Exponents

Exponential with base $a$: Domain $= R$, Range $= R^+$

$$f(x) = a^x$$

- Note: $a$, the base, is fixed; $x$ varies
- You probably know: $a^n = a \times \cdots \times a$ ($n$ times)

How do we define $f(x)$ if $x$ is not a positive integer?

- **Want:** (1) $a^{x+y} = a^x a^y$; (2) $a^0 = a$

This means

- $a^2 = a^{1+1} = a^1 a^1 = a \times a$
- $a^3 = a^{2+1} = a^2 a^1 = a \times a \times a$
- 
- $a^n = a \times \cdots \times a$ ($n$ times)

We get more:

- $a = a^1 = a^{1+0} = a \times a^0$
  - Therefore $a^0 = 1$
- $1 = a^0 = a^{b + (-b)} = a^b \times a^{-b}$
  - Therefore $a^{-b} = 1/a^b$

Logarithms

Logarithm base $a$: Domain $= R^+$; Range $= R$

$$y = \log_a(x) \iff a^y = x$$

- $\log_2(8) = 3$; $\log_2(16) = 4$; $3 < \log_2(15) < 4$

The key properties of the log function follow from those for the exponential:

1. $\log_a(1) = 0$ (because $a^0 = 1$)
2. $\log_a(a) = 1$ (because $a^1 = a$)
3. $\log_a(xy) = \log_a(x) + \log_a(y)$

**Proof:** Suppose $\log_a(x) = z_1$ and $\log_a(y) = z_2$.

Then $a^{z_1} = x$ and $a^{z_2} = y$.

Therefore $xy = a^{z_1} \times a^{z_2} = a^{z_1+z_2}$.

Thus $\log_a(xy) = z_1 + z_2 = \log_a(x) + \log_a(y)$.

4. $\log_a(x^r) = r \log_a(x)$
5. $\log_a(1/x) = - \log_a(x)$ (because $a^{-y} = 1/a^y$)
6. $\log_a(x) = \log_a(x)/\log_a(b)$

**Examples:**

- $\log_2(1/4) = - \log_2(4) = -2$.
- $\log_2(-4)$ undefined

\[
\begin{align*}
\log_2(2^{10}3^5) &= \log_2(2^{10}) + \log_2(3^5) \\
&= 10 \log_2(2) + 5 \log_2(3) \\
&= 10 + 5 \log_2(3)
\end{align*}
\]
Limit Properties of the Log Function

\[ \lim_{x \to \infty} \log(x) = \infty \]
\[ \lim_{x \to 0} \frac{\log(x)}{x} = 0 \]

As \( x \) gets large \( \log(x) \) grows without bound.

But \( x \) grows MUCH faster than \( \log(x) \).

In fact, \( \lim_{x \to \infty} (\log(x)^m)/x = 0 \)

Why Rates of Growth Matter

Suppose you want to design an algorithm to do sorting.

- The naive algorithm takes time \( n^2/4 \) on average to sort \( n \) items
- A more sophisticated algorithm times time \( 2n \log(n) \)

Which is better?

\[ \lim_{n \to \infty} \left( \frac{2n \log(n)}{(n^2/4)} \right) = \lim_{n \to \infty} (8 \log(n)/n) = 0 \]

For example,

- if \( n = 1,000,000 \), \( 2n \log(n) = 40,000,000 \) — this is doable
  \( n^2/4 = 250,000,000,000 \) — this is not doable

Algorithms that take exponential time are hopeless on large datasets.

Polynomials

\[ f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_kx^k \]

is a polynomial function.

- \( a_0, \ldots, a_k \) are the coefficients

You need to know how to multiply polynomials:

\[
\begin{align*}
(2x^3 + 3x)(x^2 + 3x + 1) &= 2x^5 + 6x^4 + 2x^3 + 3x^2 + 9x + 3x \\
&= 2x^5 + 6x^4 + 5x^3 + 9x^2 + 3x
\end{align*}
\]

Exponentials grow MUCH faster than polynomials:

\[ \lim_{x \to \infty} \frac{a_0 + \cdots + a_kx^k}{b^x} = 0 \text{ if } b > 1 \]

Sum and Product Notation

\[
\begin{align*}
\sum_{i=0}^{k} a_i x^i &= a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k \\
\sum_{i=2}^{5} i^2 &= 2^2 + 3^2 + 4^2 + 5^2 = 54
\end{align*}
\]

