

Sprawozdanie z projektu 1 z kursu Struktury Baz Danych

Projekt wykonałem w pythonie, w środowisku google colab , w którym pracuję na co dzień. Środowisko to pozwala na realizację kodu poszczególnych komórek osobno, co bardzo ułatwia pisanie kodu, jak i weryfikację wykresów tuż pod kodem.

W swoim projekcie zastosowałem metodę scalania naturalnego o schemacie 2+1. Podczas wykonywania projektu sugerowałem się materiałami wykładowymi.

Przykład z wykładu:

Example: Natural merge, scheme 2+1.

File: ($N=8$)

44 55 | 12 42 94 | 18 | 06 67 (4 runs of different length)

The runs are distributed *alternately* on 2 tapes:

1st phase:

t1: 44 55 | 18

t2: 12 42 94 | 06 67

t3: 12 42 44 55 94 | 06 18 67 (2 runs)

2nd phase:

t1: 12 42 44 55 94

t2: 06 18 67

t3: 06 12 18 42 44 55 67 94 (1 run, file sorted)

Wytłumaczenie:

Na samym początku mamy 4 serie i 8 rekordów. Pierwsza seria zawiera rekordy 44 i 55. Druga 12, 42 i 94. Trzecia zawiera jeden rekord 18. Natomiast 4 seria zawiera rekordy 06 i 67.

Podczas pierwszej fazy na pierwszą taśmę dam 44 i 55, ponieważ 12 jest już mniejsze niż 55. Dlatego też kolejne rekordy dopóki kolejny rekord nie spotka rekordu mniejszego od poprzedniego daje na taśmę t2. Dlatego też daje na taśmę t2 wartości 12, 42 i 94, ponieważ napotkałem 18, która jest mniejsza niż 94. W związku z tym 18 daję na taśmę nr. 1. I tak dalej.

Gdy już rozdałem rekordy na taśmę t1 i t2. Pora połączyć je na taśmę t3. Stąd ta metoda 2+1. Czyli rozdzielam na 2 taśmy (t1 i t2) i następnie łączę w jedną taśmę (t3).

Następnie przechodzę do drugiej fazy i rozdzielam rekordy z taśmy z pierwszej fazy na taśmę t1 i t2 w drugiej fazie, analogicznie do tego co robiłem podczas 1 fazy.

Ostatecznie otrzymuję posortowane rekordy na taśmie t3 podczas drugiej fazy.

W moim projekcie zastosowałem analogiczny schemat.

Rozmiar rekordu w pliku (z wykładów "average record size in the file") ustaliłem na 4 bajty. 1 bajt reprezentuje długość rekordu, pozostałe 3 reprezentują odpowiednio bok, bok i kąt.

Reprezentuje to zmienna `Single_record_size`

Liczbę rekordów, które będą przechowywane w buforze w jednym momencie ustaliłem na 16. Reprezentuje to zmienna `Number_of_records`

Całkowity rozmiar bufora (z wykładów "the size of a disk block that is a read/write unit") wynosi: liczba rekordów * rozmiar rekordu w pliku, czyli $16 * 4 = 64$ bajty

Współczynnik blokowania wyznaczyłem jako całkowity rozmiar bufora / rozmiar rekordu w pliku, czyli $64/4 = 16$.

W ramach eksperymentu wartość współczynnika była zmieniana: 16, 32 i 256.

OPIS KLAS ZAIMPLEMENTOWANYCH W KODZIE

Klasy zaimplementowane w kodzie:

Record : Reprezentuje zbiór liczb, które mogą być przechowywane i manipulowane jako rekordy, zapewniając konwersję obiektu Record na listę liczb całkowitych, gdzie pierwszy element to liczba elementów i na odwrót. Umożliwia porównywanie rekordów.

BufferToRead : Odpowiada za odczytywanie rekordów z pliku binarnego. Kontroluje operacje odczytu. Zwraca rekordy z bufora lub ładuje nowe dane, gdy bufor się opróżni.

BufferToWrite: Obsługuje zapisywanie rekordów do pliku, przechowując je w buforze przed zapisaniem na dysk. Zapisuje dane do pliku, kiedy bufor jest pełny.

SeriesIterator: Klasa użyteczna do weryfikacji kiedy kończy się dana seria. Zwraca rekordy z bieżącej serii i ustala koniec serii na podstawie wartości kolejnych rekordów.

Funkcje:

show_tape : Użyteczna do zobrazowania jak wyglądają poszczególne taśmy, uwzględniając serie (w tym przypadku użyłem do rozróżnienia serii znaku "|"). Dodatkowo, podaje liczbę serii oraz rekordów, które zostały odczytane.

split : Funkcja rozdziela rekordy z pliku źródłowego na dwa pliki (taśmy), zapisując rekordy w kolejnych posortowanych sekwencjach. Wykorzystuje bufora do optymalizacji operacji odczytu i zapisu danych.

series_merge : Funkcja scala dwie posortowane serie rekordów z dwóch taśm do jednego bufora zapisu, porównując rekordy i zapisując je w odpowiedniej kolejności. Kontynuuje operację do momentu, aż wszystkie rekordy zostaną scalone i zapisane.

merge: Funkcja łączy posortowane serie z dwóch taśm w jedną większą sekwencję, zapisując wyniki na nową taśmę. Funkcja ta wielokrotnie wywołuje funkcję series_merge, aby scalić posortowane serie z obu taśm do bufora zapisu.

