Universität des Saarlandes
FR 6.2 Informatik
Prof. Dr. Hans-Peter Lenhof
Dipl. Inform. Andreas Hildebrandt

Programmierung II, SS 2003
Musterlösung zur 7. Übung

Aufgabe 1: Rekurrenz, Substitutionsmethode (10 Punkte)
Der erste Aufgabenteil war leider ein wenig schwerer als eigentlich vorgesehen. Daher wird jeder, der Aufgabe 1 überhaupt bearbeitet hat, die für den ersten Teil vorgesehenen Punkte erhalten.

- $T(n) = T(\lfloor \log_2(\sqrt{n}) \rfloor) + 12$
  Sehen wir uns zunächst mal nur die einfachere Rekurrenz $T(n) = T(\log_2(n)) + 12$ an. Iteratives selbsteinsetzen bringt uns auf folgende Idee:

  $$T(n) = T(\log_2(\log_2(\log_2(n)))) + 12 + 12 + 12$$

  usw. D.h. wir kommen offenbar auf Kosten von $12 \log_2^*(n)$[1] und damit gilt $T(n) \in \Theta(log_2^*(n))$. Für unsere eigentliche Rekursion verwenden wir jetzt noch $\log_2(\sqrt{n}) = \frac{1}{2}\log_2(n)$ und sehen uns wieder ein, was iteratives Selbsteinsetzen ergibt:

  $$T(n) = T(\frac{1}{2}\log_2(\frac{1}{2}\log_2(n))) + 2 * 12$$

  Definieren wir schliesslich analog zu $\log_2^*$ die Funktion

  $$f^i(n) := \begin{cases} n & \text{falls } i = 0 \\ \frac{1}{2}\log_2(f^{(i-1)}(n)) & \text{falls } i > 0 \text{ und } f^{i-1} > 0 \\ \text{nicht definiert} & \text{sonst} \end{cases}$$

  und die Funktion $g(n) := \min_{i \geq 0}\{f^i(n) \leq 1\}$, so folgt: $T(n) \in \Theta(g(n))$. Schon $\log_2^*(n)$ ist eine der am langsamsten wachsenden Funktionen ($\log_2^*(2^{65536}) = 5$), und offenbar wächst $g(n)$ noch wesentlich langsamer.

- $T(n) = 4\,T(\lfloor \frac{n}{2} \rfloor) + n$
  Der Rekursionsbaum bringt uns auf die Formel $T(n) = n^2 + \sum_{i=0}^{log_2(n)-1} 2^i n = n(2^{\log_2(n)} - 1) + n^2 = n(n-1) + n^2 \in \mathcal{O}(n^2)$, die wir nun per Induktion beweisen:
  Behauptung: $T(n) \leq 2n^2 - n$ **Beweis:**

  $$\begin{aligned} n = 1 : T(1) &= 1 \leq 2 - 1 = 1 \\ I.S. : T(n) &\leq 4T(\lfloor \frac{n}{2} \rfloor) + n \\ &\underset{I.A.}{\leq} 4(2\lfloor \frac{n^2}{4} \rfloor - \lceil \frac{n}{2} \rceil) + n \\ &\leq 4(2(\frac{n^2}{4}) - 2n + n = 2n^2 - n \blacksquare \end{aligned}$$

[1] Die log_2^* - Funktion wendet solange den Logarithmus auf ihr Argument an, bis das Resultat 0 ist und liefert die Anzahl der Logarithmusaufrufe zurück

