Motivation

Want to compare algorithms – particularly ones that can solve arbitrarily large instances.

We would like to be able to talk about the resources (running time, memory, energy consumption) required by a program/algorithm as a function $f(n)$ of the size $n$ of its input.

Example

How long does a given sorting algorithm take to run on a list of $n$ elements?

Problem 1: the exact running time may depend on
- compiler optimisations
- processor speed
- cache size

Each of these may affect the resource usage by up to a linear factor, making it hard to state a general claim about running times.
Problem 2: Many algorithms that arise in practice have resource usage that can be expressed only as a rather complicated function. E.g.,

\[ f(n) = 20n^2 + 2n \log(n) + (n - 100) \log(n)^2 + \frac{1}{2^n} \log(\log(n)) \]

The main contribution to the value of the function for “large” input sizes \( n \) is the term of the highest order:

\[ 20n^2 \]

We would like to be able to *ignore the terms of lower order*

\[ 2n \log(n) + (n - 100) \log(n)^2 + \frac{1}{2^n} \log(\log(n)) \]

**Order of Growth**

**Example**

Consider two algorithms, one with running time \( f_1(n) = \frac{1}{10}n^2 \), the other with running time \( f_2 = 10n \log n \) (measured in milliseconds).

<table>
<thead>
<tr>
<th>Input size</th>
<th>( f_1(n) )</th>
<th>( f_2(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.01s</td>
<td>2s</td>
</tr>
<tr>
<td>1000</td>
<td>1s</td>
<td>30s</td>
</tr>
<tr>
<td>10000</td>
<td>1m40s</td>
<td>6m40s</td>
</tr>
<tr>
<td>100000</td>
<td>2h47m</td>
<td>1h23m</td>
</tr>
<tr>
<td>1000000</td>
<td>11d14h</td>
<td>16h40h</td>
</tr>
<tr>
<td>10000000</td>
<td>3y3m</td>
<td>8d2h</td>
</tr>
</tbody>
</table>

**Order of growth** provides a way to abstract away from these two problems, and focus on what is essential to the size of the function, by saying that “the (complicated) function \( f \) is of roughly the same size (for large input) as the (simple) function \( g \)”

**“Big-Oh” Asymptotic Upper Bounds**

**Definition**

Let \( f, g : \mathbb{N} \to \mathbb{R} \). We say that \( g \) is asymptotically less than \( f \) (or: \( f \) is an upper bound of \( g \)) if there exists \( n_0 \in \mathbb{N} \) and a real constant \( c > 0 \) such that for all \( n \geq n_0 \),

\[ g(n) \leq c \cdot f(n) \]

Write \( O(f(n)) \) for the class of all functions \( g \) that are asymptotically less than \( f \).

**Example**

\[ g(n) = 3n + 1 \quad \rightarrow \quad g(n) \leq 4n, \text{ for all } n \geq 1 \]

Therefore, \( 3n + 1 \in O(n) \)

**NB**

Asymptotic analysis is about how costs scale as the input increases.
The traditional notation has been

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

instead of \( g(n) \in O(f(n)) \).

It allows one to use \( O(f(n)) \) or similar expressions as part of an equation; of course these ‘equations’ express only an approximate equality. Thus,

\[ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) \]

means

“There exists a function \( f(n) \in O(n) \) such that \( T(n) = 2T\left(\frac{n}{2}\right) + f(n) \).”

**Examples**

\[ 5n^2 + 3n + 2 = O(n^2) \]

\[ n^3 + 2^{100}n^2 + 2n + 2^{2^{100}} = O(n^3) \]

Generally, for constants \( a_k \ldots a_0 \),

\[ a_kn^k + a_{k-1}n^{k-1} + \ldots + a_0 = O(n^k) \]
**“Big-Omega” Asymptotic Lower Bounds**

**Definition**

Let $f, g : \mathbb{N} \to \mathbb{R}$. We say that $g$ is asymptotically greater than $f$ (or: $f$ is a lower bound of $g$) if there exists $n_0 \in \mathbb{N}$ and a real constant $c > 0$ such that for all $n \geq n_0$,

$$g(n) \geq c \cdot f(n)$$

Write $\Omega(f(n))$ for the class of all functions $g$ that are asymptotically greater than $f$.

