

# Übungsaufgaben zur Klausur

Die Aufgaben folgen keiner speziellen Anordnung, weder was die Schwierigkeit, noch was das Thema betrifft. Fangt also nicht unbedingt mit Aufgabe 1 an, sondern sucht euch erstmal eine Aufgabe, bei der ihr euch sicher seid, dass ihr sie schafft und arbeitet euch dann Stück für Stück weiter.

## Aufgabe 0 (Kurzfragen)

- a) Warum sind Turing-Maschinen im Zusammenhang mit dem Begriff der Berechenbarkeit so interessant? Was ist die Church-Turing-These?
- b) Was ist das Halteproblem? Zu welcher "Klasse" von Problemen gehört es?
- c) Programmiersprachen, in denen man (wenn unendlich viel Speicher zur Verfügung stünde) eine Turingmaschine emulieren kann, nennt man Turing-vollständig. Warum ist diese Eigenschaft interessant, bzw. was sagt das über die Programmiersprache aus?
- d) Was ist eine Referenz, wie wird sie deklariert und wozu benutzt man sie? Warum würde man Referenzen an Stelle von Zeigern benutzen? Wann können Referenzen Zeiger nicht ersetzen?
- e) Welche verschiedenen Anwendungsfälle haben Zeiger?
- f) Was versteht man unter einem Seiteneffekt einer Funktion oder Methode?
- g) Wann spricht man von Funktionen, wann von Methoden?
- h) Welchen Vorteil hat es, Speicher dynamisch zu allokatieren? Und wie geht das überhaupt?
- i) Was heißt es, dass C++ streng typgebunden ist? Und was versteht man unter statischer Typisierung?
- j) Was sind typische Eigenschaften funktionaler Programmierung? Welche Vor- und Nachteile hat funktionale Programmierung?
- k) Was ist ein Konstruktor? Was ist der Default-Konstruktor? Was ist der Copy-Konstruktor?

- l) Was ist der Destruktor?
- m) Was bedeutet Sichtbarkeit im Zusammenhang mit Klassen in C++? Welche verschiedenen "Sichtbarkeiten" gibt es in C++? Wofür werden sie jeweils verwendet?
- n) Was ist Vererbung? Gebe ein Minimalbeispiel an.
- o) Erläutere grob die einzelnen Schritte die durchgeführt werden, um vom Quellcode zum fertigen Programm zu kommen.
- p) Was ist Moore's law? Ist es immer noch gültig? Welche Auswirkungen hat das auf die Softwareentwicklung?
- q) Was versteht man unter Overloading (deutsch: Überladen) von Funktionen?
- r) Was versteht man unter *Call by Value* und *Call by Reference*?
- s) Was versteht man unter der Signatur einer Funktion/ Methode?
- t) Was ist die Rule of Three?
- u) Was sind die Unterschiede zwischen einer *shallow copy* und einer *deep copy*?
- v) Mittels `using namespace std`; kann man alles aus dem Namensraum `std` "bekannt" machen. Welche Nachteile könnte das haben (bspw. in größeren Projekten)?
- w) Was ist eine `friend`-Class?
- x) Was macht das Schlüsselwort `inline` vor einer Funktionendeklaration?
- y) Was ist eine abstrakte Klasse?
- z) Was ist der Unterschied zwischen öffentlicher und privater Vererbung?
- aa) Was sind (rein) virtuelle Methoden?
- bb) Was ist der Unterschied zwischen Overloading (Überladen) und Overriding (Überschreiben)?
- cc) Wozu verwendet man das Schlüsselwort `template`?

- dd) Was ist der Unterschied zwischen `template <class T>` und `template <typename T>`?
- ee) Was ist RISC, was ist CISC?
- ff) Was ist dynamischer Polymorphismus?
- gg) Was ist statischer Polymorphismus?
- hh) Was ist ein vollständiger binärer Baum? Was ist der Unterschied zu einem Heap?
- ii) Wann ist ein Sortierverfahren stabil?

## Aufgabe 1

Ist ein Polynom in der Form  $p(x) = a_0 + a_1x + \dots + a_nx^n$  gegeben, so können einzelne Werte  $p(\xi)$  mit Hilfe des *Horner-Schemas* berechnet werden:

$$\begin{aligned}b_n &= a_n, \\b_k &= a_k + \xi b_{k+1} \text{ für } k = n-1, \dots, 0, \\p(\xi) &= b_0\end{aligned}$$

