POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

WPROWADZENIE DO INFORMATYKI

Strukturalne typy danych

Dr hab. Małgorzata Charytanowicz

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

Formalna definicja typu wartości

Formalna definicja typu wartości

Z każdą wartością będącą przedmiotem przetwarzania w algorytmie wiąże się **przynależność do pewnego zbioru wartości**, określonego nie tylko z punktu ich podobieństwa „algebraicznego”, ale także pod względem:

- *sposobu zapisywania* takich wartości na nośniku informacji,
- *ilości zajmowanego miejsca* przez taki zapis (wyrażonego w bitach lub bajtach).

Formalna definicja typu wartości

Na treść pojęcia **typu wartości** składają się:

- *zbiór wartości tego typu*, które mogą być przechowywane przez dane tego typu,
- *zbiór operacji* (działań, przy pomocy których konstruujemy wyrażenia, w których wartości tego typu mogą być operandami),
- *zbiór relacji*, których zachodzenie między wartościami tego typu i ewentualnie typów z nim zgodnych można badać.

Typ wartości

Typ wartości *T* jest uporządkowaną trójką zbiorów:

$$T = (\Omega, \phi, \psi) ,$$

gdzie

- Ω jest zbiorem wartości typu *T*,
- ϕ jest zbiorem operacji wykonalnych na tych wartościach i wartościach innych, zgodnych typów,
- ψ jest zbiorem relacji, które mogą zachodzić pomiędzy wartościami tego typu i typów z nim zgodnych.

Zbiór wartości (uniwersum) typu *T* jest zbiorem skończonym.

- Najmniejsza sensowna liczebność zbioru wartości dowolnego typu wynosi 2.
- Wartości należące do tego zbioru mogą być wartościami danych stałych (także przedstawianych jako literały) jak i zmiennych.

Typ wartości

- Gdy wartości danego typu *T* są wartościami prostymi, mówimy że **typ *T* jest prosty**, gdy są strukturalnymi, mówimy że typ *T* jest typem strukturalnym.
- Wartości danego typu mogą być również **wartościami referencyjnymi**, tj. wskazującymi miejsca na nośniku pamięci, w których znajdują się dane przechowujące inne wartości.
- Jeżeli zbiór wartości jednego typu zawiera się w zbiorze wartości innego typu, mówimy że pierwszy z tych typów jest **podtypem lub okrojeniem** drugiego.
- Typy o nierozłącznych zbiorach wartości **mogą być typami niezgodnymi** w sensie przypisania.
- Typy o rozłącznych zbiorach wartości z reguły **mogą być niezgodne**, zarówno w zwykłym sensie jak i w sensie przypisania.

Zbiór operacji

Zbiór operacji typu *T* zawiera działania, przy pomocy których ze zmiennych i stałych tego typu i typów z nim zgodnych można budować wyrażenia.

- Jeżeli wynik działania nie jest wartością typu *T,* mamy do czynienia z **konwersją typów**.
- **Przypisanie nie jest operacją ze zbioru operacji typu *T***, nie wytwarza bowiem żadnej nowej wartości, tylko już wytworzoną przypisuje do innej zmiennej.
- Istnieją typy, których zbiór operacji jest pusty, w typie tym nie jest możliwe zbudowanie wyrażenia ani dokonanie przypisania.
- **Zbiór operacji** składa się z **predefiniowanych**, ujętych w samej definicji samego języka programowania **operatorów**; zbiór ten programista może dowolnie **rozszerzać** o algorytmy dodatkowych operacji, opisane w postaci funkcji lub procedur.

Zbiór relacji

Zbiór relacji typu *T* może być zbiorem pustym. Jeżeli jest niepusty, należy do niego zwykle relacja równości i jej zaprzeczenie („*nie jest równe*”).

- Większość relacji w językach programowania to relacje dwuargumentowe, rozumiane jako relacje o wartościach logicznych – wartość *Prawda* oznacza, że relacja zachodzi.
- Istotną rolę odgrywa relacja porządku zupełnego „≤” i pochodne od niej: <, ≤, >, ≥, =.

Wartości logiczne (*prawda*, *fałsz*) oprócz zapisu literalnego mogą być otrzymywane jako wartości wyrażeń logicznych i używane w instrukcjach jako wartości przypisywane do zmiennych oraz jako kryteria wyboru w selekcjach i iteracjach.

- Najprostsze wyrażenia logiczne to zapisy relacji porównania pomiędzy dwoma wyrażeniami arytmetycznymi, a także należenia lub zawierania się zbiorów itp.

Wyrażenia logiczne

- Złożone wyrażenia logiczne można budować używając **funktorów logicznych**: negacji *not*, alternatywy *or* oraz koniunkcji *and*.
- Dopuszcza się czasem s**krócony sposób obliczania złożonych wyrażeń logicznych**: obliczanie alternatywy przerywa się, gdy badany składnik okazał się prawdziwy, a koniunkcji, gdy badany składnik okazał się fałszywy.
- **Wyrażenie logiczne** samo już jest zdaniem oznajmującym i może być użyte w charakterze warunku.
- Niektóre języki nie wyróżniają typu logicznego, używając w to miejsce **wartości całkowitych**: 0 jako *fałsz*, każda inna wartość jako *prawda*.

Wyrażenia logiczne

Przykłady

```
a = 2
if a != 0:
    print('sukces')


a = 2
if a:
    print('sukces')
```

Struktury danych i ich reprezentacje

Termin struktura oznacza w matematyce dowolny zbiór, w którym została zdefiniowana przynajmniej jedna relacja. Jest to twór abstrakcyjny.

Struktura danych jest strukturą:

- której elementami są fizycznie istniejące dane,
- w której została zdefiniowana relacja równości: dwie dane są równe wtedy i tylko wtedy, gdy mają identyczne nazwy.

Oprócz relacji równości danych, w strukturze danych mogą zostać zdefiniowane inne relacje między jej elementami:

- dotyczące wartości danych, np. relacja mniejszości,
- relacje strukturalne, pomiędzy nazwami, a więc i tożsamościami poszczególnych danych (porządek w liście, hierarchia w drzewach).

Struktury danych

Struktury danych stanowią nośniki reprezentacji obiektów i systemów rzeczywistych.

I Zasada Jacksona

Struktura danych opisująca system rzeczywisty powinna odzwierciedlać strukturę tego systemu.

II Zasada Jacksona

Struktura algorytmu jest zdeterminowana przez struktury danych, które ten algorytm przetwarza.

Reprezentowanie obiektów rzeczywistych

- **Wszystkie informacje** w aspekcie danego problemu, dotyczące cech pojedynczego przedmiotu **powinny być zestawione jako struktura danych** reprezentowana przez pojedynczą daną strukturalną lub referencyjną.
- Struktury lub dane strukturalne przedstawiające dane o przedmiotach tego samego rodzaju (posiadających te same atrybuty) powinny mieć **tą samą konfigurację** (budowę wewnętrzną struktury: kolejność atrybutów, zgodność typów, itp.).
- Dane, z których każda zestawia informacje o jednym z wielu obiektów tego samego rodzaju , w przypadku ich większej liczby **powinny być gromadzone w strukturach danych wyższej rangi**, na ogół jednorodnych.

Algorytmizacja przetwarzania

W zależności od tego czy wszystkie wartości struktury są tego samego typu wyróżnia się:

- struktury jednorodne – tablice, pliki jednorodne, listy jednorodne, drzewa,
- struktury niejednorodne – rekordy.

Z uwagi na identyczną charakterystykę wartości wszystkich składowych struktur jednorodnych algorytmy ich przetwarzania będą wykorzystywały **powtarzalność** jednokrotnie opisanego przebiegu operacji. Wyróżniamy zatem:

- algorytmy iteracyjne – konstruowane w oparciu o zasadę redukcji rozmiaru przetwarzanej struktury,
- algorytmy rekursywne – w których rozwiązanie problemu przetworzenia składowych struktury konstruuje się na podstawie wyników przetwarzania jego drobniejszych podzbiorów.

Typ tablicowy, tablice

Tablice i typy tablicowe

- Struktura tablicowa jest **najstarszą** w dziejach informatyki i całkowicie naturalną dla komputera formą jednorodnej struktury danych.
- Jest to **skończony ciąg danych tego samego typu**, ponumerowanych kolejnymi liczbami całkowitymi z pewnego zakresu, zazwyczaj zaczynających się od 0 lub od 1. Liczba elementów tablicy jest znana.
- Domyślnie składowe tego ciągu rozmieszczone są **konsekutywnie, jedna po drugiej,** w kolejności tej numeracji, w spójnym obszarze pamięci operacyjnej.
- Numeracja ta ustanawia **porządek strukturalny**.
- Dostęp do danych jest **swobodny**.
- Daną reprezentującą tę strukturę bezpośrednio nazywamy **tablicą**, a dane składowe tej struktury – składowymi tablicy.
- Tablica może być daną całościową lub składową pewnej danej strukturalnej.
- Tablica może mieć zadeklarowany **identyfikator** bądź może być wskazywana przez **wskaźnik**.

Tablice

- Numer porządkowy składowej tablicy nazywa się jej **indeksem**.
- Zakres indeksowania tablic może mieć status typu wartości i może mieć zadeklarowany identyfikator (np. w języku Pascal, typ okrojony z typu całkowitego).
- **Indeksowanie tablicy** zależnie od założeń danego języka programowania, może być rozpoczynane:
  - od 1,
  - od 0,
  - do wyboru programisty, o ile pozwala na to język (np. Pascal).
- Każda składowa tablicy jest pełnoprawną daną typu składowego tablicy. Jej oznaczeniem jest oznaczenie tablicy uzupełnione indeksem **ujętym w nawiasy kwadratowe** (lub okrągłe), zgodnie ze składnią danego języka. Indeks może być przedstawiony w postaci wyrażenia, zwanego wyrażeniem indeksowym, o wartościach całkowitych.

Tablice

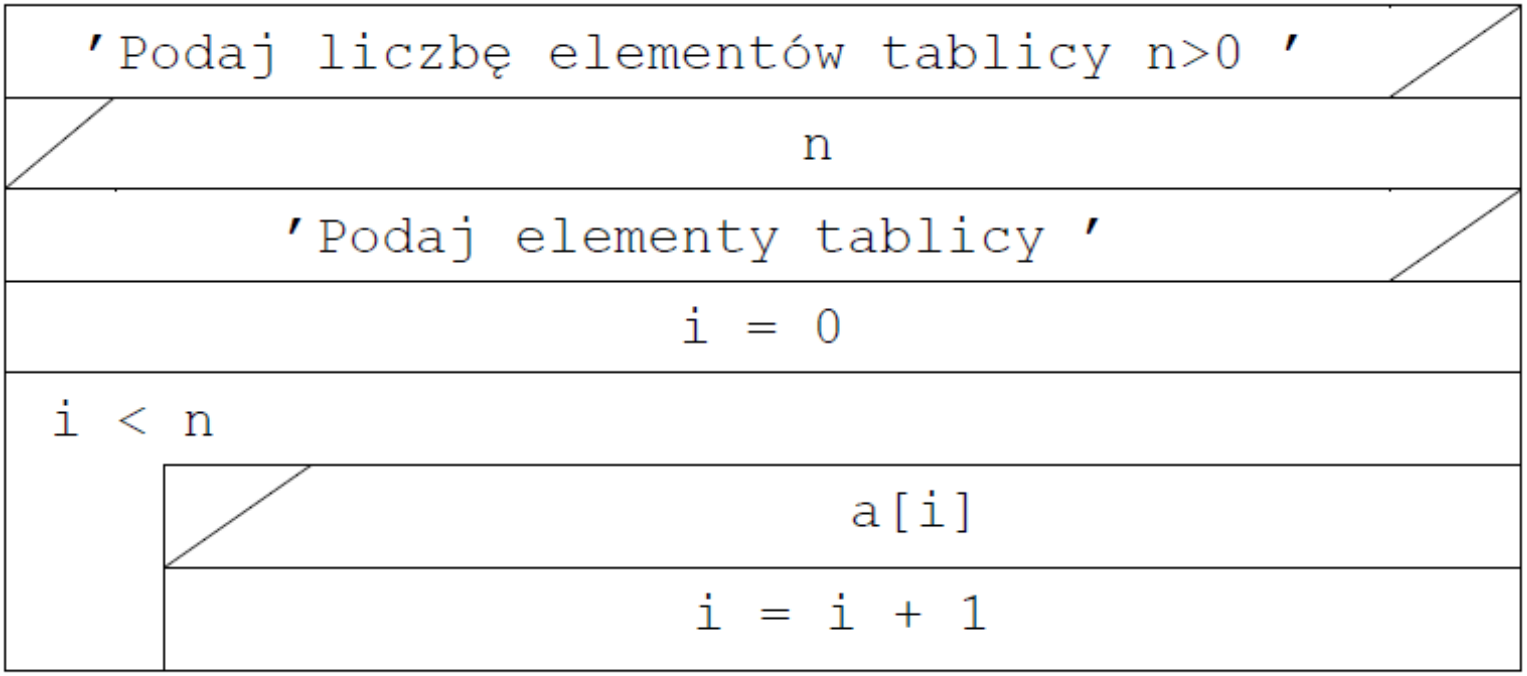
Typem składowym tablicy może być z reguły dowolny typ wartości zdefiniowany w danym języku:

- może to być dowolny typ prosty, wskaźnikowy lub strukturalny,
- jeżeli typ składowy tablicy **nie jest sam typem tablicowym**, tablicę nazywamy **jednowymiarową**,
- jeżeli typ składowy jest innym typem tablicowym, to taka tablica tablic nazywa się **tablicą wielowymiarową**,
- jeżeli składowe tablicy są tablicami jednowymiarowymi, to tablica jest dwuwymiarowa, a jej składowe tablice nazywane są **wierszami**; składowe o tym samym indeksie w poszczególnych wierszach tworzą **kolumnę** tablicy, a ten wspólny indeks to numer kolumny,
- w większości języków programowania można budować tablice wielowymiarowe.

