Laboratorium grafowe 2

Projekt „Matematyka dla Ciekawych Świata”,
Jakub Jałowiec, Tomasz Świerczewski, Łukasz Mazurek

29.03.2021

1 Forkowanie replit'a *MDCS-Lab2*

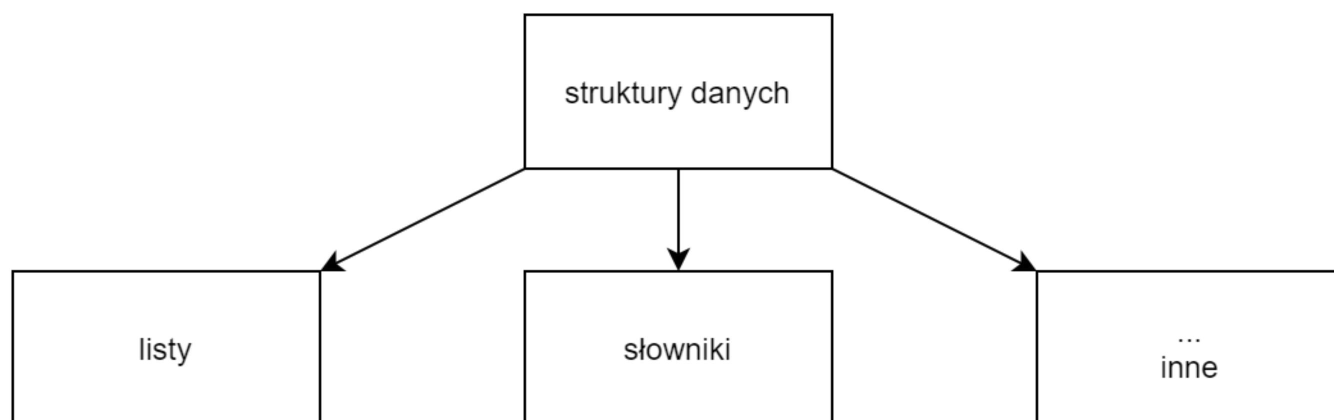
Zajęcia zaczniemy od sklonowania (*sforkowania*) replit'a z funkcjami do rysowania grafów. Funkcje te przydadzą się pod koniec zajęć i w pracy domowej.

2 Słowniki

2.1 Wstęp

Na poprzednich zajęciach dowiedzieliśmy się m. in. czym są *listy* w języku Python. Warto wiedzieć, że listy to jedna ze *struktur danych* - to znaczy jest to jeden z wielu sposobów reprezentowania danych w komputerze. Kolejnym sposobem reprezentowania danych w komputerze, który omówimy są *słowniki*.

Rysunek 1: Typy struktur danych. Będziemy korzystać z dwóch różnych typów struktur danych: **list** i **słowników**. Warto mieć z tyłu głowy, że istnieje więcej typów struktur danych, ale nie będziemy nimi się zajmować na naszych zajęciach.



Różne struktury danych potrzebne są do przetwarzania danych na różne sposoby. Na tych zajęć poznamy **słowniki** - pod koniec zajęć wykorzystamy je do naszej pierwszej implementacji grafów.

Zacznijmy od pytania po co potrzebujemy kolejnej struktury danych poza listami.

Czasami potrzebujemy przechowywać pary wartości (`klucz`, `wartość`). Na przykład możemy potrzebować zapamiętać kto jest czyim sąsiadem: (`Jan`: `Ola`), (`Ola`: `Ala`), (`Ala`: `Jan`) itd. Moglibyśmy to robić przy pomocy zwykłej listy przechowującej napisy, np. `['Jan:Ola', 'Ola:Ala', 'Ala:Jan']`. Innymi słowy, aby osiągnąć pożądany efekt możemy przechowywać na liście wpisy kto jest czyim sąsiadem w taki sposób, że oddzielamy w napisie jedną osobę od drugiej jakąś ściśle określoną sekwencją znaków, np. dwukropkiem: ':'. Taki sposób przechowywania relacji bycia sąsiadem ma dwie wady - zobrazujmy to na przykładzie sprawdzania jakiego sąsiada ma Ala:

- musimy przeszukać całą listę takich napisów i sprawdzić czy którykolwiek z tych napisów zaczyna się od imienia "Ala"
- po znalezieniu wpisu zaczynającego się od "Ala" musimy usunąć fragment 'Ala:' ze znalezionego napisu i dopiero tak uzyskany napis jest imieniem sąsiada Ali.

