Kapitel 1

Einführung

Anhand des folgenden Problems soll deutlich gemacht werden, welche Schwierigkeiten beim Vergleich verschiedener Lösungsansätze auftreten können, um dann einige sinnvolle Kriterien festzulegen.

1.1 Problem (Auswahlproblem)
„Bestimme das k-t kleinste von n Elementen", z.B. für Zahlen: k-SELECT

gegeben: *Elemente $a_1, \ldots, a_n$ mit einer Ordnung $\leq$ sowie ein $k \in \{1, \ldots, n\}$*

gesucht: $a_{\pi(k)}$ *für eine Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ mit $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$*

1.2 Bemerkung
Natürliche Spezialfälle des Auswahlproblems mit festem k sind:

$$k = \begin{cases} 1 & \textit{Minimumssuche} \\ n & \textit{Maximumssuche} \\ \lfloor \frac{n}{2} \rfloor & \textit{Median} \end{cases}$$

Die Bestimmung des Mittelwerts von n Zahlen ist selbstverständlich kein Spezialfall des Auswahlproblems.

Wir diskutieren vier verschiedene Ansätze zur Lösung des Auswahlproblems und nehmen dabei an, dass die Elemente in einem Array $M[0], \ldots, M[n-1]$ gespeichert sind.

Ansatz A: Die konzeptionell einfachste Methode besteht darin, die Elemente des Arrays zunächst bezüglich $\leq$ nicht-absteigend zu sortieren und dann das k-te auszugeben.

Algorithmus 1: Auswahl nach Sortieren

```
sort(M)
print M[k − 1]
```

Um die Güte dieses Vorgehens beurteilen zu können, muss mindestens mal der verwendete Sortieralgorithmus bekannt sein. Möglicherweise hängt dessen Güte jedoch auch noch von der Eingabe und der Art der Elemente ab (vgl. Kapitel 2).

Ansatz B: Das Auswahlproblem lässt sich auch elementar, d.h. ohne Verwendung eines anderen Algorithmus, lösen. Statt alle Elemente zu sortieren, genügt auch die k-malige Bestimmung und Wegnahme eines Minimums. Das letzte davon ist das gesuchte Element.

Algorithmus 2: Wiederholte Minimumssuche

```
for i = 1, ..., k − 1 do M ← M \ {min M}
print min M
```

Im vorstehenden Pseudo-Code ist ein Spezialfall des Auswahlproblems, die Minimumssuche, als elementare Operation aufgeführt. Um die tatsächliche Komplexität besser beurteilen zu können, geben wir eine ausführlichere Implementation an. Darin wird das jeweils kleinste Element der Restfolge an die erste Stelle geholt, sodass schliesslich in $M[0], \ldots, M[k-1]$ die k kleinsten Elemente stehen.

Algorithmus 3: Wiederholte Minimumssuche (detailliert)

```
for i = 1, ..., k do
    m ← i − 1
    for j = i, ..., n − 1 do
        if M[j] < M[m] then m ← j
    a ← M[i];   M[i] ← M[m];   M[m] ← a
print M[k − 1]
```

1.3 Bemerkung
Wird die wiederholte Minimumssuche n mal ausgeführt, entspricht die Ausgabereihenfolge der Elemente einer vollständigen Sortierung der Elemente. `MinSort`

Ansatz C: Statt wie in Ansatz B immer das kleinste Element der Restfolge zu suchen, können wir auch alle Elemente durchgehen und für jedes testen, ob es unter den bisher betrachteten zu den k kleinsten gehört. Dies ist gerade dann der Fall, wenn das Element kleiner ist als das größte der k bisher kleinsten. Sind alle Element durchgetestet, ist das gröste der k kleinsten das gesuchte Element. Statt wiederholter Minimumssuchen in der (anfangs sehr langen) Restfolge führen wir also Maximumssuchen in einem Anfangsstück der Länge k aus.

