Wstęp do programowania
semestr zimowy 2024/2025

Dr Anna Muranova
UWM w Olsztynie

Wykład 8

Zasady pisania czystego kodu

Kod napisany w Pythonie musi być pisany jak kod w Pythonie. Z wykorzystaniem mechanizmów, które daje sam język i zgodnie z jego założeniami.
Porównuj

```
names = ["Kasia", "Marek", "Ania", "Bartosz"]
for i in range(len(names)):
    name = names[i]
    print("Hi " + name + "!")
```

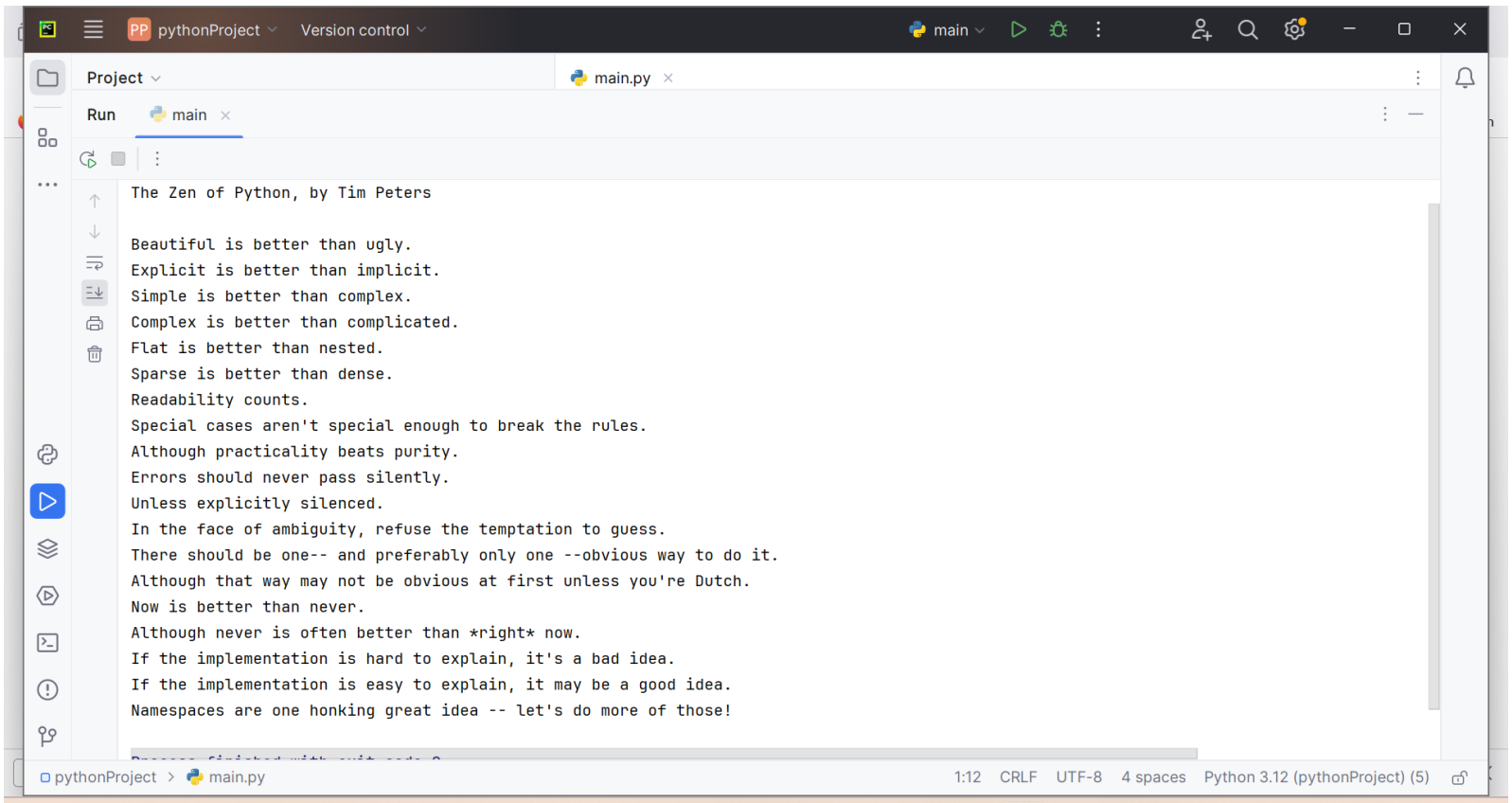
oraz

```
names = ["Kasia", "Marek", "Ania", "Bartosz"]
for name in names:
    print(f"Hi {name}!")
```

Zen of Python

Zen of Python to lista założeń w okół których był tworzony ten język. Co ciekawe, zasady te są dostępne z poziomu samego języka. Spróbuj uruchomić taki plik:

```
import this
```



Zen of Python 1

```
Piękne jest lepsze niż brzydkie.
Wyrażone wprost jest lepsze niż domniemane.
Proste jest lepsze niż złożone.
Złożone jest lepsze niż skomplikowane.
Płaskie jest lepsze niż wielopoziomowe.
Rzadkie jest lepsze niż gęste.
Czytelność się liczy.
Sytuacje wyjątkowe nie są na tyle wyjątkowe, aby łamać reguły.
Choć praktyczność przeważa nad konsekwencją.
Błędy zawsze powinny być sygnalizowane.
Chyba że zostaną celowo ukryte.
W razie niejasności powstrzymaj pokusę zgadywania.
```

Zen of Python 2

```
Powinien być jeden -- i najlepiej tylko jeden --
            oczywisty sposób na zrobienie danej rzeczy.
Choć ten sposób może nie być oczywisty jeśli nie jest się Holendrem.
Teraz jest lepsze niż nigdy.
Chociaż nigdy jest często lepsze niż natychmiast.
Jeśli rozwiązanie jest trudno wyjaśnić, to jest ono złym pomysłem.
Jeśli rozwiązanie jest łatwo wyjaśnić, to może
        ono być dobrym pomysłem.
Przestrzenie nazw to jeden z niesamowicie genialnych pomysłów --
        miejmy ich więcej!
```

Odróżniaj!