- $T(n) = 3\,T(\lfloor \frac{n}{4} \rfloor) + 2\,T(\lceil \frac{n}{8} \rceil) + 4$
  Hier kann man direkt raten, dass $T(n) \in \mathcal{O}(n)$. Versuchen wir aber induktiv zu Zeigen, dass $T(n) \leq cn$, so stossen wir auf ein Problem, da sich unsere Induktionsannahme als zu schwach herausstellen wird, um die Annahme zu beweisen. Wir zeigen statt dessen die (stärkere) Annahme: $T(n) \leq cn - b$ für ein $b > 0$. **Beweis:**

$$\begin{aligned} n = 1 : T(1) &= 1 \leq c - b \text{ für } c \geq b + 1 \\ I.S. : T(n) &= 3\,T(\lfloor \frac{n}{4} \rfloor) + 2\,T(\lceil \frac{n}{8} \rceil) + 4 \\ &\underset{\text{I.A.}}{\leq} 3c\frac{1}{4}n - 3b + 2\frac{1}{8}n - 2b + 4 \\ &= cn - 5b + 4 \\ &\leq cn - b \text{ für } b = 1 \blacksquare \end{aligned}$$

Aufgabe 2: Rekurrenz, Mastertheorem (10 Punkte)
Das Mastertheorem ermöglicht es, sehr einfach die Lösung von Rekurrenzen der Form $T(n) = aT(\frac{n}{b}) + f(n)$ anzugeben. Dabei wird überprüft, ob die Funktion f oder der Teil $aT(\frac{n}{b})$ die Kosten dominieren werden. Allerdings gibt es zwei Definitionslücken, in denen das Theorem keine Aussage machen kann, nämlich dann, wenn $f(n)$ zwar kleiner (größer) als der von der Rekursion herrührende Term ist, aber nicht *polynomiell* kleiner (größer).

- $T(n) = 36\,T(\frac{n}{6}) + \frac{n^2}{\ln(n)}$ (für $n > 1$)
  Für die Anwendung des Mastertheorems müssen wir zunächst $f(n) := \frac{n^2}{\ln(n))}$ und $n^{\log_6(36)} = n^2$ miteinander vergleichen. Offenbar gilt $f(n) \in \mathcal{O}(n^2)$! Aber um den ersten Fall des Mastertheorems anwenden zu können, müsste $f(n)$ sogar polynomiell kleiner sein als n^2, also müsste ein $\varepsilon > 0$ existieren, so dass $f(n) \in \mathcal{O}(n^{2-\varepsilon})$. Da aber der Logarithmus *langsamer* wächst als *jede Potenz* n^ε, denn

$$\lim_{n\to\infty} \frac{\ln(n)}{n^\varepsilon} \underset{\text{L'Hôpital}}{=} \lim_{n\to\infty} \frac{\frac{1}{n}}{\varepsilon\frac{n^\varepsilon}{n}} = \lim_{n\to\infty} \frac{1}{\varepsilon n^\varepsilon} = 0$$

  , können wir kein $\varepsilon > 0$ finden mit $\frac{n^2}{\ln(n)} \in \mathcal{O}(\frac{n^2}{n^\varepsilon})$, und damit können wir mit dem Mastertheorem hier keine Aussage treffen!

- $T(n) = 2\sqrt{2}\,T(\frac{n}{2}) + 42\sqrt{n^3} + 4711$
  Hier gilt $f(n) := 42\sqrt{n^3} + 4711 = 42n^{\frac{3}{2}} + 4711$ und $n^{log_2(2\sqrt{2})} = n^{\log_2(\sqrt{2^3})} = n^{\log_2(2^{\frac{3}{2}})} = n^{\frac{3}{2}}$. Offenbar gilt $f(n) \in \Theta(n^{\frac{3}{2}})$, und damit nach Fall 2 des Mastertheorems: $T(n) \in \Theta(n^{\frac{3}{2}} \log_2(n))$.

- $T(n) = 125\,T(\frac{n}{5}) + n^2\sqrt{n}$
  Hier gilt $f(n) := n^2\sqrt{n}$ und $n^{\log_5(125)} = n^3$. Damit ist offenbar $f(n) \in \mathcal{O}(n^3)$, und nun müssen wir für den 1.Fall des Mastertheorems noch ein $\varepsilon > 0$ finden, so dass auch $f(n) = n^{\frac{5}{2}} \in \mathcal{O}(n^{3-\varepsilon})$ liegt. Mit $\varepsilon := \frac{1}{2}$ folgt $n^{3-\varepsilon} = n^{\frac{5}{2}} = f(n)$, und damit offenbar $f(n) \in \mathcal{O}(n^{3-\varepsilon})$. Damit folgt nach Fall 1 des Mastertheorems: $T(n) \in \Theta(n^3)$.

