What is Computer Algebra?

Computer algebra is a sub-field of mathematics and computer science that deals with the exact solution of equations.

Main Topics:

- Computing with integers, rationals and algebraic numbers
- Polynomials: Multiplication and factorization
- Solving polynomial equations: Gröbner bases and computational algebraic geometry
- Applications in cryptography, optimization and many other fields of computational science
Efficient algorithms, $O$-notation

Basic arithmetic $\times, \div, -, +$

Implementation in Python

Newton iteration and running-time equivalence of $\times, /$

Modular arithmetic, fast exponentiation

Randomized primality tests, distribution of primes, RSA

Chinese remainder theorem and computing determinants

The Schwartz-Zippel Lemma and perfect matchings in graphs

Matrix multiplication, Gaussian elimination and matrix inversion

Polynomials: Evaluation, interpolation and the Fast Fourier Transform (FFT), efficient multiplication

Symbolic FFT in rings

Lattices, Hermite-normal forms and integer linear algebra
You can collect bonus points by handing in solutions to selected exercises from the assignment sheets.

If you solve 50% or more of the exercises, the grade of your final exam will be improved by a half grade.

If you solve 90% or more of the exercises, the grade of your final exam will be improved by a full grade.

Condition: grade ≥ 4.0

Groups up to 3 people (e.g. submission) except for programming (e.g. II!)
Main literature

   
   *Use this for first 1/3 of class.*

2. *Modern Computer Algebra*, by J. von zur Gathen and J. Gerhard
Python to the level of need in this course is really easy and can be learned on the fly.

A very nice introduction is here:
http://cscircles.cemc.uwaterloo.ca/

disopt.epfl.ch  follow teaching link.
Analysis of Algorithms
Recall the definition of Fibonacci numbers

- $F_0 = 0, F_1 = 1$
- If $n \geq 2$: $F_n = F_{n-1} + F_{n-2}$

\[ F_2 = F_1 + F_0 = 1 \]
\[ F_3 = F_2 + F_1 = 2 \]
\[ F_4 = 3 \]
The bad

```
def fib1(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib1(n-1)+fib1(n-2)
```

\[ F_n = F_{n-1} + F_{n-2} \]

\[ \geq 2 \cdot F_{n-2} \]

\[ \geq 2^2 \cdot F_{n-4} \]

\[ \geq 2^{\frac{n-1}{2}} \]

F is nonmonotonously with \( n \).

Recursive calls.
def fib1(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib1(n-1) + fib1(n-2)

Analysis: Count basic arithmetic operations.

\[ T(n) := \# \text{ basic arithmetic operations.} \]

\[ n \geq 2: \]

\[ T(n) = T(n-1) + T(n-2) \]

\[ T(n) \geq 2^{\log_2 2} = \text{Exponential.} \]
def fib2(n):
    A = [0, 1]
    i = 1
    while i < n:
        A.append(A[i-1]+A[i])
        i = i+1
    return A[n]

n = 3.
Comparison of running times

- Algorithm 1: \( f_1(n) = n^2 \)
- Algorithm 2: \( f_2(n) = 2 \cdot n + 10 \)

\( f_2(x) \) will be dominated from some position.

Would like:

\( f_2 \leq f_1 \)
**O-notation**

**Definition**
Let \( f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \). We say \( f = O(g) \) if there exists a constant \( c > 0 \) and a number \( N_o \in \mathbb{N} \) such that

\[
f(n) \leq c \cdot g(n) \quad \text{for all } n \geq N_0.
\]

\( f(n) = 5n^3 + n \)
\( g(n) = n^4 \).

\( l : n^5, \ n \log n \)
\( m \approx n. \)

\( q : n^{1+\varepsilon}, \ \varepsilon > 0 \) fixed and small.

\( f = O(g) \) with \( N_0 = 1, c = g \)
\( g, n^3 + n \leq g \cdot n^4, \ \forall n \in \mathbb{N}. \)

\( h = O(g) \)

\( q, g, g = O(q) \).
Definition
Let \( f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \).

