TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Effizienz von Algorithmen und Datenstrukturen

Effizienz von Algorithmen und Datenstrukturen

Dieser Abschnitt behandelt Algorithmen und Datenstrukturen und deren Analyse bzgl.

- Speicherbedarf
- Laufzeit

Er liefert einen Einblick in die Speicherverwaltung und präsentiert klassische algorithmische Problemstellungen und Methoden.

Literatur und Quelle: *Sedgewick, Wayne: Einführung in die Programmierung mit Java, Kap. 4.1.*

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Einführung in die Algorithmenanalyse

Charakteristiken von Programmen

Wie lange wird das Programm brauchen?
Reicht mein Speicherplatz?

- Die Antwort ist von vielen Faktoren abhängig und sehr schwierig präzise zu beantworten.
- In der Regel hängt die Laufzeit und der Speicherbedarf von der konkreten Eingabedaten ab.
- Idee: Ermittle ein Modell unter Zuhilfenahme von Messungen, um Laufzeit (bzw. Speicherbedarf) in Abhängigkeit der Eingabe abzuschätzen!
- Messungen sind nötig, um das Modell zu validieren und wichtige Kenngrößen zu ermitteln.
- Wir betrachten hier zunächst die Laufzeit von Programmen.

Experimentelle Vorgehensweise

Um ein Modell zu ermitteln und zu validieren, gehen wir zunächst experimentell vor:

1. *Beobachte* bestimmte Eigenschaften des Programms (hier: quantitativ durch Messungen von Laufzeit).
2. Erstelle ein *Hypothese*, die mit den Beobachtungen übereinstimmt.
3. *Prognostiziere* mit Hilfe dieser Hypothese Ereignisse.
4. *Validiere* die Vorhersagen durch weitere Beobachtungen. Falls die Vorhersagen nicht beobachtet werden können, verwerfe die Hypothese und erstelle eine neue, angepasste Hypothese.

Wichtig:

- Die Experimente zur Beobachtung müssen reproduzierbar sein.
- Hypothesen müssen falsifizierbar sein (d.h. müssen prinzipiell durch Experimente widerlegbar sein).

Beispiel: Sortieren durch Auswahl auf Arrays

- Eingabe: Array f der Größe N mit Integern
- Algorithmische Idee:
  1. Finde einen Index *imin* des Feldes, so dass $f[imin]$ ein minimales Element von $f[0]$ bis $f[N-1]$ enthält
  2. Vertausche $f[imin]$ und $f[0]$
  3. Sortiere dann den Bereich $f[1]$ bis $f[N-1]$

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Programm: Sortieren durch Auswahl auf Arrays I

```
public static void main (String[] args) {
  // Generieren des Arrays
  int n = args.length;
  int[] f = new int[n];
  for (int i = 0; i < n; i++) {
    f[i] = Integer.parseInt(args[i]);
  }

  // Sortieren
  sort(f);

  // Ausgabe
  for (int i = 0; i < n; i++) {
    StdOut.print(f[i] + " ");
  }
}
```

Programm: Sortieren durch Auswahl auf Arrays II

```
static void sort (int[] f) {  // Annahme: f != null
  for (int i = 0; i < f.length - 1; i++) {  // Schleife A

    // Finde Index fuer minimales Element
    int imin = i;
    for (int j = i+1; j < f.length; j++) {  // Schleife B
      if (f[j] < f[imin]) {
        imin = j;
      }
    }

    // Vertausche Elemente
    int temp = f[i];
    f[i] = f[imin];
    f[imin] = temp;
  }
}
```

Schritt 1: Beobachtungen

- Zunächst messen wir die Laufzeit des Programms für verschiedene Eingaben.
- Messung der Laufzeit ist auf verschiedene Arten möglich.

