IPI WS23/24 Solutions

Igor Dimitrov

2023-10-30

Table of contents

Preface

Solutions of the assignment sheets for the lecture "IPI WS23/24" at Uni Heidelberg.

Zettel 01

Aufgabe 3

3.1

- In der VL beschriebe TM ist ein "Transducer", d.h. ein Automat, das aus einem Input ein Output produziert. Die Beschreibung in der Online-version definiert die TM als ein "Acceptor". D.h. ein Automat, das fuer eine gegebene Eingabe "Yes" oder "No" produziert. Jedoch kann die Online Version auch als ein Transducer betrieben werden.
- Die online Version erlaubt dem Schreib-/Lesekopf keine Bewegung bei einem Uebergang. Also darf der Kopf auf dem gleichen Feld bleiben. In der VL-version sind dagegen nur die Bewegungen "links" oder "rechts" definiert.
- Die Online-version hat einen "Blank" Symbol, die VL-version hingegen nicht.

3.2

Wie im Online-tutorial erklaert entsprechen die Zustaende der TM dem "Rechenfortschritt" der Berechnung. (Computational Progress).

Bei der "Even number of Zeros"-TM gibt es zwei Zustaende q_0 und q_1:

- q_0 entspricht der Situation, dass bis jetzt eine **gerade** Anzahl von 0's gelesen wurde.
- q_1 enptricht der Situation, dass bist gelesene Anzahl von 0's **ungerade** ist.

Oder kuerzer:

$$\begin{aligned} q_0 &\iff \#0's \equiv 0 \mod 2 \\ q_1 &\iff \#0's \equiv 1 \mod 2 \end{aligned}$$

Am Anfang der Berechnung ist die Anzahl der gelesenen 0's gleich 0. Somit ist q_0 der initiale Zustand. Die Uebergaenge sind so definiert, dass das Ablesen einer 0 einen Zustanduebergang $q_i \to q_{i\oplus 1}$ verursacht, wobei $i \oplus 1$ Addition mod 2 ist. Hingegen verursacht das Ablesen einer 1 keinen Zustanduebergang: $q_i \to q_i$ D.h. das Ablesen einer 0 'flippt' die Paritaet der 0's und Ablesen einer 1 hat keinen Einfluss darauf. Der Kopf bewegt sich rechts bis das 'Blank' erreicht wird. Falls dann der Zustand q_0 ist, ist ein Uebergang auf q_{accept} definiert und die Maschine akzeptiert somit die Eingabe. Sonnst sind keine Uebergange mehr definiert und die Berechnung terminiert in einem nicht-akzeptierenden Zustand.

Siehe Figure 1 und Figure 2 fuer die **Uebergangstabelle** und den **Ubergangsgraph**

Zustand	**Input**	**Operation**	**Next State**	**Comment**
*q0	0	0, >	q1	
	1	1, >	q0	Initialer Zustand
	_	_, -	qAccept	
q1	0	0, >	q0	
	1	1, >	q1	
qAccept				Endzustand

Figure 1: Uebergangstabelle

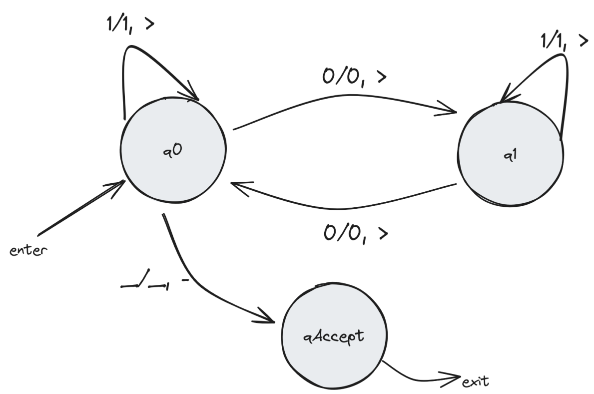


Figure 2: Uebergangsgraph

3.3

In der VL definierte TM enthaelt kein "Blank"-symbol. Stattdessen symbolisiert "0" das Ender einer Zeichenkette von Einsen. Da, in der Online-version es "Blanks" gibt, ersetzten wir 0 durch "Blanks".

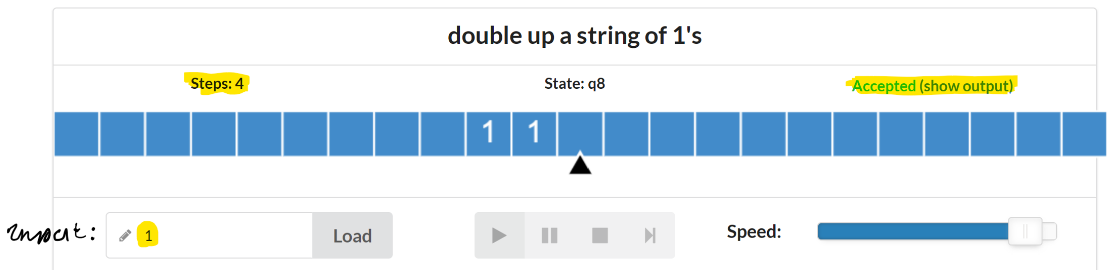
Das Programm zur Verdoppelung einer Einsenkette (Auch im Zip als txt datei enthalten):

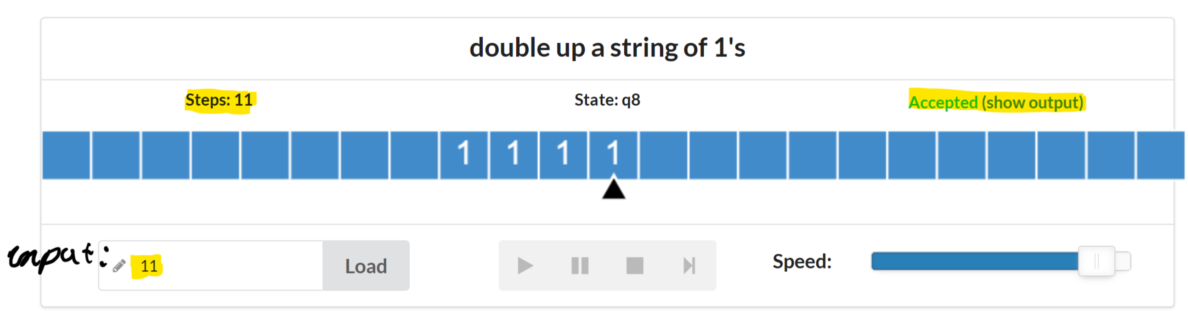
```
// Input: a string of 1's of length n
// Ouput: a string of 1's of length 2n
```

