- Achtung: Groß–O definiert keine totale Ordnungsrelation auf der Menge aller Funktionen $\mathbb{R} \to \mathbb{R}$!
- Beweis: Es gibt positive Funktionen f und g so, dass

$$f \notin O(g) \text{ und auch } g \notin O(f) .$$

Wähle zum Beispiel $f(n) = \sin(n) + 1$ und $g(n) = \cos(n) + 1$.

Beweishilfe für die Zuordnung in eine Komplexitätsklasse

Lemma:

Es gilt:

$$(1) \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c > 0 \Rightarrow O(f(n)) = O(g(n))$$

$$(2) \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f(n)) \subset O(g(n))$$

Insbesondere hat man dann in Fall (1): $f \in \Theta(g)$

Und in Fall (2): $f \in O(g)$

Wie überprüft man diesen Limes?

Satz 7.44. (3. Regel von de l'Hôpital: "$x \to \infty$") *Seien f und g auf dem Intervall $[a, \infty[$ dinfferenzierbar und es gelte* $\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) = 0$ *(bzw. $= \infty$). Es existiere* $\lim_{x\to\infty} \frac{f'(x)}{g'(x)} =: L$. *Dann existiert auch* $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ *und ist gleich L. Kurz:*

$$\lim_{x\to\infty} \frac{f'(x)}{g'(x)} = \lim_{x\to\infty} \frac{f(x)}{g(x)}.$$

Einige wichtige Funktionenklassen

Klasse	Bezeichnung	Beispiele
O(1)	Konstante Funtionen	17
O(log n)	Logarithmische Funktionen	$\log_2 n$, $\log_3 n$
O(n)	Lineare Funktionen	5n - 3
O(n log n)	nlog n-wachsende Funktionen	$7n \cdot \log_2 n$
O(n^2)	Quadratische Funktionen	$3n^2+1$
O(n^3)	Kubische Funktionen	n^3, $4n^3 + n - 3$
O(n^k), k fest	Polynomielle Funktionen	$8n^5 - 2n^3 + n^2 + 2n$
O(2^n)	Exponentielle Funktionen	2^n, 3^n

- Definition "polynomielle / exponentielle Zeit" :
  wir sagen, ein Algorithmus A mit Komplexität $f(n)$ braucht höchstens polynomielle Rechenzeit (*is in polynomial time*), falls es ein Polynom $P(n)$ gibt, so dass $f(n) \in O\big(P(n)\big)$.
  A braucht höchstens exponentielle Rechenzeit (*exponential time*), falls es eine Konstante $a \in \mathbb{R}^+$ gibt, so dass $f(n) \in O\big(a^n\big)$.

Problemgröße und Rechenzeit

- Wenn wir annehmen, dass jedes Element einer Eingabe der Größe n in 1 msec verarbeitet werden kann, dann lassen sich Probleme folgender Größe lösen:

Laufzeit	in 1 Sekunde	in 1 Minute	in 1 Stunde
$T(n) = n$	1000	60 000	3 600 000
$T(n) = n \log n$	140	4 895	204 094
$T(n) = n^2$	31	244	1 897
$T(n) = n^3$	10	39	153
$T(n) = 2^n$	9	15	21

- Wenn wir einen doppelt so schnellen Rechner kaufen, dann lässt sich anstatt eines Problems der Größe N in der gleichen Zeit ein Problem folgender Größe lösen:

Laufzeit $f(n)$	Neue Problemgröße
n	$2N$
$n \log n$	
n^2	
n^3	
2^n	

- Wenn wir einen doppelt so schnellen Rechner kaufen, dann lässt sich anstatt eines Problems der Größe N in der gleichen Zeit ein Problem folgender Größe lösen:

Laufzeit $f(n)$	Neue Problemgröße
n	$2N$
$n \log n$	etwas weniger als $2N$
n^2	$1.41N$
n^3	$1.26N$
2^n	$N+1$

- *Fazit: wir brauchen schnelle Algorithmen, damit sich Moore's Law von der CPU auch auf die beherrschbaren Problemgrößen überträgt!*