Die Stopwatch-Klasse

```
// Misst Zeiten (in ms) vom Erzeugen einer Stopwatch
// bis zum Aufruf von elapasedTime()
public class Stopwatch {
  private long start; // Startzeit der Messung

  // Startet die Messung
  public Stopwatch() {
    start = System.currentTimeMillis();
  }

  // Liefert die Zeit in ms seit Beginn der Messung
  public long elapsedTime() {
  long now = System.currentTimeMillis();
  return now - start;
  }
}
```

Ermitteln von Laufzeiten mittels Stopwatch

```
public static void main (String[] args) {
  // Generieren des Arrays
  int n = args.length;
  int[] f = new int[n];
  for (int i = 0; i < n; i++) {
    f[i] = Integer.parseInt(args[i]);
  }

  Stopwatch s = new Stopwatch();

  // Sortieren
  sort(f);

  // Ermitteln der Laufzeit
  StdOut.println("Laufzeit: " + s.elapsedTime());

  // Ausgabe
  for (int i = 0; i < n; i++) {
    StdOut.print(f[i] + " ");
  }
}
```

Beobachtungen aus verschiedenen Messungen

- Laufzeit ist abhängig von der Größe N des zu sortierenden Arrays (insbesondere für große N)
- Laufzeit variiert leicht für *gleiche* Eingabe bei verschiedenen Durchläufen
- Keine offensichtliche Abhängigkeit von den zu sortierenden Zahlen
- Insbesondere scheint der Algorithmus insensitive in Bezug auf Vorsortierung
- Laufzeit ändert sich, wenn andere Hardware / Software verwendet wird, Trend bleibt aber gleich

Problemgrößen

- Typischerweise beeinflusst eine (bisweilen auch mehrere) **Problemgröße** die Komplexität eines Programms.
- Die Laufzeit sollte mit der wachsender Problemgröße zunehmen.
- Typischerweise ist dies die Größe der Eingabe oder auch der Wert von Befehlszeilenargumenten.
- Im Beispiel: Größe N des zu sortierenden Arrays

