Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
WS 2013 / 2014

Vorlesung 8, Donnerstag 12. Dezember 2013
(Cache-Effizienz, Teile und Herrsche)

Junior-Prof. Dr. Olaf Ronneberger
Image Analysis Lab
Institut für Informatik
Universität Freiburg

[basiert auf der letztjährigen Veranstaltung von Prof. Dr. Hannah Bast, Lehrstuhl für Algorithmen und Datenstrukturen, mit Folien aus „Informatik 2" von Prof. Dr. Matthias Teschner]

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü6 (Priority Queue)
- Cache-Effizienz
  - Bisher haben wir zur Abschätzung der Laufzeit immer die Anzahl der Operationen gezählt
  - Heute sehen wir, dass das nicht immer ein gutes Maß ist
  - Insbesondere: Wie funktioniert ein Cache, Blockoperationen
  - Übungsblatt 8: Sortieren vs. Hashing
    - Vergleich #Operationen vs. #Block-Operationen
    - Vergleich der tatsächlichen Laufzeiten
  - Algorithmen-Entwurf mit „Teile und Herrsche“

Erfahrungen mit dem Ü6 (Priority Queues)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 11. Dezember
  - War einfach und schnell erledigt (mit std::string)
  - C++ templates (optional) haben sehr viel Zeit gefressen. Einführung dazu wäre prima. Hier in aller Kürze:
    - Template-Funktionen gehören in *.h oder *–inl.h Dateien (siehe Coding-styles), sonst kann der Compiler keinen Code für eine neue Klasse erzeugen.
    - Ein template< typename T> vor der Klassen-Deklaration oder vor der Methode reicht eigentlich aus, danach kann T so benutzt werden, als sei es z.B. mit einem #define T float definiert worden
    - Benutzung der template-Klasse dann wie die STL Container, z.B. PriorityQueueItem<float>

Cache-Effizienz

UNI FREIBURG

- Hintergrund
  - Bisher haben wir immer die Anzahl der Operationen gezählt
  - In der Annahme, dass das ein gutes Maß für die Laufzeit eines Algorithmus / Programms ist
  - Heute sehen wir Beispiele, wo das kein gutes Maß ist
- Beispiel
  - Wir addieren die Elemente eines Feldes der Größe n auf

    ... in der natürlichen Reihenfolge: a[1] + a[2] + a[3] + a[4] + a[5]

    ... in einer zufälligen Reihenfolge: a[2] + a[5] + a[1] + a[3] + a[4]

Cache-Effizienz Beispielprogramm

UNI FREIBURG

```
import java.text.DecimalFormat;
import java.util.Arrays;
import java.util.Random;

/**
 * Compare running times for adding entries of an array in different orders
 * (left to right versus random).
 */
public class ArraySumMain {

  /**
   * Main function.
   */
  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("Usage: java -jar ArraySumMain <array size>");
      System.exit(1);
    }
    int n = Integer.parseInt(args[0]);
```

Cache-Effizienz Beispielprogramm

UNI FREIBURG

```
DecimalFormat df = new DecimalFormat("###,###,###");

// Generate array with the numbers 1..n.
System.out.println();
System.out.print("Generating array with " + df.format(n) + " numbers ... ");
int[] array = new int[n];
for (int i = 0; i < n; i++) { array[i] = i + 1; }
System.out.println("done");
```

Cache-Effizienz Beispielprogramm

UNI FREIBURG

```java
// Generate two permutations: the trivial one 1,2,...,n, and a random one.
Random random = new Random(System.currentTimeMillis());
int[] order1 = new int[n];
for (int i = 0; i < n; i++) { order1[i] = i; }
int[] order2 = Arrays.copyOf(order1, n);
for (int i = 0; i < n; i++) {
  // Random int from i..n-1.
  int j = i + random.nextInt(n - i);
  // Swap order2[i] and order2[j].
  int tmp = order2[i];
  order2[i] = order2[j];
  order2[j] = tmp;
}
```

Cache-Effizienz Beispielprogramm

```
  // Now iterate over the first array and sum up the values.
  System.out.println();
  for (int run = 1; run <= 3; run++) {
    System.out.print("Summing over the array from left to right ... ");
    long startTime = System.currentTimeMillis();
    int sum = 0;
    for (int i = 0; i < n; i++) { sum += array[order1[i]]; }
    long endTime = System.currentTimeMillis();
    System.out.println("done in " + (endTime - startTime)
        + "ms (sum = " + sum + ")");
  }

  // Now iterate over the second array and sum up the values.
  // [...] (same as above but for order2)
 }
}
```

