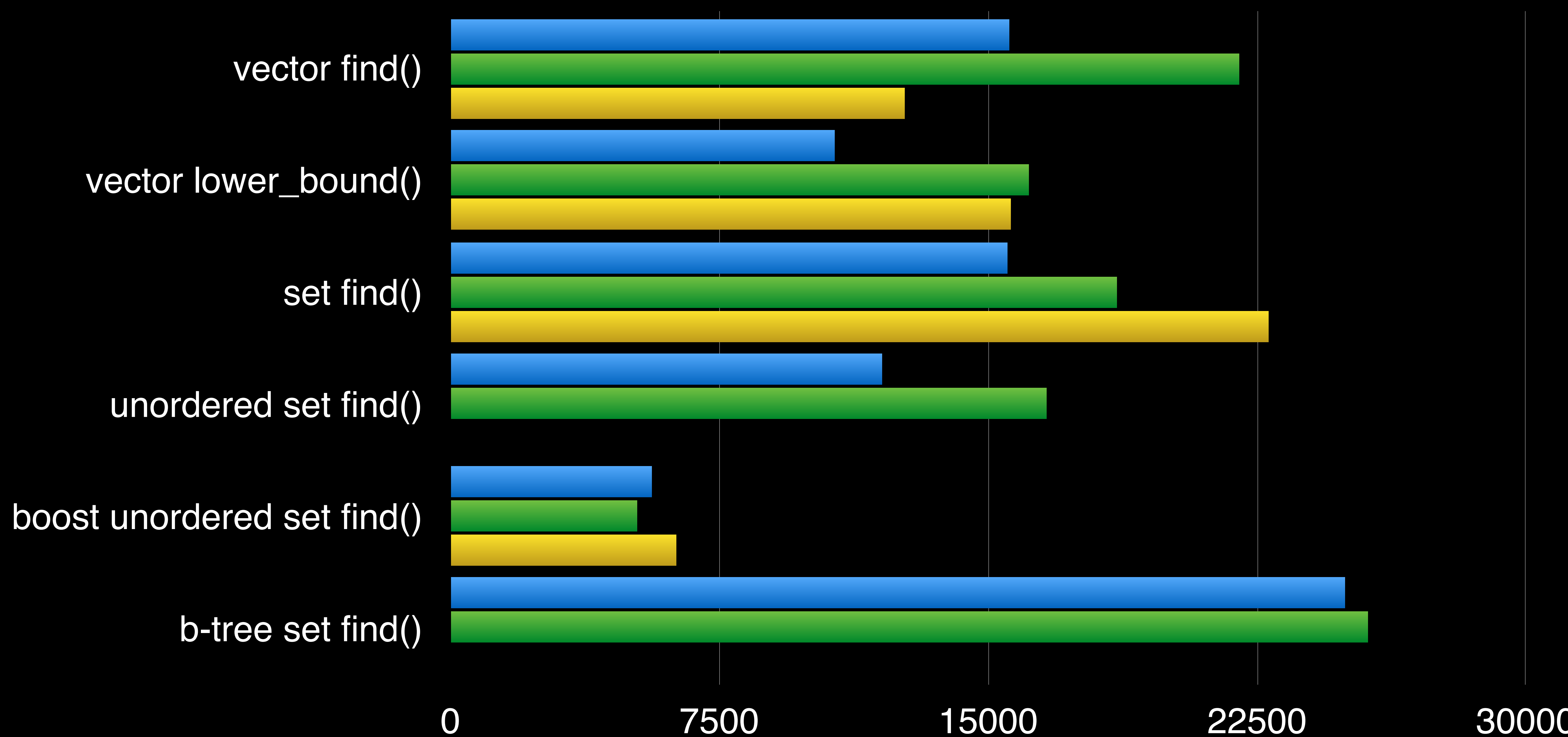
Search<int> 10



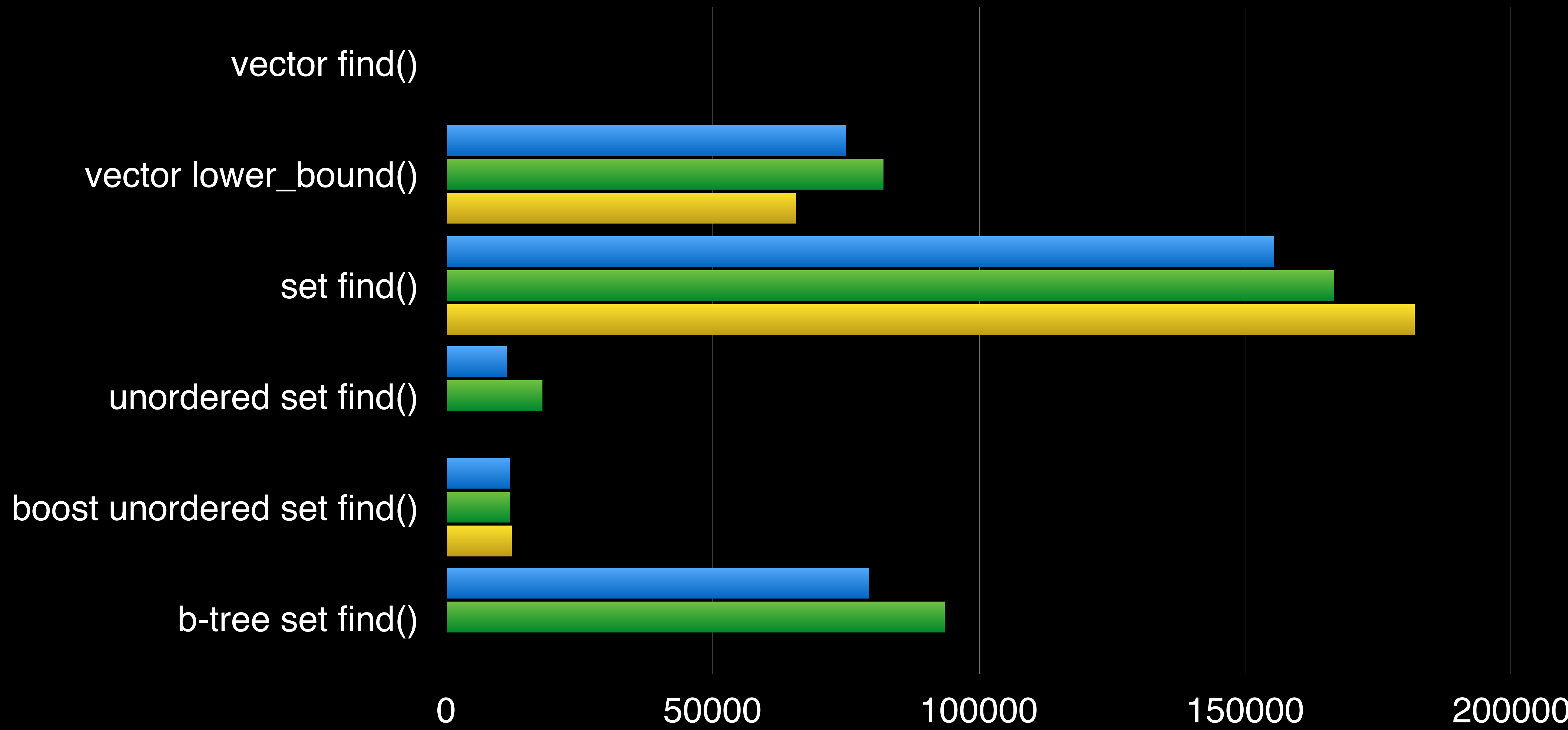
Search<int> 20



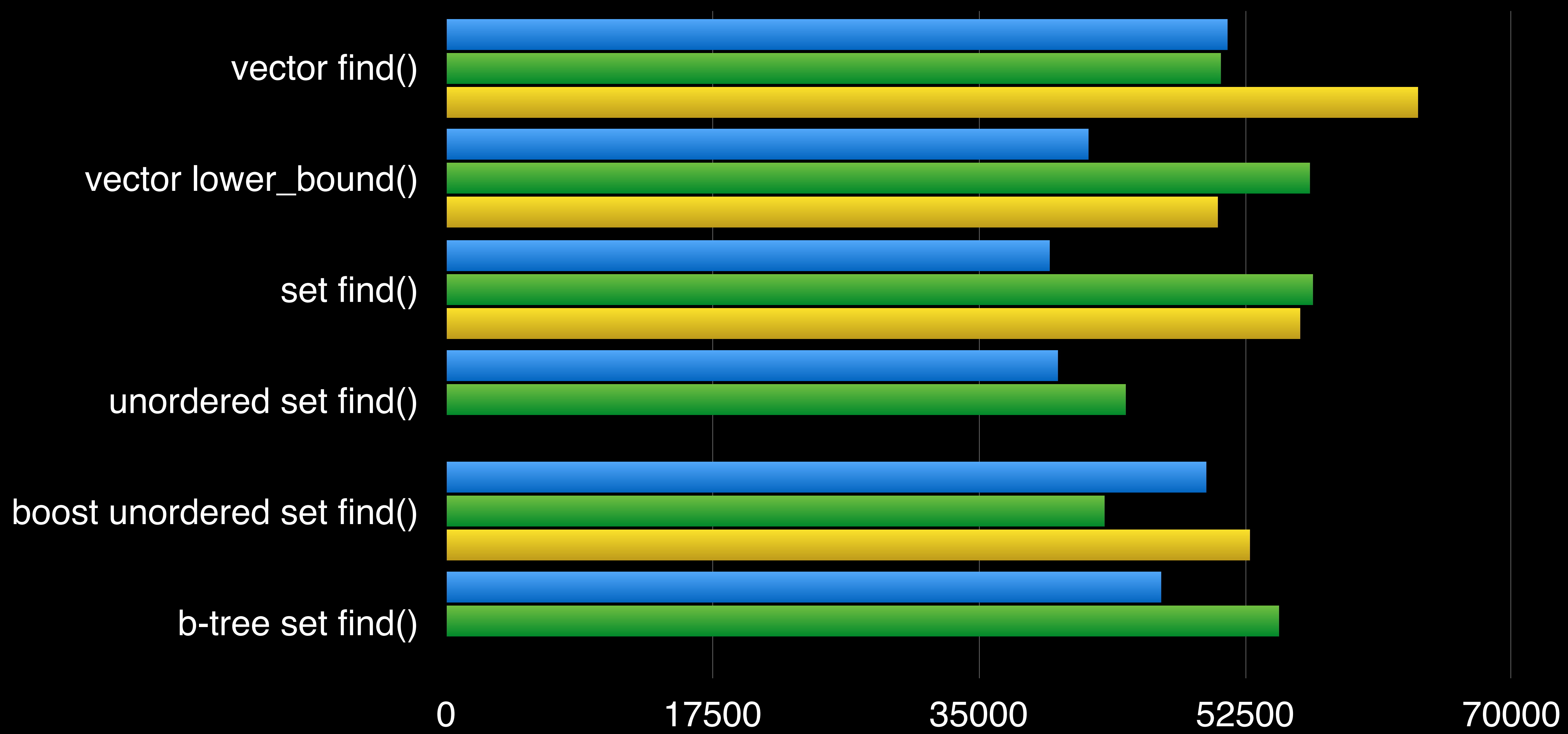
Search<int> 40



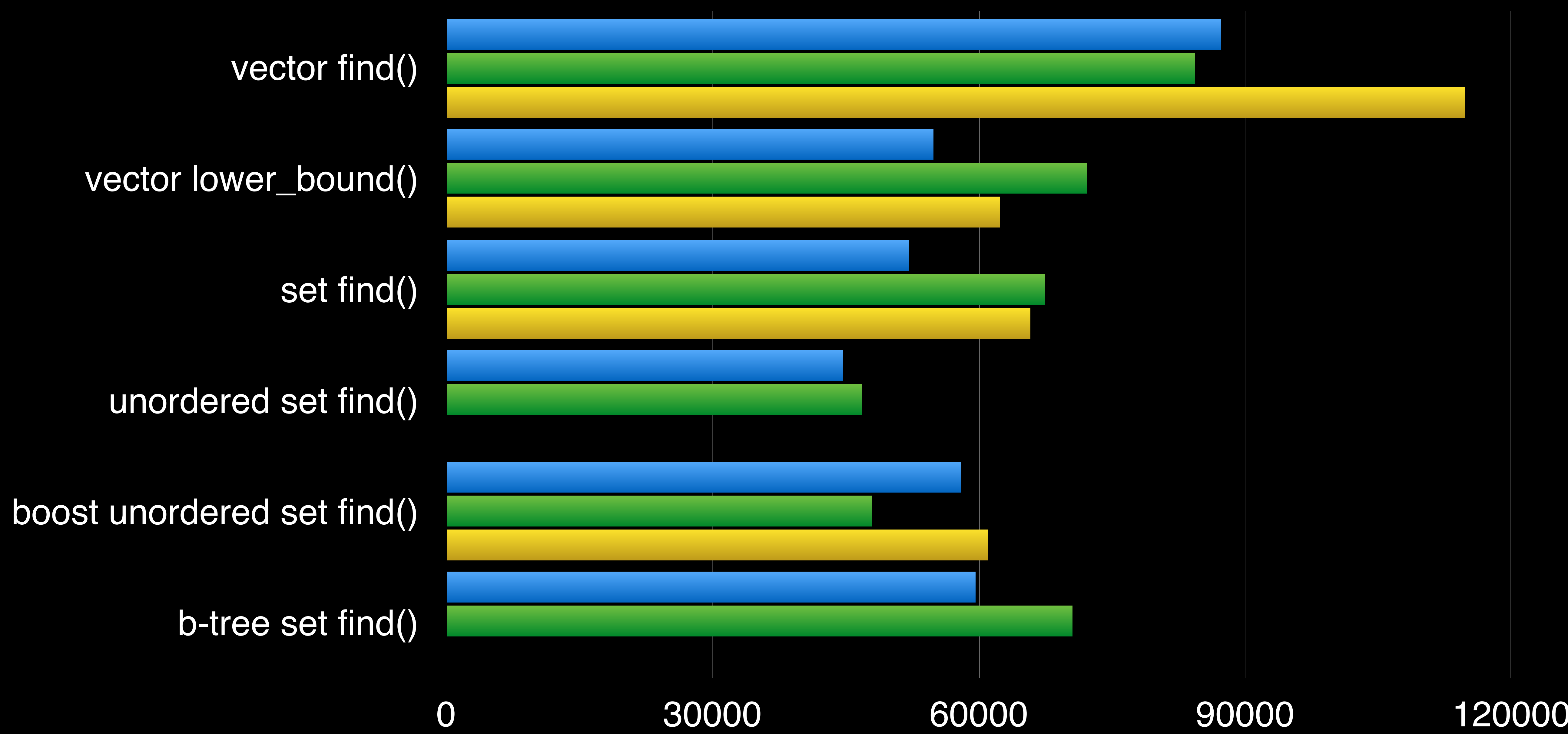
Search<int> 80,000



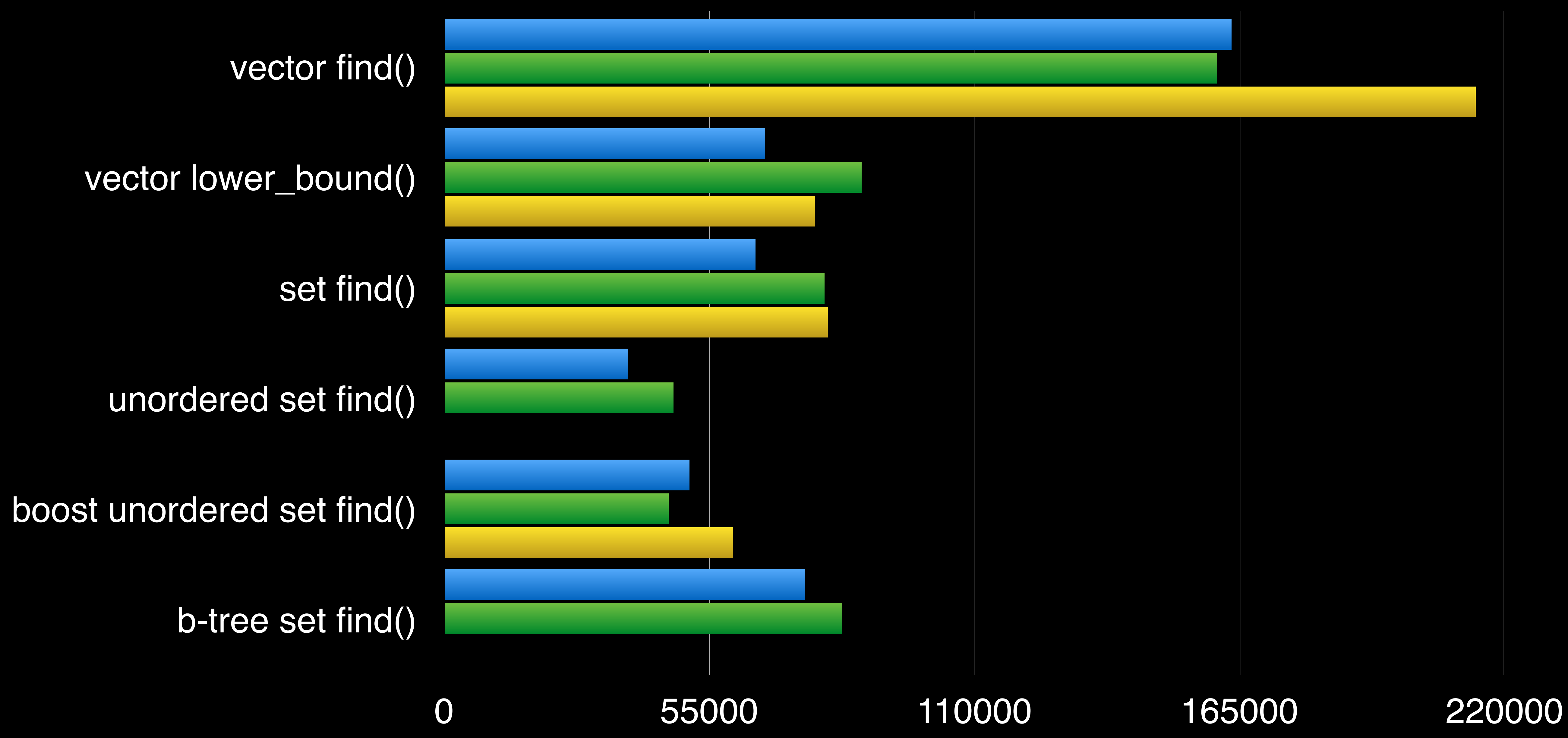
Search<ShortString> 10



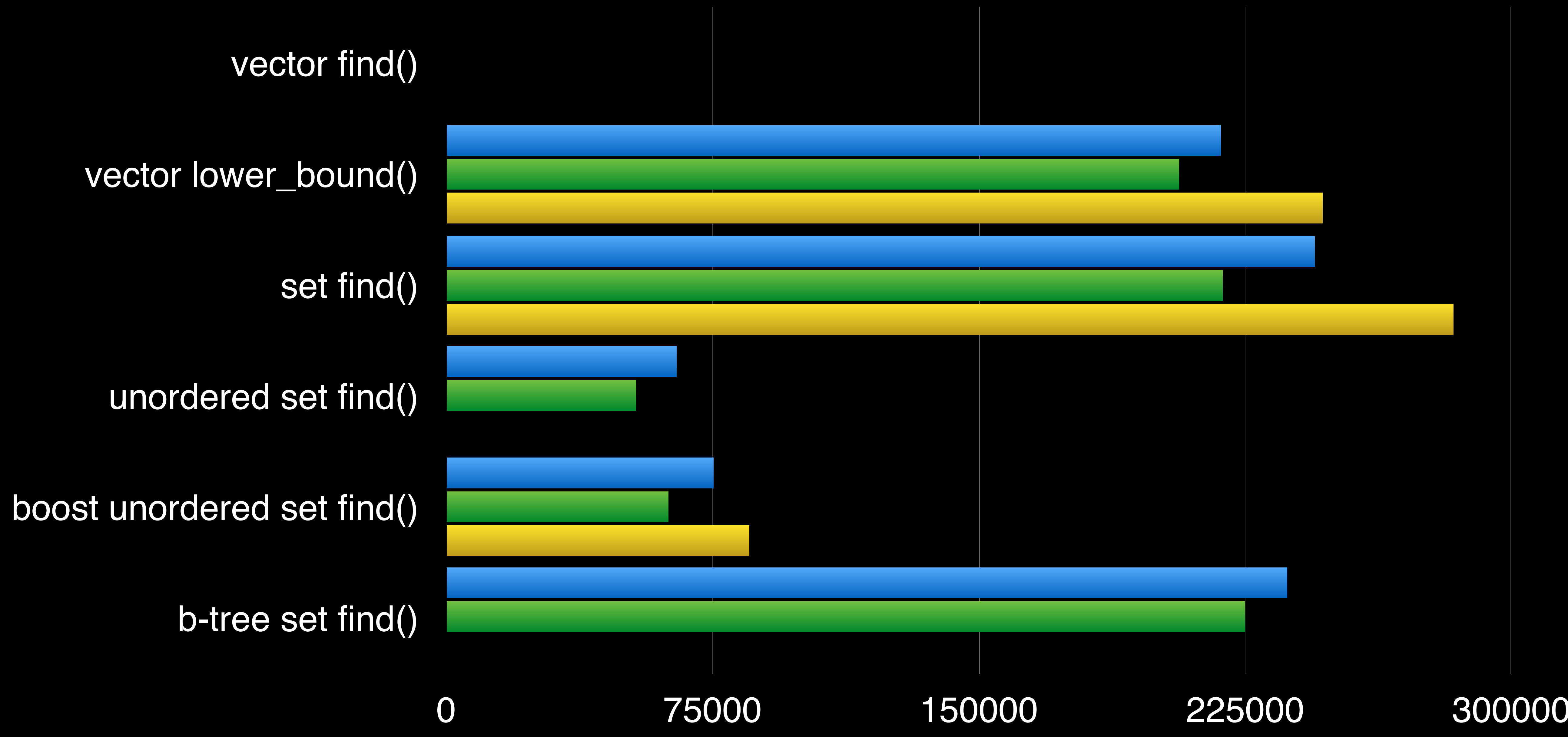
