Codebeschleunigung mit Big O

- Die Big-O-Notation ermöglicht das Vergleichen der Performance konkurrierender Algorithmen.

  Bsp.: Vergleich der linearen mit der binären Suche, Letztere ist wesentlich schneller (O (log N)) als die lineare mit O(N).

- Vergleichen Sie auch stets Ihre eigenen Algorithmen mit dem Rest der Welt!

- Finden Sie einen Weg, sie so zu optimieren, dass sie in eine schnellere Big-O-Kategorie fallen!

- Nächster Schritt:
  Wir schreiben Code für die Lösung eines praktischen Problems, und messen unseren Algorithmus mit Big-O.

  Dann werden wir sehen, ob wir seine Effizienz noch boosten können…

Praktisches Problem: Sortieralgorithmen

Eine der Hauptaufgaben der Informatik ist die Optimierung von Datenstrukturen und Algorithmen für eine möglichst effiziente **Suche**.

Die **Sortierung der Eingangsdaten ist oft** eine wichtige Voraussetzung!

Sortieralgorithmen sind im Fokus der Informatikforschung – und Tonnen solcher Algorithmen wurden seit den 60er Jahren entwickelt. Sie alle lösen folgendes Problem:

Gegeben sei ein Array unsortierter Werte –
wie können wir diese sortieren, so dass sie eine aufsteigende Folge ergeben?

Hunderte dieser Algorithmen sind veröffentlicht – einfachere (weniger effiziente) und komplexere (effizientere)…

Sortierproblem

- Gegeben: die Ordnung $\leq$ auf der Menge M möglicher Werte
- Eingabe: Sequenz $s = \{e_1, \ldots, e_n\}$

Beispiel: | 3 | 7 | 8 | 6 | 4 | 2 |

- Ausgabe: Permutation $s' = \{e'_1, \ldots, e'_n\}$ von s,

  so daß Wert $(e'_1) \leq$ Wert (e'_{i+1})

  für alle i $\{1, \ldots, n\}$ (Totale Ordnung)

Beispiel: | 2 | 3 | 4 | 6 | 7 | 8 |

- M kann eine Menge zusammengesetzter Objekte sein

Auswahlkriterien

- Beispiele für unterschiedliche Verfahren:
  - besonders platzsparend (für intern verfügbare Daten)
  - Besonders geeignet zur effizienten Sortierung großer externer Datenmengen mit grossem Wertebereich
  - besonders schnell bei sehr vielen Elementen

(paarweise) VERGLEICHEND	vs.	NICHT VERGLEICHEND
Selectionsort, Bubblesort, Insertionsort, Quicksort, Mergesort, Heapsort, Blocksort, Coctailsort, Combsort, Cyclesort, Gnomesort, Introsort, Librarysort, Patiencesorting, Shellsort, Smoothsort, Stoogesort, Strandsort, Timsort, Turnamentsort, Unshufflesort usw.		Countingsort, Bucketsort, Radixsort, Pigeonholesort, Spreadsort, usw.

Bubble sort: Sortieren durch Aufsteigen

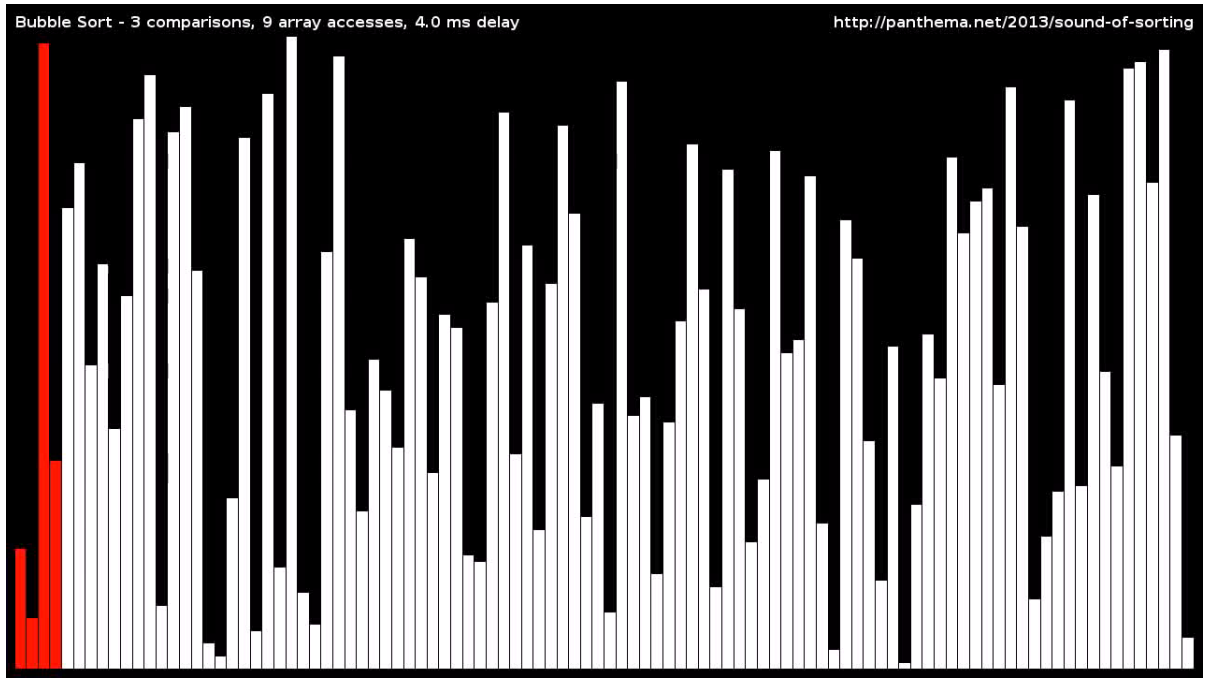
Idee:

Alle Spielkarten kommen in die Hand, anschließend werden benachbarte Karten a-b, b-c, c-d,… paarweise in die korrekte Reihenfolge getauscht, bis alle sortiert sind. Bei jedem Durchgang steigen so die „grossen Blasen“ auf…

Oder:

Wir gehen unser Bücherregal Position für Position durch, und tauschen jedes Buch mit seinem rechten Nachbarn, wenn nötig.