Styl kodowania – im określa się nieformalne zbiory reguł.
Standard kodowania – formalny dokument, wdrożony w konkretnym projekcie.
Większość standardów kodowania związana jest z odpowiadającym im językiem programowania.
PEP8 jest stylem kodowania ale może być wdrożony jako standard.
`https://peps.python.org/pep-0008/`

Standardy kodowania języków programowania jak i styl kodowania zazwyczaj obejmują takie aspekty kodu jak:

- **Formatowanie kodu** – szerokość wcięcia, maksymalna długość wiersza, liczba pustych wierszy między kolejnymi definicjami i deklaracjami funkcji bądź klas.
- **Konwencje nazewnicze** – schemat nazywania funkcji, klas, zmiennych, modułów, przestrzeni nazw, plików i tym podobnych.
- **Komentowanie kodu** – sposób komentowania kodu, opisywania zmian, konieczność udokumentowania algorytmów użytych do rozwiązania konkretnego fragmentu kodu.
- **Konstrukcje programistyczne** – zależne od języka programowania, obejmują polecane i zabraniane konstrukcje, wynikające na przykład z ograniczeń platformy docelowej lub użytych narzędzi programistycznych.

Dlaczego potrzebne jest standard kodowania? - 1

Korzystanie ze standardów kodowania:

- zmniejsza koszty związane z konserwacją oprogramowania
- zwiększa jakość kodu poprzez ułatwianie jego zrozumienia przez bardziej doświadczonych deweloperów
- przyśpiesza proces jego restrukturyzacji
- umożliwia deweloperom na płynniejsze korzystanie z narzędzi automatyzacji ich pracy

Standardy kodowania szczególnie wymagane są podczas pracy w dużych instytucjach i projektach programistycznych, w których występuje częsta rotacja kadr.

Dlaczego potrzebne jest standard kodowania? - 2

We wstępie do opisu konwencji kodowania dla języka Java, firma Sun Microsystems wymieniła następujące fakty:

- 80% kosztów oprogramowania wynosi jego utrzymanie,
- jedynie nieliczne systemy są rozwijane ciągle przez jeden, niezmienny zespół programistów,
- konwencje formatowania kodu zwiększają jego czytelność, pozwalając na szybsze wdrożenie nowych inżynierów w prace nad rozwojem oprogramowania,
- jeśli sprzedajesz kod źródłowy jako produkt, musisz upewnić się, że jest on zaparkowany i czytelny tak, jak w przypadku innych produktów.

Styl kodowania

Styl kodowania – mniej lub bardziej sformalizowany zestaw reguł i zaleceń określający, jak powinien wyglądać kod źródłowy programu od strony jego czytelności i wyglądu.

Styl kodowania NIE ma wpływu na sposób interpretacji lub kompilacji programu lecz jest bardzo ważne dla programistów, którzy go rozwijają.

Czytelność poszczególnych zasad jest subiektywna, dlatego nie istnieje jedna, uniwersalna konwencja.

Przyjęte reguły zależą od wybranego języka programowania.

Niektóre wskazówki do pisania czystego kodu w Pythonie 1

- Używaj f-napisów:

```
print(f"You are a fantastic programmer, {first} {middle} {last}")
#You are a fantastic programmer, Sam Douglas Miller.
```

Niektóre wskazówki do pisania czystego kodu w Pythonie 2.1

- funkcje powinny robić tylko jedną rzecz:

```
def print_page(banner_text, banner_images, content, footer_text):
    # Print banner
    print("")
    print("Title:")
    print(banner_text)
    print("")

    # Do some image processing
    for images in banner_images:
        compress(banner_images)
        height, width = banner_images["size"]
        new_image = size(height, width)

    # Print banner image
    render(new_image)
# Print content
print("")
print("")
print(content)
```

Niektóre wskazówki do pisania czystego kodu w Pythonie 2.2

```
# Print footer
footer_length = len(footer_text)
for i in range(0, footer_length):
    print("---------")
print(footer_text)

and so on.....
```

Porownaj z:

```
def print_page(banner_text, banner_images, content, footer_text):
    print_banner(banner_text)
    render_images(banner_images)
    print_content(content)
    print_footer(footer_text)
```

Długość linii

Ponieważ szerokość monitora jest ograniczona, tylko pewna liczba znaków w linii może być jednocześnie widoczna dla programisty. Dlatego stosuje się ograniczenie długości linii kodu. Popularne limity liczby znaków w linii to 80, 78 lub 120.

Nazewnictwo funkcji i zmiennych 1

Każda zmienna oraz funkcja w programie musi posiadać swoją nazwę, która będzie później wykorzystywana przez programistę we wszystkich odwołaniach do niej. Z tego powodu standardy kodowania poświęcają dużo miejsca ujednoliceniu zasad nazewnictwa. Rozpatrywane są zarówno kwestie wyglądu, jak i sensowności poszczególnych nazw. Dominującymi stylami zapisu są:

- podkreślenia (snake case): `to_jest_nazwa`,
- camelCase: `toJestNazwa`,
- PascalCase: `ToJestNazwa`
- wielkie litery: `TO_JEST_NAZWA`.

Pojedynczy standard może wykorzystywać kilka stylów do oznaczania różnych elementów.

Nazewnictwo funkcji i zmiennych 2

PEP8 sugeruje `snake_case` dla funkcji i zmiennych oraz `CamelCase` dla klasów. Nazwy znajdujące się w dużych zakresach powinny być w miarę długie i opisowe, np. vector, `window_with_border` czy `department_number`. Natomiast nazwy należące do niewielkich zakresów powinny być krótkie i konwencjonalne, np. `x`, `i` czy `p`. Pod względem semantycznym reguły uznają przeważnie za niepoprawny kod, w którym nazwy zmiennych i funkcji nie mówią nic o tym, do czego one służą (np. `a`, `b`, `c`, oprócz oczywistych przypadkach). Nazwy powinny być krótkie, lecz znaczące, np. `object_width`. Ponadto dobrze jest, gdy często używane nazwy są krótkie, a rzadziej używane - dłuższe.

