informatyka+

Algorytmika i programowanie

Bazy danych

Multimedia, grafika i technologie internetowe

Sieci komputerowe

Tendencje w rozwoju informatyki i jej zastosowań

Człowiek – najlepsza inwestycja

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

informatyka+

Kuźnia Talentów Informatycznych: Algorytmika i programowanie

Przegląd podstawowych algorytmów

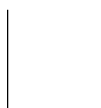
Marcin Andrychowicz,

Tomasz Kulczyński, Błażej Osiński

Człowiek – najlepsza inwestycja

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Przegląd podstawowych algorytmów



Rodzaj zajęć: Kuźnia Talentów Informatycznych
Tytuł: Przegląd podstawowych algorytmów
Autor: Marcin Andrychowicz, Tomasz Kulczyński, Błażej Osiński

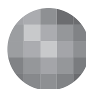
Redaktor merytoryczny: prof. dr hab. Maciej M Sysło

Zeszyt dydaktyczny opracowany w ramach projektu edukacyjnego **Informatyka+** — ponadregionalny program rozwijania kompetencji uczniów szkół ponadgimnazjalnych w zakresie technologii informacyjno-komunikacyjnych (ICT).
www.informatykaplus.edu.pl
kontakt@informatykaplus.edu.pl

Wydawca: Warszawska Wyższa Szkoła Informatyki
ul. Lewartowskiego 17, 00-169 Warszawa
www.wwsi.edu.pl
rektorat@wwsi.edu.pl

Projekt graficzny okładki: FRYCZ I WICHA

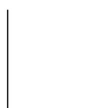
Warszawa 2010
Copyright © Warszawska Wyższa Szkoła Informatyki 2009
Publikacja nie jest przeznaczona do sprzedaży.



Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Przegląd podstawowych algorytmów

Marcin Andrychowicz,
Tomasz Kulczyński, Błażej Osiński



Streszczenie

Celem tego kursu jest przekazanie podstawowej wiedzy o popularnych problemach i ich rozwiązaniach (algorytmach) lub metodach tworzenia rozwiązań. Omówione zagadnienia są bardzo różnorodne. Obejmują techniki budowania algorytmów takie, jak: programowanie dynamiczne, rekurencja, strategie zachłanne. Zawarte są także podstawowe zagadnienia z kilku ważnych dziedzin algorytmiki: teorii grafów, przetwarzania tekstów i geometrii obliczeniowej. Każdy z tych tematów zostaje wprowadzony wraz z najbardziej znanymi problemami i algorytmami.

Zakładana jest znajomość języka C++, ale programujący w Pascalu czy jeszcze innym języku także powinni poradzić sobie ze zrozumieniem zawartych w kursie treści.

W tekście użyte zostają wielokrotnie pojęcia matematyczne, które mogą okazać się nowe nawet dla ucznia kończącego szkołę ponadgimnazjalną. Ich nieznajomość nie stanowi problemu w korzystaniu z kursu, gdyż są jednak omawiane w niezbędnym zakresie.

Spis treści

Streszczenie **4**

1 Programowanie dynamiczne **5**
1.1 Idea . . . 5
1.2 Znane problemy . . . 6

2 Sortowanie **8**
2.1 Sortowanie przez wybór . . . 8
2.2 Sortowanie przez scalanie . . . 10

3 Wyszukiwanie binarne **12**

4 Sortowanie pozycyjne **13**
4.1 Sortowanie przez zliczanie . . . 13
4.2 Sortowanie leksykograficzne . . . 14

5 Algorytmy zachłanne **15**

6 Rekurencja **17**
6.1 Algorytm Euklidesa . . . 17
6.2 Wieże Hanoi . . . 17

7 Przeszukiwanie z nawrotami (backtracking) **18**
7.1 Problem 8 hetmanów . . . 18

8 Grafy — Wprowadzenie **19**
8.1 Co to jest graf? . . . 19
8.2 Reprezentacja grafu na komputerze . . . 20
8.3 Przeszukiwanie grafów . . . 22

9 Algorytmy grafowe **24**
9.1 Problem cyklu i ścieżki Eulera . . . 24
9.2 Algorytm Dijkstry . . . 25
9.3 Algorytm Bellmana-Forda . . . 27

10 Algorytmy tekstowe **27**
10.1 Algorytm naiwny wyszukiwania wzorca . . . 28
10.2 Algorytm Knutha-Morrisa-Pratta . . . 28

11 Algorytmy geometryczne **29**
11.1 Podstawy . . . 29
11.2 Pole powierzchni . . . 31
11.3 Problem znajdowania wypukłej otoczki . . . 31

Literatura **32**

1 Programowanie dynamiczne

1.1 Idea

Programowaniem dynamicznym nazywamy strategię projektowania algorytmów, która opiera się na obliczaniu wyniku pewnego problemu na podstawie wyników dla tego samego problemu z innymi argumentami (najczęściej mniejszymi). Jak się przekonamy, najczęściej stosuje się ją do problemów optymalizacyjnych, czyli takich, w których mamy znaleźć najlepsze (w jakimś sensie, np. najtańsze, najkrótsze) rozwiązanie.

Liczby Fibonacciego

Definicja 1. Liczbami Fibonnaciego nazywamy ciąg liczbowy F_n zadany następującymi równościami:

1. $F_0 = F_1 = 1$.
2. $F_n = F_{n-1} + F_{n-2}$ dla $n \geqslant 2$.

Jak można obliczyć dla danego n, wartość F_n?

Oczywiście, wystarczy zapisać w programie powyższe równości. Można to zrobić tak:

```
int fib(int n) {
    if (n>=2)
        return fib(n-1) + fib(n-2);
    return 1;
}
```

Metoda powyższa nazywana jest rekurencją i będzie omówiona dokładnie później. Można też tak:

```
int F[1000];

int fib(int n) {
    F[0] = 1;
    F[1] = 1;
    for (int i = 2; i <= n; i++)
        F[i] = F[i-1] + F[i-2];
    return F[n];
}
```

Takie podejście nawiązuje do metody programowania dynamicznego: liczymy i zapamiętujemy wyniki problemu dla mniejszych argumentów, aby później w łatwiejszy sposób obliczyć wynik dla większych, aż w końcu dojdziemy do tego nas interesującego.

Można pamiętać tylko te wyniki, które będą nam później potrzebne, aby zredukować zużycie pamięci:

```
int fib(int n) {
    int a = 1, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

Ćwiczenie 1. Policz (być może za pomocą programu), ile operacji dodawania wykonuje każde z powyższych trzech podejść dla $n = 20$. Zastanów się, jakie korzyści niesie ze sobą programowanie dynamiczne (tzn. drugie i trzecie podejście).

Założenia

Programowania dynamicznego można używać, jeśli zależności pomiędzy poszczególnymi podproblemami nie tworzą cykli. Najpierw musimy bowiem obliczyć pewne wyniki, aby potem z nich skorzystać do obliczenia kolejnych. Ważne przy takim podejściu jest ustalenie kolejności obliczeń.

Ćwiczenie 2. Jaką kolejność obliczeń musimy zastosować przy następujących zależnościach?

- $a_n = a_{n-5} \cdot a_{n-7}$
- $b[n][m] = b[n+m-1][0] \cdot n + b[n+m-2][1] \cdot (n-1) + \dots + b[0][n+m-1] \cdot (1-m)$
- $c_p = -1$ dla p będących liczbami pierwszymi, $c_{n \cdot m} = c_n \cdot c_m$

1.2 Znane problemy

Każdy z niżej opisanych problemów możesz spróbować najpierw rozwiązać samodzielnie. Warto też zakodować rozwiązanie każdego z nich.

Wydawanie reszty

Jednym z najbardziej znanych problemów informatycznych jest problem **wydawania reszty**. Mamy dany pewien zbiór monet i/lub banknotów, których możemy użyć do wydania konkretnej kwoty. Pytanie brzmi, jak to zrobić korzystając z najmniejszej możliwej liczby monet lub banknotów. Istnieją co najmniej dwie odmiany tego problemu:

1. Istnieją pewne rodzaje monet (nominały) i w każdym z tych rodzajów mamy dostatecznie dużo monet (*Taką sytuację ma bank, o którym możemy założyć, że dysponuje zawsze odpowiednią ilością pieniędzy.*).
2. Mamy konkretnie ustalone ilości dostępnych monet w każdym z nominałów (*Bardziej codzienna sytuacja, obrazuje działanie kasy albo zwykłego portfela.*).

W pierwszym przypadku, można myśleć o prostych taktykach które mogłyby prowadzić nas do rozwiązania. Nie najgorszym pomysłem jest, na przykład, wybieranie zawsze największych nominałów, które mieszczą się w pozostałej do wydania kwocie. Takie podejście nazywa się **zachłannym**, o algorytmach zachłannych będzie więcej na jednych z kolejnych zajęć. I tak, mając 9zł 48gr do wydania, weźmiemy kolejno przy standardowych polskich nominałach: 5zł, 2zł, 2zł, 20gr, 20gr, 5gr, 2gr, 1gr. Nie da się tej kwoty wydać mniejszą ilością monet.

Ćwiczenie 3. Czy takie rozwiązanie jest zawsze optymalne (tzn. czy zawsze wydaje kwotę minimalną ilością monet)? Dla polskich nominałów? A dla innych zestawów monet?

Przedstawimy teraz rozwiązanie, którego można użyć w obu przypadkach. Co prawda, opis dotyczy przypadku drugiego, ale daje się łatwo zmodyfikować również dla pierwszego przypadku.

Definicja 2. Niezmiennikiem nazywamy stwierdzenie, które jest prawdziwe, za każdym razem, gdy wykonywanie algorytmu dochodzi do określonego punktu (np. pewnej instrukcji).

Nasz algorytm będzie brał kolejne monety (pojedyncze sztuki, nie nominały) z dostępnego zbioru i cały czas pamiętał tablicę $t[\,]$ o następującej własności: $t[i]$ to aktualnie minimalna liczba monet (spośród dotychczas przetworzonych) potrzebna do wydania kwoty i. Własność ta będzie naszym niezmiennikiem, a więc będzie zachodzić po przetworzeniu każdej kolejnej monety. Zauważmy, że

$$t'[i+x] = \min(t[i+x],\, t[i]+1)$$

gdzie t jest tablicą po pewnej ilości monet, a t' jest zmodyfikowaną tablicą, po przetworzeniu kolejnej monety, o nominale x. Korzystając z niezmiennika sprzed takiej akcji, łatwo wykazać prawdziwość tegoż samego niezmiennika po uwzględnieniu tej kolejnej monety. Po przetworzeniu wszystkich monet, otrzymujemy wynik końcowy, który jest poprawny zgodnie z utrzymywanym niezmiennikiem. Co więcej, dostajemy od razu wyniki dla wszystkich możliwych kwot, a nie tylko jednej.

Ćwiczenie 4. Jak zmienić ten algorytm, aby działał w przypadku pierwszym problemu?

Ćwiczenie 5. Oszacuj, jak szybko działa takie rozwiązanie. Czy można to poprawić?

Najdłuższy wspólny podciąg

Mając dane dwa ciągi (np. ciągi liczb naturalnych a_n i b_n), należy znaleźć ich najdłuższy wspólny podciąg, tzn. takie $i_1 < i_2 < \cdots < i_k$ oraz $j_1 < j_2 < \cdots < j_k$, że $a_{i_m} = b_{j_m}$ dla każdego $m \in 1, 2, \ldots, k$. W tym celu należy obliczyć kolejne wartości macierzy t, gdzie $t[g][h]$ oznacza najdłuższy wspólny podciąg (a raczej jego długość) spośród pierwszych g wyrazów ciągu a_n oraz pierwszych h wyrazów ciągu b_n. Łatwo wtedy obliczymy kolejne wyrazy t. Jeśli bowiem $a_g = b_h$, to

$$t[g][h] = 1 + t[g-1][h-1]$$

a w przeciwnym wypadku:

$$t[g][h] = \max(t[g-1][h], t[g][h-1])$$

.

Ćwiczenie 6. Uzasadnij poprawność powyższych równości. Ile pamięci potrzebuje to rozwiązanie? W jakim czasie podaje wynik?

Aby zoptymalizować zużycie pamięci, możemy zauważyć, że nasz algorytm korzysta co najwyżej z dwóch ostatnich wierszy macierzy (o ile obliczamy ją wierszami po kolei!), tak więc zamiast pamiętać całą tablicę t, możemy pamiętać tylko dwa ostatnie wiersze, a po obliczeniu kolejnego, jeden usuwać z pamięci. Komplikuje to nieco rozwiązanie, ale wystarczy je zmodyfikować tak, aby uzyskać żądany efekt:

```
// A i B to długości ciągów a i b
// tablica t[2][B] jest początkowo wyzerowana
for(int g=1; g<=A; g++)
    for(int h=1; h<=B; h++)
        if(a[g] == b[h])
            t[g % 2][h] = 1 + t[1 - g % 2][h - 1];
        else
            t[g % 2][h] = max(t[1 - g % 2][h], t[g % 2][h - 1]);
```

Optymalne mnożenie ciągu macierzy

Mnożenie macierzy to ciekawa operacja, którą szczegółowo omówimy przy innej okazji. W tej chwili istotne jest dla nas, że można pomnożyć dwie macierze o rozmiarach $a \times b$ i $b \times c$ wykonując $a \cdot b \cdot c$ mnożeń i uzyskując w wyniku macierz o rozmiarze $a \times c$.

Mając dany ciąg macierzy o rozmiarach, kolejno, $a_1 \times a_2$, $a_2 \times a_3$, ..., $a_n \times a_{n+1}$, chcemy obliczyć ich iloczyn (w podanej kolejności, gdyż mnożenie macierzy nie jest przemienne). Ale mnożenie macierzy jest operacją łączną, więc możemy dowolnie wstawić nawiasy przed rozpoczęciem mnożeń. Ile co najmniej mnożeń musimy łącznie wykonać?

Aby znaleźć optymalne rozwiązanie, zastanówmy się najpierw, jakie mnożenie będzie wykonane jako ostatnie. Oczywiście, będzie to mnożenie macierzy o rozmiarach $a_1 \times a_i$ i $a_i \times a_{n+1}$. Należy więc spróbować wszystkich takich możliwych i, wiedząc wcześniej jakie są koszty pomnożenia odpowiednio ciągów macierzy $a_1 \times a_2$, $a_2 \times a_3$, ..., $a_{i-1} \times a_i$ oraz $a_i \times a_{i+1}$, ..., $a_n \times a_{n+1}$.

Dochodzimy w ten sposób do rozwiązania, które dla każdego przedziału $[i, j]$ oblicza (zaczynając od najkrótszych przedziałów i kontynuując z coraz dłuższymi) optymalny koszt mnożenia ciągu macierzy

$a_i \times a_{i+1}$, $a_{i+1} \times a_{i+2}$, ..., $a_j \times a_{j+1}$. Oznaczmy taką wartość przez $w[i][j]$. Wynikiem całego zadania jest oczywiście $w[1][n]$. Mamy więc $w[i][i] = 0$ oraz dla $i < j$:

$$w[i][j] = \min_{i \leqslant k \leqslant j} (w[i][k] + w[k+1][j] + a_i \cdot a_{k+1} \cdot a_{j+1})$$

Ćwiczenie 7. Czy w problemie optymalnego mnożenia macierzy można zredukować zużycie pamięci, tak jak wcześniej? Jak?

2 Sortowanie

W następnej kolejności zajmiemy się ważnym problemem algorytmicznym, jakim jest sortowanie.

Definicja 3. Sortowaniem nazywamy porządkowanie zbioru danych względem pewnych cech charakterystycznych każdego elementu zbioru.

W informatyce spotykamy się z koniecznością sortowania różnych obiektów względem różnych kryteriów, np. słów w porządku leksykograficznym, czy prostych po kątach nachylenia. Na zajęciach skupimy się na najprostszym przypadku, sortowaniu tablicy liczb całkowitych, jednak przedstawiane pomysły są uniwersalne.

2.1 Sortowanie przez wybór

Sortowanie przez wybór (ang. *selection sort*) jest bardzo proste i naturalne, opiera się na następującym pomyśle:

> Wybierz najmniejszy element z tablicy i zamień go z pierwszym elementem.
> Następnie porządkuj tablicę bez pierwszego elementu.

Ćwiczenie 8. Zasymuluj działanie sortowania przez wybór na tablicy t:

i	0	1	2	3	4
$t[i]$	22	13	7	11	9
krok 1.					
krok 2.					
krok 3.					
krok 4.					

Jak można się było spodziewać, implementacja tego algorytmu jest bardzo prosta:

```
/* Dane: n-elementowa tablica t[] */
for (int i = 0; i < n; i++)
{
    int ind = i; /* Indeks najmniejszego elementu */
    for (int j = i+1; j < n; j++)
        if (t[j] < t[ind])
            ind = j;
    int pom = t[i]; /* Zamiana z wykorzystaniem zmiennej pomocniczej */
    t[i] = t[ind];
    t[ind] = pom;
    /* Niezmiennik: elementy od 0 do i są już na właściwych  *
     * miejscach - tych samych co w posortowanej tablicy.    */
}
/* Wynik: posortowana niemalejąco tablica t[] */
```

Zwróćmy uwagę na sformułowany w powyższym programie niezmiennik. Dowiedzenie jego poprawności w prosty sposób pociąga za sobą poprawność sortowania przez wybór. Przypomnijmy sobie definicję z poprzedniego rozdziału:

> **Definicja 4. Niezmiennikiem** nazywamy stwierdzenie, które jest prawdziwe, za każdym razem, gdy wykonywanie algorytmu dochodzi do określonego punktu (np. pewnej instrukcji).

W naszym przypadku niezmiennik jest bardzo prosty, problemu też nie przedstawia jego dowiedzenie. Posługujemy się metodą podobną do indukcji matematycznej znanej z lekcji matematyki.

Najpierw pokażemy, że gdy pierwszy raz algorytm natrafia na wiersz z definicją niezmiennika, jest on zachowany. Istotnie: wtedy $i = 0$, a w $t[0]$ jest najmniejszy element tablicy. Przy każdym następnym obrocie pętli licznik j nie przesuwa się po elementach mniejszych niż i, nie zostanie zmieniony porządek elementów $0, 1, \ldots, i-1$. Na miejscu i-tym zostaje umieszczony najmniejszy element z pozostałej części tablicy (od i do $n-1$), a więc znów elementy od 0 do i-tego są na właściwych miejscach. Tym samym udowodniliśmy prawdziwość niezmiennika.

Oczywiście dowodzenie poprawności niezmiennika nie jest sztuką dla sztuki, ale zwykle w prosty sposób wiedzie nas do dowiedzenia poprawności całego algorytmu. Tak jest także w tym przypadku: Skoro w trakcie ostatniego wykonania zewnętrznej pętli $i = n - 1$, więc po jej zakończeniu na mocy prawdziwości niezmiennika wynika, że elementy od 0 do $n - 1$ tablicy są już na właściwym miejscu, czyli cała tablica jest uporządkowana.

> **Ćwiczenie 9.** Sformułuj i udowodnij poprawność niezmiennika, dotyczącego zmiennej ind i tablicy t, który jest prawdziwy w każdym obrocie zewnętrznej pętli, przed wykonaniem instrukcji `int pom = t[i]` .

Analiza złożoności

Na przykładzie algorytmu sortowania przez wybór zastanowimy się, jak mierzyć i porównywać szybkość algorytmów. Najprostszym sposobem może wydawać się zaimplementowanie algorytmu, uruchomienie go na przykładowych danych i zmierzenie czasu, w jakim działa (np. za pomocą stopera). Przedstawia to jednak pewne trudności — chociażby to, że na różnych komputerach będziemy uzyskiwali zupełnie nieporównywalne wyniki.

Jednak algorytmy to coś więcej niż tylko ich późniejsza implementacja w konkretnym języku programowania, na konkretnej maszynie. Istnieją też bardziej uniwersalne, teoretyczne metody ich analizy, nie wymagające kodowania. Poznaliśmy już jedną z nich: dowodzenie poprawności za pomocą niezmienników. Teraz zajmiemy się metodą mierzenia wydajności algorytmu.

Aby określić złożoność algorytmu trzeba najpierw ustalić operację dominującą. W przypadku sortowania będzie to zwykle porównywanie dwóch elementów tablicy. Oznacza to, że uznajemy, iż szybkość algorytmu jest proporcjonalna jedynie do liczby porównań. Obliczmy teraz, ile porównań jest wykonywanych w trakcie sortowania przez wybór tablicy złożonej z n elementów:

- W każdym kroku algorytm wybiera minimum z fragmentu tablicy i umieszcza go na jego początku.
- Wykona się zatem n takich kroków.
- Fragmenty te mają długości kolejno $n, n-1, \ldots, 1$, a do wybrania minimum z k liczb potrzeba $k - 1$ porównań.
- Całkowita liczba porównań wynosi zatem:

$$(n-1) + (n-2) + \ldots + 1 + 0 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Jeżeli kiedyś wymyślimy ciekawy algorytm sortowania, który jednak wykonuje np. $n^3 + 3n^2 + 5n$ porównań, to stwierdzimy bez trudu, że jest on wolniejszy od sortowania przez wybór i pewnie nie bardzo użyteczny. A co, gdy otrzymamy algorytm działający w czasie $3n^2 + n$, lub $\frac{n^2}{6} + n$? Zwykle uznajemy te algorytmy za działające równie szybkie. Do porównywania szybkości używamy zwykle notacji „wielkie O".

Definicja 5. Dane są dwie funkcje $f(n)$, $g(n)$. Mówimy, że *f(n) jest O(g(n))* (zapisujemy $f(n) = O(g(n))$), gdy istnieje stała c, taka, że dla każdego $n \geqslant n_0$ dla pewnego n_0 zachodzi

$$f(n) \leqslant c\, g(n)$$

.

Na przykład $3n^5$ jest $O(n^5)$ (np. dla $c = 3$). a wszystkie funkcje: $\frac{n^2}{2} - \frac{n}{2}, 3n^2 + n, \frac{n^2}{6} + n$ są $O(n^2)$ (np. dla $c = 10^6$).

Algorytmy, dla których liczba operacji dominujących na wejściu wielkości n, jest funkcją klasy $O(n^2)$ nazywamy algorytmami o złożoności **kwadratowej**.

Ćwiczenie 10. Oblicz dokładną liczbę operacji przypisania wykonywaną w algorytmie sortowania przez wybór i określ klasę złożoności tego algorytmu, gdyby za operację dominującą uznać właśnie przypisanie.

2.2 Sortowanie przez scalanie

Okazuje się, że można sortować szybciej niż w czasie $O(n^2)$. W tym celu przydatna może okazać się metoda dziel i zwyciężaj:

> Dziel większy problem na mniejsze i buduj rozwiązanie całego problemu z rozwiązań problemów częściowych.

W przypadku problemu sortowania dzielimy tablicę na dwie części, sortujemy oddzielnie każdą z nich, a następnie scalamy (złączamy) dwie uporządkowane tablice w jedną. Zajmijmy się najpierw scalaniem: jak z dwóch uporządkowanych list uzyskać jedną? Należy w tym celu porównać ze sobą pierwsze elementy z obu list i mniejszy z nich umieścić na początku nowej tablicy. Dalej należy kontynuować z jedną tablicą krótszą.

Ćwiczenie 11. Przeprowadź scalanie posortowanych tablic *a* i *b*:

i	0	1	2	3
$a[i]$	3	4	4	5

i	0	1	2	3
$b[i]$	1	2	6	7

i	0	1	2	3	4	5	6	7
$a[i]$								

Przy implementacji należy zwracać uwagę na pewne szczegóły:

```
/* Dane - posortowane tablice:  *
 * n-elementowa a[]             *
 * m-elementowa b[]             */
int i = 0, j = 0;
while (i < n && j < m)
{
    if (a[i] <= b[j])
    {
        t[i+j] = a[i]; /* Dołączamy element z pierwszej tablicy. */
        i++;
    }
    else
    {
        t[i+j] = b[j]; /* Dołączamy element z drugiej tablicy. */
        j++;
    }
}
```

```
/* Może się okazać, że po skończeniu jednej z tablic, druga ma jeszcze  *
 * trochę elementów. Trzeba je zatem przepisać.                         */
while (i < n)
{
    t[i+j] = a[i];
    i++;
}
/* W praktyce tylko jedna z tych pętli while zostanie wykonana.   *
 * Dlaczego?                                                      */
while (j < m)
{
    t[i+j] = a[j];
    j++;
}
/* Wynik: posortowana tablica t[] o n+m elementach */
```

Policzmy jeszcze, ile porównań między elementami (operacji dominujących) jest wykonywanych przy scalaniu dwóch tablic o n i m elementach. Nie trudno zauważyć, że po wykonaniu każdego porównania `a[i] <= b[j]` wynik jest wydłużany o jeden element, a później już nie ma porównywania elementów tablic. Zatem wykonane zostanie co najwyżej tyle porównań, co elementów w ciągu wynikowym, czyli $n + m$ (tak na prawdę można to oszacowanie poprawić do $n + m - 1$, gdyż ostatniego wyrazu nigdy się nie porównuje, a jedynie przepisuje w którejś z pętli `while`).

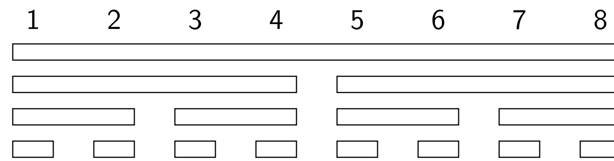
Implementacja rekurencyjna

Wróćmy do metody dziel i zwyciężaj. Wiemy, że mamy podzielić tablicę na dwie części, w jakiś sposób je posortować, a następnie scalić je w jedno. A zatem jak posortować owe połówki? Oczywiście moglibyśmy wykorzystać tu np. sortowanie przez wybór, ale nie uzyskalibyśmy złożoności lepszej niż $O(n^2)$. Rozwiązaniem jest ponowne zastosowanie tej samej metody: każdą z połówek podzielić na dwie, posortować w ten sam sposób i scalić. Działania te nie będą trwały w nieskończoność: nie trzeba dzielić tablicy jednoelementowej, która jest już uporządkowana. Rozwiązania tego typu nazywamy rekurencjami, będzie jeszcze o nich mowa w dalszej części kursu.

Ogólnie metodę sortowania przez scalanie prezentuje zatem poniższy pseudokod:

```
void sortuj(int t[], int p, int k)
/* Dane:                                *
 * tablica t[], zakres indeksów [p,k]    *
 * Wynik:                               *
 * elementy od t[p] do t[k] posortowane  */
{
    if (p < k)
    {
        int m = (p+k+1)/2;
        sortuj(t, p, m);
        sortuj(t, m+1, k);
        scal t[p],..,t[m] oraz t[m+1],..,t[k]
    }
}
```

Poniższy schemat ukazuje przedziały sortowane (a później scalane) dla ośmiu elementów na kolejnych poziomach wywołania rekurencyjnego:



Analiza złożoności

Zajmijmy się teraz oszacowaniem liczby porównań, a zatem i złożoności czasowej, sortowania tablicy n-elementowej. Zauważmy najpierw, że przy każdym zagłębieniu rekurencyjnym długości przedziałów zmniejszają się dwukrotnie, a przy długości 1 rekurencja się zatrzymuje. Oznacza to, że maksymalna głębokość zagłębienia rekurencyjnego to $\lceil \log n \rceil$.

Ponadto, na danym poziomie zagłębienia wszystkie scalane przedziały mają sumaryczną długość n (bo są całą tablicą). Zgodnie ze wcześniejszym spostrzeżeniem o liczbie porównań przy scalaniu otrzymujemy, że sumaryczna liczba porównań na danym poziomie zagłębienia rekurencyjnego nie przewyższa n.

Łącznie w całym algorytmie wykonanych jest zatem co najwyżej $n \lceil \log n \rceil$ porównań. Złożoność sortowania przez scalanie należy zatem do klasy $O(n \log n)$. Jest to tzw. złożoność **liniowo-logarytmiczna**. W ogólnym przypadku (tzn. nie zakładającym nic o sortowanych elementach poza możliwością porównywania) jest to rozwiązanie optymalne pod względem czasowym.

3 Wyszukiwanie binarne

Idea wyszukiwania binarnego nie jest nam obca, stosujemy ją choćby w zabawie „Jaka to liczba", gdy zgadujący stara się odkryć liczbę otrzymując jedynie odpowiedzi „za mała" lub „za duża". Najlepszą strategią wydaje się być wybieranie liczby w środku przedziału, by maksymalnie zmniejszyć przedział w jakim musimy dalej poszukiwać. Jak nie trudno zauważyć ponownie stosujemy tu metodę „dziel i zwyciężaj".

Wyszukiwanie elementu w tablicy

Metodę wyszukiwania binarnego ukażemy na przykładzie następującego problemu:
Mamy daną tablicę n liczb. Chcemy szybko sprawdzać czy jakaś liczba znajduje się w tej tablicy.

Ćwiczenie 12. Spróbuj wymyślić przykład urządzenia, bądź aplikacji, które muszą rozwiązywać taki problem.

i	0	1	2	3	4	5	6	7	8
$t[i]$	1	2	2	6	8	11	15	17	22

Najprostsze rozwiązanie tego problemu to liniowe przejrzenie całej tablicy w momencie otrzymania zapytania. Oznacza to jednak w pesymistycznym scenariuszu każdorazowe przejrzenie wszystkich n elementów tablicy.

Lepszym (szybszym) rozwiązaniem jest początkowe posortowanie całej tablicy (widać pierwsze zastosowanie poznanych algorytmów). Następnie, aby sprawdzić, czy elementu x znajduje się w tablicy, stosujemy wyszukiwania binarnego, zupełnie jak we wspomnianej zabawie:
Sprawdzamy element o indeksie $\frac{n}{2}$:

- Jeżeli $t[\frac{n}{2}] = x$, to już zakończyliśmy wyszukiwanie.
- Jeżeli $t[\frac{n}{2}] < x$, to szukaj dalej wśród elementów $t[\frac{n}{2}+1], \ldots, t[n-1]$.
- Jeżeli $t[\frac{n}{2}] > x$, to szukaj dalej wśród elementów $t[0], \ldots, t[\frac{n}{2}-1]$.

Ćwiczenie 13. Na powyższej przykładowej tablicy zasymuluj wyszukiwanie binarne liczb 2 i 7.

Analiza złożoności wyszukiwania binarnego nie przedstawia większych trudności: w każdym kroku zmniejszamy przedział, w jakim poszukujemy, co najmniej dwukrotnie, wykonamy zatem co najwyżej $\lceil \log n \rceil$ kroków. W każdym kroku wykonujemy stałą liczbę porównań (można po drobnej modyfikacji zawsze wykonywać tylko 1), zatem złożoność wyszukiwania binarnego należy do klasy $O(\log n)$ — złożoność **logarytmiczna**.

Zadania

W zadaniach A i B na wejściu znajdują się w dwóch wierszach:

- liczba n
- pewna permutacja ciągu $a_1, a_2, \ldots, a_n$ $(1 \leqslant a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n \leqslant 10^9)$

A. **Ciąg rosnący do środka**
Wypisz na wyjście pojedynczy wiersz zawierający następującą permutacje danego ciągu n-elementow $(1 \leqslant n \leqslant 1000)$:

$$a_1, a_3, a_5, \ldots, a_4, a_2$$

B. **Liczba inwersji**

Definicja 6. Mamy daną pewną permutację ciągu $a_1, a_2, \ldots, a_n$.
Inwersją w ciągu nazywamy parę liczb a_j, a_k gdzie $j < k$ oraz $a_j > a_k$.

Policz i wypisz liczbę inwersji w danej permutacji ciągu n-elementowym $(1 \leqslant n \leqslant 10^6)$.

C. **Pierwiastek dyskretny**

Definicja 7. Pierwiastkiem dyskretnym liczby k nazywamy największą liczbę naturalną x, dla której $x^2 \leqslant k$.

Mając daną na wejściu liczbę całkowitą k $(1 \leqslant k \leqslant 10^9)$ oblicz jej pierwiastek dyskretny.

4 Sortowanie pozycyjne

4.1 Sortowanie przez zliczanie

Czy da się posortować szybciej niż w czasie $O(n \log n)$? Odpowiedź brzmi: czasami tak.

Załóżmy, że mamy daną tablicę liczb z zakresu $O, 1, ..., M$. Aby uporządkować ją w czasie liniowym, proporcjonalnym do długości, należy policzyć ile jest w nim zer, jedynek, dwójek i tak dalej aż do M. Następnie wiedząc, że jest w niej c_i liczb o wartości i, pierwsze c_0 miejsc tablicy należy wypełnić zerami, następne c_1 jedynkami itd. Algorytm ten nazywany **sortowaniem przez zliczanie** (ang. *counting sort*).

Przyjrzyjmy się prostej implementacji tego algorytmu:

```
/* Dane: n-elementowa tablica t[] o liczbach z zakresu 0..M */
int c[M+1];
for (int i = 0; i <= M; i++)
  c[i] = 0;
for (int i = 0; i < n; i++)
  c[t[i]]++;
for (int i = 0, j = 0; i < n; i++)
{
  while (c[j]==0)
    j++;
```

