Matura 2021

Repetytorium opracowane przez Macieja Kaszkowiaka (maciej@kaszkowiak.org).

Spis treści

Informatyka – algorytmy

Algorytmy sortowania

Sortowanie bąbelkowe (bubble sort)

Złożoność czasowa **O(n^2)**, pamięciowa **O(1)**

Polega na porównywaniu i zamiany par elementów aż do uzyskania posortowanej tablicy.

```
T = [6, 2, 0, 3, 2.7, 1.5, 5, 2, 1, 0, 7, 1, 9, 9, 4]

size = len(T)
for iteration in range(size):
        unsorted_max = size - iteration
        # iterujemy po nieposortowanej czesci tablicy
        for pair in range(unsorted_max - 1):
                # unsorted_max - 1, aby uwzglednic drugi element
                if T[pair] > T[pair + 1]:
                        # zamiana elementow pary
                        T[pair], T[pair + 1] = T[pair + 1], T[pair]

print(T)
```

Sortowanie przez wybór (selection sort)

Złożoność czasowa **O(n^2)**, pamięciowa **O(1)**

Polega na wyszukiwaniu co iterację najmniejszego elementu oraz zamianą z kolejnymi miejscami w tablicy.

Dla tablicy [6, 2, 0, 3, 5]:

[6, 2, 0, 3, 5] min = 0, pozycja = 0

[0, 2, 6, 3, 5] min = 2, pozycja = 1 (nie trzeba zamieniać, jest na miejscu)

[0, 2, 6, 3, 5] min = 3, pozycja = 2

[0, 2, 3, 6, 5] min = 4, pozycja = 3

[0, 2, 3, 5, 6] tablica posortowana

```
T = [6, 2, 0, 3, 2.7, 1.5, 5, 2, 1, 0, 7, 1, 9, 9, 4]

size = len(T)
for replacing_index in range(size):
        min_index = replacing_index
        # szukamy najmniejszego elementu z nieposortowanej części T
        for check_index in range(replacing_index+1, size):
                if T[check_index] < T[min_index]:
                        min_index = check_index

        # zamieniamy miejscami najmniejszy i wybrany do zamienienia
        T[replacing_index], T[min_index] = T[min_index], T[replacing_index]

print(T)
```

Sortowanie przez wstawianie (insertion sort)

Złożoność czasowa **O(n^2)**, pamięciowa **O(1)**

Mając nieposortowaną tablicę T:

- tworzymy tablicę A do posortowanych elementów
- wstawiamy pierwszy element z tablicy T do tablicy A

Następnie dla każdego elementu z tablicy T:

- iterujemy po kolei przez A, dopóki nie znajdziemy większego elementu lub tablica się nie skończy – wtedy w te miejsce wstawiamy nasz element

```
T = [6, 2, 0, 3, 5, 2, 1, 8]
A = []

A.append(T.pop(0))

while T:
        element = T.pop(0)
        docelowy_index = len(A)

        for index, value in enumerate(A):
                if value > element:
                        docelowy_index = index
                        break

        A.insert(docelowy_index, element)

print(A)
```

Sortowanie przez scalanie (merge sort)

Złożoność czasowa **O(n * log(n))**, pamięciowa **O(n)**

Algorytm funkcji sortującej tablicę:

- jeśli tablica jest jednoelementowa, zwracamy ją
- dzielimy tablicę na dwie połowy
- sortujemy połowy rekursywnie (przez scalanie)
- zwracamy połączone połowy funkcją łączącą

Algorytm funkcji łączącej tablice L i R:

- tworzymy tablicę A do posortowanych elementów
- dopóki L i R mają elementy:
    - porównujemy ich pierwsze elementy
    - usuwamy z tablicy mniejszy i dodajemy do tablicy A

Następnie musimy dodać pozostałe niedodane elementy:

- dodajemy elementy tablicy L na koniec tablicy A
- dodajemy elementy tablicy R na koniec tablicy A
- zwracamy tablicę A

```
T = [6, 2, 0, 3, 2.7, 1.5, 5, 2, 1, 0, 7, 1, 9, 9, 4]

def mergesort(array):
        size = len(array)
        if size <= 1:
                return array

        middle = size // 2
        L = array[0:middle]
        R = array[middle:size]

        L, R = mergesort(L), mergesort(R)
        return merge(L, R)

def merge(L, R):
        A = []
        while L and R:
                if L[0] > R[0]:
                        # element z R jest mniejszy
                        A.append(R.pop(0))
                else:
                        # element z L jest mniejszy
                        A.append(L.pop(0))

        A.extend(L) # laczymy pozostale elementy
        A.extend(R)
        return A

print(mergesort(T))
```

Sortowanie szybkie (quicksort)

Złożoność czasowa: najlepiej **O(n*log(n)),** najgorzej **O(n^2)**

Złożoność pamięciowa: najlepiej **O(log(n)),** najgorzej **O(n)**

Algorytm funkcji sortującej tablicę T:

- jeśli tablica jest jednoelementowa, zwracamy ją
- wybieramy element rozdzielający (*pivot*), np. środkowy lub losowo wybrany
- tworzymy puste tablice MNIEJ, ROWNE, WIECEJ
- iterujemy po tablicy T - przyrównujemy elementy do pivota i umieszczamy w odpowiedniej tablicy
- sortujemy tablice MNIEJ i WIECEJ
- zwracamy złączone tablice MNIEJ + ROWNE + WIECEJ

Złożoność algorytmu zależy od wyboru elementu rozdzielającego (*pivota)* – najlepsza jest mediana, która rozdziela tablicę na dwie równe części. Najmniej wydajna jest wartość minimalna/maksymalna. W mojej implementacji wybieram środkowy element tablicy.

```
T = [6, 2, 0, 3, 2.7, 1.5, 5, 2, 1, 0, 7, 1, 9, 9, 4]

def quicksort(array):
        size = len(array)
        if size <= 1:
                return array

        middle = size // 2
        pivot = array[middle]

        smaller, equal, bigger = [], [], []
        for element in array:
                if element > pivot:
                        bigger.append(element)
                elif element == pivot:
                        equal.append(element)
                else:
                        smaller.append(element)

        smaller, bigger = quicksort(smaller), quicksort(bigger)
        sorted_array = smaller + equal + bigger

        return sorted_array


print(quicksort(T))
```

Sortowanie kubełkowe (bucket sort)

Złożoność czasowa: najlepiej **O(n),** najgorzej **O(n^2)**, złożoność pamięciowa: **O(n)**

Przy sortowaniu kubełkowym elementy powinny być równomiernie rozłożone.