Nazwa powinna odzwierciedlać znaczenie nazywanej jednostki, a nie jej implementację. Na przykład nazwa `phone_book` jest lepsza niż `number_vector`, nawet jeśli numery telefoniczne są przechowywane w wektorze.

Nazewnictwo funkcji i zmiennych 3

Nie należy do nazwy dodawać informacji o typie, chociaż zwyczaj taki jest pielęgnowany w językach o dynamicznej i luźnej kontroli typów.

- Określenie typu w nazwie obniża poziom abstrakcji programu. W szczególności uniemożliwia programowanie ogólne (którego podstawą jest możliwość odnoszenia się nazw do jednostek różnych typów).
- Kompilator lepiej radzi sobie z zapamiętywaniem nazw niż człowiek.
- Każdy system skrótów nazw typów, jaki wymyślisz, stanie się kulą u nogi, gdy zaczniesz używać dużej liczby typów do różnych celów.

Choć język na ogół nie jest określany przez standardy kodowania, niepisaną regułą jest wykorzystanie angielskiego zamiast języków narodowych.

Wybór właściwych nazw to sztuka.

Python ma zarezerwowanych 35 słów kluczowych

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	async
def	for	lambda	return	await

Komentarzy

Każdy praktycznie stosowany język programowania zezwala na tworzenie **komentarzy** – fragmenty tekstu, które są pomijane przez interpreter bądź kompilator.

- W komentarzach programista może zapisać słownie dodatkowe informacje na temat działania czy zastosowania określonego kawałka kodu.
- Komentarze są powszechnie wykorzystywane przez narzędzia do automatycznego generowania dokumentacji na podstawie kodu źródłowego. Analizują one komentarze umieszczone nad funkcjami, zmiennymi i klasami, wyciągając z nich opis działania oraz dodatkowe znaczniki zawierające np. opisy argumentów.
- Komentarze wykorzystuje się również do umieszczenia na początku każdego pliku informacji o prawach autorskich oraz licencji, którą objęty jest dany kod.

Komentarze mogą zawierać objaśnienie, co robi dany fragment kodu, uwagi dotyczące jego użycia bądź informacje techniczne dla innych programistów (np. o znalezionych błędach albo pozostałych do zaimplementowania funkcjach).

Pamiętaj o `docstringach`!

Dokumentowanie kodu

- Podstawowym narzędziem opisywania działania kodu są umieszczone w nim komentarze ze słownym opisem w języku naturalnym, których zawartość jest ignorowana przez programy.
- Dokładniejsze dokumentacje mają postać osobnych dokumentów szczegółowo opisujących wszystkie elementy kodu źródłowego w pewien ustandaryzowany sposób. Opis każdego elementu sporządzony jest w języku naturalnym, może zawierać odnośniki do powiązanych elementów i przykłady użycia. Programista pragnący użyć danego elementu, może go szybko odnaleźć w dokumentacji i zapoznać się ze wszystkimi dostępnymi na jego temat informacjami. Pozostałe tematy związane z budową i działaniem kodu źródłowego opracowane są najczęściej w formie klasycznych artykułów.

Istnieje szereg wyspecjalizowanych narzędzi umożliwiających tworzenie dokumentacji bezpośrednio z istniejącego kodu źródłowego, na przykład Doxygen. Dzięki znajomości gramatyki języka programowania potrafią automatycznie określić wiele związków między poszczególnymi elementami. Dodatkowe informacje oraz opis są importowane ze specjalnych komentarzy umieszczonych nad każdym elementem.

Operatory

- Operatory są otoczone zawsze spacjami;
- Można używać nawiasy dla najlepszej czytelności

Zły przykłady 1

- `apples, oranges, fruits = 4, 5, 6`
  Można używać nprz, dla współrzędnych:

```
i, j = 3, 5
(i, j) = (3, 5)  # same as above
n, m = 1, 1
x, y = (0, 0)  # parentheses are optional
```

- `numbers = digits = [0,1,2]`

Zły przykłady 2

- Nie używaj `while` zamiast `for`

```
i = 0
while i < 10:
 do_x()
```

  Trzeba:

```
for i in range(10):
 do_x()
```

- `numbers = digits = [0,1,2]`

Zły przykłady 3

- Nie nadużywaj `try--except` zamiast `if--else`
- Nie nadużywaj `break` w `while`

Dobra wiadomość

Nowoczesny środowiska (takie jak `Visual Studio` oraz `PyCharm`) pomagają w formatowaniu koda źródłowego.
Astyle: `https://astyle.sourceforge.net/`
clang format: `https://clang.llvm.org/docs/ClangFormat.html`
itd.

Na tym kończymy z szczególnościami Python i krotko omówimy algorytmy, tzn logika programowania.

Złożoność algorytmu

Zasoby: pamięć i czas.

- Jeśli znamy kilka algorytmów rozwiązujących pewne zadanie, warto je porównać, aby wybrać najlepszy.
- Jednym z kryteriów służących do porównania algorytmów są zasoby, których potrzebują: ilość pamięci komputera i czas działania.
- Ilość pamięci (czasu) niezbędnej do zrealizowania algorytmu nazywamy jego złożonością pamięciową (czasową).

