Aufgabe 2: Geburtstag

Team-ID: 00619

Team-Name: ncrypted

Bearbeiter/-innen dieser Aufgabe:
Oliver Schirmer

10. April 2020

Inhaltsverzeichnis

Lösungsidee

Die grundlegende Idee basiert auf dynamischer Programmierung, wobei ein Bottom-Up-Ansatz verfolgt wird. Da von Beginn an nicht klar ermittelbar ist, welche Terme und Termverbindungen das optimale Ergebnis, also den Term mit den wenigsten Ziffern, ergeben, wird, mit der Anzahl eins beginnend, steigend geprüft, welche Zahlen mit der aktuellen Anzahl an zu nutzenden Ziffern berechnet werden können. Dabei wird bei jeder Iteration (der aktuellen Anzahl an zu nutzenden Ziffern) geprüft, ob die Verbindung einer vorherigen bereits gebildeten Zahl und der entsprechenden größtmöglichen mehrstelligen Zahl, bestehend aus der vorgegebenen Ziffer, eine neue Zahl ergibt. Ist dies der Fall, wird sie gespeichert. Zudem muss bei jeder Iteration geprüft werden, ob die Verbindung zweier bereits gebildeter beliebiger Zahlen, eine neue Zahl ergibt. Entspricht eine neu gebildete Zahl dem Endergebnis, terminiert der Algorithmus, da dies mit Sicherheit das optimale Ergebnis ist.

Bei beiden eben genannten „Verbindungs-Schritten“ werden je zwei Zahlen miteinander verbunden, womit die Grundoperationen (Addition, Subtraktion, Multiplikation, Division), als auch Potenzrechnung abgedeckt werden können, da all diese Operationen zwei Operanden benötigen. Dazu muss bei jeder Iteration geprüft werden, ob die Berechnung der Fakultät der Zahlen, die mit der aktuellen Anzahl an zu nutzenden Ziffern auskommen, möglich ist und diese Ergebnisse ebenfalls gespeichert werden. Dabei ist vor allem bei der Potenzrechnung und Berechnung von Fakultäten wichtig, dass eine Obergrenze, d.h. ein maximales Ergebnis, festgelegt wird. Diese hilft nicht nur die Laufzeit bei den letzten beiden Operationen zu optimieren, da bei der Potenzierung (mit Hilfe von Potenzierung durch Quadrierung) maximal eine Laufzeit von $O=\log b$, wobei b der Exponent ist, und bei der Berechnung von Fakultäten maximal eine lineare Laufzeit erreicht werden kann (unter Voraussetzung, dass Multiplikation konstant viel Laufzeit benötigt), sondern auch bei den Grundoperationen, wie Multiplikation. Betrachtet man die Komplexität der Multiplikation nicht wie üblich als konstant, sondern in abhängig von der Länge der beiden Zahlen erhält man eine Laufzeit von $O\left(n^2\right)$, was bei länger werdenden Zahlen (Zahlen mit mehr Stellen) stark ins Gewicht fällt. Dies lässt sich zwar theoretisch durch schnellere Multiplikations-Algorithmen wie Harvey-Hoeven auf $O\left(n*\log n\right)$ reduzieren, was in der Praxis allerdings keine Rolle spielt, da solche Algorithmen erst einen Effekt bei Zahlen ab $10^{85\text{ Billionen}}$ Stellen haben.

Umsetzung

Bei allen Operationen wurden negative (da, solange keine negative Zahl gebildet werden soll, immer durch Umkehrung von Plus und Minus, ein ebenfalls optimales Ergebnis erzielt werden kann) und nicht-ganzzahlige (da die exakte Berechnung durch Double und Float nicht sichergestellt werden kann) Ergebnisse gefiltert, als auch zu große Ergebnisse, damit der Rechenaufwand nicht zu groß wird. Welche Obergrenze man dabei festlegt ist sehr wichtig, da bei einer zu kleinen Obergrenze entweder kein Term mehr gefunden wird oder ein suboptimaler mit mehr Ziffern, als minimal nötig. Die in der Aufgabenstellung zu berechnenden Zahlen waren vierstellig, wobei das maximale Zwischenergebnis bei 129208 lag. Deshalb wurde die Obergrenze bei 10^7 angesetzt, womit die Berechnung beliebiger maximal vierstelliger Zahlen reibungslos ablaufen sollte. Diese Obergrenze ist, wie bei der Lösungsidee bereits beschrieben, vor allem für Potenzrechnung, als auch Berechnung von Fakultäten wichtig, da die Zahlen und damit auch die Rechenzeit ohne eine solche Grenze explodieren würde. Da beide Operationen auf der Multiplikation basieren, kann die Überprüfung, ob das Ergebnis der Operation zu groß ist, bei jedem Zwischenschritt durchgeführt werden. Da sich bei der Multiplikation die Länge des Ergebnisses (Anzahl an Ziffern) aus der Summe der Länge der beiden Faktoren bildet, kann schnell überprüft werden, ob das aktuelle Zwischenergebnis größer als die Obergrenze ist. Ist dies der Fall wird kein Ergebnis zurückgegeben und der aktuelle Term verworfen. Da keine negativen, als auch keine gebrochenen Zahlen berechnet werden, können bei der Subtraktion, als auch bei der Division keine diese Grenze überschreitenden Zahlen entstehen, weshalb die Überprüfung dort wegfällt. Lediglich bei der Addition muss die Überprüfung ebenfalls durchgeführt werden, was mit Hilfe eines simplen „ist-größer“-Checks realisiert wird.