**Example**

$$g(n) = 3n + 1 \to g(n) \geq 3n, \text{ for all } n \geq 1$$

Therefore, $3n + 1 \in \Omega(n)$

Observe that, somewhat symmetrically

$$g \in \Theta(f) \iff f \in \Theta(g)$$

We obviously have

$$\Theta(f(n)) \subseteq O(f(n)) \quad \text{and} \quad \Theta(f(n)) \subseteq \Omega(f(n)),$$

in fact

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

At the same time the 'Big-Oh' is not a symmetric relation

$$g \in O(f) \not\Rightarrow f \in O(g),$$

but

$$g \in O(f) \iff f \in \Omega(g)$$

**More Examples**

- All logarithms $\log_b x$ have the same order, irrespective of the value of $b$

$$O(\log_2 n) = O(\log_3 n) = \ldots = O(\log_{10} n) = \ldots$$

- Exponentials $r^n, s^n$ to different bases $r < s$ have different orders, e.g. there is no $c > 0$ such that $3^n < c \cdot 2^n$ for all $n$

$$O(r^n) \not\subseteq O(s^n) \subseteq O(t^n) \ldots \quad \text{for} \quad r < s < t \ldots$$

- Similarly for polynomials

$$O(n^k) \not\subseteq O(n^l) \subseteq O(n^m) \ldots \quad \text{for} \quad k < l < m \ldots$$

**“Big-Theta” Notation**

**Definition**

Two functions $f, g$ have the same order of growth if they scale up in the same way:

There exists $n_0 \in \mathbb{N}$ and real constants $c > 0, d > 0$ such that for all $n \geq n_0$,

$$c \cdot f(n) \leq g(n) \leq d \cdot f(n)$$

Write $\Theta(f(n))$ for the class of all functions $g$ that have the same order of growth as $f$.

If $g \in O(f)$ (or $\Omega(f)$) we say that $f$ is an upper bound (lower bound) on the order of growth of $g$; if $g \in \Theta(f)$ we call it a tight bound.

Observe that, somewhat symmetrically

$$g \in \Theta(f) \iff f \in \Theta(g)$$

We obviously have

$$\Theta(f(n)) \subseteq O(f(n)) \quad \text{and} \quad \Theta(f(n)) \subseteq \Omega(f(n)),$$

in fact

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

At the same time the 'Big-Oh' is not a symmetric relation

$$g \in O(f) \not\Rightarrow f \in O(g),$$

but

$$g \in O(f) \iff f \in \Omega(g)$$

**More Examples**

- All logarithms $\log_b x$ have the same order, irrespective of the value of $b$

$$O(\log_2 n) = O(\log_3 n) = \ldots = O(\log_{10} n) = \ldots$$

- Exponentials $r^n, s^n$ to different bases $r < s$ have different orders, e.g. there is no $c > 0$ such that $3^n < c \cdot 2^n$ for all $n$

$$O(r^n) \not\subseteq O(s^n) \subseteq O(t^n) \ldots \quad \text{for} \quad r < s < t \ldots$$

- Similarly for polynomials

$$O(n^k) \not\subseteq O(n^l) \subseteq O(n^m) \ldots \quad \text{for} \quad k < l < m \ldots$$
Here are some of the most common functions occurring in the analysis of the performance of programs (algorithm complexity):

1, \log \log n, \log n, \sqrt{n}, \sqrt{n}(\log n)^k, \sqrt{n}(\log n)^2, \ldots
n, n \log \log n, n \log n, n^{1.5}, n^2, n^3, \ldots
2^n, 2^n \log n, n2^n, 3^n, \ldots
n!, n^n, n^{2n}, \ldots, n^2, n^3, \ldots

Notation: \( O(1) \equiv \text{const} \), although technically it could be any function that varies between two constants \( c \) and \( d \).

Exercise

4.3.5 True or false?
(a) \( 2^{n+1} = O(2^n) \) — true
(b) \( (n + 1)^2 = O(n^2) \) — true
(c) \( 2^{2n} = O(2^n) \) — false
(d) \( (200n)^2 = O(n^2) \) — true

4.3.6 True or false?
(b) \( \log(n^{73}) = O(\log n) \) — true
(c) \( \log n^n = O(\log n) \) — false
(d) \( (\sqrt{n} + 1)^4 = O(n^2) \) — true

Analysing the Complexity of Algorithms

We want to know what to expect of the running time of an algorithm as the input size goes up. To avoid vagaries of the specific computational platform we measure the performance in the number of elementary operations rather than clock time.

Typically we consider the four arithmetic operations, comparisons, and logical operations as elementary; they take one processor cycle (or a fixed small number of cycles).