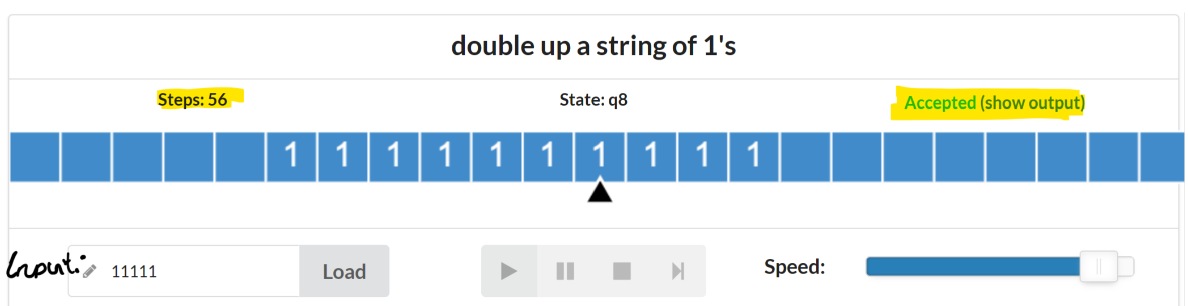
```
// Example: if 111 is given as input. The machine terminates at an
 ↪ accepting state
// with 111111 as the string on the band.
//
//


name: double up a string of 1's
init: q1
accept: q8

q1, 1
q2,X,>

q2,_
q3,Y,<

q2,1
q2,1,>

q3,1
q3,1,<

q3,X
q4,1,>

q4,1
q5,X,>

q4,Y
q8,1,>

q5,1
q5,1,>

q5,Y
q6,Y,>

q6,1
q6,1,>
```

```
q6,_
q7,1,<

q7,1
q7,1,<

q7,Y
q3,Y,<
```

Wir haben das Program auf die Inputs `1`, `11` und `11111` getestet und richtige Ergebnisse erhalten:







Zettel 02

Aufgabe 2

Idee: Vertausche erstes 0 und letzets 1 und interpretiere die Anzahl der 1'en auf dem Band als das Ergebniss.

Seien z.B.: $n := 4, m := 3$. Dann gilt:

$$
\begin{aligned}
4 + 3 &\equiv 1111\boxed{0}11\boxed{1}0 && \text{(Kodieren der Eingabe)} \\
&\Rightarrow 1111\boxed{1}11\boxed{0}0 && \text{(Vertausche erstes 0 und letztes 1)} \\
&\equiv 7 && \text{(Dekodieren der Ausgabe)}
\end{aligned}
$$

Die TM - gegeben durch den folgenden Uebergangsgraph und Uebergangstabelle (Siehe Figure 4 und Figure 3) - realisiert diese Berechnung:

Begruendung/Erklaerung der Vorgehensweise dieser TM:

- q_0: Das ist der initialer Zustand. Lese 1'en und bewege den Kopf rechts bis erstes 0 gefunden wird. Ersetze diesen 0 durch einen 1, bewege den Kopf rechts und gehe zum Zustand q_1 ueber
- q_1: Erstes 0 wurde gefunden und durch 1 ersetzt. Lese 1'en und bewege den Kopf rechts bis der zweite 0 gefunden wird. Das ist das Ende der Eingabe. Bewege den Kopf ein mal links zurueck und gehe zum Zustand q_2 ueber.
- q_2: Der Kopf steht auf den letzten 1 der Eingabe. Ersetze diesen 1 durch einen 0 und bewege den Kopf ein mal links. Da das Ziel erreicht wurde (vertauschen der ersten 0 und letzten 1) gehe zum Zustand q_A ueber.
- q_A: Das ist der akzeptierende Zustand. Falls die Eingabe gueltig ist haelt der TM im Zustand q_A mit dem richtigen Ergebniss auf dem Band.

Folgendes Programm realisiert diese TM auf dem TM simulator, wobei 0's durch blanks ersetz wurden, und letzte Bewegung 'hold' statt 'links' ist. (Das Programm ist auch als txt datei im Zip enthalten)

Zustand	Input	Operation	Next State	Comment
*q0	1	1, >	q0	Initialer Zustand
	0	1, >	q1	
q1	1	1, >	q1	
	0	0, <	q2	
q2	1	0, <	qA	
qA				Endzustand

Figure 3: Uebergangstabelle

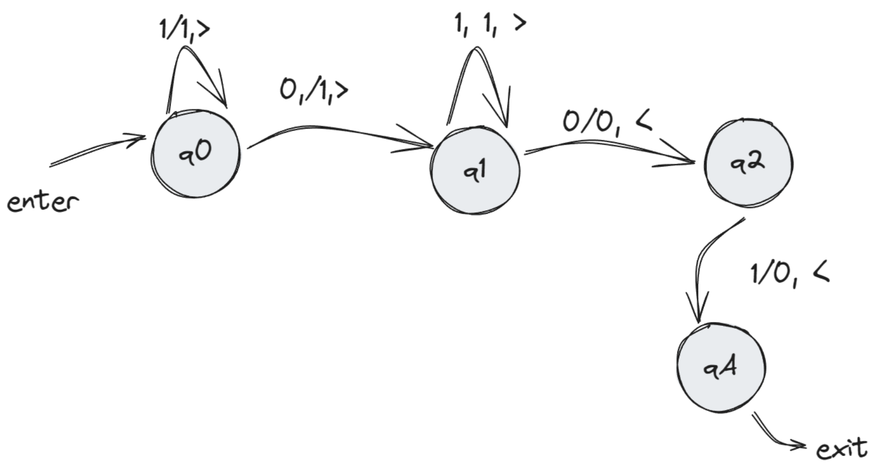


Figure 4: Uebergangsgraph

```
//TM machine to add two numbers n and m
//Input: string of n 1's and a string of m 1's seperated my a blank
//Output: string of m + n 1's
//Example: Input: "1111 111"
//         Output: "1111111"

name: add two numbers
init: q0
accept: qA

q0,1
q0,1,>

q0,_
q1,1,>

q1,1
q1,1,>

q1,_
q2,_,<

q2,1
qA,_,-
```

Alternativ: link zur realisierung der TM auf der Webseite.