Ausgabe des Beispielprogrammes

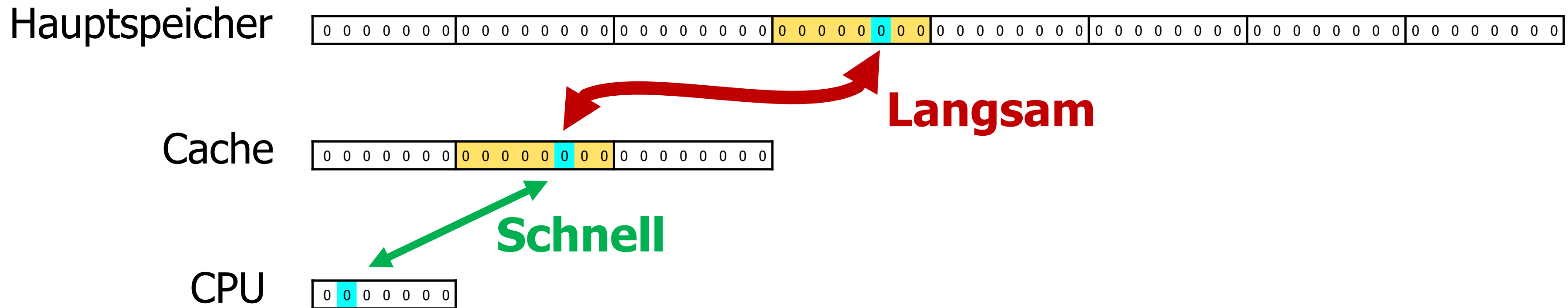
UNI FREIBURG

```
Generating array with 10.000.000 numbers ... done

Summing over the array from left to right ... done in 22ms (sum = -2004260032)
Summing over the array from left to right ... done in 22ms (sum = -2004260032)
Summing over the array from left to right ... done in 21ms (sum = -2004260032)

Summing over the array in random order ... done in 205ms (sum = -2004260032)
Summing over the array in random order ... done in 204ms (sum = -2004260032)
Summing over the array in random order ... done in 212ms (sum = -2004260032)
```

- Die Anzahl der Operationen ist für beide **identisch**
- Aber die Laufzeiten unterscheiden sich sehr (Faktor 10), warum?
- Offensichtlich sind die Kosten für einen Speicherzugriff sehr unterschiedlich

