6.2. Komplexität

6.2.1. Einleitung

* bisher qualitative Merkmale: Überschaubarkeit; Korrektheit

∃ auch quantitative Merkmale: Effizienz; Ressourcenverbrauch

hier: Aufwand zur Abarbeitung eines A. durch einen Prozessor;

Jedoch: unterschiedl. Prozessoren + Algorithmen ⇒
„langsamer" A. auf Pentium vielleicht mehr Rechenzeit als
„schneller" auf 386er;
Oder: Ein „erstbester" A. ist gut für kleinere Probleme und
schrecklich für große Probleme.

→ Frage 1: Wie Aufwand messen?
→ Frage 2: Was sind „kleine" bzw. „große" Probleme?

1 Antwort: Wählen eine Basisoperation und bestimmen bzw. schätzen deren Anzahl ab.

* Bsp.: MULT1

Basisoperation → Add. oder Subtr.;
Aufwand := $A_{MULT1}(m, n) \approx$ Anz. Durchläufe $= 2 \cdot n$

Aber: Aufwand von Eingabe n abh. →
wenn m klein und n groß ⇒ vertausche m und n:

MULT2 mit vorbereitendem Schritt
(0) WENN $n > m$ DANN $h \leftarrow n$; $n \leftarrow m$; $m \leftarrow h$;

⇒ $A_{MULT2}(m, n) \approx 2 \cdot \min(m, n)$!

⇒ MULT2 ≻ MULT1

* Bsp.: SUCHE

Wie hier Aufwand? Basisoperation: Vergleich; Wertzuweisung;

Wenn Suche bei 1. Element begonnen →

$$A(n) = \begin{cases} 2, & \text{falls } a_1 = a; \\ 2 \cdot n, & \text{falls } a_n = a; \end{cases}$$

$$A(n) = 2 \cdot i, \quad \text{falls } a_i = a, \; i = 1(1)n;$$

⇒ Nutzen ist gering;

$\Rightarrow$ 3 übliche Maße:

$A_{min}(n)$ – Aufwand im günstigsten Fall (best–case–Analyse)
$A_{max}(n)$ – Aufwand im ungünstigsten Fall (worst–case–Analyse)
$\overline{A}(n)$ – Aufwand im Mittel (average–case–Analyse)
$\downarrow$
?

Nur klar: Abh. von Werten der Eingabeparameter. Benötigen Wahrscheinlichkeitsverteilung für Position des gesuchten Elements $\rightarrow$ vereinfachte Annahme: Alle Positionen sind gleich wahrscheinlich.

$$\Rightarrow \quad \overline{A}(n) = \sum_{i=1}^{n} 2 \cdot i \cdot P(a_i = a) = \frac{1}{n} \cdot 2 \cdot \sum_{i=1}^{n} = \frac{2}{n} \cdot \frac{n}{2}(n+1) = \underline{\underline{n+1}}$$

* Vergleich von A. $\Rightarrow$ i. d. R. bzgl.

a) $\overline{A}(n)$
b) $A_{max}(n)$ und
c) asymptotischer Aufwand

Warum c) ?

Seien z.B. zwei Algorithmen I und II gegeben mit $\overline{A_I}(n) = 2n^2 + 50$ und $\overline{A_{II}}(n) = 100 \cdot n$

Welcher ist „besser“?

Für I ignorieren wir 2 und 50 und sagen: Aufwand ist proportional n^2.
Da n^2 größer n sagen wir: I benötigt höheren (asymptot.) Aufwand als II.
! Ungeachtet, daß für $n \leq 49$ I $\succ$ II.

$\Rightarrow$ Auch Antwort: Große/kleine Probleme?
Relation hier: Für kleine Werte, d.h. $n \leq 49$ ist I $\succ$ II;
für große Werte ab $n = 50$ ist II $\succ$ I.

