COMP9020 22T1
Week 7
Induction, Recursion, Big-Oh Notation

- Textbook (R & W) - Ch. 4, Sec. 4.2-4.6
- Problem set week 7 + quiz

Inductive Reasoning

Suppose we would like to reach a conclusion of the form

P(x) *for all* x (of some type)

Inductive reasoning (as understood in philosophy) proceeds from examples.

E.g. From "This swan is white, that swan is white, in fact every swan I have seen so far is white"

Conclude: "Every Swan is white"

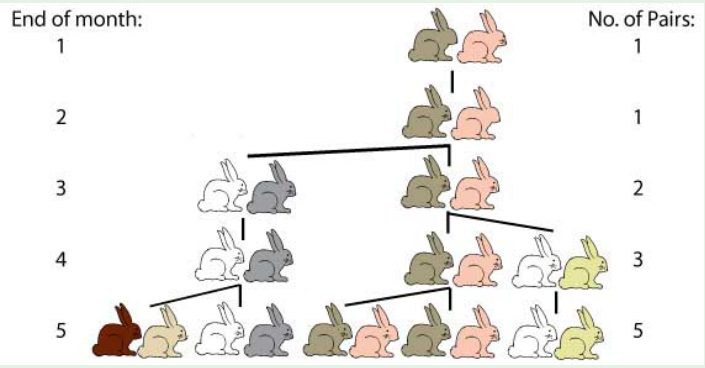
NB

This may be a good way to discover hypotheses.

But it is not a valid principle of reasoning!

Mathematical induction is a variant that is valid.

Example



Fibonacci Numbers:

$\text{FIB}(1) = 1$
$\text{FIB}(2) = 1$
$\text{FIB}(n) = \text{FIB}(n-1) + \text{FIB}(n-2) \quad \text{for all } n > 2$

$\text{FIB}(1)$	1
$\text{FIB}(2)$	1
$\text{FIB}(3)$	2
$\text{FIB}(4)$	3

$\text{FIB}(5)$	5
$\text{FIB}(6)$	8
$\text{FIB}(7)$	13
$\text{FIB}(8)$	21

$\text{FIB}(9)$	34
$\text{FIB}(10)$	55
$\text{FIB}(11)$	89
$\text{FIB}(12)$	144

Claim: Every 4th Fibonacci number is divisible by 3.
How can we prove this?

Mathematical Induction

Mathematical Induction is based not just on a set of examples, but also a rule for deriving new cases of $P(x)$ from cases for which P is known to hold.
General structure of reasoning by mathematical induction:

Base Case [B]: $P(a_1), P(a_2), \ldots, P(a_n)$ for some small set of examples $a_1 \ldots a_n$ (often $n = 1$)
Inductive Step [I]: A general rule showing that if $P(x)$ holds for some cases $x = x_1, \ldots, x_k$ then $P(y)$ holds for some new case y, constructed in some way from $x_1, \ldots, x_k$.

Conclusion: Starting with $a_1 \ldots a_n$ and repeatedly applying the construction of y from existing values, we can eventually construct all values in the domain of interest.

Mathematical Induction on $\mathbb{N}$

Suppose we start with $x = 0$ and repeatedly apply the construction $x \mapsto x + 1$.
Then we construct values
$0, 0 + 1 = 1, 1 + 1 = 2, 2 + 1 = 3, 3 + 1 = 4, \ldots$
In the limit, this is all of $\mathbb{N}$

The corresponding principle of Mathematical Induction on $\mathbb{N}$:

Base Case [B]: $P(0)$
Inductive Step [I]: $\forall k \geq 0\,(P(k) \Rightarrow P(k+1))$

Conclusion: $\forall n \in \mathbb{N}\ P(n)$

Inductive Hypothesis

To prove the Inductive Step [I], $P(k) \Rightarrow P(k+1)$ for $k \geq 0$, we typically proceed as follows:

Assume $P(k)$, for an arbitrary $k \geq 0$

$\vdots$ (steps of reasoning, often using the assumption that $P(k)$)

Conclude $P(k+1)$.

