Hasso-Plattner-Institut Potsdam
FG Algorithm Engineering
Pascal Lenzner

Material zur Vorlesung
Algorithmix WS 17/18
Potsdam, den 7. Februar 2018

Skript Algorithmix WS 2017/18

1 Einleitung

Die Vorlesung „Algorithmix“ hat das Ziel eine grundlegende *Denkweise* für den Entwurf von Algorithmen zu vermitteln. Vor allem soll eine strukturierte und kritische Herangehensweise an algorithmische Probleme eingeübt werden. Weiterhin wiederholt bzw. festigt diese Vorlesung grundlegende Algorithmen, Datenstrukturen und viele wichtige Analyse- und Beweistechniken. Neben eher klassischen Themen wird es Exkurse in andere algorithmische Themengebiete geben. Diese Exkurse dienen dazu, um Einblick in die thematische Breite der Algorithmik zu geben und auf Spezialvorlesungen vorzubereiten.
Ein Algorithmus ist eine endlich lange eindeutige Handlungsvorschrift zur Lösung eines genau spezifizierten Problems. Ähnlich wie bei einem Kochrezept[1], beschreibt ein Algorithmus genau, wie ein spezielles Problem schrittweise gelöst wird und wie die zugehörigen Daten dafür verarbeitet werden müssen. Meist soll eine spezielle *Eingabe*, d.h. eine Instanz eines zu lösenden Problems, mit Hilfe des Algorithmus in eine passende *Ausgabe* umgewandelt werden. Dieser Prozess soll möglichst immer terminieren, d.h. die Ausgabe wird nach endlich langer Zeit tatsächlich produziert, und die Ausgabe soll die gewünschten Eigenschaften haben, d.h. der Algorithmus soll korrekt sein.
Es gibt zu den meisten Anwendungsproblemen in der Informatik unzählige Lösungsvarianten und die daraus resultierenden algorithmischen Vorgehensweisen. Meist will man den bestmöglichen Algorithmus nutzen, doch es stellt sich schnell die Frage, wie man unterschiedliche Lösungsstrategien miteinander vergleichen kann, um eine Beste unter ihnen zu ermitteln. Es ist somit wünschenswert die Güte von Algorithmen bewerten zu können, doch um zu solch einem Urteil zu gelangen, ist eine sorgfältige Analyse des Algorithmus und der zugehörigen *Datenstruktur* nötig. Die gewonnenen Erkenntnisse tragen dann meist zu optimierten Strategien mit einer passenden und effizienten Datenstruktur bei und sind die Grundlage für eine praxistaugliche Lösung der zu bewältigenden Probleme. Algorithmen sind immer auf eine für das vorliegende Problem angepasste Repräsentation der Daten angewiesen und somit sind Algorithmen und Datenstrukturen untrennbar. *Zu jedem guten Algorithmus gehört also eine passende Datenstruktur.*

1.1 Motivation

Algorithmen sind *das* zentrale Thema der Informatik. In allen Teilgebieten sind sie anzutreffen und auf ihnen beruht die Leistungsfähigkeit der Computer. Doch wie sieht es

[1] Leider sind viele Kochrezepte zu ungenau, deshalb eignet sich der Vergleich nur bedingt.

mit dem technischen Fortschritt aus? Brauchen wir gute Algorithmen, auch wenn wir unglaublich leistungsfähige Rechner haben? Die Antwort darauf lautet eindeutig ja und anhand des Problem des Handelsreisenden (*Travelling Salesman Problem* (kurz: TSP)) soll dies verdeutlicht werden.

Beispiel 1. *Informell kann man das Problem des Handelsreisenden wie folgt beschreiben:*

> *TSP: Auf einer gegebenen Landkarte mit Städten und Straßen, welche diese miteinander verbinden, soll die kürzeste Rundtour durch alle Städte bestimmt werden. Dabei soll aber jede Stadt nur genau einmal besucht werden.*

Es soll also die Summe der Distanzen zwischen den Stationen der Rundtour minimiert werden. Instanzen des TSP treten in vielen Anwendungen auf, von der Planung einer realen Rundreise bis zur Optimierung der Bewegung eines Bohrers bei der Herstellung von Platinen.
Eine Lösungsmöglichkeit besteht darin, dass man die Länge aller Rundtouren berechnet und dann die Kürzeste als Lösung ausgibt. Allerdings gibt es für n Städte n! viele Rundtouren[2] *und dies ist eine rasant wachsende Funktion. Beispielsweise konnte man im Jahre 1975 die optimale Tour für etwa 135 Städte in akzeptabler Zeit berechnen, doch der enorme Fortschritt der Technik bis heute hat nur zu einer Verbesserung auf etwa 165 Städte geführt. Die Zahl der zu testenden Möglichkeiten steigt so schnell, dass eine tausendfache Verbesserung der Rechenleistung nur eine Instanz des Problems mit höchstens 10 weiteren Städten lösen kann*[3].
Allerdings kann man heute durch algorithmische Fortschritte bereits optimale Rundtouren für einige Instanzen mit mehr als 15000 Städten berechnen, z.B. für 15112 Städte in Deutschland. Hier ist die kürzeste Rundtour, die jede Stadt genau ein mal besucht[4]*:*

[2] $n!$ ist die Anzahl der Permutationen von n Elementen und jede Permutation ergibt eine Rundtour.

[3] Die Zeit, die ein solcher Algorithmus für die Berechnung benötigt, kann man für n Städte etwa mit 2^n abschätzen. Eine 1000-fache Beschleunigung bringt also einen Gewinn von $\log_2 1000 \approx 10$ Städten. Die Rechenleistung von 1975 bis heute hat sich etwa um den Faktor 10^9 verbessert.

[4] Siehe: http://www.math.uwaterloo.ca/tsp/d15sol/index.html

Vielleicht wird daran deutlich, wie dramatisch sich Verbesserungen an den zugrunde liegenden Algorithmen auswirken können. In vielen Fällen führen verfeinerte Algorithmen überhaupt erst dazu, dass ein Problem gelöst werden kann.

1.2 Warm-Up: Kidney Exchange

Als erstes algorithmisches Problem der Vorlesung betrachten wir ein Problem, von dessen Lösung in der Praxis tatächlich Leben abhängen. Es geht um die Zuweisung von Spenderorganen an Patienten, speziell geht es hierbei um Nieren. Wird bei einem Patient Nierenversagen festgestellt, so benötigt dieser dringed eine neue Niere. Menschen haben zwei Nieren, benötigen aber eigentlich nur eine - somit ist es für die Betroffenen möglich z.B. bei Verwandten oder Freunden einen Spender zu finden. Es gibt jedoch ein Problem: Nicht jede Niere ist mit jedem Patienten kompatibel. Ob ein Spenderorgan vom Körper angenommen wird hängt von vielen Faktoren, z.B. der Blutgruppe, ab. Diese Kompatibilität zwischen Spenderorgan und Patient kann man vor einer potentiellen Operation ermitteln. Die Idee ist nun, die Spender zu tauschen, so dass alle Patienten ein für sie besseres Spenderorgan erhalten.
Wir haben also folgende Situation vorliegen: Es gibt n Patienten $P = \{p_1, \ldots, p_n\}$, die eine neue Niere benötigen und es gibt n Spendernieren $N = \{n_1, \ldots, n_n\}$, wobei Niere n_i vom Spender von Patient p_i bereitgestellt wird.
Wie bereits erwähnt, ist es den Patienten nicht egal, welche Spenderniere sie zugewiesen bekommen. Wir modellieren dies, indem wir annehmen, dass jeder Patient eine *Präferenzliste* über die Spendernieren hat. D.h. für jeden Patient p_i gibt es eine Liste

$$L_i = n_1^i, n_2^i, \ldots, n_n^i,$$

welche eine Permutation von N ist. Hierbei ist n_1^i die Niere, die am beste mit Patient p_i kompatibel, Niere n_2^i ist die zweitbeste, usw[5]. Im Folgenden nennen wir das erste Element der Liste L_i die *Top-Präferenz* von Patient p_i. Wir führen noch eine Kurzschreibweise ein. Für beliebige Nieren $n_a, n_b \in N$ schreiben wir

$$n_a \geq^i n_b,$$

falls die Niere n_a mindestens so kompatibel mit Patient p_i ist, wie Niere n_b[6]. Falls in L_i das Niere n_a vor n_b vorkommt, dann schreiben wir

$$n_a >^i n_b.$$

Nun kommen wir zum algorithmischen Problem: Wir erhalten von allen Patienten ihre Spender und ihre Präferenzlisten und nun wollen wir die Spendernieren so verteilen, dass alle Patienten eine Niere bekommen haben und mit ihrer *Zuweisung* zufrieden sind.[7]
Eine Zuweisung ist hierbei eine Permutation von T, welche wir als n-dimensionalen Vektor

$$\mathbf{z} = (z_1, \ldots, z_n)$$

[5]Es gibt somit keine Nieren, die für p_i als gleichwertig eingestuft werden!

[6]$a = b$ ist hier durchaus möglich.

[7]Selbstverständlich werden hier zunächst die Spender zu Patienten zugewiesen und die nötigen Operationen erst später durchgeführt. Wir behandeln dies hier aber synonym und gehen davon aus, dass die Spendernieren direkt zugewiesen werden.

aufschreiben, wobei hier der i-te Eintrag des Vektors $\mathbf{z}$ die zugewiesene Niere des i-ten Patienten ist[8].
Um das Problem angehen zu können, müssen wir zunächst klären, wann genau ein Patient zufrieden mit einer Zuweisung ist. Betrachten wir dazu ein Beispiel:

Beispiel 2. *Wir betrachten drei Patienten p_1, p_2, p_3 mit folgenden Präferenzlisten:*

$$L_1 : n_3, n_2, n_1$$
$$L_2 : n_1, n_2, n_3$$
$$L_3 : n_2, n_3, n_1.$$

Angenommen die Zuweisung wäre (n_1, n_2, n_3). In diesem Fall wären beispielsweise p_1 und p_2 nicht mit der Zuweisung zufrieden, denn sie könnten ihre Spendernieren tauschen und beide davon profitieren!
Betrachten wir, was nach dem Tausch passiert. Wir haben dann die Zuweisung (n_2, n_1, n_3). Patient p_2 hat damit die für ihn beste Niere erhalten und wird folglich keinem weiteren Tausch mehr zustimmen. Allerdings könnten die anderen beiden Patienten nochmal tauschen, um sich beide zu verbessern.
Damit erhalten wir die Zuweisung (n_3, n_1, n_2), welche allen Patienten ihre bestkompatible Niere zuweist. Mit dieser Zuweisung sind garantiert alle zufrieden. Übrigens hätten wir die Zuweisung auch erhalten, wenn ausgehend von (n_1, n_2, n_3) alle drei Patienten zyklisch getauscht hätten, d.h. p_3 erhält n_2 von p_2, p_2 erhält n_1 von p_1 und p_1 erhält n_3 von p_3.
Es ist leicht zu sehen, dass Zuweisung (n_3, n_1, n_2) die einzige ist, in der alle Patienten zufrieden sind: Wir haben schon Zuweisungen (n_1, n_2, n_3) und (n_2, n_1, n_3) eliminiert. Es verbleiben noch

- *(n_1, n_3, n_2), hier könnten p_1 und p_2 bzw. p_2 und p_3 tauschen.*
- *(n_2, n_3, n_1), hier könnten p_1 und p_2 bzw. p_2 und p_3 tauschen.*
- *(n_3, n_2, n_1), hier könnten p_2 und p_3 tauschen.*

$\triangleleft$

Beispiel 2 zeigt, dass es durchaus möglich ist, dass jeder Patient die für ihn/sie bestmögliche Niere bekommt.
Aus dem Beispiel können wir eine mögliche Definition für die Zufriedenheit der Patienten ableiten. Im Beispiel waren Patienten unzufrieden, wenn sie durch einen Nierentausch mit einem oder mehreren willigen Tauschpartnern eine bessere Niere erhalten können. Dies machen wir jetzt formal.

Formale Definition der Zufriedenheit Sei $Perm(X)$ die Menge aller Permutationen über der Menge X. Somit ist $Perm(N)$ die Menge aller möglichen Zuweisungen der Spendernieren, da jede Permutation von N einer Zuweisung der Nieren an die Patienten entspricht, in der jeder Patient genau eine Spenderniere bekommt.

[8]Wir benutzen die Vektorenschreibweise hauptsächlich, um eine Zuweisung von einer Präferenzliste zu unterscheiden. Beides sind nur Permutationen der Elemente aus N.

Sei $\mathbf{z} = (z_1, \ldots, z_n) \in Perm(N)$ eine beliebige Zuweisung. Für eine beliebige Teilmenge der Patienten $K \subseteq P$ sei $T(\mathbf{z}, K)$ die Menge der Spendernieren, die unter Zuweisung $\mathbf{z}$ an Patienten aus K zugewiesen wurden, d.h.

$$T(\mathbf{z}, K) = \{n \in N \mid \text{es gibt ein } p_j \in K \text{ mit } z_j = n\}.$$

Wir sagen dass K eine *unzufriedene Koalition* für die Zuweisung $\mathbf{z}$ ist, falls die Patienten aus K untereinander Nieren tauschen können und sich dabei alle Patienten echt verbessern. Formal:

Definition 1 (Unzufriedene Koalition für $\mathbf{z}$)**.** *K ist eine* unzufriedene Koalition für die Zuweisung $\mathbf{z}$*, falls eine Zuweisung $\mathbf{x} \in Perm(N)$ existiert, so dass*

(1) $T(\mathbf{x}, K) = T(\mathbf{z}, K)$ und

(2) für alle $p_i \in K$ gilt $x_i >^i z_i$.

Falls für Zuweisung $\mathbf{z}$ eine unzufriedene Koalition K existiert, dann sind folglich mindestens zwei Patienten aus K unzufrieden. Die Menge der Zuweisungen, mit welchen alle Patienten zufrieden sind, ist somit die Menge der Zuweisungen für die es *keine* unzufriedene Koalition gibt. Wir nennen solche Zuweisungen *schwach stabil*, da die Patienten kein Interesse haben, eigenhändig etwas an einer schwach stabilen Zuweisung zu ändern.

Definition 2 (Schwach stabile Zuweisung)**.** *Eine Zuweisung $\mathbf{z}$ heißt* schwach stabil*, falls es keine unzufriedene Koalition für $\mathbf{z}$ gibt.*

Wir ein weiteres Beispiel, welches zeigt, dass viele schwach stabile Zuweisungen möglich sind:

Beispiel 3. *Was würde passieren, wenn die Präferenzlisten wie folgt aussähen?*

$$L_1 : n_1, n_2, n_3$$
$$L_2 : n_1, n_2, n_3$$
$$L_3 : n_1, n_2, n_3.$$

Offensichtlich gibt es hier keine Zuweisung, so dass alle Patienten die für sie beste Niere erhalten.
Betrachten wir die Zuweisung (n_1, n_2, n_3). Patient p_1 ist zufrieden, denn er/sie bekommt die für ihn/sie bestmögliche Niere. Patient p_2 bekommt seine/ihre zweitbeste Niere, doch da Patient p_1 sich keinesfalls von seiner Spenderniere n_1 trennen würde (jedes andere Niere wäre eine Verschlechterung), besteht keine Chance auf eine bessere Niere, folglich muss auch p_2 mit Niere n_2 zufrieden sein (ein Tausch mit p_3 kommt für p_2 nicht in Frage). Somit muss p_3 zufrieden mit seiner Niere n_3 sein, da sich kein Tauschpartner findet. Zuweisung (n_1, n_2, n_3) ist somit schwach stabil, da keine unzufriedene Koalition existiert.
Interessanterweise gilt dies aber auch für alle anderen Zuweisungen! D.h. auch die Zuweisungen (n_1, n_3, n_2), (n_2, n_1, n_3), (n_2, n_3, n_1), (n_3, n_1, n_2) und (n_3, n_2, n_1) sind schwach stabil! ◁

Beispiel 3 zeigt, dass sogar alle möglichen Zuweisungen schwach stabil sein können. In dem Fall stimmen die Präferenzlisten aller Patienten überein. Die Frage ist nun, welche der vielen möglichen schwach stabilen Zuweisungen die beste ist?

Berücksichtigung der Spender: Die Frage kann leicht beantwortet werden. Bisher haben wir komplett ignoriert, dass jeder Patient ja sozusagen einen *eigenen* Spender hat, den er/sie persönlich kennt. D.h. wir starten eigentlich nicht von einer leeren Zuweisung, sondern wir haben eine *Startzuweisung* vorliegen!
Die Startzuweisung lautet $\mathbf{z}^* = (n_1, \ldots, n_n)$. Da somit jeder einen *eigenen* Spender hat, gehen im Folgenden davon aus, dass Patient p_i auf jeden Fall die Niere n_i sicher hat. Dies führt uns zu einer stärkeren Definition für die Stabilität einer Zuweisung:
Wir sagen dass K eine *blockierende Koalition* für die Zuweisung $\mathbf{z}$ ist, falls die Patienten aus K untereinander ihre Nieren aus $\mathbf{z}^*$ tauschen können und sich dabei im Vergleich zu $\mathbf{z}$ kein Patient verschlechtert und sich mindestens ein Patient echt verbessert. Formal:

Definition 3 (Blockierende Koalition für $\mathbf{z}$)**.** *K ist eine* blockierende Koalition für die Zuweisung $\mathbf{z}$*, falls eine Zuweisung $\mathbf{x} \in Perm(N)$ existiert, so dass*

(1) $T(\mathbf{x}, K) = T(\mathbf{z}^*, K)$ und

(2) für alle $p_i \in K$ gilt $x_i \geq^i z_i$ und

(3) es existiert ein $p_j \in K$ mit $x_j >^j z_j$.

Der Unterschied zu einer unzufriedenen Koalition ist der Referenzpunkt, nämlich die Anfangszuweisung $\mathbf{z}^*$.
Wir können uns die Situation wie folgt vorstellen: Zunächst übermitteln alle Patienten ihre Präferenzliste an eine zentralle Koordinationsstelle. Diese führt dann einen Algorithmus aus, der eine neue (hoffentlich bessere) Zuweisung berechnet. Diese neue Zuweisung wird dann von der Koordinationsstelle vorgeschlagen. Die neue Zuweisung wird genau dann von allen Patienten akzeptiert, wenn sich keine Koalition von Patienten durch Tausch untereinander verbessern kann und somit die neue Zuweisung blockiert. Eine Koalition blockiert eine Zuweisung, wenn sie ausgehend von der Anfangszuweisung eine für alle an der Koalition beteiligten Patienten mindestens gleich gute Zuweisung durch Tausch untereinander erzielen kann, wobei sich mindestens ein Patient echt verbessert. Anders formuliert: Die Koordinationsstelle soll eine Zuweisung vorschlagen, so dass keine Koalition von Patienten einen Anreiz hat, die Organverteilung untereinander (und somit außerhalb des Systems auf einem Schwarzmarkt) durchzuführen.
Zuweisungen, die von allen Patienten in dem Sinne akzeptiert werden, nennen wir *stabile Zuweisung.*

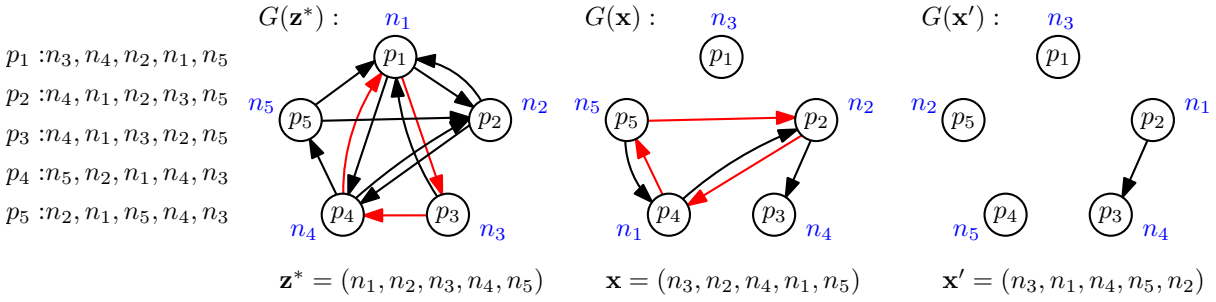
Definition 4 (Stabile Zuweisung)**.** *Eine Zuweisung $\mathbf{x}$ heißt* stabil*, wenn es keine blockierende Koalition für $\mathbf{x}$ gibt.*

Betrachten wir ein Beispiel, um auf den Unterschied zwischen schwach stabilen und stabilen Zuweisungen einzugehen.

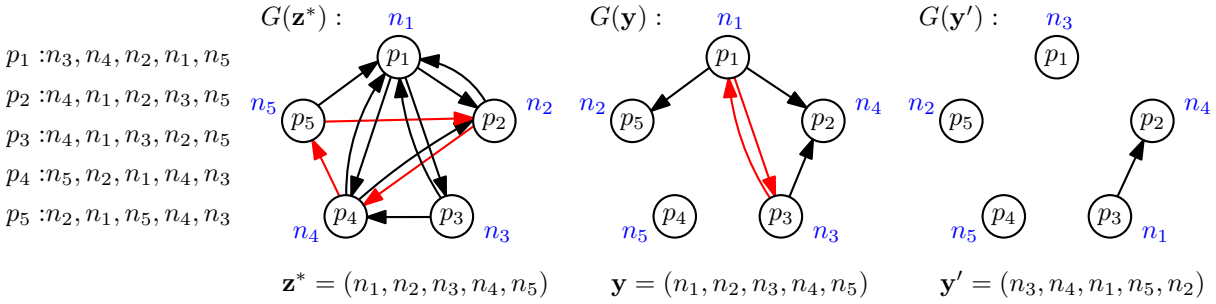
Beispiel 4. *Wir betrachten ein Beispiel mit fünf Patienten und betrachten dazu den sogenannten* Tauschgraph $G(\mathbf{x})$*, welcher wie folgt definiert ist: $V = P$, d.h. für jeden Patienten gibt es einen Knoten in $G(\mathbf{x})$. Die Kantenmenge ist wie folgt definiert*

$$E = \{(p_i, p_j) \mid x_j >^i x_i\},$$

d.h. es existiert eine gerichtete Kante von Knoten p_i zu p_j, falls p_i die Niere x_j besser verträgt als seine/ihre aktuell zugewiesene Niere x_i. Ein gerichteter Kreis im Tauschgraph entspricht somit einem möglichen zyklischen Tausch der Nieren unter den jeweiligen Patienten. Der Tausch wird hierbei entgegen der Pfeilrichtungen durchgeführt.



Die Zuweisung $\mathbf{x}'$ ist schwach stabil, da in $G(\mathbf{x}')$ keine gerichteten Kreise mehr vorkommen. Allerdings ist $\mathbf{x}'$ nicht *stabil: Die Menge $\{p_2, p_4, p_5\}$ ist eine blockierende Koalition für $\mathbf{x}'$, da sie ausgehend von der Ausgangszuweisung $\mathbf{z}^* = (n_1, n_2, n_3, n_4, n_5)$ durch Tausch untereinander die Zuweisung $\mathbf{x}'' = (n_1, n_4, n_3, n_5, n_2)$ erzeugen könnten. Im Vergleich zu $\mathbf{x}'$ würden sich unter Zuweisung $\mathbf{x}''$ der/die Patienten p_2 verbessern und p_4 und p_5 würden sich nicht verschlechtern.*
Dies ist ein paradoxes Phänomen! Zuweisung $\mathbf{x}'$ wurde erzielt, indem in jedem Schritt immer alle am Tausch beteiligten Patienten profitiert haben. Im zweiten Schritt hat sogar die Koalition $\{p_2, p_4, p_5\}$ dem Tausch zugestimmt. Trotzdem endet der Prozess in einer Zuweisung, die Koalition $\{p_2, p_4, p_5\}$ blockiert.
Schauen wir, was passiert, wenn wir nochmals andere Tauschkreise auswählen, diesmal lassen wir die Koalition $\{p_2, p_4, p_5\}$ zuerst Tauschen:



Die ermittelte Zuweisung $\mathbf{y}'$ ist nicht nur schwach stabil, sie ist auch stabil! Die Patienten p_1, p_2, p_4 und p_5 haben ihre bestmögliche Niere erhalten, somit kann keine Teilmenge dieser Patienten eine blockierende Koalition bilden, da sich niemand echt verbessern könnte. Wir müssen also nur potentielle blockierende Koalitionen betrachten, die Patient p_3 beinhalten. Die einzige Möglichkeit, die zu einer Verbesserung für p_3 führt, wäre eine Koalition, die p_3 und p_2 enthält. Allerdings müsste dann p_2 sein/ihre beste Niere abgeben und würde sich somit verschlechtern. Folglich gibt es keine blockierenden Koalitionen für Zuweisung $\mathbf{y}'$.
◁

Das Beispiel zeigt einerseits, dass nicht jede schwach stabile Zuweisung auch stabil sein muss. Andererseits kann es stabile Zuweisungen geben, die auch schwach stabil sind.[9]

[9]Wir werden uns in der Übung genauer damit beschäftigen, wie sich schwach stabile und stabile Zuweisungen zueinander verhalten!

Nun kommen wir zum algorithmischen Problem: Wir wollen einen Algorithmus entwickeln, der uns eine stabile Zuweisung konstruiert - sofern es diese überhaupt immer gibt.

Algorithmisches Problem:

> **Geg.:** n Präferenzlisten der Patienten und die Anfangszuweisung $\mathbf{z}^*$.
>
> **Aufgabe:** Berechne eine stabile Zuweisung.

Idee: In Beispiel 4 haben wir gesehen, dass die richtige Wahl der Tauschkreise zu einer stabilen Zuweisung führen kann. Betrachten wir den gewählten Tauschkreis in $G(\mathbf{z}^*)$ aus dem Beispiel etwas genauer, dann sehen wir, dass jede der Kanten im Tauschkreis eine Kante zur besten Niere für den entsprechenden Patienten zeigt. Einen solchen Kreis nennen wir *Top-Präferenz-Kreis*. Solche Kreise in $G(\mathbf{z}^*)$ sind interessant, da sie uns etwas über mögliche blockierende Koalitionen sagen: Falls in Zuweisung $\mathbf{x}$ die Patienten in einem Top-Präferenz-Kreis nicht alle ihre bestmögliche Niere erhalten, dann bildet diese Menge von Patienten eine blockierende Koalition für $\mathbf{x}$, denn durch Tausch untereinander könnte jeder dieser Patienten seine bestmögliche Niere erhalten und somit kann sich niemand bezüglich $\mathbf{x}$ verschlechtern und mindestens ein Patient verbessert sich. So lange es Top-Präferenz-Kreise gibt, müssen wir also auch entlang dieser Kreise tauschen!
Für jeden Tausch entlang eines Top-Präferenz-Kreises gilt, dass wir die beteiligten Patienten nicht mehr betrachten müssen - sie würden sich ohnehin nicht mehr an irgendwelchen Tauschen beteiligen. Top-Präferenz-Kreise führen also zusätzlich auch noch dazu, dass unser Problem kleiner wird!
Gehen wir einen Schritt weiter und nehmen an, dass wir entlang aller Top-Präferenz-Kreise getauscht haben. Die zugehörigen Patieten werden ihre Spendernieren nicht mehr freigeben, somit können wir die beteiligten Nieren von allen Präferenzlisten streichen. Damit ergeben sich neue (leicht modifizierte) Präferenzlisten und wir können den Prozess iterieren.

Der Top-Trading-Cycle Algorithmus Die oben beschriebene Idee lässt sich leicht algorithmisch umsetzen und ist unter dem Name Top-Trading-Cycle Algorithmus bekannt. Er wurde von David Gale Anfang der 60er Jahre entdeckt und später von Lloyd Shapley und Al Roth weiter untersucht.
Um Top-Präferenz-Kreise zu finden, müssen wir nicht den kompletten Tauschgraphen betrachten. Es genügt, wenn wir für jeden Patienten genau eine Kante zu seiner/ihrer bestmöglichen Nieren (genauer gesagt zum Patient, dessen eigener Spender diese Niere har) im Graph haben. Da Patienten auch ihre anfangs zugewiesene Niere als Top-Präferenz haben können, müssen wir in diesem Graph auch Schleifen[10], d.h. Kanten den Form (p_i, p_i), zulassen.

Definition 5 (Top-Präferenz-Graph)**.** *Für eine Teilmenge $X \subseteq P$ und Anfangszuweisung $\mathbf{z}^* = (n_1, n_2, \ldots, n_n)$ nennen wir den zugehörigen Graph $H(X) = (X, E_X)$* Top-Präferenz-Graph*. Er ist wie folgt definiert: Die Knotenmenge ist X und*

$$E_X = \{(p_i, p_j) \mid p_j \in X \text{ und } n_j \text{ ist beste verfügbare Niere für } p_i\}.$$

[10]Schleifen werden meist self-loop genannt.

Insbesondere ist $H(\emptyset)$ der leere Graph $(\emptyset, \emptyset)$.

Nach Definition hat der Top-Präferenz-Graph $H(X)$ genau $|X|$ viele Kanten, da jeder Patient genau eine bestkompatible Niere aus der Menge der Nieren, die unter Zuweisung z^* an Patienten aus X vergeben wurden, hat.
Nun sind wir bereit für den Algorithmus:

TopTradingCycle(n Präferenzlisten, $\mathbf{z}^*$)

Input: n Präferenzlisten, Anfangszuweisung $\mathbf{z}^*$
Output: Stabile Zuweisung

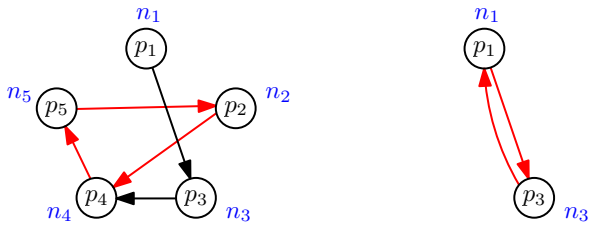
1: $X \leftarrow P$
2: $H \leftarrow$ konstruiere Top-Präferenz-Graph $H(X)$
3: **while** H enthält einen gerichteten Kreis mit Knotenmenge C **do**
4: weise jedem Patienten in C die Niere zu, auf die seine/ihre ausgehende Kante zeigt[a]
5: $X \leftarrow X \setminus C$
6: aktualisiere $H(X)$
7: **end while**
8: **return** berechnete Zuweisung

[a] Gemeint ist hier die Niere, die der eigene Spender des Patienten, auf den die Kante zeigt, zur Verfügung stellt.

Schauen wir zunächst, was TopTradingCycle in unserem Beispiel macht:

$p_1 : n_3, n_4, n_2, n_1, n_5$
$p_2 : n_4, n_1, n_2, n_3, n_5$
$p_3 : n_4, n_1, n_3, n_2, n_5$
$p_4 : n_5, n_2, n_1, n_4, n_3$
$p_5 : n_2, n_1, n_5, n_4, n_3$

$H(\{p_1, \ldots, p_5\})$: $H(\{p_1, p_3\})$: $H(\emptyset)$:



TopTradingCycle würde somit korrekterweise die Zuweisung $(n_3, n_4, n_1, n_5, n_2)$ ausgeben.
Wir zeigen nun, dass der Algorithmus TopTradingCycle immer mit einer stabilen Zuweisung terminiert.

Korrektheit des TopTradingCycle Algorithmus: Nach der Konstruktion des Top-Präferenz-Graphen wird getestet, ob ein Kreis vorhanden ist und falls dies der Fall ist, dann wird entlang dieses Kreises getauscht und die zugehörigen Knoten aus X und damit indirekt auch die betroffenen Nieren gelöscht. So lange es so ein Kreis gibt, wird also mindestens ein Element aus X entfernt und somit kann die while-Schleife nur höchstens n mal durchlaufen bevor der Algorithmus terminiert.
Wir zeigen zunächst, dass in jedem Schleifendurchlauf tatsächlich ein gerichteter Kreis gefunden wird. Der Grund hierfür ist die spezielle Struktur des Top-Präferenz-Graphen: Jeder Knoten hat genau eine ausgehende Kante.

Lemma 1. *In jedem gerichteten Graphen G mit $k > 0$ Knoten und genau einer ausgehenden Kante pro Knoten gibt es einen gerichteten Kreis.*

Beweis. Wir zeigen die Aussage per Widerspruch. Angenommen in G gibt es keinen gerichteten Kreis. Wir wählen einen beliebigen Knoten v_1 aus G (mindestens einen gibt es, da $k > 0$) und folgen der einzigen ausgehenden Kante zu Knoten v_2. Da G keinen Kreis hat, gilt $v_2 \neq v_1$. Wir folgen der ausgehenden Kante von v_2 zu v_3. Es gilt $v_3 \neq v_1$ und $v_3 \neq v_1$, sonst gäbe es einen gerichteten Kreis. Wie wiederholen dies, bis wir schließlich beim k-ten (und letzten) Knoten v_k angekommen sind. Für v_k gilt, dass $v_k \neq v_i$ für $1 \leq i \leq k$, da es sonst einen Kreis gäbe. Da jeder Knoten genau eine ausgehende Kante hat, muss auch v_k eine ausgehende Kante haben. Deren Endpunkt muss aber ein Knoten sein, den wir bereits besucht haben und somit gibt es einen gerichteten Kreis! Widerspruch. □

Das Lemma sagt, dass es mindestens einen Kreis gibt. Es ist leicht zu sehen, dass auch viele davon geben kann: Jede Kante könnte eine Schleife sein, d.h. der Graph könnte k viele gerichtete Kreise haben.
Da im Top-Präferenz-Graph immer jeder Knoten genau eine ausgehende Kante hat, wird somit in jedem Schleifendurchlauf ein gerichteter Kreis gefunden und den zugehörigen Patienten eine Niere zugewiesen (durch Tausch entgegen der Kreisrichtung). Danach werden diese Patienten aus X entfernt und der neue modifizierte Top-Präferenz-Graph betrachtet. Somit erhält im Laufe des Algorithmus jeder Patient genau eine Niere und somit gibt der Algorithmus eine Zuweisung aus. Wir müssen nun noch zeigen, dass dies tatsächlich eine stabile Zuweisung ist. Die Idee hierfür haben wir in Beispiel 4 bereits gesehen. Wir müssen nur zeigen, dass wir gerichtete Kreise im Top-Präferenz "richtig" behandeln, d.h. dass in jeder stabilen Zuweisung die Nieren in einem Top-Präferenz-Kreis genau so vergeben sein müssen, wie dies der Algorithmus tut. Wir zeigen sogar noch etwas Stärkeres:

Theorem 1. *Für jede Anfangszuweisung existiert genau eine stabile Zuweisung und dies ist genau die, die der* TopTradingCycle *Algorithmus berechnet.*

Beweis. Wir zeigen zunächst, dass wenn es überhaupt eine stabile Zuweisung gibt, dann muss es die sein, die der TopTradingCycle Algorithmus findet. Danach zeigen wir, dass die gefundene Zuweisung stabil ist.
In jedem Schleifendurchlauf wählt der Algorithmus einen gerichteten Kreis im Top-Präferenz-Graph aus. Sei C_i die Menge der Patienten, die im i-ten Schleifendurchlauf am ausgewählten Kreis beteiligt sind.
Wir zeigen per Induktion über i, dass die Koalition C_i die Zuweisung blockieren würde, falls sie eine andere Zuweisung als die Zuweisung aus dem Algorithmus erhalten würden.
Induktionsanfang, $i = 1$**:** Für jeden Patienten in C_1 gilt, dass die zugewiesene Niere dessen bestkompatible Niere ist. Somit würde C_1 jede andere Zuweisung blockieren, da in dem Fall mindestens ein Patient aus C_1 eine schlechtere Niere bekommen würde.
Induktionsschritt, $i \to i+1$ **für** $i \geq 1$**:** Angenommen die Aussage gilt für alle Patienten in den Kreisen $C_1, C_2, \ldots, C_i$. Wir zeigen nun, dass dann auch alle Patienten im Kreis C_{i+1} jede Zuweisung, die von der Zuweisung des TopTradingCycle Algorithmus abweicht, blockieren würden. Nach Konstruktion des Top-Präferenz-Graphen gilt, dass jeder Patient aus C_{i+1} im Graph eine Kante zu seiner/ihrer bestmöglichen Niere unter den noch verfügbaren Nieren hat. Nach Induktionsvoraussetzung gibt es keine stabile Zuweisung, in der ein Patient aus C_{i+1} eine Niere bekommen kann, welche bereits in einer früheren Runde zugewiesen wurde. Falls mindestens ein Patient aus C_{i+1} eine andere Niere zugewiesen bekommt, dann muss dies somit eine für diesen Patienten schlechtere Niere sein

und somit würde die Koalition C_{i+1} die Zuweisung blockieren.

Jetzt zeigen wir noch, dass die gefundene Zuweisung tatsächlich stabil ist. Wieder erfolgt dies per Induktion über i, indem wir zeigen, dass sich Patienten aus C_i niemals an Nierentauschen innerhalb von blockierenden Koalitionen beteiligen würden. Da in jeder blockierenden Koalition mindestens ein Tausch erfolgt, genügt dies um die Stabilität der Zuweisung zu beweisen.
Induktionsanfang, $i = 1$: Die Patienten in C_1 haben ihre bestmögliche Niere bekommen und somit würde kein Patient aus C_1 einem Tausch zustimmen.
Induktionsschritt, $i \to i + 1$ für $i \geq 1$: Patienten aus C_{i+1} würden sich nur an einem Tausch beteiligen, wenn sie dadurch eine bessere Niere erhalten könnten. Nach Konstruktion des Top-Präferenz-Graphen erhalten sie im TopTradingCycle Algorithmus die für sie beste Niere, welche nicht bereits in früheren Runden vergeben wurde. Folglich bräuchte ein Patient aus C_{i+1} einen Tauschpartner aus C_j mit $1 \leq j \leq i$, um sich zu verbessern. Nach Induktionsvoraussetzung gibt es aber keine tauschwilligen Patienten in $C_1, \ldots, C_j$. □

Bemerkung 2. *Sollten im Top-Präferenz-Graphen mehrere Kreise vorkommen, dann bleiben all diese Kreise garantiert erhalten, bis sie irgendwann vom TopTradingCycle Algorithmus ausgewählt werden. Es ist also tatsächlich egal, in welcher Reihenfolge diese Kreise abgearbeitet werden.*

Wir haben somit bewiesen, dass es für jede Anfangszuweisung überhaupt nur eine stabile Zuweisung gibt und dass unser Algorithmus genau diese findet. Das bedeutet auch, dass jeder andere korrekte Algorithmus genau die Zuweisung aus dem TopTradingCycle Algorithmus finden muss!

Analyse des TopTradingCycle Algorithmus: Wir betrachten wieder den Initialisierungsaufwand und die Kosten eines Schleifendurchlaufs. Wir haben bereits festgestellt, dass die Schleife höchstens n mal durchlaufen wird, da jedes mal mindestens ein Patient seine Niere endgültig zugewiesen bekommt.
Initialisierungsaufwand: Der Fakt, dass im anfänglichen Top-Präferenz-Graph mit n Knoten nur genau n Kanten vorhanden sind, liefert uns eine obere Schranke von $\mathcal{O}(n)$ für die Initialisierung. Insbesondere müssen wir bei allen Patienten nur den ersten Listeneintrag anschauen, um den Graph zu konstruieren.
Kosten eines Schleifendurchlaufs:

- Lemma 1 garantiert, dass immer ein gerichteter Kreis vorhanden ist, wir müssen diesen nur schnell finden. Interessanterweise liefert hier der Beweis von Lemma 1 einen einfachen Algorithmus: Starte bei einem beliebigen Knoten und folge den ausgehenden Kanten so lange, bis ein Knoten ein zweites mal besucht wird. Wir müssen uns dazu nur merken, welchen Knoten wir bereits gesehen haben und dies ist leicht mit einem weiteren Array möglich. Wir können somit in $\Theta(n)$ Zeit und mit $\Theta(n)$ zusätzlichem Speicher einen gerichteten Kreis finden.

- Das Zuweisen der Nieren geht leicht in $\mathcal{O}(n)$, da wir nur entlang des Kreises tauschen müssen und somit immer wissen, wo genau wir nachschauen müssen.

- Für die Aktualisierung der Menge X benötigen wir eine geeignete Datenstruktur, um Mengen zu repräsentieren. Wir werden später verschiedene Datenstrukturen für sogenannte *Wörterbücher*[11] behandeln. Hier benötigen wir allerdings nicht den vollen Umfang von Wörterbüchern, insbesondere weil wir im Laufe des Algorithmus nur Elemente aus X gelöscht werden. Wir können uns X somit leicht als Bitarray der Länge n speichern, wobei ein Bit i genau dann auf 1 gesetzt ist, falls $p_i \in X$. Damit können wir die Mengendifferenz in $\mathcal{O}(n)$ bilden, indem wir die entsprechenden Einträge auf 0 setzen.

- Das Aktualisieren des Top-Präferenz-Graphen ist etwas schwieriger. Das Problem ist, dass wir für jeden verbliebenen Patienten in X die bestmögliche noch verfügbare Niere finden müssen. Dazu speichern wir uns auch für die Nieren ein Bitarray, um schnell nachzusehen, ob eine spezielle Niere noch verfügbar ist. Für jeden Patienten können wir dann mit einem Pointer die Präferenzliste von vorn nach hinten durchlaufen und jedes mal testen, ob die aktuelle Niere noch verfügar ist. Wir stoppen beim ersten Treffer. Da in einem Schleifendurchlauf $\Omega(n)$ viele Patienten eine Niere bekommen können, bedeutet dies, dass wir im worst case auch für jeden Patienten $\Omega(n)$ weit in seiner/ihrer Präferenzliste laufen müssen. Eine Aktualisierung des Top-Präferenz-Graphen kann somit im worst-case $\Theta(n^2)$ kosten.

Die obige Analyse zeigt, dass ein Schleifendurchlauf im schlimmsten Fall $\mathcal{O}(n^2)$ kosten kann.
Gesamtkosten: Wir haben maximal n Schleifendurchläufe mit Kosten von $\mathcal{O}(n^2)$ pro Durchlauf. Dies ergibt eine obere Schranke von $\mathcal{O}(n^3)$.

Gesamtkosten TopTradingCycle Algorithmus – amortisierte Analyse: Bei unserer bisherigen Analyse haben wir angenommen, dass jeder Schleifendurchlauf im worst-case $\mathcal{O}(n^2)$ kostet. Das stimmt für einen einzelnen Schleifendurchlauf, aber dies kann nicht für jeden der n Schleifendurchläufe passieren. Für jeden Patienten suchen wir die Präferenzliste von vorn nach hinten ab. Wir gehen dabei niemals zurück und somit fallen pro Patienten insgesamt nur n dieser Suchschritte verteilt über alle Schleifendurchläufe an. Über alle Patienten summiert sind das insgesamt über alle Schleifendurchläufe nur $\mathcal{O}(n^2)$ Schritte. Alle anderen Aktionen innerhalb der Schleife haben Kosten in $\mathcal{O}(n)$ und somit fallen auch für diese über alle Durchläufe summiert nur Kosten in $\mathcal{O}(n^2)$ an. Diese genauere Analyse, die immer noch eine worst-case Garantie gibt, zeigt, dass die Gesamtkosten von TopTradingCycle sogar nur in $\mathcal{O}(n^2)$ sind!

Wir haben hier nur die Analysetechnik geändert. Statt herkömmlicher worst-case Analyse, bei der für jede Operation der schlimmste Fall angenommen wird, haben wir die worst-case Kosten *einer Folge von Operationen*, nämlich vielen Aktualisierungen von $H(X)$ nacheinander, betrachtet. Diese Analyseart nennt man *amortisierte Analyse* und sie ist oft genauer als die Analyse der einzelnen Operationen/Schritte eines Algorithmus.

Unsere neue Analyse ist sogar so genau, dass wir die Gesamtkosten damit sogar bestmöglich abgeschätzt haben. Dies gilt, weil eine obere Schranke von $\mathcal{O}(n^2)$ optimal ist, da

[11] Ein Wörterbuch ist jede Datenstruktur, mit der man Daten einfügen, suchen und löschen kann.

die Eingabe für das Kidney Exchange Problem bereits $\Theta(n^2)$ groß ist und jeder korrekte Algorithmus wenigstens die Eingabe komplett lesen muss[12]. D.h. unsere ermittelte obere Schranke für die Gesamtkosten stimmt mit der trivialen unteren Schranke von $\Omega(n^2)$ überein. Die Kosten von TopTradingCycle sind somit in $\Theta(n^2)$.[13]

Tuning des TopTradingCycle Algorithmus

Wir haben zwar eben argumentiert, dass eine Laufzeit in $\mathcal{O}(n^2)$ bestmöglich ist, doch wir können trotzdem noch die Frage stellen, ob wir den Algorithmus noch verbessern bzw. vereinfachen können.
Eine wichtige Komponente des TopTradingCycle Algorithmus ist das Verwalten des Top-Präferenz-Graphen $H(X)$ und das Finden eines gerichteten Kreises darin. Wir oben beschrieben, können wir einen solchen Kreis leicht in $\mathcal{O}(n)$ Zeit mit $\mathcal{O}(n)$ Zusatzspeicher (für das Markieren der bereits besuchten Knoten) finden. An dieser Stelle können wir optimieren! Wir werden nun zeigen, dass wir den Graphen $H(X)$ eigentlich gar nicht brauchen und dass wir den gerichteten Kreis mit nur $\mathcal{O}(1)$ Zusatzspeicher finden können.

Der TopTradingCycle Algorithmus ohne Top-Präferenz-Graph: Der Top-Präferenz-Graph hat eine sehr spezielle Struktur: Jeder Knoten hat genau eine ausgehende Kante. Für eine Kante (p_i, p_j) in $H(X)$ sagen wir im Folgenden, dass p_j der *Favorit von* p_i *innerhalb von* X ist, da unter allen Patienten in X der Patient p_i seine Spenderniere am liebsten mit p_j tauschen würde. Der Top-Präferenz-Graph kodiert somit nur die totale[14] Funktion

$$f : \{1, \ldots, n\} \to \{1, \ldots, n\},$$

wobei $f(i) = j$ bedeutet, dass p_j der Favorit von p_i ist. Diese Funktion können wir leicht als Array F repräsentieren, wobei der i-te Eintrag von F den Funktionswert von i beinhaltet, d.h. $F[i] = f(i)$. Somit müssen wir keine Adjazenzliste für die Repräsentation des Graphen $H(X)$ anlegen und verwalten.

Finden eines gerichteten Kreises: Unser obiger Algorithmus zum Finden eines gerichteten Kreises startet bei einem beliebigen Knoten p_i in $H(X)$ und folgt dann so lange den ausgehenden Kanten, bis ein Knoten ein zweites Mal besucht wird. Ein solcher Suchlauf entspricht somit der folgenden Sequenz

$$i, f(i), f(f(i)), f(f(f(i))), \ldots.$$

Wir schreiben die Sequenz etwas anders, sei α_0 der erste Wert der Sequenz und für alle $j > 0$ sei $\alpha_j = f(\alpha_{j-1})$. D.h. die Sequenz lautet

$$\alpha_0, \alpha_1 = f(\alpha_0), \alpha_2 = f(\alpha_1), \alpha_3 = f(\alpha_2), \ldots.$$

[12]Dieses Argument wird oft *triviale untere Schranke* genannt und für viele Probleme kann man auch formal beweisen, dass es keinen korrekten Algorithmus geben kann, der nicht die komplette Eingabe liest. Für das Kidney Exchange Problem gilt dies vermutlich auch - wir gehen hier aber nicht weiter darauf ein.

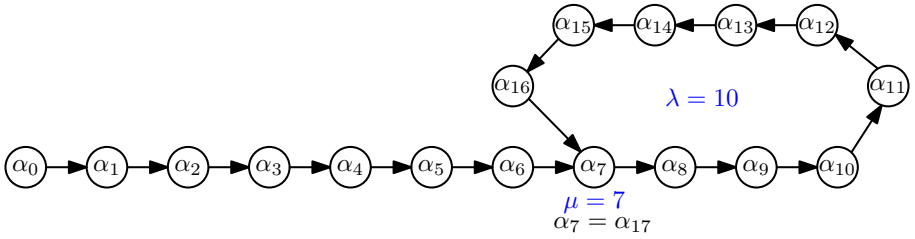
[13]Wir werden die Definitionen von $\mathcal{O}, \Omega$ und Θ demnächst wiederholen.

[14]Bei einer totalen Funktion hat jedes Argument einen Funktionswert, d.h. es gibt keine undefinierten Stellen.

In dieser Sequenz wollen wir eigentlich nur herausfinden, wann zwei Werte der Sequenz gleich sind. Sobald dies passiert haben wir einen gerichteten Kreis gefunden. Genauer gesagt, suchen wir den ersten Index, ab dem dieser Kreis betreten wird, d.h. wir suchen den kleinsten Index μ, so dass α_μ unendlich oft in der Sequenz

$$\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \dots$$

vorkommt.

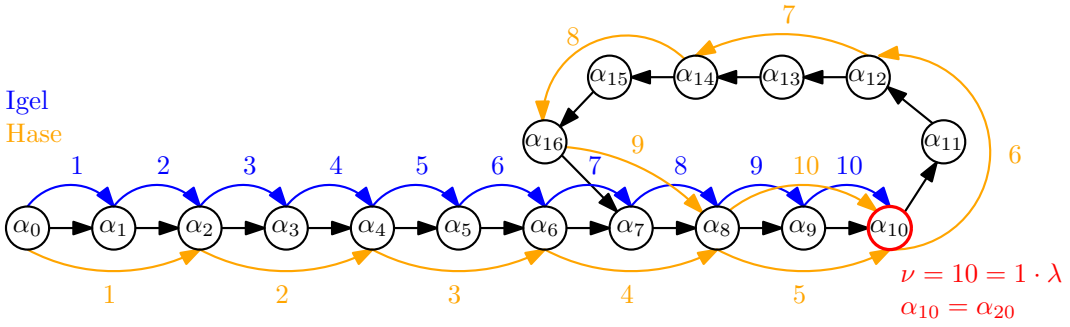


Um den gerichteten Kreis zu extrahieren, benötigen wir noch die Kreislänge λ, d.h. die kleinste natürliche Zahl λ so dass $\alpha_\mu = \alpha_{\mu+\lambda}$ gilt. Wir werden nun zeigen, dass wir für jede Funktion

$$f : \{1, \dots, n\} \to \{1, \dots, n\}$$

und jeden beliebigen Startwert α_0 die Indizes μ und λ mit nur konstantem Zusatzspeicher finden können!

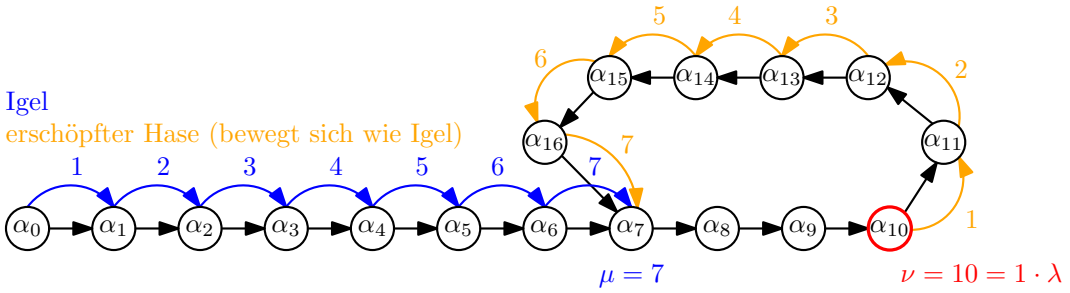
Floyd's Hase-und-Igel Algorithmus: Der Hase-und-Igel Algorithmus[15], genannt nach Aesops Fabel, basiert auf der Idee, dass zwei Läufer - ein Igel und ein Hase - einen Wettlauf auf der Sequenz machen und beide unterschiedlich schnell laufen. Der Hase läuft doppelt so schnell wie der Igel. Beide starten zur selben Zeit am Anfang der Sequenz bei α_0 und somit gilt, dass falls irgendwann der Hase und der Igel auf dem selben Feld stehen, d.h. falls beide Sequenzwerte übereinstimmen, dann muss der Hase den Igel überrundet haben und somit muss die Rennstrecke (d.h. die Sequenz $\alpha_0, \alpha_1, \dots$) einen Kreis enthalten.
Genauer, sobald Hase und Igel sich treffen, muss das auf einer Postion ν passieren, die ein Vielfaches der Kreislänge λ ist, d.h. $\nu = k \cdot \lambda$ für ein $k > 0$.



Der Grund hierfür ist, dass der Hase doppelt so schnell wie der Igel läuft. Falls sich Hase und Igel an Position ν zum ersten Mal wiedertreffen, dann gilt $\alpha_\nu = \alpha_{2\nu}$ für irgendein

[15]Es unklar, ob der Algorithmus wirklich von Robert W. Floyd entwickelt wurde - er wird ihm zugeschrieben, da Floyd sehr früh Algorithmen für ähnliche Probleme publiziert hat.

$\nu \geq \mu$, d.h. dies muss innerhalb des Kreises passieren. Außerdem gilt dann $\alpha_\nu = \alpha_{\nu+k\cdot\lambda}$ und somit folgt, dass $\nu = k \cdot \lambda \geq \mu$.
Sobald ν gefunden ist, kann μ ermittelt werden. Dafür wird ein zweites Rennen veranstaltet, in dem der Igel bei Position α_0 startet und der Hase bei Position ν. Allerdings ist der Hase jetzt erschöpft und bewegt sich deshalb nur so schnell wie ein Igel. Interessanterweise treffen sich beide dann genau bei α_μ.[16]



Mit der Kenntnis von μ können wir dann leicht λ bestimmen. Dazu wartet der Igel einfach an Position α_μ und der Hase macht so lange Igelschritte, bis er den Igel wiedertrifft, d.h. dann gilt $\alpha_\mu = \alpha_{\mu+\lambda}$. Der gerichtete Kreis lautet dann

$$\alpha_\mu, \alpha_{\mu+1}, \alpha_{\mu+2}, \ldots, \alpha_{\mu+\lambda-1}, \alpha_{\mu+\lambda}.$$

Hier der Algorithmus:

HaseUndIgel(f,α_0)

Input: Nachfolgerfunktion f, Anfangswert α_0
Output: μ und λ

```
1: Igel ← f(α0); Hase ← f(f(α0));
2: while Hase ≠ Igel do
3:     Igel ← f(Igel); Hase ← f(f(Hase));
4: end while
5: μ ← 0; Igel ← α0
6: while Hase ≠ Igel do
7:     Igel ← f(Igel); Hase ← f(Hase); μ ← μ + 1
8: end while
9: λ ← 1; Hase ← f(Igel)
10: while Hase ≠ Igel do
11:     Hase ← f(Hase) ; λ ← λ + 1
12: end while
13: return  μ, λ
```

Der Algorithmus benötigt nur konstant viel Zusatzspeicher, da für den Hasen und den Igel nur zwei Pointer verwaltet bzw. zwei Indizes gespeichert werden müssen und auch sonst nur konstant viel Speicher, z.B. für das Auswerten von $f(f(\cdot))$, benötigt wird.
Wir zeigen in der Übung, dass der Algorithmus HaseUndIgel Gesamtkosten in $\mathcal{O}(\mu + \lambda)$ hat. Da $\mu \leq n$ und $\lambda \leq n$ gilt, ist dies in $\mathcal{O}(n)$.

[16] Wir beweisen die Aussage in der Übung!

1.3 Warm-Up: Stable Matching

Wir wenden uns einem weiteren Problem zu, welches viele wichtige Anwendungen in der Praxis hat. Eine Beispielanwendung ist folgendes Szenario: Es gibt n Krankenhäuser, die jeweils einen Praktikumsplatz zu vergeben haben und es gibt n Medizinstudenten, die gern ein Praktikum in einem der Krankenhäuser machen wollen. Wie die Studierenden an die Krankenhäuser zugewiesen werden, ist allerdings nicht egal, denn sowohl die Krankenhäuser als auch die Studierenden haben Präferenzen, welche Studierende bzw. Krankenhäuser sie bevorzugen. Wir wollen nun die Studierenden so an Krankenhäuser zuweisen, dass folgendes gilt:

1. Jedes Krankenhaus soll genau einen Studierenden zugewiesen bekommen und jeder Studierende soll einen Praktikumsplatz erhalten.

2. Alle sollen mit dieser Zuweisung zufrieden sein. Zufriedenheit definieren wir dabei wie folgt: Es soll kein Paar von Krankenhaus und Studierenden geben, die einander gegenüber ihrer Zuweisung bevorzugen. (Sonst würden die beiden sich unter der Hand einigen und damit den Zuweisungsmechanismus sabotieren.)

Unser Ziel ist es, einen Algorithmus zu finden, der uns eine Zuweisung berechnet, welche die obigen Eigenschaften erfüllt. Allerdings ist von vornherein überhaupt nicht klar, ob es so eine Zuweisung überhaupt gibt.
Das Problem sieht dem Kidney Exchange Problem sehr ähnlich, da wir auch wieder spezielle Paare suchen, allerdings unterscheidet es sich in einigen Punkten. Z.B.:

- Beim Kidney Exchange Problem haben wir angenommen, dass es eine Anfangszuweisung gibt und somit, dass Patienten schon garaniert eine spezielle Spenderniere sicher haben. Beim Stable Matching wird das nicht der Fall sein.

- Beim Stable Matching Problem haben beide Beteiligten eines Paares Präferenzen über die jeweils andere Seite. Beim Kidney Exchange hat nur eine Seite Präferenzen (den Nieren ist es egal, in wem sie transplantiert werden).

- Die Bedingung für Stabilität wird sich unterscheiden. Bei Kidney Exchange haben wir Koalitionen beliebiger Größe betrachtet, beim Stable Matching wollen wir "nur" verhindern, dass spezielle Koalitionen (bestehend aus einem Krankenhaus und einem Studierenden) die Zuweisung blockieren.

Zunächst formalisieren wir das Problem, d.h. wir definieren, was genau eine Zuweisung ist und geben eine Bedingung an, wann Krankenhäuser bzw. Studierende mit einer Zuweisung einverstanden sind.

1.3.1 Definitionen

Zunächst benötigen wir den Begriff eines *Matchings*. Intuitiv ist ein Matching eine eindeutige Zuweisung von Elementen zu Paaren von Elementen.

Definition 6. *(Matching, perfektes Matching, bipartites Matching) Sei X eine Menge. Wir bezeichnen $\{x, y\}$ als ein Paar, falls $x, y \in X$ und $x \neq y$ gilt. Ein* Matching über

X, kurz $\mathcal{M}(X)$, ist eine Menge von Paaren, so dass jedes Element von X in höchstens einem Paar vorkommt.
Falls jedes Element von X in genau einem *Paar vorkommt, dann bezeichnen wir $\mathcal{M}(X)$ als* perfektes Matching.
Seien nun X und Y disjunkte Mengen. Wir nennen $\{x, y\}$, mit $x \in X$ und $y \in Y$, ein bipartites Paar. *Ein* bipartites Matching über X, Y, *kurz $\mathcal{M}(X, Y)$, ist eine Menge von bipartiten Paaren, so dass jedes Element von $X \cup Y$ in höchstens einem bipartiten Paar vorkommt. Falls jedes Element von X und jedes Element von Y in genau einem Paar vorkommt, dann ist $\mathcal{M}(X, Y)$ ein* perfektes bipartites Matching.

Beispiel 5. *Wählen wir beispielsweise $X = \{m_1, m_2, m_3\}$ und $Y = \{f_1, f_2, f_3\}$, dann wäre die Menge*

$$\{\{m_1, f_1\}, \{m_2, f_3\}\}$$

ein bipartites Matching über X und Y, die Menge

$$\{\{m_1, f_1\}, \{m_2, f_1\}\}$$

hingegen nicht, da f_1 in zwei Paaren vorkommt. Die Menge

$$\{\{m_1, f_1\}, \{m_2, f_3\}, \{m_3, f_2\}\}$$

ist ein perfektes bipartites Matching. ◁

Wir nehmen an, dass den Elementen aus X und Y nicht egal ist, mit wem sie ein Paar bilden. Dazu führen wir, genau wie beim Kidney Exchange Problem, Präferenzlisten ein. Im Weiteren nehmen wir an, dass $|X| = |Y| = n$ gilt.
Um es etwas anschaulicher zu machen, nennen wir

$$X = \{m_1, \ldots, m_n\} \text{ die Menge der Männer}$$

und

$$Y = \{f_1, \ldots, f_n\} \text{ die Menge der Frauen.}$$

Jeder Mann $m_i \in X$ hat eine Präferenzliste L_{m_i} über alle Frauen und jede Frau $f_j \in Y$ hat eine Präferenzliste L_{f_j} über alle Männer. Die Listen L_{m_i} bzw. L_{f_j} sind Permutationen von $f_1, \ldots, f_n$ bzw. $m_1, \ldots, m_n$. Wir bezeichnen mit $L_{m_i}[k]$ das k-te Element in der Liste L_{m_i}. Analog für L_{f_j}. Das erste Element auf m_i's Präferenzliste, d.h. $L_{m_i}[1]$, ist Mann m_i's Top-Präferenz, d.h. Mann m_i würde am liebsten mit der Frau $L_{m_i}[1]$ ein Paar bilden, am zweitliebsten wäre Mann m_i die Frau $L_{m_i}[2]$, usw. Wir sagen dass Mann m_i die Frau f_k über Frau f_l präferiert, falls f_k in L_{m_i} vor f_l vorkommt. Analog gilt dies auch für die Präferenzen der Frauen über die Männer.
Mit diesen Präferenzlisten kann man nun ein bipartites Matching $\mathcal{M}(X, Y)$ daraufhin untersuchen, ob alle Männer und Frauen mit ihrem zugewiesenen Partner zufrieden sind. Dazu definieren wir Folgendes:

Definition 7. *(instabiles Paar[17]) Sei $\{a, b\}, \{c, d\} \in \mathcal{M}(X, Y)$, mit $a, c \in X$ und $b, d \in Y$. Das Paar $\{a, d\}$ heißt* instabiles Paar, *falls Mann a die Frau d über die Frau b präferiert und falls Frau d den Mann a über den Mann c präferiert.*

[17] In der Literatur wird auch oft der Name "blockierendes Paar" verwendet.

Intuitiv: Wenn $\{a, d\}$ ein instabiles Paar bilden, dann wollen sich beide von ihrem aktuellen Partner trennen und gemeinsam durchbrennen. Wir interessieren uns für bipartite Matchings, wo genau dies nicht passieren kann.

Definition 8. *(Stabiles Matching) Ein perfektes bipartites Matching $\mathcal{M}(X, Y)$ heißt* stabil*, falls kein instabiles Paar existiert.*

Die Einschränkung auf perfekte Matchings kann leicht beseitigt werden, aber im Folgenden ist dies nicht nötig.

Beispiel 6. *Wir betrachten wieder unser Beispiel von oben, diesmal mit Präferenzlisten: Sei $X = \{m_1, m_2, m_3\}$ und $Y = \{f_1, f_2, f_3\}$ mit den folgenden Präferenzen:*

$$L_{m_1} = f_2, f_1, f_3 \qquad L_{f_1} = m_1, m_2, m_3$$
$$L_{m_2} = f_2, f_3, f_1 \qquad L_{f_2} = m_1, m_2, m_3$$
$$L_{m_3} = f_3, f_1, f_2 \qquad L_{f_3} = m_3, m_1, m_2$$

Ist das Matching $\{\{m_1, f_1\}, \{m_2, f_3\}, \{m_3, f_2\}\}$ stabil?
Nein, da z.B. $\{m_1, f_2\}$ ein instabiles Paar bildet. Mann m_1 hat Frau f_1 zugewiesen bekommen, bevorzugt aber f_2 über f_1. Frau f_2 hat Mann m_3 zugewiesen bekommen, bevorzugt aber m_1 über m_3. D.h. Mann m_1 und Frau f_2 wären mit der Zuweisung unzufrieden und würden miteinander durchbrennen. ◁

Es stellt sich die Frage, ob überhaupt ein stabiles Matching existiert und falls ja, wie man ein solches finden kann.
Wir müssen also das folgende Problem lösen:

Stable Matching Problem:

> **Gegeben:** n Männer $X = \{m_1, \ldots, m_n\}$ und n Frauen $Y = \{f_1, \ldots, f_n\}$, jeweils mit Präferenzlisten.
>
> **Aufgabe:** Finde ein perfektes bipartites stabiles Matching über X und Y, falls ein solches existiert.

1.3.2 Erste Lösungsansätze

Brute Force Die vermutlich einfachste Idee ist alle perfekten Matchings durchzuprobieren und für jedes Matching zu prüfen, ob es ein instabiles Paar gibt.
Dabei stellt sich die Frage, wie viele perfekte Matchings es über den Mengen X und Y überhaupt gibt?
Leider sind das sehr viele: Für Mann m_1 gibt es n mögliche Partnerinnen. Wenn m_1 eine davon zugewiesen bekommen hat, dann bleiben für m_2 noch $n-1$ mögliche Partnerinnen übrig. Wenn m_2 eine Partnerin bekommen hat, dann bleiben für m_3 noch $n-2$ viele übrig usw.
Folglich gibt es $n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1 = n!$ viele perfekte Matchings. Selbst wenn wir annehmen, dass der Test, ob ein gegebenes Matching stabil ist, sehr schnell geht[18]

[18] Genauer: Wenn wir nur konstanten Aufwand dafür benötigen, d.h. der Aufwand dafür hängt nicht von n ab.

(was übrigens nicht stimmt), dann haben wir einen Algorithmus dessen Gesamtkosten sich etwa wie die Funktion $n!$ verhält[19].

Greedy Wir betrachten drei mögliche Ansätze für einen Greedy-Algorithmus, d.h. einen Algorithmus, der nach und nach immer das Erstbeste tut:

1. Suche nach Paar $\{m_i, f_j\}$, so dass f_j die Top-Präferenz von m_i und m_i die Top-Präferenz von f_j ist. Bilde das Paar und streiche m_i und f_j von allen Listen. Fahre iterativ fort.

   Im obigen Beispiel entsteht damit sogar ein stabiles Matching, nämlich

   $$\{\{m_1, f_2\}, \{m_2, f_1\}, \{m_3, f_3\}\}.$$

   Falls der Algorithmus terminiert, dann wird garantiert ein stabiles Matching gefunden.[20]

   Problem: Was passiert, falls es keine Paare mit gegenseitiger Top-Präferenz gibt?

2. Beginne mit einem beliebigen perfekten bipartiten Matching. Falls es ein instabiles Paar gibt, dann Bilde dieses und das Paar aus den verlassenen Partnern und fahre iterativ fort.

   Problem: Dieser Algorithmus kann endlos laufen![21]

3. Gehe Männer der Reihe nach durch und weise jedem Mann seine noch mögliche beste Partnerin zu.

   Im obigen Beispiel entsteht das Matching $\{\{m_1, f_2\}, \{m_2, f_3\}, \{m_3, f_1\}\}$. Es ist nicht stabil, da m_3 die Frau f_3 über f_1 präferiert und f_3 den Mann m_3 über m_2 präferiert.

Alle obigen Ansätze sind somit problematisch. Ansatz 1 versucht sowohl Männern als auch Frauen die bestmögliche Zuweisung zu geben, d.h. er ist zu beiden Parteien fair, allerdings ist dies manchmal unmöglich. Ansatz 3 versucht die Zuweisung nur für die Männer in einer festgelegten Reihenfolge zu optimieren, was dann dazu führt dass Männer, die erst spät an die Reihe kommen, und evtl. alle Frauen mit ihrer Zuweisung unzufrieden sind[22]. Ansatz 2 klingt auf dem ersten Blick plausibel, allerdings gibt es Instanzen, bei denen dieses Verfahren nicht terminiert.

Ähnlich wie im TopTradingCycle Algorithmus: Wir testen noch einen Greedy-Algorithmus, der dem TopTradingCycle Algorithmus ähnelt. Wir könnten doch auch einfach den Top-Präferenz-Graph aufbauen und in diesem Graph nach Kreisen suchen. Da jeder Mann und jede Frau eine Top-Präferenz hat, muss es auch immer einen Kreis geben. Da der Graph bipartit ist (es gibt nur Kanten zwischen Männern und Frauen), haben diese Kreise immer gerade Länge.

[19] Wie bereits beim Travelling Salesman Problem erwähnt, wächst diese Funktion rasend schnell.

[20] Die Aussage kann leicht mit Hilfe der Definition der Stabilität bewiesen werden.

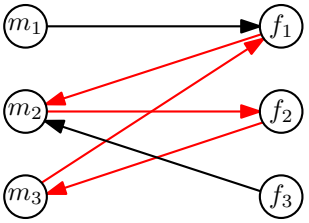
[21] Eine passende Instanz zu konstruieren, für die der Algorithmus endlos läuft, wird eine spannende Übungsaufgabe sein.

