Optimierungen für optimistische Szenarios

Wer mit dem Schlimmsten rechnet, wird positiv überrascht…

Das worst-case Szenario ist nicht die einzige betrachtenswerte Situation!

In der Lage zu sein, alle Szenarien zu betrachten, ist eine wichtige Fähigkeit, um den adäquaten Algorithmus für jede Situation auswählen zu können.

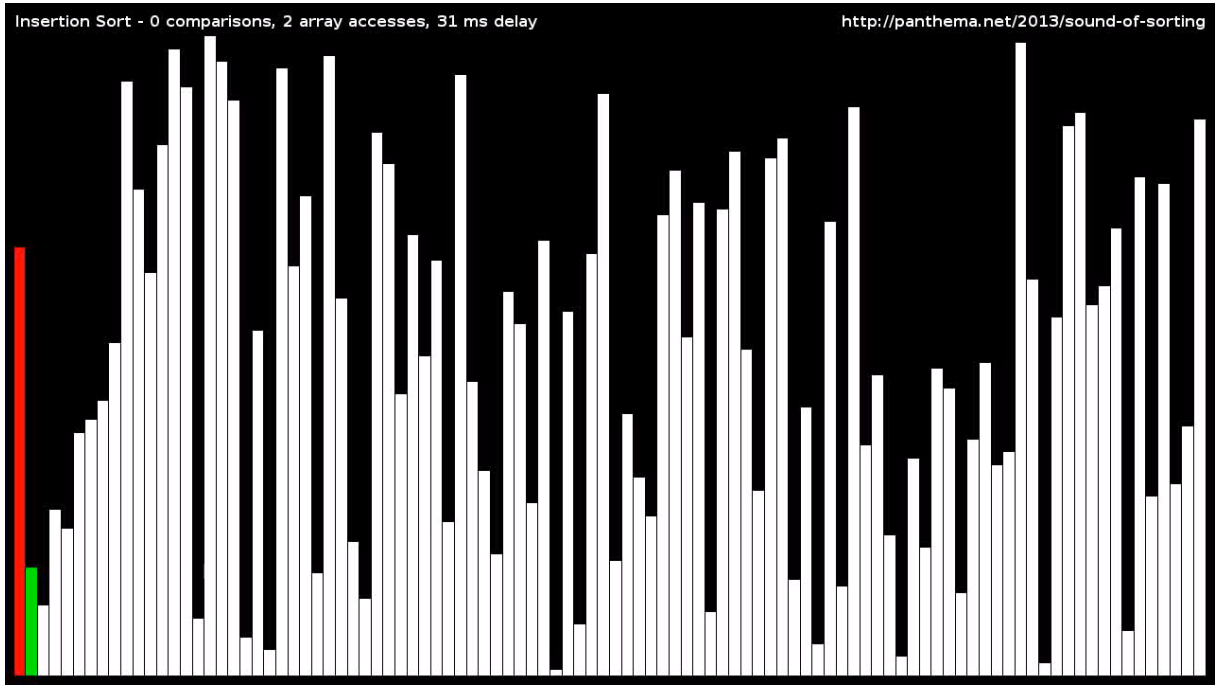
Insertion sort: Sortieren durch Einfügen

Idee:

Karten werden nacheinander in beliebiger Reihenfolge einzeln vom Tisch in die (sortierte) Hand genommen und jeweils an der richtigen Stelle (die zwischen den sortierten Karten gesucht werden muss) eingefügt

Oder:

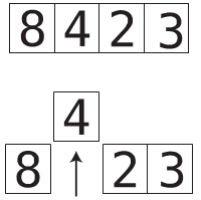
Ich nehme das nächste Buch aus dem Regal und füge es korrekt zwischen den bereits sortierten Büchern ein.



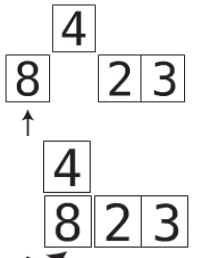
http://panthema.net/2013/sound-of-sorting

Insertion sort

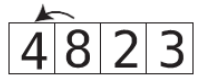
1. Im ersten Durchgang entfernen wir temporär den Wert am Index 1 (2. Zelle) und speichern ihn in einer temporären Variable. Das erzeugt eine Lücke an diesem Index, weil er keinen Wert mehr enthält.
   In den folgenden Durchgängen entfernen wir die Werte der folgendes Indizes.



2. Dann beginnen wir eine Schiebephase, in der wir jeden Wert auf der linken Seite der Lücke bewegen, nachdem wir ihn mit dem Wert der temporären Variable verglichen haben:
   Wenn der Wert links von der Lücke größer als die temporäre Variable ist, schieben wir den Wert nach rechts:
   Indem wir Werte nach rechts schieben, wandert die Lücke nach links. Sobald wir auf einen Wert treffen, der niedriger als der temporär entfernte Wert ist, oder das linke Ende des Arrays erreichen, endet die Schiebephase



3. Wir fügen dann den temporär entfernten Wert in die aktuelle Lücke ein:



4. Wir wiederholen Schritt 1-3 bis das Array vollständig sortiert ist.

Fügt das nächste Element der Restmenge (Karten auf dem Tisch) an die richtige Stelle der bereits sortierten Menge ein.

Insertion sort: Implementierung

```
void insertion_sort (int i_array[], int i_size)
{
    int position=0, temp_value;
    for (int i=1; i<i_size; i++)   // ab dem 2. Arrayelement (erstes ist schon auf der Hand) jedes Element
    {                              // der Restmenge  zur Prüfung/ Einsortierung entnehmen
        position = i;              // zählt von i runter -> 0
        temp_value = i_array[i];   // Lücke bei i erzeugt
        while ((position > 0) && (i_array[position - 1] > temp_value))
        {
            i_array[position] = i_array[position - 1];  // Element nach rechts in Lücke schieben
            position = position - 1;
        }
        i_array[position] = temp_value;    // einzusortierendes Element in entstandene Lücke einsortieren
    }
}
```

Insertion sort: Effizienz

Insertion sort enthält vier verschiedene Typen von Schritten:
Zwischenspeicherung, Vergleiche, Verschiebungen, und Einfügungen.

Vergleiche: jeden Temp-Wert mit jedem Wert links davon: 1+… +n-1 worst case: $N^2/2$

Verschiebungen: worst case: wie Vergleiche: $N^2/2$

Zwischenspeicherung und Einfügen: je 1x pro Durchlauf: N-1 Durchläufe = 2N -2 Schritte

Summe: $N^2/2 + N^2/2 + 2N - 2 = O(N^2 + 2N - 2)$ worst case

Big-O hat jedoch noch eine wichtige Regel:
Die Big-O-Notation betrachtet bei mehreren addierten Ordnungen nur die höchste Ordnung von N!

Insertion sort: $O(N^2)$ Gleiche Komplexität wie Bubble sort und Selection sort bzgl. worst case!

Insertion sort: Effizienz

Für einen Algorithmus, der $N^4 + N^3 + N^2 + N$ Schritte benötigt, sehen wir nur N^4 als signifikant an — und nennen ihn $O(N^4)$.