W praktyce taki sposób modelowania relacji między dwoma rzeczami nie jest wykorzystywany ze względu na jego niewygodę. W języku Python istnieje oddzielna struktura danych do przechowywania par (`klucz, wartość`) - nazywa się *słownik* (lub inaczej *tablica asocjacyjna*, *mapa*; po angielsku *hash map*, *dictionary*, *hash table*, my będziemy konsekwentnie używać nazwy "*słownik*"). Hasła w słowniku "informatycznym" są podobnie jak w "książkowym" słowniku posortowane alfabetycznie (rosnąco, jeśli kluczami są liczby). Przez to umożliwiają szybkie - tzn. nie wymagające przeglądania całej listy elementów - sprawdzanie wartości odpowiadającej "hasłom". Za chwilę przekonamy się jak to wygląda w praktyce. Warto wiedzieć, że *słownik* jako struktura danych znajduje szerokie zastosowanie w informatyce. Przy ich pomocy modelowane są dowolne relacje, np. "ktoś jest czyimś sąsiadem", "coś jest odległe od czegoś innego o X kilometrów", "ktoś jest czyimś znajomym" itd. Z tego powodu przy pomocy słowników często implementowane są np. bazy danych, a my użyjemy ich do implementacji grafów.

Przykład słownika nr 1. Kilka haseł z "Encyklopedia Szkolna. Fizyka." Wydawnictwo Zielona Sowa.

hasło	definicja
"ablacja"	"proces będący jednym z rodzajów wymiany ciepła przez konwekcję"
"amper"	"(ozn. A) - w układzie SI jednostka natężenia prądu elektrycznego"
"Baran"	"gwiazdozbiór równikowy"

Przykład słownika nr 2. Wartościami w słowniku mogą być liczby.

imię	wiek
"Ala"	15
"Jan"	16
"Ola"	17

Przykład słownika nr 3. Wartościami w słowniku mogą być listy.

imię	sąsiedzi
"Ala"	["Jan", "Ola"]
"Jan"	["Ola", "Ala"]
"Ola"	["Ala", "Jan", "Ela"]
"Ela"	["Ola"]

Przykład słownika nr 4. Kluczami w słowniku mogą być liczby.

numer	imię
1	"Jan"
2	"Ola"
3	"Ala"
4	"Ola"

Powyżej przedstawione są cztery przykłady *słowników*. Wielkości w lewej kolumnie nazywamy *kluczami*, a wielkości w prawej kolumnie nazywamy *wartościami*.

2.2 Słowniki w języku Python

Przejdźmy do tego jak w praktyce korzysta się ze słowników w języku Python: jak utworzyć słownik, jak dodawać do niego wartości pod podanym kluczem, jak zmieniać wartości dla istniejącego już w słowniku klucza, jak iterować po słowniku.

Przykład Poniższy fragment kodu pokazuje najprostszą możliwą definicję słownika i jak iterować po jego kluczach pętlą **for**.

```
1 slownik = {"Ala": 15, "Jan": 16, "Ola": 17}
2 for klucz in slownik:
3     print(klucz + " -> " + str(slownik[klucz]))
```

```
1 Ala -> 15
2 Jan -> 16
3 Ola -> 17
```

W powyższym przykładzie:

- nawiasy klamrowe - {...} - oznaczają początek i koniec definiowania słownika,
- klucz i wartość w każdej parze są rozdzielane dwukropkiem - :, np. "Ala": 15 znaczy że Ali odpowiada wartość 15 (np. lat)
- gdy wymieniamy w definiowanym słowniku wszystkie pary klucz: wartość to rozdzielamy je przecinkiem, np. "Ala": 15, "Jan": 16.
- slownik[klucz] oznacza dostęp do wartości w słowniku slownik pod kluczem przechowywanym przez zmienną klucz,
- jeśli chcemy mieć dostęp do każdego klucza w słowniku (a co za tym idzie do każdej wartości, bo mamy do niej dostęp przez slownik[klucz]) to robimy to w pętli w następujący sposób: for klucz in slownik.

Przykład Poniższy fragment kodu pokazuje jak dodawać nowe wartości do słownika i jak je zmieniać.

```
1 slownik = {"Ala": 15, "Jan": 16, "Ola": 17}
2 slownik["Stas"] = 18
3 slownik["Ala"] = 19
4 for klucz in slownik:
5     print(klucz + " -> " + str(slownik[klucz]))
```

```
1 Ala -> 19
2 Jan -> 16
3 Ola -> 17
4 Stas -> 18
```

W powyższym przykładzie:

- w 1. linijce zdefiniowaliśmy słownik z następującymi parami klucz, wartość: "Ala": 15, "Jan": 16 i "Ola": 17,
- w 2. linijce dodaliśmy do tego słownika wartość 18 pod kluczem "Stas",
- w 3. linijce zmieniliśmy wartość dla klucza "Ala" z 15 na 19
- w 4. i 5. linijce wypisaliśmy zawartość słownika

Dla dociekliwych

Słowniki a listy

Głównym powodem, dla którego słowniki są lepsze w niektórych zastosowaniach od list jest to, że dostęp do wartości kryjącej się pod danym kluczem jest ma *stałą złożoność czasową* (mówi się także: *jest w czasie stałym*). Oznacza to, że możemy w wygodny sposób dostawać się ”od ręki” do wartości pod danym kluczem w słowniku. Dla porównania, gdybyśmy chcieli przechowywać w liście te same dane, które przechowujemy w słowniku to najprawdopodobniej musielibyśmy przeitereować po całej liście, aby znaleźć poszukiwany element.

