Klassen von Algorithmen

Themenverzeichnis

Eingrenzung des Algorithmus-Begriffes ➔ Kriterien

- *Beschreibbarkeit*
  - *Ein-/Ausgabe-Beschreibung*
  - *Funktion des Algorithmus*
- *Endlichkeitsbedingung*
- *Unendlichkeitsbedingung*
- *Variabilitätsbedingung*
- *Lösbarkeitsbedingung*
  - *Kategorien der Lösbarkeit*
  - *Nachweislich unlösbare Problemstellungen*

Terminierung von Prozeduren und das Halteproblem

Die Klassen P und NP

Beispiele von bekannten Problemen

Strategien für Optimierungsprobleme

- *Brute force*
- *Greedy*
- *Rekursion*
  - *Backtracking*
  - *Branch-and-Bound*
- *Dynamische Programmierung*
- *Hill-Climbing*
- *Annealing*
- *Evolutionäre Ansätze*

Repetition

Es werden einige fortgeschrittene theoretische Aspekte thematisiert.

Die Themen werden nicht vertieft ➔ es wird hauptsächlich ein Überblick von Begriffen samt repräsentativen Beispielen gegeben (➔ viele von diesen Beispielen sind „klassische“ und oft zitierte Algorithmen).

Wie definiert man grundsätzlich eine algorithmische Problemlösung?

Am Anfang haben wir aus pragmatischen Gründen eine sehr intuitive und unscharfe Definition benutzt ➔

- ⇨ "Eine Vorschrift oder ein Verfahren, um ein bestimmtes Ziel zu erreichen".

Jetzt wollen wir etwas genauer sein und schränken diese breite Definition mit untenstehenden Kriterien ein.

Kriterium 1
Das Verfahren muss beschreibbar sein (eine naheliegende Anforderung; „kreativ sein“ wäre z.B. kein algorithmisches Verfahren, weil man dabei keine genaue Beschreibung befolgen kann).
Ein üblicher Teil der Beschreibung ist die *funktionale Beschreibung* (Ein-/Ausgabe-Beschreibung):

Menge von erlaubten Eingaben ➔ Menge von erwarteten zugeordneten Ausgaben

Diese Beziehung Eingabe ➔ Ausgabe nennt man "*die Funktion*" des Algorithmus (wie bereits erwähnt, gibt es mehrere Wege, die Funktion umzusetzen; die Funktion beschreibt nicht den Algorithmus an sich).
Die Funktion kann auch als Problemstellung der algorithmischen Lösung verstanden werden.

Einige Beispiele der Ein-/Ausgabe-Beschreibung

Eingabe: eine Folge von ganzen Zahlen
Ausgabe: die maximale Teilsumme der Folge

Eingabe: ein (längerer) Text und ein (kürzeres) Wort
Ausgabe: eine natürliche Zahl n, die angibt, ab welcher Stelle im Text das Wort (zum ersten Mal) vorkommt bzw. -1, falls das Wort nirgends im Text vorkommt

Eingabe: eine Reihung von Objekten mit Schlüsselkomponente
Ausgabe: eine Reihung, welche die gleichen Objekte enthält, aber sortiert ist nach Schlüsseln

Eingabe: eine positive ganze Zahl n, d.h. $n \in \{ 1, 2, 3, \dots \}$
Ausgabe: die n-te Primzahl

Kriterium 2

Jede einzelne erlaubte Eingabe und jede einzelne erwartete Ausgabe muss durch eine *endliche* Zeichenkette darstellbar sein.
(➔ *Endlichkeitsbedingung*)

Die folgenden zwei Ein-/Ausgabe-Beschreibungen erfüllen die Endlichkeitsbedingung nicht.

Eingabe: eine positive, ganze Zahl n, d.h. $n \in \{ 1, 2, 3, \dots \}$
Ausgabe: der Kehrwert von n (d.h. $1/n$), dargestellt als Dezimalbruch, vollständig und ohne „Abkürzungen" wie „usw. usw." oder „…" oder Ähnliches

Eingabe: keine
Ausgabe: die Zahl π in Dezimalbruch-Schreibweise

Kriterium 3

Es müssen *unendlich* viele verschiedene Eingaben erlaubt sein.
(➔ *Unendlichkeitsbedingung*)

Probleme mit einer endlichen Anzahl von Eingaben sind trivial ➔ man kann sie lösen durch Rückgabe von gespeicherten fest zugewiesenen Ausgabenwerten.

Die folgende Problemstellung erfüllt die Unendlichkeitsbedingung nicht.

Eingabe: Student der Klasse *Bsc INF 2014 ZH1* mit seinem Geburtsdatum
Ausgabe: Sternzeichen des Studenten

Kriterium 4

Es müssen mehr als nur ein Ausgabenwert möglich sein.
(➔ *Variabilitätsbedingung*)

Wenn nur ein Wert als Ausgabe möglich ist, besteht kein Lösungsbedarf.

Die folgende Ein-/Ausgabe-Beschreibung erfüllt die Variabilitäts-bedingung nicht.

Eingabe: zwei natürliche, ungerade Zahlen *n*, *m* d.h. *n* ε { 1, 3, 5, 7, … } , *m* ε { 1, 3, 5, 7, … }
Ausgabe: (n – m) % 2

Bei der untenstehenden Problemstellung ist die obige Bedingung ebenfalls nicht erfüllt, wobei der Nachweis davon etwas subtiler ist.

Eingabe: Ein Dorf mit genau einem Barbier, in dem sich alle Männer am Samstag einer Rasur Unterziehen (wir nehmen dazu fiktive, virtuelle Dörfer, um die Unendlichkeitsbedingung zu wahren)

Ausgabe: „**true**" wenn der Barbier genau diejenige Männer rasiert, die sich nicht selber rasieren, ansonsten „**false**"

Eine Prüfung von sämtlichen Optionen zeigt, dass „**true**" nicht möglich ist.

Kriterium 5

Für jede einzelne erlaubte Eingabe muss es „im Prinzip“ möglich sein, die entsprechende erwartete Ausgabe zu ermitteln.
(➔ *Lösbarkeitsbedingung*)

Andere Formulierung der Lösbarkeitsbedingung: Es darf kein Beweis bekannt sein, dass für eine bestimmte Eingabe keine Lösung existiert.