Betrachtet man die Anzahl an aktuell zur Termbildung benötigten Ziffern als n, wird n anfangs auf 1 gesetzt und in jedem Durchlauf inkrementiert. Dabei wird beim ersten Schritt, der Verbindung aller bereits berechneten Zahlen mit der entsprechenden größtmöglichen mehrstelligen, nur aus den zu nutzenden Ziffern bestehende Zahl immer nur ein Ergebnis, das mit n Ziffern auskommt erreicht. Damit die mehrstelligen Zahlen nicht bei jeder Berechnung neu erstellt werden müssen, werden diese zusätzlich in einem Array gecachet (ebenso wie die Fakultäten). Im zweiten Schritt, der Verbindung aller bisherigen berechneten Zahlen wäre eine naive Implementierung alle im ersten Schritt gebildeten Zahlen mit allen vorherigen Zahlen zu verbinden, womit allerdings Zahlen berechnet werden können, deren n größer ist, als das des Endergebnisses – sie also unnötig berechnet werden. Deshalb werden lediglich für jedes n zu Beginn der Iteration alle möglichen Verbindungen gebildet. D.h. alle Verbindungen von Zahlen mit Anzahl benötigter Ziffern x und $n-x$,

wobei $x \leq \frac{n}{2}$ sein muss.

Bei der Überprüfung der Verbindung zweier Zahlen oder Berechnung von Fakultäten wird zudem geprüft, ob das dafür nötige n kleiner als das vorherige ist. Dafür wird in einer HashMap die aktuelle minimale Anzahl n für jede bereits gebildete Zahl gespeichert. Zusätzlich müssen für jede Anzahl n die damit möglich zu bildenden Zahlen gespeichert werden. Dies wird in einer zweidimensionalen ArrayList getan, d.h. der Index der äußeren ArrayList entspricht n und die Zahlen innerhalb der inneren ArrayList entsprechen der möglich zu bildenden Zahlen. Als letztes muss für jede gebildete Zahl gespeichert werden, wie diese gebildet wurde. D.h. es wird wieder mittels einer HashMap jeder gebildeten Zahl ein Tupel aus drei Zahlen zugeordnet: Der vorherigen Zahl, die darauf angewendete Operation (kodiert als Integer) und die zweite Zahl der Operation. Bei der Fakultät wurde diese zweite Zahl einfach auf unendlich (eine Zahl außerhalb der Obergrenze) gesetzt, da dieses Ergebnis nie erreicht werden kann, und in der Rekonstruktion des endgültigen Terms entsprechend gefiltert. Bei der Rekonstruktion wird entsprechend auf beide an der Operation beteiligten Zahlen rekursiv die Rekonstruktionsmethode aufgerufen bis die zu rekonstruierende Zahl nur noch aus den erlaubten Ziffern besteht, wodurch auch der endgültige Term nur aus diesen Ziffern besteht.

Zusätzlich muss erwähnt werden, dass bei jeder Iteration überprüft wird, ob überhaupt eine neue Zahl gebildet werden konnte. Ist dies nicht der Fall und es wurde noch kein Term für die gesuchte Zahl gefunden, ist klar, dass

die Obergrenze zu niedrig angesetzt ist, um die Zahl zu bilden, da jeder positive ganze Zahl mit genügend Ziffern gebildet werden kann. Diesen Fehler gibt der Algorithmus dann aus und terminiert.

