ETH Zürich
Institut für Theoretische Informatik
Prof. Dr. A. Steger
Ueli Peter

Algorithmen und Komplexität
Lösungsvorschlag Musterklausur 2

Lösungsvorschlag zu Aufgabe 1

a) Nein. Zum Beispiel bei einer bereits sortierten Sequenz benötigt Quicksort $\Omega(n^2)$ Vergleiche (wenn man das erste Element als Pivot benutzt).

b) Ja. Aus den Übungen kennen wir einen Algorithmus, welcher in $O(|E| + |V|)$ eine topologische Ordnung in einem gerichteten Graphen findet oder 'Nein' zurückgibt, falls der Graph einen gerichteten Kreis enthält.

c) Ja. Wir berechnen zuerst in $O(\log n)$ Operationen die Werte $n^{2^i} = n^{2^{i-1}} \cdot n^{2^{i-1}}$ für $i = 1, \ldots, \log n$. In einem zweiten Schritt betrachten wir die binäre Darstellung $b_\ell, b_{\ell-1}, \ldots, b_0$ von n. Es gilt, dass $n = \sum_{i=0}^{\ell} 2^i \cdot b_i$. Folglich können wir in weiteren $O(\ell) = O(\log n)$ Operationen

$$n^n = n^{\sum_{i=0}^{\ell} 2^i \cdot b_i} = \prod_{i=0}^{\ell} n^{2^i \cdot b_i}$$

berechnen.

d) Nein. Die Zahl n^n benötigt $n \log n$ viele Bits. Wir müssen sie demnach in $\Omega(n)$ vielen Registern speichern.

e) Nein. Sei $V := \{1, 2, 3, 4\}$ und $E := \{\{1, 2\}, \{2, 3\}, \{3, 1\}\}$. Der Graph $G = (V, E)$ hat $|E| = |V| - 1$ viele Kanten und enthält einen Kreis.

f) Ja. Aus $A \in P$ wissen wir, dass es einen Algorithmus gibt, welcher A in polynomieller Zeit entscheidet. Dieser Algorithmus kann also jede Eingabe effizient als Ja- oder Nein-Instanz prüfen.

g) Ja. Weil 2^{20} eine Konstante ist.

h) Ja. Weil $\log n = o(\sqrt{n})$.

i) Nein. Für jedes noch so kleine $\varepsilon > 0$ kann man ein n finden, so dass $e^{n \sin n} \leq e^{-n/2} < \varepsilon$.

j) Ja. $\lim_{n\to\infty} \sum_{i=1}^{n} \frac{n^i}{i!} = e^n - 1 = O(e^n)$.

Lösungsvorschlag zu Aufgabe 2

a) Wir definieren das Potential $\phi(i) :=$ Anzahl innerer Knoten mit 5 Kindern nach i Operationen.

Nun müssen wir die amortisierten Kosten der i-ten Operation abschätzen. Nehmen wir an, dass nach dem i-ten Einfügen $k \geq 0$ Rebalancierungen vorgenommen werden müssen. Da wir bei jeder Rebalancierung einen Knoten mit 5 Kindern verlieren, gilt

$$\phi(i) \leq \phi(i-1) - k + 1 \text{ und } t(i) \leq t(i-1) + k.$$

Folglich sind die amortisierten Kosten der i-ten Operation

$$a(i) = (t(i) + \phi(i)) - (t(i-1) + \phi(i-1)) \leq k - k + 1 = 1$$

und die totalen amortisierten Kosten für n Inserts betragen höchstens

$$\sum_{i=1}^{n} a(i) \leq n.$$

b) Da nun auch Knoten gelöscht werden können, muss unsere potential Funktion auch den Fall eines Unterlaufes einbeziehen.

Wir definieren folglich $\phi(i) :=$ Anzahl innerer Knoten mit 5 oder 2 Kindern nach i Operationen.

Ist die i-te Operation ein Insert so ist die Analyse analog zu a). Nehmen wir also an, dass die i-te Operation ein Delete ist und wir k mal wegen eines Unterlaufes rebalancieren müssen. Eine Rebalancierung wird nur dann vorgenommen, wenn die Nachbarn des Knotens bereits nur 2 Kinder haben. In diesem Fall wird der Knoten selbst mit einem Nachbarn verschmolzen und der gemeinsame Vater verliert ein Kind. Im anderen Fall, kann ein Kind eines Nachbars übernommen werden. Dadurch kriegen wir höchstens zwei neue Knoten mit Grad zwei. Es gilt daher

$$\phi(i) \leq \phi(i-1) - k + 2 \text{ und } t(i) \leq t(i-1) + k.$$

Folglich sind die amortisierten Kosten der i-ten Operation

$$a(i) = (t(i) + \phi(i)) - (t(i-1) + \phi(i-1)) \leq k - k + 2 = 2$$

und die totalen amortisierten Kosten für n Operationen betragen höchstens

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

Lösungsvorschlag zu Aufgabe 3

a) Wir beobachten, dass es genau dann wenn

$$b_{k-i} < a_i \text{ und } a_i < b_{k-i+1}$$

gelten, genau $k - i - 1 + i - 1 = k$ Elemente gibt, welche kleiner als a_i sind. Er reicht also aus diese beiden Bedingungen zu prüfen.

