Algorithmentheorie

03 - Randomisierung

Prof. Dr. S. Albers

Randomisierung

- Klassen von randomisierten Algorithmen
- Randomisierter Quicksort
- Randomisierter Primzahltest
- Kryptographie

1. Klassen von randomisierten Algorithmen

- **Las Vegas Algorithmen**
  immer korrekt; erwartete Laufzeit

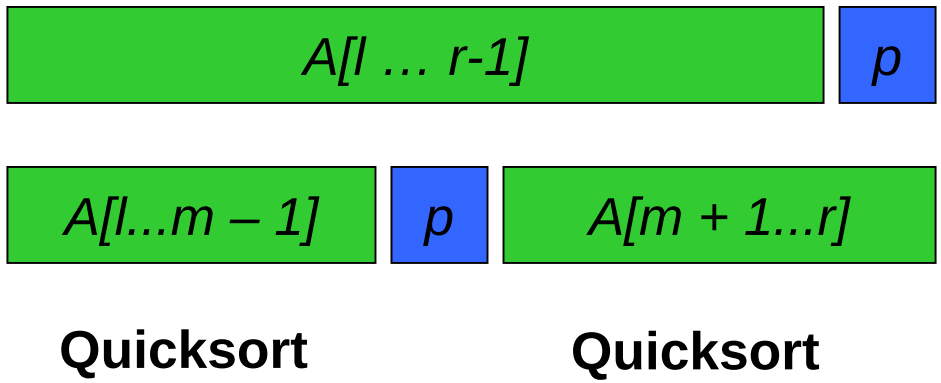
  Beispiel: randomisierter Quicksort

- **Monte Carlo Algorithmen** (most correctly):
  wahrscheinlich korrekt; garantierte Laufzeit