Schritt 2: Hypothese

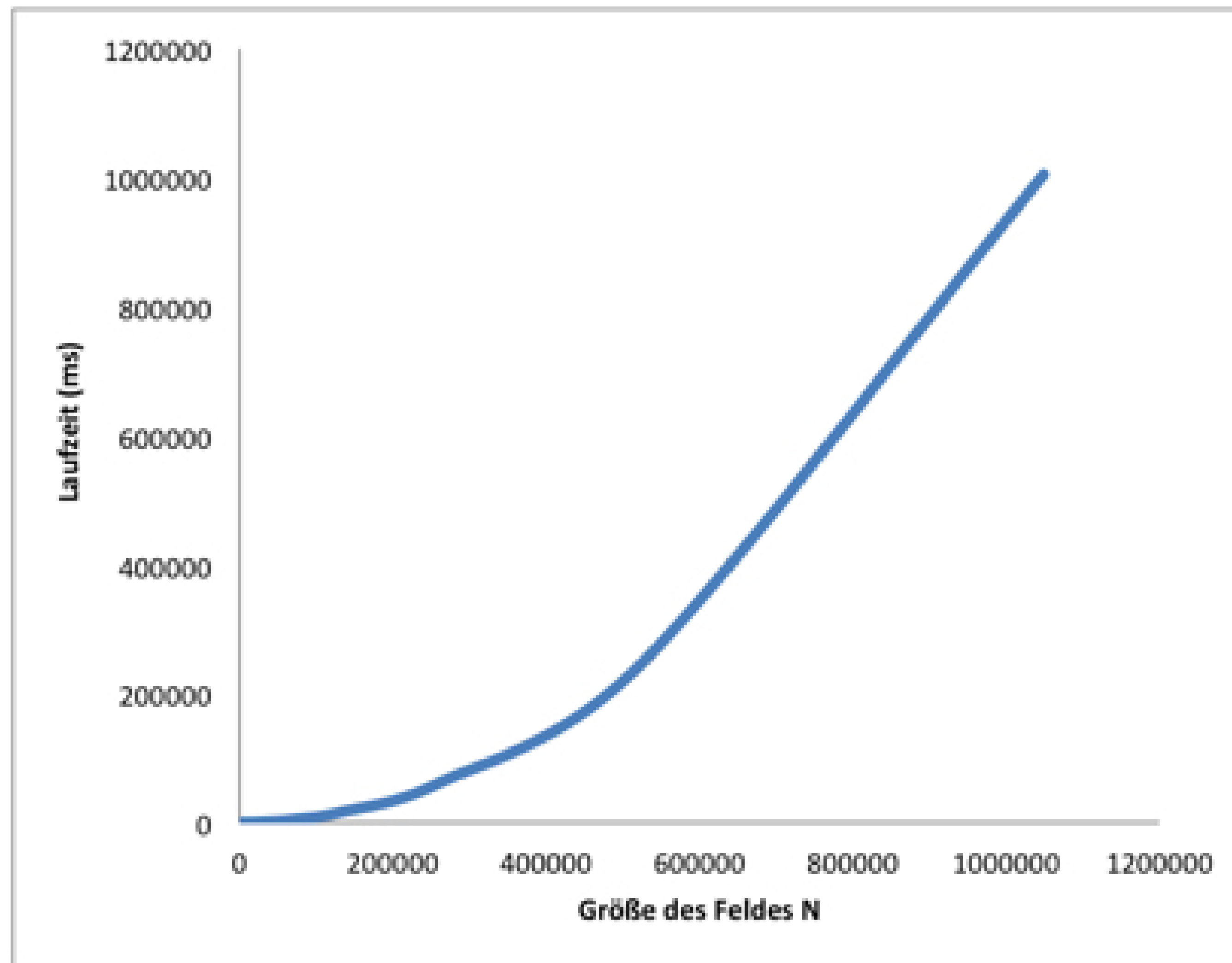
Um die Frage zu beantworten, wie viel tatsächlich die Laufzeit mit wachsender Arraygröße zunimmt, wird häufig die **Verdopplungshypothese** verwendet.
Man erhält diese Hypothese als Antwort auf die folgende Frage:

Welche Auswirkungen hat es auf die Laufzeit,
wenn wir die Problemgröße verdoppeln?

Messungen für verschiedene Eingabengrößen I

N	Laufzeit (ms)
2	0
4	0
8	0
16	0
32	0
64	0
128	0
256	1
512	2
1024	4
2049	7
4096	15
8193	61
16384	244
32769	966
65536	3849
131072	15443
262145	62219
524288	246468
1048576	1004586

Messungen für verschiedene Eingabengrößen II



Schritt 2: Ermitteln der Hypothese

Hypothese zum Laufzeitverhalten von Sortieren durch Auswahl

Die Laufzeit erhöht sich ungefähr um den Faktor 4, wenn sich die Eingabegröße verdoppelt.

- Weitere Messungen stützen diese Hypothese (Schritt 3 + 4)
- Können wir diese Hypothese durch eine mathematische Analyse verifizieren?

Mathematisches Modell zum Laufzeitverhalten

Nach D. E. Knuth ist die Laufzeit eines Programms bestimmt durch

- die (Laufzeit-)Kosten für die Ausführung der verschiedenen Operationen
- die Häufigkeit der Ausführung der verschiedenen Operationen

Es muss also zunächst geklärt werden:

- Welche Operationen werden konkret betrachtet?
- Welche Kosten entstehen bei diesen Operationen?

Kostenmodell für die Sortierung auf Integer-Arrays

Wir betrachten die folgenden Operationen

- Vergleich
- Zuweisung bzw. Inkrement / Dekrement
- Addition / Subtraktion

Vereinfachende Annahmen:

- Alle Operationen brauchen die gleiche Zeit.
- Alle anderen Aspekte vernachlässigen wir zunächst.

