Kapitel 1

Lösungsidee

Mein Algorithmus zum Zählen der Übungen basiert auf einem interessanten Effekt, der entsteht wenn man zu einer Folge von Übungen eine weitere hinzufügt. Dann nimmt nämlich die Anzahl der Möglichkeiten Übungen durchzuführen um genau die Anzahl der bisherigen Möglichkeiten zu, abzüglich der Anzahl der Möglichkeiten, die sich ergeben, wenn man nur die Übungen vor dem letzten Auftreten der neuen Übung benutzt.
Ist es eine neue Übung, die bisher nicht vorkam, gibt es logischerweise kein letztes Auftreten dieser Übung, weshalb die Anzahl der Möglichkeiten um sich selbst zu nimmt, bzw. sie verdoppelt sich.
Das kann man gut an einem Beispiel sehen:

Folge	Möglichkeiten	Teilfolge	Änderung
leere Folge	1	-	
a	2	-	
a**b**	4	-	
ab**a**	7		+4 -1
aba**c**	14	-	
ab**a**c**a**	24	ab	+14 -4
a**b**aca**b**	46	a	+24 -2
abac**a**b**a**	78	abac	+46-14

Auf diese Weise kann man nun aus einer gegebenen Folge von Übungen, die Anzahl der Möglichkeiten berechnen, indem man einfach immer eine Übung hinzufügt und wie im Beispiel die Anzahl der Möglichkeiten mit dynamischer Programmierung neuberechnet.

Das Komplizierte ist nun zu beweisen, dass dieses Verfahren auch wirklich funktioniert.
Als ersten Test kann man das Verfahren für verschiedene Folgen durchführen, und die damit berechnete Anzahl von Möglichkeiten mit einer Anzahl vergleichen, die man auf andere Art (z.B.: durch Aufzählen aller Folgen) ermittelt hat.
Damit kann man zwar nicht sicherstellen, dass das Verfahren fehlerfrei ist, da man ja möglicherweise gerade die Folge nicht getestet hat, bei der das Verfahren versagen würde.
Deshalb nimmt man besser einen anderen Weg um die Korrektheit des Algorithmus zu zeigen, und zwar in der Art eines mathematischen Beweises. (was allerdings wesentlich länger dauert, als einfaches ausprobieren)
Dabei muss man beweisen, dass der gleich am Anfang beschriebene Effekt beim Hinzufügen einer Übung wirklich auftritt und man auf diese Weise auch tatsächlich die Anzahl der neuen Möglichkeiten erhält.
Um diesen Beweis übersichtlicher zu gestalten, verwende ich folgende Notationen:
Um eine Folge von Übungen anzugeben, schreibe ich die Übungen direkt hintereinander. Eine Folge, die direkt von einer Übung gefolgt wird, bedeutet die Folge die entsteht, wenn man an die alte Folge, die neue Übung anhängt. Ist v eine Übung, so ist v^i mit $i \in \mathbb{N}$ die Folge die ensteht, wenn man i-mal hintereinander die Übung v schreibt. (Also $v^0 =$, $v^1 = v$, $v^2 = vv$, ...). Ist F eine Folge, so meint F^* die Menge aller Teilfolgen von F und F^{*v} die Menge aller Teilfolgen von F, die mit v enden. Ist M eine Menge, so ist $|M|$ wie normal die Anzahl ihrer Elemente.
Sei $F = u_1u_2u_3 \dots u_{n-1}u_n$ eine Folge von Übungen und v eine neue Übung.
Offenbar gilt $|Fv^*| = |F^*|+|Fv^{*v}|-|F^{*v}|$, also die Anzahl aller Teilfolgen von Fv ist genau so groß wie die Anzahl der Teilfolgen von F plus die Anzahl der Teilfolgen von Fv, die keine Teilfolgen von F sind. Da diese Teilfolgen mit v enden müssen, ist ihre Anzahl natürlich so groß, wie die Zahl der Teilfolgen von Fv, die mit v enden, abzüglich der Teilfolgen von F, die ebenfalls mit v enden.
Um den oben erwähnten Effekt zu beweisen, ist also nur die Zahl $|Fv^{*v}|-|F^{*v}|$ zu bestimmen.
Zuerst zeige ich, dass $|Fv^{*v}| = |F^*|$, also dass es eine bijektive Funktion zwischen F^* und Fv^{*v} gibt, nämlich $f : Fv^{*v} \to F^*$ mit $Tv \mapsto T$.
Diese Funktion existiert wirklich, da jede Teilfolge in Fv^{*v} mit v enden muss,

und die Teilfolge vor dem letzten v eine Teilfolge in F^* sein muss.
Zu zeigen ist daher nur noch, dass f bijektiv ist:

- f ist injektiv:
  Sind $S, S' \in Fv^{*v}$ mit $S \neq S'$, so kann man sie als $S = Tv$ und $S' = T'v$ mit $T, T' \in F^*$ schreiben, wobei $T \neq T'$ gilt.
  Dann folgt $f(S) = f(Tv) = T \neq T' = f(T'v) = f(S')$.

- f ist surjektiv:
  Für jedes $T \in F^*$ gilt nach Definition von f, dass $f(Tv) = T$.

Der Fall, dass die neue Übung v noch nicht der Folge F vorkommt ist damit bereits bewiesen, da dann natürlich $|F^{*v}| = 0$ gilt und daraus folgt, dass $|Fv^*| = |F^*| + |Fv^{*v}| - |F^{*v}| = 2 * |F^*|$.
Die Anzahl der möglichen unterschiedlichen Teilfolgen verdoppelt sich also.
Daher ist nur noch der Fall zu betrachten, dass die Übung v bereits vorkommt. Das heißt es gibt ein $i \in \mathbb{N}$, so dass $u_i = v$.
Sei $F' = u_1 u_2 \dots u_{i-2} u_{i-1}$, wobei $i \in \mathbb{N}$ das größte i mit $u_i = v$ sei. (Es gilt also $F = F'v u_{i+1} u_{i+2} \dots u_{n-1} u_n$, wobei $u_j \neq v$ für alle $j > i$).
Zu zeigen ist nun, dass $|F^{*v}| = |F'^*|$ gilt.
Weiter oben habe ich bereits bewiesen, dass $|Fv^{*v}| = |F^*|$ für alle Folgen F.
Daraus folgt, dass gilt $|F'v^{*v}| = |F'^*|$. Deshalb muss jetzt nur noch gezeigt werden, dass $|F'v^{*v}| = |F^{*v}|$ bzw. dass sogar $F'v^{*v} = F^{*v}$ gilt.

- $F'v^{*v} \subseteq F^{*v}$:
  Da $F'v$ eine Teilfolge von F ist, ist auch jede Teilfolge von $F'v$ eine Teilfolge von F.

- $F'v^{*v} \supseteq F^{*v}$:
  Sei $T \in F^{*v}$. Dann endet T mit v, das heißt, dass in T keine Übung u_j mit $j > i$ vorkommt. Das wiederum bedeutet, dass T aus den Übungen u_1 bis u_i besteht, und damit eine Teilfolge von $F'v$ ist.