[22] Es wäre ja auch seltsam, wenn ein stabiles Matching entstehen würde, obwohl die Präferenzen der Frauen überhaupt keine Rolle bei der Zuweisung spielen.

Falls wir in diesem Algorithmus Kreise der Länge 2 antreffen, dann haben wir den obigen Greedy-Ansatz-1 wiederentdeckt. Interessant ist somit der Fall, wenn es keine Kreise der Länge 2 gibt. Wir müssen uns nun überlegen, wie wir mit diesen Kreisen umgehen. Hier ist ein Beispiel:

Beispiel 7. *Wir betrachten die folgende Instanz und den zugehörigen Top-Präferenz-Graph:*

$$L_{m_1} = f_1, f_2, f_3 \qquad L_{f_1} = m_2, m_1, m_3$$
$$L_{m_2} = f_2, f_3, f_1 \qquad L_{f_2} = m_3, m_1, m_2$$
$$L_{m_3} = f_1, f_3, f_2 \qquad L_{f_3} = m_2, m_1, m_3$$



Im Tauschgraph gibt es nur einen Kreis der Länge 4. *Wir haben zwei Optionen:*

1. *Wir weisen allen Männern im Tauschkreis ihre Top-Präferenz-Frau zu (d.h.* m_2 *bekommt* f_2 *und* m_3 *bekommt* f_1*), entfernen alle Beteiligten und iterieren. Damit erhalten wir folgendes Matching:* $\{\{m_1, f_3\}, \{m_2, f_2\}, \{m_3, f_1\}\}$.

   Das Matching ist nicht stabil, da $\{m_1, f_2\}$ *ein blockierendes Paar ist.*

2. *Wir weisen allen Frauen im Tauschkreis ihren Top-Präferenz-Mann zu (d.h.* f_1 *bekommt* m_2 *und* f_2 *bekommt* m_3*), entfernen alle Beteiligten und iterieren. Damit erhalten wir:* $\{\{m_1, f_3\}, \{m_2, f_1\}, \{m_3, f_2\}\}$.

   Das Matching ist nicht stabil, da $\{m_2, f_3\}$ *blockiert.*

Es ist somit vollkommen unklar, wie wir die Top-Präferenz-Kreise behandeln müssen. $\triangleleft$

Erkenntnis aus den ersten Versuchen: Wir brauchen einen Algorithmus, welcher die Präferenzen beider Parteien berücksichtigt, aber welcher im Zweifelsfall eine Partei bevorzugt, falls sonst keine Lösung möglich wäre.
Erstaunlicherweise gibt es einen solchen Algorithmus, d.h. dieser Algorithmus garantiert auch die Existenz eines stabilen perfekten Matchings für jede Instanz des Problems!

1.3.3 Der Gale-Shapley-Algorithmus

Die Lösung des Problems liefert ein etwas ausgefallenerer Greedy-Algorithmus. Die Hauptidee liegt darin, dass man beim Greedy-Algorithmus 2 von oben bereits "vergebene" Frauen nicht aus den Präferenzlisten streicht und dass die Frauen sich verbessern dürfen, wenn ein für sie besserer Mann anfragt.

Algorithmus Gale-Shapley Stable Matching

Input: Präferenzlisten der n Männer und n Frauen
Output: perfektes bipartites stabiles Matching

```
1: initialisiere alle Männer und Frauen mit „frei“
2: while es gibt freien Mann m do
3:    w ← beliebteste Frau von Mann m, die noch nicht von m gefragt wurde
4:    Mann m fragt bei Frau w an
5:    if w ist frei then
6:       bilde Paar {m, w}
7:    else
8:       if w bevorzugt m über ihren aktuellen Partner m' then
9:          bilde Paar {m, w}
10:         setze Mann m' auf „frei“
11:      end if
12:   end if
13:   markiere w auf Präferenzliste von m
14: end while
```

Achtung: Der obige Algorithmus ist in Zeile 2 nicht eindeutig. Man kann allerdings beweisen, dass es für die Terminierung und Korrektheit keine Rolle spielt, welcher freie Mann m gewählt wird. Es gilt sogar, dass das berechnete stabile Matching immer dasselbe ist![23]
Wir betrachten nun den Algorithmus etwas näher:

Beobachtungen zum Gale-Shapley-Algorithmus

1. Falls eine Frau jemals einen Partner bekommt, dann wird sie nie wieder frei.
2. Frauen können sich im Verlauf des Algorithmus nur verbessern.
3. Die Anzahl der gefundenen Paare bleibt gleich oder steigt in jedem Durchlauf der Schleife.
4. Männer fragen niemals die selbe Frau ein zweites Mal.
5. In jedem Durchlauf wird auf irgendeiner Männer-Präferenzliste eine weitere Markierung hinzugefügt.

Korrektheit und Terminierung des Gale-Shapley-Algorithmus

Aus Beobachtung 5 folgt, dass der Algorithmus immer terminiert. Anfangs sind alle Listeneinträge unmarkiert und in jedem Durchlauf kommt eine weitere Markierung hinzu. Da keine Einträge doppelt markiert werden und es nur endlich viele Einträge gibt, muss die while-Schleife irgendwann abbrechen.
Die Korrektheit des Algorithmus ist etwas schwieriger zu sehen. Wir zeigen zunächst, dass der Algorithmus immer ein perfektes bipartites Matching erzeugt.

[23] Wir untersuchen demnächst, welches stabile Matching gefunden wird und beweisen die Aussage, dass unabhängig von der Reihenfolge in Zeile 2 immer dasselbe stabile Matching gefunden wird.

Lemma 2. *Der Gale-Shapley-Algorithmus muss mit einem perfekten bipartiten Matching terminieren.*

Beweis. Angenommen dies ist nicht der Fall, dann muss eine Frau oder ein Mann nach Terminierung des Algorithmus frei sein. Falls ein Mann frei ist, dann wäre die `while`-Schleife nicht abgebrochen[24]. Falls eine Frau frei ist, dann muss auch noch ein Mann frei sein, denn es gibt genau n Männer und n Frauen und jeder kommt in höchstens einem Paar vor. □

Nun zeigen wir noch, dass das gefundene perfekte bipartite Matching tatsächlich ein stabiles perfektes bipartites Matching ist.

Theorem 3. *Das gefundene perfekte bipartite Matching des Gale-Shapley-Algorithmus ist stabil.*

Beweis. Angenommen das gefundene Matching ist nicht stabil, d.h. es gibt zwei Paare $\{x, y\}, \{u, v\} \in \mathcal{M}$ und es gilt, dass Mann x die Frau v über y präferiert und Frau v den Mann x über u präferiert. D.h. $\{x, v\}$ bilden ein instabiles Paar.
Da bei Mann x die Frau v weiter vorn als Frau y in der Präferenzliste steht, muss Mann x bei Frau v angefragt haben, bevor er bei Frau y angefragt hat[25]. Es gibt zwei Fälle:

1. Frau v hat Mann x bei seiner Anfrage abgelehnt. Es folgt, dass es zum Anfragezeitpunkt bereits ein Paar $\{x', v\}$ gab und dass Frau v den Mann x' über x präferiert. Beobachtung 2 besagt, dass sich Frauen im Laufe des Algorithmus nur verbessern können. Folglich präferiert Frau v den Mann u über x' und somit transitiv auch über x, oder $u = x'$. In beiden Fällen erhalten wir einen Widerspruch zur Aussage, dass Frau v den Mann x über u bevorzugt.

2. Frau v hat Mann x zunächst akzeptiert und später wieder verlassen. Aufgrund Beobachtung 2 folgt auch hier, dass Frau v von einem Mann x' angefragt wurde, den sie über x präferiert. Dies ist ein Widerspruch zur Annahme.

In beiden Fällen erhalten wir einen Widerspruch und somit kann es kein instabiles Paar im gefundenen perfekten bipartiten Matching geben. □

1.3.4 Welches stabile Matching wird gefunden?

Nun wollen wir noch etwas Zeit darauf verwenden, wirklich zu verstehen, welche Eigenschaften, dass vom Algorithmus gefundene stabile Matching hat. Dies ist wichtig, da der Algorithmus in hochsensiblen Anwendungsbereichen, wie z.B. der Organspende, eingesetzt wird. Wir sollten also besser sehr gut verstehen, welches Ergebnis der Algorithmus liefert.

[24] Es ist unmöglich, dass ein Mann frei ist, nachdem er bei allen Frauen angefragt hat. Dazu müssten alle n Frauen an einen jeweils für sie besseren Mann vergeben sein, aber es gibt nur $n-1$ andere Männer.
[25] Dies folgt aus Zeile 3 des Algorithmus.

Mehrere stabile Matchings Bisher haben wir nur bewiesen, dass der Gale-Shapley-Algorithmus immer mit einem perfekten bipartiten stabilen Matching terminiert. D.h. wir haben eigentlich "nur" gezeigt, dass stabile Matchings immer existieren, bzw. genauer, dass wir für jede Eingabe immer ein stabiles Matching konstruieren können. Wir klären nun zunächst, die folgende Frage:

Kann es für eine Instanz mehrere stabile Matchings geben?

Die Antwort lautet ja. Hier ist ein Beispiel: Wir wählen $X = \{m, m'\}$ und $Y = \{f, f'\}$ mit folgenden Präferenzen:

$$L_m = f, f' \qquad L_f = m', m$$
$$L_{m'} = f', f \qquad L_{f'} = m, m'$$

Wenn wir nun den Gale-Shapley-Algorithmus ausführen, dann erhalten wir das stabile Matching

$$M = \{\{m, f\}, \{m', f'\}\}.$$

Dieses Matching ist für die Männer optimal, jedoch schlimmstmöglich für die Frauen. Wir werden später beweisen, dass dies kein Zufall ist. Auch gilt, dass jede mögliche Reihenfolge der freien Männer in Zeile 2 des Algorithmus zum obigen stabilen Matching führt. Auch dies ist kein Zufall. Allerdings gibt es noch ein weiteres Matching, welches vom Gale-Shapley-Algorithmus garantiert nicht gefunden wird:

$$M' = \{\{m, f'\}, \{m', f\}\}.$$

Das Matching M' ist tatsächlich stabil, da keine der beiden Frauen in einem instabilen Paar vorkommen können, da beide ihre Top-Präferenz bekommen haben.
Es kann also stark unterschiedliche stabile Matchings geben. Eines, welches die Männer bevorzugt und die Frauen benachteiligt und eines, welches die Frauen bevorzugt und die Männer benachteiligt. In beiden Fällen ist das Ergebnis unfair. Wir werden nun beweisen, dass dies leider immer so ist.

Männeroptimale Matchings Wenn wir in den Pseudocode des Gale-Shapley-Algorithmus schauen, dann sehen wir, dass dieser in Zeile 2 unterspezifiziert ist. Dort wird nicht festgelegt, in welcher Reihenfolge die freien Männer zum Zuge kommen. Wir werden nun beweisen, dass die Reihenfolge in Zeile 2 vollkommen egal ist. Jede Reihenfolge der freien Männer wird zum bestmöglichen stabilen Matching für alle Männer führen! Dies klingt auf den ersten Blick vollkommen überraschend.
Um die Aussage zu zeigen, werden wir genau charakterisieren, welches stabile Matching vom einem speziellen Durchlauf des Algorithmus gefunden wird. Außerdem zeige wir, dass alle möglichen Durchläufe (d.h. Reihenfolgen der freien Männer) exakt dasselbe stabile Matching liefern.

Definition 9. *Eine Frau f heißt* zulässige Partnerin *für Mann m, falls es ein perfektes bipartites stabiles Matching M gibt, in dem $\{m, f\}$ als Paar enthalten ist. Frau f heißt* bestmögliche zulässige Partnerin *für Mann m, falls f eine zulässige Partnerin ist und es keine zulässige Partnerin f' gibt, die Mann m über f präferiert. Im Folgenden bezeichnet $best(m)$ die beste zulässige Partnerin von Mann m.*

Wir werden nun zeigen, dass der Gale-Shapley-Algorithmus immer das Matching

$$M^* = \{\{m_i, best(m_i)\} \mid 1 \leq i \leq n\}$$

findet. Von vornherein ist noch nicht einmal klar, dass M^* überhaupt ein perfektes bipartites Matching ist. Zudem ist es tatsächlich stabil.

Theorem 4. *Jeder Durchlauf des Gale-Shapley-Algorithmus erzeugt das Matching M^*.*

Beweis. Wir führen den Beweis per Widerspruch. Angenommen es gibt einen Durchlauf D des Gale-Shapley-Algorithmus, der ein Matching $M \neq M^*$ erzeugt. D.h. in M gibt es mindestens einen Mann m, der nicht mit seiner besten zulässigen Partnerin zusammen ist.
Im Algorithmus fragen die Männer jeweils die Frauen in absteigender Reihenfolge an. D.h. Männer fragen zuerst ihre Top-Präferenz, dann die zweitbeste Frau usw. Im Durchlauf D des Algorithmus muss also Mann m irgendwann zum ersten Mal von einer zulässigen Partnerin f abgelehnt oder verlassen worden sein. Aufgrund der Anfragereihenfolge muss dann $f = best(m)$ gelten.
Wir nehmen im weiteren an, dass Mann m derjenige Mann ist, dem dies im Durchlauf D zuerst passiert (es könnte ja mehrere Männer geben, die in M nicht mit ihrer besten zulässige Partnerin zusammen sind - wir fokussieren hier, auf den ersten Mann, dem im Laufe des Algorithmus dieses Schicksal trifft).
Es ist egal, ob Mann m von Frau f direkt abgeleht, oder erst später verlassen wird. In beiden Fällen können wir annehmen, dass Frau f bei Mann m' landet, den sie über Mann m bevorzugt.
Da Frau f eine zulässige Partnerin ist, muss es allerdings ein stabiles Matching M' geben, in dem das Paar $\{m, f\}$ vorkommt. Wir betrachten nun, welche Partnerin der Mann m' im stabilen Matching M' abbekommen hat. Da M' ein perfektes Matching ist, muss m' eine Partnerin $f' \neq f$ in M' haben.
Nach unserer Wahl von Mann m gilt, dass m der erste Mann ist, der im Durchlauf D des Algorithmus von einer zulässigen Partnerin abgelehnt oder verlassen wurde. D.h. Mann m' wurde im Durchlauf D des Algorithmus zum Zeitpunkt als er mit Frau f zusammenkommt noch nicht von einer zulässigen Frau abgelehnt oder verlassen. Da Mann m' aber in absteigender Reihenfolge bei den Frauen anfragt und Frau f' eine zulässige Partnerin von m' ist, muss gelten, dass Mann m' die Frau f über Frau f' bevorzugt. Außerdem gilt, dass Frau f den Mann m' über Mann m bevorzugt (da m von ihr abgelehnt oder verlassen wird und Frau f dann bei Mann m' landet und sich Frauen stets nur verbessern).
Das Paar $\{m', f\}$ ist allerdings nicht im stabilen Matching $\{M'\}$ enthalten, da dort ja die Paare $\{m, f\}$ und $\{m', f'\}$ vorkommen. Wir haben somit, dass Mann m' lieber Frau f anstatt Frau f' und Frau f lieber Mann m' anstatt Mann m hätte. Somit ist $\{m', f\}$ ein instabiles Paar und das Matching M' kann nicht stabil sein. Wir haben also einen Widerspruch und folglich kann das Matching M nicht existieren. □

Wir haben somit bewiesen, dass der Gale-Shapley-Algorithmus immer das bestmögliche Matching für die Männer findet. Nun kommen wir zur negativen Eigenschaft, dass dieses Matching leider immer das schlimmstmögliche für alle Frauen ist.

Theorem 5. *Im stabilen Matching M^* ist jede Frau mit dem schlimmstmöglichen zulässigen Partner zusammen.*

Hier ist der zulässige Partner für Frauen analog zur zulässigen Partnerin für Männer definiert.

Beweis. Angenommen es gibt ein Paar $\{m, f\}$ in M^*, so dass Mann m nicht der schlimmstmögliche zulässige Partner von Frau f ist. Folglich muss es ein stabiles Matching M' geben, in dem Frau f mit einem noch schlimmeren Mann m' zusammen ist. (D.h. Frau f bevorzugt m über m'.) Im Matching M' muss m eine Partnerin haben und diese sei $f' \neq f$. Da nach Theorem 4 die Frau f die beste zulässige Partnerin von Mann m ist, gilt, dass Mann m die Frau f über Frau f' bevorzugt.
Somit haben wir, dass m Frau f über Frau f' bevorzugt und Frau f Mann m über Mann m' bevorzugt. Damit folgt, dass das Paar $\{m, f\}$ ein instabiles Paar im Matching M' sein muss. Folglich ist M' kein stabiles Matching und wir haben einen Widerspruch. □

1.3.5 Manipulation/Betrug von Männern bzw. Frauen

Als letztes beleuchten wir eine vollkommen andere Perspektive des Stable Matching Problems. Eigentlich haben wir hier ein sogenanntes *Mechanism Design* Problem aus der Welt der *algorithmischen Spieltheorie* vorliegen. D.h. wir wollen ein Problem lösen und den Input dafür (d.h. die Präferenzlisten) erhalten wir von verschiedenen (eigennützigen) Personen. Jeder Mann bzw. Frau hat eine Präferenzliste und bisher sind wir davon ausgegangen, dass alle beteiligten Personen vollkommen ehrlich sind und dem Gale-Shapley-Algorithmus ihre tatsächliche Präferenzliste offenbart haben.
Doch was passiert, falls ein Mann bzw. eine Frau sich nicht daran hält? Im Prinzip will doch jede Person nur einen für ihn/sie möglichst guten Partner bekommen. Da liegt es doch nahe, dass Personen auf die Idee kommen könnten, dass sie dem Algorithmus eine andere Präferenzliste als ihre tatsächliche Liste mitzuteilen, um dann einen besseren Partner zu erhalten, als er/sie bekommen hätte, wenn er/sie die Wahrheit gesagt hätte. Ein solches Verhalten nennen wir *Betrug*.
D.h. wir nehmen an, dass Personen nur dann betrügen würden, falls es für sie selbst eine Verbesserung bewirkt. Somit haben wir die Frage:

> Kann im Gale-Shapley-Algorithmus ein Mann oder eine Frau eine gefälschte Präferenzliste angeben, um damit einen besseren Partner zu erhalten?

Die Antwort lautet, dass in der Männerversion des Algorithmus (d.h. die Männer fragen an) die Frauen betrügen können die Männer jedoch nicht.[26]

Frauen können betrügen

Theorem 6. *Frauen können im Gale-Shapley-Algorithmus mit einer manipulierten Präferenzliste einen besseren Partner erhalten.*

[26]Man kann diese Antwort, d.h. die Betrugsmöglichkeit der Frauen, auch positiv auffassen: Wie weiter oben erwähnt benachteiligt der Algorithmus die Frauen in dem Sinne, dass das für die Frauen schlimmstmögliche stabile Matching berechnet wird. Immerhin können die Frauen die Vergabe sabotieren, um einen besseren Partner zu bekommen. Aus Mechanism Design Sicht ist das natürlich keine wünschenswerte Eigenschaft.

Beweis. Um zu zeigen, dass Frauen durch angeben einer falschen Präferenzliste betrügen können, betrachten wir die folgende Beispielinstanz:

$$L_{m_1} = f_2, f_1, f_3 \qquad L_{f_1} = m_1, m_3, m_2$$
$$L_{m_2} = f_1, f_3, f_2 \qquad L_{f_2} = m_3, m_1, m_2$$
$$L_{m_3} = f_1, f_2, f_3 \qquad L_{f_3} = m_1, m_3, m_2$$

Der Gale-Shapley-Algorithmus würde für diese Instanz das folgende stabile Matching produzieren:

$$M = \{\{m_1, f_2\}, \{m_2, f_3\}, \{m_3, f_1\}\}.$$

In diesem Matching erhält Frau f_1 den für sie zweitbesten Partner m_3. Wir zeigen nun, dass Frau f_1 betrügen kann, um sogar ihre Top-Präferenz m_1 zu bekommen. Angenommen Frau f_1 übergibt dem Algorithmus folgende Präferenzliste: $L'_{f_1} = m_1, m_2, m_3$ dann haben wir folgende Instanz:

$$L_{m_1} = f_2, f_1, f_3 \qquad L'_{f_1} = m_1, m_2, m_3$$
$$L_{m_2} = f_1, f_3, f_2 \qquad L_{f_2} = m_3, m_1, m_2$$
$$L_{m_3} = f_1, f_2, f_3 \qquad L_{f_3} = m_1, m_3, m_2$$

Der Gale-Shapley-Algorithmus erzeugt dann das folgende Matching:

$$M' = \{\{m_1, f_1\}, \{m_2, f_3\}, \{m_3, f_2\}\}.$$

□

Männer können nicht betrügen Wir zeigen nun noch, dass Männer nicht lügen können, um eine bessere Partnerin zu erhalten. D.h. für Männer ist es tatsächlich die bestmögliche Strategie, dem Algorithmus die tatsächliche Präferenzliste mitzuteilen.

Theorem 7. *Männer können sich im Gale-Shapley-Algorithmus durch eine gelogene Präferenzliste nicht verbessern.*

Beweis. Wir führen einen Widerspruchsbeweis. Angenommen es gibt eine Instanz I des Stable Matching Problems und darin einen Mann, sagen wir Mann m_1, der durch Übermittlung einer gelogenen Präferenzliste eine bessere Partnerin bekommt, als er bekommen hätte, wenn er seine wahre Präferenzliste angegeben hätte.
Sei W das stabile Matching, welches der Gale-Shapley-Algorithmus für Instanz I ausgibt, falls Mann m_1 (und auch alle anderen Teilnehmer) die Wahrheit sagen.
Wir nehmen an, dass Mann m_1 betrügen kann. Also nehmen wir an, dass Mann m_1 eine andere Präferenzliste als seine tatsächliche Liste übermittelt. Damit erhalten wir eine modifizierte Instanz I', die sich von I nur durch die Präferenzliste von m_1 unterscheidet.
Sei M nun das stabile Matching, dass der Gale-Shapley-Algorithmus für die Instanz I' berechnet. D.h. M ist das Matching, welches aufgrund der Manipulation von m_1 erzeugt wird.
Sei $W(m_1)$ die Partnerin von m_1 im unmanipulierten Matching W und sei $M(m_1)$ die Partnerin von m_1 im manipulierten Matching M. Die Partner von Frauen können wir analog definieren.

Wir zeigen nun, dass falls m_1 die Frau $M(m_1)$ besser findet als Frau $W(m_1)$, d.h. falls sich der Betrug für m_1 lohnt, dann ist M kein stabiles Matching für Instanz I'. Dies ist dann ein Widerspruch zur Korrektheit des Gale-Shapley-Algorithmus.

Sei B die Menge der Männer, die im manipulierten Matching M für Instanz I' eine bessere Partnerin erhalten als in Matching W für Instanz I (d.h. in der Instanz, in der alle Männer die Wahrheit sagen). D.h.

$$B = \{m \mid m \text{ findet } M(m) \text{ besser als } W(m)\}.$$

Wir zeigen zunächst, die folgende Behauptung:

Behauptung 8. *Für jeden beliebigen Mann $m \in B$ und dessen Partnerin f im manipulierten Matching M, d.h. $f = M(m)$, ist auch der Partner $m^* = W(f)$ von Frau f im unmanipulierten Matching W in der Menge B enthalten.*

Aus Behauptung 8 folgt, dass wenn eine Frau in M mit einem Mann aus B zusammen ist, dann ist dieselbe Frau auch in W mit einem Mann aus B zusammen. Die Menge S der Frauen, die mit Männern aus B zusammen sind, ist somit in M und W identisch. D.h. es gilt

$$S = \{f \mid M(f) \in B\} = \{f \mid W(f) \in B\}.$$

Beweis von Behauptung 8: Sei $m \in B$, $f = M(m)$, $m^* = W(f)$ und sei $f^* = W(m)$. Da $m \in B$, sieht die Präferenzliste von m wie folgt aus:

$$L_m = \ldots, f, \ldots, f^*, \ldots$$

Wir unterscheiden zwei Fälle:

- Es gilt $m^* = m_1$. In dem Fall gilt die Behauptung offensichtlich, da m_1 sich durch seinen Betrug ja verbessert und somit in B enthalten ist.
- Es gilt $m^* \neq m_1$. Da in L_m Frau $f = M(m)$ vor Frau $f^* = W(m)$ steht, erzwingt die Stabilität von Matching W, dass Frau f den Mann $m^* = W(f)$ besser findet als Mann m. (Sonst wäre $\{m, f\}$ ein instabiles Paar in W, da sich beide gegenseitig besser finden als ihre Partner in W.) Somit sieht die Präferenzliste von f wie folgt aus:

  $$L_f = \ldots, m^*, \ldots, m, \ldots$$

  Aufgrund der Stabilität von M für Instanz I' folgt, dass Mann m^* die Frau $M(m^*)$ über Frau f bevorzugt. (Sonst wäre $\{m^*, f\}$ in instabiles Paar in M.) Somit folgt, dass auch Mann m^* im Matching M eine bessere Partnerin hat, als im Matching W und somit gilt $m^* \in B$.

□

Für jede Frau $w \in S$ gilt nun, dass w den Mann $W(w)$ besser findet als Mann $M(w)$ (sonst wäre $\{M(w), w\}$ ein instabiles Paar in W). Somit lehnt jede Frau $w \in S$ den Mann $M(w) \in B$ in irgend einer Iteration des Gale-Shapley-Algorithmus auf Instanz I ab.

Sei $\hat{m}$ der letzte Mann aus B, der im Ablauf des Algorithmus auf Instanz I einen Antrag an eine Frau macht. Dieser Antrag wird akzeptiert und muss somit an Frau $w = W(\hat{m}) \in S$ gerichtet sein. Außerdem muss, nach Wahl von Mann $\hat{m}$, die Frau w irgendwann vorher einen Antrag von Mann $M(w)$ abgelehnt haben.
Das bedeutet, dass zum Zeitpunkt als $\hat{m}$ der Frau w einen Antrag macht, muss sie ihren aktuellen Partner $m' \notin B$, den sie besser findet als $M(w)$, verlassen, um mit $\hat{m}$ ein Paar zu bilden. Somit sieht die Präferenzliste von Frau w wie folgt aus:

$$L_w = \ldots, \hat{m}, \ldots, m', \ldots, M(w), \ldots$$

Der Mann m' kann nicht in B sein, da sonst $\hat{m}$ nicht der letzte Mann aus B wäre, der einer Frau einen Antrag macht (m' wäre dann wieder solo und würde weiter anfragen).
Da $m' \notin B$, folgt, dass m' die Frau $W(m')$ mindestens so gut findet, wie die Frau $M(m')$ (er verbessert sich unter Matching M nicht). Da Mann m' von w verlassen wird und später bei Frau $W(m')$ landet, hat er Frau w vor Frau $W(m')$ gefragt. Somit sieht die Präferenzliste von Mann m' wie folgt aus:

$$L_{m'} = \ldots, w, \ldots, W(m'), \ldots, M(m'), \ldots$$

wobei auch $W(m') = M(m')$ gelten kann.
Außerdem folgt aus $m' \notin B$, dass $m' \neq m_1$ und somit ist die Präferenzliste von m' in beiden Instanzen I und I' identisch. Damit folgt, dass $\{m', w\}$ ein instabiles Paar für Matching M für Instanz I' ist. □

1.3.6 Abschließende Bemerkungen

Das Kidney Exchange Problem wird in der Literatur auch als *House Allocation Problem* bezeichnet. Das Stable Matching Problem wird gelegentlich auch *Stable Marriage Problem* genannt.
Beide Probleme sind hochgradig praxisrelevant und noch immer Gegenstand aktiver Forschung. Diverse Varianten des TopTradingCycle- und des Gale-Shapley-Algorithmus werden in vielen Bereichen tatsächlich eingesetzt. Unter anderem bei der Vergabe von Ausbildungsplätzen in Krankenhäusern an angehende Ärzte in den USA, bei der Zuweisung von Studenten zu Wohnheimen (bzw. Zimmern) und sogar bei der Zuweisung von Spenderorganen zu Patienten (hier ist die Präferenz jeweils die Kompatibilität von Spenderorgan zum jeweiligen Patient).
Das erstaunliche an beiden Algorithmen ist, dass diese wichtige Zuweisungsprobleme auf eine relativ faire Weise lösen und alle Beteiligten danach zufrieden sind. Zudem können sie (zumindest von einer Seite her) nicht durch Einzelaktionen sabotiert/manipuliert werden. Ähnliche Zuweisungsprobleme werden in der Praxis sonst meist durch den Einsatz von Geld z.B. in Form von Auktionen gelöst. Gerade in Anwendungen, wo der Einsatz von Geld ethisch fragwürdig ist, z.B. in der Organspende, ist es wichtig, dass es effiziente Algorithmen gibt, die diese Zuweisung ohne monetären Einfluss regeln. Dies ist sogar so wichtig, dass Lloyd S. Shapley unter anderem für den Gale-Shapley-Algorithmus und seine Beiträge zum House Allocation Problem und anderen ähnlichen Zuweisungsproblemen 2012 den Wirtschaftsnobelpreis erhalten hat[27].

[27]Der Preis heißt eigentlich "Preis für Wirtschaftswissenschaften der schwedischen Reichsbank im Ge-

2 Greedy Scheduling

Greedy Algorithmen zu entwerfen, ist leicht. Für so gut wie jedes Problem ist ein solcher Ansatz meist die erste Idee, die uns einfällt. Wir haben dies bereits beim Kidney Exchange Problem und beim Stable Matching erlebt.
Viel schwieriger ist es allerdings, zu erkennen, wann ein solch naiver Ansatz wie ein Greedy Algorithmus tatsächlich funktioniert, um die bestmögliche Lösung zu finden. Generell gilt folgende Grundregel:

> **Jeder Greedy Algorithmus ist mit äußerster Vorsicht zu betrachten. Vertraue einem Greedy Algorithmus nur, wenn du seine Korrektheit bewiesen hast!**

Wir beschäftigen uns deshalb hauptsächlich damit, wie wir beweisen können, dass ein gegebener Greedy Algorithmus tatsächlich korrekt ist.
Wir werden nun zwei Probleme aus dem Bereich des *Scheduling* betrachten und für diese Greedy Algorithmen entwickeln und analysieren. Beim Scheduling geht es darum, dass man Ressourcen (z.B. Hörsäle, Drucker, Maschinen, Speicherzellen im Cache) und Jobs (z.B. Vorlesungen, Druck- oder Fertigungsaufträge oder angefragte Pages) hat und die Jobs möglichst gut mit den vorhandenen Ressourcen abarbeiten will.

2.1 Problem 1: Druckjobs planen

Wir betrachten eine Großdruckerei, die einen Spezialdrucker so auslasten möchte, um damit möglichst viele Jobs abzuarbeiten. Jeder Job hat eine Startzeit und eine Druckdauer und somit können wir annehmen, dass wir für jeden Druckjob eine Start- und eine Endzeit haben.
Der Drucker kann pro Zeitpunkt immer nur einen Job abarbeiten. Falls sich mehrere Jobs zu einem Zeitpunkt überlappen, dann müssen alle bis auf einen abgelehnt werden - es gibt somit keine Warteschlange. In diesem Szenario wollen wir nun so viele Druckjobs wie möglich abarbeiten.

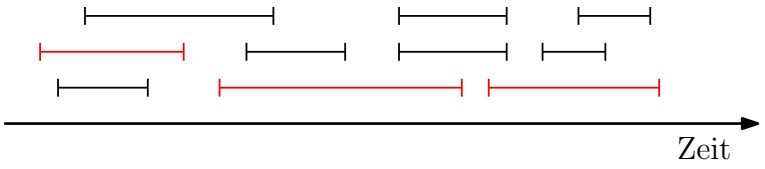
> **Geg:** n Druckjobs mit Start- und Endzeiten $(s(i), e(i))$, für $1 \leq i \leq n$.
>
> **Aufgabe:** Finde die größtmögliche Menge von sich paarweise nicht überlappenden Jobs.

Auf dem Weg zu einem Greedy Algorithmus können wir einfach ein paar einfache aber plausible Ideen durchtesten:

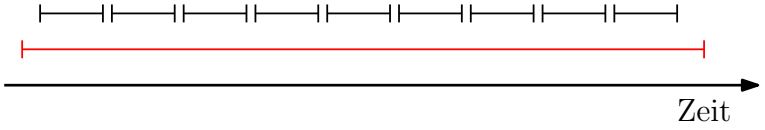
- Wir können einfach mit dem frühesten Job beginnen, alle überlappenden Jobs verwerfen und mit dem nächst frühesten Job weitermachen.

  Leider funktioniert das nicht:

denken an Alfred Nobel" und Shapley hat ihn gemeinsam mit Alvin E. Roth "Für die Theorie stabiler Verteilungen und die Praxis des Marktdesigns" erhalten. Roth hat auch an diesen Themen geforscht und maßgeblich dazu beigetragen, dass diese Algorithmen auch tatsächlich eingesetzt wurden. Der Miterfinder David Gale (dem auch der TopTradingCycle Algorithmus zugeschrieben wird) war zu diesem Zeitpunkt bereits verstorben.

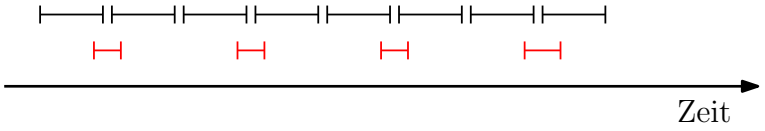


Der Algorithmus würde hier nur 3 anstatt 4 Jobs auswählen. Es kommt sogar noch schlimmer, der Algorithmus könnte beliebig schlecht werden, d.h. 1 anstatt $n-1$ viele Jobs auswählen:



- Das Problem im obigen Ansatz sind Jobs, die sehr lange dauern. Wie wäre es also, wenn wir mit dem kürzesten Job beginnen, alle überlappenden verwerfen und den nächst kürzesten nehmen, usw.?

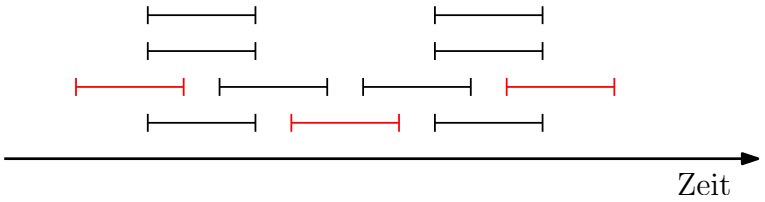
  Leider funktioniert das auch nicht:



  Der Algorithmus würde nur 4 statt 8 Jobs auswählen.

- Das Problem des obigen Ansatzes ist, dass sich ein kurzer Job mit mehreren anderen Jobs überlappen kann und wir im Beispiel gerade die Jobs ausgewählt haben, die die meisten Überlappungen haben. Dies bringt uns zur nächsten Idee: Wir berechnen für jeden Job die Anzahl der anderen Jobs, mit denen es eine Überlappung gibt. Dann beginnen wir mit dem Job mit den wenigsten Überlappungen, verwerfen alle überlappenden Jobs und fahren iterativ so fort.

  Leider funktioniert auch das nicht:



  Hier würde der Algorithmus 3 statt 4 Jobs auswählen. Das Problem ist, hier, dass der mittlere Job die wenigsten Überlappungen hat.

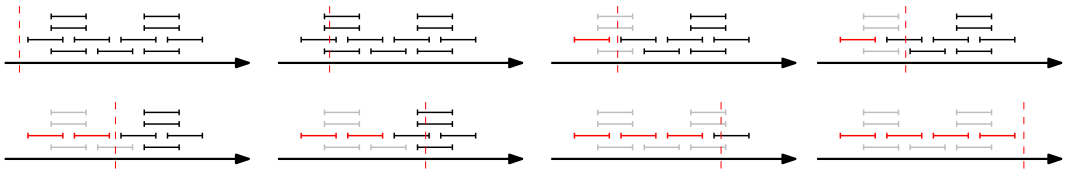
Wir sehen somit, dass es viele plausible Ideen gibt, die allerdings alle nicht zu einem bestmöglichen Algorithmus führen. Tatsächlich gibt es eine weitere plausible Idee, die funktioniert:

> Wähle immer den Job, der den frühesten Endzeitpunkt hat, entferne alle überlappenden Jobs und fahre iterativ fort, bis keine Jobs mehr verfügbar sind.

Die Idee ist also, die Ressource immer so schnell wie möglich wieder freizugeben, um somit Platz für möglichst viele spätere Jobs zu schaffen. Das klingt vernünftig, doch wir hatten auch für alle anderen Ideen von oben eine vernünftige Begründung.
Zuerst sehen wir, dass dieser Greedy-Algorithmus in all den obigen Beispielen die korrekte Lösung liefert. Wir müssen allerdings ausschließen, dass wir auch diesmal wieder ein Gegenbeispiel konstruieren können! Wir benötigen einen Korrektheitsbeweis. Für diesen Benutzen wir ein interessantes Argument, dass man *greedy stays ahead* nennt.

Korrektheitsbeweis via "greedy stays ahead": Um zu zeigen, dass unser Greedy-Algorithmus die bestmögliche Lösung liefert, müssen wir nur beweisen, dass wir genau so viele Jobs drucken können, wie der optimale Algorithmus (der z.B. per Brute-Force arbeitet). Es kann mehrere bestmögliche Lösungen geben, wichtig ist allerdings nur, wie viele Jobs gedruckt werden können.
Sei Opt die Menge der Jobs, die der optimale Algorithmus druckt und sei Alg die Menge der Jobs, die unser Greedy-Algorithmus auswählt. Wir müssen also $|Alg| = |Opt|$ zeigen.

Unser Greedy-Algorithmus arbeitet die Jobs nach aufsteigendem Endzeitpunkt ab, d.h. er wählt anfangs einen Job aus, dann einen Job der später startet (und somit auch später endet), usw. Unser Greedy-Algorithmus erweitert also immer eine bestehende partielle Lösung um einen weiteren Job, der später startet und endet als alle bisher gewählten Jobs in der partiellen Lösung. Die partielle Lösung ist somit die Lösung des Greedy-Algorithmus für eine Teilmenge aller Jobs, nämlich genau für diejenigen Jobs, deren Endzeitpunkte alle vor einem Zeitpunkt t liegen.
Wir betrachten, was im obigen Beispiel passiert:



Nun kommt das *"greedy stays ahead"* Argument:

> Für jeden Zeitpunkt t können wir die aktuelle partielle Lösung des Greedy Algorithmus mit der partiellen optimalen Lösung Opt, die nur Jobs aus Opt enthält, die bis zum Zeitpunkt t beendet sind, vergleichen.
>
> Falls für jeden Zeitpunkt t die aktuelle partielle Lösung unseres Greedy Algorithmus *mindestens so gut ist* wie die partielle Lösung des optimalen Algorithmus, dann gilt für $t = \infty$[28], dass $|Alg| \geq |Opt|$. Da Opt eine optimale Lösung ist, folgt $|Alg| = |Opt|$.

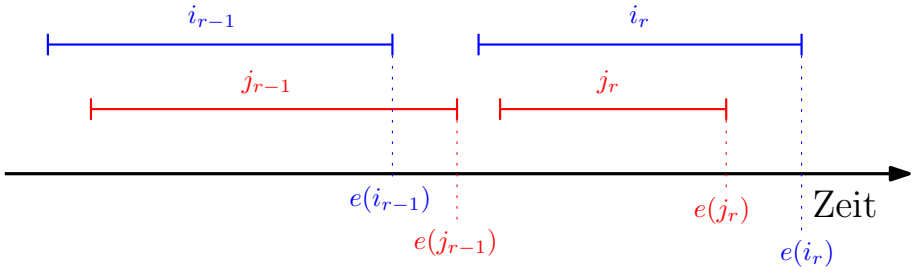
Für unseren Beweis nehmen wir an, dass $i_1, \ldots, i_k$ die Jobs aus Alg sind, wobei i_1 der erste zur Lösung hinzugefügte Jobs ist, i_2 der zweite, usw. Somit gilt $|Alg| = k$. Analog dazu nehmen wir an, dass $j_1, \ldots, j_m$ die Reihenfolge der Jobs aus Opt ist, wobei j_1 der früheste Job aus Opt ist und j_m der späteste. Da Opt eine zulässige Lösung ist, muss somit j_1 einen früheren Start- und Endzeitpunkt als j_2 haben, usw.

[28] Hier genügt ein t, welches größer ist als der Endzeitpunkt des am spätesten endenden Jobs.

Unser Greedy Algorithmus wählt den ersten Job so, dass der Drucker so schnell wie möglich wieder frei wird. Folglich gilt $e(i_1) \leq e(j_1)$. In diesem Sinne soll der Greedy Algorithmus also immer vor dem optimalen Algorithmus bleiben. Allgemein soll folgendes gelten:

Lemma 3. *Für alle $r \leq k$ gilt: $e(i_r) \leq e(j_r)$.*

Beweis. Die Form der Aussage legt einen Beweis per Induktion über r nahe.
Für $r = 1$ gilt die Aussage, da unser Greedy Algorithmus per Definition den Job mit dem frühesten Endzeitpunkt wählt.
Betrachten wir nun den Induktionsschritt von $r-1$ nach r für ein beliebiges $2 \leq r \leq k-1$: Wir nehmen nach Induktionsvoraussetzung an, dass $e(i_{r-1}) \leq e(j_{r-1})$ gilt. Wir wollen nun zeigen, dass daraus folgt, dass $e(i_r) \leq e(j_r)$ gilt.
Wir zeigen dies per Widerspruch. Dafür überlegen wir uns, was passieren müsste, damit die Behauptung nicht stimmt. Wir hätten in dem Fall folgende Situation:



Wir sehen schon an der Abbildung, dass diese Situation nicht eintreten kann, denn der Greedy Algorithmus hätte in dieser Situation nicht i_r sondern j_r gewählt, da j_r früher als i_r endet und später als $e(i_{r-1})$ beginnt. Letzteres gilt, da nach IV $e(i_{r-1}) \leq e(j_{r-1})$ gilt und da sich in *Opt* keine Jobs überlappen muss $e(j_{r-1}) \leq s(j_r)$ gelten. Somit haben wir

$$e(i_{r-1}) \leq e(j_{r-1}) \leq s(j_r)$$

und folglich einen Widerspruch. □

Nun beweisen wir, dass unser Greedy Algorithmus eine optimale Lösung erzeugt:

Theorem 9. *Es gilt $|Alg| = |Opt|$.*

Beweis. Wir beweisen die Aussage per Widerspruch. Angenommen *Alg* ist keine optimale Lösung, d.h. es gilt $|Alg| < |Opt|$ und somit muss $k < m$ gelten. Wenn wir nun Lemma 3 mit $r = k$ anwenden, dann erhalten wir, dass $e(i_k) \leq e(j_k)$ gilt. Da $m > k$, muss es also einen Job j_{k+1} in *Opt* geben, der nach $e(j_k)$ beginnt. Da $e(i_k) \leq e(j_k)$, beginnt j_{k+1} somit auch nach $e(i_k)$. D.h. nachdem der Greedy Algorithmus alle mit Job i_k überlappenden Jobs entfernt hat, muss Job j_{k+1} noch übrig sein. Der Greedy Algorithmus hätte somit nicht bei Job i_k gestoppt, da es noch verfügbare (zulässige) Jobs gibt. Wir haben somit einen Widerspruch. □

Implementierung des Greedy Algorithmus Nachdem wir nun überzeugt sind, dass unser Greedy Algorithmus die optimale Lösung erzeugt, müssen wir diesen nur noch geeignet implementieren. Dies ist einfach:

- Sortiere die Jobs aufsteigend nach Endzeitpunkt. Sei $x_1, \ldots, x_n$ diese Reihenfolge.
- Erzeuge ein Array S der Länge n, so dass $S[i] = s(x_i)$.
- Wir durchlaufen nun $x_1, \ldots, x_n$ und wählen den ersten Job x_1, dann testen wir aufsteigend für alle weiteren Jobs, ob $s(x_i) \geq e(x_1)$. Den ersten solchen Job, sagen wir x_j, wählen wir und fahren nun iterativ mit dem Test $s(x_i) \geq e(x_j)$ fort, usw.

Der erste Schritt kostet uns $\mathcal{O}(n \log n)$, der zweite und dritte jeweils $\mathcal{O}(n)$. Insgesamt haben wir Kosten in $\mathcal{O}(n \log n)$.

2.2 Problem 2: Paging

Wir betrachten als nächstes eine andere Art von Scheduling Problem, welches in so gut wie jeder Rechenhardware rund um die Uhr gelöst werden muss. Es geht darum, wie die begrenzte Ressource namens Cache möglichst effizient genutzt werden kann. Zur Erinnerung, der Cache[29] ist ein besonders schneller aber sehr kleiner Speicher. Wird eine Speicherseite genannt *Page*, vom Prozessor angefragt, dann wird zuerst im Cache danach gesucht. Ist die Seite nicht im Cache, dann gibt es einen sogenannten *cache miss* und die Page muss aus dem Hauptspeicher nachgeladen werden. Der Cache hat im Vergleich zum Hauptspeicher eine viel geringere Zugriffszeit, er ist jedoch viel kleiner. Wir abstrahieren dieses Setting etwas und betrachten folgendes Modell:

Speichermodell: Wir nehmen an, dass U die Gesamtmenge aller n Pages in unserem Speichersystem ist und dass unser Cache genau $k < n$ Pages speichern kann. Wir nehmen außerdem an, dass unser Cache immer gefüllt ist, d.h. auch zu Beginn der Bearbeitung enthält er k (beliebige) Pages. Den Inhalt des Caches notieren wir als Array $[q_1, \ldots, q_k]$. Unser Speichersystem soll nun eine Sequenz von m Anfragen

$$P = p_1, p_2, \ldots, p_m$$

bedienen, wobei alle p_i Pages aus U sind. Während der Abarbeitung der Anfragen müssen wir entscheiden, welche Pages wir zu welchem Zeitpunkt im Cache speichern. Ist eine angefragte Page p_i nicht im Cache, dann muss diese Nachgeladen werden und - falls der Cache voll ist - eine andere Page aus dem Cache verdrängt werden. Dies bezeichnen wir als cache miss. Das Hauptproblem hier ist aber nicht der cache miss an sich, sondern, dass das Nachladen einer Page aus dem Hauptspeicher sehr teuer (d.h. langsam) ist.

Das algorithmische Problem: Wir benötigen somit einen cleveren Algorithmus, um zu entscheiden, welche Pages zu welcher Zeit im Cache sind, um möglichst wenige Pages nachladen zu müssen.

[29] Eigentlich gibt es eine ganze Hierarchie von Caches, aber wir fokussieren hier auf eine einfache Architektur, die nur aus Hauptspeicher und einem Cache besteht.

Für eine gegebene Anfragesequenz P müssen wir somit einen *Verdrängungsplan*, der genau festlegt, wann wir welche Page nachladen und welche Page wir dafür aus dem Cache entfernen, entwerfen.
Unser algorithmisches Problem lautet somit:

(Offline) Paging Problem:

> **Geg.:** Cache der Größe k mit Anfangsbelegung und Anfragesequenz $P = p_1, \ldots, p_m$.
>
> **Aufgabe:** Finde einen Verdrängungsplan, der auf Sequenz P möglichst wenige Pages nachladen muss.

Das obige Problem ist die sogenannte *Offline-Variante* des Pagings, d.h. wir nehmen an, dass am Anfang die gesamte Anfragesequenz P bekannt ist. Im Gegensatz dazu gibt es die - selbstverständlich realistischere - *Online-Variante*, in der zu jedem Zeitpunkt nur die nächste Anfrage aus der Anfragesequenz bekannt ist und somit kein Blick in die Zukunft erlaubt ist. Algorithmen, die in diesem Setting arbeiten werden als *Online-Algorithmen* bezeichnet.
Das Betrachten der Offline-Variante des Pagings ist ein wichtiges Problem, immerhin liefert es uns ein Vergleichsmaß, wie gut ein Online-Algorithmus überhaupt werden kann. D.h. die Anzahl der Nachladeoperationen des besten Offline-Verdrängungsplans ist eine untere Schranke für die Anzahl der Nachladeoperationen für *alle* Online-Algorithmen. Das Verhältnis dieser beiden Werte, d.h. Anzahl Nachladeoperationen eines Online-Algorithmus versus Anzahl der Nachladeoperationen des besten Offline-Algorithmus, bezeichnet man als *competitive ratio*. Je niedriger die competititve ratio eines Online-Algorithmus, desto besser kompensiert dieser den Mangel an Information über zukünftige Page-Anfragen.

Beispiel 8. *Betrachten wir $U = \{a, b, c\}$ und einen Cache der Größe 2, der anfangs die Pages a und b enthält, d.h. der Cache ist $[a, b]$. Die Anfragesequenz lautet:*

$$P = a, c, c, b, c, a, a.$$

Betrachten wir drei mögliche Verdrängungspläne:

1. *Bei der 2. Anfrage erhalten wir einen cache miss, wir laden c nach und verdrängen damit b aus dem Cache und erhalten $[a, c]$. Wir können dann die Anfrage 3 beantworten, erhalten aber bei Anfrage 4 einen cache miss und verdrängen a durch b und erhalten $[b, c]$. Anfrage 5 wird bedient, doch Anfrage 6 erzeugt wieder ein cache miss und wir ersetzen b durch a, erhalten $[a, c]$, und bedienen die letzte Anfrage. Insgesamt erzeugt dieser Verdrängungsplan 3 Nachladeoperationen (und 3 cache misses).*

2. *Wir könnten direkt bei der ersten Anfrage schon b durch c ersetzen, erhalten $[a, c]$ und vermeiden somit den cache miss auf der zweiten Anfrage. Wir können dann Anfrage 2 und 3 bedienen. Bei Anfrage 4 laden wir b nach und verdrängen damit a, damit erhalten wir $[b, c]$. Anfrage 5 wird bedient, bei Anfrage 6 laden wir a nach und verdrängen b, erhalten $[a, c]$ und bedienen dann Anfrage 7. Dieser Verdrängungsplan erzeugt 3 Nachladeoperationen (und 2 cache misses).*

3. *Beim cache miss auf der 2. Anfrage ersetzen wir a durch c und erhalten* $[b, c]$*. Wir können dann die Anfrage 3 bis 5 ohne cache miss bedienen. Bei Anfrage 6 erhalten wir einen cache miss und verdrängen b durch a, erhalten* $[a, c]$ *und können dann die letzte Anfrage bedienen. Insgesamt haben wir damit* 2 *Nachladeoperationen (und 2 cache misses).*

Es ist recht leicht zu sehen, dass es keinen Verdrängungsplan gibt, der mit der gegebenen Anfangsbelegung auf Sequenz P mit weniger als 2 Nachladeoperationen auskommt. ◁

2.2.1 Erste Ansätze:

Spontan fallen uns gleich mehrere sinnvoll klingende Ansätze für Verdrängungspläne ein:

- *Rarest-in-the-Future (RF):* Falls wir eine Page aus dem Cache verdrängen müssen, dann verdrängen wir immer die Page im Cache, die in Zukunft am seltensten angefragt wird.
- *Farthest-in-the-Future (FF):* Wir verdrängen immer die Page im Cache, deren nächstes Auftreten in der Anfragesequenz so weit wie möglich in der Zukunft liegt.
- *Least-Recently-Used (LRU):* Wir verdrängen immer die Page im Cache, die wir am längsten nicht mehr benötigt haben.

Die Schwierigkeit ist, dass wir eine Verdrängungsstrategie haben wollen, die für alle möglichen Anfangsbelegungen auf allen möglichen Anfragesequenzen optimal ist, d.h. die immer so wenige Nachladeoperationen wie möglich erzeugt. Auf den ersten Blick ist es vollkommen unklar, ob es so eine Verdrängungsstrategie überhaupt gibt - es könnte ja sein, dass es für jede Strategie eine spezielle Anfragesequenz gibt, auf der diese extrem schlecht ist.

Es gibt noch eine weitere Schwierigkeit: Die drei Verdrängungsstrategien RF, FF und LRU verdrängen eine Page nur dann aus dem Cache, wenn es wirklich nötig ist, d.h. wenn die angefragte Page nicht im Cache ist und somit ein cache miss vorliegt. Wir haben aber bereits im Beispiel oben gesehen, dass es auch Verdrängungspläne gibt, die Elemente aus dem Cache ersetzen *bevor* ein cache miss auftritt. Es könnte ja sein, dass die optimale Strategie (sofern es sie gibt) versucht, so viele cache misses wie möglich zu vermeiden.
Betrachten wir noch ein Beispiel:

Beispiel 9. $U = \{a, b, c, d, e\}$ *und* $k = 3$ *und anfangs ist* $[a, b, c]$ *im Cache. Die Anfragesequenz lautet:*

$$P = a, b, c, d, a, d, e, a, d, b, c.$$

- *(RF): Bei Anfrage 4 wird d nachgeladen und b oder c verdrängt (beide werden nur 1 mal in der Zukunft angefragt). Nehmen wir an, dass b verdrängt wird und wir erhalten* $[a, d, c]$*. Bei Anfrage 7 wird e nachgeladen und a oder d oder c verdrängt (alle drei werden nur 1 mal in der Zukunft angefragt). Sagen wir a wird verdrängt, dann erhalten wir* $[e, d, c]$*. Bei Anfrage 8 haben wir wieder eine cache miss, a wird nachgeladen und e verdrängt (e wird gar nicht mehr angefragt) und wir erhalten* $[a, d, c]$*. Bei Anfrage 10 haben wir einen cache miss, laden b nach und verdrängen a oder d (beide werden nicht mehr benötigt). Somit erhalten wir z.B.* $[b, d, c]$*. Die letzte Anfrage kann dann bedient werden. Insgesamt haben wir 4 mal nachgeladen.*

- *(FF): Bei Anfrage 4 wird d nachgeladen und c verdrängt, wir erhalten* $[a, b, d]$*. Bei Anfrage 7 wird e nachgeladen b verdrängt. Wir erhalten* $[a, e, d]$*. Bei Anfrage 10 wird b nachgeladen und eine beliebige Page verdrängt, wir erhalten z.B.* $[b, e, d]$*. Bei Anfrage 11 wird c nachgeladen und wieder beliebig verdrängt. Insgesamt sind das* 4 *Nachladeoperationen.*

- *(LRU): Bei Anfrage 4 wird d nachgeladen und a verdrängt, wir erhalten* $[d, b, c]$*. Bei Anfrage 5 wird a nachgeladen und b verdrängt, wir erhalten* $[d, a, c]$*. Bei Anfrage 7 wird e nachgeladen und c verdrängt, wir erhalten* $[d, a, e]$*. Bei Anfrage 10 wird b nachgeladen und e verdrängt, wir erhalten* $[d, a, b]$*. Bei Anfrage 11 wird c nachgeladen und a verdrängt. Insgesamt wurde 5 mal nachgeladen. Wir sehen hier also schon, dass LRU auf dieser Sequenz schlechter als RF oder FF ist! LRU kann somit nicht optimal sein.* ◁

Nach diesem Beispiel sind die Verdrängungsstrategien RF und FF noch im Rennen. Allerdings könnte es auch noch eine ganz andere Strategie geben, die nicht erst nachlädt, wenn ein cache miss passiert. Dazu definieren wir folgendes:

Definition 10 (fauler Verdrängungsplan)**.** *Ein Verdrängungsplan heißt* faul*, wenn er nur bei einem cache miss eine Page nachlädt.*

Die Strategien RF, FF und LRU sind alle faule Verdrängungspläne.

Faule Verdrängungspläne genügen: Wir zeigen zunächst, dass wir uns bei der Suche nach einem optimalen Verdrängungsplan auf die faulen Varianten beschränken können.

Lemma 4. *Ein beliebiger Verdrängungsplan kann ohne zusätzliche Nachladeoperationen in einen faulen Verdrängungsplan umgewandelt werden.*

Beweis. Siehe Übung. □

Für jeden cache miss muss nachgeladen werden. Allein daraus folgt schon, dass es nicht weniger Nachladeoperationen als cache misses geben kann. Das obige Lemma sagt uns, dass wenn wir uns auf faule Verdrängungsstrategien beschränken, dann ist das Minimierungen der Anzahl der Nachladeoperationen äquivalent zum Minimieren der Anzahl der cache misses.

2.2.2 Optimalität von Farthest-in-the-Future via Austauschargument

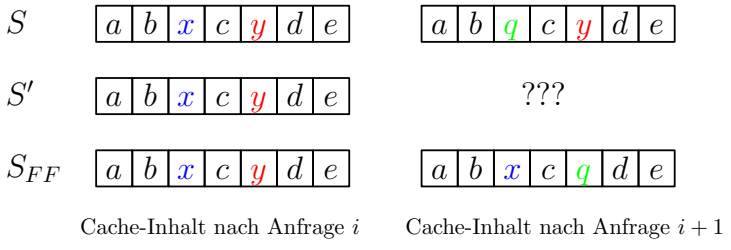
Wir zeigen nun, dass FF eine optimale Verdrängungsstrategie für alle Kombinationen von anfänglichen Cacheinhalten und Anfragesequenzen ist.
Als Beweistechnik nutzen wir ein *Austauschargument*. Dies ist ein wichtiger Trick mit dessen Hilfe man die Optimalität vieler Greedy Algorithmen beweisen kann. Streng genommen, ist auch das "greedy stays ahead" Argument ein Austauschargument.

Idee des Austauscharguments: Wir fixieren eine beliebige Instanz des Paging Problems, d.h. eine beliebige Anfragesequenz $P = p_1, p_2, \ldots, p_m$ und einen beliebigen anfänglichen Cache-Inhalt. Für diese Instanz ist S_{FF} der Verdrängungsplan, der durch Anwendung der Farthest-in-the-Future Regel entsteht. Außerdem sei S^* ein optimaler Verdrängungsplan für dieselbe Instanz, d.h. S^* erzeugt so wenige Nachladeoperationen wie möglich. Mit Hilfe von Lemma 4 können wir annehmen, dass S^* faul ist, d.h. in S^* werden nur dann Pages nachgeladen, wenn ein cache miss auftritt.
Wir zeigen dann, dass wir den faulen Verdrängungsplan S^* nach und nach in den Verdrängungsplan S_{FF} umwandeln können, *ohne dabei die Anzahl der Nachladeoperationen zu erhöhen.*
Wenn wir das geschafft haben, dann haben wir bewiesen, dass auch S_{FF} optimal für die gegebene Instanz ist. Da wir dieses Argument für jede Instanz durchführen können, folgt, dass FF optimal ist.

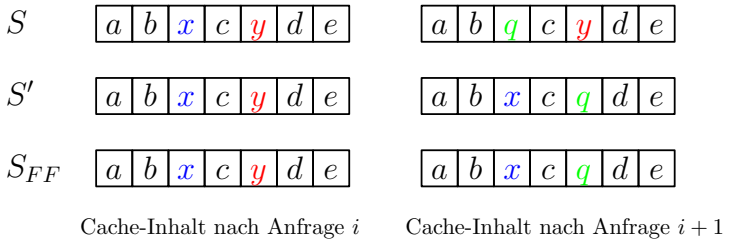
Funktionsweise eines einzelnen Austauschschritts: Bevor wir die Optimalität von S_{FF} beweisen, betrachten wir zunächst einen einzelnen Austauschschritt und zeigen, dass wir diesen ohne zusätzliche Nachladeoperationen ausführen können. Später benutzen wir diese Aussage dann induktiv, um S^* in S_{FF} ohne zusätzliche Nachladeoperationen umzuwandeln.

Lemma 5. *Für ein beliebiges $1 \leq i \leq m-1$ sei S ein fauler Verdrängungsplan, der auf den ersten i Anfragen dieselben Verdrängungsentscheidungen wie S_{FF} trifft. Dann gibt es einen faulen Verdrängungsplan S', der die selben Verdrängungsentscheidungen wie S_{FF} auf den ersten $i+1$ Anfragen trifft und nicht mehr Nachladeoperationen als S hat.*

Beweis. Wir betrachten die $i+1$-te Anfrage nach Page $q = p_{i+1}$. Da nach Annahme S und S_{FF} bis zu dieser Anfrage identische Verdrängungsentscheidungen getroffen haben, muss auch der Cache-Inhalt bei beiden Strategien zu diesem Zeitpunkt identisch sein.
Falls Page q im Cache ist, dann muss bei beiden Strategien S und S_{FF} kein Element verdrängt werden, da beide faul sind. In diesem Fall treffen somit S und S_{FF} auf der $i+1$-ten Anfrage dieselbe Entscheidung und wir können $S' = S$ wählen.
Falls q nicht im Cache ist, aber S und S_{FF} auf der $i+1$-ten Anfrage dieselbe Page aus dem Cache verdrängen, um Platz für q zu machen, dann können wir auch einfach $S' = S$ wählen.
Es verbleibt somit der Fall, dass q nicht im Cache ist, aber S und S_{FF} unterschiedliche Pages aus dem Cache verdrängen, um Platz für q zu machen. Wir nehmen an, dass S die Page x verdrängt und S_{FF} die Page y, für beliebige $x \neq y$, die zu dem Zeitpunkt im Cache sind. Die Cache-Inhalte von S und S_{FF} sind somit nach Anfrage $i+1$ nicht mehr identisch und wir müssen uns überlegen, wie genau wir S' in diesem Fall konstruieren müssen.



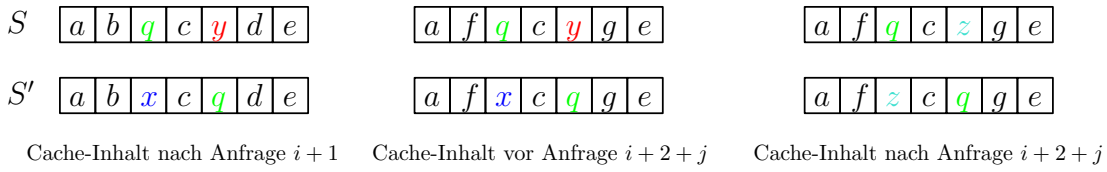
Als ersten Schritt sorgen wir dafür, dass S' die Page y anstatt x verdrängt:



Nun müssen wir noch sicherstellen, dass S' nicht mehr Nachladeoperationen als S benötigt. Das wäre einfach, wenn wir es schaffen könnten, dass die Cache-Inhalte von S und S' für alle weiteren Anfragen übereinstimmen. Allerdings ist das nicht möglich, da zu diesem Zeitpunkt S und S' leicht unterschiedliche Cache-Inhalte haben. S hat x entfernt, aber y noch im Cache, S' hat y entfernt, aber x noch im Cache.
Wir versuchen deshalb den Cache-Inhalt von S' so schnell wie möglich an den Cache-Inhalt von S anzugleichen, ohne dabei zusätzliche Nachladeoperationen zu erhalten. Sobald wir das geschafft haben, kann sich S' genau wie S verhalten und wir sind fertig.
Wir machen Folgendes: Beginnend bei Anfrage $i+2$ wird sich S' genau wie S verhalten, bis eines der folgenden Ereignisse in Schritt $i+2+j$, für ein $j \geq 0$, eintritt:

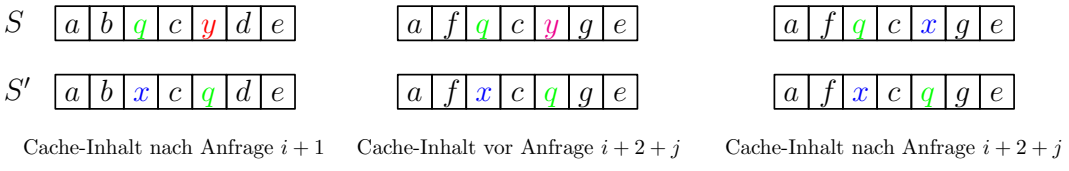
(i) Es gibt eine Anfrage nach einer Page z mit $z \neq x$ und $z \neq y$, wobei z nicht im Cache von S ist und S die Page y verdrängt, um Platz für z zu machen.

Da sich der Cache-Inhalt von S und S' nur bei x und y unterscheidet, muss die Anfrage nach z auch bei S' zu einem cache miss führen. Somit können wir in S' die Page x verdrängen und haben damit erreicht, dass nun die Cache-Inhalte von S und S' übereinstimmen. Somit kann sich S' dann genau wie S verhalten.

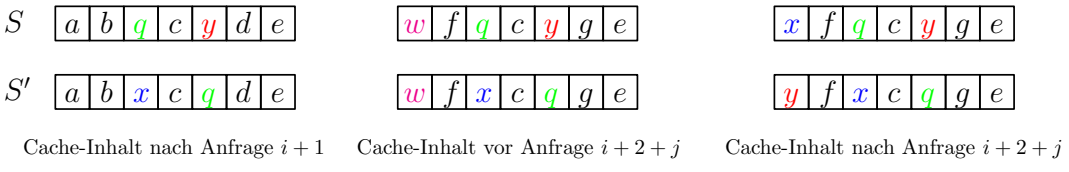


(ii) Es gibt eine Anfrage nach x und S verdrängt dafür die Page w aus dem Cache.

- Falls $w = y$, dann kann S' einfach die Anfrage nach x bedienen und die Cache-Inhalte von S und S' sind danach gleich.



- Falls $w \neq y$, dann wird auch S' die Page w aus dem Cache entfernen und dafür Page y in den Cache laden. Danach stimmen die Cache-Inhalte von S und S' überein.



Wir sind noch nicht fertig, da nun S' keine faule Verdrängungsstrategie mehr ist. Es wurde ja Page y nachgeladen, obwohl gar kein cache miss vorlag (da x ja im Cache war). Wir nutzen nun die Konstruktion aus dem Beweis von Lemma 4, um S' in seine faule Version $\hat{S}'$ umzuwandeln. Wir wissen bereits, dass diese Umwandlung keine zusätzlichen Nachladeoperationen benötigt und dass die Cache-Inhalte von $\hat{S}'$ und S_{FF} bei allen Anfragen bis nach $i+1$ übereinstimmen.

In beiden Fällen haben wir eine neue Verdrängungsstrategie S' konstruiert, deren Verdrängungsentscheidungen in den ersten $i+1$ Schritten mit S_{FF} übereinstimmen und die nicht mehr Nachladeoperationen als S erzeugt.
Wichtig ist, dass einer der Fälle (i) oder (ii) eintreten muss *bevor* in P das nächste mal eine Anfrage nach Page y vorkommt. Dies gilt, da bei Anfrage $i+1$ in S_{FF} die Page verdrängt wird, die am weitesten in der Zukunft wieder angefragt wird und dies ist genau Page y. Insbesondere muss Fall (ii) eintreten, d.h. eine Anfrage nach x in P, bevor in P das nächste mal eine Anfrage nach Page y vorkommt, da beide Pages x und y nach Anfrage i bei S und S_{FF} im Cache waren und S_{FF} die Page y verdrängt hat. □

Nun nutzen wir Lemma 5, um die Optimalität von S_{FF} zu zeigen.

Theorem 10. *S_{FF} benötigt nicht mehr Nachladeoperationen als jede andere Verdrängungsstrategie S^* und ist somit optimal.*

Beweis. Für eine gegebene Instanz starten wir mit einem optimalen faulen Verdrängungsplan (den muss es laut Lemma 4 geben) und benutzen Lemma 5 um ohne zusätzliche Nachladeoperationen einen Verdrängungsplan S_1 zu konstruieren, der mit der Verdrängungsentscheidung von S_{FF} auf der ersten Anfrage in P übereinstimmt. Danach wenden wir induktiv für $j = 1, 2, 3, \ldots, m$ Lemma 5 an, um ohne zusätzliche Nachladeoperationen Verdrängungsplan S_j zu konstruieren, dessen Verdrängungsentscheidungen mit den Entscheidungen von S_{FF} auf den ersten j Anfragen in P übereinstimmen. Jeder der Verdrängungspläne S_j benötigt nicht mehr Nachladeoperationen als S_{j-1}. Außerdem gilt, dass $S_m = S_{FF}$, da dann alle Verdrängungsentscheidungen auf allen Anfragen in P übereinstimmen. Somit benötigt S_{FF} nicht mehr Nachladeoperationen als der bestmögliche Verdrängungsplan S^* und ist somit optimal. □

2.3 Exkurs: Online Paging

Bei unserer bisherigen Diskussion des Paging Problems haben wir stets angenommen, dass die Anfragesequenz P komplett bekannt ist. D.h. unsere Algorithmen kannten den gesamten Input und alle Entscheidungen wurden auf dieser Datenbasis getroffen. Speziell beim Paging, aber auch bei vielen anderen algorithmischen Problemen, ist diese Annahme extrem unrealistisch. Beim Paging in der Praxis (z.B. innerhalb des Betriebssystems) müssen Verdrängungsentscheidungen getroffen werden, obwohl nichts oder nur wenig über die folgenden Anfragen bekannt ist.
Wir beschräftigen uns daher mit der *online* Version des Paging Problems:

Online Paging Problem:

> **Geg.:** Cache der Größe k mit Anfangsbelegung und Anfragesequenz $P = p_1, \ldots, p_m$, welche nach und nach offengelegt wird. Für alle $1 \leq i \leq m-1$ wird Anfrage p_{i+1} erst bekannt, nachdem die Anfrage p_i abgearbeitet wurde.
>
> **Aufgabe:** Finde einen Verdrängungsplan, der auf Sequenz P möglichst wenige Pages nachladen muss.