Laufzeitanalyse von Sortieren durch Einfügen

Zunächst wandeln wir die beiden `for`-Schleifen zu `while`-Schleife um:

```
static void sort (int[] f) {  // Annahme: f != null
  int i = 0;
  while (i < f.length -1) { // Schleife A

    // Finde Index fuer minimales Element
    int imin = i;
    int j = i+1;
    while (j < f.length) {  // Schleife B
      if (f[j] < f[imin]) {
        imin = j;
      }
      j++;
    }

    // Vertausche Elemente
    int temp = f[i];
    f[i] = f[imin];
    f[imin] = temp;
    i++;
  }
}
```

Laufzeitanalyse von Sortieren durch Einfügen: Schleife B

In jedem Schleifendurchlauf von Schleife B:

- ein Vergleich (Abbruchbedingung)
- ein Vergleich (if-Bedingung)
- eine Zuweisung (zu `imin`, falls die `if`-Bedingung sich zu `true` ergibt)
- ein Inkrement (von `j`)

Insgesamt: 4 Operationen

Laufzeitanalyse von Sortieren durch Einfügen: Schleife B

In jedem Schleifendurchlauf von Schleife A:

- eine Subtraktion (in Abbruchbedingung)
- ein Vergleich (Abbruchbedingung)
- eine Zuweisung (`imin`)
- eine Addition (`i`+1)
- eine Zuweisung (`j`)
- $N - (i + 1)$ mal der Rumpf der B-Schleife (jeweils 4 Operationen im schlimmsten Fall)
- ein Vergleich (Abbruchbedingung der Schleife)
- eine Zuweisung (`temp`)
- eine Zuweisung (`f[i]`)
- eine Zuweisung (`f[imin]`)
- ein Inkrement (`i`)

Insgesamt: $10 + 4 \cdot (N - (i + 1))$ Operationen

Laufzeitanalyse von Sortieren durch Einfügen: Gesamt I

Gesamte Prozedur `sort()`:

- eine Zuweisung (`i`)
- $N-1$ mal der Rumpf der A-Schleife (je $10+4\cdot(N-(i+1))$ Operationen für Werte $i=0,1,\ldots,N-2$)
- eine Subtraktion (Abbruchbedingung der Schleife)
- ein Vergleich (Abbruchbedingung der Schleife)

$$T_A(N) = 3 + \sum_{i=0}^{N-2}(10+4\cdot(N-(i+1)))$$

Laufzeitanalyse von Sortieren durch Einfügen: Gesamt II

$$
\begin{aligned}
T_A(N) &= 3 + \sum_{i=0}^{N-2}(10 + 4 \cdot (N - (i+1))) \\
&= 3 + \sum_{i=1}^{N-1}(10 + 4 \cdot (N - i)) \\
&= 3 + \sum_{i=1}^{N-1}(10 + 4 \cdot N - 4 \cdot i)) \\
&= 3 + \sum_{i=1}^{N-1} 10 \quad + \sum_{i=1}^{N-1} 4 \cdot N \quad - \sum_{i=1}^{N-1} 4 \cdot i \\
&= 3 + (N-1) \cdot 10 + (N-1) \cdot 4 \cdot N - 4 \cdot \sum_{i=1}^{N-1} i \\
&\overset{\text{für } N>1}{=} 3 + (N-1) \cdot 10 + (N-1) \cdot 4 \cdot N - 4 \cdot \frac{N \cdot (N-1)}{2} \\
&= 2 \cdot N^2 + 8 \cdot N - 7
\end{aligned}
$$

Bemerkungen

- Eine exakte Analyse kann bisweilen sehr komplex werden.
- Im Beispiel: Schwierig genau zu analysieren, wie oft die Zuweisung `imin = j;` im Allgemeinen erfolgt.
- Häufig genügt es die **Größenordnung** des Wachstums einer **Aufwandsfunktion** $A(N)$ (im obigen Beispiel $T_A(N)$) zu betrachten, die den Zeit- bzw. Speicheraufwand in Abhängigkeit von der Problemgröße N beschreibt.
- Die Größenordnung ist unabhängig vom tatsächlichen Rechner und erlaubt es insbesondere Laufzeiten verschiedener Algorithmen für große Problemgrößen zu vergleichen.
- Um Laufzeitgarantien zu geben, wird dabei oft konservativ der schlimmste Fall (engl. *worst case*) betrachtet.