Es ist nun gezeigt, das gilt $|F^{*v}| = |F'^*|$, daraus folgt $|Fv^*| = 2|F^*| - |F'^*|$.
Zusammenfassend kann man nun sagen, dass $|Fv^*| = 2|F^*|$ ist, falls v nicht in F vorkommt, und dass $|Fv^*| = 2|F^*| - |F'^*|$, falls v schon vorkommt und man mit F' die Teilfolge von F bezeichnet, die entsteht, wenn man von F nur den Teil bis zum letzten Vorkommen von v (exklusive) betrachtet.
Das ist nun genau dann der am Anfang beschriebene Effekt, dass die Anzahl der Teilfolgen von Fv genau um die Anzahl der Teilfolgen von F abzüglich der Teilfolgen der Teilfolge von F vor dem letzten Auftreten von v zu nimmt.

Kapitel 2

Programmdokumentation

Das Programm implementiert den im letzten Kapitel beschriebenen Algorithmus über dynamische Programmierung.
Eine Folge von Übungen wird dabei als ASCII-String gespeichert, in dem jedes Zeichen für eine Übung steht. und der an die Funktion **countParts** übergeben wird. Dort wird berechnet, wie die Anzahl der Möglichkeiten ist, wenn man nur das erste Zeichen benutzt, dann wie viele es gibt, wenn man nur die ersten beiden benutzt, dann wie viele es bei drei Zeichen gibt, usw., bis berechnet wurde, wie viele Möglichkeiten insgesamt existieren.

countParts

Die bisher betrachtete Folge wird also um ein Zeichen erweitert. Dazu muss bei jedem Zeichen festgestellt werden, ob es schon einmal vorkam. Wenn es sich um eine Wiederholung handelt, muss die Anzahl der Möglichkeiten vor dem letzten Auftreten des Zeichen ermittelt werden und vom doppelten der Anzahl der bisherigen Möglichkeiten subtrahiert werden. Ansonsten kann die Anzahl der Möglichkeiten einfach verdoppelt werden.
Damit möglichst schnell ermittelt werden kann, wie viele Möglichkeiten es bei dem letzten Vorkommen gab, wird für jedes mögliche Zeichen die Anzahl der Möglichkeiten bei eben diesem letzten Vorkommmen in einem Array namens `lastParts` gespeichert, wo sie einfach gefunden und ausgelesen werde können.
Im Prinzip reicht das schon aus, um die Anzahl der Möglichkeiten zu berechnen, allerdings werden die entstehenden Zahlen zu groß, um sie in einem normalen Datentyp zu speichern, weshalb ich einen eigenen namens `BigNumber` programmiert habe.
Dieser speichert die Zahlen im 10^9-System in einem dynamischen Array von integer, wobei jedes Element des Arrays eine Zahl zwischen 0 und 999999999

enthält und das erste Element das höchstwertige ist.
Eine Alternative zu diesem Verfahren, wäre es eine auf dem binären statt auf dezimalen System basierende Basis zu wählen, z.B.: 2^32, wodurch das Arrays für eine Zahl nur ein Viertel von dem für eines mit dem 10^9-System benötigten Speicherplatz belegt hätte und die Addition und Subtraktion fast viermal so schnell gewesen wäre.
Das hätte aber den Nachteil, dass man die Zahl zur Ausgabe in das dezimale Zahlensystem umwandeln müsste, was im Vergleich zur einfachen Ausgabe bei einem dezimalen System sehr viel umständlicher erscheint. Außerdem wäre der Beweis für die Korrektheit der Ausgabefunktion unübersichtlicher geworden.
Durch die Verwendung eines nicht nativen Datentyps kann man natürlich nicht mehr die normalen Rechenoperationen gebrauchen, weshalb ich diese auch neu programmiert habe.
Die Addition wird von der Procedure **add** erledigt, die zwei `BigNumber`s `n1` und `n2` als Parameter entgegen nimmt. Die berechnete Summe wird dabei in dem Array von `n1` gespeichert. **add**
Die Berechnung läuft, wie bei dem alltäglichen schriftlichen Rechenverfahren. Es werden immer je zwei Ziffern, beginnend mit der am niedrigstwertigen, addiert, und bei einem Überlauf - der wegen dem 10^9-ner System zum Glück noch in den integer passt - die nächst höhere Stelle inkrementiert. Dabei gibt es nur noch ein Problem, wenn der Überlauf in der höchstwertigsten Stelle auftritt (wie z.B. bei $9 + 8$ im Dezimalsystem). Dann nämlich müsste die Zahl um eine Stelle verlängert werden. Da das relativ langsam ist (auf jeden Fall im Vergleich zum gleich lang lassen), gehe ich einfach davon aus, dass die Zahl immer genügend führende Nullen hat. Damit diese Annahme berechtigt ist, berechne ich die Anzahl der Stellen bei der maximalen Anzahl von Möglichkeiten, also $log_{10^9}2^n$ bei einer n-langen Folge und setzt die Länge jeder verwendeten Zahl auf diese Stellenanzahl.
Die Subtraktionsroutine **sub** arbeitet in äquivalenter Weise, nur werden die Ziffern eben subtrahiert, statt addiert, und der Überlauf tritt auf, wenn das Ergebnis bei einer Ziffer kleiner als 0 ist, und nicht wenn es größer gleich 10^9 ist. Dank `integer` passt auch dieser Überlauf noch in den Speicher und kann wie oben behandelt werden. **sub**
Andere Rechenoperationen auf dem `BigNumber` werden nicht benötigt, da ja nur Verdopplungen (erledigt die Additíonsroutine) und Subtraktionen vorkommen. Die Umwandlung der `BigNumber` in einen String wird von der **toStr** übernommen, die die umzuwandelte `BigNumber` als Parameter entgegen **toStr**

nimmt und den Ergebnisstring als Rückgabewert zurückgibt. (Was könnte man auch sonst mit einem Rückgabewert machen?)
Dank der Verwendung eines 10^9-Systems geht die Ausgabe relativ simpel, es reicht einen leeren (bzw. mit Nullen gefüllten) String mit der neunfachen Länge der `BigNumber` zu erzeugen, jede Stelle der `BigNumber` mit den Standardfunktionen in einen String umzuwandeln und diesen in den Ergebnisstring zu kopieren.
Man muss dabei nur auf führende Nullen achten, das erste Mal, wenn man die in einen String umgewandelten Stellen in den Ergebnisstring kopiert (da sie nicht unbedingt 9 Stellen haben und man sie nach rechts ausrichten muss) und das zweitemal, wenn man alle Stellen der `BigNumber` kopiert hat. (dann muss man nämlich die führenden Nullen löschen).
Aufgerufen wird die am Anfang beschriebene Funktion **countParts** von der Methode **countClick**, die beim Klick auf den Button `count` aufgerufen wird. **countClick**
Diese liest den String mit den Übungen entweder aus dem Editfeld `folge` aus, oder aus einer Datei deren Name in dem eben erwähnten Editfeld eingegeben wird. (welcher Fall konkret eintrit,hängt von der Markierung des Radiobuttons `direct` ab).
Das Ergebnis wird über das Editfeld `calculatedCount` ausgegeben, und während der Berechnung zeigt der Fortschrittsbalken `ProgressBar1` an, wie weit die Rechnung ist (was auch sonst). Die Position des Balkens wird in der Funktion **countParts** gesteuert.
Das Programm kann die Möglichkeiten zusätzlich zum Zählen auch noch auflisten (allerdings natürlich nur bei kurzen Strings), wofür die Methode **listClick** zuständig ist. **listClick**
Zuerst werden alle möglichen Teilfolgen berechnet und danach alle mehrfach vorkommenden herausgefiltert.
Da die Reihenfolge der Übungen in einem Teilstring dieselbe ist, wie in der Ausgangsfolge kann jede Teilfolge vollständig dadurch beschrieben werden, welche Übungen vorkommen. **listClick** kann also alle Teilfolgen eines Strings mit Länge n dadurch erzeugen, dass alle Zahlen von 0 bis $2^n - 1$ berechnet, sie als Bitfolge betrachtet und nur die durch Einsen markierten Übungen für die Teilfolge nimmmt.
Diese werden dann in einer Stringlist `sl` gespeichert, die dann sortiert wird, um jede Teilfolge nur einmal zu haben. Nach der Sortierung kann man die Liste nämlich einfach durchlaufen, und jede Teilfolge, die gleich zu der vorherigen, ist löschen. (bzw. alle, für die das nicht zutrifft, in eine Extraliste kopieren).

