Algorithmen und Datenstrukturen Zusammenfassung

Emanuel Regnath

February 26, 2012

Ich übernehme keine Gewähr für Vollständigkeit oder Korrektheit!
Fehler bitte an emanuel.regnath@tum.de senden! Hompage: www.emareg.de

Emanuel Regnath

1 Allgemeines

1.1 Mengenalgebra

Potenzmenge: Die Menge aller Teilmengen. $\mathcal{P}(A) = \{U \mid U \subseteq A\}$
Karthesisches Produkt: $A \times B = \{(a,b) \mid a \in A \wedge b \in B\}$
Zwei Mengen A und B heißen disjunkt, wenn $A \cap B = \emptyset$

1.1.1 Relationen

Eine zweistellige Relation R zwischen zwei Mengen A und B ist eine Teilmenge von $A \times B$
Eigenschaften von $R \subseteq A \times B$:

reflexiv: $\forall a \in A : (a,a) \in R \qquad aRa$ (ist wahr)

symmetrisch: $\forall a, b \in A : (a,b) \in R \Rightarrow (b,a) \in R \qquad aRb \Leftrightarrow bRa$

antisymmetrisch: $\forall a, b \in A : (a,b) \in R \wedge (b,a) \in R \Rightarrow a = b \qquad aRb \;\wedge\; bRa \Rightarrow a = b$

transitiv: $\forall a, b, \in A : (a,b) \in R \wedge (b,c) \in R \Rightarrow (a,c) \in R \qquad aRb \;\wedge\; bRc \Rightarrow aRc$

R heißt partielle Ordnung, falls R reflexiv, antisymmetrisch und transitiv ist.
R heißt Äquivalenzrelation, falls R reflexiv, symmetrisch und transitiv ist.
Eine partielle Ordnung hießt totale Ordnung, falls alle Elemente miteinander vergleichbar sind.

1.2 Abbildung

Eine Abbildung ist eine Relation $R \subseteq A \times B$:
$\boxed{\forall a \in A : |\{b \in B \mid (a,b) \in R\}| = 1}$
Schreibweise: $f : A \to B, a \mapsto f(a)$

1.3 Sonstiges:

Auf/Abrunden: $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
$\lfloor 3,7 \rfloor = 3 \qquad \lceil 3,1 \rceil = 4 \qquad \lceil \frac{a}{b} \rceil \leq \frac{a+(b-1)}{b}$

Modulo $\boxed{a\%n = a \mod n = a - \left(\left\lfloor \frac{a}{n} \right\rfloor \cdot n \right)}$
Gesucht r mit $a = nq + r \qquad 0 \leq r < n, q \in \mathbb{Z}$

Fakultät: $n! = \begin{cases} n \cdot (n-1)! & \text{für } n \geq 1 \\ 1 & \text{für } n = 0 \end{cases}$

Fibonacci-Zahl: $f(n) = \begin{cases} f(n-1) + f(n-2) & \text{für } n \geq 2 \\ 1 & \text{für } n = 1 \\ 0 & \text{für } n = 0 \end{cases}$
Fibonacci-Folge: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$

Alphabet A: Endliche, nichtleere Menge von Elementen.
$(A, <)$ heißt geordnetes Alphabet, wenn $<$ eine totale Ordnung auf A ist.
Wort über A: $w = a_1 a_2 \dots a_k$ $A^k := \{w \mid |w| = k\}$
$A^* = \bigcup_{k=0}^{\infty} A^k$
$A^+ = \bigcup_{k=1}^{\infty} A^k$

1.4 Pseudocode

Abstraktion von verschiedenen Programmiersprachen.

```
1  //Komentar: einige Pseudocode Strukturen:
2  i=j=5                          //Gleichzeitiges Zuweisen des Wertes 5
3  for i = 0 {to|downto} end (by steps) do
4     Schleifenanweisungen        //Einrückung beachten!
5  while i < struct.attribut do
6     FUNCTION(par_1, par_2)      //Funktionen werden GROSS geschrieben
7     if i == 0 oder i == 1
8        print "i ist null oder eins!"
9     elseif i < 3 und i > 0
10       print "i ist zwei!"
11    else
12       print "i ist " i
13 return A[j]    // A ist ein Feld
```

1.5 Newton-Verfahren

NEWTONS-METHOD$(x_0, f, f', \varepsilon)$
$error = \varepsilon + 1$
while $error > \varepsilon$
$x_{next} = x - \frac{f(x)}{f'(x)}$
$error = |x_{next} - x|$

$x = x_{next}$
return x

2 Algorithmen allgemein

Ein Algorithmus ist ein Verfahren mit einer **präzisen** (d.h. in einer genau festgelegten Sprache abgefassten) **endlichen** Beschreibung, unter Verwendung **effektiver** (tatsächlich ausführbarer), **elementarer** Verarbeitungsschritte. Ein Algorithmus besitzt eine oder mehrere Eingaben (Instanz mit Problemgröße n) und berechnet daraus eine oder mehrere Ausgaben. Die Qualität eines Algorithmus ergibt sich aus seiner Effizienz, Komplexität, Robustheit und Korrektheit.

