Week 7 Induction, Recursion, Big-Oh Notation

Friday, 1 May 2020 4:59 PM

Fibonacci Numbers:

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

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

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

Induction From m Upwards

If
[B] $P(m)$
[I] $\forall k \geq m\,(P(k) \Rightarrow P(k+1))$
then
[C] $\forall n \geq m\,(P(n))$

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 $n \geq m$

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 $n \geq m$

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*
  $P(k_1) \Rightarrow P(k_1 - 1) \Rightarrow P(k_1 - 2) \Rightarrow \ldots \Rightarrow P(k_0 + 1)$
  $P(k_2) \Rightarrow P(k_2 - 1) \Rightarrow P(k_2 - 2) \Rightarrow \ldots \Rightarrow P(k_1 + 1)$
  .....

NB

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

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*

Structural Induction

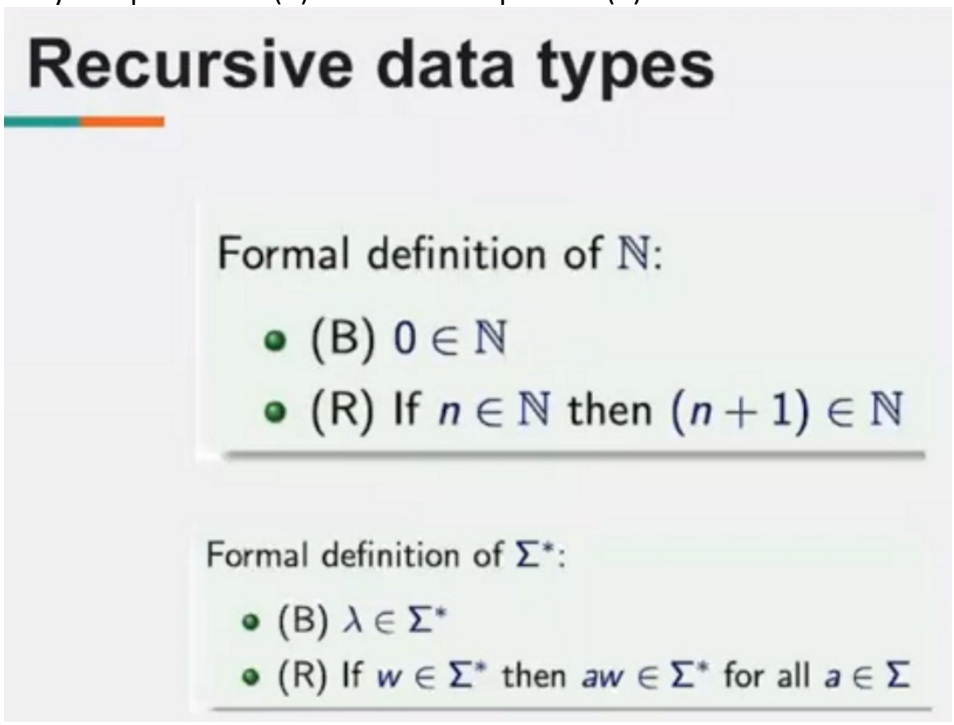
The basic approach is always the same — we need to verify that

