

Mat 2345

Chapter Three

Mathematical Reasoning

Fall 2007

Section 3.3 — Recursive Definitions

Recursive or **inductive** definitions of sets, and functions on recursively defined sets are similar:

▶ **Basis step:**

- ▶ For **sets** — state the basic building blocks (BBBs) of the set
- ▶ For **functions** — state the values of the function on the BBBs.

▶ **Inductive** or **recursive** step:

- ▶ For **sets** — show how to build new things from old with some construction rules
- ▶ For **functions** — show how to compute the value of a function on the new things that can be built knowing the value of the old things.

▶ Extremal Clause

- ▶ For **sets** — if you can't build it with a finite number of applications of steps 1 and 2, then it isn't in the set.
- ▶ For **functions** — a function defined on a recursively defined set does not require an extremal clause.

Note: To prove something is **in** the set, you must show how to **construct** it with a **finite number** of applications of the basis and inductive steps.

To prove something is **not** in the set is often more difficult.

Example: A Recursive Definition of \mathbb{N}

1. **Basis:** 0 is in \mathbb{N} — 0 is in the BBB
2. **Induction:** if n is in \mathbb{N} , then so is $n + 1$
3. **Extremal clause:** If you can't construct it with a finite number of applications of steps 1 and 2, it isn't in \mathbb{N} .

Recursive Function Definitions

Now, given the recursive definition of \mathbb{N} , we can give recursive definitions of functions on \mathbb{N} . One example is the factorial function, $f(n)$:

1. $f(0) = 1$ — the **initial condition** or the value of the function on the BBBs.
2. $f(n + 1) = (n + 1)f(n)$ — the **recurrence** equation, how to define f on the new objects based on its value on old objects.

f is the **factorial function**: $f(n) = n!$ — note how it follows the recursive definition of \mathbb{N}

Proving Assertions

Proofs of assertions about inductively defined objects usually involve a **proof by induction**.

- ▶ Prove the assertion is true for the BBBs in the basis step.
- ▶ Prove that if the assertion is true for the old objects, it must be true for the new objects you can build from the old objects.
- ▶ Conclude the assertion must be true for all objects.

Example: a^n , $n \in \mathbb{N}$

We define a^n inductively, where $n \in \mathbb{N}$:

- ▶ Basis: $a^0 = 1$
- ▶ Induction: $a^{(n+1)} = a^n \times a$

Theorem. $\forall m \forall n [a^m a^n = a^{m+n}]$

Proof. Since the powers of a have been defined inductively, we must use a proof by induction somewhere.

Get rid of the first quantifier of m by Universal Instantiation: assume m is arbitrary.

Now prove the remaining quantified assertion:

$$\forall n [a^m a^n = a^{m+n}]$$

Induction Proof: $\forall m \forall n [a^m a^n = a^{m+n}]$

1. **Base Case.** Let $n = 0$.

$$\text{LHS. } a^m a^0 = a^m (1) = a^m$$

$$\text{RHS. } a^{m+0} = a^m$$

Hence, the two sides are equal.

2. **Inductive Hypothesis.** Assume for some arbitrary $n \geq 0$, that $a^m a^n = a^{m+n}$.

3. **Inductive Step.** Show that $a^m a^{n+1} = a^{m+n+1}$

$$\begin{aligned} a^m a^{n+1} &= a^m (a^n a) && \text{inductive step in definition of } a^n \\ &= (a^m a^n) a && \text{associativity of multiplication} \\ &= a^{m+n} a && \text{IH, substitution} \\ &= a^{(m+0n)+1} && \text{inductive definition of powers of } a \\ &= a^{m+n+1} && \text{associativity of addition} \end{aligned}$$

Thus, $a^m a^n = a^{m+n} \quad \forall n \in \mathbb{N}$

Since m was arbitrary, by *Universal Generalization*,

$$\forall m \forall n [a^m a^n = a^{m+n}]$$

Example: the Fibonacci Sequence

The **Fibonacci sequence** is defined recursively:

1. **Basis.** $f(0) = f(1) = 1$ — two initial conditions
2. **Induction.** $f(n + 1) = f(n) + f(n - 1)$ — the recurrence equation