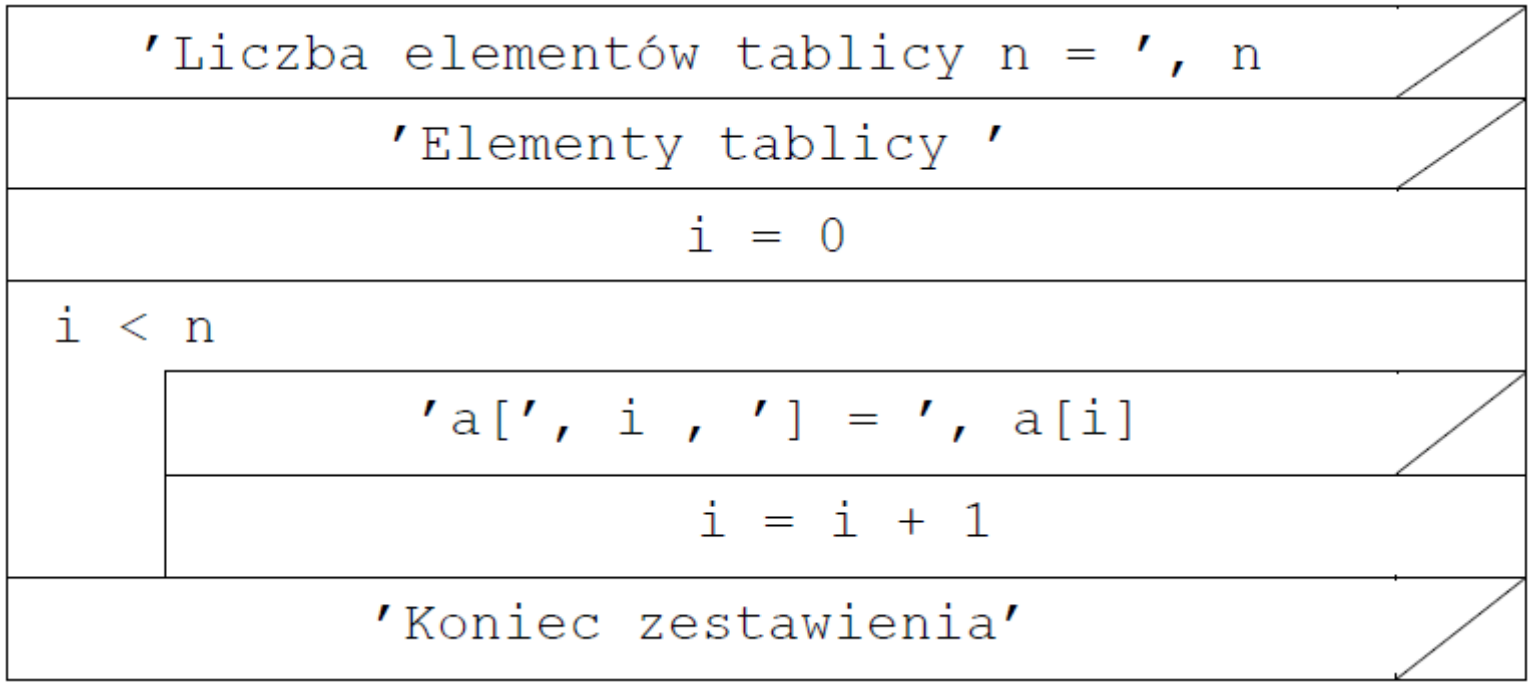
Tablice

- Tablica jednowymiarowa o *n* elementach jest naturalnym pojemnikiem do przechowywania **skończonych ciągów** o *n* wyrazach, może też być traktowana jako reprezentacja **wektora** *n*-wymiarowego.
- Tablica dwuwymiarowa może być naturalnie interpretowana jako **tabela** wartości funkcji dwóch zmiennych i być wykorzystywana do przechowywania **macierzy**.
- Podstawowym schematem algorytmu przetwarzania struktury tablicowej powinna być **pętla** „dopóki”.
- W językach udostępniających pętlę automatyczną „for” wykorzystuje się ją w naturalny sposób do przetwarzania tablic.

