Theoretische Informatik I - Blatt 4 - Tutor: Martin Klossek

Sebastian Steinhorst

Aufgabe 4.1

Betrachtet man die möglichen Aussagen der Kollegenpaare im Worst - Case (Lügner verhalten sich wie ehrliche), kommt man zu folgendem Ergebnis:

Definiere e als ehrlich, l als lügen.

Betrachte mögliche Anordnungen und Aussagen der Kollegenpaare:

Kollegen sind e,e: sie sagen immer [e,e]
Kollegen sind e,l: sie sagen [e,l], [l,e] oder [l,l]
Kollegen sind l,l: sie sagen [e,e],[l,l],[e,l],[l,e]

Damit erkennen wir, dass auf jeden Fall ein Paar, das nicht [e,e] sagt, mindestens einen Lügner enthalten muss.
Die Kollegen werden nun in einer Reihe so befragt, dass jeweils Kollege a und Kollege a+1, Kollege a+1 und Kollege a+2.... miteinander befragt werden. Wird anders als [e,e] geantwortet, wird das Paar sofort eliminiert und die Abfrage geht mit dem Vorgänger und dem Nachfolger des Paares weiter. Am Ende der Reihe angelangt, muss mindestens ein ehrlicher übrigbleiben, da mehr als die Hälfte der Kollegen ehrlich ist. Im Worst – Case stellen wir für jedes Paar 2 Fragen, es gibt n-1 Paare, also ist im Worst – Case die Laufzeit 2n-2 Fragen. Den letzten der ehrlichen Kollegen können wir dann mit n-1 Fragen über die anderen Kollegen ausfragen. Insgesamt haben wir also O(n) Fragen.

Induktionsargumentation für Worst – Case:

Nehmen wir an, wir haben 3 Kollegen. Nach Voraussetzung müssen es dann 2 Ehrliche und 1 Lügner sein. Egal, in welcher Permutation die Abfragereihe auftritt, stellen wir fest, dass wir zwei Paare gebildet haben und so 4 = 2*3-2 Fragen gestellt haben. Dort, wo ein ehrlicher und ein Lügner aufeinandertreffen, erhalten wir eine von [e,e] abweichende Antwort und erhalten unseren ehrlichen Kollegen.
Nehmen wir an, die Methode funktioniere für n Kollegen. Für n+1 Kollegen Brauchen wir nach Voraussetzung eine Gruppe mehr, also kommen 2 Fragen hinzu und wir haben 2+2n-2 Fragen = 2+2(n-1) = 2((n+1)-1) Fragen. Damit zeigt sich, dass die Worst – Case Laufzeit für alle n>3 Kollegenzahlen gilt.

Aufgabe 4.2 a)

Algorithmus: Der Algorithmus beruht auf der Funktion `Höhe` mit dem Parameter v, einem Zeiger auf den Knoten eines Baumes. Der Initialaufruf erfolgt in `main` mit dem Zeiger auf die Wurzel des Baumes. `Höhe` sucht unter allen Kindern des übergebenen Knotens rekursiv den mit der maximalen Höhe aus und gibt dessen Höhe + 1 als Höhe des Vaterknotens zurück. Hat ein Knoten keine Kinder, ist dessen Höhe 0, was die Terminierungsbedingung für die Rekursion darstellt.

```
Höhe (v Zeiger auf Baumknoten)
{
     if v hat Kinder:
          für alle Kinder w1,...,wn von v:
               h=max(Höhe(w1),..., Höhe(wn))+1
          print v,h
          return h
     else:
          print v,0
          return 0
}

main (root Zeiger auf Baumknoten)
{
     Höhe(root)
}
```

Korrektheit: Induktive Argumentation über Höhe jedes Knotens: Die Höhe eines Knotens ohne Kinder (also Blatt) ist 0. Dies entspricht der Rekursionsverankerung und wird im else-Zweig der Funktion `Höhe` korrekt ausgegeben. Für einen Vater eines solchen Blattes wird die Höhe des Sohnes mit der größten Höhe, hier also 0, um eins erhöht zurückgegeben. Nehmen wir nun also an, für Höhe n arbeitet der Algorithmus korrekt. Betrachten wir nun einen Knoten der Höhe n+1. Für einen Sohn wurde also zuvor korrekt Höhe n ermittelt, der Schritt von Höhe n nach n+1 erfolgt also, indem die Höhe dieses höchsten Knotens um eins erhöht als Höhe des Vaterknotens zurückgegeben wird, die also n+1 ist. Das ist somit korrekt, jeder Vater nimmt die Höhe seines höchsten Sohns + 1 an.