In Bezug auf Lösbarkeit verteilen sich die algorithmischen Problemstellungen in drei Kategorien:

- **(i)** Lösung existiert
- **(ii)** Man kann beweisen, dass keine Lösung existiert
- **(iii)** Weder hat jemand eine Lösung gefunden, noch die Unlösbarkeit bewiesen

Die Lösbarkeitsbedingung gilt nur für die Kategorie **(ii)** als nicht erfüllt.

Die untenstehende Problemstellung gehört offensichtlich zur Kategorie **(ii)** und erfüllt somit die Lösbarkeitsbedingung nicht.

Eingabe: eine ganze Zahl z
Ausgabe: eine ganze Zahl w, für die gilt: $w \cdot w = z$

Für **z == 5** und **z == -2** gibt es keine Ausgabe.

Bei anderen Problemstellungen ist eine oft aufwändige Analyse notwendig, um ihre Unlösbarkeit zu beweisen.
Ein klassisches Beispiel dieser Art ist das sog. *Halteproblem* (siehe den nächsten Abschnitt).
Es sind auch zahlreiche Beispiele der Kategorie **(iii)** bekannt, bei welchen bezüglich Lösbarkeit gegenwärtig keine Aussage möglich ist. Bis auf weiteres gilt für sie die Lösbarkeitsbedingung als erfüllt.

Terminierung von Prozeduren und das Halteproblem

Mit Terminierung ist hier ein Ende des Ablaufs aufgrund der Prozedur-Logik gemeint, nicht ein technisch bedingter Abbruch (wie Stack-Overflow bei Rekursion, kein Speicherplatz auf der Festplatte u.ä.).

In vielen Situationen kann leicht erkannt werden, ob eine Prozedur terminiert oder nicht (siehe untenstehende Beispiele).

```
static void haeltNie(int n) {
  while (true)
    Console.println(n);
}
```

➔ endlose Schleife für jedes **n**

```
static void haeltImmer(final String s) {
  Console.println(s);
}
```

➔ terminiert für jedes **s**

```
static void haeltWennGroesser1(int n) {
  while (n != 1)
    n = n/2;
}
```

➔ terminiert für **n >= 1**, sonst nicht

```
static void haeltWennNichtLeer(final String s) {
  if (s.length() > 10)
    Console.println(s);
  else
    haeltWennNichtLeer(s + s);
}
```

➔ keine Terminierung, wenn **s** ein leerer String ist, sonst terminiert die Prozedur

Andererseits konnte noch niemand beweisen, dass die folgende sog. Collatz-Prozedur für jedes **n** terminiert, obgleich kein Fall der Nicht-Terminierung bekannt ist.

```
static void haeltVielleicht(int n) { // requires n > 0
  while (n != 1)
    if ((n % 2) == 0) // n ist gerade
      n = n / 2;
    else // n ist ungerade
      n = 3 * n + 1;
}
```

Beim Halteproblem geht es um die Frage, ob prinzipiell ein Algorithmus denkbar wäre, der den Code einer beliebigen Prozedur analysiert und ermittelt, ob sie terminiert oder nicht.

Es kann bewiesen werden, dass ein solcher Terminierung-Analysator nicht existieren kann.
Dieses Ergebnis kann man anschaulich nachvollziehen für den Spezialfall von zu analysierenden Funktionen, in welchen die Eingabe aus einem String besteht.
Dabei nehmen wir zuerst an, dass der fragliche Terminierung-Analysator für Algorithmen der obigen Kategorie (Eingabe = 1 String) existiert und bezeichnen seine Java-Umsetzung als Methode *goedel()*:

```
static boolean goedel(String programm, String eingabe);
```

goedel() gibt folgerichtig **true** zurück, wenn die analysierte Prozedur *programm* mit dem Parameter *eingabe* terminieren wird, ansonsten wird **false** zurückgegeben.

Anschliessend konstruieren wir auf dieser Grundlage die untenstehende Prozedur *programm* (mit einem internen *goedel()*-Verweis) und rufen dann die *programm* –Prozedur mit ihrem eigenen Code als Parameter auf:

```
static void programm(String s) {
  if (goedel(s, s))
    while (true); // hier gerät programm in eine Endlosschleife
  else
    ; // hier hält programm sofort
}

programm(
  "void programm(String s) {if (goedel(s, s)) while (true); else ;}"
);
```

Wir wissen nicht, ob der interne Methodenaufruf *goedel(s, s)* **true** oder **false** zurück gibt, seine Prognose wird aber in beiden Fällen dem effektiven Ergebnis der Ausführung von *programm* widersprechen.

Aus diesem Widerspruch folgt zwingend, dass der Terminierung-Analysator *goedel()* nicht existieren kann.

Die Klassen P und NP

Viele Problemstellungen sind theoretisch lösbar (oft sogar sehr einfach lösbar), die Ausführung der Lösung erfordert jedoch einen so hohen Zeitaufwand, dass sie ab einer bestimmten Eingabegrösse nicht praktikabel ist.

Als praktisch lösbar werden Problemstellungen betrachtet, für die eine Lösung mit polynomialer Komplexität (oder eine noch weniger aufwändige) existiert.
Wenn nur Lösungen mit exponentieller Komplexität (oder noch aufwändigere) bekannt sind, gilt die Problemstellung als praktisch unlösbar. Als „Ausweg zweiter Wahl“ bleibt in solchen Fällen der Einsatz von sog. *nicht-deterministischen Maschinen*.
Bei der nicht-deterministischen Vorgehensweise werden die erfolgversprechenden Varianten anhand von intuitiven Regeln selektiert (die Maschinen „raten“). Demzufolge garantiert das Vorgehen nicht, dass sämtliche gültige Ausgaben gefunden werden, dafür bleibt aber der Zeitaufwand polynomial.
Problemstellungen, die nur auf diese Art praktisch lösbar sind, bezeichnet man als ***NP*** - Probleme (***N***icht-deterministisch ***P***olynomial), wobei für sog. ***P*** – Probleme eine ***P***olynomiale und vollumfassende (deterministische) Lösung existiert.