Różnice pomiędzy series_merge i merge:

series_merge: łączy dwie poszczególne serie

merge: łączy wszystkie serie z dwóch taśm w jedną posortowaną taśmę, wywołując wielokrotnie series_merge.

sort_and_info: Dopóki liczba serii jest różna do 1 (liczba serii równa 1 = rekordy są posortowane), czyli elementy nie są posortowane, wywołuję funkcję split i merge. Dodatkowo obliczam liczbę serii, odczytów i zapisów dyskowych.

W kodzie dodatkowo zaimplementowałem funkcję, która pozwala zobaczyć ile wynoszą pola poszczególnych rekordów, jak powinny zostać posortowane elementy wraz z ich polami oraz funkcję, która tworzy wykres wartości pola od posortowanych rekordów, co jest użyteczne do weryfikacji, czy rekordy, a co za tym idzie ich pola zostały poprawnie posortowane.

OPIS TYPU REKORDU I TO JAK GO WYKORZYSTAŁEM

Na początku jednak przedstawię typ rekordu, który definiuję w jaki sposób będę sortował rekordy.

24. Rekordy pliku: Równoległoboki - długości boków i jeden z kątów.
Uporządkowanie wg pola.

Jako rekord przyjąłem dwa boki równoległoboku i jeden z kątów.

Tworząc rekord wykorzystałem informację ze slajdów wykładowych:

"Typically, records have headers that show the structure of the record".

Jako header wykorzystałem długość pojedynczego rekordu.

Przykładowa wartość podczas debugowania:

```
[[4, 5, 95]]  
[3, 4, 5, 95]
```

Po wykonaniu poniższego kodu:

```
# wyciągam fragmenty listy z bufora tymczasowego, które odpowiadają pełnym rekordom danych  
for i in range(0, len(temporary_buffer), Single_record_size):  
    # Fragment listy odpowiadający pełnemu rekordowi danych  
    full_record = temporary_list_buffer[i:i + Single_record_size]  
  
    # tworze obiekt Record na podstawie wyciągniętej listy liczb całkowitych  
    self.buffer.append(Record.load_integers(full_record))
```

Wyjaśnienie:

Single_record_size jest ustawiony na 4, ponieważ mój typ rekordu to bok, bok oraz kąt czyli trzy wartości i dodatkowo długość rekordu, czyli sumarycznie rozmiar jest 4. Zatem przechodzę w pętli od 0 do określonej liczby bajtów z pliku, z krokiem co 4.

Następnie wycinam z temporary_list_buffer wartości od 0 do +4, następnie od 4 do 8 itp..

Dzięki temu otrzymam poszczególne rekordy wraz z ich długością.

Poszczególne wyniki przypisuje do zmiennej "full_record", które potem dodaje do bufora jako załadowane liczby całkowite w następujący sposób:

```
def load_integers(x):  
    return Record(x[1: x[0] + 1])
```

W powyższym kodzie ładuje elementy bez 1 wartości aż do elementu, który wskazuje wartość pierwszej wartości rekordu, która oznacza w tym przypadku długość elementów w rekordzie (wykorzystałem informację wykładową, o której wspominałem wcześniej).

W kodzie zaimplementowałem dwa bufory. Bufor odczytu i bufor zapisu. Warto zwrócić uwagę, że w buforze zapisu dla każdy rekord konwertuję na listę liczb całkowitych, ponieważ liczby całkowite są łatwiejsze do zapisania w postaci binarnej.

Poniżej przedstawiam konwersję:

Dodatkowo dopowiem, że fragment : + [0] *.... jest dodatkowy i jest to forma zabezpieczenia. Rozmiar rekordu jest stały, więc nie jest to potrzebne, ale zabezpiecza przypadek, gdyby nagle pojawił się przypadek rekordu różnej długości.

```
# dla każdego rekordu wywołuje metodę save_integers, która konwertuje rekord na listę liczb całkowitych  
for record in self.buffer[0:self.write_position]:  
    integers_to_write += record.save_integers()
```

Wykorzystując wcześniej zdefiniowaną funkcję:

```
def save_integers(self):
    #pdb.set_trace()
    return [len(self.elements)] + self.elements + [0] * (Single_record_size - len(self.elements) - 1)
```

Wy tłumaczenie:

Gdy `len(self.elements)` jest równy 3 i `Single_record_size` równy 4, zwrócę jako zerowy element długość czyli `len(self.elements)` i następnie poszczególne wartości w rekordzie do tego w ramach zabezpieczenia, żeby nie zapisać rekordu innej długości niż zostało założone dodaje zera o długości (poszczególnego rozmiaru rekordu odjąć długość elementów, w tym wypadku bok, bok oraz kąt i jeszcze jeden jako header). Jeśli elementy (`self.elements`) będą zgodne z założeniem nie będzie potrzeby dodania zer.

