Programmiertechnik 2

Unit 5: Analyse von Algorithmen

Ablauf

- Vernetzungsprobleme – ein Beispiel
  - Quickfind, Quickunion
  - Laufzeitbetrachtung
- Empirische Analyse
- Mathematische Analyse
- Komplexitätsklassen
- O() Notation
- Lineare und binäre Suche

Verwenden vs. Implementieren

- Computerprogramme sind oft überoptimiert
  - Sorgfältige, einfache Implementierung vs. Effizienteste Implementierung
  - Wie häufig eingesetzt?
  - Aufgabengröße?
- Sorgfältiger Algorithmenentwurf essentiell bei großen Problemen
  - Beschleunigung um Faktor 10 oder 100 denkbar
  - → neueste, teuerste Hardware dagegen nur Faktor 2-4 schneller
- Probleme durch Zerlegung in Teilaufgaben lösen
  - Nach Zerlegung sind Teilaufgaben idR leichter zu lösen
  - Zerlegung ist nichttrivial
  - Wiederverwendbarkeit

Beispielproblem: Vernetzung

- Folge von Ganzzahlpaaren (p, q) – „p ist verbunden mit q“
- Geeignet zur Darstellung von Mengen, Graphen
- Problem: Herausfinden nicht-zuammenhängender Teilmengen

3-4	3-4	
4-9	4-9	
8-0	8-0	
2-3	2-3	
5-6	5-6	
2-9		2-3-4-9
5-9	5-9	
7-3	7-3	
4-8	4-8	
5-6		5-6
0-2		0-8-4-3-2
6-1	6-1	

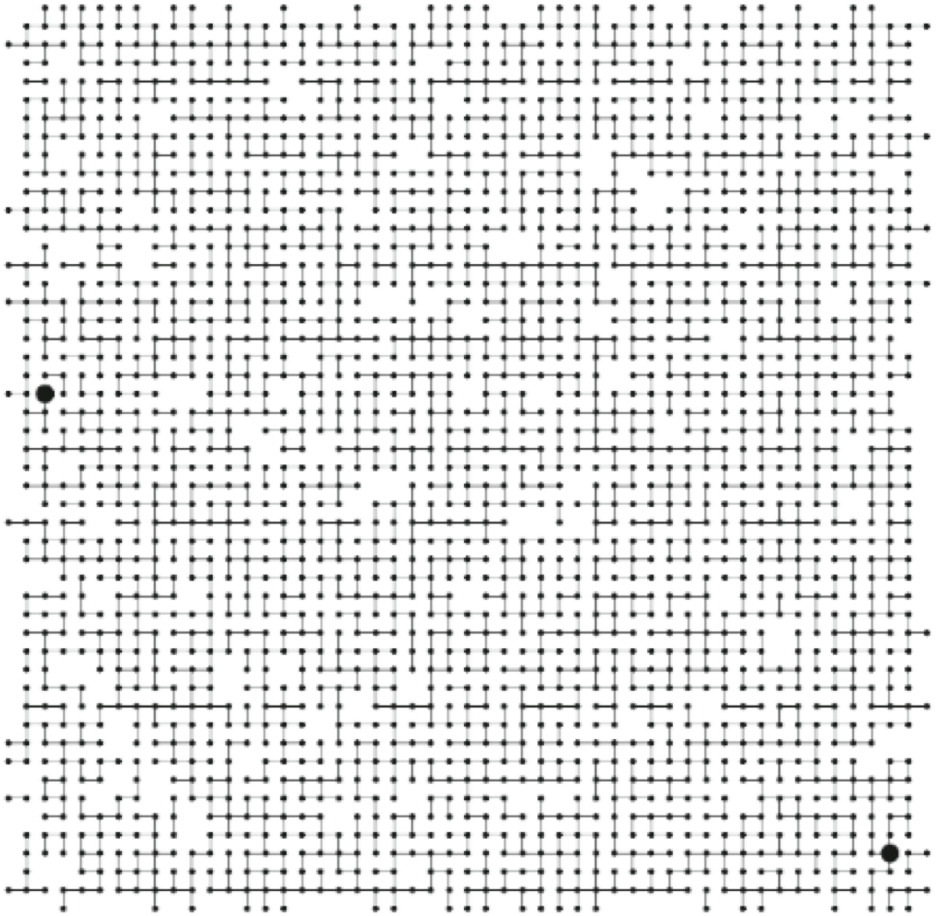
Für eine gegebene Folge von Ganzzahlpaaren (links), die Verbindungen zwischen Objekten repräsentieren, soll ein Vernetzungsalgorithmus die Paare ausgeben, die neue Verbindungen liefern (Mitte). Zum Beispiel gehört das Paar 2–9 nicht zur Ausgabemenge, weil sich die Verbindung 2–3–4–9 aus den bereits eingelesenen Verbindungen ableitet (zum Nachweis rechts dargestellt).