- a) Schreibe eine rekursive Funktion `double horner(double coeffs[], int deg, double x)`, die für ein Polynom vom Grad `deg`, dessen Koeffizienten im Array `coeffs` stehen, das Polynom an der Stelle `x` auswertet. Die Koeffizienten sollen dabei folgendermaßen in einem Array gespeichert werden:  
Für  $p(x) = 3 - 2x + 3x^2 - 4x^4$  ist `coeffs = { -4, 0, 3, -2, 3 }` (d.h. der höchste Koeffizient kommt zuerst). Außerdem ist `deg = 4` und bspw.  $p(2)$  würde ausgewertet durch den Aufruf `horner(coeffs, 4, 2) = -981`.
- b) Schreibe nun die Funktion aus a) iterativ. Was ist die Zeitkomplexität in Abhängigkeit von `n` der iterativen Variante?
- c)\* Was könnten Vorteile des Horner-Schemas sein? Warum wertet man das Polynom nicht einfach "normal" aus, indem man einfach  $\xi$  einsetzt?

## Aufgabe 2

Schreibe eine Funktion, die ein Polynom der Form  $p(x) = a_0 + a_1x + \dots + a_nx^n$  von der Standardeingabe liest und die Koeffizienten  $a_0, \dots, a_n$  extrahiert.

Bsp.: Eingabe: `2 + 2x - 3x^2 + 4x^3`, Ausgabe: `{ 2, 2, -3, 4 }`

Hinweise:

- Der Einfachheit halber, kann davon ausgegangen werden, dass alle Koeffizienten einstellig sind, also dass  $|a_i| \leq 9$ , für alle  $i = 0, \dots, n$
- Ersetze zunächst alle '-' durch '+-' mit der Methode `std::string::replace` (um bspw. im String `s` alle 'a' durch 'b' zu ersetzen: `replace(s.begin(), s.end(), 'a', 'b')`)
- "Splitte" nun den String jeweils an den '+'. Du kannst annehmen, dass eine Funktion `std::vector<std::string> split(const std::string& s, char delimiter)` gegeben ist, die einen String `s` bei jedem Vorkommen von `delimiter` aufsplittet und die einzelnen Teile in einen `std::`

vector schreibt (also bspw.: `split("abc+def+ghi+jkl", '+')` == { "abc", "def", "ghi", "jkl" }

- Nun kannst du den im vorherigen Schritt erstellten Vektor durchlaufen und die Koeffizienten extrahieren. Hier musst du nur aufpassen, falls ein Koeffizient 0 ist und das dazugehörige Monom gar nicht in der Eingabe vorkam (also bspw.  $3 - 4x^2$ , hier fehlt der Term  $0x$ , d.h. der Koeffizientenvektor ist hier { 3, 0, -4 })

### Aufgabe 3

Schreibe eine Funktion `std::vector<std::string> split(const std::string &s, char delimiter)`, die einen String `s` bei jedem Vorkommen von `delimiter` aufsplittet und die einzelnen Teile in einen `std::vector` schreibt (also bspw. sollte das Ergebnis von `split("abc+def+ghi+jkl", '+')` das hier sein: { "abc", "def", "ghi", "jkl" })

- Einen neuen `std::vector`, der `std::string`s enthält, legt man so an:  
`std::vector<std::string> res;`
- Um `res` ein neues Element hinzuzufügen, verwende `res.push_back("test")`
- Die Funktion (siehe auch Übungsblatt 10)  
`std::istream& getline(std::istream& is, std::string &s)`  
akzeptiert auch noch einen dritten Parameter `char` `delimiter` und kann auch auf `std::stringstream`s operieren (um einen `std::stringstream` aus einem String zu erstellen, übergibt man einfach den jeweiligen String dem Konstruktor).

### Aufgabe 4

Implementiere die Funktion `void reverse(std::string& s)`, die den String `s` "umdreht", also bspw. aus `Hallowelt tlewo1laH` macht, auf die folgenden Arten:

- Implementiere die Funktion mit einer `for`-Schleife und der Funktion `std::swap(char &a, char &b)`, die den Wert von `a` und `b` tauscht. Die `for`-Schleife soll dabei *nicht* den ganzen String durchlaufen, sondern (falls `n` die Länge des Strings ist) nur von `i = 0` bis `i < n / 2`.
- Implementiere die Funktion rekursiv, die Signatur der Funktion ändert sich dabei zu: `std::string reverse(const std::string &s)` und die Funk-

tion ändert nicht den ursprünglichen String sondern gibt das Ergebnis zurück.

### Aufgabe 5

Rechne die folgenden Zahlen um:

a)  $11011101_2$  ins Dezimalsystem

b)  $116_{10}$  ins Dualsystem

### Aufgabe 6

Gegeben seien zwei Rechtecke in einem 2D-Koordinatensystem (jeweils parallel zu den Koordinatenachsen). Diese können eindeutig durch Angabe der Koordinaten der linken oberen Ecke und der rechten unteren Ecke bestimmt werden:

```
class Rectangle {  
    public:  
        double x1, y1; // linke obere Ecke  
        double x2, y2; // rechte untere Ecke  
}
```

Schreibe eine Methode `bool Rectangle::doesOverlap(const Rectangle other)`, die für ein Rechteck prüft, ob sich seine Fläche und die Fläche eines anderen Rechtecks überschneiden.

