UNIWERSYTET JAGIELLOŃSKI

Wydział Fizyki, Astronomii i Informatyki Stosowanej

PRACA MAGISTERSKA

Temat: Implementacja wybranych algorytmów i struktur danych przy pomocy języka Python - algorytmy grup permutacji.

Tomasz Gądek

Promotor:
dr hab. Andrzej Kapanowski

Kraków, 2013

3589886368(1)

Kraków, dnia 27.09.2013 r.

Ja niżej podpisany Tomasz Gądek (nr indeksu: 1084583) student Wydziału Fizyki, Astronomii i Informatyki Stosowanej Uniwersytetu Jagiellońskiego kierunku Informatyka, oświadczam, że przedłożona przeze mnie praca magisterska pt. „Implementacja wybranych algorytmów i struktur danych przy pomocy języka Python - algorytmy grup permutacji" przedstawia wyniki badań wykonanych przeze mnie osobiście, pod kierunkiem dr. hab. Andrzeja Kapanowskiego. Pracę napisałem samodzielnie.

Oświadczam, że moja praca dyplomowa została opracowana zgodnie z Ustawą o prawie autorskim i prawach pokrewnych z dnia 4 lutego 1994 r. (Dziennik Ustaw 1994 nr 24 poz. 83 wraz z późniejszymi zmianami).

Jestem świadom, że niezgodność niniejszego oświadczenia z prawdą ujawniona w dowolnym czasie, niezależnie od skutków prawnych wynikających z ww. ustawy, może spowodować unieważnienie tytułu nabytego na podstawie tej pracy.

podpis studenta

Składam serdeczne podziękowania Promotorowi Panu dr. hab. Andrzejowi Kapanowskiemu za życzliwość, cenne uwagi, wszechstronną pomoc oraz poświęcony czas.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów i struktur danych związanych z grupami permutacji. Stworzono trzy klasy: *Perm* dla permutacji, *Group* dla grup permutacji, oraz wyjątek *PermError* do raportowania błędów w poprzednich klasach.

Klasa *Perm* przechowuje wewnętrznie permutację jako listę liczb całkowitych o ustalonej długości. Interfejs permutacji zawiera metody do obliczania rzędu, parzystości, permutacji odwrotnej, porównywania i mnożenia permutacji, potęgowania, znajdowania struktury cykli, permutacji przypadkowej, rangi permutacji.

Klasa *Group* reprezentuję grupę zawierającą permutacje o ustalonej długości. Interfejs klasy pozwala na obliczanie rzędu grupy, testowanie przynależności permutacji do grupy, wstawianie nowej permutacji do grupy, iterowanie po elementach grupy. Można sprawdzać, czy grupa jest trywialna, abelowa, czy jest podgrupą (zwykłą lub normalną) innej grupy, jakie jest centrum grupy. Można szukać centralizatora i normalizatora w danej grupie, a także podgrupy składającej się z elementów o danej właściwości. Można tworzyć nowe grupy jako iloczyny proste innych grup. W klasie *Group* zaimplementowano również metody związane z działaniem grupy na zbiorze. Można wyznaczyć orbity i stabilizatory. Można znaleźć nową grupę indukowaną na jednej orbicie przez działanie starej grupy.

W pracy umieszczono przykładowe obliczenia dla grup kostek Rubika o różnych rozmiarach. Zamieszczono także wyniki testów wydajnościowych oraz testy poprawności kodu, na bazie modułu *unittest*. W celu lepszego zobrazowania wydajności algorytmów, w pracy przedstawiono dwie implementacje klasy *Group*, mające ten sam interfejs. W implementacji podstawowej grupa jest reprezentowana przez słownik, który przechowuje wszystkie permutacje należące do grupy. W implementacji zaawansowanej wykorzystuje się strukturę danych wprowadzoną przez Simsa z silnymi generatorami i układem równoległym.

Słowa kluczowe: permutacje, grupy permutacji, obliczeniowa teoria grup, algorytm Schreiera-Simsa.

Spis treści

1. Wstęp

Praca magisterska poświęcona jest implementacji wybranych algorytmów i struktur danych na potrzeby grup permutacji. Do analizy małych grup wystarcza przeprowadzenie obliczeń na kartce papieru. Jednak dla większych grup spotykanych w praktyce (np. grupy sporadyczne) niezbędne jest stosowanie obliczeń komputerowych. Zagadnienia te zalicza się do obliczeniowej teorii grup. W dalszych podrozdziałach zostanie opisana pokrótce obliczeniowa teoria grup, a następnie podamy cele niniejszej pracy i organizację materiału.

1.1. Obliczeniowa teoria grup

Obliczeniowa teoria grup (OTG) zajmuje się projektowaniem, analizą i implementacją algorytmów działających na grupach [15]. Jest to dziedzina na styku matematyki i informatyki. Główne obszary OTG to m.in. algorytmy dla grup policyklicznych, grup permutacji, grup macierzy i teorii reprezentacji. Grupy permutacji są jednym z najstarszych sposobów reprezentacji grup. Za początek OTG uważa się prace Galois nad grupami permutacji, jeszcze przed zdefiniowaniem abstrakcyjnego pojęcia grupy. Obecnie algorytmy grup permutacji należą do najlepiej rozwiniętych w OTG.

Podstawowe idee dotyczące traktowania grup permutacji pochodzą od Simsa z lat siedemdziesiątych. Nawet dziś metody Simsa są głównymi elementem większości algorytmów. Na pierwszy rzut oka efektywność algorytmów grup permutacji może być zaskakująca. Wejście algorytmu składa się z listy generatorów. Takie podejście jest bardzo wydajne, ponieważ kilka permutacji grupy symetrycznej S_n może opisać grupę o rozmiarze $n!$ Zwięzłość takiej reprezentacji ma swoją cenę. Wymaga nietrywialnych algorytmów do odpowiedzi na podstawowe pytania, takie jak pytanie o rząd grupy, czy o przynależność permutacji do danej grupy.

Kluczowe pomysły Simsa dotyczyły bazy i silnych generatorów grupy. Technika konstruowania silnych generatorów może być stosowana do innych zadań, takich jak obliczanie domknięcia normalnego, czy obsługa homomorfizmów grup. Inna generacja algorytmów używa techniki dziel i zwyciężaj, która wykorzystuje strukturę orbit i strukturę blokową grupy.

Warto podkreślić, że jedną z przyczyn zainteresowania OTG był związek między algorytmami OTG i problemem izomorfizmu grafów. W roku 1982 Luks podał algorytm wielomianowy do testowania izomorfizmu grafów o ograniczonym stopniu wierzchołków. Problem sprowadza się do obliczenia pewnych stabilizatorów.

Do najważniejszych osiągnięć OTG zalicza się wyznaczenie wszystkich grup skończonych o rzędzie do 2000, oraz wyliczenie reprezentacji nieredukowalnych wszystkich grup sporadycznych. Ważnymi programami algebry komputerowej, które znajdują zastosowanie w OTG, są GAP [5] i Magma [6].

1.2. Cele pracy

Głównym celem pracy jest implementacja w języku Python wybranych algorytmów i struktur danych grup permutacji. Dzięki przejrzystej składni Pythona zapis algorytmów będzie bardzo przypominał pseudokod używany w literaturze. Z drugiej strony kod Pythona można wykonać i eksperymentalnie sprawdzić jego poprawność.

Cel ten ma podłoże dydaktyczne. Chcemy pokazać przydatność Pythona do nauki informatyki, programowania, rozwijania myślenia algorytmicznego. Kod Pythona reprezentuje programowanie zorientowane obiektowo, mamy przykłady tworzenia klas, instancji klas, dziedziczenia, kompozycji. Przy okazji prezentowane są dobre praktyki programistyczne (czytelny kod, dobrze dobrane nazwy zmiennych, poprawne komentarze). W pracy zwracamy uwagę na testowanie kodu. Moduły Pythona *unittest* i *doctest* ułatwiają pisanie niezawodnego kodu. Z drugiej strony zwracamy uwagę na złożoność czasową i pamięciową wykorzystywanych algorytmów. Przykładowo, przy tym samym interfejsie grup permutacji pokazujemy dwie różne implementacje, podstawową i zaawansowaną.

Warto podkreślić, że dzięki odpowiednim algorytmom program napisany w Pythonie może być wystarczający do wykonania jakiegoś zadania, mimo że Python zwykle nie jest szybszy niż C/C++/Java.

Przy tworzeniu oprogramowania w pewnym zakresie korzystaliśmy z kodu biblioteki Pythona do matematyki symbolicznej o nazwie *SymPy* [13], a w szczególności z modułu *Combinatorics* [14]. *SymPy* zawiera wiele zaawansowanych technik Pythona oraz występują w nim powiązania obiektów z różnych działów matematyki. Dlatego w wielu miejscach dokonaliśmy uproszczenia kodu i interfejsu, ale bez zmniejszenia funkcjonalności. Korzystaliśmy również z dokumentacji programu GAP [5]. Kod programu GAP jest napisany w języku podobnym do języka Pascal, więc również nie nadawał się do bezpośredniego wykorzystania.

1.3. Organizacja pracy

Rozdział 2 zawiera podstawowe informacje na temat składni, typów i struktur danych języka Python. Przykładowe skrypty opatrzone są komentarzami, które są pomocne w analizie kodu. Rozdział 3 to część teoretyczno-matematyczna, wprowadzająca podstawowe pojęcia z teorii grup. Częścią praktyczną pracy dyplomowej jest rozdział 4, który zawiera ogólny opis metod wchodzących w skład klas *Perm* i *Group*. Praktyczne użycie interfejsu przedstawione zostało w przykładowych sesjach interaktywnych. Zamieszczono tutaj również szczegółowe opisy zadań, jakie realizują metody oraz testy wydajnościowe. Rozdział 5 zawiera przykłady dłuższych obliczeń wykonanych przy pomocy stworzonego oprogramowania. Analizowane będą grupy kostek Rubika o różnych rozmiarach.

2. Wprowadzenie do Pythona

Python jest językiem skryptowym wysokiego poziomu, opracowanym w latach dziewięćdziesiątch przez Guido van Rossuma. Jego ważne cechy takie jak: przejrzystość kodu, możliwość pisania programów zorientowanych obiektowo, dynamiczne typy danych, czy obsługa wyjątków, pozwalają na implementację algorytmów obejmujących różne dziedziny nauki. Sam język nie narzuca stylu programowania, w Pythonie programista może sam zdecydować, czy dogodne dla niego jest programowanie proceduralne, obiektowe, czy funkcyjne. Dodatkowe cechy Pythona takie jak bogata biblioteka standardowa oraz dynamiczne zarządzanie pamięcią są dodatkowymi atutami języka. Python znajduje szerokie zastosowanie na rynku biznesowym. Korzystają z niego prawdziwi giganci, tacy jak Google, Yahoo, Nokia, IBM czy NASA [1].

Wprowadzenie do języka Python ma na celu pokazanie jego cech charakterystycznych, które zostały nadmienione we wstępie. Zauważmy, że listingi ilustrujące omawiane zagadnienia są poprawnymi skryptami Pythona, włączonymi do kodu źródłowego pracy magisterskiej napisanej w systemie LaTeX.

2.1. Arytmetyka i typy danych

Język Python można wykorzystać jako prosty kalkulator, podając polecenia w trybie interaktywnym lub skryptowym. Listing 2.1 przedstawia wykorzystanie operacji arytmetycznych.

Listing 2.1. Operacje arytmetyczne.

```
# -*- coding: cp1250 -*-
# arytmetyka w Pythonie

x, y = 4, 2              # operator przypisania, x = 4 oraz y = 2

print x + y              # dodawanie
print x - y              # odejmowanie
print x * y              # mnożenie
print x / y              # dzielenie całkowitoliczbowe
print x / float(y)       # dzielenie, wynik zmiennoprzecinkowy
print x % y              # dzielenie modulo
print x ** y             # potęgowanie

# można stosować operator przypisania połączony z operatorami
# arytmetycznymi np.:

z = x
z += y                   # równoważnie: z = z + y
```

W języku Python zmienne mogą zawierać dane dowolnego typu. Typy są dynamiczne. Zmienne są uniwersalne, zawierają referencję do obiektu. Python oprócz typów podstawowych: całkowity, zmiennoprzecinkowy, zespolony, string, zawiera również szeroki wachlarz typów kolekcji takich jak: listy, krotki, zbiory [3]. Listing 2.2 przedstawia przykłady wymienionych typów.

Listing 2.2. Typy danych.

```
# -*- coding: cp1250 -*-
# typy danych w Pythonie

calkowita = 100
zmiennoprzecinkowa = 100.0
zespolona = 3 + 4j
string = "napis"
logiczny = 1 < 100          # (True)

lista = [4, -0.5, "v"]
krotka = (1, 10, 100, 1000)
slownik = {"jeden": 1, "dwa": 2, "trzy": 3}
zbior = set([0, 1, 3, 4, 5, 6, 7, 8, 9])
```

Typy w Pythonie są klasami, z których możemy wyprowadzić własne klasy pochodne. Nie wszystkie typy danych mogą podlegać modyfikacjom. Istnieje podział na typy, które podlegają zmianom (mutowalne) i takie, które zmianom nie podlegają (niemutowalne). Zestawienie typów i wymienionych własności opisuje tabelka 2.1 [3].

Tabela 2.1. Podział typów w Pythonie na typy mutowalne i niemutowalne.

Opis obiektu	**Typ**	**Mutowalny/Niemutowalny**
Lista	*list*	Mutowalny
String	*str*	Niemutowalny
Krotka	*tuple*	Niemutowalny
Zbiór	*set*	Mutowalny
Słownik	*dict*	Mutowalny
Liczba	*int, long, float, complex, bool*	Niemutowalny

2.2. Łańcuchy znaków

Łańcuch znaków, czyli string, zaliczany jest do sekwencji znaków, których nie można modyfikować bezpośrednio poprzez podanie indeksu odpowiedniego znaku. Modyfikacja łańcucha znakowego jest możliwa jedynie poprzez zastosowanie zabiegów wycinania i konkatenacji. Zapis łańcuchów znakowych odbywa się poprzez zastosowanie apostrofów lub cudzysłowów [3]. Podstawowe zastosowanie i operacje na stringach przedstawia listing 2.3.

Listing 2.3. Łańcuchy znaków.

```
# -*- coding: cp1250 -*-
# łańcuchy znaków w Pythonie

S = ""; S = ''                # tworzenie pustego napisu
S = "python"                  # tworzenie napisu
S = str([1, 2, 3])            # postać napisowa obiektu
print S
print len(S)                  # długośc napisu - liczba znaków w stringu
```

```
print "Jezyk " + S               # konkatenacja
print 2 * S                      # powielanie stringu
print S[2]                       # odniesienie się do konkretnego znaku
                                 # za pomocą indeksu
print S[:3]                      # wycięcie 3 pierwszych znaków
print S[2:5]                     # wycięcie znaku o indeksie 2, 3 i 4
S_k = S[:]                       # kopiowanie stringów
S_k = str(S)
print S_k in S                   # zawieranie, zwraca typ logiczny
                                 # (True, False)
print S_k not in S
print "%s 2.6" % S               # formatowanie łańcucha znakowego
print "%s+%s=%s" % (2, 3, 2+3)
del S                            # usuwanie stringu
```

2.3. Listy

Listy w Pythonie są to pewnego rodzaju tablice dynamiczne, które w razie potrzeby automatycznie powiększą swój rozmiar. Interesujące jest to, że elementy listy wcale nie muszą być tego samego typu. Elementami listy mogą być inne listy. Obiekty listy mogą być zagnieżdżone na dowolną głębokość [3]. Listing 2.4 demonstruje przykłady zastosowania list.

Listing 2.4. Listy.

```
# -*- coding: cp1250 -*-
# listy w Pythonie

L = []                    # tworzenie listy pustej
L = ["1", 1, 1.1]         # tworzenie listy, elementy nie muszą być
                          # tego samego typu
L2 = list("abc")          # tworzenie listy z sekwencji
print len(L)              # funkcja zwraca liczbę elementów listy
print L + L2 + [1]        # listy można łączyc ze sobą
print 3 * L               # listy można powielać
print L[2]                # odwołanie do elementu listy za pomoca indeksu
print L[0:2]              # odwołanie do elementów listy o indeksie 0 i 1
                          # (wycinek)
L[1] = 1000               # nadpisanie elementu listy znajdującego się
                          # pod indeksem 1 wartością 1000
L[1:3] = [0]              # nadpisanie wycinka
L_k = list(L)             # kopiowanie listy
L_k = L[:]
print 1 in L              # czy element znajduje się na liście (bool)
print 1 not in L          # czy element nie znajduje się na liście (bool)
print range(6)            # buduje listę od 0 do 5
print range(2, 6)         # buduje listę od 2 do 5
print range(1, 6, 2)      # buduje listę od 1 do 5 z krokiem iteracji = 2
del L2[1]                 # usuwanie elementu z listy o zadanym indeksie
del L[0:2]                # usuwanie wycinka listy o zadanym zakresie
del L                     # usuwanie calej listy
```

2.4. Krotki

Krotki w Pythonie występują jako uporządkowane ciąci obiektów. Stosujemy je wtedy, gdy nie chcielibyśmy zmieniać ich struktury (w przeciwieństwie do list). Do elementów krotek można odwoływać się za pomocą indeksu [3]. Praktyczne ich zastosowanie przedstawione jest na listingu 2.5.

Listing 2.5. Krotki.

```
# -*- coding: cp1250 -*-
# krotki w Pythonie

T = ()                  # utworzenie pustej krotki
T = (1,)                # utworzenie krotki z jedną składową
T = (0, 1, 2, 3, 4)     # utworzenie krotki z 5 skladowymi
T = 0, 1, 2, 3, 4       # jeżeli ilośc składowych > 1 nawiasy nie są
                        # obowiązkowe
T = tuple(range(0, 5))  # utworzenie krotki z sekwencji

print T[4]              # dostęp do składowej krotki przy pomocy indeksu
print T[1:4]            # wycinek elementów zawierających indeksy
                        # od 1 do 3 krotki macierzystej
print len(T)            # rozmiar krotki
print T + (10, 11, 12)  # konkatenacja krotek
print 4 * T             # powielanie krotek
print 100 in T          # czy element znajduje się w krotce (bool)
print 100 not in T      # czy element nie znajduje się w krotce (bool)
a, b = 0, 1             # a = 0, b = 1, rozpakowanie krotki
a, b = b, a             # a = 1, b = 0, zamiana wartości zmiennych
del T                   # usuwanie krotki
```

2.5. Słowniki

Słowniki w Pythonie to pewnego rodzaju tablice asocjacyjne. Do elementów słownika mamy dostęp przy pomocy klucza, który może być praktycznie dowolnego typu niezmiennego (np. liczby, stringi, krotki) i nie może się powtarzać. Zawartość słownika można modyfikować. Słowniki obsługują zagnieżdżenia obiektów na dowolną głębokość [3]. Przykład przedstawiono poniżej (listing 2.6).