Następnie otwieram plik w trybie `append binary`, co pozwala na dodawanie danych na końcu pliku. Informację o tym również znalazłem w materiałach wykładowych: "A new record is appended to the end of the file". Dodałem również brak buforowania, czyli zapis będzie natychmiastowy. Na sam koniec zwiększam licznik zapisów na dysk.

Kod:

```
with open(self.path, "ab", buffering = 0) as f:
    f.write(bytearray(integers_to_write))
    f.close()
    self.disk_writes_count += 1
```

W buforze odczytu zaimplementowałem funkcję `see_next`, która podgląda kolejny rekord bez przesuwania wskaźnika, co przyda mi się podczas weryfikowania, czy zakończyła się dana seria, czy nie:

```
# podgladam jaki bedzie nastepny rekord
next_record = self.read_buffer.see_next()

# tu jesli poglądniety rekord jest mniejszy od poprzedniego to oznacza ze zakonczyła sie pewna seria
if next_record is not None and next_record < self.current_record:
    # i wtedy nalezy ustawic flage konca serii na true
    self.end_of_series = True
```

Kod:

```
def see_next(self):
    return None if self.read_position == self.loaded_size else self.buffer[self.read_position]
```

Wy tłumaczenie:

Jeśli pozycja odczytu jest taka jak załadowana to oznacza, że nie ma już, więcej dlatego zwracam None. W innym przypadku zwracam wartość z bufora o pozycji odczytu.

ODCZYT I ZAPIS

Odczyt:

Aby wczytywać rekordów z pliku binarnego używam poniższego kodu, gdzie "rb" - oznacza read binary :

```
with open(self.path, "rb", buffering =0) as f
```

Zapis:

"ab" - append binary - nowe dane będą dodane na końcu pliku w trybie binarnym, buffering = 0 oznacza natychmiastowy zapis danych na dysk

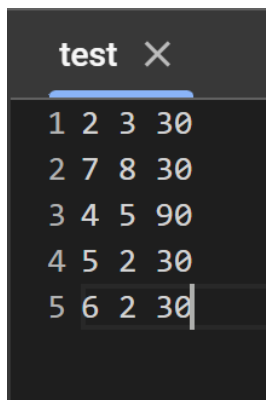
bytearray - przekształci dane na tablicę bajtów. Dzięki temu dane mogą być przechowywane i przesyłane w postaci binarnej.


```
with open(self.path, "ab", buffering = 0) as f:  
    f.write(bytearray(integers_to_write))
```

FORMAT PLIKU TEKSTOWEGO

Format pliku tekstowego wygląda w następujący sposób:

Każdy wiersz reprezentuje pojedynczy rekord, który posiada 3 wartości: bok, bok oraz kąt.



1	2	3	30
2	7	8	30
3	4	5	90
4	5	2	30
5	6	2	30

OPIS PROCESU SORTOWANIA WRAZ Z PRZEDSTAWIENIEM SPOSOBU PREZENTOWANIA WYNIKÓW

Proces sortowania powyższych danych z pliku:

Opiszę proces sortowania na tym przykładzie. Dalsze procesy w sprawozdaniu wykonywane są w analogiczny sposób.

Na samym początku wyświetlam taśmę przed sortowaniem.

Rozpaczynam pierwszą fazę i dzielam rekordy na 2 taśmy. Pole rekordu [2, 3, 20] wynosi 3. Zapisuje je na taśmę nr. 1. Pole rekordu [7,8,30] wynosi 28, zatem

zapiszę ten rekord na taśmie pierwszej ze względu na to, że 28 jest większe od 3. Kolejnym rekordem jest [4,5,90], którego pole wynosi 20. Mamy przypadek, gdy nie mogę zapisać już tego rekordu na taśmę nr.1, ponieważ 20 jest mniejsze niż 28. W takim wypadku muszę zapisać ten rekord na taśmę nr.2. Kolejnym rekordem jest [5, 2, 30], którego pole wynosi 5. 5 jest mniejsze niż 20, więc nie mogę zapisać tego rekordu na taśmę nr. 2, dlatego zapisuję go na taśmie nr.1. W tym momencie muszę również pamiętać (zostało to również uwzględnione tutaj za pomocą pionowej kreski, tak jak na wykładzie), o tym, że 5 jest mniejsze niż wynik z [7,8,30], czyli 28. Na samym końcu jest [6,2,30], którego pole jest równe 6. 6 jest większe od 5 dlatego spokojnie mogę to zapisać na taśmie nr.1. Analogicznie działa to dalej.

Podczas każdego wyświetlania taśmy wypisywana jest jej liczba serii i ilość rekordów.

