ONLINE ALGORITHMEN II.

Das Ski Problem

Wir fahren in den Skiurlaub. Wir wissen, dass der Urlaub an einem Tag T plötzlich endet, wir wissen aber nicht, an welchem. (Diesen Tag bestimmt die Zukunft, unser bösartiger Gegner: der Schnee schmilzt, oder wir brechen uns das Bein. Theoretisch kann aber der Urlaub beliebig lang werden.)

Sollen wir für 1 €/Tag Skier ausleihen, oder für K € Skier kaufen (und falls ja, an welchem Tag)?

Ziel: Minimiere die Gesamtkosten!

→ Wie kann das als Online-Problem formalisiert werden?

Die Eingabe kann als eine Bitfolge der Form

$$\sigma = 1,1,1,1,1,1,\overset{T}{1},0$$

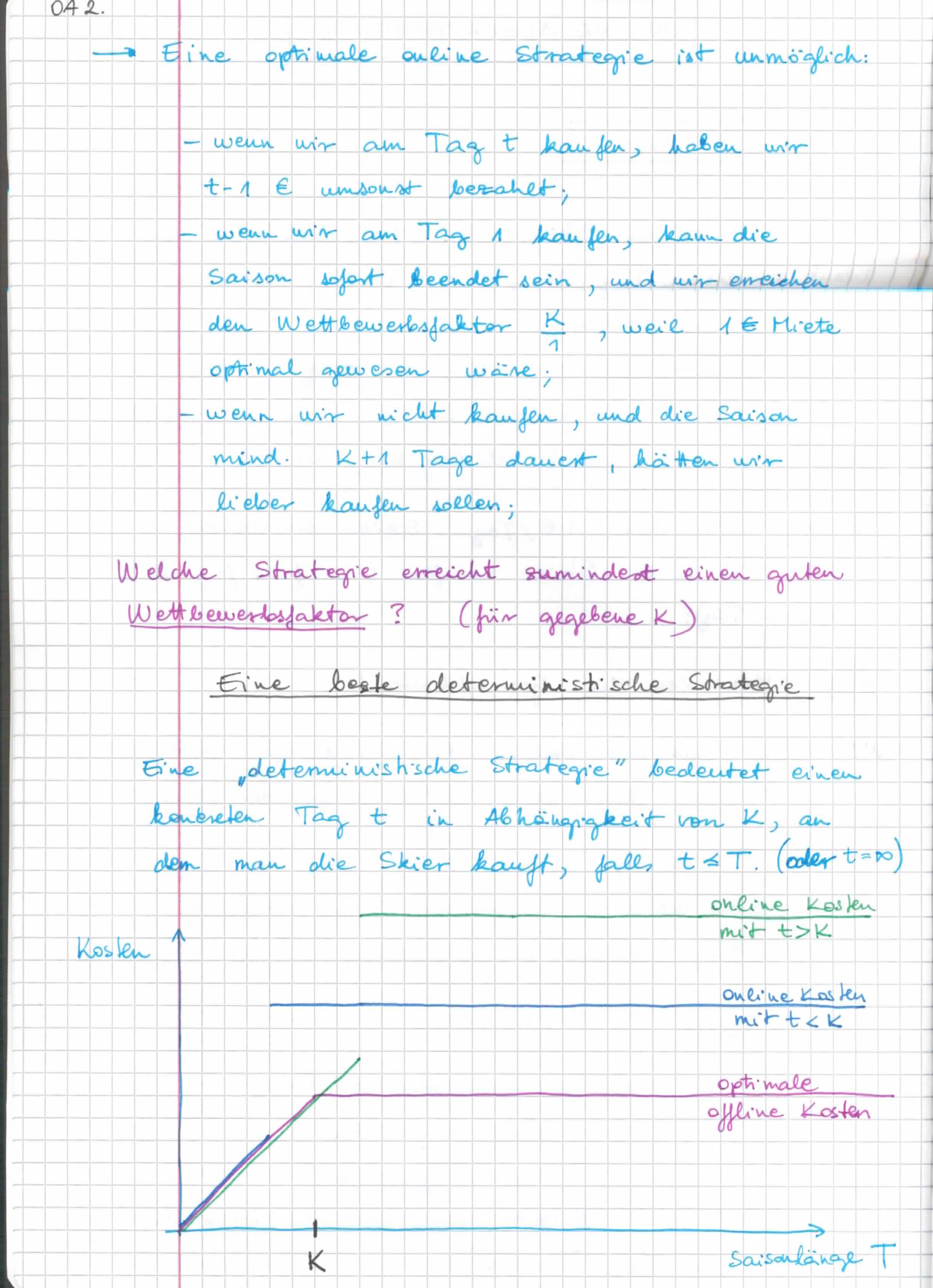
aufgefasst werden (Vorsicht! Keine Zufallsbits!)

Jeden Tag ein Bit bis 0 ankommt und die Folge endet.

Die Ausgabe $\tau = \tau_1 \tau_2 \tau_3 \ldots \tau_i \ldots$

$\tau_i \in$ { - miete am i-ten Tag
- kaufe am i-ten Tag
- schon gekauft }

(Der Einfachheit halber geben wir die Eingabe als Tag T an)



→ Eine optimale online Strategie ist unmöglich:

- wenn wir am Tag t kaufen, haben wir $t-1$ € umsonst bezahlt;
- wenn wir am Tag 1 kaufen, kann die Saison sofort beendet sein, und wir erreichen den Wettbewerbsfaktor $\frac{K}{1}$, weil 1 € Miete optimal gewesen wäre;
- wenn wir nicht kaufen, und die Saison mind. $K+1$ Tage dauert, hätten wir lieber kaufen sollen;

Welche Strategie erreicht zumindest einen guten <u>Wettbewerbsfaktor</u>? (für gegebene K)

Eine beste deterministische Strategie

Eine „deterministische Strategie" bedeutet einen konkreten Tag t in Abhängigkeit von K, an dem man die Skier kauft, falls $t \leq T$. (oder $t = \infty$)

<u>Theorem:</u> Wenn wir am Tag K kaufen (Strategie $t=K$), erreichen wir Wettbewerbsfaktor < 2, und das ist das beste das ein deterministischer online Algorithmus erreichen kann.

<u>Beweis:</u>

<u>Die Strategie $t=K$ erreicht Wettbewerbsfaktor <2:</u>

- für Eingabe $T \leq K-1$ ist die Strategie optimal, wir zahlen nur die $T \leq K-1$ Miete
- für Eingabe $T \geq K$ kauft ein optimaler Algorithmus für K € und unsere Strategie zahlt $K-1+K$ $\quad \frac{K-1+K}{K} < 2$ ✓

<u>Kein deterministischer online Algorithmus erreicht besseren Wettbewerbsfaktor:</u>

wir müssen für jede Strategie t die Worst-Case Eingabe T für diese Strategie betrachten (worst-case bezüglich des Wettbewerbsfaktors!)

- für $t > K$ ist $T \geq t$ Worst-Case Eingabe (für $t=\infty$ gibt es kein „Worst" case, nur immer schlechter wird $T \to \infty$)
  Die Kosten der Strategie sind $t-1+K \geq 2K$, und die optimalen Kosten sind K.
- für $t < K$ ist die Worst-Case Eingabe $T=t$
  Der Wettbewerbsfaktor ist

$$\frac{t-1+K}{t} \geq \frac{t-1+t+1}{t} = 2$$ □

(→ $T=t$ ist immer eine Worst-Case Eingabe)

OA 4.

Wiederholung: Wettbewerbsfaktor von randomisierten Algorithmen

→ Wenn wir eine randomisierte online Strategie betrachten,

für fixierte Eingabe σ analysieren wir den erwarteten Zielfunktionswert $\mathbb{E}[ALG_\sigma]$ wobei die Erwartung über die gewürfelten Zufallsbits des Algorithmus gilt. Der Wettbew.faktor für σ ist (grob) $\frac{\mathbb{E}[ALG_\sigma]}{OPT_\sigma}$

→ Die worst-case vom Wettbewerbsfaktor wird von (mind.) einer konkreten schlechtesten Eingabe σ für diesen randomisierten Algorithmus, erreicht.

→ Wie wählt der bösartige Gegner die schlechteste Eingabe?

Unser Gegner ist ein sog. blinder Gegner (oblivious adversary): er kennt unseren randomisierten Algorithmus, aber nicht die gezogenen Zufallsbits während der Berechnung des Algorithmus (wie bei offline Algorithmen)

Insbesondere, darf die Eingabe σ mitten im Ablauf, unter Kenntnis der bisherigen Ausgaben nicht geändert werden! (Es gibt durchaus andere Gegner-Modelle.)

Definition: Der Wettbewerbsfaktor eines randomisierten online Algorithmus ist α, falls für jede Eingabe σ

$$\mathbb{E}[ALG_\sigma] \leq \alpha \cdot OPT_\sigma + c$$

(bei Minimierungsproblemen).

Eine beste randomisierte Strategie für das Ski-Problem:

Eine randomisierte Strategie ist eine Verteilung (Wahrscheinlichkeiten) $\pi_1\ \pi_2\ \pi_3 \ldots \pi_t \ldots$ s.d. $\sum_i \pi_i = 1$

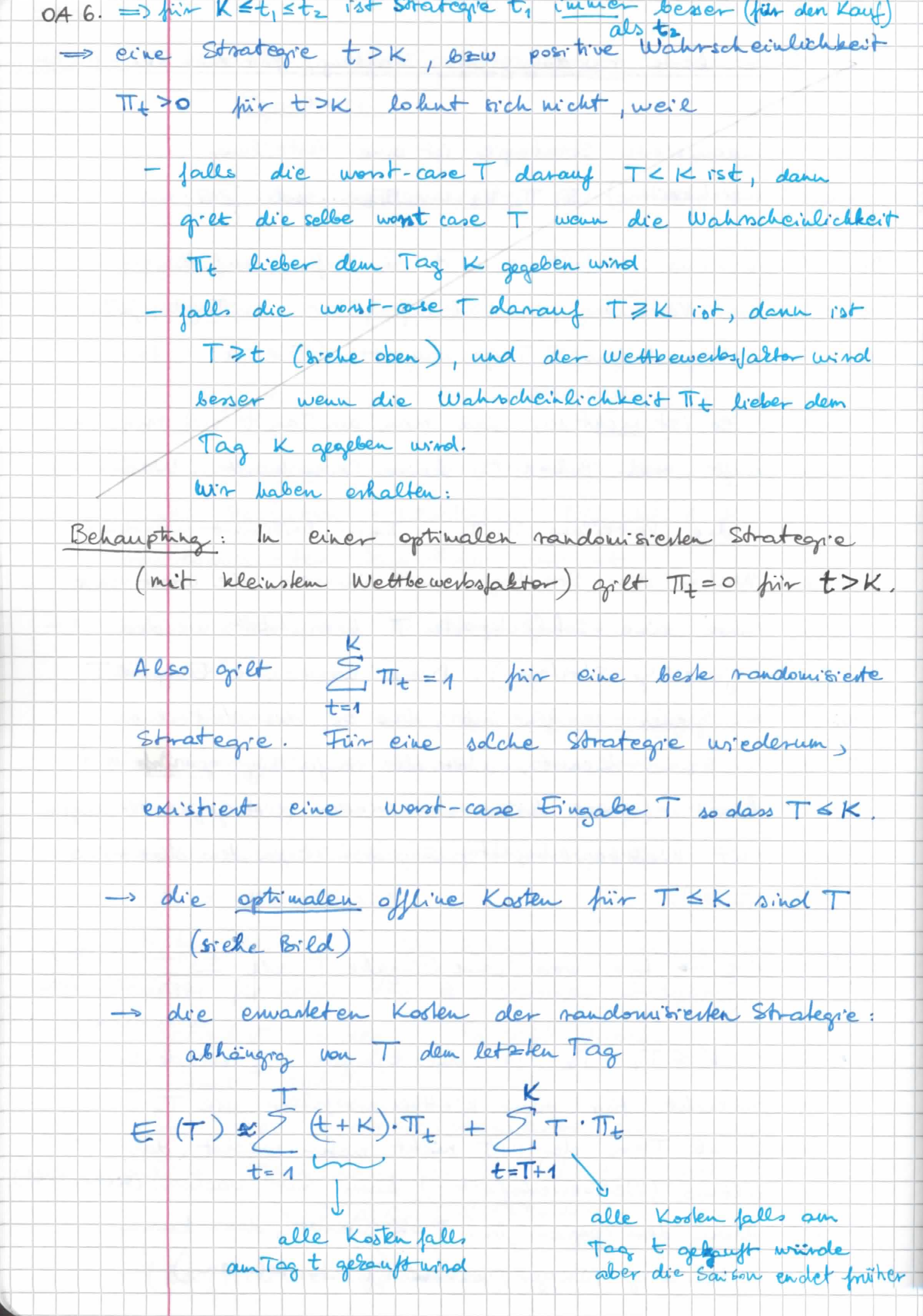
Der Algorithmus würfelt eine t am Anfang zufällig, gemäß dieser Verteilung, und kauft Skier am Tag t, falls die Saison mindestens t Tage dauert $(T \geq t)$

(Beachte: es ist nicht so, dass man am Tag t mit Prob $= \pi_t$ Skier kauft, und Prob $= 1 - \pi_t$ keine Skier kauft... Warum?)

Wie soll die Verteilung $\pi_1, \pi_2 \ldots \pi_t \ldots$ bestimmt werden?
Bei der Wahl der Strategie soll immer die worst-case Eingabe, also eine schlechteste T betrachtet werden (für die jeweilige Strategie). Diese worst-case Eingabe T wird vom Gegner so generiert, dass er die Verteilung $\pi_1, \pi_2 \ldots \pi_t \ldots$ kennt, aber die zufällig gewürfelte t nicht kennt (sonst hätte man wieder die Gegnerstrategie $T = t$ mit Wettbewerbsfaktor 2; das ist ja der deterministische Fall).

Beobachtung → Die worst-case Eingabe T soll $\frac{E(\text{online Kosten})}{\text{optimale Kosten}}$ maximieren

→ Für $K \leq T_1 < T_2$ ist die Eingabe T_2 mindestens so schlecht für den Wettbewerbsfaktor wie T_1 weil ab $T = K$ der OPT nicht mehr wächst, das Verhältnis (Wettbewerbsfaktor) kann nur wachsen (siehe Bild)
⇒ Wenn der Gegner „bis $T = K$ wartet", dann kann er beliebig länger warten ($T > K$ nehmen).



OA 6. $\Rightarrow$ für $K \le t_1 \le t_2$ ist Strategie t_1 immer besser (für den Kauf) als t_2

$\Rightarrow$ eine Strategie $t > K$, bzw positive Wahrscheinlichkeit $\pi_t > 0$ für $t > K$ lohnt sich nicht, weil

- falls die worst-case T darauf $T < K$ ist, dann gilt die selbe worst case T wenn die Wahrscheinlichkeit π_t lieber dem Tag K gegeben wird
- falls die worst-case T darauf $T \ge K$ ist, dann ist $T \ge t$ (siehe oben), und der Wettbewerbsfaktor wird besser wenn die Wahrscheinlichkeit π_t lieber dem Tag K gegeben wird.

Wir haben erhalten:

Behauptung: In einer optimalen randomisierten Strategie (mit kleinstem Wettbewerbsfaktor) gilt $\pi_t = 0$ für $t > K$.

Also gilt $\sum_{t=1}^{K} \pi_t = 1$ für eine beste randomisierte Strategie. Für eine solche Strategie wiederum, existiert eine worst-case Eingabe T so dass $T \le K$.

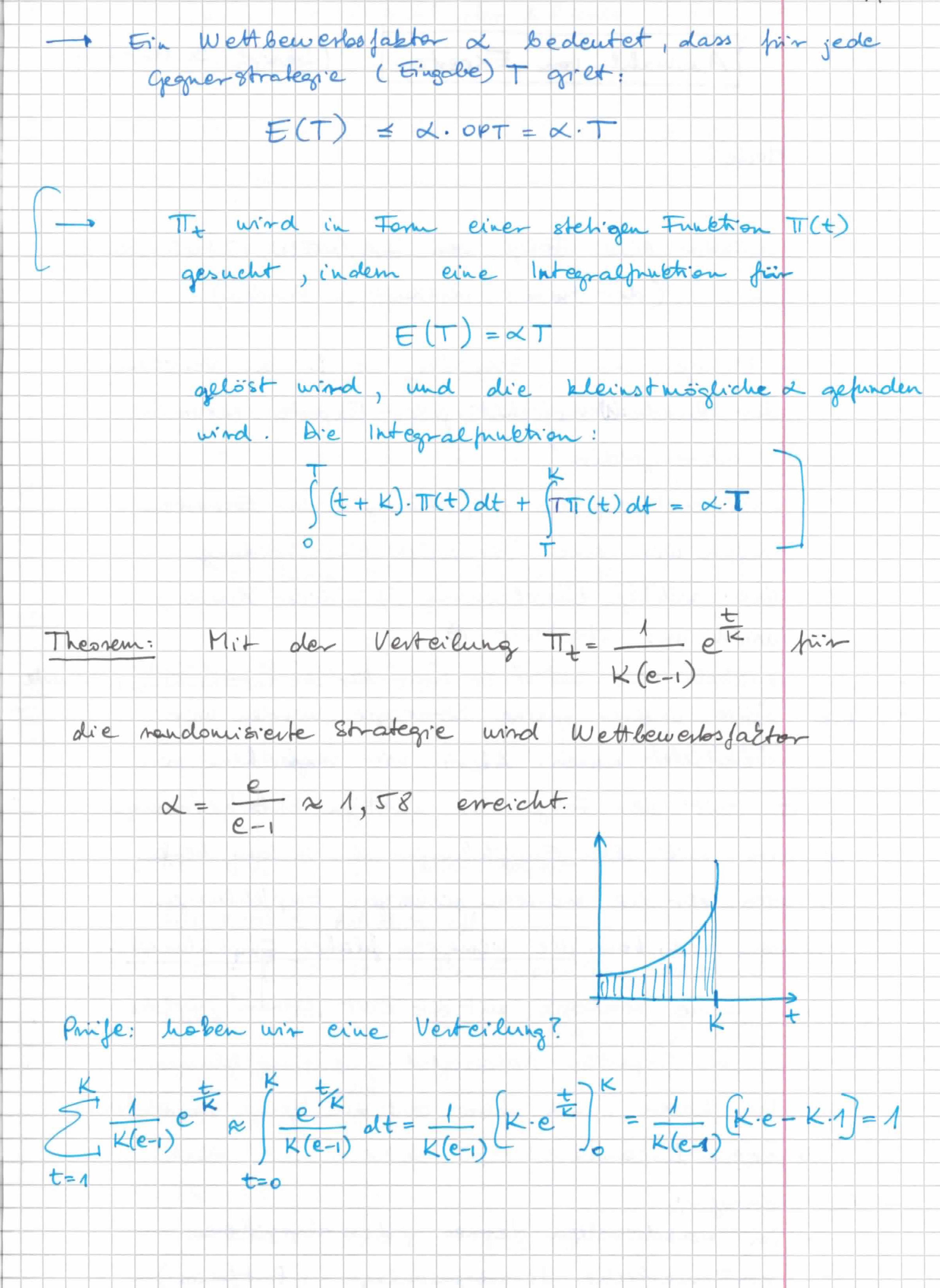
$\rightarrow$ die optimalen offline Kosten für $T \le K$ sind T (siehe Bild)

$\rightarrow$ die erwarteten Kosten der randomisierten Strategie: abhängig von T dem letzten Tag

$$E(T) = \sum_{t=1}^{T} (t+K)\cdot \pi_t + \sum_{t=T+1}^{K} T \cdot \pi_t$$

$(t+K)$: alle Kosten falls am Tag t gekauft wird

$T \cdot \pi_t$: alle Kosten falls am Tag t gekauft würde aber die Saison endet früher



→ Ein Wettbewerbsfaktor α bedeutet, dass für jede Gegnerstrategie (Eingabe) T gilt:

$$E(T) \leq \alpha \cdot OPT = \alpha \cdot T$$

→ π_t wird in Form einer stetigen Funktion $\pi(t)$ gesucht, indem eine Integralfunktion für

$$E(T) = \alpha T$$

gelöst wird, und die kleinstmögliche α gefunden wird. Die Integralfunktion:

$$\int_0^T (t+K)\cdot \pi(t)\,dt + \int_T^K T\pi(t)\,dt = \alpha \cdot T$$

Theorem: Mit der Verteilung $\pi_t = \frac{1}{K(e-1)} e^{\frac{t}{K}}$ für die randomisierte Strategie wird Wettbewerbsfaktor

$$\alpha = \frac{e}{e-1} \approx 1{,}58$$ erreicht.

Prüfe: haben wir eine Verteilung?

$$\sum_{t=1}^{K} \frac{1}{K(e-1)} e^{\frac{t}{K}} \approx \int_{t=0}^{K} \frac{e^{t/K}}{K(e-1)}\,dt = \frac{1}{K(e-1)} \left[K \cdot e^{\frac{t}{K}}\right]_0^K = \frac{1}{K(e-1)} \left[K\cdot e - K \cdot 1\right] = 1$$

Auswahl von Experten

(Skript S. 115–117.)