NP-vollständige Probleme
Bei der Frage, ob ein bestimmtes Problem polynomial unlösbar ist kann die Untersuchung seiner Zugehörigkeit zu sog. NP-vollständigen (NP-C) Problemen behilflich sein. Probleme dieser Kategorie sind miteinander derart verwandt, dass allfällige polynomiale Lösung eines NP-C – Problems unmittelbar auf alle anderen NP-C – Probleme übertragbar wäre („ein polynomial lösbar ==> alle polynomial lösbar“).
Es ist allerdings keine polynomiale Lösung eines NP-C – Problems bekannt. Demzufolge wird allgemein (ohne Beweis) angenommen, dass NP-C – Probleme eine Teilmenge der NP-Klasse sind.
Wenn bei einem spezifischen Problem gezeigt werden kann, dass es NP-vollständig ist, wird es der NP-Klasse zugeordnet.
Für die NP-C – Klassifizierung existieren schlüssige Verfahren, die jedoch nicht im Rahmen dieses Kurses liegen.

Zusammenfassung

P- vs. NP-Probleme

- Grenze zwischen **effizient** lösbaren (polynomialer Aufwand) und **nicht effizient** lösbaren (exponentieller Aufwand) Problemen
- **Problemklasse P**: Menge aller Probleme, die mithilfe deterministischer Algorithmen in polynomialer Zeit gelöst werden können
- **Problemklasse NP**: Menge aller Probleme, die nur mithilfe nichtdeterministischer Algorithmen in polynomialer Zeit gelöst werden können
  - Nichtdeterminismus: „Raten“ der richtigen Variante bei mehreren Lösungen
  - Umsetzung mit deterministischen Algorithmen: exponentieller Aufwand

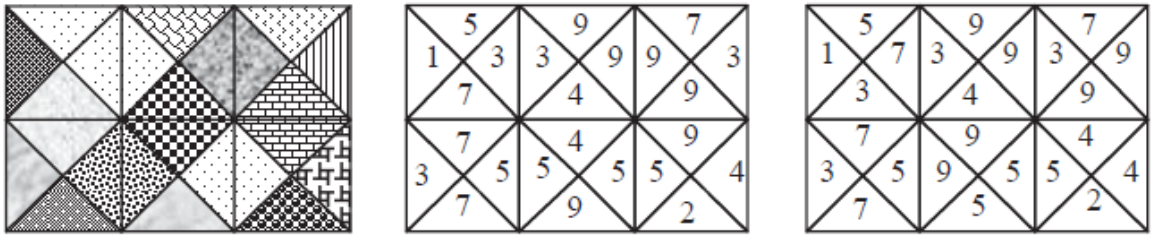
Beispiele von bekannten Problemen

In diesem Abschnitt werden einige oft zitierte Problemstellungen skizziert.
Ein Teil davon gehört zu den Klassen P und NP, andere Beispiele sind theoretisch unlösbar.

Das Kachelproblem

Gegeben ist ein Katalog von quadratischen Kacheln, welche wie folgt beschriftet oder gefärbt sind:

Eine rechteckige Fläche soll mit Kacheln aus diesem Katalog so bedeckt werden, dass anliegende Dreiecke immer die gleiche Beschriftung (die gleiche Farbe) haben:



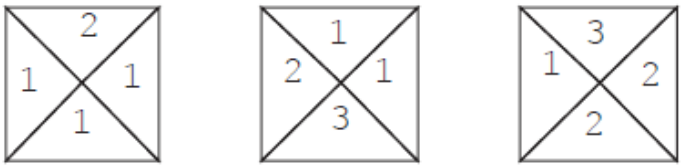
richtig richtig falsch

(Jede Kachelseite hat die Länge 1 und die Abmessungen der zu belegenden Fläche sind natürliche Zahlen).

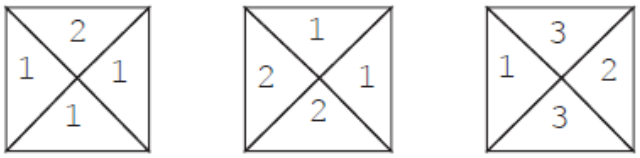
Für einen konkreten Katalog und eine konkrete Fläche können die möglichen Belegungen z.B. mit einem "Brute Force" - Verfahren ermittelt werden.

Es lässt sich auch zeigen, dass mit bestimmten Katalogen beliebige Flächen belegt werden können, mit anderen nicht.

Mit Kacheln aus dem folgenden Katalog kann man jede endliche Fläche kacheln:



Mit Kacheln aus dem folgenden Katalog kann man *nicht* jede endliche Fläche kacheln:



Für die untenstehende Problemstellung gibt es jedoch keine Lösung (kann bewiesen werden).

Eingabe: ein Kachelkatalog
Ausgabe: „Ja“, wenn man mit (Kacheln aus) diesem Katalog jede endliche Fläche kacheln kann; „Nein“ sonst

Das zweite Kachelproblem

Eingabe: n^2 viele Kacheln, wobei $n = 2, 3, 4, 5, \ldots$ ist.
Ausgabe: „Ja“, wenn man mit diesen Kacheln ein $n \cdot n$-Quadrat kacheln kann, „Nein“ sonst

Schon für **n == 5** gibt es **25!** (ca. **$1.55 * 10^{25}$**) „Brute Force“ - Alternativen, die überprüft werden müssen ➔ das dauert Millionen von Jahren.

Dieses Problem gehört zur Klasse NP.

*Das **Post'sche** Korrespondenzproblem (auch als Paligrammproblem bekannt)*

Gegeben sind zwei Wortfolgen gleicher Länge, z.B.:

```
String[] u = { "abb", "a", "bab", "baba", "aba" };
String[] v = { "bbab", "aa", "ab", "aa", "a" };
```

Index	*1*	*2*	*3*	*4*	*5*
u	abb	a	bab	baba	aba
v	bbab	aa	ab	aa	a

Es soll eine Indexfolge gefunden werden, mit der sich via Verkettung von entsprechenden Einzelwörtern in beiden Folgen das gleiche Wort ergibt:

```
u[2] + u[1] + u[1] + u[4] + u[1] + u[5] == "aabbabbbabaabbaba"
v[2] + v[1] + v[1] + v[4] + v[1] + v[5] == "aabbabbbabaabbaba"
```

Für die folgenden zwei Wortfolgen ist keine solche Verkettung zum gemeinsamen Resultat möglich:

Index	1	2	3	4	5
u1	bb	a	bab	baba	aba
v1	bab	aa	ab	aa	a

Für konkrete zwei Wortfolgen können Lösungen ermittelt werden, für die untenstehende Problemstellung gibt es jedoch keine Lösung (kann bewiesen werden).