$\Rightarrow$ Vergleich nie (i. d. R.) absolut, sondern in Abhängigkeit von Problemgröße.

6.2.2. Die O – und die Ω – Notation

* brauchen best. Formalismus, da es um quant. Aussagen geht;

- ein Problem P $\rightarrow$ unendlich viele Fragestellungen p;
  z.B.: Eine Fragestellung des Max.-problems wäre, die größte der vier Zahlen 13, 6, 31, 12 zu bestimmen;

- $\forall\, p \in$ P wird eine nat. Zahl $g(p)$ als Größe zugeordnet (ergibt sich meist in nat. Weise $\rightarrow$ 4 Zahlen zu sortieren; …);
- T(p, A) – Rechenzeitaufwand für einen Algor. A zur Lösung des Problems P an der Fragestellung $p \in$ P;

- im konkreten Fall durch Messungen; interessanter (und objektiver) sind globale Aussagen zu beliebigen Fragestellungen der Größe n

$\Rightarrow$ 3 Abstraktionen:

(a) Verhalten im ungünstigsten Fall (worst case)
$T^*(n, A) := sup \{ T(p, A): p \in P \text{ und } g(p) = n \}$

(b) Verhalten im besten Fall (best case)
$T_*(n, A) := inf \{ T(p, A): p \in P \text{ und } g(p) = n \}$

(c) Verhalten im Mittel (average case)
$\overline{T}(n, A) := E \{T(\underline{p}, A) \mid \underline{p} \text{ verteilt in } P \text{ und konzentriert auf } g(p) = n \}$

Klar: $\overline{T}(n, A) \leq T^*(n, A)$.

Aber: Welche Wahrscheinlichkeitsverteilung?

$\Rightarrow$ meist: (i) worst–case–Analyse;
(ii) average–case–Analyse mit Gleichverteilung;

- bei Analyse $\rightarrow$ nur Größenordnung der Laufzeit in Abhängigkeit von der Größe der Eingabe

für Analyse solcher oberer und unterer Schranken bzw. Wachstumsordnungen
$\Rightarrow$ Groß–O– und Groß–Omega–Notation

a) $f: \mathbb{N} \rightarrow \mathbb{N}$ sei

$O(f) := \{ h: \exists c_1 > 0, c_2 > 0, n_0 \geq 0 \text{ so, daß } h(n) \leq c_1 \cdot f(n) + c_2 \text{ für } n \geq n_0 \}$

{ für hinreichend großes n ist die Fkt. h nicht mehr als das c_1–fache der Fkt. f }

$\Rightarrow$ Sprechweise: Statt „Die Laufzeit $\overline{T}(n, A)$ erfüllt für alle $n \in \mathbb{N}$: $\overline{T}(n, A) \leq c_1 \cdot n + c_2$!“
sagt man
$\overline{T}(n, A)$ ist von der Ordnung n oder
$\overline{T}(n, A)$ ist $O(n)$ oder
$\overline{T}(n, A)$ ist in $O(n)$;
schreibt man
$\overline{T}(n, A) = O(n)$ oder
$\overline{T}(n, A) \in O(n)$;
{ Exakt wäre: $\overline{T}(n, A) \in O(f)$ mit $f(n) = n$. }

Bem.: Die Konstanten c_1 und c_2 interessieren nicht!
$\Rightarrow$ Abschätzung nach oben!

b) Abschätzung von unten $\Rightarrow$

$\Omega(g) := \{ h : \exists c > 0$ so, daß für unendlich viele $n \in \mathbb{N}$ gilt $h(n) \geq c \cdot g(n) \}$

* Bem.: Bisher Komplexität von Algorithmen.
In Praxis auch Komplexität von Problemen:

Problem P hat (Zeit-)Komplex. $O(n^2)$ heißt, es existiert ein A. zur Lösung von P, dessen Laufzeit durch eine quadratische Funktion beschränkt ist.

