Computer Algebra

- Fritz Eisenbrand 2008 at EPFL
- Alfonso Cevallos Manzano.

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: large

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

Syllabus

- Basic arithmetic *,/,-,+
- Implementation in *Python*
- Modular arithmetic, fast exponentiation $(\mathbb{Z}_N, +, \cdot)$
- 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

Bonus rule

(*)

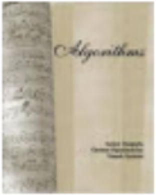
- You can collect bonus points by handing in solutions to selected exercises from the assignment sheets.
- If you solve 50% or more of the exercises, the grade of your final exam will be improved by a half grade.
- If you solve 90% or more of the exercises, the grade of your final exam will be improved by a full grade.

provided that your grade in final exam ≥ 4.0

Main literature

1. Any book with the title *Algebra* Groups, Rings & Fields
2. *Algorithms*, by S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani

first part of course (1/3)

Algorithms for integers, Fast Fourier Transform.... →

3. *Modern Computer Algebra*, by J. von zur Gathen and J. Gerhard

disopt.epfl.ch. Follow teaching link.

Python

- 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/`

Analysis of Algorithms

Algorithms: The good, the bad . . .

Recall the definition of Fibonacci numbers

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

$F_2 = F_1 + F_0 = 1 + 0 = 1$

$F_3 = 1 + 1 = 2$, $F_4 = 2 + 1$ ....

- F_N is monotonically increasing.
- $F_N = \underbrace{F_{N-1}}_{\geq F_{N-2}} + F_{N-2} \geq 2 \cdot F_{N-2} \geq 4 \cdot F_{N-4} \geq 8 \cdot F_{N-6}$

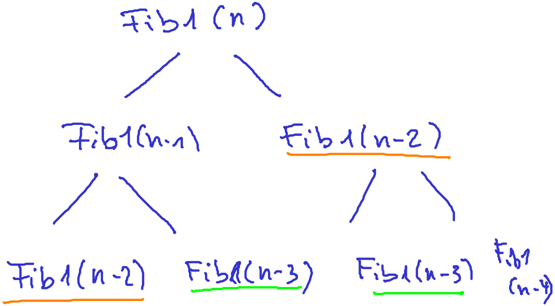
exponentially increasing.

$F_N \geq 2^{\lfloor N/2 \rfloor} \cdot F_1$

- exponentially increasing sequence

The bad

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



Let $T(n)$ be the number of basic operations that are performed by Fib1(n)

$T(n) = 1$ $n \leq 2$

$T(n) = T(n-1) + T(n-2)$ ← looks like definition of Fibonacci numbers themselves.

⇒ Alg. performs exponential number of basic operations ($2^{\lfloor n/2 \rfloor}$)

The good

```
def fib2(n):
    A = [0,1]
    i = 1
    while i < n:
        A.append(A[i-1]+A[i])
        i = i+1
    return A[n]
```

<u>Thm:</u> This algorithm performs a linear (in N) number of basic operations. $O(N)$ basic operations.

O- Notation

describes growth of functions.

Analysis

Comparison of running times

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

$\boxed{\text{Algorithm 1.}}$ ← input of "length" N

↪ performs $F_1(n)$ basic op.

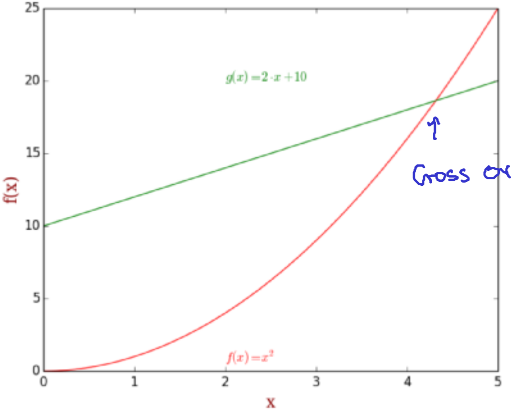
$\boxed{\text{Alg2.}}$ ← Input of "length" N.

↪ performs $f_2(n)$ basic op.

$f_2 = O(f_1)$

But

$f_1 \neq O(f_2)$



↑ Cross overs.

O-notation

Definition

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

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

Previous example.

$C = 1$

$N_0 = 5$

$1 \cdot f_1(N) \geq f_2(N)$

$f(n) = 3n^3 + n^2 + 1$

$g(n) = n^4 + n + 1.$

$\Rightarrow \quad f = O(g).$

O-notation

Definition

Let $f, g : \mathbb{N} \longrightarrow \mathbb{R}_{\geq 0}$.

