Practicumopgaven

Algoritmen
en
Datastructuren.

Inhoudsopgave

Practicum Algoritmen en Datastructuren.

Richtlijnen:

- Geef bij de opgave aan, hoe de oplossing getest is. Beperk je niet tot één test.
  Test ook op grenswaarden en op waarden die niet zijn toegestaan
- Bouw de python-code op uit functies. Geef een beschrijving van de parameters en de return-waarde. Geef aan waar de parameters aan moeten voldoen en controleer dit m.b.v. de assert opdracht.
- Voorzie iedere functie van documentatie:

  """
  description

  Parameters
  ----
  parameter-naam1 : parameter-type1
  	description1
  parameter-naam2 : parameter-type2
  	description2

  Return
  ------
  return-naam: return-type
  	description

  """

Het is handig om niet meteen achter het beeldscherm te zitten en code in te kloppen. Denk eerst na. Maak een schets op papier.

Practicum week 1

Doel: oefenen met Python-concepten:

- lijsten
- strings
- random-functie

1. Schrijf in Python een functie

```
mymax(a)
```

De functie bepaalt het maximum van a , zonder gebruik te maken van de ingebouwde 'max'-functie.
De functie geeft een foutmelding als len(a) == 0 of als één van de elementen geen getal (dus niet van het type 'int' of 'double') is. Gebruik de assert-functie.

2. Schrijf een functie

getNumbers(s)

De functie maakt een lijst van de getallen, die in de string voorkomen

Voorbeeld:

a = 'een123zin45 6met-632meerdere+7777getallen'

getNumbers(a) geeft terug: **[123,45,6,632,7777]**

3. Een priemgetal is een getal groter dan 1 dat alleen deelbaar is door 1 en door zichzelf.

Voorbeelden van priemgetallen zijn 2,3,5,7,11

Maak een lijst van alle priemgetallen kleiner dan 1000 .

Gebruikt de "Zeef van Eratosthenes":

- ga uit van de lijst met de getallen 2 t/m 1000
- verwijder alle veelvouden van 2
- verwijder vervolgens alle veelvouden van 3
- etc

Zie https://nl.wikipedia.org/wiki/Zeef_van_Eratosthenes

4. Een klas bestaat uit 23 personen. Ga via een random-experiment na hoe groot de kans is, dat twee studenten op dezelfde dag jarig zijn.
Ga als volgt te werk.

- Genereer honderd keer een lijst van 23 random gehele getallen tussen de 1 en de 365.
- Bereken het aantal keer dat de lijst twee dezelfde getallen bevat.

Extra opdracht week 1:

Look-and-say sequence

De look-and-say sequences (de rij van Conway) is een rij van natuurlijke getallen die niet op een berekening gebaseerd zijn, maar op een taalkundige beschrijving. Elk volgende element van de rij is een "beschrijving" van het vorige element, vandaar dat de Engelse naam *Look-and-say sequence* is.

Een voorbeeld van de rij is:

1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, ….

Deze rij begint met 1, zodat het volgende element bepaald wordt door de beschrijving van 1:

- De beschrijving van 1 is "één 1" : 11

Vervolgens:

- De beschrijving van 11 is "twee 1-en" : 21
- De beschrijving van 21 is "één 2 en dan één 1" : 1211
- De beschrijving van 1211 is "één 1, dan één 2 en dan twee 1-en" : 111221
- De beschrijving van 111221 is “drie 1, dan twee 2 en dan één 1 : 312211

Een look-and-say sequence kan ook met een ander getal beginnen dan 1.

Implementeer de functie zodanig dat gegeven een element van de look-and-say squence opgeschreven als een integer lijst *x,* de functie het volgende element van de look-and-say sequence geeft (ook als integer lijst).

Bijvoorbeeld:

```
>>> x = [3,3,4,1,1,6,6,6,1]
>>> n = next_las_seq( x )
>>> print n output : [2,3,1,4,2,1,3,6,1,1]
```

Practicum week 2.

