TECHNISCHE UNIVERSITÄT DARMSTADT

Effiziente Software

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Überblick: was bedeutet Effizienz?

Bevor wir ins Detail gehen, schauen wir uns erst einmal an, worum es bei diesem Thema eigentlich geht.

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- Laufzeit
- Speicherplatz
- Netzwerkbelastung
- Reservierung von Ressourcen

Das sind die vier wesentlichen Punkte bei der Analyse von Software hinsichtlich Effizienz, und zwar auf allen Ebenen: von einzelnen Anweisungen über Subroutinen, Klassen bis hin zu ganzen Programmen. Wir werden diese Punkte in diesem Kapitel von verschiedenen Seiten ansprechen.

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- Laufzeit
- Speicherplatz
- Netzwerkbelastung
- Reservierung von Ressourcen

Das wahrscheinlich wichtigste Kriterium für die Effizienz von Software dürfte die Laufzeit sein, also wie schnell die Software mit ihrer Aufgabe fertig ist. Das hängt natürlich von verschiedenen Dingen ab, etwa von der Hardwareplattform und der aktuellen Auslastung der Hardwareplattform, aber natürlich auch davon, wie geschickt oder ungeschickt die Software entworfen und implementiert ist. Natürlich geht es in der FOP um die Software, nicht um die Hardware. Laufzeit von Software wird der zentrale Punkt des gesamten Kapitels sein.

In späteren Lehrveranstaltungen wird auch vorausgesetzt, dass Sie das Thema Laufzeit von Software in der FOP intellektuell durchdrungen haben.

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- Laufzeit
- Speicherplatz
- Netzwerkbelastung
- Reservierung von Ressourcen

Heutzutage ist Speicherplatz recht billig geworden und spielt gegenüber der Laufzeit häufig keine große Rolle mehr. Es gibt aber auch weiterhin Anwendungen, in denen sparsamer Umgang mit dem Speicherplatz erforderlich ist, entweder weil die Datenmenge riesig ist oder weil der Speicherplatz etwa in Spezialprozessoren stark beschränkt ist. Daher werden wir auch diesen Punkt kurz beleuchten.

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- Laufzeit
- Speicherplatz
- Netzwerkbelastung
- Reservierung von Ressourcen

Bei Programmen, die Daten über ein Netzwerk schicken, ist natürlich auch die Frage wesentlich, ob das Programm wirklich nur so viel an Daten verschickt, wie es zur Erfüllung seiner Zwecke unbedingt muss.

Das ist heutzutage sicherlich ein ebenso wichtiges Thema wie der erste Punkt, die Laufzeit. Das Thema Netzwerkbelastung wird allerdings in späteren Lehrveranstaltungen wieder aufgegriffen, in denen es primär um die Kommunikation über Netzwerke geht. Daher streifen wir das Thema in der FOP nur kurz.

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- Laufzeit
- Speicherplatz
- Netzwerkbelastung
- Reservierung von Ressourcen

Mit Ressourcen sind in der Informatik ganz speziell Entitäten gemeint, die exklusiv von einem Prozess reserviert werden, so dass ein anderer Prozess nicht oder nur eingeschränkt darauf zugreifen darf. Das sind zum Beispiel Dateien, die zum Schreiben geöffnet sind, oder Elemente in einer Datenbank oder auch manche externe Geräte.

Dieses Thema wird in späteren Lehrveranstaltungen zu Datenbanksystemen eingehender behandelt, daher streifen wir hier in der FOP auch dieses Thema nur kurz.

Pareto-Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

- Allgemeine statistische Regel aus der Volkswirtschaftslehre
  - benannt nach ihrem Schöpfer: Vilfredo Pareto
- Übertragung auf das Thema effiziente Software:
  - Nur wenige Stellen im Quelltext sind für den Löwenanteil von Laufzeit / Speicherverbrauch / Netzwerkbelastung / Ressourcenreservierung verantwortlich.
- Nur eine *heuristische* Regel!
  - Aber: in der Praxis bewährt
- Konsequenz für Effizienzverbesserungen von Software:
  - Erst prüfen, wo genau die Effizienzverluste wirklich auftreten.
  - Sich bei Effizienzverbesserungen auf diese Stellen konzentrieren.

Man muss sich klarmachen, dass typischerweise nicht der gesamte Quelltext gleichermaßen dafür verantwortlich ist, dass ein Programm zu ineffizient ist. Dahinter steht eine ganz allgemeine Regel, die in den verschiedensten Bereichen menschlichen Bemühens erfolgreich Anwendung findet, die Pareto-Regel aus der Ökonomie.

Pareto-Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

- Allgemeine statistische Regel aus der Volkswirtschaftslehre
  - benannt nach ihrem Schöpfer: Vilfredo Pareto
- Übertragung auf das Thema effiziente Software:
  - Nur wenige Stellen im Quelltext sind für den Löwenanteil von Laufzeit / Speicherverbrauch / Netzwerkbelastung / Ressourcenreservierung verantwortlich.
- Nur eine *heuristische* Regel!
  - Aber: in der Praxis bewährt
- Konsequenz für Effizienzverbesserungen von Software:
  - Erst prüfen, wo genau die Effizienzverluste wirklich auftreten.
  - Sich bei Effizienzverbesserungen auf diese Stellen konzentrieren.

Konkret bei uns bedeutet die Pareto-Regel: An den meisten Stellen im Quelltext braucht man überhaupt nichts herumzuschrauben. Es lohnt sich einfach nicht, weil diese Stellen auch mit moderat ineffizienter Implementation kaum zur Gesamtlaufzeit beitragen.

Pareto-Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

- Allgemeine statistische Regel aus der Volkswirtschaftslehre
  - benannt nach ihrem Schöpfer: Vilfredo Pareto
- Übertragung auf das Thema effiziente Software:
  - Nur wenige Stellen im Quelltext sind für den Löwenanteil von Laufzeit / Speicherverbrauch / Netzwerkbelastung / Ressourcenreservierung verantwortlich.
- Nur eine *heuristische* Regel!
  - Aber: in der Praxis bewährt
- Konsequenz für Effizienzverbesserungen von Software:
  - Erst prüfen, wo genau die Effizienzverluste wirklich auftreten.
  - Sich bei Effizienzverbesserungen auf diese Stellen konzentrieren.

Ich denke, man kann das Wort Heuristik, so wie es hier gebraucht wird, gut und gerne als gehobenes Synonym für Faustregel verstehen.

Pareto-Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

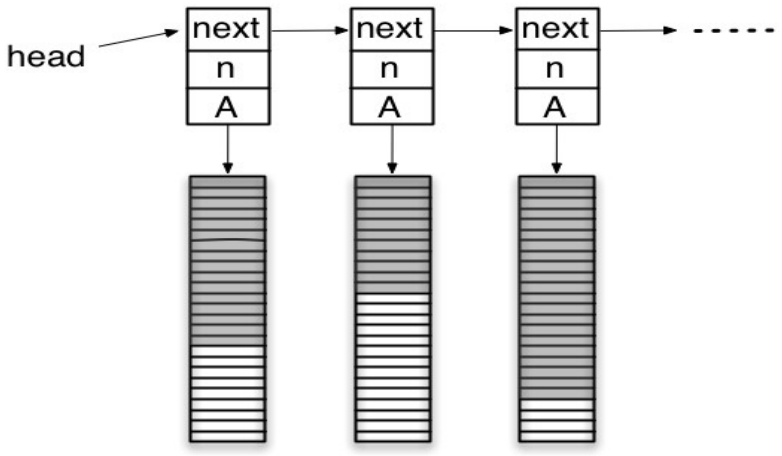
- Allgemeine statistische Regel aus der Volkswirtschaftslehre
  - benannt nach ihrem Schöpfer: Vilfredo Pareto
- Übertragung auf das Thema effiziente Software:
  - Nur wenige Stellen im Quelltext sind für den Löwenanteil von Laufzeit / Speicherverbrauch / Netzwerkbelastung / Ressourcenreservierung verantwortlich.
- Nur eine *heuristische* Regel!
  - Aber: in der Praxis bewährt
- Konsequenz für Effizienzverbesserungen von Software:
  - Erst prüfen, wo genau die Effizienzverluste wirklich auftreten.
  - Sich bei Effizienzverbesserungen auf diese Stellen konzentrieren.

Das ist der grobe Fahrplan dann für das Hauptthema: Erst müssen wir klären, wie wir überhaupt bestimmen können, was Sache ist; und dann müssen wir die entscheidenden Punkte eben verbessern.

Aber vor dem Hauptaspekt Laufzeit, wie gesagt, erst einmal die anderen drei Aspekte, als erstes der Aspekt Speicherplatz.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

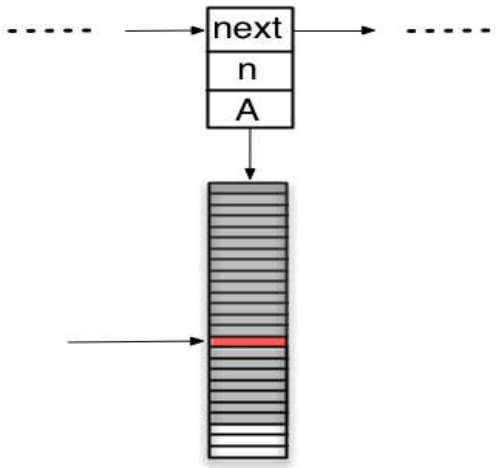


***Erinnerung*: Gegen Ende von Kapitel 07 hatten wir uns diese alternative Implementation des Interface List angeschaut.**

Dies ist ein Beispiel für einen wesentlichen Punkt, auf den man immer achten sollte: nicht unnötig Objekte festhalten, die man eigentlich nicht mehr braucht.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT




```
n--;
for ( int j = i; j < p.n; j++ )
    p.a[j] = p.a[j+1];
p.a[p.n] = null;
```

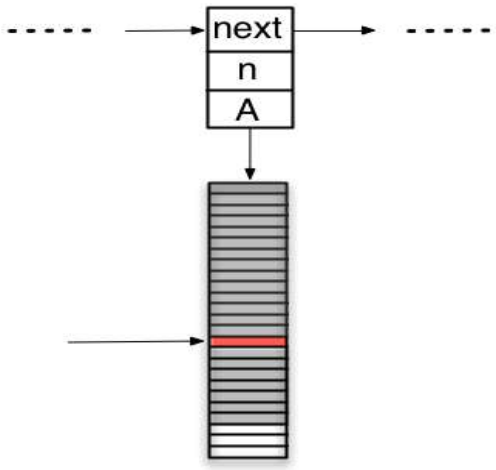
Das unnötige Festhalten von Speicherplatz hatten wir dort schon thematisiert, nämlich beim Entfernen eines Elements aus der Liste. Diese Folie fasst zwei aufeinanderfolgende, zusammengehörige Folien von dort zusammen.

***Erinnerung*: In Kapitel 06, Abschnitt zu den Einschränkungen bei Generics in Java, hatten wir gesehen, dass primitive Datentypen nicht als Instanziierungen von generischen Typparametern möglich sind.**

Anders herum gesagt: Die Elemente einer generischen Liste sind in Java immer Referenzen, das heißt, verweisen auf Objekte, die außerhalb dieser Listenstruktur irgendwo im Speicherplatz angelegt sind.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT




```
n--;
for ( int j = i; j < p.n; j++ )
    p.a[j] = p.a[j+1];
p.a[p.n] = null;
```

Solange der Verweis auf dieses Objekt nicht überschrieben wird, kann der für dieses Objekt reservierte Speicherplatz nicht vom Garbage Collector wieder freigegeben werden, auch wenn das Objekt außerhalb dieser Listenstruktur längst aufgegeben ist. Aus diesem – und keinem anderen – Grund hatten wir die Arraykomponente, die nun nicht mehr zur Liste gehört, auf null gesetzt.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Heap Size:

```
ListItem<String> head = null;
while ( true ) {
    ListItem<String> newItem = new ListItem<String>();
    newItem.key = new String ( “Hello“ );
    newItem.next = head;
    head = newItem;
}
```

Für alle Objekte zusammengenommen, die mit Operator new eingerichtet werden, ist nur ein begrenzter Bereich im Speicherplatz reserviert. Dieser Bereich wird bei Java und auch bei anderen Sprachen im Allgemeinen der *Heap* genannt.

Für jedes neu einzurichtende Objekt muss das Laufzeitsystem in diesem Bereich noch einen ausreichend großen Speicherplatz finden und notfalls den Garbage Collector aufrufen, um unnötig reservierten Speicherplatz wieder freizugeben und wiederzuverwenden. Das hilft aber natürlich nicht in einem Fall wie hier, in dem wir mutwillig den Heap an seine Grenze bringen. Sobald kein Platz mehr gefunden wird, wird ein OutOfMemoryError geworfen.

***Erinnerung*: Abschnitt zu den Klassen Throwable und Error in Kapitel 05, Abschnitt zu eigener LinkedList-Klasse in Kapitel 07.**

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory Leaks in C und C++

```
int* p = malloc ( sizeof(int) );
*p = 1;
.....
free ( p );
*p = 2;
free ( p );
p = malloc ( sizeof(int) );
```

```
int* p = new int;
int* a = new int [ 357 ];
.....
delete p;
delete [ ] a;
```

Nicht jede Programmiersprache hat einen Garbage Collector in ihrem Laufzeitsystem eingebaut. Zum Beispiel die Sprachen C und C++ haben keinen Garbage Collector. Dort muss man jeden Speicherbereich, den man sich vom Laufzeitsystem auf dem Heap hat einrichten lassen, manuell wieder freigeben. Das ist eine massive Fehlerquelle und einer der Gründe, warum C und C++ in vielen kritischen Anwendungsbereichen nicht eingesetzt werden dürfen.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory Leaks in C und C++

```
int* p = malloc ( sizeof(int) );
*p = 1;
.....
free ( p );
*p = 2;
free ( p );
p = malloc ( sizeof(int) );
```

```
int* p = new int;
int* a = new int [ 357 ];
.....
delete p;
delete [ ] a;
```

So sieht es in der Sprache C aus: Der Stern hinter dem Typnamen int sagt aus, dass p keine int-Variable, sondern ein *Verweis* auf ein int-Objekt ist. Im Gegensatz zu Java gibt es in C keine Referenztypen, sondern Objekte jeden Typs können entweder ohne oder mit Stern angelegt werden. Ohne Stern heißt: wie primitive Datentypen in Java, keine Trennung von Objekt und Referenz; mit Stern heißt: wie Referenztypen in Java.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory Leaks in C und C++

```
int* p = malloc ( sizeof(int) );
*p = 1;
.....
free ( p );
*p = 2;
free ( p );
p = malloc ( sizeof(int) );
```

```
int* p = new int;
int* a = new int [ 357 ];
.....
delete p;
delete [ ] a;
```

Und in C muss dann auch noch explizit die Größe des anzulegenden Bereichs angegeben werden.

***Nebenbemerkung*: malloc steht für memory allocation und wird daher m-alloc ausgesprochen.**

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory Leaks in C und C++

```
int* p = malloc ( sizeof(int) );
*p = 1;
.........
free ( p );
*p = 2;
free ( p );
p = malloc ( sizeof(int) );
```

```
int* p = new int;
int* a = new int [ 357 ];
.........
delete p;
delete [ ] a;
```

Wenn das int-Objekt auf dem Heap abgelegt ist und p nur darauf verweist, dann muss das in C auch bei jedem Zugriff mit einem Stern angezeigt werden.

Solche Verweise wie p heißen Zeiger oder englisch Pointer.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory Leaks in C und C++

```
int* p = malloc ( sizeof(int) );
*p = 1;
.....
free ( p );
*p = 2;
free ( p );
p = malloc ( sizeof(int) );
```

```
int* p = new int;
int* a = new int [ 357 ];
.....
delete p;
delete [ ] a;
```

Das ist jetzt der entscheidende Punkt zu Sprachen wie Java: Ohne Garbage Collector muss jedes Objekt auf dem Heap manuell wieder freigegeben werden. Und das muss auch im richtigen Moment passieren.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory Leaks in C und C++

```
int* p = malloc ( sizeof(int) );
*p = 1;
.....
free ( p );
*p = 2;
free ( p );
p = malloc ( sizeof(int) );
```

```
int* p = new int;
int* a = new int [ 357 ];
.....
delete p;
delete [ ] a;
```

Natürlich darf das Objekt erst dann freigegeben werden, wenn es wirklich nicht mehr benötigt wird. Ein lesender oder schreibender Zugriff nach der Freigabe führt zu etwas, das es in Racket und Java nicht gibt: undefiniertes Verhalten.

Undefiniertes Verhalten kann heißen, dass das Programm abbricht oder korrekt terminiert, aber mit falschen Werten. Oder das Programm bricht viel später an einer ganz anderen Stelle ab, die man überhaupt nicht mit dieser Zeile in Verbindung bringen würde. Die Fehlersuche ist dann entsprechend aufwändig und frustrierend.

Es kann auch passieren, dass das Programm fehlerfrei durchläuft. Alles ist möglich bei undefiniertem Verhalten. Leider bedeutet undefiniertes Verhalten auch: Was tatsächlich passiert oder nicht passiert, kann sich von Programmlauf zu Programmlauf ändern, sogar mit denselben Eingabedaten. Ein Fehler, der nicht deterministisch reproduzierbar ist, ist natürlich besonders schwer zu finden.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory Leaks in C und C++

```
int* p = malloc ( sizeof(int) );
*p = 1;
.....
free ( p );
*p = 2;
free ( p );
p = malloc ( sizeof(int) );
```

```
int* p = new int;
int* a = new int [ 357 ];
.....
delete p;
delete [ ] a;
```

Zweimal dasselbe Objekt freigeben führt ebenfalls zu undefiniertem Verhalten.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory Leaks in C und C++

```
int* p = malloc ( sizeof(int) );
*p = 1;
.....
free ( p );
*p = 2;
free ( p );
p = malloc ( sizeof(int) );
```

```
int* p = new int;
int* a = new int [ 357 ];
.....
delete p;
delete [ ] a;
```

Selbstverständlich darf dem Pointer p auch ein anderes Objekt zugewiesen werden. Aber vorher sollte unbedingt free aufgerufen werden, denn das alte Objekt ist ja von dieser Zeile an nicht mehr vom Programm aus erreichbar. Und es gibt, wie gesagt, keinen Garbage Collector, der diesen Speicherplatz dann irgendwann wieder freigibt.

Das nennt man ein Memory Leak. Wenn sich die Memory Leaks häufen, ist irgendwann der gesamte Heap belegt, der nächste Aufruf von malloc findet keinen Platz mehr, und das Programm bricht fehlerhaft ab.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory Leaks in C und C++

```
int* p = malloc ( sizeof(int) );
*p = 1;
.....
free ( p );
*p = 2;
free ( p );
p = malloc ( sizeof(int) );
```

```
int* p = new int;
int* a = new int [ 357 ];
.....
delete p;
delete [ ] a;
```

In C++ ist die Syntax komfortabler, aber das Problem ist prinzipiell dasselbe.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory Leaks in C und C++

```
int* p = malloc ( sizeof(int) );
*p = 1;
.....
free ( p );
*p = 2;
free ( p );
p = malloc ( sizeof(int) );
```

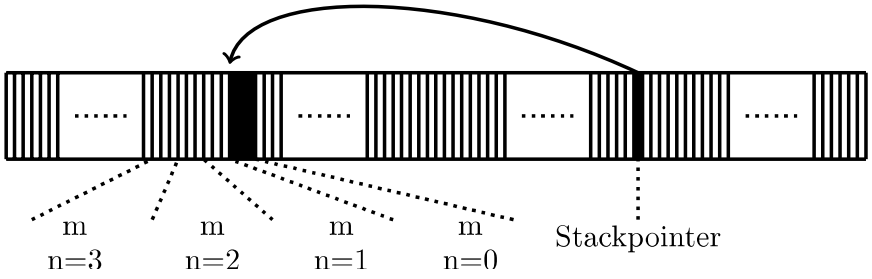
```
int* p = new int;
int* a = new int [ 357 ];
.....
delete p;
delete [ ] a;
```

Auch bei Arrays. Zu beachten ist, dass new und delete zusammenpassen müssen: Das delete für Arrays muss genau dann verwendet werden, wenn das new für Arrays bei der Einrichtung des Objektes verwendet wurde. Eine Verwechslung führt wieder zu undefiniertem Verhalten.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursion: der Call-Stack



Ein spezielles Speicherproblem ergibt sich potentiell bei Rekursion: Für den Call-Stack ist ebenfalls nur ein begrenzter Bereich im Speicherplatz reserviert. Geht die Rekursion zu tief, dann reicht dieser Speicherplatz nicht aus. In Java wird dann ein StackOverflowError wird geworfen.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursion: der Call-Stack

```
( define ( contains? cmp elem list )
    ( cond
        [ ( empty? list )  #f ]
        [ ( cmp elem ( first list ) ) #t ]
        [ else ( contains? cmp elem ( rest list ) ) ] ) )
```

Zum Beispiel bei Rekursionen auf Listen; wie schon in diesem einfachen Beispiel braucht die Liste nur lang genug zu sein, und dann ist der Platz für den Call-Stack aufgebraucht.

Aber Tools wie DrRacket sind bei vielen rekursiven Funktionen in der Lage, die Rekursion intern durch eine Schleife zu ersetzen, so dass dieses Problem vermieden wird. Bei allen Beispielen, die wir in der FOP betrachtet haben und betrachten werden, kann ein Racket-Interpreter das.

Speicherplatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursion: der Call-Stack

```
boolean contains ( Item<T> head, T key, Comparator<T> cmp ) {
    if ( head == null )
        return false;
    if ( cmp ( head.key, key ) == 0 )
        return true;
    return contains ( head.next, key, cmp );
}
```

Hier noch einmal eine analoge Methode in Java. Durch diverse Aspekte der Sprache Java kann der Compiler weit weniger aggressiv optimieren als der Racket-Interpreter. Insbesondere kann man selbst bei einfachen Rekursionen wie dieser nicht davon ausgehen, dass der Compiler sie bei der Übersetzung in eine Schleife umwandeln kann.

TECHNISCHE UNIVERSITÄT DARMSTADT

Netzwerkbelastung

Als nächsten Aspekt von Effizienz die Netzwerkbelastung. Wie gesagt, wird dieses Thema in späteren Lehrveranstaltungen intensiver aufgegriffen, so dass wir hier nur kurz draufschauen.

Netzwerkbelastung

TECHNISCHE UNIVERSITÄT DARMSTADT

Generisches Beispiel: dezentrale Datenhaltung

- **Daten werden redundant auf mehreren oder vielen Netzwerkknoten (Servern) gehalten.**
- **Vorteile:**
  - **Anfragen zu den Daten können je nach momentaner Situation im Netzwerk zu verschiedenen Servern gehen.**
  - **Keine langen Wege im Netzwerk, nur bis zum „nächsten“ Server.**
- **Nachteil: Abgleich der Daten nach Änderungen erfordert möglichst baldige Kommunikation zwischen allen Servern.**
- **Zielkonflikt muss je nach Situation durch einen spezifischen Kompromiss u.a. bei der Platzierung der Server im Netzwerk gelöst werden.**

Ein einzelnes Beispiel, das aber von allgemeinerer Natur ist und sehr häufig in der Praxis vorkommt, mag hier genügen.

Netzwerkbelastung

TECHNISCHE UNIVERSITÄT DARMSTADT

Generisches Beispiel: dezentrale Datenhaltung

- **Daten werden redundant auf mehreren oder vielen Netzwerkknoten (Servern) gehalten.**
- **Vorteile:**
  - **Anfragen zu den Daten können je nach momentaner Situation im Netzwerk zu verschiedenen Servern gehen.**
  - **Keine langen Wege im Netzwerk, nur bis zum „nächsten“ Server.**
- **Nachteil: Abgleich der Daten nach Änderungen erfordert möglichst baldige Kommunikation zwischen allen Servern.**
- **Zielkonflikt muss je nach Situation durch einen spezifischen Kompromiss u.a. bei der Platzierung der Server im Netzwerk gelöst werden.**

Verteilte Datenhaltung ist heutzutage eine sehr häufige Situation. Teilweise werden dieselben Daten dupliziert auf mehreren Servern gehalten. Der Sinn dieser Redundanz ist, dass es zu möglichst jedem Client einen Server mit diesen Daten gibt, der von diesem Client aus direkt oder über möglichst wenige Zwischenstationen erreichbar ist.

Netzwerkbelastung

TECHNISCHE UNIVERSITÄT DARMSTADT

Generisches Beispiel: dezentrale Datenhaltung

- Daten werden redundant auf mehreren oder vielen Netzwerkknoten (Servern) gehalten.
- Vorteile:
  - Anfragen zu den Daten können je nach momentaner Situation im Netzwerk zu verschiedenen Servern gehen.
  - Keine langen Wege im Netzwerk, nur bis zum „nächsten“ Server.
- Nachteil: Abgleich der Daten nach Änderungen erfordert möglichst baldige Kommunikation zwischen allen Servern.
- Zielkonflikt muss je nach Situation durch einen spezifischen Kompromiss u.a. bei der Platzierung der Server im Netzwerk gelöst werden.

Die Vorteile liegen auf der Hand: Wenn es genug Server sind und diese Server auch gut im Netzwerk verteilt sind, dann wird kein Server durch Anfragen überlastet, und auch das Netzwerk wird möglichst gering belastet.

Netzwerkbelastung

TECHNISCHE UNIVERSITÄT DARMSTADT

Generisches Beispiel: dezentrale Datenhaltung

- **Daten werden redundant auf mehreren oder vielen Netzwerkknoten (Servern) gehalten.**
- **Vorteile:**
  - **Anfragen zu den Daten können je nach momentaner Situation im Netzwerk zu verschiedenen Servern gehen.**
  - **Keine langen Wege im Netzwerk, nur bis zum „nächsten“ Server.**
- **Nachteil: Abgleich der Daten nach Änderungen erfordert möglichst baldige Kommunikation zwischen allen Servern.**
- **Zielkonflikt muss je nach Situation durch einen spezifischen Kompromiss u.a. bei der Platzierung der Server im Netzwerk gelöst werden.**

Ein Problem entsteht dann, wenn die redundanten Daten aktualisiert werden. Jede Aktualisierung muss so schnell wie möglich durch das gesamte Netzwerk hindurch zu allen Servern propagiert werden, um Inkonsistenzen in der redundanten Datenhaltung zu vermeiden.

Netzwerkbelastung

TECHNISCHE UNIVERSITÄT DARMSTADT

Generisches Beispiel: dezentrale Datenhaltung

- **Daten werden redundant auf mehreren oder vielen Netzwerkknoten (Servern) gehalten.**
- **Vorteile:**
  - **Anfragen zu den Daten können je nach momentaner Situation im Netzwerk zu verschiedenen Servern gehen.**
  - **Keine langen Wege im Netzwerk, nur bis zum „nächsten“ Server.**
- **Nachteil: Abgleich der Daten nach Änderungen erfordert möglichst baldige Kommunikation zwischen allen Servern.**
- **Zielkonflikt muss je nach Situation durch einen spezifischen Kompromiss u.a. bei der Platzierung der Server im Netzwerk gelöst werden.**

Je nach den Umständen – wie das Netzwerk strukturiert ist und wie viele Aktualisierungen es im Vergleich zu rein lesenden Anfragen gibt –, muss die Zahl an Servern und ihre Platzierung in der Netzwerktopologie geeignet gewählt und vielleicht später auch noch einmal modifiziert werden, sobald es erste Erfahrungen mit der Auslastung gibt beziehungsweise falls sich etwas an den Umständen ändert.

TECHNISCHE UNIVERSITÄT DARMSTADT

Reservierung von Ressourcen

Der letzte Aspekt, bevor es an die Laufzeit geht. Auch für diesen Aspekt schauen wir uns nur kurz ein einzelnes Beispiel an, und zwar eines, das wir schon kennen.

Reservierung von Ressourcen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileOutputStream out1 = new FileOutputStream ( fileName );
      BufferedOutputStream out2 = new BufferedOutputStream ( out1 );
      PrintStream out3 = new PrintStream ( out2 ) ) {

   out3.print ( “pi = “ );
   out3.print ( 3.14 );
   out3.println ( ´!´ );

} catch ( FileNotFoundException | IOException exc ) .....
```

***Erinnerung*: Wir hatten dieses Beispiel schon in Kapitel 08, Abschnitt zu Bytedaten.**

Reservierung von Ressourcen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileOutputStream out1 = new FileOutputStream ( fileName );
      BufferedOutputStream out2 = new BufferedOutputStream ( out1 );
      PrintStream out3 = new PrintStream ( out2 ) ) {
   out3.print ( “pi = “ );
   out3.print ( 3.14 );
   out3.println ( ´!´ );
} catch ( FileNotFoundException | IOException exc ) .....
```

Für dieses Konstrukt hatten Sie den Namen try-with-resources kennen gelernt. Es sorgt dafür, dass die in runden Klammern eingerichteten Ressourcen nach dem Ende des try-catch-Blocks auf jeden Fall wieder geschlossen werden, egal *wie* der try-catch-Block verlassen wurde. Die Klassen müssen dafür das Interface AutoCloseable implementieren; dessen Methode close wird aufgerufen.

Damit wird der Zugriff auf die im Kopf eingerichteten Ressourcen auf das absolut notwendige zeitliche Minimum beschränkt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Laufzeit messen

Nun kommen wir zum Hauptaspekt des Themas Effizienz, der Laufzeit. Zuerst schauen wir uns an, wie wir überhaupt die Laufzeit einzelner Bestandteile eines Programms messen können.

