1 Bizz Buzz Woof (ca. 3+9+6=18 Punkte)

In dieser Aufgabe sollen Zahlen codiert durch Zahlenbereiche (z.B. "`50,30-35,104-102`") aus einer Datei eingelesen werden. Diese Zahlenbereich werden dann in eine Liste von Zahlen umgewandelt und unter gewissen Regeln (siehe unten) wieder in eine Datei zurück geschrieben. Eine typische Verwendung der Funktionen sieht wie folgt aus:

```
s = read_file('zahlenbereiche.txt')
l = num_list(s)
write_list('bizzbuzzwoof.txt', l)
```

Die drei Funktionen `read_file`, `num_list` und `write_file` müssen von ihnen geschrieben werden. Diese Teilaufgaben können unabhängig voneinander bearbeitet werden.

Schreiben Sie eine Funktion `read_file`, welche als Parameter den Dateinamen bekommt und den Inhalt der Datei als String zurück gibt.

```
def read_file(file_in):
  fp = open(file_in, 'r')
  l = fp.read()
  fp.close()
  return l
```

Schreiben Sie nun eine Funktion `num_list`, welche einen String der Form "`50,30-35,104-102`" als Parameter bekommt. Die Funktion soll dann eine Liste mit den entsprechenden Zahlen zurück geben. Im Beispiel also `[50,30,31,32,33,34,35,104,103,102]`. D.h. der Zahlenbereich von z.B. 30 bis 35 wird durch "30-35" codiert. Achten Sie darauf, dass auch Zahlenbereiche wie "104-102" vorkommen könnne, wo rückwärts gezählt werden muss. Außerdem können "Zahlenbereiche" auch nur aus einer Zahl bestehen. Zahlenbereiche werden durch Kommata getrennt. Sie können die in Python eingebaute `split` Funktion verwenden.

```
def num_list(s):
  ll = []
  l = s.split(',')
  for p in l:
     z = p.split('-')
     z = map(int, z)

     if(len(z) == 2):
        step = 1
        if(z[0] > z[1]):
           step = -1
        for i in range(z[0], z[1] + step, step):
           ll.append(i)
     else:
        ll.append(z[0])

  return ll
```

Nun sollen die Zahlen wieder in eine Datei geschrieben werden, jedoch mit folgenden Einschränkungen. Falls die Zahl ganzzahlig durch 3 teilbar ist, soll das Wort “bizz” ausgegeben werden, bei Teilbarkeit durch 5 das Wort “buzz” und bei Teilbarkeit durch 7 das Wort “woof”. Muss ein Wort statt einer Zahl ausgegeben werden, wird die Zahl selbst nicht mehr ausgegeben. Ist eine Zahl durch mehrere der Faktoren teilbar, müssen alle zugehörigen Worte ausgegeben werden. Für jede Zahl soll eine neue Zeile in der Datei begonnen werden. Es ergibt sich also fuer die Zahlen von 7 bis 15 die Datei

```
woof
8
bizz
buzz
11
bizz
13
woof
bizz buzz
```

Schreiben Sie nun die Funktion `write_file` welche eine Liste von Zahlen bekommt und diese entsprechend den genannten Regeln in eine Datei schreibt. Verwenden Sie den Modulo Operator `%`. Die Funktion soll den Dateinamen und die Liste mit Zahlen als Parameter erhalten.

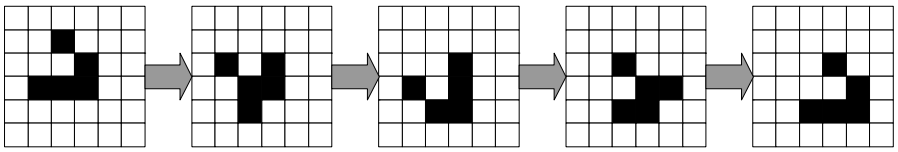
```
def write_file(file_out, ll):
    fp = open(file_out, "w")
    for i in ll:
        if(i % 3 == 0):
            fp.write('bizz ')
        if(i % 5 == 0):
            fp.write('buzz ')
        if(i % 7 == 0):
            fp.write('woof ')
        if(i % 3 != 0 and i % 5 != 0 and i % 7 != 0):
            fp.write(str(i) + ' ')
        fp.write('\ n')
    fp.close()
```

2 Game of Life (ca. 6+8=14 Punkte)

Das Game of Life ist ein zellulärer Automat, d.h. ein zweidimensionales Gebiet aus Zellen, mit dem eine primitive Form von Leben simuliert wird. Jede Zelle ist zu einen gegebenen Zeitpunkt tot (`False`) oder lebendig (`True`). Der Zustand im nächsten Zeitschritt hängt dann von der Nachbarschaft (acht umgebende Zellen) jeder Zelle (alter Zeitschritt!) ab:

- Ist eine Zelle tot und hat genau drei lebende Nachbarn, lebt sie im nächsten Schritt
- Ist die Zelle lebendig und hat zwei oder drei lebende Nachbarn, lebt sie weiter. Bei weniger Nachbarn stirbt sie an Einsamkeit, bei mehr Nachbarn an Überbevölkerung

Ein exemplarischer Ablauf für fünf Zeitschritte ist in der folgenden Abbildung dargestellt.



