- 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 $f(n) \in O(P(n))$.
  A braucht höchstens exponentielle Rechenzeit (*exponential time*), falls es eine Konstante $a \in \mathbb{R}^+$ gibt, so dass $f(n) \in O(a^n)$.

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

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):         → O(1)      i·O(1) = O(i) ⊆ O(n), da i ≤ n
        a += X[j]
    A[i] = a / (i + 1)             → O(1)
return A
```

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

n·O(n) = O(n²)

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)             O(1)
return A
```

n·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

- Taktzahl (Anzahl Bitwechsel) eines seriellen Addierers bei Addition von 1 zu einer in Binärdarstellung gegebenen Zahl i der Länge n, d.h. $0 \le i \le 2^{n-1}$
- Sie beträgt 1 plus der Anzahl der Einsen am Ende der Binärdarstellung von i
- 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 also:

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

- Es ist

$$\begin{aligned}
\sum_{1 \le k \le n} 2^{n-k} k &= n2^{n-n} + \ldots + 2 \cdot 2^{n-2} + 1 \cdot 2^{n-1} \\
&= \quad 2^0 + \ldots + 2^{n-3} + 2^{n-2} + 2^{n-1} \\
&\quad + 2^0 + \ldots + 2^{n-3} + 2^{n-2} \\
&\quad + 2^0 + \ldots + 2^{n-3} \\
&\quad \vdots \\
&\quad + 2^0 \\
&= (2^n - 1) + \ldots + (2^1 - 1) \\
&= 2^{n+1} - 2 - n
\end{aligned}$$

- 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!

Das Maxsummen-Problem

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

- Der naive Algorithmus:
  - Berechne alle Werte $f(i, j)$, $1 \le i \le j \le n$, 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 Berechnung 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'

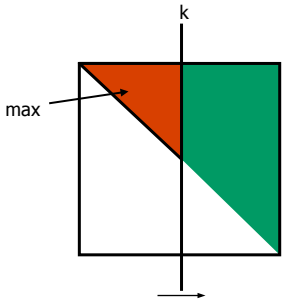
- Klar ist: Anzahl Additionen = Komplexität des Algorithmus
- #Additionen:

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

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

Der clevere 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



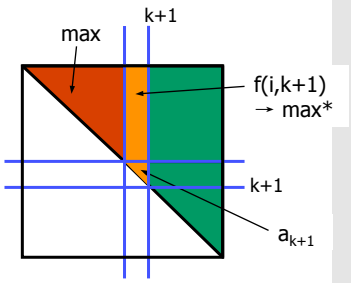
- Wenn nun a_{k+1} gelesen wird, soll max aktualisiert werden
- Dazu bestimmen wir

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

wobei

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

(ähnlich der g-Werte vom rekursiven Algorithmus)



- Deshalb verwalten wir zusätzlich

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

Akualisierung 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:

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

- 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

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