Allg. Bestimmung des Zeitaufwands mit Groß-O

- Sei A ein Programmstück, dann ist der Zeitaufwand *T*(A) :
  - A ist einfache Anweisung oder arithm./log. Ausdruck →

    $$T(A) = \text{const} \in O(1)$$

  - A ist Folge von Anweisungen → Additionsregel anwenden
  - A ist `if`-Anweisung →

    (a) `if cond: B` → T(A) = T(cond) + T(B)

    (b) `if cond: B else: C` → T(A) = T(cond) + max(T(B), T(C))

  - A ist eine Schleife (while, for, ...) →

    $$T(A) = \sum_i T(\text{Schleifendurchlauf mit } i)$$

    oft einfach

    $$T(A) = \#\ \text{Durchläufe} \cdot T(\text{worst-case Schleifendurchlauf})$$

  - A ist Rekursion → später

- Beispiel: geschachtelte Schleifen

```
for i = 0 .. n-1:
  for j = 0 .. n-1:
    print i*j
```

  - Analyse von innen nach außen

Beispiele zur Laufzeitabschätzung

Problem: *prefixAverages1(X)*

Eingabe**:** Ein Array X von n Zahlen

Ausgabe: Ein Array A von Zahlen, so daß gilt: A[i] ist das arithmetische Mittel der Zahlen X[0], …, X[i]

Algorithmus :

```
for i in range(0,n):
    a = 0                          → O(1)
    for j in range(0,i+1):
        a += X[j]                  → O(1)
    A[i] = a / (i + 1)             → O(1)
return A
```

$i \cdot O(1) = O(i) \subseteq O(n)$, da $i \leq n$

$O(1) + O(1) + O(n) = O(n+2) = O(n)$

$n \cdot O(n) = O(n^2)$

Exkurs

- Das eben gestellte Problem kann man auch effizienter lösen

Algorithmus *prefixAverages2(X)*

```
s = 0.0
for i in range(0,n):
    s += X[i]
    A[i] = s / (i + 1)
return A
```

$O(1)$

$n \cdot O(1) = O(n)$

Average-Case-Komplexität

- Nicht leicht zu handhaben, für die Praxis jedoch relevant
- Sei $p_n(x)$ die Wahrscheinlichkeit, mit der Eingabe x mit der Länge n auftritt
- Mittlere (erwartete) Laufzeit:

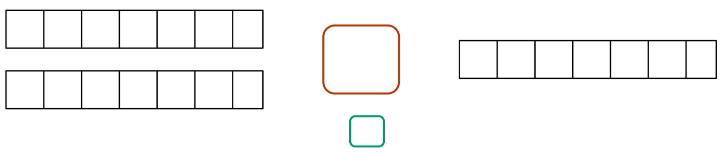
$$\bar{T}(n) = \sum_{x,\ |x|=n} T(x) p_n(x)$$

- Wichtig:
  - Worüber wird gemittelt ?
  - Sind alle Eingaben der Länge n gleichwahrscheinlich ?
- Oft: Annahme der Gleichverteilung aller Eingaben x der Länge n
  - Dann ist $p_n(x) \equiv 1/N$, N = Anzahl aller mögl. Eingaben der Länge n

$$\bar{T}(n) = \frac{1}{N} \sum_{x,\ |x|=n} T(x)$$

Beispiel: serieller Addierer

- Grundstruktur:
  - Eine kleine Schaltung, die 3 Bits addieren kann
  - 2 Eingaberegister für Binärzahlen
  - 1 Ausgaberegister für das Ergebnis
  - Taktgeber zum Durchschieben der Register
  - Start der Addition beim *least significant bit* (LSB)



- Vorteil: sehr wenig Hardware-Aufwand nötig
- Nachteil: hoher (worst-case) Aufwand

- Betrachte nun die spezielle Aufgabe:
  - Zahl i in Binärdarstellung der Länge n gegeben (also $0 \le i \le 2^{n-1}$)
  - Erhöhe i um 1
- Taktzahl (Anzahl Bitwechsel) = Anzahl der Einsen am Ende der Binärdarstellung von i, plus 1
- Worst Case: n+1 Takte
  - Beispiel: Addition von 1 zu 111. . . 1
- Average Case:
  - Wir nehmen eine Gleichverteilung auf der Eingabemenge an
  - Es gibt 2^{n-k} Eingaben der Form (x,..., x, 0, 1,..., 1) wobei k-1 Einsen am Ende stehen → Laufzeit = k Takte
  - Hinzu kommt die Eingabe i = 2^n-1 → Laufzeit = n+1 Takte

