Laufzeiten

Typische Laufzeitenklassen und zugehörige Algorithmen

	Sprechweise	Typische Algorithmen
O(1)	Konstant	
O(log n)	Logarithmisch	Suchen auf einer sortierten Menge
O(n)	Linear	Gute Sortierverfahren, z.B. mergesort, Skyline
O(n*log n)		
…		
$O(n^2)$	Quadratisch	Primitive Sortierverfahren (z.B. minSort)
$O(n^k)$, k >= 2	Polynomiell	
…		
$O(b^n)$, n > 1	Exponentiell	Backtracking-Algorithmen

→ exponentielle Algorithmen sind nur bei sehr kleinen Eingabegrößen, polynomielle Algorithmen generell noch handhabbar

Asymptotische Laufzeiten einer verketteten Liste und eines dynamischen Arrays bei n Elementen

	Einfach verkettete Liste (LinkedList)	Dynamisches Array (ArrayList)
boolean add(E elem)	O(1)	O(n)
void add (int index, E elem)	O(n)	O(n)
void clear()	O(1)	O(1)
boolean contains(Object o)	O(n)	O(n)
E get(int index)	O(n)	O(1)
boolean isEmpty()	O(1)	O(1)
E boolean remove(Object o)	O(n)	O(n)
E set(int index, E elem)	O(n)	O(1)
Int size()	O(1)	O(1)

Aufwand von SetAsList

boolean isEmpty()	O(1)
int size()	O(1)
boolean contains(Object o)	O(n)
boolean containsAll(Collection<?> c)	$O(n^2)$
boolean add(E elem)	O(n)
boolean addAll(Collection<?> c)	$O(n^2)$
boolean remove(Object o)	O(n)
boolean removeAll(Collection<?> c)	$O(n^2)$
boolean retainAll(SetAsList<E> c) → Schnittmenge	$O(n^2)$
boolean equals(Object o)	$O(n^2)$

Aufwand binäre Suche vs. lineare Suche

	binäre Suche	lineare Suche
best case	O(1)	O(1)
worst case	O(log n)	O(n)
Durchschnitt (erfolgreich)	O(log n)	O(n)
Durchschnitt(erfolglos)	O(log n)	O(n)

Vergleich von Mengenimplementierungen

	verkettete Liste (ohne Sortierung)	verkettete Liste (mit Sortierung)	dynamisches Array (mit Sortierung)	Bitvektor
boolean isEmpty()	O(1)	O(1)	O(1)	O(N)
int size()	O(1)	O(1)	O(1)	O(N)
boolean contains(Object o)	O(n)	O(n)	O(log n)	O(1)
boolean containsAll(Collection<?> c)	$O(n^2)$	O(n)	O(n)	O(N)
boolean add(E elem)	O(n)	O(n)	O(n)	O(1)
boolean addAll(Collection <? extends E> c)	$O(n^2)$	O(n)	O(n)	O(N)
boolean remove(Object o)	O(n)	O(n)	O(n)	O(1)
boolean removeAll(Collection<?> c)	$O(n^2)$	O(n)	O(n)	O(N)
boolean retainAll(setAsList<E> c)	$O(n^2)$	O(n)	O(n)	O(N)

boolean equals(Object o)	$O(n^2)$	O(n)	O(n)	O(N)

Aufwandsbetrachtung Bäume

	Suchen, Einfügen, Löschen
balancierter Baum	O(log n)
degenerierter Baum	O(n)
Berechnung von Höhe und Balancefaktor für den ganzen Baum (AVL-Baum)	O(n)
Neuberechnung bei einer Änderung (AVL-Baum)	$O(\log_2 n)$

Streutabellen

Aufwandsabschätzung offenes Hashing

worst case	h liefert immer denselben Wert, alle Elemente befinden sich in einer Liste	O(n)
best case	h streut perfekt, keine Kollisionen	O(1)
average case	1 plus durchschnittlicher Belegungsfaktor	O(1+n/m)

Min-Heap

Implementierung mit Prioritätswarteschlange	mit Halde (Array-Einbettung)	mit Liste
Entnahme Element	O(log k)	O(1)
Einfügen Element	O(log k)	O(k)