Złożoność czasowa

- W jakich jednostkach należy mierzyć czas?
- Szybkość wykonania algorytmu nie może zależeć od komputera!
- Dlatego nie możemy mierzyć czasu wykonania algorytmu w jednostkach czasu, np. mikrosekundach.
- Od czego zależy czas wykonania algorytmu, co jest niezależne od szybkości komputera?
- Złożoność czasową mierzymy liczbą operacji potrzebnych do realizacji tego algorytmu. W teorii złożoności nie staramy się być super dokładni. Dlatego w rachunkach uwzględnia się tylko operacje dominujące.
- Operacje dominujące zależą od problemu.
  - Sortowanie liczb: porównanie dwóch elementów
  - Liczenie wartości wielomianu: operacje arytmetyczne

Rozmiar problemu

- Liczba operacji potrzebna do realizacji algorytmu zależy od rozmiaru problemu. (Użycie algorytmu do posortowania 3 liczb wymaga mniej operacji niż do posortowania 1000 liczb.)
- Z każdym problemem wiążemy liczbę naturalną n reprezentującą jego rozmiar.
- Rozmiar problemu zależy od jego natury:
  - W problemie sortowania rozmiarem jest liczba liczb, które sortujemy.
  - W problemie sprawdzenia, czy dana liczba jest liczbą pierwszą rozmiar definiuje się jako liczbę cyfr badanej liczby

Złożoność czasowa algorytmu

Złożonością czasową algorytmu nazywamy liczbę operacji dominujących, które trzeba wykonać, aby rozwiązać problem o rozmiarze n. Zatem złożoność czasową można traktować jako funkcję ze zbioru liczb naturalnych w zbiór liczb naturalnych.

- $T(n) = const$ (złożoność stała)
- $T(n) = \log_2 n$ (złożoność logarytmiczna)
- $T(n) = 2 * n + 2$ (złożoność liniowa)
- $T(n) = 2n^2 - 7$ (złożoność kwadratowa)
- $T(n) = wielomian$ (złożoność wielomianowa)
- $T(n) = 2^n$ (złożoność wykładnicza)

Rzędy wielkości funkcji 1

Niech $f, g : \mathbb{N} \to \mathbb{N}$.
Mówimy, że f jest co najwyżej rzędu g, co zapisujemy

$$f(n) = O(g(n))$$

jeśli istnieją stała rzeczywista $c > 0$ oraz stała naturalna n_0 takie, że nierówność $f(n) \leq c * g(n)$ zachodzi dla każdego $n > n_0$.

Na przykład,

$$n^2 + 2n = O(n^2),$$

bo $n^2 + 2n \leq 3n^2$, dla każdego n.

Rzędy wielkości funkcji 2

Mówimy, ze f jest dokładnie rzędu g, co zapisujemy $f(n) = \Theta(g(n))$ jeśli istnieją stałe rzeczywiste c_1 i c_2 oraz stała naturalna n_0, takie że nierówność $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ zachodzi dla każdego $n > n_0$.

Na przykład, $n^2 + 2n = \Theta(n_2)$, bo $n^2 \leq n^2 + 2n \leq 3n^2$, dla każdego $n \geq n_0$.

Wielomian $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$, gdzie $a_k > 0$ jest dokładnie rzędu n^k.

Złożoność średnia i pesymistyczna

- Złożoność średnia (oczekiwana) – określa złożoność losowego przypadku.
- Złożoność pesymistyczna – określa złożoność najgorszego przypadku.

Przykład: problem wyszukiwania

Dany jest ciąg A liczb całkowitych i liczba całkowita x. Stwierdzić, czy x należy do ciągu.
Możliwości: A jest posortowany, A nie jest posortowany.
Warianty tego problemu: Podać adres w tablicy pierwszego wystąpienia x. Jeśli x nie występuje zwrócić -1. Znaleźć wszystkie wystąpienia x w A.

Wyszukiwanie liniowe Algorytm 1 (naiwny)

```
listA = [1,2,3,4,5,6,7,8,9,2,4,9,6,5,3,4,2,10,5,30]
znaleziono = False;
x = int(input("Co szukamy? "))
for i in listA:
if x==i: znaleziono = True
if znaleziono: print("tak")
else: print("nie")
```

Algorytm wykonuje n poleceń.

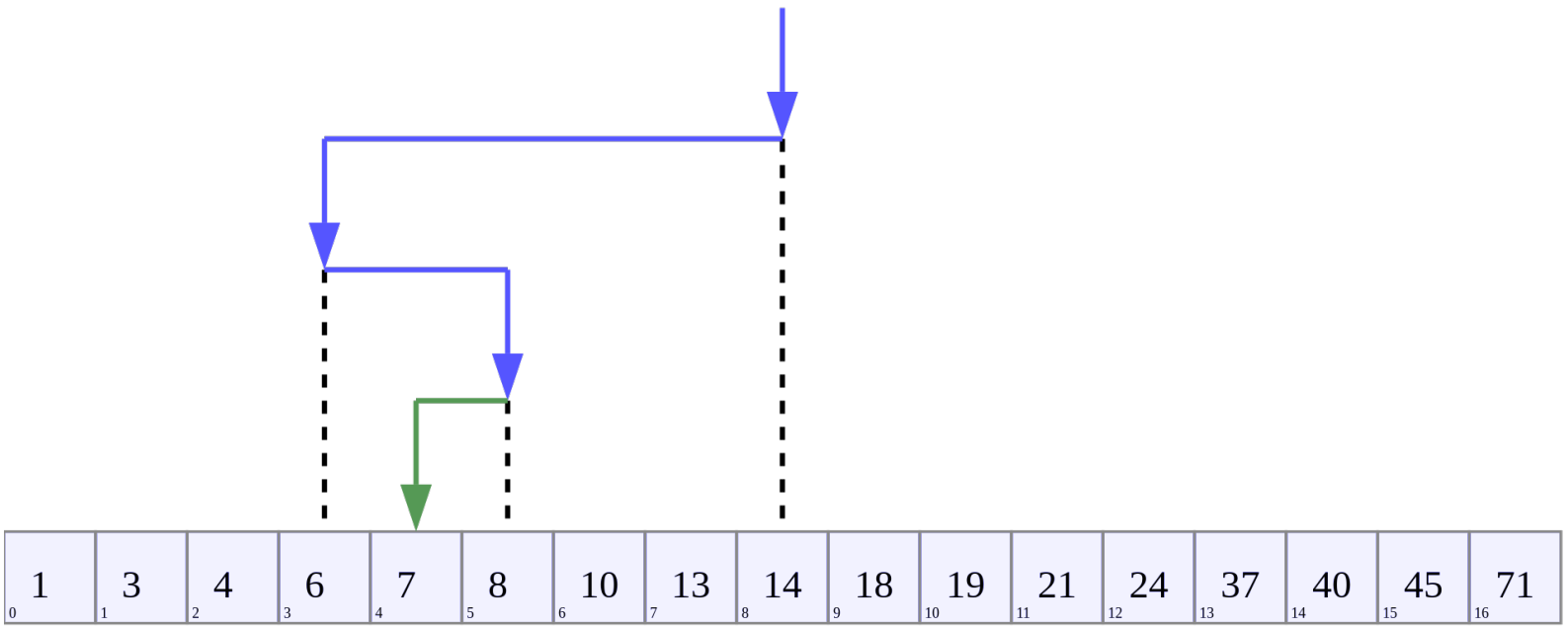
Wyszukiwanie liniowe Algorytm 2

```
listA = [1,2,3,4,5,6,7,8,9,2,4,9,6,5,3,4,2,10,5,30]
znaleziono = False;
x = int(input("Co szukamy? "))
for i in listA:
    if x==i:
        znaleziono = True
        break
if znaleziono: print("tak")
else: print("nie")
```

