Tutorium 6: (Ganzzahliges-) Sortieren, Heaps

Matthias Schimek | 6. Juni 2017

TUTORIUM ZUR VORLESUNG ALGORITHMEN I IM SS17

Gliederung

1. Vergleichsbasiertes Sortieren
2. Ganzzahliges Sortieren
3. Prioritätslisten

Quicksort

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
  if $|s| \leq 1$ **then return** s
  pick "some" $p \in s$
  $a := \langle e \in s : e < p \rangle$
  $b := \langle e \in s : e = p \rangle$
  $c := \langle e \in s : e > p \rangle$
  return concatenation of quickSort(a), b, and quickSort(c) [1]

- average case: $\mathcal{O}(n \log n)$
- worst case: $\mathcal{O}(n^2)$
- man kann optimieren

[1] Folien 'Algorithmen I' SS 2017, KIT

Quicksort

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
  if $|s| \leq 1$ **then return** s
  pick "some" $p \in s$
  $a := \langle e \in s : e < p \rangle$
  $b := \langle e \in s : e = p \rangle$
  $c := \langle e \in s : e > p \rangle$
  return concatenation of quickSort(a), b, and quickSort(c) [1]

- average case: $\mathcal{O}(n \log n)$
- worst case: $\mathcal{O}(n^2)$
- man kann optimieren

[1] Folien 'Algorithmen I' SS 2017, KIT

Quicksort - Aufgabe

Sortiert das Array $[18, 9, 5, 4, 1]$ mit Quicksort.

Pivotwahl:

- pivot = 1. Element
- pivot = Median der Liste

Quicksort - inplace

Function qsort(A : **array of** $\mathbb{N}$, $l, r : \mathbb{N}$)
    if $l \geq r$ **then**
        return
    end
    $k :=$ pickPivot(A, l, r)
    $m :=$ partition(A, l, r, k)
    qsort$(A, l, m - 1)$
    qsort$(A, m + 1, r)$

Quicksort - inplace

Function partition(a : **Array of** Element; $\ell, r, k : \mathbb{N}$)

$p := a[k]$ // pivot

swap$(a[k], a[r])$

$i := \ell$

for $j := \ell$ **to** $r - 1$ **do**

invariant

ℓ	i	j	r
$\leq p$	$> p$	?	p

if $a[j] \leq p$ **then**

swap$(a[i], a[j])$

$i{+}{+}$

assert

ℓ	i	r
$\leq p$	$> p$	p

swap$(a[i], a[r])$

assert

ℓ	i	r
$\leq p$	p	$> p$

return i

[2]

[2] Folien 'Algorithmen I' SS 2017, KIT

Quicksort - halbrekursive Implementierung

Lösung des Problems : halbrekursive Implementierung

```
Function halbrekursiv(A : array of ℕ , l, r : ℕ)
  while r − l + 1 > 0 do
    k := pickPivot(A,l,r)
    m := partition(A,l,r,k)
    if m < (l + r)/2 then
                              // Rekursion auf kleinerer Hälfte
      halbrekursiv(A, l, m − 1))
      l := m + 1
    else
      halbrekursiv(A,m+1,r)
      r := m − 1
    end
  end
```

Quicksort - Aufgabe

Implementiere eine vollständig iterative Variante von Quicksort

Tipps:

- Verwende eigenen Stack
- mittels *push*()/*pop*() können Daten auf den Stack gelegt/abgerufen werden
- *pickPivot*() zur Pivotwahl
- *partition*() zur 'inplace' Partitionierung der Eingabe

Quicksort - Aufgabe

Karlsruhe Institute of Technology

Programmgerüst

```
Function qsort_iterativ(A : array of ℕ , l, r : ℕ)
    assert l < r
    stack.initialize()
    do
        while r − l + 1 > 0 do
            k := pickPivot(A,l,r)
            ...
        end
        ...
    while ¬ stack.empty()
```

Quicksort - Aufgabe

Programmgerüst

```
Function qsort_iterativ(A : array of ℕ , l, r : ℕ)
    assert l < r
    stack.initialize()
    do
        while r − l + 1 > 0 do
            k := pickPivot(A,l,r)
            m := partition(A,l,r,k)
            stack.push (r)
            r := m − 1
        end
        ...
    while ¬ stack.empty()
```

Quicksort - Aufgabe

Programmgerüst

```
Function qsort_iterativ(A : array of ℕ , l, r : ℕ)
    assert l < r
    stack.initialize()
    do
        while r − l + 1 > 0 do
            k := pickPivot(A,l,r)
            m := partition(A,l,r,k)
            stack.push (r)
            r := m − 1
        end
        l := r + 1
        r := stack.pop()
    while ¬ stack.empty()
```

Quicksort - Mergesort

Mergesort

- deterministisch in $\mathcal{O}(n \log n)$ Zeit
- stabil
- nicht inplace

Quicksort

- average case in $\mathcal{O}(n \log n)$ Zeit
- worst-case in $\mathcal{O}(n^2)$ Zeit oder Pivot besser wählen (Median)
- funktioniert inplace

Quicksort - Mergesort

Mergesort

- deterministisch in $\mathcal{O}(n \log n)$ Zeit
- stabil
- nicht inplace

Quicksort

- average case in $\mathcal{O}(n \log n)$ Zeit
- worst-case in $\mathcal{O}(n^2)$ Zeit oder Pivot besser wählen (Median)
- funktioniert inplace

Quicksort - Anwendung: Quickselect

Definition Rang
Rang k eines Elements e in Folge S ist seine Position in *sortiert(S)*.

