

Einführung in die Programmierung

Wintersemester 2010/11

Prof. Dr. Günter Rudolph
 Lehrstuhl für Algorithm Engineering
 Fakultät für Informatik
 TU Dortmund

Kapitel 15: Fallstudien

Inhalt

- Sortieren: Mergesort (auch mit Schablonen)
- Matrixmultiplikation (Schablonen / Ausnahmen)
- Klassenhierarchien

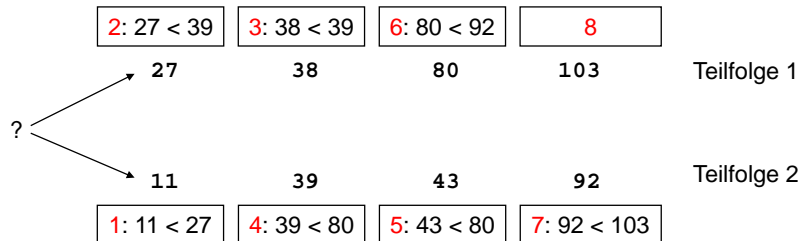
Fallstudien

Kapitel 15

Mergesort

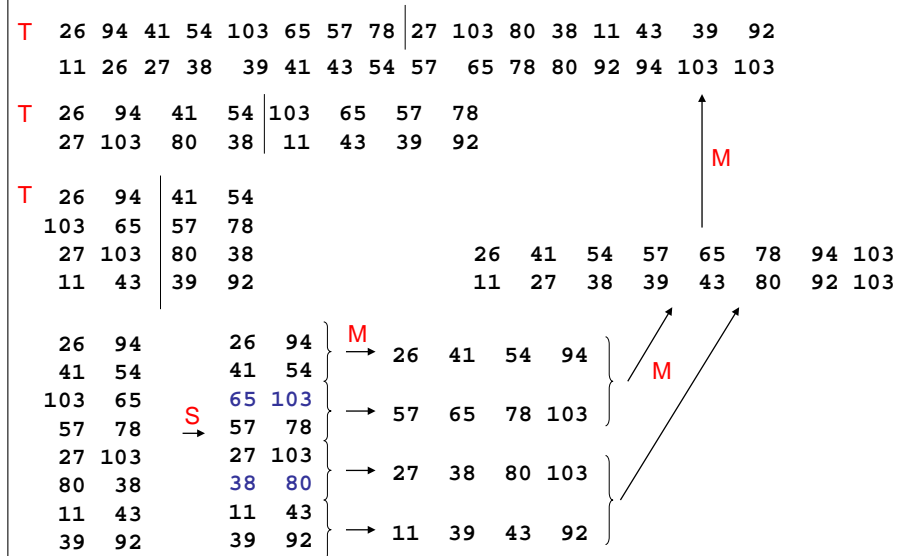
Beobachtung:

Sortieren ist einfach, wenn man zwei sortierte Teilfolgen hat.



Fallstudien

Kapitel 15

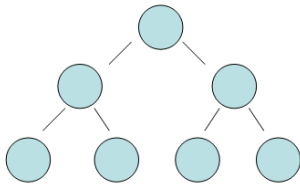


Laufzeitanalyse

Annahme: Anzahl Objekte $n = 2^k \Leftrightarrow k = \log_2 n$ 2^0 Teilsequenzen 2^k Objekte je Teilsequenz 2^1 Teilsequenzen 2^{k-1} Objekte je Teilsequenz 2^2 Teilsequenzen 2^{k-2} Objekte je Teilsequenz

⋮

⋮

 2^{k-1} Teilsequenzen $2^{k-(k-1)}$ Objekte je Teilsequenz
= 2(a) 2^{k-1} Vergleiche zum Sortieren der 2^{k-1} Paare(b) auf Ebene e : $(2^{k-e}-1)$ Vergleiche zum Mischen von 2 der 2^e Sequenzen $\Rightarrow (2^{k-e}-1) \cdot 2^{e-1} = 2^{k-1} - 2^{e-1}$ Vergleiche auf Ebene $e = 1, \dots, k-1$ $\Rightarrow 2^{k-1} + (k-1) \cdot 2^{k-1} - \text{Summe}(2^{e-1}; 1..k-1) = (k-1) \cdot 2^{k-1} + 1 < k \cdot 2^k = n \log_2 n$