Aufgabe 3

Eine sprache fuer lineare Gleichungssysteme kann z.B. durch folgende EBNF-syntaxbeschreibung definiert werden:

$$
\begin{aligned}
\langle Gleichungssystem \rangle &::= \langle Gleichung \rangle \{ \underline{\backslash n} \langle Gleichung \rangle \} \\
\langle Gleichung \rangle &::= [\langle Zahl \rangle] \underline{x} \langle Index \rangle \{ \langle Vorzeichen \rangle [\langle Zahl \rangle] \underline{x} \langle Index \rangle \} \underline{=} \langle Zahl \rangle \\
\langle Vorzeichen \rangle &::= \underline{-} | \underline{+} \\
\langle Zahl \rangle &::= \langle Ersteziffer \rangle \{ \langle Ziffer \rangle \} \\
\langle Ersteziffer \rangle &::= \underline{1}|\underline{2}|\underline{3}|\underline{4}|\underline{5}|\underline{6}|\underline{7}|\underline{8}|\underline{9}| \\
\langle Ziffer \rangle &::= \underline{0} | \langle Ersteziffer \rangle \\
\langle Index \rangle &::= \underline{{}_0} | \langle Subzahl \rangle \\
\langle Subzahl \rangle &::= \langle Erstesubziffer \rangle \{ \langle Subziffer \rangle \} \\
\langle Erstesubziffer \rangle &::= \underline{{}_1}|\underline{{}_2}|\underline{{}_3}|\underline{{}_4}|\underline{{}_5}|\underline{{}_6}|\underline{{}_7}|\underline{{}_8}|\underline{{}_9}| \\
\langle Subziffer \rangle &::= \underline{{}_0} | \langle Erstesubziffer \rangle
\end{aligned}
$$

Die Anforderung “Die Anzahl der Variablen ist gleich der Anzahl der Gleichungen” ist eine Beschreibung die von dem Kontext des Erzeugten Wortes abhaengt - gueltige Gleichungssysteme duerfen beliebige Anzahl an Variablen haben. Da mit EBNF nur kontextfreie Sprachen definiert werden koennen ist diese Anforderung nicht umsetzbar.

Zettel 03

Aufgabe 2

2.1

Folgendes Program loesst das problem (auch im zip als `potenz.cc` enthalten)

```
#include "fcpp.hh"

int quadrat (int x)
{
  return x*x;
}

int potenz(int x, int n)
{
    return cond(n == 0,
            1,
            cond(n % 2 == 0,
                quadrat(potenz(x, n / 2)),
                x * potenz(x, n - 1)));
}


int main(int argc, char** argv)
{
    return print(potenz(
      readarg_int(argc, argv, 1),
      readarg_int(argc, argv, 2)));
}
```

Argumente muessen in der Konsole eingegeben werden, z.B.:

```
$ ./potenz 4 3
81
```

Aufgabe 3

3.1

Folgendes Program realisiert die rekursive Berechnung der Binomialkoeffizienten (auch im Zip als `binomial.cc` enthalten):

```
#include "fcpp.hh"

int binomial(int n, int k)
{
    return cond(k == 0 || k == n,
            1,
            binomial(n - 1, k - 1) + binomial(n - 1, k));
}

int main(int argc, char** argv)
{
    return print(binomial(
        readarg_int(argc, argv, 1),
        readarg_int(argc, argv, 2)));
}
```

Wir haben das Program auf verschieden Werte n und k getestet (zusammen mit der `time` Funktion fuer Messung der Laufzeit) und die Ergebnisse in der folgenden Tabelle zusammengefasst:

> **Note**
>
> Specs des Systems auf der wir getestet haben:
>
> - **PC**: ThinkCentre M700
> - **Processors**: 4 × Intel® Core™ i5-6400 CPU @ 2.70GHz
> - **Memory**: 15,5 GiB of RAM

Befehl	Ergebniss	Laufzeit (real)

`time ./binomial 1 0`	1	`real 0m0,004s`
`time ./binomial 1 1`	1	`real 0m0,002s`
`time ./binomial 3 2`	3	`real 0m0,002s`
`time ./binomial 10 4`	210	`real 0m0,004s`
`time ./binomial 20 13`	77520	`real 0m0,007s`
`time ./binomial 32 15`	565722720	`real 0m3,519s`
`time ./binomial 36 13`	-1984177696	`real 0m13,791s`

Wie aus der Tabelle zu sehen ist, ist die Laufzaut $> 10s$ fuer die Berechnung `time ./binomial 36 13` jedoch mit dem falschen Ergebniss -1984177696 statt das richtige $\binom{36}{13} = 2310789600$. Wir erklaeren dieses Phaenomen in der folgenden Teilaufgabe.

3.2

Fuer $n = 34$, $k = 18$ liefert das Program

```
$ ./binomial 34 18
-2091005866
```

im Gegensatz zu dem erwarteten mathematischen Ergebniss $\binom{34}{18} = 2203961430$.

Dieser 'Fehler' liegt an der 32 bit 2er Komplement Darstellung des Datentyps `int` auf dem Computer. Darunter koennen eine endliche Anzahl von `int` Zahlen dargestellt werden, die im Bereich $[-2^{31}, 2^{31}-1] = [-2147483648, 2147483648]$ liegen. Das mathematische Ergebnis liegt also ausserhalb des darstellbaren Bereiches mit $2203961430 > 2147483648$.

Da, unter der 2er Komplement Darstellung der MSB (Most Significant Bit) den Bereich der Negativen Zahlen representiert kann die Addition zweier groessen Zahlen, deren Ergebniss ausserhalb des darstellbaren Bereiches liegt wieder bei dem negativen Bereich landen, aehnlich wie Modulorechnung. Das wird als **overflow** bezeichnet.