Um sicher zu stellen, dass das Programm fehlerfrei arbeitet, habe ich bei den für das Zählen der Möglichkeiten zuständigen Teilen, jede wichtige Anweisung mit einem mathematischen Beweis für ihre Korrektheit versehen, die man im Quellcodeausdruck sehen kann. (die dabei verwendeten Summenzeichen sind natürlich nachträglich eingefügt worden).
Theoretisch müssten die so bewiesenen Programmzeilen fehlerfrei sein. Praktisch kommt es aber häufig vor, dass man bei einer Analyse einige Aspekte übersieht, wie zum Beispiel Datentyp oder Arraygrenzen. Bei diesem Programm ist es mir sogar manchmal passiert, dass eine Funktion, nachdem ich bewiesen hatte, dass sie fehlerfrei ist, nicht mehr funktioniert hat, weil ich írgendwo einen falschen Wert eingesetzt hatte, um den Beweis zu erleichtern. Gegen solche Unaufmerksamkeiten kann man eigentlich nichts machen, außer es nochmals zu überprüfen (lassen) und mit Beispielwerten zu testen.
Außerdem muss man bei solchen Beweisen immer davon ausgehen, dass der Compiler und die Hardware korrekt arbeiten.

Kapitel 3

Ablaufprotokoll

RTRT

R : $2 * 1$
RT : $2 * 2$
RTR : $2 * 4 - 1$
$RTRT$: $2 * 7 - 2$

$\Rightarrow$ 12 Möglichkeiten
Die möglichen Teilfolgen sind:
(leere Menge),R,RR,RRT,RT,RTR,RTRT,RTT,TTR,TRT,TT

RTRBRTBTTRRBTR

R : $2 * 1$
RT : $2 * 2$
RTR : $2 * 4 - 1$
$RTRB$: $2 * 7$
$RTRBR$: $2 * 14 - 4$
$RTRBRT$: $2 * 24 - 2$
$RTRBRTB$: $2 * 46 - 7$
$RTRBRTBT$: $2 * 85 - 24$
$RTRBRTBTT$: $2 * 146 - 85$
$RTRBRTBTTR$: $2 * 207 - 14$
$RTRBRTBTTRR$: $2 * 400 - 207$
$RTRBRTBTTRRB$: $2 * 593 - 46$
$RTRBRTBTTRRBT$: $2 * 1140 - 146$
$RTRBRTBTTRRBTR$: $2 * 2134 - 400$

$\Rightarrow$ 3868 Möglichkeiten

Die ersten 25 Teilfolgen sind:
(l.M.), B, BB, BBB, BBBR, BBBT, BBBTR, BBR, BBRB, BBRBR, BBRBT, BBRBTR, BBRR, BBRRB, BBRRBR, BBRRBT, BBRRBTR, BBRRR, BB, RT, BBRRTR, BBRT, BBRTR, BBT, BBTB, BBTBR

RTBRSTBTRBTRRTS^{5}RRTTSTRTTSBBRTBRSBRRBBBSRBRBB

R :	$2 * 1$
RT :	$2 * 2$
RTB :	$2 * 4$
$RTBR$:	$2 * 8 - 1$
$RTBRS$:	$2 * 15$
$RTBRST$:	$2 * 30 - 2$
$RTBRSTB$:	$2 * 58 - 4$
$RTBRSTBT$:	$2 * 112 - 30$
$RTBRSTBTR$:	$2 * 194 - 8$
$RTBRSTBTRB$:	$2 * 380 - 58$
$RTBRSTBTRBT$:	$2 * 702 - 112$
$RTBRSTBTRBTR$:	$2 * 1292 - 194$
$RTBRSTBTRBTRR$:	$2 * 2390 - 1292$
$RTBRSTBTRBTRRT$:	$2 * 3488 - 702$
$RTBRSTBTRBTRRTS$:	$2 * 6274 - 15$
$RTBRSTBTRBTRRTSS$:	$2 * 12533 - 6274$
$RTBRSTBTRBTRRTSSS$:	$2 * 18792 - 12533$
$RTBRSTBTRBTRRTS^4$:	$2 * 25051 - 18792$
$RTBRSTBTRBTRRTS^5$:	$2 * 31310 - 25051$
$RTBRSTBTRBTRRTS^5R$:	$2 * 37569 - 2390$
[...]	
$RTBRST\,[\ldots]\,RBBBS$:	$2 * 11435237950 - 337683843$
$RTBRST\,[\ldots]\,RBBBSR$:	$2 * 22532792057 - 2301944226$
$RTBRST\,[\ldots]\,RBBBSRB$:	$2 * 42763639888 - 8744137744$
$RTBRST\,[\ldots]\,RBBBSRBR$:	$2 * 76783142032 - 22532792057$
$RTBRST\,[\ldots]\,RBBBSRBRB$:	$2 * 131033492007 - 42763639888$
$RTBRST\,[\ldots]\,RBBBSRBRBB$:	$2 * 219303344126 - 131033492007$

$\Rightarrow$ 307573196245 Möglichkeiten

Die Verwandlung

$A:$	$2*1$
$AL:$	$2*2$
$ALS:$	$2*4$
$ALSG:$	$2*8$
$ALSGR:$	$2*16$
$ALSGRE:$	$2*32$
$ALSGREG:$	$2*64-8$
$ALSGREGO:$	$2*120$
$ALSGREGOR:$	$2*240-16$
$ALSGREGORS:$	$2*464-4$
$ALSGREGORSA:$	$2*924-1$
$ALSGREGORSAM:$	$2*1847$
$ALSGREGORSAMS:$	$2*3694-464$

$\Rightarrow$ 213265550539852359962451036087185836707067840195010873078
172138095182477532897480133148602941550597 9 [. . .] 887617112250
276368163379929808743290477742497190975058338 6754043415721
536620121367007419693193866488 Möglichkeiten

