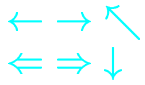
Algorithmen und Datenstrukturen (Th. Ottmann und P. Widmayer)

Folien: Suchverfahren

Autor: Stefan Edelkamp / Sven Schuierer

Institut für Informatik
Georges-Köhler-Allee
Albert-Ludwigs-Universität Freiburg

1 Überblick



Überblick

Problemstellung

Binäre Suche

Fibonacci-Suche

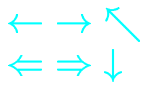
Exponentielle Suche

Interpolationssuche

Das Auswahlproblem

Selbstanordnende lineare Listen

2 Problemstellung



Problem: Gegeben Folge $F = (a_1, \ldots, a_n)$. Finde das Element mit Schlüssel k.

Rahmen:

```
class SearchAlgorithm {
  static int searchb(Orderable A[], Orderable k) {
    return search(A,k);
  }
}
```

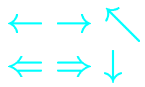
Einfachstes Verfahren: Sequentielle, lineare Suche

```
class SequentialSearch extends SearchAlgorithm {
  public static int search(Orderable A[],Orderable k){
    /* Durchsucht A[1], .., A[n] nach Element k und
       liefert den Index i mit A[i] = k; -1 sonst  */
    A[0] = k;       // Stopper
    int i = A.length;
    do i--; while (!A[i].equal(k));
    if (i !=  0)     // A[i] ist gesuchtes Element
      return i;
    else  // es gibt kein Element mit Schluessel k
      return -1;
  }
}
```

Analyse:

- schlechtester Fall: $n + 1$
- im Mittel: $\frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$

3 Binäre Suche



Klasse

```
class BinarySearch extends SearchAlgorithm
```

Hauptroutine

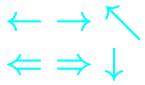
```
public static int search(Orderable A[],Orderable k){
  /* Durchsucht A[1], .., A[n] nach Element k und
     liefert den groessten Index i >= 1 mit
     A[i] <= k; 0 sonst */
  int n = A.length;
  return search(A, 1, n, k);
}
```

Rekursiver Aufruf

```
public static int search
  (Orderable A[], int l, int r, Orderable k){
  /* Durchsucht A[1], .., A[n] nach Element k
     und liefert den groessten Index
     l <= i <= r mit A[i] <= k; l-1 sonst */
  if (l > r)  // Suche erfolglos
    return l-1;

  int m = (l + r) / 2;
  if (k.less(A[m]))
    return search(A, l, m - 1, k);
  if (k.greater(A[m]))
    return search(A, m + 1, r, k);
  else  // A[m] = k
    return m;
}
```

