Aufgabe 5: Widerstand

Team-ID: 00922

Team-Name: Zweiundvierzig

Bearbeiter/-innen dieser Aufgabe:
Franz Miltz

26. November 2018

Inhaltsverzeichnis

1 Lösungsidee

Das Problem lässt sich in zwei gleichermaßen anspruchsvolle Teile gliedern. Einerseits muss herausgefunden werden, welche Widerstände man wie zusammen bauen kann und andererseits muss das Ergebnis visualisiert werden.

1.1 Ermittlung der Widerstände

Um meine Problemlösung zu verstehen, muss man sich überlegen, welche Arten von Widerständen es gibt. Gemäß der Aufgabenstellung können Widerstände entweder elementar sein oder aus genau zwei Teilwiderständen bestehen. Außerdem wurde durch die Limitierung auf vier Komponenten sichergestellt, dass die Stromrichtung stets eindeutig ist und jeder Widerstand in der Rechnung nur genau ein mal auftauchen kann. (Danke dafür an das BwInf-Team!)

Das bedeutet, dass ein Widerstand mit vier Teilen immer aus zwei Widerständen besteht, die entweder beide zwei Elementarwiderstände enthalten oder ein mal aus einem und ein mal aus drei Komponenten bestehen. Die Menge der Elementarwiderstände sei E und die Menge zusammengesetzter Widerstände C. Dabei ist C_i die Menge aller Widerstände, die sich aus i der gegebenen Elementarwiderstände bauen lässt.

$$C_1 = E \tag{1}$$

$$|C_i| = 2 \cdot \sum_{k=1}^{int(\frac{i}{2})} |C_k| \cdot |C_{i-k}| \tag{2}$$

Es können also alle zusammengesetzten Widerstände ermittelt werden, ohne die jeweils größeren zu kennen. Folglich kann damit begonnen werden alle Widerstände aus zwei Teilen zu generieren, dann jene aus drei Teilen usw. Dabei muss es jedoch vermieden werden, dass Elementarwiderstände mehrfach verwendet werden.

1.2 Visualisierung

Die Visualisierung ist von der Idee her ziemlich simpel. Das Ergebnis ist zwar nicht optimal, erfüllt aber den Zweck. Durch die verschiedenen Widerstandsarten kann hier rekursiv gearbeitet werden. Jeder Widerstand hat eine eigene Funktion, die auf einem Bild in einem bestimmten Bereich jenen Widerstand einzeichnet. Dabei verbinden Elementarwiderstände die beiden Anschlüsse und zeichnen einen Widerstand in die Mitte. Sequentielle Widerstände halbieren den Bereich horizontal, sodass in die beiden Teilrechtecke die Teile gezeichnet werden können. Prallele Widerstände haben einen Rand, in dem die Anschlüsse mit denen der Teile verbunden werden. Der restliche Teil des zur Verfügung stehenden Bereiches wird dann vertikal geteilt, sodass die beiden Teile übereinander gezeichnet werden können. In die Widerstände werden dann noch die Einzelwiderstände gezeichnet und in die Ecke des Bildes wird der Gesamtwiderstand geschrieben.

Mit dieser Herangehensweise müssen einige Konstanten von Anfang an gut gewählt werden, dazu jedoch mehr in der Umsetzung.

2 Umsetzung

Das Programm wurde für Python 3.7.1 auf einem Linux-System geschrieben und nur dort getestet. Benötigt wird außerdem eine Fork der PIL (Python Imaging Library), Pillow.

```
$ pip install pillow
```

Des Weiteren wurde die GNU FreeMono Schriftart (https://www.fontspace.com/gnu-freefont/freemono) verwendet. Das Programm (`widerstand.py`) sollte aus dem Ordner heraus ausgeführt werden, damit auf die Schriftart zugegriffen werden kann. Der Output befindet sich dann einerseits in der Konsole und andererseits wird eine Bilddatei (`output/rXXX.png`) erstellt. Dabei ist im Dateinamen der Gesamtwiderstand der Schaltung als ganze Zahl vorhanden. Die Software sollte als Proof-of-Concept betrachtet werden, da es an Nutzerfreundlichkeit mangelt.

Auch hier werde ich die Visualisierung getrennt vom Rest des Programms thematisieren.

2.1 Modellierung

Das Problem umfasst drei verschieden Widerstände. Diese wuden durch Klassen dargestellt. Die Basis-Klasse, `Resistor`, hat einen Wert (`Resistor.value`) und kann diesen über `Reistor.get_value` zurückgeben. Außerdem entspricht sowohl die Struktur, als auch die String-Repräsentation einfach dem eigenen Wert.

Weiterhin gibt es den `SequentialResistor`, der aus zwei Teilwiderständen (`part1` und `part2`) besteht, die in Reihe geschalten sind. Somit ist der Gesamtwiderstand gleich der Summe der beiden Teilwiderstände.

$$R = R_1 + R_2 \tag{3}$$

Der String besteht aus dem Gesamtwiderstand und der Struktur. Die Struktur besteht aus der Art des Widerstandes (`seq`) sowie den Strukturen der beiden Teilwiderstände.

Zuletzt existiert die Klasse (`ParallelResistor`), die Teilwiderstände in einer Parallelschaltung beschreibt. Folglich gilt für den Gesamtwiderstand R (`ParallelResistor.value`):

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} \tag{4}$$

$$\Leftrightarrow \frac{1}{R} = \frac{R_1 + R_2}{R_1 \cdot R_2} \tag{5}$$

$$\Leftrightarrow R = \frac{R_1 \cdot R_2}{R_1 + R_2} \tag{6}$$

Die String-Repräsentation sowie die Struktur verhalten sich analog zum sequentiellen Widerstand. Beide zusammegesetzte Widerstände erfordern zwei Teilwiderstände im Konstruktor. Der Basis-Widerstand erfordert lediglich einen Wert.

2.2 Berechnung der Widerstände

Mit der oben beschriebenen Lösungsidee und der soeben thematisierten Visualisierung wird es einfach, alle Widerstände zu berechnen. Zunächst muss jedoch die Liste der Elementarwiderstände in einen `Counter`, d.h. ein Dictionary mit der Anzahl der Widerstände, konvertiert werden. Diese Elementarwiderstände sind dann gleichzeitig zusammengesetzte Widerstände mit genau einem Teil.

Wenn nun also die `set_combined_parts`-Funktion aufgerufen wird, müssen ein paar Fälle behandelt werden. Sollte das übergebene $k < 2$ sein, so wird wurde bereits alles erledigt. Wenn die Widerstände für $k-1$ noch nicht berechnet wurden, muss dies zunächst erledigt werden. D.h., man muss die Methode nicht mehrmals aufrufen, die Rekursion sorgt dafür, dass alles nötige im Voraus berechnet wird.

Nun müssen alle Möglichkeiten, k auf die beiden Teilwiderstände zu verteilen, durchlaufen werden. Außerhalb dieser Schleife wird bereits eine Kandidatenliste erstellt, die nach und nach gefüllt werden soll. In der Schleife gibt es eine weitere Kandidatenliste, die am Ende auf Duplikate überprüft wird, bevor sie zu den eigentlichen Kandidaten hinzugefügt wird. Für jede der Verteilungen $(l, k-l)$ werden alle zusammengesetzten Widerstände mit l Teilen mit allen zusammengesetzten Widerständen mit $(k-l)$ teilen kombiniert. Für jede dieser Kombinationen gibt es zwei Kandidaten, d.h. sequentiell und parallel.

Das Entfernen der Duplikate erfolgt in einer sortierten Liste (nach Gesamtwiderständen), wobei benachbarte Elemente überprüft werden. Hierbei ist ein Duplikat ein Widerstand, der den gleichen Gesamtwiderstand mit den gleichen Elementarwiderständen erzeugt. Ich bin mir der Tatsache bewusst, dass dieser Ansatz nicht zu 100 Prozent funktioniert, da Duplikate eventuell durch andere Widerstände mit dem gleichen Gesamtwiderstand voneinander getrennt werden. In den meisten Fällen sollte diese Vorgehensweise jedeoch zielführend sein. Eine genauere Methode wäre in Bezug auf den Rechenaufwand unproportional zu dem Gewinn, der durch die verringerte Anzahl an Widerständen erzielt wird.

2.3 Finden des besten Widerstandes

Hierzu wird eine Binärsuche verwendet, die eine wesentlich bessere Laufzeit ($O(log(n))$) hat, als das einfache durchlaufen der Widerstandsliste ($O(n)$). Leider wird dieser Vorteil dadurch relativiert, dass die Liste sortiert werden muss, was bei dem Betrachten jedes Elementes nicht der Fall wäre.

In der Aufgabenstellung wird gefordert, dass für jedes $k \leq 4$ der zusammengesetzte Widerstand mit möglichst geringer Abweichung zu finden ist. Also wird eine Liste mit den besten Widerständen für jedes k erstellt. Des Weiteren kommen in jedem Schritt nur Widerstände hinzu. Es kann also mit einer leeren Liste an Kandidaten begonnen werden und in jeder Iteration wird die Liste erweitert und neu sortiert. Es gäbe sicherlich eine algorithmisch ausgepfeiltere Methode die neuen Widerstände mit den bereits vorhandenen zu vereinigen und gleichzeitig zu sortieren (s. Merge-Sort), aber die Laufzeit ist nicht unannehmbar und wie bereits erwähnt handelt es sich um ein Proof-of-Concept und kein nutzerfreundliches Endprodukt.