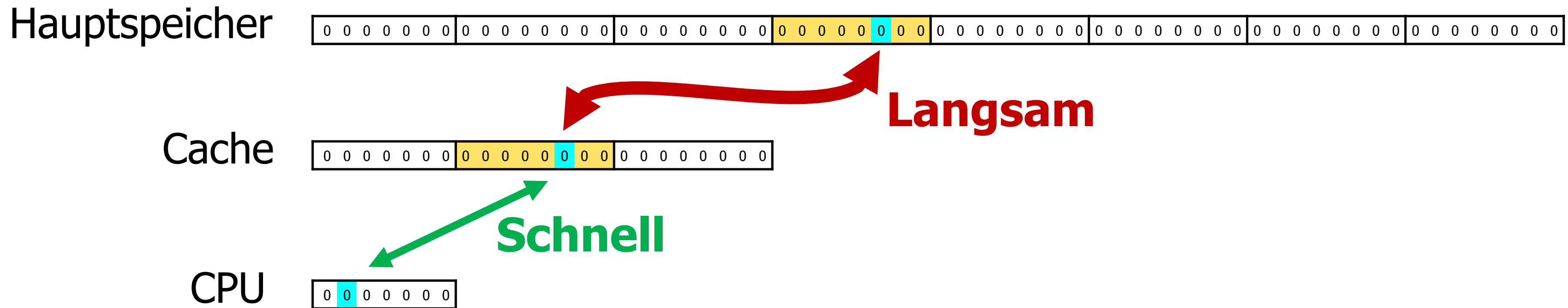
CPU Cache

UNI FREIBURG



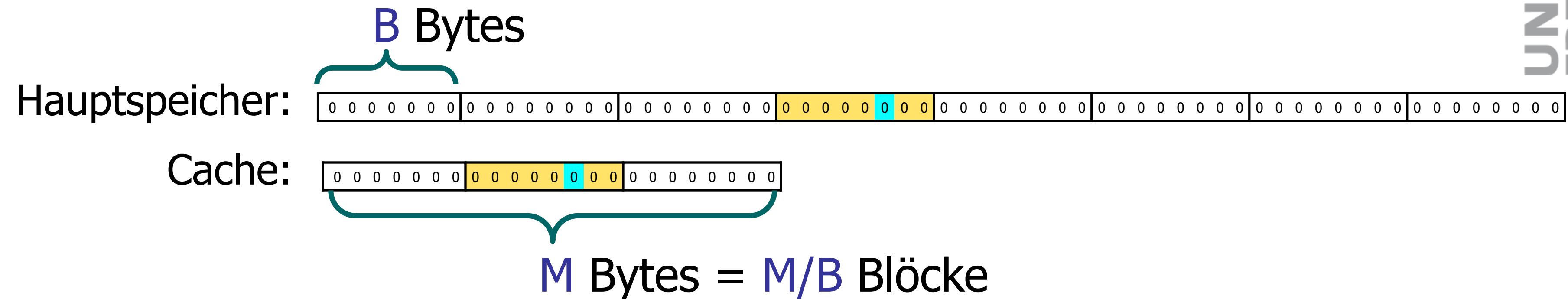
- Prinzip / Aufbau
  - Zugriff auf ein Byte im Hauptspeicher kostet ca. 100ns
  - Zugriff auf ein Byte im (L1-)Cache kostet ca. 1ns
  - Bei Zugriff auf ein oder mehrere Bytes im Hauptspeicher holt man gerade einen ganzen Block von ~ 100 Bytes in den Cache
  - Solange dieser Block im Cache ist, braucht man für Bytes aus diesem Block nicht mehr auf den Hauptspeicher zuzugreifen

CPU Cache



- Der Cache hat Platz für viele solcher Blöcke (die cache lines)
  - ein typischer L1-Cache ist ~ 100 Kilobytes groß
- Ist der Cache voll, wird einer der Blöcke entfernt
  - z.B. der least recently used (LRU) Block
  - das soll aber heute nicht das Thema sein

Block-Operationen Terminologie



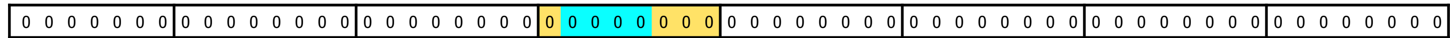
- Terminologie
  - Wir haben einen langsamen und einen schnellen Speicher
  - Der langsame Speicher ist in Blöcke der Größe B unterteilt
  - Der schnelle Speicher ist M groß (Platz für M/B Blöcke)
  - Stehen die Daten nicht im schnellen Speicher, wird der entsprechende Block in den schnellen Speicher geladen
  - Das Programm kann sich aussuchen, welche Blöcke im schnellen Speicher gehalten werden
  - Wir zählen nur die **Anzahl der Block-Operationen**

  In der Praxis hat man auch noch die Kosten für das Verwalten der Blöcke im schnellen Speicher, insbesondere welcher Block entfernt wird wenn der Speicher voll ist ... das ignorieren wir hier

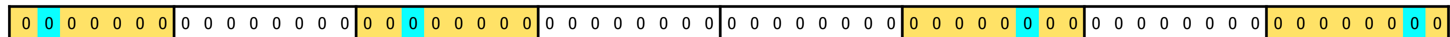
Block-Operationen Lokalität

Haupt-speicher:

Gute Lokalität



Schlechte Lokalität



- Für B Speicherzugriffe hat man also
  - im besten Fall nur 1 Block-Op. **"gute Lokalität"**
  - im schlechtesten Fall B Block-Op. **"schlechte Lokalität"**
- Die folgenden Variationen ...
  - ... machen nur einen (kleinen) konstanten Faktor in der Anzahl der Block-Operationen aus:
    - die genaue Aufteilung des langsamen Speichers in Blöcke
    - ob die Speichereinheit 1 Byte oder 4 Bytes oder 8 Bytes ist
- Man beachte:
  - Das Ganze wird erst interessant, wenn die Eingabe größer als M ist, sonst kann man einfach erstmal die gesamte Eingabe in den schnellen Speicher laden

UNI FREIBURG

Block-Operationen in der Praxis

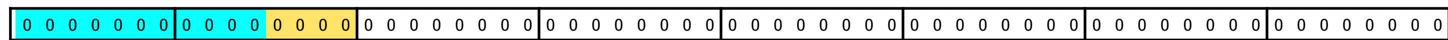
UNI FREIBURG

- Typische Werte (für einen Server)
  - CPU Cache: B = 128 Bytes, M = 6 x 32 KB (L1), 6 x 256 KB (L2)
  - Disk Cache: B = 64 Kilobytes, M = 64 GB
    - Die meisten Betriebssysteme benutzen alles was vom Hauptspeicher gerade nicht genutzt wird als Disk Cache
- Noch etwas Terminologie
  - Die Block-Operationen beim CPU Cache nennt man cache misses
  - Die Block-Operationen beim Disk Cache nennt man oft IOs
    - IO oder I/O = Input/Output
  - Man spricht auch von Cache-Effizienz und IO-Effizienz