Das Modell

- Wir müssen eine Folge von JA/NEIN Entscheidungen treffen. (der Prozess verläuft in Runden)
- Eine Menge von n Experten steht zur Verfügung:
  in jeder Runde: (1) empfiehlt jeder Experte entweder „Ja" oder „Nein"
  (2) Basierend auf den Empfehlungen der Experten, treffen wir eine JA/NEIN Entscheidung
  (3) die richtige Entscheidung (für diese Runde) wird uns mitgeteilt ⊛
- Wir entwickeln eine online Strategie für die Entscheidung-per-Runde
- Die Güte unserer online Strategie werden wir Wettbewerbsfaktor nennen. Aber: wir vergleichen sie nicht mit der Folge der richtigen Entscheidungen, sondern mit den Empfehlungen des bislang besten Experten, der die meisten richtigen Empfehlungen gegeben hat, bzw. die wenigsten falschen Empfehlungen.

  Kann unsere Strategie mit der Zeit „lernen" welche Menge von Experten überwiegend gute Empfehlungen gegeben haben, und eher auf eine Menge von solchen Experten hören?
- Anwendungen:
  - im maschinellen Lernen, Lernalgorithmen
  - kontinuierliche Vermögensanlage (Portfolios)

⊛ **Ziel:** Minimiere die Anzahl der falschen Entscheidungen!

(Einige Bemerkungen:

die Qualität eines Experten hat keine feste Wahrscheinlichkeit;
diese Qualität kann sich mit der Zeit beliebig ändern;

⇒ die Experten sollen nach jeder Runde neu bewertet werden.)

Die Online Strategie bewertet jeden Experten i mit einem Gewicht w_i. Die Gewichte werden nach jeder Runde aktualisiert je nachdem, ob die Empfehlung des Experten richtig war:

Der Weighted Majority Algorithmus

a.) einfache, deterministische Version

$1 \; . \; . \; . \; . \; . \; i \; . \; . \; . \; . \; . \; . \; . \; n$ — n Experten

- setze $w_i = 1$ für jeden Expert
- WIEDERHOLE beliebig oft
  - Sei $x_i \in \{Ja, Nein\}$ die Empfehlung des Experten i
  - FALLS $\sum_{\{i \mid x_i = Ja\}} w_i \geq \sum_{\{i \mid x_i = Nein\}} w_i$ gib JA aus;
    SONST gib NEIN aus;
  - erhalte die richtige Entscheidung $r \in \{JA, NEIN\}$
  - FÜR $i = 1$ bis n
    - FALLS $x_i \neq r$, setze $w_i = \frac{w_i}{2}$

OA 10.

In dieser einfachen Variante haben alle Experten das gleiche Gewicht am Anfang; in jeder Runde die Gewichte der Experten die eine falsche Empfehlung gegeben haben, werden jeweils halbiert.

Wichtig ist, dass eine Mehrheitsentscheidung (oder zufällige Wahl) von Experten, die bislang überwiegend gut waren, getroffen wird. Einen Experten deterministisch zu wählen, kann schief gehen: Zeige, dass die deterministische Strategie, in jeder Runde auf einen bislang besten Experten zu hören, Wettbewerbsfaktor ∞ hat!

Analyse:

Bezeichne W_t das Gesamtgewicht aller Experten am Anfang von Runde t

$$W_1 = n$$

Beobachtung 1: Wenn unsere Entscheidung in Runde t falsch ist, dann haben Experten mit Gesamtgewicht mindestens $\frac{W_t}{2}$, falsche Empfehlung gegeben.

$\Rightarrow$ In einer Runde t mit falscher Entscheidung wird mindestens die Hälfte des Gesamtgewichts halbiert, und deshalb

$$W_{t+1} \leq \frac{1}{2} \cdot \frac{W_t}{2} + \frac{W_t}{2} = \frac{3W_t}{4}$$

$\Rightarrow$ Wenn <u>bis</u> Runde t genau f falsche Entscheidungen getroffen wurden, dann folgt

$$W_t \leq \left(\frac{3}{4}\right)^f \cdot W_1 = \left(\frac{3}{4}\right)^f \cdot n \qquad (1)$$

Wir wollen jetzt die Anzahl f der falschen Entscheidungen, mit f_{OPT}, der Anzahl falscher Entscheidungen des besten Experten, vergleichen. Wir stellen hierfür ein Verhältnis zwischen W_t und f_{OPT} fest.

<u>Beobachtung 2:</u> Wenn der beste Experte bis Runde t f_{OPT} Fehler gemacht hat, dann ist sein Gewicht nach der Runde t, $\frac{1}{2^{f_{OPT}}}$. Insbesondere gilt:

$$\frac{1}{2^{f_{OPT}}} \leq W_t \qquad (2)$$

$\Rightarrow$ Aus (1) und (2) erhalten wir

$$\frac{1}{2^{f_{OPT}}} \leq \left(\frac{3}{4}\right)^f \cdot n$$

$$\left(\frac{4}{3}\right)^f \leq 2^{f_{OPT}} \cdot n$$

$$f \cdot \log_2 \frac{4}{3} \leq f_{OPT} \cdot \log_2 2 + \log_2 n$$

<u>Theorem:</u> Wenn der Weighted Majority Algorithmus mit n Experten bis zu einem Zeitpunkt f falsche, und der bis dahin beste Experte f_{OPT} falsche Entscheidungen getroffen hat, dann gilt

$$f \leq \frac{1}{\log_2 \frac{4}{3}} f_{OPT} + \frac{\log_2 n}{\log_2 \frac{4}{3}}$$

Das Ziel war, f zu minimieren. Der deterministische Weighted Majority Algorithmus erreicht somit den Wettbewerbsfaktor $\frac{1}{\log_2 \frac{4}{3}} \approx 2{,}41$ mit additivem logarithmischem Term.

b.) eine randomisierte Version von Weighted Majority

→ In jeder Runde deterministisch auf den bislang besten Experten zu hören, hat ∞ Wettbewerbsfaktor. Wir können aber zufällig, mit Wahrscheinlichkeiten proportional zu den aktuellen Gewichten einen Experten auswählen.

→ Wir verallgemeinern noch den Algorithmus mit zwei weiteren Änderungen:

- Statt Ja/Nein Empfehlungen, erlauben wir Empfehlungen in beliebiger Form von den Experten. Nachdem das richtige Ergebnis bekannt wurde, in Runde t bewerten wir jeden Experten i mit einer Note c_i^t, wobei $0 \leq c_i^t \leq 1$, und $c_i^t = 0$ sehr gut, $c_i^t = 1$ sehr schlecht ist. (Interpretation: c_i^t Kosten wären verursacht, wenn wir die Meinung von i übernommen hätten.)
- die neuen Gewichte werden in jeder Runde so berechnet: $w_i := w_i (1 - \varepsilon \cdot c_i^t)$

wobei $0 < \varepsilon \leq \frac{1}{2}$ ein Parameter ist.

(unsere frühere Benotung entspricht dem Fall
$c_i^t \in \{0,1\}$ und $\varepsilon = \frac{1}{2}$)

ein sehr kleines ε wird die Gewichte langsamer anpassen, aber dann schließlich beliebig guten Wettbewerbsfaktor $(1+\varepsilon)$ ergeben (mit höherer additiver Konstante)

<u>Randomisierter Weighted Majority Algorithmus</u>

- setze $w_i = 1$ für jeden Experten i
- wähle den Parameter $\varepsilon \in (0, \frac{1}{2}]$
- WIEDERHOLE beliebig oft $t = 1, 2, 3, \ldots$
  - wähle einen Experten i zufällig, gemäß der Verteilung

$$p_i = \frac{w_i}{\sum_{k=1}^{n} w_k}$$

    und übernimm seine Entscheidung
  - nachdem das richtige Ergebnis bekannt wurde, und die Noten c_i^t ausgeteilt wurden, setze die Gewichte

$$w_i = w_i \left(1 - \varepsilon \cdot c_i^t\right)$$

OA 14.

Definition: Die erwarteten Kosten in Runde t sind

$$K_t = \sum_{i=1}^{n} p_i \cdot c_i^t .$$

Die erwarteten Gesamtkosten bis Runde T sind

$$K = \sum_{t=1}^{T} K_t$$

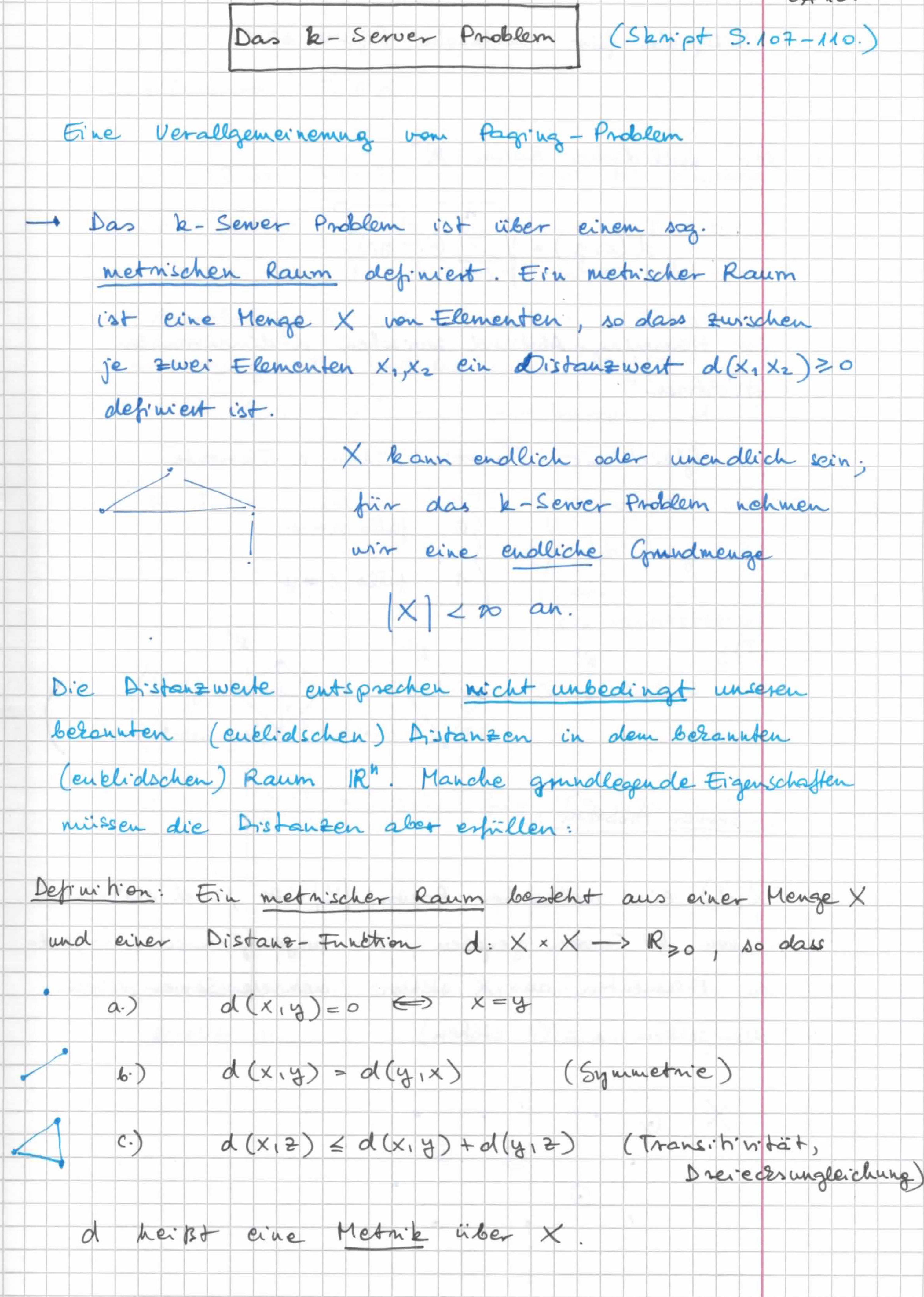
Bezeichne $K_{OPT} = \sum_{t=1}^{T} c_{OPT}^t$ die Gesamtkosten des besten Experten bis Runde T (die wären verursacht wenn wir auf Experte OPT gehört hätten in jeder Runde). K_{OPT} ist also minimal über alle i, unter $\sum_{t=1}^{T} c_i^t$ Werten

Theorem: $$K \leq (1+\varepsilon) \cdot K_{OPT} + \frac{\ln n}{\varepsilon}$$

(ohne Beweis)

→ man kann beliebig guten Wettbewerbsfaktor $1+\varepsilon$ erreichen

→ der additive Term $\frac{\ln n}{\varepsilon}$ entspricht der „Lernzeit". Ein guter Wettbewerbsfaktor braucht längere Lernzeit (vorsichtigere Anpassung der Gewichte)



Das k-Server Problem

(Skript S. 107–110.)

Eine Verallgemeinerung vom Paging-Problem

→ Das k-Server Problem ist über einem sog. metrischen Raum definiert. Ein metrischer Raum ist eine Menge X von Elementen, so dass zwischen je zwei Elementen x_1, x_2 ein Distanzwert $d(x_1, x_2) \geq 0$ definiert ist.

X kann endlich oder unendlich sein; für das k-Server Problem nehmen wir eine endliche Grundmenge $|X| < \infty$ an.

Die Distanzwerte entsprechen nicht unbedingt unseren bekannten (euklidschen) Distanzen in dem bekannten (euklidschen) Raum $\mathbb{R}^n$. Manche grundlegende Eigenschaften müssen die Distanzen aber erfüllen:

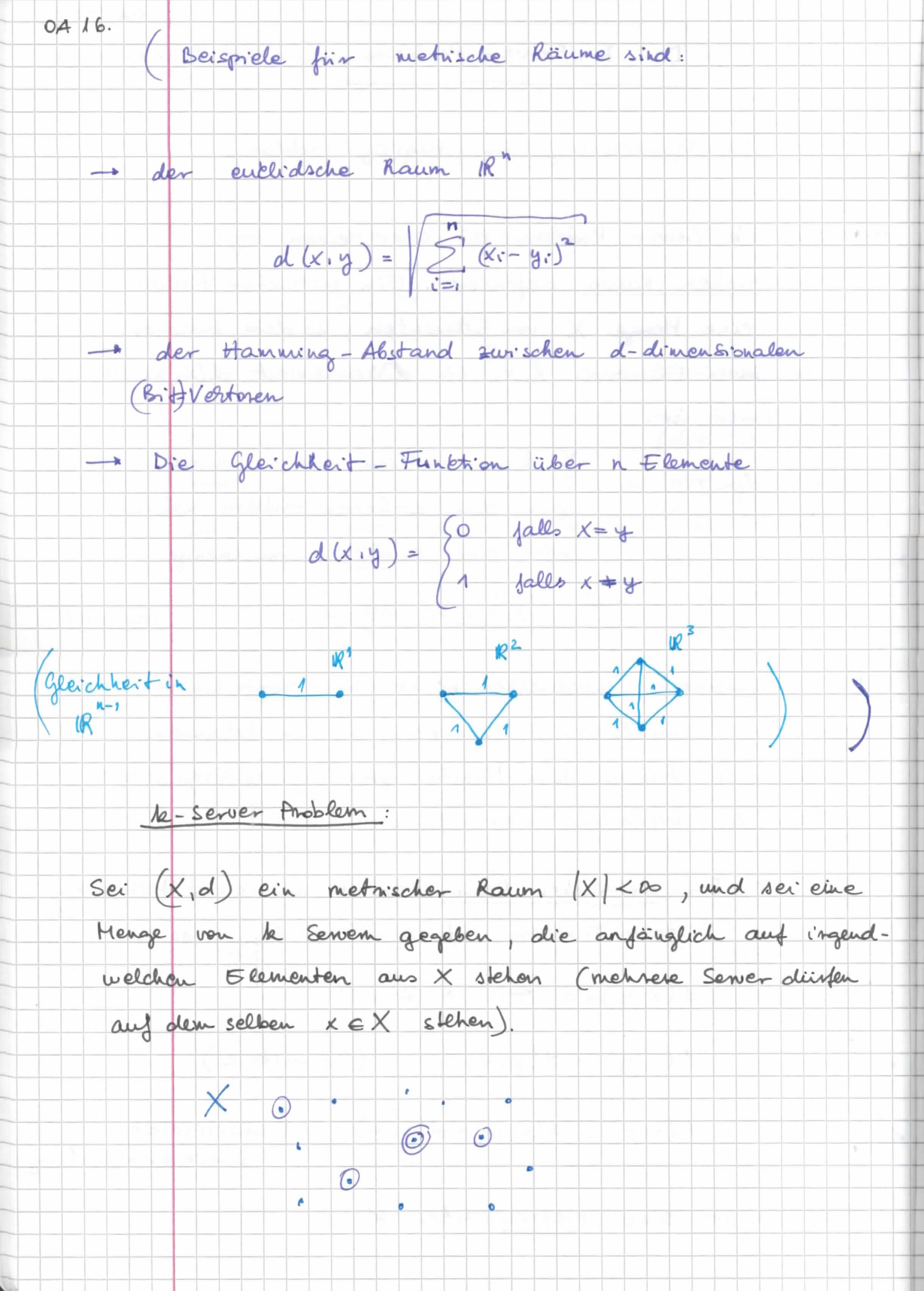
Definition: Ein metrischer Raum besteht aus einer Menge X und einer Distanz-Funktion $d: X \times X \longrightarrow \mathbb{R}_{\geq 0}$, so dass

a.) $d(x,y) = 0 \iff x = y$

b.) $d(x,y) = d(y,x)$ (Symmetrie)

c.) $d(x,z) \leq d(x,y) + d(y,z)$ (Transitivität, Dreiecksungleichung)

d heißt eine Metrik über X.



04.16.

(Beispiele für metrische Räume sind:

→ der euklidsche Raum $\mathbb{R}^n$

$$d(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

→ der Hamming-Abstand zwischen d-dimensionalen (Bit)Vektoren

→ Die Gleichheit-Funktion über n Elemente

$$d(x,y) = \begin{cases} 0 & \text{falls } x = y \\ 1 & \text{falls } x \neq y \end{cases}$$

(Gleichheit in $\mathbb{R}^{n-1}$

))

k-Server Problem:

Sei (X,d) ein metrischer Raum $|X| < \infty$, und sei eine Menge von k Servern gegeben, die anfänglich auf irgendwelchen Elementen aus X stehen (mehrere Server dürfen auf dem selben $x \in X$ stehen).

