Algorithmen und Datenstrukturen
Übungsserie 1

Markus Pawellek
144645

markuspawellek@gmail.com
Übung: Montag 10-12

Aufgabe 1

Sei $n \in \mathbb{N}$ und $N_n \coloneqq \{k \in \mathbb{N}_0 \mid k < n\}$. Dann wird die Menge B_n aller n bit-Kodierungen durch die folgende Definition beschrieben.

$$B_n \coloneqq \{(x_i)_{i \in N_n} \mid x_i \in \{0,1\} \text{ für alle } i \in N_n\}$$

Man definiert den Speicherbedarf dieser Elemente durch die Abbildung sizeof.

$$\text{sizeof}\colon B \to \mathbb{N}, \qquad \text{sizeof}((x_i)_{i \in N_n}) \coloneqq n\,\text{bit} \qquad \text{mit} \qquad B \coloneqq \bigcup_{n \in \mathbb{N}} B_n$$

Für alle weiteren Betrachtungen kodieren wir sowohl Zeiger auf Speicheradressen als auch Zahlen durch Elemente aus B_{64} ($8\,\text{byte} = 64\,\text{bit}$). Weiterhin soll eine Sequenz von kodierten Zahlen $x \coloneqq (x_i)_{i \in N_n}$ aus der Menge B_{64}, bestehend aus $n \in \mathbb{N}$ Elementen, gegeben sein. Die betrachteten Datenstrukturen zur Speicherung dieser Sequenz werden auf einer »Random Access Machine« (RAM) mit den Speicherzellen $s_i \in B_{64}$ mit $i \in N_{2^{64}}$ implementiert. Es wird davon ausgegangen, dass n klein genug ist, sodass alle Implementierungen in der RAM gespeichert werden können.

(a) Wird x in Form eines Arrays gespeichert, liegen alle Elemente von x beginnend bei einer beliebigen Speicheradresse $o \in N_{2^{64}-n}$ kontinuierlich im Speicher, sodass für alle $i \in N_n$

$$s_{o+i} = x_i$$

Sind o und n zur Zeit der Übersetzung des Quelltextes bekannt und unveränderbar, so müssen diese nicht gespeichert werden. In diesem Falle wäre die Größe des benötigten Speichers $m \in \mathbb{N}$

$$m = \sum_{i \in N_n} \text{sizeof}(s_{o+i}) = 64n\,\text{bit} = 8n\,\text{byte}$$

Je nachdem, ob nun noch o oder n gespeichert werden, ergeben sich noch zwei weitere Fälle m^* und m^{**} für den Speicheraufwand, da diese jeweils in einer weiteren Speicherzelle kodiert werden können.

$$m^* = 64(n+1)\,\text{bit} = 8(n+1)\,\text{byte}$$
$$m^{**} = 64(n+2)\,\text{bit} = 8(n+2)\,\text{byte}$$

(b) Im Falle der einfach verketteten Liste liegen die einzelnen Werte nicht kontinuierlich im Speicher. Ist x_i für $i \in N_n$ an der Adresse $a \in N_{2^{64}-1}$ gespeichert, so befindet sich in s_{a+1} die kodierte Form der Adresse des nächsten Elements x_{i+1} sofern dieses existiert. Ist dies nicht der Fall, so enthält s_{a+1} einen bestimmten (vorher festgelegten) Wert, der das Ende der Liste aufzeigt. Dadurch muss die Länge der Liste nicht extra gespeichert werden. Es ergeben sich auch hier wieder zwei Fälle, je nachdem ob die Adresse o von x_0 feststeht oder gespeichert werden muss.

$$m = 64(2n)\,\text{bit} = 16n\,\text{byte}$$
$$m^* = 64(2n+1)\,\text{bit} = 8(2n+1)\,\text{byte}$$

(c) Für die doppelt verkettete Liste gelten ähnliche Betrachtungen, wie für die einfach verkettete Liste. Nimmt man wieder die Adresse $a \in N_{2^{64}-2}$ eines Elementes $x_i, i \in N_n$, so setzt man s_{a+1} wie vorher. Hinzu kommt, dass s_{a+2} nun die kodierte Adresse des Elementes x_{i-1}, sofern dieses existiert, enthält. Ist dies nicht der Fall, enthält auch diese Speicherzelle den Wert, welcher das Ende der Liste beschreibt. Auch hier kann wieder die Adresse o von x_0 gespeichert werden. Häufig kommt es auch vor, dass sogar die Adresse e von x_{n-1} gespeichert wird.

