Hochschule Darmstadt

Zusammenfassung: 1. Semester

Algorithmen und Datenstrukturen

Leonhard Breuer
3. Januar 2024

Inhaltsverzeichnis

Kapitel 1

Der Begriff Algorithmus

Den Begriff Algorithmus gibt es bereits seit dem Mittelalter.

Das Konzept Algorithmus war bereits den alten Griechen bekannt.

1.1 Sieb des Erasthostenes

Das S.d.E. ermöglicht das Finden aller Primzahlen bis zu einer gewählten maximalen Zahl n.

Ablauf des Algorithmus ist wie folgt

1. Schreibe alle Zahlen von 1 bis n auf.
2. Für alle Zahlen i von 2 bis $\sqrt{n}$: Alle Vielfachen von i streichen.
3. Schreibe alle nicht gestrichenen Zahlen herraus.

1.2 Algorithmus des Euklied (ggT)

Der Algorthimus des Euklied dient zum Herausfinden des *größten gemeinsamen Teilers* zweier Zahlen.

Ablauf des Algorithmus ist wie folgt.
Ausgehend von den Zahlen m und m ¡ 0 sowie r welche den Rest der (ganzzahligen) Division repräsentiert.

1. Teile m durch n mit Rest r (es ergibt sich $r \geq n$).
2. Wenn $r = 0$, fertig mit Ergebnis n.
3. Ersetze m mit n und n mit r und mache mit *1.* weiter.

1.3 Algorism

Das Wort *Algorism* beschreibt das Addieren im Stellenwertsystem.

Ablauf des Algorithmus ist wie folgt

1. Schreibe die zu addierenden Zahlen rechtsbündig untereinenander.
2. Beginne ganz rechts mit der ersten Zahl.
3. Addiere die über dem Strich stehenden Ziffern der aktuellen Stelle zusammen.
4. Schreibe das Ergebnis unter den Strich
5. Setze diese Operationen für alle restlichen Stellen (beider Zahlen) fort.

Kapitel 2

Graphen und Bäume

Graphen und Bäume werden in der Informatik oftmals zur Beschreibung und Vereinfachung von Problemen benötigt.

Ein Graph besteht aus `Knoten (nodes)` und `Kanten (edges)`.

Knoten bilden die Menge `V` von Objekten v.

Kanten bilden die Menge `E` von Tupeln $e = (v, w)$ mit $v, w \in V$.

2.1 Graphen

2.1.1 Pfade

Man unterscheidet zwischen `offenen` und `geschlossenen` Pfaden.

offener Pfad Start- und Endpunkt sind unterschiedlich

geschlossener Pfad Führt am Ende an der Startpunkte zurück (Schleife)

2.1.2 Annotationen

Annotationen können anliegen, an

- `Knoten`: Als Knotengewicht
- `Kanten`: Als Kantengewicht

Annotationen, seltener Gewichtungen, ermöglichen es Algorithmen, bzw. den kürzesten Weg zu ermitteln.

2.1.3 Grad eines Knotens

Bei gerichteten Kanten unterscheidet man Aus- und Eingangsgrad.

Bei ungerichteten Kanten spricht man nur vom "Grad".

Vereinfacht stellt der "Gradëines Knoten die Anzahl der ein-/ausgehenden Verbindungen des Knoten(im Falle eines gerichteten Graphen) und der Gesamtanzahl der Verbindungen des Knoten (im Falle eines ungerichteten Graphens) dar.

2.1.4 Gerichtete Graphen

- Alle Pfade sind *offen*.
- auch *Digraph* oder *directed graph*

2.1.5 Gewichtete Graphen

Ein Graph ist gewichtet, sobald er über Annotationen verfügt.

2.1.6 Dependenzgraph (DAG)

Bei einem Dependenzgraphen beschreiben die Kanten, welche Tätigkeiten/Aufgaben vor einer anderen erfüllt werden müssen. Enthält der gerichtete Graph hierbei keine Zyklen, so spricht man vom *directed acyclic graph* oder kurz *DAG*.

