

Wprowadzenie - jak opisać algorytm?

Tomasz M. Gwizdałła

Notes

Algorytm

Dobrze zdefiniowana procedura obliczeniowa, pobierająca oraz oddającą pewien zestaw wartości.

Dokładny przepis podający sposób rozwiązania określonego zadania w skończonej liczbie kroków; zbiór poleceń odnoszących się do pewnych obiektów, ze wskazaniem porządku, w jakim mają być realizowane.

Notes

Cechy algorytmów

- ▶ skończoność
- ▶ określoność
- ▶ wejście
- ▶ wyjście
- ▶ efektywność
- ▶ poprawność!!!

Notes

Etymologia nazwy i pierwszy algorytm

Abu Abdullah Muhammad ibn Musa al-Chwarizmi (780-850 AD) - matematyk perski/uzbecki/turkmeński, pracujący w Bağdadzie.

Algorytm Euklidesa (365-300 AC) - algorytm wyznaczania największego wspólnego dzielnika dwóch liczb.

Notes

Jak działa algorytm Euklidesa?

Algorytm Euklidesa, którego celem jest wyznaczenie największego wspólnego dzielnika dwóch liczb naturalnych (NWD, GCD), bazuje na pewnej prostej obserwacji. Jeżeli pewna liczba jest dzielnikiem dwóch liczb całkowitych, to jest także dzielnikiem różnicy tych liczb.

Aby zatem znaleźć największy wspólny dzielnik należy odjąć od liczby większej mniejszą i powtórzyć proces dla liczby mniejszej z pary początkowej oraz różnicy liczb początkowych.

Algorytm należy przerwać, gdy obie liczby są sobie równe.

Notes

Metody przedstawiania algorytmów

- ▶ Opis słowny (na poprzednim slajdzie)
- ▶ Lista kroków
- ▶ Schemat blokowy
- ▶ Drzewa algorytmiczne
- ▶ Język programowania/pseudokod

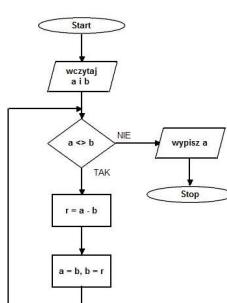
Notes

Lista kroków

- ▶ Wczytaj dwie liczby
- ▶ Znajdź wynik odejmowania mniejszej od większej
- ▶ Zastąp liczbę większą obliczoną różnicą
- ▶ Kiedy obie liczby są równe zakończ

Notes

Schemat blokowy

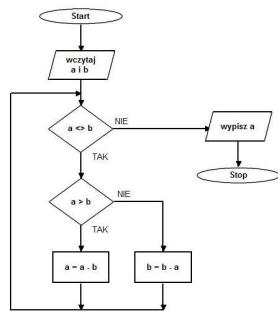


Skrzynki:

- ▶ operacyjna
- ▶ warunkowa
- ▶ wejścia/wyjścia
- ▶ graniczne
- ▶ komentarza
- ▶ łącznikowa

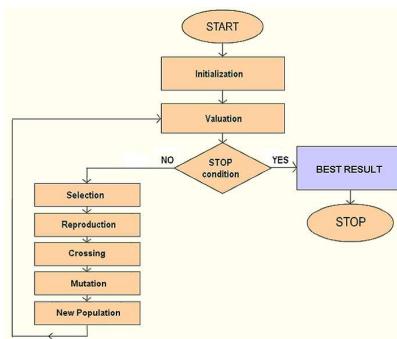
Notes

Schemat blokowy



Notes

Schemat blokowy



Notes

Kod źródłowy

```

long nwd1(long a, long b)
{
    while(a != b) {
        if(a > b) a=a-b;
        else b=b-a;
    }
    return a;
}
    
```

Notes

Czy algorytm Euklidesa jest poprawny?

Pojęcie **niezmiennika pętli**.

Mówimy, że pewne zdanie jest niezmiennikiem pętli, jeżeli po każdym jej przebiegu jest ono prawdziwe.

Co może być niezmiennikiem pętli w algorytmie Euklidesa?

Notes

$$NWD(a, b) = NWD(\min(a, b), \max(a, b) - \min(a, b))$$

Założymy, dla ustalenia uwagi, $a > b$ i niech $n = NWD(a, b)$

$$\begin{aligned} a &= k * n, \quad b = l * n \\ a - b &= k * n - l * n = (k - l) * n \end{aligned}$$

Czy algorytm Euklidesa się zatrzyma?

Musimy rozpatrzyć inne wartości, nie będące niezmiennikami.

Niech będzie nim suma $c = a + b$.

Wiadomo, że suma $a + b$ w pewnym kroku jest równa większej z liczb w kroku poprzedzającym.

Ponumerujmy zatem kroki indeksem i .

Ciąg c_i jest ścisłe malejący $c_1 > c_2 > \dots > c_m$.

Notes

Notes

Notes

Notes

Złożoność obliczeniowa i metody analizy algorytmów

Tomasz M. Gwizdała

- Złożoność i metody analizy
 - └ Złożoność pamięciowa
 - └ Pojęcie

Złożoność algorytmu

Musimy uzależnić parametry opisujące działanie algorytmu od wielkości zbioru danych, na którym ten algorytm operuje.

złożoność pamięciowa

przyjmuje się, że jednostką złożoności pamięciowej jest słowo maszyny

złożoność obliczeniowa

przyjmuje się, że jednostką złożoności czasowej jest wykonanie jednej operacji dominującej

Złożoność i metody analizy
└ Złożoność pamięciowa
 └ Niebezpieczeństwa

Złożoność pamięciowa

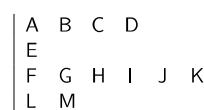
Zwykle nie ma problemu

f	z	e	j	r	a	2	5	8	1	6	3
						9	4	2	7	1	4
						0	2	7	3	5	9

Jakaś funkcja n

Jakaś funkcja $n^2 / (n \times m)$

Ale struktury mogą być organizowane w inny sposób np.



Złożoność i metody analizy

- └ Złożoność obliczeniowa

Złożoność obliczeniowa

Jaka jest złożoność obliczeniowa operacji mnożenia liczb całkowitych? Np. 5^*7 , czyli 101^*111 ?

$$\begin{array}{r}
 & 1 & 1 & 1 \\
 & 1 & 0 & 1 \\
 \hline
 & 1 & 1 & 1 \\
 1 & 1 & 1 \\
 \hline
 1 & 0 & 0 & 0 & 1 & 1
 \end{array}$$

No to weźmy liczby trochę większe: $1010100 \cdot 1048576$
czyli
 $11110110100110110100 \cdot 1000000000000000000000000000$

Podsumujmy

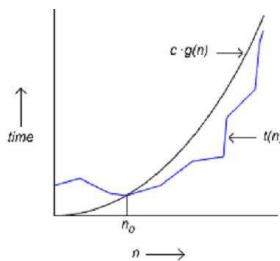
- Może nastąpić przekroczenie zakresu danego typu zmiennych
- Wbudowane w kompilator mechanizmy działania funkcjonują niepoprawnie
- Zachowanie programu nie jest jednoznaczne
- Musimy umieć odróżnić sytuacje, w których pewna zależność opisuje złożoność jednoznacznie, od tych, w których jest to tylko szacowanie
- Musimy umieć określić, co jest operacją dominującą

Notes

Notacja O

Niech $f(n)$ oznacza funkcję opisującą zależność liczby operacji dominujących od rozmiaru struktury danych, $g(n)$ pewną funkcję referencyjną

$$f(n) = O(g(n)) \Leftrightarrow \exists c > 0, n_0 > 0 \forall n > n_0 \ 0 < f(n) < cg(n)$$

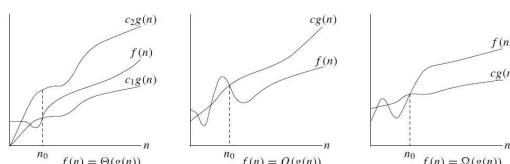


Notes

Notacja Θ i Ω

$$f(n) = \Omega(g(n)) \Leftrightarrow \exists c > 0, n_0 > 0 \forall n > n_0 \ 0 < cg(n) < f(n)$$

$$f(n) = \Theta(g(n)) \Leftrightarrow \exists c_1 > 0, c_2 > 0, n_0 > 0 \forall n > n_0 \ c_1 g(n) < f(n) < c_2 g(n)$$



Notes

Przykład

Niech $f(n) = n^2 - 2n + 1$

$$n^2 - 2n + 1 = O(n^2), \text{ bo } 1.0001n^2 > n^2 - 2n + 1 \quad (1)$$

$$\text{dla } n_0 = 1$$

$$n^2 - 2n + 1 = O(n^3), \text{ bo } 10^{-100}n^3 > n^2 - 2n + 1$$

$$\text{dla } n_0 \approx 1 * 10^{100}$$

$$n^2 - 2n + 1 \neq O(n), \text{ bo } n < n^2 - 2n + 1$$

$$\text{dla } n_0 \approx 3$$

Notes

Przykład

Niech $f(n) = n^2 - 2n + 16$

$$\begin{aligned} n^2 - 2n + 1 &= \Omega(n^2), \text{ bo } 0.9999n^2 < n^2 - 2n + 1 & (2) \\ \text{dla } n_0 &\approx 30000 \\ n^2 - 2n + 1 &\neq \Omega(n^3), \text{ bo } n^3 > n^2 - 2n + 1 \\ \text{dla } n_0 &= 1 \\ n^2 - 2n + 1 &= \Omega(n), \text{ bo } 10^{100}n < n^2 - 2n + 1 \\ \text{dla } n_0 &\approx 1 * 10^{100} \end{aligned}$$

Notes

Przykład

Dla dowolnych funkcji $f(n)$ i $g(n)$ spełnione jest

$$f(n) = \Theta(g(n)) \Leftrightarrow (f(n) = O(g(n)) \wedge f(n) = \Omega(g(n)))$$

$$\begin{aligned} n^2 - 3n + 2 &= \Theta(n^2) & , \text{ bo zarówno } O, \text{ jak i } \Omega \\ n^2 - 3n + 2 &\neq \Theta(n^3) & , \text{ bo tylko } O \\ n^2 - 3n + 2 &\neq \Theta(n) & , \text{ bo tylko } \Omega \end{aligned}$$

Notes

Notacja o i ω

$$f(n) = o(g(n)) \Leftrightarrow \forall c > 0 \exists n_0 > 0 \forall n > n_0 0 < f(n) < cg(n)$$

$$f(n) = \omega(g(n)) \Leftrightarrow \forall c > 0 \exists n_0 > 0 \forall n > n_0 cg(n) < f(n)$$

A co z notacją θ ?

Notes

Przykład

Niech $f(n) = n^2 - 2n + 1$

$$\begin{aligned} n^2 - 2n + 1 &\neq o(n^2), \text{ bo } 0.9999n^2 < n^2 - 2n + 1 & (3) \\ n^2 - 2n + 1 &= o(n^3), \text{ bo } 10^{-100}n^3 > n^2 - 2n + 1 \\ n^2 - 2n + 1 &\neq o(n), \text{ bo } n < n^2 - 2n + 1 \end{aligned}$$

Notes

$$\begin{aligned} n^2 - 2n + 1 &\neq \omega(n^2), \text{ bo } 1.0001n^2 > n^2 - 2n + 1 & (4) \\ n^2 - 2n + 1 &\neq \omega(n^3), \text{ bo } n^3 > n^2 - 2n + 1 \\ n^2 - 2n + 1 &= \omega(n), \text{ bo } 10^{100}n < n^2 - 2n + 1 \end{aligned}$$

Jak można inaczej określić złożoność obliczeniową?

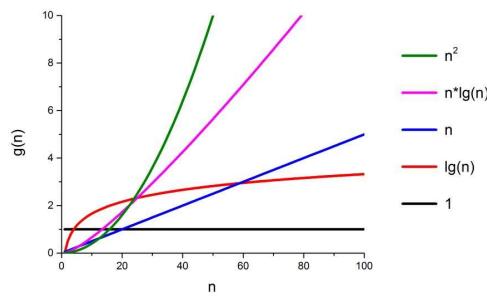
$$f(n) = O(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \quad f(n) = o(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \Omega(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0 \quad f(n) = \omega(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$$f(n) = \Theta(g(n)) \Leftrightarrow 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

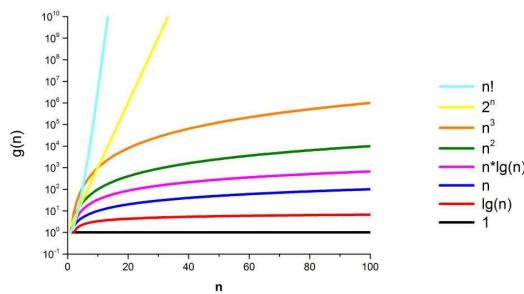
Notes

Przykłady złożoności obliczeniowej



Notes

Przykłady złożoności obliczeniowej



Notes

Inne charakterystyki

Oznaczmy

- D_n - przestrzeń konfiguracyjna zbiorów danych wejściowych
- $t(d \in D_n)$ - liczba operacji dominujących dla konfiguracji d
- X_n - zmienna losowa, której wartością jest $t(d)$

Notes

	pesymistyczna	oczekiwana
złożoność czasowa	$W(n) = \max(X_n)$	$< X_n >$
wrażliwość czasowa	$W(n) = \max(X_{n1} - X_{n2})$	$\sqrt{(< X_n^2 > - < X_n >^2)}$

Metody

- ▶ Metoda ząbkowa
- ▶ Metoda „dziel i zwyciężaj”
- ▶ Metoda dynamiczna
- ▶ Metoda zstępująca (top-down) i wstępująca (bottom-up)

Notes

Metoda ząbkowa - iteracja

```
double power_iterative(double base, long exponent)
{
    double ret=1.0;
    for(int i=0;i<exponent;i++) ret*=base;
    return ret;
}
```

Złożoność pamięciowa - $O(1)$
Złożoność obliczeniowa - $O(n)$

Notes

Metoda ząbkowa - rekursja

```
double power_recursive(double base, long exponent)
{
    if(exponent==0) return 1.0;
    else return base*power_recursive(base,exponent-1);
}
```

Złożoność pamięciowa - $O(n)$
Złożoność obliczeniowa - $O(n)$

Notes

Metoda dynamiczna

```
double power_dynamic (double base, long exponent)
{
    if(exponent==0) return 1.0;
    double b=power_dynamic (base,exponent/2);
    if(exponent%2==0) return b*b;
    else return b*b*base;
}
```

Złożoność pamięciowa - $O(\log(n))$
Złożoność obliczeniowa - $O(\log(n))$

Notes

Metody analizy algorytmów

- ▶ Analiza równań rekurencyjnych
- ▶ Analiza funkcji tworzącej

Notes

Analiza równań rekurencyjnych

- ▶ Metoda podstawienia (Substitution)
- ▶ Metoda drzew rekurencyjnych (Recurrence-tree)
- ▶ Metoda rekurencji uniwersalnej (Master method)

Notes

Podstawowe oznaczenia

Niech $T(n)$ oznacza czas potrzebny dla wykonania pewnego algorytmu

Założymy, że

$$\begin{aligned}T(1) &= 0 \\T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n = 2T(\lfloor n/2 \rfloor) + n\end{aligned}$$

Notes

Metoda podstawienia

Wykorzystanie indukcji matematycznej dla odgadniętego rozwiązania

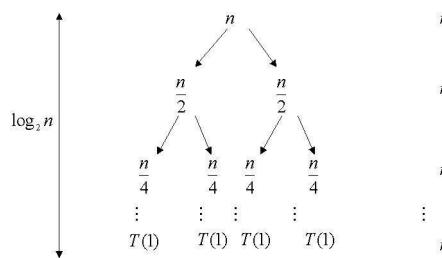
Odgadnijmy, że $T(n) = O(n \log(n))$

Notes

$$\begin{aligned}T(n) &\leq 2c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor) + n \\&\leq cn\log(n/2) + n \\&= cn\log(n) - cn\log(2) + n \\&= cn\log(n) + n(1 - c\log(2)) \\&\leq cn\log(n)\end{aligned}$$

ponieważ $c > 0$, $\log(2) > 0$, jeśli c jest odpowiednio duże

Drzewa rekurencyjne



Ostatecznie $O(n \log(n))$

Notes

Rekurencja uniwersalna

Wykorzystywana dla zagadnień typu

$$T(n) = aT(n/b) + f(n)$$

$$\exists_{\epsilon > 0} f(n) = O(n^{\log_b a - \epsilon}) \Rightarrow T(n) = n^{\log_b a}$$

$$f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = n^{\log_b a} \log(n)$$

$$\exists_{\epsilon > 0} f(n) = \Omega(n^{\log_b a + \epsilon}) \wedge \exists_{c < 1} f(n/b) \leq c f(n) \Rightarrow T(n) = f(n)$$

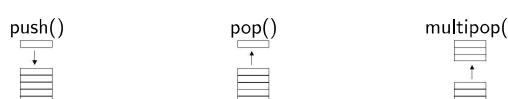
Notes

Metody kosztu zamortyzowanego

- ▶ Koszt sumaryczny (aggregate analysis)
- ▶ Księgowanie (accounting method)
- ▶ Metoda potencjału (potential method)

Notes

Koszt sumaryczny



Jaka jest złożoność obliczeniowa operacji zdjęcia wszystkich elementów ze stosu o rozmiarze n?

