Cppcon
The C++ Conference
2021
October 24-29
IN AURORA COLORADO USA AND ONLINE

Algorithmic Complexity

Amir Kirsh & Adam Segoli Schubert

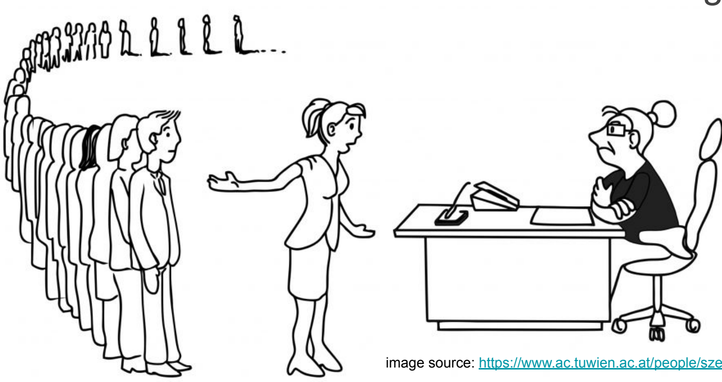


image source: https://www.ac.tuwien.ac.at/people/szeider/cartoon

Why this talk?

Why this talk?

Performance is the name of the game

Why this talk?

Performance is the name of the game

You all (hopefully) know that $O(n)$ is better than $O(n^2)$

Why this talk?

Performance is the name of the game

You all (hopefully) know that $O(n)$ is better than $O(n^2)$

But there is still important stuff that might be overlooked

Why this talk?

Performance is the name of the game

You all (hopefully) know that $O(n)$ is better than $O(n^2)$

But there is still important stuff that might be overlooked

And…

Why this talk?

Performance is the name of the game

You all (hopefully) know that O(n) is better than O(n^2)

But there is still important stuff that might be overlooked

And… the academic answer isn't always the practical answer

About Us

Amir Kirsh

Lecturer
Academic College of Tel-Aviv-Yaffo
and Tel-Aviv University

Developer Advocate
INCREDIBUILD

Co-Organizer of the **CoreCpp** conference and meetup group

Adam Segoli Schubert

Software consultant and programming instructor.

Part of Dev Advocate Office at
INCREDIBUILD

Collaborates and passionate about projects which focus on decentralization, parallelization, with the objective of advancing transparency, freely available distributed knowledge, and autonomy.

Algorithmic Complexity

Performance is the name of the game

Algorithmic Complexity

It’s actually about something BIGGER than just performance

Algorithmic Complexity

What's BIGGER than just performance?

Algorithmic Complexity

What's BIGGER than just performance?

Scalability

Computational Complexity

Computational Complexity or simply **Complexity** of an algorithm is the *amount of **resources*** required to run it.

- from Computational complexity in wikipedia

Resources:
The amount of time, storage, or any other resource.

Computational Complexity

$n \rightarrow f(n)$

n - is size of input

$f(n)$ - is the amount of resources required to run the algorithm

Time - number of required elementary operations

Often denoted by $T(n)$ or $t(n)$

Space - Amount of memory required

Often denoted by $S(n)$ or $s(n)$

Computational Complexity

Worst-Case Complexity

Maximum amount of resources needed over all inputs of size n.

Average-Case Complexity

Average amount of resources over all inputs of size n.

Best-Case Complexity

Minimum amount of resources needed over all inputs of size n.

Big O Notation

We say that:

$$f(n) \in O(g(n)) \qquad [f(n) = O(g(n)) \text{ as } n \to \infty]$$

iff:

$$\exists\, k > 0 \;\exists\, n_0 \;\forall\, n > n_0: \quad f(n) \le k \cdot g(n)$$

Only the dominating factor counts

In Big O, we care about asymptotic analysis, when $n \longrightarrow \infty$

Thus, for example:

$t(n) = k_1 \cdot n \cdot \log(n) + k_2 \cdot n$

Only the dominating factor counts

In Big O, we care about asymptotic analysis, when $n \longrightarrow \infty$

Thus, for example:

$t(n) = k_1 \cdot \mathbf{n \cdot \log(n)} + k_2 \cdot n$

Only the dominating factor counts

In Big O, we care about asymptotic analysis, when $n \longrightarrow \infty$

Thus, for example:

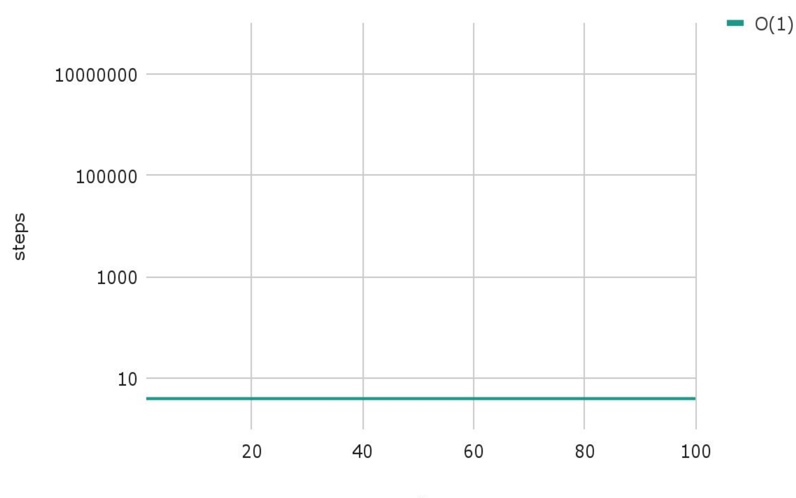
$$t(n) = k_1 \cdot \mathbf{n} \cdot \mathbf{log(n)} + k_2 \cdot n \;\in\; O(\mathbf{n} \cdot \mathbf{log(n)})$$

Not in this talk...

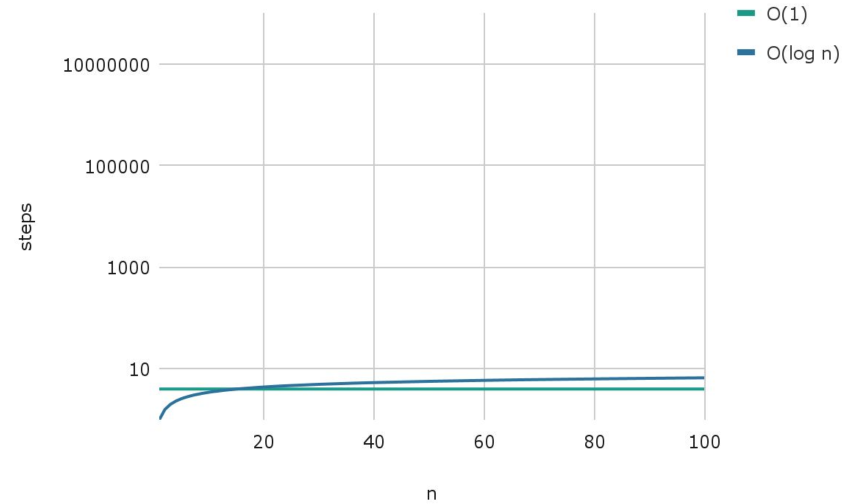
Notation	Name	Short Explanation
$\Theta(g(n))$	Theta	f(n) is bounded both above and below by g(n) asymptotically
$\Omega(g(n))$	Omega	f(n) is bounded below by g(n) asymptotically
$o(g(n))$	Small o	f(n) is dominated by g(n) asymptotically
$\omega(g(n))$	Small omega	f(n) dominates g(n) asymptotically
$\tilde{O}(g(n))$	Tilde O	same as big O, but “ignores” logarithmic factors