b) In den Übungen haben wir gesehen, wie man in $O(\log n)$ Operationen den Median von zwei sortierten Arrays bestimmen kann. Das k-grösste Element ist genau der Median der beiden Sortierten Arrays $[a_1, \ldots, a_k]$ und $[b_1, \ldots, b_k]$.

Lösungsvorschlag zu Aufgabe 4

a) Wir betrachten die Zeichenkette $s :=< s_1, \ldots, s_{2n} >$ mit $s_i = a$ für $1 \leq i \leq n$ und $s_i = b$ für $n + 1 \leq i \leq 2n$. Offensichtlich werden in den ersten n Rekursionsschritten von $pal(s, 1, 2n)$ immer zwei Aufrufe getätigt und somit ist die Laufzeit von $pal(s, 1, 2n)$ mindestens $\Omega(2^n)$. Wir folgern daraus, dass es für Eingaben der Länge n Zeichenketten gibt, welche eine Laufzeit von $\Omega(\sqrt{2}^n)$ erzwingen.

b) Unser Program soll für eine Zeichenkette der Länge n die Funktion $pal(s, 1, n)$ berechnen. Dazu definieren wir für $1 \leq i \leq j \leq n$ die Tabelle $T[i, j] := pal(s, i, j)$, welche wir mit folgendem Algorithmus iterativ ausfüllen können.

Offensichtlich benutzen wir die beschriebene Rekursion und berechnen den Eintrag $T[1, n]$. Es bleibt zu zeigen, dass wir nie auf noch nicht ausgefüllte Elemente zugreifen. Wenn wir $T[i, i + \ell]$ berechnen greifen wir auf die Elemente $T[i + 1, i + \ell - 1], T[i + 1, i + \ell]$ und $T[i, i +$

Algorithm 1 Pal(n)

```
for i = 1 ... n do
  T[i, i] := 0
  T[i, i − 1] := 0
end for
for ℓ = 1 ... n do
  for i = 1 ... n − ℓ do
    if s_i = s_{i+ℓ} then
      T[i, i + ℓ] = T[i + 1, i + ℓ − 1]
    else
      T[i, i + ℓ] = 1 + min{T[i + 1, i + ℓ], T[i, i + ℓ − 1]}
    end if
  end for
end for
return T[1, n]
```

$\ell - 1]$ zu. Wie man leicht sieht, wurden diese Elemente alle bereits in einer früheren Iteration berechnet. Da die äussere Schleife von 1 bis n und die innere von 1 bis $n - \ell \leq n$ geht, haben wir eine totale Laufzeit von $O(n^2)$. Für die Tabelle benötigen wir $O(n^2)$ Speicher.

c) Im ersten Schritt berechnen wir die Tabelle T wie im Teil b). Aus dieser Tabelle können wir durch aufrufen des Algorithmus $FindPal(1, n)$ ein optimales Palindrom finden. Wir benutzen den Operator $++$ um Zeichenketten miteinander zu verbinden.

Algorithm 2 FindPal(i, j)

```
if i = j then
  return s_i
else if i < j then
  if s_i = s_j then
    return s_i ++ FindPal(i + 1, j − 1) ++ s_j
  else if T[i, j] = T[i + 1, j] + 1 then
    return s_i ++ FindPal(i + 1, j) ++ s_i
  else
    return s_j ++ FindPal(i, j − 1) ++ s_j
  end if
end if
return T[1, n]
```

Der Algorithmus interpretiert offensichtlich rekursiv die in b) getätigten Entscheide. Da der Algorithmus in $O(n)$ Operationen mit $O(1)$ zusätzlichem Speicheraufwand läuft, verändern sich Laufzeit und Speichernutzung asymptotisch nicht.

