Spis treści

Rozdział 1. Odkrywamy i stosujemy algorytmy **7**

1.1. Wstęp 7

1.2. Struktura programu 8

1.3. Lista bibliotek używanych w przykładach 9

1.4. Najczęściej używane typy danych i przykłady deklaracji 9

1.5. Podstawowe konstrukcje algorytmiczne 10

1.5.1. Instrukcja warunkowa 10

1.5.2. Instrukcje iteracyjne 12

1.5.3. Operacje na łańcuchach 13

1.6. Przykłady programów wykorzystujących opisane konstrukcje 14

1.7. Algorytmy numeryczne i teorioliczbowe 21

1.7.1. Obliczanie pierwiastka kwadratowego — algorytm Newtona-Raphsona 22

1.7.2. Obliczanie miejsca zerowego funkcji metodą połowienia przedziałów — metoda bisekcji 23

1.7.3. Obliczanie pola obszaru pod krzywą w zadanym przedziale w układzie współrzędnych — całkowanie numeryczne 23

1.7.4. Rozszerzony algorytm Euklidesa 25

1.7.5. Arytmetyka modularna 28

1.7.6. Chińskie twierdzenie o resztach (jedno z najważniejszych w teorii liczb i kryptografii) 30

1.7.7. Potęgowanie modularne 32

1.8. Algorytmy rekurencyjne 36

1.8.1. Ciąg Fibonacciego 39

1.8.2. QuickSort 40

1.8.3. Problem Józefa Flawiusza 41

1.9. Struktury danych 42

1.9.1. Drzewa 43

1.9.2. Stos 49

1.9.3. Kolejka 51

1.9.4. Kopiec 52

1.10. Algorytmy grafowe 54

1.10.1. Reprezentacja grafu w pamięci komputera 55

1.10.2. Przeszukiwanie grafu 58

1.10.3. Przeszukiwanie wszerz BFS 59

1.10.4. Przeszukiwanie w głąb DFS 63
1.10.5. Cykl Eulera — przez każdą krawędź przechodzimy tylko raz 64
1.10.6. Cykl Hamiltona 66
1.11. Elementy teorii gier 68
1.11.1. Gra w życie 68
1.11.2. Gra logiczna NIM 71
1.11.3. Dodawanie nimliczb 73
1.11.4. Mnożenie nimliczb 73
1.11.5. Potęgowanie nimliczb 74
1.12. Sprawdź się — zadania na koniec rozdziału 75

Rozdział 1.

ODKRYWAMY I STOSUJEMY ALGORYTMY

„Większość programistów programuje nie dlatego, że spodziewa się zapłaty lub uwielbienia tłumów, ale dlatego, że programowanie jest dla nich zabawą"
— Linus Torvalds[1]

1.1. Wstęp

Celem każdego podręcznika jest zaznajomienie uczniów z nowymi dla nich informacjami z danej dziedziny i pokazanie, jak można je zastosować w praktyce. Ze względu na zróżnicowany poziom wiedzy informatycznej u uczniów szkół ponadgimnazjalnych niniejsza publikacja została zaplanowana tak, że kolejność realizacji poszczególnych zagadnień jest praktycznie dowolna. Niektóre z nich zostały omówione dokładnie, a inne są tylko zasygnalizowane, co daje czytelnikom możliwość samodzielnego wyszukiwania optymalnego rozwiązania.

Osoby, które już miały do czynienia z programowaniem, mogą opuścić informacje zawarte we wstępie. Przedstawiono tutaj najważniejsze konstrukcje i polecenia, których analiza powinna wystarczyć do zapoznania się ze specyfiką języka C++.

> Jeżeli natomiast przedstawione informacje wydadzą się zbyt lakoniczne, warto sięgnąć po podręcznik *Informatyka Europejczyka* dla szkół ponadgimnazjalnych — zakres rozszerzony, część 1, rozdział 3, „Programowanie języka C++".

Co jest potrzebne do pisania dobrych programów? Przede wszystkim wiedza matematyczna i umiejętność logicznego (algorytmicznego) myślenia — znajomość poleceń języka jest sprawą wtórną. Na pewno zgodzisz się ze mną, że osoba, która potrafi się porozumiewać w kilku językach, wcale nie musi być autorem bestsellerowych książek! Zresztą zawsze można zajrzeć do słownika. Tak samo w programowaniu — najważniejsza jest koncepcja

---

1 Linus Benedict Torvalds — fiński programista, twórca jądra Linuksa.

rozwiązania problemu. Kodowanie jest o wiele prostsze ☺. Przecież — tak jak ze słowników — możemy korzystać z dołączonej dokumentacji, jeżeli nie pamiętamy sposobu zapisu jakiejś konstrukcji.

Zaczynamy. Miłej zabawy ☺.

Definicja

Algorytm jest skończonym ciągiem zdefiniowanych czynności (zadań) prowadzących do określonego celu (wyniku).

Program jest to algorytm napisany w języku programowania.

P

W naszej podróży po algorytmach do ich implementacji będziemy stosować język C++ (środowisko Dev-C++ ver. 4.9.9.2 na licencji GNU).

.. 1.2. Struktura programu

Program w języku C++ składa się z części deklaracyjnej i operacyjnej.

Część deklaracyjna

Deklarujemy w niej elementy programu, które są wykorzystywane w części operacyjnej, oraz dołączamy biblioteki dyrektywą `include`:

```
#include <nazwa_biblioteki>
```

W celu uniknięcia problemu konfliktu nazw (gdy kilka obiektów lub funkcji ma takie same nazwy) używamy tzw. przestrzeni nazw. W naszych przykładach będziemy się ograniczać do standardowej — `std`.

```
using namespace std;
```

Część operacyjna

Zawiera funkcję `main` („ciało główne" naszego programu), od której program rozpoczyna swoje działanie:

```
int main ()
{
system ("pause");
return 0;
}
```

Instrukcja `system("pause")` zatrzymuje wywołane okno konsoli (żebyśmy zdążyli przeczytać wyniki na ekranie).