Let the Charts talk

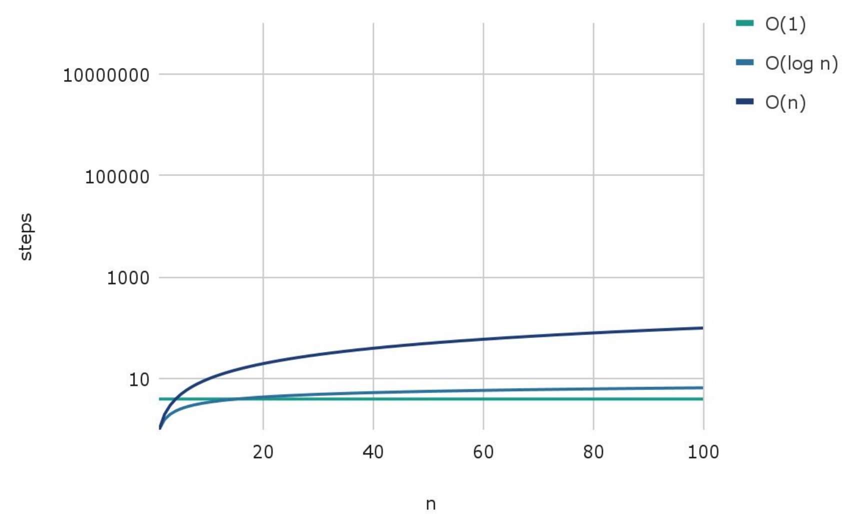
Let the Charts talk



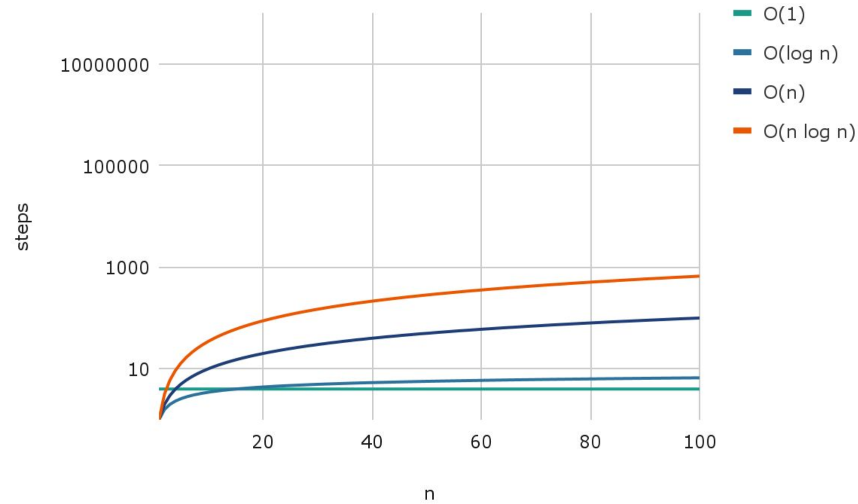
Let the Charts talk



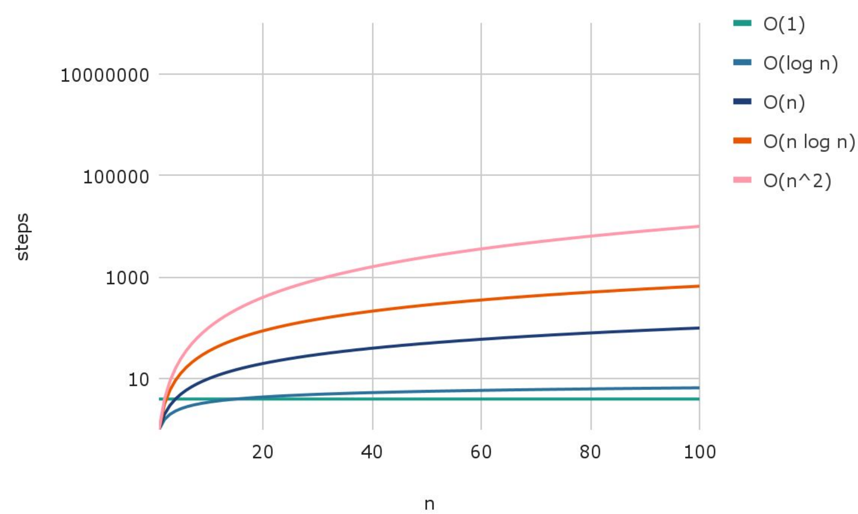
Let the Charts talk



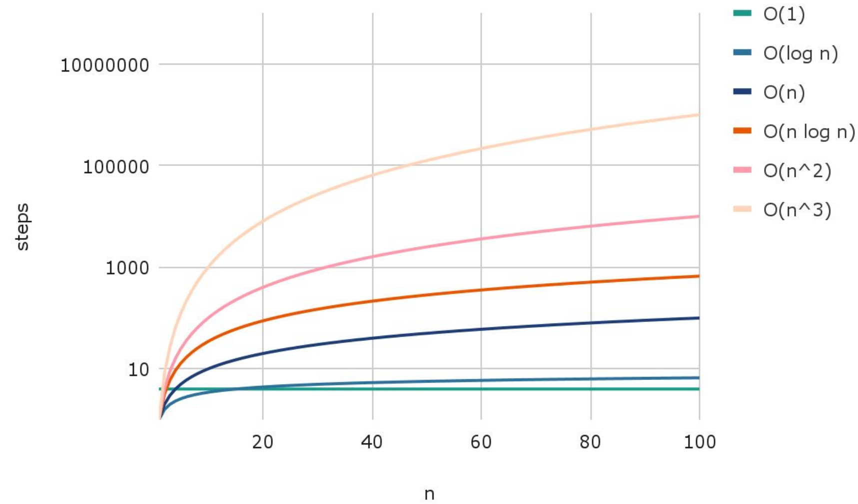
Let the Charts talk



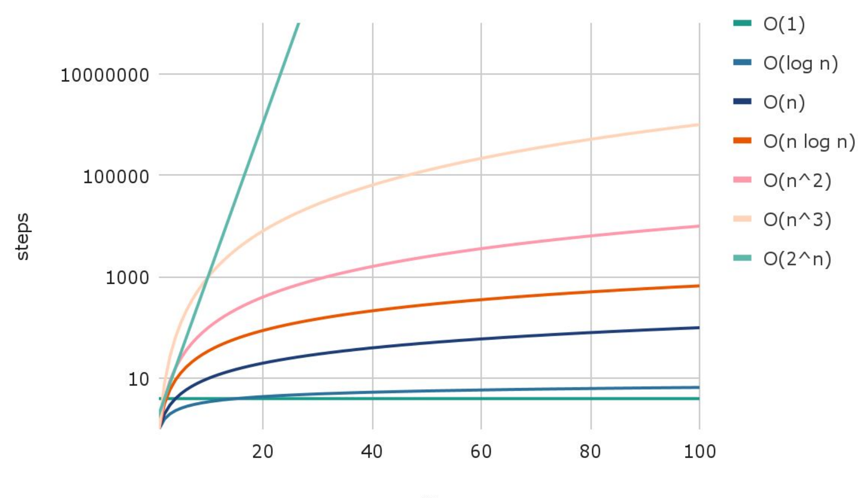
Let the Charts talk



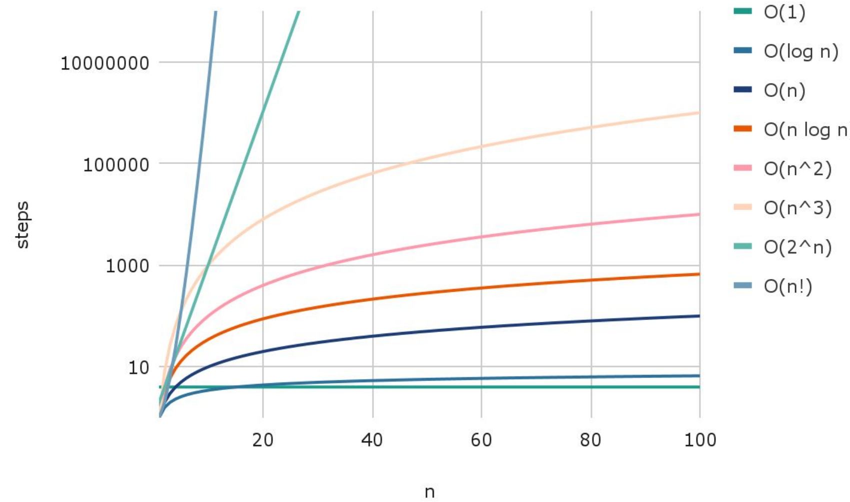
Let the Charts talk



Let the Charts talk



Let the Charts talk



Time Estimates

Time Estimates

n	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(n^3)	O(2^n)	O(n!)
1	1 μs	1 μs	1 μs	1 μs	1 μs	1 μs	2 μs	1 μs
10	1 μs	3 μs	10 μs	34 μs	100 μs	1 ms	1 ms	3.6 seconds
100	1 μs	6 μs	100 μs	665 μs	10 ms	1 sec	>400 trillion centuries	>googol centuries
1,000	1 μs	9 μs	1 ms	~10 ms	1 sec	16.67 min	…	…
10,000	1 μs	13 μs	10 ms	~133 ms	1.67 min	~12 days	…	…
100,000	1 μs	16 μs	100 ms	1.67 sec	2.78 hours	~32 years	…	…
1,000,000	1 μs	19 μs	1 sec	~20 sec	~12 days	~32,000 years	…	…

