Complexity Analysis of Algorithms

Jordi Cortadella
Department of Computer Science
Estimating runtime

• What is the runtime of $g(n)$?

```c
void g(int n) {
    for (int i = 0; i < n; ++i) f();
}
```

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

```c
void g(int n) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) f();
}
```

Runtime($g(n)$) $\approx n^2 \cdot \text{Runtime}(f())$
Estimating runtime

• What is the runtime of \( g(n) \)?

```c
void g(int n) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j <= i; ++j) f();
}
```

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

\[
\approx \frac{n^2 + n}{2} \cdot \text{Runtime}(f())
\]
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).
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 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 $n \geq n_0$. 

![Graph showing $T(n)$ and $c \cdot f(n)$, with $n_0$ as the threshold.]
Big O: example

- 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.
### Big O: examples

<table>
<thead>
<tr>
<th>$T(n)$</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5n^3 + 200n^2 + 15$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>$3n^2 + 2^{300}$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$5 \log_2 n + 15 \ln n$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$2 \log n^3$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$4n + \log n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$2^{64}$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$\log n^{10} + 2\sqrt{n}$</td>
<td>$O(\sqrt{n})$</td>
</tr>
<tr>
<td>$2^n + n^{1000}$</td>
<td>$O(2^n)$</td>
</tr>
</tbody>
</table>
## Complexity ranking

<table>
<thead>
<tr>
<th>Function</th>
<th>Common name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n!$</td>
<td>factorial</td>
</tr>
<tr>
<td>$2^n$</td>
<td>exponential</td>
</tr>
<tr>
<td>$n^d$, $d &gt; 3$</td>
<td>polynomial</td>
</tr>
<tr>
<td>$n^3$</td>
<td>cubic</td>
</tr>
<tr>
<td>$n^2$</td>
<td>quadratic</td>
</tr>
<tr>
<td>$n \sqrt{n}$</td>
<td></td>
</tr>
<tr>
<td>$n \log n$</td>
<td>quasi-linear</td>
</tr>
<tr>
<td>$n$</td>
<td>linear</td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>root - $n$</td>
</tr>
<tr>
<td>$\log n$</td>
<td>logarithmic</td>
</tr>
<tr>
<td>1</td>
<td>constant</td>
</tr>
</tbody>
</table>
Complexity analysis: examples

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

for (int i = 0; i < n; ++i) \( f() \);
\( \rightarrow O(n) \)

for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j) \( f() \);
\( \rightarrow O(n^2) \)

for (int i = 0; i < n; ++i)
  for (int j = 0; j <= i; ++j) \( f() \);
\( \rightarrow O(n^2) \)

for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    for (int k = 0; k < n; ++k) \( f() \);
\( \rightarrow O(n^3) \)

for (int i = 0; i < m; ++i)
  for (int j = 0; j < n; ++j)
    for (int k = 0; k < p; ++k) \( f() \);
\( \rightarrow O(mnp) \)
Complexity analysis: recursion

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

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

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

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

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

\[T(n) \text{ is } O(n)\]
Complexity analysis: recursion

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

\[
T(n) = n + 2 \cdot T\left(\frac{n}{2}\right)
\]
\[
= n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + 8 \cdot \frac{n}{8} + \cdots
\]
\[
= n + n + n + \cdots + n = \underbrace{n + n + \cdots + n}_{\log_2 n} = n \log_2 n
\]

\[
T(n) \text{ is } O(n \log n)
\]
Complexity analysis: recursion

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

\[
T(n) = n + T(n - 1)
\]

\[
T(n) = n + (n - 1) + (n - 2) + \cdots + 2 + 1
\]

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

\[T(n) \text{ is } O(n^2)\]
Complexity analysis: recursion

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

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

\[
= 1 + 2 + 4 \cdot T(n - 2)
\]

\[
= 1 + 2 + 4 + 8 \cdot T(n - 3)
\]

\[
\vdots
\]

\[
= 1 + 2 + 4 + 8 + \cdots + 2^{n-1}
\]

\[
= \sum_{i=0}^{n-1} 2^i = 2^n - 1
\]

\[T(n) \text{ is } O(2^n)\]
Asymptotic complexity (small values)
Asymptotic complexity (larger values)

- $2^n$
- $n^3$
- $n^2$
- $n \log n$
- $n$
Execution time: example

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>
How about “big data”?

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

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</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>&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>&lt; 1 sec</td>
<td>18 min</td>
<td>10^{25} years</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>
<td>36 years</td>
<td>very long</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>
<td>10^{17} years</td>
<td>very long</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>
<td>very long</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>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</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>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

This is often the practical limit for big data.
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.