Can limit the set of values taken on by the index \( i \):

\[
\sum_{\{i: 2 \leq i \leq k, \text{ even}\}} a_i = a_2 + a_4 + a_6 + a_8
\]

Can have double sums:

\[
\begin{align*}
\sum_{j=0}^{3} \sum_{i=0}^{2} a_{ij} &= \sum_{j=0}^{3} (\sum_{i=0}^{2} a_{ij}) \\
&= \sum_{j=0}^{3} a_{1j} + \sum_{j=0}^{3} a_{2j} \\
&= a_{10} + a_{11} + a_{12} + a_{13} + a_{20} + a_{21} + a_{22} + a_{23}
\end{align*}
\]

Product notation similar:

\[ \prod_{i=0}^{k} a_i = a_0 a_1 \cdots a_k \]
Changing the Limits of Summation

This is like changing the limits of integration.
\[ \sum_{i=0}^{n+1} a_i = \sum_{i=0}^{n} a_{i+1} = a_1 \cdots + a_{n+1} \]

Steps:
\begin{itemize}
  \item Start with \( \sum_{i=0}^{n+1} a_i \).
  \item Let \( j = i - 1 \). Thus, \( i = j + 1 \).
  \item Rewrite limits in terms of \( j \): \( i = 1 \rightarrow j = 0 \);
  \hspace{1cm} i = n + 1 \rightarrow j = n \)
  \item Rewrite body in terms of \( a_i \rightarrow a_{j+1} \)
  \item Get \( \sum_{j=0}^{n} a_{j+1} \)
  \item Now replace \( j \) by \( i \) (\( j \) is a dummy variable). Get
  \[ \sum_{i=1}^{n} a_i \]
\end{itemize}

Matrix Algebra

An \( m \times n \) matrix is a two-dimensional array of numbers, with \( m \) rows and \( n \) columns:
\[
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m1} & a_{m2} & \cdots & a_{mn}
\end{bmatrix}
\]

\begin{itemize}
  \item A \( 1 \times n \) matrix \( [a_1 \ldots a_n] \) is a row vector.
  \item An \( m \times 1 \) matrix is a column vector.
\end{itemize}

We can add two \( m \times n \) matrices:
\begin{itemize}
  \item If \( A = [a_{ij}] \) and \( B = [b_{ij}] \), then \( A + B = [a_{ij} + b_{ij}] \),
  \[ \begin{bmatrix}2 & 3 \\ 5 & 7\end{bmatrix} + \begin{bmatrix}3 & 7 \\ 4 & 2\end{bmatrix} = \begin{bmatrix}5 & 10 \\ 9 & 9\end{bmatrix} \]
\end{itemize}

Another important operation: transposition.
\begin{itemize}
  \item If we transpose an \( m \times n \) matrix, we get an \( n \times m \) matrix by switching the rows and columns.
  \[ \begin{bmatrix}2 & 3 & 9 \\ 5 & 7 & 12\end{bmatrix}^T = \begin{bmatrix}2 & 5 \\ 3 & 7 \\ 9 & 12\end{bmatrix} \]
\end{itemize}

Matrix Multiplication

Given two vectors \( \vec{a} = [a_1, \ldots, a_k] \) and \( \vec{b} = [b_1, \ldots, b_k] \),
their inner product (or dot product) is
\[ \vec{a} \cdot \vec{b} = \sum_{i=1}^{k} a_i b_i \]
\[ [1, 2, 3] \cdot [-2, 4, 6] = (1 \times -2) + (2 \times 4) + (3 \times 6) = 24. \]

We can multiply an \( n \times m \) matrix \( A = [a_{ij}] \) by an \( m \times k \) matrix \( B = [b_{ij}] \), to get an \( n \times k \) matrix \( C = [c_{ij}] \):
\begin{itemize}
  \item \( c_{ij} = \sum_{i=0}^{m-1} a_{ij} b_{ij} \)
  \item this is the inner product of the \( i \)th row of \( A \) with
  the \( j \)th column of \( B \)
\end{itemize}

\begin{align*}
  \begin{bmatrix}2 & 3 & 1 \\ 5 & 7 & 4\end{bmatrix} \times \begin{bmatrix}3 & 7 \\ 4 & 2 \\ -1 & -2\end{bmatrix} &= \begin{bmatrix}17 & 18 \\ 39 & 41\end{bmatrix} \\
  17 &= (2 \times 3) + (3 \times 4) + (1 \times -1) \\
  18 &= (2 \times 7) + (3 \times 2) + (1 \times -2) \\
  39 &= (5 \times 3) + (7 \times 4) + (4 \times -1) \\
  41 &= (5 \times 7) + (7 \times 2) + (4 \times -2)
\end{align*}
Why is multiplication defined in this strange way?

