.

About complexity

We define the class informally P in the following way:

P = The set of all problems that can be solved by a polynomial time algorithm, i.e., an algorithm that runs in time $O(n^k)$ in the worst case, where k is some integer and n is the size of the input.

We can contrast this class with

EXP=The set of all problems that can be solved by an exponential time algorithm, i.e., an algorithm that runs in time $O(c^{n^k})$ in the worst case, where k is some integer, $c > 1$ some real number and n is the size of the input.

It is universally agreed that an algorithm is efficient if and only if it is polynomial.

This makes it critical to define the size of the input in a ''correct way''. For instance, we must be careful if we have numbers as input.

Number theoretical algorithms

Number theoretical algorithms are algorithms handling problems such as deciding if a number is a prime, finding greatest common divisor and so on. Input to the algorithms are integers. The natural measure of the size of the input is the logarithm of the numbers.

Ex: Test if a number is a prime.

PRIME(n)
(1) **for** $i \leftarrow 2$ **to** $\sqrt{n}$
(2)   **if** $i|n$
(3)   **return** Not prime
(4) **return** Prime

This algorithm has complexity O($\sqrt{n}$). It is to slow for large numbers. We would like to have an algorithm that runs in time $O((\log n)^k)$ for some k.

Greatest Common Divisor

Greatest common divisor: $gcd(a,b) =$ is the largest integer that divides both a and b.

Euclides' algorithm:

The $gcd(a,b)$ can be computed by the following method:

$r_1 = a \mod b$

$r_2 = b \mod r_1$

$r_3 = r_1 \mod r_2$

...

$r_{n+1} = r_{n-1} \mod r_n = 0$

Then $gcd(a,b) = r_n$.

It is easy to verify that $r_{i+2} < \frac{r_i}{2}$ for all i. This means that the algorithm stops after $O(\log n)$ steg. So the algorithm is efficient.

The algorithm can be implemented recursively.

EUKLIDES(a, b)
(1) **if** $b = 0$
(2) **return** a
(3) **return** EUKLIDES($b, a \mod b$)

If $gcd(a, b) = d$ there are integers x, y such that $ax + by = d$. (x, y can be negative). In fact, d is the smallest integer > 0 on that form. The integers x, y can be found by a modified version of Euclides' algorithm:

MOD-EUKLIDES(a, b)
(1) **if** $b = 0$
(2) **return** $(a, 1, 0)$
(3) $(d', x', y') \leftarrow$ MOD-EUKLIDES(b, a mod b)
(4) $(d, x, y) \leftarrow (d', y'x' - [\frac{a}{b}]y')$
(5) **return** (d, x, y)

Finding the inverse: If $gcd(a, n) = 1$ there are integers x, y such that $ax + ny = 1$. Then $x = a^{-1} \mod n$. So we can find a^{-1} by using MOD-EUKLIDES(a, n).

Modular exponentiation

In cryptography it is important to be able to compute $a^b \mod n$ for very large numbers in an efficient way. The following simple algorithm is not efficient:

POT(a, b, n)
(1) $d \leftarrow 1$
(2) **for** $i \leftarrow 2$ **to** b
(3) $d \leftarrow d \cdot a \mod n$
(4) **return** d

The following modified algorithm, though, is efficient:

MOD-EXP(a, b, n)
(1) $d \leftarrow 1$
(2) Let $(b_k, b_{k-1}, ..., b_0)$ be the binary representation off b
(3) **for** $i \leftarrow k$ **to** 0
(4) $d \leftarrow d \cdot d \mod n$
(5) **if** $b_i = 1$
(6) $d \leftarrow d \cdot a \mod n$
(7) **return** d

To decide of a number is a prime

Fermat's Theorem: If p is a prime and a is an integer such that an then $a^{p-1} \equiv 1 \pmod{p}$.

We can set $a = 2$. If n is such that $2^{n-1} \not\equiv 1 \pmod{n}$ then n cannot be a prime. Therefore, we can use the following algorithm to test if n is a prime:

FERMAT(n)
(1) $k \leftarrow$ MOD-EXP($2, n-1, n$)
(2) **if** $k \not\equiv 1 \pmod{n}$
(3) **return** FALSE
(4) **return** TRUE

If FERMAT returns FALSE we know for sure that n is not a prime. But unfortunately, FERMAT might return TRUE even if n is not a prime. For instance, $2^{340} \equiv 1 \mod 341$ but 341 is not a prime.

We can use a so *probabilistic* algorithm which randomly choses a number a in $[2, n-1]$ and does a Fermat test with a.

```
PROB-FERMAT(n, s)
for j ← 1 to s
    a ← RANDOM(2, n − 1)
    k ← MOD-EXP(a, n − 1, n)
    if k ≢ 1( mod n)
        return FALSE
return TRUE
```

The algorithm is probabilistic in the sence that it can give different answers at different times even if it starts with the same input. The following must, however, be true:

if n is a prime then the algorithm must return TRUE. This means that the algorithm returns FALSE then we know that n is not a prime. So FALSE is the only definite answer we can get.

$P(n$ is not prime |
The algorithm returns FALSE $) = 1$

What about the probability
$P(n$ Is prime |The algorithm returns TRUE)?
It can be shown that for almost all non-prime n we get:

$P($ The algorithm returns FALSE $) > \frac{1}{2}$.

(For primes n we have
$P($ Algoritmen svarar TRUE $) = 1$.)

Problem are caused by so called **Carmichael numbers**.