Abbildungen aus:
Robert Sedgewick, Algorithmen in Java,
3. Auflage, Pearson Studium 2003

Vernetzungsproblem - Anwendungen

- Ganzzahlen repräsentieren Computer in einem Netzwerk
  - Paare repräsentieren Verbindungen
  - Frage: muss für die Kommunikation zwischen p und q eine neue Leitung etabliert werden?
- Ganzzahlen repräsentieren Kontaktpunkte in einer elektrischen Schaltung
  - Paare stellen Verbindungsdrähte dar
  - Frage: Wie sieht ein Weg aus, der alle Punkte miteinander verbindet? (es gibt keine Garantie, dass die Kanten in der Liste ausreichen...)
- Programmierumgebungen: Äquivalenz von Variablennamen
  - Frage: sind nach einer Reihe von Deklarationen zwei gegebene Namen gleich

Ein großes Vernetzungsbeispiel



- Besteht eine Verbindung zwischen den beiden Knoten, die durch große schwarze Punkte markiert sind?

Lösungsidee

- Zwei abstrakte Operationen:
  - Suchen (find)
  - Vereinigen (union)
- Idee:
  - Nach Einlesen eines neuen Paars (p, q)

    → Suchen-Op. für jedes Element des Paares ausführen

    Gehören beide Elemente zur selben Menge → nächstes Paar bearbeiten

    sonst: Vereinigen-Op. ausführen (Paar ausgeben)
  - Mengen stellen verbundene Komponenten dar
    - Teilmengen aller Objekte
    - Alle Objekte innerhalb einer Komponente sind verbunden
- Also:
  - Datenstruktur definieren
  - Algorithmen für Suchen und Vereinigen auf dieser Datenstruktur entwickeln

Ein einfacher Algorithmus

- Naheliegend: alle Eingaben speichern und dann durchsuchen
  - Problem: kennen Größe der Eingabe nicht
  - Eingabe kann Größe des Speichers überschreiten
- Besser:
  - Speichern für jedes Objekt die Zugehörigkeit zu Komponente
  - Objekte durch ganze Zahlen beschrieben (id)
  - Objekt-id als Index für Array (id)
    Objekte mit gleichem Wert gehören zu einer Komponente
- Vereinigen: Array durchlaufen
  - alle Einträge i mit dem id[i] == id[p] werden auf id[q] geändert
  - Aufwändige Operation
- Suchen:
  - Array-Einträge auf Gleichheit testen (schnell)

QuickF.java

Quickfind - Java

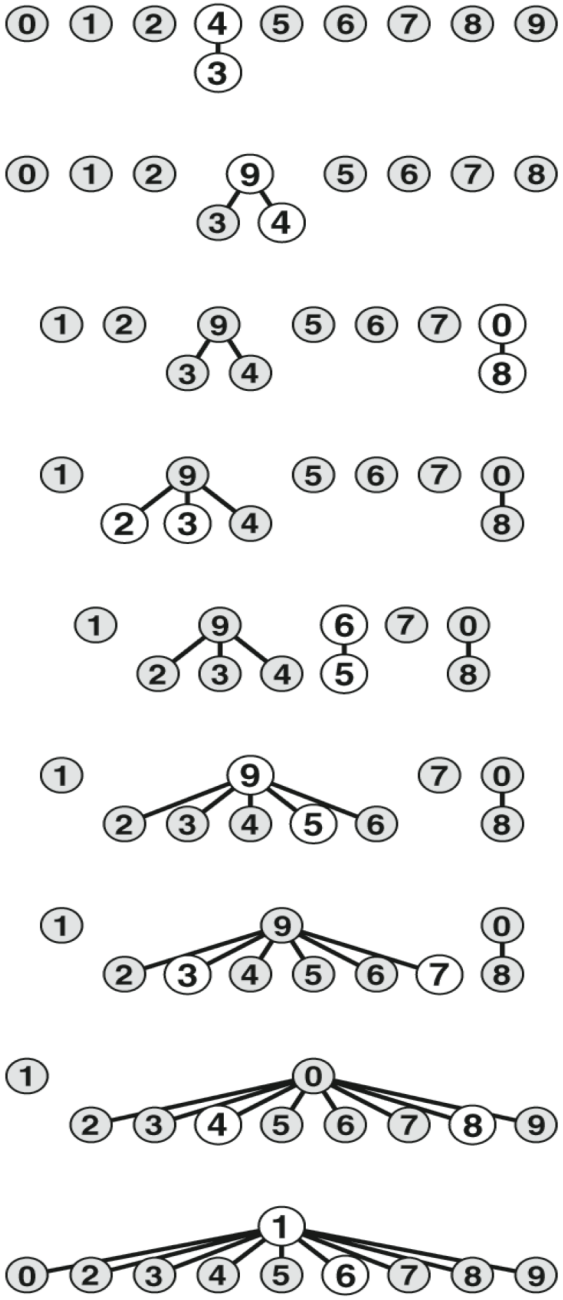
```
public class QuickF {
	public static void main(String[] args) {
		int N = Integer.parseInt(args[0]);
		 int id[] = new int[N];
		for (int i = 0; i < N ; i++) id[i] = i;
		for( In.init(); !In.empty(); ) {
			int p = In.getInt(), q = In.getInt();
			int t = id[p];
			if (t == id[q]) continue;
			for (int i = 0; i < N; i++)
				if (id[i] == t) id[i] = id[q];
			Out.println(" " + p + " " + q);
		}
	}
}
```