Überblick über Sortierverfahren

Vergleichsbasierte Sortierverfahren

		Best Case	Average Case	Worst Case	Stabil	in situ
einfache Sortierver-fahren	Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	leicht möglich	wenn mit Array
	InsertionSort	O(n)	$O(n^2)$	$O(n^2)$	x	wenn mit Array
	BubbleSort	O(n)	$O(n^2)$	$O(n^2)$	x	x

verfeinertes Auswählen	HeapSort	O(n*logn)	O(n*logn)	O(n*logn)		x
Divide & Conquer	MergeSort	O(n*logn)	O(n*logn)	O(n*logn)	x	I.d.R. nicht
	QuickSort	O(n*logn)	O(n*logn)	$O(n^2)$		x

Nicht-vergleichsbasierte Sortierverfahren

		Zeit	stabil	in situ
Sortieren durch Fachverteilen	BucketSort	O(n+m)	x	
	RadixSort	O(n*k)	x	

Partition einer Menge → union/find

- union verschmilzt zwei Teilmengen (→ Vereinigung ihrer Elemente)
- find stellt für ein gegebenes Element fest, zu welcher Teilmenge es gehört
- Aufwand abhängig von der Höhe der Bäume
  - Höhe der Bäume im Worst Case O(n)
  - Höhe der Bäume unter Beachtung der Größe bei union-Operation O(log n)
- Aufwand für m union-/find-Operationen auf n Elementen: O((m+n)log n)
- Aufwand mit Pfadkompression: O(m+n)

Überblick über Graphen und -algorithmen

Adjazenzmatrizen

- mit Laufzeit O(1) feststellbar, ob Kante von v nach w existiert
- hoher Platzbedarf von $O(n^2)$
- Auffinden aller k Nachfolger eines Knotens benötigt O(n) Zeit
- Initialisierung der Matrix benötigt Laufzeit $O(n^2)$

Adjazenzliste

- geringerer Platzbedarf von O(|V| + |E|)
- alle k Nachfolger eines Knotens sind in Zeit O(k) erreichbar

Adjazenzfeld

- geringerer Platzbedarf von O(|V| + |E|)

Graphendurchlauf

Tiefensuche (rekursive Implementierung)	O(|V| + |E|)
Breitensuche (Adjazenzlistendarstellung)	O(|V| + |E|)

Kürzeste Wege in Graphen

Algorithmus von Dijkstra	Implementierung mit Adjazenslisten und Halde	O((|V| + |E|)*log(|V|)	wenn alle Kanten dieselben Kosten haben, dann ist Breitensuche effizienter
	Implementierung mit Adjazenzmatrix	$O(\|V\|^2)$	
	billigster Pfad zwischen allen Knoten eines Graphen (minimaler Spannbaum)	O(|V|*(|V| + |E|)*log(|V|)	worst case: $O(\|V\|^3 * \log\|V\|)$
Algorithmus von Floyd	Implementierung mit Adjazenzmatrix	$O(\|V\|^3)$	

Minimaler Spannbaum

Algorithmus von Prim	O((|V| + |E|)*log(|V|)	Gesamtaufwand mit Optimierung (min-Halde als Priorityqueue zur Verwaltung der noch zu betrachtenden Kanten)
Algorithmus von Kruskal	O(|E| * log |E|)	

Überblick über geometrische Algorithmen

	Aufwand	//
Punkt-in-Polygon-Problem	O(n)	n Kanten
Konstruktion von Polygonen	$O(n*\log_2 n)$	n Knoten
Konvexe Hülle/Ausdehnungsalgorithmus	$O(n^2)$	
Konvexe Hülle /Einpackalgorithmus	$O(n^2)$	
Konvexe Hülle/Grahams Einpackalgorithmus	$O(n*\log_2 n)$	jeder Punkt wird höchstens einmal zur Hülle hinzugefügt und höchstens einmal wieder entfernt (O(n)). Das Sortieren

		der Winkel kostet O(n*log2n).
Ballung und nächstes Paar/naiver Algorithmus (wähle das Paar mit der kleinsten Distanz)	$O(n^2)$	
Ballung und nächstes Paar	$O(n*\log^2 n)$	$T(n) = 2 * T(n/2) + O(n*\log_2 n)$ Aufwand für jede Hälfte + Sortieren nach y-Koordinate Möglich ist auch $O(n*\log_2 n)$