10 Randomized algorithms

Summer Term 2011

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

Randomized algorithms

Overview

- Classes of randomized algorithms
- Quicksort
- Randomized Quicksort
- Randomized primality test
- Cryptography

1. Classes of randomized algorithms

- **Las Vegas** algorithms
  always correct; expected running time “probably fast”

  Example: randomized Quicksort

- **Monte Carlo** algorithms (mostly correct):
  probably correct; guaranteed running time

  Example: randomized primality test

2. Quicksort

Unsorted range *A*[*l*, *r*] in array *A*

A[l … r-1]	*p*

A[l…m – 1]	*p*	*A[m + 1…r]*

Quicksort **Quicksort**

Quicksort

Algorithm: *Quicksort*

Input: unsorted range $[l, r]$ in array A

Output: sorted range $[l, r]$ in array A

if $r > l$

  then choose pivot element $p = A[r]$

    $m = divide(A, l, r)$

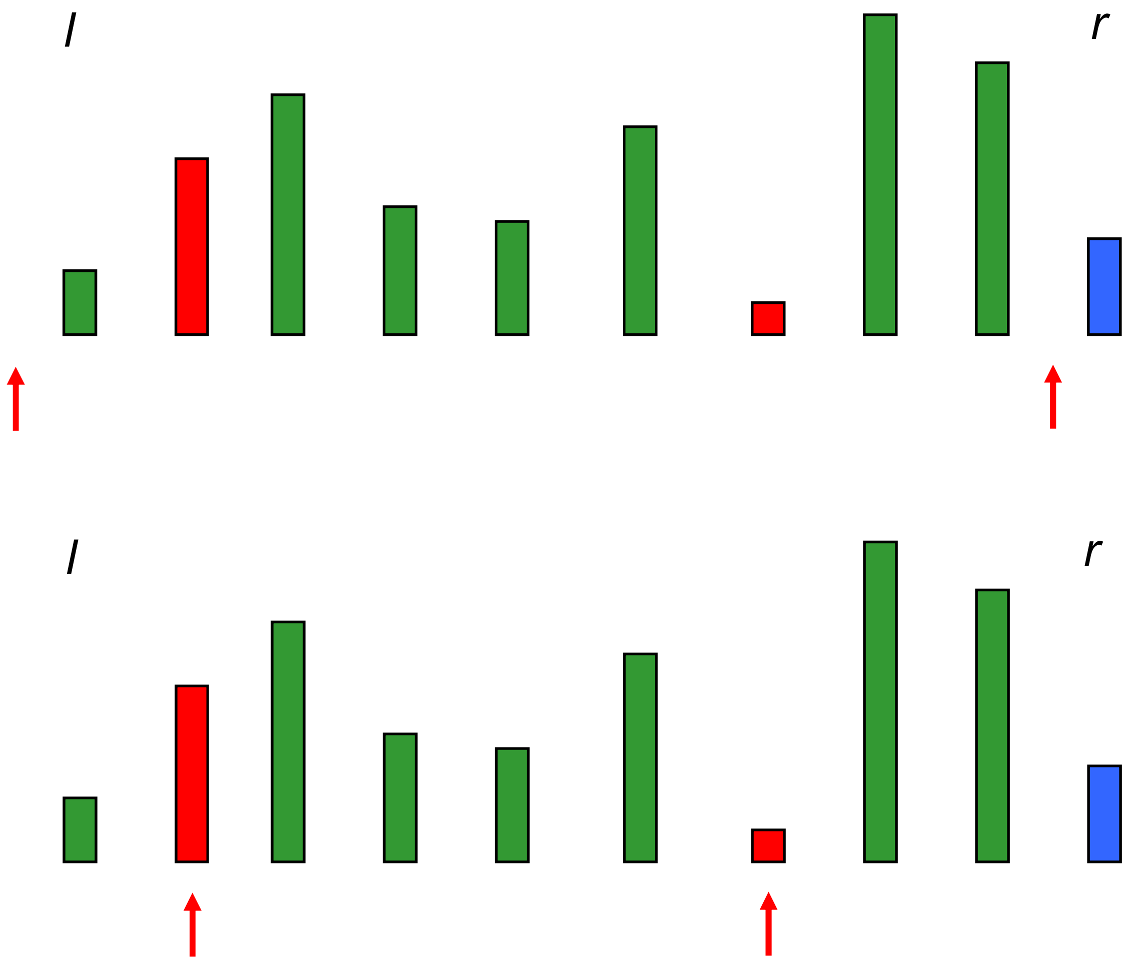
      /* Divide A according to p:

        $A[l],\ldots,A[m-1] \leq p \leq A[m+1],\ldots,A[r]$

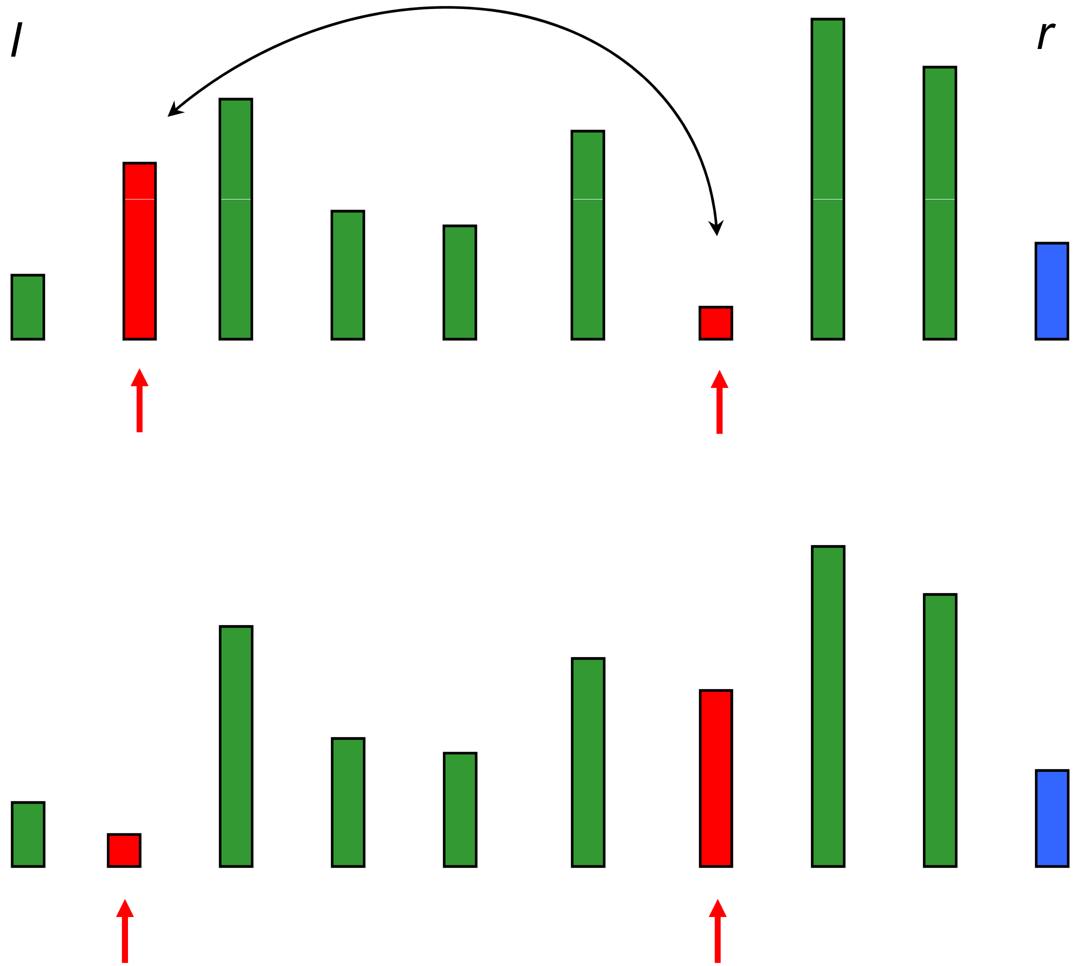
      */

Quicksort($A, l, m - 1$)
Quicksort ($A, m + 1, r$)

