Laufzeitanalyse

Einige Formeln:

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, \text{ für } c \neq 1, \quad \sum_{i=0}^{n} i = \frac{n(n+1)}{2}, \quad \sum_{i=0}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$$

O-Notation:

$$f = O(g) \Leftrightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$
$$f = \Omega(g) \Leftrightarrow g = O(f)$$
$$f = \Theta(g) \Leftrightarrow f = O(g) \text{ und } g = O(f)$$
$$f = o(g) \Leftrightarrow \forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) < c \cdot g(n)$$
$$f = \omega(g) \Leftrightarrow g = o(f)$$

Zusammenhänge:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n)) \text{ und } f(n) = O(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ mit } 0 < c \in R \Rightarrow f(n) = O(g(n)) \text{ und } f(n) = \Theta(g(n)) \text{ und } f(n) = \Omega(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n)) \text{ und } f(n) = \Omega(g(n))$$

Einige Beziehungen:

$k \geq 1,\ l > 0,\ c > 1$

A	B	$A = \omega(B)$	$A = \Omega(B)$	$A = \Theta(B)$	$A = O(B)$	$A = o(B)$
$(\log n)^k$	n^l				wahr	wahr
n^k	c^n				wahr	wahr
$\sqrt{n}$	$n^{\sin n}$					
2^n	$2^{\frac{n}{2}}$	wahr	wahr			
$n^{\log m}$	$m^{\log n}$		wahr	wahr	wahr	
$\log(n!)$	$\log(n^n)$		wahr	wahr	wahr	

Beweis durch Induktion

Behauptung, Induktionsanfang, (Induktionsvoraussetzung), Induktionsschluss

Master-Theorem

$a \geq 1,\ b > 1,\ c > 0, f : N \to N$

$T(1) = c,\ T(n) = aT(\frac{n}{b}) + f(n)$ für $n > 1$

Falls $f(n) = O(n^{\log_b a - \varepsilon})$ für eine Konstante $\varepsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$

Falls $f(n) = \Theta(n^{\log_b a})$, dann $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

Falls $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für eine Konstante $\varepsilon > 0$ und $a \cdot f(\frac{n}{b}) \leq d \cdot f(n)$
für eine Konstante $d < 1$, dann $T(n) = \Theta(f(n))$

Spezialfälle:

$T(n) = aT(\frac{n}{b}) + n,\ T(1) = c$

$a > b \Rightarrow T(n) = \Theta(n^{\log_b a}),\ a = b \Rightarrow T(n) = \Theta(n \log n),\ a < b \Rightarrow T(n) = \Theta(n)$

$T(n) = aT(\frac{n}{b}) + 1,\ T(1) = c$

$a = 1 \Rightarrow T(n) = \Theta(\log n),\ a > 1 \Rightarrow T(n) = \Theta(n^{\log_b a})$

Sortieren

Untere Schranke für vergleichsbasierte Sortierverfahren

$\lfloor \log(n!) \rfloor \approx n \log n - n \log e$

Bubble-Sort

Vergleiche (= Compare/Exchange-Schritte):

$\sum_{i=1}^{n-1} (n - i) = \Theta(n^2)$

Modifikation: Verbesserung des best und average case: Falls in Runde i die letzte vorgenommene Vertauschung $(n\text{-}j, n\text{-}j\text{+}1)$ war mit $j > i$, dann werden die Runden i+1,…,j übersprungen, und erst mit Runde j+1 weitergemacht.

Vergleiche:

Worst Case: $\Theta(n)$, Average Case: $\Theta(n^2)$, Best Case: $\Theta(n^2)$, asymptotische Verbesserung nur in Spezialfällen

Insertion Sort

Durchlauf eines Arrays A[1:n] in n-1 Runden. In jeder Runde i (i=2,…n) wird für das aktuelle Array-Element A[i] die richtige Position im Teilarray A[1:i] gesucht. A[1:i-1] ist dabei schon sortiert. Es wird lineare Suche beginnend beim Element A[i-1] benutzt (d.h. Richtung i-1 → 1). Laufzeit: Best, Worst Case: Quadratisch Best Case: Linear

Quicksort: Laufzeiten: Worst Case: Quadratisch, Best Case: $O(n \log n)$

Durchschnitt: ca. $1{,}386n \log n - 2{,}846n$

Heapsort

1. Aufbau eines Heaps $O(n)$: Heapify* wird von n…1 für alle Array-Elemente aufgerufen

2. Heap sortieren $O(n \log n)$: n Mal ein ExtractMax durchführen. Dabei muss die Heapeigenschaft jeweils neu hergestellt werden, jedoch nur bis zu den Elementen, die vorher noch nicht aus dem Heaps entfernt worden sind.