Wskazówka

Program wysyłany na konkurs (olimpiadę) jest sprawdzany przez specjalne narzędzia, tzw. sprawdzarki. W takim przypadku instrukcję należy usunąć albo poprzedzić symbolem komentarza –`//system("pause")`.

Do zwracania wartości przez funkcję używamy polecenia `return`.

Jeżeli funkcja `main` zwraca wartość 0, oznacza to prawidłowe wykonanie programu (sukces); inna wartość jest interpretowana jako błąd.

Zmienne w programie możemy deklarować wewnątrz funkcji (zmienne lokalne) lub przed rozpoczęciem funkcji `main` (zmienna globalna).

Zmienna lub stała zadeklarowana jako globalna jest dostępna w całym kodzie programu, natomiast lokalna — jak sama nazwa wskazuje — lokalnie, czyli wewnątrz funkcji lub procedury, w której została zadeklarowana.

Więcej na temat struktury w podręczniku *Informatyka Europejczyka* dla szkół ponadgimnazjalnych — zakres rozszerzony, część 1, rozdział 3.2, „Wprowadzenie do programowania".

1.3. Lista bibliotek używanych w przykładach

- `iostrem` — zawiera funkcje wejścia/wyjścia konsoli.
- `cmath` — to biblioteka funkcji matematycznych.
- `string` — to biblioteka funkcji tekstowych (łańcuchów znaków).
- `algorithm` — zawiera wiele funkcji z zaimplementowanymi algorytmami (bardzo wygodna i przydatna).
- `cstdlib` — to biblioteka funkcji zarządzających pamięcią, konwersji podstawowych typów danych, generowania liczb pseudolosowych.
- `ctime` — zawiera funkcje służące do zarządzania czasem.
- `fstream` — niezbędna do korzystania ze strumieniowych operacji plikowych (dostęp do plików dyskowych) w celu zapisu, pobierania lub modyfikacji danych.

1.4. Najczęściej używane typy danych i przykłady deklaracji

Przed użyciem w programie identyfikatora (stałej, zmiennej) musimy go zadeklarować, czyli podać nazwę oraz typ.

Przykłady deklaracji

Deklaracja zmiennych całkowitych:

```
int a, b;
long long x, y;
```

Deklaracja zmiennych rzeczywistych:

```
double p;
float x;
```

Deklaracja zmiennej typu logicznego (zmienna może przyjmować wartość `true` lub `false`):

```
bool pierwsza;
```

Deklaracja zmiennej typu znakowego:

```
char znak;
```

Deklaracja tablicy 10-elementowej zawierającej liczby całkowite (indeks pierwszego elementu wynosi 0, ostatniego — 9). Dostęp do elementów tablicy odbywa się poprzez podanie nazwy tablicy i indeksu elementu.

```
int tablica[10];
tablica[9]=18
```

Deklaracja zmiennej typu łańcuchowego — ciąg znaków. Dostęp do poszczególnych elementów łańcucha jest taki sam jak do elementów tablicy.

```
string nazwa;
```

1.5. Podstawowe konstrukcje algorytmiczne

1.5.1. Instrukcja warunkowa

Stosujemy ją wtedy, gdy dalsze działanie programu jest uzależnione od spełnienia warunku określonego w instrukcji:

```
if (warunek) instrukcja;
else instrukcja;
```

lub

```
if (warunek) instrukcja;
else if (warunek) instrukcja;
else instrukcja;
```

Wskazówka

W przypadku ciągu instrukcji stosujemy nawiasy klamrowe:

```
if (warunek)
   {instrukcja1;
    instrukcja2;
    ...
   instrukcjan;
   }
    else if (warunek)
{instrukcja1;
 instrukcja2;
 ...
instrukcjan;
}
   else
{instrukcja1;
 instrukcja2;
 ...
instrukcjan;
}
```

Przykład

Fragment programu sprawdzającego, czy wprowadzona zmienna jest parzysta:

```
if (a%2==0) cout<<"TAK";
else cout<<"NIE";
```

Operatory arytmetyczne

Dodawanie: +.

Odejmowanie: -.

Mnożenie: *.

Dzielenie całkowitoliczbowe: /.

Reszta z dzielenia: %.

Operatory porównania

Równy: ==.

Mniejszy: <.

Większy: >.

Większy lub równy: >=.

Mniejszy lub równy: <=.

Różny: !=.

Operatory logiczne

&& — iloczyn logiczny.

`(a % 3 = = 0) && (a % 5= =0)` — wynik jest *true*, jeżeli *a* dzieli się przez *3* i przez *5*.

|| — suma logiczna.

`(a % 3 = = 0) || (a % 5= =0)` — wynik jest *true*, jeżeli *a* dzieli się przez *3* lub przez *5*.

Odpowiednikiem *true* jest *1*, odpowiednikiem *false* — *0*.

Po napisaniu instrukcji:

```
cout <<(5>7);
```

wyświetlone zostanie *0*.

1.5.2. Instrukcje iteracyjne

Stosujemy je przy wielokrotnym wykonywaniu określonego ciągu operacji.

Definicja

Iteracja — wielokrotne powtórzenie tego samego ciągu instrukcji, przy czym dane wyjściowe i-tej operacji są wejściowymi i+1.

Jeżeli wiemy, ile razy będzie wykonana pętla, stosujemy instrukcję `for`.

```
for (zmienna=wartość_początkowa; zmienna<=wartość_końcowa;
zmienna++)
{
ciąg instrukcji;
}
```

Przykład

Fragment programu sumującego liczby naturalne od 1 do 10:

```
for (int i = 1; i<=10; i++)
 suma = suma + i;
```

Jeżeli liczba iteracji zależy od spełnienia określonego warunku, stosujemy instrukcję `while`. Warunek możemy sprawdzać na początku pętli (warunek wejścia) lub na końcu (warunek wyjścia).