Block-Operationen Beispiel 1

- Beispiel 1: unser ArraySumMain Programm
  - Wenn wir über die Elemente in der Reihenfolge 1, 2, 3, ... iterieren, ist die Anzahl Block-Operationen: ceil(n/B)

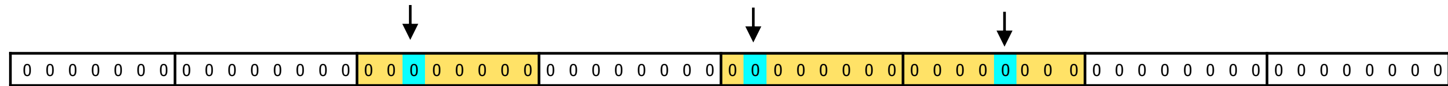
Speicherzugriffe:



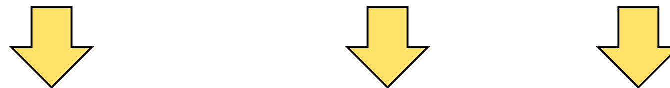
Block-Operationen:

  - Wenn wir über die Elemente in einer zufälligen Reihenfolge iterieren, ist die Anzahl Block-Op. im schlechtesten Fall: n

Speicherzugriffe:



Block-Operationen:



  - Das ist theoretisch ein Faktor von B Unterschied und das ist der Hauptgrund für den beobachteten Laufzeitunterschied

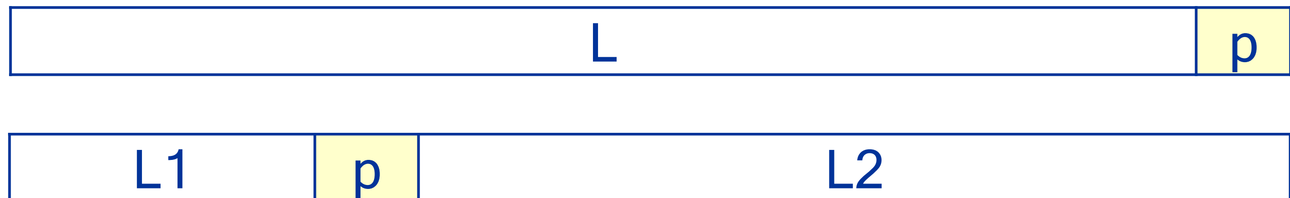
Block-Operationen Beispiel 1

UNI FREIBURG

- In der Praxis ist der Faktor deutlich $< B$
  – auch bei der zufälligen Reihenfolge wird pro Element auf 4 benachbarte Bytes (ein int) auf einmal zugegriffen
  – Außerdem wird, wenn nicht $n >> M$, das nächste Element manchmal zufällig schon im Cache Speicher stehen

