Algorithmen

- Sortieren durch Auswählen, Sortieren durch Mischen und Vergleich der Laufzeit
- Abschätzung der Laufzeit eines Algorithmus, O-Notation.
- Rekursion

Sortieren durch Auswählen

Wir wollen ein Array `a` von `int`-Zahlen der Größe nach sortieren:

Z.B. $11, 9, 17, 5, 12$ soll $5, 9, 11, 12, 17$ werden.

Wir suchen das kleinste Element, hier `a[3]` $= 5$, und schaffen es nach vorne durch Vertauschen mit dem ersten Element:

$$\boxed{11}\ 9\ 17\ \boxed{5}\ 12$$
$$\boxed{5}\ 9\ 17\ \boxed{11}\ 12$$

Dann suchen wir das kleinste Element von `a[1..4]`. Es ist schon an der richtigen Stelle.

Dann das kleinste Element von `a[2..4]`. Es ist `a[3]=11`. Vertauschen mit `a[2]` führt auf

$$5\ 9\ \boxed{11}\ \boxed{17}\ 12$$

Das kleinste Element von `a[3..4]` wird noch mit `a[3]` vertauscht und wir sind fertig.

Dasselbe in Java

Zunächst das Testprogramm

```
public class SelSortTest
{  public static void main(String[] args)
   {  int[] a = ArrayUtil.randomIntArray(20, 100);

      ArrayUtil.print(a);
      SelSort.sort(a);
      ArrayUtil.print(a);
   }
}
```

Siehe Javadoc zu ArrayUtil, SelSort.

Sortieren durch Auswählen in Java

```
package sorting;
public class SelSort
{  /**
      Finds the smallest element in an array range.
      @param a the array to search
      @param from the first position in a to compare
      @return the position of the smallest element in the
      range a[from]...a[a.length - 1]
   */
   public static int minimumPosition(int[] a, int from)
   {  int minPos = from;
      for (int i = from + 1; i < a.length; i++)
         if (a[i] < a[minPos]) minPos = i;
      return minPos;
   }
```

Sortieren durch Auswählen in Java

```
   /**
      Sorts an array.
      @param a the array to sort
   */
   public static void sort(int[] a)
   {  for (int n = 0; n < a.length - 1; n++)
      {  int minPos = minimumPosition(a, n);
         if (minPos != n)
            ArrayUtil.swap(a, minPos, n);
      }
   }
}
```


Testen

```
mhofmann@branford:~/work/teaching/EinfInfo> java sorting/SelSortTest
52 23 37 65 79 95 21 27 12 12 78 66 66 51 7 39 81 86 95 74
7 12 12 21 23 27 37 39 51 52 65 66 66 74 78 79 81 86 95 95
```

Für längere Arrays die `print` Statements 'rauskommentieren.

Bis Größe 10000 ist die Laufzeit $2s$.

Bei 50000 dauert es ca. $20s$.

Bei 100000 dauert es mehrere Minuten.

Bei 1500000 dauert es mehrere Stunden.

Wir führen eine genauere empirische Analyse durch:

Stoppuhr

Die Methode `System.currentTimeMillis()` liefert die Anzahl der Millisekunden, die seit 00:00 am 1.1.1970 verstrichen sind (ca. 1 Trillion $> 2^{31}$ daher ist `long` erforderlich.)

Damit können wir eine "Stoppuhr-Klasse" bauen, die die Methoden

`reset()`
`start()`
`stop()`
`getElapsedTime()`

bereitstellt (Details siehe Javadoc).

