1 Laufzeit

Notations	Asymptotischer Vergleich	Formale Definition	Grenzen
$f(n) \in \omega(g(n))$	$f(n)$ wächst schneller als $g(n)$	$\forall c \exists n_0 \forall n > n_0 f(n) > c \cdot g(n)$	$\lim\sup_{n\to\infty} \frac{f}{g} = \infty$
$f(n) \in \Omega(g(n))$	$f(n)$ wächst min. so schnell wie $g(n)$	$\exists c \exists n_0 \forall n > n_0 c \cdot f(n) \leq g(n)$	$0 < \lim\inf_{n\to\infty} \frac{f}{g} \leq \infty$
$f(n) \in \Theta(g(n))$	$f(n)$ und $g(n)$ wachsen gleich schnell	$f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n))$	$0 < \lim_{n\to\infty} \frac{f}{g} < \infty$
$f(n) \in \mathcal{O}(g(n))$	$f(n)$ wächst max. so schnell wie $g(n)$	$\exists c \exists n_0 \forall n > n_0 f(n) \leq c \cdot g(n)$	$0 \leq \lim\sup_{n\to\infty} \frac{f}{g} < \infty$
$f(n) \in o(g(n))$	$f(n)$ wächst langsamer als $g(n)$	$\forall c \exists n_0 \forall n > n_0 c \cdot f(n) < g(n)$	$\lim_{n\to\infty} \frac{f}{g} = \infty$

1.1 Vergleich

1	$\log^* n$	$\log n$	$\log^2 n$	$\sqrt[3]{n}$	$\sqrt{n}$	n	n^2	n^3	$n^{\log n}$	$2^{\sqrt{n}}$	2^n	3^n	4^n	$n!$	2^{n^2}

Transitivität

$f_1(n) \in \mathcal{O}(f_2(n)) \wedge f_2(n) \in \mathcal{O}(f_3(n))$
$\Rightarrow f_1(n) \in \mathcal{O}(f_3(n))$

Summen

$f_1(n) \in \mathcal{O} f_3(n)) \wedge f_2(n) \in \mathcal{O}(f_3(n))$
$\Rightarrow f_1(n) + f_2(n) \in \mathcal{O}(f_3(n))$

Produkte

$f_1(n) \in \mathcal{O}(g_1(n)) \wedge f_2(n) \in \mathcal{O}(g_2(n))$
$\Rightarrow f_1(n) \cdot f_2(n) \in \mathcal{O}(g_1(n) \cdot g_2(n))$

1.2 Master-Theorem

Sei $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ mit $f(n) \in \Theta(n^c)$ und i $T(1) \in \Theta(1)$. Dann gilt

$$T(n) \in \begin{cases} \Theta(n^c) & \text{wenn } a < b^c, \\ \Theta(n^c \log n) & \text{wenn } a = b^c, \\ \Theta(n^{\log_b(a)}) & \text{wenn } a > b^c. \end{cases}$$

1.2.1 Monome

- $a \leq b \Rightarrow n^a \in \mathcal{O}(n^b)$
- $n^a \in \Theta(n^b) \Leftrightarrow a = b$
- $\sum_{v \in V} deg(v) = \Theta(m)$
- $\forall n \in \mathbb{N} : \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$

- $\sum_{i=a}^{b} c^i \in \begin{cases} \Theta(c^a) & \text{wenn } c < 1, \\ \Theta(c^b) & \text{wenn } c > 1, \\ \Theta(b-a) & \text{wenn } c = 1. \end{cases}$
- $\log(ab) = \log(a) + \log(b)$
- $\log(\frac{a}{b}) = \log(a) - \log(b)$
- $a^{\log_a(b)} = b$
- $a^x = e^{ln(a) \cdot x}$
- $\log(a^b) = b \cdot \log(a)$
- $\log_b(n) = \frac{\log_a(n)}{\log_a(b)}$

2.1 Heaps

Bin.-Heap	Laufzeit
push(x)	$\mathcal{O}(\log n)$
popMin()	$\mathcal{O}(\log n)$
decPrio(x, x')	$\mathcal{O}(\log n)$
build([ℕ; n])	$\mathcal{O}(n)$

- linkes Kind: $2v + 1$
- rechts Kind: $2v + 2$
- Elternknoten: $\lfloor \frac{v-1}{2} \rfloor$

2 Sortieren

Algorithmus	best case	average	worst	Stabilität
Insertion-Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	stabil
Bubble-Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	stabil
Merge-Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	stabil
Quick-Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	i.A. nicht stabil
Heap-Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	nicht stabil
Bucket-Sort	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	stabil $e \in [0, m)$
Radix-Sort	$\Theta(c \cdot n)$	$\Theta(c \cdot n)$	$\Theta(c \cdot n)$	stabil $e \in [0, n^c)$

3 Datenstrukturen

3.1 Listen

Operation	DLL	SLL	Array	Erklärung(*)
first	1	1	1	
last	1	1	1	
insert	1	1*	n	nur insertAfter
remove	1	1*	n	nur removeAfter
pushBack	1	1	1*	amortisiert
pushFront	1	1	n	
popBack	1	n	1*	amortisiert
popFront	1	1	n	
concat	1	1	n	
splice	1	1	n	
findNext	n	n	n	

3.2 Hash-Tabelle

$\mathcal{H}$ heißt **universell**, wenn für ein zufälliges gewähltes $h \in \mathcal{H}$ gilt: $U \to \{0, ..., m-1\}$
$\forall k, l \in U, k \neq l : Pr[h(k) = h(l)] = \frac{1}{m}$
$h_{a,b}(k) = ((a \cdot k + b) \mod p) \mod m$

