

1 Invariante

- Vorm ersten Durchlauf erfüllt sein (Initialisierung, Induktionsanfang)
- Muss bei jedem Schleifendurchlauf erhalten bleiben (Erhaltung, Induktionsschritt)
- Invar nach Beendigung der Schleife zeigt Korrektheit (Terminierung)

1.1 Rekursionsgleichungen

- ineinander einsetzen
- Summe erkennen und zusammengefasst aufschreiben
- $k = \log(n)$ beachten und vereinfachen

1.2 Mastertheorem

1.2.1 Additives Mastertheorem

- a, b, c positiv $n = b^k$

$$T(n) \leq \begin{cases} c & n = 1 \\ a \cdot T(\frac{n}{b}) + c & n > 1 \end{cases}$$

$$\begin{aligned} T(n) &\leq c \cdot \frac{a}{a-1} n^{\log_b(a)} - \frac{c}{a-1} &= O(n^{\log_b(a)}) && \text{falls } a > 1 \\ T(n) &\leq c \cdot \frac{a}{a-1} n - \frac{c}{a-1} &= O(n) && \text{falls } a = b > 1 \\ T(n) &\leq c \cdot \log_b(n) + c &= O(\log(n)) && \text{falls } a = 1 \end{aligned}$$

1.2.2 allgemeines Mastertheorem

- $a, b, d, q \geq 1$

$$T(n) \leq \begin{cases} d & n \leq q \\ a \cdot T(\frac{n}{b}) + f(n) & n > q \end{cases}$$

$$\begin{aligned} f(n) &= O(n^{\log_b(a)-\epsilon}) \text{ mit } \epsilon > 0 & T(n) &= O(n^{\log_b(a)}) \\ f(n) &= \Theta(n^{\log_b(a)}) & T(n) &= O(n^{\log_b(a)} \log(n)) \\ f(n) &= \Omega(n^{\log_b(a)+\epsilon}) \text{ mit } \epsilon > 0, & T(n) &= \Theta(f(n)) \\ a \cdot f(\frac{n}{b}) &\leq \delta \cdot f(n), \delta < 1, n \rightarrow \infty \end{aligned}$$

2 O-Notation

2.1 Definition

- $f = O(g) \Leftrightarrow \exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) \leq cg(n)$
(f wächst asymptotisch höchstens so schnell wie g)

- $f = \Omega(g) \Leftrightarrow g = O(f)$
(f wächst asymptotisch mindestens so schnell wie g)
- $f = \Theta(g) \Leftrightarrow f = O(g) \text{ und } g = O(f)$
(f und g wachsen asymptotisch gleich schnell)
- $f = o(g) \Leftrightarrow \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) < cg(n)$
 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
(f wächst asymptotisch langsamer als g)
- $f = \omega(g) \Leftrightarrow g = o(f)$
(f wächst asymptotisch langsamer als g)

2.2 Eigenschaften

- $f = o(g) \Rightarrow f = O(g)$
- $f = o(g) \text{ und } h = o(g) \Rightarrow f + h = o(g)$ (auch $O, \Omega, \omega, \Theta$)
- $f = o(g) \text{ und } h = o(g) \Rightarrow f \cdot h = o(g^2)$ (auch $O, \Omega, \omega, \Theta$)

2.3 Reihenfolge

$$c < \log(n) < n^{\frac{1}{k}} < n < n \log(n) < n^2 < n^k < 2^n$$

3 Sortieralgorithmen

3.1 Bubblesort

- Jeden Durchlauf wird das größte Element auf die n-ite Stelle getauscht. Jeder Vergleich ggf ein Swap.

3.2 Insertionsort

- Key wird in sortiertes Array eingeordnet. Key wird gemerkt, falls kleiner wird das größere Element auf Pos von Key kopiert aber Key wird erst kopiert, wenn die richtige Stelle gefunden worden ist.

3.3 Mergesort

- Teile Array bis auf ein Element Array und sortiere beim rekursiven zusammenfügen. Geteilt wird p bis q, q+1 bis r. Merge vergleicht erste Pos von den Arrays und fügt immer das kleinere ins Zielarray ein.

3.4 Quicksort(**A**,**p**,**r**)

- Teile Array nach Pivotelement und füge Pivot dann an Grenze ein. Dann partition bis q-1 und

4 Heap

4.1 Definition

- $Heap := A[i] \geq A[2i] \text{ und } A[i] \geq A[2i+1]$
- jeder Knoten hat mindestens so großen Wert wie seine Kinder
- Wird als binärer Baum dargestellt.

