Zusammenfassung für die mündliche Diplomprüfung im Fach Höhere Algorithmik

Naja von Schmude

18. Oktober 2010

Inhaltsverzeichnis

Dies ist meine Zusammenfassung des Lernstoffs für die mündliche Diplomprüfung im Bereich theoretische Informatik an der Freien Universität Berlin. Sie umfasst den Stoff der Vorlesung Höhere Algorithmik aus dem Wintersemester 2008/2009 bei Prof. Alt, es wurde entsprechend mit der dritten Auflage „Introduction to Algorithms“ von Thomas H. Cormen et al. ergänzt.

1 NP-Vollständigkeit

1.1 Definitionen

- Ein Problem ist in Klasse P, wenn es in polynomieller Zeit lösbar ist. Formeller:

$$P = \{L | \exists \text{ Algorithmus } A, \text{ der } L \text{ in polynomieller Zeit entscheidet}\}$$

- Ein Problem ist in Klasse NP, wenn es in polynomieller Zeit verifizierbar ist, d.h. wenn unter Vorgabe eines Zeugen geprüft werden kann, ob dieser das Problem erfüllt. Formeller:

$$NP = \{x | \exists \text{ Zertifikat } w \text{ mit } |w| = O(|x|^c), \text{ so dass A(x,w)} = 1\}$$

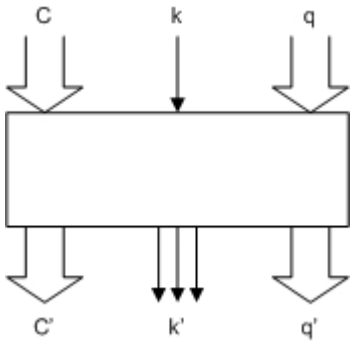
- Ein Problem ist in Klasse *NPC* (NP vollständig), wenn es in NP liegt und zudem NP-schwer ist. Ein Problem L ist NP-schwer, wenn $L' \leq_P L$ für alle $L' \in NP$ gilt.

- Man unterscheidet *Optimierungsprobleme*, bei denen man nach der „besten" Lösung sucht und *Entscheidungsprobleme*, die lediglich „ ja" oder „nein" als Antwort zulassen. Dabei kann man ein Optimierungsproblem in ein Entscheidungsproblem umwandeln, in dem man danach fragt, ob man das Problem für einen Wert k lösen kann.

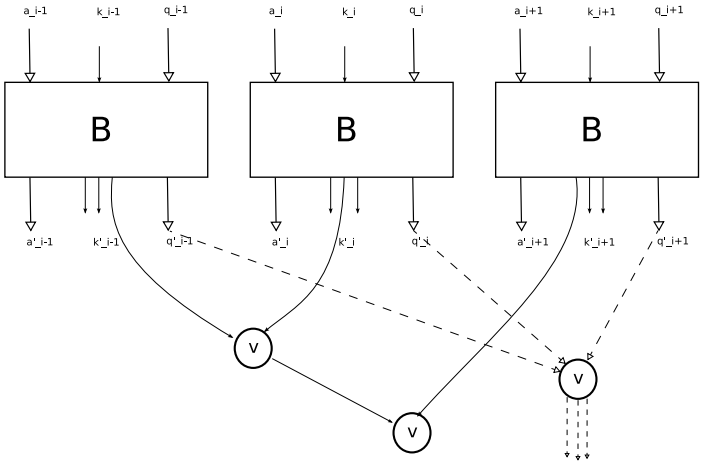
- *Reduktion* nennt man das Verfahren, bei dem man ein Entscheidungsproblem A in ein anderes Entscheidungsproblem B überführt um es einfacher lösen zu können oder dadurch etwas über die Komplexität auszusagen. Dazu wandelt man den Input für Problem A durch eine Transformationsfunktion so um, dass es als Input für Problem B fungiert. Dabei muss die Transformationsfunktion in polynomieller Zeit durchführbar sein und das Ergebnis, was B nach der Transformation liefert, genau dann wahr sein, wenn es nur für A auch wahr wäre.
  Formeller: Die Spräche L_1 ist auf die Sprache L_2 reduzierbar ($L_1 \leq_P L_2$), wenn eine in polynomieller Zeit berechenbare Funktion f existiert, so dass für alle x gilt $x \in L_1 \Leftrightarrow f(x) \in L_2$.
  - Falls B in P und eine Transformationsfunktion existiert, die obige Bedingung erfüllt, dann ist A ebenfalls in P. ($A \leq_P B$ mit $B \in P$ impliziert $A \in P$).
  - Falls A in NPC und wir eine Transformationsfunktion finden können, dann ist B ebenfalls in NPC. ($A \leq_P B$ mit $A \in NPC$ impliziert B ist NP-schwer. Wenn dann noch $B \in NP$ gilt $B \in NPC$.)

1.2 NPC Probleme

CSAT Die Frage ist hier, ob ein Schaltkreis erfüllbar ist. Wir zeigen direkt, dass $CSAT \in NPC$, in dem wir 1. zeigen, dass $CSAT \in NP$ (logisch) und 2. $\forall L \in NP : L \leq_P CSAT$.
Idee: Man versucht den Verifikationsalgorithmus A für L (in Form einer Turingmaschine) so als Schaltkreis zu konstruieren, dass der Schaltkreis 1 liefert, genau dann wenn der Verifikationsalgorithmus für einen Zeugen x und ein Wort w sagt, dass es geht. Die Überführungsfunktion der Turingmaschine von A kann dabei als konstanter Baustein B verdrahtet werden.



Wir simulieren dann die Ausführung der Turingmaschine M_A für das feste Wort w mit Eingabe des Zeugen x als Schaltkreis. Dazu werden $N = P(n)$ (polynomiell in $n = |w|$) Bausteine in einer Reihe als Block gepackt, so dass jeder Baustein einer Zelle der Turingmaschine entspricht (länger kann die Eingabe ja nicht sein). Von diesen Blöcken werden dann N untereinander gepackt, die Zeilen entsprechen dann dabei den Zeitschritten. Es kann da natürlich auch nur polynomiell viele geben, sonst wäre A ja nicht polynomiell. Man hat also ein $N \times N$-Gitter von Bausteinen, die wie folgt verdrahtet werden:



Dabei ist k_i das Kontrollbit, dass angibt ob der Baustein aktiv ist (repräsentiert quasi den Lese/Schreibkopf der Turingmaschine), logischerweise kann pro Zeitschritt nur einer aktiv sein. Genau werden die Belegungen des Folgezustandes q'_i,

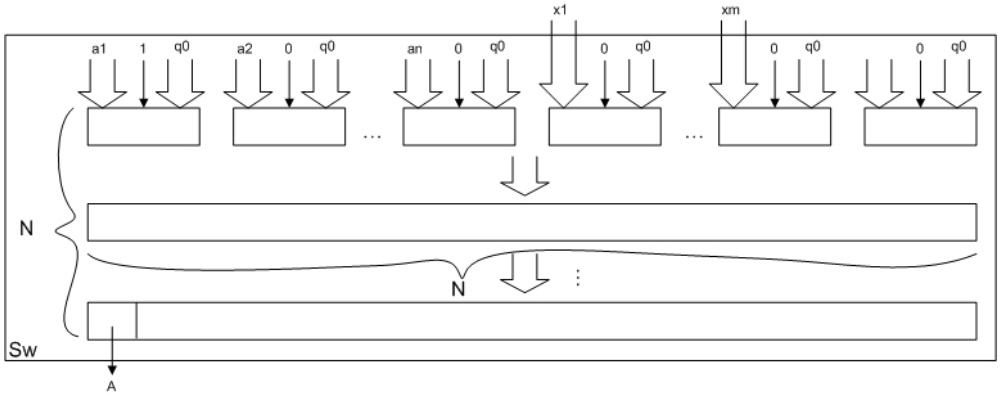
des Folgezeichens a'_i und des Folgekontrollbits k'_i wie folgt gesetzt:

$$q'_i = \begin{cases} \tilde{q} & \text{falls } k = 1 \\ 0 \dots 0 & \text{sonst} \end{cases}$$

$$a'_i = \begin{cases} \tilde{a_i} & \text{falls } k = 1 \\ a_i & \text{sonst} \end{cases}$$

$$k'_i = \begin{cases} 000 & \text{falls } k = 0 \\ 100 & \text{falls } k = 1 \wedge R = L \\ 010 & \text{falls } k = 1 \wedge R = 0 \\ 001 & \text{falls } k = 1 \wedge R = R \end{cases}$$

Insgesamt sieht das dann wie folgt aus:



Zudem nehmen wir an, dass zu Beginn Startzustand q_0 anliegt und der Kopf an der ersten Position steht und am Ende das Ergebnis ebenfalls in der ersten Zelle auszulesen ist.
Durch eben diese Konstruktion wird gewährleistet, dass $w \in L \Leftrightarrow S_w$ erfüllbar. Durch die quadratische Größe ist die Laufzeit auch in $O(N^2)$.