Die Eingabe $\sigma = \sigma_1 \sigma_2 \sigma_3 \ldots \sigma_i \ldots$ ist eine Folge von Elementen $\sigma_i \in X$

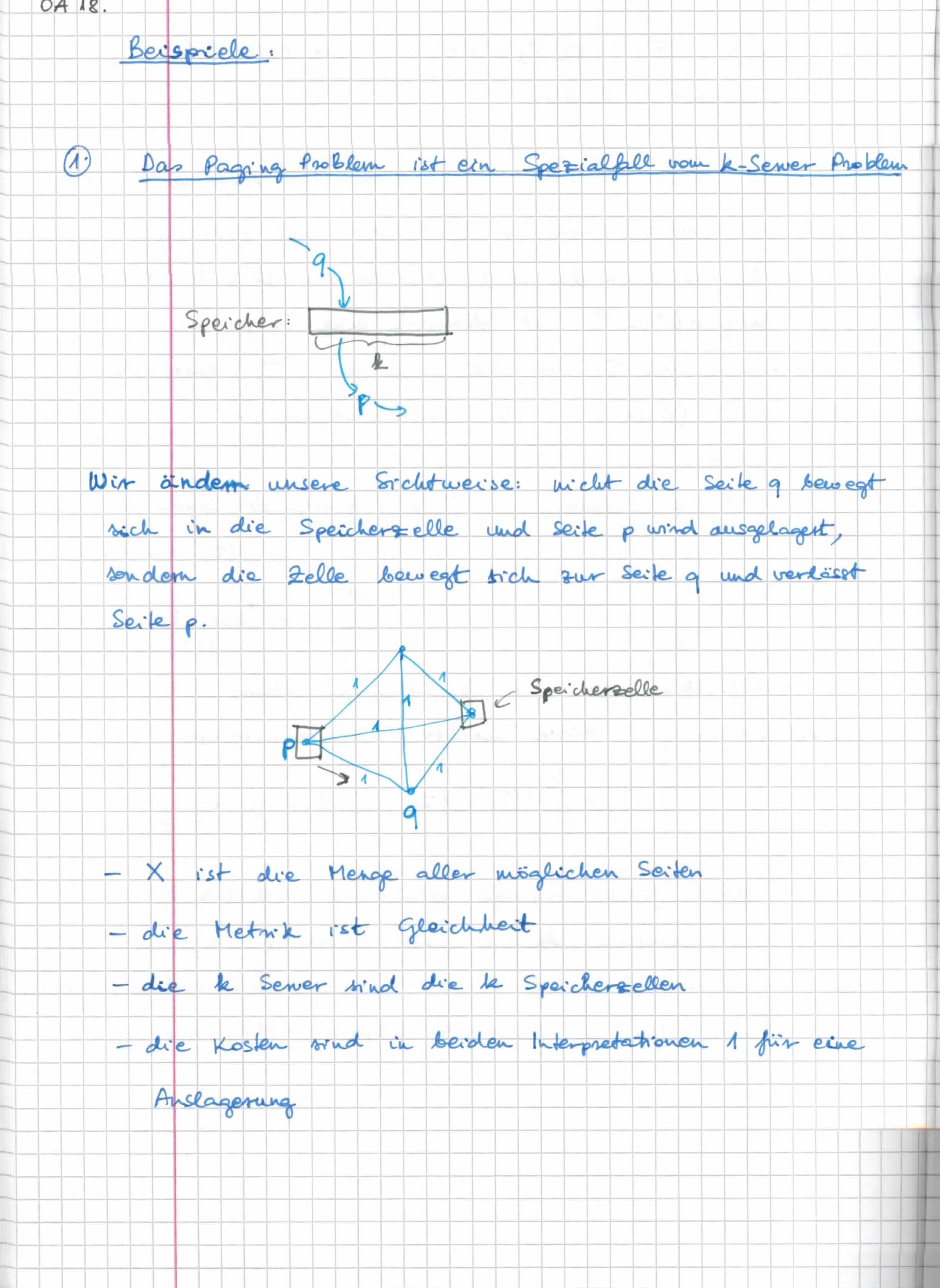
Aufgabe: Für $i = 1, 2, \ldots$ wenn die Anfrage $\sigma_i \in X$ ankommt, bewege einen der k Server auf Position σ_i (falls keiner schon da ist), von seiner aktuellen Position. Die Ausgabefolge τ ist demnach eine Folge von Serverbewegungen τ_i und die Kosten $f(\tau_i)$ entsprechen der Länge (Distanz d) der Serverbewegung, d.h.

sei $\sigma_i = x$, dann $\tau_i = \begin{cases} \text{bewege den Server von } y \text{ auf } x \\ f(\tau_i) = d(x, y) \\ \text{oder:} \\ \text{ein Server steht schon auf } x \\ f(\tau_i) = 0 \end{cases}$

Die Zielfunktion ist die Gesamtlänge aller Serverbewegungen, diese möchte man minimieren.

$$f(\tau) = \sum_{i=1}^{n} f(\tau_i) \quad \text{für} \quad \sigma = \sigma_1 \sigma_2 \ldots \sigma_n$$

Beachte: Nach jeder Anfrage genügt es (höchstens) einen Server zu bewegen. Andere Server vorzeitig irgendwo zu schicken, lohnt sich nicht (wegen der Dreiecksungleichung: der direkte Weg ist immer ein kürzester Weg).



Beispiele:

① **Das Paging Problem ist ein Spezialfall vom k-Server Problem**

Wir ändern unsere Sichtweise: nicht die Seite q bewegt sich in die Speicherzelle und Seite p wird ausgelagert, sondern die Zelle bewegt sich zur Seite q und verlässt Seite p.

- X ist die Menge aller möglichen Seiten
- die Metrik ist Gleichheit
- die k Server sind die k Speicherzellen
- die Kosten sind in beiden Interpretationen 1 für eine Auslagerung

2. Zwei Köpfe einer Festplatte

3. 4 Fahrstühle in einem Wohnturm
(Fahrstühle bewegen sich aber auch zwischenzeitlich zum Ziel des Benutzers).

4. Taxizentrale: fassen wir die Stadt als einen Graphen auf, der nächstliegende freie Taxi erfüllt die nächste Anfrage

Einfache Heuristiken

1. Greedy: erfülle die Anforderung durch den nächstliegenden Server.

Probleme

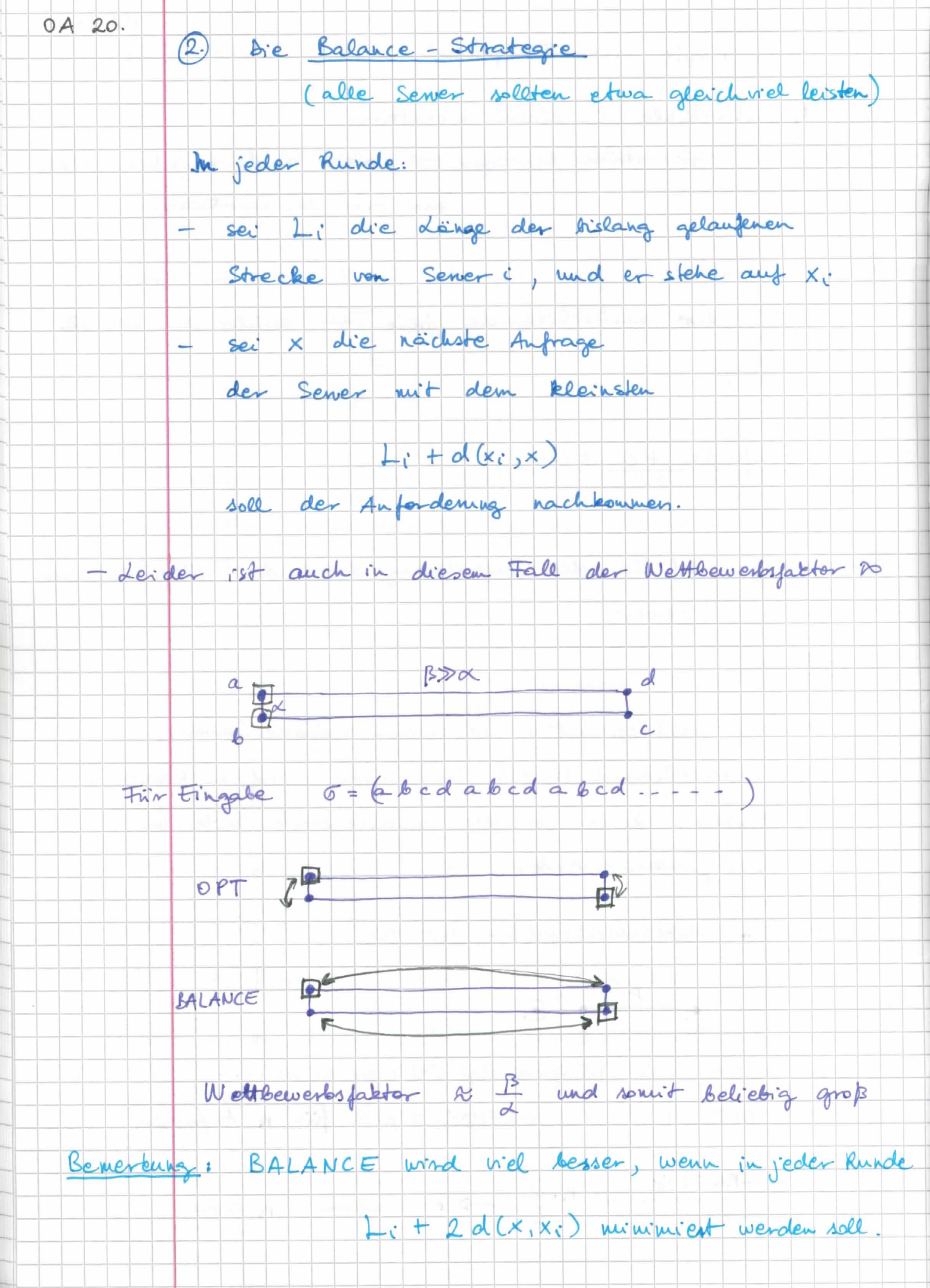
- nicht wohldefiniert: was wenn mehrere Server nächstliegend sind? Insbesondere wird im Paging-Problem keine konkrete auszulagernde Seite von Greedy ausgewählt.

- der Wettbewerbsfaktor ist ∞

Betrachte die Eingabefolge $\sigma = (b\ c\ b c\ b c\ b c \dots)$

$OPT_\sigma = \beta$ $\quad$ $GREEDY_\sigma = n \cdot \alpha$ für n Anfragen

$$\frac{GREEDY_\sigma}{OPT_\sigma} = \frac{n \cdot \alpha}{\beta} \to \infty \quad \text{wenn} \quad n \to \infty$$



(2.) Die Balance-Strategie

(alle Server sollten etwa gleichviel leisten)

In jeder Runde:

- sei L_i die Länge der bislang gelaufenen Strecke von Server i, und er stehe auf x_i
- sei x die nächste Anfrage

der Server mit dem kleinsten

$$L_i + d(x_i, x)$$

soll der Anforderung nachkommen.

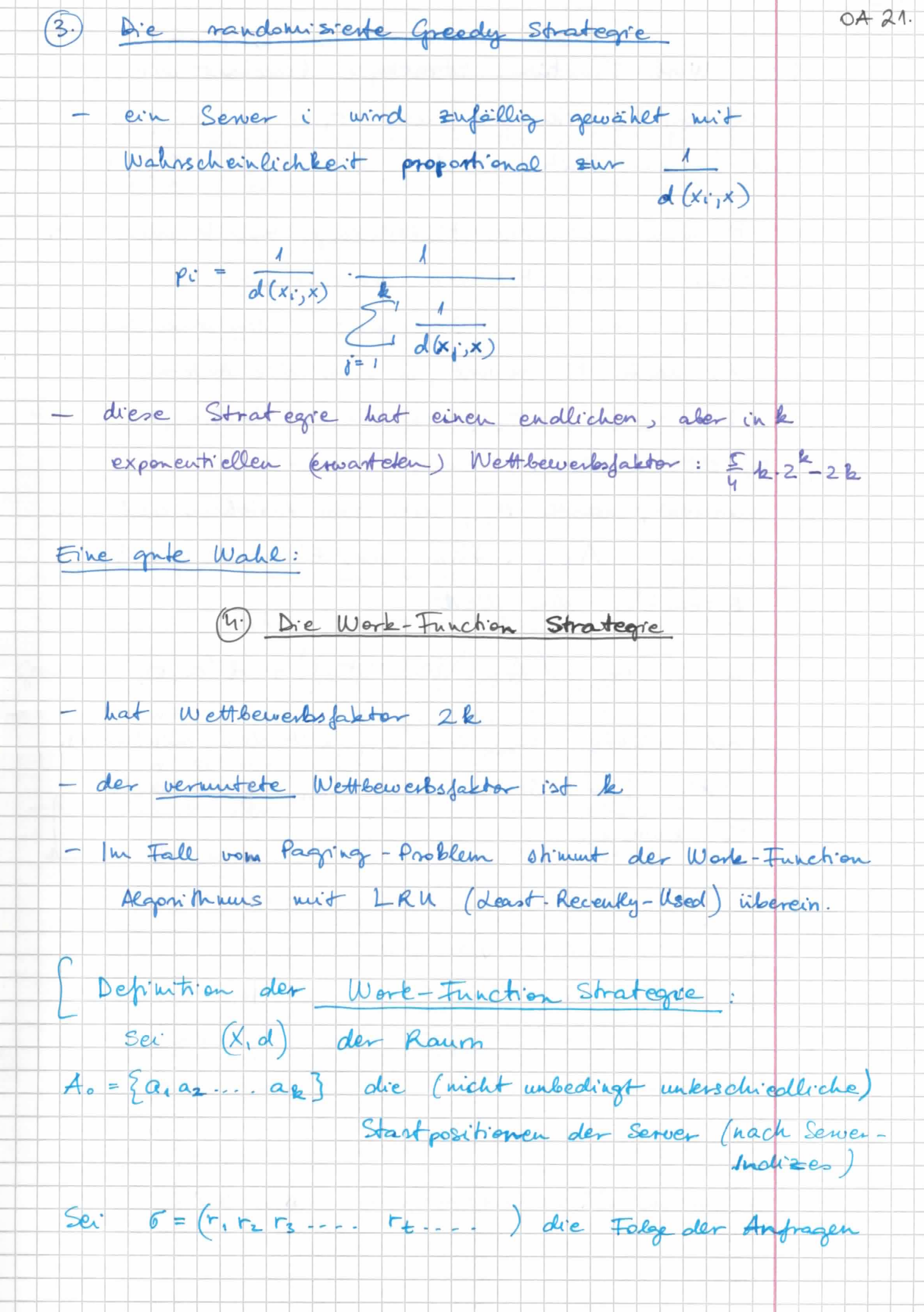
- Leider ist auch in diesem Fall der Wettbewerbsfaktor ∞

Für Eingabe $\sigma = (a\, b\, c\, d\, a\, b\, c\, d\, a\, b\, c\, d \ldots)$

Wettbewerbsfaktor $\approx \frac{\beta}{\alpha}$ und somit beliebig groß

Bemerkung: BALANCE wird viel besser, wenn in jeder Runde

$$L_i + 2\,d(x, x_i)$$ minimiert werden soll.



(3.) Die randomisierte Greedy Strategie

- ein Server i wird zufällig gewählt mit Wahrscheinlichkeit proportional zur $\frac{1}{d(x_i, x)}$

$$p_i = \frac{1}{d(x_i, x)} \cdot \frac{1}{\sum_{j=1}^{k} \frac{1}{d(x_j, x)}}$$

- diese Strategie hat einen endlichen, aber in k exponentiellen (erwarteten) Wettbewerbsfaktor: $\frac{5}{4} k \cdot 2^k - 2k$

Eine gute Wahl:

(4.) Die Work-Function Strategie

- hat Wettbewerbsfaktor $2k$
- der vermutete Wettbewerbsfaktor ist k
- Im Fall vom Paging-Problem stimmt der Work-Function Algorithmus mit LRU (Least-Recently-Used) überein.

[Definition der Work-Function Strategie:

Sei (X, d) der Raum

$A_0 = \{a_1 a_2 \dots a_k\}$ die (nicht unbedingt unterschiedliche) Startpositionen der Server (nach Server-Indizes)

Sei $\sigma = (r_1 r_2 r_3 \dots r_t \dots)$ die Folge der Anfragen

Die Work-Function Strategie berechnet die Ausgabe(n) τ_t iterativ, aus $r_1 r_2 \dots r_{t-1} r_t$ und aus den Positionen der Server nach τ_{t-1}:

Sei M eine beliebige Multimenge von k Orten in X. Bezeichne $w_{t-1}(M)$ die kleinstmögliche Gesamtlänge von Server-Bewegungen, die die Server von A_0 nach M bewegt, so dass die Anforderungen $r_1 r_2 r_3 \dots r_{t-1}$ in dieser Reihenfolge erfüllt werden.

Angenommen, nach r_{t-1} die Server stehen auf den Positionen A_{t-1}. Für jedes Element $a \in A_{t-1}$ betrachte den Wert

$$w_{t-1}\left(A_{t-1} \setminus \{a\} \cup \{r_t\}\right) + d(a, r_t)$$

und wähle $a_{min} \in A_{t-1}$ mit dem minimalen Wert. Der Server auf a_{min} wird nach r_t bewegt.

Amortisierte Kostenanalyse

Da die „Kosten" in unseren Beispielen der „Laufzeit" entsprechen, können wir das Thema auch <u>Amortisierte Laufzeitanalyse</u> nennen.

Worum geht's?

→ Wir wollen eine <u>Folge von Operationen</u> ausführen
zB. Datenstruktur-Operationen, oder andere offline/online Berechnung von Ausgabefolgen

→ manche Operationen haben viel zu große Laufzeit, aber solche Operationen kommen nur selten vor.

Genauer: Also, die Worst-Case Laufzeit der Operation wird in einer langen Operationenfolge (mit Sicherheit) nur selten realisiert.

→ Anstatt der (worst-case) Laufzeit einzelner Operationen, betrachten wir die Gesamtlaufzeit von n aufeinanderfolgenden Operationen. Diese wird also viel kleiner sein, als n-mal die worst-case Laufzeit einer Operation.
Somit ist <u>die durchschnittliche* Laufzeit einer Operation in einer Folge von n Operationen</u> viel kleiner als die worst-case Laufzeit einer Operation.

* hier reden wir über tatsächlichen Durchschnitt, der mit Zufall oder Erwartungswert nichts zu tun hat.

01.24.

Grob gesagt: → Die amortisierte Laufzeit von n Operationen

ist (eine obere Schranke für) die Gesamtlaufzeit der n (aufeinanderfolgenden) Operationen.

→ Die amortisierte Laufzeit einer Operation ist (eine obere Schranke für) die durchschnittliche Laufzeit pro Operation über n aufeinander folgende Operationen.

Lernbeispiel:

Die Paging-Strategie Flush-When-Full leert den Speicher vollständig, wenn Seitenfehler (Page-Fault) auftritt und der Speicher gerade voll ist.

!! Jetzt betrachten wir nicht die Anzahl der Seitenfehler oder den Wettbewerbsfaktor, sondern

Als Lernbeispiel weisen wir die folgenden Laufzeiten der i-ten Anfrage $\sigma_i = p$ zu:

Sei → $Laufzeit_i = 0$ wenn die Seite p im Speicher vorhanden

[p_1 p_2 p_3] ← $p = p_2$

→ $Laufzeit_i = 1$ wenn die Seite p nicht vorhanden ist, aber es noch im Speicher Platz gibt

[p_1 p_2 p_3] ← p

→ $Laufzeit_i = k+1$ wenn Seitenfehler auftritt und der Speicher vorhin geleert werden soll.

p →? [p_1 p_2 p_3 p_4 p_5 p_6 p_7 ... p_k]

Beachte, dass der Speicher nicht öfter als 1 mal pro k Anfragen geleert wird.

Angenommen, die Operationenfolge fängt mit leerem Zustand des Systems an... (dies wird meistens angenommen)

1.) Was wäre die worst-case Laufzeit von $k+1$ Operationen? Anfragen

$k+1$ Operationen haben Kosten höchstens

$$k+1+k = 2k+1$$

(max. $k+1$ für das Speichern und max k für eine Entleerung)

2.) Was wäre die worst-case Laufzeit von $n = l \cdot k + 1$ Operationen?

Grob: $l \cdot k$ Operationen haben $\leq l \cdot k + 1$ Laufzeit für die Einlagerung und $\leq l \cdot k$ Laufzeit für die höchstens l Entleerungen (l mal Laufzeit k)

Insgesamt: $\sim 2 \cdot l \cdot k = 2n$

3.) Was wäre die amortisierte Laufzeit einer Anfrage?

2 könnten wir als amortisierte Kosten pro Anfrage rechnen: $2n$ ist eine obere Schranke für die Laufzeit von n aufeinanderfolgenden Operationen; 2 ist somit obere Schranke für die durchschnittliche Laufzeit einer Operation.

Methoden der Analyse

(mathematisch sind die folgenden zwei Argumente/ Methoden äquivalent)

I. Buchhalter-Argument

- Jede Operation zahlt ihre amortisierte Kosten auf ein Konto.
- Die tatsächlichen Kosten werden sofort vom Konto abgezogen
- Da die amortisierten Kosten obere Schranke für die Durchschnittskosten einer Operation sind, reicht „das Geld" auf dem Konto immer für die tatsächlichen Kosten. Wenn die amortisierten Kosten korrekt sind, darf das Konto nie negativ sein.

Erklärung: manche Operationen zahlen mehr (amortisierte) Kosten als ihre tatsächlichen Kosten. Sie zahlen somit auch für Kosten, die von ihnen indirekt verursacht wurden und später anfallen.
Manche Operationen haben weniger amortisierte Kosten als tatsächliche Kosten. Diese bekommen Geld vom Konto um die tatsächlichen Kosten zu decken.

