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) $a^1 = a$

This means

• $a^2 = a^{1+1} = a^1 a^1 = a \times a$

• $a^3 = a^{2+1} = a^2 a^1 = a \times a \times a$

• …

• $a^n = a \times \ldots \times a$ ($n$ 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 $a^{-b} = 1/a^b$
\[ a = a^1 = a^{\frac{1}{2} + \frac{1}{2}} = a^{\frac{1}{2}} \times a^{\frac{1}{2}} = (a^{\frac{1}{2}})^2 \]

\( \circ \) Therefore \( a^{\frac{1}{2}} = \sqrt{a} \)

\[ a^{\frac{1}{k}} = \sqrt[k]{a} \]

• Similar arguments show that \( a^{\frac{1}{k}} = \sqrt[k]{a} \)

• \( a^{mx} = a^x \times \cdots \times a^x \) (\( m \) times) = \( (a^x)^m \)

\( \circ \) Thus, \( a^{\frac{m}{n}} = (a^{\frac{1}{n}})^m = (\sqrt[n]{a})^m. \)

This determines \( a^x \) for all \( x \) rational. The rest follows by continuity.
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. $\log_b(x) = \log_a(x) / \log_a(b)$
Examples:

- \( \log_2(1/4) = -\log_2(4) = -2 \).
- \( \log_2(-4) \) undefined
- \[
\log_2(2^{10}3^5) \\
= \log_2(2^{10}) + \log_2(3^5) \\
= 10 \log_2(2) + 5 \log_2(3) \\
= 10 + 5 \log_2(3)
\]
Limit Properties of the Log Function

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

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

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

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

\[ f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k \] is a polynomial function.

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

You need to know how to multiply polynomials:

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

Exponentials grow MUCH faster than polynomials:

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

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

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.
Sum and Product Notation

\[ \sum_{i=0}^{k} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k \]

\[ \sum_{i=2}^{5} i^2 = 2^2 + 3^2 + 4^2 + 5^2 = 54 \]

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

\[ \sum_{\{i:2 \leq i \leq 8 \mid i \text{ even}\}} a_i = a_2 + a_4 + a_6 + a_8 \]

Can have double sums:

\[ \sum_{i=1}^{2} \sum_{j=0}^{3} a_{ij} \]
\[ = \sum_{i=1}^{2} \left( \sum_{j=0}^{3} a_{ij} \right) \]
\[ = \sum_{j=0}^{3} a_{1j} + \sum_{j=0}^{3} a_{2j} \]
\[ = a_{10} + a_{11} + a_{12} + a_{13} + a_{20} + a_{21} + a_{22} + a_{23} \]

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.

- $\sum_{i=1}^{n+1} a_i = \sum_{i=0}^{n} a_{i+1} = a_1 + \cdots + a_{n+1}$

Steps:

- Start with $\sum_{i=1}^{n+1} a_i$.
- Let $j = i - 1$. Thus, $i = j + 1$.
- Rewrite limits in terms of $j$: $i = 1 \rightarrow j = 0$; $i = n + 1 \rightarrow j = n$
- Rewrite body in terms of $a_i \rightarrow a_{j+1}$
- Get $\sum_{j=0}^{n} a_{j+1}$
- Now replace $j$ by $i$ ($j$ is a dummy variable). Get $\sum_{i=0}^{n} a_{i+1}$
Matrix Algebra

An $m \times n$ matrix is a two-dimensional array of numbers, with $m$ rows and $n$ columns:

$$
\begin{bmatrix}
    a_{11} & a_{12} & \cdots & a_{1n} \\
    a_{21} & a_{22} & \cdots & a_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{m1} & a_{m2} & \cdots & a_{mn}
\end{bmatrix}
$$

- A $1 \times n$ matrix $[a_1 \ldots a_n]$ is a row vector.
- An $m \times 1$ matrix is a column vector.

We can add two $m \times n$ matrices:

- If $A = [a_{ij}]$ and $B = [b_{ij}]$ then $A + B = [a_{ij} + b_{ij}]$.

$$
\begin{bmatrix}
    2 & 3 \\
    5 & 7
\end{bmatrix} + 
\begin{bmatrix}
    3 & 7 \\
    4 & 2
\end{bmatrix} = 
\begin{bmatrix}
    5 & 10 \\
    9 & 9
\end{bmatrix}
$$

Another important operation: transposition.

- If we transpose an $m \times n$ matrix, we get an $n \times m$ matrix by switching the rows and columns.

$$
\begin{bmatrix}
    2 & 3 & 9 \\
    5 & 7 & 12
\end{bmatrix}^T = 
\begin{bmatrix}
    2 & 5 \\
    3 & 7 \\
    9 & 12
\end{bmatrix}
$$
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 \( C = [c_{ij}] \):

- \( c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \)
- this is the inner product of the \( i \)th row of \( A \) with the \( j \)th column of \( B \)
\[ \begin{bmatrix} 2 & 3 & 1 \\ 5 & 7 & 4 \end{bmatrix} \times \begin{bmatrix} 3 & 7 \\ 4 & 2 \\ -1 & -2 \end{bmatrix} = \begin{bmatrix} 17 & 18 \\ 39 & 41 \end{bmatrix} \]

17 = (2 \times 3) + (3 \times 4) + (1 \times -1) \\
= (2, 3, 1) \cdot (3, 4, -1)

18 = (2 \times 7) + (3 \times 2) + (1 \times -2) \\
= (2, 3, 1) \cdot (7, 2, -2)

39 = (5 \times 3) + (7 \times 4) + (4 \times -1) \\
= (5, 7, 4) \cdot (3, 4, -1)

41 = (5 \times 7) + (7 \times 2) + (4 \times -2) \\
= (5, 7, 4) \cdot (7, 2, -2)
Why is multiplication defined in this strange way?
• Because it’s useful!