Koszt pojedynczej operacji

$O(1)$
Ile operacji

n
 $O(n)$

$O(1)$

n
 $O(n)$

$O(n)$

n
 $O(n^2)$

Notes

Koszt sumaryczny

Złożoność obliczamy „od tyłu”

- ▶ Wiemy, że czas niezbędny na zdobycie wszystkich elementów ze stosu nie może być większy, niż „n”.
- ▶ Zatem złożoność obliczeniowa operacji zdobycia wszystkich elementów ze stosu nie może być gorsza, niż $O(n)$.
- ▶ Czyli złożoność obliczeniowa operacji multipop była określona nieprawidłowo. Trzeba znaleźć uśredzoną złożoność $O(n)/n = O(1)$

Notes

Księgowanie

- ▶ Poszczególnym operacjom przypisywane są różne koszty
- ▶ Mogą one być zarówno mniejsze, jak i większe od kosztu faktycznego
- ▶ Koszt zamortyzowany operacji jest sumą kosztu faktycznego i „kredytu”

operacja	koszt faktyczny	koszt zamortyzowany
push	1	2
pop	1	0
multipop	$\min(k, s)$	0

Notes

Metoda potencjału

- ▶ Energia potencjalna (lub potencjał) jest funkcją stanu.
- ▶ Koszt zamortyzowany operacji jest sumą jej kosztu faktycznego i zmiany wartości pewnej funkcji związanej z aktualnym stanem struktury danych.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)$$

Notes

Metoda potencjału

Niech $\Phi(i)$ będzie rozmiarem stosu

Dla operacji push()

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$

Dla operacji multipop(k)

$$\hat{c}_i = c_i + \Phi(D_{i-k}) - \Phi(D_i) = k - k = 0$$

Notes

Notes

Podstawowe struktury danych

Tomasz M. Gwizdałła

Definicje

Skończony i uporządkowany ciąg elementów

$$q = [node_1, node_2, \dots, node_n]$$

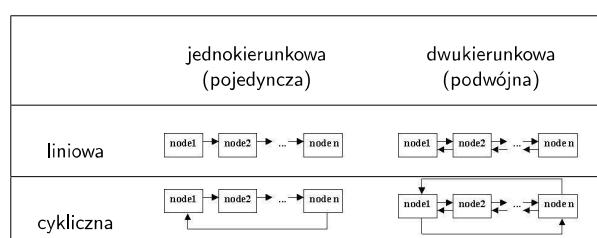
- ▶ node - węzeł
- ▶ n - rozmiar listy
- ▶ $node_1, node_n$ - końce listy (lewy, prawy)
- ▶ jeden z końców - głowa, korzeń, wartownik

Operacje dostępne na liście

	left end	right end
add (wstawienie)	push (prepend)	inject (append)
remove (usunięcie)	pop	eject
get (pobranie)	front	rear
	go left	go right

Problem implementacji związany z zagadnieniem dostępu

Typy list



- ▶ posortowana
- ▶ pusta

Przykłady

Założymy, że mamy listę, dla której zdefiniowano następujące operacje:

- ▶ push, pop, front
- ▶ inject, pop, front

Notes

Przykłady

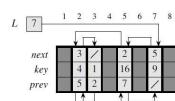
Założymy, że mamy listę, dla której zdefiniowano następujące operacje:

- ▶ push, pop, front - STOS
- ▶ inject, pop, front - KOLEJKA

Notes

Implementacje

- ▶ Dowiązaniowa (pomiędzy sąsiednimi elementami):
 singly/doubly linked list
 multiple array



- ▶ Tablicowa:
 spójny obszar pamięci

Notes

Definicje

- ▶ Zbiór obiektów (O) należących do tej samej klasy
- ▶ Każdy obiekt w zbiorze charakteryzowany jest przez dwa atrybuty element (E) oraz klucz (K)
- ▶ Dla wartości kluczowych musi być jednoznacznie zdefiniowany operator większości $<$

Notes

Rozmiar fizyczny i logiczny tablicy

int table[10]; - ustalone rozmiar fizyczny (10)

int n = 5; for(int i = 0; i < n; i++) table[i] = i;
ustalone rozmiar logiczny (5)

$$\text{load factor} = \frac{\text{rozmiar logiczny}}{\text{rozmiar fizyczny}}$$

Notes

Funkcjonalność tablicy z kluczem

- ▶ Tablica uporządkowana - przeszukiwanie binarne
Większa złożoność obliczeniowa operacji "add", mniejsza "get"
- ▶ Tablica nieuporządkowana - przeszukiwanie sekwencyjne
Większa złożoność obliczeniowa operacji "get", mniejsza "add"

Notes

Definicje

Hash table - tablica z haszowaniem

- ▶ Tablica, w której lokalizacja obiektu wyznaczana jest przez specjalną funkcję, zwany mieszającą lub haszującą
- ▶ Tablice z haszowaniem są stosowane, gdy z pewnego dużego zbioru danych w danej chwili musimy znać stosunkowo jego niewielki podzbiór
- ▶ Tablice z haszowaniem są stosowane dla obiektów z kluczami

Notes

Podstawy

Ideą haszowania jest zmniejszenie

- ▶ złożoności pamięciowej (w przypadku tworzenia tablicy "od zera")
- ▶ złożoności obliczeniowej (w przypadku wyszukiwania w istniejącej tablicy)

Notes

Podstawą tablic z haszowaniem jest dobra funkcja haszująca. Miarą tej "dobroci" jest stopień, w jakim spełnia ona zasadę prostego równomiernego haszowania, tzn: losowo wybrany klucz jest z takim samym prawdopodobieństwem odwzorowywany na każdą z pozycji

Przykład

Założymy, że rozmiar fizyczny tablicy jest równy n (np. niech $n = 10$)
 Jako funkcję haszującą wybierzmy funkcję typu $k \bmod n$ ($k \bmod 10$)

i	klucz
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

i	klucz
0	
1	
2	
3	
4	
5	
6	
7	517
8	
9	

$\xrightarrow{\text{hash}(517) = 517 \% 10 = 7}$

Notes

Przykład

Dodajmy kilka następnych elementów

i	klucz
0	
1	
2	
3	
4	
5	
6	
7	517
8	
9	

i	klucz
0	
1	
2	332
3	
4	284
5	725
6	
7	517
8	
9	

$\xrightarrow{\text{hash}(332) = 332 \% 10 = 2}$

$\xrightarrow{\text{hash}(284) = 284 \% 10 = 4}$

$\xrightarrow{\text{hash}(725) = 725 \% 10 = 5}$

$\xrightarrow{\text{hash}(167) = 167 \% 10 = 7 \text{ !!!}}$

Notes

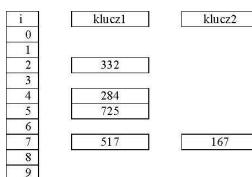
Adresowanie otwarte

$$\text{hash} \leftarrow (\text{hash} + 1) \% n$$

i	klucz	status
0		pusta
1		pusta
2	332	zajęta
3		pusta
4	284	zajęta
5	725	zajęta
6		pusta
7	517	łancuch
8	167	zajęta
9		pusta

Notes

Adresowanie łańcuchowe



Notes

Uwagi

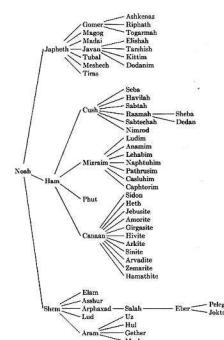
- ▶ Zwykle przyjmuje się jako rozmiar fizyczny tablicy odpowiednio dużą liczbę pierwszą
- ▶ Odpowiednio dużą oznacza zwykle taką aby load factor był równy około 0.6-0.7
- ▶ Wzrost load factor powoduje również wzrost czasu przeszukiwania tablicy

Notes

Definicje

Drzewem nazywamy zbiór węzłów spełniających następujące warunki

- ▶ Istnieje wybrany węzeł, nazywany korzeniem
- ▶ Pozostałe węzły są podzielone na $m \geq 0$ zbiorów, z których każdy też jest drzewem. Zbiory te nazywamy poddrzewami.

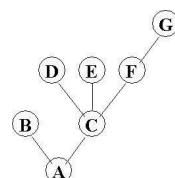


Notes

Budowa

Wierzchołki

- ▶ poprzedniki i następni
- ▶ rodzice i dzieci



Trzy rodzaje wierzchołków
korzeń
gałęzie
liście

A
C, F
B, D, E, G

Alternatywne rozumienie pojęcia gałęzi: abstrakcyjny fakt potwierdzający powiązanie dwóch węzłów drzewa

Notes

Właściwości drzewa i jego wierzchołków

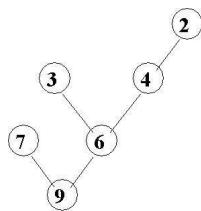
- ▶ (w) długość drogi - liczba wierzchołków, przez które należy przejść od korzenia do wierzchołka w
- ▶ (w) wysokość - liczba węzłów na drodze od w do liścia będącego najbliższym następnikiem
- ▶ wysokość, głębokość - wysokość korzenia+1
- ▶ (w1,w2) ścieżka - zbiór wierzchołków, przez które należy przejść z wierzchołka w1 do w2
- ▶ (w) stopień - liczba bezpośrednich następników
- ▶ (d) stopień - najwyższy stopień wierzchołka

Notes

Typy drzew

Drzewo binarne - stopień drzewa jest równy 2

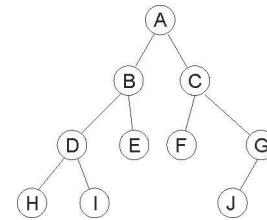
Kopiec (stógi) - drzewo binarne, w którym następnik jest niewiększy od swojego poprzednika (*warunek kopca*)



Notes

Pre-order, przeszukiwanie wzdłużne

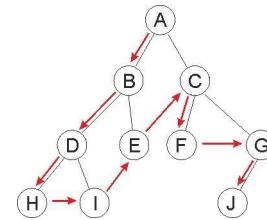
```
pre-order(node& v)
{
    oper(v);
    if(v->left!=0)
        pre-order(v->left)
    if(v->right!=0)
        pre-order(v->right)
}
```



Notes

Pre-order, przeszukiwanie wzdłużne

```
pre-order(node& v)
{
    oper(v);
    if(v->left!=0)
        pre-order(v->left)
    if(v->right!=0)
        pre-order(v->right)
}
```

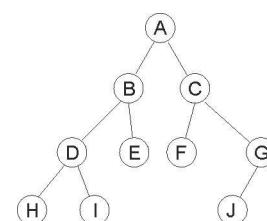


kolejność: A,B,D,H,I,E,C,F,G,J

Notes

In-order, przeszukiwanie poprzeczne

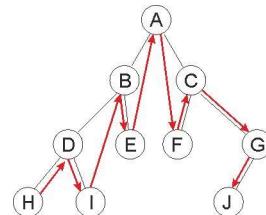
```
in-order(node& v)
{
    if(v->left!=0)
        in-order(v->left)
    oper(v);
    if(v->right!=0)
        in-order(v->right)
}
```



Notes

In-order, przeszukiwanie poprzeczne

```
in-order(node& v)
{
    if(v->left!=0)
        in-order(v->left)
    oper(v);
    if(v->right!=0)
        in-order(v->right)
}
```

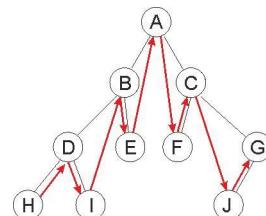


kolejność: **H,D,I,B,E,A,F,C,G,J**

Notes

In-order, przeszukiwanie poprzeczne

```
in-order(node& v)
{
    if(v->left!=0)
        in-order(v->left)
    oper(v);
    if(v->right!=0)
        in-order(v->right)
}
```

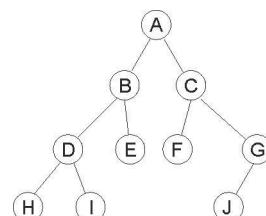


kolejność: **H,D,I,B,E,A,F,C,J,G**

Notes

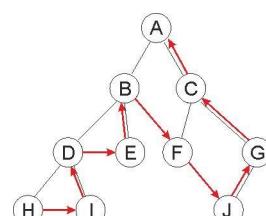
Post-order, przeszukiwanie wsteczne

```
post-order(node& v)
{
    if(v->left!=0)
        post-order(v->left)
    if(v->right!=0)
        post-order(v->right)
    oper(v);
}
```



Post-order, przeszukiwanie wsteczne

```
post-order(node& v)
{
    if(v->left!=0)
        post-order(v->left)
    if(v->right!=0)
        post-order(v->right)
    oper(v);
}
```

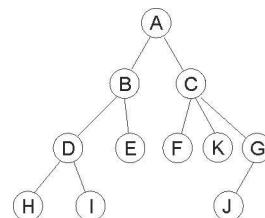


kolejność: **H,I,D,E,B,F,J,G,C,A**

Notes

Inne pytania

Które metody przechodzenia drzewa można zastosować dla drzew o wyższym stopniu?



