# Algorithmen

* Algorithmen sind genaue Anweisungen, um ein Ziel zu erreichen
* Müssen komplett beschrieben sein (nichts auslassen)
* müssen reproduzierbar sein (Kochrezept ist kein Algorithmus)

# Programmiersprachen

Eine Programmiersprache ist eine Hochsprache, sodass Menschen mit der Maschine (Sprache –> Maschinensprache) kommunizieren können.

Übersetzt wird sie durch den Compiler. Sie ist wie eine normale Sprache mit Schlüsselwörtern, Grammatik, … definiert.

Es gibt Hochsprachen (Java, Python …) und maschinennahe Sprachen (Assembler).

**Interpretersprachen (Java, C#, C++):**Compiler wandelt Programmcode nur in Bytecode um (z.B. .class Datei in Java).  
Der Interpreter verarbeitet diesen Bytecode in Maschinensprache (Binärcode).  
🡺 Vorteil: plattformunabhängig

**Compilersprachen (C):**Hier wird der Programmcode vom Compiler sofort in Maschinensprache umgewandelt.  
🡺 nicht plattformunabhängig (für alle Betriebssysteme: 16 / 32 / 64 – Bit Versionen einzeln)

Java basiert auf C 🡺 C basiert auf Assembler

# Allgemeines

* Static referenziert auf eine Klasse und nicht auf eine Instanz 🡺 static - Members gibt es daher nur einmal (für alle Instanzen gleich)
* Protected Memebers…man hat nur in dieser Klasse sowie in den Unterklassen Zugriff.
* Default…im gleichen Package verwendbar

# Prozesse und Threads

Wird ein Programm gestartet reserviert das Betriebssystem einen bestimmten Speicherplatz im RAM für das Programm (z.B. beim Browser wird ganzer Inhalt in den Prozessspeicher geschrieben)  
🡺 = Prozessspeicher.

**Prozesse** haben ihren eigenen Speicher (Prozessspeicher). Sie sind immer abgeschirmt und können daher keine Daten untereinander austauschen.

**Threads** sind innerhalb von Prozessen (Prozess kann aus mehreren Threads bestehen). Sie sind „Unterprogramme“. Diese können untereinander kommunizieren.

# Binär Suche

Ist ein effizienter Algorithmus, mit dem ein Objekt in einer sortierten Liste gefunden werden kann.

Teil der Liste, in dem sich das Objekt befinden könnte wird mit jedem Durchgang halbiert.

Z.B.: um ein Objekt in einem Array zu suchen.

**Zum Unterschied von linearer/iterativer Suche (Im Schlimmsten Fall bei n Elementen n Schritte**) **hat man wesentlich weniger Schritte (bei n Elemente: log2(n) + 1 Schritte und abrunden).**

**Funktionsweise:**

1. Setze min = 1 und max = n;
2. Schätze den Durchschnitt von max und min auf eine ganze Zahl abgerundet
3. Wenn diese Zahl die gesuchte Zahl ist, hör auf.
4. Wenn die Schätzung zu niedrig war, setze min um 1 höher als die Schätzung
5. Wenn die Schätzung zu hoch war, setze max um 1 kleiner als die Schätzung.
6. Wiederholen ab 2.

Soll so lange funktionieren, bis max < min ist (Objekt nicht gefunden).

int doSearch = function(array, targetValue) {

double min = 0;

double max = array.length - 1;

int guess;

while(min <= max){

guess = (min + max) / 2; //int daher wird gerundet

if(array[guess] == targetValue){return guess;}

else if(array[guess] < target Value){min = guess + 1;}

else{max = guess – 1;}

}

return -1;

};

# Stack & Heap

Sind Speichemanagementsysteme.

**Stack:**

* Stapel – nur oberstes Element nehmen
* Ist im Prozesspeicher eines Prozesses.
* Speichert lokale Variablen
* Schnell
* Wenn Terminierungsbedingung nicht funktioniert 🡺 Stak-Overflow-Error
* Ruft eine Methode eine andere auf, wird alles der zuvor aufgerufenen Methode in den Stack gepackt. Bei der letzten Methode angekommen, wird der Stack von unten nach oben abgearbeitet.