Eingabe: zwei gleich lange Reihungen `u` und `v` von Wörtern.
Ausgabe: „Ja", wenn es eine Folge `i1, i2, … , in` von Indices gibt, sodass

```
u[i1] + u[i2] + … + u[in] == v[i1] + v[i2] + … + v[in]
```

gilt, und „Nein" sonst

.

Das Rucksackproblem (auch als Einbrecherproblem bekannt)

Eingabe: eine Liste von „Gegenständen“ (z.B. Schlafsack, Feldflasche, Zelt usw.). Jeder Gegenstand hat ein Gewicht (eine natürliche Zahl, z.B. die Anzahl der Gramm) und einen „Wert“ (ebenfalls durch eine natürliche Zahl ausgedrückt). Außerdem ist bekannt, wie viel Gramm in den Rucksack passen, z.B. 15 000 Gramm.
Ausgabe: eine Liste der Gegenstände, die eine möglichst wertvolle Füllung des Rucksacks ausmachen. Dies bedeutet, dass keine andere Menge von Gegenständen, die in den Rucksack passt, einen höheren Wert hat.

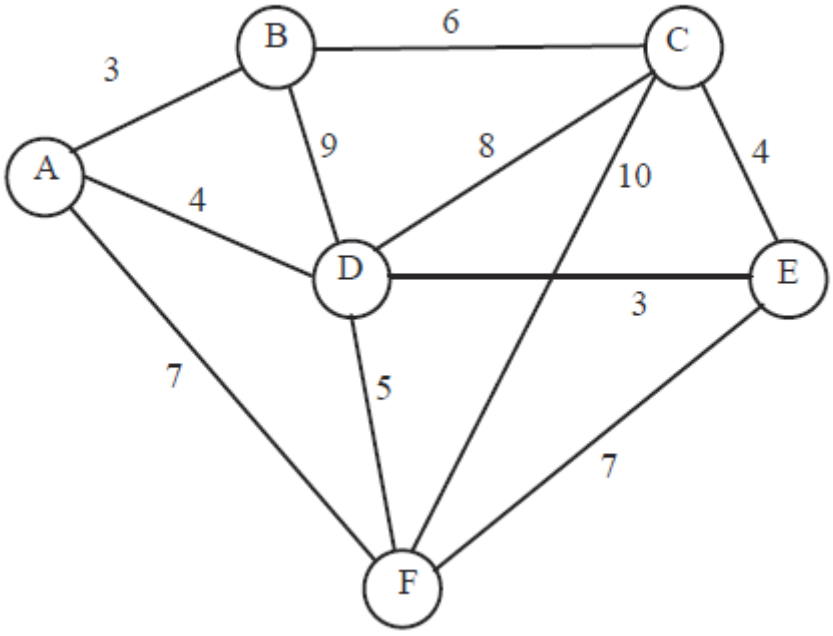
Ein eleganter Lösungsansatz setzt Rekursion / Backtracking ein, führt jedoch zu exponentieller Komplexität.
Das Verfahren kann man allerdings optimieren, indem Teilergebnisse geschickt vorausberechnet und in einer Tabelle gespeichert werden, um Wiederholungen von gleichen Berechnungen zu vermeiden (sog. „*dynamisches Programmieren*“).
Damit erreicht man eine polynomiale Komplexität ➔

⇨ das Rucksackproblem gehört somit zur Klasse P.

Das Problem des Handelsreisenden („traveling salesman problem“, oder kurz „TSP“)

Eingabe: ein Graph mit „Städten“ als Knoten und „Straßen“ als Kanten. Jede Straße zwischen zwei Städten ist mit einer (natürlichen) Zahl markiert, die die Entfernung der Städte voneinander angibt.
Ausgabe: ein Plan für die kürzeste „Rundreise“, die bei einer beliebigen Stadt anfängt und bei derselben Stadt endet und genau einmal bei jeder anderen Stadt vorbeiführt.

Beispiel:



Ausgabe: A, B, C, E, D, F, A, Gesamtlänge: 28

Es kann gezeigt werden, dass es sich hier um ein NP-vollständiges (NP-C) Problem handelt, das somit zur Klasse NP gehört.

Das Erfüllbarkeitsproblem

- Problem der Klasse NP
  - Gegeben: aussagenlogischer Ausdruck mit Variablen *a*, *b*, *c* etc., logischen Operatoren $\wedge$, $\vee$, $\neg$ und Klammern
  - Gesucht: Algorithmus der prüft, ob Formel erfüllbar ist, d.h., ob Belegung der Variablen mit booleschen Werten `true` und `false` existiert, sodass Formel `true` liefert

Beispiel in der Java-Notation:

```
a  &&  (b  ||  !a)  &&  !b
```

Für alle vier mögliche Belegungen der Variablen **a** und **b** ergibt der obige Ausdruck **false** ➔ die Formel ist nicht erfüllbar.

Bei diesem Problem ist unüblicher Weise auch die exponentielle Lösung durchaus brauchbar, weil z.B. bei 20 Variablen nur ca. 1 Million (2^{20}) Alternativen geprüft werden muss, wobei in praktischen Situationen Formeln mit mehr als 20 Variablen eher selten vorkommen.

Repetition

Aufgabe R1

Zum Speichern von ganzzahligen Werten zwischen **0** und **99** soll eine Hashtabelle benutzt werden. Die Hashtabelle soll **17** Plätze (Indizes **0** - **16**, siehe unten) haben.
Als Hashfunktion wird

h(x) = x mod 17 (d.h. **h(x)** ist der Rest der Division von **x** durch **17**)

eingesetzt.
Kollisionen werden mit linearem Sondieren gehandhabt.

a.
Tragen Sie in die untenstehende Tabelle das Ergebnis der Speicherung von Werten

7, 83, 50, 49, 74, 23

in dieser Reihenfolge (von links nach rechts) ein.

