Objektorientierte Programmierung mit C++ (WS 2001/2002)

Abgabetermin: 22. November 2001

Beispiellösung

6 Endlich fertig! (10 Punkte)

Sie haben erfolgreich Ihr Studium beendet und sind froh, es endlich geschafft zu haben. Gratuliere! Das ist ja auch eine Leistung. Nun geht's endlich los mit dem Berufsleben. Am ersten Arbeitstag begrüßt Sie Ihr Chef gleich persönlich und sagt: „ Schön, dass Sie da sind. Wir haben da nämlich ein Problem für einen Mathematiker mit Programmierkenntnissen. Und da hab' ich natürlich sofort an Sie gedacht. Und außerdem haben Sie ja auch von Objektorientierung eine Ahnung. Dann ist das Ganze für Sie ja ein Kinderspiel, nicht wahr! Also: Wir haben tausend verschiedene Stellplätze in unserem Lager. Wieviele Möglichkeiten gibt es eigentlich, unsere Waren zu lagern, wenn auf jedem einem Platz genau eine Ware lagert? Ich hab' von meinem Studium da noch so 'ne dunkle Ahnung, dass das wohl etwas mit der Fakultät zu tun hat – nein, nicht mit der für Mathematik, oder so, sondern mit der mathematischen! Na, das kriegen Sie doch locker raus. Außerdem haben wir noch zwei ganz große Zahlen:

```
a = 437126568088199322221531093992681680292255135889347272
    793142525738932129765481395774330739364283702105495918
    167180870335630234794888178058034034942524901390649773
    135897528619263112248521
b = 856572819620880044442447278942064239828036082950049805
    477762946438216943311117552526407306054570280412531298
    828437326941230867502237021161939914313647243498788655
    858622732834457132989677148326203924560311074871462722
    389629243239810751013863939268822009572313261108536188
    711684947825436264030303664622617392013714687318464880
    497281240722722445776663586579288181928821514478734370
    7864038601
```

Von diesen Zahlen brauchen wir unbedingt den ggT. Sie müssen wissen, dass das Rechnen mit großen Zahlen ein Steckenpferd von mir ist. Na, dann schreiben Sie doch mal ein kleines C++-Programm mit dem wir dann die beiden Problemchen lösen können.“ Na, war das Studium

nicht schön, da sind Sie noch vor solchen Problemen verschont gewesen, aber Sie wollten ja unbedingt schon arbeiten. Tja, da kann man wohl nichts machen. Ach ja, beim Hinausgehen sagt Ihr Chef noch zu Ihnen, dass Sie sich doch überlegen sollten, wieviele Nullen die Zahl 1000! am Ende besitzt. Wenn Sie Ihm dies erklären könnten, wäre er Ihnen sehr dankbar. Das wäre doch die Gelegenheit, bei Ihrem neuen Chef (Tutor) Eindruck zu schinden.
Sie erinnern sich jetzt doch noch an die C++-Vorlesung von Matthias Grabert. Dort haben Sie doch gelernt, wie man sogar Operatoren für eigene Datentypen „überladen" kann. Na das bietet sich doch für diesen Fall perfekt an. Da schreiben Sie einen Datentyp für beliebig große Integer-Zahlen mit den zugehörigen Operatoren. Damit können Sie dann die beiden Problemchen lösen. Und außerdem macht das sicher einen sehr guten Eindruck auf Ihren neuen Chef, der dann ein wenig mit großen Zahlen herumspielen kann. Dazu müssen Sie aber mindestens die 4 Grundrechenarten implementiert haben, also Addition, Subtraktion, Multiplikation und Division. Na, vielleicht klappt's dann ja schon bald mit der ersten Gehaltsaufbesserung. Also, machen Sie sich am besten gleich an die Arbeit. Das schöne ist ja, dass sie in einem Team arbeiten. Ihre Kollegen warten doch sicher schon darauf, dass Sie ihnen etwas zu tun geben. Also verteilen Sie gleich mal die Aufgaben in Ihrem Team, dann geht's schneller und Sie machen sich sofort Freunde unter den neuen Kollegen! :-))
P.S.: Ihr Chef sieht Ihre erste Aufgabe wohl als Test Ihrer Fähigkeiten an und würde Ihnen wohl keine Punkte geben, wenn sie nicht das komplette Programm selbst schreiben! ;-)