Das wiederholen wir, bis wir in einem kompletten Durchgang nicht mehr tauschen mußten.

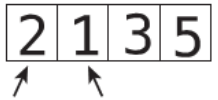


http://panthema.net/2013/sound-of-sorting

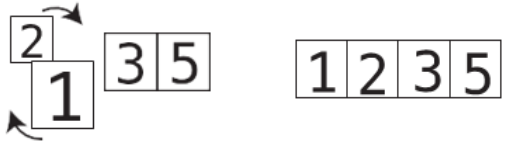
Bubble sort: Sortieren durch Aufsteigen

Bubble Sort ist ein sehr einfacher und bekannter Sortieralgorithmus, und folgt folgenden Schritten:

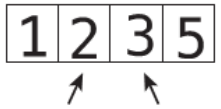
1. Vergleiche die beiden benachbarten Elemente eines Arrays.



2. Vertausche die Elemente, wenn nicht aufsteigend oder gleich, und merke Dir, dass Du getauscht hast:



3. Bewege die Zeiger eine Position nach rechts:



4. Wiederhole Zeilen 1…3, bis ein Zeiger vor das Element zeigt, das in der letzten Runde als letztes verglichen wurde (die letzte „grosse Blase“).

5. Wiederhole Zeilen 1…4, solange Du im Durchlauf 1…3 getauscht hast.

6 5 3 1 8 7 2 4

Von Swfung8 - Eigenes Werk, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=14953478

Bubble Sort: Implementierung

```
void Swap (int *i_xp, int *i_yp)
{
    int tmp=*i_xp;
    *i_xp=*i_yp;
    *i_yp=tmp;
}
```

```
int main(int argc, char** argv)
{
    int array []={4,-3,2,10,-9};
    BubbleSort(array,
               sizeof(array)/ sizeof(array[0]));
    return (EXIT_SUCCESS);
}
```

```
void BubbleSort(int io_array[], int i_size)
{
    int upperIndex=i_size-1;
    bool isSorted =false;
    while (! isSorted)
    {
        isSorted=true; // could be sorted
        for (int j=0; j<upperIndex; j++)
        {   // compare 1 element less than last run
            if (io_array[j] > io_array[j+1])
            {   // swap elements!
                Swap (&io_array[j], &io_array[j+1]);
                isSorted=false; // was not sorted...
            }

        }
        upperIndex--; // next bubble reached the end
    }
}
```

Bubble Sort: Big O - Bestimmunung

2 Arten von Schritten:

- Vergleiche
- Vertauschungen

Vergleiche: max. (N - 1) + (N - 2) + (N - 3) … + 1

Vertauschungen:

im worst case einen pro Vergleich,

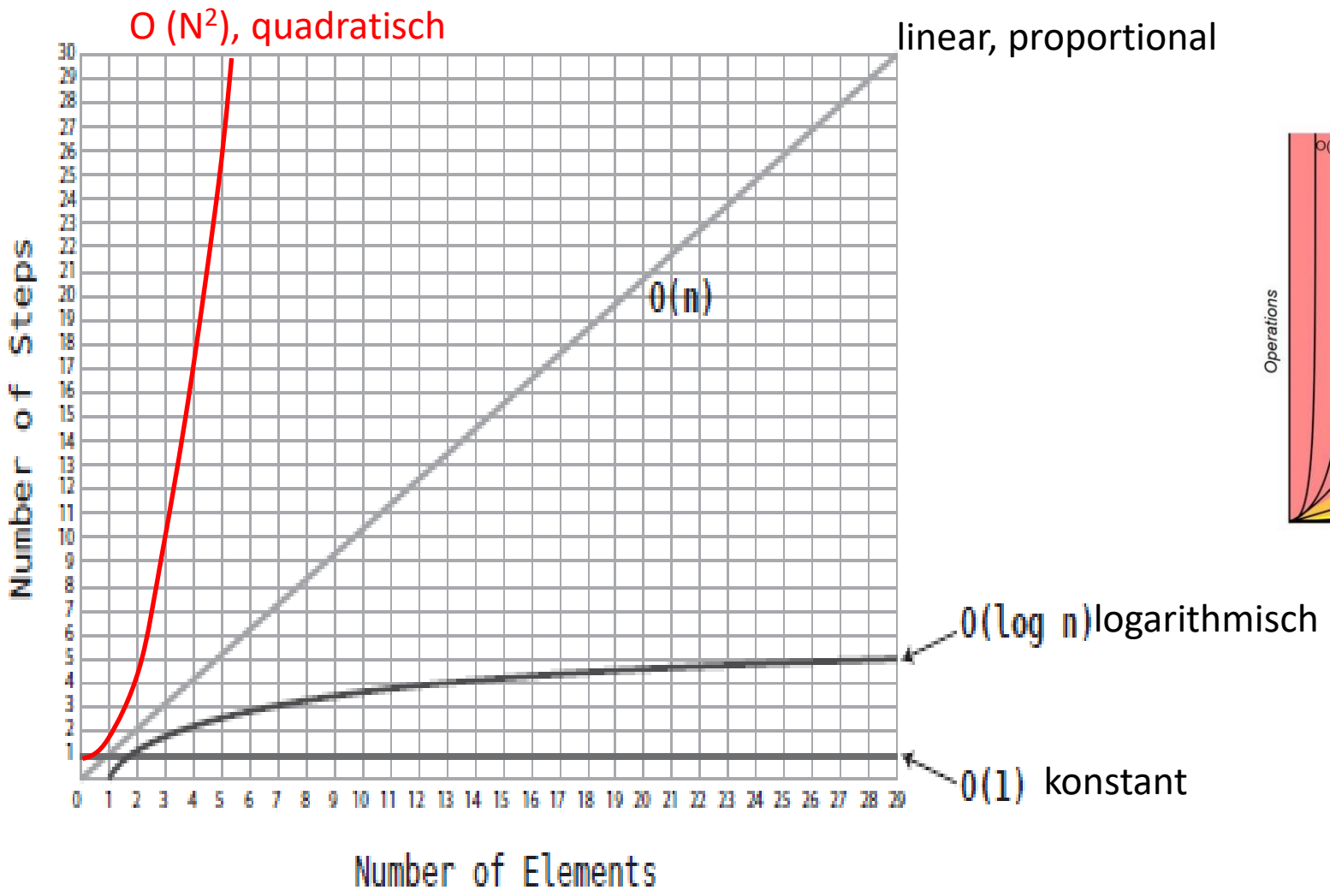
max. (N - 1) + (N - 2) + (N - 3) … + 1

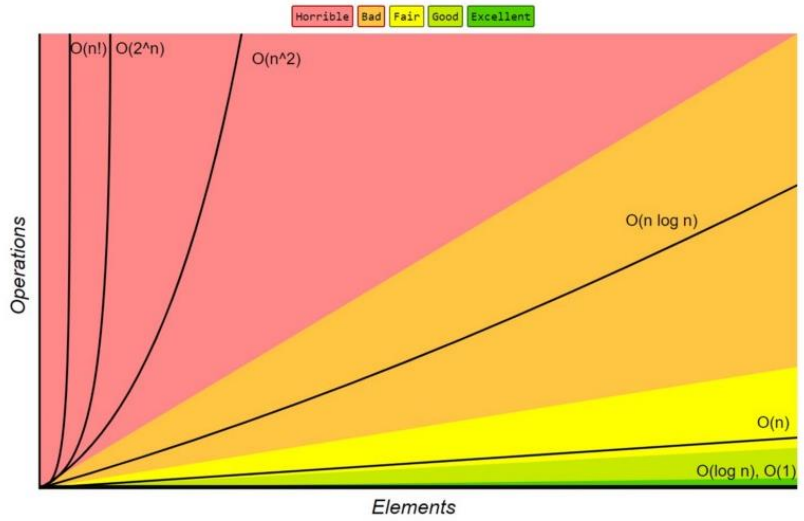
N data elements	Max # of steps
5	20
10	90
20	380
40	1560
80	6320

N-mal wird das Array aus n Elementen (fast vollständig für Vergleiche und Vertauschungen) durchgegangen:

Big O: Schritte wachsen um ca. N^2: O (N^2)

Bubble Sort: Big O - Bestimmung





Complexity Growth Illustration from Big O Cheatsheet

Ein weiteres Bsp. für quadratische Komplexität

Problem: **Prüfen, ob ein Array Duplikate enthält**:

Array [1, 5, 3, 9, 1, 4] hat 2 Instanzen der Zahl 1: return true

Erster Ansatz: geschachtelte for-Schleifen

```
bool HasDuplicateValue (unsigned 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])
            {
                return true;
            }
        }
    }
    return false;
}
```

Anzahl Vergleiche: N^2 (size x size)

$O(N^2)$

-> Nachdenken über effizientere Lösung lohnt sich!

Eine lineare Lösung

Lösung ohne verschachtelte for-Schleifen:

```
#define MAX_NUM 10  // largest valid unsigned number in array
bool HasDuplicateValue (unsigned int array[], int size)
{
    int existingNumbers [MAX_NUM]={0};

    for(int i = 0; i < size; i++)
    {
        if (0 == existingNumbers [array[i]] )
        {
            existingNumbers [array[i]] = 1;
        }
        else    return true;
    }
    return false;
}
```

Anzahl Vergleiche: N (size)

O (N) << O (N^2)

Nachteil:
evtl. hoher Platzbedarf für Array „existingNumbers“!

Zusammenfassung/ Ausblick

- Die Big-O-Notation kann uns erlauben, langsamen Code zu finden und den schnelleren zweier konkurrierender Algorithmen zu identifizieren

- Es gibt Situationen, wo Big-O uns glauben macht, zwei Algorithmen wären gleich schnell, während in Wirklichkeit einer schneller ist

- In der nächsten Lektion erfahren wir, wie wir die Effizienz verschiedener Algorithmen bewerten können, selbst wenn Big-O nicht trennscharf genug ist

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Codeoptimierung mit und ohne Big O

- Big-O ist nicht das einzige Tool, um Algorithmen vergleichen und bestimmen zu können, welche in einer bestimmten Situation verwendet werden sollten

- Möglich:

  Zwei konkurrierende Algorithmen werden mit Big-O-Kategorien exakt genauso bewertet, obwohl einer signifikant schneller ist als der andere