- Because it’s useful!

Suppose

\[
\begin{align*}
z_1 & = 2y_1 + 3y_2 + y_3, \quad y_1 = 3x_1 + 7x_2, \\
z_2 & = 5y_1 + 4y_2 + 4y_3, \quad y_2 = 4x_1 + 2x_2, \\
y_3 & = -x_1 - 2x_2.
\end{align*}
\]

Thus, \[
\begin{bmatrix}
z_1 \\
z_2
\end{bmatrix} = \begin{bmatrix}
2 & 3 & 1 \\
5 & 7 & 4
\end{bmatrix} \begin{bmatrix}
y_1 \\
y_2 \\
y_3
\end{bmatrix} \quad \text{and} \quad \begin{bmatrix}
y_1 \\
y_2 \\
y_3
\end{bmatrix} = \begin{bmatrix}
3 & 7 \\
4 & 2 \\
-1 & -2
\end{bmatrix} \begin{bmatrix}
x_1 \\
x_2
\end{bmatrix}.
\]

Suppose we want to express the \( z \)'s in terms of the \( x \)'s:

\[
z_1 = 2y_1 + 3y_2 + y_3 \\
= 2(3x_1 + 7x_2) + 3(4x_1 + 2x_2) + (-x_1 - 2x_2) \\
= (2 \times 3 + 3 \times 4 + (-1))x_1 + (2 \times 7 + 3 \times 2 + (-2))x_2 \\
= 17x_1 + 18x_2
\]

Similarly, \( z_2 = 39x_1 + 41x_2 \).

\[
\begin{bmatrix}
z_1 \\
z_2
\end{bmatrix} = \begin{bmatrix}
2 & 3 & 1 \\
5 & 7 & 4
\end{bmatrix} \begin{bmatrix}
3 & 7 \\
4 & 2 \\
-1 & -2
\end{bmatrix} \begin{bmatrix}
x_1 \\
x_2
\end{bmatrix}.
\]

---

### Logic Concepts

The most common mathematical argument is an implication.

- If \( x = 2 \) then \( x^2 = 4 \)

The implication is sometimes not as obvious:

- \( x^2 = 4 \) if \( x = 2 \)
- \( x^2 = 4 \) when \( x = 2 \)
- \( x = 2 \) implies \( x^2 = 4 \)
- Suppose \( x = 2 \). Then \( x^2 = 4 \).
- Whenever \( x = 2 \), \( x^2 = 4 \)
- \( x = 2 \) only if \( x^2 = 4 \)
- The condition \( x = 2 \) is sufficient for \( x^2 = 4 \)
- The condition \( x^2 = 4 \) is necessary for \( x = 2 \)

Note that the order of \( x = 2 \) and \( x^2 = 4 \) change.

We denote the implication “If \( A \) then \( B \)” by

\[ A \Rightarrow B \]

YOU NEED TO LEARN TO RECOGNIZE IMPLICATIONS.

---

### Equivalence

If both \( A \Rightarrow B \) and \( B \Rightarrow A \) are true, we write:

\[ A \iff B \]

\( A \) is equivalent to \( B \) (\( A \) if and only if \( B \); \( A \) iff \( B \))

\[ (A \Rightarrow B) \iff (\neg B \Rightarrow \neg A) \]

\( S \) is a square if and only if \( S \) is both a rectangle and a rhombus.

- \( S \) being a rectangle and a rhombus is sufficient for \( S \) to be a square
- \( S \) being a rectangle and a rhombus is necessary for \( S \) to be a square
Quantifiers

Quantifiers are words like every, all, some:
  • Every prime other than two is odd
  • Some real numbers are not integers
Any is ambiguous: sometimes it means every, and sometimes it means some
  • Anybody knows that 1 + 1 = 2
  • He’d be happy to get an A in any course
Avoid any. use every (= all) or some.

Negation

The negation of A, written \( \neg A \), is true exactly if A is false:
  • The negation of \( x = 2 \) is \( x \neq 2 \)
Be careful when negating quantifiers!
  • What is the negation of \( A = \) “Some of John’s answers are correct”
  • Is it \( B = \) “Some of John’s answers are not correct”
    • No! A and B can be simultaneously true
  • It’s “All of John’s answers are incorrect”.