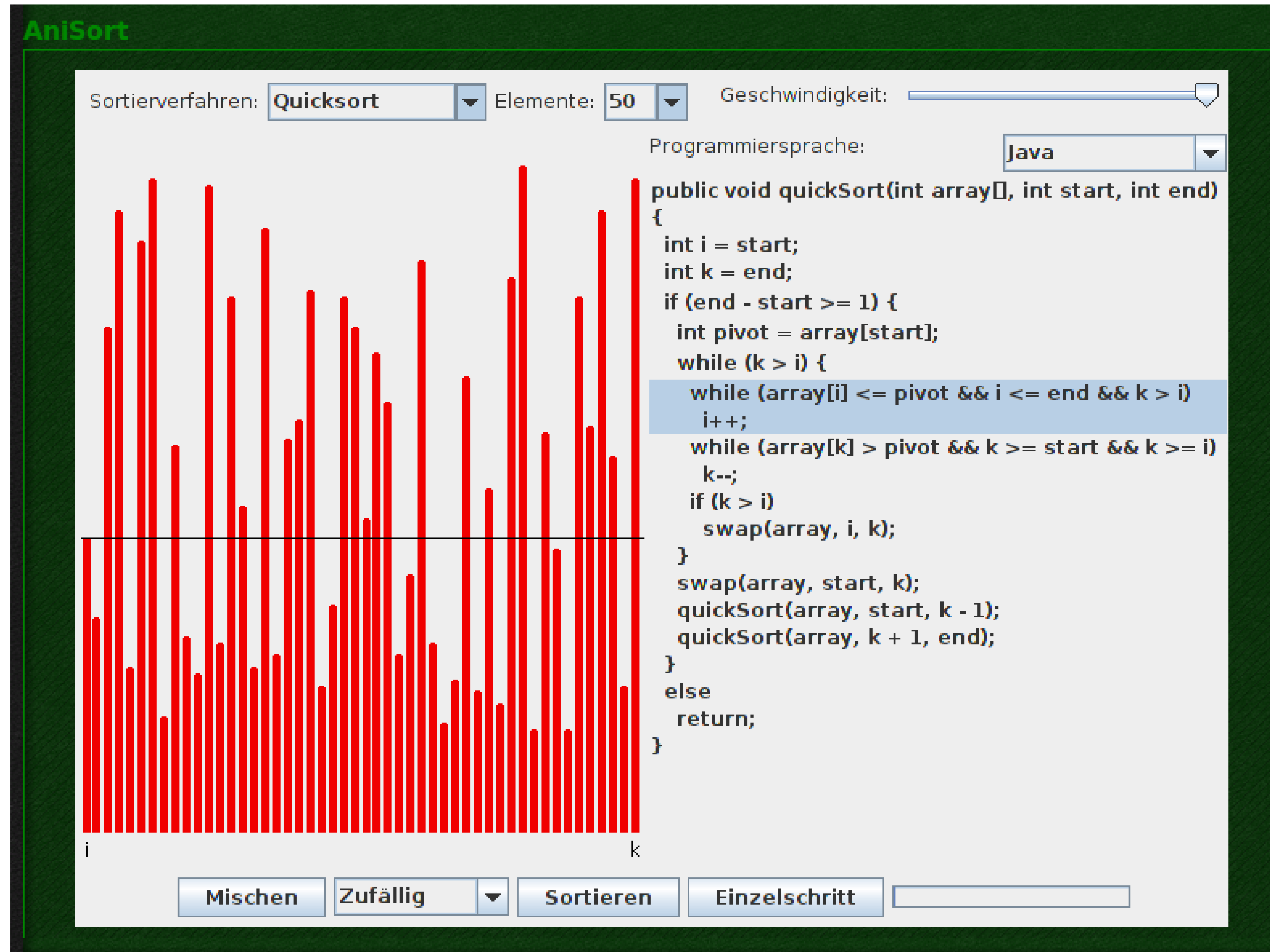
Beispiel 2: Quicksort

- Beispiel 2: Sortieren mit QuickSort
  - Vorweg kurz die Funktionsweise von QuickSort
  - Strategie: Teile und Herrsche – dazu später mehr
    - Teile das Eingabefeld in zwei Teile, so dass alle Elemente im linken Teil $\leq$ alle im rechten Teil sind
    - Zum Teilen wird jeweils irgendein Element genommen („Pivot"-Element), z.B. das erste oder das letzte



- Idealerweise sind die beiden Teile gleich groß
- Sortiere dann jeden der beiden Teile rekursiv

Einschub Quicksort-Algorithmus



http://siebn.de/index.php?page=anisort/anisort

UNI FREIBURG

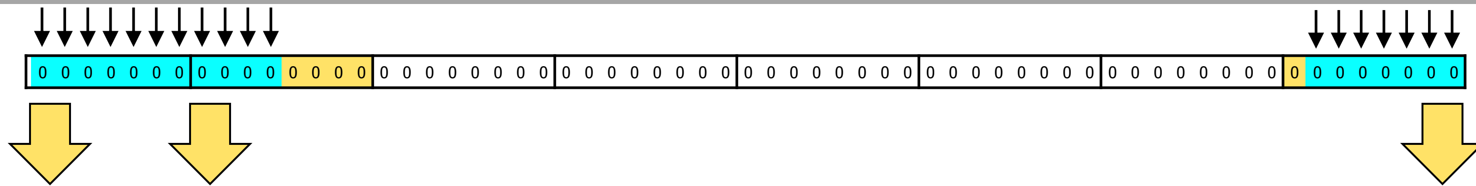
Anzahl gewöhnlicher Operationen bei Quicksort

UNI FREIBURG

- Sei T(n) die Laufzeit für die Eingabegröße n
- Annahmen
  - Felder werden immer perfekt in der Mitte getrennt
  - n sei eine Zweierpotenz und Rekursionstiefe $k = \log_2 n$

$$
\begin{aligned}
T(n) &\leq \underbrace{A \cdot n}_{\substack{\text{zum Teilen}\\ \text{in zwei Teile}}} + \underbrace{2 \cdot T(n/2)}_{\text{für die rek. Aufrufe}} \\
&\leq A \cdot n + 2 \cdot \Big(A \cdot n/2 + 2 \cdot T(n/4)\Big) \\
&= 2 \cdot A \cdot n + 4 \cdot T(n/4) \\
&\leq 3 \cdot A \cdot n + 8 \cdot T(n/8) \\
&\leq \cdots \\
&\leq k \cdot A \cdot n + 2^k \cdot T(1) \\
&= \log_2 n \cdot A \cdot n + n \cdot T(1) \\
&\leq \log_2 n \cdot A \cdot n + n \cdot A \quad \in \quad \mathcal{O}(n \log_2 n)
\end{aligned}
$$

