Aufgabe 3: Voll daneben

Team-ID: 00922

Team-Name: Zweiundvierzig

Bearbeiter/-innen dieser Aufgabe:
Franz Miltz

25. November 2018

Inhaltsverzeichnis

1 Lösungsidee

Das Problem kann in zwei Schritten gelöst werden. Die Eingaben (d.h. die von den Spielern gewählten Zahlen) werden immer als sortierte Liste betrachtet.

Zunächst muss festgestellt werden, was die Intervalle kosten, wenn man genau eine Zahl in jenes Intervall setzt. Dabei ist klar, dass der Preis minimal wird, wenn die eigene Zahl in der Mitte liegt. Dabei ist der Begriff Mitte vollkommen losgelöst von den Werten, die die einzelnen Eingaben haben. Die Mitte ist immer dort, wo sowohl links, als auch rechts, gleich viele Eingaben befinden. Bei einer ungeraden Anzahl von Werten ist meine eigene Zahl also gleich der mittleren. Bei einer ungeraden Anzahl kann ich zwischen den beiden mittleren oder irgendeiner Zahl dazwischen wählen. Allgemein kann ich also die Anzahl der Eingaben im Intervall ganzzahlig dividieren, um die Mitte zu finden.

Auf diese Weise kann nun für jedes Intervall $[a, b]$ ein Preis und die damit verbundene Mitte errechnet werden. Wenn man beachtet, dass $a \leq b \leq n$, ergeben sich $\frac{1}{2}n(n-1)$ Paare (a, b). Wenn wir davon ausgehen, dass ermitteln des Preises eines Segmentes in konstanter Laufzeit erfolgt, ist dieser Teil des Problems in $O(n^2)$ lösbar.

Anschließend müssen diese Segmente möglichst optimal zusammengesetzt werden. Dazu kann man versuchen ein optimales Interval $[1, b]$ aus den Intervallen $[1, a-1]$ und $[a, b]$ zu ermitteln. Der hintere Teil

wurde bereits, wie oben beschrieben berechnet. Der vordere Teil ist ebenfalls bereits vorhanden, wenn man in der richtigen Reihenfolge vorgeht. Denn jenes Intervall enthält genau eine Zahl der Bank weniger, als das was insgesamt errechnet werden soll. Somit kann das Problem der Reihe nach mit 1 bis 10 Zahlen gelöst werden. Für jedes der n Intervalle ergeben sich also $n-2$ (beide Teilintervalle müssen mindestens ein Element haben) mögliche Aufteilungen, von denen die beste gefunden werden muss. Die Preise der Teilintervalle wurden bereits berechnet, sind also in konstanter Laufzeit verfügbar. Also lässt sich auch diese Problem in $O(n^2)$ lösen.

Um dann die vollständige Liste der zehn Zahlen zu erhalten, muss einfach zurückverfolgt, welche Intervalle für jeden Schritt hinzugekommen sind. Deren Mitten sind dann das Ergebnis.

2 Umsetzung

Das Programm wurde für den Python 3.7.1 Interpreter geschrieben und auf einem Linux-System getestet. Es sind keine weiteren Packages erforderlich.

Es gibt zwei Möglichkeiten die Inputs an das Programm zu übergeben: Einerseits kann die Datei als Argument in der Command Line übergeben werden, andererseits kann das Programm auch mit dem Pipe-Operator (`<`) umgehen. Die Ausgabe ist stets der Preis und die Liste der sogenannten Picks (Zahlen, die von der Bank gewählt werden, damit der Preis minimal ist). Außerdem überprüft das Programm, ob diese Picks bei dieser Eingabe zu diesem Preis führen und teilt das Ergebnis mit.

Wie oben beschrieben, kann die Problemlösung in zwei Teile gegliedert werden. Analog dazu werden in der Klasse, die zur Lösung des Problems genutzt wird, `volldaneben_solver`, nacheinander die Methoden `set_segment_prices` (zum Berechnen der Einzelpreise aller Segmente) und `set_combined_prices` (zum Berechnen der optimalen Verkettung der Segmente) aufgerufen. Zuvor wird jedoch die Input-Liste sortiert.

2.1 Einzelsegmentpreise

Aus einer Liste von Eingaben zu ermitteln, was der minimale Preis ist, ist einfach. Man wählt das mittlere Element und berechnet die Summe der Beträge der Differenzen der Elemente zu jener Mitte.

```
1 def segment_price(self, segment):
    pick = segment[len(segment) // 2]
3   return sum([abs(x - pick) for x in segment])
```

Um nun diese Berechnung für jedes Segment zu speichern, benötigen wir eine geeignete Datenstruktur. Dabei habe ich mich für ein Dictionary entschieden. Mir ist bewusst, dass ein Array, wie es die Numpy-Bibliothek bereitstellt, Laufzeiteffizienter wäre. Ich möchte jedoch einerseits die vielen ungenutzten Elemente vermeiden und andererseits soll vermieden werden unnötige Imports zu verwenden.

