

Komplexitätsbetrachtungen

Frage: Wie berechnet man $n!$ am effizientesten?

1. Ansatz: $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$ $O(n)$
2. Ansatz: $n! = n \cdot (n-1)!$ auch $O(n)$
3. Ansatz: Idee: $n!$ sowieso nur darstellbar für kleine n (z.B. $n < 200$)
also
 $n! = \text{Tabelle}[n]$ $O(1)$

Ist nicht jeder Computerspeicher und damit alle darstellbaren Probleme endlich? Kann mit dem 3. Ansatz jedes Problem in $O(1)$ gelöst werden?

Komplexitätsbetrachtungen

In der Tat kann jedes Problem in $O(1)$ gelöst werden, wenn man sich auf eine maximale Größe der Eingabe beschränkt.

Meist übliche Sichtweise: jede Operation (Addition, Vergleich, Multiplikation,...) gleich aufwendig.

Dabei wird angenommen, daß die Eingabe eine maximale Größe hat (z.B. maximal n 32-bit Integers).

Theoretisch jede Aufgabe in $O(1)$ lösbar (durch Erstellen einer Tabelle)

Dies ist oft sogar sinnvoll (z.B. $n!$ sowieso nur für kleines n interessant), aber meist reicht Speicher nicht aus.

Komplexitätsbetrachtungen

Aufwand für eine Multiplikation: bisher $O(1)$

Was, wenn beliebige Eingaben (nicht nur 32-bit Integers) erlaubt?

Rechenbeispiel: $a \cdot b$, mit $a=123$, $b=34$,
Länge von a ist $|a|=n=3$, Länge von b ist $|b|=m=2$

$123 \cdot 34$
+ 492

4182

n Einzelziffer-Multiplikationen und -Additionen (je $O(1)$)
 m mal
 $m \cdot n$ Einzelziffer-Additionen

Aufwand dieser Art zu multiplizieren ist also eigentlich $O(|a| \cdot |b|)$

Bemerkung:
Es geht auch schneller (FFT).

Komplexitätsbetrachtungen

Je nach Aufgabenstellung und möglichen Eingaben, muß bei der Aufwandsabschätzung eines Algorithmus auch die Größe der eingegebenen Zahlen berücksichtigt werden.

Typische Zahlendarstellung im Rechner:

zur Basis 2: $1 = 1$ $2 = 10$
 $17 = 10001$ $2.5 = 10.1$
 $36 = 100100$ $0.25 = 0.01$

(in der Praxis natürlich weitere Besonderheiten, insbesondere für negative Zahlen)

So wird üblicherweise als Länge einer Zahl $|a| = \log_2 a$ benutzt.

Bemerkung: $O(\log_b n) = O(\log_c n) = O(\log n)$

Komplexitätsbetrachtungen Exponentiation

Aufgabe: gegeben n und k , gesucht n^k

1. Ansatz: $P=1$
für $i=1$ bis k
 $P^*=n$

Aufwandsabschätzung: $O(k)$ bei beschränkter Länge von n

Abschätzung in Abhängigkeit der Länge von n und $k \rightarrow$ Übung

Frage: geht es auch schneller?

Komplexitätsbetrachtungen Exponentiation

Überlegung: $n^{2k} = (n^k)^2 = n^k \cdot n^k$
 $O(2k)$ $O(k)+1$

allgemein ist: $n^{2^i} = (\dots(n^2)^2 \dots)^2$
 i Exponenten

2^i Multiplikationen
eine Multiplikation für n^2
 $i-1$ weitere Multiplikationen

Komplexitätsbetrachtungen

Exponentiation

Man kann also n^k für $k = 2^i$ in $O(\log k)$ berechnen.

Was aber, wenn es kein i gibt mit $k = 2^i$?

Für gerade k gilt $n^k = (n^{k/2})^2$
 und für ungerade k gilt $n^k = n \cdot (n^{(k-1)/2})^2$

Aufwand ist also für gerades k : eine Multiplikation
 plus Aufwand für $k/2$
 für ungerades k : zwei Multiplikationen
 plus Aufwand für $(k-1)/2$

gesamt: best-case $\log_2 k$ Multiplikationen bei $k = 2^i$
 worst-case $2 \log_2 k$ Multiplikationen bei $k = 2^i - 1$

Der Euklidische Algorithmus

Aufgabe: gegeben m und n , gesucht größter gemeinsamer Teiler k ,
 d.h. $m \pmod k = 0, n \pmod k = 0$, und
 wenn $m \pmod l = 0, n \pmod l = 0$, dann $k \geq l$

Erster Ansatz:

$$\begin{aligned} & i=1 \\ & \text{solange } i \leq m \text{ und } i \leq n \\ & \quad \text{falls } m \pmod i = 0 \text{ und } n \pmod i = 0 \\ & \quad \text{dann } k=i \\ & \quad i++ \end{aligned}$$

Aufwand: $O(n+m)$

Frage: geht es noch besser?