Beispiele

Die folgenden Beispiele wurden der Aufgabenstellung entnommen und teilweise erweitert.

Zahl	Ziffer	Term nur mit Grundoperationen ~ n	Term mit Potenzen und Fakultät ~ n
2019	1	(((1+1)*((11111-1)/11))-1) ~ 11	(((1+1)*((11111-1)/11))-1) ~ 11
	2	((2*((22*(2+(2*22)))-2))-(2/2)) ~ 10	((((((2/2)+(2*22))^2)-2)-2)-2) ~ 9
	3	(3+((3+3)*(3+333))) ~ 7	(3+(((3!)!)+((3!)*((3!)^3)))) ~ 5
	4	((((44*(4+(4+(4*44))))-4)/4)-4) ~ 10	(4+((((4+4)!)/((4!)-4))-(4/4))) ~ 7
	5	(((5*(5+(5*((5*5)+55))))-(5/5))-5) ~ 10	((((((5!)-5)*((5!)-5))-(5^5))-5)/5) ~ 8
	6	((6*(6+(6*(6+666))))/(6+6)) ~ 9	((6!)+((6+(6+(6+((6^6)/6))))/6)) ~ 8
	7	(((77-7)/7)+(7*((7*((7*7)-7))-7))) ~ 10	((7777-((((7!)-7)-7)/7))-(7!)) ~ 9
	8	((((8*(8+(8*((8*(8*8))-8))))-8)-8)/(8+8)) ~ 11	(((8+((8!)+88888))/8)/8) ~ 9
	9	((9+(9+(9+((9+9)*(9+999)))))/9) ~ 10	(99+((9!)/(9+((9*9)+99)))) ~ 8
2020	1	((1+1)*((11111-1)/11)) ~ 10	((1+1)*((11111-1)/11)) ~ 10
	2	(2*((22*(2+(2*22)))-2)) ~ 8	(2*((2*(22+(22^2)))-2)) ~ 8
	3	(3+((3/3)+((3+3)*(3+333)))) ~ 9	((3*((3!)!))-((((3!)!)+(((3!)!)/(3!)))/(3!))) ~ 6
	4	(4+((4+4)*((4*(4*(4*4)))-4))) ~ 8	(4+(((4+4)!)/((4!)-4))) ~ 5
	5	((5*(5+(5*((5*5)+55))))-5) ~ 8	(((((5!)-5)*((5!)-5))-(5^5))/5) ~ 7
	6	((6-(6/6))*(6+(((6+6)/6)+(6*66)))) ~ 10	((6!)+(((((6!)*66)-(6!))/6)/6)) ~ 7
	7	((77/7)+(7*((7*((7*7)-7))-7))) ~ 9	(7+(((7+(7+7))*(((7!)/7)-(7*7)))/7)) ~ 9
	8	((8*(8+(8*((8*(8*8))-8))))/(8+8)) ~ 9	(((((8!)-(88*88))/(8+8))-8)-8) ~ 9
	9	(((9+9)*((99/9)+999))/9) ~ 9	(((9+9)*((99/9)+999))/9) ~ 9
2030	1	((11-1)*(1+((1+1)*(1+(111-11))))) ~ 12	(((((1+1)^11)-((1+(1+1))!))-11)-1) ~ 10
	2	(2+(2*(2+(22*(2+(2*22)))))) ~ 9	((2*(2*(((2+2)!)+(22^2))))-2) ~ 8
	3	((33+((3+3)*333))-(3/3)) ~ 9	((3*(((3!)!)-3))-(((3!)+((3!)!))/(3!))) ~ 6
	4	((4*((4*(4*(4*(4+4))))-4))-((4+4)/4)) ~ 10	((((4!)/4)+((4+4)*(4^4)))-(4!)) ~ 7
	5	(5+(5*(5+(5*((5*5)+55))))) ~ 8	((((5!)*(5+((5!)/(5+5))))-5)-5) ~ 7
	6	((6-(6/6))*((6*66)+((66-6)/6))) ~ 10	(((6!)+((((6!)+(6^6))/6)/6))-6) ~ 7
	7	((7*(7+(7*((7*7)-7))))-77) ~ 8	(7*(7+(((7!)/(7+7))-77))) ~ 7
	8	(((((8+8)*((8*(8*(8+8)))-8))-8)-8)/8) ~ 10	(((((8!)/8)/8)+((8+8)*88))-8) ~ 8
	9	((9+(9+(99/9)))*((9*9)-(99/9))) ~ 10	(((((9+9)/9)^(99/9))-9)-9) ~ 8
2080	1	(1+(11*(((1+1)*(111-11))-11))) ~ 12	((((1+1)^11)+(11*(1+(1+1))))-1) ~ 10
	2	(2*(2*((22-2)*(2+(2+22))))) ~ 9	(2*(2*((22-2)*(2+((2+2)!))))) ~ 8
	3	((3/3)+(3*(33*(3+(3*(3+3)))))) ~ 9	((3*((3!)!))-((((3!)!)/3)/3)) ~ 5
	4	((4+4)*(4+(4*(4*(4*4))))) ~ 7	((4+4)*(4+(4^4))) ~ 5
	5	((5+(5*(5*5)))*(5+(55/5))) ~ 8	(5+(5*((5!)+((5!)+((5!)+55))))) ~ 7
	6	((66-(6/6))*(((6*6)+((6+6)/6))-6)) ~ 10	((((((6!)*((6!)+((6!)/6)))/6)/6)/6)-(6!)) ~ 8
	7	((7+(77*(((7+7)*(7+7))-7)))/7) ~ 9	((7+(77*(((7+7)*(7+7))-7)))/7) ~ 9