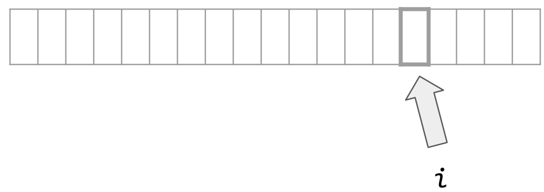
* let's assume our single operation takes 1 μs

Let's see if we got it right

Are you ready for a short quiz?

(1) What is the Complexity of:

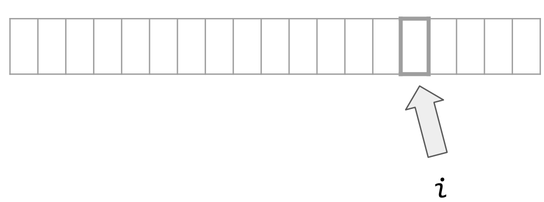
Get an element **in a vector**, at index i



(1) What is the Complexity of:

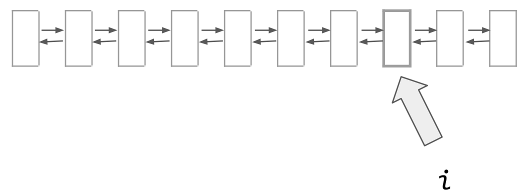
Get an element **in a vector**, at index i

O(1)



(2) What is the Complexity of:

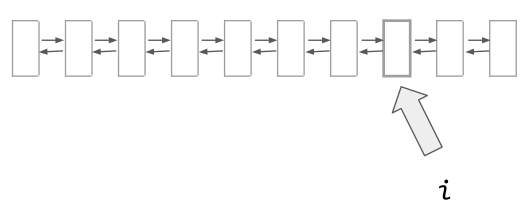
Get an element **in a list**, at position i



(2) What is the Complexity of:

Get an element **in a list**, at position i

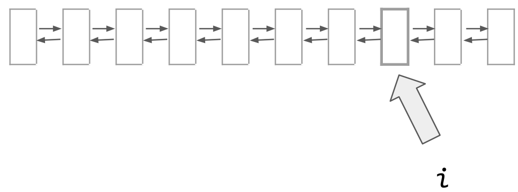
O(n)



(2) What is the Complexity of:

Get an element **in a list**, at position i

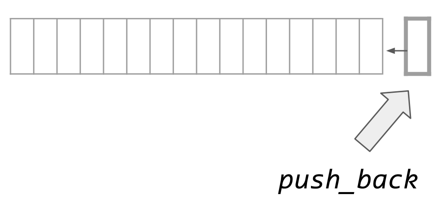
O(n)



See this chart

(3) What is the Complexity of:

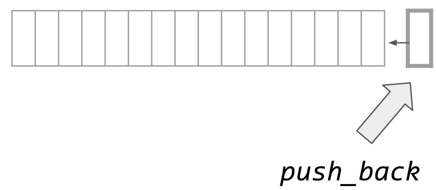
push_back **to a vector**



(3) What is the Complexity of:

push_back **to a vector**

well, we need to talk about
Amortized Complexity



Amortized Complexity

Amortized complexity considers the total worst case complexity of a sequence of operations, instead of just one operation.

Example 1:
If the *total* for *n* operations is in the worst case O(n) then the ***amortized complexity*** is O(1)

Example 2:
If the *total* for *n* operations is in the worst case O(n^2) then the ***amortized complexity*** is O(n)

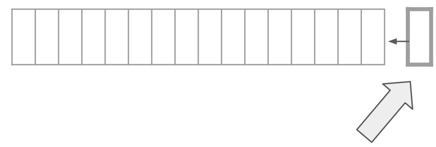
Note:
Amortized complexity is NOT the *average* complexity over different inputs of size n!