Programmlauf

- Quickfind-Algorithmus führt mindestens M * N-Anweisungen aus, um ein Vernetzungsproblem mit N Objekten zu lösen, für das M Vereinigungs-Operationen auszuführen sind

p	q	0	1	2	3	4	5	6	7	8	9
3	4	0	1	2	**4**	4	5	6	7	8	9
4	9	0	1	2	**9**	**9**	5	6	7	8	9
8	0	0	1	2	9	9	5	6	7	**0**	9
2	3	0	1	**9**	9	9	5	6	7	0	9
5	6	0	1	9	9	9	**6**	6	7	0	9
2	9	0	1	9	9	9	6	6	7	0	9
5	9	0	1	9	9	9	**9**	**9**	7	0	9
7	3	0	1	9	9	9	9	9	**9**	0	9
4	8	0	1	**0**	**0**	**0**	**0**	**0**	**0**	0	**0**
5	6	0	1	0	0	0	0	0	0	0	0
0	2	0	1	0	0	0	0	0	0	0	0
6	1	**1**	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Diese Sequenz zeigt den Inhalt des Arrays `id`*, nachdem der Quickfind-Algorithmus (Programm 1.1) jedes Paar auf der linken Seite verarbeitet hat. Grau unterlegte Einträge sind diejenigen, die sich bei der Vereinigungs-Operation ändern. Wenn wir das Paar* `p q` *verarbeiten, ändern wir alle Einträge mit dem Wert* `id[p]` *auf den Wert* `id[q]`*.*



Baumdarstellung

- Bäume:
  - Grundlegende kombinatorische Strukturen
  - Lassen sich schnell erstellen (*Vereinigen*)
  - *Suchen* lässt sich durch Wandern zur Wurzel implementieren
- Eigenschaften:
  - Jedes Objekt weist auf genau ein anderes Objekt
  - Nur Wurzelobjekt weist auf sich selbst
- QuickFind-Baum ist flach
  - Vereinigen teuer (wird häufig ausgeführt)
  - Suchen billig (wird einmal ausgeführt !)

p	q	0	1	2	3	4	5	6	7	8	9
3	4	0	1	2	4	4	5	6	7	8	9
4	9	0	1	2	4	9	5	6	7	8	9
8	0	0	1	2	4	9	5	6	7	0	9
2	3	0	1	9	4	9	5	6	7	0	9
5	6	0	1	9	4	9	6	6	7	0	9
2	9	0	1	9	4	9	6	6	7	0	9
5	9	0	1	9	4	9	6	9	7	0	9
7	3	0	1	9	4	9	6	9	9	0	9
4	8	0	1	9	4	9	6	9	9	0	0
5	6	0	1	9	4	9	6	9	9	0	0
0	2	0	1	9	4	9	6	9	9	0	0
6	1	1	1	9	4	9	6	9	9	0	0
5	8	1	1	9	4	9	6	9	9	0	0

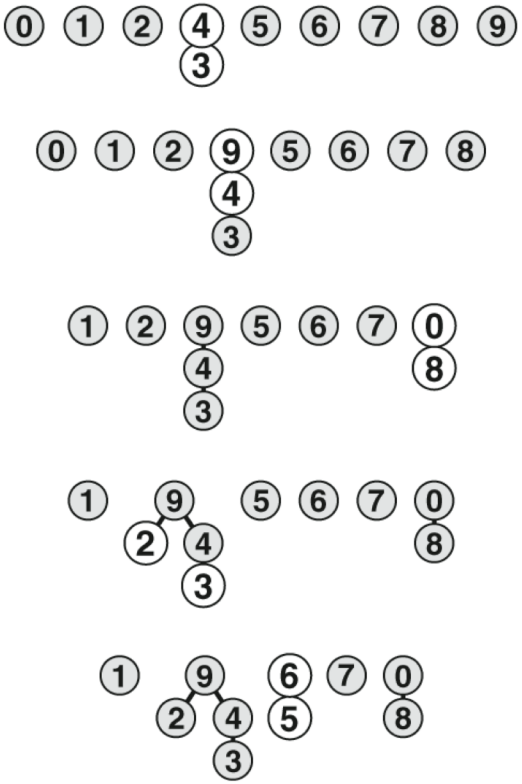
QuickUnion

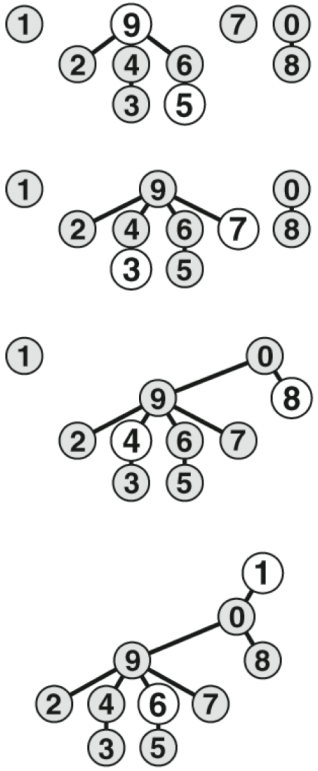
- Idee: Benutzen Einträge im Array als Verweise zum Aufbau einer mehrstufigen Baumstruktur

