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·O(1) = O(n)

Average-Case-Komplexität

- Nicht leicht zu handhaben, für die Praxis jedoch relevant
- Sei $p_n(x)$ Wahrscheinlichkeitsverteilung, mit der Eingabe x mit 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.
  - Hinzu kommt die Eingabe $i = 2^n-1$, für die das Addierwerk n+1 Takte benötigt.

- 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 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. Als Rechenschritte zählen arithmetische Operationen und Vergleiche.
- Der naïve Algorithmus:
  Berechne alle Werte f(i, j), 1 ≤ i ≤ j ≤ n, und ermittle davon den maximalen f-Wert.
  Offensichtlich genügen zur Berechung von f(i, j) genau j–i viele Additionen.
  Der Algoritmus startet mit $\max \leftarrow f(1,1)$ und aktualisiert $\max$ wenn nötig.

Analyse des naïven Algorithmus'

- Es gibt j Paare der Form (·, j), nämlich (1, j), … , (j, j)
- #Vergleiche: $V_1(n) = \sum_{1 \le j \le n} j - 1 = n(n+1)/2 - 1$
- #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:

$$T_1(n) = V_1(n) + A_1(n) = \tfrac{1}{6}n^3 + \tfrac{1}{2}n^2 + \tfrac{1}{3}n - 1 \in O(n^3)$$

Der etwas bessere Algorithmus

- Beobachtung: Der naïve Ansatz berechnet $a_1 + a_2$ für f(1, 2), f(1, 3), ... , f(1, n), also (n–1)-mal.
- Besser geh's mit folgender Erkenntnis. Es gilt:

$$f(i, j+1) = f(i, j) + a_{j+1}$$

- Damit braucht man für alle f(i, ·)-Werte genau (n – i) Additionen.
- Aufwand:
  - #Vergleiche: $V_2(n) = V_1(n) = n(n+1)/2 - 1$
  - #Additionen:

$$A_2(n) = \sum_{1 \le i \le n} (n-i) = \sum_{1 \le k \le n-1} k = n(n-1)/2$$

  - #Zusammen:

$$T_2(n) = V_2(n) + A_2(n) = n^2 - 1 \in O(n^2)$$

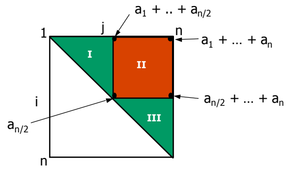
Divide-and-Conquer

- Annahme: $n = 2^k$
- Unterteilung der zu untersuchenden Paare (i, j) in drei Klassen:

I. $1 \le i \le j < n/2$

II. $1 \le i \le n/2 \le j \le n$

III. $n/2 < i \le j \le n$



- Wenn wir das Problem für die 3 Klassen gelöst haben, erhalten wir den „Gesamtsieger" in 3 Vergleichen
- Problem I und III sind vom gleichen Typ wie das Ausgangs-probelm → rekusiv mit dem gleichen Ansatz lösen

Analyse Divide-and-Conquer

- Für Problem II gibt es eine effiziente direkte Lösung. Betrachte:

$$g(i) = a_i + \ldots + a_{n/2} \text{ und } h(j) = a_{n/2+1} + \ldots + a_j$$

- Dann gilt:
  - $f(i, j) = g(i) + h(j)$
  - Um f zu maximieren reicht es aus, g und h einzeln zu maximieren:

$$\max_{i,j}\{f(i,j)\} = \max_{i,j}\{g(i) + h(j)\} = \max_{i}\{g(i)\} + \max_{j}\{h(j)\}$$

- Berechne nacheinander (wie bei Algo Nr 2)

$$\begin{aligned} g(n/2) &= a_{n/2}, \\ g(n/2-1) &= a_{n/2-1} + a_{n/2}, \\ &\ldots, \\ g(1) &= a_1 + \ldots + a_{n/2} \end{aligned}$$

- Max{g(i)} benötigt n/2-1 Additionen und n/2–1 Vergleiche
- Analog: max{h(j)} benötigt n/2 Additionen und n/2–1 Vergleiche
- Damit ergeben sich für Problem II insgesamt n-1 Additionen und n-2 Vergleiche, also insgesamt O(n) viele Operationen (obwohl die Klasse $O(n^2)$ Paare (i, j) enthält!)

- Für das Gesamtproblem ergibt sich folgende rekursive Gleichung für den Aufwand (3 Vergleiche um Maximum der Gruppen zu finden), n = 2^k:

$$\begin{aligned} T_3(1) &= 0 \\ T_3(2^k) &= 2T_3(2^{k-1}) + 2\cdot 2^k - 3 + 3 \\ &= 2(2T_3(2^{k-2}) + 2\cdot 2^{k-1}) + 2\cdot 2^k \\ &= 4T_3(2^{k-2}) + 2^{k+1} + 2^{k+1} \\ &= 4(2T_3(2^{k-3}) + 2\cdot 2^{k-2}) + 2^{k+1} + 2^{k+1} \\ &= 8T_3(2^{k-3}) + 2^{k+2} + 2^{k+1} + 2^{k+1} \\ &= \ldots \end{aligned}$$

- Nun raten wir die Lösung der Rekursionsgleichung

$$T_3(2^k) = \sum_{i=1}^{k} 2^{k+1}$$

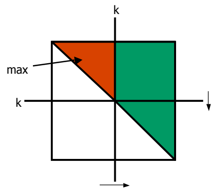
- ... und verifizieren die Vermutung mit einem Induktionsbeweis.

- Damit ist $T_3(2^k) = 2k2^k$

$$\begin{aligned} &= 2(\log n)\cdot n \\ &\in O(n \log n) \end{aligned}$$

Der clevere Algorithmus

- 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 ≤ i ≤ j ≤ k.
- Für k=1 ist max = a_1



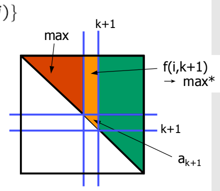
- Wenn nun a_{k+1} gelesen wird, soll max aktualisiert werden
- Dazu bestimmen wir

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

wobei

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

(ähnlich der g-Werte vom Divide-and-Conquer 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
  $g_{\text{neu}}(i) = g_{\text{alt}}(i) + a_{k+1}$, 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:
  $1 \le i \le j \le k$ (maximaler Wert $\max_{\text{alt}}$)
  $1 \le i \le k, j = k+1$ (maximaler Wert $\max^*_{\text{neu}}$)
- 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

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

Zusammenstellung der Ergebnisse

$$T_1(n) = 1/6n^3 + 1/2n^2 + 1/3n - 1$$
$$T_2(n) = n^2 - 1$$
$$T_3(n) = (2\log n - 1)n + 1$$
$$T_4(n) = 3n - 3$$

n	$T_1(n)$	$T_2(n)$	$T_3(n)$	$T_4(n)$
2^2	19	15	13	9
2^4	815	255	113	45
2^6	45759	4095	705	189
2^8	2829055	65535	3841	765
2^{10}	179481599	1048575	19457	3069
2^{15}	$> 5 \cdot 10^{12}$	$\approx 10^9$	950273	98301

- Es sollte sich das beruhigende Gefühl breitmachen, daß es sich lohnt, clever zu sein `;-)`