Algorithms

An algorithm is a recipe for solving a problem.
In the book, a particular language is used for describing algorithms.
  • You need to learn the language well enough to read the examples
  • You need to learn to express your solution to a problem algorithmically and unambiguously
  • YOU DO NOT NEED TO LEARN IN DETAIL ALL THE IDIOSYNCRACIES OF THE PARTICULAR LANGUAGE USED IN THE BOOK.
    • You will not be tested on it, nor will most of the questions in homework use it

Main Features of the Language

  • Assignment statements
    • \( x \leftarrow 3 \)
  • if . . . then . . . else statements
    • if \( x = 3 \) then \( y \leftarrow y + 1 \) else \( y \leftarrow z \) endif
    • \( x = 3 \) is a test or predicate; it evaluates to either true or false
  • Selection statement
    
    if \( B_1 \) then \( S_1 \)
    \( B_2 \) then \( S_2 \)
    \( ! \)
    \( B_k \) then \( S_k \)
    [else \( S_{k+1} \)]
    endif
**Iteration**

Lots of variants:
- repeat until \( B \)
  \( S \)
endrepeat
or
- repeat
  \( S \)
endrepeat when \( B \)
or
- repeat while \( B \)
  \( S \)
endrepeat
(Same as while \( B \) do \( S \))
or
- for \( C = 1 \) to \( n \)
  \( S \)
endfor

**Input and Output**

Programs start with input statements of the form:

**Input** \( x, a_0, \ldots, a_k \)

- the values of the variables \( x, a_0, \ldots, a_k \) are assumed to be available at the beginning of the program

Programs end with output statements of the form:

**Output** \( P \)

**Example**

**Input** \( a_0, a_1, \ldots, a_n, x \)

\[
P \leftarrow a_n \\
\text{for } k = 1 \text{ to } n \\
P \leftarrow Px + a_{n-k}
\]

**Output** \( P \)

What does this compute?

---

**The Euclidean Algorithm**

The greatest common divisor of two natural numbers is the largest positive integer that divides both.

- \( \gcd(12, 15) = 3; \gcd(34, 51) = 17; \gcd(6, 45) = 3 \)
- By convention, \( \gcd(n, 0) = n \).

There is a method for calculating the gcd that goes back to Euclid:

- **Key observation**: if \( n > m \) and \( q \) divides both \( n \) and \( m \), then \( q \) divides \( n - m \) and \( n + m \).
  - Proof: If \( n = aq \) and \( m = bq \) then \( n - m = (a - b)q \) and \( n + m = (a + b)q \).

Therefore \( \gcd(n, m) = \gcd(m, n - m) \).

- Proof: Show that \( q \) divides both \( n \) and \( m \) iff \( q \) divides both \( m \) and \( n - m \). (If \( q \) divides \( n \) and \( m \), then \( q \) divides \( n - m \) by the argument above. If \( q \) divides \( m \) and \( n - m \), then \( q \) divides \( m + (n - m) = n \).
  - This allows us to reduce the gcd computation to a simpler case.

We can do even better:

- \( \gcd(n, m) = \gcd(m, n - m) = \gcd(m, n - 2m) = \ldots \)
- keep going as long as \( n - qm \geq 0 \) — \( \lceil n/m \rceil \) steps

Going back to \( \gcd(6, 45) \):

- \( \lceil 45/6 \rceil = 7; \) remainder \((45 \mod 6)\) is \( 3 \)
- \( \gcd(6, 45) = \gcd(6, 45 - 7 \times 6) = \gcd(6, 3) = 3 \)
We can keep this up this procedure to compute $\text{gcd}(n_1, n_2)$:

- If $n_1 \geq n_2$, write $n_1$ as $q_1n_2 + r_1$, where $0 \leq r_1 < n_2$
  - $q_1 = \lfloor n_1/n_2 \rfloor$
  - $\text{gcd}(n_1, n_2) = \text{gcd}(r_1, n_2)$
- Now $r_1 < n_2$, so switch their roles:
  - $n_2 = qr_1 + r_2$, where $0 \leq r_2 < r_1$
  - $\text{gcd}(r_1, n_2) = \text{gcd}(r_1, r_2)$
- Notice that $\max(n_1, n_2) > \max(r_1, n_2) > \max(r_1, r_2)$
- Keep going until we have a remainder of 0 (i.e., something of the form $\text{gcd}(r_k, 0)$ or $(\text{gcd}(0, r_k))$
  - This is bound to happen sooner or later