In diesem Zusammenhang sei noch einmal auf die Zusammenfassung „Programme und Prozesse“ verwiesen.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende Unterscheidungen:

- **Gewöhnliche Zeit („wall clock time“) vs. CPU-Zeit**
  - **Gewöhnliche Zeit: wie viel Zeit seit dem Start vergangen ist.**
  - **CPU-Zeit: wie viel Rechenzeit dem Prozess bzw. Thread bisher zugestanden wurde.**
- **User Time vs. System Time**
  - **User Time: bislang verbrauchte CPU-Zeit für den eigentlichen Prozess.**
  - **System Time: vom System bislang für diesen Prozess verbrauchte CPU-Zeit.**

→ CPU-Zeit = User Time + System Time

Aber zuallererst müssen wir klären, was überhaupt mit Laufzeit gemeint ist.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende Unterscheidungen:

- **Gewöhnliche Zeit („wall clock time“) vs. CPU-Zeit**
  - **Gewöhnliche Zeit: wie viel Zeit seit dem Start vergangen ist.**
  - **CPU-Zeit: wie viel Rechenzeit dem Prozess bzw. Thread bisher zugestanden wurde.**
- **User Time vs. System Time**
  - **User Time: bislang verbrauchte CPU-Zeit für den eigentlichen Prozess.**
  - **System Time: vom System bislang für diesen Prozess verbrauchte CPU-Zeit.**

→ CPU-Zeit = User Time + System Time

Mehrere, grundlegend verschiedene Zeiten lassen sich für einen Prozess abrufen.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende Unterscheidungen:

- Gewöhnliche Zeit („wall clock time“) vs. CPU-Zeit
  - Gewöhnliche Zeit: wie viel Zeit seit dem Start vergangen ist.
  - CPU-Zeit: wie viel Rechenzeit dem Prozess bzw. Thread bisher zugestanden wurde.
- User Time vs. System Time
  - User Time: bislang verbrauchte CPU-Zeit für den eigentlichen Prozess.
  - System Time: vom System bislang für diesen Prozess verbrauchte CPU-Zeit.

→ CPU-Zeit = User Time + System Time

Zunächst einmal ist zwischen der normalen Zeit, die man mit der Uhr misst, und der CPU-Zeit zu unterscheiden. Einem Prozess wird eine CPU ja immer nur für kurze Zeitintervalle zugeteilt, dazwischen ist der Prozess schlafengelegt. Diese Schlafenszeiten sind in der gewöhnlichen Zeit mit dabei, in der CPU-Zeit nicht. Die CPU-Zeit ist also in der Regel die interessantere für Effizienzbetrachtungen.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende Unterscheidungen:

- **Gewöhnliche Zeit („wall clock time“) vs. CPU-Zeit**
  - **Gewöhnliche Zeit: wie viel Zeit seit dem Start vergangen ist.**
  - **CPU-Zeit: wie viel Rechenzeit dem Prozess bzw. Thread bisher zugestanden wurde.**
- **User Time vs. System Time**
  - **User Time: bislang verbrauchte CPU-Zeit für den eigentlichen Prozess.**
  - **System Time: vom System bislang für diesen Prozess verbrauchte CPU-Zeit.**

→ CPU-Zeit = User Time + System Time

Wir werden gleich sehen, dass wir die CPU-Zeit genauer gesagt für jeden Thread einzeln messen können.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende Unterscheidungen:

- **Gewöhnliche Zeit („wall clock time“) vs. CPU-Zeit**
  - **Gewöhnliche Zeit: wie viel Zeit seit dem Start vergangen ist.**
  - **CPU-Zeit: wie viel Rechenzeit dem Prozess bzw. Thread bisher zugestanden wurde.**
- **User Time vs. System Time**
  - **User Time: bislang verbrauchte CPU-Zeit für den eigentlichen Prozess.**
  - **System Time: vom System bislang für diesen Prozess verbrauchte CPU-Zeit.**

→ CPU-Zeit = User Time + System Time

Die CPU-Zeit wiederum setzt sich aus zwei Zeiten zusammen, die jede für sich interessant ist. Daher ist es in Java und vielen anderen Sprachen so eingerichtet, dass jede der beiden Zeiten separat bestimmt werden kann. Die User Time ist die Zeit in Summe, in der das *eigentliche* Programm ausgeführt wird. Die System Time hingegen ist die Zeit, in der die Laufzeitumgebung im Dienste des Programms arbeitet.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Wall clock time messen:

```
double sum ( double [ ] a ) {
   double result = 0;
   long startTime = System.currentTimeMillis();
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   System.out.print ( System.currentTimeMillis() – startTime );
   return result;
}
```

Achtung: Methode nanoTime statt currentTimeMillis liefert zwar Anzahl Nanosekunden, aber nicht unbedingt präzise!

Als erstes sehen wir uns an, wie die gewöhnliche Zeit gemessen werden kann. Ein kleines Beispiel reicht dafür aus, denn mehr ist dazu nicht zu sagen. Das Beispiel summiert einfach nur die Werte in einem Array auf, und wir wollen messen, wieviel Zeit während der Ausführung der Schleife vergeht.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Wall clock time messen:

```
double sum ( double [ ] a ) {
   double result = 0;
   long startTime = System.currentTimeMillis();
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   System.out.print ( System.currentTimeMillis() – startTime );
   return result;
}
```

Achtung: Methode nanoTime statt currentTimeMillis liefert zwar Anzahl Nanosekunden, aber nicht unbedingt präzise!

Wir kennen die Klasse System aus dem Package java.lang schon aus verschiedenen Kontexten. Diese Methode nun liefert als long die Anzahl der Millisekunden seit Beginn des ersten Januar 1970 zurück.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Wall clock time messen:

```
double sum ( double [ ] a ) {
   double result = 0;
   long startTime = System.currentTimeMillis();
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   System.out.print ( System.currentTimeMillis() – startTime );
   return result;
}
```

Achtung: Methode nanoTime statt currentTimeMillis liefert zwar Anzahl Nanosekunden, aber nicht unbedingt präzise!

Die Differenz aus zwei dieser Zeitangaben ergibt natürlich die zwischen den beiden Abrufen vergangene Zeit in Millisekunden.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Wall clock time messen:

```
double sum ( double [ ] a ) {
   double result = 0;
   long startTime = System.currentTimeMillis();
   for ( int i = 0; i < a.length; i++ )
      result += a[i];
   System.out.print ( System.currentTimeMillis() – startTime );
   return result;
}
```

Achtung: Methode nanoTime statt currentTimeMillis liefert zwar Anzahl Nanosekunden, aber nicht unbedingt präzise!

Sie werden bei Klasse System noch eine zweite Methode finden, die dasselbe leistet, aber feingranularer. Die Maßeinheit ist Nanosekunden, aber Sie können nicht davon ausgehen, dass der zurückgelieferte Wert wirklich auf die Nanosekunde genau ist. Die einzige plattformübergreifende Garantie ist, dass die Rückgabe der Methode nanoTime auf *Milli*sekunden genau ist.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit messen:

```
import java.lang.management.*;
boolean printSystemTime () {
   ThreadMXBean bean = ManagementFactory.getThreadMXBean();
   if ( ! bean.isCurrentThreadCpuTimeSupported() )
      return false;
  long startTimeCpu = bean.getCurrentThreadCpuTime();
  long startTimeUser = bean.getCurrentThreadUserTime();
   for ( int i = 0; i < 1000000; i++ )
      new double [ i ];
   long cpuTime = bean.getCurrentThreadCpuTime() – startTimeCpu;
   long userTime = bean.getCurrentThreadUserTime() - startTimeUser;
   System.out.print ( cpuTime – userTime );
  return true;
 }
```

Jetzt kommen wir zur CPU-Zeit, genauer: zur CPU-Zeit des Threads, in dem wir die Zeit messen. Besteht der Prozess aus nur einem Thread, dann ist das natürlich die CPU-Zeit des gesamten Prozesses.

Wie man die Zeit in anderen Threads messen kann, sehen wir gleich im nächsten Beispiel.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit messen:

```
import java.lang.management.*;
boolean printSystemTime () {
    ThreadMXBean bean = ManagementFactory.getThreadMXBean();
    if ( ! bean.isCurrentThreadCpuTimeSupported() )
        return false;
   long startTimeCpu = bean.getCurrentThreadCpuTime();
   long startTimeUser = bean.getCurrentThreadUserTime();
    for ( int i = 0; i < 1000000; i++ )
        new double [ i ];
    long cpuTime = bean.getCurrentThreadCpuTime() – startTimeCpu;
    long userTime = bean.getCurrentThreadUserTime() - startTimeUser;
    System.out.print ( cpuTime – userTime );
   return true;
  }
```

Dazu brauchen wir Funktionalität aus einem Package, das wir bisher nicht betrachtet hatten.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit messen:

```
import java.lang.management.*;
boolean printSystemTime () {
   ThreadMXBean bean = ManagementFactory.getThreadMXBean();
   if ( ! bean.isCurrentThreadCpuTimeSupported() )
      return false;
  long startTimeCpu = bean.getCurrentThreadCpuTime();
  long startTimeUser = bean.getCurrentThreadUserTime();
   for ( int i = 0; i < 1000000; i++ )
      new double [ i ];
   long cpuTime = bean.getCurrentThreadCpuTime() – startTimeCpu;
   long userTime = bean.getCurrentThreadUserTime() - startTimeUser;
   System.out.print ( cpuTime – userTime );
  return true;
 }
```

Hierdurch wird ein Objekt zurückgeliefert, das einiges an Informationen für diesen Thread bietet.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit messen:

```
import java.lang.management.*;
boolean printSystemTime () {
   ThreadMXBean bean = ManagementFactory.getThreadMXBean();
   if ( ! bean.isCurrentThreadCpuTimeSupported() )
      return false;
  long startTimeCpu = bean.getCurrentThreadCpuTime();
  long startTimeUser = bean.getCurrentThreadUserTime();
   for ( int i = 0; i < 1000000; i++ )
      new double [ i ];
   long cpuTime = bean.getCurrentThreadCpuTime() – startTimeCpu;
   long userTime = bean.getCurrentThreadUserTime() - startTimeUser;
   System.out.print ( cpuTime – userTime );
  return true;
 }
```

Die Variable bean verweist auf ein Objekt, das für den hier ausgeführten Thread eingerichtet wurde. Diese Methode nun liefert die gesamte CPU-Zeit, die dieser Thread seit seiner Einrichtung bis dahin verbraucht hat.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit messen:

```
import java.lang.management.*;
boolean printSystemTime () {
   ThreadMXBean bean = ManagementFactory.getThreadMXBean();
   if ( ! bean.isCurrentThreadCpuTimeSupported() )
      return false;
  long startTimeCpu = bean.getCurrentThreadCpuTime();
  long startTimeUser = bean.getCurrentThreadUserTime();
   for ( int i = 0; i < 1000000; i++ )
      new double [ i ];
   long cpuTime = bean.getCurrentThreadCpuTime() – startTimeCpu;
   long userTime = bean.getCurrentThreadUserTime() - startTimeUser;
   System.out.print ( cpuTime – userTime );
  return true;
 }
```

Will man die CPU-Zeit einer Passage im Quelltext messen – so wie hier die Schleife –, dann lässt man sich die CPU-Zeit einmal unmittelbar davor und einmal unmittelbar danach ausgeben und nimmt die Differenz beider Werte.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit messen:

```
import java.lang.management.*;
boolean printSystemTime () {
   ThreadMXBean bean = ManagementFactory.getThreadMXBean();
   if ( ! bean.isCurrentThreadCpuTimeSupported() )
      return false;
  long startTimeCpu = bean.getCurrentThreadCpuTime();
  long startTimeUser = bean.getCurrentThreadUserTime();
   for ( int i = 0; i < 1000000; i++ )
      new double [ i ];
   long cpuTime = bean.getCurrentThreadCpuTime() – startTimeCpu;
   long userTime = bean.getCurrentThreadUserTime() - startTimeUser;
   System.out.print ( cpuTime – userTime );
  return true;
 }
```

Dasselbe funktioniert auch speziell mit der User Time.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit messen:

```
import java.lang.management.*;
boolean printSystemTime () {
   ThreadMXBean bean = ManagementFactory.getThreadMXBean();
   if ( ! bean.isCurrentThreadCpuTimeSupported() )
      return false;
  long startTimeCpu = bean.getCurrentThreadCpuTime();
  long startTimeUser = bean.getCurrentThreadUserTime();
   for ( int i = 0; i < 1000000; i++ )
      new double [ i ];
   long cpuTime = bean.getCurrentThreadCpuTime() – startTimeCpu;
   long userTime = bean.getCurrentThreadUserTime() - startTimeUser;
   System.out.print ( cpuTime – userTime );
  return true;
 }
```

Für die System Time gibt es keine eigene Methode. Um die System Time zu erhalten, kann man aber einfach die User Time von der CPU-Zeit abziehen. Gemäß ihrem Namen ist das auch das die Aufgabe der Methode, die wir auf dieser Folie geschrieben haben.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit messen:

```
import java.lang.management.*;
boolean printSystemTime () {
   ThreadMXBean bean = ManagementFactory.getThreadMXBean();
   if ( ! bean.isCurrentThreadCpuTimeSupported() )
      return false;
  long startTimeCpu = bean.getCurrentThreadCpuTime();
  long startTimeUser = bean.getCurrentThreadUserTime();
   for ( int i = 0; i < 1000000; i++ )
      new double [ i ];
   long cpuTime = bean.getCurrentThreadCpuTime() – startTimeCpu;
   long userTime = bean.getCurrentThreadUserTime() - startTimeUser;
   System.out.print ( cpuTime – userTime );
  return true;
 }
```

Nicht auf jeder Plattform ist diese Art von Zeitmessung möglich. Ob die CPU-Zeit und die User Time abgefragt werden können, wird durch diese Methode angezeigt. Fragt man die Zeiten ab, obwohl das auf dieser Plattform nicht geht, wird eine UnsupportedOperationException geworfen.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit in anderen Threads messen:

```
Runnable [ ] runnables = new Runnable [ numberOfThreads ];
.....
Thread [ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) [
   thread[i] = new Thread ( runnables[i] )
   thread[i].start();
}
.....
```

Bisher haben wir nur Zeiten im eigenen Thread gemessen. Man kann aber auch die Zeiten anderer Threads desselben Prozesses abfragen.

***Erinnerung*: Kapitel 09, Abschnitt zu Threads.**

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit in anderen Threads messen:

```
Runnable [ ] runnables = new Runnable [ numberOfThreads ];
.....
Thread [ ] threads = new Thread [ numberOfThreads ];
for ( int i = 0; i < numberOfThreads; i++ ) [
   thread[i] = new Thread ( runnables[i] )
   thread[i].start();
}
.....
```

Zu Demonstrationszwecken richten wir uns ein Array von Threads ein. Jeder Thread wird mit einem eigenen Runnable-Objekt gestartet. Diese sind ebenfalls in einem Array organisiert. Wir die einzelnen Runnable-Objekte eingerichtet werden, interessiert uns hier nicht und ist deshalb nur durch Punkte angedeutet.

Die eigentliche Zeitmessung folgt auf der nächsten Folie.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit in anderen Threads messen:

```
.........
long totalTime = 0;
for ( Thread thread : threads ) {
   long time = bean.getThreadCpuTime ( thread.getID() );
   if ( time != -1 )
      totalTime += time;
}
```

Hier sehen Sie den gesamten Quelltext, um die bislang verbrauchten CPU-Zeiten aller Threads im Array aufzusummieren.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit in anderen Threads messen:

```
.........
long totalTime = 0;
for ( Thread thread : threads ) {
    long time = bean.getThreadCpuTime ( thread.getID() );
    if ( time != -1 )
        totalTime += time;
}
```

Auf die sattsam bekannte Weise gehen wir alle Threads im Array durch.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit in anderen Threads messen:

```
.........
long totalTime = 0;
for ( Thread thread : threads ) {
   long time = bean.getThreadCpuTime ( thread.getID() );
   if ( time != -1 )
      totalTime += time;
}
```

***Erinnerung*: In Kapitel 09, Abschnitt zu Threads, hatten wir schon die Methode getID kennen gelernt, die für einen Thread seine interne ID zurückliefert.**

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit in anderen Threads messen:

```
.........
long totalTime = 0;
for ( Thread thread : threads ) {
   long time = bean.getThreadCpuTime ( thread.getID() );
   if ( time != -1 )
      totalTime += time;
}
```

Mit der farblich unterlegten Methode kann man dann die bisher verbrauchte CPU-Zeit eines Threads auslesen, dessen ID man kennt. Völlig analog kann man hier wieder die User Time abfragen.

Voraussetzung ist auch hier wieder, dass die Messung der CPU-Zeit auf der Plattform, auf der der Prozess läuft, unterstützt wird, sonst wird auch hier wieder eine UnsupportedOperationException geworfen.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU-Zeit in anderen Threads messen:

```
.........
long totalTime = 0;
for ( Thread thread : threads ) {
   long time = bean.getThreadCpuTime ( thread.getID() );
   if ( time != -1 )
      totalTime += time;
}
```

Falls der Thread schon terminiert ist, dann wird statt dessen minus 1 zurückgeliefert. Das hier vorgestellte Demonstrationsbeispiel summiert also genauer gesagt nicht unbedingt die CPU-Zeiten *aller* Threads im Array auf, sondern nur derjenigen, die nicht zwischenzeitlich beendet wurden.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Profiler (am Beispiel Java):

- **JVM Profilers: CPU-Zeit, Methodenaufrufe und Speichernutzung direkt an der JVM**
  - **Privilegierter Zugriff notwendig**
- **Instrumentalisierende Profiler: Fügen weiteren Code zum Monitoring ein**
  - **Begriff: *instrumentalisieren* den zu überwachenden Code**
- **Application Performance Monitoring (APM): instrumentalisierende Profiler mit minimalistischer Datensammlung**
  - **Geringer Laufzeit-Overhead**
  - **Geeignet für Monitoring im produktiven Einsatz**

Zum Schluss des Abschnitts zur Laufzeitmessung soll noch kurz erwähnt werden, dass es neben den in der Sprache eingebauten Möglichkeiten zur Zeitmessung auch separate Tools dafür gibt, meist kommerzielle. Wir schauen uns speziell Java an; das hier Gesagte lässt sich aber auch auf andere Sprachen mehr oder weniger übertragen.

Ein Profiler klinkt sich in den Prozess geeignet ein und sammelt Daten über die Ausführung des Prozesses.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Profiler (am Beispiel Java):

- **JVM Profilers: CPU-Zeit, Methodenaufrufe und Speichernutzung direkt an der JVM**
  - **Privilegierter Zugriff notwendig**
- **Instrumentalisierende Profiler: Fügen weiteren Code zum Monitoring ein**
  - **Begriff: *instrumentalisieren* den zu überwachenden Code**
- **Application Performance Monitoring (APM): instrumentalisierende Profiler mit minimalistischer Datensammlung**
  - **Geringer Laufzeit-Overhead**
  - **Geeignet für Monitoring im produktiven Einsatz**

Die erste Art von Profilern schaut ganz tief hinein in die Java Virtual Machine und muss dafür entsprechenden Zugriff haben.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Profiler (am Beispiel Java):

- JVM Profilers: CPU-Zeit, Methodenaufrufe und Speichernutzung direkt an der JVM
  - Privilegierter Zugriff notwendig
- Instrumentalisierende Profiler: Fügen weiteren Code zum Monitoring ein
  - Begriff: *instrumentalisieren* den zu überwachenden Code
- Application Performance Monitoring (APM): instrumentalisierende Profiler mit minimalistischer Datensammlung
  - Geringer Laufzeit-Overhead
  - Geeignet für Monitoring im produktiven Einsatz

Die zweite Art von Profilern fügt zusätzlichen Code in die Übersetzung des zu instrumentalisierenden Codes ein. Je nachdem, was der Profiler an Möglichkeiten anbietet, lassen sich sehr vielfältige Daten aus dem überwachten Prozess gewinnen. Allerdings produziert das in der Regel einen erheblichen Laufzeit-Overhead, also zusätzliche Laufzeit durch Ausführung des Monitoring.

Laufzeit messen

TECHNISCHE UNIVERSITÄT DARMSTADT

Profiler (am Beispiel Java):

- **JVM Profilers: CPU-Zeit, Methodenaufrufe und Speichernutzung direkt an der JVM**
  - **Privilegierter Zugriff notwendig**
- **Instrumentalisierende Profiler: Fügen weiteren Code zum Monitoring ein**
  - **Begriff: *instrumentalisieren* den zu überwachenden Code**
- **Application Performance Monitoring (APM): instrumentalisierende Profiler mit minimalistischer Datensammlung**
  - **Geringer Laufzeit-Overhead**
  - **Geeignet für Monitoring im produktiven Einsatz**

Einen Kompromiss zwischen Laufzeit-Overhead und Datenreichtum bieten APM-Werkzeuge. Diese sind dann auch im realen Einsatz der Software verwendbar, da ihr Laufzeit-Overhead ausreichend klein ist. Der Preis dafür ist, dass die gesammelten Daten eher nur ausreichen, um *Hinweise* zu geben, wo und wie es zu Effizienzverlusten kommt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Laufzeitverbesserungen

Nachdem wir gesehen haben, wie die Laufzeit *gemessen* werden kann, schauen wir uns als nächstes ein paar einfache, gängige Beispiele an, wie sie *verbessert* werden kann. Dabei konzentrieren wir uns auf Java, aber vieles lässt sich auch analog auf andere Programmiersprachen übertragen.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Assert-Anweisungen abschalten

```
if ( n < 0 || n % 2 == 1 )
    throw new AssertionError ( “Bad n!“ );


assert n >= 0 && n % 2 == 0 : “Bad n!“;
```

Als erstes ein schon betrachtetes Beispiel.

***Erinnerung*: Wir hatten die Assert-Anweisung in Kapitel 05, Abschnitt zu Throwable, Error und Assert-Anweisung gesehen. Dort hatten wir gesehen, dass Assert-Anweisungen den Vorteil haben, dass man sie über Compiler-Optionen einfach abschalten kann.**

Stellen Sie sich vor, sie haben eine Assert-Anweisung an einer Stelle, die zwar bei einer einzelnen Ausführung sehr wenig Laufzeit kostet, die aber inklusive Assert-Anweisung so häufig ausgeführt wird, dass sie dadurch doch laufzeitkritisch ist. Wenn diese Stelle also pro Ausführung sehr wenig Laufzeit kostet, hat die Assert-Anweisung schon einen gewichtigen Anteil an der Summe der Laufzeiten für alle Ausführungen.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Werte nicht mehrfach berechnen

```
int sum = 0;
for ( int i = 0; i < numberOfIterations(); i++ )
   sum += i;
```

```
final int n = numberOfIterations();
int sum = 0;
for ( int i = 0; i < n; i++ )
   sum += i;
```

Ein häufiges und auch oft vermeidbares Laufzeitproblem ergibt sich, wenn dieselben zeitraubenden Operationen unnötigerweise mehrfach nacheinander ausgeführt werden.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Werte nicht mehrfach berechnen

```
int sum = 0;
for ( int i = 0; i < numberOfIterations(); i++ )
   sum += i;
```

```
final int n = numberOfIterations();
int sum = 0;
for ( int i = 0; i < n; i++ )
   sum += i;
```

Zum Beispiel ist es nicht notwendig, die Begrenzung für den Laufindex in jedem Durchlauf wieder und wieder zu berechnen, falls er sich nicht während der Schleife ändert. Es reicht dann natürlich, diesen einen Wert einmal vor der Schleife zu berechnen und in einer Konstanten zu speichern und dann stattdessen die Konstante zu verwenden.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Werte nicht mehrfach berechnen

```
double m ( double x, double y ) {
    .....
    double a = ..... Math.pow ( x, y ) .....;
    .....
    double b = ..... Math.pow ( x, y ) .....;
    .....
}
```

Ein anderes häufiges Beispiel sehen Sie hier. Es tritt insbesondere in großen, unübersichtlichen Code-Stücken auf. Da kann es leicht passieren, dass man an zwei oder mehr verschiedenen Stellen dasselbe Zwischenergebnis berechnet. Das kann man natürlich wieder genauso vermeiden, nämlich indem man das Ergebnis nur einmal berechnet und in einer Konstanten speichert und dann nur noch die Konstante verwendet.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Werte nicht mehrfach berechnen ... *Achtung Seiteneffekte*!

```
public class X {
   double y;
   public double m ( double d ) {
      y += d;
      return y;
   }
}
```

```
X x = new X();
.........
double a = ..... x.m ( 3.14 );
.........
double b = ..... x.m ( 3.14 );
```

Allerdings müssen wir in einem solchen Fall immer aufpassen, dass die Berechnungsmethode nicht Seiteneffekte hat. Es macht ja einen Unterschied, ob der Seiteneffekt mehrfach oder nur einmal ausgeführt wird. Hier ein einfaches künstliches Beispiel, in dem einmalige oder mehrfache Ausführung der Methode m nicht nur entsprechend häufige Seiteneffekte auf x.y hat, sondern indirekt damit sogar unterschiedliche Rückgaben produziert.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Werte nicht mehrfach berechnen ... *Achtung Seiteneffekte*!

```
public class X {
   double y;
   public double m ( double d ) {
     y += d;
     return y;
   }
}
```

```
X x = new X();
.....
double a = ..... x.m ( 3.14 );
.....
double b = ..... x.m ( 3.14 );
```

Hier passiert der Seiteneffekt, der eine unmittelbare Auswirkung auf den Rückgabewert der Methode hat.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Werte nicht mehrfach berechnen ... *Achtung Seiteneffekte*!

```
public class X {
   double y;
   public double m ( double d ) {
      y += d;
      return y;
   }
}
```

```
X x = new X();
.........
double a = ..... x.m ( 3.14 );
.........
double b = ..... x.m ( 3.14 );
```

Die Konsequenz ist, dass die beiden Aufrufe der Methode doch nicht ohne Änderung der Semantik durch einen einzigen Aufruf ersetzt werden können.

***Erinnerung*: In Kapitel 04d hatten wir unter dem Stichwort referentieller Transparenz schon geklärt, dass Methoden, die wie mathematische Funktionen aussehen, besser seiteneffektfrei sein sollten.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Primitive Datentypen sind schneller!

- **Auf Generizität usw. punktuell verzichten**

```
int n = .....;

Function<Integer,Double> fct = .....:
double y = fct.apply ( n );

IntToDoubleFunction fct = .....;
double y = fct.applyAsDouble ( n );
```

Ein weiterer Punkt, bei dem man häufig Laufzeit sparen kann, ist die Verwendung von primitiven Datentypen anstelle der Wrapper-Klassen.

***Erinnerung*: Wrapper-Klassen wurden am Beginn von Kapitel 06 eingeführt.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Primitive Datentypen sind schneller!

- **Auf Generizität usw. punktuell verzichten**

```
int n = .....;

Function<Integer,Double> fct = .....:
double y = fct.apply ( n );

IntToDoubleFunction fct = .....;
double y = fct.applyAsDouble ( n );
```

Wrapper-Klassen bringen primitive Datentypen in Generics hinein. Will man auf Wrapper-Klassen verzichten, muss man also auf Generizität verzichten.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Primitive Datentypen sind schneller!

- **Auf Generizität usw. punktuell verzichten**

```
int n = .....;

Function<Integer,Double> fct = .....:
double y = fct.apply ( n );

IntToDoubleFunction fct = .....;
double y = fct.applyAsDouble ( n );
```

Das entscheidende Wort ist das hier farblich unterlegte. Laut Pareto-Regel sind Maßnahmen zur Effizienzsteigerung ja nur an einzelnen punktuellen Stellen im Quelltext erfolgversprechend. An solchen Stellen lohnt es sich dann potentiell, auf Generizität zu verzichten.

Das kann allerdings durchaus bedeuten, dass man beispielsweise ganze generische Listen und ähnliches vor Eintritt in eine solche Stelle in eine effizientere Datenstruktur kopiert, darauf dann die zeitkritischen Operationen ausführt und die Ergebnisse dann wieder in „schöne“, aber nicht maximal effiziente Datenstrukturen zurücktransformiert.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Primitive Datentypen sind schneller!

- **Auf Generizität usw. punktuell verzichten**

```
int n = .....;

Function<Integer,Double> fct = .....:
double y = fct.apply ( n );

IntToDoubleFunction fct = .....;
double y = fct.applyAsDouble ( n );
```

Bei der kontinuierlichen Weiterentwicklung der Java-Standardbibliothek war und ist die Verwendung von primitiven Datentypen anstelle der Wrapper-Klassen immer wieder ein Thema, so auch bei diesem Beispiel, das zwar selbst gebastelt, aber den Functional Interfaces in Package java.util.function nachempfunden ist.

***Erinnerung*: Kapitel 04c, Abschnitt zu Functional Interfaces und Lambda-Ausdrücken in Java.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Primitive Datentypen sind schneller!

- **Auf Generizität usw. punktuell verzichten**

```
int n = .....;

Function<Integer,Double> fct = .....:
double y = fct.apply ( n );

IntToDoubleFunction fct = .....;
double y = fct.applyAsDouble ( n );
```

An diesen zwei Stellen passiert Boxing beziehungsweise Unboxing. Zumindest Boxing ist eine potentiell aufwändige Operation, da ein Objekt der Wrapper-Klasse mit Operator new eingerichtet wird.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Primitive Datentypen sind schneller!

- **Auf BigInteger und BigDecimal eher verzichten**

```
BigInteger n = .....;
final BigInteger two = new BigInteger ( "2" );
for ( BigInteger i = BigInteger.ZERO; i.compareTo(n) < 0;
        i = i.add ( BigInteger.ONE )
    if ( i.remainder(two).equals ( BigInteger.ONE ) )
        .....
```

***Erinnerung*: In Racket können Zahlen prinzipiell beliebig groß und beliebig genau werden.**

In Package java.math bietet Java eine analoge Funktionalität. Der Preis ist eine deutlich höhere Laufzeit, was natürlich auch in Racket der Fall ist.

Glücklicherweise kommt man wohl bei fast allen Berechnungen in der Praxis mit den primitiven Datentypen aus, so dass man auf solche Klassen meist ganz gut verzichten kann. Zumal die Quelltexte durch solche Klassen auch nicht leichter lesbar werden, wie schon dieses kleine Beispiel zeigt: Methode compareTo statt Vergleichsoperator, Methode remainder statt Modulo-Operator, und so weiter.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Inlining:

```
public class X {
   private int n;

   .....

   public int getN () {
      return n;
   }

   .....

   public double m1 () {
      return 1 + getN ();
   }

   public double m2 () {
      return 1 + n;
   }
}
```

Methodenaufrufe kosten Laufzeit. An laufzeitkritischen Stellen ist es eine Überlegung wert, ob man nicht den Aufruf einer Methode durch eine äquivalente Anweisungssequenz ersetzen könnte. Das nennt man *Inlining* der Methode.

