

Aufgabe 5: Widerstand

Team-ID: 00133

Team-Name: CubeFlo

Bearbeiter dieser Aufgabe:
Florian Rädiker (15 Jahre)

19. Oktober 2018

Inhalt

Lösungsidee	2
Kombinationen für Widerstände	2
Umsetzung.....	4
Finden der besten Kombination.....	4
Aufbau der Diagramme	5
Grafische Darstellung als SVG	5
Beispiele	5
Beispiele der BwInf-Seite	6
Quellcode	7
Bedeutung der Skripte	7
class ResistanceFinder	7
def __init__(resistances).....	7
find(r, k).....	9
print_results(r)	9
find_best(r).....	10
Das Hauptprogramm	10

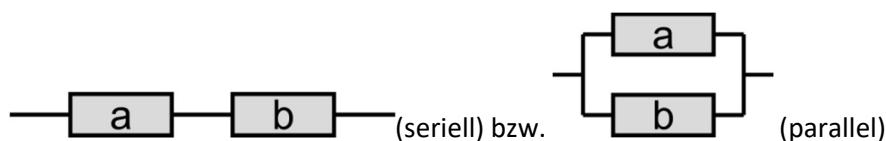
Lösungsidee

Die Aufgabenstellung legt nahe, dass alle möglichen Kombinationen für bis zu 4 Widerständen zuerst gefunden und ihr jeweiliger Wert ausgerechnet werden soll, um dann die bestmögliche (und nicht nur annähernde) Kombination für einen Widerstandswert zu finden. Eine Kombinationsmöglichkeit, selbst wenn es durch Umstellung gleichbedeutende Kombinationen gibt, reicht dabei aus, das heißt, dass es zum Beispiel für zwei seriell geschaltete Widerstände mit den Werten R_1 und R_2 in dieser Reihenfolge nicht auch noch die andere Reihenfolge berücksichtigt wird. Dies entsteht dadurch, dass beim Finden aller möglichen Widerstandswerte mit den Kombinationen nacheinander für $k = 1, \dots, 4$ immer alle möglichen Kombinationen aus der Widerstandsliste genommen und in alle für dieses k existierenden Kombinationsmöglichkeiten von Widerständen eingefügt werden. Bei „allen möglichen Kombinationen aus der Widerstandsliste“ handelt es sich um alle Permutationen für k Widerstandswerte mit Wiederholung, weshalb keine gleichbedeutenden Widerstandskombinationen nötig sind.

Kombinationen für Widerstände

Die Kombinationen für $k = 1$ müssen nicht weiter besprochen werden, da hier die Schaltungen nur aus einem Widerstand bestehen.

Für $k = 2$ können die Widerstände auf 2 verschiedene Arten kombiniert werden, nämlich seriell oder parallel:

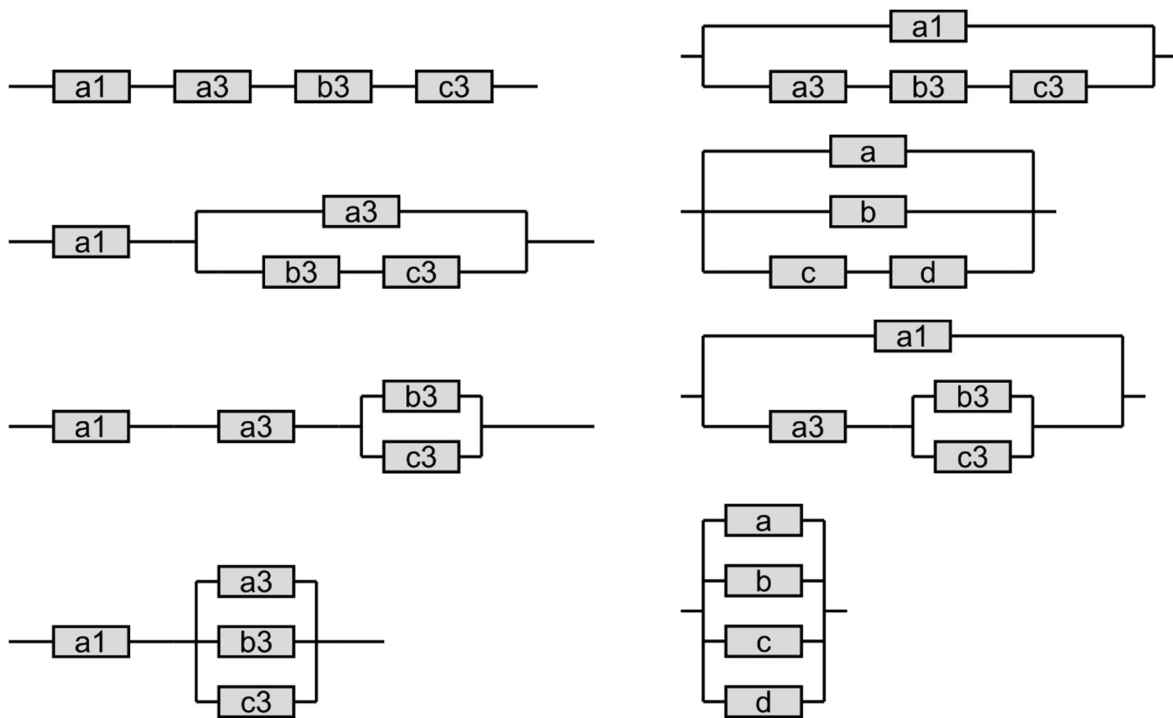


3 Widerstände ergeben sich aus der Kombination von einem Widerstand mit zwei Widerständen, und zwar entweder parallel oder seriell. Da es für einen Widerstand eine Möglichkeit zur Anordnung gibt und für 2 Widerstände 2, gibt es 2 Möglichkeiten, diese zwei Kombinationen zu kombinieren. Da das aber entweder seriell oder parallel geschehen kann, kommt man auf 4 Kombinationen. Dabei ist zu beachten, dass die Reihenschaltung von seriellen Widerständen nur wieder eine Reihenschaltung ergibt, wie eine Parallelschaltung von Parallelschaltungen vereinfacht werden kann. Man erhält also die folgenden Kombinationen (jeweils „+“ für seriell und „|“ für parallel):

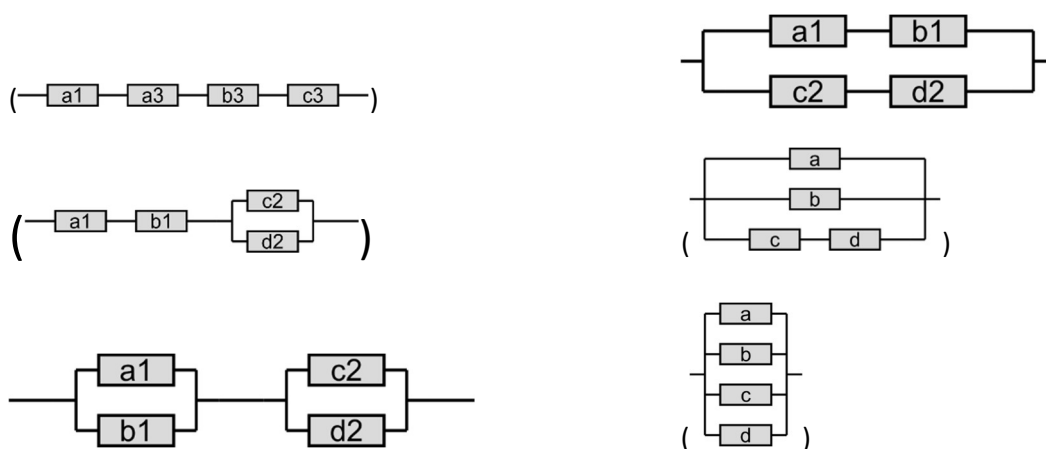
(k=1) + (k=2 seriell)	
(k=1) (k=2 seriell)	
(k=1) + (k=2 parallel)	
(k=1) (k=2 parallel)	<div style="display: inline-block; vertical-align: middle; text-align: center;"> vereinfacht sich zu </div>

Für Kombinationen mit 4 Widerständen gibt es die Möglichkeit, 1 Widerstand mit 3 Widerständen oder 2 Widerständen mit 2 Widerständen oder einfach nur 4 Widerstände zu kombinieren. 4 Widerstände seriell oder parallel erhält man aber auch allein durch Kombination von 1 Widerstand mit 3 Widerständen bzw. 2 Widerständen mit 2 Widerständen.