Doelen:

- Een O(n)-algoritme vervangen door een O(log(n))-algoritme.
- Een klasse in Python realiseren
- Een parsing-probleem oplossen
- Een performance-meting doen
- Een worst-case scenario afdwingen

1. In Python kun je machtsverheffen op verschillende manieren realiseren:

Methode 1: gebruik de operator '**'

```
def machtv1(a,n):
   return a**n
```

Methode 2: pas herhaald vermenigvuldigen toe:

```
def machtv2(a,n):
   assert n > 0
   m = 1
   for _ in range(0,n):
      m = m*a;
   return m
```

De methode is O(n).

Methode 3:

a^{16} kan berekend worden door achtereenvolgens a^2 , $(a^2)^2$, $((a^2)^2)^2$, $(((a^2)^2)^2)^2$ te berekenen.
Dit wordt 'herhaald kwadrateren' genoemd.

De berekening is nu eenvoudig omdat 16 een 2-macht is.

a^{11} kan als volgt berekend worden: $a^{11} = a * a^{10} = a * (a^2)^5 = a * a^2 * (a^2)^4 = a * a^2 * ((a^2)^2)^2$

De strategie voor de berekening van a^n is als volgt

- n is even: $a^n = (a^2)^{n//2}$ kwadrateer a en halveer n
- n is oneven: verlaag n met 1: $a^n = a*a^{n-1}$

Schrijf in Python een functie die hiervan gebruikt maakt.

In de functie wordt een lokale variabele 'm' gebruikt.
De variabele 'm' heeft beginwaarde 1.
In een while-opdracht wordt 'n' gehalveerd of 'n' wordt met 1 verminderd.
Verder blijft na iedere iteratiestap ' m * a^n ' steeds dezelfde waarde houden.
Na afloop heeft 'n' de waarde 0 .

De functie heeft de volgende opbouw:

```
def machtv3(a,n):
  assert n > 0

  m = 1
  while n > 0:
      if n%2 == 0:
          ...
          ...
      else:
          ...
          ...
    return m
```

De methode is O(log(n)).

Ga na hoe vaak wordt vermenigvuldigd bij n = 10000 .

2. Schrijf in Python de klasse 'mystack'.

'mystack' is een uitbreiding van 'list' en heeft de volgende methoden:

- push
- pop
- peek (raadpleeg zonder weghalen)
- isEmpty

3. Het haakjesprobleem.

We bekijken uitdrukkingen die opgebouwd uit haken :

<, >, [,], (,)

De volgende uitdrukkingen zijn toegestaan:

((<>)())
[(<>)]( )(( )( ))
((<>))

De volgende uitdrukkingen zijn niet toegestaan:

([ )]
(((<) >))

De regels zijn als volgt:

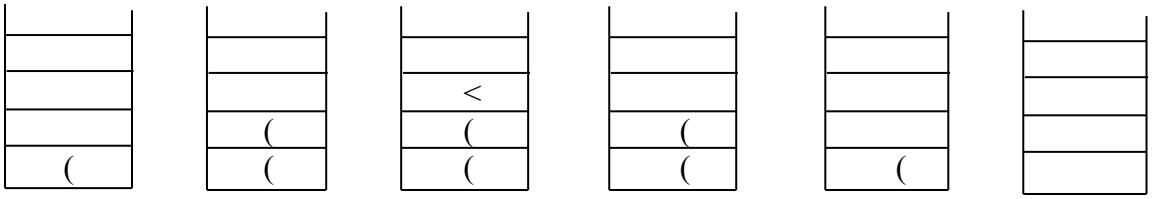
- bij ieder openingshaakje hoort een sluitingshaakje. De openingshaakje komt eerst.
- tussen een openingshaakje en het bijbehorende sluitingshaakje moet bij ieder openingshaakje ook het bijbehorende sluitingshaakje aanwezig zijn

Schrijf een programma, dat bij een gegeven string nagaat of het een geldige haakjesuitdrukking is. Maak gebruik van een stack. Hierbij wordt een openingshaakje op de stack geplaatst. Deze wordt weer verwijderd, zodra het corresponderende sluitingshaakje wordt gelezen.