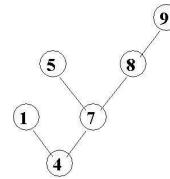
Co z odwróceniem порядku przechodzenia w drzewie binarnym?

Notes

Typy drzew - BST

Drzewo poszukiwań binarnych (BST - Binary Search Tree)

- ▶ istnieją wyróżnione kierunki lewy i prawy
- ▶ jeżeli element znajduje się w lewym poddrzewie, to jego klucz jest mniejszy od klucza korzenia
- ▶ jeżeli element znajduje się w prawym poddrzewie, to jego klucz jest większy od klucza korzenia



Notes

Algorytmy związane z BST

- ▶ algorytm znajdowania wierzchołka o kluczu minimalnym (maksymalnym)
- ▶ algorytm przeszukiwania (znajdowanie wierzchołka o zadanym kluczu)
- ▶ algorytm dodawania wierzchołka
- ▶ algorytm usuwania wierzchołka

Założymy, że mamy konstruktor węzła postaci

```
construct()  
{  
    left=0; right=0;  
}
```

Notes

Algorytm przeszukiwania

```
node& search(key v)  
{  
    node x = root;  
    while( (x!=0) && (key(x)!=v) ) {  
        if(v<key(x)) x = left(x);  
        else x=right(x);  
    }  
    return x;  
}
```

Notes

Algorytm przeszukiwania

```
node& search(key v, node& y)
{
    node x = root;
    while( (x!=0) && (key(x)!=v) ) {
        y=x;
        if(v<key(x) ) x = left(x);
        else x=right(x);
    }
    return x;
}
```

Notes

Algorytm dodawania elementu do BST

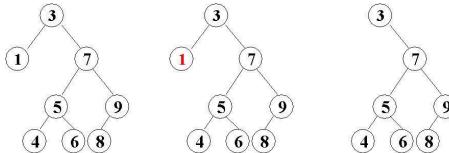
```
void add(node v)
{
    node y;
    if( search(key(v),y)==0) {
        if( key(v)<key(y) ) left(y) = v;
        else right(y) = v;
    }
}
```

Notes

Algorytm usuwania elementu z BST

W algorytmie usuwania występuje kilka możliwości
Niech x będzie elementem do usunięcia, a y jego poprzednikiem

Jeśli x jest liściem, element kasujemy ustawiając odpowiednią
wartość poprzednika y na 0

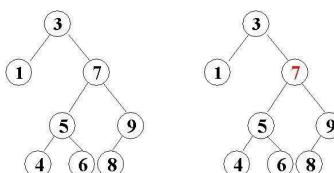


Notes

Algorytm usuwania elementu z BST

Jeśli x nie jest liściem, w zależności od tego, czy x.left!=0, czy
x.right!=0 możemy:

- ▶ Zastąpić x przez liść o największym kluczku z lewego poddrzewa.
- ▶ Zastąpić x przez liść o najmniejszym kluczku z prawego poddrzewa

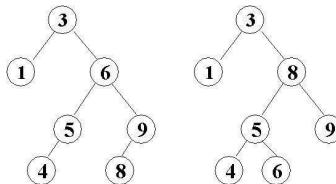


Notes

Algorytm usuwania elementu z BST

Wstawienie w miejsce kasowanego elementu jego bezpośredniego potomka prowadzi do utworzenia drzewa, które nie jest drzewem BST.

Musimy zatem zrobić tak:



Notes

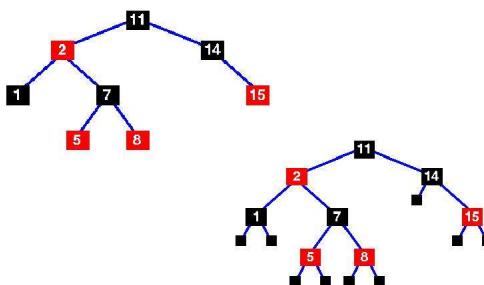
Drzewa czerwono-czarne - RBT

Drzewem czerwono-czarnym nazywamy BST, w którym każdy wierzchołek posiada dodatkową cechę - kolor, spełniające następujące warunki

- ▶ Każdy wierzchołek jest czerwony lub czarny
- ▶ Korzeń jest czarny
- ▶ Każdy liść jest czarny
- ▶ Jeśli wierzchołek jest czerwony, to oba jego następcy są czarne
- ▶ Na każdej ścieżce prowadzącej z danego wierzchołka do liścia będącego jego następcą jest jednakowa liczba czarnych wierzchołków

Notes

RBT - przykład



Drzewo czerwono-czarne o n wierzchołkach ma wysokość co najwyżej $2 * \log(n + 1)$

Notes

RBT - dodawanie węzła

```
rb_insert( Tree T, node x ) {
    tree.insert( T, x ); //dodaj do drzewa
    x->colour = red; //pokoloruj na czerwono
    while ( (x != T->root) && (x->parent->colour == red) ) {
        if ( x->parent == x->parent->parent->left ) {
            // wykonaj działania, jeśli rodzic x jest lewym dzieckiem
        } else {
            // wykonaj działania, jeśli rodzic x jest prawym dzieckiem
        }
        T->root->colour = black; // korzeń musi być czarny
    }
```

Notes

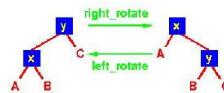
RBT - dodawanie węzła - c.d.

```
y = x->parent->parent->right; //y jest prawym "wujkiem"
if ( y->colour == red ) { //y jest czerwony -> zmień kolory
    x->parent->colour = black; y->colour = black;
    x->parent->parent->colour = red; x = x->parent->parent;
} else {
    if ( x == x->parent->right ) {
        x = x->parent; left_rotate( T, x );
    }
    right_rotate( T, x->parent->parent );
    x->parent->colour = black;
    x->parent->parent->colour = red;
}
```

Notes

RBT - rotacja

Rotacja jest operacją która zamienia pozycję rodzica i jednego z dzieci



Notes

Inne drzewa

- ▶ drzewa AVL (Adielson-Wielski i Łandis) - BST, w którym dla każdego wierzchołka jego oba poddrzewa różnią się wysokością co najwyżej o 1
- ▶ drzewa poszukiwań pozycyjnych (RST, TRIE, PATRICIA)
- ▶ drzewa BSP - Binary Space Partition

Notes

Notes

Notes

Algorytmy sortowania

Tomasz M. Gwizdała

Notes

Różne kryteria klasyfikacji

Ze względu na złożoność pamięciową:

- ▶ sortujące w miejscu (in situ/in place)

Ze względu na zachowanie porządku:

- ▶ stabilne (zachowujące porządek elementów o tym samym kluczu)
- ▶ niestabilne

Ze względu na możliwość przechowywania danych:

- ▶ zewnętrzne

Notes

Sortowanie przez wstawianie

W ramach algorytmu sortowania przez wstawianie wykonujemy następujące czynności:

1. Wyszukujemy miejsce, w które należy wstawić nowy element, przechowywany chwilowo na innej liście.
2. Wstawiamy element.

 0 1 2 3 4 5 6 7 8 9 ← 7 1 5 6 2 4 8 9 3 0

Notes

Sortowanie przez wstawianie

7 1 5 6 2 4 8 9 3 0

7

1 5 6 2 4 8 9 3 0

1 7

5 6 2 4 8 9 3 0

1 5 7

6 2 4 8 9 3 0

...

0 1 2 3 4 5 6 7 8 9 7 1 5 6 2 4 8 9 3 0

Notes

Sortowanie przez wstawianie

Złożoność pamięciowa:

$$O(n)$$

Złożoność obliczeniowa:

Tablica - szukanie(n), przesuwanie (n), wykonywane n razy

$$O(n^2)$$

Lista - szukanie(n), przesuwanie (1), wykonywane n razy

$$O(n^2)$$

Notes

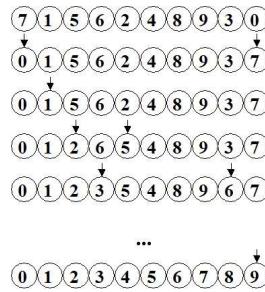
Sortowanie przez wybór

W ramach algorytmu sortowania przez wybór wykonujemy następujące czynności:

1. Wyszukujemy najmniejszy element w tablicy (na liście).
2. Zamieniamy go z elementem zerowym.
3. Operację tę powtarzamy dla kolejnych elementów.

Notes

Sortowanie przez wybór



Notes

Sortowanie przez wybór

Złożoność pamięciowa:

$$O(n)$$

Złożoność obliczeniowa:

Tablica i lista - szukanie(n), przestawianie (1), wykonywane n razy

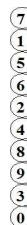
$$O(n^2)$$

Notes

Sortowanie bąbelkowe

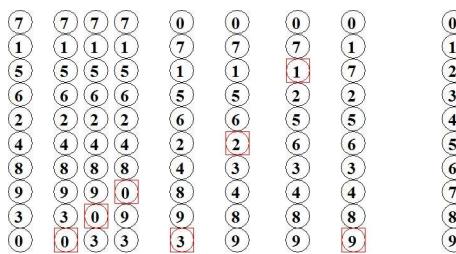
Algorytm sortowania bąbelkowego nosi swoją nazwę ze względu na podobieństwo do bąbelków pary uciekających w góre naczynia.

Zaczynając od ustawienia się w „najwyższym” punkcie tablicy przeszukujemy pozostałą część tablicy od dołu zawsze przesuwając ku górze mniejszy element z porównywanej pary.



Notes

Sortowanie bąbelkowe



Notes

Sortowanie bąbelkowe

Złożoność pamięciowa:

$$O(n)$$

Złożoność obliczeniowa:

$$O(n^2)$$

Duża wrażliwość.

Króliki i żółwie, możliwości optymalizacji.



Notes

Sortowanie koktajlowe

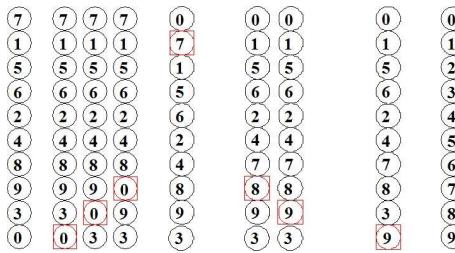
Sortowanie koktajlowe jest drobną modyfikacją sortowania bąbelkowego.

Różnica polega na tym, że kolejne etapy ustawiania odpowiedniego obiektu (o właściwym kluczu) na zadanej pozycji odbywają się w przeciwnych kierunkach



Notes

Sortowanie koktajlowe



Algorytmy sortowania

- Sortowanie w czasie kwadratowym
 - Cocktailsort

Sortowanie koktajlowe

Złożoność pamięciowa

$O(n)$

Złożoność obliczeniowa:

$O(n^2)$



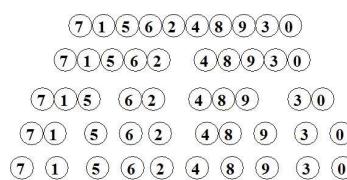
Sortowanie przez łączenie

Jest to metoda rekurencyjna, bazująca na rekurencyjnych podziałach sortowanej tablicy

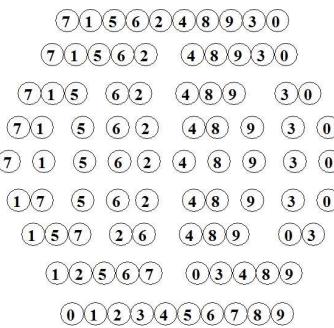
Jej wykonanie polega na:

- Dzieleniu tablicy na mniejsze tablice rozmiarów $\lfloor \frac{n}{2} \rfloor$ i $\lceil \frac{n}{2} \rceil$
 - Scalaniu już posortowanych mniejszych tablic

Sortowanie przez łączenie



Sortowanie przez łączenie



Notes

Sortowanie przez łączenie

Złożoność pamięciowa:

$$O(n)$$

Notes

Sortowanie stogowe

W sortowaniu stogowym wykorzystujemy charakterystyczną cechę tego drzewa binarnego - rodzic jest zawsze większy od potomka.

Proces sortowania składa się z dwóch etapów:

- ▶ Uformowanie kopca.
- ▶ Zdejmowanie z kopca i jego równoczesne naprawianie.

Notes

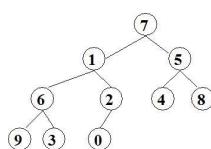
Tworzenie kopca

7 1 5 6 2 4 8 9 3 0

Numeracja od 1.

Implementacja tablicowa
Kopiec zupełny

numer potomka: i numer
rodzica: $[i/2]$ numer

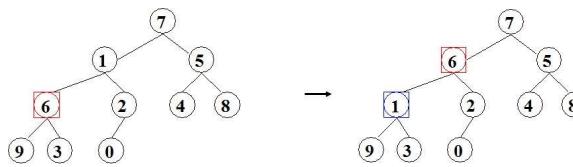


rodzica: i numer potomka:
 $2i, 2i + 1$

Notes

Tworzenie i formowanie kopca

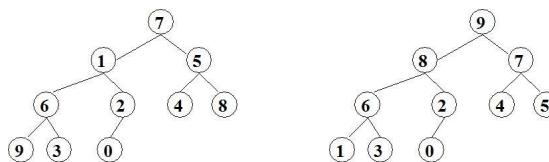
W procedurze formowania kopca zaczynając od korzenia i schodząc do dół sprawdzamy, czy zawsze dziecko jest mniejsze od rodzica, jeśli nie, zamieniamy je miejscami wracając jednocześnie do korzenia.



Notes

Tworzenie i formowanie kopca

Ostateczna postać kopca utworzonego z danych wejściowych:

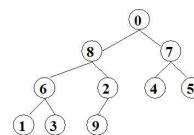


Postać nie jest unikalna i może zależeć od konkretnej realizacji algorytmu formowania.

Notes

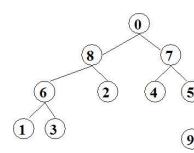
Zdejmowanie z kopca

Zamiana korzenia i elementu ostatniego



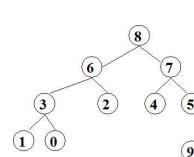
Notes

Zdjęcie z kopca elementu ostatniego i zapisanie go jako ostatniego w tablicy



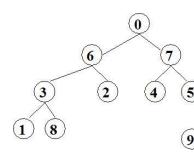
Zdejmowanie z kopca

Powtórnego formowania kopca:
schodząc od korzenia
zastępowanie rodzica
największym z potomków
(o ile są większe)



Notes

I jeszcze raz,
aż kopiec będzie pusty
(zamiana)



Notes

Zdejmowanie z kopca



Notes

Sortowanie stogowe

Złożoność pamięciowa:

$$O(n)$$

Złożoność obliczeniowa:

$$O(n \lg(n))$$

Taka sama jest złożoność operacji dodawania i właściwego sortowania.

Notes

Sortowanie szybkie

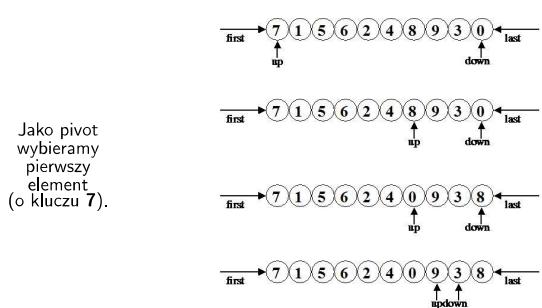
Sortowanie szybkie używa techniki „dziel i zwyciężaj”.