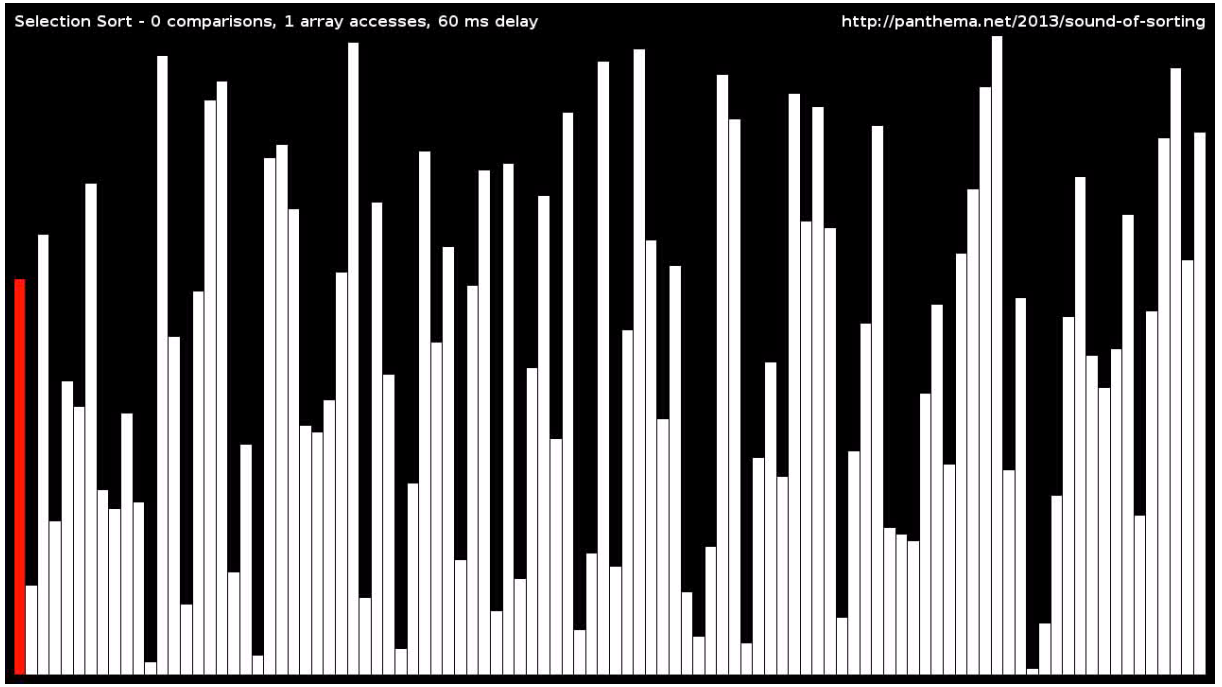
Selektion sort: Sortieren durch Auswahl

Idee:

Die jeweils niedrigste Karte wird vom Tisch (unsortierter Teil) gesucht, in die Hand genommen und an die schon sortierten dort angefügt („Kleine zuerst!“) – evtl. durch Platztausch mit der nächsten Karte im Array.

Oder:

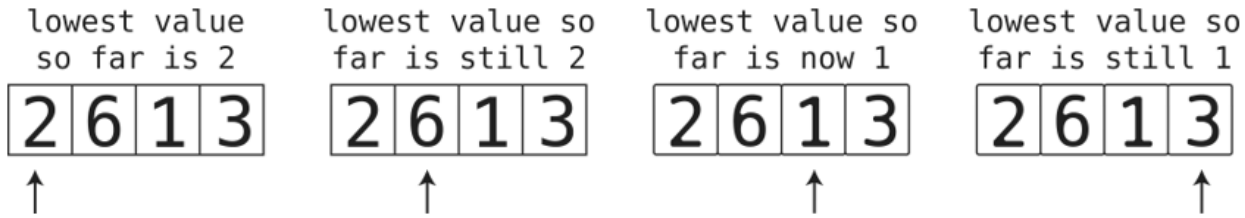
Im Bücherregal wird im noch nicht sortierten Bereich das nächste Buch im Alphabet gesucht, und mit dem aktuellen vertauscht.



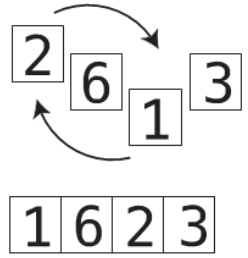
http://panthema.net/2013/sound-of-sorting

Selection sort

1. Wir prüfen von links nach rechts (ab den bereits sortierten) jede Zelle, um das kleinste Element zu finden, und merken uns jeweils das bislang kleinste über seinen Index.



2. Wenn wir wissen, welcher Index den kleinsten Wert enthält, tauschen wir seinen Wert mit dem Wert der Zelle, mit der wir die Vergleiche in diesem Durchlauf begonnen haben. Im ersten Durchlauf wäre das der Index 0, Index 1 im zweiten, usw. .
In der Abbildung wird der Tausch beim ersten Durchlauf gezeigt.



3. Wir wiederholen Schritte 1 und 2, bis alle Daten sortiert sind.

Sucht (selektiert) das kleinste Element der Restmenge und tauscht es mit dem ersten Element dieser Menge.