In diesem Dictionary werden nun Paare (a, b) auf andere Paare (p, m) abgebildet. Dabei geben a und b das Intervall an, p ist der Preis des Segmentes und m die Mitte, die zu jenem Preis geführt hat. Um wirklich alle Werte zu errechnen, wird die `segment_price`-Funktion mit für alle $b \in [0, \texttt{len(inp)}]$ und alle dazugehörigen $a \in [0, b]$ aufgerufen. Dabei wird der Funktion ein Slice der Input-Liste übergeben, das alle Indizes $[a, b]$ umfasst.

2.2 Segmentvereinigungen

Auch hierfür wird ein Dictionary verwendet. Dabei werden Paare (i, b) auf $(p, a-1)$ abgebildet. Hierbei ist i die Anzahl der Picks im Intervall $[1, b]$, für das der Preis p ermittelt wurde. Das Intervall $[a, b]$ ist im letzten Schritt hinzugekommen. Somit kann mit a ermittelt werden, wie die optimale Verteilung der Picks zusammengesetzt ist.

Für $i = 1$ können bereits alle Werte eingetragen werden, denn diese sind äquivalent zu den Einzelsegmentpreisen. Um alle weiteren (i, b) in das Dictionary einzutragen, werden zwei Schleifen benötigt. Einerseits i im Intervall $[2, 10]$ (bzw. $[1, 9]$) und andererseits b im Intervall $[1, \texttt{len(inp)}]$. Nun wird man im Code das Intervall $[i, \texttt{len(inp)}]$ finden. Das liegt daran, dass i angibt, wie viele Picks bereits vor dem Intervall lagen. Somit müssen dort mindestens genauso viele Werte liegen, ansonsten wäre mindestens ein Pick doppelt und die Verteilung somit suboptimal. Für jedes a wird dann der Gesamtpreis aus dem Einzelsegment (a, b) und dem bereits optimierten Segment $(1, a-1)$ berechnet. Sollte dieser Preis besser sein, als der bisher beste, so wird er in das Dictionary eingetragen. Somit steht, nachdem alle a betrachtet wurden, die beste Aufteilung des Intervalls und der damit verbundene Preis im Dictionary.

Folglich beschreibt das Tupel $(10, \texttt{len(inp)} - 1)$ am Ende den optimalen Preis. Dieser Wert wird dann in `volldaneben_solver.price` eingetragen.

Zuletzt müssen die tatsächlichen Picks errechnet werden. Hierzu wird für alle i von 10 bis 1 die Mitte des Einzelsegments am Ende des optimalen Intervalls berechnet. Dabei wird das Intervall dadurch getrackt, dass b zunächst $\texttt{len(inp)} - 1$ ist und anschließend Schritt mit Hilfe des in `combined_prices` gespeicherten a verkleinert wird. ($a - 1$ ist das neue b)

3 Beispiele

Die Dateien für die Beispiele befinden sich im Ordner `beispieldaten`. Da das Programm deterministisch arbeitet, sollten sich die hier dargestellten Outputs reproduzieren lassen.

Die ersten drei Beispiele entsprechen den Vorgaben von der BwInf-Website. Anschließend habe ich mich mit den Primzahlen unter 1000 beschäftigt.

3.1 Beispiel 1 (BwInf)

Datei: `beispieldaten/beispiel1.txt`
Ausgabe (`output/beispiel1.txt`):

```
Inputs:  [5, 10, 15, 20, 25, 30, 35, 40, 45, 50,
          55, 60, 65, 70, 75, 80, 85, 90, 95, 100,
          105, 110, 115, 120, 125, 130, 135, 140, 145, 150,
          155, 160, 165, 170, 175, 180, 185, 190, 195, 200,
          205, 210, 215, 220, 225, 230, 235, 240, 245, 250,
          255, 260, 265, 270, 275, 280, 285, 290, 295, 300,
          305, 310, 315, 320, 325, 330, 335, 340, 345, 350,
          355, 360, 365, 370, 375, 380, 385, 390, 395, 400,
          405, 410, 415, 420, 425, 430, 435, 440, 445, 450,
          455, 460, 465, 470, 475, 480, 485, 490, 495, 500,
          505, 510, 515, 520, 525, 530, 535, 540, 545, 550,
          555, 560, 565, 570, 575, 580, 585, 590, 595, 600,
          605, 610, 615, 620, 625, 630, 635, 640, 645, 650,
          655, 660, 665, 670, 675, 680, 685, 690, 695, 700,
          705, 710, 715, 720, 725, 730, 735, 740, 745, 750,
          755, 760, 765, 770, 775, 780, 785, 790, 795, 800,
          805, 810, 815, 820, 825, 830, 835, 840, 845, 850,
          855, 860, 865, 870, 875, 880, 885, 890, 895, 900,
          905, 910, 915, 920, 925, 930, 935, 940, 945, 950,
          955, 960, 965, 970, 975, 980, 985, 990, 995]
Price: 4950
Picks: [945, 840, 735, 630, 525, 430, 335, 240, 145, 50]
```