- We say \( f = \Omega(g) \) if \( g = O(f) \).
- We say \( f = \Theta(g) \) if \( f = O(g) \) and \( f = \Omega(g) \).

\[
8n^3 + 6 = \Theta\left(\frac{1}{2}n^3\right)
\]

If leading exponents of two polynomials \( p(x), q(x) : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \) is the same \( \Leftrightarrow q = \Theta(p) \).
Basic Arithmetic
Natural numbers

- $\mathbb{N} = \{1, 2, 3, \ldots\}$
- Bit-representation
  \[
  \langle 1, 0, 1, 0 \rangle = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4
  \]
  \[
  = 2^2 + 2^3 + 2^4
  \]
  \[
  = 26
  \]
- $\text{size}(a) = \lceil \log_2(a + 1) \rceil$, $a \in \mathbb{N}$.
- Representation in Python
  \[
  L = [0, 1, 0, 1, 1] \text{ represents number 26.}
  \]
  $\text{size}(a) = \text{Bits needed to represent } a$.
  \[
  \text{size}(a) = \lceil \log_2 (a+1) \rceil
  \]
Remember the old school days?

\[ \begin{array}{c}
\text{a} : 1010110 \\
+ \quad 0110101 \\
\hline 
\text{1000101} \\
\end{array} \]
Algorithm

```python
def Add(L1, L2):

    if len(L1) < len(L2):
        L1, L2 = L2, L1  # swaps the pointers to the two lists

    Out = []
    carry = 0
    n = max(len(L1), len(L2))

    for i in range(len(L1)):
        if i < len(L2):
            h = carry + L1[i] + L2[i]
        else:
            h = carry + L1[i]

        Out.append(h % 2)
        if h > 1:
            carry = 1
        else:
            carry = 0

    if carry == 1:
        Out.append(1)

    return Out
```

Analysis:

- Time Complexity: O(n)


```plaintext
Input: [0, 1, 1, 0, 3]
Output: [0, 0, 1, 0]
```

```plaintext
Input: [1, 0, 1, 0, 1]
Output: [0, 0, 1, 0, 1]
```
Theorem

Two $n$-bit numbers can be added in time $O(n)$. 
Exercise
Write a python function `subtract(L1, L2)` that returns the representation of \( \frac{L2 - L1}{L1 - L2} \) if \( L2 \geq L1 \) and \(-1\) if \( L2 < L1 \).
**Multiplication**

\[ \begin{array}{c}
  a & 11010 \\
  b & 01000 \\
  \hline \\
  11010 \\
  00000 \\
  1101000 \\
  \hline \\
  10100000 \\
  \end{array} \]

Add them up.

Add up ≤n times 2n-bit numbers.

Running time: \( O(n^2) \) \( \Omega(n^2) \)

2.12

\( \text{a.b on both n-bit numbers.} \)

How many bits does \( a \times b \) have?

Next week:

\( \text{O}(n \log_2 3) \)

even lew

\( \text{O}(n \log_2 n \log \log n) \)
function multiply(x, y)

    if y == 0:
        return 0
    z = multiply(x, [y/2])
    if y is even:
        return 2z
    else:
        return x + 2z

    z = x * \lceil y/2 \rceil

    \text{Case 1: } \lfloor y/2 \rfloor = \lfloor y/2 \rfloor \quad \text{← } x \text{ even.}
    \text{Case 2: } \lfloor y/2 \rfloor = \lfloor y/2 \rfloor - \frac{1}{2}

    \text{Out} = 2 \cdot z

    x \cdot y = x \cdot \left( \lfloor y/2 \rfloor \cdot 2 + 1 \right)
    = \lfloor y/2 \rfloor \cdot 2 + x
def Multiply(L1, L2):
    # condition: L2 does not represent 0
    if Leading1(L2) == -1:
        return [0]
    else:
        H = list(L2)
        b = H.pop(0)
        Z = Multiply(L1, H)
        Z.insert(0, 0)
        if b == 0:
            return Z
        else:
            return Add(Z, L1)
Theorem

Two \( n \)-bit integers can be multiplied in time \( O(n^2) \).
Multiplication faster than \( O(n^2) \).