2.1.7 Ungerichteter Graph / Cliquen

Eine *ungerichtete* Kante verhält sich wie zwei gerichtete Kanten. (eine hin und eine andere zurück)

Zusammenhang Wenn von einem Element aus, alle anderen Elemente (Knoten) in einem Graphen erreichbar sind, dann spricht man von einem zusammenhängenden Graphen. Gruppen, welche zwischen sich verbunden aber nicht mit anderen Gruppen verbunden sind, nennt man *Zusammenhangskomponenten*

2.1.8 Multigraph

Gibt es mehrere Kanten zwischen zwei Knoten (sowohl gerichtet als auch ungerichtet), so spricht man von einem Multigraph.

2.2 Bäume

Ein Baum ist *zusammenhängender ungerichteter* Graph ohne *geschlossenen* Pfad.

2.2.1 Knoten und Blätter

Knoten mit Grad 1 werden als Blatt (oder engl. Leaf) bezeichnet.

Andere Knoten werden als innere Knoten bzw. *inner leaf* bezeichnet.

Elter(n) ist der übergeordnete Knoten

Kind ist der untergeordnete Knoten

2.2.2 Bäume - Höhe, Grad

Grad Der Grad eines Baumes ist immer der höchste Grad eines Knotens.

Höhe maximale Pfadlänge

2.3 Kontrollstrukturen

Sequenz Dinge nacheinander ausführen.

Alternative Bedingte Befehlsausführung (if/else; switch; ternary)

Schleifen (do/while; while; for)

Funktionen

Sprünge nicht verwendet (goto)

Kapitel 3

Felder und Strings

3.1 Felder

- feststehender Datentyp
- konstankte Größe

3.1.1 Speicherung

Einträge in Felder werden immer nur im jeweiligen Datentyp gemacht. Die Größe eines Arrays beträgt also immer

$$size_{array} = count \times size_{data\_type}$$

3.1.2 Namenskonventionen

i repräsentiert den Index (bei mehreren Schleifen ineinander auch j oder k)

n repräsentiert die Größe (selten auch m, *n1* oder *n2*)

3.1.3 Array vs. Vector

Während bei einem *Array* die Größe bereits zur Compiletime feststehen muss, kann die Größe eines *Vectors* auch noch zur Runtime geändert werden.

3.1.4 Mehrdimensionale Felder

Ein mehrdimensionales Feld enthält mehrere (meist eindimensionale) Felder.

3.2 Strings

ASCII *(American Standard Code for Information Interchange)* 32-127 Zeichen (7bit)

ISO-8859 enthält alle *ASCII*-Zeichen (0-127) und weitere Zeichen (128-256) anderer Sprachen.

UTF-8 enthält alle *ASCII*-Zeichen (0-127) und weitere Zeichen (128 - ...) anderer Sprachen.

3.2.1 Besonderheit von UTF-8

Die Besonderheit von *UTF-8* gegenüber *ISO-8859* liegt in der variablen Speicherlänge der einzelnen Zeichen. Die meisten Zeichen (des normalen Alphabets) besitzen eine Speicherlänge von einem Bit. Besondere Zeichen wie die deutschen Umlaute (*ä, ü, ö*) besitzen eine Speicherlänge von 2 Bit.

UTF-8 ermöglicht es einem Zeichen eine Länge von 1 - 4 bit zu haben.

3.2.2 Nutzung in C++

char* einfach...

- Zeichenkette solange bis eine *0* kommt.
- Pro String nur ein Byte.
- beliebige Länge
- Die meisten Betriebssysteme liefern Strings/Werte in diesem Format.

... aber echt unpraktisch.

std::string *std::string* bietet eine Speicherkapselung, was die manuelle Speicherverwaltung überflüssig macht.

- Implementierung abhängig von der Laufzeitumgebung.
- Häufig: Angelegt auf Heap -¿ Adresse durch Variable gehalten.

- sizeof(int) = 4
- sizeof(std::string) = 32

Der String ermöglicht also Zeichenketten von ca. 32 Zeichen ohne zusätzlichen Platz auf dem Heap.

Kapitel 4

Iteration und Rekursion

Iteration bezieht sich auf die Wiederholung durch Aneinanderreihung mithilfe von Schleifen.