Warum:

N	N^2	N^3	N^4
2	4	8	16
5	25	125	625
10	100	1,000	10,000
100	10,000	1,000,000	100,000,000
1,000	1,000,000	1,000,000,000	1,000,000,000,000

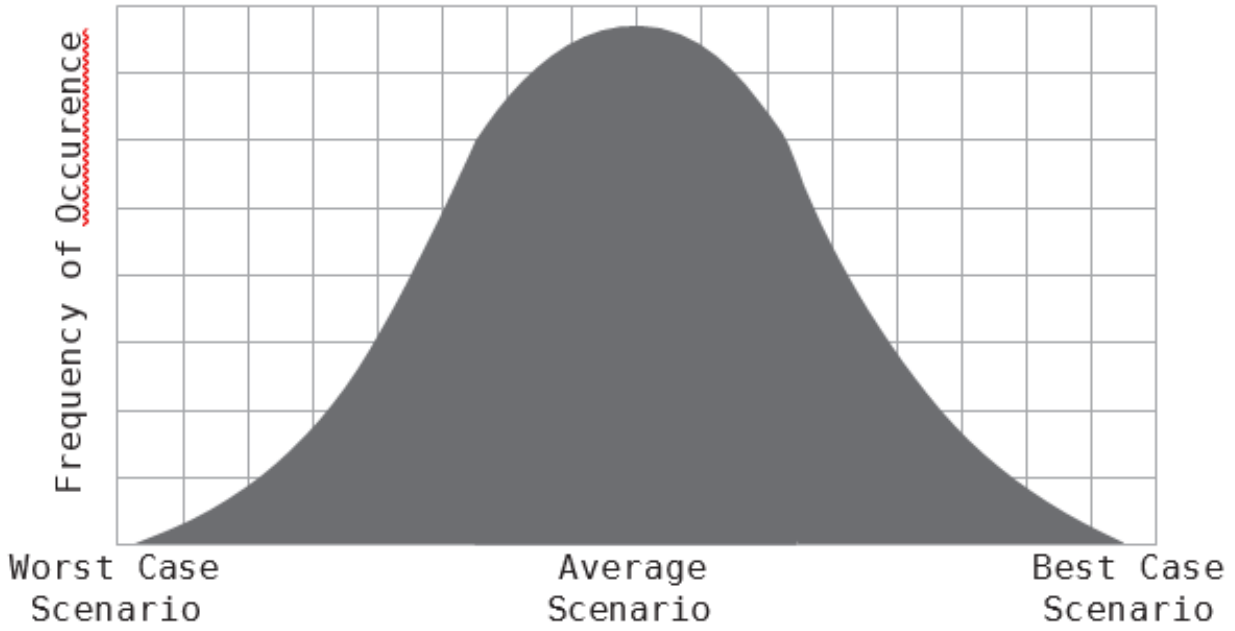
Insertion sort: Effizienz

Schritte im worst case:

Bubble sort:	N^2
Selection Sort (schnellste Suche):	$N^2 / 2$
Insertion Sort:	$N^2 + 2N - 2$

Average case

Die am häufigsten auftretenden Fälle sind Average cases (durchschnittliche Szenarien).
Glockenkurvenverteilung:



Beispiel Bubble sort:

Worst case: Array ist absteigend sortiert
Average case: Array ist unsortiert
Best case: Array ist bereits aufsteigend sortiert

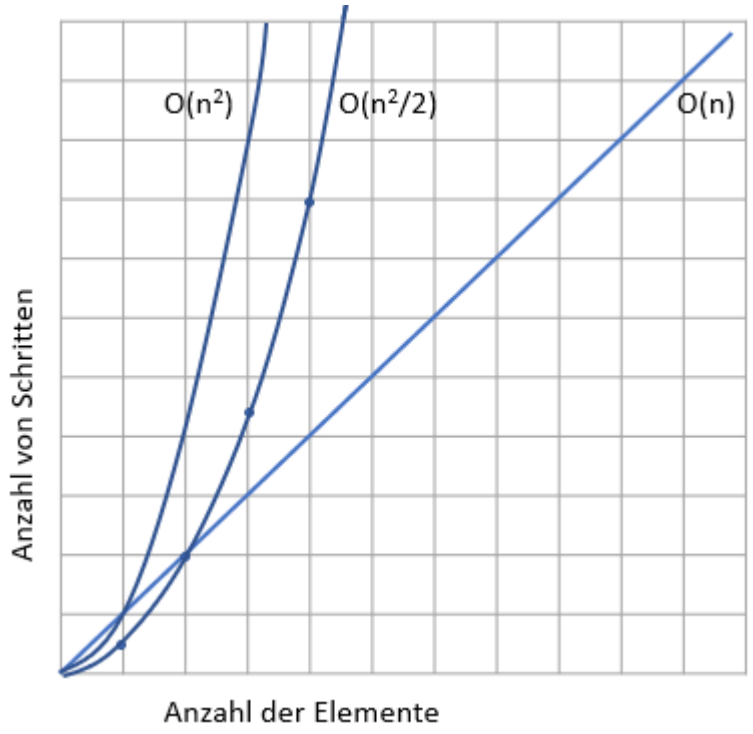
Insertion sort

Worst case: **Vergleichen und Schieben aller Daten:** N^2 Schritte

Best case: **Kein Verschieben, N Vergleiche:** N Schritte
Nur 1 Vergleich pro Durchlauf, und keine einzige Verschiebung, weil jeder Wert bereits an seinem korrekten Platz ist.

Average case:
Wir müssen in den Durchläufen manchmal alle Daten vergleichen und verschieben, manchmal manche, und manchmal keine. Dadurch endet ein grosser Teil der Durchläufe früher.

Vergleichen und Schieben der Hälfte der Daten: $N^2/2$ Schritte (O (N^2))



Insertion sort vs. Selection sort

		Insertion sort	Selection sort
Worst case:	**Vergleichen und Schieben aller Daten**	N^2 Schritte	$N^2/2$ Schritte
Best case:	**Kein Verschieben von Daten, N Vergleiche**	N Schritte	$N^2/2$ Schritte
Average case:	**Vergleichen und Schieben der Hälfte aller Daten**	$N^2/2$ Schritte	$N^2/2$ Schritte

Was ist besser? It depends.

Insertion sort vs. Selection sort

		Insertion sort	Selection sort
Worst case:	**Vergleichen und Schieben aller Daten**	N^2 Schritte	$N^2/2$ Schritte
Best case:	**Kein Verschieben von Daten, N Vergleiche**	N Schritte	$N^2/2$ Schritte
Average case:	**Vergleichen und Schieben der Hälfte aller Daten**	$N^2/2$ Schritte	$N^2/2$ Schritte

- Average case (Array ist **unsortiert**): Performance ist gleich.
- Man kann begründet davon ausgehen, dass die Eingangsdaten **weitgehend sortiert** sind: Insertion Sort ist die bessere Wahl.
- Man kann begründet davon ausgehen, dass die Eingangsdaten **weitgehend absteigend sortiert** sind: Selection Sort ist schneller.
- Man hat **keine Ahnung bzgl. der Sortierung** (wie beim Average case): Beide sind gleich.

Praktisches Beispiel für Optimierung AC/ BC

Nehmen wir an, Sie brauchen irgendwo in Ihrer Applikation die Schnittmenge zweier gleichgroßer Sets.

Die Schnittmenge ist die Menge der Werte, die in beiden Sets vorkommen.

