Algorithmen und Datenstrukturen
BAI3-AD – SoSe 2018

Prof. Dr. Marina Tropmann-Frick

BT7, Raum 10.86

marina.tropmann-frick@haw-hamburg.de

HAW HAMBURG

1. Einführung
   Motivation, Übersicht, Begriffsklärung, Größenordnungen
2. Lineare Datenstrukturen
   Liste, Stack, Queue
3. Algorithmengrundlagen
   Die ersten Algorithmen mit Komplexitäts-(Aufwands-)analyse
4. Sortieren
   Sortieralgorithmen
5. Verzweigte Datenstrukturen
   Bäume, Operationen und Implementierungsmethoden
6. Grundlegende Graph-Algorithmen
   Graphen und Implementierungsmethoden, kürzeste Pfade, Breitensuche, Tiefensuche, Dijkstra's Algorithmus
7. **Hash-Verfahren**
   Hashfunktionen, Analyse
8. Optimierung
   Lineare Optimierung, Lösung Nichtlinearer Probleme

-- (Ausblick – weitere effiziente Algorithmen)*

(1) Einleitung

(2) Hashfunktionen

(3) Kollisionsvermeidungsstrategien

(4) Weitere Eigenschaften

Lineares Sondieren

Das lineare Sondieren kann zu folgender Formel verallgemeinert werden: Mit einer gegebener Konstante $c \in \mathbb{N}$ setzt man

$$h_i(k) = (k + c \cdot i) \bmod m.$$

Das Verfahren führt dazu, dass im Falle von Kollisionen Ketten mit Abstand c entstehen. Dies bringt jedoch keine wirkliche Verbesserung des Verfahrens. Voraussetzung ist außerdem, dass c und m teilerfremd sind, damit alle Zellen getroffen werden.

Übung 7.4. Fügen Sie die Schlüssel 6, 4, 18, 16, 13 nacheinander in ein Array der Länge 5. Verwenden Sie lineares Sondieren ($h_i(k) = (k + i) \mod 5$).

$h(6) = 1 \rightarrow \begin{array}{|c|c|c|c|c|}\hline & 6 & & & \\ \hline\end{array}$

$h(4) = 4 \rightarrow \begin{array}{|c|c|c|c|c|}\hline & 6 & & & 4 \\ \hline\end{array}$

$h(18) = 3 \rightarrow \begin{array}{|c|c|c|c|c|}\hline & 6 & & 18 & 4 \\ \hline\end{array}$

$h(16) = 1$ belegt, daher $2 \rightarrow \begin{array}{|c|c|c|c|c|}\hline & 6 & 16 & 18 & 4 \\ \hline\end{array}$

$h(13) = 3$ belegt, 4 belegt, daher $0 \rightarrow \begin{array}{|c|c|c|c|c|}\hline 13 & 6 & 16 & 18 & 4 \\ \hline\end{array}$

Quadratisches Sondieren

- Bei diesem Verfahren werden neue Adressen mit quadratischem Abstand erzeugt.

- Die Idee ist, im Falle einer Kollision, nicht nur die unmittelbare Nachbarschaft zu prüfen, sondern in der Annahme dort auch vorwiegend auf besetzte Felder zu stoßen, die Schrittweite mit der Anzahl der Kollisionen zu erhöhen.

- Dies führt dazu, dass keine Ketten mehr entstehen.
- Die Folge der Hashfunktionen lautet hier:

$$h_i(k) = (h_0(k) + i^2) \bmod m.$$

Quadratisches Sondieren

Wählt man für m eine Primzahl der Form $m = 4 \cdot j + 3$, so werden alle Zellen getroffen ([Rad70]). Konkret hat man also z.B. für $m = 1019$ (Primzahl und bei Division durch 4 Rest 3) und für

$$h_0(k) = k \bmod m$$

$$\begin{aligned} h_0(k) &= k \bmod 1019 \\ h_1(k) &= k + 1 \bmod 1019 \\ h_2(k) &= k + 4 \bmod 1019 \\ h_3(k) &= k + 9 \bmod 1019 \\ h_4(k) &= k + 16 \bmod 1019 \end{aligned}$$

