ETH Zürich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger, Dr. Johannes Lengler Len
Florian Meier

HS 2018

Algorithmen und Komplexität
Lösungsvorschlag zu Übungsblatt 1

Lösungsvorschlag zu Aufgabe 1

(a) Ja, da $\lim_{n\to\infty} \frac{\ln n}{\log_2 n} = \ln 2$.

(b) Nein, da $\limsup_{n\to\infty} \frac{n}{\log_2 n} = \infty$.

(c) Ja, da $\limsup_{n\to\infty} \frac{n!}{n^n} = \lim_{n\to\infty} \frac{n\cdot(n-1)\cdots 1}{n\cdot n\cdots n} \leq \lim_{n\to\infty} \frac{1}{n} = 0$. Via

$$\lim_{n\to\infty} \frac{n!}{n^n} \leq \lim_{n\to\infty} \left(\frac{\lceil n/2 \rceil}{n}\right)^{\lceil n/2 \rceil} \left(\frac{n}{n}\right)^{\lfloor n/2 \rfloor} \leq \lim_{n\to\infty} \left(\frac{1}{2} + \frac{1}{n}\right)^{\lceil n/2 \rceil}$$

(oder mit der Stirling-Formel) sieht man sogar dass der Limes exponentiell gegen 0 geht.

(d) Nein, siehe (c).

(e) Nein, da $\limsup_{n\to\infty} \frac{2^{2n}}{2^n} = \limsup_{n\to\infty} 2^n = \infty$.

(f) Ja, da $\lim_{n\to\infty} \frac{1/n}{1} = 0$.

Lösungsvorschlag zu Aufgabe 2

Das folgende Programm verdoppelt die Zahlen in den Speicherzellen $M_1, \ldots, M_n$:

Eingabe: n natürliche Zahlen in $M_1, \ldots, M_n$ sowie $M_0 = n$.
Ausgabe: Die Zahlen in $M_1, \ldots, M_n$ verdoppelt.

1: $M_{-2} \leftarrow 1$
2: $M_{-1} \leftarrow M_{M_0}$
3: $M_{-1} \leftarrow M_{-1} + M_{-1}$
4: $M_{M_0} \leftarrow M_{-1}$
5: $M_0 \leftarrow M_0 - M_{-2}$
6: GOTO 2 IF $M_0 > 0$.

Die Zahl in M_0 sagt uns, wieviele Zahl wir verdoppeln müssen. Wir starten bei M_n und kopieren in Befehlszeile 2 die Zahl von M_n nach M_{-1}. Dort wird sie verdoppelt, anschliessend überschreiben wir M_n mit dem neuen Wert. Wir haben M_n somit verdoppelt, machen unseren Zähler in M_0 um eins kleiner und fahren danach mit M_{n-1} fort. Dieses Prozedere wiederholen wir solange bis $M_0 = 0$. Dann sind alle Zahlen verdoppelt und das Programm ist fertig.

Lösungsvorschlag zu Aufgabe 3

Seien $L_A(n)$, $L_B(n)$ bzw. $L_C(n)$ die Laufzeiten der Algorithmen FIBA, FIBB bzw. FIBC für die Eingabe n.

Wir bemerken zuerst, dass die Fibonacci-Zahlen exponentiell wachsen, indem wir folgende Schranken mittels vollständiger Induktion zeigen

$$2^n \geq F_n \geq \left(\frac{3}{2}\right)^{n-2}$$

Als Induktionsschritt haben wir

$$2^{n+1} \geq 2^n + 2^{n-1} \geq F_n + F_{n-1} = F_n$$

und

$$F_{n+1} = F_n + F_{n-1} \geq \left(\frac{3}{2}\right)^{n-2} + \left(\frac{3}{2}\right)^{n-3} = \left(\frac{3}{2}\right)^{n-3}\left(\frac{3}{2}+1\right) \geq \left(\frac{3}{2}\right)^{n-1};$$

als Induktionsanfang $2^1 \geq F_1 = 1 \geq \left(\frac{3}{2}\right)^{-1} = \frac{2}{3}$ und $2^2 \geq F_2 = 1 \geq \left(\frac{3}{2}\right)^0 = 1$. Aus obiger Schranke folgt, dass

$$n \log_2 2 = \log_2 2^n \geq \log_2 F_n \geq \log_2 \left(\frac{3}{2}\right)^{n-2} = (n-2)\log_2 \frac{3}{2},$$

insbesondere werden $\Theta(n)$ Bits benötigt um F_n abzuspeichern.