- Die *average-case Rechenzeit* $\bar{T}(n)$ beträgt:

$$\bar{T}(n) = \frac{1}{2^n}((n+1) + \sum_{1 \le k \le n} 2^{n-k}k)$$

- Es gilt: $\sum_{k=1}^{n} 2^{n-k}k = 2^{n+1} - 2 - n$
- Demnach ist

$$\bar{T}(n) = 2^{-n}(2^{n+1} - 2 - n + (n+1)) = 2 - 2^{-n}$$

- Es genügen also im Durchschnitt 2 Takte, um eine Addition von 1 zu einer beliebig großen Zahl durchzuführen!

The Maximum Subarray Problem

- Motivationsbeispiel:
  - Ein Club hat einen Eingang und einen Ausgang
  - An beiden Türen gibt es je einen Sensor, der die Anzahl Personen zählt, die in einem 5-Min-Zeitintervall hindurchgehen
  - Dies ergibt z.B. folgende Zu- und Abgänge:

    1 2 -3 3 -1 0 -4 0 -1 -4 2 4 1 1 3 1 0 -2 -3 -3 -2 3 1 1 4 5 -3 -2 -1 ...
  - Die Frage ist nun: in welchem 1-stündigen Zeitintervall steigt die Zahl der Personen im Club am stärksten an?

- Problemstellung: Finde ein Index-Paar (i, j) in einem Array a[1..n] von Zahlen, für das $f(i, j) = a_i + \ldots + a_j$ maximal ist

- Der naive Algorithmus:
  - Berechne alle Werte $f(i, j)$, $1 \leq i \leq j \leq n$, und ermittle davon den maximalen f-Wert
  - Alle $f(i, j)$ berechnen geht mit 2 geschachtelten Schleifen, eine für i=1,…,n , eine für j=i,…,n
  - Offensichtlich genügen zur Berechung von einem $f(i, j)$ genau j–i viele Additionen
  - Der Algoritmus startet mit max = f(1,1) und aktualisiert max wenn nötig

Analyse des naiven Algorithmus'

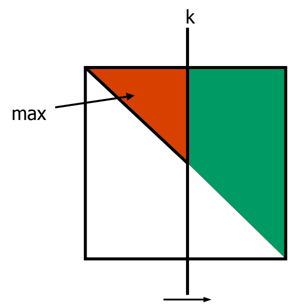
- Klar ist: Anzahl Additionen = Komplexität des Algorithmus
- #Additionen:

$$\begin{aligned} A_1(n) &= \sum_{1 \le i \le n} \sum_{i \le j \le n} (j-i) \\ &= \sum_{1 \le i \le n} \sum_{1 \le k \le n-i} k \\ &= \sum_{1 \le i \le n} \sum_{1 \le k \le i} k \\ &= \sum_{1 \le i \le n} i(i+1)/2 \\ &= \frac{1}{2}\Big(\sum_{1 \le i \le n} i^2 + \sum_{1 \le i \le n} i \Big) \\ &= \frac{1}{2}\big(\tfrac{1}{6}(n-1)n(2(n-1)+1) + \tfrac{1}{2}(n+1)n\big) \\ &= \tfrac{1}{6}n^3 - \tfrac{1}{6}n \end{aligned}$$

- Zusammen: $T_1(n) = A(n) \in O\left(n^3\right)$

Der Kadane-Algorithmus

- Verwende das Scanline-Prinzip (wichtige Algorithmentechnik!)
  - Idee: betrachte ein 2D-Problem nicht insgesamt, sondern immer nur auf einer Gerade, die über die Ebene "gleitet" → *Scanline*
  - Löse das Problem immer nur auf dieser Scanline, und aktualisiere die Lösung, wenn die Scanline beim nächsten interessanten "Ereignis" ankommt
- Hier: Wir verwalten nach dem Lesen von a_k in max den größten Wert von f(i, j) aller Paare (i, j) für $1 \le i \le j \le k$.
- Für k=1 ist max = a_1