Klar: Für obere Schranke $\rightarrow$ 1 Algorithmus benötigt.
Für untere Schranke $\rightarrow$ muß alle beachten (schwer; wenige Fälle bisher)

* häufigsten Funktionen zur Effizienzmessung von Algorithmen:

Wachstum			
logarithmisch	:	log n	(i. d. R. zur Basis 2)
linear	:	n	
n * log n	:	n · log n	
quadratisch	:	n^2	($n^k \rightarrow$ polynomiales Wachstum)
exponential	:	c^n	

$\Rightarrow$ z. Z. allg. Überzeugung:

praktikabel höchstens A. mit polynomialem Wachstum; A. mit exponentialem Wachstum sind schon für kleine Problemgrößen nicht mehr ausführbar.

6.2.3. Einige Prinzipien zur Berechnung der Komplexität von Algorithmen

(Lit.: U. Manber: Introduction to algorithms / Addison Wesley 1989)

* A. bestehe aus versch. Teilen $\Rightarrow$ seine Komplexität ist gleich der Summe seiner Teile; nicht so einfach bei Schleifen oder Rekursion.

Bsp.: Schleife; n Durchläufe; Durchlauf i erfordere i Operationen $\Rightarrow$
Gesamtzahl der Operationen: $1 + 2 + \ldots + n = \frac{n}{2}(n+1)$
Wie aber, wenn i. Durchlauf i^2 Operationen?

Lemma 6.1: Für $s_2(n) = \sum_{i=1}^{n} i^2$ gilt $s_2(n) = \frac{n(n+1)(2n+1)}{6}$.

Beweis: Klar: $s_2(n) \leq n^3 = n \cdot n^2$ $\Rightarrow$ Differenz ist endlich.
Prüfen diese Vermutung mittels Induktion.
Vermutung $\Rightarrow$
$s_2(n) = P(n) = an^3 + bn^2 + cn + d$ mit
$P(1) = 1$ und

$$P(n+1) = P(n) + (n+1)^2$$

$$a(n+1)^3 + b(n+1)^2 + c(n+1) + d - an^3 - bn^2 - cn - d = n^2 + 2n + 1$$

$$\Rightarrow \begin{array}{ll} 3a + b - b = 1 & (\text{für } n^2) \\ 3a + 2b + c - c = 2 & (\text{für } n) \\ a + b + c + d - d = 1 & (\text{für } 1) \\ P(1) = 1 \;\Rightarrow a + b + c + d = 1 & \end{array} \Bigg\} \Rightarrow \underline{\underline{d = 0}}$$

$$\underline{\underline{a = \frac{1}{3}}} \quad \underline{\underline{b = \frac{1}{2}}} \quad \underline{\underline{c = \frac{1}{6}}}$$

$$\Rightarrow s_2(n) = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = \frac{n(n+1)(2n+1)}{6}$$

■

∃ weiteres Beweisverfahren, das allg. anwendbar:

Sei $s_3(n) = \sum_{i=1}^{n} i^3$. Nun transformieren wir die Summationsgrenzen $\rightarrow$

$$s_3(n) = \sum_{i=1}^{n} i^3 = \sum_{i=0}^{n-1} (i+1)^3 = \sum_{i=0}^{n-1} (i^3 + 3i^2 + 3i + 1)$$

gleiche Glieder weg ergibt

$$n^3 = 0^3 + \sum_{i=0}^{n-1} (3i^2 + 3i + 1) = 3 \cdot \left[s_2(n) - n^2 \right] + 3\frac{n(n-1)}{2} + n$$

$$\Rightarrow s_3(n) = \frac{n^3}{3} + n^2 - \frac{1}{2}n^2 + \frac{1}{2}n - \frac{n}{3} = \underline{\underline{\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}}}$$

$\Rightarrow$ Idee/Trick: Nutzen 2 spez. Summen, und zwar so, daß sich ihre meisten Glieder gegenseitig aufheben.