a) Wir zeigen dass unter Annahme des Einheitskostenmodell $L_A(n) = \Omega((\frac{3}{2})^n)$, $L_B(n) = \Theta(n)$ und $L_C(n) = \Theta(\log n)$ gilt. Es folgt, dass Algorithmus FIBC schneller als Algorithmus FIBB und dieser schneller als FIBA ist.

   FIBA berechnet F_n rekursiv, indem er FIBA$(n-1)$ und FIBA$(n-2)$ aufruft. gilt aufgrund der rekursiven Definition

$$L(n) = L(n-1) + L(n-2) + \mathcal{O}(1) \tag{1}$$

   (die Addition der beiden vorhergehenden Fibonacci-Zahlen und die vorhergehenden Befehle benötigen $\mathcal{O}(1)$ Zeit), wobei

$$L(0) = L(1) \geq 1\,.$$

   Man sieht, dass $L(n)$ selbst einer Fibonacci-ähnlichen Rekursionsgleichung folgt und $L(n) \geq F_n$ gilt.

   Es folgt, dass $L_A(n) = \Omega(\left(\frac{3}{2}\right)^n)$ gilt, d.h. die Lauzeit von FIBA ist exponentiell. Ganz genau könnte man durch Lösen der inhomogenen Differenzengleichung (1) zeigen, dass die Laufzeit $\Theta(\Phi^n)$ ist, wobei $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.61$ („Goldener Schnitt"). Dies hängt natürlich zusammen mit der expliziten Darstellung der Fibonacci-Folge $F_n = \frac{1}{\sqrt{5}}(\Phi^n - (1-\Phi)^n)$, welche man als Lösung der zu (1) zugehörigen homogenen Differenzengleichung erhält. Die Laufzeit von FIBA wächst also gleich schnell wie die Fibonacci-Zahlen selbst.

   Nun zu FIBB. Hier wird die FOR-Schleife $(n-1)$-mal durchlaufen, wobei in jedem Durchlauf konstant viele Operationen ausgeführt werden. Da wir das Einheitskostenmass benutzen benötigt jede dieser Operatioinen eine Zeiteinheit. Es folgt $L_B(n) = \Theta(n)$.

   Zuletzt betrachten wir FIBC. Beachte, dass die Multiplikation von zwei 2×2 Matrizen nur konstant viele Multiplikationen und Additionen benötigt. Das bedeutet, dass jede Iteration der For-Schleife konstant viel Zeit benötigt. Da die For-Schleife $\log_2(n)$ mal durchlaufen wird, gilt $L_C(n) = \Theta(\log n)$.

b) Untersuchen wir also noch die Laufzeiten mit dem logarithmischen Kostenmass.

   Die Laufzeit von FIBA folgt nun der Rekursion

$$L(n) = L(n-1) + L(n-2) + \log F_{n-1} + \log F_{n-2} + \mathcal{O}(1),$$

   da der Zeitbedarf für die Addition der zwei vorhergehenden Fibonaccizahlen nun gleich der Summe der Anzahl involvierter Bits ist. Wie oben wächst $L_A(n)$ mindestens so schnell wie F_n und gleichermassen folgt dass $L_A(n) = \Omega((\frac{3}{2})^n)$ gilt.