SAT Die Frage ist hier, ob eine boolsche Formel ϕ (in KNF) erfüllbar ist. Wir zeigen, dass $SAT \in NPC$ durch Reduktion von $CSAT \leq_P SAT$ und $SAT \in NP$.
Bei einem Schaltkreis ordnet man zu jeder Komponente (außer Eingänge, Verzweigungen) eine boolsche Variable x_i zu, die man dann verwendet, um aus dem Schaltkreis die boolsche Formel zu extrahieren (z.B für das Element x AND y $z \Leftrightarrow x \wedge y$). Zusätzlich erhält der Ausgang auch eine Variable. Alle so entstehenden Klauseln werden verundet und es entsteht eine KNF (bzw. man kann aus jeder boolschen Formel leicht eine KNF herstellen).
Eine erfüllende Belegung für S erzeugt in dem Fall dann genau eine erfüllende Belegung für ϕ und umgekehrt.

3SAT Das Frage ist hier, ob eine 3-KNF-Formel ϕ, also eine konjunktive Normalform, deren Klauseln maximal drei Literale enthält, erfüllbar ist. Wir zeigen, dass $3SAT \in NPC$ durch Reduktion von $SAT \leq_P 3SAT$ und $3SAT \in NP$.
Wir überführen die KNF-Formel α von SAT in eine 3-KNF-Formel β für $3SAT$. Dazu nehmen wir schonmal einfach alle Klauseln, die kleinergleich drei Literale enthalten auf in β. Jede andere Klausel $C = (x_1 \vee x_2 \vee \ldots \vee x_k)$ aus α wird folgendermaßen durch y_i ergänzt:

$$C' = (x_1 \vee x_2 \vee y_1) \wedge (\overline{y_1} \vee x_3 y_2) \wedge (\overline{y_2} \vee x_4 \vee y_3) \wedge \ldots \wedge (\overline{y_{k-3}} \vee x_{k-1} \vee x_k)$$

Mindestens ein x_i muss in C auf 1 gesetzt werden, in C' setzen wir dann alle y_i links von dem auf 1 gesetzten x_i auf 1 und alle rechts davon auf 0. Durch diese Belegung gilt genau, dass C erfüllt $\Leftrightarrow$ C' erfüllt (und das für alle Klauseln).

Clique Das Problem ist hier, ob es eine Knotenmenge $V' \subseteq V$ mit $|V'| = k$ eines ungerichteten Graphen $G = (V, E)$ gibt, die in sich einen vollständigen Graphen bildet. Wir reduzieren $3SAT \leq_P CLIQUE$ und zeigen zusätzlich durch $CLIQUE \in NP$, dass $CLIQUE \in NPC$.
Für jedes Literal x_i jeder Klausel r von ϕ kreiert man einen Knoten x_i^r im Graphen G für $CLIQUE$. Dann verbindet man die Knoten wie folgt mit Kanten: $(x_i^r, x_j^s) \in E$ wenn $r \neq s$ (also unterschiedliche Klauseln) und die beiden Literale sich nicht widersprechen (also $x_i^r = x$ und $x_j^s = \bar{x}$). Ist die Formel ϕ von $3SAT$ erfüllbar und hat k Literale, dann bildet sich im Graphen eine k-Clique, denn pro Klausel von ϕ wird mind. ein Literal auf 1 gesetzt, welches durch die Konstruktionsvorschriften mit allen anderen Literalen, die auf 1 gesetzt wurden in den anderen Klauseln, verbunden ist und somit einen vollständigen Teilgraph bildet.

Unabhängige Knotenmenge (UK) Die Frage ist hier, ob es eine unabhängige Knotenmenge der Größe k gibt, d.h. dass k Knoten des Graphen nicht miteinander verbunden sind ($\forall u, v \in V' : (u, v) \notin E$). Wir zeigen dies über eine Reduktion von $CLIQUE \leq_P UK$, dass $UK \in NPC$.
Wir konstruieren den komplementäre Graphen $\overline{G} = (V, \binom{V}{2} \backslash E)$ zu dem Graphen G von $CLIQUE$. Der Graph $\overline{G}$ hat genau dann eine unabhängige Knotenmenge der Größe k, wenn G eine k-Clique hat.

Vertex-Cover (ÜK) Die Frage ist hier, ob es eine Knotenüberdeckung der Größe k gibt, d.h. ob eine Teilmenge $V' \subseteq V$ von k Knoten, von der alle Kanten in E ausgehen. $VERTEX - COVER \in NPC$ wird über die Reduktion von $CLIQUE \leq_P VERTEX - COVER$ gezeigt. Zusätzlich muss natürlich noch $VERTEX - COVER \in NP$ gelten.
Ein Graph einer k-Clique kann man invertieren (d.h. die Kantenmenge ändert sich so, dass man genau die Kanten nimmt, die vorher nicht im Graph waren) und der so entstehende Graph enthält ein $n-k$-Vertex-Cover, wobei n die Anzahl der Knoten ist.

SUBSET-SUM Hier ist die Frage, ob zu einer endlichen Menge von Zahlen S und einem Wert b es ein Teilmenge $S' \subseteq S$ gibt, so dass $\sum_{i \in S'} i = b$ gilt. Um zu zeigen, dass $SUBSET - SUM \in NPC$ liegt, reduzieren wir $VERTEX - COVER \leq_P SUBSET - SUM$ und zeigen, dass $SUBSET - SUM \in NP$.
Von der Eingabe von $VERTEX - COVER$ nimmt man sich die Inzidenzmatrix des Graphen her (Kanten als Spalten in absteigender Reihenfolge, Knoten als Zeilen aufsteigend). Die so entstehende Matrix erweitert man nach unten um eine Einheitsmatrix der Größe m, wobei m die Anzahl der Kanten ist. Ganz links fügt man dann noch eine Spalte ein, die im Bereich der Inzidenzmatrix 1en enthält und im Bereich der Einheitsmatrix 0en. Ganz ans Ende fügt man dann noch eine Zeile ein, die in der ersten Zelle das k (Größe des Vertex-Covers) enthält und in allen übrigen Zellen 2en. Jede Zeile interpretiert man dann als Zahl zur Basis 4 (Größte 4er-Potenz ganz links). Die Eingabe für $SUBSET - SUM$ sind dann die als Zahlenmenge S die Zahlen der Zeilen (bis auf letzte Zeile), das b bildet dann die verbleibende Zeile. Zu b summieren sich dann genau die Zeilen auf, die den Zeilen der im Vertex-Cover V' enthaltenen Knoten entsprechen und die Zeilen des Einheitsmatrixabschnitts, der den Kanten entspricht, die nur einen Endpunkt in V' haben.

PARTITION Das Problem $PARTITION$ fragt danach, ob man eine endlichen Menge von Zahlen S so in zwei Hälften aufteilen kann, dass sich beide Hälften zum selben Wert addieren. Um $PARTITION \in NPC$ zu zeigen, wird $SUBSET - SUM \leq_P PARTITION$ geprüft.
Wir konstruieren die Eingabe für $PARTITION$ wie folgt: Man nimmt zur Eingabe S von $SUBSET - SUM$ die Werte $x = 2 * A - b$ und $y = A + b$ hinzu, wobei $A = \sum_{a \in S} s$ gilt. Die beiden Mengen $S' \cup \{x\}$ und $S \backslash S' \cup \{y\}$ sind dann in der Summe gleich groß, wobei S' die richtige Teilmenge von $SUBSET - SUM$ ist.

Gerichteter Hamilton-Kreis Ein Hamilton-Kreis eines gerichteten Graphen ist ein Kreis, der jeden Knoten enthält. $HC \in NPC$ zeigt man über eine Reduktion von $VERTEX - COVER \leq_P HC$.

Traveling Salesperson Problem (TSP) Die Frage ist hier, ob es eine Rundreise in einem vollständigen Graphen minimalen Gewichts gibt (bzw. mit Gewicht $\leq k$). Wir reduzieren $HC \leq_P TSP$ um zu zeigen, dass $TSP \in NPC$.
Zu dem Graphen vom Hamilton-Kreis-Problem nehmen wir den vollständigen Graphen $G' = (V, E')$ und setzten die Kosten auf 0 für jede Kante in E und 1 andernfalls. Es existiert genau dann eine Rundreise mit Kosten 0, wenn es vorher einen Hamilton-Kreis gab.

1.3 Andere Problemklassen

- Ein Problem ist in Klasse *Co-NP*, wenn das zu L komplementäre Problem $\overline{L} \in NP$.

- $PSPACE$ ist die Problemklasse, in der alle Probleme auf einer deterministischen oder nichtdeterministischen Turingmaschine in polynomiellen Platz gelöst werden können.
  Ein Beispiel für ein Problem in $PSPACE$ ist die Erfüllbarkeit von quantifizierten boolschen Formeln.

- $EXPTIME$ ist die Klasse von Problemen, die auf einer deterministischen Turingmaschine in exponentieller Zeit entschieden werden können. Ein Beispiel für ein Problem ist die Äquivalenz von regulären Ausdrücken.
  Da $P \subset EXPTIME$ (echte Teilmenge!) ist, muss zwischen $P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$ irgendwo eine echte Teilmenge liegen. Man weiß bloß noch nicht wo.