Beispiel (Quadratisches Sondieren). Gegeben seien Elemente mit den folgenden Schlüsseln:

$$\kappa = \{24, 51, 63, 77, 85, 99\}.$$

Die Arrayelemente nennen wir $a[i]$. Als Hashfunktion $h()$ verwenden wir

$$h(k) = k \bmod m \quad \text{mit } m = 7.$$

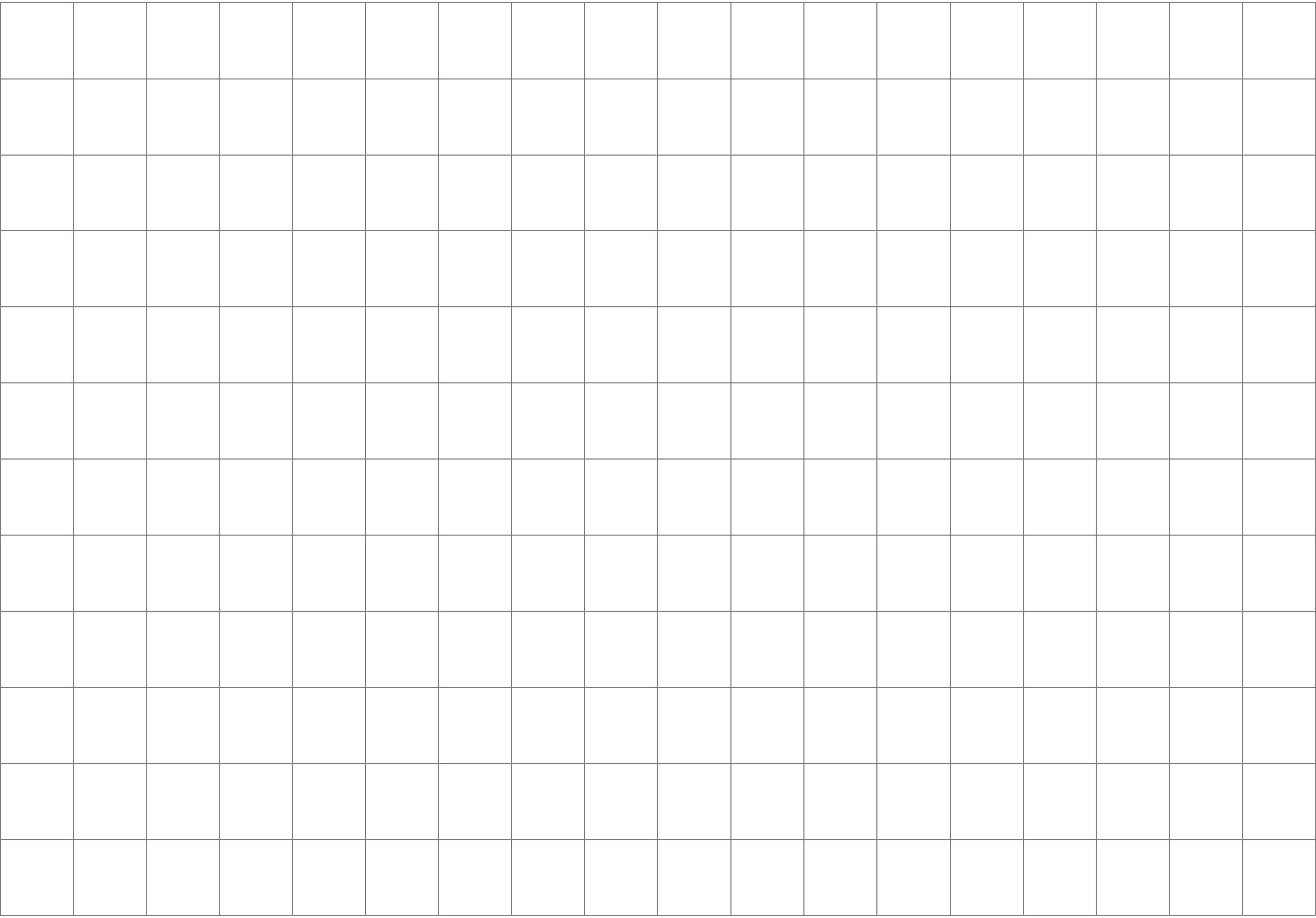
Wir wollen bei Kollision quadratisch sondieren. Zunächst erhalten wir mit Hashfunktion h_0