Es folgt die Binärsuche. Hierfür wird ein Intervall solange verkleinert, bis es entweder minimal groß ist oder die Mitte genau den gesuchten Wert enthält. Dabei umfasst das Intervall $[a, b]$ zunächst die gesamte

Liste. Anschließend wird die Mitte $c = \frac{a+b}{2}$ berechnet. Nun gibt es drei Fälle (berachtet wird immer der Gesamtwiderstand des Elementes):

1. Die Mitte ist größer als der gesuchte Wert.
   ⇒Die Mitte ist das neue rechte Intervallende b.

2. Die Mitte ist kleiner als der gesuchte Wert.
   ⇒Die Mitte ist das neue linke Intervallende a.

3. Die Mitte ist genauso groß wie der gesuchte Wert.
   ⇒Die Suche ist fertig.

Nachdem die Suche beendet ist, muss natürlich überprüft werden, welche Abbruchbedingung erfüllt wurde. Entweder ist $a + 1 = b$, dann muss überprüft werden, ob a oder b näher am gesuchten Wert liegt oder das ist nicht so, dann steht fest, dass c den gesuchten Wert enthält. Nun wird der beste Wert der Liste für die besten Widerstände für jedes k hinzugefügt. Sollte der exakte Wert gefunden worden sein, so wird die Liste mit jenem Widerstand gefüllt, da der Widerstand von k auch für $k + 1$ verwendet werden kann und keine geringere Abweichung mehr existiert.

2.4 Visualisierung

Der Code für die Visualisierung befindet sich hauptsächlich in den Widerstandsklassen. Es gibt jedoch eine Funktion, die die erste `draw`-Funktion aufruft und ein Bild erstellt. Außerdem ist diese Funktion für das Abspeichern verantwortlich. Da es sich hierbei um eine Bibliothek handelt, möchte ich den Quellcode genauer thematisieren.

```
def draw_resistor(r, path):
  width, height = 800, 600
  im = Image.new('RGBA', (width, height), (255, 255, 255, 255))
  dctx = ImageDraw.Draw(im)
  fnt = ImageFont.truetype('font/FreeMono.ttf')
  r.draw(dctx, fnt, 0, 0, width, height,
         width//9, width//27)
  dctx.text((10, 10), 'R␣=␣' + str(r.get_value()) + '␣Ohm', fill=(0, 0, 0, 255))
  im.save(path+'.png')
```

Zunächst wird die Größe des Bildes auf 800×600 festgelegt. Diese Werte haben keinen tieferen Sinn und eignen sich lediglich gut, um den Text lesbar zu halten, während die Schaltung noch auf das Bild passt.

Anschließend wird ein neues Bild erstellt, welches zunächst weißgefüllt ist. Außerdem wird die oben bereits beschriebene Schriftart geladen und eine Draw-Umgebung (*draw context*) erstellt. Nun kann der Widerstand mit Hilfe der `draw`-Methode des Widerstandsobjektes auf das Bild gemalt werden. Übergeben wird der Bereich, in dem gemalt werden darf (das gesamte Bild), die Schriftart, die Draw-Umgebung sowie die Größe des Widerstandes als Schaltungsbaustein. Diese Größe ist wichtig. Zunächst habe ich geschaut, welches Seitenverhältnis gut aussieht und bin auf 3:1 gekommen. Nun ist es wichtig, dass genügend Widerstände nebeneinander passen. Hierbei könnte man meinen, es müssten nur vier sein. Da das Bild jedoch immer halbiert wird und die Widerstände nicht zwingend gleichmäßig verteilt sind, müssen acht Bausteine einkalkuliert werden (In jeder Hälfte bis zu vier). Dabei kann der konstante Rand der parallelen Widerstände unbeachtet bleiben, da dieser nicht auftritt, wenn vier Widerstände (die maximale Anzahl) in Reihe geschalten werden. Folglich darf ein Widerstand nicht breiter als ein Achtel des Bildes sein. Wenn man ein Neuntel wählt, sind die Verbindungen zwischen den Widerständen noch erkennbar. Solange das Bild nicht mehr als drei mal so breit wie hoch ist, spielt die Höhe der Einzelwiderstände keine Rolle.

Nachdem die Schaltung gemalt wurde, wird der Gesamtwiderstand in die Ecke links oben geschrieben. Anschließend kann das Bild gespeichert werden.

Der Code für das Zeichnen der Einzelwiderstände ist sehr unverständlich. Am einfachsten ist hierbei der Reihenwiderstand, bei dem nur die `draw`-Methoden der beiden Teile für die jeweiligen Hälften des vorgesehenen Bereiches aufgerufen werden. Auch relativ einfach ist der Elementarwiderstand. Hier wird die Mitte berchnet und ein Rechteck gezeichnet, was die Größe hat, die durch `xr, yr` festgelegt wurde. In das Rechteck wird zentriert Text geschrieben. Außerdem werden die Mitten der Bildränder mit den Mitten der Rechtecksseiten verbunden.

Wirklich unübersichtlich ist die Parallelschaltung. Hier werden zunächst acht Punkte, vier für jede Seite bestimmt. Diese Pukte bezeichnen die vertikale Mitte des Bildrandes, die vertikale Mitte des inneren

Randes sowie die vertikalen Mitten der Rände der Teilwiderstände. Diese werden so, wie in den Beispielen zu sehen, verbunden. Falls es sie genauer interessiert, werfen Sie bitte einen Blick in den Quellcode und zeichnen Sie es ich auf, denn das ist zielführender als es hier in Worte zu fassen. Nachdem der benötigte Teil der Schaltung gemalt wurde, können nun die Teilwiderstände in den vorgesehenen Bereichen dargestellt werden.

3 Beispiele

Die Beispiele sind auf Grund der Bilder ziemlich umfangreich. Zu jedem Beispiel gehört eine Konsolenausgabe (die sich auch als `.txt`-Datei im `output`-Ordner finden lässt) sowie vier Bilder.

3.1 Beispiel 1 (R=500)

Ausgabe:

```
k=1: 470.0
k=2:   500.3802: par(4700.0, 560.0)
k=3:   500.0000: seq(100.0, seq(180.0, 220.0))
k=4:   500.0000: seq(100.0, seq(180.0, 220.0))
```

Bild für $k = 1$:

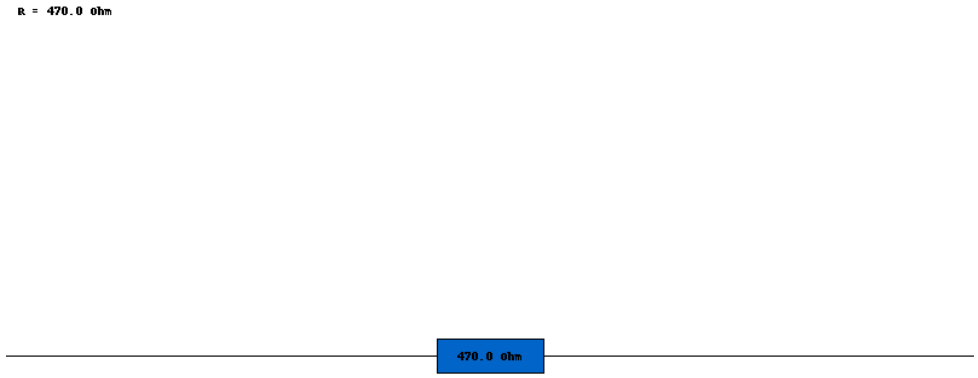


Bild für $k = 2$:

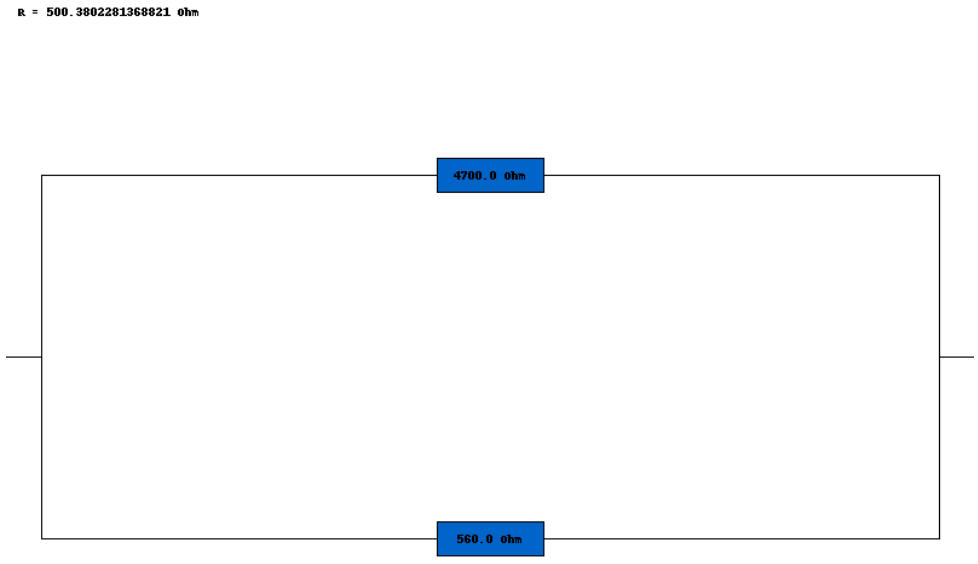


Bild für $k = 3$:

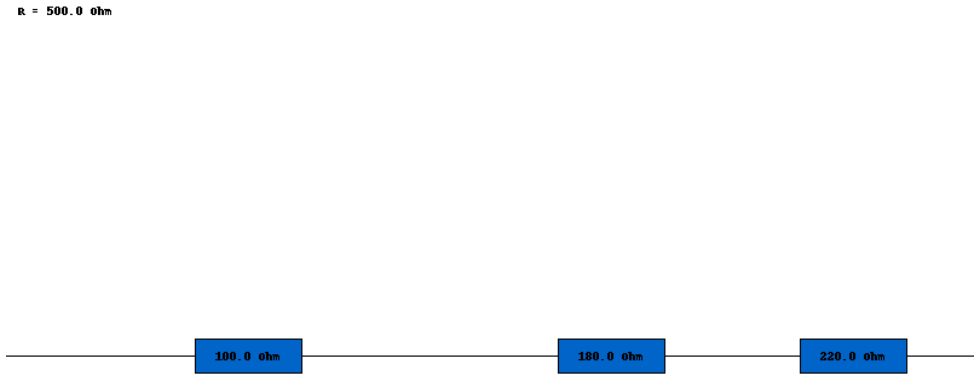
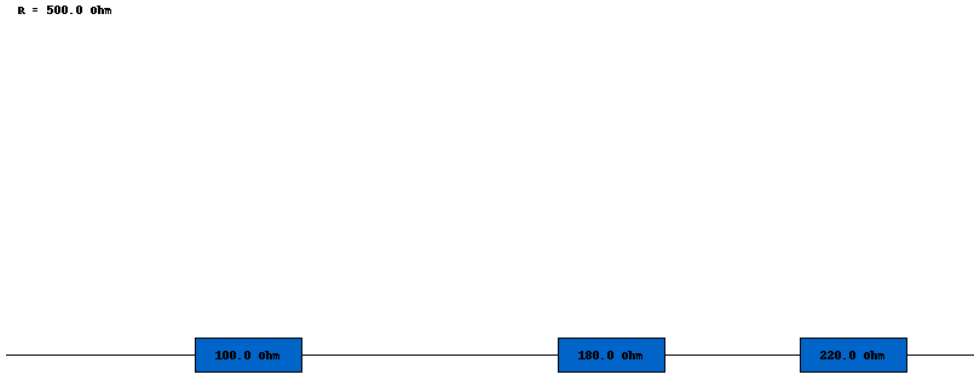


Bild für $k = 4$:



3.2 Beispiel 2 (R=140)

```
k=1: 150.0
k=2:   140.4255: par(150.0, 2200.0)
k=3:   139.9494: par(2700.0, par(820.0, 180.0))
k=4:   140.0000: par(150.0, seq(680.0, seq(220.0, 1200.0)))
```

Bild für $k = 1$:

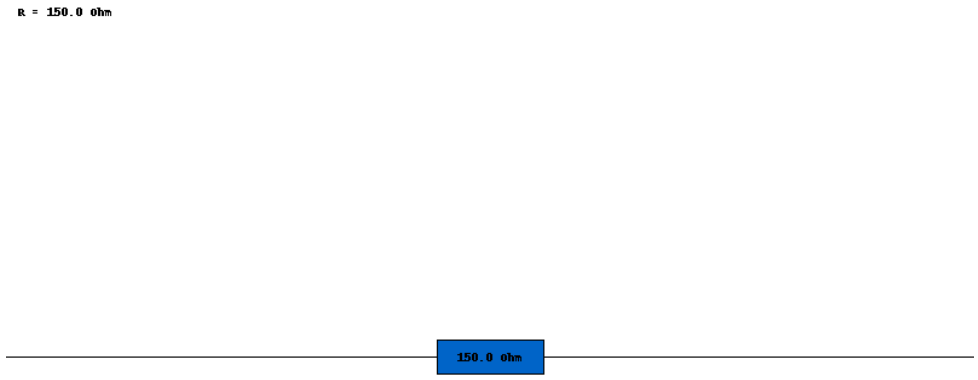


Bild für $k = 2$:

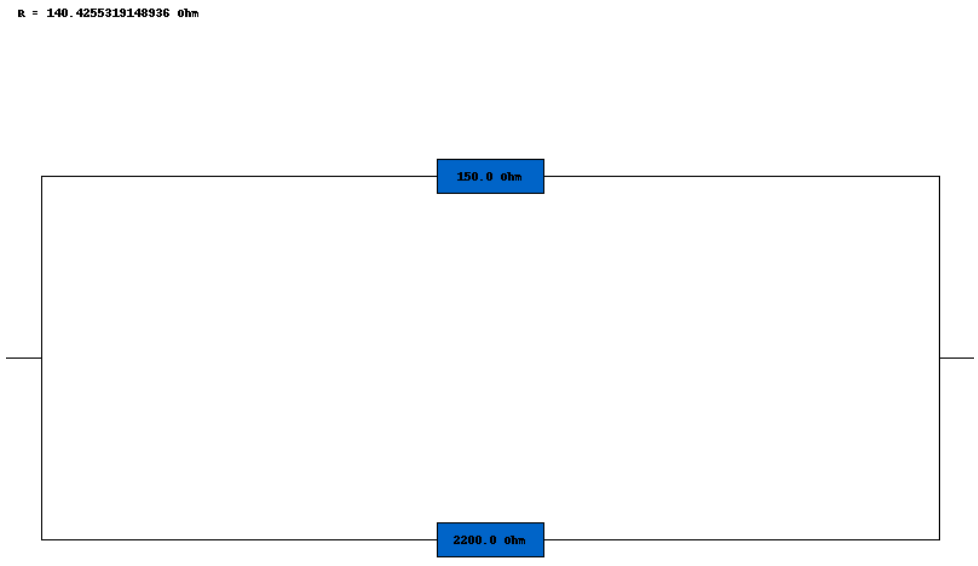


Bild für $k = 3$:

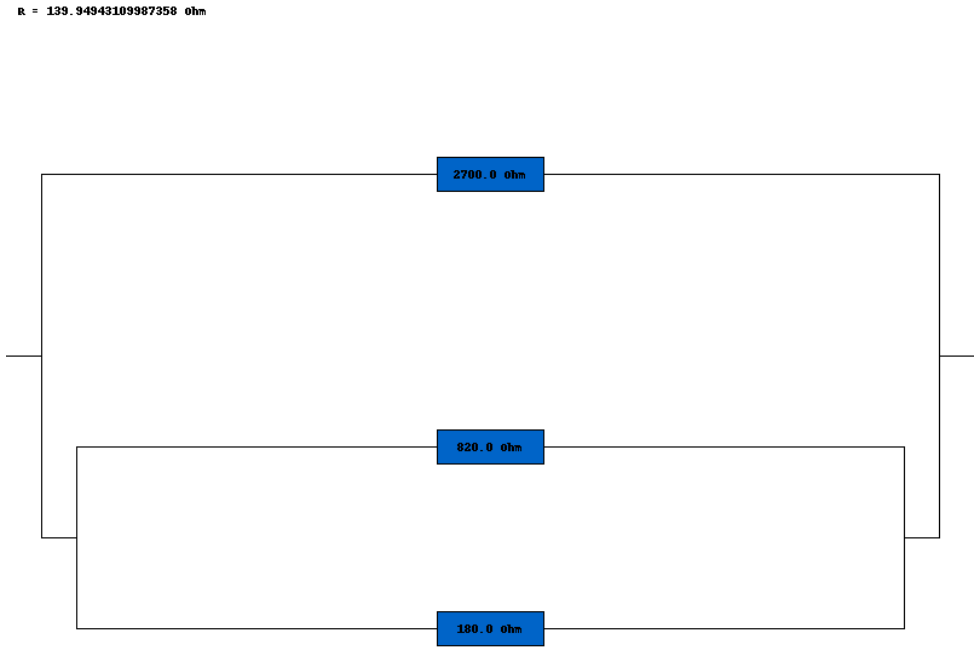
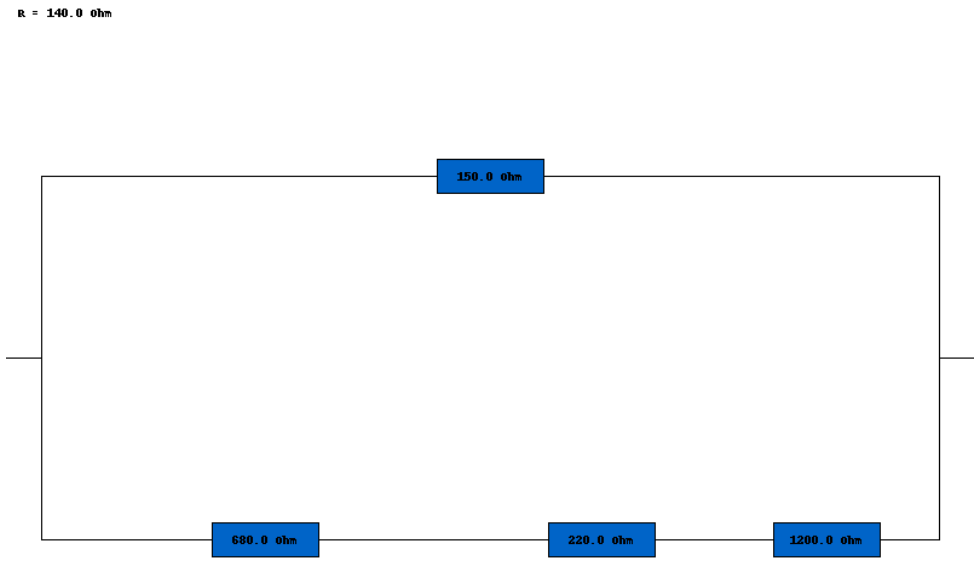


