Computer Algebra

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

Syllabus

- Efficient algorithms, O-notation
- Basic arithmetic *,/,-,+
- Implementation in *Python*
- Newton iteration and running-time equivalence of *,/
- Modular arithmetic, fast exponentiation $\mathbb{Z}_N$
- 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 Fourrier Transform (FFT), efficient multiplication
- Symbolic FFT in rings
- Lattices, Hermite-normal forms and integer linear algebra

Bonus rule

Move to → GR B3 30

- 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 $\geq$ 4.0

Groups up to 3 people (ex. submission)
except for programming ex.!!!

Main literature

1. *Algorithms*, by S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani Use this for first $\frac{1}{3}$-rd of class.

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

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/`

disopt.epfl.ch follow teaching link.

Analysis of Algorithms

Algorithms: The good, the bad . . .

Recall the definition of Fibonacci numbers

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

$F_2 = \underbrace{F_1}_{1} + \underbrace{F_0}_{0} = 1$

$F_3 = F_2 + F_1 = 2$

$F_4 = 3$

The bad

$n \in \mathbb{N}_0$

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

Recursive calls.

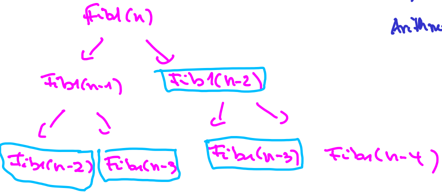
F is monotonously increasing.

$$F_N = F_{N-1} + F_{N-2}$$
$$\geq 2 \cdot F_{N-2}$$
$$\geq 2^2 N_{N-4}$$
$$\geq 2^{\lfloor N/2 \rfloor}$$

The bad

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

Arithmetic op.



Analysis: Count basic arithmetic operations.

$T(n) :=$ # Basic arithmetic operations.

$n \geq 2$:

$$T(n) = 1 + T(n-1) + T(n-2)$$

$$T(n) \geq T(n-1) + T(n-2)$$

growth ~~comparable~~ at least like the Fibonacci numbers themselves.

$\geq 2^{\lfloor n/2 \rfloor}$ ← Exponential.

The good

0	1	1	2

index 0 1 2 3

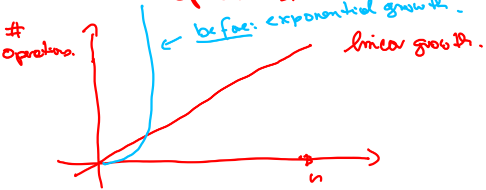
```python
def fib2(n):
    A = [0,1]
    i = 1
    while i < n:
        A.append(A[i-1]+A[i])
        i = i+1
    return A[n]
```

constant # of operations

n x

n= 3.

≤ constant * n basic arithmetic operations.

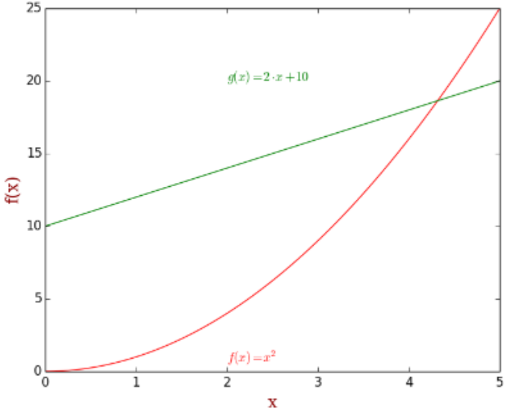


Comparison of running times

- 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: $f_2 \le f_1$.



O-notation

$\smile$ $f \in O(g)$

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

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

$f(n) = 8n^3 + n$ $\quad$ $f = O(g)$ $\quad$ with $N_0 = 1$

$g(n) = n^4.$ $\quad$ $c = 9$

$8 \cdot n^3 + n \leq 9\, n^4. \quad \forall n \in \mathbb{N}.$

$f: n^5, \quad n^{\overset{g}{\ddot{\ }}} \log n$

$\sqrt{n} \approx h.$

$h = O(g)$

$\varphi: \quad n^{1+\varepsilon}, \ \varepsilon > 0$ fixed and small.

$\varphi, g \quad g = O(\varphi).$

Definition

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

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

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

if leading exponent of two polynomials $p(x), q(x) : \mathbb{N} \to \mathbb{R}_{\geq 0}$
is the same $\Leftrightarrow q = \Theta(p)$.

Basic Arithmetic

Natural numbers

- $\mathbb{N} = \{1, 2, 3, \cdots\}$
- Bit-representation
- size$(a) = \lceil log_2(a+1) \rceil$, $a \in \mathbb{N}$.
- Representation in Python

$\langle 1,1,0,1,0 \rangle = 0 \cdot 2^0 + \underbrace{1 \cdot 2^1}_{=2} + 0 \cdot 2^2 + \underbrace{1 \cdot 2^3}_{=8} + \underbrace{1 \cdot 2^4}_{=16} = 26$

L = [0, 1, 0, 1, 1] represents number 26.

size(a) = # Bits needed to represent a.

$\text{size}(a) := \lceil \log_2 (a+1) \rceil$

Addition

Remember the old school days?

$$
\begin{array}{rl}
a: & 1010110 \\
b & 0110101 \\
+ & \\
\hline
 & 10001011
\end{array}
$$

Algorithm

bitlen([0, 1, 0]) = 3

```
def Add(L1,L2):

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

    Out = []
    carry = 0

    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
```

n = max{len(L1), len(L2)}

Analysis.