***Erinnerung*: An verschiedenen Stellen hatten wir schon gesehen, wieviel Aufwand bei einem Methodenaufruf in Java im Hintergrund so getrieben wird, Stichwort Methodentabelle.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Inlining:

```
public class X {
   private int n;

   .....

   public int getN () {
      return n;
   }

   .....
}
```

```
public double m1 () {
   return 1 + getN ();
}

public double m2 () {
   return 1 + n;
}
```

Hier nur ein primitives Beispiel zur Illustration: Anstatt wie eigentlich empfohlen die get-Methode zu benutzen, wird der Attributwert direkt ausgelesen.

In Fällen, in denen die get-Methode mehr macht als nur einen Attributwert auszulesen, muss man dann halt aufpassen, dass man exakt dieselbe Berechnung durchführt. Und sollte man an der Implementation der get-Methode etwas ändern, dann muss an allen Stellen, an denen die get-Methode durch äquivalenten Code ersetzt wurde, jede dieser Änderungen sofort nachvollzogen werden. Wenn die Klasse größer ist, sollte das besser durch einen kurzen Kommentar dokumentiert sein.

***Erinnerung*: In Kapitel 02 hatten wir mit getCenterX und setCenterX von Klasse Circle ein get-/set-Methodenpaar, das nicht einfach nur einen Attributwert liest beziehungsweise setzt.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Unrolling bei Schleifen: `for ( int i = 0; i < a.length; i++ ) a[i] = i;`

```
final int lengthRoundedDown10 = a.length / 10 * 10;
for ( int i = 0; i < lengthRoundedDown10;  ) {
    a[i] = i; i++;
    .....   // Noch achtmal dieselbe Zeile
    a[i] = i; i++;
}
for ( int i = lengthRoundedDown10; i < a.length; i++ )
    a[i] = i;
```

Der Test der Fortsetzungsbedingung im Kopf einer Schleife kostet ebenfalls Laufzeit. Falls der Rumpf pro Durchlauf wenig Laufzeit kostet, kann die Laufzeit für die Fortsetzungsbedingung durchaus ins Gewicht fallen.

Die Idee ist, sich die Auswertung der Fortsetzungsbedingung nicht ganz, aber weitgehend zu sparen, indem man mehrere Durchläufe durch die Schleife zusammenfasst. Das nennt man *Unrolling*.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Unrolling bei Schleifen:

```
for ( int i = 0; i < a.length; i++ )
    a[i] = i;
```

```
final int lengthRoundedDown10 = a.length / 10 * 10;
for ( int i = 0; i < lengthRoundedDown10;  ) {
    a[i] = i; i++;
    .....   // Noch achtmal dieselbe Zeile
    a[i] = i; i++;
}
for ( int i = lengthRoundedDown10; i < a.length; i++ )
    a[i] = i;
```

Dies ist ein Beispiel für eine Schleife, bei der die Auswertung der Fortsetzungsbedingung in punkto Laufzeit durchaus ins Gewicht fallen kann. Diese Schleife transformieren wir unten durch Unrolling mit Faktor 10.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Unrolling bei Schleifen:

```
for ( int i = 0; i < a.length; i++ )
    a[i] = i;
```

```
final int lengthRoundedDown10 = a.length / 10 * 10;
for ( int i = 0; i < lengthRoundedDown10;  ) {
    a[i] = i; i++;
    .....   // Noch achtmal dieselbe Zeile
    a[i] = i; i++;
}
for ( int i = lengthRoundedDown10; i < a.length; i++ )
    a[i] = i;
```

Das heißt, in jedem Durchlauf der Schleife führen wir jetzt zehn Durchläufe der eigentlichen Schleife nacheinander aus. Dabei müssen wir natürlich daran denken, dass die Schleife auch noch eine Fortschaltung hat, nämlich den Laufindex jeweils um 1 zu erhöhen.

Normalerweise ist es aus gutem Grund eher verpönt, zwei Anweisungen in eine Zeile zu schreiben. Beim Unrolling ist es aber vorteilhaft, den gesamten Schleifenrumpf plus Fortschaltung in eine Zeile zu schreiben, damit die einzelnen Durchläufe jeweils auf einen Blick erfasst werden können.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Unrolling bei Schleifen:

```
for ( int i = 0; i < a.length; i++ )
    a[i] = i;
```

```
final int lengthRoundedDown10 = a.length / 10 * 10;
for ( int i = 0; i < lengthRoundedDown10;  ) {
    a[i] = i; i++;
    .....   // Noch achtmal dieselbe Zeile
    a[i] = i; i++;
}
for ( int i = lengthRoundedDown10; i < a.length; i++ )
    a[i] = i;
```

Da die eigentliche Anzahl der Schleifendurchläufe in der Regel *kein* Vielfaches von 10 ist, müssen wir am Ende noch die überzähligen Durchläufe mit einer normalen Schleife erledigen, die erst beim letzten ganzen Vielfachen von 10 einsetzt und die letzten maximal 9 Komponenten des Arrays durchläuft. Maximal 9 Komponenten sollten bei einem ordentlich großen Array sicherlich *nicht* ins Gewicht fallen.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Unrolling bei Schleifen: `for ( int i = 0; i < a.length; i++ ) a[i] = i;`

```
final int lengthRoundedDown10 = a.length / 10 * 10;
for ( int i = 0; i < lengthRoundedDown10;  ) {
    a[i] = i; i++;
    .....   // Noch achtmal dieselbe Zeile
    a[i] = i; i++;
}
for ( int i = lengthRoundedDown10; i < a.length; i++ )
    a[i] = i;
```

Das größte ganzzahlige Vielfache von 10, das kleiner oder gleich der Arraylänge ist, kann man dank ganzzahliger Division mit diesem mathematischen Ausdruck erhalten.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Unrolling bei Schleifen: `for ( int i = 0; i < a.length; i++ ) a[i] = i;`

```
final int lengthRoundedDown10 = a.length / 10 * 10;
for ( int i = 0; i < lengthRoundedDown10;  ) {
    a[i] = i; i++;
    .....   // Noch achtmal dieselbe Zeile
    a[i] = i; i++;
}
for ( int i = lengthRoundedDown10; i < a.length; i++ )
    a[i] = i;
```

Dadurch, dass der Laufindex i zehnmal in jedem Schleifendurchlauf fortgeschaltet wird, durchläuft er im Test der Fortsetzungsbedingung nacheinander die ganzzahligen Vielfachen von 10. Daher bricht die Schleife korrekt in dem Moment ab, wenn der Wert von i gleich dem größten Vielfachen von 10 ist, das kleiner oder gleich der Arraylänge ist.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Unrolling bei Schleifen: `for ( int i = 0; i < a.length; i++ ) a[i] = i;`

```
final int lengthRoundedDown10 = a.length / 10 * 10;
for ( int i = 0; i < lengthRoundedDown10;  ) {
    a[i] = i; i++;
    .....   // Noch achtmal dieselbe Zeile
    a[i] = i; i++;
}
for ( int i = lengthRoundedDown10; i < a.length; i++ )
    a[i] = i;
```

Da die Fortschaltungen des Laufindex innerhalb des Schleifenrumpfes geschehen, bleibt die Stelle, die im Schleifenkopf für die Fortschaltung vorgesehen ist, einfach leer. Wir verwenden dennoch eine for-Schleife, keine while-Schleife, um zu unterstreichen, dass diese Schleife von ihrer Logik her eine Zählschleife ist.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
int n = ( count + 7 ) / 8;
switch ( count % 8 ) {
   case 0: do { *p++ = *q++;
   case 7:      *p++ = *q++;
   case 6:      *p++ = *q++;
   case 5:      *p++ = *q++;
   case 4:      *p++ = *q++;
   case 3:      *p++ = *q++;
   case 2:      *p++ = *q++;
   case 1:      *p++ = *q++;
} while ( – – n  > 0 ); }
```

***Vorgriff*: In Kapitel 14, Abschnitt zum KISS-Prinzip, werden wir dieses Beispiel für Unrolling in der Sprache C genauer betrachten und trotz der unbestreitbaren Kreativität, die darin steckt, eher kritisch sehen.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

StringBuilder bzw. StringBuffer statt + bei String:

```
String str1 = “Hello“ + ‘ ‘ + “ World “ + 123 + “ !“;

StringBuilder str2 = new StringBuilder ( “Hello“ );
str2.append ( 123 );
str2.append ( “ !“ );
str2.insert ( 5, “World“ );
str2.insert ( 5, ‘ ‘ ):
```

→ Analog StringBuffer

In vielen textorientierten Anwendungen geht erhebliche Laufzeit in den schrittweisen Aufbau von Strings. Dafür haben wir bisher nur Klasse String gesehen.

***Erinnerung*: Abschnitt zur Klasse String in Kapitel 04b.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

StringBuilder bzw. StringBuffer statt + bei String:

```
String str1 = “Hello“ + ‘ ‘ + “ World “ + 123 + “ !“;

StringBuilder str2 = new StringBuilder ( “Hello“ );
str2.append ( 123 );
str2.append ( “ !“ );
str2.insert ( 5, “World“ );
str2.insert ( 5, ‘ ‘ ):
```

→ Analog StringBuffer

Anstelle der Methode append kann bekanntlich Operator plus für die Konkatenation verwendet werden, und auch für die übliche Schreibweise für den Aufruf von Operator new gibt es bei Klasse String eine Vereinfachung.

Wichtig für das Thema Laufzeit ist, dass jeder Aufruf von append beziehungsweise Operator plus ein neues String-Objekt erzeugt. In diesem noch eher kleinen Beispiel werden schon drei String-Objekte erzeugt.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

StringBuilder bzw. StringBuffer statt + bei String:

```
String str1 = “Hello“ + ‘ ‘ + “ World “ + 123 + “ !“;

StringBuilder str2 = new StringBuilder ( “Hello“ );
str2.append ( 123 );
str2.append ( “ !“ );
str2.insert ( 5, “World“ );
str2.insert ( 5, ‘ ‘ ):
```

→ Analog StringBuffer

Wieder Name schon suggeriert, ist Klasse StringBuilder speziell dafür entwickelt worden, Strings schrittweise aufzubauen, *ohne* dass in jedem Schritt ein neues String-Objekt erzeugt wird.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

StringBuilder bzw. StringBuffer statt + bei String:

```
String str1 = “Hello“ + ‘ ‘ + “ World “ + 123 + “ !“;

StringBuilder str2 = new StringBuilder ( “Hello“ );
str2.append ( 123 );
str2.append ( “ !“ );
str2.insert ( 5, “World“ );
str2.insert ( 5, ‘ ‘ ):
```

→ Analog StringBuffer

Für das Einrichten eines neuen Objektes von Klasse StringBuffer mit Operator new gibt es *keine* abkürzende Schreibweise.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

StringBuilder bzw. StringBuffer statt + bei String:

```
String str1 = “Hello“ + ‘ ‘ + “ World “ + 123 + “ !“;

StringBuilder str2 = new StringBuilder ( “Hello“ );
str2.append ( 123 );
str2.append ( “ !“ );
str2.insert ( 5, “World“ );
str2.insert ( 5, ‘ ‘ ):
```

→ Analog StringBuffer

Der wesentliche Unterschied zu Klasse String ist, dass nicht ein neues Objekt erzeugt wird, sondern das vorhandene Objekt wird verändert. Klasse StringBuuilder ist so implementiert, dass dies wesentlich schneller geht als die Konkatenation in einem neuen Objekt wie bei Klasse String.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

StringBuilder bzw. StringBuffer statt + bei String:

```
String str1 = “Hello“ + ‘ ‘ + “ World “ + 123 + “ !“;

StringBuilder str2 = new StringBuilder ( “Hello“ );
str2.append ( 123 );
str2.append ( “ !“ );
str2.insert ( 5, “World“ );
str2.insert ( 5, ‘ ‘ ):
```

→ Analog StringBuffer

Bei einer Klasse zur Repräsentation von Zeichenketten, die veränderlich sind, macht auch eine Einfügemethode Sinn. Bei Klasse String gibt es – wie bei Arrays – keine Möglichkeit zum Einfügen oder Entfernen von einzelnen Zeichen.

Der erste Parameter der Einfügemethode von Klasse StringBuilder ist die Position in der schon vorhandenen Zeichenkette, an der der zweite Parameter eingefügt wird. Der erste Parameter kann daher ein Wert zwischen 0 und der Länge des Strings sein. Insbesondere wird im Fall gleich 0 die neue Zeichenkette vorne und im Fall gleich Länge hinten an die alte Zeichenkette angehängt.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

StringBuilder bzw. StringBuffer statt + bei String:

```
String str1 = “Hello“ + ‘ ‘ + “ World “ + 123 + “ !“;

StringBuilder str2 = new StringBuilder ( “Hello“ );
str2.append ( 123 );
str2.append ( “ !“ );
str2.insert ( 5, “World“ );
str2.insert ( 5, ‘ ‘ ):
```

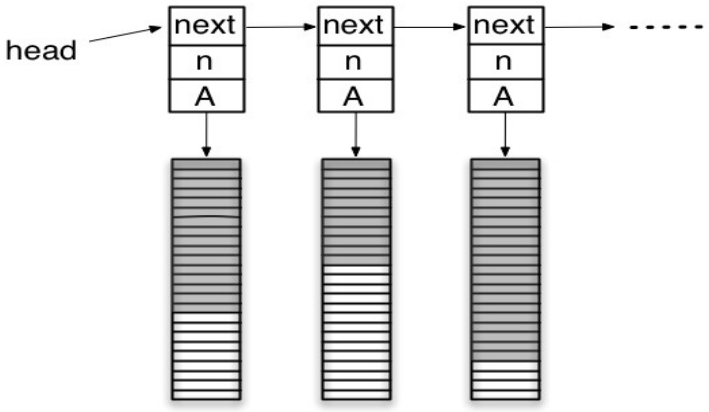
→ Analog StringBuffer

Neben StringBuilder gibt es noch eine Klasse StringBuffer, die im Prinzip genau dasselbe leistet. Der Unterschied ist, dass StringBuilder in der Regel etwas schneller ist, dafür können bei StringBuffer mehrere Threads gleichzeitig an demselben Objekt arbeiten.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Speicherplatz spendieren für bessere Laufzeit:



Oftmals kann man die Laufzeit verbessern, indem man Speicherplatz opfert. Da Speicherplatz heutzutage in normal großen Anwendungen praktisch kein Engpass mehr ist, kann man sich das typischerweise leisten.

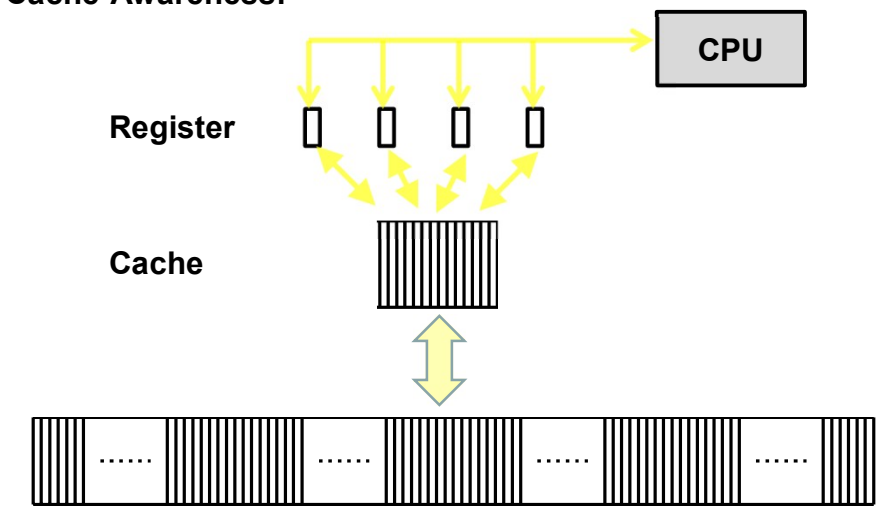
Ein Beispiel dafür hatten wir schon gesehen: eine Implementation des Interface List, bei der die einzelnen Listenelemente auf Arrays verteilt sind. Das Einfügen eines neuen Elements an einer vorgegebenen Position erfordert Operator new nur in dem eher seltenen Fall, dass das entsprechende Array voll ist.

***Erinnerung*: Kapitel 07, Abschnitt zu dieser alternativen Definition von Klasse LinkedList.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Cache-Awareness:



Wir hatten den Speicher bislang immer vereinfacht als ein großes Array von Maschinenwörtern betrachtet, das hatte ausgereicht. Bestimmte Techniken zur Laufzeitverbesserung benötigen allerdings ein realistischeres Bild.

Auf typischen Hardwareplattformen gibt es neben dem großen Hauptspeicher noch einen vergleichsweise kleinen Cache. Aus dem Cache werden die Daten über weitere einzelne Speicherstellen, die Register heißen, in die CPU zur Verarbeitung geladen und die Ergebnisse wieder über Register in den Cache zurückgeschrieben. Zwischen Hauptspeicher und Cache werden nicht einzelne Bits oder Bytes oder Maschinenwörter transferiert, sondern immer gleich eine größere, feste, plattformabhängige Anzahl von Kilobytes.

***Terminologie*: Zugriffe auf Daten, die nicht im Cache sind, nennt man *Cache Misses*. Cache Misses kosten vergleichsweise viel Laufzeit.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Cache-Awareness:

```
for ( int i = 0; i < numberOfRows; i++ )
   for ( int j = 0; j < numberOfColumns; j++ )
      matrix [ i ] [ j ] = 1;

for ( int i = 0; i < numberOfColumns; i++ )
   for ( int j = 0; j < numberOfRows; j++ )
      matrix [ j ] [ i ] = 1;
```

In vielen Fällen sind es relativ kleine Details, die darüber entscheiden, ob der Prozess wenige oder viele Cache Misses produziert. Hier sehen Sie ein häufig zitiertes Beispiel.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Cache-Awareness:

```
for ( int i = 0; i < numberOfRows; i++ )
    for ( int j = 0; j < numberOfColumns; j++ )
        matrix [ i ] [ j ] = 1;

for ( int i = 0; i < numberOfColumns; i++ )
    for ( int j = 0; j < numberOfRows; j++ )
        matrix [ j ] [ i ] = 1;
```

Man kann in erster Näherung davon ausgehen, dass die einzelnen Komponenten eines Arrays hintereinander im Speicher abgelegt sind. Das heißt, wird auf eine Komponente lesend oder schreibend zugegriffen, dann wird nicht nur diese Komponente vom Hauptspeicher in den Cache geladen, sondern gleich eine große Zahl von Komponenten mit aufeinanderfolgenden Indizes. Als Konsequenz sind dann auch die nächsten Komponenten schon im Cache, wenn in einer solchen Schleife darauf zugegriffen wird, und müssen nicht erst zeitraubend in den Cache geladen werden.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Cache-Awareness:

```
for ( int i = 0; i < numberOfRows; i++ )
   for ( int j = 0; j < numberOfColumns; j++ )
      matrix [ i ] [ j ] = 1;