Rekursion bezieht sich auf die Wiederholung durch Ineinanderschachtelung mithilfe von Funktionen.

Beispiel

Iteration mit einer for-Schleife

```
void f1(int n, int m){
for (int i=n; i<=m; i++){
cout << i << endl;
}
}
```

Rekursion mit einer Funktion

```
void f6 (int n, int m){
if (m<n) return;
f6(n,m-1);
cout << m << endl;
}
```

Es gibt hier kein klares besseres Element. Beide haben ihre Daseinsberechtigung. Eine "bessere"Lösung mit einem der beiden Arten ist immer von dem vorliegenden Problem abhängig.

Kapitel 5

O-Notation

5.1 Union-Find

Der Union-Find Algorithmus ist ein äußerst vielseitiger Algorithmus.

5.1.1 Anwendungsbeispiele

- Pixel auf einem Foto
- Freunde in Sozialen Netzwerken
- Computer in einem Netzwerk
- Elemente in einer mathematischen Menge

5.1.2 Funktionen

find Diese Funktion prüft, ob sich zwei Elemente in der selben Zusammenhangskomponente befinden bzw. es einen Weg von der einen zur anderen gibt.

unify Diese Funktion ersetzt die Komponenten zweier Elemente mit ihrer Vereinigungsmenge bzw. verknüpgt zwei Elemente.

5.1.3 Abwandlungen des Union-Find

Quick-Find

Basis Die Basis für den *Quick-Find* macht ein Feld **id**[] mit Größe n und Datentypen *int*. Zwei Objekte sind dann verbunden, wenn sie diesselbe ID haben.

Geschwindigkeit Man stellt fest, dass der Quick-Find-Algorithmus zulangsam ist.

Algorithmus	**init**	**unify**	**find**
Quick-Find	N	N	1

Die *unify*-Operation vom Quick-Find ist zu langsam. Da die Laufzeit bei N Objekten N^2 beträgt.

Quick-Union

Basis Die Basis ist identisch zum *Quick-Find*. id[i] ist *Elter* von i - Wurzel i ist $id[id[id[...id[i]...]]]$.

Geschwindigkeit Auch der Quick-Union hat keine großen Vorteile im Vergleich zum Quick-Union, auch wenn er häufig schneller ist.

Algorithmus	**init**	**unify**	**find**
Quick-Find	N	N (inkl. Wurzel suche)	N

Weighted Quick Union Ein Problem des Quick Union waren die zu groß werdenden Bäume. Dies soll mit dem Gewichten verbessert werden.

Geschwindigkeit Die Geschwindigkeit des Weighted Quick Union ist deutlich besser als die beiden anderen Implementierungen.

Algorithmus	**init**	**unify**	**find**
Weighted Quick Union	N	ld N (inkl. Wurzel suche)	ld N

Quick-Union mit Pfadkompression

Basis/Idee Nach dem Fund der Wurzel, werden alle besuchten Knoten direkt an die Wurzel angehängt.

Geschwindigkeit Die Auswertung der Laufzeitkomplexität von Weighted Quick Union mit Pfadkompression ist zu kompliziert. Laut Hopcraft-Ulman und Tarjan benötigt man **M** Union-Find-Operationen auf **N**-Elemente weniger als c(N + M × N) Feldzugriffe. Das führt zu einer beinahe linearen Laufzeitkomplexität.

5.1.4 Zusammenfassung Union-Find-Algorithmen

Die Algorithmen folgende Laufzeitkomplexität (im Worst Case) auf:

Algorithmus	Laufzeitkomplexität (Worst Case)
Quick Find	M × N
Quick Union	M × N
Weighted Quick Union	N + M × *log* N
Quick Union (Pfadkompression)	N + M × *log* N
Weigthed Quick Union (Pfadkompression)	N + M × *ld* N

5.2 Skalierung quadratischer Probleme

Die Skalierung von Quadratischen Problemen ist sehr schwer. Mit stetiger Evolution von Computer steigen auch die Anforderungen an diese. Die zu lösenden Problematiken werden immer größer, komplexer und somit letztendlich auch rechenaufwendiger.
Die einzige Lösung für diese Problematik:

Effizientere Algorithmen

5.3 Komplexitätstheorie

Die zentrale Frage der Komplexitätstheorie lautet

$$P = NP$$

P Klasse von Problemen, die praktisch lösbar sind. (deterministisch & polynomiell)

NP Klasse von Problemen, die praktisch überprüfbar sind. (nicht deterministisch & polynomiell)

5.3.1 O-Notation

Die *O-Notation* beschreibt das Verhalten von Algorithmen in Richtung unendlich.
O(g(n)) ist eine Menge, die alle FUnktionen enthält, die unendlich von g durch Multiplikation mit einer Konstanten c übertrumpft werden.

Einige Rechenregeln mit der O-Notation

- $f(n) = O(f(n))$
- $O(c \times f(n)) = O(f(n))$
- $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$

- $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$
- $f(n) \times O(g(n)) = O(f(n) \times g(n))$

Addition Bei der Addition der O-Notation (dem Ausführen einer Aufgabe vor einer anderen) spielt nur die komplexere Aufgabe eine Rolle. Beispiel:
$O(logN) + O(N) = O(N)$
$O(N) + O(N^2) = O(N^2)$

Multiplizieren Im Gegensatz zum Addieren, erhöht die Multiplikation die Gesamtkomplexität.

Das lässt sich sehr einfach am Beispiel einer Schleife darstellen. Der Schleifenkörper hat eine Komplexität $O(g(n))$ und wird $f(n)$-Mal ausgeführt. Das erhöt den Gesamtaufwand zum Produkt $O(g(n)\dot{f}(n))$

```
for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
                ... irgendeine Operation mit O(1) ...
        }
}
```

- Der Schleifenkörper der inneren Schleife wird n mal ausgeführt: $n\dot{O}(1) = O(n)$
- Der Schleifenkörper der äußeren Schleife wird n mal ausgeführt: $n\dot{O}(n) = O(n\dot{n}) = O(n^2)$

Limitierungen Die getroffenen Entscheidungen und darausfolgenden Vereinfachungen haben jedoch einige Nachteile:

- Algorithmen gleicher Komplexitätsklasse sind nicht unbedingt gleich schnell.
- Ein Algorithmus der in der Theorie schneller ist, muss nicht in der Praxis gleichschnell sein.

Kapitel 6

Liste, Heap, Stack und Queue

6.1 Liste

Listen finden sich überall im täglichen Leben.

6.1.1 Liste als Datenstruktur

Es gibt mehrere Möglichkeiten zur Verwaltung von Listen

- In einem Array oder *vector*
- Als verkettete Liste

6.1.2 Liste als Array

Vorteile:

- Einfach zu Implementierungen
- Zugriff auf n-tes Element mit O(1)
- Prüfung leerer Liste O(1)
- Einfügen und Löschen am Ende O(1)

Nachteile:

- Einfügen am Anfang und zwischendrin: O(N)
- Löschen am Anfang und zwischendrin aufwändig O(N)
- Maximaler Platz muss bekannt sein, Speicherschwendung. (Abhilfe durch Vector)

6.1.3 Liste als verkettete Liste

Eine verkettete Liste besteht aus der Information selbst und einem Zeiger auf das nächste Element in der Liste.
Beispielcode:

```
struct node {
        string data;
        node* next;
}
```

Das letzte Element der Liste erhält einen *nullptr* als Zeiger auf das "nächsteËlement.

Operation	Als Array	Als verkettete Liste
Element einfügen am Anfang	$O(N)$	$O(1)$
Element einfügen am Ende	$O(1)$	$O(N)$
Element einfügen an n-ter Stelle	$O(N)$	$O(1)$ (Suche: $O(N)$)
Element löschen am Anfang	$O(N)$	$O(1)$
Element löschen am Ende	$O(1)$	$O(N)$
Element löschen an n-ter Stelle	$O(N)$	$O(1)$ (Suche: $O(N)$)
Element am Anfang	$O(1)$	$O(1)$
Element am Ende	$O(1)$	$O(N)$
Element an n-ter Stelle	$O(1)$	$O(N)$
Element (nach Inhalt) suchen	$O(N)$	$O(N)$
Element suchen und löschen	$O(N)$	$O(N)$
Anzahl der Elemente ermitteln	$O(1)$	$O(N)$
Leere Liste erkennen	$O(1)$	$O(1)$
Liste vorne anhängen	$O(N+M)$	$O(M)$
Liste hinten anhängen	$O(M)$	$O(N)$

Tabelle 6.1: Vergleich von Operationen zwischen Arrays und verketteten Listen

doppelt verkette Listen Das doppelte Verketten (Endzeiger und Zähler) von Listen ermöglicht für fast alle Operationen auf Listen **O(1)**.