Selection sort: Implementierung

```
void SelectionSort (int io_array[], int i_size)
{
    int lowestNumberIndex;
    for (int i=0; i<i_size; i++)
    {
        lowestNumberIndex=i; // sorted up to i-1
        for (int j=i+1; j<i_size; j++)
        {   // find lowest numer, start behind  sorted elements
            if (io_array[j] < io_array[lowestNumberIndex])
                                        lowestNumberIndex = j;
        }
        if (lowestNumberIndex != i)
            Swap(&io_array[i], &io_array[lowestNumberIndex]);
    }
}
```

Selection sort: Effizienz

Zwei Arten von Schritten: Vergleichen und Vertauschen

für 5 Elemente:

- Vergleiche (in Restmenge):
  $4 + 3 + 2 + 1 = (N - 1) + (N - 2) + (N - 3) \ldots + 1$ Vergleich = $n*(N-1)/2$ Vergleiche (ca. $N^2/2$)

- Vertauschungen: maximal ein Tausch pro Durchlauf = N

-> $O(N^2/2+N)$

Selection sort: Effizienz

Vergleich der Schritte von Bubble sort und Selection sort in Abhängigkeit von der Elementzahl:

N Elemente	Max. # v. Schritten in Bubble Sort	Max. # v. Schritten in Selection Sort
5	20	14 (10 Vergleiche + 4 Swaps)
10	90	54 (45 Vergleiche + 9 Swaps)
20	380	199 (180 Vergleiche + 19 Swaps)
40	1560	819 (780 Vergleiche + 39 Swaps)
80	6320	3239 (3160 Vergleiche + 79 Swaps)

Selection sort ist doppelt so schnell: $O(N^2)$ $O(N^2/2 + N)$

Selection sort: Effizienz

Eine Hauptregel von Big-O ist:

Die Big-O-Notation ignoriert Konstanten! *(kümmert sich nur um die Größenordnung)*

Oder:

Die Big-O-Notation enthält nie Zahlenwerte ausser Exponenten (und der '1' für konstante Komplexität).

In unserem Fall wird so aus $O(N^2 / 2)$ einfach $O(N^2)$.

Genauso wird aus einer Komplexität:

O(2N) → O(N),
O(N /2) → O(N)
O(100N) → O(N), obwohl sie Faktor 100 langsamer ist.

Selection sort: Effizienz

Ist die Big-O-Notation nicht völlig nutzlos, wenn es zwei Algorithmen geben kann,

die mit der exakt gleichen Big-O-Komplexität bewertet werden,

obwohl einer 100x schneller läuft als der Andere?

Big - O - Kategorien

Die Big-O-Notation denkt nur in *generellen Kategorien* von Algorithmen-Geschwindigkeiten.

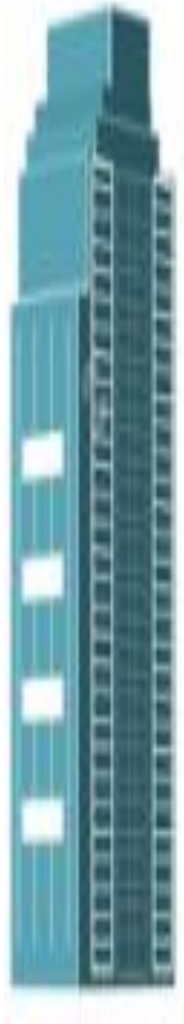
Bsp. Gebäude:
Es gibt eingeschossige Einfamilienhäuser, zweigeschossige Einfamilienhäuser
und grosse Stadtvillen für Familien.
Es gibt Mehrfamilienhäuser mit sehr unterschiedlicher Kapazität.
Und es gibt verschiedenste Wolkenkratzer.

KategorieN: “EFH”, ..., “Wolkenkratzer”.
Die generelle Kategorie zu nennen reicht, um die Hauptunterschiede zu verdeutlichen.

O(1)

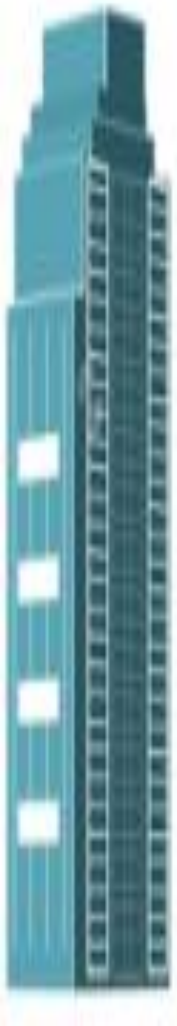
O(log N)

O (N)



O(N^2)

Big - O - Kategorien



O(N) versus $O(N^2)$

Die beiden Effizienzen sind so unterschiedlich, dass es keine Rolle spielt,

ob der O(N) Algorithmus in Wirklichkeit ein O(2N), O(N /2) oder sogar O(100N)-Algorithmus ist.

Warum ist das so?

O(1)

O(log N)

O (N)

$O(N^2)$

Big-O - Kategorien

Die “Seele” von Big-O:

Die Big-O-Notation betrachtet nicht nur die Anzahl der Schritte eines Algorithmus.

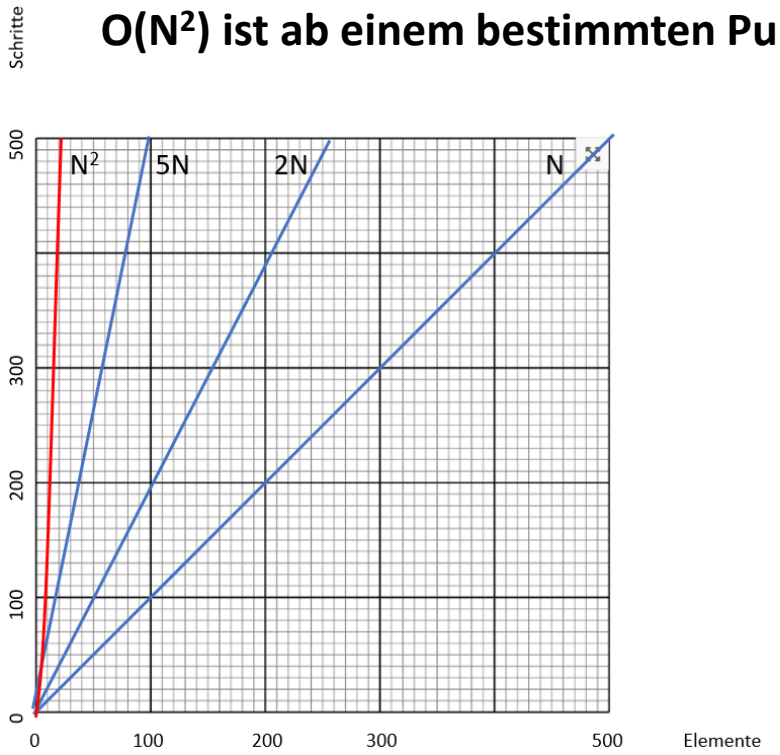
Sie betrachtet den Langzeitverlauf der Algorithmenschritte bei steigender Datenmenge.