Na początku:

```
while (warunek)
{
ciąg instrukcji;
}
```

Na końcu:

```
do
{
ciąg instrukcji;
}
while (warunek);
```

1.5.3. Operacje na łańcuchach

Łańcuch jest to ciąg znaków (tablica znaków zakończona znakiem o kodzie 0).

Najczęściej wykonywane operacje:

- dodawanie (konkatenacja) — +,
- relacyjne — >, <, ==, !=,
- sprawdzanie długości tekstu `T` — `T.size()`.

Przykłady

Dla łańcuchów `T1` i `T2`:

```
T1="Puchatek ";
T2="i przyjaciele";
```

Konkatenacja:

```
cout<<T1+T2<<endl;
//Wynik – Puchatek i przyjaciele
```

Porównania:

```
cout<<(T1==T2)<<endl;
//Wynik – 0
cout<<(T1>T2)<<endl;
//Wynik – 0
```

Sprawdzanie długości łańcucha:

```
cout<<T1.size()<<endl;
//Wynik – 9
```

1.6. Przykłady programów wykorzystujących opisane konstrukcje

Wypisywanie większej z dwóch liczb — zastosowanie instrukcji warunkowej

Program porównuje dwie liczby `a` i `b`. W zależności od wprowadzonych wartości zwraca wartość zmiennej `a`, `b` lub komunikat „Liczby są równe".

```
#include <iostream>
using namespace std;
int main ()
{int a, b
//Wprowadzamy liczby – wartości zmiennych
 cin >> a >> b;
//Sprawdzamy, czy a jest większe od b
if (a > b)
//Jeżeli wynik porównania jest prawdą – wypisujemy wartość zmiennej a
  cout<< a << endl;
//W przeciwnym wypadku...
  else
//Sprawdzamy, czy liczby są równe
 if (a == b)
//Wypisujemy komunikat "Liczby są równe"
  cout << "Liczby są równe" << endl;
//W przeciwnym wypadku...
  else
//Pozostała tylko ta opcja :-)
 cout<< b << endl;
 system("pause");
 return 0;
}
```

Wypisywanie dzielników liczby a — zastosowanie instrukcji iteracyjnej

Program wypisuje wartości liczbowe zmiennej `i`, dla których operacja `a % i` (reszta z dzielenia) zwraca 0.

```
#include <iostream>
using namespace std;
int main ()
{
```

```
//Deklarujemy zmienną a (naszą liczbę) oraz i (kandydata na dzielnik)
//Zmiennej i przypisujemy od razu wartość równą 2
 int a, i = 2;

      //Wprowadzamy zmienną
 cin >> a;
//Jedynka na pewno jest dzielnikiem, wypisujemy ją bez sprawdzania
  cout << "1 ";

//Sprawdzamy kandydatów na dzielniki do połowy liczby
//Jeżeli i jest mniejsze lub równe połowie liczby, wchodzimy do pętli
   while (i <= a/2)
 {

//Sprawdzamy, czy reszta z dzielenia a przez i jest równa 0
   if (a % i == 0)

//Jeżeli tak, wypisujemy i
   cout<< i <<" ";
//Zwiększamy wartość i o 1 i kończymy pętlę. Zapis i++ jest
//równoważny i = i + 1
//Jest to operator inkrementacji. Operator dekrementacji: i-- i++;
  }
//Wypisujemy liczbę a
 cout << a << endl;          system("pause");
 return 0;
}
```

Ile reszek, ile orłów — zastosowanie instrukcji iteracyjnej

Program wykorzystuje liczby losowe do symulacji rzutów monetą. Przyjmujemy, że wylosowana liczba nieparzysta odpowiada wyrzuceniu reszki, a parzysta — orła.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
int main()
{
//Zmienne orly i reszki służą odpowiednio do zliczania zer
//i jedynek
//Przyjmujemy, że zero odpowiada wyrzuceniu orła, a jeden –
//wyrzuceniu reszki
```

```
  int ile, orly=0, reszki=0,rzut;
//Określamy liczbę rzutów monetą
  cin>>ile;
//Uruchamiamy generator liczb pseudolosowych
  srand(( time( NULL);
//Wykonujemy liczbę losowań określonych zmienną ile
  for ( int i=0; i<ile; i++)
  {
//Do losowania używamy funkcji rand(), która losuje liczby
//całkowite z zakresu od 0 do RAND_MAX
      rzut = rand()%2;
      if (rzut==0)
       orly++;
       else
       reszki++;
       }
     cout<<"Reszki="<<reszki<<endl;
     cout<<"Orły="<<orly<<endl;
      system("pause");
   return 0;
}
```

Znajdowanie maksymalnej wartości w tablicy z wylosowanymi liczbami — zastosowanie instrukcji iteracyjnej

W algorytmie przyjmujemy, że maksymalną wartość ma pierwsza wylosowana liczba (`max = t[0]`). Następnie porównujemy zmienną `max` z kolejną liczbą z tablicy — w przypadku gdy liczba jest większa, zmienna `max` przyjmuje jej wartość.

```
#include<iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
  int i, t[10];
  srand( time( NULL ) );
//Wypełniamy 10-elementową tablicę liczbami od 0 do RAND_MAX
  for ( i = 0; i < 10; i++)
   t[i] = rand();
//Zmiennej max przypisujemy wartość pierwszego elementu
```

```
  int max=t[0];
//Porównujemy max z kolejnymi elementami tablicy
//Jeżeli kolejny element tablicy jest większy od wartości max,
//to zmienna max przyjmuje tę wartość
  for (int i=1; i<10; i++)
  if (t[i]>max)
   max=t[i];
  cout<<max<<endl;
  system("pause");
  return 0;
}
```

Program sprawdzający, czy podany wyraz jest palindromem — zastosowanie operacji na łańcuchach

Definicja

Palindrom — wyraz, który brzmi jednakowo czytany od lewej do prawej i od prawej do lewej (palindromom przypisywano właściwości magiczne ☺).

Program porównuje kolejno znaki z lewej i prawej strony łańcucha aż do osiągnięcia środkowej pozycji lub porównania dwóch sąsiednich znaków. Jeżeli porównywane znaki są różne — program zwraca komunikat „Nie" i kończy działanie. Jeżeli pętla została wykonana do końca (porównywane znaki były jednakowe), program zwraca komunikat „TAK".