	8	(8*((8*(8+(8*(8*8))))/(8+8))) ~ 8	(8*(8+((8!)/(8+((8*8)+88))))) ~ 8
	9	((9+(99*(9+((9*9)+99))))/9) ~ 9	((9*(9*(9*9)))-((9+((9!)/9))/9)) ~ 8
2980	1	(1+(((11-1)-1)*((((1+(1+1))*111)-1)-1))) ~ 13	(((((1+((1+(1+1))!))!)-((1+1)^11))-11)-1) ~ 11
	2	(2*((2*((2*(22*22))-222))-2)) ~ 11	(2*(2+(2*(((2+2)!)+((2+(2+2))!))))) ~ 8
	3	((3/3)+(3*(3+(3*(333-3))))) ~ 9	(((3!)!)+((3/3)+(3*(((3!)!)+33)))) ~ 7
	4	((((44*(4+(4*(4*4))))-4)-4)-4) ~ 9	(4+(4*((4!)+(((4!)/4)!)))) ~ 5
	5	(5+(5*((5*((5*(5*5))-5))-5))) ~ 8	(5+(5*((5*(5!))-5))) ~ 5
	6	(((((6+(6*6))*((6*(6*(6+6)))-6))-6)-6)/6) ~ 11	(((6!)+(((6+6)*((6!)+(6!)))-((6!)/6)))/6) ~ 8
	7	((((((7*7)-7)*(7+(7*(77-7))))-7)-7)/7) ~ 11	(((7!)+((7!)+((7+7)*(777-7))))/7) ~ 9
	8	((8/8)+((8+(8*(8*(8*(8*(8*8))))))/88)) ~ 11	(8+(8+(((8!)+(8*888))/(8+8)))) ~ 9
	9	((((9+((9+(9+9))*999))/9)-9)-9) ~ 10	((9*(9*9))+((99+((9!)/(9+9)))/9)) ~ 9

Im Vergleich der Spalten 3 und 4 fällt deutlich der Unterschied zwischen den jeweiligen n auf. Durch die Nutzung von Potenzen und Fakultät wird die Anzahl an Ziffern deutlich reduziert, wie auch in Teilaufgabe b beschrieben. Dabei scheint die n im Durchschnitt unabhängig von der zu nutzenden Ziffer zu sein. Lediglich bei der Ziffer 1 lässt sich ein allgemein geringfügig höheres n erkennen.

Zahl	Ziffer	Term mit Potenzen und Fakultät ~ n
12345	1	(((1+(1+(1+1)))!)+(111^(1+1))) ~ 9
	2	(((2+2)!)+((222/2)^2)) ~ 7
	3	(((((3!)+((3!)*(3!)))^3)/(3!))-3) ~ 6
	4	((4!)+((4/4)+(44*((4!)+(4^4))))) ~ 8
	5	(((5+(5+(5!)))*((5!)-(5*5)))-5) ~ 7
	6	((6+(6+((6+((6!)*(6!)))/(6+(6/6))))))/6) ~ 9
	7	(((7+(7+(7/7)))*((7!)+((7+(7!))/7)))/7) ~ 9
	8	(8+(8+(8+((888/8)^((8+8)/8))))) ~ 10
	9	(9+(((((9!)/9)-9)/9)+(9*(((9*99)-9)-9)))) ~ 11
123456	1	(((1+(1+(1+1)))!)+(111*(1+1111))) ~ 12
	2	(((2+2)!)*(((2+2)!)+((22-2)*(2^(2*(2+2)))))) ~ 11
	3	(((((((3!)!)/(3!))-(3!))^3)/((3!)+(3!)))-(3!)) ~ 7
	4	((4!)*((4!)+((4^4)*((4!)-4)))) ~ 6
	5	(((5!)*(5+((5-(5/5))^5)))-((5!)/5)) ~ 8
	6	((((((6!)/6)-6)^((6+(6+6))/6))/(6+6))-6) ~ 10
	7	((7+(7*(7*((7*(7!))-7))))/(7+7)) ~ 8
	8	(((((8!)+((8!)+((8!)+((8!)/(8+8)))))-8)-8)-8) ~ 9
	9	((((9+(9+9999999))/9)-9)/9) ~ 12