$$\begin{aligned} m &= 64(3n)\,\mathrm{bit} = 24n\,\mathrm{byte} \\ m^* &= 64(3n+1)\,\mathrm{bit} = 8(3n+1)\,\mathrm{byte} \\ m^{**} &= 64(3n+2)\,\mathrm{bit} = 8(3n+2)\,\mathrm{byte} \end{aligned}$$

Aufgabe 2

Seien $n \in \mathbb{N}$ durch 12 teilbar und N_n wie in Aufgabe 1 definiert. Dann sind die einzigen Vergleiche, die im gegebenen Algorithmus vorkommen, in Zeile 4 zu sehen. Im Allgemeinen werden diese beiden Vergleiche unterschiedlich oft aufgerufen. Hier soll jedoch angenommen werden, dass die Anzahl der Vergleiche durch den ersten Vergleich bestimmt wird, da dieser mindestens genauso oft aufgerufen wird wie der Zweite. Aus der Vorlesung ist nun bekannt, das die Anzahl der Aufrufe $t \in \mathbb{N}$ der Zeile 4 durch die folgende Gleichung beschrieben wird, wobei $t_j \in \mathbb{N}, t_j \leq j$ für alle $j \in \mathbb{N}, 2 \leq j \leq n$.

$$t = \sum_{j=2}^{n} t_j$$

Um die weitere Notation zu vereinfachen, sei $J := \{j \in \mathbb{N} \mid 2 \leq j \leq n\}$.

(a) Nach Voraussetzung gilt für alle $i \in N_n$

$$x_i := i + 1$$

Bei der gegebenen Sequenz x handelt es sich um eine bereits geordnete Sequenz. Aus der Vorlesung ist bekannt, dass dann $t_j = 1$ für alle $j \in J$ gilt.

$$\implies \qquad t = \sum_{j \in J} 1 = n - 1$$

(b) Nach Voraussetzung gilt für alle $i \in N_n$

$$x_i := n - i$$

Bei der gegebenen Sequenz handelt es sich um eine umgekehrt geordnete Sequenz. Auch hier ist aus der Vorlesung bekannt, dass $t_j = j$ für alle $j \in J$ gilt.

$$\implies \qquad t = \sum_{j \in J} j = \frac{n(n+1)}{2} - 1$$

(c) Die gegebene Sequenz wird für alle $k \in N_{\frac{n}{3}}$ beschrieben durch

$$x_{3k} = 3k + 3, \qquad x_{3k+1} = 3k + 2, \qquad x_{3k+2} = 3k + 1$$

Die Sequenz x kann damit in bereits geordnete Blöcke der Größe 3 eingeteilt werden. Durch Anwendung der Lösungen aus (a) und (b) folgt für alle $k \in \mathbb{N}$ mit $2 \leq k \leq \frac{n}{3}$ gilt

$$t_2 = 2, \qquad t_3 = 3, \qquad t_{3k-2} = 1, \qquad t_{3k-1} = 2, \qquad t_{3k} = 3$$

$$\Longrightarrow \qquad t = t_2 + t_3 + \sum_{k=2}^{\frac{n}{3}} (t_{3k-2} + t_{3k-1} + t_{3k}) = 5 + 6\left(\frac{n}{3} - 1\right) = 2n - 1$$

(d) Die gegebene Sequenz wird für alle $k \in N_{\frac{n}{4}}$ beschrieben durch

$$x_{4k} = 2k+1, \qquad x_{4k+1} = 2k+2, \qquad x_{4k+2} = n-2k, \qquad x_{4k+3} = n-2k-1$$

