IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(10 - Suchverfahren)

Prof. Dr. Susanne Albers

Problemstellung

Problem: Gegeben Folge $F = (a_1,...,a_n)$. 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 Schlüssel k
       return -1;
  }
}
```

Analyse: - schlechtester Fall : $n + 1$

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

Binäre Suche

Gegeben: Folge $a_1, ..., a_n$ aufsteigend sortiert:

	a_1													a_n

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 Index i >= 1 mit
    A[i] = k; 0 sonst */
    int n = A.length - 1;
    return search(A, 1, n, k);
}
```

Rekursiver Aufruf

```
public static int search(Orderable A[], int l, int r, Orderable k){
    /* Durchsucht A[l], .., A[r] nach Element k und liefert 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;
}
```

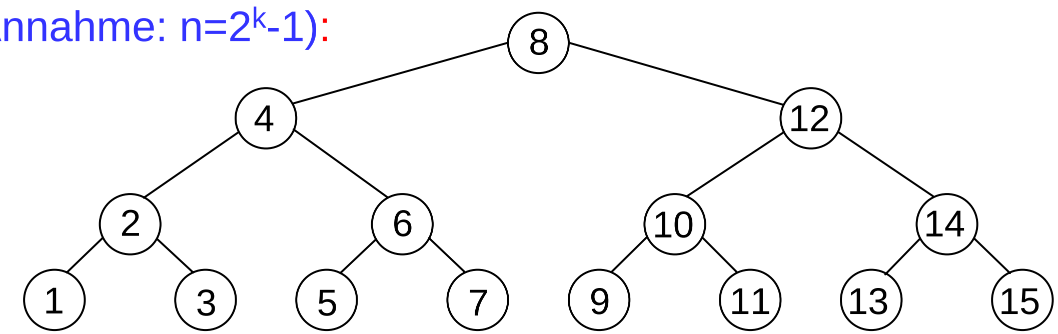
Binäre Suche ohne Rekursion

```
public static int search(Orderable A[], Orderable k){
      int n = A.length-1, 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;
}
```

Situation: Binärer Vergleichsoperator mit 3 Ausgängen; Berechnung der mittleren Position

Worst case (Annahme: $n=2^k-1$): Suche benötigt $k = \log(n + 1)$ Vergleiche

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



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Analyse

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

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

Erwartungswert:

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

Fibonacci-Suche

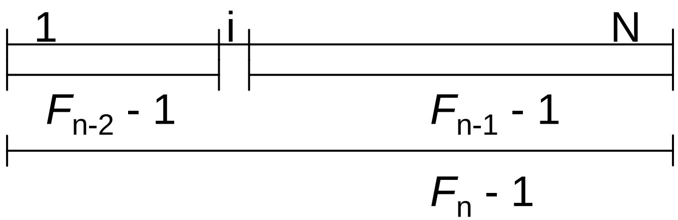
Erinnerung:

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

Verfahren:

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

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



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

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

$$\approx c * 1.618^n, \text{ mit einer Konstanten } c.$$

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 i = min(offset + fibMinus2,n);
       if (k.less(A[i])) {
       // Durchsuche [offset+1,offset+fibMinus2-1]
           fib = fibMinus2;
           fibMinus1 = fibMinus1 - fibMinus2;
           fibMinus2 = fib - fibMinus1;
       }
       else if (k.greater(A[i])) {
       // Durchsuche [offset+fibMinus2+1,offset+fib-1]
           offset = i;
           fib = fibMinus1;
           fibMinus1 = fibMinus2;
           fibMinus2 = fib - fibMinus1;
       }
       else // A[i] = k
           return i;
       }
   return -1;
   }
}
```

Exponentielle Suche

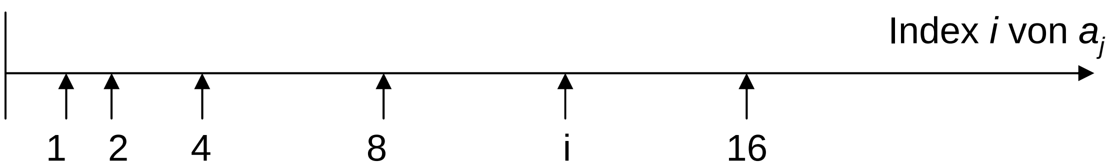
Situation: n sehr groß, i mit $a_i = k$ klein

Ziel: Finde beliebigen Schlüssel mit einer Anzahl von Vergleichsoperationen, die logarithmisch in der Position i des gesuchten Schlüssels ist.

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);
```



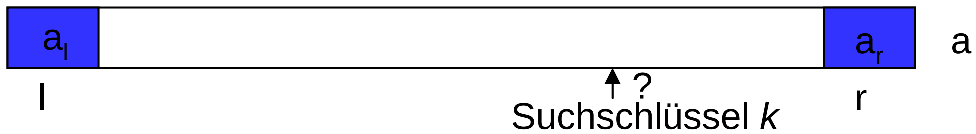
Analysis:

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

Gesamtaufwand: $O(\log i)$

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)

Das Auswahlproblem

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

1. 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: $O(i * n)$ für $i = n/2$ (Median): $O(n^2)$ (Sortieren ist besser)

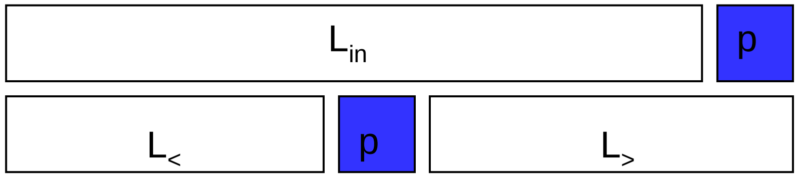
2. Verfahren mit Heap

```
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: $O(n+i*\log n)$ für $i = n/2$ (Median): $O(n \log n)$

Divide-and-Conquer-Lösung

Idee: Aufteilung von $F = a_1,...,a_n$ 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 = divide(A, l, r);
    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.

divide-Methode von Quicksort

```
static int divide (Orderable A [], int l, int r) {
  // teilt das Array zwischen l und r mit Hilfe
  // des Pivot-Elements in zwei Teile auf und gibt
  // die Position des Pivot-Elementes zurueck
  int i = l-1;            // linker Zeiger auf Array
  int j = r;                      // rechter Zeiger auf Array
  Orderable pivot = A [r]; // das Pivot-Element
  while (true){ // "Endlos"-Schleife
        do i++; while (i < j && A[i].less(pivot));
        do j--; while (i < j && A[j].greater(pivot));
        if (i >= j) {
                swap (A, i, r);
                return i;       // Abbruch der Schleife
        }
        swap (A, i, j);
  }
}
```

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 $a_1,\dots,a_n$

3. Median-of-Median

Bestimme Pivotelement als Median von Medianen von 5-er Gruppen

$\rightarrow$ lineare Komplexität