The sequence begins: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Example: The Set of Strings over an Alphabet

- ▶ A **finite alphabet**, Σ , is defined as a finite set of symbols
- ▶ A **string** (over an alphabet, Σ) is a sequence of 0 or more characters from Σ .
- ▶ The empty or null string is denoted: λ (lambda)
- ▶ We can define the **set of all strings** over a **finite alphabet** Σ recursively.
- ▶ This set is called Σ^* (Sigma star), and is equivalent to $\Sigma^+ \cup \lambda$

Recursive Definition of Strings

► The recursive definition of Σ^* :

1. **Basis:** The empty string, Λ , is in Σ^*
2. **Induction:** If w is in Σ^* and a is a symbol in Σ , then wa is in Σ^* . (a string + a letter)

Note: we can concatenate a on the right or left, but it makes a difference in proofs, since concatenation is not commutative.

3. **Extremal clause:** infinitely long strings cannot be in Σ^* . (Why not?)

Example: Let $\Sigma = \{a, b\}$. Then $aab \in \Sigma^*$

Proof: We construct aab with a finite number of applications of the basis and inductive steps in the definition of Σ^* .

1. $\lambda \in \Sigma^*$ by the basis step
2. By step 1, the induction clause in the definition of Σ^* and the fact that a is in Σ , we can conclude that $\lambda a = a \in \Sigma^*$.
3. Since $a \in \Sigma^*$ from step 2, and $a \in \Sigma$, applying the induction clause again, we conclude that $aa \in \Sigma^*$.
4. Since $aa \in \Sigma^*$ from step 3, and $b \in \Sigma$, applying the induction clause again, we conclude that $aab \in \Sigma^*$.

We have shown $aab \in \Sigma^*$ with a finite number of applications of the basis and induction clauses in the definition, thus we are done.

Example: Well Formed Parenthesis Strings

An inductive definition of the well formed parenthesis strings P :

1. **Basis:** $() \in P$
2. **Induction:** if $w \in P$, then so are $()w$, (w) , and $w()$
3. **Extremal clause:** must be able to construct such strings in a finite application of steps 1 & 2

Prove: $(())()) \in P$

1. $() \in P$ by basis clause
2. $()() \in P$ by step 1 and the induction clause
3. $(())()) \in P$ by step 2 and the induction clause

QED.

Note: $))(() \notin P$ — why not? Can you prove it? (hint: what can we say about the length of strings in P ? How can we order the strings in P ?)

Bit Strings

The set S of bit strings with no more than a single 1 in them can be defined recursively:

1. **Basis:** $\lambda, 0, 1 \in S$
2. **Induction:** if w is in S , then so are $0w$ and $w0$
3. **Extremal clause:** must be able to construct such strings in a finite application of steps 1 & 2

Can we prove that $0010 \in S$?

Section 3.4 Recursive Algorithms

A recursive procedure to find the max in a non-empty list.

We will assume we have built-in functions:

- ▶ **Length()** — which returns the number of elements in the list
- ▶ **Max()** — which returns the larger of two values
- ▶ **Listhead()** — which returns the first element in a list

Note: **Max()** requires one comparison

```
procedure maxlist(...list...){  
  
  // PRE:  list is not empty  
  // POST: returns the largest element in list  
  // strip off head of list and pass the remainder  
  
  if Length(list) is 1 then  
    return Listhead(list)  
  else  
    return Max(Listhead(list),  
               maxlist(remainder_of_list))  
}
```

What happens with the list {29}?

With the list {3, 8, 5}?

How Many Comparisons?

The recurrence equation for the number of comparisons required for a list of length n , $C(n)$ is:

$$C(1) = 0 \quad \text{the initial condition}$$

$$C(n) = 1 + C(n-1) \quad \text{the recurrence equation}$$

$$\text{So, } C(n) \in O(n) \quad \text{as we would expect}$$

A Variant of `Maxlist ()`

Assuming the list length is a power of 2, here is a variant of `maxlist ()` using a Divide-and-Conquer approach.