3.3 Graphen

Algorithmus	Laufzeit
BFS/DFS	$\Theta(n+m)$
topoSort	$\Theta(n)$
Kruskal	$\Theta(m \log n)$
Prim	$\Theta((n+m) \log n)$
Dijksta	$\Theta((n+m) \log n)$
Bellmann-Ford	$\Theta(nm)$
Floyd-Warshall	$\Theta(n^3)$

3.3.1 DFS

Kante	DFS	FIN
Vorkante	klein → groß	groß → klein
Rückkante	groß → klein	klein → groß
Querkante	groß → klein	groß → klein
Baumkante	klein → groß	groß → klein

3.4 Bäume

3.4.1 Heap

Priorität eines Knotens $\geq$ ($\leq$) Priorität der Kinder. **BubbleUp**, **SinkDown**. **Build** mit **sinkDown** beginnend mit letztem Knoten der vorletzten Ebene weiter nach oben. **decPrio** entweder updaten, Eigenschaft wiederherstellen; löschen, mit neuer Prio einfügen oder Lazy Evaluation.

3.4.2 (ab)-Baum

Balanciert. **find**, **insert**, **remove** in $\Theta(logn)$. Zu wenig Kinder: **rebalance** / **fuse**. Zu viele Kinder: **split**.

Linker Teilbaum $\leq$ Schlüssel k $<$ rechter Teilbaum

Unendlich-Trick, für Invarianten.

3.5 Union-Find

Rang: höhe des Baums, damit ist die Höhe h mind. 2^h Knoten, h $\in \mathcal{O}(\log n)$. Union hängt niedrigen Baum an höherrängigen Baum. Pfadkompression hängt alle Knoten bei einem **find** an die Wurzel.

4 Amortisierte Analyse

4.1 Aggregation

Summiere die Kosten für alle Operationen. Teile Gesamtkkosten durch Anzahl Operationen.

4.2 Charging

Verteile Kosten-Tokens von teuren zu günstigen Operationen (Charging). Zeige: jede Operation hat am Ende nur wenige Tokens.

4.3 Konto

Günstige Operationen bezahlen mehr als sie tatsächlich kosten (ins Konto einzahlen). Teure Operationen bezahlen tatsächliche Kosten zum Teil mit Guthaben aus dem Konto. **Beachte: Konto darf nie negativ sein!**

4.4 Potential (Umgekehrte Kontomethode)

Definiere Kontostand abhängig vom Zustand der Datenstruktur (Potentialfunktion)

amortisierten Kosten = tatsächliche Kosten $+\Phi(S_{\text{nach}}) - \Phi(S_{\text{vor}})$

5 Pseudocode

```
DFS(Graph G, Node v)
mark v
dfs[v] := dfsCounter++
low[v] := dfs[v]
for u ∈ N(v) do
    if not marked u then
        dist[u] := dist[v] + 1
        par[u] := v
        DFS(G, u)
        low[v] := min(low[v],
          low[u])
    else
        low[v] := min(low[v],
          dfs[u])
fin[v] := fin++
```

```
BFS(Graph G, Start s, Goal z)
Queue Q := empty queue
Q.push(s)
s.layer = 0
while Q ≠ ∅ do
    u := Q.pop()
    for Node v in N(u) do
        if v.layer = −∞ then
            Q.push(v)
            v.layer = u.layer +
              1
        if v = z then
            return z.layer
```

```
topoSort(Graph G)
fin := [∞; n]
curr := 0
for Node v in V do
    if v is colored then
        DFS(G,v)
return V sorted by decreasing fin
```

```
Dijkstra(Graph G, Node s)
d := [∞; n]
d[s] := 0
PriorityQueue Q := empty
  priority queue
for Node v in V do
    Q.push(v, d[v])
while Q ≠ ∅ do
    u := Q.popMin()
    for Node v in N(u) do
        if d[v] > d[u] + len(u,
          v) then
            d[v] := d[u] +
              len(u, v)
            Q.decPrio(v, d[v])
```

```
Kruskal(Graph G)
U := Union-Find(G.v)
PriorityQueue Q := empty
for Edge e in E do
    Q.push(e, len(e))
while Q ≠ ∅ do
    e := Q.popMin()
    if U.find(v) ≠ U.find(u)
      then
        L.add(e)
        U.union(v, u)
```

```
BellManFord(Graph G, Node s)
d := [∞, n]
d[s] := 0
for n-1 iterations do
    for (u, v) ∈ E do
        if d[v] > d[u] + len(u,
          v) then
            d[v] := d[u] +
              len(u, v)
for (u, v) ∈ E do
    if d[v] > d[u] + len(u, v)
      then
        return negative cycle
return d
```

```
Prim(Graph G)
Priority Queue Q := empty
p := [0; n]
for Node v in V do
    Q.push(v, ∞)
while Q ≠ ∅ do
    u := Q.popMin()
    for Node v in N(u) do
        if v ∈ Q ∧ (len(u, v) <
          Q.prio(v)) then
            p[v] = u
            Q.decPrio(v, len(u,
              v)
```

```
FloydWarshall(Graph G)
D := [∞, n × n]
for (u, v) ∈ E do
    D[u][v] := len(u, v)
for v ∈ V do
    D[v][v] := 0
for i ∈ 1, ..., n do
    for (u,v) ∈ V × V do
        D[u][v] := min(D[u][v],
          D[u][v_i] + D[v_i][v])
return D
```