Here $P(k)$ is called the *Inductive Hypothesis*

Example

Theorem. For all $n \in \mathbb{N}$, we have

$$P(n): \quad \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Proof.

[B] $P(0)$, i.e.

$$\sum_{i=0}^{0} i = \frac{0(0+1)}{2}$$

[I] $\forall k \geq 0\,(P(k) \Rightarrow P(k+1))$, i.e.

$$\sum_{i=0}^{k} i = \frac{k(k+1)}{2} \quad \Rightarrow \quad \sum_{i=0}^{k+1} i = \frac{(k+1)(k+2)}{2}$$

(proof?) □

Example (cont'd)

Proof.

Inductive step [I]:

$$\begin{aligned}
\sum_{i=0}^{k+1} i &= \left(\sum_{i=0}^{k} i\right) + (k+1) \\
&= \frac{k(k+1)}{2} + (k+1) \qquad \text{(by the inductive hypothesis)} \\
&= \frac{k(k+1) + 2(k+1)}{2} \\
&= \frac{(k+1)(k+2)}{2}
\end{aligned}$$

Exercise

Consider an **increasing** function $f : \mathbb{N} \longrightarrow \mathbb{N}$
i.e., $\forall m, n\,(m \leq n \;\Rightarrow\; f(m) \leq f(n))$
and a function $g : \mathbb{N} \longrightarrow \mathbb{N}$ such that

- $f(0) < g(0)$
- $f(1) = g(1)$
- if $f(k) \geq g(k)$ then $f(k+1) \geq g(k+1)$, for all $k \in \mathbb{N}$

Always true, false or could be either?
(a) $f(n) > g(n)$ for all $n \geq 1$ — false
(b) $f(n) > g(n)$ for some $n \geq 1$ — could be either
(c) $f(n) \geq g(n)$ for all $n \geq 1$ — true
(d) g is decreasing $(m \leq n \;\Rightarrow\; g(m) \geq g(n))$ — could be either

Exercise

Consider an **increasing** function $f : \mathbb{N} \longrightarrow \mathbb{N}$
i.e., $\forall m, n\,(m \leq n \;\Rightarrow\; f(m) \leq f(n))$
and a function $g : \mathbb{N} \longrightarrow \mathbb{N}$ such that

- $f(0) < g(0)$
- $f(1) = g(1)$
- if $f(k) \geq g(k)$ then $f(k+1) \geq g(k+1)$, for all $k \in \mathbb{N}$

Always true, false or could be either?
(a) $f(n) > g(n)$ for all $n \geq 1$ — false
(b) $f(n) > g(n)$ for some $n \geq 1$ — could be either
(c) $f(n) \geq g(n)$ for all $n \geq 1$ — true
(d) g is decreasing ($m \leq n \;\Rightarrow\; g(m) \geq g(n)$) — could be either

Induction From m Upwards

If

[B] $P(m)$

[I] $\forall k \geq m\,(P(k) \Rightarrow P(k+1))$

then

[C] $\forall n \geq m\,(P(n))$

Example

Theorem. For all $n \geq 1$, the number $8^n - 2^n$ is divisible by 6.

Induction proof:

[B] $8^1 - 2^1$ is divisible by 6

[I] if $8^k - 2^k$ is divisible by 6, then so is $8^{k+1} - 2^{k+1}$, for all $k \geq 1$

Prove [I] using the "trick" to rewrite 8^{k+1} as $8 \cdot (8^k - 2^k + 2^k)$ which allows you to apply the Ind. Hyp. on $8^k - 2^k$

Induction Steps $\ell > 1$

If

[B] $P(m)$

[I] $P(k) \Rightarrow P(k+\ell)$ for all $k \geq m$

then

[C] $P(n)$ for every ℓ'th $n \geq m$

Example

$$\text{FIB}(1) = \text{FIB}(2) = 1$$
$$\text{FIB}(n) = \text{FIB}(n-1) + \text{FIB}(n-2)$$

Every 4th Fibonacci number is divisible by 3.