Index	Eintrag
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	

b.
Jemand schlägt vor, die oben festgelegte Hashfunktion **h(x)** durch

h‘(x) = (Quersumme von x) mod 17

oder durch

h‘‘ (x) = (x * 1024) mod 14

zu ersetzen.
(Quersumme ist die Summe der beiden Ziffern, z.B. *Quersumme von 36 = 3 + 6 = 9*).
Welchen Einfluss hätte jede von diesen Änderungen auf die durchschnittlichen Zeitaufwände für die Speicherung und das Suchen eines Eintrages? Begründen Sie Ihre Antwort.

Aufgabe R2

Gegeben ist die Methode **methodeR2** :

```
public static void methodeR2(int n)  {
    if (n < 2)
        return;

    etwas();
    etwas();

    if (n % 2 > 0)
        etwas();

    methodeR2(n / 2);
}
```

Die Laufzeit von **methodeR2** ist durch die Zahl von **etwas()** – Aufrufen bestimmt. Geben Sie eine obere Schranke (= *worst case* - Schätzung) für die Zahl dieser Aufrufe an.
Welche Komplexitätsklasse (in Landau-Notation) ergibt sich aus dieser Schätzung?

Aufgabe R3

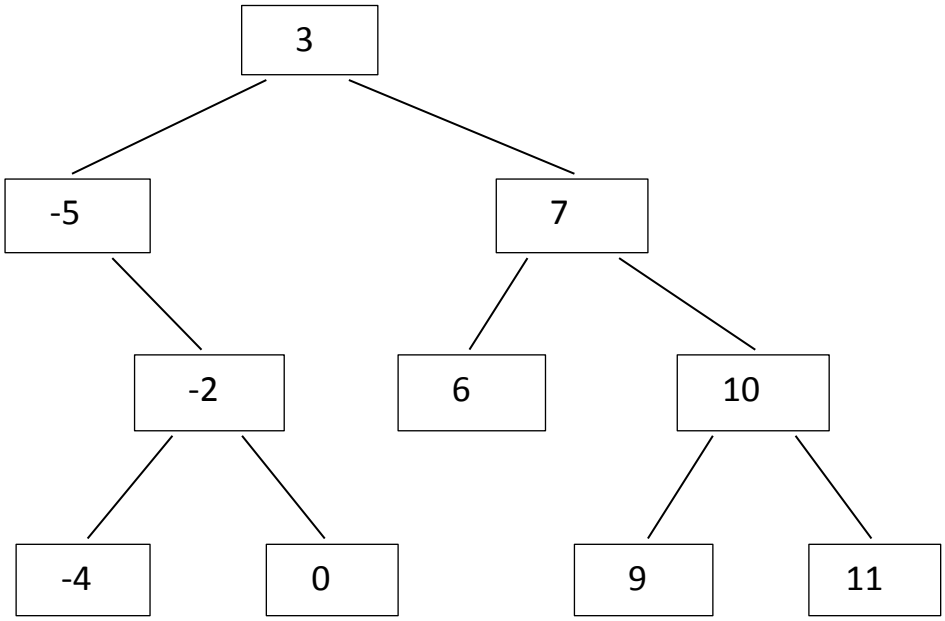
Die Java-Klasse *Portfolio* stellt die Bestände eines Wertschriften-Portfolios in Form einer doppelt verketteten Liste dar, welche die Klasse *Item* für die Knoten und die Klasse *Asset* für die Dateninhalte der Knoten verwendet (siehe unten). Sie können annehmen, dass die Methode *splice* gemäss Beschreibung im Kommentar korrekt implementiert ist.
Vervollständigen Sie die Methode *bigAssetsFirst* der Klasse *Portfolio*, welche Bestände mit dem Marktwert > 1000 CHF zum Anfang der Liste verschiebt (siehe Beschreibung im Kommentar).

```
public class Asset {
    String assetId;                  // Wertschrift-Id.
    String buyDate;                  // Kaufdatum: Format ‚yyyymmdd'
    int value;                       // Marktwert: CHF ohne Rappen

    // Ermittelt den Marktwert
    public int getValue() {
        return value;
    }

      ... Konstruktoren und übrige Methoden hier ausgelassen
}
```

```
// Ein Knoten der Liste
public class Item {
    Asset e;                          // Element (= Dateninhalt)
    Item next;                        // nächster Knoten
    Item prev;                        // vorheriger Knoten

    public Item getNext() {
        return next;
    }

    public Item getPrev() {
        return prev;
    }

    public Asset getAsset() {
        return e;
    }
}


public class Portfolio {
     private Item head;               /* Dummy-Knoten, beinhaltet
                                         keine realen Daten
                                         eines Bestandes. Zeigt
                                         vorwärts auf den ersten,
                                         rückwärts auf den letzten
                                         Bestand. */


      /* Entferne die zusammenhängende Teilliste a, … ,b aus der
         Liste und füge sie hinter den Zielknoten t ein.
         assert : b steht nicht vor a und t ist kein Glied der
         Teilliste a, … ,b .
      */

      public void splice(Item a, Item b, Item t) {
            // Die Implementierung hier ausgelassen
      }


      /* Die Methode  bigAssetsFirst  verschiebt die Bestände mit
         Marktwert > 1000 CHF zum Anfang der Liste, damit sie vor den
         übrigen Einträgen positioniert sind.
         In jeder von den so entstandenen zwei Gruppen können die
         Bestände eine beliebige, unsortierte Reihenfolge haben.
         Rückgabewert: Anzahl der Bestände mit Marktwert > 1000 CHF.
      */

      public int bigAssetsFirst() {
            // Ihre Implementierung:
      }

      ... Konstruktoren und übrige Methoden hier ausgelassen
}
```

Aufgabe R4

Gegeben ist der folgende binäre Baum:



Zur Darstellung von Baumknoten wir die folgende Klasse verwendet:

```
public  class  Node {
    int    number;                    // Schlüssel
    Node   leftChild;
    Node   rightChild;

    public Node getLeftChild() {
        return leftChild;
    }

    public Node getRightChild() {
        return rightChild;
    }

    public int getKey() {
        return number;
    }

        ... Konstruktoren und übrige Methoden hier ausgelassen
}
```

a.
Implementieren Sie in Java die Methode *traversiereDFS* mit der Signatur

public static void traversiereDFS(Node n)