Man sieht, dass die Gleichverteilung der Eingaben für einen sehr hohen Preis sorgt.

3.2 Beispiel 2 (BwInf)

Datei: `beispieldaten/beispiel2.txt`
Ausgabe (`ouptut/beispiel2.txt`):

```
Inputs:  [11, 12, 22, 27, 42, 43, 58, 59, 60, 67,
          69, 84, 87, 91, 94, 123, 124, 135, 167, 170,
          172, 178, 198, 211, 226, 229, 276, 281, 305, 313,
          315, 324, 327, 335, 336, 362, 364, 367, 368, 368,
          370, 373, 383, 386, 393, 399, 403, 413, 421, 421,
          426, 429, 434, 456, 492, 505, 526, 530, 537, 539,
          540, 545, 567, 582, 584, 586, 649, 651, 676, 690,
          729, 736, 739, 750, 754, 763, 777, 782, 784, 788,
          793, 802, 808, 814, 846, 857, 862, 862, 873, 886,
          895, 915, 919, 925, 926, 929, 932, 956, 980, 996]
```

```
Price: 1924
Picks: [929, 862, 777, 651, 539, 421, 368, 315, 172, 59]
```

3.3 Beispiel 3 (BwInf)

Datei: `beispieldaten/beispiel3.txt`
Ausgabe (`output/beispiel3.txt`):

```
Inputs:  [20, 40, 60, 80, 100, 100, 120, 120, 140, 160,
          160, 180, 200, 220, 220, 240, 240, 240, 240, 260,
          260, 260, 280, 280, 300, 300, 340, 340, 340, 340,
          360, 360, 380, 380, 400, 400, 420, 420, 440, 440,
          460, 460, 460, 480, 480, 500, 520, 520, 520, 520,
          520, 520, 540, 540, 540, 560, 580, 580, 580, 580,
          600, 600, 620, 640, 640, 640, 660, 680, 680, 680,
          680, 700, 700, 720, 720, 720, 720, 720, 740, 740,
          760, 780, 780, 800, 800, 820, 840, 840, 860, 860,
          860, 880, 900, 900, 920, 920, 960, 980, 980, 1000]
Price: 2160
Picks: [960, 860, 720, 660, 580, 520, 440, 340, 240, 100]
```

3.4 Beispiel 4 (Zweiundvierzig)

Primzahlen < 1000
Datei: `beispieldaten/beispiel4.txt`
Ausgabe (`output/beispiel4.txt`):

```
Inputs:  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
          37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
          79, 83, 89, 97, 101, 103, 107, 109, 113, 127,
          131, 137, 139, 149, 151, 157, 163, 167, 173, 179,
          181, 191, 193, 197, 199, 211, 223, 227, 229, 233,
          239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
          293, 307, 311, 313, 317, 331, 337, 347, 349, 353,
          359, 367, 373, 379, 383, 389, 397, 401, 409, 419,
          421, 431, 433, 439, 443, 449, 457, 461, 463, 467,
          479, 487, 491, 499, 503, 509, 521, 523, 541, 547,
          557, 563, 569, 571, 577, 587, 593, 599, 601, 607,
          613, 617, 619, 631, 641, 643, 647, 653, 659, 661,
          673, 677, 683, 691, 701, 709, 719, 727, 733, 739,
          743, 751, 757, 761, 769, 773, 787, 797, 809, 811,
          821, 823, 827, 829, 839, 853, 857, 859, 863, 877,
          881, 883, 887, 907, 911, 919, 929, 937, 941, 947,
          953, 967, 971, 977, 983, 991, 997]
Price: 4055
Picks: [947, 853, 751, 653, 569, 457, 353, 239, 137, 37]
```