Diese Sequenz zeigt den Inhalt des Arrays `id`*, nachdem der Quickfind-Algorithmus (Programm 1.1) jedes Paar auf der linken Seite verarbeitet hat. Grau unterlegte Einträge ändern sich bei der Vereinigungs-Operation (nur einer pro Operation). Wenn wir das Paar* `p q` *verarbeiten, folgen wir Zeigern von* `p`*, um einen Eintrag* `i` *mit* `id[i] == i` *zu erhalten; dann folgen wir Zeigern von* `q`*, um einen Eintrag j mit* `id[j] == j` *zu erhalten; wenn sich danach* `i` *und* `j` *unterscheiden, setzen wir* `id[i] = id[j]`*. Bei der Suchen-Operation für das Paar* 5-8 *(letzte Zeile) nimmt* `i` *die Werte* 5 6 9 0 1 *und* `j` *die Werte* 8 0 1 *an.*

QuickU.java

Baumdarstellung Quickunion





- Problem:
  - Pfade in Bäumen können sehr lang werden
- Idee:
  - Wichten
  - Verbinden jeweils kleineren Teilbaum mit Größerem
- Gewichteter Quickunion-Alg.
  - Verfolgt höchstens ***ld N*** Zeiger, um zu bestimmen, ob zwei der N Objekte verbunden sind

QuickUW.java

Laufzeitabschätzung

- Quickfind:
  - Mindestens M*N Anweisungen für Vereinigung und Suche
- Quickunion:
  - Vereinigung funktioniert linear
  - Suche kann lange dauern:
    - Für M > N kann Quickunion mehr als M*N/2 Anweisungen benötigen, um ein Vernetzungsproblem mit M Paaren von N Objekten zu lösen
- Gewichteter Quickunion:
  - Benötigt M * ld N Anweisungen, um M Kanten von N Objekten zu verarbeiten
  - → was für eine Verbesserung
- Existiert womöglich auch ein Algorithmus mit linearer Laufzeit?
  - Pfadkomprimierung
  - Online-Algorithmus? – betrachtet Eingabedaten nur einmal

Algorithmenentwurf

- Algorithmen sind oft Teil einer größeren Anwendung
  - operieren auf Daten der Anwendung, sollen aber unabhängig von konkreten Typen sein
    - Darstellung der Algorithmen mit Hilfe generischer Typen
  - Eigenschaften der Gesamtanwendung hängen von Auswahl des Algorithmus ab
  - Algorithmen oft nicht direkt einsetzbar
    - Aufbereitung der Daten in Eingabeformat des Algorithmus oder Anpassung des Algorithmus?
    - Behandlung spezifischer Fehlerfälle
- Ziele des Algorithmenentwurfs
  - Korrektheit
  - Universalität: Anpassbarkeit an unterschiedliche Einsatzszenarien
  - Effizienz: Suche nach Algorithmen mit geringer Laufzeit
  - Einfachheit: Lesbarkeit, Validierbarkeit, Anpassbarkeit

Empirische Analyse

- Laufzeit von verschiedenen Algorithmen wird durch Messung bestimmt
  - Vergleich nur möglich bei gleichen Testbedingungen
    - gleiche Hardware, gleiche Software (mit Ausnahme des Algorithmus), gleiche Eingabedaten
    - Ausschluss von anderen Einwirkungen
      - Maschine soll unbelastet (*idle*) sein
- Problem: Abhängigkeit des Messergebnisses von Eingabe
  - Analyse mit “echten” Daten, Zufallsdaten, Sonderfälle
- Problem: Statistische Streuung
  - Mehrfache Wiederholung des Testlaufs
  - Umfangreiche Eingabedaten
- Problem: Einfluss des Messverfahrens auf das Ergebnis
  - Analyse des Messverfahrens selbst

Häufige Fehler

1. Ignoranz von Effizienzaspekten
   - Auswahl eines einfachen Algorithmus aus Angst vor Kompliziertheit des effizienten
2. Überbewertung von Effizienzaspekten
   - Komplizierter (effizienter) Algorithmus wird “aus Prinzip” gewählt, selbst wenn der Algorithmus nur auf kleinen Datenmengen operieren und selten aufgerufen wird