Induction proof:

[B] $\text{FIB}(4) = 3$ is divisible by 3

[I] if $3 \mid \text{FIB}(k)$, then $3 \mid \text{FIB}(k+4)$, for all $k \geq 4$

Prove [I] by rewriting $\text{FIB}(k+4)$ in such a way that you can apply the Ind. Hyp. on $\text{FIB}(k)$

Strong Induction

This is a version in which the inductive hypothesis is stronger. Rather than using the fact that $P(k)$ holds for a single value, we use *all* values up to k.

If

[B] $P(m)$

[I] $[P(m) \wedge P(m+1) \wedge \ldots \wedge P(k)] \Rightarrow P(k+1)$ for all $k \geq m$

then

[C] $P(n)$, for all $n \geq m$

Example

Claim: All integers ≥ 2 can be written as a product of primes.

[B] 2 is a product of primes

[I] If all x with $2 \leq x \leq k$ can be written as a product of primes, then $k+1$ can be written as a product of primes, for all $k \geq 2$

Negative Integers, Backward Induction

NB

Induction can be conducted over any subset of $\mathbb{Z}$ with least element. Thus m can be negative; eg. base case $m = -10^6$.

NB

One can apply induction in the 'opposite' direction $p(m) \Rightarrow p(m-1)$. It means considering the integers with the *opposite* ordering where the next number after n is $n-1$. Such induction would be used to prove some $p(n)$ for all $n \leq m$.

NB

Sometimes one needs to reason about all integers $\mathbb{Z}$. This requires *two* separate simple induction proofs: one for $\mathbb{N}$, another for $-\mathbb{N}$. They both would start form some initial values, which could be the same, e.g. zero. Then the first proof would proceed through positive integers; the second proof through negative integers.

Forward-Backward Induction

Idea

To prove $P(n)$ *for all* $n \geq k_0$

- *verify* $P(k_0)$
- *prove* $P(k_i)$ *for infinitely many* $k_0 < k_1 < k_2 < k_3 < \ldots$
- *fill the gaps*
  $P(k_1) \Rightarrow P(k_1 - 1) \Rightarrow P(k_1 - 2) \Rightarrow \ldots \Rightarrow P(k_0 + 1)$
  $P(k_2) \Rightarrow P(k_2 - 1) \Rightarrow P(k_2 - 2) \Rightarrow \ldots \Rightarrow P(k_1 + 1)$
  .....

[B] $P(k_0)$

[I] if $P(k)$ then $P(k')$ for *some* $k' > k$

[I] and [B] alone imply $P(k_i)$ for infinitely many $k_0 < k_1 < k_2 < \ldots$

[D] $P(k) \Rightarrow P(k-1)$ for all k between k_i's and k_{i+1}'s (downward step)

[C] $\forall n \geq k_0 \, (P(n))$

NB

This form of induction is extremely important for the analysis of algorithms.

Example

Binary search in an (ordered) list of $n-1$ elements requires no more than $\lceil \log_2 n \rceil$ comparisons.

Induction proof:

(i) it holds for $n = 1$

(ii) if it holds for k then it holds for $2k$,
thus true for $2,\ 4,\ 8,\ 16,\ \ldots$

(iii) if it holds for 2^i then it holds for $2^i - 1,\ 2^i - 2, \ldots, 2^{i-1} + 1$,
thus true for all n.

Infinite Descent

To prove that $Q(n)$, for all $n \geq m$, show

- $\neg Q(n) \Rightarrow \neg Q(n')$ for some $n' < n$
- there cannot be arbitrarily small n s.t. $Q(n)$ is false; in particular the "base case" $Q(m)$ is *true*

This amounts to a proof by contradiction: to verify $\forall n\, Q(n)$ we assume (provisionally) its negation $\exists n \neg Q(n)$ and proceed to show that there would have to exist a smaller n' such that $\neg Q(n')$. Usually the conditions of the problem make it clear that no such infinite decreasing chain $\ldots < n'' < n' < n$ can possibly exist.