Bild für $k = 4$:



3.3 Beispiel 3 (R=314)

```
k=1: 330.0
k=2:   314.7265: par(6800.0, 330.0)
k=3:   314.0556: par(330.0, seq(1800.0, 4700.0))
k=4:   313.9993: par(1000.0, seq(120.0, par(1200.0, 470.0)))
```

Bild für $k = 1$:

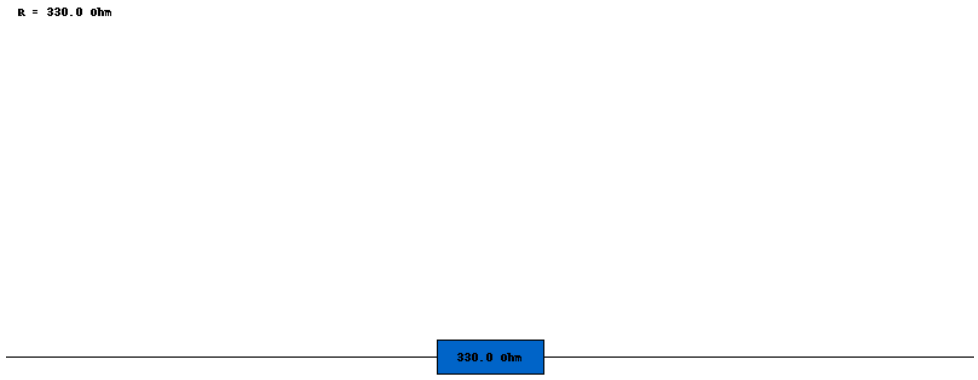


Bild für $k = 2$:

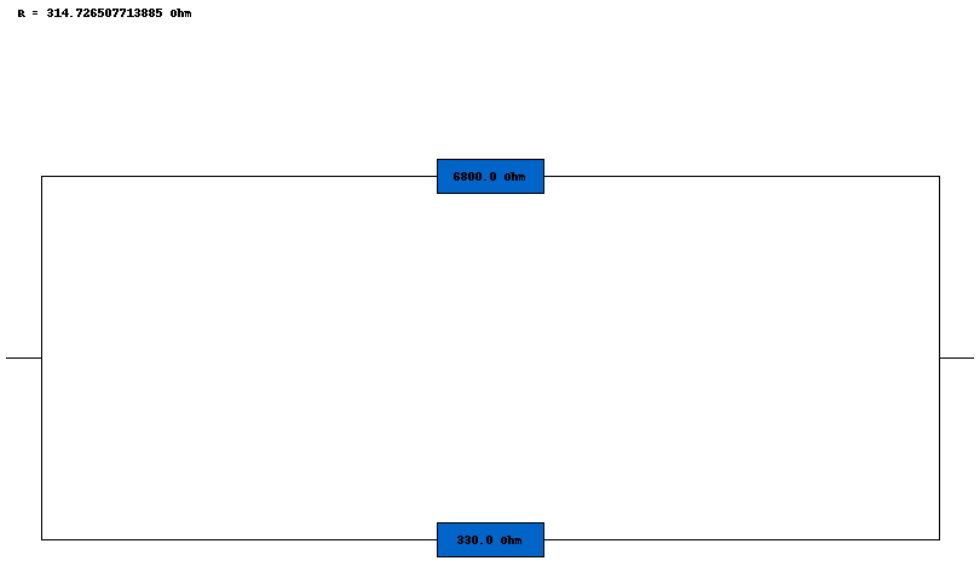


Bild für $k = 3$:

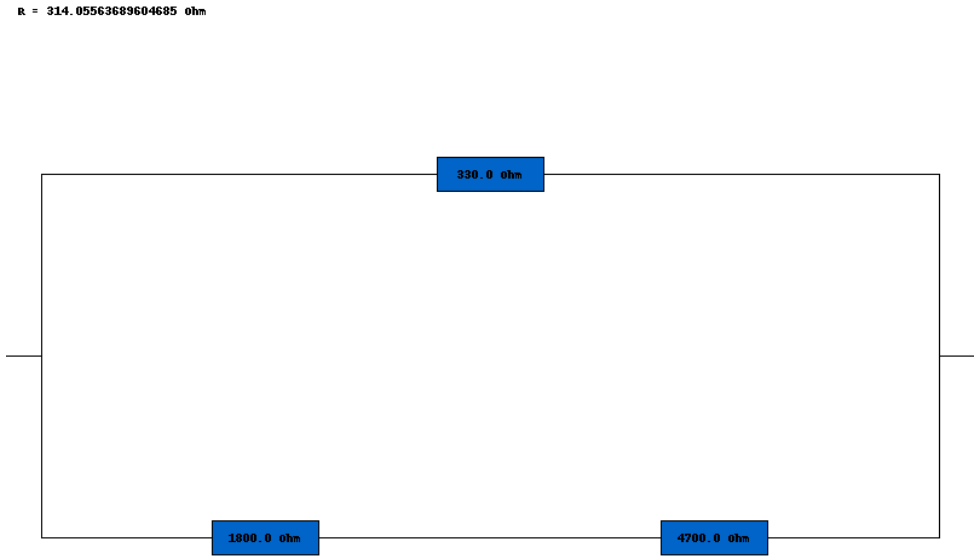
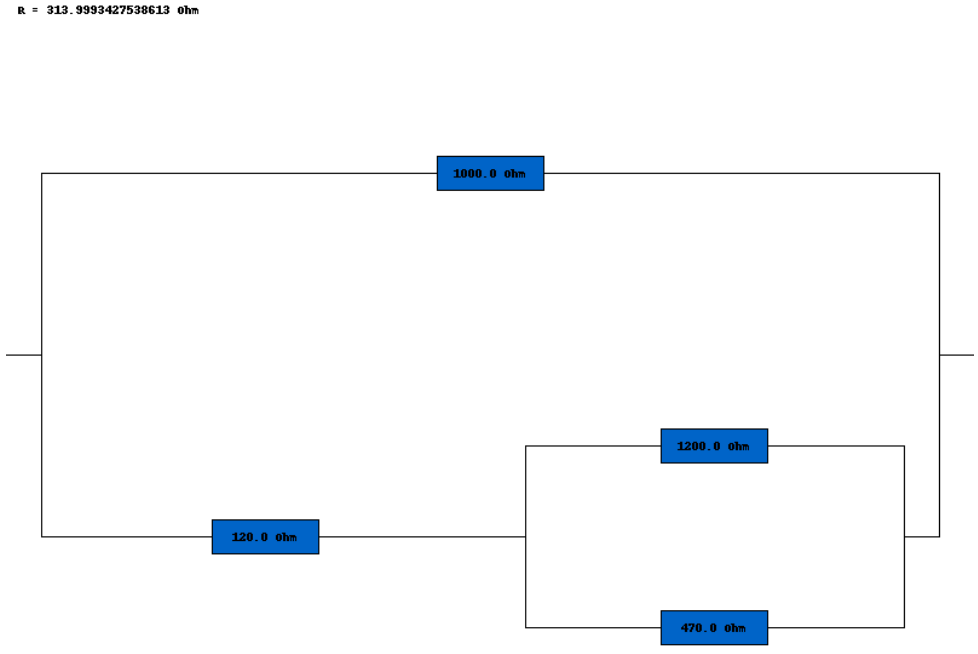


Bild für $k = 4$:



3.4 Beispiel 4 (R=315)

```
k=1: 330.0
k=2:   314.7265: par(330.0, 6800.0)
k=3:   315.0000: par(560.0, seq(390.0, 330.0))
k=4:   315.0000: par(560.0, seq(390.0, 330.0))
```

Bild für $k = 1$:

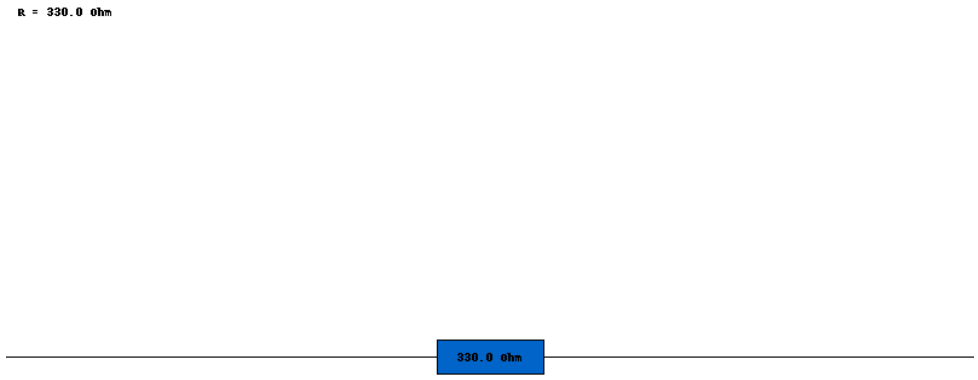


Bild für $k = 2$:

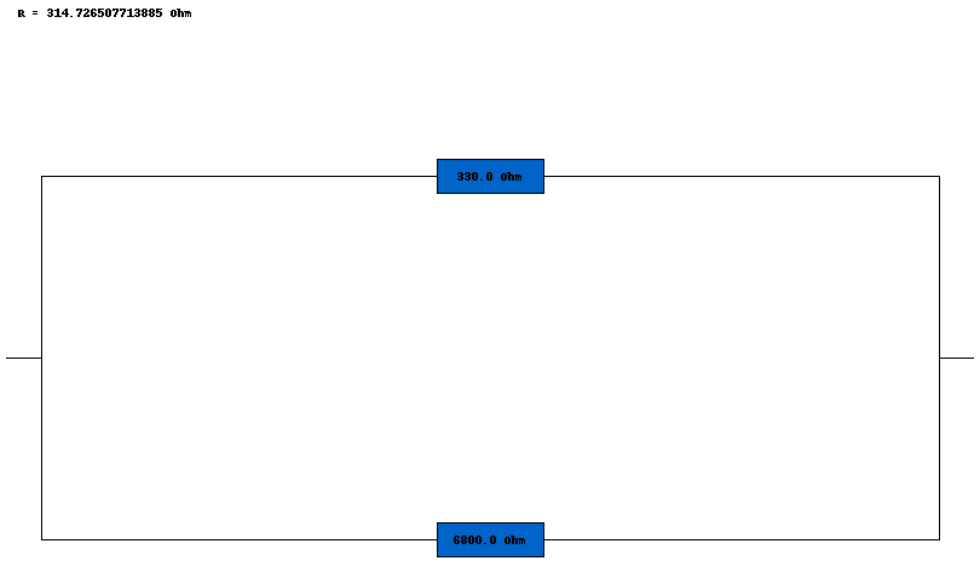


Bild für $k = 3$:

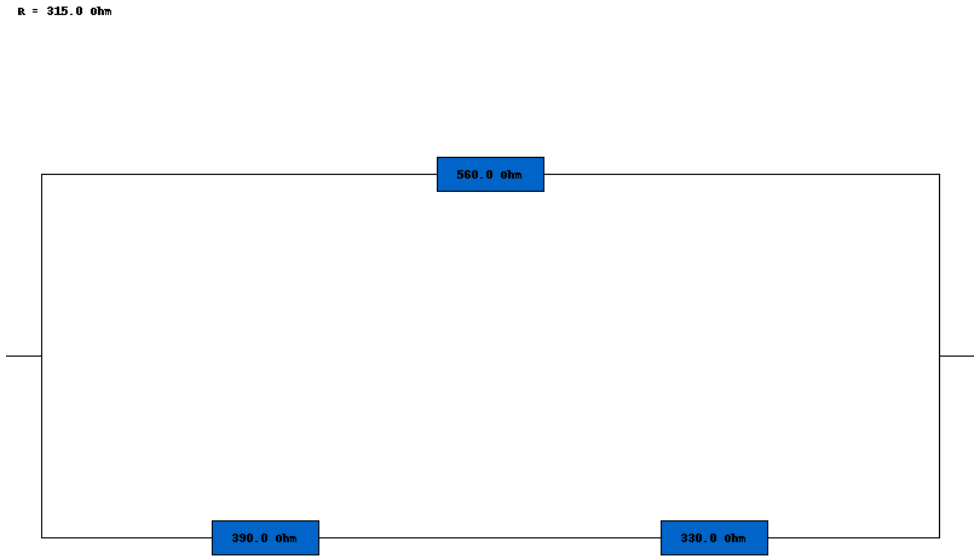
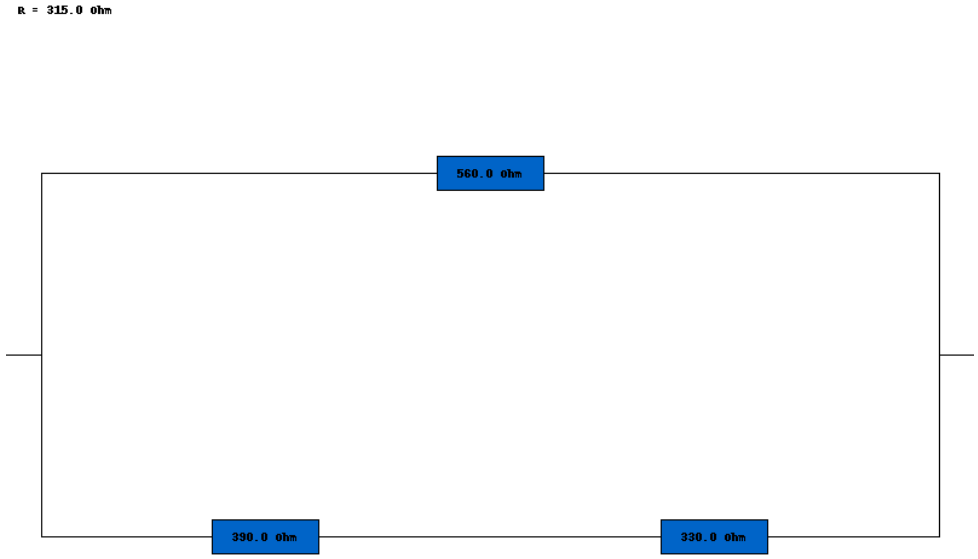


Bild für $k = 4$:



3.5 Beispiel 5 (R=1620)

```
k=1: 1500.0
k=2:  1620.0000: seq(1500.0, 120.0)
k=3:  1620.0000: seq(1500.0, 120.0)
k=4:  1620.0000: seq(1500.0, 120.0)
```

Bild für $k = 1$:

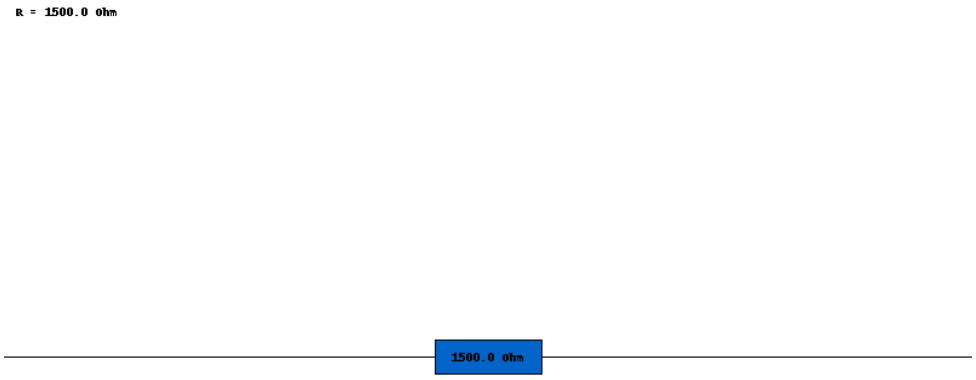


Bild für $k = 2$:

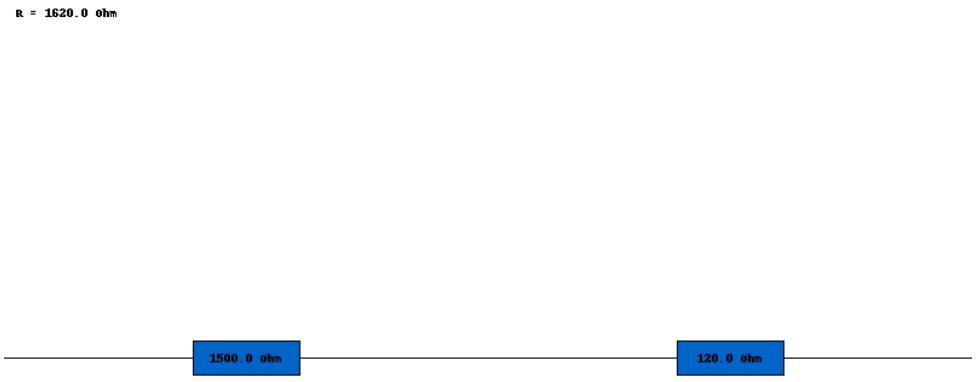


Bild für $k = 3$:

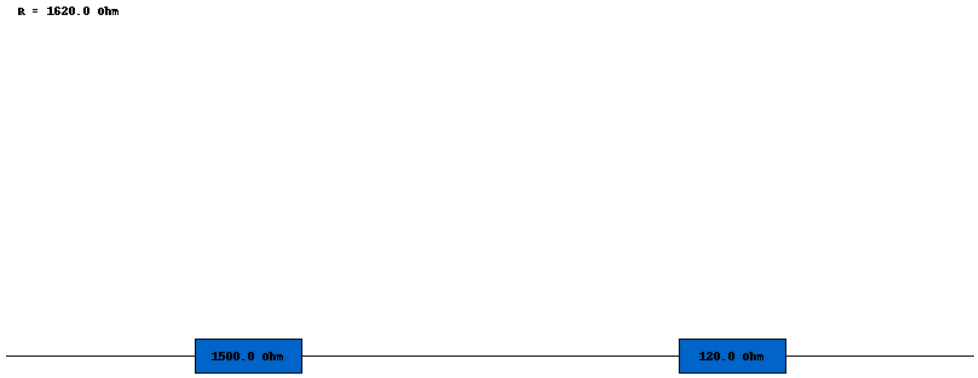


Bild für $k = 4$:

3.6 Beispiel 6 (R=2719)

```
k=1: 2700.0
k=2:  2700.0000: seq(1200.0, 1500.0)
k=3:  2720.0000: seq(1000.0, seq(220.0, 1500.0))
k=4:  2719.0000: seq(par(220.0, 180.0), seq(820.0, 1800.0))
```

