DAP2 UB6
Anja Rey, Gr.23 , Briefkasten 22

Max Springenberg, 177792

May 28, 2017

6.1 Gierige Algorithmen

6.1.1

Gegeben ist:
Distanz in Kilometern: l
Anzahl der maximalen Kilometer pro Tag: k
Index der Raststaette: $i, 1 \leq i \leq n$
Array, der jedem i eine Anzahl von kilometern zuordnet: A
$A[n] = l$
$0 < A[i+1] - A[i] \leq k$
Damit ist A aufsteigend sortiert.

Dem Algorithmus wird A uebergeben und es soll ein Array B zurueck gegeben werden, sodass $|B|$ minimal ist.
Gesucht wird also immer eine Raststaette die $A[i_n]$ moeglichst weit weg ist, wobei von ihr aus wieder eine Rasstaette in der Distanz k erreichbar sein muss.
Wir wissen, dass gilt:
$0 < A[i+1] - A[i] \leq k$
Damit koennen wir nach dem Index i_x, bei dem der Abstand von $A[i_0]$ zu $A[i_0]$ moeglichst nahe kleiner gleich k ist, suchen.
Diesen erhalten wir, an der stelle i mit:
A[i] $\leq$ k and A[i+1] > k
da A aufsteigend sortiert ist. Anschliessend erhoehen wir k um A[i], da die Distanz immer zum Startpunkt gemessen wurde.

```
Minimize(A)
1       k ←VALUE
2       tmpK ←k
3       counter ←2
4       for i ←1 to length[A] - 1 do
5           if A[i] ≤ k and A[i+1] > k then
6               counter ←counter + 1
7               tmpK ←k + A[i]
```

```
8       new Array B[1 ... counter]
9       tmpK ←k
10      for i ←1 to length[A] - 1 do
11            if A[i] ≤ k and A[i+1] > k then
12                  B[i] ←A[i]
13                  tmpK ←k + A[i]
14      B[counter] ←A[length[A]]
15      return B
```

6.1.2

Operationen nach lines geordnet:
$1 - 3 : \Theta(3)$

$ab\ 4 : \Theta(length[A1] - 1 = n)$
$5 : \Theta(1)$
$6 - 7 : \Theta(2)$

$8 - 9 : \Theta(2)$

$ab10 : \Theta(length[A] - 1 = n)$
$11 : \Theta(1)$
$12 - 13 : \Theta(2)$

$14 - 15 : \Theta(2)$

Daraus ergibt sich fuer Die Laufzeit im "Worst-Case":
$3 + n * (1 + 2) + 2 + n * (1 + 2) + 2$
$= 7 + 6n \in O(n), fuer\ c = 6$

6.1.3

Wird betrachten jeweils den Abstand h von A[n] zu A[i+n], $n > 0, i < length[A]$

Als Aussage nehmen wir, dass fuer alle A[i] ein A[i+n] mit A[i+n] $\leq$ k und A[i+n+1] > k existiert, sodass es keinen Abstand h $\leq$ k gibt, der naeher an k liegt, solange es sich nicht um die letzte Raststaette (das Ziel) handelt.

I.A.
$n = 1$

$A[n] = h_n$
$A[n + i] = h_{n+i}$

Es gilt:
$h_n < h_{n+i}$
$h = h_{n+i} - h_n$

Fallunterschiedung:
I $A[n+i+1] - A[n] > k$:
    es gilt:
    $A[n+i+m] - A[n] \geq A[n+i+1] - A[n] > k, m > i$
II sonst:
    i wird erhoeht
im Falle der letzten Rastaette gaebe es kein $A[n+i+1] - A[n] > k$ mehr und der letzte Wert des Arrays wird uebernommen.
Fuer n = 1 gibt es damit mindestens eine Rastaette (das Ziel), oder eine optimale vor dem Ziel.

I.V.
Die Aussage gelte fuer $n \in \mathbb{N}$ beliebig, aber fest.

I.S.
$n \to n+1$
Da das Problem auf n + 1 zu n kompatibel ist gilt auch hier die I.V.
Damit ist der Algorithmus optimal.

6.2 Gierige Algorithmen

6.2.1

Gegeben ist:
Simon kann sich nur eine Muenze pro Monat kaufen.
Array mit Muenzen:
A[1 ... n]
Array mit Faktoren fuer die Muenzen:
P[1 ... n]
Startpreis aller Muenzen:
20 (Euro)
Preissteigung je Monat:
A[i] ←A[i] * P[i]

Ferner gilt fuer den Preis p der Muenze am Index i nach m Monaten:
p = 20 * P[i]$^{m-1}$

Gesucht wird nun in Abhaengigkeit von P der minimale Gesamtbetrag, den Simon ausgeben muss, wenn er sich pro Monat eine Muenze kauft.
Dazu muss zunachst ein Array B mit den Indice von A ueber ihren Wert in P aufsteigend sortiert angelegt werden.
Anschliessend wird A durchgelaufen und von der Gleichung:
p = 20 * P[i]$^{m-1}$
gemacht und fuer jeden wert in A nach der durch B vorgegebenen Reihenfolge der Gesamtbetrag aufaddiert.

```
MinMoney(P)
1       new Array B [n ... length[P]]
2       B = revert(mergeSort(P))
3       endBetrag ←0
4       monthCounter ←1
5       for i ←1 to length(B) do
6           j ←B[i]
7           endBetrag ←endBetrag + (20 * P[j]^(monthCounter−1))
8           monthCounter ←monthCounter + 1
9       return endBetrag
```

6.2.2

Operationen nach lines geordnet:
$1 : \Theta(1)$
$2 : O(n * log(n))$
$3, 4 : \Theta(2)$

$5 : \Theta(n)$
$6, 7, 8 : \Theta(3)$
$9 : \Theta(1)$

Fuer die Laufzeit im "Worst-Case" ergibt sich:
$1 + n * log(n) + 2 + n * (3) + 1$
$= n * log(n) + 3 * n + 4 \in O(n * log(n))$

6.2.3

Der Algorithmus basiert darauf die Muenzen mit den groessten Werten aus P zuerst zu kaufen.

Zu zeigen ist nun, dass der Gesamtbetrag genau dann optimal ist, wenn wie beschrieben vorgegangen wird.
Dies gelingt, indem man zeigt, dass geraden mit groesserer Steigung nach einem bestimmten Zeitraum t einen hoeheren Wert erlangen als solche mit niedrigerer Steigung.

Gegeben sei:
$g_1(n) = i * n,\ g_2(n) = j * n,\ i < j$

I.A.
n = 1 (da wir auch im Monat 1 starten)
$g1(1) = i < j = g_2(1)$
I.V.
Die Aussage gelte fuer $n \in \mathbb{N}$ beliebig, aber fest.
I.S.
n $\rightarrow$ n + 1
$g_1(n+1) = (n+1) * i <^{I.V.} (n+1) * j = g_2(n+1)$

Damit ist offensichtlich der Endbetrag am kleinsten, wenn die Muenzen mit der groessten Wertsteigung zuerst gekauft werden und der Algorithmus optimal.