Mergesort

- Eingabe: unsortiertes Feld von Zahlen
- Ausgabe: sortiertes Feld
- Algorithmisches Konzept: „Teile und herrsche“ (*divide and conquer*)
 - Zerlege Problem solange in Teilprobleme bis Teilprobleme lösbar
 - Löse Teilprobleme
 - Füge Teilprobleme zur Gesamtlösung zusammen

Hier:

1. Zerteile Feld in Teilfelder bis Teilproblem lösbar (\rightarrow bis Feldgröße = 2)
2. Sortiere Felder der Größe 2 (\rightarrow einfacher Vergleich zweier Zahlen)
3. Füge sortierte Teilfelder durch Mischen zu sortierten Feldern zusammen

Mergesort

- Programmwurf

Annahme:

Feldgröße ist
Potenz von 2

1. Teilen eines Feldes \rightarrow einfach!
2. Sortieren
 - a) eines Feldes der Größe 2 \rightarrow einfach!
 - b) eines Feldes der Größe $> 2 \rightarrow$ rekursiv durch Teilen & Mischen
3. Mischen \rightarrow nicht schwer!

Mergesort: Version 1

```
void Msort(const int size, int a[]) {
    if (size == 2) { // sortieren
        if (a[0] > a[1]) Swap(a[0], a[1]);
        return;
    }
    // teilen
    int k = size / 2;
    Msort(k, &a[0]);
    Msort(k, &a[k]);
    // mischen
    Merge(k, &a[0], &a[k]);
}
```

sortieren (einfach)

sortieren durch Teilen
& Mischen

```
void Swap(int& a, int& b) {
    int c = b; b = a; a = c;
}
```

Werte vertauschen
per Referenz

Mergesort: Version 1

```
void Merge(const int size, int a[], int b[]) {
    int* c = new int[2*size];
    // mischen
    int i = 0, j = 0;
    for (int k = 0; k < 2 * size; k++)
        if ((j == size) || (i < size && a[i] < b[j]))
            c[k] = a[i++];
        else
            c[k] = b[j++];
    // umkopieren
    for (int k = 0; k < size; k++) {
        a[k] = c[k];
        b[k] = c[k+size];
    }
    delete[] c;
}
```

← dynamischen Speicher anfordern

← dynamischen Speicher freigeben

Mergesort: Version 1

```
void Print(const int size, int a[]) {
    for (int i = 0; i < size; i++) {
        cout << a[i] << "\t";
        if ((i+1) % 8 == 0) cout << endl;
    }
    cout << endl;
}

int main() {
    const int size = 32;
    int a[size];
    for (int k = 0; k < size; k++) a[k] = rand();
    Print(size, a);
    Msort(size, a);
    Print(size, a);
}
```

Hilfsfunktion für Testprogramm

Programm zum Testen

Mergesort: Version 1

Ausgabe:

41	18467	6334	26500	19169	15724	11478	29358
26962	24464	5705	28145	23281	16827	9961	491
2995	11942	4827	5436	32391	14604	3902	153
292	12382	17421	18716	19718	19895	5447	21726
41	153	292	491	2995	3902	4827	5436
5447	5705	6334	9961	11478	11942	12382	14604
15724	16827	17421	18467	18716	19169	19718	19895
21726	23281	24464	26500	26962	28145	29358	32391

OK, funktioniert für int ... was ist mit char, float, double ... ?

⇒ Idee: Schablonen!