Der Euklidische Algorithmus

Überlegung: (sei oBdA $m > n$)

k ist Teiler von $m \Rightarrow m = k \cdot u$

k ist Teiler von $n \Rightarrow n = k \cdot v$

$\Rightarrow (m-n) = k(u-v)$
 ganzzahlig, da u, v ganzzahlig

also ist k auch Teiler von $m-n$

per Induktion zeigt man: $\text{ggT}(m, n) = \text{ggT}(m-n, n)$

Der Euklidische Algorithmus

Neuer Ansatz:

$$\begin{aligned} & \text{solange } m \pmod n > 0 \\ & \quad m = m - n \\ & \quad \text{falls } m < n \text{ dann vertausche } m \text{ und } n \\ & \quad \text{gib } n \text{ aus} \end{aligned}$$

Beispiel für $\text{ggT}(210, 66)$:

$m=210-66=144$
 $m=144-66=78$
 $m=78-66=12, m=66, n=12$
 $m=66-12=54$
 $m=54-12=42$
 $m=42-12=30$
 $m=30-12=18$
 $m=18-12=6, m=12, n=6$
 und jetzt ist $m \pmod n = 0 \Rightarrow \text{ggT}=6$

Und was wäre der Aufwand für $\text{ggT}(1000, 3)$?

Der Euklidische Algorithmus

Noch ein paar Überlegungen:

wir wissen schon:
 wenn k Teiler von m und n ist, dann auch von $m-n$,
 dann aber auch von $m-n-n$, und von $m-n-\dots-n$
 $\underbrace{\hspace{10em}}_{m/n \text{ Stück}}$

Oder formal:

wenn k Teiler von m und n ist, dann auch von

$$m - \left\lfloor \frac{m}{n} \right\rfloor \cdot n = m \pmod n$$

Der Euklidische Algorithmus

Verbessern wir den Algorithmus also:

$$\begin{aligned} & \text{solange } m \pmod n > 0 \\ & \quad m = m - n \\ & \quad \text{falls } m < n \text{ vertausche } m \text{ und } n \\ & \quad \text{gib } n \text{ aus} \end{aligned}$$

\Rightarrow

$$\begin{aligned} & \text{solange } m \pmod n > 0 \\ & \quad m = m \pmod n \\ & \quad \text{vertausche } m \text{ und } n \\ & \quad \text{gib } n \text{ aus} \end{aligned}$$

Beispiel $\text{ggT}(210, 66)$ jetzt: $m=210 \pmod{66}=12, m=66, n=12$
 $m=66 \pmod{12}=6, m=12, n=6$
 $m=12 \pmod{6}=0$
 $\Rightarrow \text{ggT}=6$

und $\text{ggT}(1000, 3)$ geht jetzt sehr schnell.

Der Euklidische Algorithmus

Aufwandsabschätzung:

(jede Rechenoperation benötigt konstante Zeit)

sei m_i der Wert von m in der i -ten Iteration
dann ist $m_i = n_{i-1} \pmod{m_{i-1}} = (m_{i-2} \pmod{n_{i-2}}) \pmod{m_{i-1}}$

allgemein ist $a \pmod{b} < a/2$ wenn $b < a$ (trivial)

also ist auch $m_i < m_{i-2}/2$

d.h. m_i nimmt exponentiell ab, der Aufwand ist logarithmisch: $O(\log(n+m))$

Polynommultiplikation

z.B. $(x^2 + 2x + 1) \cdot (x^2 - x + 2) = (x^4 + x^3 + x^2 + 3x + 2)$

allgemein: gegeben: $P = \sum_{i=0}^n p_i x^i, \quad Q = \sum_{i=0}^n q_i x^i$

gesucht: $R = \sum_{i=0}^{2n} r_i x^i, \quad \text{mit} \quad r_j = \sum p_k q_{j-k}$

Aufwandsabschätzung: $O(n^2)$

Bemerkung: falls P und Q verschiedene Grade haben, fülle fehlende Koeffizienten mit 0 auf bis beide Grade gleich sind.

Polynommultiplikation

Geht es schneller als in $O(n^2)$?