for ( int i = 0; i < numberOfColumns; i++ )
   for ( int j = 0; j < numberOfRows; j++ )
      matrix [ j ] [ i ] = 1;
```

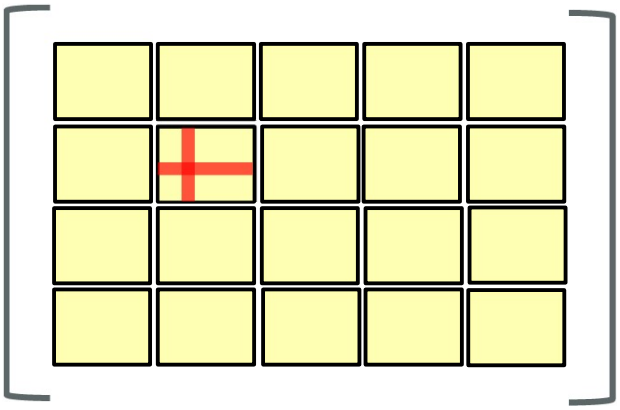
Durchläuft man hingegen eine zeilenweise abgelegte Matrix spaltenweise und ist die Matrix zu groß, um ganz in den Cache zu passen, dann wird hingegen häufiger auf eine Komponente zugegriffen, die nicht im Cache ist. Das kostet natürlich Laufzeit.

Muss man in einer Anwendung Matrizen wirklich wie in der unteren Doppelschleife durchgehen – also übergeordnet spaltenweise und untergeordnet zeilenweise –, dann bietet es sich an, die Matrix so zu definieren, dass zuerst der Spaltenindex kommt.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Cache-Awareness:

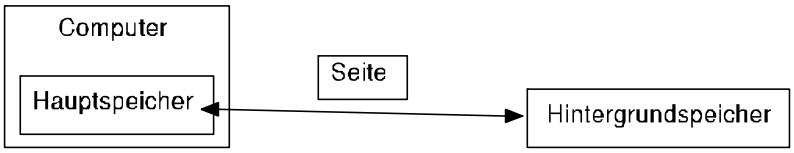


***Nebenbemerkung*: Wenn man auf Matrizen sowohl übergeordnet spaltenweise als auch an anderer Stelle übergeordnet zeilenweise zugreifen muss – zum Beispiel Matrixmultiplikation –, dann hat es sich als Kompromiss bewährt, dass die Matrix sozusagen in Rechtecke zerlegt wird, das heißt, jedes Rechteck enthält alle Einträge, die sowohl in einem bestimmten kleinen Intervall der Spaltenindizes als auch in einem bestimmten kleinen Intervall der Zeilenindizes sind. Diese beiden Intervalle sind so dimensioniert, dass das gesamte Rechteck auf einmal in den Cache geladen wird. Also: Greift man auf einen Matrixeintrag in einem Rechteck zu, dann wird das gesamte Rechteck in den Cache geladen. Und egal ob Sie zeilenweise oder spaltenweise durch die Matrix gehen – zumindest ein kleines Intervall von Zeilenindizes beziehungsweise Spaltenindizes haben Sie dabei immer gleich mit in den Cache eingeladen, was im Bild durch zwei rote Strecken in einem der Rechtecke angedeutet wird.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:



Es geht aber noch weiter: Der lesende oder schreibende Zugriff auf Hintergrundspeicher wie zum Beispiel Festplatten ist eine derart kostspielige Operation, dass man sie fast um jeden Preis minimieren sollte.

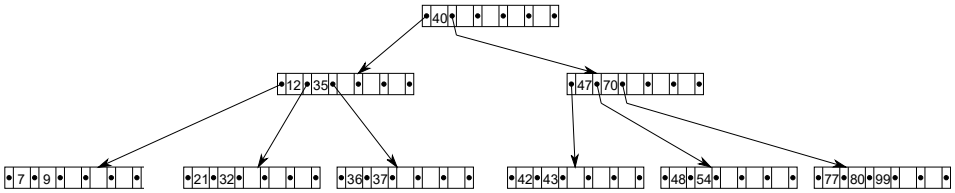
Wie beim Cache wird dieses Ziel typischerweise von der Hardware schon unterstützt. Konkret werden auch hier immer Blöcke von fester Größe hin- und herkopiert. Solche Blöcke heißen *Seiten*.

***Terminologie*: Wann immer auf Information zugegriffen wird, die nicht im Hauptspeicher ist, sondern nur im Hintergrundspeicher, nennt man das ein *Page Fault*.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:



Implementationsinvariante für B-Baum der Ordnung *M*:

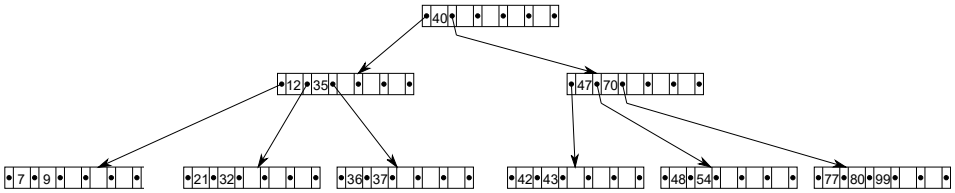
- **In jedem Knoten Platz für 2*M*-1 Keys und 2M Nachfolger**
- **In jedem Knoten außer der Wurzel mindestens M-1 Keys**
- **Alle Blätter auf derselben Höhe**
- **Sortierung der Keys von links nach rechts**

***Vorgriff*: In der Lehrveranstaltung zu Algorithmen und Datenstrukturen im zweiten Fachsemester werden Sie unter anderem die Datenstruktur kennen lernen, die die Grundlage wahrscheinlich jedes Datenbanksystems der Welt ist, eben weil sie die Anzahl Zugriffe auf den Hintergrundspeicher minimiert, die notwendig sind, um auf ein einzelnes Element zuzugreifen.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:



Implementationsinvariante für B-Baum der Ordnung *M*:

- **In jedem Knoten Platz für 2*M*-1 Keys und 2M Nachfolger**
- **In jedem Knoten außer der Wurzel mindestens M-1 Keys**
- **Alle Blätter auf derselben Höhe**
- **Sortierung der Keys von links nach rechts**

Diese Datenstruktur ist unter dem Namen B-Baum bekannt. Jeder B-Baum hat eine Ordnung, das ist eine positive ganze Zahl. Der B-Baum auf der Folie oben hat Ordnung 2.

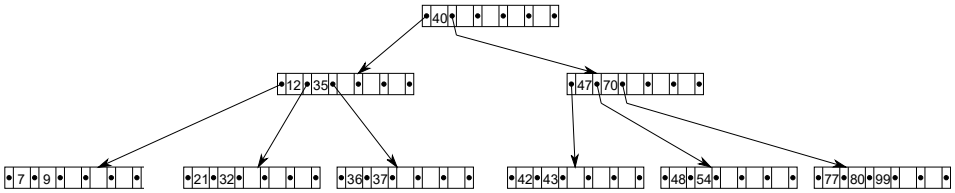
Die vier Punkte bilden die Implementationsinvariante eines B-Baums.

Die Ordnung M ist so zu wählen, dass ein Baumknoten gerade so eben noch vollständig auf eine Speicherseite passt. Datenbanksysteme organisieren die Kommunikation mit dem Hintergrundspeicher am Betriebssystem vorbei selbst und können daher gewährleisten, dass jeder Baumknoten vollständig auf einer eigenen Seite abgelegt ist. Das heißt, greift man auf eine einzelne Information in einem Baumknoten zu, dann wird der gesamte Baumknoten aus dem Hintergrundspeicher in den Hauptspeicher kopiert, und beim Zurückkopieren in den Hintergrundspeicher ebenso. Heutzutage realistisch ist eine Ordnung M im mindestens dreistelligen Bereich.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:



Implementationsinvariante für B-Baum der Ordnung *M*:

- **In jedem Knoten Platz für 2*M*-1 Keys und 2M Nachfolger**
- **In jedem Knoten außer der Wurzel mindestens M-1 Keys**
- **Alle Blätter auf derselben Höhe**
- **Sortierung der Keys von links nach rechts**

Es hat sich eingebürgert, dass die Ordnung in einem etwas unintuitiven Verhältnis zur Größe eines Baumknotens steht. Jeder Baumknoten enthält eine ungerade Anzahl von Keys und eine um eins größere Anzahl von Verweisen auf andere Baumknoten. Die Ordnung ist dann die Hälfte dieser Anzahl.

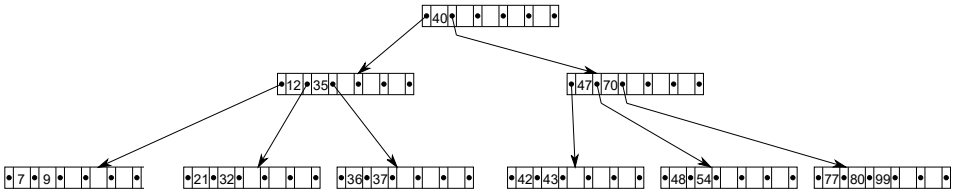
Sie können sich das so vorstellen, dass die Keys und die Verweise jeweils ein eigenes Array der Länge 2M-1 beziehungsweise 2M bilden. Das sind zwei Attribute der Klasse für Baumknoten. Ein drittes Argument ist eine positive ganze Zahl, in dem für den Baumknoten gespeichert wird, wie viele Plätze für Keys tatsächlich verwendet werden.

***Erinnerung*: In Kapitel 07 hatten wir einen Abschnitt zu einer alternativen Implementation von LinkedList, in der analog – mit einem zusätzlichen ganzzahligen Attribut in der Klasse für Listenknoten – gespeichert wurde, wie viele Komponenten im Array des Listenknotens durch reale Schlüsselwerte in der repräsentierten Sequenz belegt sind. Dieses Grundprinzip taucht hier wieder auf.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:



Implementationsinvariante für B-Baum der Ordnung *M*:

- **In jedem Knoten Platz für 2*M*-1 Keys und 2M Nachfolger**
- **In jedem Knoten außer der Wurzel mindestens M-1 Keys**
- **Alle Blätter auf derselben Höhe**
- **Sortierung der Keys von links nach rechts**

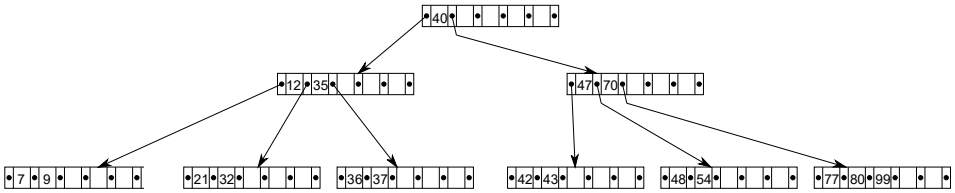
Dies ist eine Vorgabe: Wenn die nicht eingehalten wird, dann nennt man den Baum nicht B-Baum. Gleich auf der nächsten Folie werden wir sehen, was für dramatische positive Konsequenzen diese Vorgabe hat.

***Vorgriff*: In der AuD werden Sie sehen, dass das genau die richtige Mindestzahl ist, so dass das Einfügen und Entfernen von Schlüsselwerten recht schnell vonstatten geht und die Implementationsinvariante für B-Bäume dabei eingehalten wird.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:



Implementationsinvariante für B-Baum der Ordnung *M*:

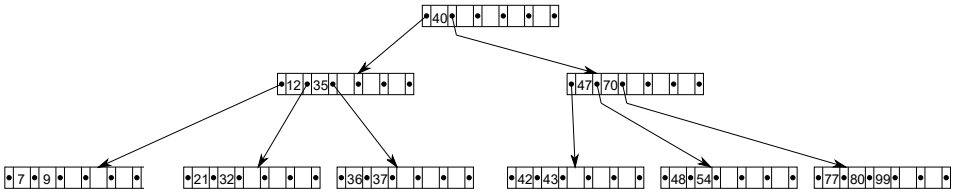
- **In jedem Knoten Platz für 2*M*-1 Keys und 2M Nachfolger**
- **In jedem Knoten außer der Wurzel mindestens M-1 Keys**
- **Alle Blätter auf derselben Höhe**
- **Sortierung der Keys von links nach rechts**

Auch diese Vorgabe ist an den dramatisch positiven Konsequenzen beteiligt, die wir uns gleich auf der nächsten Folie ansehen werden.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:



Implementationsinvariante für B-Baum der Ordnung *M*:

- **In jedem Knoten Platz für 2*M*-1 Keys und 2M Nachfolger**
- **In jedem Knoten außer der Wurzel mindestens M-1 Keys**
- **Alle Blätter auf derselben Höhe**
- **Sortierung der Keys von links nach rechts**

Die Sequenz von Schlüsselwerten ist so auf die Baumknoten verteilt, dass die *Suchbaumeigenschaft* erfüllt ist: Stellen Sie sich vor, Sie würden schrittweise von unten nach oben jeden Baumknoten außer der Wurzel in seinem unmittelbaren Vorgängerknoten einfügen, und zwar genau an der Stelle, an der im Vorgängerknoten der Verweis auf diesen Knoten dargestellt ist. Dann würde am Ende die Wurzel alle Schlüsselwerte enthalten, und zwar in aufsteigender Reihenfolge.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:

Stufe	Mindestzahl Knoten	Mindestzahl Keys
0	1	1
1	2	$2 \cdot (M-1)$
2	$2M$	$2M \cdot (M-1)$
3	$2M^2$	$2M^2 \cdot (M-1)$
4	$2M^3$	$2M^3 \cdot (M-1)$
S	$2M^{S-1}$	$2M^{S-1} \cdot (M-1)$

Auf dieser Folie jetzt die angekündigten dramatisch positiven Konsequenzen.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:

Stufe	Mindestzahl Knoten	Mindestzahl Keys
0	1	1
1	2	$2 \cdot (M-1)$
2	$2M$	$2M \cdot (M-1)$
3	$2M^2$	$2M^2 \cdot (M-1)$
4	$2M^3$	$2M^3 \cdot (M-1)$
S	$2M^{S-1}$	$2M^{S-1} \cdot (M-1)$

Auf der obersten Stufe, bei der Wurzel, ist noch nichts Bemerkenswertes zu sehen. In der Wurzel muss nach Vorgabe ja auch nur ein einziger Schlüsselwert enthalten sein.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:

Stufe	Mindestzahl Knoten	Mindestzahl Keys
0	1	1
1	2	$2 \cdot (M-1)$
2	$2M$	$2M \cdot (M-1)$
3	$2M^2$	$2M^2 \cdot (M-1)$
4	$2M^3$	$2M^3 \cdot (M-1)$
S	$2M^{S-1}$	$2M^{S-1} \cdot (M-1)$

Da die Wurzel mindestens einen Schlüsselwert enthält, hat die Wurzel mindestens zwei direkte Nachfolger. Und jeder dieser Baumknoten hat gemäß Implementationsinvariante mindestens M minus 1 Schlüsselwerte.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:

Stufe	Mindestzahl Knoten	Mindestzahl Keys
0	1	1
1	2	$2 \cdot (M-1)$
2	$2M$	$2M \cdot (M-1)$
3	$2M^2$	$2M^2 \cdot (M-1)$
4	$2M^3$	$2M^3 \cdot (M-1)$
S	$2M^{S-1}$	$2M^{S-1} \cdot (M-1)$

Da jeder der beiden Baumknoten mindestens M Nachfolger hat, gibt es auf der zweiten Stufe mindestens zweimal M Baumknoten. Und die Vorgabe, dass jeder Baumknoten außer der Wurzel mindestens M minus 1 Schlüsselwerte enthält, ergibt beinahe zweimal das Quadrat von M als Anzahl Keys auf der zweiten Stufe.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:

Stufe	Mindestzahl Knoten	Mindestzahl Keys
0	1	1
1	2	$2 \cdot (M-1)$
2	2M	$2M \cdot (M-1)$
3	$2M^2$	$2M^2 \cdot (M-1)$
4	$2M^3$	$2M^3 \cdot (M-1)$
S	$2M^{S-1}$	$2M^{S-1} \cdot (M-1)$

Diese Argumentation setzt sich dann auf den nächsten Stufen fort, ...

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:

Stufe	Mindestzahl Knoten	Mindestzahl Keys
0	1	1
1	2	$2 \cdot (M-1)$
2	$2M$	$2M \cdot (M-1)$
3	$2M^2$	$2M^2 \cdot (M-1)$
4	$2M^3$	$2M^3 \cdot (M-1)$
S	$2M^{S-1}$	$2M^{S-1} \cdot (M-1)$

Und so sieht daher die allgemeine Formel für die Mindestzahl Baumknoten und Schlüsselwerte auf einer beliebigen Stufe S aus.

Wie gesagt, wir können davon ausgehen, dass M heutzutage mindestens dreistellig ist. Wenn wir einfach nur mit 100 rechnen, dann reicht schon Stufe 2 für ungefähr zwanzigtausend Schlüsselwerte aus, Stufe 3 für zwei Millionen, Stufe 4 für zweihundert Millionen und Stufe 5 für zwanzig Milliarden, und so weiter.

Und da alle Blätter gemäß Implementationsinvariante auf derselben Stufe sind, hat ein B-Baum der Ordnung 100 selbst bei zwanzig Milliarden Schlüsselwerten nur Baumknoten auf den Stufen 0 bis 5. Mit anderen Worten: Selbst bei zwanzig Milliarden Schlüsselwerten sind für den Zugriff auf jeden beliebigen Schlüsselwert maximal sechs Zugriffe auf den Hintergrundspeicher notwendig.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Minimierung Anzahl Zugriffe auf Hintergrundspeicher:

Stufe	Mindestzahl Knoten	Mindestzahl Keys
0	1	1
1	2	$2 \cdot (M-1)$
2	$2M$	$2M \cdot (M-1)$
3	$2M^2$	$2M^2 \cdot (M-1)$
4	$2M^3$	$2M^3 \cdot (M-1)$
S	$2M^{S-1}$	$2M^{S-1} \cdot (M-1)$

***Nebenbemerkung*: An einem Schlüsselwert würde in der Praxis ein Verweis auf die zugehörige Information stehen, zum Beispiel die Kontodaten zu einer Kontonummer oder die Studierendenstammdaten zu einer Matrikelnummer. Das erfordert dann noch einen weiteren, letzten Zugriff auf den Hintergrundspeicher.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Threads vermeiden

- **Falls Threads zur Effizienzsteigerung verwendet?**
- **Falls Threads zum besseren Design verwendet!**

Bei Ihren Laufzeitmessungen kann es sich herausstellen, dass ein großer Teil der Laufzeit dadurch verursacht wird, dass Sie Ihr Programm in mehrere Threads aufgeteilt haben.

***Erinnerung*: Im Abschnitt zu Threads in Kapitel 09 hatten wir zwei prinzipiell verschiedene Motivationen für Threads gesehen.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Threads vermeiden

- **Falls Threads zur Effizienzsteigerung verwendet?**
- **Falls Threads zum besseren Design verwendet!**

Einerseits können Threads verwendet werden, um die Laufzeit zu reduzieren, indem nämlich die Arbeitslast auf mehrere Threads verteilt wird, die auf unterschiedlichen Prozessoren abgearbeitet werden.

Allerdings hatten wir auch festgestellt, dass dieses Ziel nicht zwangsläufig erreicht wird. Das Laufzeitverhalten von Threads ist nur bedingt vorhersehbar. Daher kann es schlimmstenfalls durchaus sein, dass der Versuch, die Arbeitslast mittels Threads auf mehrere Prozessoren zu verteilen, zu einer *Erhöhung* statt zu einer *Verminderung* der Laufzeit führt.

Also sollte man sogar in dem Fall, dass Threads zur Effizienzsteigerung eingesetzt werden, prüfen, ob sich die Laufzeit durch Rückführung der gesamten Arbeitslast in einen einzigen Thread nicht doch eher *vermindert*.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Threads vermeiden

- Falls Threads zur Effizienzsteigerung verwendet?
- Falls Threads zum besseren Design verwendet!

In dem Fall, dass Threads nicht zur Effizienzsteigerung, sondern zur Verbesserung des Programmdesigns eingeführt wurden, sieht die Sachlage natürlich anders aus. Oft genug ist die Eliminierung von Threads kaum oder gar nicht möglich, beispielsweise wenn Sie Funktionalität aus der Standardbibliothek benutzen, in der Threads verwendet werden.

Aber auch in dem Fall, in dem Sie die Threads selbst verwalten, sollten Sie genau überlegen, ob es das wert ist. Wenn Sie Threads zum besseren Programmdesign eingeführt haben, dann resultiert die Eliminierung der Threads naturgemäß in einem deutlich schlechteren Programmdesign. So etwas sollten Sie in der Regel nur tun, wenn der Effizienzverlust durch die Threads wirklich unerträglich hoch ist.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Tools suchen und verwenden

- **Aggressive Optimierung**
  - **Virtual Dispatch u.a. an möglichst vielen Stellen wegoptimieren**
- **Native Code Compilation**

→ Überwiegend kommerziell

Bis jetzt hatten wir nur Möglichkeiten zur Laufzeitverbesserungen innerhalb der Programmiersprache betrachtet. Es gibt aber auch zusätzliche Tools, die man nutzen kann, um die Laufzeit zu verbessern.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Tools suchen und verwenden

- Aggressive Optimierung
  - ➢ Virtual Dispatch u.a. an möglichst vielen Stellen wegoptimieren
- Native Code Compilation

→ Überwiegend kommerziell

Der Java Byte Code, in den Ihr Quelltext übersetzt wird, ist nicht besonders gut optimiert, da ist noch viel Luft nach oben.

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Tools suchen und verwenden

- **Aggressive Optimierung**
  - ➢ **Virtual Dispatch u.a. an möglichst vielen Stellen wegoptimieren**
- **Native Code Compilation**

→ **Überwiegend kommerziell**

***Erinnerung*: Wir hatten schon des Öfteren gesehen, wie viel Aufwand ein einziger Methodenaufruf kostet. Der Grund ist, dass der dynamische Typ offengehalten werden soll und der dynamische Typ über die angesteuerte Implementation entscheidet. Der dynamische Typ kann aber in vielen Situationen schon aus dem Quelltext erschlossen werden. In diesem Fall kann man Virtual Dispatch aus der Übersetzung des Quelltextes eliminieren und den Aufruf der anzusteuernden Implementation direkt einsetzen.**

Laufzeitverbesserungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Tools suchen und verwenden

- Aggressive Optimierung
  - ➢ Virtual Dispatch u.a. an möglichst vielen Stellen wegoptimieren
- Native Code Compilation

→ Überwiegend kommerziell

Ein anderer Weg ist die Übersetzung in den Maschinencode der Plattform statt in Java Byte Code. Die Ausführung von Maschinencode ist um Größenordnungen schneller als die von Java Byte Code, insbesondere wenn der übersetzte Code auf die Plattform hin optimiert wird. Nachteil ist, dass der Maschinencode nicht portabel ist.

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Komplexität

Aus Sicht der wissenschaftlichen Informatik ist die Asymptotische Komplexität der Kernpunkt des gesamten Themas Laufzeit.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zur Vereinfachung: erst einmal nur User Time
- Schätzung der Laufzeit durch eine mathematische Funktion
  - In Kennzahlen, die die Problemgröße beschreiben
  - Durch mathematische Überlegungen und/oder empirische Laufzeitstudien
- Representative Operation Counts
  - Ausführungen zählen für mathematische Überlegungen
  - Zeiten messen und akkumulieren bei Laufzeitstudien

Auf dieser Folie schauen wir uns das Thema erst einmal grob und abstrakt an und gehen danach erst ins Detail.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zur Vereinfachung: erst einmal nur User Time
- Schätzung der Laufzeit durch eine mathematische Funktion
  - In Kennzahlen, die die Problemgröße beschreiben
  - Durch mathematische Überlegungen und/oder empirische Laufzeitstudien
- Representative Operation Counts
  - Ausführungen zählen für mathematische Überlegungen
  - Zeiten messen und akkumulieren bei Laufzeitstudien

Die Idee ist, dass man versucht, die Laufzeit durch eine möglichst einfache mathematische Funktion zu beschreiben.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zur Vereinfachung: erst einmal nur User Time
- Schätzung der Laufzeit durch eine mathematische Funktion
  - In Kennzahlen, die die Problemgröße beschreiben
  - Durch mathematische Überlegungen und/oder empirische Laufzeitstudien
- Representative Operation Counts
  - Ausführungen zählen für mathematische Überlegungen
  - Zeiten messen und akkumulieren bei Laufzeitstudien

Dreh- und Angelpunkt ist die Problemgröße. Natürlich hängt die Laufzeit insbesondere von der Menge der verarbeiteten Daten ab. Diese Menge muss geeignet in Kennzahlen ausgedrückt werden.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zur Vereinfachung: erst einmal nur User Time
- Schätzung der Laufzeit durch eine mathematische Funktion
  - In Kennzahlen, die die Problemgröße beschreiben
  - Durch mathematische Überlegungen und/oder empirische Laufzeitstudien
- Representative Operation Counts
  - Ausführungen zählen für mathematische Überlegungen
  - Zeiten messen und akkumulieren bei Laufzeitstudien

Grundsätzlich sind zwei Wege möglich: Entweder man schaut sich den Quelltext genau an und überlegt sich, wie die mathematische Funktion aussehen könnte; oder man lässt das Programm mit verschiedenen Problemgrößen laufen und versucht, daraus die Funktion zu schätzen. Beziehungsweise man kann sich auch erst die Funktion theoretisch überlegen und das Ergebnis dieser Überlegungen dann durch solche Laufzeitstudien überprüfen.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zur Vereinfachung: erst einmal nur User Time
- Schätzung der Laufzeit durch eine mathematische Funktion
  - In Kennzahlen, die die Problemgröße beschreiben
  - Durch mathematische Überlegungen und/oder empirische Laufzeitstudien
- Representative Operation Counts
  - Ausführungen zählen für mathematische Überlegungen
  - Zeiten messen und akkumulieren bei Laufzeitstudien

Um nicht die Laufzeit selbst, sondern eine beschreibende mathematische Funktion zu finden, müssen wir die Anweisungen im Programm identifizieren, die potentiell dafür infrage kommen, dass sie die Laufzeit bei großen Problemgrößen vielleicht dominieren könnten. In der Literatur finden Sie dazu auch diesen englischen Fachbegriff.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zur Vereinfachung: erst einmal nur User Time
- Schätzung der Laufzeit durch eine mathematische Funktion
  - In Kennzahlen, die die Problemgröße beschreiben
  - Durch mathematische Überlegungen und/oder empirische Laufzeitstudien
- Representative Operation Counts
  - Ausführungen zählen für mathematische Überlegungen
  - Zeiten messen und akkumulieren bei Laufzeitstudien

Wenn wir uns darauf beschränken, uns rein theoretisch Gedanken über eine die Laufzeit beschreibende Funktion zu machen, dann haben wir es ja nicht mit echten Laufzeiten zu tun. Alles, was wir tun können, ist zu bestimmen, wie häufig eine Anweisung durchlaufen wird bei bestimmten Werten der Kennzahlen.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zur Vereinfachung: erst einmal nur User Time
- Schätzung der Laufzeit durch eine mathematische Funktion
  - In Kennzahlen, die die Problemgröße beschreiben
  - Durch mathematische Überlegungen und/oder empirische Laufzeitstudien
- Representative Operation Counts
  - Ausführungen zählen für mathematische Überlegungen
  - Zeiten messen und akkumulieren bei Laufzeitstudien

Wenn wir hingegen empirisch vorgehen, dann messen wir ja echte Laufzeiten für die verschiedenen Passagen im Quelltext, für die wir eben die Laufzeiten messen wollen. Die Laufzeiten für eine Passage kann man dann aufsummieren, und die Summe ist die Gesamtlaufzeit, die das Programm in dieser Passage verbracht hat.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
UserTimeAccumulator accumulator = new UserTimeAccumulator();
while ( ..... ) {
   .....
   accumulator.startUserTimeRecording();
   .....
   accumulator.finishUserTimeRecording();
   .....
}
```

Laufzeiten verschiedener Passagen im Quelltext kann man selbst akkumulieren, zum Beispiel die User Time mit einer Klasse wie dieser, die wir auf dieser Folie im Einsatz sehen und auf der nächsten Folie implementieren werden.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
UserTimeAccumulator accumulator = new UserTimeAccumulator();
while ( ..... ) {
   .....
   accumulator.startUserTimeRecording();
   .....
   accumulator.finishUserTimeRecording();
   .....
}
```

Das ist die Passage im Quelltext, deren User Time gemessen werden soll. Da diese Passage Teil einer Schleife ist, wird sie in der Regel mehrfach durchlaufen, in der Praxis vielleicht millionenfach. Relevant ist also die *akkumulierte* Laufzeit.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
UserTimeAccumulator accumulator = new UserTimeAccumulator();
while ( ..... ) {
   .....
   accumulator.startUserTimeRecording();
   .....
   accumulator.finishUserTimeRecording();
   .....
}
```

Mit diesen beiden Methoden soll die Laufzeit akkumuliert werden, wie bei einer Stoppuhr. Das heißt, durch Aufruf der Methode startUserTimeRecording wird dafür gesorgt, dass die Zeitmessung weiterläuft, und mit Methode finishUserTime Recording, dass die Zeitmessung bis zum nächsten Aufruf von startUserTimeRecording angehalten wird.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class UserTimeAccumulator {
   private long accumulatedUserTimeSoFar;
   private long startTimeOfRecording;
   private ThreadMXBean bean;

   public UserTimeAccumulator () {
       accumulatedUserTimeSoFar = 0;
       bean = ManagementFactory.getThreadMXBean();
   }

   public long getAccumulatedUserTimeSoFar {
       return accumulatedUserTimeSoFar;
   }

   .....
}
```

Nun kommen wir zur angekündigten Implementation dieser Klasse-

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class UserTimeAccumulator {
  private long accumulatedUserTimeSoFar;
  private long startTimeOfRecording;
  private ThreadMXBean bean;

  public UserTimeAccumulator () {
     accumulatedUserTimeSoFar = 0;
     bean = ManagementFactory.getThreadMXBean();
  }

  public long getAccumulatedUserTimeSoFar {
     return accumulatedUserTimeSoFar;
  }

  .....
}
```

In dieser Objektvariablen werden die Laufzeiten akkumuliert. Logischerweise wird sie im Konstruktor auf 0 gesetzt, da noch keine Zeiten gemessen worden sind. Da diese Objektvariable private ist, wird eine get-Methode für den Zugriff benötigt. Eine set-Methode wäre offensichtlich kontraproduktiv, höchstens vielleicht eine *re*set-Methode, die den akkumulierten Wert wieder auf 0 setzt.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class UserTimeAccumulator {
   private long accumulatedUserTimeSoFar;
   private long startTimeOfRecording;
   private ThreadMXBean bean;

   public UserTimeAccumulator () {
       accumulatedUserTimeSoFar = 0;
       bean = ManagementFactory.getThreadMXBean();
   }

   public long getAccumulatedUserTimeSoFar {
       return accumulatedUserTimeSoFar;
   }

   .....
}
```

Diese zweite Objektvariable dient nur internen Hilfszwecken, wie wir gleich sehen werden, also keine get- oder set-Methode.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class UserTimeAccumulator {
   private long accumulatedUserTimeSoFar;
   private long startTimeOfRecording;
   private ThreadMXBean bean;

   public UserTimeAccumulator () {
      accumulatedUserTimeSoFar = 0;
      bean = ManagementFactory.getThreadMXBean();
   }

   public long getAccumulatedUserTimeSoFar {
      return accumulatedUserTimeSoFar;
   }

   .....
}
```

***Erinnerung*: Das ist eins-zu-eins wie im Abschnitt zur Laufzeitmessung weiter vorne in diesem Kapitel, braucht also hier nicht noch einmal erläutert zu werden.**

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class UserTimeAccumulator {
   private long accumulatedUserTimeSoFar;
   private long startTimeOfRecording;
   private ThreadMXBean bean;
   .....
   public void startUserTimeRecording () {
      startTimeOfRecording = bean.getCurrentThreadUserTime();
   }
   public void finishUserTimeRecording () {
      accumulatedUserTimeSoFar += bean.getCurrentThreadUserTime()
                                  – startTimeOfRecording;
   }
}
```

Hier sehen Sie nun die beiden Methoden, die wir schon eingangs im Einsatz gesehen hatten.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class UserTimeAccumulator {
   private long accumulatedUserTimeSoFar;
   private long startTimeOfRecording;
   private ThreadMXBean bean;

   .........

   public void startUserTimeRecording () {
      startTimeOfRecording = bean.getCurrentThreadUserTime();
   }
   public void finishUserTimeRecording () {
      accumulatedUserTimeSoFar += bean.getCurrentThreadUserTime()
                                  – startTimeOfRecording;
   }
}
```

Hier sehen Sie die Funktion der Hilfsvariable: Zu Beginn der Zeitmessung wird hierin der momentane Zeitpunkt festgehalten.

Asymptotische Komplexität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class UserTimeAccumulator {
   private long accumulatedUserTimeSoFar;
   private long startTimeOfRecording;
   private ThreadMXBean bean;

   .........

