

# Codeoptimierungen

Voraussetzung für Codeoptimierung:

- 1) Aktuelles Big O für Laufzeit und Speicherbedarf bestimmen

und:

- 2) Über das denkbar beste Big-O nachdenken!

- Bsp.: Wir schreiben eine Funktion, die jedes Element eines Arrays ausgeben soll.

Das denkbar beste Big O dafür ist:  $O(N)$ ! Wenn wir nämlich jedes Element ausgeben sollen, haben wir keine andere Chance, als jedes Element zu besuchen...

Für Optimierung also zwei Big O bestimmen:

- das aktuelle
- das denkbar beste

-> Sind sie nicht gleich: Optimierungspotential!

Nicht immer kann das denkbar beste Big O erreicht werden, aber vielleicht zumindest annähernd...

# Denkbar bestes Big O

Idee zu Erhöhung der Vorstellungskraft:

“Würde ich jemandem glauben, der behauptet, mein Problem mit einem Big O von ... gelöst zu haben?”

Wenn ja, dann kann das mein denkbar bestes Big O sein...

# Codeoptimierungen I: Schneller Lookup

Fragen Sie sich:

„Wenn ich durch Magie eine gewünschte Information in  $O(1)$  finden könnte, würde das meinen Algorithmus beschleunigen?“

Wenn ja: dann Hashtabelle benutzen, um die Magie wahr werden zu lassen.

Bsp.: Zweier-Summen-Problem

Eine Funktion soll „true“ zurückgeben, wenn in einem übergebenen Array mindestens 1 Zahlenpaar ist, das gemeinsam z.B. 10 ergibt. Der Einfachheit halber sind Duplikate ausgeschlossen (Array ist ein Set).

Unser Array sei: [2, 0, 4, 1, 7, 9]    Funktion gibt true zurück aufgrund 1 + 9!    (false für [2, 0, 4, 5, 3, 9]).

Erster Lösungs-Ansatz: Verschachtelte Schleifen

# Codeoptimierungen I: Schneller Lookup

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

Aktuelles Big O:  $O(N^2)$

Denkbar bestes Big O:    BigBlueButton: was wäre möglich?

- A     $O(1)$
- B     $O(\log N)$
- C     $O(N)$
- D     $O(N \cdot \log N)$
- E     $O(N^2)$

# Codeoptimierungen I: Schneller Lookup

Magische Lookup-Frage:

„Wenn ich durch Zauberhand eine gewünschte Information in  $O(1)$  finden könnte, würde das meinen Algorithmus beschleunigen?“

Wir laufen mit dieser Frage durch das Zweier-Summen-Problem:

Für jede Zahl  $x_i$ : ist die Zahl „ $y_i=10-x_i$ “ im Array enthalten?

Könnten wir uns alle  $y_i$  merken, könnten wir true zurückgeben, sobald wir auf eins davon stoßen – ansonsten false.

-> wir brauchen eine extra Datenstruktur: eine Hashtabelle für magische Lookups .

Unsere Hashtabelle könnte dann so aussehen:

Index	0	1	2	3	4	5	6	7	8	9
Key	0	1	2	3	4	5	6	7	8	9
Value	true	true	true	false	true	false	false	true	false	true

# Codeoptimierungen I: Schneller Lookup

Index	0	1	2	3	4	5	6	7	8	9
Key	0	1	2	3	4	5	6	7	8	9
Value	true	true	true	false	true	false	false	true	false	true

Für jede Zahl kann die Differenz zu 10 jetzt mit einem Lookup von  $O(1)$  nachgesehen werden:

```
#define MAX_VALUE 100
bool TwoSumHash (int array[], int size)
{
    int hashTable [MAX_VALUE]={false};
    for (int i = 0; i < size; i++)    hashTable [array[i]] =true;
    for(int i = 0; i < size; i++)
    {
        if (true == hashTable[10 - array[i]])    return true;
    }
    return false;
}
```

Resultierende Zeit-Performance:  $O(2N) = O(N)$ !