Poniższa tabela pokazuje różnice między listami a słownikami. Różnice dotyczą tego jak zdefiniować nową listę/słownik, jak do nich wstawiać elementy i mieć dostęp do elementów oraz jak iterować po nich.

cecha	**lista**	**słownik**
definiowanie	`lista = [1, 2, 3]`	`slownik = {1: 2, 3: 4}`
dodawanie elementu	`lista = lista + [element]`	`slownik[klucz] = wartosc`
dostęp do elementu	`lista[i] # (i < len(lista))`	`slownik[klucz]`

iteracja po elementach	`for element in lista:`
	`    print(element)`
		`for klucz in slownik:`
		`    print(klucz)`
		`    print(slownik[klucz])`

Zadanie 2.2.1

Zwiększ wiek każdej osoby ze słownika {"Ala": 15, "Jan": 16, "Ola": 17} dwa razy.

Zadanie 2.2.2

Napisz kod, który utworzy *słownik*, w którym znajdują się pary liczb `(i, i+1)` dla i w przedziale `<0,3>`. Innymi słowy, kod powinien generować taki słownik: `{0: 1, 1: 2, 2: 3, 3: 4}`.

Zadanie 2.2.3

Wartościami przechowywanymi w słownikach mogą być listy. Napisz kod, który utworzy *słownik*, w którym znajdują się pary `(i, [0, 1, ..., i])` dla i w przedziale `<0,3>`. Innymi słowy, kod powinien generować taki słownik: `{0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 2, 3]}`.

3 Funkcje

Wszystkie programy, które do tej pory pisaliśmy można podzielić na dwa rodzaje:

- ciąg jednolinijkowych poleceń, które wpisujemy bezpośrednio w konsoli Pythona (prawe, czarne okienko na stronie `repl.it`),
- kod źródłowy w pliku (lewe, białe okienko na stronie `repl.it`)

Często będziemy chcieli wielokrotnie wykorzystać raz napisany fragment kodu. W tym celu będziemy tworzyć własne *funkcje*.

Definicja funkcji ma następującą postać:

```
def nazwa_funkcji(argument1, argument2, ...):
  linijka 1...
  linijka 2...
  ...
  (return ...)
```

Zwróć uwagę na następujące rzeczy w definicji *funkcji*:

- w pierwszej linijce znajdują się:
  - słowo kluczowe i nazwa funkcji: `def nazwa_funkcji`
  - następnie argumenty (inaczej parametry) oddzielane przecinkami w nawiasach okrągłych; ewentualnie nawias okrągły może być pusty, jeśli nie chcemy, aby funkcja przyjmowała argumenty
  - dwukropek za argumentami (`:`); argumenty to zmienne, które istnieją tylko w funkcji,
- każda z kolejnych linijek (`linijka 1...` i `linijka 2...`) jest *wcięta* względem pierwszej linijki
- linijki `1`, `2`, itd. mogą zawierać wszystkie znane nam do tej pory konstrukcje językowe Python, np.:
  - definicje zmiennych i operacje na nich, np. `x = 1`, `z = x * 2`
  - instrukcje sterujące jak `for element in lista:` albo `if zmienna = 1:`
  - wywołania innych funkcji, np. `print(...)`
- ostatnia linijka może zawierać słowo kluczowe `return`, aby *zwracać* wartość z funkcji, np. jedną z obliczonych wewnątrz funkcji zmiennych; jeśli funkcja kończy się klauzulą `return` to wartość zwracaną przez funkcję można przypisać do zmiennej.

Jako że definicje funkcji składają się zawsze z co najmniej kilku linijek, będziemy je pisać (podobnie jak pętle `for`) w **lewym okienku** interpretera (w pliku z kodem źródłowym).

Przykład Napiszmy funkcję, która zwraca swój argument podniesiony do kwadratu i wypiszmy jej wynik dla kilku przykładów:

```python
def kwadrat(x):
  return x * x
a = kwadrat(7)
print(a)
b = kwadrat(2+3)
print(b)
```

```
> a = kwadrat(7)
> print(a)
49
> b = kwadrat(2 + 3)
> print(b)
25
```

W powyższym przykładzie:

- w 1. linijce otworzyliśmy definicję funkcji `kwadrat`
- w 2. linijce podaliśmy jednolinijkowe *ciało* (zawartość) funkcji `kwadrat` - po prostu zwracam przez słowo kluczowe `return` wartość argumentu `x` podniesioną do kwadratu
- w 3. i 5 linijce linijce przypisaliśmy wartość zwróconą przez wywołanie `kwadrat(7)` i `kwadrat(2+3)` do zmiennych
- w 4. i 6. linijce wypisaliśmy zmienne przechowujące wyniki

Przykład Funkcje mogą mieć po kilka argumentów i nie muszą zwracać wartości - zamiast tego mogą np. wypisywać coś na ekran.

```python
def dodaj(a, b, c):
  print(a + b + c)
dodaj(7, 8, 9)
dodaj(7, -1, -6)
```

```
24
0
```

W powyższym przykładzie:

- w 1. i 2. linijce zdefiniowaliśmy funkcję `dodaj`
- w 3. i 4. linijce wywołaliśmy ją dla różnych argumentów.