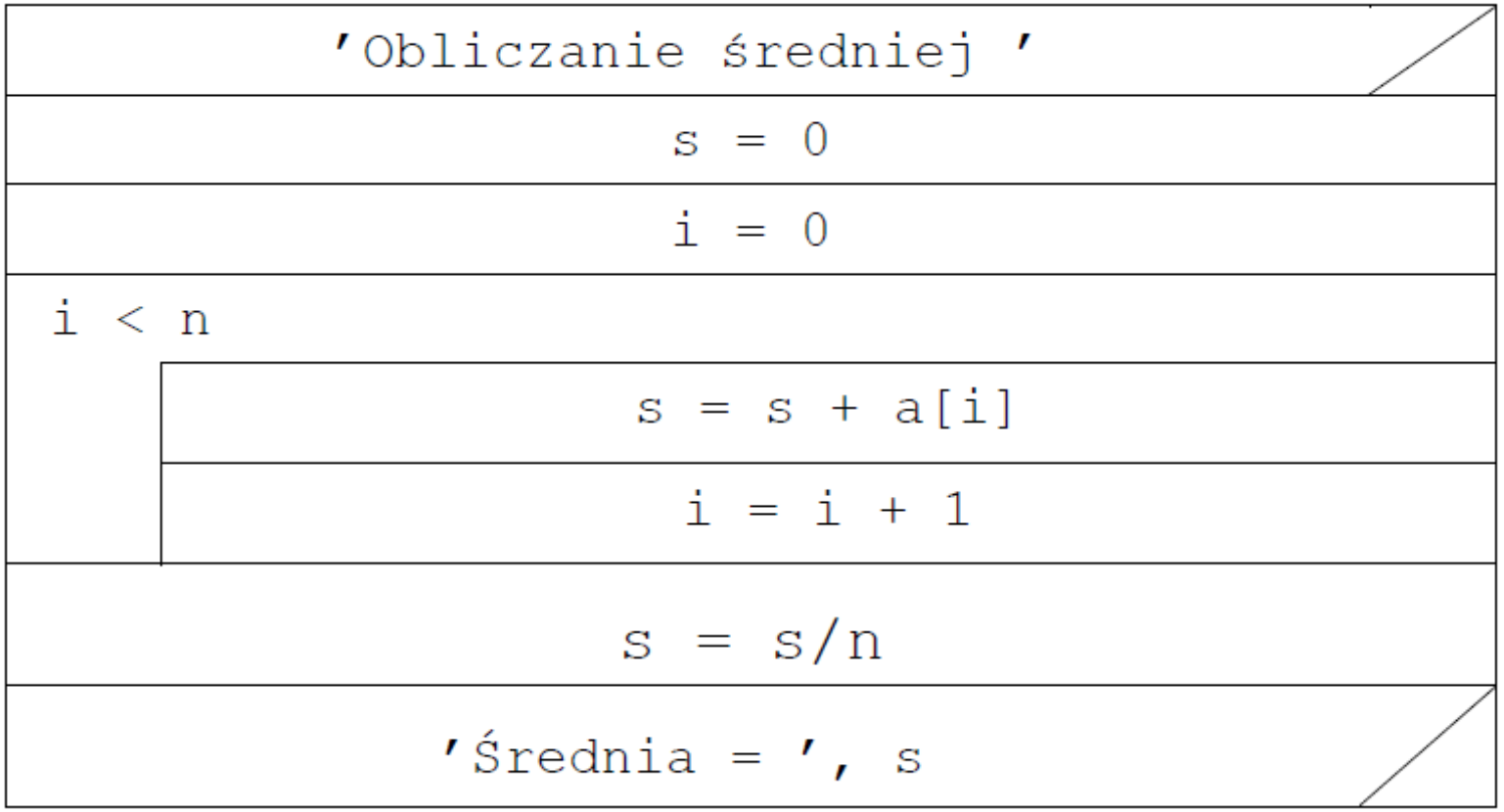
Tablice – wczytanie danych



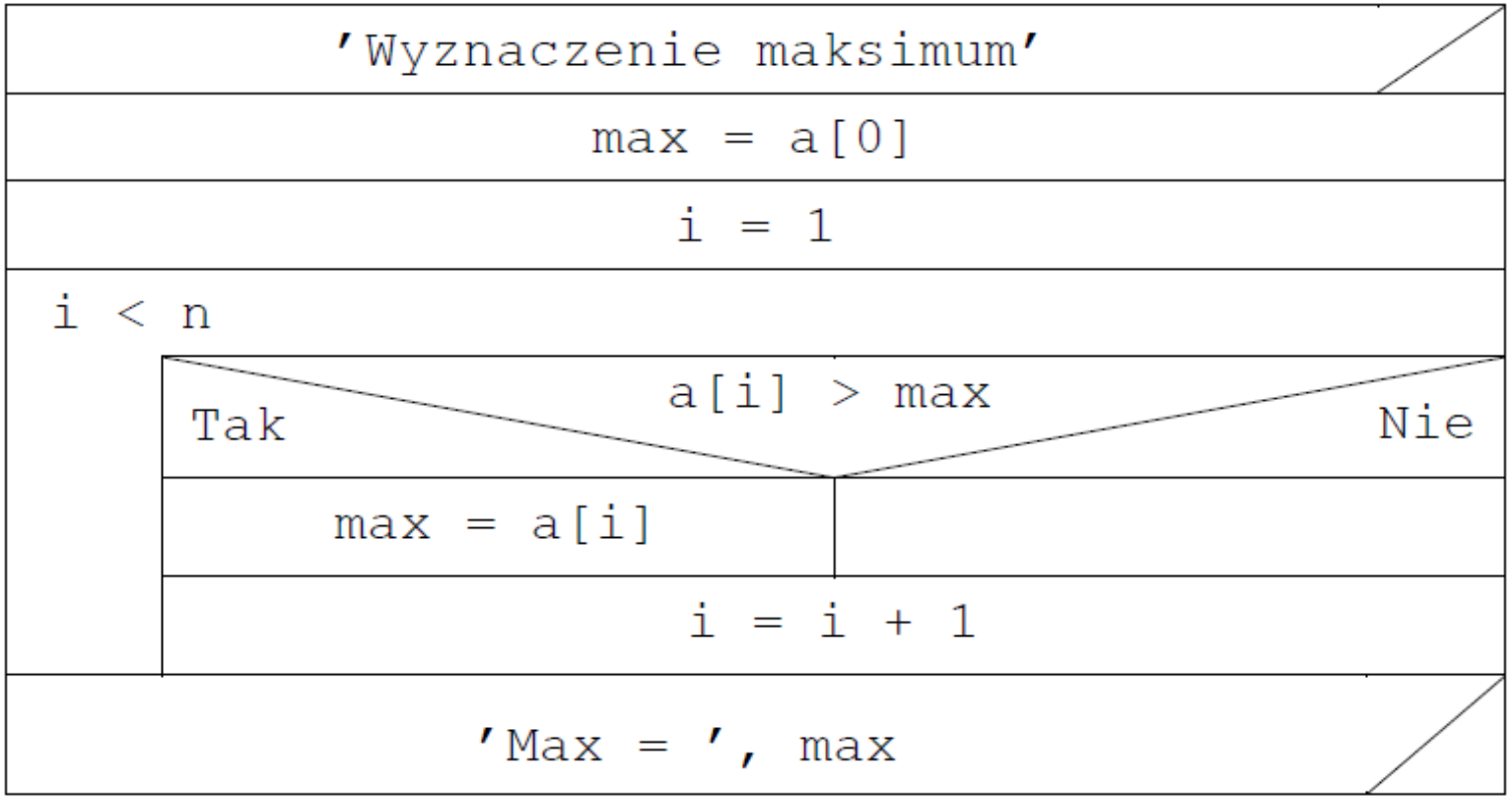
Tablice – wyświetlenie danych



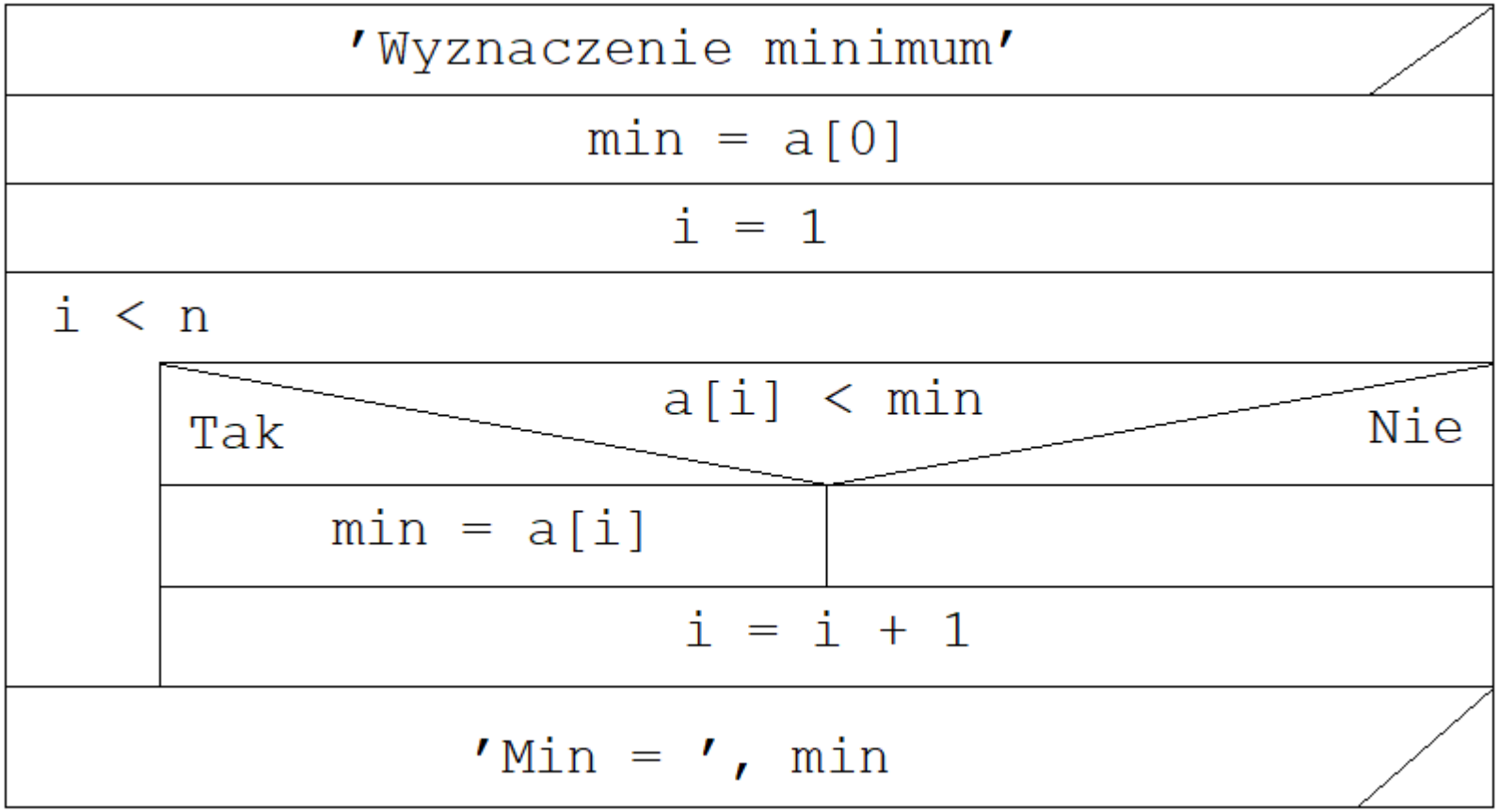
Tablice – obliczanie średniej



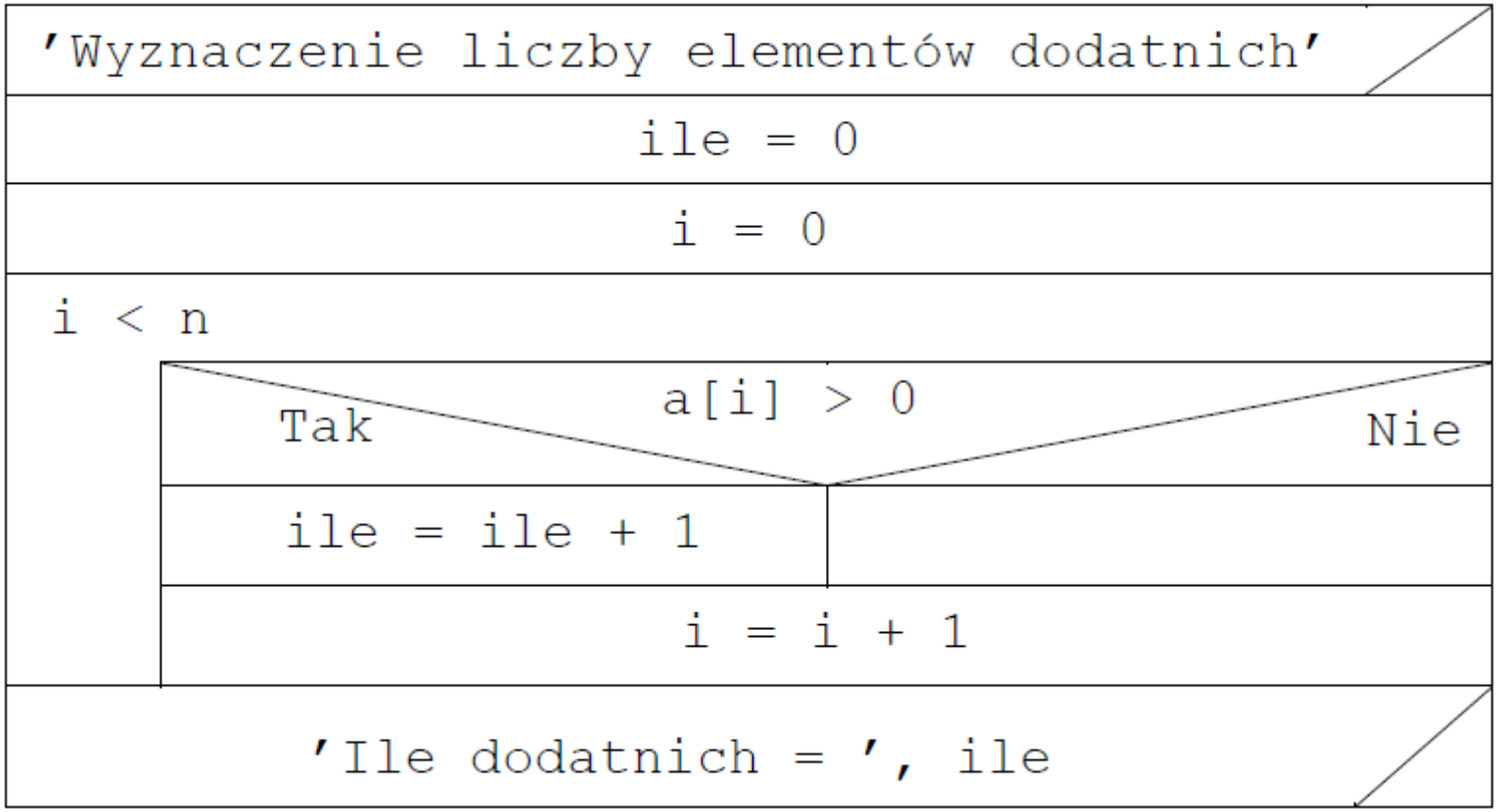
Tablice – wyznaczanie maksimum



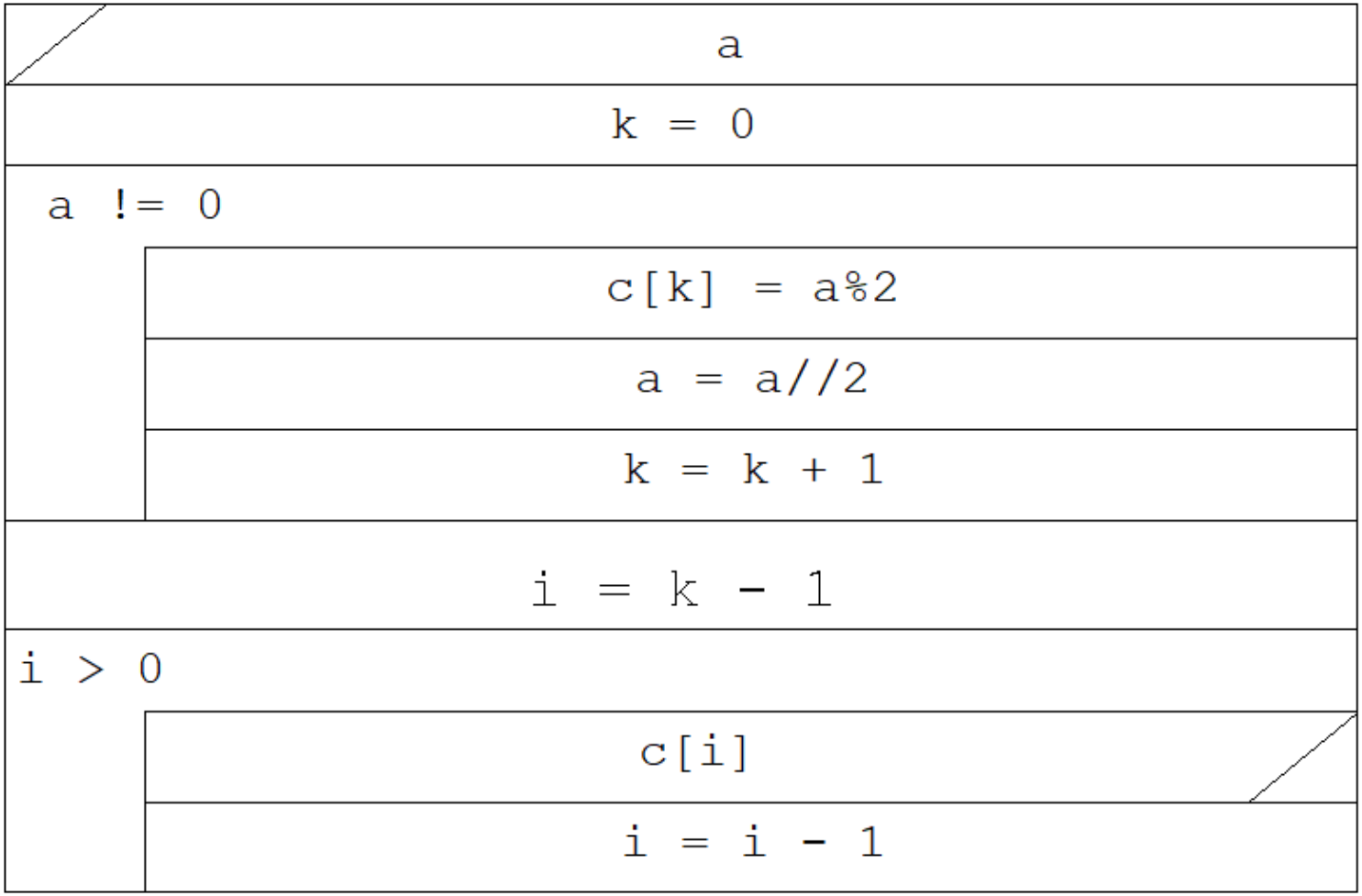
Tablice – wyznaczanie minimum



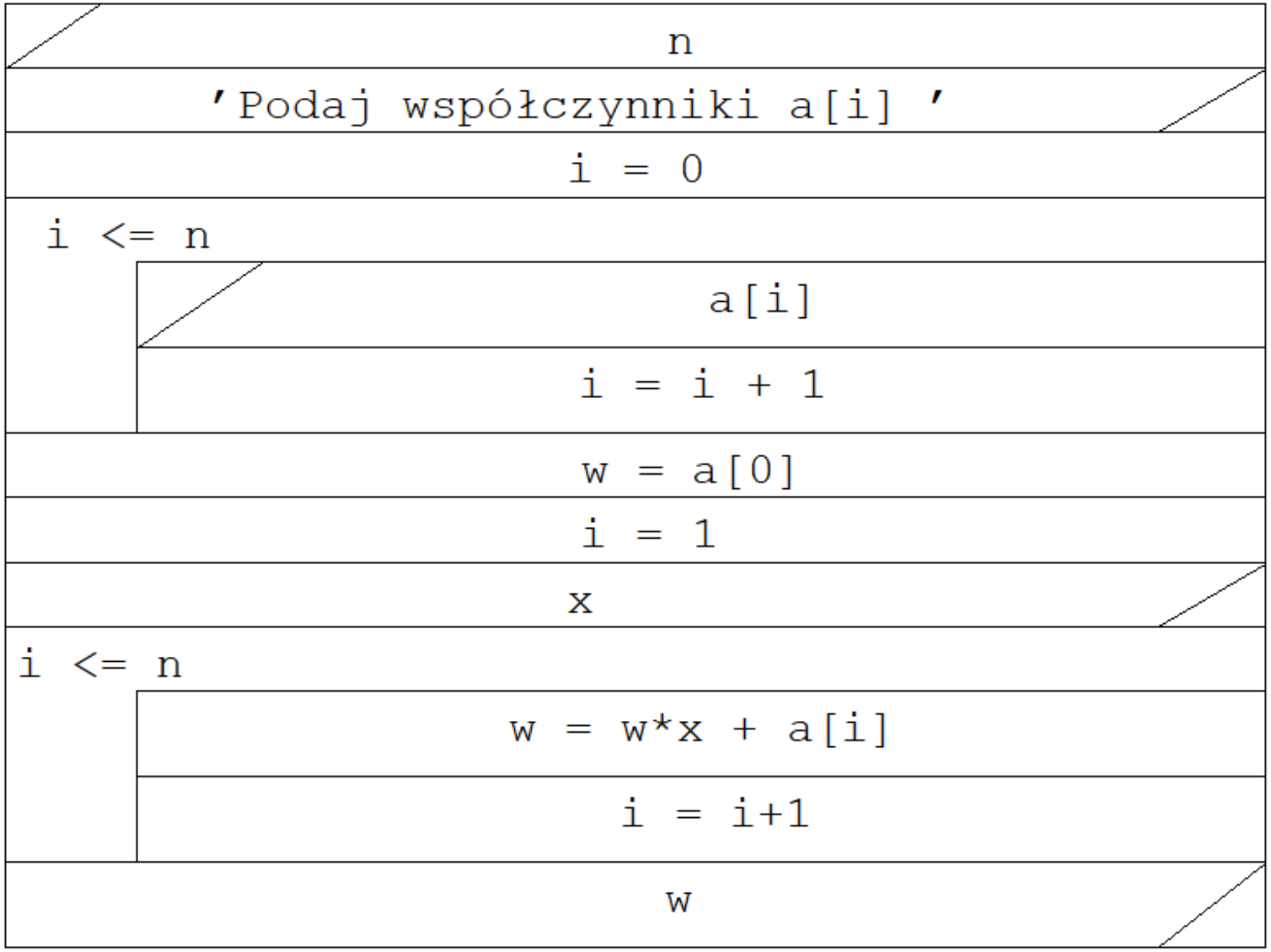
Tablice – zliczanie elementów



Tablice – rozwinięcie binarne



Schemat Hornera



Automaty komórkowe

- **Automat komórkowy** jest definiowany poprzez sieć komórek przestrzeni d-wymiarowej, zbiór stanów pojedynczej komórki oraz regułę określającą kolejny stan komórki w zależności od jej stanu i komórek ją otaczających.
- Najprostszymi, a jednocześnie posiadającymi wiele ciekawych własności, są automaty komórkowe jednowymiarowe, zwane też elementarnymi. Otocznie o promieniu równym 1 takiego automatu jest postaci:

- Komórki tych automatów, przyjmujące dwa możliwe stany: 0 lub 1, są ustawione wzdłuż jednej linii.
- Dzięki temu łatwo pokazać ewolucję takiego automatu, umieszczając kolejne układy jeden pod drugim.

Automaty komórkowe

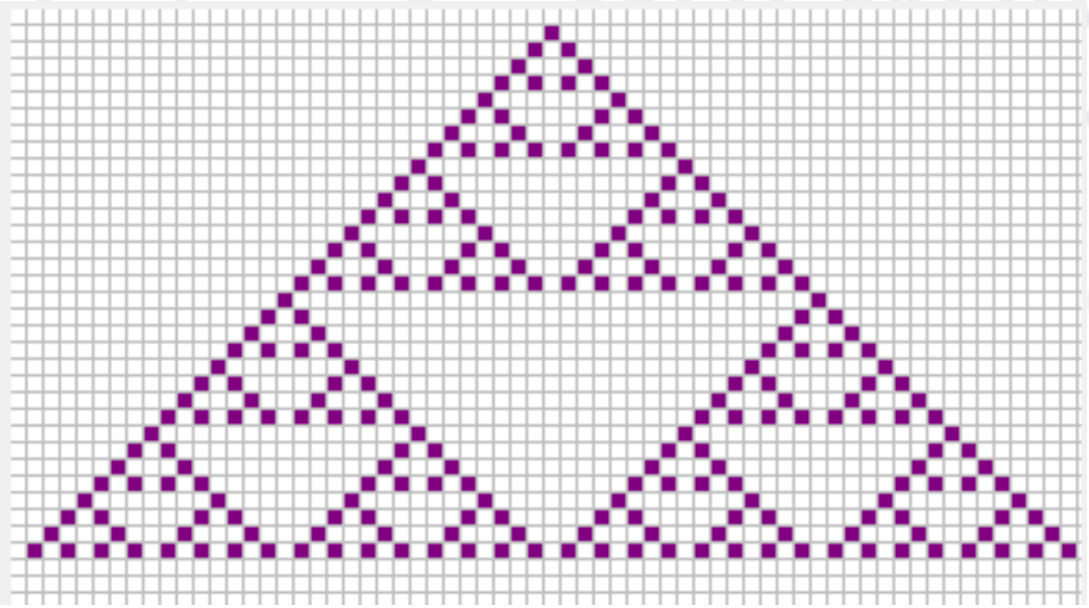
- Określenie nowego stanu komórki zależy od stanu samej komórki i liczby jej żywych sąsiadów. Jeżeli rozważymy trzy komórki (dana komórka i komórki z nią sąsiadujące), z których każda może być w jednym z dwóch stanów, to otrzymamy osiem możliwych konfiguracji. Każdej konfiguracji możemy przyporządkować stan życia 1 lub śmierci 0.
- Przykładową regułę ewolucji jednowymiarowego automatu komórkowego można przedstawić następująco:

111	110	101	100	011	010	001	000
↓	↓	↓	↓	↓	↓	↓	↓
0	1	0	1	1	0	1	0

- Reguła ta określana jest w postaci binarnej jako reguła 01011010 lub w zapisie dziesiętnym jako reguła 90.

Automaty komórkowe

- Jeżeli w wierszach umieścimy kolejne stany automatu posiadającego w stanie początkowym jedną żywą komórkę, działającego według reguły 90, to otrzymamy figurę przypominającą Trójkąt Sierpińskiego.



Automaty komórkowe

- Najbardziej znanym automatem komórkowym jest „Gra w życie”, automat wymyślony pod koniec lat 60. XX wieku przez angielskiego matematyka z Cambridge, Johna H. Conwaya.
- W roku 1970 Martin Gardner, prowadzący w czasopiśmie „Scientific American” dział gier matematycznych, rozpropagował pomysł Conwaya.
- „Gra w życie” dzieje się w przestrzeni i w czasie. Otoczenie komórki stanowią wszystkie komórki stykające się z nią bokiem lub rogiem (otocznie Moore’a). Każda komórka może mieć zatem co najwyżej ośmiu sąsiadów i może być w jednym z dwóch stanów: życia lub śmierci. W odmierzanym czasie komórki zmieniają swój stan zgodnie z regułami gry, tworząc kolejne pokolenia.

Zadanie 2.

Stan początkowy automatu komórkowego zapisany jest w tablicy n-elementowej. Napisz program wyznaczający kolejny stan tego automatu, działającego według reguły 90.

Funkcje, rekurencja

Funkcje

- **Pojęcie funkcji** jest bardzo przydatne w programowaniu **strukturalnym** i **obiektowym**.
- Funkcja jest fragmentem programu zaprogramowanego jako **oddzielna całość** , tak by jej używać w wielu różnych programach.
  - Taka funkcja będzie miała własne zmienne zadeklarowane w bloku funkcji, zwane zmiennymi lokalnymi.
  - Ponadto w funkcji będziemy używać parametrów zadeklarowanych w nagłówku funkcji. Ich nazwy muszą być inne niż nazwy zmiennych lokalnych.
  - Deklaracja funkcji jest jej nagłówkiem.
    Definicja funkcji zawiera nagłówek funkcji i blok funkcji.
- Wynik funkcji jest zwracany instrukcją

  ```
  return wyrażenie
  ```

  która przypisuje wartość obliczonego wyrażenia nazwie funkcji i przekazuje sterowanie z funkcji do programu.