3.3

Sei $A_{n,n} = \alpha = A_{n,0}$ und $\beta :=$ Die konstanten Kosten der Addition. Dann gilt:

$$\begin{aligned} A_{n,k} &= A_{n-1,k-1} + A_{n-1,k} + \beta && \text{(Rekursive Beziehung des Rechenaufwands)} \\ A_{n,0} &= \alpha = A_{n,n} \end{aligned}$$

Da die rekursive Beziehungs des Rechenaufwands eine aehnliche Beziehung wie die Binomialkoeffizienten erfuellen koennen diese in einem paskalschen Dreieck wie folgt eingetragen werden (Siehe Figure 5)

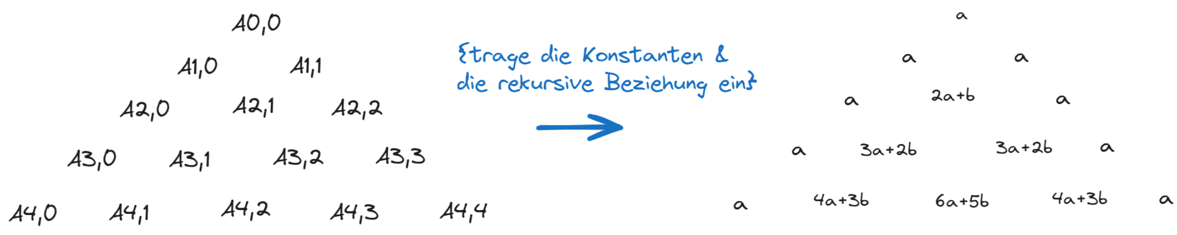


Figure 5: Kosten-dreieck

Betrachten wir nur die Koeffizienten von α und β seperat so erhalten wir folgende paskalschen Dreiecke (Siehe Figure 6)

Figure 6: Konstanten-dreiecke

Von diesen Figuren ist es leicht zu sehen, dass $\alpha_{n,k} = B_{n,k}$ und $\beta_{n,k} = B_{n,k} - 1$, wobei $\alpha_{n,k}, \beta_{n,k}$ die Koeffizienten von α bzw. β sind bzgl der Rechenaufwands $A_{n,k}$.

Somit erhalten wir:

$$A_{n,k} = B_{n,k}\alpha + (B_{n,k} - 1)\beta$$

Formaler Beweis:

Fuehre die Variablentransformation $\tilde{A}_{n,k} := \frac{A_{n,k}+\beta}{\alpha+\beta}$. Dann erhalten wir die folgende rekursive Gleichung:

$$\tilde{A}_{n,k} = \tilde{A}_{n-1,k-1} + \tilde{A}_{n-1,k}$$
$$\tilde{A}_{n,0} = 1 = \tilde{A}_{n,n}$$

Das ist genau die Definition des Binomialkoeffizientes $B_{n,k}$. Somit gilt:

$$\begin{aligned}\tilde{A}_{n,k} &= B_{n,k} \\ \Rightarrow A_{n,k} &= (\alpha+\beta)\tilde{A}_{n,k} - \beta \\ &= (\alpha+\beta)B_{n,k} - \beta \quad \blacksquare\end{aligned}$$

3.4

Folgend geben wir die iterative Implementierung der Binomialkoeffizienten anhand einer tail-rekursiven Implemenierung der Fakultaet-funktion an (auch im Zip als `binomial_fast.cc` erhalten):

```
#include "fcpp.hh"

//iterative Implementierung der Fakultaetsfunktion durch Tail-recursion
// mit fakit und fak
int fakit(int res, int n)
{
    return cond(
        n > 1,
        fakit(res * n, n - 1),
        res
    );
}

int fak(int n)
{
    return fakit(1, n);
}

int binomial_fast(int n, int k)
{
    return fak(n) / (fak(k) * fak(n - k));
}

int main(int argc, char** argv)
{
    return print(binomial_fast(
        readarg_int(argc, argv, 1),
```

```
        readarg_int(argc, argv, 2)));
}
```

Da, die Implementierung von Fakultaet `fak` $\mathcal{O}(n)$ ist , `binomial_fast` nur drei mal `fak` aufruft und nur zwei weitere Basisoperationen verwendet - eine Multiplikation und eine Division - hat diese Implementierung eine Laufzeitkomplexitaet von $\mathcal{O}(n)$.

Wir haben `binomial` und `binomial_fast` auf verschiedene Eingaben hinsichtlich Ausgaben und Geschwindigkeit getestet und die Ergebnisse in der folgenden Tabelle zusammengefasst (Table 2):

Table 2: `binomial` vs `binomial_fast`

Befehl	**Ergebniss**	**Laufzeit (real)**
`time ./binomial 10 4`	`210`	`real 0m0,004s`
`time ./binomial_fast 10 4`	`210`	`real 0m0,004s`
`time ./binomial 11 6`	`462`	`real 0m0,005s real`
`time ./binomial_fast 11 6`	`462`	`0m0,005s`
`time ./binomial 13 10`	`286`	`real 0m0,005s`
`time ./binomial_fast 13 10`	`88`	`real 0m0,004s`
`time ./binomial 20 13`	`77520`	`real 0m0,006s`
`time ./binomial_fast 20 13`	`-2`	`real 0m0,002s`
`time ./binomial 27 15`	`17383860`	`real 0m0,123s`
`time ./binomial_fast 27 15`	`0`	`real 0m0,004s`
`time ./binomial 32 15`	`565722720`	`real 0m3,519s`
`time ./binomial_fast 32 15`	`-3`	`real 0m0,004s`
`time ./binomial 34 19`	`1855967520`	`real 0m11,689s`
`time ./binomial_fast 34 19`	`0`	`real 0m0,002`
`time ./binomial 36 13`	`-1984177696`	`real 0m15,388s`
`time ./binomial_fast 36 13`	`0`	`real 0m0,004s`
`time ./binomial 39 23`	`-943444674`	`real 3m47,934s`
`time ./binomial_fast 39 23`	`core dumped`	`real 0m0,086s`

Da `binomial_fast` eine lineare Laufzeitkomplezitaet hat bleibt die Laufzeit c. .004s fuer alle Eingaben.