Lösung:

```
1000! = 402387260077093773543702433923003985719374864210714632543799910
        429938512398629020592044208486969404800479988610197196058631666
        872994808558901323829669944590997424504087073759918823627727188
        732519779505950995276120874975462497043601418278094646496291056
        393887437886487337119181045825783647849977012476632889835955735
        432513185323958463075557409114262417474349347553428646576611667
        797396668820291207379143853719588249808126867838374559731746136
        085379534524221586593201928090878297308431392844403281231558611
        036976801357304216168747609675871348312025478589320767169132448
        426236131412508780208000261683151027341827977704784635868170164
        365024153691398281264810213092761244896359928705114964975419909
        342221566832572080821333186116811553615836546984046708975602900
        950537616475847728421889679646244945160765353408198901385442487
        984959953319101723355556602139450399736280750137837615307127761
        926849034352625200015888535147331611702103968175921510907788019
        393178114194545257223865541461062892187960223838971476088506276
        862967146674697562911234082439208160153780889893964518263243671
        616762179168909779911903754031274622289988005195444414282012187
        361745992642956581746628302955570299024324153181617210465832036
        786906117260158783520751516284225540265170483304226143974286933
        061690897968482590125458327168226458066526769958652682272807075
        781391858178889652208164348344825993266043367660176999612831860
        788386150279465955131156552036093988180612138558600301435694527
```

```
224206344631797460594682573103790084024432438465657245014402821
885252470935190620929023136493273497565513958720559654228749774
011413346962715422845862377387538230483865688976461927383814900
140767310446640259899490222221765904339901886018566526485061799
702356193897017860040811889729918311021171229845901641921068884
387121855646124960798722908519296819372388642614839657382291123
125024186649353143970137428531926649875337218940694281434118520
158014123344828015051399694290153483077644569099073152433278288
269864602789864321139083506217095002597389863554277196742822248
757586765752344220207573630569498825087968928162753848863396909
959826280956121450994871701244516461260379029309120889086942028
510640182154399457156805941872748998094254742173582401063677404
595741785160829230135358081840096996372524230560855903700624271
243416909004153690105933983835777939410970027753472000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
```

Am Ende von 1000! befinden sich genau 249 Nullen. Dies kann man berechnen, indem man die Anzahl des Faktors 10 in $1000! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot 999 \cdot 1000$ bestimmt. Es gilt $10 = 2 \cdot 5$. Also muß man Zählen, wie oft der Faktor 5 vorkommt und wie oft der Faktor 2 vorkommt. Das Minimum dieser beiden Anzahlen ist dann die Häufigkeit des Faktors 10 und somit die Anzahl der Nullen am Ende. Der Faktor 5 kommt $1000/5 = 200$ mal einmal vor, $1000/25 = 40$ mal noch ein zweites Mal, $\lfloor 1000/125 \rfloor = 8$ mal noch ein drittes Mal und $\lfloor 1000/625 \rfloor = 1$ mal zu guter Letzt noch ein viertes Mal. Der Faktor 5 kommt also genau 249-mal vor. Der Faktor 2 kommt analog zur obigen Berechnung öfters vor. Also ist die Anzahl der Nullen am Ende 249! Der größte gemeinsame Teiler von den beiden Zahlen a und b ist 823. Die Zahlen a und b sind unterschiedliche Mersenne-Primzahlen multipliziert mit 823.