Wichtig bei einem Online Problem ist, dass die zugehörigen Algorithmen, *Online Algorithmen* genannt, Entscheidungen ohne vollständige Information treffen müssen *und dass diese Entscheidungen nicht mehr rückgängig gemacht werden können.* Im Fall des Pagings ist dies klar, da die Anfragen nach und nach beantwortet werden müssen.

Die Güte eines Online-Algorithmus: Bei der Suche nach einen geeigneten Algorithmus für das Online Paging Problem stellt sich zunächst die Frage, was überhaupt ein guter Algorithmus im Online-Setting ist. D.h. wir müssen einen Weg finden, wie wir die Güte von verschiedenen Online-Algorithmen vergleichen können.
Eine relativ einfache Möglichkeit wurde von Sleator und Tarjan 1985 vorgeschlagen: Ein Online-Algorithmus wird einfach mit dem bestmöglichen *Offline*-Algorithmus verglichen. Genauer, die Güte eines Online-Algorithmus wird durch das Verhältnis der Kosten des Online-Algorithmus zu den Kosten des optimalen Offline-Algorithmus definiert. Die nennt man auch die *competitive-ratio.*[30]

Definition 11 (α-kompetitiv)**.** *Seinen $C_{ALG}(I)$ die Kosten des Online-Algorithmus ALG auf Instanz I und seien $C_{OPT}(I)$ die Kosten des optimalen Offline-Algorithmus für dieselbe Instanz. Der Online-Algorithmus ALG heißt* α-kompetitiv*, falls eine Konstante c existiert, so dass für alle Instanzen I gilt:*

$$C_{ALG}(I) \leq \alpha \cdot C_{OPT}(I) + c.$$

Falls ein Online-Algorithmus α-kompetitiv ist, dann bedeutet dies, dass der Mangel an Information zu Kosten führt, die um einen Faktor α höher sind, als mit vollständiger Information möglich wäre. Meist kann man $c = 0$ setzen[31], dann gilt

$$C_{ALG}(I) \leq \alpha \cdot C_{OPT}(I) \iff \frac{C_{ALG}}{C_{OPT}} \leq \alpha.$$

Online-Algorithmen für das Online Paging Problem Wir haben bereits beweisen, dass Farthest-in-the-Future ein optimaler Offline-Algorithmus für das Paging Problem ist. D.h. wir haben bereits unseren Referenzalgorithmus, mit dem wir einen möglichen Online-Algorithmus vergleichen können.
Betrachten wir unsere Kandidaten aus dem vorherigen Abschnitt: FF und RF sind keine Online-Algorithmen, da für die Verdrängungsentscheidung Wissen über zukünftige Anfragen ausgenutzt wird! Es bleibt somit nur LRU übrig. Schauen wir, wie sich LRU im Online-Setting schlägt:

[30] Es gibt auch Gütemaße, die zwei verschiedene Online-Algorithmen direkt miteinander vergleichen. Solche Maße sind aber deutlich komplizierter zu definieren.

[31] Da c eine Konstante ist, spielt c für wachsende Instanzgrößen ohnehin keine Rolle. Man benötigt c nur, um einige Randfälle zu behandeln.

Beispiel 10. *$U = \{a, b, c, d, e\}$ und $k = 3$ und anfangs ist $[a, b, c]$ im Cache. Die Anfragesequenz lautet:*

$$P = a, b, c, d, a, d, e, a, d, b, c.$$

Wir haben dieses Beispiel bereits beim Offline Paging betrachtet und dabei festgestellt, dass LRU auf P genau 5 mal nachlädt, während FF nur 4 Nachladeoperationen benötigt. Das Verhältnis der Kosten beider Algorithmen ist somit $\frac{5}{4}$.
Betrachten wir eine weitere Anfragesequenz:

$$P' = a, b, c, d, a, b, c, d, a, b, c, d$$

Hier würde LRU die ersten drei Anfragen bedienen und müsste danach auf jeder Anfrage nachladen. D.h. LRU benötigt 9 Nachladeoperationen.
Schauen wir, wie sich FF auf P' schlägt: Die ersten drei Anfragen werden beantwortet. Bei der 4. Anfrage wird d nachgeladen und c verdrängt. Anfrage 5 und 6 werden beantwortet. Bei Anfrage 7 wird c nachgeladen und b verdrängt- Anfrage 8 und 9 werden dann beantwortet. Bei Anfrage 10 wird b nachgeladen und Page a verdrängt. Danach wird Anfrage 11 und 12 beantwortet. Insgesamt sind dass 3 Nachladeoperationen.
Das Verhältnis der beiden Algorithmen auf P' ist somit

$$\frac{C_{LRU}(P')}{C_{FF}(P')} = 3 = k.$$

Es gibt somit mindestens eine Anfragesequenz, auf der LRU k-mal so viele Nachladeoperationen wie FF benötigt. ◁

Die Anfragesequenz P' scheint für LRU wirklich problematisch zu sein. Tatsächlich können wir die Idee verallgemeinern und zeigen, dass es für jeden deterministischen Online-Algorithmus für das Online Paging Problem eine solche problematische Anfragesequenz gibt. Wir zeigen im Folgenden eine allgemeine untere Schranke für die Kompetitivität aller Online-Algorithmen für das Online Paging Problem:

Theorem 11. *Falls ein deterministischer Online-Algorithmus ALG α-kompetitiv für das Online Paging Problem ist, dann gilt $\alpha \geq k$, wobei k die Größe des Caches ist.*

Beweis. Sei $U = \{p_1, \ldots, p_k, p_{k+1}\}$ eine Menge von $k + 1$ beliebigen Pages. Wir nehmen o.B.d.A. an, dass ALG und OPT anfangs die Pages $p_1, \ldots, p_k$ im Cache haben.
Nun betrachten wir die folgende Anfragesequenz P: Es wird immer genau die Page angefragt, die ALG gerade nicht im Cache hat[32].
Auf Sequenz P muss ALG somit auf jeder Anfrage einen cache miss erzeugen (genau so ist P definiert). Schauen wir nun, was bei OPT auf Anfragesequenz P passiert: Angenommen OPT hat bei der i-ten Anfrage einen cache miss und muss eine Page nachladen[33]. OPT kann nun eine Page aus dem Cache verdrängen, die in den nächsten $k - 1$ Anfragen, d.h. die Anfragen $i + 1$ bis $i + k - 1$, *nicht* angefragt wird. Somit muss OPT auf k aufeinander folgenden Anfragen nur höchstens ein mal nachladen. □

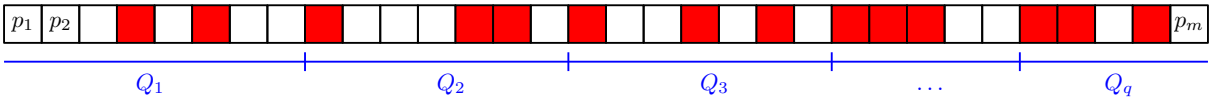
[32]Hier brauchen wir die Eigenschaft, dass ALG deterministisch ist. Sonst wäre P nicht wohldefiniert.
[33]Wir können davon ausgehen, dass OPT faul ist, d.h. nur dann nachlädt, wenn ein cache miss vorliegt.

Das Theorem sagt uns, dass wir die Suche nach einem deterministischen Online-Algorithmus mit möglichst geringer Kompetitivität beenden können, wenn wir einen Online-Algorithmus haben, der k-kompetitiv ist.
Als nächstes zeigen wir, dass die Kompetitivität von LRU genau diese untere Schranke trifft und LRU somit optimal ist.

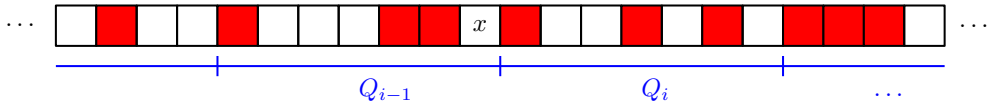
Theorem 12. *LRU ist k-kompetitiv.*

Beweis. Wir fixieren einen beliebigen Anfangsinhalt eines Caches der Größe k und eine beliebige Anfragesequenz P und markieren an welchen Stellen in P ein cache miss mit dem Verdrängungsplan LRU entstehen würde. Damit erhalten wir dann folgendes Bild:

Nun teilen wir die Anfragesequenz P in *Phasen* $Q_1, \ldots, Q_q$ auf, wobei in jeder Phase $Q_2, \ldots, Q_q$ LRU genau k cache misses hat und in Phase Q_1 sind es höchstens k viele. Diese Aufteilung können wir leicht finden, in dem wir P von hinten her durchlaufen und immer nach k cache misses eine neue Phase beginnen ($k = 3$ im Beispiel):



Um unseren Beweis zu beenden, zeigen wir nun, dass der optimale Offline-Algorithmus OPT, d.h. Farthest-in-the-Future (FF), mindestens einen cache miss in jeder der q Phasen haben muss. Damit folgt, dass LRU höchstens k cache misses pro cache miss von FF hat und somit, dass LRU k-kompetitiv ist.
In Phase Q_1 muss FF einen cache miss haben, denn LRU und FF starten beide mit der selben Cache-Anfangsbelegung und beide sind faul. D.h. der erste cache miss von LRU muss auch ein cache miss für FF sein.
Betrachten wir nun eine beliebige Phase Q_i, mit $2 \leq i \leq q$. Sei x die Page, die vor der ersten Anfrage in Q_i angefragt wurde.

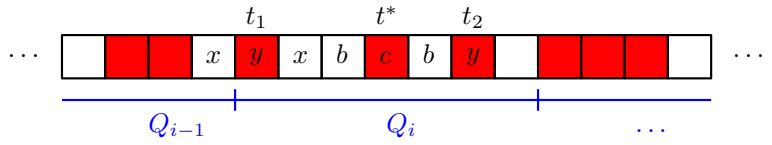


Wir zeigen nun, dass in Q_i mindestens k unterschiedliche Pages angefragt werden und all diese k Pages sind unterschiedlich zu x. Damit folgt, dass FF mindestens einen cache miss innerhalb von Q_i haben muss, da der Cache nur k groß ist und anfangs x im Cache sein muss.
Wir betrachten unterschiedliche Fälle:

1. Falls alle cache misses von LRU in Q_i bei unterschiedlichen Pages passieren und all diese Pages sind unterschiedlich zu x, dann gilt die Aussage offensichtlich.

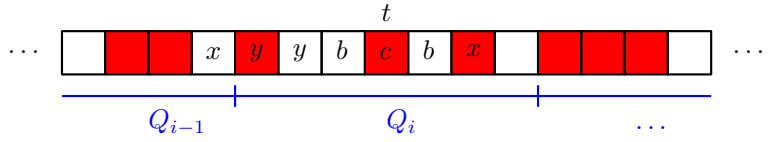
2. Falls LRU in Q_i mindestens zwei cache misses auf derselben Page y hat, dann muss Page y zum Zeitpunkt t_1 innerhalb von Q_i nicht im Cache von LRU gewesen sein. Nach dem cache miss wird y nachgeladen, doch zum Zeitpunkt $t_2 > t_1$ gibt es einen weiteren cache miss auf Page y. Somit muss die Page y irgendwann vor Zeitpunkt t_2 aus dem Cache von LRU verdrängt worden sein (sonst gäbe es den zweiten cache

miss nicht). Zu dem Zeitpunkt $t_1 < t^* < t_2$, als y verdrängt wird, muss y die Page im Cache gewesen sein, die zum Zeitpunkt t^* am längsten nicht mehr angefragt wurde (genau so ist LRU definiert).



Somit müssen zwischen dem ersten cache miss auf y, d.h. Zeitpunkt t_1 und Zeitpunkt t^* mindestens $k+1$ Anfragen nach unterschiedlichen Pages erfolgt sein. Unter diesen $k+1$ vielen Anfragen müssen somit mindestens k viele Anfragen dabei sein, die nicht Page x anfragen.

3. Falls LRU in Q_i nicht mindestens zwei cache misses auf derselben Page hat, aber einer der k cache misses auf Page x passiert, dann betrachten wir den Zeitpunkt t, als x zum ersten mal von LRU aus dem Cache verdrängt wird (x muss im Cache sein, da x ja direkt vor Phase Q_i angefragt wurde). Zum Zeitpunkt t muss x also die Page im Cache gewesen sein, die am längsten nicht mehr angefragt wurde.



Somit muss die Anfrageteilsequenz die mit der Anfrage nach x vor Q_i beginnt und bis zur Anfrage zum Zeitpunkt t reicht, mindestens $k+1$ unterschiedliche Pages (insbesondere k viele zu x verschiedene unterschiedliche Pages) enthalten. □

Wir haben eben bewiesen, dass LRU die optimale Verdrängungsstrategie im Online-Paging ist und dass der Mangel an Information über zukünftige Anfragen zu k-fachen Mehrkosten führen. Dieses Ergebnis ist paradox: Es besagt, dass LRU (und auch alle anderen deterministischen Online-Algorithmen) umso schlechter wird *je größer der Cache ist*! Tatsächlich würden wir genau das Gegenteil erwarten und Daten aus der Praxis bestätigen dies auch.
Der Grund hierfür ist der extrem starke "Gegner", den wir im Beweis der allgemeinen unteren Schranke benutzt haben. Dort wurde die Anfragesequenz immer genau so gewählt, dass jeder deterministische Online-Algorithmus bei jeder Anfrage einen cache miss erzeugt. Dieser Trick funktioniert nicht mehr so einfach, wenn wir *randomisierte Online-Algorithmen* verwenden. Tatsächlich hängt es hier sehr stark davon ab, wie genau der Gegner die Anfragesequenzen generiert. Falls der Gegner die Anfragesequenz ohne Wissen über die Ergebnisse der zufälligen Entscheidungen des randomisierten Online-Algorithmus generiert und später keine Änderungen mehr daran vornehmen darf[34], dann kann man zeigen, dass es randomisierte Online-Algorithmen gibt, die im Erwartungswert $\Theta(\log k)$-kompetitiv sind und dass es auch nicht besser geht. Das Ergebnis ist bemerkenswert! Nur durch den Einsatz von Zufall ergibt sich hier plötzlich eine exponentielle Verbesserung in der Güte!

[34]Einen solchen Gegner nennt man *oblivious adversary*.

Ein randomiserter Online-Algorithmus mit erwarteter Kompetitivität $\Theta(\log k)$ ist der Marking Algorithmus, der wie folgt arbeitet[35]:

- Anfangs sind alle Pages unmarkiert.
- Sobald eine Page angefragt wird, wird diese markiert.
- Bei einem cache miss wird eine unmarkierte Page uniform zufällig unter allen unmarkierten Pages im Cache verdrängt.
- Falls irgendwann alle Pages im Cache markiert sind und ein cache miss auftritt, dann entferne alle Markierungen und iteriere.

3 Minimum Spanning Trees

Als nächste Problemdomäne für Greedy-Algorithmen wenden uns einem sehr wichtigen Graphenproblem zu. Dieses Problem tritt beim Design von Netzwerken, wie z.B. Wasser/-Abwassernetzwerke, Strom/Telefonnetzwerken, usw. auf und ist deshalb sehr gut untersucht. Es gab sogar schon Algorithmen für dieses Problem, bevor es die Graphentheorie, wie wir sie heute kennen, gab.
Wir stellen uns folgendes Beispielszenario vor: Wir haben n Städte gegeben und wir wollen dafür sorgen, dass alle n Städte miteinander durch ein möglichst günstiges Glasfasernetz verbunden sind. Hierbei soll das Netzwerk die Kommunikation zwischen jeden Paar von Städten ermöglichen. Wir modellieren das als Graphenproblem. Gegeben ist ein ungerichteter, gewichteter Graph $G = (V, E, l)$, wobei $l : E \to \mathbb{R}^+$ eine Funktion ist, die jeder Kante eine nicht-negative Länge zuweist. Der Graph legt also fest, zwischen welchen Städten eine Direktverbindung möglich ist und wir nehmen einfach an, dass $l(\{u, v\})$ die Distanz von Stadt u zu Stadt v ist. Dies ist realistisch, wenn wir z.B. fordern dass die Glasfaserkabel entlang schon vorhandener Straßen verlegt werden sollen.
Wir wollen nun eine Teilmenge aller Kanten auswählen, so dass das Netzwerk bestehend aus den gewählten Kanten noch immer alle Knoten miteinander verbindet und die Summe über alle Kantenlängen unserer Auswahl minimal ist. Um n Knoten miteinander zu verbinden, benötigen wir mindestens $n - 1$ viele Kanten und dies passiert genau dann, wenn wir die n Knoten zu einem Baum verbinden. D.h. wir suchen die $n-1$ vielen Kanten im Graph, die einerseits alle Knoten verbinden, d.h. diese Kanten bilden einen sogenannten *Spannbaum* und die Summe über alle Längen soll minimal sein. Deshalb heißt dieses Problem *Minimaler Spannbaum* (engl: minimum spanning tree) (MST).

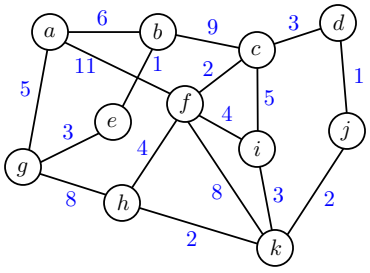
Das Minimum Spanning Tree Problem:

> **Gegeben:** Ein ungerichteter, gewichteter zusammenhängender[36] Graph $G = (V, E, l)$ mit $l : E \to \mathbb{R}^+$.
>
> **Aufgabe:** Bestimme einen Spannbaum von G, dessen Summe über alle Kantenlängen minimal ist.

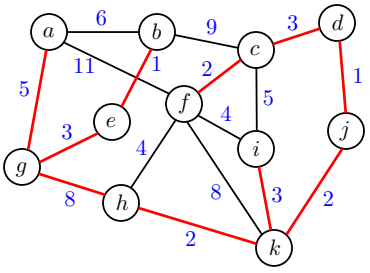
[35] Die Aussage bzgl. der Kompetitivität bezieht sich nur auf den oblivious adversary.

[36] Zusammenhang ist nicht wichtig. Alle Algorithmen lassen sich leicht auf nicht-zusammenhängende Graphen übertragen. In diesen werden dann pro Zusammenhangskomponente ein MST gefunden, ein sogenannter *minimum spanning forest*.

Für unser Beispiel bedeutet das: Wir wollen die n Städte so zu einem Telefonnetz verbinden, so dass wir insgesamt so wenig wie möglich Glasfaserkabel verlegen müssen. Im Folgenden werden wir immer davon ausgehen, dass der gegebene Graph gewichtet, zusammenhängend und ungerichtet ist. Betrachten wir zunächst ein Beispiel:



Graph G



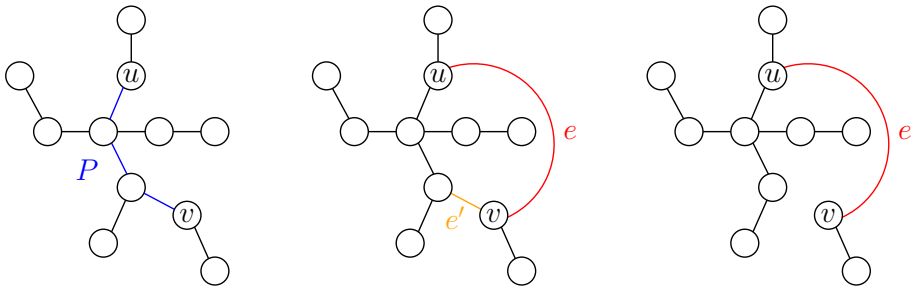
MST in Graph G

3.1 Eindeutigkeit von Minimum Spanning Trees

Bevor wir Greedy-Algorithmen für das MST-Problem betrachten, werden wir uns zunächst das Leben etwas leichter machen. Im Prinzip kann es für einen gegebenen Graph G viele MSTs geben. Allerdings gilt dies nicht, wenn alle Kantenlängen paarweise verschieden sind. Wir beweisen dies mit Hilfe eines Austauscharguments!

Lemma 6. *Sei G ein Graph in dem alle Kantenlängen paarweise verschieden sind, dann hat G einen eindeutigen MST.*

Beweis. Wir beweisen die Aussage per Widerspruch. Angenommen für G gibt es zwei unterschiedliche MSTs T_1 und T_2.
Da beide MSTs unterschiedlich sind, muss es mindestens eine Kante geben, die in einem der beiden MSTs enthalten ist, jedoch nicht im anderen MST. Da alle Kantenlängen paarweise unterschiedlich sind, muss es eine *kürzeste* Kante $e = \{u, v\}$ mit dieser Eigenschaft geben und diese Kante betrachten wir im Weiteren. O.B.d.A. können wir annehmen[37], dass e in T_1 enthalten ist, aber nicht in T_2.
Jetzt betrachten wir, was passiert, wenn wir die Kante e zum MST T_2 hinzufügen: Da T_2 ein Spannbaum ist, muss es (genau) einen Pfad P in T_2 geben, der die Endpunkte u und v von Kante e in T_2 verbindet. Wenn wir nun e hinzufügen, dann bildet der Pfad P gemeinsam mit der Kante e einen Kreis C im Graph $T_2 + e$.



[37] Wir können einfach T_1 und T_2 umbenennen.

Da der MST T_1 ein Baum ist, können nicht alle Kanten aus dem Kreis C in T_1 enthalten sein. Es muss also mindestens eine Kante e' geben, die in C ist, aber nicht in T_1. Da e in T_1 ist, muss e' auf dem Pfad P liegen.
Somit ist auch e' eine Kante, die in einem der beiden MSTs enthalten ist, jedoch nicht in dem anderen. Da wir Kante e als kürzeste solche Kante gewählt haben und da alle Kantenlängen paarweise verschieden sind, muss also

$$\ell(e) < \ell(e')$$

gelten. Wenn wir nun also e' aus T_2 entfernen und dafür e einfügen, d.h. wir betrachten den Baum $T_2 + e - e'$, dann erhalten wir einen MST, der geringere Gesamtkantenlänge als T_2 hat. Folglich kann T_2 kein MST sein. □

Das obige Lemma vereinfacht das algorithmische Problem der Bestimmung eines MSTs deutlich, da im Fall der paarweise unterschiedlichen Kantenlängen das Objekt, das wir konstruieren wollen, eindeutig ist. Tatsächlich können wir uns mit einem kleinen Trick behelfen, um immer in diesem vereinfachten Fall zu landen: Falls Kanten mit der selben Kantenlänge auftreten, dann müssen wir nur konsistentes *tie-breaking* betreiben, um diese Kanten zu unterscheiden. Genauer: Wir wollen eine konsistente Regel, die uns auch bei Kanten gleicher Länge sagt, welche von beiden die "kürzere" ist. Die Regel muss dabei die Eigenschaft haben, dass sie tatsächliche Kantenlängenunterschiede respektiert und nur bei Kanten gleicher Länge ein zweites Vergleichskriterium heranzieht und dieses zweite Vergleichskriterium konsistent anwendet. Da unsere Kanten immer Zweiermengen der Form $\{u, v\}$, mit $u \neq v$ sind[38] könnten wir einfach die Knotennamen als Vergleichskriterium heranziehen!

Konsistentes tie-breaking: Wenn wir die Länge zweier Kanten $\{u, v\}$ und $\{x, y\}$ vergleichen wollen, könnten wir, anstatt nur die Längen $\ell(\{u, v\})$ und $\ell(\{x, y\})$ zu betrachten, auch einfach den folgenden Vergleichsalgorithmus kürzer(u, v, x, y) starten:

kürzer(u, v, x, y)

Input: Knotennamen zweier Kanten $\{u, v\}$ und $\{x, y\}$
Output: die "kürzere" der beiden Kanten

```
1: if ℓ({u,v}) < ℓ({x,y}) then
2:     return {u,v}
3: else if ℓ({u,v}) > ℓ({x,y}) then
4:     return {x,y}
5: else if min{u,v} < min{x,y} then
6:     return {u,v}
7: else if min{u,v} > min{x,y} then
8:     return {x,y}
9: else if max{u,v} < min{x,y} then
10:    return {u,v}
11: end if
12: return {x,y}
```

[38] Zur Erinnerung: Schleifen, d.h. Kanten von einem Knoten zu sich selbst, haben wir bei ungerichteten Graphen explizit ausgeschlossen.

Da wir die Kantenlängen ℓ korrekt behandeln und nur bei Gleichheit von Kantenlängen konsistentes tie-breaking betreiben, ist jeder MST Algorithmus, der als Vergleichsoperator kürzer benutzt, trotzdem korrekt, d.h. es wird trotzdem ein korrekter MST gefunden. Wir "verlieren" durch die Verwendung von kürzer nur die Möglichkeit, auch andere MSTs zu finden, falls es mehrere gibt. Falls es ohnehin nur einen MST gibt (z.B. wenn alle Kantenlängen paarweise unterschiedlich sind), dann spielt es keine Rolle, ob wir kürzer als Vergleichsoperator benutzen.

Vereinfachende Annahme: Durch die Verwendung von kürzer können wir somit im Folgenden davon ausgehen, dass es für jeden zusammenhängenden, gewichteten und ungerichteten Graph G immer nur *genau einen* MST gibt. Wir werden sehen, dass uns das an einigen Stellen die Argumentation erleichtert und unsere Algorithmen etwas eleganter werden. Selbstverständlich kann man auch alle Algorithmen so abändern, dass ohne Verwendung von kürzer ein korrekter MST gefunden wird.

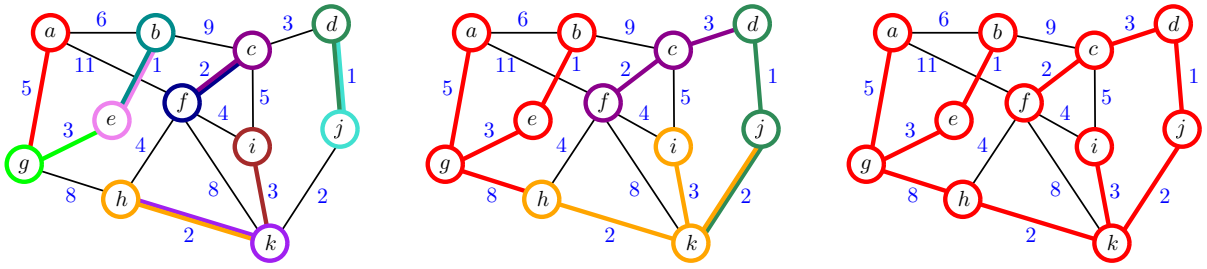
3.2 Drei Greedy-Algorithmen für das MST-Problem

Wir betrachten die drei bekanntesten Algorithmen zur Bestimmung eines MSTs. Interessanterweise sind alle drei Algorithmen eigentlich nur Instanzen eines einzigen generischen Algorithmus.
Alle Algorithmen erhalten als Eingabe einen zusammenhängenden, gewichteten und ungerichteten Graphen G und produzieren als Ausgabe die Kantenmenge des eindeutigen[39] MSTs von G.

Borůvka(G)

```
1: jeder Knoten ist ein Baum im Wald F
2: while F besteht aus mehr als einem Baum: do
3:     S ← ∅
4:     for jeden Baum T in F do
5:         füge kürzeste Kante, die in G aus T hinausführt, zu S hinzu
6:     end for
7:     Füge alle Kanten aus S in den Wald F ein
8: end while
9: return Kantenmenge von F
```

Borůvka(G)



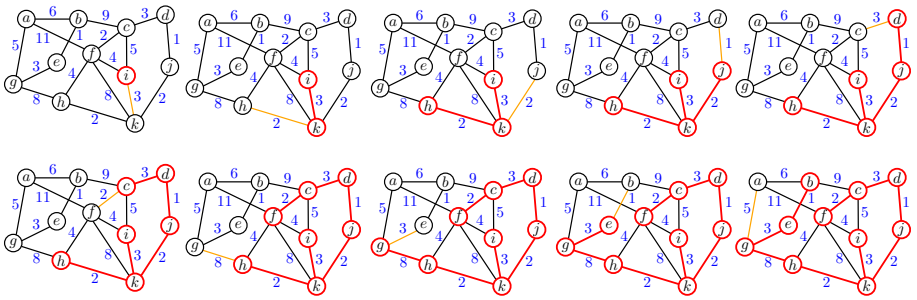
nach 1. while Durchlauf | nach 2. while Durchlauf | nach 3. while Durchlauf

[39] Zur Erinnerung: Wir können annehmen, dass es nur genau einen MST gibt.

Jarník/Prim(G)

1: wähle beliebigen Startknoten s
2: F ist ein Baum, der nur aus Knoten s besteht
3: **while** F hat weniger als n Knoten **do**
4: füge kürzeste Kante, die in G aus F hinausführt, zu T hinzu
5: **end while**
6: **return** Kantenmenge von F

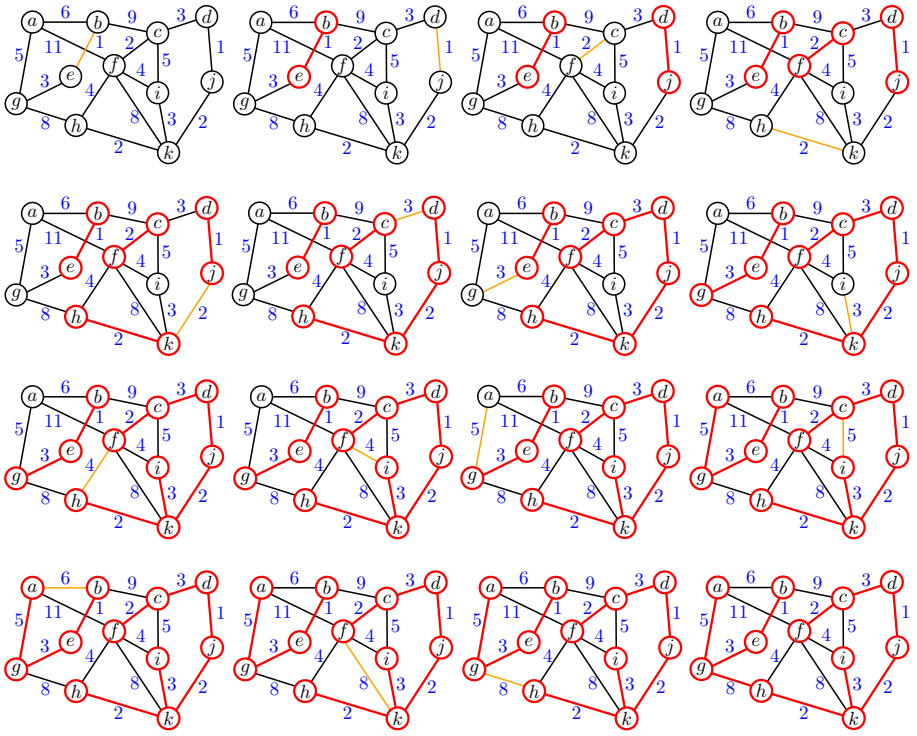
Jarník/Prim(G)



Kruskal(G)

1: jeder Knoten ist ein Baum im Wald F
2: **repeat**
3: füge Kanten in aufsteigend sortierter Reihenfolge zu F hinzu, ignoriere dabei Kanten, die einen Kreis schließen würden
4: **until** F besteht nur aus einen Baum
5: **return** Kantenmenge von F

Kruskal(G)



In all den obigen Beispielen wurde der Vergleichsoperator kürzer benutzt!

3.3 Korrektheit von Borůvka, Jarník/Prim und Kruskal

Als nächstes zeigen wir, dass die drei obigen Greedy-Algorithmen tatsächlich korrekt sind. Dazu betrachten wir zunächst die Gemeinsamkeiten aller drei Algorithmen.

Gemeinsamkeiten der Algorithmen: Alle drei Algorithmen fügen nach und nach Kanten zu einem Subgraph[40] F hinzu. An einem beliebigen Zeitpunkt innerhalb der Algorithmen gilt folgendes:

Lemma 7. *F ist ein Wald[41], d.h. F besteht aus einem oder mehreren Bäumen.*

Beweis. Für Jarník/Prim und Kruskal ist dies offensichtlich, da bei Ersterem immer nur eine Kante gewählt wird, die genau einen Endpunkt im aktuellen Baum hat. Bei Letzterem gilt dies, da explizit Kanten ausgeschlossen werden, die Kreise schließen würden. Für Borůvka zeigen wir die Aussage in der Übung. □

Insbesondere ist F in allen drei Algorithmen auch zu Beginn ein Wald! Bei Borůvka und Kruskal hat F anfangs n Bäume, die jeweils nur aus einem Knoten bestehen, bei Jarník/Prim hat F anfangs nur genau einen Baum, der aus einem Knoten besteht.
Ausgehend vom Anfangswald F werden nun bei allen drei Algorithmen Kanten zu F hinzugefügt. Sei e eine solche Kante, dann gilt für Kante e im Moment ihres Hinzufügens zu F folgendes:

Lemma 8. *Es gibt einen Baum T in F, so dass e die kürzeste Kante ist, die genau einen Endpunkt in T hat.*

Beweis. Für Jarník/Prim und Borůvka ist die Aussage klar, denn die Kanten werden genau mit dieser Eigenschaft gewählt. Bei Borůvka kann es sogar vorkommen, dass es für Kante e zum Zeitpunkt ihres Hinzufügens sogar zwei Bäume in F gibt, so dass e die kürzeste Kante mit genau einem Endpunkt in diesen Bäumen ist.
Für Kruskal gilt die Aussage aufgrund der Reihenfolge, in der die Kanten abgearbeitet werden. Die Kanten werden zunächst aufsteigend sortiert und dann wird versucht die Kanten in dieser Reihenfolge zu F hinzuzufügen, ohne dass ein Kreis entsteht. Für alle Kanten e, die zu F hinzugefügt werden, gilt somit, dass sie keinen Kreis schließen. Folglich müssen die Endpunkte von e zum Zeitpunkt des Hinzufügens in *verschiedenen* Bäumen von F liegen. Somit gibt es zwei Bäume T_1 und T_2 in F, in denen e genau einen Endpunkt hat.
Aufgrund der aufsteigend sortierten Abarbeitungsreihenfolge der Kanten, muss Kante e dann auch die kürzeste Kante mit *genau einem* Endpunkt in den Bäumen T_1 und T_2 sein, denn alle noch kürzeren Kanten e' wurden vorher entweder schon zu F hinzugefügt (und sind somit Teil eines Baumes in F) oder vorher verworfen. In beiden Fällen sind die Endpunkte von Kante e' *im selben Baum* in F. □

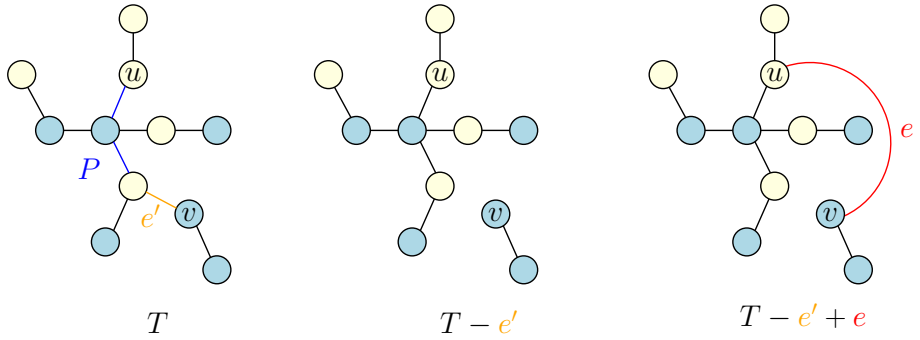
[40] Ein Subgraph $G' = (V', E')$ eines Graphen $G = (V, E)$ ist ein Graph, für den gilt, dass $V' \subseteq V$ und $E' \subseteq E$, wobei in E' nur Kanten zwischen Knoten aus V' enthalten sind.

[41] Nicht wirklich überraschend: Ein Wald ist eine Menge von Bäumen.

Wir sind fast fertig, wir müssen "nur" noch zeigen, dass alle Kanten, die wir zu F hinzufügen, auch tatsächlich im eindeutigen MST enthalten sind. Der Beweis ist überraschend einfach und benutzt ein Austauschargument, welches analog zum Beweis der Eindeutigkeit von MSTs bei paarweise unterschiedlichen Kantenlängen ist. Wir zeigen damit sogar eine noch stärkere Aussage, denn es ist völlig egal, ob wir Bäume betrachten:

Lemma 9. *Sei $X \subset V$ eine beliebige Teilmenge der Knoten von G, dann muss der MST von G die kürzeste Kanten enthalten, die genau einen Endpunkt in X hat.*

Beweis. Sei $X \subset V$ und sei $e = \{u, v\}$ die kürzeste Kante in G, die genau einen Endpunkt in X hat. Sei T ein beliebiger Spannbaum von G, der nicht die Kante e enthält.
Da T zusammenhängend ist, muss es in T genau einen[42] Pfad P geben, der die Endpunkte u und v von e verbindet. Da dieser Pfad P bei einem Knoten aus X startet und bei einem Knoten endet, der nicht in X ist, muss es auf Pfad P eine Kante e' geben, die genau einen Endpunkt in der Menge X hat. Da T keine Kreise enthält, zerfällt T in zwei Teilbäume T_1 und T_2, wenn wir die Kante e' aus T entfernen. Da e' auf Pfad P liegt und es keine Kreise in T gibt, muss u in einem der beiden Teilbäumen liegen und v im anderen.



Wenn wir nun die Kante e zum Wald $T - e'$ hinzufügen, dann muss $T - e' + e$ ein Spannbaum von G sein. Nach Wahl von e gilt $\ell(e) < \ell(e')$ und somit muss $T - e' + e$ eine geringere Gesamtkantenlänge als T haben. Der Spannbaum T kann somit kein MST von G sein. □

Somit haben wir bewiesen, dass alle drei Algorithmen nur Kanten zu F hinzufügen, die auch im MST von G sein müssen. Da es bei allen drei Algorithmen insgesamt $n-1$ viele solcher Kanten sind[43], folgt, dass bei allen drei Algorithmen am Ende F genau der MST von G ist. Wir haben also folgendes gezeigt:

Theorem 13. *Die Algorithmen Borůvka, Jarník/Prim und Kruskal sind korrekt.*

Aus der Aussage, dass F nach Terminierung der drei Algorithmen genau der MST von G ist, folgt auch, dass Kanten $\hat{e}$ die zu irgendeinem Zeitpunkt nicht in F sind, aber in diesem Moment beide Endpunkte im selben Baum T in F haben, auch später nicht mehr zu F hinzukommen. Insbesondere gilt dies für die längste Kante in einem Kreis von G.

Korollar 14. *Sei C ein Kreis in G, dann ist die längste Kante in C nicht im MST.*

[42] In jedem Baum gibt es nur genau einen Pfad zwischen zwei Knoten - wäre dies nicht so, dann enthielte der Baum einen Kreis und wäre somit kein Baum.

[43] Jeder Baum mit n Knoten hat genau $n-1$ Kanten.

3.4 Exkurs: Greedy-Algorithmen und Matroide

Alle drei betrachteten Algorithmen für das Minimum Spanning Tree Problem sind sehr ähnlich. Sie erweitern nach und nach eine Teillösung und fügen dabei jeweils eine oder mehrere Kanten hinzu, die zum Zeitpunkt ihres Einfügens die momentan kürzesten Kanten sind. Im Prinzip erweitern alle drei Algorithmen einen Wald F nach und nach um geeignete Kanten und es wird dafür gesorgt, dass F nach der Erweiterung wieder ein Wald ist. Wir werden nun sehen, dass dieses Vorgehen eigentlich nur eine Instanz eines viel allgemeineren Greedy-Algorithmus ist, der für alle Probleme funktioniert, die dem MST-Problem in gewisser Hinsicht ähnlich sind.
Beim MST-Problem hatten wir die Eigenschaft, dass der Subgraph F immer ein Wald ist und wir so lange Kanten hinzufügen, bis der Wald ein Spannbaum geworden ist. Ab diesem Zeitpunkt könnten wir keine weitere Kante mehr hinzufügen, ohne einen Kreis zu erzeugen. Wir haben also dafür gesorgt, dass wir eine gewisse Eigenschaft, nämlich die Kreisfreiheit von F, stets erhalten haben. Für F gilt folgendes:

- Zu Beginn der Algorithmen ist F kreisfrei.
- Zu jedem Zeitpunkt gilt, dass jeder Subgraph von F auch kreisfrei ist.
- So lange F noch kein Spannbaum ist, können wir eine Kante zu F hinzufügen, ohne die Kreisfreiheit zu verletzen.

Wir versuchen nun die "Quintessenz" dieser Eigenschaften zu isolieren, damit wir diese auf andere Probleme übertragen können. Dazu definieren wir, was ein *Unabhängigkeitssystem* ist.

Definition 12 (Unabhängigkeitssystem)**.** *Sei X eine endliche Menge und sei $U \subseteq \mathcal{P}(X)$ eine Teilmenge aller Mengen*[44] *über X. Das Tupel (X, U) ist ein Unabhängigkeitssystem, falls folgendes gilt:*

(i) $\emptyset \in U$ *und*

(ii) Falls $B \in U$, dann ist jede Teilmenge $A \subseteq B$ auch in U enthalten.

Die Menge U wird auch als die Menge der *unabhängigen Mengen* über X bezeichnet und X heißt auch *Grundmenge* des Unabhängigkeitssystems (X, U). Eigenschaft (i) besagt eigentlich nur, dass U nicht leer ist. Eigenschaft (ii) besagt, dass U bezüglich Teilmengenbildung abgeschlossen ist. Eine unabhängige Menge $A \in U$ heißt *Basis*, falls A keine echte Teilmenge einer unabhängigen Menge ist.
Ein Unabhängigkeitssystem wird mit einer weiteren Eigenschaft zu einem *Matroid*:

Definition 13 (Matroid)**.** *Ein Unabhängigkeitssystem (X, U) ist ein* Matroid*, falls:*

(iii) Für zwei beliebige $A, B \in U$, mit $|A| < |B|$, gibt es ein Element $e \in B \setminus A$, so dass $A \cup \{e\} \in U$.

[44] $\mathcal{P}(X)$ ist die Potenzmenge der Menge X, d.h. die Menge aller Teilmengen von X.

Eigenschaft (iii) wird auch *Austauscheigenschaft* genannt und besagt, dass falls wir eine unabhängige Menge B und eine kleinere unabhängige Menge A haben, dann können wir die Menge A durch hinzufügen eines geeignet gewählten Elements aus der Grundmenge in eine größere unabhängige Menge verwandeln. Dazu "tauschen" wir ein Element aus $B \setminus A$, d.h. ein Element aus B, welches nicht in A ist[45], in die Menge A hinein.
Wir zeigen in der Übung, dass aus der Austauscheigenschaft folgt, dass alle Basen eines Matroids dieselbe Anzahl Elemente haben müssen.

Beispiel 11. *Wir betrachten einige Beispiele:*

- *Sei M eine Matrix mit m Spalten und sei S die Menge der Spaltenvektoren von M. Wir betrachten (S, U_M), wobei eine Teilmenge $A \subseteq S$ in der Menge U_M enthalten ist, wenn die zugehörigen Spaltenvektoren in A linear unabhängig sind*[46]*.*

  (S, U_M) ist ein Matroid, da (i) eine leere Menge von Vektoren linear unabhängig ist, (ii) jede Teilmenge einer linear unabhängigen Menge von Spaltenvektoren selbst linear unabhängig ist. Eine Basis von (S, U_M) ist auch eine Basis[47] *des Vektorraums der durch die Spaltenvektoren von M aufgespannt wird. (iii) gilt, da jede Menge von linear unabhängigen Spaltenvektoren, die weniger Vektoren enthält, als die Dimension des aufgespannten Vektorraums beträgt, durch einen weiteren Basis-Vektor zu einer größeren linear unabhängigen Vektorenmenge umgewandelt werden kann.*

- *Sei Z eine beliebige endliche Menge und sei U_k die Menge aller Teilmengen von Z, die höchstens $k < |Z|$ viele Elemente enthalten.*

  (Z, U_k) ist ein Matroid, da (i) die leere Menge höchstens k Elemente enthält, (ii) jede Teilmenge einer Menge mit höchstens k Elementen auch nur höchstens k Elemente enthält und (iii) für zwei Teilmengen A und B in U_k mit $|A| < |B|$ ein Element $e \in B$ gibt, so dass $A \cup \{e\}$ nur höchstens k Elemente enthält.

- *Sei $G = (V, E)$ ein beliebiger ungerichteter Graph*[48]*. Wir betrachten $\mathcal{M}_G = (E, U_E)$, wobei U_E die Menge alle Kantenteilmengen $E' \subseteq E$ ist, so dass der Graph (V, E') keine Kreise enthält.*

  Wir beweisen in der Übung, dass $\mathcal{M}_G = (E, U_E)$ ein Matroid ist.

  Für $\mathcal{M}_G = (E, U_E)$ gilt, dass die Kantenmenge eines jeden Spanning Forests[49] *von G eine Basis in $\mathcal{M}_G = (E, U_E)$ ist. D.h. die Kantenmenge eines Spanning Forests ist eine unabhängigen Menge in U_E, die nicht Teilmenge einer größeren unabhängigen Menge ist.* ◁

Nachdem wir nun ein Matroid $\mathcal{M} = (X, U)$ definiert haben, betrachten wir, was passiert, wenn wir annehmen, dass die Elemente aus der Grundmenge X ein Gewicht haben.

[45] Ein solches Element muss es geben, da A noch nicht maximale Größe hat.

[46] Dies ist das "Urmatroid", d.h. das Originalbeispiel, welches von Whitney 1935 erwähnt wurde.

[47] Im Sinne der linearen Algebra - daher übrigens auch der Name "Basis".

[48] Wir könnten hier auch einen ungerichteten Multigraph zulassen, d.h. einen Graphen, der auch mehr als eine Kante zwischen zwei Knoten haben kann.

[49] Ein Spanning Forest eines Graphen G ist ein Wald, der so wenige Zusammenhangskomponenten wie möglich hat. D.h. ein Spanning Forest besteht aus Spannbäumen auf den Zusammenhangskomponenten von G. Falls G zusammenhängend ist, dann ist jeder Spanning Forest von G einem Spannbaum von G.

Definition 14 (Gewichteter Matroid)**.** *Ein* gewichteter Matroid $(\mathcal{M}, w)$ *ist ein Matroid* $\mathcal{M} = (X, U)$ *zusammen mit einer Gewichtsfunktion* $w : X \to \mathbb{R}^+$*, die jedem Element der Grundmenge* X *ein nicht-negatives Gewicht zuweist.*

Wenn wir uns nun die (evtl.) verschiedenen Basen eines solchen gewichteten Matroids anschauen, dann können diese unterschiedliche Gesamtgewichte haben. Dies bringt uns direkt zu folgendem Optimierungsproblem:

Matroid Optimierungsproblem

Geg.: Ein gewichteter Matroid $(\mathcal{M}, w)$.

Aufgabe: Finde eine Basis von $\mathcal{M}$ mit größtmöglichem Gesamtgewicht.

Ein Greedy-Algorithmus für das Matroid Optimierungsproblem

Wir zeigen, dass das Matroid Optimierungsproblem mit einem sehr einfachen Greedy-Algorithmus gelöst werden kann. Der Algorithmus lautet:

GreedyBasis($\mathcal{M}, w$)

Input: Gewichteter Matroid $(\mathcal{M} = (X, U), w)$
Output: Basis von $\mathcal{M}$ mit größtmöglichem Gesamtgewicht

```
1: A_X ← Array, welches alle Elemente aus X enthält
2: sortiere A_X absteigend nach Gewicht
3: B ← ∅
4: for i ← 1 to |X| do
5:    if B ∪ {A[i]} ∈ U then
6:       B ← B ∪ {A[i]}
7:    end if
8: end for
9: return B
```

Bevor wir zur Korrektheit von GreedyBasis kommen, analysieren wir zunächst die Kosten.

Kosten von GreedyBasis: Angenommen der Test in Zeile 5 kostet $f(n)$, wobei $n = |X|$ die Anzahl der Elemente in der Grundmenge ist, dann hat GreedyBasis Gesamtkosten in

$$\mathcal{O}(n \log n + n \cdot f(n)).$$

Wir müssen ja nur die Elemente der Grundmenge sortieren und dann für höchstens n mal Zeile 5 durchlaufen.

Korrektheit von GreedyBasis: Wir zeigen das folgende Theorem.

Theorem 15. *Für jeden Matroid* $\mathcal{M} = (X, U)$ *und jede Gewichtsfunktion* $w : X \to \mathbb{R}^+$ *berechnet* GreedyBasis$(\mathcal{M}, w)$ *eine Basis von* $\mathcal{M}$ *mit größtmöglichem Gewicht.*

Beweis. Nicht wirklich überraschend: Wir nutzen ein Austauschargument, um die Korrektheit von GreedyBasis zu beweisen!
Sei $B = \{b_1, \ldots, b_k\}$ die Menge von Grundelementen, die von GreedyBasis$(\mathcal{M}, w)$ zurückgeliefert wird. Nach Konstruktion des Algorithmus, muss B eine Basis von $\mathcal{M}$ sein, denn es kann kein Element mehr hinzugefügt werden, um B noch zu vergrößern (sonst hätte der Algorithmus das gemacht).
Für einen Beweis per Widerspruch nehmen wir an, dass es eine Menge $A \in U$, mit $A = \{a_1, a_2, \ldots, a_j\}$ gibt, für die

$$\sum_{i=1}^{k} w(b_i) < \sum_{i=1}^{j} w(a_i)$$

gilt. D.h. A hat ein größere Gesamtgewicht als B. Wir können im Weiteren annehmen, dass auch A eine Basis von $\mathcal{M}$ ist (falls nicht, dann könnten wir noch Elemente der Grundmenge zu A hinzufügen und erhöhen damit das Gewicht von A noch weiter). Da B und A beides Basen sind, folgt, dass $|B| = |A|$ gilt.
Wir nehmen nun an, dass die Mengen B und A nach absteigendem Gewicht indiziert sind. D.h. b_1 ist das schwerste Element in B, b_2 das zweitschwerste usw. Analog ist a_1 das schwerste Element in A.
Jetzt wählen wir i als kleinsten Index, für den $w(b_i) < w(a_i)$ gilt und betrachten die folgenden unabhängigen Mengen[50]

$$B_{i-1} = \{b_1, b_2, \ldots, b_{i-1}\} \text{ und } A_i = \{a_1, a_2, \ldots, a_{i-1}, a_i\}.$$

Es gilt $|B_{i-1}| < |A_i|$ und somit muss es laut Eigenschaft (iii) ein Element $a^* \in A_i$ geben, so dass $B_{i-1} \cup \{a^*\} \in U$ ist. Es folgt, dass

$$w(a^*) \geq w(a_i) > w(b_i)$$

gilt. GreedyBasis muss also das schwerere Element a^* in Erwägung gezogen *und verworfen* haben, bevor das leichtere Element b_i gewählt wird. Da $B_{i-1} \cup \{a^*\} \in U$, hätte dies nicht passieren können. □

3.4.1 Zwei einfache Anwendungen von GreedyBasis

Kommen wir zurück zu einem unserer Beispielmatroide:

> Sei $G = (V, E)$ ein beliebiger ungerichteter Graph. Wir betrachten $\mathcal{M}_G = (E, U_E)$, wobei U_E die Menge alle Kantenteilmengen $E' \subseteq E$ ist, so dass der Graph (V, E') keine Kreise enthält.

Für den Matroid $\mathcal{M}_G = (E, U_E)$ gilt, dass jede Basis die Kantenmenge eines Spannbaums von G ist. Wenn der Graph $G = (V, E, \ell)$ nun gewichtet ist, dann können wir den zugehörigen gewichteten Matroid wie folgt definieren: $(\mathcal{M}_G, \ell)$.

Maximum Spanning Tree: Das Matroid Optimierungsproblem für den gewichteten Matroid $(\mathcal{M}_G, \ell)$ entspricht somit der Bestimmung eines Spannbaums von G mit größtmöglichem Gesamtgewicht!
GreedyBasis$(\mathcal{M}_G, \ell)$ berechnet somit einen *Maximum Spanning Tree* für G!

[50] Die Mengen sind unabhängig, da A und B unabhängig sind und Eigenschaft (ii) gilt.

Minimum Spanning Tree: Angenommen wir ändern die Gewichtsfunktion für den Matroid $\mathcal{M}_G = (E, U_E)$ wie folgt ab: Sei $\ell_{max} = \max_{e \in E} \ell(e)$, d.h. ℓ_{max} ist die Länge der längsten Kante[51] in G. Nun wählen wir die Gewichtsfunktion für jede Kante $e \in E$ wie folgt:

$$w(e) = \ell_{max} - \ell(e).$$

Damit folgt, dass GreedyBasis$(\mathcal{M}_G, w)$ die Kantenmenge eines *Minimum Spanning Trees* berechnet. Dies ist wie folgt zu sehen: Sei B die Basis von $(\mathcal{M}_G, w)$, die von GreedyBasis gefunden wurde. Somit gilt, dass $\sum_{e \in B} w(e)$ unter allen Basen größtmöglich ist. Damit folgt, dass

$$\sum_{e \in B} w(e) = \sum_{e \in B} (\ell_{max} - \ell(e)) = (n-1)\ell_{max} - \sum_{e \in B} \ell(e)$$

unter allen Basen größtmöglich ist. Da alle Basen gleich groß sind, nämlich genau $n-1$ Kanten enthalten, folgt, dass $\sum_{e \in B} \ell(e)$ unter allen Basen kleinstmöglich ist.
Bei genauerer Betrachtung sehen wir, dass wir eigentlich nur den Algorithmus Kruskal wiederentdeckt haben!

3.4.2 Abschlussbemerkungen

Die Optimalität von GreedyBasis liefert uns für viele Probleme einen korrekten Greedy-Algorithmus, ohne, dass wir dafür viel tun müssen. Wir müssen lediglich testen, ob Problem unserer Wahl als ein Matroid Optimierungsproblem formuliert werden kann. D.h. die obigen Ausführungen kann man als die folgende Aussage verstehen:

Korollar 16. *Falls Problem P als Matroid Optimierungsproblem formuliert werden kann, dann gibt es einen korrekten Greedy-Algorithmus für P.*

Achtung, die Umkehrung diese Aussage gilt nicht! D.h. nicht jedes Problem, für das es einen korrekten Greedy-Algorithmus gibt, kann als Matroid Optimierungsproblem formuliert werden.

3.5 Kostenanalyse von Kruskal

Wir haben bereits bewiesen, dass die obigen Algorithmen korrekt sind. Jetzt überlegen wir uns noch eine effiziente Implementierung und analysieren deren Kosten.

Analyse und Implementierung von Kruskal Um den Algorithmus Kruskal analysieren zu können, müssen wir uns noch überlegen, wir wir den Test, ob eine spezielle Kante einen Kreis schließt, effizient durchführen können. Wenn wir hier für jede Kante e den gesamten Graph $G[F \cup \{e\}]$ auf Kreisfreiheit testen, dann wäre das zu teuer. Genauer: Wir hätten dann Kosten in $\Theta(m(n+m))$, wenn wir für jede Kante per Tiefen- oder Breitensuche nach einem Kreis suchen.
Schneller geht es, wenn wir uns eine spezielle passende Datenstruktur wählen. Die Idee hierbei ist, dass wir uns jeweils merken, welche Teilmengen von Knoten bereits zu ein und demselben Teilbaum gehören. Dafür definieren wir einfach einen der Knoten des Teilbaums als Repräsentant und die Anfrage für einen Knoten des Teilbaums wird dann

[51] Hier gehen wir wieder davon aus, dass es eine eindeutige längste Kante in G gibt.

als Ergebnis diesen Repräsentanten liefern. Fragen wir also für zwei Knoten an und haben beide denselben Repräsentanten, dann müssen beide im selbem Teilbaum liegen.
Um diese Idee zu realisieren, benötigen wir eine Datenstruktur, die folgende Operationen unterstützt.

- MakeSet(u). Es wird eine Menge erzeugt, die nur u enthält und u ist der Repräsentant der Menge.
- Find(u). Für einen gegebenen Knoten u wird der Repräsentant der Menge ermittelt, in der Knoten u liegt.
- Union(u, v). Für zwei Knoten mit unterschiedlichen Repräsentanten, werden die jeweiligen Knotenmengen so vereinigt, dass danach alle Knoten der Vereinigung denselben Repräsentanten haben.

Eine solche Datenstruktur nennt man auch Union-Find-Struktur.

Kruskal mit Union-Find-Struktur:

Kruskal(G) mit Union-Find-Struktur

Input: Adjazenzliste A des ungerichteten, gewichteten Graph G

```
1:  F = ∅
2:  I ← Array aller m Kanten von G, aufsteigend nach Länge sortiert
3:  for jeden Knoten u in V do
4:      Makeset(u)
5:  end for
6:  for i ← 1 to m do
7:      u, v ← Endpunkte der Kante I[i]
8:      if Find(u) ≠ Find(v) then
9:          F ← F ∪ {I[i]}
10:         Union(u, v)
11:     end if
12: end for
13: return F
```

Gesamtkosten von Kruskal mit Union-Find-Struktur: Im Algorithmus werden zunächst die Kanten aufsteigend nach Länge soritert. Danach wird $\mathcal{O}(m)$ mal Find und genau $n-1$ mal Union aufgerufen.
Seien $sort(m)$ die Kosten für das Sortieren der m Kanten, c_{Find} die Gesamtkosten für $\mathcal{O}(m)$ Find-Aufrufe und sei c_{Union} die Gesamtkosten für $n-1$ Union-Aufrufe. Dann betragen die Gesamtkosten von Kruskal:

$$\mathcal{O}(sort(m) + c_{\text{Find}} + c_{\text{Union}}).$$

Das Sortieren der Kanten kostet, z.B. mit MergeSort $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$ - doch bei vielen Graphen mit kleinen Kantenlängen kann dies durch den Einsatz von CountingSort

oder Radixsort sogar nur mit Kosten $\mathcal{O}(m)$ realisiert werden[52]. Die Kosten c_{Find} und c_{Union} hängen stark von der verwendeten Union-Find-Struktur ab. Wir machen deshalb einen kleinen Exkurs in die Welt der Union-Find Datenstrukturen.

3.6 Exkurs: Union-Find Datenstrukturen

Wir betrachten verschiedene Union-Find Datenstrukturen. Diese werden nicht nur in Kruskals Algorithmus benötigt, sondern sind Grundbausteine vieler Algorithmen. Wir interessieren uns bei allen Versionen für die Gesamtkosten von $\mathcal{O}(m)$ Find-Aufrufen und $n-1$ Union-Aufrufen.

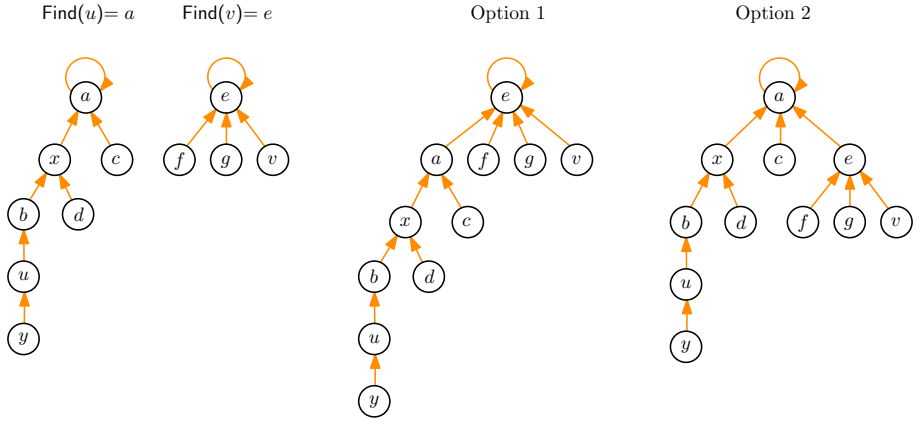
3.6.1 Disjoint-Set-Forests

Wir beginnen mit einer einfachen Union-Find Datenstruktur. Wir repräsentieren jede Menge als Baumstruktur und die Wurzel eines Baumes wird der Repräsentant der jeweiligen Menge sein.

MakeSet: Die Operation MakeSet(x) erstellt somit nur einen Baum, der nur aus einem Knoten besteht, der den Schlüssel x enthält. Außerdem hat jeder Knoten noch einen `parent`-Zeiger, der auf den Vater des Knotens zeigt. Falls ein Knoten die Wurzel des Baums ist, dann zeigt der `parent`-Zeiger auf den Knoten selbst. Die Kosten für MakeSet sind somit in $\mathcal{O}(1)$.

Find: Die Operation Find(x) ist auch sehr einfach. Wir starten beim Knoten x und folgen so lange den `parent`-Zeigern, bis wir bei der Wurzel des Baumes von x angekommen sind, d.h. bis sich ein Knoten wiederholt (Wurzeln zeigen ja auf sich selbst). Die Kosten von Find(x) hängen somit von der *Tiefe* des Baumes[53], welcher Schlüssel x enthält, ab.

Union: So richtig spannend wird es bei der Operation Union(u, v). Hier entscheidet sich, wie genau die Bäume aussehen und welche Tiefe sie haben (somit hängen die Kosten von Find davon ab, wie genau wir das Union implementieren).
Betrachten wir ein Beispiel. Angenommen die Bäume von u und v sehen wie folgt aus und wir betrachten zwei Optionen, wie wir diese Vereinigen könnten:



[52]Wir behandeln später noch die erwähnten Sortierverfahren.
[53]Die Tiefe eines Baumes ist die Länge des längsten Pfades von einem Blatt zur Wurzel des Baumes.

Bei beiden Optionen mussten wir nur jeweils einen `parent`-Zeiger umsetzen. Somit kosten uns beide Optionen nur $\mathcal{O}(1)$, sofern wir die Wurzeln der Bäume von u und v bereits kennen. Wir wissen allerdings, dass die Kosten von Find von der Tiefe des jeweiligen Baumes abhängen. Somit ist klar, dass wir Option 2 bevorzugen sollten!
Union(u, v) funktioniert somit wie folgt: Wir führen zunächst Find(u) und Find(v) aus, um die Wurzeln der Bäume von u und v zu ermitteln. Falls beide Wurzeln identisch sind, dann ist nichts zu tun. Falls die Wurzeln ungleich sind, dann hängen wir den Baum mit geringerer Tiefe unter die Wurzel des Baumes mit größerer Tiefe. Um dies zu tun, merken wir uns nebenbei noch in den Wurzeln die Tiefe des jeweiligen Baumens und aktualisieren diesen Eintrag entsprechend, wenn wir zwei Bäume vereinigen. Die Kosten von Union(u, v) werden von den Kosten der beiden Find-Aufrufe dominiert. Somit hängen auch hier die Kosten von der Tiefe der betroffenen Bäume ab.
Da die Kosten von Find und Union von der Tiefe der Bäume abhängen, sollten wir die Tiefe genauer analysieren. Wir zeigen, dass wenn wir Union so durchführen, wie oben beschrieben, dann ist die Tiefe in allen Bäumen immer sehr klein.

Lemma 10. *In einem Disjoint-Set-Forest mit n Elementen haben die Bäume höchstens eine Tiefe von $\log n$.*

Beweis. Siehe Übung. Hinweis: Induktion über die Tiefe der Bäume.

□

Somit gilt, dass jede Find- und Union-Operation im worst-case $\mathcal{O}(\log n)$ kostet.

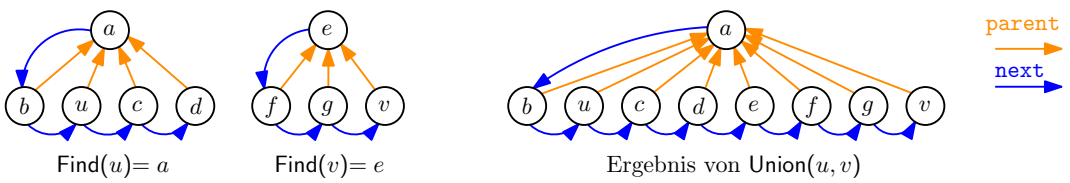
Kruskal mit Disjoint-Set-Forests: Wenn wir Disjoint-Set-Forests als Union-Find Datenstruktur in Kruskals Algorithmus nutzen, dann haben wir Gesamtkosten in

$$\mathcal{O}(sort(m) + m \cdot \mathcal{O}(\log n) + (n-1) \cdot \mathcal{O}(\log n)) = \mathcal{O}(sort(m) + \mathcal{O}(m \log n)).$$

Da $sort(m) \in \mathcal{O}(m \log n)$ haben wir Gesamtkosten in $\mathcal{O}(m \log n)$. Wir sehen hier, dass wir selbst im Fall, wenn wir die Kantenlängen schneller sortieren könnten, z.B. wenn $sort(m) \in \mathcal{O}(m)$, wir trotzdem Gesamtkosten in $\mathcal{O}(m \log n)$ haben.

3.6.2 Shallow-Trees

Nun betrachten wir eine noch einfachere Union-Find Datenstruktur.
Shallow-Trees funktionieren wie folgt: Für jede Menge von Elementen stellen wir ein Repräsentantenelement der Menge als Wurzel eines gerichteten Baums dar und alle anderen Elemente der Menge haben einen `parent`-Zeiger auf diesen Repräsentaten. Zudem gibt es noch, beginnend beim Repräsentanten, einen `next`-Zeiger bei jedem Knoten, damit wir einfach alle Elemente einer Menge durchlaufen können.
Somit muss bei MakeSet(u) nur ein gerichteter Baum mit einem Knoten angelegt werden. Bei Find(u) muss nur geprüft werden, wer der Vater von u im jeweiligen gerichteten Baum, in dem u liegt, ist bzw. ob u die Wurzel eines Baums ist. Bei Union(u, v) passiert folgendes:



Zunächst werden die Repräsentanten von u und v ermittelt. Es wird geprüft, wieviele Kinder die jeweiligen Repräsentanten haben. Dann werden alle Knoten des Baums, dessen Knoten weniger Kinder hatte, als neue Kinder in den anderen Baum eingefügt[54]. Haben die Repräsentanten gleich viele Kinder, dann ist es egal, welcher Baum gewählt wird.
Für die Gesamtkosten der Shallow-Trees gilt Folgendes:

Lemma 11. *Eine beliebige Folge von $\mathcal{O}(m)$ Find-Aufrufen und $n-1$ Union-Aufrufen kostet bei Shallow-Trees insgesamt $\mathcal{O}(m + n \log n)$.*

Beweis. Ein Aufruf von Find kostet $\mathcal{O}(1)$. Somit kosten alle $\mathcal{O}(m)$ vielen Find-Aufrufe insgesamt $\mathcal{O}(m)$. Wir müssen also nur noch die Gesamtkosten aller Union-Aufrufe bestimmen.
Im worst-case kostet ein Union-Aufruf $\Theta(n)$, da z.B. beim Vereinigen zweier Mengen, die jeweils etwa die Hälfte aller n Elemente enthalten, bei $\Theta(n)$ vielen Elementen der `parent`-Zeiger aktualisiert werden muss. Für das Anpassen der `next`-Zeiger fallen pro Union nur konstante Kosten an, da lediglich zwei verkettete Listen zusammengefügt werden müssen. Mit herkömmlicher worst-case Analyse erhalten wir, dass $n-1$ Union-Aufrufe im worst-case $\mathcal{O}(n^2)$ kosten. Tatsächlich ist das viel zu pessimistisch.
Der Trick für eine genauere Analyse ist, die Gesamtkosten aller `parent`-Zeiger-Aktualisierungen *eines einzelnen Elements* abzuschätzen. Wir fragen uns, wie oft ein Element im kleineren der beiden Bäume bei einem Union-Aufruf liegen kann.
Die Antwort lautet $\log n$ mal, da sich nach jedem Aufruf die Anzahl der Knoten im Baum des betrachteten Elements mindestens verdoppelt. Dies kann nur höchstens $\log n$ mal passieren, bis der Baum des betrachteten Elements alle n Elemente enthalten muss. Wir haben n Elemente und somit betragen die Gesamtkosten aller `parent`-Zeiger-Aktualisierungen, die in den $n-1$ Union-Aufrufen anfallen $\mathcal{O}(n \log n)$. Wir haben also gezeigt, dass jeder Union-Aufruf *amortisiert* Kosten in $\mathcal{O}(\log n)$ hat. □

Kruskal mit Shallow-Trees: Wenn wir Shallow-Trees als Union-Find Datenstruktur in Kruskals Algorithmus nutzen, dann haben wir Gesamtkosten in

$$\mathcal{O}(sort(m) + m + n \log n).$$

Dies können wir auch mit $\mathcal{O}(m \log n)$ abschätzen, doch hier sehen wir, dass wir tatsächlich schneller sind, falls wir die Kantenlängen schneller sortieren können. Genauer: Falls $sort(m) \in \mathcal{O}(m)$, dann betragen die Gesamtkosten nur $\mathcal{O}(m + n \log n)$. Das ist schon recht nah ander trivialen unteren Schranke von $\Omega(n + m)$, die sich allein daraus ergibt, dass wir den gesamten Graph betrachten müssen.
Wir werden nun sehen, dass wir im Fall $sort(m) \in \mathcal{O}(m)$ noch erheblich viel näher an die triviale untere Schranke herankommen können. Dafür verwenden wir wieder die Disjoint-Set-Forests, jedoch mit einer einfachen aber genialen Modifikation.