Für die Sets [3, 1, 4, 2] und [4, 5, 3, 6] ist die Schnittmenge das Set [3, 4].

Ansatz: Vergleiche jedes Element der Menge 1 mit jedem Element der Menge 2, und speichere gemeinsame Elemente in Menge 3.

Praktisches Beispiel für Optimierung AC/ BC

Vergleiche jedes Element der Menge 1 mit jedem Element der Menge 2, und speichere gemeinsame Elemente in Menge 3.

```
int Intersection(int firstArray[], int secondArray[], int resultArray[], int size)
{
    int count=0;
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (firstArray[i] == secondArray[j])
            {    // gemeinsames Element gefunden
                resultArray[count] = firstArray[i];
                count++;
            }
        }
      }
    return (count);
}
```

Schritte:

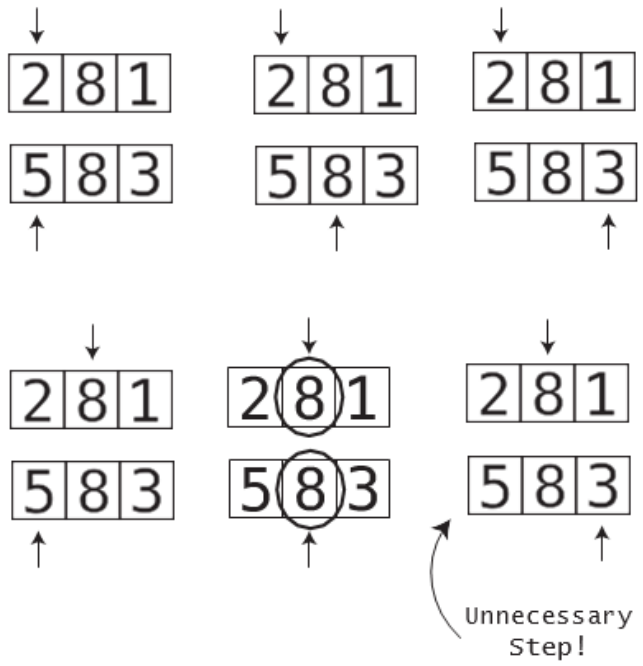
Kopien: M (Anzahl gemeinsamer Elemente)

Vergleiche: NxN = $O(N^2)$ -> $O(N^2+M)$ Verbesserbar?

Praktisches Beispiel für Optimierung AC/ BC

Worst case: Die beiden Arrays haben keine gemeinsamen Werte.
Wir müssen wir jeden Wert beider Arrays prüfen, um festzustellen, dass die Schnittmenge leer ist.

Average case: Die beiden Arrays haben gemeinsame Elemente.
Sobald wir einen gemeinsamen Wert gefunden haben (s.u. die 8), gibt es z.B. keinen Grund mehr, die innere Schleife fortzusetzen, um nach weiteren 8en zu suchen.




```
int Intersection(int firstArray[], int secondArray[], int resultArray[], int size)
{
    int count=0;
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (firstArray[i] == secondArray[j])
            {
                resultArray[count] = firstArray[i];
                count++; // gemeinsames Element gefunden
                break;
            }
        }
    }
    return (count);
}
```

Praktisches Beispiel für Optimierung AC/ BC

Worst case: Auch nach der Optimierung unverändert: **N^2**.

Best case (beiden Arrays sind identisch): nur noch **N** (1+2+…+N) Vergleiche!

Average case (die beiden Arrays teilen manche Werte): Performance wird irgendwo zwischen **N und N^2** liegen.

Zusammenfassung/ Ausblick

Fähigkeit, ein Best-, Average- oder Worst-case Szenario zu erkennen: Schlüsselkompetenz

- für die Auswahl des richtigen Algorithmus für eine Situation, sowie
- für die Geschwindigkeitsoptimierung existierender Algorithmen.

-> Vorbereitet sein auf den worst case – aber Average cases sind doch wesentlich häufiger.

Übung 3

1. Die folgende Funktion prüft, ob ein Zahlenarray ein Zahlenpaar enthält, das zusammen 10 ergibt:

```
bool PairSum(int array[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if ((i != j) && (10 == (array[i] + array[j]))) return true;
        }
    }
    return false;
}
```

Welche Komplexität hat das Worst-case-Szenario in Big-O-Notation?

A: O (1)
B: O (log N)
C: O (N)
D: O (N * log N)
E: O (N^2)

Übung 3

2. Die folgende Funktion prüft, ob ein Zahlenarray ein Zahlenpaar enthält, das zusammen 10 ergibt:

```
bool PairSum(int array[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if ((i != j) && (10 == (array[i] + array[j]))) return true;
        }
    }
    return false;
}
```

Welches dieser Szenarien ist der Worst case?

A: Die ersten beiden Zahlen ergeben zusammen 10.
B: Kein Zahlenpaar ergibt 10.
C: Irgendein Zahlenpaar ergibt 10.

Übung 3

3. Die folgende Funktion prüft, ob ein Zahlenarray ein Zahlenpaar enthält, das zusammen 10 ergibt:

```
bool PairSum(int array[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if ((i != j) && (10 == (array[i] + array[j]))) return true;
        }
    }
    return false;
}
```

Welches dieser Szenarien ist der Best case?

A: Die ersten beiden Zahlen ergeben zusammen 10.
B: Kein Zahlenpaar ergibt 10.
C: Irgendein Zahlenpaar ergibt 10.

Übung 4

Benutzen Sie die Big-O-Notation, um die Effizienz eines Algorithmus zu beschreiben, der $3N^2 + 2N + 1$ Schritte benötigt.

A: O (1)
B: O (logN)
C: O (N)
D: O (N^2)

Übung 5

Benutzen Sie die Big-O-Notation, um die Effizienz eines Algorithmus zu beschreiben, der N + log N Schritte benötigt.

A: O (1)
B: O (logN)
C: O (N)
D: O (N^2)

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Übung 6

Benutzen Sie die Big-O-Notation, um die Effizienz der folgenden Funktion zu beschreiben.

Sie mischt 2 sortierte Arrays unterschied-licher Länge ineinander, um ein neues sortiertes Array zu erzeugen, das alle Werte der beiden Arrays enthält:

```
void merge(int array1[], int array2[], int array3[], int size1, int size2)
{
    int index1=0, index2=0, index3=0;
    // Run the loop until we've reached end of both arrays:
    while (index1 < size1 || index2 < size2)
    {
        if (index1 >= size1)
        {  // If we already reached the end of the 1st array, add item from 2nd array:
            array3[index3] = array2[index2]; index2++;
        }
        else
        {  // If we already reached the end of the 2nd array, add item from 1st array:
            if (index2 >= size2)
            {
                array3[index3] = array1[index1]; index1++;
            }
            else
            {  // If the current # in 1st array is less than current # in 2nd array, add from 1st array:
                if (array1[index1] < array2[index2])
                {
                    array3[index3] = array1[index1]; index1++;
                }
                else
                {  // If the current # in 2nd array is less than or equal to current # in 1st array, add from 2nd array:
                    array3[index3] = array2[index2]; index2++;
                }
            }
        }
        index3++;
    }
}
```

A: O (1)
B: O (logN)
C: O (N)
D: O (N^2)