Dla dociekliwych

Biblioteka standardowa języka Python
Wcześniej mogliśmy o tym nie mówić, ale do funkcji wbudowanych w język Python (takich, które możemy "od ręki" używać w pliku `main.py` po lewej stronie lub w konsoli po prawej stronie replit'a) należą funkcje takie jak: `print()`, `range()`, `str()`. Są one częścią *biblioteki standardowej języka Python*. Więcej o funkcjach dostępnych domyślnie w Pythonie można przeczytać na tej stronie: The Python Standard Library.

Zadanie 3.0.1

Napisz funkcję `radosc(lista)`, która dla listy przekazanej jako argument wypisze każdy element listy z dopiskiem `" :)"` w oddzielnej linijce. Zignoruj wariant, kiedy przekazujemy do funkcji nieprawidłowe argumenty (np. liczbę zamiast listy).

Zadanie 3.0.2

Napisz funkcję `mnozenie(a, b)`, która zwraca (przez `return`) iloczyn dwóch liczb, np. `mnozenie(7, 8)` powinno zwrócić `56`. Zignoruj wariant, kiedy przekazujemy do funkcji nieprawidłowe argumenty (np. słowa zamiast liczb).

Zadanie 3.0.3

Napisz funkcję `1_do_n(N)`, która utworzy listę liczb od 1 do N. Przykładowo `1_do_n(5)` powinno zwrócić `[1,2,3,4,5]`. Zignoruj wariant, kiedy przekazujemy do funkcji nieprawidłowe argumenty (np. słowo zamiast liczby).

Zadanie 3.0.4

Napisz funkcję `utworz_slownik_z_listy_2d(lista_2d)`, która zwraca słownik utworzony z listy list dwuelementowych, np. dla listy `["Ala", 15], ["Jan", 16], ["Ola", 17]]` funkcja powinna zwrócić słownik `{"Ala": 15, "Jan": 16, "Ola": 17}`. Zignoruj wariant, kiedy przekazujemy do funkcji nieprawidłowe argumenty (np. liczbę zamiast listy list).

4 Grafy

4.1 Wstęp

Jak już zapewne wiesz, graf to para zbiorów `(wierzchołki, krawędzie)`, gdzie krawędzie muszą zaczynać się i kończyć w elementach zbioru `wierzchołki`. Przykładowy wygląd grafu jest pokazany na Rysunku 2.

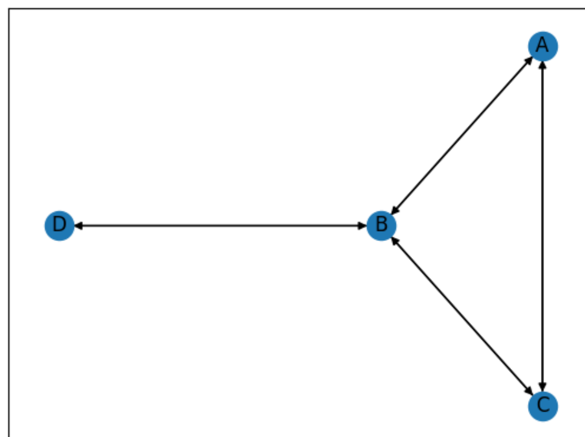
4.2 Implementacja

Grafy można implementować przy pomocy słownika list. Niech wierzchołki będą numerowane liczbami od 1 do N. Wtedy graf można zaimplementować jako słownik, w którym kluczami będą liczby od 1 do N, a krawędzie będą reprezentowane przez listę pozostałych wierzchołków, odpowiadającą danemu kluczowi w słowniku. To może brzmieć trochę skomplikowanie, dlatego przejdźmy do przykładu.

Przykład Spróbujmy w możliwie prosty sposób zdefiniować graf w języku Python. Rysunek 2 pokazuje jak ten słownik wyglądałby na papierze.

```
1 graf = {"A": ["B", "C"], "B": ["A", "D", "C"], "C": ["A", "B"], "D": ["B"]}
```

Rysunek 2: Przykład grafu. *Wierzchołki* to owale o nazwach `A, B, C, D`. *Krawędzie* to strzałki łączące wierzchołki; widoczne są krawędzie: `(A, B), (A, C), (B, A), (B, C), (B, D), (C, A), (C, B), (D, B)`. Zauważ, że strzałki na rysunku są ”dwukierunkowe”. Na przykład strzałka `A<->B` jest złożona z dwóch strzałek ”jednokierunkowych”: `A->B` i `A<-B`. Ogólnie możemy także zostawić strzałki w jedną stronę.



4.3 Rysowanie grafów

Upewnij się, że po sforkowaniu replit'a z funkcjami do rysowania grafów folder *wizualizacje* znajduje się obok pliku `main.py`. Teraz możemy zacząć rysować grafy.