Mając nieposortowaną tablicę T z zakresem wartości od MIN do MAX:

- tworzymy N kubełków (zależy od rozkładu danych) w postaci pustych tablic

Przykładowo dla N=10 i zakresu wartości od -20 do 180:

Kubełek #1 będzie przyjmował wartości w przedziale <-20, -10)

Kubełek #2 będzie przyjmował wartości w przedziale <-10, 0)

...Kubełek #10 będzie przyjmował wartości w przedziale <170, 180>

- każdy element z tablicy T przydzielamy do odpowiedniego kubełka
- sortujemy poszczególne kubełki (dowolnym algorytmem)

Teraz wystarczy zebrać wszystkie elementy:

- tworzymy pustą tablicę A do posortowanych elementów
- dodajemy po kolei elementy kubełków do A

```
import math
T = [6, 2, 0, 3, 2.7, 1.5, 5, 2, 1, 0, 7, 1, 9, 9, 4]
A = []

N = 10  # ilosc kubelkow
MAX, MIN = max(T), min(T)  # skrajne wartosci
kubelki = [[] for i in range(N)]

# przeskok pomiedzy kubelkami
# (N-1) - poniewaz np. min-max 5-15 i N=3
przeskok = (MAX-MIN) / (N-1)
for element in T:
        n_kubelka = (element - MIN) / przeskok
        n_kubelka = math.ceil(n_kubelka)

        kubelki[n_kubelka].append(element)

print(kubelki) # demonstracyjnie
for n_kubelka in range(N):
        posortowany_kubelek = sorted(kubelki[n_kubelka])
        A.extend(posortowany_kubelek)

print(A)
```

Algorytmy na liczbach całkowitych

Reprezentacja liczb w dowolnym systemie pozycyjnym (base conversion)

Zamiana z systemu dziesiętnego

Zamiana liczb jest prosta, gdy znamy działanie systemów pozycyjnych. Liczbę 10456 (base10) można przedstawić jako 1 * 10 000 + 0 * 1 000 + 4 * 100 + 5 * 10 + 6 * 1. Pozostałe systemy funkcjonują w identyczny sposób – przykładowo, liczbę 1E9 (base16) można przedstawić jako 1 * 256 + 14 * 16 + 9 * 1.

Dlaczego E = 14? Ponieważ E jest 14. znakiem wykorzystywanym w alfabecie opisującym base16: „0123456789ABCDEF”. Dlaczego 1, 16, 256? Ponieważ to kolejne potęgi liczby 16 (16^0, 16^1, 16^2).

Tym samym, przy zamianie sprawdzamy ile w liczbie znajduje się 16^0, następnie 16^1, 16^2, ... aż do wyczerpania liczby. Ilość zamieniamy na odpowiedni znak z alfabetu [np. nie mamy znaku 10, mamy znak A]

Dla liczby N zapisanej w systemie w dziesiętnym i zamianie na base16:

- tworzymy zmienną W przechowującą wynik
- dopóki N istnieje:
  - niech M będzie wynikiem N % 16
  - dodaj na początku W znak występujący w alfabecie na pozycji M
  - wykonaj dzielenie całkowite N przez 16

Przy innych systemach pozycyjnych (base16, base4, base8, ...) zamieniamy 16 na bazę wybranego systemu (16, 4, 8, ...).

```
import string

alfabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# alternatywnie:
alfabet = string.digits + string.ascii_uppercase

def dziesietny_do_dowolnego(N, system):
        wynik = ""
        while N:
                znak = alfabet[N % system]
                # znak dodajemy na początku,
                #   ponieważ iterujemy liczbę z prawej do lewej
                #   tj. od najmniejszych do największych:
                # znaki jedności -> dziesiątki -> setki, ...
                #
                #   a liczba musi mieć format:
                # setki -> dziesiątki -> znaki jedności, ...
                wynik = znak + wynik
                N = N // system

        return wynik

print(dziesietny_do_dowolnego(65152, 16))
```

Zamiana na system dziesiętny

Zamiana na system dziesiętny jest jeszcze prostsza. Mając zmienną N przechowującą naszą liczbę w base16, iterujemy ją co znak od prawej do lewej. Co iterację dodajemy do wyniku kolejną potęgę 16.

Przykładowo, mając 1E9:

- odwracamy liczbę (1E9 zamieniamy na 9E1)
- tworzymy zmienne wykładnik = 0, wynik = 0

Dla każdego znaku:

- odszukujemy wartość znaku w alfabecie (np. E -> 14, 9 -> 9)
- do wyniku dodajemy (wartość znaku * 16 ^ wykładnik)
- podnosimy wartość wykładnika potęgi

Wynik to 9 * 16^0 + 14 * 16^1 + 1 * 16^2.

```
import string

alfabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# alternatywnie:
alfabet = string.digits + string.ascii_uppercase

def dowolny_do_dziesietnego(N, system):
        wynik = 0
        wykladnik = 0
        # iteruje od prawej do lewej odwracajac string
        for znak in N[::-1]:
                wartosc = alfabet.find(znak)
                wynik += wartosc * system**wykladnik
                wykladnik += 1
        return wynik

print(dowolny_do_dziesietnego("65152", 10))
```