Listing 2.6. Słowniki.

```
# -*- coding: cp1250 -*-
# słowniki w Pythonie

D = {}                          # utworzenie pustego słownika
D = {"I": 1, "II": 2, "V": 5}   # słownik o 3 elementach (klucz - wartośc)
print len(D)                    # liczba elementów klucz - wartośc
D["III"] = 3                    # dodanie pozycji do słownika o nowym
                                # kluczu
print D["V"]                    # dostęp do wartości przy pomocy klucza
D_k = dict(D)                   # kopiowanie słownika
print "VI" in D_k               # czy klucz istnieje w słowniku (bool)
print "VI" not in D_k           # czy klucz nie istnieje w słowniku (bool)
del D_k["I"]                    # usuwanie podanego klucza ze słownika
del D_k                         # usuwanie całego słownika
```

2.6. Zbiory

Zbiory w pewnym sensie przypominają słowniki. Nie ma tutaj konstrukcji typu klucz -wartość tak jak w słownikach, tylko pojedyncze elementy, a ich kolejność nie jest z góry ustalona. Zbiory zawierają kolekcje niepowtarzalnych elementów, nie wspierają dostępu do elementów za pomocą indeksów [3]. Przykładowe operacje na zbiorach przedstawia listing 2.7.

Listing 2.7. Zbiory.

```
# -*- coding: cp1250 -*-
# zbiory w Pythonie

S_a = set([0, 1, 2, 3, 3, 4])    # utworzenie zbioru z sekwencji
S_b = set([3, 2, 1])

print 3 in S_a                   # czy element nalży do zbioru (bool)
print 3 not in S_a               # czy element nie należy do zbioru (bool)
print S_a <= S_b                 # czy zbiórA zawiera sie w zbiórB (bool)
print S_a >= S_b                 # czy zbiórB zawiera sie w zbiórA (bool)
print S_a | S_b                  # suma zbiorów
print S_a & S_b                  # iloczyn zbiorów
print S_a - S_b                  # różnica zbiorów
print S_a ^ S_b                  # różnica symetryczna zbiorów
S_a = S_b.copy()                 # kopiowanie zbiorów
```

2.7. Instrukcja warunkowa

Instrukcja warunkowa pozwala zmienić przebieg algorytmu w zależności od zaistniałych warunków logicznych. W Pythonie mamy do czynienia z instrukcją *if* lub kombinacją *if, elif* oraz *else*. Instrukcja *switch* nie jest wspierana, można ją bez problemu zastąpić opisaną w tym podrozdziale konstrukcją *if, elif, else*. Przykład został przedstawiony w listingu 2.8

Listing 2.8. Instrukcja warunkowa.

```
# -*- coding: cp1250 -*-
# instrukcja warunkowa w Pythonie

# przykład zastosowania instrukcji warunkowej - if, elif, else
x = 17
if x % 2 == 0:
    print x, " jest parzysta!"
else:
    print x, " jest nieparzysta!"

if x > 0:
    print x, " jest dodatnia!"
elif x < 0:
    print x, " jest ujemna!"
else:
    print x, " jest zerem!"

# przykład zastosowania wyrażenia trójargumentowego, A if B else C
# operatory logiczne: or (alternatywa), and (koniunkcja)
```

```
prawda_czy_falsz = (True or False)
print "Wynik zapytania: ",
print "Jednak prawda!" if prawda_czy_falsz else "Jednak fałsz!"
```

W języku Python instrukcje blokowe powinny zawierać wcięcia. Należy stosować tabulator lub spację, przy czym nie powinno się mieszać tych znaków. Programista powinien przyjąć jedną konwencję i ją konsekwentnie stosować.

2.8. Pętle programowe

Trudno sobie wyobrazić język programowania wysokiego poziomu bez pętli. Język Python wyposażony jest w dwie pętle programowe: *for* i *while*. W pętlach można stosować instrukcje *break* i *continue*. Przykład demonstruje listing 2.9.

Listing 2.9. Pętle programowe.

```
# -*- coding: cp1250 -*-
# pętle w Pythonie

L = list(range(10, 90, 10))
S = set(range(1, 11))

# pętla for (przykład iteracji po indeksach)
for i, x in enumerate(L):
    print "lista[", i, "] = ", x

# przykład iteracji po elementach
for y in S:
    for x in S:
        print x * y, "\t",
    print

# petla while (przykład odwracania elementów w liście)
i = 0
j = len(L) - 1

while i < j:
    L[i], L[j] = L[j], L[i]
    i += 1
    j -= 1
```

2.9. Standardowe wejście i wyjście

Standardowe wejście i wyjście pozwala na interakcję z użytkownikiem. Poprzez standardowe wejście (w programie reprezentowane jest przez metodę *raw_input*) użytkownik może dostarczyć potrzebne dane do programu, a za pomocą standardowego wyjścia (w programie reprezentowane jest przez *print*) może analizować wyniki. Przykład został umieszczony w listingu 2.10.

Listing 2.10. Standardowe wejście / wyjście.

```
# -*- coding: cp1250 -*-
# standardowe wejście / wyjście w Pythonie
```

```
# metody odpowiedzialne za odczytywanie napisów
# wprowadzanych na standardowe wejście

napis = raw_input("Podaj napis: ")
cyfra = int(raw_input("Podaj cyfre: "))       # zastosowanie rzutowania

# niektóre sposoby formatowania tekstu
print "Podano napis %s o liczbie znakow %d!" % (napis, len(napis))
print "Wprowadzona cyfra to %d!" % cyfra
```

2.10. Funkcje

Funkcja jest podprogramem, która wykonuje pewne operacje. Może, ale nie musi przyjmować argumenty wejściowe. Zwraca *None* lub coś bardziej przydatnego. Tworzenie funkcji ma na celu uniknąć niepotrzebnego powtarzania się kodu w programie. W Pythonie mamy do dyspozycji funkcje wbudowane, anonimowe, również istnieje możliwość implementacji funkcji przez samego programistę. Przykłady zostały opisane w listingu 2.11.

Listing 2.11. Funkcje.

```
# -*- coding: cp1250 -*-
# funkcje w Pythonie

# przykład funkcji zwracającej wynik
def srednia_arytmetyczna(dane):
    srednia = 0.0

    for d in dane:
        srednia += d

    return srednia / float(len(dane))

# przykład funkcji anonimowej (lambda)
# ciałem musi byc pojedyncze wyrażenie, nie może zawierać print czy return
srednia_arytmetyczna_lambda = lambda dane: sum(dane) / float(len(dane))

# przykład funkcji nie zwracąjacej wyniku
def rysuj_szachownice(bok_szachownicy = 0, bok_pola = 0, grafika = "o"):
    pola = list([bok_pola * " ", bok_pola * grafika])

    i = 0
    while i < bok_szachownicy * bok_pola:
        j = 0
        while j < bok_szachownicy:
            print pola[j%2],
            j += 1
        i += 1

        if i % bok_pola == 0:
            pola[0], pola[1] = pola[1], pola[0]
        print
    print

# wywołania funkcji
```

```
print srednia_arytmetyczna([1, 2, 3, 4, 5, 6])
print srednia_arytmetyczna_lambda([1, 2, 3, 4, 5, 6])
rysuj_szachownice(10, 2)
```

2.11. Moduły

Moduły to pewnego rodzaju biblioteki. Zaimportowane tworzą przestrzeń nazw, za pomocą której można się odwoływać do zamieszczonych wewnątrz nich definicji funkcji i zmiennych[3]. Sposób implementacji modułu i jego importu przedstawiają listingi 2.12 i 2.13.

Listing 2.12. Przykład własnego modułu.

```
# -*- coding: cp1250 -*-
# własne moduły w Pythonie

# zestawienie definicji kilku prostych funkcji
def dodawanie(a, b):
    return a + b

def odejmowanie(a, b):
    return a - b

def mnozenie(a, b):
    return a * b

def dzielenie(a, b):
    return a / float(b)

# sposób testowania modułu
# dzięki poniższemu warunkowi uruchamiając aktualny plik
# testy będą dostępne i widoczne, ale importując ten oto
# plik testy zostana pominięte
if __name__ == "__main__":
    print "1 + 2 = ", dodawanie(1, 2)
    print "3 - 4 = ", odejmowanie(3, 4)
    print "3 * [ 2, 1 ] = ", mnozenie(3, [2, 1])
    print "7 / 3 = ", dzielenie(7, 3)
```

Listing 2.13. Import modułów.

```
# -*- coding: cp1250 -*-
# importowanie modułów w Pythonie

import mymodule      # można importować własne moduły / listing 12
import sys           # można importować moduły wbudowane (systemowy)
import math          # moduł matematyczny
import random        # moduł odpowiedzialny za generowanie zmiennych
                     # losowych

# przykład dostępu do metod z modułu

print mymodule.dodawanie(5, 6)
print mymodule.mnozenie(1.5, 4)
```

```
# przykłady dostępu do metod modułów wbudowanych
print random.random()
print sys.version
print math.sin(math.pi/2)
```

2.12. Klasy

Programowanie zorientowane obiektowo jest jedną z najlepszych technik programowania. Dzięki niej programista może opisywać rzeczywistość przy pomocy obiektów. W Pythonie obiekty tworzy się przy pomocy klas. Klasy mogą być rozbudowywane na podstawie klas podstawowych (dziedziczenie) lub być składową innej klasy (kompozycja), a przeciążanie operatorów pozwala programiście na to, aby obiekty zachowywały się tak jak typy wbudowane [3]. Przykład dziedziczenia został przedstawiony w listingu 2.14.

Listing 2.14. Klasy.

```
# -*- coding: cp1250 -*-
# klasy i dziedziczenie w Pythonie

class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

class Czworokat:
    # podajemy lewy dolny i prawy górny róg czworokąta
    def __init__(self, x1 = 0, y1 = 0, x2 = 0, y2 = 0):
        # kompozycja w klasie Czworokat
        self.pt1 = Point(x1, y1)
        self.pt2 = Point(x2, y2)
    def __str__(self):
        return "Czworokat[%s, %s]" % (self.pt1, self.pt2)

class Kwadrat(Czworokat):
    # kwadrat to czworokąt o równych bokach
    def __init__(self, x1 = 0, y1 = 0, x2 = 0, y2 = 0):
        Czworokat.__init__(self, x1, y1, x2, y2)
        # sprawdzenie równości boków
        if (x2-x1) != (y2-y1):
            raise Exception("różne boki")
    def __str__(self):
        return "Kwadrat[%s, %s]" % (self.pt1, self.pt2)

# utworzenie obiektu
kwadrat = Kwadrat(0, 0, 6, 6)
```

2.13. Wyjątki

Czasami podczas działania programu zdarzają się sytuacje wyjątkowe, które należy wykryć i odpowiednio obsłużyć. Język Python posiada mechanizm obsługi wyjątków dostosowany do takich sytuacji [3]. Przykłady obrazuje listing 2.15.

Listing 2.15. Wyjątki.

```
# -*- coding: cp1250 -*-
# wyjątki w Pythonie

# przechwytywanie wyjątków (podstawy)
try:
    # dzielenie przez 0 (wyjatek: ZeroDivisionError)
    1/0
except ZeroDivisionError:
    print "Wyjątek: Dzielenie przez 0!"
else:
    print "Wyjątek nie wystąpił!"
finally:
    print "Tutaj zawsze sie wykona."
print "Dalsze instrukcje..."

# wyjątki jako klasy (przykład), klasa dziedziczy po Exception
class MyError(Exception):
    def __init__(self, message):
        self.message = message
    def __str__(self):
        return "Wyjatek: " + str(self.message)

# zgłaszanie wyjątku
raise MyError("Wystąpił wyjątek...")
```

2.14. Uzyskiwanie pomocy w Pythonie

Python wyposażony jest w mechanizm uzyskiwania pomocy na temat konkretnych typów czy sposobu działania metod, których programista chciałby użyć do tworzenia swoich aplikacji. Przykłady takiego dostępu do pomocy w Pythonie przedstawione są w listingu 2.16.

Listing 2.16. Pomoc w Pythonie.

```
# -*- coding: cp1250 -*-
# uzyskiwanie pomocy w Pythonie

print dir(int)      # zestaw metod dla typów całkowitych
print dir(float)    # zestaw metod dla typów zmiennoprzecinkowych
print dir(complex)  # zestaw metod dla typów zespolonych
print dir(str)      # zestaw metod dla stringów
print dir(bool)     # zestwa metod dla typu logicznego
print dir(list)     # zestaw metod dla list
print dir(tuple)    # zestaw metod dla krotek
print dir(dict)     # zestaw metod dla słowników
print dir(set)      # zestaw metod dla zbiorów

# sposób pozyskiwania pomocy w Pythonie dla konkretnych
# typów i atrybutów typów

help(int)
help(float)
help(complex)
help(str)
```

```
help(str.lower)
help(bool)
help(list)
help(list.sort)
help(tuple)
help(dict)
help(set)
```

3. Teoria grup

Teoria grup bada struktury algebraiczne zwane grupami. Pojęcie grupy pojawiło się po raz pierwszy w badaniach Galois nad rozwiązalnością równań algebraicznych. Definicję abstrakcyjnego pojęcia grupy zawdzięczamy Cayleyowi (1854). Od tego czasu pojęcia teorii grup przeniknęły do wielu działów matematyki, fizyki, czy chemii.

3.1. Grupy abstrakcyjne

Ważnymi klasami grup są grupy permutacji, grupy macierzy, czy grupy transformacji. Jednak często abstrahujemy od natury elementów grupy i rozważamy grupy abstrakcyjne, jako zbiory elementów spełniających określone postulaty.

3.1.1. Grupa

Grupa G jest strukturą algebraiczną składającą się z niepustego zbioru G oraz ustalonego działania $(\cdot)$. Działanie powinno spełniać następujące warunki [8]:

1. **Łączności:** Jeżeli $a, b, c \in G$, to $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
2. **Neutralności:** Dla pewnego elementu jednostkowego $e \in G$ i $a \in G$ zachodzi $a \cdot e = e \cdot a = a$. Dowodzi się, że element jednostkowy jest dokładnie jeden w grupie.
3. **Odwrotności:** Dla każdego $a \in G$ istnieje element $a^{-1} \in G$, taki że $a \cdot a^{-1} = a^{-1} \cdot a = e$.

3.1.2. Grupa abelowa

Grupa abelowa jest strukturą algebraiczną, w której oprócz cech *łączności, neutralności i odwrotności* zachodzi warunek *przemienności działania* wewnątrz grupy, $a \cdot b = b \cdot a$ dla każdego $a, b \in G$ [8].

3.1.3. Rząd grupy

Rząd grupy G określamy jako liczbę n elementów grupy G, zakładając że mamy do czynienia z grupą skończoną, $n < \infty$ [8]. W niniejszej pracy będziemy rozważać wyłącznie grupy skończone.

3.1.4. Podgrupa

Niech G będzie grupą, H podzbiorem niepustym G. H jest podgrupą grupy G (oznaczenie $H \leqslant G$), jeżeli H jest grupą względem tego samego działania, które istnieje w G. Jeżeli S jest podzbiorem G, to $\langle S \rangle$ oznacza podgrupę generowaną przez S.

3.1.5. Warstwy

Warstwą nazywamy zbiór, który powstał w wyniku iloczynu elementu $a \in G$ z każdym elementem podgrupy H. Rozróżniamy dwa rodzaje warstw [8]:

1. **Warstwa lewostronna:** zbiór elementów postaci ah, gdzie $h \in H$; zbiór ten oznaczamy przez aH.
2. **Warstwa prawostronna:** zbiór elementów postaci ha, gdzie $h \in H$; zbiór ten oznaczamy przez Ha.

3.1.6. Podgrupa niezmiennicza

Podgrupą niezmienniczą (inwariantną, normalną) nazywamy podgrupę H grupy G, której warstwy lewostronne i prawostronne są takie same, a mianowicie musi zachodzić zależność $aH = Ha$, oraz $a^{-1}H = Ha^{-1}$ dla każdego $a \in G$.

3.1.7. Twierdzenie Lagrange'a

Rząd podgrupy H jest podzielnikiem rzędu grupy G.

3.1.8. Klasy elementów sprzężonych

W grupie G określamy relację sprzężenia $\sim$ następująco: $a \sim b$ wtedy i tylko wtedy, gdy istnieje $c \in G$ takie, że $cbc^{-1} = a$. Relacja $\sim$ jest zwrotna, symetryczna i przechodnia, więc jest *relacją równoważności.* W grupie G możemy więc wyróżnić *rozłączne* klasy równoważności, klasy elementów sprzężonych.

3.2. Grupy permutacji

Grupy permutacji pełnią w matematyce szczególną rolę. Pojawiają się w bardzo różnorodnych działach matematyki; są z jednej strony niemal najprostrzymi grupami nieprzemiennymi, ale też są wystarczająco złożone, aby zawierać wszystkie grupy skończone.

3.2.1. Definicja permutacji

Elementy zbioru $A = \{1, 2, 3\}$ można przestawiać na $3! = 6$ różnych sposobów: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1). Każde takie uporządkowanie nazywamy permutacją elementów zbioru A [4].

3.2.2. Grupa symetryczna

Grupa symetryczna [oznaczenie S_n lub $\mathrm{Sym}(\Omega)$] jest grupą wszystkich permutacji zbioru n elementowego Ω, na ogół zbioru liczb od 1 do n. Każdą podgrupę grupy symetrycznej nazywamy *grupą permutacji.* Elementy takiej grupy zapisywane są w następującej postaci:

$$P = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}. \tag{3.1}$$

Przykład 3.1 przedstawia element grupy S_3, której rząd wynosi $n! = 3! = 6$. Porządek kolumn w tym zapisie nie odgrywa żadnej roli.

3.2.3. Grupa alternująca

Grupa alternująca [oznaczenie A_n lub $\mathrm{Alt}(\Omega)$] jest podgrupą permutacji parzystych z grupy symetrycznej.