main.cc

```
#include <iostream>
#include "bigint.h"
#include "recursion.h"

int main() {
    BigInteger fak(1);
    for (int i = 1; i <= 1000; i++)
        fak = fak * i;
    cout << "1000! = " << fak << endl;

/*
    // Erzeugung der beiden Zahlen a und b

    // a ist Mersenne-Primzahl 2^607 - 1
    // multipliziert mit 823
    BigInteger a(1);
```

```
    for (int i = 1; i <= 607; i++)
        a = a * *new BigInteger(2);
    a = a - *new BigInteger(1);
    a = a * *new BigInteger(823);
    cout << "a = " << a << endl;

    // a ist Mersenne-Primzahl 2^1279 - 1
    // multipliziert mit 823
    BigInteger b(1);
    for (int i = 1; i <= 1279; i++)
        b = b * *new BigInteger(2);
    b = b - *new BigInteger(1);
    b = b * *new BigInteger(823);
    cout << "b = " << b << endl;
*/

    BigInteger a("43712656808819932222153109399268168029225513588934727 2"
                 "79314252573893212976548139577433073936428370210549591 8"
                 "16718087033563023479488817805803403494252490139064977 3"
                 "1358975286192631122485 21"),
               b("85657281962088004444244727894206423982803608295004980 5"
                 "47776294643821694331111755252640730605457028041253129 8"
                 "82843732694123086750223702116193991431364724349878865 5"
                 "85862273283445713298967714832620392456031107487146272 2"
                 "38962924323981075101386393926882200957231326110853618 8"
                 "71168494782543626403030366462261739201371468731846488 0"
                 "49728124072272244577666358657928818192882151447873437 0"
                 "7864038601"),
               c;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "ggT(a, b) = " << ggt(a,b) << endl;

#ifdef EXP
    // Berechnung von 2^2^17
    BigInteger exp(2);
    cout << "i = 0: " << exp << endl;
    for (int i = 1; i <= 17; i++) {
        exp = exp * exp;
        cout << "i = " << i << ": " << exp << endl;
    }
#endif

    // Und noch ein kleiner Rekursions-Test ...

#ifdef FIB
    for (int i = 0; i < 100; i++)
        cout << "fib(" << i << ") = " << fib((BigInteger) i) << endl;
```

```
#endif

#ifdef ACK
    for (int x = 0; x < 5; x++)
        for (int y = 0; y < 5; y++)
            cout << "ackermann(" << x << ", " << y << ") = " << ackermann((BigInteger) x, (BigInt
#endif

    return 0;
}
```

bigint.h

```
#ifndef _BIGINT_H

#include <iostream>

class BigInteger {
private:
    char *digits;       // Array mit den Ziffen (digit[0] ist die niederwertigste)
    int maxlen;         // Laenge des Arrays digits
    int len;            // Laenge der aktuellen Zahl (mit len <= maxlen)
    bool negative;      // ist die Zahl negativ?

    // konstruiert Ziffernarray NEU mit Groesse >= n
    void constructSize(int n);
    // Ziffernarray evtl. neu mit Groesse >= n anlegen
    void ensureSize(int n);
    // Zahl normalisieren: +0 statt -0 und fuehrende Nullen entfernen
    void normalize();

    friend ostream &operator<<(ostream &out, BigInteger number);
public:
    BigInteger(int number = 0);         // Init-Werte im Header-File angeben!!!
    // BigInteger-Zahl einfach kopieren
    BigInteger(const BigInteger &number);
    // BigInteger aus String erzeugen
    // evtl. Exception, falls ein ungueltiges Zeichen vorkommt
    BigInteger(char *number);

    ~BigInteger();

    // konvertierung in eine int (funktioniert nur bei kleinen Zahlen!)
    int toInt() const;

    BigInteger &operator=(const BigInteger &number);
    BigInteger &operator=(int number);
    BigInteger &operator=(char *number);

    bool operator==(const BigInteger &number) const;
```

```
    bool operator!=(const BigInteger &number) const;
    bool operator<(const BigInteger &number) const;
    bool operator<=(const BigInteger &number) const;
    bool operator>(const BigInteger &number) const;
    bool operator>=(const BigInteger &number) const;

    BigInteger operator-() const;

    BigInteger operator+(const BigInteger &number) const;
    BigInteger operator-(const BigInteger &number) const;
    BigInteger operator*(int number) const;
    BigInteger operator*(const BigInteger &number) const;
    // Dezimal-Zahl um n Stellen nach rechts schieben
    // d.h. Division durch 10^n

    BigInteger operator>>(int n) const;
    // Dezimal-Zahl um n Stellen nach links schieben
    // d.h. Multiplikation mit 10^n
    BigInteger operator<<(int n) const;

    BigInteger operator/(const BigInteger &number);
    // Modulo-Berechnung (=> das Ergebnis liegt im Intervall [0,number)!)
    BigInteger operator%(const BigInteger &number);
};

template <class T>
T abs(const T &number) {
    return number < 0 ? -number : number;
}

template <class T>
T ggt(T a, T b) {
    T tmp;

    while (b != 0) {
        tmp = a % b;
        a = b;
        b = tmp;
    }

    return a;
}

#endif
```