Wykonywane jest metodą rekurencyjną.

Podstawowym problemem jest podział struktury na dwie mniejsze, przy czym wybieramy pewien element (pivot) i z jednej jego strony grupujemy elementy mniejsze od niego, z drugiej większe.

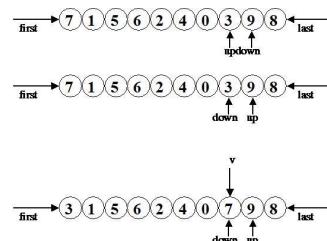
Notes

Przebieg



Notes

Przebieg



W efekcie udało nam się podzielić tablicę na dwie mniejsze

3 1 5 6 2 4 0 7 9 8

Notes

Sortowanie szybkie

Złożoność pamięciowa:

$$O(n)$$

Notes

Złożoność obliczeniowa:

przypadek najgorszy: $O(n^2)$
przypadek najlepszy: $O(n \lg(n))$
przypadek przeciętny: $O(2n \ln(n)) = O(1.39n \lg(n))$

Duża wrażliwość.

Notes

Algorytm Hoare (szukanie n-tej liczby)

Spróbujmy znaleźć medianę zbioru nieuporządkowanego

7 1 5 6 2 4 8 9 3 0

Po pierwszym podziale tablicy mamy

3 1 5 6 2 4 0 7 9 8

Od tej chwili zajmujemy się tylko pierwszym podziobrem

2 1 0 3 6 4 5

Notes

Sortowanie Shella

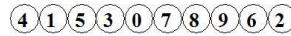
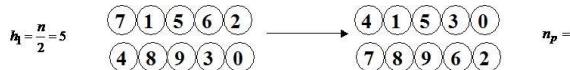
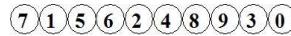
Sortowanie Shella jest algorytmem efektywnym dla danych, co do których mamy wiedzę o ich wstępny, częściowym uporządkowaniu

W algorytmie Shella:

1. Wybieramy ze zbioru elementy odległe od siebie o pewien skok i sortujemy je inną techniką.
2. Zmniejszamy skok, aż do wartości 1.

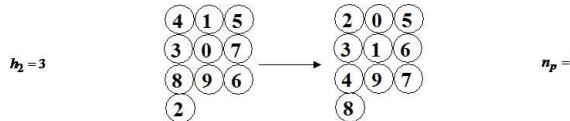
Notes

Sortowanie Shella



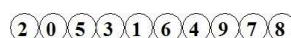
Notes

Sortowanie Shella



Notes

Sortowanie Shella



Notes

Sortowanie Shella

Złożoność pamięciowa:

$$O(n)$$

Złożoność obliczeniowa:

$$O(n^2)$$

Notes

- ▶ Czynnikiem krytycznym jest sposób określenia ciągu skoków h_i .
- ▶ Algorytm sortowania "lokalnego" powinien być wykonywany w miejscu i zapewniać małą liczbę przestawień

Sortowanie pozycyjne

W algorytmie sortowania pozycyjnego dokonujemy szeregu sortowań względem różnych kluczów.

- ▶ Nie dokonujemy porównań całych kluczów/obiektów, a tylko pewnych wartości występujących na określonych pozycjach.
- ▶ Sortowania na poszczególnych pozycjach muszą być wykonywane algorytmem stabilnym.

Notes

Sortowanie pozycyjne

7 3	7 2 0	4 1 6	6 8
7 2 0	3 8 1	7 2 0	7 3
1 3 6	5 3 2	5 3 2	1 3 6
1 8 5	7 3	1 3 6	1 8 5
9 9 6	1 8 5	6 8	3 8 1
6 8	7 9 5	7 3	4 1 6
5 3 2	1 3 6	3 8 1	5 3 2
4 1 6	9 9 6	1 8 5	7 2 0
3 8 1	4 1 6	7 9 5	7 9 5
7 9 5	6 8	9 9 6	9 9 6

Notes

Sortowanie pozycyjne

Złożoność pamięciowa:

$$O(n) \dots O(n+k)$$

Notes

Złożoność obliczeniowa:

$$O(n) \dots O(d(n+k)) \dots O(n^2)$$

Sortowanie przez zliczanie

Algorytm działa tylko dla kluczów enumerowanych ze zdefiniowaną operacją " $<$ ".

W algorytmie sortowania przez zliczanie:

1. Przeszukujemy całą tablicę (listę) wyszukując ilość wystąpień danego elementu.
2. Nową tablicę (listę) tworzymy w oparciu o pomocniczą tablicę ilości wystąpień.

Notes

Sortowanie przez zliczanie

7 1 5 6 3 4 8 4 3 0

0 - 1
1 - 1
2 - 0
3 - 2
4 - 2
5 - 1
6 - 1
7 - 1
8 - 1
9 - 0

0 1 3 3 4 4 5 6 7 8

Notes

Sortowanie przez zliczanie

Złożoność pamięciowa:

$$O(n + n_v)$$

Notes

Inne algorytmy sortujące

$O(n)$	Bucket sort
$O(n \lg(n))$	Bitonic Merge sort, Avl sort, B Sort
$O(n \lg(n))..O(n^2)$	Shear sort, Comb sort, Bitonic sort
$O(n^2)$	Shaker sort, Ripple sort, Oct sort, Min sort, Siple sort, Plasel sort, Jump sort, Partit sort
$> O(n^2)$	Trippel sort
???	Bogo sort

Notes

Notes

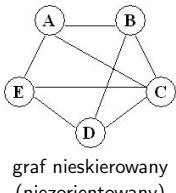
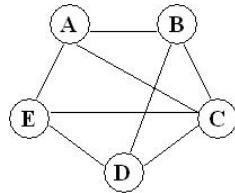
Notes

Grafy

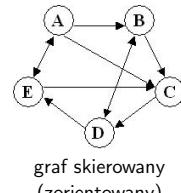
Tomasz M. Gwizdałła

Definicja

Grafem nazywamy strukturę $G = (V, E)$ składającą się z węzłów (wierzchołków, oznaczanych przez V) wzajemnie połączonych za pomocą krawędzi (oznaczanych przez E).



graf nieskierowany (nieorientowany)



graf skierowany (orientowany)

Krawędź - para wierzchołków grafu.

krawędzie łączą wierzchołki w obu kierunkach

krawędzie mogą łączyć wierzchołki tylko w jednym kierunku

- ▶ **Rząd grafu** - liczba wierzchołków grafu
- ▶ **Rozmiar grafu** - liczba krawędzi grafu
- ▶ Jeśli krawędź łączy dwa wierzchołki to jest z nimi **incydentna** (krawędź incydentna może mieć w wierzchołku początek lub koniec)
- ▶ **Sąsiad** - dwa wierzchołki są sąsiadami, jeśli istnieje krawędź pomiędzy nimi
- ▶ **Pętla własna** - krawędź łącząca wierzchołek z samym sobą

Notes

Notes

► **Stopień wierzchołka**

- ▶ w grafie nieskierowanym - liczba jego sąsiadów
- ▶ w grafie skierowanym - rozróżniamy stopień wejściowy (liczbę krawędzi, dla których dany wierzchołek jest pierwszym w parze) i wyjściowy (liczbę krawędzi, dla których dany wierzchołek jest drugim w parze)

► **Podgraf grafu G** - graf, którego wierzchołki stanowią podzbiór zbioru wierzchołków G (**nadgrafu**), a krawędzie podzbiór zbioru krawędzi G zawierający wszystkie krawędzie incydentne do podzbioru wierzchołków

► **Graf pełny** - graf, którego każdy wierzchołek jest połączony krawędzią z każdym innym (K_n)

► **Klika** - pełny podgraf grafu

Notes

► **Acentryczność wierzchołka** - maksymalna z odległości tego wierzchołka do innych wierzchołków grafu (eccentricity, ekscentyczność)

► **Promień grafu** - najmniejsza acentryczność wierzchołka wśród wszystkich wierzchołków grafu

► **Średnica grafu** - największa acentryczność wierzchołka wśród wszystkich wierzchołków grafu.

Notes

► **Ścieżka** - ciąg wierzchołków, w którym każdy wierzchołek jest sąsiadem zarówno poprzedniego, jak i następnego

► **Ścieżka prosta** - ścieżka, w której żaden z wierzchołków nie powtarza się

► Jeśli od każdego wierzchołka grafu istnieje ścieżka do dowolnego innego, to graf jest **spójny**

► **Cykł** Jest to ścieżka, która zaczyna się i kończy w tym samym wierzchołku (graf cykliczny/acykliczny)

► **Drzewo** - spójny graf acykliczny

Notes

► **Graf planarny** - graf, który można przedstawić (dla którego istnieje graf izomorficzny) na płaszczyźnie tak, by żadne dwie krawędzie się nie przecinają

► **Graf płaski** - izomorficzne przedstawienie grafu takie, że żadne dwie krawędzie się nie przecinają

► **Graf dwudzielny** - graf, który może być podzielony na dwa podgrafia takie, że nie istnieje żadna krawędź łącząca dwa wierzchołki tego samego podgrafa

► **Klika dwudzielna** - graf dwudzielny taki, że pomiędzy wszystkimi parami wierzchołków należących do różnych podgrafów istnieje krawędź ($K_{n,m}$)

► W grafie **planarnym** nie może występować podgraf homeomorficzny z K_5 lub $K_{3,3}$

Notes

- ▶ **Graf r -regularny** - graf, w którym każdy wierzchołek ma taki sam stopień (r)
- ▶ **Krawędzie równoległe (wielokrotne)** - krawędzie łączące te same pary wierzchołków
- ▶ **Krawędzie sąsiednie** - krawędzie kończące się w jednym wierzchołku
- ▶ **Graf prosty** - to graf bez pętli własnych i krawędzi równoległych

Notes

- ▶ **Graf ważony** - graf, w którym z każdą krawędzią związana jest pewna liczba, zwana wagą (długością)
- ▶ **Niezmienik grafu** - to liczba lub ciąg liczb, który zależy tylko od struktury grafu a nie od sposobu jego poetykietowania (np. liczba wierzchołków, liczba krawędzi)
- ▶ **Liczba chromatyczna grafu** - to najmniejsza liczba kolorów potrzebnych do pokolorowania wierzchołków grafu tak, by żadne dwa przyległe wierzchołki nie były tego samego koloru

Notes

Twierdzenie o czterech kolorach

Dla każdego skończonego grafu planarnego możliwe jest przypisanie każdemu z jego wierzchołków jednej z czterech liczb 1, 2, 3 i 4 w taki sposób, aby żadne sąsiednie wierzchołki nie miały przyporządkowanej tej samej liczby
Dowolną mapę polityczną na płaszczyźnie lub sferze można zabarwić czterema kolorami tak, aby każde dwa kraje mające wspólną granicę (a nie tylko wspólny wierzchołek) miały inne kolory

Dowód:
Appel and W. Haken - 1976 (prawie 2000 przypadków szczególnych)
N. Robertson, D. Sanders, P. Seymour, R. Thomas -1994

Notes

Macierz sąsiedztwa

Niech n_V będzie liczbą wierzchołków, n_E - krawędzi Budujemy tablicę $n_V \times n_V$ wypełniając ją zerami, jeśli wierzchołki nie są sąsiadami, jedynkami, jeśli są

	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	1	0
C	1	1	0	1	1
D	0	1	1	0	1
E	1	0	1	1	0

	A	B	C	D	E
A	0	1	1	0	1
B	0	0	1	1	0
C	0	0	0	1	0
D	0	1	0	0	1
E	1	0	1	0	0

Złożoność pamięciowa - $O(n_V^2)$

Notes

Lista incydencji

Dla każdego wierzchołka tworzymy listę, na której przechowujemy zbiór wierzchołków z nim połączonych

A: B, C, E
 B: A, C, D
 C: A, B, D, E
 D: B, C, E
 E: A, C, D

A: B, C, E
 B: C, D
 C: D
 D: B, E
 E: A, C

Złożoność pamięciowa - $O(n_V + n_E)$

Notes

Lista krawędzi

Lista, na której przechowujemy wszystkie krawędzie

A-B, A-C, A-E,
 B-A, B-C, B-D,
 C-A, C-B, C-D,
 C-E, D-B, D-C,
 D-E, E-A, E-C
 E-D

A-B, A-C,
 A-E, B-C,
 B-D, C-D,
 D-B, D-E,
 E-A, E-C

Złożoność pamięciowa - $O(n_E)$

Notes

Macierz incydencji

W wierszach macierzy zapisujemy krawędzie, w kolumnach wierzchołki. Wypełniamy wartością $\{-1, 1, 2\}$ (pętla własna)

	A	B	C	D	E
A-B	-1	1	0	0	0
A-C	-1	0	1	0	0
A-E	-1	0	0	0	0
B-A	1	-1	0	0	0
E-A	1	0	0	0	-1

Złożoność pamięciowa - $O(n_V * n_E)$

Notes

DFS

Depth First Search - przeszukiwanie grafu w głęb

```
void DFS(long i)
{
    if(!node[i].visited) {
        node[i].visited=true;
        for(j=1; j<node[i].next_number;j++) {
            DFS(node[i].next[j]);
        }
    }
}
```

Kolejność odwiedzania wierzchołków: A, B, C, D, E

Notes

BFS

Breadth First Search - przeszukiwanie grafu wszerz

```
void BFS() {  
    node.list nl;  
    nl.Add(start);  
    while(!nl.empty()) {  
        nl[current].visited = true;  
        for(j=1; j<node[current].next_number;j++)  
            nl.Add(node[current].next[j]);  
        nl.Remove(current);  
    }  
}
```

Kolejność odwiedzania wierzchołków: A, B, C, E, D

Notes

Najkrótsza droga

Szukanie najkrótszej drogi od zadanego wierzchołka do pozostały - bazujące na BFS

```
void shortest_path() {  
    node.list nl;  
    nl.Add(start);  
    while(!nl.empty()) {  
        nl[current].visited = true;  
        for(j=1; j<node[current].next_number;j++) {  
            nl.Add(node[current].next[j]);  
            node[i].next[j].distance = node[i].distance+1;  
        }  
        nl.Remove(current);  
    }  
}
```

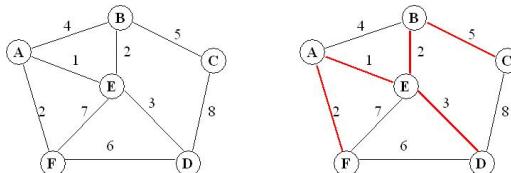
Notes

Minimalne drzewo rozpinające

MST (Minimum Spanning Tree)

Rozważmy graf nieskierowany, spójny, ważony.

Podzbiór T zbioru krawędzi E, łączący wszystkie wierzchołki oraz taki, że suma wag krawędzi zbioru T jest najmniejsza nazywamy minimalnym drzewem rozpinającym.



