Algorithmen und Datenstrukturen

Blatt 2

Uni Hamburg, Wintersemester 2018/19

Präsentation am 13.–15. November 2019

Jede Teilaufgabe (a, b, ...) zählt als ein einzelnes Kreuzchen.

Übung 1.

Betrachten Sie folgende Funktionsdefinition zur Prüfung ob zwei gegebene Listen disjunkt sind, also keine gleichen Elemente enthalten:

```
1 def disjoint(a:list, b:list):
2     different_counter = 0
3     for x in a:
4         for y in b:
5             if x != y:
6                 different_counter += 1
7     return different_counter == len(a) * len(b)
```

(a) Erläutern Sie die Vorgehensweise des Algorithmus und beweisen Sie seine Korrektheit.

(b) Geben Sie die Laufzeitkomplexität von `disjoint` an – welches sind die entscheidenden Einflussfaktoren auf die Laufzeit?

(c) Schlagen Sie Möglichkeiten vor, den Code zu optimieren. Lohnt es sich, die Listen `a` und `b` extra für den Aufruf dieser Funktion zu sortieren?

Lösung 1.

(a) Der Algorithmus prüft für jede Elementpaarung $x \in a$ und $y \in b$, ob $x \neq y$ und erhöht dann den Zähler `different_counter`. Insgesamt gibt es $|a| \cdot |b|$ Elementpaarungen und der Algorithmus prüft, ob der Zähler diesen Wert erreicht, also für alle Paare die jeweilige Ungleichheit festgestellt hat.

Widerspruchsbeweis: Angenommen $x \in a$ und $x \in b$. Dann wird mindestens einmal $x \neq x$ geprüft und der Zähler `different_counter` nicht erhöht. Da nur insgesamt $|a| \cdot |b|$ mal der Zähler um (maximal) 1 erhöht wird, kann der Zähler diesen Wert nicht mehr erreichen, wenn einmal nicht um 1 erhöht wurde.

(b) Die Laufzeitkomplexität beträgt $O(|a| \cdot |b|)$. Entscheidend für die Laufzeit ist also die Länge der untersuchten Listen und weiterhin der Aufwand für die Vergleichsoperation, der aber nur als Konstante in die Komplexitätsbetrachtung eingeht.

(c) Kein Gewinn in Sachen Komplexität wäre es, sobald eine Ungleichheit festgestellt ist `False` zurückzugeben, auch wenn die mittlere Laufzeit sich hierdurch reduziert.

Sofern die Listen ähnlich lang sind, lohnt es sich, die Listen zu sortieren, weil dies in $O(n \log n)$ möglich ist und danach die Prüfung in $O(|a| + |b|)$ erfolgen

kann, wenn man jeweils die Listenköpfe vergleicht und bei Ungleichheit jenen Listenkopf entfernt der kleiner ist. Sind die Listen sehr unterschiedlich lang (z.B. $|a| < \log(\log(|b|)))$, so würde es jedoch länger dauern b zu sortieren als einfach mit a zu vergleichen.

Übung 2.
Wir betrachten den rekursiven Algorithmus MULTIPLY zur Multiplikation zweier Zahlen. Dabei sollen folgende Annahmen gelten:

- n und m sind natürliche Zahlen.
- LENGTH(n) berechnet die Länge (also die Anzahl an Ziffern) von n in Binärdarstellung in Laufzeit $\mathcal{O}(1)$.
- SPLIT(n, k) gibt natürliche Zahlen n_1 und n_0 zurück, sodass $n = n_1 \cdot 2^k + n_0$ sowie $n_0 < 2^k$ gilt, und hat eine Laufzeit von $\mathcal{O}(\text{LENGTH}(n))$.
- SHIFTLEFT(n, k) berechnet $n \cdot 2^k$ in einer Laufzeit von $\mathcal{O}(\text{LENGTH}(n) + k)$.
- Vergleichsoperationen mit 0 oder 1 haben eine Laufzeit von $\mathcal{O}(1)$.
- Die Laufzeit der Multiplikation und Division von k mit 2 ist $\mathcal{O}(\text{LENGTH}(k))$.
- $n + m$ bzw. $n - m$ haben eine Laufzeit von $\mathcal{O}(\max(\text{LENGTH}(n), \text{LENGTH}(m)))$.

```
MULTIPLY(n, m)
 1  if n == 0 or m == 0
 2      return 0
 3  elseif n == 1
 4      return m
 5  elseif m == 1
 6      return n
 7
 8  k = max(LENGTH(n), LENGTH(m)) / 2
 9  n_1, n_0 = SPLIT(n, k)              // n = n_1 · 2^k + n_0;  n_0 < 2^k
10  m_1, m_0 = SPLIT(m, k)              // m = m_1 · 2^k + m_0;  m_0 < 2^k
11
12  a_2 = MULTIPLY(n_1, m_1)
13  a_0 = MULTIPLY(n_0, m_0)
14  a_1 = MULTIPLY(n_1 + n_0, m_1 + m_0) − a_2 − a_0
15
16  return SHIFTLEFT(a_2, 2k) + SHIFTLEFT(a_1, k) + a_0
```

(a) Sind die Annahmen (insbesondere in Bezug auf die Laufzeit) plausibel? Erklären Sie, warum bzw. warum nicht.