* weitere Beispiele:

Lemma 6.2: Für $F(n) = \sum_{i=0}^{n} 2^i$ gilt $F(n) = 2^{n+1} - 1$.

Beweis:

$$2F(n) = \sum_{i=1}^{n+1} 2^i$$

$$2F(n) - F(n) = F(n) = 2^{n+1} - 1$$

■

Lemma 6.3: Für $G(n) = \sum_{i=1}^{n} i \cdot 2^i$ gilt $G(n) = n \cdot 2^{n+1} - 2^{n+1} + 2$.

Beweis:

$$G(n) = 2G(n) - G(n) = \left[1 \cdot 2^2 + 2 \cdot 2^3 + \ldots + n \cdot 2^{n+1} \right] - \left[1 \cdot 2 + 2 \cdot 2^2 + \ldots + n \cdot 2^n \right]$$

$$= n \cdot 2^{n+1} - \underbrace{(1 \cdot 2 + 1 \cdot 2^2 + \ldots + 1 \cdot 2^n)}_{\text{L. 6.2: } 2^{n+1} - 2} = (n-1) \cdot 2^{n+1} + 2$$

■

Lemma 6.4: Für $G(n) = \sum_{i=1}^{n} i \cdot 2^{n-i}$ gilt $G(n) = 2^{n+1} - 2 - n$.

Beweis: $G(n) = 2G(n) - G(n) = \sum_{i=1}^{n} i \cdot (2^{n+1-i} - 2^{n-i})$

$= 1 \cdot (2^n - 2^{n-1}) + 2 \cdot (2^{n-1} - 2^{n-2}) + ... + n \cdot (2-1)$

$= 2^n + 2^{n+1} + ... + 2 - n = 2^{n+1} - 2 - n$ ■

* rekursive Beziehungen:

Fibonacci $\rightarrow$ $F(n) = F(n-1) + F(n-2), \quad n > 2; \quad (*)$
$F(2) = F(1) = 1$

Frage: $\exists$ geschl. Darstellung? Wenn ja, so viel Ressourcenersparnisse.

(1) Intelligentes „Raten“:

Fibonacci $\Rightarrow$

- da $F(n) = F(n-1) + F(n-2) \Rightarrow F(n) = c \cdot 2^n$ als Vermutung
  setzen in (*) ein $\Rightarrow$ $c \cdot 2^n = c \cdot 2^{n-1} + c \cdot 2^{n-2}$
  $\Rightarrow$ keine Lösung, da $2^n > c \cdot 2^{n-1} + c \cdot 2^{n-2}$ für $n \geq 2$

- 2. Versuch mit anderen Expon.-fkt, aber mit kleinerer Basis $\rightarrow$ $F(n) = a^n$
  setzen in (*) ein $\Rightarrow$ $a^n = a^{n-1} + a^{n-2}$ bzw. $a^2 = a + 1$

$\Rightarrow$ 2 Lösungen $a_{1,2} = \frac{1}{2} \pm \sqrt{\frac{1+4}{4}} = \frac{1}{2} \pm \sqrt{\frac{5}{4}} = \frac{1}{2} \pm \frac{1}{2}\sqrt{5}$

Wenn 2 Lösungen, so auch deren Linearkombination:

$c_1 \cdot a_1^n + c_2 \cdot a_2^n$

c_1, c_2 aus $F(2) = F(1) = 1 \Rightarrow c_1 = \frac{1}{\sqrt{5}}; \quad c_2 = -\frac{1}{\sqrt{5}}$

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

(2) Divide–and–Conquer Beziehungen:

Divide–and–Conquer Algorithmen sehr häufig und auch recht elegant;
z.B. beim Sortieren; 3 Schritte $\Rightarrow$

