

Scalanie naturalne

Paweł Radosz, 180046

1. Wprowadzenie

Celem projektu było napisanie programu sortującego plik przy użyciu jednej z trzech dostępnych metod. Zdecydowałem się na implementację metody sortującej plik przez scalanie naturalne, a konkretniej na schemat 2+1 składający się z 3 taśm. Metoda bazuje na 2 operacjach występujących naprzemiennie. Są to operacje rozdzielania i sklejanie.

2. Szczegóły implementacji

a) Technologie

Całość programu została wykonana w języku Python (funkcjonalność i tworzenie wykresów). Do rysowania wykresów użyłem funkcji z biblioteki *matplotlib* i *numpy*. Uzyskane wyniki zostały wpisane do plików .csv, na podstawie których generowane zostały wykresy. Pliki z rekordami były trzymane w plikach .txt.

b) Właściwości (teoretyczne) rekordu:

Rekord składa się z 10 znaków. Jego wartość jest wyznaczana za pomocą liczby jedynek w zapisie binarnym kodów ASCII tych znaków.

c) Implementacja klasy rekordu

```
RECORD_SIZE = 10

class Record:
    def __init__(self, sign_list):
        self.sign_list = sign_list
        temp = 0
        for sign in sign_list:
            temp += sign.get_ones_amount()
        self._value = temp

    def get_value(self):
        return self._value
```


self.sign_list - lista znaków ASCII z których składa się rekord

self._value - wartość rekordu (suma jedynek w zapisie binarnym kodów ASCII)

d) Sposób przedstawiania rekordu w plikach

Rekordy w plikach są zapisywane i odczytywane bajtowo. Dlatego pojedyncza instrukcja w programie zapisuje/odczytuje jeden znak. Operacja zapisu/odczytu rekordu wymaga zapisu/odczytu każdego znaku z rekordu, dlatego wymaga ona dziesięciu wywołań danej instrukcji.


1) przykład pliku wejściowego przed sortowaniem (wyświetlanie każdego znaku w rekordzie)

 input — Notatnik

Plik Edycja Format Widok Pomoc

```
#i(v0!YL2Q:Li>Z%}LSVKB/ U#At)]*! qmUCNxp,FbJ^|-H_7`]pF#  
8Nw}19bF&^BEYf{FaNTgKj"BkGD&1|o]H`pahg!K+xfV5Hahz1T cWL
```


2) przykład pliku pomocniczego przedstawiającego wartości rekordów (przydatny do śledzenia działania programu)

 value — Notatnik

Plik Edycja Format Widok Pomoc

```
31 33 33 39 40 44 31 32 33 38 41 45 33 36 37 39 41 32 34 34 35 37 40 40 45 32 38 40
```

3) przykład pliku wejściowego po sortowaniu (wyświetlanie każdego znaku w rekordzie)

 input — Notatnik

Plik Edycja Format Widok Pomoc

```
AYP4Q@PE`ehR":IJ+@b@ PH ,SVE6Y:dg-B*@AJ`DbTRE\5!hP,hBp@w^A`d53A<#!a  
NmtWfu%j|60E*rm|G}Xn0heIL$?uh-kL:tAEb\v{^ gfBV,%V@f0tr?Tl.-_"qiear\
```

e) Funkcje klasy bufora umożliwiające dostęp do plików

Klasa bufora została zaimplementowana jako oddzielna warstwa logiki programu. Udostępnia ona funkcje, takie jak zapis i odczyt blokowy z pliku, głównej warstwie.