1.4 Approximationsalgorithmen

- Für TSP
- Für Vertex-Cover
- Für Subset-Sum

2 Graphen und Flüsse

2.1 Elementare Graphalgorithmen

Breitensuche Bei der Breitensuche wird der Graph ausgehend von einem Startknoten der Breite nach durchsucht, d.h. man entdeckt zunächst erst alle direkt benachbarten Knoten, dann alle von den Nachbarn benachbarten Knote usw.. Daraus folgt, dass man mit der Breitensuche den kürzesten Weg von dem Startknoten zu allen anderen Knoten findet.
Laufzeit: Jeder Knoten wird einmal angeschaut (in die Warteschlange aufgenommen und entnommen) und beim Anschauen wird die Adjazentliste des Knotens durchsucht. D.h. man hat eine Laufzeit von insgesamt $O(|V| + |E|)$.

Tiefensuche Bei der Tiefensuche wird der Graph ausgehend von einem Startknoten in die Tiefe durchsucht, d.h. man besucht zunächst von dem gerade entdeckten Knoten rekursiv alle noch nicht entdeckten Knoten.
Laufzeit: Wie bei der Breitensuche wird jeder Knoten und jede Kante einmal angefasst, d.h. $O(|V| + |E|)$.

Topologische Sortierung Bei der topologischen Sortierung sollen die Knoten eines DAG (directed acyclic graph) so horizontal angeordnet werden, dass alle Kanten nur von links nach rechts zeigen, also bei einer Kante (u, v) u vor v in der Sortierung erscheint.
Algorithmus: Man kann über die Tiefensuche und die dabei entstehenden Timestamps die Sortierung vornehmen, in dem immer die fertigen Knoten vorne an eine Liste angehangen werden.

Laufzeit: Durch die Benutzung von Tiefensuche ist die Laufzeit natürlich $O(|V|+|E|)$.

Starke Zusammenhangskomponente Eine starke Zusammenhangskomponente eines gerichteten Graphen ist eine maximale Teilmenge der Knoten, in der man paarweise für alle Knoten u, v von u nach v kommt und von v nach u.
Algorithmus: Man findet starke Zusammenhangskomponenten in dem man DFS auf G durchführt, und dann in absteigender Reihenfolge der Finish-Timestamps DFS auf dem Graphen G' durchführt, wobei G' der transponierte Graph ist, in dem alle Kanten umgedreht sind. Die Zusammenhangskomponenten sind dann die Bäume, die bei dem zweiten DFS entstehen.
Laufzeit: Pro DFS $O(|V| + |E|)$ und auch $O(|V| + |E|)$ für den transponierten Graphen, insgesamt also auch $O(|V| + |E|)$.

2.2 Minimal Spanning Trees (MST)

Ein MST ist ein Baum T, der alle Knoten eines gewichteten Graphen $G = (V, E)$ enthält und minimale Gesamt-Kantenkosten $w(T) = \sum_{(u,v)\in T} w(u, v)$ hat.

Generischer Ansatz Greedy-Strategie, bei der in jedem Schritt der MST um eine Kante vergrößert wird. Dabei wird stets die Invariante gehalten, dass die Kantenmenge A eine Untermenge eines gültigen MSTs ist. Als *sichere Kante* bezeichnet man dann eine Kante (u, v), deren Hinzunahme zu A die Invariante noch erfüllen lässt. Eine *leichte Kante* ist dabei eine Kante, die genau einen Endpunkt in der von A überspannten Knotenmenge hat und den anderen Endpunkt außerhalb und dabei minimales Gewicht unter allen solchen Kanten hat.
Algorithmus: Starte mit $A = \emptyset$. Nimm dann jeweils eine sichere Kante hinzu, solange bis es keine sicheren Kanten mehr gibt.

Kruskal Bei dem Algorithmus wird jeweils die sichere Kante minimalen Gewichts hinzugenommen, die zwei getrennte Komponenten des MST-Graphen verbindet. Dazu verwendet man eine Union-Find-Struktur.

```
KRUSKAL(G,w) {
  A = {}
  foreach v in V
    put v in own set
  sort edges of G by weight
  for e= (u,v) in E {
    if FIND(u) != FIND(v) {
      A = A + e
      UNION(u,v)
    }
  }
  return A
}
```

Laufzeit: Kanten sortieren kostet $O(|E| \log |E|)$. Die Schleife wird $|E|$ mal aufgerufen, mit der Initialisierung der Mengen ergibt sich dann $O((|V|+|E|) \log^* |V|)$. Insgesamt haben wir dann eine Laufzeit von $O(|E| \log |E|)$ (mit $|E| < |V|^2$ gilt auch $O(|E| \log |V|)$.

Prim Hier wird ausgehend von einem Startknoten der Baum aufgebaut, in dem immer eine sichere Kante minimalen Gewichts dazu genommen wird, die einen noch unerreichten Knoten verbindet. Alle Kanten werden dabei in einer Prioritätswarteschlange gehalten.

```
PRIM(G,w,r) {
  foreach u in V {
    u.weight = infinity
    u.predecessor = null
  }
  r.weight = 0
  Q = V
  while Q != {} {
    u = extractMin(Q)
    foreach v adjacent to u {
      if v in Q and w(u,v) < v.weight {
        v.predecessor = u
        v.weight = w(u,v)
      }
    }
  }
}
```

Laufzeit: Die Heap-Operationen benötigen $O(|V| \log |V|)$, das Aktualisieren der Gewichte (ändert ja die Struktur des Heaps) benötigt $O(|E| \log |V|)$, was auch die insgesamt Laufzeit des Prim-Algorithmus ist.

2.3 Kürzeste Wege

Man will zu einem gegebenen gerichteten Graphen $G = (V, E)$ mit der Gewichtsfunktion $w : E \to \mathbb{R}$ Wege zwischen Knoten finden, so dass die Summe der Kantengewichte minimal ist.

2.3.1 Single Source Shortest Paths

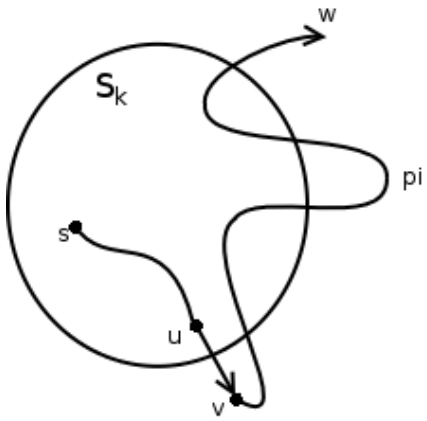
Findet den kürzesten Weg von einem Startknoten s zu allen anderen Knoten.
Wir bekommen Probleme, wenn es Kreise negativen Gewichts gibt, da man dann immer einen noch kürzeren Weg über den Kreis finden kann. Daher dürfen keine negative Kreise enthalten sein im Graphen.

Algorithmus von Dijkstra Der Algorithmus funktioniert nur für Gewichte ≥ 0. Es ist ein *Greedy-Algorithmus*, und die Idee ist, dass man eine Menge S von Knoten verwaltet, zu denen schon die kürzesten Wege gefunden wurde, und diese sukzessive um neue Knoten erweitert, die noch nicht erreicht wurden. Dabei nimmt man immer den Knoten in einem Schritt dazu, dessen aktuelle Schätzung des kürzesten Weges die kleinste ist und dann die Schätzungen seiner Nachbarn aktualisiert.
Die Schätzung des aktuell kürzesten Weges von s nach v wird in dem Feld $D[v]$ verwaltet, in $Pred[v]$ der Vorgänger im kürzesten Pfad. Die verbleibenden Knoten, die nicht in S sind, werden in einer Prioritätswarteschlange Q gehalten, die nach $D[v]$ sortiert ist.

```
DIJKSTRA(G,w,s) {
  S = {}
  foreach v in V {
    D[v] = infinity
    Pred[v] = null
  }
  D[s] = 0
  Q = V

  while Q != {} {
    u = extractMin(Q)
    S = S + u
    foreach v adjacent to u {
      if D[v] > D[u] + w(u,v) {
        D[v] = D[u] + w(u,v)
        Pred[v] = u
      }
    }
  }
}
```

Korrektheit: Zu zeigen ist, dass am Ende für alle $v \in V : D[v] = \delta(s, v)$ enthält, wobei $\delta(s, v)$ die Länge des kürzesten Weges von s nach v ist. Dazu zeigen wir die Invariante, dass $\forall v \in S : D[v] = \delta(s, v)$ zu Beginn jedes Schleifendurchlaufs gilt.
Zur Initialisierung wird die Invariante trivialerweise erfüllt, denn da ist $S = \emptyset$. In der $k+1$-ten Iteration werde Knoten w ausgewählt, für den gelten soll, dass $D[w] > \delta(s, w)$. Sei S_k die Menge S nach der k-ten Iteration. Wir betrachten die Kante (u, v) auf einem Pfad $s \to w$, für die gelten soll, dass $u \in S_k$ und $v \notin S_k$. Nach Voraussetzung gilt, dass $D[u] = \delta(s, u)$ und $\delta(s, v) = \delta(s, u) + w(u, v) = D[u] + w(u, v) \leq D[v]$ auf Grund des Update-Prozesses, dass bei der Hinzunahme von $u \in S$ ausgelöst wurde. Es muss $\delta(s, v) = D[v]$ gelten. Da der Knoten v in dem Weg vor w ist, muss $D[w] > \delta(s, w) \geq \delta(s, v) = D[v]$ gelten. Dies ist aber ein Widerspruch, denn wenn $D[w] > D[v]$ wäre in dem Schritt v anstelle von w ausgewählt worden. Demnach muss $D[w] = \delta(s, w)$ sein und die Invariante wäre bewiesen.