**Heap:**

* Ist der Speicher außerhalb von Prozessen (kann im RAM oder auf Festplatte sein)
* RAM, begrenzt Größe vom Heapspeicher
* Speichert größere Dinge (Listen, Membervariablen = in Klasse aber außerhalb Methoden)
* Arbeitet mit Zeigern (Adresse des nächsten Objektes wird gespeichert
* Beim Löschen wird nur überschrieben (Objekte werden verteilt gespeichert wegen entstehenden Lücken 🡺 Festplatte muss aber um ein Objekt zu lesen zu allen Teilen hinspringen 🡺 man defregmentiert (Umordnung, dass Objekte wieder zusammen sind)
* Defregmentiert 🡺 deshalb ist er langsamer

# Iterativ

* Mehrfache Ausführung einer oder mehrerer Anweisungen.
* Mit Abbruchbedingung wird die Schleife beendet.

Sind performanter als Rekursion (Speicherbedarf bleibt bei jedem Durchlauf gleich).

Compiler versteht nur Iterativ.

# Rekursiv

* Methode ruft sich selbst mit anderen Parametern auf.
* Muss eine Terminierungsbedingung haben (zum Beenden).
* Muss einen return-Type haben (void geht nicht).

Rekursion wird verwendet, da es übersichtlicher und weniger Code ist.

Compiler wandelt Rekursion in Endrekursion und dann in Iteration um.

**Fakultät:**

**int** fak(int i){

**if** (i <= 1){

**return** 1;

}

**return** i \* fak(i - 1);

}

Der benötigte Speicherplatz wächst bei jedem Durchlauf linear an, da jedes Mal ein neuer Speicherbereich für die Variable und die Rücksprungadresse reserviert werden muss.

Ein Bild, das Tisch enthält.

Automatisch generierte BeschreibungRekursion wird durch ein Stack implementiert indem immer Platz für das Ergebnis, das Argument und die Rücksprungadresse gelassen wird. Am Schluss wird das ganze Stack von unten nach oben abgearbeitet.

sum(3) = 3 + sum(2)

sum(2) = 2 + sum(1)

sum(1) = 1 + sum(0)

sum(0) = 0

sum(1) = 1 + 0 = 1

sum(2) = 2 + 1 = 3

sum(3) = 3 + 3 = 6

Die letzte Aktion ist dann eine Rechnung von Zahlen (z.B. 3 + 3 = 6).

# Endrekursion

* Letzte Rechnung ist ein Rekursionsaufruf (z.B. sumRek(6, 0) = 6)
* Bei erneutem Aufruf wird kein zusätzlicher Speicherplatz benötigt (effektiver als Rekursion).
* Hat gleich viele Schritte wie die Iteration.

Wird vom Compiler (wenn er optimiert wurde) in Iteration umgewandelt.

Gerechnet wird nur innerhalb des Methodenkopfes.   
Meistens gibt es eine Hilfsfunktion (Mit Zähler und Result).

sumRek(m, n){

if (n == 0){

return m;

}

return sumRek(m + n, n - 1);

}

Zwischenergebnisse warden im Paramter m gespeichert.

sum(3) = add\_sum(0, 3)

= add\_sum(3, 2)

= add\_sum(5, 1)

= add\_sum(6, 0)

= 6

# Zusammenhang Iteration – Rekursion – End R.

Prozessor kann nur Iterativ rechnen. Compiler schafft es nicht, die Rekursion direkt in die Iterative Variante umzuwandeln 🡺 daher gibt es die Endrekursion (Zwischending) in die zuerst umgewandelt wird. Erst bei der Endrekursion kann der Compiler sagen wie viele Schritte er für Rekursion braucht.

Rekursion ist langsamer als Iteration 🡺 da beim Umwandeln nicht in die effektivste Version gewandelt wird, sondern in die Codeähnlichste.

--------------------------------------------1.Test----------------------------------------

# Zufall

Zufall, der durch Software erzeugt wird, ist kein richtiger Zufall (Pseudozufall)

* Formeln im Computer werden immer gleich ausgeführt
* Möglichst einzigartiger Ausgangspunkt wird genommen (z.B. Millisekunden aus Uhrzeit)
* Bei Auswertung: Zufall wiederholt sich (Häufungen entstehen)

🡺 Problem: Verschlüsselungen oft durch Zufall 🡺 werden zurückrechenbar

Daher nimmt man Primzahlen 🡺 sie wiederholen sich nicht 🡺 möglichst große Primzahl nehmen

# Aufwandsklassen

Aufwandsklasse gibt an, wie viele Schritte (z.B. Schleifendurchläufe) es braucht, um einen Algorithmus (z.B. Sortieralgorithmus) abzuarbeiten.