Mathematische Analyse

- Ziele
  - Vergleich mehrerer Algorithmen für die gleiche Aufgabenstellung
  - Vorhersage der Leistung (performance) eines Algorithmus auf einem neuen Zielsystem
  - Festlegung von Parametern für einen Algorithmus
- Ideal: Definition eines exakten math. Modells der Leistung
  - math. Formeln, die Leistung in Abhängigkeit von Eingabeparametern ausdrücken
  - Komplexität: Zahl, die die Leistung ausdrückt
- Probleme:
  - Analyse ist führt zu mathematisch ungelösten Problemen
  - Analyse benötigt Informationen, die nicht bekannt sind
    - Eigenschaften von Zielcodeanweisungen (etwa JVM Bytecodes)
    - Eigenschaften der Eingabedaten

Grundlagen der Analyse

- Metriken: Messgrößen der Leistung eines Programms
  - Benötigte Rechenzeit (in Abhängigkeit von Eingabe)
  - Benötigter Hauptspeicher
  - Zahl der Speicherzugriffe
  - Zahl der Gleitkommaoperationen
  - …
- Abstrakte Messgrößen: abstrakte Operationen
  - Messgröße hängt nicht von Zielmaschine ab
    - es werden nur die “teuren” Operationen gezählt
  - Zahl der Zugriffe auf ein Feld
  - Zahl der Vergleichsoperationen
  - Ideal: tatsächliche Rechenzeit ergibt sich aus Skalierung der abstrakten Größen
  - Real: tatsächliche Rechenzeit hängt zusätzlich von anderen Faktoren ab
    - Beispiel: Speicherzugriff hängt davon ab, ob Wert im Cache steht oder im Hauptspeicher

Grundlagen der Analyse

- Zahl der Operationen hängt von Eingabe ab
  - Verallgemeinerung: Aussagen über Mengen von Eingabedaten
- Durchschnitt (average-case performance)
  - Annahme: alle Eingabedaten sind statistisch gleichverteilt
  - Annahme ist u.U. unrealistisch
- Maximalwert (worst-case performance)
  - Ermittlung des Eingabedatensatzes mit maximaler Komplexität
  - dieser Fall tritt u.U. nie ein
- Unterschied zwischen Durchschnitt und Maximum gibt Einblick in die Datenabhängigkeit der Komplexität
  - bei großen Abweichungen muss untersucht werden, unter welchen Umständen die “schlechten” Fälle auftreten

Wachstum von Funktionen

- Die meisten Algorithmen haben *primären Parameter N*, der die Laufzeit am stärksten beeinflusst
  - Grad eines Polynoms
  - Größe einer zu durchsuchenden oder zu sortierenden Datei
  - Anzahl der Zeichen in einem Textstring
- Oftmals direkt proportional zur Größe der zu verarbeitenden Daten
- Bei mehreren Parametern häufig Reduktion auf nur einen Parameter
  - Zweiten Parameter als konstant annehmen
  - Zweiten Parameter als Funktion des anderen ausdrücken
- Ziel: Ressourcenbedarf (Laufzeit) in Abhängigkeit von N auszudrücken

Typische Laufzeiten

- 1:
  - die meisten Anweisungen werden einmal (einige Male) ausgeführt
  - Konstante Laufzeit
- Log N:
  - Logarithmische Laufzeit: Programm wird mit wachsendem N allmählich langsamer
  - Problem wird in Folge kleinerer Teilprobleme überführt
  - Problemgröße um konstanten Bruchteil in jedem Schritt beschnitten
- N:
  - Lineare Laufzeit
  - Optimal für einen Algorithmus der N Eingabewerte verarbeiten muss
- N log N:
  - Problem wird in Teilprobleme zerlegt; diese werden unabhängig bearbeitet
  - Teillösungen werden zu Gesamtlösung zusammengefügt
- N^2, N^3:
  - Quadratische / kubische Laufzeit; Algorithmus eignet sich nur für kleine Probleme
- 2^N:
  - Exponentielle Laufzeit; Brachiallösung

Bestimmung der asymptotischen Komplexität

- Komplexität hängt von der Eingabemenge ab
  - Eingabedaten oft iterativ oder rekursiv verarbeitet
- Schleifen: Zählen der Durchläufe, Abschätzung der Kosten eines Durchlaufs
  - evtl. Abschätzung der Zahl der Durchläufe (obere Schranke)
- Rekursion: induktive Berechnung der Komplexität
  - Berechnung des nicht-rekursiven Falls (Abbruch der Rekursion)
  - induktives Folgern der rekursiven Fälle
- Laufzeit eines Programmes
  - Konstante multipliziert mit *führendem Term* plus einiger kleinerer Terme
  - Konstante Koeffizienten und eingeschlossene Terme hängen von Analyse und Implementierungsdetails ab
  - Für große N dominiert Einfluss des führenden Terms

Relative Größe einiger Funktionen

log N	$\sqrt{N}$	N	N log N	N (log N)^2	N$^{3/2}$	N^2
3	3	10	33	110	32	100
7	10	100	664	4.414	1.000	10.000
10	32	1.000	9.966	99.317	31.623	1.000.000
13	100	10.000	132.877	1.765.633	1.000.000	100.000.000
17	316	100.000	1.660964	27.588.016	31.622.777	10.000.000.000
20	1.000	1.000.000	19.931.569	397.267.426	1.000.000.000	1.000.000.000.000

Sekunden werden zu Stunden...