, welche den Baum gemäss dem *depth-first* – Schema (DFS) traversiert, wenn man sie mit der Wurzel des Baums als Parameter aufruft, wobei sie bei jedem besuchten Knoten seinen Schlüssel am Bildschirm ausgibt (ein Schlüssel pro Zeile).
Schreiben Sie die damit erzeugte Bildschirmausgabe auf.

b.
Lösen Sie gleiche Aufgabe wie in **a.** mit der Methode *traversiereBFS* (Signatur analog), welche den obigen Baum gemäss dem *breadth-first* – Schema (BFS) verarbeitet.
Benutzen Sie dazu die folgende Implementierung einer Warteschlange:

```
public  class  Queue_R4 {

    public Queue_R4() {..}              // erzeugt leere Warteschlange

    public  void pushBack(Node n) {..} // nimmt einen Knoten in =>
                                        // => die Warteschlange auf

    public  boolean isEmpty() {..}     // prüft, ob die =>
                                       // => Warteschlange leer ist

    public  Node  first() {..}         // gibt den vordersten =>
                                       // => Knoten zurück

    public  void  popFront() {..}      // entfernt den vordersten =>
                                       // => Knoten

        ... Datenfelder und übrige Methoden hier ausgelassen
}
```

Aufgabe R5

a.
Die Klasse **R5** ist wie folgt implementiert:

```
public class R5 {

    // Konstruktor
    public R5() { }

    public static double pow(double x, int m) {
        if (m < 1)
            return 1;
        else if (m == 1)
            return x;

        return x * pow(x, m - 1);
    }

    public static double geomReihe(double a, double q, int n) {
        double s = 0.0;

        for (int i = 0; i < n; i++)
            s = s + pow(q, i);

        return a * s;
    }
}
```

Welche Laufzeitkomplexität in Landau-Notation bezüglich Abhängigkeit vom Parameter **n** hat die Methode **geomReihe()**?

b.

In einem Betrieb sind 4 verschiedene Anwendungen A, B, C, D im Einsatz, die täglich Datenbestände von 32 Kunden auswerten. Das Datenvolumen pro Kunde ist für alle Kunden ca. gleich. Die Anwendungen werden auf einem Rechner des Typs Ra ausgeführt.
Die untenstehende Tabelle beinhaltet Laufzeitkomplexitäten der obigen Anwendungen bezüglich Anzahl Kunden **n** und die effektiven Laufzeiten für 32 Kunden auf dem Rechner Ra.
Infolge einer Umstrukturierung wird der Rechner Ra aus dem Betrieb genommen und die zu verarbeitende Datenmenge wird gleichmässig auf zwei Rechner des Typs Rb verteilt (d.h. 16 Kunden pro Rb-Rechner). Die zwei Rb-Rechner führen jede Anwendung parallel (zu gleicher Zeit) aus. Dabei ist ein Rechner des Typs Rb viermal schneller als ein Rechner des Typs Ra (Rb führt eine vierfache Anzahl von Programmanweisungen pro Sekunde aus als Ra).
Tragen Sie in die Tabelle die zu erwartenden Ausführungszeiten für zwei parallel laufenden Rb-Rechner ein.

Anwendung	**A**	**B**	**C**	**D**
Laufzeitkomplexität bezüglich Anzahl Kunden **n**	**$\log_2 n$**	**$\log_2 n \cdot n$**	**n**	**n^2**
Laufzeit auf einem Ra-Rechner (Minuten)	**40**	**120**	**80**	**320**
Laufzeit auf zwei Rb-Rechnern (Minuten)				

Aufgabe R6

Ein kompletter binärer Baum ist über dem Array

int[] halde = { 21, 19, 9, 15, 7, 3 } ;

aufgebaut.

a.
Zeichnen Sie den Baum.
Wie können Sie sicherstellen, dass dieser Baum eine Halde darstellt?

b.
Ordnen Sie die Halde aufsteigend mit dem standardmässigen HeapSort - Verfahren. Notieren Sie untenstehend (von oben nach unten in der Reihenfolge der Ausführung) alle Zwischenzustände des Arrays, die nach einzelnen Schritten entstehen (1 Schritt = jeder Zuwachs der sortierten Folge samt Wiederherstellung der verbleibenden Halde).

21	19	9	15	7	3

Aufgabe R7

Gegeben ist eine unsortierte Datei **I** mit einer Anzahl **N** von Datensätzen. Die maximale Kapazität des vorhandenen Arbeitsspeichers **n** beträgt nur einen Bruchteil von **N**, folglich kann **I** nicht vollständig in den Arbeitsspeicher des Rechners eingelesen werden.
Mit dem unten beschriebenen Sortierkanal-Verfahren können aus **I** sortierte Sequenzen von ca. **2n** Datensätzen extrahiert und als Dateien abgelegt werden.
Die Erstellung einer solchen Sequenz („Sortierkanal-Lauf“) erfolgt in folgenden Schritten:

- Mit den ersten **n** Datensätzen der Input-Datei **I** wird ein Heap **H1** erstellt
- Der nächste Datensatz **D** wird von **I** gelesen
    - **D >** Wurzel von **H1** ➔ die Wurzel wird in die aufgebaute Output-Sequenz **S** verschoben und **D** ins Heap **H1** versenkt
    - **D <=** Wurzel von **H1** aber **D >=** der zuletzt geschriebene Datensatz der Sequenz **S** oder **S** ist noch leer ➔ **D** wird direkt in die Sequenz verschoben
    - **D <** der zuletzt geschriebene Datensatz der Sequenz **S** ➔ die Wurzel wird in die Sequenz **S** verschoben und **D** in ein zweites Heap **H2** integriert, das im Arbeitsspeicher hinter dem (schrumpfenden) Heap **H1** wächst und für den nächsten Lauf bestimmt ist
- Wenn das Heap **H1** leer ist, ist der Lauf abgeschlossen

Wenn die Input-Datensätze in **I** eine rein zufällige Reihenfolge haben, wird die resultierende Datei **S** im Schnitt eine Länge von ca. **2n** Datensätzen haben. (Das als Nebenprodukt entstandene Heap **H2** kann bei Bedarf in einem „Folgelauf“ als **H1** eingesetzt werden).

a.
Beschreiben Sie das obige Verfahren in Pseudocode als *Procedure Sortierkanal_Lauf* . Nehmen Sie an, dass die Datensätze Textzeilen sind, die alphabetisch zu sortieren sind.

b.
Die unsortierte Input-Datei **I** beinhaltet **3n** Datensätze. Mit der obigen Prozedur *Sortierkanal_Lauf wurde* sie in eine sortierte Datei **S** (2n) und ein Heap **H2** im Arbeitsspeicher zerlegt.
Schreiben Sie in Pseudocode eine Prozedur *Mischen*, welche **S** und **H2** zu einer sortierten Output-Datei **O** zusammenfügt.