   public void startUserTimeRecording () {
      startTimeOfRecording = bean.getCurrentThreadUserTime();
   }
   public void finishUserTimeRecording () {
      accumulatedUserTimeSoFar += bean.getCurrentThreadUserTime()
                                  – startTimeOfRecording;
   }
}
```

Und der Zeitpunkt am Ende der Zeitmessung wird mit dieser Startzeit abgeglichen, das Ergebnis wird zur akkumulierten Zeit hinzuaddiert.

Suche im sortierten Array

TECHNISCHE UNIVERSITÄT DARMSTADT

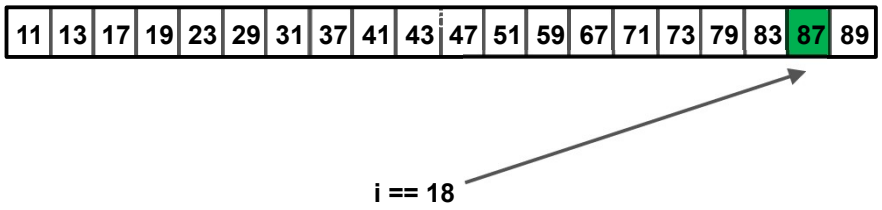
Gesucht: 87

Als konkretes Beispiel nehmen wir uns lineare und binäre Suche vor, zuerst lineare Suche.

***Erinnerung*: Beide Algorithmen hatten wir im Kapitel zu korrekter Software gesehen, Abschnitt zur Korrektheit von Schleifen. Zurückgeliefert werden soll die Information, ob ein gegebener Wert im Array ist oder nicht. Das Array muss aufsteigend sortiert sein.**

Lineare Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

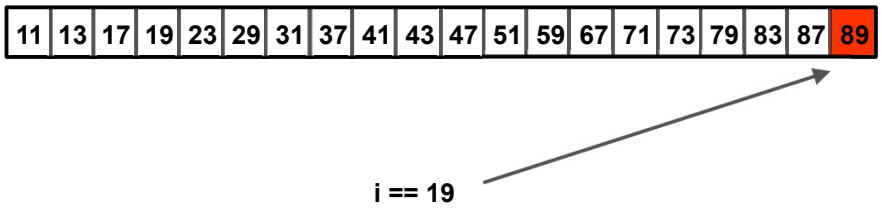


Gesucht: 87

Die Suche geht alle Komponenten des Arrays der Reihe nach durch, bis der gesuchte Wert gefunden ist, ...

Lineare Suche

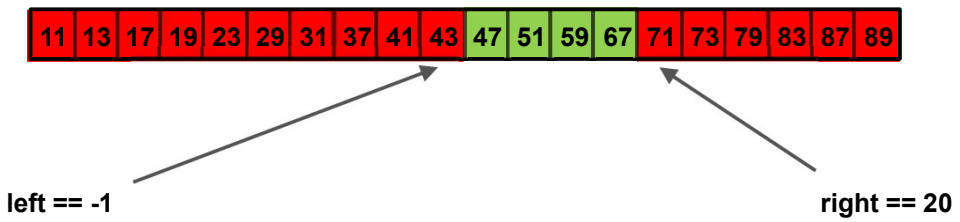
TECHNISCHE UNIVERSITÄT DARMSTADT



... oder bis entweder das Ende des Arrays erreicht oder ein größerer als der gesuchte Wert gefunden wird. In beiden Fällen kann man sicher sein, dass der gesuchte Wert *nicht* im Array ist.

Lineare Suche

TECHNISCHE UNIVERSITÄT DARMSTADT



Gesucht: 67

Im Gegensatz dazu grenzt die binäre Suche die Stelle, an der der gesuchte Wert sein müsste, immer weiter ein, wobei die Größe des verbliebenen Intervalls in jedem Schritt mindestens halbiert wird. Daher lässt sich die Anzahl der Durchläufe durch die Schleife nach oben durch den auf die nächste ganze Zahl aufgerundeten Zweierlogarithmus der Arraylänge beschränken.

Vergleich linear / binär

TECHNISCHE UNIVERSITÄT DARMSTADT

N	$\log_2 N$	N	$\log_2 N$
2	1	1.024	10
4	2	1.024^2	20
8	3	1.024^3	30
16	4	1.024^4	40
32	5	1.024^5	50
64	6	1.024^6	60

Hier sehen Sie den dramatischen Unterschied in der Anzahl Durchläufe durch die Schleife bei linearer und binärer Suche. Höchstwahrscheinlich wird ein einzelner Durchlauf durch die Schleife bei binärer Suche mehr Laufzeit kosten als bei linearer Suche. Aber diese beiden Laufzeiten werden sich sicherlich nur um einen Faktor unterscheiden, der so klein ist, dass er schon bei moderaten Arraylängen nicht mehr ins Gewicht fällt gegenüber dem Auseinanderklaffen von logarithmischem und linearem Wachstum.

Vergleich linear / binär

TECHNISCHE UNIVERSITÄT DARMSTADT

N	$\log_2 N$	N	$\log_2 N$
2	1	1.024	10
4	2	1.024^2	20
8	3	1.024^3	30
16	4	1.024^4	40
32	5	1.024^5	50
64	6	1.024^6	60

Und genau das ist mit asymptotischer Komplexität gemeint: Die asymptotische Komplexität eines Algorithmus gibt an, in welcher Größenordnung die Laufzeit des Algorithmus mit der Problemgröße wächst. Die asymptotische Komplexität von linearer Suche ist also linear, die von binärer Suche ist logarithmisch. Unterscheiden sich zwei Algorithmen so stark in ihrer asymptotischen Komplexität wie diese beiden, dann weiß man, welchen der beiden Algorithmen man für richtig große Problemgrößen wählen sollte, egal was die genauen Laufzeiten pro Schleifendurchlauf sind.

Insbesondere ist die asymptotische Komplexität eines Algorithmus unabhängig von Hardware, Programmiersprache, Geschicktheit der Implementation, und so weiter.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch Beispiel lineare vs. binäre Suche im sortierten Array.
- Die Problemgröße ist hier ausgedrückt durch die Zahl *N* der zu durchsuchenden Werte.
  - Also Länge des Arrays.
- Für jede Problemgröße gibt es einen *Worst Case* und einen *Best Case*.
- Best Case bei beiden Suchstrategien: das erste angeschaute Element ist größer / gleich dem gesuchten.
  - Laufzeit hängt *hier* im Best Case nicht von der Problemgröße ab.
- Worst Case: Man muss die Schleife ganz durchlaufen.
  - Lineare Suche: *N* Durchläufe
  - Binäre Suche: ca. $\log_2 N$ Durchläufe

Wir bleiben noch kurz bei diesen beiden Algorithmen und nutzen sie zur Einführung zweier wichtiger allgemeiner Begriffe: Worst Case und Best Case.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch Beispiel lineare vs. binäre Suche im sortierten Array.
- Die Problemgröße ist hier ausgedrückt durch die Zahl *N* der zu durchsuchenden Werte.
  - Also Länge des Arrays.
- Für jede Problemgröße gibt es einen *Worst Case* und einen *Best Case*.
- Best Case bei beiden Suchstrategien: das erste angeschaute Element ist größer / gleich dem gesuchten.
  - Laufzeit hängt *hier* im Best Case nicht von der Problemgröße ab.
- Worst Case: Man muss die Schleife ganz durchlaufen.
  - Lineare Suche: *N* Durchläufe
  - Binäre Suche: ca. $\log_2 N$ Durchläufe

Die Problemgröße ist bei der algorithmischen Problemstellung, ein Element in einem sortierten Array zu suchen, recht einfach: nur die Arraylänge.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch Beispiel lineare vs. binäre Suche im sortierten Array.
- Die Problemgröße ist hier ausgedrückt durch die Zahl *N* der zu durchsuchenden Werte.
  - Also Länge des Arrays.
- Für jede Problemgröße gibt es einen *Worst Case* und einen *Best Case*.
- Best Case bei beiden Suchstrategien: das erste angeschaute Element ist größer / gleich dem gesuchten.
  - Laufzeit hängt *hier* im Best Case nicht von der Problemgröße ab.
- Worst Case: Man muss die Schleife ganz durchlaufen.
  - Lineare Suche: *N* Durchläufe
  - Binäre Suche: ca. $\log_2 N$ Durchläufe

Der Worst Case und der Best Case sind zwei mathematische Funktionen in der Problemgröße, bei unserem Beispiel also zwei mathematische Funktionen in der Arraylänge.

***Nebenbemerkung*: Diese beiden Begriffe werden in der Regel auch in der deutschsprachigen Literatur nicht aus dem Englischen übersetzt.**

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch Beispiel lineare vs. binäre Suche im sortierten Array.
- Die Problemgröße ist hier ausgedrückt durch die Zahl *N* der zu durchsuchenden Werte.
  - Also Länge des Arrays.
- Für jede Problemgröße gibt es einen *Worst Case* und einen *Best Case*.
- Best Case bei beiden Suchstrategien: das erste angeschaute Element ist größer / gleich dem gesuchten.
  - Laufzeit hängt *hier* im Best Case nicht von der Problemgröße ab.
- Worst Case: Man muss die Schleife ganz durchlaufen.
  - Lineare Suche: *N* Durchläufe
  - Binäre Suche: ca. $\log_2 N$ Durchläufe

Der Name „Best Case“ sagt ja schon, worum es geht, nämlich der Fall, der die geringste Laufzeit produziert. Das ist natürlich genau *der* Fall, dass schon bei der allerersten Arraykomponente aufgehört werden kann mit der Suche.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch Beispiel lineare vs. binäre Suche im sortierten Array.
- Die Problemgröße ist hier ausgedrückt durch die Zahl *N* der zu durchsuchenden Werte.
  - Also Länge des Arrays.
- Für jede Problemgröße gibt es einen *Worst Case* und einen *Best Case*.
- Best Case bei beiden Suchstrategien: das erste angeschaute Element ist größer / gleich dem gesuchten.
  - Laufzeit hängt *hier* im Best Case nicht von der Problemgröße ab.
- Worst Case: Man muss die Schleife ganz durchlaufen.
  - Lineare Suche: *N* Durchläufe
  - Binäre Suche: ca. $\log_2 N$ Durchläufe

Und für die Laufzeit in diesem bestmöglichen Fall ist die Arraylänge völlig egal.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch Beispiel lineare vs. binäre Suche im sortierten Array.
- Die Problemgröße ist hier ausgedrückt durch die Zahl *N* der zu durchsuchenden Werte.
  - Also Länge des Arrays.
- Für jede Problemgröße gibt es einen *Worst Case* und einen *Best Case*.
- Best Case bei beiden Suchstrategien: das erste angeschaute Element ist größer / gleich dem gesuchten.
  - Laufzeit hängt *hier* im Best Case nicht von der Problemgröße ab.
- Worst Case: Man muss die Schleife ganz durchlaufen.
  - Lineare Suche: *N* Durchläufe
  - Binäre Suche: ca. $\log_2 N$ Durchläufe

Betonung liegt auf diesem Wort, denn das ist nicht bei allen Algorithmen so. Zum Beispiel wird jeder Algorithmus zum Sortieren eines Arrays sich jeden zu sortierenden Wert wenigstens einmal anschauen müssen. Daher ist beim Sortieren der Best Case bei jedem Algorithmus mindestens linear. Das werden wir später in diesem Abschnitt bei Selection Sort sehen.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch Beispiel lineare vs. binäre Suche im sortierten Array.
- Die Problemgröße ist hier ausgedrückt durch die Zahl *N* der zu durchsuchenden Werte.
  - Also Länge des Arrays.
- Für jede Problemgröße gibt es einen *Worst Case* und einen *Best Case*.
- Best Case bei beiden Suchstrategien: das erste angeschaute Element ist größer / gleich dem gesuchten.
  - Laufzeit hängt *hier* im Best Case nicht von der Problemgröße ab.
- Worst Case: Man muss die Schleife ganz durchlaufen.
  - Lineare Suche: *N* Durchläufe
  - Binäre Suche: ca. $\log_2 N$ Durchläufe

Analog ist der Worst Case der Fall mit der höchsten Laufzeit. Bei beiden Algorithmen ist das genau *der* Fall, dass die Schleife nicht vorzeitig abgebrochen werden kann.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch Beispiel lineare vs. binäre Suche im sortierten Array.
- Die Problemgröße ist hier ausgedrückt durch die Zahl *N* der zu durchsuchenden Werte.
  - Also Länge des Arrays.
- Für jede Problemgröße gibt es einen *Worst Case* und einen *Best Case*.
- Best Case bei beiden Suchstrategien: das erste angeschaute Element ist größer / gleich dem gesuchten.
  - Laufzeit hängt *hier* im Best Case nicht von der Problemgröße ab.
- Worst Case: Man muss die Schleife ganz durchlaufen.
  - Lineare Suche: *N* Durchläufe
  - Binäre Suche: ca. $\log_2 N$ Durchläufe

Konkret bei linearer Suche ist der Worst Case demnach der Fall, dass das Array ganz durchlaufen wurde.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch Beispiel lineare vs. binäre Suche im sortierten Array.
- Die Problemgröße ist hier ausgedrückt durch die Zahl *N* der zu durchsuchenden Werte.
  - Also Länge des Arrays.
- Für jede Problemgröße gibt es einen *Worst Case* und einen *Best Case*.
- Best Case bei beiden Suchstrategien: das erste angeschaute Element ist größer / gleich dem gesuchten.
  - Laufzeit hängt *hier* im Best Case nicht von der Problemgröße ab.
- Worst Case: Man muss die Schleife ganz durchlaufen.
  - Lineare Suche: *N* Durchläufe
  - Binäre Suche: ca. $\log_2 N$ Durchläufe

Und konkret bei binärer Suche ist der Worst Case der Fall, dass die schrittweise Eingrenzung so lange weitergeht, bis das Intervall leer ist oder nur noch aus dem gesuchten Wert besteht.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei großen Werten von *N* sind alle Operationen neben der Schleife für die Laufzeit unerheblich.
  - Je größer *N*, um so weniger ins Gewicht fallend.
- Die Laufzeit pro Schleifendurchlauf variiert nur in engen Grenzen.
- Konsequenz: Die Laufzeit bei linearer / binärer Suche landet mit wachsendem N schnell in einem Korridor
  - [$c_1 \cdot N$ ... $c_2 \cdot N$] bei linearer Suche im Worst Case,
  - [$c_3 \cdot \log_2 N$ ... $c_4 \cdot \log_2 N$] bei binärer Suche im Worst Case.
- Im Best Case bei beiden: [c_5 ... c_6] unabhängig von *N*.
- Dabei sind $c_1 ... c_6$ unbekannte, aber feste Konstanten:
  - unabhängig von *N*,
  - abhängig von Plattform, Programmiersprache, Compiler usw.

Wir bleiben immer noch beim Beispiel lineare und binäre Suche. Beide Algorithmen bestehen im Wesentlichen jeweils aus einer Schleife.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei großen Werten von *N* sind alle Operationen neben der Schleife für die Laufzeit unerheblich.
  - Je größer *N*, um so weniger ins Gewicht fallend.
- Die Laufzeit pro Schleifendurchlauf variiert nur in engen Grenzen.
- Konsequenz: Die Laufzeit bei linearer / binärer Suche landet mit wachsendem N schnell in einem Korridor
  - [$c_1 \cdot N$ ... $c_2 \cdot N$] bei linearer Suche im Worst Case,
  - [$c_3 \cdot \log_2 N$ ... $c_4 \cdot \log_2 N$] bei binärer Suche im Worst Case.
- Im Best Case bei beiden: [c_5 ... c_6] unabhängig von *N*.
- Dabei sind $c_1 ... c_6$ unbekannte, aber feste Konstanten:
  - unabhängig von *N*,
  - abhängig von Plattform, Programmiersprache, Compiler usw.

Alles andere – der Methodenaufruf, die Initialisierung, das Beenden der Methode – passiert nur einmal und hat zusammengenommen daher eine extrem geringe Laufzeit, die gegenüber der Schleife ab einer gewissen, sicherlich nicht übermäßig großen Problemgröße völlig unbedeutend wird.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei großen Werten von *N* sind alle Operationen neben der Schleife für die Laufzeit unerheblich.
  - Je größer *N*, um so weniger ins Gewicht fallend.
- Die Laufzeit pro Schleifendurchlauf variiert nur in engen Grenzen.
- Konsequenz: Die Laufzeit bei linearer / binärer Suche landet mit wachsendem N schnell in einem Korridor
  - [$c_1 \cdot N$ ... $c_2 \cdot N$] bei linearer Suche im Worst Case,
  - [$c_3 \cdot \log_2 N$ ... $c_4 \cdot \log_2 N$] bei binärer Suche im Worst Case.
- Im Best Case bei beiden: [c_5 ... c_6] unabhängig von *N*.
- Dabei sind $c_1 ... c_6$ unbekannte, aber feste Konstanten:
  - unabhängig von *N*,
  - abhängig von Plattform, Programmiersprache, Compiler usw.

Bei beiden Algorithmen kann die konkrete Laufzeit für einen Schleifendurchlauf natürlich von Durchlauf zu Durchlauf unterschiedlich sein. Aber die Schleifen sind so einfach, dass man sicherlich sagen kann, dass die User Time nicht allzu stark von Durchlauf zu Durchlauf schwanken dürfte.

***Erinnerung*: Im Abschnitt zu asymptotischer Komplexität betrachten wir bis auf Weiteres nur User Time, nicht System Time.**

***Nebenbemerkung*: Beim Durchlauf durch das Array kann es natürlich durch Cache-Effekte und ähnliches zu Schwankungen bei der Laufzeit pro Durchlauf kommen. Aber wenn das Array richtig groß ist, mitteln sich diese über die Gesamtlänge hinweg aus.**

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei großen Werten von *N* sind alle Operationen neben der Schleife für die Laufzeit unerheblich.
  - Je größer *N*, um so weniger ins Gewicht fallend.
- Die Laufzeit pro Schleifendurchlauf variiert nur in engen Grenzen.
- Konsequenz: Die Laufzeit bei linearer / binärer Suche landet mit wachsendem N schnell in einem Korridor
  - [$c_1 \cdot N$ ... $c_2 \cdot N$] bei linearer Suche im Worst Case,
  - [$c_3 \cdot \log_2 N$ ... $c_4 \cdot \log_2 N$] bei binärer Suche im Worst Case.
- Im Best Case bei beiden: [c_5 ... c_6] unabhängig von *N*.
- Dabei sind $c_1 ... c_6$ unbekannte, aber feste Konstanten:
  - unabhängig von *N*,
  - abhängig von Plattform, Programmiersprache, Compiler usw.

Dass die Laufzeit pro Durchlauf nur gering schwankt, kann man auch so ausdrücken: Die Laufzeiten der einzelnen Durchläufe bewegen sich in einem relativ engen Korridor.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei großen Werten von *N* sind alle Operationen neben der Schleife für die Laufzeit unerheblich.
  - Je größer *N*, um so weniger ins Gewicht fallend.
- Die Laufzeit pro Schleifendurchlauf variiert nur in engen Grenzen.
- Konsequenz: Die Laufzeit bei linearer / binärer Suche landet mit wachsendem N schnell in einem Korridor
  - [$c_1 \cdot N$ ... $c_2 \cdot N$] bei linearer Suche im Worst Case,
  - [$c_3 \cdot \log_2 N$ ... $c_4 \cdot \log_2 N$] bei binärer Suche im Worst Case.
- Im Best Case bei beiden: [c_5 ... c_6] unabhängig von *N*.
- Dabei sind c_1...c_6 unbekannte, aber feste Konstanten:
  - unabhängig von *N*,
  - abhängig von Plattform, Programmiersprache, Compiler usw.

Wenn der einzelne Durchlauf bei linearer Suche also in einem Intervall zwischen zwei Konstanten bleibt, dann bleibt die Laufzeit für *N* Durchläufe im Intervall zwischen den beiden mit N multiplizierten Konstanten.

***Nebenbemerkung*: Bei extrem großen Arrays kann es natürlich zu Cache Misses und Page Faults kommen. Im Prinzip stimmt die Argumentation dann immer noch, nur dass in die nach oben beschränkende Konstante die zusätzliche Laufzeit für Cache Misses beziehungsweise Page Faults sozusagen eingepreist werden müsste. Aber dann wären die untere und die obere Konstante um etliche Größenordnungen auseinander, so dass der Zweck der Eingrenzung sicherlich verfehlt wäre.**

Man kann sich aber eine feinere Argumentation überlegen: Auch die Anzahl der Cache Misses beziehungsweise Page Faults ist linear in der Arraygröße, nur treten diese nicht bei jeder Komponente auf, sondern nur bei jeder so-und-sovielten. Damit bleibt es beim linearen Gesamtergebnis.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei großen Werten von *N* sind alle Operationen neben der Schleife für die Laufzeit unerheblich.
  - Je größer *N*, um so weniger ins Gewicht fallend.
- Die Laufzeit pro Schleifendurchlauf variiert nur in engen Grenzen.
- Konsequenz: Die Laufzeit bei linearer / binärer Suche landet mit wachsendem N schnell in einem Korridor
  - [$c_1 \cdot N$ ... $c_2 \cdot N$] bei linearer Suche im Worst Case,
  - [$c_3 \cdot \log_2 N$ ... $c_4 \cdot \log_2 N$] bei binärer Suche im Worst Case.
- Im Best Case bei beiden: [c_5 ... c_6] unabhängig von *N*.
- Dabei sind $c_1 ... c_6$ unbekannte, aber feste Konstanten:
  - unabhängig von *N*,
  - abhängig von Plattform, Programmiersprache, Compiler usw.

Bei binärer Suche kommen wir zu einem analog konstruierten Korridor, nur eben logarithmisch statt linear.

***Nebenbemerkung*: Bezüglich Cache Misses und Page Faults wäre der Worst Case, dass jeder einzelne Durchlauf einen Page Fault erzeugt. Aufgrund der geringen Anzahl von Durchläufen durch die Schleife sind das auch bei extrem großen Arrays immer noch erfreulich wenige.**

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei großen Werten von *N* sind alle Operationen neben der Schleife für die Laufzeit unerheblich.
  - Je größer *N*, um so weniger ins Gewicht fallend.
- Die Laufzeit pro Schleifendurchlauf variiert nur in engen Grenzen.
- Konsequenz: Die Laufzeit bei linearer / binärer Suche landet mit wachsendem N schnell in einem Korridor
  - [$c_1 \cdot N$ ... $c_2 \cdot N$] bei linearer Suche im Worst Case,
  - [$c_3 \cdot \log_2 N$ ... $c_4 \cdot \log_2 N$] bei binärer Suche im Worst Case.
- Im Best Case bei beiden: [c_5 ... c_6] unabhängig von *N*.
- Dabei sind $c_1 ... c_6$ unbekannte, aber feste Konstanten:
  - unabhängig von *N*,
  - abhängig von Plattform, Programmiersprache, Compiler usw.

Wir hatten schon festgestellt, dass der Best Case bei beiden Algorithmen unabhängig von *N* ist. Mit anderen Worten: Egal wie groß *N* ist, die Laufzeit im Best Case bleibt im Intervall zwischen zwei Konstanten – die für lineare und binäre Suche natürlich unterschiedlich sein können.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei großen Werten von *N* sind alle Operationen neben der Schleife für die Laufzeit unerheblich.
  - Je größer *N*, um so weniger ins Gewicht fallend.
- Die Laufzeit pro Schleifendurchlauf variiert nur in engen Grenzen.
- Konsequenz: Die Laufzeit bei linearer / binärer Suche landet mit wachsendem N schnell in einem Korridor
  - [$c_1 \cdot N$ ... $c_2 \cdot N$] bei linearer Suche im Worst Case,
  - [$c_3 \cdot \log_2 N$ ... $c_4 \cdot \log_2 N$] bei binärer Suche im Worst Case.
- Im Best Case bei beiden: [c_5 ... c_6] unabhängig von *N*.
- Dabei sind $c_1 ... c_6$ unbekannte, aber feste Konstanten:
  - unabhängig von *N*,
  - abhängig von Plattform, Programmiersprache, Compiler usw.

Wenn wir nur theoretische Betrachtungen betreiben, ohne echte Laufzeiten zu messen, dann wissen wir natürlich nicht, wie groß diese Konstanten sind. Das ist aber für rein asymptotische Betrachtungen egal.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei großen Werten von *N* sind alle Operationen neben der Schleife für die Laufzeit unerheblich.
  - Je größer *N*, um so weniger ins Gewicht fallend.
- Die Laufzeit pro Schleifendurchlauf variiert nur in engen Grenzen.
- Konsequenz: Die Laufzeit bei linearer / binärer Suche landet mit wachsendem N schnell in einem Korridor
  - [$c_1 \cdot N$ ... $c_2 \cdot N$] bei linearer Suche im Worst Case,
  - [$c_3 \cdot \log_2 N$ ... $c_4 \cdot \log_2 N$] bei binärer Suche im Worst Case.
- Im Best Case bei beiden: [c_5 ... c_6] unabhängig von *N*.
- Dabei sind $c_1 ... c_6$ unbekannte, aber feste Konstanten:
  - unabhängig von *N*,
  - abhängig von Plattform, Programmiersprache, Compiler usw.

Wichtig ist, dass die Konstanten nicht in irgendeiner Weise von *N* abhängen, sondern eben echte Konstanten sind. Denn würden sie beispielsweise linear in *N* wachsen, dann wäre die asymptotische Komplexität vom linearer Suche nicht linear, sondern quadratisch. Mit anderen Worten: Sind die Konstanten von *N* abhängig, dann ist die asymptotische Analyse noch nicht konsequent durchgeführt.

Worst und Best Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei großen Werten von *N* sind alle Operationen neben der Schleife für die Laufzeit unerheblich.
  - Je größer *N*, um so weniger ins Gewicht fallend.
- Die Laufzeit pro Schleifendurchlauf variiert nur in engen Grenzen.
- Konsequenz: Die Laufzeit bei linearer / binärer Suche landet mit wachsendem N schnell in einem Korridor
  - [$c_1 \cdot N$ ... $c_2 \cdot N$] bei linearer Suche im Worst Case,
  - [$c_3 \cdot \log_2 N$ ... $c_4 \cdot \log_2 N$] bei binärer Suche im Worst Case.
- Im Best Case bei beiden: [c_5 ... c_6] unabhängig von *N*.
- Dabei sind $c_1 ... c_6$ unbekannte, aber feste Konstanten:
  - unabhängig von *N*,
  - abhängig von Plattform, Programmiersprache, Compiler usw.

Selbstverständlich sind die Werte für die Konstanten, die man mit Laufzeitstudien herausbekommt, abhängig von den konkreten Umständen, unter denen der Prozess läuft.

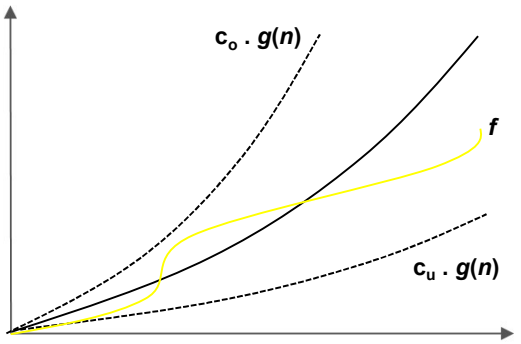
Schreibweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- Seien $f : \mathbb{N} \rightarrow \mathbb{R}$ und $g : \mathbb{N} \rightarrow \mathbb{R}$.
- Annahme: Es gibt beliebige, aber feste $c_u, c_o \in \mathbb{R}$ $(0 < c_u \leq c_o)$, so dass ab einer gewissen Größe der Eingabe n gilt

$$c_u \cdot g(n) \leq f(n) \leq c_o \cdot g(n).$$

- Dann schreiben wir: $f \in \Theta(g)$.
- Bei einer konstanten Funktion g schreiben wir: $f \in \Theta(1)$.
- Wieder Beispiel Laufzeit bei linearer / binärer Suche:
  - Lineare Suche im Worst Case: $\in \Theta(N)$
  - Binäre Suche im Worst Case: $\in \Theta(\log_2 N)$
  - Beide im Best Case: $\in \Theta(1)$



Wir müssen nun noch die gängige Notation für das, was wir bisher gesagt hatten, einführen

So wie in der Skizze kann man sich das bildlich vorstellen: Wenn es einen Korridor um die eine Funktion gibt, der von der anderen Funktion nicht verlassen wird, dann sind die beiden Funktionen asymptotisch gleich. Im Bild ist eine der beiden Funktionen die durchgezogene schwarze Linie, und der Korridor darum ist durch die beiden gestrichelten Linien angedeutet. Die andere Funktion ist die gelbe Linie.

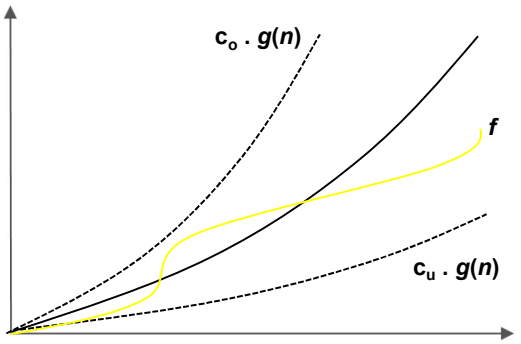
Schreibweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- Seien $f : \mathbb{N} \rightarrow \mathbb{R}$ und $g : \mathbb{N} \rightarrow \mathbb{R}$.
- Annahme: Es gibt beliebige, aber feste $c_u, c_o \in \mathbb{R}$ $(0 < c_u \leq c_o)$, so dass ab einer gewissen Größe der Eingabe n gilt

$$c_u \cdot g(n) \leq f(n) \leq c_o \cdot g(n).$$

- Dann schreiben wir: $f \in \Theta(g)$.
- Bei einer konstanten Funktion g schreiben wir: $f \in \Theta(1)$.
- Wieder Beispiel Laufzeit bei linearer / binärer Suche:
  - Lineare Suche im Worst Case: $\in \Theta(N)$
  - Binäre Suche im Worst Case: $\in \Theta(\log_2 N)$
  - Beide im Best Case: $\in \Theta(1)$



Mathematisch gesprochen, haben wir die ganze Zeit *zwei* Funktionen in der Problemgröße betrachtet: erstens die Laufzeit selbst im Worst Case oder Best Case als Funktion der Problemgröße; zweitens die mathematische Funktion, die diese Laufzeit beschreiben soll, also konstant, logarithmisch oder linear. Konkret hat sich ergeben, dass die beiden Funktionen nur in einem Korridor voneinander abweichen. Allgemein kann das so wie hier farblich unterlegt formulieren.

Wir sagen, die beiden Funktionen *f* und *g* sind *asymptotisch äquivalent*.

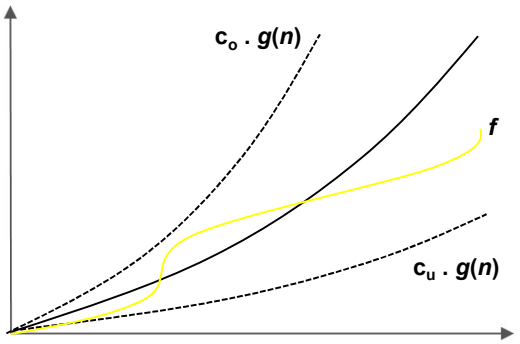
Schreibweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- Seien $f : \mathbb{N} \rightarrow \mathbb{R}$ und $g : \mathbb{N} \rightarrow \mathbb{R}$.
- Annahme: Es gibt beliebige, aber feste $c_u, c_o \in \mathbb{R}$ $(0 < c_u \leq c_o)$, so dass ab einer gewissen Größe der Eingabe n gilt

$$c_u \cdot g(n) \leq f(n) \leq c_o \cdot g(n).$$

- Dann schreiben wir: $f \in \Theta(g)$.
- Bei einer konstanten Funktion g schreiben wir: $f \in \Theta(1)$.
- Wieder Beispiel Laufzeit bei linearer / binärer Suche:
  - Lineare Suche im Worst Case: $\in \Theta(N)$
  - Binäre Suche im Worst Case: $\in \Theta(\log_2 N)$
  - Beide im Best Case: $\in \Theta(1)$



Das ist die angekündigte Notation: der griechische Großbuchstabe Theta. Mit Theta wird also offenbar eine Menge von Funktionen benannt, nämlich alle Funktionen, die asymptotisch äquivalent zum Parameter sind, hier also alle Funktionen, die asymptotisch äquivalent zu g sind.

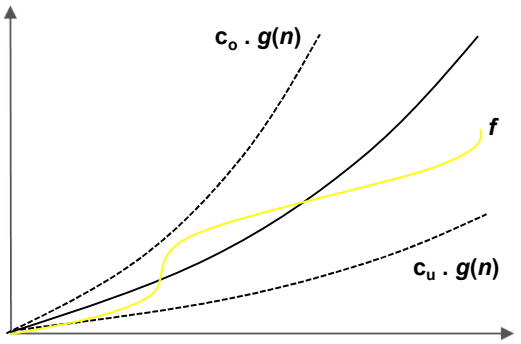
Schreibweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- Seien $f : \mathbb{N} \rightarrow \mathbb{R}$ und $g : \mathbb{N} \rightarrow \mathbb{R}$.
- Annahme: Es gibt beliebige, aber feste $c_u, c_o \in \mathbb{R}$ $(0 < c_u \leq c_o)$, so dass ab einer gewissen Größe der Eingabe n gilt

$$c_u \cdot g(n) \leq f(n) \leq c_o \cdot g(n).$$

- Dann schreiben wir: $f \in \Theta(g)$.
- Bei einer konstanten Funktion g schreiben wir: $f \in \Theta(1)$.
- Wieder Beispiel Laufzeit bei linearer / binärer Suche:
  - Lineare Suche im Worst Case: $\in \Theta(N)$
  - Binäre Suche im Worst Case: $\in \Theta(\log_2 N)$
  - Beide im Best Case: $\in \Theta(1)$



Speziell *diese* Schreibweise hat sich eingebürgert für den Fall, dass die Vergleichsfunktion konstant ist. Das heißt, f bleibt in einem horizontalen, nichtsteigenden Korridor.

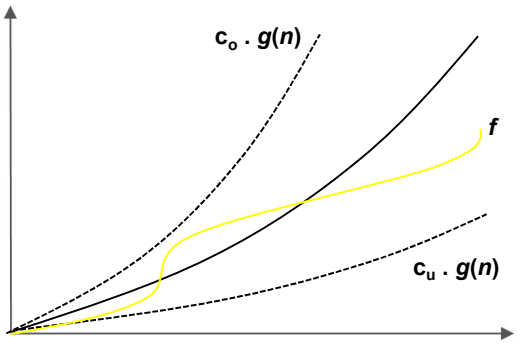
Schreibweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- Seien $f : \mathbb{N} \rightarrow \mathbb{R}$ und $g : \mathbb{N} \rightarrow \mathbb{R}$.
- Annahme: Es gibt beliebige, aber feste $c_u, c_o \in \mathbb{R}$ $(0 < c_u \leq c_o)$, so dass ab einer gewissen Größe der Eingabe n gilt

$$c_u \cdot g(n) \leq f(n) \leq c_o \cdot g(n).$$

- Dann schreiben wir: $f \in \Theta(g)$.
- Bei einer konstanten Funktion g schreiben wir: $f \in \Theta(1)$.
- Wieder Beispiel Laufzeit bei linearer / binärer Suche:
  - Lineare Suche im Worst Case: $\in \Theta(N)$
  - Binäre Suche im Worst Case: $\in \Theta(\log_2 N)$
  - Beide im Best Case: $\in \Theta(1)$



Unser momentanes Beispiel dient weiterhin gut zur Illustration.

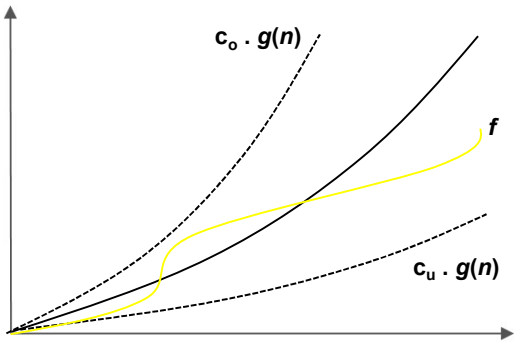
Schreibweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- Seien $f : \mathbb{N} \rightarrow \mathbb{R}$ und $g : \mathbb{N} \rightarrow \mathbb{R}$.
- Annahme: Es gibt beliebige, aber feste $c_u, c_o \in \mathbb{R}$ $(0 < c_u \leq c_o)$, so dass ab einer gewissen Größe der Eingabe n gilt

$$c_u \cdot g(n) \leq f(n) \leq c_o \cdot g(n).$$

- Dann schreiben wir: $f \in \Theta(g)$.
- Bei einer konstanten Funktion g schreiben wir: $f \in \Theta(1)$.
- Wieder Beispiel Laufzeit bei linearer / binärer Suche:
  - Lineare Suche im Worst Case: $\in \Theta(N)$
  - Binäre Suche im Worst Case: $\in \Theta(\log_2 N)$
  - Beide im Best Case: $\in \Theta(1)$



Diese drei Aussagen sind identisch mit den schon festgestellten Tatsachen zum Worst und Best Case bei beiden Algorithmen, nur jetzt mit der neuen Notation.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n > 2.
// Returns: true iff n is a prime number, that is,
//              n only has two factors: 1 and n.

boolean isPrime ( int n ) {
    if ( n % 2 == 0 )
      return false;
    final int maxPossibleDivisor = (int) Math.floor ( Math.sqrt(n) );
    for ( int i = 3; i <= maxPossibleDivisor; i += 2 )
        if ( n % i == 0 )
            return false;
    return true;
}
```