Dodatkowo uwzględniłem liczbę faz. Liczba faz jest wypisywana nad taśmą nr.3, a po skończeniu sortowania wypisywane są informacje na temat sumarycznej liczby faz, odczytów i zapisów dyskowych.

```
▶ Sorting tape tapes/t1
⇌ Tape before sorting:
[2, 3, 30] [7, 8, 30] | [4, 5, 90] | [5, 2, 30] [6, 2, 30]

Series count: 3
Records count: 5

Tape 1:
[2, 3, 30] [7, 8, 30] | [5, 2, 30] [6, 2, 30]

Series count: 2
Records count: 4

Tape 2 :
[4, 5, 90]

Series count: 1
Records count: 1

Phase: 1
Tape 3:
[2, 3, 30] [4, 5, 90] [7, 8, 30] | [5, 2, 30] [6, 2, 30]

Series count: 2
Records count: 5

Tape 1:
[2, 3, 30] [4, 5, 90] [7, 8, 30]

Series count: 1
Records count: 3

Tape 2 :
[5, 2, 30] [6, 2, 30]

Series count: 1
Records count: 2

Phase: 2
```

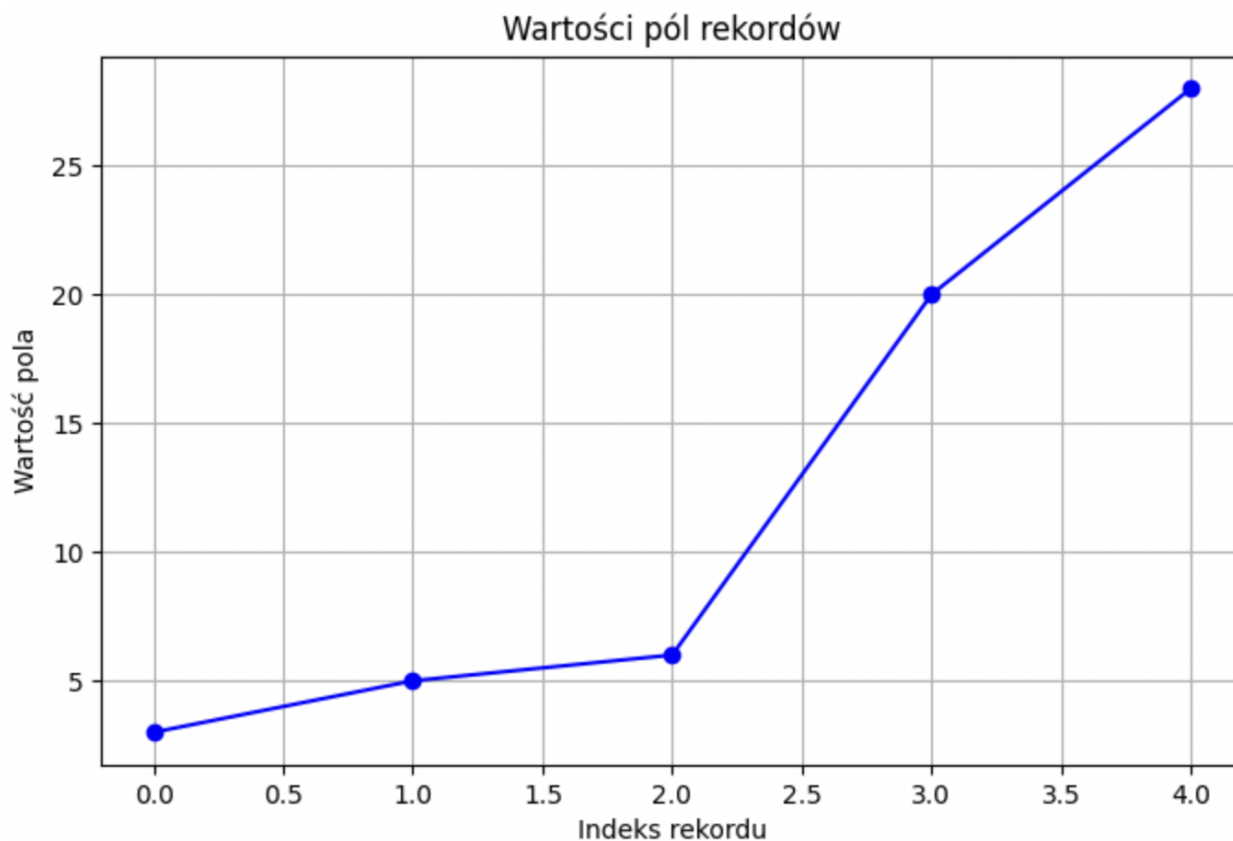
```
Phase: 2
Tape 3:
[2, 3, 30] [5, 2, 30] [6, 2, 30] [4, 5, 90] [7, 8, 30]

Series count: 1
Records count: 5

Tape after sorting:
[2, 3, 30] [5, 2, 30] [6, 2, 30] [4, 5, 90] [7, 8, 30]

Series count: 1
Records count: 5
Metadata:
Phase counter: 2
Reads counter: 6
Writes counter: 6
```

Weryfikacja poprawnego posortowania na podstawie wykresu pola wraz z kolejnymi posortowanymi już rekordami. To zostało zaimplementowane dodatkowo.



Na rysunku widać, że kolejne posortowane już rekordy mają coraz to większą wartość pola co sugeruje, że proces sortowania przebiegł prawidłowo.

