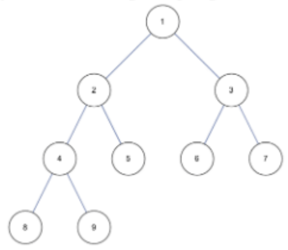
Lista 4 (Lab) Termin wysłania na SVN do 12.04.2020

Przy pisaniu programów (funkcji) **NALEŻY** stosować się do ogólnie przyjętego stylu programowania w języku Python. Proszę dokładnie przeczytać PEP 8 (tutorial, google python style) i PEP 257.

1. (5pt) Napisz dekorator, mający za zadanie
   - drukować informacje o czasie wykonywania funkcji
2. (5pt) Załóżmy, że mamy drzewo i reprezentujemy je na liście np. drzewo

   reprezentujemy jako

   ```
   ["1", ["2", ["4", ["8", None, None], ["9",None,None]], ["5",None,None]], ["3", ["6", None, None], ["7", None, None]]]
   ```

   Napisz funkcję która generuje w sposób losowy drzewo podanej wysokości oraz funkcję która wyświetla drzewo w porządku inorder.
3. (5pt) Napisz klasę **class Node(object)** do reprezentacji pojedynczego węzła drzewa z dowolną liczbą potomków. Podobnie jak w zadaniu poprzednim napisz funkcję która generuje losowo drzewo o danej wysokości i funkcje które przechodzą drzewo w porządku inorder.

4. (10pt) Przeciążenie funkcji (function overloading) daje możliwość wykorzystania tej samej nazwy funkcji, ale z różnymi parametrami. Na przykład w innych językach możemy napisać

   ```
   float norm(float x, float y) {           // norma Euklidesowa
     return sqrt(x*x + y*y)
   }
   float norm(float x, float y, float z) {   // norma taksówkowa
     return abs(x) + abs(y) + abs(z)
   }
   ```

   W języku Python nie ma przeciążenia funkcji, po prostu następna definicja nadpisuje poprzednią. Napisz dekorator nazwijmy go **@overload**, który pozwala na taką własność. Przykładowy program powinien wyglądać tak

   ```
   @overload
   def norm(x,y):
     return math.sqrt(x*x + y*y)

   @overload
   def norm(x,y,z):
     return abs(x) + abs(y) + abs(z)

   print(f"norm(2,4) = {norm(2,4)}")

   print(f"norm(2,3,4) = {norm(2,3,4)}")
   ```

   Otrzymujemy:

   ```
   norm(2,4) = 4.47213595499958
   norm(2,3,4) = 9
   ```

   Wskazówka: Napisz dekorator, który wraca klasę z odpowiednio przeciążonym operatorem **__call__**, która przechowuje nazwy funkcji z parametrami. Do odróżnienia funkcji można wykorzystać np. **getfullargspec(f).args** z modułu **inspect** (from inspect import getfullargspec).

5. (15pt)* Mnożenie dużych liczb o n cyfrach można wykonać w $O(n \log n)$ zamiast klasycznie $O(n^2)$, dzięki szybkiej transformacie Fouriera. Napisz klasę **FastBigNum** do obliczania iloczynu dwóch bardzo dużych liczb. W programie zaimplementuj szybką transformatę Fouriera (FFT) oraz w klasie FastBigNum zdefiniuj **__mul__** oraz **__str__**. Mówimy, że wektor $y = (y_0, y_1, \ldots, y_{n-1})$ jest **dyskretną transformatą Fouriera** (DFT) wektora $x = (x_0, x_1, \ldots, x_{n-1})$ i piszemy $y = DFT(x)$, jeśli

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{j \cdot k}$$

dla $k = 0, \ldots, n-1$ oraz $\omega_n = e^{-2\pi i/n}$. Podobnie definiujemy odwrotną dyskretną transformatę Fouriera (DFT^{-1}) i piszemy $x = DFT^{-1}(y)$, jeśli

$$x_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-k \cdot j}$$

dla $j = 0, \ldots, n-1$. Niech $n \in \mathbb{N}$ (w przypadku FFT n jest potęgą dwójki) oraz X i Y będą dużymi liczbami całkowitymi takimi, że:

$$X = \sum_{j=0}^{n-1} x_j 10^j, \quad Y = \sum_{j=0}^{n-1} y_j 10^j$$

Algorytm mnożenia liczb $Z = X \cdot Y$:

- $X^* = (x_0^*, x_1^*, \ldots, x_{2n-1}^*) = DFT_{2n}(x_0, x_1, \ldots, x_{n-1}, 0, \ldots, 0)$
- $Y^* = (y_0^*, y_1^*, \ldots, y_{2n-1}^*) = DFT_{2n}(y_0, y_1, \ldots, y_{n-1}, 0, \ldots, 0)$
- $Z^* = (z_0^*, z_2^*, \ldots, z_{2n-1}^*), \quad z_i^* = x_i^* \cdot y_i^*$
- $Z = (z_0, z_2, \ldots, z_{2n-1}) = DFT_{2n}^{-1}(Z^*)$
- $Z = \sum_{i=0}^{2n-1} z_i 10^i$

Do testowania można na początku wykorzystać poniższą prostą implementację DFT:

```
from cmath import exp
from math import pi

def omega(k,n):
    return exp(-2j*k*pi/n)

def dft(x,n):
    return [sum(x[i]*omega(i*k,n) if i<len(x) else 0 for i in range(n)) for k in range(n)]

def idft(x,n):
    return [int(round(sum(x[i]*omega(-i*k,n) if i<len(x) else 0 for i in range(n)).real)/n) for k in 
range(n)]
```

Przykład zastosowania **dft** i **idft**:

```
>>> x = dft([1, 2, 3, 4, 5], 5)
>>> x
[(15+0j), (-2.5000000000000001+3.440954801177934j), (-2.5+0.8122992405822647j), (-2.499999999999999-
0.8122992405822673j), (-2.4999999999999964-3.440954801177935j)]
>>> idft(x, 5)
[1, 2, 3, 4, 5]
```

Przykład działania całego programu:

```
>>> A = '13123122312321312312312312312312312123312332313349'
>>> B = '12123123112231231213123123213212312311123231231'

>>> a = FastBigNum(A)
>>> b = FastBigNum(B)

>>> print(a*b*a*b)

25310655007456290142240367588665613884637684032037764664O
728789005971339977090446005669753867738453444565943594360
601270478845270512230832242981112065324812816791957946651
0444942972440921967161
```

Napisz trzy implementacje z różnymi algorytmami dla transformaty Fouriera. W pierwszej wykorzystaj podane kody dla dft i idft. W drugiej napisz swoją własną implementacje FFT (w języku Python). W trzeciej wykorzystaj **numpy.fft**. Porównaj czasy wykonywania i wklej do tabelki (plik tekstowy z wynikami np. fastbignum_benchmark.txt). Do testów wykorzystaj liczby 100 000, 500 000, 1 000 000. Na przykład możemy wygenerować liczby tak

```
>>> a = ''.join([random.choice("0123456789") for i in range(500000)])
>>> b = ''.join([random.choice("0123456789") for i in range(500000)])
```

później testujemy czas wykonywania mnożenia **a*b**, klasycznie mnożenie można wykonać tak **int(a)*int(b)**, aby sprawdzić, czy kod poprawnie je wykonał!.