Derde serie opgaven

Bart van den Aardweg, Daan van Ingen,
Daan Meijers, Steven Raaijmakers

September 2015

1. Zie dominant.py

   (a) Het algoritme in de functie "dominant" is $O(n * log(n))$ omdat de array telkens door de helft gedeeld wordt waardoor er een balanced tree ontstaat, waarvan de hoogte bekent is: $O(log(n))$. Het maken van deze rijen kost $O(n)$ wat gezamenlijk $O(n * log(n))$ oplevert.

   (b) Ja, zie de boyermoore functie. Deze heeft $O(n)$ omdat hierbij elk element in de array een keer bezocht wordt. De functie returned echter ook een waarde wanneer een array geen dominante waarde heeft. Dit kan nog gecontroleerd worden maar dit zou een extra lineaire stap kosten.

   (c) Uitgaande van het feit dat 2 kleuren met elkaar gelijk zijn wanneer de rood-, groen- en blauwwaarde exact hetzelfde zijn is de klasse RGB gemaakt. Wanneer de algoritmes van a en b op het plaatje worden toegepast blijkt dat het plaatje geen waarde bevat die in meer dan de helft van de entries voorkomt.

   Testresultaten:

Algoritme A		**Algoritme B**	
Poging	**Tijd in sec**	**Poging**	**Tijd in sec**
1	2.34	1	0.92
2	2.30	2	0.90
3	2.28	3	0.93
4	2.32	4	0.91
5	2.30	5	0.91
Gemiddeld	**2.31**	**Gemiddeld**	**0.91**

2. a)
   Het maximale aantal punten dat er gehaald kan worden is vijf. Dit antwoord is te vinden door bij elk aantal dagen te kijken wanneer het het gunstigst is om een bepaald vak te volgen. Bij nul dagen is het slimste om te leren vak B, dit levert namelijk een punt op, terwijl dat niet het geval

is bij vakken A en C. Bij een enkele dag leren maakt het niet uit wat je leert, alles zal een enkele punt opleveren. Bij twee dagen leren is het het gunstigst om vak B of C te leren, deze leven beiden drie punten op en vak A dan slechts een enkele punt. Bij drie dagen leren is het gunstigste om te leren vak B, dit levert vier punten op terwijl vakken A en C slechts drie punten opleveren. Nu pakken we de gunstigste gevallen erbij en kijken we hoeveel punten de combinaties opleveren, te beginnen bij drie dagen vak B leren. Dit levert vier punten op. Stel dat we twee dagen vak B of C leren, dit zou betekenen dat we nog een dag hebben om een ander vak te leren. Wanneer er voor kiezen om twee dagen lang voor vak B te leren kunnen we nog een enkele dag leren voor vak A of C en niet voor het overgebleven vak. Hier maakt het niet uit wat we kiezen, we zullen op vier punten uitkomen. Als we voor vak C kiezen wordt het echter interessanter. Als we dan naast vak C een enkele dag vak B leren en niet voor vak A komen we uit op vier punten. Maar als we dit andersom doen (een dag voor A en geen voor C) komen we echter op vijf punten uit. Dit omdat zonder voor vak B te leren we toch een punt zullen verdienen. Om deze reden is het gunstigste om te leren: een enkele dag voor A, geen dagen voor B en twee dagen voor C. Dit is de enige manier om op vijf punten uit te komen.

3. Zie 3b.py

   (a) Bij een balk van lengte 20 die op plaatsen 5, 10 en 11 gezaagd moet worden geeft het greedy algoritme de volgorde [10, 5, 11] met als koste 20 + 10 + 10 = 40. Er is hier een betere oplossing: de volgorde [11, 5, 10] heeft als koste 20 + 11 + 6 = 37.

