Prof. Dr. Leif Kobbelt
Stefan Dollase, Ira Fesefeldt, Alexandra Heuschling, Gregor Kobsik

Lösung - Übung 4

Tutoraufgabe 1 (Ordnungen und Sortieren):

a) Wir definieren die Relationen $\sqsubset_2$, $\equiv_2$ und $\sqsubseteq$ auf natürlichen Zahlen als

$$\begin{array}{lll} n \sqsubset_2 m & \text{genau dann wenn} & n \bmod 2 < m \bmod 2\ , \\ n \equiv_2 m & \text{genau dann wenn} & n \bmod 2 = m \bmod 2\ , \\ n \sqsubseteq m & \text{genau dann wenn} & n \sqsubset_2 m \quad \text{oder} \quad n \equiv_2 m \text{ und } n \leq m \end{array}$$

Wobei $n \bmod 2$ den Rest nach Division von n durch 2 angibt.

Zeigen oder widerlegen Sie, dass $\sqsubseteq$ eine totale Ordnung ist.

b) Gegeben folgender Sortieralgorithmus der das Array a sortiert:

```
from random import shuffle

def is_sorted(a):
    i = 1
    while i < len(a):
        if a[i - 1] > a[i]:
            return False
        i += 1
    return True

def sort(data) -> list:
    """Shuffle data until sorted."""
    while not is_sorted(data):
        shuffle(data)
    return data
```

Ist dieser Sortieralgorithmus sinnvoll?

c) Die Laufzeiten der Sortieralgorithmen Bubblesort, Insertionsort und Selectionsort sind bereits im Bestcase sehr unterschiedlich.

Ist es möglich, die Bestcase Laufzeiten aller drei auf einfache Weise zu optimieren?

Falls ja, welche Konsequenzen hat dies für die Averagecase und Worstcase Laufzeit?

Lösung

a) Um zu zeigen, dass $\sqsubseteq$ eine totale Ordnung ist, müssen wir Reflexivität, Transitivität, Antisymmetrie und Trichotomie zeigen.

- Reflexivität: Für n gilt sowohl $n \bmod 2 = n \bmod 2$ als auch $n \equiv_2 n$. Zusammen mit $n \leq n$ haben wir schließlich $n \sqsubseteq n$.
- Transitivität: Für $n \sqsubseteq m$ und $m \sqsubseteq l$ haben wir vier Fälle:
  - l ist gerade, dann ist $l \bmod 2 = 0$ und somit muss $m \equiv_2 l$, $n \equiv_2 m$ und $n \equiv_2 l$. Dann ist aber auch $m \leq l$ und $n \leq m$, somit ist auch $n \leq l$. Zusammen ist somit $n \sqsubseteq l$.
  - l ist ungerade und m ist gerade. Dann ist $l \bmod 2 = 1$ und $m \bmod 2 = 0$. Somit muss $m \sqsubset_2 l$ und $n \equiv_2 m$, somit ist aber auch $n \sqsubset_2 l$. Schließlich folgt daraus direkt $n \sqsubseteq l$.
  - l ist ungerade, m ist ungerade und n ist gerade. Dann ist $l \bmod 2 = 1 = m \bmod 2$ und $n \bmod 2 = 0$. Also ist $n \sqsubset_2 m$ und $m \equiv_2 l$. Damm muss aber auch $n \sqsubset_2 l$. Schließlich folgt daraus direkt $n \sqsubseteq l$.
  - l ist ungerade, m ist ungerade und n ist ungerade. Dann ist $l \bmod 2 = m \bmod 2 = n \bmod 2 = 1$,

also ist auch $m \equiv_2 l$ und $n \equiv_2 m$ und somit ist auch $n \equiv_2 l$. Weiterhin muss wegen $m \sqsubseteq l$ und $n \sqsubseteq m$ auch $m \leq l$ und $n \leq m$ gelten, womit auch $n \leq l$ gilt. Zusammen folgt daraus $n \sqsubseteq l$.

- Antisymmetrie: Für $n \sqsubseteq m$ und $m \sqsubseteq n$ haben wir zwei Fälle:
  - m ist gerade, dann ist $m \bmod 2 = 0$, somit ist mit $n \sqsubseteq m$ auch $n \bmod 2 = 0$. Daraus folgt dass $n \equiv_2 m$. Dann ist aber weiterhin auch $m \leq n$ und $n \leq m$, woraus aus Antisymmetrie von $\leq$ folgt, dass $n = m$ ist.
  - m ist ungerade, dann ist $m \bmod 2 = 1$, somit ist mit $m \sqsubseteq n$ auch $n \bmod 2 = 1$. Daraus folgt dass $n \equiv_2 m$. Dann ist aber weiterhin auch $m \leq n$ und $n \leq m$, woraus aus Antisymmetrie von $\leq$ folgt, dass $n = m$ ist.
- Trichotomie: Sei $\sqsubset$ der strikte Anteil von $\sqsubseteq$. Hier haben wir vier Fälle:
  - Wenn m gerade und n gerade ist, dann ist $m \bmod 2 = 0 = n \bmod 2$, somit ist auch $m \equiv_2 n$. Da $\leq$ total ist, haben wir entweder $m < n$, $m = n$ oder $m > n$, womit auch entweder $m \sqsubset n$, $m = n$ oder $m \sqsupset n$ gelten muss.
  - Wenn m gerade und n ungerade ist, dann ist $m \bmod 2 = 0 < 1 = n \bmod 2$, womit direkt $m \sqsubset_2 n$ und damit auch $m \sqsubset n$ gilt.
  - Wenn m ungerade und n gerade ist, dann gilt ein symmetrischer Beweis zum 2. Fall.
  - Wenn m ungerade und n ungerade ist, dann gilt ein symmetrischer Beweis zum 1. Fall.