Przykład 3.2 przedstawia permutację parzystą z A_4 z rozkładem na cykle:

$$Q = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix} = (14)(23). \tag{3.2}$$

3.2.4. Twierdzenie Cayleya

Każda grupa skończona G rzędu n jest *izomorficzna* z pewną podgrupą grupy symetrycznej S_n. Do przeprowadzenia dowodu należy dokonać przyporządkowania następujących elementów [8] $(a_i, a_j \in G)$:

$$a_i \to Pa_i = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ a_i a_1 & a_i a_2 & \dots & a_i a_n \end{pmatrix}, \tag{3.3}$$

$$a_j \to Pa_j = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ a_j a_1 & a_j a_2 & \dots & a_j a_n \end{pmatrix}. \tag{3.4}$$

Dla $a_i a_j \in G$ otrzymamy:

$$a_i a_j \to Pa_i Pa_j = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ a_i a_j a_1 & a_i a_j a_2 & \dots & a_i a_j a_n \end{pmatrix}. \tag{3.5}$$

Aby przeprowadzić dowód *izomorfizmu* doprowadza się do relacji

$$Pa_i Pa_j = Pa_i a_j. \tag{3.6}$$

Należy zwrócić uwagę na sytuacje, w której przestawienie szyku elementów i ich indeksów nie tworzy nowej grupy. Sytuacje demonstruje relacja 3.7:

$$\begin{pmatrix} a_1 & a_2 & \dots & a_n \\ a_i a_1 & a_i a_2 & \dots & a_i a_n \end{pmatrix} = \begin{pmatrix} a_1 & a_n & \dots & a_2 \\ a_i a_1 & a_i a_n & \dots & a_i a_2 \end{pmatrix}. \tag{3.7}$$

Można zapisać:

$$\begin{aligned} Pa_i &= \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ a_i a_1 & a_i a_2 & \dots & a_i a_n \end{pmatrix} = \begin{pmatrix} a_j a_1 & a_j a_2 & \dots & a_j a_n \\ a_i(a_j a_1) & a_i(a_j a_2) & \dots & a_i(a_j a_n) \end{pmatrix} = \\ &= \begin{pmatrix} a_j a_1 & a_j a_2 & \dots & a_j a_n \\ a_i a_j a_1 & a_i a_j a_2 & \dots & a_i a_j a_n \end{pmatrix}. \end{aligned} \tag{3.8}$$

Relacja składania Pa_i oraz Pa_j ma następującą postać:

$$\begin{aligned} Pa_i Pa_j &= \begin{pmatrix} a_j a_1 & a_j a_2 & \dots & a_j a_n \\ a_i a_j a_1 & a_i a_j a_2 & \dots & a_i a_j a_n \end{pmatrix} \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ a_j a_1 & a_j a_2 & \dots & a_j a_n \end{pmatrix} = \\ &= \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ a_i a_j a_1 & a_i a_j a_2 & \dots & a_i a_j a_n \end{pmatrix} = Pa_i a_j. \end{aligned} \tag{3.9}$$

3.2.5. Cykle

Cykl jest specyficzną strukturą, którą można sobie wyobrazić jako ciąg kolejnych przestawień składników grupy od pewnego elementu, który zamyka i otwiera ciąg zastępujących się elementów [8].

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix} = (16)(25)(34). \tag{3.10}$$

Przykład 3.10 demonstruje rozbicie permutacji na trzy rozłączne cykle o długości 2. Cykl (16) odczytujemy w następujący sposób: 1 zastępujemy 6, a 6 zastępujemy 1. Element 1 otwiera i zamyka cykl. Cykl o długości 2 nazywamy *transpozycją*. Cykle jednoelementowe zwykle pomijamy w zapisie.

3.2.6. Znak permutacji

Znak permutacji jest to liczba $(-1)^N$, gdzie N jest liczbą transpozycji w rozkładzie danej permutacji. Jeżeli N jest parzyste to *znak permutacji* przyjmuje wartość $+1$, a permutację nazywamy *parzystą*. Jeżeli N jest nieparzyste to znak permutacji przyjmuje wartość -1, a permutację nazywamy nieparzystą [8]. Czasem definiuje się parzystość równą 0 dla permutacji parzystej i 1 dla permutacji nieparzystej. Rozkład permutacji na transpozycje nie jest jednoznaczny, ale parzystość liczby N jest jednoznacznie określona.

Przykład permutacji nieparzystej:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 4 & 1 & 2 & 5 & 3 \end{pmatrix} = (163)(24)(5) = (13)(16)(24). \tag{3.11}$$

W rozkładzie permutacji 3.11 występują 3 transpozycje.

Zamiana cyklu k elementowego na iloczyn transpozycji możemy robić według wzoru:

$$(a_1, a_2, a_3 \cdots a_k) = (a_1, a_k)(a_1, a_{k-1}) \cdots (a_1, a_3)(a_1, a_2). \tag{3.12}$$

Ze wzoru 3.12 widać, że cykl k elementowy ma znak $(-1)^{k+1}$.

3.2.7. Centralizator

Niech G będzie grupą oraz $a \in G$. *Centralizator* elementu a w grupie G jest to zbiór $C_G(a) = \{g \in G : ga = ag\}$. Jeżeli $A \subset G$, to można określić $C_G(A) = \{g \in G : ga = ag$ dla każdego $a \in A\}$. Centralizator jest podgrupą grupy G.

Centrum grupy G jest to zbiór $Z(G) = C_G(G)$. Centrum grupy G jest to zbiór elementów przemiennych z każdym elementem grupy G. Centrum grupy $Z(G)$ jest zawsze podgrupą normalną grupy G.

3.2.8. Komutant

Załóżmy, że mamy daną grupę G oraz podzbiory A, B. Komutantem nazywamy zbiór $[A, B]$ generowany przez zbiór komutatorów $\{[a, b] = aba^{-1}b^{-1} : a \in A, b \in B\}$. Można zdefiniować pochodną grupy następująco: $G^{(0)} = G$, $G^{(n+1)} = [G^{(n)}, G^{(n)}]$. Komutant $[G, G]$ grupy G jest jej podgrupą normalną.

Jeżeli grupa $[G, G]$ jest trywialna, to G jest abelowa. Jeżeli $[G, G] = G$, to grupa G jest *doskonała*. Najmniejsza nietrywialna grupa doskonała to A_5.

3.2.9. Iloczyn prosty grup

Niech G i H będą grupami. Iloczynem prostym grup G i H nazywamy zbiór $G \times H$ wszystkich par uporządkowanych $\{(g, h) : g \in G, h \in H\}$ z działaniem $(g_1, h_1) \cdot (g_2, h_2) = (g_1 \cdot g_2, h_1 \cdot h_2)$ [12].

3.2.10. Działanie grupy na zbiorze

Mamy daną grupę G oraz zbiór Ω. Działaniem grupy na zbiorze nazywamy funkcję $F : G \times \Omega \longrightarrow \Omega$ o własnościach:

— $F(e, \omega) = \omega$ dla każdego $\omega \in \Omega$.
— $F(gh, \omega) = F(g, F(h, \omega))$ dla każdego $g, h \in G$, $\omega \in \Omega$.

3.2.11. Orbita

Orbitą nazywamy zbiór $F(G, \omega) = \{F(g, \omega) : g \in G\}$, który jest podzbiorem Ω. Działanie grupy jest tranzytywne, jeżeli $F(G, \omega) = \Omega$ dla każdego $\omega \in \Omega$, czyli istnieje tylko jedna orbita.

3.2.12. Stabilizator

Stabilizator jest to zbiór $\mathrm{Stab}_G(\omega) = \{g \in G : F(g, \omega) = \omega\}$. Stabilizator jest podgrupą grupy G. Jeżeli ω_1, $\omega_2 \in \Omega$, to

$$\mathrm{Stab}_G(\omega_1, \omega_2) = \{g \in G : F(g, \omega_1) = \omega_1, F(g, \omega_2) = \omega_2\} = \mathrm{Stab}_G(\omega_1) \cap \mathrm{Stab}_G(\omega_2). \quad (3.13)$$

Analogicznie rozumiemy stabilizator ciągu większej liczby punktów.

3.2.13. Silne generatory

Baza dla grupy $G \leqslant \mathrm{Sym}(\Omega)$, to ciąg punktów $B = (\omega_1, \cdots, \omega_m)$, $\omega_i \in \Omega$, dla którego $\mathrm{Stab}_G(\omega_1, \cdots, \omega_m) = 1$. Baza określa łańcuch podgrup

$$G = G^{[0]} \geqslant G^{[1]} \geqslant \cdots \geqslant G^{[m-1]} \geqslant G^{[m]} = 1, \quad (3.14)$$

gdzie $G^{[1]} = \mathrm{Stab}_G(\omega_1)$, $G^{[2]} = \mathrm{Stab}_G(\omega_1, \omega_2)$, $G^{[i]} = \mathrm{Stab}_G(\omega_1, \cdots, \omega_i)$.

Silny zbiór generujący S związany z bazą B, to zbiór generujący grupę G o tej własności, że

$$\langle S \cap G^{[i]} \rangle = G^{[i]} \text{ dla } 0 \leqslant i \leqslant m. \quad (3.15)$$

Zbiór S nie jest wyznaczony jednoznacznie [10].

3.2.14. Bloki

Jeżeli działanie F grupy G na zbiorze Ω jest tranzytywne, a Δ jest podzbiorem Ω, wtedy Δ nazywamy blokiem dla G, jeżeli dla każdego $g \in G$ zachodzi:

$$F(g, \Delta) = \Delta \text{ lub } F(g, \Delta) \cap \Delta = \emptyset, \quad (3.16)$$

gdzie $F(g, \Delta) = \{F(g, \omega) : \omega \in \Delta\}$.

Jeżeli Δ jest blokiem, to zbiór obrazów Δ tworzy podział zbioru Ω, który jest nazywany układem bloków. Działanie F indukuje działanie F_1 na układzie bloków.

4. Algorytmy i implementacje

W tym rozdziale zostaną przedstawione interfejsy dla permutacji oraz grup permutacji, które pozwalają na wygodne prowadzenie obliczeń komputerowych. Ponadto zostaną przedstawione wybrane algorytmy grup permutacji, dzięki którym obliczenia są szybkie, a przy tym można badać duże obiekty matematyczne.

4.1. Interfejs permutacji

Permutacje będą instancjami klasy Perm. Interfejs permutacji przedstawiony jest w tabelach 4.1 4.2.

Tabela 4.1. Interfejs permutacji [3].

Operacja	**Znaczenie**	**Metoda**
perm = Perm(size)	tworzenie perm	__init__()
perm = Perm(size)(3,4)(4,5)	tworzenie perm	
perm = Perm(3, data=[2,1,0])	tworzenie perm	
repr(perm)	reprezentacja	__repr__()
len(perm)	długość perm	__len__()
perm.is_identity()	czy identyczność	is_identity()
~perm	perm odwrotna	__invert__()
perm * perm2	mnożenie perms	__mul__()
cmp(perm, perm2)	porównywanie	__cmp__()
perm[i]	perm jako funkcja	__getitem__()
pow(perm, n), perm ** n	potęgowanie perms	__pow__()
perm.support()	lista k przesuwanych	support()
perm.max()	największe k	max()
perm.min()	najmniejsze k	min()
perm.list(), perm.list(size)	jako lista	list()
perm.label()	etykieta tekstowa	label()
perm.cycles()	zwraca listę cykli	cycles()
perm.order()	rząd perm	order()
perm.parity()	parzystość (0 lub 1)	parity()
perm.is_even()	czy parzysta	is_even()
perm.is_odd()	czy nieparzysta	is_odd()
perm.sign()	znak perm (+1, lub -1)	sign()
perm.commutes_with(perm2)	komutacja (bool)	commutes_with()
perm.commutator(perm2)	komutator permutacji	commutator()
Perm.random(size)	losowa perm	random()
perm.inversion_vector(size)	wektor inwersji	inversion_vector()

Tabela 4.2. Interfejs permutacji (ciąg dalszy) [3].

perm.rank_lex(size)	leksykograficzne rankowanie	rank_lex()
Perm.unrank_lex(size, rank)	przeciwieństwo rankowania	unrank_lex()
perm.rank_mr(size)	rankowanie Myrvold'a i Ruskey'a	rank_mr()
Perm.unrank_mr(size, rank)	przeciwieństwo rankowania	unrank_mr()
perm.next_lex()	następna permutacja	next_lex()
perm.prev_lex()	poprzednia permutacja	prev_lex()

Klasa Perm zaimplementowana jest w module *perms*. Poniższy zapis sesji interaktywnej Pythona ilustruje korzystanie z wybranych metod interfejsu permutacji.

```
>>> from perms import *
>>> N = 4
>>> a, b, c = Perm(N)(0, 1), Perm(N)(1, 2), Perm(N)(2, 3)
>>> a * b
Perm(4)(0, 1, 2)
>>> b * a
Perm(4)(0, 2, 1)
>>> a.commutes_with(b)
False
>>> a.commutes_with(c)
True
>>> a.is_even()
False
>>> a.commutator(c)
Perm(4)
>>> (a * c).cycles()
[[0, 1], [2, 3]]
>>>
```

4.2. Implementacja permutacji

Do obsługi błędów w klasie *Perm* i klasie *Group* przygotowujemy wyjątek *PermError*.

4.2.1. Klasa Perm - podstawowe metody

Permutacja jest przechowywana wewnętrznie w instancji klasy Perm w atrybucie *data* jako lista różnych liczb całkowitych od 0 do $size - 1$.

Konstruktor klasy Perm (metoda `__init__()`) wymaga podania rozmiaru permutacji (*size*). Opcjonalnie możemy podać listy różnych liczb całkowitych stanowiące zapis permutacji w notacji macierzowej.

Działanie większości metod czytelnie przedstawia sam kod. Porównywanie permutacji (metoda `__cmp__()`) odwołuje się do porównywania list wbudowanego w Pythona. Daje to porównywanie leksykograficzne permutacji.

Metoda `__call__()` pozwala na stworzenie permutacji podanej za pomocą cykli (list lub krotek).

Metoda `list()` pozwala wyeksportować permutacje w postaci listy innej długości niż ta przechowywana w atrybucie *data*.

Metoda `label()` zapisuje permutacje w postaci stringu, przy czym liczby większe od 9 kodowane są za pomocą liter alfabetu. Występuje tu ograniczenie na rozmiar permutacji, związana z ograniczeniem ilości znaków. Etykiety tekstowe wykorzystywane są w podstawowej implementacji grup permutacji jako niezmienne klucze dla obiektu permutacji. W razie potrzeby można określić inne sposoby tworzenia etykiet tekstowych, gdzie nie będzie już ograniczeń na rozmiar permutacji.

4.2.2. Klasa Perm - metody związane z cyklami

Permutacje można zapisać jako iloczyny cykli rozłącznych. Metoda cycles() zwraca listę cykli o długości powyżej 1. W kodzie metoda cycles() jest wykorzystana w kilku różnych punktach [13]:

1. Reprezentacja permutacji jako string wykorzystuje listę cykli, aby dostać zwarty zapis.
2. Obliczanie rzędu permutacji zawiera rozkład na cykle i sukcesywne korzystanie z funkcji lcm(), która oblicza najmniejszą wspólną wielokrotność. Warto zwrócić uwagę, że kod metody order() jest bardzo zwięzły dzięki skorzystaniu z wbudowanej funkcji reduce(). Dla porównania można podać naiwną implementację wymagającą liczby operacji rzędu $O(n \cdot \text{order})$.
3. Potęgowanie permutacji w metodzie __pow__() algorytmem binarnego potęgowania, który nie odwołuje się do struktury cykli i może być wykorzystany ogólnie, np. dla grup macierzy. Ale możemy podać kod, który może być jeszcze szybszy, jeżeli wykorzystamy strukturę cykli.

```
class Perm:
# ... pozostałe metody
    def __pow__(self, n):
        """Znajduje potęgę permutacji."""
        if n == 0:
            return Perm(self.size)
        elif n < 0:
            return pow(~self, -n)
        elif n == 1:
            return self
        elif n == 2:
            return self * self
        else:
            tmp = Perm(self.size)
            for cycle in self.cycles():
                c_len = len(cycle)
                m = n % c_len
                if m > 0:
                    new_cycle = [cycle[(i * m) % c_len] for i in range(c_len)]
                    tmp = tmp * Perm(self.size)(*new_cycle)
        return tmp
```

4.2.3. Klasa Perm - parzystość

Główną metodą związaną z badaniem parzystości permutacji jest metoda parity(). Zwraca ona zero dla permutacji parzystej oraz jeden dla permutacji nieparzystej. Za pomocą tej metody tworzymy dalsze wygodne metody zwracające wartości typu logicznego (is_even(), is_odd()) lub znak permutacji (sign()). W kodzie metody parity() wyznaczamy liczbę wszystkich cykli rozłącznych występujących w rozkładzie danej permutacji, łącznie z cyklami o długości jeden. Można pokazać, że parzystość różnicy rozmiaru permutacji i liczby cykli jest równa parzystości liczby transpozycji występujących w rozkładzie permutacji na transpozycje [13].

4.2.4. Klasa Perm - komutatory i permutacje przypadkowe

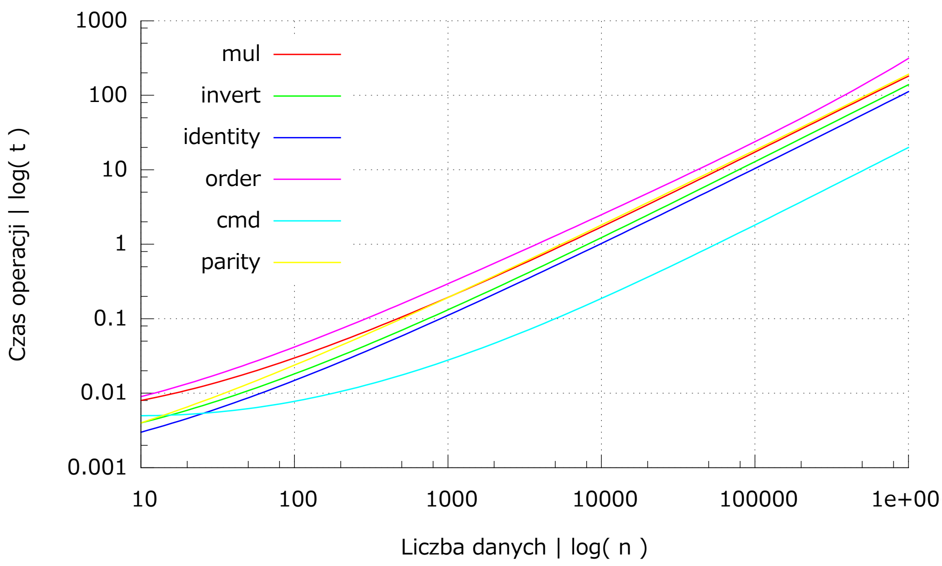
Klasa Perm dostarcza prostych metod do badania przemienności. Można sprawdzić przemienność dwóch permutacji (commutes_with()) lub obliczać komutator dwóch permutacji (commutator()). Mamy również metodę klasy, która pozwala otrzymać permutację przypadkową o danym rozkładzie n. Wykorzystywany jest generator losowy zawarty w metodzie random() [13]. Generalnie dla permutacji przypadkowych kluczowym zagadnieniem jest udowodnienie, że dana metoda pozwala wylosować dowolną permutacje z jednakowym prawdopodobieństwem $1/n!$

4.2.5. Klasa Perm - rangi permutacji

Funkcja rankująca dla n-elementowej permutacji przypisuje unikatowy numer z zakresu od 0 do $n! - 1$ dla każdej $n!$ permutacji. *Unrank* jest funkcją odwrotną do rankowania. Algorytm rankowania ustawia permutacje w porządku leksykograficznym, wykorzystuje wektor inwersji, jego wydajność jest rzędu $O(n^2)$. Wektor inwersji składa się z elementów, których wartość wskazuje liczbę elementów, które są mniejsze od niej i leżą po jej prawej stronie. W 2001 roku Myrvold i Ruskey przedstawili prosty algorytm rankowania i unrankowania permutacji, którego wydajność jest rzędu $O(n)$. Algorytm wykorzystuje pomysł inspirowany standardowym generatorem permutacji losowych. Algorytmy Myrvold'a i Ruskey'a zostały przedstawione w metodach rank_mr() oraz unrank_mr() [13].

