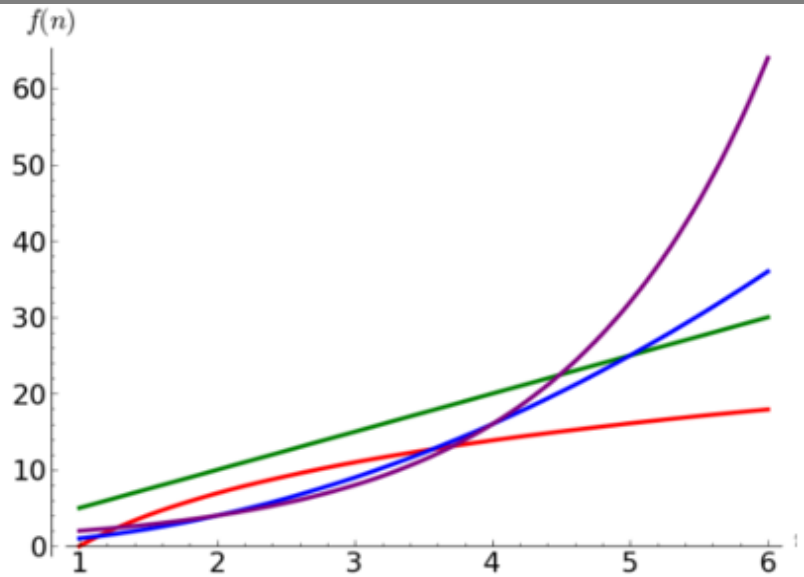
Karlsruher Institut für Technologie

Tutorium Algorithmen 1

Simon Bischof (simon.bischof2@student.kit.edu) | 28. 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)
```

Mittsemesterklausur

- Nächsten Montag (statt VL)
- Pünktlich sein!
- Hilfsmittel: 1 DIN A4 einseitig handbeschriebener Zettel
- Unterschrift für selbständige Anfertigung
- Tutoriumsnummer auf die Klausur schreiben

Zum Übungsblatt

- Hashing: bei Verketten braucht Initialisierung $\Theta(|t|)$
- Hashing: Hinweise aus letztem Tut beachten!
- Wenn ihr eine Prozedur in Pseudocode hinschreibt: Name und Parameter angeben
- Algorithmen kann man auch sprachlich angeben (falls nicht anders verlangt)
- Bei Aufgabe 3 hätte Mergesort genügt
- A3: bei Mergesort minimal 4 oder 5 Vergleiche je nach Implementierung
- Quicksort aus VL verwenden

Auswahl

- Gegeben: Folge s, Zahl $k \in \mathbb{N}$
- Gesuckt: k.-kleinstes Element von s
- Spezialfall: Median ($k = \lceil \frac{|s|}{2} \rceil$) oder allg. Quantile

Auswahl

- Gegeben: Folge s, Zahl $k \in \mathbb{N}$
- Gesuckt: k.-kleinstes Element von s
- Spezialfall: Median ($k = \lceil \frac{|s|}{2} \rceil$) oder allg. Quantile

Quickselect (Laufzeit erwartet linear)

```
1  Function select(s : Sequence of Element;
2                  k : ℕ) : Element
3    assert |s| ≥ k
4    pick p ∈ s uniformly at random //pivot key
5    a := ⟨e ∈ s : e < p⟩
6    if |a| ≥ k then return select(a, k)
7    b := ⟨e ∈ s : e = p⟩
8    if |a| + |b| ≥ k then return p
9    c := ⟨e ∈ s : e > p⟩
10   return select(c, k − |a| − |b|)
```

Bucketsort

```
1 Procedure KSort(s : Sequence of Element)
2   b := ⟨⟨⟩, ⟨⟩, ..., ⟨⟩⟩
3         : Array[0..K − 1] of Sequence of Element
4   foreach e ∈ s do
5     b[key(e)].pushBack(e)
6   s := concatenation of b[0], ..., b[K − 1]
```

Bucketsort

- Laufzeit $O(n + K)$
- Arrayimplementierung möglich (siehe VL)
- Ist stabil

LSD-Radixsort

- sortiere Zahlen mit d Stellen
- d-mal Bucketsort nach den Stellen $0,1,\ldots,d-1$ (von hinten)
- Laufzeit $O(d(n + K))$

Bucketsort

- Laufzeit $O(n + K)$
- Arrayimplementierung möglich (siehe VL)
- Ist stabil

LSD-Radixsort

- sortiere Zahlen mit d Stellen
- d-mal Bucketsort nach den Stellen $0,1,\ldots,d-1$ (von hinten)
- Laufzeit $O(d(n + K))$

Prioritätslisten

Prioritätslisten

- Gegeben: Elemente mit Schlüsseln
- Verwalte Menge M von Elementen mit folgenden Operationen:
- insert(e): $M := M \cup \{e\}$
- deleteMin: return and remove min M

Prioritätslisten

- Gegeben: Elemente mit Schlüsseln
- Verwalte Menge M von Elementen mit folgenden Operationen:
- insert(e): $M := M \cup \{e\}$
- deleteMin: return and remove min M

Kreativaufgabe

Gegeben sei ein Universum von Elementen die in einer Prioritätsliste verwaltet werden sollen. Die Wichtigkeit eines Elements hägt nur von seinem Schlüssel ab, wobei es nur k verschiedene Schlüssel geben soll. Man kennt den Algorithmus, der die Prioritätsliste verwenden soll, so gut, dass man weiß, dass alle Einfügungen eine geringere Priorität als das zuletzt entnommene Element haben.
Finden Sie eine Datenstruktur, in die neue Elemente in O(1) eingefügt werden können, und die Entnahme armortisiert in $O(\frac{k}{n} + 1)$ funktioniert. n sei dabei die Gesamtzahl jemals eingefügter Elemente.
Interessant insbesondere für n > k.

(Min-)Heaps

- Heap-Eigenschaft : Bäume (oder Wälder) mit $\forall v : \mathit{parent}(v) \leq v$
- Binärer Heap: Heap, Binärbaum, Höhe $\lfloor \log n \rfloor$, fehlende Blätter rechts unten
- Minimum = Wurzel

(Min-)Heaps

- Heap-Eigenschaft : Bäume (oder Wälder) mit $\forall v : \mathit{parent}(v) \leq v$
- Binärer Heap: Heap, Binärbaum, Höhe $\lfloor \log n \rfloor$, fehlende Blätter rechts unten
- Minimum = Wurzel

(Min-)Heaps

- Heap-Eigenschaft : Bäume (oder Wälder) mit $\forall v : \mathit{parent}(v) \leq v$
- Binärer Heap: Heap, Binärbaum, Höhe $\lfloor \log n \rfloor$, fehlende Blätter rechts unten
- Minimum = Wurzel

Implizite Baum-Repräsentation

- Array h[1..n]
- Schicht für Schicht
- parent(j)=$\lfloor \frac{j}{2} \rfloor$
- linkes Kind(j)=$2j$
- rechtes Kind(j)=$2j + 1$

Implizite Baum-Repräsentation

- Array h[1..n]
- Schicht für Schicht
- parent(j)=$\lfloor \frac{j}{2} \rfloor$
- linkes Kind(j)=$2j$
- rechtes Kind(j)=$2j + 1$

Pseudocode

```
1 Procedure insert(e:Element)
2   n++; h[n]:=e
3   siftUp(n) //siehe VL
4
5 Function deleteMin : Element
6   result = h[1] : Element
7   h[1] := h[n]; n--
8   siftDown(1) //siehe VL
9   return result
```