Search<ShortString> 20



Search<ShortString> 40



Search<ShortString> 8000



Searching Summary

- unordered sets are the way to go for large sets
  - beware different costs
  - depends on good and fast hash functions
- order is needed: consider btrees
- other approaches don't really seem to pay off

Custom Function Object

- how to pass a function object?

```
template <typename InIt, typename Pred>
std::size_t count(InIt it, InIt end, Pred pred) {
    std::size_t rc{};
    for (; it != end; ++it) {
        rc += pred(*it);
    }
    return rc;
}
```

Function Object Variations

- global function (i.e., a pointer): normal, inline
  bool isalnum(uc c); count(b, e, islanum);
  inline bool isalnum(uc c); count(b, e, &islanum);

- [bound] member function: normal, inline, virtual
  struct s { bool isalnum(uc c) const; };
  count(b, e, bind(&s::isalnum, s(), _1));

- function call operator: normal, inline, virtual
  struct isalnum { bool operator()(uc c) const; };
  count(b, e, isalnum());

Function Object Variations

- lambda function, pointer to lambda function

  ```
  count(b, e, [](uc c){ … });
  count(b, e, &[](uc c){ … });
  ```

- std::ref(object): normal, inline, virtual

  ```
  struct isalnum { bool operator(uc c) const; };
  count(b, e, std::ref(isalnum()));
  ```