Das Buchhalter-Argument im Flush-When-Full Beispiel

→ Wir belasten jede Anforderung (mit Seitenfehler!) mit 2 Kosteneinheiten als amortisierten Kosten:

1 für die Einlagerung

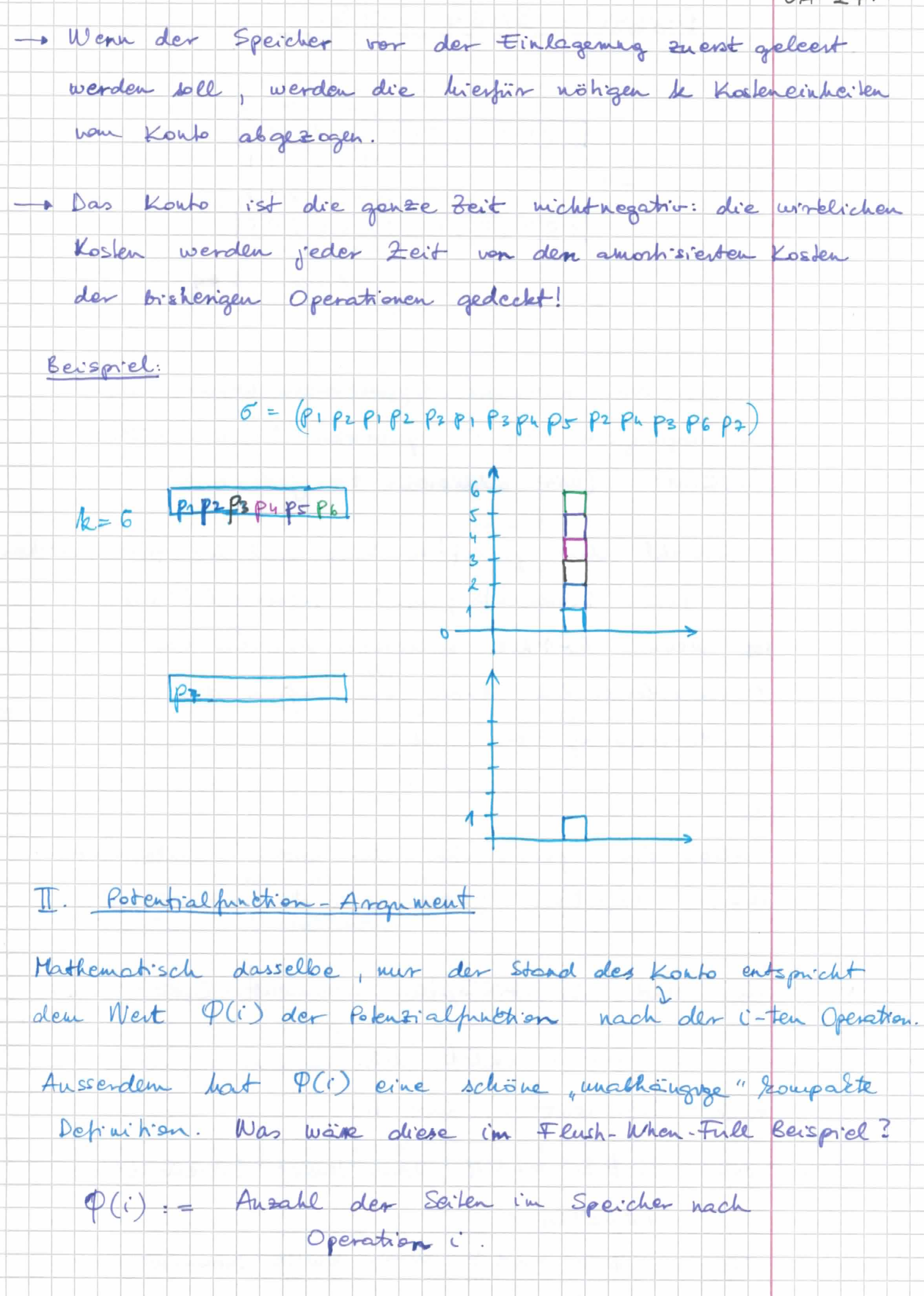
1 wird auf das Konto bezahlt, für die spätere Auslagerung dieser Seite bei der nächsten Entleerung

→ Wenn der Speicher vor der Einlagerung zuerst geleert werden soll, werden die hierfür nötigen k Kosteneinheiten vom Konto abgezogen.

→ Das Konto ist die ganze Zeit nichtnegativ: die wirklichen Kosten werden jeder Zeit von den amortisierten Kosten der bisherigen Operationen gedeckt!

Beispiel:

$$\sigma = (p_1\ p_2\ p_1\ p_2\ p_2\ p_1\ p_3\ p_4\ p_5\ p_2\ p_4\ p_3\ p_6\ p_7)$$



II. Potentialfunktion-Argument

Mathematisch dasselbe, nur der Stand des Konto entspricht dem Wert $\Phi(i)$ der Potenzialfunktion nach der i-ten Operation.

Aussendem hat $\Phi(i)$ eine schöne „unabhängige" kompakte Definition. Was wäre diese im Flush-When-Full Beispiel?

$\Phi(i)$:= Anzahl der Seiten im Speicher nach Operation i.

Beispiel: Binäre Zähler

$k-1$

1	0	0	1	0	1	0	0	0	1	1	1	1	1	1

$+1$

1	0	0	1	0	1	0	0	1	0	0	0	0	0	0

$\text{Kosten}_i = k$

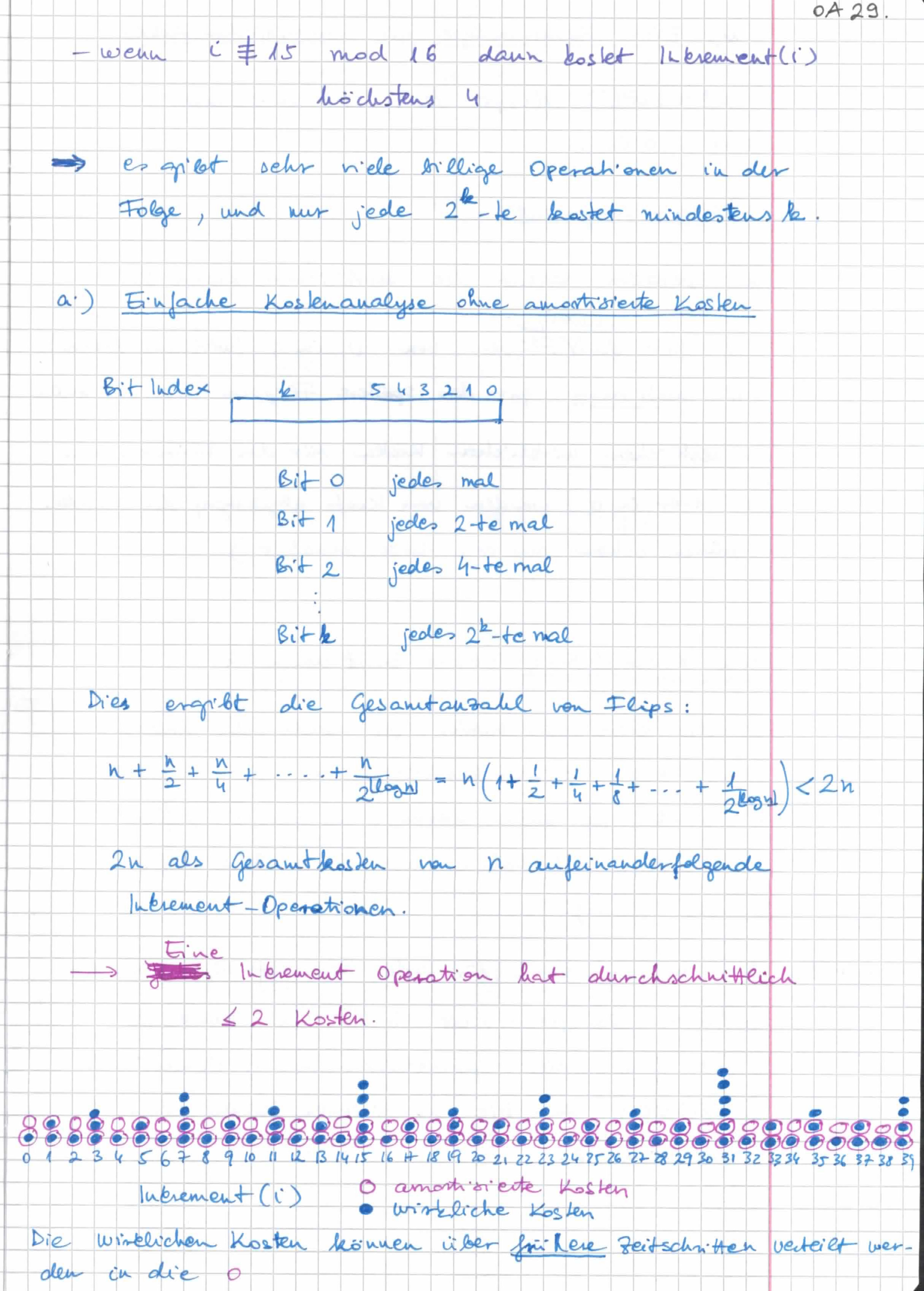
- Eine Folge von Increment Operationen werden auf einem anfänglich auf 0 gesetzten binären Zähler ausgeführt (wir addieren $+1$ jedes mal)
- die Anzahl der geflippten Bits sind die tatsächlichen Kosten eines Increments; d.h. Increment(i) kostet k wenn $k-1$ die maximale Anzahl von aufeinanderfolgenden 1-en als Suffix der Binärdarstellung von i ist.

Aufgabe:

Wir wollen die Gesamtkosten der ersten n Increment-Operationen berechnen (bzw. nach oben abschätzen).

→ Welche sind die billigsten Increment-Operationen?

- wenn i mit 0 endet, d.h. gerade ist, dann kostet Increment(i) nur 1.
- wenn $i \not\equiv 3 \mod 4$, dann kostet Increment(i) höchstens 2
- wenn $i \not\equiv 7 \mod 8$ dann kostet Increment(i) höchstens 3



– wenn $i \not\equiv 15 \mod 16$ dann kostet Increment(i) höchstens 4

⇒ es gibt sehr viele billige Operationen in der Folge, und nur jede 2^k-te kostet mindestens k.

a.) <u>Einfache Kostenanalyse ohne amortisierte Kosten</u>

Bit Index k … 5 4 3 2 1 0

Bit 0 jedes mal

Bit 1 jedes 2-te mal

Bit 2 jedes 4-te mal

⋮

Bit k jedes 2^k-te mal

Dies ergibt die Gesamtanzahl von Flips:

$$n + \frac{n}{2} + \frac{n}{4} + \dots + \frac{n}{2^{\lfloor \log n \rfloor}} = n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{2^{\lfloor \log n \rfloor}}\right) < 2n$$

$2n$ als Gesamtkosten von n aufeinanderfolgende Increment-Operationen.

→ Eine Increment Operation hat durchschnittlich ≤ 2 Kosten.

Die wirklichen Kosten können über <u>frühere</u> Zeitschritten verteilt werden in die ○

OA 30.

b) Eine Kostenanalyse mit amortisierten Kosten

Jedes Inkrement(i) flippt $k-1$ 1-er in 0 und eine 0 in 1.

I. Buchhalter-Argument

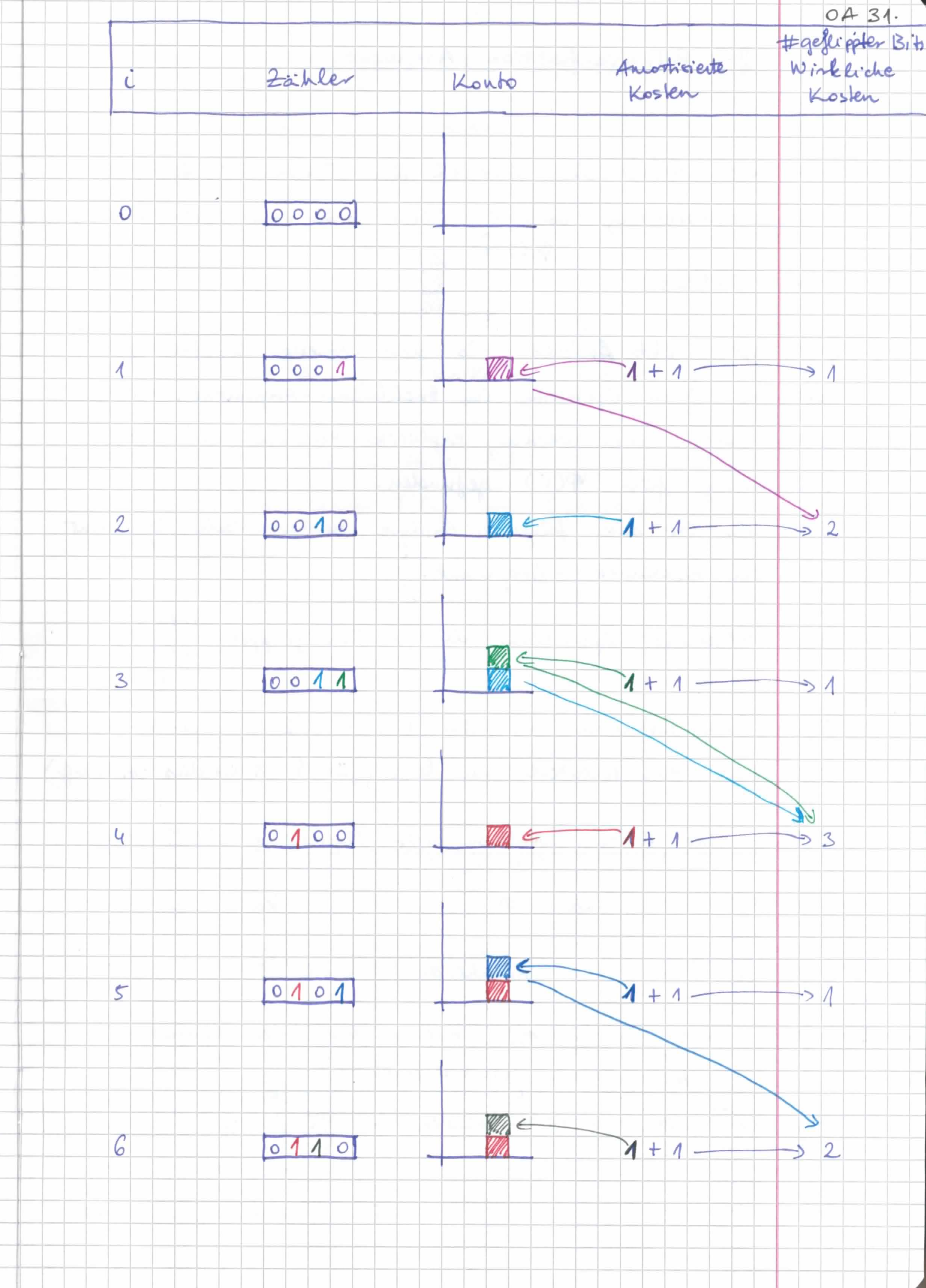
- jede Operation zahlt die amortisierten Kosten 2:
  - 1 für das (einzige) Flippen von 0 in 1 (wirkliche Kosten)
  - 1 aufs Konto für das spätere Flippen dieser 1 in 0.

(die restlichen wirklichen Kosten für das Flippen von $k-1$ 1-er in 0 werden vom Konto abgezogen; die wurden ja früher schon auf das Konto bezahlt)

Wichtige Beobachtung (siehe Bild unten):

Der aktuelle Wert des Kontos ist stets die Anzahl der 1-en in der Binärdarstellung von i (diese warten ja noch darauf, in eine 0 geflippt zu werden, und haben ihre Einzahlung dafür auf dem Konto).

→ Da die Zahlung von 2 für jede Inkrement-Operation die Gesamtkosten zu jeder Zeit deckt (das Konto ist nie leer), sind die Gesamtkosten von n Operationen stets höchstens $2n$. Die amortisierten Kosten einer Operation sind 2.



OA 31.
i
Zähler
Konto
Amortisierte Kosten
#geflippter Bits Wirkliche Kosten
0
0000
1
0001
1 + 1
1
2
0010
1 + 1
2
3
0011
1 + 1
1
4
0100
1 + 1
3
5
0101
1 + 1
1
6
0110
1 + 1
2

OA 32.

II. Potentialfunktion – Argument

Bezeichne $\Phi(i)$ die Anzahl der Einsen in der Binärdarstellung von i (bzw. nach dem i-ten Schritt des Zählers).

$\Phi(i)$

(Φ(i) ist nichts Anderes, als der aktuelle Kontostand nach der i-ten Operation im Buchhalter-Argument. Wir haben aber eine weitere, explizite Problem-abhängige Bestimmung für Φ(i) gefunden.
Ebenfalls ist die folgende Analyse mathematisch äquivalent mit dem Buchhalter-Argument.)

Eine Inkrement-Operation die k Bits flippt,

$$\underbrace{\begin{matrix}0&1&1&1&1\\1&0&0&0&0\end{matrix}}_{k}$$

ändert das Potential um $1-(k-1) = -(k-2)$, d.h. sie senkt das Potential um $k-2$ (für $k=1$ ist dies Erhöhung)

Für eine Potentialfunktion gilt allgemein: (soll gelten)

- $\Phi(0)=0$, und $\Phi(i) \geq 0$ (Konto nichtnegativ)

$\boxed{0000}$ ↓ Anzahl der Einsen ≥ 0

Definition: Die amortisierten Kosten der i-ten Operation seien (bei gegebener Potentialfunktion)

$$\overset{2}{\text{Amortisiertekosten}_i} = \overset{k}{\text{Wirklichekosten}_i} + \underbrace{\overset{-(k-2)}{\Phi(i) - \Phi(i-1)}}_{\text{Änderung des Potentials (Ein/Auszahlung vom Konto)}}$$

<u>Theorem:</u> Angenommen, dass die folgenden gelten:

→ $\text{Amortisierte Kosten}_i = \text{Wirkliche Kosten}_i + \Phi(i) - \Phi(i-1) \quad (\forall i)$

→ $\Phi(0) = 0$

→ $\Phi(i) \geq 0 \quad (\forall i)$

dann sind die Gesamt-Amortisierte-Kosten mindestens so groß wie die wirklichen Gesamtkosten nach <u>jeder</u> Operation n.

⟹ Finden wir eine gute Potentialfunktion für die Kostenanalyse einer Folge von Operationen, dann sind die Gesamt-**Amortisierten**-Kosten eine gute obere Schranke für die wirklichen Kosten nach jeder Operation.

Der <u>Beweis</u> vom Theorem ist trivial:

$$\sum_{i=1}^{n} \text{Amortisierte-Kosten}_i = \sum_{i=1}^{n} \left(\text{Wirkliche-Kosten}_i + \Phi(i) - \Phi(i-1) \right) =$$

$$= \left(\sum_{i=1}^{n} \text{Wirkliche-Kosten}_i \right) + \Phi(n) - \Phi(n-1) + \Phi(n-1) - \Phi(n-2) + \Phi(n-2) -$$

$$\ldots\ldots + \Phi(2) - \Phi(1) + \Phi(1) - \underbrace{\Phi(0)}_{=0} = \left(\sum_{i=1}^{n} \text{Wirkliche-Kosten}_i \right) + \underbrace{\Phi(n)}_{\geq 0} \geq$$

$$\geq \sum_{i=1}^{n} \text{Wirkliche-Kosten}_i$$

□

Beispiel: Dynamische Hashtabelle

Wir betrachten Hashing mit Verkettung oder offener Adressierung

Wie große Hashtabelle braucht man?

- Wenn die Hashtabelle viel zu groß ist
  → braucht sie unnötig Platz
- Wenn sie zu klein ist
  → es gibt zu viele Kollisionen, und die Operationen nehmen zu viel Zeit

Sei $\lambda = \frac{\text{Anzahl gespeicherter Schlüssel}}{\text{Tabellengröße}}$ der <u>Auslastungsfaktor</u>

Wir werden die Tabellengröße dynamisch anpassen:

→ fange mit einer Tabelle der Größe 1 an; anfänglich ist die Tabelle leer

→ wenn $\lambda \geq 1$ (zu viele Schlüssel), dann füge alle Schlüssel in eine Tabelle doppelter Größe ein

→ wenn $\lambda \leq \frac{1}{4}$ (zu wenige Schlüssel), füge alle Schlüssel in eine neue Tabelle der halben Größe ein.

- die Anpassung / Reorganisation der Hash-Tabelle braucht viel Zeit (sagen wir, proportional zur Tabellengröße)

- hoffentlich wird aber eine Reorganisation relativ selten durchgeführt; insbesondere können zwei (nah) aufeinanderfolgende Operationen nicht beide zur Reorganisation führen, weil:

Wie groß ist der Auslastungsfaktor nach einer Reorganisation?

→ bei Vergrößerung der Tabelle:

$\lambda_{neu} = \frac{1}{2}$

$\lambda = 1$

→ bei Verkleinerung der Tabelle:

$\lambda_{neu} = \frac{1}{2}$

$\lambda = \frac{1}{4}$

⇒ in beiden Fällen wird $\lambda_{neu} = \frac{1}{2}$

→ Bezeichne m die aktuelle Tabellengröße. m ist somit eine obere Schranke für die Anzahl der aktuell gespeicherten Schlüssel (weil $\lambda \leq 1$ gilt nach unseren Regeln)

→ die Operationen sind die üblichen Hash-Operationen:

Insert(x), Lookup(x), Delete(x)

→ sei $\sigma = \sigma_1 \sigma_2 \sigma_3 \dots \sigma_i \dots$ eine Folge solcher Operationen

Angenommen:

→ Wirkliche Kosten$_i$:= $\begin{cases} 1 & \text{wenn keine Reorganisation erforderlich} \\ m & \text{wenn Reorganisation gemacht wird} \\ & \text{(m ist die \underline{alte} Tabellengröße)} \end{cases}$

→ große Kosten werden nur durch Reorganisation verursacht

Amortisierte Laufzeitanalyse :

sei: m die Tabellengröße

→ nach der letzten Reorganisation war $\lambda = \frac{1}{2}$
also die Tabelle war halb voll und hat $\approx \frac{m}{2}$ Schlüssel gespeichert

→ bei einer Reorganisation

FALL 1 : m ist zu klein geworden, weil $\lambda \geq 1$
dann wurden seit der letzten Reorganisation mindestens $\frac{m}{2}$ (Insert-) Operationen ausgeführt

⇒ wenn jede Operation neben den wirklichen Kosten von 1, zusätzlich 2 auf das Konto legt, dann wären im FALL 1 die Kosten einer Reorganisation gedeckt $(2 \cdot \frac{m}{2} = m)$
→ dies würde Amortisierte Kosten$_i$ = 3 implizieren
ABER wir haben auch einen anderen Fall

FALL 2 : m ist zu groß geworden weil $\lambda \leq \frac{1}{4}$
dann wurden seit der letzten Reorganisation mindestens $\frac{m}{4}$ (Delete (x)) Operationen ausgeführt

→ Wenn jede Operation zusätzlich zu seiner Kosten von 1 noch 4 auf das Konto zahlt, dann werden im FALL 2 und auch im FALL 1 die Kosten einer Reorganisation vom Konto gedeckt. Dies impliziert Amortisierte Kosten von 5 pro Operation (wenn wir mit einer leeren Hashtabelle anfangen).