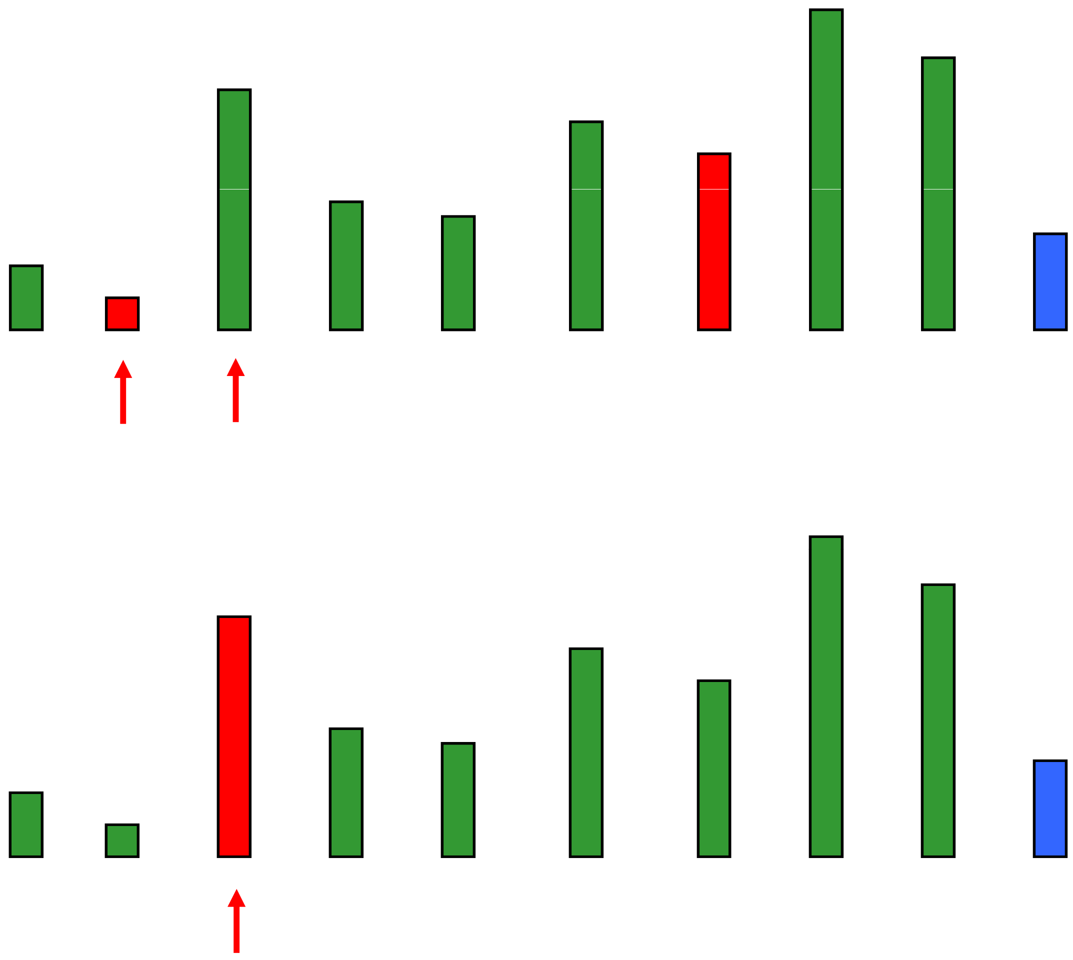
The *divide* step



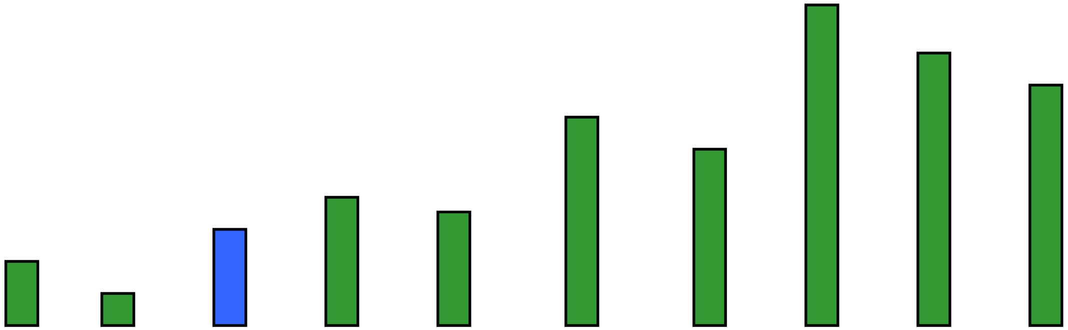
The *divide* step



The *divide* step



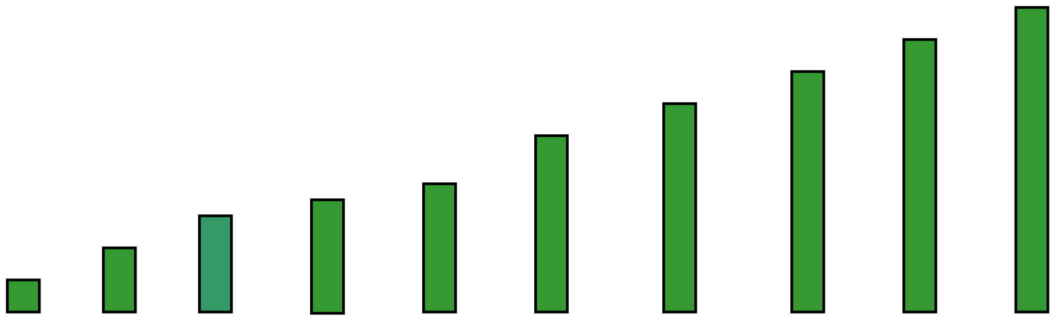
The *divide* step



divide(*A, l , r*):

- returns the index of the pivot element in *A*
- can be done in time *O(r – l)*

Worst case input



n elements*:*

Running time: $(n\text{-}1) + (n\text{-}2) + \ldots + 2 + 1 = n \cdot (n\text{-}1)/2 = \Omega(n^2)$

3. Randomized Quicksort

Algorithm: Quicksort
Input: unsorted range $[l, r]$ in array A
Output: sorted range $[l, r]$ in array A

```
if r > l
    then randomly choose a pivot element p = A[i] in range [l, r]
        swap A[i] and A[r]
        m = divide(A, l, r)
        /* Divide A according to p:
            A[l],....,A[m – 1] ≤ p ≤ A[m + 1],...,A[r]
         */
         Quicksort(A, l, m - 1)
         Quicksort(A, m + 1, r)
```

Analysis 1

n elements; let S_i be the i-th smallest element

- S_1 is chosen as pivot with probability $1/n$:
- Sub-problems of sizes 0 and n-1

•
•
•

- S_k is chosen as pivot with probability $1/n$:
- Sub-problems of sizes k-1 and n-k

•
•
•

- S_n is chosen as pivot with probability $1/n$:
- Sub-problems of sizes n-1 and 0

Analysis 1

Expected running time:

$$E(T(n)) = \frac{1}{n}\sum_{k=0}^{n-1}(E(T(k)) + E(T(n-k-1))) + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} E(T(k)) + \Theta(n)$$

($\Theta(n)$ *because* of the divide-procedure)

One can show that $E(T(n)) \in O(n \log n)$.

With high probability, bad inputs cannot spoil the performance.

4. Primality test

Definition:

An integer $p \geq 2$ is prime **iff** ($a \mid p \rightarrow a = 1$ **or** $a = p$).

Algorithm: deterministic primality test (naive)

Input: integer $n \geq 2$

Output: answer to the question: Is n prime?

```
if n = 2 then return true
if n even then return false
for i = 1 to √n/2 do
        if 2i + 1 divides n
        then return false
return true
```

Complexity: $\Theta(\sqrt{n})$ where n is the input, but the input size is just log n.

Primality test

Goal:

Randomized method

- Polynomial time complexity (in the length of the input)
- If answer is “not prime”, then n is not prime
- If answer is “prime”, then the probability that n is not prime is at most $p>0$

k iterations: probability that n is not prime is at most p^k.