  Beispiel: randomisierter Primzahltest

2. Quicksort

Unsortierter Bereich A[l,r] in einem Array A



Quicksort

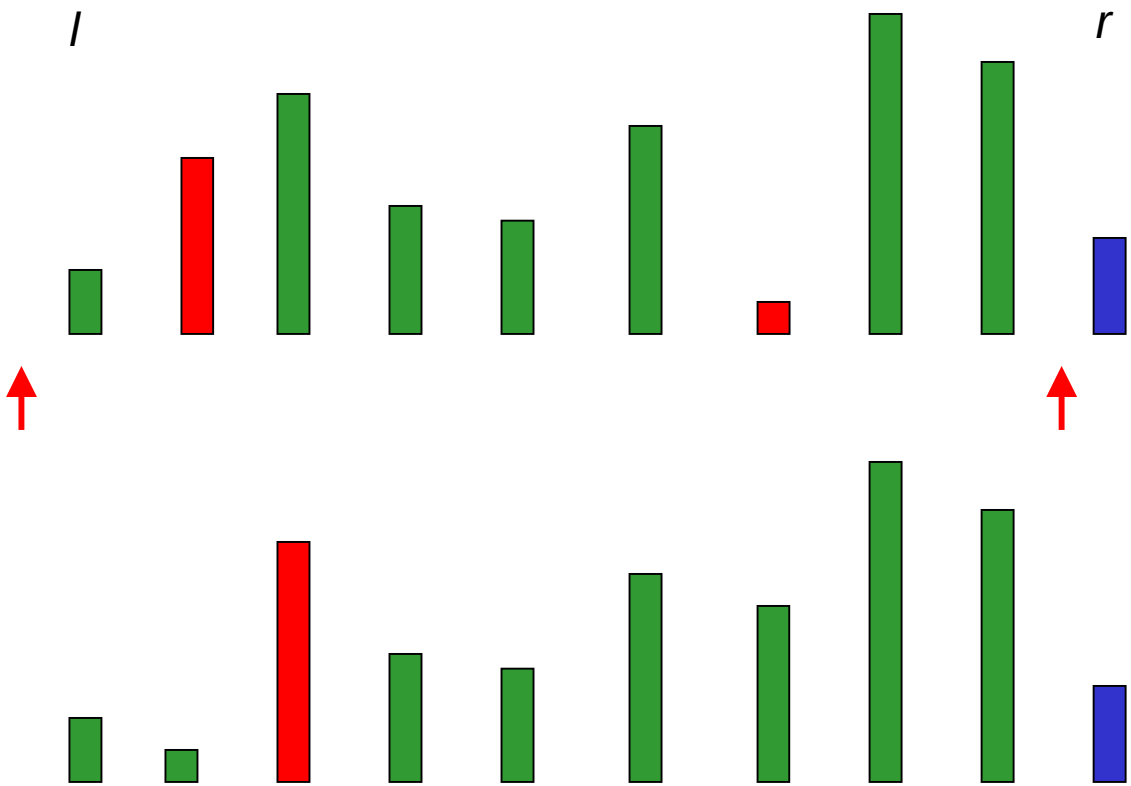
Algorithmus: *Quicksort*

Input: unsortierter Bereich [*l, r*] in Array *A*
Output: sortierter Bereich [*l, r*] in Array *A*

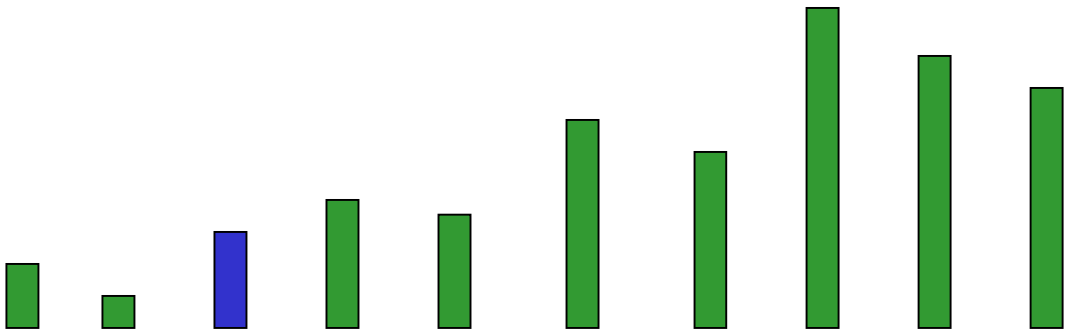
1 **if** $r > l$
2 **then** wähle Pivotelement $p = A[r]$
3 $m = divide(A, l, r)$
/* Teile *A* bzgl. *p* auf:
$A[l],....,A[m-1] \leq p \leq A[m+1],...,A[r]$
*/

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

Der Aufteilungsschritt



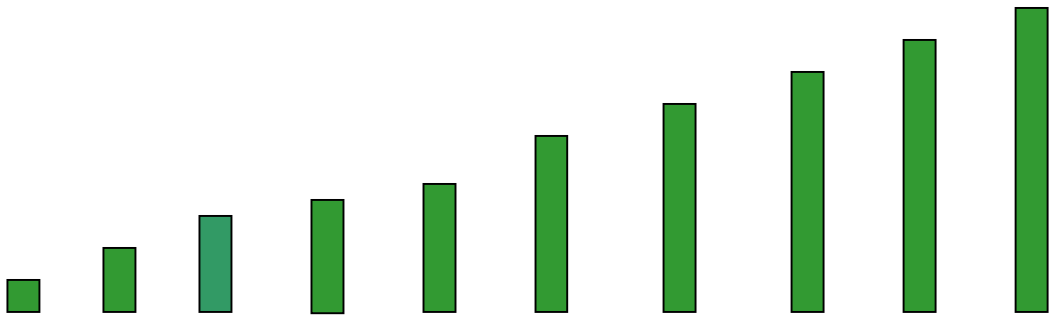
Der Aufteilungsschritt



divide(*A, l , r*):

- liefert den Index des Pivotelements in *A*
- ausführbar in Zeit *O(r – l)*

Worst-Case-Eingabe



n Elemente*:*

Laufzeit: *(n-1) + (n-2) + … + 2 + 1 = n(n-1)/2*

3. Randomisierter Quicksort

Algorithmus: Quicksort

Input: unsortierter Bereich [*l*, *r*] in Array *A*

Output: sortierter Bereich [*l*, *r*] in Array *A*

```
1   if   r > l
2        then wähle zufälliges Pivotelement p = A[i]  im Bereich [l, r]
3             tausche A[i] mit A[r]
4             m = divide(A, l, r)
              /* Teile A bzgl. p auf:
                 A[l],....,A[m – 1] ≤ p ≤ A[m + 1],...,A[r]
              */
5             Quicksort(A, l, m - 1)
6             Quicksort(A, m + 1, r)
```