- ▶ Divide the list in half, and find the maximum of each half
- ▶ Find the `Max ()` of the maximum of the two halves
- ▶ Apply these steps to each list half recursively.
- ▶ What could the base case(s) be?

Maxlist2 () Algorithm

```
procedure maxlist2(...list...){  
  
  // PRE:  list is not empty  
  // POST: returns the largest element in list  
  // Divide list into two lists, take the max of  
  //   the two halves (recursively)  
  
  if Length(list) is 1 then  
    return Listhead(list)  
  else  
    a = maxlist2(first half of list)  
    b = maxlist2(second half of list)  
    return Max(a, b)  
}
```

What happens with the list {29,7}?

With the list {3,8,5,7}?

How Many Comparisons in `maxlist2()`?

- ▶ There are two calls to `maxlist2()`, each of which requires $C(\frac{n}{2})$ operations to find maximum.
- ▶ One comparison is required by the `Max()` function

The recurrence equation for the number of comparisons required for a list of length n , $C(n)$, is:

$$\begin{array}{ll} C(1) = 0 & \text{the initial condition} \\ c(n) = 2C(\frac{n}{2}) + 1 & \text{the recurrence equation} \end{array}$$

Consider A Sampling

n	$C(n) = 2C(\frac{n}{2}) + 1$
$2^0 = 1$	$1 = 2^1 - 1$
$2^1 = 2$	$3 = 2^2 - 1$
$2^2 = 4$	$7 = 2^3 - 1$
$2^3 = 8$	$15 = 2^4 - 1$
$2^4 = 16$	$31 = 2^5 - 1$
\vdots	\vdots
$2^{\log n} = n$	$2^{\log(n)+1} - 1 = 2n - 1 \in O(n)$

Thus, $C(n) = 2^{\log(n)+1} - 1 \in O(n)$

Practice I: Prove $3n^2 + 5n + 4 \in O(n^2)$

Definition of Big-Oh: $f(n) \in O(g(n))$ if there exists positive constants c and N_0 such that $\forall n \geq N_0$ we have $f(n) \leq cg(n)$

We need to find $c > 0$ and $N_0 > 0$ such that:

$$3n^2 + 5n + 4 \leq cn^2 \quad \forall n \geq N_0$$

We note that

$$3n^2 + 5n + 4 \leq 3n^2 + 5n^2 + 4n^2, \text{ when } n > 0 \\ \leq 12n^2$$

and we can choose $c = 12$

$$\begin{aligned} \text{To find } N_0: \quad 3n^2 + 5n + 4 &= 12n^2 \\ 0 &= 9n^2 - 5n - 4 \end{aligned}$$

$$\text{when } n = 1, 9(1)^2 - 5(1) - 4 = 9 - 5 - 4 = 0$$

Thus, $3n^2 + 5n + 4 \leq 12n^2 \quad \forall n \geq 1$, and

therefore, $3n^2 + 5n + 4 \in O(n^2)$

Practice II.

Given $T(n) = 2n - 1$, prove that $T(n) \in O(n)$

Practice III.

Prove that $T(n) = 3n + 2$ if

$$T(n) = \begin{cases} 2 & n = 0 \\ 3 + T(n-1) & n > 0 \end{cases}$$

MergeSort Algorithm

```
list MergeSort(list[1..n]){  
  
  // PRE: none  
  // POST: returns list[1..n] in sorted order  
  // Functional dependency: Merge()  
  
  if n is 0  
    return an empty list  
  else if n is 1  
    return list[1]  
  else {  
    list A = MergeSort(list[1..n/2])  
    list B = MergeSort(list[n/2 + 1..n])  
    list C = Merge(A, B)  
    return C  
  }  
}
```

Time Complexity of MergeSort ()

Prove by induction that the time complexity of MergeSort (), $T(n) \in O(n \log n)$

What we need to do:

- ▶ Establish a Base Case for some small n
- ▶ Prove $T(n) \leq cf(n) \rightarrow T(2n) \leq cf(2n)$

In particular, we need to prove $\forall n \geq N_0$ that:

$$\begin{aligned} T(n) \leq cn \log n \quad \rightarrow \quad T(2n) &\leq c2n \log(2n) \\ &= c2n(\log 2 + \log n) \\ &= c2n \log n + c2n \end{aligned}$$

where $n = 2^k$ for some $k \geq 0$, wlog*

* without loss of generality

Base Case

Let $n = 1$.

$$n \log n = (1) \log(1) = 1(0) = 0$$

But, $T(n)$ is always positive, so this is not a good base case.
Try another one.