Notes

Algorytm Kruskala

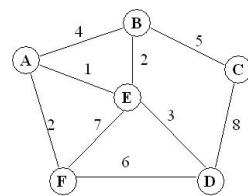
Algorytm Kruskala znajdowania MST jest algorytmem zachłannym

- ▶ Posortuj wszystkie krawędzie w rosnącym porządku wag
- ▶ Wybieraj po kolej krawędzie z tak posortowanej listy
- ▶ Krawędź może:
 - ▶ łączyć wierzchołki nienależące do żadnego poddrzewa - tworzymy nowe poddrzewo
 - ▶ łączyć wierzchołki, z których tylko jeden należy do pewnego poddrzewa - powiększamy to poddrzewo
 - ▶ łączyć wierzchołki należące do dwóch różnych poddrzew - scalamy je
 - ▶ łączyć wierzchołki należące do tego samego poddrzewa - nie bierzemy jej pod uwagę
- ▶ Kiedy wszystkie węzły należą do jednego drzewa, zatrzymaj

Notes

Algorytm Kruskala

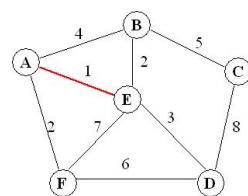
AE: 1
AF: 2
BE: 2
DE: 3
AB: 4
BC: 5
FD: 6
EF: 7
CD: 8



Notes

Algorytm Kruskala

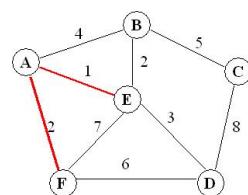
AE: 1
AF: 2
BE: 2
DE: 3
AB: 4
BC: 5
FD: 6
EF: 7
CD: 8



Notes

Algorytm Kruskala

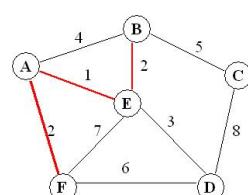
AE: 1
AF: 2
BE: 2
DE: 3
AB: 4
BC: 5
FD: 6
EF: 7
CD: 8



Notes

Algorytm Kruskala

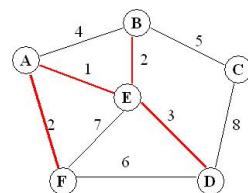
AE: 1
AF: 2
BE: 2
DE: 3
AB: 4
BC: 5
FD: 6
EF: 7
CD: 8



Notes

Algorytm Kruskala

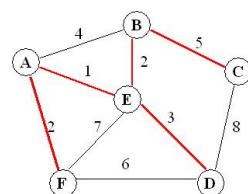
AE: 1
AF: 2
BE: 2
DE: 3
AB: 4
BC: 5
FD: 6
EF: 7
CD: 8



Notes

Algorytm Kruskala

AE: 1
AF: 2
BE: 2
DE: 3
AB: 4
BC: 5
FD: 6
EF: 7
CD: 8



Notes

Algorytm Prima

Algorytm Prima znajdowania MST jest algorytmem zachłannym

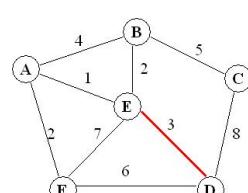
- ▶ Wybierz dowolny wierzchołek
- ▶ Dla istniejącego drzewa powtarzaj
 - ▶ Utwórz posortowaną ze względu na wagę listę krawędzi wychodzących z wierzchołków drzewa
 - ▶ Dodaj do drzewa gałąź o najmniejszej wadze (o ile jest ona incydentna do wierzchołka, którego nie ma jeszcze w drzewie)

Notes

Algorytm Prima

Wybierzmy jako początkowy wierzchołek D

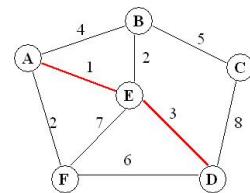
DE: 3
DF: 6
DC: 8



Notes

Algorytm Prima

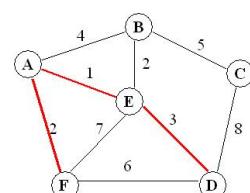
EA: 1
EB: 2
DF: 6
EF: 7
DC: 8



Notes

Algorytm Prima

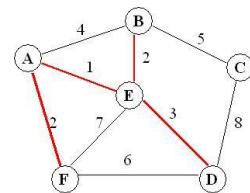
AF: 2
EB: 2
AB: 4
DF: 6
EF: 7
DC: 8



Notes

Algorytm Prima

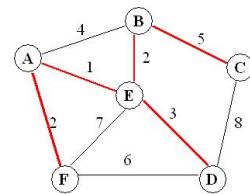
EB: 2
AB: 4
DF: 6
EF: 7
DC: 8



Notes

Algorytm Prima

AB: 4
BC: 5
DF: 6
EF: 7
DC: 8



Notes

Algoritmy wyznaczania najmniejszej odległości od ustalonego wierzchołka do wszystkich pozostałych w grafie skierowanym

- ▶ Algorytm Dijkstry.
- ▶ Algorytm Forda-Bellmana
- ▶ Algorytm Floyda-Warshalla
- ▶ A* (A-star, A-gwiazdka)

Rozważmy graf skierowany, spójny, ważony

Notes

Algorytm Dijkstry

Algorytm Dijkstry jest algorytmem zachłannym

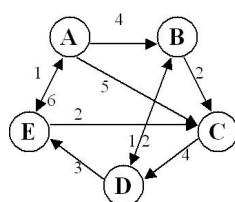
- ▶ Wybierz wierzchołek, utwórz tablicę odległości do pozostałych wierzchołków
- ▶ Wybierz wierzchołek o najmniejszej odległości
- ▶ Zmodyfikuj tablicę odległości uwzględniając wagę krawędzi wychodzących z dodanego wierzchołka
- ▶ Powtarzaj tak długo, aż wszystkie wierzchołki będą uwzględnione

Notes

Algorytm Dijkstry

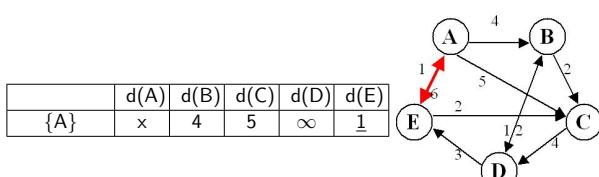
Graf nie może zawierać krawędzi o wagach ujemnych.

Jako początkowy wybierzmy wierzchołek A



Notes

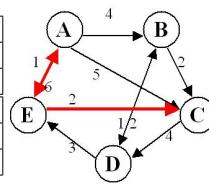
Algorytm Dijkstry



Notes

Algorytm Dijkstry

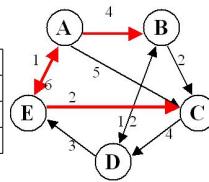
	d(A)	d(B)	d(C)	d(D)	d(E)
{A}	x	4	5	∞	<u>1</u>
{A,E}	x	4	<u>3</u>	∞	x
d(A-w)	x	4	5	∞	1
d(A-E-w)	x	∞	3	∞	x



Notes

Algorytm Dijkstry

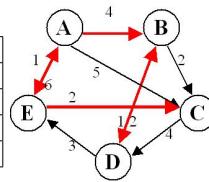
	d(A)	d(B)	d(C)	d(D)	d(E)
{A}	x	4	5	∞	<u>1</u>
{A,E}	x	4	<u>3</u>	∞	x
{A,C,E}	x	<u>4</u>	x	7	x



Notes

Algorytm Dijkstry

	d(A)	d(B)	d(C)	d(D)	d(E)
{A}	x	4	5	∞	<u>1</u>
{A,E}	x	4	<u>3</u>	∞	x
{A,C,E}	x	<u>4</u>	x	7	x
{A,B,C,E}	x	x	x	<u>5</u>	x



Notes

Algorytm Forda-Bellmana

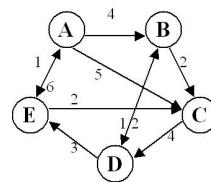
- ▶ Graf nie może zawierać cykli o długości ujemnej
- ▶ Algorytm Forda Bellmana jest algorymem zachłannym
- ▶ Wychodząc od początkowego wierzchołka analizujemy długość dróg
- ▶ Są dwie wersje algorytmu o różnych złożonościach
 - ▶ $O(n_V * n_E)$
 - ▶ $O(n_V^3)$

Notes

Algorytm Forda-Bellmana

Tworzymy tablicę wag

	A	B	C	D	E
A	0	4	5	∞	1
B	∞	0	2	1	∞
C	∞	∞	0	4	∞
D	∞	2	∞	0	3
E	6	∞	2	∞	0



Notes

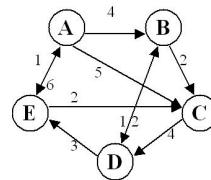
Algorytm Forda-Bellmana

Analizujemy tablicę odległości od wierzchołka A

	A	B	C	D	E
1	0	4	5	∞	1

AB(4), AC(5), AE(1), BC(2), BD(1), CD(4), DB(2), DE(3), EA(6), EC(2)

	A	B	C	D	E
2	0	4	3	5	1



Notes

Algorytm Floyda-Warshalla

- ▶ Algorytm ten służy do wyznaczania najmniejszej odległości pomiędzy wszystkimi parami wierzchołków
- ▶ Stosuje się go do grafów skierowanych bez cykli o długości ujemnej
- ▶ Opiera się na technice podobnej do algorytmu Forda-Bellmana

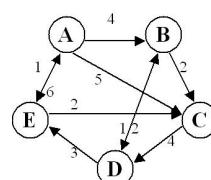
$$\begin{aligned} \triangleright d_{ij}^0 &= w_{ij} \\ \triangleright d_{ij}^{k+1} &= \min(d_{ij}^k, d_{i,k+1}^k + d_{k+1,j}^k) \end{aligned}$$

Notes

Algorytm Floyda-Warshalla

Rozważmy znany graf

W algorytmie Floyda-Warshalla wychodzimy z tablicy wag, jako najmniejszych odległości w przybliżeniu "zerowym", po czym modyfikujemy ją dla kolejnych wierzchołków w oparciu o dane dla wierzchołków rozpatrywanych wcześniej



Notes

Algorytm Floyda-Warshalla

	A	B	C	D	E
A	0	4	5	∞	1
B	∞	0	2	1	∞
C	∞	∞	0	4	∞
D	∞	2	∞	0	3
E	6	10	2	∞	0

	A	B	C	D	E
A	0	4	5	5	1
B	∞	0	2	1	∞
C	∞	∞	0	4	∞
D	∞	2	4	0	3
E	6	10	2	11	0

1

	A	B	C	D	E
A	0	4	5	5	1
B	∞	0	2	1	∞
C	∞	∞	0	4	∞
D	∞	2	4	0	3
E	6	10	2	11	0

Notes

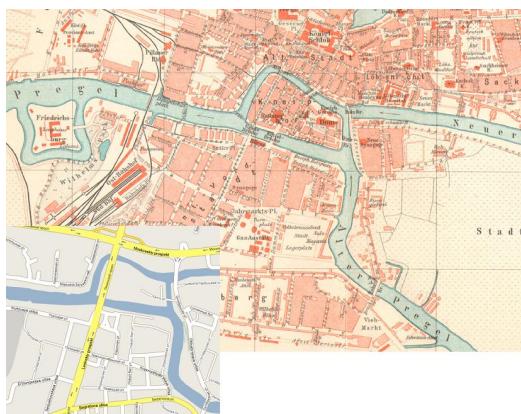
Algorytm Floyda-Warshalla

	A	B	C	D	E
A	0	4	5	5	1
B	∞	0	2	1	4
C	∞	6	0	4	7
D	∞	2	4	0	3
E	6	8	2	6	0

	A	B	C	D	E
A	0	4	3	5	1
B	10	0	2	1	4
C	13	6	0	4	7
D	9	2	4	0	3
E	6	8	2	6	0

Notes

Zagadka. Co to za miasto?



Notes

Zagadnienie Eulera

Zagadnienie mostów królewieckich

Brzegi Pregoły i dwie wyspy na niej połączone są przy pomocy siedmiu mostów. Czy można przejść przez wszystkie mosty tak, aby każdym przejść tylko raz?



Notes

- ▶ Cykl Eulera - cykl zawierający każdą krawędź dokładnie raz.
- ▶ Łąćuch Eulera - ścieżka zawierająca każdą krawędź dokładnie raz

Zagadnienie Eulera

Warunki istnienia cyklu Eulera

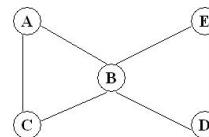
- ▶ Graf musi być spójny
- ▶ W grafie nieskierowanym liczba krawędzi incydentnych do każdego wierzchołka musi być parzysta
- ▶ W grafie skierowanym liczba krawędzi wchodzących musi być równa liczbie gałęzi wychodzących (i oczywiście parzysta)

Notes

Zagadnienie Eulera

Rozważmy graf

Rozpoczniemy analizę od wierzchołka A



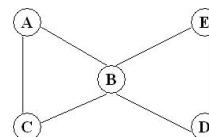
W ramach algorytmu znajdowania cyklu Eulera należy

- ▶ Dopóki istnieje możliwość osiągnięcia innych wierzchołków dodawać je na stos, usuwając krawędzie między nimi
- ▶ Jeśli takiej możliwości nie ma przekładać wierzchołki ze stosu do cyklu tak dugo, aż się pojawi

Notes

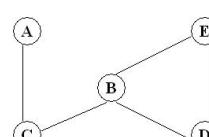
Zagadnienie Eulera

Stos:
A



Cykl:
-

Stos:
A,B

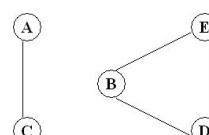


Cykl:
-

Notes

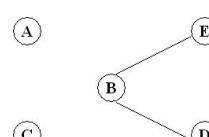
Zagadnienie Eulera

Stos:
A,B,C



Cykl:
-

Stos:
A,B,C,A

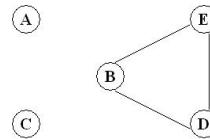


Cykl:
-

Notes

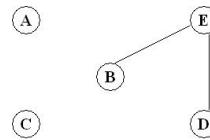
Zagadnienie Eulera

Stos:
A,B



Cykl:
A,C

Stos:
A,B,D

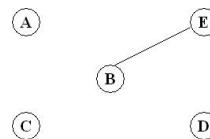


Cykl:
A,C

Notes

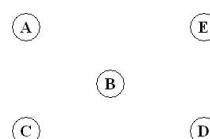
Zagadnienie Eulera

Stos:
A,B,D,E



Cykl:
A,C

Stos:
A,B,D,E,B



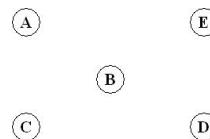
Cykl:
A,C

Notes

Zagadnienie Eulera

Ostatecznie

Stos:



Cykl:
A,C,B,E,D,B,A

Notes

Cykł Hamiltona

Cykł Hamiltona - cykl zawierający każdy wierzchołek dokładnie jeden raz.

Ścieżka (łańcuch) Hamiltona - ścieżka zawierająca każdy wierzchołek dokładnie jeden raz

Istnieje wiele kryteriów określających warunek dostateczny tego, aby graf był hamiltonowskim (Ore'go, Dirac'a, Chvatal'a, ilości krawędzi).

Notes

Cykł Hamiltona, zagadnienie komiwojażera

Komiwojażer musi odwiedzić n miast, każde jeden raz i wrócić do domu, z którego wyjechał, przebywając przy tym najkrótszą z możliwych dróg.

Problem ten oznacza konieczność znalezienia w grafie reprezentującym położenia i odległości miast cyklu Hamiltona charakteryzującego się najkrótszą sumą wag.

