

Pytania – egzamin inżynierski

Lidia J. Opuchlik

Polsko – Japońska Akademia Technk Komputerowych

s16478@pjwstk.edu.pl

6 lutego 2021

Agenda

- 1 Pytanie nr 8
 - Twierdzenie Bayesa.
- 2 Pytanie nr 22
 - Najważniejsze algorytmy wyszukiwania i sortowania.
 - Wyszukiwanie
 - Sortowanie
- 3 Pytanie nr 34
 - Przetwarzanie strumieniowe (środki pakietu `java.util.stream`).
- 4 Źródła
 - Najważniejsze źródła i odnośniki.

Twierdzenie Bayesa

Twierdzenie Bayesa

Twierdzenie Bayesa

- 1 Thomas Bayes, presbiteriański pastor, statystyk, filozof w 18-wiecznej Anglii.
- 2 Mówi o **prawdopodobieństwie warunkowym**. Pozwala określić prawdopodobieństwo zajścia jakiegoś zdarzenia, o ile zaszło jakieś inne zdarzenie.

Treść (dla dwóch zdarzeń)

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}, \quad \text{oraz } P(B) > 0 \text{ i } P(A) \geq 0$$

gdzie:

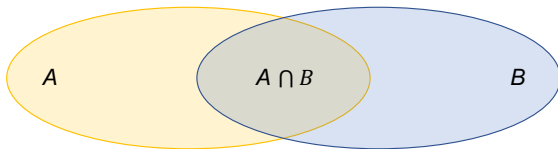
$P(A|B)$ – prawdopodobieństwo zajścia zdarzenia A, gdy zachodzi zdarzenie B (posterior probability),

$P(B|A)$ – prawdopodobieństwo zajścia zdarzenia B, gdy zachodzi zdarzenie A (likelihood),

$P(A)$ – prawdopodobieństwo zajścia zdarzenia A (prior probability),

$P(B)$ – prawdopodobieństwo zajścia zdarzenia B (evidence).

Dowód (dla dwóch zdarzeń*)



Rysunek: Iloczyn zbiorów A i B.

Z użyciem prawdopodobieństwa łączonego (joint probability):

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \Leftrightarrow P(A \cap B) = P(A|B) \cdot P(B)$$

$$P(B|A) = \frac{P(A \cap B)}{P(A)} \Leftrightarrow P(A \cap B) = P(B|A) \cdot P(A)$$

Dowód c. d.

$$P(A \cap B) = P(A|B) \cdot P(B)$$

$$P(A \cap B) = P(B|A) \cdot P(A)$$

$$P(A|B) \cdot P(B) = P(B|A) \cdot P(A)$$



$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

*dowód dla większej liczby zdarzeń jest dużo bardziej skomplikowany

Trywialny przykład

Treść zadania:

Obliczyć prawdopodobieństwo, że osoba jest chora na grypę, gdy występuje u niej gorączka, wiedząc, że: zdarzenie A – osoba jest chora na grypę, zdarzenie B – u osoby występuje gorączka. W zadaniu dane są następujące prawdopodobieństwa: $P(B|A) = 0.7$, $P(A) = 0.1$, $P(B) = 0.2$.

Rozwiązanie:

Korzystając ze wzoru Bayesa obliczamy $P(\text{grypa}|\text{goraczka})$:

$$P(\text{grypa}|\text{goraczka}) = \frac{P(\text{goraczka}|\text{grypa}) \cdot P(\text{grypa})}{P(\text{goraczka})}$$

$$P(\text{grypa}|\text{goraczka}) = \frac{0.7 \cdot 0.1}{0.2} = \mathbf{0.35}$$

Nieco bardziej złożony przykład

Klasyczny przykład zastosowania twierdzenia Bayesa, to przypadek testowania na dosyć rzadką chorobę. Szukamy jakie jest prawdopodobieństwo, że osoba jest chora, jeżeli wyszedł jej pozytywny wynik testu. Założmy, że recyzja testu to 95 %, choruje 1/100 osób.

Intuicyjnie nasuwa się, że prawdopodobieństwo powinno być dosyć wysokie. Jest to jednak błędne myślenie.

Analiza:

$$P(\text{choroba}|\text{pozytywny}) = \frac{P(\text{pozytywny}|\text{choroba}) \cdot P(\text{choroba})}{P(\text{pozytywny})}$$

P(choroba|pozytywny) – prawdopodobieństwo, że osoba ma chorobę bazując na tym, że test wyszedł pozytywny.