Beispiel Rang von '5' in $[4, 55, 3, 1, 8, 5]$ ist 4,
da *sortiert*$([4, 55, 3, 1, 8, 5]) = [1, 3, 4, 5, 8, 55]$

Idee: Verwende Vorgehen von Quicksort zur Berechnung des Rangs

Quicksort - Anwendung: Quickselect

Karlsruhe Institute of Technology

Definition Rang
Rang *k* eines Elements *e* in Folge *S* ist seine Position in *sortiert(S)*.

Beispiel Rang von '5' in $[4, 55, 3, 1, 8, 5]$ ist 4,
da *sortiert(*$[4, 55, 3, 1, 8, 5]$*)* = $[1, 3, 4, 5, 8, 55]$

Idee: Verwende Vorgehen von Quicksort zur Berechnung des Rangs

Quicksort - Anwendung: Quickselect

Definition Rang
Rang *k* eines Elements *e* in Folge *S* ist seine Position in *sortiert(S)*.

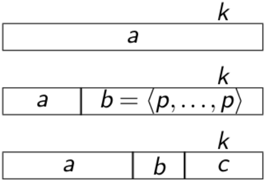
Beispiel Rang von '5' in $[4, 55, 3, 1, 8, 5]$ ist 4,
da *sortiert(*$[4, 55, 3, 1, 8, 5]$*)* = $[1, 3, 4, 5, 8, 55]$

Idee: Verwende Vorgehen von Quicksort zur Berechnung des Rangs

Quickselect

Karlsruhe Institute of Technology

Function select(s : Sequence **of** Element; k : $\mathbb{N}$) : Element
  assert $|s| \geq k$
  pick $p \in s$ uniformly at random // pivot key
  a := $\langle e \in s : e < p \rangle$
  if $|a| \geq k$ **then return** select(a, k) //
  b := $\langle e \in s : e = p \rangle$
  if $|a| + |b| \geq k$ **then return** p //
  c := $\langle e \in s : e > p \rangle$
  return select$(c, k - |a| - |b|)$ //



[3]

[3] Folien 'Algorithmen I' SS 2017, KIT

Ganzzahliges Sortieren

Satz Vergleichsbasiertes Sortieren benötigt $\Omega(n \log n)$ Vergleiche

- Geht es anders?
- $\Rightarrow$ Ganzzahliges Sortieren

Ganzzahliges Sortieren

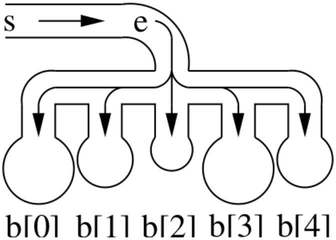
Satz Vergleichsbasiertes Sortieren benötigt $\Omega(n \log n)$ Vergleiche

- Geht es anders?
- $\Rightarrow$ Ganzzahliges Sortieren

Bucketsort

Procedure KSort(s : Sequence **of** Element)
 b=$\langle\langle\rangle,\ldots,\langle\rangle\rangle$: **Array** $[0..K-1]$ **of** Sequence **of** Element
 foreach $e \in s$ **do** $b[\mathrm{key}(e)]$.pushBack(e)
 s := concatenation of $b[0],\ldots,b[K-1]$

Zeit: $O(n+K)$



4

[4] Folien 'Algorithmen I' SS 2017, KIT

Radixsort

Least-Significant-Digit-Radixsort

- Idee Bucketsort auf einzelne Stellen nacheinander anwenden
- Vorraussetzung: Stabilität von Bucketsort
- iterativ
- Laufzeit bei d-stelligem Schlüssel?

Most-Significant-Digit-Radixsort

- Idee Bucketsort auf MSB anwenden
- rekursiv weitersortieren

Radixsort

Least-Significant-Digit-Radixsort

- Idee Bucketsort auf einzelne Stellen nacheinander anwenden
- Vorraussetzung: Stabilität von Bucketsort
- iterativ
- Laufzeit bei d-stelligem Schlüssel?

Most-Significant-Digit-Radixsort

- Idee Bucketsort auf MSB anwenden
- rekursiv weitersortieren

LSD-Radixsort - Aufgabe

Gegeben sei ein Array von $n = 2^k, k \in \mathbb{N}$ Zahlen im Interval $0 \ldots n^3 - 1$ gegeben. Wie können diese mit LSD-Radixsort in $\mathcal{O}(n)$ Zeit sortiert werden?

Prioritätsliste

Prioritätsliste

- Verwaltet Menge M von Element/Schlüssel Paaren (k, e)
- $insert()$: Einfügen von neuen Elementen in M
- $deleteMin()$: Löschen des Elements mit kleinestem Schlüssel

Binary Heaps

Binary Heaps

- (Binärer) Baum mit Eigenschaft $\forall v: \ \text{parent}(v) \leq v$
- Einfügen/Löschen: Änderungen nur entlang eines Wurzel-Blatt-Pfades
- Vollständiger Binärbaum hat Höhe $\log n$

  $\Rightarrow$ *insert*/*delete* in $\mathcal{O}(\log n)$

Aufgabe

Entwerfe einen (vergleichsbasierten) Sortieralgorithmus der in Zeit $\mathcal{O}(n \log n)$ läuft.

Zusammenfassung

- Quicksort
- Quickselect
- Ganzzahliges Sortieren
- Binary Heaps