COMP9020 Lecture 8
Session 2, 2017
Running Time of Programs

aka "Big-Oh Notation"

- Textbook (R & W) - Ch. 4, Sec. 4.3, 4.5
- Problem set 8
- Supplementary Exercises Ch. 4 (R & W)

Lecture 7 recap

Recursion: Capturing "arbitrarily large in a finite description"

- Recursion in algorithms
- Recursion in data structures
- Analysis of recursion
  - Recursive sequences
  - Structural induction

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).

Input size	$f_1(n)$	$f_2(n)$
100	0.01s	2s
1000	1s	30s
10000	1m40s	6m40s
100000	2h47m	1h23m
1000000	11d14h	16h40h
10000000	3y3m	8d2h

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"

NB

Asymptotic analysis is about how costs **scale** *as the input increases.*

"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

$$g(n) = 3n+1 \quad \to \quad g(n) \leq 4n\,, \text{ for all } n \geq 1$$

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

Example

$$\frac{1}{10}n^2 \in O(n^2) \qquad 10n\log n \in O(n\log n) \qquad O(n\log n) \subsetneq O(n^2)$$

The traditional notation has been

$$g(n) = O(f(n))$$

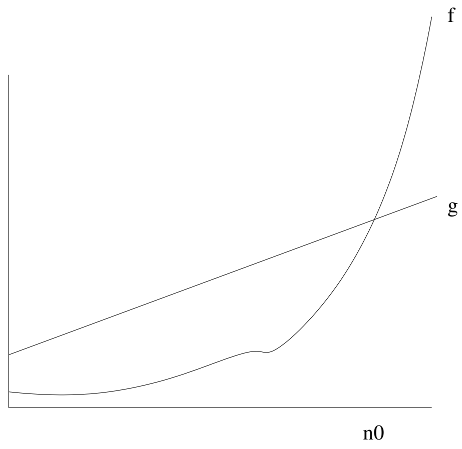
instead of $g(n) \in O(f(n))$.

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,

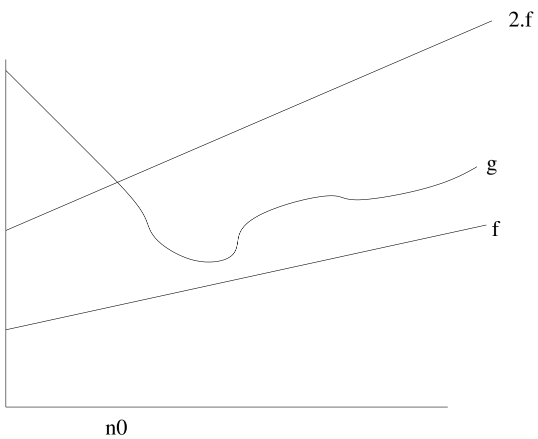
$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

means

"There exists a function $f(n) \in O(n)$ such that $T(n) = 2T(\frac{n}{2}) + f(n)$."



g=O(f)



g=O(f)

Examples

$$5n^2 + 3n + 2 = O(n^2)$$

$$n^3 + 2^{100}n^2 + 2n + 2^{2^{100}} = O(n^3)$$

Generally, for constants $a_k \ldots a_0$,

$$a_k n^k + a_{k-1}n^{k-1} + \ldots + a_0 = O(n^k)$$

“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 an 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

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

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

“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

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

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

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

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

$$O(r^n) \subsetneq O(s^n) \subsetneq O(t^n) \ldots \quad \text{for} \quad r < s < t \ldots$$

- Similarly for polynomials

$$O(n^k) \subsetneq O(n^l) \subsetneq O(n^m) \ldots \quad \text{for} \quad k < l < m \ldots$$

Here are some of the most common functions occurring in the analysis of the performance of programs (algorithm complexity):

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

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

Exercise

$\boxed{4.3.5}$ 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

$\boxed{4.3.6}$ 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

Exercise

$\boxed{4.3.5}$ 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

$\boxed{4.3.6}$ 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 InsSort(L)

InsSort(L):
Input list $L[0..n-1]$ containing n elements

if $n \leq 1$ **then return** L $\quad$ cost $= 0$
let $L_1 =$ InsSort($L[0..n-2]$) $\quad$ cost $= T(n-1)$
let $L_2 =$ result of inserting element $L[n-1]$ into L_1 (sorted!) in the appropriate place $\quad$ cost $\leq n-1$
return L_2

$$T(n) = T(n-1) + n - 1 \qquad T(1) = 0$$

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 InsSort(L)