3.6.3 Disjoint-Set-Forests mit Pfadkompression

Wir haben gesehen, dass bei den Disjoint-Set-Forests das Find teuer ist und die Kosten des Union von den Kosten für Find dominiert werden. Wenn beide Wurzeln der zu vereinigenden Bäume bekannt ist, dann würde Union nur $\mathcal{O}(1)$ kosten. Bei den Shallow-Trees war es

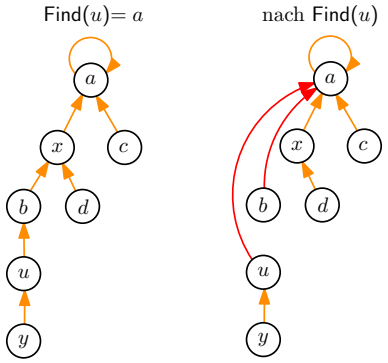
[54] Man kann hier auch die Anzahl Knoten im Baum nehmen - da der Baum nur Tiefe 1 hat, ist das äquivalent.

andersherum, dort war das Find billig und das Union teuer. Das Find ist bei Shallow-Trees deshalb billig, weil alle Knoten direkt unter der Wurzel der Bäume hängen.
Es stellt sich nun die Frage, ob es nicht eine "Mischung" aus beiden Datenstrukturen gibt, welche die Vorteile beider Seiten hat. Die Disjoint-Set-Forests scheinen ein guter Ausgangspunkt zu sein, denn da ist noch offensichtliches Verbesserungspotential vorhanden.
Um eine Idee dafür zu bekommen, betrachten wir, was bei einem Disjoint-Set-Forests bei zwei Find-Anfragen nach demselben Element x passiert, wenn zwischen beiden Anfragen keine Union-Anfrage, die den entsprechenden Baum betrifft, vorkommt: In diesem Fall machen beide Find-Anfragen exakt dasselbe! Wir hätten uns auch einfach das Ergebnis der Anfrage merken können und hätten die zweite Anfrage damit in konstanter Zeit beantworten können! Dies geht auch ohne Zusatzspeicher, wir hätten ja den Knoten x auch einfach direkt an die Wurzel hängen können. Tatsächlich können wir diese Idee nicht nur für den Knoten x umsetzen, sondern auch *für alle Knoten auf dem Pfad von x zur Wurzel*. Diese Idee nennt man auch *Pfadkompression*. Dahinter steckt, dass wir bei einem Find etwas zusätzliche Arbeit verrichten, die uns dann später viel Arbeit erspart[55].
Wir ändern einfach die Funktion Find leicht ab:

Find(x)

```
1: if x ≠ parent(x) then
2:     parent(x) ← Find(parent(x))
3: end if
4: return parent(x)
```

Betrachten wir am Beispiel, was bei Find(x) passiert:



Für das Union hatten wir uns die Tiefe der Bäume in der jeweiligen Wurzel gespeichert. Mit der obigen Version von Find ist das nicht mehr korrekt - wir ändern die Tiefe evtl. bei jedem Find. Wir ignorieren das Problem komplett und geben der Tiefe einfach einen anderen Namen, nämlich *rank*. Die Funktionen MakeSet und Union lauten dann:

[55] Die "Alltagsversion" davon wäre: Wenn du etwas suchst, dann räume das gesuchte Objekt und alle anderen, die du bei deiner Suche angetroffen hat, während der Suche gleich an die richtige Stelle. Die nächste Suche nach diesen Objekten geht dann deutlich schneller

MakeSet(x)

```
1: parent(x) ← x
2: rank(x) ← 0
```

Union(x, y)

```
1: x* ← Find(x)
2: y* ← Find(y)
3: if rank(x*) > rank(y*) then
4:    parent(y*) ← x*
5: else
6:    parent(x*) ← y*
7:    if rank(x* = rank(y*)) then
8:       rank(y* ← rank(y*)+1)
9:    end if
10: end if
```

Zu beachten ist, dass sich die Kosten für Find nur um einen konstanten Faktor erhöhen. D.h. Find hat immer noch worst-case Kosten in $\mathcal{O}(\log n)$. Dies ist viel zu pessimistisch abgeschätzt - wir haben Find ja so abgeändert, dass bei einem Aufruf spätere Find-Aufrufe für Knoten, die auf dem selben Pfad zur Wurzel lagen, extrem billig sind. In einer *Folge* von Find-Aufrufen müssten die Kosten pro Operation amortisiert deutlich weniger sein. Dies stimmt! Wir werden nun zeigen, dass die *amortisierten* Kosten einer Find-Operation in $\mathcal{O}(\log^* n)$ liegen, wobei $\log^* n$ der *iterierte Logarithmus von* n ist. d.h. die Anzahl, wie oft wie logarithmieren müssen, um einen Wert kleiner als 1 zu erhalten:

$$\log^* n = \begin{cases} 1 & \text{falls } n \leq 2 \\ 1 + \log^*(\log n) & \text{sonst.} \end{cases}$$

Die Funktion $\log^* n$ wächst lächerlich langsam, z.B. ist

$$\begin{aligned} \log^*(2^{1024}) &= 1 + \log^*(1024) = 1 + 1 + \log^*(10) = 1 + 1 + 1 + \log^*(3.3219\ldots) \\ &= 1 + 1 + 1 + 1 + \log^*(1.7320\ldots) = 5 \end{aligned}$$

3.6.4 *Analyse: Amortisierte Kosten von Find in $\mathcal{O}(\log^* n)$

Für die Analyse benöten wir die folgenden Fakten, die fast alle direkt aus der Definition von Find und Union folgen.

- Falls ein Knoten x nicht die Wurzel eines Baumes ist, dann muss der rank von x kleiner als der rank seines Vaters sein.
- Sobald parent(x) sich ändert, muss der neue Vater größeren rank als der alte Vater haben.
- Die Anzahl Knoten in einem Baum ist exponentiell im rank der Wurzel des Baumes. Speziell gilt, dass wenn die Wurzel eines Baumes rank d hat, dann befinden sich mindestens 2^d Knoten im Baum.
- Der größtmögliche rank ist $\lfloor \log n \rfloor$.
- Für jede beliebige natürliche Zahl r gibt es höchstens $\frac{n}{2^r}$ viele Knoten mit rank r.

Die letzte Aussage beweisen wir noch, da sie nicht komplett offensichtlich ist:
Wähle ein beliebiges r. Nur Wurzelknoten eines Baumes können ihren rank ändern. Sobald sich der rank eines Wurzelknotens x von $r-1$ auf r ändert, markieren wir alle Knoten im Baum von x. Da sich ranks nur erhöhen können, wird folglich jeder Knoten im Wald nur höchstens ein mal markiert (immer dann, wenn die zugehörige Wurzel von rank $r-1$ nach rank r wechselt). Insgesamt gibt es n Knoten im Wald und jeder Wurzelknoten mit rank r hat mindestens 2^r viele Knoten markiert. Damit folgt, dass es höchstens $\frac{n}{2^r}$ viele Knoten mit rank r geben kann.
Nun kommen wir zur Analyse. Bis vor wenigen Jahren gab es für diese Analyse nur sehr komplizierte Beweise[56] - 2005 haben dann Seidel und Sharir einen recht einfachen Beweis vorgeschlagen.
Für die Analyse betrachten wir zwei neue Operationen auf Disjoint-Set-Forests, nämlich die Operation Compress(x, y), welche einen beliebigen Pfad in einem Baum komprimiert, d.h. nicht nur Pfade zur Wurzel, und die Operation Shatter(x), die jeden Knoten auf einem Wurzel-Blatt-Pfad zu seiner eigenen Wurzel macht.

Compress(x, y)

Input: x und y, wobei y auf dem Pfad von x zur Wurzel liegen muss

```
1: if x ≠ y then
2:     Compress(parent(x), y)
3:     parent(x) ← parent(y)
4: end if
```

Shatter(x)

```
1: if parent(x) ≠ x then
2:     Shatter(parent(x))
3:     parent(x) ← x
4: end if
```

Im Prinzip passiert bei jedem Aufruf Find(x) ein Aufruf von Compress(x, y), wobei y die Wurzel des Baumes von x ist. Deshalb können wir auch einfach eine beliebige Folge von Union und Compress Operationen analysieren. Wir sehen später noch, wozu wir die Operation Shatter brauchen.

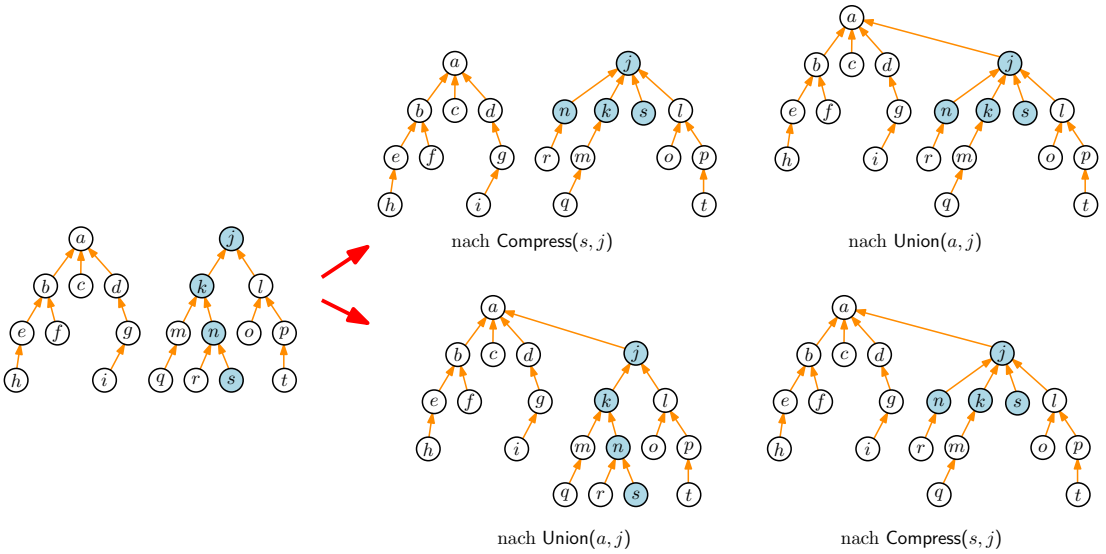
Annahmen über die Operationenfolge: Wir wollen die Kosten einer Folge von $n-1$ Union- und m Compress-Operationen analysieren.
Wir treffen hierbei zwei Annahmen, die uns die Analyse erleichtern, aber nichts an den Gesamtkosten der Operationenfolge ändern:

(1) Sämtliche Union-Operationen erhalten immer nur Wurzelknoten als Argumente.

(2) In der Operationenfolge kommen zuerst alle Union-Operationen und danach erst alle Compress-Operationen vor.

Die erste Annahme können wir treffen, da wir einfach die in Union enthaltenen Find-Aufrufe (und somit auch die zugehörigen Compress-Aufrufe) als extra Operation auslagern können. Wir erhalten somit, dass jede Union-Operation nur konstante Kosten hat!
Die zweite Annahme können wir treffen, da die Compress-Operation ja explizit definiert, bis zu welchem Knoten komprimiert wird. Damit ist leicht zu sehen, dass die Anzahl der Zeigeränderungen exakt gleich bleibt, wenn wir die Reihenfolge einer Compress- und einer Union-Operation vertauschen. Hier ein Beispiel:

[56] Das ist auch der Grund, warum in vielen Lehrbüchern diese Analyse gern mal weggelassen wird.



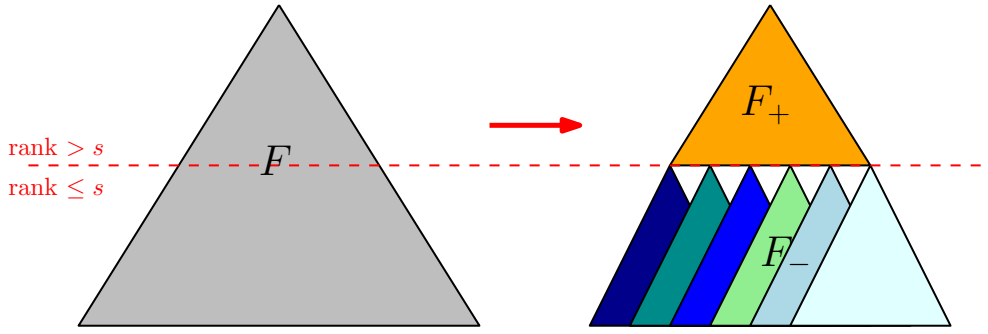
Gesamtkosten der Operationenfolge: Durch Annahme (1) haben wir, dass jede Union-Operation konstante Kosten hat. Wir müssen somit nur noch die amortisierten Kosten von Compress ermitteln. Die Kosten von Compress bestehen aus den Kosten der Zeigeränderungen und konstant viel Zusatzaufwand, d.h. es genügt hier, wenn wir uns auf die Analyse der Zeigeränderungen beschränken.
Wir definieren daher die Funktion $T(m, n, r)$ als Kostenmaß für Compress. $T(m, n, r)$ steht für die worst-case Anzahl der Zeigeränderungen in einer beliebigen Folge von höchstens m Compress-Operationen, die auf einem Disjoint-Set-Forest mit höchstens n Knoten ausgeführt wird, wobei der größtmögliche rank im Wald höchstens r ist. Eine einfache obere Schranke für $T(m, n, r)$ ist Folgende:

Theorem 17. $T(m, n, r) \leq nr$.

Beweis. Jeder Knoten kann höchstens r mal seinen `parent`-Zeiger ändern, da der neue Vater bei einer solchen Änderung immer größeren rank als der entsprechende Knoten hat. □

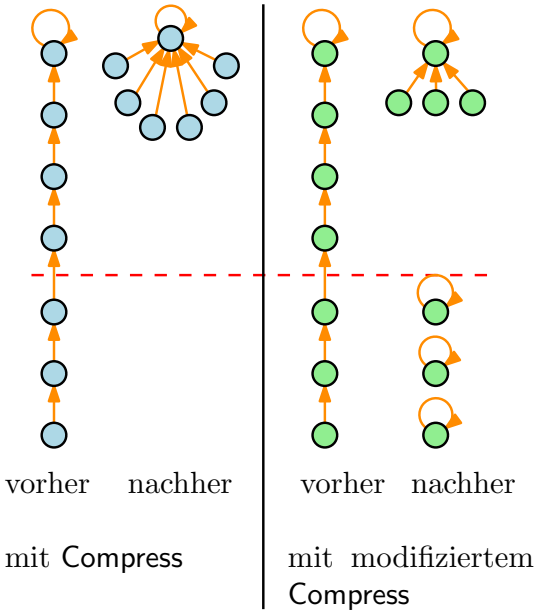
Ein Divide-and-Conquer Ansatz: Wir überlegen uns nun, wie wir unsere Operationenfolge rekursiv in Teiloperationen zerlegen können. Dies ist der eigentliche clevere Schritt unserer Analyse, denn es ist von vornherein nicht offensichtlich, wie so eine Aufteilung aussehen kann.
Wir fixieren einen Disjoint-Set-Forest F mit n Knoten und maximalem rank r und eine Folge C von m Compress-Operationen auf F. Wir bezeichnen mit $T(F, C)$ die Gesamtanzahl von Zeigeränderungen die Folge C auf Wald F auslöst.
Unser Ziel ist eine rekursive Formulierung für $T(F, C)$ und dafür "sägen" wir die Bäume im Wald F einfach "horizontal" durch. Genauer, wir partitionieren den Wald F in zwei Wälder F_- und F_+, wobei der Wald F_- alle Knoten enthält, deren rank höchstens s ist und F_+ enthält alle Knoten deren rank größer als s ist. D.h. F_+ und F_- ergeben sich, indem in jedem Baum oberhalb von allen Knoten mit rank s der Baum aufgetrennt wird. Da der rank steigt, je näher wir zur Wurzel eines Baumes kommen, folgt, dass jeder Vorgänger[57] eines Knotens mit rank größer als s auch einen rank größer als s haben muss.

[57] Vorgänger eines Knotens x sind alle Knoten, die auf dem Pfad von x zur Wurzel liegen.



F bestehend aus nur einem Baum Die Wälder F_+ und F_-

Seien n_- und n_+ die Anzahlen der Knoten in F_- und F_+ und sei m_+ die Anzahl der Compress-Operationen, die mindestens einen Knoten aus F_+ betreffen und sei $m_- = m - m_+$.
Jetzt kommen wir zum Trick der Zerlegung: Jede Folge von Compress-Operationen auf F kann somit in eine Folge von Compress-Operationen auf F_+ und eine Folge von Compress- und Shatter-Operationen auf F_- zerlegt werden, ohne dabei die Kosten zu erhöhen. Dies geht ganz leicht, in dem wir einfach verbieten, dass ein Knoten aus F_- einen Vater aus F_+ bekommt. Falls dies passieren würde, dann setzen wir den `parent`-Zeiger des Knotens einfach auf ihn selbst. Dies entspricht der folgenden Änderung am Pseudocode:



Compress(x, y, F)

Input: x und y, wobei y in F auf dem Pfad von x zur Wurzel liegen muss

```
1:  if rank(x) > s then
2:      Compress(x, y, F_+)
3:  else if rank(y) ≤ s then
4:      Compress(x, y, F_-)
5:  else
6:      z ← x
7:      while rank(parent(z)) ≤ s) do
8:          z ← parent(z)
9:      end while
10:     Compress(parent(z), y, F_+)
11:     Shatter(x, z, F_-)
12:     parent(z) ← z
13: end if
```

Es ist leicht zu sehen, dass in der modifizierten Version von Compress genau so viele Zeigeränderungen passieren, wie in der Originalversion. Falls nur innerhalb von F_+ oder nur innerhalb von F_- Zeigeränderungen vorkommen, dann passieren genau dieselben Zeigeränderungen auch in der modifizierten Version (Zeile 2 bzw. Zeile 4). Falls ein Compress-Aufruf einen Pfad komprimiert, der von einem Knoten in F_- bis zu einem Knoten in F_+ verläuft, dann finden die zugehörigen Zeigeränderungen in F_+ ebenso statt (Zeile 10). Für jede Zeigeränderung in F_- wird durch Shatter (Zeile 11) eine entsprechende Zeigeränderung ausgeführt. Es verbleiben die Zeigeränderungen, von Knoten in F_-, die einen Vater in F_+ haben. Diese Zeigeränderungen werden in Zeile 12 nachgeahmt.

Aufstellen der Rekursionsformel:

- jeder Knoten im Wald wird von höchstens einer Shatter-Operation berührt. Somit gibt es höchstens n Zeigeränderungen durch Shatter-Aufrufe.
- Jeder Zeigeränderung in Zeile 12 entspricht einer Zeigeränderung, die durch eine Compress-Operation in F_+ ausgelöst wurde. Somit gibt es genau m_+ viele solche Zeigeränderungen.
- Durch das Partitionieren des Walds F in F_+ und F_- haben wir somit auch die Folge C der Compress-Aufrufe in zwei Folgen C_+ und C_-, mit jeweils m_+ und m_- vielen Aufrufen, zerlegt.

Wenn wir alles zusammenfassen, dann gilt somit folgende Ungleichung:

$$(*) \qquad T(F, C) \leq T(F_+, C_+) + T(F_-, C_-) + m_+ + n.$$

Da es nur $\frac{n}{2^i}$ viele Knoten mit rank i gibt, gilt

$$n_+ \leq \sum_{i>s} \frac{n}{2^i} = \frac{n}{2^s}.$$

Außerdem ist die Anzahl der unterschiedlichen ranks in F_+ höchstens $r - s < r$. Mit Theorem 17 folgt nun, dass

$$T(F_+, C_+) < r \cdot \frac{n}{2^s}.$$

Wenn wir nun $s = \log r$ setzen, dann gilt $T(F_+, C_+) \leq n$ und somit können wir die Ungleichung $(*)$ zu folgender Ungleichung vereinfachen:

$$(**) \qquad T(F, C) \leq T(F_-, C_-) + m_+ + 2n.$$

Eine erste verbesserte obere Schranke: Wir können die Ungleichung $(**)$ für ein Zwischenresultat benutzen. Wenn wir für $T(F_-, C_-)$ wieder Theorem 17 aufrufen, dann erhalten wir

$$T(F_-, C_-) < ns = n \log r.$$

Da $r = \lfloor \log n \rfloor$, folgt mit $(**)$ unmittelbar, dass wir eine obere Schranke von

$$\mathcal{O}(m + n \log \log n)$$

für die Gesamtkosten $T(F, C)$ haben.

Eine noch bessere obere Schranke: Als nächstes formen wir $(**)$ äquivalent um und erhalten:

$$(***) \qquad T(F, C) - m \leq T(F_-, C_-) - m_- + 2n.$$

Wir erhalten eine viel bessere obere Schranke, wenn wir die Ungleichung $(***)$ einfach rekursiv immer wieder in sich selbst einsetzen. Unsere Analyse galt für jeden beliebigen Wald F und jede beliebige Operationenfolge C, somit liefert uns Ungleichung $(***)$ folgendes:

$$T'(m, n, r) \leq T'(m, n, \lfloor \log r \rfloor) + 2n,$$

wobei $T'(m,n,r) = T(m,n,r) - m$. Wir haben damit eine *Rekurrenz*, die wir nur noch lösen müssen. Dazu überlegen wir uns den Abbruchfall der Rekurrenz, d.h. die Kosten von $T'(m,n,1)$. Es gilt

$$T'(m,n,1) \leq n,$$

da, falls der größtmögliche rank 1 ist, jeder Knoten höchstens ein mal einen neuen Vater bekommen kann.
Die vollständige Rekurrenz lautet also:

$$T'(m,n,r) \leq \begin{cases} T'(m,n,\lfloor \log r \rfloor) + 2n, & \text{falls } r > 1 \\ n & \text{falls } r = 1. \end{cases}$$

Wir lösen die Rekurrenz per *iterativem Einsetzen*[58]. Dazu setzen wir einfach die Ungleichung mehrfach in sich selbst ein. Wir ignorieren dabei die Rundungen, da es damit nur noch länger dauert, bis wir beim Abbruchfall angekommen. Es gilt:

$$\begin{aligned} T'(m,n,r) &\leq T'(m,n,\log r) + 2n \\ &\leq T'(m,n,\log\log r) + 2n + 2n \\ &\leq T'(m,n,\log\log\log r) + 2n + 2n + 2n \\ &\dots \end{aligned}$$

Wir vermuten, dass wir nach dem j-ten Mal einsetzen folgendes erhalten:

$$T'(m,n,r) \leq T'(m,n,\overbrace{\log\log\dots\log}^{j \text{ mal}} r) + 2n \cdot j.$$

Dies können wir leicht per Induktion über j zeigen: Für $j = 1$ gilt die Aussage offensichtlich. Für den Induktionsschritt setzen wir in die Ungleichung

$$T'(m,n,r) \leq T'(m,n,\overbrace{\log\log\dots\log}^{k \text{ mal}} r) + 2n \cdot k,$$

für ein beliebiges $k \geq 1$ ein weiteres mal ein und erhalten

$$T'(m,n,r) \leq T'(m,n,\overbrace{\log\log\dots\log}^{k+1 \text{ mal}} r) + 2n \cdot (k+1).$$

Nun müssen wir uns nur noch überlegen, wie oft wir $T'(m,n,r)$ in sich selbst einsetzen müssen, um zum Abbruchfall $r = 1$ zu gelangen. Bei jedem Einsetzen wird der dritte Eintrag logarithmiert. D.h. die Anzahl der Einsetzungen entspricht genau der Anzahl der Logarithmierungen (und Abrundungen) von r bis ein Wert von 1 erreicht ist. Dies ist exakt $\log^* r$. Nach $\log^* r$-maligem Einsetzen haben wir:

$$T'(m,n,r) \leq T'(m,n,\lfloor \log^* r \rfloor) + 2\log^* r \cdot n = T'(m,n,1) + 2n\log^* r < 2n\log^* n.$$

Da $T'(m,n,r) = T(m,n,r) - m$, haben wir folgendes gezeigt:

Theorem 18. $T(m,n,r) \leq m + 2n\log^* n$.

Damit haben wir eine obere Schranke von $\mathcal{O}(m + n\log^* n)$ für die Gesamtkosten einer Folge von m Find- und $n-1$ Union-Aufrufen in einem Disjoint-Set-Forest mit n Knoten bewiesen.

[58] Wir behandeln diese Technik später noch ausführlicher.

Noch bessere obere Schranken: Man kann die obige Analyse noch an einigen Stellen verfeinern und damit eine obere Schranke in $\mathcal{O}(m + n\alpha(m, n))$ erhalten, wobei $\alpha(m, n)$ die *inverse Ackermann-Funktion* ist. Diese Funktion wächst noch deutlich langsamer als die ohne hin schon extrem langsam wachsende Funktion $\log^* n$. Man kann damit dann zeigen, dass für $n < 2^{66}$ gilt: $T(m, n, r) \leq \min\{m + 4n, 2m + 2n\}$.

Kruskal mit Disjoint-Set-Forests und Pfadkompression: Für Kruskal ergeben sich somit für $n < 2^{66}$ Gesamtkosten in $\mathcal{O}(sort(m) + m + n)$. D.h. für solch "kleine" n treffen wir die triviale untere Schranke, falls wir in $\mathcal{O}(m)$ die Kantenlängen sortieren können.

3.7 Exkurs: Schnelles Sortieren

Wir erinnern uns an folgende Aussage aus der letzten Vorlesung:

Kruskal mit Disjoint-Set-Forests und Pfadkompression: Für Kruskal ergeben sich für $n < 2^{66}$ Gesamtkosten in $\mathcal{O}(sort(m) + m + n)$. D.h. für solch "kleine" n treffen wir die triviale untere Schranke von $\Omega(m + n)$, falls wir die Kantenlängen in $\mathcal{O}(m)$ sortieren können.
Dies bringt uns direkt zur Frage, in welchem Fall wir die Kantenlängen tatsächlich in $\mathcal{O}(m)$ sortieren können. Bevor wir dazu kommen, zeigen wir zunächst, dass es kein vergleichsbasiertes Sortierverfahren gibt, welches im worst-case Linearzeit benötigt.

3.7.1 Untere Schranke für vergleichsbasiertes Sortieren

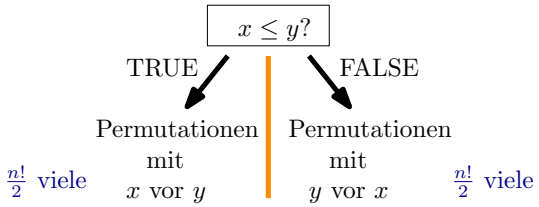
Die in der Vorlesung bisher erwähnten Sortierverfahren Mergesort und Heapsort haben im worst-case Kosten von $\mathcal{O}(n \log n)$. Wir zeigen nun, dass diese Kosten unvermeidlich sind, sofern wir keine weitere Information über die zu sortierenden Schlüssel haben und diese nur vergleichen können. Wir nehmen also an, dass wir für je zwei Schlüssel x und y die Frage $x \leq y$ beantworten können. Dabei ist es uns egal, ob x und y Zahlen oder Zeichenketten oder noch komplexere Objekte sind. Wichtig ist nur, dass wir einen *Komparator* haben, d.h. einen Algorithmus, der bei gegebenen x und y ausgibt, ob $x \leq y$ gilt oder nicht.

Ein *vergleichsbasiertes Sortierverfahren* ist somit jedes Sortierverfahren, welches die relative Position zweier Elemente zueinander nur mit Hilfe des Komparators ermittelt. Fast alle Standardsortierverfahren fallen in diese Kategorie.
Wir werden nun beweisen, dass *jedes* vergleichsbasierte Sortierverfahren $\Omega(n \log n)$ viele Anfragen an den Komparator ausführen *muss*, sofern es korrekt ist. D.h. wir argumentieren über Algorithmen, die wir eventuell gar nicht kennen, weil sie eventuell erst in ferner Zukunft erfunden werden!

Theorem 19. *Jeder korrekte vergleichsbasierte Sortieralgorithmus benötigt im worst-case $\Omega(n \log n)$ Vergleiche auf einer Instanz mit n paarweise unterschiedlichen Elementen.*

Beweis. Wir betrachten dazu einen *beliebigen* Sortieralgorithmus namens Sort. Die Eingabe für Sort ist eine Folge X von n beliebigen Elementen, die jeweils paarweise mit Hilfe eines Komparators verglichen werden können. Da Sort ein Sortierverfahren ist, muss es die Folge X in eine aufsteigend sortierte Folge X^*, welche dieselben Elemente wie X enthält, umwandeln. D.h. X^* ist eine *Permutation* von X.

Wir wissen nicht wie Sort funktioniert, aber, falls Sort korrekt sortiert, dann muss es für jede mögliche Eingabefolge X korrekt sortieren. Daraus folgt direkt, dass *jede* Permutation von X eine mögliche Ausgabe von Sort sein muss. Da wir annehmen, dass X genau n Elemente enthält und es $n!$ viele verschiedene Permutation von n Elementen gibt, muss Sort somit mindestens $n!$ viele mögliche unterschiedliche Ausgaben erzeugen können. Falls alle Elemente von X paarweise unterschiedlich sind, dann gibt es zu X unter all den $n!$ vielen Permutationen von X nur *genau eine*, die alle Elemente von X in aufsteigender Reihenfolge enthält. Sort muss somit mit Hilfe der Eingabefolge die zugehörige einzige korrekte Permutation finden. Um dies zu tun, muss Sort irgendwie zwischen unterschiedlichen Permutationen *unterscheiden* können. Und genau hier kommt der Komparator ins Spiel: Falls für zwei beliebige Elemente x und y aus X gilt, dass $x \leq y$, dann unterscheidet dieser Vergleich *alle* $\frac{n!}{2}$ vielen Ausgabepermutationen von X, in welchen x vor y vorkommt von *allen* $\frac{n!}{2}$ vielen Ausgabepermutationen in welchen x hinter y vorkommt[59]. Ein *beliebiger* Vergleich von Sort liefert uns somit folgendes Ergebnis bei der Suche nach der richtigen Permutation:

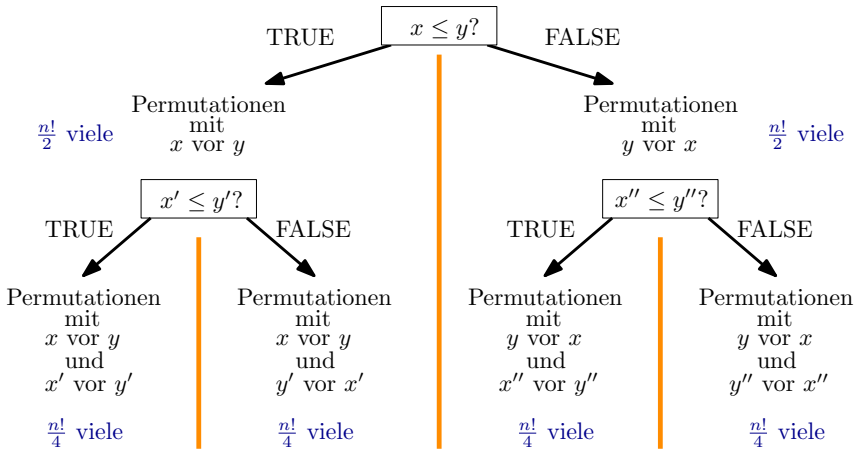


Nach einem Vergleich sind somit von den ursprünglich $n!$ vielen Ausgabepermutationen noch $\frac{n!}{2}$ viele möglich[60]. Um unter diesen die richtige Permutation zu finden, werden wieder zwei Elemente miteinander verglichen. Da wir keine unnötigen Vergleiche wollen, nehmen wir an, dass der nächste Vergleich von Sort nicht $x \leq y$ lautet. Angenommen der Vergleich ist $x' \leq y'$, wobei durchaus entweder $x = x'$ oder $y = y'$ gelten kann. Dieser Vergleich liefert uns neue Information für unsere Suche nach der richtigen Permutation: Unter allen $\frac{n!}{2}$ vielen Permutationen, die nach dem ersten Vergleich noch in Frage kommen, unterscheidet der Vergleich $x' \leq y'$ zwischen den $\frac{n!}{4}$ vielen Permutationen, in welchen x' vor y' vorkommt, und den $\frac{n!}{4}$ vielen Permutationen, in welchen x' nach y' vorkommt. Wir haben also schon wieder unseren Suchraum halbiert!
Generell gilt, dass jeder neue Vergleich, d.h. jeder noch nicht durchgeführte Vergleich von Sort, die Zahl der aktuell noch in Frage kommenden Permutationen höchstens halbiert[61].

[59] Hier benötigen wir die Annahme, dass alle Elemente paarweise unterschiedlich sind.

[60] In jeder Permutation kann man die Reihenfolge von x und y vertauschen und erhält somit zwei verschiedene Permutationen. Somit gibt es $\frac{n!}{2}$ viele Permutationen in welchen x vor y vorkommt.

[61] Es kann durchaus Vergleiche geben, die den Suchraum weniger stark einschränken - der in diesem Sinne bestmögliche Vergleich halbiert den Suchraum exakt.



D.h. nach i vielen (unterschiedlichen) Vergleichen hat Sort im besten Fall genug Information gesammelt, um die Zahl der Kandidaten für eine korrekte Permutation auf $\frac{n!}{2^i}$ viele zu reduzieren. Dies passiert genau dann, wenn jeder Vergleich den aktuellen Suchraum exakt halbiert.
Wir wissen ja bereits, dass es genau eine richtige Ausgabepermutation gibt. Also muss *jeder* korrekte Sortieralgorithmus mindestens so viele unterschiedliche Vergleiche tätigen, damit die Permutation eindeutig feststeht! D.h. wir müssen i so wählen, dass

$$\frac{n!}{2^i} = 1$$

gilt. Stellen wir nach i um, haben wir:

$$\frac{n!}{2^i} = 1 \iff n! = 2^i \iff \log(n!) = i.$$

Nun müssen wir nur noch verstehen, wie schnell $\log(n!)$ eigentlich wächst. D.h. wir suchen eine Funktion f so dass $\log(n!) \in \Theta(f)$ ist. Es gilt folgendes:

Lemma 12. $\log(n!) \in \Theta(n \log n)$.

Beweis. Es gilt

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} = \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \dots \cdot \frac{n}{2}}_{\frac{n}{2} \text{ mal}} < \underbrace{n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot \frac{n}{2}}_{\frac{n}{2} \text{ Terme, alle bis auf einen} > \frac{n}{2}} \cdot \left(\frac{n}{2} - 1\right) \cdot \dots \cdot 2 \cdot 1 = n!.$$

Außerdem gilt

$$n! = \underbrace{n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1}_{n \text{ Terme, alle bis auf einen} < n} < \underbrace{n \cdot n \cdot \dots \cdot n}_{n \text{ mal}} = n^n.$$

Somit haben wir

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} < n! < n^n.$$

Da der Logarithmus eine monoton wachsende Funktion ist, gilt:

$$\log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) < \log(n!) < \log\left(n^n\right).$$

Laut Logarithmengesetz gilt $\log(a^b) = b \cdot \log a$ für beliebige a und b. Somit haben wir

$$\frac{n}{2} \cdot \log\left(\frac{n}{2}\right) < \log(n!) < n \cdot \log\left(n\right).$$

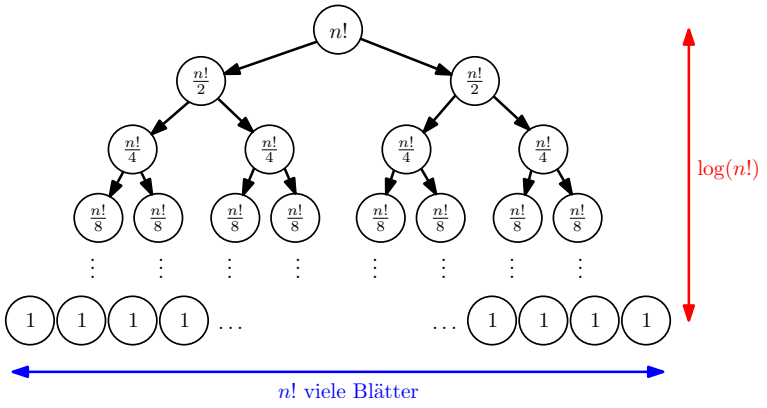
Aus der rechten Ungleichung folgt direkt, dass $\log(n!) \in \mathcal{O}(n \log n)$. Aus der linken Ungleichung folgt $\log(n!) \in \Omega(n \log n)$, da mit $\log(\frac{a}{b}) = \log a - \log b$ gilt, dass

$$\frac{n}{2} \log\left(\frac{n}{2}\right) = \frac{n}{2}(\log n - \log 2) = \frac{n}{2}(\log n - 1) = \frac{n}{2} \log n - \frac{n}{2}.$$

□

Damit haben wir bewiesen, dass *jeder* korrekte vergleichsbasierte Sortieralgorithmus $\Omega(n \log n)$ viele Vergleiche machen *muss*. □

Man kann den obigen Beweis auch noch anders veranschaulichen. Stellen wir uns vor, dass die Wurzel eines gerichteten Binärbaums die Menge aller $n!$ vielen Permutationen der Eingabefolge darstellt. Wird nun ein Vergleich ausgeführt, dann unterscheidet dieser Vergleich zwischen zwei Hälften des Suchraums. Die Wurzel hat somit zwei Kinder, die jeweils für eine der Hälften, d.h. je für $\frac{n!}{2}$ viele Permutationen, steht. Jeder weitere Vergleich erzeugt wieder zwei Kinder, deren zugehörigen Menge an Permutationen die Permutationsmenge des Vaterknotens aufteilt[62]. Die Blätter dieses Baumes stehen dann für Permutationsmengen, die nur noch genau eine Permutation enthalten und somit nicht mehr geteilt werden können.
Somit entsteht ein Binärbaum mit $n!$ vielen Blättern, den man üblicherweise *Entscheidungsbaum* nennt. Die Anzahl der Vergleiche des Sortierverfahrens im worst-case entspricht somit der Länge des längsten Pfades von der Wurzel zu einem Blatt. Wir können also *jedem* korrekten vergleichsbasierten Sortieralgorithmus einen Entscheidungsbaum zuordnen und ein Durchlauf des Algorithmus entspricht dann einem Pfad von der Wurzel zu einem Blatt in diesem Entscheidungsbaum.



[62]Es ist hier durchaus möglich, dass der Vergleich die aktuelle Permutationsmenge nicht genau halbiert.

Die Länge des längsten Pfades in einem beliebigen Entscheidungsbaumes ist genau dann so klein wie möglich, wenn das tiefste Blatt so nah wie möglich an der Wurzel ist, d.h. wenn der Entscheidungungsbaum ein *vollständiger balancierter Binärbaum* ist. In einem solchen Baum hat jeder innere Knoten genau zwei Kinder und alle Blätter haben dieselbe Tiefe. Ein solcher Entscheidungsbaum mit $n!$ vielen Blättern hat Tiefe $\log(n!)$.

3.7.2 Untere Schranke für vergleichsbasiertes Sortieren: Average-case

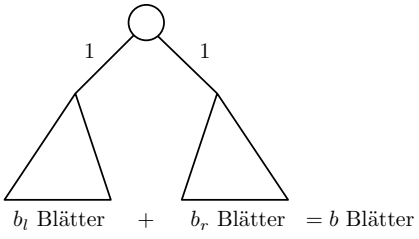
Wir haben eben eine untere Schranke für *vergleichsbasierte Sortierverfahren* bewiesen. Wir haben gezeigt, dass es keinen Sortieralgorithmus geben kann, welcher mit Hilfe von Elementvergleichen worst-case Gesamtkosten in $o(n \log n)$ hat. D.h. jeder vergleichsbasierte Sortieralgorithmus hat im schlimmsten Fall Kosten in $\Omega(n \log n)$ und somit sind z.B. Mergesort und Heapsort im worst-case optimal.
Es gibt allerdings vergleichsbasierte Sortierverfahren, die im worst-case Kosten in $\mathcal{O}(n \log n)$ und im best-case Kosten in $\mathcal{O}(n)$ haben. Z.B. hat Timsort, das Standardsortierverfahren in Python und Java 7[63], diese Eigenschaft.
Da stellt sich sofort die Frage, wie sich solche Sortierverfahren eigentlich im *average-case* verhält, d.h. was sind die durchschnittlichen Kosten solcher Sortierverfahren. Hierbei wird der Durchschnitt über alle möglichen Permutationen der zu sortierenden Folge gebildet. Oder etwas allgemeiner:

> Gibt es vergleichsbasierte Sortierverfahren, die im average-case Kosten in $o(n \log n)$ haben?

Wir werden zeigen, dass dies unmöglich ist. Der Grund hierfür ist in den sogenannten Entscheidungsbäumen, die wir schon bei der unteren Schranke für den worst-case angetroffen haben, zu finden. Wir müssen eigentlich nur zeigen, dass ein Blatt des Entscheidungsbaumes (d.h. eine ermittelte Permutation der Eingabefolge) im Durchschnitt sehr weit unten im Baum liegt.
Betrachten wir also die Durchschnittliche „Tiefe“ eines Blattes. Dazu zeigen wir, dass für *jeden* Binärbaum Folgendes gilt:

Lemma 13. *Ein Binärbaum mit b Blättern hat eine durchschnittliche Blatttiefe $\geq \log b$.*

Beweis. Per Widerspruch:
Wir wählen das kleinste $b \geq 2$, so dass die Behauptung nicht gilt und betrachten die Wurzel und die Teilbäume:



[63] Timsort wird in Java 7 als Standardsortierverfahren für nicht-primitive Datentypen eingesetzt. Für primitive Datentypen wird Dual-Pivot-Quicksort genutzt.

Nach Wahl von b, gilt die Behauptung nun aber für die Teilbäume, denn diese müssen echt weniger Blätter haben. Somit gilt für die durchschnittliche Blatttiefe Folgendes:

$$\begin{aligned} &= \frac{b_l}{b}(1+\log b_l) + \frac{b_r}{b}(1+\log b_r) \\ &= \frac{1}{b}\big(b_l(1+\log b_l) + b_r(1+\log b_r)\big) \\ &= \frac{1}{b}\big(b_l \cdot \log(2b_l) + b_r \cdot \log(2b_r)\big) \end{aligned}$$

Das Minimum dieser Funktion liegt bei $b_l = b_r = \frac{b}{2}$ und somit gilt:

$$= \frac{1}{b}\left(\frac{b}{2}\log b + \frac{b}{2}\log b\right) = \log b.$$

Und wir haben den gewünschten Widerspruch erzeugt. □

Wir haben somit auch eine untere Schranke für die average-case Kosten von vergleichsbasierten Sortierverfahren.

Korollar 20. *Jedes vergleichsbasierte Sortierverfahren hat im average-case Kosten in* $\Omega(n \log n)$.

3.7.3 Knacken der unteren Schranke

Wir betrachten nun, wie wir die untere Schranke für allgemeine Sortierverfahren "umgehen" können. Zur Erinnerung: Im Beweis der unteren Schranke sind wir davon ausgegangen, dass wir zwei Schlüssel nur mit Hilfe eines Komparators vergleichen können, d.h. wir haben insbesondere angenommen, dass wir nichts über die Schlüssel an sich wissen.
Wir betrachten nun zwei Algorithmen, die in Linearzeit sortieren, *falls die Schlüssel Zahlen aus einem gewissen Intervall sind.* Wir haben hier somit starkes Zusatzwissen über unsere Schlüssel und dies hilft uns, um schneller zu sortieren.

Countingsort Eines der einfachsten Sortierverfahren überhaupt ist Countingsort. Die Idee ist, sich für jeden Schlüssel einfach einen Zähler anzulegen, dann durch alle Schlüssel zu iterieren und immer den entsprechenden Zähler inkrementieren. Zum Schluss rekonstruieren wir aus den Zählerständen das korrekt sortierte Array.

Countingsort(Array A)

Input: Array A von n ganzen Zahlen im Bereich 0 bis z.
Output: aufsteigend sortiertes Array A.

```
1: Initialisiere Array B der Länge z + 1, welches überall auf 0 gesetzt ist
2: n ← |A|; Initialisiere Array C der Länge n
3: for i = 0 to n − 1 do
4:     B[A[i]] ← B[A[i]] + 1
5: end for
6: B[0] ← B[0] − 1
7: for j = 1 to z do
8:     B[j] ← B[j] + B[j − 1]
9: end for
10: for i = n − 1 downto 0 do
11:     C[B[A[i]]] ← A[i]
12:     B[A[i]] ← B[A[i]] − 1
13: end for
14: return C
```

Wichtig ist hierbei, dass wir vor dem Sortieren wissen, dass alle zu sortierenden Schlüssel ganze Zahlen aus dem Bereich $[0, z]$ sind. Es ist leicht zu sehen, dass Countingsort Kosten von $\mathcal{O}(n + z)$ hat.
Selbstverständlich können wir das Verfahren leicht abwandeln, so dass wir auch ganze Zahlen im Bereich $[-z, z]$ sortieren können. Es folgt, dass Countingsort dann in Linearzeit sortiert, wenn das Intervall, aus dem die ganzzahligen Schlüssel stammen, nur $\mathcal{O}(n)$ viele Elemente enthält[64].

Radixsort Radixsort ist ein weiterer Sortieralgorithmus, mit dessen Hilfe wir sogar für polynomiell große Zahlenintervalle in Linearzeit sortieren können. Die wesentliche Änderung von Radixsort zu Countingsort liegt darin, dass wir nun annehmen, dass wir auf die *einzelnen Stellen* der Zahlen zugreifen können.
Betrachten wir die einfachste Version von Radixsort - die in manchen Lehrbüchern auch *Sortieren durch Fachverteilung* genannt wird.

Radixsort(Array A)

Input: Array A von ganzen Dezimalzahlen mit jeweils k Stellen.
Output: Aufsteigend sortiertes Array A.

```
1: {die k-te Stelle ist die Stelle ganz rechts}
2: for j := k downto 1 do
3:     A ← A nach Stelle j mit Countingsort sortiert
4: end for
5: return A
```

[64]Mit einem Scan über die Eingabe können wir uns ggf. die Intervallgrenzen beschaffen, falls wir diese vorher nicht kennen.

Vom Prinzip her, könnten wir in Zeile 3 auch ein anderes Sortierverfahren als Countingsort verwenden. Countingsort bietet sich hier allerdings an, da wir ja genau wissen, aus welchem Bereich die einzelnen Stellen einer Zahl kommen und da dieser Bereich sehr klein ist. Wenn wir Dezimalzahlen sortieren, dann gibt es nur die Werte von 0 bis 9 pro Stelle. Somit läuft Countingsort hier offensichtlich in $\mathcal{O}(n)$ pro Durchlauf der `for`-Schleife.
Jede Stelle der Zahlen sorgt für einen Durchlauf von Countingsort. Wenn wir also k-stellige Zahlen haben, dann entstehen Kosten in $\mathcal{O}(k \cdot n)$. In vielen praktischen Anwendungen ist k eine Konstante - z.B. wenn viele Postleitzahlen (alle 5-stellig) oder 64-bit Zahlen sortiert werden sollen.
Doch wie verhält es sich mit dem zu sortierenden Intervall? Wenn z.B. unsere Zahlen aus dem Bereich $[0, z]$ kommen, dann kann es Zahlen geben, die $\log_{10} n$ viele Stellen haben. Somit würden wir mit Radixsort dann Kosten von $\Theta(n \log n)$ haben. Wir sind also in diesem Fall sogar schlechter als Countingsort!

Radixsort - verbessert Wir können das obige Problem vermeiden. Genauergesagt, können wir Radixsort deutlich verbessern und damit dann ganze Zahlen im Intervall $[0, n^c]$ für jedes konstante c in Linearzeit sortieren. Dies ist überaus bemerkenswert - schließlich vergrößern wir im Vergleich zu Countingsort die Intervallgröße von linear auf ein beliebiges Polynom in n. Selbstverständlich können wir damit dann auch ganze Zahlen im Intervall $[-n^c, n^c]$ in Linearzeit sortieren.
Hier nun der Trick, wie wir Radixsort verbessern können:
Jede Dezimalzahl x mit $x \leq n^c - 1$ hat eine Länge von höchstens $c \cdot \log_{10} n$. Wenn wir führende Nullen einführen, dann können wir davon ausgehen, dass alle Schlüssel in A genau $c \cdot \log_{10} n$ Stellen haben. Der Trick besteht jetzt darin, dass man Stellen zusammenfassen kann, um weniger als $c \cdot \log_{10} n$ Runden mit `Radixsort` zu brauchen.
Die Kernidee ist die Folgende: Wenn man r Stellen zu einem "Bit" zusammenfasst, dann vergrößert man den sogenannten *Radix* auf 10^r und somit muss das Zählarray für Countingsort dann von 0 bis $10^r - 1$ laufen. Dafür muss man jetzt aber im Verlauf von Radixsort nur noch

$$\frac{c \cdot \log_{10} n}{r}$$

viele Runden von Countingsort machen.
Wir haben uns oben überlegt, dass Countingsort eine Laufzeit in $\mathcal{O}(n + z)$ hat, wenn das Zählarray Elemente von 0 bis z zählt. Hier haben wir somit eine Laufzeit von $\mathcal{O}(n + 10^r)$ für einen Durchlauf mit Countingsort.
Zusammen ergibt das eine Laufzeit in

$$\mathcal{O}\left(\frac{c \cdot \log_{10} n}{r}\left(n + 10^r\right)\right).$$

Wählen wir nun $r = \log_{10} n$, dann benötigt `Radixsort` nur $\mathcal{O}(c)$ Runden und jeder Aufruf von `Countingsort` läuft in $\mathcal{O}(n + 10^{\log_{10} n}) = \mathcal{O}(n)$ Schritten. Wir erhalten somit Gesamtkosten in $\mathcal{O}(c \cdot n)$.

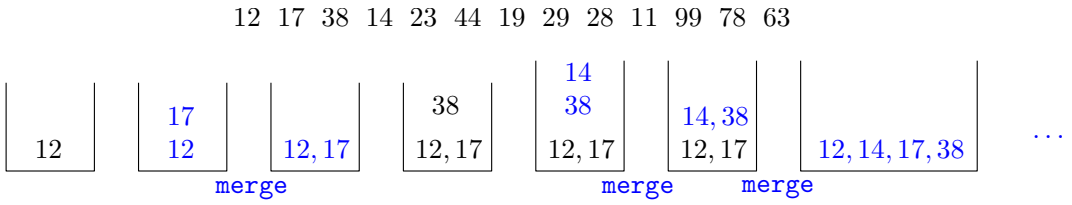
3.7.4 State-of-the-Art Sortieren: Timsort

Wir machen noch einen kleinen Ausflug in die Praxis. D.h. wir betrachten ein Sortierverfahren, welches momentan State-of-the-Art ist. Dabei sehen wir ein schönes Beispiel für

Algorithm Engineering, d.h. für das clevere Tuning von existierenden Algorithmen.
Wir werden Timsort etwas näher betrachten. Timsort ist das Standardsortierverfahren von Python und Java 7.[65]
Timsort, genannt nach seinem Erfinder Tim Peters, ist vom Prinzip her nur ein getuntes Mergesort, welches für kleine Teilarrays jedoch Insertionsort nutzt.
Timsort verbessert Mergesort in mehreren Punkten:

- Es ist adaptiv, d.h. Timsort nutzt vorsortierte Bereiche der Eingabe aus.
- Timsort versucht die Zahl der merge-Operationen zu minimieren.
- Gleichzeitig versucht Timsort jedes merge möglichst günstig zu realisieren.
- Die Tiefe des Rekursionsbaums bei Timsort ist geringer, da der Rekursionsabbruch früher erreicht wird.

Vom Grundprinzip her, arbeitet Timsort wie ein von "innen nach außen gestülptes" Mergesort. D.h. es beginnt direkt mit dem Verschmelzen von Teilarrays und arbeitet den Rekursionsbaum somit von unten nach oben ab. Diese Variante von Mergesort wird auch oft als bottom-up Mergesort bezeichnet. Dabei wird von links nach rechts über die Eingabe gelaufen und es werden immer möglichst gleichgroße benachbarte Teilarrays miteinander verschmolzen. Die einzelnen Teilarrays, die noch auf ihrere Verschmelzung warten, werden dabei in einem `Stack` zwischengespeichert. Hier ein Beispiel:



Tatsächlich müssen wir uns auf dem `Stack` nicht die einzelnen Teilarrays merken, eigentlich genügt die Startposition der Teilarrays und deren Länge. Für das Ausführen der merge-Operationen benötigen wir trotzdem Zusatzspeicher, nämlich genau die Länge eines der beiden zu verschmelzenden Teilarrays. Allerdings ist auch leicht zu sehen, dass der Stack nie mehr als $\mathcal{O}(\log n)$ viele Teilarrays enthält. Der worst-case wäre ein Stack der ein Teilarray der Länge 2^k, eines der Länge 2^{k-1}, eines der Länge 2^{k-2}, usw. bis zur Länge 2^0 enthält. Im schlimmsten Fall sind alle Elemente der Eingabe beteiligt und somit muss $\sum_{i=0}^{k} 2^i \leq n$ gelten. Damit folgt, dass

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \leq n \iff k + 1 \leq \log(n+1)$$

und somit $k \in \mathcal{O}(\log n)$.
Vom Prinzip her, haben wir aber noch keine Verbesserung gegenüber Mergesort erzielt, wir haben eigentlich nur den Algorithmus von einem rekursiven Algorithmus in einen iterativen Algorithmus umgewandelt.
Ausgehend von diesem bottom-up Mergesort werden wir nun einige Verbesserungen hinzufügen.

[65] Bei Java 7 wird Timsort nur für nicht-primitive Datentypen verwendet, für primitive Datentypen ist Dual Pivot Quicksort der Standard.

Füherer Rekursionsabbruch Wir wollen die Tiefe des Rekursionsbaumes (bzw. die Höhe des Stacks) etwas verringern. Die rekursiven Aufrufe bei Mergesort sorgen im Prinzip nur dafür, dass wir bereits sortierte Teilarrays erhalten, die wir dann miteinander verschmelzen können. Dies bringt uns auf zwei Ideen:

1. Wir könnten nach vorhandenen sortierten Teilarrays suchen und diese direkt benutzen.

2. Wir könnten uns mit einem anderen Verfahren die sortierten Teilarrays erzeugen - es zwingt uns ja niemand dazu, dass wir Mergesort bis zur Arraygröße 1 benutzen müssen.

Suche nach vorhandenen sortierten Teilarrays: Wir bezeichnen im Folgenden ein Teilarray $A[i], A[i+1], \ldots, A[i+k]$ als `run`, falls $A[i] \leq A[i+1] \leq \cdots \leq A[i+k]$ und $k \geq 1$ gilt. Unser Beispiel von oben enthält die folgenden nichtüberlappenden `runs`:

12 17 38 14 23 44 19 29 28 11 99 78 63
run 1 run 2 run 3 run 4

Im schlimmsten Fall kann es passieren, dass überhaupt keine `runs` vorhanden sind, z.b. bei einer absteigend sortierten Eingabe. In diesem Fall gibt es dann aber etwas ähnliches, nämlich "absteigende runs", d.h. Teilarrays $A[i], A[i+1], \ldots, A[i+k]$ für die $A[i] > A[i+1] > \cdots > A[i+k]$ gilt. Für unser Beispiel haben wir:

12 17 38 14 23 44 19 29 28 11 99 78 63
run 1 run 2 abst. run 1 abst. run 2

Unser Ziel ist es, in der Eingabe möglichst viele und jeweils möglichst lange runs zu finden. Denn für diese Teilarrays müssen wir nichts mehr tun - wir müssen sie nur noch verschmelzen.
Hier nun eine der Kernideen von Timsort: Wir suchen gleichzeitig nach auf- und absteigenden `runs`. Haben wir einen absteigenden `run` gefunden, dann drehen wir diesen einfach in-place um und haben somit einen aufsteigenden `run`. Für das "Umdrehen" laufen wir einfach mit zwei Zeigern von beiden Enden in die Mitte und Vertauschen die Elemente. Wir bezahlen somit im worst-case etwa doppelt so viel, wie ein einfacher Scan über die Eingabe kosten würde. Dieser kleine Mehraufwand lohnt sich aber langfristig, da wir weniger rekursive Aufrufe bekommen. Für unser Beispiel passiert Folgendes:

12 17 38 14 23 44 19 29 28 11 99 78 63
run 1 run 2 abst. run 1 abst. run 2

12 17 38 14 23 44 19 11 28 29 63 78 99
run 1 run 2 run 3 run 4

12 17 38 14 23 44 19 11 28 29 63 78 99
run 1 run 2 run 3

Hierbei haben wir im letzten Schritt noch getestet, ob benachbarte `runs` verschmolzen werden können. Auch dies ist nicht teuer und kann direkt nach dem "Umdrehen" mit getestet werden.

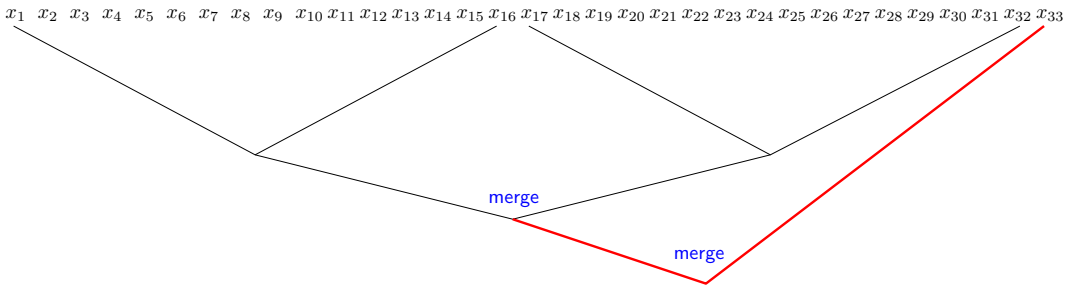
Dieser Schritt ermittelt somit `runs`, die natürlicherweise in der Eingabe vorkommen. Falls die Eingabe bereits vorsortiert ist, dann finden wir viele recht lange `runs`. Bei perfekt zufälligen Eingaben, sind allerdings alle `runs` sehr kurz und wir müssen uns etwas anderes einfallen lassen.

Erzeugung langer `runs` zu Beginn: Wir erhalten natürlich auch einen `run`, wenn wir ein Teilarray $A[i], \ldots, A[i+k]$ einfach sortieren! Sollten in der Eingabe zu kurze `runs` vorliegen, dann können wir uns diese mit diesem Trick recht einfach verlängern.
Bei Timsort wird mit diesem Trick dafür gesorgt, dass alle `runs` mindestens eine Länge zwischen 32 und 64 haben. Diese Länge nennen wir absofort *minrun*. Kleinere Teilarrays (die evtl. zu kurze `runs` enthalten) werden mit Hilfe von `Insertionsort` sortiert und somit zu einem langen `run` umgewandelt. Timsort auf Eingaben mit weniger als 64 Elementen ist sogar identisch zu Insertionsort. Die Wahl von Insertionsort ist hier kein Zufall: Insertionsort ist adaptiv, d.h. es ist umso schneller je vorsortierter das Teilarray ist.[66]

Minimieren der Anzahl der merge-Aufrufe Die Wahl der Länge der kürzest möglichen `runs` hängt eng mit der Anzahl der merge-Aufrufe zusammen. Timsort versucht hier generell wenige merge-Aufrufe zu erzeugen und wählt deshalb *minrun* wie folgt:

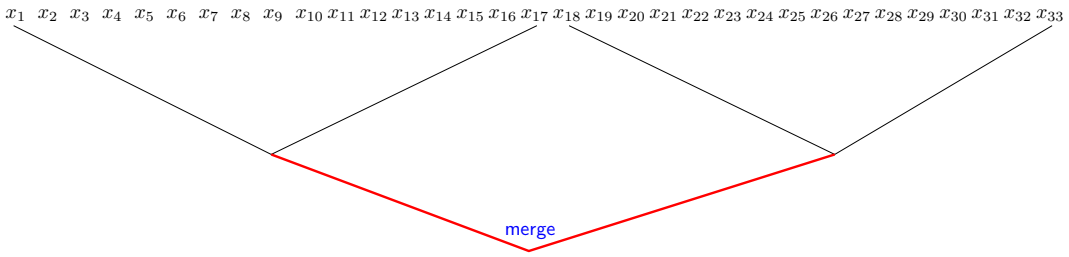
- Falls $n \leq 64$, dann gilt $\mathit{minrun} = n$.
- Sonst, $\mathit{minrun} \in (32, 65)$, so dass n/minrun sehr nah an einer Zweierpotenz, jedoch kleiner oder gleich als diese ist.

Letzteres ist recht leicht zu erreichen: *minrun* ist einfach die Zahl, die von den 6 most significant bits von n codiert wird, wobei noch eine 1 addiert wird, falls mindestens ein weiteres bit auf 1 gesetzt ist.
Falls z.B. $n = 2^6 = 64 = 100000_2$, dann ist $\mathit{minrun} = n$ und daran sehen wir, dass dann Timsort identisch zu Insertionsort ist. Gilt z.B. $n = 5000 = 1001110001000_2$, dann wird $\mathit{minrun} = 100111_2 + 1_2 = 101000_2 = 40$ gewählt. Es gilt $5000/40 = 125$ und das ist tatsächlich recht nah an der nächsten Zweierpotenz $128 = 2^7$.
Der Grund, warum *minrun* so gewählt wird, ist auch recht leicht zu sehen. Beim merge werden immer in etwa gleich große Teilarrays verschmolzen (mehr dazu später). Nehmen wir einfach mal an, dass $\mathit{minrun} = 16 = 2^4$ ist und $n = 33 = 2^5 + 1$. Dann wäre n/minrun etwas größer als eine Zweierpotenz. Wir hätten damit folgende merge-Aufrufe beim bottom-up Mergesort:

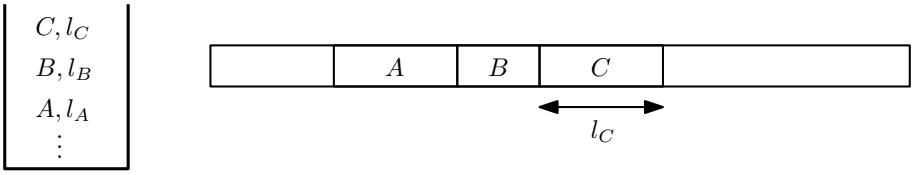


[66] Siehe Aufgabe 4a von Übungsblatt 07.

Wir sehen, dass der letzte merge-Aufruf zwei Teilarrays mit sehr unterschiedlicher Länge verschmilzt. Genau dies will Timsort vermeiden. Wählen wir für unser Beispiel *minrun* analog zu oben, nur diesmal mit den ersten 5 bits von $n = 33 = 100001_2$, dann erhalten wir $minrun = 10001_2 = 17$ und somit ein merge-Aufruf mit fast gleich großen Arrays:



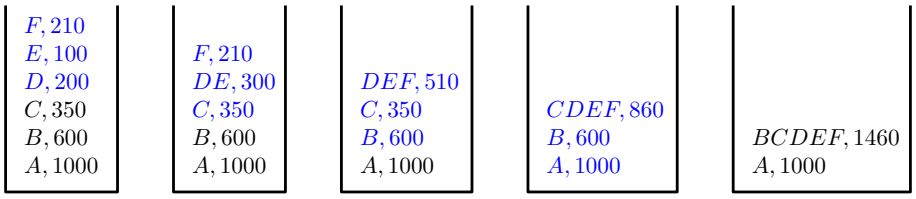
Die Verschmelzungsstrategie von Timsort Nun sehen wir, wie tatsächlich verschmolzen wird. Timsort geht ähnlich wie im oben erwähnten bottom-up Mergesort vor, jedoch haben wir bereits gesehen, dass statt bei Länge 2 bei Länge *minrun* begonnen wird. Außerdem werden bei Timsort nur *benachbarte* `runs` verschmolzen, die in etwa die selbe Länge haben. Es wird auch ein `Stack` benutzt, um bereits verschmolzene `runs` oder solche, die noch auf Verschmelzung warten, zu verwalten. Im `Stack` befinden sich dabei von unten nach oben die (evtl. verschmolzenen) `runs` von links nach rechts in der Eingabe. Wird ein neuer `run` betrachtet, dann wird dieser auf den `Stack` gelegt und dessen Länge wird mit den obersten beiden Einträgen im `Stack` verglichen und je nach deren Länge wird verschmolzen. Es passiert Folgendes:
Angenommen die oberen drei `runs` auf dem Stack sind A, B und C und seien l_A, l_B und l_C die Längen der `runs`.



Timsort testet nun, ob folgende Eigenschaften erfüllt sind:

1. $l_A > l_B + l_C$
2. $l_B > l_C$.

Falls ja, dann passiert nichts. Falls Eigenschaft 1 verletzt ist, d.h. falls $l_A \leq l_B + l_C$, dann wird der kleinere der beiden `runs` A bzw. C mit B verschmolzen (falls $l_A = l_C$, dann wird C gewählt) und das Ergebnis unter bzw. über den verbleibenden `run` auf den `Stack` gelegt. Danach wird wieder getestet ob beide obigen Eigenschaften erfüllt sind. Falls Eigenschaft 2 verletzt ist, dann werden B und C verschmolzen und das Ergebnis zurück auf den `Stack` gelegt.
Dieser Prozess wird so lange wiederholt, bis beide Eigenschaften erfüllt sind (oder weniger als 3 Elemente auf dem `Stack` liegen).
Hier ein Beispiel:



Die beiden Eigenschaften haben ihren Grund. Eigenschaft 2 führt dazu, dass die `runs` auf dem `Stack` von unten nach oben betrachtet immer absteigende Länge haben. Eigenschaft 1 führt dazu, dass die Längen der `runs` auf dem `Stack`, von oben nach unten gelesen, mindestens so schnell wachsen, wie die Fibonaccizahlen[67]. Dies hilft uns, denn für die i-te Fibonacci-Zahl $F(i)$ gibt es eine geschlossene Formel:[68]

$$F(i) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - \left(\frac{1-\sqrt{5}}{2} \right)^i \right).$$

Der hintere Term in der Klammer ist für jedes $i \geq 1$ größer als -1, somit erhalten wir folgende untere Schranke an die i-te Fibonacci-Zahl:

$$F(i) \geq \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - 1 \right).$$

Wir sehen also, dass die Anzahl der Elemente, die in allen auf dem `Stack` liegenden `runs` enthalten sind, exponentiell mit der Stackhöhe wächst. Da wir nur insgesamt n Elemente haben, bedeutet dies im Umkehrschluss, dass der Stack somit zu jeder Zeit nur logarithmische Höhe in n haben kann. Dies hätten wir auch erreicht, wenn wir, wie bei bottom-up Mergesort nur Teilarrays gleicher Länge verschmolzen hätten - doch die obige Version ist etwas flexibler und spart somit ein paar Verschmelzungen, die entstehen würden, um Zweierpotenzlängen zu erzeugen.

Verschmelzung zweier `runs` Als letztes betrachten wir noch, wie Timsort zwei `runs` miteinander verschmilzt. Erstaunlicherweise kann man hierbei auch noch optimieren. Es gilt zwar, dass merge von Mergesort optimal ist, doch diese Aussage gilt nur im worst-case[69]. Wir können tatsächlich das merge noch verbessern, damit es für einige Fälle schneller wird.
Um etwas Intuition zu gewinnen, betrachten wir das folgende Beispiel:

1 2 3 4 5 17 19 21 28 7 8 9 10 11 12 13 14 15

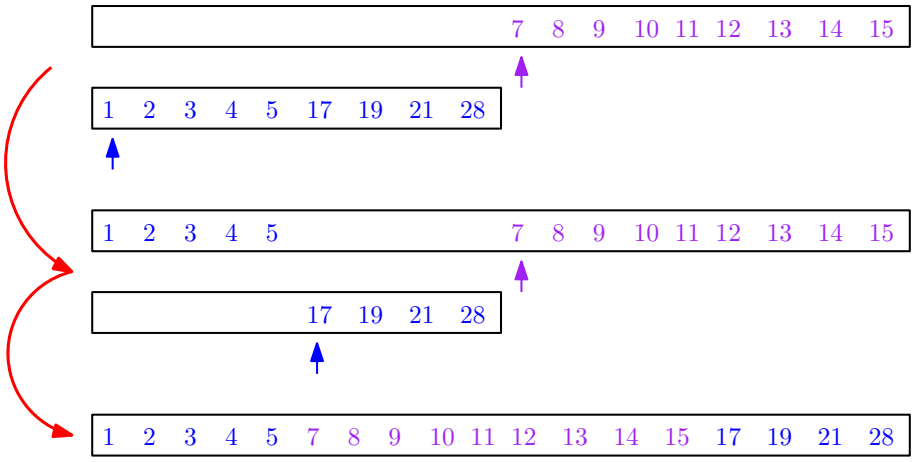
run 1 run 2

[67] Zur Erinnerung: Die Folge der Fibonaccizahlen lautet:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Diese werden wie folgt gebildet: $F_1 = 1$ und $F_2 = 1$, $F_{i+1} = F_i + F_{i-1}$ für $i \geq 2$.
[68] Die Formel von Moivre und Binet.
[69] Für den Beweis betrachtet man eine "Reißverschlussinstanz".