Stoppuhr

```
package sorting;
/**
   A stopwatch accumulates time when it is running. You can
   repeatedly start and stop the stopwatch. You can use a
   stopwatch to measure the running time of a program.
*/

public class StopWatch
{
   private long elapsedTime;
   private long startTime;
   private boolean isRunning;

   /**
      Starts the stopwatch. Time starts accumulating now.
   */
   public void start()
```

Stoppuhr

```
   {  if (isRunning) return;
      isRunning = true;
      startTime = System.currentTimeMillis();
   }

   /**
      Stops the stopwatch. Time stops accumulating and is
      is added to the elapsed time.
   */
   public void stop()
   {  if (!isRunning) return;
      isRunning = false;
      long endTime = System.currentTimeMillis();
      elapsedTime = elapsedTime + endTime - startTime;
   }

   /**
```


Stoppuhr

```
      Returns the total elapsed time.
      @return the total elapsed time
   */
   public long getElapsedTime()
   {  if (isRunning)
      {  long endTime = System.currentTimeMillis();
         elapsedTime = elapsedTime + endTime - startTime;
         startTime = endTime;
      }
      return elapsedTime;
   }

   /**
      Stops the watch and resets the elapsed time to 0.
   */
   public void reset()
   {  elapsedTime = 0;
```


Stoppuhr

```
      isRunning = false;
   }

  /**
      Constructs a stopwatch that is in the stopped state
      and has no time accumulated.
   */
   public StopWatch()
   {  reset();
   }

}
```


Laufzeit von `SelSort`

```
public class SelSortTime
{  public static void main(String[] args)
   {  ConsoleReader console = new ConsoleReader(System.in);
      System.out.println("Enter array size:");
      int n = console.readInt();
      int[] a = ArrayUtil.randomIntArray(n, 100);
      StopWatch timer = new StopWatch();
      timer.start();
      SelSort.sort(a);
      timer.stop();
      System.out.println("Elapsed time: "
         + timer.getElapsedTime() + " milliseconds");
   }
}
```

Laufzeitmessung

n	Laufzeit in ms
500	7
1000	14
1500	27
2000	54
2500	66
3000	93
3500	133
$10K$	992
$20K$	3939
$30K$	8848
$40K$	15858

Analyse der Laufzeit

Als grobes Maß für die Laufzeit wählen wir die Anzahl der Arrayzugriffe.

Die wirkliche Laufzeit ist auf jeden Fall größer als ein festes Vielfaches dieser Zahl.

Wir schätzen die Zahl der Arrayzugriffe von unten ab:

Sei n die Arraygröße.

Abschätzung der Laufzeit

Finden des kleinsten Elements: n Zugriffe.

Finden des 2.kleinsten Elements: $n-1$ Zugriffe.

Finden des 3.kleinsten Elements: $n-2$ Zugriffe.

Finden des $n-1$.kleinsten Elements: 2 Zugriffe.

Macht zusammen $2+3+4+5+\cdots+n = \frac{n(n+1)}{2} - 1 = \frac{1}{2}n^2 + \frac{1}{2}n - 1$.

Das Vertauschen haben wir gar nicht gerechnet!

Größenordnung der Laufzeit

Zahl der Arrayzugriffe $\geq \frac{1}{2}n^2 + \frac{1}{2}n - 1$.

Der lineare Term spielt für große n keine Rolle.

Der Faktor $1/2$ auch nicht, da die exakte Laufzeit sowieso durch Multiplikation mit einem maschinen- und implementationsabhängigen Wert entsteht.

Nur das quadratische Wachstum interessiert. Wir schreiben $1/2n^2 + 1/2n - 1 = O(n^2)$.

Die Laufzeit von *Selection Sort* ist $O(n^2)$

O-Notation

Zur Bestimmung der O-Notation finde man den am schnellsten wachsenden Term und lasse eventuelle Vorfaktoren weg.

$0,9n^3 + 890n^2 = O(n^3)$

$n^2(n^2 + 4n) = O(n^4)$

$2^n + n^{30000} = O(2^n)$

Für Pedanten

Eigentlich müsste man schreiben

$$1/2n^2 \in O(n^2)$$

denn $O(n^2)$ ist die Klasse der Funktionen von höchstens quadratischem Wachstum.

Das Gleichheitszeichen hat sich aber eingebürgert.

Formal (für Physiker und andere Matheinteressierte, nicht Prüfungsstoff!):

$$O(f(n)) = \{g(n) \mid \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty\}$$

Sortieren durch Mischen

Gegeben folgendes Array der Größe 10.

$$5, 9, 10, 12, 17, \quad 1, 8, 11, 20, 32$$

Die beiden "Hälften" sind hier bereits sortiert!