4 Quellcode

4.1 Hauptklasse

```
# The main class to solve the problem; just for some parameters to be pseudo global
class volldaneben_solver:

  # constructor ...
  def __init__(self, inputs):
    # sorting inputs for the algorithm to work
    self.inputs = sorted(inputs)
    # getting prices for intervals [a, b] with exactly one pick
    self.set_segment_prices()
    # getting prices for intervals [1, b] with two to ten picks
```

```
    self.set_combined_prices()
    # setting self.values to the actual values
    self.set_values()

  # see self.__init__
  def set_segment_prices(self): #[...]

  # calculates the minimal cost of a given segment with one pick
  def segment_price(self, segment): #[...]

  # see self.__init__
  def set_combined_prices(self): #[...]

  # setting the picks from the self.combined_prices
  def set_values(self): #[...]
```

4.2 Einzelsegmentpreise

```
def set_segment_prices(self):
  # just shorter code
  inp = self.inputs
  # just shorter code
  l = len(inp)
  # (start, end) --> (price, pick)
  self.segment_prices = {}
  # cycling the end of the interval in [0, l)
  for b in range(l):
    # cycling the beginning of the interval in [0, b); a <= b!
    for a in range(b+1):
      # getting the price and writing it into the dictionary
      self.segment_prices[(a, b)] = (
        self.segment_price(inp[a:b+1]), (a+b)//2)

# calculates the minimal cost of a given segment with one pick
def segment_price(self, segment):
  # optimization & readability
  pick = segment[len(segment) // 2]
  # the price is the sum of deltas of each element of the segment to the pick
  return sum([abs(x - pick) for x in segment])
```

4.3 Segmentvereinigung

```
def set_combined_prices(self):
    # readability
    inp = self.inputs
    l = len(inp)
    # (pick count, end) --> (price, previous end)
    self.combined_prices = {}
    # setting the prices of segments with one pick
    for b in range(l):
        self.combined_prices[(1, b)] = (self.segment_prices[(0, b)][0], 0)
    # [1, 9] (i) ^= [2, 10] (i') picks
    for i in range(1, 10):
        # the end can't be less than i, 'cause there are i picks already
        for b in range(i, l):
            # the beginning of the single pick interval at the end of the combined one
            for a in range(i-1, b):
                # combinded prices without the single pick segment [a, b]
                prev = self.combined_prices.get((i, a-1), (0, a))
                # total price is the previous price + the single segment price
                price = prev[0] + self.segment_prices[(a, b)][0]
                # there might have been a previous calculation; the current best
                best = self.combined_prices.get((i+1, b), (sys.maxsize, 0))
                # the new price is only relevant if it's actually lower than the old one
                if best[0] > price:
```

```
                    # if so, updated the current best for [1, b] with i' picks
                    self.combined_prices[(i+1, b)] = (price, a)
    # the acutal price is the very last one
    self.price = self.combined_prices[(10, l-1)]
```

4.4 Wertberechnung

```
def set_values(self):
  # a list of picks ... (indices)
  picks = []
  # track [1, b]
  b = len(self.inputs)-1
  # each consecutive combined segment has one less pick
  for i in range(10, 0, -1):
    # get the data for the current segment
    current = self.combined_prices[(i, b)]
    # append the pick of the intervall [new b + 1, b] to picks
    picks.append(self.segment_prices[(current[1], b)][1])
    # the new end of the remaining interval
    b = current[1]-1
  # indices --> values
 self.picks = [self.inputs[x] for x in picks]
```

4.5 Hauptmethode

```
# main method; solve the problem and generate outputs
def volldaneben():
    # if no path is given, read input
    if len(sys.argv) == 1:
        inputs = from_input()
    # if a path is given, read file
    else:
        inputs = from_file(sys.argv[1])
    # solve the problem
    solver = volldaneben_solver(inputs)
    print('Inputs: ', solver.inputs)
    # price output
    print('Price:', solver.price[0])
    # picks output
    print('Picks:', solver.picks)
    # validating price from inputs and picks
    actual_price = get_price(solver.inputs, solver.picks)
    if solver.price[0] != actual_price:
        print('WARNING: Invalid output!')
        print('Actual price:', actual_price)

# calculating the price according to the task from inputs and picks
def get_price(inputs, picks):
    # price: sum of minimal differences of each input to any pick
    return sum([min([abs(i-p) for p in picks]) for i in inputs])

# calling the main function ...
if __name__ == '__main__':
    volldaneben()
```

4.6 Einlesen

```
# I/O: reading inputs from file
def from_file(path):
  inputs = []
  with open(path) as f:
```

```
    l = f.readline().strip('\n')
    while l != '':
      inputs.append(int(l))
      l = f.readline().strip('\n')
  return inputs

# I/O: reading inputs from input stream; < - opeartor
def from_input():
  inputs = []
  l = input()
  while l != '':
    inputs.append(int(l))
    l = input()
  return inputs
```