Algorithmus 4: Aktualisierung einer vorläufigen Lösung

```
begin
    m ← maxpos(M,0,k-1)
    for i = k, ..., n − 1 do
        if M[i] < M[m] then
            a ← M[m];  M[m] ← M[i];  M[i] ← a
            m ← maxpos(M,0,k-1)
    print M[m]
end

int maxpos(array M, int l, r) begin
    m ← l
    for i = l + 1, ..., r do
        if M[i] > M[m] then m ← i
    return m
end
```

Ansatz D: Gibt es in der verwendeten Programmiersprache (d.h. in der zum Sprachumfang gehörigen Standardbibliothek) oder einem zur Verfügung stehen Paket bereits eine entsprechende Methode, kann einfach diese aufgerufen werden. Für die Beurteilung dieses Vorgehens ist dann allerdings detaillierte Kenntnis über das Verhalten der Methode erforderlich, da die (meist unbekannte) Implementation (und sei es nur im aktuellen Kontext, in dem das Auswahlproblem gelöst werden soll) sehr ineffizient sein könnte.

Beim Vergleich dieser vier Lösungsansätze stellt man schnell fest, dass es keinen eindeutig besten gibt. Die Beurteilung erfordert mehr Information: bei den Ansätzen A und D über die Implementation selbst, und in allen Fällen über typische Eingaben. Insbesondere das Verhältnis der Größen von n und k ist von Bedeutung (da die Laufzeit von Ansatz A nur von n abhängt, die von B und C aber auch von k), aber z.B. auch, ob Vergleiche und Umspeicherungen ähnlich schnell ausgeführt werden können.

Schon für einen Vergleich auf Basis der Ausführungszeiten stellen sich viele Detailfragen. Wird die Laufzeit etwa in Sekunden gemessen, lässt sie sich nicht für alle Eingaben im Vorhinein angeben und hängt zudem von zahlreichen Faktoren ab. Drei einfache Beispiele:

- In welcher Programmiersprache wurde implementiert?
- Auf welchem Rechner wird das Programm ausgeführt (Aufbau, Taktfrequenz, Speicherzugriffszeiten, etc.)?
- Wie sind die Daten gespeichert (Organisation, Medium, etc.)?

Die Beurteilung von Algorithmen und Datenstrukturen werden wir von diesen Faktoren weitgehend unabhängig machen, indem wir z.B. statt der Ausführungszeiten die Anzahl der elementaren Schritte zählen, die ein Algorithmus ausführt. Um vereinbaren zu können, was ein elementarer Schritt sein soll, müssen wir allerdings ein paar Festlegungen treffen, die nach Möglichkeit realistisch und trotzdem unabhängig von konkreten Rechnern sein sollten.

Um die Komplexität eines Verfahrens sinnvoll beurteilen zu können, führen wir daher zunächst ein Maschinenmodell ein, in dem Laufzeit und Speicherplatzbedarf hinreichend genau und auf standardisierte Weise gemessen werden können.

1.4 Definition (Random Access Machine)
Die Random Access Machine (RAM) ist ein abstraktes Maschinenmodell mit (siehe Abb. 1.1)

- *einer endlichen Zahl von Speicherzellen für das Programm,*
- *einer abzählbar unendlichen Zahl von Speicherzellen für Daten, (Speicheradressen aus* $\mathbb{N}_0$*),*

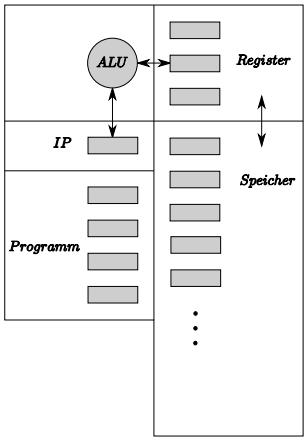

Abbildung 1.1: Aufbau der Random Access Machine

- *einer endlichen Zahl von Registern,*
- *einem Befehlszähler (spezielles Register) und*
- *einer arithmetisch-logischen Einheit (ALU).*