Aufgabe R8

Wie viele verschiedene binäre Suchbäume kann man aus vier Knoten mit den Schlüssel-Werten **1**, **2, 3** und **4** bilden? Welche von diesen Bäumen haben die bestmögliche Balancierung?

Aufgabe R9

Nehmen Sie an, dass die Prozedur *blubs* einen konstanten Zeitbedarf hat (d.h. jeder Aufruf von *blubs* benötigt für seine Ausführung gleich für Zeit, z.B. 37 Milisekunden).
Welche Zeitkomplexitäten haben die untenstehenden Methoden *proz1* - *proz9* ?

```
static void proz1(int n) {
    for (int i = 1; i <= n; i++)
        blubs();
}

static void proz2(int n) {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            blubs();
}

static void proz3(int n) {
    for (int i = 1; i <= n; i++)
        blubs();
    for (int i = 1; i <= n; i++)
        blubs();
}

static void proz4(int n) {
    for (int i = 1; i <= 100; i++)
        for (int j = 1; j <= n; j++)
            for (int k = 1; k <= 100; k++)
                blubs();
}

static void proz5(int n) {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++)
            blubs();
}

static void proz6(int n) {
    for (int i = 1; i <= n/2; i++)
        for (int j = 1; j <= n/4; j++)
            for (int k = 1; k <= n/8; k++)
                blubs();
}

static void proz7(int n) {
    for (int i = 1; i <= n * n; i++)
        for (int j = 1; j <= n * n * n; j++)
            blubs();
}

static void proz8(int n) {
    blubs();
    if (n > 1)
        proz8(n-1);
}
```

```
static void proz9(int n) {
    blubs();
    if (n > 1) {
        proz9(n-1);
        proz9(n-1);
    }
}
```

Lösungen

Aufgabe R1

a.

Index	Eintrag
0	49
1	
2	
3	
4	
5	
6	74
7	7
8	23
9	
10	
11	
12	
13	
14	
15	83
16	50

b.
Beim Einsatz der Hashfunktion **h(x) = x mod 17** gibt es für jeden Tabellenplatz entweder 5 oder 6 mögliche zu speichernden Werte, die Kollisionen sind somit gleichmässig verteilt.

Mit **h'(x) = (Quersumme von x) mod 17** wird z.B. der Platz mit Index 9 zehnfach belegt (9, 18, 27 … 90), wobei es für den Platz mit Index 0 nur drei mögliche Werte gibt (0, 89, 98).

Mit **h''(x) = (x * 1024) mod 14** werden die Plätze mit Indizes 14 – 16 nicht benutzt und von den restlichen Plätzen können nur die geraden Indizes belegt werden.

Die Kollisionen sind somit in beiden Fällen offensichtlich weniger gleichmässig verteilt. Demzufolge ist für das Speichern wie auch fürs Lesen im Durchschnitt ein höherer Zeitaufwand zu erwarten.

Aufgabe R2

Die Zahl **n** kann ca. $\mathbf{\log_2 n}$ - mal halbiert werden ==> daraus ergeben sich $\mathbf{\log_2 n}$ rekursive Aufrufe von **methodeR2** mit jeweils maximal 3 **etwas()** – Aufrufen.

Resultierende obere Schranke für **etwas()** – Aufrufe: $\mathbf{3 * \log_2 n}$
Entsprechende Laufzeitkomplexität: **O(log n)**

Aufgabe R3

```
public int bigAssetsFirst()
{
    int count = 0;
    Item checkedNode = head.getNext();

    while (checkedNode != head) {   // Referenzen-Vergleich
        Item nextNode = checkedNode.getNext();
        if (checkedNode.getAsset()!= null) {
            if (checkedNode.getAsset().getValue() > 1000) {
                splice(checkedNode, checkedNode, head);
                count++;
            }
        }
        checkedNode = nextNode;
    }

    return  count;
}
```

Aufgabe R4

a.

```
public static void traversiereDFS(Node n) {
    if (n == null)
        return;

    traversiereDFS(n.getLeftChild());
    traversiereDFS(n.getRightChild());
    System.out.println(n.getKey());
}
```

Bildschirmausgabe:

-4
0
-2
-5
6
9
11
10
7
3

b.

```
public static void traversiereDFS(Node n)  {
    if (n == null)
        return;
    Queue_R4 q = new Queue_R4(); // leere Warteschlange
    q.pushBack(n);               // Wurzel in die =>
                                 // => Warteschlange aufnehmen
    while (!q.isEmpty()) {
        Node n = q.first();      // den vordersten Knoten =>
                                 // => verarbeiten
        System.out.println(n.getKey());
        if (n.getLeftChild() != null)
            q.pushBack(n.getLeftChild()) // linkes Kind =>
                                 // => eintragen
        if (n.getRightChild() != null)
            q.pushBack(n.getRightChild()) // rechtes Kind =>
                                 // => eintragen

        q.popFront();            // den verarbeiteten =>
                                 // => Knoten entfernen
    }
}
```

Bildschirmausgabe:

```
3
-5
7
-2
6
10
-4
0
9
11
```

Aufgabe R5

a.