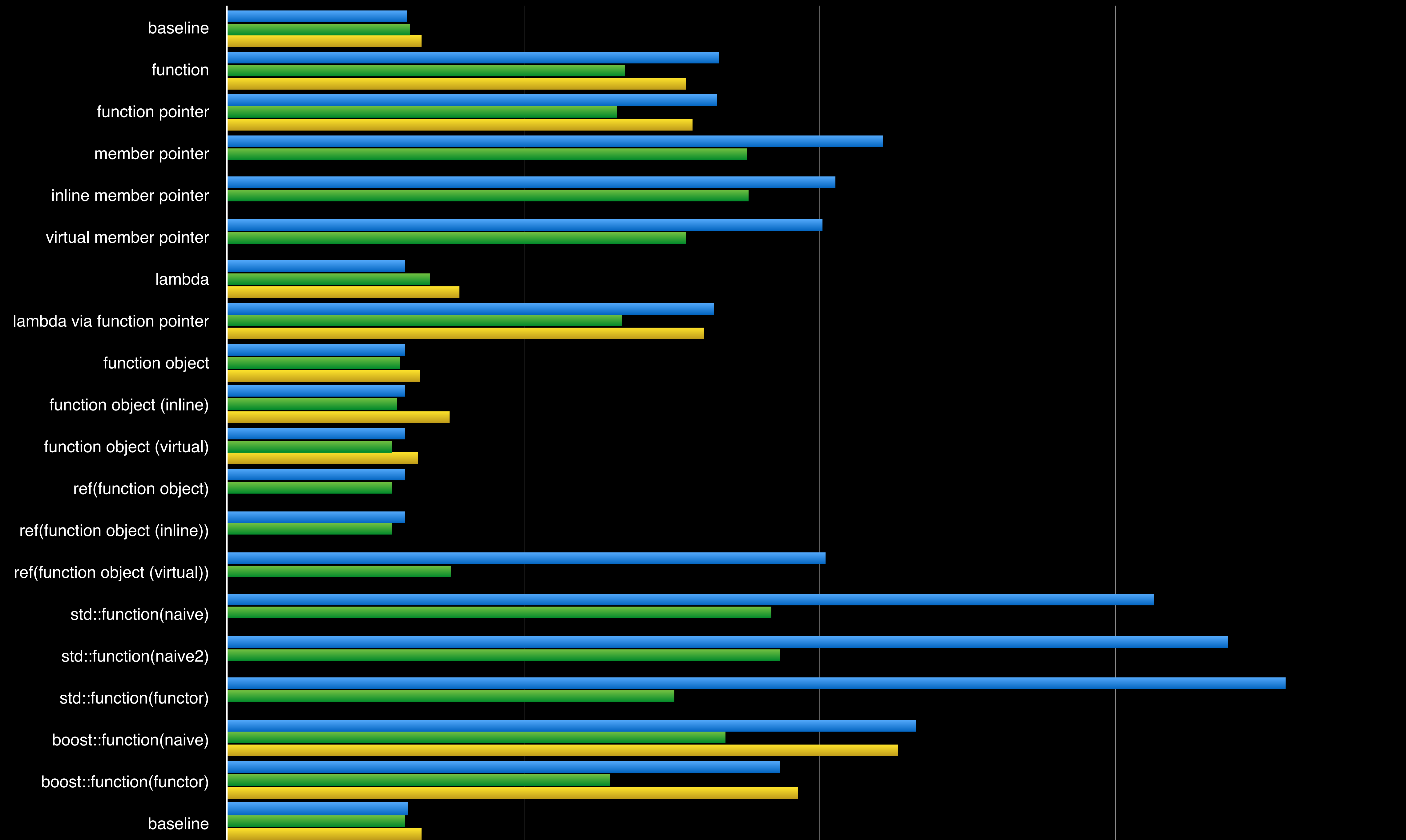
- std::function or boost::function

  ```
  std::function<bool(uc)>(&isalnum)
  std::function<bool(uc)>(isalnum());
  ```

Functions Results

	gcc	clang	intel
baseline	754	772	820
function	2077	1681	1935
function pointer	2069	1646	1962
member pointer	2770	2191	
inline member pointer	2565	2202	
virtual member pointer	2513	1935	
lambda	751	852	977
lambda via function pointer	2057	1662	2013
function object	752	730	814
function object (inline)	751	716	935
function object (virtual)	752	693	806
ref(function object)	751	694	
ref(function object (inline))	753	693	
ref(function object (virtual))	2526	946	
std::function(naive)	3912	2293	
std::function(naive2)	4224	2330	
std::function(functor)	4470	1887	
boost::function(naive)	2905	2103	2831
boost::function(functor)	2329	1616	2404
baseline	765	752	817

Functions Results



Passing Function Objects

- avoid function pointers:
```
std::transform(begin, end, begin, &toupper);
```
- prefer something the compiler can inline:
```
std::transform(begin, end, begin,
    [](unsigned char c){ return toupper(c); });
```
- use std::function<…> only when necessary

Everything a Template?