Voorbeeld 1.

((<>))

De stack ziet er achtereenvolgens als volgt uit:

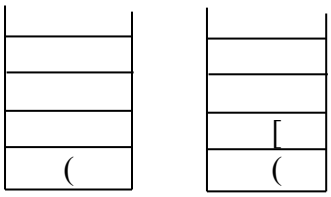


De expressie is gelezen en de stack is leeg. Dus de expressie is geldig.

Voorbeeld 2:

([ )]

De stack ziet er achtereenvolgens als volgt uit:



Nu gaat het fout want op de stack staat "[", terwijl het volgende haakje ")" is.

Maak in het programma gebruik van een stack die char-objecten bevat.

Test het programma uitgebreid.

4. In Python kan als volgt de binaire representatie van een getal worden achterhaald:

```
>>> bin(100)
'0b1100100'
```

Schrijf een recursieve functie

mybin(n)

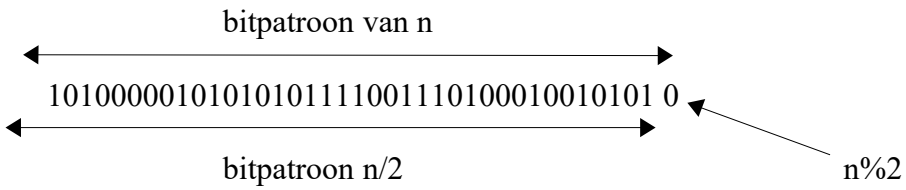
die hetzelfde doet. De functie geeft een string terug.

Eisen:

- invoer : een geheel getal
- uitvoer : een string
- assert: n >= 0

Ga na wat de basisgevallen van de functie zijn.

Baseer het algoritme op de volgende eigenschap:



Test het programma uitgebreid.

5. Zie reader blz. 35 (Quicksort)

Ga na hoe vaak twee elementen worden vergeleken bij een lijst met 10.000 elementen.

Wijzig Quicksort als volgt in een worst-case scenario: kies als 'pivot' steeds het kleinste element van de deelrij.

Hoe vaak worden twee elementen vergeleken bij dit scenario bij een lijst van 10000 elementen?

Practicum week 3.

Doelen:

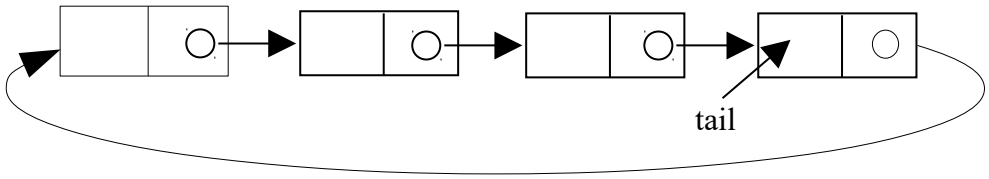
- werken met recursie.
- analyseren en aanpassen van een algoritme
- werken met 'linked list'
- wereken met boomstructuren

1. Zie reader blz. 36. (het koninginnenprobleem)
De functie 'research(N)' stopt zodra een oplossing is gevonden.

Wijzig de functie zodanig dat de functie alle oplossingen achterhaalt.
Beperk je bij de uitvoer tot de lijst-weergave.

2. Wijzig MyLinkedList in MyCircularLinkedList.
MyCircularLinkedList verschilt als volgt met MyLinkedList:

- MyCircularLinkedList bevat alleen een attribuut tail .
- tail.next verwijst naar de eerste node.