   (b)
```
"""
Gegeven een balk van lengte N, een prijs L voor het zagen in een balk van
lengte L en een aantal plaatsen waarop gezaagd moet worden.
Geeft de volgorde van plaatsen om te zagen om de kosten te minimaliseren.

Met hulp van:
http://stackoverflow.com/questions/21106595/cutting-a-stick-such-that-
cost-is-minimized
"""

import sys

def optimal_cost(cuts, length):
  # Positie van alle cuts plus het begin en eind (0 en de lengte van
  # de balk)
  cuts = [0] + cuts + [length]
  n = len(cuts)

  # 2D lijst van n*n voor sub-oplossingen koste en volgorde
  cost = [([0] * n) for row in xrange(n)]
  order = [([''] * n) for row in xrange(n)]
```

```
  # Loop van 2 tot en met n - 1
  for right in range(2, n):
    # Loop van right - 2 tot en met 0, groot naar klein
    for left in range(right - 2, -1, -1):
      cost[left][right] = sys.maxint

      # Zoek de meest optimale cuts in dit subprobleem
      for cut in range(left + 1, right):
        sum_cost = cost[left][cut] + cost[cut][right]

        if sum_cost < cost[left][right]:
          # Betere oplossing gevonden
          cost[left][right] = sum_cost
          order[left][right] = str(cuts[cut]) + ',' + \
            order[left][cut] + order[cut][right]

      # Tel de huidige lengte op bij de koste
      cost[left][right] += cuts[right] - cuts[left]

  # Optimale koste en volgorde op index [0][n - 1]
  return [cost[0][n - 1], order[0][n - 1][0:-1]]

solution = optimal_cost([6,13,17,20,30,40,44,50,56,57,67,74,84,93], 100)
print 'Koste:', solution[0]
print 'Volgorde:', solution[1]
```

Complexiteit $O(n^3)$

(c) Koste: 387
Volgorde: 40,20,13,6,17,30,67,50,44,57,56,84,74,93

4. Zie 4b.py

(a) Bij 4 kies je zelf posities om te zagen, bij 3 moet je je aan gegeven plaatsen houden. Bij 4 is er voor een aantal lengtes een bepaalde (verschillende) prijs, bij 3 is de prijs altijd gelijk aan de lengte. 3 is met dynamic programming op te lossen in $O(n^3)$ en 4 in $O(n^2)$.

(b)
```
"""
Geef de optimale cuts voor een bepaalde prijstabel en lengte.

Aangepaste versie van:
http://www.java.achchuthan.org/2014/08/dynamic-programming-rod-cutting.html
"""

import sys

def optimal_cuts(p, n):
  # Vul aan met nullen
  if n > len(p) - 1:
```

```
    p = p + [0] * (n - len(p) + 1)

  r = [0] * (n + 1)
  s = [0] * (n + 1)
  r[0] = 0

  # Vul de tabel
  for j in range(1, n + 1):
      q = -sys.maxint - 1
      for i in range(1, j + 1):
          if (q < (p[i] + r[j - i])):
              q = p[i] + r[j - i]
              s[j] = i

      r[j] = q

  # Tel de suboplossingen op
  result = []
  cost = 0
  while (n > 0):
    result += [s[n]]
    n = n - s[n]

  for i in result:
    cost += p[i]

  return result, cost

order, cost = optimal_cuts([0, 4, 5, 7, 9, 14, 15, 17, 20, 23, 27], 100)
print 'Koste:', cost
print 'Volgorde:', order
```

Complexiteit O(n^2)
Output:
Koste: 400
Volgorde: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

5. (a) Het doel is om het uit te geven bedrag te maximaliseren. Dit kan het best worden gedaan door over een zo lang mogelijke tijd, over een zo groot mogelijk bedrag de constante vergrotingsfactor a te ontvangen.
Stel $\mathrm{u_t} = 0$
$x_0 = x_\mathrm{t}$
$x_1 = x_\mathrm{t} + a \cdot x_0$
$x_2 = x_\mathrm{t} + a \cdot x_\mathrm{t} + a^2 \cdot x_\mathrm{t}$
.
.

$x_{20} = x_{\mathrm{t}}(1 + a + a^2 + ... + a^{19})$

Een dynamic programming algoritme die de uitgaven maximaliseert zal voor ieder jaar zoveel proberen zo veel mogelijk te sparen zodat in het volgende jaar meer uitgegeven kan worden.

(b) Stel dat L niks uitgeeft en paar jaar 2% rente ontvangt. Dan kan gebruik worden gemaakt van de volgende directe formule:
$FVA = C \cdot 1/r((1 + r)^n - 1)$
(http://www.investopedia.com/terms/f/future-value-annuity.asp)
Waarbij FVA het totaalbedrag is dat na n aantal jaar is gespaard met rente r en vaste periodieke stortingen C.
$FVA = 35.000 \cdot 1/0.02((1 + 0.02)^{20} - 1) =$ €850407,94