```
    t[i] = j;
    c[j]--;
  }
  /* Wynik: posortowana rosnąca tablica t[] */
```

Ćwiczenie 14. Zmodyfikuj powyższy program tak, by działał dla liczb z przedziału od $-M$ do M.

Patrząc na powyższy kod bez trudu możemy zauważyć ograniczenia możliwości stosowania tego algorytmu. Aby zliczyć wystąpienia poszczególnych liczb potrzebna jest tablica $M+1$-elementowa, którą na dodatek trzeba wyzerować, co zajmuje czas liniowy: $O(M)$. Poza tym należy nadmienić, że wymagana dodatkowa tablica `c[]` może zajmować dużo miejsca w pamięci komputera: gdyby chcieć sortować zmienne całkowite 64-bitowe (np. `long long` na większości obecnych systemów) to potrzebowalibyśmy jakieś 2^{66} bajtów, czyli ponad 67 milionów terabajtów.

Warto jeszcze dokładnie oszacować złożoność powyższej implementacji w zależności o długości tablicy n i wielkości zakresu M. Pierwsze dwie pętle `for` są wykonywane się w czasie $O(n+M)$. Kolejna pętla `for` ma jednak zagnieżdżoną w sobie pętlę `while`. Wystarczy jednak zauważyć, że licznik j przebiega po całej tablicy `c[]` po indeksach od 0 do M, więc łącznie wszystkich wykonań wewnętrznej pętli będzie co najwyżej $M+1$.

Ćwiczenie 15. W formalny sposób (np. stosując niezmiennik) udowodnij, że wewnętrzna pętla `while` jest wykonywane łącznie co najwyżej $M+1$ razy.

W ten sposób otrzymujemy, że złożoność algorytmu to $O(n+M)$. Oznacza to, że sortowanie przez zliczanie warto stosować jedynie, gdy zakres z jakiego są elementy nie jest dużo większy od długości danej tablicy. W innym przypadku szybsze będzie np. sortowanie przez scalanie.

4.2 Sortowanie leksykograficzne

Metodę sortowania przez zliczanie możemy zastosować do sortowania słów w porządku leksykograficznym (czyli słownikowym). Dla uproszczenia zajmiemy się porządkowaniem słów o równej długości, złożonych z k liter każde.

Ćwiczenie 16. Oszacuj złożoność porządkowania słów opartego na sortowaniu przez scalanie.

Będziemy chcieli nałożyć na sortowanie przez zliczanie dodatkowe ograniczenie:

Definicja 8. Sortowanie nazywamy **stabilnym**, gdy dwa równe elementu w ciągu pozostawia w tym samym porządku co przed wykonaniem algorytmu.

Do dalszej analizy przyda nam się dodatkowy termin:

Definicja 9. **Sufiks** danego słowa s, to słowo na końcu s, powstające poprzez odcięcie od s kilku początkowych liter. Na przykład pak jest sufiksem słowa $rzepak$.

Algorytm sortowania słów wygląda następująco: k razy stabilnie sortujemy przez zliczanie ciąg słów. W kolejnych fazach porównujemy względem jednej, ustalonej pozycji w słowach, poczynając od ostatniej a kończąc na pierwszej. Nasuwa się pytanie: dlaczego sortujemy w tej kolejności?

Stwierdzenie: Po k sortowniach słowa są uporządkowane względem swoich k-literowych sufiksów.

Dowód indukcyjny: Po jednym sortowaniu uporządkowaliśmy słowa względem ostatniej litery, czyli jednoliterowych sufiksów. Krok indukcyjny: przed k-tym sortowaniem, mamy słowa uporządkowane względem $(k-1)$-literowych sufiksów.

- Jeżeli słowa różnią się na k-tej od końca literze, to zostaną ułożone w odpowiedniej kolejności.
- Jeżeli słowa mają tą samą literę na k-tej pozycji od końca, to stabilność sortowania zapewnia nam że zostaną w kolejności jaką wyznaczają ich $k-1$-literowe sufiksy. Zgadza się to z porządkiem leksykograficznym.

Poprawność całego algorytmu wynika natychmiast z powyższego stwierdzenia.

Poniższa implementacja nie sortuje tablicy słów `s[]`, a jedynie ciąg indeksów `t[]`.

```
/* Dane: n-elementowa tablica s[] słów k-literowych */
for (int i = 0; i < n; i++)
  t[i] = i;
int c[256];
for (int l = k-1; l>=0; l--)
{
  /* sortowanie względem liter na l-tej pozycji
   * c[i] - liczba słów o l-tej literze nie mniejszej niż i
   */
  for (int i = 'a'; i <= 'z'; i++)
    c[i] = 0;
  for (int i = 0; i < n; i++)
    c[s[t[i]][l]]++;          // zliczanie poszczególnych liter
  for (int i = 'b'; i <= 'z'; i++)
    c[i] += c[i-1];
  for (int i = n-1; i >= 0; i--)
  {
    char znak = s[t[i]][l];
    // znak z przetwarzanego słowa, z ustalonej pozycji
    a[c[znak]] = t[i];
    c[znak]--;
  }
  for (int i = 0; i < n; i++)
    t[i] = a[i+1];            // przepisywanie z pomocniczej tablicy a[]
}
/* Wynik: tablica t[] zawierająca indeksy słów z s[]    *
 *        w kolejności leksykograficznej                */
```

Algorytm ten polega na k-krotnym uruchomieniu sortowania przez scalanie. Jego złożoność czasowa jest zatem równa: $O(k\,(n + |\Sigma|))$, gdzie przez $|\Sigma|$ rozumiem wielkość alfabetu.

Ćwiczenie 17. *(Ambitne)* Jak zmodyfikować algorytm, by działał dla słów o różnej liczbie liter w czasie proporcjonalnym do sumy ich długości?

5 Algorytmy zachłanne

Przy poszukiwaniu rozwiązania problemu złożonego z ciągu decyzji okazuje się czasem, że najlepiej jest w każdym kroku dokonywać najlepszego wyboru. Istnieją problemy, w których taka strategia daje nam rozwiązanie optymalne. Algorytmy oparte na tym pomyśle nazywamy **zachłannymi**.

Problem kinomana

Kinoman ma do dyspozycji repertuar kina z godzinami rozpoczęcia i zakończenia seansów. Jak powinien wybierać filmy, by zobaczyć ich jak najwięcej?

Rozwiązanie zachłanne wydaje się oczywiste: należy zawsze wybierać film kończący się najwcześniej. Nie trudno przeprowadzić formalny dowód:

Załóżmy, że rozwiązanie optymalne zawiera więcej filmów niż to, które zostało uzyskane przez nasz algorytm. Weźmy pierwsze miejsce, na którym się różnią: w rozwiązaniu optymalnym został wzięty film, który kończy się później niż ten wybrany przez algorytm zachłanny. Ponieważ wybrany film kończy się później, można bez żadnego problemu wziąć zamiast niego film z algorytmu zachłannego. Przechodząc tak do końca uzyskujemy, że rozwiązanie zachłanne było równie dobre co optymalne. Uzyskana sprzeczność dowodzi poprawności algorytmu.

Wydawanie reszty

Powyższy dowód może wydawać się zbyt wydumany i niepotrzebny, bo algorytm wydaje się oczywiście poprawny. Często jednak strategia zachłanna, choć z pozoru poprawna, nie daje optymalnego rozwiązania. Jako przykład niech posłuży znany nam problem wydawania reszty minimalną liczbą monet. Strategia zachłanna wydaje się odpowiednia: wybieramy zawsze monetę o największym nominale, która mieści się w wydawanej kwocie.

Czy takie rozwiązanie działa? Nie, wystarczy wziąć następujące nominały: $1, 4, 8, 10$ i wydawać kwotę 12. Rozwiązanie zachłanne będzie potrzebowało trzech monet, podczas gdy wystarczą tylko dwie.

Ćwiczenie 18. Problem plecakowy: Dany jest plecak o danej wytrzymałości i ciąg przedmiotów o określonych wagach i wartościach. Wymyśl dwie różne, niebanalne strategie zachłanne próbujące zapakować najcenniejszy plecak i wskaż kontrprzykłady, dla każdej z nich, że nie generują rozwiązań optymalnych.

Minimalizacja kar

Firma zwleka z wykonaniem n zadań. Wykonanie i-tego zadania zajmuje d_i dni, a za każdy dzień opóźnienia trzeba zapłacić z_i złotych kary. W jakiej kolejności należy wykonywać zadania, by zapłacić jak najniższą karę?

Ćwiczenie 19. Wskaż kontrprzykłady dla algorytmów wybierających najpierw:

- zadania o najkrótszym czasie wykonania,
- zadania o największej karze za opóźnienie.

Rozwiązanie polega na uszeregowaniu zadań względem malejącego współczynnika $\frac{z_i}{d_i}$.

Dowód: Załóżmy, że w rozwiązaniu optymalnym zdania nie są uszeregowane w taki sposób. Wówczas istnieją dwa zadania, na pozycjach k i $k+1$ dla których $\frac{z_k}{d_k} < \frac{z_{k+1}}{d_{k+1}}$. Suma kar jaką generują te dwa zadania to:

$$K_1 = d_k \cdot z_k + (d_k + d_{k+1}) \cdot z_{k+1} = d_k \cdot z_k + d_{k+1} \cdot z_{k+1} + d_k \cdot z_{k+1}$$

Gdyby zamienić je miejscami to generowałyby karę:

$$K_2 = d_{k+1} \cdot z_{k+1} + (d_k + d_{k+1}) \cdot z_k = d_k \cdot z_k + d_{k+1} \cdot z_{k+1} + d_{k+1} \cdot z_k$$

Otrzymujemy zatem, że:

$$K_1 - K_2 = d_k \cdot z_{k+1} - d_{k+1} \cdot z_k > 0$$

bo

$$\frac{z_k}{d_k} < \frac{z_{k+1}}{d_{k+1}} \iff d_{k+1} \cdot z_k < d_k \cdot z_{k+1}$$

Oznacza to, że gdyby zamienić je, miejscami ogólna suma kar zmniejszyłaby się, zatem nie jest to rozwiązanie optymalne.

Zadania

A. **Deski**

Chcemy przybić n desek do podłogi. i-ta deska rozpoczyna się w miejscu p_i i kończy w k_i. Do przybicia każdej deski wystarczy jeden gwóźdź, jeden gwóźdź przybić może dowolnie wiele desek. Ile co najmniej gwoździ potrzeba?

Wejście:
n
$p_1\ k_1$
$p_2\ k_2$
...

Wyjście:
Pojedyncza liczba — minimalna liczba gwoździ.

B. **Plecak ciągły**
Pakujemy najbardziej wartościowy plecak o objętości V. Mamy do dyspozycji n różnych substancji, i-ta jest warta z_i złotych i zajmuje objętość v_i. Zakładając, że możemy do plecaka włożyć dowolny ułamek substancji, jaki najcenniejszy plecak uda nam się ułożyć?

Wejście:
$V\ n$
$v_1\ z_1$
$v_2\ z_2$
$\ldots$
$v_n\ z_n$

Wyjście:
Pojedyncza liczba oznaczająca wartość najcenniejszego plecaka, z dokładnością do dwóch miejsc po przecinku.

6 Rekurencja

Algorytm **rekurencyjny** rozwiązuje problem przez rozwiązanie pewnej liczby prostszych przypadków tego samego problemu. Zapisujemy go za pomocą funkcji rekurencyjnej, czyli wywołującej samą siebie.

6.1 Algorytm Euklidesa

Definicja 10. *Największym Wspólnym Dzielnikiem (NWD)* dwóch liczb naturalnych a i b nazywamy największą liczbę naturalną dzielącą zarówno a jak i b. Przykładowo $NWD(35, 20) = 5$.

Do obliczenia NWD dwóch liczb służy algorytm Euklidesa, którego implementacja znajduje się poniżej.

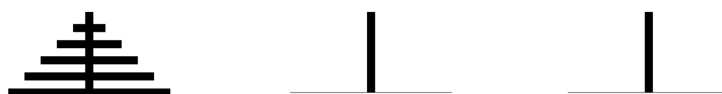
```
int nwd(int a,int b){
    if(b == 0) return a;
        return nwd(b, a%b);
}
```

$a\ mod\ b < b$, więc wartość b maleje przy każdym wywołaniu, co gwarantuje, że algorytm się zatrzymuje. Do poprawności algorytmu wystarczy pokazać, że $NWD(a, b) = NWD(b, a\ mod\ b)$:

1. niech $a = b \cdot n + m$, gdzie $m < b$
2. chcemy wykazać, że $NWD(b \cdot n + m, b) = NWD(b, m)$
3. wspólne dzielniki $b \cdot n + m$ i b dzielą też m
4. wspólne dzielniki b i m dzielą też $b \cdot n + m$

Algorytm ten działa w czasie $O(\log min(a, b))$.

6.2 Wieże Hanoi



Sytuacja początkowa.

W problemie **wież Hanoi** mamy 3 paliki i n krążków o różnej średnicy. Na początku są one ułożone jak na rysunku. Zadanie polega na przełożeniu wszystkich krążków na prawy palik, przy przestrzeganiu

pewnych ograniczeń — można przekładać tylko jeden krążek na raz oraz nie można kłaść krążka większego na mniejszy.

Oznaczmy krążki 1, 2, ...n — od najmniejszego do największego. Problem ten można rozwiązać rekurencyjnie. W tym celu zdefiniujemy procedurę `hanoi(m,k)`, która przenosi `12...m` w kierunku k, gdzie $k = 1$ oznacza przesunięcie na prawo a $k = -1$ na lewo od obecnej pozycji (cyklicznie, tzn. przyjmujemy, że na lewo od lewego palika znajduje się prawy). Przed uruchomieniem tej procedury `12...m` będą leżały na jednym palik (w ten dokładnie kolejności). Funkcja ta będzie przestawiała tylko krążki nie większe niż `m`. Zauważ, że położenia krążków większych niż `m` nie są istotne wewnątrz ten funkcji, gdyż w żaden sposób nie ograniczają możliwych ruchów. Jeśli `move(m,k)` oznacza przesunięcie `m` w kierunku k, to funkcję `hanoi` możemy zapisać następująco:

```
void hanoi(int m,int k){
   if(m == 0) return;
   hanoi(m-1, -k);
   move(m, k);
   hanoi(m-1, -k);
}
```

7 Przeszukiwanie z nawrotami (backtracking)

Backtracking to ogólna technika służąca do przeglądania możliwych rozwiązań i szukania tych, które spełniają pewne warunki. Polega ona na stopniowym rozbudowywaniu rozwiązania. Jeśli jednak aktualne rozwiązanie nie może być rozbudowane, to następuje powrót do poprzedniego kroku, gdzie podejmowana jest próba znalezienia innej możliwości. Proces ten można zapisać rekurencyjnie — aby przetworzyć rozwiązanie x:

1. jeśli x jest pełnym rozwiązaniem, to je wypisz
2. jeśli x nie może być rozbudowane, to zakończ procedurę rekurencyjną (następuje wówczas rekurencyjne *cofanie się*)
3. spróbuj wszystkich możliwości rozbudowania x i przetwórz każdą z nich (rekurencyjnie)

7.1 Problem 8 hetmanów

Przykładem zastosowania może być próba ustawienia 8 hetmanów na szachownicy tak, aby żadne dwa się nie atakowały. Hetman atakuje wszystkie pola leżące w tym samym wierszu, kolumnie lub na tej samej przekątnej.

W rozwiązaniu skorzystamy z obserwacji, że w każdym wierszu musi stać dokładnie jeden hetman i będziemy dostawiać kolejne hetmany w kolejnych wierszach. Moglibyśmy oczywiście przejrzeć wszystkie $8!$ możliwych ustawień hetmanów i sprawdzić, które są poprawne, jednak poszukiwanie z nawrotami pozwala pominąć wiele sytuacji, przez co w efekcie obliczenia będę trwały o wiele krócej. Poniższy program wypisuje wszystkie poprawne ustawienia 8 hetmanów na szachownicy.

```
#include <algorithm> //pozwala korzystać z funkcji abs, która zwraca wartość bezw
int het[9]; //pozycje hetmanów w kolejnych wierszach

void wypisz(){ //wypisuje bieżące ustawienie
   for(int i=1;i<=8;i++){
      for(int j=1;j<=8;j++)
         printf(het[i] == j ? "X" : "O");
      printf("\n");
   }
   printf("\n");
```

```
}

int backtrack(int n){ //n-liczba już rozstawionych hetmanów
   if(n == 8){
      wypisz();
   }else{
      for(het[n+1]=1;het[n+1]<=8;het[n+1]++){ //próbuj ustawić (n+1)-szego hetman
         bool ok=true; //czy w tym ustawieniu hetmany się nie atakują?
         for(int j=1;j<=n;j++)
            if(het[j] == het[n+1] || abs(het[n+1]-het[j]) == (n+1)-j) //jeśli j-t
               {ok=false; break;} //...hetmany się atakują, to to zapisz
         if(ok)
            backtrack(n+1);
      }
   }
}
...
backtrack(0);
```

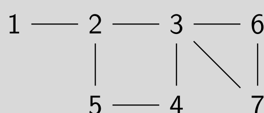
Choć trudno jest oszacować złożoności rozwiązań opartych na poszukiwaniu z nawrotami, to często działają one o wiele szybciej niż rozwiązania brutalne. Ich czas działania nie jest jednak wielomianowy.

8 Grafy — Wprowadzenie

8.1 Co to jest graf?

Definicja 11. Graf to obiekt matematyczny, który można wyobrażać sobie jako mapę zawierającą drogi i miasta; bardziej formalnie składa się ze zbioru wierzchołków (miast), które będziemy oznaczać kolejnymi liczba naturalnymi $1, 2, \ldots$ i zbioru krawędzi (dróg). Przyjmijmy oznaczenia:

- $V(G)$ (skrótowo V) — zbiór wierzchołków grafu G,
- $E(G)$ (skrótowo E) — zbiór krawędzi grafu G.



Przykładowy graf o 7 wierzchołka i 8 krawędziach. $V = \{1, 2, 3, 4, 5, 6, 7\}$.
$E = \{(1, 2), (2, 3), (3, 4), (4, 5), (2, 5), (3, 6), (6, 7), (7, 3)\}$

Definicja 12. Ścieżką łączącą wierzchołki v_0 i v_n nazywamy ciąg $(v_0, v_1, \ldots, v_n)$ taki, że każde dwa kolejne wierzchołki tego ciągu są połączone krawędzią. Długością ścieżki nazywamy liczbę krawędzi na tej ścieżce. Szczególne rodzaje ścieżek:

- **Droga** — ścieżka, której wszystkie wierzchołki są różne.
- **Cykl** — ścieżka, której pierwszy i ostatni wierzchołek są takie same.
- **Cykl prosty** — cykl, w którym wierzchołki się nie powtarzają (nie licząc pierwszego i ostatniego).

Definicja 13. **Stopień wierzchołka** to liczba wychodzących z niego krawędzi. Stopień wierzchołka i oznaczamy $deg[i]$.

Definicja 14. Graf nazywamy **spójnym**, jeśli każde dwa wierzchołki są połączone ścieżką.

Rodzaje grafów

- **Multigraf** to graf w którym parę wierzchołków łączy więcej niż jedna krawędź (tzw. **krawędzie wielokrotne**) lub istnieje krawędź łączącą wierzchołek z samym sobą (tzw. **pętla**).
- **Digraf**, czyli inaczej graf skierowany, to graf którego krawędzie są **skierowane**, czyli mają wyróżniony początek i koniec. Skierowanie może przykładowo oznaczać, że dana droga jest jednokierunkowa.
- **Drzewo** to graf:
  - spójny i niezawierający cykli prostych (tzw. **acykliczny**)
  - w którym dokładnie jedna droga łączy każdą parę wierzchołków
  - spójny, ale usunięcie dowolnej krawędzi rozspójnia go

  Wszystkie powyższe definicje są równoważne! Dla drzewa D, zachodzi $|E(D)| = |V(D)| - 1$. Mówimy, że drzewo A **rozpina** graf B, gdy $V(A) = V(B)$ i $E(A) \subset E(B)$.
- Drzewo **ukorzenione** to drzewo, którego jeden z wierzchołków został wyszczególniony (tzw. **korzeń**). **Ojcem** wierzchołka v nazywamy najbliższy wierzchołek na ścieżce od v do korzenia. Analogicznie **synem** wierzchołka v nazywamy dowolny wierzchołek, którego ojcem jest v.

8.2 Reprezentacja grafu na komputerze

Opis grafu

Jeśli chcemy pisać programy, które operują na grafach, to powinniśmy ustalić najpierw, w jakiej postaci będziemy je zapisywać. Najczęściej stosowany (m. in. w większości zadań olimpijskich) jest poniższy zapis:

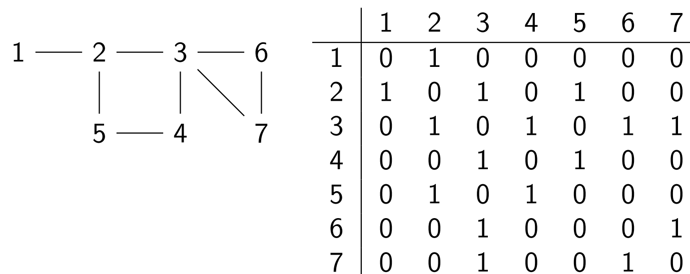
```
n m
a_1 b_1
a_2 b_2
...
a_m b_m
```

gdzie:

- n — liczba wierzchołków grafu
- m — liczba krawędzi grafu
- a_i, b_i — końce kolejnych krawędzi grafu

Macierz sąsiedztwa

W reprezentacji grafu, jako **macierzy sąsiedztwa**, trzymamy dwuwymiarową tablicę wartości logicznych sas, przy czym $sas[a][b]$ oznacza, czy istnieje krawędź pomiędzy wierzchołkami a i b. Dzięki temu możemy szybko sprawdzić, czy istnieje krawędź pomiędzy zadaną parą wierzchołków ($O(1)$), ale jednocześnie wyznaczenie wszystkich sąsiadów danego wierzchołka zabiera czas $O(|V|)$. Graf w tej postaci zajmuje $O(|V|^2)$ pamięci.



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

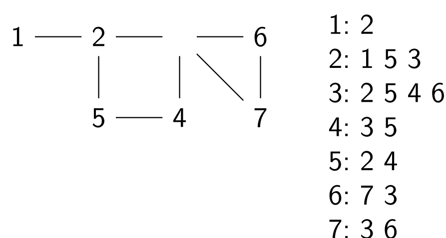
Graf i jego macierz sąsiedztwa.

Poniższy program wczytuje ze standardowego wejścia opis grafu i tworzy jego reprezentację w postaci macierzy sąsiedztwa.