GENEROWANIE REKORDÓW

Zgodnie z wymaganiami do zadania zaimplementowałem funkcję, która sama generuje rekordy. Oto wygenerowane rekordy:

```
Podaj liczbę wierszy do wygenerowania: 8
Added 8 records to tape
```

Showing what was generated

[16] all_rows

```
⇒ [[9, 4, 37],  
   [8, 2, 49],  
   [3, 1, 25],  
   [10, 4, 9],  
   [1, 4, 32],  
   [3, 4, 49],  
   [3, 10, 23],  
   [9, 3, 75]]
```

Dodatkowo zaimplementowałem funkcję, która oblicza pola poszczególnych rekordów:

```
⇒ Result for row: 1: 21.6653  
   Result for row: 2: 12.0754  
   Result for row: 3: 1.2679  
   Result for row: 4: 6.2574  
   Result for row: 5: 2.1197  
   Result for row: 6: 9.0565  
   Result for row: 7: 11.7219  
   Result for row: 8: 26.0800
```

Zaimplementowałem również taką, która sortuje pola, żeby łatwiej można było sprawdzić czy wynik się zgadza (dla mniejszej ilości rekordów oczywiście :)).

```
↔ Posortowane wyniki:  
Wynik: 1.2679, Wiersz: [3, 1, 25]  
Wynik: 2.1197, Wiersz: [1, 4, 32]  
Wynik: 6.2574, Wiersz: [10, 4, 9]  
Wynik: 9.0565, Wiersz: [3, 4, 49]  
Wynik: 11.7219, Wiersz: [3, 10, 23]  
Wynik: 12.0754, Wiersz: [8, 2, 49]  
Wynik: 21.6653, Wiersz: [9, 4, 37]  
Wynik: 26.0800, Wiersz: [9, 3, 75]
```

Rezultat:

```

Sorting tape tapes/t1
Tape before sorting:
[9, 4, 37] | [8, 2, 49] | [3, 1, 25] [10, 4, 9] | [1, 4, 32] [3, 4, 49] [3, 10, 23] [9, 3, 75]

Series count: 4
Records count: 8

Tape 1:
[9, 4, 37] | [3, 1, 25] [10, 4, 9]

Series count: 2
Records count: 3

Tape 2 :
[8, 2, 49] | [1, 4, 32] [3, 4, 49] [3, 10, 23] [9, 3, 75]

Series count: 2
Records count: 5

Phase: 1
Tape 3:
[8, 2, 49] [9, 4, 37] | [3, 1, 25] [1, 4, 32] [10, 4, 9] [3, 4, 49] [3, 10, 23] [9, 3, 75]

Series count: 2
Records count: 8

Tape 1:
[8, 2, 49] [9, 4, 37]

Series count: 1
Records count: 2

Tape 2 :
[3, 1, 25] [1, 4, 32] [10, 4, 9] [3, 4, 49] [3, 10, 23] [9, 3, 75]

Series count: 1
Records count: 6

Phase: 2

```

```

Phase: 2
Tape 3:
[3, 1, 25] [1, 4, 32] [10, 4, 9] [3, 4, 49] [3, 10, 23] [8, 2, 49] [9, 4, 37] [9, 3, 75]

Series count: 1
Records count: 8

Tape after sorting:
[3, 1, 25] [1, 4, 32] [10, 4, 9] [3, 4, 49] [3, 10, 23] [8, 2, 49] [9, 4, 37] [9, 3, 75]

Series count: 1
Records count: 8
Metadata:
Phase counter: 2
Reads counter: 6
Writes counter: 6

```


Dzięki wcześniej zaimplementowanej funkcji sortującej wyniki jestem w stanie porównać je z wynikiem taśmy po posortowaniu.

WCZYTYWANIE Z KŁAWIATURY

Dodałem również możliwość wpisania rekordów z klawiatury:

Przykładowy proces wpisywania poszczególnych wartości, gdzie najpierw należy podać wartość pierwszego boku, następnie drugiego boku i na samym końcu kąt. Tak należy zrobić dla każdego rekordu:

```
... Podaj liczbę rekordów do wprowadzenia: 6
Rekord 1:
  Wprowadź wartości równoległoboka (bok, bok, kąt) 1: 2
  Wprowadź wartości równoległoboka (bok, bok, kąt) 2: 3
  Wprowadź wartości równoległoboka (bok, bok, kąt) 3: 30
Rekord 2:
  Wprowadź wartości równoległoboka (bok, bok, kąt) 1: 6
  Wprowadź wartości równoległoboka (bok, bok, kąt) 2: 4
  Wprowadź wartości równoległoboka (bok, bok, kąt) 3: 90
Rekord 3:
  Wprowadź wartości równoległoboka (bok, bok, kąt) 1: 4
  Wprowadź wartości równoległoboka (bok, bok, kąt) 2: 7
  Wprowadź wartości równoległoboka (bok, bok, kąt) 3: 12
Rekord 4:
  Wprowadź wartości równoległoboka (bok, bok, kąt) 1: 9
  Wprowadź wartości równoległoboka (bok, bok, kąt) 2: 13
```

Przykładowe dane wprowadzone z klawiatury.

```
Podaj liczbę rekordów do wprowadzenia: 6
Rekord 1:
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 1: 2
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 2: 3
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 3: 30
Rekord 2:
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 1: 6
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 2: 4
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 3: 90
Rekord 3:
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 1: 4
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 2: 7
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 3: 12
Rekord 4:
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 1: 9
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 2: 13
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 3: 76
Rekord 5:
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 1: 5
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 2: 8
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 3: 43
Rekord 6:
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 1: 7
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 2: 9
  Wprowadź wartosci rownlolegloboka (bok, bok, kat) 3: 12
Added 6 records to tape
```