[0,1,2,3,4]

const # of op.

L1 1 0 1 0

→ L2 1 0 1

carry = 0

h = 1

Out [1]

carry = 0

i = 1 Out [1,1] carry = 0

i = 2 Out [1,1,0] carry = 1, h = 2

i = 3

O(n)

Analysis

Theorem

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

Subtraction

Exercise

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

$L1 - L2$ $\quad$ $L1 \geq L2$ $\quad$ $L1 < L2$.

Multiplication

$a \quad 1\ 1\ 0\ 1\ 0$

$b \quad 0\ \boxed{1}\ 0\ \boxed{1}\ \boxed{1}$

a, b are both n-bit numbers.

How many bits has $a * b$?

$2 \cdot n$

$$\left.\begin{array}{r} 11010 \\ 110100 \\ 11010000 \end{array}\right\} \text{Add them up.}$$

Add up $\leq n$ times $2n$-bit numbers.

Running time: $O(n^2)$

$\Omega(n^2)$

next week.

$O(n^{\log_2 3})$

even later

$O(n \cdot \log n \log \log n)$

a

$\underline{11111}$ · n bits.

$$\left.\begin{array}{l} a \\ a0 \\ \vdots \\ a\underbrace{000000}_{n} \end{array}\right\} \Omega(n^2) \text{ space.}$$

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
```

$Out = x \cdot y$

$z = x \cdot \lfloor y/2 \rfloor$

Case 1: $\lfloor y/2 \rfloor = y/2$ ← y even.

Case 2: $\lfloor y/2 \rfloor = y/2 - \frac{1}{2}$

$Out = 2 \cdot z$

$x \cdot y = x(\lfloor y/2 \rfloor \cdot 2 + 1)$

$= \underbrace{\cancel{x} \lfloor y/2 \rfloor 2} + x$

Python implementation

```
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

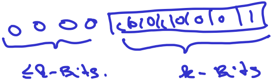
Theorem

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

Multiplication faster than $O(n^2)$.

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

possibly by adding additional zeros.



a, b : n-bit numbers.

$a = a_1 \cdot 2^{n/2} + a_2$, a_1, a_2 are $n/2$-bit numbers.

$b = b_1 \cdot 2^{n/2} + b_2$, b_1, b_2 — " —

$$a * b = (a_1 \cdot 2^{n/2} + a_2)(b_1 \cdot 2^{n/2} + b_2)$$

$$= \boxed{\underbrace{a_1 \cdot b_1}_{\leq n\text{-bits}}} \cdot 2^n + \overbrace{(\boxed{a_1 b_2} + \boxed{b_2 a_1})}^{\leq n+1 \text{ Bits.}} \cdot 2^{n/2} + \boxed{\underbrace{a_2 \cdot b_2}_{\leq n \text{ bits}}}.$$

to be taken care by recursive calls.

Running time:

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

for + &

Running time:

for +

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

for now: assume $c = 1$.

$$\geq 4 \cdot \left(4 \cdot T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n$$

$$\geq 4^i \cdot T(1) + \dots + \qquad i = \log_2 n$$

$$4^{\log_2 n} = n^2$$

The Karatsuba Trick:

$$a * b = (a_1 \cdot 2^{n/2} + a_2)(b_1 \cdot 2^{n/2} + b_2) = a_1 \cdot b_1 \cdot 2^n + (a_1 b_2 + a_2 b_1) * 2^{n/2} + a_2 \cdot b_2$$

Compute: $\underbrace{(a_1 + a_2)}_{n/2+1 \text{ Bits.}} \underbrace{(b_1 + b_2)}_{n/2+1 \text{ Bits}}$

$$= a_1 \cdot b_1 + a_1 b_2 + a_2 \cdot b_1 + a_2 \cdot b_2$$

$a_1 \cdot b_1$

$a_2 \cdot b_2$

compute recursively.

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 $c > 0$.

Thm: Two n-bit numbers can be multiplied in time $O(n^{\log_2 3})$

$$T(n) \leq 3\left(3 \cdot T(n/2^2) + c \cdot \frac{n}{2}\right) + c \cdot n$$

$$= 3^2 \cdot T(n/2^2) + \frac{3 \cdot c}{2} \cdot n + c \cdot n$$

$$\leq 3^2 \cdot \left(3 \cdot T\left(\frac{n}{2^3}\right) + c \cdot \frac{n}{2^2}\right) + c \cdot 3 \cdot \frac{n}{2} + c \cdot n$$

$$\vdots$$

$$\leq 3^{\overset{\log_2 n}{\shortparallel}{i}} \cdot T(1) + c \cdot n \left(\frac{3}{2}\right)^{i-1} + c \cdot n \; 2 + \dots + c \cdot n \left(\frac{3}{2}\right)^0$$

$$= n^{\log_2 3} T(1) + c \cdot n \cdot \sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^i = n^{\log_2 3} + c \cdot n \frac{\left(\frac{3}{2}\right)^{\log_2(n)} - 1}{3/2 - 1}$$

$$\leq n^{\log_2 3} + c \cdot n \, n^{\log_2(3/2)}$$

$$= n^{\log_2 3} + c \cdot \cancel{n} \cdot \frac{n^{\log_2 3}}{\cancel{n}}$$

$$= O(n^{\log_2 3})$$

$\log_2 3 < 2.$