Big-O - Kategorien

O(N) steht für lineare Proportionalität
—> die Anzahl der Schritte steigt in einem linearen Verhältnis zur Datenmenge (auch für 100N).

O(N^2) steht hingegen für exponentielles Wachstum!

O(N^2) ist ab einem bestimmten Punkt langsamer als O(N), multipliert mit einem beliebigen Faktor!



Big O - Kategorien

O(2N) mit O(N^2) zu vergleichen ist, wie ein zweistöckiges Haus einem Wolkenkratzer gegenüberzustellen!

Wir können sagen, dass O(2N) Teil der generellen Kategorie von O(N) ist.

Wenn jedoch 2 Algorithmen in dieselbe Kategorie von Big-O fallen, heißt das nicht zwangsläufig, dass sie genauso schnell sind bei gleicher Datenmenge.

Big-O ist geeignet für die Gegenüberstellung von Algorithmen, die in verschiedene Kategorien von Big-O fallen –

gleichen Kategorie: weitere Analysen sind notwendig, um zu bestimmen, welcher Algorithmus schneller ist.

Komplexitätsklassen - Größenordnungen

Klasse	Art der Komplexität	Beispiel
O(1)	Die Rechenzeit ist unabhängig von der Problemgröße.	Sequenz
O(log n)	Die Rechenzeit wächst logarithmisch (Basis 2) mit der Problemgröße.	Häufig bei Zerlegung eines Problems in Teilproblem und Berechnung eines Teilproblems (z.B. binäre Suche)
O(n)	Die Rechenzeit wächst linear mit der Problemgröße.	Abarbeitung eines eindimensionalen Arrays über den Eingabedatenbereich (z.B. lineare Suche)
O(n log n)	Die Rechenzeit wächst linear logarithmisch (Basis 2) mit der Problemgröße.	Häufig bei Zerlegung eines Problems in Teilproblem und Berechnung aller Teilprobleme
$O(n^2)$	Die Rechenzeit wächst quadratisch mit der Problemgröße.	einfache Sortieralgorithmen (z.B. bubble sort), Abarbeiten von 2-D Strukturen
$O(n^3)$	Die Rechenzeit wächst kubisch mit der Problemgröße.	z.B. Matrixmultiplikation, Abarbeiten von 3-D Strukturen
$O(2^n)$	Die Rechenzeit wächst exponentiell mit der Problemgröße.	Typisch für kombinatorische Aufgaben (Türme von Hanoi)
O(n!)		Abarbeitung aller Permutationen (z.B. Weglängenbestimmung in Bäumen)