Divide–Schritt:	Input in kleine Teilmengen aufgeteilt. {Folge in 2 Teilfolgen}
Conquer–Schritt:	Problem für jede der Teilmengen gelöst. {Teilfolgen werden sortiert}
Merge–Schritt:	Teillösungen zur Gesamtlösung zusammengemischt. {sort. Teilfolgen zur sort. Folge}

allgemein:

Divide–Schritt	$\Rightarrow$	a Teilmengen(-probleme)
Conquer–Schritt	$\Rightarrow$	jede hat Aufwand $\frac{n}{b}$ des Ausgangsproblems
Merge–Schritt	$\Rightarrow$	$c \cdot n^k$

$\Rightarrow \quad T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^k$ für a, b, c, k = const.

vereinfach. Ann.: $n = b^m$, $b > 1$ sei Integer $\Rightarrow$ $\frac{n}{b}$ ist immer Integer

sehen einige Schritte an $\rightarrow$

$$T(n) = a \cdot \left[a \cdot T\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^k \right] + c \cdot n^k$$

$$= a \cdot \left\{ a \cdot \left[a \cdot T\left(\frac{n}{b^3}\right) + c\left(\frac{n}{b^2}\right)^k \right] + c\left(\frac{n}{b}\right)^k \right\} + c \cdot n^k = \ldots$$

Nehmen an, daß $T(1) = c$. Anderenfalls würde sich Endergebnis um eine Konstante ändern.

$$\Rightarrow \quad T(n) = c \cdot a^m + c \cdot a^{m-1} \cdot b^k + c \cdot a^{m-2} \cdot b^{2k} + \ldots + c \cdot b^{mk}$$

$$\Rightarrow \quad T(n) = c \cdot \sum_{i=0}^{m} a^{m-i} \cdot b^{ik} = c \cdot a^m \cdot \underbrace{\sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i}$$

geom. Reihe $\rightarrow$ 3 Fälle

<u>Fall 1:</u> $a > b^k$

$\rightarrow$ Summe konvergiert bei $m \rightarrow \infty$:

$\Rightarrow \quad T(n) = O(a^m)$

wegen $n = b^m \quad \rightarrow \quad m = \log_b n$ bzw. $a^m = a^{\log_b n} = n^{\log_b a}$

$\Rightarrow \quad T(n) = O(n^{\log_b a})$

<u>Fall 2:</u> $a = b^k$

$\rightarrow$ Summe ist gleich $m + 1$

$\Rightarrow \quad T(n) = O(a^m \cdot m)$

wegen $a = b^k$ gilt $\log_b a = k$ und $m = O(\log n)$

$\Rightarrow \quad T(n) = O(n^k \cdot \log n)$

Fall 3: $a < b^k$

$\rightarrow$ Summe ist $\dfrac{\left(\dfrac{b^k}{a}\right)^{m+1} - 1}{\left(\dfrac{b^k}{a} - 1\right)}$

$\Rightarrow \quad T(n) = \mathrm{O}\left(a^m \cdot \left(\dfrac{b^k}{a}\right)^m\right) = \mathrm{O}\left(b^{k \cdot m}\right) = \mathrm{O}\left(n^k\right) \quad \Rightarrow$

Satz 6.1:

Für die Lösung der Rekurrenzgleichung $T(n) = a \cdot T\left(\dfrac{n}{b}\right) + c \cdot n^k$

mit ganzz. Konstanten $a \geq 1$ und $b \geq 1$ sowie Konstanten $c, k > 0$ gilt

$$T(n) = \begin{cases} \mathrm{O}(n^{\log_b a}) & \text{bei} \quad a > b^k; \\ \mathrm{O}(n^k \log n) & \text{bei} \quad a = b^k; \\ \mathrm{O}(n^k) & \text{bei} \quad a < b^k. \end{cases}$$

Bem.: $a = 2b \quad \Rightarrow \quad T(n)=\mathrm{O}(n^2)$!