0	1	2	3	4	5	6
63		51	24			

für die Elemente 24, 51 und 63. Bei 77 entsteht eine Kollision bei $a[0]$. Die lösen wir auf, indem wir Hashfunktion $h_1(77)$ anwenden

$$h_1(77) = (h_0(77) + 1) \bmod 7 = 1.$$

0	1	2	3	4	5	6
63	77	51	24			



Beispiel (Quadratisches Sondieren). Gegeben seien Elemente mit den folgenden Schlüsseln:

$$\kappa = \{24, 51, 63, 77, 85, 99\}.$$

0	1	2	3	4	5	6
63	77	51	24			

Das Element $k = 85$ liefert $h_0(85) = 1$.

Diese Zelle ist nun gerade belegt worden. Hätte man 85 vor 77 einsortiert, dann wäre 85 an die Position 1 gekommen. Man sieht, dass die Belegung der Hashtabelle von der Reihenfolge der Eingabedaten abhängt. Wir wenden also h_1 an und erhalten

$$h_1(85) = (h_0(85) + 1) \bmod 7 = (1 + 1) \bmod 7 = 2.$$

Diese Zelle ist auch schon belegt. Also wenden wir h_2 an

$$h_2(85) = (h_0(85) + 4) \bmod 7 = (1 + 4) \bmod 7 = 5.$$

Hier sortieren wir also 85 ein

0	1	2	3	4	5	6
63	77	51	24		85	

Beispiel (Quadratisches Sondieren). Gegeben seien Elemente mit den folgenden Schlüsseln:

$$\kappa = \{24, 51, 63, 77, 85, 99\}.$$

0	1	2	3	4	5	6
63	77	51	24		85	

Nun soll noch die 99 untergebracht werden. Anwendung von h_0 ergibt

$$h_0(99) = 99 \bmod 7 = 1.$$

Diese Zelle ist aber schon vergeben. Welches h_i bringt 99 auf einen freien Platz?

Um eine Verbesserung zu erreichen, wird das quadratische Sondieren oft mit alternierenden Schritten eingesetzt, mal nach vorne ($k + i^2$), mal nach hinten ($k - i^2$).

Double Hashing

In diesem Fall wird zusätzlich zur Hashfunktion h eine zweite Hashfunktion h' verwendet, welche zur ersten *unabhängig* ist. Dies bedeutet, dass die Wahrscheinlichkeiten bei beiden Hashfunktionen gleichzeitig eine Kollision zu erzeugen unabhängig sind. Oft begnügt man sich damit, eine *intuitiv* unabhängige zweite Hashfunktion h' zu finden. Die Idee ist also, den Schlüsselwerten, die unter der ersten Funktion h eine Kollision haben, mit der zweiten Funktion h' eine individuelle Schrittweite zu geben.
Ist m eine Primzahl und

$$h(k) = k \mod m,$$

dann ist

$$h'(k) = 1 + (k \mod (m-2))$$

eine gute Wahl. Die Folge der Hashfunktionen definiert man dann wie folgt:

$$h_i(k) = (h(k) + h'(k) \cdot i^2) \mod m.$$

Double Hashing

Beispiel. Gegeben seien Elemente mit den folgenden Schlüsseln:

$$\kappa = \{24, 51, 63, 77, 85, 99\}.$$

$$h(k) = k \bmod m \quad \text{mit } m = 7.$$

Wir wollen bei Kollision mit der Hashfunktion $h'(k) = 1 + (k \mod 5)$ sondieren. Bei den ersten drei Elementen gibt es keine Kollision.