Sekunden	
10^2	1,7 Minuten
10^4	2,8 Stunden
10^5	1,1 Tage
10^6	1,6 Wochen
10^7	3,8 Monate
10^8	3,1 Jahre
10^9	3,1 Jahrzehnte
10^{10}	3,1 Jahrhunderte
10^{11}	*niemals*

Abbildung 2.1: Umrechnungstabelle für Sekunden

Der riesige Unterschied zwischen Zahlen wie 10^4 und 10^8 tritt deutlicher zutage, wenn wir diese Werte als Anzahl der Sekunden auffassen und in die uns gewohnten Zeiteinheiten umrechnen. Es kann durchaus akzeptabel sein, wenn ein Programm 2,8 Stunden läuft, auf den Abschluss eines Programmlaufs, der sich über mindestens 3,1 Jahre erstreckt, werden wir aber wohl kaum warten wollen. Da 2^{10} ungefähr 10^3 ist, lässt sich diese Tabelle auch für Zweierpotenzen nutzen. Beispielsweise sind 2^{32} Sekunden ungefähr 124 Jahre.

Zeit für Lösung sehr großer Probleme

Operationen je Sekunde	Problemgröße 1 Million			Problemgröße 1 Milliarde		
	N	***N log N***	***N^2***	***N***	***N log N***	***N^2***
10^6	Sekunden	Sekunden	Wochen	Stunden	Stunden	Niemals
10^9	Sofort	Sofort	Stunden	Sekunden	Sekunden	Jahrzehnte
10^{12}	Sofort	Sofort	Sekunden	Sofort	Sofort	Wochen

Logarithmus und ganze Zahlen

- Logarithmusfunktion zentral bei Analyse von Algorithmen
  - log N – Schreibweise ohne Angabe der Basis
  - Änderung der Basis drück sich nur durch konstanten Faktor aus...
  - Natürlicher Logarithmus – zur Basis 2 – Schreibweise $\log_2 N \equiv \mathrm{ld}\ N \equiv \mathrm{lg}\ N$

$\lfloor x \rfloor$ *größte ganze Zahl kleiner oder gleich x*

$\lceil x \rceil$ *kleinste ganze Zahl größer oder gleich x*

- Gesucht: “kleinste ganze Zahl größer als ld N”
  - Zahl der Bits, die zur Darstellung von N nötig sind
- Lösung:
  - for (ldN = 0; N > 0; ldN++, N /= 2);
  - for (ldN = 0, t = 1; t < N; ldN++, t+=t);

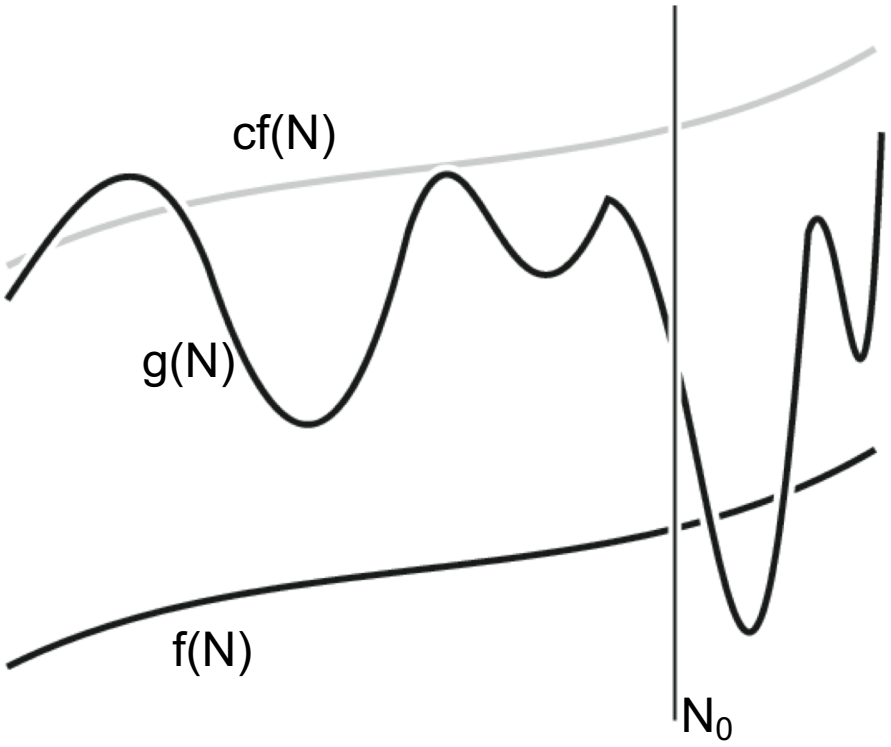
O-Notation (Big-Oh, Landau-Notation)