6.2 Stack

Der Stack (Stapel) ist eine Datenstruktur, die dem LIFO (last in, first out) Prinzip folgt. Beispiel: Das zuletzt hinzugefügte Element wird als erstes wieder entfernt.

6.2.1 Anwendungen

Stacks werden verwendet um den Programmfluss, Funktionsaufrufe und lokale Variablen effiziernt zu verwalten. z.B. Bei der Rekursion.

6.2.2 Operationen

push(x) legt das Element x auf den Stack.

x=pop() holt das oberste Element vom Stack.

Es gibt aber noch die Methoden *peek*, *top*, welche dem Aufruf von *x=pop(); push(x)*, also dem Herausheben eines Elements, entsprechen. Häufig kommt noch die Methode *empty* hinzu, welche den Stack auf seinen Inhalt überprüft (Vermeidung von Laufzeitfehlern.)

6.2.3 Geschwindigkeit

Alle Operationen an einem Stack benötigen *O(1)*.

6.2.4 Zusammenhang mit Rekursion

Stapel (Stacks) werden oft in der Informatik verwendet, um den Aufruf von Funktionen und die Verwaltung von lokalen Variablen während der Rekursion zu unterstützen, da die letzte aufgerufene Funktion immer als erste zurückkehrt, was dem Last-In-First-Out (LIFO)-Prinzip entspricht. Dies ermöglicht die effiziente Verwaltung von Funktionsaufrufen und Rückgabewerten.

6.3 Queue

Eine Queue (Schlange) folgt dem FIFO (first in, first out) Prinzip.

6.3.1 Anwendungen

Einfache Anwendungen von Queues finden sich in den meisten Algorithmen. Sie werden häufig als Datenpuffer / Zwischenspeicher verwendet, z.B. wenn die Daten momentan nicht verarbeitet werden können, jedoch zu einem späteren Zeitpunkt in der selben Reihenfolge wieder abgerufen werden müssen.

6.3.2 Operationen

enqueue(x) Hinzufügen zur Queue (auch *push(x)*).

x = dequeue() Entnehmen aus der Queue (auch $x = pop()$).

Auch hier gibt es mehere weitere Operationen, wie z.B. *empty* mit gleicher Funktionsweise wie bei dem Stack. Oder auch *front*, welche das nächste Element liefert, ohne es zu entnehmen.

6.3.3 Geschwindigkeit

Alle Operationen an einer Queue benötigen *O(1)*.

6.4 Heap

Der Heap dient auch als Zwischenspeicher.

6.4.1 Anwendungen

Der Heap wird genutzt um Platz für Daten zu bieten, deren Größe sich zur Laufzeit ändern kann.

6.4.2 Operationen

push_heap(x) packt ein Element auf den Heap.

x = pop_heap() nimmt ein Element vom Heap.

6.4.3 Geschwindigkeit

Die Geschwindigkeit der Operationen an einem Heap sind O(log N).

Kapitel 7

Sortier- und Suchalgorithmen

Das Sortieren stellt eine der ältesten Anwendungen für Computer dar.

7.1 Eigenschaften von Sortieralgorithmen

Ein Sortieralgorithmus ist ...

natürlich , wenn eine bereits (fast) sortierte Liste schneller fertig ist, als eine komplett unsortierte.

stabil , wenn Reihenfolge der Elemente bleibt bei gleichem Schlüsselwert identisch.

in place / in situ , wenn die Speichergröße konstant bleibt.

extern , wenn die Daten nicht im Arbeitsspeicher sondern auf Band oder Platte vorliegen.