Laufzeit: Die `while`-Schleife läuft $|V|$ mal durch, wobei die Extraktion des Minimums aus der Prioritätswarteschlange (als Heap implementiert) $\log|V|$ benötigt. Die `foreach`-Schleife geht über alle Kanten genau einmal rüber, also $|E|$ mal. Durch die Aktualisierung des $D[v]$ wird der Heap neu sortiert, was auch $\log|V|$ Zeit benötigt. Wir erhalten insgesamt $O(|V|\log|V| + |E|\log|V|) = O((|V| + |E|)\log|V|)$.

2.3.2 All-Pairs-Shortest Paths

Findet den kürzesten Weg zwischen jedem Paar von Knoten des Graphen.

Floyd-Warshall-Algorithmus Der Floyd-Warshall-Algorithmus funktioniert auch für Gewichte < 0, aber es dürfen auch hier keine negativen Zyklen enthalten sein.
Der Algorithmus basiert auf *dynamischer Programmierung* und die Idee ist, dass ein kürzester Weg von Knoten i nach j, der nur Zwischenknoten von $1,\ldots,k$ benutzt (wobei o.B.d.A. angenommen werden kann, dass die Knoten durchnummeriert sind von $1,\ldots,n$) sich aufteilen lässt in den Pfad, der k gar nicht benutzt (sondern nur Knoten bis $k-1$) oder, falls k benutzt wird man den Weg unterteilen kann in die Strecke von $i \to k$ und von $k \to j$, wobei dort jeweils auf den Teilstrecken nur die Zwischenknoten bis $k-1$ benutzt werden.
Die Eingabe ist eine $|V| \times |V|$-Matrix W, wobei in der Zelle $(w)_{ij} = w(i,j)$ steht. Die Ausgabe soll entsprechend eine Matrix D sein, die in der Zelle $(d)_{ij}$ die Länge des Weges von i nach j enthält.

```
FLOYD-WARSHALL(W) {
  n = |V|
  d_ij = w_ij

  for k = 1 to n {
    for i = 1 to n {
      for j = 1 to n {
        d_ij^k = min (d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1) )
```

```
      }
    }
  }
}
```

Korrektheit: Die Korrektheit ist über die oben genannten Eigenschaften begründet.
Laufzeit: Die Laufzeit wird durch die verschachtelten `for`-Schleifen definiert. In der innersten Schleife hat man jeweils pro Iteration konstanten Aufwand, daher ergibt sich insgesamt $\Theta(|V|^3)$.

Man kann mit Floyd-Warshall auch noch andere Sachen berechnen, in dem man min und + austauscht. Wenn W als Bitmatrix gegeben ist, kann man z.B. durch Austauschen von min durch $\vee$ und + durch $\wedge$ prüfen, ob es einen Weg von i nach j gibt.

Für lichte Graphen, wo $|E|$ viel kleiner als $|V|^2$ ist, ist es effizienter $|V|$ mal Dijkstra auszuführen.

2.4 Maximaler Fluss

- Ein *Flussnetzwerk* ist ein gerichteter Graph $G = (V, E)$, in der jede Kante (u, v) eine nicht-negative Kapazität $c(u, v)$ besitzt. Der Graph ist zusammenhängend und jeder Knoten $v \in V$ liegt auf einem Weg von der Quelle s zur Senke t.
- Der *Fluss* ist eine Funktion $f : V \times V \to \mathbb{R}$, die folgende zwei Bedingungen erfüllen muss:
  1. Kapazitätsbedingung: $0 \leq f(u, v) \leq c(u, v)$ für alle $u, v \in V$.
  2. Flusskonservation: $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$ für alle $u \in V \backslash \{s, t\}$.

  Der Wert des Flusses ist $|f| = \sum_{v \in V} f(s, v)$.
- Wenn man mehrere Quellen / Senken hat, kann man durch Einführen einer Superquelle / Supersenke, die jeweils mit jeder Quelle / Senke verbunden sind das Problem wieder wie ein normales Flussnetz betrachten.
- Das *Restnetz* G_f zu eine Graphen G mit Fluss f gibt an, um wieviel Restkapazität die Kanten aus G noch mit dem aktuellen Fluss haben und in wie weit der Fluss erniedrigt werden kann.

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{für } (u, v) \in E \\ f(v, u) & \text{für } (v, u) \in E \\ 0 & \text{sonst} \end{cases}$$

- Ein *augmentierender Pfad* p ist ein einfacher Weg im Restnetz G_f von s nach t. Nach Definition des Restnetzes kann der Fluss an einer Kante von p um maximal $c_f(u, v)$ erhöht werden. Die *Restkapazität* auf dem augmentierenden Pfad ist $c_f(p) = \min\{c_f(u, v) \mid (u, v) \text{ liegt auf } p\}$.

- Ein Schnitt (S, T) eines Netzwerks ist eine Partition von V in S und $T = V \backslash S$, so dass $s \in S$ und $t \in T$. Der *Net-Flow eines Schnittes* ist $f(S,T) = \sum_{s \in S} \sum_{t \in T} f(s,t) - \sum_{s \in S} \sum_{t \in T} f(t,s)$. Die *Kapazität des Schnitts* ist $c(S,T) = \sum_{s \in S} \sum_{t \in T} c(s,t)$.
- Der *minimale Schnitt* eines Netzwerks ist der Schnitt mit minimaler Kapazität.

2.4.1 Ford-Fulkerson-Methode

Der Algorithmus funktioniert nur für rationalle oder ganze Zahlen, denn mit irrationalen Zahlen terminiert er eventuell nicht.
Die Idee ist hierbei, dass man den Fluss, der anfänglich auf null gesetzt wird, Schritt für Schritt erhöht. Und zwar wird der Fluss immer entlang eines augmentierenden Pfades erhöht, der mit Hilfe des zugehörigen Restnetzes gefunden wird. Wenn es keinen augmentierenden Pfad mehr gibt, dann ist der maximale Fluss erreicht.
Laufzeit: In jedem Schritt wird der Fluss erhöht, also hat man maximal $|f^*|$ Iterationen, wobei f^* den maximalen Fluss angibt. Pro Iteration wird dann ein augmentierender Pfad gesucht, was mit Tiefensuche geht, also $O(|E||f^*|)$.

2.4.2 Edmonds-Karp-Algorithmus

Da der Fluss f^* exponentiell sein kann, ist Ford-Fulkerson nicht besonders toll. Durch Verwendung der Breitensuche statt der Tiefensuche um die augmentierenden Wege zu finden, geht das besser. Dabei nimmt man dann immer den kürzesten Weg von s nach t im Restnetz, wobei jede Kante das gleiche Gewicht hat. Der kürzeste Weg von u nach v im Restnetz wird dabei mit $\delta_f(u,v)$ bezeichnet.
Lemma 1: Es gilt, dass während der Ausführung von Edmonds-Karp $\delta_f(s,v)$ für alle $v \in V \backslash \{s,t\}$ monoton wächst.
Beweis: Angenommen für einen Knoten v gilt dies nicht, also $\delta_f(s,v) > \delta_{f'}(s,v)$. v sei der einzige Knoten für den dies gilt. Auf dem Weg von s nach v in $G_{f'}$ liegt der Knoten u mit direkter Kante (u,v). Für die muss folglich gelten $\delta_f(s,u) \leq \delta_{f'}(s,u)$ und $\delta_{f'}(s,v) = \delta_{f'}(s,u) + 1$. Wir behaupten, dass auch $(u,v) \in G_f$. Es gilt dann

$$\begin{aligned}\delta_f(s,v) &= \delta_f(s,u) + 1 \\ \delta_f(s,v) &\leq \delta_{f'}(s,u) + 1 \\ \delta_f(s,v) &= \delta_{f'}(s,v)\end{aligned}$$

Das ist ein Widerspruch zu der Annahme. Also kann (u,v) nicht in G_f sein, muss aber in $G_{f'}$ nach Annahme. Wie kann also das passieren, dass (u,v) in $G_{f'}$ und nicht in G_f? Es muss in G_f eine Kante (v,u) gegeben haben.

$$\begin{aligned}\delta_f(s,v) &= \delta_f(s,u) - 1 \\ \delta_f(s,v) &\leq \delta_{f'}(s,u) - 1 \\ \delta_f(s,v) &= \delta_{f'}(s,v) - 2\end{aligned}$$

Das ist auch ein Widerspruch, also kann es einen solchen Knoten v nicht geben.