```
#include <iostream>
#include <string>
using namespace std;
int main()
{ string wyraz;
  cin>>wyraz;
//Obliczamy długość wyrazu
  int l = wyraz.size();
  int i = 0;
//Dopóki nie doszliśmy do połowy wyrazu...
  while (i <= l/2)
//Porównujemy litery (elementy tablicy) z lewej strony z kolejnymi
//literami z prawej
  {if (wyraz[0]!=wyraz[l-1])
//Jeżeli są różne, wypisujemy NIE
  {cout <<"NIE"<< endl;
    system("pause");
```

```
//i wychodzimy z programu
    return 0;
  }
    i++;
 }
  cout<<"TAK"<<endl;
  system("pause");
  return 0;
}
```

Wskazówka

Każdy palindrom liczbowy w systemie dziesiętnym jest podzielny przez 11. Ta zasada dotyczy również liczb lustrzanych (liczby, które są lustrzanym odbiciem).

Program sprawdzający, czy podana liczba jest liczbą pierwszą — zastosowanie instrukcji iteracyjnej

Liczba pierwsza dzieli się tylko przez 1 i samą siebie — nie ma innych dzielników. W programie sprawdzamy, czy reszta z dzielenia liczby a (kandydata na liczbę pierwszą) przez liczbę `i` (kandydata na dzielnik), która przyjmuje kolejno wartości od 2 do pierwiastka kwadratowego z danej liczby, jest różna od zera. Jeżeli tak — liczba jest pierwsza i nie ma dzielników różnych od 1 i samej siebie. W przeciwnym razie liczba nie jest pierwsza. Znalezienie pierwszego dzielnika przerywa dalsze sprawdzanie.

```
#include <iostream>
#include <string>
#include <cmath>
using namespace std;
  int main ()
   {int a, i=2;
    //Wprowadzamy pomocniczą zmienną prawda
    bool prawda=true;
    cin>>a;
 //W przypadku sprawdzania, czy liczba ma dzielniki, wystarczy
 //sprawdzić liczby z zakresu od 0 do wartości równej
 //pierwiastkowi kwadratowemu danej liczby (i*i <= a)
//Każdy kolejny dzielnik jest wielokrotnością znalezionej
//wcześniej wartości
  while (i*i<=a && prawda)
  {
```

```
    if (a%i==0)
    prawda=false;
    i++;
  }
//Instrukcja warunkowa zawiera zawsze operację, której wynikiem
//jest „true” lub „false”. W przypadku zmiennych logicznych nie
//musimy wykonywać porównania, bo przyjmują tylko wartości „true”
//lub „false”
    if (prawda) cout<<"TAK"<<endl;
    else
      cout<<"NIE"<<endl;
          system("pause");
          return 0;
    }
```

Program „odwracający” 10-elementową tablicę — zastosowanie instrukcji iteracyjnej

Zadaniem programu jest zamiana miejscami elementów tablicy — „odwrócenie”. Program pokazuje, jak można tego dokonać z wykorzystaniem funkcji swap.

```
#include <iostream>
#include <algorithm>
using namespace std;
 int main ()
 {int tablica[10];
 int i;
 srand(time(NULL));
 for (i=0; i<10; i++)
  {
       //Wypisujemy zawartość tablicy wypełnionej losowymi liczbami
   tablica[i]=rand();
    cout<<tablica[i]<<endl;
  }
  cout<<endl;
     //Zmienna przechowująca indeks elementu lewego
  int lewy = 0;
   //Zmienna przechowująca indeks elementu prawego
  int prawy = 9;
//Dopóki indeks elementu prawego jest większy od indeksu elementu
//lewego...
 while (prawy>lewy)
```

```
//Zamieniamy elementy miejscami – wykorzystujemy funkcję swap (arg1, arg2)
   {swap(tablica[lewy], tablica[prawy]);
    lewy++;
    prawy--;}
       for (i=0; i<10; i++)
          cout<<tablica[i]<<endl;
   system("pause");
   return 0;
}
```

Program szyfrujący tekst napisany wielkimi literami metodą podstawieniową (szyfr Cezara) — zastosowanie operacji na łańcuchach

Wskazówka

Wprowadzone znaki i symbole są konwertowane na ich reprezentacje liczbowe — kod ASCII. Tabela kodów ASCII zawiera 128 liczb i odpowiadające im znaki; np. A to 65, spacja — 32.

```
Wczytujemy łańcuch znaków
cin>>tekstj;
Do pierwszego elementu dodajemy liczbę trzy
teksts=tekstj[0]+3;
Na przykład A w kodzie ASCII równa się 65. Poniższa instrukcja powinna
zwrócić 68
cout<<tekstj[0]+3<<endl;
Wartość łańcucha powinna zwrócić literę oddaloną o trzy pozycje od A,
czyli D
cout<<teksts<<endl;
```

Wykorzystamy to w programie szyfrującym. Szyfr Cezara polega na zamianie litery tekstu jawnego na inną literę, oddaloną od niej o stałą liczbę pozycji w alfabecie.

Szyfr Cezara dla wielkich liter

```
//ile – przesunięcie
#include<iostream>
#include<algorithm>
#include<string>
using namespace std;
int main()
```