Binäre Suche ohne Rekursion

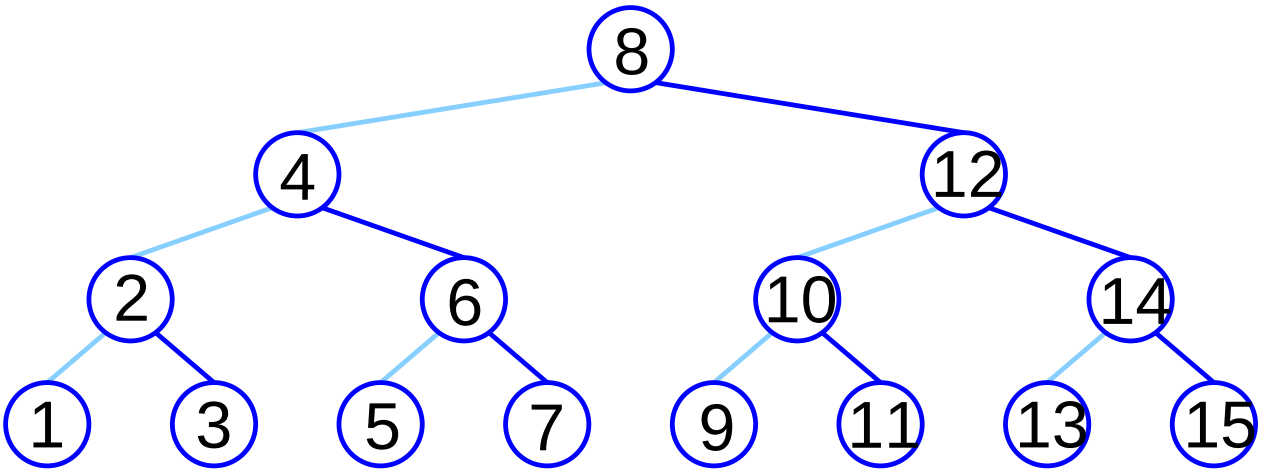


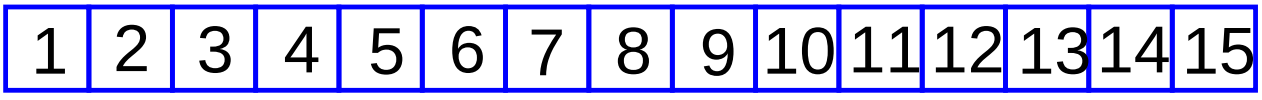
```
public static int search(Orderable A[],Orderable k){
  int n = A.length, l = 1, r = n;
  while (l <= r) {
    int m = (l + r) / 2;
    if (k.less(A[m])) { r = m - 1; }
    else if (k.greater(A[m])) { l = m + 1; }
    else return m;
  }
  return l-1;
}
```

Annahme: Binärer Vergl.-Operator mit 3 Ausgängen

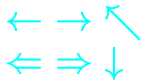
Worst Case ($n = 2^k - 1$): bei $k = \log(n + 1)$ Vergl.

Average Case ($n = 2^k - 1$):





Analyse



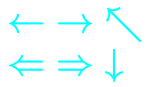
Auswertung: $\sum_{i=1}^{k} i2^{i-1}$

$$
\begin{array}{rcl}
1 \cdot 2^{k-1} & = & 2^k - 2^{k-1} \\
\vdots & = & \vdots \\
1 \cdot 2^1 + \ldots + 1 \cdot 2^{k-1} & = & 2^k - 2 \\
1 \cdot 2^0 + 1 \cdot 2^1 + \ldots + 1 \cdot 2^{k-1} & = & 2^k - 1 \\
\hline
 & = & k2^k - 2^k + 1
\end{array}
$$

Erwartungswert:

$$
\begin{aligned}
E &= (\sum_{i=1}^{k} i2^{i-1})/n \\
&= (k2^k - 2^k + 1)/n \\
&= ((n+1)\log(n+1))/n - (n+1)/n + 1/n \\
&= (n+1)\log(n+1)/n \approx \log(n+1) - 1
\end{aligned}
$$

4 Fibonacci-Suche



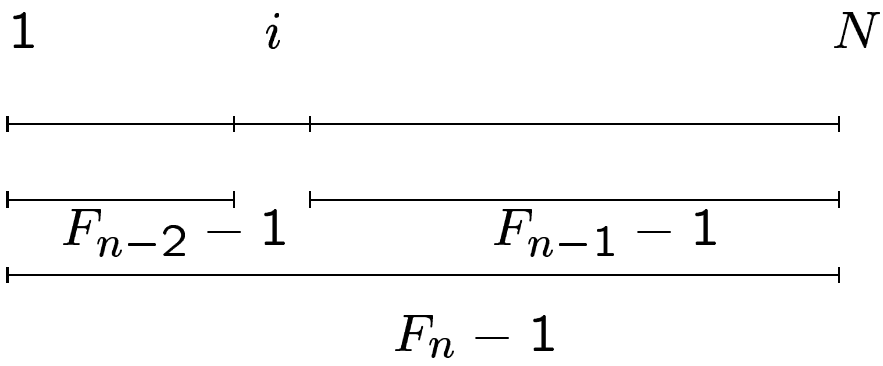
Erinnerung:

$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$ für $(n \geq 2)$.

Verfahren:

Vergleiche den Schlüssel an $i = F_{n-2}$ mit k.