4.2.6. Testy wydajnościowe dla permutacji

W celu sprawdzenia, czy implementacja ma wydajność przewidywaną przez teorię, przeprowadzono testy komputerowe z modułem *timeit*. Na wykresie widać, że podstawowe operacje mają złożoność czasową $O(n)$.



Rysunek 4.1. Testy dla klasy Perm

4.3. Interfejs grup permutacji

W pracy będą przedstawione dwie implementacje grup permutacji, ale interfejs będzie wspólny, zawarty w tabeli 4.3.

Tabela 4.3. Interfejs grup permutacji [3].

Operacja	**Znaczenie**	**Metoda**
G = Group(size)	tworzenie grupy	__init__()
G.order()	rząd grupy	order()
G * H	iloczyn prosty	__mul__()
G.is_trivial()	czy grupa trywialna	is_trivial()
perm **in** G	przynależność do grupy	__contains__()
G.insert(perm)	generowanie z perm	insert()
G.iterperms()	iterator po perms	iterperms()
G.iterlabels()	iterator po etykietach	iterlabels()
G.is_abelian()	czy grupa abelowa	is_abelian()
G.base()	baza grupy	base()
H.is_subgroup(G)	czy grupa jest podgrupą	is_subgroup()
H.is_normal(G)	czy H jest podgrupą normalną G	is_normal()
G.subgroup_search(prop)	zwraca podgrupę	subgroup_serch()
G.normalizer(H)	normalizator H w G	normalizer()
G.centralizer(H)	centralizator H w G	centralizer()
G.center()	zwraca centrum	center()
G.orbits(points)	zwraca listę orbit	orbits()
G.is_transitive()	czy grupa tranzytywna	is_transitive()
G.stabilizer(point)	zwraca stabilizator	stabilizer()
G.normal_closure(H)	zwraca domknięcie normalne	normal_closure()
G.commutator(H,K)	zwraca komutant podgrup	commutator()
G.derived_subgroup()	zwraca komutant $[G, G]$	derived_subgroup()
G.action(points)	zwraca grupę indukowaną	action()
G.blocks()	zwraca układ bloków (nie zaimplementowane)	blocks()

Podany zapis sesji interaktywnej pokazuje użycie interfejsu grup permutacji podczas badania podgrup grupy symetrycznej S_4.

```
>>> from groups import *
>>> N = 4
>>> s4 = Group(N)
>>> s4.insert(Perm(N)(0,1))
>>> s4.insert(Perm(N)(0,1,2,3))
>>> s4.order()
24
>>> [p.order() for p in s4.iterperms()]
[3, 4, 4, 3, 2, 3, 2, 3, 1, 2, 2, 2, 4, 2, 2, 4, 3, 4, 2, 3, 4, 3, 2, 3]
>>> a4 = Group(N)
>>> a4.insert(Perm(N)(0,1,2))
>>> a4.insert(Perm(N)(1,2,3))
>>> a4.order()
12
>>> all(p.is_even() for p in a4.iterperms())
```

```
True
>>> a4.is_subgroup(s4)
True
>>> a4.is_normal(s4)
True
>>> c4 = Group(N)
>>> c4.insert(Perm(N)(0,1,2,3))
>>> c4.order()
4
>>> c4.is_subgroup(s4)
True
>>> c4.is_normal(s4)
True
>>> c4.is_abelian()
True
>>> a4 = s4.derived_subgroup()
>>> a4.order()
12
>>> all(p.is_even() for p in a4.iterperms())
True
>>> v4 = a4.derived_subgroup()
>>> v4.order()
4
>>> v4.is_abelian()
True
>>> v4comm = v4.derived_subgroup()
>>> v4comm.is_trivial()
True
>>>
```

4.4. Implementacja grup permutacji

Przedstawiony interfejs grup permutacji częściowo pokrywa się z interfejsem występującym w module *SymPy*. W kilku metodach występują uproszczenia, które nie powodują zmniejszenia funkcjonalności, np. `centralizer()`. Korzystano również z rozwiązań występujących w programie GAP.

Implementacja zaawansowana grup permutacji bazuje na artykule Knutha [10]. Niech $P(k)$ oznacza zbiór permutacji, które nie przesuwają elementów większych od k. Dla $0 \leqslant j \leqslant k$, albo $\sigma_{kj} = \emptyset$, albo σ_{kj} jest permutacją z $P(k)$, która przesuwa k na j. Zakładamy, że σ_{kk} jest permutacją identycznościową. Niech $S(k)$ będzie niepustym zbiorem permutacji σ_{kj}, przy czym $S(0)$ zawiera tylko identyczność.

Niech $R(k)$ będzie zbiorem wszystkich permutacji, które mogą być zapisane jako iloczyny $\sigma_k \ldots \sigma_0$, gdzie σ_i należy do $S(i)$.

Niech $T(k)$ będzie podzbiorem $P(k)$ o tej własności, że każdy element ze zbioru $R(k)$ może być zapisany jako iloczyn elementów ze zbioru $\cup_{s=0}^{k} T(s)$. Zakładamy, że elementami $T(k)$ nie są elementy z $P(k-1)$.

Mówimy, że struktura jest aktualna do rzędu n, jeżeli $T(k)$ jest podzbiorem $R(k)$ i jeżeli $R(k)$ zamknięte ze względu na mnożenie, dla $0 \leqslant k \leqslant n$. Wtedy permutacje $\cup_{k=0}^{n} S(k)$ tworzą *układ równoległy* dla $R(n)$, a permutacje $\cup_{k=0}^{n} T(k)$ tworzą *zbiór silnych generatorów* dla $R(n)$. W naszej implementacji $R(n)$ jest badaną grupą permutacji, dla której wyznaczamy i przechowujemy zbiory permutacji $T(k)$ i $S(k)$.

4.4.1. Klasa Group - podstawowe metody

Konstruktor klasy *Group* wymaga podania rozmiaru permutacji należących do grupy. W implementacji podstawowej tworzony jest wewnętrzny słownik ze wstawioną permutacją identycznościową, czyli tworzymy grupę trywialną. W implementacji zaawansowanej przygotowana jest struktura danych składająca się z listy silnych generatorów all_T oraz listy list Sigma do przechowywania permutacji σ_{kj}. Wstawiane są permutacje identycznościowe na pozycje σ_{kk}. Odpowiada to również tworzeniu grupy trywialnej.

Metoda order() zwraca rząd grupy. W implementacji podstawowej rząd grupy jest równoważny liczbie kluczy słownika wewnętrznego. W implementacji zaawansowanej rząd grupy jest obliczany jako iloczyn liczb permutacji σ_{kj} dla wszystkich k. Szacowana złożoność czasowa metody order() dla implementacji zaawansowanej wynosi $O(n^2)$ (mamy pętlę po k i zagnieżdżoną pętlę po σ_{kj}).

Metoda __contains__() zwraca wartość bool, która mówi czy dana permutacja należy do grupy. W implementacji podstawowej wystarczy sprawdzić, czy etykieta tekstowa permutacji jest kluczem w słowniku wewnętrznym. W implementacji zaawansowanej wykonywana jest próba rozkładu danej permutacji na permutacje σ_{kj}. Jeżeli próba się powiedzie, to permutacja już jest zawarta w grupie.

Metoda is_abelian() sprawdza, czy grupa jest abelowa. W implementacji podstawowej sprawdzana jest przemienność wszystkich elementów grupy z innymi elementami grupy [bardzo nieefektywna metoda rzędu $O(n|G|^2)$]. W implementacji zaawansowanej wystarczy sprawdzić, czy silne generatory grupy nawzajem komutują. Korzystamy z faktu, że każdy element grupy może być zapisany jako skończony iloczyn silnych generatorów grupy [13]. Szacowana złożoność czasowa wynosi $O(n|S_G|^2)$, gdzie $|S_G|$ oznacza liczbę silnych generatorów.

Metoda base() zwraca bazę dla grupy G, związaną z silnymi generatorami i układem równoległym. Metoda występuje tylko w implementacji zaawansowanej. Dla każdego k sprawdzane jest występowanie permutacji σ_{kj}. Punkt k zaliczany jest do bazy, jeżeli istnieje permutacja σ_{kj} inna niż σ_{kk}. Szacowana złożoność czasowa metody wynosi $O(n^2)$.

4.4.2. Klasa Group - generowanie wszystkich permutacji

W zastosowaniach praktycznych może pojawić się potrzeba wygenerowania wszystkich permutacji z danej grupy. Zwykle nie jest potrzebne, ani możliwe, stworzenie listy wszystkich permutacji, a chodzi o odwiedzenie każdej permutacji po kolei. Bardzo dobrze wpasowuje się tu koncepcja generatora w Pythonie - jest to funkcja zachowująca swój stan i zwracająca dane na żądanie.

Metoda iterperms() zwraca generator permutacji pochodzących z danej grupy. W implementacji podstawowej zwracany jest standardowy generator dla słowników, itervalues(), który generuje permutacje będące wartościami w słowniku wewnętrznym. W implementacji zaawansowanej wykorzystujemy algorytm generowania wszystkich n-krotek przy mieszanych podstawach [9]. Algorytm przebiega po wszystkich miejscach na permutacje σ_{kj}, przy czym pomija pozycje zawierające *None*. Kolejność generowanych permutacji zależy od permutacji umieszczonych na pozycjach σ_{kj}. Jest to pewna odmiana Algorytmu G opisanego przez Knutha [9].

Dla grupy S_n, czyli wszystkich permutacji n elementów, istnieje wiele możliwości generowania permutacji. Knuth opisuje np. Algorytm P (proste wymiany) do generacji permutacji różniących się kolejno zamianą pary sąsiednich elementów. Opuszczając co drugą permutację, czyli permutacje nieparzyste, możemy otrzymać permutacje z grupy A_n.

Dla porównania zamieszczamy generator n-krotek, który został uogólniony w metodzie iterperms().

```
def itertuple(M):
    """Generator wszystkich n-krotek przy mieszanych podstawach."""
    n = len(M)
    a = [0] * n

    while True:
        yield a
        j = n - 1
        while a[j] == M[j]-1 and j >= 0:
            a[j] = 0
            j = j - 1
        if j < 0:
            break
        else:
            a[j] = a[j] + 1
```

4.4.3. Klasa Group - wstawianie permutacji do grupy

Tworzenie grupy w naszych implementacjach polega na kolejnym wstawianiu wybranych permutacji (generatorów grupy) do grupy trywialnej. W metodzie insert() najpierw sprawdza się, czy permutacja nie należy już do grupy. Następnie występują operacje zmierzające do zapewnienia, że aksjomaty grupy będą spełnione.

W implementacji podstawowej nowa permutacja mnoży wszystkie permutacje istniejące wcześniej w grupie, przez co powstaje nowa permutacja. W kolejnych iteracjach nowo powstała permutacja mnoży poprzednio istniejące aż do momentu, gdy nie uzyskamy nowych permutacji. Nie jest to zbyt wydajna metoda.

W implementacji zaawansowanej metoda insert() przekazuje daną permutacje do algorytmu A, który ma za zadanie uaktualniać zbiór silnych generatorów oraz przekazać do algorytmu B wszystkie iloczyny danej permutacji ze wszystkimi permutacjami σ_{kj}. Z kolei algorytm B uaktualnia zestaw permutacji σ_{kj} oraz ewentualnie wywołuje algorytm A [10]. Metody alg_A() i alg_B() wywołują się wzajemnie rekurencyjnie, co czasem prowadzi do głębokiego zagnieżdżenia dla dużych grup. W Pythonie możemy zmieniać maksymalną dopuszczalną głębokość rekurencji za pomocą polecenia sys.setrecursionlimit().

Nasza implementacja zaawansowana działa poprawnie, choć często zbiór silnych generatorów nie jest optymalny. Możemy ręcznie poprawić zbiór all_T przez ponowne wygenerowanie grupy, wstawiając kolejno silne generatory posortowane rosnąco ze względu na kryterium Perm.max().

4.4.4. Klasa Group - podgrupy

Metoda is_subgroup() sprawdza, czy grupa H jest podgrupą grupy G o takim samym rozmiarze permutacji. W implementacji podstawowej sprawdzamy wprost, czy wszystkie permutacje z H należą do G. Szacowana złożoność czasowa metody wynosi $O(n|H|)$ (pętla po H, test przynależności do G i wyliczenie etykiety). W implementacji zaawansowanej wystarczy sprawdzić czy wszystkie silne generatory z H należą do G. Szacowana złożoność czasowa wynosi $O(|S_H| \cdot n^2)$, gdzie S_H oznacza zbiór silnych generatorów H, a test przynależności do G jest rzędu $O(n^2)$.

Metoda is_normal() sprawdza, czy grupa H jest podgrupą normalną grupy G o takim samym rozmiarze permutacji. W implementacji podstawowej sprawdzamy z definicji równość warstw lewych i prawych. W implementacji zaawansowanej wystarczy dokonać sprawdzenia dla silnych generatorów.

Metoda subgroup_search() jest wykorzystywana do wygenerowania podgrupy H grupy G, która zawiera permutacje spełniające podany warunek prop(). Funkcja prop() jako argument przyjmuje permutację, a zwraca wartoć typu *bool*. Funkcja musi być zgodna z aksjomatami grupy, np. jeżeli prop(p) i prop(q) zwracają *True*, to prop(p*q) też musi zwracać *True*. Metoda subgroup_search() jest stosowana np. przy obliczaniu normalizatora i centralizatora. W najprostrzej implementacji przebiegamy przez wszystkie permutacje grupy G i sprawdzamy wynik funkcji prop(), co jest czasochłonne. Zaawansowane implementacje, jak w *SymPy*, wykonują szereg testów, które pozwalają pomijać gałęzie z permutacjami nie rokującymi nadziei. Wykonuje się przy tym wiele operacji na bazie grupy. W wielu przypadkach udaje się tym sposobem rozwiązać dany problem w akceptowalnym czasie.

4.4.5. Klasa Group - centralizator, centrum, normalizator

Metoda centralizer () korzysta z prostego faktu, że $C_G(S) = C_G(\langle S \rangle)$. Dzięki temu argumentem metody może być grupa i nie musimy komplikować interfejsu obsługą pojedynczego elementu grupy, czy podzbioru nie będącego grupą [13]. W implementacji podstawowej z definicji centralizatora sprawdzamy przemienność elementów grupy. Szacowana złożoność czasowa wynosi $O(n|G| \cdot |H|)$. W implementacji zaawansowanej korzystamy z metody subgroup_search(), gdzie dostarczana funkcja sprawdza warunek centralizatora na bazie generatorów grupy H. Szacowana złożoność czasowa wynosi $O(n|G| \cdot |S_H|)$, o ile operację insert () ograniczymy przez stałą.

Kod obliczający centrum grupy [matoda center()] wygląda w obu implementacjach tak samo i jest bezpośrednim zastosowaniem metody centralizer ().

Metoda normalizer() ma prosty interfejs dzięki skorzystaniu z faktu, że $N_G(S) = N_G(\langle S \rangle)$. W implementacji podstawowej sprawdzana jest równość warstw lewych i prawych wprost z definicji. Szacowana złożoność czasowa wynosi $O(n|G| \cdot |H|)$. W implementacji zaawansowanej korzystamy z metody subgroup_search(), gdzie dostarczana funkcja sprawdza warunek normalizatora na bazie generatorów grupy H. Szacowana złożoność czasowa wynosi $O(n^3|G| \cdot |S_H|)$, o ile operację insert () ograniczymy przez stałą.

4.4.6. Klasa Group - domknięcie normalne, komutant

Metoda normal_closure() zwraca domknięcie normalne podzbioru grupy, czyli najmniejszą podgrupę normalną zawierającą ten podzbiór. Bez zmniejszenia ogólności przyjmujemy, że argumentem metody jest podgrupa generowana przez ten zbiór. Różnica między implementacją podstawową a zaawansowaną polega na stosowaniu pętli po generatorach, a nie po wszystkich elementach grup.

Metoda commutator() zwraca komutant dwóch podgrup. Implementacja podstawowa jest wprost zastosowaniem definicji komutanta. Implementacja zaawansowana jest również prosta, ale nietrywialna. Wymaga obliczenia domknięcia normalnego zbioru komutatorów wszystkich generatorów [13].

Metoda derived_subgroup() jest bezpośrednim zastosowaniem metody commutator() do obliczenia komutanta grupy.

4.4.7. Klasa Group - iloczyn prosty grup

Metoda __mul__() pozwala znaleźć grupę będącą iloczynem prostym dwóch grup. Rozmiar permutacji końcowej grupy jest sumą rozmiarów permutacji grup początkowych.

W implementacji podstawowej rozszerzamy odpowiednio listy wewnętrzne wszystkich permutacji z obu grup początkowych. Następnie wstawiamy permutacje do nowo utworzonej grupy trywialnej, a metoda `insert()` zapewnia spełnienie aksjomatów grupy. W implementacji zaawansowanej wystarczy dokonać odpowiednich rozszerzeń list wewnętrznych dla silnych generatorów w obu początkowych grupach, a następnie wstawić te zmienione generatory do nowej grupy trywialnej [11].

4.4.8. Klasa Group - działanie grupy na zbiorze

Rozważamy standardowe działanie grupy na zbiorze $\Omega = \{0, 1, \cdots, n-1\}$, gdzie $F(p, k) = p[k]$, p jest permutacją z S_n. Metoda `orbits()` dla danej listy punktów z Ω zwraca listę orbit. Orbita w naszej implementacji jest listą liczb, przy czym kolejność liczb nie jest istotna. W implementacji podstawowej mamy pętlę po podanych punktach i zagnieżdżoną pętlę po wszystkich permutacjach grupy. Pomocnicza lista *used* zapewnia, że punkty na orbitach nie będą się powtarzać. Szacowana złożoność czasowa metody wynosi $O(|G| \cdot \text{(liczba orbit)})$. W implementacji zaawansowanej, oprócz pętli po podanych punktach, mamy pętlę po budowanej orbicie i trzecią pętlę po silnych generatorach. Istotne jest, że zakres drugiej pętli rośnie, ponieważ trzecia pętla dodaje do orbity nowe punkty. Szacowana złożoność czasowa dla jednego punktu $O(\text{(długość orbity)} \cdot |S_G|)$, a dla wszystkich punktów będzie $O(n \cdot |S_G|)$ [13].