Eigenschaften:
Determiniert: Der Algorithmus liefert bei gleichen Startbedingungen das gleiche Ergebnis.
Deterministisch: Die nächste, anzuwendende Regel ist zu jedem Zeitpunkt definiert.

2.1 Effizienz

Die Effizienz eines Algorithmus ist seine Sparsamkeit bezüglich der Ressourcen, Zeit und Speicherplatz, die er zur Lösung eines festgelegten Problems beansprucht.

2.2 Komplexität

Schrankenfunktionen: $1 < \log_{10}(n) < \ln(n) < \log_2(n) < \sqrt{n} < n < n \cdot \ln(n) < (\log n)! < n^2 < e^n < n! < n^n < 2^{2^n}$ Aber $\left\{\log_{10}(n), \ln(n), log_2(n)\right\} \in \Theta(\log n)$

Landau-Symbole:

Notation	Definition	
$f \in \mathcal{O}\big(g(n)\big)$	$0 \leq f(n) \leq c \cdot g(n)$	$\forall n > n_0$
$f \in \Omega\big(g(n)\big)$	$f(n) \geq c \cdot g(n) \geq 0$	$\forall n > n_0$
$f \in \Theta\big(g(n)\big)$	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$\forall n > n_0$

Substitutionsmethode: Lösung raten, einsetzen und mit Induktion beweisen.

Laufzeituntersuchung rekursiver Algorithmen mit Master Theorem:
Ggegeben: $\boxed{T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \text{ mit } a \geq 1, b > 1}$
$a \geq 1$: Anzahl der Unterprobleme innerhalb einer Rekursionstiefe(meist 1 oder 2)
$b > 1$: Faktor um den jedes Unterproblem verkleinert ist.
$f(n)$: Aufwand der durch Division des Problems und Kombination der Teillösungen entsteht(nicht rekursiver Anteil, von $T(n)$ unabhängig.

- Falls $f(n) \in \mathcal{O}\left(n^{\log_b a - \varepsilon}\right)$
  Dann ist $T(n) \in \Theta\left(n^{\log_b a}\right)$
- Falls $f(n) \in \Theta\left(n^{\log_b a}\right)$
  Dann ist $T(n) \in \Theta\left(n^{\log_b a} \log(n)\right)$

- Falls $f(n) \in \Omega\left(n^{\log_b a+\varepsilon}\right)$
  Dann ist $T(n) \in \Theta(f(n))$

2.3 Robustheit

Die Robustheit eines Algorithmus beschreibt die Fähigkeit auch bei ungünstigen Eingaben korrekt und effizient zu terminieren.

2.4 Korrektheit

Ein Algorithmus heißt korrekt, wenn er für jede Eingabeinstanz mit korrekter Ausgabe terminiert.
Qualitätskontrolle:

- Überprüfung mit geeigneten Eingabedaten die alle möglichen Fälle testen. Deckt einzelne Fehler auf, aber fehlerfreiheit nicht garantiert.

- Formaler Beweis mit Hoare-Kalkül etc. meist sehr schwierig.

2.5 Strategien

Greedy-Strategie: Wähle die aktuell beste Möglichkeit, wenn es mehrere gibt.
Teile und Herrsche: Teile ein komplexes Problem in kleine, löse diese und füge am Ende alles zusammen.

