Complexity Analysis of Algorithms

Jordi Cortadella
Department of Computer Science

Estimating runtime

What is the runtime of \( g(n) \)?

\[
\text{void } g(\text{int } n) \{
\quad \text{for (int } i = 0; i < n; ++i) f();
\}\]

\[
\text{Runtime}(g(n)) \approx n \cdot \text{Runtime}(f())
\]

\[
\text{void } g(\text{int } n) \{
\quad \text{for (int } i = 0; i < n; ++i)
\quad \quad \text{for (int } j = 0; j < n; ++j) f();
\}\]

\[
\text{Runtime}(g(n)) \approx n^2 \cdot \text{Runtime}(f())
\]

Complexity analysis

A technique to characterize the execution time of an algorithm independently from the machine, the language and the compiler.

Useful for:

- evaluating the variations of execution time with regard to the input data
- comparing algorithms

We are typically interested in the execution time of large instances of a problem, e.g., when \( n \to \infty \), (asymptotic complexity).

Runtime \( g(n) \) \approx (1 + 2 + 3 + \cdots + n) \cdot \text{Runtime}(f())

\approx \frac{n^2 + n}{2} \cdot \text{Runtime}(f())
**Big O**

- A method to characterize the execution time of an algorithm:
  - Adding two square matrices is $O(n^2)$
  - Searching in a dictionary is $O(\log n)$
  - Sorting a vector is $O(n \log n)$
  - Solving Towers of Hanoi is $O(2^n)$
  - Multiplying two square matrices is $O(n^3)$
  - ...

- The $O$ notation only uses the dominating terms of the execution time. Constants are disregarded.

---

**Big O: formal definition**

- Let $T(n)$ be the execution time of an algorithm when the size of the input data is $n$.
- $T(n)$ is $O(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \leq c \cdot f(n)$ when $n \geq n_0$.

---

**Big O: examples**

- Let $T(n) = 3n^2 + 100n + 5$, then $T(n) = O(n^2)$
- Proof:
  - Let $c = 4$ and $n_0 = 100.05$
  - For $n \geq 100.05$, we have that $4n^2 \geq 3n^2 + 100n + 5$

- $T(n)$ is also $O(n^3)$, $O(n^4)$, etc.
  Typically, the smallest complexity is used.
### Complexity analysis: examples

Let us assume that \( f() \) has complexity \( O(1) \)

\[
\text{for (int } i = 0; i < n; ++i) f(); \rightarrow O(n)
\]

\[
\text{for (int } j = 0; j < n; ++j) f(); \rightarrow O(n^2)
\]

\[
\text{for (int } i = 0; i < n; ++i)
\text{for (int } j = i; j < n; ++j) f(); \rightarrow O(n^2)
\]

\[
\text{for (int } i = 0; i < n; ++i)
\text{for (int } j = 0; j < i; ++j) f(); \rightarrow O(n^3)
\]

\[
\text{for (int } i = 0; i < n; ++i)
\text{for (int } j = 0; j < n; ++j)
\text{for (int } k = 0; k < n; ++k) f(); \rightarrow O(mnp)
\]

### Complexity analysis: recursion

```c
void f(int n) {
  if (n > 0) {
    DoSomething(n); // O(n)
    f(n/2);
  }
}
```

\[
T(n) = n + T(n/2)
\]

\[
T(n) = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 2 + 1
\]

\[
2 \cdot T(n) = 2n + n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 4 + 2
\]

\[
2 \cdot T(n) - T(n) = T(n) = 2n - 1
\]

\( T(n) \) is \( O(n) \)

### Complexity analysis: recursion

```c
void f(int n) {
  if (n > 0) {
    DoSomething(n); // O(n)
    f(n/2); f(n/2);
  }
}
```

\[
T(n) = n + 2 \cdot T(n/2)
\]

\[
= n + 2 \cdot \left( \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots \right)
\]

\[
= n + n + n + \cdots + n^\log_2 n
\]

\( T(n) \) is \( O(n \log n) \)
Complexity analysis: recursion

```c
void f(int n) {
    if (n > 0) {
        DoSomething(n); // O(n)
        f(n-1);
    }
}
```

$T(n) = n + T(n-1)$

$T(n) = n + (n-1) + (n-2) + \cdots + 2 + 1$

$T(n) = \frac{n^2 + n}{2}$

$T(n)$ is $O(n^2)$

Asymptotic complexity (small values)

$T(n) = 2 \cdot T(n-1)$

$= 2 \cdot 2 \cdot T(n-2)$

$= 2 \cdot 2 \cdot 2 \cdot T(n-3)$

$\vdots$

$= 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot \cdots 2 = 2^n$

$T(n)$ is $O(2^n)$

Asymptotic complexity (larger values)
Let us consider that every operation can be executed in 1 ns ($10^{-9}$ s).

<table>
<thead>
<tr>
<th>Function</th>
<th>$(n = 10^3)$</th>
<th>$(n = 10^4)$</th>
<th>$(n = 10^5)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log_2 n$</td>
<td>10 ns</td>
<td>13.3 ns</td>
<td>16.6 ns</td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>31.6 ns</td>
<td>100 ns</td>
<td>316 ns</td>
</tr>
<tr>
<td>$n$</td>
<td>1 $\mu$s</td>
<td>10 $\mu$s</td>
<td>100 $\mu$s</td>
</tr>
<tr>
<td>$n \log_2 n$</td>
<td>10 $\mu$s</td>
<td>133 $\mu$s</td>
<td>1.7 ms</td>
</tr>
<tr>
<td>$n^2$</td>
<td>1 ms</td>
<td>100 ms</td>
<td>10 s</td>
</tr>
<tr>
<td>$n^3$</td>
<td>1 s</td>
<td>16.7 min</td>
<td>11.6 days</td>
</tr>
<tr>
<td>$n^4$</td>
<td>16.7 min</td>
<td>116 days</td>
<td>3171 yr</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$3.4 \cdot 10^{284}$ yr</td>
<td>$6.3 \cdot 10^{2993}$ yr</td>
<td>$3.2 \cdot 10^{30086}$ yr</td>
</tr>
</tbody>
</table>

**Summary**

- Complexity analysis is a technique to analyze and compare algorithms (not programs).
- It helps to have preliminary back-of-the-envelope estimations of runtime (milliseconds, seconds, minutes, days, years?).
- Worst-case analysis is sometimes overly pessimistic. Average case is also interesting (not covered in this course).
- In many application domains (e.g., big data) quadratic complexity, $O(n^2)$, is not acceptable.
- Recommendation: avoid last-minute surprises by doing complexity analysis before writing code.

Source: Jon Kleinberg and Éva Tardos, Algorithm Design, Addison Wesley 2006.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$n^4$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>30</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
</tr>
<tr>
<td>50</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
</tr>
<tr>
<td>100</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
</tr>
<tr>
<td>1,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
</tr>
<tr>
<td>10,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1 sec</td>
<td>20 sec</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

This is often the practical limit for big data.