* Mit O-Notation angegeben
* Aufwandsklasse immer von n (Anzahl der zu sortierenden Elemente) abhängig

**3 Aufwandsklassen:**

* **Worst-Case** (alle Elemente der Ausgangsfolge 🡺 entgegen unserer Sortierlogik geordnet)
* **Average-Case** (… 🡺 durchschnittliche/irgendwie verteilt)
* **Best-Case** (…🡺schon in der richtigen Reihenfolge, für unsere Logik)

**Wie kann ein Durchlauf ausschauen?**

* **Vergleichen** von 2 Elementen 🡺 ist weniger Aufwand (nur durch CPU; sie ist schnell)
* **Vertauschen** von 2 Elementen 🡺 ist mehr Aufwand (speichern dauert länger)

Bei Angabe der Aufwandsklasse geht man **immer vom Worst-Case aus**.

Stabilität  
Beim Sortieren werden gleiche Elemente so sortiert, dass sie in der gleichen Reihenfolge, wie in der Eingabefolge sortiert sind.

🡺 ganz normal sortieren 🡺 sind 2 Elemente gleich, relative Reihenfolge beibehalten

* Erfolgen mehrere Sortierungen 🡺 vorige Sortierung nicht zerstören
* Wenn Daten aus unbekannten Quellen kommen 🡺 man weiß nicht, ob sie sortiert sind 🡺 dann immer stabilen Algorithmus verwenden
* Kostet manchmal Leistung

# In-place

* Konstante Menge an Speicherplatz (kein zusätzlicher Speicherplatz benötigt)
* Von n unabhängig
* Ist auf den Heap bezogen (gilt nicht für Stack)
* Wird Sortierverfahren mit Rekursion verwendet 🡺 Stack kann überlaufen

# BubbleSort

Ein Element wird mit dem nächsten verglichen 🡺 ist dieses Element kleiner, werden die beiden vertauscht.

Bei jedem Durchgang der äußeren Schleife wird ein Element an die richtige Stelle gebracht 🡺 letztes Element des vorigen Durchlaufes nicht mehr beachten 🡺 kleine Elemente „blubbern“ vor

* In-place & stabil
* Worst-Case: n² (Äußere Schleife, Innere Schleife)
* Average-Case: n²
* Best-Case: n

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

# SelectionSort

Kleinstes Element suchen 🡺 mit dem ersten vertauschen 🡺 zum nächsten Element gehen 🡺 wieder kleinstes Element suchen 🡺 mit dem zweiten vertauschen 🡺 …

* Ein Bild, das Text enthält.

  Automatisch generierte BeschreibungIn-place & nicht stabil
* Worst-Case: n²
* Average-Case: n²
* Best-Case: n²

# InsertionSort

Alle Elemente von links nach rechts durchgehen 🡺 1. Stehen lassen 🡺 wo muss 2. hin  
Anschauen, ob es kleiner als das Linke ist, wenn ja 🡺 alle größeren Elemente um 1 verschieben, damit man Platz für das neue Element hat 🡺 Element an diese Stelle schreiben

* In-place & stabil
* Worst-Case: n²
* Average-Case: n²
* Best-Case: n

Ein Bild, das Text, Bildschirm, schwarz enthält.

Automatisch generierte Beschreibung

# Vergleich Sortieralgorithmen

Insertion- oder SelectionSort? 🡺 besser Insertionsort bei weniger Zahlen.

BubbleSort am schlechtesten, weil man die Zahlen so oft angreift.

SelectionSort besser 🡺 eine Zahl einmal angreifen und an richtige Stelle schieben

Es kommt immer auf zur Verfügung stehende **Leistung** und **Speicher** (anz der Elemente) an

---------------------------------------------- 2. Test --------------------------------------------------------

# Strings in Java

Java String Pool

String anlegen 🡺 es steht im Speicher

Bei dieser String + 1 🡺 es steht dann

Java merkt sich, dass es einen String schon gibt und legt daher eine Referenz dran 🡺 legt ihn also nicht neu an.

Muteable = an ort und stelle veränderbar

Inmuteable 🡺 Kopie wird angelegt und verändert.

# QuickSort

Welche Tiefe kann auch dem Stack maximal von einm Algorythums verbraucht werden?

N

* Rekursiv – trotzdem inplace
* Nicht stabil

**Aufwandsklasse**:

* n\*log(n) 🡺 Binärbaum entsteht (average und best)
* Worst-Case: n²

Zwei Pfeile definieren: L und R

Pivot Element markieren ganz hinten (willkürlich wählen)

Alle Zahlen die kleiner sind als dieses Element 🡺 links

Alle Zahlen die größer sind als dieses Element 🡺 rechts

🡺 pivot ist jetzt sortiert

Im Linken und Rechten Teil wieder ein pivot Element suchen (letztes…)

Wieder kleinere E 🡺 links & größere E 🡺 rechts

… irgendwann fertig