Anzahl Block-Operationen bei Quicksort



– Sei IO(n) die Anzahl der Block-Operationen für die Eingabegröße n

– Annahmen wie auf der Folie zuvor, aber Rekursionstiefe
  $k = \log_2 (n/B)$ warum?

$$
\begin{aligned}
IO(n) &\leq A \cdot n/B \;+\; 2 \cdot IO(n/2) \\
&\leq A \cdot n/B \;+\; 2 \cdot \Big(A \cdot n/2B \;+\; 2 \cdot IO(n/4)\Big) \\
&= 2 \cdot A \cdot n/B \;+\; 4 \cdot IO(n/4) \\
&\leq 3 \cdot A \cdot n/B \;+\; 8 \cdot IO(n/8) \\
&\leq \cdots \\
&\leq k \cdot A \cdot n/B \;+\; 2^k \cdot IO(n/2^k) \\
&= \log_2(n/B) \cdot A \cdot n/B \;+\; n/B \cdot IO(B) \\
&\leq \log_2(n/B) \cdot A \cdot n/B \;+\; n/B \cdot A \qquad \in \quad \mathcal{O}\Big(\frac{n}{B} \log_2 \frac{n}{B}\Big)
\end{aligned}
$$

Literatur / Links

UNI FREIBURG

- Cache-Effizienz / IO-Effizienz
  - In Mehlhorn/Sanders:

    2 Introduction   2.2.1 External Memory
  - In Wikipedia

    http://en.wikipedia.org/wiki/Cache

    http://de.wikipedia.org/wiki/Cache

    (da wird das Prinzip eines Caches beschrieben, es gibt aber keinen separaten Artikel zur Cache-Effizienz bei Algorithmen)

Teile und Herrsche

UNI FREIBURG

- Prinzip
- Maximale Teilsumme
  - erste Lösungsansätze
  - Teile-und-Herrsche-Ansatz
  - Laufzeit
- Rekursionsgleichungen
  - Substitutionsmethode
  - Rekursionsbaum-Methode
  - Mastertheorem

Prinzip

- Teile das Gesamtproblem in kleinere Teilprobleme auf.
- Herrsche über die Teilprobleme durch rekursives Lösen. Wenn die Teilprobleme klein sind, löse sie direkt.
- Verbinde die Lösungen der Teilprobleme zur Lösung des Gesamtproblems.

- rekursive Anwendung des Algorithmus auf immer kleiner werdende Teilprobleme
- direkte Lösung eines hinreichend kleinen Teilproblems

Prinzip

UNI FREIBURG

- Methode V zur Lösung des Problems P der Größe n

- Methode V(P)

```
Methode V(P)
    wenn n<d
        dann
            löse das Problem P direkt;
        sonst
            teile P in mehrere Teilprobleme
            P1, P2, …, Pk mit k≥2;
            Methode V(P1); …; Methode V(Pk);
            verbinde die Teillösungen für
            P1, P2, …, Pk zur Gesamtlösung für P;
```

Eigenschaften

UNI FREIBURG

- kann bei konzeptionell schwierigen Problemen hilfreich sein
  - Lösung für den Trivialfall muss bekannt sein
  - Aufteilung in Teilprobleme muss möglich sein
  - Kombination der Teillösungen muss möglich sein
- kann effiziente Lösungen realisieren
  - wenn die Lösung des Trivialfalls in O(1) liegt, Aufteilung in Teilprobleme und Kombination der Teillösungen in O(n) liegt und die Zahl der Teilprobleme beschränkt ist, liegt der Algorithmus in O(n log n)
- für Parallelverarbeitung geeignet
  - Teilprobleme werden unabhängig voneinander verarbeitet

Implementierung

UNI FREIBURG

- Definition des Trivialfalls
  - Möglichst kleine direkt zu lösende Teilprobleme sind elegant und einfach.
  - Andererseits wird die Effizienz verbessert, wenn schon relativ große Teilprobleme direkt gelöst werden.
  - Rekursionstiefe sollte nicht zu groß werden.
- Aufteilung in Teilprobleme
  - Wahl der Anzahl der Teilprobleme und konkrete Aufteilung kann anspruchsvoll sein.
- Kombination der Teillösungen
  - typischerweise konzeptionell anspruchsvoll

