Abgabe 1. Dezember 2011
Adrianus Kleemans [07-111-693]
Gino Torriani [06-128-714]

Übungsserie 9 – Komplexität

Aufgabe 1 – Obere Schranke

Vermutung: p_k = O(n^k), wobei p_(n) = a_k*n^k + a_(k-1)*n^(k-1) + ... + a_1*n + a_0

Zu zeigen: f(n) ≤ c * g(n) für alle n ≥ n_0

Bei einem beliebigen p_(n) kann gemäss Beispiel folgende Schritte gemacht werden:

p_(n) = a_k*n^k + a_(k-1)*n^(k-1) + ... + a_1*n + a_0

≤ a_k*n^k + a_(k-1)*n^k + ... + a_1*n^k + a_0*n^k (für n_0 > 0)

≤ (a_k + a_(k-1) + ... + a_1 + a_0) * n^k (für n_0 > 0)

≤ (a_k + a_(k-1) + ... + a_1 + a_0) * g(n) (für n_0 > 0)

Somit sollte für ein Polynom vom Grad k eine obere asymptotische Schranke **(a_k + a_(k-1) + ... + a_1 + a_0) * g(n)** mit **n_0** > **0** existieren.

Aufgabe 2 – Bin Packing

Naiver Algorithmus:

1. Für jedes Element a_1 bis a_n (mit Zählerveriable *pos* = 0):
   - Überprüfe, ob es aktuellen Behälter noch Platz genug hat.
     - Wenn Platz: Füge das Objekt an Stelle *pos* ein.
     - Wenn kein Platz mehr:
       - Wenn letzter Behälter (*pos* = *k-1*), gebe **falsch** zurück.
       - Ansonsten, erhöhe *pos* und füge das Objekt ein.
   - Gebe **wahr** zurück.

Begründung, weshalb dies zu NP gehört:

- Lösungsraum: O(k^n), wobei k = Anzahl der Behälter und n = Anzahl der Objekte. Da er sozusagen „Brute Force“ alle Lösungsmöglichkeiten ausprobiert und deshalb den Lösungsraum durchläuft, ist er nicht in P (exponentielle, nicht polynomiale Berechnungszeit).
- Darauf aufbauend kann aber gezeigt werden: Es ist in linearer Zeit verifizierbar, ob ein gegebener Lösungsvorschlag richtig ist oder nicht (einen Lösungsvorschlag auszuprobieren kostet O(n)). Deshalb ist es in NP.

Abgabe 1. Dezember 2011
Adrianus Kleemans [07-111-693]
Gino Torriani [06-128-714]

Aufgabe 3 – DNF

Algorithmus:

1. Erstelle einen String-Array *list*[] und erstelle ein erstes Element, setze i = 0 [O(1)]
2. (je nach Form der DNF: Entferne alle ^ (und), wenn vorhanden [O(n)])
3. Gehe die gesamte Eingabe durch und unterteile sie [O(n)]
   - Wenn das Zeichen nicht **v** (oder) ist, kopiere das Zeichen ans Ende von list[i]
   - Wenn das Zeichen **v** (oder) ist:
     - Erstelle ein neues Element *list*[i+1]
     - Erhöhe i um 1.
4. Erstelle einen neuen String-Array *variables*[], zwei boolean-array *vars*[] und *varsNegated*[], einen boolean *isNegated* = false und setze i = 0 [O(1)]
5. Für jedes Element aus *list*, tue folgendes: [O(n*k)]
   - Für jedes Zeichen im Listenelement *list*
     - Wenn Zeichen = ¬, setze *isNegated* = true gehe eins weiter
     - Wenn nicht: Iteriere über *variables* und überprüfe, ob das Zeichen vorhanden ist.
       - Wenn vorhanden: Merke Stelle in *variables* als *pos*
       - Wenn nicht vorhanden: Füge zu variables hinzu und merke Position als *pos.*
       - Dann tue folgendes:
         - Wenn isNegated wahr: *varsNegated*[pos] = true
         - Wenn *isNegated* falsch: *vars*[pos] = true
     - Setze *isNegated* = false
   - Für jede Variable in *variables* mit Position *pos:*
     - Gehe *vars*[] und *varsNegated*[] durch, wenn bei einer bestimmten Position beide wahr sind, gehe zum nächsten Listenelement
     - Sind wir am Ende der Liste angelangt und der vorherige Fall ist nicht eingetreten, gebe **wahr** zurück.
6. Gebe **falsch** zurück.

Gesamte Laufzeit: O(2n + n*k) = O(n*k), wobei n = Länge der Eingabe, und k = Anzahl untersch. Variablen.

Für eine mögliche Implementation siehe die angehängten Java-Files inkl. Test-Output.