14. Hashing

Hashtabellen, Pre-Hashing, Hashing, Kollisionsauflösung durch Verketten, Einfaches gleichmässiges Hashing, Gebräuchliche Hashfunktionen, Tabellenvergrösserung, offene Addressierung: Sondieren, Gleichmässiges Hashing [Ottman/Widmayer, Kap. 4.1-4.3.2, 4.3.4, Cormen et al, Kap. 11-11.4]

Einstiegsbeispiel

Ziel: Effiziente Verwaltung einer Tabelle aller n ETH-Studierenden.

Anforderung: Schneller Zugriff (Einfügen, Löschen, Finden) von Datensätzen nach Name.

Wörterbuch (Dictionary)

Abstrakter Datentyp (ADT) D zur Verwaltung einer Menge von Einträgen[18] $i = (k, v)$ mit Schlüsseln $k \in \mathcal{K}$. Operationen mindestens:

- **`insert`**(D, i): Hinzufügen oder Überschreiben von i im Wörterbuch D.
- **`delete`**(D, i): Löschen von i aus dem Wörterbuch D. Nicht vorhanden $\Rightarrow$ Fehlermeldung.
- **`search`**(D, k): Liefert Eintrag mit Schlüssel k, wenn er existiert.

[18] Schlüssel-Wert Paare (k, v), im Folgenden betrachten wir hauptsächlich die Schlüssel.

Wörterbuch in C++

Assoziativer Container `std::unordered_map<>`

```
// Create an unordered_map of strings that map to strings
std::unordered_map<std::string, std::string> colours = {
  {"RED","#FF0000"}, {"GREEN","#00FF00"}
};

colours["BLUE"] = "#0000FF"; // Add

std::cout << "The hex value of color red is: "
          << colours["RED"] << "\n";

for (const auto& entry : colours) // iterate over key-value pairs
  std::cout << entry.first << ": " << entry.second << '\n';
```

Motivation/Anwendungen

Wahrscheinlich *die* gängigste Datenstruktur

- Unterstützt in vielen Programmiersprachen (C++, Python, Javascript, Java, C#, Ruby, …)
- Offensichtliche Verwendung
  - Datenbankenn
  - Symboltabellen in Compilern und Interpretern
  - Objekte in dynamisch typisierten Sprachen, z.B. Python, Javascript
- Weniger offensichtlich
  - Substring-Suche (e.g. Rabin-Karp)
  - Ähnlichkeit von Zeichenfolgen (z.B. Dokumentenvergleich, DNA)
  - Dateisynchronisation (z.B. git, rsync)
  - Kryptographie (z.B. Identifizierung, Authentifizierung)

Idee: Schlüssel als Indizes

Index	Eintrag
0	-
1	-
2	-
3	[3,wert(3)]
4	-
5	-
⋮	⋮
k	[k,wert(k)]
⋮	⋮

Probleme

1. Schlüssel müssen nichtnegative ganze Zahlen sein
2. Grosser Schlüsselbereich $\Rightarrow$ grosses Array

Lösung des ersten Problems: Prehashing

Prehashing: Bilde Schlüssel ab auf positive Ganzzahlen mit einer Funktion $ph : \mathcal{K} \to \mathbb{N}$

- Theoretisch immer möglich, denn jeder Schlüssel ist als Bitsequenz im Computer gespeichert
- Theoretisch auch: $x = y \Leftrightarrow ph(x) = ph(y)$
- In der Praxis: APIs bieten Funktionen zum Pre-hashing an (Java: **`object.hashCode()`**, C++: **`std::hash<>`**, Python: **`hash(object)`**)
- APIs bilden einen Schlüssel aus der Schlüsselmenge ab auf eine Ganzzahl mit beschränkter Grösse[19]

[19] Somit gilt die Implikation $ph(x) = ph(y) \Rightarrow x = y$ **nicht** mehr für alle x,y.

Prehashing-Beispiel: String

Zuordnung Name $s = s_1 s_2 \ldots s_{l_s}$ zu Schlüssel

$$ph(s) = \left(\sum_{i=1}^{l_s} s_i \cdot b^i \right) \bmod 2^w$$

b so, dass verschiedene Namen möglichst verschiedene Schlüssel erhalten.