Funkcje

- Funkcja może nie zwracać wartości pod swoją nazwą. W niektórych językach wprowadzamy wtedy pojęcie procedury.
- **Nazwa funkcji** jest identyfikatorem będącym ciągiem liter alfabetu łacińskiego, cyfr i podkreśleń, zaczynającą się od litery. Nazwa funkcji nie może być słowem kluczowym.
- Lista deklaracji parametrów może być pusta, jednoelementowa, wieloelementowa.
- Funkcję wywołujemy poprzez jej nazwę

  ```
  nazwa_funkcji(argumenty)
  ```

  lista argumentów musi być zgodna z listą parametrów funkcji z jej deklaracji.
- Nazwy argumentów mogą być różne od nazw parametrów.
- W schematach N-S proces już zdefiniowany (funkcja, procedura) jest oznaczany symbolem prostokąta z podwojonymi liniami .

Funkcje rekurencyjne

- Wiele problemów da się zapisać wzorem rekurencyjnym. Rząd tych wzorów mówi o liczbie poprzednich wielkości wymaganych do obliczenia kolejnej wielkości. Jest to liczba naturalna jeden, dwa, itd.
- Obowiązuje zasada, że wzory rekurencyjne rzędu 1 wymagają jednej zmiennej do ich zaprogramowania, rzędu 2 – dwóch zmiennych itd.
- Zwróćmy uwagę, że wzory rekurencyjne opisują algorytm Euklidesa oraz zamianę liczby dziesiętnej na jej postać binarną.
- Przy programowaniu wzorów rekurencyjnych korzystamy z mechanizmu pozwalającego na programowanie przy pomocy instrukcji `if`.
- Innymi słowy nie używamy w sposób jawny instrukcji pętli.

Funkcje rekurencyjne

- Funkcje rekurencyjne są to funkcje, których cechą charakterystyczną jest to, iż przed zakończeniem bieżącego wywołania funkcji następuje kolejne wywołanie tej funkcji, czyli w ciele danej funkcji następuje jej wywołanie.
- Dokładnie przed zakończeniem jednego wywołania następuje drugie wywołanie, trzecie wywołanie, itd.
- Wszystkie informacje o niedokończonych operacjach w trakcie kolejnych wywołań muszą być zapamiętane w pamięci, w jej fragmencie zwanym stosem podręcznym.
- Klasycznym przykładem funkcji rekurencyjnej jest silnia:

$$n! = \begin{cases} 1 & \text{dla} \quad n = 0 \\ n(n-1)! & \text{dla} \quad n > 0 \end{cases}$$

Silnia(n)

$$n! = \begin{cases} 1 & \text{dla} \quad n = 0 \\ n(n-1)! & \text{dla} \quad n > 0 \end{cases} \qquad \text{silnia}(n) = \begin{cases} 1 & \text{dla} \quad n = 0 \\ n\,\text{silnia}(n-1) & \text{dla} \quad n > 0 \end{cases}$$

Przykładowo dla *n*=4 mechanizm rekurencji spowoduje następujące wywołania funkcji silnia:

1-sze wywołanie: silnia(4), operacja odłożona s = 4 * silnia(3),
2-gie wywołanie: silnia(3), operacja odłożona s = 3 * silnia(2),
3-cie wywołanie: silnia(2), operacja odłożona s = 2 * silnia(1),
4-te wywołanie: silnia(1), operacja odłożona s = 1 * silnia(0),
5-te wywołanie: silnia(0), wynik = **silnia**(**0**) = 1**.**

Operacje odłożone wykonują się w odwrotnej kolejności:

s = 1 * 1 = 1
s = 2 * 1 = 2
s = 3 * 2 = 6
s = 4 * 6 = 24

Przykłady zastosowań

- Obliczanie *n*-tego wyrazu ciągu

$$f_n = \begin{cases} 0 & gdy \quad n = 0 \\ 1 & gdy \quad n = 1 \\ f_{n-1} + f_{n-2} & gdy \quad n > 1 \end{cases}$$

$$f_n = \begin{cases} 1 & gdy \quad n = 1,2 \\ \left(f_{\frac{n+1}{2}}\right)^2 + \left(f_{\frac{n-1}{2}}\right)^2 & gdy \quad n = 2k+1 \\ \left(f_{\frac{n+2}{2}} + f_{\frac{n-2}{2}}\right)\left(f_{\frac{n+2}{2}} - f_{\frac{n-2}{2}}\right) & gdy \quad n = 2k \end{cases}$$

Funkcje rekurencyjne

- Najsłynniejsze algorytmy rekurencyjne
  - Algorytm wieża Hanoi
  - Algorytm FFT
  - Algorytm QuickSort
- Algorytmy rekurencyjne stosuje się do takich zadań jak:
  - obliczanie symbolu Newtona,
  - obliczanie współczynników wielomianów Czebyszewa, Jacobiego,
  - przeglądanie drzew katalogów,
  - i wiele innych.
- Programista powinien pamiętać o tym, by konstruować funkcję opartą na rekurencji w ten sposób, aby jej wywoływanie nie odbywało się w nieskończoność. Musi być warunek stopu.
- Rekurencji nie należy nadużywać, rekurencja jest nieefektywna dla problemów rzędu większego niż 1.

Python – funkcje

Funkcje

W języku Python funkcje definiowane są za pomocą słowa kluczowego `def`, po którym następuje podanie

- nazwy funkcji,
- nawiasów okrągłych,
- opcjonalnych parametrów ,
- symbolu dwukropka na końcu.

O zasięgu ciała funkcji decyduje wcięcie. Każda funkcja domyślnie zwraca wartość `None`, chyba że wykorzystamy return.

`None` to specjalny wartość w Pythonie, która oznacza po prostu brak wartości.

Procedury są specyficznymi funkcjami, które nie zwracają wartości.

Funkcje

```
# Definiujemy funkcję
def Funkcja(bok):
    if bok > 0:
        pole = bok * bok
        print('Pole =', pole)
    else:
        print('Bledne dane')

# Wywołanie funkcji w kodzie
bok = 5
Funkcja(bok)
```

Funkcje

```
def maksimum(x, y):
    if x  > y:
          return x
     else:
          return y

# Wywołanie funkcji
x = 5
y = 6
print('Maks z liczb',x,y,'wynosi',maksimum(x, y))
```

Funkcje rekurencyjne

Funkcja rekurencyjna to taka funkcja, która wywołuje samą siebie oraz posiada warunek stopu, który **uniemożliwia** takie wywołania w nieskończoność, np.:

$$f_n = \begin{cases} 0 & gdy \quad n = 0 \\ 1 & gdy \quad n = 1 \\ f_{n-1} + f_{n-2} & gdy \quad n > 1 \end{cases}$$

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

Funkcje rekurencyjne

```
def  silnia(n):
    if n>1:
        return n*silnia(n-1)
    else:
        return 1
```

$$n! = \begin{cases} 1 & \text{dla} \quad n = 0 \\ n(n-1)! & \text{dla} \quad n > 0 \end{cases}$$

```
def stars(n):
    if n > 0:
        print('*', end=' ')
        stars(n-1)
```

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, n \in N_0, k = 1,\ldots,n-1,$$

$$\binom{n}{0} = 1 \quad \binom{n}{n} = 1$$

```
def symbol(n, k):
    if k == 0 or k == n:
        return 1
    else:
        return symbol(n-1, k-1) + symbol(n-1, k)
```

Zadanie 1.

Jaka będzie różnica w działaniu algorytmów:

- **Algorytm 1**

```
def bin(n):
    if n == 1:
        print(1, end=" ")
    else:
        bin(n//2)
        print(n % 2, end=" ")
```

- **Algorytm 2**

```
def bin(n):
    if n == 1:
        print(1, end=" ")
    else:
        print(n % 2, end=" ")
        bin(n//2)
```

Schemat Hornera

Zadanie 1.

Wyprowadź wzory i napisz funkcję zwracającą wartość sumy $\sum_{k=0}^{n} \frac{1}{k!}$ dla liczby naturalnej k podanej jako parametr.

Rozwiązanie:

$$S = \sum_{k=0}^{n} \frac{1}{k!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!} = \frac{1}{n!} + \frac{1}{(n-1)!} + \cdots + \frac{1}{2!} + \frac{1}{1!} + 1$$

$$= \left(\frac{1}{2 \cdot \ldots \cdot n} + \frac{1}{2 \cdot \ldots \cdot (n-1)} + \cdots + \frac{1}{2} + 1 \right) \cdot \frac{1}{1} + 1$$

$$= \left(\ldots \left(\left(\frac{1}{n} + 1 \right) \cdot \frac{1}{n-1} + 1 \right) \frac{1}{n-2} + \cdots + 1 \right) \frac{1}{1} + 1$$

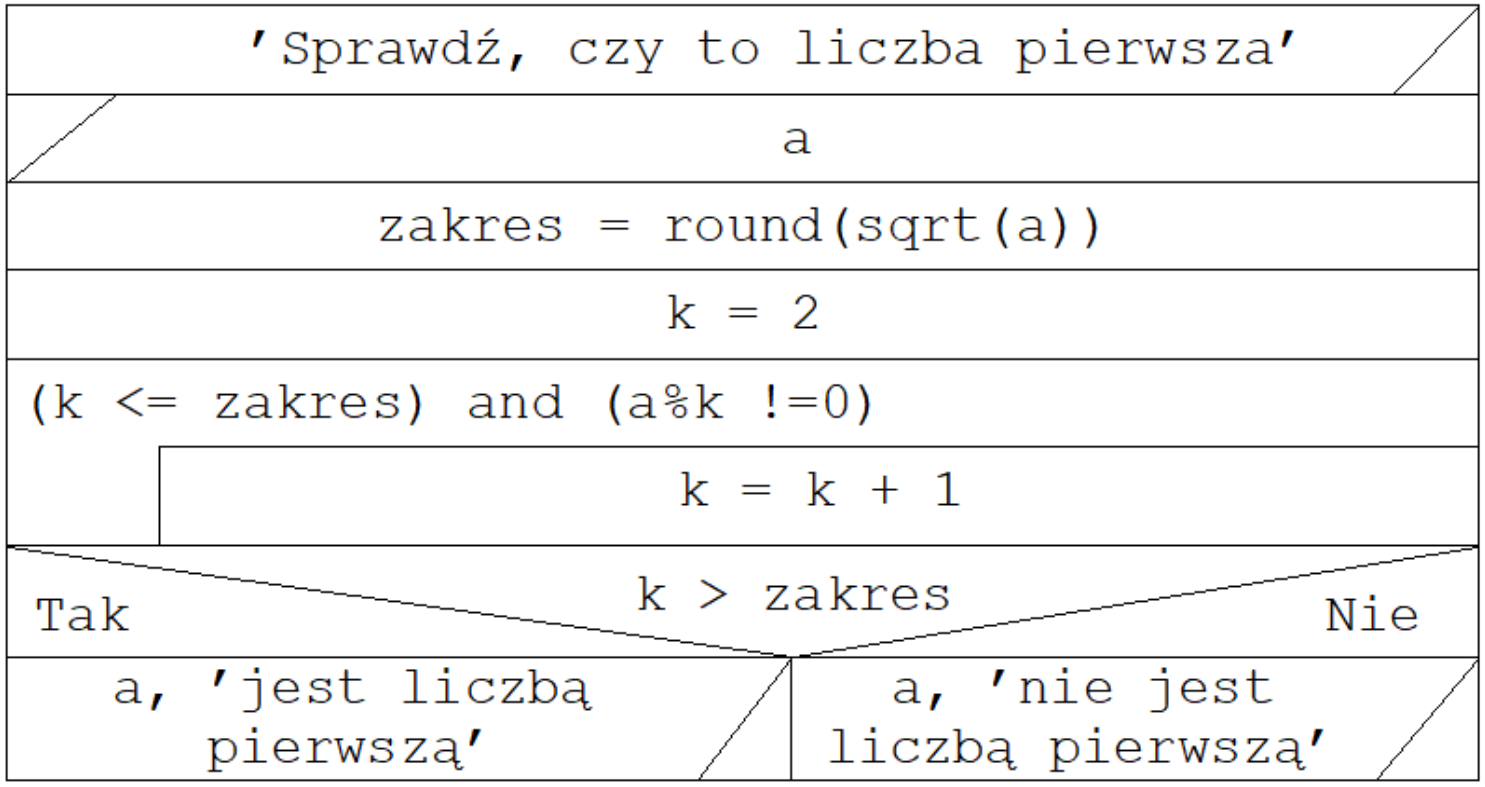
$w = 1$

$w = w * 1/k + 1$ dla $k = n, n-1, \ldots, 1$.

Liczby pierwsze

Zadanie 2.

Napisz funkcję sprawdzającą, czy liczba całkowita podana jako parametr jest liczbą pierwszą.



Moduły

Moduł math zawiera definicje podstawowych funkcji matematycznych:

```
from math import *

ceil(x)
floor(x)
fabs(x)
exp(x)
log(x) – logarytm naturalny
log(x, p)
pow(x, p)
sqrt(x)
sin(x)
cos(x)
tan(x)
…
```

Moduły

Funkcje można agregować i tworzyć moduły (biblioteki funkcji), które można zaimportować do wielu różnych programów przy pomocy instrukcji **`import.`**

Istnieje standardowa biblioteka Pythona, którą importujemy : **`import`** `sys`

Do funkcji i zmiennych wewnątrz modułu odwołujemy się przy pomocy kropki.

Można wybiórczo importować funkcje z wybranych modułów instrukcją

`from` … **`import`** …

na przykład

```
from sys import argv
```

lub zaimportować wszystkie funkcje z wybranego modułu

```
from sys import *
```

Należy jednak unikać importowania wszystkiego z każdego modułu, powoduje to zaśmiecenie przestrzeni nazw.

Importujemy tylko najczęściej stosowane funkcje w celu ominięcia konieczności stosowania przedrostków wskazujących na nazwy modułów.