<u>Bemerkungen</u>: 1. Beim binären Zähler war 2 die genaue (fast) Durchschnittskosten. Dieses Beispiel zeigt, dass die amortisierten Kosten allgemein nur eine obere Schranke für die Durchschnittskosten sind

2. Wir hätten unterschiedliche amortisierte Kosten für Insert (3) und Delete (5) bestimmen können (Durchschnittswert nur bzgl. Insert (bzw. Delete) Operationen). Bei asymptotisch gleichen Kosten lohnt es sich nicht.

3. Wir haben hier keine schöne kompakte Definition für eine Potentialfunktion (Kontostand) $\Phi(i)$.

Beispiel: Binomische Heaps und Fibonacci Heaps

(S.a. Cormen - Leiserson - Rivest - Stein: Introduction to Algorithms)

<u>Zur Erinnerung</u>: Was für Datenstrukturen sind Heaps? (Min-Heap)

(Offiziell: eine <u>Implementierung</u> des Datentyps: <u>Prioritätswarteschlange</u> für die Unterstützung von Insert(x), DeleteMin, ChangePriority (wo), Delete(wo))

→ die Datenstruktur <u>Heap</u> unterstützt die Operationen

Insert(x) → $O(\log m)$ ← Delete(wo)

DeleteMin → $O(\log m)$
↳ entferne den minimalen Schlüssel

ChangePriority (wo, Δ) → $O(\log m)$
(DecreaseKey) → reduziere den Schlüssel mit Adresse „wo" um Δ

wobei m ist die Anzahl der gespeicherten Schlüssel

OA 38.

→ Es fällt uns ein Binärbaum ein

Array Reihenfolge

i

2i

2i+1

- ein Heap „ist" ein Binärbaum, s.d. jeder Knoten einen Schlüssel enthält

- der Baum hat <u>Heap-Struktur</u>
  die Knoten füllen von links nach rechts alle Ebenen
  die Blatt-Ebene kann unvollständig sein

- die Schlüssel (die Prioritäten) haben <u>Heap-Ordnung</u>:

x

z y

$x \leq y$ und $x \leq z$

→ eigentlich ist ein Heap ein Array

i 2i 2i+1

die Schlüssel werden in der obigen Reihenfolge gespeichert (i hat Kinder 2i und 2i+1 als Array-Indizes)

→ Was können Heaps nicht? Lookup(x)
  insbesondere sind sie <u>keine Suchbäume</u>
  (deshalb soll Delete(wo) und ChangePriority(wo) die Position wo übergeben werden)

→ Laufzeit = O (Tiefe des Baumes) = $O(\log m)$

Unser Ziel: Wir möchten eine Datenstruktur entwickeln, die dieselben Operationen unterstützt, aber mit viel besseren (amortisierten) Laufzeiten als Heaps ....

Wofür braucht man eine solche Datenstruktur?

zum Beispiel: (siehe später)

→ für die schnelle Implementierung von <u>Dijkstra's Algorithmus</u> für kürzeste Wege in $G(V,E)$

→ für die schnelle Implementierung von <u>Prim's Algorithmus</u> für minimale Spannbäume

Eine schnelle Implementierung dieser Algorithmen gelingt, falls die folgenden amortisierten Laufzeiten erreicht werden (für maximal m Schlüssel): $(m = |V|)$

Insert(x), DecreaseKey(wo, Δ) : $O(1)$

DeleteMin, Delete(wo) : $O(\log_2 m)$

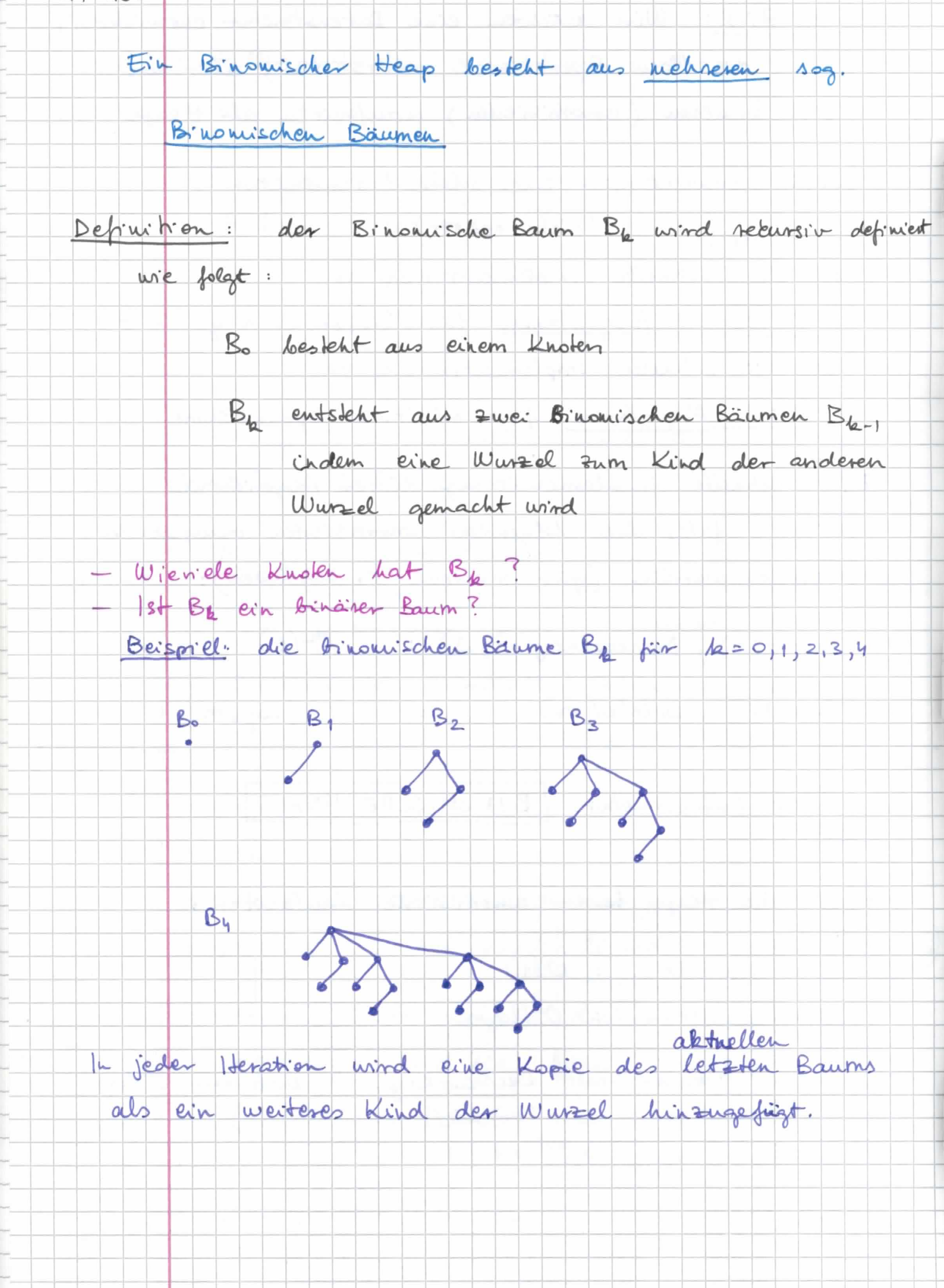
Die erste Datenstruktur: **BINOMISCHE HEAPS**

Binomische Heaps haben amortisierte Laufzeiten:

Insert(x) : $O(1)$

Delete(wo), DeleteMin : $O(\log m)$

Aber sie scheitern mit DecreaseKey! → auch nur $O(\log m)$



Ein Binomischer Heap besteht aus <u>mehreren</u> sog. <u>Binomischen Bäumen</u>

<u>Definition</u>: der Binomische Baum B_k wird rekursiv definiert wie folgt:

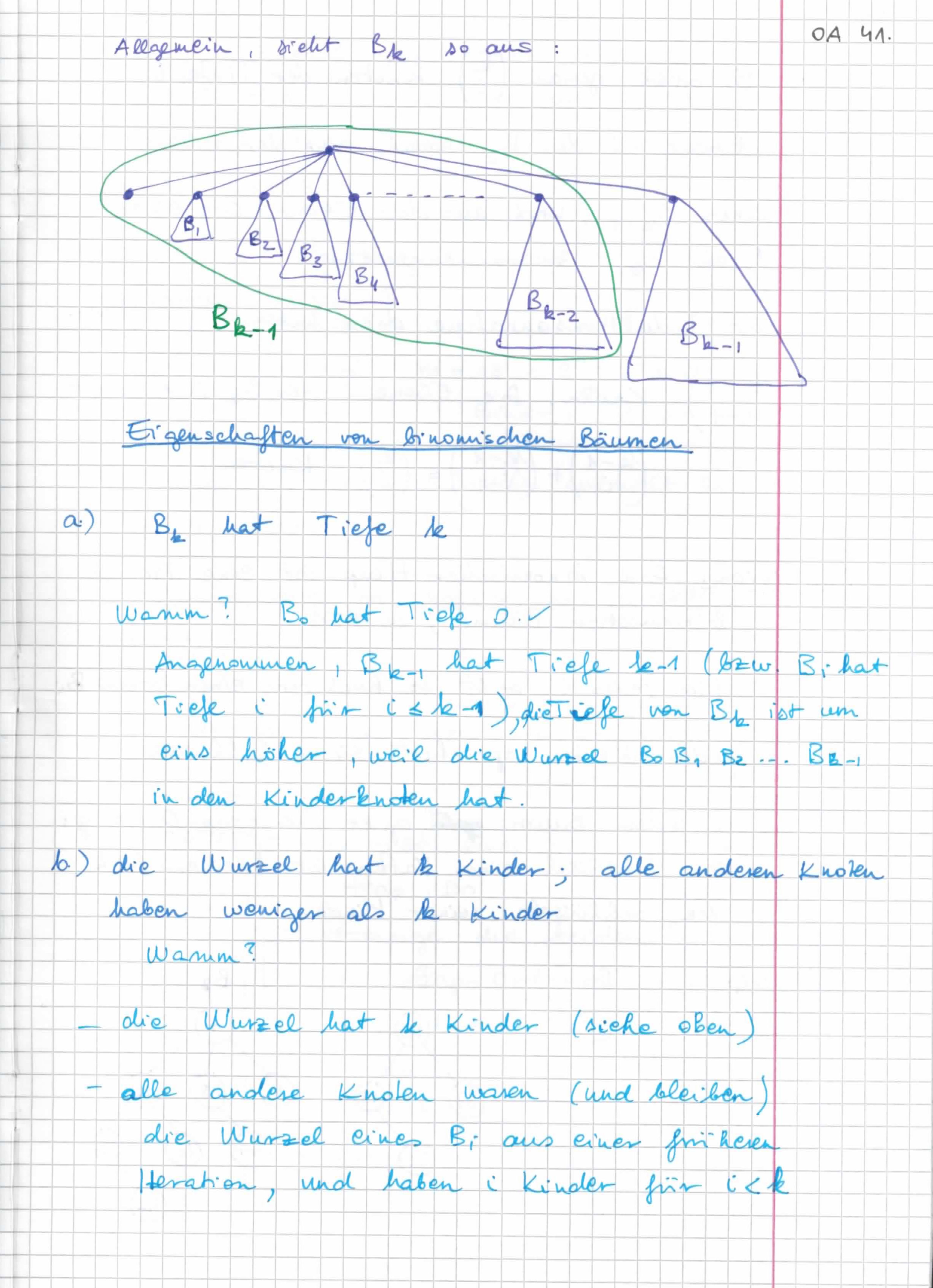
B_0 besteht aus einem Knoten

B_k entsteht aus zwei Binomischen Bäumen B_{k-1} indem eine Wurzel zum Kind der anderen Wurzel gemacht wird

- Wieviele Knoten hat B_k?
- Ist B_k ein binärer Baum?

<u>Beispiel</u>: die Binomischen Bäume B_k für $k = 0, 1, 2, 3, 4$

In jeder Iteration wird eine Kopie des ~~letzten~~ aktuellen Baums als ein weiteres Kind der Wurzel hinzugefügt.



Allgemein, sieht B_k so aus:

Eigenschaften von binomischen Bäumen

a) B_k hat Tiefe k

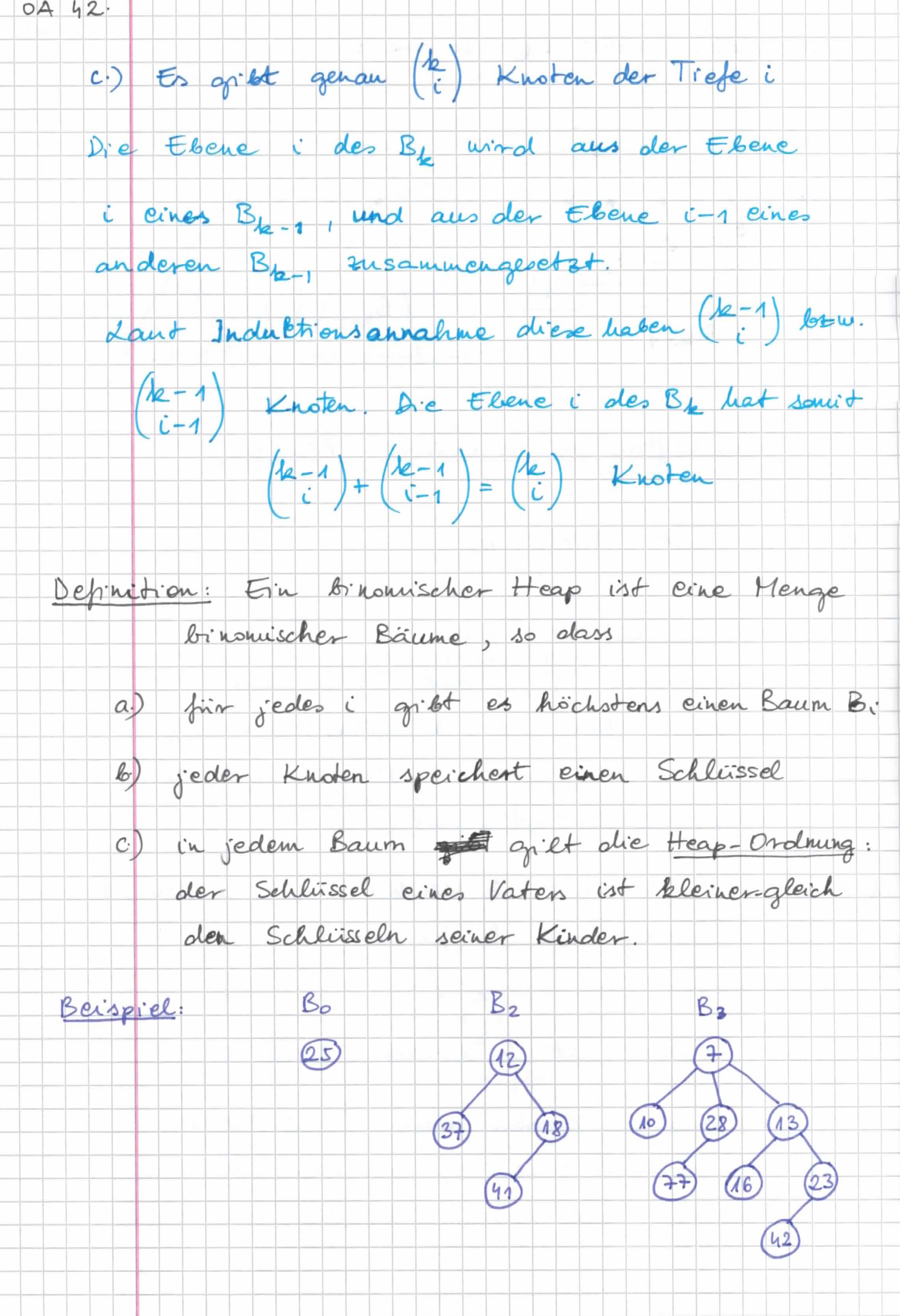
Warum? B_0 hat Tiefe 0. ✓

Angenommen, B_{k-1} hat Tiefe $k-1$ (bzw. B_i hat Tiefe i für $i \le k-1$), die Tiefe von B_k ist um eins höher, weil die Wurzel B_0 B_1 B_2 ... B_{k-1} in den Kinderknoten hat.

b) die Wurzel hat k Kinder; alle anderen Knoten haben weniger als k Kinder

Warum?

- die Wurzel hat k Kinder (siehe oben)
- alle andere Knoten waren (und bleiben) die Wurzel eines B_i aus einer früheren Iteration, und haben i Kinder für $i < k$



c.) Es gibt genau $\binom{k}{i}$ Knoten der Tiefe i

Die Ebene i des B_k wird aus der Ebene i eines B_{k-1}, und aus der Ebene $i-1$ eines anderen B_{k-1} zusammengesetzt.

Laut Induktionsannahme diese haben $\binom{k-1}{i}$ bzw. $\binom{k-1}{i-1}$ Knoten. Die Ebene i des B_k hat somit

$$\binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i} \text{ Knoten}$$

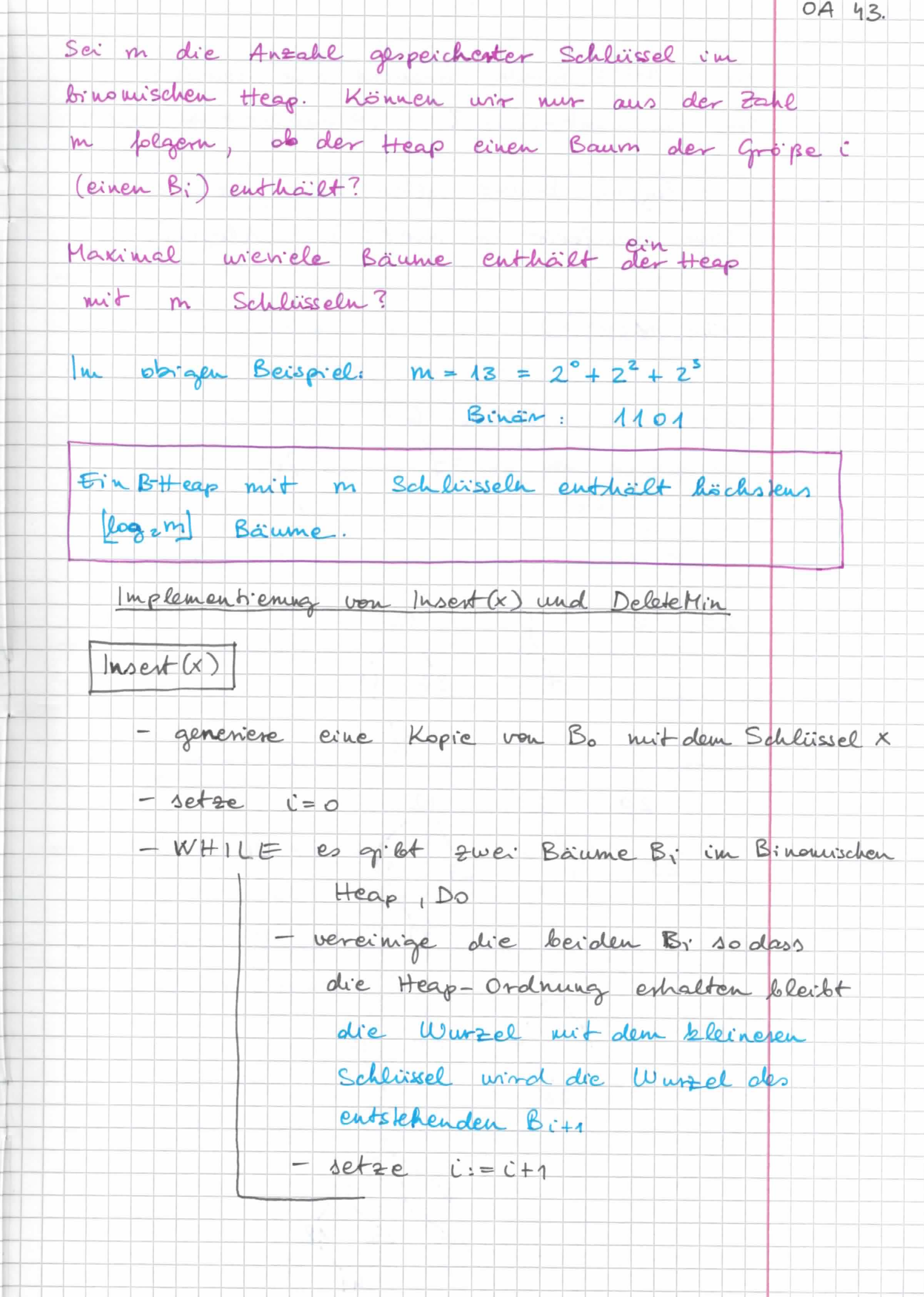
<u>Definition:</u> Ein binomischer Heap ist eine Menge binomischer Bäume, so dass

a) für jedes i gibt es höchstens einen Baum B_i

b) jeder Knoten speichert einen Schlüssel

c) in jedem Baum gilt die <u>Heap-Ordnung</u>: der Schlüssel eines Vaters ist kleiner-gleich den Schlüsseln seiner Kinder.