Ein weiteres Fallbeispiel: testen, ob eine gegebene positive ganze Zahl eine Primzahl ist oder nicht.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n > 2.
// Returns: true iff n is a prime number, that is,
//              n only has two factors: 1 and n.

boolean isPrime ( int n ) {
    if ( n % 2 == 0 )
      return false;
    final int maxPossibleDivisor = (int) Math.floor ( Math.sqrt(n) );
    for ( int i = 3; i <= maxPossibleDivisor; i += 2 )
        if ( n % i == 0 )
            return false;
    return true;
}
```

Da wir hier nicht wirklich an Primzahlberechnung interessiert sind, sondern dies nur ein Beispiel für das Thema asymptotische Komplexität ist, dürfen wir den Code etwas vereinfachen, indem wir die beiden speziell zu behandelnden Fälle n gleich 1 beziehungsweise n gleich 2 von vornherein ausschließen.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n > 2.
// Returns: true iff n is a prime number, that is,
//              n only has two factors: 1 and n.

boolean isPrime ( int n ) {
   if ( n % 2 == 0 )
      return false;
   final int maxPossibleDivisor = (int) Math.floor ( Math.sqrt(n) );
   for ( int i = 3; i <= maxPossibleDivisor; i += 2 )
       if ( n % i == 0 )
           return false;
   return true;
}
```

Die durch 2 teilbaren Zahlen können wir vorab ausschließen.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n > 2.
// Returns: true iff n is a prime number, that is,
//              n only has two factors: 1 and n.

boolean isPrime ( int n ) {
    if ( n % 2 == 0 )
      return false;
    final int maxPossibleDivisor = (int) Math.floor ( Math.sqrt(n) );
    for ( int i = 3; i <= maxPossibleDivisor; i += 2 )
        if ( n % i == 0 )
            return false;
    return true;
}
```

Die entscheidende Einsicht ist, dass kein echter Teiler einer natürlichen Zahl größer als die Quadratwurzel dieser Zahl ist.

Beachten Sie, dass die Methode floor von Klasse Math zwar eine ganze Zahl zurückliefert, nämlich die nächstkleinere, aber im Datentyp double, so dass die Rückgabe nach int konvertiert werden muss, um sie in einer int-Variablen zu speichern.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n > 2.
// Returns: true iff n is a prime number, that is,
//              n only has two factors: 1 and n.

boolean isPrime ( int n ) {
    if ( n % 2 == 0 )
      return false;
    final int maxPossibleDivisor = (int) Math.floor ( Math.sqrt(n) );
    for ( int i = 3; i <= maxPossibleDivisor; i += 2 )
        if ( n % i == 0 )
            return false;
    return true;
}
```

Da wir 2 als Teiler schon vor der Schleife abgeprüft haben, reicht es, alle ungeraden Zahlen von der 3 an bis zur Quadratwurzel als Teiler zu prüfen.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n > 2.
// Returns: true iff n is a prime number, that is,
//              n only has two factors: 1 and n.

boolean isPrime ( int n ) {
    if ( n % 2 == 0 )
      return false;
    final int maxPossibleDivisor = (int) Math.floor ( Math.sqrt(n) );
    for ( int i = 3; i <= maxPossibleDivisor; i += 2 )
        if ( n % i == 0 )
            return false;
    return true;
}
```

Und wenn kein Teiler gefunden wird, ist die gegebene Zahl prim.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: der Wert des Parameters *n*.**
- **Die Quadratwurzelberechnung wird nur einmal ausgeführt.**
  - **Unerheblich bei großen Zahlen *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist**
  - **im Worst Case ca. $\sqrt{n}$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *n* abhängigen Korridor.**
- **Also: Die Laufzeit des Primzahltests ist**
  - **im Worst Case $\in \Theta(\sqrt{n})$,**
  - **im Best Case $\in \Theta(1)$.**

Wie sieht nun die asymptotische Komplexität dieser Methode im Worst Case und im Best Case aus?

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- Problemgröße: der Wert des Parameters *n*.
- Die Quadratwurzelberechnung wird nur einmal ausgeführt.
  - Unerheblich bei großen Zahlen *n*.
- Die Anzahl der Durchläufe durch die Schleife ist
  - im Worst Case ca. $\sqrt{n}$,
  - im Best Case 0.
- Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *n* abhängigen Korridor.
- Also: Die Laufzeit des Primzahltests ist
  - im Worst Case $\in \Theta(\sqrt{n})$,
  - im Best Case $\in \Theta(1)$.

Zunächst ist zu klären, welche Kennzahl die Problemgröße zielführend beschreibt. Bei linearer und binärer Suche war die Wahl der Kennzahl offensichtlich: die Länge des Arrays.

Da die maximale Anzahl von Durchläufen durch die Schleife hier von der eingegebenen natürlichen Zahl abhängt, bietet sich die Größe dieser Zahl an.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: der Wert des Parameters *n*.**
- **Die Quadratwurzelberechnung wird nur einmal ausgeführt.**
  - **Unerheblich bei großen Zahlen *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist**
  - **im Worst Case ca. $\sqrt{n}$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *n* abhängigen Korridor.**
- **Also: Die Laufzeit des Primzahltests ist**
  - **im Worst Case $\in \Theta(\sqrt{n})$,**
  - **im Best Case $\in \Theta(1)$.**

Mit der Quadratwurzelberechnung haben wir eine einzelne sehr komplexe Berechnung, die wir an eine Methode aus der Standardbibliothek delegiert haben, so dass wir auch nicht wissen, was die genau tut. Mit etwas Vertrauen auf den gesunden Menschenverstand und das mathematische Hintergrundwissen der Entwickler dieser Methode kann man aber sehr sicher sein, dass der Aufruf dieser Methode bei ernsthaften Problemgrößen nicht ins Gewicht fallen wird.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- Problemgröße: der Wert des Parameters *n*.
- Die Quadratwurzelberechnung wird nur einmal ausgeführt.
  - Unerheblich bei großen Zahlen *n*.
- Die Anzahl der Durchläufe durch die Schleife ist
  - im Worst Case ca. $\sqrt{n}$,
  - im Best Case 0.
- Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *n* abhängigen Korridor.
- Also: Die Laufzeit des Primzahltests ist
  - im Worst Case $\in \Theta(\sqrt{n})$,
  - im Best Case $\in \Theta(1)$.

Die Anzahl Durchläufe im Worst Case und Best Case lassen sich direkt aus der Schleife ablesen: Der beste Fall ist, dass die Schleife gar nicht erst beginnt, weil die eingegebene Zahl durch 2 teilbar ist. Der schlechteste Fall ist, dass die Zahl tatsächlich prim ist, denn dann wird die Schleife bis zur Abbruchbedingung durchlaufen.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: der Wert des Parameters *n*.**
- **Die Quadratwurzelberechnung wird nur einmal ausgeführt.**
  - **Unerheblich bei großen Zahlen *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist**
  - **im Worst Case ca. $\sqrt{n}$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *n* abhängigen Korridor.**
- **Also: Die Laufzeit des Primzahltests ist**
  - **im Worst Case $\in \Theta(\sqrt{n})$,**
  - **im Best Case $\in \Theta(1)$.**

Auch hier kann man wieder davon ausgehen, dass die Laufzeiten der einzelnen Schleifendurchläufe nicht allzu sehr variieren.

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: der Wert des Parameters *n*.**
- **Die Quadratwurzelberechnung wird nur einmal ausgeführt.**
  - **Unerheblich bei großen Zahlen *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist**
  - **im Worst Case ca. $\sqrt{n}$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *n* abhängigen Korridor.**
- **Also: Die Laufzeit des Primzahltests ist**
  - **im Worst Case $\in \Theta(\sqrt{n})$,**
  - **im Best Case $\in \Theta(1)$.**

Im Worst Case ist die Anzahl der Schleifendurchläufe und damit die Gesamtlaufzeit ausreichend gut durch die Quadratwurzel der Eingabezahl beschrieben, ...

Primzahltest

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: der Wert des Parameters *n*.**
- **Die Quadratwurzelberechnung wird nur einmal ausgeführt.**
  - **Unerheblich bei großen Zahlen *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist**
  - **im Worst Case ca. $\sqrt{n}$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *n* abhängigen Korridor.**
- **Also: Die Laufzeit des Primzahltests ist**
  - **im Worst Case $\in \Theta(\sqrt{n})$,**
  - **im Best Case $\in \Theta(1)$.**

... im Best Case hängt die Gesamtlaufzeit auch hier wieder nicht von der Problemgröße ab.

Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void selectionSort ( T[ ] a, Comparator<T> cmp ) {
   for ( int i = a.length – 1; i > 0; i – – ) {
      int indexOfMax = 0;
      for ( int j = 1; j <= i; j++ )
         if ( cmp.compare ( a[indexOfMax], a[j] ) <= 0 )
            indexOfMax = j;
      if ( i != indexOfMax ) {
         T tmp = a[i];
         a[i] = a[indexOfMax];
         a[indexOfMax] = tmp;
      }
   }
}
```

Als nächstes schauen wir uns ein Beispiel an, bei dem die Laufzeit auch im Best Case höher als linear ist.

***Erinnerung*: Wir hatten diesen Algorithmus namens Selection Sort schon in Kapitel gesehen, Abschnitt zur Korrektheit von Schleifen. Hier erläutern wir diesen Algorithmus nicht nochmals, ...**

Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: die Länge *N* des Arrays a.**
- **Die äußere Schleife wird (N–1)-mal durchlaufen.**
- **Im h-ten Durchlauf der äußeren Schleife wird die innere Schleife (N–h+1)-mal durchlaufen.**
  - **Insgesamt (N–1) + (N–2) + (N–3) + ... + 3 + 2 + 1 = N · (N–1) / 2 Durchläufe.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *N* abhängigen Korridor.**
- **Also: Die Laufzeit von Selection Sort ist $\Theta(N^2)$ sowohl im Worst Case als auch im Best Case.**

... sondern wenden uns gleich der asymptotischen Analyse zu.

Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: die Länge *N* des Arrays a.**
- **Die äußere Schleife wird (N–1)-mal durchlaufen.**
- **Im h-ten Durchlauf der äußeren Schleife wird die innere Schleife (N–h+1)-mal durchlaufen.**
  - **Insgesamt (N–1) + (N–2) + (N–3) + ... + 3 + 2 + 1 = N · (N–1) / 2 Durchläufe.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *N* abhängigen Korridor.**
- **Also: Die Laufzeit von Selection Sort ist $\Theta(N^2)$ sowohl im Worst Case als auch im Best Case.**

Es bietet sich an, die Anzahl der zu sortierenden Elemente als Maßstab zu nehmen.

Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: die Länge *N* des Arrays a.**
- **Die äußere Schleife wird (N–1)-mal durchlaufen.**
- **Im h-ten Durchlauf der äußeren Schleife wird die innere Schleife (N–h+1)-mal durchlaufen.**
  - **Insgesamt (N–1) + (N–2) + (N–3) + ... + 3 + 2 + 1 = N · (N–1) / 2 Durchläufe.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *N* abhängigen Korridor.**
- **Also: Die Laufzeit von Selection Sort ist $\Theta(N^2)$ sowohl im Worst Case als auch im Best Case.**

Der Algorithmus besteht im Wesentlichen aus zwei ineinander geschachtelten Schachteln. Bei der äußeren Schleife kann man die Anzahl der Durchläufe unmittelbar im Kopf der Schleife ablesen.

Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: die Länge *N* des Arrays a.**
- **Die äußere Schleife wird (N–1)-mal durchlaufen.**
- **Im h-ten Durchlauf der äußeren Schleife wird die innere Schleife (N–h+1)-mal durchlaufen.**
  - **Insgesamt (N–1) + (N–2) + (N–3) + ... + 3 + 2 + 1 = N · (N–1) / 2 Durchläufe.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *N* abhängigen Korridor.**
- **Also: Die Laufzeit von Selection Sort ist $\Theta(N^2)$ sowohl im Worst Case als auch im Best Case.**

Die Anzahl der Durchläufe der inneren Schleife ist hingegen vom Laufindex der äußeren Schleife abhängig, offensichtlich aber nach einem einfachen Prinzip.

Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: die Länge *N* des Arrays a.**
- **Die äußere Schleife wird (N–1)-mal durchlaufen.**
- **Im h-ten Durchlauf der äußeren Schleife wird die innere Schleife (N–h+1)-mal durchlaufen.**
  - ➢ **Insgesamt (N–1) + (N–2) + (N–3) + ... + 3 + 2 + 1 = N · (N–1) / 2 Durchläufe.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *N* abhängigen Korridor.**
- **Also: Die Laufzeit von Selection Sort ist $\Theta(N^2)$ sowohl im Worst Case als auch im Best Case.**

Im Ergebnis kommt jede Zahl von 1 bis N minus 1 genau einmal als Anzahl Durchläufe durch die innere Schleife vor. Im Durchschnitt sind das N halbe viele Durchläufe. Die Gesamtzahl Durchläufe durch die innere Schleife – über alle Durchläufe der äußeren Schleife aufsummiert – ist daher ziemlich genau die Hälfte vom Quadrat von N. Diese Gleichung finden Sie auch in jeder Formelsammlung (meist bis N statt N-1).

Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: die Länge *N* des Arrays a.**
- **Die äußere Schleife wird (N–1)-mal durchlaufen.**
- **Im h-ten Durchlauf der äußeren Schleife wird die innere Schleife (N–h+1)-mal durchlaufen.**
  - **Insgesamt (N–1) + (N–2) + (N–3) + ... + 3 + 2 + 1 = N · (N–1) / 2 Durchläufe.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *N* abhängigen Korridor.**
- **Also: Die Laufzeit von Selection Sort ist $\Theta(N^2)$ sowohl im Worst Case als auch im Best Case.**

Die Laufzeit für die einzelnen Durchläufe durch die innere Schleife variiert wieder nicht allzu stark, ...

Selection Sort

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit:

- **Problemgröße: die Länge *N* des Arrays a.**
- **Die äußere Schleife wird (N–1)-mal durchlaufen.**
- **Im h-ten Durchlauf der äußeren Schleife wird die innere Schleife (N–h+1)-mal durchlaufen.**
  - **Insgesamt (N–1) + (N–2) + (N–3) + ... + 3 + 2 + 1 = N · (N–1) / 2 Durchläufe.**
- **Die Laufzeit pro Schleifendurchlauf bleibt in einem engen, nicht von *N* abhängigen Korridor.**
- **Also: Die Laufzeit von Selection Sort ist $\Theta(N^2)$ sowohl im Worst Case als auch im Best Case.**

... so dass die asymptotische Komplexität rein durch die Gesamtsumme aller Durchläufe durch die innere Schleife dominiert wird. In diesem Beispiel gehen Worst Case und Best Case *nicht* asymptotisch auseinander.

Grenzen der Asymptotik

TECHNISCHE UNIVERSITÄT DARMSTADT

- Asymptotische Laufzeitbetrachtungen sind ***nicht*** sinnvoll, wenn es um kleine Problemgrößen geht.
  - Also kleine Werte für die Kennzahl.
- Wenn sehr viele Aufgaben dieser Art zu lösen sind, kann das durchaus das Laufzeitproblem sein.
- Beispiel: Millionen sortierte Arrays mit einstelliger Elementzahl nach jeweils einem Element zu durchsuchen.
- Lineare Suche ist mutmaßlich schneller als binäre Suche aufgrund der mutmaßlich geringeren Laufzeit pro Schleifendurchlauf.

Jetzt haben wir gesehen, wofür asymptotische Komplexität gut ist. Aber natürlich hat das Konzept auch Grenzen der Anwendbarkeit.

Grenzen der Asymptotik

TECHNISCHE UNIVERSITÄT DARMSTADT

- Asymptotische Laufzeitbetrachtungen sind ***nicht*** sinnvoll, wenn es um kleine Problemgrößen geht.
  - ➢ Also kleine Werte für die Kennzahl.
- Wenn sehr viele Aufgaben dieser Art zu lösen sind, kann das durchaus das Laufzeitproblem sein.
- Beispiel: Millionen sortierte Arrays mit einstelliger Elementzahl nach jeweils einem Element zu durchsuchen.
- Lineare Suche ist mutmaßlich schneller als binäre Suche aufgrund der mutmaßlich geringeren Laufzeit pro Schleifendurchlauf.

Asymptotik ist ja naturgemäß ein Aspekt, der nur bei großen Zahlen eine Aussagekraft hat.

Grenzen der Asymptotik

TECHNISCHE UNIVERSITÄT DARMSTADT

- Asymptotische Laufzeitbetrachtungen sind ***nicht*** sinnvoll, wenn es um kleine Problemgrößen geht.
  - ➢ Also kleine Werte für die Kennzahl.
- Wenn sehr viele Aufgaben dieser Art zu lösen sind, kann das durchaus das Laufzeitproblem sein.
- Beispiel: Millionen sortierte Arrays mit einstelliger Elementzahl nach jeweils einem Element zu durchsuchen.
- Lineare Suche ist mutmaßlich schneller als binäre Suche aufgrund der mutmaßlich geringeren Laufzeit pro Schleifendurchlauf.

Nun kann man natürlich sagen, ist doch egal, bei kleinen Problemgrößen ist die Laufzeit doch eh verschwindend gering.

Aber es gibt auch Anwendungsfälle, in denen sehr viele Probleme von sehr kleiner Größe zu lösen sind. Die akkumulierte Laufzeit kann dann natürlich schon ins Gewicht fallen.

Grenzen der Asymptotik

TECHNISCHE UNIVERSITÄT DARMSTADT

- Asymptotische Laufzeitbetrachtungen sind ***nicht*** sinnvoll, wenn es um kleine Problemgrößen geht.
  - ➢ Also kleine Werte für die Kennzahl.
- Wenn sehr viele Aufgaben dieser Art zu lösen sind, kann das durchaus das Laufzeitproblem sein.
- Beispiel: Millionen sortierte Arrays mit einstelliger Elementzahl nach jeweils einem Element zu durchsuchen.
- Lineare Suche ist mutmaßlich schneller als binäre Suche aufgrund der mutmaßlich geringeren Laufzeit pro Schleifendurchlauf.

Dafür nehmen wir wieder lineare und binäre Suche als Beispiel.

Grenzen der Asymptotik

TECHNISCHE UNIVERSITÄT DARMSTADT

- Asymptotische Laufzeitbetrachtungen sind ***nicht*** sinnvoll, wenn es um kleine Problemgrößen geht.
  - ➢ Also kleine Werte für die Kennzahl.
- Wenn sehr viele Aufgaben dieser Art zu lösen sind, kann das durchaus das Laufzeitproblem sein.
- Beispiel: Millionen sortierte Arrays mit einstelliger Elementzahl nach jeweils einem Element zu durchsuchen.
- Lineare Suche ist mutmaßlich schneller als binäre Suche aufgrund der mutmaßlich geringeren Laufzeit pro Schleifendurchlauf.

Wenn das Array so klein ist, dass die logarithmische Anzahl Durchläufe bei binärer Suche nicht nennenswert kleiner als die lineare Anzahl Durchläufe bei linearer Suche ist, dann wird wohl eher die Laufzeit pro Durchlauf entscheiden.

Wie sich die beiden Laufzeiten genau verhalten und bis zu welcher Größe lineare Suche besser ist, hängt natürlich von der Hardware, der Programmiersprache, der Geschicklichkeit der Softwareentwickler und so weiter ab.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Oft lässt sich das asymptotische Verhalten $f : \mathbb{N} \rightarrow \mathbb{R}^+$ eines Programms im Worst oder Best Case nicht genau einschätzen.
- Behelf: obere und untere Schranken.
- Das heißt: zwei mathematische Funktionen g_u und g_o, so dass f asymptotisch
  - *mindestens* so schnell wie g_u und
  - *höchstens* so schnell wie g_o wächst.
- Bis jetzt haben wir als asymptotischen Vergleich nur „$\Theta(\cdot)$“.
  - Für asymptotische Gleichheit.
- Wir brauchen auch eine Notation für kleiner / größer.
  - Nächste Folie.

Es gibt noch ein grundsätzliches methodisches Problem bei der Bestimmung der mathematischen Funktion, die das asymptotische Laufzeitverhalten beschreiben soll.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Oft lässt sich das asymptotische Verhalten $f : \mathbb{N} \rightarrow \mathbb{R}^+$ eines Programms im Worst oder Best Case nicht genau einschätzen.**
- **Behelf: obere und untere Schranken.**
- **Das heißt: zwei mathematische Funktionen g_u und g_o, so dass f asymptotisch**
  - ***mindestens* so schnell wie g_u und**
  - ***höchstens* so schnell wie g_o wächst.**
- **Bis jetzt haben wir als asymptotischen Vergleich nur „$\Theta(\cdot)$“.**
  - **Für asymptotische Gleichheit.**
- **Wir brauchen auch eine Notation für kleiner / größer.**
  - **Nächste Folie.**

Tatsächlich wird man es nicht immer schaffen können, die korrekte mathematische Funktion zu finden.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Oft lässt sich das asymptotische Verhalten $f : \mathbb{N} \rightarrow \mathbb{R}^+$ eines Programms im Worst oder Best Case nicht genau einschätzen.
- Behelf: obere und untere Schranken.
- Das heißt: zwei mathematische Funktionen g_u und g_o, so dass f asymptotisch
  - *mindestens* so schnell wie g_u und
  - *höchstens* so schnell wie g_o wächst.
- Bis jetzt haben wir als asymptotischen Vergleich nur „$\Theta(\cdot)$“.
  - Für asymptotische Gleichheit.
- Wir brauchen auch eine Notation für kleiner / größer.
  - Nächste Folie.

Man kann sie dann aber immer noch eingrenzen, und das Ziel besteht darin, eine möglichst enge Eingrenzung zu finden.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Oft lässt sich das asymptotische Verhalten $f : \mathbb{N} \rightarrow \mathbb{R}^+$ eines Programms im Worst oder Best Case nicht genau einschätzen.
- Behelf: obere und untere Schranken.
- Das heißt: zwei mathematische Funktionen g_u und g_o, so dass f asymptotisch
  - *mindestens* so schnell wie g_u und
  - *höchstens* so schnell wie g_o wächst.
- Bis jetzt haben wir als asymptotischen Vergleich nur „$\Theta(\cdot)$“.
  - Für asymptotische Gleichheit.
- Wir brauchen auch eine Notation für kleiner / größer.
  - Nächste Folie.

Anstelle von *einer* mathematischen Funktion, die die Asymptotik *genau* beschreibt, reden wir also von *zwei* mathematischen Funktionen, die die Asymptotik von oben und von unten eingrenzen.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Oft lässt sich das asymptotische Verhalten $f : \mathbb{N} \rightarrow \mathbb{R}^+$ eines Programms im Worst oder Best Case nicht genau einschätzen.
- Behelf: obere und untere Schranken.
- Das heißt: zwei mathematische Funktionen g_u und g_o, so dass f asymptotisch
  - *mindestens* so schnell wie g_u und
  - *höchstens* so schnell wie g_o wächst.
- Bis jetzt haben wir als asymptotischen Vergleich nur „$\Theta(\cdot)$“.
  - Für asymptotische Gleichheit.
- Wir brauchen auch eine Notation für kleiner / größer.
  - Nächste Folie.

Für exakten Vergleich hatten wir schon eine Notation, nämlich mit dem Theta.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

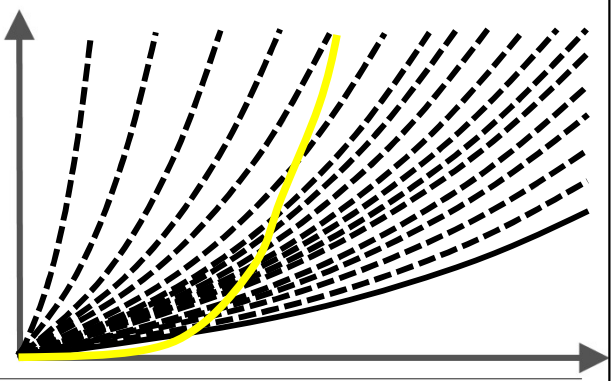
- Oft lässt sich das asymptotische Verhalten $f : \mathbb{N} \rightarrow \mathbb{R}^+$ eines Programms im Worst oder Best Case nicht genau einschätzen.
- Behelf: obere und untere Schranken.
- Das heißt: zwei mathematische Funktionen g_u und g_o, so dass f asymptotisch
  - *mindestens* so schnell wie g_u und
  - *höchstens* so schnell wie g_o wächst.
- Bis jetzt haben wir als asymptotischen Vergleich nur „$\Theta(\cdot)$“.
  - Für asymptotische Gleichheit.
- Wir brauchen auch eine Notation für kleiner / größer.
  - Nächste Folie.

Für die Abschätzung nach oben und nach unten haben sich ebenfalls Notationen in der Informatik etabliert.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$.
- Wir schreiben $f \in o(g)$, wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- Erste Beobachtung: $f \in o(g)$ und $f \in \Theta(g)$ schließen sich gegenseitig logisch aus.
- Zweite Beobachtung: Es gibt auch Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$, so dass weder $f \in o(g)$ noch $f \in \Theta(g)$ gilt.
- Einfaches Beispiel für die zweite Beobachtung:
  - $f : n \rightarrow | n \cdot \sin(n) |$
  - $g : n \rightarrow | n \cdot \cos(n) |$

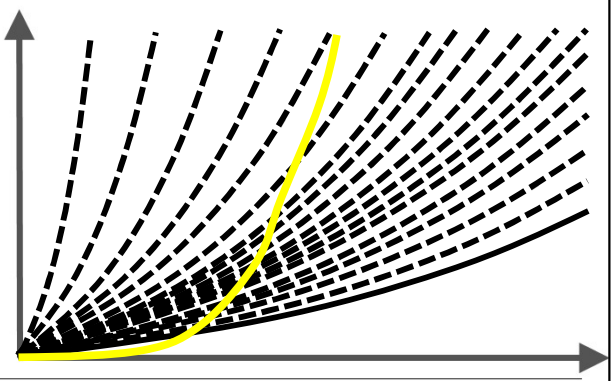


Das sehen wir uns auf dieser Folie an.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$.
- Wir schreiben $f \in o(g)$, wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- Erste Beobachtung: $f \in o(g)$ und $f \in \Theta(g)$ schließen sich gegenseitig logisch aus.
- Zweite Beobachtung: Es gibt auch Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$, so dass weder $f \in o(g)$ noch $f \in \Theta(g)$ gilt.
- Einfaches Beispiel für die zweite Beobachtung:
  - $f : n \rightarrow | n \cdot \sin(n) |$
  - $g : n \rightarrow | n \cdot \cos(n) |$

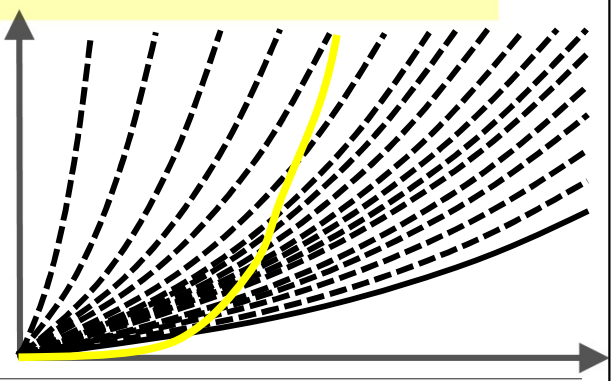


Zunächst einmal die gängige Notation für den Fall, dass eine Funktion echt schneller wächst als eine andere. Diese Definition über eine Limesbildung trifft sicherlich die intuitive Vorstellung von echt unterschiedlichem asymptotischem Wachstum.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$.
- Wir schreiben $f \in o(g)$, wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- Erste Beobachtung: $f \in o(g)$ und $f \in \Theta(g)$ schließen sich gegenseitig logisch aus.
- Zweite Beobachtung: Es gibt auch Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$, so dass weder $f \in o(g)$ noch $f \in \Theta(g)$ gilt.
- Einfaches Beispiel für die zweite Beobachtung:
  - $f : n \rightarrow | n \cdot \sin(n) |$
  - $g : n \rightarrow | n \cdot \cos(n) |$

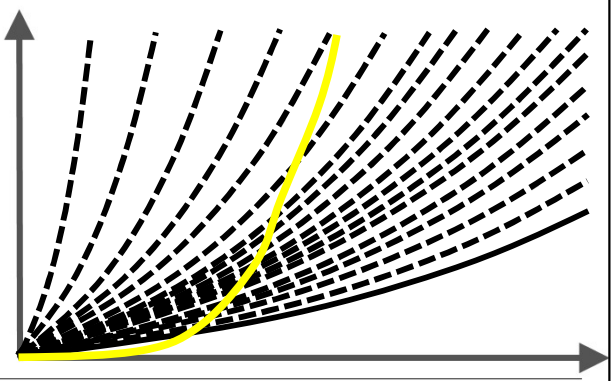


Mit Theta drücken wir ja aus, dass zwei Funktionen asymptotisch gleich schnell sind. Gleiche Asymptotik und echt unterschiedliche Asymptotik sollten sich natürlich ausschließen, sonst haben wir sie noch nicht geeignet definiert. Aber offensichtlich schließen sie sich gegenseitig aus, denn Theta bedeutet ja, dass beide Funktionen in einem Korridor umeinander verlaufen, aber wenn der Quotient gegen 0 geht, dann wird die schneller wachsende Funktion jeden Korridor um die langsamer wachsende zwangsläufig verlassen.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$.
- Wir schreiben $f \in o(g)$, wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- Erste Beobachtung: $f \in o(g)$ und $f \in \Theta(g)$ schließen sich gegenseitig logisch aus.
- Zweite Beobachtung: Es gibt auch Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$, so dass weder $f \in o(g)$ noch $f \in \Theta(g)$ gilt.
- Einfaches Beispiel für die zweite Beobachtung:
  - $f : n \rightarrow | n \cdot \sin(n) |$
  - $g : n \rightarrow | n \cdot \cos(n) |$



Wir haben also jetzt eine Notation für gleich schnelles Wachstum von Funktionen und eine für echt unterschiedlich schnelles Wachstum. Mit diesen beiden Notationen kann man aber nicht alle Paare von Funktionen miteinander vergleichen, viele bleiben unvergleichbar, zum Beispiel die beiden Funktionen unten.