```
#include <stdio.h>

#define MAXN 1000 //maksymalna ilość wierzchołków w grafie

int n; //ilość wierzchołków
int m; //ilość krawędzi
bool sas[MAXN+1][MAXN+1]={0}; //macierz sąsiedztwa

int main(){
    scanf("%d%d",&n,&m);
    for(int i=1;i<=m;i++){
        int a,b; //końce wczytywanej krawędzi
        scanf("%d%d",&a,&b);
        sas[a][b]=true;
        sas[b][a]=true;
    }
}
```

Listy sąsiedztwa

W reprezentacji grafu w postaci **list sąsiedztwa**, dla każdego wierzchołka przechowujemy listę jego sąsiadów. Dzięki temu możemy sąsiadów wierzchołka v wyznaczyć w czasie $O(deg[v])$, ale sprawdzenie czy istnieje krawędź łącząca a i b wymaga już czasu $O(min(deg[a], deg[b]))$. Graf w tej postaci zajmuje $O(|V| + |E|)$ pamięci.



1: 2
2: 1 5 3
3: 2 5 4 6
4: 3 5
5: 2 4
6: 7 3
7: 3 6

Graf i jego listy sąsiedztwa.

Być może zastanawiasz się, w jaki sposób będziemy przechowujemy w programie te listy. Będzie do tego służyła klasa `vector`, która jest w pewnym sensie tablicą o zmieniającym się rozmiarze. Poniższy kod ilustruje użycie tej klasy.

```
#include <stdio.h>
#include <vector>
using namespace std;
```

```
int main(){
    vector<int> v; //tworzy pusty vector
    v.push_back(10); //dodaje do niego kolejne elementy
    v.push_back(11);
    v.push_back(12);
    printf("%d %d\n",v[1],v.size()); //wypisuje element na pozycji nr 1
                                     //oraz liczbę elementów
    for(int i=0;i<v.size();i++) //wypisuje wszystkie elementy
        printf("%d\n",v[i]);
}
```

Poniższy program wczytuje ze standardowego wejścia opis grafu i tworzy jego reprezentację w postaci list sąsiedztwa.

```
#include <stdio.h>
#include <vector>
using namespace std;

#define MAXN 1000 //maksymalna ilość wierzchołków w grafie

int n; //ilość wierzchołków
int m; //ilość krawędzi
vector<int> kraw[MAXN+1]; //listy sąsiedztwa

int main(){
    scanf("%d%d",&n,&m);
    for(int i=1;i<=m;i++){
        int a,b; //końce wczytywanej krawędzi
        scanf("%d%d",&a,&b);
        kraw[a].push_back(b);
        kraw[b].push_back(a);
    }
}
```

8.3 Przeszukiwanie grafów

DFS

DFS (depth-first search), czyli **przeszukiwanie w głąb** jest to rekurencyjny algorytm przeszukiwania grafu. Bada on kolejne nieodwiedzone jeszcze wierzchołki, a gdy takich nie ma, cofa się. Bardziej formalnie, działa on według schematu:

Aby zbadać wierzchołek K:

- oznacz K jako odwiedzony
- zbadaj rekurencyjnie wszystkie nieodwiedzone wierzchołki sąsiadujące z K

Dodatkowo, w trakcie przeszukiwania DFS, będziemy tworzyli tablicę ojc, która przyda się później. $ojc[v]$ oznacza numer wierzchołka, z którego weszliśmy do v po raz pierwszy, czyli z którego wywołaliśmy `DFS(v)` (tzw. **ojciec** v). Poniższy kod wykonuje przeszukiwanie DFS z wierzchołka nr. 1 dla grafu w postaci list sąsiedztwa:

```
int ojc[MAXN+1]; //tablica ojców
bool vis[MAXN+1]={0}; //vis[i] - czy i jest odwiedzony?
```

```
//dzięki ={0} tablica ta będzie początkowo zapełniona wartością false

void DFS(int v){
    vis[v]=true; //oznacz v jako odwiedzony
    for(int i=0;i<kraw[v].size();i++) //dla każdego sąsiada v
        if(!vis[kraw[v][i]]){ //jeśli nie jest odwiedzony
            ojc[kraw[v][i]]=v; //to aktualizuj tablicę ojców
            DFS(kraw[v][i]); //...i go odwiedź
}
}
...
DFS(1);
```

Aby znaleźć dowolną ścieżkę z wierzchołka A do B, uruchamiamy algorytm DFS z wierzchołka A, a szukana ścieżka to $(A, \ldots, ojc[ojc[ojc[B]]], ojc[ojc[B]], ojc[B], B)$ (o ile $vis[B] == true$). Złożoność czasowa i pamięciowa algorytmu DFS wynosi:

- $O(|V|^2)$ dla macierzy sąsiedztwa,
- $O(|V| + |E|)$ dla list sąsiedztwa.

BFS

BFS (breadth-first search), czyli **przeszukiwanie wszerz** jest to metoda przeszukiwania grafu, która jeśli zostanie zainicjowana z wierzchołka v, to najpierw odwiedzi sąsiadów v, potem sąsiadów sąsiadów v, potem sąsiadów sąsiadów sąsiadów v .... Algorytm BFS korzysta ze struktury danych zwanej **kolejką FIFO**. Algorytm BFS działa według schematu:

- tworzymy pustą kolejkę
- wrzucamy do niej dowolny wierzchołek
- dopóki kolejka nie jest pusta:
  - wyciągnij wierzchołek z kolejki (oznaczmy go v)
  - rozpatrz każdego sąsiada v i jeśli nie był jeszcze w kolejce, to wrzuć go do kolejki

Poniższy kod realizuje algorytm BFS dla grafu w postaci macierzy sąsiedztwa, począwszy od wierzchołka nr 1:

```
int queue[MAXN+1]; //kolejka
int head=1; //indeks początku kolejki
int tail=1; //indeks końca kolejki (pierwszego za)
//zawartość kolejki tworzą elementy (począwszy od przodu):
//queue[head], queue[head+1], ..., queue[tail-1]

int ojc[MAXN+1]; //tablica ojców
bool vis[MAXN+1]={0}; //vis[i] - czy i był już dodany do kolejki?
...
queue[tail++]=1; //dodaje 1 do kolejki (na koniec)
vis[1]=true; //...i oznacza jako wrzuconą do kolejki
while(head != tail){ //dopóki kolejka nie jest pusta
    int v=queue[head++]; //weź element z początku kolejki
    for(int i=1;i<=n;i++) if(sas[v][i]) //dla każdego sąsiada v
        if(!vis[i]){ //jeśli nie był wrzucony do kolejki
            queue[tail++]=i; //...to go wrzuć,
            vis[i]=true; //...oznacz jako wrzuconego
```

```
            ojc[i]=v; //...i uaktualnij tablicę ojców
        }
}
```

Aby znaleźć dowolną ścieżkę z wierzchołka A do B, uruchamiamy algorytm BFS z wierzchołka A, a szukana ścieżka to $(A, \ldots, ojc[ojc[ojc[B]]], ojc[ojc[B]], ojc[B], B)$ (o ile $vis[B] == true$). Złożoność czasowa i pamięciowa algorytmu BFS jest taka sama jak DFS.

Zauważ, że w obu algorytmach przeszukiwania, używamy dwóch tablic — vis i ojc — a może wystarczyła by jedna?

9 Algorytmy grafowe

9.1 Problem cyklu i ścieżki Eulera

Definicja 15. Cyklem Eulera nazywamy cykl, który przechodzi przez każdą krawędź w grafie dokładnie raz.

Definicja 16. Ścieżką Eulera nazywamy ścieżkę, nie będącą cyklem, która przechodzi przez każdą krawędź w grafie dokładnie raz.

Warunki na istnienie

Warunki konieczne i dostateczne na istnienie:

- cyklu Eulera w grafie nieskierowanym:
  - graf jest spójny
  - każdy wierzchołek ma parzysty stopień
- ścieżki Eulera w grafie nieskierowanym:
  - graf jest spójny
  - dokładnie dwa wierzchołki mają nieparzysty stopień
- cyklu Eulera w grafie skierowanym:
  - z wierzchołka nr 1 można dojść do każdego innego
  - do każdego wierzchołka wchodzi tyle samo krawędzi ile z niego wychodzi

Algorytm Fleury'ego

Do szukania cyklu Eulera służy algorytm **Fleury'ego**. Poniżej znajduje się jego implementacja dla grafu skierowanego w postaci list sąsiedztwa:

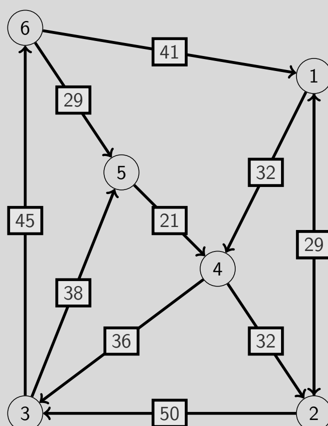
```
vector<int> cykl; //cykl Eulera w odwrotnej kolejności

void go(int v){ //aby odwiedzić v:
    while(!kraw[v].empty()){ //dopóki istnieje krawędź z v
        int w=kraw[v].back(); //zapamiętaj dokąd ona prowadzi
        kraw[v].pop_back(); //usuń ją
        go(w);  //i odwiedź wierzchołek, do którego prowadziła
        cykl.push_back(v); //a następnie dodaj v do cyklu
    }
```

```
}
...
go(1); //zaczynamy z dowolnego wierzchołka
for(int i=cykl.size()-1;i>=0;i--) //wypisujemy zawartość cykl
    printf("%d ",cykl[i]);        //...w odwrotnej kolejności
```

Złożoność zarówno czasowa jak i pamięciowa tego algorytmu wynosi $O(|E|)$.

Najkrótsze ścieżki

Definicja 17. Grafem **ważonym** lub inaczej **siecią** nazywamy graf, którego krawędzie mają w pewnym sensie "długość".



Przykład grafu ważonego

Liczby na krawędziach będziemy nazywać **wagą**, **długością** albo **kosztem** krawędzi. Długość ścieżki to suma długości jej krawędzi.

Problemem, który będziemy rozważać, będzie wyznaczanie najkrótszych ścieżek z wierzchołka s do wszystkich pozostałych wierzchołków. W każdej chwili obliczeń będziemy przechowywali pewne ścieżki z s do pozostałych wierzchołków. Konkretniej, będziemy przechowywali tablice:

- $dis[v]$ — długość najkrótszej znalezionej ścieżki z s do v,
- $ojc[v]$ — ostatni, nie licząc v, wierzchołek na najkrótszej ścieżce z s do v.

Podstawową operacją w rozważanych algorytmach będzie **relaksacja** krawędzi. Relaksacja krawędzi $v \to w$ polega na sprawdzeniu, czy przejście najkrótszą znaną ścieżką z s do v a następnie krawędzią $v \to w$ nie daje krótszej ścieżki z s do w niż obecnie znana. W postaci kodu:

```
if(dis[v]+waga(v,w) < dis[w]){
    dis[w]=dis[v]+waga(v,w);
    ojc[w]=v;
}
```

9.2 Algorytm Dijkstry

Algorytm **Dijkstry** służy do znajdowania najkrótszych ścieżek w grafach o *nieujemnych* wagach krawędzi. Działa on według schematu:

1. oznacz wszystkie wierzchołki jako nieodwiedzone ($vis[v] = false$)
2. dla każdego $v \in V$ przyjmij $dis[v] = \infty$

3. przyjmij $dis[s] = 0$

4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:

   (a) niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości

   (b) oznacz v jako odwiedzony

   (c) zrelaksuj wszystkie krawędzie wychodzące z v

A oto jego implementacja:

```
#define MAXN 10000 //maksymalna liczba wierzchołków
#define INF 1000000000 //nieskończoność

int n; //liczba wierzchołków
int m; //liczba krawędzi

vector<int> kraw[MAXN+1]; //listy sąsiedztwa
vector<int> waga[MAXN+1]; //waga[i][j] - waga krawędzi i-kraw[i][j]

int dis[MAXN+1]; //te dwie tablice
int ojc[MAXN+1]; //...opisane są w tekście powyżej
bool vis[MAXN+1]; //czy wierzchołek był już przetworzony

int main(){
//wczytanie grafu
    scanf("%d%d",&n,&m);
    while(m--){ //ile razy wykona się ta pętla?
        int a,b,c;
        scanf("%d%d%d",&a,&b,&c);
        //krawędź a->b o koszcie c
        kraw[a].push_back(b);
        waga[a].push_back(c);
    }
    //algorytm Dijkstry (s=1)
    for(int i=1;i<=n;i++) dis[i]=INF;
    dis[1]=0;
    for(int i=1;i<=n;i++) vis[i]=false;
    while(true){
            //wybierz nieodwiedzony wierzchołek o najmniejszym dis
        int v=1;
            //aktualnie wybrany wierzchołek
        for(int i=1;i<=n;i++) if(!vis[i]) v=i;
            //wybierz dowolny nieodwiedzony wierzchołek
        for(int i=1;i<=n;i++) //przejrzyj wszystkie wierzchołki
            if(!vis[i] && dis[i]<dis[v])
                //i jeśli ma mniejsze dis niż obecnie wybrany
                v=i; //to go weź
        if(vis[v] || dis[v]==INF) break;
            // jeśli nie ma już nieodwiedzonych wierzchołków
            // o skończonej odległości to kończymy
        vis[v]=true; //oznacz v jako odwiedzony
        for(int i=0;i<kraw[v].size();i++){
                //przejrzyj wszystkie krawędzie wychodzące z v
            int w=kraw[v][i];
```

```
            int k=waga[v][i];
            //relaksujemy krawędź v->w o koszcie k:
            if(dis[v] + k < dis[w]){
                dis[w]=dis[v]+k;
                ojc[w]=v;
            }
        }
    }
}
```

Algorytm ten ma złożoność pamięciową $O(|V| + |E|)$ i czasową $O(|V|^2)$. Tę ostatnią złożoność można poprawić, ale o tym będzie może później. Ponownie użyliśmy dwóch tablic — vis i ojc — a może wystarczyłaby tylko jedna?

9.3 Algorytm Bellmana-Forda

Algorytm **Bellmana-Forda** służy do wyznaczania najkrótszych ścieżek z pojedynczego źródła w sieciach mogących zawierać krawędzie o ujemnych wagach, ale nie zawierających ujemnych cykli. Jego schemat działania jest bardzo prosty:

$(|V| - 1)$-krotnie powtórz:

- zrelaksuj wszystkie krawędzie w dowolnej kolejności

A oto jego implementacja:

```
for(int i=1;i<=n;i++) dis[i]=INF;
dis[s]=0;
for(int q=1;q<=n-1;q++) //powtórz (n-1)-krotnie
    for(int v=1;v<=n;v++) //dla każdego wierzchołka
        for(int i=0;i<kraw[v].size();i++){ //dla każdej krawędzi
            int w=kraw[v][i];
            int k=waga[v][i];
            //relaksacja krawędzi v->w o koszcie k:
            if(dis[v] + k < dis[w]){
                dis[w]=dis[v]+k;
                ojc[w]=v;
            }
        }
```

Ten algorytm ma złożoność pamięciową $(|V| + |E|)$ i czasową $O(|V| \cdot |E|)$. Aby sprawdzić, czy graf zawiera ujemne cykle, możemy uruchomić pętlę $|V|$-ty raz i sprawdzić, czy udało się zrelaksować jakąkolwiek krawędź.

10 Algorytmy tekstowe

Dla informatyków pojęcia związane z tekstami mają nieco inne znaczenie niż normalnie.

Definicja 18. Alfabetem nazywamy zbiór dostępnych **znaków**, przy czym nie ograniczamy się tu tylko do liter.

Alfabetem mogą być np. znaki kodu ASCII o numerach 0-127.

Definicja 19. Słowo to ciąg znaków należących do rozpatrywanego alfabetu.

Słowo nie musi mieć żadnego znaczenia, tak jak w językach mówionych. Dobrym przykładem słowa jest $abbababa$, przy czym alfabet tu rozpatrywany zawiera co najmniej znaki a i b.

Wyszukiwanie wzorca

Najważniejszym i najbardziej popularnym problem algorytmiki tekstowej jest **wyszukiwanie wzorca** (ang. **pattern-matching**). Wzorzec i tekst to słowa nad zadanym alfabetem, nas interesuje zaś, ile razy i w których miejscach wzorzec występuje w tekście, jako podsłowo (czyli spójny fragment). Tak więc tor jest podsłowem $retoryki$, ale pol ani $cyko$ nie są podsłowami $encyklopedii$.

Ćwiczenie 20. Które dwuznakowe słowo występuje najczęściej w powyższym akapicie?

Ćwiczenie 21. Napisz program wyszukujący we wczytanym słowie wzorzec ala.

10.1 Algorytm naiwny wyszukiwania wzorca

Ćwiczenie 22. Ile razy wzorzec $yyyyyt$ występuje w słowie $yyyyyyyyyyyyyyyy$?

Podstawowym pomysłem, jak rozwiązać problem wyszukiwania wzorca, jest sprawdzenie w każdym miejscu w tekście, czy występuje tam wzorzec. Prowadzi to do tzw. **algorytmu naiwnego**. Niestety, jak łatwo sprawdzić, algorytm ten może wykonywać nawet około nm operacji, gdzie n jest długością tekstu, a m – długością wzorca. Dzieje się tak chociażby w przypadku, gdy tekst i wzorzec składają się z jednego znaku, powtórzonego po sobie n lub m razy. Nietrudno zauważyć, że w ćwiczeniu powyżej, algorytm naiwny wykonuję masę niepotrzebnych obliczeń, które człowiek potrafi pominąć (dlaczego?). Daje to podstawy sądzić, że algorytm naiwny można w jakiś sposób przyspieszyć.

10.2 Algorytm Knutha-Morrisa-Pratta

Definicja 20. Prefiks danego słowa s, to słowo na początku tego słowa, powstające poprzez odcięcie znaków od pewnej pozycji słowa s, do jego końca. Prefiksem słowa $abcdef$ jest więc a, albo $abcdef$, ale nie jest nim bc ani def. Analogicznie, **sufiksem** nazywamy słowo z końca danego słowa, powstające poprzez odcięcie pewnego prefiksu.

Ćwiczenie 23. Uszereguj alfabetycznie sufiksy słowa $encyklopedia$. Co możesz powiedzieć o szeregowaniu prefiksów?

Definicja 21. Prefikso-sufiks danego słowa, to jego podsłowo które jest zarówno prefiksem, jak i sufiksem.

W słowie ala, a, ala oraz słowo puste są prefikso-sufiksami, przy czym a jest jedynym **nietrywialnym** prefikso-sufiksem.

Ćwiczenie 24. Podaj inne przykłady słów z nietrywialnymi prefikso-sufiksami.

$P[i]$ – długość najdłuższego właściwego prefikso-sufiksu prefiksu o długości i danego słowa s. Inaczej: największe takie $k < i$, że $s_1 s_2 \dots s_k = s_{i-k+1} s_{i-k+2} \dots s_{i-1} s_i$.

Ćwiczenie 25. Wypełnij tablicę $P[\,]$ dla słowa $eeabeeaeeab$:

słowo w		e	e	a	b	e	e	a	e	e	a	b
$i =$	0	1	2	3	4	5	6	7	8	9	10	11
$P[i] =$												

Algorytm szybkiego obliczania elementów tablicy $P[\,]$ dla słowa s ma postać:

```
P[0] = 0;
P[1] = 0;
j = 0;
for(i = 2; i <= m; i++) {
    while( (j > 0) && (s[j+1] != s[i]) )
        j = P[j];
    if(s[j+1] == s[i])
```

```
        j++;
    P[i] = j;
}
```

Kluczowe jest spostrzeżenie, że najdłuższym prefikso-sufiksem krótszym od j (gdzie j też jest długością prefikso-sufiksu) jest ten o długości $P[j]$. Ponadto, aby uzyskać najdłuższy prefikso-sufiks słowa z dodatkową literą na końcu, należy ją dodać do pewnego (jak najdłuższego) prefikso-sufiksu słowa bez tejże litery tak, żeby nadal otrzymać prefikso-sufiks (równość $s[j+1] == s[i]$). Przeglądanie takich prefikso-sufiksów, od najdłuższych aż do słowa pustego, w poszukiwaniu odpowiedniego, jest właśnie zaimplementowane w wyżej przytoczonym fragmencie kodu. Pozostaje pytanie, jak szybko działa ten program. Można jednak zauważyć, że w każdej operacji zmieniamy wartość zmiennej j, która przy tym jest cały czas liczbą naturalną, a zwiększamy ją co najwyżej $m-1$ razy. Stąd, łączna ilość wykonanych operacji jest nie większa niż $O(j)$.

Wyszukiwanie wzorca s w tekście t, przy pomocy tablicy $P[]$, obliczonej jak wyżej, jest bardzo podobne:

```
j = 0;
for(i = 1; i <= n; i++) {
    while(j > 0 && s[j+1] != t[i])
        j = P[j];
    if(s[j+1] == t[i])
        j++;
    if(j==m) {
        printf("Wystąpienie wzorca w słowie na pozycjach od %d do %d.\n",i-m+1,i)
        j = P[j];
    }
}
```

Definicja 22. Dwa słowa są **równoważne cyklicznie**, jeśli jedno z nich powstaje z drugiego przez przesunięcie pewnego sufiksu na początek słowa (obrót cykliczny).

Ćwiczenie 26. Napisz program, który wczyta dwa słowa i sprawdzi, czy są one równoważne cyklicznie.

Ćwiczenie 27. Mając dane słowa a i b, wypisz wszystkie długości prefiksów słowa a występujących jako podsłowa w słowie b.

11 Algorytmy geometryczne

11.1 Podstawy

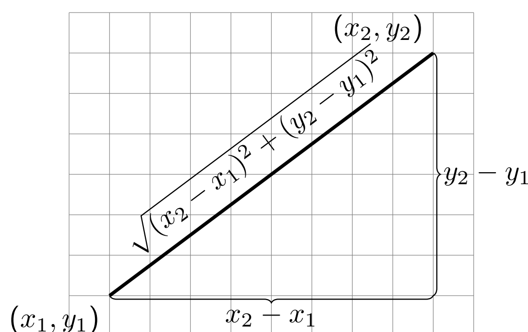
Jakkolwiek geometria jest w pewnym sensie odległa od informatyki, to jednak w geometrii komputery odgrywają także pewną rolę. Należy jednak od razu uświadomić sobie, że będzie mowa o geometrii obliczeniowej, gdzie wszelkie obiekty są opisane za pomocą liczb (a więc są zrozumiałe dla maszyny). Będziemy więc mówić o punktach na płaszczyźnie o konkretnych współrzędnych, odcinkach z długościami, miarach kątów.

Odległość

Podstawowym problemem jest obliczanie odległości pomiędzy punktami na podstawie współrzędnych. Mając punkty o współrzędnych (x_1, y_1) i (x_2, y_2), dystans pomiędzy nimi wynosi

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

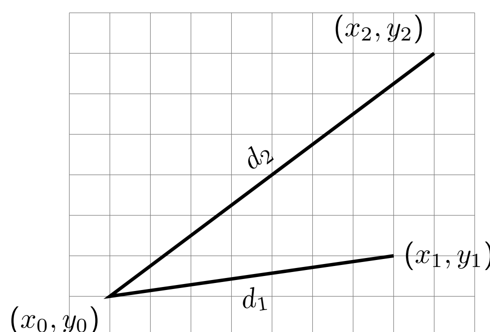
Wynika to z twierdzenia Pitagorasa, jak widać na rysunku:



Ćwiczenie 28. Napisz program sprawdzający, czy dane trzy punkty są wierzchołkami trójkąta prostokątnego.

Iloczyn skalarny

Iloczyn skalarny pomaga w dowiedzeniu się czegoś o kącie.



W sytuacji, jak na rysunku, iloczynem skalarnym tych dwóch wektorów nazywamy wartość

$$(x_1 - x_0)(x_2 - x_0) + (y_1 - y_0)(y_2 - y_0)$$

która jest jednocześnie równa $d_1 d_2 \cos \alpha$, a więc jest dodatnia gdy kąt jest ostry, równa zero dla kąta prostego i ujemna dla rozwartego.

Iloczyn wektorowy

Podobnie jak iloczyn skalarny, iloczyn wektorowy opisuje kąty. W tej samej sytuacji co powyżej, wynosi

$$(x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

co jest jednocześnie równe $d_1 d_2 \sin \alpha$, przy czym w tym wypadku α jest kątem skierowanym od półprostej z (x_1, y_1) do zawierającej (x_2, y_2).

Znak iloczynu wektorowego ma zatem następujące znaczenie:

- jest dodatni, gdy stojąc w punkcie (x_0, y_0) i patrząc w kierunku (x_1, y_1), a następnie odwracając się, żeby spojrzeć na (x_2, y_2), robimy to przeciwnie do ruchu wskazówek zegara;
- jest ujemy, gdy robiąc to samo, poruszamy się zgodnie z ruchem wskazówek zegara;
- jest równy zero, gdy wszystkie trzy punkty leżą na jednej prostej (a więc kąt jest zerowy lub półpełny).

Ćwiczenie 29. Napisz program który, mając dane współrzędne punktów A, B i C stwierdzi, czy idąc z A przez B do C skręcamy w punkcie B w lewo czy prawo.

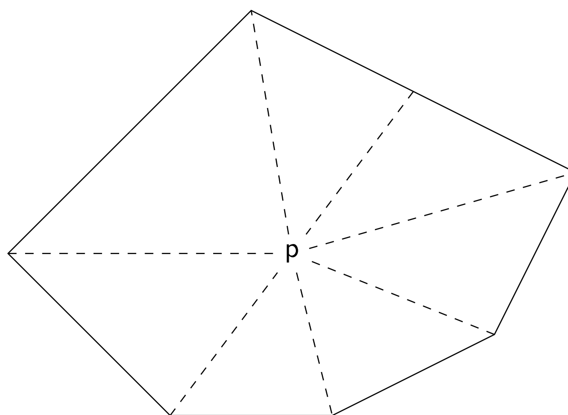
Jednocześnie, iloczyn wektorowy jest sposobem mierzenia pola. Wiemy bowiem, że wartość bezwzględna iloczynu wektorowego jest równa podwojonemu polu trójkąta $(x_0, y_0), (x_1, y_1), (x_2, y_2)$.

Dla wszystkich trzech powyższych wartości geometrycznych, warto mieć oddzielne funkcje implementujące je tak, żeby nie pomylić się przy wielokrotnym wpisywaniu współrzędnych.

Ćwiczenie 30. Napisz program znajdujący przecięcie dwóch prostych, wyznaczonych przez pary punktów, przez które przechodzą.
Podpowiedź: najlepiej wyprowadzić równania prostych w postaci $Ax + By + C = 0$. Mając dwie pary współrzędnych (x, y) można wyznaczyć A, B i C z odpowiedniego układu równań. Następnie, punkt przecięcia tych dwóch prostych to rozwiązanie kolejnego układu, złożonego z równań tych prostych.
Uwaga! Proste mogą być równoległe, lub się pokrywać. Spróbuj wychwycić takie przypadki.

11.2 Pole powierzchni

Wiemy już, jak obliczyć pole trójkąta. Chcielibyśmy jednak umieć obliczyć pole dowolnego wielokąta $A_1A_2 \ldots A_n$. Okazuje się, że wystarczy obrać dowolny punkt p, chociażby (0,0) dla uproszczenia obliczeń, i od razu mamy, że pole jest równe $|w(A_1, A_2, p) + w(A_2, A_3, p) + \cdots + w(A_{n-1}, A_n, p) + w(A_n, A_1, p)|$, gdzie $w(A, B, C)$ jest iloczynem wektorowym wektorów $\overrightarrow{CA}$ i $\overrightarrow{CB}$. Łatwo to zauważyć, gdy wielokąt jest wypukły, a p leży w jego środku:



bo wtedy, albo wszystkie iloczyny wektorowe są dodatnie, albo wszystkie ujemne, ale sumują się do pola całego wielokąta z odpowiednim znakiem. Wzór jest prawdziwy także wtedy, gdy p leży na zewnątrz i/lub wielokąt nie jest wypukły. Wtedy odpowiednie trójkąty z przeciwnymi znakami składają się dokładnie tak, że do sumy brane jest pole wielokąta, a obszary poza nim nie.

Ćwiczenie 31. Sprawdź to!

Ćwiczenie 32. Podaj przykład obiektu na płaszczyźnie, poza wielokątami, którego pole mógłby obliczać komputer.

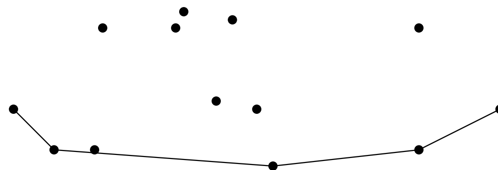
Ćwiczenie 33. Mając dany wielokąt wypukły (zadany przez współrzędne kolejnych wierzchołków), weźmy obszar, który składa się z punktów odległych od brzegu tego wielokąta o co najwyżej r. Jak można obliczyć pole tego obszaru?

11.3 Problem znajdowania wypukłej otoczki

Mając zbiór punktów na płaszczyźnie, chcemy znaleźć najmniejszy (pod względem obwodu) wielokąt który je wszystkie zawiera. Możemy sobie wyobrazić deskę z wbitą pewną liczbą gwoździ, wokół których kładziemy sznurek. Po naciągnięciu tak, by opierał się na niektórych gwoździach, sznurek tworzy właśnie wypukłą otoczkę zbioru gwoździ.

Ćwiczenie 34. Znajdź wypukłą otoczkę zbioru punktów. Rozwiązanie nie musi być optymalne. Może być $O(n^3)$, gdzie n jest liczbą punktów. A może potrafisz szybciej?

Aby szybko rozwiązać ten problem, posortujmy punkty po współrzędnych, od lewej do prawej. Zajmijmy się najpierw dolną częścią wypukłej otoczki, potem analogicznie skonstruujemy górną. Zauważmy, że skrajnie lewy punkt i ten skrajnie prawy, oba należą do otoczki (a więc do części dolnej). Otoczka dolna (bo tak będziemy nazywać ten dolny obrys) składa się z odcinków które w punktach zbioru skręcają przeciwnie do ruchu wskazówek zegara.



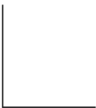
Żeby znaleźć otoczkę dolną, bierzmy kolejne punkty, od lewej do prawej i dorzucajmy je do aktualnie tworzonej otoczki. Jeśli trzy ostatnie punkty na naszej otoczce tworzą skręt zgodny ze wskazówkami zegara, to należy przedostatni usunąć i kontynuować ten proces, aż do uzyskania skrętu w lewo. Następnie bierzemy kolejny na prawo punkt, i tak dalej. Wszystkie usuwane punkty pozostają w ten sposób na górze od naszej otoczki, a ponieważ przeglądamy wszystkie punkty zbioru, więc każdy z nich jest albo na otoczce albo nad nią, a więc dokładnie tak jak chcieliśmy. Daje to rozwiązanie w złożoności $O(n \log n)$ (bo musimy na początku posortować punkty). Bardzo podobny (choć nie identyczny) algorytm nazywa się **algorytmem Grahama**. Przedstawiona tutaj wersja jest nieco łatwiejsza w implementacji od oryginału.

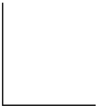
Ćwiczenie 35. Zastanów się, dlaczego faza tworzenia otoczki po posortowaniu punktów ma złożoność $O(n)$.

Ćwiczenie 36. Zaimplementuj algorytm znajdowania wypukłej otoczki.

Literatura

1. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C., *Wprowadzenie do algorytmów*, WNT, Warszawa 2004
2. Banachowski L., Diks K., Rytter W., *Algorytmy i struktury danych*, WNT, Warszawa 2003
3. Kubica M., *Wstęp do programowania i Metody programowania (potok funkcyjny)*
4. de Berg M., van Kreveld M., Overmars M., Schwarzkopf O., *Geometria obliczeniowa: algorytmy i zastosowania*, WNT, Warszawa 2007





W projekcie **Informatyka +**, poza wykładami i warsztatami, przewidziano następujące działania:

- 24-godzinne kursy dla uczniów w ramach modułów tematycznych
- 24-godzinne kursy metodyczne dla nauczycieli, przygotowujące do pracy z uczniem zdolnym
- nagrania 60 wykładów informatycznych, prowadzonych przez wybitnych specjalistów i nauczycieli akademickich
- konkursy dla uczniów, trzy w ciągu roku
- udział uczniów w pracach kół naukowych
- udział uczniów w konferencjach naukowych
- obozy wypoczynkowo-naukowe.

Szczegółowe informacje znajdują się na stronie projektu

www.informatykaplus.edu.pl