Example

For a planar, connected graph let F be the number of *faces* (enclosures) including the exterior face, E the number of edges, and V the number of vertices.

Euler's formula: $$V - E + F = 2 \qquad \text{(EF)}$$

Proof.

Suppose $G = (V, E)$ is a planar connected graph that **violates** (EF).

First observe that G must have an edge because it is connected and the graph with just one vertex satisfies (EF).

If G has an outside edge, that is, an edge separating the exterior face from an interior face, then removing that edge results in a smaller (planar, connected) graph, also violating (EF) because both E and F are reduced by 1.

If G has no outside edge then it has a vertex v of degree 1. Removing v reduces both V and E by 1 while F remains unchanged. It follows that we again found a smaller (planar, connected) graph violating (EF). □

Structural Induction

The induction schemes can be applied not only to natural numbers (and integers) but to any partially ordered set in general.

The basic approach is always the same — we need to verify that

- **[I]** for any given object, if the property in question holds for all its predecessors ('smaller' objects) then it holds for the object itself
- **[B]** the property holds for all minimal objects — objects that have no predecessors; they are usually very simple objects allowing immediate verification

Example: Induction on Rooted Trees

We write $T = \langle r;\ T_1, T_2, \ldots, T_k \rangle$ for a tree T with root r and k subtrees at the root $T_1, \ldots, T_k$

If

[B]	$p(\langle v; \rangle)$	for trees with only a root
[I]	$p(T_1) \wedge \ldots \wedge p(T_k) \Rightarrow p(T)$	for all trees $T = \langle r;\ T_1, T_2, \ldots, T_k \rangle$

then

[C] $p(T)$ for every tree T

Example

Theorem

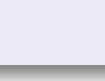
In any rooted tree the number of vertices is one more than the number of edges.

Proof.

[B] If $T = \langle v; \rangle$ then $v(T) = 1$ and $e(T) = 0$

[I] If $T = \langle r;\ T_1, T_2, \ldots, T_k \rangle$ then
$v(T) = 1 + \sum_{i=1}^{k} v(T_i)$ and $e(T) = k + \sum_{i=1}^{k} e(T_i)$

From the Ind. Hyp. on $T_1, \ldots, T_k$ it follows that
$\sum_{i=1}^{k} v(T_i) = \sum_{i=1}^{k} (e(T_i) + 1) = (\sum_{i=1}^{k} e(T_i)) + k$
Therefore
$v(T) = 1 + (\sum_{i=1}^{k} e(T_i)) + k = 1 + e(T)$

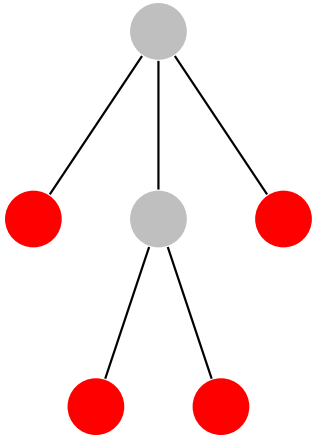


Example

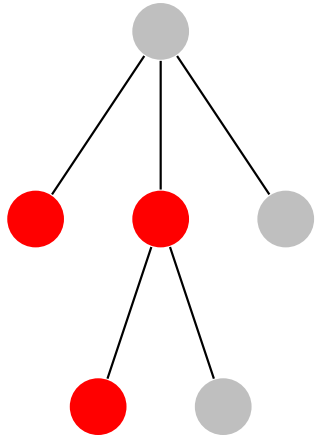
Theorem

In any rooted tree the number of leaves is one more than the number of vertices that have a right sibling.

Proof $\Rightarrow$ this week's problem set



4 leaves



3 vertices with right sibling

Recursive Definitions

They comprise basis (B) and recursive process (R).
A sequence is recursively defined when
(B) some initial terms are specified, perhaps only the first one;
(R) later terms stated as functional expressions of the earlier terms.