- $T(n) = 1000000\,T(\frac{n}{10}) + \frac{1}{4711}e^n$
  Hier gilt $f(n) := \frac{1}{4711}e^n$ und $n^{\log_1 0(1000000)} = n^6$. Offenbar gilt damit auch $f(n) \in \Omega(n^6)$ (siehe Blatt 4), und da die Exponentialfunktion schneller wächst als *jede* Potenz, können wir ein beliebiges $\varepsilon > 0$ wählen, so dass auch $f(n) \in \Omega(n^{6+\varepsilon})$. Nun bleibt für den 3. Fall des Mastertheorems noch zu Zeigen, dass für irgendein $c < 1$ und genügend großes n auch $1000000e^{\frac{n}{10}} \leq ce^n$ gilt. Dazu betrachten wir den Limes $\lim_{n\to\infty} 1000000\frac{e^{\frac{n}{10}}}{e^n} = \lim_{n\to\infty} 1000000e^{-\frac{9}{10}n} = 0$, und damit ist die Behauptung gezeigt. Also gilt nach Fall 3 des Mastertheorems: $T(n) \in \Theta(e^n)$.

- $T(n) = 27\,T(\frac{n}{3}) + 12\,n^3\log^2(n)$
  Hier gilt $f(n) = 12\,n^3\log^2(n)$ und $n^{\log_3(27)} = n^3$. Damit gilt natürlich auch $f(n) \in \Omega(n^3)$, aber können wir ein $\varepsilon > 0$ finden, so dass $f(n) \in \Omega(n^{3+\varepsilon})$? Nein, denn auch das Quadrat des Logarithmus wächst langsamer als jede Potenz, wie man sich leicht klarmachen kann:

$$\begin{aligned}\lim_{n\to\infty} \frac{n^3\log^2(n)}{n^3 n^\varepsilon} &= \lim_{n\to\infty} \frac{\log^2(n)}{n^\varepsilon} \\ &\underset{\text{L'Hôpital}}{=} \lim_{n\to\infty} \frac{2\log(n)}{n^\varepsilon\varepsilon} \\ &\underset{\text{L'Hôpital}}{=} \lim_{n\to\infty} 2\frac{1}{n^\varepsilon\varepsilon^2} = 0\end{aligned}$$

  Und damit können wir das Mastertheorem hier nicht anwenden!

Aufgabe 3: Markov- und Chebychev Ungleichung (10 Punkte)
Zuerst muessen wir aus den gegebenen Werten den Erwartungswert und die Varianz bestimmen. Dazu dividieren wir alle gemessenen Anzahlen durch die Zahl der untersuchten Schüler, um Wahrscheinlichkeiten zu erhalten:

Größe	15 cm	18 cm	20 cm	22 cm	24 cm	26 cm	28 cm
Anzahl	1509	5139	12747	15838	12441	5336	2038
Wahrscheinlichkeit	0.0274	0.093	0.23	0.287	0.226	0.097	0.037

Damit können wir $E[X] = \sum_{x_i} x_i p(x_i)$ bestimmen, wobei x_i die einzelnen Fußgrößen beschreibt. Wir erhalten $E[X] \approx 22.03cm$. Die Varianz bestimmen wir durch $Var[X] = E[X^2] - (E[X])^2 \approx 7.5$.
Damit können wir nun die Abschätzungen nach Markov und nach Chebychev vornehmen.
Markov: $pX \geq t \times E[X]) \leq \frac{1}{t}$.
Eine Abweichung von 20% erhalten wir bei $t = 1.2$ und damit:

$$p(X \geq 1.2 \times E[X]) \leq 0.83$$

was offenbar keine sehr harte Schranke darstellt!

Chebychev: $p(|X - E[X]| \geq t) \leq \frac{Var[X]}{t^2}$.
Eine Abweichung von 20% erhalten wir hier bei $t \approx 4.4$ und damit folgt:

$$p(|X - E[X]| \geq 4.4) \leq 0.38$$

Was unsere erste Abschätzung schon erheblich verbessert!