P(pozytywny|choroba) – prawdopodobieństwo, że test wyszedł pozytywnie jeżeli rzeczywiście jest zdiagnozowana choroba. Mówi ono o czułości testu — true positive (95 %) oraz specyficzności testu — true negative (95 %).

P(choroba) – prawdopodobieństwo wystąpienia choroby u danej osoby. $P = 0.01$.

P(pozytywny) – prawdopodobieństwo, że testowana osoba będzie miała wynik pozytywny. Może być tak, że test da prawidłowy wynik (true positive) lub zakłamany wynik (false positive).

Nieco bardziej złożony przykład c. d.

$$P(\text{choroba}|\text{pozytywny}) = \frac{P(\text{pozytywny}|\text{choroba}) \cdot P(\text{choroba})}{P(\text{pozytywny})}$$

Rozwiązanie:

$$\begin{aligned} P(\text{choroba}|\text{pozytywny}) &= \frac{0.95 \cdot 0.01}{(0.01 \cdot 0.95) + (0.99 \cdot 0.05)} = \\ &= \frac{0.0095}{0.0095 + 0.0495} = \frac{0.0095}{0.059} = 0.1610169 = \mathbf{0.16} \end{aligned}$$

Jedynie 16 % osób z wynikiem pozytywnym ma chorobę, a więc otrzymując wynik testu dużo bardziej prawdopodobne jest to, że test wyszedł fałszywie pozytywny, niż to, że osoba jest chora – w końcu zachorowalność to tylko 1 %.

W czym przydatne jest twierdzenie Bayesa?

- ① stosowane do lepszego zrozumienia danych
- ② ułatwia zrozumienie i interpretację wyników testów A/B
- ③ wykorzystywane przez algorytmy machine learningu
 - Naive Bayes,
 - Optymalny Klasyfikator Bayesowski,
 - Bayesowskie Sieci Przekonań (Bayesian Belief Networks)

Wyszukiwanie i sortowanie

Najważniejsze algorytmy wyszukiwania i sortowania

Wyszukiwanie i sortowanie

Wyszukiwanie

- 1 liniowe
- 2 binarne
- 3 skok co k-ty element

Sortowanie

- 1 przez wybieranie (selection sort)
- 2 bąbelkowe (bubble sort)
- 3 przez scalanie (merge sort)
- 4 szybkie (quick sort)
- 5 przez wstawianie (insertion sort)
- 6 zliczeniowe (counting sort)

Wyszukiwanie

Wyszukiwanie

Wyszukiwanie – na czym polega?

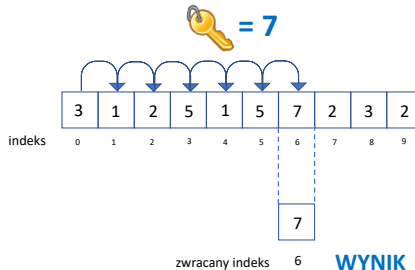
Polega na odszukaniu elementu o danej wartości (klucza) w nieposortowanej lub posortowanej strukturze danych (tablicy lub liście) i zwrócenie indeksu, pod którym znaleziony element się znajduje*. Jeżeli element nie zostanie znaleziony, wyświetlany jest stosowny komunikat (często wartość -1).

*jeżeli jest kilka takich samych elementów w ciągu, to zwracany jest indeks pierwszego napotkanego

** klucz i indeksy są liczbami całkowitymi

Wyszukiwanie liniowe

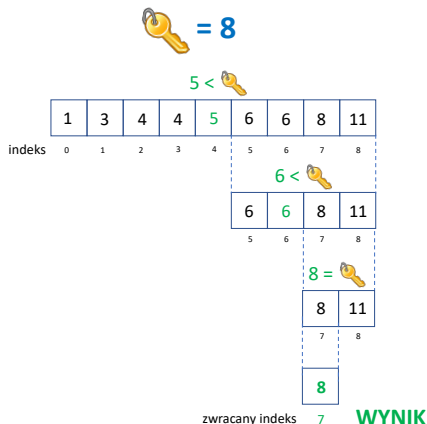
- struktura danych **nie musi** być posortowana
- pesymistyczna złożoność $O(n)$



Rysunek: Wyszukiwanie liniowe.