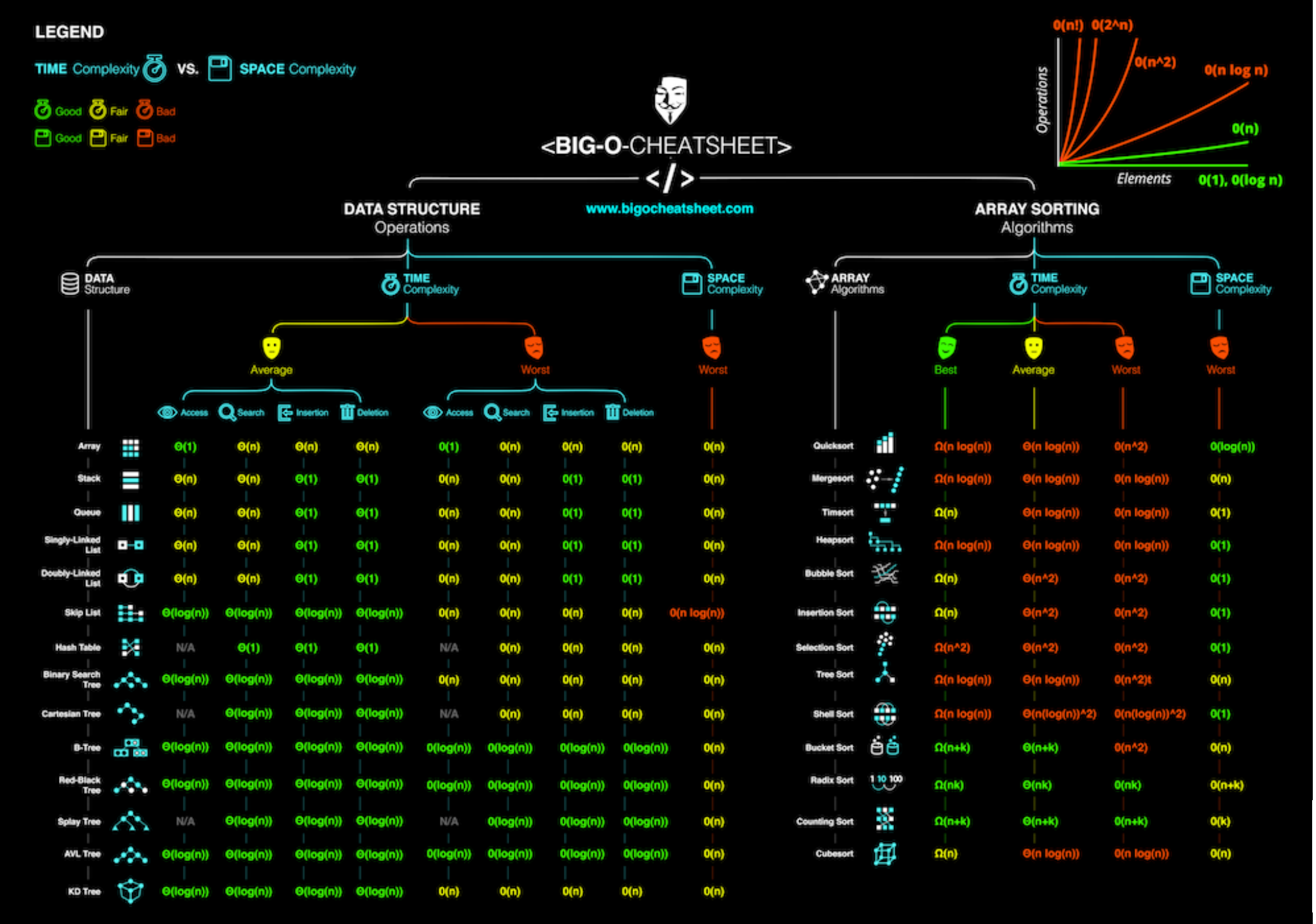
Zusammenfassung und Ausblick

Wir können nun Big-O für die grobe Bestimmung nutzen, wie effizient ein Algorithmus ist,
und wir können auch zwei Algorithmen vergleichen, die in die gleiche Big-O-Kategrie fallen.

Bislang haben wir uns darauf fokussiert, wie Algorithmen im **worst-case** performen.

Per Definition geschehen “worst”-case Szenarios aber eben nicht ständig.
Im Mittel sind die Szenarien natürlich average-case, also Durchschnitts-Szenarien.

-> nächstes Thema



https://www.redbubble.com/de/i/poster/Offizielles-Big-O-Spickzettel-Poster-von-immortalloom/22929408.G7H47

Übung

1) Nutze die Big-O-Notation, um die Zeit-Komplexität der folgenden Funktion zu beschreiben, die bestimmt, ob ein bestimmtes Jahr ein Schaltjahr ist:

```
bool IsLeapYear (int year)
{
    return (year % 100 == 0) ? (year % 400 == 0) : (year % 4 == 0);
}
```

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

Übung

2) Nutze die Big-O-Notation, um die Zeit-Komplexität der folgenden Funktion zu beschreiben, die alle Zahlen eines gegebenen Arrays summiert:

```
long ArraySum (int *array, int size)
{
    long sum = 0;
    for (int i = 0; i < size; i++)      sum += array[i];
    return sum;
}
```

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

Übung

3) Die folgende Funktion basiert auf einer alten Erzählung, die die Macht des Zinseszins beschreibt:

Sie haben ein Schachbrett (8 x 8), und legen ein einzelnes Reiskorn auf das erste Feld.
Auf das zweite Feld legen Sie 2 Körner – die doppelte Anzahl wie auf das erste.
Auf das dritte Feld kommen 4 Reiskörner.
Auf das vierte 8, auf das fünfte 16, und so weiter...

Die folgende Funktion berechnet, auf welches Feld eine gegebene Anzahl an Reiskörnern kommt.
Die Funktion gibt für 16 Körner z.B. 5 zurück, weil 16 Körner auf das 5. Feld gehören.

Nutzen Sie die Big-O-Notation, um die Komplexität der Funktion zu beschreiben.

```
int ChessboardSpace (long double numberOfGrains)
{
    int chessboardSpaces = 1;
    long double placedGrains = 1;
    while (placedGrains < numberOfGrains)
    {
        placedGrains *= 2;
        chessboardSpaces += 1;
    }
    return chessboardSpaces;
}
```

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

Übung

4) Die folgende Funktion erhält ein Array von Strings, und gibt ein neues Array zurück, das nur die Strings enthält, die mit einem ‘a’ beginnen. Nutzen Sie die Big-O-Notation, um die Komplexität der Funktion zu beschreiben.

```
#define STRINGS 5
#define MAX_LENGTH 5

void SelectAStrings (char *i_array[STRINGS], char *i_newArray[STRINGS])
{
    int new=0;
    for(int i = 0; i < STRINGS; i++)
    {
        if (i_array[i][0] == 'a' )
        {
            char *newString = malloc(MAX_LENGTH+1);
            if (newString)
            {
                strncpy(newString, i_array[i], MAX_LENGTH+1);
                i_newArray[new]=newString;
                new++;
            }
        }
    }
}
```

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

Übung

5) Die folgende Funktion berechnet den Median eines sortierten Arrays.
Nutzen Sie die Big-O-Notation, um die Komplexität der Funktion zu beschreiben.

```
int Median (int array[], int size)
{
    int middle = size/ 2, result=0;
    if (0 == size% 2) // If array has even amount of numbers:
    {
        result= (array[middle -1 ] + array[middle]) / 2;
    }
    else // If array has odd amount of numbers:
    {
        result= array[middle];
    }
    return result;
}
```

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

Übungen 2.2

1. Ersetzen Sie die Fragezeichen in der Tabelle unten so, dass sie aufzeigen, wie viele Schritte für eine gegebene Menge an Daten bei den verschiedenen Komplexitätsstufen anfallen:

N elements	O(N)	O(log N)	$O(N^2)$
100	100	?	?
2000	?	?	?