See: Tarjan, Robert Endre (April 1985). *Amortized Computational Complexity*

(3) What is the Complexity of:

push_back **to a vector**

Amortized O(1)

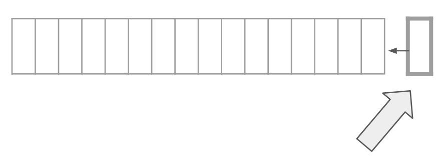


(3) What is the Complexity of:

push_back **to a vector**

Amortized O(1)

How do we know?



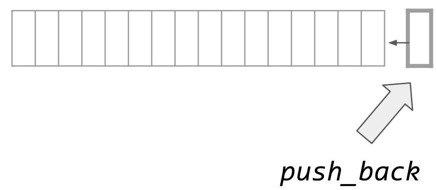
(3) What is the Complexity of:

push_back **to a vector**

Amortized O(1)

How do we know?

Because *the spec* *requires it*!



C++ Specifications - Complexity Requirements

In the spec (*examples*):
containers requirements
unordered associative containers + requirements
complexity of std::sort algorithm
complexity of std::ranges::partition algorithm

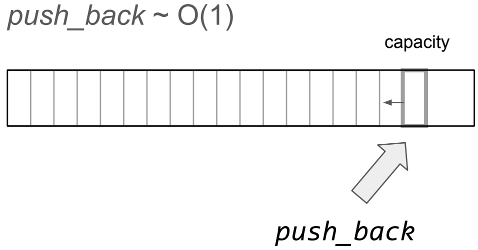
Then in CppReference (*examples*):
complexity of *std::vector::insert*
complexity of *std::list::insert*
complexity of *std::unordered_map::insert*
complexity of *std::search algorithm*
complexity of *std::sort algorithm*

std::vector resizing following push_back

Case A

There is enough capacity

push_back ~ O(1)



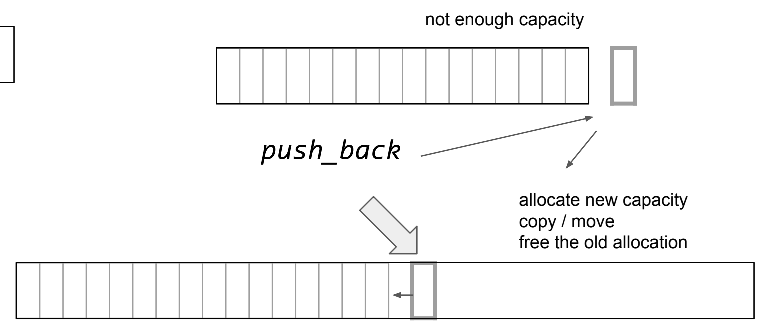
To have: *push_back* ~ amortized O(1):

At most *1* of *n* calls may be of case B

Case B

There isn't enough capacity

Needs to move / copy the vector ~ O(n)



An important side note on vector resizing!

There are 3 options when moving / copying the elements from the old allocation:

(a) **For trivially copyable elements:** vector may use **memcpy**
(b) **If the elements are nothrow_move_constructible:** vector **moves** the elements
(c) Otherwise: the elements are **copied**

=> if you implement your own move make sure it is marked with `noexcept`

```
Widget(Widget&& w) noexcept { /* … */ }
```

See benchmark

Back to our quiz

Back to our quiz

Are you ready?

(4) What is the Complexity of:

Sorting **a vector** using std::sort or std::ranges::sort / **a list** using list::sort

(4) What is the Complexity of:

Sorting **a vector** using std::sort or std::ranges::sort / **a list** using list::sort

O(n log(n))

See the spec [for std::sort] and [for list::sort]

(4) What is the Complexity of:

Sorting **a vector** using std::sort or std::ranges::sort / **a list** using list::sort

O(n log(n))

See the spec [for std::sort]() and [for list::sort]()

Note: it might be more efficient to copy the list into a vector, sort the vector, then copy back
Why?
[See benchmark]()

(5) What is the Complexity of:

Finding the median of *n* items

(5) What is the Complexity of:

Finding the median of *n* items

There is an algorithm, PICK, with O(n) worst case complexity!

However, another algorithm, Quickselect, which is $O(n^2)$ at worst case, is usually faster.

They are both O(n) on average.