```
{string tekstj,teksts;
 int ile;
 cin>>tekstj>>ile;
 for (int i=0;i<tekstj.size();i++)
 {
//W zależności od wartości zmiennej ile określona  liczba ostatnich
//znaków nie będzie miała swoich następników wśród wielkich liter.
//Poniższą operacją „zawijamy” alfabet, po Z mamy A. Dzielimy przez 26,
//bo w kodzie ASCII A ma 65, Z – 90
   teksts[i]=65+(tekstj[i]-(65-ile))%26;
    cout<<teksts[i];
 }
 cout<<teksts;
 cout<<endl;
 system("pause");
 return 0;
}
```

Otrzymany szyfrogram składa się również z wielkich liter. Spróbuj uzupełnić program tak, by można było szyfrować dowolny ciąg znaków wprowadzanych z klawiatury i w zależności od wprowadzonego znaku otrzymywać jego odpowiednik, tzn. mała litera tekstu jawnego w szyfrogramie będzie reprezentowana przez małą literę itd.

1.7. Algorytmy numeryczne i teorioliczbowe

Nie wszystkie obliczenia można wykonać przez podstawianie do wzoru lub jego przekształcenia. Istnieją przypadki, dla których wzory albo nie są znane, albo są zbyt skomplikowane lub niepraktyczne w użyciu. W takich sytuacjach mają zastosowanie metody numeryczne.

Definicja

Metody numeryczne — metody rozwiązywania problemów za pomocą operacji na liczbach.

W celu uzyskania wyniku generujemy ciąg liczb, którego kolejne wyrazy są coraz bliższe szukanego rozwiązania. Otrzymujemy w ten sposób wyniki przybliżone.

Do klasycznych algorytmów wykorzystujących metody numeryczne należą algorytmy obliczające pierwiastek kwadratowy danej liczby, miejsce zerowe funkcji oraz ograniczone krzywą pole obszaru w układzie współrzędnych.

Szczegółowy opis wymienionych przykładów znajduje się w podręczniku *Informatyka Europejczyka* dla szkół ponadgimnazjalnych — zakres rozszerzony, rozdział 2.11, „Metody numeryczne i obliczenia przybliżone".

1.7.1. Obliczanie pierwiastka kwadratowego — algorytm Newtona-Raphsona

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

$$| x_{n+1} - x_n | \leq e$$

Pierwiastek otrzymamy, obliczając kolejne wyrazy ciągu według podanego wyżej wzoru, aż uzyskamy zadaną dokładność *e*. Ostatni obliczony wyraz przyjmujemy jako przybliżoną wartość pierwiastka.

Bardzo często wprowadzamy również ograniczenia liczby iteracji.

W podanym niżej przykładzie podajemy tylko dokładność.

```
#include <iostream>
#include <cmath>
using namespace std;
int main ()
{
  double a;
  double x=1.0,xp,e;
  cin>>a>>e;
   do
   {
   xp=x;
   x=0.5*(xp+a/xp);
   }
   while (abs(x-xp)>=e);
     cout<<x<<endl;
  system ("pause");
 return 0;
}
```

Wskazówka

Dodaj do programu warunek ograniczający liczbę iteracji. Co dzięki temu zyskasz? Czym ryzykujesz, jeśli go nie wprowadzisz?

1.7.2. Obliczanie miejsca zerowego funkcji metodą połowienia przedziałów — metoda bisekcji

Aby można było znaleźć miejsce zerowe funkcji $f(x)$ w przedziale $<a, b>$, muszą być spełnione dwa warunki:

1. $f(a)*f(b) < 0$,
2. funkcja w przedziale $<a, b>$ jest ciągła.

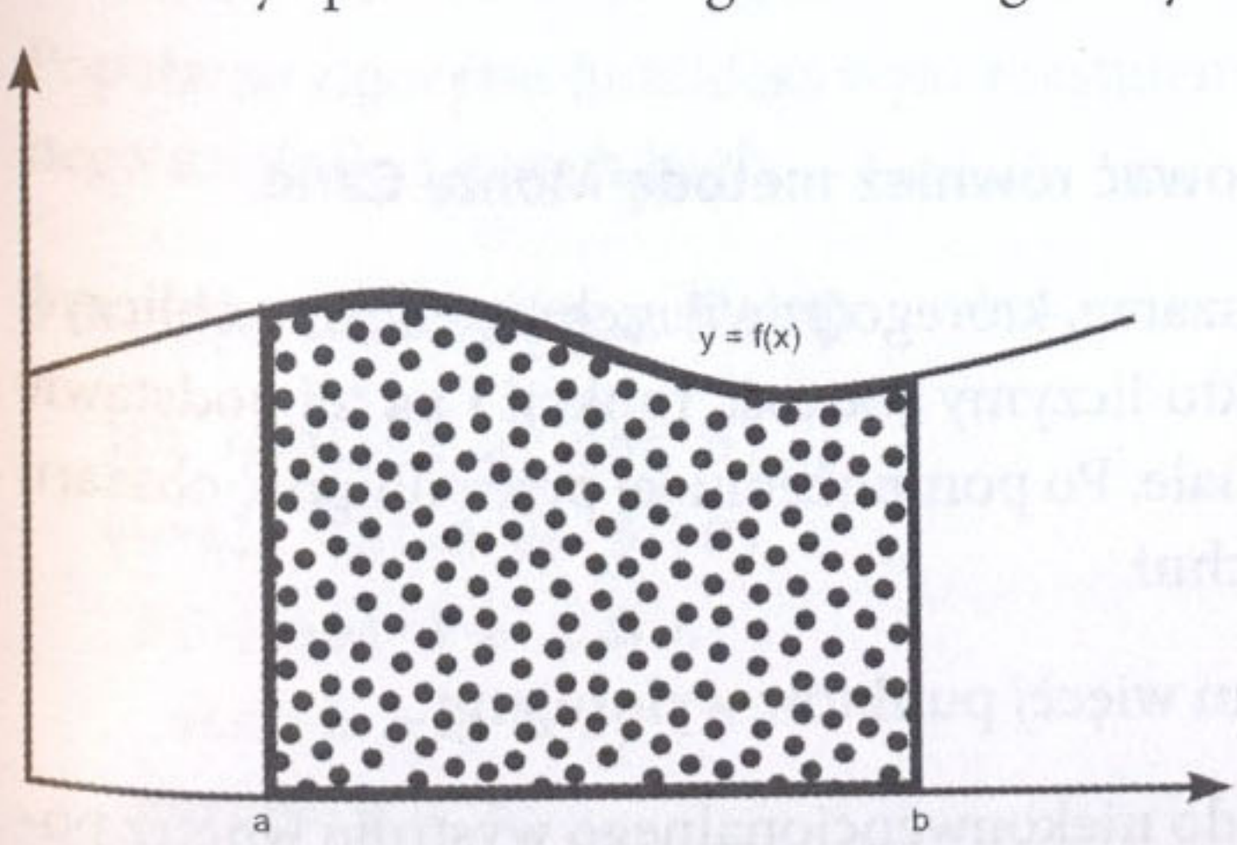
Wyznaczamy punkt środkowy przedziału $c = (a+b)/2$ i sprawdzamy, czy $f(c) = 0$. Jeżeli tak, to c jest miejscem zerowym. Jeżeli nie, sprawdzamy, czy $f(a)*f(c) < 0$. Jeżeli tak, to b przyjmuje wartość c i powtarzamy podział; w przeciwnym razie a przyjmuje wartość c i postępujemy jak wyżej.

Warunkiem zakończenia programu jest znalezienie miejsca zerowego, osiągnięcie zadanej dokładności lub określona liczba iteracji.

TP **Zadanie 1.1** Napisz funkcję, która obliczy przybliżone miejsce zerowe $f(x) = 3*x^3+2*x^2+20$ w przedziale $<0, 3>$ z dokładnością do $0{,}001$.

1.7.3. Obliczanie pola obszaru pod krzywą w zadanym przedziale w układzie współrzędnych — całkowanie numeryczne

Jak obliczyć pole obszaru ograniczonego krzywą $y = f(x)$ w przedziale $<a, b>$ (rysunek 1.1)?



Rysunek 1.1. Pole obszaru ograniczonego krzywą y = f(x)

Najczęściej spotykany algorytm to metoda prostokątów lub trapezów.

Dzielimy zadany przedział <*a*, *b*> na *n* jednakowych fragmentów. Im większe *n*, tym większa dokładność. Podzielony w ten sposób obszar obliczamy jako sumę pól prostokątów o jednym boku równym $d = (b-a)/n$ i drugim równym wartości funkcji *f(a)*, gdzie $a = a+i*d$ (*i* jest numerem kolejnego prostokąta).

W metodzie trapezów postępujemy analogicznie.

Program obliczający pole obszaru metodą prostokątów