- Sei f: W $\rightarrow$ W eine Funktion. Die Menge O(f) enthält alle Funktionen g, die ab einem gewissen n_0 höchstens so schnell wachsen wie f, abgesehen von jeweils einem konstanten Faktor c:

$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} | \ \exists c > 0 \ \exists n_0 \in \mathbb{N} \ \forall n >= n_0 : g(n) <= c \cdot f(n)\}$$

- c, n_0: Parameter der konkreten Implementierung
  - In der Praxis oft entscheidend für die tatsächliche Rechenzeit
- O-Notation gibt nur die Größenordnung an
  - $O(n^2+n) = O(n^2)$
  - Komplexitätsklasse

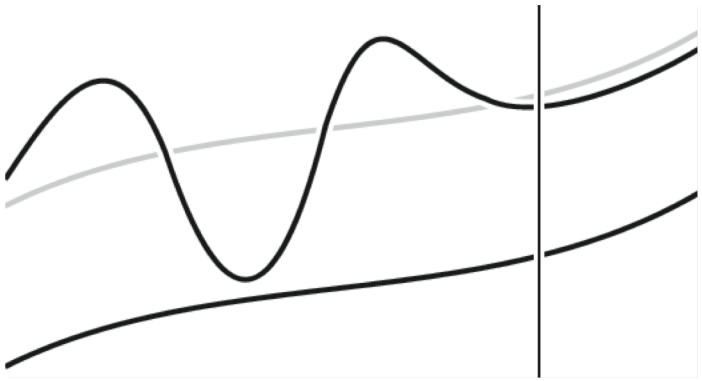
Begrenzen einer Funktion mit O-Näherung

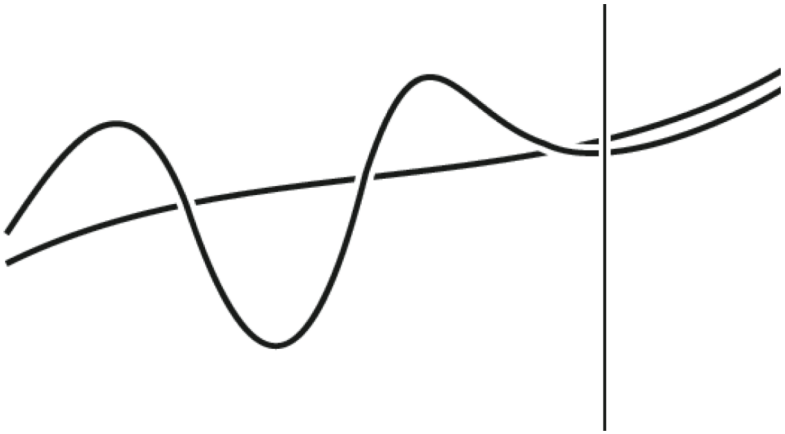


- g(N) = O(f(N)), d.h.:
  Wert von g(N) fällt unter eine Kurve der Gestalt f(N) rechts von einer bestimmten vertikalen Linie N_0

- Für $N > N_0$ gilt $g(N) < c * f(N)$
- Konstante c ist nicht näher spezifiziert
- Für kleine N (und kleine f(N)) überwiegt Einfluss von c !!

Funktionelle Näherungen





Wenn wir sagen, dass g(N) zu f(N) proportional ist (links) *erwarten wir, dass die Funktion schließlich wie f(N) wächst, eventuell aber um eine unbekannte Konstante verschoben ist. Für einen gegebenen Wert von g(N) können wir die Funktion für größere N abschätzen. Wenn wir sagen, dass g(N) ungefähr f(N) ist* (rechts) *erwarten wir, dass wir letztendlich mithilfe von f den Wert von g genau schätzen können.*

Komplexitätsklassen

- Parameter ist implizit “n”:
  - O(n) = O(f) mit f(n)=n
  - O(1): konstant
  - O(log n): logarithmisch
  - O(n): linear
  - …
- O(f) gibt obere Schranke an
  - $8n \in O(n)$
  - $8n*n \in O(n^2)$
  - $65 \in O(1)$

Rechnen mit der O-Notation

- Algebraische Ausdrücke erweitern
  - (als wäre O gar nicht vorhanden)
  - Alle, außer dem größten Term fallen lassen
- Bsp:

  $(N + O(1))(N + O(\log N) + O(1))$

  $= N^2 + O(N) + O(N \log N) + O(\log N) + O(N) + O(1)$

  $\equiv N^2 + O(N \log N)$

- N^2 stellt eine gute Näherung dar, wenn N groß ist
- Formel mit O-Term heißt *asymptotischer Ausdruck*

Laufzeiten

- Einfluss der Verdopplung der Problemgröße auf die Laufzeit

1	kein
$\lg N$	leichter Anstieg
N	Verdopplung
$N \lg N$	etwas mehr als das Doppelte
$N^{3/2}$	Faktor $2\sqrt{2}$
N^2	Faktor 4
N^3	Faktor 8
2^N	quadratisch