Examples

Factorial:

(B) $0! = 1$

(R) $(n+1)! = (n+1) \cdot n!$

Fibonacci numbers:

(B) $\text{FIB}(1) = \text{FIB}(2) = 1$

(R) $\text{FIB}(n) = \text{FIB}(n-1) + \text{FIB}(n-2)$

NB

(R) also called **recurrence formula**

Inductive Proofs About Recursive Definitions

Proofs about recursively defined function very often proceed by a mathematical induction following the structure of the definition.

Example

$\forall n \in \mathbb{N}\left(n! \geq 2^{n-1}\right)$

Proof.

[B] $0! = 1 \geq \frac{1}{2} = 2^{0-1}$

[I] Assume $n \geq 1$.

$$\begin{aligned}(n+1)! = n! \cdot (n+1) &\geq 2^{n-1} \cdot (n+1) && \text{by Ind. Hyp.}\\ &\geq 2^{n-1} \cdot 2 && \text{by } n \geq 1\\ &= 2^n\end{aligned}$$

□

Exercise

4.4.2 Define $s_1 = 1$ and $s_{n+1} = \frac{1}{1+s_n}$ for $n \geq 1$

Then $s_1 = 1,\ s_2 = \frac{1}{2},\ s_3 = \frac{2}{3},\ s_4 = \frac{3}{5},\ s_5 = \frac{5}{8}, \ldots$

Nominators/denominators remind one of the Fibonacci sequence.

Prove by induction that

$$s_n = \frac{\mathrm{FIB}(n)}{\mathrm{FIB}(n+1)}$$

$\Rightarrow$ this week's problem set

NB

$$\lim_{n\to\infty} s_n = \frac{2}{\sqrt{5}+1} = \frac{\sqrt{5}-1}{2} \approx 0.6$$

This is obtained by showing (using induction!) that

$$\mathrm{FIB}(n) = \frac{1}{\sqrt{5}}(r_1^n - r_2^n) \quad \text{where } r_1 = \tfrac{1+\sqrt{5}}{2} \text{ and } r_2 = \tfrac{1-\sqrt{5}}{2}$$

Exercise

4.4.4 (a) Give a recursive definition for the sequence

$$(2,\ 4,\ 16,\ 256,\ \ldots)$$

To generate $a_n = 2^{2^n}$ use $a_n = (a_{n-1})^2$.
(The related "Fermat numbers" $F_n = 2^{2^n} + 1$ are used in cryptography.)

(b) Give a recursive definition for the sequence

$$(2,\ 4,\ 16,\ 65536,\ \ldots)$$

To generate a "stack" of n 2's use $b_n = 2^{b_{n-1}}$.
(These are *Ackermann's numbers*, first used in logic. The inverse function is extremely slow growing; it is important for the analysis of several data organisation algorithms.)

Exercise

4.4.4 (a) Give a recursive definition for the sequence

$$(2,\ 4,\ 16,\ 256,\ \ldots)$$

To generate $a_n = 2^{2^n}$ use $a_n = (a_{n-1})^2$.
(The related "Fermat numbers" $F_n = 2^{2^n} + 1$ are used in cryptography.)

(b) Give a recursive definition for the sequence

$$(2,\ 4,\ 16,\ 65536,\ \ldots)$$

To generate a "stack" of n 2's use $b_n = 2^{b_{n-1}}$.
(These are *Ackermann's numbers*, first used in logic. The inverse function is extremely slow growing; it is important for the analysis of several data organisation algorithms.)

Correctness of Recursive Definitions

A recurrence formula is correct if the computation of any later term can be reduced to the initial values given in (B).

Example (Incorrect definition)

- Function $g(n)$ is defined recursively by

$$g(n) = g(g(n-1)-1)+1, \qquad g(0) = 2.$$

The definition of $g(n)$ is incomplete — the recursion may not terminate:
Attempt to compute $g(1)$ gives

$$g(1) = g(g(0)-1)+1 = g(1)+1 = \ldots = g(1)+1+1+1\ldots$$

When implemented, it leads to an overflow; most static analyses cannot detect this kind of ill-defined recursion.