Aufgabe 4: Fibonacci-Zahlen (10 Punkte)

(a) Eine Laufzeit von $\log_2(n)$ deutet meistens darauf hin, dass wir in unserem Algorithmus das ursprüngliche Problem immer wieder in halb so grosse Teilprobleme aufteilen müssen. Und das lässt sich hier sehr einfach durchführen: wenn n gerade ist, berechnen wir einfach $(a \times a)^{\frac{n}{2}}$. Wenn n ungerade ist, dann ist $n-1$ gerade, also berechnen wir $(a \times a)^{\frac{n-1}{2}} \times a$.

In C++-Code ausgedrückt, lässt sich dies z.B. für den Typ `int` folgendermassen realisieren:

```
int exp(int a, int n)
{
  // Sind wir vielleicht schon fertig?
  if (n == 1)
  {
    return a;
  }
  if (n == 0)
  {
    return 1;
  }
  // ist die Potenz ungerade?
  if (n % 2)
  {
    return exp(a*a, (n-1)/2)*a;
  }
  else
  {
    return exp(a*a, n/2);
  }
}
```

Die Laufzeit lässt sich offensichtlich durch die Rekurrenz

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + \Theta(1)$$

beschreiben. Zum Beweis der Laufzeitabschätzung verwenden wir das Mastertheorem: es gilt $f(n) := 1$ und $n^{\log_2(1)} = n^0 = 1$ und damit offenbar $f(n) \in \Theta(n^{\log_2(1)}) = \Theta(1)$. Daher gilt nach Fall 2 des Mastertheorems $T(n) \in \Theta(\log_2(n))$■.

(b) Die *Fibonacci-Zahlen* Fib(n) sind definiert durch

$$Fib(0) = 0, Fib(1) = 1, Fib(n+2) = Fib(n+1) + Fib(n), n \geq 0$$

Der naiv rekursive Algorithmus zur Bestimmung von Fib(n+2) hat eine Laufzeit von $\Theta(2^n)$. Diese wollen wir nun erheblich verbessern.

(i) **zu Zeigen:**

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} Fib(n-1) & Fib(n) \\ Fib(n) & Fib(n+1) \end{pmatrix} = \begin{pmatrix} Fib(n) & Fib(n+1) \\ Fib(n+1) & Fib(n+2) \end{pmatrix}$$

Beweis:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} Fib(n-1) & Fib(n) \\ Fib(n) & Fib(n+1) \end{pmatrix} = \begin{pmatrix} 0Fib(n-1)+1Fib(n) & 0Fib(n)+1Fib(n+1) \\ 1Fib(n-1)+1Fib(n) & 1Fib(n)+1Fib(n+1) \end{pmatrix} = \begin{pmatrix} Fib(n) & Fib(n+1) \\ Fib(n+1) & Fib(n+2) \end{pmatrix} \blacksquare$$

(ii) Wenn wir Teil (i) iterieren, dann erhalten wir offenbar irgendwann:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} Fib(0) & Fib(1) \\ Fib(1) & Fib(2) \end{pmatrix} = \begin{pmatrix} Fib(n) & Fib(n+1) \\ Fib(n+1) & Fib(n+2) \end{pmatrix}$$

Wenn wir nun noch ausnutzen, dass: Fib(0) = 0, Fib(1) = Fib(2) = 1, dann folgt:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n+1} = \begin{pmatrix} Fib(n) & Fib(n+1) \\ Fib(n+1) & Fib(n+2) \end{pmatrix}$$

Bedenkt man, dass in unserem obigen Exponentiationsalgorithmus keine Bedingungen an den Typ von a gestellt wurden, so können wir damit offenbar auch die n-te Potenz einer Matrix a konstanter Grösse in $\Theta(\log_2(n))$ bestimmen. Also benutzen wir den Algorithmus aus Teil (i) um die $2x2$-Matrix

$$a := \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n+1}$$

zu bestimmen, und erhalten das gesuchte Fib(n+2) als a_{22}.