Lösungsvorschlag zu Aufgabe 5

a) Wir transformieren das Problem in ein Graphen Problem. Wir ordnen jeder Koordinate einen Knoten im K_n zu und definieren das Gewicht einer Kante $\{u, v\}$ als die euklidische Distanz $d(u, v)$ zwischen u und v. Eine Lösung welche die Länge des Kabels minimiert entspricht nun genau einem minimalen Spannbaum. Ein solcher kann mit dem Algorithmus von Prim in $O(n^2)$ gefunden werden.

b) Nun müssen wir eine Menge von Kanten finden welche im Graphen genau zwei verbundene Komponenten induziert, so dass das Gewicht dieser Kanten minimiert wird. Eine solche Menge findet der Algorithmus von Kruskal wenn man die letzte Kante weglässt. Allerdings

hat der Algorithmus von Kruskal Laufzeit $\Omega(|E| \log |E|) = \Omega(n^2 \log n)$. Aus den Übungen wissen wir allerdings, dass die schwerste Kante in jedem minimalen Spannbaum eines Graphen gleichschwer ist. Wir können folglich wieder einen MST mit dem Algorithmus von Prim in $O(n^2)$ finden und die schwerste Kante durch einen Router ersetzen.

c) Um exakt wie in b) vorgehen zu können und von einem beliebigen MST die schwersten k Kanten zu ersetzen, reicht es zu beweisen, dass die Sequenz der Kantengewichte in jedem MST identisch ist. Um dies zu zeigen, betrachten wir einen Graphen $G = (V, E)$ und eine Gewichtsfunktion $w : E \longrightarrow \mathbb{N}$ und wir nehmen an, dass es zwei MST T_1 und T_2 gibt, so dass die sortierten Sequenzen der Gewichte $w_1^1, \ldots, w_1^n$ und $w_2^1, \ldots, w_2^n$ unterschiedlich sind. Wir nummerieren die Kanten von G von 1 bis $|E|$ und nehmen an, dass für $i \in \{1, 2\}$, T_i gemäss dieser Ordnung lexikographisch minimale Kanten Menge unter allen Spannbäumen mit Gewichtssequenz $w_i^1, \ldots, w_i^n$ hat.

Sei i die kleinste Zahl, so dass $w_1^i \neq w_2^i$ und wir nehmen ohne Verlust der Allgemeinheit an, dass $w_1^i < w_2^i$. Sei e die kleinste Kante mit Gewicht w_1^i, welche in T_1 aber nicht in T_2 ist. Offensichtlich enthält $T_2 \cup \{e\}$ einen Kreis C. Folglich gibt es eine Kante $f \in T_2$ welche nicht in T_1 ist, so dass $(T_1 \setminus \{e\}) \cup \{f\}$ und $(T_2 \setminus \{f\}) \cup \{e\}$ beides Spannbäume von G sind. Aus der Minimalität von T_1 und T_2 folgt $w(e) = w(f)$. Da beide Bäume aber auch lexikographisch minimale Kantensequenzen haben müssen die Kanten e und f identische Nummern haben, was einen Widerspruch zu unserer Nummerierung der Kanten ist. Folglich gibt es keine zwei minimale Spannbäume mit unterschiedlichen Gewichtssequenzen.

Lösungsvorschlag zu Aufgabe 6

Wir beweisen zuerst, dass Zero-One Linear Inequalities in NP liegt. Wie man leicht sieht, dient eine gegebene Lösung des Gleichungssystems als Zertifikat, welches einfach in $O(n \cdot m)$ überprüft werden kann.

Weiter zeigen wir Rucksack $\leq_p$ Zero-One Linear Inequalities. Sei $\phi = B, k, w_1, \ldots, w_n, p_1, \ldots, p_n$ eine Eingabe für das Rucksack Problem. Wir konstruieren in polynomieller Zeit eine Eingabe $f(\phi)$ für Zero-One Linear Inequalities, so dass gilt

$$\phi \in \text{Rucksack} \longleftrightarrow f(\phi) \in \text{Zero-One Linear Inequalities.}$$

Die Instanz $f(\phi)$ konstruieren wir wie folgt: $b_1 = K$, $b_2 = -B$ und für $1 \leq i \leq n$ gilt $a_{1_i} = p_i$ und $a_{2_i} = -w_i$.

Nehmen wir an, dass es eine Lösung $z_1, \ldots, z_n$ zu $f(\phi)$ gibt. Wir definieren $I := \{i \in [n] | z_i = 1\}$. Offensichtlich ist I eine Lösung zu ϕ.

Nehmen wir umgekehrt an, dass es eine Lösung I zum Rucksackproblem ϕ gibt. Wir setzten $z_i = 1$ genau dann wenn $i \in I$ und konstruieren so eine Lösung zu $f(\phi)$.

Die Transformation kann offensichtlich in $O(n)$ Operationen durchgeführt werden.