Example (continued)

However, the definition could be repaired. For example, we can add the specification specify $g(1) = 2$.

Then $g(2) = g(2-1) + 1 = 3$,
$g(3) = g(g(2)-1) + 1 = g(3-1) + 1 = 4$,
$\ldots$

In fact, by induction $\ldots$ $g(n) = n + 1$

This illustrates a very important principle: the boundary (limiting) cases of the definition are evaluated *before* applying the recursive construction.

Mutual Recursion

Several more sophisticated programs employ a technique of two procedures calling each other. Of course, it should be designed so that each consecutive call refers to ever smaller parameters, so that the entire process terminates. This method is often used in computer graphics, in particular for generating fractal images (basis of various imaginary landscapes, among others).

Big-Oh Notation: Motivation

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: 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 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*:

$$20n^2$$

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 g is of *roughly the same size* (for large input) as the (simple) function f"

Asymptotic Upper Bounds

Example

MatrixMultiply(A, B):
  Input matrices $A[1..n, 1..n], B[1..n, 1..n]$

  for $i = 1 \ldots n$ **do**
    for $k = 1 \ldots n$ **do**
      C[i,k] = 0.0
      for $j = 1 \ldots n$ **do**
        C[i,k] = C[i,k] + A[i,j]*B[j,k]

Cost = no. of floating point operations and assignments
$= n^2 + 3n^3$ (why?)

cost function $g(n) = n^2 + 3n^3$ is *asymptotically bound by* function $f(n) = n^3$

"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 $\mathcal{O}(f(n))$ for the class of all functions g that are asymptotically less than f.

Example

$$g(n) = 3n^3 + n^2 \quad \Rightarrow \quad g(n) \leq 4n^3, \text{ for all } n \geq 1$$

Therefore, $3n^3 + n^2 \in \mathcal{O}(n^3)$

Example

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

The traditional notation has been

$$g(n) = \mathcal{O}(f(n))$$

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

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

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

means

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

More Examples

- Generally, for constants $a_k \ldots a_0$,

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

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

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

- 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

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

- Similarly for polynomials

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

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

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

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

Basic math needed for complexity analysis:

- Logarithms
  $\log_b(xy) = \log_b x + \log_b y,\ \log_b(\frac{x}{y}) = \log_b x - \log_b y,$
  $\log_b x^a = a\log_b x,\ \log_b a = \frac{\log_x a}{\log_x b}$
- Exponentials
  $a^{b+c} = a^b a^c,\ a^{bc} = (a^b)^c,\ \frac{a^b}{a^c} = a^{b-c},\ \sqrt[c]{a^b} = a^{\frac{b}{c}},\ b = a^{\log_a b},$

Exercise

4.3.5 True or false?

(a) $2^{n+1} = \mathcal{O}(2^n)$ — true

(b) $(n+1)^2 = \mathcal{O}(n^2)$ — true

(c) $2^{2n} = \mathcal{O}(2^n)$ — false

(d) $(200n)^2 = \mathcal{O}(n^2)$ — true

4.3.6 True or false?

(b) $\log(n^{73}) = \mathcal{O}(\log n)$ — true

(c) $\log n^n = \mathcal{O}(\log n)$ — false

(d) $(\sqrt{n}+1)^4 = \mathcal{O}(n^2)$ — true

Exercise

4.3.5 True or false?

(a) $2^{n+1} = \mathcal{O}(2^n)$ — true

(b) $(n+1)^2 = \mathcal{O}(n^2)$ — true

(c) $2^{2n} = \mathcal{O}(2^n)$ — false

(d) $(200n)^2 = \mathcal{O}(n^2)$ — true

4.3.6 True or false?

(b) $\log(n^{73}) = \mathcal{O}(\log n)$ — true

(c) $\log n^n = \mathcal{O}(\log n)$ — false

(d) $(\sqrt{n}+1)^4 = \mathcal{O}(n^2)$ — true

"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 $g \in \mathcal{O}(f)$ we say that f is (gives) an *upper 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

$$\Theta(f(n)) \subseteq \mathcal{O}(f(n))$$

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

$$g \in \mathcal{O}(f) \not\Rightarrow f \in \mathcal{O}(g)$$

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

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

Solving the Recurrence

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

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

Hence, Insertion Sort is in $\mathcal{O}(n^2)$

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

Exercise

The linear recurrence formula

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

has the precise solution (proof?)

