Informatik II
Suchen

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Problemstellung

- Gegeben ist eine Menge von Datensätzen $\{A_1,...,A_n\}$
- Gesucht sind die Datensätze, deren Schlüssel (*Key*) = **`A[i].key`**
- Betrachte ab jetzt o.B.d.A. nur noch die Folge der Keys, d.h., die A_i sind die *Keys* der Datensätze
- Allgemeine Spezifikation für das Suchproblem:
  - Eingabe: Array A, gesuchtes Element x
  - Ausgabe: Index i mit A[i] = x, falls x in A

Lineare Suche

- Wenn nichts über die (An-)Ordnung der Elemente in der Datenstruktur bekannt ist, kann nur die lineare Suche benutzt werden:
  - Besuche der Reihe nach die Elemente aus dem Container, bis ein Element mit der gewünschten Eigenschaft gefunden wurde, oder alle besucht wurden
- Beispiel: suche gegebene Telefonnummer in einem Telefonbuch
- Aufwand
  - worst case: n Schleifendurchläufe
  - average case:

$$\frac{1}{n}(1+2+\ldots+n) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} \approx \frac{n}{2}$$

Binäre Suche

- Nutze die totale Ordnung auf den Elementen
- Annahme: die n Elemente seien nun aufsteigend sortiert (geordnet), d.h.

$$\forall i, 0 \leq i < n-1 : A_i \leq A_{i+1}$$

- Beispiel: suche nach einem Namen im Telefonbuch
- Lösungsstrategie: "Intervallhalbierung"
  - Schlage Telefonbuch in der Mitte auf, vergleiche gesuchten Namen mit einem Namen auf der Seite
  - Entscheide, in welcher Hälfte des Telefonbuches sich der gesuchte Name befindet (halbiert den Suchraum grob gerechnet)
  - Wiederhole Verfahren mit dieser Hälfte, bis richtige Seite gefunden ist

Beispiel

Gesucht: x=32

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	: Index
12	17	23	24	31	32	36	37	42	47	53	55	67	67	87	89	91	91	93	Min=0, Max=18

Der rekursive Algorithmus ...

```
def binsearch_start( A, key ):
    """ Return index i such that 0 <= i < len(A)
        and A[i] == key, or, if no such i exists,
        returns -1. """
    if len(A) == 0:   return -1
    if len(A) == 1:
        if A[0] == key: return 0
        else:           return -1
    return binsearch_work( A, key, 0, len(A)-1 )
```

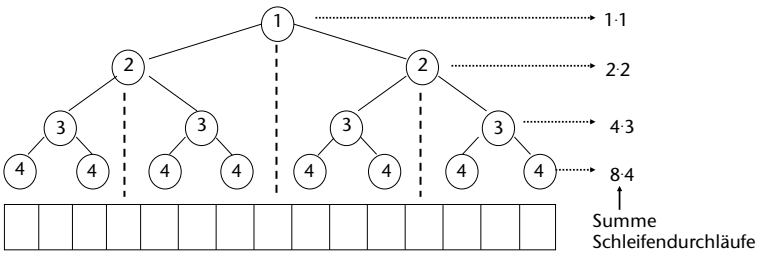
```
def binsearch_work( A, key, l, r ):
    """ The work horse for binsearch_start. """
    if r < l:
        return -1                    # k not found
    m = (l + r) / 2
    if key < A[m]:
        return binsearch_work( A, key, l, m-1 )
    if key > A[m]:
        return binsearch_work( A, key, m+1, r )
    return m
```

... und nicht-rekursiv

```
def binsearch( A, key ):
    l = 0
    r = len(A) - 1
    while l <= r:
        m = (l + r) / 2
        if key < A[m]:
            r = m - 1
        elif key > A[m]:
            l = m + 1
        else:
            return m
    return -1
```


Laufzeit-Analyse

- Annahme: $n = 2^k - 1$
- Beispiel:



- Worst case: #Schleifendurchläufe = $k = \log(n+1) \approx \lceil \log n \rceil$
- Mittlere Anzahl Schleifendurchläufe: $\frac{1}{n}\sum_{i=1}^{k} i2^{i-1} = \frac{1}{n}((k-1)2^k + 1) \approx \log n$