In Speicherzellen und Registern stehen wiederum natürliche Zahlen, die auch von der ALU verarbeitet werden können. Der Befehlszähler wird nach jeder Befehlsausführung um eins erhöht, kann aber auch mit einem Registerinhalt überschrieben werden. Als Anweisungen stehen zur Verfügung

- *Transportbefehle: Laden, Verschieben, Speichern,*
- *Sprungbefehle: bedingte und unbedingte,*
- *arithmetische und logische Verknüpfungen.*

Die Adressierung erfolgt direkt (Angabe der Speicherzelle) oder indirekt (Adressierung über Registerinhalt).

1.5 Bemerkung

1. *Mit den Sprungbefehlen sind alle Schleifentypen (***for***,* **while***,* **repeat-until***) und auch Rekursionen realisierbar.*

2. *Der Unterschied zur* Registermaschine *besteht in der Möglichkeit zur indirekten Adressierung, anders als bei der* Random Access Stored Program (RASP) Machine *sind Programm und Daten jedoch getrennt.*

kein von-Neumann Modell

Wir werden die Komplexität von Algorithmen vor allem durch zwei Größen beschreiben:

Laufzeit: Anzahl Schritte

Speicherbedarf: Anzahl benutzter Speicherzellen

Tatsächlich wäre selbst die genaue Anzahl der Schritte zu mühsam zu bestimmen. Wir müssten z.B. präzise angeben, auf welche Weise genau ein Wert aus dem Speicher über Register in die ALU kommt und weiterverarbeitet wird. Es soll uns aber reichen, dass Vorgänge dieser Art durch eine unbekannte, aber konstante Anzahl von Schritten realisiert werden können. Entsprechend werden wir konstante Faktoren in Laufzeiten und Speicherplatz weitgehend ignorieren und uns auf das asymptotische Wachstum der Komplexität im Verhältnis zur Größe der Eingabe konzentrieren.

1.6 Definition (Asymptotisches Wachstum)
Zu einer Funktion $f : \mathbb{N}_0 \to \mathbb{R}$ *wird definiert:*

(i) Die Menge

$$\mathcal{O}(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \; : \; \begin{array}{l} \text{es gibt Konstanten } c, n_0 > 0 \text{ mit} \\ |g(n)| \leq c \cdot |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

der Funktionen, die höchstens so schnell wachsen wie f.

(ii) Die Menge

$$\Omega(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \; : \; \begin{array}{l} \text{es gibt Konstanten } c, n_0 > 0 \text{ mit} \\ c \cdot |g(n)| \geq |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

der Funktionen, die mindestens so schnell wachsen wie f.

(iii) Die Menge

$$\Theta(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \; : \; \begin{array}{l} \text{es gibt Konstanten } c_1, c_2, n_0 > 0 \text{ mit} \\ c_1 \leq \frac{|g(n)|}{|f(n)|} \leq c_2 \text{ für alle } n > n_0 \end{array} \right\}$$

der Funktionen, die genauso schnell wachsen wie f.

(iv) Die Menge

$$o(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \; : \; \begin{array}{l} \textit{zu jedem } c > 0 \textit{ ex. ein } n_0 > 0 \textit{ mit} \\ c \cdot |g(n)| \leq |f(n)| \textit{ für alle } n > n_0 \end{array} \right\}$$

der Funktionen, die gegenüber f verschwinden.

(v) Die Menge

$$\omega(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \; : \; \begin{array}{l} \textit{zu jedem } c > 0 \textit{ ex. ein } n_0 > 0 \textit{ mit} \\ |g(n)| \geq c \cdot |f(n)| \textit{ für alle } n > n_0 \end{array} \right\}$$

der Funktionen, denen gegenüber f verschwindet.

Das erste Beispiel zeigt, dass mit den eingeführten Notationen nicht nur Konstanten ignoriert, sondern oft auch komplizierte Laufzeitfunktionen vereinfacht werden können.