Suppose

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

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

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

\[
\begin{align*}
z_1 &= 2y_1 + 3y_2 + y_3 \\
&= 2(3x_1 + 7x_2) + 3(4x_1 + 2x_2) + (-x_1 - 2x_2) \\
&= (2 \times 3 + 3 \times 4 + (-1))x_1 + (2 \times 7 + 3 \times 2 + (-2))x_2 \\
&= 17x_1 + 18x_2
\end{align*}
\]

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.
Implications chain:

- If $A \Rightarrow B$ and $B \Rightarrow C$ then $A \Rightarrow C$
- $((A \Rightarrow B) \land (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$

The converse of $A \Rightarrow B$ is $B \Rightarrow A$.

- They are not equivalent.
- $x = 2 \Rightarrow x^2 = 4$ is true; $x^2 = 4 \Rightarrow x = 2$ is not
  ($x$ could be $-2$)

The contrapositive of $A \Rightarrow B$ is $\neg B \Rightarrow \neg A$.

- $\neg$ stands for negation
- A statement is equivalent to its contrapositive.
- If $x^2 \neq 4$ then $x \neq 2$.
- If you’re asked to prove $A \Rightarrow B$, one way to do it
  (which is sometimes easier) is to show $\neg B \Rightarrow \neg A$
**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 $A = \text{“Some of John’s answers are correct”}$
- Is it $B = \text{“Some of John’s answers are not correct”}$
  - 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** $a_0, a_1, \ldots, a_n, x$

\[
P \leftarrow a_n
\]

**for** $k = 1$ **to** $n$

\[
P \leftarrow P x + a_{n-k}
\]

**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:
• \( \text{gcd}(n, m) = \text{gcd}(m, n - m) = \text{gcd}(m, n - 2m) = \ldots \)
• keep going as long as \( n - qm \geq 0 \) — \([n/m]\) steps

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

- If $n_1 \geq n_2$, write $n_1$ as $q_1n_2 + r_1$, where $0 \leq r_1 < n_2$
  
  - $q_1 = \lfloor n_1/n_2 \rfloor$

- $\gcd(n_1, n_2) = \gcd(r_1, n_2)$

- Now $r_1 < n_2$, so switch their roles:

- $n_2 = q_2r_1 + r_2$, where $0 \leq r_2 < r_1$

- $\gcd(r_1, n_2) = \gcd(r_1, r_2)$

- 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 $\gcd(r_k, 0)$ or $(\gcd(0, r_k))$

  - This is bound to happen sooner or later
An algorithm for gcd

**Input** \( m, n \) \([m, n \text{ natural numbers, } m \geq n]\)
\[\text{num} \leftarrow m; \text{denom} \leftarrow n \text{ [Initialize } \text{num} \text{ and } \text{denom}]\]

**repeat until** \( \text{denom} = 0 \)

\[q \leftarrow \lfloor \text{num}/\text{denom} \rfloor\]
\[\text{rem} \leftarrow \text{num} - (q \times \text{denom}) \text{ [rem} = \text{num mod den}om]\]
\[\text{num} \leftarrow \text{denom} \text{ [New } \text{num]}\]
\[\text{denom} \leftarrow \text{rem} \text{ [New } \text{denom}; \text{ note } \text{num} \geq \text{denom]}\]

**endrepeat**

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

**Example:** \( \gcd(84, 33) \)

**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:** \( \text{num} = 3, \text{denom} = 0 \Rightarrow \gcd(84, 33) = 3 \)
How do we know this works?

- We have two *loop invariants*, which are true each time we start the loop:
  - $\gcd(m, n) = \gcd(num, denom)$
  - $num \geq denom$
- At the end, $denom = 0$, so $\gcd(num, denom) = num$. 
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)
    fact ← 1
    m ← n
    repeat until m = 1
        fact ← fact × m
        m ← m − 1
    endrepeat
    return fact
endpro
```
Recursion

Recursion occurs when a procedure calls itself.

Example: A recursive procedure for computing gcd

```c
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 \( \text{robot}(r \rightarrow s) \) is a command to a robot to move the top ring on pole \( r \) to pole \( s \)
- Note that if \( r, s \in \{1, 2, 3\} \), then \( 6 - r - s \) is the other number in the set

```plaintext
procedure H(n, r, s)  [Move \( n \) disks from \( r \) to \( s \)]
    if \( n = 1 \) then robot(r \rightarrow s)
    else H(n - 1, r, 6 - r - s)
        robot(r \rightarrow 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.