Algorytm wykonuje n poleceń w pesymistycznym przypadku, ale wykonuje $n/2$ poleceń średnie.

Czyli złożoność $\Theta(n)$.

Wyszukiwanie binarne (w uporządkowanej tablice)



Wyszukiwanie binarne (w uporządkowanej tablice)

```
listA = [1,2,3,4,5,6,7,8,9,10,12,13,
14,50,60,70,80,90,200,300,400]
i=0
j=len(listA)-1
znaleziono = False
x = int(input("Co szukamy?"))
while not znaleziono and j>=i:
    k=(i+j)//2
    if listA[k]==x: znaleziono = True
    elif listA[k]<x: i=k+1
    else: j=k-1

if znaleziono: print("tak")
else: print("nie")
```

Złożoność (pesymistyczna) wyszukiwania binarnego

- Jaka jest maksymalna liczba porównań dla tablicy n-elementowej?
- Każde porównanie skraca długość tablicy o połowę.
- Proces ten kończy się gdy znajdziemy element lub tablica jest pusta.

Pesymistyczna liczba porównań jest rzędu $\Theta(log_2 n)$.

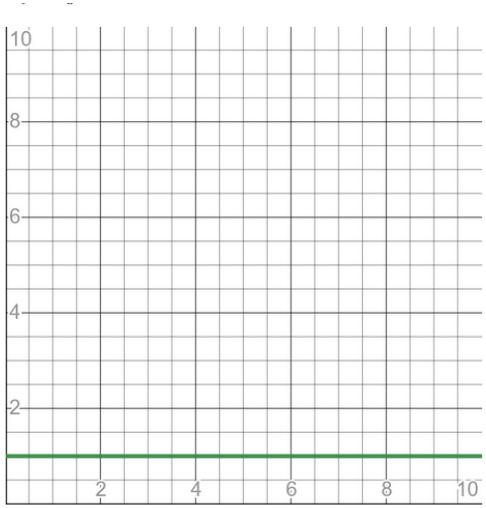
n	$log_2 n$
10	3
100	6
1000	9
1000000	19
1000000000	29
10^{18}	59

Często spotykane złożoności obliczeniowe

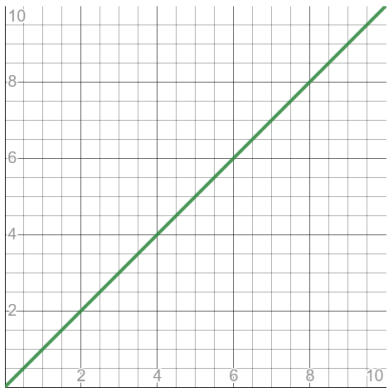
- **Stała złożoność obliczeniowa** $\Theta(1)$;
- **Złożoność liniowa** $\Theta(n)$;
- **Złożoność logarytmiczna** $\Theta(\log n)$;
- $\Theta(n * \log n)$ (naprz. sortowanie przez łączenie);
- **Złożoność wielomianowa** $\Theta(n^k)$;

- **Złożoność wykładnicza** $\Theta(k^n)$;
- $\Theta(n!)$.

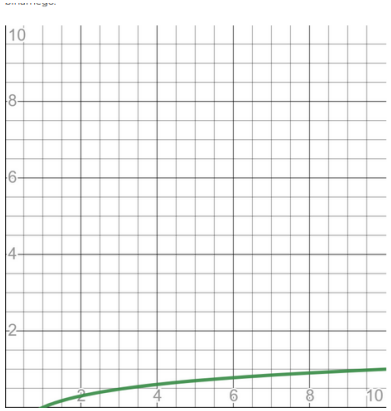
Często spotykane złożoności obliczeniowe



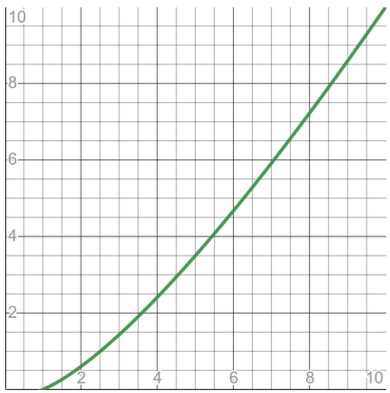
(a) Stała złożoność obliczeniowa $\Theta(1)$



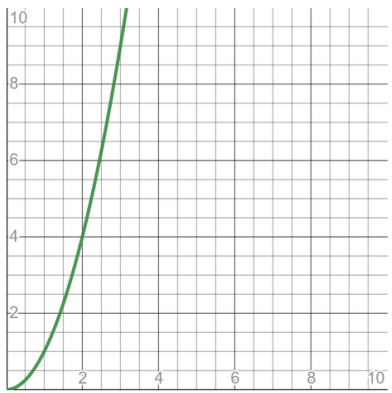
(b) Złożoność liniowa $\Theta(n)$



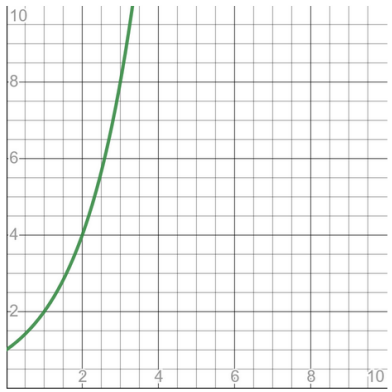
(c) Złożoność logarytmiczna $\Theta(\log n)$