Sprawdzanie, czy liczba jest liczbą pierwszą (prime number check)

Złożoność czasowa: **O(√N),** złożoność pamięciowa: **O(1)**

Rozwiązanie, które się nasuwa to sprawdzenie dzielników od 2 do N/2. Istnieje równie proste, ale znacznie bardziej optymalne rozwiązanie:

Dla liczby N wystarczy sprawdzić dzielniki od 2 do √N. Liczba złożona (nie pierwsza) będzie zawierać dzielnik w tym przedziale. Dlaczego? Przyjmijmy hipotetycznie, że istnieje liczba, która posiada dwa dzielniki A i B, które są większe od √N. W tym przypadku ich iloczyn (A*B) będzie większy od N (√N * √N) – absurd! Dlatego nawet jeśli liczba złożona posiada dzielniki większe od √N, musi mieć również dzielnik mniejszy od √N.

Jako przykład weźmy liczbę 407 (37*11). Testując jedynie w zakresie <2, 20> zamiast <2, 203> również udowodnimy, że liczba 407 nie jest pierwsza (znajdziemy 11, nie trzeba szukać 37). Już przy tak małej liczbie widać różnicę w zaoszczędzonych cyklach procesora.

```
import math

def prime(liczba):
        if liczba < 2:
                return False

        max_dzielnik = math.sqrt(liczba)
        # range() przyjmuje inty
        max_dzielnik = int(max_dzielnik)+1
        # +1, aby range() iterowało włącznie z max_dzielnik
        for dzielnik in range(2, max_dzielnik):
                if liczba % dzielnik == 0:
                        # dzieli się, więc nie jest pierwsza
                        return False

        return True
```

Wypisywanie N liczb pierwszych (prime number generation)

Złożoność czasowa: **O(n * log(log(n))),** złożoność pamięciowa: **O(n)**

Najprostsze rozwiązanie to sprawdzenie, czy każda liczba od 2 do N jest pierwsza.

Istnieje bardziej optymalne rozwiązanie, zwane Sitem Eratostenesa.

- tworzymy zbiór T indeksowany od 2 do N, każdej liczbie przypisujemy True (czy jest pierwsza)
- ze zbioru wybieramy najmniejszą liczbę pierwszą P, czyli T[P] = True
    - wielokrotnościom P przypisujemy False (nie są liczbami pierwszymi)
- gdy P będzie większe od √N przerywamy wykreślanie

Uzasadnienie dla √N zapisałem przy algorytmie sprawdzającym pierwszość liczby.

```
import math

N = 1000
liczby_pierwsze = {}
# N+1 aby range() było włącznie
for i in range(2, N+1):
        liczby_pierwsze[i] = True

P = 2
P_MAX = math.sqrt(N)
while P <= P_MAX:
        # np. dla P=2 iterujemy od 4 do N, przeskakujac co 2
        for wielokrotnosc in range(P*2, N+1, P):
                liczby_pierwsze[wielokrotnosc] = False

        # szukamy nastepnej pierwszej od P+1 do N
        for nastepne_P in range(P+1, N+1):
                if liczby_pierwsze[nastepne_P]:
                        P = nastepne_P
                        break

for liczba, pierwszosc in liczby_pierwsze.items():
        if pierwszosc:
                print(liczba, end=", ")
```

Sprawdzanie, czy liczba jest liczbą doskonałą (perfect number check)

Liczba doskonała to liczba naturalna, która jest sumą wszystkich swoich dzielników właściwych (czyli z wykluczeniem jej samej). Przykładowo:

[doskonała] **6** ma dzielniki {1, 2, 3, 6}, a 1+2+3 = **6**.

[doskonała] **28** ma dzielniki {1, 2, 4, 7, 14, 28}, a 1+2+4+7+14 = **28**.

[niedoskonała] **15** ma dzielniki {1, 3, 5, 15}, a 1+3+5 = **9**.

Tutaj również wystarczy, że sprawdzimy dzielniki od 2 do √N. Dla każdego dzielnika dodajemy również odpowiadający mu dzielnik – np. dla liczby 28 znajdziemy 2 i 4, więc dodajemy od razu 14 i 7. Trzeba zwrócić uwagę, aby nie powtórzyć dodawania przy liczbie kwadratowej, np. 25 – znajdziemy 5, więc nie dodajemy kolejnej 5 do sumy.

Nieco prostszym, lecz mniej wydajnym sposobem jest zwykła iteracja od 1 do N/2.

```
import math

def jest_doskonala(N):
        MAX_DZIELNIK = int(math.sqrt(N))

        suma = 1 # uwzględniamy jedynkę jako dzielnik
        for dzielnik in range(2, MAX_DZIELNIK+1):
                if N % dzielnik == 0:
                        suma += dzielnik
                        # sprawdzamy, czy nie dodajemy tego samego dzielnika
                        if (N / dzielnik) != dzielnik:
                                suma += (N / dzielnik)

        return suma == N

print(jest_doskonala(6))
print(jest_doskonala(28))
print(jest_doskonala(15))
```

Rozkładanie liczby na czynniki pierwsze

Sam proces rozkładania liczby jest dość prosty, 25 = 5 * 5, 75 = 3 * 5 * 5, itp.

Algorytm z iteracją od 1 do N jest również prosta (dzielimy aż do uzyskania 1), kod jest samowyjaśniający:

```
import math

def czynniki(N):
        czynniki = [1]

        for dzielnik in range(2, N+1):
                while N % dzielnik == 0:
                        czynniki.append(dzielnik)
                        N /= dzielnik

        return czynniki
```

Otrzymujemy w tym przypadku listę – dla 25 uzyskamy [1, 5, 5], dla 75 uzyskamy [1, 3, 5, 5].