Mischen

Wir können das Array sortieren, indem wir jeweils von der ersten oder der zweiten Hälfte ein Element wegnehmen, je nachdem, welches "dran" ist:

$5, 9, 10, 12, 17, \quad \not{1}, 8, 11, 20, 32$	1
$\not{5}, 9, 10, 12, 17, \quad \not{1}, 8, 11, 20, 32$	$1, 5$
$\not{5}, 9, 10, 12, 17, \quad \not{1}, \not{8}, 11, 20, 32$	$1, 5, 8$
$\not{5}, \not{9}, 10, 12, 17, \quad \not{1}, \not{8}, 11, 20, 32$	$1, 5, 8, 9$
$\ldots$	$\ldots$

. . . und die weggenommenen Elemente in ein anderes Array kopieren.

Sortieren durch Mischen

Falls die beiden Hälften nicht schon sortiert sind, dann müssen wir sie eben vorher sortieren.

Wie? Durch Mischen der jeweiligen Hälften (also Viertel).

Und wenn die nicht schon sortiert sind? Dann werden wiederum die jeweiligen Hälften (also Achtel) gemischt.

Usw. bis man bei Arrays der Größe Eins angelangt ist, die ja stets sortiert sind.

Sortieren durch Mischen in Java

```
public static void mergeSort(int[] a, int from, int to)
{  if (from == to) return;
   int mid = (from + to) / 2;
    // sort the first and the second half
   mergeSort(a, from, mid);
   mergeSort(a, mid + 1, to);
   merge(a, from, mid, to);
}
```


Mischen

```
public static void merge(int[] a,
   int from, int mid, int to)
{  int n = to - from + 1;
      // size of the range to be merged

   // merge both halves into a temporary array b
   int[] b = new int[n];

   int i1 = from;
      // next element to consider in the first range
   int i2 = mid + 1;
      // next element to consider in the second range
   int j = 0;
      // next open position in b

   // as long as neither i1 nor i2 past the end, move
   // the smaller element into b
```


Mischen

```
   while (i1 <= mid && i2 <= to)
   {  if (a[i1] < a[i2])
      {  b[j] = a[i1];
         i1++;
      }
      else
      {  b[j] = a[i2];
         i2++;
      }
      j++;
   }

   // note that only one of the two while loops
   // below is executed

   // copy any remaining entries of the first half
   while (i1 <= mid)
```

Mischen

```
    {  b[j] = a[i1];
       i1++;
       j++;
    }

    // copy any remaining entries of the second half
    while (i2 <= to)
    {  b[j] = a[i2];
       i2++;
       j++;
    }

    // copy back from the temporary array
    for (j = 0; j < n; j++)
       a[from + j] = b[j];
}
```


Laufzeit von Merge Sort

n	Laufzeit in ms
500	7
3500	17
$10K$	35
$20K$	41
$30K$	56
$40K$	69
$50K$	94
$60K$	109
$80K$	138
$5M$	9612

Analytische Bestimmung der Laufzeit

Sei $T(n)$ die Zahl der Arrayzugriffe von Merge Sort bei Arraygröße n.

Es gilt:

$$T(1) = 0$$
$$T(n) = 2T(\tfrac{n}{2}) + 5n$$

falls $n = 2^k$ (ansonsten stimmt's immer noch "größenordnungsmäßig").

Die 5n kommen vom Mischen: 3n für's eigentliche Mischen, 2n für's Zurückschreiben.

Lösung der Gleichung:

$$T(n) = T(2^k) = 5{\cdot}2^k + 2T(2^{k-1}) = 5{\cdot}2^k + 2{\cdot}5\cdots 2^{k-1} + 4T(2^{k-2}) \cdots + 2^k{\cdot}T(1)$$

Wir raten: $T(2^k) = k \cdot 5 \cdot 2^k + 2^k \cdot T(1) = k \cdot 5 \cdot 2^k$.