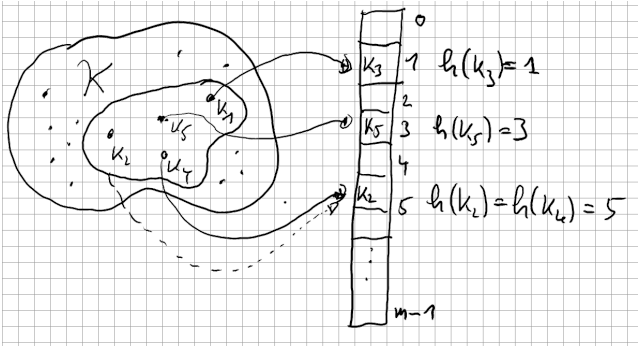
w Wortgrösse des Systems (z.B. 32 oder 64).

Beispiel mit $b = 31$, $w = 32$, ASCII-Werte s_i

Anna $\mapsto 92966272$
Anne $\mapsto 96660356$
Heinz-Harald $\mapsto 81592996699304236533 \bmod 2^{32} = 631641589$

Lösung des zweiten Problems: Hashing

Reduziere das Schlüsseluniversum: Abbildung (Hashfunktion)
$h : \mathcal{K} \to \{0, ..., m-1\}$ ($m \approx n =$ Anzahl Einträge in der Tabelle)



Kollision: $h(k_i) = h(k_j)$.

Nomenklatur

Hashfunktion h: Abbildung aus der Menge der Schlüssel $\mathcal{K}$ auf die Indexmenge $\{0, 1, \ldots, m-1\}$ eines Arrays (*Hashtabelle*).

$$h : \mathcal{K} \to \{0, 1, \ldots, m-1\}.$$

Meist $|\mathcal{K}| \gg m$. Es gibt also $k_1, k_2 \in \mathcal{K}$ mit $h(k_1) = h(k_2)$ (*Kollision*).

Eine Hashfunktion sollte die Menge der Schlüssel möglichst gleichmässig auf die Positionen der Hashtabelle verteilen.

Beispiele gebräuchlicher Hashfunktionen

Divisionsmethode

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

Ideal: m Primzahl, nicht zu nahe bei Potenzen von 2 oder 10
(siehe z.B. Cormen et al. „Introduction to Algorithms“, Donald E. Knuth „The Art of Computer Programming“).

Aber oft: $m = 2^r - 1$ ($r \in \mathbb{N}$), da Tabellenwachstum per Verdoppelung (mehr später).

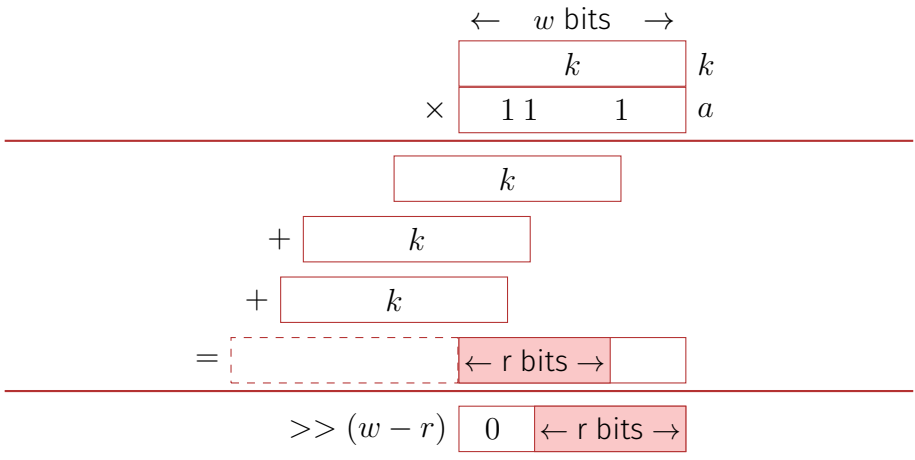
Beispiele gebräuchlicher Hashfunktionen

Multiplikationsmethode

$$h(k) = \left\lfloor (a \cdot k \bmod 2^w)/2^{w-r} \right\rfloor \bmod m$$

- Guter Wert für a: $\left\lfloor \frac{\sqrt{5}-1}{2} \cdot 2^w \right\rfloor$: Integer, der die ersten w Bits des gebrochenen Teils der irrationalen Zahl darstellt.
- Tabellengrösse $m = 2^r$, $w =$ Grösse des Maschinenworts in Bits.
- Multiplikation addiert k entlang aller Bits von a, Ganzzahldivision durch 2^{w-r} und $\bmod m$ extrahieren die oberen r Bits.
- Als Code geschrieben sehr einfach: `a * k >> (w-r)`