O-Notation

Eine Funktion $f : \mathbb{N} \to \mathbb{R}^+$ heißt **obere Schranke** einer Aufwandsfunktion A, wenn gilt:

Es gibt $c, d \in \mathbb{N}$, sodass für alle $N \in \mathbb{N}$ gilt:

$$A(N) \leq c * f(N) + d$$

Man sagt auch, A wächst wie f bzw. ist von der Größenordnung f.
Die Menge aller Funktionen von der Größenordnung f bezeichnet man mit $O(f)$:

$$O(f) = \{g \mid \exists c, d \in \mathbb{N} : \forall N \in \mathbb{N} : g(N) \leq c * f(N) + d\}$$

Beispiel: Bestimmung oberer Schranken

Der Zeitaufwand T vom Sortieren durch Auswahl ist für $N \geq 2$ nach oben abschätzbar durch:

$$T_A(N) \leq 2N^2 + 8N - 7$$

und damit in $O(N^2)$; denn mit $c = 3$ und $d = 9$ gilt:

$$T_A(N) \leq 3N^2 + 9$$

Wichtige Komplexitätsklassen

Klasse	Bezeichnung	Beispiel
$O(1)$	*konstant*	Schaltjahrberechnung, Hashverfahren
$O(\log N)$	*logarithmisch*	binäre Suche in Bäumen
$O(N)$	*linear*	Mittelwerts von Arrayeinträgen
$O(N \log N)$	*linearithmetisch*	Mergesort
$O(N^2)$	*quadratisch*	Sortieren durch Einfügen
$O(N^3)$	*kubisch*	Matrixmultiplikation
$O(2^N)$	*exponentiell*	diverse Optimierungsverfahren

Algorithmen mit einem Aufwand in $O(N^k)$ nennt man **polynomiell** (engl. *polynomial*).

Laufzeitabschätzung basierend auf Komplexitätsklassen

Annahme: Programm benötigt für ein Problem der Größe N auf einem Rechner wenige Sekunden.

- Wie verhält sich die Laufzeit des Programm, wenn man die Problemgröße erhöht?
- Um wieviel kann ein schnellerer Rechner die Berechnungszeit verringern?

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Lösbarkeit großer Probleme

Wenn man die Problemgröße um den Faktor 100 erhöht, wächst die Laufzeit von einigen Sekunden auf . . .

Klasse	Laufzeitabschätzung
linear	einige Minuten
linearithmetisch	einige Minuten
quadratisch	mehrere Stunden
kubisch	einige Wochen
exponentiell	Milliarden von Jahren

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Nutzung schnellerer Rechner

Wenn man einen Computer verwendet, der um den Faktor 10 schneller rechnet, kann in der gleichen Zeit eine Probleminstanz gelöst werden, die um den Faktor ... größer ist.

Klasse	Faktor der Erhöhung der Problemgröße
linear	10
linearithmetisch	10
quadratisch	3-4
kubisch	2-3
exponentiell	1

Bemerkungen

- O-Notation liefert nur eine grobe Klassifizierung durch obere Schranke
- Fokus auf asymptotisches Verhalten bei großen Eingaben
- (Relativ einfache) theoretische Einordnung und Vergleichsbasis
- In der Praxis können konstante Faktoren und programmiersprachenspezifische Aspekte entscheidend sein $\Rightarrow$ Kostenmodell!