Analyse 1

n Elemente; sei S_i das *i*-t kleinste Element

Mit WSK *1/n* ist S_1 Pivotelement:
Teilprobleme der Größen *0* und *n-1*

-
-
-

Mit WSK *1/n* ist S_k Pivotelement:
Teilprobleme der Größen *k-1* und *n-k*

-
-
-

Mit WSK *1/n* ist S_n Pivotelement:
Teilprobleme der Größen *n-1* und *0*

Analyse 1

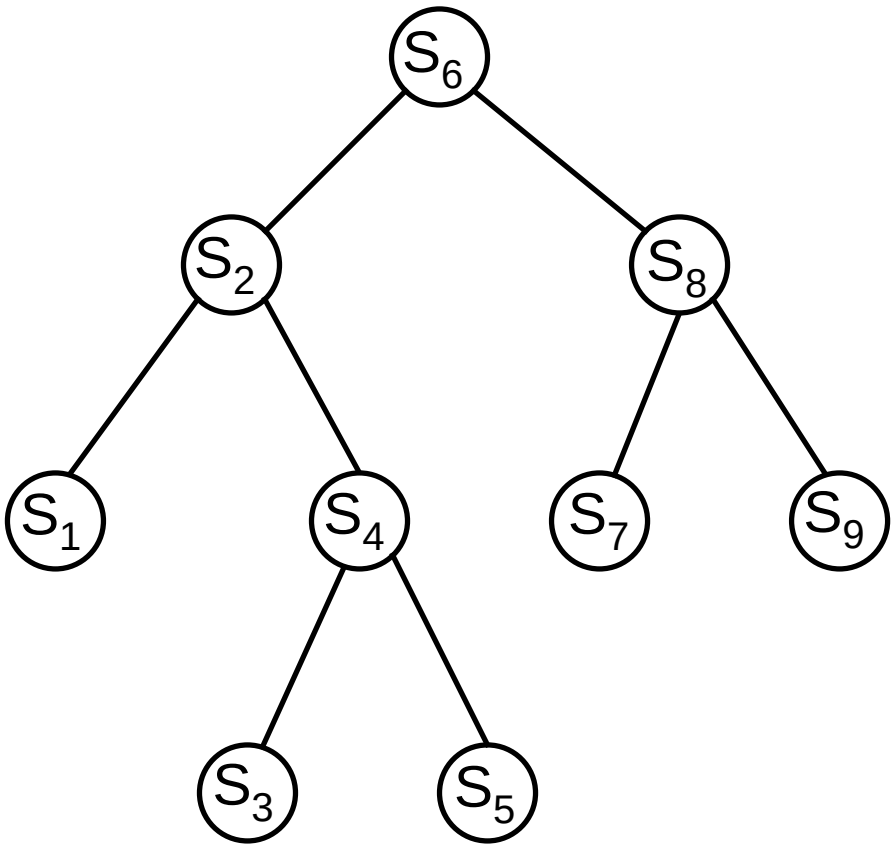
Erwartungswert für die Laufzeit:

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

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

$$= O(n\log n)$$

Analyse 2: Darstellung von QS als Baum



$\pi = S_6\ S_2\ S_8\ S_1\ S_4\ S_7\ S_9\ S_3\ S_5$

Analyse 2

Erwartete Anzahl der Vergleiche:

$$X_{ij} = \begin{cases} 1 & \text{falls } S_i \text{ mit } S_j \text{ verglichen wird} \\ 0 & \text{sonst} \end{cases}$$

$$E\left[\sum_{i=1}^{n} \sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j>i} E[X_{ij}]$$

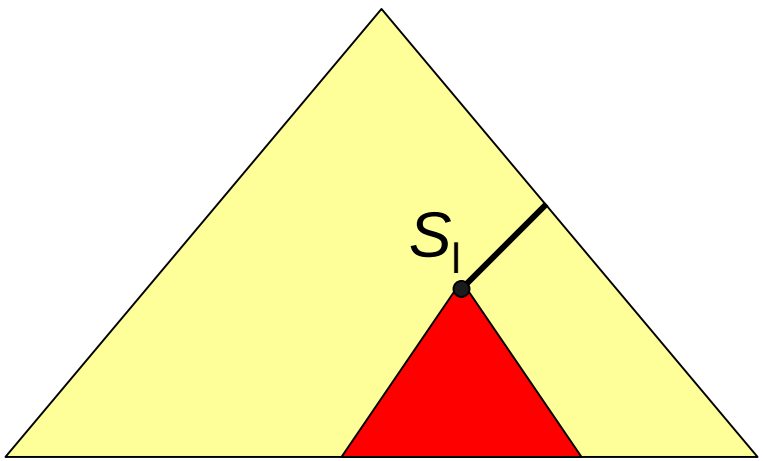
p_{ij} = WSK S_i wird mit S_j verglichen