Illustration

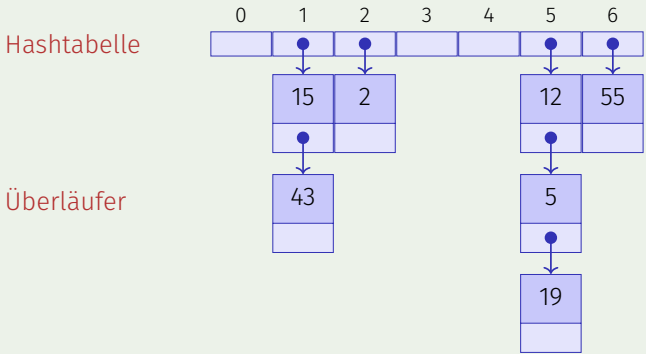


Behandlung von Kollisionen: Verkettung

$m = 7, \mathcal{K} = \{0, \ldots, 500\}, h(k) = k \bmod m.$

Schlüssel 12 , 55 , 5 , 15 , 2 , 19 , 43

Direkte Verkettung der Überläufer



Algorithmen zum Hashing mit Verkettung

Es sei H eine Hashtabelle mit Überlauflisten.

- **`insert`**(H, i) Prüfe ob Schlüssel k vom Eintrag i in Liste an Position $h(k)$. Falls nein, füge i am Ende der Liste ein; andernfalls ersetze das Element durch i.
- **`find`**(H, k) Prüfe ob Schlüssel k in Liste an Position $h(k)$. Falls ja, gib die Daten zum Schlüssel k zurück. Andernfalls Rückgabe eines leeren Elements **`null`**.
- **`delete`**(H, k) Durchsuche die Liste an Position $h(k)$ nach k. Wenn Suche erfolgreich, entferne das entsprechende Listenelement.

Worst-case Analyse

Schlechtester Fall: alle Schlüssel werden auf den gleichen Index abgebildet.

$\Rightarrow \Theta(n)$ pro Operation im schlechtesten Fall. 

Einfaches Gleichmässiges Hashing

Starke Annahmen: Jeder beliebige Schlüssel wird

- mit gleicher Wahrscheinlichkeit (Uniformität)
- und unabhängig von den anderen Schlüsseln (Unabhängigkeit)

auf einen der m verfügbaren Slots abgebildet.

Einfaches Gleichmässiges Hashing

Unter der Voraussetzung von einfachem gleichmässigen Hashing:
Erwartete Länge einer Kette, wenn n Elemente in eine Hashtabelle mit m Elementen eingefügt werden

$$\begin{aligned}\mathbb{E}(\text{Länge Kette j}) &= \mathbb{E}\left(\sum_{i=0}^{n-1} \mathbb{1}(h(k_i) = j)\right) = \sum_{i=0}^{n-1} \mathbb{P}(h(k_i) = j) \\ &= \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m}\end{aligned}$$

$\alpha = n/m$ heisst *Belegungsfaktor* oder *Füllgrad* der Hashtabelle.

Einfaches Gleichmässiges Hashing

Theorem 17

Sei eine Hashtabelle mit Verkettung gefüllt mit Füllgrad $\alpha = \frac{n}{m} < 1$. *Unter der Annahme vom einfachen gleichmässigen Hashing hat die nächste Operation erwartete Laufzeitkosten von* $\Theta(1 + \alpha)$.

Folgerung: ist die Anzahl der Slots m der Hashtabelle immer mindestens proportional zur Anzahl Elemente n in der Hashtabelle, $n \in \mathcal{O}(m) \Rightarrow$ Erwartete Laufzeit der Operationen Suchen, Einfügen und Löschen ist $\mathcal{O}(1)$.