A typical approach to determining the complexity of an algorithm, i.e. an asymptotic estimate of its running time, is to write down a recurrence for the number of operations as a function of the size of the input.

We then solve the recurrence up to an order of size.
Example: Insertion Sort

Consider the following recursive algorithm for sorting a list. We take the cost to be the number of list element comparison operations. Let \( T(n) \) denote the total cost of running \( \text{InsSort}(L) \).

\[
\text{InsSort}(L):
\]
\[
\text{Input} \quad \text{list } L[0..n-1] \text{ containing } n \text{ elements}
\]
\[
\text{if } n \leq 1 \quad \text{return } L \quad \text{cost} = 0
\]
\[
\text{let } L_1 = \text{InsSort}(L[0..n-2]) \quad \text{cost} = T(n-1)
\]
\[
\text{let } L_2 = \text{result of inserting element } L[n-1] \text{ into } L_1 \text{ (sorted!)} \quad \text{in the appropriate place} \quad \text{cost} \leq n - 1
\]
\[
\text{return } L_2
\]
\[
T(n) = T(n-1) + n - 1 \quad T(1) = 0
\]

Solving the Recurrence

Unwinding \( T(n) = T(n-1) + (n-1) \), \( T(1) = 0 \)

\[
T(n) = T(n-1) + (n-1)
\]
\[
= T(n-2) + (n-2) + (n-1)
\]
\[
= T(n-3) + (n-3) + (n-2) + (n-1)
\]
\[
\vdots
\]
\[
= T(1) + 1 + \ldots + (n-1)
\]
\[
= 0 + 1 + \ldots + (n-1)
\]
\[
= \frac{n(n-1)}{2}
\]
\[
= O(n^2)
\]

Hence, Insertion Sort is in \( O(n^2) \)

We also say: “The complexity of Insertion Sort is quadratic.”

Exercise

Linear recurrence

\[
T(n) = T(n-1) + g(n), \quad T(0) = a
\]

has the precise solution

\[
T(n) = a + \sum_{j=1}^{n} g(j)
\]

Give a tight big-Oh upper bound on the solution if \( g(n) = n^2 \)

\[
T(n) = a + \sum_{j=1}^{n} j^2 = a + \frac{n(n+1)(2n+1)}{6} = O(n^3)
\]
Exercise

Linear recurrence

\[ T(n) = T(n-1) + g(n), \quad T(0) = a \]

has the precise solution

\[ T(n) = a + \sum_{j=1}^{n} g(j) \]

Give a tight big-Oh upper bound on the solution if \( g(n) = n^2 \)

\[ T(n) = a + \sum_{j=1}^{n} j^2 = a + \frac{n(n+1)(2n+1)}{6} = O(n^3) \]

A General Result

Recurrences for algorithm complexity often involve a linear reduction in subproblem size.

Theorem

- **(case 1)** \( T(n) = T(n-1) + bn^k \)
  - solution \( T(n) = O(n^{k+1}) \)

- **(case 2)** \( T(n) = cT(n-1) + bn^k \), \( c > 1 \)
  - solution \( T(n) = O(c^n) \)

This contrasts with divide-and-conquer algorithms, where we solve a problem of size \( n \) by recurrence to subproblems of size \( \frac{n}{c} \) for some \( c \) (often \( c = 2 \)).
A Divide-and-Conquer Algorithm: Merge Sort

MergeSort(L):
Input list L of n elements
if n ≤ 1 then return L cost = 0
let L1 = MergeSort(L[0 ⌈ n/2] − 1]) cost = T(⌈ n/2))
let L2 = MergeSort(L[⌈ n/2 ⌋ .. n − 1]) cost = T(⌊ n/2 ⌋)
merge L1 and L2 into a sorted list L3 cost ≤ n − 1
by repeatedly extracting the least element from L1 or L2
(both are sorted!) and placing in L3
return L3

Let T(n) be the number of comparison operations required by
MergeSort(L) on a list L of length n

\[ T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) \]
\[ T(1) = 0 \]

Solving the Recurrence

\[ T(n) = 2T\left(\frac{n}{2}\right) + (n - 1), \quad T(1) = 0 \]