Zuerst die Kombinationen mit einem Widerstand seriell oder parallel mit einer Kombination von drei Widerständen. Für bessere Übersichtlichkeit sind jetzt immer die Schaltungen für eine Kombination in einer Zeile, die erste in einer Zeile ist seriell, die zweite parallel. Die Reihenfolge ist dieselbe wie die der Kombinationen mit 3 Widerständen. Die parallele Kombination in der 3. Zeile entspricht übrigens der in der Aufgabenstellung.



Für die Kombinationen aus 2 Widerständen gibt es 6 Möglichkeiten, 3 für beide seriell, beide parallel oder gemischt und das dann entweder seriell oder parallel:



Damit gibt es also 10 Möglichkeiten, 4 Widerstände zu kombinieren.

Umsetzung

Das Programm ist in Python 3 geschrieben.

Die Klasse `ResistanceFinder` des Moduls `Aufgabe5/resistancefinder.py` lädt beim Initialisieren alle möglichen Widerstandswerte für $k = 1, \dots, 4$, indem für die Liste von Widerstandswerten alle Permutationen durchgegangen werden. Für $k = 1$ reicht die Liste ja schon aus, während für $k = 2$ auch nur alle Kombinationen geladen werden müssen, weil bei beiden die Reihenfolge der Widerstände keine Rolle spielt, alle sind also beliebig vertauschbar. Dies ist auch für $k = 3$ und $k = 4$ bei allen Schaltungen, die ausschließlich seriell oder parallel geschaltet sind zutreffend, also entweder alle 3 bzw. 4 seriell bzw. parallel. Durch Permutation würde für diese Schaltungen, bei denen die Reihenfolge beliebig ist, der Wert ständig von neuem ausgerechnet. Deshalb werden diese Schaltungen separat über die Kombinationen berechnet.

Um durch alle Permutationen bzw. Kombinationen zu iterieren werden die Funktionen `permutations` und `combinations` des Moduls `itertools` verwendet. Beiden Funktionen werden als ersten Parameter ein iterierbares Objekt und als zweiten Parameter die Länge der Permutationen bzw. Kombinationen übergeben. Diese Längen sind also mit k identisch. Die Funktionen geben jeweils einen Generator zurück, der über alle Permutationen bzw. Kombinationen iteriert.

Für jedes k hat die Klasse ein `dict`, `self.resistors1`, ..., `self.resistors4`, das jeweils einen Widerstandswert einem zugehörigen Diagramm als String zuordnet. Das Diagramm benutzt dabei stets nur so viele Widerstände, wie im Attributnamen angegeben.

Durch die Generatoren wird mit `for`-Schleifen iteriert und jeweils die Permutation bzw. Kombination der Widerstände auf alle gefundenen Kombinationen für das jeweilige k angewandt und zum zugehörigen `dict` hinzugefügt. Die Kombinationen werden dabei nicht etwa durch Funktionen oder ähnlichem angewandt, sondern stehen direkt im Quellcode, wodurch unnötige Aufrufe erspart werden. Der Widerstandswert für eine bestimmte Kombination mit den aktuellen Widerstandswerten wird ausgerechnet und zum `dict` zusammen mit dem Diagramm hinzugefügt. Natürlich werden dadurch unter Umständen bereits gefundene Möglichkeiten, einen Widerstandswert zu erhalten, überschrieben. Es ist in der Aufgabe allerdings nicht gefordert, alle möglichen Schaltungen auszugeben, und um das Programm schneller arbeiten zu lassen (es handelt sich ja um sehr viele Möglichkeiten) wird nur die zuletzt gefundene Kombination gespeichert. Alle davor – also mit dem gleichen Schlüssel für das Dictionary – werden überschrieben.

