1 Invariante

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

1.1 allgemeines Mastertheorem

- $a, b, d, q \geq 1$
- $T(n) \leq \begin{cases} d & n \leq q \\ a \cdot T(\frac{n}{b}) + f(n) & n > q \end{cases}$

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

2 O-Notation

2.1 Definition

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

2.2 Eigenschaften

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

2.3 Reihenfolge

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

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 gößere 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

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

4.2 Eigenschafen

- 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 richtit 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 $\leq$ 2 #Vertauschungen

4.5 Heap-Sort

- Build-Heap
- Sort-Heap
  $VertauscheA[1]undA[i]$
  Danach Heapify(A[i ... i-1])

4.6 Bucket-Sort

- Hänge a_j an Liste L[a_j] an. Gebe alle Listen aus

5 Dynamische Arrays

- Falls Array A nicht mehr ausreicht (n¿w). Verdoppel Array
- Falls $\frac{1}{4}$ und $n > 0$ 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 vonn starte 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)
- left(v), right(v), down(v), up(v), Search(L,x), Insert(L,v) (mit RandomHight)
- Niveau 0, wo jeder Knoten ist (2^0)
- Niveau k behinaltet jeden 2^k-ten Knoten

10 Binäre Suchbäume

10.1 Binäre Suchbäume

- lc[x], rc[x], p[x],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 = root[T]
- Min/Max linkestes/rechtestes 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 (AVL)

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

11 Hashing

11.1 Geschlossene Adressierung

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

11.2 Offene Adressierung

- (nächste) freie Stelle in der Hashtabelle
- Lineares Hashing: $h(k,i) = (h'(k) + i) mod\ m$
- Quadratisches Hashing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) mod\ m$
- 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 je einer Tabelle
- Insert: Falls belegt, neuen Wert einfügen und alten mit anderer Funktion neu hashen
- max $dlog(n)$ 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

- Queue zum Speichern der grauen Knoten
- BFS entdeckt alle Knoten $v \in V$, die von s aus erreichbar sind
- entdeckt die Zusammenhangskomponenten

12.2.2 DFS

- Neue Konten vom zuletzt gefundenen Knoten entdeckt. Fals alle adj[v] entdeckt gehe zu π[v]
- löst nicht SSSP, bildet Spannbaum
- Stack zum Speichern der grauen Knoten
- Baumkante: rot, Rückkante: grün (auf grauen), Sonstigekante: blau (auf schwarzen)
- G enthält einen Kreis $\Leftrightarrow$ DFS(G) erzeugt mindestens eine Rückwärtskante
- DAG $\Leftrightarrow$ es existiert eine topologische Sortierung (alle Kanten einzeichnen)
- TopoligischesSortieren(G) legt eine Liste an und fügt die schwarzen Knoten vorne an

12.2.3 Dijkstra

- setze alle Knoten auf ∞ und update alle adj(u) mit dem Gewicht, dalls geringer als aktueller Wert

12.2.4 Bellman-Ford

- $|V| - 1$ Zeilen und Knoten als Spalten
- Gucke, ob neuer $Weg + E$ kleiner ist als alter Weg

12.3 APSP

12.3.1 Floyd-Warshall(W,n)

- Initialisiere Adjazenzmatrix mit 1 Nachbarknoten
- halte k mal Spalte K und Zeit K fest und guck, ob geringere Summen entstehen

13 Greedy

13.1 Allgemein

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

13.2 U-F-Kruskal

- Nimm Kante mit geringstem Gewicht, die zwei Bäume im aktuellen aufspannenden Wald verbindet und füge diese zu A hinzu

13.3 Prim

- Lässt einen Wald wachsen. Immer nur von bekannten Knoten aus auswählen

14 Uebersicht

14.1 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	$\Theta(nlog(n))$	$\Theta(nlog(n))$	$\Theta(nlog(n))$
Quick	$D\&C$	Ja	Nein	$\Theta(nlog(n))$	$\Theta(nlog(n))$	$\Theta(n^2)$
Heap	$D\&C$	Ja	Nein	$\Theta(nlog(n))$	$\Theta(nlog(n))$	$\Theta(nlog(n))$
Bucket	Inkrementell	Nein	Ja	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n^2)$

14.2 Graphenalgos

Algo	Laufzeit	Bedingung	Return	Funktionsweise
BFS(G,s)	$\|V\|+\|E\|$	keine Gewicht	d[v]	Queue
DFS(G,s)	$\|V\|+\|E\|$	keine Gewicht	Tiefenswald	Stack
Dijkstra(G,w,s)	$(\|V\|+\|E\|)log(\|V\|)$	no neg Gewicht	SSSP	vom min Koten aus updaten
BellmanFord(G,s)	$O(\|V\|^2+\|V\|\cdot\|E\|)$	no neg Zyklen	SSSP	Jede Iteration Knoten abgehen, ggf next[Knoten] updaten
Floyd-Warshall(W,n)	$O(\|V\|^3)$	no neg Zyklen	APSP	Spalte/Reihe festhalten
Kruskal(G)	$\|E\|\cdot log(\|E\|)$	unger, zsmh	MST	min Kante
Prim(G,r)	$O(\|E\|log(\|V\|)$	unger, zsmh	MST	wachsen

15 Rechentricks

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