(a) $\Theta(n * \log n)$

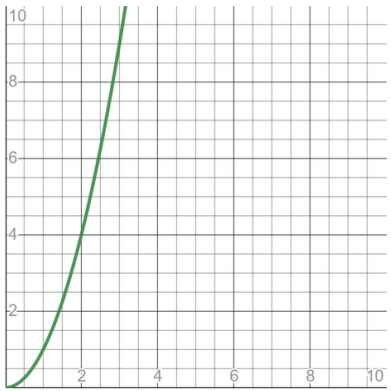


(b) Złożoność wielomianowa $\Theta(n^k)$

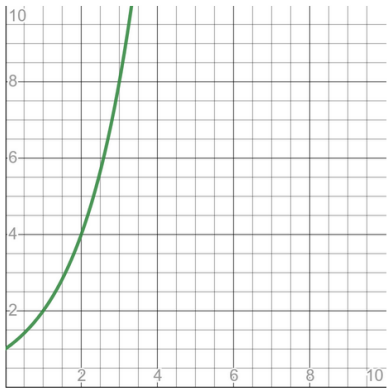


(c) Złożoność wykładnicza $\Theta(k^n)$

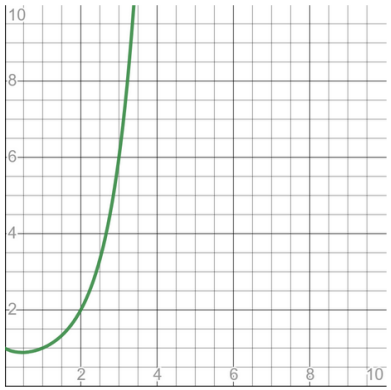
Często spotykane złożoności obliczeniowe



(a) Złożoność wielomianowa $\Theta(n^k)$



(b) Złożoność wykładnicza $\Theta(k^n)$



(c) $\Theta(n!)$

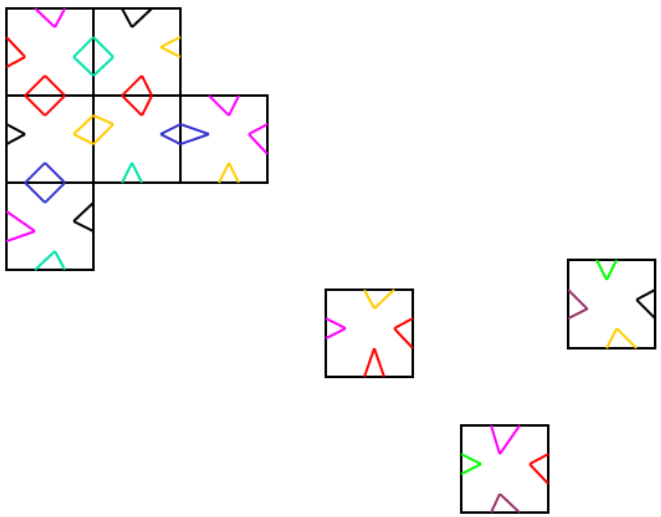
Zapotrzebowanie na czas

Funkcja	N=10	N=20	N=50	N=100	N=300
N^2	1/10000 sekundy	1/2500 sekundy	1/400 sekundy	1/100 sekundy	9/10 sekundy
N^5	1/10 sekundy	3.2 sekundy	5.2 minuty	2.8 godziny	28.1 dnia
2^N	1/1000 sekundy	1 sekunda	35.7 lat	$400*10^9$ stuleci	75-cyfrowa liczba stuleci
N^N	2.8 godziny	3.3 biliony lat	70-cyfrowa liczba stuleci	185-cyfrowa liczba stuleci	728-cyfrowa liczba stuleci

1 instrukcja = 1 milisekunda

Od wielkiego wybuchu minęło 15 miliardów lat

Układanka



- Zakładamy, że mamy układankę 5×5
- Zakładamy, że każdy kwadrat ma ustalony kierunek góra-dół i prawo-lewo, zatem nie musimy kwadratów obracać.
- Chcemy stwierdzić, czy można ułożyć dany zestaw 25 kwadratów.

Rozwiązanie naiwne: sprawdzamy wszystkie możliwe układy. (Metoda ta wymaga, posiadania procedury generowania kolejnych układów, aby się nie powtarzały).

- Ile wynosi liczba wszystkich możliwych ułożeń 25 kwadratów?
- $25 * 24 * 23 * \ldots * 2 * 1 = 25!$
- Liczba 25 składa się z 26 cyfr.
- Jeśli założymy, że nasz komputer będzie realizował milion ułożeń na sekundę, to przejrzenie wszystkich ułożeń zajmie ?

ponad 400 milionów lat!!!

Znajdowanie szybszych algorytmów jest bieżącym tematem badawczym.

Złożoność problemu

Złożoność problemu – złożoność algorytmu o minimalnej złożoności rozwiązującego ten problem

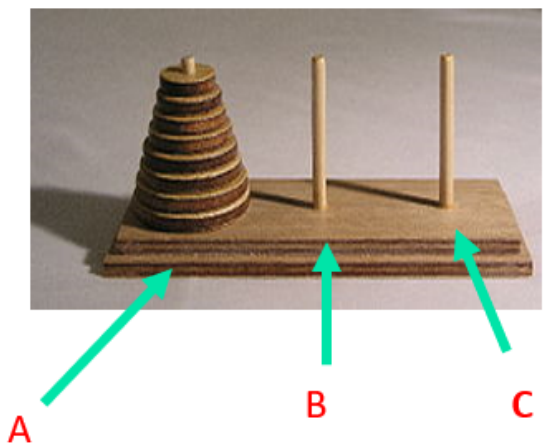
- Problem sortowania: $\Theta(n \log_2 n)$
- Problem wieży z hanoi: $\Theta(2^n)$

Istnieje bardzo wiele problemów, których złożoność jest nieznana!

