# Suchalgorithmen

Lineare Suche
Divide-and-Conquer-Suche





## Einleitung und Problemstellung

- Gegeben: abstrakter Datentyp
  *Folge von Elementen*
- Methode: Suche Element *a* in Folge *f*
  - Output: eine der (evtl. mehreren) Positionen *P(f,a)* aus *S = 0,1,...,len(f)-1*
    - *P(f,a)* ∈ *I*, *I*={ *p* | *f[p]=a* }, wenn *I* nicht leer
    - *P(f,a)* = *-1* g.d.w. *a* kommt in *f* nicht vor (*I* leer)





## Abstrakte Allgemeine Suchprozedur

suchePosition (`f`,`a`)
// `f` ist eine Folge, `a` ein Element. Dann ist *res*
// eine Position von `a` in `f`, falls `a` in `f` vorkommt,
// andernfalls ist *res* = -1.
1. **Initialisiere:** *res* = -1; `S` = Menge der Suchpositionen in `f`.
2. **Trivialfall:** if (`S` ==∅) return(*res*).
3. **Reduktion:** Wähle die nächste Suchposition `p` und entferne `p` aus `S`.
4. **Rekursion?** if (`f[p]` == `a`) return (`p` );
andernfalls weiter mit Schritt 2.

- Implementierungsdetails offen (Algorithmen-Schema)
  - Wahl der nächsten Suchposition
  - Realisierung des Tests *S* ==∅
- Realisierung führt zu konkretem Algorithmus.





## Lineare Suche

- Teste jedes Element der Folge in natürlicher Reihenfolge
- Entwurfsschema ist *greedy*
- Generischer Fall:
  - Folge f von Elementen des Typs Object (z.B. Reihung oder Liste)
  - Test auf Gleichheit: Virtuelle Funktion equals
- Implementierung der Methode linearSearch in Beispielklasse SearchClass
- linearSearch deklariert als static (greift nicht auf Instanzvariablen der kapselnden Klasse zu)

## Linear Search

```
public class SearchClass {
// ...
  /**
   * Anforderungen:
   * f eine Folge aus Elementen vom Typ Object
   * Zusicherungen:
   * Res == -1, falls a nicht in f vorkommt.
   * res == i, falls i die erste
   * Position mit f[i].equals(a).
   */
   public static int linearSearch(Object[] f, Object a)
{
     // Initialisierung
     int res = -1;
     // Iteration
     for (int i = 0; i < f.length; i++) {
       // Trivialfall: Suche erfolgreich
       if (f[i].equals(a))
         return (res = i);
     }
     return res;
 } }
```





## Lineare Suche mit Wächter

- Iteration der linearen Suche in while-Notation
  while ( i< f.length && !(f[i].equals(a)) ) i++;
- Zwei Tests je Iteration erforderlich
- Test i< f.length überflüssig, falls a in f vorkommt
- Daher Einführen eines *Wächters (engl. Sentinel)*
  - Füge a an das Ende von f an
  - Nach while-Schleife Prüfung ob ein a in f oder der Wächter gefunden wurde (Erkennung anhand von Zähler i)
- Erfinder des Wächters: N. Wirth
  - Motivation: mit traditionellen Programmiersprachen wird Ende von Reihung nicht automatisch erkannt





## Lineare Suche in Java ohne Wächter

- Java macht Test (*array bounds check*) automatisch
- Java erzeugt Ausnahme (*exception*)
  IndexOutOfBoundsException
  falls f[i] ungültigen Index i enthält
- Anstelle eines Wächters die Ausnahme auffangen:

```
try{
  while (!f[i].equals(a)) i++;
   // now f[i].equals(a)
  return(i);
}
catch (IndexOutOfBoundsException ioobe)
    { return(-1); }
```

- Kein doppelter Test, dafür aber schlechter Stil:
  Reguläre Ergebnisse nie durch Exceptions generieren!





## Lineare Suche: Komplexität I

- Aufwand:
  - $k_1$ Operationen (Ops) für Initialisierung
  - $k_2$ Ops für Test auf Trivialfall
  - $k_3$ Ops für Reduktion und Rekursion (Schleifenverwaltung)
- Maximal n=f.length Iterationen, also
  - $k_1 + n\,(k_2 + k_3)$ Ops maximal
- Wenn *a* erstes Element von *f*
  - $k_1 + k_2 + k_3$ Ops minimal
- Konstanten $k_1$ , $k_2$ , $k_3$ rechnerabhängig, daher asymptotischer Aufwand
  - $O(n)$

## Lineare Suche: Komplexität II

- Aufwand im Mittel:
  - Modellierung der Anfragen: Nach jedem Objekt *a* in *f* wird gleich oft gefragt
  - Anzahl der Schleifendurchläufe im Mittel

$$\frac{1+2+\ldots+n}{n} = \frac{n\cdot(n+1)}{n\cdot 2} = \frac{n+1}{2} \approx \texttt{Schleifendurchgänge.}$$

- Die asymptotische Komplexität von linearer Suche ist linear (im *worst case* als auch im Mittel)
- Bemerkung: hier wurde wahlfreier Zugriff auf Elemente in *f* mit *O(1)* Ops angenommen (*random access*)
  - Bei verlinkten Listen naiver Zugriff in *O(n)* Ops
  - Damit ergibt sich Komplexität von $O(n^2)$
  - Zugriff in *O(1)* Ops und Gesamtkomplexität von *O(n)* möglich, wenn Zeiger auf f[i] gehalten wird