Onderscheid bij 'append' de volgende situaties:

- de lijst bevat geen node: er moet een node gemaakt worden, die naar zichzelf verwijst
- de lijst bevat minstens één node

Onderscheid bij 'delete' de volgende situaties:

- de lijst is leeg
- de lijst bevat één node en deze bevat de opgegeven waarde
- de lijst bevat meerdere nodes en alleen de laatste node bevat de opgegeven waarde
- de lijst bevat meerdere nodes en een node, die niet de laatste is, bevat de opgegeven waarde.

3. Breidt de Binary Search Tree uit met de volgende methoden:

- max(self) : bereken het maximale element
- rsearch(self,e) : zoek recursief e
- rinsert(self,e) : voeg recursief e toe
- showLevelOrder(self) : toon de boom level na level.
  Gebruik hierbij een queue. Plaats in de queue eerst de root.

Pas de klassen 'BST' en 'BSTNode' aan.

4. Maak een frequentietabel van de woorden uit een grote tekstfile.
Doe dit op twee manieren:

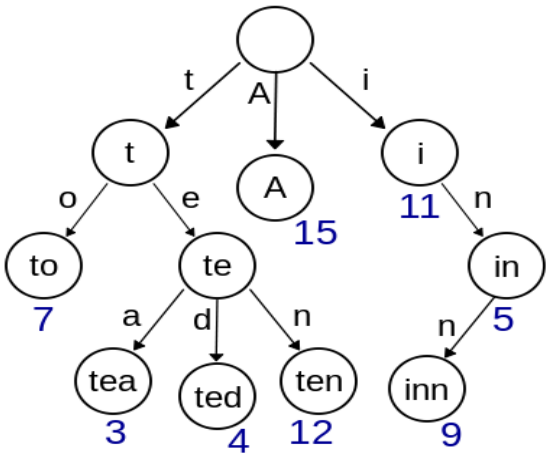
Methode 1

- schrijf een functie die de woorden in een dictionary plaatst. In de dictionary wordt bijgehouden hoe vaak een woord voorkomt. (key : woord, value : frequentie)
  De dictonary wordt teruggegeven.
- Schrijf een functie die op basis van de dictionary een frequentietabel naar een file (of een excelsheet) wegschrijft.

Methode 2

- schrijf een functie die de woorden in een Trie plaatst.

  (zie http://en.wikipedia.org/wiki/Trie)



A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn".

Een TrieNode correspondeert met een begin-gedeelte van een woord

De klasse TrieNode heeft de volgende attributen:
n : houdt bij hoe vaak een woord voorkomt
d : een dictionary met keys : gevonden karakters na het deelwoord
values : TrieNodes die corresponderen met deelwoord + key

De functie geeft de root-TrieNode terug

- Schrijf een functie die op basis van de root_TrieNode een frequentietabel naar een file (of een excelsheet) wegschrijft.

Ga met een programma na of de frequentietabellen overenstemmen.
Test het programma met grote tekstfiles.

Practicum week 4

Doelen:

- werken met 'hashing'.
- werken met 'dynamic programming'.

1. Implementeer "separate chaining hashing".

Maak een klasse met de volgende methoden:

- search(self,e)
- insert(self,e)
- delete(self,e)
- rehash(self,new_len)

Doe dit als volgt:

- Gebruik als tabel-elementen sets.
- De vullingsgraad (= aantal "elementen in alle sets" / lengte tabel) mag niet groter zijn dan 0.75. Is dit wel het geval dan moet m.b.v. rehash de tabel twee keer zo groot worden gemaakt.
- Maak een test, waarbij 200 random gebroken getallen worden toegevoegd en vervolgens 100 worden verwijderd.
- Toon na iedere rehash de tabel.

2. Bepaal twee verschillende gebroken getallen met dezelfde hashwaarde.

Voorbeeld:

```
hash(0.9137260267282177) == hash(0.9400579282662489) == -2008789643
```

Maak hiervoor gebruik van een dictionary. In de dictonary worden getal en bijbehorende hash-waarde opgeslagen.