Bild für $k = 1$:

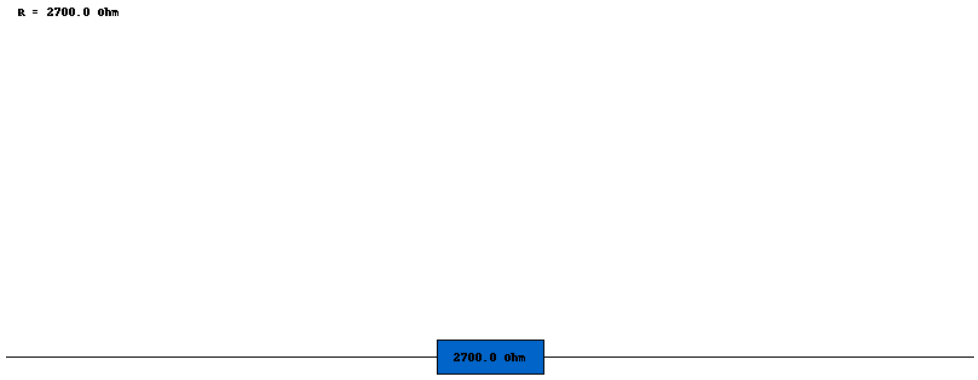


Bild für $k = 2$:

Bild für $k = 3$:

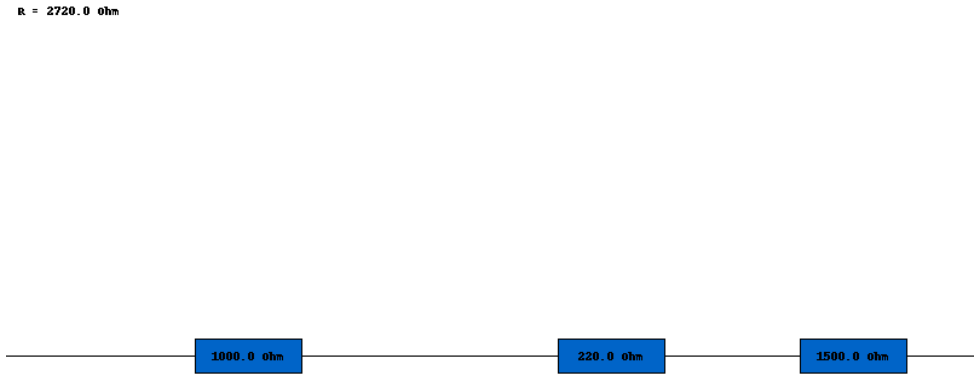
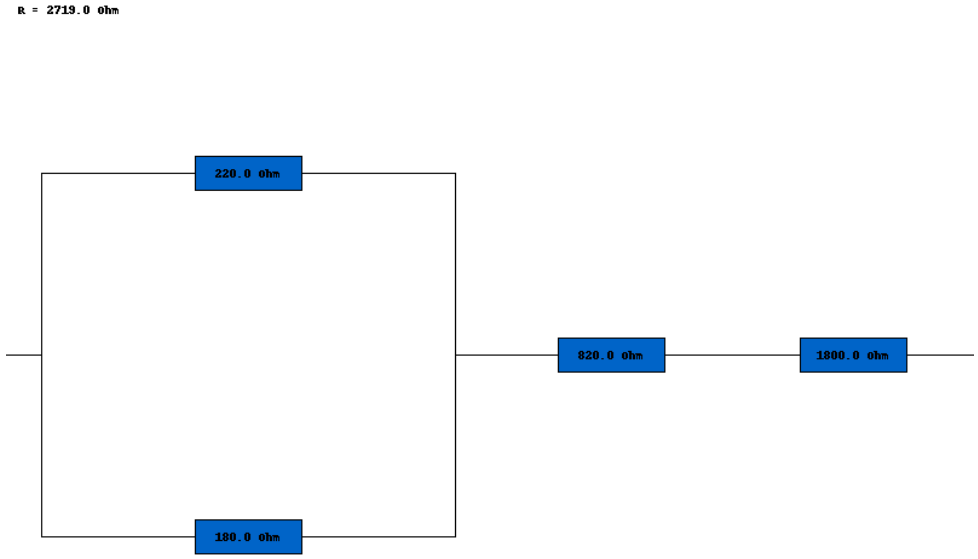


Bild für $k = 4$:



3.7 Beispiel 7 (R=4242)

```
k=1: 3900.0
k=2:  4230.0000: seq(3900.0, 330.0)
k=3:  4240.7767: seq(3900.0, par(390.0, 2700.0))
k=4:  4242.0000: seq(seq(270.0, 3900.0), par(120.0, 180.0))
```

Bild für $k = 1$:

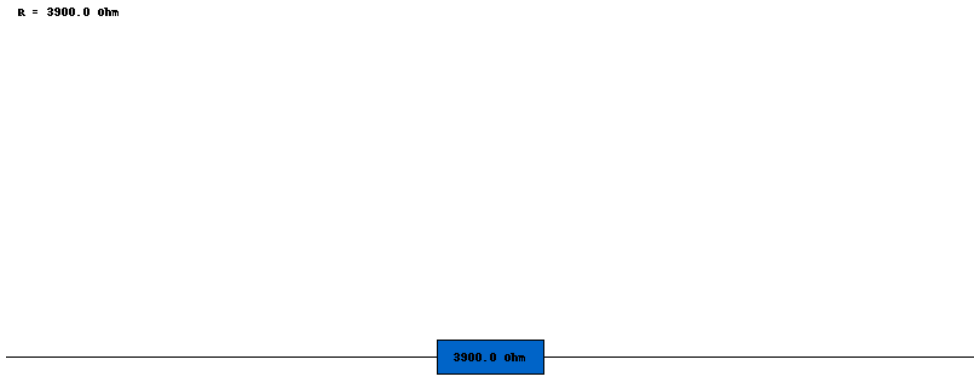


Bild für $k = 2$:

Bild für $k = 3$:

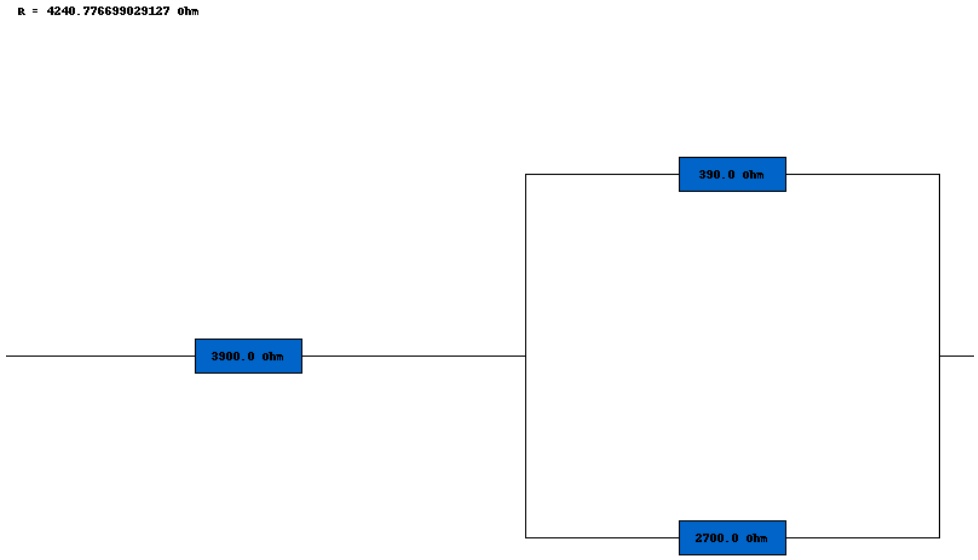
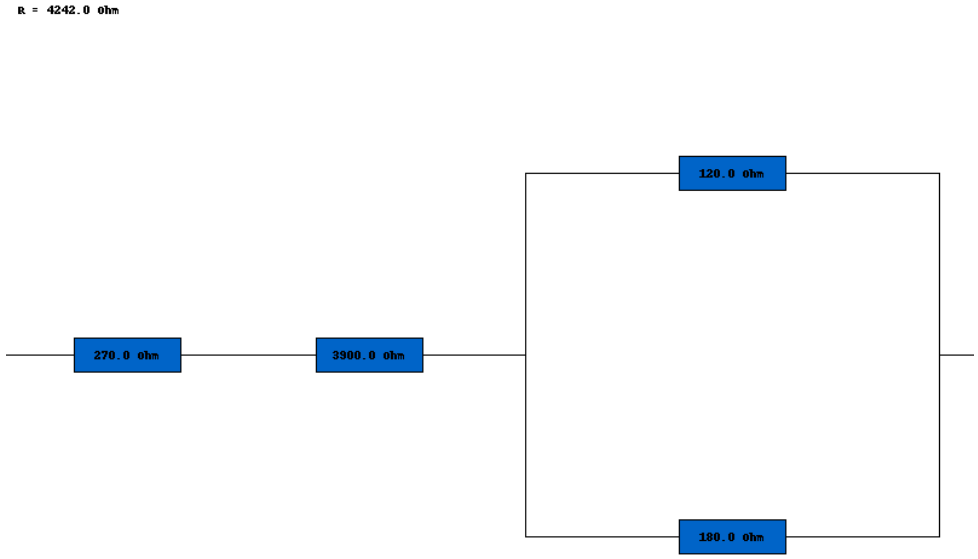