Problem ma złożoność obliczeniową $O(n!)$. Jest chyba naj słynniejszym problemem NP trudnym.

Notes

Notes

Notes

Notes

Notes

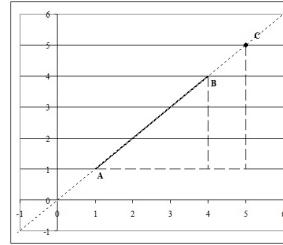
Podstawy geometrii obliczeniowej

Tomasz M. Gwizdałła

Współliniowość punktów

$$\frac{y_C - y_A}{x_C - x_A} = \frac{y_B - y_A}{x_B - x_A}$$

$$x_A(y_B - y_C) + x_B(y_C - y_A) + x_C(y_A - y_B) = 0$$



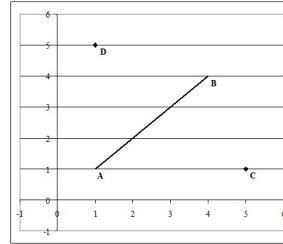
$$\det(A, B, C) = \begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix} = 0$$

Lewo i prawo

Narysujmy odcinek pomiędzy punktami $A(1, 1)$ i $B(4, 4)$

oraz punkty
 $C(5, 1)$ i $D(1, 5)$

Jak można określić położenie punktów C i D względem odcinka AB ?



Lewo i prawo

$$\det(A, B, C) = \begin{vmatrix} 1 & 1 & 1 \\ 4 & 4 & 1 \\ 5 & 1 & 1 \end{vmatrix} = -12$$

$$\det(B, A, C) = \begin{vmatrix} 4 & 4 & 1 \\ 1 & 1 & 1 \\ 5 & 1 & 1 \end{vmatrix} = 12$$

$$\det(A, B, D) = \begin{vmatrix} 1 & 1 & 1 \\ 4 & 4 & 1 \\ 1 & 5 & 1 \end{vmatrix} = 12$$

$$\det(B, A, D) = \begin{vmatrix} 4 & 4 & 1 \\ 1 & 1 & 1 \\ 1 & 5 & 1 \end{vmatrix} = -12$$

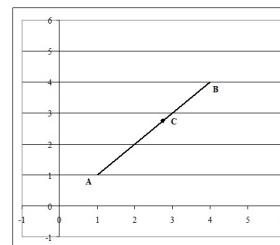
Jeśli $\det(A, B, C) < 0$ to punkt C jest z prawej strony odcinka AB
 Jeśli $\det(A, B, C) > 0$ to punkt C jest z lewej strony odcinka AB

Odcinka, czy półprostej lub wektora?

Notes

Współliniowość punktów

Jak jednoznacznie określić, czy punkt C należy do odcinka AB ?



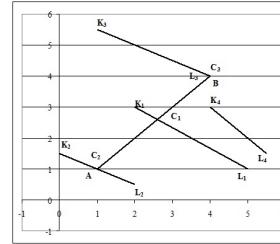
Aby punkt C należał do odcinka AB potrzeba i wystarcza, aby punkty te były współliniowe, oraz

$$\min(x_A, x_B) \leq x_C \leq \max(x_A, x_B) \wedge \min(y_A, y_B) \leq y_C \leq \max(y_A, y_B)$$

Przecinanie się odcinków

Jak widać odcinek AB jest przecinany przez odcinki:

$$\begin{aligned} & K_1 L_1 \\ & K_2 L_2 \\ & K_3 L_3 \end{aligned}$$



Jak zapisać regułę przecinania?

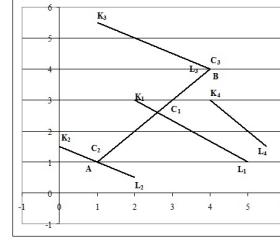
Przecinanie się odcinków

$$\begin{aligned} & \det(A, B, L_1) < 0, \det(A, B, K_1) > 0 \\ & \det(K_1, L_1, A) < 0, \det(K_1, L_1, B) > 0 \end{aligned}$$

$$\begin{aligned} & \det(A, B, L_2) < 0, \det(A, B, K_2) > 0 \\ & \det(K_2, L_2, A) = 0, \det(K_2, L_2, B) > 0 \end{aligned}$$

$$\begin{aligned} & \det(A, B, L_3) = 0, \det(A, B, K_3) > 0 \\ & \det(K_3, L_3, A) < 0, \det(K_3, L_3, B) = 0 \end{aligned}$$

$$\begin{aligned} & \det(A, B, L_4) < 0, \det(A, B, K_4) < 0 \\ & \det(K_4, L_4, A) < 0, \det(K_4, L_4, B) > 0 \end{aligned}$$



Przecinanie się odcinków

Warunek przecinania się odcinków można sformułować następująco:

Aby dwa odcinki się przecinały nie może mieć miejsca taka sytuacja, że oba krańce jednego z odcinków są po tej samej stronie drugiego.

Oba wyznaczniki nie mogą mieć tego samego znaku.

Notes

Notes

Notes

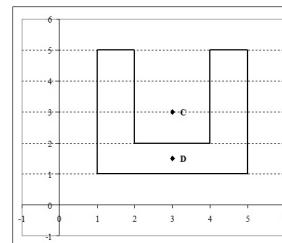
Notes

Przynależność do figury

Jak sprawdzić, że:

punkt D leży wewnątrz figury,

punkt C leży na zewnątrz figury?



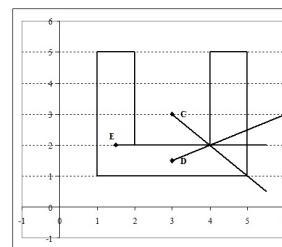
Notes

Przynależność do figury

Najprostszą metodą jest:

1. przeprowadzić półprostą od danego punktu do dowolnego punktu leżącego na pewno poza figurą

2. policzyć punkty przecięcia, z odcinkami będącymi bokami figury



Notes

Przecięcia w zbiorze punktów

Jak znaleźć wszystkie punkty przecięć odcinków należących do pewnego zbioru?

Metoda zachłanna zakłada sprawdzenie wszystkich możliwych par odcinków. Jej złożoność obliczeniowa jest oczywiście $O(n^2)$.

Algorytmem szybszym jest tzw. algorytm zmiataania.

Notes

Pojęcia

„Miotłą” jest prosta równoległa do osi OY poruszająca się w kierunku wzrostających wartości x .

Dla danego położenia miotły wyróżniamy trzy rodzaje odcinków:

- ▶ przetworzone (oba końce na lewo od „miotły”)
- ▶ aktywne (aktualnie przecinające „miotłę”)
- ▶ oczekujące (oba końce na prawo od „miotły”)

Notes

Struktury

Wykorzystujemy dwie struktury danych:

► X, Harmonogram zdarzeń

posortowany względem współrzędnej x zbiór punktów końcowych odcinków oraz punktów przecięć odcinków aktywnych, które były sąsiadami w strukturze Y

► Y, Struktura stanu

zbiór zawierający nazwy odcinków aktywnych uporządkowanych względem współrzędnej y punktu przecięcia z miotłą, często zbalansowane BST

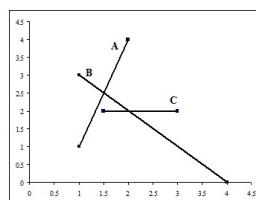
Notes

Przebieg

- W strukturze X umieść uporządkowane końce wszystkich odcinków
- Wybierz i usuń z X element minimalny. Jeśli jest on:
 - początkiem odcinka - dodajemy nazwę do Y; sprawdzamy przecięcia z sąsiadami w Y; jeśli takie istnieją umieszczamy współrzędne przecięć w X
 - końcem odcinka - usuwamy nazwę z Y
 - punktem przecięcia odcinków - zamieniamy położenia odcinków w Y; sprawdzamy przecięcia na prawo od „miotły” z nowymi sąsiadami; jeśli takie istnieją umieszczamy współrzędne przecięć w X
- Te operacje powtarzamy aż do opróżnienia X

Notes

Przykład



Rozważmy zbiór składający się z odcinków:

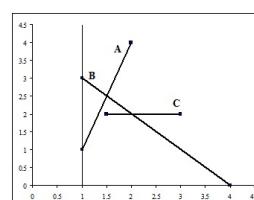
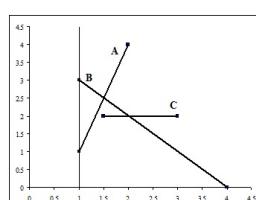
$$\begin{aligned}A &[(1, 1), (2, 4)] \\B &[(1, 3), (4, 0)] \\C &[(1.5, 2), (3, 2)]\end{aligned}$$

Punkty przecięcia

$$\begin{aligned}A \text{ z } B &(1.5, 2.5) \\B \text{ z } C &(2, 2)\end{aligned}$$

Notes

Przykład (1)



$$X: (1_{A,b}, 1_{B,b}, 1.5_{C,b}, 2_{A,e}, 3_{C,e}, 4_{B,e})$$

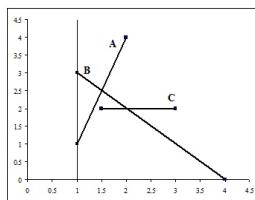
Y:

$$X: (1_{B,b}, 1.5_{C,b}, 2_{A,e}, 3_{C,e}, 4_{B,e})$$

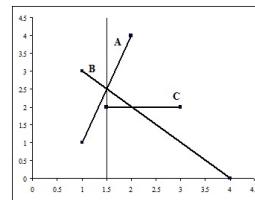
Y: A(1)

Notes

Przykład (2)



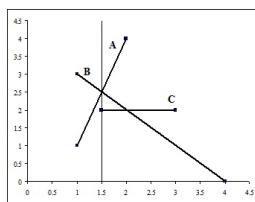
X: $(1.5_{C,b}, 2_{A,e}, 3_{C,e}, 4_{B,e})$
Y: $A(1), B(3)$
X: $(1.5_{AB}, 1.5_{C,b}, 2_{A,e}, 3_{C,e}, 4_{B,e})$



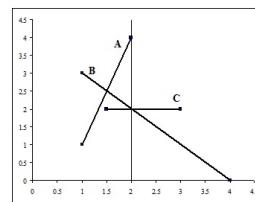
X: $(1.5_{C,b}, 2_{A,e}, 3_{C,e}, 4_{B,e})$
Y: $B(2.5), A(2.5)$

Notes

Przykład (3)



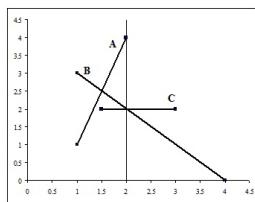
X: $(2_{A,e}, 3_{C,e}, 4_{B,e})$
Y: $C(2), B(2.5), A(2.5)$
X: $(2_{A,e}, 2_{BC}, 3_{C,e}, 4_{B,e})$



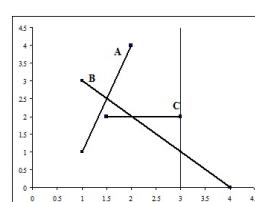
X: $(2_{BC}, 3_{C,e}, 4_{B,e})$
Y: $C(2), B(2.5)$

Notes

Przykład (4)



X: $(3_{C,e}, 4_{B,e})$
Y: $B(2), C(2)$

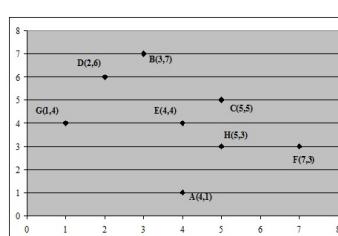


X: $(4_{B,e})$
Y: $B(2)$
 itd.

Notes

Cel

Procedura sortowania ze względu na kąt promienia wodzącego poprowadzonego do danego punktu jest procedurą pomocniczą, wykorzystywaną w innych zagadnieniach.

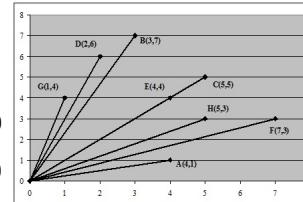


Notes

Algorytm

Możemy wykorzystać funkcje trygonometryczne lub uniknąć liczenia sum szeregów definiując funkcję według schematu

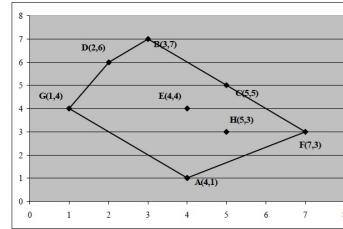
$$\alpha = \begin{cases} \frac{y_i}{d_i}, x_i \geq 0 \wedge y_i \geq 0 \\ 2 - \frac{y_i}{d_i}, x_i < 0 \wedge y_i > 0 \\ 2 + \frac{|y_i|}{d_i}, x_i \leq 0 \wedge y_i \leq 0 \\ 4 - \frac{|y_i|}{d_i}, x_i > 0 \wedge y_i < 0 \end{cases}$$



Wypukła otoczka

Wypukła otoczka (convex hull) pewnego zbioru punktów nazywamy najmniejszy wielokąt wypukły, taki, że wszystkie punkty należące do zbioru, należą do tego wielokąta.

Uwaga! Brzeg też należy do wielokąta.



Wypukłą otoczkę zadanego zbioru tworzą punkty:

A, F, C, B, D, G

Algorytm

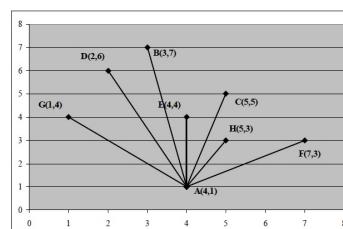
Algorytm znajdowania wypukłej otoczki nosi nazwę algorytmu Grahama.

- ▶ Wybieramy punkt, który na pewno będzie należał do wypukłej otoczki.
Najczęściej jest to punkt o najmniejszej lub największej wartości wybranej współrzędnej.
Jeśli takich punktów jest więcej bierzemy pod uwagę drugą współrzędną
- ▶ Sortujemy względem kątów promieni wodzących wyprowadzonych z tego punktu.
- ▶ Punkt wybrany, jako początkowy oraz dwa pierwsze na posortowanej liście zapisujemy do struktury przechowującej dane o otoczce.

Punkt początkowy

Punkt A na pewno będzie należał do wypukłej otoczki.

Kolejność punktów posortowanych, to: F, H, C, E, B, D, G.



Notes

Notes

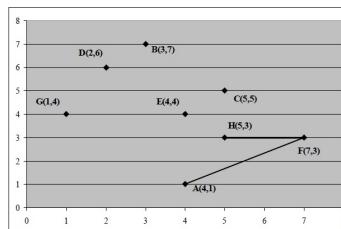
Notes

Notes

Kolejne dwa punkty

WO: A, F, H

Wiemy też, że do wypukłej otoczki będzie należał punkt F.