Istnieje szybsza metoda – iterujemy od 2 do √N analogicznym algorytmem. Jest drobna zmiana - po zakończonej iteracji dodajemy wartość N do znalezionych czynników.

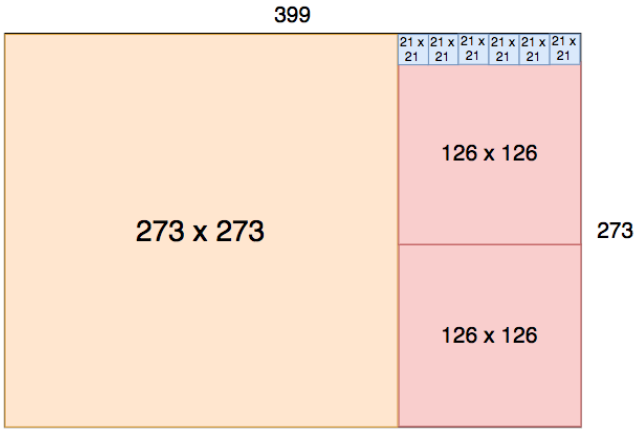
Dlaczego? Przyjmijmy liczbę 8914 (2*4457) za przykład – będziemy mogli jedynie podzielić przez 2. Wartość N po zakończonej iteracji będzie wynosić 4457, czyli nasz ostatni czynnik. Przy liczbie pierwszej 197 nie podzielimy przez nic, jedyny czynnik to jej wartość, czyli N. Przy liczbie 25 (5*5) w zmiennej N zostanie nam nieszkodliwe 1.

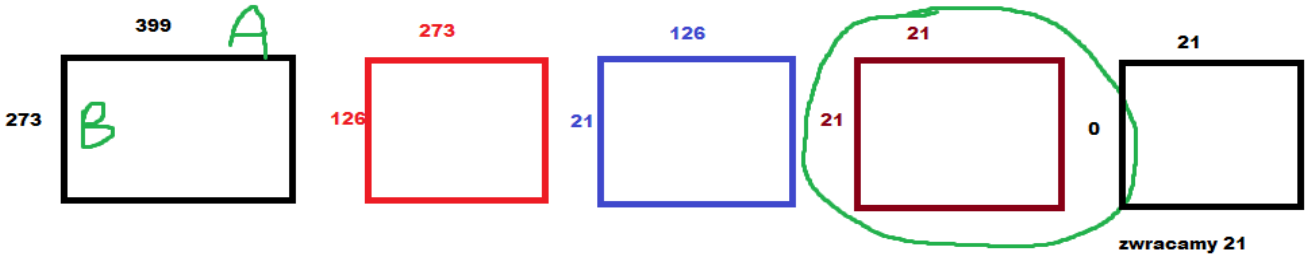
```
import math

def czynniki(N):
        czynniki = [1]
        max_dzielnik = int(math.sqrt(N))

        for dzielnik in range(2, max_dzielnik+1):
                while N % dzielnik == 0:
                        czynniki.append(dzielnik)
                        N /= dzielnik

        # nie ma sensu dodawać już występującej jedynki
        if N != 1:
                czynniki.append(int(N))

        return czynniki
```

Algorytm Euklidesa (NWD) – iteracyjna i rekurencyjna wersja (GCD)

[1]

Algorytm Euklidesa służy do wyznaczania największego wspólnego dzielnika (NWD) dwóch liczb. Do tego algorytmu załączam graficzne wytłumaczenie – celem jest znaleźć największy kwadrat, który wypełniłby dany prostokąt.





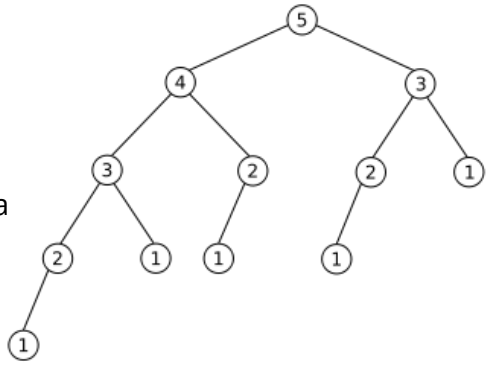

```
def NWD_iteracyjnie(a, b):
        while b:
                a, b = b, a%b
        return a

def NWD_rekurencyjnie(a, b):
        if b:
                return NWD_rekurencyjnie(b, a%b)
        return a

print(NWD_iteracyjnie(399, 273))
print(NWD_rekurencyjnie(399, 273))
```

[1] Grafika: „Extended Euclidean Algorithm — Number Theory", data dostępu: 2021-02-28

Ciąg Fibonacciego – iteracyjna i rekurencyjna wersja

Ciąg Fibonacciego to następujący ciąg: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
Jego pierwszy i drugi wyraz mają wartość 1, kolejne są sumą poprzednich dwóch wyrazów ciągu:
1+2 = **3**, 2+3 = **5**, 3+5 = **8**, ...

Niektórzy zaliczają również 0 do elementów ciągu (jako element zerowy), wtedy element drugi jest nadal równy 0+1. Funkcję F(n) zwracającą wyraz ciągu Fibonacciego możemy tym samym określić jako:
F(0) = 0
F(1) = 1
F(n) = F(n – 1) + F(n – 2)



Wersja rekurencyjna nasuwa się sama, lecz jest to dość niewydajny sposób – wymaga policzenia tej samej wartości wielokrotnie (załączona grafika to wizualizuje). Istnieją oczywiście usprawnienia, lecz nie jest to materiał na maturę. Dla zainteresowanych: Efficient calculation of Fibonacci series na Stack Overflow.

```
def F(n):
        if n <= 0:
                return 0
        if n == 1:
                return 1

        return F(n-1) + F(n-2)
```