Kapitel 4

Quellcode

```
unit a2_u;  [...]
type
  BigNumber=array of integer; //Datentyp für große Zahlen
  TDWORDArray=array of cardinal;//Speichert die verwendeten Übungen
  TPartFunction=function (const part:TDWORDArray;const data:pointer):boolean;
  TForm1 = class(TForm)
    folge: TEdit;                 //Eingabefeld für die gebaute Folge von Übungen
    count: TButton;               //Button zum Zählen aller Möglichkeiten
    list: TButton;                //Button zum Auflisten aller Möglichkeiten
    calculatedCount: TEdit;       //Ausgabefeld für die Anzahl                    10
    possibilities: TMemo;         //Ausgabefeld für die Möglichkeiten
    ProgressBar1: TProgressBar; //Fortschrittsanzeige während der Berechnung
    fromFile: TRadioButton;       //gibt der Wert in folge eine Datei an
    direct: TRadioButton;         // oder eine Folge
    Title, Label1: TLabel;        //design
    procedure countClick(Sender: TObject);          //Zählt alle Möglichkeiten
    procedure directClick(Sender: TObject);         //unwichtig(design)
    procedure listClick(Sender: TObject);           //Listet alle Möglichkeiten auf
  end;
                                                                                  20
procedure add(n1:BigNumber;const n2:BigNumber); //Addiert 2 BigNumbers
procedure sub(n1:BigNumber;const n2:BigNumber); //Subtrahiert 2 BigNumbers
function toStr(n: BigNumber):string; //Wandelt eine BigNumber in einen String um
function countParts(const str: string):BigNumber; //Zählt alle Teilfolgen
procedure createParts(const func:TPartFunction;
                      numbers:TDWORDArray;const data:pointer); //Berechnet alle Teilfolgen
function partFound(const part:TDWORDArray;const slp:pointer):boolean; //Speichert eine Teilfolge

const BASE=1000000000;  //Verwendete Zahlenbasis 10^9
```

```
var Form1: TForm1;

implementation

{$R *.DFM}

//==================Zählen aller Teilfolgen================
{ Addiert zwei Zahlen n1 und n2 gleicher Länge und speichert das Ergebnis in n1

(*) Notationen
        n1[i]       = i-te Ziffer von n1 (n1[0] ist die höchstwertige)
        n2[i]       = i-te Ziffer von n2 (n2[0] ist die höchstwertige)
        n1#[i]      = i-te Ziffer von n1, vor Aufruf der Funktion
        n1'[i]      = i-te Ziffer von n1, nach der nächsten Anweisung
        length(n1)  = Anzahl der Ziffern in n1
        length(n2)  = Anzahl der Ziffern in n2   (wie bei Arrays)
        length(n1)  = m + 1
        BASE        = 1000000000 = Die verwendete Zahlenbasis

(*) Annahme:1. length(n1)     = length(n2)
        2. 0 ≤ n1[i]      ≤ BASE - 1 für alle i
        3. 0 ≤ n2[i]      ≤ BASE - 1 für alle i
        4. n1[0] + n2[0] + 1 < BASE
        5. n1[i], n2[i]       ist definiert für 0≤i≤m
           (folgt aber schon aus den Delphiarrayspezifikationen)


(*) Gewünschtes Ergebnis:
    1. sum_{i=0}^{m} n1'[i] * BASE^{m-i} = sum_{i=0}^{m} n1#[i] * BASE^{m-i} + sum_{i=0}^{m} n2[i] * BASE^{m-i}
    2. 0 ≤ n1'[i] ≤ BASE - 1 für alle 0≤i≤m
}
procedure add(n1:BigNumber;const n2:BigNumber);
var i:integer; //Stellenindex
    overflow: boolean; //gab es einen Überlauf nach oben?
begin
  {Arrayzugriffe wie n[i] sind definiert, wenn 0≤i≤m}

  overflow:=false;
  for i:=high(n1){m} downto 0 do begin //Es gilt immer: 0≤i≤m
    { n1[i] = n1#[i], da nur die Ziffern n[j] mit j > i verändert wurden.
      (Der Ausdruck ist definiert, da wegen der Schleife 0≤i≤m ist)
    }
    if overflow then begin //Fall 1: overflow = true
      {Voraussetzung:
```

$$\sum_{k=i+1}^{m} n1[k] * BASE^{m-k} = \sum_{k=i+1}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i+1}^{m} n2[k] * BASE^{m-k} - BASE^{m-i}$$

(erfüllt durch letzten Schleifendurchlauf)

(Arrayzugriffe auf k sind definiert, da auf Grund der Summendefinitionen i+1≤k≤m gilt, also 0≤k≤m)

}

n1[i]:=n1[i]+n2[i]+1

{ *Folgen der letzten Anweisung:*

$$\sum_{k=i}^{m} n1[k] * BASE^{m-k}$$

Zerlegung der Summe:

$$= \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} + n1[i] * BASE^{m-i}$$

Zerlegung der Summe:

$$= \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} + (n1\#[i] + n2[i] + 1) * BASE^{m-i}$$

Vorraussetzung einfügen:

$$= \sum_{k=i+1}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i+1}^{m} n2[k] * BASE^{m-k} - BASE^{m-i} + (n1\#[i] + n2[i] + 1) * BASE^{m-i}$$

Summen zusammenfassen:

$$= \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

(An den Summendefinitionen sieht man, dass nur Arrayzugriffe n1[k] und n2[k] mit i ≤ k ≤ m vor, die definiert sind, da wegen der Schleife 0 ≤ i ≤ m ist)

Es ist nicht garantiert, dass n1#[i] + n2[i] + 1 < BASE, also kann nun gelten: n1[i] >= BASE

Da n1#[i] >= 0 und n2[i] >= 0 ist auch n1[i] >= 0.

*Da n1#[i] ≤ BASE - 1 und n2[i] ≤ BASE - 1 ist n1 ≤ 2*BASE - 1*

(Ausdrücke sind definiert, da wegen der Schleife 0≤i≤m ist)

}

end else begin *//Fall 2: overflow = false*

{*Voraussetzung:*

$$\sum_{k=i+1}^{m} n1[i] * BASE^{m-i} = \sum_{k=i+1}^{m} n1\#[i] * BASE^{m-i} + \sum_{k=i+1}^{m} n2[i] * BASE^{m-i}$$

(im ersten Durchlauf sind beide Seiten 0, da dann i+1>m, ansonsten ist

sie durch den letzten Schleifendurchlauf erfüllt)

(An den Summendefinitionen sieht man, dass nur Arrayzugriffe n1[k] und n2[k] mit $i \leq k \leq m$ vor, die definiert sind, da wegen der Schleife $0 \leq i \leq m$ ist)

}

n1[i]:=n1[i]+n2[i];

{ *Folgen der letzten Anweisung:*

$\sum_{k=i}^{m} n1[k] * BASE^{m-k}$

$= \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} + n1[i] * BASE^{m-i}$ *Zerlegung der Summe*

$= \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} + (n1\#[i] + n2[i]) * BASE^{m-i}$ | *Zuweisung einsetzen*

$= \sum_{k=i+1}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i+1}^{m} n2[k] * BASE^{m-k}$

$+ (n1\#[i] + n2[i]) * BASE^{m-i}$ | *Vorraussetzung einfügen*

$= \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i}^{m} n2[k] * BASE^{m-k}$| *Summen zusammenfassen*