In FIBB benötigt die i-te Iteration der For-Schleife $\Theta(\log F_i) = \Theta(i)$ Zeit, da die Fibonacci-Zahlen exponentiell wachsen. Als Gesamtlaufzeit ergibt sich

$$\sum_{i=1}^{n} \Theta(i) = \Theta(n^2) .$$

Für FIBC bemerken wir zuerst, dass

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

gilt. Die i-te Iteration der For-Schleife benötigt $\Theta(\log F_{2^i}) = \Theta(2^i)$ Zeit, da Zahlen der Grösse F_{2^i} involviert sind und konstant viele Operationen ausgeführt werden. Als Gesamtlaufzeit ergibt sich

$$\sum_{i=1}^{k} \Theta(2^i) = \Theta(n) .$$

Demnach ist auch unter der Annahme des logarithmischen Kostenmasses Algorithmus *FibC* schneller als *FibB*, welcher schneller als *FibA* ist.

Wir bemerken, dass auch das logarithmische Kostenmass nicht ganz der Wahrheit entspricht. Zwar benötigt eine Addition zweier Zahlen m und n tatsächlich annähernd $\log_2 m + \log_2 n$ Zeit. Die Mutliplikation von m und n würde allerdings unter der Verwendung der schriftliche Multiplikation $\log_2 m \log_2 n$ Zeit benötigen, da man $\log_2 m$ Zahlen der Länge $\log_2 n$ addieren muss. Es gibt effizientere Verfahren zur Multiplikation zweier Zahlen, zum Beispiel der Schönhage-Strassen-Algorithmus (1971) kann zwei Zahlen der Länge n in $O(n \log n \log\log n)$ Bitoperationen multiplizieren. Allerdings ist dieser Algorithmus für praktische Zwecken meist nicht geeignet, da die Konstanten die sich hinter der O-Notation verstecken riesig sind.

c) Leider haben wir bei der Analyse in a) gemogelt: die Fibonaccizahl F_n ist exponentiell viel grösser als die Eingabe n, d.h., wir haben die "ungeschriebenen Regeln" verletzt. Diese besagen, dass unter Verwendung des Einheitskostenmasses nur *polynomial beschränkte* Zahlen in Speicherzellen abgespeichert werden dürfen, d.h. Zahlen, die kleiner sind als n^k für ein beliebiges aber festes k. Beachte, wenn k fest ist, werden $\Theta(\log n)$ Bitoperationen für die Addition zweier Zahlen der Grösse n^k benötigt. Für die Addition zweier Zahlen der Grösse 2^n benötigen wir allerdings $\Theta(n)$ Bitoperationen, was deutlich grösser als $\Theta(\log n)$ ist und nicht mit guten Gewissen vernachlässigt werden kann. Deshalb sollte für Algorithmen, bei denen Zahlen exponentiell viel Grösser als die Eingabe n werden, das logarithmische Kostenmass verwendet werden. Für diese Aufgabe ist deshalb die Analyse in b) besser geeignet. Für den Rest der Vorlesung (mit Ausnahme des Kapitels Komplexitätstheorie) werden wir zur Laufzeitanalyse das Einheitskostenmass verwenden.

d) Ähnlich zur Binärdarstellung von n stellen wir A^n als Produkt der von FIBC berechneten Matrizen dar.

Sei $k = \lceil \log_2 n \rceil$ die Anzahl Stellen in der Binärdarstellung $n_1 n_2 \dots n_k$ von n. Der Algorithmus FIBC berechnet die Matrizen $A^{2^1}, A^{2^2}, A^{2^3}, \dots$. Wir speichern alle A^{2^i} für $i = 0, \dots k$ ab und benutzen diese Matrizen um A^n zu berechnen:

$$A^n = A^{\sum_{i=1}^{k} 2^{k-i} n_i} = \prod_{i=1}^{k} A^{2^{k-i} n_i}$$

Um dieses Produkt zu berechnen brauchen wir $O(\log n)$ zusätzliche Matrixmultiplikationen, das heisst, die asymptotische Laufzeit bleibt unverändert.