Wersja iteracyjna polega na przechowywaniu dwóch ostatnich wyrażeń ciągu i dodawaniu ich do siebie co iterację pętli. Możemy ustalić N pierwszych wyrazów ciągu dodając je na bieżąco do tablicy, lub jedynie N-ty poprzez zwrócenie zmiennej po zakończonej iteracji.

```
def F(n):
        a, b, wyrazy = 0, 1, []
        for i in range(n):
                wyrazy.append(b)
                a, b = b, a+b
        return wyrazy

def F_Nty_wyraz(n):
        # nie trzeba przechowywac wszystkich wyrazow
        a, b = 0, 1
        for i in range(n):
                a, b = b, a+b
        return a

wyrazy = F(10)
dziesiaty_wyraz = F_Nty_wyraz(10)
print(wyrazy)  # [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
print(dziesiaty_wyraz)  # 55
```

Wydawanie reszty metodą zachłanną

Algorytm zachłanny – algorytm który w celu wyznaczenia rozwiązania w każdym kroku dokonuje zachłannego (najkorzystniejszego) w danym kroku rozwiązania częściowego.

Problem wydawania reszty – wybranie z danego zbioru monet o określonych nominałach takiej konfiguracji, by wydać żądaną kwotę przy użyciu minimalnej liczby monet. Jego rozwiązania są wykorzystywane w automatach z napojami, bankomatach itd.

Przykładowo:
wypłacając 148zł chcemy użyć nominałów 100, 20, 20, 5, 2, 1
wypłacając 330zł chcemy użyć nominałów 200, 100, 20, 10

Algorytm zachłanny jest intuicyjny i działa poprawnie dla większości systemów monetarnych (zależy od kwot nominałów).

Dla kwoty N:

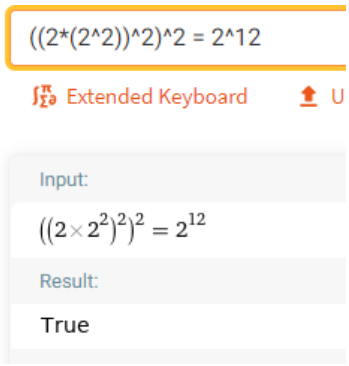
- tworzymy tablicę T przechowującą wynik
- tworzymy tablicę nominały w porządku malejącym, np. [500, 200, 100, 50, 20, 10, 5, 2, 1]
- iterujemy co nominał:
    - dopóki nominał jest mniejszy lub równy od kwoty N:
        - odejmujemy od kwoty N wartość nominału
        - dodajemy nominał do tablicy T

```
def reszta(N):
        nominaly = [500, 200, 100, 50, 20, 10, 5, 2, 1]
        T = []
        for nominal in nominaly:
                while nominal <= N:
                        N -= nominal
                        T.append(nominal)
        return T

print(reszta(148))  # [100, 20, 20, 5, 2, 1]
print(reszta(330))  # [200, 100, 20, 10]
```

Algorytmy numeryczne

Szybkie podnoszenie do potęgi

Wersja rekurencyjna



Normalne podnoszenie do potęgi staje się nieoptymalne przy większych wykładnikach,
na przykład: 2^12 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2

Szybkie podnoszenie do potęgi wykorzystuje własności potęgowania.
Możemy je zastosować przy wykładniku naturalnym.
Aby obliczyć 2^12, musimy jedynie znać wartość 2^6 i podnieść do kwadratu: 2^6 * 2^6, czyli (2^6)^2.
Aby obliczyć 2^6, możemy wykorzystać (2^3)^2.
Aby obliczyć 2^3, możemy wykorzystać 2*(2^2).

Nasze 2^12 zamieniło się w (2*(2^2))^2)^2, które wykonuje jedynie 4 operacje mnożenia.

Wydajność widać gołym okiem: przejście z 2^6 do 2^12 wymaga jednego mnożenia (2^6 * 2^6) zamiast sześciu przy standardowej metodzie (2^6 * 2 * 2 * 2 * 2 * 2 * 2).

Algorytm rekurencyjny dla podstawy A i wykładnika naturalnego N:

- jeśli wykładnik N = 0:
  - zwróć 1 (A^0 = 1)
- jeśli wykładnik N jest nieparzysty:
  - zwróć A pomnożone przez A^(N-1)
- jeśli wykładnik N jest parzysty:
  - zwróć A^(N/2) do kwadratu

```
def potega(A, N):
        if N == 0:
                return 1
        if N % 2:  # nieparzysty
                return A * potega(A, N-1)
        else: # parzysty
                return potega(A, N/2) ** 2
```

Wersja iteracyjna

Istnieje również wersja iteracyjna, która wykorzystuje przesunięcia bitowe. Jej działanie przedstawię na wyrażeniu 2^10:

potega(2, 10):
wynik = 1, A = 2, N = 1010 (binarnie)

wynik = 1, A = 4, N = 101
wynik = 1 * 4, A = 16, N = 10
wynik = 1 * 4, A = 256, N = 1
wynik = 1 * 4 * 256, A = 65536, N = 0, działanie zakończone

```
def potega(A, N):
        wynik = 1
        while N > 0:
                # sprawadzanie nieparzystosci na podstawie AND z 1
                if N & 1:
                        wynik = wynik * A
                A = A ** 2
                # przesuniecie bitowe w prawo
                N = N >> 1
        return wynik
```

Działa to na identycznej zasadzie jak iteracyjna wersja, lecz jest bardziej wydajne i nie trzeba się martwić o przepełnienie stosu.

Przesunięcie bitowe w prawo liczby np. 46 (101110) dzieli ją przez dwa – 101110 >> 1 = 101111, czyli 23. Ostatni bit jest tracony, jest to tym samym dzielenie całkowite.