0	1	2	3	4	5	6
63		51	24			

Beispiel. Gegeben seien Elemente mit den folgenden Schlüsseln:

0	1	2	3	4	5	6
63		51	24			

$$\kappa = \{24, 51, 63, 77, 85, 99\}.$$

Bei 77 gibt es eine Kollision an Position 0. Diese versuchen wir nun mit h_1 aufzulösen.

$$h_1(77) = (h(77) + h'(77)) \mod 7 = (0 + (1 + (77 \mod 5))) \mod 7 = (0 + (1 + 2)) = 3.$$

Dies ergibt eine Kollision. Wir wenden nun die nächste Hashfunktion h_2 an und erhalten:

$$h_2(77) = (h(77) + h'(77) \cdot 2^2) \mod 7 = (0 + 3 \cdot 2^2) \mod 7 = 12 \mod 7 = 5.$$

Diese Zelle ist frei und wir tragen ein

0	1	2	3	4	5	6
63		51	24		77	

Beispiel. Gegeben seien Elemente mit den folgenden Schlüsseln:

$$\kappa = \{24, 51, 63, 77, 85, 99\}.$$

Für 85 erhalten wir $h_0(85) = 85 \bmod 7 = 1$

und für die 99 $h_0(99) = 99 \bmod 7 = 1$ wieder eine Kollision

Der zweite Anlauf liefert

$$h_1(99) = (h(99) + h'(99)) \bmod 7 = (1 + (1 + (99 \bmod 5))) \bmod 7 = (1 + (1 + 4)) = 6.$$

0	1	2	3	4	5	6
63	85	51	24		77	99

Löschen von Elementen

Das Löschen von Elementen aus einer Hash-Tabelle mit offener Adressierung erfordert einen Kniff. Folgendes Beispiel beschreibt die Problematik und den Kniff.

Beispiel. Gegeben seien Elemente mit den folgenden Schlüsseln:

$$\kappa = \{63, 77\}.$$

Als Hashfunktion verwenden wir double hashing aus dem vorherigen Beispiel mit

$$h(k) = k \mod 7 \text{ und } h'(k) = 1 + (k \mod 5).$$

Bei 77 gibt es eine Kollision mit 63. Also fügen wir 77 ein mit $h_1(77) = 3$.

0	1	2	3	4	5	6
63			77			

Löschen von Elementen

0	1	2	3	4	5	6
63			77			

Nun soll 63 gelöscht werden:

0	1	2	3	4	5	6
			77			

Jetzt wollen wir 77 wiederfinden. Wir suchen bei $h_0(77) = 0$. Dort finden wir die 77 nicht und schließen fälschlicherweise daraus, dass die 77 nicht in der Hashtabelle enthalten ist. Wir wollen ja auch nicht alle Hashfunktionen $h_0, \ldots, h_r$ durchprobieren.

Der Kniff besteht nun darin, an der gelöschten Stelle ein Flag zu setzen, welches kennzeichnet, dass dort einmal ein Element stand. Wir löschen 63 und setzen das Flag:

0	1	2	3	4	5	6
$Flag{=}X$			77			

Löschen von Elementen

0	1	2	3	4	5	6
$Flag{=}X$			77			

Jetzt wollen wir 77 wiederfinden. Wir suchen bei $h_0(77) = 0$. Dort finden wir das $Flag = X$. Dies bedeutet, wir müssen bei $h_1(77) = 3$ auch noch suchen und finden dort die 77.