bigint.cc

```
#include <iostream>
#include <string.h>
#include <stdlib.h>
```

```
#include "bigint.h"

#define N 128                   // Groesse eines Ziffernarrays ist ein Vielfaches von N
#define INT_LEN 12              // max. Ziffernanzahl fuer den Typ int
#define MIN_KARATSUBA 110       // Groesse der Faktoren, ab denen Karatsuba verwendet wird

// konstruiert Ziffernarray NEU mit Groesse >= n
void BigInteger::constructSize(int n) {
    digits = (char*) 0; // damit kein Speicherplatz freigegeben wird ;-)
    maxlen = len = 0;   // damit garaniert ein neues Array angelegt wird
    ensureSize(n);
}

// Ziffernarray evtl. neu mit Groesse >= n anlegen
void BigInteger::ensureSize(int n) {
    if (maxlen < n) {
        // Groesse als Vielfaches von N (und >= n) bestimmen
        if (n % N != 0)
            n += N - (n % N);
        // neues Array anlegen
        digits = (char*) realloc(digits, n);
        // ... und den Rest mit 0en fuellen
        memset(digits+len, 0, n-len);
        // Laenge des neuen Arrays speichern
        maxlen = n;
    }
}

// Zahl normalisieren: +0 statt -0 und fuehrende Nullen entfernen
void BigInteger::normalize() {
    // (unnoetige) fuehrende Nullen entfernen
    while (digits[len-1] == 0 && len > 1)
        len--;
    // aus -0 machen wir +0
    if (len == 1 && digits[0] == 0)
        negative = false;
    // und alle nicht verwendeten Ziffern setzen wir mal auf 0
    memset(digits+len, 0, maxlen-len);
}

BigInteger::BigInteger(int number = 0) {
    constructSize(INT_LEN);
    *this = number;
}

// BigInteger-Zahl einfach kopieren
BigInteger::BigInteger(const BigInteger &number) {
    constructSize(number.len);
    *this = number;
```

```
}

// BigInteger aus String erzeugen
// evtl. Exception, falls ein ungueltiges Zeichen vorkommt
BigInteger::BigInteger(char *number) {
    constructSize(strlen(number));
    *this = number;
}

BigInteger::~BigInteger() {
    free(digits);
}

int BigInteger::toInt() const {
    int value = 0;

    for (int i = len-1; i >= 0; i--)
        value = value * 10 + digits[i];

    return value;
}

BigInteger &BigInteger::operator=(const BigInteger &number) {
    ensureSize(number.maxlen);

    memcpy(digits, number.digits, number.len);
    len = number.len;
    negative = number.negative;

    normalize();

    return *this;
}

BigInteger &BigInteger::operator=(int number) {
    ensureSize(INT_LEN);

    negative = number < 0;
    number = ::abs(number);

    len = 0;
    do {
        digits[len++] = number % 10;
        number /= 10;
    } while (number > 0);

    normalize();

    return *this;
```

```
}

BigInteger &BigInteger::operator=(char *number) {
    ensureSize(strlen(number));

    if (*number == '-') {
        number++;
        negative = true;
    }
    else
        negative = false;

    len = strlen(number);
    for (int i = len-1; i >= 0; i--, number++) {
        if (*number < '0' || *number > '9')
            throw "ungueltiges Zeichen im String: Ziffer erwartet";
        digits[i] = *number - '0';
    }

    normalize();

    return *this;
}

inline bool BigInteger::operator==(const BigInteger &number) const {
    return negative == number.negative &&
           len == number.len &&
           memcmp(digits, number.digits, len) == 0;
/*
    // durch die Normalisierung bedeutet versch. Laenge auch Ungleichheit!
    if (negative != number.negative || len != number.len)
        return false;

    // ziffernweise vergleichen
    for (int i = len-1; i >= 0; i--)
        if (digits[i] != number.digits[i])
            return false;

    return true;
*/
}

bool BigInteger::operator!=(const BigInteger &number) const {
    return !(*this == number);
}

bool BigInteger::operator<(const BigInteger &number) const {
    // Vorzeichen ungleich?
    if (negative != number.negative)
```

```
        return negative;
    // Laenge ungleich?
    else if (len < number.len)
        return !negative;
    else if (len > number.len)
        return negative;