1) Odczyt

```
def write_page(self): |
    for _ in range(self._MAX_SIZE):
        record = []
        byte = self.read_from_file()
        if byte != b'':
            record.append(Sign(byte))
            for i in range(RECORD_SIZE-1):
                byte = self.read_from_file()
                if byte != b'':
                    record.append(Sign(byte))
                else:
                    raise ValueError('record not complete?')
            if len(record) != 0:
                self._buffer.append(Record(record))
                self._current_size += 1
    if self._current_size != 0:
        Buffer.read_counter += 1
```

2) Zapis

```
def save_page(self):
    if self._current_size != 0:
        Buffer.write_counter += 1
    if self._name == 'B':
        for l in range(self._current_size):
            for i in range(RECORD_SIZE):
                self._file_B.write(self._buffer[l].sign_list[i].get_sign())
    elif self._name == 'C':
        for l in range(self._current_size):
            for i in range(RECORD_SIZE):
                self._file_C.write(self._buffer[l].sign_list[i].get_sign())
    elif self._name == 'D':
        for l in range(self._current_size):
            for i in range(RECORD_SIZE):
                self._read_file.write(self._buffer[l].sign_list[i].get_sign())
            self._value_file.write(str(self._buffer[l].get_value()))
            self._value_file.write(' ')
    else:
        raise ValueError('this buffer doesnt save pages')
    self._current_size = 0
    self._buffer.clear()
```

3. Eksperyment

a) Opis

Eksperyment polegał na wyznaczeniu liczby faz sortowania i operacji dyskowych w zależności od liczby rekordów, a następnie porównaniu ich z wynikami teoretycznymi (uzyskanymi ze wzorów).

1. $\lceil \log_t r \rceil$ - teoretyczna liczba faz

2. $4N \lceil \log_2 r \rceil / b$ - teoretyczna liczba operacji

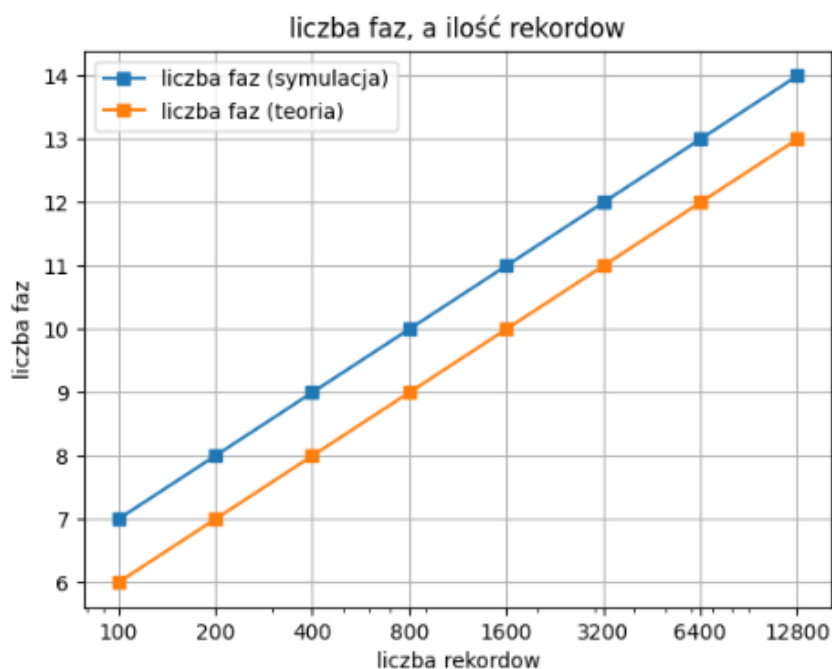
b) Tabela uzyskanych wyników dla (rozmiar bufora = 100, rozmiar rekordu = 10).

liczba rekordów	liczba faz (teoria)	liczba faz (symulacja)	operacje zapisu (symulacja)	operacje odczytu (symulacja)	suma operacji dyskowych (symulacja)	suma operacji dyskowych (teoria)
100	6	7	19	19	38	23
200	7	8	36	36	72	53
400	8	9	76	76	152	122
800	9	10	161	161	322	278
1600	10	11	346	346	692	615
3200	11	12	747	747	1494	1366
6400	12	13	1612	1612	3224	2980
12800	13	14	3468	3468	6936	6475