Im Gegensatz waechst die Laufzeit von `binomial` proportional zu dem mathematischen Wert $B_{n,k}$ fuer Eingaben n und k. Somit ist die Laufzeit sehr hoch fuer grosse Werte von $B_{n,k}$.

Wie aus der Tabelle zu sehen ist, gibt es bereits ab $n = 27, k = 15$ einen Unterschied und fuer hoehere Werte wie $n = 34, k = 19$ oder $n = 36, k = 13$ unterscheiden sich die Laufzeiten um mehr als 10s. Fuer $n = 39, k = 23$ ist die Laufzeit von `binomial` sogar fast 4 minuten, wobei `binomial_fast` immer noch bei ~0.004s bleibt.

3.5

Fuer diese Teilaufgabe verwenden wir wieder die obige Tabelle Table 2. Fuer die ersten zwei Zeilen, also fuer $n = 10, k = 4$ und $n = 11, k = 6$ liefern beide Programme die richtige mathematische Ergebnisse $\binom{10}{4} = 210$, bzw. $\binom{11}{6} = 462$. Jedoch liefert `binomial_fast` bereits ab der dritten Zeile, also fuer $n = 13, k = 10$ ein falsches Ergebnis, wobei `binomial` bis der 7en Zeile, d.h fuer $n = 34, k = 19$ richtige Ergebnisse liefert.

Wie in der Teilaufgabe 3.3 erklaert wurde, gibt es das sogenannte Phaenomen "**overflow**" fuer den Datentyp `int`. Die Fakultaetsfunktion waechst sehr schnell und hat bereits fuer die Eingabe 13 den Wert $13! = 6227020800 > 2147483647 = 2^{31} - 1$. Somit fuehrt bereits `fak(13)` zu einem overflow. Da, die Implementierung von`binomial_fast` die Funktion `fak` benutzt, liefert dieses Program fuer Eingaben $n \geq 13$ ein falsches Ergebnis.

Zettel 04

Aufgabe 1

Laut link existieren folgende verschiedene "Data Models", die die Groesse der Grunddatengypen festlegen:

1.1

- **LP32**(32 bit Systeme):
  - `int`: 16-bit
  - `long`: 32-bit
  - `pointer`: 32-bit
- **ILP32**(32 bit Systeme):
  - `int`: 32-bit
  - `long`: 32-bit
  - `pointer`: 32-bit
- **LLP64**(32 bit Systeme):
  - `int`: 32-bit
  - `long`: 32-bit
  - `pointer`: 64-bit
- **LP64**(32 bit Systeme):
  - `int`: 32-bit
  - `long`: 64-bit
  - `pointer`: 64-bit

Somit ist `long` mindestens 32-bit kann aber auch 64-bit sein.

1.2

Bezeichne $[\texttt{S|E|M}]_{\texttt{FP32}}$ Die IEEE754 Fliesskommadarstellung einer zahl, wobei $S :=$ sign bit, $E :=$ Exponent, $M :=$ Mantisse, und sei $[\texttt{S|E|M}]_2$ die 2-Bit Ganzzahldarstellung der selben Bitfolge.

Dann gilt:

$$
\begin{aligned}
\log_2(y) &= \log([\texttt{0|E|M}]_{\texttt{FP32}}) && \text{(fuer ein } y >= 0 \text{ im gueltigen Bereich)} \\
&= \log_2((1 + \frac{M}{2^{23}})2^{\texttt{E}-127}) && \text{(Interpretation von } [\bullet]_{\texttt{FP32}}) \\
&= \log_2(1 + \frac{M}{2^{23}}) + \texttt{E} - 127 && \text{(Rechenregeln fuer log)} \\
&\approx \frac{\texttt{M}}{2^{23}} + \mu + \texttt{E} - 127 && (\log_2(1+x) \approx x + \mu) \\
&= \frac{1}{2^{23}}(\mu + 2^{23} + E) + \mu - 127 && \text{(arithmetik)} \\
&= \frac{1}{2^{23}}[\texttt{0|E|M}]_2 + \mu - 127 && \text{(Interpretation von } [\bullet]_2) \\
&= \alpha \cdot [\texttt{0|E|M}]_2 + \beta && (\alpha, \beta \text{ bestimmte Konstanten})
\end{aligned}
$$

Diese Umformungen stellen die Bezieuhung zwischen dem Logarithmus einer zahl y und der Bitfolge ihrer IEEE754 FP Darstellung dar: Interpreteriert man die Bitfolge als eine 2-Bit Ganzahl, so besteht ein linearer Zusammenhang zwischem dem Logarithmus und die zur 2-Bit Bitfolge entsprechende Ganzzahl, sie unterscheiden sich approximativ lediglich um eine Skalierung und Verschiebung.

1.3

`y` ist ein `float`. Mit `i = * (long *) &y` weisen wir die im `y` enthaltene Bitfolge zu der Variable `i` zu und stellen den Typ von `i` als `long` fest. Somit enthaelt `i` die genaue Bitfolge von `y` aber wird als `long` interpretiert statt als `float`. Der numerische Wert von `i` ist ganz anderes als dies von `y` aber mit der selben Bitfolge.

Hingegen fuehrt `i = (long) y` eine automatische Typkonvertierung durch und weist den Wert von `y` gerundet auf einer Ganzzahl zu `i` zu. Somit tragen `i` und `y` aehnliche numerische Werte aber `i` hat eine ganz andere Bitfolge von `y`. Da, wir uns fuer die genaue Bitfolge interessieren und nicht fuer den numerischen wert wuerde das gar nicht funktioneren.