Deze aanpak is gebaseerd op de verjaardagsparadox: de kans dat twee mensen uit een groep van 23 personen op dezelfde jarig zijn is groter dan ½ .

Voor het correct tonen van een gebroken getal moet de opdracht

```
print(repr(x))
```

i.p.v.

```
print(x)   # de laatste cijfers worden weggelaten
```

gegeven worden

3. Maak de functie B(n,k) die (n!//(k!))//(n-k)! berekent.

Doe dit op basis van dynamic programming.

Wat is B(100,50) ?

4. Bereken bij een gegeven geldbedrag (uitgedrukt in eurocenten) op hoeveel verschillende manieren het betaald kan worden.
Doe dit op basis van dynamic programming.

Maak een functie F(n) met

n: het bedrag uitgedrukt in eurocenten
return-waarde: het aantal manieren waarmee het geldbedrag betaald kan worden.

Beperk je tot n <= 10000 .

Voorbeeld:

n = 7

n = 1+1+1+1+1+1+1
= 1+1+1+1+1+2
= 1+1+1+2+2
= 1+2+2+2
= 1+1+5
= 2+5

Er kan dus op 6 manieren betaald worden

Je kunt de volgende strategie gebruiken.

m = [1,2,5,10,20,50,100,200,500,1000,2000,5000,10000]

In de matrix A is A[i,j] = het aantal manieren waarmee bedrag j betaald kan worden met de munten m[0] t/m m[i].

Er geldt: A[i,0] = 1 voor alle i
A[0,j] = 1 voor alle j
Als j >= m[i] : A[i,j] = A[i-1,j] + A[i,j-m[i]]
j < m[i] : A[i,j] = A[i-1,j]

m		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	**0**	1	1	1	1	1	1	1	1
2	**1**	1	1	2	2	3	3	4	4
5	**2**	1	1	2	2	3	4	5	6

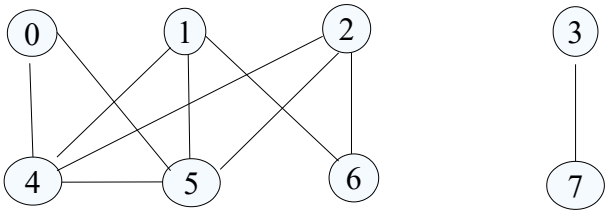
Practicum week 5

Doel:

- werken met grafen

1. Een niet-gerichte graaf is samenhangend (connected) als het uit één component bestaat.

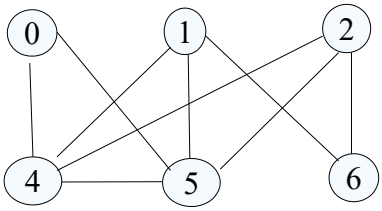
   De volgende graaf bestaat uit twee componenten:



   Schrijf de functie

```
is_connected(G)
```

   De functie gaat na of een niet-gerichte graaf G samenhangend is. De return-waarde is een boolean. Test de functie o.a. met bovenstaande graaf en met de graaf



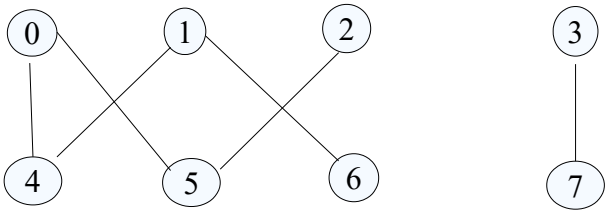
   Maak gebruik van het BFS-algoritme.