Überblick

UNI FREIBURG

- Prinzip
- Maximale Teilsumme
  - erste Lösungsansätze
  - Teile-und-Herrsche-Ansatz
  - Laufzeit
- Rekursionsgleichungen
  - Substitutionsmethode
  - Rekursionsbaum-Methode
  - Mastertheorem

Maximale Teilsumme

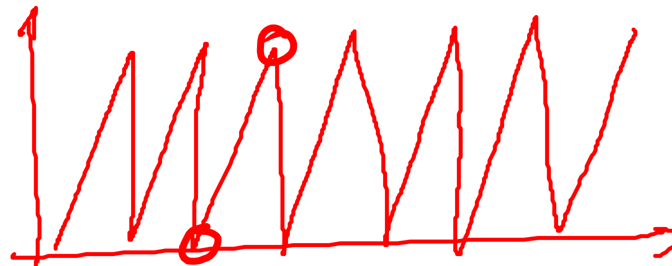
UNI FREIBURG

- Eingabe: Folge X von n ganzen Zahlen
- Ausgabe: Maximale Summe einer zusammenhängenden Teilfolge von X und deren Index-Grenzen

- Eingabe:

Index	0	1	2	3	4	5	6	7	8	9
Wert	31	-41	59	26	-53	58	97	-93	-23	84

- Ausgabe: 187, 2, 6



Anwendung

- Maximal möglicher Gewinn bei Kauf und Verkauf einer Aktie

Deutsche Telekom, 1997 - 2009

www.comdirect.de

Naive Lösung (Brute Force)

UNI FREIBURG

```
maxSubArray(X)
result.sum=X(0); result.u=0; result.o=0;
   for u=0 to länge(X)-1 do
      for o=u to länge(X)-1 do
      begin
         summe=0;
         for i=u to o do summe=summe+X(i);
         if result.sum<summe then
         begin
            result.sum = summe;
            result.u = u; result.o = o;
         end;
      end;
return result.sum, result.u, result.o;
```

Java-Implementierung

UNI FREIBURG

```
public class MaxSubArray {
   public static void main(String[] args) {
   }
   public Result maxSubArray(int[] X) {
      Result result = new Result();
      result.sum = X[0]; result.u = 0; result.o = 0;
      for (int u = 0; u < X.length; u++) {
         for (int o = u; o < X.length; o++) {
            int summe = 0;
            for (int i = u; i <= o; i++) {
               summe = summe + X[i];
               if (result.sum < summe) {
                  result.sum = summe; result.u = u; result.o = o;
               }
            }
         }
      }
      return result;
   }
}

class Result {
   int sum; int u; int o;
}
```

Laufzeit obere Schranke

UNI FREIBURG

```
for u=0 to länge(X)-1 do
    for o=u to länge(X)-1 do
    begin
        summe=0;
        for i=u to o do summe=summe+X(i);
        if result.sum<summe then
        begin
            result.sum = summe;
            result.u = u; result.o = o;
        end;
    end;
```

n Schleifendurchläufe → O(n)

maximal n Schleifendurchläufe → O(n)

maximal n Schleifendurchläufe → O(n)

O(1)

- drei verschachtelte Schleifen O(n) also insgesamt $O(n^3)$

Laufzeit untere Schranke

u	Additionen	o
	X	
n/3	n/3	n/3

- Es gibt wenigstens n/3 Werte für u, für die n/3 Werte von o durchlaufen werden, für die wiederum n/3 Additionsschritte in der innersten Schleife durchlaufen werden.
- Damit werden mindestens $T(n) = (n/3)^3 \in \Omega(n^3)$ Schritte benötigt.
- Aus $T(n) \in O(n^3)$ und $T(n) \in \Omega(n^3)$ folgt $T(n) \in \Theta(n^3)$
- Es ist schwer, das Problem schlechter zu lösen…

Effizientere Alternative

UNI FREIBURG

… aber leicht, es besser zu lösen.