- passing function object requires templates
- making everything a template isn't viable
    - coupling between components is too tight
    - not viable for dynamic polymorphism
- potentially use a hybrid approach

Outline: Hybrid Approach

```
struct Processor {
    template <typename It, typename Function>
    void process(It b, It e, Function fun) {
        std::vector<T> tmp;
        std::transform(b, e, std::back_inserter(tmp), fun);
        process(tmp);
    }
    virtual void process(std::vector<T>& range);
};
```

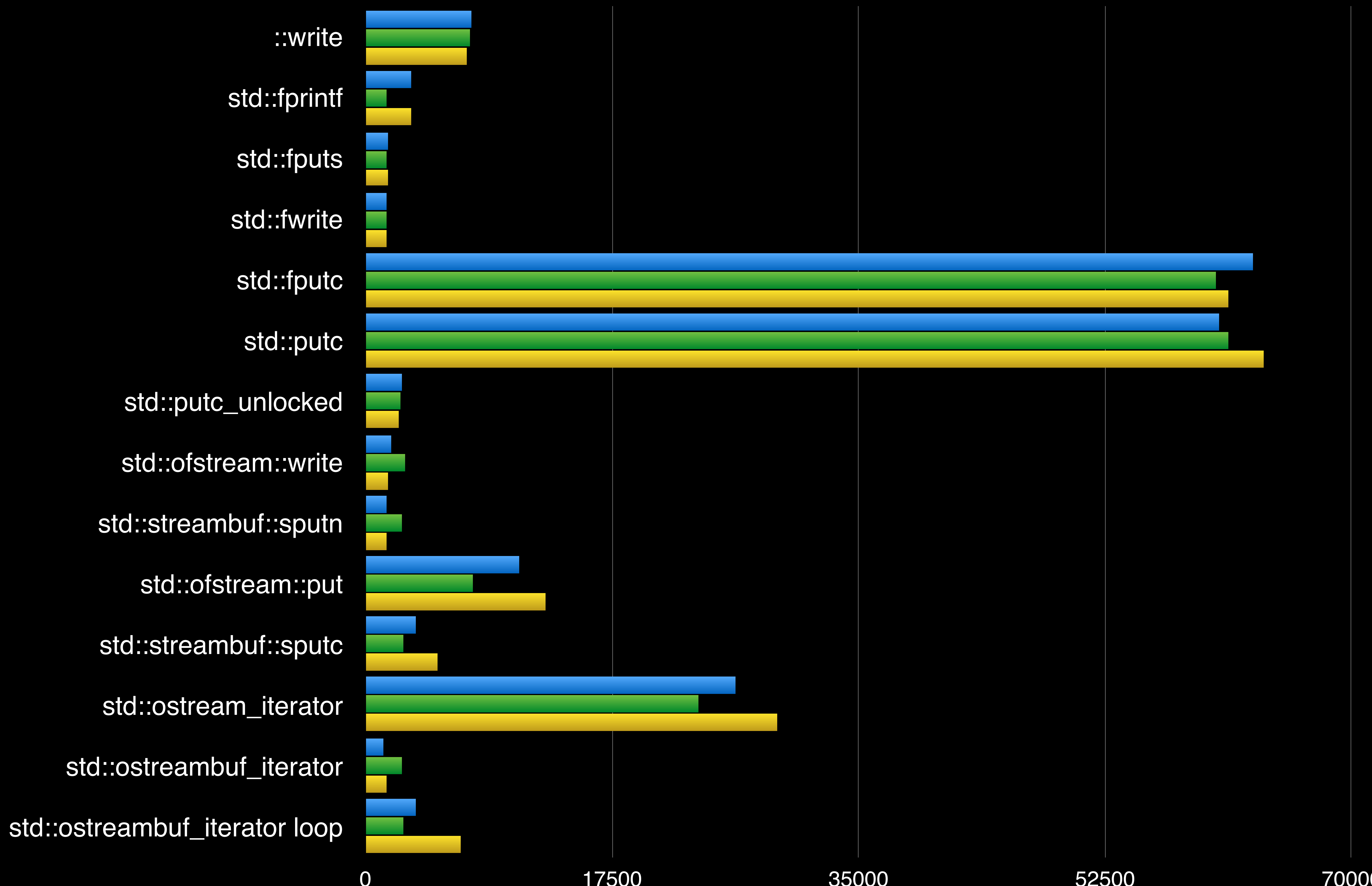
Writing Characters

- three APIs: system, C, C++
  - write()
  - putc(), fputc(), fputs(), fprintf(), fwrite()
  - std::ostream::write(), std::ostream::put()
    std::streambuf::sputn(), std::streambuf::sputc()
    copy(…std::ostream_iterator<char>(stream))
    copy(…std::ostreambuf_iterator<char>(stream))

Write Characters Results

	gcc	clang	Intel
::write	7474	7435	7172
std::fprintf	3276	1512	3262
std::fputs	1595	1478	1553
std::fwrite	1507	1501	1531
std::fputc	63017	60323	61238
std::putc	60619	61203	63798
std::putc_unlocked	2631	2442	2359
std::ofstream::write	1797	2784	1623
std::streambuf::sputn	1521	2624	1469
std::ofstream::put	10916	7607	12827
std::streambuf::sputc	3572	2732	5143
std::ostream_iterator	26234	23585	29274
std::ostreambuf_iterator	1278	2572	1502
std::ostreambuf_iterator loop	3569	2705	6709