Nun wollen wir 14 einfügen. Es ist $h_0(14) = 0$. Dort ist die Stelle frei. Das Flag beachten wir nicht. Wir können die 14 reinschreiben. Das Flag wird nicht zurückgesetzt. Wir erhalten:

0	1	2	3	4	5	6
$14^{Flag=X}$			77			

Tatsächlich benötigt man in jedem Feld ein Flag. Die Hashtabelle sieht also so aus:

0	1	2	3	4	5	6
$14^{Flag=X}$	$Flag{=}0$	$Flag{=}0$	$77^{Flag=0}$	$Flag{=}0$	$Flag{=}0$	$Flag{=}0$

- Die Begriffe **load factor** und **capacity** (Kapazität) beschreiben die Auslastung und Größe einer Hashtabelle.
- Ist eine Hashtabelle zu voll, dann muss ein **resize** durchgeführt werden.

load factor

load factor gibt an, wie weit eine Hashtabelle ausgelastet ist. Der *load factor* liegt zwischen 0.0 und 1.0. Er bestimmt sich als

$$\textit{load factor} = \frac{\text{Anzahl der belegten Felder}}{\text{Gesamtgröße } m \text{ der Hashtabelle}}.$$

Ist ein maximaler *load factor*, beispielsweise 0,8 bzw. 80%, überschritten, so wird die Hashtabelle ineffektiv und muss vergrößert werden.
Wird ein minimaler *load factor* unterschritten, so wird zu viel Speicher verbraucht.

capacity

Die *capacity* ist einfach die Größe m der Hashtabelle.
Die Hashtabelle soll automatisch vergrößert werden wenn der *load factor* einen maximalen *load factor*, beispielsweise 80%, überschritten hat. Man erhält eine neue *capacity* m_{neu}. In diesem Fall spricht man vom *resize* einer Hashtabelle.
Die Daultwerte in Java sind für den *load factor* 0.75 und für die *capacity* 11, d.h. Java vergrössert die Hashtabelle, wenn die bestehende Tabelle zu 75% ausgelastet ist.

resize

Beim *resize* wird eine neue *capacity* m_{neu} bestimmt. Diese wird so gewählt, dass die bereits eingefügten Elemente einen neuen minimalen *load factor*, beispielsweise 50%, ergeben.
Es ist zu beachten, dass beim *resize* alle bereits eingefügten Elemente neu eingefügt werden müssen. Die Hashfunktionen haben sich ja nun verändert. Statt $\bmod m$ hat man nun $\bmod m_{neu}$.

Fragen / Anregungen ?

1. Einführung

    Motivation, Übersicht, Begriffsklärung, Größenordnungen

2. Lineare Datenstrukturen

    Liste, Stack, Queue

3. Algorithmengrundlagen

    Die ersten Algorithmen mit Komplexitäts-(Aufwands-)analyse

4. Sortieren

    Sortieralgorithmen

5. Verzweigte Datenstrukturen

    Bäume, Operationen und Implementierungsmethoden

6. Grundlegende Graph-Algorithmen

    Graphen und Implementierungsmethoden, kürzeste Pfade, Breitensuche, Tiefensuche, Dijkstra's Algorithmus

7. Hash-Verfahren

    Hashfunktionen, Analyse

8. **Optimierung**

    Lineare Optimierung, Lösung Nichtlinearer Probleme

-- (Ausblick – weitere effiziente Algorithmen)*

Anwendungsbereiche

- Von privat bis zu Großkonzernen → Gewinn maximieren
  - Fehlerfunktion bei numerischen Simulationen
  - Wirtschaftswissenschaften
  - Finanzmathematik
  - Data Science (z.B. Parameter-Optimierung in Modellen)
  - Klimaforschung (Optimierung physikalischer Phänomene unter Randbedingungen wie Luftfeuchtigkeit, Luftdruck, Temperatur, ...)
  - Spieltheorie (siehe Damenproblem)
  - etc ...