Wyszukiwanie binarne

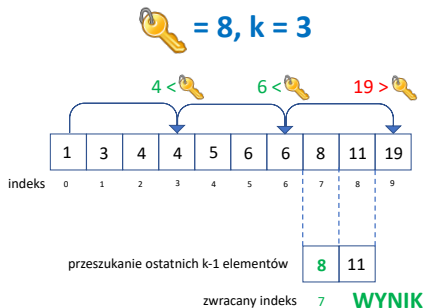
- struktura danych **musi** być posortowana
- wykorzystywany jest algorytm "dziel i rządź"
- pesymistyczna złożoność $O(\log n)$



Rysunek: Wyszukiwanie binarne.

Wyszukiwanie poprzez skok co k-ty element

- struktura danych **musi** być posortowana
- algorytm jest k razy szybszy niż algorytm liniowy



Rysunek: Wyszukiwanie co k-ty element.

Sortowanie

Sortowanie

Sortowanie

Polega na ułożeniu elementów jakiejś nieuporządkowanej struktury danych w określonym porządku, np. rosnąco, malejąco, niemalejąco, ale też np. alfabetycznie.

Algorytmy sortowania w zależności od implementacji charakteryzują się różną wydajnością sortowania.

- ograniczę się do sortowania liczb całkowitych
- przedstawione opisy algorytmów będą dotyczyć sortowania niemalejącego (od najmniejszego do największego elementu)

Sortowanie – kilka przydatnych pojęć

Operacja domunująca

Operacja decydująca o złożoności algorytmu, np. porównanie dwóch elementów.

Złożoność algorytmu

Mówi o tym ile operacji dominujących musi wykonać dany algorytm w najmniej korzystnym (pesymistyczna), przeciętnym (przeciętna) i najbardziej korzystnym (optymistyczna) przypadku wyjściowego ułożenia elementów w strukturze danych.

Sortowanie – kilka przydatnych pojęć c.d.

Stabilność

Mówi o tym, czy elementy o jednakowej wartości występują w niezmienionej kolejności w strukturze początkowej i po przeprowadzeniu algorytmu sortowania.

Stabilne: np. bąbelkowe, przez scalanie, posiada $O(1)$ space complexity.

Niestabilne: np. szybkie.

Algorytmy "in-place", "out-place"

Mówi o tym, czy do wykonania algorytmu sortowania niezbędne jest wprowadzenie dodatkowych struktur danych (list/tablic).

"In-place": np. szybkie, przez wstawianie.

"Out-place": np. przez scalanie.

Sortowanie przez wybieranie

Sortowanie przez wybieranie

Złożoność czasowa pesymistyczna	n^2
Złożoność czasowa przeciętna	n^2
Złożoność czasowa optymistyczna	n
Stabilność	—
Złożoność pamięciowa	1

nr iteracji (wartość i)	tablica
0	[9,1,6,8,4,3,2, 0]
1	[0, 1 ,6,8,4,3,2,9]
2	[0,1,6,8,4,3, 2 ,9]
3	[0,1,2,8,4, 3 ,6,9]
4	[0,1,2,3, 4 ,8,6,9]
5	[0,1,2,3,4,8, 6 ,9]
6	[0,1,2,3,4,6, 8 ,9]

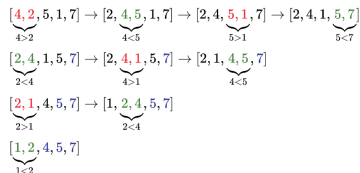
Rysunek: Sortowanie przez wybieranie.

Legenda: czerwony – minimum, czarny pogrubiony – elementy do posortowania, niebieski – elementy na właściwym miejscu.

Sortowanie bąbelkowe

Sortowanie bąbelkowe

Złożoność czasowa pesymistyczna	n^2
Złożoność czasowa przeciętna	n^2
Złożoność czasowa optymistyczna	n
Stabilność	+
Złożoność pamięciowa	1

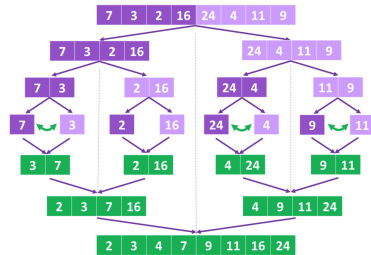


Rysunek: Sortowanie bąbelkowe.

Sortowanie przez scalanie

Sortowanie przez scalanie

Złożoność czasowa pesymistyczna	$n \cdot \log n$
Złożoność czasowa przeciętna	$n \cdot \log n$
Złożoność czasowa optymistyczna	$n \cdot \log n$
Stabilność	+
Złożoność pamięciowa	n



Rysunek: Sortowanie przez scalanie.