In der obigen Tabelle wurden nochmals eigene Beispiele überprüft, die in diesem Falle allerdings fünf- und sechsstellig waren. Dafür wurde die Obergrenze auf 10^{12} erhöht. Hierbei lässt sich geringfügig erkennen, dass n in gewisser Weise abhängig von der Länge der zu bildenden Zahl ist und mit steigender Anzahl an Stellen im Durchschnitt auch n und somit die Laufzeit steigt.

Quellcode

```
#include <bits/stdc++.h>

using namespace std;
using namespace chrono;
typedef long long ll;
#define MAX 10000000
#define INF MAX+1

ll targetNr = -1;
int figure = -1;
vector<ll> figurePows;
// predNr, predOp, predOpNr
map<ll, tuple<ll, int, ll>> pred;

map<ll, int> curBest;
vector<vector<ll>> prevNrs;

int curAmount;
int nrFound;

// --- CUSTOM OPERATIONS ---
ll multiply(ll nr1, ll nr2) {
    if (to_string(nr1).length() + to_string(nr2).length() >
to_string(MAX).length()) {
        return -1;
    }
    ll res = nr1 * nr2;
    if (res < -MAX || MAX < res) {
        return -1;
    }
    return res;
}

ll fastPow(ll base, ll exp) {
    ll res = 1;
    ll temp;
    while (exp > 0) {
        if (exp % 2 == 1) {
            temp = multiply(res, base);
            if (temp == -1) {
                return -1;
            }
            res = temp;
        }
        temp = multiply(base, base);
        if (temp == -1) {
            return -1;
        }
        base = temp;
        exp /= 2;
    }
    return res;
}

vector<ll> factorials = {1, 1};
ll maxFact = INT_MAX;

ll fastFact(ll nr) {
    if (nr > maxFact) {
```

```
            return -1;
        }
        while (factorials.size() <= nr) {
            ll curFact = multiply(factorials[factorials.size() - 1],
    factorials.size());
            if (curFact == -1) {
                maxFact = factorials.size() - 1;
                return -1;
            }
            factorials.push_back(curFact);
        }
        return factorials[nr];
    }


    // --- GENERATE NRS USABLE IN TERM ---
    bool isFigurePow(ll nr) {
        string s = to_string(nr);
        for (int i = 0; i < s.length(); i++) {
            if (s[i] != ((char) (figure + 48))) {
                return false;
            }
        }
        return true;
    }

    ll getFigurePow(int length) {
        while (figurePows.size() < length) {
            figurePows.push_back(figurePows[figurePows.size() - 1] *
    10 + figure);
        }
        return figurePows[length - 1];
    }

    // --- ALGORITHM ---
    bool checkOp(ll opRes, tuple<ll, int, ll> op, int figureAmount) {
        if (0 < opRes && opRes <= MAX && (curBest[opRes] == 0 ||
    curBest[opRes] > figureAmount)) {
            pred[opRes] = op;
            curBest[opRes] = figureAmount;
            prevNrs[figureAmount].push_back(opRes);
            if (opRes == targetNr) {
                nrFound = figureAmount;
            }
            return true;
        }
        return false;
    }

    bool check(ll prevNr, ll curNr, int figureAmount) {
        if (nrFound != 0 && figureAmount >= nrFound) {
            return false;
        }
        bool found = false;
        // additional operation: factorial
        if (prevNr == INF && curNr != 0) {
            ll fact = fastFact(curNr);
            if (fact != -1) {
                found = max(found, checkOp(fact, {curNr, 6, prevNr},
    figureAmount));
            }
```

```
            return found;
        }

        ll plus = prevNr + curNr;

        found = max(found, checkOp(plus, {prevNr, 1, curNr},
    figureAmount));

        ll minus;
        // only use positive results - negative terms can be replaced
    by positive ones and inverted operation sign
        if (prevNr > curNr) {
            minus = prevNr - curNr;
            found = max(found, checkOp(minus, {prevNr, 2, curNr},
    figureAmount));
        } else {
            minus = curNr - prevNr;
            found = max(found, checkOp(minus, {curNr, 2, prevNr},
    figureAmount));
        }

        // check beforehand, whether result would be out of bounds
        ll multi = multiply(prevNr, curNr);