(b) Gehen Sie MULTIPLY Schritt für Schritt durch und erklären Sie die Funktionsweise. (Hinweis: Versuchen Sie, sowohl das Produkt $n \cdot m$ als auch den von MULTIPLY zurückgegebenen Term durch n_1, n_0, m_1, m_0 und k auszudrücken.) *Optional:* Versuchen Sie, ihre Erklärung als (Korrektheits-)Beweis zu formulieren.

(c) Geben Sie die asymptotische Laufzeit von $\textsc{Multiply}(n, m)$ an, wobei wir Annehmen, dass $\textsc{Length}(n) = \textsc{Length}(m)$ gilt. Stellen Sie dazu eine Rekurrenzgleichung für die Laufzeit von $\textsc{Multiply}(n, m)$ auf und wenden Sie daraufhin das Master-Theorem an.

Lösung 2.

Anmerkung: Bei Multiply *handelt es sich um den Karatsuba-Algorithmus (1962 von Karatsuba und Ofman veröffentlicht). Dieser ist asymptotisch schneller als der typische Algorithmus zum Multiplizieren "per Hand", welcher eine quadratische Laufzeit hat. Der Karatsuba-Algorithmus und seine Generalisierung, der Toom-Cook-Algorithmus, wird bei Faktoren ab einer gewissen Größe in z.B. Computeralgebrasystemen eingesetzt.*

(a) Ja, die Annahmen sind plausibel.

- Length: In üblichen Bignum-Implementationen (also Implementationen von beliebig großen natürlichen Zahlen) werden Zahlen als Arrays von z.B. 64-bit-Zahlen dargestellt. Dabei wird die Länge des Arrays während anderer Operationen verwaltet (da die Information da dort auch gebraucht wird), insofern ist die Berechnung der Länge einfach nur das Laden der Länge aus dem Speicher, was unter unseren allgemeinen Annahmen in $\mathcal{O}(1)$ Laufzeit möglich ist.
- ShiftLeft: Diese Funktion kann durch eine Kombination von Shiften sowie bitweisem Und und Oder (auf den einzelnen Teilzahlen) implementiert werden, wobei nur eine konstante Anzahl an Operationen pro Teilzahl benötigt wird. Das Resultat hat eine Länge von $\textsc{Length}(n) + k$, und braucht somit mindestens diese Laufzeit um gespeichert zu werden.
- Split: n_1 kann durch Shift nach rechts um k berechnet werden, n_0 durch bitweises Und mit $2^k - 1$; beides ist in der angegebenen Laufzeit möglich.
- Vergleich mit 0 oder 1: Ist durch Vergleich der Längen und dann Vergleich der Zahl in $\mathcal{O}(1)$ umsetzbar.
- Multiplikation und Division von k mit 2: Ist durch Shift nach Links/Rechts machbar, also in Laufzeit $\mathcal{O}(\textsc{Length}(k) + 1) = \mathcal{O}(\textsc{Length}(k))$.
- Addition und Subtraktion: Bein Addieren und Subtrahieren eines einzelnen Gliedes (also der z.B. 64-Bit-Zahlen) fällt immer höchstens eine Ziffer an Übertrag an; somit ist die Operation in linearer Laufzeit umsetzbar (mit dem aus der Schule bekannten Algorithmus).

(b) Zeilen 1-6 stellen den Induktionsanfang dar (für $n, m < 1$).

Im Induktionsschritt sei nun also $n, m \geq 2$ und für alle $i < n, j < m$ sei $\textsc{Multiply}(i, j) = i \cdot j$. Insbesondere sind also alle rekursiven Aufrufe von Multiply korrekt. Drückt man nun $n \cdot m$ wie im Hinweis vorgeschlagen aus, und benutzt man $a_2 = n_1 m_1$, $a_0 = n_0 m_0$, sowie $a_1 = (n_1 + n_0)(m_1 + m_0) - a_2 - a_0$, ergibt sich

$$\begin{aligned}
n \cdot m &= (n_1 2^k + n_0)(m_1 2^k + m_0) \\
&= n_1 m_1 2^{2k} + (n_1 m_0 + n_0 m_1) 2^k + n_0 m_0 \\
&= a_2 2^{2k} + (n_1 m_1 + n_1 m_0 + n_0 m_1 + n_0 m_0 - n_1 m_1 - n_0 m_0) 2^k + a_0 \\
&= a_2 2^{2k} + ((n_1 + n_0)(m_1 + m_0) - a_2 - a_0) 2^k + a_0 \\
&= a_2 2^{2k} + a_1 2^k + a_0,
\end{aligned}$$

was die Ausdrucksweise des Algorithmus ist.