(An den Summendefinitionen sieht man, dass nur Arrayzugriffe n1[k] und n2[k] mit $i \leq k \leq m$ vor, die definiert sind, da wegen der Schleife $0 \leq i \leq m$ ist)

Es ist nicht garantiert, dass n1#[i] + n2[i] < BASE, also kann nun gelten: n1[i] >= BASE
Da n1#[i] >= 0 und n2[i] >= 0 ist auch n1[i] >= 0.
*Da n1#[i] ≤ BASE - 1 und n2[i] ≤ BASE - 1 ist n1 ≤ 2*BASE - 2*

(Ausdrücke sind definiert, da wegen der Schleife $0 \leq i \leq m$ ist)

}

end;

{ *Es gilt nun:*

1. $\sum_{k=i}^{m} n1[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i}^{m} n2[k] * BASE^{m-k}$

2. n1[i] >= 0

*3. n1[i] ≤ 2*BASE - 1*

Möglicherweise ist aber n1[i] >= BASE

(Ausdrücke sind definiert, da nur Arrayzugriffe n1[k], n2[k] und n1[i] mit $i \leq k \leq m$ vorkommen und $0 \leq i \leq m$ ist)

}

if n1[i]<BASE **then begin**

{

Da n1[i] < BASE ist, sind die an das Ergebnis gestellten Bedingungen erfüllt.

}
overflow:=false
{
Da nun overflow=false ist, wird nun im nächsten Durchlauf Fall 2 eintreten, und die dort beschriebene Vorraussetzung ist erfüllt.
}
end else begin
{ *Es gilt: n1[i] >= BASE.*

*Da n1[i] ≤ 2*BASE - 1 ist n1[i] - BASE ≤ BASE - 1.*
Da n1[i] >= BASE ist n1[i] - BASE >= 0

Die zweite an das Ergebnis gestellt Bedingung ist also für n1'[i] erfüllt

Für die Summe gilt dann:

$$\sum_{k=i}^{m} n1[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

$$\Leftrightarrow n1[i] * BASE^{m-k} + \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

$$\Leftrightarrow (n1'[i] + BASE) * BASE^{m-i} + \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

$$\Leftrightarrow n1'[i] * BASE^{m-i} + BASE^{m-i+1} + \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

$$\Leftrightarrow BASE^{m-i+1} + \sum_{k=i}^{m} n1'[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

$$\Leftrightarrow \sum_{k=i}^{m} n1'[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i}^{m} n2[k] * BASE^{m-k} - BASE^{m-i+1}$$

$$\Leftrightarrow \sum_{k=i}^{m} n1'[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} + \sum_{k=i}^{m} n2[k] * BASE^{m-k} - BASE^{m-(i-1)}$$

(Ausdrücke sind definiert, da nur Arrayzugriffe n1[k], n2[k] und n1[i] mit i≤k≤m vorkommen und 0≤i≤m ist)

```
      Im nächsten Durchlauf, wenn i' = i-1 ist, stimmt also die Vorraussetung für Fall 1
      }
      n1[i]:=n1[i]−BASE; //Vorraussetzungen erfüllen (s.o)
      overflow:=true;     //Fall 2 eintreten lassen
    end;
  end;

  {
    Da nach Annahme 4 gilt n1#[0] + n2[0] + 1 < BASE gilt, ist overflow = false.
    Das heißt, dass die Vorraussetzung aus Fall 2 gilt, und das ist gerade die
    an das Ergebnis gestellte Bedingung. (Wenn man die jetztige Situation als
    Schleifenanfang mit i=-1 betrachtet).
  }
end;

{ Subtrahiert zwei Zahlen mit gleicher Länge
(*) Notationen
            n1[i]       = i-te Ziffer von n1 (n1[0] ist die höchstwertige)
            n2[i]       = i-te Ziffer von n2 (n2[0] ist die höchstwertige)
            n1#[i]      = i-te Ziffer von n1, vor Aufruf der Funktion
            n1'[i]      = i-te Ziffer von n1, nach der nächsten Anweisung
            length(n1)  = Anzahl der Ziffern in n1
            length(n2)  = Anzahl der Ziffern in n2   (wie bei Arrays)
            length(n1)  = m + 1
            BASE        = 1000000000 = Die verwendete Zahlenbasis

(*) Annahme:1. length(n1)     = length(n2) = m - 1
            2. 0 ≤ n1[i]      ≤ BASE - 1 für alle i
            3. 0 ≤ n2[i]      ≤ BASE - 1 für alle i
            4. n1[0] + n2[0] + 1 < BASE
            5. n1[i], n2[i]       ist definiert für 0≤i≤m
               (folgt aber schon aus den Delphiarrayspezifikationen)

(*) Gewünschtes Ergebnis:
    1. ∑_{i=0}^{m} n1'[i] * BASE^{m−i} = ∑_{i=0}^{m} n1#[i] * BASE^{m−i} − ∑_{i=0}^{m} n2[i] * BASE^{m−i}
    2. 0 ≤ n1'[i] ≤ BASE - 1 für alle i
}
procedure sub(n1:BigNumber;const n2:BigNumber);
var i:integer; //Stellenindex
    overflow: boolean; //gab es einen Überlauf nach unten (ins negative)?
begin
  overflow:=False;
  for i:=high(n1){m} downto 0 do begin
```

{ *n1[i] = n1#[i], da nur die Ziffern n[j] mit j > i verändert wurden.* }
if overflow **then begin**
{*Voraussetzung:*

$$\sum_{k=i+1}^{m} n1[k] * BASE^{m-k}$$
$$= \sum_{k=i+1}^{m} n1\#[k] * BASE^{m-k} - \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} + BASE^{m-i}$$

(erfüllt durch letzten Schleifendurchlauf)

(Arrayzugriffe auf k sind definiert, da auf Grund der Summenindizes i+1≤k≤m gilt, also 0≤k≤m)
}
n1[i]:=n1[i]−n2[i]−1
{ *Folgen der letzten Anweisung:*

$$\sum_{k=i}^{m} n1[k] * BASE^{m-k}$$
$$= \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} + n1[i] * BASE^{m-i} \text{ Zerlegung der Summe}$$
Zuweisung einsetzen:
$$= \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} + (n1\#[i] - n2[i] - 1) * BASE^{m-i}$$
$$= \sum_{k=i+1}^{m} n1\#[k] * BASE^{m-k} - \sum_{k=i+1}^{m} n2[k] * BASE^{m-k}$$
$$+ BASE^{m-i} + (n1\#[i] + n2[i] + 1) * BASE^{m-i} \mid \text{ Vorraussetzung einfügen}$$
$$= \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} - \sum_{k=i}^{m} n2[k] * BASE^{m-k} \mid \text{ Summen zusammenfassen}$$

(Auf Grund der Summendefinitionen kommen nur Arrayzugriffe n1[k] und n2[k] mit i ≤ k ≤ m vor, die definiert sind, da wegen der Schleife 0≤i≤m ist)

Es ist nicht garantiert, dass n1#[i] - n2[i] - 1 >= 0, also kann nun gelten: n1[i] < 0
Da n1#[i] >= 0 und n2[i] < BASE ist auch n1[i] >= - BASE .