2. Bovenstaande graaf heeft cykels.

   Schrijf de functie

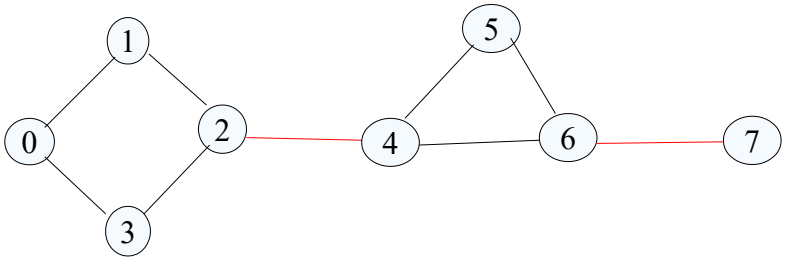
```
no_cycles(G)
```

   De functie gaat na of een niet-gerichte graaf G geen cykels bevat. De return-waarde is een boolean. Test de functie o.a. met bovenstaande graaf en met de graaf



   Maak gebruik van het BFS-algoritme.

3.



De edges (2,4) en (6,7) zijn 'bridges' . Een 'edge' is een 'bridge' als bij het verwijderen van de 'edge' het aantal componenten van de graaf toeneemt.

Schrijf de functie

```
get_bridges(G)
```

De functie geeft in een lijst alle bridges terug. Bij dit voorbeeld wordt

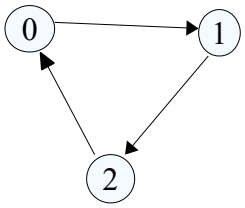
[(2,4) , (4,2) , (6,7) , (7,6)] teruggegeven.

Om na te gaan of de edge (u,v) een 'bridge' is , kun je als volgt te werk gaan:

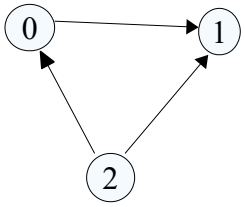
- verwijderde de edge (u,v)
- ga m.b.v. het BFS-algoritme na of 'v' nog verbonden is met 'u'.
- voeg (u,v) toe aan de lijst van bridges als 'v' niet verbonden is met 'u'
- voeg de edge (u,v) weer toe

4. Een gerichte graaf is volledig verbonden ('strongly connected') als iedere node bereikbaar is vanaf iedere andere node.

   Een voorbeeld van een 'strongly connected' graaf:



De volgende graaf is niet 'strongly connected' :



Schrijf de functie

```
is_strongly_connected(G)
```

De functie geeft een boolean-waarde terug.

Gebruik daarbij de volgende strategie:

- kies een startnode
- ga met het BFS-algoritme na of alle nodes bereikbaar zijn vanuit de startnode.
- maak een nieuwe graaf, waarbij alle pijlen van de oorspronkelijke graaf zijn omgekeerd
- ga weer met BFS-algoritme na of alle nodes bereikbaar zijn vanuit de startnode

5. Een niet-gerichte graaf is een Euler-graaf als alle nodes een even graad hebben.

(5a)

Schrijf de functie

```
is_Euler_graph(G)
```

De functie geeft een boolean-waarde terug.

(5b)

Een Euler-circuit is een wandeling door graaf, waarbij alle takken precies één keer doorlopen worden en het eindpunt gelijk is aan het beginpunt.
Bij een Euler-graaf is een Euler-circuit mogelijk.

Schrijf de functie

```
get_Euler_circuit(G,s)
```

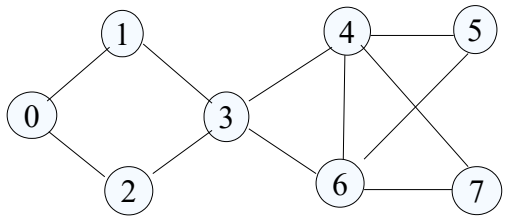
De functie geeft de lijst van de nodes, die achtereenvolgens tijdens de wandeling zijn bezocht, terug. De wandeling wordt begonnen en beëindigd bij 's'.