Muster im Problem finden: Komplexität rausnehmen!

Münzenspiel:

Zwei Spieler spielen im Wechsel, mit einem Stapel Münzen:

Jeder Spieler kann genau eine oder genau zwei Münzen herunternehmen. Wer die letzte Münze nimmt, verliert.

Mit der richtigen Strategie kann man den Gegner verlieren lassen (wenn man entscheiden darf, wer beginnt):

- Bei einer letzten Münze verliert der Spieler, der an der Reihe ist.
- Bei zwei oder drei Münzen kann der Spieler, der an der Reihe ist, den anderen verlieren lassen.
- Vier Münzen: der Spieler an der Reihe wird verlieren.





# Codeoptimierungen II: Muster erkennen

Wie können wir mit einem Algorithmus vorhersagen, ob der Start-Spieler oder der Gegner bei einer bestimmten Münzenzahl das Spiel noch gewinnen können?

Heißt auch: muss man bei einer bestimmten Münzenzahl selbst beginnen oder der Andere, um zu gewinnen?

→ Muster der Teilprobleme verwenden! Top-down-Rekursion!

# Codeoptimierungen II: Muster erkennen

```
char * GameWinner (int numberOfCoins, char *currentPlayer)
{
    char * nextPlayer="them";
    // Abbruchkriterium, currentPlayer gewinnt, weil Stapel leer ist
    if (0 >= numberOfCoins)    return currentPlayer;
    if ("you" == currentPlayer) nextPlayer="them";
    else nextPlayer="you";
    if ((currentPlayer == GameWinner(numberOfCoins - 1, nextPlayer))
        || (currentPlayer == GameWinner(numberOfCoins - 2, nextPlayer)))
        return currentPlayer;
    // currentPlayer kann 1 oder 2 Münzen wegnehmen
    else    return nextPlayer;
}
```

Algorithmus optimierbar?

Aktuelles Big O?

- A  $O(\log N)$
- B  $O(N)$
- C  $O(N \cdot \log N)$
- D  $O(N^2)$
- E  $O(2^N)$

# Codeoptimierungen II: Muster erkennen

```
#define MAX_COINS 10
char * winnerHash[MAX_COINS]={"", "", "", "", "", "", "", "", "", ""};
char * GameWinnerHash (int numberOfCoins, char *currentPlayer)
{
    char * nextPlayer="next";
    // Abbruchkriterium, currentPlayer gewinnt, weil Stapel leer ist
    if (0 >= numberOfCoins)    return currentPlayer;
    if ("you" == currentPlayer)  nextPlayer="them";
    else nextPlayer="you";
    if ("" == winnerHash[numberOfCoins - 1])
        winnerHash[numberOfCoins - 1]= GameWinner(numberOfCoins - 1, nextPlayer);
    if ("" == winnerHash[numberOfCoins - 2])
        winnerHash[numberOfCoins - 2]= GameWinner(numberOfCoins - 2, nextPlayer);
    if ((currentPlayer == winnerHash[numberOfCoins - 1])
        || (currentPlayer == winnerHash[numberOfCoins - 2]))
        // currentPlayer gewinnt in beiden Fällen
        return currentPlayer;
    else return nextPlayer;
}
```

Aktuelles Big O? BigBlueButton!

- A  $O(\log N)$
- B  $O(N)$
- C  $O(N \cdot \log N)$
- D  $O(N^2)$
- E  $O(2^N)$

# Codeoptimierungen II: Muster erkennen

Was ist das denkbar beste Big O? Müssen wir jede Münze des Stapels ansehen?

Nein - uns interessiert nur seine Höhe! Sollte  $O(1)$  nicht möglich sein?

*Ist ein Muster erkennbar?*

Unterstützung zur Musterfindung:

Wir erzeugen mehrere Beispiele aus Eingangs- und entsprechenden Ausgangsdaten.

Es ist ein Muster erkennbar!