Übung 7

Benutzen Sie die Big-O-Notation, um die Effizienz der folgenden Funktion zu beschreiben.
Diese Funktion findet das größte Produkt dreier Zahlen in einem gegebenen Array:

```
int largest_product (int array[], int size)
{
    int largest_product_so_far = array[0] * array[1] * array[2];
    int i = 0;
    while (i < size)
    {
        int j = i + 1;
        while (j < size)
        {
            int k = j + 1;
            while (k < size)
            {
                if (array[i] * array[j] * array[k] > largest_product_so_far)
                {
                    largest_product_so_far = array[i] * array[j] * array[k];
                }
                k += 1;
            }
            j += 1;
        }
        i += 1;
    }
    return largest_product_so_far;
}
```

A: O (1)
B: O (logN)
C: O (N)
D: O (N^2)
E: O (N^3)

Übung 8

Wenn wir eine Software schreiben müssten, die einen Stapel Bewerbungen reduziert bis nur noch eine übrig ist, wäre ein möglicher Ansatz, im Wechsel die oberste und unterste Hälfte des Stapels zu entsorgen, bis nur noch 1 Bewerbung übrig ist…
Benutzen Sie die Big-O-Notation, um die Effizienz dieser Funktion zu beschreiben.

```
char * pick_resume (char *resumes[], int amount)
{
    char *eliminate ="top";
    while (amount >1)
    {
        if ("top" == eliminate)
        {
            resumes +=amount/2;
            eliminate="bottom";
        }
        else
        {
            eliminate="top";
        }
        amount=amount/2;
    }
    return (resumes[0]);
}
```

A: O (1)
B: O (logN)
C: O (N)
D: O (N^2)
E: O (N^3)

Schnelles Lookup mit Hash-Tabellen

Sie wollen ein Programm schreiben:

- erlaubt Online-Bestellungen bei einem Fast-Food-Restaurant
- bietet eine Speisekarte mit den entsprechenden Preisen an

Technisch möglich: Array von structs

```
struct menue {char *dish; float price;};
struct menue dailyMenu []= {{"french fries", 0.75}, {"hamburger", 2.5}, {"hot dog", 1.5}, {"soda", 0.6}};
```

Array ist unsortiert: **Suche nach dem Preis eines bestimmten Gerichts** benötigt **O(N) Schritte** für **lineare Suche**

Array ist (nach den Namen der Gerichte) sortiert: **Suche** benötigt nur **O(log N) Schritte** für **binäre Suche**

Obwohl O(log N) nicht schlecht ist, können wir das viel besser:

Hash tables können das **Lookup in O(1)** Schritten durchführen!

Schnelles Lookup mit Hash-Tabellen

Die meisten modernen Programmiersprachen unterstützen die Datenstruktur *Hash-Tabelle*,
besondere Superpower: **schnelles Nachschlagen ("Lookup").**

Hash-Tabelle ist eine Liste geordneter Paare aus: Schlüssel und Wert (Key-value-pairs, z.B. Gericht und Preis).

Andere Namen für "Hashtabelle" sind:

- Hashes (Ruby),
- Maps (JavaScript),
- Hash maps (Java),
- Dictionaries (C#, Python), und
- Associative containers (C++)

Einen Wert in einer Hash-Tabelle nachzuschlagen, hat im Schnitt eine Effizienz von O(1).

Hashing mit Hash-Tabellen

Haben Sie auch als Kind Geheimcodes verwendet, um Nachrichten zu ver- und entschlüsseln?

Hier z.B. ein einfacher Weg, Buchstaben auf Zahlen zu mappen:

A = 1
B = 2
C = 3
D = 4
E = 5
und so weiter...

Dieser Code konvertiert: ACE in 135, CAB in 312, DAB in 412, und BAD in 214.

Der Vorgang, Zeichen oder große Zahlen in kleinere Zahlen zu konvertieren, wird **Hashing** genannt.

Und die Vorschrift, nach der diese Zeichen in bestimmte Ziffern konvertiert werden, wird **Hash-Funktion** genannt.

Hashing mit Hash-Tabellen

1. Beispiel einer Hash-Funktion: die korrespondierende Stellenzahl jedes Buchstaben nehmen und die *Summe* aller Ziffern bilden (Zerlegungsmethode).

BAD = 214 2+1+4 = 7

2. Beispiel einer Hash-Funktion: Bildung des *Produkts* der Stellenzahlen jeden Buchstabens:

BAD = 214 2*1*4 = 8

Um valide zu sein, muss eine Hash-Funktion nur ein Kriterium erfüllen:
Eine Hash-Funktion muss den gleichen String (Key) bei jeder Benutzung in die gleiche Zahl verwandeln.

Anmerkung:
Diese Hash-Funktion würde DAB genauso in die Zahl 8 konvertieren wie BAD. Dieses Probleme wird später adressiert.

Komplexere Hash-Funktionen

Divisionsrestmethode: h(x) = x mod Hashgröße

Mitt-Quadrat-Methode: x wird **quadriert** und anschließend **aus der Mitte des Quadrats bestimmte Ziffern** als Hashwert entnommen

Beispiel: x=121487

x2= 1475**9091**169 H(121487) = 9091

H(0) = 0000 000

…

H(31) = 0000 961

Zerlegungsmethode: Die Ziffern des zu verschlüsselnden Wertes werden in Blöcke geteilt. Die Addition (oder andere math. Operation) der Teile bis zur gültigen Adresse ergibt den Hash-Wert.

Beispiel: H(121487) = 121 + 487 = 608

Hash-Funktionen

Hauptproblem: Kollisionen

Einen Thesaurus bauen - aus Spaß und für Profit

Sie entwickeln eine Thesaurus App. Aber keine altmodische — sondern einen *Quickasaurus*.

Benutzer schlägt ein Wort in Quickasaurus nach
-> dieser liefert das populärste Synonym dazu, statt *jedes mögliche* Synonym

Eine Hash-Tabelle speichert ihre Daten in sequentiellen Zellen wie ein Array.
Jede Zelle hat eine korrespondierende Nummer, z.B.:

Einen Thesaurus bauen - aus Spaß und für Profit

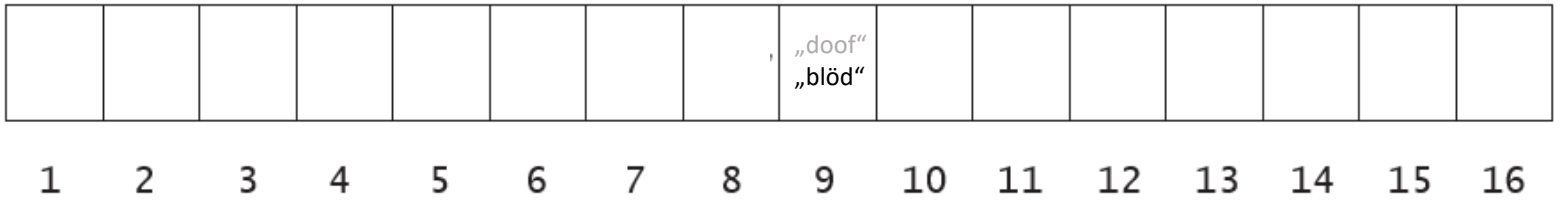
Wir fügen unseren ersten Eintrag in die Hash-Tabelle ein: Key *“doof"*: Value *“blöd"*