Wir erinnern uns an Exponentiation und versuchen Ähnliches:

$$= \begin{pmatrix} p_4 x^4 & + p_3 x^3 & & + p_2 x^2 & + p_1 x & + p_0 \\ p_4 x^2 & + p_3 x & & & & \end{pmatrix} \cdot x^2 + p_2 x^2 + p_1 x + p_0$$

Allgemein: $P = \sum_{i=0}^n p_i x^i = P_2 x^{n/2} + P_1$

mit $P_2 = \sum_{i=1}^{n/2} p_{i+n/2} x^i$ und $P_1 = \sum_{i=0}^{n/2} p_i x^i$

Polynommultiplikation

Multiplikation mit „zweigeteilten“ Polynomen

$$P = P_2 x^{n/2} + P_1 \quad Q = Q_2 x^{n/2} + Q_1$$

dann ist $PQ = P_1 Q_1 + P_1 Q_2 x^{n/2} + P_2 Q_1 x^{n/2} + P_2 Q_2 x^n$

Aufwandsabschätzung: $T(n) = 4 \cdot T(n/2) + O(n)$ $T(1) = 1$
Additionen

leider ist $T(n) = O(n^2)$
also bisher keine Verbesserung erreicht

Polynommultiplikation

Die Rekurrenzrelation $T(n) = 4 \cdot T(n/2) + O(n)$ $T(1) = 1$
wäre „freundlicher“, wenn wir statt 4 nur 3 Multiplikation von Teilpolynomen machen müßten, dann wäre $T(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$

mit 4 Multiplikationen

$$\begin{matrix} T(1)=1 \\ T(2)=4T(1)+\dots = 4+\dots \\ T(4)=4T(2)+\dots = 16+\dots \\ T(8)=4T(4)+\dots = 64+\dots \\ \dots \end{matrix}$$

n^2

mit 3 Multiplikationen

$$\begin{matrix} T(1)=1 \\ T(2)=3T(1)+\dots = 3+\dots \\ T(4)=3T(2)+\dots = 9+\dots \\ T(8)=3T(4)+\dots = 27+\dots \\ \dots \end{matrix}$$

$n^{\log_2 3}$

Polynommultiplikation

Es geht tatsächlich mit 3 Multiplikationen.

Wir schreiben kurz $A = P_1 Q_1 \quad B = P_2 Q_1 \quad C = P_1 Q_2 \quad D = P_2 Q_2$

Dann ist $PQ = A + Bx^{n/2} + Cx^{n/2} + Dx^n$

Wir benötigen aber nicht sowohl B als auch C ,
es würde genügen, wenn wir deren Summe $(B+C)$ hätten.

	P_1	P_2
Q_1	A	B
Q_2	C	D

Betrachten wir $E = A + B + C + D = (P_1 + P_2) \cdot (Q_1 + Q_2)$
 E ist Produkt zweier „kleiner“ Polynome, und
 $B+C = E - A - D$

Wir brauchen also nur A, D und E , d.h. 3 Produkte.

Polynommultiplikation

Ein Beispiel:

$$P = 1 - x + 2x^2 - x^3 \quad Q = 2 + x - x^2 + 2x^3$$

$$P_1 = 1 - x \quad Q_1 = 2 + x$$

$$P_2 = 2 - x \quad Q_2 = -1 + 2x$$

	P_1	P_2
Q_1	A	B
Q_2	C	D

$$A = P_1 Q_1 = 2 - x - x^2 \leftarrow \text{4 Multiplikationen}$$

$$D = P_2 Q_2 = -2 + 5x - 2x^2 \leftarrow \text{4 Multiplikationen}$$

$$E = (P_1 + P_2)(Q_1 + Q_2) = (3 - 2x)(1 + 3x) = 3 + 7x - 6x^2$$

$$B + C = E - A - D = 3 + 3x - 3x^2 \leftarrow \text{4 Multiplikationen}$$

$$PQ = A + (B + C)x^{n/2} + Dx^n = \dots (\text{siehe Manber})$$

Insgesamt nur 12 Multiplikationen, statt $n^2 = 16$.

Exponentiation

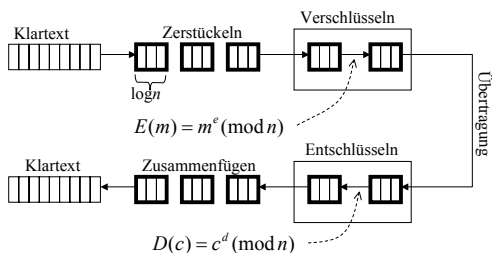
Anwendung: Kryptographie

Man hätte gerne:

- verschlüsselte Nachricht nicht länger als Original (kürzer geht nicht)
- keine Notwendigkeit für abhörsicheren Kanal (auch nicht für Code)
- sehr schwer knackbare Codierung
- Authentisierung: Eine verschlüsselte Nachricht, die nur von einer Person verschlüsselt worden sein kann.
- Abhörsicherung: Eine verschlüsselte Nachricht, die nur von einer Person entschlüsselt werden kann.
- die Ver- und Entschlüsselung sollte möglichst effizient machbar sein.
- Typische Lösung: Public Key Verfahren (z.B. RSA)

Exponentiation

Anwendung: Kryptographie



Durch modulo-Operator bleibt Länge konstant $\log n$.