- $A[i].key > k$: Durchsuche linke $F_{n-2} - 1$ Elemente
- $A[i].key < k$: Durchsuche rechte $F_{n-1} - 1$ Elemente

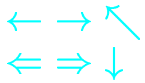


Analyse: Durchsuchen von $F_n - 1$ Elementen in max. n Schlüsselvergleichen. Nun ist

$$\begin{aligned} F_n &= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \\ &\approx c \cdot 1.618^n, \text{ mit einer Konstanten } c. \end{aligned}$$

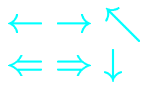
Ergo: $C_{max}(N) = O(\log_{1.618}(N+1)) = O(\log_2 N)$

Implementation



```
public static int search(Orderable A[],Orderable k){
  /* Durchsucht A[1], .., A[n] nach Element k und
       liefert den Index i mit A[i] = k; -1 sonst */
  int n = A.length-1;
  int fibMinus2 = 1, fibMinus1 = 1, fib = 2;
  while (fib - 1 < n) {
    fibMinus2 = fibMinus1;
    fibMinus1 = fib;
    fib       = fibMinus1 + fibMinus2;
  }
  int offset = 0;
  while (fib > 1) {
    /* Durchsuche den Bereich [offset+1,offset+fib-1]
       nach Schluessel k (Falls fib = 2, dann besteht
       [offset+1,offset+fib-1] aus einem Element!) */
    int m = min(offset + fibMinus2,n);
    if (k.less(A[m])) {
    // Durchsuche [offset+1,offset+fibMinus2-1]
        fib       = fibMinus2;
        fibMinus1 = fibMinus1 - fibMinus2;
        fibMinus2 = fib - fibMinus1;
      }
      else if (k.greater(A[m])) {
      // Durchsuche [offset+fibMinus2+1,offset+fib-1]
        offset     = m;
        fib       = fibMinus1;
        fibMinus1 = fibMinus2;
        fibMinus2 = fib - fibMinus1;
      }
      else // A[m] = k
        return m;
    }
    return -1;
  }
}
```

5 Exponentielle Suche

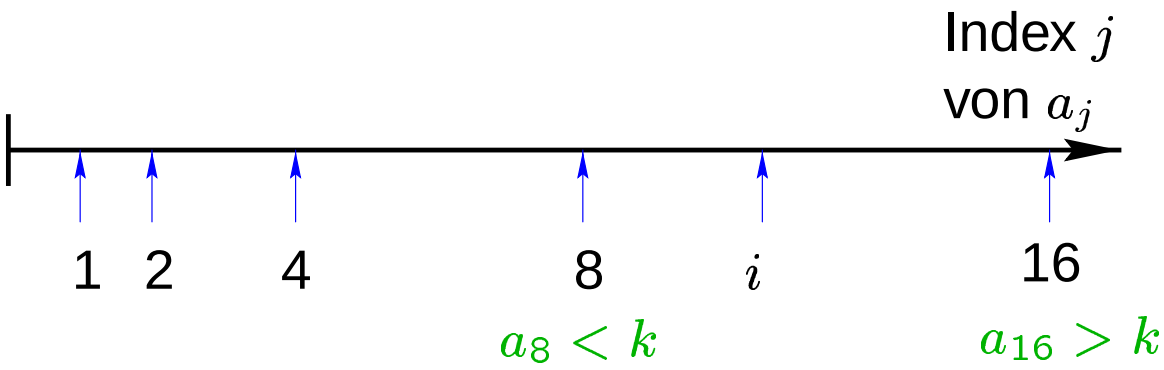


Annahme: n sehr groß, i mit $a_i = k$ klein

Denke-Zahl-Aus: Logarithmische Anzahl von Fragen

Eingabe: Sortierte Folge $a_1, \ldots, a_n$, Schlüssel k
Ausgabe: Index i mit $a_i = k$

```
j = 1;
while (k > a_j) j = 2*j;
return search (a,j/2,j,k);
```

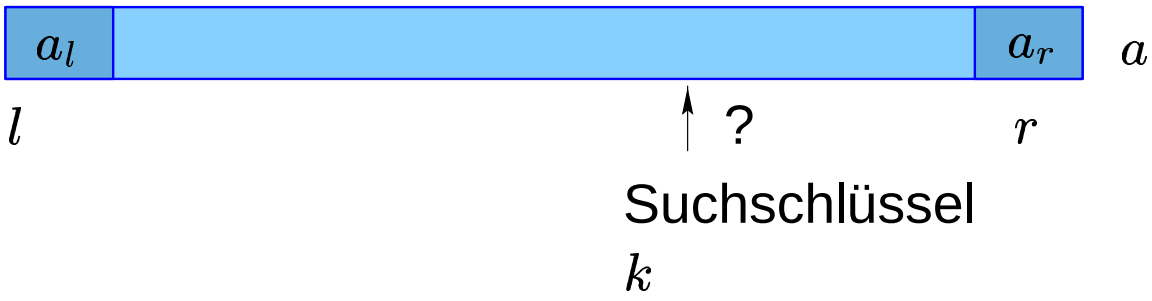


Analyse:

- $a_j \geq k$: $\lceil \log i \rceil$
- Binäre Suche: $2\lceil \log(i/2 + 1) \rceil$

Gesamtaufwand: $O(\log i)$

6 Interpolationssuche



Idee: Suche von Namen im Telefonbuch, z.B. Bayer und Zimmermann

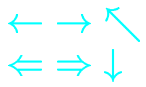
Erwartete Position von k (bei Gleichverteilung aller gespeicherten Schlüssel):

$$l + (r - l)\frac{k - a_l}{a_r - a_l}$$

Analyse:

- im schlechtesten Fall: $O(n)$
- im Mittel bei Gleichverteilung: $O(\log \log n)$

7 Das Auswahlproblem



Problem: Finde das i-kleinste Element in einer Liste F mit n Elementen

Naive Lösung

```
j = 0
while (j < i)
  bestimme kleinstes Element a_{min}
  entferne a_{min} aus F;
  j = j + 1;
return a_{min}
```

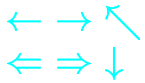
Anzahl der Schritte: $O(i \cdot n)$ für $i = n/2$ (Median): $O(n^2)$ (Sortieren ist besser)

Verfahren mit Heap

```
verwandle F in einen min-Heap
j = 0;
while (j < i)
  a_{min} = delete-min(F);
  j = j + 1;
return a_{min}
```

Anzahl der Schritte: $O(n + i \cdot \log n)$
für $i = n/2$ (Median): $O(n \log n)$

Implementation (Rahmen)



Basisklasse:

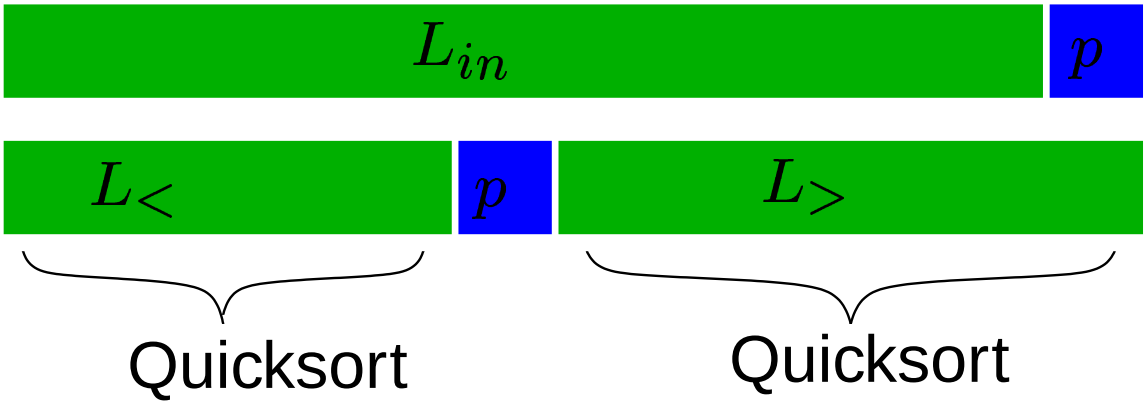
```
class SelectAlgorithm {
  static void swap (Object A[], int i, int j) {
    Object o = A[i]; A[i] = A[j]; A[j] = o;
  }
  static void select (Orderable A[], int i) {
    IthElement.select(A, i);
  }
  static void printArray (Orderable A[]) {
    for (int i = 1; i < A.length; i++)
      System.out.print(A[i].toString()+" ");
    System.out.println();
  }
}
```