Zunächst benötigen wir etwas Zusatzspeicher. Genauer: Wir brauchen so viel Zusatzspeicher, wie der kürzere der beiden `runs` lang ist. Dann kopieren wir diesen dahin und laufen dann mit zwei Zeigern durch die Teilarrays und kopieren die Elemente an die richtigen Stellen im Ausgangsarray.



An diesem Beispiel sehen wir, dass wir uns sehr viele Vergleiche hätten sparen können, wenn wir vorher gewusst hätten, dass z.B. das erste Element des rechten `runs` größer als viele der Anfangselemente des linken `runs` ist. Wüssten wir, dass das kleinste Element des rechten `runs` nach der Verschmelzung an Position 6 im linken `run` stehen muss und wüssten wir, dass rechteste Element im rechten `run` kleiner ist als 4 Elemente aus dem linken `run`, dann könnten wir in diesem Beispiel sogar ohne Vergleiche das Verschmelzen durchführen! Nun müssen wir uns nur noch überlegen, wie wir schnell die Position der Randelemente der `runs` im jeweiligen anderen `run` ermitteln können.
Dazu nutzen wir die sogenannte *Exponentialsuche*[70]. Wir testen, ob ein Randelement an Position 1 gehört, falls nicht, dann testen wir Position 2, dann testen wir Position 4, danach Position 8, usw. bis wir ein Element im anderen `run` gefunden haben, welches größer als unser betrachtetes Element ist. In jedem Durchlauf verdoppeln wir somit die Sprungweite. Im Beispiel würde folgendes passieren:

Haben wir ein solches Element an Positon 2^k gefunden, dann wissen wir, dass die richtige Position für unser "einzufügendes Element" zwischen Position $2^{k-1}+1$ und 2^k liegen muss. In diesem Bereich wenden wir dann einfach binäre Suche an, um die Position zu finden.
Was kostet uns das? Da wir die Sprungweite immer verdoppeln kostet uns die Exponentialsuche in einem `run` der Länge l höchstens $\log l$ viele Vergleiche. Die angehängte binäre Suche arbeitet im schlimmsten Fall auf einem Bereich, dass $\frac{l}{2}$ groß ist und benötigt somit höchstens $\log \frac{l}{2} + 1 = \log l$ viele Vergleiche. Insgesamt sind das höchstens $2 \log l$ viele.

[70] In der Beschreibung von Timsort wird dies "galloping mode" genannt.

So betrachtet, hätten wir auch gleich binäre Suche verwenden können und hätten nur $\log l + 1$ viele Vergleiche. Im schlimmsten Fall stimmt das, allerdings kann es sehr oft passieren, dass die gesuchte Position relativ weit am Anfang liegt, d.h. das k ist sehr klein. In diesen häufigen Fällen sparen wir somit sehr viel gegenüber der binären Suche, die immer $\log l + 1$ viele Vergleiche benötigt!
Timsort wägt nun ab, ob sich der Zusatzaufwand lohnt oder nicht. Genauer gilt, dass das Standardvorgehen ohne Suche dann zu bevorzugen ist, wenn die Position des ersten Elements des einen `runs` unter den ersten 6 Positionen im zweiten `run` liegt. Ist diese weiter hinten, dann würde die Version mit Suche tatsächlich Vergleiche einsparen. Deshalb setzt Timsort hier einen Schwellwert ein. Der Algorithmus führt zunächst 8 normale Vergleiche aus, falls die Position des betrachteten Elements damit nicht ermittelt wurde, dann wird auf die Version mit Suche umgeschalten.

4 Kürzeste Pfade und Priority Queues

Wir werden sehen, dass eine leichte Modifikation des MST-Algorithmus Jarník/Prim genutzt werden kann, um die kürzesten Pfade von einem Startknoten zu allen anderen Knoten in einem positiv gewichteten, gerichteten Graph zu berechnen.

4.1 Berechnung kürzester Wege

Das oben beschriebene Problem heißt üblicherweise *single source shortest path problem* und ist wie folgt definiert:

> **Gegeben:** Gerichteter gewichteter Graph $G = (V, E, l)$ mit n Knoten, alle Kantengewichte sind strikt positiv[71]. Startknoten s.
>
> **Aufgabe:** Bestimme jeweils einen kürzesten Pfad von s zu allen anderen Knoten von G.

Um das Problem zu lösen, müssen wir zunächst etwas über kürzeste Wege nachdenken. Wir zeigen zunächst, dass Teilpfade von kürzesten Pfaden auch kürzeste Pfade sein müssen. Allgemein nennt man diese Eigenschaft *optimal substructure property*, d.h. eine optimale Lösung (hier ein kürzester Pfad) besteht aus optimalen Teillösungen (hier Teilpfade).[72]

Lemma 14. *Sei $P = v_1, v_2, \ldots, v_k$ ein kürzester Weg von Knoten v_1 zu v_k in einem Graph $G = (V, E, l)$ mit positiven Kantenlängen, dann ist jeder Teilweg von P auch selbst ein kürzester Weg in G.*

Beweis. Wir beweisen die Aussage per Widerspruch. Angenommen es gibt zwei Knoten v_i und v_j, mit $1 \leq i < j \leq k$, auf dem Pfad P zwischen denen der Pfad $v_i, v_{i+1}, \ldots, v_{j-1}, v_j$ nicht der kürzeste Weg ist. Dann muss es aber einen echt kürzeren Pfad $Q = v_i, x_1, x_2, \ldots, v_j$ zwischen v_i und v_j geben. Wenn wir nun den Pfad P so ändern, dass zwischen v_i und

[71] Achtung, wir werden später sehen, dass wir diese Bedingung tatsächlich für unseren Algorithmus benötigen. Selbstverständlich gibt es auch Algorithmen für das single source shortest path problem in Graphen mit beliebigen Kantengewichten.

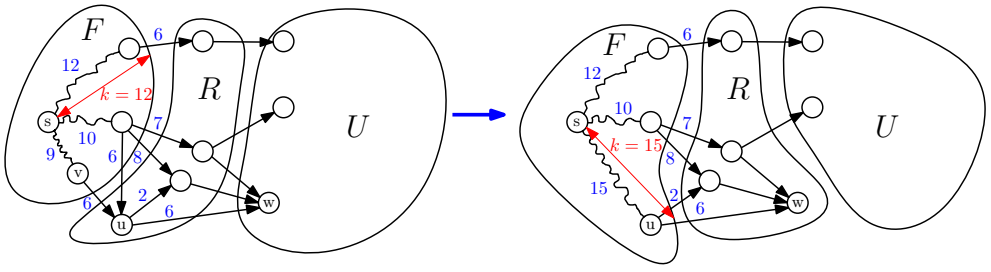
[72] Diese Eigenschaft werden wir später bei der dynamischen Programmierung wieder antreffen.

v_j der Pfad Q benutzt wird, dann, da alle Kantenlängen positiv sind, erhalten wir damit einen Pfad zwischen v_1 und v_k, der kürzer als P ist. Dies ist ein Widerspruch zur Annahme, dass P ein kürzester Weg zwischen v_1 und v_k ist. □

Wenn wir das obige Lemma nun auf alle kürzesten Wege mit Startknoten s anwenden, dann liefert uns dies die entscheidende Idee für einen Algorithmus:

> Der kürzeste Weg zwischen s und u muss aus einem kürzesten Weg zwischen s und irgendeinem Nachbarn v von u bestehen und dann folgt noch die Kante von v zu u. Dabei muss der Weg von s zu v echt kürzer sein, als der Weg von s zu u.

Wir können somit alle kürzesten Wege von s zu weit entfernten Knoten nach und nach aus anderen kürzesten Wegen von s zu näheren Knoten "aufbauen". Dieser Prozess beginnt mit dem kürzesten Weg von s zum Knoten mit geringster Distanz zu s, nämlich der Nachbar von s, der über die kürzeste von s ausgehende Kante erreicht werden kann.
Im Prinzip gehen wir wie folgt vor: Wir haben eine Menge F von Knoten, zu denen wir bereits einen kürzesten Weg von s aus gefunden haben (anfangs ist nur s in dieser Menge). Die Knoten in F haben die Eigenschaft, dass diese alle höchstens Distanz k zu s haben (anfangs ist $k = 0$).
Außerdem haben wir eine Menge R von Knoten, die eine eingehende Kante von mindestens einem Knoten aus F besitzen, welche aber selbst nicht zu F gehören. D.h. die Knoten in R sind diejenigen, die zu Knoten aus F benachbart sind, aber selbst mindestens Distanz k zu s haben.
Und wir haben eine Menge U von allen anderen Knoten, die weder zu F noch zu R gehören. Wir bezeichnen die Menge R als "Rand" der Menge F. Es folgt, dass die Mengen F, R und U eine Partition der Menge V bilden, d.h. jeder Knoten von G kommt in genau einer der drei Mengen vor.



Aus dem obigen Lemma folgt nun, dass der zu s näheste Knoten u in R einen kürzesten Weg zu s haben muss, welcher bis auf den letzten Knoten nur aus Knoten aus F besteht[73]. Auf diesem kürzesten Weg von s zu u, der bis zum vorletzten Knoten nur aus Knoten von F besteht, sei v der letzte Knoten, der in F liegt.
Nun entfernen wir u aus R, fügen ihn zu F hinzu und wir merken uns auch den Knoten v, um später den kürzesten Pfad wieder rekonstruieren zu können. Danach müssen wir R aktualisieren, da u ja ausgehende Kanten zu Knoten aus U haben könnte, die danach von U nach R verschoben werden müssen. Außerdem könnte u auch ausgehende Kanten zu anderen Knoten aus R haben, wodurch sich deren Entfernung zum nächsten Knoten aus F ändern könnte.

[73] Falls das nicht so wäre, dann gäbe es noch einen anderen Knoten in R der geringere Distanz zu s hätte.

4.1.1 Der Algorithmus von Dijkstra

Der Algorithmus, der mit dieser Idee von Startknoten s die kürzesten Wegen zu allen anderen Knoten berechnet, wird meist nach seinem Erfinder Edsger W. Dijkstra benannt. Bei genauem Hinsehen, entspricht der Algorithmus fast exakt dem Algorithmus Jarník/Prim, wenn dieser mit einer Priority Queue(PQ)[74] implementiert wird. Der einzige Unterschied besteht darin, wie die Prioitäten für die PQ gewählt werden und dass ein gerichteter Graph betrachtet wird. Dijkstras Algorithmus berechnet statt einem Minimum Spanning Tree einen sogenannten *Shortest Path Tree (SPT)*, d.h. einen Baum mit Wurzel s, welcher einen kürzesten Pfad von s zu allen anderen Knoten in G enthält. Für ein Element u in der PQ sei $P(u)$ die Priorität. Insert(u, x) fügt Element u mit Priorität x in die PQ ein und DecreaseKey(u, x) ändert die Priorität von u auf den Wert x. ExtractMin entfernt das Element mit dem kleinsten Prioritätswert aus der PQ und liefert es zurück.
Hier die Algorithmen Dijkstra und Jarník/Prim im Vergleich:

Dijkstra(G, s)

Input: G mit positiven Kantenlängen als Adjazenzliste A, Knoten s
Output: Shortest Path Tree für Knoten s

```
 1: initialisiere n-Array K überall mit ∞
 2: initialisiere n-Array B überall mit False
 3: initialisiere leere Priority Queue Q
 4: initialisiere leeren Baum F
 5: B[u] ← True
 6: for alle w mit (s,w) ∈ E do
 7:    Insert(s, ℓ((s,w))); K[w] ← s
 8: end for
 9: while Q nicht leer do
10:    u ← ExtractMin()
11:    B[u] ← True
12:    füge Kante (K[u],u) in F ein
13:    for alle w mit (u,w) ∈ E do
14:       if w ∈ Q then
15:          if P(w) > P(u)+l((u,w)) then
16:             DecreaseKey(w,P(u)+l((u,w)))
17:             K[w] ← u
18:          end if
19:       else
20:          if B[w] = False then
21:             Insert(w, P(u)+l((u,w)))
22:          end if
23:       end if
24:    end for
25: end while
26: return F
```

Jarník/Prim(G, s)

Input: G als Adjazenzliste A, Knoten s
Output: MST von G

```
 1: initialisiere n-Array K überall mit ∞
 2: initialisiere n-Array B überall mit False
 3: initialisiere leere Priority Queue Q
 4: initialisiere leeren Baum F
 5: B[s] ← True
 6: for alle w mit (s,w) ∈ E do
 7:    Insert(s, ℓ({s,w})); K[w] ← s
 8: end for
 9: while Q nicht leer do
10:    u ← ExtractMin()
11:    B[u] ← True
12:    füge Kante (K[u],u) in F ein
13:    for alle w mit {u,w} ∈ E do
14:       if w ∈ Q then
15:          if P(w) > l({u,w}) then
16:             DecreaseKey(w,l({u,w}))
17:             K[w] ← u
18:          end if
19:       else
20:          if B[w] = False then
21:             Insert(w, l({u,w}))
22:          end if
23:       end if
24:    end for
25: end while
26: return F
```

Bei beiden Algorithmen wird im Array K der Vorgängerknoten des entsprechenden Knotens gespeichert, damit in F die korrekte Kante eingefügt wird.

[74] Zur Erinnerung: Eine Priority Queue ist eine Datenstruktur, die die Operationen Insert, DecreaseKey und ExtractMin unterstützt.

Dijkstra in der obigen Form berechnet "nur" den Shortest Path Tree von Knoten s. Wir zeigen in der Übung, wie daraus in $\mathcal{O}(n)$ die Längen der Pfade ermittelt werden können.

Korrektheit des Algorithmus von Dijkstra:

Theorem 21. *Der Algorithmus Dijkstra ist korrekt.*

Beweis. Wir beweisen die folgende Invariante des Algorithmus von Dijkstra, aus welcher dann direkt die Korrekheit folgt.

> Invariante: Wenn Knoten x aus der Priority Queue (PQ) entfernt wird, dann wurde die Länge des kürzesten Pfades vom Startknoten s zu Knoten x korrekt bestimmt. Außerdem entspricht die Priorität der Knoten in der PQ der Länge eines Pfades zu s welcher nur Zwischenknoten benutzt, die bereits aus der PQ entfernt wurden.

Wir beweisen die Invariante per Induktion über den Zeitpunkt des Entfernens aus PQ.
IA: Für den ersten Knoten u, der aus PQ entfernt wird, gelten beide Aussagen, da u der näheste Knoten zu s ist und jeder andere Pfad von s zu u über Zwischenknoten länger sein muss, da alle Kantenlängen positiv sind.
IS: Sei u der i-te Knoten, der aus der PQ entfernt wurde und sei v der Vorgänger von u (d.h. $K[u] = v$). Knoten v muss somit vorher schon aus der PQ entfernt worden sein. Als u entfernt wurde, hatte u die kleinste Priorität in PQ. Nach IV und weil v Vorgänger von u ist, gilt $P(u) = P(v) + \ell((v,u)) = d_G(s,v) + \ell((v,u))$, wobei $d_G(x,y)$ die Länge eines kürzesten Pfades zwischen x und y in G ist.
Wir zeigen nun per Widerspruch, dass $P(u) = d_G(s,u)$ gelten muss. Wir nehmen also an, dass es einen Pfad Q_{su} gibt, der kürzer als $P(u)$ ist. Es gibt zwei Fälle:

- Falls Q_{su} nur Zwischenknoten benutzt, die bereits aus der PQ entfernt wurden, dann betrachte den Vorgänger w von u auf Pfad Q_{su}. Mit Lemma 14 folgt $w \neq v$. Da w schon aus der PQ entfernt wurde und da $d_G(s,w) + \ell((w,u)) < d_G(s,v) + \ell((v,u))$, müsste w der Vorgänger von u sein. Widerspruch!

- Falls Q_{su} auch andere Zwischenknoten benutzt, dann betrachte den ersten Knoten y auf in Q_{su}, der noch in der PQ ist. Sei x der Nachbar von y auf Q_{su}, der nicht mehr in der PQ ist.

  Da Q_{su} kürzer ist als $d_G(s,v) + \ell((v,u))$ und $d_G(y,u) > 0$, weil alle Kantenlängen positiv sind, folgt, dass $d_G(s,x) + \ell((x,y)) < d_G(s,v) + \ell((v,u))$ und somit hat y eine kleinere Priorität als u. Widerspruch!

Nach Entfernen von u gilt auch der zweite Teil der Invariante, da nur die Prioritäten der Nachbarn von u in der PQ aktualisiert werden. Sollte dabei eine Priorität gesenkt werden, dann verläuft der zugehörige Pfad über u und hat somit nur Zwischenknoten die bereits aus der PQ entfernt wurden. □

4.1.2 Kosten der Algorithmen Dijkstra und Jarník/Prim

Beide Algorithmen sind bis auf die Wahl der Prioritäten identisch. Es genügt also, wenn wir Dijkstra betrachten.
Dijkstra hat einen initialisierungsaufwand von $\mathcal{O}(n)$. Danach wird jeder Knoten des Graphen genau einmal in die PQ eingefügt und genau einmal per ExtractMin entfernt. Bei jedem ExtractMin werden im worst-case alle Nachbarn des entfernten Knotens abgearbeitet und für jeden Nachbarn fällt ein mal eine DecreaseKey Operation an.
Somit haben wir folgende Gesamtkosten:

$$\mathcal{O}(n) + n \cdot cost(\mathsf{Insert}) + n \cdot cost(\mathsf{ExtractMin}) + m \cdot cost(\mathsf{DecreaseKey}).$$

Bei Jarník/Prim fallen im worst-case $2m$ viele DecreaseKey Operationen an, da jede Kante von beiden Endpunkten her betrachtet werden kann.
Wir sehen hier, dass die Gesamtkosten von den Kosten der DecreaseKey-Operationen dominert werden. Für eine super effiziente Implementierung sollten wir somit eine PQ wählen, die ein besonders effizientes DecreaseKey verwendet. Bevor wir dazu kommen, betrachten wir zunächst den Standardkandidaten für eine PQ-Implementierung.

Kosten von Dijkstra und Jarník/Prim mit binären Min-Heaps: Bisher haben wir als mögliche PQ-Implementierung nur binäre Min-Heaps kennengelernt. Bei dieser Datenstruktur haben alle drei Operationen Insert, ExtractMin und DecreaseKey kosten in $\Theta(\log n)$, falls n Elemente im Heap sind. Für Dijkstra und Jarník/Prim ergeben sich damit folgende Gesamtkosten: $\mathcal{O}(n \log n + m \log n)$. Für zusammenhängende Graphen sind dies Kosten in $\mathcal{O}(m \log n)$.

4.2 Exkurs: State-of-the-Art Priority-Queues

Es gibt eine Vielzahl von möglichen Implementierungen für Priority Queues. Solche Datenstrukturen gehören zu den Standardwerkzeugen eines jeden Algorithmikers und finden in vielen Bereichen Anwendung.

Binäre Min-Heaps: Eine der einfachsten PQ-Implementierung stellen die binären Min-Heaps dar. Wie oben schon erwähnt, realisieren diese die drei Grundoperationen einer PQ jeweils in $\Theta(\log n)$. Binäre Min-Heaps haben viele Vorteile: Sie sind sehr einfach zu implementieren, benötigen für manche Anwendungen nur konstant viel Zusatzspeicher und können direkt als (dynamisches) Array gespeichert werden. Doch sie haben auch Nachteile: Einerseits werden wir sehen, dass es bezüglich der Standardoperationen Insert und DecreaseKey effizientere PQ-Implementierungen gibt, andererseits gibt es noch andere wünschenswerte Operationen, die binäre Min-Heaps nur schlecht unterstützen.

Fibonacci Heaps: Diese PQ-Implementierung ist eine der ersten, die Insert mit worst-case Kosten $\mathcal{O}(1)$, DecreaseKey mit amortisierten Kosten von $\mathcal{O}(1)$ und ExtractMin mit amortisierten Kosten von $\mathcal{O}(\log n)$ realisiert. Sie wurde 1984 von Fredman & Tarjan vorgeschlagen und ist bis heute in vielen Lehrbüchern zu finden. Implementiert man Dijkstra oder Jarník/Prim mit Fibonacci-Heaps, dann ergeben sich Gesamtkosten von $\mathcal{O}(m+n \log n)$ und dies ist für Dijkstra bis heute die beste bekannte Laufzeitschranke. Für

das Finden eines MSTs haben wir ja bereits gesehen, dass Kruskal mit Disjoint-Set-Forests mit Pfadkompression noch etwas effizienter ist.

Hollow Heaps: Diese Heaps sind eine hochaktuelle PQ-Implementierung, die 2015 von Hansen, Kaplan, Tarjan & Zwick vorgeschlagen wurde. Hollow Heaps sind einfacher und eleganter als Fibonacci Heaps und auch die Analyse ist etwas leichter. Außerdem sind Hollow Heaps noch etwas effizienter, da diese Insert und DecreaseKey in worst-case Kosten von $\mathcal{O}(1)$ und ExtractMin mit amortisierten Kosten von $\mathcal{O}(\log n)$ realisieren[75].

Siehe Foliensatz zum Thema Hollow-Heaps!

5 Suchen in dynamischen Daten

In dieser Vorlesung werden wir auf Datenstrukturen treffen, die uns helfen, effizient mit sich ändernden, d.h. dynamischen, Daten umzugehen. Dies ist ein essentieller Bestandteil einer Vielzahl von Anwendungen.
Unser Ziel ist es, Datenstrukturen bereitzustellen, die die Operationen Einfügen, Löschen und Suchen von Schlüsseln realisieren. Solche Datenstrukturen nennt man auch *Wörterbücher* (Dictionary).
Offensichtlich ist, dass wir nur dann ein Datum schnell suchen können, wenn unsere Daten "aufgeräumt" sind. Will man beispielsweise in einem unsortierten Array der Länge n einen speziellen Schlüssel suchen, so benötigt *jeder* deterministische Algorithmus dafür $\Omega(n)$ viele Vergleiche. Dies kann man leicht mit einem Gegnerargument beweisen. Somit werden wir uns in diesem Kapitel nur mit Suchproblemen beschäftigen, welche auf Datenstrukturen arbeiten, die wir für das Suchen vorbereitet haben.
Falls sich die Daten nicht ändern, dann können wir die Daten einfach sortieren und dann Standardsuchverfahren, wie z.B. binäre Suche, verwenden. Allerdings ist die Annahme, dass sich die zu durchsuchenden Daten nicht ändern, in vielen Anwendungen unrealistisch. In der Praxis werden Schlüssel nicht nur gesucht (Search), sondern auch aus der zu durchsuchenden Menge von Schlüsseln entfernt (Delete) oder hinzugefügt (Insert) und wir wollen in diesen sich dynamisch verändernden Daten trotzdem schnell suchen können[76].
Damit z.B. binäre Suche funktioniert, muss das gegebene Array immer korrekt sortiert sein. Falls sich auch nur ein Wert in einem Array der Länge n ändert, kann dies dazu führen, dass mit Kosten von $\Theta(n)$ das Array zunächst erst wieder "aufgeräumt" werden muss. Das ist leicht zu sehen: Wird beispielsweise der größte Schlüssel im Array so

[75] Im Vergleich zu Fibonacci Heaps werden also die Kosten für DecreaseKey von amortisiert $\mathcal{O}(1)$ zu worst-case $\mathcal{O}(1)$ verbessert. Es gib allerdings Modifikationen von Fibonacci Heaps, die das auch schaffen.
[76] Das Ändern von Schlüsseln kann man leicht mit Delete gefolgt von Insert simulieren.

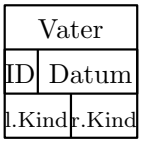
geändert, dass dieser kleiner als der kleinste Schlüssel ist, dann müssen alle Werte im Array auf einen anderen Platz getauscht werden[77]. Gleiches gilt für die Operationen Insert und Delete. Dort müsste zunächst nach der richtigen Stelle für das Einfügen bzw. nach dem zu löschenden Element gesucht und zusätzlich evtl. alle Elemente getauscht werden. Wenn wir also ein sortiertes Array verwenden um ein Dictionary zu implementieren, dann haben wir die folgenden Kosten:

	sortiertes Array
Search	$\Theta(\log n)$
Insert	$\Theta(n)$
Delete	$\Theta(n)$

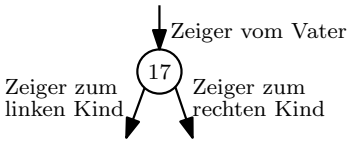
Unser Ziel ist es, eine Datenstruktur zu finden, die alle Operationen mit Kosten in $\mathcal{O}(\log n)$ realisiert. D.h. in dieser Datenstruktur wollen wir schnell suchen und bei Änderungen wollen wir schnell wieder die richtige Struktur herstellen können. Auf den ersten Blick scheinen sich diese beiden Anforderungen zu widersprechen - doch solche Datenstrukturen gibt es tatsächlich.

5.1 Binäre Suchbäume

Auf dem Weg zu einer geeigneten Datenstruktur beginnen wir mit einer sehr einfachen Idee: Wir verwenden Binärbäume, um unsere Schlüssel zu speichern. Wir benutzen sehr spezielle Binärbäume, sogenannte *binäre Suchbäume*. Ähnlich wie bei einem Min/Max-Heap speichern wir die Schlüssel in Knotenobjekten, wobei jeder Knoten im Binärbaum höchstens zwei Kinder hat - ein linkes Kind und ein rechtes Kind. Jeder Knoten enthält hierbei drei Zeiger, einen Zeiger auf den Vater, einen auf das linke Kind und einen auf das rechte Kind. Gibt es keinen Vater bzw. kein linkes oder rechtes Kind, so steht an der Stelle ein `null`-Zeiger. In unseren Abbildungen zeichnen wir nicht alle Zeiger, sondern nur die Zeiger, die vom Vater zu seinen Kindern zeigen.



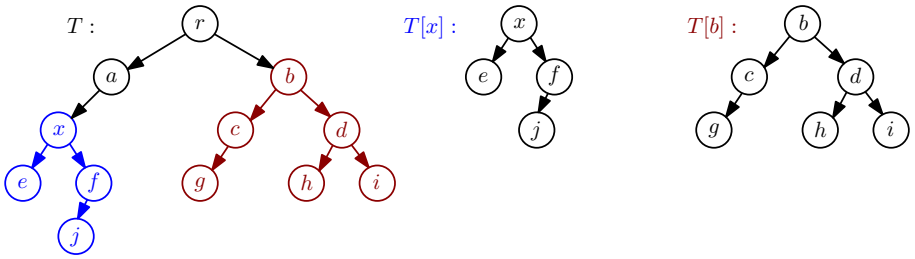
Knotenobjekt



Kurzdarstellung, hier Datum = 17

Im Gegensatz zu Min/Max-Heaps fordern wir allerdings (vorerst) keine Struktureigenschaft, jedoch eine stärkere Eigenschaft an die Schlüsselwerte von Vaterknoten und deren Kindern. Um diese Eigenschaft klar zu definieren, benötigen wir den Begriff des *Teilbaums* eines Knotens.
Sei T ein gerichteter Binärbaum und sei x ein Knoten von T. Der Teilbaum $T[x]$ von x ist der gerichtete Binärbaum der x als Wurzel hat, wenn im Baum T die Verbindung zwischen x und seinem Vater durchtrennt wird. Somit ist z.B. $T = T[r]$, wobei r die Wurzel von T ist. Ein weiteres Beispiel ist hier zu sehen:

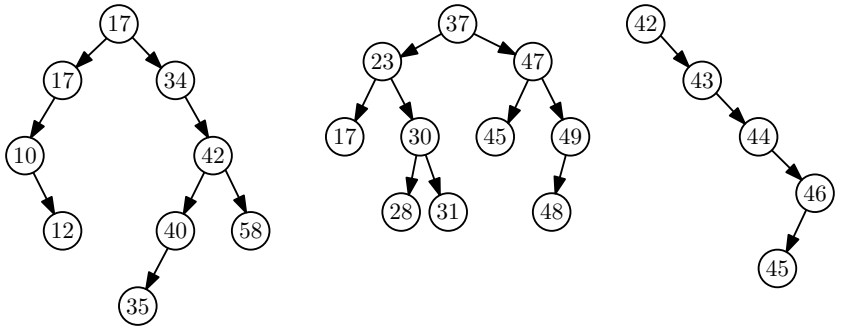
[77] Ein neues Array erstellen oder alle Werte ändern ist ebenso teuer.



Nun sind wir bereit eine wichtige Eigenschaft zu definieren: Ein gerichteter Binärbaum T hat die *Suchbaumeigenschaft*, wenn für *jeden* Knoten x in T das Folgende gilt:

- Der Schlüsselwert von x ist mindestens so groß wie jeder Schlüsselwert der Knoten im Teilbaum $T[l]$, wobei l das linke Kind von x in T ist.
- Der Schlüsselwert von x ist kleiner als jeder Schlüsselwert der Knoten im Teilbaum $T[r]$, wobei r das rechte Kind von x in T ist.[78]

Wir bezeichnen jeden gerichteten Binärbaum mit der Suchbaumeigenschaft als *binären Suchbaum*. Hier einige Beispiele für binäre Suchbäume:



Man beachte, dass wir bei binären Suchbäumen keine spezielle Struktur fordern. D.h. diese können sogar zu einem Pfad "entarten", wie oben rechts zu sehen ist. Auch müssen die einzelnen Level nicht vollständig sein.
Wie bereits bei den verketteten Listen und bei Heaps gesehen, können wir auf einen gerichteten Binärbaum nicht beliebig zugreifen, sondern nur über dessen Wurzel. D.h. wenn ein Baum T gegeben ist, dann haben wir eigentlich nur einen Zeiger auf das Wurzelobjekt von T.

5.1.1 Suchen in binären Suchbäumen

Zunächst beschäftigen wir uns mit der Operation Search, d.h. der Suche in einem binären Suchbaum.
Die Idee ist einfach: Wir vergleichen den Suchschlüssel x mit dem Schlüssel der Wurzel von T und biegen dann, je nach Ergebnis des Vergleichs in den Teilbaum des richtigen Kindes der Wurzel ab und vergleichen den Suchschlüssel mit dem Schlüssel der Wurzel dieses Teilbaums usw. Das ganze wiederholen wir so lange, bis wir entweder ein Vorkommen des Schlüssels x entdeckt haben, oder bis wir in einen leeren Teilbaum abbiegen müssten,

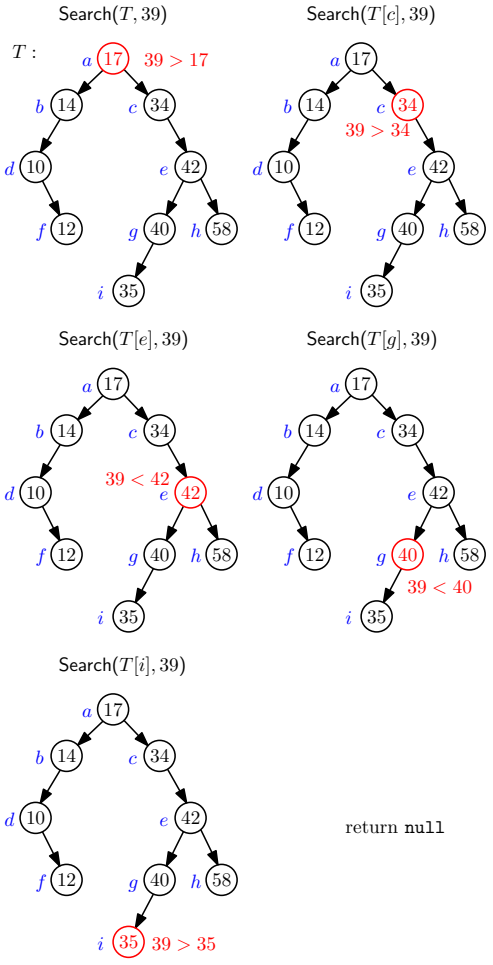
[78] Man könnte auch fordern, dass die Schlüssel, die gleich x sind, im rechten Teilbaum liegen. Wir einigen uns hier aber darauf, dass gleiche Schlüssel immer im linken Teilbaum vorkommen müssen.

d.h. zu einem Kind, welches nicht vorhanden ist (d.h. ein Kind-Zeiger ist `null`). Im ersten Fall geben wir einen Zeiger auf das entsprechende Knotenobjekt zurück, im letzteren Fall geben wir `null` zurück.
Es ist leicht zu sehen, dass jede Suche einen Pfad von der Wurzel des binären Suchbaums bis zu einem Knoten abläuft, wobei für jeden Knoten auf dem Pfad konstante Kosten anfallen. Um die Kosten von Search in einem binären Suchbaum mit n Knoten zu ermitteln, müssen wir uns nur noch überlegen, welche Länge der Suchpfad im schlimmsten Fall haben kann. Dies ist leicht zu sehen: Der Suchpfad kann im worst-case alle n Knoten enthalten, d.h. im worst-case kostet Search $\Theta(\text{Höhe}(T)) = \Theta(n)$. Dies ist sehr ungünstig, doch wir werden das später noch beheben. Das Problem liegt hier darin, dass binäre Suchbäume eine beliebige Form haben dürfen und die Kosten von der Höhe des Suchbaums abhängen - später werden wir noch eine zusätzliche Einschränkung der Form einführen, um die Höhe der Suchbäume auf $\mathcal{O}(\log n)$ beschränken.
Hier der Suchalgorithmus als Pseudocode:

Search(T, x)

Input: Binärer Suchbaum T, Schlüssel x

```
 1: w ← Wurzel von T
 2: if w ≠ null then
 3:    rootkey ← Schlüssel von w
 4:    if x = rootkey then
 5:       return w
 6:    else if x < rootkey then
 7:       l ← linkes Kind von w
 8:       if l ≠ null then
 9:          Search(T[l], x)
10:       else
11:          return null
12:       end if
13:    else
14:       r ← rechtes Kind von w
15:       if r ≠ null then
16:          Search(T[r], x)
17:       else
18:          return null
19:       end if
20:    end if
21: else
22:    return null
23: end if
```



Auch leicht zu sehen ist, dass Search im best-case für eine erfolgreiche Suche (d.h. der Suchschlüssel ist vorhanden) nur konstante Kosten hat. Der best-case für eine erfolglose Suche ist $\Theta(\log n)$, falls der binäre Suchbaum ein vollständiger Binärbaum mit Suchbaumeigenschaft ist. Dies zeigt uns schon, dass wir nicht auf weniger als logarithmische Kosten hoffen können, auch wenn wir die Struktur noch einschränken.

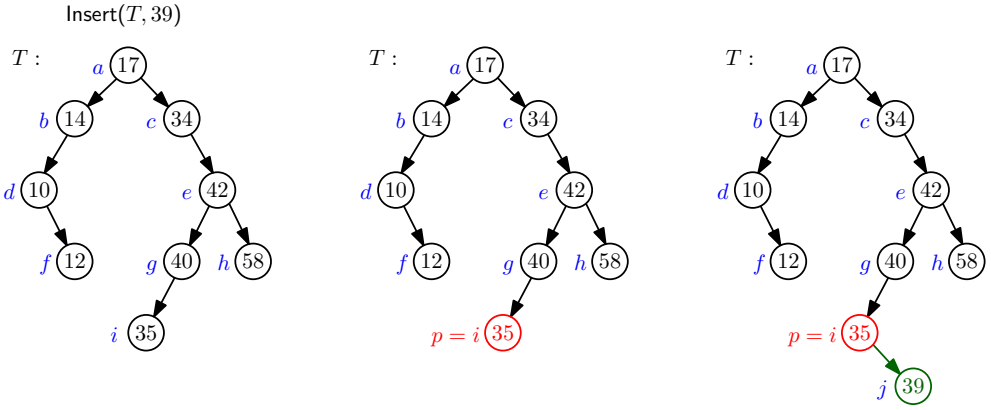
5.1.2 Einfügen in binäre Suchbäume

Wir wollen das folgende Problem lösen:

> **Gegeben:** Zeiger auf Wurzel eines binären Suchbaums T, Schlüssel x.
>
> **Aufgabe:** Füge x in den Suchbaum T ein, falls dieser noch nicht in T vorkommt. Falls x bereits in T vorkommt, dann soll nichts passieren[79].

Das Einfügen eines Schlüssels x in einen binären Suchbaum T kann ganz einfach realisiert werden:

1. Suche nach x in T. Falls x gefunden wird, dann Ende.
2. Falls x bei der Suche nicht gefunden wurde, dann muss die Suche bei einem `null`-Zeiger geendet haben. Sei p der Knoten, von dem der `null`-Zeiger ausgeht. Erzeuge ein neues Knotenobjekt mit Schlüssel x, Vater p und je zwei `null`-Zeigern für die beiden Kinder und füge ihn exakt an der Stelle des `null`-Zeigers von p, an dem die Suche abgebrochen ist, ein.



Das Einfügen ist korrekt, da durch die Suche der (einzige) richtige Platz für den neuen Schlüssel ermittelt wird.
Auch die Kosten sind leicht zu analysieren: Die Suche nach x kostet $\Theta(\text{Höhe}(T)) = \Theta(n)$ und das Erzeugen des neuen Knotenobjekts und das setzen der konstant vielen Zeiger von p zum neuen Kind, vom neuen Kind zu p und die beiden `null`-Zeiger des neuen Kindes hat konstante Kosten. Somit kostet uns das Einfügen eines Schlüssels in einen binären Suchbaum mit n Knoten im worst-case $\Theta(n)$. Auch hier sehen wir wieder, dass die Höhe von T eine entscheidende Rolle spielt. Hätte T nur eine Höhe von $\mathcal{O}(\log n)$, dann würde das Einfügen nur $\mathcal{O}(\log n)$ kosten.

5.1.3 Löschen in binären Suchbäumen

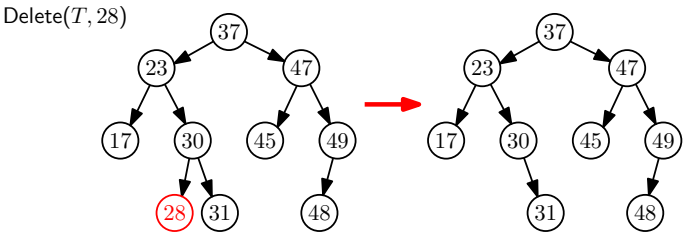
Wir wollen das folgende Problem lösen:

> **Gegeben:** Zeiger auf die Wurzel eines binären Suchbaums T, Schlüssel x.
>
> **Aufgabe:** Falls x im Suchbaum T vorkommt, dann entferne x aus T. Falls x nicht in T vorkommt, dann soll nichts passieren.

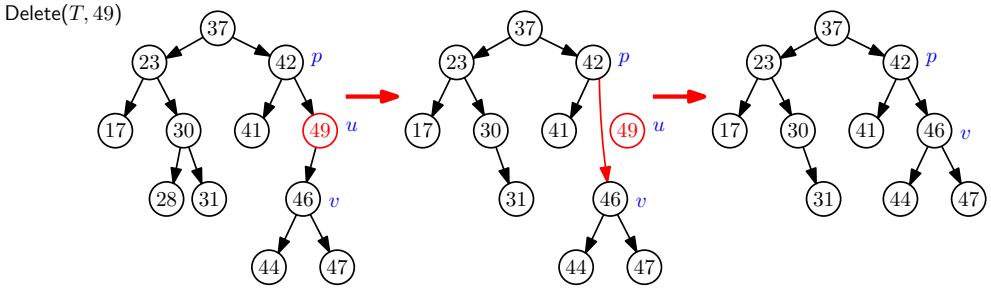
[79] Will man Duplikate im Baum zulassen, dann muss der vorgestellte Algorithmus nur leicht abgewandelt werden.

Auf den ersten Blick sieht es so aus, als könnten wir einfach so ähnlich vorgehen, wie beim Einfügen eines Schlüssels. Das stimmt, doch wir müssen hier beachten, dass der zu löschende Schlüssel auch in einem Knoten, welcher Kinder hat, vorkommen kann.
Auf jeden Fall suchen wir zunächst nach Schlüssel x in T. Falls diese Suche fehlschlägt, dann ist nichts zu tun. Falls x in T vorkommt, dann unterscheiden wir drei Fälle, je nachdem, ob Schlüssel x in einem Knoten vorkommt, der keine, genau ein oder genau zwei Kinder hat.

- **Der Knoten mit Schlüssel x hat keine Kinder:** In diesem Fall haben wir es leicht. Sei p der Vater des Knotens mit Schlüssel x. Wir löschen das Knotenobjekt mit Schlüssel x und setzen den entsprechenden Kind-Zeiger von p auf `null`.

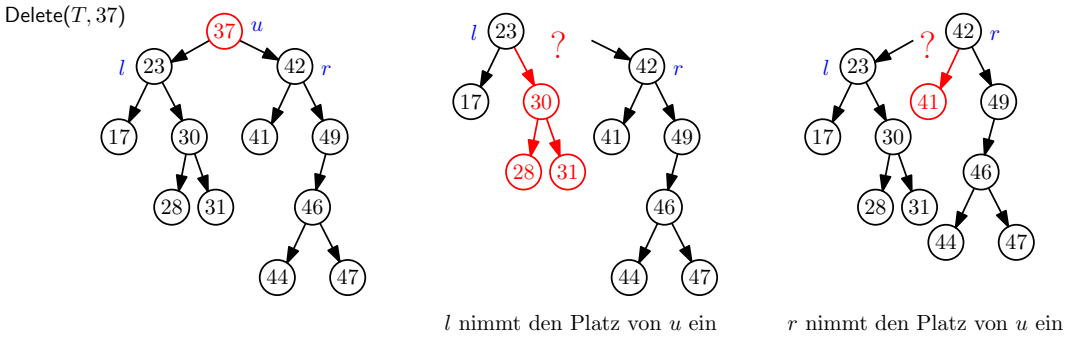


- **Der Knoten mit Schlüssel x hat genau ein Kind:** Auch dieser Fall ist leicht, denn da der Knoten mit Schlüssel x nur ein Kind hat, kann dieses einfach an dessen Stelle im Suchbaum treten. Sei u der Knoten, welcher Schlüssel x enthält, p der Vater von u und v das einzige Kind von u. Um u zu löschen, setzen wir einfach den Zeiger von p, der auf u zeigt, auf den Knoten v und den Vater-Zeiger von v auf p. Das Objekt u kann dann einfach gelöscht werden.



Da v das Kind von u ist, liegt somit v auch im Teilbaum von u. Somit steht der Schlüssel von v in der exakt derselben Beziehung zum Schlüssel von p, wie der Schlüssel von u. Da v das einzige Kind von u ist, ist v ein geeigneter "Stellvertreter" für u und die Zeiger-Änderung von p von u auf v verletzt die Suchbaumeigenschaft nicht.

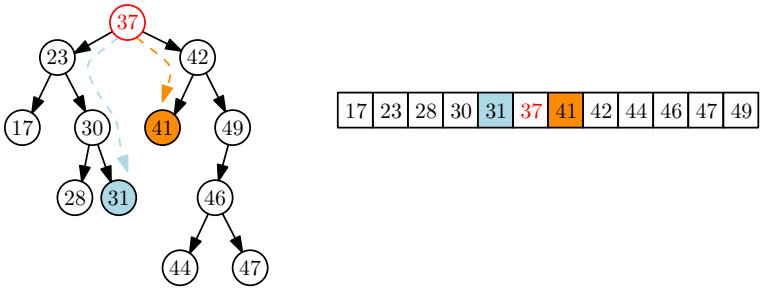
- **Der Knoten mit Schlüssel x hat genau zwei Kinder:** Dieser Fall ist nicht ganz so leicht zu lösen wie die beiden anderen. Um das Problem zu verstehen, schauen wir uns an, was passiert, wenn wir einen Knoten mit zwei Kindern löschen und - ähnlich wie im Fall mit nur einem Kind - eines der Kinder zum Stellvertreter des gelöschten Knotens machen.



Im Beispiel ist zu sehen, dass egal welches Kind wir wählen, es immer passieren kann, dass wir einen Teilbaum nicht korrekt anhängen können. Wir müssen uns also etwas clevereres einfallen lassen.

Die Stellvertreteridee ist gut, doch wir brauchen einen besseren Stellvertreter für den gelöschten Schlüssel. Außerdem wollen wir vermeiden, dass wir Zeiger im Baum ändern, denn dies kann zu den obigen Problemen führen, falls zu viele Kinder vorhanden sind.

Zuerst überlegen wir uns einen geeigneten Stellvertreter für den gelöschten Schlüssel x. Intuitiv müsste das der Schlüssel sein, der unter allen Schlüsseln im Suchbaum dem Schlüssel x am ähnlichsten ist. In unserem Kontext bedeutet das, dass der Stellvertreter von x ein Schlüssel sein muss, der ein Nachbar von x wäre, wenn alle Schlüssel sortiert in einem Array stehen würden. Übrigens gilt, dass x in diesem imaginären Array auf jeden Fall einen linken und einen rechten Nachbar haben muss, da ja der Knoten mit Schlüssel x genau zwei Kinder hat und das linke Kind einen Schlüssel der höchstens so groß wie x ist und das rechte Kind einen Schlüssel der größer als x ist, enthält.



Wir nennen den zu x nächstkleineren Schlüssel in T den *symmetrischen Vorgänger von x* und den nächstgrößeren Schlüssel in T den *symmetrischen Nachfolger von x*. In unserem Beispiel ist Schlüssel 31 der symmetrische Vorgänger von $x = 37$ und 41 ist der symmetrische Nachfolger. Praktischerweise stehen die beiden Schlüssel an einer ganz speziellen Position im Suchbaum:

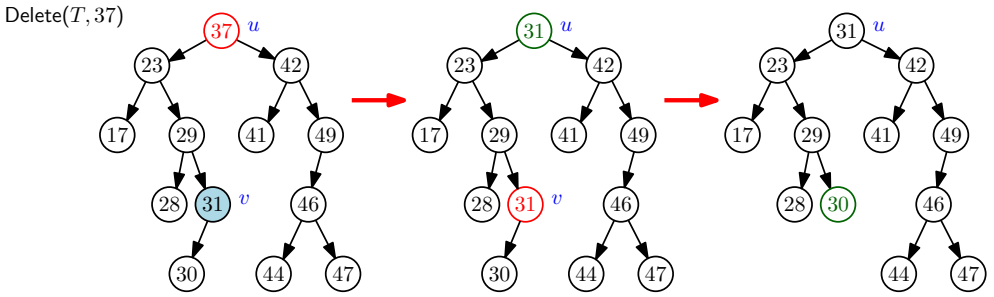
– Der symmetrische Vorgänger von x steht in dem Knoten, der erreicht wird, wenn man im Teilbaum des linken Kindes von x immer zum jeweils rechten Kind geht so lange ein solches existiert, d.h. solange man nicht auf einen `null`-Zeiger trifft.

– Der symmetrische Nachfolger von x steht in dem Knoten, der erreicht wird, wenn man im Teilbaum des rechten Kindes von x immer zum jeweils linken Kind geht so lange ein solches existiert.

Wenn wir nun den Schlüssel y des symmetrischen Vorgängers von x an die Position von x kopieren, dann verletzt dies garantiert nicht die Suchbaumeigenschaft, da ja y der größte Schlüsselwert im Teilbaum des linken Kindes des Knotens mit Schlüssel x war[80].

Nun müssen wir nur noch dafür sorgen, dass das andere Vorkommen von Schlüssel y entfernt wird. Dies ist ganz einfach denn die Knoten des symmetrischen Vor- bzw. Nachfolgers können nur höchstens ein Kind haben[81]. Somit können wir diesen überflüssigen Schlüssel wie in den anderen beiden Fällen entfernen.

Somit haben wir das folgende Vorgehen: Sei Knoten u der Knoten, der x enthält. Zuerst bestimmen wir den Knoten v, der den symmetrischen Vorgänger y von x enthält. Dies ist einfach, wir müssen nur im Teilbaum des linken Kindes von u immer zum jeweils rechten Kind gehen bis wir auf einen `null`-Zeiger treffen. Danach ersetzen wir den Schlüssel von u durch y und löschen den Schlüssel von v wie oben beschrieben. D.h. falls v kein Kind hat, dann wird v einfach gelöscht. Falls v ein Kind hat, dann wird v gelöscht und das Kind tritt an dessen Stelle.



In allen Fällen suchen wir zunächst nach dem zu löschenden Schlüssel x, d.h. wir haben Kosten von $\Theta(\text{Höhe}(T))$. Hinzu kommen noch die Kosten, die in den drei Fällen entstehen. Falls der Knoten, der x enthält, kein oder genau ein Kind hat, dann entstehen nur konstante Kosten, da nur konstant viele Zeiger geändert werden müssen.
Falls der Knoten mit Schlüssel x genau zwei Kinder hat, dann muss der symmetrische Vorgänger bestimmt werden, der Schlüssel kopiert und zusätzlich noch ein Schlüssel wie in Fall 1 oder 2 entfernt werden. Das Suchen des symmetrischen Vorgängers kostet $\Theta(\text{Höhe}(T))$, da ja x in der Wurzel stehen könnte und der Pfad zum symmetrischen Vorgänger im schlimmsten Fall der längste Wurzel-Blatt-Pfad im Suchbaum sein kann. Das Schlüsselkopieren und Entfernen des Schlüssels an der Stelle des symmetrischen Vorgängers verursacht konstante Kosten.
Insgesamt haben wir somit in allen Fällen Kosten von $\Theta(\text{Höhe}(T)) = \Theta(n)$.

[80] Man könnte an dieser Stelle auch den symmetrischen Nachfolger nehmen. Wir einigen uns aber darauf, dass immer der symmetrische Vorgänger gewählt wird. Dies ist auch konsistent mit der Entscheidung, dass gleiche Elemente immer im linken Teilbaum landen.

[81] Siehe Übung.

5.1.4 Zusammenfassung

Wenn wir ein Dictionary mit Hilfe eines binären Suchbaums implementieren, dann entstehen die folgenden Kosten:

	sortiertes Array	binärer Suchbaum
Search	$\Theta(\log n)$	$\Theta(\text{Höhe}) = \Theta(n)$
Insert	$\Theta(n)$	$\Theta(\text{Höhe}) = \Theta(n)$
Delete	$\Theta(n)$	$\Theta(\text{Höhe}) = \Theta(n)$

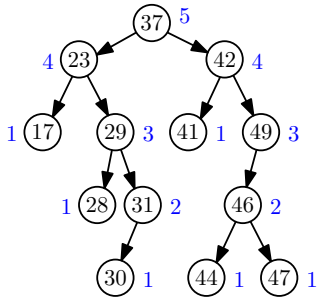
Auf unserer Suche nach einer geeigneten Datenstruktur für ein Dictionary sieht dies zunächst wie ein Rückschlag aus. Dies stimmt nur bedingt, denn die obige Tabelle zeigt uns, dass wir eigentlich nur irgendwie die Höhe des binären Suchbaums beschränken müssen, um bei allen Operationen schnell zu sein.

5.2 AVL-Bäume

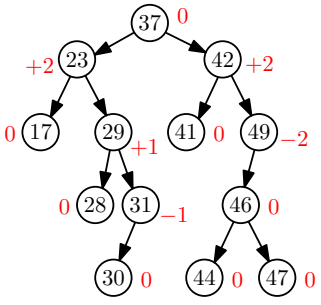
Es gibt eine beeindruckend einfache und sehr elegante Lösung, wie die Höhe von binären Suchbäumen beschränkt werden kann. Diese Lösung geht auf die beiden sowjetischen Mathematiker **A**delson-**V**elsky und **L**andis zurück und wird deshalb als *AVL-Bäume* bezeichnet.
Die Idee bei AVL-Bäumen ist einfach: Neben der Suchbaumeigenschaft wird noch eine zusätzliche Struktureigenschaft gefordert. Diese Struktureigenschaft bezieht sich auf die Höhe der Teilbäume der Kinder eines jeden Knotens. Wir nennen diese Struktureigenschaft die *AVL-Eigenschaft* und diese ist wie folgt definiert:

> Ein gerichteter Binärbaum hat die AVL-Eigenschaft, falls für jeden Knoten gilt, dass sich die Knoten-Höhe der Teilbäume seiner beiden Kinder um höchstens 1 unterscheidet.

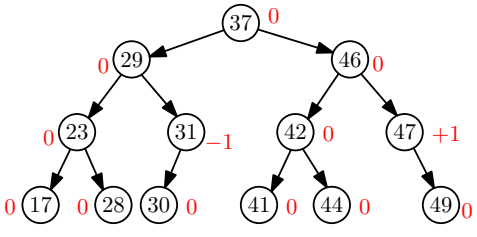
Hierbei sei die *Knoten-Höhe* eines Baums T die Anzahl der Knoten auf dem längsten Pfad von der Wurzel von T zu einem Blatt von T. D.h. es gilt, dass die Knoten-Höhe von T immer um 1 größer ist, als die Höhe von T.[82] Außerdem haben leere Teilbäume (d.h. `null`-Zeiger bei einem Kind) die Knoten-Höhe 0, da ja dort kein Knoten vorkommt.



Knoten-Höhe der Teilbäume
jeweils an der Wurzel des Teilbaums



Balance der Knoten
(kein AVL-Baum)



Balance der Knoten
(ist AVL-Baum)

[82] Zur Erinnerung: Die Höhe von T ist die Anzahl der Kanten auf dem längsten Pfad von der Wurzel von T zu einem Blatt von T.

Man kann die AVL-Eigenschaft auch noch auf andere Weise definieren. Sei u ein Knoten in einem Binärbaum T und sei l das linke Kind von u und r das rechte Kind von u. Die *Balance* von Knoten u, kurz $Bal(u)$ ist definiert als[83]:

$$Bal(u) = -(\text{Knoten-Höhe von } T[l]) + (\text{Knoten-Höhe von } T[r]).$$

Damit ergibt sich:

> Ein gerichteter Binärbaum hat die AVL-Eigenschaft, falls für alle Knoten u gilt:
>
> $$-1 \leq Bal(u) \leq +1.$$

Nun können wir AVL-Bäume definieren:

> Ein gerichteter Binärbaum T ist ein AVL-Baum, falls T die Suchbaumeigenschaft und die AVL-Eigenschaft besitzt.

5.2.1 Die Höhe von AVL-Bäumen

Zunächst ist überhaupt nicht klar, was die AVL-Eigenschaft mit der Höhe des Baumes zu tun hat. Dies klären wir, bevor wir zur Funktionsweise eines AVL-Baums kommen. Es gilt folgendes:

Theorem 22. *Ein gerichteter Binärbaum T mit n Knoten hat Höhe $\Theta(\log n)$, falls T die AVL-Eigenschaft hat.*

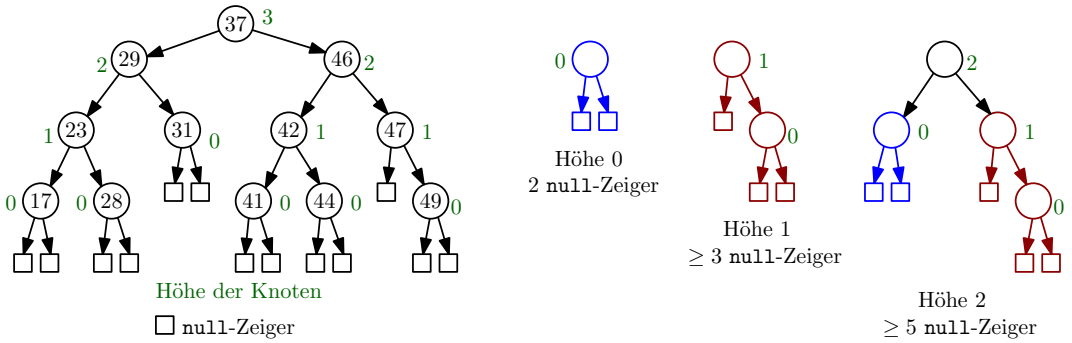
Beweis. Um die Aussage zu beweisen, werden wir uns eines kleinen Tricks bedienen. Wir interessieren uns nicht für die Anzahl der Knoten von T sondern wir werden die Anzahl der `null`-Zeiger von T betrachten. Dabei interessieren uns hierbei die `null`-Zeiger, die für fehlende Kinder stehen, d.h. wir ignorieren den `null`-Zeiger im Vater-Eintrag des Wurzelknotens. `null`-Zeiger, die für fehlende Kinder stehen, nennen wir *Kinder-`null`-Zeiger*.
Die Anzahl der `null`-Zeiger ist interessant, da die Anzahl der Kinder-`null`-Zeiger von T eine obere Schranke für die Anzahl der Knoten von T ist. Genauer gilt:

Lemma 15. *Jeder gerichtete Binärbaum mit k Knoten hat genau $k+1$ viele Kinder-`null`-Zeiger.*

Beweis. Übungsaufgabe. □

Wenn wir uns nun die Kinder-`null`-Zeiger von T als Blätter vorstellen, dann gilt, dass es genau einen Kinder-`null`-Zeiger mehr als Knoten von T gibt.

[83] Achtung, in der Literatur (z.B. Wikipedia) wird zum Teil auch die umgekehrte Version, d.h. Knoten-Höhe von $T[l]$ - Knoten-Höhe von $T[r]$, genutzt. Beide Versionen verhalten sich exakt gleich.



Jetzt versuchen wir, einen möglichst hohen gerichteten Binärbaum zu konstruieren, welcher aber trotzdem die AVL-Eigenschaft hat. Dazu versuchen wir, den Baum so "unbalanciert" wie möglich zu machen. Dies erreichen wir laut Lemma 15, in dem wir für eine gegebene Höhe h so wenige Kinder-`null`-Zeiger wie möglich verwenden.
Wir betrachten zunächst einen Baum mit Höhe 0, d.h. einen Baum der nur aus einem Knoten und zwei Kinder-`null`-Zeigern besteht. Andere Bäume mit Höhe 0 gibt es nicht. Somit benötigen wir für Höhe 0 genau zwei Kinder-`null`-Zeiger. Bei einem Baum mit Höhe 1 haben wir mehr Spielraum, dieser könnte 2 Knoten und somit 3 Kinder-`null`-Zeiger oder 3 Knoten und somit 4 Kinder-`null`-Zeiger enthalten. Für Höhe 2 sind es mindestens 4 Knoten und 5 Kinder-`null`-Zeiger und maximal 7 Knoten und 8 Kinder-`null`-Zeiger. Wenn wir uns den Baum mit Höhe 2 mit 5 Kinder-`null`-Zeigern ansehen, dann fällt auf, dass dieser aus einem Baum mit Höhe 1 mit 3 Kinder-`null`-Zeigern und einem Baum mit Höhe 0 mit 2 Kinder-`null`-Zeigern zusammengesetzt ist. Wir zeigen zuerst, dass das kein Zufall ist. Sei $Z(h)$ die kleinstmögliche Anzahl von Kinder-`null`-Zeigern eines gerichteten Binärbaums mit AVL-Eigenschaft mit Höhe h.

Lemma 16. *Für alle $h \geq 2$ gilt: $Z(h) = Z(h-1) + Z(h-2)$.*

Beweis. Wir zeigen die Aussage per Induktion über h. Der Induktionsanfang für $h = 2$ ist klar, da $Z(2) = Z(1) + Z(0) = 5$. Dass $Z(1) = 3$ und $Z(0) = 2$ gilt, haben wir uns bereits weiter oben überlegt.
Für ein beliebiges $h \geq 2$ gilt, dass die Wurzel w des Baumes T genau zwei Kinder haben muss, da sonst die AVL-Eigenschaft verletzt wäre. Sei l das linke Kind von w und r das rechte Kind von w. Da der Baum T die Höhe h hat, muss die Wurzel w die Höhe h haben. Folglich muss $T[l]$ oder $T[r]$ die Höhe $h-1$ haben. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass $T[r]$ Höhe $h-1$ hat. Nach Induktionsvoraussetzung wissen wir, dass der Teilbaum $T[r]$ somit mindestens $Z(h-1)$ viele Kinder-`null`-Zeiger hat. Für die Höhe von $T[l]$ kommen zwei Werte in Frage, der Teilbaum könnte Höhe $h-1$ oder $h-2$ haben. Nach Induktionsvoraussetzung gilt $Z(h-2) < Z(h-1)$ und somit hat $T[l]$ mindestens $Z(h-2)$ viele Kinder-`null`-Zeiger. Insgesamt hat T somit mindestens $Z(h) = Z(h-1) + Z(h-2)$ viele Kinder-`null`-Zeiger. □

Wir haben somit die Rekurrenz:

$$Z(0) = 2, Z(1) = 3 \text{ und } Z(h) = Z(h-2) + Z(h-1).$$

Betrachten wir die Werte von $Z(h)$ für kleine h:

h	0	1	2	3	4	5	6	7	8
$Z(h)$	2	3	5	8	13	21	34	55	89

Betrachten wir als Vergleich die Zahlen der *Fibonacci-Folge* F, die als $F(0) = 0, F(1) = 1$ und $F(i) = F(i-2) + F(i-1)$ für alle $i \geq 2$ definiert ist:

i	0	1	2	3	4	5	6	7	8	9	10	11
$F(i)$	0	1	1	2	3	5	8	13	21	34	55	89

Man kann leicht zeigen, dass $Z(h) = F(h+3)$ gilt. Somit wissen wir, dass ein AVL-Baum mit Höhe h mindestens $F(h+3)$ viele Kinder-`null`-Zeiger und somit mindestens $F(h+3) - 1$ viele Knoten hat. Dies hilft uns, denn für die i-te Fibonacci-Zahl $F(i)$ gibt es eine geschlossene Formel:[84]

$$F(i) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - \left(\frac{1-\sqrt{5}}{2} \right)^i \right).$$

Der hintere Term in der Klammer ist für jedes i kleiner als 1, somit erhalten wir folgende untere Schranke an die i-te Fibonacci-Zahl:

$$F(i) \geq \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - 1 \right).$$

Somit gilt

$$n \geq Z(h) - 1 = F(h+3) - 1 \geq \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \right) - 1.$$

Wenn wir dies nun nach h umstellen, dann erhalten wir $h \in \mathcal{O}(\log n)$.
Damit haben wir gezeigt, dass selbst der unbalancierteste AVL-Baum mit n Knoten eine Höhe in $\mathcal{O}(\log n)$ hat.
Da ein vollständiger gerichteter Binärbaum mit n Knoten, egal wie genau das letzte Level aussieht, auch die AVL-Eigenschaft hat und dieser der binäre Suchbaum mit n Knoten mit der geringsten Höhe ist, folgt, dass die Höhe eines AVL-Baums auch in $\Omega(\log n)$ ist. □

Nun da wir wissen, dass AVL-Bäume immer logarithmische Höhe haben, können wir uns mit der Funktionsweise beschäftigen.

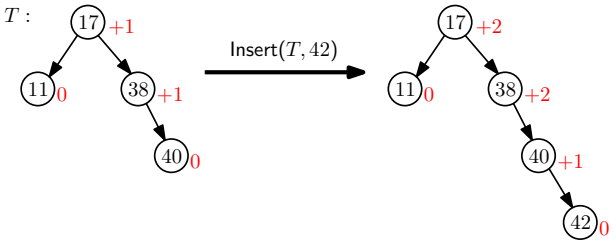
5.2.2 Suchen in einem AVL-Baum

Da AVL-Bäume spezielle binäre Suchbäume sind, können wir in AVL-Bäumen exakt genau so suchen wie in den bereits bekannten binären Suchbäumen. Somit kostet uns eine Suche in einem AVL-Baum T mit n Knoten $\Theta(\text{Höhe}(T)) = \Theta(\log n)$, da wird ja wissen, dass die Höhe von T in $\Theta(\log n)$ ist.

[84] Die Formel von Moivre und Binet.

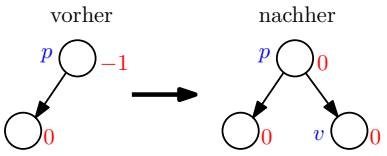
5.2.3 Einfügen in einen AVL-Baum

Da AVL-Bäume auch binäre Suchbäume sind, können wir exakt genau so neue Schlüssel einfügen, wie wir dies bereits von binären Suchbäumen kennen. Allerdings müssen wir danach noch dafür sorgen, dass wir wieder einen korrekten AVL-Baum vorliegen haben, d.h. dass die AVL-Eigenschaft noch immer gilt. Es ist leicht zu sehen, dass das Einfügen eines weiteren Schlüssels die AVL-Eigenschaft zerstören kann:



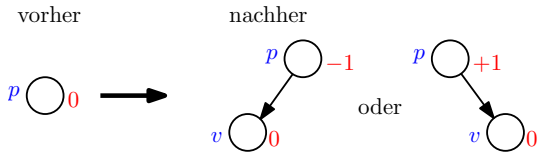
Allerdings ist auch zu sehen, dass die AVL-Eigenschaft nur an den Knoten verletzt sein kann, die auf dem Weg vom neu erstellten Knoten bis zur Wurzel liegen. Grund ist, dass sich nur bei diesen Knoten die Höhe eines Teilbaums eines Kindes ändert. D.h. um die AVL-Eigenschaft wieder herzustellen, müssen wir uns nur um diese Knoten kümmern. Wir gehen dabei vom eingefügten Knoten nach oben vor. Sei v der neu eingefügte Knoten und sei p der Vaterknoten von v.
Zunächst gilt, dass v eine Balance von 0 hat, da wir diesen ja gerade erst eingefügt haben und dieser somit keine Kinder hat. Folglich erfüllt v die AVL-Eigenschaft.
Nun betrachten wir den Knoten p und unterscheiden mehrere Fälle, je nach Wert von $Bal(p)$ *vor dem Einfügen*. Dafür müssen wir $Bal(p)$ nicht berechnen, sondern wir einigen uns darauf, dass wir die Balance-Information einfach in jedem Knoten direkt speichern und gegebenenfalls aktualisieren.

1. $Bal(p) = -1$, d.h. vor dem Einfügen von Knoten v war der Teilbaum des linken Kindes um 1 höher als der Teilbaum des rechten Kindes von p. Somit muss v als rechtes Kind von p eingefügt worden sein und vorher war an dieser Stelle nur ein `null`-Zeiger. Wir haben somit folgende Situation:

   Die Balance von p ändert sich von -1 auf 0, aber die Höhe des Teilbaums $T[p]$ ändert sich nicht. Somit sind wir fertig, denn somit hat sich auch auf dem Pfad von p bis zur Wurzel bei keinem Knoten die Höhe eines Teilbaums eines Kindes geändert und folglich muss überall die AVL-Eigenschaft gelten.

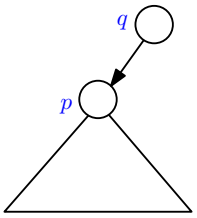
2. $Bal(p) = 0$: In diesem Fall ändert sich die Balance von p auf jeden Fall, nämlich entweder zu -1 oder zu $+1$, je nachdem, ob v als linkes oder rechtes Kind von p eingefügt wird.



Außerdem ändern sich auch die Höhe des Teilbaums $T[p]$ und somit kann sich auch bei den Knoten oberhalb von p auf dem Pfad bis zur Wurzel die Balance geändert haben. Wir müssen somit rekursiv den Vater von p betrachten. Wir tun dies, indem wir den Algorithmus upin(p) aufrufen.

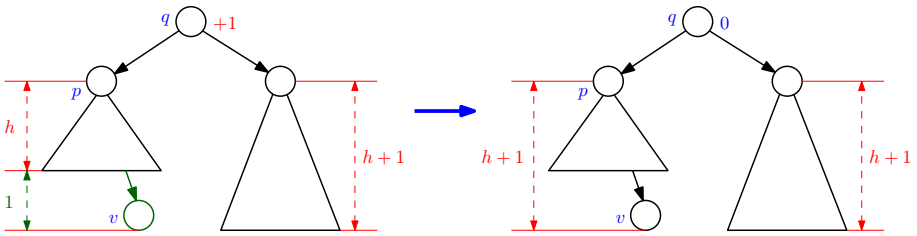
3. $Bal(p) = +1$: Dieser Falls ist symmetrisch zu Fall 1. Auch hier wird v einfach als linkes Kind zu p hinzugefügt, die Balance von p ändert sich auf 0 und da sich die Höhe des Teilbaums nicht ändert, muss die AVL-Eigenschaft bei allen Knoten im AVL-Baum weiterhin gelten.