Metoda `action()` znajduje nową grupę indukowaną przez standardowe działanie grupy na jednej orbicie starej grupy. Nowa grupa ma rozmiar równy wielkości orbity starej grupy. Permutacje ze starej grupy muszą być przetłumaczone na nowy rozmiar i wstawione do nowej grupy. W implementacji podstawowej przekształcamy wszystkie permutacje starej grupy. W implementacji zaawansowanej wystarczy przetłumaczyć silne generatory starej grupy. Interfejs metody jest wzorowany na GAP.

4.4.9. Klasa Group - stabilizator

Metoda `stabilizer()` dla danego punktu zwraca podgrupę permutacji, które nie przenoszą tego punktu. W implementacji podstawowej w pętli po wszystkich permutacjach zostawiamy tylko te, które nie przemieszczają punktu. Szacowana złożoność czasowa wynosi $O(|G|)$, jeżeli wykorzystamy bezpośrednio etykiety. W implementacji zaawansowanej korzystamy z dwóch ważnych zmiennych pomocniczych. Po pierwsze budujemy orbitę podanego punktu $\omega \in \Omega$. Po drugie budujemy słownik z permutacjami, które przenoszą dany punkt ω na inne punkty orbity. Dzięki tym zmiennym pomocniczym potrafimy na bazie silnych generatorów znaleźć permutacje generujące podgrupę stabilizatora [13]. Jeżeli przyjmiemy, że operacja `insert()` jest ograniczona przez stałą, to złożoność czasowa będzie $O(n^2 \cdot |S_G|)$.

4.4.10. Testy wydajnościowe dla grup permutacji

Testy polegały na generowaniu grup S_n o coraz większym rozmiarze. Sprawdzany był rozmiar pamięci zajmowanej przez interpreter Pythona oraz rozmiar pliku, który zawierał wyeksportowaną badaną grupę symetryczną S_n przy pomocy modułu *pickle*. Wyniki badań grup symetrycznych S_n zaprezentowane zostały w tabelach 4.4 i 4.5.

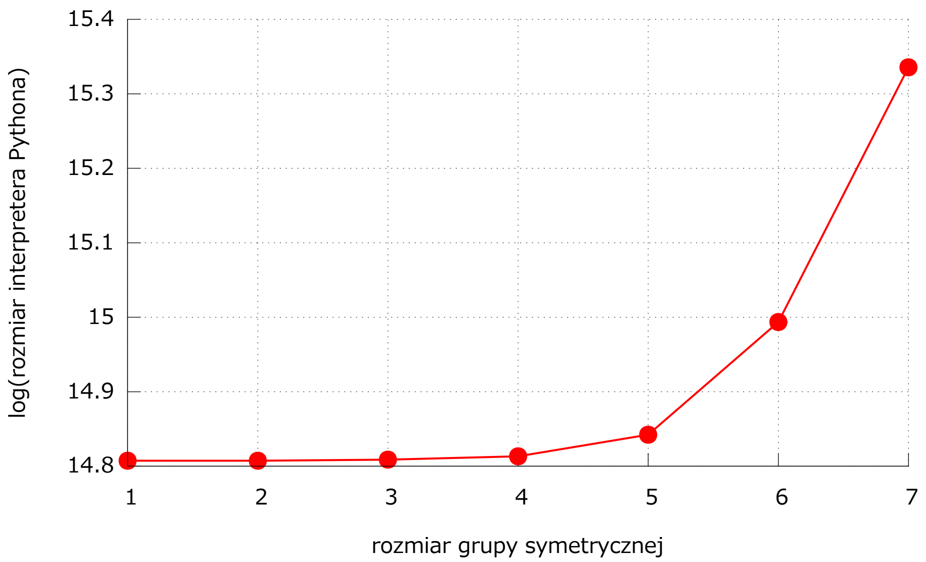
Zaprezentowane wykresy są graficzną interpretacją badań na grupach symetrycznych. Wykresy przedstawiają zależność pomiędzy rozmiarem pamięci interpretera Pythona i pliku *pickle*, a rozmiarem n grupy symetrycznej S_n. Wykresy zostały sporządzone dla implementacji podstawowej i zaawansowanej.

Tabela 4.4. Badanie grup symetrycznych dla implementacji podstawowej.

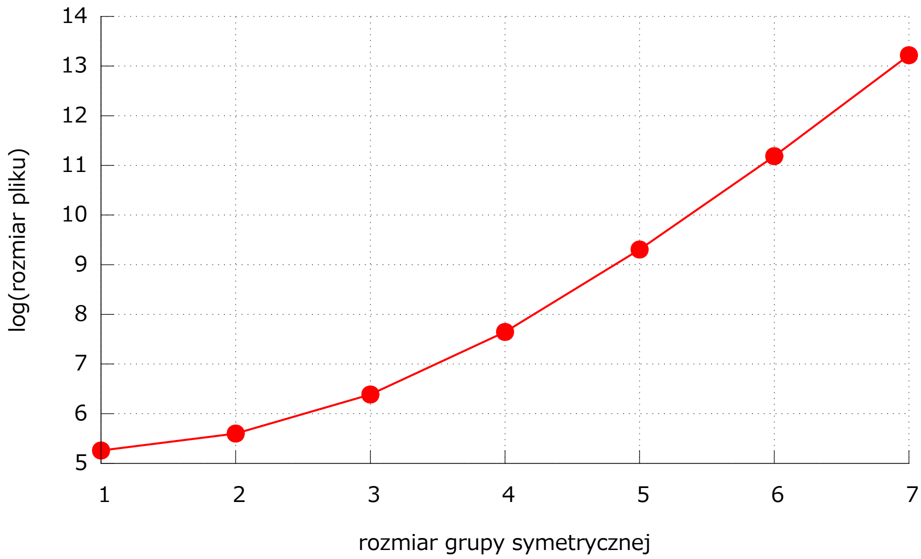
	Implementacja podstawowa	
Rozmiar grupy S_n	Rozmiar interpretera Pythona	Rozmiar pliku *pickle*
S_1	2 696 KB	192 B
S_2	2 696 KB	270 B
S_3	2 700 KB	594 B
S_4	2 712 KB	2,09 KB
S_5	2 792 KB	11 KB
S_6	3 248 KB	72,1 KB
S_7	4 572 KB	550 KB

Tabela 4.5. Badanie grup symetrycznych dla implementacji zaawansowanej.

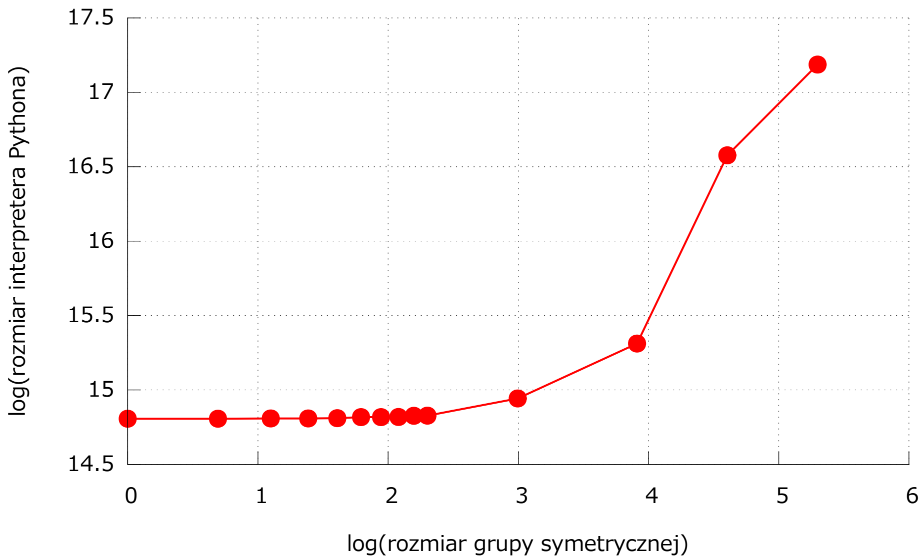
	Implementacja zaawansowana	
Rozmiar grupy S_n	Rozmiar interpretera Pythona	Rozmiar pliku *pickle*
S_1	2 696 KB	270 B
S_2	2 696 KB	487 B
S_3	2 700 KB	806 B
S_4	2 700 KB	1,36 KB
S_5	2 704 KB	1,76 KB
S_6	2 724 KB	2,81 KB
S_7	2 724 KB	3,52 KB
S_8	2 728 KB	5,08 KB
S_9	2 748 KB	6,26 KB
S_{10}	2 752 KB	8,36 KB
S_{20}	3 088 KB	47,47 KB
S_{50}	4 464 KB	542 KB
S_{100}	15 828 KB	3,12 MB
S_{200}	29 100 KB	9,79 MB



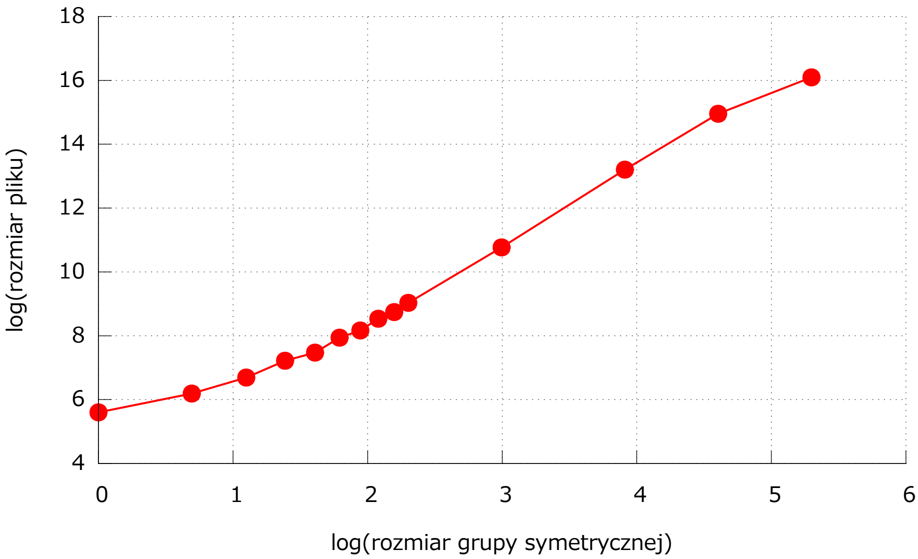
Rysunek 4.2. Zależność rozmiaru interpretera Pythona od rozmiaru n grupy symetrycznej w implementacji podstawowej.



Rysunek 4.3. Zależność rozmiaru pliku od rozmiaru n grupy symetrycznej w implementacji podstawowej.



Rysunek 4.4. Zależność rozmiaru interpretera Pythona od rozmiaru n grupy symetrycznej w implementacji zaawansowanej.



Rysunek 4.5. Zależność rozmiaru pliku od rozmiaru n grupy symetrycznej w implementacji zaawansowanej. Widać zależność $O(n^3)$.

5. Przykładowe obliczenia

W tym rozdziale chcemy przedstawić przykładowe obliczenia wykonane za pomocą stworzonego przez nas kodu. Dobrym poligonem doświadczalnym dla grup permutacji są grupy kostek Rubika o różnych rozmiarach.

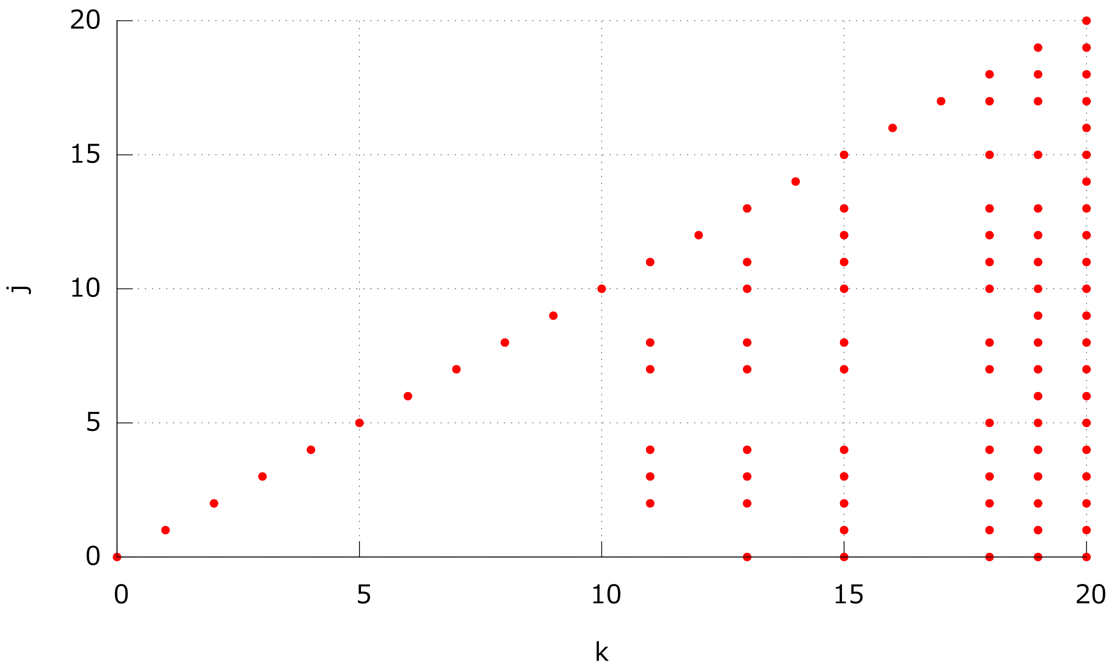
5.1. Grupa kostki Rubika $2 \times 2 \times 2$

Siatka kostki Rubika $2 \times 2 \times 2$ zawiera jeden nieruchomy wierzchołek z polami oznaczonymi przez znak X. Grupę generują trzy ćwierćobroty ścian.

```
+-----+
| 1 2 |
| X 3 |
+-----+-----+-----+-----+
| X 4 | 7 8 |11 12|15  X|
| 5 6 | 9 10|13 14|16 17|
+-----+-----+-----+-----+
|18 19|
|20 0 |
+-----+
```

```
>>> from perms import *
>>> from groups import *
>>>
>>> N = 21
>>> R1 = Perm(N)(2,13,19,4)(3,11,0,6)(7,8,10,9)
>>> D1 = Perm(N)(5,9,13,16)(6,10,14,17)(18,19,0,20)
>>> B1 = Perm(N)(1,16,0,8)(2,15,20,10)(11,12,14,13)
>>> R2 = R1 * R1 ; R3 = R1 * R2
>>> D2 = D1 * D1 ; D3 = D1 * D2
>>> B2 = B1 * B1 ; B3 = B1 * B2
>>> generators = [R1, D1, B1]
>>> face_turns = [R1, R2, R3, D1, D2, D3, B1, B2, B3]
>>> quarter_turns = [R1, R3, D1, D3, B1, B3]
>>> [perm.order() for perm in generators]
[4, 4, 4]                       # rzędy generatorów
>>> cube2 = Group(N)
>>> for perm in generators:
...         cube2.insert(perm)
...
>>> cube2.order()               # rząd grupy
3674160
>>> cube2.base()
[20, 19, 18, 15, 13, 11]
>>> cube2.orbits(range(N))
[[0, 6, 20, 8, 3, 13, 2, 17, 10, 5, 1, 18, 11, 4, 7, 12, 19, 16, 9, 15, 14]]
>>> cube2.is_transitive()
True
>>> Perm(N)(3,7,4)(6,19,9) in cube2  # obroty przeciwne
True
>>> Perm(N)(3,7,4)(6,9,19) in cube2  # obroty zgodne
False
>>> Perm(N)(3,6)(7,9)(4,19) in cube2 # zamiana wierzchołków
True
>>> len(cube2.all_T)
```

```
16
>>> import pickle
>>> file = open("cube2.pickle", "w")
>>> pickle.dump(cube2, file)
>>> file.close()
>>>
```



Rysunek 5.1. Obraz permutacji σ_{kj} należących do układu równoległego dla grupy kostki Rubika $2 \times 2 \times 2$. Widać punkty bazy $[20, 19, 18, 15, 13, 11]$.

Zestaw silnych generatorów otrzymany po wstawieniu trzech zwykłych generatorów nie jest optymalny (mamy 16 silnych generatorów). Dokładniejsza analiza pokazuje, że wystarczający jest zestaw 7 silnych generatorów. Test poprawności wyboru tych nowych silnych generatorów został umieszczony w klasie *TestGroupRubik2*.