## Divide-and-Conquer-Suche I

- Teile und Herrsche:
  - Zerlege Problem in (kleinere) Teilprobleme
  - Löse Teilprobleme (i.d.R. rekursiv)
  - Füge Lösungen zu Gesamtlösung zusammen
- Binäre Suche:
  - Wähle eine Position *p*, die die Folge in linke und rechte Teilfolge zerlegt
  - Sinnvoll wenn Folge sortiert: dann muss nur in einer der beiden Teilfolgen weiter gesucht werden
  - Dazu müssen Objekte auch eine Vergleichsoperation unterstützen also z.B. vom Referenztyp Comparable sein





## Divide-and-Conquer-Suche II

```
public class SearchClass {
// ...

  /**
   * Anforderungen:
   *   f: aufsteigend sortierte Folge von Elementen
   *      f[i], i>=0.
   *   a: gesuchtes Element in f.
   *   l: linker Rand der zu
   *      durchsuchenden Teilfolge von f;
   *      0 <= l <= i <= r.
   *   r: rechter Rand der zu
   *      durchsuchenden Teilfolge von f;
   *   0 <= l <= i <= r.
   * Zusicherungen:
   *   res == p, falls f[p] == a und l <= p <= r,
   *   res == -1, sonst.
   */
```





## Divide-and-Conquer-Suche III

```
public static int binarySearch(Comparable[] f, Comparable a, int l, int r) {
  // (1) Initialisiere
  int p = (l+r)/2;
  int c = f[p].compareTo(a);
  // (2) Trivialfall: Element gefunden
  if (c == 0)
    return p;
  // (3) Trivialfall: Element nicht vorhanden
  if (l == r)
    return -1;
  // (4) Rekursion
  if (c > 0) {
    if (p > l) return binarySearch(f, a, l, p-1);
    else return (-1);
  }
  else {
    if (p < r) return binarySearch(f, a, p+1, r);
    else return (-1);
  }
 }
}
```

- Spezialfall: Wahl vom p=l oder p=r ergibt die lineare Suche in rekursiver Form

## Binäre Suche: Komplexität

- Minimaler Aufwand wenn f[p]=a für p=(l+r)/2

  $T_{best}(n) \in O(1)$
- Maximaler Aufwand wenn a nicht in f vorkommt

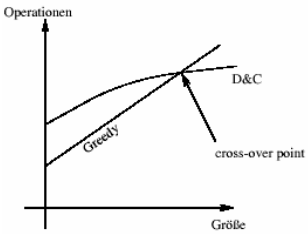
  $T_{worst}(n) = k + T_{worst}(n/2) = 2k + T_{worst}(n/4) = \ldots$
  $= k \log_2(n) + T_{worst}(1) = k\lfloor \log_2(n) \rfloor + k_1 + k_2 + k_3.$
- Das ergibt asymptotische Komplexität von *O(log n)*





## Cross-over Point

- Divide-and-Conquer-Verfahren (D&C) haben relativ großen Verwaltungsaufwand (*Overhead*) bzw. große Proportionalitätskonstanten in O()-Komplexitäten
- Greedy-Verfahren haben kleinen Overhead
- Es gibt i.d.R. einen Übernahmepunkt (*Cross-over Point*) zwischen beiden Verfahren
  - Liegt heute typischerweise bei *n*=16 bis 32
  - Kann experimentell bestimmt werden








## Kombinationsverfahren

- Benutze rekursives Divide-and-Conquer-Verfahren um große Probleme zu behandeln bis Größe kleiner als Cross-over Punkt
- Nutze Greedy-Verfahren, um kleine Probleme zu lösen
- Spezielle, schnelle Implementierungen von Greedy-Verfahren möglich, da nur sehr kleine Probleme behandelt werden müssen
- Sehr große Beschleunigungen möglich, da sehr viele kleine Problem zu lösen sind?





## Kombinationsverfahren: Analyse und Design I

- Problem: Ab welcher Problemgröße soll das greedy-Verfahren eingesetzt werden?
- Analyse: Modelliere oder bestimme empirisch Laufzeiten $T_d(n)$, $T_g(n)$, $T'_m(n)$ von D&C, greedy- und Kombinationsverfahren mit greedy-Einsatz bei $n=2^m$
- Beispielansätze (etwa bei Sortierverfahren):
  - $T_d(n) = k'n + 2\,T_d(n/2)$ mit $k'=1$ (Normierung) ergibt $T_d(n) = n\,(\log_2 n + 1)$
  - $T_g(n) = k\,n^2$
  - Bei $k=0.1$ Cross-over point zwischen 64 und 128
    - n=32: $T_g(n) = 32*32*0.1 = 102.4$; $T_d(n) = 32(5+1) = 192$
    - $n=64$: $T_g(n) = 64*64*0.1 = 409.6$; $T_d(n) = 64(6+1) = 448$
    - $n=128$: $T_g(n) = 128*128*0.1 = 1638.4$; $T_d(n) = 128(7+1) = 1024$

## Kombinationsverfahren: Analyse und Design II

- Modell liefert Anzahl von Operationen
  $T'_m(n) = n\,(\log_2 n - m + k\,2^m)$
- Also: Minimiere $T'_m(n)$ durch Minimierung von
  $f(m) = -m + k\,2^m$
- Extremwertrechnung der Analysis (löse $f'(m)=0$) liefert für $k = 0.1$ den Wert $m \approx 3.9$
- Bei $n = 2^{20}$ (= 1 Mega), $k = 0.1$, $m = 4$ erhält man
- $T_d(2^{20}) = 21$ Mops (mega Operationen)
- $T_g(2^{20}) \approx 100\,000$ Mops
- $T'_4(2^{20}) \approx 17.6$ Mops (16% Einsparung gegen D&C)