Mergesort: Version 2

```
template <class T> void Msort(const int size, T a[]) {
    if (size == 2) { // sortieren
        if (a[0] > a[1]) Swap<T>(a[0], a[1]);
        return;
    }
    // teilen
    int k = size / 2;
    Msort<T>(k, &a[0]);
    Msort<T>(k, &a[k]);
    // mischen
    Merge<T>(k, &a[0], &a[k]);
}
```

```
template <class T> void Swap(T& a, T& b) {
    T c = b; b = a; a = c;
}
```

Mergesort: Version 2

```
template <class T> void Merge(const int size, T a[], T b[]) {
    T* c = new T[2*size];

    // mischen
    int i = 0, j = 0;
    for (int k = 0; k < 2 * size; k++) {
        if ((j == size) || (i < size && a[i] < b[j]))
            c[k] = a[i++];
        else
            c[k] = b[j++];

    // umkopieren
    for (int k = 0; k < size; k++) {
        a[k] = c[k];
        b[k] = c[k+size];
    }
    delete[] c;
}
```

Mergesort: Version 2

```
template <class T> void Print(const int size, T a[]) { ... }

int main() {
    const int size = 32;

    int a[size];
    for (int k = 0; k < size; k++) a[k] = rand();
    Print<int>(size, a);
    Msort<int>(size, a);
    Print<int>(size, a);

    float b[size];
    for (int k = 0; k < size; k++) b[k] = rand() * 0.01f;
    Print<float>(size, b);
    Msort<float>(size, b);
    Print<float>(size, b);
}
```

↑
Konstante
vom Typ float
(nicht double)

Mergesort: Version 2

Ausgabe:

41	18467	6334	26500	19169	15724	11478	29358
26962	24464	5705	28145	23281	16827	9961	491
2995	11942	4827	5436	32391	14604	3902	153
292	12382	17421	18716	19718	19895	5447	21726
41	153	292	491	2995	3902	4827	5436
5447	5705	6334	9961	11478	11942	12382	14604
15724	16827	17421	18467	18716	19169	19718	19895
21726	23281	24464	26500	26962	28145	29358	32391
147.71	115.38	18.69	199.12	256.67	262.99	170.35	98.94
287.03	238.11	313.22	303.33	176.73	46.64	151.41	77.11
282.53	68.68	255.47	276.44	326.62	327.57	200.37	128.59
87.23	97.41	275.29	7.78	123.16	30.35	221.9	18.42
7.78	18.42	18.69	30.35	46.64	68.68	77.11	87.23
97.41	98.94	115.38	123.16	128.59	147.71	151.41	170.35
176.73	199.12	200.37	221.9	238.11	255.47	256.67	262.99
275.29	276.44	282.53	287.03	303.33	313.22	326.62	327.57

Mergesort: Version 2

Schablone instantiiert mit Typ `string` funktioniert auch!

Schablone instantiiert mit Typ `Complex` funktioniert **nicht!** Warum?

Vergleichsoperatoren sind nicht überladen für Typ `Complex!`

in `Msort`: `if (a[0] > a[1]) Swap<T>(a[0], a[1]);`

in `Merge`: `if ((j == size) || (i < size && a[i] < b[j]))`

Entweder Operatoren überladen oder überladene Hilfsfunktion (z.B. `Less`):

```
bool Less(Complex &x, Complex &y) {
    if (x.Re() < y.Re()) return true;
    return (x.Re() == y.Re() && x.Im() < y.Im());
}
```

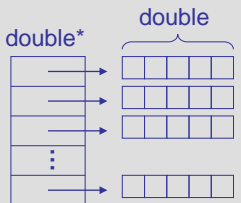
hier:
lexikographische
Ordnung

MATRIX Multiplikation (für quadratische Matrizen)

```
class Matrix {
private:
    double **fElem;
    unsigned int fDim;
public:
    Matrix(unsigned int aDim);
    Matrix(const Matrix& aMat);

    unsigned int Dim() { return fDim; }
    void Set(unsigned int aRow, unsigned int aCol, double aVal);
    double Get(unsigned int aRow, unsigned int aCol);
    void Mult(const Matrix& aMat);
    void Print();

    ~Matrix();
};
```



MATRIX Multiplikation

Konstruktor

```
Matrix::Matrix(unsigned int aDim) : fDim(aDim) {
    if (fDim == 0) throw "matrix of dimension 0";
    fElem = new double* [fDim];
    if (fElem == 0) throw "memory exceeded";
    for (unsigned int i = 0; i < fDim; i++) {
        fElem[i] = new double[fDim];
        if (fElem[i] == 0) throw "memory exceeded";
    }
}
```

Destruktor

```
Matrix::~Matrix() {
    for (unsigned int i = 0; i < fDim; i++)
        delete[] fElem[i];
    delete[] fElem;
}
```