InsSort(L):
Input list $L[0..n-1]$ containing n elements

if $n \leq 1$ **then return** L $\quad$ cost $= 0$
let $L_1 =$ InsSort($L[0..n-2]$) $\quad$ cost $= T(n-1)$
let $L_2 =$ result of inserting element $L[n-1]$ into L_1 (sorted!) in the appropriate place $\quad$ cost $\leq n-1$
return L_2

$$T(n) = T(n-1) + n - 1 \qquad T(1) = 0$$

Solving the Recurrence

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

$$\begin{aligned}
T(n) &= T(n-1) + (n-1) \\
&= T(n-2) + (n-2) + (n-1) \\
&= T(n-3) + (n-3) + (n-2) + (n-1) \\
&\vdots \\
&= T(1) + 1 + \ldots + (n-1) \\
&= 0 + 1 + \ldots + (n-1) \\
&= \tfrac{n(n-1)}{2} \\
&= O(n^2)
\end{aligned}$$

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)* $T(n) = cT(n-1) + bn^k, \quad c > 1:$
  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 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)* $T(n) = cT(n-1) + bn^k, \quad c > 1:$
  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 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)* $T(n) = cT(n-1) + bn^k, \quad c > 1:$
  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 $n \leq 1$ **then return** L cost $= 0$
let $L_1 = \text{MergeSort}(L[0 \,..\, \lceil \frac{n}{2} \rceil - 1])$ cost $= T(\frac{n}{2})$
let $L_2 = \text{MergeSort}(L[\lceil \frac{n}{2} \rceil \,..\, n - 1])$ cost $= T(\frac{n}{2})$
merge L_1 and L_2 into a sorted list L_3 cost $\leq n - 1$
by repeatedly extracting the least element from L_1 or L_2
(both are sorted!) and placing in L_3
return L_3

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

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

A Divide-and-Conquer Algorithm: Merge Sort

MergeSort(L):

Input list L of n elements

if $n \leq 1$ **then return** L cost $= 0$
let $L_1 = \text{MergeSort}(L[0 \,..\, \lceil \frac{n}{2} \rceil - 1])$ cost $= T(\frac{n}{2})$
let $L_2 = \text{MergeSort}(L[\lceil \frac{n}{2} \rceil \,..\, n - 1])$ cost $= T(\frac{n}{2})$
merge L_1 and L_2 into a sorted list L_3 cost $\leq n - 1$
by repeatedly extracting the least element from L_1 or L_2
(both are sorted!) and placing in L_3
return L_3

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

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

Solving the Recurrence

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

$$\begin{aligned}
T(1) &= 0 \\
T(2) &= 2T(1) + (2-1) & & = 0 + 1 \\
T(4) &= 2T(2) + (4-1) & = 2(0+1) + (4-1) & = 4 + 1 \\
T(8) &= 2T(4) + (8-1) & = 2(4+1) + (8-1) & = 16 + 1 \\
T(16) &= 2T(8) + (16-1) & = 2(16+1) + (16-1) & = 48 + 1 \\
T(32) &= 2T(16) + (32-1) & = 2(48+1) + (32-1) & = 128 + 1
\end{aligned}$$

Value of n	4	8	16	32
$T(n)$	5	17	49	129
Ratio	1	2	3	4

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

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

$$\begin{aligned}
T(1) &= 0 \\
T(2) &= 2T(1) + (2-1) & & = 0 + 1 \\
T(4) &= 2T(2) + (4-1) & = 2(0+1) + (4-1) & = 4 + 1 \\
T(8) &= 2T(4) + (8-1) & = 2(4+1) + (8-1) & = 16 + 1 \\
T(16) &= 2T(8) + (16-1) & = 2(16+1) + (16-1) & = 48 + 1 \\
T(32) &= 2T(16) + (32-1) & = 2(48+1) + (32-1) & = 128 + 1
\end{aligned}$$

Value of n	4	8	16	32
$T(n)$	5	17	49	129
Ratio	1	2	3	4

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

$$T(n) = 2T\Big(\frac{n}{2}\Big) + (n-1), \quad T(1) = 0$$

$$\begin{array}{llll}
T(1) & = 0 & & \\
T(2) & = 2T(1) + (2-1) & & = 0 + 1 \\
T(4) & = 2T(2) + (4-1) & = 2(0+1) + (4-1) & = 4 + 1 \\
T(8) & = 2T(4) + (8-1) & = 2(4+1) + (8-1) & = 16 + 1 \\
T(16) & = 2T(8) + (16-1) & = 2(16+1) + (16-1) & = 48 + 1 \\
T(32) & = 2T(16) + (32-1) & = 2(48+1) + (32-1) & = 128 + 1
\end{array}$$

