10 Randomized algorithms

Summer Term 2010

Robert Elsässer

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Randomized algorithms

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

1. Classes of ramdomized algorithms

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

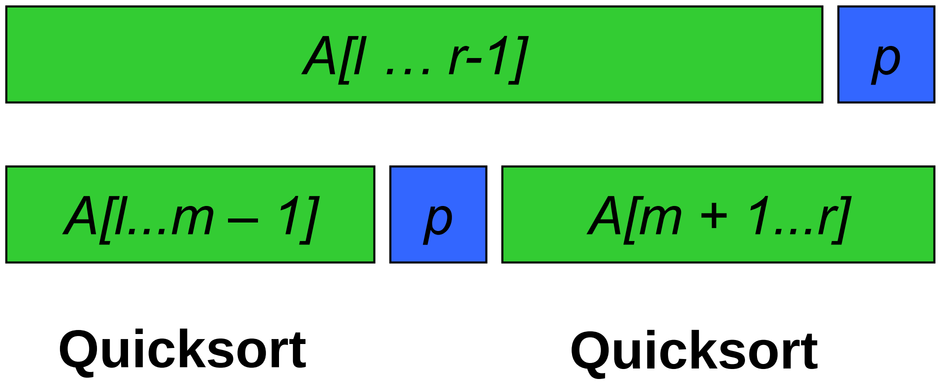
  Example: randomized Quicksort