MATRIX Multiplikation

Kopierkonstruktor

```
Matrix::Matrix(const Matrix& aMat) : fDim(aMat.fDim) {
    if (fDim == 0) throw "matrix of dimension 0";
    fElem = new double* [fDim];
    if (fElem == 0) throw "memory exceeded";
    for (unsigned int i = 0; i < fDim; i++) {
        fElem[i] = new double[fDim];
        if (fElem[i] == 0) throw "memory exceeded";
    }
    for (unsigned int i = 0; i < fDim; i++)
        for (unsigned int j = 0; j < fDim; j++)
            fElem[i][j] = aMat.fElem[i][j];
}
```

Speicherallokation
kopieren

MATRIX Multiplikation

```
void Matrix::Set(unsigned int aRow, unsigned int aCol,
                double aVal)
{
    if (aRow >= fDim || aCol >= fDim) throw "index overflow";
    fElem[aRow][aCol] = aVal;
}

double Matrix::Get(unsigned int aRow, unsigned int aCol) {
    if (aRow >= fDim || aCol >= fDim) throw "index overflow";
    return fElem[aRow][aCol];
}
```

Indexbereichsprüfung!

MATRIX Multiplikation

A und B sind quadratische Matrizen der Dimension n.

C = A · B ist dann:

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot a_{kj}$$

```
void Matrix::Mult(const Matrix& aMat) {
    if (aMat.fDim != fDim) throw "incompatible dimensions";
    Matrix tMat(*this);
    for (unsigned int i = 0; i < fDim; i++)
        for (unsigned int j = 0; j < fDim; j++) {
            double sum = 0.0;
            for (unsigned int k = 0; k < fDim; k++)
                sum += tMat.Get(i, k) * aMat.fElem[k][j];
            fElem[i][j] = sum;
        }
}
```

MATRIX Multiplikation

Nächste Schritte

1. Ausnahmen vom Typ char* ersetzen durch unterscheidbare Fehlerklassen:
z.B. class `MatrixError` als Basisklasse,
ableiten: `MatrixIndexOverflow`, `MatrixDimensionMismatch`, ...
2. Umwandeln in Schablone: `template <class T> class Matrix { ...`
also nicht nur Multiplikation für `double`,
sondern auch `int`, `Complex`, `Rational`, ...

Code nach Fehlermeldungen durchforsten ...

Konstruktoren:

```
throw "matrix of dimension 0";    ———> MatrixZeroDimension
throw "memory exceeded";        ———> MatrixMemoryExceeded
```

Get/ Set:

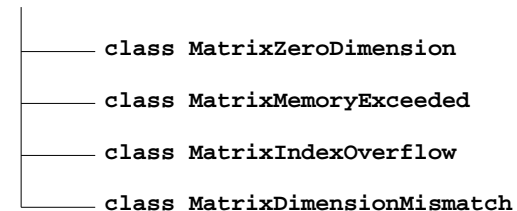
```
throw "index overflow";         ———> MatrixIndexOverflow
```

Mult:

```
throw "incompatible dimensions"; ———> MatrixDimensionMismatch
```

Klassenhierarchie für Ausnahmen

```
class MatrixError
```



Minimale Lösung in C++:

```
class MatrixError {};
class MatrixZeroDimension : public MatrixError() {};
class MatrixMemoryExceeded : public MatrixError() {};
class MatrixIndexOverflow : public MatrixError() {};
class MatrixDimensionMismatch : public MatrixError() {};
```

Schablone MATRIX Multiplikation (für quadratische Matrizen)

```

typedef unsigned int uint;

template <class T> class Matrix {
private:
    T **fElem;
    uint fDim;

public:
    Matrix(uint aDim);
    Matrix(const Matrix& aMat);

    uint Dim() { return fDim; }
    void Set(uint aRow, uint aCol, T aVal);
    T Get(uint aRow, uint aCol);
    void Mult(const Matrix& aMat);
    void Print();

    ~Matrix();
};