Anwendungsbeispiele:

1) ***Sortieren*** von n Elementen (Namen, Zahlen, …)

Algorithmus *Sortiere* (*Liste*);
IF $n > 1$
THEN BEGIN
Sortiere (*1. Listenhälfte*);
Sortiere (*2. Listenhälfte*);
Mische (*1. und 2. Listenhälfte*)
END;

$\Rightarrow \quad T(n) = 2 \cdot T\left(\dfrac{n}{2}\right) + \underbrace{c \cdot n}$ mit $T(1) = c$

Aufwand für Mischen ~ n

$\rightarrow$ $a = 2$; $b = 2$; $c = c$; $k = 1$

$\Rightarrow$ $\boxed{T(n) = \mathrm{O}(n \cdot \log n)}$

2) ***Multiplikation*** von zwei n–stelligen Zahlen

Algorithmus 1:

$$\begin{array}{rrrrrrr} 1 & 2 & 3 & \cdot & 4 & 5 & 6 \\ \hline & & & 4 & 9 & 2 & \\ & & & 6 & 1 & 5 & \\ & & & & 7 & 3 & 8 \\ \hline & & & 5 & 6 & 0 & 8 & 8 \end{array}$$

$\Rightarrow$ n Zeilen zu add.; jede Zeile n oder $n+1$ Ziffern;
je Zeile n Multiplikationen

$\rightarrow$ $n \cdot n = n^2$ Operationen: $T(n, \text{A1}) = \text{O}(n^2)$

Algorithmus 2: Suchen einen besseren A.!

Gehen von folgender Betrachtung aus $\rightarrow$

$$\overbrace{A \quad B}^{n} \times \overbrace{C \quad D}^{n} =$$
$$= 10^{\frac{n}{2}} \cdot A \times \left(10^{\frac{n}{2}} C + D \right) + B \times \left(10^{\frac{n}{2}} C + D \right) \quad \Rightarrow$$

$$\left.\begin{array}{ll} 10^{n}: & A \times C \\ 10^{\frac{n}{2}}: & A \times D + B \times G \\ 10^{0}: & B \times D \end{array}\right\} \; 4 \text{ Mult.} + 4 \text{ Add.}$$

$$T(n) = 4 \cdot T\left(\frac{n}{2}\right) + c \cdot n = \text{O}(n^2) \quad \Rightarrow \quad \text{Fehlanzeige!}$$

aber:

$10^{\frac{n}{2}}$: $(A+B) \cdot (C+D) - AC - BD$ $\Rightarrow$ 3 Multiplikationen + 5 Additionen

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + c \cdot n = \text{O}(n^{\log_2 3}) = \text{O}(n^{1.58496}) \qquad \left\{ \log_b a = \frac{\ln a}{\ln b} \right\}$$

1 2 3 $\cdot$ 4 5 6 $\Rightarrow$

$A \cdot G$:	400
$(A+B) \cdot (C+D) - AC - BD$:	148
$B \cdot D$:	1288
	56088

Bem.: n muß gerade sein $\rightarrow$ 0 1 2 3 $\cdot$ 0 4 5 6 $\Rightarrow$ n = 4 und $^{n}/_{2} = 2$

$\Rightarrow$ nach $A \cdot G = 4$ alle folg. Zeilen um $2 = \frac{n}{2}$ Stellen nach hinten

6.3. Suchalgorithmen

* Suche nach best. Datum / Daten → häufige Operation in Programmen

Suchverfahren := A., der für einen best. Schlüssel (das Suchargument) die zugehörige Inf., den eigentlich interess. Inhalt ermittelt, sofern ein Obj. mit diesem Schlüssel ex.; anderenfalls wird die Suche erfolglos abgebrochen.

formale Def. des Suchproblems:

Gegeben sei eine Folge $S = (s_1, s_2, \ldots, s_n)$ von n Objekten s_1 bis s_n. Jedes Objekt besteht aus einem Schlüssel (Namen) $s_i \cdot key$ und bestimmten Informationen, $i = 1(1)n$.