Im Speicher sieht unsere Hash-Tabelle jetzt so aus: {*“doof"* => *“blöd"*}

Wie speichert die Hash-Tabelle die Daten?
Zuerst wendet der Computer die Hash-Funktion auf den Schlüssel (Key) an.
Wir verwenden hier die Multiplikations-Hash-Funktion und bilden eine einstellige Quersumme:

DOOF = 4 * 15 * 15 * 6 = 5400 -> 9

Weil unser Schlüssel (“doof") den Hash 9 ergibt, plaziert der Rechner den Wert (“blöd") in die Zelle 9:

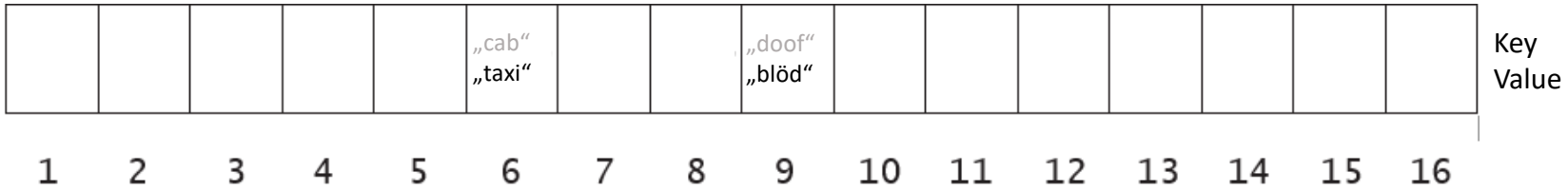


Einen Thesaurus bauen - aus Spaß und für Profit

Jetzt fügen wir ein anderes Key-Value-Paar hinzu: Key *“cab”: Value "taxi“*

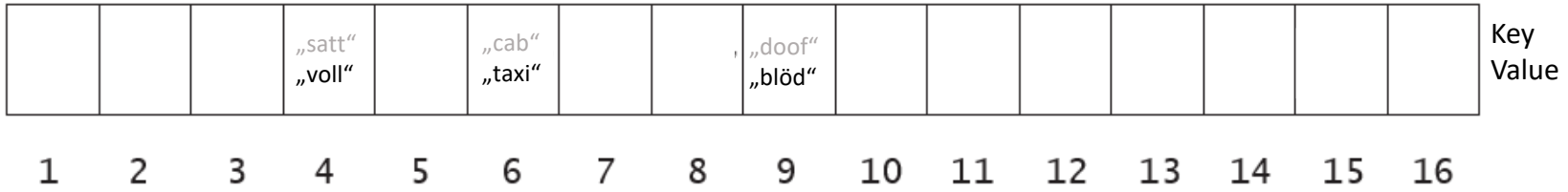
Der Rechner hashed den Schlüssel: CAB = 3 * 1 * 2 = 6

Weil das Ergebnis 6 ist, speichert der Rechner den Wert ("taxi") in die Zelle 6.



Fügen wir noch ein Key-Value-Paar hinzu:
Key *“satt“*: Value *“voll"*

SATT ergibt den Hash-Wert 4, da SATT = 19 * 1 * 20 * 20 = 7600 -> 4 ist, so wird “voll" in die Zelle 4 plaziert:



Einen Thesaurus bauen - aus Spaß und für Profit

Im Speicher sieht die Hash-Tabelle jetzt so aus: {*“doof" => “blöd", "cab" => "taxi", “satt" => “voll"*}

Nachdem wir jetzt unsere Hash-Tabelle aufgebaut haben, können wir uns das Lookup von Werten ansehen.
Wir wollen den Wert zum Key “doof“ ansehen. Im Code würden wir sagen: Value (Key *“doof“*)

Der Rechner führt 2 einfache Schritte durch:

- Er bildet den Hash-Wert des Schlüssels: DOOF = 4 * 15 * 15 * 6 = 5400 -> 9
- Da das Ergebnis 9 ist, schaut der Rechner in die Zelle 9 und gibt (bei Übereinstimmung des Keys) den dort gespeicherten Wert zurück (Value [9]). in diesem Fall würde das der String “blöd“ sein.

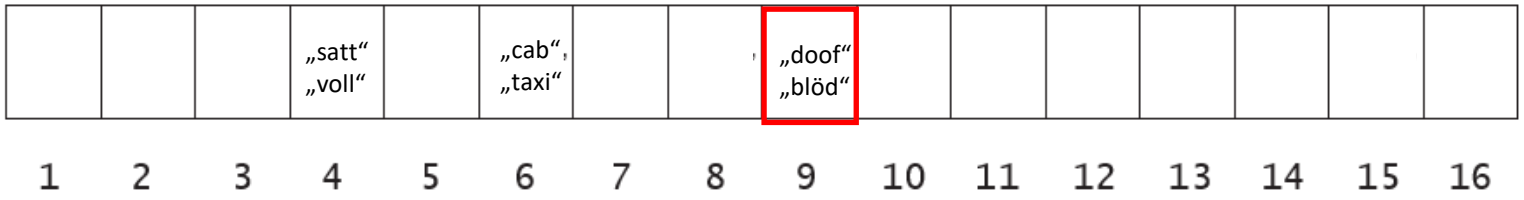
Ein einfacher Array-Zugriff kostet O(1).

Die Abbildung eines Suchbegriffs auf einen Index ist allerdings nicht zwangsläufig eineindeutig!

Behandlung von Kollisionen

Wir setzen unser Thesaurus-Beispiel fort. Was passiert, wenn wir den folgenden Eintrag hinzufügen:
Key *“kurz“*: Value *“klein"*

Der Rechner bildet den Hash-Wert des Schlüssels:
KURZ = 11 * 21 * 18 * 26 = 108108 -> 9



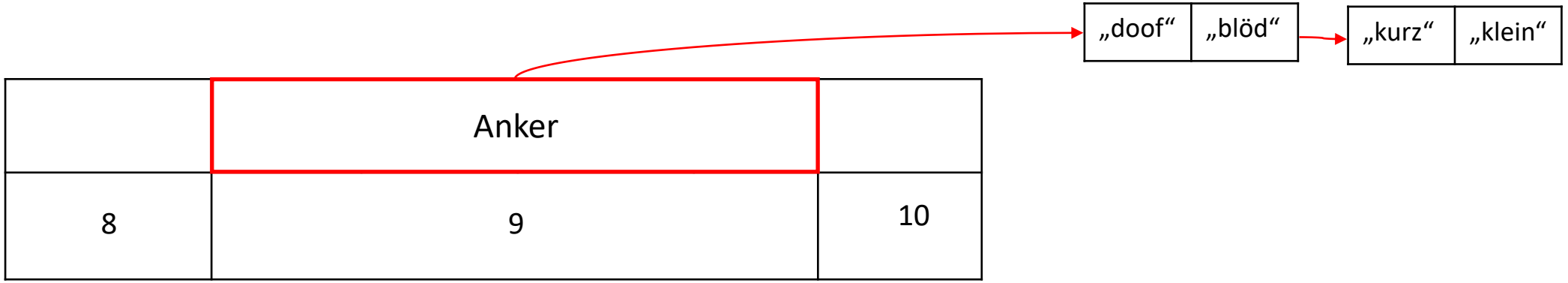
Oops. Zelle 9 enthält bereits den Wert “blöd"— wörtlich!