2.6 Rekursion

Eine Funktion ruft sich selbst auf bis ein Abbruchereignis eintritt. Danach werden die Rückgabewerte in umgekerhrter Reihenfolge verkettet.
Beispiel Fakultät: $fac(n) = n \cdot fac(n-1), fac(1) = 1$

3 Elementare Datentypen

Typ	Speicher	signed	unsigned
boolean	1 Byte	$\{0, 1\}$	–
char	1 Byte	$-128 \ldots 127$	$0, \ldots, 255$
short	2 Byte	$-32768 \ldots 32767$	$0 \ldots 65535$
int	4 Byte	$-2^{31} \ldots 2^{31}-1$	$0 \ldots 2^{32}-1$
long	8 Byte	$-2^{63} \ldots 2^{63}-1$	$0 \ldots 2^{64}-1$
float	32 Bit		–
double	64 Bit		–

3.1 Gleitkommadarstellung nach IEEE 754

$Wert = (-1)^s \cdot 2^{e-127} \cdot 1.f$	Bsp: $-0.625 = -1 \cdot 2^{-1} \cdot 1.01_2$
s: Vorzeichen, e: Exponent, f: Mantisse	$\Rightarrow$ $s = 1$, $e = 126$, $f = 01_2$

© Emanuel Regnath

Spezialwerte: $Wert = 0 \Leftrightarrow e = 0$ $Wert = \infty \Leftrightarrow e = 255$
Bitverteilung(single/double):

$s(1)$	$e(8/11)$	$f(23/52)$

4 Datenstrukturen

Eine Datenstruktur ist eine logische Anordnung von Daten mit Zugriffs- und Verwaltungsmöglichkeiten der repräsentierten Informationen über Operationen.
Eine Datenstruktur besitzt:

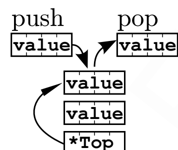
- Menge von Werten
- Literale zum Bezeichnen von Werten
- Menge von Operationen auf die Werte

4.1 Wichtige Datenstrukturen

Wichtige Datenstrukturen:

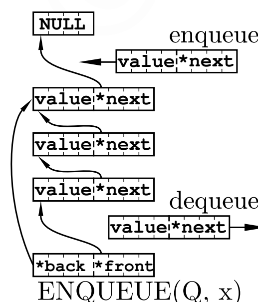
- Felder, Listen
- Bäume
- Hashtabellen
- Keller/Stapel
- Warteschlangen

4.2 Stapel (Stack)



Basieren auf dem LIFO(last in first out) Prinzip.
`push(a)`: Legt ein neues Element `a` oben auf den Stack und erhöht `*top`
`pop()`: Nimmt das oberste Element vom Stack und reduziert `*top`
`*top`: Ist ein Zeiger der auf das oberste Element zeigt.

4.3 Warteschlange (Queue)



`*front`: zeigt auf das erste Element der Warteschlage
`*back`: zeigt auf das Ende der Warteschlange.
`enqueue(x)`: x am Ende der Warteschlange hinzufügen.
`dequeue()`: Erstes Element aus der Warteschlange nehmen.

ENQUEUE(Q, x)
1 Q[ende[Q]] ← x

© Emanuel Regnath

2 if ende[Q] = länge[Q]
3 then ende[Q] ← 1
4 else ende[Q] ← ende[Q] + 1

DEQUEUE(Q)
1 x ← Q[kopf [Q]]
2 if kopf [Q] = länge[Q]
3 then kopf [Q] ← 1
4 else kopf [Q] ← länge[Q] + 1
5 return x

5 Hashtabellen

... sind Felder bei denen die Position eines Objekts durch eine Hashfunktion berechnet wird. Da es zu Kollisionen kommen kann, werden in den Feldern nur Verweise auf eine Liste gespeichert.

Schlüssel: wird von einem Schlüsselgenerator aus den Daten generiert.

5.1 Hashfunktion

... ordnet jedem Schlüssel aus einer großen Schlüsselmenge einen möglichst eindeutigen Wert aus einer kleineren Indexmenge zu. $h : key \to index$

Operatoren: Verkettete Hashtabelle: Jedes Feld entspricht einer Liste die mehrere kollidierte Daten speichern kann.

chained-hash-insert(T,x) : Füge x an den Kopf der Liste $T[h(x.schluessel)]$

chained-hash-search(T,k) : Suche Element k in der Liste $T[h(k)]$

chained-hash-delete(T,x) : entferne x aus der Liste $T[h(x.schluessel)]$

6 Sortieralgorithmen

in-place: Nur konstanter Hilfsspeicher nötig. $S : \mathcal{O}(1)$
out-of-place: Zusätzlicher Speicher abhängig von n nötig. $S : \mathcal{O}(f(n))$

6.1 Insertion-Sort

1. Wähle beginnend bei 2 das nächste Element.
2. Solange es kleiner als seine Vorgänger ist, tausche es.

Im schlimmsten Fall $\frac{n}{2}(n-1)$
$INSERTIONSORT(A)$
1 for $i \leftarrow 2$ to Länge(A) do
2 $key \leftarrow A[i]$
3 $j \leftarrow i$
4 while $j > 1$ and $A[j-1] > key$ do
5 $A[j] \leftarrow A[j-1]$