```

ursprünglichen Typ
der Nutzdaten
ersetzen durch
generischen Typ **T**

Schablone MATRIX Multiplikation mit vernünftiger Ausnahmebehandlung

Konstruktor

```

template <class T> Matrix::Matrix(uint aDim) : fDim(aDim) {
    if (fDim == 0) throw MatrixZeroDimension();
    fElem = new T* [fDim];
    if (fElem == 0) throw MatrixMemoryExceeded();
    for (uint i = 0; i < fDim; i++) {
        fElem[i] = new T[fDim];
        if (fElem[i] == 0) throw MatrixMemoryExceeded();
    }
}

```

Destruktor

```

template <class T> Matrix::~Matrix() {
    for (uint i = 0; i < fDim; i++)
        delete[] fElem[i];
    delete[] fElem;
}

```

Schablone MATRIX Multiplikation mit vernünftiger Ausnahmebehandlung

Kopierkonstruktor

```

template <class T>
Matrix::Matrix(const Matrix& aMat) : fDim(aMat.fDim) {
    if (fDim == 0) throw MatrixZeroDimension();
    fElem = new T* [fDim];
    if (fElem == 0) throw MatrixMemoryExceeded();
    for (uint i = 0; i < fDim; i++) {
        fElem[i] = new T[fDim];
        if (fElem[i] == 0) throw MatrixMemoryExceeded();
    }
    for (uint i = 0; i < fDim; i++)
        for (uint j = 0; j < fDim; j++)
            fElem[i][j] = aMat.fElem[i][j];
}

```

Speicherallokation
kopieren

Schablone MATRIX Multiplikation mit vernünftiger Ausnahmebehandlung

```

template <class T>
void Matrix::Set(uint aRow, uint aCol, T aVal) {
    if (aRow >= fDim || aCol >= fDim)
        throw MatrixIndexOverflow();
    fElem[aRow][aCol] = aVal;
}

template class <T>
double Matrix::Get(uint aRow, uint aCol) {
    if (aRow >= fDim || aCol >= fDim)
        throw MatrixIndexOverflow();
    return fElem[aRow][aCol];
}

```

Schablone MATRIX Multiplikation mit vernünftiger Ausnahmebehandlung

```
template <class T>
void Matrix::Mult(const Matrix& aMat) {
    if (aMat.fDim != fDim)
        throw MatrixIncompatibleDimensions();

    Matrix tMat(*this);
    for (uint i = 0; i < fDim; i++)
        for (uint j = 0; j < fDim; j++) {
            T sum = 0.0;
            for (uint k = 0; k < fDim; k++)
                sum += tMat.Get(i, k) * aMat.fElem[k][j];
            fElem[i][j] = sum;
        }
}
```

Klassenhierarchie

Erzwingen der Verwendung des Konstruktors der Oberklasse:

Attribute durch **private** schützen, Zugriff nur durch **public** Methoden!

```
class A {
private:
    int a;
public:
    A(int a);
    int Get_a();
    void Show(bool cr = true);
};
```



```
class B : public A {
private:
    int b;
public:
    B(int a, int b);
    int Get_b();
    void Show(bool cr = true);
};
```



```
class C : public B {
private:
    int c;
public:
    C(int a, int b, int c);
    int Get_c();
    void Show(bool cr = true);
};
```

```
A::A(int aa) : a(aa) {}
int A::Get_a() { return a; }
void A::Show(bool cr) {
    cout << a << ' ';
    if (cr) cout << endl;
}
B::B(int aa, int bb) : A(aa), b(bb) {}
int B::Get_b() { return b; }
void B::Show(bool cr) {
    A::Show(false);
    cout << b << ' ';
    if (cr) cout << endl;
}
C::C(int aa, int bb, int cc) : B(aa, bb), c(cc) {}
int C::Get_c() { return c; }
void C::Show(bool cr) {
    B::Show(false);
    cout << c << ' ';
    if (cr) cout << endl;
}
```

a(aa)



Test

```
int main(void) {

    A a(1);
    B b(1,2);
    C c(1,2,3);

    a.Show();
    b.Show();
    c.Show();

    C c2(2*c.Get_a(), 2*c.Get_b(), 2*c.Get_c());
    c2.Show();

    return 0;
}
```

Ausgabe



```
1 2
1 2 3
2 4 6
```