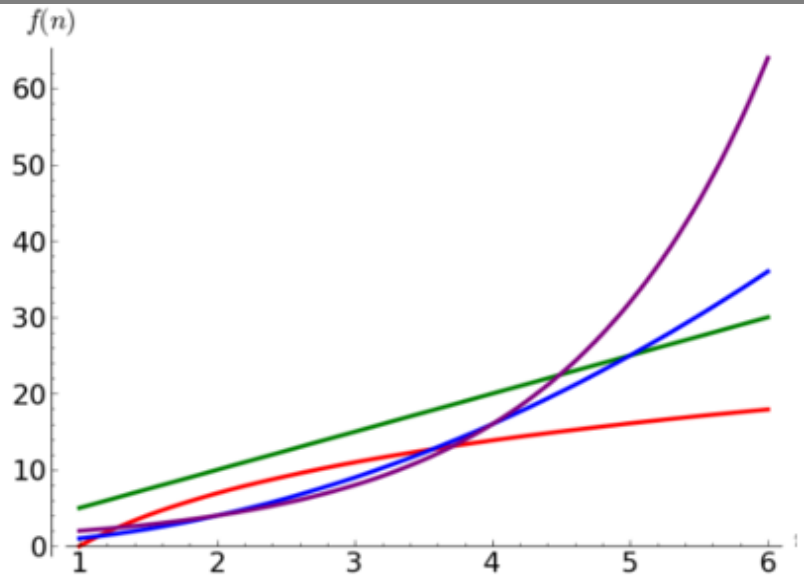
Karlsruher Institut für Technologie

# Tutorium Algorithmen 1

Simon Bischof (simon.bischof2@student.kit.edu) | 20. Mai 2013

INSTITUT FÜR THEORETISCHE INFORMATIK, PROF. SANDERS




```
if (num1 < 10) or (num2 < 10)
  return num1*num2
/* calculates the size of the numbers */
m = max(size(num1), size(num2))
low1, low2 = lower half of num1, num2
high1, high2 = higher half of num1, num2
/* 3 calls made to numbers approximately half the size */
z0 = karatsuba(low1,low2)
z1 = karatsuba((low1+high1),(low2+high2))
z2 = karatsuba(high1,high2)
return (z2*10^(m))+( (z1-z2-z0) *10^(m/2))+(z0)
```

# Zum Übungsblatt

Beachtet bei Hashing:

- Die Laufzeit O(1) für lookup/delete bei Hashing mit Verketten ist nur erwartet
- "Wähle eine zufällige Hashfunktion aus einer universellen Familie"
- Auf Satz 1 verweisen - insbesondere immer angeben, wie viele Slots die Hashtabelle hat
- vorne an Liste anfügen bei Hashing mit Verketten (warum?)
- zyklische Arrays brauchen kein Sentinel (implizit durch mod)

# Sortieren

- Gegeben sei eine Folge $s = \langle e_1, \ldots, e_n \rangle$ und eine lineare Ordnung $\leq$
- Gesucht: Folge $s' = \langle e_1\prime, \ldots, e_n\prime \rangle$, sodass $s\prime$ Permutation von $s$ und $e_1\prime \leq e_2\prime \leq \ldots \leq e_n\prime$

# Wichtige Eigenschaften von Sortieralgorithmen

- stabil: Elemente mit gleichem Wert behalten relative Reihenfolge bei
- inplace: nur "wenig" zusätzlicher Speicher nötig

# Insertionsort

- Idee: Der Anfang des Arrays ist schon sortiert
- füge erstes Element des unsortierten Teils dort richtig ein
- sortierter Teil wächst

# Pseudocode

```
1  Procedure insertionSort(a:Array[1... n] of Element)
2    for i:=2 to n do
3      invariant a[1] ≤ ... ≤ a[i − 1]
4      // move a[i] to the right place
5      e:=a[i]
6      if e<a[1] then //new minimum
7        for j:=i downto 2 do
8          a[j]:=a[j-1]
9        a[1]:=e
10     else //use a[1] as a sentinel
11       for (j:=i; a[j-1]>e; j--) a[j]:=a[j-1]
12       a[j]:=e
```