b) Der vorgestellte Sortieralgorithmus heißt in der Literatur Bogosort. Sein Nachteil ist, dass die Zeit seiner Terminierung nicht nach oben sicher abgeschätzt werden kann. Tatsächlich gibt es stets eine positive Wahrscheinlichkeit, dass der Algorithmus sich schlechter als eine gegebene Laufzeit verhalten kann. Damit ist der Algorithmus zum Sortieren ungeeignet. Selbst wenn wir dafür die erwartet Laufzeit berechnen würden, käme diese im Worstcase immernoch (ohne Beweis, da dieser sehr aufwändig wäre) auf eine Laufzeit in O((n+1)!).

c) Wir können am Anfang der Sortieralgorithmen stets einfach einmal testen, ob das Array bereits sortiert ist. Damit verbessert sich die Bestcase Laufzeit von allen dreien sofort, da die Anzahl der Vergleiche nur noch $\sim n$ ist, da das überprüfen nur maximal ein Vergleich für jedes Element kostet, und die Anzahl der Kopierungen auf 0, da wir beim überprüfen nichts kopieren müssen.

Im Averagecase ist es wahrscheinlich, dass die Liste bereits früh nicht korrekt sortiert ist, womit nur ein konstanter Mehraufwand besteht.

Im Worstcase wäre eine bis auf die letzten Elemente sortierte Liste für Bubblesort und Selectionsort unvorteilhaft. Hier müsste aber bei unserer Verbesserung erst fast die gesamte Liste überprüft werden, womit sich die Laufzeit um einen $+n$ Term verschlechtert.

Für Insertionsort wäre eine sortierte Liste sehr vorteilhaft, dagegen eine sehr unsortierte Liste unvorteilhaft. Hier wäre die Überprüfung also im Worstcase kein Mehraufwand - sie bringt aber auch im Bestcase kaum etwas.

Tutoraufgabe 2 (Memoization):

Für eine Biologie Präsentation wollen wir eine Menge von Affen als Beispiel heran nehmen, die zwar möglichst berühmt sind, aber auch weit über das Spektrum an Affen verstreut ist.
Wir nehmen dabei an, dass Affen in einem Baum $T = (A, E)$ strukturiert sind, wobei die Affenart a' direkter Nachfahre von a ist, falls $(a, a') \in E$ gilt. Wir suchen nun eine Teilmenge $B \subseteq A$ sodass keine Affenart in B direkter Nachfahre einer anderen Affenart von B ist, oder formal für alle $a, a' \in B$ gilt weder $(a, a') \in E$ noch $(a', a) \in E$. Jede Affenart a hat einen bestimmten Berühmtheitswert $b(a)$. Der Berühmtheitswert einer Teilmenge $B \subseteq A$ ist gegeben als die Summe der Berühmtheitswerte seiner Elemente, also $b(B) = \sum_{a \in B} b(a)$.
Nutzen Sie Memoization um einen Algorithmus zu entwerfen, der eine Menge $B \subseteq A$ findet, sodass keine Affenart in B direkter Nachfahre einer anderen Affenart von B ist und sodass $b(B)$ maximiert wird.

Lösung

Um Memoization zu nutzen, definieren wir zwei Funktionen die auf der Menge der Affenarten A definiert ist. `root_in(a)` gibt die maximale Summe an Berühmtheitswerten des Teilbaumes mit Wurzel a an, falls a ein Beispiel ist. `root_out(a)` gibt entsprechend die maximale Summe an Berühmtheitswerten des Teilbaumes mit Wurzel a an, falls $a \notin B$ ist. Wir können `root_in` und `root_out` rekursiv definieren als:

```
def max_popular(a):
    return max(root_in(a), root_out(a))

def root_in(a):
    sum = b(a)
    for child in a.children:
        sum += root_out(child)
    return sum

def root_out(a):
    sum = 0
    for child in a.children:
        sum += max_popular(child)
    return sum
```

Nun können wir Memoization nutzen, um die Rückgabewerte der Funktion `root_in` und `root_out` für jede Affenart a zu berechnen, wodurch wir jeden Rückgabewert maximal einmal berechnen müssen und damit auch nur drei Werte pro Affenart berechnen müssen.
Das Gesamtergebnis lässt sich nun als `max_popular(r)` berechnen wobei r die Wurzel des Evolutionsbaumes ist.

Tutoraufgabe 3 (Selectionsort):

Sortieren Sie das folgende Array mithilfe von Selectionsort. Geben Sie dazu das Array nach jeder Swap-Operation an.

2	3	9	6	7	4	1	5	8

Lösung

2	3	9	6	7	4	1	5	8

1	3	9	6	7	4	2	5	8

1	2	9	6	7	4	3	5	8

1	2	3	6	7	4	9	5	8

1	2	3	4	7	6	9	5	8

1	2	3	4	5	6	9	7	8

1	2	3	4	5	6	7	9	8

1	2	3	4	5	6	7	8	9