Aufgabe 5. (Fibonacci-Zahlen, Implementierung Bonuspunkte)
10 Den folgenden Code finden Sie in der Datei `matrix.h`:

```
#include <math.h>
#include <vector>

using namespace std;

// Die Matrizenklasse
template <typename T>
class TMatrix
```

```
{
  public:
    TMatrix(int n, int m);
    TMatrix(const TMatrix<T>& m);
    ~TMatrix();

    // Der Assignmentoperator
    TMatrix<T>& operator = (const TMatrix<T>& m);

    // Addition zweier Matrizen
    TMatrix<T> operator + (const TMatrix<T>& m);

    // Subtraktion zweier Matrizen
    TMatrix<T> operator - (const TMatrix<T>& m);

    // Multiplikation zweier Matrizen
    TMatrix<T> operator * (const TMatrix<T>& m);

    // Skalarmultiplikation
    TMatrix<T> operator * (const T& s);

    // Exponentiation
    TMatrix<T> exp(int n);

    // Elementzugriff
    T& element(int i, int j);
    const T& element(int i, int j) const;

    // Zugriff auf die Dimensionen
    int n() const;
    int m() const;

  protected:
    // Die Dimensionen unserer Matrix
    int n_, m_;
    // Die Daten, die in unserer Matrix gespeichert sind
    vector<T> daten_;
};

// Der Konstruktor
template <typename T>
TMatrix<T>::TMatrix(int n, int m)
  : n_(n),
    m_(m),
    daten_()
{
  daten_.resize(n_*m_);
}
```

```
// Der Copykonstruktor
template <typename T>
TMatrix<T>::TMatrix(const TMatrix<T>& m)
  : n_(m.n_),
    m_(m.m_),
    daten_(m.daten_)
{
}

// Der Destruktor
template <typename T>
TMatrix<T>::~TMatrix()
{
}

// Der Assignmentoperator
template <typename T>
TMatrix<T>& TMatrix<T>::operator = (const TMatrix<T>& m)
{
  n_ = m.n_;
  m_ = m.m_;
  daten_ = m.daten_;

  return *this;
}

template <typename T>
TMatrix<T> TMatrix<T>::operator + (const TMatrix<T>& m)
{
  // koennen wir ueberhaupt addieren?
  if ((n_ != m.n()) || (m_ != m.m()))
  {
    // Nein! Dann geben wir eine 0x0 - Matrix zurueck
    return (TMatrix<T>(0,0));
  }

  // Die Addition von Matrizen geht Elementweise
  TMatrix<T> resultat(n, m);

  for (int i=1; i<=n_; i++)
  {
    for (int j=1; j<=m_; j++)
    {
      resultat.element(i, j) = element(i, j) + m.element(i, j);
    }
  }
```

```
  return resultat;
}

// Subtraktion von Matrizen
template <typename T>
TMatrix<T> TMatrix<T>::operator - (const TMatrix<T>& m)
{
  // koennen wir ueberhaupt subtrahieren?
  if ((n_ != m.n()) || (m_ != m.m()))
  {
    // Nein! Dann geben wir eine 0x0 - Matrix zurueck
    return (TMatrix<T>(0,0));
  }

  // Die Subtraktion von Matrizen geht Elementweise
  TMatrix<T> resultat(n, m);

  for (int i=1; i<=n_; i++)
  {
    for (int j=1; j<=m_; j++)
    {
      resultat.element(i, j) = element(i, j) - m.element(i, j);
    }
  }

  return resultat;
}

// Multiplikation zweier Matrizen
template <typename T>
TMatrix<T> TMatrix<T>::operator * (const TMatrix<T>& m)
{
  // Zuerst ueberpruefen wir die Kompatibilitaet
  // der Faktoren
  if ( (n_ != m.m()) || (m_ != m.m()) )
  {
    // Dann geben wir eine 0x0 Matrix zurueck
    return TMatrix<T>(0,0);
  }

  // Das Resultat einer Multiplikation
  // einer nxm und einer jxk - Matrix
  // ist eine nxk Matrix
  TMatrix<T> resultat(n_, m.m());