<u>Beispiel:</u>



Sei m die Anzahl gespeicherter Schlüssel im binomischen Heap. Können wir nur aus der Zahl m folgern, ob der Heap einen Baum der Größe i (einen B_i) enthält?

Maximal wieviele Bäume enthält ~~der~~ ein Heap mit m Schlüsseln?

Im obigen Beispiel: $m = 13 = 2^0 + 2^2 + 2^3$

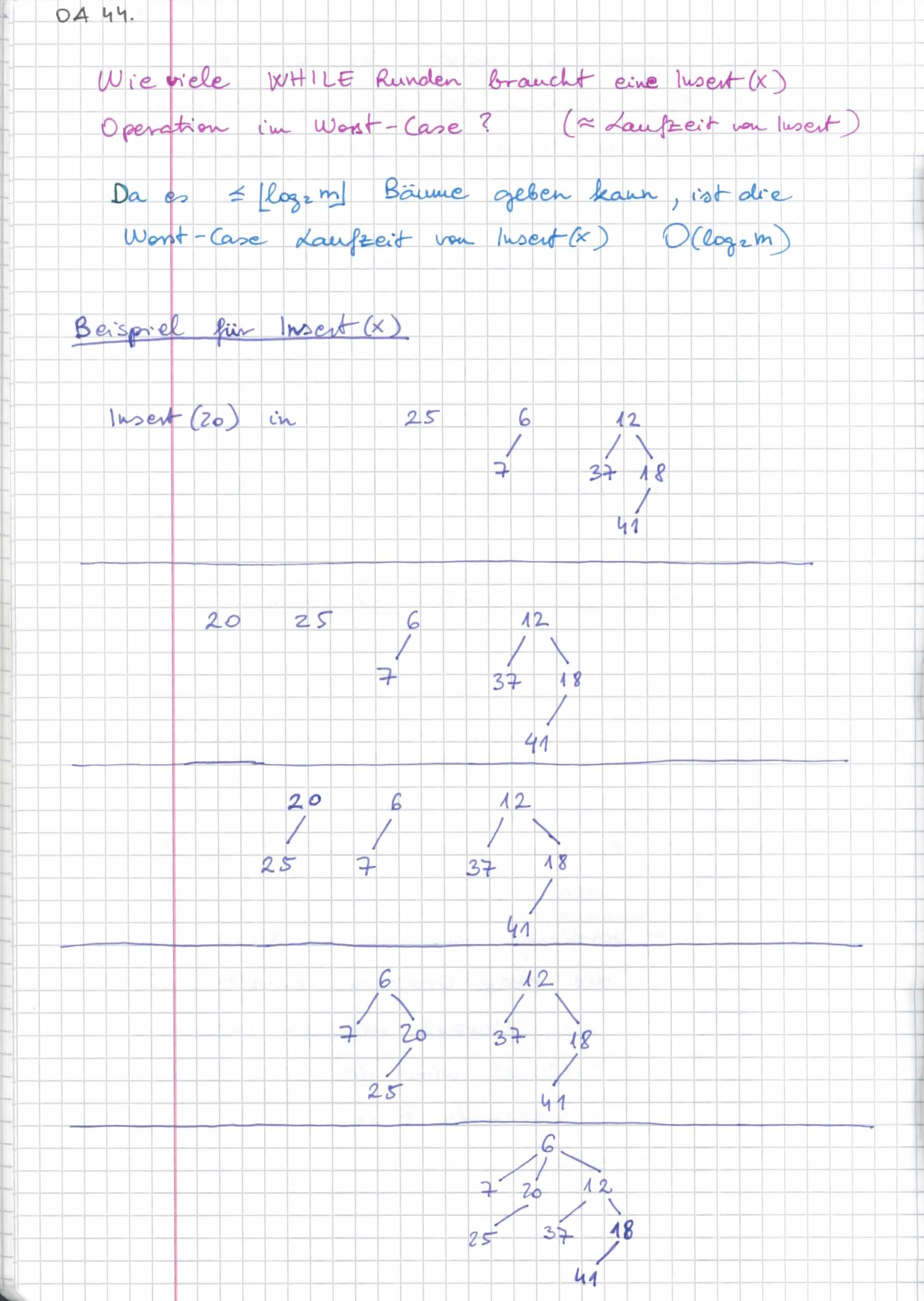
Binär: 1101

> Ein B-Heap mit m Schlüsseln enthält höchstens $\lceil \log_2 m \rceil$ Bäume.

<u>Implementierung von Insert(x) und DeleteMin</u>

Insert(x)

- generiere eine Kopie von B_0 mit dem Schlüssel x
- setze $i = 0$
- WHILE es gibt zwei Bäume B_i im Binomischen Heap, Do
  - vereinige die beiden B_i so dass die Heap-Ordnung erhalten bleibt
    die Wurzel mit dem kleineren Schlüssel wird die Wurzel des entstehenden B_{i+1}
  - setze $i := i+1$



OA 44.

Wie viele WHILE Runden braucht eine Insert(x) Operation im Worst-Case? (≈ Laufzeit von Insert)

Da es $\leq \lfloor \log_2 m \rfloor$ Bäume geben kann, ist die Worst-Case Laufzeit von Insert(x) $O(\log_2 m)$

Beispiel für Insert(x)

Insert (20) in

Erste amortisierte Analyse von Insert:

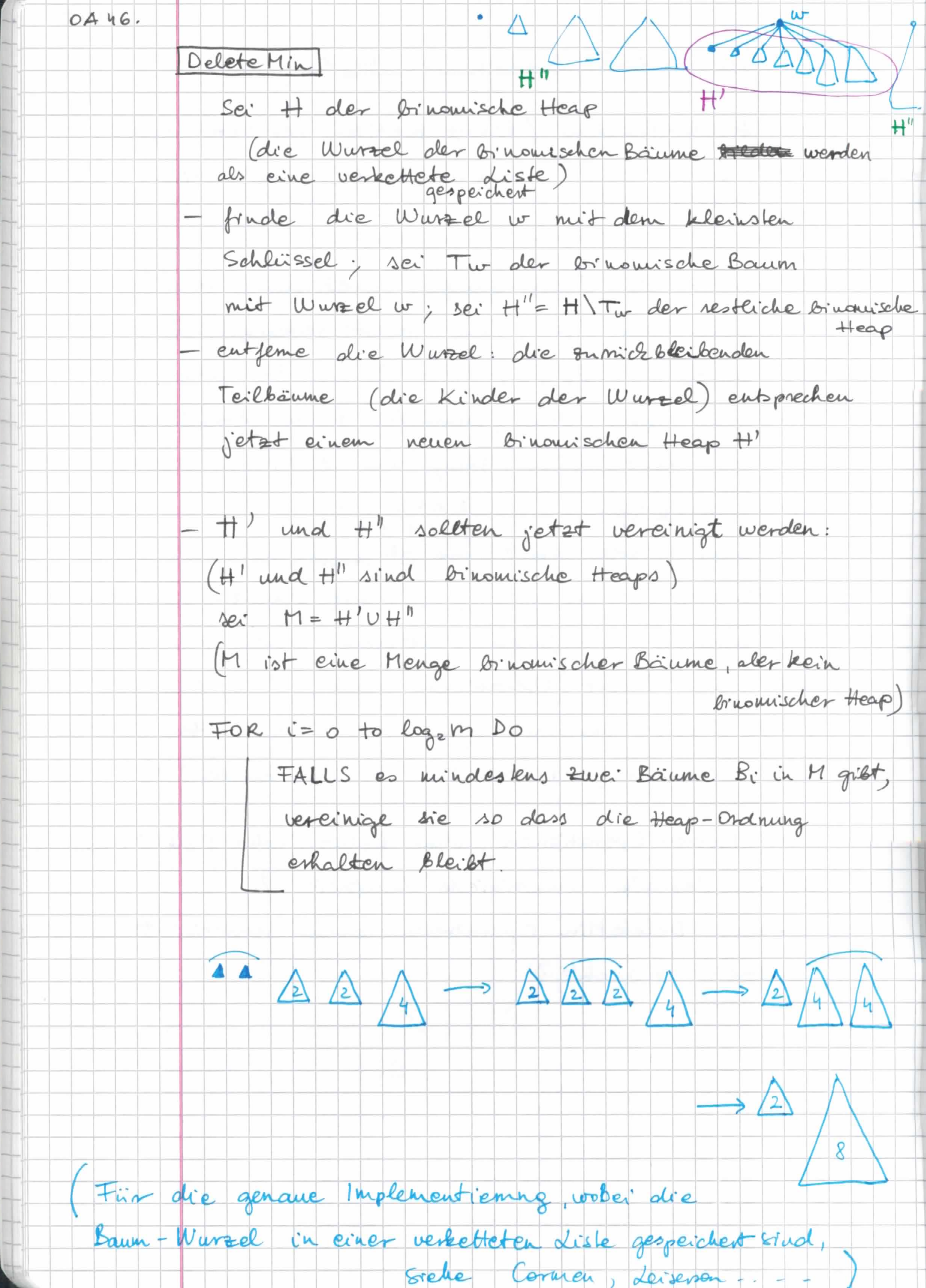
Viele WHILE-Runden (lange Ketten der Vereinigung) treten selten auf! (wenn nur Insert() gemacht wird)
Warum?

- Es werden k Vereinigungen stattfinden, wenn der binomische Heap vor dem Insert jeweils einen
  $B_0 \ B_1 \ B_2 \ldots B_{k-1}$, aber keinen B_k hatte.
- Dies gilt genau dann, wenn die Binärdarstellung von m mit einer Folge von k Einsen endet, aber $Bit_k = 0$.

k	$k-1$	...	3	2	1	0
0	1	1 1 1	1	1	1	1

  → dies würde 1mal pro 2^k Insert Operationen vorkommen, wenn wir nur Insert Operationen hätten.
- Angenommen, wir fangen mit einem leeren Heap an, und nur Insert Operationen haben, dann hat die m-te Insert-Operation genau dieselben Kosten, wie Increment $(m-1)$ bei dem binären Zähler!

⟹ NUR Insert-Operationen, ohne Delete oder DeleteMin, hätten deshalb amortisierte Laufzeit 2.
Machen DeleteMin Operationen diese amortisierte Laufzeit zunichte?



OA 46.

Delete Min

Sei H der binomische Heap

(die Wurzel der binomischen Bäume ~~[illegible]~~ werden als eine verkettete Liste gespeichert)

- finde die Wurzel w mit dem kleinsten Schlüssel; sei T_w der binomische Baum mit Wurzel w; sei $H'' = H \setminus T_w$ der restliche binomische Heap
- entferne die Wurzel: die zurückbleibenden Teilbäume (die Kinder der Wurzel) entsprechen jetzt einem neuen binomischen Heap H'
- H' und H'' sollten jetzt vereinigt werden:

(H' und H'' sind binomische Heaps)

sei $M = H' \cup H''$

(M ist eine Menge binomischer Bäume, aber kein binomischer Heap)

FOR $i = 0$ to $\log_2 M$ DO

  FALLS es mindestens zwei Bäume B_i in M gibt, vereinige sie so dass die Heap-Ordnung erhalten bleibt.

(Für die genaue Implementierung, wobei die Baum-Wurzel in einer verketteten Liste gespeichert sind, siehe Cormen, Leiserson . . . .)

Was ist die Laufzeit von DeleteMin?

<u>Beobachtung 1:</u> ~~Behauptung 1~~: Die Vereinigung von H' und H'' enthält nach <u>jeder</u> Runde höchstens drei Bäume B_i derselben Größe: ≤ 1 B_i aus H', ≤ 1 B_i aus H'', plus ≤ 1 B_i wenn eben zwei B_{i-1} vereinigt wurden in der vorherigen Runde. ("Übertrag")

$\Rightarrow$ eine Baum-Vereinigung reicht für jede Größe

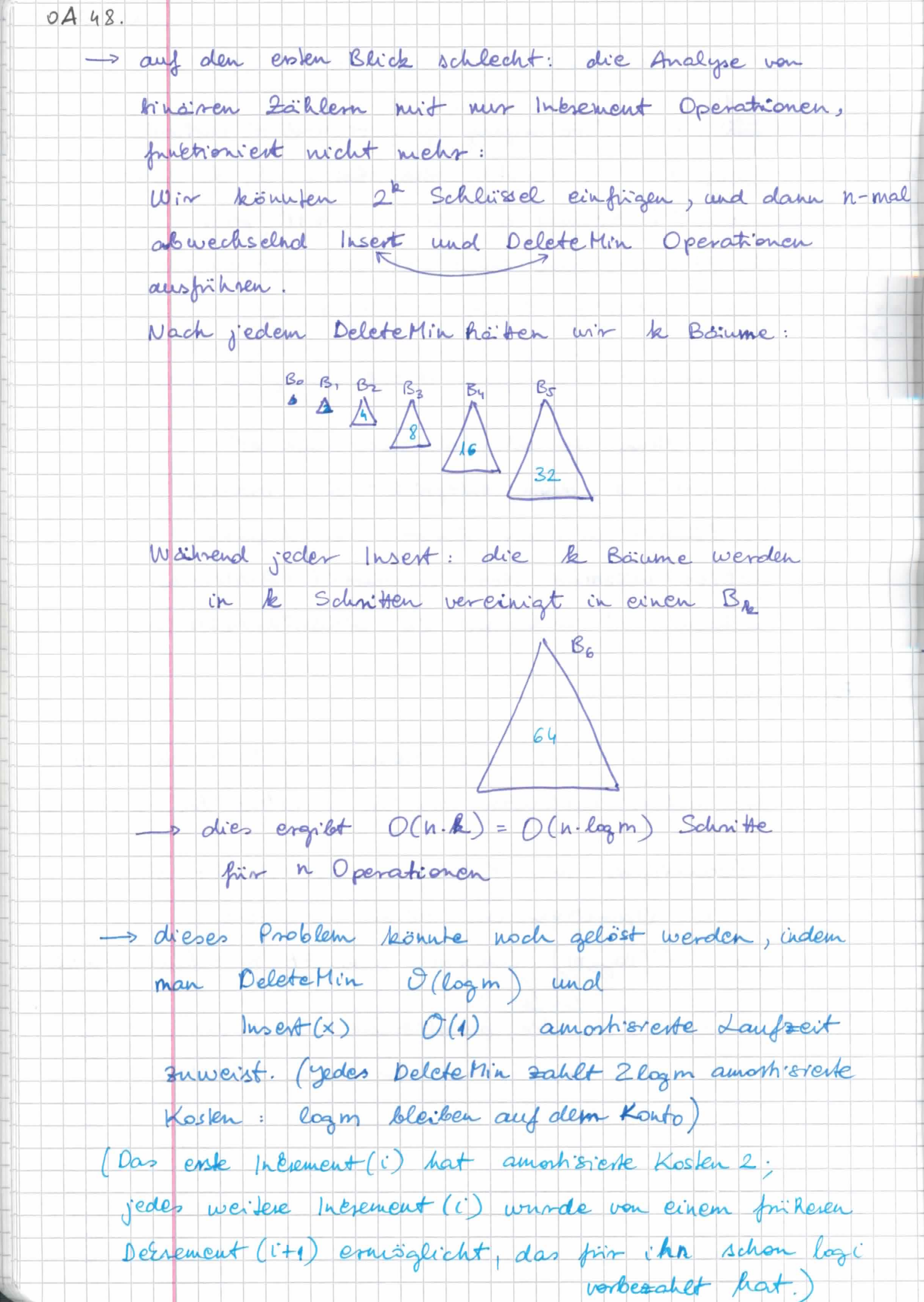
<u>Beobachtung 2:</u> Wenn H' m' Schlüssel und H'' m'' Schlüssel enthält, dann entspricht die Anzahl der B_i Bäume in jeder Runde der Anzahl der 1-Bits während des Grundschul-Addierens der Zahlen m' und m'' in Binärdarstellung.

<u>Es folgt:</u>

Sei $m = m' + m'' + 1$ die Anzahl der Schlüssel in H
$\rightarrow$ H' und H'' enthalten jeweils höchstens $\log_2 m$ Bäume.

Die Vereinigung von H' und H'' und damit die DeleteMin Operation im binomischen Heap dauert $O(\log_2 m)$ Schritte wenn H m Schlüssel speichert.

Wie ändert sich die amortisierte Laufzeit $2 = O(1)$ von Insert(x), wenn wir auch DeleteMin Operationen erlauben?



0A 48.

→ auf den ersten Blick schlecht: die Analyse von binären Zählern mit nur Increment Operationen, funktioniert nicht mehr:

Wir könnten 2^k Schlüssel einfügen, und dann n-mal abwechselnd Insert und DeleteMin Operationen ausführen.

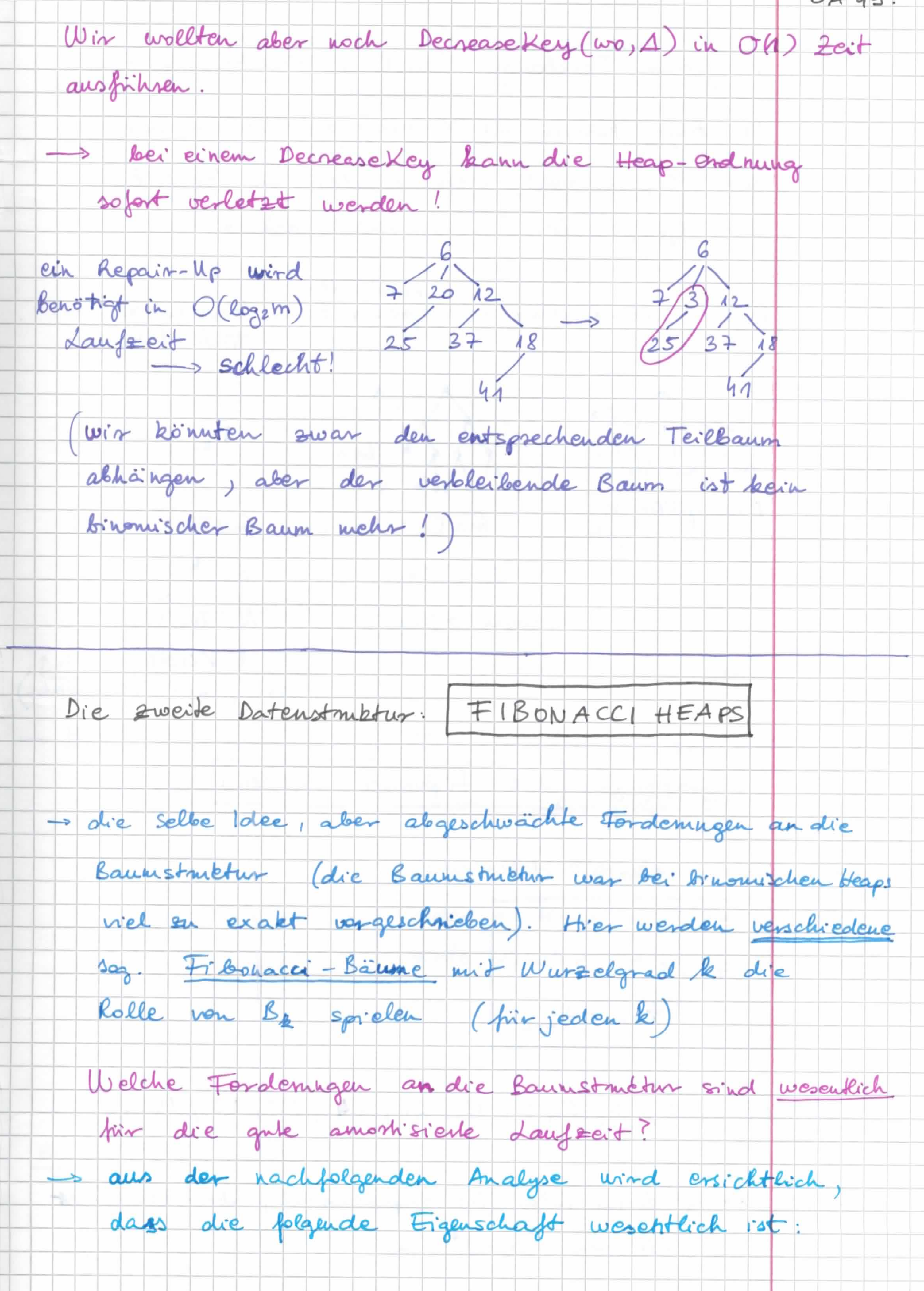
Nach jedem DeleteMin hätten wir k Bäume:

Während jeder Insert: die k Bäume werden in k Schritten vereinigt in einen B_k

→ dies ergibt $O(n \cdot k) = O(n \cdot \log m)$ Schritte für n Operationen

→ dieses Problem könnte noch gelöst werden, indem man DeleteMin $O(\log m)$ und Insert(x) $O(1)$ amortisierte Laufzeit zuweist. (Jedes DeleteMin zahlt $2 \log m$ amortisierte Kosten: $\log m$ bleiben auf dem Konto)

(Das erste Increment(i) hat amortisierte Kosten 2; jedes weitere Increment(i) wurde von einem früheren Decrement(i+1) ermöglicht, das für ihn schon $\log i$ vorbezahlt hat.)