Finden der besten Kombination

Das Finden der besten Kombination für ein festgelegtes k übernimmt die Methode `find` der `ResistanceFinder`-Klasse. Hier wird zuerst das zugehörige Dictionary für $k = 1, \dots, 4$ ausgewählt und danach der nächste Wert im `dict` mithilfe der Funktion `min()` gefunden. Sollte das Dictionary leer sein (es gibt also nicht genügend zur Verfügung stehende Widerstände, damit genug für dieses k zusammenkommen), wird `None` zurückgegeben. Die Funktion `min()` wird mit einem bestimmten `key` aufgerufen. Die als `key` übergebene Funktion gibt als Maß den Abstand zum gesuchten Widerstand zurück, sodass `min` die Kombination mit der kleinsten Differenz zum gesuchten Widerstand zurückgibt.

Ein kleines Extra bildet die Methode `find_best`, die die beste Kombination für ein beliebiges k zurückgibt. Da dies aber nicht direkt zur Aufgabe gehört, wird sie hier nicht weiter beschrieben.

Aufbau der Diagramme

Ein Diagramm kennzeichnet eine Reihenschaltung durch ein **S** mit nachstehenden Klammern:

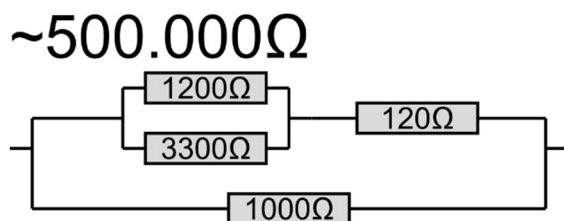
S(...+...). Alle seriell geschalteten Teile werden durch Pluszeichen voneinander getrennt.

Parallelschaltungen sind durch den Buchstaben **P** gekennzeichnet und die Bestandteile werden mit einem senkrechten Strich abgetrennt: **P(...|...)**. Als Kennzeichnung eines einzelnen Widerstands kann (muss aber nicht) das Zeichen **R(...)** mit einer Zahl (Wert in Ohm) zwischen den Klammern verwendet werden. Die Beispielschaltung der Aufgabe sieht also so aus:

P(S(P(1200 | 3300) + 120) | 1000) (Einheit ist Ohm und wird weggelassen).

Grafische Darstellung als SVG

Die grafische Darstellung ist zwar nicht explizit gefordert, aber trotzdem enthält diese Lösung das Modul `Aufgabe5/resistordiagram.py`, womit mithilfe des Drittanbietermoduls `svgwrite` SVG-Dateien aus den Diagrammen erstellt werden können. Das `svgwrite`-Modul liegt im Ordner dabei und wird von meinem Modul automatisch gefunden. Um den Umfang der Lösung begrenzt zu halten, wird das `resistordiagram`-Modul hier nicht weiter dokumentiert. Es sei aber gesagt, dass der Befehl `save` des Kommandozeilentools `Aufgabe5/aufgabe5-cmd.py` eine SVG-Grafik speichert. Als Demonstration wurden alle obigen Schaltungen mit diesem Modul erstellt, des Weiteren ist hier die Schaltung aus der Aufgabe:



Beispiele

Hier wird ein Beispiel mit lediglich drei Widerständen mit den Werten 100, 150 und 1000Ω dargestellt, um das Ganze möglichst einfach zu halten. Diese Beispielwerte befinden sich in der Datei `Aufgabe5/widerstaende-bsp.txt`. In der `__init__`-Funktion der `ResistanceFinder`-Klasse werden aus diesen 3 Werten durch die Kombinationen andere Widerstandswerte erstellt und gespeichert. Für $k = 1$ existiert das Ergebnis bereits, es muss nur umgewandelt werden. Es wird ein `dict` erstellt, das als Schlüssel alle Widerstandswerte und als Werte den Widerstandswert als String in das Diagramm **R(...)** eingesetzt. Man erhält also folgendes Dictionary:

```
{100: "R(100)", 150: "R(150)", 1000: "R(1000)"}
```