  for (int i=1; i<=n_; i++)
  {
    for (int j=1; j<=m.m(); j++)
```

```
    {
      // Belegen aller Elemente der Resultatmatrix.
      // Wir Initialisieren zuerst mit Null
      resultat.element(i,j) = 0;

      // und belegen dann mit dem Matrizenprodukt
      for (int k=1; k<=m_; k++)
      {
        resultat.element(i,j)+=element(i,k)*m.element(k,j);
      }
    }
  }

  return resultat;
}

// Skalarmultiplikation
template <typename T>
TMatrix<T> TMatrix<T>::operator * (const T& s)
{
  TMatrix<T> resultat(n_, m_);

  for (int i=1; i<=n_; i++)
  {
    for (int j=1; j<=m_; j++)
    {
      resultat.element(i,j)*=s;
    }
  }

  return resultat;
}

// Exponentiation
template <typename T>
TMatrix<T> TMatrix<T>::exp(int n)
{
  // Sind wir vielleicht schon fertig?
  if (n == 1)
  {
    return *this;
  }
  if (n == 0)
  {
    TMatrix eins(n_, m_);

    for (int i=1; i<=n_; i++)
    {
```

```
      for (int j=1; j<=m_; j++)
      {
        // Wir kennen ja die Null und die Eins des Typs T noch nicht,
        // Zum Glueck kann man fuer viele numerische Typen mit static_cast
        // das Eins- und das Nullelement erhalten.
        eins.element(i,j) = (i == j) ? static_cast<T>(1) : static_cast<T>(0);
      }
    }

    return eins;
  }
  // ist die Potenz ungerade?
  if (n % 2)
  {
    TMatrix<T> a = *this;
    return (a*a).exp( (n-1)/2 )*a;
  }
  else
  {
    TMatrix<T> a = *this;
    return (a*a).exp(n/2);
  }
}

// Elementzugriff
template <typename T>
T& TMatrix<T>::element(int i, int j)
{
  // Wir muessen irgendwie die zweidimensionalen
  // Daten (i, j) in unserem eindimensionalen Array
  // speichern. Dazu legen wir einfach alle n Zeilen
  // (die jeweils die Laenge m haben) nacheinander
  // im array ab. Dann hat das Element (i,j) die
  // Adresse j + i*m.
  // Allerdings nummeriert man Matrizen ueblicherweise
  // von 1 beginnend, arrays hingegen von 0. Deswegen
  // werden i und j noch um 1 dekrementiert
  --i;
  --j;
  return daten_[i*m_+j];
}

// Elementzugriff in der const-Variante
template <typename T>
const T& TMatrix<T>::element(int i, int j) const
{
  // Wir muessen irgendwie die zweidimensionalen
  // Daten (i, j) in unserem eindimensionalen Array
```

```
  // speichern. Dazu legen wir einfach alle n Zeilen
  // (die jeweils die Laenge m haben) nacheinander
  // im array ab. Dann hat das Element (i,j) die
  // Adresse j + i*m
  // Allerdings nummeriert man Matrizen ueblicherweise
  // von 1 beginnend, arrays hingegen von 0. Deswegen
  // werden i und j noch um 1 dekrementiert
  --i;
  --j;
  return daten_[i*m_+j];
}

// Zugriff auf die Dimensionen der Matrix
template <typename T>
int TMatrix<T>::n() const
{
  return n_;
}

template <typename T>
int TMatrix<T>::m() const
{
  return m_;
}
```

Und ein paar Tests im Programm `fib.C`:

```
#include "matrix.h"

#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
  TMatrix<int> fib(2,2);

  fib.element(1,1) = 0;
  fib.element(1,2) = 1;
  fib.element(2,1) = 1;
  fib.element(2,2) = 1;

  fib = fib.exp(atoi(argv[1]));

  cout << fib.element(1,1) << std::endl;

  return 0;
}
```