Pytania – egzamin inżynierski

Lidia J. Opuchlik

Polsko – Japońska Akademia Technk Komputerowych

s16478@pjwstk.edu.pl

3 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).

Twierdzenie Bayesa

Twierdzenie Bayesa

Twierdzenie Bayesa

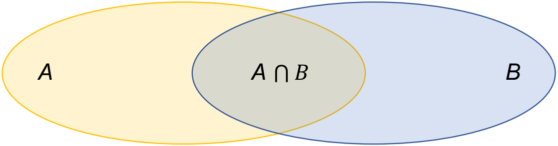
1. Thomas Bayes, presbiteriański pastor, statystyk, filozof w 18-wiecznej Anglii.
2. Mówi o **prawdopodobieństwie warunkowym**. Pozwala określić prawdopodobieństwo zajścia jakiegoś zdarzenia, o ile zaszło jakieś inne zdarzenie.

Treść (dla dwóch zdarzeń)

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

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 definicji:

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

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

Dowód c. d.

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

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

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

$$\Downarrow$$

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

*dowód dla wiekszej 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 $P(grypa|goraczka)$:

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

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

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. Precyzja testu to 95 %. Intuicyjnie nasuwa się, że prawdopodobieństwo wynosi 0.95. Jest to jednak błędne myślenie.

Analiza:

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

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. Załóżmy, że choruje 1/100 osób; 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).

$$P(choroba|pozytywny) = \frac{0.95 \cdot 0.01}{(0.01 \cdot 0.95) + (0.99 \cdot 0.05)} =$$
$$= \frac{0.0095}{0.0095 + 0.0495} = \frac{0.0095}{0.059} = 0.1610169\ldots = 16\ \%$$

Jaki to ma sens? Jedynie 16 % osób z wynikiem pozytywnym ma chorobę.

W czym przydatne jest twierdzenie Bayesa?

1. stosowane do lepszego zrozumienia danych
2. ułatwia zrozumienie i interpretację wyników testów A/B
3. w machine learningu – algorytmy – Naive Bayes, Optymalny Klasyfikator Bayesowski, optymalizacja Bayesowska, 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

Na czym polega??

1. Wartości kolejnych elementów ciągu zaczynając od tego pod indeksem 0 a kończąc na n-1 porównuje się z wartością klucza.
2. Zwraca się indeks znalezionego elementu, bądź -1 w przypadku gdy nie został on znaleziony.

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

Wyszukiwanie binarne

Na czym polega??

1. W posortowanym ciągu porównuje się wartość klucza z wartością środkowego elementu. Jeżeli są takie same, zwracany jest indeks.
2a. Jeżeli klucz $<$ środkowy element, to ogranicza się dalsze przeszukiwanie do lewego podciągu.
2b. Jeżeli klucz $>$ środkowy element, to ogranicza się dalsze przeszukiwanie do prawego podciągu.
3. Powtarza się kroki od punktu 1.
4. Zwraca się indeks pasującego elementu lub -1, gdy nie został znaleziony.

- struktura danych **musi** być posortowana
- wykorzystywany jest algorytm "dziel i rządź"
- pesymistyczna złożoność O(logn)

Wyszukiwanie poprzez skoki co k-ty element

Na czym polega??

1. W posortowanym ciągu przeskakuje się co k elementów i porównuje się wartość bieżącego elementu z wartością klucza.
2a. Gdy element jest mniejszy niż klucz, przeskakuje się o następne k elementów.
2b. Gdy element jest większy niż wartość klucza, to należy sprawdzić już tylko k-1 ostatnio przeskoczonych elementów.
3. Zwraca się indeks pasującego elementu lub -1, gdy nie został znaleziony.

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

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 poczatkowej 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

Algorytm

1. Ustawia się zmienną pomocniczą – np. i – na indeks 0.
2. Szuka się najmniejszego elementu listy.
3. Porównuje się wartość znalezionego minimum z wartością elementu znajdującego się pod bieżącym indeksem – i.
4. Jeżeli znaleziona wartość minimum jest mniejsza od bieżącego elementu to robi się zamianę wartości pod wskazanymi indeksami i inkrementuje się indeks o 1.
5. Jeżeli znaleziona wartość minimum jest większa od bieżącego elementu to inkrementuje się indeks o 1.
6. Powtarza się od 2-giego kroku aż cała lista będzie posortowana.

- algorytm ten dzieli wejściową strukturę danych na podstrukturę posortowaną (lewa) i do posortowania (prawa)

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 – minumum, czarny pogrubiony – elementy do posortowania, niebieski – elementy na właściwym miejscu.

Sortowanie bąbelkowe

Algorytm

1. Zaczynając od początku listy porównuje się wartości dwóch kolejnych elementów listy.
2. Jeżeli ich kolejność jest prawidłowa, to inkrementuje się indeks o 1.
3. Jeżeli ich kolejność jest nieprawidłowa, to zamienia się elementy miejscami i inkrementuje się indeks o 1.
4. Gdy dojdzie się do końca listy, to kroki powtarza się od początku do momentu aż cała lista będzie posortowana.