Aufgabe 2

Wir verwenden folgenden Definitionen in dem Beweis:

1. **2er Komplementdarstellung**:

$$d_n : [-2^{n-1}, 2^{n-1} - 1] \to [0, 2^{n-1}]$$

$$d_n : a \mapsto \begin{cases} a & a \geq 0 \\ 2^n - |a| & a < 0 \end{cases} \qquad \begin{matrix} \text{(Dar 1)} \\ \text{(Dar 2)} \end{matrix}$$

2. **Beschneidung**:

$$s_n(x_{m-1} \dots x_0) = \begin{cases} x_{m-1} \dots x_0 & m \leq n \\ x_{n-1} \dots x_0 & m > n \end{cases} \qquad \begin{matrix} \text{(Besch 1)} \\ \text{(Besch 2)} \end{matrix}$$

Beweis:

- $\underline{a = 0}$

$$\begin{aligned} d_n(0) &= d_n(-0) && (-0 = 0) \\ &= s_n(2^n) && \text{(Besch 2)} \\ &= s_n(2^n - 0) && \text{(Arithmetik)} \\ &= s_n(2^n - d_n(0)) && \text{(Dar 1)} \end{aligned}$$

- $\underline{0 < a < 2^{n-1}:}$

$$\begin{aligned} d_n(-a) &= 2^n - |a| && (-a < 0, \text{ Dar 2}) \\ &= 2^n - a && (a > 0 \Rightarrow |a| = a) \\ &= s_n(2^n - a) && (2^n - a < 2^n, \text{ Besch 1}) \\ &= s_n(2^n - d_n(a)) && (a > 0, \text{ Dar 1}) \end{aligned}$$

- $\underline{-2^{n-1} < a < 0:}$

$$
\begin{aligned}
d_n(-a) &= -a && (-a > 0, \text{ Dar } 1)\\
&= s_n(-a) && (-a < 2^n, \text{ Besch } 1)\\
&= s_n(2^n - (2^n + a)) && (\text{Arithmetik})\\
&= s_n(2^n - (2^n - (-a))) && (\text{Arithmetik})\\
&= s_n(2^n - (2^n - |a|)) && (a < 0 \Rightarrow |a| = -a)\\
&= s_n(2^n - d_n(a)) && (a < 0, \text{ Dar } 2)
\end{aligned}
$$

Aufgabe 3

3.1

i)

$$
\begin{aligned}
log_2(2n) &= \log_2(2) + \log_2(n)\\
&= 1 + \log_2(n)\\
&= \begin{cases} 1 + 4s = 5s, & \log_2(n) = 4s\\ 1 + 10s = 11s, & \log_2(n) = 10s\\ 1 + 100s = 101s, & \log_2(n) = 100s \end{cases}
\end{aligned}
$$

ii)

$$
2n = \begin{cases} 2 \cdot 4s = 8s, & n = 4s\\ 2 \cdot 10s = 20s, & n = 10s\\ 2 \cdot 100s = 200s, & n = 100s \end{cases}
$$

iii)

$$
2n\log_2(2n) = \begin{cases} 2 \cdot 4s(1+4) = 40s, & n = 4s\\ 2 \cdot 10s(1+10) = 220s, & n = 10s\\ 2 \cdot 100s(1+100) = 20200s, & n = 100s \end{cases}
$$

iv)

$$
(2n)^3 = 8n^3 \begin{cases} 8 \cdot 4s = 40s, & n = 4s\\ 8 \cot 10s = 80s, & n = 10s\\ 8 \cdot 100s = 800s, & n = 100s \end{cases}
$$

v)

$$2^(2n) = (2^n)^2 = \begin{cases} 4^2 s = 16s, & n = 4s \\ 10^2 s = 100s, & n = 10s \\ 100^2 s \approx 2.8std, & n = 100s \end{cases}$$

3.2

Wir fuehren die Notation $f \preceq g :\iff f \in \mathcal{O}(g)$ ein. Dann gilt:

$$\begin{aligned} 1 &\preceq \log(\log(n)) \\ &\preceq \log(n) \\ &\preceq n^{\epsilon} \\ &\preceq n^{\log(n)} \\ &\preceq c^n \\ &\preceq n^n \\ &\preceq c^{c^n} \end{aligned}$$

Aufgabe 4

4.1

Die Funktion `int kantenindex(int i, int j)` (Quellcode auch im zip im `niklaus.cc` enthalten, beachte die fuer die Vereinfachung des Syntax definierten 'Helferfunktionen' `int abs(int x)`, `int min(int i, int j)` und `int max(int i, int j)`):

```
int abs(int x){return cond(x < 0, -x, x);}

//returns minimum of i and j
int min(int i, int j){return cond(i > j, j, i);}

// returns maximum of i and j
int max(int i, int j){return cond(i > j, i, j);}

int kantenindex(int i, int j)
{
```

```
    return cond(
        i == j || abs(i - j) == 4 || (min(i, j) == 0 && max(i, j) == 3),
 ↪  //Diese Kanten existieren nicht
        -1, //error code is -1
        cond(
            min(i, j) == 2 && max(i, j) == 3,
            0,
            i + j));
}
```

4.2

a)

```
//0000 0100 & zzzz zzzz == 0000 0z00
//somit i'te kante besucht <=> (2^i & z) != 0
//wobei 2^i == 1 << i
bool kante_besucht(int kante, unsigned char zustand)
{
    return ((1 << kante) & zustand) != 0;
}
```

b)

```
//0000 0100 | zzzz z0zz == zzz z1zz
//somit besuche i'te kante: z := z | 2^i
//wobei 2^i == 1 << i
unsigned char besuche_kante(int kante, unsigned char zustand)
{
    return zustand | (1 << kante);
}
```

Zettel 05

Aufgabe 1

Siehe Figure 7

Aufgabe 2

2.1

folgendes Code realisiert die `float` version der Determinante-funktion:

```
#include "fcpp.hh"

float determinante(
    float a,
    float b,
    float c,
    float d
)
{
    return a * d - b * c;
}

int main()
{
    return print(determinante(
        100, 0.01,
        -0.01, 100
    ));
}
```