Let $n = 2$.

$$\begin{aligned} T(2) &= \text{Time to divide} \\ &\quad + \text{time to MergeSort halves} \\ &\quad + \text{time to Merge} \\ &= 1 + 1 + 1 + 2 = 5 \end{aligned}$$

$$\text{while } n \log n = 2 \log 2 = 2(1) = 2$$

Can we find a constant $c > 0$ such that $5 \leq 2c$?

$$\frac{5}{2} \leq c, \text{ so } \frac{5}{2} \text{ is a lower bound on } c$$

Inductive Hypothesis

Assume for some arbitrary $n \geq 2$ that $T(n) \leq cn \log n$

Inductive Step — Show $T(2n) \leq 2cn \log n + 2cn$

$$\begin{aligned}T(2n) &\leq 1 + T(\lceil \frac{2n}{2} \rceil) + T(\lfloor \frac{2n}{2} \rfloor) + 2n \\&\leq T(n) + T(n) + 2n + 1 \\&\leq 2T(n) + 2n + 1 \\&\leq 2(cn \log n) + 2n + 1 \\&\leq 2cn \log n + 2n + 1\end{aligned}$$

Now, can we find a c such that

$$\begin{aligned}2cn \log n + 2n + 1 &\leq 2cn \log n + 2cn \\2n + 1 &\leq 2cn \\1 &\leq 2cn - 2n \\1 &\leq 2n(c - 1)\end{aligned}$$

Since $n \geq 2$ from base case, $(c - 1) \geq \frac{1}{4}$ or $c \geq \frac{5}{4}$

We had a lower bound of $\frac{5}{2}$, so we can choose $c = 3$.

Thus, $T(n) \in O(n \log n) \quad \forall n \geq 2$.

More on Complexity

- ▶ If an algorithm is composed of several parts, then its time complexity is the **sum** of the complexities of its parts.
- ▶ We must be able to *evaluate* these *summations*.
- ▶ Things become even more complicated when the algorithm contains loops, each iteration of which is a different complexity.

An Example

Example. Suppose $S_n = \sum_{i=1}^n i^2$

- ▶ We saw $\sum_{i=1}^n i = \frac{n(n+1)}{2} = \frac{(n^2+n)}{2} \leq n^2$, and is, in fact, $\Theta(n^2)$
- ▶ So, we guess $\sum_{i=1}^n i^2 \leq \sum_{i=1}^n n^2 = n^3$. Maybe $S_n \in \Theta(n^3)$.
- ▶ We can prove our guess correct, and find the minimum constant of difference between S_n and n^3 by induction:
- ▶ Guess: $\sum_{i=1}^n i^2 = an^3 + bn^2 + cn + d = P(n)$
- ▶ Notice that $\sum_{i=1}^{n+1} i^2 - \sum_{i=1}^n i^2 = (n+1)^2$

$$\text{So, } P(n+1) = P(n) + (n+1)^2$$

Thus,

$$a(n+1)^3 + b(n+1)^2 + c(n+1) + d =$$

$$a(n^3 + 3n^2 + 3n + 1) + b(n^2 + 2n + 1) + cn + c + d =$$

$$an^3 + 3an^2 + 3an + a + bn^2 + 2bn + b + cn + c + d =$$

$$an^3 + bn^2 + cn + d + n^2 + 2n + 1$$

$$\begin{aligned} \text{Hence, } 3an^2 + 3an + a + 2bn + b + c &= n^2 + 2n + 1, \text{ or} \\ 3an^2 + (3a + 2b)n + (a + b + c) &= n^2 + 2n + 1 \end{aligned}$$