1.7 Beispiel

Ein (reelles) Polynom vom Grad $d \in \mathbb{N}_0$ *besteht aus* $d+1$ *Koeffizienten* $a_d, \ldots, a_0 \in \mathbb{R}$, *wobei* $a_d \neq 0$ *verlangt wird. Reelle Polynome* p *beschreiben Funktionen* $p : \mathbb{R} \to \mathbb{R}$ *vermöge* $p(x) = \sum_{i=0}^{d} a_i \cdot x^i$ *für alle* $x \in \mathbb{R}$. *Polynome mit anderen Zahlenbereiche für die Koeffizienten, Definitions- und Wertebereiche sind analog definiert.*

Ist $p : \mathbb{N}_0 \to \mathbb{R}$ *ein Polynom vom Grad* d, *dann gilt für alle* $n > 0$

$$p(n) = \sum_{i=0}^{d} a_i \cdot n^i = \left(a_d + a_{d-1} \cdot \frac{1}{n^1} + \ldots + a_0 \cdot \frac{1}{n^d} \right) \cdot n^d$$

und damit

$$|p(n)| \leq \left(\sum_{i=0}^{d} |a_i| \right) \cdot n^d \quad \textit{für alle } n > 0 \; ,$$

also $p(n) \in \mathcal{O}(n^d)$. *Das Polynom läßt sich weiter umschreiben zu*

$$\begin{aligned} p(n) &= a_d \cdot \left(1 + \frac{a_{d-1}}{a_d} \cdot \frac{1}{n^1} + \ldots + \frac{a_0}{a_d} \cdot \frac{1}{n^d} \right) \cdot n^d \\ &= a_d \cdot \left[1 + \frac{1}{n} \cdot \left(\frac{a_{d-1}}{a_d} + \frac{a_{d-2}}{a_d} \cdot \frac{1}{n} + \ldots + \frac{a_0}{a_d} \cdot \frac{1}{n^{d-1}} \right) \right] \cdot n^d \end{aligned}$$

so daß

$$|p(n)| \geq |a_d| \cdot \frac{1}{2} \cdot n^d \quad \textit{für alle } n > 2 \cdot \sum_{i=0}^{d-1} \left| \frac{a_i}{a_d} \right|$$

also $p(n) \in \Omega(n^d)$ *und insgesamt*

$$p(n) \in \Theta(n^d) \ .$$

Das Wachstum einer durch ein Polynom beschriebenen Zahlenfolge hängt also nur vom Grad des Polynoms ab.

Das Wachstum einiger wichtiger Folgen im Vergleich:

n	1	10	100	1 000	10 000
$\log_{10} n$	0	1	2	3	4
$\log_2 n$	0	≈ 3	≈ 7	≈ 10	≈ 13
$\sqrt{n}$	1	≈ 3	10	≈ 32	100
n^2	1	100	10 000	1 000 000	100 000 000
n^3	1	1 000	1 000 000	1 000 000 000	1 Billionen
2^n	2	1 024	$\approx 10^{30}$	$\approx 10^{301}$	$> 10^{3\,000}$
$1,1^n$	≈ 1	≈ 3	$\approx 13\,781$	$\approx 2 \cdot 10^{41}$	$> 10^{414}$
$n!$	1	3 628 800	$\approx 9 \cdot 10^{157}$	...	...
n^n	1	10 000 000 000	10^{200}	$10^{3\,000}$	$10^{40\,000}$

Merksatz: „Ein Programm mit 10^{50} Operationen wird auf keinem noch so schnellen Rechner jemals fertig werden.“

Motiviert durch das relative Wachstum der obigen Folgen halten wir einige für die Komplexitätsbeurteilung nützliche Merkregeln fest.

1.8 Satz

(i) $g \in \mathcal{O}(f)$ *genau dann, wenn* $f \in \Omega(g)$.
$g \in \Theta(f)$ *genau dann, wenn* $f \in \Theta(g)$.