Proces sortowania tych danych przedstawiam poniżej:

```
Sorting tape tapes/t1
Tape before sorting:
[2, 3, 30] [6, 4, 90] | [4, 7, 12] [9, 13, 76] | [5, 8, 43] | [7, 9, 12]

Series count: 4
Records count: 6

Tape 1:
[2, 3, 30] [6, 4, 90] [5, 8, 43]

Series count: 1
Records count: 3

Tape 2 :
[4, 7, 12] [9, 13, 76] | [7, 9, 12]

Series count: 2
Records count: 3

Phase: 1
Tape 3:
[2, 3, 30] [4, 7, 12] [6, 4, 90] [5, 8, 43] [9, 13, 76] | [7, 9, 12]

Series count: 2
Records count: 6

Tape 1:
[2, 3, 30] [4, 7, 12] [6, 4, 90] [5, 8, 43] [9, 13, 76]

Series count: 1
Records count: 5

Tape 2 :
[7, 9, 12]

Series count: 1
Records count: 1
```

```
Phase: 2
Tape 3:
[2, 3, 30] [4, 7, 12] [7, 9, 12] [6, 4, 90] [5, 8, 43] [9, 13, 76]

Series count: 1
Records count: 6

Tape after sorting:
[2, 3, 30] [4, 7, 12] [7, 9, 12] [6, 4, 90] [5, 8, 43] [9, 13, 76]

Series count: 1
Records count: 6
Metadata:
Phase counter: 2
Reads counter: 6
Writes counter: 6
```

To co przedstawię niżej zostało zaimplementowane dodatkowo dla szerszego zrozumienia i weryfikacji procesu sortowania !!!

Używając wcześniej zdefiniowanych funkcji sprawdzmy rekordy i już posortowane rekordy i porównajmy z tym co otrzymałem.

```
✓ 0s all_rows
[[2, 3, 30], [6, 4, 90], [4, 7, 12], [9, 13, 76], [5, 8, 43], [7, 9, 12]]
```

Wartości pól:

```
Result for row: 1: 3.0000
Result for row: 2: 24.0000
Result for row: 3: 5.8215
Result for row: 4: 113.5246
Result for row: 5: 27.2799
Result for row: 6: 13.0984
```

Wynik poglądowy sortowanie według pola:

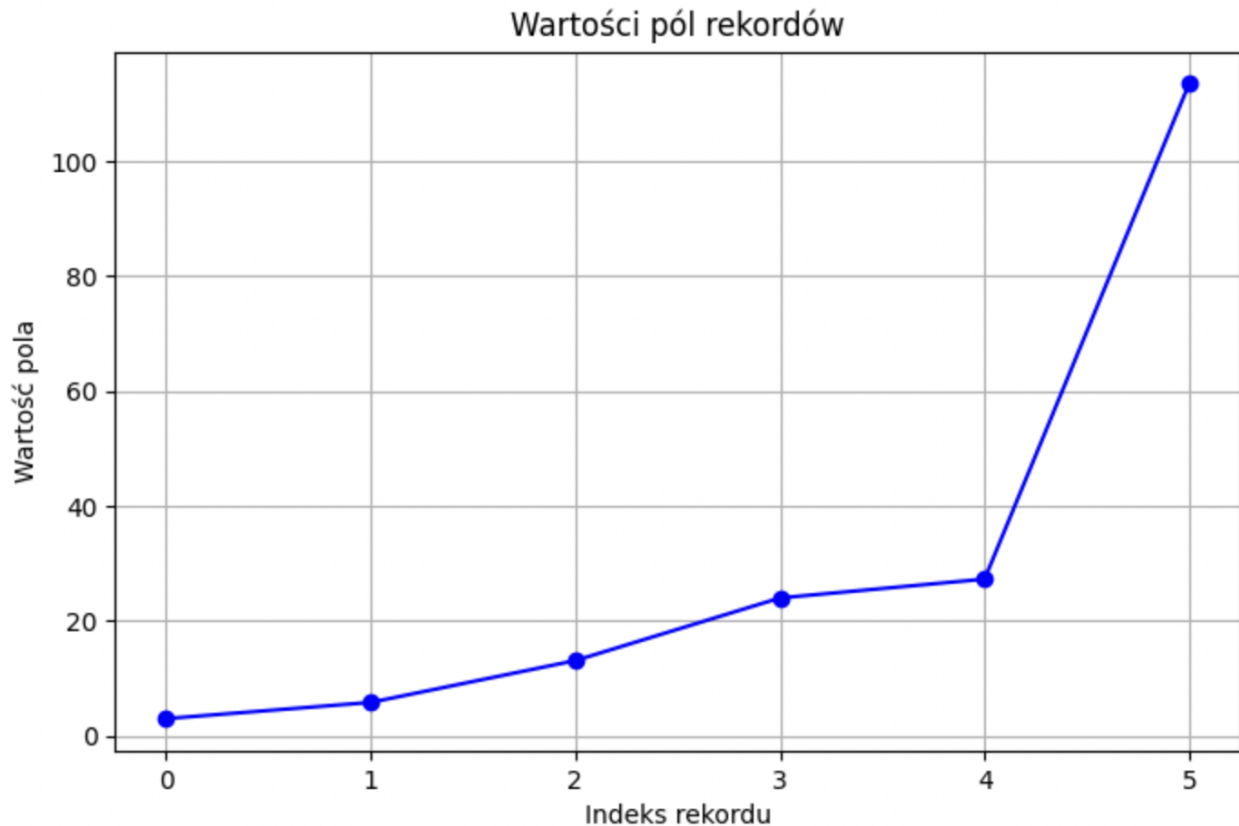
```
Posortowane wyniki:
Wynik: 3.0000, Wiersz: [2, 3, 30]
Wynik: 5.8215, Wiersz: [4, 7, 12]
Wynik: 13.0984, Wiersz: [7, 9, 12]
Wynik: 24.0000, Wiersz: [6, 4, 90]
Wynik: 27.2799, Wiersz: [5, 8, 43]
Wynik: 113.5246, Wiersz: [9, 13, 76]
```