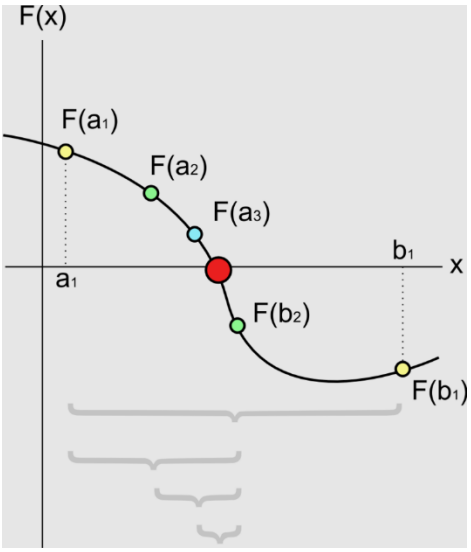
Natomiast N & 1 zwraca prawdę, gdy ostatni bit wynosi 1. Można zauważyć, że dzieje się to przy nieparzystych liczbach.

Na tej podstawie załączę jeszcze trzecią wersję – iteracyjną, ale bez działań na bitach.

```
def potega(A, N):
        wynik = 1
        while N > 0:
                # sprawadzanie nieparzystosci
                if N % 2:
                        wynik = wynik * A
                A = A ** 2
                # dzielenie calkowite
                N = N // 2
        return wynik
```

Wyznaczanie miejsc zerowych funkcji metodą połowienia

[2]



Znane również jako metoda bisekcji i metoda równego podziału.

Aby wyznaczyć miejsce zerowe:

1. należy wybrać przedział, w którym funkcja przyjmuje różne znaki na jego końcach – na ilustracji jest to przedział [a, b]. F(a) jest dodatnie, a F(b) jest ujemne.
2. funkcja musi być ciągła w przedziale [a, b].

Upraszczając – funkcja jest ciągła, gdy jej wykresem jest linia ciągła (można ją narysować bez odrywania ołówka od papieru).

Algorytm polega na sprawdzaniu kolejnych punktów, aż do odnalezienia miejsca zerowego – w przybliżeniu. Trzeba przyjąć pewien margines błędu, ponieważ nie jesteśmy zawsze w stanie ustalić dokładnej wartości miejsca zerowego.

Dla funkcji F(X) w przedziale <a, b> z dokładnością wyniku ϵ:

- wyznaczamy środek przedziału S = a+b/2
- jeśli F(S) wynosi 0 (+- ϵ), zwracamy S – jest to nasze miejsce zerowe

W tym kroku nie porównujemy bezpośrednio do 0. Obliczamy wartość bezwzględną z F(S) i sprawdzamy, czy jest mniejsza od ϵ.

- jeśli znaki F(a) i F(S) są różne:
  - miejsce zerowe znajduje się w przedziale <a, s), więc zmniejszamy przedział
  - ustawiamy B = S
- w przeciwnym wypadku, gdy znaki F(a) i F(S) są równe:
  - miejsce zerowe znajduje się w przedziale (s, b>, więc zmniejszamy przedział
  - ustawiamy A = S

Znaki są różne, gdy F(a) * F(S) < 0.

- algorytm powtarzamy dopóki abs(a-b) > ϵ, czyli dopóki nie uzyskamy ustalonej dokładności

[2] Grafika: „Bisection Method”, autor: Tokuchan, licencja CC BY-SA 3.0, data dostępu: 2021-02-28

```python
def F(x):
	return x**3 - x + 1

def bisekcja(a, b, dokladnosc):
	# dopoki nie osiagnelismy dokladnosci
	while abs(a - b) > dokladnosc:
		# ustalamy nowy srodek
		srodek = (a + b) / 2

		# sprawdzamy, czy trafilismy na miejsce zerowe
		if abs(F(srodek)) <= dokladnosc:
			return srodek

		# sprawdzamy, czy F(a) i F(s) maja rozne znaki,
		# jesli tak to miejsce zerowe jest w <a, s)
		if F(srodek) * F(a) < 0:
			b = srodek
		else:
			a = srodek

	# nie zwracam zmiennej srodek, bo nie jest z nowego przedzialu
	return (a + b) / 2

x = bisekcja(-2, 2, 0.0001)
print(f"F({x}) = {F(x):.7f}")  # F(-1.32470703125) = 0.0000466
```

Algorytmy na tekstach

Sprawdzanie, czy ciąg znaków tworzy palindrom

Palindrom to np. „KAMILŚLIMAK” – czytany od tyłu daje „KAMILŚLIMAK”.

Algorytm jest prosty, porównujemy znaki iterując jednocześnie od początku jak i od końca. Musimy sprawdzić jedynie połowę długości łańcucha z każdej strony:

KAMILŚLIMAK – K == K
KAMILŚLIMAK – A == A
KAMILŚLIMAK – M == M
KAMILŚLIMAK – I == I
KAMILŚLIMAK – L == L
KAMILŚLIMAK – Ś == Ś – koniec iteracji

Dla palindromu o parzystej liczbie znaków:

HANNAH – H == H
HANNAH – A == A
HANNAH – N == N – koniec iteracji

```
def palindrom(wyraz):
        licznik1 = 0  # od poczatku
        licznik2 = len(wyraz) - 1  # od konca

        while licznik1 < licznik2:  # dopoki sie nie zbiegna
                if wyraz[licznik1] != wyraz[licznik2]:
                        return False

                licznik1 += 1
                licznik2 -= 1

        return True

print(palindrom('KAMILŚLIMAK'))  # True
```

Sprawdzanie, czy ciąg znaków tworzy anagram

Anagram to wyraz powstały po przestawieniu liter innego wyrazu. Np. „ruda”, „udar” lub „mata”, „tama”.