*) Heapify: Aufruf für einen Knoten, dessen Kinder bereits Heaps sind, ggf. rekursiver Abstieg

Laufzeit: ca. $2n \log n$ Vergleiche, $n \log n$ Vertauschungen, insg. $O(n \log n)$

Merge-Sort

Funktionsweise:

Rekursiver Abstieg: Erst die erste Hälfte sortieren, dann die zweite. Anschließend beide sortierte Folgen mischen (mit Hilfe eines zusätzlichen Arrays) und zurückschreiben. (Das Mischen ist in Linearzeit möglich.)

Speicherplatz: Zusätzlich $O(n)$

Laufzeit:

$n \log n - n + 1$ Vergleiche, d.h. Merge-Sort hat Laufzeit $O(n \log n)$ mit Vorfaktor 1! (Quicksort: 1,386) Nachteil ist jedoch der zus. Speicherplatzbedarf, der sich nur schwer beseitigen lässt.

Bucket-Sort

Sortieren von Elementen aus einem Wertebereich Größe n durch aufteilen auf n Listen, die anschließend zusammengefügt werden.

Bucket-Sort kann auch verwendet werden, um Elemente aus der Menge $\{1,\ldots,m\}^k$ zu sortieren. Dazu werden die zu sortierenden Elemente n-adisch dargestellt. Bucket-Sort muss dann genau k-mal angewendet werden. (Dies ist möglich, da Bucket-Sort stabil sortiert.)

Vergleichsbaum

Binärer Baum, an dessen inneren Knoten Vergleiche der Form $x_i < x_j$ stehen. An den Blättern stehen die unterschiedlichen Permutationen der Folge $x_1,\ldots,x_n$, die sich durch den Pfad (die Vergleiche in den Knoten) von der Wurzel zum Blatt ergeben.

Sortierbaum

Vergleichsbaum, bei dem alle Eingaben, die dem gleichen Weg folgen, den gleichen Ordnungstyp haben.

Stabile Verfahren

Sortierverfahren, die die Reihenfolge gleicher Elemente nicht verändern

Medianberechnung

entspricht k-Selektion, d.h. Auswahl des kt-kleinsten Elements.
Kann für beliebige Werte für k mit $48n$ Vergleichen durchgeführt werden.
QuickSort erhielte damit eine Worst-Case-Laufzeit von $O(n \log n)$, jedoch mit schlechtem Vorfaktor.

Abstrakte Datentypen

Stack: Operationen: Create, IsEmpty, Push, Pop, Top

LIFO-Prinzip

Queue: Operationen: Create, IsEmpty, Enqueue, Dequeue, First

FIFO-Prinzip

Prioritäts-Warteschlange

IncreaseKey/DecreaseKey-Operationen implementierbar mit Laufzeit $O(\log n)$, wenn wir für jedes der Prio.-Schlange hinzufügbare Element den Index im Heap abspeichern (notwendig, um jedes Element in Zeit $O(\log n)$ lokalisieren zu können).

Lineare Listen

Heap: Ein Array heißt Heap, falls $A[i] \geq A[2i]$ und $A[i] \geq A[2i+1]$ für $2i$ bzw. $2i+1 \leq n$

Wörterbuch: Operationen: Create, Insert (Wert und Key), Delete, Search

Hashing

$U \mathrel{\widehat{=}}$ Wertemenge, $S \mathrel{\widehat{=}}$ Menge von Elementen im Wörterbuch, $|S| = n$

$m \mathrel{\widehat{=}}$ Kapazität des Arrays, $\alpha = \frac{n}{m}$

Killisions-Verwaltung

Hashing mit Verkettung

Gleiche Elemente werden in verketteter Liste gespeichert

Offene Adressierung

Lineares Sondieren (linear probing)

Quadratic Probing

Double Hashing: $h(x, i) = (h_1(x) + i \cdot h_2(x)) \bmod m$

Bäume, ein einziger Knoten habe Tiefe 0

Suchbäume

Beim Einfügen von $1,\ldots,n$ in beliebiger Reihenfolge:

	Einfügen	Tiefe des Baumes
best case	$n \log n$	$\lfloor \log n \rfloor$
worst case	$\frac{n(n+1)}{2}$	n-1
average case	$1,386 n \log n - \Theta(n)$	$\approx 4,31107 \log n - 1$

Das Minimun/Maximum eines Suchbaumes kann in $O(t)$, $t \triangleq$ Tiefe des Baumes gefunden werden. Um das k-kleinste/größte Element in $O(t)$ zu finden, werden die Operationen Insert, Search und Delete angepasst, so dass für jeden Knoten bekannt ist, wie viele Kinder er enthält.