Przykład Narysujmy graf z Rysunku 2. W pliku `main.py` wpisz następujący kod:

```
from wizualizacje.funkcje import *
graf = {"A": ["B", "C"], "B": ["A", "D", "C"], "C": ["A", "B"], "D": ["B"]}
rysuj_graf(graf)
```

W przykładzie wyżej:

- w 1. linijce importujemy funkcję *rysuj_graf* z modułu (biblioteki) *funkcje* znajdującej się w folderze *wizualizacje*; od tej pory mamy dostęp do wszystkich funkcji zdefiniowanych w tej bibliotece
- w 2. linijce definiujemy graf w oparciu o słownik list
- w 3. linijce wywołujemy zaimportowaną funkcję do rysowania grafu, reprezentowanego przez słownik list - efekt jest widoczny na Rysunku 2.

Powyższy przykład *importuje* bibliotekę języka Python, znajdującą się w folderze *wizualizacje*. Biblioteka ta ma dwie funkcje: `rysuj_graf()` oraz `rysuj_drzewo()`. Obie te funkcje przyjmują dokładnie jeden argument: słownik list reprezentujący graf.

Dla dociekliwych

Dlaczego w replit'cie na drugie laboratoria są dwie oddzielne funkcje do rysowania: jedna dla grafów a druga dla drzew?
Odpowiedź na to jest trochę zbyt skomplikowana na te zajęcia. Mówiąc w skrócie - nie istnieje uniwersalny algorytm "ładnego" rysowania grafów. Funkcja `rysuj_graf()` korzysta z metody rysowanej zwanej spring layout. Jest ona odpowiednia dla grafów, które nie są drzewami. Z kolei funkcja `rysuj_drzewo()` korzysta z metody rysowania, która od początku zakłada, że graf jest drzewem. Możesz spróbować wywołać funkcję `rysuj_graf()` dla grafu będącego drzewem. W efekcie dostaniesz "niezbyt ładny" rysunek np. z przecinającymi się krawędziami. Następnie narysuj ten sam graf funkcją `rysuj_drzewo()` i zobacz różnicę.

Zadanie 4.3.1

Napisz funkcję `graf_pelny(N)` rysującą graf pełny o `N` wierzchołkach.

Zadanie 4.3.2

Napisz funkcję `graf_pusty(N)` rysującą graf bez krawędzi o `N` wierzchołkach.

5 Zadania dodatkowe

Zadanie 5.0.1

Napisz funkcję `bezwzgledne(lista)`, która dla danej listy liczb wypisze listę wartości bezwzględnych tych liczb, tj. liczby ujemne zamieni na przeciwne, a liczby nieujemne pozostawi bez zmian. Poszczególne liczby powinny być oddzielone pojedynczymi spacjami. Przykładowe użycie funkcji powinno wyglądać następująco:

```
> bezwzgledne([5, -10, 15, 0])
5 10 15 0
```

Zadanie 5.0.2

Napisz funkcję `parzyste_nieparzyste(lista)`, która dla danej listy liczb wypisze ich ile było na niej list parzystych i ile nieparzystych. Przykładowe użycie funkcji powinno wyglądać następująco:

```
> parzyste_nieparzyste([1, 1, 3, 4, 4, 5, 6, 7])
parzyste: 3
nieparzyste: 5
```

Zadanie 5.0.3

Napisz funkcję `suma(lista)`, która dla danej listy liczb wypisze ich sumę. Przykładowe użycie funkcji powinno wyglądać następująco: `suma([1, 3, 5, 7])` powinno zwrócić `16`.

Zadanie 5.0.4

Napisz funkcję `zlicz(lista)`, która dla danej listy stworzy słownik zawierający zliczenia każdej z wartości. Przykładowe użycie funkcji powinno wyglądać następująco:

```
> zlicz([1, 2, 2, 3, 3, 3, "a", "a", "b"])
{1: 1, 2: 2, 3: 3, 'a': 2, 'b': 1}
```

Zadanie 5.0.5

Napisz funkcję `rozpakuj(slownik)`, która dla danego słownika stworzy słownik zawierający zliczenia każdej z wartości. Przykładowe użycie funkcji powinno wyglądać następująco:

```
> rozpakuj({'a': 2, 2: 2, 3: 3, 1: 1, 'b': 1})
['a', 'a', 2, 2, 3, 3, 3, 1, 'b']
```

Zadanie 5.0.6**

Napisz funkcję `quicksort(lista)`, która implementuje algorytm quicksort sortowania listy liczb. Przykładowe użycie funkcji powinno wyglądać następująco:

```
> quicksort([7, 2, 4, 3, 1])
[1, 2, 3, 4, 7]
```

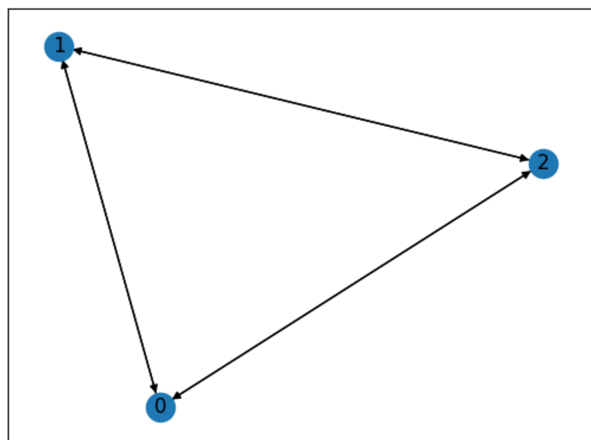
6 Praca domowa nr 2

Uwaga. O ile nie przekazano na zajęciach inaczej to pracę domową należy przesłać prowadzącym w wiadomości prywatnej na Discordzie.