Write Characters Results



Format Integers

- std::copy(b, e, std::ostream_iterator<int>(s, “ “));

- for (auto v: values) s << value << ‘ ‘;

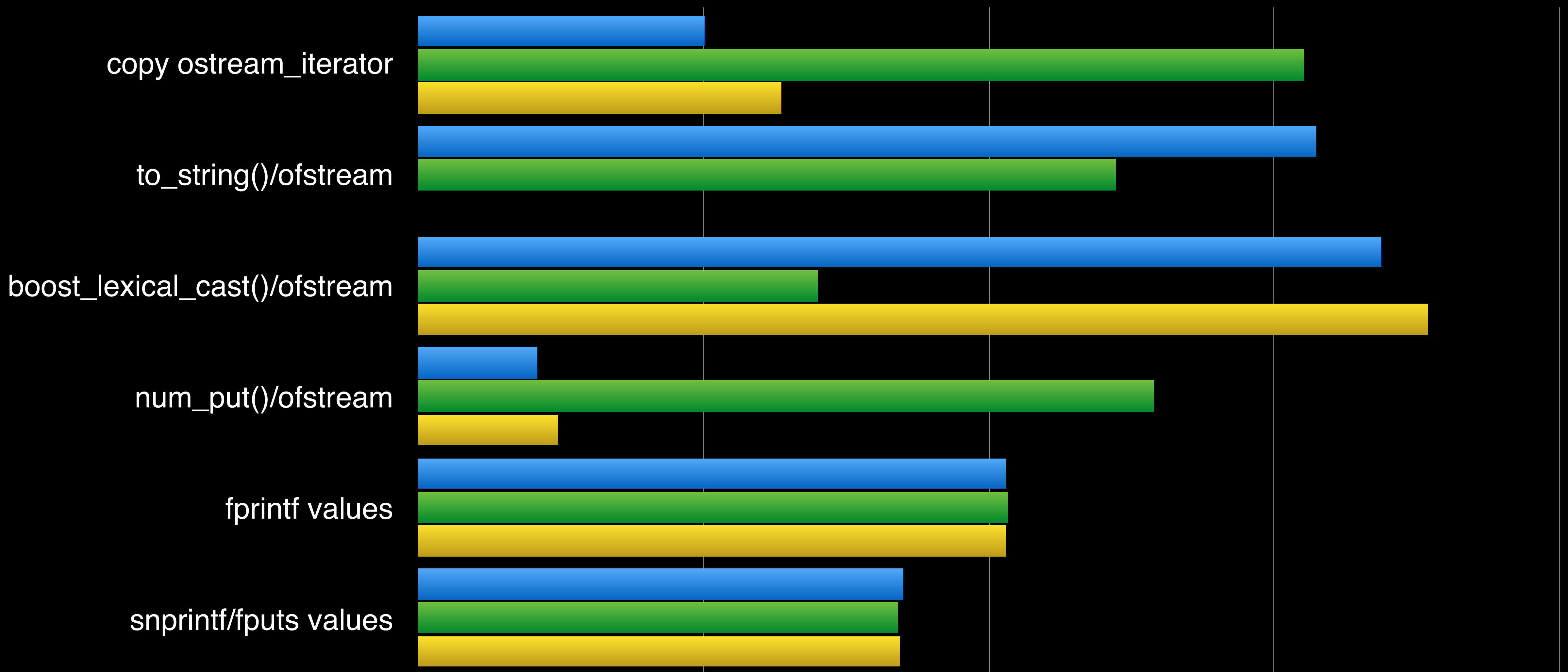
- char b[size], *to = b;
  std::nun_put<char, char*> const& np = …;
  for (auto v: values) to = np.put(to, s, ‘ ‘, v);
  copy(b, to, std::ostreambuf_iterator<char>(s);

- for (auto v: values) fprintf(file, “%d “, v);

Format Integer Results

	gcc	clang	Intel
copy ostream_iterator	751828	2328739	953140
to_string()/ofstream	2361687	1835460	
boost_lexical_cast()/ofstream	2530195	1052444	2656592
num_put()/ofstream	312799	1934857	367982
fprintf values	1545029	1551493	1546697
snprintf/fputs values	1275754	1260641	1266062

Format Integer Results



Using std::num_put

```
typedef std::num_put<char, char*> NP;
std::locale::global(std::locale(std::locale(),
                new std::num_put<char, char*>()));

char buffer[1000 * 12], *end(buffer);
NP const& np(std::use_facet<NP>(s.getloc()));
for (long x: values) {
    end = np.put(end, this->d_out, ' ', x);
    *end++ = ' ';
}
```

More Things to Do

- extend beyond this random and small example
- run on different system, compiler, hardware, etc.
- create more realistic data sets of different sizes (put differently: get a bit more scientific)
- measure better resembling real applications

Collect Measurements

The Plan

- run on arbitrary system
- record data about system characteristics
- send measurements to a common repository
- organise and represent the data

The Source

- http://github.com/dietmarkuehl/cputube