```
//funkcja f(x)=0,05*(x-5)^2*(x+10)+15;
#include <iostream>
#include <cmath>
using namespace std;
int main()
 {
  double a,b,d,x,pole=0;
  int n;
  cin>>a>>b>>n;
  x=a;
  d=(b-a)/n;
  for (int i = 1; i <= n; i++)
  { x=a+d*i;
    pole=pole+ d*(0.05*pow((x-5),2)*(x+10)+15);
  }
 cout<<pole<<endl;
 system("pause");
 return 0;
}
```

Program oblicza pole obszaru ograniczonego krzywą w dodatnim przedziale. Zastanów się, co należy zmienić w kodzie, by obliczał również pola w przedziałach ujemnych oraz obszary znajdujące się pod osią OX.

Do obliczenia pola obszaru można zastosować również metodę Monte Carlo.

Losujemy *n* punktów znajdujących się w obszarze, którego powierzchnię chcemy obliczyć. Następnie dla każdego wylosowanego punktu liczymy wartość funkcji i na tej podstawie obliczamy średnią wartość funkcji w przedziale. Po pomnożeniu jej przez długość obszaru otrzymujemy przybliżoną wartość powierzchni.

Metoda zwraca tym dokładniejszy wynik, im więcej punktów wylosujemy.

Zadanie 1.2 Pani Ela znana ze skłonności do niekonwencjonalnego wystroju wnętrz postanowiła pomalować podłogę w swoim salonie w żółte i niebieskie pasy. Podłoga ma wy-

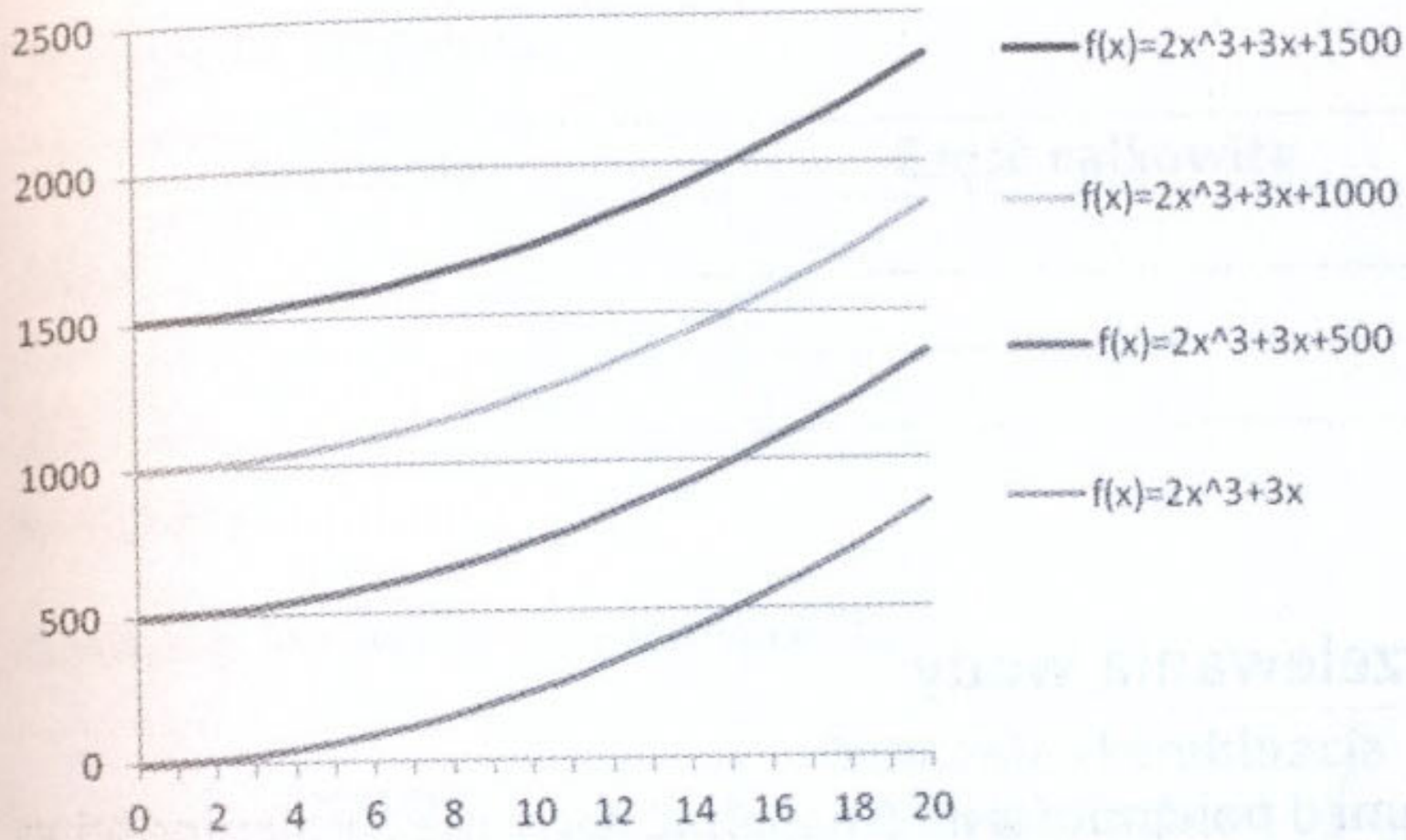
miary 20 m×25 m. Poprosiła swojego syna o przygotowanie schematu podziału podłogi. Pani Eli najbardziej spodobał się projekt przygotowany w arkuszu kalkulacyjnym (rysunek 1.2). Stosując metodę Monte Carlo, oblicz, ile litrów farby niebieskiej i żółtej potrzebuje pani Ela, jeżeli na 10 m^2 potrzeba 2 l farby, a żółtych pasów ma być więcej niż niebieskich.

Wskazówka

Do losowania punktów z zadanego przedziału <a, b> wykorzystujemy funkcję:

X = a+rand()%(a–b+1) — dla liczb całkowitych,

X = a+(double)rand()/RAND_MAX*(a–b) — dla liczb rzeczywistych.



Rysunek 1.2. Wzór podłogi

FTP **Zadanie 1.3** Piotrek narysował w układzie współrzędnych kwadrat o boku 2 jednostek, środku w punkcie (0, 0) i wpisał do niego okrąg. Następnie wybrał losowo 100 punktów w obszarze ograniczonym kwadratem. Interesuje go, czy stosunek liczby wszystkich wylosowanych punktów do punktów leżących wewnątrz okręgu jest jakąś znaną wielkością matematyczną. Niestety, Piotrek nie nauczył się jeszcze programować, a nie chce mu się liczyć… Pomóż mu rozwiązać problem.

1.7.4. Rozszerzony algorytm Euklidesa

Popularny algorytm Euklidesa wykorzystujemy do obliczania NWD (największego wspólnego dzielnika) dwóch liczb.

Funkcja obliczająca NWD