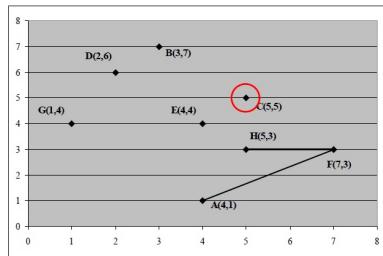


Notes

Powtarzamy

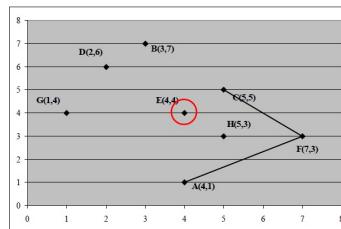
Podstawą algorytmu jest sprawdzenie, czy kolejny punkt znajduje się po prawej, czy po lewej stronie odcinka łączącego dwa ostatnie punktu w strukturze WO. Wyróżniamy dwie możliwości:

- ▶ Punkt znajduje się po lewej stronie odcinka lub jest z nim wspólniowy - kolejny punkt dodajemy do struktury WO
- ▶ Punkt znajduje się po prawej stronie odcinka - kolejny punkt zastępuje ostatni w strukturze WO



WO: A, F, C

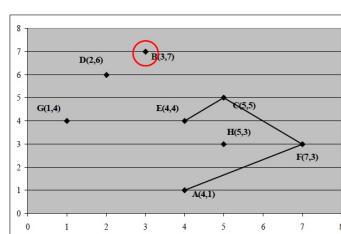
Notes



WO: A, F, C, E

Ten algorytm powtarzamy dla każdego punktu na posortowanej liście.

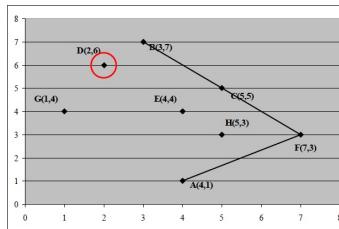
Notes



WO: A, F, C, E

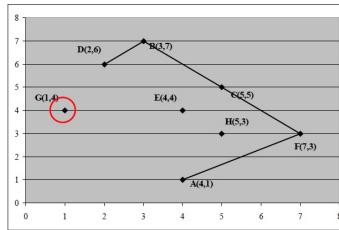
Notes

Notes



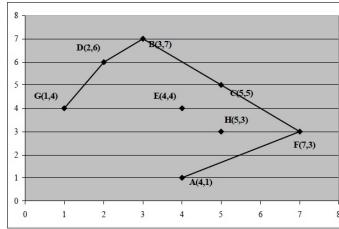
WO: A, F, C, B

Notes



WO: A, F, C, B, D

Notes



WO: A, F, C, B, D, G

Punkt G jest ostatnim analizowanym punktem.

Pojęcie

BSP – Binary Space Partition, Binarny Podział Przestrzeni

Drzewo BSP jest strukturą umożliwiającą określenie kolejności rysowania obiektów tak, aby obiekt znajdujący się dalej nie przekrywał obiektów bliższych.

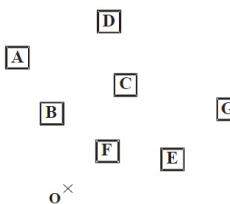
Innymi popularnymi metodami takiego porządkowania są metody typu:

depth sort
z-sort

Notes

Cel

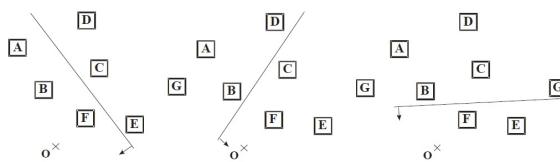
Naszym celem będzie uporządkowanie w odpowiedniej kolejności obiektów A-G, z punktu widzenia obserwatora O. Warto zwrócić uwagę, że obiekty nie są punktowe.



Notes

Obserwacja

Poprowadźmy kilka przykładowych prostych dzielących przestrzeń zawierającą obiekty na podprzestrzenie. Strzałka oznacza tę podprzestrzeń, w której znajduje się obserwator.

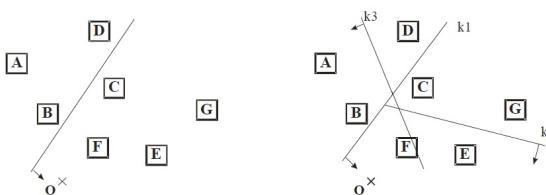


Zwróćmy uwagę, że nigdy nie wystąpi taka sytuacja, że obiekt znajdujący się w podprzestrzeni, w której nie ma obserwatora zasłoni obiekt z podprzestrzeni, w której obserwator jest.

Notes

Dalsze podziały

Wybierzmy jeden z przedstawionych podziałów i dokonujmy dalszych podziałów podprzestrzeni, zawsze na dwie. Oznaczajmy półproste symbolami $k[n]$. Zaznaczajmy też tę podprzestrzeń, w której znajduje się obserwator.

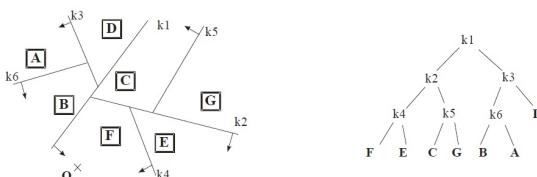


Notes

Ostateczny kształt drzewa

W oparciu o dokonany podział zbudujmy drzewo binarne według następującego schematu:

- ▶ w węzłach drzewa znajdują się prosta i półproste, przy czym relacja jest taka, że potomkami są te półproste, których początek leży na rodzicu.
- ▶ obiekty znajdują się w liściach.
- ▶ lewym potomkiem jest zawsze ten węzeł/liść, który znajduje się z tej samej strony półprostej, co obserwator (kwestia konwencji).



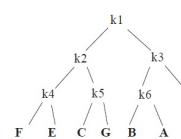
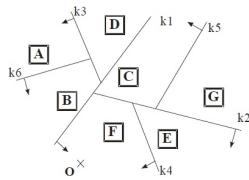
Notes

Rysowanie

Kolejność rysowania musi być taka, aby obiekty dalsze były rysowane wcześniej, czyli wybieramy zawsze najpierw prawe poddrzewa i obiekty.

Zatem kolejność rysowania jest następująca:

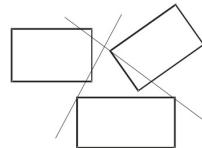
D, A, B, G, C, E, F



Notes

Problemy

- Obiekty o dużych rozmiarach, które trzeba dzielić

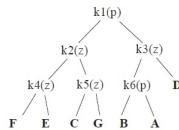
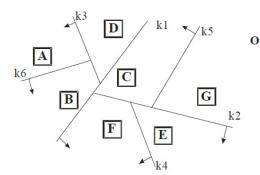


- Zmiana położenia obserwatora

Problemy

Załóżmy, że obserwator zmieni położenie.

Przeanalizujmy drzewo BSP, zapisując w k każdym węźle, czy obserwator jest w podprzestrzeni wskazywanej przez strzałkę (p), czy w drugiej (z).



Postępujmy teraz analogicznie, jak w poprzedniej sytuacji, czyli najpierw wybieramy to, co jest w dopełniającej podprzestrzeni

A B D F E C G

Notes

Notes

Wyszukiwanie wzorca String matching

Tomasz M. Gwizdałła

Podstawowe definicje

Zdefiniujmy dwa pojęcia

tekst	$T[0..n - 1]$	(text)	$ T = n$
wzorzec	$P[0..m - 1]$	(pattern)	$ P = m$

Elementy tablicy brane są z pewnego zbioru Σ zwanego alfabetem

Mówimy, że wzorzec występuje na pozycji s (z przesunięciem s), jeżeli

$$\begin{aligned} T[s..s + m - 1] &= P[0..m - 1] \\ T[s] &= P[0], \dots, T[s + m - 1] = P[m - 1] \end{aligned}$$

Mówimy też, że przesunięcie s jest przesunięciem poprawnym

Określenie problemu

Zagadnienie wyszukiwania wzorca polega na

znalezieniu
wszystkich poprawnych przesunięć
wzorca w tekście

Inne pojęcia

Niech Σ^* będzie zbiorem wszystkich łańcuchów o skończonej długości, które mogą być utworzone z alfabetu Σ

w jest prefixem łańcucha (tekstu) x , jeśli

$$w[x \iff \exists_{y \in \Sigma^*} x = wy]$$

w jest suffixem łańcucha (tekstu) x , jeśli

$$w]x \iff \exists_{y \in \Sigma^*} x = yw$$

Algorytm naiwny

W algorytmie naiwnym sprawdzamy znak po znaku, przesuwając wzorzec względem tekstu zawsze o 1

K A T A R A K T A
A R A K
K A T A R A K T A
A R A K

K A T A R A K T A
A R A K

K A T A R A K T A
A R A K
K A T A R A K T A
A R A K

K A T A R A K T A
A R A K

Notes

Algorytm naiwny

- Złożoność obliczeniowa

$$O(m * n)$$

Notes

- Algorytm charakteryzuje się dużą wrażliwością, zależną od rozmiaru alfabetu

Notes

Algorytm naiwny

a	8.91
i	8.21
o	7.75
e	7.66
y	3.76
u	2.5
ę	1.11
ą	0.99
ó	0.85

prawdopodobieństwo przypadkowego wyboru 2 takich samych znaków:
 ≈ 0.05

z	5.64	ł	1.82
n	5.52	b	1.47
r	4.69	g	1.42
w	4.65	h	1.08
s	4.32	ż	0.83
t	3.98	ś	0.66
c	3.96	ć	0.4
k	3.51	f	0.3
d	3.25	ń	0.2
p	3.13	q	0.14
m	2.8	ź	0.06
j	2.28	v	0.04
l	2.1	x	0.02

Zagadnienie

Notes

Idea algorytmu Karpa-Rabina jest przetransformowanie łańcucha znakowego do postaci liczbowej, po czym dokonywanie porównań na liczbach całkowitych.

Założenia

Założymy, że alfabet składa się tylko z czterech znaków

$$\Sigma = \{A, K, R, T\}$$

Przypiszmy każdemu znakowi jego indeks w takim alfabetie (zaczynając od jedynki) i przepiszmy zarówno tekst, jak i wzorzec używając znaków tych indeksów lub przejdźmy od razu do zapisu poniżej

KATARAKTA = "214131241"

ARAK = "1312"

Notes

Algorytm

Przedstawmy te łańcuchy znakowe, jako liczby w układzie dziesiętnym

$$t_s = \sum_{i=1}^m T[i + s] * 10^{m-i}$$
$$p = \sum_{i=1}^m P[i] * 10^{m-i}$$

Jak łatwo zauważyc warunkiem poprawności przesunięcia s jest równość:

$$t_s = p$$

Notes

Algorytm

Problemem, który trzeba rozwiązać jest duża ilość mnożeń w fazie obliczania obu wartości. Można ją oszacować

$$N_m = \frac{m(m+1)}{2}$$

Można ją zmniejszyć do ilości liniowej w zmiennej m poprzez

$$p = P[m] + 10 * (P[m - 1] + 10 * (...))$$

Notes

Algorytm

Trzeba też zmodyfikować sposób wyznaczania t_s . Można np. wyznaczać je rekurencyjnie, weźmy t_0 i t_1

$$t_0 = 2141 \quad t_1 = 1413$$
$$1413 = (2141 - 2000) * 10 + 3$$

$$t_1 = (t_0 - T[0] * 10^{m-1}) * 10 + T[4]$$

$$t_{s+1} = (t_s - T[s] * 10^{m-1}) * 10 + T[s + m]$$

Złożoność obliczeniowa operacji wyznaczania t_s jest teraz stała O(1)

Notes

Problemy

Notes

- Rozmiar alfabetu
- Zakres liczby całkowitej

Założymy teraz, że znaki z przykładu są wybrane z całego alfabetu łacińskiego, liczącego 26 znaków

Zarówno liczbę p , jak i t_s trzeba zapisać w układzie o podstawie równej rozmiarowi alfabetu. Oznaczmy tę wielkość przez d

$$t_{s+1} = (t_s - T[s + 1] * d^{m-1}) * d + T[s + m + 1]$$

Problemy

Notes

Aby rozwiązać problem zakresu liczby całkowitej trzeba

- Rozszerzyć jej zakres poprzez stworzenie własnej implementacji
- Posłużyć się działaniami modulo

$$(a * b) \bmod c = ((a \bmod c) * (b \bmod c)) \bmod c$$

Problemy

Notes

Jaką liczbę przyjąć, jako argument dzielenia modulo?

Zwyczajowo przyjmuje się takie q , że $d * q < \text{rozmiar słowa}$

$$\begin{aligned} t_{s+1} &= ((t_s - (T[s + 1] * d^{m-1}) \bmod q) * d) \bmod q + T[s + m + 1] \\ h &= d^{m-1} \bmod q \end{aligned}$$

$$t_{s+1} = ((t_s - (T[s + 1] * h) \bmod q) * d) \bmod q + T[s + m + 1]$$

Podsumowanie

Notes

- Równość $p = t_s$ nie jest w tym przypadku jednoznaczna odpowiedzią na pytanie o prawidłowe przesunięcie. Wymagane jest jeszcze sprawdzenie znaków

- Złożoność obliczeniowa

$$O(n)$$

Założenia

Zmieńmy łańcuchy znakowe. Niech nasz alfabet składa się z liter: A, C, G, T.

Szukajmy wzorca TATA w tekście TTATCTATAGCT.

Spróbujmy przesuwać wzorzec względem tekstu o większą liczbę znaków, niż w przeszukiwaniu naïwnym

Notes

Przesuwanie (1)

O ile możemy przesunąć wzorzec w takiej sytuacji.

T T
T A

1 znak

A teraz?

T T A T C
T A T A

2 znaki

Teraz znów podobna sytuacja?

T T A T C T
T A

1 znak

Notes

Przesuwanie (2)

T T A T C
T

1 znak

Sukces, ale o ile można przesunąć teraz?

T T A T C T A
T A T A

2 znaki

T T A T C T A T A G
T A T

2 znaki

Notes

Przesuwanie - podsumowanie

Notes

gdzie wystąpił błąd	ile znaków "z przodu i "z tyłu" jest takich samych	przesunięcie	C1 - C2 -1
2	1	1	0
4	1	2	2
2	1	1	0
1	0	1	0
5	2	2	2
3	0	2	2

Kryterium przesuwania

To co określić dla wzorca, to:

- analizować po kolejnych wszystkie jego prefiksy
- dla tych prefiksów wyznaczać długości prefiksów, które są równocześnie sufiksami

0	0
1	T 1
2	TA 0
3	TAT 1
4	TATA 2

Tworzymy tablicę preprocesingu $\Pi = [0, 1, 0, 1, 2]$.