Für $k = 2$ gibt es schon doppelt so viele Kombinationen, jeweils einmal seriell und einmal parallel:

```
{250: "S(100+150)", 60.0: "P(100|150)", 1100: "S(100+1000)", 90.90909090909092: "P(100|1000)", 1150: "S(150+1000)", 130.43478260869566: "P(150|1000)"}
```

Und schließlich für $k = 3$, da es $k = 4$ mit drei Widerständen ja nicht geben kann, hier alle Kombinationen für 3 Widerstände:

```
{1250: 'S(100+150+1000)', 56.60: 'P(100|150|1000)', 92.0: 'P(100|S(1000+150))', 230.43: 'S(100+P(1000|150))', 132.0: 'P(150|S(1000+100))', 240.91: 'S(150+P(1000|100))', 200.0: 'P(1000|S(150+100))', 1060.0: 'S(1000+P(150|100))'}
```

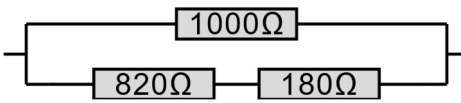
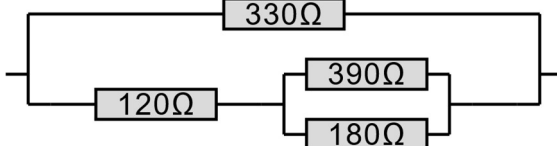
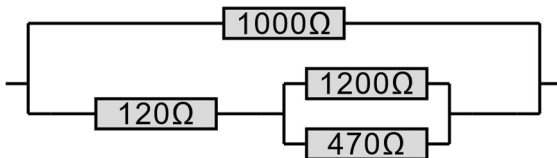
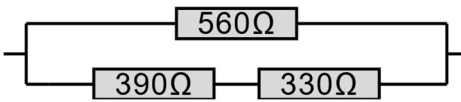

Aus Gründen der Übersichtlichkeit sind die Werte auf zwei Nachkommastellen gerundet.

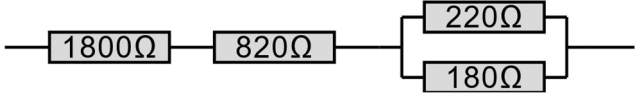
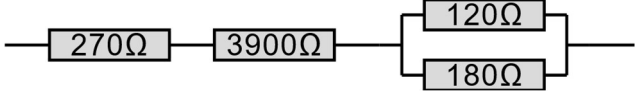
Hier ist auch gut zu sehen, dass es trotz 5 Kombinationsmöglichkeiten nicht so viele neue Werte gibt, sondern nur 11, weil es in der 2., 3. und 4. Kombination jeweils nur 3 Möglichkeiten gibt, da je zwei Werte beliebig vertauscht werden können. Natürlich werden diese Werte teilweise doppelt berechnet, aber alle Werte vorher herauszufiltern, würde länger dauern.

Nach dieser Initialisierung soll im Beispiel nun eine möglichst genaue Kombination für 500 mit $k = 2$ gefunden werden. Das Programm nimmt also `ResistorFinder.resistors2` (wegen $k = 2$) und sucht hier mithilfe von `min` nach dem kleinsten Abstand zu 500. Hier ist dies leider erst bei 250 mit dem Diagramm `S(100+150)` der Fall.

Beispiele der BwInf-Seite

Die folgende Tabelle listet alle benötigten Widerstandswerte mit der bestmöglichen Schaltung, die also am nächsten am benötigten Wert liegt, und die zugehörige Schaltung – erstellt mit dem `resistordiagramm`-Modul – auf. Die durch `resistancefinder.py` erzeugte Ausgabe enthält zusätzlich noch die Schaltungen für andere Werte von k , die hier aber aus Platzgründen nicht erscheinen.