Sortowanie szybkie

Sortowanie szybkie

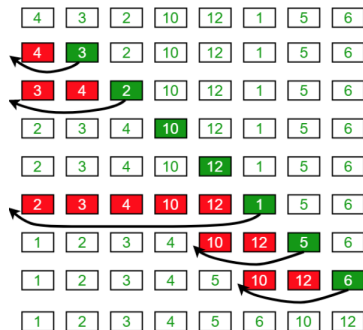
Złożoność czasowa pesymistyczna	n^2
Złożoność czasowa przeciętna	$n \cdot \log n$
Złożoność czasowa optymistyczna	$n \cdot \log n$
Stabilność	—
Złożoność pamięciowa	1

- wydajny algorytm sortowania – dobrze zaimplementowany może być nawet 2-3x szybszy od sortowania przez scalanie czy sortowania z użyciem sterty
- najczęściej stosowany algorytm sortowania
- link do animacji: [click](#)

Sortowanie przez wstawianie

Sortowanie przez wstawianie

Złożoność czasowa pesymistyczna	n^2
Złożoność czasowa przeciętna	n^2
Złożoność czasowa optymistyczna	n
Stabilność	+
Złożoność pamięciowa	1

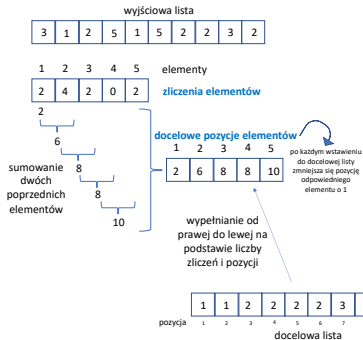


Rysunek: Sortowanie przez wstawianie.

Sortowanie zliczeniowe

Sortowanie zliczeniowe

Złożoność czasowa pesymistyczna	$n + k$
Złożoność czasowa przeciętna	$n + k$
Złożoność czasowa optymistyczna	$n + k$
Stabilność	+
Złożoność pamięciowa	$n + k$



Rysunek: Sortowanie zliczeniowe.

Porównanie wybranych algorytmów sortowania

Tabela: Porównanie parametrów wybranych algorytmów.

Rodzaj sort.	Złoż. czas. pes.	Złoż. czas. przec.	Złoż. czas. opt.	Stabilność	Złoż. pamięć.
p. wybieranie	n^2	n^2	n	—	1
bąbelkowe	n^2	n^2	n	+	1
p. scalanie	$n \cdot \log n$	$n \cdot \log n$	$n \cdot \log n$	+	n
szybkie	n^2	$n \cdot \log n$	$n \cdot \log n$	—	1
p. wstawianie	n^2	n^2	n	+	1
zliczeniowe	$n + k$	$n + k$	$n + k$	+	$n + k$

n – liczba elementów struktury danych, k – liczba możliwych wartości

Użyta notacja to $O()$ – służy do oznaczania rzędu wielkości (górnego ograniczenia tempa wzrostu) funkcji ▶

Zastosowania wyszukiwania i sortowania

Algorytmy wyszukiwania i sortowania są stosowane w celu:

- uporządkowania danych
- zwiększenia wydajności aplikacji (szybsze wykonywanie operacji, wydajniejsze działanie programów)
- prezentacji danych w sposób czytelniejszy dla człowieka

Pakiet java.util.stream

Pakiet java.util.stream

Strumień

- Java Stream to interfejs dostępny od wersji 8
- Interfejs Stream jest interfejsem generycznym. (Przechowuje on informację o typie, który aktualnie znajduje się w danym strumieniu)
- służy do przetwarzania kolekcji danych
- pozwala w łatwy sposób zrównoleglić pracę na danych — przetwarzanie dużych zbiorów danych może być dużo szybsze
- kładzie nacisk na operacje jakie należy przeprowadzić na danych
- wykorzystuje interfejsy funkcyjne, wyrażenia lambda referencje do metod
- może być przetwarzany sekwencyjnie bądź równolegle. Metoda `stream` tworzy sekwencyjny strumień danych. Metoda `parallelStream` tworzy strumień, który jest uruchamiany jednocześnie na kilku wątkach

Jak działa strumień?