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

  Example: randomized primality test

2. Quicksort

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



Quicksort

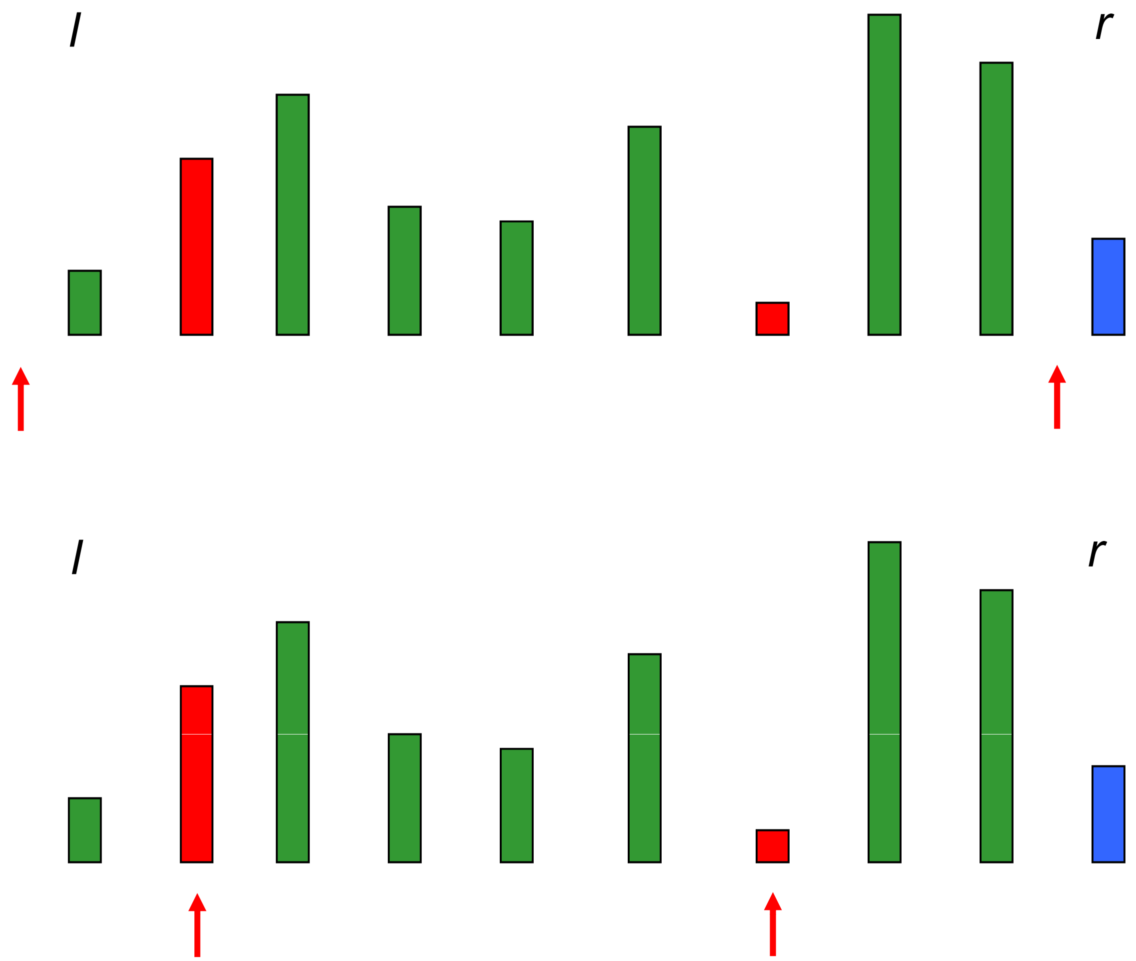
Algorithm: *Quicksort*

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

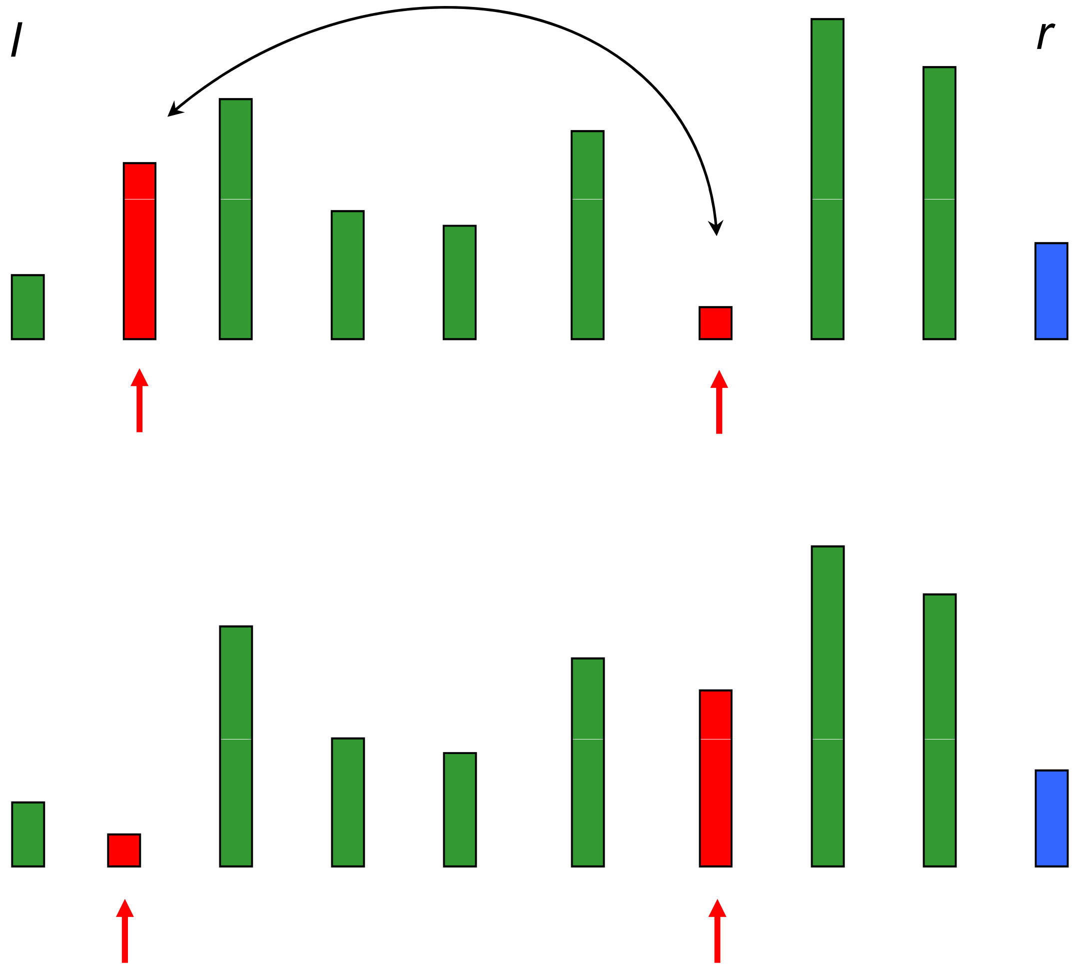
Output: sorted range $[l, r]$ in array A