<table>
<thead>
<tr>
<th>Value of n</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(n)</td>
<td>5</td>
<td>17</td>
<td>49</td>
<td>129</td>
</tr>
<tr>
<td>Ratio</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Conjecture: \( T(n) = n(\log_2 n - 1) + 1 \) for \( n = 2^k \) (Proof?)
Hence, Merge Sort is in \( O(n \log n) \)
Solving the Recurrence

\[ T(n) = 2T\left(\frac{n}{2}\right) + (n - 1), \quad T(1) = 0 \]

\[
\begin{align*}
T(1) &= 0 \\
T(2) &= 2T(1) + (2 - 1) = 0 + 1 \\
T(4) &= 2T(2) + (4 - 1) = 2(0 + 1) + (4 - 1) = 4 + 1 \\
T(8) &= 2T(4) + (8 - 1) = 2(4 + 1) + (8 - 1) = 16 + 1 \\
T(16) &= 2T(8) + (16 - 1) = 2(16 + 1) + (16 - 1) = 48 + 1 \\
T(32) &= 2T(16) + (32 - 1) = 2(48 + 1) + (32 - 1) = 128 + 1
\end{align*}
\]

<table>
<thead>
<tr>
<th>Value of n</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) )</td>
<td>5</td>
<td>17</td>
<td>49</td>
<td>129</td>
</tr>
<tr>
<td>Ratio</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Conjecture: \( T(n) = n(\log_2 n - 1) + 1 \) for \( n = 2^k \) (Proof?)

Hence, Merge Sort is in \( O(n \log n) \)

Exercise

Give a tight big-Oh upper bound on the solution to the divide-and-conquer recurrence

\[ T(n) = T\left(\frac{n}{2}\right) + g(n), \quad T(1) = a \]

for the case \( g(n) = n^2 \)

\[ T(n) = n^2 + \left(\frac{n}{2}\right)^2 + \left(\frac{n}{4}\right)^2 + \ldots = n^2 \left(1 + \frac{1}{4} + \frac{1}{16} + \ldots\right) = O\left(\frac{4}{3}n^2\right) = O(n^2) \]

Master Theorem

Theorem

The following cases cover many divide-and-conquer recurrences that arise in practice:

\[ T(n) = d^\alpha \cdot T\left(\frac{n}{d}\right) + \Theta(n^\beta) \]

- (case 1) \( \alpha > \beta \)
  - Solution: \( T(n) = O(n^\alpha) \)
- (case 2) \( \alpha = \beta \)
  - Solution: \( T(n) = O(n^\alpha \log n) \)
- (case 3) \( \alpha < \beta \)
  - Solution: \( T(n) = O(n^\beta) \)

The situations arise when we reduce a problem of size \( n \) to several subproblems of size \( n/d \). If the number of such subproblems is \( d^\alpha \), while the cost of combining these smaller solutions is \( n^\beta \), then the overall cost depends on the relative magnitude of \( \alpha \) and \( \beta \).
Example

\[ T(n) = T\left(\frac{n}{2}\right) + n^2, \quad T(1) = a \]

Here \( d = 2, \alpha = 0, \beta = 2 \), so we have case 3 and the solution is

\[ T(n) = O(n^\beta) = n^2 \]

Example

Mergesort has

\[ T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) \]

recurrence for the number of comparisons.

Here \( d = 2, \alpha = 1 = \beta \), so we have case 2, and the solution is

\[ T(n) = O(n^\alpha \log(n)) = O(n \log(n)) \]

Exercise

4.3.22 The following algorithm raises a number \( a \) to a power \( n \).

\[
\begin{align*}
p &= 1 \\
i &= n \\
\text{while } i > 0 \text{ do} \\
\quad &p = p \times a \\
\quad &i = i - 1 \\
\text{end while} \\
\text{return } p
\end{align*}
\]

Determine the complexity (no. of comparisons and arithmetic ops).

Exercise

Solve \( T(n) = 3^n T\left(\frac{n}{2}\right) \) with \( T(1) = 1 \)

Let \( n \geq 2 \) be a power of 2 then

\[ T(n) = 3^n \cdot 3^\frac{n}{2} \cdot 3^\frac{n}{4} \cdot 3^\frac{n}{8} \cdot \ldots = 3^{n(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots)} = O(3^{2n}) \]
### Solution

**4.3.22** Number of comparisons and arithmetic operations:

\[
\begin{align*}
\text{cost}(n = 1) &= 4 \quad \text{(why?)} \\
\text{cost}(n > 1) &= 3 + \text{cost}(n - 1)
\end{align*}
\]

This can be described by the recurrence

\[
T(n) = 3 + T(n - 1) \quad \text{with} \quad T(1) = 4
\]

Solution: \( T(n) = O(n) \)

### Exercise

**4.3.21** The following algorithm gives a fast method for raising a number \( a \) to a power \( n \).

\[
\begin{align*}
p & = 1 \\
q & = a \\
i & = n \\
\text{while } i > 0 \text{ do} \\
& \quad \text{if } i \text{ is odd} \text{ then} \\
& \quad \quad p = p \times q \\
& \quad \quad q = q \times q \\
& \quad \quad i = \lfloor i/2 \rfloor \\
& \text{end while} \\
\text{return } p
\end{align*}
\]

Determine the complexity (no. of comparisons and arithmetic ops).