Zbiór liter jest taki sam w anagramach, lecz w innej kolejności. Musimy tym samym zliczyć występowanie liter w obu wyrazach i porównać.

```
def anagram(wyraz1, wyraz2):
	litery1, litery2 = {}, {}

	for litera in wyraz1:
		# jesli nie ma litery w zbiorze, inicjujemy licznik
		if litera not in litery1:
			litery1[litera] = 0
		litery1[litera] += 1

	for litera in wyraz2:
		# jesli nie ma litery w zbiorze, inicjujemy licznik
		if litera not in litery2:
			litery2[litera] = 0
		litery2[litera] += 1

	return litery1 == litery2

print(anagram('ruda', 'udar'))  # True
print(anagram('ruddaa', 'udarad'))  # True
print(anagram('to nie jest', 'anagram'))  # False
```

Porządkowanie alfabetyczne

Ten punkt to zastosowanie algorytmów sortowania. Nic nowego (jeśli dobrze interpretuję).

Wyszukiwanie wzorca w tekście

W tym algorytmie sprawdzamy, czy łańcuch znaków znajduje się w innym łańcuchu znaków. Przykładowo: „asn” znajduje się w „krasnal”, „bar” znajduje się w „barek”, „dź” znajduje się w „łabędź”.

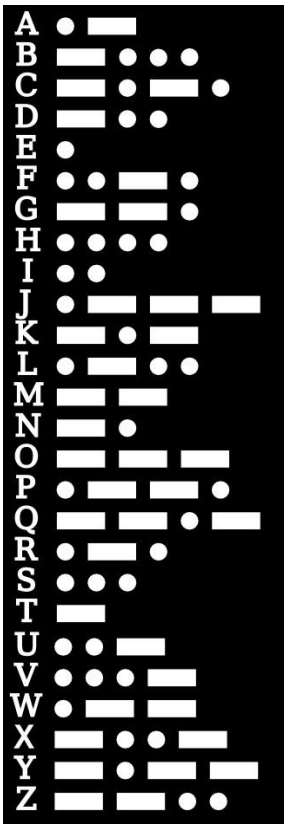
W zależności od zastosowania możemy znaleźć wszystkie wystąpienia wzorca (np. „to” w „stomatolog” występuje dwa razy) lub zatrzymać się po pierwszym wystąpieniu.

Przedstawiam algorytm naiwny, jego wydajność zależy od wielkości łańcuchów znaków:
szukając „asn” w „krasnal” dokonujemy max. 5 porównań:
[kra], [ras], [asn], [sna], [nal]
szukając „to” w „stomatolog” dokonujemy max. 9 porównań:
[st], [to], [om], [ma], [at], [to], [ol], [lo], [og]
szukając „robot” w „robotem” dokonujemy max. 3 porównań:
[robot], [obote], [botem]

```
def szukaj(wzorzec, tekst):
	"""Zwraca pozycję początku wzorca, -1 jeśli nieznaleziono"""
	dlugosc_tekstu, dlugosc_wzorca = len(tekst), len(wzorzec)

	if dlugosc_wzorca > dlugosc_tekstu:
		return -1  # za duzy wzorzec

	# pozycje okienka do porównywania
	poczatek = 0
	koniec = dlugosc_wzorca

	while koniec <= dlugosc_tekstu:
		fragment = tekst[poczatek:koniec]
		if fragment == wzorzec:
			return poczatek

		poczatek += 1
		koniec += 1

	return -1

print(szukaj('asn', 'krasnal')) # 2
print(szukaj('to', 'stomatolog')) # 1
print(szukaj('bar', 'barek')) # 0
print(szukaj('bar', 'bar')) # 0
print(szukaj('dź', 'łabędź')) # 4
print(szukaj('pies', 'kotek')) # -1
```

Algorytmy kompresji i szyfrowania

[3]Kody znaków o zmiennej długości – alfabet Morse'a



Szyfrowanie i odszyfrowanie alfabetu Morse'a polega na wykorzystaniu słownika z przypisanymi znakami do poszczególnych liter. Pomiędzy literami wstawiamy jedną spację, natomiast pomiędzy wyrazami dwie spacje.

Algorytm jest bardzo prosty - polega na iteracji co znak i dopasowywania symbolu. Jest natomiast długi ze względu na konieczność stworzenia całego dicta (jak na załączonym obrazku), z tego powodu nie zamieszczam kodu.

[3] Grafika: „MorseDecoded", data dostępu: 2021-02-28

Szyfr Cezara

Szyfr Cezara polega na przestawieniu każdej litery o określoną ilość znaków w alfabecie.

Każda litera z angielskiego alfabetu ma przypisany kod ASCII. ord(„A”) == chr(65), ord(„”B”) == chr(66), ..., ord(„Z”) = chr(90). Przykładowo literki słowa MACIEJ mają kody [77, 65, 67, 73, 69, 74].

Przy przesunięciu MACIEJ o 5 znaków nie napotkamy problemu – największa wartość 77+5 mieści się w zakresie alfabetu <65, 90>.
Przy przesunięciu MACIEJ o 20 znaków największa wartość 77+20 = 97 nie mieści się już w zakresie <65, 90>.

Musimy zatem sprowadzić znaki z <65, 90> do <0, 25>, zastosować modulo 26 i sprowadzić znaki z <0, 25> do <65, 90>. Dzięki temu np.

- znak 90 przyjmie wartość 25,
- przesunięty o 2 znaki przyjmie wartość 27,
- po modulo będzie miał wartość 2,
- po przywróceniu do <65, 90> będzie miał wartość 66 (znak B)

Sformułujmy algorytm:

Dla ciągu szyfrowanego C z kluczem przesunięcia K:

- tworzymy zmienną wynik
- dla każdego znaku Z w ciągu C:
    - W = wartość kodu ASCII dla znaku Z
    - zmniejszamy W o 65 (pozycja pierwszej litery w ASCII)
    - dodajemy do W klucz przesunięcia K
    - przypisujemy do W wartość W % 26 (ilość liter w alfabecie łacińskim)
    - zwiększamy W o 65
    - dopisujemy do zmiennej wynik literę odpowiadającą kodowi W

Gdybyśmy chcieli wykorzystać polski alfabet to nie możemy stosować wbudowanych funkcji chr() i ord(). Polskie znaki nie znajdują się w kodzie ASCII. W takim przypadku musielibyśmy oprzeć się o zmienną zawierającą polski alfabet.