Lineare Optimierung - Beispiel für Problemstellung:

Nebenjob eines Professors: Professor Milchmann und seine Familie verkaufen Speiseeis und Butter. Die Milch haben sie von ihren drei Kühen, die in der Woche zusammen 22 Fass Milch liefern. Für ein Kilogramm Butter werden 2 Fass Milch benötigt, für ein Fass Speiseeis werden 3 Fass Milch benötigt. Im Kühlschrank kann beliebig viel Butter gelagert werden. Im Gefrierschrank haben 6 Fass Eis Platz. In einer viertel Stunde können 1/4 Kilogramm Butter oder aber 1 Fass Eis hergestellt werden. Die Familie hat 6 (24 mal 15 Minuten) Stunden Zeit für die Eis und Butter Produktion.
Ein Kilogramm Butter kann mit einem Gewinn von 4€ verkauft werden, ein Fass Speiseeis mit einem Gewinn von 5€. Der Gesamtgewinn G in € berechnet sich also zu:

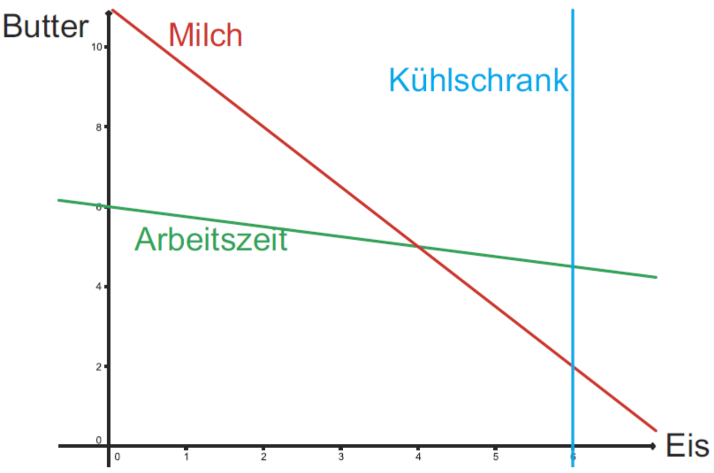
$$G = 5 \cdot x_1 + 4 \cdot x_2$$

Lineare Optimierung – Beispiel: **Nebenjob eines Professors**

Dazu müssen noch die Randbedingungen als Ungleichungen formuliert werden:

x_1 steht steht für 1 Fass Eis , x_2 für 1 kg Butter

$$x_1 \leq 6$$
$$x_1 + 4 \cdot x_2 \leq 24$$
$$3 \cdot x_1 + 2 \cdot x_2 \leq 22$$



Berechnung der optimalen Lösung unter Beachtung der Bedingungen

→ **Simplex-Algorithmus**

Lösung nicht-linearer Probleme

→ **Newton-Verfahren**

- Iteratives Verfahren zur Berechnung von Nullstellen
- Besonders wichtig, um globale Extremwerte finden zu können

Das Newton-Verfahren (benannt nach seinem Entdecker Isaac Newton) sucht nach einem Wert x_n für eine Funktion f, sodass $f(x_n) = 0$. Das Verfahren beginnt mit einem Startwert x_0. Im Folgenden werden iterativ neue Werte x_{n+1} nach folgender Vorschrift berechnet:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Das Verfahren setzt also voraus, dass die Funktion f stetig differenzierbar ist.

Fragen / Anregungen ?

Beispiel - Das Prominentensuche Problem

Ein Prominenter (celebrity) ist jemand, den alle kennen, der jedoch selbst keinen kennt.

<u>Eingabe</u>:

1. n Personen nummeriert 0,…,n-1
2. mindestens eine Person ist ein Prominenter
3. n x n boolean Matrix M, so dass für $0 \leq i, j < n$:

$$M[i,j] = \begin{cases} 1 & \text{falls Person } i \text{ kennt Person } j \\ 0 & \text{sonst} \end{cases}$$

alle kennen Person p

Person p kennt niemanden

<u>Ausgabe</u>: Sei $p \in \{0, \dots, n-1\}$, so dass Person p ein Prominenter ist d.h.:

$$\forall\ 0 \leq i < n, i \neq p \ \Rightarrow M[i,p] \text{ und } \forall\ 0 \leq i < n, i \neq p \ \Rightarrow \neg M[p,i]$$

Beispiel - Das Prominentensuche Problem

Ein Prominenter (celebrity) ist jemand, den alle kennen, der jedoch selbst keinen kennt.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Laufzeit (einfach) ? $\mathcal{O}(n^2)$

Beispiel - Das Prominentensuche Problem

Ein Prominenter (celebrity) ist jemand, den alle kennen, der jedoch selbst keinen kennt.