Randomized primality test

Theorem: (Fermat‘s little theorem)

If p prime and $0 < a < p$, then

$$a^{p-1} \bmod p = 1. \text{ (I.e., } p \text{ divides } a^{p-1} - 1\text{)}$$

Definition:

n is pseudoprime to base a, if n not prime and

$$a^{n-1} \bmod n = 1.$$

Example: $n = 11 * 31 = 341$, $a = 2$

$2^{340} \bmod 341 = 1$

but: $n = 341$, $a = 3$

$3^{340} \bmod 341 = 56 \neq 1$

Randomized primality test 1

1 Randomly choose $a \in [2, n\text{-}1]$

2 **if** a^{n-1} mod $n = 1$

3 **then** n is possibly prime

4 **else** n is definitely not prime

Advantage: This only takes polynomial time.

Examples: $n = 17$, $a = 2$: $2^{16} =$ 65536. 65536 mod 17 = 1.

$n = 23$, $2^{22} =$ 4194304. 4194394 mod 23 = 1.

$n = 341$, 2^{340} mod 341 = 1.

17 and 23 are indeed prime, 341 is not!

Prob(n is not prim, but a^{n-1} mod $n = 1$) ?

Carmichael numbers

Problem: Carmichael numbers

Definition: An integer n is called Carmichael number if

$$a^{n-1} \bmod n = 1$$

for all a with GCD(a, n) = 1. **(GCD = greatest common divisor)**

Example:

Smallest Carmichael number: 561 = 3 * 11 * 17

561 is pseudoprime to any base a that is not divisible by 3 or 11 or 17.

To show that 561 is not prime, we hence need a base a that is divisible by 3 or 11 or 17. This is still quite likely to find, but there are worse examples.

Randomized primality test 2

Theorem:

If p prime and $0 < a < p$, then the only solutions to the equation

$$a^2 \bmod p = 1$$

are $a = 1$ and $a = p - 1$.

Definition:

a is called non-trivial square root of 1 mod n, if

$$a^2 \bmod n = 1 \text{ and } a \neq 1, n-1.$$

Example: $n = 35$

$$6^2 \bmod 35 = 1$$

Fast exponentiation

Idea:

During the computation of a^{n-1} ($0 < a < n$ randomly chosen), which we need for the first primality test, test as a byproduct whether there is a non-trivial square root 1 mod n.

Method for the computation of a^n:

Case 1: [n is even]

$a^n = a^{n/2} * a^{n/2}$

Case 2: [n is odd]

$a^n = a^{(n-1)/2} * a^{(n-1)/2} * a$

Fast exponentiation

Example:

$a^{62} = (a^{31})^2$

$a^{31} = (a^{15})^2 * a$

$a^{15} = (a^{7})^2 * a$

$a^{7} \quad = (a^{3})^2 * a$

$a^{3} \quad = (a)^2 * a$

To compute a^n, the exponents are obviously divided by 2 (at least) in each step. Hence there are O(log n) intermediate steps.

In each intermediate step, we multiply and compute the square for operands of number size $O(a^n)$ and hence representation size $O(\log a^n)$, leading to $O(\log^2 a^n)$ for each intermediate step.

Overall complexity: $O(\log^2 a^n \log n)$

Fast exponentiation + squares

```
boolean isProbablyPrime;

power(int a, int p, int n) {
    /* computes a^p mod n and checks during the
       computation whether there is an x with
       x^2 mod n = 1 and x ≠ 1, n-1 */

    if (p == 0) return 1;
    x = power(a, p/2, n)
    result = (x * x) % n;
```

Fast exponentiation + squares

```
/* check whether x² mod n = 1 and x ≠ 1, n-1 */
    if (result == 1 && x != 1 && x != n -1 )
        isProbablyPrime = false;

    if (p % 2 == 1)
        result = (a * result) % n;

    return result;
}
```