Ergebniss: 10000

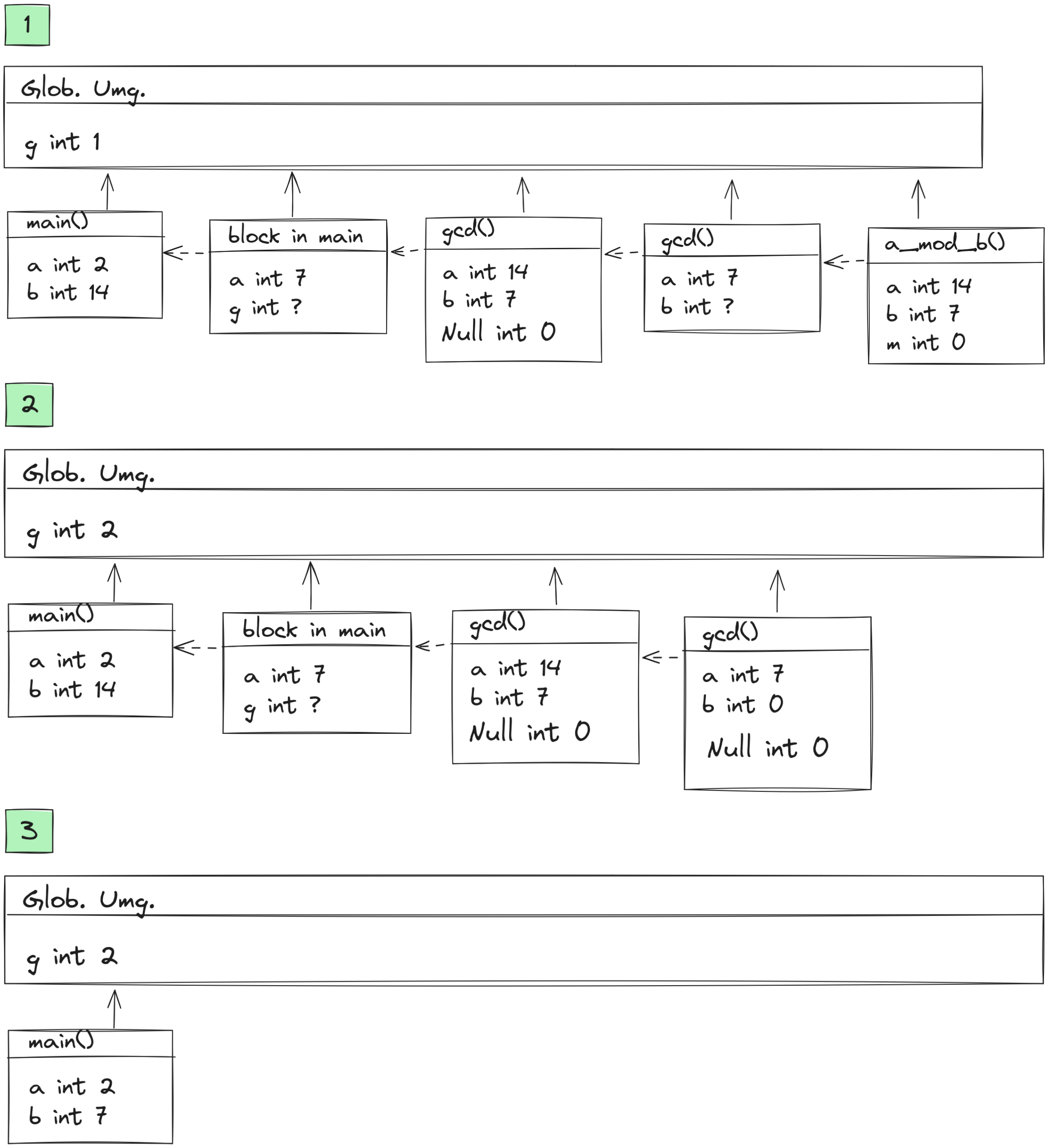


Figure 7: Umgebungen bevor $\boxed{1}$, $\boxed{2}$, $\boxed{3}$ ausgefuehrt werden.

Erklaerung: Das exakte Ergebniss der Berechnung $10000.0001 = 1.00000001 \times 10^4$ kann nicht mit 32 bit `float` dargestellt werden. Eine 32-bit `float` ist eine Bitfolge der Form `S|E|M`, wobei S der "sign-bit", 8 E's die 'Exponenten-bits' und 23 M's die 'Mantisse-bits' sind. Diese bitfolge wird interpretiert als:

$$[S|E|M]_{FP32} := (-1)^S \times 1.M \times 2^{E-127}$$

Die exakte Darstellung von 1000.0001 in 2er Basis lautet jedoch $1.00111000\ldots 11010001101101110001 \times 2^{13}$. Hier gibt mehr als 23 Stellen nach der Komma, die der Mantisse entsprechen. Jedoch erlaubt `float` 23 Stellen fuer die Mantisse. Deshalb kann diese Bitfolge nicht genau representiert werden und wird stattdessen moeglichst praezis gerundet.

Die Rundung erfuellt die Eigenschaft

$$rd(x) = x(1 + r), r < eps$$

Wobei r der relative Rundungsfehler, eps die Maschinen-genauigkeit ist. Fuer `float` lautet dies $2^{-23} \approx 10^{-7}$.

Die Rundung $rd(1000.0001) = 1000$ erfuellt tatsaechlich diese Eigenschaft, da $r = \frac{.0001}{10000} = 10^{-8}$

Die `double` Version des Programms

```
#include "fcpp.hh"

double determinante(
    double a,
    double b,
    double c,
    double d
)
{
    return a * d - b * c;
}

int main()
{
    return print(determinante(
        100, 0.01,
        -0.01, 100
    ));
```

```
}
```

Liefert das mathematisch exakte Ergebniss 10000.0001. Da, der Datentyp `double` die 64-bit Fliesskommazahlen realisiert, hat es die Maschinengenauigkeit $eps = 2^{-52}$. Somit koennen mehr siginifikanten Stellen representiert werden als mit 32-bit FP Zahlen. Die Zahl 10000.0001 hat immer noch keine exakte Darstellung aber da, *eps* viel kleiner ist liefert die Rundung die exakte Darstellung.