- 1 utworzenie strumienia – dokładnie jedna metoda tworząca
- 2 przetwarzanie danych wewnątrz strumienia – przez dowolną liczbę operacji
- 3 zakończenie strumienia – dokładnie jedna metoda kończąca

Operacje:

- operacja nie może posiadać stanu
- operacja nie może modyfikować źródła danych
- dozwolonych operacji jest bardzo dużo (kompletna wiedza w tym temacie w dokumentacji interfejsu Stream)

Przydatne/popularne operacje

- filter – filtruje dane na podstawie jakiegoś warunku
- distinct – wybiera unikalne wartości
- map – przekształca dane w określony sposób
- limit – zwraca strumień ograniczony do zadanej liczby elementów
- forEach – wykonuje jakąś operację dla każdego elementu strumienia
- count – zwraca liczbę elementów w strumieniu
- collect – pozwala na stworzenie nowego typu na podstawie strumienia, np. listy
- findFirst, findAny

Dobre praktyki

- filtrowanie na początku – warto na samym początku ograniczyć liczbę elementów, a zatem czas wykonania operacji
- unikanie skomplikowanych wyrażeń lambda – kwestie czytelności kodu
- używanie strumieni w określonych przypadkach – nie nadużywanie ich (np. przypadki gdy trzeba użyć słowa kluczowego `break` wykluczają zastosowanie strumienia)
- strumień nie służy do przechowywania danych – nie są strukturą danych

Przykład

```
public class Student {  
    private String name;  
    private double gpa;  
    private String major;  
  
    public Student(String name, double gpa, String major) {  
        this.name = name;  
        this.gpa = gpa;  
        this.major = major;  
    }  
    public String getName() {  
        return name;  
    }  
    public double getGpa() {  
        return gpa;  
    }  
    public String getMajor() {  
        return major;  
    }  
    public String getStudent() {  
        return getName() + "\n_gpa:~" + getGpa() + "\n_maj~" + getMajor() + "\n";  
    }  
}
```

*gpa – grade-point average

Przykład c. d.

```
public class Main {  
    public static void main(String[] args) {  
        List<Student> students = Arrays.asList(  
            new Student("Adam", 3.60, "computer_science"),  
            new Student("Piotr", 3.70, "computer_science"),  
            new Student("Natalia", 3.50, "medicine"),  
            new Student("Kamil", 3.00, "medicine"),  
            new Student("Monika", 3.20, "computer_science")  
        );  
    }  
}
```

Założmy, że chcemy otrzymać tylko osoby, które mają major computer science i gpa powyżej 3.50 oraz major ma być pisany cały wielkimi literami. Wyniki chcemy wydrukować w konsoli.

Opcja 1: można by użyć zagnieżdżonych pętli, w których sprawdzane byłyby kolejne warunki.

Opcja 2: lepsza – przetworzenie tej kolekcji z użyciem strumieni.

Przykład c. d.

```
Stream<Student> studentsStream = students.stream();
studentsStream
    .filter(s -> s.getMajor().equals("computer_science"))
    .filter(s -> s.getGpa() > 3.25)
    .map(s -> s.getStudent().toUpperCase())
    .forEach(System.out::println);
```

Analiza:

- Utworzenie strumienia z listy z użyciem metody `stream()` – jest to metoda domyślna zaimplementowaną w interfejsie `Collection`.
- Przy każdym przekształceniu (`filter`, `map`) tworzona jest nowa instancja strumienia – każdy strumień można by przypisać oddzielnie do zmiennej typu `Stream`.
- W tym przypadku wynik drukowany jest w konsoli przy użyciu referencji do metody `println`, ale mógłby też on być zwracany, np. w postaci nowej listy zawierającej tylko odfiltrowane i przekształcone elementy.
- Operacje na strumieniach wykorzystują wzorec łączenia metod (ang. `method chaining`), zwany także płynnym interfejsem (ang. `fluent interface`).

Źródła i linki

- Towards Data Science – Twierdzenie Bayesa
- Machine Learning Mastery – Twierdzenie Bayesa
- Sedgewick, R., Wayne, K., *Algorytmy. Wydanie IV*, Helion, 2012, ISBN: 978-83-283-3710-7
- Bhargava, A., *Algorytmy. Ilustrowany przewodnik*, Helion, 2017, ISBN: 978-83-283-3445-8
- Algorytm quicksort
- Horstmann, C. S., *Java. Techniki zaawansowane. Wydanie X*, Helion, 2017, ISBN: 978-83-283-3480-9
- Bloch, J., *Java. Efektywne programowanie. Wydanie III*, Helion, 2018, ISBN: 978-83-283-4576-8
- Samouczek programisty – Java Streams