$$E[X_{ij}] = 1 \times p_{ij} + 0 \times (1 - p_{ij}) = p_{ij}$$

Berechnung von p_{ij}

- S_i wird mit S_j verglichen gdw S_i oder S_j in π vor allen anderen S_l , $i<l<j$, als Pivotelement gewählt wird.
  $\{S_i \ \ldots \ S_l \ \ldots \ S_j\}$

- Jedes der Elemente S_i , … , S_j wird mit gleicher WSK als erstes als Pivotelement gewählt.



$\{\ldots S_i \ \ldots \ S_l \ \ldots \ S_i \ \ldots\}$

$$p_{ij} = 2/(j-i+1)$$

Analyse 2

Erwartete Anzahl der Vergleiche:

$$\sum_{i=1}^{n} \sum_{j>i} p_{ij} = \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k}$$

$$\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k}$$

$$= 2n \sum_{k=1}^{n} \frac{1}{k}$$

$$H_n = \sum_{k=1}^{n} 1/k \approx \ln n$$

4. Primzahltest

Definition:

Zahl $p \geq 2$ ist genau dann prim, wenn aus $a \mid p$ folgt $a = 1$ oder $a = p$.

Wir betrachten Primzahltests für Zahlen $n \geq 2$.

Algorithmus: Deterministischer Primzahltest (naiv)

Input: Eine ganze Zahl $n \geq 2$

Output: Antwort auf die Frage: Ist n prim?

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

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

Primzahltest

Ziel:

Randomisiertes Verfahren

- in polynomieller Zeit ausführbar
- falls Ausgabe “nicht prim”, dann ist *n* nicht prim
- falls Ausgabe “prim”, dann ist *n* nicht prim höchstens mit Wahrscheinlichkeit *p>0*

k Iterationen: *n* ist nicht prim mit Wahrscheinlichkeit p^k

Primzahltest

Fakt: Jede ungerade Primzahl p teilt $2^{p-1} - 1$.

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

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

Einfacher Primzahltest:

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

2 if $z = 1$

3 then n ist möglicherweise prim

4 else n ist definitiv nicht prim

Vorteil: Benötigt nur polynomielle Zeit

Einfacher Primzahltest

Definition:

Zahl $n \geq 2$ ist eine Pseudoprimzahl zur Basis 2, wenn n nicht prim und

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

Beispiel: $n = 11 * 31 = 341$

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

Randomisierter Primzahltest

Satz: (kleiner Fermat)

Ist p prim und $0 < a < p$, dann ist

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

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

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

Definition:

Zahl $n \geq 2$ ist eine Pseudoprimzahl zur Basis a, wenn n nicht prim und

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

Randomisierter Primzahltest

Algorithmus: Randomisierter Primzahltest 1

1 Wähle a im Bereich [2, n-1] zufällig
2 Berechne a^{n-1} mod n
3 **if** a^{n-1} mod n = 1
4 **then** n ist möglicherweise prim
5 **else** n ist definitiv nicht prim

Prob(n ist nicht prim, aber a^{n-1} mod n = 1) ?

Carmichael- Zahlen

Problem: Carmichael-Zahlen

Definition: Eine Zahl $n \geq 2$ heißt Carmichael-Zahl, falls sie nicht prim ist und für alle a mit ggT(a, n) = 1

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

Beispiel:

Kleinste Carmichael-Zahl: 561 = 3 * 11 * 17

Randomisierter Primzahltest 2

Satz: Ist p prim und $0 < a < p$, dann hat die Gleichung

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

genau die Lösungen $a = 1$ und $a = p - 1$.

Definition: a heißt nichttriviale Quadratwurzel mod n, falls

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

Beispiel: $n = 35$

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

Schnelle Exponentiation

Idee:

Teste während der Berechnung von a^{n-1}, $0 < a < n$ zufällig, ob es eine nichttriviale Quadratwurzel mod n gibt.