Aufgabe 3

3.1

Aequivalente `while`-Version lautet:

```
int fib(int n)
{
    int a = 0;
    int b = 1;
    int i = 0;
    //invariante: a == fib(i) && b == fib(i + 1) && i <= n
    while (i < n){
        int t = a + b;
        a = b;
        b = t
        i = i + 1;
    }//i == n => a == fib(n)
    return a;
}
```

- **Schleifenvektor**: $v = (a, b, t, i)$
- **Schleifenbedingung**: $B(v) :\equiv i < n$
- **Schleifentransformator**: $H(a, b, t, i) = (b, a + b, a + b, i + 1)$

3.2

- **Vorbedingung**: $P(n) :\equiv 0 \leq n$
- **Behauptung**: $Inv(v) :\equiv (a = fib(i)) \wedge (b = fib(i + 1)) \wedge (i \leq n)$
- **Nachbedingung**: $Q(v, n) :\equiv a = fib(n)$

Beweis:

Bezeichne $v_j := H^j(v)$

1) (**IB**): Vor der ersten Iteration gilt: $v_0 = (a_0, b_0, t_0, i_0) = (0, 1, ?, 0)$ und somit:

$$\begin{aligned} a_0 &= 0 \\ &= fib(0) && \text{(Def von fib)} \\ &= fib(i_0) \end{aligned}$$

$$\begin{aligned} b_0 &= 1 \\ &= fib(1) && \text{(Def von fib)} \\ &= fib(i_0 + 1) \end{aligned}$$

$$i_0 = 0 \leq n \qquad \text{(Vorbedingung } P(n)\text{)}$$

Somit gilt $Inv(v_0)$

2) (**IS**): Gelte die Invariante vor einer k-ten Iteration, d.h

$$Inv(v_k) :\equiv (a_k = fib(i_k)) \wedge (b_k = fib(i_k + 1)) \wedge (i_k \leq n)$$

z.z.: $Inv(v_k) \wedge B(v_k) \Rightarrow Inv(v_{k+1})$

Bew: Es gilt:

$$\begin{aligned} Inv(v_k) \wedge B(v_k) &\Leftrightarrow a_k = fib(i_k) \wedge b_k = fib(i_k + 1) \wedge i_k \leq n \wedge i_k < n \\ &\qquad \text{(Def von } Inv(v) \text{ und } B(v)\text{)} \\ &\Rightarrow a_k = fib(i_k) \wedge b_k = fib(i_k + 1) \wedge i_k < n && \text{(Arithmetik)} \\ &\Rightarrow a_k + b_k = fib(i_k) + fib(i_k + 1) \wedge b_k = fib(i_k + 1) \wedge i_k + 1 \leq n \\ &\qquad \text{(Arithmetik)} \\ &\Leftrightarrow a_k + b_k = fib(i_k + 2) \wedge b_k = fib(i_k + 1) \wedge i_k + 1 \leq n && \text{(Def von } fib\text{)} \\ &\Leftrightarrow a_k + b_k = fib((i_k + 1) + 1) \wedge b_k = fib(i_k + 1) \wedge i_k + 1 \leq n \\ &\qquad \text{(Arithmetik)} \\ &\Leftrightarrow b_{k+1} = fib(i_{k+1} + 1) \wedge a_{k+1} = fib(i_{k+1}) \wedge i_{k+1} \leq n \\ &\qquad \text{(Def von } H \text{ und } v_{k+1} = H(v_k)\text{)} \\ &\Leftrightarrow Inv(v_{k+1}) \quad \blacksquare && \text{(Def von } Inv(v)\text{)} \end{aligned}$$

3) Nach dem Verlassen der Schleifen bei einem allgemeinen Schleifenvektor v gilt somit:

$$\begin{aligned}
Inv(v) \land \neg B(v) &\Leftrightarrow (a = fib(i) \land b = fib(i+1) \land i \leq n) \land \neg(i < n) \\
&\hspace{15em} \text{(Def von } Inv \text{ und } B\text{)} \\
&\Leftrightarrow (a = fib(i) \land b = fib(i+1)) \land (i \leq n \land i \geq n) && \text{(DeMorgan)} \\
&\Leftrightarrow (a = fib(i) \land b = fib(i+1)) \land i = n && \text{(Arithmetik)} \\
&\Rightarrow a = fib(n) && \text{(Aussagenlogik)} \\
&\Leftrightarrow Q(v,n) \quad \blacksquare && \text{(Def der Nachbedingung } Q\text{)}
\end{aligned}$$

Aufgabe 4

Wir betrachten 4 Faelle:

Fall $a, b \geq 0$:

$$\begin{aligned}
s_n(d_n(a) \cdot d_n(b)) &= s_n(a \cdot b) && \text{(Def } d_n\text{)} \\
&= a \cdot b && (ab \leq 2^{n-1} < 2^n, \text{ Def } s_n) \\
&= d_n(ab) && (ab \geq 0)
\end{aligned}$$

Fall $a, b < 0$:

$$\begin{aligned}
s_n(d_n(a) \cdot d_n(b)) &= s_n((2^n - |a|)(2^n - |b|)) && \text{(Def } d_n\text{)} \\
&= s_n((2^n + a)(2^n + b)) && (a, b < 0) \\
&= s_n(4^n + 2^n(a+b)ab) \\
&= s_n(2^n(2^n + a + b) + ab) \\
&= a \cdot b && (0 \leq 2^n + a + b < 2^n, \text{ Def } s_n) \\
&= d_n(ab) && (ab > 0, \text{ Def } d_n)
\end{aligned}$$

Fall o.B.d.A $a < 0, b > 0$:

$$
\begin{aligned}
s_n(d_n(a)d_n(b)) &= s_n((2^n - |a|)b) && (a < 0\text{, Def } d_n)\\
&= s_n((2^n + a)b) && (a < 0)\\
&= s_n(2^n b + ab)\\
&= s_n(2^n b - |ab|) && (ab < 0)\\
&= s_n(2^n b - 2^n + (2^n - |ab|))\\
&= s_n(2^n(b-1) + (2^n - |ab|))\\
&= 2^n - |ab| && (b \geq 1\text{, Def } s_n)\\
&= d_n(ab) && (\text{Def } d_n)
\end{aligned}
$$

Fall o.B.d.A $b = 0$:

$$
\begin{aligned}
s_n(d_n(a)d_n(0)) &= s_n(d_n(a) \cdot 0)\\
&= s_n(0)\\
&= 0\\
&= d_n(0)\\
&= d_n(a \cdot 0)
\end{aligned}
$$