    // bei gleicher Laenge erste ungleiche Ziffer suchen
    for (int i = len-1; i >= 0; i--)
        if (digits[i] < number.digits[i])
            return !negative;
        else if (digits[i] > number.digits[i])
            return negative;

    return negative;
}

bool BigInteger::operator<=(const BigInteger &number) const {
    return (*this < number || *this == number);
}

bool BigInteger::operator>(const BigInteger &number) const {
    return !(*this <= number);
}

bool BigInteger::operator>=(const BigInteger &number) const {
    return !(*this < number);
}

BigInteger BigInteger::operator-() const {
    BigInteger res(*this);
    res.negative = !res.negative;
    res.normalize();
    return res;
}

BigInteger BigInteger::operator+(const BigInteger &number) const {
    // handelt es sich tatsaechlich um eine Subtraktion?
    if (negative != number.negative)
        return *this - (-number);

    BigInteger res(*this);

    int n = (len > number.len ? len : number.len) + 1;
    res.ensureSize(n);

    // Addition nach der Schulmethode
    int tmp = 0;
    for (int i = 0; i < number.len; i++) {
```

```
        res.digits[i] += number.digits[i] + tmp;
        tmp = res.digits[i] / 10;
        res.digits[i] %= 10;
    }
    for (int i = number.len; i < n; i++) {
        res.digits[i] += tmp;
        tmp = res.digits[i] / 10;
        res.digits[i] %= 10;
    }

    res.len = n;
    res.negative = negative;

    res.normalize();

    return res;
}

BigInteger BigInteger::operator-(const BigInteger &number) const {
    // handelt es sich tatsaechlich um eine Addition?
    if (negative != number.negative)
        return *this + (-number);

    const BigInteger *a, *b;
    bool swap = false;

    if (::abs(*this) <= ::abs(number))
        a = &number, b = this, swap = true;
    else
        a = this, b = &number;

    BigInteger res(*a);

    // Subtraktion nach der Schulmethode
    int tmp = 0;
    for (int i = 0; i < b->len; i++) {
        res.digits[i] -= b->digits[i] + tmp;
        if (res.digits[i] < 0) {
            res.digits[i] += 10;
            tmp = 1;
        }
        else
            tmp = 0;
    }
    for (int i = b->len; i < a->len; i++) {
        res.digits[i] -= tmp;
        if (res.digits[i] < 0) {
            res.digits[i] += 10;
            tmp = 1;
```

```
        }
        else
            tmp = 0;
    }

    if (swap)
        res.negative = !negative;

    res.normalize();

    return res;
}

BigInteger BigInteger::operator*(int number) const {
    BigInteger res(0);

    res.negative = (negative != (number < 0));
    number = ::abs(number);

    int n = len + INT_LEN;
    res.ensureSize(n);

    // Multiplikatiom nach der Schulmethode
    long tmp = 0;
    int i;
    for (i = 0; i < len; i++) {
        tmp = digits[i] * number + tmp;
        res.digits[i] = tmp % 10;
        tmp /= 10;
    }
    while (tmp > 0) {
        res.digits[i++] = tmp % 10;
        tmp /= 10;
    }

    res.len = i;
    res.normalize();

    return res;
}

// Multiplikation zweier langer Zahlen nach unterschiedlichen Methoden
// 1.) fuer ganz kleine Zahlen verwenden wir die Multiplikation mit einem Integer
// 2.) fuer Faktoren mit einer Laenge < MIN_KARATSUBA multiplizieren
//     wir nach der Schulmethode
// 3.) fuer Faktoren ab der Laenge MIN_KARATSUBA verwenden wir die Methode
//     nach Karatsuba
BigInteger BigInteger::operator*(const BigInteger &number) const {
    if (number.len <= 9)
```

```
    return *this * number.toInt();
else if (len <= 9)
    return number * toInt();
else if (len >= MIN_KARATSUBA && number.len >= MIN_KARATSUBA) {
    // ******* Methode nach Karatsuba ********
    // (siehe z.B. Algorithmen-Buch von Prof. Schoening)

    int n = (len > number.len ? len : number.len) / 2;

    // berechne x0 und x1 so, dass *this = x1 << n + x0
    BigInteger x1 = *this >> n;
    BigInteger x0 = *this - (x1 << n);
    // berechne y0 und y1 so, dass number = y1 << n + y0
    BigInteger y1 = number >> n;