Beweis für $\sum_{i=1}^{k} i2^{i-1} = (k-1)2^k + 1$

$$\sum_{i=1}^{k} i2^{i-1} = \underbrace{\begin{array}{llll} 2^0 & + & & \\ 2^1 & + & 2^1 & + \\ 2^2 & + & 2^2 & + \; 2^2 + \\ \ldots & & & \\ 2^{k-1} & + \; \ldots & \ldots \; + & 2^{k-1} \end{array}}_{k}$$

$$= \sum_{0}^{k-1} 2^i + \sum_{1}^{k-1} 2^i + \ldots + \sum_{k-1}^{k-1} 2^i$$

$$= \begin{array}{l} (2^k-1)- \\ (2^0-1) \end{array} + \ldots + \begin{array}{l} (2^k-1)- \\ (2^l-1) \end{array} + \ldots + \begin{array}{l} (2^k-1)- \\ (2^{k-1}-1) \end{array}$$

$$= k2^k - \sum_{i=0}^{k-1} 2^i = k2^k - (2^k - 1) = (k-1)2^k + 1$$

Lineare Suche vs. binäre Suche

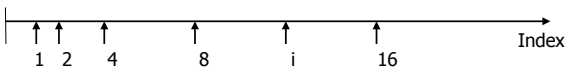
- Bei großen Datenmengen ist binäre Suche wesentlich effizienter
  - Verdoppelung der Datenmenge →
    - lineare Suche: doppelter Aufwand
    - binäre Suche: ein weiterer Vergleich
- Bei kleinen Datenmengen (n≈10) ist lineare Suche schneller!
- Nicht jede Container-Datenstruktur ist für binäre Suche geeignet
- Muss man häufig in einem unsortierten Container suchen, lohnt es sich, die Elemente 1x in ein Array zu kopieren und dieses zu sortieren (*Precomputation*). Dann kann man die binäre Suche wiederholt anwenden. (Demnächst)

Key-Indizierte Suche

- Triviale Lösung, falls Key-Menge klein
  - Ist eigtl gar keine Suche
- Idee: Speichere Daten in einem Array "mit Lücken"
  - `A[k]` enthält Datensatz mit Key `k` (inkl. "Nutzdaten")
  - Unbenutzte `A[i]` enthalten `None`
- Suche nach Key k ist trivial: liefere `A[k]`
- Bedingungen:
  - Wertebereich der Keys muss im voraus bekannt sein
  - Datensätze mit gleichen Keys kann man nicht speichern
- Verallgemeinerung: Hash-Tables (später)

Exponentielle Suche

- Situation:
  - n sehr groß
  - Gesuchtes i, mit $A_i = k$, ist relativ klein
- Idee: suche zunächst "rechten Rand" r, so dass $k < A_r$
- Algo:




```
r = 1
while A[r] < key:
    r *= 2
binsearch( A, key, r/2, r )
```

- Analyse:
  - Rechten Rand suchen: $\lceil \log i \rceil$ viele Schleifendurchläufe
  - Binärsuche: ~ $\lceil \log i \rceil$

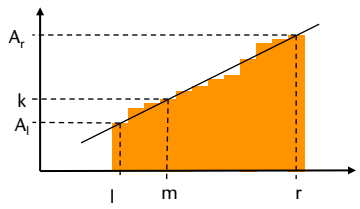
Interpolation search

- Beobachtung:
  - Bei Suche nach "Dix" oder "Zachmann" im Telefonbuch schlagen wir es eher weiter vorne bzw. weiter hinten auf
- Idee des Algorithmus':
  - Schätze die Position des gesuchten Keys im Array
- Erinnerung: bei binärer Suche wird das noch zu durchsuchende Index-Intervall bei $m = l + \frac{1}{2}(r - l)$ unterteilt
- Ersetze dort den Faktor 1/2 durch

$$m = l + (r - l)\frac{k - A_l}{A_r - A_l}$$

- Rest des Algo ist analog zu Binärsuche

- Klar ist: wenn Keys halbwegs linear aufsteigend sind, liegt dieses interpolierte m sehr dicht am gesuchten Key

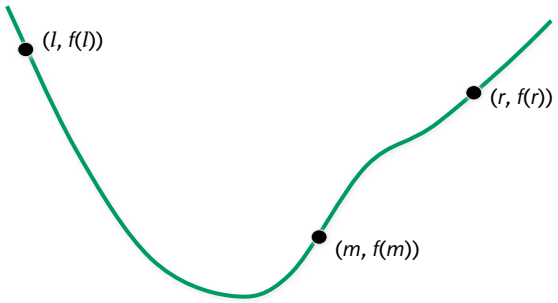


- Voraussetzung:
  - Ordnungsrelation auf den Keys reicht nicht!
  - Man muss auf den Keys rechnen können
- Aufwand (o.Bew.)
  - Average case: ~ log log n
  - Worst case: ~ n (!)

Minimumsuche mit *Golden Section Search*

- Aufgabe:
  - Gegeben eine 1-dim., stetige Funktion *f*
    - Entweder als mathematische Funktion $f : \mathbb{R} \to \mathbb{R}$
    - Oder als Array $f : \mathbb{N} \to \mathbb{R}$
  - Gesucht: ein Minimum x* von *f*
- Erinnerung: Binärsuche klammert (*to bracket*) das Intervall, in dem sich der gesuchte Key befindet, und verkleinert diese Klammer sukzessive
- Frage: kann man *Bracketing* auch bei der Minimumsuche anwenden?

- Lemma:
  Sei $l < m < r$ gegeben.
  Falls gilt
  $$f(m) < f(l) \ \wedge \ f(m) < f(r)$$
  dann muß (mindestens) ein (lokales) Minimum von f im Intervall $[l,r]$ liegen.

