

IPI WS23/24 Solutions

Igor Dimitrov

2023-10-30

Table of contents

Preface	3
Zettel 01	4
Aufgabe 3	4
3.1	4
3.2	4
3.3	5
Zettel 02	8
Aufgabe 2	8
Aufgabe 3	10
Zettel 03	12
Aufgabe 2	12
2.1	12
Aufgabe 3	13
3.1	13
3.2	14
3.3	14
3.4	16
3.5	17

Preface

Solutions of the assignment sheets for the lecture “[IPI WS23/24](#)” at Uni Heidelberg.

Zettel 01

Aufgabe 3

3.1

- In der VL beschriebene 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 erklart 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 entspricht der Situation, dass die gelesene Anzahl von 0's **ungerade** ist.

Oder kuerzer:

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

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 $q_i \rightarrow q_{i \oplus 1}$ verursacht, wobei $i \oplus 1$ Addition mod 2 ist. Hingegen verursacht das Ablesen einer 1 keinen Zustanduebergang: $q_i \rightarrow q_i$. 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. Sonst 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 **Uebergangsgraph**

Zustand	Input	Operation	Next State	Comment
*q0	0	0, >	q1	Initialer Zustand
	1	1, >	q0	
	$\overline{0}$	$\overline{0}$, -	qAccept	
q1	0	0, >	q0	
	1	1, >	q1	
qAccept				Endzustand

Figure 1: Uebergangstabelle

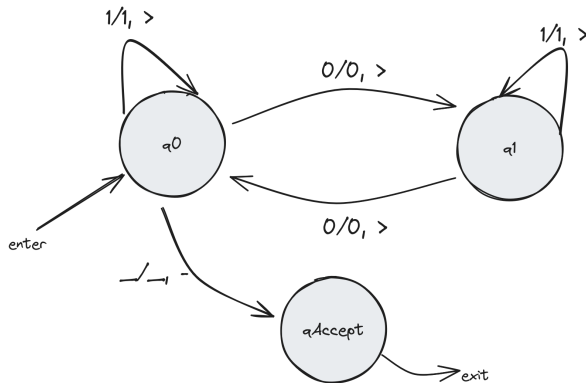


Figure 2: Uebergangsgraph

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, ersetzen 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:

double up a string of 1's


Steps: 4 State: q8 Accepted (show output)



Input: 1 Load [play] [pause] [stop] [rewind] Speed: [slider]

double up a string of 1's


Steps: 11 State: q8 Accepted (show output)



Input: 11 Load [play] [pause] [stop] [rewind] Speed: [slider]

double up a string of 1's

Steps: 56 State: q8 Accepted (show output)



Input: 11111 Load [play] [pause] [stop] [rewind] Speed: [slider]

Zettel 02

Aufgabe 2

Idee: Vertausche erstes 0 und letztes 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:

Begrundung/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 haelte 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 ersetzt 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