Value of n	4	8	16	32
$T(n)$	5	17	49	129
Ratio	1	2	3	4

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

$$T(n) = T\Big(\frac{n}{2}\Big) + g(n), \quad T(1) = a$$

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

$$T(n) = n^2 + (\frac{n}{2})^2 + (\frac{n}{4})^2 + \ldots = n^2(1 + \frac{1}{4} + \frac{1}{16} + \ldots) = O(\frac{4}{3}n^2) = O(n^2)$$

Exercise

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

$$T(n) = T\Big(\frac{n}{2}\Big) + g(n), \quad T(1) = a$$

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

$$T(n) = n^2 + (\frac{n}{2})^2 + (\frac{n}{4})^2 + \ldots = n^2(1 + \frac{1}{4} + \frac{1}{16} + \ldots) = O(\frac{4}{3}n^2) = O(n^2)$$

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^α, while the cost of combining these smaller solutions is n^β, then the overall cost depends on the relative magnitude of α and β.

Master Theorem: Examples

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

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

Let $n \geq 2$ be a power of 2 then

$$T(n) = 3^n \cdot 3^{\frac{n}{2}} \cdot 3^{\frac{n}{4}} \cdot 3^{\frac{n}{8}} \cdot \ldots = 3^{n(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots)} = O(3^{2n})$$

Exercise

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

Let $n \geq 2$ be a power of 2 then

$$T(n) = 3^n \cdot 3^{\frac{n}{2}} \cdot 3^{\frac{n}{4}} \cdot 3^{\frac{n}{8}} \cdot \ldots = 3^{n(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots)} = O(3^{2n})$$

Exercise

$\boxed{4.3.22}$ The following algorithm raises a number a to a power n.

$p = 1$
$i = n$
while $i > 0$ **do**
  $p = p * a$
  $i = i - 1$
end while
return p

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

Solution

$\boxed{4.3.22}$ Number of comparisons and arithmetic operations:

cost$(n = 1) = 4$ (why?)

cost$(n > 1) = 3 +$ cost$(n - 1)$

This can be described by the recurrence
$T(n) = 3 + T(n-1)$ with $T(1) = 4$

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

Exercise

$\boxed{4.3.21}$ The following algorithm gives a fast method for raising a number a to a power n.

$p = 1$
$q = a$
$i = n$
while $i > 0$ **do**
    if i is odd **then**
        $p = p * q$
    $q = q * q$
    $i = \lfloor \frac{i}{2} \rfloor$
end while
return p

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

Solution

$\boxed{4.3.21}$ Number of comparisons and arithmetic operations:

cost$(n = 1) = 6$ (why?)

cost$(n > 1) = 4 +$ cost$(\lfloor \frac{n}{2} \rfloor)$ if n even
cost$(n > 1) = 5 +$ cost$(\lfloor \frac{n}{2} \rfloor)$ if n odd

This can be described by the recurrence
$T(n) = 5 + T(\frac{n}{2})$ with $T(1) = 6$

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 mutliplication can also be carried out blockwise:

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

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)?

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 mutliplication can also be carried out blockwise:

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

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 mutliplication can also be carried out blockwise:

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

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

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

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

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

where

$$\begin{array}{lll} \mathbf{P_1} = \mathbf{A}(\mathbf{F}-\mathbf{H}) & \mathbf{P_3} = (\mathbf{C}+\mathbf{D})\mathbf{E} & \mathbf{P_5} = (\mathbf{A}+\mathbf{D})(\mathbf{E}+\mathbf{H}) \\ \mathbf{P_2} = (\mathbf{A}+\mathbf{B})\mathbf{H} & \mathbf{P_4} = \mathbf{D}(\mathbf{G}-\mathbf{E}) & \mathbf{P_6} = (\mathbf{B}-\mathbf{D})(\mathbf{G}+\mathbf{H}) \\ & & \mathbf{P_7} = (\mathbf{A}-\mathbf{C})(\mathbf{E}+\mathbf{F}) \end{array}$$

Its total running time is described by the recurrence

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2) \qquad (= O(n^{\log_2 7}) \simeq O(n^{2.807}))$$

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")