- Bisher: Für jedes u und für jedes o wird $S_{u,o} = X(u) + X(u+1) + \ldots + X(o)$ berechnet

- Effizienter:
  Inkrementelle Aktualisierung statt Schleifendurchlauf

aus $S_{u,o+1} = X(u) + X(u+1) + \ldots + X(o) + X(o+1)$

und $S_{u,o} = X(u) + X(u+1) + \ldots + X(o)$

folgt $S_{u,o+1} = S_{u,o} + X(o+1)$ O(1) statt O(n)

Bessere Lösung

UNI FREIBURG

```
maxSubArray(X)
result.sum=X(0); result.u=0; result.o=0;
    for u=0 to länge(X)-1 do                              O(n)
    begin
        summe=0;
        for o=u to länge(X)-1 do                          O(n)
        begin
            summe=summe+X(o);
            if result.sum<summe then                      O(1)
            begin
                result.sum = summe;
                result.u = u; result.o = o;
            end;
        end;
    end;
return result.sum, result.u, result.o;
```

Gesamtaufwand $O(n^2)$

Java-Implementierung

```
public class MaxSubArray {
   public static void main(String[] args) {
   }
   public Result maxSubArray(int[] X) {
      Result result = new Result();
      result.sum = X[0]; result.u = 0; result.o = 0;
      for (int u = 0; u < X.length; u++) {
         int summe = 0;
         for (int o = u; o < X.length; o++) {
            summe = summe + X[o];
            if (result.sum < summe) {
               result.sum = summe;
               result.u = u;
               result.o = o;
            }
         }
      }
      return result;
   }
}

class Result {
   int sum; int u; int o;
}
```

UNI FREIBURG

Überblick

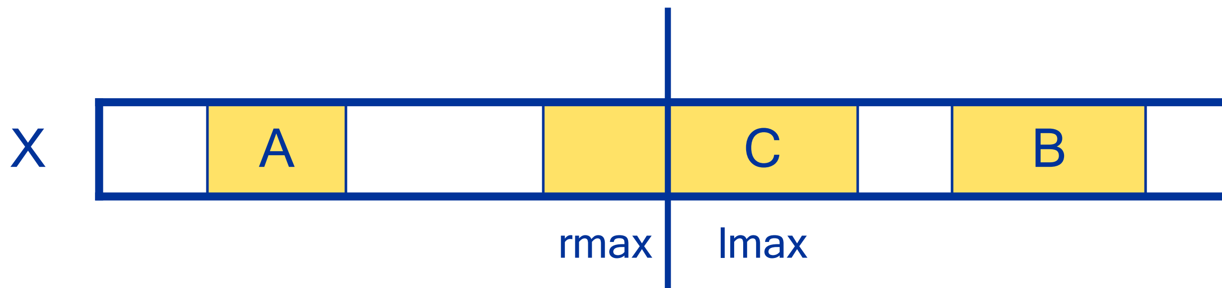
UNI FREIBURG

- Prinzip
- Maximale Teilsumme
  - erste Lösungsansätze
  - Teile-und-Herrsche-Ansatz
  - Laufzeit
- Rekursionsgleichungen
  - Substitutionsmethode
  - Rekursionsbaum-Methode
  - Mastertheorem

Idee

UNI FREIBURG

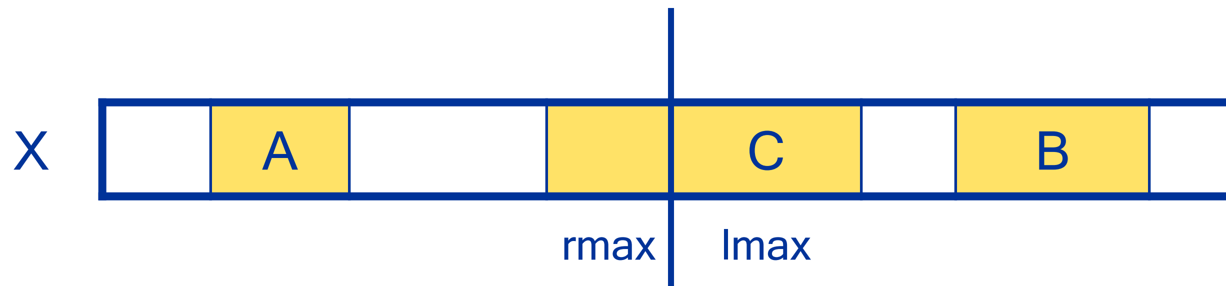
- Löse das Problem für die linke und rechte Hälfte von X.
- Setze die Teillösungen zur Gesamtlösung zusammen.



- Maximum liegt entweder in der linken Hälfte (A) oder in der rechten Hälfte (B)
- Maximum kann auch an der Grenze der Hälften liegen (C).
- Für C müssen rmax und lmax bestimmt werden.
- Gesamtlösung ist Maximum von A, B, C.

Prinzip

- Kleine Probleme werden direkt gelöst: $n=1 \Rightarrow Max = X(0)$
- Große Probleme werden in zwei Teilprobleme zerlegt und rekursiv gelöst. Teillösungen A und B werden geliefert.



- Um Teillösung C zu ermitteln, werden rmax und lmax für die Teilprobleme bestimmt.
- Die Gesamtlösung ergibt sich als das Maximum von A, B, C.