```
int NWD(int a, int b)
{while (a!=0 && b!=0)
 if (a>b) a = a % b;
  else b = b % a;
 return a + b;
}
```

Zauważmy, że jeżeli mamy parę liczb $a > b$, to zależności między nimi możemy zapisać następująco:

$a = q^*b+r$ (gdzie q jest ilorazem, a r — resztą z dzielenia),

$r = a-q^*b$,

czyli każda liczba będąca dzielnikiem a i b jest również dzielnikiem r.

NWD (a, b) = NWD (r, a)

Przykład

$a = 36, b = 14$

Możemy zapisać:

$a = 2^*b+r = 2^*14+8$

Wspólne dzielniki a i b: **2**.

Dzielniki 8: 1, **2**, 4, 8.

Jak rozwiązać problem „przelewania wody"

Sprawdźmy, jak dwoma czerpakami o pojemności a i b napełnić naczynie o pojemności v; przy czym wodę można nabierać tylko pełnymi czerpakami. Czy zawsze jest to możliwe?

Żeby rozwiązać zadanie, musimy spełnić następujący warunek:

$v = x^*a+y^*b$ (gdzie x wskazuje, ile razy użyto czerpaka a, y — czerpaka b; każda z tych liczb może być ujemna, jeżeli wodę się wylewa).

Spróbuj znaleźć rozwiązanie dla następujących przypadków:

1. $a = 3, b = 5, v = 11$

2. $a = 4, b = 6, v = 13$

3. $a = 9, b = 12, v = 15$

Czy zauważyłeś, że rozwiązanie istnieje tylko wtedy, gdy v jest równe NWD (a, b) lub jego wielokrotności?

W ten sposób przechodzimy do twierdzenia:

NWD (a, b) może być przedstawiony jako kombinacja liniowa tych liczb ze współczynnikami całkowitymi x, y.

NWD $(a, b) = x^*a+y^*b$ (równanie diofantyczne).

Definicja

Równanie diofantyczne — równanie, którego rozwiązania szuka się wśród liczb całkowitych.

Metoda wyznaczania liczb x i y nazywa się rozszerzonym algorytmem Euklidesa.

Fakt, że istnieją takie liczby, stanowi podstawę kolejnych twierdzeń i znajduje zastosowanie w kryptografii.

Rozpatrzymy przypadek dla $a = 21, b = 12$.

Oto wyniki pośrednie:

Działanie	Część całkowita	Reszta
21/12	1	9
12/9	1	3
9/3	3	0

NWD = 3

Zapiszmy te równania w ten sposób:

Reszta	Działanie (kombinacja liniowa liczb a i b)	Wartości a i b
9	= 1*21+(–1)*12	$a = 9, b = 12$
3	= 1*12+(–1)*9	$a = 9, b = 3$
0	= 1*9+(–3)*3	$a = 0, b = 3$

Aby obliczyć x i y, wystarczy dokonać właściwych przekształceń:

$3 = 1*12+(-1)*9 = 1*12+(-1)*(1*21+(-1)*12) = 1*12+(-1)*(21-12) = (-1)*21+2*12$

$x = -1, y = 2$

Rozszerzony algorytm Euklidesa — metoda iteracyjna