Weitere Analyse (direkt verkettete Liste)

1. Erfolglose Suche. Durchschnittliche Listenlänge ist $\alpha = \frac{n}{m}$. Liste muss ganz durchlaufen werden.
   $\Rightarrow$ Durchschnittliche Anzahl betrachteter Einträge

$$C'_n = \alpha.$$

2. Erfolgreiche Suche. Betrachten die Einfügehistorie: Schlüssel j sieht durchschnittliche Listenlänge $(j-1)/m$.
   $\Rightarrow$ Durchschnittliche Anzahl betrachteter Einträge

$$C_n = \frac{1}{n}\sum_{j=1}^{n}(1+(j-1)/m)) = 1 + \frac{1}{n}\frac{n(n-1)}{2m} \approx 1 + \frac{\alpha}{2}.$$

Vor- und Nachteile der Verkettung

Vorteile der Strategie:

- Belegungsfaktor grösser 1 möglich (mehr Einträge als Tabellenplätze)
- Entfernen von Schlüsseln sehr einfach (relativ zur später vorgestellten Alternative)

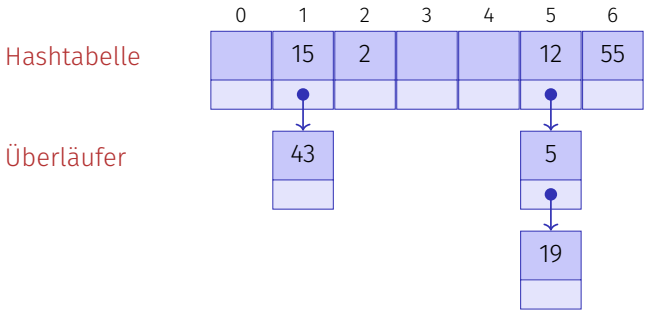
Nachteile:

- Lineare Laufzeit bei degenerierten Hashtabellen mit langen Kollisionsketten
- (Speicherverbrauch der Verkettung)

Besser: Kollisionswahrscheinlichkeit reduzieren

[Variante:Indirekte Verkettung]

Beispiel $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Schlüssel 12 , 55 , 5 , 15 , 2 , 19 , 43
Indirekte Verkettung der Überläufer



Tabellenvergrösserung

- Wir wissen nicht a priori, wie gross n sein wird
- Wir möchten $m = \Theta(n)$ zu jeder Zeit (Tabellengrösse m linear abhängig von der Anzahl Einträge n, d.h. nicht beliebig gross)

Grösse der Tabelle anpassen $\rightarrow$ Hashfunktion ändert sich $\rightarrow$ *Rehashing*

- Alloziere Array A' mit Grösse $m' > m$
- Füge jeden Eintrag von A erneut in A' ein (mit erneutem Hashing)
- Setze $A \leftarrow A'$
- Kosten: $\Theta(n + m + m')$

Wie wählt man m'?

Tabellenvergrösserung

In Abhängigkeit vom Belegungsfaktor Tabellengrösse jeweils verdoppeln.
$\Rightarrow$ Amortisierte Analyse ergibt: Jede Operation des Hashings mit Verketten hat erwartet amortisierte Kosten $\Theta(1)$.

Offene Addressierung

Speichere die Überläufer direkt in der Hashtabelle mit einer *Sondierungsfunktion* $s : \mathcal{K} \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$
Tabellenposition des Schlüssels entlang der *Sondierungsfolge*

$$S(k) := (s(k, 0), s(k, 1), \ldots, s(k, m-1)) \mod m$$

Sondierungsfolge muss für jedes $k \in \mathcal{K}$ eine Permutation sein von $\{0, 1, \ldots, m-1\}$

Begriffsklärung: Dieses Verfahren nutzt *offene Addressierung* (Positionen in der Hashtabelle nicht fixiert), ist aber trotzdem ein *geschlossenes Hashverfahren* (Einträge bleiben in der Hashtabelle).

Algorithmen zur offenen Addressierung

Es sei H eine Hashtabelle (ohne Überlauflisten).

- **`insert`**(H, i) Suche Schlüssel k von i in der Tabelle gemäss Sondierungssequenz $S(k)$. Ist k nicht vorhanden, füge k an die erste freie Position in der Sondierungsfolge ein. Andernfalls Fehlermeldung.
- **`find`**(H, k) Durchlaufe Tabelleneinträge gemäss $S(k)$. Wird k gefunden, gib die zu k gehörenden Daten zurück. Andernfalls Rückgabe eines leeres Elements **`null`**.
- **`delete`**(H, k) Suche k in der Tabelle gemäss $S(k)$. Wenn k gefunden, ersetze k durch den speziellen Schlüssel **`removed`**.