    BigInteger y0 = number - (y1 << n);

    // Multiplikationen nur einmal berechnen und Ergebnis speichern
    BigInteger x1y1 = x1 * y1;
    BigInteger x0y0 = x0 * y0;

    // insgesamt also 3 Multiplikationen der Groesse n statt
    // eine Multiplikation der Groesse 2n notwendig
    return (x1y1                                  << (2*n)) +
           ((x1y1 + x0y0 - (x0 - x1) * (y0 - y1)) <<   n  ) +
           x0y0;
}
else {
    // ******* Multiplikatiom nach der Schulmethode *******

    BigInteger res(0);

    res.negative = (negative != number.negative);

    int n = len + number.len;
    res.ensureSize(n);

    int tmp, j;
    for (int i = 0; i < len; i++) {
        tmp = 0;
        for (j = 0; j < number.len; j++) {
            res.digits[i+j] += digits[i] * number.digits[j] + tmp;
            tmp = res.digits[i+j] / 10;
            res.digits[i+j] %= 10;
        }
        while (tmp != 0) {
            res.digits[i+j] += tmp;
            tmp = res.digits[i+j] / 10;
            res.digits[i+j] %= 10;
            j++;
```

```
            }
        }

        res.len = n;
        res.normalize();

        return res;
    }
}

// Dezimal-Zahl um n Stellen nach rechts schieben
// d.h. Division durch 10^n
BigInteger BigInteger::operator>>(int n) const {
    BigInteger res(*this);

    if (n >= len)
        return 0;
    else if (n > 0) {
        memmove(res.digits, res.digits+n, len-n);
        res.len -= n;
        res.normalize();
    }
    else if (n < 0)
        return *this << (-n);

    return res;
}

// Dezimal-Zahl um n Stellen nach links schieben
// d.h. Multiplikation mit 10^n
BigInteger BigInteger::operator<<(int n) const {
    BigInteger res(*this);

    if (n > 0) {
        res.ensureSize(len+n);
        memmove(res.digits+n, res.digits, len);
        memset(res.digits, 0, n);
        res.len += n;
        res.normalize();
    }
    else if (n < 0)
        return *this >> (-n);

    return res;
}

BigInteger BigInteger::operator/(const BigInteger &number) {
    BigInteger quotient(0), dividend(::abs(*this)), divisor(::abs(number));
```

```
    // um wieviel Stellen muessen wir den Divisor verschieben?
    int n = dividend.len - divisor.len;
    if (n < 0)
        n = 0;
    divisor = divisor << n;

    // Division nach der Schulmethode
    int i, j;
    for (i = n; i >= 0; i--) {
        for (j = 0; dividend >= divisor * (j+1); j++);
        dividend = dividend - divisor * j;
        quotient = (quotient << 1) + j;
        divisor = divisor >> 1;
    }

    quotient.negative = negative != number.negative;

    return quotient;
}

// Modulo-Berechnung (=> das Ergebnis liegt im Intervall [0,number)!)
BigInteger BigInteger::operator%(const BigInteger &number) {
    BigInteger quotient(0), dividend(::abs(*this)), divisor(::abs(number));

    // um wieviel Stellen muessen wir den Divisor verschieben?
    int n = dividend.len - divisor.len;
    if (n < 0)
        n = 0;
    divisor = divisor << n;

    // Division nach der Schulmethode
    int i, j;
    for (i = n; i >= 0; i--) {
        for (j = 0; dividend >= divisor * (j+1); j++);
        dividend = dividend - divisor * j;
        quotient = (quotient << 1) + j;
        divisor = divisor >> 1;
    }

    quotient.negative = negative != number.negative;

    return dividend;
}

// Ausgabe der langen Zahl
ostream &operator<<(ostream &out, BigInteger number) {
    out << (number.negative ? "-" : "");
    for (int i = number.len-1; i >= 0; i--)
        out << (int)number.digits[i];
```

```
    return out;
}
```

recursion.h

```
#ifndef _RECURSION_H

// Fibonacci-Zahlen
// Kommentar: einigermassen rekursiv
template <class T>
T fib(T n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}

// Ackermann-Funktion
// Kommentar: MASSIV rekursiv!!!
template <class T>
T ackermann(T x, T y) {
    if (x == 0)
        return y + 1;
    else if (y == 0)
        return ackermann(x-1, (T) 1);
    else
        return ackermann(x-1, ackermann(x, y-1));
}

#endif
```