Den Versuch, Daten in eine bereits belegte Zelle zu füllen, nennt man eine *Kollision*.

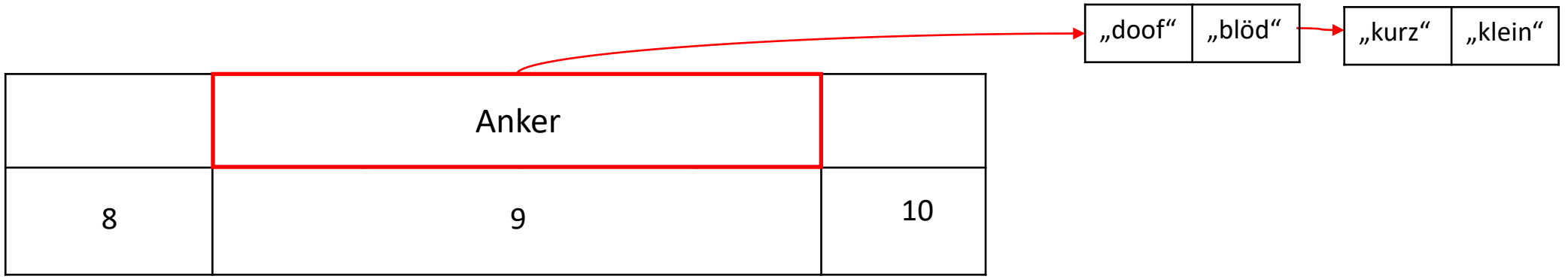
Behandlung von Kollisionen: 1) Chaining

Klassischer Ansatz, Kollisionen zu behandeln: *Verkettung (auch “separate chaining“).*
-> statt des einzelnen Wertes wird ein Anker auf eine separate Liste in die Zelle abgelegt.

Wir möchten “klein" in die Zelle 9 ablegen, aber diese enthält schon “blöd“ -> wir ersetzen den Wert in Zelle 9 mit dem Anker auf eine Liste:



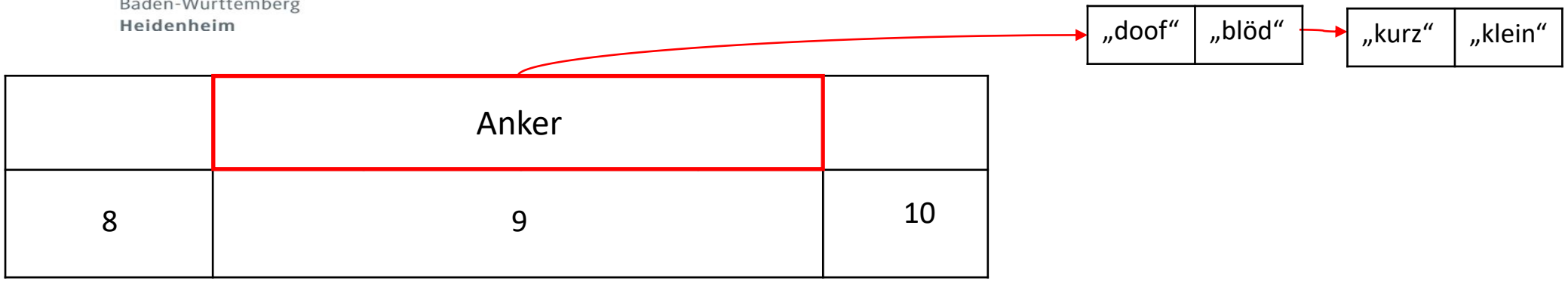
Behandlung von Kollisionen: Suche



Wenn wir Value [Key *“kurz"*] nachschlagen, führt der Rechner folgende Schritte aus:

- Er bildet den Hash-Wert aus dem Schlüssel: KURZ = 11 * 21 * 18 * 26 = 108108 -> 9.
- Er sieht in Zelle 9 nach. Er bemerkt, dass Zelle 9 einen Anker statt einem Datenwert enthält.
- Er sucht die Liste ab dem Anker sequentiell durch nach dem Schlüssel “kurz”, und gibt den dort gespeicherten Wert “klein” zurück.

Behandlung von Kollisionen



Sollten alle Daten in einer einzigen Zelle der Hash-Tabelle landen:

-> Performance wäre nicht mehr besser als die einer Liste.

→ **Worst-case performance** des Hash-Tabellen-Lookup ist nur **O(N).**

Designziel von Hash-Tabellen: wenig Kollisionen, und eher mit O(1) statt mit O(N) performen.

Wir können bei Hash-Tabellen von einer O(1)-Performance ausgehen.

Behandlung von Kollisionen: 2) Sondierung

Verfahren, welche andere Plätze benutzen (Sondieren):

- Hash-Wert ist schon belegt: nächster freier Hash wird gesucht und die Daten dort gespeichert.

- Lineare Sondierung:
  erhöht Index bei belegten Hash-Werten so lange um x mod Anzahl Plätze, bis ein freier Platz gefunden wurde.
- Nachteil: Suche ist sehr aufwendig (worst case: Abarbeitung des gesamten Arrays gemäß Sondierungskriterium)

Beispiel:
Hash-Funktion: Bildung der Quersumme, bis Index einstellig ist

Beispielwerte:	1	5	8	12	16	20	44	93	7	Keys
	1	5	8	3	7	2	8	3	7	Hash (Index)
	a	b	c	d	a	b	c	d	a	Values

Lineare Sondierung mit x=1:

Hash	0	1	2	3	4	5	6	7	8	9
K/V	7 a	1 a	20 b	12 d	93 d	5 b		16 a	8 c	44 c

Vgl.: Chaining (Verkettung):

0	1	2	3	4	5	6	7	8	9
	1 a	20 b	12 d		5 b		16 a	8 c	
			93 d				7 a	44 c	

Offenes bzw. geschlossenes Hashing

manchmal auch als „geschlossen“ bzgl. der Indexposition bezeichnet

1. Offenes Hashing

Jeder Behälter kann beliebig viele Elemente aufnehmen. Für jeden Behälter wird eine verkettete Liste angelegt, in die alle Schlüssel eingefügt werden, die auf diesen Behälter abgebildet werden (Chaining).

manchmal auch als „offen“ bzgl. der Indexposition bzw. der Nutzung offener Felder bezeichnet

2. Geschlossenes Hashing

Hier darf jeder Behälter nur eine konstante Anzahl b>=1 von Schlüsseln aufnehmen (Sondierung).

Vermeidung von Kollisionen

Wie setzen viele Programmiersprachen ihre Hashes auf, um hochfrequente Kollisionen zu vermeiden?

Die Effizienz von Hash-Tabellen hängt von 3 Kriterien ab:

- wieviele Daten werden darin gespeichert
- wieviele Zellen hat die Hash-Tabelle
- **welche Hash-Funktion wird benutzt**

Die Effizienz von Hash-Tabellen steigt, je stärker die Zahl der Kollisionen sinkt.

Der beste Weg, Kollisionen zu vermeiden: Hash-Tabelle mit einer großen Anzahl von Zellen.
Aber: Balancieren gegen unnötige Speicher-Allokierung.

