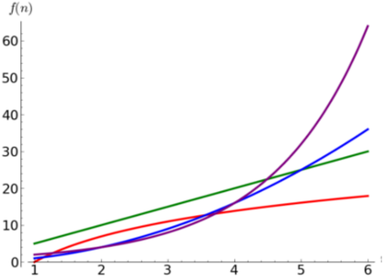
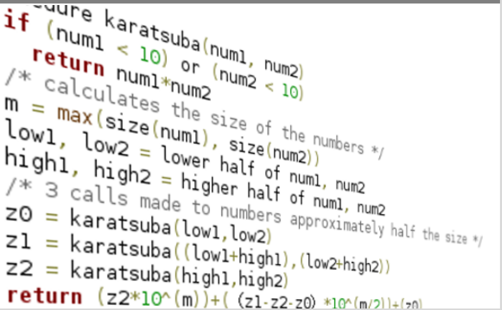
Karlsruher Institut für Technologie

Tutorium Algorithmen 1

Simon Bischof (simon.bischof2@student.kit.edu) | 21. Mai 2013

Institut für theoretische Informatik, Prof. Sanders





Mittsemesterklausur

- 3.6.2013, Audimax, 15:45-17:15
- 10-20% der Übungspunkte

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 $s' = \langle e'_1, \ldots, e'_n \rangle$, sodass s' Permutation von s und $e'_1 \leq e'_2 \leq \ldots \leq e'_n$

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

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

Mergesort

Function mergeSort($\langle e_1, \ldots, e_n \rangle$) : Sequence of Element
  if n=1 **then return** $\langle e_1 \rangle$
  else return merge(//merge siehe Blatt 2, A4(a)
        mergeSort($\langle e_1, \ldots, e_{\lfloor \frac{n}{2} \rfloor} \rangle$),
        mergeSort($\langle e_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, e_n \rangle$))

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

```haskell
qsort :: (Ord a) => [a] -> [a]
qsort []     = []
qsort (p:ps) = qsort (filter (<p) ps) ++
               [p] ++
               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

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

Quicksort – Partition

```
Function partition(a:Array of Element; l, r, k : ℕ)
  p:=a[k]
  swap(a[k],a[r])
  i:=l
  for j:=l to r-1 do
    if a[j]≤ p then
      swap (a[i],a[j])
      i++
  swap (a[i],a[r])
  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 $\frac{1}{3^{k-1}}, \frac{1}{3^{k-2}}, \dots, \frac{1}{3^2}, \frac{1}{3}$-Perzentile deterministisch in O(n) berechnet. (Nicht in O(nk)!)
- Das Ganze geht inplace.