Wert des benötigten Widerstands (in Ω)	k der besten Kombination	Schaltung
500	3	<p>$\sim 500.000\Omega$</p> 
140	4	<p>$\sim 140.000\Omega$</p> 
314	4	<p>$\sim 313.999\Omega$</p> 
315	3	<p>$\sim 315.000\Omega$</p> 
1620	2	<p>$\sim 1620.000\Omega$</p> 

2719	4	$\sim 2719.000\Omega$ 
4242	4	$\sim 4242.000\Omega$ 

Quellcode

Bedeutung der Skripte

```
python3 Aufgabe5/resistancefinder.py
```

Modul, das die Klasse `ResistanceFinder` enthält. Bei direktem Ausführen (also nicht importiert) wird die Widerstandsliste der Beispieldaten eingelesen und es wird jeweils die beste Annäherung für alle Werte für k ausgegeben, gefolgt von der Kombination, die die gesuchte Zahl am besten trifft.

```
python3 Aufgabe5/aufgabe5-cmd.py
```

Es handelt sich um ein Kommandozeilentool, das individuelle Widerstandswerte berechnet. Nach dem Start gibt es (nach optionaler Eingabe des Pfades zur Widerstandswertliste) hauptsächlich zwei Befehle: `find` und `save`. Beide sind über den `help`-Befehl dokumentiert (`help find` bzw. `help save` eingeben).

```
resistordialog.py
```

Modul, das Widerstände grafisch ausgibt. Dafür gibt es drei Klassen für drei unterschiedliche Typen von Widerstandsschaltungen: `Resistor`, für einzelne Widerstände, und `ParallelResistor` und `SerialResistor`, die beide eine Parallel- bzw. eine Reihenschaltung repräsentieren. In diese Klassen kann mit der Funktion `parse_diagram(diagram)` von einem Diagramm in diese Klassen umgewandelt werden. Jede der Klassen besitzt die Methode `.save`, die einen Dateipfad entgegennimmt und dort eine SVG-Grafik mit der Schaltung abspeichert.

Dieses Modul wird vom Befehl `save` aus `aufgabe5-cmd.py` benutzt, um den Bauplan auch grafisch auszugeben. Da dies jedoch nicht explizit zur Aufgabe in dieser Form gehört, wird die Funktionsweise hier nicht näher beschrieben.

class ResistanceFinder

```
def __init__(resistances)
```

Initialisiert den Finder mit der übergebenen Liste von Widerstandswerten `resistances`. Alle vier Dictionaries `self.resistors1`, ..., `self.resistors4` werden mit den jeweiligen Werten gefüllt (siehe „Umsetzung“). Es gibt dafür jeweils `for`-Schleifen, die im folgenden Code, der Teile der Funktion enthält (gekürzt), erklärt sind. Die einzelnen Berechnungen für unterschiedliche Kombinationen sind in derselben Reihenfolge wie unter „Lösungsidee“.