Eine gute Hash-Tabelle findet ein Gleichgewicht zwischen der Vermeidung von Kollisionen und unnötiger Speicherkonsumierung.

Der große Balanceakt

Folgendes Verhältnis von Elementen zu Tabellengröße hat sich bewährt:

Für je 7 abzuspeichernde Elemente sollte eine Hash-Tabelle 10 Zellen haben.

Verhältnis von Daten zu Zellen: ***Lastfaktor***. Idealer Lastfaktor: ca. 0.7-0.8 (7-8 Elemente / 10 Zellen).

Initial 7 Daten in einer Hash-Tabelle: Hash-Tabelle mit 10 Zellen allokieren.

Mehr Daten werden hinzugefügt:

- Bibliotheksfunktion sollte der Hash-Tabelle mehr Zellen hinzufügen und
- die Hash-Funktion wechseln, so dass die gesamten Daten gleichmäßig auch über die neuen Zellen verteilt werden.

Der große Balanceakt

Die meisten Interna einer Hash-Tabelle werden durch die benutzte Programmiersprache verwaltet.
(C++: “Associative container” (z.B. std::unordered_map) der Standard library)

Sie entscheidet:

- wie groß die Hash-Tabelle sein muß,
- welche Hash-Funktion verwendet wird und
- wann es Zeit ist, die Hash-Tabelle zu erweitern.

Sie können sich ruhig darauf verlassen, daß Ihre Programmiersprache die Hash-Tabelle für Spitzenleistung optimiert hat.

Hash-Tabellen für Datenorganisation

Manche Daten liegen natürlicherweise gepaart vor:

- Wörterbücher
- Waren und Preise
- Abstimmungen: Kandidat und Stimmenzahl
- Lagerhaltung: Ware und Anzahl
- HTTP Statuscodes: Code und seine Bedeutung

```
switch ( stateCode)
{
    case 200:  return "OK";
    case 301:  return "Moved Permanently";
    case 401:  return "Unauthorized";
    case 404:  return "Not Found";
    case 500:  return "Internal Server Error";
    default:   return "unknown fault";
}
```

Hash-Tabellen für Geschwindigkeit – Index by Value

Hash-Tabellen können eine Implementierung sogar dann schneller machen, wenn die Daten nicht paarweise (sondern nur einzeln) existieren!

Wir wollen z.B. bestimmen, ob ein Array Teil eines anderen ist:

[*1, 2, 3, 4, 5, 6*]
[2, *4, 6*]

Das zweite Array [2, *4, 6*] ist ein Subset des ersten [*1, 2, 3, 4, 5, 6*] , weil jeder Wert des zweiten Arrays im ersten enthalten ist.

Bei den Arrays

[*1, 2, 3, 4, 5, 6*]
[2, *4, 6, 8*]

ist das zweite Array kein Subset des ersten, weil das 2. Array den Wert 8 enthält, der im 1. Array nicht existiert.
Wie könnte eine Funktion aussehen, die zwei Arrays vergleicht und ermittelt, ob eins ein Subset des anderen ist?

Hash-Tabellen für Geschwindigkeit

Erster Ansatz: Geschachtelte Schleifen, $O(N^2)$ (O(N*M))

Zweiter Ansatz: Hash-Tabelle

Erzeugen: leere Hash-Tabelle.

Größe (Anzahl der Indizes): muss den Wertebereich aller Values abdecken (Anzahl = max. Wert – min. Wert).

Algorithmus: Wir iterieren über jeden Wert des größeren Arrays,
und setzen für den Wert am Index ein “true” in die Hash-Tabelle.

Interessant ist:

Wir setzen nur Dummy-Werte in der Hash-Tabelle (in diesem Beispiel “true”).

Was wollen wir erreichen, wenn uns der genaue Wert egal ist, solange er irgendwie “valid” meldet?

Hash-Tabellen für Geschwindigkeit

```
bool IsSubset (int i_array1[], int i_array2[],int size1, int size2)
{
    int *largerArray = i_array2; int *smallerArray = i_array1;
    int largerSize=size2, smallerSize=size1;
    int hashSize=0;
    bool result=true;
    if (size2 < size1) // Pointer to and size of larger and smaller array
    {
        largerArray=i_array1; largerSize= size1;
        smallerArray=i_array2; smallerSize=size2;
    }
    // check maximum value range for hash size
    for (int i=0; i<largerSize; i++)
    {
        if (hashSize < largerArray[i]) hashSize=largerArray[i];
    }
    for (int i=0; i<smallerSize; i++)
    {
        if (hashSize < smallerArray[i]) hashSize=smallerArray[i];
    }
    void *pHashtable=calloc(hashSize,sizeof(int)); // create initialized hash
    if (pHashtable)
    {
        // set exististence of values of larger array to hash as true
        for (int i=0; i<largerSize; i++)
        {
            *(bool *)(pHashtable+largerArray[i])=true;
        }
        // check hash for all values of smaller array
        for(int j=0; j<smallerSize; j++)
        {
            if (true != (*(bool *)(pHashtable+smallerArray[j])))
            {
                result=false;
                break;
            }
        }
        free (pHashtable);
        return (result);
    }
}
```

Hash-Tabellen für Geschwindigkeit

Für das obige Beispiel [*1*, *2*, *3*, *4*, *5*, *6*] erhalten wir folgende Hash-Tabelle:

{*hash[1]:* **true**, *hash[2]:* **true**, *hash[3]:* **true**, *hash[4]:* **true**, *hash[5]:* **true**, *hash[6]:* **true**}

Wieviele Schritte braucht nun dieser Algorithmus? $N + M + N + M = 2N+2M = 4N = O(N)$ statt $O(N^2)$!

Hash tables für Geschwindigkeit

Die Technik, eine Hash-Tabelle “als Inhaltsverzeichnis” zu verwenden
(welche Werte sind enthalten?):

- wird häufig in Algorithmen benutzt, die mehrfache Suche innerhalb von Arrays erfordern
- funktioniert gut, auch ohne “paired data” (nutzt den Vorteil von Indizes als Schlüssel).

Weiteres Anwendungsgebiet von Hashes: Krytographie, Chiffrierung

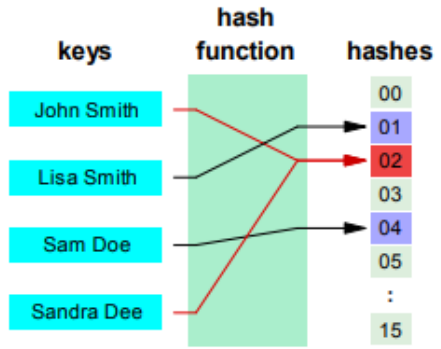
Zusammenfassung

Die Hash-Tabellen sind eine Datenstruktur mit O(1)-Komplexität beim Lesen und Einfügen.

Hashes und Kompression

Kompression dient ausschließlich der Verkleinerung einer Datenmenge.

Abbildung einer großen Eingabemenge auf eine kleinere Ausgabemenge mittels einer Hashfunktion.



Eine Hashfunktion ist ein Algorithmus, der aus einer großen Datenmenge
eine sehr kleine Zusammenfassung/ Identifikation (einen Fingerabdruck) generiert.