Python – struktury danych

Lista

- W języku Python tablice są reprezentowane przy użyciu list, numerowanie komórek rozpoczyna się od indeksu zerowego.
- Lista to struktura liniowa zawierająca uporządkowany zestaw obiektów.
- Lista jest zmiennym typem danych umożliwiającym dodawanie, modyfikowanie i usuwanie elementów z listy.
- Elementy listy zapisujemy w nawiasach kwadratowych:

```
lista = ['ja', 'ty', 'my', 'on', 'ona', 'ono']
dane = [2, 5, 3, 4]
```

- Lista to sekwencje, których elementy są ponumerowane, można się do nich odwołać za pomocą indeksu:

```
dane[2] = 3
```

Listy – funkcja list()

Lista może zawierać elementy różnych typów:

```
lista1 =list('abc' 'xyz')
print(lista1)
```

```
['a', 'b', 'c', 'x', 'y', 'z']
```

Funkcja `list()` przekształca wszystkie obiekty w apostrofach na listę liter.

```
lista2 =(list(range(0,10)))
print(lista2)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Za pomocą funkcji `list()` została utworzona lista w zakresie, jaki zdefiniowała funkcja `range()`.

Lista - metody

- Lista może być pusta: `lista = []`
- Jej długość obliczymy przy pomocy funkcji `len(lista)`
- `append(x)`
  Dodaje element do końcu listy, odpowiednik `lista[len(a):] = [x]`
- `extend(lista)`
  Rozszerza listę poprzez dołączenie wszystkich elementów podanej listy, odpowiednik `a[len(a):] = lista`
- `insert(i, x)`
  Wstawia element na podaną pozycję listy. Pierwszym argumentem wywołania jest indeks elementu, przed którym nowy element ma zostać wstawiony: `a.insert(0,x)` wstawia na początek listy, `a.insert(len(a),x)` jest odpowiednikiem `a.append(x)`

Lista

- `remove(x)`
  Usuwa pierwszy napotkany element z listy, którego wartością jest `x`.
  Jeżeli nie ma na liście takiego elementu, zgłaszany jest błąd.
- `pop([i])`
  Usuwa element z podanej pozycji na liście i zwraca go jako wynik. Jeżeli nie podano żadnego indeksu `a.pop()`, zwraca ostatni element na liście. Oczywiście, jest on z niej usuwany.
- `index(x)`
  Zwraca indeks pierwszego elementu listy, którego wartością jest `x`.
  Jeżeli nie ma takiego elementu zgłaszany jest błąd.
- `count(x)`
  Zwraca liczbę wystąpień elementu `x` na liście.
- `sort()`
  Sortuje elementy listy w niej samej.
- `reverse()`
  Odwraca porządek elementów listy w niej samej.

Przykłady

- Zbudowanie tablicy w Pythonie:

```
tab = []
tab.append(1)
tab.append(2)
tab.append(3)
print(tab[0]) #wypisze 1
print(tab[1]) #wypisze 2
print(tab[2]) #wypisze 3

# wypisze kolejno 1, 2, 3
for x in tab:
    print(x)
```

Przykłady

- Najprostszym sposobem utworzenia tablicy o określonym rozmiarze jest zastosowanie techniki wielokrotnego powielenia wybranej wartości (najczęściej będzie to zero). Operacja ta przyjmuje następującą postać:

```
tab = [0]*30
print(tab)
```

  Efektem wykonania powyższego kodu będzie utworzenie trzydziestoelementowej tablicy wypełnionej zerami.

- Utworzenie tablicy `n`-elementowej:

```
n = int(input('Podaj rozmiar tablicy: '))
tab = [0]*n
print(tab)
```

Przykłady

- Pobranie od użytkownika `n` wartości i zapisanie ich w tablicy

```
n = int(input('Podaj dlugosc tablicy: '))
tab = [0]*n
i = 0
while i < n:
    tab[i] = int(input('Podaj liczbe: '))
    i += 1
```

- Wyświetlenie parzystych elementów tablicy (2 rozwiązania)

```
i = 0
while i < n:
    if tab[i] % 2 == 0:
        print(tab[i])
    i += 1
```

```
for element in tab:
    if element % 2 == 0
        print(element)
```

Techniki programowania

Strukturalne techniki algorytmizacji

Działania programisty powinny być ukierunkowane na tworzenie programów:

- **niezawodnych**, czyli poprawnych i odpornych,
- **przyjaznych dla użytkownika**:
  - uniwersalnych,
  - dobrze udokumentowanych,
  - zrozumiałych,
- **przyjaznych dla programisty**:
  - elastycznych,
  - przenośnych ,
  - testowalnych,
  - naprawialnych,
  - udokumentowanych,
  - czytelnych.
- **tanich** w aspektach czasu pracy programisty jak i czasu pracy komputera.

Idea programowania strukturalnego

Fundamentalna zasada konstrukcyjna nakazuje przy budowaniu programu stosowanie instrukcji:

- pętli „dopóki”,
- selekcji dwuwariantowej,
- sekwencji,
- oraz kompozycji algorytmicznych dających się do tych instrukcji sprowadzić.

Konstrukcje te wraz z wywoływaniem podprogramów w zupełności wystarczają do opisania **każdego** sensownego **algorytmu**.

Selekcja z wariantem pustym, selekcja wielowariantowa, pętla „aż do” czy też automatyczna pętla typu „for” są kompozycjami pochodnymi względem tych podstawowych, dopuszczalnymi w sensie **fundamentalnej zasady** i służą większej wygodzie programistów.

Idea programowania strukturalnego

W przypadku języka wysokiego poziomu oznacza to w praktyce **rezygnację z tzw. instrukcji skoku „goto”,** zaś w językach niższego poziomu stosowanie instrukcji skoku tylko do wyrażania tych podstawowych konstrukcji.

Wynika to z faktu, iż stosowanie „niekontrolowanych” instrukcji skoku stanowiło główne **źródło zawodności** i **braku elastyczności** programów komputerowych.

W technice programowania strukturalnego zalecany jest również **kontrolowany rozmiar i stopień komplikacji poszczególnych części programu**, pozwalający na objęcie tej całości wzrokiem i umysłem autora. Praktycznie oznacza to, iż powinna się ona mieścić na jednej stronie A4, a kod jednego bloku powinien mieścić się na ekranie.

Algorytmy strukturalne:

- są **proste w implementacji** w dowolnym języku programowania,
- umożliwiają znacznie **łatwiejsze dowodzenie ich poprawności**.

Style programowania

- **Top-down** – podejście zstępujące, analityczne:
  - zadanie dzielone jest na podzadania,
  - podzadania są rozwiązywane, a wyniki zapamiętywane.

- **Bottom-up** – podejście wstępujące, syntetyczne:
  - najpierw są rozwiązywane wszystkie podzadania, które mogą być potrzebne,
  - następnie ich wyniki są używane do rozwiązywania większych zadań.

Techniki algorytmiczne

- Proces budowania algorytmu rozwiązującego dane zadanie programistyczne ma charakter twórczy.
- Istnieją ogólne metody, które można zastosować w pewnych sytuacjach.
- Ich użycie prowadzi często do skonstruowania szybkich algorytmów.

Techniki algorytmiczne

- Algorytmy siłowe (brute force) – krokowe przeszukiwanie całej przestrzeni możliwych rozwiązań [wyszukiwanie liniowe].

  Algorytm siłowy zgodnie z ideą działania generuje wszystkie możliwe rozwiązania problemu i wybiera z nich to, które maksymalizuje funkcję celu.

- Algorytmy z powrotami (backtracking) – algorytmy rekurencyjne przeszukujące całą przestrzeń rozwiązań, ale z inteligentnym porzucaniem gałęzi nie rokujących nadziei na sukces. Cały proces rozwiązywania problemu można potraktować jako proces poszukiwań w drzewie podzadań.

  Drzewo podzadań zwykle rośnie wykładniczo i równie szybko rośnie ilość pracy związana z przeglądaniem drzewa. Najczęściej drzewo można stopniowo obcinać stosując reguły heurystyczne, co pozwala zredukować obliczenia. Algorytmy z powrotami uzyskały znaczenie dzięki maszynom cyfrowym, które mogą sprawdzać ogromną liczbę wariantów.

Techniki algorytmiczne

- Algorytmy *dziel i zwyciężaj* - algorytmy rekurencyjne dzielące problem na rozłączne podproblemy, polegające na:
  - rozkładzie zadania o dużym rozmiarze zwykle na dwa zadania tego samego rodzaju, ale o znacznie mniejszym rozmiarze,
  - następnie konstruowaniu rozwiązania zadania pierwotnego bezpośrednio z rozwiązań tych zadań prostszych.

  Naturalną konstrukcją algorytmiczną jest tu rekursja.
- Programowanie dynamiczne – poprawia często metodę *dziel i zwyciężaj*
- Algorytmy zachłanne - w każdym kroku dokonuje najlepiej rokującego w danym momencie wyboru rozwiązania częściowego.

Techniki algorytmiczne

- Algorytmy heurystyczne - rozwiązywanie problemu trudnego przez rozwiązanie problemu łatwiejszego, alternatywą dla tradycyjnych algorytmów rozwiązujących zadania optymalizacyjne. *Heurystyka* to zbiór specyficznych dla danego zadania reguł, które mogą dopomóc w odkryciu jak najlepszego rozwiązania. Skuteczności kroków heurystycznych nie można w pełni udowodnić teoretycznie, można jedynie pokazać doświadczalnie ich trafność.
- Algorytmy probabilistyczne lub randomizowane - wykorzystanie losowania w działaniu algorytmu. W praktyce oznacza to, że implementacja takiego algorytmu korzysta przy obliczeniach z generatora liczb losowych, np. *algorytmy Monte Carlo* (obliczanie liczby PI przez losowanie punktów z kwadratu z wpisanym kołem).
- Metody inteligencji obliczeniowej (algorytmy heurystyczne, stosowane do rozwiązywaniem problemów, które nie są efektywnie algorytmizowalne), obliczenia ewolucyjne (algorytmy genetyczne), obliczenia rozmyte, sieci neuronowe.
- Algorytmy kwantowe, zaimplementowane na komputerach kwantowych.

Metoda „dziel i zwyciężaj”

- Rozważmy **problem znajdowania elementu maksymalnego i minimalnego** w zbiorze n-elementowym S. Załóżmy, że n jest potęgą liczby 2.
- Oczywisty sposób znajdowania tych elementów to znajdowanie każdego z nich osobno. Dowolny element zbioru jest przyjmowany za max i porównywane są z nim wszystkie pozostałe elementy. Algorytm wymaga **n-1 porównań**.
- Podobnie za pomocą **n-2 porównań** można znaleźć minimum pozostałych n-1 elementów.
- Przy n>1 daje to **2n-3 porównań**.

- Według podejścia „dziel i zwyciężaj” zbiór S dzielony jest na dwa podzbiory, każdy o n/2 elementach.
- Elementy minimalny i maksymalny każdej z tych części będą znajdowane przez rekurencyjne zastosowanie algorytmu.
- Element maksymalny i minimalny będą obliczone z elementów maksymalnego i minimalnego obu podzbiorów przez dalsze dwa porównania.

Metoda „dziel i zwyciężaj”

- **Algorytm** MAXMIN

  WE: zbiór S o n elementach, gdzie n jest potęgą liczby 2 i n>1.

  WY: element minimalny i maksymalny zbioru S.

  Jeżeli moc zbioru S wynosi 2 tzn. S={a, b}

  return (MIN(a, b), MAX(a, b))

  w przeciwnym razie

  podziel S na dwa równe podzbiory S1 i S2

  (max1, min1) := MAXMIN(S1)

  (max2, min2) := MAXMIN(S2)

  return (MAX(max1, max2), MIN(min1, min2))

Metoda „dziel i zwyciężaj”

- **Analiza algorytmu**

  Niech T(n) będzie liczbą porównań pomiędzy elementami zbioru S, wymaganych przez MAXMIN. Mamy:

  $T(n) = 1$ dla $n=2$,

  $T(n) = 2T(n/2) + 2$ dla $n>2$.

  Funkcja $T(n) = 3/2 * n – 2$ jest rozwiązaniem tej rekurencji. Znaczy to, że aby znaleźć element minimalny i maksymalny w zbirze n liczb, konieczne jest przynajmniej $3/2*n – 2$ porównań pomiędzy elementami zbioru S.

  Algorytm jest optymalny w odniesieniu do liczby porównań, gdy n jest potęgą 2.

Programowanie dynamiczne

Programowanie dynamiczne poprawia metodę *„dziel i zwyciężaj ”* w sytuacji, gdy wymaga ona wielokrotnego liczenia rozwiązań tych samych podzadań. Przykładowo, liczenie współczynnika dwumianowego

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, n \in N_0, k = 1,\ldots, n-1,$$

$$\binom{n}{0} = 1 \qquad \binom{n}{n} = 1$$

wymaga powtarzania wyznaczania współczynników wielomianowych dla mniejszych wartości *n* i *k*.

Aby tego uniknąć można proces obliczeń rozpocząć od rozwiązania najmniejszych podzadań, zapisać rozwiązania w tablicy pomocniczej, a następnie użyć tych rozwiązań przy wyznaczaniu rozwiązań większych podzadań, aż do otrzymania rozwiązania całego zadania.

Metoda zachłanna

Gdy możliwych kombinacji danych, które mogą być rozwiązaniami, jest liczba wykładnicza, rozpatrywanie danych w kolejności uporządkowanej może prowadzić do zadowalającego rozwiązania.

Przykład

Mamy danych n przedmiotów o rozmiarach v_1, v_2, …, v_n i plecak o pojemności V.

Naszym celem jest wybranie pewnej liczby przedmiotów tak, by:

- zmieściły się w plecaku, tzn. v_1, v_2, …, v_n <= V,
- pozostające wolne miejsce w plecaku było jak najmniejsze.

W metodzie zachłannej pakujemy plecak od przedmiotów najmniejszych.

Znalezione rozwiązanie za pomocą tej metody nie musi być optymalne.

Algorytmy probabilistyczne

Test pierwszości to algorytm determinujący, czy liczba podana na wejściu jest pierwsza. Wszystkie testy pierwszości szukają, tak naprawdę, świadka na złożoność liczby. Jeśli go nie znajdą odpowiadają, że liczba jest pierwsza. Powszechnie rozważane są dwa rodzaje testów pierwszości:

- deterministyczne - odpowiadają że podana na wejściu liczba *n* jest pierwsza wtedy i tylko wtedy gdy naprawdę tak jest,
- probabilistyczne - przepuszczają pewne liczby złożone (stwierdzając ich pierwszość), nigdy jednak nie mylą się na liczbach pierwszych.

Test pierwszości Fermata

Mając daną na wejściu liczbę *n:*

- Wylosuj $b \in \{1,2,\ldots,n-1\}$.
- Sprawdź algorytmem Euklidesa, czy $d=\text{NWD}(b,n)>1$. Jeśli tak, to *n* jest złożona, co więcej *d* jest nietrywialnym dzielnikiem *n*.
- Sprawdź czy $b^{n-1} \equiv_n 1$.

Jeśli nie, to *n* jest złożona i *b* jest świadkiem złożoności *n*. Jeśli tak, to *n* przechodzi test.

Metoda Monte Carlo

- Metody **Monte Carlo** są metodami obliczeniowymi, w których ważną role odgrywają generatory liczb losowych.
- Termin „Monte Carlo”, zaczerpnięty z klasycznego generatora jakim jest ruletka, został wprowadzony przez Nicolasa Metropolisa w 1949 roku. Metody te stosowane są do rozwiązywania zadań w wielu dziedzinach nauki i techniki. Początkowo wykorzystywane były w zagadnieniach fizyki neutronowej. Obecnie w znacznym stopniu korzysta się z tych metod w teorii gier, ekonomii matematycznej, fizyce statystycznej czy teorii przesyłania sygnałów w warunkach zakłóceń.
- Jednym z ważnych problemów metod numerycznych jest problem całkowania. W przypadku całek wielowymiarowych liczenie całek klasycznymi metodami jest nieefektywne, a czas szacowania jest bardzo długi. Rozwiązaniem tego problemu są metody Monte Carlo.
- Metoda orzeł-reszka polega na szacowaniu całek, które oblicza się jako stosunek liczby trafień (sukcesów) do liczby prób losowych.

Metoda Monte Carlo

Tajemnica pitagorejczyków

Załóżmy, że chcemy symulować rzucanie ziaren piasku na kwadratowy stół. Jeżeli na tym stole wykreślimy ćwierć okręgu o promieniu równym krawędzi stołu, to możemy zastanawiać się, ile ziaren spadnie wewnątrz ćwiartki okręgu, a ile poza nią. Dla uproszczenia przyjmiemy, że krawędź stołu, a więc i promień okręgu jest równy 1 metr. Musimy zapewnić całkowitą przypadkowość miejsca, na które zostanie upuszczone ziarno. Jako wynik interesuje nas pomnożony przez 4 stosunek liczby ziaren z okręgu do wszystkich rzuconych. Uzyskany wynik to stała Archimedesa.

Doświadczenie, które symulujemy, zaproponowane zostało w drugiej połowie XIX wieku przez szwajcarskiego astronoma Rudolfa Wolfa i znane jest jako **metoda Monte Carlo**.

Funkcje generujące liczby losowe

Generatory liczb losowych

- Symulacyjne badania modeli probabilistycznych, przeprowadzane w naukach technicznych, ekonomicznych i przyrodniczych, wymagają wyposażenia komputera w odpowiednie narzędzia, jakimi są generatory liczb losowych.
- Z uwagi na deterministyczny charakter generowania liczb losowych otrzymane liczby nie są w rzeczywistości liczbami przypadkowymi. Z tego względu liczby generowane przez programy komputerowe nazywane są liczbami pseudolosowymi.
- Ciągi pseudolosowe mają wszystkie własności ciągów losowych, które efektywnie można przetestować. Jedynym losowym elementem takiego ciągu jest ziarno, czyli inaczej wartość początkowa. Ziarno musi być na tyle długie, by wykluczyć możliwość jego odtworzenia poprzez przeglądanie wszystkich ziaren ciągów przez nie generowanych.

Generatory liczb losowych

- Najprostszymi generatorami liczb losowych są generatory fizyczne, takie jak moneta, urna, czy ruletka. Współcześnie generatory fizyczne zostały całkowicie wyparte przez generatory programowe, a te pierwsze służą tylko do ich inicjowania.
- Otrzymanie liczb losowych o rozkładzie równomiernym jest najważniejszym elementem generowania liczb losowych o dowolnym rozkładzie prawdopodobieństwa .

Generatory liczb losowych

Podstawowym w wielu zastosowaniach generatorem jest generator liniowy. Jest on postaci:

$$x_{n+1} = (a_1 x_n + a_2 x_{n-1} + \ldots + a_k x_{n-k+1} + c) \bmod m \ ,$$

gdzie $a_1, a_2, \ldots a_k, c$ oraz m są ustalonymi liczbami całkowitymi zwanymi parametrami generatora. Inicjując działanie generatora, użytkownik dostarcza dane początkowe $x_0, x_1, \ldots, x_k$. Najczęściej przyjmuje się $k = 1$. Wtedy:

$$x_{n+1} = (a x_n + c) \bmod m \ .$$

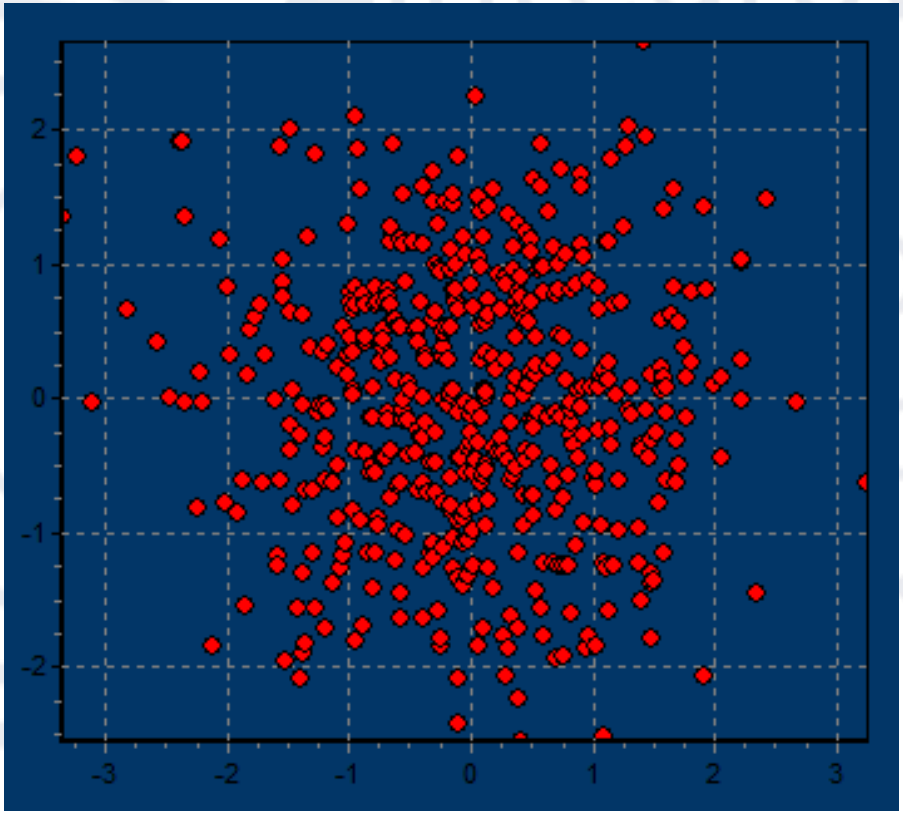
Za pomocą metody Boxa–Mullera możemy otrzymać jednocześnie dwie niezależne zmienne losowe o rozkładzie normalnym $N(0,1)$ określone wzorami:

$$X_1 = (-2 \log U_1)^{1/2} \sin(2\pi U_2) \ ,$$

$$X_2 = (-2 \log U_1)^{1/2} \cos(2\pi U_2)$$

gdzie U_1, U_2 są dwiema niezależnymi zmiennymi losowymi o rozkładzie równomiernym $U(0,1)$.

Generatory liczb losowych



Liczby losowe (X_1, X_2) o rozkładzie normalnym, metoda Boxa-Mullera.

Generatory liczb losowych

Większość zadań tekstowych korzysta z liczb pseudolosowych, najprostszym sposobem jest wykorzystanie funkcji z biblioteki `random`