Anzahl von Münzen	Gewinner (it's your turn)
1	Them
2	You
3	You
4	Them
5	You
6	You
7	Them
8	You
9	You
10	Them

# Codeoptimierungen II: Muster erkennen

Muster:

Wenn wir von der Münzenzahl 1 abziehen, gewinnt der Gegner immer dann und nur dann, wenn die Zahl der Münzen durch 3 teilbar ist.

```
char * GameWinnerDef (int numberOfCoins)
{
    if (0 == ((numberOfCoins - 1) % 3 ))
        return "them";
    else return "you";
}
```

O(1) bzgl. Laufzeit und Speicherplatz!

Anzahl von Münzen	Gewinner (it's your turn)
1	Them
2	You
3	You
4	Them
5	You
6	You
7	Them
8	You
9	You
10	Them

## Das „Sum-Swap-Problem“

In diesem Beispiel können wir sowohl die Mustererkennung als auch das magische Lookup für die Optimierung verwenden:

Unsere Funktion bekommt 2 Arrays von integer Werten, z.B. diese.:

array\_1 = [5, 3, 2, 9, 1]      Sum: 20

array\_2 = [1, 12, 5]      Sum: 18

Wir suchen in jedem Array eine Zahl, die wir vertauschen können, damit die Summen gleich werden.

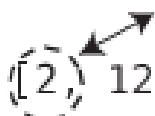
BigBlueButton:    array\_1:   A=5   B=3   C=2   D=9   E=1

array\_2:   A=1   B=12   C=5

# Codeoptimierungen II: Muster erkennen

Wir vertauschen die 2 mit der 1:

array\_1 = [5, 3, (1), 9, 1]      Sum: **19**  
array\_2 = ((2), 12, 5]      Sum: **19**



Unsere Funktion braucht den Swap nicht auszuführen – sie soll nur die beiden zu vertauschen Indizes zurückgeben, z.B. als Returnwert EXIT\_SUCCESS und in einem Array (in unserem Beispiel [2, 0]) – oder EXIT\_FAILURE im Returnwert, wenn es keine mögliche Vertauschung gibt.

Wir könnten diese Funktion mit verschachtelten Schleifen schreiben – die äussere Schleife iteriert über Array 1 und die innere versucht, jeden Wert von array\_1 mit jedem von array\_2 auszutauschen.

**Aktuelle Komplexität:  $O(N \cdot M) = O(N^2)$**

**Denkbar bestes Big O:** Wir müssen im worst case jede Zahl einmal besuchen. Das wäre  $O(N+M)$ , also  $O(N)$ .

# Codeoptimierungen II: Muster erkennen

Sind hier Muster verborgen? Beispiele rechnen:

- Das Array mit der größeren Summe tauscht eine kleinere Zahl als der Array mit der kleineren Summe.
- die Summen ändern sich um den gleichen Betrag
- Die finale Summe liegt exakt in der Mitte der beiden Ausgangssummen.

Before Swap		After Swap	
array_1 = [5, 3, 3, 7]	Sum: 18	[5, 3, 3, 4]	Sum: <b>15</b>
array_2 = [4, 1, 1, 6]	Sum: 12	[7] 1, 1, 6]	Sum: <b>15</b>
Before Swap		After Swap	
array_1 = [1, 2, 3, 4, 5]	Sum: 15	[1, 2, 6, 4, 5]	Sum: <b>18</b>
array_2 = [6, 7, 8]	Sum: 21	[3, 7, 8]	Sum: <b>18</b>
Before Swap		After Swap	
array_1 = [10, 15, 20]	Sum: 45	[5, 15, 20]	Sum: <b>40</b>
array_2 = [5, 30]	Sum: 35	[10, 30]	Sum: <b>40</b>



# Codeoptimierungen II: Muster erkennen

1. Wir können die finale Summe ermitteln.
2. Daraus können wir bestimmen, um wieviel sich jede der Summen ändern muss (Shift).
3. Wir brauchen in jedem Array eine Zahl und einen Counterpart in dem anderen Array. Der Shift muss sich aus  $\pm(\text{Zahl-Counterpart})$  ergeben.
4. Wir suchen also für jede Zahl des Arrays nach folgendem Counterpart im anderen Array:

$$\text{Counterpart} = \text{Shift} \rightarrow \text{Zahl}$$

*Wir suchen also für jede Zahl des einen Arrays nach einer ganz konkreter Zahl im anderen Array.*

Und hier können wir mit dem magischen Lookup beschleunigen.