Nun beschäftigen wir uns noch mit dem Algorithmus upin(p). Damit dieser für Knoten p aufgerufen wird, muss folgende Invariante gelten: Die Balance von p muss ungleich 0 sein und die Höhe des Teilbaums $T[p]$ muss um 1 gewachsen sein.
Wieder betrachten wir ein paar Fälle. Dazu müssen wir unterscheiden, ob p ein linkes oder ein rechtes Kind seines Vaters q ist. (Falls p die Wurzel des AVL-Baums ist, dann sind wir trivialerweise fertig.) Beide Fälle sind symmetrisch, weshalb wir uns hier darauf beschränken, dass p das linke Kind von q ist. Wir haben somit folgende Situation:



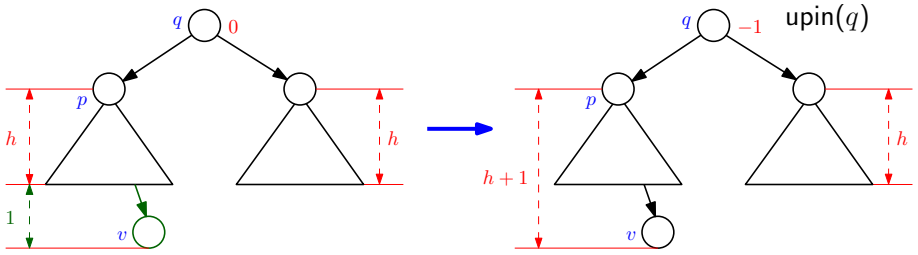
Nun kommt die Fallunterscheidung, je nach Balance von Knoten q vor dem Einfügen:

1. $Bal(q) = +1$: In diesem Fall muss die Höhe des Teilbaums des rechten Kinds von q vor dem Einfügen um 1 größer sein, als die Höhe von $T[p]$.



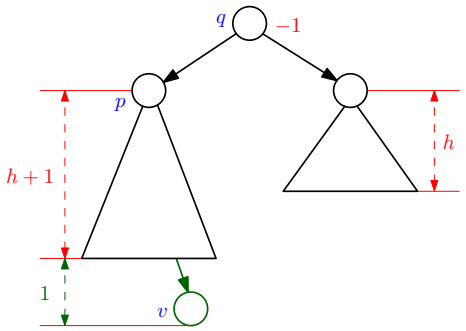
Da nach dem Einfügen die Höhe von $T[p]$ um 1 wächst, sind nun die Teilbäume beider Kinder von q gleich hoch und somit wechselt die Balance von q von $+1$ auf 0. Da sich die Höhe von Teilbaum $T[q]$ nicht verändert, sind wir fertig, denn somit muss auch bei allen Knoten auf dem Pfad von q zur Wurzel noch die AVL-Eigenschaft gelten. Somit bricht upin hier ab.

2. $Bal(q) = 0$: Wir haben folgende Situation:



Die Balance von q ändert sich zu -1 und da die Höhe des Teilbaums $T[q]$ um 1 zunimmt, müssen wir upin(q) aufrufen.

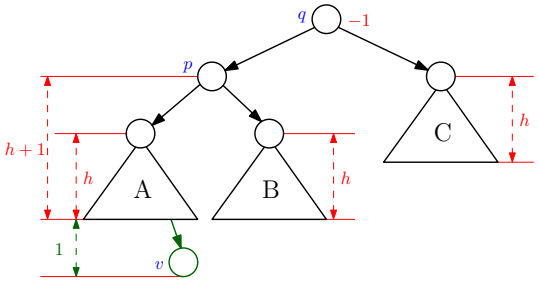
3. $Bal(q) = -1$. Hier haben wir ein Problem, denn da sich die Höhe von $T[p]$ erhöht hat, wäre die Balance von q nach dem Einfügen nun -2 und somit würde q die AVL-Eigenschaft verletzen. Die Situation sieht wie folgt aus:



Um dieses Problem zu beheben, können wir allerdings nicht einfach Knoten im Baum umhängen, denn der Baum soll ja auch weiterhin die Suchbaumeigenschaft haben. Außerdem, wollen wir das Problem am besten mit konstanten Kosten beheben, da unser Ziel insgesamt logarithmische Kosten für das Einfügen ist. Die Lösung ist ein sehr eleganter Trick, denn wir müssen, je nach Balance von p, nur wenige Zeiger "umhängen".

Betrachten wir also die Balance von p, laut unserer Invariante kann nur $Bal(p) = -1$ und $Bal(p) = +1$ gelten.

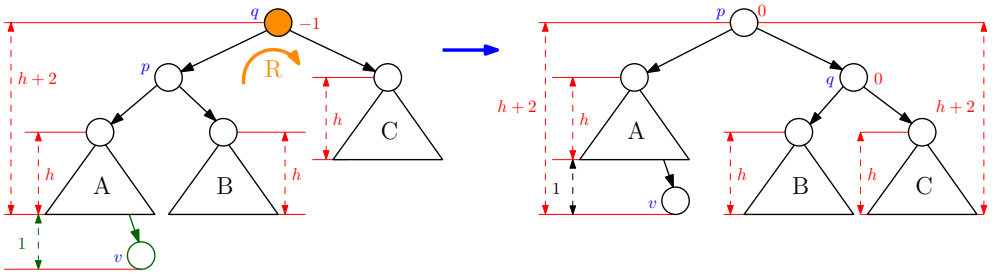
(a) $Bal(p) = -1$: In diesem Fall muss v im Teilbaum des linken Kindes von p eingefügt worden sein, somit haben wir folgende Situation:



Wenn wir nun die Schlüsselwerte betrachten, dann gilt:

$$\text{Schlüssel aus } A < p < \text{Schlüssel aus } B < q < \text{Schlüssel aus } C.$$

Jetzt führen wir eine sogenannte *einfache Rotation nach rechts* am Knoten q durch: Dabei machen wir p zur neuen Wurzel des Teilbaums, q zum rechten Kind von p und der Teilbaum B wird linkes Kind von q:



Danach haben p und q beide Balance 0 und die Höhe des Gesamtteilbaums nach der Rotation ist exakt genauso hoch wie die Höhe des Teilbaums vor dem Einfügen von v. Somit können wir upin() hier abbrechen.

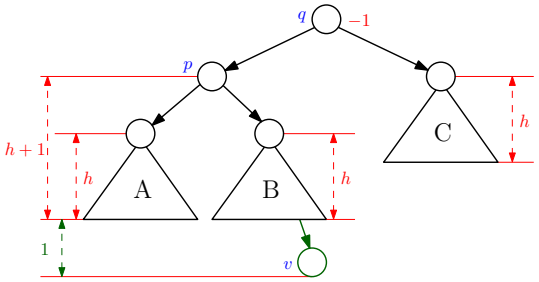
Da die Schlüsselwerte der Teilbäume A, B und C in der oben genannten Beziehung zu p und q stehen, ist der resultierende Teilbaum mit Wurzel p weiterhin ein korrekter binärer Suchbaum.

Man beachte, dass wir für die einfache Rotation nach rechts nur konstant viele Zeiger ändern mussten und die Balance von p und q aktualisieren. Es ändern sich maximal 6 Zeiger:

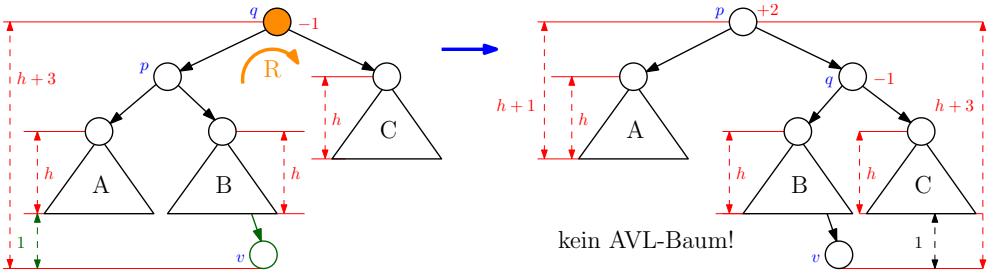
- Falls q nicht die Wurzel des AVL-Baums war, wird der Kind-Zeiger des Vaters von q von q nach p geändert.
- Der Vater-Zeiger von p wird von q auf den Vater von q geändert. Falls q die Wurzel war, denn wird der neue Vater-Zeiger von p der `null`-Zeiger.
- Der rechtes-Kind-Zeiger von p wird von der Wurzel von Teilbaum B auf q geändert.
- Der Vater-Zeiger von q zeigt nun auf p.
- Der linkes-Kind-Zeiger von q wird von p auf die Wurzel von Teilbaum B geändert.
- Der Vater-Zeiger der Wurzel von Teilbaum B wird von p auf q geändert.

Somit kostet eine einfache Rotation nach rechts bzw. eine einfache Rotation nach links nur $\Theta(1)$.

(b) $Bal(p) = +1$: Wir haben die folgende Situation:



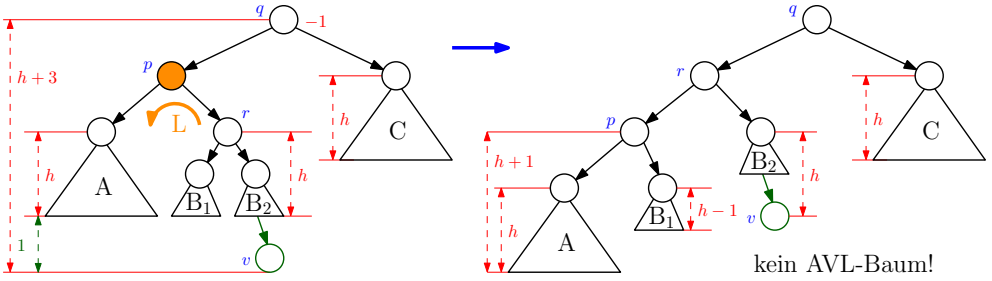
Wenn wir hier wieder den selben Trick, also eine einfache Rotation nach rechts, anwenden würden, dann passiert folgendes:



Es entstehen zwei Probleme: Erstens haben wir es nicht geschafft, dass die Höhe des gesamten Teilbaums die selbe Höhe wie vor dem Einfügen hat. Zweitens ist im neuen Teilbaum nach der Rotation die AVL-Eigenschaft für Knoten p verletzt!

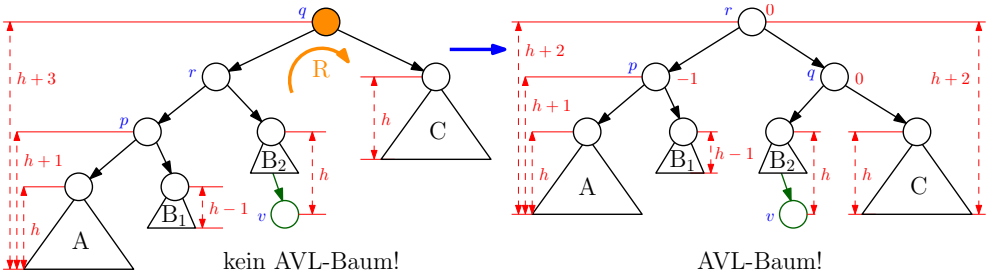
Das Problem entsteht hier, weil der Teilbaum B, also der "innere" Teilbaum von p der höhere der beiden Teilbäume von p ist. Bei der einfachen Rotation wird ja dieser innere Teilbaum auf der selben Ebene im Baum einem anderen Vater (nämlich q) zugewiesen. Somit ändert sich an der Höhe des gesamten Baumes nichts. Wir haben allerdings eben gesehen, dass eine einfache Rotation nach rechts funktioniert, wenn der "äußere" Teilbaum von p der höhere der beiden Teilbäume von p ist. Dann nämlich wird dessen Höhe um 1 verringert, da ja p die Wurzel wird und somit ein Level aufsteigt. Wir müssen also lediglich dafür sorgen, dass der äußere Teilbaum von p der höhere ist. Dies ist ganz leicht, wir führen einfach vor der einfachen Rotation nach rechts am Knoten q eine einfache Rotation nach links am Knoten p durch!

Betrachten wir, was genau passiert: Zunächst die einfache Rotation nach links am Knoten p:



Es entsteht kein AVL-Baum, da die AVL-Eigenschaft bei Knoten q verletzt ist. Allerdings haben wir jetzt genau das, was wir haben wollten, nämlich, dass der höhere Teilbaum der Kinder des linken Kinds von q, d.h. von r, jetzt der äußere ist. Übrigens spielt es keine Rolle, ob v zum Teilbaum B_1 oder B_2 gehört, in beiden Fällen erhalten wir die gewünschte Eigenschaft.

Wenn wir jetzt eine einfache Rotation nach rechts am Knoten q folgen lassen, dann erhalten wir wieder einen AVL-Baum und sogar mit der gewünschten Höhe:



Diese Kombination aus einfacher Linksrotation und einfacher Rechtsrotation nennen wir *Doppelrotation LR*. Auch hier ist klar, dass nur konstant viele Zeiger geändert und nur konstant viele Balance-Werte aktualisiert werden müssen.

Der resultierende Teilbaum ist somit ein AVL-Baum, außerdem hat dieser die selbe Höhe wie vor dem Einfügen von v. Somit können wir upin() hier abbrechen.

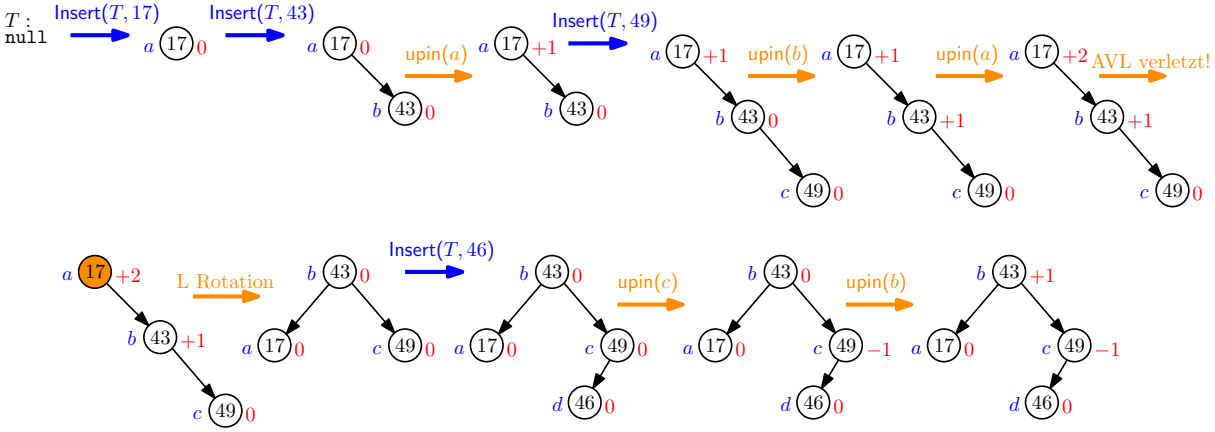
Auch hier ist klar, dass nur konstant viele Zeiger geändert und nur konstant viele Balance-Werte aktualisiert werden müssen. Somit kostet eine Doppelrotation LR bzw. eine Doppelrotation RL $\Theta(1)$.

Es bleibt noch die Frage, ob der entstandene Teilbaum weiterhin ein korrekter Suchbaum ist. Dies gilt, dass die Schlüssel der Teilbäume A, B_1, B_2 und C wie folgt im Verhältnis zu p, q und r stehen:

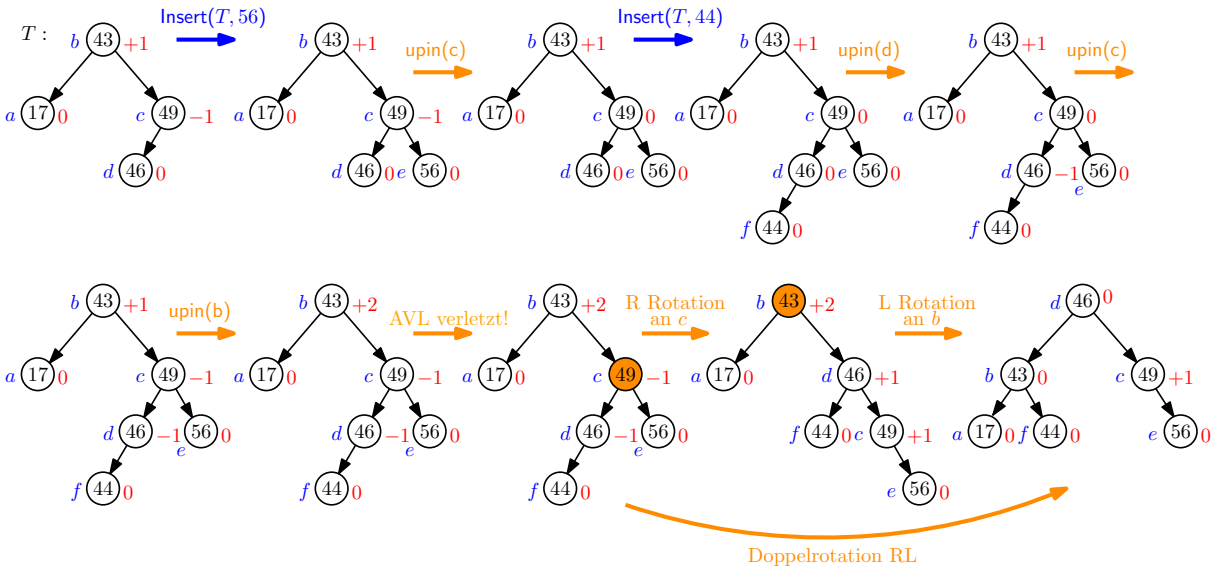
$$\{A\} < p < \{B_1\} < r < \{B_2\} < q < \{C\},$$

wobei z.B. $\{A\}$ hier für jeden Schlüssel im Teilbaum A steht.

Insgesamt kann upin() genau ein Mal für jeden Knoten auf dem Pfad von p zur Wurzel des AVL-Baums aufgerufen werden. Jeder Aufruf verursacht konstante Kosten. Da wir wissen, dass AVL-Bäume logarithmische Höhe haben, folgt, dass das Einfügen eines Schlüssels in einen AVL-Baum mit n Knoten $\Theta(\log n)$ kostet.
Bemerkenswert ist, auch, dass upin() immer dann abbricht, wenn wir eine Rotation oder Doppelrotation ausgeführt haben. Somit kann pro Einfügeoperation überhaupt nur ein mal eine solche Strukturänderung im Baum auftreten.
Betrachten wir ein Beispiel. Wir fügen die Schlüssel $17, 43, 49, 46$ nacheinander in einen anfangs leeren AVL-Baum ein:



Nun fügen wir noch die Schlüssel 56 und 44 nacheinander ein:



5.2.4 Löschen aus AVL-Bäumen

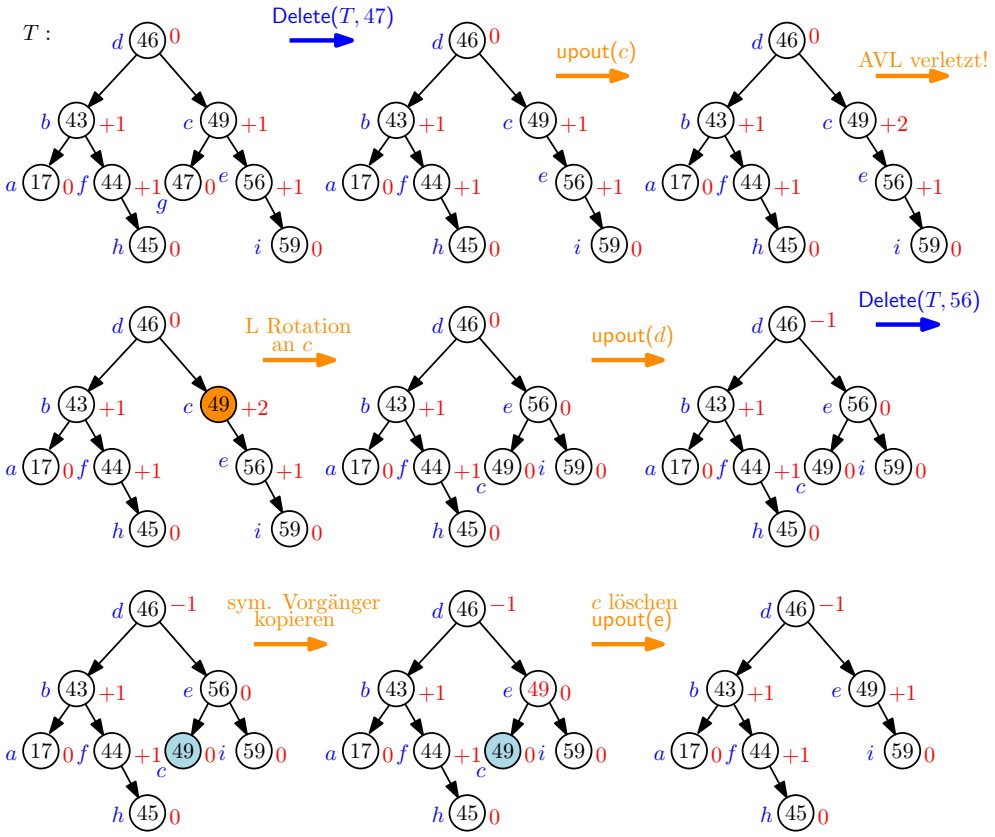
Für das Löschen eines Schlüssels aus einem AVL-Baum können wir viele der Ideen von oben wiederbenutzen. Prinzipiell gehen wir beim Löschen von Schlüssel x wie folgt vor:

1. Suche nach x.
2. Lösche x genau wie in einem binären Suchbaum.
3. Iteriere das folgende Vorgehen, genannt upout(), beginnend beim Vater p des gelöschten Knotens bis zur Wurzel. D.h. am Anfang gilt $z = p$.

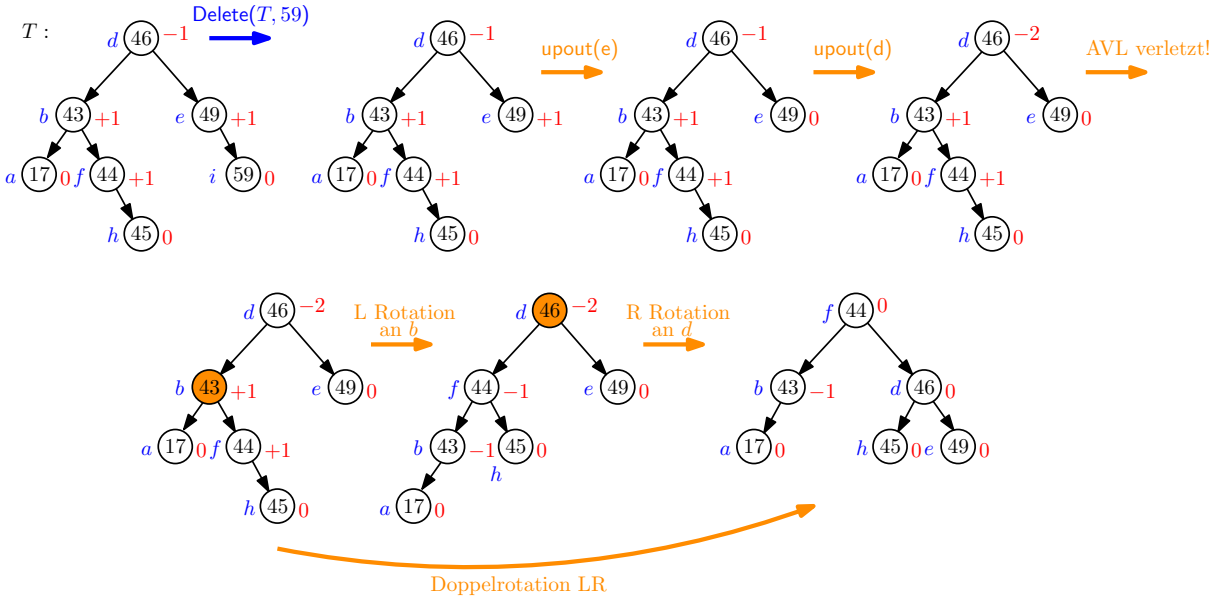
   3.1) Aktualisieren die Balance von z und überprüfe dabei, ob die AVL-Eigenschaft für z noch erfüllt ist. Falls dies gilt und falls das Knotenlöschen die Höhe von $T[z]$ nicht geändert hat, dann beende.

   3.2) Falls die AVL-Eigenschaft für z nicht gilt, dann führe, je nach Balancewert von z und seinen Kindern eine Rotation bzw. Doppelrotation durch, um diese für alle Knoten im aktuell betrachteten Teilbaum wieder herzustellen. Falls diese Rotation die Höhe des gesamten Teilbaums verändert hat, dann fahre rekursiv mit dem (evtl. ehemaligen) Vater von z fort.

Da Rotationen und Doppelrotationen nur konstante Kosten haben, ergeben sich auch hier wieder insgesamt nur Kosten von $\Theta(\log n)$ für das Löschen eines Schlüssels aus einem AVL-Baum mit n Knoten. Es gibt allerdings einen Unterschied zum Einfügen: Beim Löschen eines Schlüssels können mehrere Rotationen auftreten, im worst-case sogar bei jedem Durchlauf von Schritt 3, d.h. $\Theta(\log n)$ mal.
Wir betrachten zunächst ein Beispiel, in dem eine Linksrotation auftritt und in dem keine Rotation auftritt:



Man beachte, dass beim Löschen von 56 zunächst der symmetrische Vorgänger gesucht wird, dieser Kopiert und dann der Knoten des symmetrischen Vorgängers gelöscht wird. Nun sehen wir, was passiert, wenn wir als nächstes Schlüssel 59 löschen:



5.2.5 Zusammenfassung

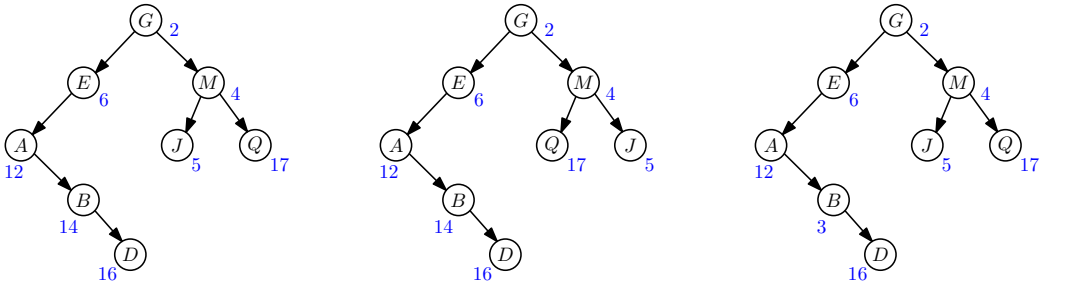
Wenn wir ein Dictionary mit Hilfe eines AVL-Baums implementieren, dann entstehen die folgenden Kosten:

	sortiertes Array	binärer Suchbaum	AVL-Baum
Search	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$
Insert	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
Delete	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$

Somit sind AVL-Bäume eine ideale Datenstruktur, um ein Dictionary zu implementieren.

5.3 Treaps - randomisierte balancierte Suchbäume

Als letztes betrachten wir noch Treaps, eine *randomisierte* Datenstruktur, die ein Dictionary implementiert. Die Idee für einen Treap ist einfach: Wir kombinieren einen binären Suchbaum mit einem Min-Heap[85]. Genauer: In jeden Knoten wird ein Schlüssel und eine Priorität gespeichert. Ein Treap ist ein binärer Suchbaum bezüglich der Schlüssel und gleichzeitig ein Min-Heap bezüglich der Prioritäten. Hier ein Beispiel für einen Treap:



Treap | kein Treap, Schlüsselbedingung verletzt | kein Treap, Heapeigenschaft verletzt

In den Beispielen wählen wir als Schlüssel immer Buchstaben und die Prioritäten sind natürliche Zahlen (blau dargestellt).
Die Prioritäten der Knoten werden bei Treaps zufällig festgelegt. Genauer: Sie werden *unabhängig* und *gleichverteilt* aus einem großen Intervall gezogen. Entweder man nimmt hierfür eine zufällige reelle Zahl zwischen 0 und 1 (damit sind gleiche Prioritäten praktisch ausgeschlossen) oder z.B. eine zufällige Zahl im Intervall $[0, 2^{64} - 1]$ oder ähnliches. Wichtig ist, dass das Intervall groß genug sein muss, damit gleiche Prioritäten nur mit vernachlässigbar kleiner Wahrscheinlichkeit auftreten können.

5.3.1 Suchen in Treaps:

Da Treaps binäre Suchbäume sind, können wir darin exakt genauso suchen, wie in binären Suchbäumen. Somit hängen die Kosten von Search von der Tiefe des Treaps ab. Die Kosten von Search sind in $\mathcal{O}$(Tiefe).

5.3.2 Einfügen in Treaps:

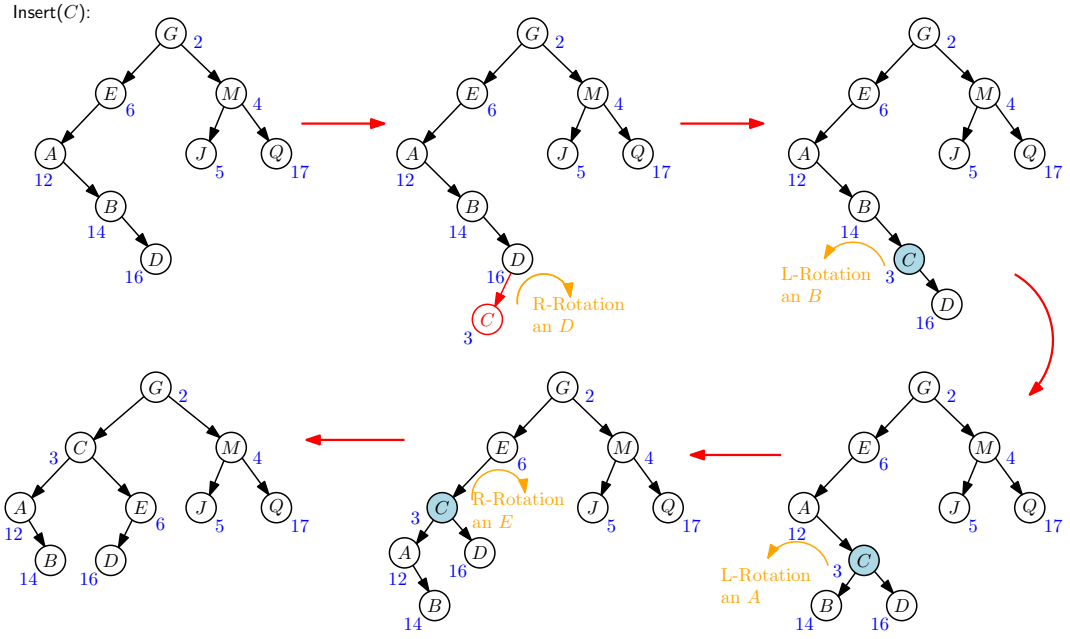
Wollen wir einen neuen Schlüssel x in einen Treap einfügen, dann machen wir folgendes:

- Suche nach dem Schlüssel x, Abbruch, falls x bereits im Treap.
- Falls x noch nicht im Treap, dann endet die Suche nach x erfolglos in einem `null`-Pointer.

[85] Daher auch der Name. Es ist eine Mischung aus tree und heap.

- Erstelle an der Position, an der die Suche endete einen neuen Knoten mit Schlüssel x und zufälliger Priorität.
- Stelle ausgehend vom neuen Knoten mittels *einfacher* Links- bzw. Rechtsrotationen (wie bei AVL-Bäumen) die Heapeigenschaft auf dem Pfad bis zur Wurzel wieder her, falls diese verletzt ist.

Betrachten wir ein Beispiel:



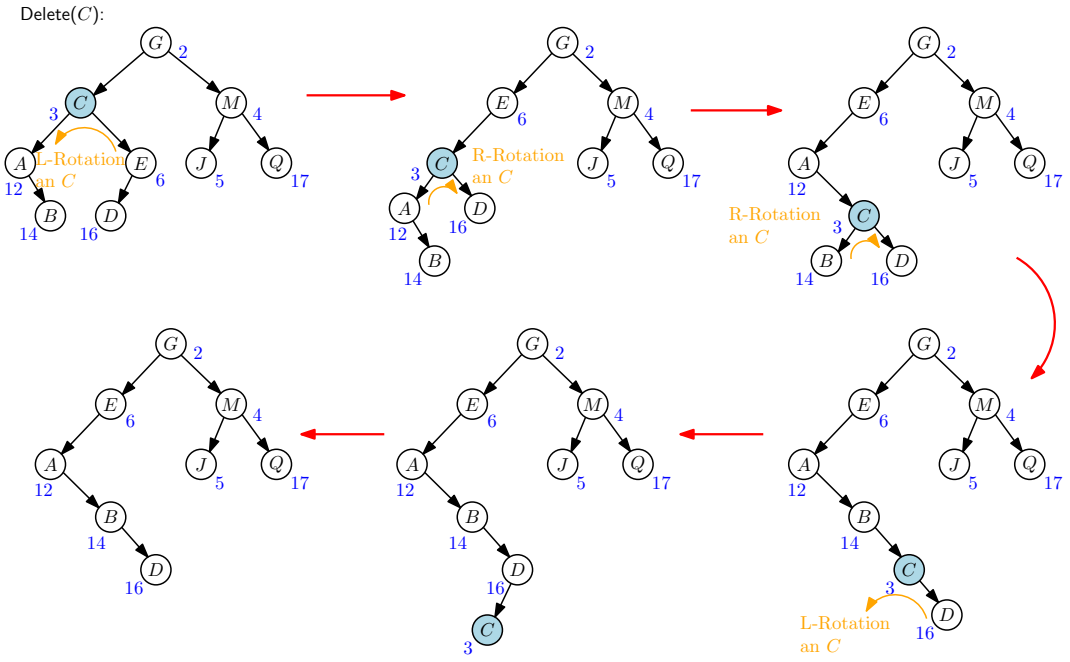
Pro Knoten auf dem Weg zur Wurzel kann eine Rotation vorkommen. Jede Rotation hat konstante Kosten, weshalb die Kosten von Insert im wesentlichen durch die Tiefe des Treaps bestimmt werden. Die Kosten von Insert sind in $\mathcal{O}(\text{Tiefe})$.

5.3.3 Löschen in Treaps:

Das Löschen eines Schlüssels funktioniert genau wie ein rückwärts ablaufende Einfügeoperation:

- Rotiere den zu löschenden Schlüssel mittels *einfacher* Links- bzw. Rechtsrotationen nach unten, bis dieser in einem Blatt steht. Tausche bei der Rotation immer das Kind mit der kleinsten Priorität nach oben.
- Sobald der zu löschende Schlüssel in einem Blatt steht, entferne den entsprechenden Blattknoten.

Betrachten wir ein Beispiel:



Im schlimmsten Fall wird pro Knoten auf dem Weg längsten Weg von der Position des zu löschenden Schlüssels bis zu einem Blatt je ein mal rotiert. Somit sind die Kosten von Delete in $\mathcal{O}(\text{Tiefe})$.
Für alle drei Operationen Search, Insert und Delete haben wir ermittelt, dass die Kosten von der Tiefe eines Treaps bestimmt wird. Wir ermitteln nun die *erwartete Tiefe* eines Treaps.

5.3.4 Erwartete Tiefe eines Treaps:

Die Struktur und somit auch die Tiefe eines Treaps hängt von den zufällig bestimmten Prioritäten der Schlüssel ab und ist somit eine *Zufallsvariable*. Im worst-case kann es passieren, dass der Treap zu einem Pfad ausartet und somit die Tiefe in $\Theta(n)$ ist. Wir zeigen nun, dass der Erwartungswert der Tiefe deutlich geringer, nämlich in $\mathcal{O}(\log n)$ ist.

Theorem 23. *Die erwartete Tiefe eines beliebigen Knotens in einem Treap mit n Knoten ist in $\mathcal{O}(\log n)$.*

Beweis. Sei x_k der k-kleinste Schlüssel im Treap. Um uns Schreibarbeit zu sparen, führen wir folgende Notation ein: $i \uparrow k$ bedeutet, dass x_i auf dem Weg von x_k zur Wurzel des Treaps liegt. D.h. $i \uparrow k$ bedeutet, dass x_i ein Vorfahre von x_k ist.
Die Tiefe eines Knotens (bzw. Schlüssels) entspricht der Anzahl der Knoten auf dem Pfad bis zur Wurzel, somit gilt für die Tiefe von x_k, kurz $t(x_k)$:

$$t(x_k) = \sum_{i=1}^{n} [i \uparrow k],$$

wobei hier $[i \uparrow k] = 1$, falls die Aussage $i \uparrow k$ gilt und $[i \uparrow k] = 0$ sonst. Für die erwartete Tiefe des Knotens x_k gilt somit

$$E[t(x_k)] = \sum_{i=1}^{n} E\big[[i \uparrow k]\big] = \sum_{i=1}^{n} Pr[i \uparrow k].$$

Hierbei haben wir bei der ersten Gleichung die *Linearität des Erwartungswerts* ausgenutzt. Es gilt, dass der Erwartungswert einer Summe von Zufallsvariablen X und Y gleich der Summe der Erwartungswerte der einzelnen Zufallsvariablen ist, d.h. $E[X+Y] = E[X]+E[Y]$. Die zweite Gleichung gilt, da $[i \uparrow k]$ eine sogenannte *Indikatorzufallsvariable* ist, d.h. die Zufallsvariable hat nur die Werte 0 und 1. Für Indikatorzufallsvariablen X gilt $E[X] = Pr[X = 1]$.
Um die erwartete Tiefe eines Knotens zu bestimmen, müssen wir also nur die Wahrscheinlichkeit berechnen, mit der ein Knoten ein Vorfahre eines anderen Knotens ist. Um dies leicht machen zu können, benötigen wir noch ein Hilfslemma und etwas zusätzliche Notation.
Sei $S(i, k)$ die Teilmenge $\{x_i, x_{i+1}, \ldots, x_k\}$ oder $\{x_k, x_{k+1}, \ldots, x_i\}$ der Knoten des Treaps, je nachdem ob $i < k$ oder $k < i$ gilt. Die Reihenfolge der Argumente von $S()$ ist egal, es gilt $S(i, k) = S(k, i)$. Außerdem enthält $S(1, n) = S(n, 1)$ alle Knoten des Treaps.

Lemma 17. *Für alle $i \neq k$ tritt das Ereignis $i \uparrow k$ genau dann ein, wenn x_i die kleinste Priorität unter allen Knoten in $S(i, k)$ hat.*

Beweis. Es gibt die folgenden Fälle:

1. Falls x_i die Wurzel ist, dann tritt Ereignis $i \uparrow k$ per Definition ein, da x_i die kleinste Priorität aller Knoten im Treap haben muss.
2. Falls x_k die Wurzel ist, dann tritt Ereignis $k \uparrow i$ ein und somit tritt Ereignis $i \uparrow k$ garantiert nicht ein. Außerdem hat x_i dann auch nicht die kleinste Priorität in $S(i, k)$, denn diese hat x_k.
3. Falls ein anderer Knoten x_j die Wurzel ist, dann gibt es zwei Unterfälle:
   - Falls x_i und x_k in unterschiedlichen Teilbäumen liegen, dann muss entweder $i < j < k$ oder $i > j > k$ gelten, d.h. $x_j \in S(i, k)$. In diesem Fall treten beide Ereignisse $i \uparrow k$ und $k \uparrow i$ nicht ein. Außerdem hat x_i nicht die kleinste Priorität in $S(i, k)$, die hat nämlich x_j.
   - Falls x_i und x_k im selben Teilbaum liegen, dann folgt die Aussage per Induktion, da der Teilbaum einfach als ein kleinerer Treap betrachtet werden kann und der leere Treap als Induktionsanfang dient. □

Da jeder Knoten in $S(i, k)$ gleich wahrscheinlich die kleinste Priorität hat[86] und da in $S(i, k)$ genau $k - i + 1$ bzw. $i - k + 1$ viele Knoten enthalten sind, gilt für die gesuchte Wahrscheinlichkeit folgendes:

$$Pr[i \uparrow k] = \begin{cases} \frac{1}{k-i+1}, & \text{falls } i < k \\ 0, & \text{falls } i = k \\ \frac{1}{i-k+1}, & \text{falls } i > k. \end{cases}$$

[86] Dies gilt, dass die Prioritäten unabhängig und gleichverteilt vergeben werden.

Um nun $E[t(x_k)]$ zu berechnen, müssen wir nur noch die obigen Wahrscheinlichkeiten einsetzen:

$$\begin{aligned} E[t(x_k)] = \sum_{i=1}^{n} Pr[i \uparrow k] &= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\ &= \sum_{j=2}^{k} \frac{1}{j} + \sum_{\ell=2}^{n-k+1} \frac{1}{\ell} \\ &= H_k - 1 + H_{n-k+1} - 1 \\ &< \ln k + \ln(n-k+1) \\ &< 2 \ln n. \end{aligned}$$

Hier haben wir in der zweiten Zeile lediglich die Summen anders aufgeschrieben. In der dritten Zeile haben wir die *Harmonische Reihe* eingesetzt: Die k-te harmonische Zahl ist $H_k = \sum_{q=1}^{k} \frac{1}{q}$. Die harmonischen Zahlen wachsen etwa so schnell wie der natürliche Logarithmus: Es gilt $\ln(k+1) < H_k < \ln k + 1$.
Somit haben wir gezeigt, dass jeder Knoten im Treap eine erwartete Tiefe in $\mathcal{O}(\log n)$ hat. □

5.3.5 Zusammenfassung

Wenn wir ein Dictionary mit Hilfe eines Treaps implementieren, dann entstehen die folgenden *erwarteten* Kosten:

	sortiertes Array	binärer Suchbaum	AVL-Baum	Treap
Search	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$^*\mathcal{O}(\log n)$
Insert	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$^*\mathcal{O}(\log n)$
Delete	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$^*\mathcal{O}(\log n)$

Hierbei steht $^*\mathcal{O}$ für eine obere Schranke der erwarteten Kosten. Treaps sind somit eine deutlich einfacher zu implementierende und trotzdem effiziente Alternative zu AVL-Bäumen. Aber Achtung, sollte es für die Anwendung wichtig sein, dass die worst-case Kosten garantiert in $\mathcal{O}(\log n)$ sind, dann sind AVL-Bäume zu bevorzugen.

5.3.6 Einsatzmöglichkeiten von Treaps

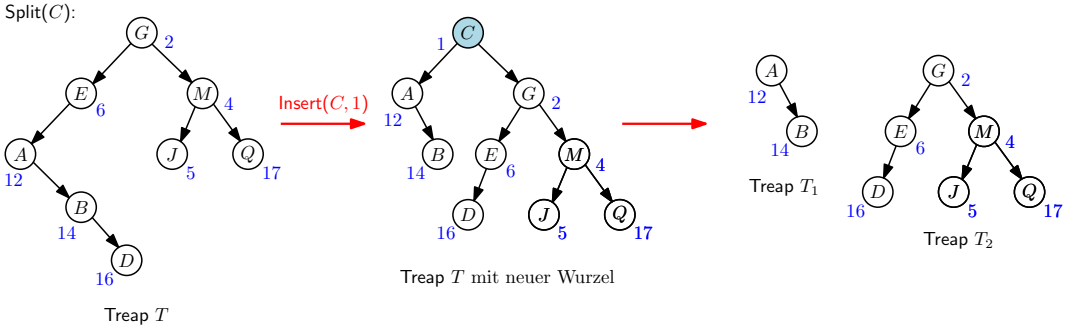
Treaps sind sehr vielfältig einsetzbar - nicht nur als Dictionary - und können als das schweizer Taschenmesser der Datenstrukturen[87] betrachtet werden. Um dies zu sehen, werden wir Treaps zunächst noch um die zwei Operationen Split und Merge erweitern.

Split eines Treaps Bei der Operation Split(x) wird ein Treap T in zwei Treaps T_1 und T_2 aufgespaltet, wobei in T_1 nur Schlüssel enthalten sind, die kleiner oder gleich x sind und in T_2 sind nur Schlüssel enthalten, die größer als x sind.
Die Operation Split(X) funktioniert wie folgt:

[87] Eierlegende Wollmilchsau der Datenstrukturen würde auch gut passen.

- Füge ein neues Element mit Schlüssel X und Priorität kleiner als die Priorität der Wurzel von T in T ein.
- Das neue Element mit Schlüssel X wird zur Wurzel des neuen Treaps.
- Der linke Teilbaum der neuen Wurzel ist dann ein Treap T_1, der alle Schlüssel enthält, die kleiner oder gleich x sind. Der rechte Teilbaum ist dann ein Treap T_2, der alle Schlüssel enthält, die größer als X sind.
- Nun können T_1 und T_2 als eigenständige Treaps genutzt werden (und die neue Wurzel kann wieder gelöscht werden).

Hier ein Beispiel:

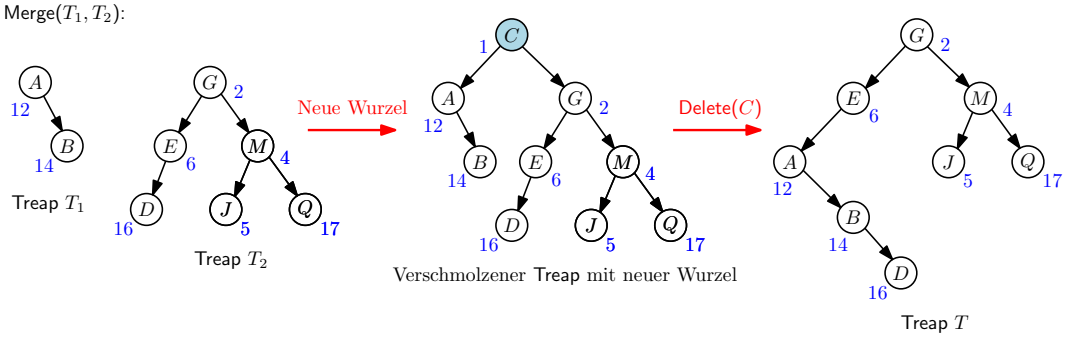


Die Kosten von Split werden von den Kosten von Insert dominiert. Somit sind die erwarteten Kosten von Split in $\mathcal{O}(\log n)$.

Merge zweier Treaps Die Operation Merge verschmilzt zwei Treaps T_1 und T_2 zu einem neuen Treap T, der alle Schlüssel von T_1 und T_2 enthält. Wichtig ist hierbei, dass alle Schlüssel in T_1 kleiner als die Schlüssel in T_2 sind! D.h. Merge kann ideal auf das Ergebnis einer Split Operation angewendet werden.
Merge funktioniert einfach wie ein rückwärts ausgeführtes Split:

- Füge eine neue Dummy-Wurzel ein, die einen Schlüssel hat, der größer oder gleich der größte Schlüssel von T_1 und kleiner als der kleinste Schlüssel von T_2 ist. Die Priorität der neuen Wurzel muss kleiner als die Prioitäten der Wurzeln von T_1 und T_2 sein.
- Entferne die neue Wurzel per Delete.

Hier ein Beispiel:



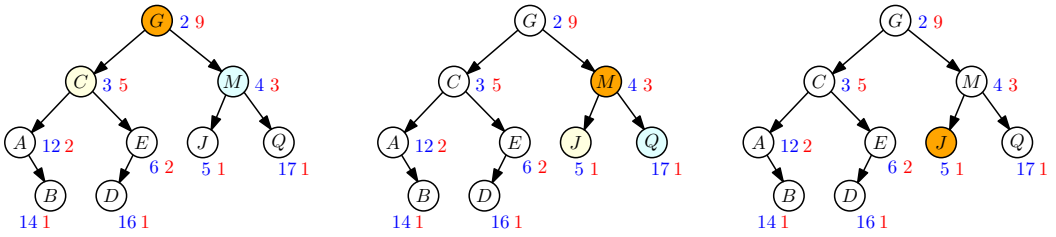
Die erwarteten Kosten von Merge sind in $\mathcal{O}(\log n)$.

Dynamisches sortiertes Array mit Range Queries

Treaps können leicht als dynamisches sortiertes Array eingesetzt werden. Man kann aber zusätzlich noch schnell auf bestimmte Elemente zugreifen.

Finden des k-kleinsten Elements in $^*\mathcal{O}(\log n)$: Um das k-kleinste Element in einem Treap zu finden, müssen wir noch eine kleine Modifikation einführen. Wir speichern in jedem Knoten nicht nur den Schlüssel und die Priorität, sondern wir merken uns auch die Anzahl der Knoten im Teilbaum dieses Knotens. Diese Teilbaumgrößen können leicht bei allen anderen Operationen mit aktualisiert werden.
Das k-kleinste Element kann dann wie folgt gefunden werden: Ausgehend von der Wurzel wird anhand der gespeicherten Teilbaumgrößen das entsprechende Element gesucht. Hier ein Beispiel:

Finde 7. kleinstes Element:



Vergleiche Teilbaumgrößen der Kinder der Wurzel. Suche entsprechend weiter.

Suche kleinstes Element im Teilbaum mit Wurzel M.

7. kleinstes Element gefunden!

Ausgeben aller Elemente zwischen L und R: Mit Treaps kann man leicht sogenannte *Range Queries* beantworten. Bei einer Range Query werden zwei Werte L und R vorgeben und es sollen alle in der Datenstruktur enthaltenen Elemente X mit $L \leq X \leq R$ ausgegeben werden. Für statische Daten kann man dies leicht per binärer Suche in $\mathcal{O}(\log n + k)$, wobei k die Anzahl der Werte zwischen L und R ist, realisieren. Für dynamische Daten können wir dieselben Kosten mit einem Treap erreichen:

1. Führe Split(L) aus und wähle dann den Treap T_2, der alle Schlüssel größer als L enthält.

2. Führe Split(R) auf T_2 aus. Betrachte dann den Treap T_{2_1}, der alle Schlüssel kleiner oder gleich R enthält.

3. Falls L die Wurzel von T_1 ist, gib L aus. Gib außerdem die InOrder-Traversierung der Schlüssel in T_{2_1} aus.

Treaps als spezielle Priority Queues

Neben Dictionaries können Treaps auch leicht als Priority Queue verwendet werden[88].
Elemente können einfach per Insert eingefügt werden. Ein ExtractMin wird durch die Suche nach dem kleinsten Schlüssel und dessen Entfernen realisiert. Ein DecreaseKey kann durch Entfernen des alten Schlüsselwerts und wiedereinfügen des neuen Schlüsselwerts

[88] Natürlich gilt das auch für AVL-Bäume

umgesetzt werden. Alle drei Standardoperationen einer Priority Queue können somit in ${}^*\mathcal{O}(\log n)$ ausgeführt werden.
Aber Treaps können noch mehr:

Sortierte Ausgabe der k kleinsten Elemente: Für manche Anwendungen ist es evtl. wichtig die k Elemente mit kleinster Priorität auszugeben. Alle bisher betrachteten Implementierungen von Priority Queues können dies mit k maligem ExtractMin in $\mathcal{O}(k \log n)$ schaffen. Mit Treaps geht das wie folgt sehr einfach in $\mathcal{O}(\log n + k)$:

- Suche das k-kleinste Element. Dies sei X.
- Führe Split(X) aus und wähle den erzeugten Treap T_1 der alle Schlüssel kleiner oder gleich X enthält.
- Traversiere T_1 in InOrder und gib die Elemente entsprechend aus.

Treaps zur Mengenverwaltung

Mit Treaps können leicht die Mengenoperationen Vereinigung, Schnitt und Mengendifferenz realisiert werden. Wir gehen im Folgenden davon aus, dass die Element der Menge A im Treap T_A und die Elemente der Menge B im Treap T_B gespeichert sind. Außerdem nehmen wir an, dass die Operation Split die Wurzeln der beiden erzeugten Treaps zurückliefert.

Vereinigung, $A \cup B$:

Union(T_A, T_B)

Input: Treaps T_A und T_B
Output: Treap $T_{A \cup B}$

if T_A ist leer **then**
  return T_B
end if
if T_B ist leer **then**
  return T_A
end if
if Priorität der Wurzel von $T_A >$ Priorität der Wurzel von T_B **then**
  Vertausche T_A und T_B
end if
$T_<, T_> \leftarrow$ führe auf T_B Split(Schlüssel der Wurzel von T_A) aus
return Treap $T_{A \cup B}$ mit Wurzel von T_A; linker Teilbaum wird durch Merge von linker Teilbaum von T_A und $T_<$ erzeugt; rechter Teilbaum wird durch Merge von rechter Teilbaum von T_A und $T_>$ erzeugt

Die Operationen $A \cap B$ und $A \setminus B$ können ähnlich realisiert werden. Außerdem kann gezeigt werden, dass die Kosten für alle drei Operationen in ${}^*\mathcal{O}(m \log \frac{n}{m})$ liegen, wobei m und n die Anzahl der Elemente der Treaps T_A und T_B, mit $m \leq n$ ist[89]. Alle drei Operationen sind zudem sehr gut parallelisierbar.

[89] Man kann sogar zeigen, dass diese Kosten optimal sind!

6 Dynamisches Programmieren

Wir wenden uns nun einem der mächtigsten algorithmischen Werkzeuge überhaupt zu: Dem *dynamischen Programmieren*. Der Name ist etwas irreführend und ist darauf zurückzuführen, dass vor langer Zeit "programmieren" mehr oder weniger nur für das Ausfüllen von Tabellen stand. Eigentlich geht es beim dynamischen Programmieren nur um clevere Rekursion. Wir betreten nun also (erneut) die Welt der rekursiven Algorithmen[90].

6.1 Berechnung der n-ten Fibonacci-Zahl

Wir starten mit folgendem Problem:

Gegeben: Eine positive Zahl n.

Aufgabe: Berechne die n-te Fibonaccizahl.

Zur Erinnerung, die Fibonaccizahlen sind wie folgt definiert:

$$F_0 = 0, F_1 = 1 \text{ und für alle } i \geq 2 : F_i = F_{i-1} + F_{i-2}.$$

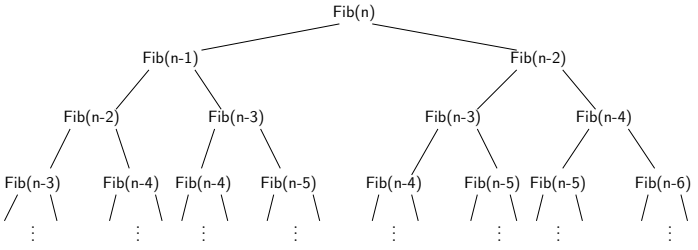
6.1.1 Naiver Ansatz:

Wir können einfach die obige Definition direkt in einen rekursiven Algorithmus Fib umwandeln:

Fib(n)

Input: positive Zahl n
Output: F_n

```
1: if n < 2 then
2:     return n
3: else
4:     return Fib(n − 1) + Fib(n − 2)
5: end if
```



Dabei stellen wir fest, das extrem viele der rekursiven Aufrufe mehrfach vorkommen. Würden wir die Aufrufe jedes mal wieder neu berechnen, dann erhalten wir folgende Rekurrenz für die Kosten von Fib:

$$T(0) = 0, T(1) = 1 \text{ und für alle } n \geq 2\text{: } T(n) = T(n-1) + T(n-2) + \mathcal{O}(n),$$

wobei hier der letzte Term in $T(n)$ die Kosten für die Addition von zwei Zahlen mit $\mathcal{O}(n)$ Bits darstellt[91]. Wir wissen schon, dass die Fibonaccizahlen exponentiell in n wachsen. Somit benötigt die n-te Fibonaccizahl etwa $\log(\phi^n) \in \Theta(n)$ viele Bits.
Wir können die Kosten mit $T(n) \in \Theta(\phi^n)$ abschätzen[92] und sehen damit, dass Fib exponentielle Kosten hat. Der direkte rekursive Algorithmus ist also keine gute Idee!

[90] MergeSort, QuickSort und binäre Suche sind rekursive Algorithmen, die wir bereits angetroffen haben.
[91] Wir wenden hier das *logarithmische Kostenmaß* an, da die Zahlen extrem schnell wachsen und hier eine Addition mit konstanten Kosten abzuschätzen, eher irreführend ist.
[92] Hierbei ist ϕ der goldene Schnitt. Die Abschätzung ist eine gute Übungsaufgabe!

6.1.2 Memoization (Top-Down)

Der Trick, um polynomielle Kosten zu erreichen, ist, die mehrfach vorkommenden rekursiven Aufrufe nur ein mal zu berechnen, da man sich das Ergebnis eines Aufrufs einfach für später speichern kann. Vom Prinzip her, gehen wir weiterhin top-down vor. Um uns etwas Schreibarbeit zu sparen, nehmen wir hierbei an, dass wir ein globales Array F mit n Einträgen haben. Anfangs ist das Array mit überall 0 initialisiert.

FibMemo(n)

Input: positive Zahl n
Output: F_n

```
1: if n < 2 then
2:     return n
3: else
4:     if F[n] = 0 then
5:         F[n] ← FibMemo(n − 1) + FibMemo(n − 2)
6:     end if
7:     return F[n]
8: end if
```

Jeder Zugriff auf eine bereits berechnete Fibonaccizahl kostet nur $\mathcal{O}(1)$. Insgesamt werden n Fibonaccizahlen berechnet und jedes mal werden dabei zwei Zahlen mit höchtens n vielen Bits addiert. Wir erhalten somit Gesamtkosten in $\mathcal{O}(n^2)$, d.h. wir haben eine exponentielle Verbesserung gegenüber Fib erzielt!
Der Algorithmus FibMemo hat allerdings noch einige Nachteile. Er ist weiterhin rekursiv und somit entstehen zusätzlich zu den Speicherkosten für das Array F auch noch die Kosten für das Verwalten des Rekursionsstacks. Dieser kann hier zwar nur maximal n Einträge enthalten, doch generell müssen wir bei solchen Verfahren aufpassen, dass der Rekursionsstack nicht überläuft.

6.1.3 Bottom-Up dynamische Programmierung

Noch eleganter und Speichereffizienter geht es mit der iterativen Version von FibMemo. Im Unterschied zum rekursiven Algorithmus gehen wir hierbei nicht top-down sondern bottom-up vor. Dazu müssen wir uns nur noch eine gülte Abarbeitungsreihenfolge der Aufrufe überlegen, so dass zu jeder Zeit alle benötigten Subaufrufe bereits berechnet zur Verfügung stehen.
Eine solche Abarbeitungsreihenfolge findet sich leicht: Wir berechnen zunächst den Eintrag von $F[2]$, danach den Eintrag von $F[3]$, dann $F[4]$ usw. Auf diese Weise sind immer alle benötigen Subprobleme bereits gelöst.

FibBottomUp(n)

Input: positive Zahl n
Output: F_n

```
1: Initialisiere n + 1-Array F mit 0
2: F[0] ← 0 ; F[1] ← 1
3: if n < 2 then
4:     return F[n]
5: end if
6: for i = 2 to n do
7:     F[i] ← F[i − 1] + F[i − 2]
8: end for
9: return F[n]
```

Die Kosten für FibBottomUp sind weiterhin in $\mathcal{O}(n^2)$, doch wir benötigen etwa nur halb so viel Speicherplatz und müssen den Rekursionsstack nicht verwalten. Allerdings benötigen wir für das Array F noch immer $\Theta(n)$ zusätzlichen Speicherplatz[93].

Noch weniger Speicherplatz: Der Algorithmus FibBottomUp speichert alle Zwischnergebnisse, benötigt für jeden Schritt jedoch nur genau zwei. Folglich genügt es, die letzten beiden Zwischenergebnisse zu speichern. Dies spart nochmal einen Faktor n bei den Speicherkosten.

Noch schneller: Der Algorithmus FibBottomUp ist nicht der beste bekannte Algorithmus zur Berechnung der n-ten Fibonaccizahl. Mit cleverer Matrixmultiplikation sind Kosten in $\mathcal{O}(n \log^2 n \cdot 2^{\log^* n})$ möglich.

Korrektheit: Die Korrektheit der obigen Algorithmen ist offensichtlich, da wir direkt die rekursive Definition der Fibonaccizahlen umgesetzt haben.

6.2 Druckjobs planen - revisited

Wir erinnern uns an das algorithmische Problem in der Großdruckerei, die einen Spezialdrucker so auslasten möchte, um damit möglichst viele Jobs abzuarbeiten. Jeder Job hat eine Startzeit und eine Druckdauer und somit können wir annehmen, dass wir für jeden Druckjob eine Start- und eine Endzeit haben.
Der Drucker kann pro Zeitpunkt immer nur einen Job abarbeiten. Falls sich mehrere Jobs zu einem Zeitpunkt überlappen, dann müssen alle bis auf einen abgelehnt werden - es gibt somit keine Warteschlange.
Als wir das Szenario beim ersten Mal angetroffen haben, wollten wir nur so viele Druckjobs wie möglich abarbeiten und haben dafür einen Greedy-Algorithmus entwickelt. Nun betrachten wir eine Variante des Problems, bei der uns Greedy-Algorithmen nicht helfen (siehe Übung) und wir die Stärke von dynamischer Programmierung benötigen.
Der Unterschied ist nun, dass jeder Job i auch noch einen beliebigen Wert $w(i)$ hat. Das bekannte Druckjobplanungsproblem ist somit der Spezialfall, in dem alle Jobs den selben Wert haben.

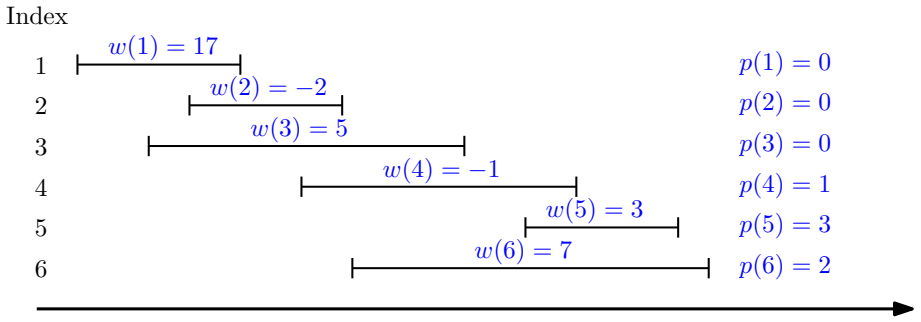
[93] Um genau zu sein, benötigen wir $\Theta(n^2)$ viel Speicherplatz im logarithmischen Kostenmaß, da wir n Zahlen mit jeweils etwa n Bits speichern müssen.

Geg: n Druckjobs mit Start- und Endzeiten $(s(i), e(i))$, für $1 \leq i \leq n$, sowie einem Wert $w(i)$.

Aufgabe: Finde die Menge von sich paarweise nicht überlappenden Jobs, die den Gesamtwert maximiert.

6.2.1 Ein rekursiver Ansatz

Als erster Schritt auf dem Weg zu einer Lösung per dynamischen Programmierung benötigen wir einen Ansatz, wie wir unser Problem rekursiv formulieren können. Dieser Anfangsschritt ist entscheidend, denn eigentlich steckt hier die ganze Stärke der dynamischen Programmierung: Wenn wir einen cleveren rekursiven Ansatz finden, dann können wir diesen per Memoization oder bottom-up Vorgehen gut ausnutzen, um uns unnötige rekursive Aufrufe zu sparen.
Zunächst benötigen wir etwas Notation. Wir nehmen an, dass die Durckjobs aufsteigend nach Endzeitpunkt $e(i)$ sortiert sind. Für jeden Druckjob j mit Endzeitpunkt $e(j)$ interessieren wir uns für den Job $p(j) = i$, der als letztes vor dem Start von Job j, d.h. vor $s(j)$, fertig wird, falls wir j auswählen. Falls es für einen Job j keinen solchen Job i gibt, dann definieren wir $p(j) = 0$.



Nun kommen wir zur rekursiven Formulierung. Wir betrachten dafür die bestmögliche Jobmenge OPT und überlegen uns, wie diese aussehen muss.
Betrachten wir den Job n, der als letztes fertig wird:

- Falls $n \in OPT$, dann kann kein Job in OPT sein, welcher sich mit Job n überschneidet. Außerdem muss OPT dann zusätzlich zum Job n die sich paarweise nicht überlappende Jobmenge auswählen, die den Gesamtwert maximiert, wenn nur die Jobs $\{1, 2, \ldots, p(n)\}$ zur Verfügung stehen.

- Falls $n \notin OPT$, dann entspricht OPT der optimalen Lösung, wenn nur die Jobs aus der Menge $\{1, 2, \ldots, n-1\}$ zur Verfügung stehen.

Uns interessieren also Lösungen, die für einen Präfix der Jobreihenfolge optimal sind. Sei OPT_j die optimale Lösung, wenn nur aus den Jobs $\{1, 2, \ldots, j\}$ ausgewählt werden darf. Somit gilt $OPT = OPT_n$.
Für die optimale Teillösung OPT_j können wir die obige Argumentation wieder anwenden:

- Falls $j \in OPT_j$, dann kann kein Job in OPT_j sein, welcher sich mit Job j überschneidet. Außerdem muss OPT dann zusätzlich zum Job j die sich paarweise nicht überlappende Jobmenge auswählen, die den Gesamtwert maximiert, wenn nur die Jobs $\{1, 2, \ldots, p(j)\}$ zur Verfügung stehen.

- Falls $j \notin OPT_j$, dann entspricht OPT_j der optimalen Lösung, wenn nur die Jobs aus der Menge $\{1, 2, \ldots, j-1\}$ zur Verfügung stehen.

Da es für Job j nur beiden obigen Möglichkeiten gibt, muss somit die optimale Lösung für Job j die bessere der beiden Optionen wählen. Sei $w(OPT_j)$ der Gesamtwert der Lösung OPT_j. Somit gilt:

$$w(OPT_j) = \max\{w(j) + w\left(OPT_{p(j)}\right), w\left(OPT_{j-1}\right)\}.$$

Falls $w(OPT_j) = w(j) + w\left(OPT_{p(j)}\right)$, dann gilt $OPT_j = \{j\} \cup OPT_{p(j)}$, sonst gilt $OPT_j = OPT_{j-1}$.
Somit gilt: Job j gehört genau dann zu OPT_j, wenn $w(j) + w\left(OPT_{p(j)}\right) \geq w\left(OPT_{j-1}\right)$.

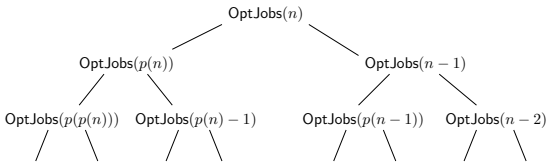
6.2.2 Vom rekursiven Ansatz zum rekursiven Algorithmus

Wir wandeln den rekursiven Ansatz direkt in den rekursiven Algorithmus OptJobs um:

OptJobs(n)

Input: Jobs $1, 2, \ldots, n$; $p()$ bereits vorberechnet
Output: OPT_n

```
 1: if n = 0 then
 2:     return ∅
 3: else
 4:     OPT_{p(n)} ← OptJobs(p(n))
 5:     OPT_{n-1} ← OptJobs(n − 1)
 6:     if w(n)+w(OPT_{p(n)}) ≥ w(OPT_{n-1}) then
 7:         return {n} ∪ OPT_{p(n)}
 8:     else
 9:         return OPT_{n-1}
10:     end if
11: end if
```



Der obige Algorithmus ist offensichtlich korrekt (die Korrektheit folgt direkt aus unserem rekursiven Ansatz), doch ähnlich wie bei den Fibonaccizahlen haben wir hier auch wieder exponentielle Kosten. Auch hier liegt der Grund für die Kostenexplosion darin, dass viele Teilprobleme mehrfach berechnet werden. Wir nutzen also wieder die Technik der *Memoization*, d.h. wir merken uns einfach die Teillösungen.

6.2.3 Memoization (Top-Down)

Wir legen uns zwei globale Hilfsarrays der Länge $n + 1$ an. Das Array O speichert an Position j die Menge OPT_j und das Array G speichert an Position j den Gesamtwert $w(OPT_j)$. Anfangs sind beide Arrays mit `null` initialisiert. Die Einträge mit Index 0 stehen für die leere Menge.

