Rechnen mit Zahlen großer Stellenzahl

- eine **exakte Arithmetik** arbeitet auf Zahlen mit beliebig hoher Genauigkeit
  $\Rightarrow$ Einsatz von Integer- und Festkommazahlen mit beliebiger Stellenzahl n
- alle in Kap. 2 vorgestellten Algorithmen können auch für Zahlen beliebig großer Stellenzahl implementiert werden:
  - **Addition** von zwei n-stelligen Zahlen ergibt eine $(n+1)$-stellige Summe
  - **Multiplikation** einer m-stelligen Zahl mit einer n-stelligen Zahl ergibt ein $(m+n)$-stelliges Produkt
  - **Division** einer $(m+n)$-stelligen Zahl durch einen m-stelligen Divisor ergibt einen $(m+1)$-stelligen Quotienten und einen n-stelligen Rest
- Rechner haben nur begrenzte Wortbreite w, große Zahlen $x > 2^w$ können aber als Zahlen zur Basis $b = 2^w$ betrachtet werden!

  Beispiel für $w = 16$: $1000001.015625 = 15 \times 2^{16} + 16961 \times 2^0 + 1024 \times 2^{-16}$
  $\Rightarrow$ interne Abspeicherung als Zahl mit 3 Stellen zur Basis 2^{16}

Addition bei großer Stellenzahl

- Seien $x = (x_{n-1} \ldots x_2\ x_1\ x_0)_b$ und $y = (y_{n-1} \ldots y_2\ y_1\ y_0)_b$ zwei ganze n-stellige Zahlen zur Basis $b = 2^w$
- Algorithmus zur Berechnung von $s = x + y = (s_n\ s_{n-1} \ldots s_2\ s_1\ s_0)_b$:

```
c=0
for j = 0,...,n-1 {
  sj = (xj + yj + c) mod 2^w
  c = (xj + yj + c) / 2^w
}
sn = c
```

- Es gilt stets: $(x_j + y_j + c) \le 2b-1 \Rightarrow$ `c=0` oder `c=1`
- sehr einfach in Maschinensprache zu realisieren, wenn eine „*Add with Carry*“ Instruktion implementiert ist
- mit kleinen Modifikationen ist obiger Algorithmus auch für die Subtraktion verwendbar!

Multiplikation bei großer Stellenzahl

- Seien $x = (x_{m-1} \ldots x_2\, x_1\, x_0)_b$ und $y = (y_{n-1} \ldots y_2\, y_1\, y_0)_b$ zwei ganze m- bzw. n-stellige Zahlen zur Basis $b = 2^w$
- Algorithmus zur Berechnung von $p = x \cdot y = (p_{m+n-1} \ldots p_2\, p_1\, p_0)_b$:

```
p=0
for j = 0,...,n-1 {
  c=0
  for i = 0,...,m-1 {
    t = (x_i · y_j + p_{i+j} + c)
    p_{i+j} = t mod 2^w
    c = t / 2^w
  }
  p_{m+j} = c
}
```

- Es gilt stets: $0 \le t < b^2$, $0 \le c < b$
- Zeitaufwand: mn Multiplikationen, $2mn$ Additionen

Multiplikation bei großer Stellenzahl (Forts.)

- Algorithmus auf letzter Folie benötigt ein Laufzeit T_n von **$O(n^2)$** zur Berechnung des Produktes zweier n-stelliger Zahlen
  $\Rightarrow$ Gibt es auch **schnellere** Algorithmen?
- Idee: Zurückführung der Multiplikation zweier $2n$-stelliger Zahlen $x = 2^n \cdot x_1 + x_0$ und $y = 2^n \cdot y_1 + y_0$ auf Multiplikationen von n-stelligen Zahlen x_i und y_i:

  $$x \cdot y = 2^{2n} \cdot x_1 \cdot y_1 + 2^n (x_1 \cdot y_0 + x_0 \cdot y_1) + x_0 \cdot y_0$$

  Hier werden **vier Teilmultiplikationen** benötigt; jedoch kann man durch Umformungen folgende Formel herleiten:

  $$x \cdot y = (2^{2n} + 2^n) \cdot x_1 \cdot y_1 + 2^n \cdot (x_1 - x_0) \cdot (y_0 - y_1) + (2^n + 1) \cdot x_0 \cdot y_0$$

  Hier sind nur noch **drei Teilmultiplikationen** erforderlich, jedoch drei zusätzliche Additionen!
- Man kann zeigen, dass mit letzter Formel rekursiv angewandt nur eine Laufzeit T_n von **$O(n^{\mathrm{ld}\,3}) \approx O(n^{1.585})$** benötigt wird!