        if (multi != -1) {
            found = max(found, checkOp(multi, {prevNr, 3, curNr},
    figureAmount));
        }

        ll div;
        if (curNr != 0 && prevNr % curNr == 0) {
            div = prevNr / curNr;
            found = max(found, checkOp(div, {prevNr, 4, curNr},
    figureAmount));
        }
        if (prevNr != 0 && curNr % prevNr == 0) {
            div = curNr / prevNr;
            found = max(found, checkOp(div, {curNr, 4, prevNr},
    figureAmount));
        }

        // additional operation: pow
        ll power = fastPow(prevNr, curNr);
        if (curNr != 0 && power != -1) {
            found = max(found, checkOp(power, {prevNr, 5, curNr},
    figureAmount));
        }
        power = fastPow(curNr, prevNr);
        if (prevNr != 0 && power != -1) {
            found = max(found, checkOp(power, {curNr, 5, prevNr},
    figureAmount));
        }

        return found;
    }

    string reconstruct(ll nr) {
        // only used for factorials (no second number for operation)
        if (nr == INF) {
            return "";
        }
        if (isFigurePow(nr)) {
```

```
            return to_string(nr);
        }
        vector<pair<ll, char>> ops;
        while (nr != 0) {
            ll predNr, opNr;
            int op;
            tie(predNr, op, opNr) = pred[nr];
            char opChar = op == 1 ? '+' : op == 2 ? '-' : op == 3 ?
    '*' : op == 4 ? '/' : op == 5 ? '^' : '!';
            ops.push_back({opNr, opChar});
            nr = predNr;
        }
        string res = reconstruct(ops[ops.size() - 1].first);
        for (int i = ops.size() - 2; i >= 0; i--) {
            res = "(" + res + ops[i].second +
    reconstruct(ops[i].first) + ")";
        }
        return res;
    }

    void run() {
        // clear prev variables - only needed when run multiple times
        pred.clear();
        curBest.clear();
        // variable initialization
        figurePows = {figure};
        curAmount = 0;
        nrFound = 0;
        prevNrs = {{0}};
        curBest[0] = 0;

        // algorithm
        while (!nrFound || nrFound > curAmount + 1) {
            curAmount++;
            bool found = false;
            prevNrs.push_back(vector<ll>());

            // try calc with previous nrs
            for (int i = 1; i <= curAmount / 2; i++) {
                int secondAmount = curAmount - i;
                for (ll prevNr : prevNrs[i]) {
                    for (ll curNr : prevNrs[secondAmount]) {
                        found = max(found, check(prevNr, curNr,
    curAmount));
                    }
                }
            }
            // try calc with prev nrs & possible "potency" 2->11, 3-
    >111...
            for (int i = 0; i < curAmount; i++) {
                int dif = curAmount - i;
                ll curFigPow = getFigurePow(dif);
                for (ll prevNr : prevNrs[i]) {
                    found = max(found, check(prevNr, curFigPow,
    curAmount));
                }
            }

            // try factorials
            queue<ll> factQueue;
            for (ll curNr : prevNrs[curAmount]) {
```

```
            factQueue.push(curNr);
        }
        while (!factQueue.empty()) {
            ll curNr = factQueue.front();
            factQueue.pop();
            if (check(INF, curNr, curAmount)) {
                factQueue.push(fastFact(curNr));
                found = true;
            }
        }

        if (!found) {
            if (!nrFound) {
                cout << "Boundaries too small - no term found" <<
endl;
                return;
            }
            break;
        }
    }

    cout << reconstruct(targetNr) << " ~ " << nrFound << endl;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    while (targetNr == -1) {
        cout << "Bitte geben sie die darzustellende Nummer ein:"
<< endl;
        ll temp;
        cin >> temp;
        if (temp >= 0) {
            targetNr = temp;
        }
    }
    for(int i = 1; i <= 9; i++) {
        figure = i;
        run();
    }
}
```