4.2 Eigenschaften

- Baumtiefe $\lfloor \log(n) \rfloor$
- $A[\lfloor \frac{n}{2} \rfloor + 1]$ bis $A[n]$ sind Kinder
- Maximum ist die Wurzel $A[1]$

4.3 Heapify

- Tausche mit größtem Kind
- wird auf getauschten Knoten erneut aufgerufen bis er richtig steht

4.4 Build-Heap

- Bei uns meist MaxHeap
- Heapify auf $A[\lfloor \frac{n}{2} \rfloor]$ *downto* $A[1]$
- $BC : \Theta(n)$ $AV : \Theta(n)$ $WC : \Theta(n)$
- #Vertauschungen $\leq \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (\log(n) - \log(i))$
- #Vergleiche \leq #Vertauschungen

4.5 Heap-Sort

- Build-Heap
- Sort-Heap
Vertausche $A[1]$ und $A[i]$
Danach Heapify($A[i \dots i-1]$)

4.6 Bucket-Sort

- Hänge a_j an Liste $L[a_j]$ an. Gebe alle Listen aus
- stabil
- Zahlen aus $\{1, \dots, m\}$ und $O(n+m)$ und braucht $O(n+m)$ Platz

5 Dynamische Arrays

- Falls Array A nicht mehr ausreicht ($n \neq w$). Verdoppelt Array
- Falls $\frac{1}{4}n > 0$ Halbiere Array

6 Stack

- Empty(S), Pop(S), Push(S), Top(S)(Pos)

7 Queue

- Enqueue, Dequeue, Head(Q), Tail(Q)(erste freie Pos)
- falls Array von vorne startete bei 1, falls frei

8 Doppelt verkettete Listen

- Head(L), Insert(L, x)(hängt vorne dran), Remove(L, x)
- key(x), next(x), prev(x)
- falls next(x) = nil, x letztes Element

9 Skiplisten

- verschiedene Niveaus
- perfekte hat $\lceil \log(n) \rceil$ Niveaus (ermöglicht binäre Suche)
- $\text{left}(v)$, $\text{right}(v)$, $\text{down}(v)$, $\text{up}(v)$, $\text{Search}(L,x)$, $\text{Insert}(L,v)$ (mit RandomHight)
- Niveau 0, wo jeder Knoten ist (2^0)
- Niveau k beinhaltet jeden 2^k -ten Knoten

10 Binäre Suchbäume

10.1 Binäre Suchbäume

- $\text{lc}[x]$, $\text{rc}[x]$, $\text{p}[x]$, $\text{root}[x]$
- Inorder-Tree-Walk(x) gebe Knoten sortiert, nach abgeben, aus.
Gehe am Ende von ganz rechts zur Wurzel zurück. $\Theta(n)$
- Baumsuche(x,k) meist mit $x = \text{root}[T]$
- Min/Max linkestes/rechteste Element
- Nachfolger(x) linkestes Element im rechten Teilbaum. Wenn nicht verfügbar im Baum aufsteigen
- Delete(x), x hat 2 Kinder, Nachfolger von x wird verschoben, ggf wiederholen

10.2 Balancierte Suchbäume

- Höhe höchstens $2\log(n+1) - 2$
- Rotation ändern nur Höhe
- $\text{Rechtsrota}(T,x)$, $\text{Linksrota}(T,x)$
- Balance(t):
Falls $\text{lc}[t]$ und $\text{rc}[\text{lc}[t]]$ größer dann $\text{Linksrota}(\text{lc}[t])$, sonst $\text{Rechtsrota}(t)$
Falls $\text{rc}[t]$ und $\text{lc}[\text{rc}[t]]$ größer dann $\text{Rechtsrota}(\text{rc}[t])$, sonst $\text{Linksrota}(t)$

11 Hashing

11.1 Geschlossene Adressierung

- Kollisionsauflösung durch Listen
- Suchen/Löschen AV: $\Theta(1 + \alpha)$, Falls $m = \Theta(n)$ $\cdot \Theta(1)$

11.2 Offene Adressierung

- (nächste) freie Stelle in der Hashtabelle
- Lineares Hashing: $h(k, i) = (h'(k) + i) \bmod m$
- Quadratisches Hashing: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
- Delete problematisch, deshalb offene Adressierung bei Anwendungen ohne Delete nutzen
- falls zu viele deleted, dann neu hashen in größerer Tabelle(amortisierte Laufzeit)

11.3 Kuckuckshashing

- 2 Hashfunktionen mit eine Tabelle
- Insert: Falls belegt, neuen Wert einfügen und alten mit anderer Funktion neu hashen
- max $d \log(n)$ Hashversuche

12 Graphentheorie

12.1 Adjazenzmatrix/liste

- Zeile: von, Spalte: nach
- Zeile: von, Einträge: erreichende Knoten (einfach verkettete liste)

12.2 SSSP

12.2.1 BFS

- berechnet Abstand von allen Knoten zu s in einem ungewichteten Graphen
- $d[u]$ ist der Abstand am Anfang ∞
- $\pi[u]$ ist der Vorgänger am Anfang NIL
- $\text{Color}[u] = \text{weiß, grau, schwarz}$ am Anfang weiß
- Queue zum Speichern der grauen Knoten
- BFS entdeckt alle Knoten $v \in V$, die von s aus erreichbar sind
- entdeckt die Zusammenhangskomponenten
- $O(|V| + |E|)$