Complexity: $O(\log^2 n \log p)$

Combined Procedure Miller-Rabin

```
primalityTest(int n) {
    /* carries out the randomized primality test for
       a randomly selected a */

    a = random(2, n-1);

    isProbablyPrime = true;

    result = power(a, n-1, n);

    if (result != 1 || !isProbablyPrime)
        return false;
    else
        return true;
}
```

Combined Procedure Miller-Rabin

Theorem:

If n is not prime, there are at most $\frac{n-9}{4}$

integers $0 < a < n$, for which the algorithm primalityTest fails. Hence the probability of failure is

$$\frac{\frac{n-9}{4}}{n} < \frac{\frac{n}{4}}{n} = \frac{1}{4}$$

If for a number n we do log n tests we get a probability of

$$\left(\frac{1}{4}\right)^{\log n} = \frac{1}{n^2}$$

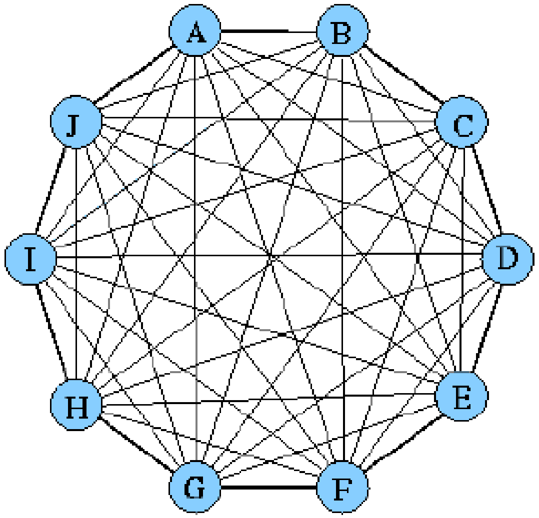
Of failure. E.g. we might take n around 2^{500}.

Application: cryptosystems

Traditional encryption of messages with secret keys

Disadvantages:

1. The key *k* has to be exchanged between A and B before the transmission of the message.
2. For messages between *n* parties $n(n-1)/2$ keys are required.



Advantage:

Encryption and decryption can be computed very efficiently.

Desired properties of cryptographic systems

- confidential transmission
- integrity of data
- authenticity of the sender
- reliable transmission

Public-key cryptosystems

Diffie and Hellman (1976)

Idea: Each participant A has two keys:

1. a public key P_A accessible to every other participant
2. a private (or: secret) key S_A only known to A.

Public-key cryptosystems

D = set of all legal messages, e.g. the set of all bit strings of finite length

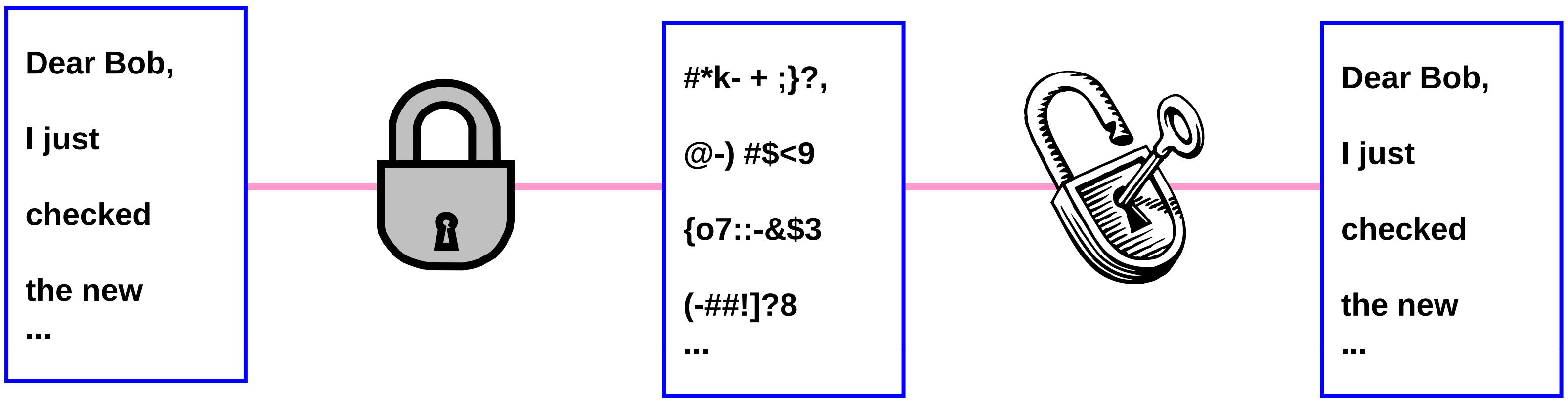
$$P_A, S_A : D \rightarrow D$$

Three conditions:

1. P_A and S_A can be computed efficiently
2. $S_A(P_A(M)) = M$ and $P_A(S_A(M)) = M$
   (P_A is the inverse function of S_A and vice-versa)
3. S_A cannot be computed from P_A with reasonable effort.

Encryption in a public-key cryptosystem

A sends a message *M* to *B*.



Encryption in a public-key cryptosystem

1. ***A*** accesses ***B***'s public key P_B (from a public directory or directly from B).
2. ***A*** computes the encrypted message $C = P_B(M)$ and sends C *to* ***B***.
3. After ***B*** has received message C, ***B*** decrypts the message with his own private key S_B: $M = S_B(C)$

Generating a digital signature

A sends a digitally signed message M' to ***B***:

1. ***A*** computes the digital signature σ for M' with her own private key:

$$\sigma = S_A(M')$$

2. ***A*** sends the pair (M', σ) to ***B***.

3. After receiving (M', σ), ***B*** verifies the digital signature:

$$P_A(\sigma) = M'$$

σ can by verified by anybody via the public P_A.

RSA cryptosystems

R. Rivest, A. Shamir, L. Adleman

Generating the public and private keys:

1. Randomly select two primes p and q of similar size, each with $l+1$ bits ($l \geq 500$).
2. Let $n = p \cdot q$
3. Let e be a (small) integer that does not divide $(p - 1) \cdot (q - 1)$.
4. Calculate $d = e^{-1} \bmod (p - 1)(q - 1)$

   i.e.:

$$d \cdot e \equiv 1 \bmod (p - 1)(q - 1)$$

RSA cryptosystems

5. Publish $P = (e, n)$ as public key

6. Keep $S = (d, p, q)$ as private key

Divide message (described in a binary string) in blocks of size $2 \cdot l$.

Interpret each block M as a binary number: $0 \leq M < 2^{2 \cdot l}$

$$P(M) = M^e \bmod n \qquad\qquad S(M) = M^d \bmod n$$

RSA has the desired properties ...

We have to show that

1. P_A and S_A can be computed efficiently.
2. $S_A(P_A(M)) = M$ and $P_A(S_A(M)) = M$
   (P_A is the inverse function of S_A and vice-versa)
3. S_A cannot be computed from P_A with reasonable effort.

1 is fulfilled because exponentiation can be computed efficiently.

P and S are inverses

We have (some basic math ...)

$$\begin{aligned} M^{(p-1)\cdot(q-1)} &\equiv 1 \quad \bmod p \\ M^{(p-1)\cdot(q-1)} &\equiv 1 \quad \bmod q \\ M^{(p-1)\cdot(q-1)} &\equiv 1 \quad \bmod p \cdot q \end{aligned}$$

and hence

$$\begin{aligned} S(P(M)) &\equiv (M^e)^d && \bmod n \\ &\equiv M^{e\cdot d} && \bmod n \\ &\equiv M^{1+r\cdot(p-1)\cdot(q-1)} && \bmod n \\ &\equiv M \cdot (M^{(p-1)\cdot(q-1)})^r && \bmod n \\ &\equiv M && \bmod n \end{aligned}$$

The other direction is analogous.

S is hard to compute

This is unproven!

According to current knowledge, to compute *d* from e one would need to know *p* and *q.*

Also according to current knowledge, computing *p* and *q* from *n* is hard.

Even the fastest computers have never cracked RSA!

Summary

- We have seen two randomised algorithms:
  - Quicksort
  - Prime test
- We have also seen an application of big prime numbers: cryptography.