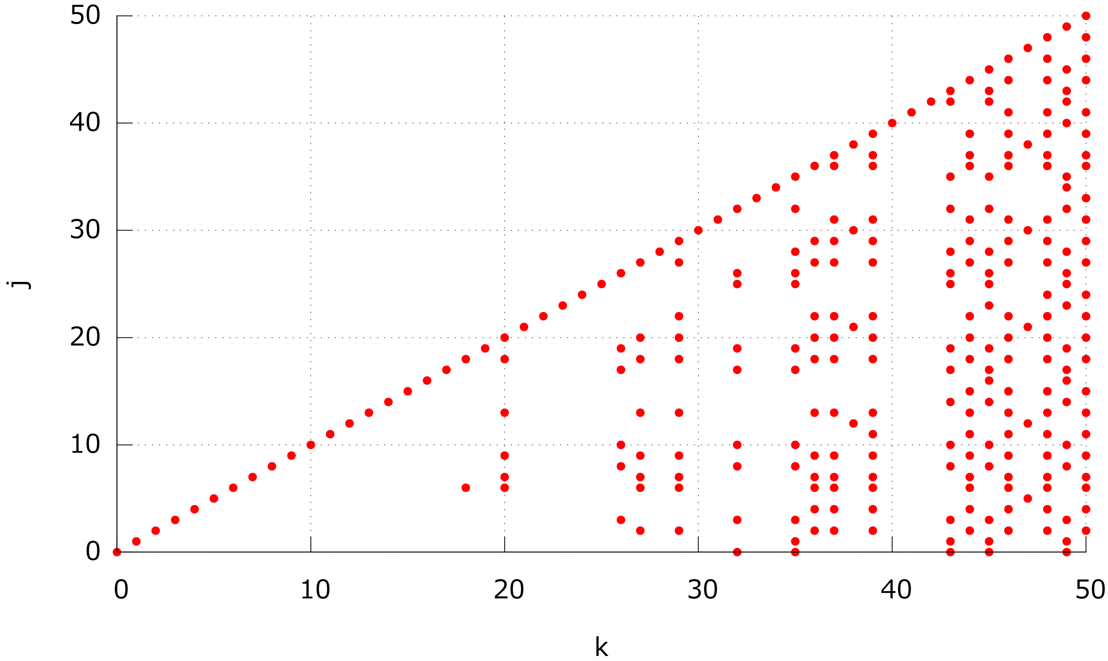
5.2. Grupa kostki Rubika $3 \times 3 \times 3$

Na rysunku przedstawiono siatkę kostki Rubika $3 \times 3 \times 3$ z nieruchomym rogiem. Analiza kostki z nieruchomymi środkami ścian jest zawarta w klasie *TestGroupRubik3*.

```
+------------+
|  1   2   3 |
|  4   5   6 |
|  X   7   8 |
+------------+------------+------------+------------+
|  X   9  10 | 17  18  19 | 26  27  28 | 35  36   X |
| 11  12  13 | 20  21  22 | 29  30  31 | 37  38  39 |
| 14  15  16 | 23  24  25 | 32  33  34 | 40  41  42 |
+------------+------------+------------+------------+
| 43  44  45 |
| 46  47  48 |
| 49  50   0 |
+------------+
```

```
>>> from perms import *
>>> from groups import *
>>>
>>> N = 51
>>> R1 = Perm(N)(3,32,45,10)(6,29,48,13)(8,26,0,16)(17,19,25,23)\
... (18,22,24,20)
>>> R2 = Perm(N)(2,33,44,9)(5,30,47,12)(7,27,50,15)
>>> D1 = Perm(N)(14,23,32,40)(15,24,33,41)(16,25,34,42)(0,49,43,45)\
... (44,48,50,46)
>>> D2 = Perm(N)(11,20,29,37)(12,21,30,38)(13,22,31,39)
>>> B1 = Perm(N)(0,19,1,40)(2,37,50,22)(3,35,49,25)(26,28,34,32)\
... (27,31,33,29)
>>> B2 = Perm(N)(4,41,48,18)(5,38,47,21)(6,36,46,24)
>>> generators = [R1, R2, D1, D2, B1, B2]
>>> cube3 = Group(N)
>>> for perm in generators:
...         cube3.insert(perm)
...
>>> cube3.order()
43252003274489856000L
>>> cube3.base()
[50, 49, 48, 47, 46, 45, 44, 43, 39, 38, 37, 36, 35, 32, 29,
 27, 26, 20, 18]
>>> cube3.orbits(range(N))
[[0, 16, 49, 14, 43, 32, 26, 10, 42, 19, 25, 3, 45, 17, 8, 23,
 28, 40, 1, 35, 34], [2, 33, 13, 6, 29, 9, 7, 44, 22, 15, 41, 20,
 37, 46, 18, 27, 31, 39, 36, 48, 4, 24, 11, 50], [5, 30, 12,
 21, 38, 47]]
>>> corners, edges, centers = cube3.orbits(range(N))
>>> len(cube3.all_T)
69
>>> g1 = cube3.action(corners)
>>> g1.size
21
>>> g1.order()
3674160
>>> g1.is_transitive()
True
>>> g1.base()
[20, 19, 15, 14, 11, 10]
>>> g2 = cube3.action(edges)
>>> g2.size
24
>>> g2.order()
980995276800L
>>> g2.is_transitive()
True
>>> g2.base()
[23, 22, 21, 20, 16, 15, 14, 13, 11, 9, 8]
>>> g3 = cube3.action(centers)
>>> g3.size
6
>>> g3.order()
24
>>> g3.is_transitive()
True
>>> g3.base()
```

```
[5, 4]
>>> import pickle
>>> file = open("cube3.pickle", "w")
>>> pickle.dump(cube3, file)
>>> file.close()
>>>
>>> alist = list(cube3.all_T)
>>> alist.sort(key=Perm.max)
>>> cube3 = Group(N)
>>> for perm in alist:
...         cube3.insert(perm)
...
>>> len(cube3.all_T)
21
>>>
```



Rysunek 5.2. Obraz permutacji σ_{kj} należących do układu równoległego dla grupy kostki Rubika $3 \times 3 \times 3$.

5.3. Grupa kostki Rubika $4 \times 4 \times 4$

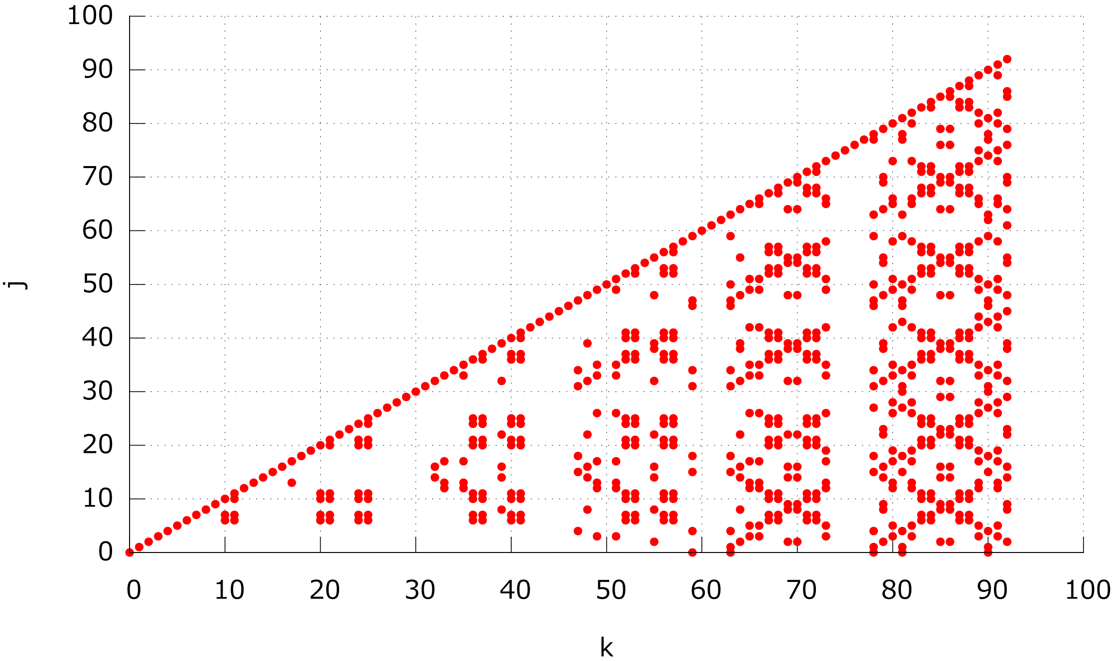
Na rysunku przedstawiono siatkę kostki Rubika $4 \times 4 \times 4$ z nieruchomym rogiem.

```
+-------------+
|  1  2  3  4 |
|  5  6  7  8 |
|  9 10 11 12 |
|  X 13 14 15 |
+-------------+-------------+-------------+-------------+
|  X 16 17 18 | 31 32 33 34 | 47 48 49 50 | 63 64 65  X |
| 19 20 21 22 | 35 36 37 38 | 51 52 53 54 | 66 67 68 69 |
| 23 24 25 26 | 39 40 41 42 | 55 56 57 58 | 70 71 72 73 |
```

```
| 27 28 29 30 | 43 44 45 46 | 59 60 61 62 | 74 75 76 77 |
+-------------+-------------+-------------+-------------+
| 78 79 80 81 |
| 82 83 84 85 |
| 86 87 88 89 |
| 90 91 92  0 |
+-------------+
```

```
>>> from perms import *
>>> from groups import *
>>>
>>> N = 93
X1 = Perm(N)(1,74,0,34)(2,70,92,38)(3,66,91,42)\
... (4,63,90,46)(47,50,62,59)(48,54,61,55)(49,58,60,51)(52,53,57,56)
X2 = Perm(N)(5,75,89,33)(6,71,88,37)(7,67,87,41)(8,64,86,45)
X3 = Perm(N)(9,76,85,32)(10,72,84,36)(11,68,83,40)(12,65,82,44)
Y1 = Perm(N)(4,59,81,18)(8,55,85,22)(12,51,89,26)\
... (15,47,0,30)(31,34,46,43)(32,38,45,39)(33,42,44,35)(36,37,41,40)
Y2 = Perm(N)(3,60,80,17)(7,56,84,21)(11,52,88,25)(14,48,92,29)
Y3 = Perm(N)(2,61,79,16)(6,57,83,20)(10,53,87,24)(13,49,91,28)
Z1 = Perm(N)(27,43,59,74)(28,44,60,75)(29,45,61,76)\
... (30,46,62,77)(78,81,0,90)(79,85,92,86)(80,89,91,82)(83,84,88,87)
Z2 = Perm(N)(23,39,55,70)(24,40,56,71)(25,41,57,72)(26,42,58,73)
Z3 = Perm(N)(19,35,51,66)(20,36,52,67)(21,37,53,68)(22,38,54,69)
>>> generators = [X1, X2, X3, Y1, Y2, Y3, Z1, Z2, Z3]
>>> cube4 = Group(N)
>>> for perm in generators:
...         cube4.insert(perm)
...
>>> cube4.order()
707195371192426622240452051915172831683411968000000000L
>>> cube4.base()
[92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78, 73,
 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 59, 57, 56, 55, 53, 52,
 51, 49, 48, 47, 41, 40, 39, 37, 36, 35, 33, 32, 25, 24, 21, 20,
 17, 11, 10]
>>> len(cube4.all_T)
301
>>> cube4.orbits(range(N))
[[0, 34, 30, 74, 63, 47, 1, 31, 46, 15, 81, 43, 18, 78, 59, 50, 4,
 90, 27, 77, 62], [2, 70, 64, 8, 38, 22, 32, 39, 48, 9, 54, 85, 55,
 14, 61, 16, 29, 45, 92, 76, 23, 69, 86, 79], [3, 66, 42, 35, 26,
 91, 33, 49, 51, 5, 75, 12, 65, 60, 17, 13, 58, 82, 80, 44, 28,
 19, 89, 73], [6, 71, 10, 36, 37, 72, 84, 11, 40, 41, 7, 83, 68,
 87, 21, 57, 52, 67, 88, 25, 53, 20, 24, 56]]
>>> corners, edges_left, edges_right, centers = cube4.orbits(range(N))
>>> g1 = cube4.action(corners)
>>> g1.size
21
>>> g1.order()
3674160
>>> g1.is_transitive()
True
>>> g1.base()
[20, 19, 16, 15, 14, 12]
>>> g2 = cube4.action(edges_left)
>>> g2.size
24
```

```
>>> g2.order()
620448401733239439360000L
>>> g2.is_transitive()
True
>>> g2.base()
[23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10,
 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> g3 = cube4.action(edges_right)
>>> g3.size
24
>>> g3.order()
620448401733239439360000L
>>> g3.is_transitive()
True
>>> g3.base()
[23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10,
 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> g4 = cube4.action(centers)
>>> g4.size
24
>>> g4.order()
620448401733239439360000L
>>> g4.is_transitive()
True
>>> g4.base()
[23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10,
 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> import pickle
>>> file = open("cube4.pickle", "w")
>>> pickle.dump(cube4, file)
>>> file.close()
>>>
>>> alist = list(cube3.all_T)
>>> alist.sort(key=Perm.max)
>>> cube3 = Group(N)
>>> for perm in alist:
...         cube4.insert(perm)
>>> len(cube4.all_T)
53
>>>
```



Rysunek 5.3. Obraz permutacji σ_{kj} należących do układu równoległego dla grupy kostki Rubika $4 \times 4 \times 4$.

6. Podsumowanie

W wyniku pracy udało się stworzyć dwie podstawowe klasy: Klasę *Perm* reprezentującą permutacje i klasę *Group* reprezentującą grupę permutacji. Dla tych klas określiliśmy interfejs, który zawiera typowe operacje znane z innych systemów algebry symbolicznej. Język Python pozwolił na przejrzystą implementacje algorytmów i struktur danych realizujące podane interfejsy. Zastosowanie modułu *unittest* pozwoliło na szybkie tworzenie kodu o dobrej jakości. W pracy zamieściliśmy przykładowe zastosowanie kodu do analizy grup kostki Rubika.

Jesteśmy przekonani, że zaprezentowane podejście ma szerokie perspektywy. Z jednej strony można znane zagadnienia zapisać w przejrzysty sposób, a z drugiej strony ułatwiona będzie droga w nieznane, czyli eksperymenty z nowymi algorytmami i rozwiązywanie nowych zadań. Już można zauważyć zjawisko pojawiania się w repozytoriach Pythona modułów dotyczących różnych dziedzin nauki, np. bioinformatyki, teorii grafów, obliczeń numerycznych, przetwarzania obrazów, itp.

Przedstawiony kod może być wykorzystany do nauki programowania w języku Python, ale również do wprowadzania podstawowych pojęć z teorii grup. Dalsze metody, jakie mogą być zrealizowane dla naszego oprogramowania to np. wyznaczanie struktury blokowej grupy tranzytywnej, wyznaczenie klas elementów sprzężonych, oraz wiele innych.

A. Permutacje

A.1. Testy dla klasy Perm

Listing A.1. Testowa klasa TestPerm.

```python
# -*- coding: cp1250 -*-
# perms_testy.py
#
# Testy dla permutacji.

from perms import *
import unittest

class TestPerm(unittest.TestCase):
    # Można określić czynności przygotowawcze.
    def setUp(self):
        self.N = 4
        self.E = Perm(self.N)
        self.R1 = Perm(self.N)(0,1)(2,3)
        self.R2 = Perm(self.N)(0,2)(1,3)
        self.P1 = Perm(self.N)(1,2)
        self.H = Perm(self.N)(0,1,3,2)
    def test_init(self):
        self.assertEqual(Perm(self.N)(1,2), Perm(self.N, data = [0, 2, 1, 3]))
        self.assertEqual(Perm(self.N), Perm(self.N)(1)(2))
        self.assertEqual(self.P1, Perm(self.N, data = [0, 2, 1, 3]))
        self.assertRaises(PermError, lambda: Perm(2, data = [0, 1, 2]))
    def test_repr(self):
        self.assertEqual(repr(self.E), "Perm(4)")
        self.assertEqual(repr(self.R1), "Perm(4)(0, 1)(2, 3)")
        self.assertEqual(repr(self.R2), "Perm(4)(0, 2)(1, 3)")
        self.assertEqual(repr(self.P1), "Perm(4)(1, 2)")
        self.assertEqual(repr(self.H), "Perm(4)(0, 1, 3, 2)")
    def test_label(self):
        self.assertEqual(self.E.label(), "0123")
        self.assertEqual(self.R1.label(), "1032")
        self.assertEqual(self.P1.label(), "0213")
        self.assertEqual(self.H.label(), "1302")
    def test_identity(self):
        self.assertTrue(self.E.is_identity())
        self.assertFalse(self.R1.is_identity())
        self.assertFalse(self.H.is_identity())
    def test_mul(self):
        self.assertEqual(self.E*self.E, self.E)
        self.assertEqual(self.R1*self.R1, self.E)
        self.assertNotEqual(self.R1*self.R2, self.E)
        self.assertRaises(PermError, lambda: Perm(2)*Perm(1))
    def test_invert(self):
        self.assertEqual(~self.E, self.E)
        self.assertEqual(~self.R1, self.R1)
        self.assertEqual(~self.R2, self.R2)
        self.assertEqual(~self.H, Perm(self.N)(0,2,3,1))
        self.assertEqual(self.H*~self.H, self.E)
    def test_order(self):
        self.assertEqual(self.E.order(), 1)
        self.assertEqual(self.R1.order(), 2)
        self.assertEqual(self.P1.order(), 2)
        self.assertEqual(self.H.order(), 4)
    def test_cmp(self):
```

```python
        self.assertFalse(self.H == self.E)
        self.assertTrue(self.E < self.H)
        self.assertTrue(self.H < self.H*self.H)
    def test_parity(self):
        self.assertEqual(self.E.parity(), 0)
        self.assertEqual(self.R1.parity(), 0)
        self.assertEqual(self.P1.parity(), 1)
        self.assertEqual(self.H.parity(), 1)
        self.assertFalse(self.H.is_even())
        self.assertTrue(self.H.is_odd())
        self.assertEqual(self.H.sign(), -1)
    def test_call(self):
        self.assertEqual(Perm(self.N)(0)(2)()(1,3), Perm(self.N)(1,3))
        self.assertEqual(Perm(self.N)(2,3)*Perm(self.N)(1,2),
            Perm(self.N)(1,3,2))
        self.assertEqual(Perm(self.N)(2,3)(1,2), Perm(self.N)(1,3,2))
        self.assertEqual(Perm(self.N)(1,2)(2,3), Perm(self.N)(1,2,3))
    def test_getitem(self):
        self.assertEqual(self.H[0], 1)
        self.assertEqual(self.H[1], 3)
        self.assertEqual(self.H[2], 0)
        self.assertEqual(self.H[3], 2)
    def test_pow(self):
        self.assertEqual(pow(self.H, 0), self.E)
        self.assertEqual(pow(self.H, 1), self.H)
        self.assertEqual(pow(self.H, -9), ~self.H)
        self.assertEqual(pow(self.H, 4), self.E)
        self.assertEqual(pow(self.H, 3), ~self.H)
        self.assertEqual(pow(self.H, 1000000), self.E)
    def test_random(self):
        self.assertTrue(self.E.random(2) in [Perm(2), Perm(2)(0,1)])
    def test_commutes_with(self):
        self.assertTrue(self.R1.commutes_with(self.R2))
        self.assertFalse(self.R1.commutes_with(self.H))
        self.assertEqual(self.R1.commutator(self.R2), self.E)
        self.assertEqual(self.R1.commutator(self.R2), self.E)
    def test_min_max(self):
        self.assertEqual(self.H.min(), 0)
        self.assertEqual(self.H.max(), 3)
        self.assertEqual(self.E.min(), 0)
        self.assertEqual(self.E.max(), 0)
        self.assertEqual(self.P1.min(), 1)
        self.assertEqual(self.P1.max(), 2)
    def test_list(self):
        self.assertEqual(self.E.list(), [0])
        self.assertEqual(self.E.list(2), [0, 1])
        self.assertEqual(self.P1.list(), [0, 2, 1])
        self.assertEqual(self.H.list(), [1, 3, 0, 2])
    def test_cycles(self):
        self.assertEqual(self.E.cycles(), [])
        self.assertEqual(self.R1.cycles(), [[0,1],[2,3]])
        self.assertEqual(self.P1.cycles(), [[1,2]])
        self.assertEqual(self.H.cycles(), [[0,1,3,2]])
    def test_support(self):
        self.assertEqual(self.E.support(), [])
        self.assertEqual(self.R1.support(), [0, 1, 2, 3])
        self.assertEqual(self.P1.support(), [1, 2])
        self.assertEqual(self.H.support(), [0, 1, 2, 3])
    def test_commutator(self):
        self.assertTrue(self.E.commutes_with(self.H))
        self.assertTrue(self.R1.commutes_with(self.R2))
        self.assertNotEqual(self.H.commutator(self.R1), self.E)
    def test_lehmer(self):
        p = Perm(4)(0,3)(1,2)   # [3,2,1,0]
        self.assertEqual(p.inversion_vector(), [3,2,1,0])
        self.assertEqual(self.E.rank_lex(), 0)
        self.assertEqual(self.R1.rank_lex(), 7)
        self.assertEqual(self.P1.rank_lex(), 2)
        self.assertEqual(self.H.rank_lex(), 10)
        self.assertEqual(Perm.unrank_lex(4,17), Perm(4)(0,2,1,3))
    def test_next_prev(self):
        pass
    def tearDown(self):
```

```
        pass

if __name__ == "__main__":
    #unittest.main()
    suite = unittest.TestLoader().loadTestsFromTestCase(TestPerm)
    unittest.TextTestRunner(verbosity=2).run(suite)
```

