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

$$\sum_{i=1}^{k} i2^{i-1} = \begin{array}{l} 2^0 + \\ 2^1 + 2^1 + \\ 2^2 + 2^2 + 2^2 + \\ \ldots \\ 2^{k-1} + \ldots \quad \ldots + 2^{k-1} \end{array} \Big\} k$$

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

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

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

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: ~$\lceil \log i \rceil$

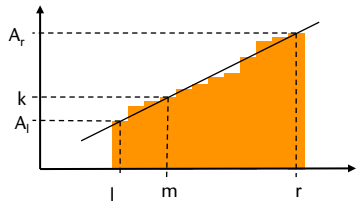
Interpolation search

- 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 (!)

Key-Indizierte Suche

- Triviale Lösung, falls Key-Menge klein
- 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)

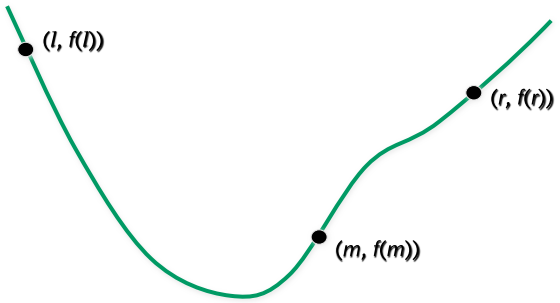
Lineare Suche vs. binäre Suche

- 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 jeder Behälter ist für binäre Suche geeignet
- Muss man häufig in einem unsortierten Behälter suchen, lohnt es sich, die Elemente einmal in ein Array zu kopieren und dieses zu sortieren. Dann kann man die binäre Suche wiederholt anwenden. (Demnächst)

Minimumsuche mit *Golden Section Search*

- Aufgabe:
  - Gegeben eine 1-dim. Funktion *f*
    - Entweder als mathematische Funktion $f : \mathbb{R} \rightarrow \mathbb{R}$
    - Oder als Array $f : \mathbb{N} \rightarrow \mathbb{R}$
  - 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.

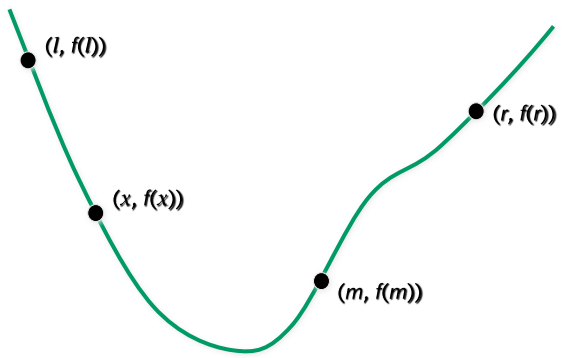


- Definition:
  Im Folgenden ist eine Klammer (für die Minimumsuche) ein Tripel (l,m,r) mit $l < m < r$ und $f(m) < f(l) \wedge f(m) < f(r)$.

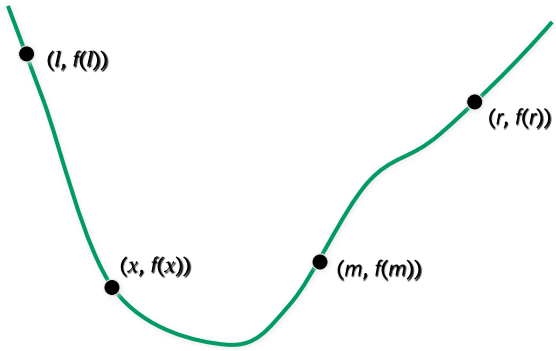
- Algorithmus zum initialen Finden einer solchen Klammer:

```
input: initiales x0, increment delta
berechne f(x0) und f(x0+delta)
falls fallend, wiederhole schrittweise
   nach rechts bis f wieder steigt
ansonsten 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)

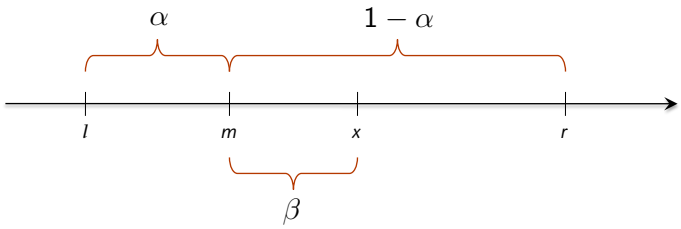


- Idee zur Verkleinerung der Klammer: werte f an einer Stelle x "in der Mitte" aus
- Fall 2: neue Klammer ist (l, x, m)



- Wo wählt man am besten x?
- 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:



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

- 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 Sucherfahrens, da

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

  der goldene Schnitt ist

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

- Definition:
  Eine 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.