Laufzeit: Für jeden Knoten v des Baumes wird `Höhe` genau einmal aufgerufen. Innerhalb eines Aufrufs von `Höhe` fällt nur konstanter Aufwand für die Ausgabe der Höhe an. Der Aufwand für die rekursiven Aufrufe ist dann O(|Kinder(v)|) = O(Anzahl Kinder von v). Die Kinder werden mittels Kind – Geschwister - Darstellung gefunden. Jeder Knoten außer der Wurzel ist Kind eines anderen Knoten, für den Baum T ergibt sich als Laufzeit somit:

$$\sum_{v \varepsilon T} O(|\, Kinder(v)\,|) = O(\sum_{v \varepsilon T} |\, Kinder(v)\,|) = O(|\, T\,|)$$

Aufgabe 4.2 b)

Algorithmus: Der Algorithmus beruht auf der Funktion `Durchmesser` mit dem Parameter v, einem Zeiger auf den Knoten eines Baumes. Der Initialaufruf erfolgt in `main` mit dem Zeiger auf die Wurzel des Baumes. `Durchmesser` ermittelt für den Baum unterhalb des übergebenen Knotens rekursiv den Durchmesser, also die größte ungerichtete Anzahl an Kanten zwischen zwei Knoten im Baum. Das kann entweder die maximale Summe der Höhen zweier nebeneinanderliegender Kinder + 2 sein, oder es existiert bereits ein Durchmesser innerhalb des Teilbaumes, der größer als diese maximale Höhensumme ist. Hat ein Knoten keine weiteren Kinder, ist dessen Durchmesser 0, was die Terminierungsbedingung für die Rekursion darstellt. Für den Algorithmus muss die Höhe jedes Knotens zuvor bekannt sein, wie in a) gezeigt, geht das in O(|T|). Wir gehen also davon aus, dass jeder Knoten eine Information über seine Höhe bereits enthält.

```
Durchmesser ( v Zeiger auf Baumknoten)
{
     if v hat Kinder:
          für alle Kinder w_1,...,w_n von v:
               return=max(Durchmesser(w_1),...,Durchmesser(w_n),Höhe(w_1)+
               Höhe(w_2)+2,Höhe(w_2)+Höhe(w_3)+2,...,Höhe(w_n-1)+Höhe(w_n)+2)
     else:
          return 0
}

main (root Zeiger auf Baumknoten)
{
     Durchmesser(root)
}
```

Korrektheit: Induktive Argumentation über den Durchmesser jedes Teilbaumes unterhalb eines Knotens v: Der Durchmesser eines Knotens ohne Kinder (also Blatt) ist 0. Dies entspricht der Rekursionsverankerung und wird im else-Zweig der Funktion `Durchmesser` korrekt ausgegeben. Für einen Vater eines solchen Blattes wird der Durchmesser folgendermaßen ermittelt:
Bilde das Maximum der mit 2 addierten Höhe jeweils zweier benachbarter Geschwister und des bisher bekannten größten Durchmessers im Teilbaum.
In unserem Fall erhalten wir also 2. Dies ist korrekt.
Nehmen wir nun also an, für zwei vollständig bestückte Teilbäume der Höhe n arbeite der Algorithmus korrekt. Der Durchmesser jedes Teilbaumes wäre dann 2n, wegen der Summe der linken und rechten Kanten der Menge n.
Betrachten wir nun einen vollständig bestückten Teilbaum der Höhe n+1, dessen Wurzel als höchste Söhne die zwei Teilbäume der Höhe n besitzt. Nach dem Algorithmus ergibt sich der Durchmesser als Summe dieser Höhen, also der längsten gerichteten Kantenfolge im Teilbaum, plus die zwei Kanten, die den Vater mit den beiden Söhnen verbinden, somit n+1 + n+1. Damit haben wir 2n+2 Kanten, also einen Durchmesser von 2(n+1). Wir vernachlässigen den Fall, dass es bereits einen Durchmesser gibt, der größer als die Summe der Höhen + 2 ist, da dieser Fall bereits auf der korrekten Ermittlung des Durchmessers vorangegangener Teilbäume beruht. Der Algorithmus arbeitet also korrekt.

Laufzeit: Für jeden Knoten v des Baumes wird `Durchmesser` genau einmal aufgerufen. Innerhalb eines Aufrufs von `Durchmesser` fällt damit auch nur noch pro Kind konstanter Aufwand für die Abfrage und Aufsummierung der Höhen an (2*Höhenabfrage und 1*Summierung pro Kind) -1. Der Aufwand für die rekursiven Aufrufe ist dann O(|Kinder(v)|) = O(Anzahl Kinder von v). Die Kinder werden mittels Kind – Geschwister - Darstellung gefunden. Jeder Knoten außer der Wurzel ist Kind eines anderen Knoten, für den Baum T ergibt sich als Laufzeit somit:

$$\sum_{v \epsilon T} O(|\, Kinder(v)\,|) = O(\sum_{v \epsilon T} |\, Kinder(v)\,|) = O(|\, T\,|)$$