(Ausdrücke sind definiert, da wegen der Schleife 0≤i≤m ist)

(Auf Grund der Summendefinitionen kommen nur Arrayzugriffe n1[k] und n2[k] mit i ≤ k ≤ m vor, die definiert sind, da wegen der Schleife 0≤i≤m ist)
}
end else begin
{*Voraussetzung:*

$$\sum_{k=i+1}^{m} n1[i] * BASE^{m-i} = \sum_{k=i+1}^{m} n1\#[i] * BASE^{m-i} - \sum_{k=i+1}^{m} n2[i] * BASE^{m-i}$$

```
    (im ersten Durchlauf sind beide Seiten 0, da dann i+1>m, ansonsten ist
     sie durch den letzten Schleifendurchlauf erfüllt)
  }
  n1[i]:=n1[i]−n2[i];
   { Folgen der letzten Anweisung:
     \sum_{k=i}^{m} n1[k] * BASE^{m-k}
     = \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} + n1[i] * BASE^{m-i}| Zerlegung der Summe
     = \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} + (n1#[i] − n2[i]) * BASE^{m-i}| Zuweisung einsetzen
     = \sum_{k=i+1}^{m} n1#[k] * BASE^{m-k} − \sum_{k=i+1}^{m} n2[k] * BASE^{m-k}
       + (n1#[i] − n2[i]) * BASE^{m-i}| Vorraussetzung einfügen
     = \sum_{k=i}^{m} n1#[k] * BASE^{m-k} − \sum_{k=i}^{m} n2[k] * BASE^{m-k} | Summen zusammenfassen

     Wie man an den Summendefintionen sieht, kommen in den Gleichungen nur
      Arrayzugriffe n1[k] und n2[k] mit i ≤ k ≤ m vor, die
      definiert sind, da wegen der Schleife 0 ≤ i ≤ m ist

     Es ist nicht garantiert, dass n1#[i] >= n2[i] also kann nun gelten:
       n1[i] = n1#[i] - n2[i] < 0
     Da n1#[i] >= 0 und n2[i] < BASE ist auch n1[i] >= - BASE.

     (Ausdrücke sind definiert, da wegen der Schleife 0≤i≤m ist)
   }
end;
{ Es gilt nun:
  1. \sum_{k=i}^{m} n1[k] * BASE^{m-k} = \sum_{k=i}^{m} n1#[k] * BASE^{m-k} − \sum_{k=i}^{m} n2[k] * BASE^{m-k}
  2. n1[i] < BASE
  3. n1[i] >= - BASE - 1
  Möglicherweise ist aber n1[i] < 0

  (Ausdrücke sind definiert, da nur Arrayzugriffe n1[k], n2[k] und n1[i] mit i≤k≤m
  vorkommen und 0≤i≤m ist)
}
if n1[i]>=0 then begin
  {
    Da n1[i] >= 0 ist, sind die an das Ergebnis gestellten Bedingungen erfüllt.
  }
  overflow:=false
```

{
Da nun overflow=false ist, wird nun im nächsten Durchlauf Fall 2 eintreten, und die dort beschriebene Vorraussetzung ist erfüllt.
}
end else begin
{ *Es gilt: n1[i] < 0*

Da n1[i] >= - BASE ist gilt: n[i] = n1[i] + BASE >= 0
Da n1[i] < 0 ist gilt: n[i] = n1[i] + BASE < BASE.

Die zweite an das Ergebnis gestellt Bedingung ist also für n1'[i] erfüllt.

Für die Summe gilt dann:

$$\sum_{k=i}^{m} n1[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} - \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

$$\Leftrightarrow n1[i] * BASE^{m-k} + \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} - \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

$$\Leftrightarrow (n1'[i] - BASE) * BASE^{m-i} + \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} - \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

$$\Leftrightarrow n1'[i] * BASE^{m-i} - BASE^{m-i+1} + \sum_{k=i+1}^{m} n1[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} - \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

$$\Leftrightarrow -BASE^{m-i+1} + \sum_{k=i}^{m} n1'[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} - \sum_{k=i}^{m} n2[k] * BASE^{m-k}$$

$$\Leftrightarrow \sum_{k=i}^{m} n1'[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} - \sum_{k=i}^{m} n2[k] * BASE^{m-k} + BASE^{m-i+1}$$

$$\Leftrightarrow \sum_{k=i}^{m} n1'[k] * BASE^{m-k} = \sum_{k=i}^{m} n1\#[k] * BASE^{m-k} - \sum_{k=i}^{m} n2[k] * BASE^{m-k} + BASEm - (i-1)$$

Daher ist im nächsten Durchlauf, wenn $i' = i - 1$ ist, die Vorraussetung für Fall 1 erfüllt

(Ausdrücke sind definiert, da nur Arrayzugriffe n1[k], n2[k] und n1[i] mit i≤k≤m

```
      vorkommen und 0≤i≤m ist)
      }
      overflow:=true;                                                    330
      n1[i]:=n1[i]+BASE;
    end;
  end;
  {
    Da nach Annahme 4 gilt n1#[0] - n2[0] - 1 >=0 gilt, ist overflow = false.
    Das heißt, dass die Vorraussetzung aus Fall 2 gilt, und das ist gerade die
    an das Ergebnis gestellte Bedingung. (Wenn man die jetztige Situation als
    Schleifenanfang mit i=-1 betrachtet).
  }
end;                                                                     340


{ Wandelt die übergebene BigNummer in einen String um

(*) Notationen
            n[i]          = i-te Ziffer von n (n[0] ist die höchstwertige)
            length(n)     = Anzahl der Ziffern in n = m + 1
            BASE          = 1000000000 = Die verwendete Zahlenbasis

            result[i]     = i-te Ziffer vom Ergebnis                 350
            length(result) = Anzahl der Ziffern in result = n

(*) Annahme: 0 ≤ n1[i]      ≤ BASE - 1 für alle i
            n1[i]    ist definiert für 0≤i≤m

(*) Gewünschtes Ergebnis:
     1. sum_{i=1}^{n} result[i] * 10^(n-i) = sum_{i=0}^{m} n[i] * BASE^(m-i)
     2. 0 ≤ result[i] ≤ 9 für alle i
     3. (result[1] <> 0) oder (result = '0')
}                                                                        360

function toStr(n: BigNumber):string;
var i:integer; //Stellenindex
    t:string; //Eine in einen String umgewandelte Stelle
begin
  SetLength(result,length(n) * 9); //result[i] ist nun definiert für 1≤i≤n=(m+1)*9
  FillChar(result[1],length(result),ord('0'));

  {Es gilt nun die zweite Bedingung, nämlich dass result nur Zeichen zwischen
   0 und 9 enthalten darf}                                               370
```

for i:=0 **to** high(n) **do begin**
{*Schleifeninvarianten:*

1. $\sum_{j=1}^{i*9} result[j] * 10^{n-j} = \sum_{j=0}^{i-1} n[j] * BASE^{m-j}$
   (im ersten Durchlauf sind beide Seiten 0)
   *(der Ausdruck ist definiert, da nur Arrayzugriffe result[j] mit 1≤j≤i*9≤m*9≤n und n[j] mit 0≤j≤i-1≤m-1 vorkommen)*

2. *result[j] = 0 für alle j mit i*9≤j≤m*9+8*
   (im ersten Durchlauf ist result[j] = 0 für alle j)