Grob gesprochen ist der Grund dafür, dass diese beiden Funktionen nicht vergleichbar sind, folgender: Beide Funktionen kommen der 0 unendlich häufig beliebig nahe, und zwar bei Werten von n, bei denen die jeweils andere Funktion immer weiter weg von 0 geht.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wir haben jetzt eine kleiner-Relation „o(·)“ definiert.
- Darauf aufbauend jetzt kleiner-gleich / größer-gleich.
- Seien wieder $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$.
- Wir schreiben:
  - $f \in O(g)$, wenn $f \in \Theta(g)$ oder $f \in o(g)$ ist;
  - $f \in \Omega(g)$, wenn $g \in \Theta(f)$ oder $g \in o(f)$ ist.
- Beobachtung: Offensichtlich gilt $f \in O(g)$ genau dann, wenn $g \in \Omega(f)$ gilt.

Für obere und untere Eingrenzung ist es sinnvoll, eine Relation kleiner-gleich statt kleiner zu haben, denn es kann ja durchaus sein, dass eine der beiden eingrenzenden Funktionen doch die Asymptotik genau beschreibt, ohne dass wir das durch mathematische Überlegungen oder Laufzeitstudien fundiert belegen können.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wir haben jetzt eine kleiner-Relation „o(·)“ definiert.
- Darauf aufbauend jetzt kleiner-gleich / größer-gleich.
- Seien wieder $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$.
- Wir schreiben:
  - $f \in O(g)$, wenn $f \in \Theta(g)$ oder $f \in o(g)$ ist;
  - $f \in \Omega(g)$, wenn $g \in \Theta(f)$ oder $g \in o(f)$ ist.
- Beobachtung: Offensichtlich gilt $f \in O(g)$ genau dann, wenn $g \in \Omega(f)$ gilt.

Die Notation, die sich für kleiner-gleich beziehungsweise größer-gleich in der Informatik eingebürgert hat, sieht etwas willkürlich aus, aber um unfallfrei mit anderen darüber zu reden, sollten wir uns daran halten.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wir haben jetzt eine kleiner-Relation „o(·)“ definiert.
- Darauf aufbauend jetzt kleiner-gleich / größer-gleich.
- Seien wieder $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$.
- Wir schreiben:
  - $f \in O(g)$, wenn $f \in \Theta(g)$ oder $f \in o(g)$ ist;
  - $f \in \Omega(g)$, wenn $g \in \Theta(f)$ oder $g \in o(f)$ ist.
- Beobachtung: Offensichtlich gilt $f \in O(g)$ genau dann, wenn $g \in \Omega(f)$ gilt.

Die beiden Funktionen sind in der Definition austauschbar, so dass die eine Funktion genau dann kleiner-gleich der anderen ist, wenn die andere Funktion größer-gleich der einen ist. So soll es bei einem Größenvergleich ja auch sein.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln für Theta, *O*, Ω und o (seien *f*, *g*, *h* : $\mathbb{N} \rightarrow \mathbb{R}^+$):

- **Θ induziert eine Äquivalenzrelation, das heißt, es gilt:**
  - **Reflexiv: $f \in \Theta(f)$.**
  - **Symmetrisch: Wenn $f \in \Theta(g)$, dann ist $g \in \Theta(f)$.**
  - **Transitiv: Wenn $f \in \Theta(g)$ und $g \in \Theta(h)$, dann auch $f \in \Theta(h)$.**
- ***O* induziert eine partielle Ordnung bzgl. Θ, das heißt, es gilt:**
  - **Reflexiv: $f \in O(f)$.**
  - **Antisymmetrisch: Wenn $f \in O(g)$ und $g \in O(f)$, dann ist $f \in \Theta(g)$.**
  - **Transitiv: Wenn $f \in O(g)$ und $g \in O(h)$, dann auch $f \in O(h)$.**
- **o induziert die strikte partielle Ordnung zu O.**
  - **Antireflexiv (d.h. $f \notin o(f)$), antisymmetrisch und transitiv.**

Ein paar einfache, aber nützliche mathematische Sachverhalte lassen sich leicht ableiten.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln für Theta, *O*, Ω und o (seien *f*, *g*, *h* : $\mathbb{N} \rightarrow \mathbb{R}^+$):

- **Θ induziert eine Äquivalenzrelation, das heißt, es gilt:**
  - **Reflexiv: $f \in \Theta(f)$.**
  - **Symmetrisch: Wenn $f \in \Theta(g)$, dann ist $g \in \Theta(f)$.**
  - **Transitiv: Wenn $f \in \Theta(g)$ und $g \in \Theta(h)$, dann auch $f \in \Theta(h)$.**
- ***O* induziert eine partielle Ordnung bzgl. Θ, das heißt, es gilt:**
  - **Reflexiv: $f \in O(f)$.**
  - **Antisymmetrisch: Wenn $f \in O(g)$ und $g \in O(f)$, dann ist $f \in \Theta(g)$.**
  - **Transitiv: Wenn $f \in O(g)$ und $g \in O(h)$, dann auch $f \in O(h)$.**
- **o induziert die strikte partielle Ordnung zu O.**
  - **Antireflexiv (d.h. $f \notin o(f)$), antisymmetrisch und transitiv.**

Als erstes können wir festhalten, dass die binäre Relation Theta tatsächlich die Axiome für Äquivalenzrelationen erfüllt.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln für Theta, *O*, Ω und o (seien *f*, *g*, *h* : $\mathbb{N} \rightarrow \mathbb{R}^+$):

- **Θ induziert eine Äquivalenzrelation, das heißt, es gilt:**
  - **Reflexiv: $f \in \Theta(f)$.**
  - **Symmetrisch: Wenn $f \in \Theta(g)$, dann ist $g \in \Theta(f)$.**
  - **Transitiv: Wenn $f \in \Theta(g)$ und $g \in \Theta(h)$, dann auch $f \in \Theta(h)$.**
- ***O* induziert eine partielle Ordnung bzgl. Θ, das heißt, es gilt:**
  - **Reflexiv: $f \in O(f)$.**
  - **Antisymmetrisch: Wenn $f \in O(g)$ und $g \in O(f)$, dann ist $f \in \Theta(g)$.**
  - **Transitiv: Wenn $f \in O(g)$ und $g \in O(h)$, dann auch $f \in O(h)$.**
- **o induziert die strikte partielle Ordnung zu O.**
  - **Antireflexiv (d.h. $f \notin o(f)$), antisymmetrisch und transitiv.**

Trivialerweise verläuft jede Funktion in einem Korridor um sich selbst.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln für Theta, *O*, Ω und o (seien *f*, *g*, *h* : ℕ → ℝ⁺):

- **Θ induziert eine Äquivalenzrelation, das heißt, es gilt:**
  - **Reflexiv: $f \in \Theta(f)$.**
  - **Symmetrisch: Wenn $f \in \Theta(g)$, dann ist $g \in \Theta(f)$.**
  - **Transitiv: Wenn $f \in \Theta(g)$ und $g \in \Theta(h)$, dann auch $f \in \Theta(h)$.**
- ***O* induziert eine partielle Ordnung bzgl. Θ, das heißt, es gilt:**
  - **Reflexiv: $f \in O(f)$.**
  - **Antisymmetrisch: Wenn $f \in O(g)$ und $g \in O(f)$, dann ist $f \in \Theta(g)$.**
  - **Transitiv: Wenn $f \in O(g)$ und $g \in O(h)$, dann auch $f \in O(h)$.**
- **o induziert die strikte partielle Ordnung zu O.**
  - **Antireflexiv (d.h. $f \notin o(f)$), antisymmetrisch und transitiv.**

Und wenn eine Funktion in einem Korridor um die andere verläuft, dann verläuft auch die andere Funktion in einem Korridor um die eine.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln für Theta, *O*, Ω und o (seien *f*, *g*, *h* : ℕ → ℝ⁺):

- **Θ induziert eine Äquivalenzrelation, das heißt, es gilt:**
  - **Reflexiv: $f \in \Theta(f)$.**
  - **Symmetrisch: Wenn $f \in \Theta(g)$, dann ist $g \in \Theta(f)$.**
  - **Transitiv: Wenn $f \in \Theta(g)$ und $g \in \Theta(h)$, dann auch $f \in \Theta(h)$.**
- ***O* induziert eine partielle Ordnung bzgl. Θ, das heißt, es gilt:**
  - **Reflexiv: $f \in O(f)$.**
  - **Antisymmetrisch: Wenn $f \in O(g)$ und $g \in O(f)$, dann ist $f \in \Theta(g)$.**
  - **Transitiv: Wenn $f \in O(g)$ und $g \in O(h)$, dann auch $f \in O(h)$.**
- **o induziert die strikte partielle Ordnung zu O.**
  - **Antireflexiv (d.h. $f \notin o(f)$), antisymmetrisch und transitiv.**

Wenn eine Funktion in einem Korridor um eine zweite Funktion verläuft und die zweite in einem Korridor um eine dritte, dann kann man die oberen Konstanten der beiden Korridore miteinander multiplizieren und ebenso die unteren Konstanten der beiden Korridore miteinander multiplizieren und erhält so einen Korridor um die dritte Funktion, in der die erste Funktion verläuft.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln für Theta, *O*, Ω und o (seien *f*, *g*, *h* : $\mathbb{N} \rightarrow \mathbb{R}^+$):

- **Θ induziert eine Äquivalenzrelation, das heißt, es gilt:**
  - **Reflexiv: $f \in \Theta(f)$.**
  - **Symmetrisch: Wenn $f \in \Theta(g)$, dann ist $g \in \Theta(f)$.**
  - **Transitiv: Wenn $f \in \Theta(g)$ und $g \in \Theta(h)$, dann auch $f \in \Theta(h)$.**
- ***O* induziert eine partielle Ordnung bzgl. Θ, das heißt, es gilt:**
  - **Reflexiv: $f \in O(f)$.**
  - **Antisymmetrisch: Wenn $f \in O(g)$ und $g \in O(f)$, dann ist $f \in \Theta(g)$.**
  - **Transitiv: Wenn $f \in O(g)$ und $g \in O(h)$, dann auch $f \in O(h)$.**
- **o induziert die strikte partielle Ordnung zu O.**
  - **Antireflexiv (d.h. $f \notin o(f)$), antisymmetrisch und transitiv.**

Die Relation für kleiner-gleich erfüllt hingegen die Axiome für partielle Ordnungen.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln für Theta, *O*, Ω und o (seien *f*, *g*, *h* : $\mathbb{N} \rightarrow \mathbb{R}^+$):

- **Θ induziert eine Äquivalenzrelation, das heißt, es gilt:**
  - **Reflexiv: $f \in \Theta(f)$.**
  - **Symmetrisch: Wenn $f \in \Theta(g)$, dann ist $g \in \Theta(f)$.**
  - **Transitiv: Wenn $f \in \Theta(g)$ und $g \in \Theta(h)$, dann auch $f \in \Theta(h)$.**
- ***O* induziert eine partielle Ordnung bzgl. Θ, das heißt, es gilt:**
  - **Reflexiv: $f \in O(f)$.**
  - **Antisymmetrisch: Wenn $f \in O(g)$ und $g \in O(f)$, dann ist $f \in \Theta(g)$.**
  - **Transitiv: Wenn $f \in O(g)$ und $g \in O(h)$, dann auch $f \in O(h)$.**
- **o induziert die strikte partielle Ordnung zu O.**
  - **Antireflexiv (d.h. $f \notin o(f)$), antisymmetrisch und transitiv.**

Da zwei Funktionen in Theta-Relation nach Definition auch in Groß-O-Relation sind, folgt aus der Reflexivität von Theta auch die Reflexivität von Groß-O.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln für Theta, *O*, Ω und o (seien *f*, *g*, *h* : $\mathbb{N} \rightarrow \mathbb{R}^+$):

- **Θ induziert eine Äquivalenzrelation, das heißt, es gilt:**
  - **Reflexiv: $f \in \Theta(f)$.**
  - **Symmetrisch: Wenn $f \in \Theta(g)$, dann ist $g \in \Theta(f)$.**
  - **Transitiv: Wenn $f \in \Theta(g)$ und $g \in \Theta(h)$, dann auch $f \in \Theta(h)$.**
- ***O* induziert eine partielle Ordnung bzgl. Θ, das heißt, es gilt:**
  - **Reflexiv: $f \in O(f)$.**
  - **Antisymmetrisch: Wenn $f \in O(g)$ und $g \in O(f)$, dann ist $f \in \Theta(g)$.**
  - **Transitiv: Wenn $f \in O(g)$ und $g \in O(h)$, dann auch $f \in O(h)$.**
- **o induziert die strikte partielle Ordnung zu O.**
  - **Antireflexiv (d.h. $f \notin o(f)$), antisymmetrisch und transitiv.**

Groß-O heißt ja, dass die beiden Funktionen entweder in Klein-o-Relation oder in Theta-Relation zueinander stehen. Es ist aber unmöglich, dass die eine Funktion in Klein-o-Relation zur anderen und die andere in Groß-O-Relation zur einen steht. Daher bleibt nur die Möglichkeit, dass die beiden Funktionen in Theta-Relation zueinander stehen.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln für Theta, *O*, Ω und o (seien *f*, *g*, *h* : $\mathbb{N} \rightarrow \mathbb{R}^+$):

- Θ induziert eine Äquivalenzrelation, das heißt, es gilt:
  - Reflexiv: $f \in \Theta(f)$.
  - Symmetrisch: Wenn $f \in \Theta(g)$, dann ist $g \in \Theta(f)$.
  - Transitiv: Wenn $f \in \Theta(g)$ und $g \in \Theta(h)$, dann auch $f \in \Theta(h)$.
- *O* induziert eine partielle Ordnung bzgl. Θ, das heißt, es gilt:
  - Reflexiv: $f \in O(f)$.
  - Antisymmetrisch: Wenn $f \in O(g)$ und $g \in O(f)$, dann ist $f \in \Theta(g)$.
  - Transitiv: Wenn $f \in O(g)$ und $g \in O(h)$, dann auch $f \in O(h)$.
- o induziert die strikte partielle Ordnung zu O.
  - Antireflexiv (d.h. $f \notin o(f)$), antisymmetrisch und transitiv.

Hier haben wir nach Definition von Groß-O vier verschiedene Fälle zu betrachten: Die erste Funktion kann in Klein-o-Relation oder in Theta-Relation zur zweiten Funktion stehen, und genauso kann die zweite Funktion in Klein-o-Relation oder in Theta-Relation zur dritten Funktion stehen.

Offensichtlich gilt: Wenn eine Funktion in Klein-o-Relation zu einer zweiten Funktion steht, dann steht sie auch in Klein-o-Relation zu jeder Funktion, die in Theta-Relation zur zweiten Funktion steht. Ebenso gilt offensichtlich: Wenn die erste Funktion in Klein-o-Relation zur zweiten steht und die zweite in Klein-o-Relation zur dritten, dann steht auch die erste in Klein-o-Relation zur dritten. Daher ist Groß-O in jedem dieser vier Fälle transitiv.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln für Theta, *O*, Ω und o (seien *f*, *g*, *h* : $\mathbb{N} \rightarrow \mathbb{R}^+$):

- **Θ induziert eine Äquivalenzrelation, das heißt, es gilt:**
  - **Reflexiv: $f \in \Theta(f)$.**
  - **Symmetrisch: Wenn $f \in \Theta(g)$, dann ist $g \in \Theta(f)$.**
  - **Transitiv: Wenn $f \in \Theta(g)$ und $g \in \Theta(h)$, dann auch $f \in \Theta(h)$.**
- ***O* induziert eine partielle Ordnung bzgl. Θ, das heißt, es gilt:**
  - **Reflexiv: $f \in O(f)$.**
  - **Antisymmetrisch: Wenn $f \in O(g)$ und $g \in O(f)$, dann ist $f \in \Theta(g)$.**
  - **Transitiv: Wenn $f \in O(g)$ und $g \in O(h)$, dann auch $f \in O(h)$.**
- **o induziert die strikte partielle Ordnung zu O.**
  - **Antireflexiv (d.h. $f \notin o(f)$), antisymmetrisch und transitiv.**

Der Unterschied zwischen Groß-o und Klein-o liegt genau darin, dass Groß-o reflexiv und Klein-o antireflexiv ist. Das ist in der Mathematik genau der Unterschied zwischen einer nichtstrikten und strikten partiellen Ordnung.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Fibonacci-Zahlen:

```
;; Type: natural -> natural
;; Returns: the n-th Fibonacci number
( define ( fib n )
    ( if ( or ( = n 0 ) ( = n 1 ) ) 1 ( + ( fib ( – n 1 ) ) ( fib ( – n 2 ) ) ) ) )
```

Asymptotische Analyse:

- **Die Laufzeit ist asymptotisch gleich der Anzahl der rekursiven Aufrufe.**
- **Die Anzahl der rekursiven Aufrufe ist**
  - **höchstens so hoch wie bei (+ (fib (– n 1)) (fib (– n 1))) → $O(2^n)$;**
  - **mindestens so hoch wie bei (+ (fib (– n 2)) (fib (– n 2))) → $\Omega(2^{n/2})$.**

Jetzt wird es langsam Zeit für ein Beispiel. Die rekursive Berechnung von Fibonacci-Zahlen ist dafür gut geeignet.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Fibonacci-Zahlen:

```
;; Type: natural -> natural
;; Returns: the n-th Fibonacci number
( define ( fib n )
   ( if ( or ( = n 0 ) ( = n 1 ) ) 1 ( + ( fib ( – n 1 ) ) ( fib ( – n 2 ) ) ) ) )
```

Asymptotische Analyse:

- **Die Laufzeit ist asymptotisch gleich der Anzahl der rekursiven Aufrufe.**
- **Die Anzahl der rekursiven Aufrufe ist**
  - **höchstens so hoch wie bei (+ (fib (– n 1)) (fib (– n 1))) → $O(2^n)$;**
  - **mindestens so hoch wie bei (+ (fib (– n 2)) (fib (– n 2))) → $\Omega(2^{n/2})$.**

Bekanntlich erfordert die Berechnung der Fibonacci-Zahl für jedes n größer 1 zwei rekursive Aufrufe, einen für n minus 1 und einen für n minus 2.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Fibonacci-Zahlen:

```
;; Type: natural -> natural
;; Returns: the n-th Fibonacci number
( define ( fib n )
    ( if ( or ( = n 0 ) ( = n 1 ) ) 1 ( + ( fib ( – n 1 ) ) ( fib ( – n 2 ) ) ) ) )
```

Asymptotische Analyse:

- **Die Laufzeit ist asymptotisch gleich der Anzahl der rekursiven Aufrufe.**
- **Die Anzahl der rekursiven Aufrufe ist**
  - **höchstens so hoch wie bei (+ (fib (– n 1)) (fib (– n 1))) → $O(2^n)$;**
  - **mindestens so hoch wie bei (+ (fib (– n 2)) (fib (– n 2))) → $\Omega(2^{n/2})$.**

Die Laufzeit innerhalb eines rekursiven Aufrufs ist Theta von 1.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Fibonacci-Zahlen:

```
;; Type: natural -> natural
;; Returns: the n-th Fibonacci number
( define ( fib n )
    ( if ( or ( = n 0 ) ( = n 1 ) ) 1 ( + ( fib ( – n 1 ) ) ( fib ( – n 2 ) ) ) ) )
```

Asymptotische Analyse:

- **Die Laufzeit ist asymptotisch gleich der Anzahl der rekursiven Aufrufe.**
- **Die Anzahl der rekursiven Aufrufe ist**
  - **höchstens so hoch wie bei (+ (fib (– n 1)) (fib (– n 1))) → $O(2^n)$;**
  - **mindestens so hoch wie bei (+ (fib (– n 2)) (fib (– n 2))) → $\Omega(2^{n/2})$.**

Um zu einer oberen Schranke zu kommen, können wir versuchsweise den Aufruf mit n minus 2 ersetzen durch einen Aufruf mit n minus 1. Dadurch kann die Gesamtzahl rekursiver Aufrufe nur größer, nicht kleiner werden, denn die Anzahl rekursiver Aufrufe für n ist sicherlich monoton wachsend in n.

Der entscheidende Punkt ist, dass man für diese Variante sehr leicht die Anzahl der rekursiven Aufrufe bestimmen kann: Für jede Verminderung des Wertes von n um 1 verdoppelt sich die Anzahl rekursiver Aufrufe. Das ergibt 1 plus 2 plus 4 und so weiter bis 2 hoch Exponent n minus 1. Aus der Mathematik wissen Sie, dass diese Summe ungefähr 2 hoch n ergibt, also erhalten wir insgesamt O von 2 hoch n.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Fibonacci-Zahlen:

```
;; Type: natural -> natural
;; Returns: the n-th Fibonacci number
( define ( fib n )
    ( if ( or ( = n 0 ) ( = n 1 ) ) 1 ( + ( fib ( – n 1 ) ) ( fib ( – n 2 ) ) ) ) )
```

Asymptotische Analyse:

- **Die Laufzeit ist asymptotisch gleich der Anzahl der rekursiven Aufrufe.**
- **Die Anzahl der rekursiven Aufrufe ist**
  - **höchstens so hoch wie bei (+ (fib (– n 1)) (fib (– n 1))) → $O(2^n)$;**
  - **mindestens so hoch wie bei (+ (fib (– n 2)) (fib (– n 2))) → $\Omega(2^{n/2})$.**

Wenn wir nun umgekehrt nicht n minus 2 durch n minus 1, sondern n minus 1 durch n minus 2 ersetzen, kann die Anzahl der rekursiven Aufrufe nur geringer werden, so dass sich eine Abschätzung nach unten ergibt. Nun verdoppelt sich die Anzahl der Aufrufe bei jeder Verminderung von n um 2 statt wie vorher bei jeder Verminderung von n um 1. Daher ist in der beschreibenden Funktion nun n durch n halbe zu ersetzen.

Diese beiden Funktionen – 2 hoch n und 2 hoch n halbe – gehen extrem schnell weit auseinander, so dass sie zusammen die wahre Asymptotik der Fibonacci-Berechnung nur sehr unscharf eingrenzen.

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Fibonacci-Zahlen:

```
;; Type: natural -> natural
;; Returns: the n-th Fibonacci number
( define ( fib n )
    ( if ( or ( = n 0 ) ( = n 1 ) ) 1 ( + ( fib ( – n 1 ) ) ( fib ( – n 2 ) ) ) ) )
```

Asymptotische Analyse:

- **Die Laufzeit ist asymptotisch gleich der Anzahl der rekursiven Aufrufe.**
- **Die Anzahl der rekursiven Aufrufe ist**
  - **höchstens so hoch wie bei (+ (fib (– n 1)) (fib (– n 1))) → $O(2^n)$;**
  - **mindestens so hoch wie bei (+ (fib (– n 2)) (fib (– n 2))) → $\Omega(2^{n/2})$.**

***Nebenbemerkung*: Die genaue Asymptotik der Anzahl rekursiver Aufrufe lässt sich über ein Konstrukt namens Fibonacci-Bäume bestimmen; die beschreibende mathematische Funktion ist allerdings wohl zu kompliziert, um für Laufzeitbetrachtungen praktikabel zu sein.**

Untere und obere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

	O	Ω	o	Θ
Worst Case	Obere Schranke im Worst Case	Untere Schranke im Worst Case	Strikte obere Schranke im Worst Case	Genaue Asymptotik im Worst Case
Best Case	Obere Schranke im Best Case	Untere Schranke im Best Case	Strikte obere Schranke im Best Case	Genaue Asymptotik im Best Case

Sehr häufig wird die Unterscheidung zwischen oberer und unterer Schranke mit der Unterscheidung zwischen Worst Case und Best Case verwechselt. Es ist aber wichtig, sich klarzumachen, dass das zwei völlig verschiedene Konzepte sind. Diese Tabelle demonstriert, dass beide Konzepte beliebig miteinander kombinierbar sind.

Jetzt auch System Time

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Returns: true iff n is a prime number, that is,
//              n > 1 and n only has two factors: 1 and n.
boolean isPrime ( BigInteger n ) {
    if ( n.compareTo(BigInteger.ONE) <= 0 )
        return false;
    if ( n.equals(BigInteger.TWO) )
        return true;
    for ( BigInteger i = BigInteger.valueOf(3); i.multiply(i).compareTo(n) < 0;
            i = i.add(BigInteger.TWO) )
        if ( n.remainder(i).equals(BigInteger.ZERO) )
            return false;
    return true;
}
```

Bisher hatten wir nur User Time betrachtet. Mit der System Time wird es schwieriger, denn die System Time ist nicht so leicht durchschaubar und daher auch nicht so leicht mathematisch analysierbar.

Wir schauen uns noch einmal den Algorithmus an, der bestimmt, ob eine gegebene Zahl n größer 1 eine Primzahl ist oder nicht. Aber jetzt verwenden wir die Klasse BigInteger anstelle des primitiven Datentyps int.

***Erinnerung*: Im Abschnitt zur Laufzeitverbesserung weiter vorne in diesem Kapitel hatten wir schon Klasse BigInteger gesehen.**

Jetzt auch System Time

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Returns: true iff n is a prime number, that is,
//              n > 1 and n only has two factors: 1 and n.
boolean isPrime ( BigInteger n ) {
    if ( n.compareTo(BigInteger.ONE) <= 0 )
        return false;
    if ( n.equals(BigInteger.TWO) )
        return true;
    for ( BigInteger i = BigInteger.valueOf(3); i.multiply(i).compareTo(n) < 0;
            i = i.add(BigInteger.TWO) )
        if ( n.remainder(i).equals(BigInteger.ZERO) )
            return false;
    return true;
}
```

Hier sehen wir eine Stelle, an der das Laufzeitsystem tatsächlich ernsthaft Laufzeit verbraucht, nämlich in der Ausführung des Operator new.