Übung: Überlegen Sie sich eine geeignete Strategie für eine **Lineare Suche in $\mathcal{O}(n)$**

(eine mögliche Aufgabe für Einstellungstest bei Google, Microsoft & Co.)

Strategie:

- starte die Suche am M[0, 0] und suche bis eine 1 gefunden wird in Zeile 0, Spalte m
- dann gilt: $\forall\ 0 \leq p < m\ \rightarrow p$ ist kein Prominenter
- die Suche geht dann weiter in Zeile m, Spalte m usw.

Beispiel - Das Prominentensuche Problem

Strategie:

- starte die Suche am M[0, 0] und suche bis eine 1 gefunden wird in Teile 0, Spalte m
- dann gilt: $\forall\ 0 \leq p < m\ \rightarrow p$ ist kein Prominenter
- die Suche geht dann weiter in Zeile m, Spalte m usw.

Übung Teil 2: Formulieren Sie den Algorithmus für die **Lineare Suche in $\mathcal{O}(n)$**

```
int promLinearSearch(bool M[], int n) {
  int row = 0; column = 0;
  while (row != n && column != n) {
    if (row != column) {
      if (!M[row,column]) { column = column + 1; }
      if (M[row,column]) { row = column; }
    } else { column = column + 1; }
  }
  return row
}
```

Beispiel - Das Prominentensuche Problem

Eigenschaften:

$\forall\ 0 < i \neq j \leq n$ gilt:

- $M[i, j] \rightarrow i$ ist kein Prominenter
- $\neg M[j, i] \rightarrow i$ ist kein Prominenter

Übung Teil 3: Formulieren Sie den Algorithmus für die **Bilineare Suche in $\mathcal{O}(n)$**

```
int promBilinearSearch(bool M[], int n) {
  int row = 0; column = n-1;
  while (row != column) {
      if (M[row,column]) { row = row + 1; }
      if (!M[row,column]) {column = column-1; }
  }
  return row
}
```

Binäre Suche

Eingabe: Sortiertes Array a mit n Einträgen und das gesuchte Element e.
Ausgabe: Ist e in a enthalten?
Idee: halbiere den Suchraum in jedem Durchlauf

Übung: Schreiben Sie 2 Varianten für einen Algorithmus für die binäre Suche

Binäre Suche → 1.Variante

```
bool binarySearch(int a[], int n, int e) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = floor((left + right) / 2)
        if (a[mid] == e) { return true; }
        if (a[mid] > e) { right = mid - 1; }
        if (a[mid] < e) { left = mid + 1; }
    }
    return false; // nicht gefunden
}
```


Binäre Suche → 2.Variante

```
bool binarySearchRecursive(int a[], int e, int left, int right) {
    if (left > right) return false;
    int mid = floor((left + right) / 2);
        if (a[mid] == e) { return true; }
        if (a[mid] > e) {
         return binarySearchRecursive(a, e, left, mid-1);
        }
        if (a[mid] < e) {
         return binarySearchRecursive(a, e, mid+1, right);
        }
    }
    return false; // nicht gefunden
}
```


Übung: Sparse Search

(eine mögliche Aufgabe für Einstellungstest bei Google, Microsoft & Co.)

Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string.

Beispiel:

Input: ball, {„at“, „“, „“, „“, „ball“, „“, „“, „car“, „“, „“, „dad“, „“, „“}

Output: 4

Fragen / Anregungen ?