Algorithmic Complexity
Amir Kirsh & Adam Segoli Schubert

Why this talk?
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

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
Amir Kirsh

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

**Developer Advocate**

Co-Organizer of the CoreCpp conference and meetup group

Adam Segoli Schubert

Software consultant and programming instructor.

Part of Dev Advocate Office at

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?

Scalability
Computational Complexity

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

- from [Computational complexity](https://en.wikipedia.org/wiki/Computational_complexity) in wikipedia

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

\[
\begin{align*}
\text{n} & \rightarrow f(n) \\
n & \text{- is size of input} \\
f(n) & \text{- is the amount of resources required to run the algorithm} \\
\text{Time} & \text{- number of required elementary operations} \\
& \text{Often denoted by } T(n) \text{ or } t(n) \\
\text{Space} & \text{- Amount of memory required} \\
& \text{Often denoted by } S(n) \text{ or } s(n)
\end{align*}
\]
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:

\[ f(n) \in O(g(n)) \]

iff:

\[ \exists k > 0 \ \exists n_0 \ \forall n > n_0: \ f(n) \leq k \cdot g(n) \]
Only the dominating factor counts

In Big O, we care about asymptotic analysis, when \( n \to \infty \)

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 \( n \to \infty \)

Thus, for example:

\[
t(n) = k_1 \cdot n \cdot \log(n) + k_2 \cdot n \in O(n \cdot \log(n))
\]

Not in this talk...

<table>
<thead>
<tr>
<th>Notation</th>
<th>Name</th>
<th>Short Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(g(n)) )</td>
<td>Theta</td>
<td>f(n) is bounded both above and below by g(n) asymptotically</td>
</tr>
<tr>
<td>( \Omega(g(n)) )</td>
<td>Omega</td>
<td>f(n) is bounded below by g(n) asymptotically</td>
</tr>
<tr>
<td>( o(g(n)) )</td>
<td>Small o</td>
<td>f(n) is dominated by g(n) asymptotically</td>
</tr>
<tr>
<td>( \omega(g(n)) )</td>
<td>Small omega</td>
<td>f(n) dominates g(n) asymptotically</td>
</tr>
<tr>
<td>( \tilde{O}(g(n)) )</td>
<td>Tilde O</td>
<td>same as big O, but “ignores” logarithmic factors</td>
</tr>
</tbody>
</table>
Let the Charts talk

![Graph showing the O(1) complexity](chart.png)
Let the Charts talk
Let the Charts talk

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
Let the Charts talk

Algorithmic Complexity @ CppCon 2021

Let the Charts talk

Algorithmic Complexity @ CppCon 2021
Let the Charts talk

Time Estimates
## Time Estimates

<table>
<thead>
<tr>
<th>n</th>
<th>O(1)</th>
<th>O(log n)</th>
<th>O(n)</th>
<th>O(n log n)</th>
<th>O(n^2)</th>
<th>O(n^3)</th>
<th>O(2^n)</th>
<th>O(n!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 μs</td>
<td>1 μs</td>
<td>1 μs</td>
<td>1 μs</td>
<td>1 μs</td>
<td>2 μs</td>
<td>1 μs</td>
<td>1 μs</td>
</tr>
<tr>
<td>10</td>
<td>1 μs</td>
<td>3 μs</td>
<td>10 μs</td>
<td>34 μs</td>
<td>100 μs</td>
<td>1 ms</td>
<td>1 ms</td>
<td>3.6 seconds</td>
</tr>
<tr>
<td>100</td>
<td>1 μs</td>
<td>6 μs</td>
<td>100 μs</td>
<td>665 μs</td>
<td>10 ms</td>
<td>1 sec</td>
<td>&gt;400 trillion centuries</td>
<td>&gt;googol centuries</td>
</tr>
<tr>
<td>1,000</td>
<td>1 μs</td>
<td>9 μs</td>
<td>1 ms</td>
<td>~10 ms</td>
<td>1 sec</td>
<td>16.67 min</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,000</td>
<td>1 μs</td>
<td>13 μs</td>
<td>10 ms</td>
<td>~133 ms</td>
<td>1.67 min</td>
<td>~12 days</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100,000</td>
<td>1 μs</td>
<td>16 μs</td>
<td>100 ms</td>
<td>1.67 sec</td>
<td>2.78 hours</td>
<td>~32 years</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1 μs</td>
<td>19 μs</td>
<td>1 sec</td>
<td>~20 sec</td>
<td>~12 days</td>
<td>~32,000 years</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

* 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 $i$

$O(1)$
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)

How do we know?

push\_back
(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)

**Case B**
- There isn’t enough capacity
- Needs to move / copy the vector ~ O(n)

To have: `push_back` ~ amortized O(1):
- At most 1 of n calls may be of case B

---

**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 noexcept 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`

```cpp
Widget(Widget&& w) noexcept { /* ... */ }
```

See benchmark
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

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 also spec requirement for std::nth_element
(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

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?

- 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 $t(n) = cn$, i.e $O(n)$

What is the complexity of the code below?

```cpp
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 $t(n) = n \cdot \log n$

Which would be better?
Ignoring the constant \( c \) in \( t(n) = c \cdot n \), i.e \( O(n) \)

What is the complexity of the code below?

```cpp
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 \( t(n) = n \cdot \log n \)

Which would be better?

\[ \log(n) < 64 < 100, \text{ for any 64 bit } n \]

\[ \log(\text{vector::size}) \leq 64 \]
Two calls to std algorithms

```cpp
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

```cpp
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

```cpp
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!

It might be better due to **data locality**
see benchmarks with `std::list` and `std::vector`
(and see also [SO discussion](https://stackoverflow.com/questions/12345678) with additional alternatives).
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?

```cpp
for (int i=0; i < n; ++i) {
    Operation 1
    for (int j=0; j < n; ++j) {
        Operation 2
    }
}
```
Which loop is more important?

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

```cpp
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)

**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

the Conference ⇔ Wardrobe complexity problem
void conclude(auto greetings) {
    while(still_time() && have_questions()) {
        ask();
    }
    greetings();
}

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