Carmichael numbers : A Carmichael number is a non-prime integer n such that $a^{n-1} \equiv 1 \pmod{n}$ for all $a \in [2, n-1]$. The smallest Carmichael number is 341.

$P($ The algorithm returns TRUE |
n is a Carmichael number $) = 1$.

In order to handle Carmichael numbers we can use the following algorithm:

```
WITNESS(a, n)
Let n − 1 = 2^t u, t ≥ 1, where u is odd
x_0 ← MOD-EXP(a, u, n)
for i ← 1 to t
    x_i ← x_{i−1}^2  mod n
    if x_i = 1 och x_{i−1} ≠ 1 och x_{i−1} ≠
    n − 1
        return TRUE
if x_t ≠ 1
    return TRUE
return FALSE
```

The following can be shown for WITNESS:

$P($ WITNESS returns TRUE $|$
n Is not prime $) > \frac{1}{2}$ for all n. If you make repeated calls to WITNESS can get arbitrarily high probability for a correct answer. This version of the algorithm is called Miller - Rabin's Test.

```
MILLER-RABIN(n, s)
    for j ← 1 to s
        a ← RANDOM(1, n − 1)
        if WITNESS(a, n)
            return Not prime
    return Prime
```

Here

$P($ The algorithm returns Prime $|$ n is prime$) = 1$.

$P($ The algorithm returns Not prime $|$ n is not prime $) > 1 - \frac{1}{2^s}$.

It is, of course, also interesting to study the ''reversed'' conditional probabilities:

$P(n$ is not prime |
The algorithm returns Not prime$) = 1$.

The probability
$P(n$ is prime | The algorithm returns Prime)
is trickier. It can be computed as

$$\frac{P(n \text{ is prime and the algorithm returns Prime})}{P(\text{ The algorithm returns Prime})} = \frac{P(n \text{ is prime})}{P(\text{ The algorithm returns Prime})}$$

But then we need to know $P(n$ is prime). If we know that the probability is α we can use Baye's law to show that
$P(n$ is prime | The algorithm returns Prime)
$> \frac{2^s}{2^s+(\frac{1}{\alpha}-1)}$.

Since August 2002 it is known that there is an algorithms that decides primality (in the usual non-probabilistic sense) in polynomial time. This algorithm is much more complicated and slower than Miller-Rabin's algorithm.

The Miller -Rabin algorithm is an example of probabilistic algorithms. The general with probabilistic algorithms are that they use a certain amount of randomness, usually from a pseudo-random generator. We will look at a more general definition of probabilistic algorithms.

Monte Carlo algorithms

Suppose that we have a decision problem, i.e. a problem with yes/no as answer. We say that F is a **Yes-based Monte Carlo algorithm** for solving the problem if F is polynomial and:

1. If the answer to the problem is yes, then $F(x) = Yes$ with probability 1.

2. If the answer to the problem is no, then $F(x) = No$ with probability $> \frac{1}{2}$.

No-based Monte Carlo algorithms are defined in the obvious, symmetrical way.

Definition: The class RP is the set of all problems that can be solved by a Yes-based Monte Carlo algorithm.

It is easily seen that $P \subseteq RP$.

Probabilistic algorithms

Ex: Is the polynomial

$f(x, y) = (x - 3y)(xy - 5x)^2 - 10x^3y + 25x^3 + 3x^2y^3 + 30x^2y^2 - 75x^2y$

identically equal to 0?

Test: Chose some values x_i, y_i randomly and test if $f(x_i, y_i) = 0$.

Two possibilities:

1. $f(x_i, y_i) \neq 0$. Then $f \neq 0$.

2. $f(x_i, y_i) = 0$ for all chosen values x_i, y_i. What is then the probability for $f = 0$?

Theorem: If $f(x_1, ..., x_m)$ is not identically equal to 0 and each variable occurs with degree at most d and M is an integer, then the number of zeros in the set $\{0, 1, ..., M-1\}^m$ is at most mdM^{m-1}. This gives us:

$P[$ A random integer in $\{0, 1, ..., M-1\}^m$ is a zero$]$ $= \frac{1}{mdM^{m-1}} = \delta$.

This means that if we have done k tests indicating $f = 0$, then $P[f = 0] \geq (1-\delta)^k$.

The probabilistic algorithms are closely related to randomized algorithms. The difference is that the randomized algorithms usually are deterministic but nevertheless, use random steps in the computation. What is uncertain is the running time rather than the result. An example is the famous Quick sort algorithm (or more exactly, the randomized version of it).

Quick sort

QuickSort($v[i..j]$)
(1) **if** $i < j$
(2) $m \leftarrow$ Partition($v[i..j], i, j$)
(3) QuickSort($v[i..m]$)
(4) QuickSort($v[m + 1..j]$)

The complexity analysis is more complicated than it is for Merge sort. It can nevertheless be shown that the complexity is $O(n \log n)$ *in the mean*.

Sorting in linear time

Sorting algorithms that only uses comparisons between elements can never be faster than $\Theta(n \log n)$. But there are algorithms which use extra information about the elements. For instance, if we want to sort integers we might know upper and lower bounds for the integers. Then it is possible to sort in linear time.

Counting sort

Assume that we have n objects $A[1..n]$ with keys which are integers in $[1, k]$. The following algorithm sorts in time $O(n + k)$:

CountingSort(A, B, k)

for $i = 1$ **to** k
    $C[i] \leftarrow 0$
for $i = 1$ **to** n
    $C[A[i]] \leftarrow C[A[i]] + 1$
for $i = 2$ **to** k
    $C[i] \leftarrow C[i-1] + C[i]$
for $j = n$ **to** 1
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$