Multiplikation bei großer Stellenzahl (Forts.)

- seien n eine 2er-Potenz, w die Wortbreite des Zielrechners und x sowie y zwei beliebige n-stellige Festkommazahlen
- rekursiver Algorithmus:

```
fastmult(x, y, n) {
  erzeuge 2n-stellige Variable p;
  if (n ≤ w)
    p = x * y;
  else
    p_{2n-1,...,n} = fastmult(x_{n-1,..,n/2}, y_{n-1,..,n/2}, n/2);
    p_{n-1,...,0} = fastmult(x_{n/2-1,..,0}, y_{n/2-1,..,0}, n/2);
    p = p + ((p_{2n-1,...,n} + p_{n-1,...,0}) << n/2);
    p = p + ((fastmult((x_{n-1,..,n/2} - x_{n/2-1,..,0}),
              (y_{n/2-1,..,0} - y_{n-1,..,n/2}), n/2) << n/2);
    return(p);
}
```

- es gibt noch schnellere Algorithmen! (Laufzeit $T_n < c(\varepsilon) \cdot n^{1+\varepsilon}$)

Rechnen mit großer Stellenzahl in Java

- In Java können **beliebig große Integer-Zahlen** mit der Klasse `BigInteger` implementiert werden (in `java.math`)
  - interne Speicherung als Binärzahl im Zweierkomplement
- einige **Methoden** der Klasse `BigInteger`:
  - Konstruktoren: `BigInteger (String s)` für Dezimalzahlen, `BigInteger (String s , int b)` für Zahlen zur Basis b
  - arithmetische Operationen: `BigInteger add (BigInteger x)`, `BigInteger multiply (BigInteger x)`, `BigInteger divide (BigInteger x)`, `BigInteger pow (int n)`, ...
  - Vergleiche: `boolean equals (Object x)`, `BigInteger min (BigInteger x)`, `BigInteger max (BigInteger x)`
  - Konvertierung: `int intValue()`, `long longValue ()`, `String toString (int b)`, `double doubleValue()`, `BigInteger valueOf (long x)`

Rechnen mit großer Stellenzahl in Java (Forts.)

- zur Darstellung **beliebig genauer Festkommazahlen** existiert in Java die Klasse `BigDecimal` (in `java.math`)
  - interne Speicherung durch beliebig große Integerzahl `I` und einen dezimalen (!) Skalierungsfaktor `scale`
  - Wert einer `BigDecimal` Zahl: $I / 10^{scale}$
- einige **Methoden** der Klasse `BigDecimal`:
  - Konstruktoren: `BigDecimal (BigInteger I, int scale)`, `BigDecimal (double x)`, `BigDecimal (String s)`
  - arithmetische Operationen mit Erhöhung der Präzision: `add (BigDecimal x)`, `multiply (BigDecimal x)`
  - Reduktion der möglichen Präzision: `divide (BigDecimal x, int scale, int roundMode)` `setScale (int scale, int roundingMode)`
  - exakte Ausgabe mittels `toString()`
  - elementare Funktionen wie `exp`, `log`, `sqrt` sind nicht definiert!

Rechnen mit großer Stellenzahl in Java (Forts.)

Beispiel: Genaue Berechnung von e mit MacLaurin-Reihe in Java

```
final int NP = 40;  // number of desired decimal places
BigDecimal one = new BigDecimal("1");
BigDecimal fact = new BigDecimal("1");
BigDecimal factmul = new BigDecimal("1");
BigDecimal sum = new BigDecimal("0");
String s = "";
for (;;) {
  // divide 1 by the current factorial and accumulate the result
  BigDecimal x = one.divide(fact, NP+1, BigDecimal.ROUND_HALF_EVEN);
  sum = sum.add(x);
  // move to the next factorial value
  fact = fact.multiply(factmul);
  factmul = factmul.add(one);
  // check convergence of the current value
  String e = sum.toString().substring(0, NP + 2);
  if (e.equals(s)) {
    System.out.println(e);
    break;
  }
  s = e;
}
```