Lemma 2: Es gilt auch, dass der Algorithmus insgesamt $O(|V||E|)$ Augmentierungen durchführt.
Beweis: Eine *kritische Kante* ist eine Kante auf dem augmentierenden Pfad p, für deren Kapazität $c_f(u,v) = c_f(p)$ gilt, d.h. die Kante verschwindet nach der Flusserhöhung aus dem Netzwerk. Sei f der Fluss, bei dem die Kante (u,v) zum ersten Mal verschwindet. Es gilt hier $\delta_f(s,v) = \delta_f(s,u) + 1$. Die Kante kann in einem späteren Fluss erst dann wieder auftauchen, wenn der augmentierende Pfad hier von v nach u ging bei f', also $\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1$. Mit obiger Aussage gilt

$$\begin{aligned}\delta_{f'}(s,u) &= \delta_{f'}(s,v) + 1 \\ &\geq \delta_f(s,v) + 1 \\ &= \delta_f(s,u) + 2\end{aligned}$$

Zwischen zwei malen, wo die Kante kritisch wird, verlängert sich also der Weg zu u um mind. zwei Schritte. Der Abstand von s zu u kann maximal $|V| - 2$ sein und jede Kante kann höchstens $\frac{|V|-2}{2}$ mal kritisch sein. Da bei jeder Augmentation mind. eine Kante kritisch wird, gibt es maximal $O(|V||E|)$ davon.
Laufzeit: Man führt $O(|V||E|)$ Augmentierungen durch. Pro Augmentierung muss man die Breitensuche durchführen, was $O(|E|)$ kostet, insgesamt also $O(|V||E|^2)$.

2.4.3 Bipartites Matching

- Ein *Matching* eines ungerichteten Graphen $G = (V,E)$ ist eine Teilmenge der Kanten $M \subseteq E$ für die gilt, dass jeder Knoten $v \in V$ zu maximal einer Kante aus M inzident ist.
- Ein *maximales Matching* (engl. maxima matching) ist ein Matching, bei dem keine Obermenge zu M gefunden werden kann, das auch ein Matching ist.
- Ein *größtes Matching* (engl. maximum matching) ist dagegen ein Matching, bei dem $|M|$ maximal ist. Jedes größte Matching ist auch ein maximales Matching.
- Man kann ein Matching-Problem eines bipartiten Graphen $G = (V,E)$ in ein Flussproblem auf dem Netzwerk $G' = (V',E')$ lösen, in dem man $V' = V \cup \{s,t\}$ definiert, wobei s die Quelle ist, die eine Kante zu allen Knoten v der linken Gruppe des bipartiten Graphen bekommt und t ist die Senke, die durch eine Kante mit allen Knoten des rechten Hälfte des bipartiten Graphen verbunden wird. Zusätzlich werden alle Kante in E' von links nach rechts gerichtet und erhalten Kapazität 1.

Lemma: M ist ein Matching in G, genau dann wenn f in G' ein ganzzahliger Fluss mit Wert $|f| = |M|$ ist.
Beweis:

$\Rightarrow$ Sei M das Matching, dann schicken wir durch jede Matchingkante in G' den Fluss der Größe 1, alle anderen Kanten bekommen den Nullfluss. Aufgrund der Eigenschaft von einem Matching sind die Wege von s nach t jeweils disjunkt und so ergibt dann die Anzahl der Matchingkanten die Größe des Flusses.

$\Leftarrow$ Sei f ein ganzzahliger Fluss. Dann nehmen wir als Matchingkanten alle Kanten (u, v), die einen Fluss > 0 haben. Jeder Knoten der linken Hälfte hat nur eine eingehende Kante (die von s aus), die Kapazität 1 hat. Falls in ein solchen Knoten ein Fluss von 1 fließt, so muss er auch wieder rausfließen, da wir ganzzahligen Fluss haben, kann auch nur maximal eine ausgehende Kante bedient werden. Der Fluss muss also auf einer Kante (u, v) 1 sein und auf allen anderen 0, wobei u in der linken Hälfte und v in der rechten.

Ein größtest Matching kann mit Hilfe des maximalen Flusses in $O(|E||V|)$ gefunden werden. Wir führen einfach Ford-Fulkerson durch, der eine Laufzeit von $O(|f * ||E|)$ hat. Der Fluss entspricht ja hier der Größte des Matchings, das maximal $\frac{|V|}{2}$ groß sein kann, also $O(|E||V|)$.

3 Datenstrukturen

3.1 Union-Find

- In der *Union-Find-Datenstruktur* verwaltet man eine Kollektion dynamische disjunkter Menge $S = \{S_1, \ldots, S_k\}$. Jede Menge wird dabei durch einen Repräsentanten i vertreten, der auch Element der Menge ist.
- Wir definieren folgende drei Operationen: `MAKESET(x)` (erstellt eine Menge aus dem Element x), `FIND(x)` (Findet die Menge (= den Repräsentanten), zu der x gehört) und `UNION(x,y)` (vereinigt die zwei verschiedenen Mengen, in denen x und y Elemente sind, zu einer Gesamtmenge).
- Wir repräsentieren eine Menge als einen Baum, wobei hier alle Kanten vom Kinderknoten zum Vaterknoten hin orientiert sind. Die Wurzel eines solchen Baums ist der Repräsentant der Menge.
- Die `UNION`-Operation hat immer Konstante Laufzeit.

Union-Find mit Höhenausgleich Beim Höhenausgleich wird bei der `UNION`-Operation der Baum mit kleinerer Höhe an den größeren Baum rangehangen, in dem der Repräsentant einen Zeiger auf die neue Wurzel bekommt. Dadurch ist garantiert, dass der Baum stets eine Höhe in $\log n$ hat, wobei n die Gesamtzahl der Elemente insgesamt ist.
Beweis: Wir führen eine Induktion über die k, die Anzahl der Elemente von T, wobei T der Baum sei, der bei eine `UNION`-Operation der Bäume T_1 und T_2 entsteht, wobei h, h_1 und h_2 die entsprechenden Höhen sind und k_1 und k_2 die entsprechenden Anzahlen von Elementen. Es gelte zudem $h_1 \geq h_2$. Zu zeigen ist, dass $2^h = k$.

Induktionsanfang $k = 1$, d.h. $2^0 = 1$, was eine wahre Aussage ist, denn wir haben hier Höhe 0.

Induktionsschritt $k = k_1 + k_2$. Durch die Höhenbalancierung gilt $h = \max(h_1, h_2 + 1)$. Es gibt hier zwei Fälle:

1. $h = h_1$. Dann ist $k = k_1 + k_2 \geq k_1 = 2^{h_1} = 2^h$.
2. $h_1 = h_2$. Dann ist $k = k_1 + k_2 = 2^{h_1} + 2^{h_2} = 2^{h_2+1} = 2^h$

Union-Find mit Pfadkompression Bei der Pfadkompression werden bei jeder `FIND`-Operation alle Knoten auf dem Pfad vom Ausgangselement bis hin zur Wurzel direkt an die Wurzel rangehangen. Das ergibt für spätere `FIND`-Operationen folglich eine bessere Laufzeit. Die amortisierte Laufzeit von `FIND` ergibt sich als $\log^* n$, wobei n die Anzahl der Elemente ist.
$\log^* n$ wächst langsamer als $\log \log n$ und gibt die Anzahl der log-Operationen an, die man anwenden muss, bis man ein Ergebnis ≤ 1 erreicht.
Wir definieren

$$\begin{aligned} F(0) &= 1 \\ F(i) &= 2^{F(i-1)} \\ G(n) &= \log^* n = \min\{k \mid F(k) \geq n\} \end{aligned}$$

Mit $Rang(v)$ wird die Höhe des Baumes T_v bezeichnet, wobei v die Wurzel ist. Es gilt für alle Knoten v:

1. Baum T_v hat mindestens $2^{Rang(v)}$ Knoten.
2. Es gibt höchstens $\frac{n}{2^r}$ Knoten mit Rang $r \in \mathbb{N}$.
3. Alle Ränge sind $\leq \log n$.
4. Falls bei der Ausführung von σ (Folge von `UNION`- und `FIND`-Operationen) w Nachkomme von v ist, dann ist der $Rang(w) < Rang(v)$.

Laufzeit: Wir betrachten eine Operationsfolge σ von $n - 1$ `UNION`- und m `FIND`-Operationen.
Wir teilen die n Knoten entsprechend ihrer Ränge in Gruppen auf, und zwar kommen die Ränge r in die Gruppe mit gleichem $G(r) = \log^*(r)$. Jede Gruppe enthält also die Elemente mit Rängen im Bereich von $]F(i-1), 2^{F(i-1)} = F(i)]$. Insgesamt kann es also nicht mehr als $\log^* n$ Gruppen geben.
Jetzt machen wir eine Buchhalteranalyse und weisen die Kosten für `FIND` einmal einem allgemeinen Konto zu und den separaten Konten der Knoten. Dabei geben alle Knoten, die direkte Kinder der Wurzel sind, ihre Kosten an das allgemeine Konto zurück und auch alle Knoten, deren Väter in einer anderen Gruppe sind als sie selbst. Alle anderen Knoten führen an die separaten Konten ab.
Das allgemeine Konto kann dabei mit maximal $\log^* n$ belastet werden, da es maximal so viele Gruppen gibt.