Die Schlüssel stammen aus einer Vielfalt U.
Gegeben ist weiter ein Element $z \in U$.
Gesucht ist ein i_0 derart, daß $s_{i_0} \cdot key = z$.

* ***lineare Suche*:**

Keine Inf. über Ordnung der Schlüssel ⇒ sequentielles Durchlaufen der Folge

$$\Rightarrow \quad T^*(n) = n;\ T_*(n) = 1;\ \overline{T}(n) = \frac{n+1}{2}$$

Im weiteren: $U = \mathbb{R}^1$ mit lin. Ordnung. Schreiben x_i statt $s_i \cdot key$.

* ***binäre Suchverfahren*:**

binäre Suche ist für Algor., was das Rad in der Mechanik;

Grundidee: Durch 1 Frage ca. ½ des Suchraumes verwerfen.

a) Reine binäre Suche:

Problem:

Sei $S = (x_1, \ldots, x_n)$ eine Folge reeller Zahlen mit der Eigenschaft $x_1 \le x_2 \le \ldots \le x_n$. Zu gegebenem reellem z ist ein Index i mit $x_i = z$ auszugeben, wenn z in der Folge ist, sonst ist 0 auszugeben.

$\Rightarrow \quad i = \min \{k: x_k = z\} \vee 0$;

Als Grundidee ⇒ sehe ein „mittleres“ El. $x_{\lceil n/2 \rceil}$ oder $x_{\lceil n/2 \rceil + 1}$ an;

$$\left.\begin{array}{l} \text{ist } z \text{ kleiner, so alle } x_i \text{ mit } i \ge \lceil n/2 \rceil \text{ weg;} \\ \text{ist } z \text{ gleich, so Ende } \{z := \lceil n/2 \rceil\}; \\ \text{ist } z \text{ größer, so alle } x_i \text{ mit } i \le \lceil n/2 \rceil \text{ weg;} \end{array}\right\} \Rightarrow$$

Algorithmus ***binäre_Suche***(*X*, *n*, *z*);

Input: *X* – sortiertes Feld mit Komponenten *x*[1] bis *x*[*n*]
z – Suchschlüssel

Output: *Position* – Index *i* mit *x*[*i*] = *z* oder 0, wenn solch Index nicht existiert

```
BEGIN
    Position := Finde(z, 1, n)
END;

FUNCTION Finde(z, Links, Rechts : Integer): Integer;
VAR Mitte : Integer;
BEGIN
    IF Links = Rechts
        THEN IF x[Links] = z
                THEN Finde := Links
                ELSE Finde := 0
        ELSE BEGIN
                Mitte := ⌊½ * (Links + Rechts)⌋;
                IF z < x[Mitte]
                    THEN Finde := Finde(z, Links, Mitte–1)
                    ELSE Finde := Finde(z, Mitte, Rechts)
             END
END;
```

Komplexität: Basisoperation sei ein Vergleich $\Rightarrow$

$$T(n) = 1 + T\left(\frac{n}{2}\right) \quad \text{mit} \quad T(1) = 1$$

Satz 6.1. $\Rightarrow$ $a = 1 \quad b = 2 \quad c = 1 \quad k = 0 \quad \Rightarrow \quad a = b^k$

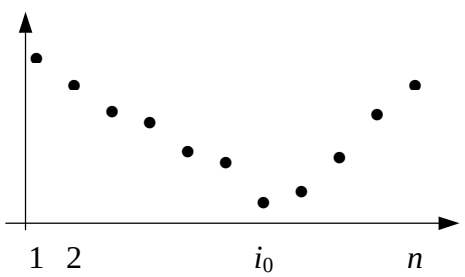
$\Rightarrow$ $\boxed{T(n) = \mathrm{O}(\log n)}$

b) Binäre Suche des Minimums einer unimodalen Folge:

Problem:

Sei $S = (x_1, \ldots, x_n)$ derart, daß $x_1 > x_2 > \ldots > \underline{x_{i_0}} < x_{i_0+1} < \ldots < x_{n-1} < x_n$ für ein i_0.
Bestimme den Index i_0.

unimodale Folge z.B.



wichtig für viele Minimierungsprobleme, z.B. folg. Lagerhaltungsproblem:

- 1 Lager, zuf. Bedarf entsprechend Verteilungsgesetz

$$B = \begin{pmatrix} 0 & 1 & \dots & n \\ p_0 & p_1 & \dots & p_n \end{pmatrix}$$

- Vorrat $x \in \mathbb{N}$
- Kosten: $k(x,b) = \begin{cases} h \cdot (x-b), & x \geq b; \\ g \cdot (b-x), & x < b; \end{cases}$

Kostenerwartungswert:

$$K(x) := E[k(x,B)] = \sum_{b=0}^{n} p_b \cdot k(x,b) =$$

$$= h \cdot \sum_{b=0}^{n} p_b \cdot (x-b) + g \cdot \sum_{b=x+1}^{n} p_b \cdot (b-x)$$

- gesucht: x^* mit $K(x^*) \leq K(x)$ für $x \in \mathbb{N}$

Lösung 1:

Berechne $K(x)$ für $x = 1 \dots n$ und wähle x^*.

Lösung 2:

Binäre Suche nach folg. Grundidee:

Berechne Fkt.-werte für „mittl." El. $x_{\lfloor n/2 \rfloor}$ und $x_{\lfloor n/2 \rfloor + 1}$;

ist $K\left(x_{\lfloor n/2 \rfloor}\right) < K\left(x_{\lfloor n/2 \rfloor + 1}\right)$, so ist Min. in $[1,\dots,\lfloor n/2 \rfloor]$, also links von $x_{\lfloor n/2 \rfloor + 1}$;

ist $K\left(x_{\lfloor n/2 \rfloor}\right) > K\left(x_{\lfloor n/2 \rfloor + 1}\right)$, so rechts von $x_{\lfloor n/2 \rfloor}$;

Komplexität: Basisoperation = Berechnung eines Funktionswertes;

$\Rightarrow$ $T(n) = 2 + T(n/2) \rightarrow a = 1 \quad b = 2 \quad c = 2 \quad k = 0 \rightarrow a = b^k$

$\Rightarrow$ $T(n) = O(\log n)$

* Bsp.: Lagerhaltungsproblem

$n = 10 \quad p_b = 1/10$ für $b = 1(1)10 \quad h = 1 \quad g = 3$

$\Rightarrow$

X	0	1	2	3	4	5	6	7	8	9	10
$K(x)$	16.5	13.5	10.9	8.7	7.0	5.5	4.5	3.9	3.7	3.9	4.5

Bei binärer Suche $\rightarrow$

$L = 0, R = 10 \quad \rightarrow \quad$ *Mitte* = 5, so daß $K(5) = 5.5$ und $K(6) = 4.5$ zu berechnen sind
$K(5) > K(6) \quad \Rightarrow$

$L = 6, R = 10 \quad \rightarrow \quad$ *Mitte* = 8, so daß $K(8) = 3.7$ und $K(9) = 3.9$ zu berechnen sind
$K(8) < K(9) \quad \Rightarrow$

$L = 6, R = 8 \quad \rightarrow \quad$ Mitte = 7, so daß $K(7) = 3.9$ zu berechnen ist
$K(7) > K(8) \quad \Rightarrow$

$L = R = 8 \quad \rightarrow \quad$ Minimum wird unter $x = 8$ erreicht
$\Rightarrow$ 5 Funktionswerte statt 11

Bem.: Analoges Vorgehen bei Bestimmung des Minimums (Maximums) einer Funktion oder des Null–Punktes einer stetigen Funktion und bei ähnlichen Problemen.