**Idea:** Assume for now \( n \) (\# of bits) is a power of 2.

Possibly by adding additional 2's:

\[ a, b : n \text{- bit numbers.} \]

\[ a \times b = (a_1 \cdot 2^{n/2} + a_2)(b_1 \cdot 2^{n/2} + b_2) \]

\[ = (a_1 \cdot b_1 + a_2 \cdot b_2) \cdot 2^{n/2} + (a_1 b_2 + b_2 a_2) \cdot 2^{n/12} \]

\[ \leq n \text{ bits.} \]

\[ a = a_1 \cdot 2^{n/2} + a_2 \text{ and } b = b_1 \cdot 2^{n/2} + b_2 \text{ are } n/2 \text{- bit numbers.} \]

Running time:

\[ T(n) \geq 4 \cdot T(n/2) + c \cdot n \]

To be taken care by massive cells.
Running time:

For \( n \geq 1 \):

\[
T(n) \geq 4 \cdot T(n/2) + c \cdot n
\]

For now, assume \( c = 1 \).

\[
\Rightarrow 4 \cdot \left( 4 \cdot T(n/2^2) + \frac{n}{2} \right) + n
\]

\[
\Rightarrow \left( 4^i \right) \cdot T(1) + \ldots + \frac{n}{2^i} \quad \text{for } i = \log_2 n
\]

\[
4^{\log_2 n} = n^2
\]
The Karatsuba Trick:

\[ \begin{align*}
A \ast b &= (a_1 \cdot 2^{n/2} + a_2) (b_1 \cdot 2^{n/2} + b_2) \\
&= a_1 b_1 \cdot 2^n + (a_1 b_2 + a_2 b_1) \\
&+ a_2 b_2 \\
&= (a_1 + a_2) (b_1 + b_2) \\
&= a_1 b_1 + a_1 b_2 + a_2 b_1 + a_2 b_2
\end{align*} \]

Compute recursively.

- \( a_1 b_1 \)
- \( a_1 b_2 \)
- \( a_2 b_1 \)
- \( a_2 b_2 \)

Running time: \( T(n) \leq 3 \cdot T(n/2) + C \cdot n \) for some constant \( C > 0 \).
Running time: \( T(n) \leq 3 \cdot T(n/2) + C \cdot n \) for some constant \( C > 0 \).

\[
T(n) \leq 3 \left( 3 \cdot T(n/2^2) + C \cdot \frac{n}{2} \right) + C \cdot n
\]

\[
= 3^2 \cdot T(n/2^3) + 3 \cdot C \cdot \frac{n}{2} + C \cdot n
\]

\[
\leq 3^2 \left( 3 \cdot T(n/2^3) + C \cdot \frac{n}{2^3} \right) + C \cdot 3 \cdot \frac{n}{2} + C \cdot n
\]

\[
\leq 3^0 \cdot T(1) + C \cdot n \left( \frac{3}{2} \right) + C \cdot n \frac{3}{2} + \cdots + C \cdot n \left( \frac{3}{2^0} \right)^0
\]

\[
= n^{\frac{\log_2 3}{\log_2 n}} + C \cdot n \cdot \sum_{i=0}^{\log_2 n - 1} \left( \frac{3}{2} \right)^i = n^{\frac{\log_2 3}{\log_2 n}} + C \cdot n \cdot \frac{(3/2)^{\log_2 n} - 1}{3/2 - 1}
\]

\[
\leq n^{\frac{\log_2 3}{\log_2 n}} + C \cdot n \cdot \frac{3^{\log_2 n}}{3/2 - 1}
\]

\[
= n^{\log_2 3} + C \cdot n \cdot \frac{\log_2 (3/2)}{n^{\log_2 3}}
\]

\[
= O(n \log_2 n)
\]

\( \log_2 3 < 2. \)