- We say $f = \underline{\Omega(g)}$ if $g = O(f)$.
- We say $f \underline{= \Theta(g)}$ if $f = O(g)$ and $f = \Omega(g)$. <-- f and g have same growth-rate if $f = O(g)$ and $g = O(f)$.

Primary Goal of Algorithm design:

- Find efficient algorithms. For example Alg A has running time $f(n)$. Goal: design Algorithm B with running time $g(n)$ and $g(n) = O(f(n))$ but $f(n) \neq O(g(n))$
- Find lower bounds for Algorithms. $\Rightarrow$ Not so much success yet!

Basic Arithmetic

Natural numbers

$\mathbb{N} = \{0, 1, 2, \dots\}, \mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$.

A *natural number* is represented by a list of bits

$$\langle a_{n-1}, a_{n-2}, \dots, a_0 \rangle, \text{ with } a_i \in \{0, 1\},\ i = 0, \dots, n-1.$$

Represented number

$$x = \sum_{i=0}^{n-1} a_i \cdot 2^i.$$

$3 = \langle 1, 1 \rangle$

$10 = \langle 1, 0, 1, 0 \rangle$

x has *n bits* and a_i is the *i*-th bit.

An *integer* $x \in \mathbb{Z}$ can be represented by its abs. value and a bit that determines the sign of the integer.

$\underline{\text{size}(x) = \lceil \log_2(|x| + 1) \rceil}$ Reflects number of bits of the representation of x.

≈ Input length.

Addition

$$\begin{array}{cccccccccc}
 & & \downarrow & & & & & & & \\
 & & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
+ & & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
 & 1 & 1 & 1 & & & & & & \\
\hline
 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
\end{array}$$

$C = 0$

Representation of Sum!

Algorithm

```
def add(u, v):
        b = []  ← Result.
        j = 0
        carry = 0
        while j < len(u) or j < len(v) or carry:
                if j < len(u):
                        carry += u[j]
                if j < len(v):
                        carry += v[j]
                b += [carry % 2]
                carry /= 2
                j += 1
        return b
```

Analysis

Theorem

Two n-bit numbers can be added in time $O(n)$.

Optimal, since all bits of input have to be read!

Subtraction

Exercise

Write a python function `Subtract(L1,L2)` that returns the representation of $L1 - L2$ if $L1 \geq L2$ and -1 if $L1 < L2$.

$(1\ 1\ 0\ 1\ 1) - (0, 1, 0, 1, 1)$

$$\begin{array}{rccccc}
 & 0 & 1 & 0 & 1 & 1 \\
+ & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} \\
\hline
 & 1 & 1 & 0 & 1 & 1
\end{array}$$

Can be done in O(n).

Multiplication

lowest-bit

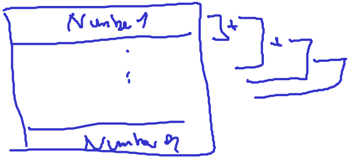
$$\boxed{\langle 1,0,1,1,0\rangle * \langle \underline{1},0,0,\underline{1},0\rangle}$$

$$a \cdot b = a \cdot \sum_{i=0}^{n-1} 2^i \cdot b_i$$

if $b = \langle b_{n-1}, \dots, b_0 \rangle$

$$= \sum_{i=0}^{n-1} \underline{b_i} \cdot a \cdot 2^i$$

$$\begin{array}{r} 1\,0\,1\,1\,0\,\textcircled{0} \\ 1\,0\,1\,1\,0\,\textcircled{0}\,\textcircled{0}\,\textcircled{0}\,\textcircled{0} \\ \hline \end{array}$$



In the end:

$O(n)$ additions of 2n-bit numbers.

running time. $O(n^2)$

Multiplication

```
function multiply(x,y)

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

Running time $O(n^2)$

Input Numbers have n bits.

Question: Is this optimal?

Python implementation