Beispiel: Stringrepräsentation

- Interne Repräsentierung von Strings hat großen Einfluss auf Laufzeit (und auch Speicher!) von Stringoperationen
- Typischerweise in Java 1.6 und davor: Strings sind gegeben als `char[]` mit Startposition im Array und Länge
  $\Rightarrow$ Substring-Extraktion ist in $O(1)$

```
String fullName = "Bill Gates";
String firstName = fullName.substring(0, 3);
```

- Seit Java 1.7: Substrings teilen nicht mehr das gleiche `char[]`-Array, sondern erhalten eine eigene Kopie des Sub-Arrays
  $\Rightarrow$ Substring-Extraktion ist nun in $O(N)$, wobei N die Länge des Substrings ist

Speicherverwaltung

Übersicht

Speicher ist eine wichtige Ressource für Softwaresysteme. Viele nicht-funktionale Eigenschaften hängen vom angemessenen Umgang mit Speicher ab.

Wir betrachten grundlegende Aspekte der Speicherverwaltung:

- Verwendung von Speicher
- Automatische Speicherbereinigung

Wir betrachten hier nur den Speicher, der für die Ausführung von Programmen benötigt wird, und zwar in Form eines Byte-adressierbaren virtuellen Adressraums.

Wie viel Speicher ein Programm in Abhängigkeit von der Eingabe genau braucht, hängt von Details der Sprachimplementierung und Plattform ab.

Aspekte der Speicherverwendung

Wie viel Speicher braucht ein Programm?

Wofür wird Speicher benötigt?

Wie ist der Speicher organisiert?

Wie wird der Speicher verwaltet?

Speicherbedarf

- Werte, Objekte, Arrays, etc.
- Programm (unabhängig von Eingabedaten)
- Verwaltung von Prozedur-/Methodenaufrufen

Der tatsächliche Speicherbedarf hängt von der konkreten Laufzeitumgebung ab.
Wir betrachten hier typische Größen.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Speicherbedarf: Primitive Datentypen