A.2. Klasa Perm

Listing A.2. Moduł perms.py [11].

```
# -*- coding: cp1250 -*-
# perms.py
#
# Badam permutacje.

import random

def gcd(a, b):
    """Oblicza największy wspólny dzielnik."""
    while b:
        a, b = b, a % b
    return a

def lcm(a, b):
    """Oblicza najmniejszą wspólną wielokrotność."""
    return a * b / gcd(a, b)

def factorial(n):
    """Silnia."""
    res = 1
    while n > 1:
        res = res * n
        n = n - 1
    return res

def swap(L, i, j):
    """Zamienia dwa elementy na liście."""
    L[i], L[j] = L[j], L[i]

class PermError(Exception):
    """Błąd permutacji."""
    pass

class Perm:
    """Klasa definiująca permutacje."""
    def __init__(self, size, data=None):
        """Tworzy instancję klasy Perm."""
        self.size = size
        if data:
            if self.size != len(data):
                raise PermError("different size and len(data)")
            self.data = data
        else:
            self.data = range(self.size)
    def __repr__(self):
        """Przedstawia permutacje w reprezentacji tekstowej."""
        tmp = ["Perm(%s)" % self.size]
        for cycle in self.cycles():
            tmp.append(str(tuple(cycle)))
        return "".join(tmp)
    def __len__(self):
        """Zwraca rozmiar permutacji."""
        return self.size
    def is_identity(self):
        """Testuje czy permutacja jest permutacją identycznościową."""
        return all(i == self.data[i] for i in range(self.size))
    def __invert__(self):
```

```
    """Znajduje permutację odwrotną."""
    perm = Perm(self.size)
    for i in range(perm.size):
        perm.data[self.data[i]] = i
    return perm
def __mul__(self, other):
    """Zwraca iloczyn permutacji."""
    if self.size != other.size:
        raise PermError("different size self and other")
    perm = Perm(self.size)
    for i in range(perm.size):
        perm.data[i] = self.data[other.data[i]]
    return perm
def __cmp__(self, other):
    """Leksykograficzne porównanie permutacji."""
    return cmp(self.data, other.data)
def __getitem__(self, k):
    """Znajduje element na podstawie zadanej pozycji."""
    return self.data[k]
def __pow__(self, n):
    """Znajduje potęgę permutacji."""
    if n == 0:
        return Perm(self.size)
    if n < 0:
        return pow(~self, -n)
    perm = self
    if n == 1:
        return self
    elif n == 2:
        return self*self
    else:
        tmp = Perm(self.size)
        while True:
            if n % 2 == 1:
                tmp = tmp*perm
                n = n-1
                if n == 0:
                    break
            if n % 2 == 0:
                perm = perm*perm
                n = n/2
    return tmp
def support(self):
    """Zwraca elementy przemieszczone przez permutację."""
    return [i for i in range(self.size) if self.data[i] != i]
def max(self):
    """Zwraca największy element przemieszczony przez permutację."""
    j_max = 0
    for j in range(self.size):
        if self.data[j] != j:
            j_max = j
    return j_max
def min(self):
    """Zwraca najmniejszy element przemieszczony przez permutację."""
    j_min = 0
    for j in range(self.size):
        if self.data[j] != j:
            j_min = j
            break
    return j_min
def list(self, size=None):
    """Zwraca permutacje w postaci listy."""
    if size is None:
        size = self.max()+1
    elif size < self.max()+1:
        raise PermError("size is too small")
    tmp = range(size)
    for i in range(min(size, self.size)):
        tmp[i] = self.data[i]
    return tmp
def label(self):
    """Zwraca tekstową etykietę permutacji."""
    if self.size > 62:
```

```
            raise PermError("problem with labels")
        letters = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        letters += "abcdefghijklmnopqrstuvwxyz_"
        tmp = []
        for item in self.data:
            tmp.append(letters[item])
        return "".join(tmp)
    def cycles(self):
        """Zwraca listę cykli permutacji."""
        unchecked = [True] * self.size
        cyclic_form = []
        for i in range(self.size):
            if unchecked[i]:
                cycle = []
                cycle.append(i)
                unchecked[i] = False
                j = i
                while unchecked[self.data[j]]:
                    j = self.data[j]
                    cycle.append(j)
                    unchecked[j] = False
                if len(cycle) > 1:
                    cyclic_form.append(cycle)
        return cyclic_form
    def order(self):
        """Zwraca rząd permutacji."""
        tmp = [len(cycle) for cycle in self.cycles()]
        return reduce(lcm, tmp, 1)
    def parity(self):
        """Zwraca parzystość permutacji (0 lub 1)."""
        unchecked = [True] * self.size
        c = 0
        for j in range(self.size):
            if unchecked[j]:
                c = c+1
                unchecked[j] = False
                i = j
                while self.data[i] != j:
                    i = self.data[i]
                    unchecked[i] = False
        return (self.size - c) % 2
    def is_even(self):
        """Testuje czy permutacja jest parzysta."""
        return self.parity() == 0
    def is_odd(self):
        """Testuje czy permutacja jest nieparzysta."""
        return self.parity() == 1
    def sign(self):
        """Zwraca znak permutacji (+1 lub -1)."""
        return (1 if self.parity() == 0 else -1)
    def commutes_with(self, other):
        """Testuje czy permutacje komutują ze sobą."""
        return not any(self.data[other.data[i]] !=
                    other.data[self.data[i]] for i in range(self.size))
    def commutator(self, other):
        """Znajduje komutator permutacji."""
        return self*other*~self*~other
    @classmethod
    def random(self, size):
        """Zwraca losową permutację o zadanym rozmiarze."""
        perm = Perm(size)
        random.shuffle(perm.data)
        return perm
    def inversion_vector(self):
        """Zwraca wektor inwersji dla permutacji."""
        lehmer = [0]*self.size
        for i in range(self.size):
            counter = 0
            for j in range(i+1, self.size):
                if self.data[i] > self.data[j]:
                    counter = counter+1
            lehmer[i] = counter
        return lehmer
```

```
def rank(self):
    """Rankowanie leksykograficzne permutacji."""
    res = 0
    data = list(self.data)
    n = self.size-1
    size = self.size
    psize = factorial(n)
    for j in range(size-1):
        res = res + data[j]*psize
        for i in range(j+1, size):
            if data[i] > data[j]:
                data[i] = data[i]-1
        psize = psize/n
        n = n-1
    return res
rank_lex = rank
@classmethod
def unrank(self, size, rank):
    """Przeciwieństwo rankowania leksykograficznego permutacji."""
    tmp = [0]*size
    psize = 1
    for i in range(size):
        new_psize = psize*(i+1)
        d = (rank % new_psize) / psize
        rank = rank - d*psize
        tmp[size-i-1] = d
        for j in range(size-i, size):
            if tmp[j] > d-1:
                tmp[j] = tmp[j]+1
        psize = new_psize
    return Perm(size, data=tmp)
unrank_lex = unrank
def rank_mr(self):
    """Rankowanie permutacji według Myrvold'a i Ruskey'a."""
    alist = self.list(self.size)
    blist = (~self).list(self.size)
    return Perm._mr_helper(self.size, alist, blist)
@classmethod
def unrank_mr(self, size, rank):
    """Przeciwieństwo rankowania permutacji według Myrvold'a i Ruskey'a."""
    tmp = range(size)
    old_size = size
    while size > 0:
        swap(tmp, size-1, rank % size)
        rank = rank/size
        size = size-1
    return Perm(old_size, data=tmp)
@classmethod
def _mr_helper(self, size, alist, blist):
    """Funkcja pomocnicza dla rankowania Myrvalda i Ruskey'ego."""
    if size == 1:
        return 0
    s = alist[size-1]
    swap(alist, size-1, blist[size-1])
    swap(blist, s, size-1)
    return s + size * Perm._mr_helper(size-1, alist, blist)
def __call__(self, *arguments):
    """Zwraca iloczyn permutacji i cyklu."""
    tmp = {}
    n = len(arguments)
    for i in range(n):
        tmp[arguments[i]] = self.data[arguments[(i+1) % n]]
    for key in tmp:
        self.data[key] = tmp[key]
    return self
def next_lex(self):
    """Leksykograficzny dostęp do następnej permutacji."""
    rank = self.rank_lex()
    if rank == factorial(self.size)-1:
        return None
    else:
        return Perm.unrank_lex(self.size, rank+1)
def prev_lex(self):
```

```
    """Leksykograficzny dostęp do poprzedniej permutacji."""
    rank = self.rank_lex()
    if rank == 0:
        return None
    else:
        return Perm.unrank_lex(self.size, rank-1)
```

B. Grupy permutacji

B.1. Testy dla klasy Group

Listing B.1. Zestawienie testów dla klasy Group.

```
# -*- coding: cp1250 -*-
# groups_testy.py
#
# Testy dla grup permutacji.

import unittest
from groups import *

class TestGroupCyclic(unittest.TestCase):
    # Można określić czynności przygotowawcze.
    def setUp(self):
        self.N = 6
        self.H = Perm(self.N)(0,1,2,4)(3,5)
        self.group = Group(self.N)
    # Test grupy cyklicznej Knutha.
    def test_insert(self):
        self.assertTrue(self.group.is_trivial())
        self.assertEqual(self.group.order(), 1)
        self.group.insert(self.H)
        self.assertEqual(self.group.order(), 4)
        self.assertFalse(self.group.is_trivial())
        self.assertTrue(Perm(self.N) in self.group)
        self.assertTrue(self.H in self.group)
        self.assertFalse(Perm(self.N)(1,2) in self.group)
        self.assertFalse(self.group.is_transitive())
        self.assertEqual(len(self.group.orbits(range(self.N))), 2)
    # Końcowe czynności czyszczące.
    def tearDown(self):
        pass

class TestGroupSymmetric(unittest.TestCase):
    # Można określić czynności przygotowawcze.
    def setUp(self):
        pass
    # Test grupy symetrycznej.
    def test_insert1(self):
        self.N = 5
        self.group = Group(self.N)
        self.assertEqual(self.group.order(), 1)
        self.group.insert(Perm(self.N)(0,1))
        self.assertEqual(self.group.order(), 2)
        self.group.insert(Perm(self.N)(1,2))
        self.assertEqual(self.group.order(), 6)
        self.group.insert(Perm(self.N)(2,3))
        self.assertEqual(self.group.order(), 24)
        self.group.insert(Perm(self.N)(3,4))
        self.assertEqual(self.group.order(), 120)
        self.assertTrue(self.group.is_transitive())
    def test_insert2(self):
        self.N = 5
        self.group = Group(self.N)
        order = 1
        for i in range(self.N-1):
            self.group.insert(Perm(self.N)(i,i+1))
```

```
            order = order*(i+2)
            self.assertEqual(self.group.order(), order)
        self.assertTrue(self.group.is_transitive())
    # Końcowe czynności czyszczące.
    def tearDown(self):
        pass

class TestGroupAlternating(unittest.TestCase):
    # Można określić czynności przygotowawcze.
    def setUp(self):
        pass
    # Test grupy symetrycznej.
    def test_insert1(self):
        self.N = 5
        self.group = Group(self.N)
        self.assertEqual(self.group.order(), 1)
        self.group.insert(Perm(self.N)(0,1,2))
        self.assertEqual(self.group.order(), 3)
        self.group.insert(Perm(self.N)(1,2,3))
        self.assertEqual(self.group.order(), 12)
        self.group.insert(Perm(self.N)(2,3,4))
        self.assertEqual(self.group.order(), 60)
        self.assertTrue(self.group.is_transitive())
    def test_insert2(self):
        self.N = 6
        self.assertTrue(self.N>2)
        self.group = Group(self.N)
        order = 1
        for i in range(self.N-2):
            self.group.insert(Perm(self.N)(i,i+1,i+2))
            order = order*(i+3)
            self.assertEqual(self.group.order(), order)
        self.assertTrue(self.group.is_transitive())
    # Końcowe czynności czyszczące.
    def tearDown(self):
        pass

class TestGroupRubik2(unittest.TestCase):
    # Można określić czynności przygotowawcze.
    def setUp(self):
        self.N = 21
        self.group = Group(self.N)
    # Test grupy symetrycznej.
    def test_insert(self):
        self.assertEqual(self.group.order(), 1)
        self.group.insert(Perm(self.N)(3,7,4)(2,8,11))
        self.assertEqual(self.group.order(), 3)
        self.group.insert(Perm(self.N)(3,8)(7,2)(4,11))
        self.assertEqual(self.group.order(), 6)
        self.group.insert(Perm(self.N)(3,10)(7,13)(4,0))
        self.assertEqual(self.group.order(), 6*9)
        self.group.insert(Perm(self.N)(3,12)(7,1)(4,15))
        self.assertEqual(self.group.order(), 6*9*12)
        self.group.insert(Perm(self.N)(3,5)(7,18)(4,17))
        self.assertEqual(self.group.order(), 6*9*12*15)
        self.group.insert(Perm(self.N)(3,6)(7,9)(4,19))
        self.assertEqual(self.group.order(), 6*9*12*15*18)
        self.group.insert(Perm(self.N)(3,14)(7,16)(4,20))
        self.assertEqual(self.group.order(), 6*9*12*15*18*21)
        self.assertTrue(self.group.is_transitive())
    # Końcowe czynności czyszczące.
    def tearDown(self):
        pass

class TestGroupRubik3(unittest.TestCase):
    # Można określić czynności przygotowawcze.
    def setUp(self):
        self.N = 48
        self.group = Group(self.N)
    # Test grupy kostki Rubika 3x3x3.
    def test_insert(self):
        U1 = Perm(self.N)(1,3,8,6)(2,5,7,4)(9,33,25,17)(10,34,26,18)(11,35,27,19)
        L1 = Perm(self.N)(33,35,40,38)(34,37,39,36)(1,9,41,32)(4,12,44,29)(6,14,46,27)
```

```
        F1 = Perm(self.N)(9,11,16,14)(10,13,15,12)(6,17,43,40)(7,20,42,37)(8,22,41,35)
        R1 = Perm(self.N)(17,19,24,22)(18,21,23,20)(8,25,0,16)(5,28,45,13)(3,30,43,11)
        B1 = Perm(self.N)(25,27,32,30)(26,29,31,28)(3,33,46,24)(2,36,47,21)(1,38,0,19)
        D1 = Perm(self.N)(41,43,0,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40)
        generators = [U1,L1,F1,R1,B1,D1]
        for perm in generators:
            self.group.insert(perm)
        self.assertEqual(len(self.group.orbits(range(self.N))), 2)
        self.assertEqual(self.group.order(), 43252003274489856000L)
    # Końcowe czynności czyszczące.
    def tearDown(self):
        pass

class TestGroupProduct(unittest.TestCase):
    # Można określić czynności przygotowawcze.
    def setUp(self):
        pass
    def test_product1(self):
        self.N1 = 3
        self.group1 = Group(self.N1)
        self.group1.insert(Perm(self.N1)(0,1,2))
        self.N2 = 5
        self.group2 = Group(self.N2)
        self.group2.insert(Perm(self.N2)(0,1,2,3,4))
        self.assertEqual(self.group1.order(), self.N1)
        self.assertEqual(self.group2.order(), self.N2)
        self.group3 = self.group1*self.group2
        self.assertEqual(self.group3.order(), self.N1*self.N2)
    def tearDown(self):
        pass

class TestGroupOrbits(unittest.TestCase):
    # Można określić czynności przygotowawcze.
    def setUp(self):
        pass
    def test_orbits1(self):
        self.N = 3
        self.group = Group(self.N)
        self.group.insert(Perm(self.N)(0,1))
        self.assertEqual(self.group.orbits(range(self.N)), [[0,1],[2]])
        self.assertEqual(self.group.orbits([0,1]),[[0,1]])
        self.assertFalse(self.group.is_transitive())
        self.assertFalse(self.group.is_transitive())
    def test_orbits2(self):
        self.N = 4
        self.group = Group(self.N)
        self.group.insert(Perm(self.N)(0,1))
        self.group.insert(Perm(self.N)(2,3))
        self.assertFalse(self.group.is_transitive())
        self.assertEqual(self.group.orbits(range(self.N)), [[0,1],[2,3]])
        self.assertEqual(self.group.orbits([0,1]), [[0,1]])
        self.assertEqual(self.group.orbits([0,1,2]), [[0,1],[2,3]])
    def test_orbits3(self):    # grupa cykliczna.
        self.N = 10
        self.group = Group(self.N)
        self.group.insert(Perm(self.N)(0,1,2,3,4,5,6,7,8,9))
        self.assertTrue(self.group.is_transitive())
    def test_orbits4(self):
        self.N = 10
        self.group = Group(self.N)
        self.group.insert(Perm(self.N)(0,1,2))
        self.assertFalse(self.group.is_transitive())
        self.assertFalse(self.group.is_transitive())
    def test_action(self):
        self.N = 9
        self.group = Group(self.N)
        self.group.insert(Perm(self.N)(0,2)(3,5)(6,8))
        self.group.insert(Perm(self.N)(0,6)(1,7)(2,8))
        self.group.insert(Perm(self.N)(1,3)(2,6)(5,7))

        corners, edges, center = self.group.orbits(range(self.N))
        g1 = self.group.action(corners)
        g2 = self.group.action(edges)
```

```
        g3 = self.group.action(center)
        self.assertTrue(g1.is_transitive())
        self.assertTrue(g2.is_transitive())
        self.assertTrue(g3.is_transitive())
        self.assertEqual(g1.size, 4)
        self.assertEqual(g2.size, 4)
        self.assertEqual(g3.size, 1)
        self.assertEqual(g1.order(), 8)
        self.assertEqual(g2.order(), 8)
        self.assertEqual(g3.order(), 1)
    def tearDown(self):
        pass

class TestSubgroup(unittest.TestCase):
    # Można określić czynności przygotowawcze.
    def setUp(self):
        self.N = 4
        # Tworzę grupę symetryczną.
        self.group = Group(self.N)
        self.group.insert(Perm(self.N)(0,1))
        self.group.insert(Perm(self.N)(0,1,2,3))
    def test_subgroup_search(self):
        self.assertEqual(self.group.order(), 24)
        # Dopuszczam permutacje parzyste - grupa alternująca.
        self.group2 = self.group.subgroup_search(lambda x: x.is_even())
        self.assertEqual(self.group2.order(), 12)
        self.assertTrue(self.group2.is_transitive())
    def test_stabilizer(self):
        self.group2 = self.group.stabilizer(3)
        self.assertEqual(self.group2.order(), 6)
    def test_centralizer(self):
        self.group1 = Group(self.N)
        # Tworzę grupę cykliczną.
        self.group1.insert(Perm(self.N)(0,1,2,3))
        self.assertEqual(self.group1.order(), self.N)
        self.group2 = self.group1.center()
        self.assertEqual(self.group2.order(), self.N)
        # Dalej dla grupy symetrycznej.
        self.group2 = self.group.center()
        self.assertEqual(self.group2.order(), 1)
    def test_is_subgroup(self):
        self.group1 = Group(self.N)
        # Tworzę grupę cykliczną.
        self.group1.insert(Perm(self.N, range(self.N)))
        self.assertTrue(self.group1.is_subgroup(self.group))
        self.assertRaises(PermError, lambda: Group(2).is_subgroup(Group(3)))
        self.assertTrue(self.group1.is_abelian())
        self.assertFalse(self.group.is_abelian())
        self.assertTrue(self.group1.is_normal(self.group))
    def tearDown(self):
        pass

if __name__ == "__main__":
    #unittest.main()      # włącza wsztstkie testy.
    suite1 = unittest.TestLoader().loadTestsFromTestCase(TestGroupCyclic)
    suite2 = unittest.TestLoader().loadTestsFromTestCase(TestGroupSymmetric)
    suite3 = unittest.TestLoader().loadTestsFromTestCase(TestGroupAlternating)
    suite4 = unittest.TestLoader().loadTestsFromTestCase(TestGroupRubik2)
    suite5 = unittest.TestLoader().loadTestsFromTestCase(TestGroupRubik3)
    suite6 = unittest.TestLoader().loadTestsFromTestCase(TestGroupProduct)
    suite7 = unittest.TestLoader().loadTestsFromTestCase(TestGroupOrbits)
    suite8 = unittest.TestLoader().loadTestsFromTestCase(TestSubgroup)

    suite = unittest.TestSuite([suite5])
    unittest.TextTestRunner(verbosity=2).run(suite)
```