Gegenprobe: $k \cdot 5 \cdot 2^k = 2(k-1) \cdot 5 \cdot 2^{k-1} + 5 \cdot 2^k$.

Analytische Bestimmung der Laufzeit

Also gilt $T(2^k) = 5 \cdot 2^k \cdot k$ oder $T(n) = 5n \log_2 n$.

Es ist: $5n \log_2(n) = O(n \log(n))$.

Die Basis lässt man weg, da alle Logarithmen proportional sind.

Die Laufzeit von Merge Sort ist $O(n \log(n))$

Sortieren in der Praxis

Will man andere Objekte als Zahlen sortieren, so verwende man

`Comparable[] a;`

und ersetze im Code jeweils `x < y` durch `x.compareTo(y) < 0`.

Man kann zeigen, dass jedes Sortierverfahren, das auf die Daten durch solche Vergleiche zugreift, mindestens $cn\log(n)$ Vergleiche braucht (für ein festes $c > 0$). Also ist Merge Sort in gewissem Sinne optimal.

Es gibt aber trotzdem bessere Verfahren, z.B. Quick Sort. Die haben zwar auch Laufzeit $O(n\log(n))$, aber eine bessere multiplikative Konstante, d.h., kleiner als die 5 bei Merge Sort.

Rekursion

Den Aufruf einer Methode in ihrem eigenen Rumpf bezeichnet man als Rekursion.

Man muss aufpassen, dass die Rekursion irgendwann zum Ende kommt.

```
public static void f() {
  f();
  }
```

ist schlecht. Sobald man `f()` aufruft, "hängt sich der Rechner auf."

Rekursion bietet sich immer dann an, wenn man die Lösung eines Problems auf die Lösung gleichartiger aber kleinerer Teilprobleme zurückführen kann.

Beispiel: Türme von Hanoi

Es gibt drei senkrechte Stäbe. Auf dem ersten liegen n gelochte Scheiben von nach oben hin abnehmender Größe.

Man soll den ganzen Stapel auf den dritten Stab transferieren, darf aber immer nur jeweils eine Scheibe entweder nach ganz unten oder auf eine größere legen.

Angeblich sind in Hanoi ein paar Mönche seit Urzeiten mit dem Fall $n = 64$ befasst.

Lösung

Für $n = 1$ kein Problem.

Falls man schon weiss, wie es für $n - 1$ geht, dann schafft man mit diesem Rezept die obersten $n - 1$ Scheiben auf den zweiten Stab (die unterste Scheibe fasst man dabei als "Boden" auf.).

Dan legt man die größte nunmehr freie Scheibe auf den dritten Stapel und verschafft unter abermaliger Verwendung der Vorschrift für $n - 1$ die restlichen Scheiben vom mittleren auf den dritten Stapel.

Laufzeit des Verfahrens:

$$T(1) = 1$$

$$T(n) = 2T(n-1) + 1$$

Lösung: $T(n) = 2^n - 1$.

Man kann sich überlegen, das jede beliebige Methode auch mindestens so lange braucht.

$\rightarrow$: die Mönche werden nie fertig.

Zusammenfassung

- Verschiedene Algorithmen (Rechenverfahren) für das gleiche Problem können sich drastisch in der Laufzeit unterscheiden.
- Die O-Notation gestattet es, Angaben über die Größenordnung einer Funktion, z.B. der Laufzeit zu machen.
- Selection Sort ist ein $O(n^2)$ Verfahren, Merge Sort ist ein $O(n \log(n))$ Verfahren zum Sortieren von Arrays. Merge Sort ist auch empirisch wesentlich performanter.
- Rekursive Verfahren beruhen auf der Zerlegung eines Problems in kleinere gleichartige Probleme.
- Formal bedeutet Rekursion den Aufruf einer Methode in ihrem eigenen Rumpf.
- Man sollte bei Rekursion die Zahl der Aufrufe in Bezug auf die Reduktion der Problemgröße im Auge behalten. Problem um eins kleiner, dafür zwei Aufrufe $\rightarrow$ exponentielle Laufzeit.