- **Problemy łatwo rozwiązywalne** – problemy o złożoności wielomianowej.
- **Problemy trudno rozwiązywalne** – problemy o złożoności ponadwielomianowej.

Pułapka – pewne problemy łatwo rozwiązywalne mogą się w praktyce okazać gorsze niż trudno rozwiązywalne (przynajmniej teoretycznie)!!

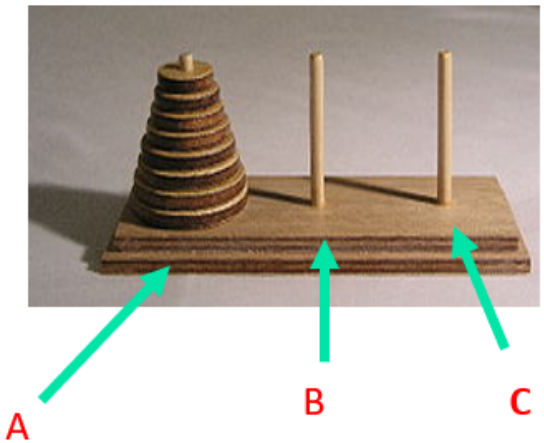
Wieże w Hanoi



Od lewej: słupek A z całą wieżą,
pusty słupek B pełniący rolę bufora
i pusty słupek docelowy C

Zadanie: Przenieść krążki z A na C, posługując się słupkiem B. Nie wolno kłaść większego krążka na mniejszym.

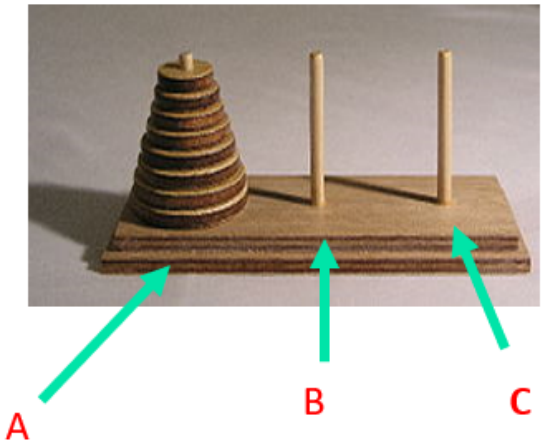
Wieży z hanoi: rozwiązanie rekurencyjne



- Przenieś (rekurencyjnie) $n-1$ krążków ze słupka A na słupek B, posługując się słupkiem C.
- Przenieś jeden krążek (największy) ze słupka A na słupek C.
- Przenieś (rekurencyjnie) $n-1$ krążków ze słupka B na słupek C, posługując się słupkiem A.

Łatwo obliczyć złożoność jako $\Theta(2^n)$.

Wieży z hanoi: rozwiązanie iteracyjne



- Przenieś najmniejszy krążek na kolejny (*) słupek.
- Wykonaj jedyny możliwy do wykonania ruch, nie zmieniając położenia krążka najmniejszego.
- Powtarzaj punkty 1 i 2, aż do odpowiedniego ułożenia wszystkich krążków.

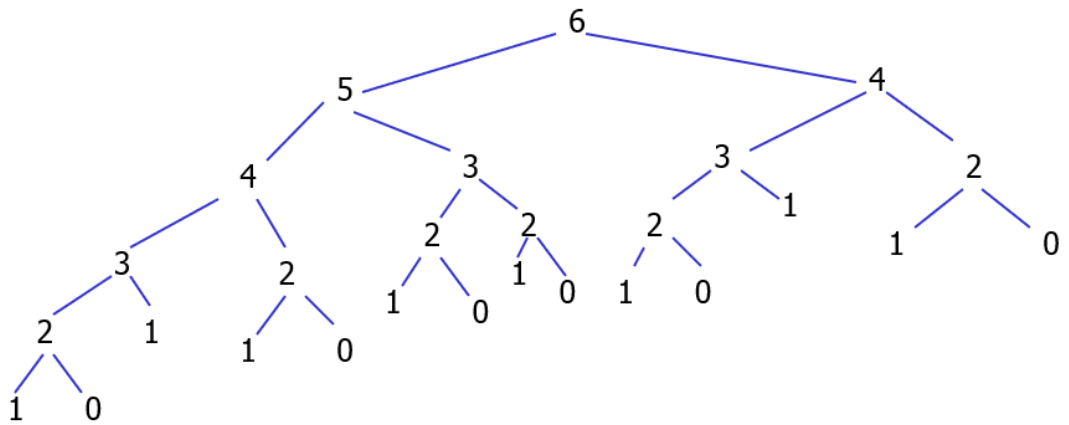
Złożoność algorytmu $\Theta(2^n)$ (bez dowodu).

(*)Kolejny słupek wyznaczany w zależności od liczby krążków. Parzysta – po prawej stronie, nie parzysta — po lewej.

Liczby Fibonacc'ego

Rekurencyjne

```
def Fib(n):
    if n < 2:
        return n
    else:
        return Fib(n-1) + Fib(n-2)
```



Złożoność (dodawania): $\Theta(2^n)$

Liczby Fibonacc'ego

Iteracyjne

```
f0 = 0
f1 = 1

for i in range(2,n+1):
    f = f1
    f1 = f1 + f0
    f0 = f
```

Złożoność (dodawania): $\Theta(n)$

Klasy P oraz NP

Tutaj mówimy TYLKO o problemach decyzyjnych (TAK/NIE).

Teoria złożoności obliczeniowej kategoryzuje problemy decyzyjne w zależności od tego jak trudno jej rozwiązać (najefektywniejszym algorytmem)

Problem P – problem decyzyjny, dla którego rozwiązanie można **znaleźć** w czasie wielomianowym.