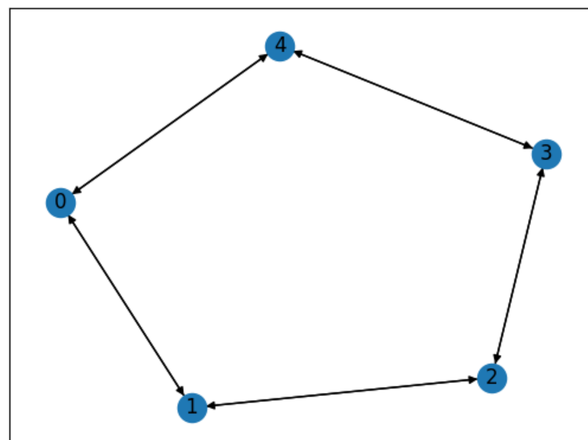
Zadanie 6.0.1 — 1 pkt

Napisz funkcję `cykl(N)` rysującą graf cykl o `N` wierzchołkach (tzn. spójny graf w którym każdy wierzchołek jest połączony tylko z dwoma innymi).
Wskazówka I: możesz użyć funkcji `rysuj_graf()` z replit'a z funkcjami do rysowania grafów do sprawdzania jak wygląda wygenerowany cykl.
Wskazówka II: Użyj dzielenia modulo (%) do obliczenia indeksów sąsiadów danego wierzchołka.
Przykładowe użycie funkcji powinno wyglądać następująco:

```
> cykl(3)
{0: [2, 1], 1: [0, 2], 2: [1, 0]}
> cykl(5)
{0: [4, 1], 1: [0, 2], 2: [1, 3], 3: [2, 4], 4: [3, 0]}
```



(a) `cykl(3)`



(b) `cykl(5)`

Rysunek 3: Dwa przykłady grafów-cykli - po lewej o liczbie wierzchołków równej 3, po prawej o liczbie wierzchołków równej 5.

Zadanie 6.0.2 — 1 pkt

Napisz funkcję `najwiekszy_kwadrat(N)`, która poda największy kwadrat mniejszy lub równy od podanej liczby naturalnej. Przykładowe użycie funkcji powinno wyglądać następująco:

```
> najwiekszy_kwadrat(24)
16
> najwiekszy_kwadrat(25)
25
> najwiekszy_kwadrat(123456789)
123454321
> najwiekszy_kwadrat(74967204759)
74966987601
```

Zadanie 6.0.3 — 1 pkt

Napisz funkcję NWD(a, b), która dla dwóch liczb naturalnych poda ich największy wspólny dzielnik. Przykładowe użycie funkcji powinno wyglądać następująco:

```
> NWD(7, 13)
1
> NWD(24, 16)
8
> NWD(100, 100)
100
```

Zadanie 6.0.4 — 1 pkt

Napisz funkcję suma_od_a_do_b(a, b), która dla dwóch liczb a i b zwróci sumę liczb całkowitych większych od a i mniejszych od b. Przykładowe użycie funkcji powinno wyglądać następująco:

```
> suma_od_a_do_b(4, 7)
11
> suma_od_a_do_b(-3, 3)
0
> suma_od_a_do_b(-6, -3)
-9
```

Zadanie 6.0.5 — 1 pkt

Napisz funkcję zwieksz_elementy_listy(lista, X), która zwraca listę liczb powiększoną o X. Przykładowo zwieksz_elementy_listy([1, 2, 3, 4], 5) powinno zwrócić [6, 7, 8, 9]. Zignoruj wariant, kiedy przekazujemy do funkcji nieprawidłowe argumenty (np. dwie liczby zamiast listy i liczby). Przykładowe użycie funkcji powinno wyglądać następująco:

```
> zwieksz_elementy_listy([1, 2, 3], 4)
[5, 6, 7]
> zwieksz_elementy_listy([], 100)
[]
> zwieksz_elementy_listy([1, 1, 1], -1)
[0, 0, 0]
```

Zadanie 6.0.6 — 1 pkt

Napisz funkcję skresl_wielokrotnosci_N(zakres, N), która dla liczb od 1 do zakres wypisze wszystkie liczby, które nie są podzielne przez N. Przykładowe użycie funkcji powinno wyglądać następująco:

```
> skresl_wielokrotnosci_N(13, 3)
[1, 2, 4, 5, 7, 8, 10, 11, 13]
> skresl_wielokrotnosci_N(6, 1)
[]
```