```
randint(a, b)
```

gdzie `a` – zakres dolny losowania, `b` – zakres górny.

Przykładowo, w celu wylosowania liczby od 1 do 10 piszemy:

```
liczba = randint(1, 10)
```

Przykłady

```
from random import randint

print("Liczba losowa", randint(1, 10))
```

Problemy rozwiązywane z użyciem rekurencji

Sortowanie bąbelkowe

Zadanie sortowania: dana jest lista elementów zbioru liniowo uporządkowanego. Naszym zadaniem jest dokonanie permutacji ustawiającej elementy w porządku niemalejącym.

- **Sortowanie bąbelkowe** polega na braniu w trakcie jednej iteracji par kolejnych liczb oraz porównanie ich ze sobą:
  - jeżeli pierwsza liczba jest mniejsza od drugiej nie zachodzą żadne zmiany,
  - w przeciwnym razie zamienia się je miejscami w tablicy.

  Powyższe powtarzamy n-1 razy.

W wersji optymalnej:

- Za każdym razem przeglądamy ciąg o jeden element krótszy.
- Przeglądanie kontynuujemy, dopóki były przestawienia.

Sortowanie bąbelkowe to algorytm o złożoności obliczeniowej klasy $O(n^2)$, gdzie n to liczba sortowanych elementów.

Można je porównać do bąbelka powietrza uwięzionego w wodzie, który próbuje się uwolnić poprzez unoszenie się do góry.

Sortowanie bąbelkowe

5	4	2	3	6

5	4	2	3	6
4	5	2	3	6
4	5	2	3	6
4	2	5	3	6
4	2	5	3	6
4	2	3	5	6
4	2	3	5	**6**

4	2	3	5	6
2	4	3	5	6
2	4	3	5	6
2	3	4	5	6
2	3	4	5	6
2	3	4	5	6
2	3	4	**5**	**6**
itd.				

Sortowanie bąbelkowe

Algorytm

Ciąg n-elementowy: `a[0], a[1], ..., a[n-1]`

Krok 1.

```
zakres = n-1
przestaw = true
```

Krok 2.

```
while przestaw == true
    2.1. przestaw = false
    2.2. i = 0
    2.2. while i<zakres
            if a[i] > a[i+1]
               przestaw = true
               rob = a[i]
               a[i] = a[i+1]
               a[i+1] = rob
            i = i+1
    2.3. zakres = zakres - 1
```

Quick Sort

Algorytm (szybkie sortowanie)
ciąg: `a[l], a[l+1], ..., a[p]), l ≤ p`

krok 1.
Wybieramy wyraz środkowy ciągu (element progowy)
`x=a[(l+p)/2]`
kładziemy `i=l, j=p`

krok 2.
znajdź najmniejszy indeks `i`, taki że `a[i] ≥ x`
znajdź największy indeks `j`, taki że `a[j] ≤ x`
jeżeli `i ≤ j` przestaw wyrazy `a[i]` oraz `a[j]` i dokonaj przypisań `i=i+1;j=j-1;`
Czynności z kroku 2 wykonuj dopóty, dopóki `i ≤ j`, zaczynając poszukiwanie od ostatnich ich wartości

krok 3.
Dla indeksów `i>j` z kroku 2 powtarzaj kroki 1 i 2 dla dwóch podciągów `(a[l],...,a[j]),l<j` oraz `(a[i],...,a[p]),i<p`, potem dla 4-ech, 8-miu, itd.
Znakiem końca algorytmu jest otrzymanie wszystkich ciągów jednoelementowych.

Quick Sort

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
27	55	25	33	75	11	48	61
27	11	25	33	*75*	*55*	*48*	*61*
27	11	25	33	*75*	*55*	*48*	*61*
27	11	25	33	*75*	*55*	*48*	*61*
11	27	25	33	*75*	*55*	*48*	*61*
11	25	27	33	*75*	*55*	*48*	*61*
11	25	27	33	*75*	*55*	*48*	*61*
11	*25*	*27*	*33*	75	55	48	61
11	*25*	*27*	*33*	75	55	48	61
11	*25*	*27*	*33*	48	55	75	61
11	*25*	*27*	*33*	48	55	75	61
11	*25*	*27*	*33*	48	55	61	75
11	25	27	33	48	55	61	75

Quick Sort

```
def qsort(T, l=0, r=None):
    if r is None: r = len(T) - 1
    i, j = l, r
    pivot = T[(l + r) / 2]
    while i <= j:
        while T[i] < pivot: i += 1
        while T[j] > pivot: j -= 1
        if i <= j:
            T[i], T[j] = T[j], T[i]
            i += 1
            j -= 1
    if l < j: qsort(T, l, j)
    if r > i: qsort(T, i, r)
```

Przeszukiwanie ciągu liczb

- Problem: sprawdzić, czy w ciągu liczb istnieje podana liczba, zwrócić indeks jej pierwszego wystąpienia.
- Rozważamy dwa przypadki:
  - ciąg liczb nie jest posortowany,
  - ciąg liczb jest posortowany.
- W przypadku ciągu nieposortowanego sprawdzamy kolejno, wyraz po wyrazie, czy występuje dany element.
- W przypadku ciągu posortowanego sprawdzanie może być zrobione dużo szybciej za pomocą wyszukiwania binarnego. W każdym kroku algorytmu przeszukiwany ciąg skraca się o połowę.

Wyszukiwanie binarne

- Problem wyszukiwania danego elementu **a** w tablicy `T` uporządkowanej rosnąco w przedziale indeksów od `left` do `right`.

- **Dopóki** przedział indeksów nie jest pusty, czyli `left <= right`.
  - Kładziemy `k = (left + right)/2`
  - Porównujemy **a** z elementem środkowym `T[k]`.
  - Jeżeli **a** == `T[k]`, to zwracamy indeks `k`.
  - Jeżeli **a** < `T[k]`, to stosujemy to postępowanie do przedziału indeksów od `left` do `k-1`.
  - Jeżeli **a** > `T[k]`, to stosujemy to postępowanie do przedziału indeksów od `k+1` do `right`.

- Po znalezieniu elementu funkcja zwraca jego indeks.
- W przypadku niepowodzenia funkcja zwraca wartość, która nie jest prawidłowym indeksem tablicy.

Wyszukiwanie binarne: a =25, n=10

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	Indeks elementu środkowego	porównanie
1	**5**	**7**	**11**	**13**	**15**	**20**	**21**	**25**	**45**	(0+9)/2 = 4	25 == 13
1	**5**	**7**	**11**	**13**	**15**	**20**	**21**	**25**	**45**	(5+9)/2 = 7	25 == 21
1	**5**	**7**	**11**	**13**	**15**	**20**	**21**	**25**	**45**	(8+9)/2 = 8	**25 == 25**

Zadanie

Zaproponuj rekurencyjną wersję algorytmu wyszukiwania binarnego.

Wyszukiwanie binarne

```
def binarne(T, left, right, a):
    while left <= right:
        k = (left+right)//2
        if a == T[k]:
            return k
        if a < T[k]:
            right = k-1
        else:
            left = k+1
    return None
```

Dla tablicy `n`-elementowej indeksowanej od 0 wywołanie funkcji:

```
left = 0, right = n-1
```

Wyszukiwanie sekwencyjne

```
def sekwencyjne(T, n, a):
    for k in range(0,n):
        if a == T[k]:
            return k
    return None
```

Porównanie algorytmów wyszukiwania

- Mówimy, że liczba operacji jest $O(\log_2 n)$, jeżeli istnieje stała C, niezależna od n, dla której liczba operacji jest równa $C\log_2 n$.

Możliwa liczba porównań w algorytmach wyszukiwania		
n	Tablica nieposortowana	Tablica posortowana
2^5	2^5	5
2^{10}	2^{10}	10
2^{20}	2^{20}	20

Kodowanie znaków

Kody Huffmana

David A. Huffman (1925-1999) – informatyk amerykański.

- Pracował jako wykładowca w MIT, a od 1967 na University of California w Santa Cruz, gdzie organizował wydział nauk informatycznych.
- Najbardziej znanym osiągnięciem Huffmana jest opracowana w 1952 metoda kompresji bezstratnej, tzw. kodowanie Huffmana.
- Huffman nie patentował swoich osiągnięć, koncentrując się na pracy edukacyjnej.
- Został uhonorowany wieloma prestiżowymi nagrodami informatycznymi, m.in. Medalem Hamminga (1999 r.).

Kody Huffmana

- Kody Huffmana to jedna z powszechnie stosowanych metod kompresji danych. Metoda ta jest oparta o strategię zachłanną.
- Kompresja polega na zastępowaniu znaków kompresowanego pliku krótszymi kodami, dzięki czemu wynikowy ciąg bitów jest znacznie krótszy od wejściowego.
- Znaki są kodowane na 8 bitach, dzięki czemu można zakodować do 256 znaków (kody ASCII od 0 do 255).
- Bardzo rzadko się jednak zdarza, by w pliku występowały wszystkie znaki ASCII, wobec tego można wykorzystać mniej bitów i przez to zaoszczędzić dużo miejsca.
- Znaki można kodować na dwa sposoby:
  - za pomocą kodów o stałej długości,
  - za pomocą kodów o zmiennej długości.

Kody Huffmana

Metoda pierwsza jest bardzo prosta, lecz mniej skuteczna niż druga. Polega ona na przypisaniu znakom liczb porządkowych:

- Załóżmy, że plik został zapisany za pomocą znaków alfabetu A={a, b, c, d, e).
- Alfabet ten składa się z 5 znaków jest to dużo mniej niż 256.
- Nie potrzebujemy 8 bitów, wystarczą nam 3, gdyż $2^3=8$, za pomocą 3 bitów można zapisać 8 znaków.
- Gdyby nasz alfabet składał się z 4 znaków, wystarczyłyby tylko 2 bity, gdyż $2^2=4$.

Kody Huffmana

Alternatywą dla opisanych powyżej kodów o stałej długości są kody o długości zmiennej. Wykazują one lepsze własności dla znaków częściej występujących.

- Zauważmy, że lepiej jest przypisać mniej bitów znakom częściej występującym i więcej bitów znakom rzadziej występującym.
- W przypadku kodów o stałej długości dekompresując plik nie było problemu z pobieraniem kolejnych kodów z ciągu bitów (każdy miał np. 3 bity). Jeśli kody mają różną długość nie wiadomo, kiedy kończy się stary kod i zacznie nowy.
- Dlatego należy posługiwać się kodami *prefiksowymi* tzn. takimi, by żaden kod znaku nie był prefiksem innego znaku, czyli nie był częścią początkową innego kodu.

Kody Huffmana

Kodowanie Huffmana daje kod optymalny. Algorytm Huffmana przebiega w dwóch częściach:

- zbudowanie drzewa binarnego,
- bezpośrednie kodowanie.

Część I.

Zbudowanie drzewa

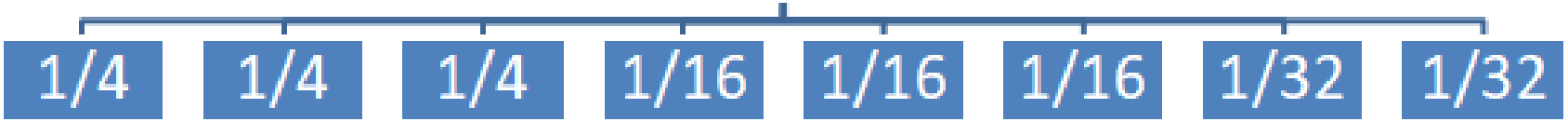
- Kryterium łączenia:

  dwa drzewa o najmniejszych wartościach korzenia stają się potomkami nowego drzewa, nadając mu wartość korzenia równą sumie korzenia wartości dzieci.

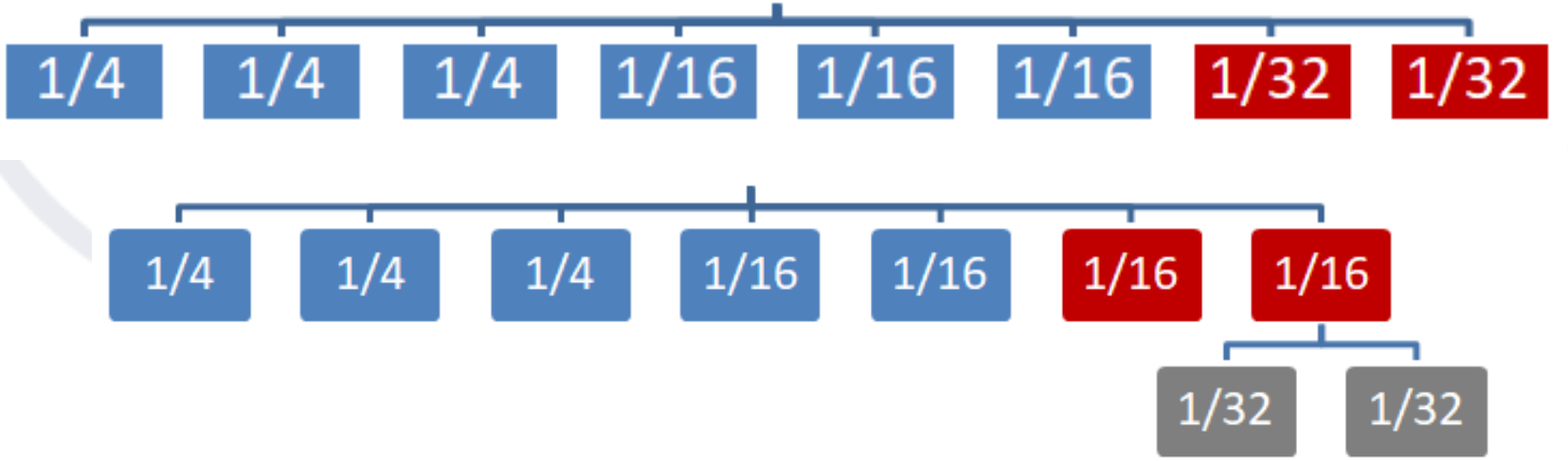
Kody Huffmana

Niech dane będą prawdopodobieństwa :

1/4, 1/4, 1/4, 1/16, 1/16, 1/16, 1/32, 1/32

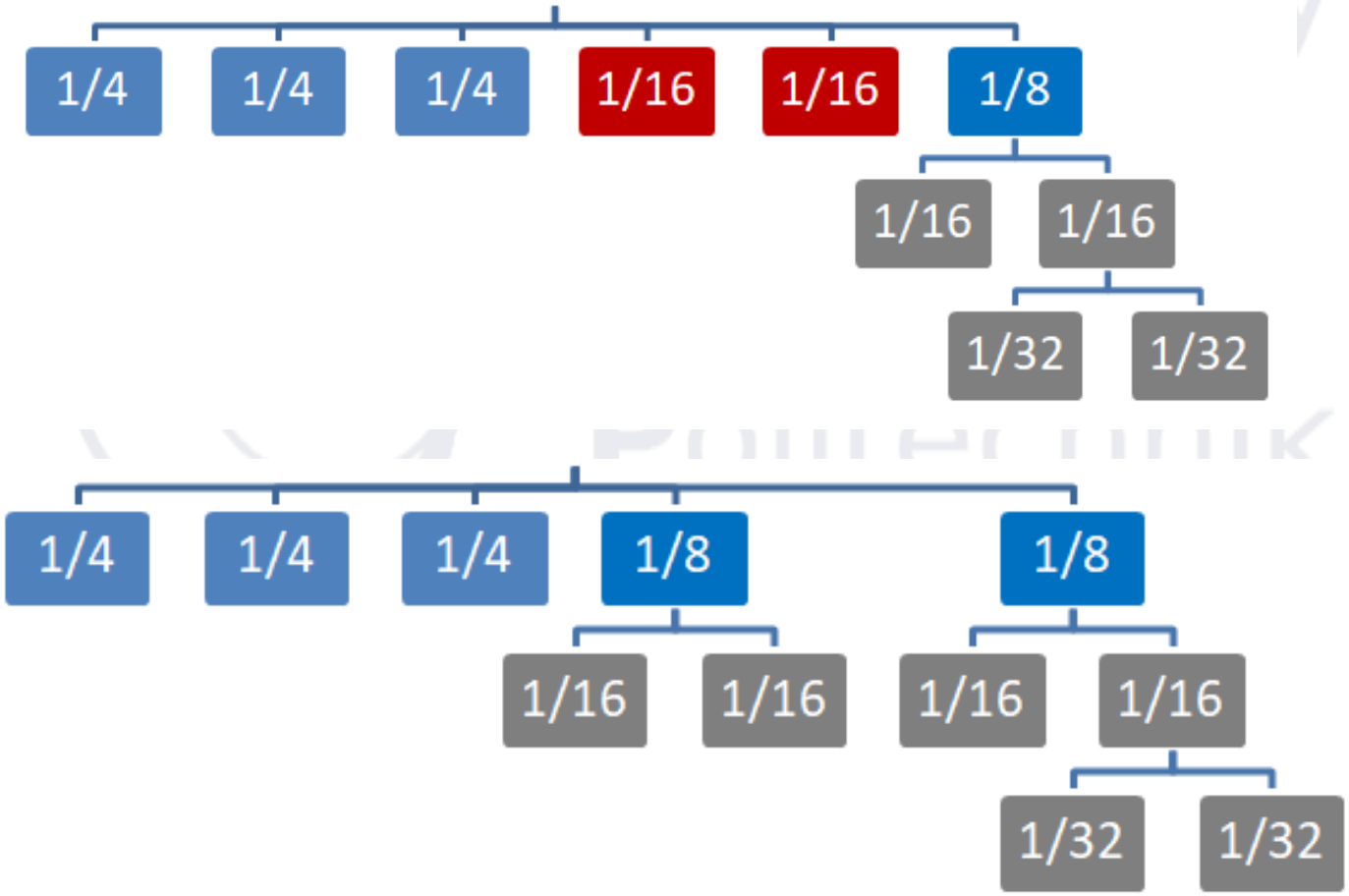


Krok 1. Łączymy drzewa o najmniejszych prawdopodobieństwach



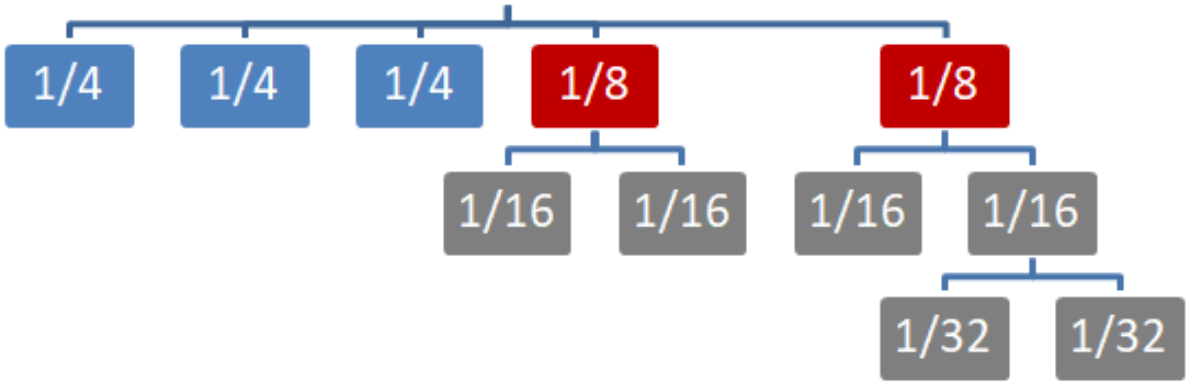
Kody Huffmana

Krok 2.

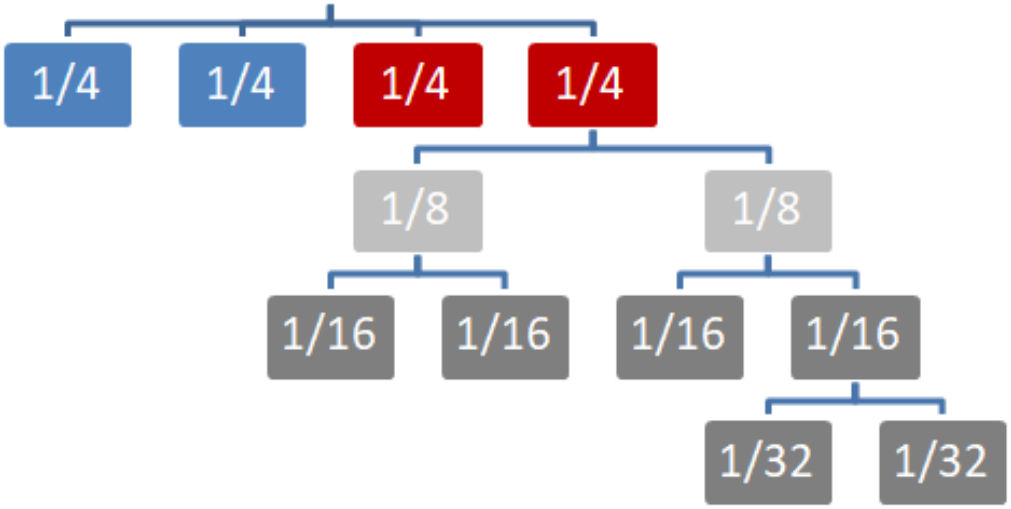


Kody Huffmana

Krok 3.

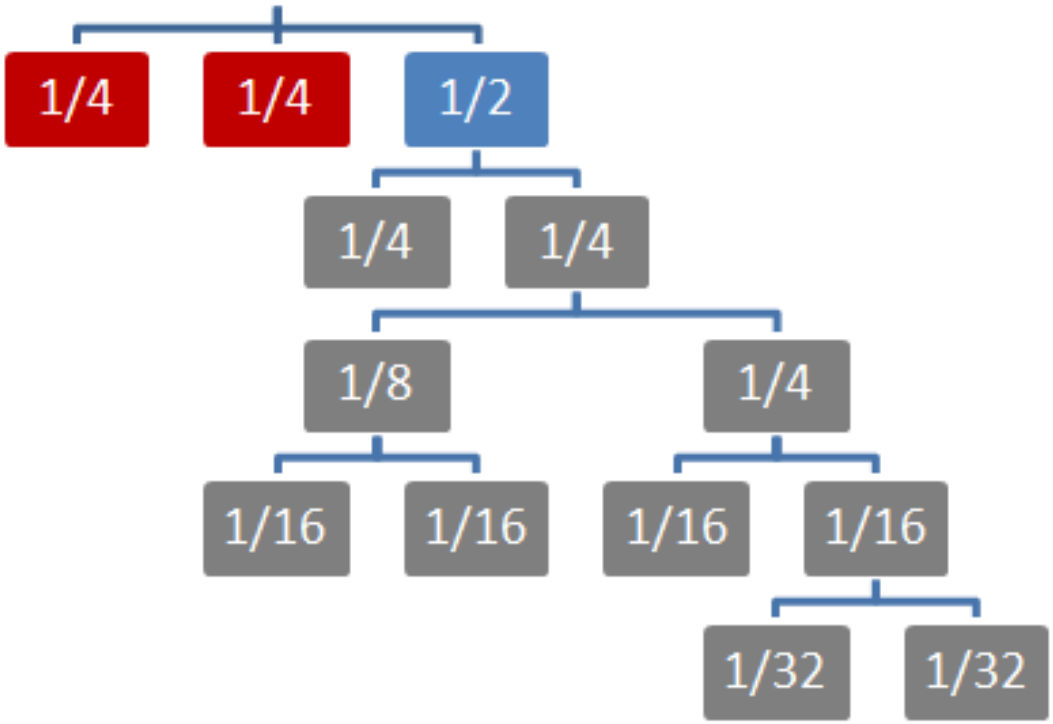


Krok 4.



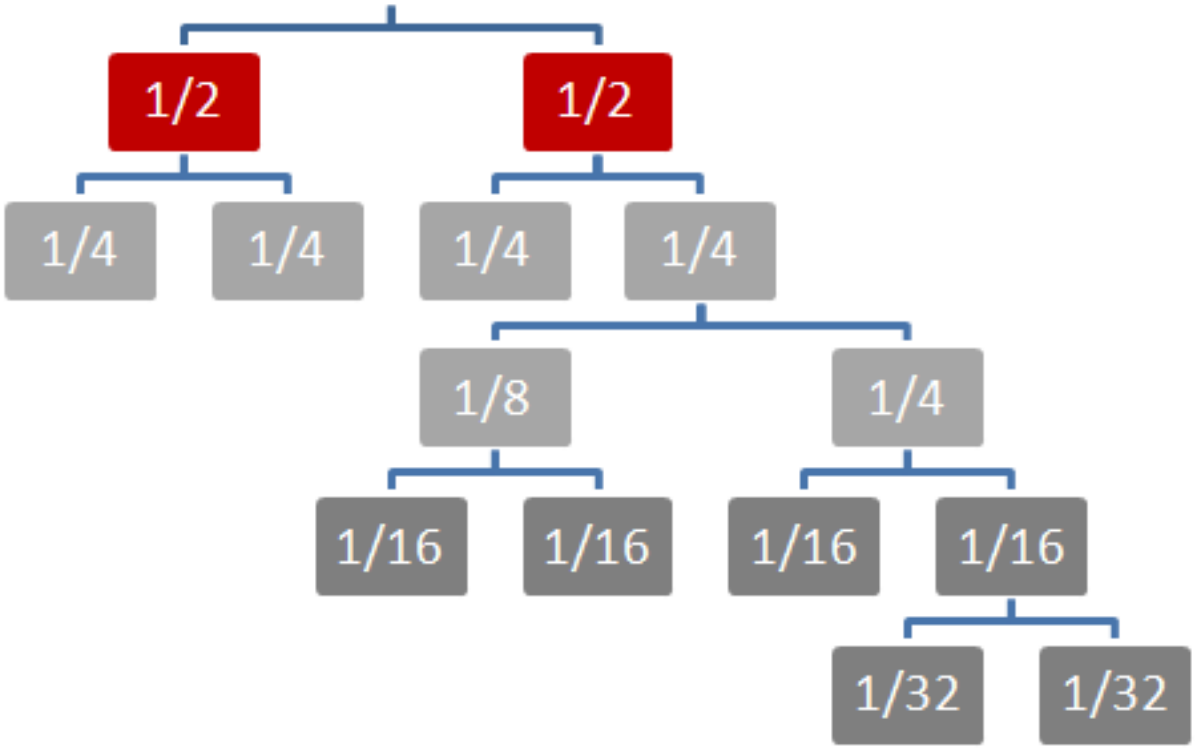
Kody Huffmana

Krok 5.



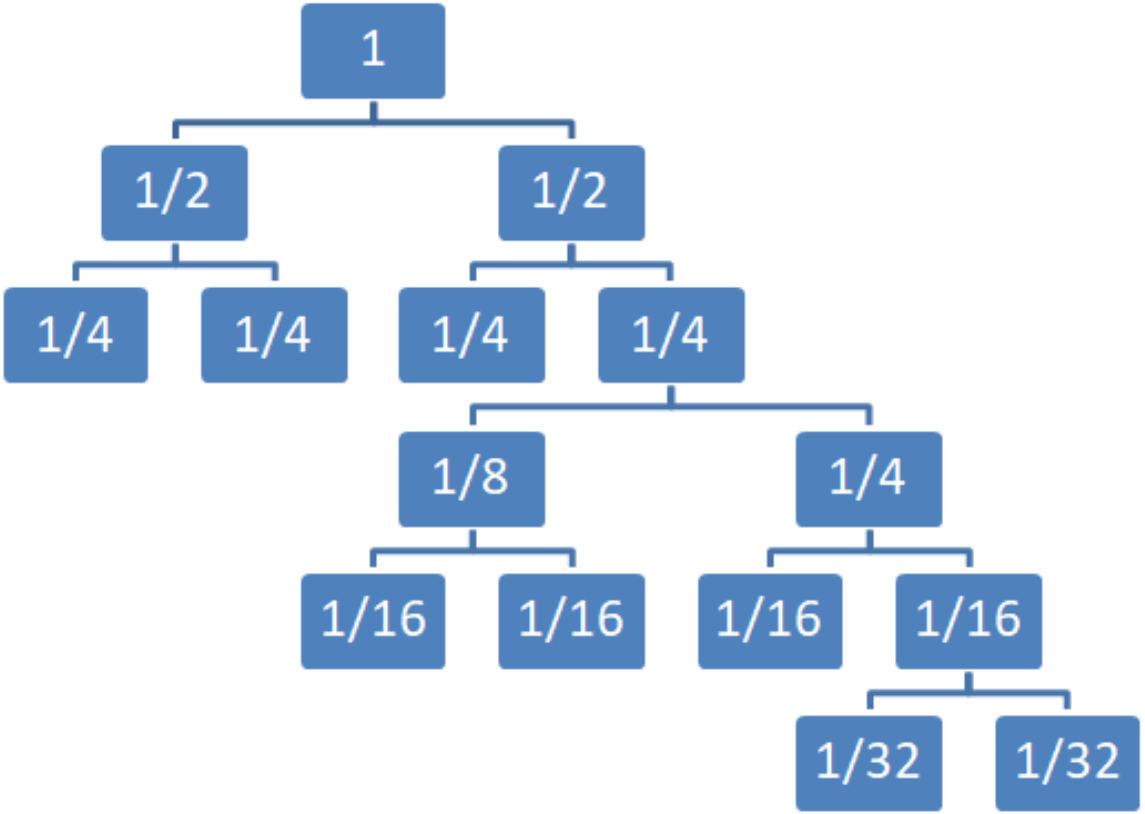
Kody Huffmana

Krok 6.



Kody Huffmana

Krok 7.



Kody Huffmana

Drzewo binarne zbudowane w pierwszej części algorytmu z liśćmi w postaci wejściowych prawdopodobieństw stanowi podstawę drugiej części algorytmu, w którym węzłom drzewa przypisywane są kody. Korzeń drzewa otrzymuje początkowo kod pusty.

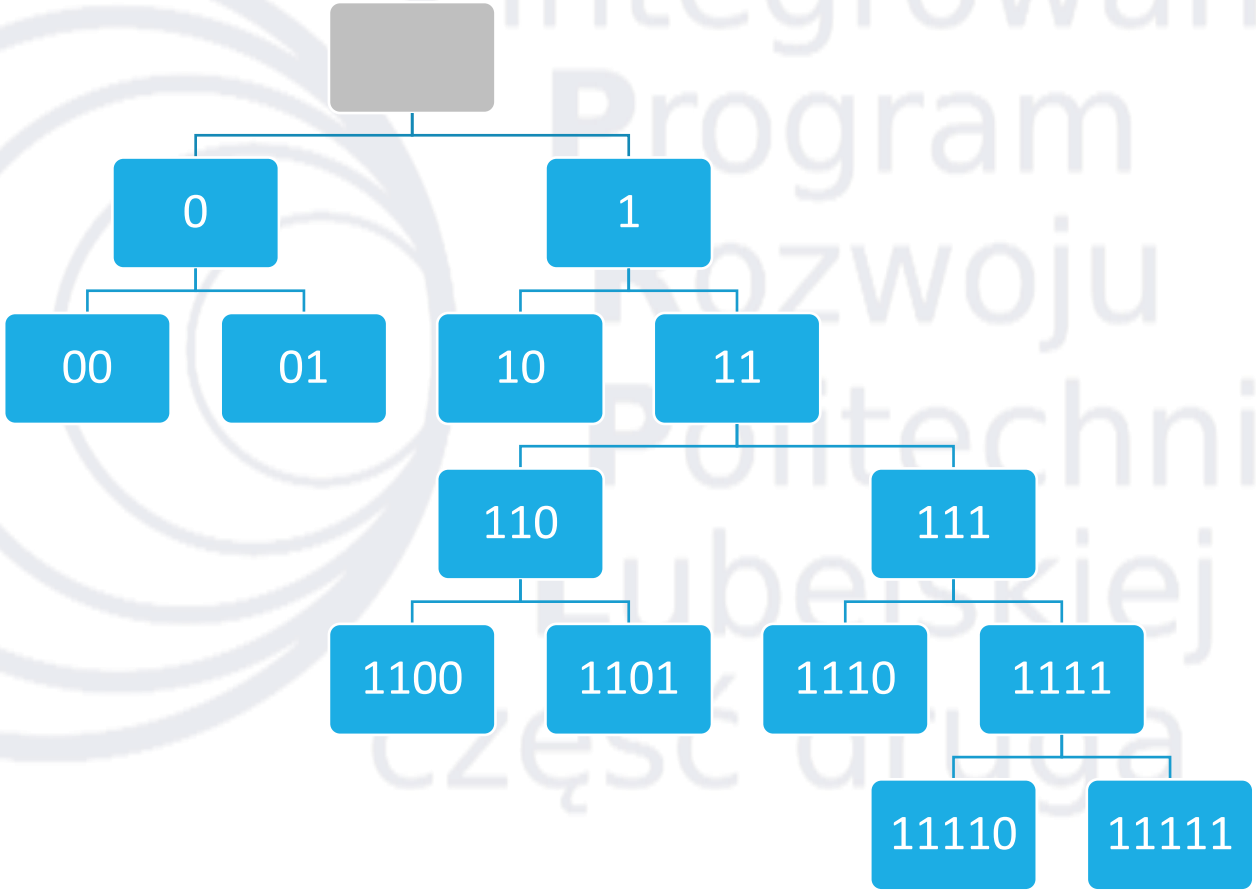
Część II.

Procedura kodowania

- Jeżeli węzeł ma lewe dziecko, przypisz mu kod rodzica z dodanym pierwszym znakiem kodowania, a następnie wykonaj opisywane czynności dla lewego dziecka.
- Jeżeli węzeł ma prawe dziecko, przypisz mu kod rodzica z dodanym drugim znakiem kodowania, a następnie wykonaj opisywane czynności dla prawego dziecka.

Kody Huffmana

Krok 8.



Kody Huffmana

- Niskie prawdopodobieństwa na wczesnym etapie przebiegu algorytmu będą stanowiły minimalne wartości, dlatego w drzewie znajdują się daleko od korzenia i otrzymają długie kody.
- Wysokie prawdopodobieństwa znajdują się blisko korzenia i otrzymują krótsze kody.
- Algorytm Huffmana nie jest jednoznaczny, ponieważ wiele możliwości wyboru drzew optymalnych prowadzi do innych kodowań.
- Dla sprawdzenia efektywności kodowania wyznaczamy entropię *H*. Efektywność kodowania wyraża się wzorem

$$H/L * 100\%,$$

gdzie *L* to średnia długość kodowania.

Pliki w języku Python

Pliki

- Python zawiera kilka wbudowanych funkcji oraz dodatkowych modułów, które pomagają w zarządzaniu plikami.
- Obiekty plików są w Pythonie głównym interfejsem do plików zewnętrznych znajdujących się na komputerze.
- Tworzymy obiekt pliku (file object) za pomocą funkcji *open*(). Po użyciu pliku powinno się go zamknąć za pomocą wbudowanej metody *close*(), aby zwolnić zasoby systemowe powiązane z otwartym plikiem. Otwarte pliki są automatycznie zamykane na zakończenie przetwarzania skryptu.
- Po zamknięciu pliku nie można go już przetwarzać.
- W Pythonie rozróżnia się pomiędzy plikami tekstowymi, a plikami binarnymi. Pliki tekstowe są podzielone na wiersze. Każdy wiersz kończy się znakiem końca wiersza (zwykle "\n", newline).
- Pliki inne niż tekstowe to pliki binarne, które powinny być przetwarzane przez aplikacje znające ich strukturę.

Pliki

Pliki używane są do odczytania konfiguracji i danych potrzebnych do działania programu, do wczytania danych użytkownika i do zapisania wyniku działania programu.

- Otwarcie pliku:

```
plik = open('ścieżka/do/pliku.txt')
```

- Odczytanie otwartego pliku:

```
print plik.read()
```

- Odczytanie otwartego pliku linijka po linijce, każda linijka na końcu zawiera domyślnie "Enter" :

```
for linia in plik:
    print linia
```

- Zamknięcie pliku

```
plik.close()
```

Pliki

- Odczytanie pliku:

```
text = open('nazwa_pliku').read()
print text
plik.close()
```

- Odczytanie pliku binarnego:

```
text = open('plik_binarny', 'rb').read()
print text
plik.close()
```

- Metoda linia.strip() usuwa zbędne znaki końca linii.
- Metoda tekst.split() dzieli tekst na wyrazy.
- Dla pojedynczego pliku można zastosować:

```
with open('plik.txt') as plik:
    for linia in plik:
        print linia.strip().split()
```

Pliki

- Plik można otworzyć w trybie do odczytu lub zapisu:

```
plik = open('plik.txt','r')
plik.close()
plik = open('plik.txt','w')
plik.write("tekst")
plik.close()
```

  Uwaga: otwarcie nieistniejącego pliku w trybie 'w' lub "a" spowoduje jego utworzenie.

- Zapis do otwartego pliku:

```
plik.write(str(55552522))
plik.write("\n")
plik.write("Tutaj masz jakis napis\t")
```

Pliki

- Najprostszy sposób pisania danych tekstowych i binarnych :

```
open('plik_do_zapisu', 'w').write("tekst")
open('plik_binarny_do_zapisu', 'wb').write("dane binarne")
```

- W przypadku gdy mamy listę (lub inną sekwencję) napisów, możemy zapisywać do pliku poprzez metodę writelines:

```
lista = ["aa ", "bb", "cc"]
plik = open('plik', 'w')
plik.writelines(lista)
plik.close
```

- W przypadku otwierania pliku w trybie "w" lub "wb" dane z tego pliku są usuwane.
- By dopisać dane należy użyć odpowiednio trybu "a" lub "ab ".

```
plik = open('plik.txt','a')      # a – append to tryb dopisywania
```

Pliki

```
plik = open(file_name, mode)
# file_name (string): nazwa pliku;
# mode (string):
# "r" (czytanie),
# "w" (pisanie; kasowanie poprzedniej zawartości; utworzy plik, gdy nie istniał),
# "a" (dopisywanie; poprzednia zawartość pozostaje),
# "r+" (czytanie i pisanie; poprzednia zawartość pozostaje),
# "w+" (czytanie i pisanie; kasowanie poprzedniej zawartości),
# "a+" (czytanie i pisanie; poprzednia zawartość pozostaje),
# "b" (dodatek do poprzednich, tryb binarny),
# "U" (uniwersalny translator nowych wierszy).
dir(plik)    # spis metod
plik.close()
```

Pliki

Przykłady

```
liczba = 41
x = 1.98
napis = " xyz"

plik.write("jeden\n")                # metoda zwraca None
plik.write(str(liczba) + "\n")       # trzeba dodawać '\n'
plik.write(str(x) + "\n")
plik.write(napis + "\n")

plik.write("%d\n" % liczba)          # zapis z procentem
plik.write("%f\n" % x)
plik.write("%s\n" % napis)
```

Prawa Murphy'ego

- Jeśli coś może się nie udać – **nie uda się** na pewno.
- Jeśli myślisz, że wszystko idzie dobrze – na pewno **o czymś nie wiesz**.
- Trudne problemy pozostawione same sobie staną się **jeszcze trudniejsze**.
- Jeśli udoskonalasz coś dostatecznie długo – **na pewno to zepsujesz**.
- Prawdopodobieństwo każdego zdarzenia jest **odwrotnie proporcjonalne** do stopnia, w jakim jest ono pożądane.
- To, czego szukasz, znajdziesz w **ostatnim z możliwych miejsc**.
- Nie ma rzeczy niemożliwych dla kogoś, kto **nie musi ich sam robić**.

Edsger Wybe Dijkstra

Edsger Wybe Dijkstra (1930-2002) – holenderski naukowiec, pionier informatyki.

- Informatyką zajmował się głównie od strony teoretycznej. Zajmował się różnymi jej aspektami, między innymi algorytmiką, językami programowania, formalną specyfikacją i weryfikacją.
- Był zafascynowany powstającym wówczas językiem Algol 60 i pracował nad jego pierwszym kompilatorem.
- W 1972 otrzymał Nagrodę Turinga za wkład w języki programowania.
- Do dziś jest pamiętany przede wszystkim dzięki algorytmowi znajdowania najkrótszych ścieżek w grafie (znanemu jako algorytm Dijkstry).
- Formułował opinie na temat zastąpienia instrukcji skoku innymi strukturami programistycznymi.

Donald Knuth

Donald Ervin Knuth – amerykański matematyk, informatyk, profesor na katedrze informatyki Uniwersytetu Stanforda.

- Jeden z pionierów informatyki, znany z dzieła *Sztuka programowania*, uznawanego za najbardziej zaawansowane opracowanie na temat analizy algorytmów.
- Jest autorem systemu składu drukarskiego TeX i języka opisu fontów METAFONT oraz twórcą i propagatorem techniki *literate programming*.
- Znacząco rozwinął algorytmikę, opracował teoretycznie wiele zagadnień z zakresu matematyki i informatyki.
- Ważniejsze nagrody: Nagroda Grace Murray Hopper (1971), Nagroda Turinga (1974), Narodowy Medal Nauki (1979), Medal Johna von Neumanna (1995), Nagroda Harveya (1995), Nagroda Kioto (1996).
- Zajmował się analizą starobabilońskich procedur obliczeniowych, uznając je za prawdziwe algorytmy.

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego

Fundusze Europejskie Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska Europejski Fundusz Społeczny