AVL-Bäume

Formel, fass ein Knoten allein Tiefe 1 hat:
$\lceil \log(n+1) \rceil \leq Tiefe(T) < 1,4404 \log(n+2)$, T ist beliebiger AVL-Baum
Operationen Seach, Insert, Delete in Zeit $O(\log n)$
Seite n_i die Anzahl der Knoten in einem AVL-Baum T_i mit der Tiefe i. Es gilt: $n_i = 1 + n_{i-1} + n_{i-2}$

BB-Bäume

Balance $\mu(T) = \frac{\text{Anz. linker Kinder } + 1}{\text{Anz. aller Kinder } + 2}$
T heißt BB(α)-Baum, für $\alpha \in (0, \frac{1}{2})$, falls jeder Teilbaum T' von T Balance $\mu(T')$ mit $\alpha \leq \mu(T') \leq 1 - \alpha$ hat.
BB(α)-Bäume gibt es nur für $\alpha \leq \frac{1}{3}$, für $\alpha > \frac{1}{3}$ aber nicht.
Beste Wahl: $\alpha = 1 - \frac{1}{\sqrt{2}} \approx 0,2928$, die Tiefe ist dann $\leq 2 \log n$. Fibonacci-Bäume (also AVL-Bäume mit minimaler Anzahl von Knoten bei gegebener Tiefe) sind solche BB($1 - \frac{1}{\sqrt{2}}$)-Bäume.

Rot-Schwarz-Bäume

Rote Knoten haben nur schwarze Kinder
Für jeden Knoten v gilt: Jeder Weg von v zu einem Blatt enthält die gleiche Anzahl schwarzer Kinder

(a, b)-Bäume, bzw. B-Bäume (Bayer, McCraight), $a \leq \frac{b+1}{2}$

Jeder Knoten ($\neq$ Wurzel) enthält a 1 und höchstens b 1 Schlüssel. Die Wurzel enthält mind. 1 und höchstens b 1 Schlüssel. Schlüssel sind in jedem Knoten aufsteigend sortiert.
Ein Knoten mit l Schlüsseln enthält $l + 1$ Nachfolger. Der i-te Subbaum, $0 \leq i \leq l$, enthält nur Schlüssel aus dem Intervall (x_i, x_{i+1}) bzw. für i=0 nur Schlüssel $< x_1$ und für i=l Schlüssel $> x_l$.
Alle Blätter haben gleiche Tiefe. Für einen (a, b)-Baum mit n Elementen gilt: $\log_b(n+1) - 1 \leq Tiefe(T) \leq \log_a(\frac{n+1}{2})$ (Ein einziger Knoten habe Tiefe 0.)

In-, Pre-, Postorder

Preorder und Inorder zusammen beschreiben nur binäre Wurzelbäume eindeutig.
Preorder und Postorder beschreiben Wurzelbäume eindeutig.

Skip-Listen

Erwartete Laufzeit für Operationen Insert, Search, Delete für jede Eingabefolge: $O(\log n)$

Erwartete Höhe einer Skip-Liste für n Elemente: $O(\log n)$

Erwartete Anzahl von Zeigern in einer Skip-Liste mit n Elementen: $\leq 2n + O(\log n)$

Höhe $h(x)$ eines Elementes ist die Anzahl der Münzwürfe, bis das erste Mal eine 1 erscheint. Erwartete (durchschnittliche) Höhe jedes Elementes ist 2 (exakt).

Graphen, Laufzeiten bei Adjazenzlisten-Darstellung (sofern nicht anders angegeben)

Darstellungsmöglichkeiten:

Adjazenzliste mit Speicherplatz: $O(|V| + |E|)$, -matrix mit Speicherplatz: $O(|V|^2)$

Eigenschaften: gerichtet, (bei gerichteten: stark) zusammenhängend, zyklisch

Topologisches Sortieren

Algorithmus mit Laufzeit $O(|V| + |E|)$: Idee: Entferne nacheinander alle Quellen aus dem Graphen (durch Entfernen der letzten Quelle eines Graphen entsteht mindestens wieder eine neue, solange der Graph nicht leer ist). Sortierung ist Reihenfolge der Quellen. Alg.: Initialisiere eine Queue, die alle Quellen aus dem Graphen enthält. Speichere für jeden Knoten seinen Ingrad in einem Array. Nacheinander werden alle Quellen aus der Queue entfernt und für alle Nachfolger der Ingrad im Array um 1 reduziert. Falls dabei ein Knoten zu einem Ingrad 0 gelangt, füge ihn der Queue hinzu.