OptJobsMemo(n)

Input: Jobs $1, 2, \ldots, n$; $p()$ bereits vorberechnet
Output: OPT_n

```
1: if n = 0 then
2:     O[0] ← ∅; G[0] ← 0
3:     return ∅
4: else if O[n] ≠ null then
5:     OPT_{p(n)} ← OptJobsMemo(p(n)); OPT_{n−1} ← OptJobsMemo(n − 1)
6:     if w(n) + G[p(n)] ≥ G[n − 1] then
7:         O[n] ← {n} ∪ O[p(n)]; G[n] ← w(n) + G[p(n)]
8:     else
9:         O[n] ← O[n − 1]; G[n] ← G[n − 1]
10:    end if
11: else
12:    return O[n]
13: end if
```

Das Nutzen der Memoization-Technik hat auch hier drastische Auswirkungen. Die Gesamtkosten fallen von exponentiellen Kosten auf lineare Kosten ab!
OptJobsMemo hat Gesamtkosten in $\mathcal{O}(n)$, da jeder der Werte im Array O bzw. G nur genau einmal berechnet wird. Später werden die entsprechenden Werte per Arrayzugriff bereitgestellt.
Auch hier gilt, dass wir neben den Arrays O und G auch den Rekursionsstack speichern und verwalten müssen. Schauen wir uns deshalb auch die bottom-up Version an.

6.2.4 Bottom-Up

Wieder überführen wir unseren rekursiven Algorithmus in einen iterativen Algorithmus. Wir wählen hier als Abarbeitungsreihenfolge der Subaufrufe einfach $0, 1, 2, \ldots, n$. Dies ist eine zulässige Reihenfolge, da wir bei Berechnung von OPT_j nur auf bereits berechnete Werte von OPT_i für $i < j$ zurückgreifen.

OptJobsBottomUp(n)

Input: Jobs $1, 2, \ldots, n$; $p()$ bereits vorberechnet
Output: OPT_n

```
1: initialisiere n + 1-Arrays O und G überall mit null
2: O[0] ← ∅; G[0] ← 0
3: if n = 0 then
4:     return ∅
5: end if
6: for i = 1 to n do
7:     if w(i) + G[p(i)] ≥ G[i − 1] then
8:         O[i] ← {i} ∪ O[p(i)]; G[i] ← w(i) + G[p(i)]
9:     else
10:        O[n] ← O[n − 1]; G[n] ← G[n − 1]
11:    end if
12: end for
13: return O[n]
```

Gesamtkosten: In allen Algorithmen haben wir angenommen, dass die Jobs bereits nach Endzeitpunkt sortiert sind. Somit müssen wir im worst-case noch zusätzlich die Jobs sortieren. Abgesehen davon hat auch OptJobsBottomUp nur Kosten in $\mathcal{O}(n)$.

Weniger Speicherplatz? Im Unterschied zur Berechnung der Fibonaccizahlen können wir hier den zusätzlichen Speicherplatz *nicht* auf $\mathcal{O}(1)$ senken. Der Grund hierfür ist, dass wir nichts über die Vorgänger $p()$ wissen.

6.3 Das SubsetSum-Problem

Wir betrachten hier ein weiteres bekanntes Problem, welches notorisch schwierig ist. Das Problem ist wie folgt definiert:

> **Geg.:** Eine Menge A von natürlichen Zahlen $a_1, \ldots, a_n$ und eine natürliche Zahl s.
>
> **Aufgabe:** Finde eine Menge $I \subseteq A$, so dass $\sum_{i \in I} a_i = s$ gilt.

Gegeben ist zum Beispiel folgende Instanz:

$$A = 25 \quad 9 \quad 11 \quad 43 \quad 108 \quad 112 \quad 29 \quad 37; \quad s = 155$$

Dies ist keineswegs ein künstliches Problem, denn es tritt zum Beispiel beim Verteilen von Arbeitslast auf zwei Prozessoren auf. Soll die Arbeit genau gleich aufgeteilt werden, also für den Spezialfall $s = \frac{1}{2} \sum_i a_i$, nennt man dies auch Partition.
Es ist kein leichtes Problem, denn wir sehen schon bei unserem kleinen Beispiel von oben nicht sofort eine Lösung. Übrigens gilt, dass sowohl SubsetSum als auch Partition NP-schwer sind - d.h. es gibt keinen Polynomialzeitalgorithmus für beide Probleme, es sei denn, dass $P = NP$ gilt[94].

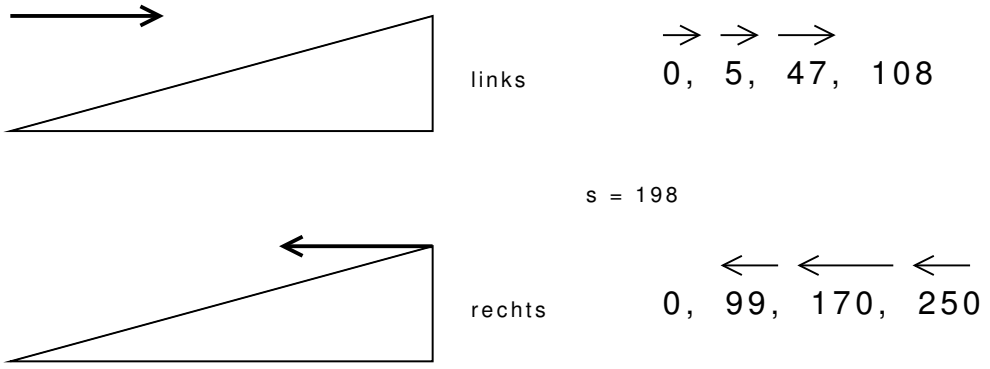
6.3.1 Naiver Ansatz

Wir prüfen für jede Teilmenge von A, ob die Summe der enthaltenen Elemente gleich dem Wert von s ist. Falls wir eine solche Teilmenge finden, dann geben wir diese aus. Falls alle Teilmengen getestet wurden und keine zur Lösung führte, dann wissen wir, dass es keine Teilsumme für s gibt. Es gibt 2^n Teilmengen und wir brauchen ungefähr linearen Aufwand um die Summe zu bilden und somit erhalten wir eine Laufzeit in $\mathcal{O}(n \cdot 2^n)$. Dieser Ansatz wirkt sehr naiv, aber wir werden sehen, dass wir diesen nicht allzu stark verbessern können.
Bei solchen hohen Laufzeiten lohnt es sich schon die Basis des exponentiellen Terms geringfügig zu verbessern, denn es gilt zum Beispiel $2^n \notin \mathcal{O}(1.99^n)$.

[94] Genauer gesagt, sind beide Probleme schwach NP-schwer. D.h. es gibt Polynomialzeitalgorithmen für beide, wenn die vorkommenden Zahlen nur höchstens polynomiell in n groß sind.

6.3.2 Semi-naiver Ansatz

Wir halbieren A und erhalten somit zwei Mengen der Größe $\frac{n}{2}$. Dann ermitteln wir alle Teilsummen im linken Teil und alle Teilsummen im rechten Teil. Dabei kann es passieren, dass schon eine Lösung gefunden wird, aber im allgemeinen wird das nicht vorkommen. Die Teilsummen können wir nun aber sortieren und wir tun dies jeweils für die linke und die rechte Seite und erhalten somit zwei sortierte Teilfolgen der Länge $2^{\frac{n}{2}}$.
Nun können wir, ähnlich wie bei merge von MergeSort, durch die beiden Folgen gehen und prüfen, ob die richtige Teilsumme enthalten ist. Dabei gehen wir in der linken Folge von links nach rechts und in der rechten Folge von rechts nach links. Falls die Summe der beiden betrachteten Elemente kleiner als s ist, dann gehen wir links einen Schritt weiter und falls sie größer ist, machen wir in der rechten Folge einen zusätzlichen Schritt.



Nach diesem Schritt erhalten wir auf alle Fälle das korrekte Ergebnis (Siehe Übung).

Gesamtkosten: Wir überschlagen grob die benötigten Schritte:

$$\underbrace{2^{\frac{n}{2}}}_{\text{Teilsummen links}} + \underbrace{2^{\frac{n}{2}}}_{\text{Teilsummen rechts}} + \underbrace{2^{\frac{n}{2}} * \underbrace{\log 2^{\frac{n}{2}}}_{O(n)}}_{\text{sort links}} + \underbrace{2^{\frac{n}{2}} * \underbrace{\log 2^{\frac{n}{2}}}_{O(n)}}_{\text{sort rechts}} + \underbrace{2 * 2^{\frac{n}{2}}}_{merge}$$

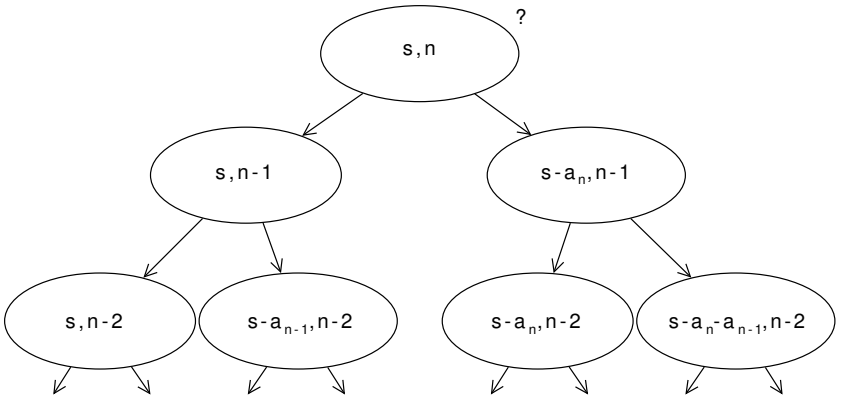
Total ergibt das $\mathcal{O}(n \cdot 2^{\frac{n}{2}}) = \mathcal{O}(n \cdot \sqrt{2^n})$. Dies ist ein gewaltiger Fortschritt und es ist auch die beste Schranke, die man überhaupt bisher für das Problem kennt.

6.3.3 SubsetSum via Dynamisches Programmieren

Als nächstes betrachten wir eine Lösung per dynamischem Programmieren.
Ein einfacher rekursiver Ansatz lässt sich wie folgt beschreiben:

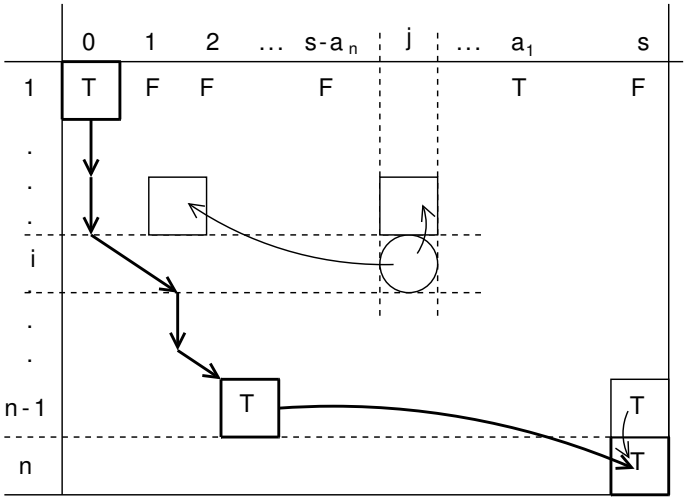
> Wir erreichen Summe s mit den ersten i Elementen genau dann, wenn
>
> - wir Summe s mit den ersten $i-1$ Elementen erreichen *oder*
> - wir Summe $s - a_i$ mit den ersten $i-1$ Elementen erreichen.
>
> Im ersten Fall wird das Element a_i nicht gewählt, wohingegen es im zweiten Fall in der Teilsumme vorkommt.

Eigentlich wollen wir wissen, ob wir Summe s mit den ersten n Elementen erreichen können. Der zugehörige Rekursionsbaum sieht dann wie folgt aus:



Wir brechen ab, wenn nur noch ein oder kein Element mehr in der betrachteten Menge der Elemente vorhanden ist. Im obigen Baum evtl. wieder sehr viele Aufrufe mehrfach vor. Hier bietet es sich wieder an, die jeweiligen Zwischenergebnisse zu speichern, um dann später schnell darauf zurückgreifen zu können.

Bottom-Up Ansatz: Wir springen diesmal direkt zum bottom-up Ansatz.
Wir legen uns eine Tabelle an, in welcher die Zellen sind mit den Wahrheitswerten `true` oder `false` gefüllt und die Zelle (i, j) enthält den Wahrheitswert, ob der Wert j mit einer Teilmenge der ersten i Zahlen erzeugt werden kann. Unten rechts, d.h. Zelle (n, s), steht dafür, ob die gesuchte Summe s mit einer Teilmenge der ersten n Zahlen erreicht werden kann. Hier können wir später das Ergebnis ablesen.



Ein Eintrag in jeder Zelle hängt nach Vorschrift der Rekursion vom Eintrag in genau zwei anderen Zellen, sofern diese existieren, ab und der Wert wird mit dem logischen Oder gebildet. Somit wird der Wahrheitswert einer Zelle nur dann auf `false` gesetzt, wenn beide eingehenden Werte `false` sind.
Wenn wir die beiden eingehenden Werte haben, dann müssen wir nur konstante Zeit aufwenden, um das logische Oder zu berechnen. Falls wir aus der Tabelle heraus rutschen, dann merken wir dies auch in konstanter Zeit.

Initialisierung der Tabelle: Wenn wir eine Zeile berechnet haben, dann können wir die ganzen restlichen Einträge darunter berechnen. Um die erste Zeile auszufüllen benutzen wir das erste Element.
Gesamtkosten: Jeder Tabelleneintrag hat konstante Kosten und wir haben $n \cdot s$ viele Einträge. Dies ergibt Kosten $\mathcal{O}(n \cdot s)$ und das ist eine gute Schranke, wenn s klein ist.
Solche Gesamtkosten nennt man *pseudopolynomiell*. Falls s ein Polynom in n ist, dann ergibt sich polynomielle Laufzeit[95]. Dies ist dann ein sogenannter „pseudopolynomieller Algorithmus“, d.h. wenn nur kleine (d.h. polynomiell große) Zahlen beteiligt sind, dann ist auch die Laufzeit des Verfahrens durch ein Polynom beschränkt. NP-schwere Probleme, für die das gilt, nennt man deshalb auch “schwach NP-schwer”.
Ist die Summe s allerdings nicht klein (z.B. exponentiell in n), dann erhalten wir eine schlechte Laufzeit. Wir müssen also bei der Bewertung von Verfahren mit solchen zusammengesetzten Laufzeiten aufpassen.
Ermittlung der Summanden: Unsere ursprüngliche Frage war die nach der Indexmenge, doch das können wir an der Tabelle ablesen. Es gibt einen Pfad von einem `true` aus der ersten Zeile zum `true` ganz rechts unten, welcher nur aus `true`-Werten besteht. Falls dieser Pfad nach rechts abbiegt, dann wird das jeweilige Element gewählt, falls der Pfad senkrecht verläuft, dann wird das Element nicht in die Teilsumme einbezogen.
Die Beschreibung der Lösung erhalten wir also durch „Rückverfolgen“ im Tableau.

6.4 Maximum Independent Set auf Bäumen

Wir betrachten noch eine wichtige Variation des obigen Ansatzes für das dynamische Programmieren. Bisher hatten wir die Lösungen der Subprobleme in einem Array gespeichert. Für einige Probleme ist dies aber unnötig umständlich. Ein Beispiel für eine ganze Klasse solcher Probleme sind Optimimierungsprobleme auf Bäumen. Wir wählen hier als Repräsentant das Maximum Independent Set Problem auf Bäumen.
Zur Erinnerung, wir haben Maximum Independent Set bereits als Druckjobscheduling Problem auf Intervallgraphen[96] kennen gelernt. Für einen beliebigen Graphen G ist es wie folgt definiert:

> **Geg.:** Ein Graph G mit n Knoten.
>
> **Aufgabe:** Finde eine größtmögliche Teilmenge I der Knoten von G, so dass es keine Kante in G zwischen Knoten aus I gibt.

Die Menge I ist eine größtmögliche *unabhängige Knotenmenge*[97], d.h. eine Menge von Knoten, die keine zu sich benachbarten Knoten enthält.

Beobachtung 24. *Befindet sich als ein Knoten x in I, dann darf keiner der Nachbarn von x in I sein.*

[95] Achtung, dies steht nicht im Widerspruch zur NP-schwere des Problems!

[96] Ein Intervallgraph entsteht aus einer Menge von Intervallen, indem jedem Intervall ein Knoten zugeordnet wird und zwei Knoten genau dann durch eine Kante verbunden sind, wenn sich die zugehörigen Intervalle überlappen.

[97] Gelegentlich findet man in deutschsprachigen Lehrbüchern dafür den eher ungeeigneten Begriff “stabile Menge”.

Rekursiver Ansatz: Diese unscheinbare Beobachtung liefert uns sofort einen rekursiven Lösungsansatz für das Maximum Independent Set Problem: Für jeden Knoten x wählen wir entweder den Knoten x selbst und entfernen alle Nachbarn oder wir wählen x nicht und entfernen x aus dem Graph. Sei $N(x)$ die *Nachbarschaft von* x in G, d.h. $N(x)$ enthält x und alle Knoten, die eine Kante zu x haben[98].

MIS(G)

Input: ungerichteter Graph $G = (V, E)$
Output: Maximum Independet Set von G

```
1: if V = ∅ then
2:     return ∅
3: end if
4: x ← beliebiger Knoten von G
5: mitx ← {x} ∪ MIS(G \ N(x))
6: ohnex ← MIS(G \ {x})
7: if |mitx| ≥ |ohnex| then
8:     return mitx
9: else
10:    return ohnex
11: end if
```

Im Algorithmus MIS wird die Notation $G \setminus Y$ benutzt, wobei Y eine Menge von Knoten ist. Der Graph $G \setminus Y$ ist der Graph, den man erhält, wenn aus dem Graph G die Knoten in Y entfernt werden.

Korrektheit und Gesamtkosten: Wir zeigen zunächst, dass der obige Algorithmus korrekt ist. Üblicherweise erfolgt der Korrektheitsbeweis eines rekursiven Algorithmus per Induktion. Der Beweis ist extrem einfach und folgt mehr oder weniger direkt aus unserer rekursiven Idee.

Theorem 25. *Der Algorithmus MIS ist korrekt.*

Beweis. Wir beweisen die Korrektheit per Induktion über die Anzahl der Knoten in G.
Induktionsanfang: Wenn G überhaupt keine Knoten hat, dann ist das Maximum Independent Set von G die leere Menge und genau diese wird von MIS ausgegeben.
Induktionsschritt: Angenommen MIS ist korrekt für alle Graphen mit höchstens i Knoten. Wir zeigen nun, dass MIS dann auch für alle Graphen mit $i + 1$ Knoten korrekt ist. Sei G ein beliebiger Graph mit $i + 1$ Knoten und sei x ein beliebiger Knoten in G.

- Falls x in einem Maximum Independent Set I von G vorkommt, dann kann kein Nachbar von x in I sein. Alle anderen Knoten in I müssen somit im Graph $G \setminus N(x)$ liegen. Da der Graph $G \setminus N(x)$ weniger als $i + 1$ Knoten hat, folgt aus der Induktionsvoraussetzung, dass MIS$(G \setminus N(x))$ ein Maximum Independent Set I' von $G \setminus N(x)$ zurück liefert. Für G muss dann $\{x\} \cup I'$ ein Maximum Independet Set sein.

[98] Wir betrachten hier nur ungerichtete Graphen. Bei gerichteten Graphen gibt es eine ähnliche aber andere Definition der Nachbarschaft.

- Falls x in keinem Maximum Independet Set von G vorkommt, dann befinden sich in jedem Maximum Independet Set von G nur Knoten aus $V \setminus \{x\}$. Durch das Entfernen von x aus G werden nur Kanten mit Endpunkt x gelöscht, alle anderen Nachbarschaften bleiben erhalten. Folglich muss jedes Maximum Independet Set in $G \setminus \{x\}$ auch ein Maximum Independent Set für G sein. Da $G \setminus \{x\}$ weniger als $i+1$ Knoten hat, berechnet $\mathsf{MIS}(G \setminus \{x\})$ nach Induktionsvoraussetzung ein korrektes Maximum Independet Set in $G \setminus \{x\}$ und somit auch für G. □

Das Problem an MIS auf beliebigen Graphen G ist, dass die Gesamtkosten exponentiell in der Anzahl Knoten von G sind. (Siehe Übung.) Das ist nicht überraschend, denn Maximum Independent Set ist eins der bekanntesten NP-schweren Probleme. Es ist also unwahrscheinlich, dass wir einen Polynomialzeitalgorithmus für *beliebige* Graphen finden. Zur Erinnerung: Wir haben bereits Algorithmen mit Kosten in $\mathcal{O}(n)$ für Intervallgraphen gesehen - dies widerspricht aber keineswegs der NP-schere des Problems. Wir betrachten nun einen weiteren solchen "netten" Fall für Maximum Independent Set.

Bäume statt beliebige Graphen

Wir betrachten nun, was passiert, wenn der Graph G ein Baum ist, d.h. ein zusammenhängender kreisfreier Graph.
An unserem rekursiven Ansatz ändert sich nicht viel. Einziger Unterschied ist nun, dass die Graphen $G \setminus \{x\}$ bzw. $G \setminus N(x)$ *Wälder* sein müssen. In diesen Wäldern können wir nun für jeden enthaltenen Baum *unabhängig* ein Maximum Independent Set bestimmen und dann alle gefundenen Teillösungen einfach vereinigen.

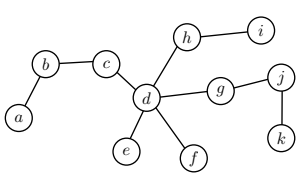
MISonTree(T)

Input: ungerichteter Baum $T = (V, E)$
Output: Maximum Independent Set von T

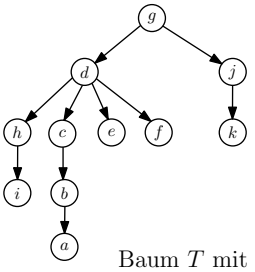
```
1: if V = ∅ then
2:     return ∅
3: end if
4: x ← beliebiger Knoten von G
5: mitx ← {x}
6: for jeden Baum T' im Wald T \ N(x) do
7:     mitx ← mitx ∪ MISonTree(T')
8: end for
9: ohnex ← ∅
10: for jeden Baum T' im Wald T \ {x} do
11:     ohnex ← ohnex ∪ MISonTree(T')
12: end for
13: if |mitx| ≥ |ohnex| then
14:     return mitx
15: else
16:     return ohnex
17: end if
```

Wir haben noch nicht viel im Vergleich zur allgemeinen Version gewonnen. Immerhin können wir nun aber versuchen unnötige rekursive Aufrufe zu vermeiden.

Memoization: Jeder Subaufruf berechnet ein Maximum Independent Set für einen Teilbaum von T. Um all diese Lösungen zu speichern, müssten wir somit für jeden Teilbaum von T eine Menge Speichern. Leider wird dies extrem teuer, da ein Baum mit n Knoten exponentiell viele Teilbäume haben kann[99]!
Es gibt allerdings einen Trick, der uns hier entscheidend hilft: Im Algorithmus dürfen wir auswählen, welcher Knoten x jeweils betrachtet wird! D.h. wir können dem Baum T etwas mehr Struktur geben, indem wir einfach einen beliebigen Knoten r zur Wurzel erklären und dann den an r gewurzelten Baum T betrachten[100].



Baum T



Baum T mit Wurzel g

Im Algorithmus wählen wir dann jeweils die Wurzel des aktuellen Teilbaums als unseren Knoten x. Somit folgt, dass jeder Knoten die Wurzel von genau einem Teilbaum von T ist und somit gibt es nur n Teilprobleme! Die Lösungen dieser Teilprobleme könnten wir uns nun in einem Array speichern, doch es geht noch viel einfacher: Wir speichern uns die Lösung eines Teilproblems einfach im entsprechenden Wurzelknoten des zugehörigen Teilbaums. Sei v ein Knoten, dann nehmen wir im Folgenden an, dass im Attribut $v.MIS$ das entsprechende Maximum Independent Set des Teilbaums mit Wurzel v gespeichert ist. Anfangs sind alle Attribute auf $\emptyset$ gesetzt. Wir erhalten folgenden Algorithmus:

MISonRootedTree(T, x)

Input: gewurzelter Baum $T = (V, E)$ mit Wurzel x
Output: Maximum Independet Set von T

```
1: ohnex ← ∅
2: for jedes Kind u von x do
3:     ohnex ← ohnex ∪ MISonRootedTree(T, u)
4: end for
5: mitx ← {x}
6: for jedes Enkelkind u von x do
7:     mitx ← mitx ∪ u.MIS
8: end for
9: if |mitx| ≥ |ohnex| then
10:    x.MIS ← mitx
11: else
12:    x.MIS ← ohnex
13: end if
14: return x.MIS
```

[99]Ein Beispiel für einen solchen Baum wäre ein Stern, d.h. ein Knoten $n-1$ Blättern. Entfernen von jeder beliebigen Teilmenge der Blätter für zu einem anderen Baum (da diese jeweils andere Knotenmengen haben) und es gibt 2^{n-1} viele Teilmengen der Blätter.
[100]Im gewurzelten Baum sind alle Kanten gerichtet, und wegwärts von der Wurzel orientiert. Wir benötigen die Kantenrichtungen aber nur, um für Endpunkte einer Kante festzulegen, wer Vater und wer Kind ist.

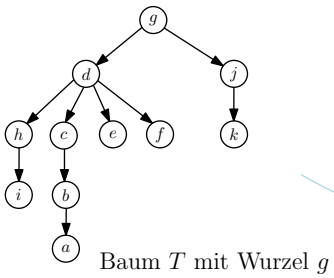
Da für jeden der n Teilbäume das zugehörige Maximum Independent Set nur genau ein mal berechnet wird und sonst alle weiteren Kosten konstant sind (insbesondere der Zugriff auf bereits berechnete Teillösungen) folgt, dass die Gesamtkosten in $\mathcal{O}(n)$ liegen! Wir haben somit gegenüber beliebigen Graphen eine exponentielle Verbesserung erzielt.
Wir können den Algorithmus noch vereinfachen, indem wir für jeden Knoten x einfach zwei Werte speichern, nämlich einmal das Maximum Independent Set, wenn x gewählt wird und einmal das Maximum Independent Set, wenn x nicht gewählt wird. Dies liefert uns folgenden Algorithmus:

MISonRootedTreeSimplified(T, x)

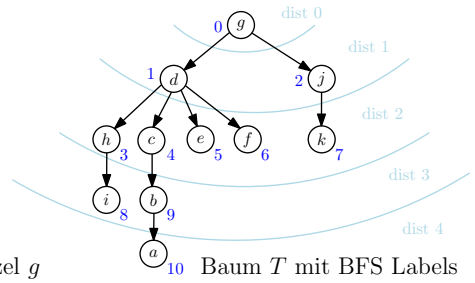
Input: gewurzelter Baum $T = (V, E)$ mit Wurzel x
Output: Maximum Independet Set von T

```
1: x.MISmit ← {x}; x.MISohne ← ∅
2: for jedes Kind u von x do
3:     x.MISohne ← x.MISohne ∪ MISonRootedTreeSimplified(T, u)
4:     x.MISmit ← x.MISmit ∪ u.MISohne
5: end for
6: if |x.MISmit| ≥ |x.MISohne| then
7:     return x.MISmit
8: else
9:     return x.MISohne
10: end if
```

Bottom-Up: Als letztes betrachten wir noch die iterative Version von MISonRootedTreeSimplified. Dazu müssen wir uns zunächst überlegen, was genau bei einem Baum als bottom-up Vorgehen zu verstehen ist.
Im obigen rekursiven Algorithmus tritt der Abbruchfall der Rekursion dann ein, wenn der betrachtete Knoten keine Kinder hat, d.h. wenn der betrachtete Knoten ein Blatt ist. Eine zulässige Abarbeitungsreihenfolge der Knoten (bzw. der Teilprobleme) muss somit sicherstellen, dass für einen betrachteten Knoten alle Kinder bereits abgearbeitet wurden. Dies ist leicht, wenn wir die Knoten im Baum einfach in *umgekehrter Breitensuchreihenfolge*, d.h. von den Blättern zur Wurzel hin, abarbeiten.
Zur Erinnerung, die Breitensuchen gibt, ausgehend vom Wurzelknoten, jedem Knoten im Graph ein Label, wobei die Knoten aber in Reihenfolge ihres Abstands zur Wurzel abgearbeitet werden. D.h. zuerst bekommt die Wurzel Label 0, dann erhalten alle k Nachbarn der Wurzel die Label $1, 2, \ldots, k$, danach erhalten die Nachbarn der Nachbarn je ein Label, usw. Sei $BFS(G, x)$ die Zuordnung der Breitensuch-Labels zu den Knoten von G.



Baum T mit Wurzel g



Baum T mit BFS Labels

Der bottom-up Algorithmus lautet dann:

MISonRootedTreeBottomUp(T, x)

Input: gewurzelter Baum $T = (V, E)$ mit Wurzel x
Output: Maximum Independet Set von T

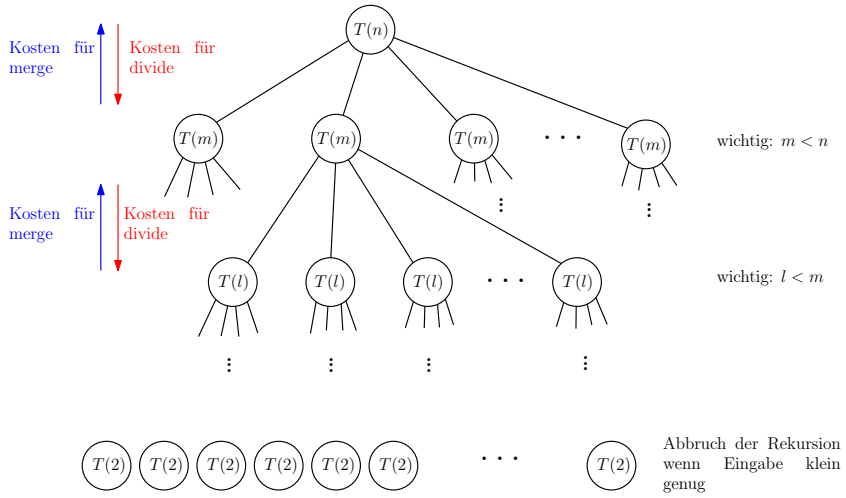
```
1: berechne BFS(T, x)
2: L ← Array der Knoten absteigend nach Label sortiert
3: for i = 0 to n − 1 do
4:     w ← L[i]
5:     w.MISmit ← {w}; w.MISohne ← ∅
6:     for jedes Kind u von w do
7:         w.MISohne ← w.MISohne ∪ u.MIS
8:         w.MISmit ← w.MISmit ∪ u.MISohne
9:     end for
10:    if |w.MISmit| ≥ |w.MISohne| then
11:        w.MIS ← w.MISmit
12:    else
13:        w.MIS ← w.MISohne
14:    end if
15: end for
16: return x.MIS
```

Hier speichern wir, hauptsächlich um etwas Schreibarbeit zu sparen, für jeden Knoten x drei Mengen, nämlich in $x.MIS$ ein Maximum Independent Set für den Teilbaum mit Wurzel x, in $x.MISmit$ ein Maximum Independent Set für den Teilbaum mit Wurzel x welches x enthält und in $x.MISohne$ ein Maximum Independent Set für den Teilbaum mit Wurzel x, welches x nicht enthält.

7 Divide & Conquer

7.1 Divide & Conquer allgemein

Divide & Conquer Algorithmen teilen große Probleminstanzen rekursiv in unabhängige kleinere Instanzen auf (divide) bis die Teilinstanzen klein genug sind, um sie direkt zu lösen (conquer). Danach werden die rekursiv erzeugten Lösungen der Teilinstanzen noch zur Lösung der Gesamtinstanz zusammengesetzt bzw. "verschmolzen" (merge). Ein Divide & Conquer-Algorithmus besteht somit meist aus den Schritten divide + conquer + merge. Wichtig ist, dass die Kosten für das Aufteilen und Zusammensetzen jeweils von der aktuellen Problemgröße abhängen und im Verlauf des Algorithmus sehr viele divide und merge Schritte vorkommen können. Um die Struktur hervorzuheben, kann man dies meist recht leicht mit Hilfe eines *Rekursionsbaums* visualisieren.
Angenommen wir haben einen D&C-Algorithmus für eine Eingabe der Größe n. Stellen wir uns nun vor, dass unser Algorithmus dieses Problem in k kleinere Instanzen der Größe $m < n$ aufteilt. Diese werden wiederum jeweils in k noch kleinere Instanzen der Größe $l < m$ aufgeteilt usw. bis wir bei Instanzen der Größe 2 ankommen. Diese werden dann direkt gelöst und die Lösungen von jeweils k vielen dieser Instanzen werden dann zur Lösung der entsprechenden Instanz, von der diese k Instanzen abstammen, zusammengesetzt. Dies setzt sich dann weiter fort, bis die Lösung für die Gesamtinstanz erzeugt wird.
Sei $T(n)$ die Kosten für das Lösen einer Instanz der Größe n, dann erhalten wir folgendes Gesamtbild:



Gesamtkosten für conquer: Anzahl Blätter $*$ Kosten für ein mal conquer

Angenommen die Kosten für das divide einer Instanz der Größe n sind $f(n)$ und die Kosten für das merge von k Teillösungen zu einer Lösung für eine Instanz der Größe n sind $g(n)$. Somit haben wir:

$$T(n) = kT(m) + f(n) + g(n).$$

Für $T(m)$ gilt dann rekursiv, dass

$$T(m) = kT(l) + f(m) + g(m).$$

Wenn wir nun $T(m)$ in $T(n)$ einsetzen, dann haben wir:

$$\begin{aligned} T(n) &= kT(m) + f(n) + g(n) \\ &= k(kT(l) + f(m) + g(m)) + f(n) + g(n) \\ &= k^2(T(l)) + kf(m) + kg(m) + f(n) + g(n) \end{aligned}$$

Wenn wir das fortführen, dann sieht man, dass sich pro Ebene des Rekursionsbaums die Kosten für das divide und für das merge aufsummieren und der vordere Term immer kleiner wird, bis er den Abbruchfall $T(2)$ erreicht. Insgesamt haben wir somit, dass die Gesamtkosten des Divide & Conquer Algorithmus sich aus der Summe der Kosten pro Ebene des Rekursionsbaumes zusammensetzen. Zu jeder Ebene, außer der letzten, gehören die Kosten für das divide und für das merge. Für die letzte Ebene, die sogenannte Blattebene, entstehen nur die Kosten für das conquer.

7.2 Bestimmung des Medians

Bei Quicksort wäre es von Vorteil, wenn als Pivot-Element jedes mal der Median des aktuellen Teilarrays gewählt werden würde. Genau mit diesem Problem werden wir uns nun beschäftigen. D.h. wir betrachten:

Gegeben: Ein Array A mit n Elementen.

Aufgabe: Finde ein Median-Element von A.

Zur Erinnerung, der *Median* $m \in A$ eines Arrays A der Länge n ist das Element von A, für das gilt:

- es gibt höchstens $\lfloor \frac{n+1}{2} \rfloor$ Elemente aus A, die höchstens so groß wie m sind, und
- es gibt höchstens $\lceil \frac{n+1}{2} \rceil$ Elemente aus A, die mindestens so groß wie m sind.

D.h. der Median ist immer das $\lceil \frac{n}{2} \rceil$-kleinste Element im Array[101] und existiert immer.

7.2.1 Erste Lösungsansätze

Wir wollen zunächst das Problem überhaupt lösen. Dafür kommen sofort die folgenden beiden Algorithmen in Frage.

Brute-Force: Wir könnten natürlich einfach jedes Element mit jedem anderen Vergleichen und dabei zählen, wieviele davon kleiner oder gleich bzw. größter oder gleich sind. Damit finden wir den Median trivialerweise mit $\Theta(n^2)$ vielen Vergleichen.

Sortieren: Noch schneller geht es, indem wir das Array A zunächst z.B. mit Mergesort sortieren und dann einfach das Element an der Position $\lceil \frac{n}{2} \rceil$ im sortierten Array auswählen. Dies ergibt Gesamtkosten in $\Theta(n \log n)$.
Sortieren scheint uns somit wirklich zu helfen, um den Median zu finden. Die große Frage lautet nun, ob wir überhaupt ohne Sortieren auskommen können, um den Median zu bestimmen. D.h. wir landen eigentlich bei folgender Frage:

> Ist die Bestimmung des Medians substantiell einfacher als Sortieren?

7.2.2 Bestimmung des Medians in Linearzeit

Die etwas überraschende Antwort auf unsere Frage lautet: Ja! Wir können den Median in Linearzeit bestimmen, d.h. dies ist ein fundamental einfacheres Problem als das Sortieren[102].
Die Lösung des Problems liefert ein etwas unintuitiver und überraschend einfacher Divide & Conquer Algorithmus, der meist *Median-der-Mediane Algorithmus* genannt wird. Der Algorithmus wurde 1973 von Blum, Floyd, Pratt, Rivest & Tarjan veröffentlicht[103]. Genauergesagt löst der Algorithmus ein noch allgemeineres Problem: Er findet das k-kleinste Element in einem beliebigen Array. Wenn wir also $k = \lceil \frac{n}{2} \rceil$ setzen, dann haben wir einen Median-Algorithmus.
Er basiert auf folgender Idee: Bestimme erstmal einen approximativen Median, d.h. ein Element, welches auf jeden Fall größer und kleiner als ein konstanter Bruchteil aller Elemente ist. Dann partitioniere ähnlich wie bei Quicksort und suche im richtigen Teilarray rekursiv nach dem k-kleinsten Element. Nun das Überraschende: Für die Bestimmung des approximativen Medians wird dabei der selbe Algorithmus einfach rekursiv verwendet!

[101] Für gerade n könnten wir natürlich auch das $\frac{n}{2} + 1$-kleinste Element wählen.

[102] Zur Erinnerung: Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ viele Vergleiche für das Sortieren von n Elementen.

[103] D.h. am Entwurf dieses Algorithmus waren einige der bekanntesten Algorithmiker überhaupt beteiligt. Wir werden Floyd später nochmal antreffen. Rivest ist übrigens das "R" der RSA-Verschlüsselung.

Hier der Algorithmus Select, der das k-kleinste Element in einem Array A der Länge n bestimmt:

1. Teile das Array A in $\lceil \frac{n}{5} \rceil$ viele Teilarrays der Länge 5 und ein Teilarray mit den übrigen Elementen.
2. Bestimme den Median der $\lceil \frac{n}{5} \rceil$ Teilarrays direkt (z.B. per Sortieren).
3. Benutze Select rekursiv, um den Median der $\lceil \frac{n}{5} \rceil$ Mediane aus Schritt 2 zu bestimmen. Es sei x dieser Median-der-Mediane.
4. Rufe eine modifizierte Version von Partition von Quicksort mit Pivot-Element x auf. Hierbei wird neben dem Partitionieren noch mitgezählt, wieviele Elemente die beiden Teilarrays enthalten. Sei $j-1$ die Anzahl der Element im Teilarray der kleineren Elemente.
5. Falls $k = j$, dann gib x aus. Sonst benutze Select rekursiv um das k-kleinste Element im linken Teilarray zu finden, falls $j > k$, oder um das $k-j$-kleinste Element im rechten Teilarray zu finden, falls $j < k$.

Schritt $1-3$ bestimmt rekursiv einen approximativen Median und die Schritte $4-5$ suchen dann rekursiv im richtigen Teilarray weiter.

Beispiel: Wir betrachten ein Beispiel mit $n = 27$ und suchen den Median, d.h. $k = 14$:

$$A = [20, 18, 25, 28, 29, 17, 15, 10, 1, 23, 24, 3, 14, 13, 22, 12, 26, 27, 5, 11, 6, 7, 9, 8, 16, 19, 21]$$

Zunächst Teilen wir in 5 Teilarrays:

$$A_1 = [20, 18, 25, 28, 29] \quad A_2 = [17, 15, 10, 1, 23] \quad A_3 = [24, 3, 14, 13, 22]$$

$$A_4 = [12, 26, 27, 5, 11] \quad A_5 = [6, 7, 9, 8, 16] \quad A_6 = [19, 21]$$

Jetzt bestimmen wir die Mediane der Teilarrays direkt, wir haben somit

$$m_{A_1} = 25, m_{A_2} = 15, m_{A_3} = 14, m_{A_4} = 12, m_{A_5} = 8, m_{A_6} = 19.$$

Nun suchen wir rekursiv den Median im Array

$$M_1 = [m_{A_1}, m_{A_2}, m_{A_3}, m_{A_4}, m_{A_5}, m_{A_6}] = [25, 15, 14, 12, 8, 19],$$

d.h. wir suchen das 3-kleinste Element in einem 6-elementigen Array. Wir teilen also in Array der Länge 5 auf:

$$B_1 = [25, 15, 14, 12, 8] \quad B_2 = [19]$$

und ermitteln die Mediane: $m_{B_1} = 14, m_{B_2} = 19$.
Wir teilen wieder in 5-elementige Teilarrays auf und erhalten

$$M_2 = [m_{B_1}, m_{B_2}] = [14, 19].$$

Nun haben wir nur noch ein Teilarray der Länge höchstens 5 und somit bestimmen wir den Median davon direkt: $m_{C_1} = 14$.

Jetzt partitionieren wir das Array M_1 mit unserem Median-der-Mediane von Array M_2: 14. Wir erhalten:

$$M_1^L = [12, 8] \quad M_1^R = [25, 15, 19].$$

M_1^L hat 2 Elemente und somit ist 14 der Median von Array M_1. Und somit ist 14 auch der Median-der-Mediane von Array M_1.
Nun partitionieren wir A mit Pivot-Element 14:

$$A^L = [10, 1, 3, 13, 12, 5, 11, 6, 7, 9, 8]$$

und

$$A^R = [21, 28, 22, 29, 26, 27, 17, 15, 20, 18, 23, 24, 16, 19, 25]$$

Es gilt $|A^L| = 11$, somit müssen wir rekursiv das $14 - 11 - 1 = 2$-kleinste Element in A^R bestimmen.
Wieder zerlegen wir A^R in Teilarrays der Länge 5:

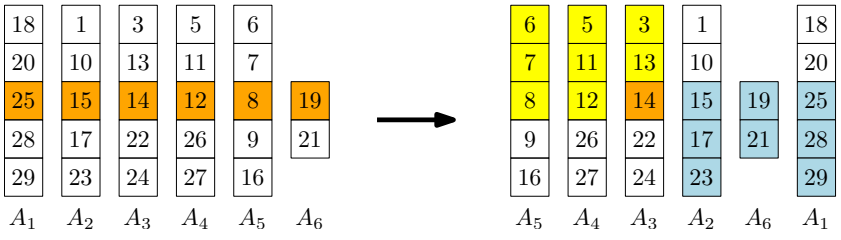
$$A_1^R = [21, 28, 22, 29, 26] \quad A_2^R = [27, 17, 15, 20, 18] \quad A_3^R = [23, 24, 16, 19, 25].$$

Das Array der Mediane davon lautet dann $M_1^R = [26, 18, 23]$ und der Median davon ist 23. Wir partitionieren dann A^R mit Pivot-Element 23 und erhalten:

$$A_R^L = [21, 22, 17, 15, 20, 18, 16, 19] \quad A_R^R = [27, 25, 24, 28, 29, 26].$$

Es gilt $|A_R^L| = 8$ und somit suchen wir rekursiv das 2-kleinste Element in A_R^L.
Wir teilen auf: $D_1 = [21, 22, 17, 15, 20]$ und $D_2 = [18, 16, 19]$. Finden die Mediane $m_{D_1} = 20$ und $m_{D_2} = 18$ und der Median davon ist 18. Nun partitionieren wir A_R^L mit Pivot-Element 18 und erhalten: $D^L = [17, 15, 16]$ und $D^R = [20, 19, 21, 22]$. Es gilt $|D^L| = 3$ und somit suchen wir rekursiv das 2-kleinste Element in D^L. Da D^L nur drei Elemente hat, bestimmen also den Median $m_{D^L} = 16$, partitionieren, und erhalten 16 als unseren gesuchten Median von A. Dies ist tatsächlich korrekt, denn 16 ist das 14-kleinste Element in A.

Warum funktioniert das? Die eigentlich spannende Frage lautet, warum der Median-der-Mediane eigentlich ein so interessantes Element ist. Um das zu verstehen, visualisieren wir die 6 Teilarrays $A_1, \ldots, A_6$ etwas anders:



Angenommen wir sortieren alle Teilarrays und danach sortieren wir die Teilarrays nach ihren Medianen. Dann erhalten wir das obige Bild. Der Median-der-Mediane 14 hat folgende Eigenschaft: Er ist auf jeden Fall größer oder gleich als alle Mediane links von ihm und größer als alle Elemente, die über ihm und den Medianen links davon stehen. All diese Element sind gelb markiert. Außerdem ist die 14 kleiner oder gleich als alle Mediane

rechts davon und auch kleiner oder gleich als alle Elemente die unter diesen Medianen und der 14 stehen. Diese Elemente sind hellblau markiert.
Somit ist der Median-der-Mediane in unserem Beispiel ein guter approximativer Median von A, denn es gibt mindestens 9 Elemente die kleiner oder gleich 14 sind und es gibt mindestens 8 Elemente die größer oder gleich 14 sind! Im sortierten Array A würde die 14 somit recht zentral zu finden sein.[104]

7.2.3 Analyse des Median-der-Mediane Algorithmus

Zunächst zeigen wir, dass der Median-der-Mediane x als Pivot-Element garantiert einen Bruchteil der Eingabeinstanz A abtrennt. Im Folgenden sei n die Anzahl der Elemente in A.

Lemma 18. *Sei x der Median-der-Mediane des n-elementigen Array A. Es gibt mindestens $\frac{3n}{10} - 6$ viele Elemente aus A, die alle höchstens so groß wie x sind und es gibt mindestens $\frac{3n}{10} - 6$ viele Elemente aus A, die alle mindestens so groß wie x sind.*

Beweis. Wir betrachten zunächst, wieviele Elemente von A kleiner oder gleich x sein müssen.
Es gibt $\lceil \frac{n}{5} \rceil$ viele Teilarrays der Länge 5. Davon ist eines das Teilarray in dem x enthalten ist und ein Teilarray hat evtl. weniger als 5 Elemente. Da x der Median der Mediane ist, gibt es $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil$ viele Teilarrays, die Mediane besitzen, die kleiner oder gleich x sind. Wir interessieren uns nun für die Teilarrays davon, die jeweils 3 Elemente, die kleiner oder gleich x sind, enthalten.
Wir ziehen das Teilarray von x ab und wir ziehen ein weiteres ab, da dieses evtl. weniger als 5 Elemente hat und somit nicht 3 solche Elemente beisteuert.
Somit gibt es $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2$ viele Teilarrays, die jeweils 3 Elementen enthalten, die kleiner oder gleich x sind. Somit gibt es mindestens

$$3\left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

viele Elemente, die kleiner oder gleich x sind.
Der Beweis für die Anzahl der Elemente, die größer oder gleich x sind, ist analog. □

Nun zu den Gesamtkosten des Algorithmus. Im worst-case verkleinert das Partitioniern mit dem Median-der-Mediane als Pivot-Element die Eingabeinstanz von Länge n auf Länge $\frac{7n}{10} + 6$. Um x überhaupt zu ermitteln, haben wir den Algorithmus auf einer Instanz der Länge $\lceil \frac{n}{5} \rceil$ ausgeführt und dann mit Kosten $\mathcal{O}(n)$ partitioniert.
Wir erhalten somit die folgende Rekurrenz:

$$T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + \mathcal{O}(n).$$

Bisher haben wir noch nicht über die Abbruchbedingung gesprochen (oben im Beispiel sind wir bis $n \leq 5$ in der Rekursion abgestiegen). Prinzipiell könnten wir das Problem

[104] Auch für Quicksort gilt, dass, wenn wir jedes mal ein Pivot-Element wählen, welches garantiert einen konstanten Bruchteil der Instanz abspaltet, wir einen Rekursionsbaum mit logarithmischer Tiefe und somit Kosten in $\mathcal{O}(n \log n)$ erhalten. Exakt das passiert hier auch, nur das wir eben nicht sortieren, sondern nach dem k-kleinsten Element suchen.

immer direkt lösen, wenn wir nur noch ein konstant großes Teilarray haben. Wir wählen hier als Abbruchfall $n \leq 140$. Den Grund dafür sehen wir gleich. D.h. unsere Rekurrenzgleichung lautet:

$$T(n) \leq \begin{cases} \mathcal{O}(1), & \text{falls } n \leq 140 \\ T\left(\lceil \frac{n}{5} \rceil\right) + T\left(\frac{7n}{10} + 6\right) + \mathcal{O}(n), & \text{falls } n > 140. \end{cases}$$

Wir zeigen nun eine lineare obere Schranke für diese Rekurrenz. Wir benutzen dafür eine neue Technik: Wir raten eine Lösung und substituieren.
Wir raten, dass $T(n) \leq cn$ gilt. Wenn wir dies und $c'n$ für den $\mathcal{O}(n)$ Term nun einsetzen, dann erhalten wir:

$$\begin{aligned} T(n) &\leq c\left\lceil \frac{n}{5} \right\rceil + c\left(\frac{7n}{10} + 6\right) + c'n \leq c \cdot \frac{n}{5} + c + c \cdot \frac{7n}{10} + 6c + c'n \\ &= c \cdot \frac{9n}{10} + 7c + c'n = cn + \left(-\frac{cn}{10} + 7c + c'n\right) \end{aligned}$$

Somit müssen wir nur noch zeigen, dass $-\frac{cn}{10} + 7c + c'n \leq 0$ gilt. Wir haben:

$$\begin{aligned} &-\frac{cn}{10} + 7c + c'n \leq 0 \iff 70c + 10c'n \leq cn \iff \frac{70c}{n} + \frac{10c'n}{n} \leq c \\ &\iff 10c' \leq c - \frac{70c}{n} \iff 10c' \leq \frac{nc - 70c}{n} \iff 10c'\frac{n}{n-70} \leq c. \end{aligned}$$

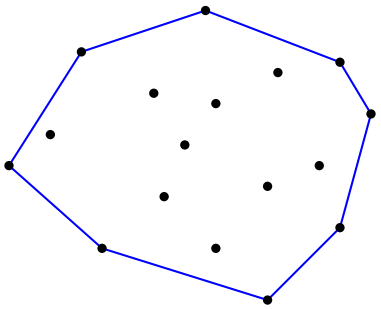
Somit haben wir für $n > 140$, dass $\frac{n}{n-70} \leq 2$. Wenn wir nun also $c \geq 20c'$ wählen, dann gilt die Ungleichung. Damit haben wir bewiesen, dass unser Algorithmus tatsächlich nur Kosten in $\mathcal{O}(n)$ hat.

7.3 Berechnung der konvexen Hülle

Wir betrachten das folgende Problem

> **Geg.:** Die Koordinaten von n Punkten der reellen Ebene.
>
> **Aufgabe:** Bestimme die konvexe Hülle der n Punkte in zyklischer Reihenfolge.

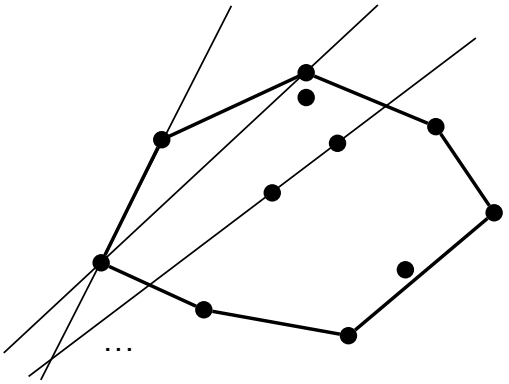
Wir werden im Folgenden annehmen, dass alle Punkte paarweise unterschiedliche x und y Koordinaten haben. Dies ist keine wirkliche Einschränkung des Problems - es erlaubt uns nur, einige Sonderfälle zu ignorieren.
Eine Menge von Punkten aus der reellen Ebene ist *konvex*, wenn die Verbindungsstrecke zwischen je zwei Punkten der Menge auch zur Menge gehört. Die *konvexe Hülle* ist die kleinste mögliche Menge, die die gegebene Menge enthält und konvex ist.
Was ist die konvexe Hülle für eine gegebene Punktmenge? Intuitiv ist dies einfach: Man stelle sich ein Brett mit Nägeln darin vor und eine Schnur, die von außen her festgezogen wird.



Als Eingabe haben wir also n Punkte $p_i = (x_i, y_i)$ und wir wollen als Ausgabe die Punkte der Hülle in zyklischer Reihenfolge, damit wir später schnell auf die Hülle zugreifen können.

7.3.1 Konvexe Hülle per Brute Force

Wir haben eine Menge von Punkten $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ gegeben und wir schauen für jedes Linienstück, ob es zur Hülle gehört. Wir prüfen dies also für jedes Paar von Punkten $(x_i, y_i), (x_j, y_j)$.

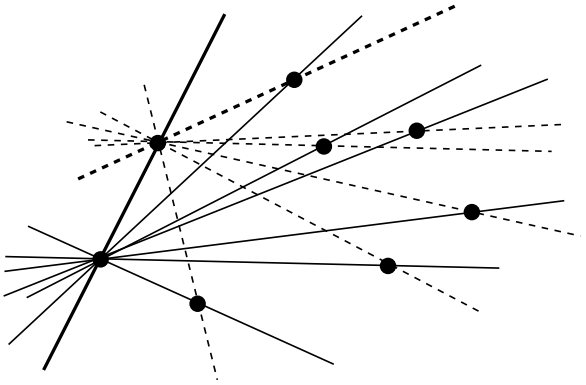


Für jeden einzelnen Test benötigen wir dabei $\mathcal{O}(n)$ Schritte, denn wir müssen die $n-2$ anderen Punkte mit unserem Linienstück vergleichen und testen, ob alle auf einer Seite davon liegen. Dies ist mit der Betrachtung der einzelnen Geradengleichungen machbar.
Total ergibt sich eine Komplexität von $\Theta(n^3)$, da wir den Test für n^2 viele Paare durchführen müssen. Weiterhin kommt noch der Zusatzaufwand für das Ausgeben der Hülle in der richtigen Reihenfolge hinzu.
Dieser sehr naive Ansatz dauert also viel zu lange, denn idealerweise streben wir an, dass wir eine genau passende obere Schranke für unsere untere Schranke finden.

7.3.2 Der Hammer-Nagel-Schnur-Ansatz (Jarvis' March)

Wir stellen uns vor, dass die Punkte auf einem Brett liegen und wir in jeden Punkt einen Nagel schlagen. Nehmen wir dafür linearen Zeitaufwand an. Im „Schnur-Teil“ legen wir diese großräumig um alle Nägel herum und ziehen sie fest. Für diesen Teil benötigen wir nur konstanten Aufwand, da das Festziehen ja nicht von der Anzahl der Nägel abhängt. Als letztes laufen wir nun der Schnur nach und wir zählen dafür einen Schritt pro Segment. Aus diesem Vorgehen können wir einen Algorithmus ableiten und dieser ist unter dem Namen *Jarvis' March* oder auch *gift wrapping* bekannt.

Wir nehmen den linkesten Punkt (bzw. den linkesten untersten, falls es mehrere gibt) und berechnen alle Geraden von diesem Punkt zu den restlichen Punkten und nehmen die Gerade mit der größten Steigung. Für je zwei Punkte können wir die entsprechende Gerade in konstanter Zeit berechnen[105].

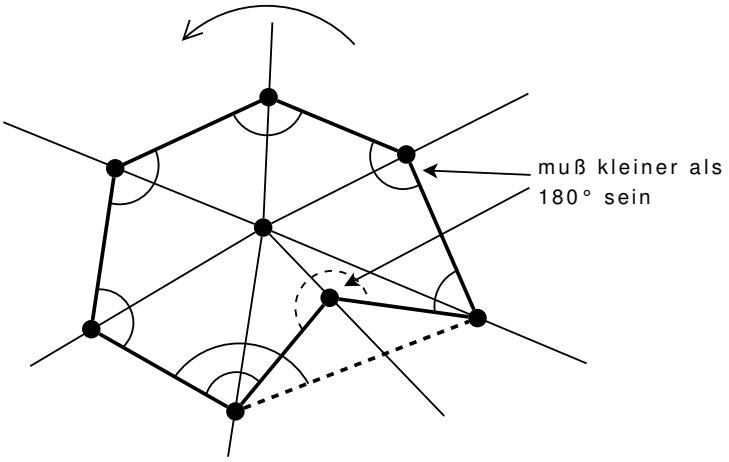


Dann nehmen wir den nächsten Punkt und betrachten die größte Steigung relativ zur aktuellen Position. Wir argumentieren hier etwas ungenau, denn man muss dabei einige degenerierte Fälle beachten. Falls zum Beispiel auf einer Gerade noch ein weiterer Punkt liegt, dann wollen wir diesen nicht ausgeben, sondern wählen den Punkt, der am weitesten entfernt liegt.
Wir brauchen mit diesem Ansatz $\mathcal{O}(n)$ für jedes Geradenstück und benötigen höchstens n Stücke. Total ergibt das $\mathcal{O}(n^2)$-Schritte für den Algorithmus von Jarvis.
Das ist keine schlechte Leistung, zum Beispiel wenn wenige Punkte auf der Hülle liegen, dann ist dieser Algorithmus sehr schnell. Für h Punkte auf der Hülle benötigen wir nur $\mathcal{O}(n \cdot h)$ Schritte. Eine solche Laufzeit nennt man übriges *Output sensitiv*, d.h. der Algorithmus ist umso schneller, je kleiner die gesuchte konvexe Hülle ist.
Diese Idee kann man aber noch besser ausnutzen und wir wollen dabei aus diesem „radialen Anschauen“ noch mehr herausholen.

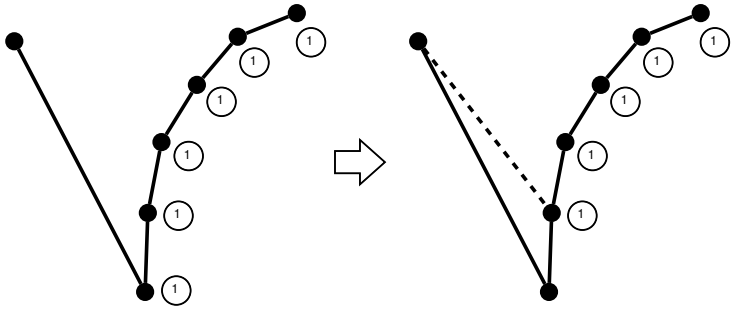
7.3.3 Radiale Sortierung (Graham's Scan)

Nehmen wir einen Punkt und sortieren die anderen anhand des Winkels der entsprechenden Geraden.
Jeder Winkel von einem Linienstück der Hülle zum nächsten Linienstück muss ≤ 180 Grad sein, dann ist dies *lokal konvex*. Falls ein Winkel diese Bedingung verletzt, dann ist der entsprechende Knoten nicht in der Hülle und wir müssen zurücklaufen und ein neues Linienstück testen.
Die Sortierung kostet uns $\mathcal{O}(n \log n)$ und das Herumlaufen $\mathcal{O}(n)$. Im Extremfall können wir beim Testen der Winkel bis an den Anfang zurückfallen, doch dies kann nicht oft passieren und wir haben hier wieder einen typischen Fall für eine amortisierte Analyse.
Ein einzelner Schritt kann sehr teuer sein, aber wenn wir einmal weit zurücklaufen mussten, dann kann das so schnell nicht wieder passieren. Wir können pro Punkt nur einmal zurückfallen und wir benötigen somit amortisiert nur $\mathcal{O}(n)$ Schritte.

[105] Für Punkte $(x_1, y_1), (x_2, y_2)$ mit $x_1 \neq x_2$ gilt: $y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$. Falls $x_1 = x_2$, dann lautet die Gerade $y = 0$.



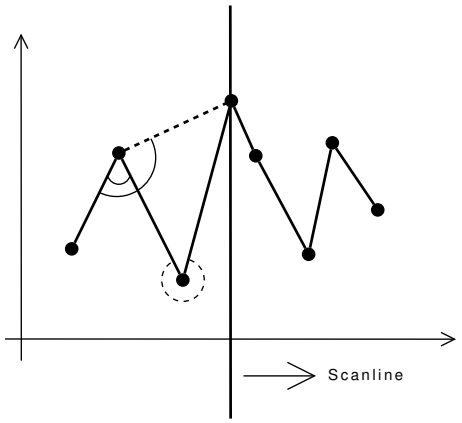
Dies sieht man auch mit der Guthabenmethode gut: Wir legen eine Geldeinheit auf jeden Punkt, von dem wir glauben, dass er auf der Hülle liegt und zahlen dann mit diesem "Geld" das Zurücklaufen, falls es nötig wird.



Wir müssen somit maximal n Einheiten auf das Konto einzahlen und für das Herumlaufen brauchen wir auch nur linear viele Schritte. Wir haben also total $O(n)$ amortisierte Kosten.
Insgesamt benötigen wir damit $\mathcal{O}(n \log n)$ für das Sortieren und $\mathcal{O}(n)$ für das finden der Hülle und somit haben wir eine Schranke von $O(n \log n)$. Diesen Ansatz nennt man *Graham's Scan*.

7.3.4 Scanline-Algorithmus

Wir sortieren zuerst nach den x-Werten und gehen dann mit einer „scanline" über die Punktmenge. Wir betrachten dabei die obere konvexe Hülle (vom linkesten bis zum rechtesten Punkt) und die untere konvexe Hülle getrennt und gehen genau wie im vorigen Ansatz vor.



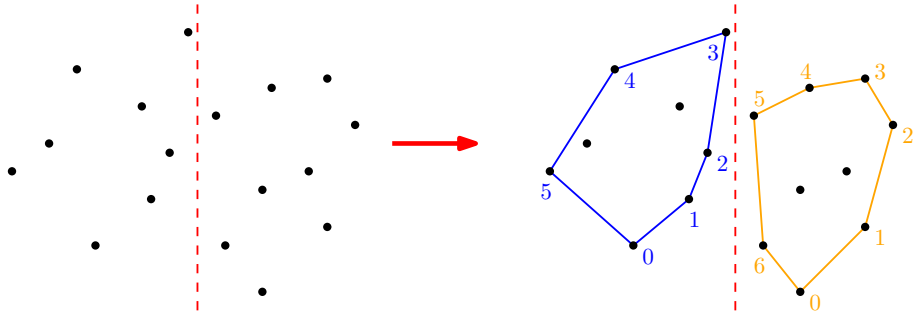
Wir brauchen auf diese Weise für die obere konvexe Hülle $\mathcal{O}(n \log n)$ Schritte und die untere konvexe Hülle berechnet man analog. Das elegante an diesem Ansatz ist, dass wir nur nach den x-Werten sortieren müssen und dies ist weitaus angenehmer als das Arbeiten mit Geradengleichungen. Außerdem können wir diesen Ansatz auch als eine Art Greedy Algorithmus verstehen. Nach der Sortierung versuchen wir immer lokal den bestmöglichen nächsten Punkt auf der Hülle zu wählen.

7.3.5 Konvexe Hülle per Divide & Conquer

Als nächstes betrachten wir einen recht einfachen Divide & Conquer Ansatz.

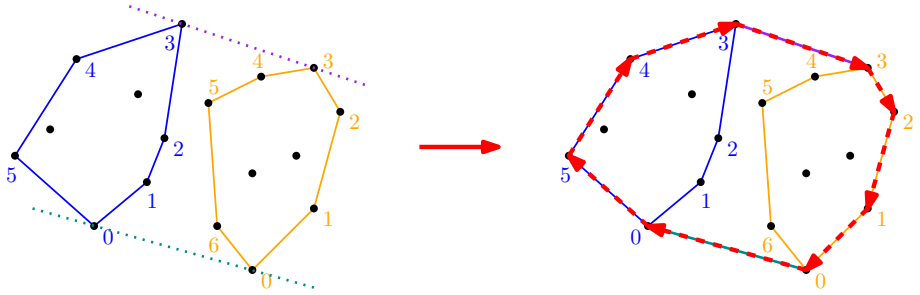
Idee: Wir sortieren die Punkte aufsteigend nach ihren x-Koordinaten. Dann partitionieren wir die Punktmenge in zwei (etwa) gleich große Hälften und suchen rekursiv nach der konvexen Hülle in beiden Hälften. Haben wir beide rekursiven Hüllen gefunden, dann konstruieren wir aus beiden die konvexen Hülle der Gesamtinstanz.

Merge: Die spannende Frage hier ist, wie wir aus den beiden rekursiv berechneten konvexen Hüllen der Teilinstanzen die konvexe Hülle der Gesamtinstanz berechnen können. Wir haben folgende Ausgangssituation: Wir haben die Reihenfolge der Punkte der konvexen Hülle aus dem linken Teilproblem und die Reihenfolge der Punkte aus dem rechten Teilproblem.



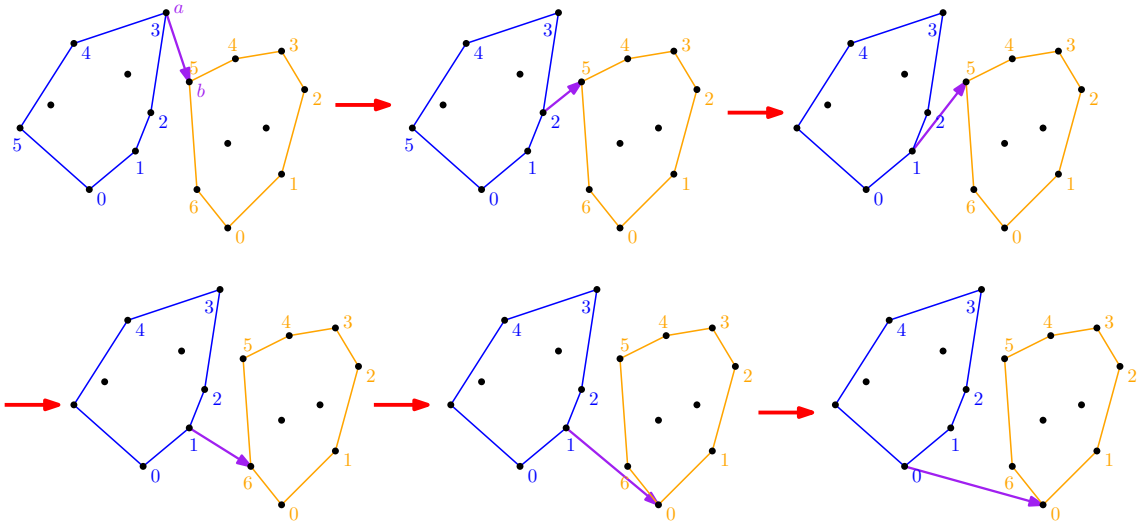
Unser Problem wäre leicht zu lösen, wenn wir die *obere und untere Tangente* der konvexen Polygone hätten. D.h. die Geraden, die beide Polygone jeweils in genau einem Punkt berühren und beide Polygone jeweils auf der selben Seite haben. Dann könnten wir ausgehend von einen Berührungspunkt einer Tangente eine Hülle traversieren, bis

wir wieder auf den Berührungspunkt der anderen Tangente treffen. Dann wechseln wir entlang der Tangente in die andere Hülle und traversieren weiter bis wir wieder einen Berührungspunkt finden. Schließlich folgen wir wieder der Tangente und sind fertig:



Wenn wir die Endpunkte der Tangenten gegeben haben, dann finden wir damit die konvexe Hülle der Gesamtinstanz in $\mathcal{O}(n)$. Es bleibt also nur noch zu klären, wir wir die Tangenten in $\mathcal{O}(n)$ bestimmen können.

Finden der unteren Tangente: Wir bestimmen die untere Tangente wie folgt. Die obere Tangente bestimmen wir dann analog.
Sei a der rechteste Punkt auf der konvexen Hülle des linken Teilproblems und b der linkeste Punkte auf der konvexen Hülle des rechten Teilproblems. Außerdem nehmen wir an, dass beide Hüllen zyklisch durchnummeriert sind.
Wir beginnen mit der Geraden, die a und b verbindet. Für eine gegebene Gerade mit Endpunkt x können wir leicht testen, ob die Gerade unterhalb des kompletten Polygons liegt. Dazu prüfen wir, ob die beiden Nachbarn von x auf der konvexen Hülle beide auf der richtigen Seite der Gerade liegen. Falls ja, dann ist die Gerade durch x eine Tangente des Polygons, sonst nicht. Wir können nun leicht mit Hilfe eines Greedy-Algorithmus eine geeignete Tangente finden, die beide Polygone an genau einem Punkt berührt. Dazu wählen wir einen der Endpunkte und testen die Tangenteneigenschaft. Ist diese verletzt, dann wählen wir den Nachbarn im Uhrzeigersinn, falls wir den Endpunkt auf der linken Hülle betrachten, und sonst wählen wir den Nachbarn gegen den Uhrzeigersinn. Dies iterieren wir so lange, bis auf beiden Seiten die Tangenteneigenschaft gilt.



Jeder Test der Tangenteneigenschaft kann mit konstanten Kosten realisiert werden, somit haben wir insgesamt Kosten in $\mathcal{O}(n)$.