Bild für $k = 4$:



3.8 Beispiel 8 (R=1337)

```
k=1: 1200.0
k=2:  1330.0000: seq(330.0, 1000.0)
k=3:  1336.7961: seq(680.0, par(3300.0, 820.0))
k=4:  1336.9985: par(2200.0, seq(3300.0, par(390.0, 150.0)))
```

Bild für $k = 1$:

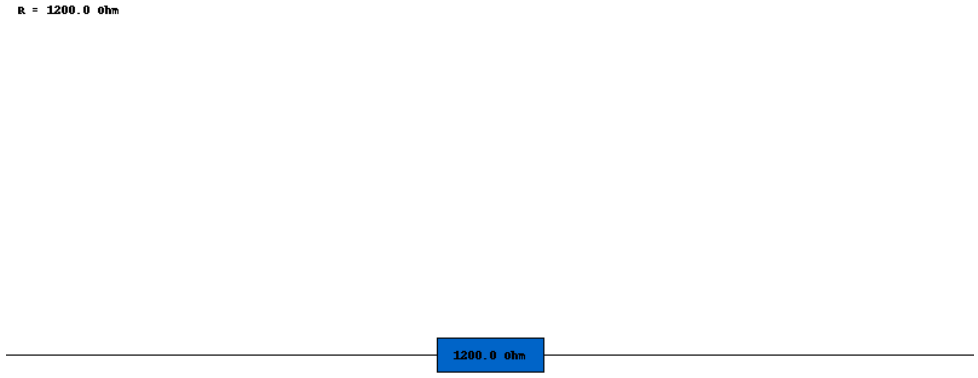


Bild für $k = 2$:

Bild für $k = 3$:

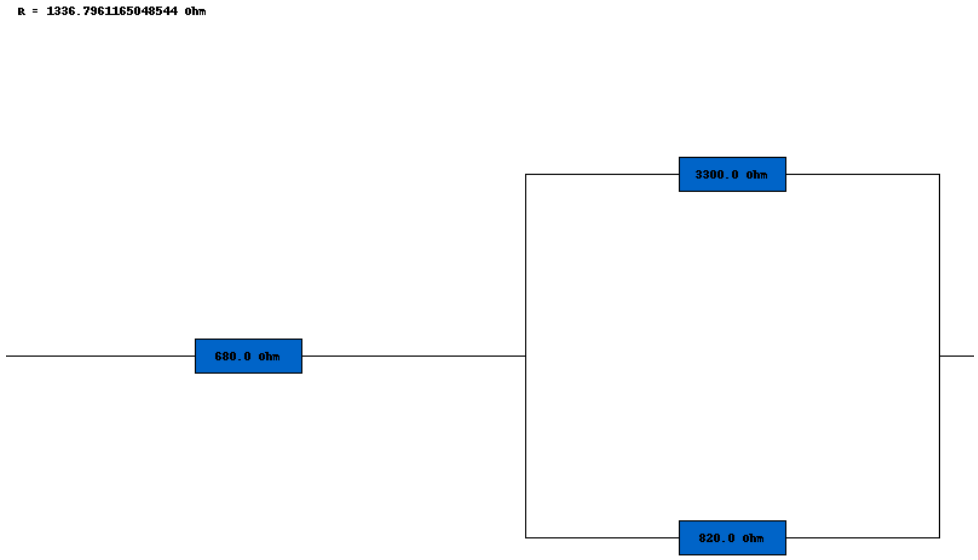
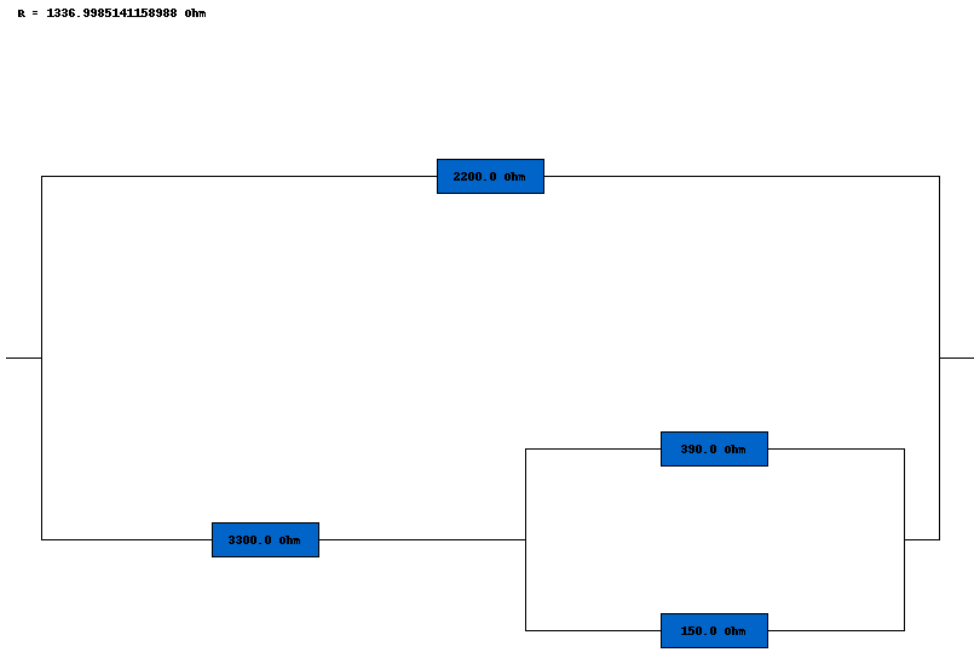


Bild für $k = 4$:



3.9 Beispiel 9 (R=2018)

```
k=1: 2200.0
k=2:  2020.0000: seq(820.0, 1200.0)
k=3:  2018.1818: seq(1200.0, par(1800.0, 1500.0))
k=4:  2018.0091: seq(1500.0, par(820.0, par(3900.0, 2200.0)))
```

Bild für $k = 1$:

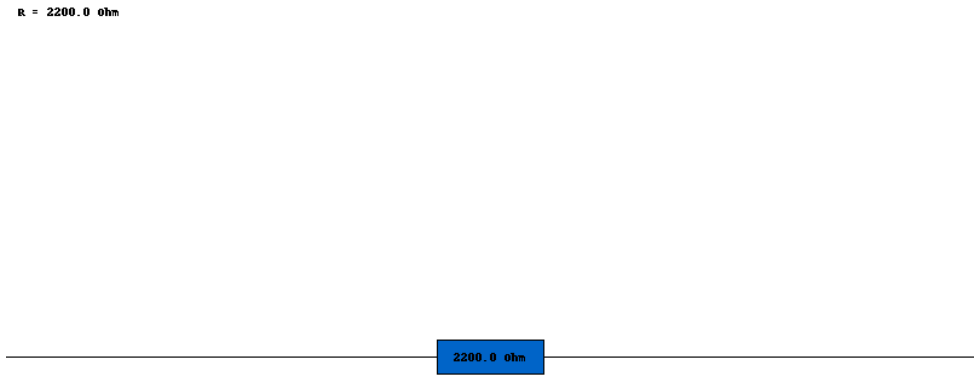


Bild für $k = 2$:

Bild für $k = 3$:

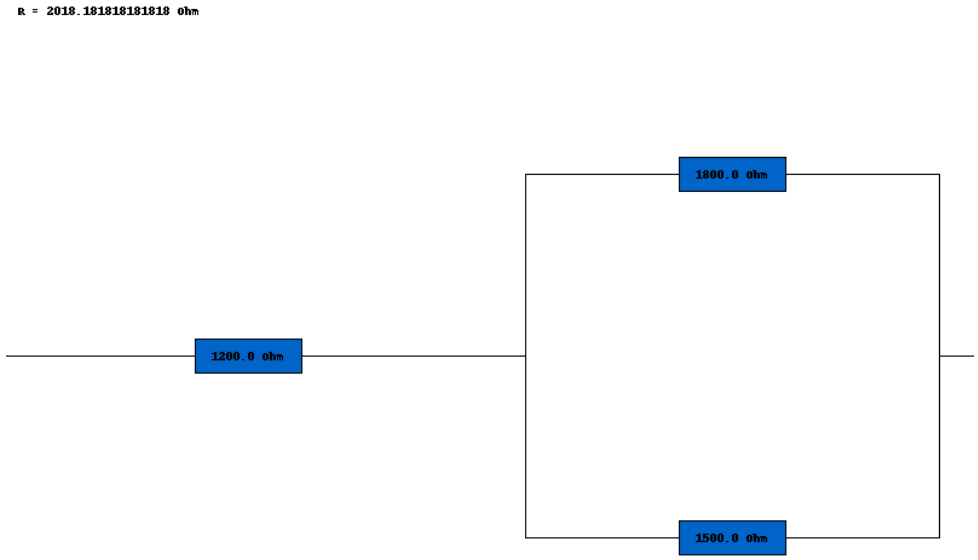
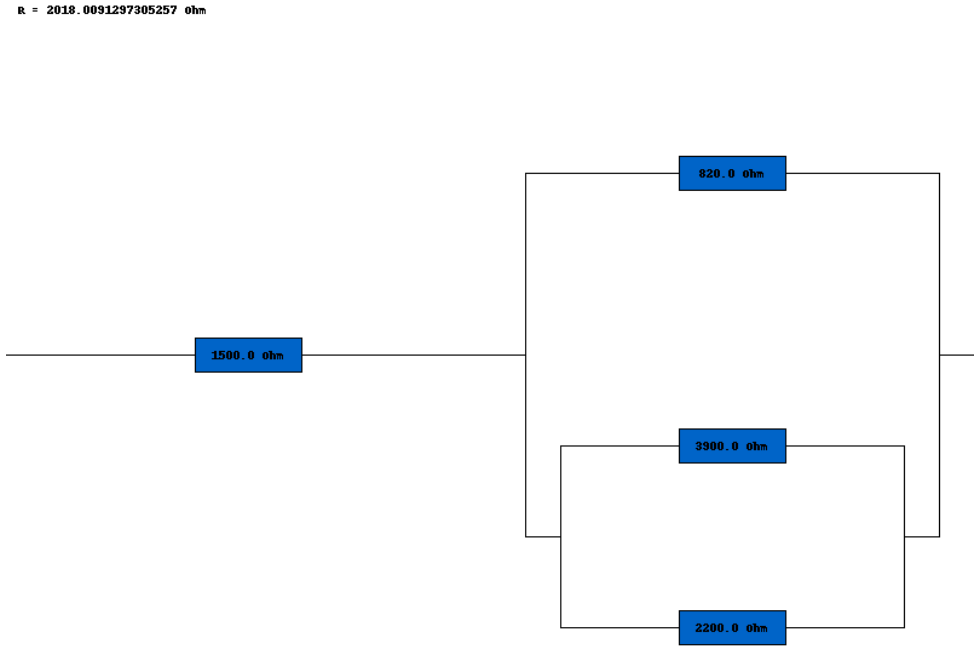