In `IthElement extends SelectAlgorithm`:

```
public static Orderable select(Orderable A[],int i){
    // Suche das i-groesste Element in A[1],...,A[n]
  int n = A.length - 1;
  A [0].minKey();                          // Stopper
  if (i <= n) return A[selectIndex (A, i, 1, n)];
  return A[0];
}
```

Divide-&-Conquer Lösung

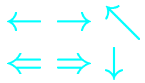
Idee: Aufteilung von $F = (a_1, \ldots, a_n)$ in zwei Gruppen bzgl. Pivotelement p (Quicksort)




```
public static int selectIndex
   (Orderable A[],int i,int l,int r) {
  // Suche den Index des i-groessten Elementes
  // in A[l],...,A[r]
  if (r > l) {
    int p = pivotElement(A, l, r);
    int m = Quicksort.divide(A, l, r, p);
    if (i <= m - l)
      return selectIndex (A, i, l, m - 1);
    return selectIndex (A, i - (m - l), m, r);
  }
  else return l;
}
```

Nur eine der zwei durch Aufteilung entstandenen Folgen wird weiter betrachtet.

Wahl des Pivotelements



1. $p = a_r$ folgt $T(n) \leq T(n-1) + O(n)$

Laufzeit im schlimmsten Fall: $O(n^2)$

Beispiel: Auswahl des Minimums in aufsteigend sortierter Folge

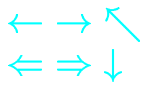
2. Randomisiert

$p =$ ein zufälliges Element aus $a_1, \ldots, a_n$

3. Median-of-Median

Bestimme Pivotelement als Median von Medianen von 5-er Gruppen
$\Rightarrow$ lineare Komplexität

8 Selbstanordnende lineare Listen



MF-Regel (Move-to-front): Mache ein Element zum ersten Element der Liste, nachdem auf das Element (als Ergebnis einer erfolgreichen Suche) zugegriffen wurde.

T-Regel (Transpose): Vertausche ein Element mit dem unmittelbar vorangehenden, nachdem auf das Element zugegriffen wurde.

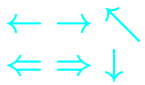
FC-Regel (Frequency Count): Nach jedem Zugriff auf ein Element wird dessen Häufigkeitszähler um 1 erhöht. Ferner wird die Liste nach jedem Zugriff neu geordnet und zwar so, daß die Häufigkeitszähler der Elemente in absteigender Reihenfolge sind.

Beispiel Liste $L = \{1, 2, 3, 4, 5, 6, 7\}$.
a) Greife 10x nacheinander auf $1, \ldots, 7$ zu.
b) Greife 10x auf 1, dann zehnmal auf 2, usw. zu.

MTF: a) $\frac{\frac{7 \cdot 8}{2} + 7 \cdot 9 \cdot 7}{10 \cdot 7} = 6.7$ b) $\frac{\frac{7 \cdot 8}{2} + 9 \cdot 7 \cdot 1}{10 \cdot 7} = 1.3$

Durchschnittliche (statischen) Zugriffskosten:
$(10 \cdot \sum_{i=1}^{7} i)/70 = 4$

Güte von Move-To-Front



Experimente: MTF besser als T und FC (Übung)

Ziel: Vergleich MTF mit beliebiger Strategie

$s = s_1 s_2 s_3 \ldots s_m$: Folge von Suchanfragen
A : Verfahren zur Selbstanordnung
$C_A(s) = \sum_{i=1}^{m} C_A(s_i)$: Kosten zur Verarbeitung von s
$V_A(s) = \sum_{i=1}^{m} V_A(s_i)$: # Vertauschungen nach vorn
$H_A(s) = \sum_{i=1}^{m} H_A(s_i)$: # Vertauschungen n. hinten

Bemerkung: Für MTF-, T- und FC-Regel gilt:
$H_{MTF}(s) = H_T(s) = H_{FC}(s) = 0$

Weiterhin: $V_A(s) \leq C_A(s) - m$.

Satz: Für jeden Algorithmus A zur Selbstanordnung von Listen und für jede Folge s von m Zugriffsoperationen gilt

$$C_{MTF}(s) \leq 2 \cdot C_A(s) + H_A(s) - V_A(s) - m.$$

D.h. (grob) MTF-Regel höchstens doppelt so schlecht ist wie jeder andere Algorithmus zur Selbstanordnung von Listen.