Zadanie 6.0.7 — 2 pkt

Napisz funkcję `kalkulator(dzialanie, a, b)`, która dla danego `działania` (`"+"`, `"-"`, `"*"` albo `"/"`) wypisze funkcją **`print`** wartość tego działania na liczbach `a` i `b`. Nie pozwól na dzielenie przez zero.

```
> kalkulator("*", 3, 4)
12
> kalkulator("+", 3, 4)
7
> kalkulator("/", 3, 4)
0.75
> kalkulator("-", 3, 4)
-1
```

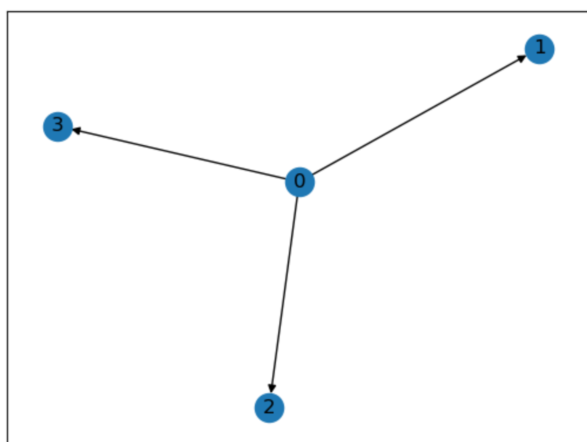
Zadanie 6.0.8 — 2 pkt

Napisz funkcję `gwiazdka(N)`, która zwróci *graf-gwiazdę* o rozmiarze N. W *gwieździe* jeden wierzchołek jest połączony z *(N-1)* pozostałymi wierzchołkami, które z kolei nie są połączone między sobą.

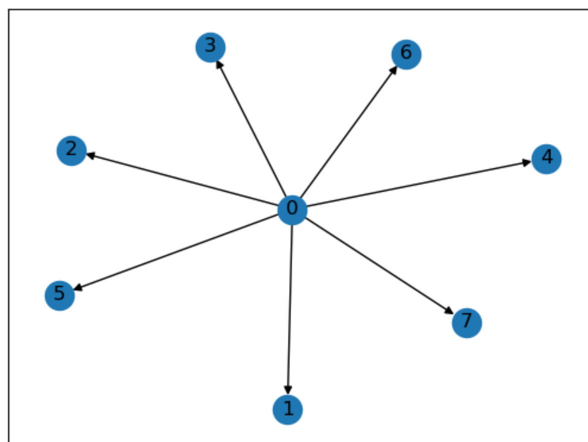
Wskazówka: możesz użyć funkcji `rysuj_graf()` replit'a z funkcjami do rysowania grafów do sprawdzania jak wygląda wygenerowana gwiazdka.

Przykładowe użycie funkcji powinno wyglądać następująco:

```
> gwiazdka(4)
{0: [1, 2, 3], 1: [], 2: [], 3: []}
> gwiazdka(8)
{0: [1, 2, 3, 4, 5, 6, 7], 1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: []}
```



(a) `gwiazdka(4)`



(b) `gwiazdka(8)`

Rysunek 4: Dwa przykłady grafów-gwiazdek - po lewej gwiazdka o 4 wierzchołkach, po prawej gwiazdka o 8 wierzchołkach.

Zadanie 6.0.9 — 2 pkt

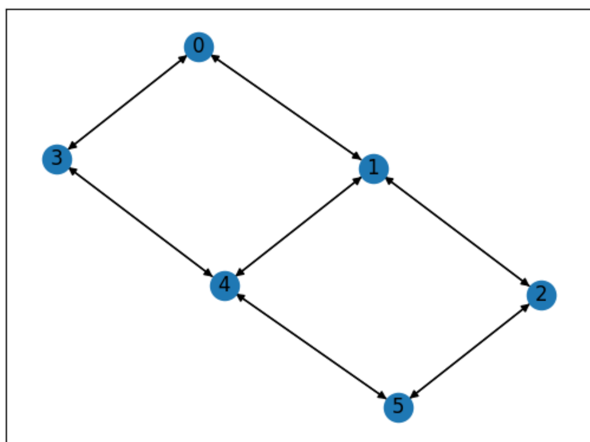
Wartościami przechowywanymi w słownikach mogą być także słowniki. Napisz funkcję `slownik_slownikow(N)`, która utworzy *słownik* postaci `1: {1: 1}, 2: {1: 1, 2: 2}, , ..., N: {1: 1, 2: 2, ..., (N-1): (N-1), N: N}` Przykładowe użycia funkcji powinny wyglądać następująco:

```
> slownik_slownikow(1)
{1: {1: 1}}
> slownik_slownikow(2)
{1: {1: 1}, 2: {1: 2, 2: 2}}
> slownik_slownikow(3)
{1: {1: 1}, 2: {1: 1, 2: 2}, 3: {1: 1, 2: 2, 3: 3}}
```