### Aufgabe 7

a) Implementiere eine Klasse, die eine doppelt verkettete Liste repräsentiert und zumindest folgendes Interface erfüllt:

```
class DLL {  
    public:  
        void addFirst(int value);  
        int removeFirst();  
  
        void print() const;  
        void printReverse() const;  
  
        ~DLL();  
  
    private:  
        struct IntListItem {  
            int value = 0;
```

```

        IntListItem *next = nullptr;
        IntListItem *prev = nullptr;
    };

    IntListItem *first = nullptr;
    IntListItem *last = nullptr;
    int count = 0;
};

```

Untenstehende main-Funktion soll folgenden Output erzeugen:

List: 4 -> 3 -> 2 -> 1

List (reversed): 1 -> 2 -> 3 -> 4

```

int main() {
    DLL list;
    list.addFirst(1);
    list.addFirst(2);
    list.addFirst(3);
    list.addFirst(4);

    list.print();
    list.printReverse();

    return 0;
}

```

- b) Welche Vorteile hat diese Variante gegenüber einfach verketteten Listen?  
Und gegenüber einem normalen Array? Welche Nachteile gibt es?

## Aufgabe 8

Gegeben sei die Turingmaschine  $M$  mit den Endzuständen  $z_3$  und  $z_5$ , dem Startzustand  $z_0$  und untenstehendem Programm.

Zustand	Eingabe	Ausgabe	Richtung	Folgez.
$z_0$	$b$	$b$	$R$	$z_1$
$z_1$	$b$	$0$	$L$	$z_3$
$z_1$	$0$	$b$	$R$	$z_2$
$z_2$	$b$	$0$	$L$	$z_4$
$z_2$	$0$	$b$	$R$	$z_1$
$z_4$	$b$	$0$	$L$	$z_5$

Die Turingmaschine berechnet die charakteristische Funktion  $\chi_M : \mathbb{N} \rightarrow \{0,1\}$  einer unbekannten Menge  $M \subseteq \mathbb{N}$ , d.h.

$$\chi_M(n) = \begin{cases} 1 & n \in M \\ 0 & n \notin M \end{cases}$$

- a) Ziel ist es, herauszufinden, um welche Menge  $M$  es sich handelt. Eingabe und Ausgabe werden dabei durch eine Unärdarstellung codiert:  $n \in \mathbb{N}$  entspricht  $0^{n+1}$ , d.h.  $0 = 0$ ,  $1 = 00$ ,  $2 = 000$  etc. Die Ausgabe der Turingmaschine ist am Ende die Zahl, die rechts vom Schreib-Lese-Kopf steht.
- b) Entwerfe nun eine Turingmaschine, welche die charakteristische Funktion  $\chi_{M^c}$  des Komplements  $M^c = \mathbb{N} \setminus M$  berechnet, d.h.  $\chi_{M^c}(n) = 1 - \chi_M(n)$ .

## Aufgabe 9

Du stößt in einem Programm auf folgenden Ausschnitt:

```
// Prüfe, ob der Eintrag an der Stelle i, j Null ist
if (std::abs(matrix[i][j]) < 1e-12) {
    ...
}
```

Erkläre, warum der Code und das Kommentar darüber (auch wenn nicht ideal formuliert) trotzdem Sinn ergeben.

## Aufgabe 10

Formalisiere den Begriff Algorithmus. Welche speziellen Typen von Algorithmen gibt es und was zeichnet sie aus? Nenne jeweils ein Beispiel.

## Aufgabe 11

Ersetze im folgenden Programmausschnitt die **if-else**-Konstruktion durch eine einzige möglichst vereinfachte **if**-Anweisung:

```
if (true) {
    if (x > 0) {
        if (x%2 == 0)
            std::cout << x;
    } else {
        if ((-x)%3 == 0)
            std::cout << x;
    }
}
```

## Aufgabe 12

- a) Was macht der folgende Algorithmus?



```

void func(int a[], int b) {
    for (int j = 1; j < b; j++)
        for (int i = j; i > 0 && a[i-1] > a[i]; i--) {
            int c = a[i];
            a[i] = a[i-1];
            a[i-1] = c;
        }
}

```

b) Welche Laufzeit hat der Algorithmus im schlechtesten Fall?