Die ersten drei Elemente jedes Viererblocks sind in sich geordnet. Sie müssen also die gleiche Anzahl an Verschiebungen ausführen, wie aus (a) und (b) folgt. Das erste Element eines solchen Blockes ist auf jeden Fall kleiner als die Hälft der bereits sortierten Elemente, da für alle $k \in N_{\frac{n}{4}}$ und alle $p \in N_k$ gilt

$$2p+1 < 2p+2 < 2k+1 < n-2p-1 < n-2p$$

Das vierte Element muss nun wegen $n - 2k - 1 < n - 2k$ genau ein Element mehr verschoben werden. Es folgt dann für alle $k \in \mathbb{N}$ mit $2 \leq k \leq \frac{n}{4}$

$$t_2 = t_3 = 1, \qquad t_4 = 2$$

$$t_{4k-3} = t_{4k-2} = t_{4k-1} = 2k-1, \qquad t_{4k} = 2k$$

Durch Einsetzen in die oben beschriebene Gleichung errechnet man

$$\begin{aligned} t &= t_2 + t_3 + t_4 + \sum_{k=2}^{\frac{n}{4}} (t_{4k-3} + t_{4k-2} + t_{4k-1} + t_{4k}) \\ &= 4 + \sum_{k=2}^{\frac{n}{4}} (8k-3) = 4 + 8\left(\frac{\frac{n}{4}\left(\frac{n}{4}+1\right)}{2} - 1\right) - 3\left(\frac{n}{4} - 1\right) = \frac{n(n+1)}{4} - 1 \end{aligned}$$

Aufgabe 3

(a) Der Algorithmus berechnet x^y.

(b) Der Algorithmus stoppt für jede Eingabe, da die Terminierung nur von der Eingabe y abhängt.

Ist $y \in \mathbb{N}$, so ist die Bedingung $y > 0$ in Zeile 2 erfüllt. y wird in der Schleife so lange um 1 dekrementiert bis schließlich $y = 0$ gilt und die Schleife nicht mehr betreten wird. Nach der Schleife folgt die Terminierung. Ist $y \in \mathbb{Z}, y \leq 0$, so wird die Schleife niemals betreten und das Programm terminiert.

Definiert man $0^0 := 1$, so gibt der Algorithmus für $y \in \mathbb{Z}, y < 0$ falsche Ausgaben. Wähle zum Beispiel $x = 2$ und $y = -1$, dann ist die Ausgabe des Algorithmus gegeben durch $r = 1$ und nicht durch $x^y = 2^{-1} = \frac{1}{2}$.

Der einfachste Bugfix wäre y nur aus der Menge $\mathbb{N}_0$ zu wählen. Eine andere Variante ist den Algorithmus mit dem Betrag von y aufzurufen. Sollte $y < 0$ gelten, so gibt man $1/r$ wieder, ansonsten r. Der folgende Pseudocode zeigt diese Variante.

Listing: Bugfix

function $\mathrm{power}(x \in \mathbb{R}, y \in \mathbb{Z}) \in \mathbb{R}$ {
  $r \leftarrow 1$
  $t \leftarrow |y|$
  while $(t > 0)$ {
    $r \leftarrow r \cdot x$
    $t \leftarrow t - 1$
  }
  if $(y < 0)$
    return $\frac{1}{r}$
  else
    return r
}

(c) Um weniger Multiplikationen auszuführen, ist es nötig Potenzen, die bereits berechnet wurden, nicht noch einmal neu zu berechnen. Seien $x \in \mathbb{R}$ und $y \in \mathbb{N}$ und die Menge N_n für ein $n \in \mathbb{N}$ wie vorher definiert. Dann gibt es ein $a \in \mathbb{N}$ und ein $b \in N_2$, sodass

$$y = 2a + b \qquad \Longrightarrow \qquad x^y = \left(x^2\right)^a \cdot x^b$$

Das Problem lässt sich damit rekursiv lösen, indem man nun auf die gleiche Weise z^a mit $z := x^2$ berechnet. Der Unterschied besteht darin, dass x^2 nur ein einziges Mal berechnet wird und nicht a-mal. Der folgende Quelltext stellt eine Implementierung dieser Variante in der Programmiersprache »C++« dar.

Listing: power_fast.cpp — schnellere Potenzberechnung

```
float power_fast(float x, int y){
    uint t = (y<0)?(-y):(y);
    float p = x;
    float r = 1;

    while (t>0){
        if (t & 0x01)
            r *= p;
        p *= p;
        t = t >> 1;
    }

    return (y<0)?(1.0f/r):(r);
}
```