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

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

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

Exercise

The linear recurrence formula

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

has the precise solution (proof?)

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

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

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

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 $T(n) = \mathcal{O}(n^{k+1})$
- *(case 2)* $T(n) = cT(n-1) + bn^k, \quad c > 1:$
  solution $T(n) = \mathcal{O}(c^n)$

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

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

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

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 $\mathcal{O}(n \log n)$

Exercise

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

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

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

$$T(n) = n^2 + \left(\tfrac{n}{2}\right)^2 + \left(\tfrac{n}{4}\right)^2 + \ldots = n^2\left(1 + \tfrac{1}{4} + \tfrac{1}{16} + \ldots\right) < \tfrac{4}{3}n^2 = \mathcal{O}(n^2)$$

Exercise

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

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

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

$$T(n) = n^2 + \left(\tfrac{n}{2}\right)^2 + \left(\tfrac{n}{4}\right)^2 + \ldots = n^2\left(1 + \tfrac{1}{4} + \tfrac{1}{16} + \ldots\right) < \tfrac{4}{3}n^2 = \mathcal{O}(n^2)$$

Master Theorem

Theorem

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

$$T(n) = d^{\alpha} \cdot T\left(\frac{n}{d}\right) + \mathcal{O}(n^{\beta})$$

- *(case 1)* $\alpha > \beta$
  solution $T(n) = \mathcal{O}(n^{\alpha})$
- *(case 2)* $\alpha = \beta$
  solution $T(n) = \mathcal{O}(n^{\alpha} \log n)$
- *(case 3)* $\alpha < \beta$
  solution $T(n) = \mathcal{O}(n^{\beta})$

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.

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

$$T(n) = \mathcal{O}(n^\beta) = n^2$$

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

$$T(n) = \mathcal{O}(n^\alpha \log(n)) = \mathcal{O}(n \log(n))$$

Beyond the Master Theorem

Example

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

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

Exercise

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).

Exercise

4.3.22

Number of comparisons and arithmetic operations:

$\text{cost}(n = 1) = 4$ (why?)

$\text{cost}(n > 1) = 3 + \text{cost}(n - 1)$

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

Solution: $T(n) = \mathcal{O}(n)$

Exercise

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).

Exercise

4.3.21

Number of comparisons and arithmetic operations:

$\text{cost}(n = 1) = 6$ (why?)

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

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

Solution: $T(n) = \mathcal{O}(\log n)$

Application: Efficient Matrix Multiplication

The running time of a straightforward algorithm for the multiplication of two $n \times n$ matrices is $\mathcal{O}(n^3)$ (cf. slide 34).

Matrix multiplication 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 $\mathcal{O}(n^2)$-time matrix additions.

Recurrence to describe the total running time:

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

Solution (Master Theorem!): $\mathcal{O}(n^3)$

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

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

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

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

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

Summary

- Mathematical induction:
  base case(s), inductive hypothesis $P(k)$,
  inductive step $\forall k\,(P(k) \Rightarrow P(k+1))$, conclusion
- Variations:
  strong ind., forward-backward ind., structural ind.
- Recursive definitions
- "Big-Oh" notation $\mathcal{O}(f(n))$ for the class of functions for which $f(n)$ is an upper bound; $\Theta(f(n))$
- Analysing the complexity of algorithms, solving recurrences
- General results for recurrences with linear reductions (slide 49) and exponential reductions ("Master Theorem")

Coming up ...

- Ch. 5, Sec. 5.1-5.3 (Counting and Probability)