Zusätzlich wird am Ende im `dict self.resistors` jedem Wert für k das zugehörige `dict` zugeordnet, um diese später schneller finden zu können.

```

def __init__(self, resistances):
    # für k=1
    # Es gibt keine Kombinationsmöglichkeiten, nur die Diagramme werden für jeden
    # Widerstandswert zugeordnet
    self.resistors1 = {r: "R("+str(r)+")" for r in resistances}

    # für k=2
    # Für Kombinationen aus 2 Widerständen gibt es nur welche, die in der Reihenfolge
    # beliebig sind.
    # Deshalb werden nur Kombinationen, keine Permutationen verwendet.
    self.resistors2 = {}
    for r1, r2 in itertools.combinations(resistances, 2):
        # in str konvertieren, damit das später nicht mehrmals geschehen muss
        r1_ = str(r1)
        r2_ = str(r2)
        # seriell
        self.resistors2[r1 + r2] = "S("+r1_+" "+r2_+")"
        # parallel
        self.resistors2[1/(1/r1 + 1/r2)] = "P("+r1_+" | "+r2_+")"

    # für k=3
    self.resistors3 = {}
    # Kombinationen für alle, die in der Reihenfolge beliebig sind,
    # also 3 seriell oder parallel
    for r1, r2, r3 in itertools.combinations(resistances, 3):
        r1_ = str(r1)
        r2_ = str(r2)
        r3_ = str(r3)
        # seriell
        self.resistors3[r1 + r2 + r3] = "S("+r1_+" "+r2_+" "+r3_+")"
        # parallel
        self.resistors3[1/(1/r1 + 1/r2 + 1/r3)] = "P("+r1_+" | "+r2_+" | "+r3_+")"
    # Permutationen für alle, die in verschiedener Reihenfolge auftreten können
    for r1, r2, r3 in itertools.permutations(resistances, 3):
        r1_ = str(r1)
        r2_ = str(r2)
        r3_ = str(r3)
        # P( a | S(b+c) )
        self.resistors3[1/(1/r1 + 1/(r2+r3))] = "P("+r1_+" | S("+r2_+" "+r3_+")"
        # S( a + P(b|c) )
        self.resistors3[r1+1/(1/r2 + 1/r3)] = "S("+r1_+" + P("+r2_+" | "+r3_+")"

    # für k=4
    self.resistors4 = {}
    # Kombinationen für alle, die in der Reihenfolge beliebig sind,
    # also 4 seriell oder parallel
    for r1, r2, r3, r4 in itertools.combinations(resistances, 4):
        r1_ = str(r1)
        r2_ = str(r2)
        r3_ = str(r3)
        r4_ = str(r4)
        # seriell
        self.resistors4[r1+r2+r3+r4] = "S("+r1_+" "+r2_+" "+r3_+" "+r4_+")"
        # parallel
        self.resistors4[1/(1/r1+1/r2+1/r3+1/r4)] = "P("+r1_+" | "+r2_+" | "+r3_+" | "+r4_+")"
    # Permutationen für alle, die in verschiedener Reihenfolge auftreten können
    for r1, r2, r3, r4 in itertools.permutations(resistances, 4):
        r1_ = str(r1)
        r2_ = str(r2)
        r3_ = str(r3)
        r4_ = str(r4)

    # Kombinationen für 1 Widerstand mit 3 Widerständen

```



```

# (jeweils durch Leerzeichen im Diagramm getrennt)
# P(a | S(b+c+d))
self.resistors4[1/(1/r1 + 1/(r2+r3+r4))] = "P("+r1_+"|"+S("+r2_+"+"r3_+"+"r4_+"))"

# S(a + P(b|S(c+d)))
self.resistors4[r1+1/(1/r2 + 1/(r3+r4))] = "S("+r1_+"+P("+r2_+"|S("+r3_+"+"r4_+")))"
# P(a | P(b|S(c+d)))
self.resistors4[1/(1/r1 + 1/r2 + 1/(r3+r4))] = \
    "P("+r1_+"|"+r2_+"|S("+r3_+"+"r4_+"))"

# [...]

# Kombinationen aus Kombinationen von 2 Widerständen (ohne die bereits berechneten)
# P(S(a|b) | S(c|d))
self.resistors4[1/(1/(r1+r2)+1/(r3+r4))] = "P(S("+r1_+"+"r2_+"|S("+r3_+"+"r4_+"))"
# S(P(a|b) + P(c|d))
self.resistors4[1/(1/r1+1/r2)+1/(1/r3+1/r4)] = \
    "S(P("+r1_+"|"+r2_+"+P("+r3_+"|"+r4_+")))"

# Um die Dictionaries schnell finden zu können
self.resistors = {1: self.resistors1,
                  2: self.resistors2,
                  3: self.resistors3,
                  4: self.resistors4}

```

find(r, k)

Gibt die bestmögliche Kombination zum Widerstand r mit k verwendeten Widerständen zurück. Wenn die Zahl der Widerstände kleiner ist als k , das dict also leer ist, wird `None` zurückgegeben. Andernfalls wird ein Tupel bestehend aus dem Widerstand und dem zugehörigen Diagramm zurückgegeben.