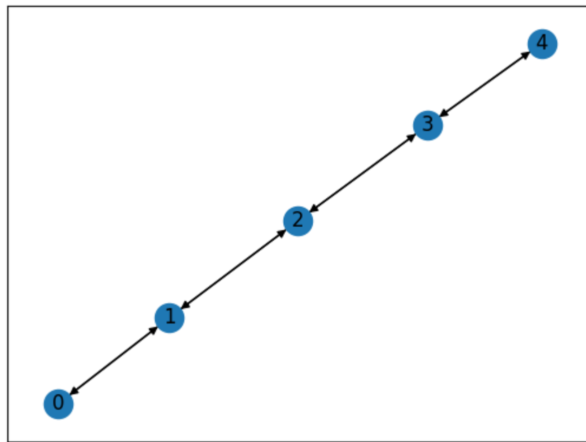
Zadanie 6.0.10 — 3 pkt

Napisz funkcję `krata(szerokosc, wysokosc)`, która generuje *graf-kratę* o rozmiarach `szerokosc x wysokosc`. *Krata* to taki graf, który układa się w siatkę 2D. Sąsiadami są tylko te wierzchołki, które mają różnią się o 1 współrzędną X lub Y.
Przykładowe użycie funkcji powinno wyglądać następująco:

```
> krata(2, 3)
{0: [3, 1], 1: [4, 2, 0], 2: [5, 1], 3: [0, 4], 4: [1, 5, 3], 5: [2, 4]}
> krata(1, 5)
{0: [1], 1: [2, 0], 2: [3, 1], 3: [4, 2], 4: [3]}
```



(a) `krata(2, 3)`



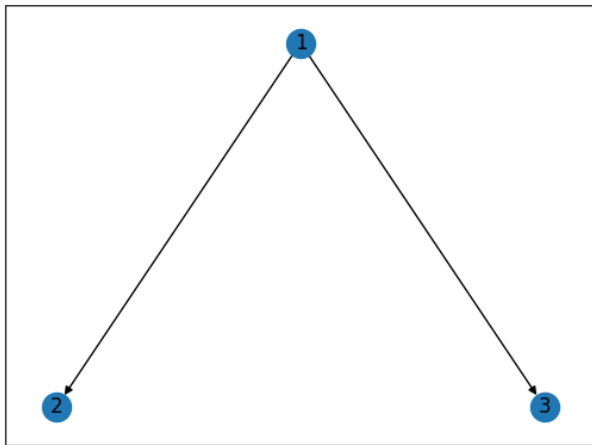
(b) `krata(1, 5)`

Rysunek 5: Dwa przykłady grafów-krat - po lewej o wymiarze 2x3, po prawej o wymiarze 1x5.

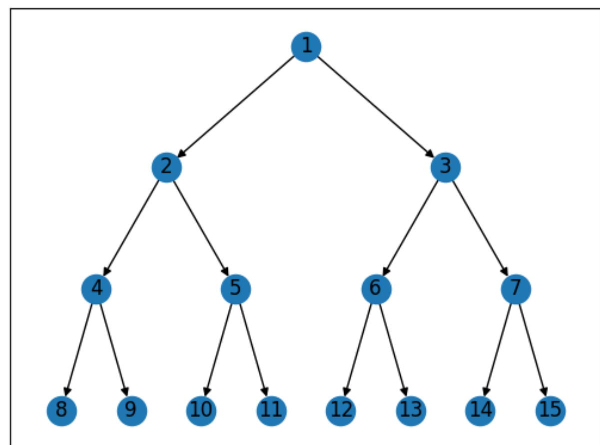
Zadanie 6.0.11 — 3 pkt

Napisz funkcję `drzewo_binarne(wysokosc)`, która wygeneruje drzewo binarne o podanej wysokości. Wartością zwracaną powinien być słownik list, gdzie kluczami są numery wierzchołków, a wartościami listy z numerami wierzchołków, do których wychodzą krawędzie z wierzchołka o danym numerze.
Drzewo binarne to takie drzewo, w którym każdy sąsiad ma dokładnie 2 lub 0 sąsiadów.
Wskazówka I: użyj funkcji `rysuj_drzewo()` z replit'a z funkcjami do rysowania grafów do sprawdzania jak wygląda wygenerowane drzewo.
Wskazówka II: w drzewie binarnym o wysokości N jest 2^N - 1 wierzchołków. Wierzchołek o numerze N jest połączony z wierzchołkami o numerach $N * 2$ oraz $N * 2 + 1$.
Przykładowe użycie funkcji powinno wyglądać następująco:

```
> drzewo_binarne(2)
{1: [2, 3], 2: [], 3: []}
> drzewo_binarne(4)
{1: [2, 3], 2: [4, 5], 3: [6, 7], 4: [8, 9], 5: [10, 11], 6: [12, 13], 7: [14, 15],
↪  8: [], 9: [], 10: [], 11: [], 12: [], 13: [], 14: [], 15: []}
```



(a) `drzewo_binarne(2)`



(b) `drzewo_binarne(4)`

Rysunek 6: Dwa przykłady drzew binarnych - po lewej o wysokości 2, po prawej o wysokości 4.