A:

N elements	O(N)	O(log N)	$O(N^2)$
100	100	ca. 7	10.000
2000	2000	ca. 14	400.000

B:

N elements	O(N)	O(log N)	$O(N^2)$
100	100	ca. 7	10.000
2000	2000	ca. 11	4.000.000

C:

N elements	O(N)	O(log N)	$O(N^2)$
100	100	ca. 7	10.000
2000	2000	ca. 14	4.000.000

D:

N elements	O(N)	O(log N)	$O(N^2)$
100	100	ca. 7	10.000
2000	2000	ca. 11	400.000

Übungen 2.2

2. Wenn wir einen $O(N^2)$ Algorithmus haben, der ein Array in 256 Schritten verarbeitet – wie groß ist dann das Array?

A: 8 Elemente
B: 16 Elemente
C: 128 Elemente
D: 65536 Elemente

Übungen 2.2

3. Nutzen Sie die Big-O-Notation, um die Zeitkomplexität der folgenden Funktion zu beschreiben:

   Sie findet das maximale Produkt aller Paare von 2 Zahlen innerhalb eines gegebenen Arrays.

```
int MaximumProduct (int array[], int size)
{
    int maximumProductSoFar = array[0] * array[1];
    for (int i=0; i<size; i++)
    {
        for (int j=0; j<size; j++)
        {
            if ( (i != j) && (array[i] * array[j] > maximumProductSoFar))
                maximumProductSoFar = array[i] * array[j] ;
        }
    }
    return maximumProductSoFar;
}
```

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

Übungen 2.3

1. Nutzen Sie die Big-O-Notation, um die Zeitkomplexität eines Algorithmus zu beschreiben, der 4N + 16 Schritte braucht.

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

Übungen 2.3

2. Nutzen Sie die Big-O-Notation, um die Zeitkomplexität eines Algorithmus zu beschreiben, der $N^2 / 2$ Schritte braucht.

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

Übungen 2.3

3. Nutzen Sie die Big-O-Notation, um die Zeitkomplexität der folgenden Funktion zu beschreiben, welche die Summe aller zuvor verdoppelten Elemente eines Arrays zurückgibt:

```
long int Double_then_sum(int array[], int size)
{
    long doubled_array [size];
    long sum=0;
    for (int i=0; i<size; i++)  doubled_array [i]= array[i]<<1;
    for (int i=0; i<size; i++)  sum += doubled_array [i];
    return sum;
}
```

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

Übungen 2.3

4. Nutzen Sie die Big-O-Notation, um die Zeitkomplexität der folgenden Funktion zu beschreiben, welche die Elemente eines Arrays aus Strings mehrfach verschieden ausgibt (ab ‘a’) (strchr() gibt einen Zeiger auf das erste Auftreten des Zeichens zurück):

```
void Multiple_cases (char *array[], int size)
{
    for (int i; i<size; i++)
        printf ("%s %s\n", array[i], strchr(array[i], 'a'));
}
```

```
abc abc
daef aef
ghai ai
jkla a
amno amno
paqr aqr
stau au
vwxa a
ayz ayz
eaee aee
```

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

Übungen 2.3

5. Die nächste Funktion iteriert über ein Array von Zahlen, und für jede, deren Index geradzahlig ist, gibt sie die Summe aus dieser Zahl plus jeder Zahl des Arrays aus. Wie ist die Effizienz dieses Algorithmus in Big-O-Notation?

```
void EvenSum (int array[], int size)
{
    for (int i=0; i<size; i++)
    {
        if (0 == i%2)
        {
            int sum=array[i];
            for (int j=0; j<size; j++) sum+=array[j];
            printf ("%d %d\n", array[i], sum);
        }
    }
}
```

```
0 45
2 47
4 49
6 51
8 53
```

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

Übungen

Schätzen Sie, indem Sie das Programm für kleine Werte durchlaufen lassen, wie lange die Implementierung des Siebs des Erastothenes für einen Durchlauf mit N=1.000.000 benötigen würde (wenn genug Speicherplatz vorhanden wäre).

```
#include <time.h>
#include <sys/time.h>

int gettimeofday(struct timeval *zeit, void *tzp);

void Sieb_des_Erastothenes (long int n)   // Bestimmung aller Primzahlen zwischen 2 und n
{
    int i, a[n+1];
   struct timeval start, end;
   gettimeofday(&start, 0);

    for(a[1]=0,i=2; i<=n; i++)  a[i]=1; // Initialisierung aller Zahlen als Primzahl 1

    for (i=2; i <= n/2; i++)                    // fuer die erste Haelfte der Zahlen (weil ab da alle Zahlen Vielfache sind): alle Vielfachen <*n/i streichen (=0)
          for (int j=2; j<=n/i; j++) a[i*j]=0;

   gettimeofday(&end, 0);

   printf("Sek Mikrosekunden: %d+%d\n", end.tv_sec - start.tv_sec, end.tv_usec - start.tv_usec);
   for (i=1; i<=n; i++)   if (a[i]) printf ("%d ", i);     // Ausgabe der Primzahlen als Index (Wert !=0)
   printf ("\n");
}
```

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

```
int main(int argc, char** argv)
{
   Sieb_des_Erastothenes (100);
   return (0);
}
```