Zadanie maturalne wykorzystujące szyfr Cezara: MIN-R2_1P-162, zadanie 6

```
def cezar(oryginal, klucz):
        wynik = ""
        for znak in oryginal:
                kod_znaku = ord(znak)
                kod_znaku -= 65
                kod_znaku += klucz
                kod_znaku %= 26
                kod_znaku += 65
                wynik += chr(kod_znaku)
        return wynik

print(cezar("ZABAWA", 13))
```

Szyfr przestawieniowy

Szyfry przestawieniowe polegają na zamianie kolejności znaków w tekście. Istnieje bardzo dużo szyfrów przedstawieniowych, każdy może wymyślić swój. Nauka implementacji wybranego szyfru może pomóc przy samodzielnej implementacji na maturze, lecz z pewnością nie będzie wymagana wiedza dot. działania konkretnych szyfrów. Zostawiam tym samym implementację jako zadanie dla czytelnika.

Przykładowe szyfry w oparciu o artykuł Transposition cipher [Wikipedia]:

Rail Fence cipher

'WE ARE DISCOVERED FLEE AT ONCE'

```
W . . . E . . . C . . . R . . . L . . . T . . . E
. E . R . D . S . O . E . E . F . E . A . O . C .
. . A . . . I . . . V . . . D . . . E . . . N . .
```

Then reads off:

```
WECRL TEERD SOEEF EAOCA IVDEN
```

Scytale

```
W . . E . . A . . R . . E . . D . . I . . S . . C
. O . . V . . E . . R . . E . . D . . F . . L . .
. . E . . E . . A . . T . . O . . N . . C . . E .
```

In this example, the cylinder is running horizontally and the ribbon is wrapped around vertically. Hence, the cipherer then reads off:

```
WOEEV EAEAR RTEEO DDNIF CSLEC
```

Route cipher

In a route cipher, the plaintext is first written out in a grid of given dimensions, then read off in a pattern given in the key. For example, using the same plaintext that we used for rail fence:

```
W R I O R F E O E
E E S V E L A N J
A D C E D E T C X
```

The key might specify "spiral inwards, clockwise, starting from the top right". That would give a cipher text of:

```
EJXCTEDEC DAEWRIORF EONALEVSE
```

Dodatkowe algorytmy

Poprzednie algorytmy są wymienione w oparciu o tegoroczne zagadnienia od CKE. Zabrakło mi w nich kilku prostych, lecz istotnych algorytmów:

Wyszukiwanie binarne

Wyszukiwanie binarne służy do szybkiego znalezienia elementu w posortowanej tablicy.

Przyjmijmy, że chcemy znaleźć 10 w tablicy [1, 7, 8, 9, 10, 13, 15, 20, 25, 30, 33, 100, 101].

Środkowy element to 15 – jest większy od 10. Szukamy więc w [1, 7, 8, 9, 10, 13].
Środkowy element to 8 – jest mniejszy od 10. Szukamy więc w [9, 10, 13].
Środkowy element to 10 – znaleźliśmy!

Niech X będzie szukanym elementem, a T uporządkowaną tablicą:

- ustawiamy lewo = 0, prawo = długość T
- dopóki lewo <= prawo:
  - ustawiamy środek jako lewo+prawo // 2
  - jeśli T[środek] == X:
    - zwracamy środek – to nasz element
  - jeśli T[środek] > X:
    - ustawiamy prawo jako środek-1 (zawężamy do liczb mniejszych)
  - jeśli T[środek] < X:
    - ustawiamy lewo jako środek+1 (zawężamy do liczb większych)
- jeśli pętla się skończyła to nie znaleziono X – zwracamy np. -1

```
def wyszukiwanie_binarne(szukany, tablica):
	lewo = 0
	prawo = len(tablica) - 1

	while lewo <= prawo:
		# dzielenie calkowite
		srodek = (lewo+prawo) // 2

		if tablica[srodek] == szukany:
			return srodek
		elif tablica[srodek] > szukany:
			prawo = srodek - 1
		else:  # tablica[srodek] < szukany
			lewo = srodek + 1

	return -1

T = [1, 7, 8, 9, 10, 13, 15, 20, 25, 30, 33, 100, 101]
for element in T:
	print(wyszukiwanie_binarne(element, T))
```

Lista algorytmów nieobowiązujących na maturze 2021:

Poniżej przedstawiam wykreślone punkty z zagadnień od CKE:

2.1. jednoczesne znajdowanie największego i najmniejszego elementu w zbiorze: algorytm naiwny i optymalny
2.2 [...] sortowanie przez wstawianie binarne [...]

3.1. obliczanie wartości pierwiastka kwadratowego
3.2. obliczanie wartości wielomianu za pomocą schematu Hornera
3.3. zastosowania schematu Hornera: reprezentacja list w różnych systemach liczbowych [...]
3.5. obliczanie pola obszarów zamkniętych

4.4. obliczanie wartości wyrażenia podanego w postaci odwrotnej notacji polskiej

5.1. [...] kod Huffmana
5.4. szyfr z kluczem jawnym (RSA)
5.5. wykorzystanie algorytmów szyfrowania, np. w podpisie elektronicznym

6. algorytmy badające własności geometryczne, np.:
6.1. sprawdzanie warunku trójkąta
6.2. badanie położenia punktów względem prostej
6.3. badanie przynależności punktu do odcinka
6.4. przecinanie się odcinków
6.5. przynależność punktu do obszaru
6.6. konstrukcje rekurencyjne: drzewo binarne, dywan Sierpińskiego, płatek Kocha