# Mergesort

```
1 Function mergeSort(⟨e_1,...,e_n⟩) : Sequence of Element
2   if n=1 then return ⟨e_1⟩
3   else return merge(   //merge siehe Blatt 2, A4(a)
4                 mergeSort(⟨e_1,...,e_⌊n/2⌋⟩),
5                 mergeSort(⟨e_⌊n/2⌋+1,...,e_n⟩))
```

# Untere Schranke

- Satz: Deterministische vergleichsbasierte Sortieralgorithmen brauchen $n \log n - O(n)$ Vergleiche im schlechtesten Fall.
- Dasselbe gilt auch für den average case
- Damit ist die worst- und average-case-Laufzeit von Mergesort optimal

# Untere Schranke

- Satz: Deterministische vergleichsbasierte Sortieralgorithmen brauchen $n \log n - O(n)$ Vergleiche im schlechtesten Fall.
- Dasselbe gilt auch für den average case
- Damit ist die worst- und average-case-Laufzeit von Mergesort optimal

# Untere Schranke

- Satz: Deterministische vergleichsbasierte Sortieralgorithmen brauchen $n \log n - O(n)$ Vergleiche im schlechtesten Fall.
- Dasselbe gilt auch für den average case
- Damit ist die worst- und average-case-Laufzeit von Mergesort optimal

# Quicksort

- Teile und herrsche wie bei Mergesort
- Aufwand jedoch vor der Rekursion
- Quicksort ist kompliziert?
- Trenne Idee von Implementierung

# Quicksort

- Teile und herrsche wie bei Mergesort
- Aufwand jedoch vor der Rekursion
- Quicksort ist kompliziert?
- Trenne Idee von Implementierung

# Quicksort

- Teile und herrsche wie bei Mergesort
- Aufwand jedoch vor der Rekursion
- Quicksort ist kompliziert?
- Trenne Idee von Implementierung

# Quicksort – Grundidee

```
1 qsort :: (Ord a) => [a] -> [a]
2 qsort []     = []
3 qsort (p:ps) = qsort (filter (<p) ps) ++
4                [p] ++
5                qsort (filter (>=p) ps)
```

# Laufzeit

- Best Case?

# Laufzeit

- Best Case? $O(n \log n)$

# Laufzeit

- Best Case? $O(n \log n)$
- Worst Case?

# Laufzeit

- Best Case? $O(n \log n)$
- Worst Case? $O(n^2)$

# Laufzeit

- Best Case? $O(n \log n)$
- Worst Case? $O(n^2)$
- Average Case (und erwartete Laufzeit bei zufälliger Pivotwahl) in $O(n \log n)$

# Quicksort – Arrayimplementierung

```
1 Procedure qSort(a:Array of Element; l, r : ℕ)
2   if l ≥ r then return
3   k:=pickPivotPos(a,l,r)
4   m:=partition(a,l,r,k)
5   qSort(a,l,m-1)
6   qSort(a,m+1,r)
```

# Quicksort – Partition

```
1  Function partition(a:Array of Element; l, r, k : ℕ)
2    p:=a[k]
3    swap(a[k],a[r])
4    i:=l
5    for j:=l to r-1 do
6      if a[j]≤ p then
7        swap (a[i],a[j])
8        i++
9    swap (a[i],a[r])
10   return i
```

# Verbesserungen

- Wenn Teilarray klein genug, Insertionsort verwenden
- Im Worst-Case O(n) Speicher nötig für (Rekursions-)Stack $\Rightarrow$ halbrekursive Implementierung (Rekursion nur auf kleinerem Teil)

# Verbesserungen

- Wenn Teilarray klein genug, Insertionsort verwenden
- Im Worst-Case O(n) Speicher nötig für (Rekursions-)Stack $\Rightarrow$ halbrekursive Implementierung (Rekursion nur auf kleinerem Teil)

# Kreativaufgabe

Gegeben sei ein Array mit n verschiedenen Elementen (unsortiert, aber mit Ordnung) und eine Medianfunktion, die für ein (Teil-)Array mit m Elementen den Median deterministisch in O(m) berechnet.

- Finde einen Algorithmus, der das $\frac{1}{3}$-Perzentil deterministisch in O(n) berechnet.
- Finde einen Algorithmus, der die $\frac{1}{3^{k-1}}, \frac{1}{3^{k-2}}, \ldots, \frac{1}{3^2}, \frac{1}{3}$-Perzentile deterministisch in O(n) berechnet. (Nicht in O(nk)!)
- Das Ganze geht inplace.