Datentyp	Bytes (= 8 Bit)
boolean	1
char	2
int	4
float	4
long	8
double	8

Speicherbedarf: Objekte

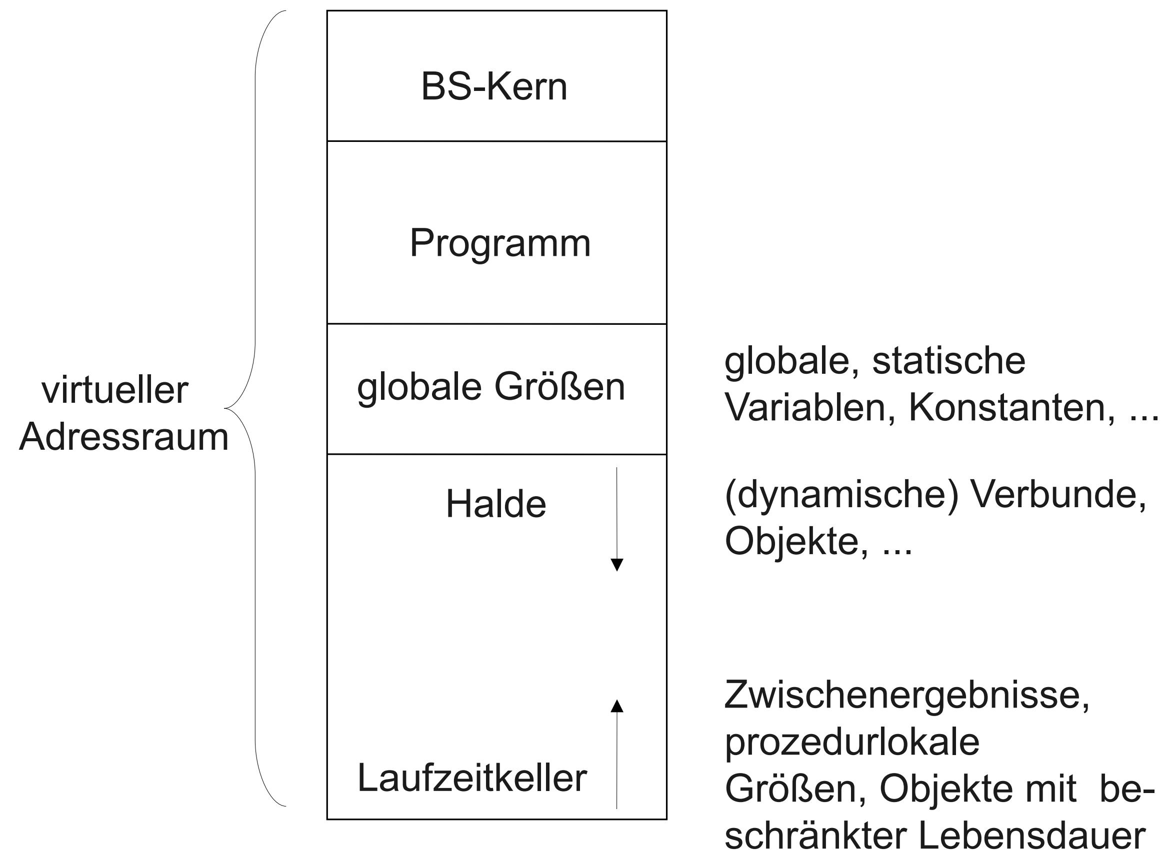
- Referenz auf ein Objekt: 4 Byte
- Jedes Objekte benötigt Speicher für jedes seiner Objektattribute (abhängig von dessen Typ) sowie einmalig 8 Byte Overhead zur Verwaltung des Objekts
- Sonderfall Strings: 24 Byte Overhead sowie Speicher für das zugehörige interne `char[]`-Array

Speicherbedarf: Arrays

- 16 Byte Overhead (u.a. Größe des Arrays)
- Wenn das Array N Elemente eines Typs enthält und M Byte zum Speichern eines Elements benötigt werden, brauchen das Array zusätzlich $N \times M$ Byte für die Elemente.
- Beispiel: Integer-Array der Größe N $\Rightarrow$ belegt $16 + 4N$

Speicherorganisation

Die Speicherorganisation ist bei den meisten prozeduralen bzw. objektorientierten Programmiersprachen ähnlich:



Speicherverwaltung I

1. Globaler Speicher und Keller werden automatisch verwaltet (der Übersetzer erzeugt dafür Code).
2. Je nach Programmiersprache wird die **Halde** (engl. **heap**) unterschiedlich verwaltet:
   - mit automatischer Speicherbereinigung,
   - durch den Programmierer (Deallokation).

Speicherverwaltung II

Operationen zur Verwaltung des Heap

→ Anfordern von Speicher bei Verbund-/Objekterzeugung

→ Freigabe von Speicher

- Wenn kein Speicher mehr verfügbar, gebe die Speicherbereiche von Objekten frei, die nicht mehr erreichbar sind.
- Gebe Speicher von Objekten auf Anweisung des Programms frei.

Die Speicherverwaltung kostet auch Laufzeit!

Begriffsklärung: De-/Allokation

Die Bereitstellung des Speicherbereichs bei der Erzeugung von Verbunden und Objekten nennt man **Allokation** (engl. *allocation*).

Die Freigabe solcher Speicherbereiche nennt man **Deallokation** (engl. *deallocation*).

Automatische Speicherbereinigung

Immer mehr Programmiersprachen kommen mit **automatischer Speicherbereinigung** (engl. *automatic garbage collection*).