Problem NP – problem decyzyjny, dla którego rozwiązanie TAK można **sprawdzić** w czasie wielomianowym.
Wszystkie problemy klasy P są NP.

Problem milenijny: czy $P = NP$?

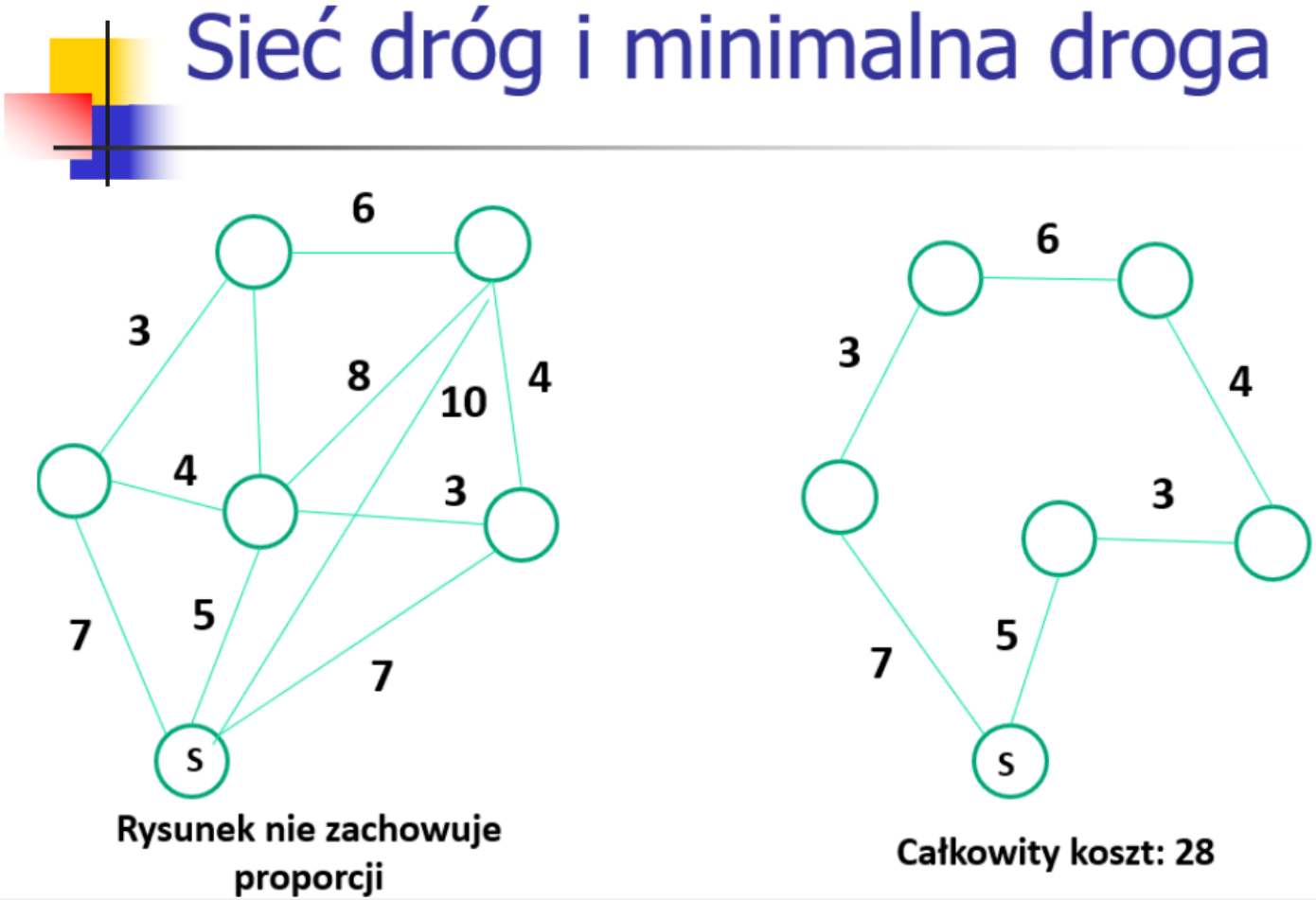
Problemy $NP-$zupełne

Tutaj mówimy TYLKO o problemach decyzyjnych (TAK/NIE).
Każdy problem NP-zupełny charakteryzuje się następującymi własnościami:

- Znane jest jego rozwiązanie wykładnicze
- Nie wiadomo, czy istnieje rozwiązanie wielomianowe
- Jeśli posiada rozwiązanie wielomianowe, to wszystkie inne problemy NP-zupełne też posiadają takie rozwiązanie (jeśli A i B są dowolnymi problemami NP-zupełnymi, to A można w czasie wielomianowym sprowadzić do B).

Problem komiwojażera (wersja decyzyjna)

Dany jest zbiór n miast wraz z odległościami między nimi. Komiwojażer chce odwiedzić wszystkie miasta, każde dokładnie raz, i powrócić do punktu wyjścia. Dla danej liczby naturalnej k stwierdzić, czy istnieje trasa komiwojażera krótsza od k.



Problem komiwojażera

- Nie jest znana złożoność problemu komiwojażera.
- Znane algorytmy rozwiązujące ten problem mają złożoność wykładniczą.
- Charakterystyczną własnością problemu komiwojażera jest łatwość potwierdzenia rozwiązania: jeśli odpowiedź brzmi „tak”, łatwo jest o tym kogoś przekonać podając potwierdzenie zawierające dowód tego faktu.

Ciekawostka

Problem Hamiltona i problem Eulera

- Ścieżka Hamiltona – droga w grafie przechodząca przez każdy wierzchołek (miasto) dokładnie raz. Problem Hamiltona – czy w danym grafie istnieje ścieżka Hamiltona?
- Ścieżka Eulera – droga w grafie przechodząca przez każdą krawędź dokładnie raz. Problem Eulera – czy w danym grafie istnieje ścieżka Eulera?

Problem Hamiltona jest problemem *NP*–zupełnym. Problem Eulera posiada algorytm wielomianowy (graf musi mieć 0 albo 2 wierzchołka parzystego stopnia)!!!