Verfahren zur Berechnung von a^n:

Fall 1: [n ist gerade]

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

Fall 2: [n ist ungerade]

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

Schnelle Exponentiation

Beispiel:

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

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

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

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

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

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

Schnelle Exponentiation

```
boolean isProbablyPrime;

power(int a, int p, int n){

    /* berechnet a^p mod n und prüft , ob bei der Berechnung
       ein x auftritt mit x^2 mod n = 1 und x ≠ 1, n-1 */

        if (p == 0) return 1;

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

Schnelle Exponentiation

```
/* prüfe ob x^2 mod n = 1 und x ≠ 1, n-1 */

        if (result == 1 && x != 1 && x != n –1 )
        isProbablyPrime = false;

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

    return result;
}
```

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

Randomisierter Primzahltest 2

```
primeTest(int n) {
      /* führt den randomisierten Primzahltest für ein zufällig a aus */

a = random(2, n-1);

isProbablyPrime = true;

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

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

}
```

Randomisierter Primzahltest 2

Satz:

Ist *n* nicht prim, so gibt es höchstens

$$\frac{n-9}{4}$$

Zahlen $0 < a < n$, für die Algorithmus primeTest versagt.

5. Exkurs

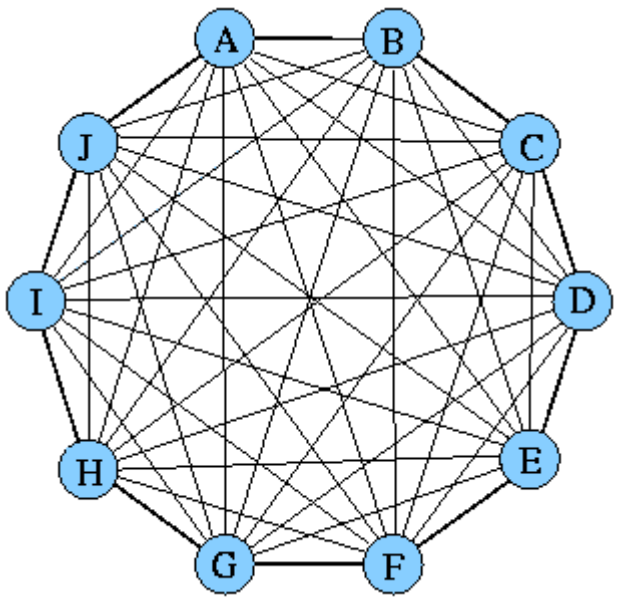
Public Key Verschlüsselungssysteme

Verschlüsselungssysteme mit geheimen Schlüsseln

Traditionelle Verschlüsselung von Nachrichten

Nachteile:

1. Schlüssel *k* muss zwischen A und B vor Übersendung der Nachricht ausgetauscht werden.
2. Für Nachrichten zwischen *n* Parteien sind $n(n-1)/2$ Schlüssel erforderlich.



Verschlüsselungssysteme mit geheimen Schlüsseln

Vorteile:

Ver- und Entschlüsselung ohne spürbaren Geschwindigkeitsverlust

Aufgabe von Sicherheitsdiensten

Gewährleistung von

- Vertraulichkeit der Übertragung
- Integrität der Daten
- Authentizität des Senders
- Verbindlichkeit der Übertragung

Public-Key Verschlüsselungssysteme

Diffie and Hellman (1976)

Idee: Jeder Teilnehmer *A* besitzt zwei Schlüssel:

1. einen öffentlichen Schlüssel P_A, der jedem anderen Teilnehmer zugänglich gemacht wird.
2. einen geheimen Schlüssel S_A, der nur *A* bekannt ist.

Public-Key Verschlüsselungssysteme

D = Menge der zulässigen Nachrichten,