6 $\quad j \leftarrow j - 1$
7 $A[j] \leftarrow key$

6.2 Quicksort

1. Wähle ein Pivotelement, welches die Liste in zwei Hälften teilt.
2. Sortiere die Liste so um, das Elemente die kleiner als das Pivotelement in der einen Hälfte und größere in der anderen Hälfte sind. Suche dazu mit zwei Laufvariablen das Feld ab, bis jede eine unpassende Variable gefunden hat, dann tausche diese.
3. Wiederhole die Schritte 1. bis 3. mit beiden Teillisten, bis jede Teilliste sortiert ist.

$QUICKSORT(A, p, r)$
1 if $p < r$
2 then $q = PARTITION(A, p, r)$
3 $QUICKSORT(A, p, q–1)$
4 $QUICKSORT(A, q + 1, r)$

$PARTITION(A, p, r)$
1 $x = A[r]$ //Pivotelement
2 $i = p - 1$
3 for $j = p$ to $r - 1$ //Alle Elemente durchlaufen
4 if $A[j] \leq x$
5 $i = i + 1$
6 vertausche $A[i] \leftrightarrow A[j]$
7 vertausche $A[i + 1] \leftrightarrow A[r]$
8 return $i + 1$

Komplexität:
Worst-Case bei aufsteigend oder absteigend sortierter Liste wenn Pivot-Element am Rand des Feldes gewählt wird.

6.3 Mergesort

Feld in der Mitte rekursiv halbieren, bis Feldlänge = 1
Teilsortierte Felder zusammenfügen(Reißverschluss)

6.4 Heapsort

HEAPSORT (A)
1. BUILD-MAX-HEAP (A)
2. for i = länge [A] downto 2
3. do vertausche A [1] ↔ A[i]
4. heap-größe [A] = heap-größe [A] -1
5. MAX-HEAPIFY (A,1)

© Emanuel Regnath

BUILD-MAX-HEAP (A)
1. heap-größe [A] = länge [A]
2. for i = $\lfloor$ länge [A] /2 $\rfloor$ downto 1
3. do MAX-HEAPIFY (A,i)

MAX-HEAPIFY (A, i)
1. l = LEFT (i)
2. r = RIGHT (i)
3. if l $\leq$ heap-größe [A] und A[l] ¿ A[i]
4. then maximum = l
5. else maximum = i
6. if r $\leq$ heap-größe [A] und A[r] ¿ A [maximum]
7. then maximum = r
8. if maximum $\neq$ i
9. then vertausche A[i] $\leftrightarrow$A [maximum]
10. MAX-HEAPIFY (A, maximum)

6.5 Laufzeiten und Speicherbedarf von Sortieralgorithmen

Die Laufzeit bzw. Taktzyklen in Abhängigkeit einer (meist großen) Eingabemenge n.

Name	Best	Avg	Worst	Zusätzlicher Speicher
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	in-place
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	in-place
Bubblesort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	in-place
Merge	$\mathcal{O}(n \cdot \log_2 n)$	$\mathcal{O}(n \cdot \log_2 n)$	$\mathcal{O}(n \cdot \log_2 n)$	$\mathcal{O}(n)$
Heap-Sort	$\mathcal{O}(n \cdot \log_2 n)$	$\mathcal{O}(n \cdot \log_2 n)$	$\mathcal{O}(n \cdot \log_2 n)$	in-place
Quick	$\mathcal{O}(n \cdot \log_2 n)$	$\mathcal{O}(n \cdot \log_2 n)$	$\mathcal{O}(n^2)$	in-place

in-place bedeutet uusätzlicher Speicher von $\mathcal{O}(1) = \text{const.}$
Vergleichende Sortieralgorithmen brauchen mindestens $\Omega(n \log_2 n)$ Vergleiche, egal wie clever sie sind!

6.6 Suchalgorithmen

Algorithmus	Geeignet für	Laufzeit	Idee
sequentielles Suchen	statische kleine Mengen	$\mathcal{O}(n)$	Feld durchlaufen und vergleichen.
binäres Suchen	statische große Mengen	$\mathcal{O}(\log n)$	Vorsortieren, Vergleich mit Mitte.
binärer Suchbaum	dynamisch		

6.7 String-matching

In einer Zeichenfolge $T[0...n]$ wird ein Muster $P[0...m]$ gesucht.