Exponentiation

Anwendung: Kryptographie

Wann funktioniert das Schema „Potenzieren/Modulo“?

Es funktioniert, wenn

$$D(c) = c^d \pmod n = (m^e \pmod n)^d \pmod n = m$$

$$\Rightarrow (m^e)^d \pmod n = m$$

$$\Rightarrow m^{ed} \pmod n = m$$

d.h. wenn e und d so gewählt werden, daß sie zueinander reziprok sind bzgl. Potenzieren modulo n

Exponentiation

Anwendung: Kryptographie

Wann gibt es solche e und d ?

Nach einem Satz von Euler gilt $m^{S(n)} \pmod n = 1$

wenn m und n keine gemeinsamen Teiler haben.

also

$$m^{S(n)} \pmod n = 1$$

$$\Rightarrow m^{(k \cdot S(n) + 1)} \pmod n = m$$

$$\Rightarrow \text{wähle } e \cdot d = k \cdot S(n) + 1$$

$$\Rightarrow e \cdot d \pmod{S(n)} = 1$$

$S(n)$ = Anzahl zu n teilerfremden Zahlen $< n$
 $S(10) = \{1, 3, 7, 9\}$

z.B. 3, 10 teilerfremd
 $3^4 \pmod{10} = 81 \pmod{10} = 1$



Exponentiation

Anwendung: Kryptographie

Wir suchen also ein n, e, d , wobei

- $ed = kS(n) + 1$ (e, d teilerfremd zu $S(n)$)
- wenn n und e veröffentlicht werden, kann man (Hacker) d daraus nicht (einfach) berechnen

Diese Art Verschlüsselung heißt **RSA** (Rivest, Shamir, Adleman)

Berechnung von d aus n und e wäre nicht schwierig, wenn $S(n)$ leicht zu berechnen wäre.

Wie lange braucht man wohl um $S(100\text{-stellige Zahl})$ zu berechnen?

1977 geschätzt: 40 Billionen Jahre (Wettbewerb von R., S. und A.)
 1994 geknackt in 8 Monaten auf 1600 Internet-Rechnern

Exponentiation

Anwendung: Kryptographie

Hacker soll $S(n)$ nicht leicht berechnen können,
aber der rechtmäßiger Benutzer (er muß ja e und d definieren).

Idee: Es ist kein Verfahren bekannt, das sehr große Zahlen
in akzeptabler Zeit (Existenz des Universums) faktorisiert.

Also: Wähle $n = p \cdot q$, wobei p und q sehr große Primzahlen sind.
(große Primzahlen finden geht schon einigermaßen gut,
zumindest mit ausreichend hoher Wahrscheinlichkeit)

Dann ist $S(n) = p \cdot q - (p-1 + q-1 + 1) = (p-1)(q-1)$

Vielfache von q ---
Vielfache von p ---
 $p \cdot q$ ---

Exponentiation

Anwendung: Kryptographie

Beispiel für RSA Verschlüsselung:

$$E(m) = m^e \pmod{n}$$

$$D(c) = c^d \pmod{n}$$

$$p=3, q=5 \Rightarrow n=15, \quad d=3, e=11$$

es ist $d \pmod{p-1} > 0$ und $d \pmod{q-1} > 0$

außerdem ist $d \cdot e \pmod{(p-1)(q-1)} = 33 \pmod{8} = 1$

$$E(2) = 2^{11} \pmod{15} = 2048 \pmod{15} = 8$$

$$D(8) = 8^3 \pmod{15} = 512 \pmod{15} = 2$$

Exponentiation

Anwendung: Kryptographie

Aufwandsabschätzung für RSA Algorithmus: $E(m) = m^e \pmod{n}$

Mit Hilfe der Exponentiation mittels iterativen Quadrierens und
Halbierens des Exponenten kann m^e mit $O(\log e)$ Multiplikationen
berechnet werden.

Aber wird m^2, m^3, m^4, \dots nicht immer größer?

Nicht nötig, denn es gilt $ab \pmod{n} = (a \pmod{n}) b \pmod{n} \pmod{n}$,
d.h. $m^i \pmod{n} = (m^{i-1} \pmod{n}) \cdot m \pmod{n}$

Also $\log e$ Multiplikationen und Divisionen,
bzw. modulo-Multiplikationen