Since coefficients of the same power of n must be equal:

$$3a = 1 \qquad (3a+2b) = 2 \qquad a + b + c = 1$$

$$a = \frac{1}{3} \qquad 3\left(\frac{1}{3}\right) + 2b = 2 \qquad \frac{1}{3} + \frac{1}{2} + c = 1$$

$$2b = 1 \qquad c = 1 - \frac{1}{3} - \frac{1}{2}$$

$$b = \frac{1}{2} \qquad c = \frac{1}{6}$$

And we can choose $d = 0$

Hence,

$$\begin{aligned}P(n) &= \frac{1}{3}(n^3) + \frac{1}{2}(n^2) + \frac{1}{6}(n) \\&= \frac{2}{6}(n^3) + \frac{3}{6}(n^2) + \frac{1}{6}(n) \\&= \frac{(2n^3+3n^2+n)}{6} \\&= \frac{n(2n^2+3n+1)}{6} \\&= \frac{n(n+1)(2n+1)}{6}\end{aligned}$$

Now, we wish to prove $S_n \in \Theta(n^3)$, or, that S_n is a third degree polynomial, by induction.

Base Case

Let $n = 1$

$$\text{lhs: } \sum_{i=1}^1 i^2 = 1^2 = 1$$

$$\text{rhs: } P(1) = \frac{1(1+1)(2(1)+1)}{6} = \frac{1(2)(3)}{6} = \frac{6}{6} = 1$$

Inductive Hypothesis

Assume for some arbitrary $n \geq 1$, that $S_n = P(n)$

$$\text{That is, } \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

Inductive Step

$$\text{Show } S_{n+1} = P(n+1) = \frac{(n+1)(n+2)(2n+3)}{6}$$

$$\begin{aligned} \text{lhs} = S_{n+1} &= S_n + (n+1)^2 && \text{defn of } \Sigma \\ &= \frac{n(n+1)(2n+1)}{6} + (n+1)^2 && \text{IH \& subst.} \\ &= \frac{n(n+1)(2n+1) + 6(n+1)^2}{6} && \text{Alg. Man.} \\ &= \frac{(n+1)[n(2n+1) + 6(n+1)]}{6} && \text{Alg. Man.} \\ &= \frac{(n+1)(2n^2 + n + 6n + 6)}{6} && \text{Alg. Man.} \\ &= \frac{(n+1)(2n^2 + 7n + 6)}{6} && \text{Alg. Man.} \\ &= \frac{(n+1)(n+2)(2n+3)}{6} = \text{rhs} && \text{Alg. Man.} \end{aligned}$$

Thus, $S_n = P(n) \forall n \geq 1$

Hence, $S_n \in \Theta(n^3)$

Section 3.5 — Program Correctness

- ▶ A brief introduction to the area of program verification, tying together the **rules of logic**, **proof techniques**, and the concept of an **algorithm**.
- ▶ **Program verification** means to prove the correctness of the program.
- ▶ Why is this important? Why can't we merely run testcases?
- ▶ A program is said to be **correct** if it produces the correct output for every possible input.

Correctness Proof

A correctness proof for a program consists of **two parts**:

1. Establish the **partial correctness** of the program.
If the program terminates, then it halts with the correct answer.
2. Show that the program **always terminates**.

Proving Output Correct

We need two propositions to determine what is meant by *produce the correct output*.

1. **Initial Assertion:** the properties the input values must have. (p)
2. **Final Assertion:** the properties the output of the program should have if the program did what was intended. (q)

A program (segment) S is said to be **partially correct with respect to p and q , $[p \{S\} q]$** , if — whenever p is TRUE for the input values of S *and* S *terminates*, —
then q is TRUE for the output values of S .

Example

```
p : x = 1           // initial assertion
    y = 2           // segment
    z = x + y       // S
q : z = 3           // final assertion
```

Is $[p \{S\} q]$ TRUE?

Composition Rule: $[p \{S_1\} q]$ and $[q \{S_2\} r] \rightarrow [p \{S_1; S_2\} r]$

Rules of Inference of Program Statements

For Conditional Statements:

IF condition **THEN** Block

BLOCK is executed when **condition** is TRUE, and it is not executed when **condition** is FALSE.