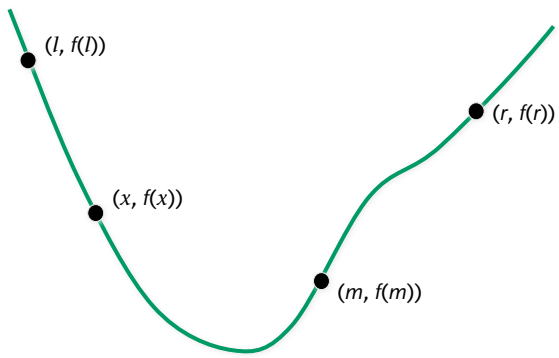


- Definition:
  Im Folgenden ist eine Klammer (für die Minimumsuche) ein Tripel (l,m,r) mit $l < m < r$ und $f(m) < f(l) \wedge f(m) < f(r)$.

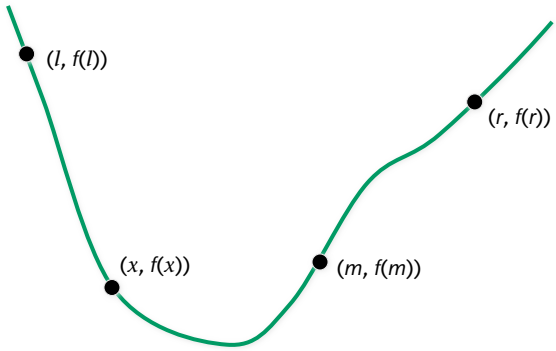
- Algorithmus zum Finden einer solchen initialen Klammer:

```
Input: initiales x0, increment delta
Berechne f(x0) und f(x0+delta)
If fallend: wiederhole schrittweise
   nach rechts bis f wieder steigt
Else: gehe analog nach links
Erhöhe delta bei jedem Schritt
```

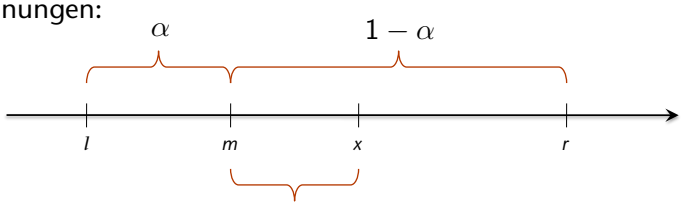

- Idee zur Verkleinerung der Klammer: werte f an einer Stelle x "in der Mitte" aus
- Fall 1: neue Klammer ist (x, m, r)



- Idee zur Verkleinerung der Klammer: werte f an einer Stelle x "in der Mitte" aus
- Fall 2: neue Klammer ist (l, x, m)



- Wo wählt man am besten x?
  - Klar ist, dass man x besser in dem längeren der beiden Intervalle (l,m) und (m,r) plaziert
- Gesucht ist die Stelle x (auch in folgenden Iterationen), so daß das Minimum auch dann möglichst schnell gefunden wird, wenn die Funktion so "unkooperativ" wie möglich ist
- Bezeichnungen:



$$\frac{m-l}{r-l} = \alpha \qquad \beta = \frac{x-m}{r-l} \qquad \frac{r-m}{r-l} = 1-\alpha$$

- Je nach Fall hat das nächste Intervall die (relative) Länge
  $\alpha + \beta$ oder $1 - \alpha$
- Ziel: beide gleich groß machen
- Folgerung:

$$\beta = 1 - 2\alpha \quad \Rightarrow \quad \alpha < \tfrac{1}{2} \qquad (1)$$

- Weitere Überlegung: wenn α optimal ist, muß eine Skalenähnlichkeit gelten, d.h. (im Fall 1)

$$\frac{\beta}{1-\alpha} = \frac{\alpha}{1} \qquad (2)$$

- (1) in (2) einsetzen liefert:

$$\alpha^2 - 3\alpha + 1 = 0 \quad \Rightarrow \quad \alpha = \frac{3-\sqrt{5}}{2} \approx 0.38197$$

- Daher kommt der Name dieses Suchverfahrens, da

$$1 - \alpha = \overline{\varphi} \approx 0.61803$$

  der goldene Schnitt ist

- Laufzeit: nur lineare Konvergenz
  - Pro Schritt verkleinert sich das Intervall um ca. 30%
  - Dafür aber garantierte Konvergenz (im Gegensatz zu anderen Verfahren)

- Definition:
  Eine stetige Funktion f heißt unimodal im Intervall $[a,b]$ gdw. es gibt ein eindeutiges x^*, so daß
  $f(x^*)$ = Minimum auf $[a,b]$ und
  f ist streng monoton fallend auf $[a,x^*]$ und streng monoton steigend auf $[x^*,b]$.

- Klar ist:
  Golden Section Search findet garantiert das globale Minimum einer unimodalen Funktion.