Wir wollten aber noch DecreaseKey(wo, Δ) in O(1) Zeit ausführen.

→ bei einem DecreaseKey kann die Heap-Ordnung sofort verletzt werden!

ein Repair-Up wird benötigt in $O(\log_2 m)$ Laufzeit
→ schlecht!

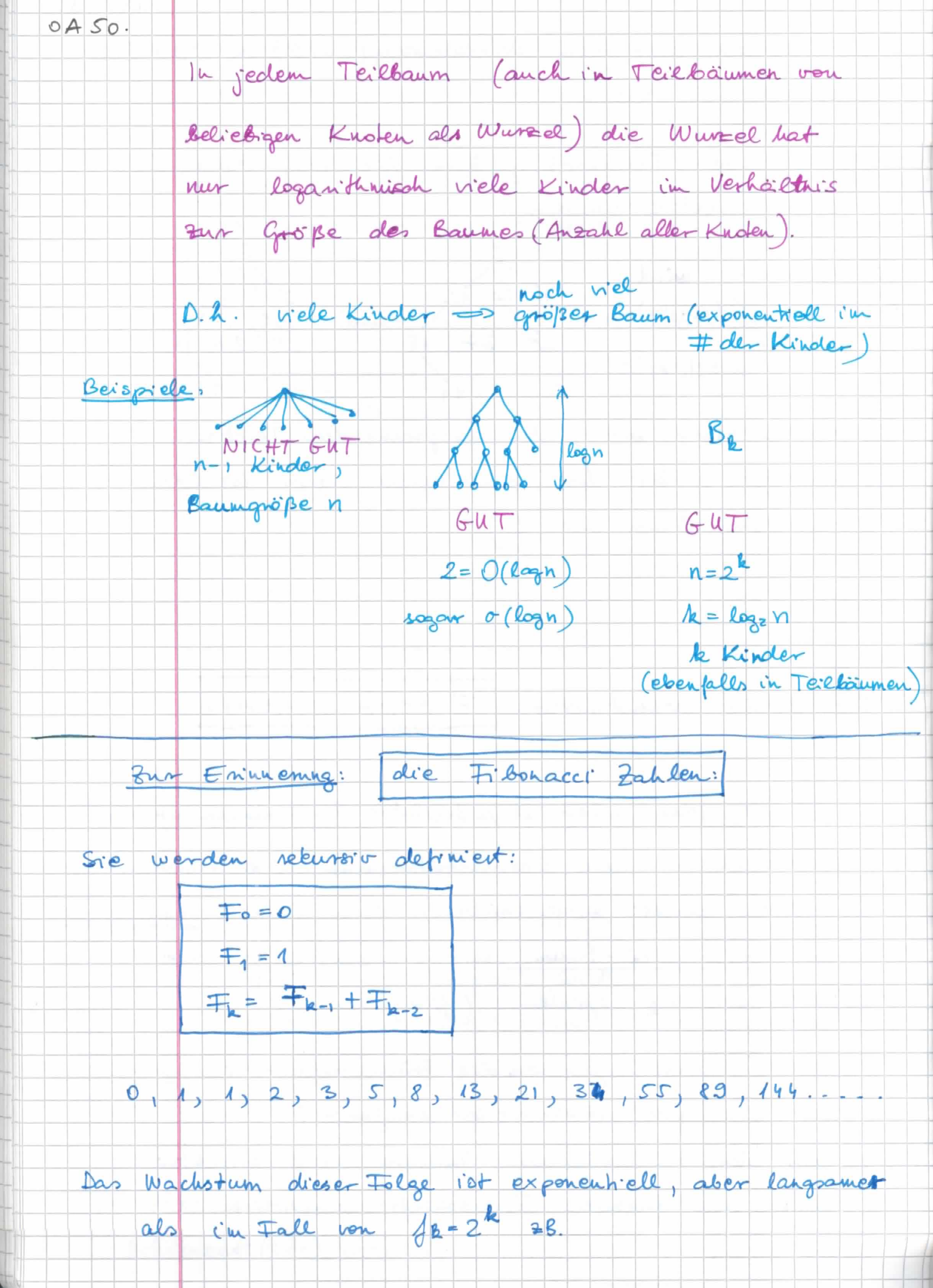
(wir könnten zwar den entsprechenden Teilbaum abhängen, aber der verbleibende Baum ist kein binomischer Baum mehr!)

Die zweite Datenstruktur: **FIBONACCI HEAPS**

→ die selbe Idee, aber abgeschwächte Forderungen an die Baumstruktur (die Baumstruktur war bei binomischen Heaps viel zu exakt vorgeschrieben). Hier werden <u>verschiedene</u> sog. <u>Fibonacci-Bäume</u> mit Wurzelgrad k die Rolle von B_k spielen (für jeden k)

Welche Forderungen an die Baumstruktur sind <u>wesentlich</u> für die gute amortisierte Laufzeit?

→ aus der nachfolgenden Analyse wird ersichtlich, dass die folgende Eigenschaft wesentlich ist:



OA 50.

In jedem Teilbaum (auch in Teilbäumen von beliebigen Knoten als Wurzel) die Wurzel hat nur logarithmisch viele Kinder im Verhältnis zur Größe des Baumes (Anzahl aller Knoten).

D.h. viele Kinder $\Rightarrow$ noch viel größer Baum (exponentiell im # der Kinder)

Beispiele:

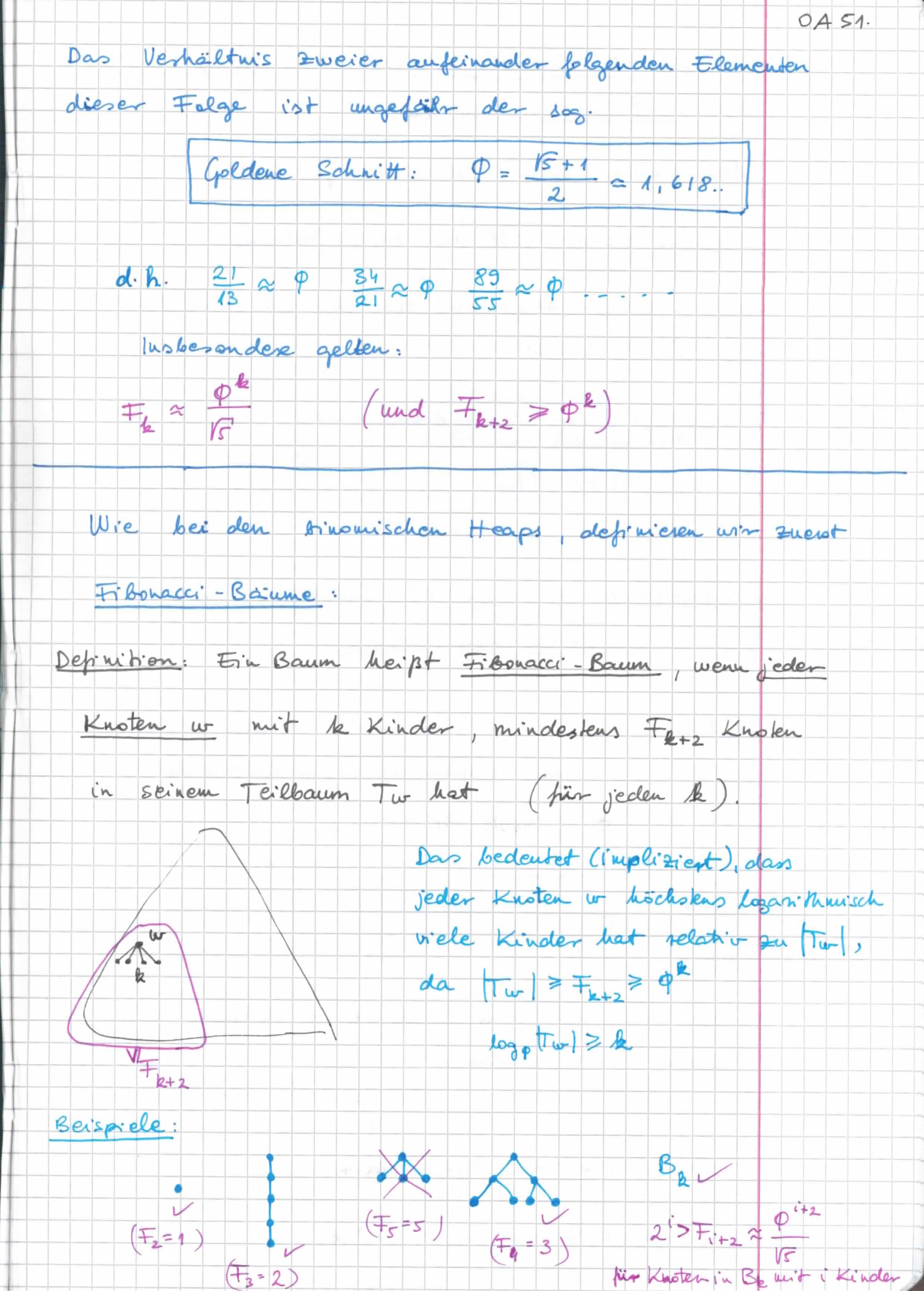
Zur Erinnerung: die Fibonacci Zahlen:

Sie werden rekursiv definiert:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_k = F_{k-1} + F_{k-2}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ....

Das Wachstum dieser Folge ist exponentiell, aber langsamer als im Fall von $f_k = 2^k$ z.B.



Das Verhältnis zweier aufeinander folgenden Elementen dieser Folge ist ungefähr der sog.

Goldene Schnitt: $\phi = \frac{\sqrt{5}+1}{2} \approx 1{,}618..$

d.h. $\frac{21}{13} \approx \phi \quad \frac{34}{21} \approx \phi \quad \frac{89}{55} \approx \phi \quad \ldots$

Insbesondere gelten:

$F_k \approx \frac{\phi^k}{\sqrt{5}}$ $\quad$ (und $F_{k+2} \geq \phi^k$)

Wie bei den binomischen Heaps, definieren wir zuerst

<u>Fibonacci-Bäume</u>:

<u>Definition</u>: Ein Baum heißt <u>Fibonacci-Baum</u>, wenn <u>jeder Knoten w</u> mit k Kinder, mindestens F_{k+2} Knoten in seinem Teilbaum T_w hat (für jeden k).

Das bedeutet (impliziert), dass jeder Knoten w höchstens logarithmisch viele Kinder hat relativ zu $|T_w|$, da $|T_w| \geq F_{k+2} \geq \phi^k$

$\log_\phi |T_w| \geq k$

<u>Beispiele</u>:

$2^i > F_{i+2} \approx \frac{\phi^{i+2}}{\sqrt{5}}$

für Knoten in B_k mit i Kinder

Definition: Ein Fibonacci-Heap ist eine Menge von Fibonacci-Bäumen, so dass

- jeder Knoten einen Schlüssel speichert
- in jedem Baum gilt die Heap-Ordnung
- ein Min Zeiger zeigt auf die Wurzel mit dem kleinsten Schlüssel

Beispiel: Sogar das ist ein Fibonacci-Heap:

(25) (37) (12) (41) (18) (3) (6)

Implementierung von Insert(x), DeleteMin, und DecreaseKey(w, Δ)

Wir möchten, dass immer nur DeleteMin die Bäume vereinigt. Die anderen Operationen zerstückeln den Heap indem sie neue Bäume produzieren.

⇒ er kann beliebig viele Bäume (sogar n) auf einmal enthalten!

Insert(x)

- speichere den neuen Schlüssel x in einem neuen Baum mit 1 Knoten

(x) w

- aktualisiere den Min Zeiger

---

die wirkliche Laufzeit von Insert: $O(1)$

- wir überprüfen noch die Fibonacci-Eigenschaft für den neuen Baum : $F_{0+2} = F_2 = 1 \leq |T_w|$ ✓

<u>DeleteMin</u> (wie bereits angekündigt, wird jetzt aufgeräumt)

- entferne das Minimum (es wird mit dem Min Zeiger gefunden)
  es entstehen k neue Bäume wenn das Minimum k Kinder hatte
- WHILE es gibt mind. 2 Bäume mit gleichem Wurzelgrad,
  - vereinige die 2 Bäume; behalte dabei die Heap-Ordnung
- aktualisiere den Min Zeiger

Die wirkliche Laufzeit von DeleteMin:

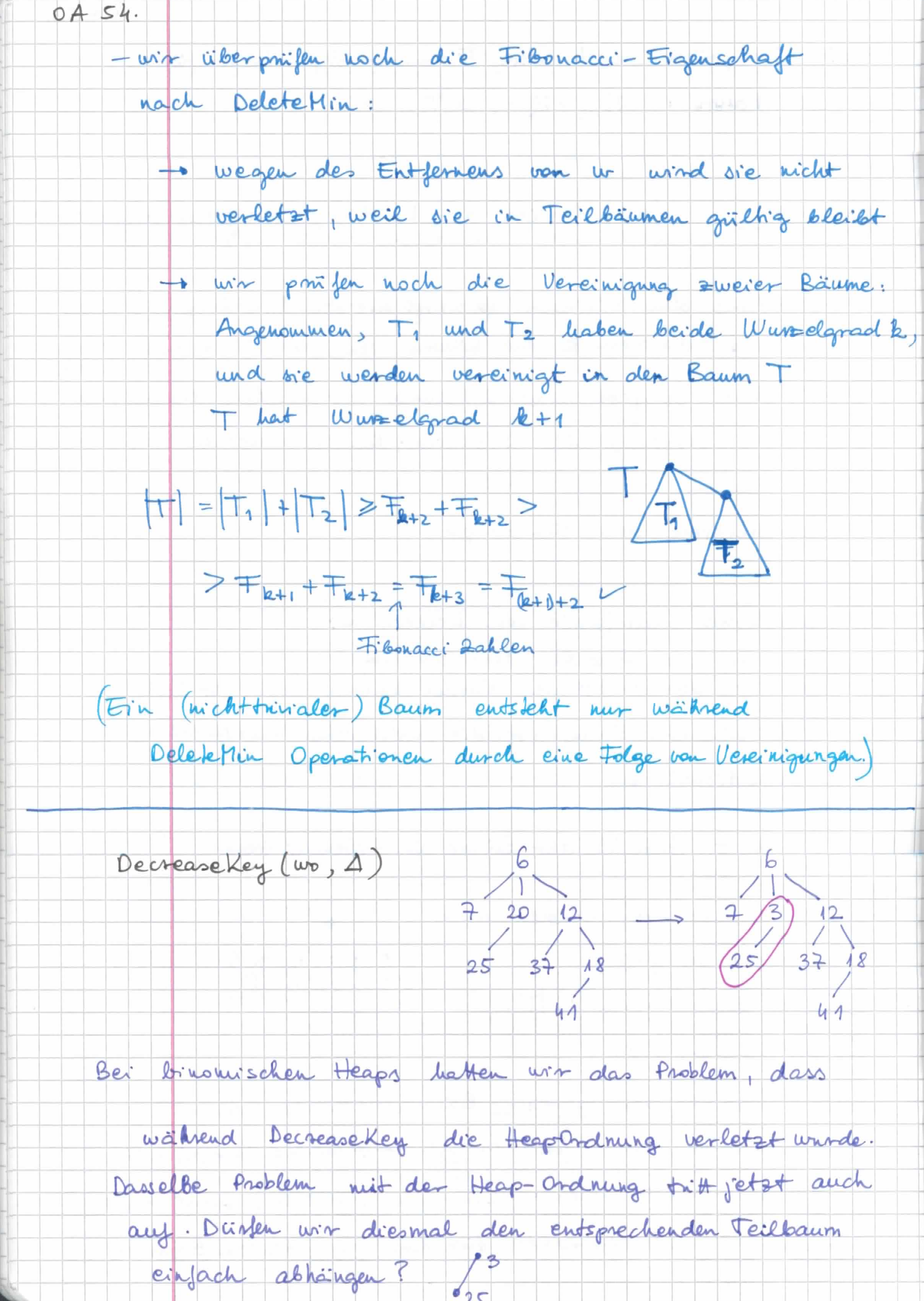
Sei m die Anzahl der Schlüssel.
Sei w der entfernte Wurzel-Knoten, und w habe k Kinder. Dann ist die wirkliche Laufzeit proportional zu

k + Anzahl der Bäume im Fibonacci-Heap

- k, die Anzahl der Kinder von w ist nicht zu groß
  sie ist logarithmisch in $|T_w|$
  und $|T_w| \leq m$

w
k

- die Anzahl der <u>anderen</u> Bäume hingegen, kann recht groß sein, fast m ggf. → keine Sorge, wir werden amortisieren: z.B. jede Insert (x) zahlt zusätzlich 1 auf das Konto für den produzierten Baum.



OA 54.

- wir überprüfen noch die Fibonacci-Eigenschaft nach DeleteMin:

  → wegen des Entfernens von w wird sie nicht verletzt, weil sie in Teilbäumen gültig bleibt

  → wir prüfen noch die Vereinigung zweier Bäume: Angenommen, T_1 und T_2 haben beide Wurzelgrad k, und sie werden vereinigt in den Baum T. T hat Wurzelgrad $k+1$

$$|T| = |T_1| + |T_2| \geq F_{k+2} + F_{k+2} >$$

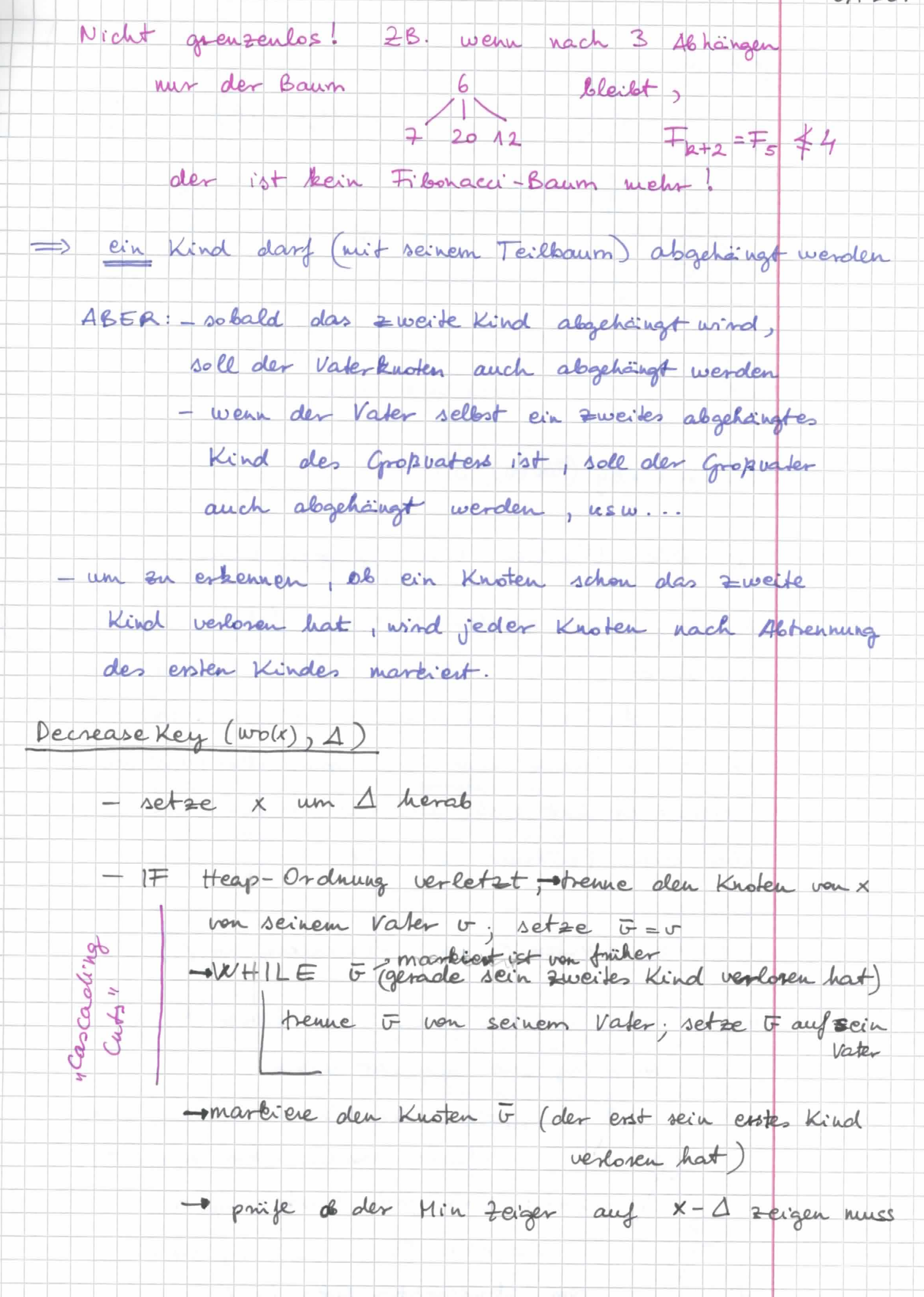
$$> F_{k+1} + F_{k+2} = F_{k+3} = F_{(k+1)+2} \checkmark$$

↑ Fibonacci Zahlen

(Ein (nichttrivialer) Baum entsteht nur während DeleteMin Operationen durch eine Folge von Vereinigungen.)