Jetzt auch System Time

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Returns: true iff n is a prime number, that is,
//               n > 1 and n only has two factors: 1 and n.
boolean isPrime ( BigInteger n ) {
    if ( n.compareTo(BigInteger.ONE) <= 0 )
        return false;
    if ( n.equals(BigInteger.TWO) )
        return true;
    for ( BigInteger i = BigInteger.valueOf(3); i.multiply(i).compareTo(n) < 0;
            i = i.add(BigInteger.TWO) )
          if ( n.remainder(i).equals(BigInteger.ZERO) )
              return false;
    return true;
}
```

Aber auch diese Methode von Klasse BigInteger liefert ein Objekt zurück, das natürlich erst erzeugt werden muss.

***Nebenbemerkung*: Es ist durchaus möglich, dass diese Erzeugung von Objekten innerhalb von Methoden der Standardbibliothek effizienter gestaltet ist als einfach nur durch Aufruf des Operators new für jedes zurückzuliefernde Objekt einzeln. Zum Beispiel könnte man sich vorstellen, dass intern beim ersten Aufruf gleich Platz für mehrere Objekte von BigInteger geschaffen wird, etwa in einem Array, und spätere Aufrufe bedienen sich dann aus diesem Reservoir, bis das aufgebraucht ist, und erst dann wird Operator new intern nochmals aufgerufen, wieder gleich für mehrere Objekte, und so weiter.**

Jetzt auch System Time

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit bei BigInteger:

- **Problemgröße: wieder der Wert des Parameters *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist wieder**
  - **im Worst Case $\Theta(\sqrt{n})$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf hängt ab von**
  - **der Laufzeit der Methoden von BigInteger,**
  - **der Laufzeit von Operator new.**
- **Daher kann nichts abschließendes zur Gesamtlaufzeit gesagt werden.**
  - **Empirische Laufzeitmessungen hinzunehmen.**
  - **Speicherverwaltung und BigInteger selbst implementieren, um die Gesamtlaufzeit unter Kontrolle zu halten.**

Wir kommen zur Analyse dieser Methode.

Jetzt auch System Time

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit bei BigInteger:

- **Problemgröße: wieder der Wert des Parameters *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist wieder**
  - **im Worst Case $\Theta(\sqrt{n})$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf hängt ab von**
  - **der Laufzeit der Methoden von BigInteger,**
  - **der Laufzeit von Operator new.**
- **Daher kann nichts abschließendes zur Gesamtlaufzeit gesagt werden.**
  - **Empirische Laufzeitmessungen hinzunehmen.**
  - **Speicherverwaltung und BigInteger selbst implementieren, um die Gesamtlaufzeit unter Kontrolle zu halten.**

Wie bei der Variante mit int ist auch in der zweiten Variante mit BigInteger der zu testende Wert die geeignete Kennzahl für die Problemgröße.

Jetzt auch System Time

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit bei BigInteger:

- **Problemgröße: wieder der Wert des Parameters *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist wieder**
  - **im Worst Case $\Theta(\sqrt{n})$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf hängt ab von**
  - **der Laufzeit der Methoden von BigInteger,**
  - **der Laufzeit von Operator new.**
- **Daher kann nichts abschließendes zur Gesamtlaufzeit gesagt werden.**
  - **Empirische Laufzeitmessungen hinzunehmen.**
  - **Speicherverwaltung und BigInteger selbst implementieren, um die Gesamtlaufzeit unter Kontrolle zu halten.**

Die Anzahl der Schleifendurchläufe im Worst Cast und im Best Case ist natürlich dieselbe, denn der Algorithmus ist derselbe, nur der Datentyp zur Repräsentation der Zahlen ist ein anderer.

Jetzt auch System Time

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit bei BigInteger:

- **Problemgröße: wieder der Wert des Parameters *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist wieder**
  - **im Worst Case $\Theta(\sqrt{n})$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf hängt ab von**
  - **der Laufzeit der Methoden von BigInteger,**
  - **der Laufzeit von Operator new.**
- **Daher kann nichts abschließendes zur Gesamtlaufzeit gesagt werden.**
  - **Empirische Laufzeitmessungen hinzunehmen.**
  - **Speicherverwaltung und BigInteger selbst implementieren, um die Gesamtlaufzeit unter Kontrolle zu halten.**

Allerdings enthält die Variante mit BigInteger einige Unbekannte, was die Laufzeit eines einzelnen Schleifendurchlaufs angeht.

Jetzt auch System Time

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit bei BigInteger:

- **Problemgröße: wieder der Wert des Parameters *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist wieder**
  - **im Worst Case $\Theta(\sqrt{n})$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf hängt ab von**
  - **der Laufzeit der Methoden von BigInteger,**
  - **der Laufzeit von Operator new.**
- **Daher kann nichts abschließendes zur Gesamtlaufzeit gesagt werden.**
  - **Empirische Laufzeitmessungen hinzunehmen.**
  - **Speicherverwaltung und BigInteger selbst implementieren, um die Gesamtlaufzeit unter Kontrolle zu halten.**

Hier stößt die mathematische Analyse definitiv an ihre Grenze.

Jetzt auch System Time

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit bei BigInteger:

- **Problemgröße: wieder der Wert des Parameters *n*.**
- **Die Anzahl der Durchläufe durch die Schleife ist wieder**
  - **im Worst Case $\Theta(\sqrt{n})$,**
  - **im Best Case 0.**
- **Die Laufzeit pro Schleifendurchlauf hängt ab von**
  - **der Laufzeit der Methoden von BigInteger,**
  - **der Laufzeit von Operator new.**
- **Daher kann nichts abschließendes zur Gesamtlaufzeit gesagt werden.**
  - **Empirische Laufzeitmessungen hinzunehmen.**
  - **Speicherverwaltung und BigInteger selbst implementieren, um die Gesamtlaufzeit unter Kontrolle zu halten.**

Es bleibt aber natürlich immer die Möglichkeit, die Asymptotik durch Laufzeitmessungen zu schätzen. Die Ergebnisse hängen aber natürlich stark davon ab, wie die Klasse BigInteger und der Operator new implementiert sind.

Jetzt auch System Time

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Analyse der Laufzeit bei BigInteger:

- Problemgröße: wieder der Wert des Parameters *n*.
- Die Anzahl der Durchläufe durch die Schleife ist wieder
  - im Worst Case $\Theta(\sqrt{n})$,
  - im Best Case 0.
- Die Laufzeit pro Schleifendurchlauf hängt ab von
  - der Laufzeit der Methoden von BigInteger,
  - der Laufzeit von Operator new.
- Daher kann nichts abschließendes zur Gesamtlaufzeit gesagt werden.
  - Empirische Laufzeitmessungen hinzunehmen.
  - Speicherverwaltung und BigInteger selbst implementieren, um die Gesamtlaufzeit unter Kontrolle zu halten.

Will man die Laufzeit unter Kontrolle halten, muss man die Funktionalität aus dem Laufzeitsystem und der Standardbibliothek re-implementieren.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double [ ] [ ]  product ( double [ ] [ ] m1, double [ ] [ ] m2 ) {
   double [ ] [ ] result = new double [ m1.length ] [ ];
   for ( int i = 0; i < result.length; i++ )
        result[i] = new double [ m2[0].length ];
   for ( int i = 0; i < result.length; i++ )
       for ( int j = 0; j < result[0].length; j++ ) {
           result[i][j] = 0   // redundant, may be omitted
           for ( int k = 0; k < m2.length; k++ )
              result[i][j] += m1[i][k] * m2[k][j];
       }
}
```

Bisher haben wir die Problemgröße in nur einer einzigen Kennzahl ausgedrückt. Das ist aber nicht immer adäquat, denn es kann durchaus sein, dass mehrere relevante Kennzahlen unabhängig voneinander variieren können. Dazu schauen wir uns ein wohlbekanntes Beispiel an: die Multiplikation zweier Matrizen.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double [ ] [ ]  product ( double [ ] [ ] m1, double [ ] [ ] m2 ) {
    double [ ] [ ] result = new double [ m1.length ] [ ];
    for ( int i = 0; i < result.length; i++ )
        result[i] = new double [ m2[0].length ];
    for ( int i = 0; i < result.length; i++ )
        for ( int j = 0; j < result[0].length; j++ ) {
            result[i][j] = 0   // redundant, may be omitted
            for ( int k = 0; k < m2.length; k++ )
                result[i][j] += m1[i][k] * m2[k][j];
        }
}
```

Die beiden zu multiplizierenden Matrizen sind die Parameter der Methode, die Ergebnismatrix der Rückgabewert. Da die Matrixmultiplikation nicht kommutativ ist, sollte besser im Vertrag explizit festgelegt sein, dass der erste Parameter zugleich auch der erste Faktor ist. Darüber hinaus erfordert die Matrixmultiplikation, dass die Spaltenzahl des ersten Faktors gleich der Zeilenzahl des zweiten Faktors ist, auch das gehört sicherheitshalber in den Vertrag.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double [ ] [ ]  product ( double [ ] [ ] m1, double [ ] [ ] m2 ) {
   double [ ] [ ] result = new double [ m1.length ] [ ];
   for ( int i = 0; i < result.length; i++ )
        result[i] = new double [ m2[0].length ];
   for ( int i = 0; i < result.length; i++ )
       for ( int j = 0; j < result[0].length; j++ ) {
           result[i][j] = 0   // redundant, may be omitted
           for ( int k = 0; k < m2.length; k++ )
              result[i][j] += m1[i][k] * m2[k][j];
       }
}
```

Hier wird die Ergebnismatrix schrittweise in der üblichen Weise erzeugt. Die Ergebnismatrix hat bekanntlich dieselbe Zeilenzahl wie der erste Faktor und dieselbe Spaltenzahl wie der zweite Faktor.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double [ ] [ ]  product ( double [ ] [ ] m1, double [ ] [ ] m2 ) {
   double [ ] [ ] result = new double [ m1.length ] [ ];
   for ( int i = 0; i < result.length; i++ )
        result[i] = new double [ m2[0].length ];
   for ( int i = 0; i < result.length; i++ )
      for ( int j = 0; j < result[0].length; j++ ) {
          result[i][j] = 0   // redundant, may be omitted
          for ( int k = 0; k < m2.length; k++ )
             result[i][j] += m1[i][k] * m2[k][j];
      }
}
```

Jetzt die Befüllung der einzelnen Matrixeinträge mit Werten. Diese beiden Schleifen gehen über alle Zeilen und alle Spalten der Ergebnismatrix, kommen also insgesamt bei jedem Matrixeintrag genau einmal vorbei.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double [ ] [ ]  product ( double [ ] [ ] m1, double [ ] [ ] m2 ) {
   double [ ] [ ] result = new double [ m1.length ] [ ];
   for ( int i = 0; i < result.length; i++ )
        result[i] = new double [ m2[0].length ];
   for ( int i = 0; i < result.length; i++ )
       for ( int j = 0; j < result[0].length; j++ ) {
           result[i][j] = 0   // redundant, may be omitted
           for ( int k = 0; k < m2.length; k++ )
              result[i][j] += m1[i][k] * m2[k][j];
       }
}
```

Der Rumpf der inneren Schleife ist einfach die Umsetzung der definitorischen Formel der Matrixmultiplikation. Da die Komponenten eines Arrays mit seinem Null-Wert initialisiert werden, müssen wir sie hier nicht explizit mit 0 initialisieren.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Analyse User Time:

- Seien *m* und *l* die Zeilen- und Spaltenzahl von m1.
- Seien *l* und *n* die Zeilen- und Spaltenzahl vom m2.
- Anzahl Durchläufe innerste Schleife: $m \cdot l \cdot n$.
- Laufzeit pro Durchlauf hängt nicht von diesen drei Kennzahlen ab.
- Also asymptotische Komplexität: $\Theta(m \cdot l \cdot n)$.
  - Im Worst Case und Best Case.

Bei der Analyse beschränken wir uns hier auf die User Time. Das reicht, um den Punkt zu erläutern, um den es hier geht – mehrere Kennzahlen – und vermeidet Komplikationen, die hier irrelevant sind.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Analyse User Time:

- **Seien *m* und *l* die Zeilen- und Spaltenzahl von m1.**
- **Seien *l* und *n* die Zeilen- und Spaltenzahl vom m2.**
- **Anzahl Durchläufe innerste Schleife: $m \cdot l \cdot n$.**
- **Laufzeit pro Durchlauf hängt nicht von diesen drei Kennzahlen ab.**
- **Also asymptotische Komplexität: $\Theta(m \cdot l \cdot n)$.**
  - **Im Worst Case und Best Case.**

Da die Spaltenzahl der ersten Matrix gleich der Zeilenzahl der zweiten Matrix sein muss, gibt es nur *drei* Kennzahlen, nicht *vier*.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Analyse User Time:

- **Seien *m* und *l* die Zeilen- und Spaltenzahl von m1.**
- **Seien *l* und *n* die Zeilen- und Spaltenzahl vom m2.**
- **Anzahl Durchläufe innerste Schleife: $m \cdot l \cdot n$.**
- **Laufzeit pro Durchlauf hängt nicht von diesen drei Kennzahlen ab.**
- **Also asymptotische Komplexität: $\Theta(m \cdot l \cdot n)$.**
  - **Im Worst Case und Best Case.**

Wenn wir nur die User Time betrachten, dann ist die Befüllung der Matrixeinträge mit Werten der Teil, der die asymptotische Komplexität dominiert.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Analyse User Time:

- **Seien *m* und *l* die Zeilen- und Spaltenzahl von m1.**
- **Seien *l* und *n* die Zeilen- und Spaltenzahl vom m2.**
- **Anzahl Durchläufe innerste Schleife: $m \cdot l \cdot n$.**
- **Laufzeit pro Durchlauf hängt nicht von diesen drei Kennzahlen ab.**
- **Also asymptotische Komplexität: $\Theta(m \cdot l \cdot n)$.**
  - **Im Worst Case und Best Case.**

Die Laufzeit für den einzelnen Schleifendurchlauf hängt nicht von den Dimensionen der beiden Matrizen ab.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Analyse User Time:

- **Seien *m* und *l* die Zeilen- und Spaltenzahl von m1.**
- **Seien *l* und *n* die Zeilen- und Spaltenzahl vom m2.**
- **Anzahl Durchläufe innerste Schleife: $m \cdot l \cdot n$.**
- **Laufzeit pro Durchlauf hängt nicht von diesen drei Kennzahlen ab.**
- **Also asymptotische Komplexität: $\Theta(m \cdot l \cdot n)$.**
  - **Im Worst Case und Best Case.**

Daher ergibt sich diese asymptotische Komplexität in drei Kennzahlen.

Mehrere Kennzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Analyse User Time:

- **Seien *m* und *l* die Zeilen- und Spaltenzahl von m1.**
- **Seien *l* und *n* die Zeilen- und Spaltenzahl vom m2.**
- **Anzahl Durchläufe innerste Schleife: $m \cdot l \cdot n$.**
- **Laufzeit pro Durchlauf hängt nicht von diesen drei Kennzahlen ab.**
- **Also asymptotische Komplexität: $\Theta(m \cdot l \cdot n)$.**
  - ➢ **Im Worst Case und Best Case.**

Der Worst Case und der Best Case gehen bei diesem Algorithmus offensichtlich nicht auseinander.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ausgangssituation:
  - Worst Case und Best Case gehen ernsthaft auseinander.
  - Der Algorithmus wird sehr viele Male aufgerufen.
- Ziel: durchschnittliche Laufzeit durch eine mathematische Funktion beschreiben.
  - Average Case
- Methodisches Problem: Average Case basiert darauf, wie wahrscheinlich die einzelnen möglichen Eingaben sind.
  - Oft nicht einmal ungefähr bestimmbar.
  - Daher Average Case nur selten theoretisch betrachtet.
  - Laufzeitstudien auf den realen Daten gehen natürlich problemlos.

Wir schauen uns noch einen dritten Fall neben dem Best Case und dem Worst Case an.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ausgangssituation:
  - Worst Case und Best Case gehen ernsthaft auseinander.
  - Der Algorithmus wird sehr viele Male aufgerufen.
- Ziel: durchschnittliche Laufzeit durch eine mathematische Funktion beschreiben.
  - Average Case
- Methodisches Problem: Average Case basiert darauf, wie wahrscheinlich die einzelnen möglichen Eingaben sind.
  - Oft nicht einmal ungefähr bestimmbar.
  - Daher Average Case nur selten theoretisch betrachtet.
  - Laufzeitstudien auf den realen Daten gehen natürlich problemlos.

Sehr häufig ist die Situation die, dass ein Algorithmus eine große Zahl von Malen mit verschiedenen Eingaben aufgerufen wird. Man kann sich natürlich den Worst Case und den Best Case für die Gesamtzahl der Aufrufe überlegen. Aber wenn der Worst Case und der Best Case weit auseinandergehen, ist die wahre Gesamtlaufzeit über alle Aufrufe natürlich irgendwo dazwischen, wahrscheinlich weit weg von beiden.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ausgangssituation:
  - Worst Case und Best Case gehen ernsthaft auseinander.
  - Der Algorithmus wird sehr viele Male aufgerufen.
- Ziel: durchschnittliche Laufzeit durch eine mathematische Funktion beschreiben.
  - Average Case
- Methodisches Problem: Average Case basiert darauf, wie wahrscheinlich die einzelnen möglichen Eingaben sind.
  - Oft nicht einmal ungefähr bestimmbar.
  - Daher Average Case nur selten theoretisch betrachtet.
  - Laufzeitstudien auf den realen Daten gehen natürlich problemlos.

Man müsste sich also eher die *durchschnittliche* Laufzeit anschauen. Diese multipliziert mit der Anzahl Aufrufe ergibt dann eine realistischere Einschätzung der Gesamtlaufzeit.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ausgangssituation:
  - Worst Case und Best Case gehen ernsthaft auseinander.
  - Der Algorithmus wird sehr viele Male aufgerufen.
- Ziel: durchschnittliche Laufzeit durch eine mathematische Funktion beschreiben.
  - Average Case
- Methodisches Problem: Average Case basiert darauf, wie wahrscheinlich die einzelnen möglichen Eingaben sind.
  - Oft nicht einmal ungefähr bestimmbar.
  - Daher Average Case nur selten theoretisch betrachtet.
  - Laufzeitstudien auf den realen Daten gehen natürlich problemlos.

Der Fachbegriff wird wieder in der Regel nicht aus dem Englischen übersetzt.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ausgangssituation:
  - Worst Case und Best Case gehen ernsthaft auseinander.
  - Der Algorithmus wird sehr viele Male aufgerufen.
- Ziel: durchschnittliche Laufzeit durch eine mathematische Funktion beschreiben.
  - Average Case
- Methodisches Problem: Average Case basiert darauf, wie wahrscheinlich die einzelnen möglichen Eingaben sind.
  - Oft nicht einmal ungefähr bestimmbar.
  - Daher Average Case nur selten theoretisch betrachtet.
  - Laufzeitstudien auf den realen Daten gehen natürlich problemlos.

So ganz einfach ist die Sache allerdings nicht. Wenn wir hier von Durchschnitt reden, meinen wir eigentlich den Erwartungswert, das ist eine Kenngröße für Wahrscheinlichkeitsverteilungen. Für mathematische Überlegungen muss man also die Wahrscheinlichkeitsverteilung auf den Eingaben hernehmen, also für jede *mögliche* Eingabe die Wahrscheinlichkeit, dass diese die *tatsächliche* Eingabe ist.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ausgangssituation:
  - Worst Case und Best Case gehen ernsthaft auseinander.
  - Der Algorithmus wird sehr viele Male aufgerufen.
- Ziel: durchschnittliche Laufzeit durch eine mathematische Funktion beschreiben.
  - Average Case
- Methodisches Problem: Average Case basiert darauf, wie wahrscheinlich die einzelnen möglichen Eingaben sind.
  - Oft nicht einmal ungefähr bestimmbar.
  - Daher Average Case nur selten theoretisch betrachtet.
  - Laufzeitstudien auf den realen Daten gehen natürlich problemlos.

Leider gestaltet sich das häufig sehr schwierig, weil man nicht die dafür notwendigen Informationen über die realen Daten aus dem Anwendungsfall hat.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ausgangssituation:
  - Worst Case und Best Case gehen ernsthaft auseinander.
  - Der Algorithmus wird sehr viele Male aufgerufen.
- Ziel: durchschnittliche Laufzeit durch eine mathematische Funktion beschreiben.
  - Average Case
- Methodisches Problem: Average Case basiert darauf, wie wahrscheinlich die einzelnen möglichen Eingaben sind.
  - Oft nicht einmal ungefähr bestimmbar.
  - Daher Average Case nur selten theoretisch betrachtet.
  - Laufzeitstudien auf den realen Daten gehen natürlich problemlos.

Damit fehlt natürlich die Basis für die mathematische Analyse im Average Case.

Hinzu kommt, dass der Average Case erfahrungsgemäß weitaus komplexer mathematisch zu behandeln ist als der Worst Case und der Best Case.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ausgangssituation:
  - Worst Case und Best Case gehen ernsthaft auseinander.
  - Der Algorithmus wird sehr viele Male aufgerufen.
- Ziel: durchschnittliche Laufzeit durch eine mathematische Funktion beschreiben.
  - Average Case
- Methodisches Problem: Average Case basiert darauf, wie wahrscheinlich die einzelnen möglichen Eingaben sind.
  - Oft nicht einmal ungefähr bestimmbar.
  - Daher Average Case nur selten theoretisch betrachtet.
  - Laufzeitstudien auf den realen Daten gehen natürlich problemlos.

Mit Laufzeitstudien af den realen Daten kann man dem Average Case natürlich empirisch beikommen, auch wenn man die Wahrscheinlichkeitsverteilung nicht mathematisch bestimmen kann.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Primzahltest:

1. **beispielhafte Verteilung: Alle Zahlen 2 ... *N* sind gleich wahrscheinlich.**
   - **$\Omega(\sqrt{n}/\log_e n)$ und *O*($\sqrt{n}$) im Average Case**
2. **beispielhafte Verteilung: Primzahlen und Nichtprimzahlen sind gleich wahrscheinlich.**
   - **$\Theta(\sqrt{n})$ im Average Case**
3. **beispielhafte Verteilung: nur gerade Zahlen.**
   - **Θ(1) im Average Case**

Wir schauen uns zum Average Case zwei Beispiele an, die wir schon kennen, zuerst Primzahltest.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Primzahltest:

1. **beispielhafte Verteilung: Alle Zahlen 2 ... *N* sind gleich wahrscheinlich.**
   - **$\Omega(\sqrt{n}/\log_e n)$ und *O*($\sqrt{n}$) im Average Case**
2. **beispielhafte Verteilung: Primzahlen und Nichtprimzahlen sind gleich wahrscheinlich.**
   - **$\Theta(\sqrt{n})$ im Average Case**
3. **beispielhafte Verteilung: nur gerade Zahlen.**
   - **$\Theta(1)$ im Average Case**

Eine naheliegende Verteilung ist, dass jede natürliche Zahl, die den Vertrag erfüllt, also größer als 1 ist, gleich wahrscheinlich ist. Da es unendlich viele solcher Zahlen gibt und unendlich viele Zahlen nicht alle dieselbe positive Wahrscheinlichkeit haben können, muss man sich genauer gesagt eine sehr große natürliche Zahl *N* vorgeben und alle natürlichen Zahlen von 2 bis *N* als gleich wahrscheinlich annehmen.

Für diesen Fall dürfte es extrem schwierig sein, die asymptotische Komplexität im Average Case mit mathematischen Überlegungen exakt zu bestimmen.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Primzahltest:

1. **beispielhafte Verteilung: Alle Zahlen 2 ... *N* sind gleich wahrscheinlich.**
   - **$\Omega(\sqrt{n}/\log_e n)$ und *O*($\sqrt{n}$) im Average Case**
2. **beispielhafte Verteilung: Primzahlen und Nichtprimzahlen sind gleich wahrscheinlich.**
   - **$\Theta(\sqrt{n})$ im Average Case**
3. **beispielhafte Verteilung: nur gerade Zahlen.**
   - **$\Theta(1)$ im Average Case**

Wir können aber zumindest eine Abschätzung nach unten und nach oben vornehmen. Diese beiden Abschätzungen grenzen die wahre Asymptotik schon sehr eng ein, denn der Logarithmus ist ja eine sehr langsam wachsende Funktion und ändert daher im Nenner nicht allzu viel.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Primzahltest:

1. **beispielhafte Verteilung: Alle Zahlen 2 ... *N* sind gleich wahrscheinlich.**
   - **$\Omega(\sqrt{n}/\log_e n)$ und *O*($\sqrt{n}$) im Average Case**
2. **beispielhafte Verteilung: Primzahlen und Nichtprimzahlen sind gleich wahrscheinlich.**
   - **$\Theta(\sqrt{n})$ im Average Case**
3. **beispielhafte Verteilung: nur gerade Zahlen.**
   - **$\Theta(1)$ im Average Case**

Der Worst Case ist natürlich immer eine obere Schranke für den Average Case.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Primzahltest:

1. **beispielhafte Verteilung: Alle Zahlen 2 ... *N* sind gleich wahrscheinlich.**
   - **$\Omega(\sqrt{n}/\log_e n)$ und $O(\sqrt{n})$ im Average Case**
2. **beispielhafte Verteilung: Primzahlen und Nichtprimzahlen sind gleich wahrscheinlich.**
   - **$\Theta(\sqrt{n})$ im Average Case**
3. **beispielhafte Verteilung: nur gerade Zahlen.**
   - **$\Theta(1)$ im Average Case**

Nach dem Primzahlsatz ist der Anteil der Primzahlen an allen Zahlen bis groß *N* asymptotisch gleich dem Kehrwert des natürlichen Logarithmus von *N*. Primzahlen sind aber gerade der Worst Case, also Wurzel n. Egal wie groß die asymptotische Komplexität bei Nichtprimzahlen ist, kann der Erwartungswert nicht kleiner als Wurzel n mal der Anteil der Primzahlen an allen Zahlen sein.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Primzahltest:

1. **beispielhafte Verteilung: Alle Zahlen 2 ... *N* sind gleich wahrscheinlich.**
   - **$\Omega(\sqrt{n}/\log_e n)$ und *O*($\sqrt{n}$) im Average Case**
2. **beispielhafte Verteilung: Primzahlen und Nichtprimzahlen sind gleich wahrscheinlich.**
   - **$\Theta(\sqrt{n})$ im Average Case**
3. **beispielhafte Verteilung: nur gerade Zahlen.**
   - **$\Theta(1)$ im Average Case**

Im zweiten Fall ist die Bestimmung im Average Case hingegen sehr einfach.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Primzahltest:

1. **beispielhafte Verteilung: Alle Zahlen 2 ... *N* sind gleich wahrscheinlich.**
   - **$\Omega(\sqrt{n}/\log_e n)$ und $O(\sqrt{n})$ im Average Case**
2. **beispielhafte Verteilung: Primzahlen und Nichtprimzahlen sind gleich wahrscheinlich.**
   - **$\Theta(\sqrt{n})$ im Average Case**
3. **beispielhafte Verteilung: nur gerade Zahlen.**
   - **$\Theta(1)$ im Average Case**

Bei jeder Primzahl muss die Schleife bis zur Quadratwurzel von n ganz durchlaufen werden. Da also in jedem zweiten Fall der Worst Case eintritt, kann die Komplexität im Average nicht kleiner als die im Worst Case sein.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Primzahltest:

1. **beispielhafte Verteilung: Alle Zahlen 2 ... *N* sind gleich wahrscheinlich.**
   - **$\Omega(\sqrt{n}/\log_e n)$ und *O*($\sqrt{n}$) im Average Case**
2. **beispielhafte Verteilung: Primzahlen und Nichtprimzahlen sind gleich wahrscheinlich.**
   - **$\Theta(\sqrt{n})$ im Average Case**
3. **beispielhafte Verteilung: nur gerade Zahlen.**
   - **$\Theta(1)$ im Average Case**

Der konkrete Anwendungsfall könnte aber auch beispielsweise so aussehen, dass überhaupt nur gerade Zahlen hereinkommen.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Primzahltest:

1. **beispielhafte Verteilung: Alle Zahlen 2 ... *N* sind gleich wahrscheinlich.**
   - **$\Omega(\sqrt{n}/\log_e n)$ und $O(\sqrt{n})$ im Average Case**
2. **beispielhafte Verteilung: Primzahlen und Nichtprimzahlen sind gleich wahrscheinlich.**
   - **$\Theta(\sqrt{n})$ im Average Case**
3. **beispielhafte Verteilung: nur gerade Zahlen.**
   - **$\Theta(1)$ im Average Case**

Bei dieser Verteilung kommt der Algorithmus kein einziges Mal überhaupt in die Schleife.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel lineare Suche:

1. **beispielhafte Verteilung: Alle Suchwerte 0 ... Integer.MAX_VALUE sind gleich wahrscheinlich.**
   - **Θ(*n*) im Average Case bei „normalen“ Werten im Array**
2. **beispielhafte Verteilung: nur die Zahlen, die im Array sind (alle gleich wahrscheinlich).**
   - **Θ(*n*) im Average Case**
3. **beispielhafte Verteilung: nur Zahlen, die nicht größer als der kleinste Wert im Array sind.**
   - **Θ(1) im Average Case**

Nun das zweite Beispiel: lineare Suche in einem aufsteigend sortierten Array.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel lineare Suche:

1. **beispielhafte Verteilung: Alle Suchwerte 0 ... Integer.MAX_VALUE sind gleich wahrscheinlich.**
   - **Θ(*n*) im Average Case bei „normalen“ Werten im Array**
2. **beispielhafte Verteilung: nur die Zahlen, die im Array sind (alle gleich wahrscheinlich).**
   - **Θ(*n*) im Average Case**
3. **beispielhafte Verteilung: nur Zahlen, die nicht größer als der kleinste Wert im Array sind.**
   - **Θ(1) im Average Case**

Als erstes betrachten wir den Fall, dass der Datentyp int ist und alle nichtnegativen int-Werte gleich wahrscheinlich sind.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel lineare Suche:

1. **beispielhafte Verteilung: Alle Suchwerte 0 ... Integer.MAX_VALUE sind gleich wahrscheinlich.**
   - **Θ(*n*) im Average Case bei „normalen“ Werten im Array**
2. **beispielhafte Verteilung: nur die Zahlen, die im Array sind (alle gleich wahrscheinlich).**
   - **Θ(*n*) im Average Case**
3. **beispielhafte Verteilung: nur Zahlen, die nicht größer als der kleinste Wert im Array sind.**
   - **Θ(1) im Average Case**

Falls eher normal kleine Werte im Array sind, wird ein sehr hoher Anteil der Suchen dazu führen, dass das gesamte Array durchsucht werden muss. Damit kann dann die Komplexität im Average Case nicht kleiner als im Worst Case sein.

***Nebenbemerkung*: Falls man lineare Suche in einem solchen Szenario anwenden will, bietet es sich an, das Array absteigend zu durchsuchen, weil dann jeder Suchwert, der größer als das größte Arrayelement ist, zum Abbruch im ersten Schleifendurchlauf führt.**

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel lineare Suche:

1. **beispielhafte Verteilung: Alle Suchwerte 0 ... Integer.MAX_VALUE sind gleich wahrscheinlich.**
   - **Θ(*n*) im Average Case bei „normalen“ Werten im Array**
2. **beispielhafte Verteilung: nur die Zahlen, die im Array sind (alle gleich wahrscheinlich).**
   - **Θ(*n*) im Average Case**
3. **beispielhafte Verteilung: nur Zahlen, die nicht größer als der kleinste Wert im Array sind.**
   - **Θ(1) im Average Case**

Man könnte denken, die Komplexität im Average Case ist besser, wenn nur Werte gesucht werden, die auch im Array enthalten sind.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel lineare Suche:

1. **beispielhafte Verteilung: Alle Suchwerte 0 ... Integer.MAX_VALUE sind gleich wahrscheinlich.**
   - **Θ(*n*) im Average Case bei „normalen“ Werten im Array**
2. **beispielhafte Verteilung: nur die Zahlen, die im Array sind (alle gleich wahrscheinlich).**
   - **Θ(*n*) im Average Case**
3. **beispielhafte Verteilung: nur Zahlen, die nicht größer als der kleinste Wert im Array sind.**
   - **Θ(1) im Average Case**

Aber so ist es natürlich nicht. Wenn alle Werte im Array gleich wahrscheinlich sind, dann wird im Durchschnitt die Hälfte des Arrays durchsucht, bis der gesuchte Wert gefunden ist. Das ist wieder lineare Komplexität.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel lineare Suche:

1. **beispielhafte Verteilung: Alle Suchwerte 0 ... Integer.MAX_VALUE sind gleich wahrscheinlich.**
   - **Θ(*n*) im Average Case bei „normalen“ Werten im Array**
2. **beispielhafte Verteilung: nur die Zahlen, die im Array sind (alle gleich wahrscheinlich).**
   - **Θ(*n*) im Average Case**
3. **beispielhafte Verteilung: nur Zahlen, die nicht größer als der kleinste Wert im Array sind.**
   - **Θ(1) im Average Case**

Ganz anders sieht es hingegen aus, wenn die gesuchten Werte eher klein sind im Vergleich zu den Werten im Array.

Average Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel lineare Suche:

1. **beispielhafte Verteilung: Alle Suchwerte 0 ... Integer.MAX_VALUE sind gleich wahrscheinlich.**
   - **Θ(*n*) im Average Case bei „normalen“ Werten im Array**
2. **beispielhafte Verteilung: nur die Zahlen, die im Array sind (alle gleich wahrscheinlich).**
   - **Θ(*n*) im Average Case**
3. **beispielhafte Verteilung: nur Zahlen, die nicht größer als der kleinste Wert im Array sind.**
   - **Θ(1) im Average Case**

Denn dann ist die Suche immer schon schnell zu Ende.

Damit ist der Abschnitt zur asymptotischen Komplexität und das gesamte Kapitel zu effizienter Software beendet.