Hashwerte sind kleiner als die Eingabedaten, und erreichen dadurch eine gewisse Kompression.

Sehr einfaches Bsp.: Hashfunktion = Quersummierung bis 1-stellig

Datenwort: 4281, Hash: 6
Datenwort: 7294, Hash: 4

Hashes in der Kryptograhie

Kryptographie schützt Informationen durch das Verschlüsseln der Originaldaten in ein unlesbares Format.

Kollisionsresistente Hashfunktion: keine 2 unterschiedliche Eingabewerte führen zum identischen Hashwert

Anwendungsgebiete:

- Integritätsprüfung von Dateien oder Nachrichten,
- die sichere Speicherung von Passwörtern und digitale Signaturen,
- Pseudo-Zufallszahlengeneratoren ,
- Konstruktion von Blockchiffren,
- Generierung von Session-IDs

Hashfunktion, weitere Definition:

- bildet effizient eine Zeichenfolge beliebiger Länge (Eingabewert) auf eine Zeichenfolge mit fester Länge (Hashwert) ab (u.U. nicht eineindeutig).

Konstruktionsstandards:
Bis 2015: Merkle-Damgard-Konstruktion (SHA2), seitdem SHA3 Keccak (Gewinner des SHA-3-Wettbewerbs 2012)

Hashes in der Kryptograhie

Bsp.: Hashen eines Passworts, Hashfunktion: Quersummierung bis 1-stellig.

- Passwort kann nicht wiederhergestellt werden
- gespeicherter Passwort-Hashwert eines Benutzers muss bei der Prüfung mit dem Hashwert des vom Benutzer eingegebenen Passworts übereinstimmen

Passwort: 4281, Quersumme: 15, Hash: 6
Passwort: 7294, Quersumme: 22, Hash: 4

Kriterien für gute kryptografische Hashes:

- Geringe Kollisionswahrscheinlichkeit
- Gleichverteilung der Hashwerte im Hash-Bereich
- Effizienz: schnell berechenbar, geringer Speicherverbrauch, die Eingabedaten nur einmal lesen.
- Jeder Ergebniswert soll möglich sein (Surjektivität).
- Hashwert soll viel kleiner sein als Eingabedaten (zusätzlich gute Kompression)

Hashes für Fehlererkennung und -korrektur

Datenübertragung: **Hashes** werden **als Prüfsummen** verwendet -> Erkennen von Übertragungsfehlern von Daten
(Quersummen bzw. Teile davon, evtl. auch mehrere verschiedene bzw. gestufte…)

Einfache Verfahren: Parity-Bits in Bytes/ Worten (erkennen nur Einfachfehler, keine Korrektur möglich)

CRC: Cyclic redundancy check (mit zusätzl. redundanten Daten), basierend auf Polynomdivision.
Die Eingangsdatenbits werden als binäres Polynom betrachtet und mit einem anderen Polynom multipliziert.

Polynom: Summiert die Vielfachen aller Potenzen einer Variablen oder Konstanten

$$P(x) = \sum_{i=0}^{n} a_i x^i, \quad n \in \mathbb{N}_0.$$

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n, \quad n \in \mathbb{N}_0$$

Binäres Polynom: x und a sind Binärzahlen

Hashes für Fehlererkennung und -korrektur

CRC: Cyclic redundancy check (mit zusätzl. redundanten Daten), basierend auf Polynomdivision.
Die Eingangsdatenbits werden als binäres Polynom betrachtet und mit einem anderen Polynom multipliziert

Die Eingangs-Bitfolge 1,1,0,1,1 entspricht dem Polynom $1*2^4+1*2^3+0*2^2+1*2^1+1*2^0$
Generatorpolynom z.B.: 11010111 ist n=siebten Grades (2^7+... 2^0) (erkennt Bündelfehler der Länge <= n).
Der zu übertragenden Eingangs-Bitfolge werden zunächst n (7) Nullen angehängt: 110110000000
Modulo-Division (XOR) von links her durch das Generatorpolynom (führende Nullen werden ignoriert):

```
110110000000   D8=13*16+8= 216
11010111
------------
000011110000
    11010111
------------
    00100111    Rest der Division
```

An die Nutzdaten werden nun n (7) Bit des Rests angehängt: 110110100111

Hashes für Fehlererkennung und -korrektur

Der Empfänger besitzt ebenfalls das Generatorpolynom (Fehlererkennung – nicht Kryptographie!).

Kommt die Nachricht nach der Übertragung beim Empfänger, kann dieser prüfen, ob sie korrekt ist (erneute Modulo-Division durch das Generatorpolynom):

Korrekt übertragene Nachricht: 110110100111
Generatorpolynom (wie oben): 11010111

```
110110100111
11010111
-------------
000011010111
    11010111
------------
    00000000   kein Fehler aufgetreten!
```

Übungen

Lösen Sie in den folgenden 60 min. mindestens eine der Übungsaufgaben (lauffähig, in einer Sprache Ihrer Wahl), und messen Sie in main(), ob Sie die Zielkomplexität in Abhängigkeit von der Eingangsmenge erreicht haben! Bereiten Sie sich darauf vor, Ihren Kommilitonen die Lösung vorzustellen:

Aufgabe 1: MissingLetter()
Aufgabe 2: Intersection()
Aufgabe 3: SingleChar()

Nachnamen B-Hi: Aufgabe 1, 2, 3
Ho-P: Aufgabe 2, 3, 1
R-W: Aufgabe 3, 1, 2

Übung 1

Schreiben Sie eine Funktion *MissingLetter()*, die unter Nutzung von Hashes für einen String, der alle Buchstaben des englischen Alphabets bis auf einen enthält, den fehlenden Buchstaben zurückgibt.

Der String "the quick brown box jumps over a lazy dog" enthält z.B. alle Buchstaben des Alphabets außer dem "f".

Ihre Funktion soll eine Zeitkomplexität von O(N) haben.

Tip: Die Buchstaben von ‘a’ bis ‘z’ sind fortlaufend ASCII-codiert von 0x61 bis 0x7A.
Ebenso die Buchstben ‘A’ bis ‘Z’ von 0x41 bis 0x5A.
Bilden Sie also einen Kleinbuchstaben auf Arrayindizes ab, indem Sie von ihm den Buchstaben ‘a’ abziehen – dann erhalten Sie für den Buchstaben ‘a’ den Index 0 und für ‘z’ den Index 25.

Übung 2

2. Schreiben Sie eine Funktion **Intersection(…)**, die unter Nutzung von Hashes die Schnittmenge von 2 Arrays ermittelt.
   Diese Schnittmenge ist ein 3. Array, das alle Elemente enthält, die sowohl im ersten als auch im 2. Array enthalten sind.

   Ihre Funktion soll eine Komplexität von O(N) haben.

   Bsp:. Die Schnittmenge von [1, 2, 3, 4, 5] und [0, 2, 4, 6, 8] ist [2, 4].

Übung 3

3. Schreiben Sie eine Funktion **SingleChar()** , die das erste **nicht doppelte Zeichen** eines Strings im englischen Alphabet zurückgibt.

   Der String “minimum” z.B. enthält 2 Zeichen, die nur 1x vorkommen: das "n" und das "u", deshalb sollte Ihre Funktion das "n“ zurückgeben. Die Funktion sollte eine Effizienz von O(N) haben.