z.B. Menge aller Bitstrings endl. Länge

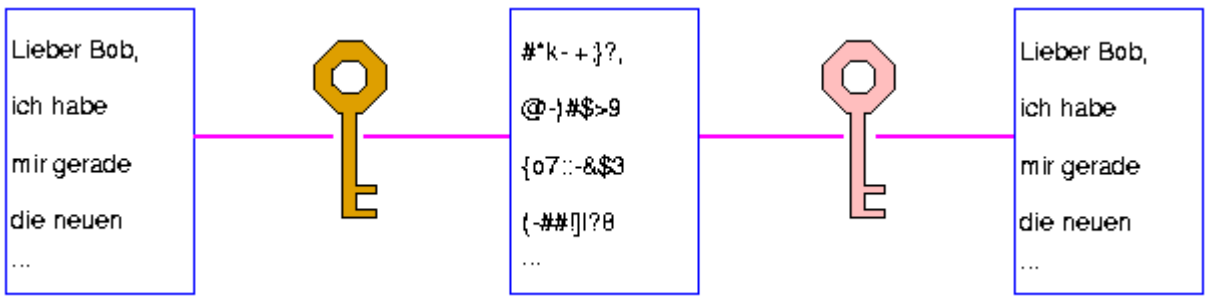
$$P_A(\ ), S_A(\ ): D \xrightarrow{1\text{–}1} D$$

Drei Bedingungen:

1. $P_A()$, $S_A()$ effizient berechenbar
2. $S_A(P_A(M)) = M$ und $P_A(S_A(M)) = M$
3. $S_A()$ nicht aus $P_A()$ zu berechnen (mit realisierbarem Aufwand)

Verschlüsselung im Public-Key System

A schickt eine Nachricht *M* an *B*:



Verschlüsselung im Public-Key System

1. *A* verschafft sich *B`s* öffentlichen Schlüssel P_B (aus einem öffentlichen Verzeichnis oder direkt von *B*).

2. *A* berechnet die chiffrierte Nachricht $C = P_B(M)$
   und sendet *C* an *B.*

3. Nachdem *B* die Nachricht *C* empfangen hat,
   dechiffriert *B* die Nachricht mit seinem geheimen Schlüssel S_B: $M = S_B(C)$

Erstellen einer digitalen Unterschrift

A schickt eine digital unterzeichnete Nachricht *M´* an *B*:

1. *A* berechnet die digitale Unterschrift σ für *M´* mit Hilfe des geheimen Schlüssels:

   $$\sigma = S_A(M´)$$

2. *A* schickt das Paar (*M´*, σ) an *B.*

3. Nach Erhalt von (*M´*, σ) überprüft *B* die digitale Unterschrift:

   $P_A(\sigma) = M´$

σ kann von jedem mit P_A überprüft werden (z.B. für Schecks).

RSA-Verschlüsselungssysteme

R. Rivest, A. Shamir, L. Adleman

Erstellen der öffentlichen und geheimen Schlüssel

1. Man wähle zufällig zwei etwa gleich große Primzahlen p und q mit je $l+1$ Bits ($l \geq 500$).

2. Sei $n = pq$

3. Sei e eine natürliche Zahl, die teilerfremd zu $(p-1)(q-1)$ ist.

4. Berechne $d = e^{-1}$

$$d*e \equiv 1 \pmod{(p-1)(q-1)}$$

RSA-Verschlüsselungssysteme

5. Veröffentliche $P = (e, n)$ als öffentlichen Schlüssel

6. Behalte $S = (d, n)$ als geheimen Schlüssel

Zerlege Nachricht (binär kodiert) in Blöcke der Länge $2l$.
Fasse jeden Block M als Binärzahl auf: $0 \leq M < 2^{2l}$

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

Multiplikative Inverse

Satz: (ggT-Rekursions-Satz)

Für alle Zahlen *a* und *b* mit *b*>0 gilt

$$ggT(a,b) = ggT(b, a \bmod b).$$

Algorithmus: Euklid

Input: Zwei ganze Zahlen *a* und *b* mit $b \geq 0$

Output: ggT(*a*,*b*)

```
if b = 0
then return a
else return Euklid(b, a mod b)
```

Multiplikative Inverse

Algorithmus: Erw-Euklid

Input: Zwei ganze Zahlen a und b mit $b \geq 0$

Output: ggT(a,b) und zwei ganze Zahlen x und y mit

$$xa + yb = \text{ggT}(a,b)$$

if $b = 0$ **then return** (a, 1, 0);

(d, x', y') := Erw-Euklid(b, a mod b);

$x := y'$; $y := x' - \lfloor a/b \rfloor y'$;

return (d, x, y);

Anwendung: $a=(p-1)(q-1)$ $b=e$

Zahlen x und y mit

$$x(p-1)(q-1) + ye = \text{ggT}((p-1)(q-1),e) = 1$$