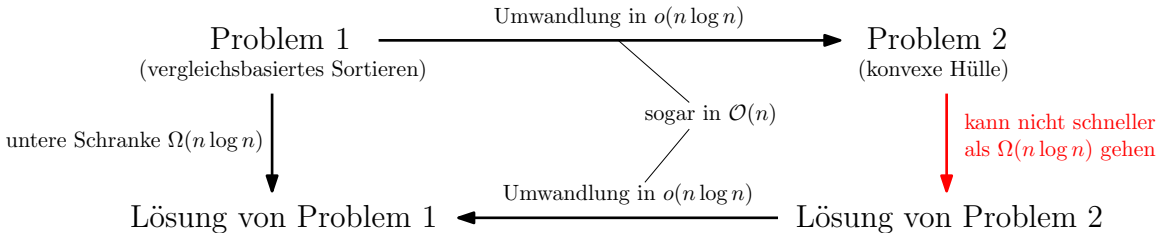
Analyse des Divide & Conquer Algorithmus: Der oben Beschriebene Algorithmus liefert uns folgende Rekurrenz:

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n).$$

Als Abbruchfall könnten wir beispielsweise $n \leq 3$ wählen, denn für höchstens 3 Punkte können wir leicht in konstanter Zeit die konvexe Hülle finden. Wir kennen die Rekurrenz bereits von MergeSort und erhalten somit Gesamtkosten in $\mathcal{O}(n \log n)$.

7.3.6 Untere Schranke für die Berechnung der konvexen Hülle

Als nächstes untersuchen wir, ob wir die gezeigte obere Schranke von $\mathcal{O}(n \log n)$ noch verbessern können. Wir suchen also eine untere Schranke für das Problem der Berechnung der konvexen Hülle einer Punktmenge.
Wir haben recht mühsam die Schranke von $\Omega(n \log n)$ für vergleichsbasierte Sortierverfahren hergestellt, aber wir werden nun sehen, dass wir nicht für jedes Problem einen so aufwändigen Weg gehen müssen.
Es ist möglich untere Schranken von einem Problem auf ein anderes zu übertragen und in diesem Abschnitt wollen wir dies für unsere Schranke für das Sortieren tun[106].
Wir wollen zeigen, dass man die konvexe Hülle im worst-case nicht schneller als in $\Omega(n \log n)$ Schritten bestimmen kann. Wir benutzen eine Konstruktion, die man *Reduktion* nennt und die in der theoretischen Informatik von fundamentaler Bedeutung ist. Wir werden die untere Schranke des Sortierens auf das vorliegende Problem reduzieren.
Die Denkweise ist wie folgt: Angenommen wir könnten mit Kosten in $o(n \log n)$ die konvexe Hülle bestimmen, dann könnten wir auch mit Kosten $o(n \log n)$ vergleichsbasiert Sortieren. Da wir aber bewiesen haben, dass das nicht möglich ist, folgt, dass es keinen $o(n \log n)$ Algorithmus zur Berechnung der konvexen Hülle geben kann.



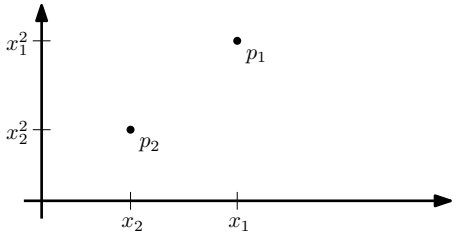
Dazu stellen wir für jeden konkreten Input des einen Problems einen Input für das Problem, für das wir die untere Schranke zeigen wollen, her. Zu beachten ist, dass die Transformationen nur $o(n \log n)$ Schritte benötigen sollten[107].
Wir nehmen an, dass wir die Lösung für die konvexe Hülle haben und stellen daraus die Lösung für das Sortieren her. Wir nehmen somit einen „Umweg“ über die konvexe Hülle, um zu einer Lösung für das Sortieren zu kommen, doch dieser Umweg kann auch nicht schneller als $\Omega(n \log n)$ sein, da wir sonst schneller sortieren könnten. Dies ist die „Mechanik“, welche die untere Schranke von einem Problem auf ein anderes überträgt.

[106] Wir haben die Idee bereits angetroffen, als wir gezeigt haben, dass es keine Priority Queue geben kann, die Insert in $\mathcal{O}(1)$ und ExtractMin in $o(\log n)$ realisiert.
[107] Wir betrachten hier also keine *Polynomialzeitreduktion*, sondern eine Einschränkung davon.

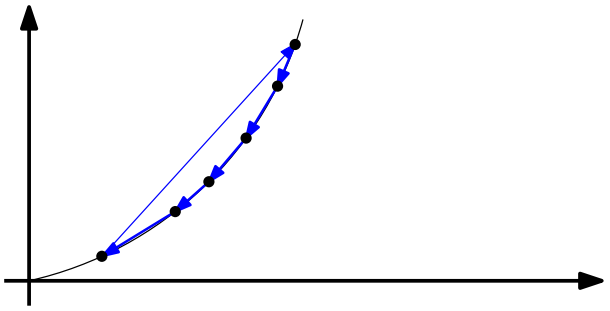
Doch wie realisieren wir die Transformationen? Wir betrachten zunächst die Transformation vom Sortieren zur konvexen Hülle:

$$\underbrace{(x_1, x_2, \ldots, x_n)}_{\text{Input für Sortieren}} \longmapsto \underbrace{(p_1, p_2, \ldots, p_n)}_{\text{Input für konvexe Hülle}}$$

Und wir transformieren den Input einfach mit $p_i = (x_i, x_i^2)$.
Wir zeichnen die entstehenden Punkte und erhalten einen Punkthaufen, für den wir die KH bestimmen können.



Alle Punkte liegen auf der Parabel $y = x^2$. Da die Hülle in zyklischer Reihenfolge geliefert wird, erhalten wir die sortierte Folge in zyklischem Zustand, sie ist also irgendwo gebrochen.



Wir können aber mit einem Durchlauf die richtige Reihenfolge herstellen und haben somit unsere gesuchte Lösung für das Sortierproblem. Wir können somit sogar in $O(n)$ die beiden Transformationen realisieren.
Somit haben wir die untere Schranke des Sortierens auf das Problem der Berechnung der konvexen Hülle übertragen. Dies zeigt uns, dass untere Schranken also auch für viele andere Probleme hilfreich sein können.

7.4 Closest Pair of Points

Wir betrachten ein weiteres Problem aus der Geometrie, das sogenannte *Closest Pair of Points* Problem. Es ist wie folgt definiert:

> **Geg:** Die Koordinaten von n Punkten P in der euklidischen Ebene.
>
> **Aufgabe:** Finde die beiden Punkte, die den geringsten Abstand zueinander haben.

Wieder bezeichnen wir den i-ten Punkt mit $p_i = (x_i, y_i)$. Der Abstand zwischen zwei Punkten $p_i \in P$ und $p_j \in P$ entspricht der Euklidischen Distanz zwischen beiden, d.h.

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

Wir nehmen wieder an, dass alle Punkte paarweise unterschiedliche x- bzw. y-Koordinaten haben. Auch hier ist dies keine Einschränkung und kann z.B. durch eine geeignete Rotation sichergestellt werden.

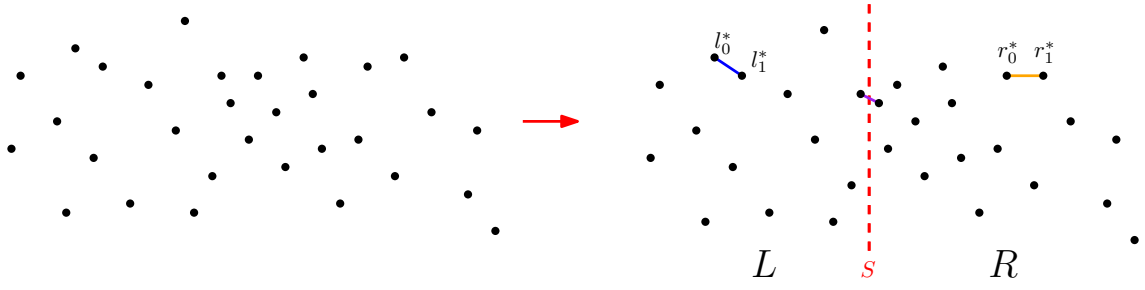
Naive Lösung: Wir könnten einfach für jedes Punktepaar den Abstand berechnen und uns das Minimum merken. Die Kosten dafür sind in $\Theta(n^2)$.

1-Dimensionale Version: Als Aufwärmübung betrachten wir das Closest Pair of Points Problem auf einer Linie.
Dies können wir leicht in $\mathcal{O}(n \log n)$ lösen, indem wir einfach die Punkte sortieren und dann durch die sortierte Liste gehen und jeweils den Abstand zum Vorgänger und Nachfolger überprüfen und das Minimum speichern.

Verallgemeinerung der 1-Dimensionalen Version: Wir wollen der Problem eigentlich in der Ebene, d.h. in $2D$, lösen. Eine einfache Verallgemeinerung des obigen Ansatzes für die $1D$-Version wäre, einfach nach x oder y Koordinaten zu sortieren und dann die Distanzen zu Vorgänger und Nachfolger zu bestimmen. Es ist allerdings leicht zu sehen, dass dieser Ansatz nicht korrekt ist.

Divide & Conquer Ansatz:

Wir versuchen die Punktmenge geeignet zu Partitionieren, dann in den Partitionen das Problem rekursiv zu lösen und schließlich aus den Teillösungen die Gesamtlösung zu konstruieren.
Das Partitionieren ist leicht. Dazu sortieren wir die Punkte nach x-Koordinate und halbieren die Instanz in der Mitte mittels einer vertikalen Schnittlinie S.



Das Problem hierbei ist: Angenommen wir kennen die Punktepaare, welche in der linken bzw. rechten Partition am dichtesten beisammen liegen. Wie finden wir damit das Punktepaar, welches in der Gesamtinstanz am dichtesten beisammen liegt? Es könnte eines der beiden Paare sein, doch es könnte auch sein, dass das Punktepaar auf beide Partitionen verteilt ist!

Repräsentation der Punktmengen und Ansetzen der Rekursion: Bevor wir das obige Problem angehen, überlegen wir uns noch ein paar Details zur Implementierung. Für eine Punktmenge $P' \subseteq P$ für einen rekursiven Aufruf verwalten wir jeweils zwei Listen P'_x und P'_y, wobei in ersterer die Punkte in P' aufsteigend nach x-Koordinate und in letzterer die Punkte in P' aufsteigend nach y-Koordinate sortiert sind. Wir sorgen dafür, dass dies für jeden rekursiven Aufruf gilt. Dies geht so:

Zu Beginn, d.h. vor dem ersten rekursiven Aufruf, sortieren wir alle Punkte jeweils nach x- und y-Koordinate und erhalten damit die Listen P_x und P_y. Zudem speichern wir uns in jedem Eintrag noch, welche Position der entsprechende Punkt in beiden Listen hat.
Für das Divide gehen wir nun durch P_x und wählen L als die Liste der ersten $\lfloor \frac{n}{2} \rfloor$ Einträge in P_x und R als die restliche Liste.
Indem wir nun ein mal durch P_x und P_y scannen, können wir die Listen L_x, L_y, R_x und R_y mit den Positionsangaben in der jeweils zugehörigen anderen Liste in $\mathcal{O}(n)$ erzeugen[108].
Nun bestimmen wir rekursiv das Punktepaar (l_0^*, l_1^*), welches in L am dichtesten beisammen liegt und das Punktepaar (r_0^*, r_1^*), welches in R am dichtesten beisammen liegt.

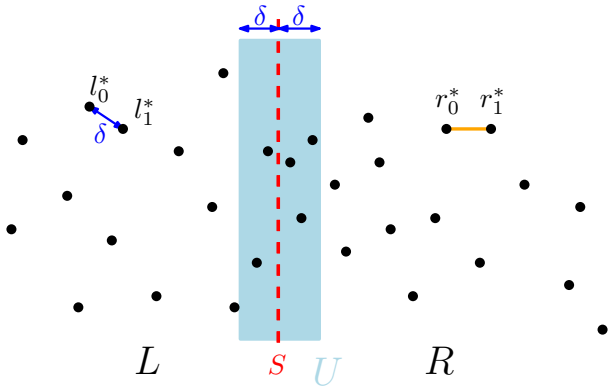
Zusammensetzen der Teillösungen Nun kommen wir zum interessantesten Punkt des Algorithmus, nämlich zum Merge. Wir gehen also davon aus, dass wir die Punktepaare (l_0^*, l_1^*) und (r_0^*, r_1^*) aus den rekursiven Aufrufen kennen und wollen nun das Punktepaar finden, welches in der Gesamtinstanz am dichtesten beisammen liegt. Dazu nehmen wir zunächst den kleinsten der Abstände der beiden rekursiv ermittelten Punktepaare als Referenzwert δ, d.h.

$$\delta = \min\left\{d(l_0^*, l_1^*), d(r_0^*, r_1^*)\right\}.$$

Falls δ der korrekte kürzeste Abstand zwischen zwei Punkten in der Gesamtinstanz ist, dann sind wir fertig und wir haben die beiden entsprechenden Punkte bereits gefunden. Falls es in der Gesamtinstanz zwei Punkte q_0 und q_1 gibt, deren Abstand kleiner als δ ist, dann muss einer der beiden Punkte in L liegen und der andere in R. Nun kommt die entscheidende Einsicht, welche auf dieser Trennung der Punkte basiert:

Beobachtung 26. *Falls es $q_0 \in L$ und $q_1 \in R$ mit $d(q_0, q_1) < \delta$ gibt, dann müssen q_0 und q_1 beide einen Abstand von höchstens δ zur Schnittlinie S haben.*

Um also herauszufinden, ob es ein solches paar $q_0 \in L$ und $q_1 \in R$ gibt, müssen wir nur in dem schmalen Band mit Abstand δ von S suchen!



Sei U die Menge der Punkte, die in Abstand höchstens δ zu S liegt und sei U_y die nach y-Koordinate sortierte Liste der Punkte in U. U_y kann in $\mathcal{O}(n)$ mit einem Scan der Liste P_y erzeugt werden. Wir können somit die obige Beobachtung noch etwas präzisieren:

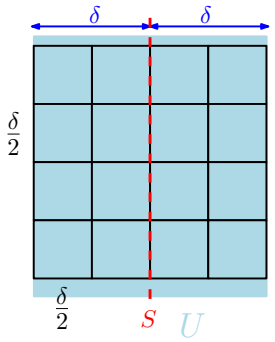
Beobachtung 27. *Es gibt genau dann $q_0 \in L$ und $q_1 \in R$ mit $d(q_0, q_1) < \delta$, wenn es zwei Knoten $u_0, u_1 \in U$ gibt, für die $d(u_0, u_1) < \delta$ gilt.*

[108] Die Details hierfür auszuarbeiten ist eine schöne Übungsaufgabe.

Betrachten wir nun die Menge U. Es ist durchaus möglich, dass $U = P$ gilt, d.h. alle Punkte der Gesamtinstanz liegen sehr nah bei S. So gesehen, ist es vollkommen unklar, ob uns die Menge U überhaupt nützt.
Sie nützt uns tatsächlich, denn es gilt die folgende verblüffende Aussage:

Theorem 28. *Falls für $u_0, u_1 \in U$ gilt, dass $d(u_0, u_1) < \delta$, dann sind u_0 und u_1 in der Liste U_y nur höchstens 15 Plätze auseinander.*

Beweis. Wir betrachten den Streifen, der alle Punkte der Ebene enthält, die Abstand höchstens δ zur Linie S haben, und unterteilen diesen Streifen in Quadrate mit Seitenlänge $\frac{\delta}{2}$, so dass eine Zeile der Unterteilung aus vier Quadraten besteht und die Quadrate in gleicher Höhe aneinandergereiht sind.



Angenommen es gibt zwei Punkte aus U liegen im selben Quadrat unserer Unterteilung, dann müssen beide Punkte in L oder beide Punkte in R liegen. Außerdem haben die Punkte dann einen Abstand von höchstens $\delta\frac{\sqrt{2}}{2} < \delta$ und dies widerspricht unserer Wahl von δ. Somit folgt, dass jedes Quadrat nur höchstens einen Punkt aus U enthalten kann!
Angenommen es gibt zwei Punkte $u_0, u_1 \in U$ mit $d(u_0, u_1) < \delta$ und dass beide Punkte mindestens 16 Plätze in der Liste U_y auseinander liegen. O.B.d.A. kommt u_0 vor u_1 in der Liste U_y. Da nur höchstens ein Punkt aus U pro Quadrat der Unterteilung möglich ist, müssen somit mindestens drei Zeilen der Unterteilung zwischen u_0 und u_1 liegen. Allerdings muss jedes Punktepaar, was durch drei Zeilen von Quadraten der Länge $\frac{\delta}{2}$ getrennt ist, mindestens Abstand $3\frac{\delta}{2}$ haben. Dies ist ein Widerspruch! □

Im obigen Beweis ist noch etwas Spielraum für Verbesserung, doch wichtig ist nur, dass 15 eine Konstante ist.
Um herauszufinden, ob es ein Knotenpaar $q_0 \in L$ und $q_1 \in R$ gibt, die näher als δ beisammen liegen, müssen wir somit nur die Liste U_y durchgehen und jeden Punkt mit den Punkten 15 Plätze davor und danach vergleichen und uns die kleinste gefundene Distanz merken.
Falls wir ein solches Punktepaar in U finden, dann geben wir das Paar mit kleinstem Abstand aus, falls nicht, dann das Paar, welches ursprünglich δ definiert hat.

Vergleich zur 1D-Version: Vom Prinzip her machen wir also doch eine Verallgemeinerung unseres $1D$-Ansatzes. Der Unterschied ist nur, das wir nach den rekursiven Aufrufen den Wert von δ kennen!

Korrektheit: Die Korrektheit kann leicht per Induktion über die Anzahl der Knoten in P gezeigt werden. Der Induktionsanfang sind Punktemengen mit höchstens 3 Punkten und für diese können wir das gesuchte Punktepaar direkt ermitteln.
Für den Induktionsschritt verwenden wir die obige Argumentation, dass das gesuchte Punktepaar entweder komplett auf einer Seite der Schnittlinie liegt, oder durch diese getrennt ist. In beiden Fällen, finden wir das Punktepaar.

Gesamtkosten: Wir sortieren einmal zu Beginn die Punktemenge nach x- und y-Koordinate. Dies kostet uns $\mathcal{O}(n \log n)$. Die restlichen Kosten erfassen wir mit der Rekurrenz

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n), \quad T(3) = T(2) = T(1) \in \mathcal{O}(1)$$

und somit haben wir Gesamtkosten in $\mathcal{O}(n \log n)$.

8 Computational complexity and efficient computation

In class, so far, we have designed and analysed algorithms for a variety of computational problems. Our main concern was to give an upper bound on the run time of the algorithms, in order to solve the problems efficiently. We will call a problem tractable (or efficiently solvable) if there exists an algorithm that solves the problem within polynomial time. Examples of efficient problems are sorting an array, finding an element in a sorted array or deciding whether there exists a path between two vertices of a graph.
It turns out that there is a large number of problems that we are not able to solve efficiently (i.e. no good algorithm is known). The majority of these problems — which arise in optimisation, artificial intelligence, combinatorics, logic, and elsewhere — have the following property. If we are given a solution to these problems we can verify, efficiently, that the solution is indeed a valid solution for our problem. For the purposes of this lecture, problems that have polynomial time algorithms are problems that belong in the class P. Problems for which we can check if a solution is valid within polynomial time are the problems of NP.
It is a big open question whether $\mathsf{P} = \mathsf{NP}$, though it is strongly conjectured that $\mathsf{P} \neq \mathsf{NP}$. In this course we will provide strong evidence that many problems in NP have no polynomial time algorithm.

8.1 Reductions

We first define a notion for comparing problems in terms of their hardness. That is we want to be able to say that problem B is at least as hard as problem A. A formal way of doing this is through a reduction.

Definition 15. *Given two problems A and B we say that A reduces to B ($A \leq_p^T B$) under polynomial-time Turing reductions if the is a polynomial-time algorithm that solves A using polynomial-time calls to a polynomial-time algorithm for B.*

Immediate from the above definition is the following observation.

Beobachtung 29. *If B can be solved in polynomial time and $A \leq_p^T B$ then A can also be solved in polynomial time.*

We denote with $A \equiv_p^{\mathrm{T}} B$, when $A \leq_p^{\mathrm{T}} B$ and $B \leq_p^{\mathrm{T}} A$

8.2 Decision problems

The problems we will be concerned in this lecture will be mainly decision problems. In a decision problem (it can also be seen as a language) we are required to answer with "yes" or "no". Most problems we have seen in this course are optimisation or search problems. It turns out that the decision version of a problem is as hard as the optimisation version of the problem. That is for most problems A it holds that $A_{\mathrm{opt}} \leq_p^{\mathrm{T}} A_d$. To see this consider the following problem.

Name: IS

Input: A graph G

Output: A maximum size independent set

The decision version of the problem is the following:

Name: IS(D)

Input: A graph G and an integer k.

Output: "Yes" if G has an independent set of size at least k, "No" otherwise.

To see that IS$\leq_p^{\mathrm{T}}$IS (D) assume that we have an oracle A where we can ask if a graph G with n vertices has an independent set of size at least k. We first find the size k_0 of the maximum independent set using at most $\log n$ calls to A. Any independent set of G has at most n vertices so we can perform binary search to decide on the size k_0 of the maximum independent set of G.
In order to find the independent set of size k_0 we use at most n calls to A as follows: Let $v \in V$ and now call $A(G - v, k_0)$. If the answer is yes then there is an independent set of G of size k_0 that does not contain v so our instance can become $(G - v, k_0)$. If the answer is no, then every independent set of size k_0 must contain v, so we need to include v in our independent set and our instance can now become $(G - v, k_0 - 1)$. We continue this process recursively until we find an independent set of size k_0.

8.3 Independent set, vertex cover and clique

From now on we will focus exclusively on decision problems so we will drop the decision version indicator from IS(D) and simply denote it as IS. Consider the following problem.

Name: VC

Input: A graph G and an integer k.

Output: "Yes" if G has a vertex cover of size at most k, "No" otherwise.

Theorem 30. VC $\equiv_p^T$ IS

Beweis. Let (G, k) be an instance of IS, where $n = |V|$. $(G, n-k)$ is an instance for VC, such that if the answer is yes to the VC problem if and only if the answer is yes to the independent set problem. To see this notice that if S is an independent set of G then $V \setminus S$ is a vertex cover for G. To see this notice that for every pair of vertices $u, v \in S$ there can be no edge between u and v. Hence $V \setminus S$ covers every edge in G. The inverse also holds. □

We have also discussed in class the following problem:

Name: Clique

Input: A graph G and an integer k.

Output: "Yes" if G has a clique of size at least k, "No" otherwise.

Theorem 31. Clique $\equiv_p^T$ IS

Beweis. Let $\bar{G}$ be the graph defined as follows. $V(\bar{G}) = V(G)$ and $E(\bar{G}) = \{\{u, v\} \mid \{u, v\} \notin E(G)\}$. Let (G, k) be an instance of IS, then $(\bar{G}, k)$ is an instance of Clique such that if the answer is yes to the Clique problem if and only if the answer is yes to the independent set problem. It is easy to see that if S is an independent set of size k for G, then S is a clique of size k for $\bar{G}$. □

8.4 NP-complete problems

Now we will define the notion of NP-completeness, and provide evidence that NP-complete problems are unlikely to have a polynomial time algorithm. We will also see examples of problems that are NP-complete.

8.4.1 Reductions

We have already defined polynomial time Turing reductions. For the purposes of this part we will define a more restrictive notion of reduction. For a decision problem A and $x \in \Sigma_A^*$ we will denote $x \in A$ if x is a yes instance of A.

Definition 16. *Given two decision problems A and B we say that A reduces to B ($A \leq_p^m B$) under polynomial-time many-one (Karp) reductions if there exists a polynomial-time computable function $f : \Sigma_A^* \to \Sigma_B^*$, such that for every $x \in \Sigma_A^*$, $x \in A$ if and only if $f(x) \in B$.*

The notion of hardness we will use is the following.

Definition 17. *We say that a problem A is* NP*-hard if $B \leq_p^m A$ for every $B \in$ NP. We say that A is* NP*-complete if A is* NP*-hard and $A \in$ NP.*

Theorem 32. *1. If $A \leq_p^m B$ and $B \leq_p^m C$ then $A \leq_p^m C$.*

2. If A is NP*-hard and $A \in$ P, then* P = NP*.*

3. If A is NP*-complete, then $A \in \mathsf{P}$ if and only if $\mathsf{P} = \mathsf{NP}$.*

Beweis. 1. Let $f : \Sigma_A^* \to \Sigma_B^*$ be the function for $A \leq_p^{\mathrm{m}} B$ and $g : \Sigma_B^* \to \Sigma_C^*$ be the function for $A \leq_p^{\mathrm{m}} C$. Consider the function $h : \Sigma_A^* \to \Sigma_C^*$, where $h(x) = g(f(x))$. h is computable in polynomial time as both f and g are computable in polynomial time. Furthermore, $x \in A$ if and only if $f(x) \in B$ and $f(x) \in B$ if and only if $g(f(x)) \in C$, which concludes the proof for A.

2. Let $X \in \mathsf{NP}$. Since A is NP-hard, $X \leq_p^{\mathrm{m}} A$. Let f be the function associated with the reduction $X \leq_p^{\mathrm{m}} A$. $A \in \mathsf{P}$, so there is an algorithm that runs in polynomial time deciding, for every $x_a \in \Sigma_A^*$ whether $x_a \in A$. But then we can decide in polynomial time if $x \in X$, by using the polynomial time to decide if $f(x) \in A$. Thus every $X \in \mathsf{NP}$ has a polynomial-time algorithm, so $\mathsf{P} = \mathsf{NP}$.

3. This is left as an exercise. □

From 2 (3) of the Theorem 32 we can see that if a problem is NP-hard (complete) then it cannot have a polynomial time algorithm unless $\mathsf{P} = \mathsf{NP}$. It is strongly believed that $\mathsf{P} \neq \mathsf{NP}$, hence NP-hardness (completeness) is considered to be strong evidence that a problem does not have a polynomial-time algorithm.
From 1 of Theorem 32 we see that in order to show that a problem A is NP-hard, it is sufficient to show that for some known NP-hard problem B, $B \leq_p^{\mathrm{m}} A$.

8.4.2 Satisfiability

Let ϕ be a Boolean formula on n variables. For a variable x we denote with $\bar{x}$ its negation. A literal is a variable x or the negation of a variable x. A clause is a disjunction of literals. We say that ϕ is in conjunctive normal form (CNF) if ϕ is a conjunction of clauses. For example, the formula

$$(x_1) \wedge (x_1 \vee \bar{x_2}) \wedge (x_3 \vee \bar{x_5} \vee x_6) \wedge (x_4 \vee \bar{x_6} \vee x_7 \vee \bar{x_8})$$

is in CNF. Sometimes we will omit the $\wedge$ symbol and present formulas in CNF by simply listing its clauses.
The first natural problem to be shown to be NP-complete is SAT.

> Name: SAT.
>
> Input: A boolean formula ϕ with n variables in conjunctive normal form.
>
> Output: "Yes" if there exists a truth assignment to the variables of ϕ that satisfies it, "no" otherwise.

Theorem 33 (Cook '71, Levin '73)**.** SAT *is* NP*-complete.*

The proof of Cook's theorem is long and technical, so we will not go through it in class. The proof idea as stated in Wikipedia, is the following:

Given any decision problem in NP, construct a non-deterministic machine that solves it in polynomial time. Then for each input to that machine, build a Boolean expression which says that the input is passed to the machine, the machine runs correctly, and the machine

halts and answers "yes". Then the expression can be satisfied if and only if there is a way for the machine to run correctly and answer "yes", so the satisfiability of the constructed expression is equivalent to asking whether or not the machine will answer "yes".

Now consider the following special case of SAT.

> Name: kSAT.
>
> Input: A Boolean formula ϕ with n variables in conjunctive normal form, where each clause contains exactly k literals.
>
> Output: "Yes" if there exists a truth assignment to the variables of ϕ that satisfies it, "no" otherwise.

It turns out that the problem is NP-complete when $k > 2$. We show this for $k = 3$ and the proof for the rest of the values for k follow in a similar way.

Theorem 34. 3SAT *is* NP*-complete.*

Beweis. 3SAT $\in$ NP, as if we are given an assignment for a 3SAT formula ϕ we can easily check if this assignment satisfies ϕ in polynomial time. To show that 3SAT is NP-hard we reduce SAT to 3SAT in the following way.
Let ϕ' be a SAT formula, we produce a 3SAT formula ϕ, such that ϕ' is satisfiable if and only if ϕ' is satisfiable.
Every clause of ϕ' that contains exactly 3 literals is a clause of ϕ.
For every clause of ϕ' containing one literal ℓ we include the following set of clauses in ϕ:

$$(\ell \vee x \vee y)(\ell \vee \bar{x} \vee y)(\ell \vee x \vee \bar{y})(\ell \vee \bar{x} \vee \bar{y}),$$

where x, y are new variables. It is easy to see that the only way to satisfy all these clauses at once is to set ℓ as true.
For every clause of ϕ' containing two literals $(\ell_1 \vee \ell_2)$ we include the following clauses in ϕ:

$$(\ell_1 \vee \ell_2 \vee x)(\ell_1 \vee \ell_2 \vee \bar{x}),$$

where x is a new variable. It is easy to see that $(\ell_1 \vee \ell_2)$ is satisfiable if and only if $(\ell_1 \vee \ell_2 \vee x)(\ell_1 \vee \ell_2 \vee \bar{x})$ is satisfied.
Finally for every k-clause $(\ell_1 \dots \ell_k)$ of ϕ', where $k > 3$ we include the following clauses in ϕ:

$$(\ell_1 \vee \ell_2 \vee x_1)(\bar{x}_1 \vee \ell_3 \vee x_3) \dots (\bar{x}_{k-3} \vee \ell_{k-1} \vee \ell_k),$$

where for $k \in [k-3]$, x_i are new variables. It is easy to see that $(\ell_1 \dots \ell_k)$ can be satisfied if and only if all the previous clauses are satisfied. □

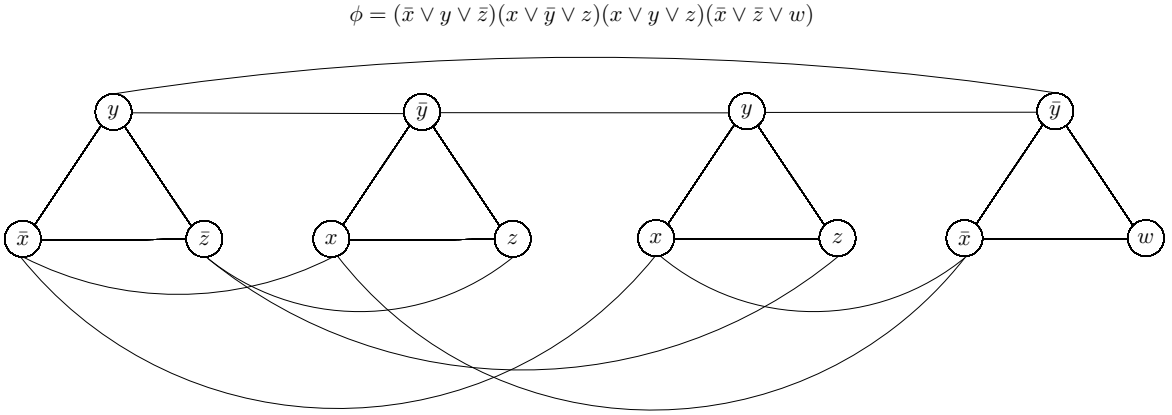


Abbildung 1: An example for 3SAT $\leq_p^{\mathrm{m}}$ IS.

8.4.3 Hard graph-theoretic problems

In this section we will show that the problems IS,VC and Clique, which we defined in the previous lecture, are all NP-complete. In the previous class we saw that IS $\equiv_p^{\mathrm{T}}$ VC $\equiv_p^{\mathrm{T}}$ Clique. In fact the proofs we gave actually show that IS $\equiv_p^{\mathrm{m}}$ VC $\equiv_p^{\mathrm{m}}$ Clique. Thus establishing NP-hardness for one of them is equivalent to establishing NP-hardness for all of them.
It is also easy to see that IS,VC and Clique are all in NP (exercise). For an integer n we denote with $[n]$ the set $\{1 \dots n\}$.

Theorem 35. IS *is* NP*-complete*

Beweis. It is easy to see that IS $\in$ NP. To prove the theorem we will show that 3SAT $\leq_p^{\mathrm{m}}$ IS. Let ϕ be a 3CNF formula, with n variables and m clauses. We will construct a graph G with $3m$ vertices which will be our input for IS as follows (see Figure 1 for an example of the construction). For each clause $c_i = \ell_i^1 \vee \ell_i^2 \vee \ell_i^3$, we add 3-clique with vertices ℓ_i^1, ℓ_i^2 and ℓ_i^3 in G. Thus, $V(G) = \{\ell_i^j \mid i \in [m] \wedge j \in [3]\}$. For $i, i' \in [m]$ and $j, j' \in [3]$, if $\ell_i^j = \neg \ell_{i'}^{j'}$ we add the edge $\{\ell_i^j, \ell_{i'}^{j'}\}$ in G. The graph G can be clearly constructed in polynomial time in n.
Our instance for IS is (G, m). If ϕ is satisfiable, then there is an assignment that satisfies every clause in ϕ. By choosing one of the positive literals for each formula we can find an independent set of size m in G. For a variable $x \in \phi$ we can't have both x and $\bar{x}$ being true simultaneously so for the vertices x and $\bar{x}$ they will not both be in the independent set of size m for G.
Now assume that there is an independent set of size m in G, then ϕ is satisfiable. Let I be the independent set. By setting each literal that corresponds to the vertices of I to true we can satisfy every clause in ϕ, hence ϕ is satisfiable. Once again the edges of the form $\{x, \bar{x}\}$ ensure that we will not evaluate both x and $\bar{x}$ true at the same time as they cannot both be in I. □

8.4.4 A hard numerical problem

Consider the decision problem SubsetSum, whose search version we have seen in a previous lecture.

Name: SUBSETSUM

Input: A set of integers S and an integer k.

Output: "Yes" if there exists a set $S' \subseteq S$, such that $\sum_{s \in S'} s = k$, "no" otherwise.

Theorem 36. SUBSETSUM*is* NP*-complete.*

Beweis. It is easy to see that SUBSETSUM $\in$ NP. If we are given S' we can easily check if $S' \subseteq S$ and if $\sum_{s \in S'} s = k$. Now we show that 3SAT $\leq_p^{\mathrm{m}}$ SUBSETSUM. Let ϕ be a 3CNF formula with n variables and m clauses, we will construct an instance of SUBSETSUM. S will contain $2n + 2m$ numbers and each number has $n + m$ digits. For each variable x_i we have two numbers t_i and f_i in S. For each clause c_j we have two numbers g_j and h_j in S. Each t_i has a 1 on the i-th digit and if x_i is positive in clause c_j, then t_i has also a 1 on the $(n + j)$-th digit; every other digit of t_i is 0. Each f_i has a 1 on the i-th digit and if x_i appears negated in clause c_j, then t_i has also a 1 on the $(n + j)$-th digit; every other digit of f_i is 0. Numbers g_j, h_j are all 0 except the $(n+j)$-th digit, which is 1. $k = 111 \ldots 133 \ldots 3$, where the 1s occur n times and the 3s occur m times. An example can be seen on the following table.

	1	2	3	4	…	n	c_1	c_2	…	c_m
t_1	1	0	0	0	…	0	1	0	…	0
f_1	1	0	0	0	…	0	0	0	…	0
t_2		1	0	0	…	0	0	1	…	0
f_2		1	0	0	…	0	1	0	…	0
t_3				1	…	0	1	0	…	0
f_3				1	…	0	0	0	…	1
⋮					⋱	⋮	⋮		⋮	⋮
t_n	0	0	0	0	…	1	0	0	…	0
t_n	0	0	0	0	…	1	0	0	…	0
g_1						1	0	0	…	0
g_1						1	0	0	…	0
g_2						0	1	0	…	0
h_2						0	1	0	…	0
⋮									⋱	⋮
g_m						0	0	0	…	1
h_m						0	0	0	…	1
k	1	1	1	1	…	1	3	3	…	3

Note that c_j denotes the $(n + j)$th digit of each number.
Assume that ϕ has a satisfying assignment, we construct a subset of S as follows. We select t_i if x_i is assigned "true" in the satisfying assignment of ϕ and we select f_i otherwise. Summing what we have selected so far we get 1 for the first n digits and the last m digits range from 1 to 3. Then for each j where the digit $n + j$ is smaller than 3, we add the numbers g_j and, if needed h_j. Hence we get a sum of k.
Now assume that we have a subset whose elements sum to k. No carries happen in any sum of the elements of S, so there must be exactly one of t_i, f_i for each i. Hence we can obtain the corresponding satisfying assignment for ϕ. □

8.4.5 Travelling Salesperson Problem

We consider the following problem.

> Name: TSP.
>
> Input: The complete graph on n vertices, weights $w_{i,j}$ for every edge of G and an integer k.
>
> Output: "Yes" if there exists a permutation π of $[n]$, such that $\sum_{i=1}^{n-1} w_{\pi(i),\pi(i+1)} + w_{\pi(n),\pi(1)} \leq k$.

We will not show that TSP is NP-complete directly, but instead will we first show that an intermediate problem is NP-complete, and then reduce this problem to TSP. The intermediate problem we will use is the following.

> Name: HC.
>
> Input: A graph G with n vertices.
>
> Output: "Yes" if there exists a simple cycle of G with n vertices (we call that a Hamilton cycle).

Lemma 19. HC *reduces to* TSP *via polynomial time many one reductions.*

Beweis. Let G be a graph with n vertices, input for HC. The input for TSP is the complete graph on n vertices, where every edge e that is an edge of G has weight $w_e = 1$ and every non-edge of G that is present in the complete graph has weight 2. Let $k = n$. If G has a Hamilton cycle then there is a cycle on the complete graph, that visits all the vertices and has total edge weight n. Vice versa if such a cycle exists in the complete graph, then it must only use edges with weight one, which corresponds to a Hamilton cycle for the graph G. □

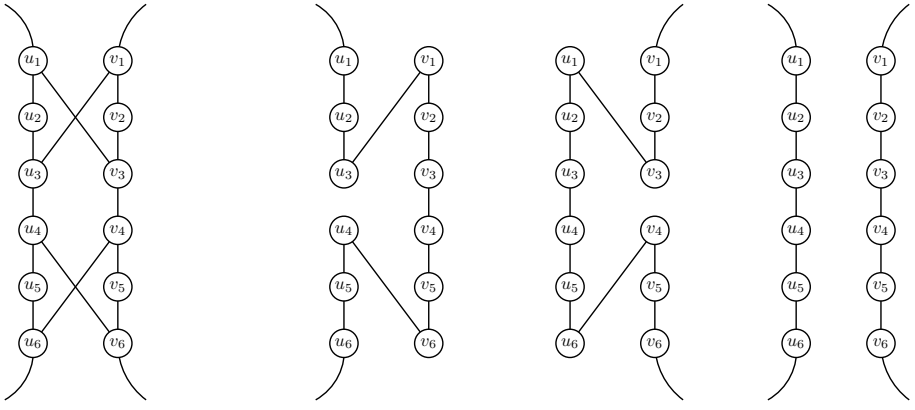


Abbildung 2: The edge gadget corresponding to the edge (u, v) and the 3 ways of traversing all its edges.

Lemma 20. VC *reduces to* HC *via polynomial time many one reductions.*

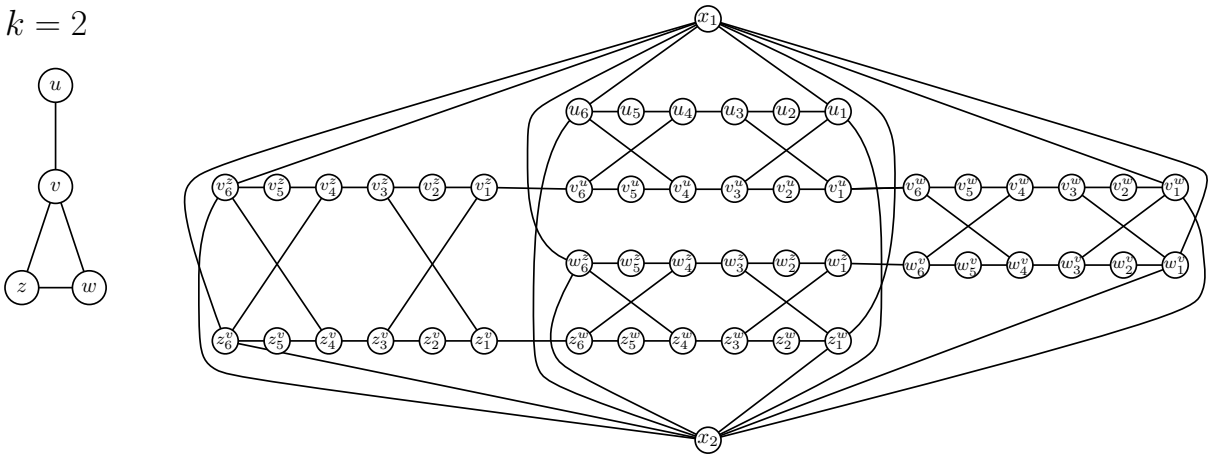


Abbildung 3: An example of the construction of G' from G as described in the proof of Lemma 20

Beweis. Let G, k be an input for VC. To describe how the graph G', which will be the input for HC, is constructed, we first display in Figure 2 the edge gadget. A key property of the edge gadget, as shown in Figure 2, is that there are exactly 3 ways of traversing all the edges of the edge gadget.
The construction of G' is as follows. For every edge u, v of G we have a copy of the edge gadget in G'. If vertex v is present on more than one edges, we connect the multiple edge gadgets with an edge. For example if v is contained in edges (u, v), (v, w) and (v, z), we will have three copies of the edge gadget. Let $v_1^u, \ldots, v_6^u$ be the corresponding vertices of the edge gadget for the edge (u, v), let $v_1^w, \ldots, v_6^w$ be the corresponding vertices of the edge gadget for the edge (v, w) and let $v_1^z, \ldots, v_6^z$ be the corresponding vertices of the edge gadget for the edge (v, z). G' will additionally have the edges (v_6^u, v_1^w) and (v_6^w, v_1^z). Hence the vertices of G' corresponding to v form a path. Finally, G' has k more vertices $x_1, \ldots x_k$, where for every vertex v, each x_i is connected to the first and the last vertex of the v path. An example of a G' can be seen in Figure 3.
First assume that G has a vertex cover $A \subseteq V(G)$ of size k. Then G' has a Hamilton cycle. The Hamilton cycle starts from x_1 then traverses the first path corresponding to the first vertex of A (in any order), then goes to x_2 then the second path until x_k and the last "vertex path" is traversed and then back to x_1. The three ways of covering all the vertices of the (u, v)-gadget shown in Figure 2 correspond to either $u \in A$, or $v \in A$ or both $u, v \in A$. Since these three ways are the only ways of traversing all the vertices of an edge gadget, the construction ensures that a Hamilton cycle in G' can also be translated to an independent set of G. □

We can now prove the following.

Theorem 37. TSP *is* NP*-complete.*

Beweis. It is easy to see that TSP is in NP. By Lemma 20, we have that HC is NP-hard. By Lemma 19 we have that TSP is also NP-hard. □

9 Umgang mit (NP-)schweren Problemen

Die Klassifikation eines gegebenen Problems als NP-schwer besagt nur, dass es vermutlich keinen Polynomialzeitalgorithmus für das Problem geben wird. In der Praxis muss das Problem allerdings trotzdem gelöst werden und Algorithmiker können sich nicht einfach hinter der Komplexität des Problems verstecken. Es stellt sich also die folgende Frage:

> Mein Problem ist NP-schwer, was nun?

Hier ein paar der gängigsten Antworten auf diese Frage:

- Falls die Instanzgröße klein genug ist, dann löse das Problem per Brute-Force - oder besser per Brute-Force via Backtracking.
- Vielleicht handelt es sich bei der gegebenen Instanz des Problems um eine "nette" Instanz, d.h. sie hat strukturelle Eigenschaften, die bei der Lösung behilflich sind. Das Ausnutzen solcher Eigenschaften kann dann zu einem Polynomialzeitalgorithmus für diese Klasse von Instanzen führen.

  Wir haben dies schon am Beispiel von Maximum Independent Set gesehen. Falls der gegebene Graph ein Baum oder ein Intervallgraph ist, dann kann das NP-schwere Maximum Independent Set Problem sogar in $\mathcal{O}(n)$ gelöst werden!
- Vielleicht beschränkt sich die kombinatorische Expolsion des Problems nur auf gewisse Parameter. D.h. vielleicht ist, für einen gegebenen Parameter k ein $\mathcal{O}(f(k) \cdot poly(n))$ - Algorithmus möglich, wobei $f(k)$ eine beliebige Funktion ist, die von k aber nicht von n abhängt.

  Ein Beispiel für einen solchen Parameter k ist die sogenannte Baumweite, d.h. ein Maß dafür, wie Baum-artig ein gegebener Graph ist. Generell werden Algorithmen, die speziell solche Parameter ausnutzen in der *Parametrisierten Algorithmik* behandelt.
- Vielleicht muss das Problem ja gar nicht exakt gelöst werden! Vielleicht genügt eine gute Approximation der optimalen Lösung. Dies ist bei vielen praktischen Problemen der Fall - oft genügt es, gut genug zu sein. Anstatt nun aber einfach ein beliebige Heuristik einzuseten, wollen wir hier mehr: Wir wollen eine Garantie, dass die ermittelte Lösung nah an der optimalen Lösung ist. Solche Algorithmen nennt man *Approximationsalgorithmen* und auf diese werden wir uns nun fokussieren.

9.1 Greedy Approximationsalgorithmen

Bevor wir zu den ersten Approximationsalgorithmen kommen, müssen wir zunächst definieren, was wir unter der genannten Garantie der Approximationsgüte verstehen.

9.1.1 Approximationsgüte

Sei X eine Instanz eines Optimierungsproblems, d.h. wir wollen für X eine Zielfunktion minimieren oder maximieren. Sei $OPT(X)$ der Wert der optimalen Lösung für Instanz

X, d.h. $OPT(X)$ ist der Wert, den wir durch Brute-Force erhalten würden. Sei $Alg(X)$ der Wert, den unser Approximationsalgorithmus Alg für die Instanz X ermittelt hat.
Der Algorithmus Alg ist genau dann ein $\alpha(n)$-Approximationsalgorithmus, falls für alle Eingaben mit Größe n gilt:

$$\frac{OPT(X)}{Alg(x)} \leq \alpha(n) \quad \text{und} \quad \frac{Alg(x)}{OPT(X)} \leq \alpha(n).$$

Die Funktion $\alpha(n)$ wird auch *Approximationsfaktor* genannt und ein Algorithmus mit Approximationsfaktor $\alpha(n)$ heißt auch $\alpha(n)$-approximativ. Z.B. liefert ein 2-approximativer Algorithmus bei einem Minimierungsproblem eine Lösung die höchstens doppelt so groß ist, wie die optimale Lösung.

9.1.2 Vertex Cover

Wir betrachten ein weiteres NP-schweres Problem, welches wir auch schon angetroffen haben: Vertex Cover. Zur Erinnerung, bei der Optimierungsversion von Vertex Cover sollen so wenige Knoten wie möglich ausgewählt werden, so dass jede Kante mindestens einen ausgewählten Endpunkt hat, d.h. so dass jede Kante "ge-covert" ist.
Auch hier betrachten wir zunächst einen sehr einfachen Greedy-Algorithmus: Die Idee ist, dass Knoten mit hohem Knotengrad viel Kanten abdecken. Somit könnten wir solche Knoten einfach greedy wählen, den Knoten und alle benachbarten Kanten aus dem Graph entfernen und dies iterieren bis alle Kanten abgedeckt sind. Wir nennen den Algorithmus GreedyVertexCover.
Auch dieser banale Algorithmus ist recht gut:

Theorem 38. *GreedyVertexCover ist $\mathcal{O}(\log n)$-approximativ.*

Beweis. Sei G eine Graph mit m Kanten und sei G_i der Graph G nach i Iterationen von GreedyVertexCover und sei d_i der Maximalgrad im Graph G_{i-1}. Außerdem sei m_i die Anzahl der Kanten im Graph G_i und sei C^* das optimale Vertex Cover von G, welches OPT viele Knoten enthält.
Da C^* auch ein Vertex Cover für den Graph G_{i-1} ist, folgt

$$\sum_{v \in C^*} deg_{G_{i-1}}(v) \geq m_{i-1}).$$

Hierbei ist $deg_{G_{i-1}}(v)$ der Knotengrad von Knoten v im Graph G_{i-1}.
Anders formuliert, bedeutet dies, dass der durchschnittliche Knotengrad in G_{i-1} eines jeden Knotens aus C^* mindestens m_{i-1}/OPT ist. Somit hat G_{i-1} mindestens einen Knoten mit Grad mindestens m_{i-1}/OPT. Da d_i der Maximalgrad über alle Knoten von G_{i-1} ist, gilt $d_i \geq m_{i-1}/OPT$.
Außerdem gilt für jedes $j \geq i-1$, dass der Subgraph G_j nur höchstens so viele Kanten wie Graph G_{i-1} hat. Somit gilt auch $d_i \geq m_j/OPT$. Damit folgt nun

$$\sum_{i=1}^{OPT} d_i \geq \sum_{i=1}^{OPT} \frac{m_{i-1}}{OPT} \geq \sum_{i=1}^{OPT} \frac{m_{OPT}}{OPT} = m_{OPT} = m - \sum_{i=1}^{OPT} d_i.$$

Damit haben wir gezeigt, dass die ersten OPT vielen Iterationen von GreedyVertexCover mindestens die Hälfte aller Kanten von G entfernt. Wenden wir dies nun wiederholt an, dann folgt, dass nach höchstens $OPT \cdot \log m \leq 2 \cdot OPT \cdot \log n$ vielen Iterationen alle Kanten von G entfernt wurden. Somit berechnet GreedyVertexCover ein Vertex Cover der Größe $\mathcal{O}(OPT \cdot \log n)$. □

Man kann auch eine passende untere Schranke von $\Omega(\log n)$ für den Approximationsfaktor von GreedyVertexCover zeigen. D.h. wir haben den Algorithmus bestmöglich analysiert.

Trivialer Approximationsalgorithmus für Vertex Cover: Das VertexCover Problem kann man noch viel besser und einfacher approximieren. Wir nennen den folgenden Algorithmus TrivialVertexCover:

- Starte mit einem leeren Vertex Cover C
- So lange G noch mindestens eine Kante hat, wähle füge beide Endpunkte der Kante zu C hinzu und entferne beide Knoten aus dem Graph.
- Gib C aus, falls G keine Kanten mehr hat.

Der Algorithmus ist korrekt, da jedes Vertex Cover - also auch das kleinste - mindestens einen Endpunkt von jeder Kante des Graphen enthalten muss. Wir wählen hier einfach beide Endpunkte und somit wird das konstruierte Vertex Cover C höchstens doppelt so groß wie das optimale Vertex Cover. Somit haben wir:

Theorem 39. *TrivalVertexCover ist 2-approximativ.*

Erstaunlicherweise ist TrivialVertexCover der beste bekannte Approximationsalgorithmus für VertexCover! Einen $(2-\varepsilon)$-approximativen Algorithmus zu finden, ist eines der großen ungelösten Probleme in diesem Bereich. Es gibt sogar einige komplexitätstheoretische Aussagen, die auf der Annahme beruhen, dass man VertexCover in Polynomialzeit nicht besser als mit Faktor 2 approximieren kann[109].

9.1.3 Knapsack

Als Warnung, dass Greedy-Approximationsalgorithmen auch schief gehen können, betrachten wir das NP-schwere Knapsack Problem[110], welches wie folgt definiert ist:

> **Geg.:** n Gegenstände, jeweils mit Gewicht g_i und Wert w_i. Eine Gewichtsschranke G für den Rucksack.
>
> **Aufgabe:** Finde die Teilmenge der Gegenstände, die die Gewichtsschranke G nicht verletzt und gleichzeitig den größten Gesamtwert hat.

Ein naheliegender Greedy-Algorithmus, genannt GreedyKnapsack wäre folgender:

- Entferne Gegenstände, die Gewicht größer als G haben.
- Sortiere alle Gegenstände absteigend nach Wert pro Kilogramm Gewicht.

[109] Ein Stichwort hierzu ist die sogenannte *unique games conjecture*.
[110] Siehe Übung, dort hieß das Problem SKIURLAUBKOFFERRAUM.

- Packe die Gegenstände in dieser Sortierung in den Rucksack, bis zum ersten mal ein Gegenstand i die Gewichtsschranke verletzt.
- Ignoriere i und alle weiteren Gegenstände.

Der Algorithmus wirkt plausibel, doch es gilt folgendes:

Theorem 40. *Es gibt keine Funktion $\alpha(n)$ für die GreedyKnapsack $\alpha(n)$-approximativ ist. D.h. GreedyKnapsack kann beliebig schlecht werden!*

Beweis. Wir betrachten die folgende generische Instanz:

$$w_1 = g_1 = 1, w_2 = k - 1, G = g_2 = k.$$

Damit gilt $\frac{w_1}{g_1} > \frac{w_2}{g_2}$ und es ist nicht möglich beide Gegenstände einzupacken. GreedyKnapsack wählt nur Gegenstand 1, optimal wäre es aber, nur Gegenstand 2 zu wählen. Somit gilt

$$\frac{OPT}{ALG} = \frac{k-1}{1}.$$

Für beliebig großes k wird somit das Verhältnis beliebig groß. □

Interessanterweise kann man den obigen beliebig schlechten Approximationsalgorithmus leicht reparieren. Wir nennen die neue Version CleverGreedyKnapsack:

- Führe GreedyKnapsack aus
- Nimm die bessere der beiden folgenden Lösungen: Entweder die Lösung von GreedyKnapsack oder Gegenstand i, d.h. der erste Gegenstand, den GreedyKnapsack nicht mehr einpacken konnte, allein.

Mit diesem einfachen Trick erhalten wir das folgende erstaunliche Ergebnis:

Theorem 41. *CleverGreedyKnapsack ist 2-approximativ.*

Beweis. Sei ALG der Wert, den CleverGreedyKnapsack eingepackt hat und sei OPT der optimale Wert. Wir zeigen, dass $OPT/ALG \leq 2$.
Sei $1, \ldots, n$ die nach Wert pro Gewicht absteigend sortierte Folge der n Gegenstände. Sei i der erste Gegenstand, der nach dieser Reihenfolge nicht mehr in den Rucksack passt. Da wir nach Wert pro Gewicht sortiert haben, gilt:

$$OPT < w_1 + w_2 + \cdots + w_i.$$

Der Grund hierfür ist, dass es bestmöglich ist, wenn wir die Gegestände 1 bis $i-1$ einpacken und dann noch so viel wie möglich vom Gegenstand i "abschneiden", um exakt die Gewichtsschranke von G zu treffen. Natürlich dürfen wir in unserem Modell keine Gegenstände zerschneiden, aber die Überlegung hilft uns trotzdem: Angenommen wir nehmen einen Bruchteil von $\frac{1}{c}$ von Gegenstand i und erhalten dafür den Bruchteil $\frac{1}{c}$ von w_i. Da nach Annahme $\frac{w_i}{g_i} \geq \frac{w_j}{g_j}$ für alle $j > i$ gilt, folgt dass

$$OPT' = w_1 + \cdots + w_{i-1} + \frac{w_i}{c} < w_1 + \cdots + w_i.$$

Offensichtlich gilt in unserem Modell, dass $OPT \leq OPT'$, d.h. Gegenstände zerschneiden hilft uns zu einer evtl. besseren Lösung, aber natürlich ist jede Lösung ohne Zerschneiden der Gegenstände noch immer eine zulässige Lösung im Modell mit erlaubten Schnitten.
Nun überlegen wir uns, wieviel Wert CleverGreedyKnapsack mindestens einpackt: Es wird dies bessere der Alternativen $1, \ldots, i-1$ bzw. i allein gewählt. Somit gilt

$$ALG = \max\{w_1 + w_2 + \cdots + w_{i-1}, w_i\}.$$

Da $a + b \leq 2\max\{a, b\}$ für beliebige a und b, folgt, dass

$$OPT < w_1 + \cdots + w_i \leq 2\max\{w_1 + w_2 + \cdots + w_{i-1}, w_i\} = 2 \cdot ALG.$$

□

9.1.4 Druckjobscheduling (ein vorletztes Mal)

Wir betrachten wieder eine Variante des Druckjobscheduling. Diesmal befinden wir uns in einer Großdruckerei und wir haben m identische Drucker zur Verfügung. Wir erhalten n Druckjobs $j_1, \ldots, j_n$, die jeweils eine Dauer ℓ_i haben und wir wollen alle Jobs so auf die Drucker aufteilen, dass wir so schnell wie möglich alle Jobs abarbeiten[111]. Wir wollen also den sogenannten *makespan* minimieren. Wir nennen das Problem deshalb Makespanminimierung.

> **Geg.:** n Jobs jeweils mit ihrer Dauer.
>
> **Ges.:** Eine Aufteilung A der Jobs auf die m Maschinen, so dass der am spätesten gedruckte Job so früh wie möglich fertig wird. Hierbei sei A ein Array, in dem für jeden der n Jobs steht auf welche der m Maschinen er gedruckt werden soll, d.h. $A[j] = i$ bedeutet Job j wird auf Maschine i gedruckt. Es soll folgende Zielfunktion minimiert werden:
>
> $$makespan(A) = \max_{1 \leq i \leq m} \sum_{A[j]=i} \ell_j.$$

Makespanminimierung ist NP-schwer, sogar wenn nur zwei Drucker zur Verfügung stehen, doch wir verzichten hier auf den Beweis. Unser Ziel ist es, einen Approximationsalgorithmus zu entwerfen.
Wir probieren es mit der einfachst möglichen Idee und wir nennen diesen Algorithmus GreedyMakespan

- Gehe die Jobs der Reihe nach durch.
- Weise jeden Job auf den momentan am wenigsten ausgelasteten Drucker zu.

Erstaunlicherweise ist GreedyMakespan schon extrem gut.

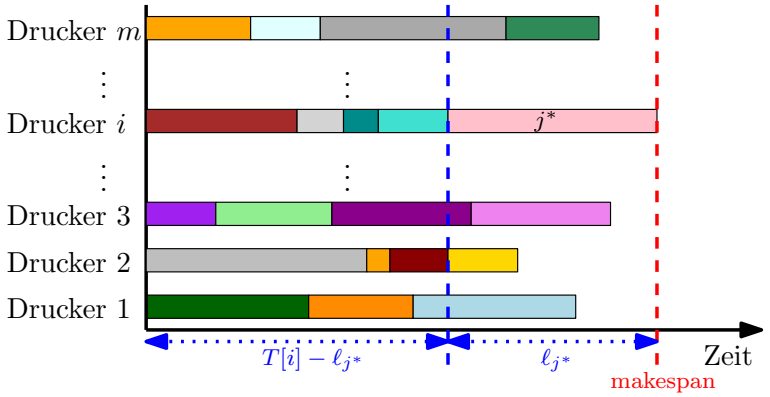
Theorem 42. *GreedyMakespan ist 2-approximativ.*

[111] Wir nehmen hier an, dass alle Drucker frei sind und somit zur selben Zeit mit dem Drucken beginnen können.

Beweis. Sei A die von GreedyMakespan gefundene Zuweisung der Druckjobs zu Druckern und sei $T[i] = \sum_{A[j]=i} \ell_j$ die Gesamtlast von Drucker i unter der Druckjobaufteilung A. Sei OPT der makespan einer optimalen Zuweisung.
Wir betrachten zunächst OPT etwas genauer. Erstens gilt, dass der optimale makespan mindestens der Dauer des längsten Jobs entsprechen muss, da auch die optimale Zuweisung den längsten Druckjob drucken muss. Zweitens gilt, dass OPT mindestens so groß ist, wie die Gesamtlänge aller Druckjobs geteilt durch m. In diesem Fall würde alles perfekt aufgeteilt werden. Wir haben also:

$$OPT \geq \max_j \ell_j \quad \text{und} \quad OPT \geq \frac{1}{m} \sum_{i=1}^{n} \ell_i.$$

Nun betrachten wir die Zuweisung A. Angenommen Drucker i druckt unter A am längsten und sei j^* der letzte Job, der unter unserem Algorithmus an Drucker i zugewiesen wurde. Da $OPT \geq \max_j \ell_j$, folgt $OPT \geq \ell_{j^*}$. Somit müssen wir nur noch zeigen, dass $T[i] - \ell_{j^*} \leq OPT$ gilt.



Job j^* wurde laut Algorithmus an Drucker i zugewiesen, da dieser zum Zeitpunkt, als j an die Reihe kam, der am wenigsten ausgelastete Drucker war. D.h. $T[i] - \ell_{j^*} \leq T[k]$ für alle Drucker $1 \leq k \leq m$. Insbesondere muss somit $T[i] - \ell_{j^*}$ kleiner oder gleich dem Durchschnitt aller Druckergesamtlasten sein. Somit haben wir

$$T[i] - \ell_{j^*} \leq \frac{1}{m} \sum_{k=1}^{m} T[k] = \frac{1}{m} \sum_{i=1}^{n} \ell_i \leq OPT.$$

Somit haben wir gezeigt, dass $T[i] = \ell_{j^*} + T[i] - \ell_{j^*} \leq 2 \cdot OPT$ gilt. □

Tatsächlich können wir den obigen Beweis benutzen, um eine leicht stärkere Aussage zu erhalten. Im letzten Teil des Beweises haben wir Druckjob j^* mit in die obere Schranke einbezogen, obwohl wir die Länge von $T[i] - \ell_{j^*}$ abgeschätzt haben. Wenn wir Durckjob j^* aus der Betrachtung entfernen, dann erhalten wir

$$T[i] - \ell_{j^*} \leq \frac{1}{m} \sum_{i \neq j^*} \ell_i \leq OPT.$$

Somit ist der erzeugte makespan bei

$$\ell_{j^*} + \sum_{i \neq j^*} \frac{\ell_i}{m} = \left(1 - \frac{1}{m}\right) \ell_{j^*} + \sum_{i=1}^{n} \frac{\ell_i}{m} \leq \left(2 - \frac{1}{m}\right) \cdot OPT.$$

Diese obere Schranke wird also immer besser, je weniger Drucker zur Verfügung stehen. Mit einem einfachen Trick, können wir sogar noch besser als GreedyMakespan sein: Wir sortieren erst alle Jobs absteigend nach Länge und dann lassen wir GreedyMakespan darauf los. Wir nennen dies SortedGreedyMakespan. Sortieren hilft:

Theorem 43. *SortedGreedyMakespan ist $\frac{3}{2}$-approximativ.*

Beweis. Sei i der Drucker, der unter der von SortedGreedyMakespan berechneten Zuweisung A am längsten druckt. Falls nur ein Durckjob an Drucker i zugewiesen wurde, dann muss A sogar optimal sein und wir sind fertig. Nehmen wir also an, dass j^* der letzte Job ist, der an Drucker i zugewiesen wurde. Da jeder der ersten m Jobs vom Algorithmus an einen anderen Drucker zugewiesen wird (alle Drucker haben anfangs keine Last) muss $j^* \geq m+1$ gelten. Genau wie im Beweis von Theorem 42 wissen wir, dass $T[i]-\ell_{j^*} \leq OPT$ gilt.
Unter jeder Druckjobzuweisung müssen mindestens zwei Druckjobs, Jobs p und q, zum selben Drucker zugewiesen werden. Somit gilt $\ell_p+\ell_q \leq OPT$. Da $\max\{p,q\} \leq m+1 \leq j^*$ gilt und die Durckjobs absteigend nach Länge sortiert waren, folgt

$$\ell_{j^*} \leq \ell_{m+1} \leq \ell_{\max\{p,q\}} = \min\{\ell_p, \ell_q\} \leq \frac{OPT}{2}.$$

Somit gilt, dass $T[i] \leq \frac{3}{2}OPT$. □

Die obige Analyse kann noch verbessert werden. Tatsächlich gilt, dass SortedGreedyMakespan $\frac{4}{3}$- approximativ ist. Wir werden aber gleich sehen, dass es sogar noch viel besser geht.

9.2 Beiliebig gute Approximation

Zum Abschluß betrachten wir noch Approximationsalgorithmen, die *beliebig* nah an das gesuchte Ergebnis kommen. D.h. Algorithmen, die $(1+\varepsilon)$-approximativ sind, für beliebiges $0 < \varepsilon$. Solche Algorithmen nennt man *polynomial time approximation scheme (PTAS)* und es handelt sich hier eigentlich um eine Familie von Algorithmen - für jedes ε sieht der Algorithmus nämlich leicht anders aus.

Druckjobscheduling (das letzte Mal!)

Wir wollen einen noch besseren Approximationsalgorithmus für die Makespanminimierung beim Druckjobscheduling entwickeln. Zur Erinnerung: Wir haben bereits gezeigt, dass der Algorithmus GreedyMakespan einen makespan von

$$\ell_{j^*} + \sum_{i \neq j^*} \frac{\ell_i}{m}$$

erzeugt. Wir sehen also, dass der makespan von der Länge des Jobs j^*, welcher als letztes fertig wird, abhängt. Es wäre also eine gute Idee, wenn der zuletzt fertig werdende Druckjob kurz ist!
Die Idee können wir wie folgt umsetzen: Zunächst definieren wir, was ein langer und was ein kurzer Druckjob ist. Dann führen wir folgenden Algorithmus, genannt MakespanPTAS, aus:

- Berechne per Brute Force den optimalen Schedule für die langen Druckjobs.
- Füge dann noch die kurzen Durckjobs per GreedyMakespan hinzu.

Wir müssen noch definieren, wann ein Druckjob kurz bzw. lang ist:

Definition 18. *Ein Druckjob j ist kurz genau dann, wenn*

$$\ell_j \leq \frac{1}{km} \sum_{i=1}^{n} \ell_i.$$

Aus dieser Definition folgt, dass es höchstens km viele lange Druckjobs geben kann. Außerdem folgt damit, dass GreedyMakespan Kosten von $\mathcal{O}(m^{km} \cdot m * n + m \log n)$ hat. Der erste Term ergibt sich aus den m^{km} vielen Möglichkeiten, die km vielen langen Druckjobs auf die m Maschinen aufzuteilen und dann jeweils bei allen Druckern nachzuschauen, wie lange sie insgesamt arbeiten. Der zweite Term ist eine obere Schranke für die Kosten von GreedyMakespan, wenn wir die Auslastung der Drucker in einer Priority Queue verwalten, um schnell an den am wenigsten ausgelasteten Drucker zu kommen. Falls m konstant ist, dann ist MakespanPTAS ein Polynomialzeitalgorithmus!
Für die Approximationsgüte erhalten wir folgendes:

Theorem 44. *MakespanPTAS(k) ist ein $\left(1 + \frac{1}{k}\right)$-approximativer Polynomialzeitalgorithmus, falls $m \in \mathcal{O}(1)$.*

Beweis. Sei $ALG(k)$ der makespan, der von MakespanPTAS(k) erzeugt wird. Es gibt zwei Fälle für den Druckjob j^*, welcher als letztes fertig wird:

- Falls j^* lang ist, dann bedeutet dies, dass der makespan von einem Drucker bestimmt wird, auf dem nur lange Druckjobs abgearbeitet wurden. Da wir für die langen Druckjobs den optimalen Schedule bestimmt haben, folgt, dass in dem Fall sogar ein optimaler makespan erzeugt wird.
- Falls j^* kurz ist, dann wissen wir, dass $\ell_{j^*} \leq \frac{1}{km} \sum_{i=1}^{n} \ell_i$ gilt. Somit gilt für den erzeugten makespan:

$$ALG(k) \leq \frac{1}{km} \sum_{i=1}^{n} \ell_i + \frac{1}{m} \sum_{i \neq j^*} \ell_i \leq \left(1 + \frac{1}{k}\right) \sum_{i=1}^{n} \frac{\ell_i}{m} \leq \left(1 + \frac{1}{k}\right) OPT.$$

□

Tatsächlich kann man die Einschränkung auf konstant viele Drucker noch entfernen und ein PTAS sogar für den Fall mit beliebig vielen Maschinen erweitern. D.h. es gibt ein PTAS für das Druckjobscheduling, welches Kosten hat, die polynomiell in n und m sind. Die Idee hierfür ist, dass wir ja im obigen Beweis im ersten Fall eigentlich etwas zu streng waren. Dort wurde der optimale Schedule für die langen Druckjobs benutzt aber es hätte auch eine $(1 + 1/k)$-Approximation genügt. Dies kann man ausnutzen, um das PTAS auf den allgemeinen Fall zu erweitern. Die Details hierfür würden aber den Rahmen dieser Vorlesung sprengen.