7.2 Selectionssort

7.2.1 Idee

- größte Karte → ganz rechts.
- kleinste Karte → ganz links.

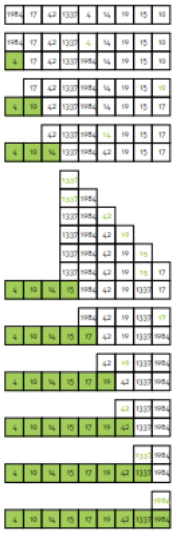
7.2.2 Algorithmus

Der Algorithmus vom *Selectionssort* läuft wie folgt:

1. Alle Elemente → Array

2. Kleinste Zahl suchen
   (a) Merke Zahl ganz links als **min** und aktuelle Position **pos**.
   (b) Prüfe ob **pos+1** kleiner als **min**. Wenn Ja, neue **min** und **pos**.
   (c) Mache weiter mit *2.a* bis Ende
3. Vertausche die Zahl an der Stelle **pos** mit der ersten Zahl und in den sortierten Bereich.
4. Mache mit 2 weiter, bis Ende.

7.2.3 Funktionsweise



7.2.4 Eigenschaften

natürlich ja.

stabil ja.

in place ja.

7.2.5 Laufzeitkomplexität

Schleifendurchläufe Es gibt $(n\dot{(}n-1))/2$ Schleifendurchläufe

Best-/Worstcase Bei der *Selectionssort* gibt es keinen Best-/Worstcase, da die innere Schleifen immer durchlaufen wird.

Komplexitätsklasse Die Komplexitätsklasse des *Selectionssort*-Algorithmus ist

$$O(\frac{(n\dot{(}n-1))}{2})$$
$$= O(\frac{n^2-n}{2})$$
$$= O(n^2)$$

7.3 Insertionsort

7.3.1 Idee

Bei der Grundidee für den Insertionssort gibt es 2 Varianten:

- Nimm nächste Karte → in Hand einsortieren von links ein (klein -¿ groß)
- Nimm nächste Karte → in Hand einsortieren von rechts ein (groß -¿ klein)

7.3.2 Algorithmus

1. Zu Beginn umfasst der unsortierte Bereich alle Elemente des Arrays bis auf eines. Das eine Element des sortierten Bereichs ist trivialerweise bereits sortiert.
2. Sortiere die ganz links stehende Zahl des unsortierten Bereichs in den bereits sortierten Bereich von rechts (große Zahlen) her ein wie folgt:
   (a) Tausche die neu einzusortierende Zahl immer weiter nach vorne, bis entweder
   (b) die Zahl links davon kleiner als die neu einzusortierende Zahl ist – der richtige Platz ist dann gefunden – oder
   (c) das Ende erreicht ist – dann ist die neu einzusortierende Zahl die kleinste bisher. (In beiden Fällen wandern dadurch alle Zahlen, die größer als die neu einzusortierende Zahl ist, um eine Position nach rechts.)
3. Mache mit Schritt 1 weiter, bis du am Ende bist.

7.3.3 Eigenschaften

natürlich ja

in place ja

stabil ja

7.3.4 Laufzeitkomplexität

Schleifenläufe Hier gibt es maximal $(n\dot{(}n-1))/2$ Schleifendurchläufe. Ein früheres Ende ist möglich.

Best-/Worstcase Ja, die innere Schleife wird garnicht durchlaufen. Von n bis $n\dot{(}n-1)/2$

Komplexitätsklasse Hier gibt es mehrere Klassen:
Best Case:

$$O(n)$$

Worst Case:

$$O(n^2)$$

7.4 TODO

Andere Sortieralgorithmen hinzufügen.

7.5 Binäre Suche

7.5.1 Idee

Schritt 0: Gehe zur Mitte der Liste

1. ist das Element genau dort, fertig.
2. ist das gesuchte Element kleiner als die Mitte, wiederhole für die vordere Hälfte.
3. ist das gesuchte Element größer als die Mitte, wiederhole für die hintere Hälfte.

Diese Art zu Suchen, nennt man **Divide'n'Conquer**. Nach spätestens **ld N** haben wir nur noch ein Element.