Abbildung 2.3: Einfluss der Verdoppelung der Problemgröße auf die Laufzeit

Der Einfluss der Verdoppelung der Problemgröße auf die Laufzeit lässt sich leicht voraussagen, wenn die Laufzeit zu bestimmten einfachen Funktionen proportional ist, wie sie in dieser Tabelle angegeben sind. In der Theorie können wir uns nicht auf diesen Effekt verlassen, außer wenn N sehr groß ist, dennoch ist diese Methode überraschend wirkungsvoll. Umgekehrt lässt sich das funktionelle Wachstum der Laufzeit eines Programms schnell bestimmen, wenn man die Laufzeit des Programms empirisch ermittelt, indem man die Eingabegröße für ein möglichst großes N verdoppelt und dann von dieser Tabelle aus rückwärts schließt.

Beispiel: Lineare Suche

- Auch: sequentielle Suche

```
static int search(int a[], int v, int l, int r)
{
    int i;
    for(i=l; i <= r; i++)
        if (v == a[i])
            return i;
    return –1;
}
```

Analyse des Algorithmus

- Laufzeit hängt davon ab, ob das gesuchte Objekt im Array vorhanden ist
  - Worst case: erfolglose Suche
  - Laufzeit hängt von den Daten ab
  - *Vorhersage* vs. *Garantie* der Laufzeit
- Eigenschaft 1:
  - Sequentielle Suche prüft N Zahlen für jede erfolglose und im Mittel N/2 Zahlen für jede erfolgreiche Suche
  - Kosten bei gleicher Auftretenswahrscheinlichkeit aller Elemente:
    (1 + 2 + ... + N) / N = (N + 1) / 2
- Eigenschaft 2:
  - Sequentielle Suche in einer sortierten Tabelle prüft für jede Suchoperation im ungünstigsten Fall N Zahlen und im Durchschnitt N/2 Zahlen
- Laufzeit ist propotional zu M * N für M Transaktionen

Beispiel: Binäre Suche

- Annahme: Feld ist aufsteigend sortiert

```
static int search(int a[], int v, int l, int r)
{
    while (r >= l) {
        int m = (l+r)/2;
        if (v == a[m]) return m;
        if (v < a[m]) r = m-1;
        else l = m+1;
    }
    return –1;
}
```

Analyse des Algorithmus

1488	1488			
1578	1578			
1973	1973			
3665	3665			
4426	4426			
4548	4548			
5435	5435	5435	5435	5435
5446	5446	5446	5446	
6333	6333	6333		
6385	6385	6385		
6455	6455	6455		
6504				
6937				
6965				
7104				
7230				
8340				
8958				
9208				
9364				
9550				
9645				
9686				

Binäre Suche prüft niemals mehr als $\lfloor \mathrm{ld}\ N \rfloor + 1$ Zahlen

Abbildung 2.7: Binäre Suche

Um festzustellen, ob 5025 in der Tabelle der Nummern in der linken Spalte enthalten ist, vergleichen wir sie zuerst mit 6504; daraus ergibt sich, dass wir die erste Hälfte des Arrays betrachten. Dann vergleichen wir mit 4548 (die Nummer in der Mitte der ersten Hälfte); das führt uns zur zweiten Hälfte in der ersten Hälfte. Wir fahren fort und arbeiten immer auf einem Teilarray, das die gesuchte Nummer enthalten könnte, falls sie in der Tabelle existiert. Schließlich erhalten wir ein Teilarray mit genau 1 Element, das nicht mit 5025 gleich ist, sodass 5025 in der Tabelle nicht vorhanden ist.

Garantien, Vorhersagen, Beschränkungen

- Laufzeit der meisten Algorithmen hängt von Eingabedaten ab
- Untersuchung des ungünstigsten Falls:

  → Garantie für die Laufzeit eines Programms
  - Aber: mitunter großer Unterschied zwischen ungünstigstem und durchschnittlichem Fall
  - Idee: Analyse des Verhaltens bei zufälliger Verteilung der Eingabedaten

- Verhalten im ungünstigsten Fall: obere Schranke für Laufzeit
  - Suchen untere Schranke für Laufzeit
  - Nachweisen, dass jede Lösung eines Problems eine bestimmte Zahl fundamentaler Operationen ausführen muss
  - Erfordert Berücksichtigung eines genauen Maschinenmodells

- Fallen obere und untere Schranke zusammen, so werden wir keinen grundsätzlich schnelleren Algorithmus finden können - Optimalität
  - Bsp: Binäre Suche ist optimal - kein Algorithmus kommt im ungünstigste Fall mit weniger Vergleichen aus

Zusammenfassung

- Vernetzungsprobleme – ein Beispiel
  - Quickfind, Quickunion
  - Laufzeitbetrachtung
- Empirische Analyse
- Mathematische Analyse
- Komplexitätsklassen
- O() Notation
- Lineare und binäre Suche