Wie oft kann nun einem Knoten x in Gruppe i mit $Rang(x) \in \{F(i-1)+1, \ldots, F(i)\}$ kosten zugewiesen werden? Dies geht nur so lange, wie x und sein Vater in der selben Gruppe sind, dies sind sie aber nach spätestens $F(i)-F(i-1)$ Operationen nicht mehr (also maximal $F(i) - F(i-1)$ mal können Kosten zugewiesen werden). Sei $N(i)$ die Anzahl der Knoten in Gruppe i.

$$\begin{aligned} N(i) &= \sum_{r=F(i-1)+1}^{F(i)} \frac{n}{2^r} \\ &= \frac{n}{2^{F(i-1)+1}} \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) \\ &\leq \frac{n}{F(i)} \end{aligned}$$

Insgesamt kann einer Gruppe dann $(F(i)-F(i-1))*N(i)$ abgeschätzt durch $F(i)N(i) \leq n$ Kosten zugewiesen werden. Da man $\log^* n$ Gruppen hat, ergibt sich für alle Gruppen $n \log^* n$.
Insgesamt ergibt sich dann die Laufzeit von $O(m \log^* n + n \log^* n)$.

Es gibt noch eine bessere (genauere) Abschätzung über die Ackermannfunktion (bzw. der Umkehrung davon). Die Laufzeit ist dann $O(m\alpha(m, n))$.

3.2 Optimale Binäre Suchbäume

Man hat zu einem festen Satz an Schlüsseln k_i mit $i = \{1, \ldots, n\}$, die zudem einer Ordnungsrelation unterliegen, Zugriffswahrscheinlichkeiten p_i. Zu allen Schlüsseln, die nicht vorkommen, gibt es „dummy Schlüssel“ d_i mit $i = \{0, \ldots, n\}$, die auch entsprechende Zugriffswahrscheinlichkeiten q_i haben. Alle Wahrscheinlichkeiten addieren sich zu 1. Dabei steht d_0 für die Zugriffe auf Elemente $< k_1$ und d_n für die Zugriffe $> k_n$. Der Dummy d_i repräsentiert das Intervall zwischen k_i und k_{i+1}. Daraus wollen wir nun einen binären Suchbaum basteln, der die Schlüssel in den Knoten hält und die Dummys in den Blättern.
Das Ziel ist es, einen optimalen binären Suchbaum zu erstellen, in dem die Anzahl der zu betrachtenden Knoten für die Suchen minimiert wird. Dazu betrachten wird die erwartete Kosten einer Suche in einem solchen Baum, die wir minimieren wollen:

$$E(Kosten\ der\ Suche) = P_T = \sum_{i=1}^{n} p_i * (Tiefe(i) + 1) + \sum_{j=0}^{n} q_j * Tiefe(j)$$

Wir machen dynamische Programmierung und zeigen, dass man die erwartete Anzahl von Vergleichen von $T_{i,j}$ (das ist der Teilbaum, der Elemente $[k_i, k_j]$ enthält) durch die erwartete Anzahl der Vergleiche im linken und rechten Teilbaum ausdrücken lässt.
Wir definieren $w_{i,j} = \sum_{k=i}^{j} p_k + \sum_{l=i-1}^{j} q_l$, $\tilde{p_k} = \frac{p_k}{w_{i,j}}$ und $\tilde{q_k} = \frac{q_k}{w_{i,j}}$. Der linke Teilbaum

von $T_{i,j}$ sei $T_{i,m-1}$, der rechte $T_{m+1,j}$ mit der Wurzel m.

$$\begin{aligned}
P_{T_{i,j}} &= \sum_{k=i}^{j} \tilde{p_k} * (Tiefe(k)+1) + \sum_{k=i-1}^{j} \tilde{q_k} * Tiefe(k) \\
&= \sum_{k=i}^{m-1} \tilde{p_k} * (Tiefe(k)+1) + \tilde{p_m} + \sum_{k=m+1}^{j} \tilde{p_k}(Tiefe(k)+1) \\
&+ \sum_{k=i-1}^{m-1} \tilde{q_k} * Tiefe(k) + \sum_{k=m+1}^{j} \tilde{q_k} * Tiefe(k) \\
&\vdots \\
&= 1 + \frac{w_{i,m-1} * P_{T_l} + w_{m+1,j} * P_{T_r}}{w_{i,j}}
\end{aligned}$$

Algorithmus:

```
OPT-BIN() {
  for i = 0 to n {
    w_{i+1,i} = q_i  // Wahrscheinlichkeit, dass gesuchtes k in [k_i, k_j]
    c_{i+1,i] = 0    // Kosten von T_{i,j} = w_{i,j} * P_{T_{i,j}}
  }
  for k = 0 to n-1 {
    for i = 1 to n - k {
      j = i+k
      Bestimme m mit i <= m <= j, so dass c_{i,m-1} + c_{m+1,j} minimal
      r_{i,j} = m      // Index der Wurzel für T_{i,j}
      w_{i,j} = w{i,m-1} + w_{m+1,j} + p_m
      c_{i,j} = c_{i,m-1} + c_{m+1,j} + w_{i,j}
    }
  }
}
```

Speicherplatz: Wir machen das ganze in einer Matrix der Grüße $n \times n$, also ist der Speicherplatzbedarf $O(n^2)$.
Laufzeit: Die Laufzeit ist in $O(n^3)$, der quadratische Anteil folgt aus dem Ansatz der dynamischen Programmierung, der Rest kommt aus dem Finden der Wurzel m (Abhängig von aktuellem k).
Hier ist auch noch Optimierungsmöglichkeit. Man kann zeigen, dass man die Wurzel auf das Intervall $r_{i,j-1} \leq r_{i,j} \leq r_{i+1,j}$ beschränken kann. Damit verbessert sich die Laufzeit auf $O(n^2)$.

3.3 Wörterbuch

Ein *Wörterbuch* ist ein abstrakter Datentyp, der eine Menge $S \in U$, wobei U meist ein linear geordnetes Universum ist, verwaltet und zwar mit den folgenden Operationen:

`SUCH(a,S)` mit $a \in U$, `EINF(a,s)` und `STREICHE(a,S)`.

3.3.1 Sortiertes Array

Laufzeit: Alle Einfügen und Löschen gehen in $O(n)$, Suchen in $O(\log n)$, z.B. mit Binärsuche.

3.3.2 Hashing

Eine Hashfunkiton $h : U \to \mathbb{N}$ berechnet für jedes Element den Index, an welche Stelle das Objekt gespeichert werden soll (man braucht also keine Ordnung auf U!). Da der Speicherplatz begrenzt ist, ist h nicht injektiv. Sollten also mehrere Objekte an eine Stelle kommen, verwaltet man pro Stelle eine verkettete Liste.
Laufzeit: Bei einer guten Hashfunktion alle Operationen in $O(1)$.

3.3.3 Binärer Suchbaum

In einem binären Suchbaum stehen die Elemente in den inneren Knoten und die Blätter stehen für erfolglose Suchen. Dabei befinden sich für jeden inneren Knoten mit Wurzel w im linken Baum nur Elemente $\leq w$ und im rechten Teilbaum $> w$.
Laufzeit: Alle Operationen gehen in $O(h)$, wobei h die Höhe des Baumes ist. Die Höhe kann im besten Fall logarithmisch sein und im schlechtesten Fall linear.

3.3.4 AVL-Baum

Ein AVL-Baum ist ein balancierter Binärbaum. Die Balancierung wird durch Eigenschaft sichergestellt, dass für jeden Knoten v der Höhenunterschied des linken und rechten Teilbaums maximal 1 betragen darf. Durch Rotation und Doppelrotation wird beim Einfügen bzw. Streichen die AVL-Eigenschaft beibehalten.
Der schlechteste Fall ist hier, dass linke Teilbaum jedes inneren Knotens um 1 größer ist als der rechte. Dann gilt für die minimale Anzahl der Knoten mit Höhe h $n_h = n_{h-1} + n_{h-2} + 1$. Die lässt sich mit der Fibonaccizahl als $n_h > Fib(h-1)$. Die Fibonaccizahl wächst exponentiell, damit ist die Höhe also logarithmisch.
Laufzeit: Für alle Operationen ist die Laufzeit $O(\log n)$.