### Solution

**4.3.21** Number of comparisons and arithmetic operations:

\[
\begin{align*}
\text{cost}(n = 1) &= 6 \quad \text{(why?)} \\
\text{cost}(n > 1) &= 4 + \text{cost}(\lfloor n/2 \rfloor) \quad \text{if } n \text{ even} \\
&= 5 + \text{cost}(\lfloor n/2 \rfloor) \quad \text{if } n \text{ odd}
\end{align*}
\]

This can be described by the recurrence

\[
T(n) = 5 + T(\lfloor n/2 \rfloor) \quad \text{with} \quad T(1) = 6
\]

Solution: \( T(n) = O(\log n) \)

### Application: Efficient Matrix Multiplication

The running time of a straightforward algorithm for the multiplication of two \( n \times n \) matrices is \( O(n^3) \). (Why?)

Matrix multiplication can also be carried out blockwise:

\[
\begin{bmatrix}
A & B \\
C & D
\end{bmatrix}
\cdot
\begin{bmatrix}
E & F \\
G & H
\end{bmatrix}
= 
\begin{bmatrix}
AE + BG & AF + BH \\
CE + DG & CF + DH
\end{bmatrix}
\]

This can be implemented by a divide-and-conquer algorithm, recursively computing eight size-\( \frac{n}{2} \) matrix products plus a few \( O(n^2) \)-time matrix additions.

Determine a recurrence to describe the total running time!

\[
T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2)
\]

Solution (Master Theorem)? \( O(n^3) \)
Application: Efficient Matrix Multiplication

The running time of a straightforward algorithm for the multiplication of two \(n \times n\) matrices is \(O(n^3)\). (Why?)

Matrix multiplication can also be carried out blockwise:

\[
\begin{bmatrix}
[A] & [B] \\
[C] & [D]
\end{bmatrix} \cdot \begin{bmatrix}
[E] & [F] \\
[G] & [H]
\end{bmatrix} = \begin{bmatrix}
[AE + BG] & [AF + BH] \\
[CE + DG] & [CF + DH]
\end{bmatrix}
\]

This can be implemented by a divide-and-conquer algorithm, recursively computing eight size-\(\frac{n}{2}\) matrix products plus a few \(O(n^2)\)-time matrix additions.

Determine a recurrence to describe the total running time!

\[
T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2)
\]

Solution (Master Theorem)? \(O(n^3)\)

Strassen's algorithm improves the efficiency by some clever algebra:

\[
X = \begin{bmatrix}
[A] & [B] \\
[C] & [D]
\end{bmatrix} \quad Y = \begin{bmatrix}
[E] & [F] \\
[G] & [H]
\end{bmatrix}
\]

\[
X \cdot Y = \begin{bmatrix}
[P_5 + P_4 - P_2 + P_6] & [P_1 + P_2] \\
[P_3 + P_4] & [P_1 + P_5 - P_3 - P_7]
\end{bmatrix}
\]

where

\[
\begin{align*}
P_1 &= A(F - H) \\
P_2 &= (A + B)H \\
P_3 &= (C + D)E \\
P_4 &= D(G - E) \\
P_5 &= (A + D)(E + H) \\
P_6 &= (B - D)(G + H) \\
P_7 &= (A - C)(E + F)
\end{align*}
\]

Its total running time is described by the recurrence

\[
T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2) \quad (= O(n^{\log_2 7}) \approx O(n^{2.807}))
\]

Summary

- "Big-Oh" notation \(O(f(n))\) for the class of functions for which \(f(n)\) is an upper bound; \(\Omega(f(n))\) and \(\Theta(f(n))\)
- Analysing the complexity of algorithms using recurrences
- Solving recurrences
- General results for recurrences with linear reductions (slide 28) and exponential reductions ("Master Theorem")