Gebruik daarbij de volgende strategie (afkomstig van Fleury):

- begin met een lege lijst
- kies een buur t, waarbij de edge (s,t) geen brug is (tenzij het niet anders kan) en voeg t aan de lijst
- verwijder de edge (s,t) (en ook (t,s)) uit de graaf
- kies vervolgens een buur u van node t, waarbij de edge (t,u) geen brug is (tenzij het niet anders kan) en voeg u aan de lijst
- verwijder de edge (t,u) (en ook (u,t)) uit de graaf
- herhaal dit proces totdat alle edges zijn verwijderd

Test de functie aan de hand van de volgende graaf:

Graaf G:



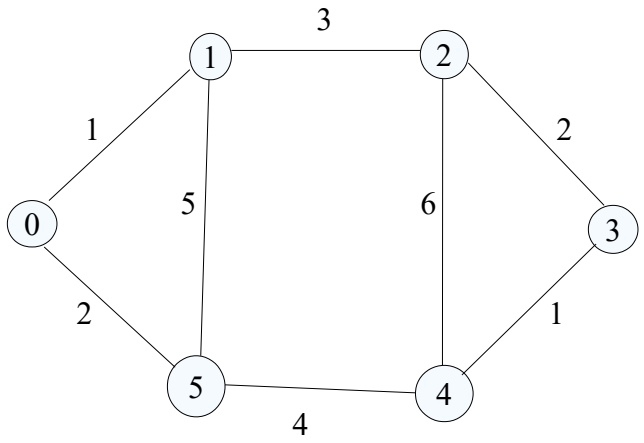
De functie kan de volgende lijst teruggeven: **[0,1,3,4,5,6,4,7,6,3,2,0]**

Probeer verschillende startpunten uit.

Extra practicum-opdracht.

Het Chinese postbode-probleem.

Ga uit van graaf G:



Dit is geen Euler-graaf (waarom niet?)

De algemene opdracht is: zoek een route met de laagste kosten.
De route moet alle takken doorlopen. Omdat de graaf geen Euler-graaf is worden sommige takken vaker doorlopen.

In Python wordt graaf G als volgt gedefinieerd:

```
v = [Vertex(i) for i in range(6)]

G = {v[0]:{v[1]:1,v[5]:2},
     v[1]:{v[0]:1,v[2]:3, v[5]:5},
     v[2]:{v[1]:3,v[3]:2, v[4]:6},
     v[3]:{v[2]:2,v[4]:1},
     v[4]:{v[2]:6,v[3]:1,v[5]:4},
     v[5]:{v[0]:2,v[1]:5,v[4]:4}}
```

Hoe wordt bepaald welke extra takken moeten worden toegevoegd?

De nodes 1, 2, 4 en 5 hebben een oneven graad. Het aantal nodes met een oneven graad is altijd even. (Waarom?)

Uit een even aantal nodes kan een lijst van paren gemaakt worden.
Bij dit voorbeeld zijn de mogelijke paar-lijsten:

{ (1,2), (4,5) }
{ (1,4), (2,5) }
{ (1,5), (2,4) }

Bij ieder paar kan (met het Dijkstra-algoritme) het kortste pad tussen de twee nodes bepaald worden.

paar	**korste pad**	**lengte**
(1,2)	1-2	3
(1,4)	1-2-3-4	6
(1,5)	1-0-5	3
(2,4)	2-3-4	3
(2,5)	2-1-0-5	6
(4,5)	4-5	4