}
t:=IntToStr(n[i]);
{*Für temp gilt:*

1. $\sum_{i=1}^{length(temp)} temp[i * 10^{length(temp)-i}] = n[i]$
2. *temp enthält nur Zeichen zwischen 0 und 9.*
3. *length(temp) ≤ 9, da temp = n[i] < 1 000 000 000*

Folgen der nächsten Anweisung:
Es werden length(t) Ziffern ersetzt, was zuerst einmal bedeutet:

$$result'[j] = \begin{cases} result[j] & \text{für } (j \le i*9+9-length(t)) \text{ oder } (j > i*9+9) \\ t[j - (i*9+9-length(t))] & \text{sonst} \end{cases}$$

*Da result[j] = 0 für alle j mit i*9≤j≤m*9+9, ist das äquivalent zu:*

$$result'[j] = \begin{cases} result[j] & \text{für } (j \le i*9+9-length(t)) \text{ oder } (j > i*9+9) \\ t[j - (i*9+9-length(t))] & \text{für } i*9+10-length(t) \le j \le i*9+9 \\ 0 & \text{sonst} \end{cases}$$

$$\begin{aligned}
&\sum_{j=1}^{i*9+9} result'[j] * 10^{n-j} \\
&= \sum_{j=1}^{i*9+9} result[j] * 10^{n-j} + \sum_{j=i*9+1}^{i*9+9} result'[j] * 10^{n-j} \\
&= \sum_{j=1}^{i*9+9} result[j] * 10^{n-j} \\
&\quad + \sum_{j=i*9-(i*9+9-length(t))+1}^{i*9+9-(i*9+9-length(t))} result'[j + (i*9+9-length(t))] * 10^{n-j-(i*9+9-length(t))} \\
&= \sum_{j=1}^{i*9+9} result[j] * 10^{n-j} + \sum_{j=1}^{length(t)} t[j] * 10^{n-j-(i*9+9-length(t))}
\end{aligned}$$

$$= \sum_{j=1}^{i*9+9} result[j] * 10^{n-j} + \sum_{j=1}^{length(t)} t[j] * 10^{length(t)-j+n-i*9-9}$$
$$= \sum_{j=1}^{i*9+9} result[j] * 10^{n-j} + \sum_{j=1}^{length(t)} t[j] * 10^{length(t)-j} * 10^{n-i*9-9}$$
$$= \sum_{j=0}^{i-1} n[j] * BASE^{m-j} + n[i] * 10^{n-i*9-9}$$
$$= \sum_{j=0}^{i-1} n[j] * BASE^{m-j} + n[i] * 10^{(m+1)*9-i*9-9}$$
$$= \sum_{j=0}^{i-1} n[j] * BASE^{m-j} + n[i] * 10^{m*9-i*9}$$
$$= \sum_{j=0}^{i-1} n[j] * BASE^{m-j} + n[i] * 10^{m-i}$$
$$= \sum_{j=0}^{i-1} n[j] * BASE^{m-j} + n[i] * BASE^{m-i}$$
$$= \sum_{j=0}^{i} n[j] * BASE^{m-j}$$

wodurch die Schleifeninvariante für den nächsten Durchlauf bewiesen ist

Alle Ausdrücke sind definiert, da nur Arrayzugriffe auf result[j] mit $1 \leq j \leq i * 9 + 9 \leq n$ *auf result'[j] mit i*9+1≤j≤i*9+9 mit 1≤i*9+1, auf t[j] mit 1≤j≤length(t) und auf n[j] mit 0≤j≤i≤m erfolgen.*

```
}
  move(t[1],result[i*9+10-length(t)],length(t));
end;

{Bedingungen 1 und 2 für das Ergbenis sind erfüllt, jetzt kommt die dritte}

for i:=1 to length(result) do
  if result[i] <> '0' then begin //also result[j]=0 für alle j<i
```

$$\{ \quad \sum_{j=1}^{n} result'[j] * 10^{n-j})$$
$$= \sum_{j=1}^{n} result[j] * 10^{n-j}$$
$$= \sum_{j=1}^{n} result[j] * 10^{n-j} - \sum_{j=1}^{i-1} result[j] * 10^{n-j}$$
$$= \sum_{j=1}^{n} result[j] * 10^{n-j} - 0 \text{ da result[j]=0 für alle } j<i$$
$$= \sum_{j=1}^{n} n[j] * BASE^{m-i}$$