3.3.5 BB[α]-Baum

Ein BB[α]-Baum ist auch ein binärer Suchbaum. Die Balance wird hier ausgedrückt durch das Verhältnis von der Anzahl der Blätter im linkem Teilbaum T_l zu der Blätteranzahl im Gesamtbaum T jedes Knotens v.

$$balance(v) = \frac{\text{Anzahl der Blätter in } T_l}{\text{Anzahl der Blätter in } T}$$

Die Balance jedes inneren Knotens muss dabei im Intervall $[\alpha, 1-\alpha]$ liegen, mit $\alpha \in]0, \frac{1}{2}]$. Durch diese geforderte Eigenschaft ist die Höhe logarithmisch. Auch hier wird über Rotation und Doppelrotation die Balance sichergestellt.
Laufzeit: Für alle Operationen ist die Laufzeit in $O(\log n)$.

3.3.6 (a,b)-Baum

Ein Baum ist ein (a,b)-Baum, wenn alle Blätter gleiche Tiefe haben und alle inneren Knoten $\geq a$ und $\leq b$ Kinder haben (Wurzel kann auch weniger als a Kinder haben). a und b müssen dabei im folgenden Verhältnis stehen: $a \geq 2$ und $b \geq 2a-1$. Die Anzahl der Blätter ist dabei beschränkt durch

$$2a^{h-1} \leq n \leq b^h$$
$$(h-1)\log a \leq \log n \leq h \log b$$

D.h. mit n Blättern hat man logarithmische Höhe. In ein einem (a,b)-Baum werden die Daten aufsteigend in den Blättern gespeichert, in jedem inneren Knoten werden dann $k-1$ Schlüssel verwaltet, wobei k die Anzahl der Kinder des Knotens ist. Die Schlüssel repräsentieren jeweils das größte Element des i-ten Teilbaums.
Laufzeit: Alle Operationen können in $O(\log n)$ durchgeführt werden.

3.3.7 Rot-Schwarz-Baum

Ein Rot-Schwarz-Baum ist ein Binärbaum, bei dem die Knoten rot oder schwarz gefärbt sind. Dabei sind alle Blätter schwarz gefärbt, ein roter Knoten hat zwei schwarze Kinder und auf dem Weg von der Wurzel zum Blatt liegen auf jedem Pfad gleich viele schwarze Knoten. Auch hier kann durch Rotation/ Doppelrotation die Balance hergestellt werden.
Laufzeit: Die Laufzeit ist auch hier $O(\log n)$ für alle Operationen des Wörterbuchs.

3.3.8 Tries

Ein Trie ist eine Datenstruktur über Σ^*, wobei Σ ein endliches Alphabet ist. Es ist ein Baum, in dem die Wörter von der Wurzel an nach unten verlaufen, so dass jeder Buchstabe in einem Knoten ist. Das Ende eines Wortes wird dann speziell durch ein Sonderzeichen markiert. Wörter mit gleichem Präfix teilen sich dann natürlich den gleichen Pfad am Anfang.
Laufzeit: Alle Operationen gehen in $O(|W|)$, wobei $|W|$ die Länge des Wortes ist. Man kann das noch etwas besser machen, in dem man Knoten zusammenfasst, zwischen denen keine Verzweigungen sind.

4 Sortieren / Suchen / Selektieren

4.1 Sortieren

4.1.1 Bogosort

Man erzeugt systematisch alle Permutationen und überprüft dann, ob diese sortiert ist.
Laufzeit: $n!$ viele Permutationen mit jeweils $n-1$ Vergleichen, also Laufzeit von $O(n!)$.

4.1.2 Selection-Sort

In jedem Durchlauf findet man das aktuelle Minimum und gibt es aus.
Laufzeit: Zum Finden des erste Minimums braucht man $n-1$ Vergleiche, für das zweite $n-2$ usw. Insgesamt hat man $\frac{n(n-1)}{2}$ Vergleiche, was $O(n^2)$ entspricht.

4.1.3 Merge-Sort

Rekursiver Algorithmus, der die zu sortierende Menge in zwei gleichgroße Teile splittet, diese rekursiv sortiert und die sortierten Teile dann zusammen mischt.
Laufzeit:

$T(1) = 0$ Kein Vergleich nötig bei nur einem Element

$$
\begin{aligned}
T(n) \leq & \underbrace{2T(\frac{n}{2})}_{\text{Rek. sortieren}} + \underbrace{n}_{\text{Mischen}} \\
& \leq 2(2T(\frac{n}{4}) + \frac{n}{2}) + n \\
& \leq 4T(\frac{n}{4}) + 2n \\
& \leq 4(2T(\frac{n}{8}) + \frac{n}{4}) + 2n \\
& \leq 8T(\frac{n}{8}) + 3n \\
& \leq 2^k T(\frac{n}{2^k}) + kn \\
& \leq 2^{\log n} T(\frac{n}{2^{\log n}}) + n \log n \\
& \leq n \log n
\end{aligned}
$$

Laufzeit ist also $O(n \log n)$.

4.1.4 Quicksort

Quicksort ist ein randomisierter Algorithmus. Hier wählt man zufällig ein Pivot-Element p, mit dem man alle Elemente vergleicht. Alle Elemente $< p$ werden in eine Menge S_1 gepackt, alle $= p$ in S_2 und alle anderen in Menge S_3. Die Folgen werden dann rekursiv sortiert und entsprechend $S_1\ S_2\ S_3$ zusammengefügt.

Laufzeit: Wir analysieren die Laufzeit im Mittel.

$$
\begin{aligned}
T(n) &= \frac{1}{n}\sum_{k=1}^{n})(T(k-1)+T(n-k))+(n-1)\\
&= \frac{2}{n}\sum_{k=0}^{n-1}T(k)+(n-1) \quad |*n\\
I.\ \ nT(n) &= 2\sum_{k=0}^{n-1}T(k)+n(n-1)\\
II.\ \ (n-1)T(n-1) &= 2\sum_{k=0}^{n-2}T(k)+(n-1)(n-2)\\
I.-II.\ \ nT(n)-(n-1)T(n-1) &= 2\sum_{k=0}^{n-1}T(k)+n(n-1)-\left(2\sum_{k=0}^{n-2}T(k)+(n-1)(n-2)\right)\\
nT(n)-(n-1)T(n-1) &= 2T(n-1)+n(n-1)-(n-1)(n-2)\\
nT(n)-(n-1)T(n-1) &= 2T(n-1)+2(n-1)\\
nT(n) &= (n+1)T(n-1)+2(n-1)\\
T(n) &= \frac{n+1}{n}T(n-1)+2\frac{n-1}{n}\\
&\leq \frac{n+1}{n}T(n-1)+2\\
&\leq \frac{n+1}{n}(\frac{n}{n-1}T(n-2)+2)+2\\
&\leq \frac{n+1}{n-1}T(n-2)+\frac{2(n+1)}{n}+2\\
&\leq \frac{n+1}{n-1}\left(\frac{n-1}{n-2}T(n-3)+2\right)+\frac{2(n+1)}{n}+2\\
&\leq \frac{n+1}{n-2}T(n-3)+\frac{2(n+1)}{n-1}+\frac{2(n+1)}{n}+2\\
&\leq \frac{n+1}{n-k+1}T(n-k)+\frac{2(n+1)}{n-k+2}+\frac{2(n+1)}{n-k+3}+\ldots+\frac{2(n+1)}{2n-k+1}\\
&\leq (n+1)T(0)+2(n+1)\underbrace{\left(\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n+1}\right)}_{\text{Abschätzbar durch harmonische Zahl } H_n}
\end{aligned}
$$

H_n ist abschätzbar durch $\ln n$, damit ist also Quicksort auch in $O(n \log n)$ im Mittel durchführbar (im schlechtesten Fall quadratisch).

- Vergleichsbasierte Algorithmen haben als untere Schranke der Laufzeit $O(n \log n)$. Dies kann man mit dem Vergleichsbaummodell zeigen. In einem Vergleichsbaum sind in den Blättern alle Permutationen der Eingabefolge aufgetragen, das sind

$n!$ viele. Die Höhe ergibt sich also als $\log n!$. $n!$ lässt sich abschätzen durch $\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$. Also $\frac{n}{2}\log\frac{n}{2} \leq \log n! \leq n \log n$.

- Auf endlichen Universen kann man auch in linearer Zeit sortieren (Counting Sort, Bucketsort, Radixsort)

4.2 Select

Man will aus einer unsortierten Folge das k-te Element finden.

4.2.1 Randomisiertes Select

```
SELECT(S,k) {
  if S == {a}
    return a
  p = wähle zufällig ein a aus S
  forall a in S {
    if a < p
      S_1 = S_1 + a
    else if a == p
      S_2 = S_2 + a
    else
      S_3 = S_3 + a
  }
  if |S_1| >= k
    SELECT(S_1,k)
  if |S_1| + |S_2| >= k
    return p
  SELECT(S_3, k - (|S_1|+|S_2|))
}
```

Laufzeit: Wir ermitteln die mittlere Laufzeit.

$$
\begin{aligned}
T(1) &= b \\
T(n) &= \frac{1}{n}\left(\underbrace{\sum_{p=1}^{k-1}(T(n-p)}_{p<k\Rightarrow \text{ Suchen in } S_2} + \underbrace{\sum_{p=k+1}^{n} T(p-1)}_{p>k\Rightarrow \text{ Suchen in } S_1}\right) + cn \\
&\leq \frac{2}{n}\sum_{i=\lfloor \frac{n}{2}\rfloor}^{n-1} T(i) + cn \quad \mid \text{für } k \geq \lfloor\frac{n}{2}\rfloor
\end{aligned}
$$

Wir behaupten, dass $T(n) = O(n)$, d.h. $\exists d > 0 : T(n) \leq dn$. Für führen einen Induktionsbeweis.