DecreaseKey (w_0, Δ)

Bei binomischen Heaps hatten wir das Problem, dass während DecreaseKey die HeapOrdnung verletzt wurde. Dasselbe Problem mit der Heap-Ordnung tritt jetzt auch auf. Dürfen wir diesmal den entsprechenden Teilbaum einfach abhängen?



Nicht grenzenlos! zB. wenn nach 3 Abhängen nur der Baum

```
    6
  / | \
 7  20 12
```

bleibt, $F_{k+2} = F_5 \neq 4$

der ist kein Fibonacci-Baum mehr!

⇒ ein Kind darf (mit seinem Teilbaum) abgehängt werden

ABER:
- sobald das zweite Kind abgehängt wird, soll der Vaterknoten auch abgehängt werden
- wenn der Vater selbst ein zweites abgehängtes Kind des Großvaters ist, soll der Großvater auch abgehängt werden, usw...

- um zu erkennen, ob ein Knoten schon das zweite Kind verloren hat, wird jeder Knoten nach Abtrennung des ersten Kindes markiert.

Decrease Key (wo(x), Δ)

- setze x um Δ herab
- IF Heap-Ordnung verletzt → trenne den Knoten von x von seinem Vater v; setze $\bar{v} = v$

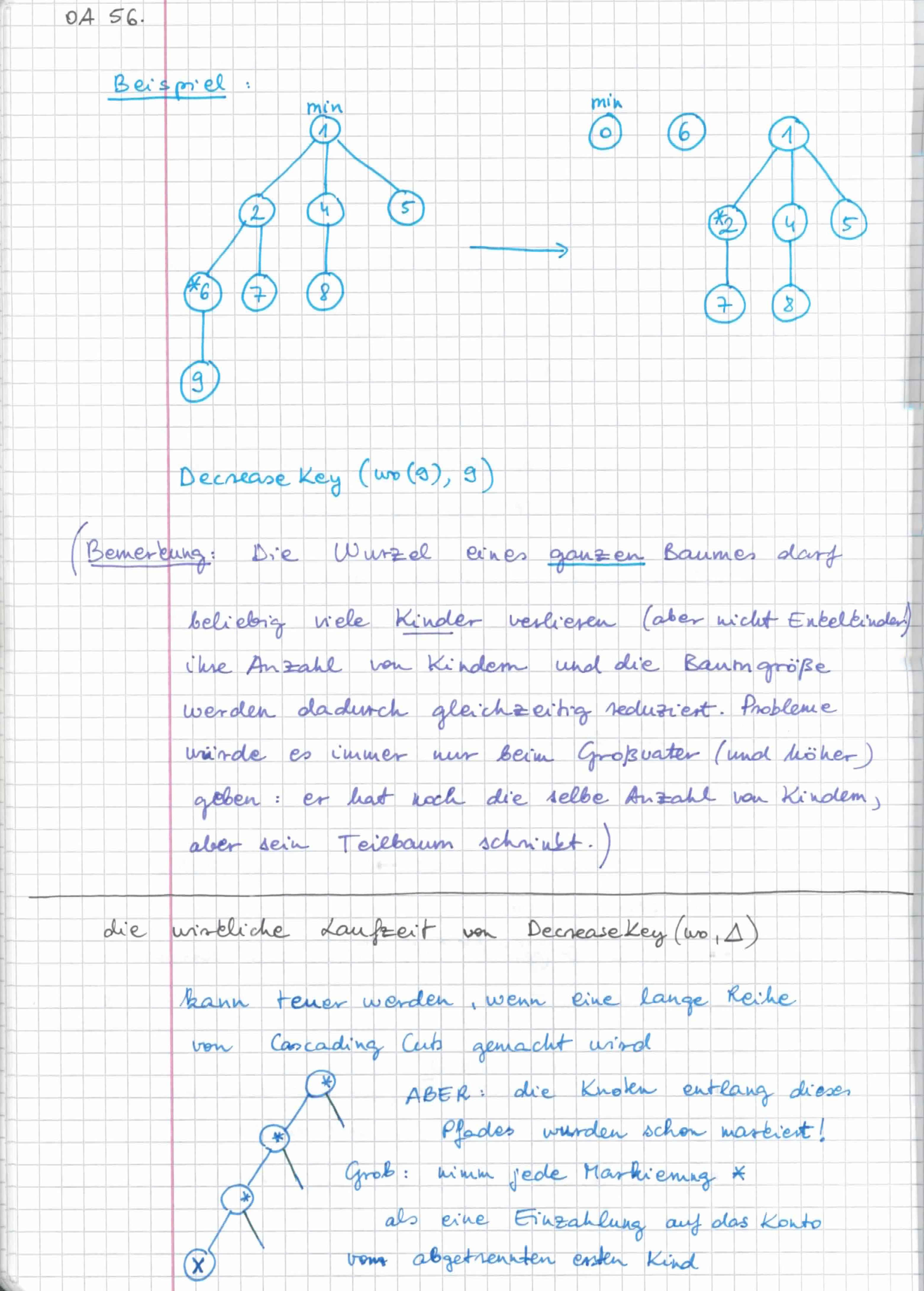
"Cascading Cuts"

→ WHILE $\bar{v}$ markiert ist von früher (gerade sein zweites Kind verloren hat)

  trenne $\bar{v}$ von seinem Vater; setze $\bar{v}$ auf sein Vater

→ markiere den Knoten $\bar{v}$ (der erst sein erstes Kind verloren hat)

→ prüfe ob der Min Zeiger auf $x - \Delta$ zeigen muss



OA 56.

Beispiel:

DecreaseKey (wo(9), 9)

(Bemerkung: Die Wurzel eines ganzen Baumes darf beliebig viele Kinder verlieren (aber nicht Enkelkinder) ihre Anzahl von Kindern und die Baumgröße werden dadurch gleichzeitig reduziert. Probleme würde es immer nur beim Großvater (und höher) geben: er hat noch die selbe Anzahl von Kindern, aber sein Teilbaum schrumpft.)

---

die wirkliche Laufzeit von DecreaseKey(wo, Δ)

kann teuer werden, wenn eine lange Reihe von Cascading Cuts gemacht wird

ABER: die Knoten entlang dieses Pfades wurden schon markiert!

Grob: nimm jede Markierung * als eine Einzahlung auf das Konto vom abgetrennten ersten Kind

(Das Bild (der Graph der abgetrennten Knoten) sieht allgemein etwas komplizierter aus: wir malen eine Kante zwischen Vater und Kind nur wenn die Abtrennung vom Kind zur Abtrennung des Vaters beigetragen hat.

Es entsteht ein <u>Binärbaum von abgetrennten Knoten</u> wobei:

- für jedes Blatt ein DecreaseKey gerufen wurde
- für jeden innere Knoten zwei Kinder abgetrennt wurden

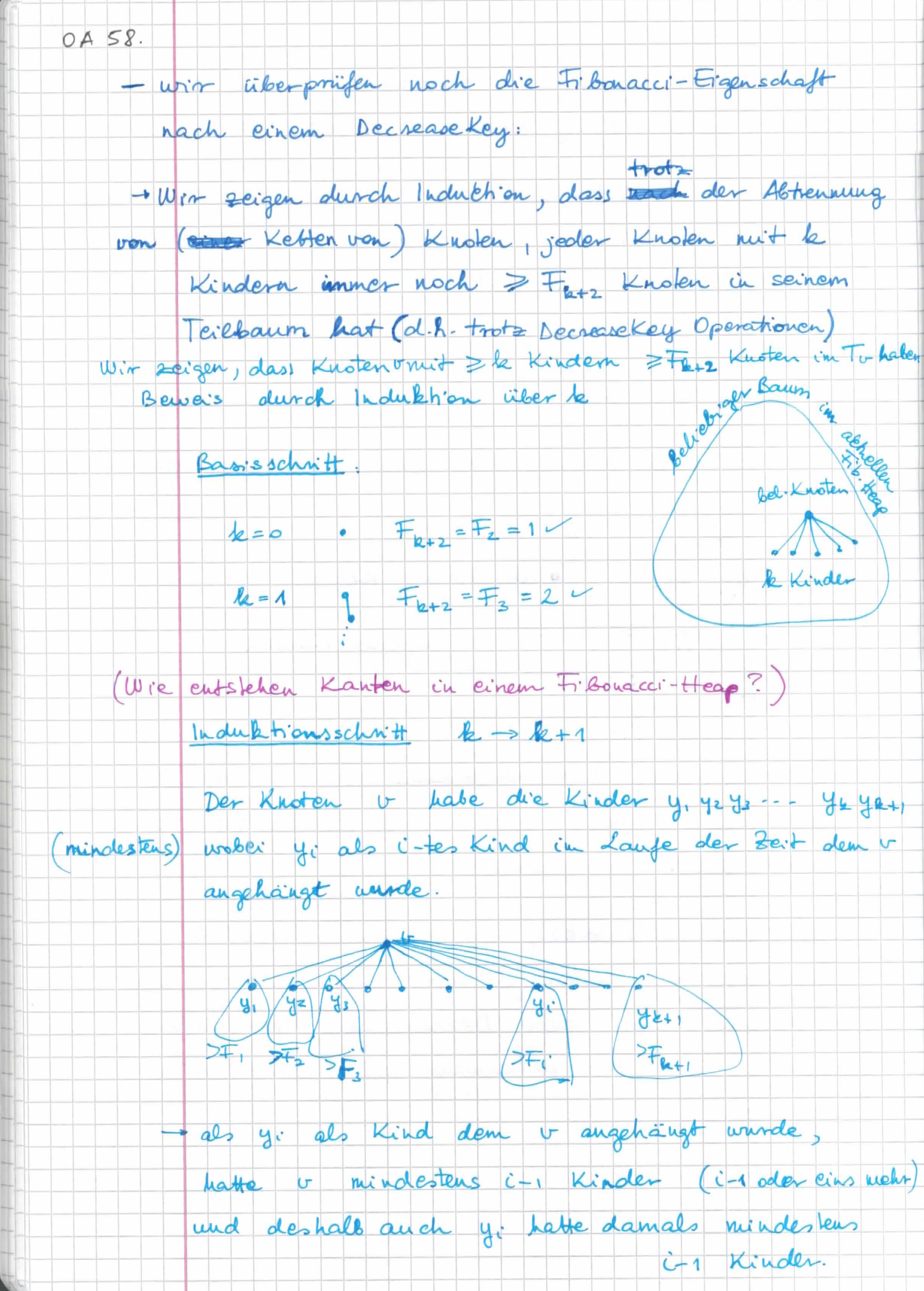
□ DecreaseKey

• abgetrennte Knoten

\ andere Kanten

Ein (Binär)baum hat ~~mindestens~~ mehr Blätter als innere Knoten, wenn jeder innere Knoten (mind.) zwei Kinder hat. Warum?

Wenn jede DecreaseKey Operation (also Blatt) die amortisierten Kosten 2 hat, dann werden die wirklichen Kosten aller DecreaseKey Operationen gedeckt (d.h. die Abtrennung aller Knoten dieses Baumes).)



OA 58.

– wir überprüfen noch die Fibonacci-Eigenschaft nach einem DecreaseKey:

→ Wir zeigen durch Induktion, dass ~~nach~~ trotz der Abtrennung von (~~einer~~ Ketten von) Knoten, jeder Knoten mit k Kindern immer noch $\geq F_{k+2}$ Knoten in seinem Teilbaum hat (d.h. trotz DecreaseKey Operationen)

Wir zeigen, dass Knoten v mit $\geq k$ Kindern $\geq F_{k+2}$ Knoten im T_v haben

Beweis durch Induktion über k

Basisschritt:

$k=0$ • $F_{k+2} = F_2 = 1$ ✓

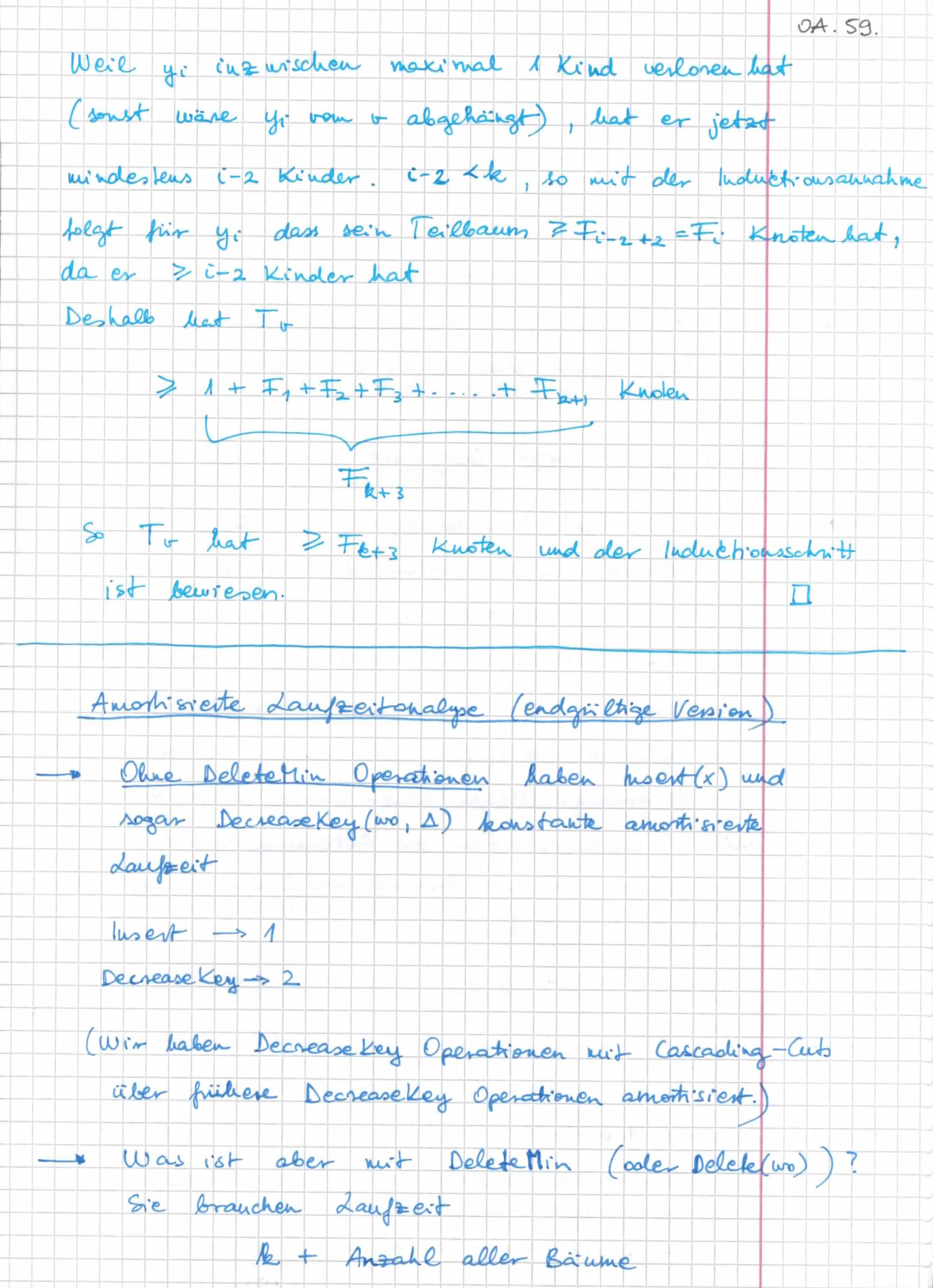
$k=1$ $F_{k+2} = F_3 = 2$ ✓

(Wie entstehen Kanten in einem Fibonacci-Heap?)

Induktionsschritt $k \to k+1$

Der Knoten v habe (mindestens) die Kinder $y_1\ y_2\ y_3 \dots y_k\ y_{k+1}$ wobei y_i als i-tes Kind im Laufe der Zeit dem v angehängt wurde.

→ als y_i als Kind dem v angehängt wurde, hatte v mindestens $i-1$ Kinder ($i-1$ oder eins mehr) und deshalb auch y_i hatte damals mindestens $i-1$ Kinder.



Weil y_i inzwischen maximal 1 Kind verloren hat (sonst wäre y_i vom v abgehängt), hat er jetzt mindestens $i-2$ Kinder. $i-2 < k$, so mit der Induktionsannahme folgt für y_i dass sein Teilbaum $\geq F_{i-2+2} = F_i$ Knoten hat, da er $\geq i-2$ Kinder hat

Deshalb hat T_v

$$\geq \underbrace{1 + F_1 + F_2 + F_3 + \ldots + F_{k+1}}_{F_{k+3}} \text{ Knoten}$$

So T_v hat $\geq F_{k+3}$ Knoten und der Induktionsschritt ist bewiesen. □

<u>Amortisierte Laufzeitanalyse (endgültige Version)</u>

→ <u>Ohne DeleteMin Operationen</u> haben Insert(x) und sogar DecreaseKey(wo, Δ) konstante amortisierte Laufzeit

Insert → 1

DecreaseKey → 2

(Wir haben DecreaseKey Operationen mit Cascading-Cuts über frühere DecreaseKey Operationen amortisiert.)

→ Was ist aber mit DeleteMin (oder Delete(wo))? Sie brauchen Laufzeit

k + Anzahl aller Bäume

k + Anzahl aller Bäume

$\downarrow$

$O(\log n)$

die Bäume können viele sein, weil Insert und DecreaseKey viele neue Bäume produzieren aber nie aufräumen.

→ Wir lassen Insert(x) 1 Kosteneinheit auf das Konto vorzahlen für den neuen Baum den er produziert hat

Insert zahlt jetzt insgesamt amortisierte Kosten 2

→ Wir lassen auch DecreaseKey (wo, Δ) für jeden neuen Baum den sie produziert 1 Kosteneinheit auf das Konto vorzahlen. Jede Abtrennung eines Knotens produziert einen neuen Baum. Die amortisierten Kosten von 2 sind Zahlungen und Vorzahlungen für sofortige und spätere Abtrennungen, 1 Kosteneinheit pro Abtrennung. Wegen neuer Bäume sollte man mit 2 Kosteneinheiten pro Abtrennung rechnen, also alle Kosten in der Analyse von DecreaseKey sollen verdoppelt werden.

DecreaseKey zahlt somit amortisierte Kosten 4

→ Die Kosten von DeleteMin (oder Delete(wo)) werden gedeckt, wenn DeleteMin selbst $O(\log_2 n)$ für die k Bäume (der Kinder vom Minimum) zahlt, weil für die vielen anderen Bäume des Fibonacci-Heaps schon vorbezahlt wurde.

DeleteMin zahlt amortisierte Kosten $O(\log_2 n)$

$(\log_\phi n)$

Abschließende Bemerkung

Der Algorithmus von Dijkstra berechnet einen kürzesten Weg zu jedem Knoten v von einem Startknoten s.
Die aktuellen Distanzwerte für jeden Knoten werden in einem Heap verwaltet $(n = |V|)$. Diese Distanzwerte können ggf. insgesamt $\Omega(|E|)$ mal aktualisiert werden (DecreaseKey), aber nur $|V|$ DeleteMin Operationen werden ausgeführt als die Distanzwerte der einzelnen Knoten endgültig werden.
Wir erhalten die Laufzeiten

Mit „gewöhnlichen" Heaps $\quad O(|E| \cdot \log|V|)$

mit Fibonacci-Heaps $\quad O(|E| + |V| \cdot \log|V|)$

($|E|$ ↙ DecreaseKey, $|V| \cdot \log|V|$ ↘ DeleteMin)

⟶ das ist $O(|E|)$ (linear) wenn $|E| = \Omega(|V| \cdot \log|V|)$

Bemerkung:

→ Für verschiedene Gegner-Modelle (adversary models) in der Analyse von randomisierten online Algorithmen siehe Borodin, El-Yaniv: S. 44–45.

→ Für optimale Lösungen bei verschiedenen Zielfunktionen im Ski-Problem (d. h. andere als die Worst-Case Analyse des Wettbewerbsfaktors, siehe Borodin, El-Yaniv: S. 335–339