Notes

Kryterium przesuwania

Ostatecznie, przesunięcie następuje o:

$$\max(1, j - \Pi[j] - 1)$$

Notes

Analiza złożoności obliczeniowej algorytmu BM wymaga podania jej dla dwóch faz osobno

- preprocessing $O(m)$
- przeszukiwanie $O(n)$

Założenia

Zmieniamy sposób porównywania znaków, zaczynamy teraz porównania od końca łańcucha wzorca

K A T **A** R A K T A
A R A **K**

O ile możemy przesunąć wzorzec względem tekstu; pokrywa się mogą litery A, zatem przesuwamy o 1

K A T A **R** A K T A
A R A **K**

Teraz pokryć się mogą litery R

K A T **A R A K** T A
A R A **K**

Notes

Założenia

Ale można też inaczej

O P A N C **E** R Z O N Y
G R Z E G **O** R Z

Mogemy przesunąć o 2 znaki, tak aby podpisać

O P A N C **E** R Z O N Y
G R Z **E** G O R Z

Mogemy przesunąć o 2 znaki, tak aby podpisać

O P A N C E **R** Z O N Y
G R Z **E** G O R Z

Notes

BCS

Mamy dwa rodzaje porównań i dwa preprocessing

Bad character shift (occurrence shift)

Przesunięcie następuje do takiej pozycji, żeby pozycji tekstu, gdzie nastąpiła niezgodność odpowiadał zgodny z nią znak wzorca

Tablica pre..BC ma rozmiar alfabetu

A	R	A	K
[A]	[K]	[R]	[T]
[1]	[4]	[2]	[4]

Notes

GSS

Good suffix shift (matching shift)

Przesunięcie następuje do takiej pozycji, żeby pokrył się pasujący suffix wzorca, jednak poprzedzony innym znakiem, niż ten, na którym wystąpiła niezgodność. Szukamy oczywiście wystąpienia tego suffiku jak najbardziej z prawej strony wzorca

Tablica pre..GS ma rozmiar równy długości wzorca

A	R	A	K
[4]	[4]	[4]	[1]

Notes

Założenia

Notes

K A T [A] R A K T A
A R A K

pre..BC[A] = 1, pre..GS[4] = 1, przesunięcie=1

K A T A [R] A K T A
A R A K

pre..BC[R] = 2, pre..GS[4] = 1, przesunięcie=2

K A T [A R A K] T A
A R A K

A co z przesunięciem? Zawsze pre..GS[0].

Podsumowanie

Notes

Mając dwie możliwe wartości przesunięć, w algorytmie Boyera-Moore'a wybieramy zawsze tę większą)

Analiza złożoności obliczeniowej algorytmu BM wymaga podania jej dla dwóch faz osobno

- preprocessing $O(m + \sigma)$
- przeszukiwanie $O(n)$

Notes

Wybrane algorytmy arytmetyczne

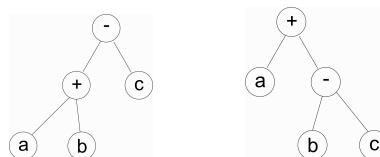
Tomasz M. Gwizdała

Drzewo działań

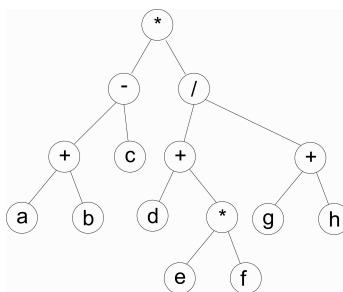
Rozważmy działanie:

$$(a + b - c) * (d + e * f) / (g + h)$$

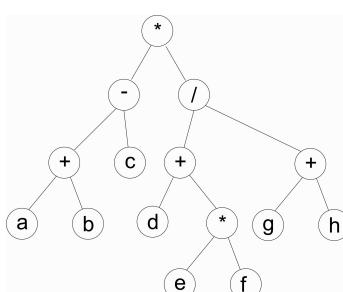
Utwórzmy drzewo binarne według reguły, mówiącej, że rodzic zawiera operator, którego operandami są jego potomkowie

**Drzewo działań**

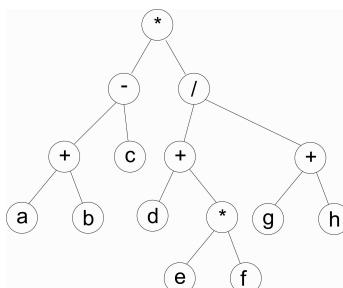
Ostatecznie drzewo przyjmuje postać:



Wykonajmy dla niego przejścia poznymi wcześniejszymi metodami.

Drzewo działań: in-orderKolejność: **a+b-c*d+e*f/g+h**

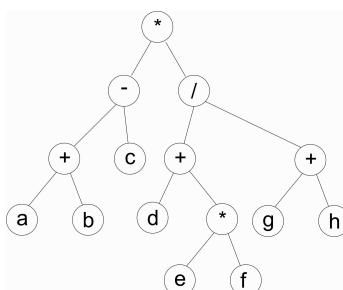
Drzewo działań: pre-order



Kolejność: $* - + abc / + d * ef + gh$

Notes

Drzewo działań: post-order



Kolejność: $ab + c - def * + gh + / *$

Notes

Drzewo działań: podsumowanie

In-order: $a+b-c*d+e*f/g+h$

Po pogrupowaniu: $((a+b)-c)*(d+e*f)/(g+h)$

Typowa notacja infiksowa.

Pre-order: $* - + abc / + d * ef + gh$

Notacja Polska (Jan Łukasiewicz, 1920).

Kolejność: $ab + c - def * + gh + *$

Odwrotna Notacja Polska

Reverse Polish Notation (Burks, Warren, Wright, 1954)

Notes

Realizacja działania

Operator występuje po operandach:

$a+b \rightarrow ab+$, $a/b \rightarrow ab/$, $a^b \rightarrow ab^$

Notacja jest postfiksowa i nienawiasowa

Stosując ONP możemy posługiwać się dwoma strukturami:
wejściem i stosem.

Mogliśmy natknąć tylko jeden z dwóch typów znaków:

- ▶ operand - przenosimy go na stos
- ▶ operator - powoduje pobranie dwóch ostatnich operandów ze stosu, wykonanie operacji i położenie wyniku na stos

Ostatecznie wynik działania znajduje się na stosie.

Notes

Struktury danych

Postługujemy się dwoma strukturami danych:

- ▶ wyjściem, na które zapisujemy gotowy łańcuch znaków w RPN
- ▶ stosem, strukturą tymczasową, na którą zapisujemy znaki, które w danym momencie nie mogą zostać zapisane do łańcucha wyjściowego

Ostatecznie wynik konwersji znajduje się w łańcuchu wyjściowym.

Notes

Algorytm

Na wejściu może pojawić się jeden z czterech typów znaków

- ▶ operand - wypisujemy ją na wyjściu
- ▶ nawias otwierający - kładziemy go na stosie
- ▶ operator - zdejmujemy ze stosu i wypisujemy na wyjściu wszystkie operatory priorytetu niemniejszym niż ten wczytany. Następnie kładziemy wczytany operator na stosie.
- ▶ nawias zamykający - zdejmujemy ze stosu i wypisujemy na wyjściu wszystkie działania aż do napotkania nawiasu otwierającego, którego nie wypisujemy

Kiedy łańcuch wejściowy jest pusty przepisujemy ze stosu wszystkie pozostałe tam znaki.

Notes

Przykład

$(a+b)*(c+d)$

znak	wyjście	stos
((
a	a	(
+	a	(+
b	ab	(+)
)	ab+	
*	ab+	*
(ab+	*(
c	ab+c	*(
+	ab+c	*(+)
d	ab+cd	*(+)
)	ab+cd+	*
	ab+cd+*	

Notes

Arytmetyka modularna

Celem jest wyznaczenie wartości wyrażenia:

$$x^n \bmod p$$

Zacznijmy od prostej obserwacji:

$$\begin{aligned} a * b \bmod p &= (a \bmod p) * (b \bmod p) \bmod p \\ a * b &= p(a \bmod p) * (b \bmod p) \\ a &= k * p + r_1, \quad b = l * p + r_2 \end{aligned}$$

$$P = r_1 * r_2 \bmod p$$

$$\begin{aligned} L &= (kp + r_1)(lp + r_2) \bmod p = \\ &= (klp^2 + kpr_2 + lpr_1 + r_1r_2) \bmod p = r_1 * r_2 \bmod p \end{aligned}$$

Notes

Arytmetyka modularna

$$x^2 \text{ mod } p =_p (x \text{ mod } p) * (x \text{ mod } p)$$

$$x^4 \text{ mod } p =_p (x^2 \text{ mod } p) * (x^2 \text{ mod } p)$$

$$x^8 \text{ mod } p =_p (x^4 \text{ mod } p) * (x^4 \text{ mod } p)$$

$$\dots$$

$$x^{2k} \text{ mod } p =_p (x^k \text{ mod } p) * (x^k \text{ mod } p)$$

$$x^n \text{ mod } p = \prod_{i=0}^{k-1} x^{b_i 2^i} \text{ mod } p =_p \prod_{i=0}^{k-1} (x^{b_i 2^i} \text{ mod } p),$$

gdzie $b = \{b_k, \dots, b_1, b_0\}$ jest binarnym zapisem wykładnika n .

Algorytm potęgowania

- ▶ przedstawiamy wykładnik potęgi w postaci liczby binarnej
- ▶ wynaczamy wartości $x^{2^i} \text{ mod } p$ wykorzystując wartości policzone dla niższych potęg
- ▶ jeśli na zadanej pozycji w zapisie binarnym występuje 1 uwzględnij resztę z dzielenia modulo w całkowitym iloczynie

Algorytm potęgowania - przykład

Załóżmy, że $x = 7$, $n = 11$, $p = 23$.
 $7^{11} \text{ mod } 23 = 1977326743 \text{ mod } 23 = 22$

$$11_{10} = 1011_2$$

i	b_i	2^i	7^m	$7^m \text{ mod } 23$	$\frac{p_i}{p_{i-1}^2} \% 23$	reszta $r_i =$ $p_i * r_{i-1}$
0	1	1	7	7	7	7
1	1	2	49	3	3	21
2	0	4			9	21
3	1	8	5764801	12	12	22

Notes

Notes

Notes

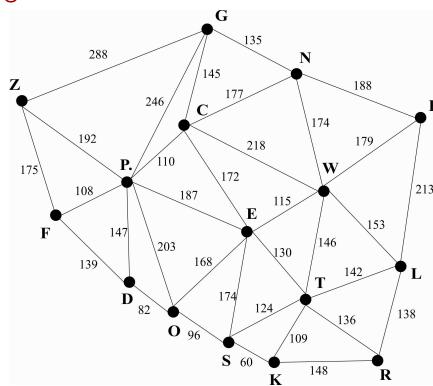
Notes

Notes

Grafy - dodatek

Tomasz M. Gwizdała

Znajomy graf



Szukanie najkrótszej drogi

Znajdźmy najkrótszą drogę od węzła O do węzła Z

Najkrótsza droga to O-P-Z (395 km).

Jak odtworzymy ją używając algorytmu Dijkstry?

Zanim uzyskamy odległość do węzła Z będziemy musieli przeanalizować wszystkie węzły, do których odległość jest mniejsza, niż do Z, czyli:
D(82), **S**(96), **K**(156), **E**(168), **P**(203), **T**(220), **F**(221), **W**(283),
C(303), **R**(304), **L**(362)

A*, czyli dodajemy heurystykę

Próbujemy wzbogacić naszą wiedzę o jakąś ocenę odległości od pomiędzy wierzchołkiem, do którego właśnie dotarliśmy oraz wierzchołkiem docelowym.

Na przykład zapiszmy współrzędne geograficzne wierzchołków, potraktujmy je jako współrzędne euklidesowe na płaszczyźnie i zastosujmy przelicznik *minuta kątowa = kilometr*

Za każdym razem będziemy analizować wartość funkcji będącej sumą

$$f(x) = g(x) + h(x),$$

gdzie:
x - aktualnie analizowany wierzchołek,
f(x) - droga pomiędzy wierzchołkiem początkowym a x (określona dokładnie),
h(x) - przewidywana przez heurystykę droga od x do wierzchołka docelowego.

Notes

Notes

Notes

Notes

Oszacowanie odległości

$$h(\Delta\phi, \Delta\Theta) = 1.8 * 60 * (0.25 * \Delta\phi^2 + \Delta\Theta^2)^{0.5}$$

węzeł	szerokość (Θ)	długość (ϕ)	oszacowanie heurystyczne
Z	53.426	14.561	0
O	50.660	17.939	350
S	50.257	19.019	418
D	51.111	17.053	284
F	51.932	15.511	169
P	52.409	16.934	169
E	51.777	19.455	319

Notes

Przebieg

{ O }				
	D	P	S	E
g+h	366	372	514	487
g	82	203	96	168
h	284	169	418	319

{ O, D }				
	P	F	S	E
g+h	372	390	514	487
g	203	221	96	168
h	169	169	418	319

Notes

Przebieg

{ O, D, P }				
	Z(P)	F	S	E
g+h	395	390	514	487
g	395	221	96	168
h	0	169	418	319

{ O, D, P, F }				
	Z(P)	Z(F)	S	E
g+h	395	396	514	487
g	395	396	96	168
h	0	0	418	319

Notes

A*, podsumowanie

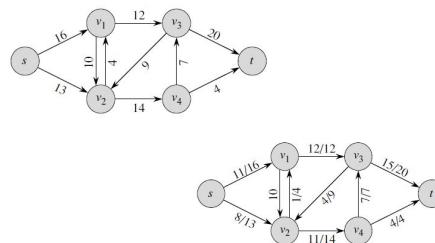
Na co trzeba zwrócić uwagę:

- ▶ warunkiem stopu jest znalezienie się węzła docelowego na szczytce kolejki priorytetowej
- ▶ warunkiem poprawności algorytmu jest dopuszczalność rozwiązań, funkcja heurystycznej oceny pozostałą odległość musi spełniać dwa warunki
 - ▶ funkcja musi być uralniona, niepowrotnie w naszym przykładzie odległości przez umowny przelicznik dawałyby fałszywe sygnały
 - ▶ wartości funkcji muszą być mniejsze, niż realne wartości

Notes

Inny graf

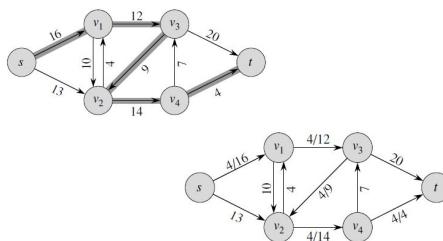
Jaki jest maksymalny przepływ pomiędzy s i t ?



Znaczenie notacji "z ukośnikiem".

Notes

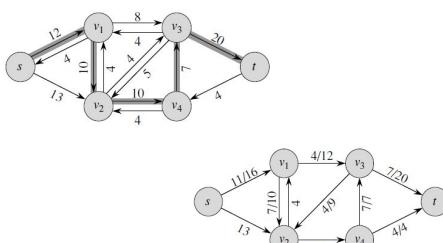
Iteracja 1



Przepustowość ścieżki - 4

Notes

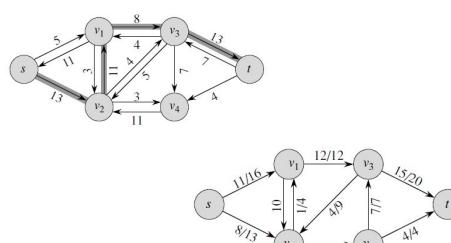
Iteracja 2



Przepustowość ścieżki - 7

Notes

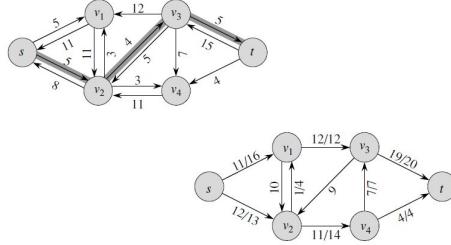
Iteracja 3



Przepustowość ścieżki - 10

Notes

Iteracja 4



Przepustowość ścieżki - 4

Notes

Notes

Notes

Notes