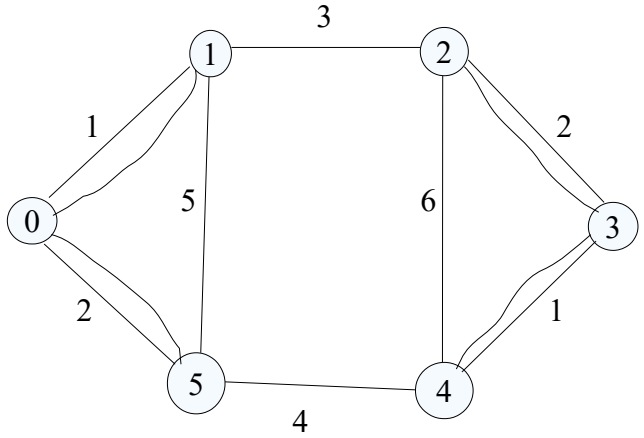
Per paar-lijst wordt de som van de lengten bepaald

paar-lijst	**lengte-som**
{ (1,2), (4,5) }	3+4 = 7
{ (1,4), (2,5) }	6+6 = 12
{ (1,5), (2,4) }	3+3 = 6

De paar-lijst { (1,5), (2,4) } heeft de kleinste lengte-som.
De kortste paden, die hier bij horen, zijn: 1-0-5 en 2-3-4 .
De takken (0,1), (0,5), (2,3) en (3,4) moeten dus extra toegevoegd worden.

De graaf wordt omgezet in een multi-graaf. Hierbij kunnen twee nodes verbonden zijn met meerdere takken.

Bij dit voorbeeld wordt de multigraaf G2:



Deze multi-graaf is wel een Euler-graaf. (waarom?)

De multi-graaf G2 wordt als volgt vastgelegd:

G2 = {v[0]:{v[1]:{'d':1,'n':2},v[5]:{'d':2,'n':2}},
v[1]:{v[0]:{'d':1,'n':2},v[2]:{'d':3,'n':1}, v[5]:{'d':5,'n':1}},
v[2]:{v[1]:{'d':3,'n':1},v[3]:{'d':2,'n':2}, v[4]:{'d':6,'n':1}},
v[3]:{v[2]:{'d':2,'n':2},v[4]:{'d':1,'n':2}},
v[4]:{v[2]:{'d':6,'n':1},v[3]:{'d':1,'n':2},v[5]:{'d':4,'n':1}},
v[5]:{v[0]:{'d':2,'n':2},v[1]:{'d':5,'n':1},v[4]:{'d':4,'n':1}}}

Hierbij 'd' het gewicht van de tak vast en 'n' het aantal takken tussen twee nodes.

Een korste pad dat start bij node '0' is: 0-1-2-3-2-4-3-4-5-1-0-5-0
Deze wordt met de 'get_Euler_circuit'-functie verkregen.
Hierbij worden de takken (0,1) , (2,3) , (3,4) en (0 ,5) twee keer doorlopen.

De concrete opdracht: maak de functie **get_shortest_postman_path(G,s)** .

De functie geeft de lijst van de nodes, die achtereenvolgens tijdens de wandeling zijn bezocht, terug. De wandeling wordt begonnen en beëindigd bij node 's'.

Pas dit toe op bovenstaande graaf.
Verzin zelf ook een interessante graaf.

Voor het verkrijgen van alle paar-combinaties kan onderstaande (recursieve) functie gebruikt worden.

```
def get_pair_lists(a):
    if len(a) == 0:
        return []
    if len(a) == 2:
        return [[(a[0],a[1])]]
    pair_lists = []
    for i in range(1,len(a)):
        pa1 = [(a[0],a[i])]
        pair_lists2 = get_pair_lists(a[1:i]+a[i+1:])
        for pa2 in pair_lists2:
            pair_lists.append(pa1+pa2)
    return pair_lists
```

De opdracht:

```
print(get_pair_lists([1,2,4,5]))
```

heeft als uitvoer:

```
[[(1, 2), (4, 5)], [(1, 4), (2, 5)], [(1, 5), (2, 4)]]
```

Websites met achtergrond-informatie:

http://nl.wikipedia.org/wiki/Chinees_postbodeprobleem en
https://www.youtube.com/watch?v=JCSmxUO0v3k