- **[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

E.g. Induction on Rooted trees

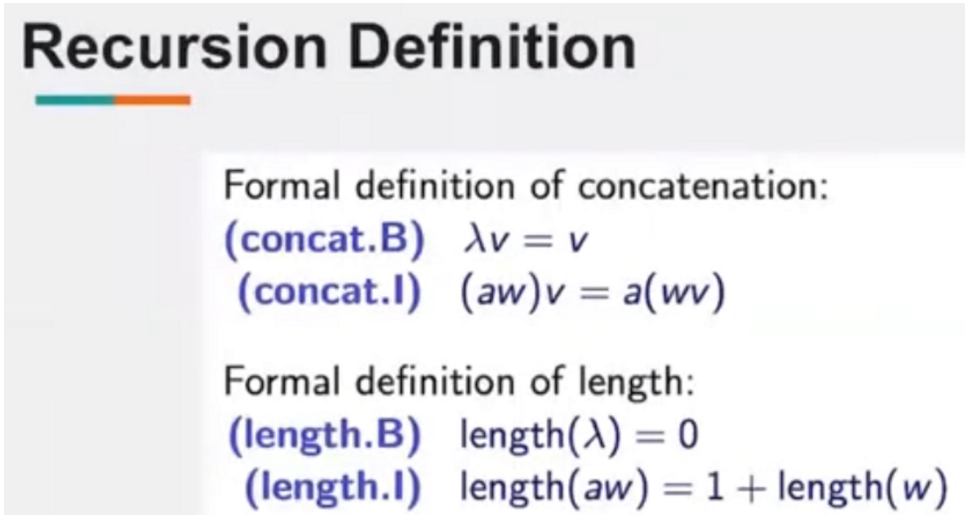
Recursive Definitions

They comprise basis (B) and recursive process (R).



w 单词，加一个a 字母

递归可以定义不同的数据类型也可以定义一个函数



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.

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

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

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

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

Big O

How to describe the time complexity of an algorithm ?

T(n) = O (f(n))

Asymptotic Upper Bounds

Example

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

```
for i = 1...n do
  for k = 1...n do
    C[i,k] = 0.0
    for j = 1...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?)

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

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

The main contribution to the value of the function for "large" input sizes n is the term of the *highest order*:

$$20n^2$$

"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.

Big O

Complexity	Description
1	Execution time does not depend on the input size
$\log(n)$	Very slow increase in running time as n increases
	Whenever n doubles, the running time increases by a constant
n	Linear algorithm are optimal if you need to process n inputs
	Whenever n doubles, then so does the running time
$n\log(n)$	Like linear algorithms, 'linearithmic' algorithms scale to huge problems
	Whenever n doubles, the running time more (but not much more) than doubles
n^2	Quadratic algorithms are practical on relatively small problems
	Whenever n doubles, the running time increases fourfold
n^3	Cubic algorithms are practical on only small problems
	Whenever n doubles, the running time increases eightfold
2^n	Exponential algorithms are generally not practical
	Whenever n doubles, the running time squares!
$n!$	Factorial algorithms are very bad in general to compute

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

- All logarithms $\log_b x$
- Exponentials r^n $\quad O(\log_2(n)) \subseteq O(n^k) \subseteq O(t^n)$
- Similarly for polynomials

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
  $a^{b+c} = a^b a^c$, $a^{bc} = (a^b)^c$, $\frac{a^b}{a^c} = a^{b-c}$, $\sqrt[c]{a^b} = a^{\frac{b}{c}}$, $b = a^{\log_a b}$

"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.

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

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

Algorithms Analysing

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

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

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

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 $L_1 =$ MergeSort($L[0 .. \lceil \frac{n}{2} \rceil - 1]$)	cost $= T(\frac{n}{2})$
let $L_2 =$ 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

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

Master 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 $T(n) = \mathcal{O}(c^n)$

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

Exercise

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 e

Example

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

Induction proof:

Example

$$\begin{aligned} \textsc{fib}(1) &= \textsc{fib}(2) = 1 \\ \textsc{fib}(n) &= \textsc{fib}(n-1) + \textsc{fib}(n-2) \end{aligned}$$

Every 4th Fibonacci number is divisible by 3.

Induction proof:

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 $k \geq 2$

Example

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

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: $V - E + F = 2$ (EF)

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

Theorem

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

Proof.

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 cr

(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. T
function is extremely slow growing; it is important for t
of several data organisation algorithms.)

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

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

用数学归纳法证明Sn成立

4.3.5 True or false?

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

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

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

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

4.3.6 True or false?

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

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

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

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).

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).

Big O exercises

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

Mergesort has

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

recurrence for the number of comparisons.

Beyond the Master Theorem

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