To verify correctness with respect to p and q , we must show:

1. When p is TRUE and **condition** is also TRUE, then q is TRUE after BLOCK terminates.
2. When p is TRUE and **condition** is FALSE, q is TRUE (since BLOCK does not execute).

This leads to the following rule of inference:

$$[(p \wedge \text{condition})\{Block\}q \text{ and } (p \wedge \neg\text{condition}) \rightarrow q] \\ \rightarrow p\{\text{if condition then Block}\}q$$

Example

`p : none`

`if x > y then y = x // Segment S`

`q : y >= x`

Is $[p \{S\} q]$ TRUE?

IF...THEN...ELSE statements

```
IF   condition   THEN  Block1
                                ELSE  Block2
```

If **CONDITION** is TRUE, then **Block1** executes; if **CONDITION** is FALSE, then **Block2** executes.

To verify correctness with respect to p and q , we must show:

1. When p is TRUE and **condition** is also TRUE, then q is TRUE after **BLOCK1** terminates.
2. When p is TRUE and **condition** is FALSE, q is TRUE after **BLOCK2** terminates.

This leads to the following rule of inference:

$$[(p \wedge \text{condition})\{Block1\}q \text{ and } (p \wedge \neg\text{condition})\{Block2\}q] \\ \rightarrow p\{\text{if condition then Block1 else Block2}\}q$$

Example

`p : none`

```
if x < 0 then abs = -x // Segment S
else abs = x
```

`q : abs = |x|`

Is $[p \{S\} q]$ TRUE?

I.e., is the segment correct?

Loop Invariants — While Loops

WHILE **condition** **Block**

Where **BLOCK** is repeatedly executed until **condition** becomes **FALSE**.

Loop Invariant: an assertion that remains **TRUE** **each** time **BLOCK** is executed.

I.e., p is a loop invariant if $(p \wedge \text{condition})\{\text{Block}\}p$ is **TRUE**

Let p be a loop invariant.

If p is **TRUE** before **Segment** S is executed, then p and **-condition** are **TRUE** after the loop terminates (if it does).

Hence: $(p \wedge \text{condition})\{S\}p$

$\therefore p\{\text{while condition } S\}(\neg\text{condition} \wedge p)$

Example

We wish to verify the following code segment terminates with `factorial = n!` when `n` is a positive integer.

Our loop invariant p is: `factorial = i!` and $i \leq n$

```
i = 1
factorial = 1
while i < n {
    i = i + 1
    factorial = factorial * i
}
```

[Base Case] p is TRUE before we enter the loop since `factorial = 1 = 1!`, and $1 \leq n$.

[Inductive Hypothesis] Assume for some arbitrary $n \geq 1$ that p is TRUE. Thus $i < n$ (so we enter the loop again), and `factorial = (i-1)!`.

[Induction Step] Show p is still TRUE after execution of the loop. Thus $i \leq n$ and `factorial = i!`.

First, i is incremented by 1

Thus $i \leq n$ since we assumed $i < n$, and i and $n \geq 1$.

Also, **factorial**, which was $(i - 1)!$ by IH, is set to $(i - 1)! * i = i!$

Hence, p remains true.

Since p remains TRUE, p is a loop invariant and thus the assertion:

$$[p \wedge (i < n)]\{S\}p \quad \text{is TRUE}$$

It follows that the assertion:

$$p\{\text{while } i < n \text{ } S\}[(i \geq n) \wedge p] \text{ is also true.}$$

Furthermore, the loop terminates after $n - 1$ iterations with $i = n$, since:

1. i is assigned the value 1 at the beginning of the program,
2. 1 is added to i during each iteration of the loop, and
3. the loop terminates when $i \geq n$

Thus, at termination, **factorial = n!**.

We split larger segments of code into component parts, and use the rule of composition to build the correctness proof.

$$(p = p_1)\{S_1\}q_1, \quad q_1\{S_2\}q_2, \quad \dots, \quad q_{n-1}\{S_n\}(q_n = q) \rightarrow \\ p\{S_1; S_2; \dots; S_n\}q$$