See https://cs.stackexchange.com/questions/1914/find-median-of-unsorted-array-in-on-time

(5) What is the Complexity of:

Finding the median of *n* items

There is an algorithm, PICK, with O(n) worst case complexity!

However, another algorithm, Quickselect, which is $O(n^2)$ at worst case, is usually faster.

They are both O(n) on average.

See https://cs.stackexchange.com/questions/1914/find-median-of-unsorted-array-in-on-time
See also spec requirement for std::nth_element

(6) What is the Complexity of:

find / insert - **unordered_map**

(6) What is the Complexity of:

find / insert - **unordered_map**

O(1) average case

O(n) worst case

See [the spec for find]
See [the spec for insert]

(6) What is the Complexity of:

Performing equality (==) between two unordered_maps of the same type

(6) What is the Complexity of:

Performing equality (==) between two unordered_maps of the same type

$O(n)$ average case

$O(n^2)$ worst case

See the spec

Other Examples

Examples: O(1)

std::list::insert (at any position)

std::list::erase (for a single iterator)

std::vector::pop_back

Examples: O(n)

std::find

std::max

std::min

Examples: O(log n)

std::binary_search

std::map::find

std::map::insert

Examples: O(n log n)

std::sort

Examples: O(n^2)

Bubble sort

Examples: $O(n^2)$

Bubble sort

Is there any reason whatsoever for using bubble sort?

Examples: $O(n^2)$

Bubble sort

Is there any reason whatsoever for using bubble sort?

- In space complexity, maybe?
- Being stable? (what is stable sorting algorithm?)

Examples: $O(n^2)$

Bubble sort

Is there any reason whatsoever for using bubble sort?

- In space complexity, maybe?
- Being stable? (what is stable sorting algorithm?)

Well, no - even though Bubble Sort is O(1) for space complexity and it is stable - there are other sorting algorithms with same attributes and better complexity.

Examples: $O(2^n)$

Computing a perfect Strategy for n x n Chess

Print the Power set of a set of size n.

Think about finding a collision in SHA256 / SHA512 …

Ignoring the constant *c* in t(n) = c*n, *i.e O(n)*

What is the complexity of the code below?

```
std::vector<Widget> vec;
for(auto& widget: vec) {
    for(int j=0; j<100; ++j) {
        // assume that below is O(1)
        widget.doSomething();
    }
}
```


Ignoring the constant *c* in t(n) = c*n, *i.e O(n)*

What is the complexity of the code below?

```
std::vector<Widget> vec;
for(auto& widget: vec) {
    for(int j=0; j<100; ++j) {
        // assume that below is O(1)
        widget.doSomething();
    }
}
```

Suppose that we can achieve the same, with $t(n) = n * \log n$

Which would be better?

Ignoring the constant *c* in t(n) = c*n, *i.e O(n)*

What is the complexity of the code below?

```
std::vector<Widget> vec;
for(auto& widget: vec) {
    for(int j=0; j<100; ++j) {
        // assume that below is O(1)
        widget.doSomething();
    }
}
```

Suppose that we can achieve the same, with *t(n) = n * log n*
Which would be better?

log(n) < 64 < 100, for any 64 bit *n*

Ignoring the constant *c* in t(n) = c*n, *i.e O(n)*

What is the complexity of the code below?

```
std::vector<Widget> vec;
for(auto& widget: vec) {
    for(int j=0; j<100; ++j) {
        // assume that below is O(1)
        widget.doSomething();
    }
}
```

Suppose that we can achieve the same, with *t(n) = n * log n*
Which would be better?

log(n) < 64 < 100, for any 64 bit *n*

log(vector::size) <= 64

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```


Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```


Above calls iterate over vec twice.

Would it be better to perform the two operations inside a single loop?

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```

Above calls iterate over vec twice.
Would it be better to perform the two operations inside a single loop?

Two loops ~ n + n = O(n)
Single loop with two operations ~ 2n = O(n)

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```

Above calls iterate over vec twice.
Would it be better to perform the two operations inside a single loop?

Two loops ~ n + n = O(n)
Single loop with two operations ~ 2n = O(n)

So are they the same?

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```

Above calls iterate over vec twice.
Would it be better to perform the two operations inside a single loop?

Two loops ~ n + n = O(n)
Single loop with two operations ~ 2n = O(n)

So are they the same? Complexity-wise yes, practically - not necessarily!