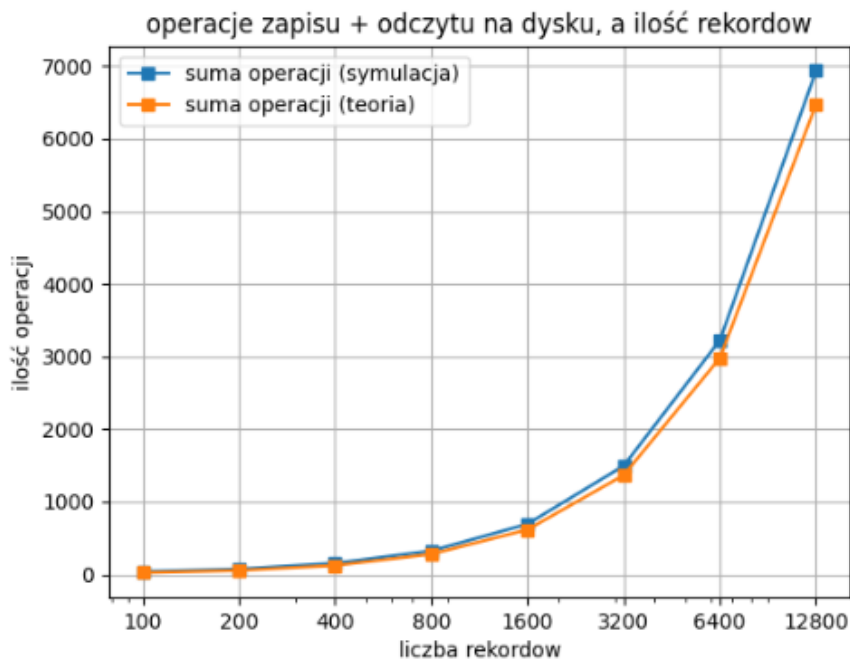
a) wykresy



Komentarz do wykresu 1: Wykres przedstawiający ilość operacji zapisu/odczytu w zależności od liczby rekordów. Dla przejrzystości wykresu została umieszczona tylko operacja zapisu, gdyż i tak różnica pomiędzy operacjami jest niezauważalna (1 operacja więcej dla odczytu).



komentarz do wykresu 2: Wykres przedstawiający dwie funkcje. Pierwsza z nich to liczba **rozpoczętych** faz sortowania (rozdzielanie + sklejanie) w zależności od liczby rekordów wynikająca z symulacji, zaś druga wynika z podstawienia do teoretycznego wzoru (avg case).



komentarz do wykresu 3: Wykres przedstawiający dwie funkcję. Pierwsza z nich to suma operacji (zapisu i odczytu rekordów z dysku) wynikająca z symulacji, zaś druga to teoretyczna (avg case) suma operacji dyskowych.

4. Wnioski

Główną rzeczą, która rzuca się w oczy jest różnica pomiędzy ilością faz w symulacji i w teorii. Wynosi ona 1 niezależnie od liczby rekordów. Jest to spowodowane inną implementacją warunku stopu faz sortowania. W przypadku symulacji jest ona ustawiona na moment, w którym druga taśma po fazie rozdzielania jest pusta. Oznacza to, że plik wejściowy został już posortowany (we wcześniejszej fazie), wynika to z faktu, że istnieje tylko jedna seria trafiająca prosto do pierwszej z taśm, dlatego druga taśma pozostaje pusta. Przez co niezależnie od ilości rekordów zawsze będziemy mieli 1 fazę więcej, co można zaobserwować na wykresie nr 2. Ściślej mówiąc faz sklejanie będzie tyle samo, lecz faz rozdzielania będzie o jedną więcej w symulacji. Przez co symulacja będzie wymagała także więcej operacji dyskowych zwłaszcza przy większej ilości rekordów, co można zauważyć na wykresie nr 3 (funkcja uzyskana z symulacji rośnie szybciej).