An algorithm for $\text{gcd}$

**Input** $m, n$  
$m, n$ natural numbers, $m \geq n$

$num \leftarrow m; \quad denom \leftarrow n$  
[Initialize num and denom]

**repeat**  
until $\text{denom} = 0$

$q \leftarrow \lfloor \text{num}/\text{denom} \rfloor$

$rem \leftarrow \text{num} - (q \times \text{denom})$  
[rem = num mod denom]

$num \leftarrow \text{denom}$

$denom \leftarrow \text{rem}$  
[New num; note num $\geq$ denom]

**endrepeat**

**Output** $\text{num}$  
[num = gcd(m, n)]

Example: $\text{gcd}(84, 33)$

Iteration 1: $\text{num} = 84, \text{denom} = 33, q = 2, \text{rem} = 18$

Iteration 2: $\text{num} = 33, \text{denom} = 18, q = 1, \text{rem} = 15$

Iteration 3: $\text{num} = 18, \text{denom} = 15, q = 1, \text{rem} = 3$

Iteration 4: $\text{num} = 15, \text{denom} = 3, q = 5, \text{rem} = 0$

Iteration 5: $\text{num} = 3, \text{denom} = 0 \Rightarrow \text{gcd}(84, 33) = 3$

How do we know this works?

- We have two loop invariants, which are true each time we start the loop:
  - $\text{gcd}(m, n) = \text{gcd}(\text{num}, \text{denom})$
  - $\text{num} \geq \text{denom}$
- At the end, $\text{denom} = 0$, so $\text{gcd}(\text{num}, \text{denom}) = \text{num}$

Procedure Calls

It is useful to extend our algorithmic language to have procedures that we can call repeatedly. For example, we may want to have a procedure for computing gcd or factorial, that we can call with different arguments. Here’s the notation used in the book:

**procedure** Name(variable list)  
procedure body (includes a **return** statement)

**endpro**

- The **return** statement returns control to the portion of the algorithm from where the procedure was called

Example:

**procedure** Factorial(n)  

$\text{fact} \leftarrow 1$

$m \leftarrow n$

**repeat**  
**until** $m = 1$

$\text{fact} \leftarrow \text{fact} \times m$

$m \leftarrow m - 1$

**endrepeat**

**return** $\text{fact}$

**endpro**
Recursion

Recursion occurs when a procedure calls itself.

Example: A recursive procedure for computing gcd

procedure gcd-rec(i, j)
    if j = 0 then answer ← i
    else gcd-rec(j, i − ⌊i/j⌋j)
    endif
    return answer
endpro
gcd-rec(m, n)

To compute gcd-rec(84, 33), we call
• gcd-rec(33, 18)
• gcd-rec(18, 15)
• gcd-rec(15, 3)
• gcd-rec(3, 0)

How do we know that the chain of recursive calls is finite?
• Same reasoning as before

Towers of Hanoi

Problem: Move all the rings from pole 1 and pole 2, moving one ring at a time, and never having a larger ring on top of a smaller one.

How do we solve this?
• Think recursively!
• Suppose you could solve it for n − 1 rings? How could you do it for n?

Solution

• Move top n − 1 rings from pole 1 to pole 3 (we can do this by assumption)
  • Pretend largest ring isn’t there at all
• Move largest ring from pole 1 to pole 2
• Move top n − 1 rings from pole 3 to pole 2 (we can do this by assumption)
  • Again, pretend largest ring isn’t there

This solution translates to a recursive algorithm:
• Suppose robot(r → s) is a command to a robot to move the top ring on pole r to pole s
• Note that if r, s ∈ {1, 2, 3}, then 6 − r − s is the other number in the set

procedure H(n, r, s)  [Move n disks from r to s]
    if n = 1 then robot(r → s)
    else H(n − 1, r, 6 − r − s)  robot(r → s)  H(n − 1, 6 − r − s, s)
    endif
    return
endpro

Tree of Calls

Suppose there are initially three rings on pole 1, which we want to move to pole 2:
Analysis of Algorithms

For a particular algorithm, we want to know:

- How much time it takes
- How much space it takes

What does that mean?

- In general, the time/space will depend on the input size
  - The more items you have to sort, the longer it will take

- Therefore want the answer as a function of the input size
  - What is the best/worst/average case as a function of the input size.

Given an algorithm to solve a problem, may want to know if you can do better.

- What is the intrinsic complexity of a problem?

This is what computational complexity is about.