Two calls to std algorithms

```
unsigned long sum = std::accumulate(vec.begin(), vec.end(), 0);
double inner_product =
        std::inner_product(vec.begin(), vec.end(), vec.begin(), 0.0);
```

Above calls iterate over vec twice.
Would it be better to perform the two operations inside a single loop?

It might be better due to ***data locality***
see benchmarks with std::list and std::vector
(and see also SO discussion with additional alternatives).

Two calls to std algorithms

A note:

std::ranges allows consecutive algorithm calls to be “lazily attached” into a single loop

Two calls to std algorithms

A note:

std::ranges allows consecutive algorithm calls to be “lazily attached” into a single loop

ranges require its own talk, but if you are interested…

Here is a relevant code example (courtesy of Dvir Yitzchaki)

You may also want to watch Dvir’s CppCon 2019 talk on ranges

Best Practices

Which loop is more important?

```
for (int i=0; i < n; ++i) {
    Operation 1
    for (int j=0; j < n; ++j) {
        Operation 2
    }
}
```

Which loop is more important?

```
for (int i=0; i < n; ++i) {
    Operation 1 ← preformed n times
    for (int j=0; j < n; ++j) {
        Operation 2 ← performed n² times
    }
}
```


Break out of loops

Design your algorithm to break as early as possible from the loop

e.g. Bubble Sort with a flag on whether no swaps made in inner loop.

Break when it takes too long...

Design your algorithms to break if it doesn't conclude within reasonable time. Otherwise, you are stucking your entire process, usually when the result is no longer required.

** This has more to do with algorithm design than algorithmic complexity*

Simple calls may hide non-constant complexity

Remember to take into account the inner loops

```
std::vector<Trigger> triggers;
triggers.reserve(matrix.rows());
for (const auto& row: matrix) {
    triggers.push_back(Trigger::create(row));
}
```

Time vs. Space Complexity

max_occurences_item(vec)

Time vs. Space Complexity

max_occurences_item(vec)

Option 1 - sort and count:

O(1) in space	[worst case]
O(n log n) in time	[worst and average case]

Option 2 - index and count:

O(n) in space	[worst case]
O(n) in time	[on average]

Setup Time (1)

Setup time vs. query time: indexing (e.g. previous slide)

Setup Time (2)

What is the best practical way to sort a list?
It may be: copy to a vector, sort the vector, assign back to a list
(already presented above...)

See benchmark
See also SO discussion

Setup Time (3)

Setup time to achieve cache locality / branch prediction / other accelerations:
[This is one of the most famous questions in SO]

On the other hand, benchmarks are quite confusing…
- [a benchmark without optimization] (not a good way to benchmark)
- [a benchmark with -O3] (unsorted wins!)
- [another benchmark with -O3] (now pre-sort wins!)

Picking the right container

std::vector is the best, it's not *us* who say that, [it is the spec]:

> When choosing a container, remember **`vector`** is best;
> leave a comment to explain if you choose from the rest!

std::unordered_map
make sure to provide a good enough hash function for your key,
or forget about amortized O(1) operations…

[hash function requirements in the spec]

Using std algorithms

Don’t reinvent the wheel

e.g. don’t implement your own sort, you may accidentally implement bubble sort

Summary

Summary

Implications of bad algorithms and improper use of data structures are potentially much bigger than other micro-performance improvements

Switching to a better algorithm can decrease runtime dramatically!

Summary

Thinking about algorithmic complexity is not pre-optimization

It’s an essential element of your design and its ability to scale

Summary

The theoretical worst case Big O shouldn't be your only decision factor:

- In real life, **constants** are important: 2n is better than 4n
- In real life, we might choose an algorithm with better **average performance** but *worse worst case complexity*
- **Memory locality** is highly important

Summary

Remember the tradeoffs:

- **Prior setup** (e.g. sorting / indexing)
- **Space vs. Time** - using space to save runtime (e.g. caching, indexing)

Summary

A final note on Space Complexity

Summary

A final note on Space Complexity

the *Conference* ⇔ *Wardrobe* complexity problem

Thank you!

```
void conclude(auto greetings) {
    while(still_time() && have_questions()) {
        ask();
    }
    greetings();
}

conclude([]{ std::cout << "Thank you!"; });
```