Lineares Sondieren

$$s(k,j) = h(k) + j \Rightarrow S(k) = (h(k), h(k)+1, \ldots, h(k)+m-1) \mod m$$

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Schlüssel 12 , 55 , 5 , 15 , 2 , 19

[Analyse Lineares Sondieren (ohne Herleitung)]

1. Erfolglose Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

2. Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right).$$

Diskussion

Beispiel $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 200 Tabelleneinträge!
(Hier ohne Herleitung.).

Grund für die schlechte Performance?

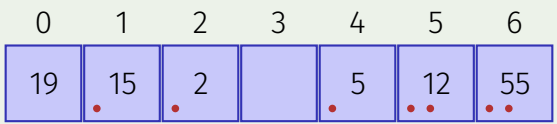
Primäre Häufung: Ähnliche Hashaddressen haben ähnliche Sondierungsfolgen $\Rightarrow$ lange zusammenhängende belegte Bereiche.

Quadratisches Sondieren

$$s(k,j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

$m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Schlüssel 12 , 55 , 5 , 15 , 2 , 19



[Analyse Quadratisches Sondieren (ohne Herleitung)]

1. Erfolglose Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

2. Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C_n \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}.$$

Diskussion

Beispiel $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 22 Tabelleneinträge
(Hier ohne Herleitung.)

Grund für die schlechte Performance?

Sekundäre Häufung: Synonyme k und k' (mit $h(k) = h(k')$) durchlaufen dieselbe Sondierungsfolge.

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(k,j) = h(k) + j \cdot h'(k)$.
$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Schlüssel 12 , 55 , 5 , 15 , 2 , 19

Double Hashing

- Sondierungsreihenfolge muss Permutation aller Hashadressen bilden. Also $h'(k) \neq 0$ und $h'(k)$ darf m nicht teilen, z.B. garantiert mit m prim.
- h' sollte möglichst unabhängig von h sein (Vermeidung sekundärer Häufung).

Unabhängigkeit:

$$\mathbb{P}\big((h(k) = h(k')) \ \wedge \ (h'(k) = h'(k'))\big) = \mathbb{P}\big(h(k) = h(k')\big) \cdot \mathbb{P}\big(h'(k) = h'(k')\big).$$

Unabhängigkeit weitgehend erfüllt von $h(k) = k \bmod m$ und $h'(k) = 1 + k \bmod (m-2)$ (m prim).

[Analyse Double Hashing]

Sind h und h' unabhängig, dann:

1. Erfolglose Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C'_n \approx \frac{1}{1-\alpha}$$

2. Erfolgreiche Suche. Durchschnittliche Anzahl betrachteter Einträge

$$C_n \approx \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

Gleichmässiges Hashing

Starke Annahme: Die Sondierungssequenz $S(k)$ eines Schlüssels k ist mit gleicher Wahrscheinlichkeit eine der $m!$ vielen Permutationssequenzen von $\{0, 1, \ldots, m-1\}$.

(Double Hashing kommt dem am ehesten nahe)

Analyse gleichmässiges Hashing mit offener Addressierung

Theorem 18

Sei eine Hashtabelle mit offener Addressierung gefüllt mit Füllgrad $\alpha = \frac{n}{m} < 1$. *Unter der Annahme vom gleichmässigen Hashing hat die nächste Operation erwartete Laufzeitkosten von* $\leq \frac{1}{1-\alpha}$.

Analyse: Beweis des Theorems

Zufallsvariable X: Anzahl Sondierungen bei einer erfolglosen Suche.

$$\mathbb{P}(X \geq i) \overset{*}{=} \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}$$
$$\overset{**}{\leq} \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}. \qquad (1 \leq i \leq m)$$

* : Ereignis A_j: Slot beim j-ten Schritt belegt.
$\mathbb{P}(A_1 \cap \cdots \cap A_{i-1}) = \mathbb{P}(A_1) \cdot \mathbb{P}(A_2|A_1) \cdot \ldots \cdot \mathbb{P}(A_{i-1}|A_1 \cap \cdots \cap A_{i-2})$,