Bild für $k = 4$:



4 Quellcode

Ich habe fast alle Kommentare entfernt und der Code folgt nicht mehr dem pep8-Standard, da die vielen Lücken nur Platzverschwendung wären.

4.1 Imports

```
from PIL import Image, ImageDraw, ImageFont
from collections import Counter
import sys
from pathlib import Path
```

4.2 Widerstandsklassen

```
class Resistor():
  def __init__(self, value):
    self.value = value
    self.parts = [value, ]
  def __str__(self):
    return str(self.value)
  def get_structure(self):
    return str(self.value)
  def get_value(self):
    return self.value
  def draw(self, dctx, fnt, x1, y1, x2, y2, xr, yr):
    center = ((x1 + x2) // 2, (y1 + y2) // 2)
    dctx.line((x1, center[1], center[0]-xr//2, center[1]), fill=(0, 0, 0, 255))
    dctx.line((center[0]+xr//2, center[1], x2, center[1]), fill=(0, 0, 0, 255))
    dctx.rectangle((center[0]-xr//2, center[1]-yr//2, center[0] +
      xr//2, center[1]+yr//2), outline=(0, 0, 0, 255), fill=(0, 100, 200, 255))
    t = str(self.get_value()) + ' Ohm'
    ts = dctx.textsize(t)
    dctx.text((center[0]-ts[0]//2, center[1]-ts[1]//2), t, fill=(0, 0, 0, 255))

class ParallelResistor():
  def __init__(self, part1, part2):
    self.part1 = part1
    self.part2 = part2
    self.parts = self.part1.parts + self.part2.parts
  def get_value(self):
    return (self.part1.get_value()
            * self.part2.get_value()) \
        / (self.part1.get_value()
            + self.part2.get_value())
  def draw(self, dctx, fnt, x1, y1, x2, y2, xr, yr):
    margin = xr // 3
    lc = (x1, (y1 + y2) // 2)          # center, left
    lm = (x1+margin, lc[1])            # center of margin, left
    llc = (lm[0], lm[1]+(y2-y1)//4)    # lower corner, left
    luc = (lm[0], lm[1]-(y2-y1)//4)    # upper corner, left
    rc = (x2, lc[1])                   # center, right
    rm = (x2-margin, rc[1])            # center of margin, right
    rlc = (rm[0], rc[1]+(y2-y1)//4)    # lower corner, right
    ruc = (rm[0], rc[1]-(y2-y1)//4)    # upper corner, right
    dctx.line((lc, lm), fill=(0, 0, 0, 255))
    dctx.line((rc, rm), fill=(0, 0, 0, 255))
    dctx.line((llc, luc), fill=(0, 0, 0, 255))
    dctx.line((rlc, ruc), fill=(0, 0, 0, 255))
    self.part1.draw(dctx, fnt, x1+margin,
                    y1, x2-margin, (y1+y2)//2, xr, yr)
    self.part2.draw(dctx, fnt, x1+margin,
                    (y1+y2)//2, x2-margin, y2, xr, yr)
  def get_structure(self):
    return f'par({self.part1.get_structure()}, {self.part2.get_structure()})'
  def __str__(self):
    return f'{self.get_value():>10.4f}: {self.get_structure()}'

class SequentialResistor():
```

```
  def __init__(self, part1, part2):
    self.set_parts(part1, part2)
    self.parts = self.part1.parts + self.part2.parts
  def set_parts(self, p, q):
    if p.get_value() > q.get_value():
      self.set_parts(q, p)
    self.part1 = p
    self.part2 = q
  def draw(self, dctx, fnt, x1, y1, x2, y2, xr, yr):
    self.part1.draw(dctx, fnt, x1, y1, (x1+x2)//2, y2, xr, yr)
    self.part2.draw(dctx, fnt, (x1+x2)//2, y1, x2, y2, xr, yr)
  def get_value(self):
    return self.part1.get_value() + self.part2.get_value()
  def get_structure(self):
    return f'seq({self.part1.get_structure()}, {self.part2.get_structure()})'
  def __str__(self):
    return f'{self.get_value():>10.4f}: {self.get_structure()}'
```

4.3 Finder-Klasse

```
class ResistorFinder:
  def __init__(self, max_parts, parts, img_path=str(Path.home())):
    # constant to define how many resistors can be combined
    self.k = max_parts
    self.img_path = img_path
    # initiates the base parts
    self.set_base_parts(parts)
    # calculates all the combined parts
    self.set_combined_parts(self.k)
    self.combined_parts = [sorted(self.combined_parts[i], key=lambda x: x.get_value(
    )) for i in range(len(self.combined_parts))]
  def set_base_parts(self, parts): #[...]
  def get(self, value, max_k=4): #[...]
  def set_combined_parts(self, k): #[...]
  def remove_duplicates(self, resistors): #[...]
  def base_parts_available(self, resistor):
    c = Counter(resistor.parts)
    for key in c.keys():
        if c[key] > self.base_parts.get(key, 0):
            return False
    return True
```

4.4 Finden aller Widerstände

```
def set_base_parts(self, parts):
  self.base_parts = Counter()
  self.combined_parts = [[], ]
  for p in parts:
    self.base_parts[p] = self.base_parts.get(p, 0) + 1
    self.combined_parts[0].append(Resistor(p))

def set_combined_parts(self, k):
  cb = self.combined_parts
  if k < 2:
    return
  if len(cb) < k - 1:
    self.set_combined_parts(k - 1)
  print(f'Finding for k={k}')
  candidates = []
  count = 0
  max_count = sum([len(cb[i-1]) * len(cb[k-i-1]) for i in range(1, k//2+1)])
  for l in range(1, k // 2 + 1):
    sub_candidates = []
    for p1 in cb[l-1]:
      for p2 in cb[k-l-1]:
```

```
        par = ParallelResistor(p1, p2)
        if self.base_parts_available(par):
          sub_candidates.append(par)
        seq = SequentialResistor(p1, p2)
        if self.base_parts_available(seq):
          sub_candidates.append(seq)
        count += 1
        print(f'{count/max_count*100:.1f} %', end='\r')
    self.remove_duplicates(sub_candidates)
    candidates.extend(sub_candidates)
  self.remove_duplicates(candidates)
  cb.append(candidates)
  print(f'Found all for k={k}')
```

4.5 Finden und Zurückgeben des besten Widerstandes

```
def get(self, value, max_k=4):
  best_by_k = []
  parts = []
  for i, p in enumerate(self.combined_parts):
    parts.extend(p)
    parts = sorted(parts, key=lambda x: x.get_value())
    a, b = 0, len(parts)-1
    while a + 1 != b:
      c = (a + b) // 2
      c_value = parts[c].get_value()
      if c_value > value:
        b = c
      elif c_value < value:
        a = c
      else:
        break
    if a + 1 == b:
      if abs(value-parts[a].get_value()) < abs(value-parts[b].get_value()):
        best_by_k.append(parts[a])
      else:
        best_by_k.append(parts[b])
    else:
      for j in range(i, len(self.combined_parts)):
        best_by_k.append(parts[c])
      break
  for i, r in enumerate(best_by_k):
    draw_resistor(r, f'output/r{value}-k{i+1}')
  with open(f'output/r{value}.txt', 'w') as f:
    for i, r in enumerate(best_by_k):
      f.write(f'k={i+1}: {r}\n')
  return best_by_k
```

4.6 Hauptfunktion

```
def widerstand():
  path = 'bwinf37-runde1/a5-Widerstand/beispieldaten/test'
  if len(sys.argv) > 1:
    path = sys.argv[1]
  with open(path) as f:
    parts = [float(x) for x in f.readlines()]
  print(parts)
  rf = ResistorFinder(4, parts)
  while True:
    v = float(input('R: '))
    print('----------------')
    options = rf.get(v)
    for i, r in enumerate(options):
      print(f'k={i+1}: {r}')
    print('----------------')
```

```
17 if __name__ == '__main__':
       widerstand()
```