Breitensuche: Laufzeit: $O(|V| + |E|)$

Tiefensuche: Laufzeit: $O(|V| + |E|)$

Durchlaufe alle Knoten des Graphen. (DFS) Durchlaufe alle Nachfolger des akt. Knoten rekursiv. (DFS-Visit)

Kann verwendet werden für:

Überprüfung auf Kreisfreiheit

Ermittlung der Anz. der Zusammenhangskomponenten

Ermittlung der Kantentypen

Kantentypen feststellen

bei gerichteten Graphen:

Baumkante

Kanten, die im Algorithmus genutzt werden, um zu einem noch nicht besuchten Knoten zu kommen.

Vorwärtskante

Kanten (i, j), die keine Baumkanten sind, für die es aber einen Weg von j nach i über Baumkanten gibt.

Rückwärtskante

Kanten (i, j), die keine Baumkanten sind, für die es aber einen Weg von i nach j über Baumkanten gibt.

Kreuzkante

Sonstige Kanten

Bei ungerichteten Graphen macht eine Einteilung über Baum-Kanten und Nicht-Baumkanten hinweg keinen Sinn!

Minimaler Spannbaum

Algorithmus von Kruskal, $O(|E| \cdot \log |E|)$, falls Kanten sortiert: $O(|V| \cdot \log |V| + |E|)$

Sortierung aufsteigend nach Kantengewicht

Hinzufügen der jeweils kleinsten noch zur Verfügung stehenden Kante, die keinen Kreis schließt.
Kruskals Algorithmus kann (wenn mit allen möglichen Sortierungen angewandt) jeden möglichen MST ausgeben.
Implementierung:
UNION-FIND-Datenstruktur:
Operationen (Laufzeiten bei Speicherung der Sets mit Bäumen):
Make-Set $O(1)$, Union $O(\log|V|)$, Find-Set $O(\log|V|)$
Make-Set erzeugt Menge(Baum) aus nur einem einzigen Knoten
Union vereinigt zwei Mengen (Bäume)
Find-Set ermittelt die Menge (den Baum), in der (dem) sich ein Knoten befindet

Algorithmus von Prim, mit Fibonacci-Heap: $O(|V| \cdot \log|V| + O(|E|))$

Schrittweiser Aufbau des Spannbaums T. T wird dabei in jeder Runde um eine Kante mit minimalem Gewicht erkänzt, die T mit einem Knoten außerhalb von T verbindet.
gut für dünne Graphen

Weg-Komprimierung

Beispiel: Kruskal-Algorithmus
Die Teilbäume, die der Kurskal-Algorithmus aufbaut, werden so abgespeichert, dass von jedem Knoten sein Vorgänger-Knoten auffindbar ist (mit der Find-Set-Funktion). Ist der Vorgänger-Knoten gleich dem untersuchte Knoten, so handelt es sich um das namesgebende Element für den aktuellen Teilbaum.
Find-Set kann nun so modifiziert werden, dass
a) Seine Laufzeit nicht verändert wird
b) Zukünftige Find-Set-Operationen schneller ablaufen
Dies geschieht, indem für jeden Knoten, den Find-Set bei der Suche nach dem namensgebenden Element untersucht, das namensgebende Element zum Vorgänger gemacht wird.

Algorithmenentwurf

Divide & Conquer

Zerlegung in disjunkte Teilprobleme und rekursive Lösung

Dynamische Programmierung

Zerlegung in Teilprobleme, die nicht disjunkt sind. Lösung des Teilproblems lässt sich zur Lösung des Gesamtproblems erweitern.

Greedy

Bei der Zusammensetzung der Lösung wird in jedem Schritt grundsätzlich die Möglichkeit gewählt, die zu diesem Zeitpunkt am besten aussieht.

Backtracking

Durchlauf des gesamten Lösungsbaumes, siehe Branch & Bound

Branch & Bound

Anwendung bei NP-vollständigen Problemen
Tiefensuche im Lösungsbaum. Berechnung von Approximation (Lösung mit wahrscheinlich nicht optimalen Kosten) und Relaxierung (obere Schranke, z.B. fraktionaler Rucksack) in jedem Schritt.
Abbruch an einer Teillösung, wenn Relaxierung schlechter oder gleich bester bekannter Lösung.

On-Line

Der Algoithmus erhält seine Daten nur Schritt für Schritt. Trotzdem soll, über alle Eingabemöglichkeiten gesehen, die Lösung optimal sein.

(**Beispiel-**)**Algorithmen:**

Min. & Max. einer Menge gleichzeitig bestimmen (Laufzeit)
Beide größte Elemente einer Menge bestimmen (Laufzeit)
Maximum in zyklisch sortierter Liste
i-te Potenz einer Matrix in $o(i)$
Pattern Matching mit Hashing
Jobverteilung auf 2 Maschinen (dynamische Programmierung)
Rucksack-Problem: Dynamische Programmierung (optimal) vs. Greedy (gut)