Wir speichern zunächst die Zahlen des einen Arrays in eine Hashtabelle, in der wir dann für jede Zahl der anderen Tabelle ihren Counterpart nachschlagen.

```
#define LIMIT 15
int sum_swap( unsigned int array1[], unsigned int array2[],
              int size1, int size2, int *result)
{
    // Hash table to store values of first array - limit
    int sum1 = 0, sum2=0;
    for (int i=0; i<LIMIT; i++) hashTable[i]=-1;
    for (int i=0; i<size1; i++)
    {
        // Get sum of first array, while storing its values # in a hash table
        hashTable [array1[i]]=i;
        sum1 += array1[i];
    }
    for (int i=0; i<size2; i++)
    {
        // Get sum of second array
        sum2 += array2[i];
    }
    int shift = (sum1 - sum2) / 2;    // Calculate shift of array2

    for (int i=0; i<size2; i++)
    {
        // look for counterpart in first array for each number of second array
        if ((LIMIT > (array2[i] + shift)) && (hashTable[array2[i]+shift] != -1))
        {
            result[0]= hashTable [array2[i] + shift] ;
            result[1]=i;
            return EXIT_SUCCESS;
        }
    }
    return EXIT_FAILURE;
}
```

Dieser Algorithmus läuft in  $(N+2*M)$ -Zeit!

Wir benötigen Speicherplatz für eine extra Hashtabelle der Größe LIMIT – des maximalen Wertes, bzw. der Größe N, wenn wir aufgrund der Größe von LIMIT eine Hashfunktion einsetzen (Limit  $\gg$  N)

-> wir opfern Platz für Laufzeit.

# Codeoptimierungen III: Datenstruktur wechseln

Neues Gedankenspiel:

Was passiert, wenn wir eine andere Datenstruktur verwenden?

Bsp.:

Wir arbeiten an einem Problem, bei dem uns die Daten als Array gegeben sind.

Überlegung: würden uns Hash-Tabelle, Baum oder andere Struktur bei der Optimierung helfen?

## Bsp. dazu: Anagramm checker:

Sind 2 gegebene Strings Anagramme voneinander?

Denkbar bestes Big O?

Wir müssen jeden Buchstaben beider Strings bestimmt mindestens 1x besuchen ( $O(2N) = O(N)$ ). Höhere Effizienz ist nicht vorstellbar.

Möglicher Ansatz:

Verschachtelte Schleifen, um die beiden Strings zu vergleichen.

Die äußere Schleife iteriert über jedes Zeichen des ersten Strings, und vergleicht dieses mit jedem Zeichen des 2. Strings. Finden wir einen match, löschen wir das Zeichen aus dem 2. String.

Wenn jedes Zeichen des 1. Strings auch im 2. String enthalten ist, darf der 2. String nach Ende der äußeren Schleife keine Zeichen mehr enthalten.

Implementierung:

```
bool AreAnagrams (char *firstString, char *secondString)
{
    for (int i=0; i<strlen(firstString); i++)
    {
        // If we're still iterating through the firstString,
        // but the secondStringArray is already empty:
        if (0 == strlen(secondString)) return false;

        for (int j=0; j<strlen(secondString); j++)
        { // If we find the same character in firstString and secondString
            if (firstString[i] == secondString[j])
            { // Delete the char from the second array and continue outer loop
                for (int k=j; k<=strlen(secondString); k++)
                    secondString [k]= secondString [k+1];
                break;
            }
        }
    }
    // The two strings are only anagrams if secondString has no chars remaining
    return (0 == strlen(secondString));
}
```

# Codeoptimierungen III: Datenstruktur wechseln

Komplexität:  $O(N^2)$

Schnellerer Ansatz: Beide Strings sortieren: müssen anschliessend identisch sein!