Dabei wird periodisch oder bei Bedarf ermittelt, welche Verbunde/Objekte nicht mehr erreichbar (s.u.) sind.
Dieser Speicherplatz wird freigegeben und gegebenenfalls wird die Speicherstruktur umgeordnet und kompaktifiziert.

- Vereinfachung der Programmierung
- Aufwand an Speicher und Zeit ist vertretbar

Beispiel: Automatische Speicherbereinigung I

Programm, das unerreichbare Objekte erzeugt:

```
public static void main (String[] args) {
  int count = 1;
  while (true) {
    int[] feld = new int[1000000];
    StdOut.println(count++);
  }
}
```

Dieses Programm bekommt keine Speicherprobleme.

Beispiel: Automatische Speicherbereinigung II

Programm, dessen erzeugte Objekte erreichbar bleiben:

```
class ListOfArrays {
  // Einfach verkettete Liste von Integer-Arrays
  ...
}

public static void main (String[] args) {
  ListOfArray la = new ListOfArrays();
  int count = 0;
  while (true) {
    la.add(new int[1000000]);
    count++;
    StdOut.println(count);
  }
}
```

Dies Programm terminiert mit einem `OutOfMemoryError`.

Begriffsklärung: Erreichbarkeit

Ein Objekt X heißt von einer Variablen v **direkt erreichbar**, wenn v eine Referenz auf X enthält.

X heißt von v **erreichbar**, wenn es von v direkt erreichbar ist oder wenn es ein Objekt Y mit Attribut w gibt, so dass X von w direkt erreichbar ist und Y von v erreichbar ist.

Die Menge der **Wurzelvariablen** zu einem Ausführungszustand A umfasst alle globalen Variablen sowie die aktuell im Keller vorhandenen lokalen Variablen und Parameter.

Ein Verbund bzw. Objekt heißt **erreichbar** *in einem Ausführungszustand* A, wenn es von einer Wurzelvariablen zu A erreichbar ist.

Deallokation von Verbunden/Objekten

Einige prozedurale Programmiersprachen unterstützen die De-/Allokation durch den Programmierer.

- Vorteil:
  - ermöglicht sehr effiziente und flexible Benutzung von Speicher
- Nachteile:
  - zusätzlicher Programmieraufwand
  - potentielle Fehlerquelle, schwierig zu debuggen

Zusammenfassung

Das Laufzeitverhalten eines Programms wird bestimmt durch:

- die Anweisungen des Programms
- den Übersetzer
- die Laufzeitumgebung (insbesondere Speicherverwaltung und Systemumgebung)

Speicherverwaltung kostet Zeit

Speicherverwaltung ist aufwendig, wenn der verfügbare Speicher knapp wird.

- Garbage Collector muss häufig aufgerufen werden
- Aufsuchen ausreichend großer freier Speicherbereiche wird aufwendiger

In der Praxis ist der Aufwand nicht leicht abzuschätzen, weil

- der Speicherverbrauch der Bibliotheksklassen und anderer fremder Programmteile häufig nicht klar spezifiziert ist;
- die Details der Speicherverwaltung eine wichtige Rolle spielen.

Problematisch ist das insbesondere bei Echtzeitanforderungen.

Systemumgebung beeinflusst das Laufzeitverhalten

Zur Gesamtbeurteilung des Laufzeitverhaltens eines Softwaresystems muss auch die Systemumgebung berücksichtigt werden.

- Benutzerinteraktion
- Anzahl von Benutzern
- Kommunikationszeiten
- Laufzeitverhalten der Plattform
- Interaktion mit anderen Systemen

Abschließender Hinweis

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.
[Donald Knuth, 1974]

- Es ist essentiell das Laufzeitverhalten eines Programms im Blick zu behalten.
- Die meisten Optimierungen sind allerdings kontraproduktiv:
  - meist komplex und schwer zu verstehen
  - nur geringe Verbesserungen
  - oft inkorrekt
- Unser Ziel ist zunächst klarer und korrekter Code!