Rabin-Karp Algorithmus: Jedes Zeichen wird als Zahl interpretiert.
Muster $p = P[m] + 10P[m-1] + ... + 10^i P[m-i]$
- Bilde modulu mit einer Primzahl q (Vorfiltern): $T[s...s+m]\%q$
- Bei Übereinstimmung des Restes: genauere Prüfung.
Laufzeit: $\mathcal{O}(m+n)$ bei $m << n$: $\mathcal{O}(n)$

7 Graphen

$G = (E, V)$ E:Edge, V: Vnode
Adjazenzmatrix ($V \times V$): 1 oder 0 für Verbindung.
Adjazenzliste: Für jeden Knoten alle Nachbarknoten angeben.
Startknoten s

Breitensuche(BFS): Von einem Startknoten S werden alle Knoten mit Abstand k durchsucht. Der Suchradius breitet sich aus.
Tiefensuche(DFS): Von einem Knoten werden alle Nachfolger rekursiv durchsucht. Mehrere Verzweigungsmöglichkeiten werden zwischengespeichert.
DFS(G): Suche im Graph G den nächsten unbesuchten Knoten a. Rufe DFS-VISIT(a) auf.
DFS-VISIT(a): Finde rekursiv alle Nachfolgerknoten von a und markiere sie als durchsucht.

7.1 Minimaler Spannbaum

Kruskal-Algorithmus:

1. Sortiere alle Kanten aufsteigend nach Gewicht.
2. Wähle immer die nächst schwerere Kante, wenn sie keine Schleife bildet.

7.2 Kürzeste Pfade finden

Satz: Teilpfade von kürzesten Pfaden sind auch kürzeste Pfade!

Dijkstra-Algorithmus:
Menge S: Knoten mit dem Gewicht ihres kürzesten Pfades von s
Menge Q: Min-Prioritätswarteschlange mit den ungeprüften Knoten.
Relaxationsschritt: Überprüfe ob ein Umweg über einen anderen Knoten kürzer ist.
Muss für jede Kante nur genau einmal überprüft werden.

7.3 Bäume

... sind spezielle Graphen mit einer Wurzel, Zweigen und Blätter.
Er besitzt keine zyklischen Strukturen.

Begriffe:
Grad deg(v): Anzahl der Unterbäume(Äste)
Blatt: Knoten mit $deg(v) = 0$
Tiefe $d(v)$: Länge des Pfades von der Wurzel bis zum Knoten v.
Höhe $h(v)$: Längster Pfad von v zu einem Blatt.
Niveau: Knoten mit gleicher Tiefe.
Pfad: Folge von verbundenen Knoten von der Wurzel weg.

7.4 Binärbaum

... ist ein geordneter Baum und $\forall v \in V : deg(v) \leq 2$
Beim Binärbaum spricht man vom linken oder rechten Sohn.
Ein Binärbaum ist vollständig, wenn $\forall v : deg(v) = 2 \ \vee \ deg(v) = 0$

Falls Höhe k, dann gilt: $2^{k+1} - 1$ Knoten und 2^k Blätter.

Bedeutung: Verdopllung der Eingabegröße, mit logarithmische(ldn) Vergrößerung der Struktur.
Binärbaum als verkettete Liste: Knoten x mit $links[x], rechts[x], parent[x], key[x]$

7.5 Heap-Sort

Heap-Datenstruktur: fast vollständiger Binärbaum mit Indizierung von links nach rechts und von oben nach unten.
Max-Heap: Wurzel hat den größten Wert, Min-Heap: Wurzel hat den kleinsten Wert.

1. Erzeuge Max-Heap aus A

Wähle $|A|/2$ als Starknoten, da größter Knoten mit $deg > 1$ Betrachte Knoten i und seine beiden Kinder $2i$ und $2i + 1$

8 Entropie

... ist ein Maß für den mittleren Informationsgehalt bzw. Informationsdichte eines Zeichensystems(Alphabet).
Seltene Zeichen haben einen hohen Informationsgehalt.
Das i-te Zeichen z_i mit Auftrittswahrscheinlichkeit p_i:
Informationsgehalt $I(z_i) = -\log_2(p_i)$

8.1 Kompression

Huffmancode: Wahrscheinliche Zeichen werden mit weniger Bit kodiert als seltene. (z.B UTF-8)
Kommt in Klausur dran: Huffmancode!!!

© Emanuel Regnath