12.2.2 DFS

- Neue Knoten vom zuletzt gefundenen Knoten entdeckt. Falls alle $\text{adj}[v]$ entdeckt gehe zu $\pi[v]$
- löst nicht SSSP, bildet Spannbaum
- $d[u]$ ist der Abstand am Anfang ∞
- $\pi[u]$ ist der Vorgänger am Anfang NIL
- $\text{Color}[u] = \text{weiß, grau, schwarz}$ am Anfang weiß
- Stack zum Speichern der grauen Knoten
- Baumkante: rot, Rückkante: grün (auf grauen), SonstigeKante: blau (auf schwarzen)
- G enthält einen Kreis $\Leftrightarrow \text{DFS}(G)$ erzeugt mindestens eine Rückwärtskante
- DAG \Leftrightarrow es existiert eine topologische Sortierung (alle Kanten einzeichnen)
- TopologischesSortieren(G) legt eine Liste an und fügt die schwarzen Knoten vorne an

12.3 Dijkstra

- kein negativer Kreis aber gewichtet
- setze alle Knoten auf ∞ und update alle $\text{adj}(u)$ mit dem Gewicht, falls geringer als aktueller Wert
- Laufzeit mit Heaps: $O((|V| + |E|)\log(|V|))$

13 Dynamische Programmierung

- Berechne die Funktionswerte iterativ und bottom-up und nutze Teilergebnisse

13.1 Bellman-Ford

- ohne negative Zyklen erhält man kürzesten Pfad
- Laufzeit $O(|V|^2 + |V||E|)$ und Speicherbedarf $O(|V|^2)$
- $|V| - 1$ Zeilen und Knoten als Spalten
- Gucke, ob neuer $\text{Weg} + E$ kleiner ist als alter Weg

13.2 APSP

13.2.1 Floyd-Warshall(W,n)

- Keine negative Zyklen
- Initialisiere Adjazenzmatrix mit 1 Nachbarknoten
- halte k mal Spalte K und Zeit K fest und guck, ob geringere Summen entstehen
- Laufzeit $O(|V|^3)$ mit Platz $O(|V|^2)$

14 Greedy

14.1 Allgemein

- das zum Zeitpunkte beste auswählen, was man machen kann
- EDF für Scheduling Probleme

14.2 U-F-Kruskal

- gewichteter, ungerichteter, zusammenhängender Graph errechnet daraus einen MST
- Nimm Kante mit geringstem Gewicht, die zwei Bäume im aktuellen aufspannenden Wald verbindet und füge diese zu A hinzu
- rote Kante: Spannbaum, grüne Kante: ungebraucht

14.3 Prim

- Lässt einen Wald wachsen. Immer nur von bekannten Knoten aus auswählen
- Laufzeit $O(|E|\log(|V|))$

15 Uebersicht

15.1 Sortieralgos

Algo	Art	InPlace	Stabilität	BC	AC	WC
Bubble	Vergleich	Ja	Ja	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion	Inkrementell	Ja	Ja	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge	$D\&C$	Nein	Ja	$\Theta(n\log(n))$	$\Theta(n\log(n))$	$\Theta(n\log(n))$
Quick	$D\&C$	Ja	Nein	$\Theta(n\log(n))$	$\Theta(n\log(n))$	$\Theta(n^2)$
Heap	$D\&C$	Ja	Nein	$\Theta(n\log(n))$	$\Theta(n\log(n))$	$\Theta(n\log(n))$
Bucket	Inkrementell	Nein	Ja	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n^2)$

15.2 Graphenalgos

Algo	Laufzeit	Bedingung	Return	Funktion
BFS(G,s)	$ V + E $	keine Gewicht	d[v]	
DFS(G,s)	$ V + E $	keine Gewicht	Tiefenswald	
Dijkstra(G,w,s)	$(V + E) \log(V)$	no neg Gewicht	SSSP	vom min
BellmanFord(G,s)	$O(V ^2 + V \cdot E)$	no neg Zyklen	SSSP	Jede Iteration Knoten abg
Floyd-Warshall(W,n)	$O(V ^3)$	no neg Zyklen	APSP	Spalte/
Kruskal(G)	$ E \cdot \log(E)$	unger, zsmh	MST	n
Prim(G,r)	$O(E \log(V))$	unger, zsmh	MST	

15.3

•

16 Rechentricks

- Arithmetische Reihe $\sum_{i=1}^n k = \frac{n(n+1)}{2}$
- Geometrische Reihe $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$
- logarithmus!!! bsp laufzeit aus probe 1
- Stirling'sche Formel $n! \approx \left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n}$
 $\Rightarrow \log(n!) \approx n \log(n) - n \log(e)$
- $\log\left(\left(\frac{P}{Q}\right)^k\right) = k \cdot \log\left(\frac{P}{Q}\right) = k \cdot (\log(P) - \log(Q))$
- $\log_a(P) = \frac{\log_b(P)}{\log_b(a)}$
- $a^{\log_b(n)} = n^{\log_b(a)}$
- $4^{\log(n^2)} = 4^{2\log(n)} = 2^{4\log(n)} = n^{4\log(2)} = n^4$
- Set Cover problem