** : $\frac{n-1}{m-1} < \frac{n}{m}$ da $n < m$: $\frac{n-1}{m-1} < \frac{n}{m} \Leftrightarrow \frac{n-1}{n} < \frac{m-1}{m} \Leftrightarrow 1 - \frac{1}{n} < 1 - \frac{1}{m} \Leftrightarrow n < m$
$(n > 0, m > 0)$

Ausserdem $\mathbb{P}(x \geq i) = 0$ für $i \geq m$. Also

$$\mathbb{E}(X) \overset{\text{Anhang}}{=} \sum_{i=1}^{\infty} \mathbb{P}(X \geq i) \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

[Erfolgreiche Suche beim gleichmässigen offenen Hashing]

Theorem 19

Sei eine Hashtabelle mit offener Addressierung gefüllt mit Füllgrad $\alpha = \frac{n}{m} < 1$. *Unter der Annahme vom gleichmässigen Hashing hat die erfolgreiche Suche erwartete Laufzeitkosten von* $\leq \frac{1}{\alpha} \cdot \log \frac{1}{1-\alpha}$.

Beweis: Cormen et al, Kap. 11.4

Übersicht

	$\alpha = 0.50$		$\alpha = 0.90$		$\alpha = 0.95$	
	C_n	C'_n	C_n	C'_n	C_n	C'_n
(Direkte) Verkettung	1.25	0.50	1.45	0.90	1.48	0.95
Lineares Sondieren	1.50	2.50	5.50	50.50	10.50	200.50
Quadratisches Sondieren	1.44	2.19	2.85	11.40	3.52	22.05
Gleichmässiges Hashing	1.39	2.00	2.56	10.00	3.15	20.00

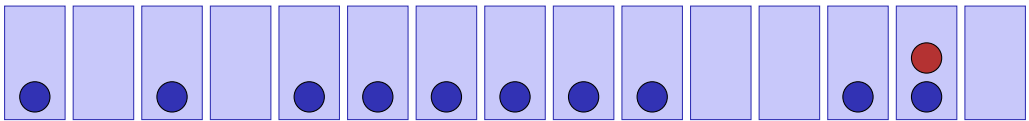
α: Belegungsgrad.
C_n: Anzahl Schritte erfolgreiche Suche,
C'_n: Anzahl Schritte erfolglose Suche

14.8 Anhang

Mathematische Formeln

[Geburtstagsparadoxon]

Annahme: m Urnen, n Kugeln (oBdA $n \leq m$).
n Kugeln werden gleichverteilt in Urnen gelegt.



Wie gross ist die Kollisionswahrscheinlichkeit?

Geburtstagsparadoxon: Bei wie vielen Personen (n) ist die Wahrscheinlichkeit, dass zwei am selben Tag ($m = 365$) Geburtstag haben grösser als 50%?

[Geburtstagsparadoxon]

$\mathbb{P}(\text{keine Kollision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^m}$.

Sei $a \ll m$. Mit $e^x = 1 + x + \frac{x^2}{2!} + \ldots$ approximiere $1 - \frac{a}{m} \approx e^{-\frac{a}{m}}$. Damit:

$$1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \ldots \cdot \left(1 - \frac{n-1}{m}\right) \approx e^{-\frac{1+\cdots+n-1}{m}} = e^{-\frac{n(n-1)}{2m}}.$$

Es ergibt sich

$$\mathbb{P}(\text{Kollision}) = 1 - e^{-\frac{n(n-1)}{2m}}.$$

Auflösung zum Geburtstagsparadoxon: Bei 23 Leuten ist die Wahrscheinlichkeit für Geburstagskollision 50.7%. Zahl stammt von der leicht besseren Approximation via Stirling Formel. $n! \approx \sqrt{2\pi n} \cdot n^n \cdot e^{-n}$

[Erwartungswertformel]

$X \geq 0$ diskrete Zufallsvariable mit $\mathbb{E}(X) < \infty$

$$\begin{aligned}\mathbb{E}(X) &\overset{(def)}{=} \sum_{x=0}^{\infty} x\mathbb{P}(X = x) \\ &\overset{\text{Aufzählen}}{=} \sum_{x=1}^{\infty} \sum_{y=x}^{\infty} \mathbb{P}(X = y) \\ &= \sum_{x=0}^{\infty} \mathbb{P}(X > x)\end{aligned}$$