B.2. Klasa Group - implementacja podstawowa

Listing B.2. Moduł groups.py [11].

```
# -*- coding: cp1250 -*-
# groups.py
#
# Badam grupy permutacji (implementacja podstawowa).

from perms import Perm, PermError

class Group(dict):
    """Klasa definiująca grupę permutacji."""
    def __init__(self, size):
        """Tworzy instancje Group."""
        self.size = size
        perm = Perm(self.size)
        self[perm.label()] = perm
    # def __str__
    #    """__str__ dziedziczone z dict."""
    order = dict.__len__
    #    """Zwraca rząd grupy."""
    def __mul__(self, other):
        """Zwraca iloczyn prosty grup permutacji."""
        size1 = self.size
        size2 = other.size
        size3 = size1+size2
        left = range(size1)
        right = range(size1, size3)
        group3 = Group(size3)
        for perm in self.iterperms():
            group3.insert(Perm(size3, data=perm.data+right))
        for perm in other.iterperms():
            shiftdata = [x+size1 for x in perm.data]
            group3.insert(Perm(size3, data=left+shiftdata))
        return group3
    def is_trivial(self):
        """Testuje czy grupa jest trywialna."""
        return self.order() == 1
    def __contains__(self, perm):
        """Testuje czy permutacja należy do grupy."""
        return dict.__contains__(self, perm.label())
    def insert(self, perm):
        """Wprowadza permutację do grupy generując nowe
        permutacje spełniające właściwości grupy."""
        if self.size != perm.size:
            raise PermError("different size")
        label1 = perm.label()
        if perm in self:
            return
        old_order = len(self)
        self[label1] = perm
        tmp1 = {}
        tmp1[label1] = perm
        tmp2 = {}
        new_order = len(self)
        while new_order > old_order:
            old_order = new_order
            for label1 in tmp1:
                for label2 in self.iterlabels():
                    perm3 = tmp1[label1]*self[label2]
                    label3 = perm3.label()
                    if perm3 not in self and label3 not in tmp2:
                        tmp2[label3] = perm3
            self.update(tmp2)
            tmp1 = tmp2
            tmp2 = {}
            new_order = len(self)
    def iterperms(self):
        """Generator permutacji z grupy."""
        return self.itervalues()
    def iterlabels(self):
        """Generator etykiet permutacji z grupy."""
        return self.iterkeys()
    def is_abelian(self):
        """Testuje czy grupa jest abelowa."""
```

```python
        for perm1 in self.iterperms():
            for perm2 in self.iterperms():
                if perm2 <= perm1:
                    continue
                if not perm1.commutes_with(perm2):
                    return False
        return True
    def is_subgroup(self, other):
        """H.is_subgroup(G). Testuje czy H jest podgrupą G.
        Zwraca prawdę, jeśli wszystkie elementy H należą do G."""
        if self.size != other.size:
            raise PermError("different size")
        if other.order() % self.order() != 0:
            return False
        return all(perm in other for perm in self.iterperms())
    def is_normal(self, other):
        """H.is_normal(G) testuje czy H jest podgrupą normalną w G.
        Dla każdego h w H, g w G, g*h*~g należą do G."""
        for perm1 in self.iterperms():
            for perm2 in other.iterperms():
                if perm2*perm1*~perm2 not in self:
                    return False
        return True
    def subgroup_search(self, prop):
        """Zwraca podgrupę wszystkich elementów spełniających
        właściwość prop()."""
        newgroup = Group(self.size)
        for perm in self.iterperms():
            if prop(perm):
                newgroup.insert(perm)
        return newgroup
    def normalizer(self, other):
        """G.normalizer(H) - zwraca normalizator H w G."""
        return self.subgroup_search(lambda perm:
        all((perm*perm2*~perm in other) for perm2 in other.iterperms()))
    def centralizer(self, other):
        """G.centralizer(H) - zwraca centralizator H w G."""
        if self.size != other.size:
            raise PermError("different size")
        if other.is_trivial() or self.is_trivial():
            return self
        newgroup = Group(self.size)
        for perm1 in self.iterperms():
            if all(perm1.commutes_with(perm2) for perm2 in other.iterperms()):
                newgroup.insert(perm1)
        return newgroup
    def center(self):
        """Zwraca centrum grupy."""
        return self.centralizer(self)
    def orbits(self, points):
        """Zwraca listę orbit."""
        used = [False]*self.size
        orblist = []
        for pt1 in points:
            if used[pt1]:
                continue
            orb = [pt1]
            used[pt1] = True
            for perm in self.iterperms():
                pt2 = perm[pt1]
                if not used[pt2]:
                    orb.append(pt2)
                    used[pt2] = True
            orblist.append(orb)
        return orblist
    def is_transitive(self, strict=True):
        """Testuje czy grupa jest tranzytywna (ma jedną orbitę)."""
        return len(self.orbits(range(self.size))) == 1
    def stabilizer(self, point):
        """Zwraca podgrupę stabilizatora."""
        newgroup = Group(self.size)
        for perm in self.iterperms():
            if perm[point] == point:
```

```
            newgroup.insert(perm)
        return newgroup
    def normal_closure(self, other):
        """Zwraca domknięcie normalne."""
        newgroup = Group(self.size)
        for perm in self.iterperms():
            for perm2 in other.iterperms():
                newgroup.insert(perm*perm2*~perm)
        return newgroup
    def commutator(self, group1, group2):
        """Zwraca komutant podgrup."""
        newgroup = Group(self.size)
        for perm1 in group1.iterperms():
            for perm2 in group2.iterperms():
                newgroup.insert(perm1.commutator(perm2))
        return newgroup
    def derived_subgroup(self):
        """Zwraca komutant grupy."""
        return self.commutator(self, self)
    def action(self, points):
        """Zwraca grupę indukowaną przez działanie."""
        if not self.is_transitive(points):
            raise PermError("grupa nie jest tranzytywna na punktach")
        tmp = {}
        n = len(points)
        for i in range(n):
            tmp[points[i]] = i
        newgroup = Group(n)
        for perm in self.iterperms():
            d = [tmp[perm[pt]] for pt in points]
            newgroup.insert(Perm(n, data=d))
        return newgroup
```

B.3. Klasa Group - implementacja zaawansowana

Listing B.3. Moduł groups.py.

```
# -*- coding: cp1250 -*-
# groups.py
#
# Badam grupy permutacji (implementacja zaawansowana).

from perms import Perm, PermError

class Group():
    """Klasa definiująca grupę permutacji."""
    def __init__(self, size):
        """Tworzy instancje Group."""
        self.size = size
        self.Sigma = [(k+1)*[None] for k in range(self.size)]
        for k in range(self.size):
        # identyczność sigma_kk.
            self.Sigma[k][k] = Perm(self.size)
        # Silne generatory.
        self.T = [[] for k in range(self.size)]
        self.all_Sigma = [Perm(self.size)]
        self.all_T = []
    def __str__(self):
        """Zwraca grupę permutacji i jej silne generatory w postaci string."""
        t = len(self.all_T)
        return "Group(%s) with %s strong generators" % (self.size, t)
    def order(self):
        """Zwraca rząd grupy."""
        res = 1
        for k in range(self.size):
            res = res * sum(1 for perm in self.Sigma[k] if perm)
        return res
    def __mul__(self, other):
```

```
        """Zwraca iloczyn prosty permutacji."""
        size1 = self.size
        size2 = other.size
        size3 = size1 + size2
        left = range(size1)
        right = range(size1, size3)
        group3 = Group(size3)

        for perm in self.all_T:
            group3.insert(Perm(size3, data=perm.data+right))
        for perm in other.all_T:
            shiftdata = [x+size1 for x in perm.data]
            group3.insert(Perm(size3, data=left+shiftdata))
        return group3
    def is_trivial(self):
        """Testuje czy grupa jest trywialna."""
        return self.order() == 1
    def __contains__(self, perm):
        """Testuje czy permutacja należy do grupy."""
        k = self.size - 1
        while k > 0:
            j = perm.data[k]
            if self.Sigma[k][j] is None:
                return False
            perm = (~self.Sigma[k][j]) * perm
            k = k - 1
        return True
    def insert(self, perm):
        """Wprowadza permutację do grupy generując nowe
        permutacje spełniające właściwości grupy."""
        if self.size != perm.size:
            raise PermError("wrong size of the perm")
        self.alg_A(perm.max(), perm)
    def iterperms(self):
        """Generator permutacji z grupy."""
        a = [0] * self.size
        while True:
            if all(self.Sigma[k][a[k]] != None for k in range(self.size)):
                perm = Perm(self.size)
                for k in range(self.size):
                    perm = self.Sigma[k][a[k]] * perm
                yield perm
            j = self.size-1
            while a[j] == j and j >= 0:
                a[j] = 0
                j = j-1
            if j < 0:
                break
            else:
                a[j] = a[j]+1
    def iterlabels(self):
        """Generator permutacji z grupy."""
        a = [0] * self.size
        while True:
            if all(self.Sigma[k][a[k]] != None for k in range(self.size)):
                perm = Perm(self.size)
                for k in range(self.size):
                    perm = self.Sigma[k][a[k]] * perm
                yield perm.label()
            j = self.size-1
            while a[j] == j and j >= 0:
                a[j] = 0
                j = j-1
            if j < 0:
                break
            else:
                a[j] = a[j]+1
    def is_abelian(self):
        """Testuje czy grupa jest abelowa."""
        for perm1 in self.all_T:
            for perm2 in self.all_T:
                if perm2 <= perm1:
                    continue
```

```
            if not perm1.commutes_with(perm2):
                return False
        return True
    def base(self):
        """Zwraca k dla których Sigmy nie są identycznością."""
        b = []
        for k in range(self.size):
            for j in range(k+1):
                if self.Sigma[k][j] and k != j:
                    b.append(k)
                    break
        b.reverse()
        return b
    def is_subgroup(self, other):
        """H.is_subgroup(G) - testuje czy H jest podgrupą G."""
        if self.size != other.size:
            raise PermError("different size")
        if other.order() % self.order() != 0:
            return False
        return all(perm in other for perm in self.all_T)
    def is_normal(self, other):
        """H.is_normal(G) - testuje czy H jest podgrupą normalną w G."""
        for perm1 in self.all_T:
            for perm2 in other.all_T:
                if perm2 * perm1 * ~perm2 not in self:
                    return False
        return True
    def subgroup_search(self, prop):
        """Zwraca podgrupę wszystkich elementów spełniających
        właściwość prop()."""
        newgroup = Group(self.size)
        for perm in self.iterperms():
            if prop(perm):
                newgroup.insert(perm)
        return newgroup
    def normalizer(self, other):
        """G.normalizer(H) - zwraca normalizator H w G."""
        return self.subgroup_search(lambda perm:
        all((perm*perm2*~perm in other) for perm2 in other.all_T))
    def centralizer(self, other):
        """G.centralizer(H) - zwraca centralizator H w G."""
        if other.is_trivial() or self.is_trivial():
            return self
        return self.subgroup_search(lambda perm:
        all(perm*perm2 == perm2*perm for perm2 in other.all_T))
    def center(self):
        """Zwraca centrum grupy."""
        return self.centralizer(self)
    def orbits(self, points):
        """Zwraca listę orbit."""
        used = [False] * self.size
        orblist = []
        for pt1 in points:
            if used[pt1]:
                continue
            orb = [pt1]
            used[pt1] = True
            for pt2 in orb:
                for perm in self.all_T:
                    pt3 = perm[pt2]
                    if not used[pt3]:
                        orb.append(pt3)
                        used[pt3] = True
            orblist.append(orb)
        return orblist
    def is_transitive(self, points=None):
        """Testuje czy grupa jest tranzytywna (ma jedną orbitę)."""
        if points is None:
            points = range(self.size)
        return len(self.orbits(points)) == 1
    def stabilizer(self, point):
        """Zwraca podgrupę stabilizatora."""
        orb = [point]
```

```python
        table = {point: Perm(self.size)}
        used = [False] * self.size
        used[point] = True
        stab = Group(self.size)
        for pt2 in orb:
            for perm in self.all_T:
                pt3 = perm[pt2]
                if not used[pt3]:
                    perm3 = perm * table[pt2]
                    orb.append(pt3)
                    table[pt3] = perm3
                    used[pt3] = True
                else:
                    perm2 = ~table[pt3] * perm * table[pt2]
                    stab.insert(perm2)
            return stab
    def normal_closure(self, other):
        """Zwraca domknięcie normalne."""
        newgroup = Group(self.size)
        for perm in self.all_T:
            for perm2 in other.all_T:
                newgroup.insert(perm*perm2*~perm)
        return newgroup
    def commutator(self, group1, group2):
        """Zwraca komutant podgrup."""
        newgroup = Group(self.size)
        for perm1 in group1.all_T:
            for perm2 in group2.all_T:
                newgroup.insert(perm1.commutator(perm2))
        return self.normal_closure(newgroup)
    def derived_subgroup(self):
        """Zwraca komutant grupy."""
        return self.commutator(self, self)
    def action(self, points):
        """Zwraca grupę indukowaną przez działanie."""
        if not self.is_transitive(points):
            raise PermError("grupa nie jest tranzytywna na punktach")
        tmp = {}
        n = len(points)
        for i in range(n):
            tmp[points[i]] = i
        newgroup = Group(n)
        for perm in self.all_T:
            d = [tmp[perm[pt]] for pt in points]
            newgroup.insert(Perm(n, data=d))
        return newgroup
    def alg_A(self, k, perm):
        """Dołącza permutację do silnych generatorów."""
        if perm in self:
            return
        j = perm.data[k]
        if self.Sigma[k][j] is not None:
            perm2 = ~self.Sigma[k][j] * perm
            # Trzeba się upewnić, jakiego rzędu jest perm.
            self.alg_A(perm2.max(), perm2)
            return
        self.T[k].append(perm)
        self.all_T.append(perm)
        for item in (self.all_Sigma):
            # Trzeba się upewnić jakiego rzędu jest perm.
            perm2 = perm * item
            self.alg_B(perm2.max(), perm2)
    def alg_B(self, k, perm):
        """Aktualizuje Sigmy."""
        if perm in self:
            return
        j = perm.data[k]

        if self.Sigma[k][j] is None:
            self.Sigma[k][j] = perm
            self.all_Sigma.append(perm)

            for item in (self.all_T):
```

```
                # Trzeba się upewnić, jakiego rzędu jest perm.
                perm2 = item * perm
                k_max = perm2.max()
                if k_max != k:
                    self.alg_A(k_max, perm2)
                else:
                    self.alg_B(k_max, perm2)
            return
        item = (~self.Sigma[k][j]) * perm
        self.alg_A(item.max(), item)
    def status(self, name="data.txt"):
        file = open(name, "w")
        for k in range(self.size):
            for j in range(k+1):
                if self.Sigma[k][j]:
                    file.write("%s %s\n" % (k, j))
        file.close()
```

Bibliografia

[1] Python Programming Language - Official Website,
`http://www.python.org/`.

[2] Piotr Maliński, *Przedstawienie Pythona - opis języka, kto go używa i do czego można go wykorzystać* [online], [dostęp: 5 sierpnia 2009],
`http://www.python.rk.edu.pl/w/p/python-co-jest-i-do-czego-mozna-go-uzyc/`.

[3] Andrzej Kapanowski, *Algorytmy i struktury danych z językiem Python* [online], [dostęp: 1 października 2012],
`http://users.uj.edu.pl/~ufkapano/algorytmy/`.

[4] *Wstęp do teorii grup* [online],
`http://www.impossible-technologies.eu/down/wstetegr.pdf`.

[5] GAP - Groups, Algorithms, Programming - a System for Computational Discrete Algebra,
`http://www.gap-system.org/`.

[6] Magma Computational Algebra System,
`http://magma.maths.usyd.edu.au/magma/`.

[7] Grupa alternująca [online],
`http://www.edupedia.pl/words/index/show/539157_slownik_matematyczny-grupa_alternujaca.html`.

[8] Frederic W. Byron, Robert W Fuller, *Matematyka w fizyce klasycznej i kwantowej*, Tom 2, Wydanie 2, PWN, Warszawa, 1975.

[9] Donald E. Knuth, *Sztuka programowania*, Tom4, zeszyt 2, WNT, Warszawa, 2007.

[10] Donald E. Knuth, *Efficient representation of perm groups* [online], [dostęp: 1 stycznia 1991],
`http://arxiv.org/abs/math/9201304`.

[11] Andrzej Kapanowski, *Python for education: permutations* [online], [dostęp: 26 lipca 2013],
`http://arxiv.org/abs/1307.7042`.

[12] Logika i teoria mnogości. Grupy [online],
`http://studia.elka.pw.edu.pl/pub/13L/LTM.A/teoria/ltm_wyklad_07.pdf`.

[13] SymPy, a Python library for symbolic mathematics [online],
`http://www.sympy.org/`.

[14] SymPy Combinatorics Module [online],
`http://docs.sympy.org/latest/modules/combinatorics/index.html`.

[15] Akos Seress, *Permutation group algorithms* [online],
`http://catdir.loc.gov/catdir/samples/cam033/2002022291.pdf`