(c) Sei $l = \textsc{Length}(n) = \textsc{Length}(m)$. Alle nichtrekursiven Operationen haben eine Laufzeit von insgesamt $\mathcal{O}(l)$. Es gibt drei rekursive Aufrufe, dabei ist die Problemgröße $\lceil \frac{l}{2} \rceil$. Somit ist die Rekurrenzgleichung für die Laufzeit

$$T(l) = \begin{cases} \mathcal{O}(1) & l \leq 1 \\ 3T(\lceil \frac{l}{2} \rceil) + \mathcal{O}(l) & l > 1 \end{cases}.$$

Somit ist im Mastertheorem $a = 3$, $b = 2$ sowie $f(l) = l$, der kritische Exponent ist $\log_b(a) = \log_2(3) \approx 1.585$; wir sind also im 1. Fall (sprich: die Teilprobleme dominieren die Laufzeit). Also ist die asymptotische Laufzeit $\mathcal{O}(l^{\log_2(3)}) \approx \mathcal{O}(l^{1.585})$.

Übung 3.
zur Heap-Datenstruktur:

(a) Ausgehend von einem leeren (max-)Heap, fügen Sie die Elemente 28, 37, 55, 31, 22, 40, 7 in dieser Reihenfolge ein. Geben Sie jeweils den Inhalt des Heaps nach jeder Einfügung an.

(b) Welche Höhe h hat ein Heap mit n Elementen höchstens? Welche Höhe hat er mindestens? Beweisen Sie die Korrektheit Ihrer Antwort.

(c) Was ist die Laufzeit von Heap-Sort für n Elemente wenn diese schon auf- oder absteigend geordnet sind? Begründen Sie Ihre Antwort.

Lösung 3.
Folgende Fakten über Heaps sind nützlich:

- Heaps sind Binärbäume deren Äste sich in der Tiefe (Anzahl der Knoten von Wurzel bis Blatt) höchstens um 1 unterscheiden.
- Als Höhe eines Baums bezeichnet man seine größte Tiefe (also der längste Ast zwischen Wurzel und einem der Blätter).
- A heap can be viewed as an array where each level of the binary tree is stored contiguously, one after the other, starting with level 0.

(a) in Array-Darstellung (für Baum-Darstellung siehe zum Beispiel `https://visualgo.net/en/heap` .

28

,

37	28

,

55	28	37

,

55	31	37	28

,

55	31	37	28	22

,

55	31	40	28	22	37

,

55	31	40	28	22	37	7

.

(b) Heaps sind Binärbäume, deren Äste sich in der *Tiefe* höchstens um 1 unterscheiden. Da die *Höhe* eines Baumes jeweils als die größte Tiefe (irgend-)eines Blattes des Baums definiert ist, ist die mindeste Höhe des Baums identisch mit seiner höchsten Höhe. Im weiteren ergeben sich zwei Fälle: (a) der Baum ist vollständig balanciert (und die Tiefen der Blätter unterscheiden sich nicht), oder (b) der Baum ist bis zu einer bestimmten Ebene vollständig gefüllt aber auf der untersten Ebene nur teilweise gefüllt (und die Tiefen der Blätter unterscheiden sich um 1). Für vollständig balancierte Binärbäume gilt, dass $n = 2^h - 1$ (innere Knoten zählen mit!) und damit $h = \log_2(n+1)$. Für balancierte Bäume, bei denen die unterste Ebene nicht vollständig gefüllt ist, gilt $h = \lceil \log_2(n+1) \rceil$. Da im Fall eines vollständig balancierten Baums $\log_2(n+1)$ ganzzahlig ist (und dann $\log_2(n+1) = \lceil \log_2(n+1) \rceil$), gilt $h = \lceil \log_2(n+1) \rceil$ für alle Heaps.

(c) Die Laufzeit für Heap-Sort ist immer $\mathcal{O}(n \log n)$. Heap-Sort besteht aus zwei Schritten: 1. baue aus der Eingabe einen Heap auf, 2. extrahiere n-mal das Wurzel-Element aus dem Heap. Während der erste Schritt für eine sortierte Liste trivial ist ($\mathcal{O}(1)$), da diese bereits ein Heap ist (für max-Heaps: absteigend sortierte Liste), ist der zweite Schritt (die Heap-Eigenschaft jeweils durch Vertauschungen wieder sicherstellen) jeweils mit Aufwand verbunden (und zwar in der Höhe des Baums, also $\mathcal{O}(\log n)$). Da n-mal ein Element entnommen ($\mathcal{O}(1)$) und die Heap-Eigenschaft wieder hergestellt werden muss ($\mathcal{O}(\log n)$) ergibt sich die Komplexität $\mathcal{O}(n \log n)$.