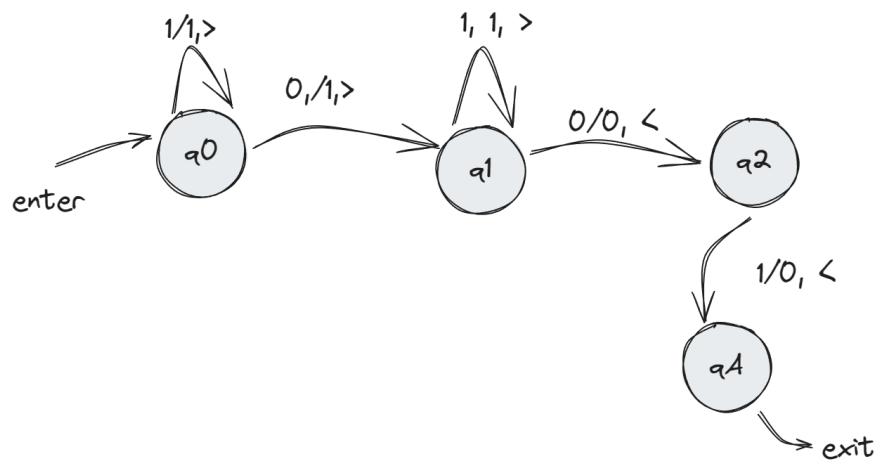


Figure 4: Uebergangsgraph

```

//TM machine to add two numbers n and m
//Input: string of n 1's and a string of m 1's seperated by 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 für lineare Gleichungssysteme kann z.B. durch folgende EBNF-Syntaxbeschreibung definiert werden:

$$\begin{aligned}
\langle \text{Gleichungssystem} \rangle &::= \langle \text{Gleichung} \rangle \{ \backslash n \langle \text{Gleichung} \rangle \} \\
\langle \text{Gleichung} \rangle &::= [\langle \text{Zahl} \rangle] \underline{x} \langle \text{Index} \rangle \{ \langle \text{Vorzeichen} \rangle [\langle \text{Zahl} \rangle] \underline{x} \langle \text{Index} \rangle \} \equiv \langle \text{Zahl} \rangle \\
\langle \text{Vorzeichen} \rangle &::= \pm | \pm \\
\langle \text{Zahl} \rangle &::= \langle \text{Ersteziffer} \rangle \{ \langle \text{Ziffer} \rangle \} \\
\langle \text{Ersteziffer} \rangle &::= \underline{1} | \underline{2} | \underline{3} | \underline{4} | \underline{5} | \underline{6} | \underline{7} | \underline{8} | \underline{9} | \\
\langle \text{Ziffer} \rangle &::= \underline{0} | \langle \text{Ersteziffer} \rangle \\
\langle \text{Index} \rangle &::= \underline{0} | \langle \text{Subzahl} \rangle \\
\langle \text{Subzahl} \rangle &::= \langle \text{Erstesubziffer} \rangle \{ \langle \text{Subziffer} \rangle \} \\
\langle \text{Erstesubziffer} \rangle &::= \underline{1} | \underline{2} | \underline{3} | \underline{4} | \underline{5} | \underline{6} | \underline{7} | \underline{8} | \underline{9} | \\
\langle \text{Subziffer} \rangle &::= \underline{0} | \langle \text{Erstesubziffer} \rangle
\end{aligned}$$

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 verschiedenen 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:** $4 \times$ 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 Laufzeit $> 10s$ fuer die Berechnung `time ./binomial 36 13` jedoch mit dem falschen Ergebniss -1984177696 statt das richtige $\binom{36}{13} = 2310789600$. Wir erklaren 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 Ergebnis ausserhalb des darstellbaren Bereiches liegt wieder bei dem negativen Bereich landen, ahnlich 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 Beziehung des Rechenaufwands eine ahnliche Beziehung wie die Binomialkoeffizienten erfuellen koennen diese in einem paskalschen Dreieck wie folgt eingetragen werden (Siehe Figure 5)

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

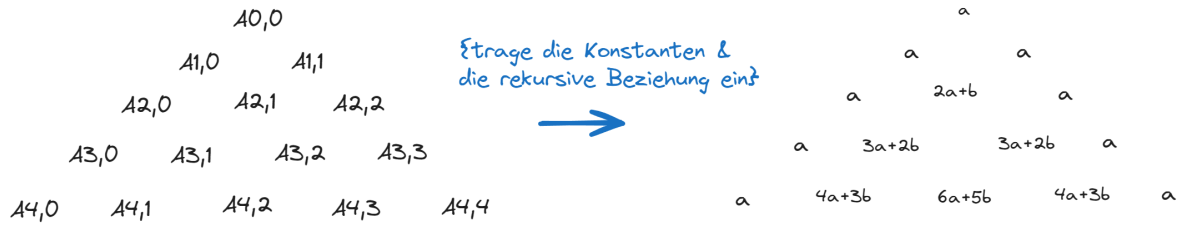


Figure 5: Kosten-dreieck



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. des Rechenaufwands $A_{n,k}$.

Somit erhalten wir:

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

Formaler Beweis:

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

$$\begin{aligned} \tilde{A}_{n,k} &= \tilde{A}_{n-1,k-1} + \tilde{A}_{n-1,k} \\ \tilde{A}_{n,0} &= 1 = \tilde{A}_{n,n} \end{aligned}$$

Das ist genau die Definition des Binomialkoeffizienten $B_{n,k}$. 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 Implementierung der Fakultätsfunktion an (auch im Zip als `binomial_fast.cc` erhalten):

```
#include "fcpp.hh"

//iterative Implementierung der Fakultätsfunktion 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 Fakultät `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 Laufzeitkomplexität 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 Laufzeitkomplexitaet 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 $\sim 0.004s$ 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 erklart 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.