if $r > l$
  then choose pivot element p = A[r]
    $m = divide(A, l, r)$
      /* Divide A according to p:
        A[l],....,A[m – 1] ≤ p ≤ A[m + 1],...,A[r]
      */
Quicksort(A, l, m - 1)
Quicksort (A, m + 1, r)

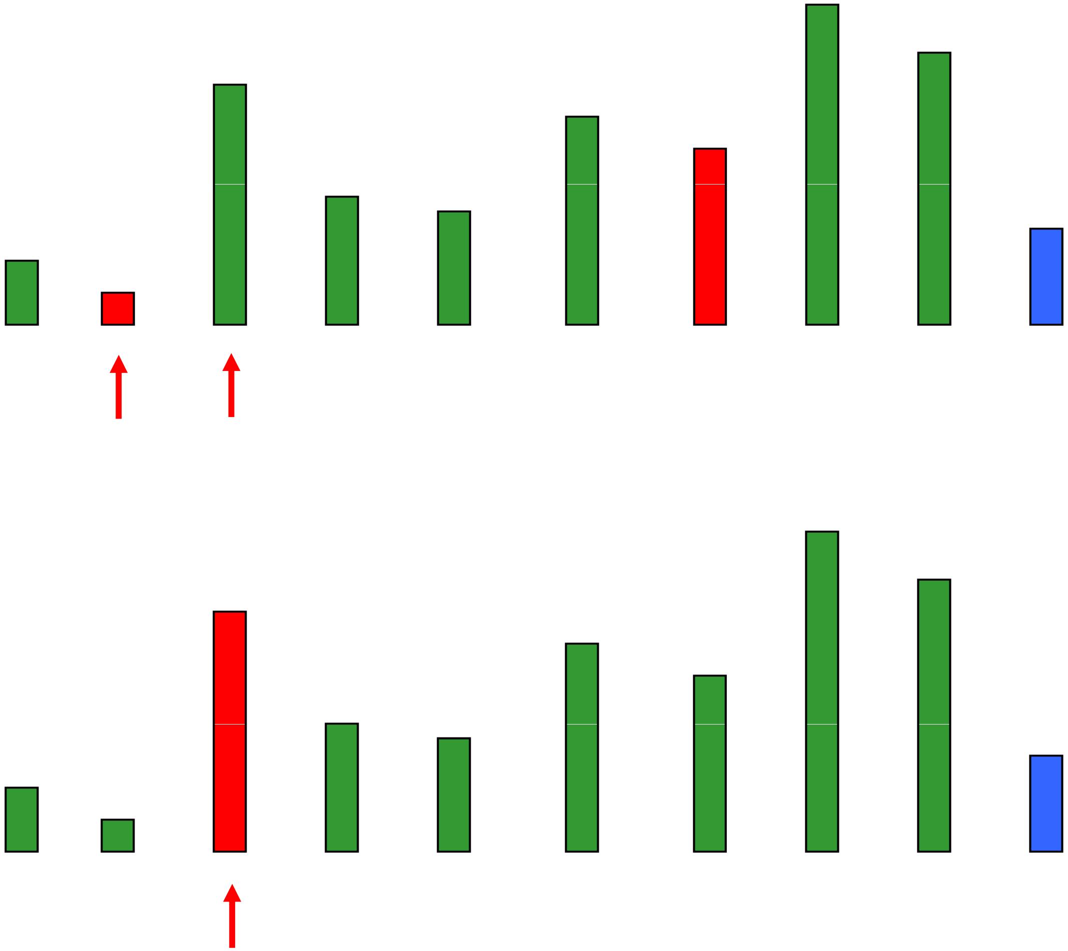
The *divide* step



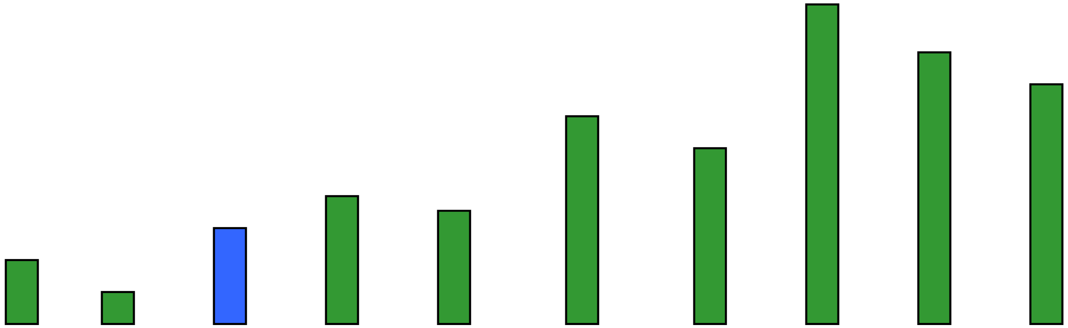
The *divide* step



The *divide* step



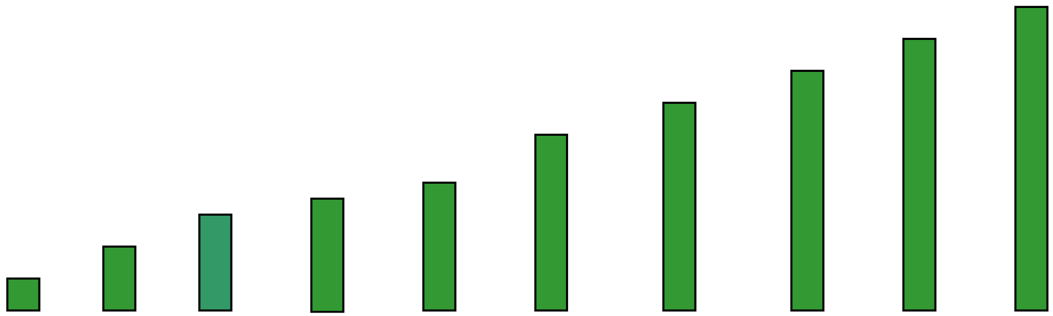
The *divide* step



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: $(n-1) + (n-2) + \ldots + 2 + 1 = n \cdot (n-1)/2$

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:

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

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

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

Claim: $E[T(n)] \le cn \lg n$ for $n \ge 2$ and some $c > 0$

proof : choose c large enough s.t. $T(n) \le cn \lg n$ for n = 2

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

$$\le \frac{2c}{n}\sum_{k=2}^{n-1} k \lg k + \Theta(n) \text{(include T(0) and T(1) into } \Theta(n))$$

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

$$(\sum_{k=2}^{n-1} k \lg k \le \frac{n^2}{2}\lg n - \frac{n^2}{8})$$

(Proof as exercise!)

$$E(T(n)) \le \frac{2c}{n}(\frac{n^2}{2}\lg n - \frac{n^2}{8}) + \Theta(n)$$

$$\le cn \lg n - (\frac{cn}{4} - \Theta(n))$$

$$E(T(n)) \leq cn \lg n - (\frac{cn}{4} - \Theta(n))$$

$$E(T(n)) \le cn\lg n - (\frac{cn}{4} - \Theta(n))$$

desired

Should be >=0

$$E(T(n)) \le cn\lg n - (\frac{cn}{4} - \Theta(n))$$

desired

Should be >=0

$E(T(n)) \le cn \lg n$ where we choose c large enough s.t. $cn/4$ dominate $\Theta(n)$

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: $\Theta(\sqrt{n})$

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

Primality test

Observation:

Each odd prime number p divides $2^{p-1} - 1$.

