# Suchalgorithmen

Lineare Suche
Divide-and-Conquer-Suche

- 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)* $\in$ *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: 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

  $T_{\text{best}}(n) \in O(1)$

- Maximaler Aufwand wenn a nicht in f vorkommt

  $T_{\text{worst}}(n) = k + T_{\text{worst}}(n/2) = 2k + T_{\text{worst}}(n/4) = \ldots$
  $= k \log_2(n) + T_{\text{worst}}(1) = k\lfloor \log_2(n) \rfloor + k_1 + k_2 + k_3.$

- Das ergibt asymptotische Komplexität von *O(log n)*