alle Bedingungen sind erfüllt

```
        (Arrayszugriffe sind definiert, da nur Zugriffe auf result(')[j] mit
        1≤j≤n und auf n[j] mit 0≤j≤m erfolgen.
      }
      delete(result,1,i−1);
      exit;
    end;

  //das heißt result[i] = 0 für alle i, da ansonsten in der vorherigen
  //for−Schleife die Prozedur beendet worden wäre.
  Result:='0';
end;


(*
 Gegeben:
     str         eine Zeichenfolge
     n           length(str)

  Notation:
     str[1..i]   der String der entsteht, wenn man nur die ersten i Zeichen nimmt
     teilfolgen[1..i]   Die Anzahl der möglichen Teilfolgen von str[1..i]
     last[i]     Das größte j mit j<i und str[j] = str[i], oder 0 falls kein
                 solches existiert

  Gesucht:
    Anzahl aller möglichen (unterschiedlichen) Teilfolgen von str,
    also teilfolgen[1..n]


*)
function countParts(const str: string):BigNumber;
var i:integer; //Index der aktuell betrachteten Stelle
    lastParts: array[#0..#255] of BigNumber; //Alte Möglichkeitenanzahlen
    c: char; //Aktuelles Zeichen/Übung

    numberLen,numberSize:integer; //Maximale Stellenzahl und dafür nötiger Speicher
    oldPossibilities:BigNumber; //Aktuelle Möglichkeitenanzahl
begin
//Maximale Stellenzahl berechnen:
  numberLen:=trunc(length(str)*ln(2)/ln(BASE))+1;//numberLen := log_BASE 2^n
  {Das ist wirklich die maximale Stellenzahl:
    BASE^numberLen
    = BASE^trunc(length(str)*ln(2)/ln(BASE))+1
    > BASE^length(str)*ln(2)/ln(BASE)−1+1
    = BASE^length(str)*ln(2)/ln(BASE)
```

```
  = BASE^(length(str)*ln(2)/ln(BASE))
  = 2^length(str)
}
numberSize:=numberLen*sizeof(integer);

//Speicher für result und oldPossibilities reservieren
SetLength(result,numberLen);
SetLength(oldPossibilities,numberLen);
//result mit 1 initialisieren
zeromemory(@result[0],numberSize);                                    490
Result[high(result)]:=1;
//Alle Einträge in lastParts mit nil initialisieren
ZeroMemory(@lastParts,sizeof(lastParts));

//Fortschrittsbalken initialisieren
Form1.ProgressBar1.Max:=length(str);

{Es gilt:
  1. result = 1 = Anzahl der möglichen Teilfolgen eines leeren Strings
  2. last[1] = 0, da vor dem ersten Zeichen kein anderes kommen kann  500
  3. lastParts[c] = nil für alle c
}
for i:=1 to length(str) do begin
  {Schleifeninvarianten:
    1. result = teilfolgen[1..i-1]
    2. Ist last[i] = 0, dann ist lastParts[str[i]] = nil,
       ansonsten gab es einen Durchlauf mit i = last[i], und es gilt
       lastParts[str[i]] = teilfolgen[1..last[i]-1]
    3. 0 ≤ result[i] < BASE für alle i
  }                                                                   510
  c:=str[i];
  if lastParts[c]=nil then begin
    {Es gilt:
      last[i] = 0
    }
    SetLength(lastParts[c],numberLen);
    {Auswirkungen der nächsten Anweisung.
      lastPart'[c] := result = teilfolgen[1..i-1]

     Angenommen es gibt ein k mit last[k] = i, dann gilt:             520
       teilfolgen[1..last[k]-1] = teilfolgen[1..i-1] = lastPart'[c] = lastPart'[str[i]]
     Das heißt in diesem Durchlauf ist die zweite Schleifeninvariante erfüllt.
    }
    Move(result[0],lastParts[c][0],numberSize);
```

```
      {Auswirkungen der nächsten Anweisung.
        result' := result + result = 2*result, das heißt im nächsten Durchlauf
        ist die erste Schleifeninvariante erfüllt. (siehe Lösungsidee)

        Die Bedingungen für add sind erfüllt:
        1. length(result) = length(result)
        2. erfüllt nach dritter Schleifeninvariante
        3. erfüllt nach dritter Schleifeninvariante
      }
      add(result,result);
    end else begin
      {Auswirkungen der nächsten Anweisung.
        oldPossibilities = lastParts[c] = teilfolgen[1..last[k]-1]
      }
      move(lastParts[c][0],oldPossibilities[0],numberSize);
      {Auswirkungen der nächsten Anweisung.
        lastPart'[c] = result = teilfolgen[1..i-1]

       Angenommen es gibt ein k mit last[k] = i, dann gilt:
         teilfolgen[1..last[k]-1] = teilfolgen[1..i-1] = lastPart'[c] = lastPart'[str[i]]
       Das heißt in diesem Durchlauf ist die zweite Schleifeninvariante erfüllt.
      }
      move(result[0],lastParts[c][0],numberSize);
      {Auswirkungen der nächsten beiden Anweisung.
        result' = result + result - oldPossibilities = 2*result - teilfolgen[1..last[k]-1],
        das heißt im nächsten Durchlauf ist die erste Schleifeninvariante erfüllt.
        (siehe Lösungsidee)

        Die Bedingungen für add sind erfüllt:
        1. length(result) = length(result)
        2. erfüllt nach dritter Schleifeninvariante
        3. erfüllt nach dritter Schleifeninvariante

        Die Bedingungen für sub sind erfüllt:
        1. length(result) = numberLen = length(oldPossibilities)
        2. erfüllt nach dritter Schleifeninvariante
        3. erfüllt nach dritter Schleifeninvariante
      }
      add(result,result);
      sub(result,oldPossibilities);
    end;
    if i mod 50=0 then Form1.ProgressBar1.Position:=i;
  end;
end;
```

```
//Diese Methode ruft countParts mit den eingegebenen Werten auf, und gibt
  das Ergebnis aus
procedure TForm1.countClick(Sender: TObject);
var i:integer;
    s:string;//Übungsfolge
    f:file;
begin
  ProgressBar1.Visible:=true;
  if direct.Checked then//Der eingegebene String ist die Übungsfolge
    s:=folge.text                                                          580
  else begin//Der eingegebene String ist ein Dateiname
    //Datei öffnen und Inhalt in den String s kopieren
    AssignFile(f,folge.text);
    reset(f,1);
    SetLength(s,FileSize(f));
    BlockRead(f,s[1],filesize(f));
    CloseFile(f);

    //Alle Zeilenumbrüche in s löschen und
    //  bei anderen nicht Großbuchstaben eine Fehlermeldung ausgeben         590
    for i:=length(s)downto 1 do
      if not (s[i] in ['A'..'Z','Ü','Ö','Ä','ß',#$A]) then begin
        ShowMessage('error: '+inttostr(i)+': '+s[i-1]+s[i]+s[i+1]);
        exit;
      end else if s[i]=#$A then delete(s,i,1);
    //Sinnloser Weise überprüfen ob wirklich nur Großbuchstaben vorkommen
    for i:=length(s)downto 1 do
      if not (s[i] in ['A'..'Z','Ü','Ö','Ä','ß']) then begin
        ShowMessage('error: '+inttostr(i)+': '+s[i-1]+s[i]+s[i+1]);
        exit;                                                              600
      end;
  end;
  //Ergebnis ausgeben
  calculatedCount.text:='Es gibt '+ToStr(countParts(s))+' Teilfolgen';
  ProgressBar1.Visible:=false;
end;


procedure TForm1.directClick(Sender: TObject);
begin                                                                      610
  if direct.Checked then Title.Caption:='Folge der Übungen:'
  else Title.Caption:='Dateiname:';
end;
```

```
//Auflisten aller Teilfolgen
procedure TForm1.listClick(Sender: TObject);
var start:string;      //Ausgangsfolge
    max:integer;       //Maximale Anzahl von Teilfolgen
    i,j:integer;       //Schleifenindizes
    sel: string;       //Aktuelle Teilfolge
    selSize: integer; //Länge der aktuellen Teilfolge                    620
    sl:TStringList;    //Alle Teilfolgen
begin
  if not direct.Checked then //Keinen String aus der Datei laden
    raise Exception.create('String muss direkt eingegeben werden');
  start:=folge.text;
  if length(start)>31 then //Zu lang für 32-Bit Datentypen
    raise Exception.create('Folge zu lang');

  sl:=TStringList.create();
                                                                         630
  max:=1 shl length(start) - 1;
  for i:=0 to max do begin
    //i durchläuft zwischen 000..000 bis 111..111, alle möglichen Werte
    //0 gibt an, dass der Wert an dieser Stelle nicht genommen werden soll
    //1 gibt an, dass der Wert zu sel hinzugefügt werden soll
    setlength(sel,length(start));
    selSize:=0;
    for j:=1 to length(start) do //Alle Bits durchlaufen
      if 1 shl (j-1) and i <> 0 then begin //Bit gesetzt
        inc(selSize);             //Verlängern                           640
        sel[selSize]:=start[j]; //Zeichen eintragen
      end;
    setlength(sel,selSize);       //sel auf die tatsächliche Länge setzten
    sl.Add(sel);                  //In die Liste einfügen
  end;

  //Teilfolgen alphabetisch sortieren
  sl.sort;

  //Ausgabe Memo Feld mit nur einer leer Zeile initialisieren            650
  possibilities.Clear;
  possibilities.Lines.BeginUpdate;
  possibilities.lines.add('');
  //Doppelte Teilfolgen aus sl herausfiltern
  for i:=1 to sl.Count-1 do
    if (sl[i]<>sl[i-1]) then //Ist der vorherige Strin ein anderer (erstes Auftreten)
      possibilities.lines.add(sl[i]); //dann kann sl[i] ausgegeben werden
  possibilities.Lines.EndUpdate;
```

```
  calculatedCount.text:='Es gibt '+inttostr(possibilities.lines.Count)+' Teilfolgen';
  sl.free;
end;

end.
```