Widać, że uzyskany wynik jest poprawny.

Dodatkowo zdefiniowałem funkcję, która na sam koniec służy do jeszcze wyraźniejszego zobrazowania tego w jaki sposób na wykresie wartości pól rosną, co oznacza, że zostały dobrze posortowane.

W przypadku, gdyby wartości już posortowanych pól, w jakimkolwiek miejscu na wykresie nie rosłyby wraz z każdym kolejnym już posortowanym rekordzie byłby to błąd.

Wykres dla wartości wpisanych z klawiatury:



Przedstawiłem już randomowy generowanie rekordów, wpisywanie ich z klawiatury, jak i możliwość odczytu rekordów z pliku.

EKSPERYMENT I JEGO WYNIKI

Teraz pora na coś większego...

Wygeneruję randomowe wartości dla 100 rekordów, 500 itd...

```
num_rows_list = [100, 500, 1000, 5000, 10000]  
generate_and_sort_records(tape, num_rows_list)
```

Dzięki temu będę mógł w szerszy sposób spojrzeć na poszczególne wyniki i porównać je z pozostałymi. Utworzyłem również wykres liczby faz w zależności od liczby rekordów oraz wykres liczby operacji dyskowych w zależności od liczby rekordów.

Przed utworzeniem wykresów wyznaczyłem teoretyczną liczbę faz i liczbę operacji dyskowych, wykorzystując blocking factor omówiony na wykładzie:

Blocking factor

$$b = B / R$$

where:

B – the size of a disk block that is a read/write unit; usually from a few to a dozen kB (such a block is also called a *disk page*);

R – average record size in the file.

Liczba faz i ilość odczytów i zapisów dyskowych przedstawiona jest poniżej:

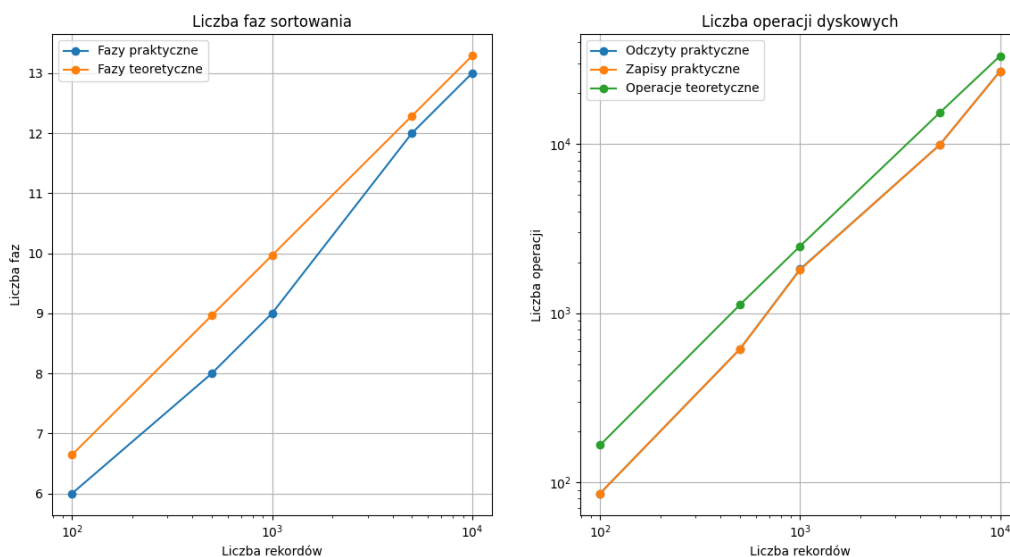
In the worst case (file ordered inversely), the number of phases is $\lceil \log_2 N \rceil$, because the initial number of runs is N . The, the number of disk reads/writes (scheme 2+1) is:

$$4N \lceil \log_2 N \rceil / b$$

Liczba faz została zapisana jako zmienna "phases ", a liczba operacji dyskowych jako operations, wykorzystując wzory wspomniane wyżej.

```
def theoretical_phases(num_rows):  
    # teoretyczna liczba faz  
    phases = math.log(num_rows, 2)  
    #phases = math.ceil(math.log(num_rows, 2))  
    #phases = math.log2(num_rows)  
    # teoretyczna liczba operacji (odczytów i zapisów)  
    operations = ((4 * num_rows) * phases) / b  
  
    return phases, operations
```