(ii) $\log_b n \in \Theta(\log_2 n)$ *für alle* $b > 1$.
„Die Basis eines Logarithmus' spielt für das Wachstum keine Rolle“

$b^x = a \iff \log_b a = x$

(iii) $(\log_2 n)^d \in o(n^\varepsilon)$ *für alle* $d \in \mathbb{N}_0$ *jedes* $\varepsilon > 0$.
„Logarithmen wachsen langsamer als alle Polynomialfunktionen“

(iv) $n^d \in o((1+\varepsilon)^n)$ *für alle* $d \in \mathbb{N}_0$ *und jedes* $\varepsilon > 0$.
„Exponentielles Wachstum ist immer schneller als polynomiales“

(v) $b^n \in o((b+\varepsilon)^n)$ *für alle* $b \geq 1$ *und jedes* $\varepsilon > 0$.
„Jede Verringerung der Basis verlangsamt exponentielles Wachstum"

Beweis. (skizzenhaft)

(i) Folgt unmittelbar aus den Definitionen.

(ii) Wegen $b^n = (2^{\log_2 b})^n = 2^{n \cdot \log_2 b}$ gilt $log_b n = (\log_2 b) \cdot \log_2 n$.

$\log b^x = x \log b$
$b^x = 2^{x \log_2 b}$

(iv) $\lim\limits_{n\to\infty} \frac{n^d}{(1+\varepsilon)^n} = 0$; Plausibilitätsargument: $\frac{(n+1)^d}{n^d} = \frac{n^d + \mathcal{O}(n^{d-1})}{n^d} \underset{n\to\infty}{\longrightarrow} 1$, das prozentuale Wachstum von n^d wird also immer kleiner, wohingegen das von $(1+\varepsilon)^n$ konstant $\varepsilon > 0$ beträgt.

(iii) $\lim\limits_{n\to\infty} \frac{(\log_2 n)^d}{n^\varepsilon} = \lim\limits_{n\to\infty} \frac{(og_2 n)^d}{(2^\varepsilon)^{\log_2 n}}$ und dann wie in (iv) mit $\log_2 n$ statt n.

(iii) $\lim\limits_{n\to\infty} \frac{b^n}{(b+\varepsilon)^n} = \lim\limits_{n\to\infty} \left(\frac{b}{b+\varepsilon}\right)^n = 0$, da $\frac{b}{b+\varepsilon} < 1$.

□

Die nächste Aussage ist vor allem für Algorithmen interessant, in denen Teilmengen fester Größe betrachtet werden.

1.9 Satz
Für festes $k \in \mathbb{N}_0$ *gilt*

$$\binom{n}{k} \in \Theta(n^k) \ .$$

Beweis. Für alle $n > k =: n_0$ gilt

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!} = \frac{n}{k} \cdot \frac{n-1}{k-1} \cdot \ldots \cdot \frac{n-(k-1)}{k-(k-1)} \ .$$

Wegen $\frac{n}{k} \leq \frac{n-i}{k-i} \leq n$, $i = 0, \ldots, k-1$, folgt daraus

$$\left(\frac{n}{k}\right)^k = \left(\frac{1}{k}\right)^k \cdot n^k \leq \binom{n}{k} \leq n^k$$

und damit $\binom{n}{k} \in \Omega(n^k) \cap \mathcal{O}(n^k) = \Theta(n^k)$. □

In den folgenden oft verwendeten Näherungsformeln wird statt der Funktion selbst der Fehler der Abschätzung asymptotisch angegeben, und zwar einmal additiv und einmal multiplikativ. Die Schreibweise bedeutet, dass es in der jeweiligen Wachstumsklasse eine Folge gibt, für die Gleichheit herrscht.

1.10 Satz
Für alle $n \in \mathbb{N}_0$ gilt

(i) $H_n := \sum_{k=1}^{n} \frac{1}{k} = \ln n + \mathcal{O}(1)$

(ii) $n! = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right)$ *(Stirlingformel)*