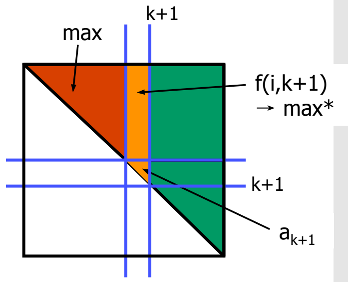
- Wenn nun a_{k+1} gelesen wird, soll max aktualisiert werden
- Dazu bestimmen wir

$$\max_i\{f(i,k+1)\} = \max_i\{g(i)\}$$

wobei

$$g(i) := a_i + \ldots + a_{k+1}$$

(ähnlich der g-Werte vom rekursiven Algorithmus)



- Deshalb verwalten wir zusätzlich

$$\max^* := \max_{1\le i\le k}\{g(i) \mid \text{mit } g(i) = a_i + \ldots + a_k\}.$$

Aktualisierung und Analyse

- Sei nun a_{k+1} gelesen. Wir erhalten die neuen g-Werte

$$g_{\text{neu}}(i) = g_{\text{alt}}(i) + a_{k+1}, \text{für } 1 \le i \le k$$
$$g_{\text{neu}}(k+1) = a_{k+1}$$

- Also: $\max^*_{\text{neu}} = \max\{\max^*_{\text{alt}} + a_{k+1}, a_{k+1}\}$
- Für $\max_{\text{neu}}$ kommen folgende Paare (*i*,*j*) in Frage:

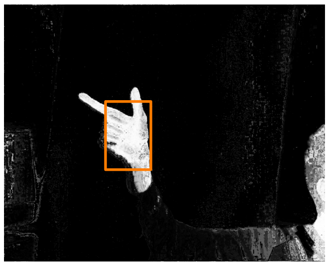
$$1 \le i \le j \le k \quad \rightarrow \quad \text{max. steht in } \max_{\text{alt}}$$
$$\left.\begin{array}{l} 1 \le i \le k\,,\, j = k+1 \\ i = k+1\,,\, j = k+1 \end{array}\right\} \rightarrow \quad \text{max. steht in } \max^*_{\text{neu}}$$

- Also: $\max_{\text{neu}} = \max\{\max_{\text{alt}}, \max^*_{\text{neu}}\}$
- Bei der Verarbeitung von a_k, $2 \le k \le n$, genügen also 3 Operationen, demnach ist

$$T_4(n) = 3n - 3 \in O(n)$$

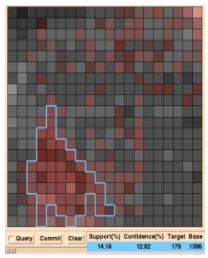
Anwendungsbeispiel: einfaches Hand-Tracking

- Gegeben ein Grauwert-Bild
  - Z.B. Hautfarbe-Wahrscheinlichkeit
- Gesucht: das hellste Rechteck
- Ansatz:
  - Ziehe von allen Grauwerten den Mittelwert ab
  - Array hat jetzt Werte in [-0.5,+0.5]
  - Löse das Maximum-Subarray-Problem im 2D



Anwendungsbeispiel: einfaches Data-Mining

- Ort des Geschehenes: eine Bank
  - Vergibt Kredite, manche davon werden nicht zurückgezahlt
- Gegeben eine 3D-Tabelle mit
  - Zeile = Alter des Kunden
  - Spalte = Höhe des Guthabens des Kunden (bei der Bank, oder Schufa)
  - Schicht = Höhe des gewünschten Kredites
  - Eintrag = Häufigkeit, mit der Kredit zurückgezahlt wird



- Aufgabe:
  - Anfrage eines Kunden nach einem Kredit bestimmter Höhe
  - Ziel: Entscheidung, ob Kredit gewährt werden soll
- Mögliche (simple) Lösung:
  - Ziehe Mittelwert aller Einträge von diesen ab
  - Berechne maximales Subarray (= Sub-Quader in der Tabelle)
  - Kunde erhält Kredit gdw (Alter,Guthaben,Kredit) ∈ Sub-Quader