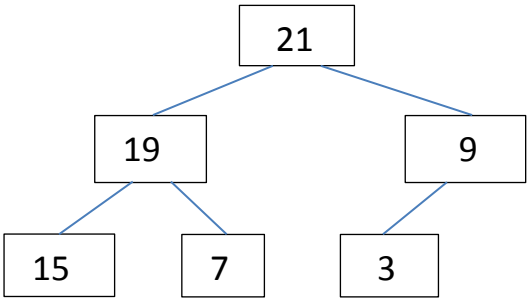
In der **for** – Schleife führt jeder **pow(q, i)** – Aufruf mit **1 < i < n** aufgrund der Rekursion **(i – 1)** Multiplikationen durch.
Somit führt die Methode **geomReihe(a, n)** insgesamt **(n – 2) * (n - 1) / 2 + 1** Basisschritte (Multiplikationen) durch und hat dementsprechend die Laufzeitkomplexität **$O(n^2)$** .

b.

Anwendung	**A**	**B**	**C**	**D**
Laufzeitkomplexität bezüglich Anzahl Kunden **n**	**$\log_2 n$**	**$\log_2 n \cdot n$**	**n**	**n^2**
Laufzeit auf einem Ra-Rechner (Minuten)	**40**	**120**	**80**	**320**
Laufzeit auf zwei Rb-Rechnern (Minuten)	**8**	**12**	**10**	**20**

Aufgabe R6

a.



Der Baum ist komplett und der Wert jedes Vaterknotens ist höher (oder gleich) als die Werte seiner Kinder => *max-Heap*.

b.

21	19	9	15	7	3
19	15	9	3	7	21
15	7	9	3	19	21
9	7	3	15	19	21
7	3	9	15	19	21
3	7	9	15	19	21

Aufgabe R7

a.

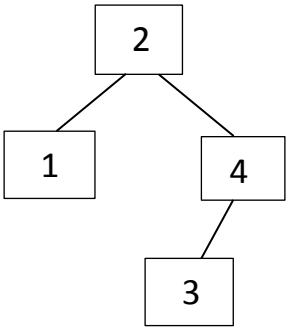
```
Procedure Sortierkanal_Lauf ( I, S : File; n : N; H2 : Heap ) // Input- und Output-Datei;  =>
                                                              // => Speicherkapazität;
                                                              // => Output-Heap (leer)
   Di : Array of Char                                         // Buffer für die aus I gelesene Zeile
   Ds : Array of Char                                         // Buffer für die in S zuletzt =>
                                                              // => geschriebene Zeile
   // Anmerkung: Im Folgenden bedeutet Di < Ds „Di alphabetisch vor Ds“,
   //                                  Ds <= Di „Ds alphabetisch vor oder gleich Di“ usw.

   H1 : Heap
   while Grösse H1 < n und Ende I nicht erreicht do        // H1 aus ersten n Input-Zeilen bilden
      Di := Nächste Zeile aus I
      Integriere Di in H1

   while H1 nicht leer und Ende I nicht erreicht do
      Di := Nächste Zeile aus I
      if Di > Wurzel H1 then
         Kopiere die Wurzel in Ds und entferne sie aus H1
         Füge Kopie von Ds ans Ende von S
         Reorganisiere H1 zum gültigen Heap
      else if Di <= Wurzel H1 und (Di >= Ds oder Ds leer) then
         Kopiere Di in Ds
         Füge Kopie von Ds ans Ende von S
      else                                                  // Di < Ds
         Kopiere Wurzel in Ds und entferne sie aus H1
         Füge Kopie von Ds ans Ende von S
         Reorganisiere H1 zum gültigen Heap
         Integriere Di in H2
   Schliesse S
```

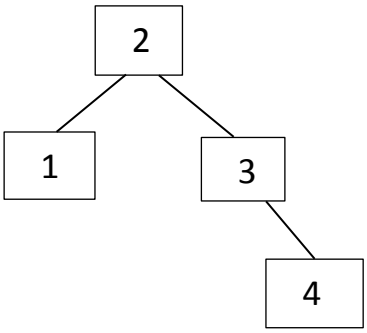
b.

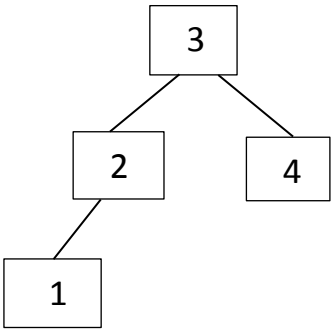
```
Procedure Mischen ( S, O : File; H2 : Heap )              // Input- und Output-Datei; Input-Heap
   Ds : Array of Char                                     // Buffer für die aus S gelesene Zeile
   S_fertig = false  : Boolean                            // true wenn S vollständig in O übertragen

   while H2 nicht leer oder Ende S_fertig = false do
      if H2 leer then                                     // nur S wird in O kopiert
         Ds := Nächste nicht in O kopierte Zeile aus S
         Füge Ds ans Ende von O
         if Ds die letzte Zeile von S then
            S_fertig := true
      else if S_fertig = true then                        // nur H2 wird in O übertragen
         Füge Wurzel ans Ende von O und entferne sie aus H2
         Reorganisiere H2 zum gültigen Heap
      else
         Ds := Nächste nicht in O kopierte Zeile aus S
         if Ds < Wurzel H2 then
            Füge Ds ans Ende von O
            if Ds die letzte Zeile von S then
               S_fertig := true
         else
            Füge Wurzel ans Ende von O und entferne sie aus H2
            Reorganisiere H2 zum gültigen Heap
   Schliesse O
```

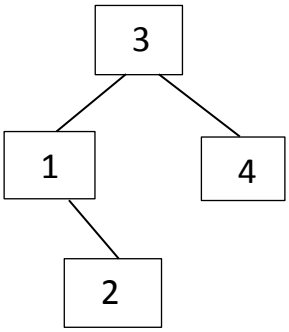
Aufgabe R8

Es können 14 verschiedene Bäume gebildet werden.
Davon haben die folgenden 4 die bestmögliche Balancierung:









Aufgabe R9

proz1 : **O(n)**
proz2 : **O(n^2)**
proz3 : **O(n)**
proz4 : **O(n)**
proz5 : **O(n^2)**
proz6 : **O(n^3)**
proz7 : **O(n^5)**
proz8 : **O(n)**
proz9 : **O(2^n)**