Induktionsanfang $n = 1$, funktioniert, wenn $d \geq b$ gewählt.

Induktionsschritt $n - 1 \to n$.

$$
\begin{aligned}
T(n) &\leq \frac{2}{n} \sum_{\lfloor \frac{n}{2} \rfloor}^{n-1} T(i) + cn \quad \mid \text{I.V} \\
&\leq \frac{2}{n} \sum_{\lfloor \frac{n}{2} \rfloor}^{n-1} di + cn \quad \text{mit Gaußsumme abschätzen} \\
&\leq \frac{2d}{n} \left(\frac{n(n-1)}{2} - \frac{(\frac{n}{2}-1)(\frac{n}{2}-2)}{2} \right) + cn \\
&\leq \frac{2d}{n} \left(\frac{n^2-n}{2} - \frac{n^2}{8} + \frac{n}{2} + \frac{n}{4} - 1 \right) + cn \\
&\leq \frac{2d}{n} \left(\frac{3n^2}{8} + \frac{n}{4} \underbrace{-1}_{\text{Wegfallen lassen}} \right) + cn \\
&\leq \frac{3dn}{4} + \frac{d}{2} + cn \quad 0,75n + 0,5 \leq 0,8n \text{ für } n \geq 10, \text{ also} \\
&\leq 0,8dn + cn
\end{aligned}
$$

Es ist nun $0,8dn + cn \leq dn$ wenn $5c \leq d$. Somit ist die Laufzeit in $O(n)$. Im schlechtesten Fall kann man auch hier zu einer quadratischen Laufzeit kommen.

4.2.2 Deterministisches Select

Wir wählen das Pivot-Element deterministisch so aus, dass eine lineare Laufzeit immer garantiert wird.

```
DETSELECT(S,k) {
  if |S| < n_0
    Bestimme Element per Brute-Force
  else {
    Teile S in n/5 (abgerundet) Blöcke
    Bestimme von jedem Block den Median
    p = Median der Mediane (rekursiver Aufruf)
    forall a in S {
      if a < p
        S_1 = S_1 + a
      else if a == p
        S_2 = S_2 + a
      else
        S_3 = S_3 + a
    }
```

```
    if |S_1| >= k
      SELECT(S_1,k)
    if |S_1| + |S_2| >= k
      return p
    SELECT(S_3, k - (|S_1|+|S_2|))
  }
}
```

Laufzeit: Wir überlegen uns, wie groß S_1 und S_3 sein können. Es sind (siehe Bild) $n - 3 * \frac{1}{2} * \lfloor \frac{n}{5} \rfloor = n - 3\lfloor \frac{n}{10} \rfloor$ Elemente. Für $n \geq 60$ ist dies $\leq 0,75n$. Damit ist die Rekursionsgleichung

$$T(n) \leq T(\lfloor \frac{3}{4}n \rfloor) + T(\lfloor \frac{n}{5} \rfloor) + cn$$

Es gilt für $T(n) \leq T(\lfloor \alpha n \rfloor) + T(\lfloor \beta n \rfloor) + cn$ mit $\alpha + \beta < 1$, dass $T(n) = O(n)$.
Somit ist die Laufzeit immer in $O(n)$.

4.3 Suchen

Man hat ein statisches, sortiertes Feld, in dem ein Element gesucht werden soll.

4.3.1 Binärsuche

```
BINSEARCH(S,a) {
  k = |S| / 2
  if S[k] == a
    return true
  else if S[k] > a
    BINSEARCH(S[0]..S[k-1],a)
  else
    BINSEARCH(S[k+1]..S[|S|-1],a)
}
```

Laufzeit: Die Laufzeit ist hier $O(\log n)$, da im jedem Schritt die Menge halbiert wird.

4.3.2 Interpolationssuche

Statt wie bei der Binärsuche als k immer die Mitte zu nehmen, versucht man die Position besser abzuschätzen (sofern die Folge einigermaßen gleichverteilt ist). Wir bilden die Folge auf das Intervall zwischen $[0; 1]$ ab, wobei das erste Elment 0 ist und das $n + 1$-te 1.

$$k = l - 1 + \lceil \frac{a - S[l-1]}{S[r-1] - S[l-1]} * (r - l + 1) \rceil$$

Laufzeit: Im schlechtesten Fall hat man lineare Laufzeit, im mittel allerdings $O(\log \log n)$.

4.3.3 Quadratische Binärsuche

Wir berechnen das k wie bei der Interpolationssuche. Wenn wir das Element nicht treffen, dann springen wir in $\sqrt{m}$ mit $m = r - l + 1$ Abstand nach links oder rechts, so lange, bis sich das Vorzeichen für den Vergleich ändert. Dadurch schränkt man dann den Suchbereich auf die Breite von $\sqrt{m}$ ein.
Man kann berechnen, dass man im Schnitt nur $2,41$ solcher Sprünge macht.
Laufzeit:

$$\begin{aligned}
T(n) &= T(\sqrt{n}) + c \quad \text{wir nehmen an } n = 2^{2^k} \\
T(2^{2^k}) &= T(2^{2^{k-1}}) + c \\
&= T(2^{2^{k-2}}) + 2c \\
&= T(2^{2^{k-i}}) + ic \\
&= T(2) + kc \quad \text{für } k = i \text{ folgt } n = 2^{2^k} \\
&= T(2) + c \log\log n
\end{aligned}$$

Die Laufzeit ist also in $O(\log\log n)$.

5 String-Matching

Man hat einen Text $T \in \Sigma^*$ und ein Muster $P \in \Sigma^*$ und will alle Vorkommen von P in T finden.

5.1 Naiver Ansatz

Man legt für jede Stelle von T das Muster an und vergleicht Zeichenweise. Das führt zu einer Laufzeit von $O(|T| * |P|)$.

5.2 Knuth-Morris-Pratt

Die Idee ist hierbei, dass man das Muster um so viele Stellen weiterschiebt, wie es keine Übereinstimmungen geben kann. So wird eine lineare Laufzeit erreicht. Um das zu erreichen, muss man das Muster analysieren.

5.2.1 Präfixfunktion

Die Präfixfunktion gibt für jedes Präfix eines Wortes an, wie lang der maximale Suffix dieses Präfixes ist, der ebenfalls Präfix vom Präfix ist.
Laufzeit: Die Präfixfunktion kann in linearer Zeit berechnet werden.

```
KNUTH-MORRIS-PRATT(T,P) {
  Pi = Berechne Präfixfunktion von P
  q = 0
  for i = 1 to n {
```

```
    while q > 0 && P[q+1] != T[i] do
      q = Pi(q)
    }
    if P[q+1] = T[i]
      q = q+1
    if q == m
      Match an Stelle i-m gefunden
      q = Pi(q)
  }
}
```

Laufzeit: Die Präfixfunktion lässt sich in $O(|P|)$ berechnen, die `for`-Schleife iteriert $n = |T|$ mal. Interessant ist die `while`-Schleife: Sei $s = i - q - 1$ der Abstand vom Anfang von P zum Anfang von T. Da q durch die Zuweisung in der `while`-Schleife nur kleiner werden kann, wird also s mit jeder Iteration größer. Da $s \leq |T| - |P]$ ist, wird sie maximal $|T| - |P| + |T| = O(|T|)$ mal ausgeführt. Das macht eine Laufzeit von $O(|T| + |P|)$ insgesamt.

5.3 Suffixbaum

Ein Suffixbaum ist eine Datenstruktur für einen festen Text T mit $|T| = n$. Der Baum hat genau n Blätter, die von 1 bis n durchnummeriert sind. Jeder innere Knoten hat mind. 2 Kinder, die Kanten dabei sind mit nichtleeren Teilwörtern von T beschrieben und zwar so, dass zwei Kanten, die von einem Knoten abgehen, keine zwei gleichen Anfänge haben. Für das Blatt i gilt, dass auf dem Pfad von der Wurzel zu dem Blatt das Wort $T[i] \dots T[n]$ steht, also das i-te Suffix.
Man findet nun ein Muster P in T, wenn P Präfix eines Suffixes ist. Die Stelle, an der P anfängt ist dann an dem Index im Blatt abzulesen.
Laufzeit: Ein Suffixbaum kann in $O(|T|^2)$ Zeit aufgebaut werden und benötigt auch quadratischen Speicherplatz. Um ein Muster zu suchen benötigt, folgt man der Wurzel bis zu einem Knote v oder einem Blatt. Wenn man in einem Knoten landet, dann sind alle Blätter des Teilbaums mit Wurzel v Positionen des Musters im Text. Die Laufzeit ist dann $O(|P| + r)$, wobei r die Anzahl der Vorkommen von P im Text ist.