```python
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)
```

Analysis

Theorem

Two n-bit integers can be multiplied in time $O(n^2)$.

Karatsuba: Main idea

- a and b two n-bit natural numbers,
  $\underline{n = 2}^l$ for some $l \in \mathbb{N}_0$.

\# of Bits of input-numbers is power of 2.

$$a = \langle \underbrace{a_{2^n-1}, \dots}_{n/2}, \underbrace{, a_0}_{n/2} \rangle \text{ bits}$$

- Divide: $a = \underbrace{a_1}_{n/2 \text{ bits}} \cdot 2^{n/2} + \underbrace{a_0}_{n/2 \text{ bits}}$, $b = b_1 \cdot 2^{n/2} + b_0$

$$a \cdot b = a_1 \cdot b_1 \cdot 2^n + (a_1 b_0 + b_1 \cdot a_0) \cdot 2^{n/2} + a_0 \cdot b_0$$

- Compute recursively: $\boxed{s_1 = a_1 \cdot b_1}$, $\boxed{s_2 = a_0 \cdot b_0}$, $s_3 = \boxed{(a_1 + a_0) \cdot (b_1 \cdot b_0)}$

3 n/2-bit number multiplications + $O(n)$ time

- Return $s_1 \cdot 2^n + (s_3 - s_1 - s_2) \cdot 2^{n/2} + s_2$

\+ $O(n)$ time for these additions,

$=$

$$= a_1 \cdot b_1 \cdot 2^n + (a_1 \cdot b_0 + b_1 \cdot a_0) \cdot 2^{n/2} + a_0 \cdot b_0 = a \cdot b.$$

Karatsuba: The algorithm

```
function Multiply $(a, b)$
Input: Two $n$-bit integers $a, b \in \mathbb{N}_0$
Output: Their product $a \cdot b$
if $n = 1$ return $a \cdot b$
else
        $a_1, a_0$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of $a$
        $b_1, b_0$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of $b$

        $s_1 = $ Multiply$(a_1, b_1)$
        $s_2 = $ Multiply$(a_0, b_0)$
        $s_3 = $ Multiply$(a_1 + a_0, b_1 + b_0)$

        return $s_1 \cdot 2^n + (s_3 - s_1 - s_2) \cdot 2^{n/2} + s_2$
```

<u>Running time:</u>

$$T(n) \leq \underline{3 \cdot T(n/2)} + O(n)$$

There exists a constant $c > 1$ s.th.

$$T(n) \leq 3 \cdot T(n/2) + c \cdot n$$

whenever $n \geq 1$.

Analysis

Karatsuba: $T(n) = \underline{3}\cdot T(n/\underline{2}) + O(n^1)$, $a=3$, $b=2$, $d=1$

more efficient than $O(n^2)$

because $\log_2 3 < 2$

Theorem

The Karatsuba algorithm runs in time $O(n^{\log_2 3})$.

Theorem (Master theorem)

If $T(n) = \underline{a} \cdot T(\lceil n/\underline{b} \rceil) + O(n^d)$ *for some constants* $a > 0$ $b > 1$ *and* $d \geq 0$*, then*

$$T(n) = \begin{cases} O(n^d), \quad ✓ & \text{if } d > \log_b a \\ O(n^d \log n), & \text{if } d = \log_b a \\ \boxed{O(n^{\log_b a})}, & \text{if } d < \log_b a \quad \Leftarrow \end{cases}$$

$\Leftrightarrow$ $a/b^d < 1$

$\log_2 3 > 1$.

Proof Master Thm: Assume w.l.o.g. n is power of b

$$(b^d)^{\log_b n} = (b^{\log_b n})^d = n^d$$

$T(n) \le a\cdot T(n/b) + c\cdot n^d$ For some constant c.

$$\le a\cdot\left[a\cdot T(n/b^2) + c\cdot\left(\frac{n}{b}\right)^d\right] + c\cdot n^d$$

$$\le a\left[a\left[a\cdot T(n/b^3) + c\left(\frac{n}{b^2}\right)^d\right] + c\cdot\left(\frac{n}{b}\right)^d\right] + c\cdot n^d$$

$$= a^3\cdot T\left(\frac{n}{b^3}\right) + \underline{a^2\cdot c\left(\frac{n}{b^2}\right)^d} + a\cdot c\cdot\left(\frac{n}{b}\right)^d + c\cdot n^d$$

$$= a^3\cdot T\left(\frac{n}{b^3}\right) + \left[\left(\frac{a}{b^d}\right)^2 + \frac{a}{b^d} + 1\right] c\cdot n^d$$

....

$$= c\cdot n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i$$

if $\frac{a}{b^d} < 1$, then Geometric Sequence is bounded even if $\sum_{i=0}^{\infty}$

$\Rightarrow T(n) = O(n^d)$

if $\frac{a}{b^d} > 1$: $T(n) = O\left(n^d\cdot \left(a/b^d\right)^{\log_b n}\right) = O\left(n^{\log_b a}\right)$ | if $\frac{a}{b^d} = 1 \Rightarrow T(n) = O(n^d\cdot \log_b n)$