->  $O(2 \cdot N \cdot \log N + N)$  mit einem schnellen Sortieralgorithmus wie Quicksort.

Wir möchten aber  $O(N)$ !

Fragestellung: Sind in jeder Zeichenkette genausoviel Buchstaben des gleichen Typs?

Alternative Datenstrukturen:

Hashtabelle (Schritt 1 des Counting sort)? Zeichen als Key/ Index, speichert die Anzahl des Vorkommens im String?

-> erfüllt genau unsere Fragestellung!

Neue Fragestellung: sind die beiden resultierenden Hashtabellen identisch?

```
bool AreAnagramsHash (char *firstString, char *secondString)
{
    // for lower case strings only
    if (strlen(firstString) != strlen(secondString)) return false;
    // Create hash table out of first string and secondString
    int hash1[26]={0}, hash2[26]={0};
    for (int i=0; i<strlen(firstString); i++)
    {
        ++hash1[firstString[i]-'a'];
        ++hash2[secondString[i]-'a'];
    }
    for (int i=0; i<26; i++)
    {
        if (hash1[i] != hash2[i]) return false;
    }
    return true;
}
```



# Codeoptimierungen II: Datenstruktur wechseln

Zeitkomplexität:  $O(N + N + 26)$

Speicherplatzkomplexität:  $O(2 \cdot 26) = O(1)$

Bsp. für andere Datenstruktur:

Wir haben ein Array mit mehreren verschiedenen Werten, die wir neu anordnen wollen, so dass gleiche Werte zusammengruppiert werden. Die Reihenfolge der Gruppen interessiert uns allerdings nicht.

Wir haben z.B. das Array:

```
["a", "c", "d", "b", "b", "c", "a", "d", "c", "b", "a", "d"]
```

Die Gruppenbildung könnte z.B. ergeben:

```
["c", "c", "c", "a", "a", "a", "d", "d", "d", "b", "b", "b"]
```

Ebenfalls gültige Ergebnisse wären:

```
["d", "d", "d", "c", "c", "c", "a", "a", "a", "b", "b", "b"]   und  
["b", "b", "b", "c", "c", "c", "a", "a", "a", "d", "d", "d"]
```

# Codeoptimierungen III: Datenstruktur wechseln

Jeder klassische Sortieralgorithmus würde mit  $O(N \cdot \log N)$  ergeben:

["a", "a", "a", "b", "b", "b", "c", "c", "c", "d", "d", "d"]

Geht es schneller?

Denkbar bestes Big O:  $O(N)$  vorstellbar, da nur die abzählbaren Häufigkeiten interessieren.

Zählen in Hashtabelle + wieder rausschreiben (analog Counting sort). Wir erreichen  $O(N)$ -Zeit!

```
void GroupArray (char *array, int size)
{
    // for lower case strings only
    int hash[26]={0};
    for (int i=0; i<size; i++)
    {
        ++hash[array[i]-'a'];
    }
    int j=0;
    for (int i=0; i<26; i++)
    {
        if (0 != hash[i] )
        {
            for (int k=0; k<hash[i]; k++)
            {
                array[j] = i+'a';
                j++;
            }
        }
    }
}
```

Optimierung:

1. Aktuelles Big O bestimmen
2. Denkbar bestes Big O bestimmen
3. Algorithmus optimieren über
  - magisches Lookup,
  - Auffinden von Ergebnismustern,
  - andere Datenstrukturen ...

Diese Vorlesung sollte Ihnen ermöglichen, fundierte Entscheidungen bzgl. der Effizienz von Algorithmen zu treffen.

Und:

Themen wie diese, die als komplex und esoterisch verschrien sind, basieren auf einfachen Konzepten.

Lassen Sie sich nicht einschüchtern , wenn etwas zunächst komplex und schwierig aussieht – mit Hilfe von Beispielen kann alles einfach erklärt werden.