Wykres zależności liczby faz od liczby rekordów oraz liczby operacji dyskowych od liczby rekordów dla współczynnika blokowania równego 16:

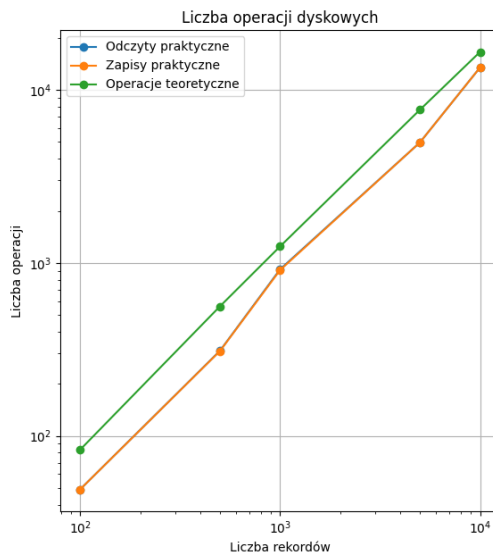
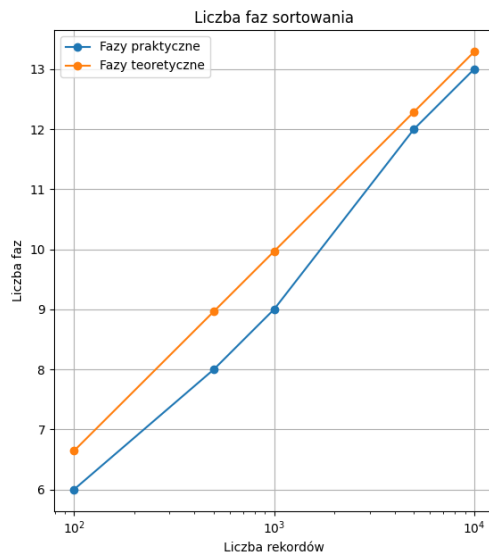


Widać, że wyniki są do siebie bardzo zbliżone, jednak występują pewne różnice.

Liczba faz w odczytach praktycznych jest zawsze niższa, co oznacza, że mój algorytm radzi sobie bardzo dobrze. Podobnie sprawa ma się w przypadku liczby operacji dyskowych, gdzie odczyty i zapisy praktyczne, są również mniejsze.

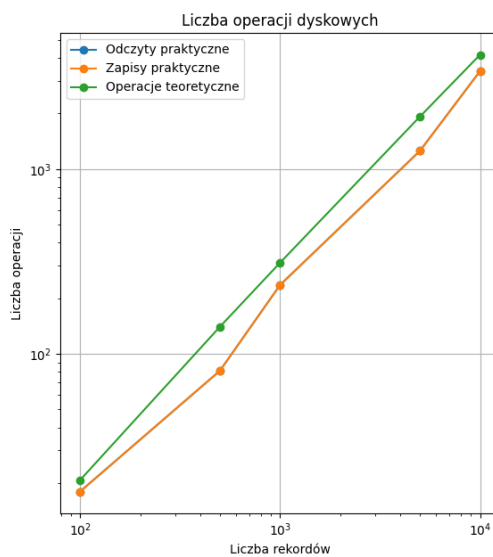
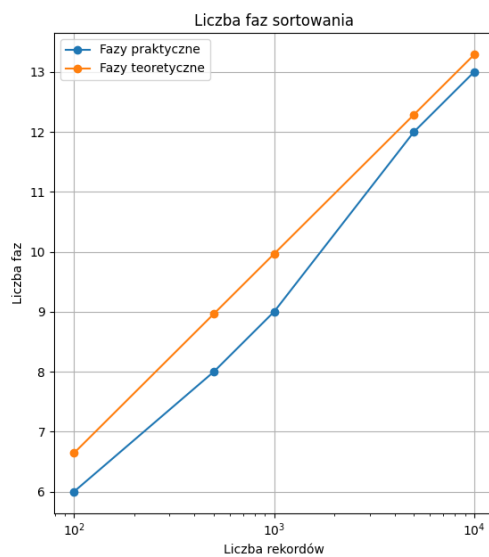
Na różnice w wynikach praktycznych i teoretycznych może wpływać np. sklejanie się serii.

Tutaj dodam jeszcze rezultat dla współczynnika blokowania równego 32:



Można zauważyć, że zwiększając współczynnik blokowania liczba operacji dyskowych maleje.

Jeszcze bardziej widać to przy współczynniku blokowania równym 256:



Wniosek z tego prosty: zwiększając liczbę rekordów, które będą przechowywane w buforze w jednym momencie zmniejszamy liczbę operacji dyskowych.

Dodatkowa implementacja wykresu wartości pól posortowanych rekordów.

Poniżej przedstawiam wykres wartości pól w zależności od posortowanych już rekordów. Widać, że wartości pól wraz z posortowanymi rekordami rosną, co jest poprawnym wynikiem.