- listę przechodzi się tyle razy aż wszystkie elementy będą na swoich miejscach
- algorytm ten jest zbyt wolny i niepraktyczny w użyciu

Sortowanie bąbelkowe

Sortowanie bąbelkowe

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

$$[\underbrace{4,2}_{4>2},5,1,7] \to [2,\underbrace{4,5}_{4<5},1,7] \to [2,4,\underbrace{5,1}_{5>1},7] \to [2,4,1,\underbrace{5,7}_{5<7}]$$

$$[\underbrace{2,4}_{2<4},1,5,7] \to [2,\underbrace{4,1}_{4>1},5,7] \to [2,1,\underbrace{4,5}_{4<5},7]$$

$$[\underbrace{2,1}_{2>1},4,5,7] \to [1,\underbrace{2,4}_{2<4},5,7]$$

$$[\underbrace{1,2}_{1<2},4,5,7]$$

Rysunek: Sortowanie bąbelkowe.

Sortowanie przez scalanie

Algorytm

1. Dzieli się początkowy zbiór na kolejne "połowy" dopóki taki podział jest możliwy (tzn. podzbiór zawiera co najmniej dwa elementy).
2. Następnie dla każdych dwóch łączonych na danym poziomie podzbiorów definiuje się wskaźniki i porównuje się elementy przez nie wskazywane.
3. Elementy łączy się (przenosi na kolejny poziom drzewa) w odpowiedniej kolejności. Przy tym inkrementowany jest wskaźnik tego podzbioru, którego element trafił do podzbioru kolejnego poziomu, po czym wykonuje się następne porównanie na tym samym podzbiorze.
4. Porównywanie powtarza się na każdym z poziomów aż do całkowitego posortowania zbioru wyjściowego.

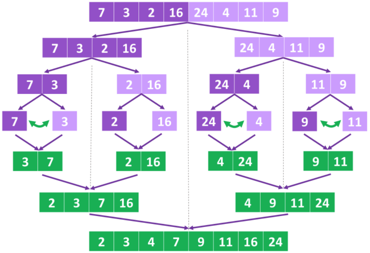
- John von Neumann, 1945
- "dziel i rządź"* – algorytm rekurencyjny

*polega na podziale zadania głównego na zadania mniejsze dotąd, aż rozwiązanie stanie się oczywiste

Sortowanie przez scalanie

Sortowanie przez scalanie

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



Rysunek: Sortowanie przez scalanie.

Sortowanie szybkie

Algorytm

1. Określa się pierwszy element listy jako piwot.
2. Definiuje się dwie zmienne pomocnicze – np. i oraz j – i ustawia się je odpowiednio na pierwszym i ostatnim elemencie listy.
3. Inkrementuje się i dopóki element lista[i] $>$ piwota.
4. Dekrementuje się j dopóki element lista[j] $<$ piwota.
5. Gdy i $<$ j wtedy zamienia się miejscami elementy lista[i] i lista[j].
6. Powtarza się kroki 3, 4, 5 dopóki i $>$ j.
7. Zamienia się miejscami piwot z elementem lista[j].
8. Kontunuuje się na prawej i lewej podstrukturze aż wszystkie lementy będą na swoich miejscach docelowych.

*wartości równe piwotowi idą na arbitralnie ustalaną stronę

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 zaimplemetowany 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

Algorytm

1. Począwszy od drugiego elementu (wskaźnik na drugiej pozycji), wybieramy dany element i porównujemy go z każdym elementem z lewej strony (we wstępnie posortowanej części).
2. Przesuwamy i wstawiamy go na odpowiednie miejsce.
3. Powtarzamy działanie dla kolejnych elementów ciągu (wskaźnik zawsze przesuwa się o 1 dalej) aż wszystkie elementy będą na swoim miejscu.

- algorytm wydajny do sortowania małych struktur
- jest bardziej wydajny niż pozostałe algorytmy posiadające złożoność kwadratową

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

Algorytm

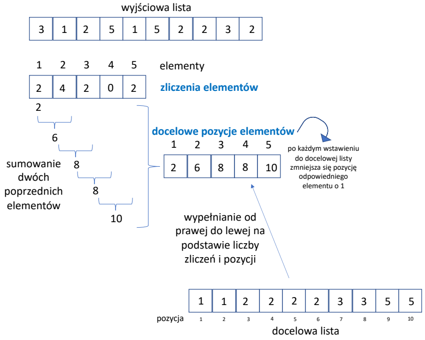
1. Dla każdej wartości w zbiorze przygotowujemy licznik.
2. Przeglądamy kolejne elementy zbioru i zliczamy ich wystąpienia w odpowiednich licznikach.
3. Poczynając od drugiego licznika sumujemy zawartość licznika oraz jego poprzednika*.
4. Przeglądamy jeszcze raz zbiór wejściowy idąc od ostatniego elementu do pierwszego. Każdy element umieszczamy w zbiorze wynikowym na pozycji równej zawartości licznika dla tego elementu. Po wykonaniu tej operacji licznik zmniejszamy o 1. Dzięki temu następna taka wartość trafi na wcześniejszą pozycję.

* w każdym liczniku otrzymaliśmy ilość wartości mniejszych lub równych numerowi licznika.

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. pamięc.
p. wybieranie	n^2	n^2	n	$-$	1
bąbelkowe	n^2	n^2	n	$+$	1
p. scalanie	$n \cdot logn$	$n \cdot logn$	$n \cdot logn$	$+$	n
szybkie	n^2	$n \cdot logn$	$n \cdot logn$	$-$	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

Wykorzystują je interaktywne aplikacje, np. webowe, desktopowe, do księgowości, naukowe, zakupowe, platformy streamingowe, portale i wiele innych. Posiadają one wbudowane moduły wyszukiwania i sortowania żeby ułatwić użytkownikowi znalezienie porządanego (optymalnego) rozwiązania.

Pakiet java.util.stream

Pakiet java.util.stream

Pakiet java.util.stream