Der Konstruktor der Klasse `GameOfLife` ist im folgenden gegeben, der Parameter `feld` ist ein zweidimensionales numpy-Array mit Datentyp `bool`.

```
def __init__(self, feld):
  self.feld = copy(feld)
  (self.numx, self.numy) = shape(self.feld)
```

Schreiben Sie eine Methode `lebendeNachbarn`, die bei Aufruf mit zwei Parametern `ix` und `iy` die Anzahl an lebenden Nachbarnzellen der Zelle mit Index (ix, iy) zurückgibt. Es kann davon ausgegangen werden, dass die zugehörige Zelle keine Randzelle ist.

```
def lebendeNachbarn(self, ix,iy):
  count = 0
  for ni in range(i-1, i+2):
    for nj in range(j-1, j+2):
      if ((ni != i or nj != j) and
          self.feld[ni,nj] == True):
        count += 1
  return count
```

Schreiben Sie eine Methode `neueGeneration`, die für jede innere Zelle (keine Randzellen) des Arrays gemäß obiger Regeln den neuen Zustand (`True` für lebend bzw. `False` für tot) berechnet und das Ergebnis für sämtliche Zellen wieder in der Membervariablen(`self.feld`) speichert.

```
def neueGeneration(self):
  temp = zeros(self.feld.shape, dtype=bool)
  for i in range(1, self.numx-1):
    for j in range(1, self.numy-1):
      count = self.lebendeNachbarn(i,j)
      if (count == 3):
        temp[i,j] = True
      elif (self.feld[i,j] == True and count == 2):
        temp[i,j] = True
  self.feld = temp
```

3 Sortieren mit einem Binärbaum (max. 6+5+4=15 Punkte)

In dieser Aufgabe wird ein Binärbaum dazu verwendet, eine Liste von Zahlen zu sortieren. Es kann davon ausgegangen werden, dass die Liste nicht leer ist und keine doppelten Zahlen enthält. Ein `Knoten` des Binärbaums enthält jeweils den `wert` einer Zahl und mit `links` und `rechts` zwei Kindknoten. Falls ein Kindknoten nicht existiert, ist der Wert der zugehörigen Variable `False`. Der Konstruktor der Klasse `Knoten` ist gegeben:

```
def __init__(self, wert):
  self.wert = wert
  self.links = False
  self.rechts = False
```

Schreiben sie nun für die Klasse `Knoten` eine Methode `einfuegen`, die eine Zahl als Parameter bekommt und diese in Form eines neuen Knotens korrekt in den Baum einsortiert. Für den gesamten Binärbaum muss zu jeder Zeit gelten, dass linke Kinder eines Knotens kleinere und rechte Kinder eines Knotens größere Werte haben.

```
def einfuegen(self, wert):
  if wert < self.wert:
    if not self.links:
      self.links = Knoten(wert)
    else:
      self.links.einfuegen(wert)
  else:
    if not self.rechts:
      self.rechts = Knoten(wert)
    else:
      self.rechts.einfuegen(wert)
```

Zur sortierten Ausgabe der Werte im Binärbaum schreiben Sie für die Klasse `Knoten` eine Methode `sortiere`, die als Parameter eine leere Liste übergeben bekommt. Die Methode hängt an die Liste die Werte aller Knoten den Baumes (dessen Wurzelknoten der aktuelle Knoten ist) in sortierter Reihenfolge an. Dies kann durch einen In-Order-Durchlauf durch den Baum erreicht werden.

```
def sortiere(self, liste):
  if self.links:
    self.links.sortiere(liste)
  liste.append(self.wert)
  if self.rechts:
    self.rechts.sortiere(liste)
```

Die Klasse `Knoten` ist nun abgeschlossen, jetzt wird noch ein kurzes Programm benötig, das die Sortierung auf eine Liste anwendet. In der Variable `liste` sei eine unsortierte Liste gespeichert. Geben Sie ein Programm an, das mit Hilfe der Klasse `Knoten` den Inhalt der Variable `liste` sortiert und in der Variable `sortierteListe` abspeichert.

```
baum = Knoten(liste[0])
for i in liste[1:]:
  baum.einfuegen(i)
  print "inserted %d" % i

sortierteListe = []
baum.sortiere(sortierteListe)
```