### Aufgabe 13

a) Berechne die Zweierkomplementdarstellung der folgenden Zahlen:

i) -30

ii) 25

iii) 123

b) Rechne die folgenden Zahlen, die in Zweierkomplementdarstellung vorliegen, ins Dezimalsystem um:

i) 11111111

ii) 11100000

iii) 00000001

c) Welche Vorteile hat die Zweierkomplementdarstellung?

### Aufgabe 14

Implementiere eine Klasse, die einen Stack modelliert. Der Typ der Werte auf dem Stack, soll durch templates allgemein gehalten werden, d.h. die Klassendefinition sieht folgendermaßen aus:

```

template <class T>
class Stack {
public:
    Stack(int capacity);
    ~Stack();
    bool push(T);
    T pop();
}

```

```
private:
    ...
};
```

## Aufgabe 15

- a) Erstelle eine Funktion `double mean(const std::vector<double>& v)`, die den Mittelwert  $\mathbb{E}[v]$  aller Einträge in dem Vektor  $v$  zurückliefert:

$$\mathbb{E}[v] = \frac{1}{N} \sum_{i=1}^N v_i,$$

wobei  $N$  die Anzahl der Einträge im Vektor angibt.

- b) Erstelle eine Funktion `double median(const std::vector<double>& v)`, die den Median aller Einträge in dem Vektor  $v$  zurückliefert. Um den Median zu berechnen, erstelle eine sortierte Kopie  $\tilde{v}$  von  $v$  und bestimme  $M(v)$  als

$$M(v) = \begin{cases} \tilde{v}_{\frac{N+1}{2}} & N \text{ ungerade} \\ \frac{1}{2}(\tilde{v}_{\frac{N}{2}} + \tilde{v}_{\frac{N}{2}+1}) & N \text{ gerade} \end{cases}$$

wobei  $N$  die Anzahl der Einträge im Vektor angibt. Beachte die unterschiedlichen Indizierungsstrategien (1-basiert in der Formel oben, 0-basiert in C++), sowie den Spezialfall, dass der Vektor leer ist. Benutze zum Sortieren `std::sort(v.begin(), v.end())`.

## Aufgabe 16

- a) Schreibe eine Funktion `bool check_parentheses(std::string symbols)`, die für eine Zeichenkette `symbols` überprüft, ob diese genau so viele öffnende Klammern '(' wie schließende ')' enthält. Die Funktion soll auch `false` zurückgeben, wenn eine schließende Klammer vor einer öffnenden Klammer auftritt.
- Verwende zunächst einfach eine `int`-Variable um mitzuzählen.
  - Verwende `std::stack<char>`.
- b) Kann man einen endlichen Automaten definieren, der dieselbe Aufgabe löst?

### Aufgabe 17

Gegeben seien folgende Zahlen:

$$\{5, 8, 2, 10, 1, 3, 2, 6\}$$

- Baue aus diesen Zahlen einen Max-Heap (d.h. das größte Element steht in der Wurzel). Zeichne den vollständigen Baum nach jedem Schritt und erkläre kurz, wie das Element eingefügt wurde.
- Lösche nun die Wurzel aus dem Baum und erkläre die dabei auftretenden Schritte.
- Wieviele "Ebenen" hat ein Heap (in Abhängigkeit der Anzahl der Elemente  $n$ )? Wie kann man daraus die Komplexität der Operation "Einfügen" ableiten?
- Erläutere kurz die Funktionsweise von Heapsort.

### Aufgabe 18

Schreibe eine templatisierte Funktion `quicksort`, die einen `std::vector<T>` mittels dem Quicksort-Algorithmus sortiert. Das Pivot-Element soll dabei immer das "mittlere" Element der jeweiligen Liste sein (bei einer geraden Anzahl an Elementen soll das letzte Element der ersten Hälfte ausgewählt werden).

### Aufgabe 19

Es seien zwei Algorithmen  $f, g$  gegeben, die beide die selbe Aufgabe lösen. Es ist bekannt, dass

$$f(n) = \mathcal{O}(n) \quad \text{und} \quad g(n) = \mathcal{O}(n^2)$$

a) Diskutiere die folgende Aussage:

*Der Algorithmus  $f$  ist schneller als  $g$ .*

b) Es wurde nun die Zeit in ms gemessen, die die beiden Algorithmen für verschiedene Eingabelängen  $n$  brauchten:

Eingabelänge	Laufzeit von $f$	Laufzeit von $g$
10	1000000	100
100	10000000	10000
1000	100000000	1000000
10000	1000000000	100000000

Offenbar ist  $g$  für all diese Eingabelängen schneller als  $f$ . Steht das im Widerspruch zu deiner Erklärung zur Aussage von a)?

- c) Nenne die (asymptotischen) Laufzeiten der in der Vorlesung besprochenen Sortieralgorithmen und sortiere diese.