Hier die Funktion mit beschriebener Funktionsweise:

```

def find(self, r, k):
    try:
        # Das zu k gehörende dict
        resistors = self.resistors[k]
    except KeyError:
        # k ist kein Schlüssel von dict, also ist k nicht richtig
        raise ValueError("Parameter 'k' must be 1, 2, 3 or 4, not "+str(k))

    if not resistors:
        # das dict ist leer, weil es nicht genügend Widerstandswerte für k gibt.
        return None
    # Das Schlüssel-Wert-Paar (als Tupel) mit der kleinsten Differenz zu r wird
    # zurückgegeben. Die lambda-Funktion gibt die Differenz zu r zurück.
    return min(resistors.items(), key=lambda item: abs(r-item[0]))

```

print_results(r)

Gibt alle Kombinationen und die bestmögliche Schaltung für den Widerstandswert r aus.

Mit `self.find` wird für jedes k eine Möglichkeit ausgegeben, sofern die Methode nicht `None` zurückgibt. Aus diesen Möglichkeiten wird wie in `self.find_best` die beste Schaltung gefunden und ebenfalls ausgegeben.

```

def print_results(self, r):
    results = [] # alle möglichen Schaltungen mit beliebigem k
    for k in range(1, 5):
        res = self.find(r, k)

```

```

    if res is not None:
        # die Schaltung muss möglich sein
        # Liste enthält Tupel aus: (k, Widerstandswert (float), Diagramm (str))
        results.append((k, res[0], res[1]))
# alle möglichen Schaltungen ausgeben
for k, resistance, diagram in results:
    print("k={k} {r:.4f}Ω Diagramm: {d}".format(k=k, r=resistance, d=diagram))
# die Schaltung mit der kleinsten Differenz zu r auswählen und ausgeben
best_k, best_r, best_diagram = min(results, key=lambda res: abs(r-res[1]))
print("##\nBESTE KOMBINATION: k={k} {r:.4f}Ω (Abweichung ~{diff:.2%}) Diagramm:"
      "{d}".format(k=best_k, r=best_r, diff=abs(best_r-r)/r, d=best_diagram))

```

find_best(r)

Gehört nicht direkt zur Aufgabe. Wie in `self.find` wird die Schaltung mit der kleinsten Differenz ausgewählt; es wird mit 4 Schaltungen von `self.find` mit den vier Werten von k gearbeitet.

Das Hauptprogramm

Zum Schluss in `resistancefinder.py` befindet sich das eigentliche Hauptprogramm. Durch die erste `if`-Bedingung wird es nur ausgeführt, wenn das Skript direkt ausgeführt, also nicht importiert wird. Die Datei `Aufgabe5/widerstaende.txt` wird eingelesen und jede ihrer Zeilen in einen Integer umgewandelt und das entstandene Tupel in der Variablen `resistances` gespeichert. Die Klasse `ResistanceFinder` wird mit diesen Widerstandswerten initialisiert und erstellt dabei auch alle Kombinationen. Es wird durch alle Widerstandswerte, die gefunden werden sollen (siehe BwInf-Webseite), iteriert und zu jedem wird mit `finder.print_results(r)` ein Ergebnis ausgegeben. Die unwichtigen `print`-Ausgaben sind hier heller dargestellt.

```

if __name__ == "__main__":
    with open("widerstaende.txt", "r") as f:
        resistances = tuple(int(r.strip()) for r in f.readlines())
    print("Lade Widerstände... (dauert bei mir weniger als 7 Sekunden)")
    t1 = time.perf_counter()
    finder = ResistanceFinder(resistances)
    t2 = time.perf_counter()
    print("Liste in", t2-t1, "geladen")
    print("Alle Widerstände auf 4 Nachkommastellen gerundet. "
          "Werte in Diagrammen in Ohm. ")

    for r in (500, 140, 314, 315, 1620, 2719, 4242):
        print("\n#####", r, "==", sep="\n")
        finder.print_results(r)

```