Examples: $p = 17$, $2^{16} - 1 = 65535 = 17 * 3855$

$p = 23$, $2^{22} - 1 = 4194303 = 23 * 182361$

Simple primality test:

1 Calculate $z = 2^{n-1} \bmod n$

2 if $z = 1$

3 then n is possibly prime

4 else n is definitely not prime

Advantage: This only takes polynomial time

Simple primality test

Definition:

n is called pseudoprime to base 2, if *n* is not prime and

$$2^{n-1} \bmod n = 1.$$

Example: $n = 11 * 31 = 341$

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

Randomized primality test

Theorem: (Fermat's little theorem)

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

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

Definition:

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

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

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

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

Randomized primality test

Algorithm: Randomized primality test 1

```
Randomly choose a ∈ [2, n-1]
Calculate a^(n-1) mod n
if a^(n-1) mod n = 1
    then n is possibly prime
    else n is definitely not prime
```

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

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), test 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:

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

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

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

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

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

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

Fast exponentiation

```
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

```
/* 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: $O(\log^2 n \log p)$

Randomized primality test 2

```
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;
}
```

Randomized primality test 2

Theorem:

If *n* is not prime, there are at most

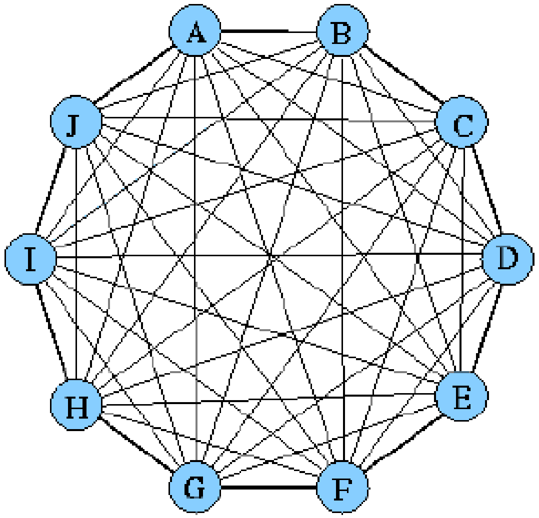
integers $0 < a < n$, for which the algorithm primalityTest fails.

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

$$P_A, S_A : D \to D$$

Three conditions:

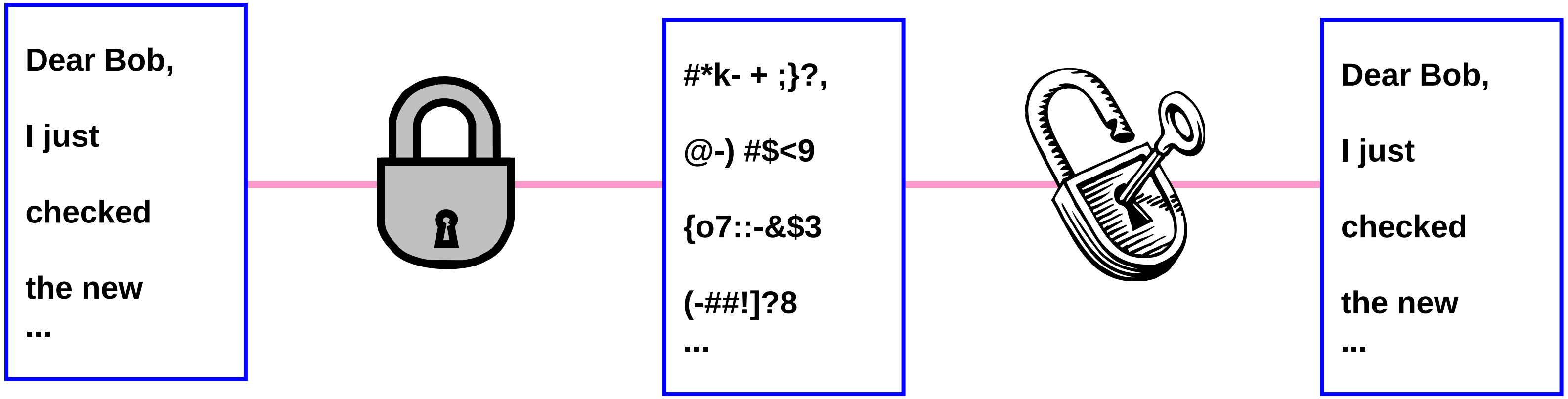
1. P_A and S_A can be computed efficiently

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 (without unreasonable 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 an 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: $0 \le M < 2^{2 \cdot l}$

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