```
#include <iostream>
using namespace std;
int main ()
{int a, b;
 //Zakładamy, że a > b
 cin>>a>>b;
 int a0 = a;
 int b0 = b;
  //x*a0+y*b0 = a oraz r*a0+y*b0 = b
 int x=0, x0 = 1, y = 1, y0=0, r, pom;
```

```
//Algorytm
 while (b0 != 0)
 { r = a0 / b0;
   pom = b0;
   b0=a0-r*b0;
   a0=pom;
   pom=x0;
   x0=x- r*x0;
   x=pom;
   pom=y0;
   y0=y-r*y0;
   y=pom;
 }
 cout<<a<<endl;
 cout<<x<<endl;
 cout<<y<<endl;
 system ("pause");
 return 0;
}
```

FTP **Zadanie 1.4** Funkcja NWD jest wykorzystywana przy skracaniu ułamków zwykłych. Napisz program mnożący ułamki i zwracający wynik w postaci ułamka zwykłego, ale tak, żeby uniknąć operacji na dużych liczbach.

FTP **Zadanie 1.5** Kasia bardzo dba o higienę swoich piesków. Zaplanowała kąpiel zwierząt na podwórku w misce o pojemności 25 l. Wodę musi przynosić z domu w wiaderkach. Ma do dyspozycji zielone wiaderko o pojemności 3 l i czerwone o pojemności 2 l. Zastanawia się, ile razy powinna przynieść wodę w czerwonym wiaderku, a ile w zielonym, żeby jak najmniej chodzić i nie wylewać wody. Pomożesz jej?

1.7.5. Arytmetyka modularna

O dwóch liczbach całkowitych a i b mówimy, że są równoważne (przystają) modulo m (gdzie m jest liczbą naturalną różną od zera), jeżeli $a\%m = b\%m (a = b(\%m))$.

Dla dodatnich a i b: $a = b(\%m)$, jeżeli a i b mają takie same reszty z dzielenia przez m.

Przykłady

$5 = 11(\%3)$

$5 = 0(\%5)$

$14 = -16(\%5)$

Wskazówka

Zwróć uwagę na ostatni przykład:

$14\%5 = 4$

$-16\%5 = 4$

Zgodnie z regułami matematyki reszta z dzielenia musi być dodatnia.

Ponieważ nie wszystkie kompilatory języka uwzględniają tę regułę, bardzo często przy obliczeniach reszty z dzielenia należy uwzględniać warunek:

```
#include <iostream>
using namespace std;
int main ()
{
 int x, k, reszta;
 cin >> x;
 cin >> k;
 reszta=x%k;
 if (reszta<0) reszta=k+reszta;
 cout<<reszta<<endl;
 system ("pause");
 return 0;
}
```

W przypadku gdy d i m są względnie pierwsze (nie mają wspólnych dzielników oprócz jedynki):

$(a*d)\%m = (b*d)\%m$, to $a\%m = b\%m$.

Ostatnia cyfra ($a^x\%10$)

Przy wypisywaniu ostatniej cyfry liczby naturalnej podniesionej do potęgi x zauważamy, że zależy ona tylko od ostatniej cyfry liczby a. Rozpatrzmy to dla dwójki:

$122^3 = 1\,815\,848$ — ostatnia cyfra 8.

$27852^3 = 21\,605\,740\,694\,208$ — ostatnia cyfra 8.

$1862^3 = 6\,455\,635\,928$ — ostatnia cyfra 8.

Zauważamy jeszcze jedną własność — ostatnie cyfry powtarzają się cyklicznie:

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$2^5 = 32$

$2^6 = 64$

$2^7 = 128$

$2^8 = 256$

W przypadku dwójki widzimy, że są to liczby: 2, 4, 8, 6 (okres 4).

Wykorzystując te zależności, możemy napisać funkcję, która zwraca ostatnią cyfrę liczby mającej 2 na pozycji jedności podniesionej do dowolnej potęgi *x*:

```
int ostatnia_cyfra (int x)
  {if (x%4==1) return 1;
   if (x%4==2) return 4;
   if (x%4==3) return 8;
   if (x%4==0) return 6;
  }
```

Powyższe zasady dotyczą wszystkich pozostałych cyfr. Dla każdej wystarczy sprawdzić, jakie cyfry się powtarzają i z jaką częstotliwością.

FTP **Zadanie 1.6** Napisz program, który wczytuje liczbę *n* i wypisuje ostatnią cyfrę liczby 9^n.

FTP **Zadanie 1.7** Rodzice Jasia lubią łamigłówki. Rzadko spełniają prośby syna, jeśli uprzednio nie rozwiąże on jakiegoś problemu matematycznego. Gdy chłopiec poprosił o pieniądze na nową grę komputerową, mama powiedziała, że dostanie je, jeżeli weźmie najmniejszą liczbę pierwszą, podniesie ją do potęgi kolejnej liczby pierwszej, otrzymany wynik podniesie do kolejnej liczby pierwszej i zrobi tak 10 razy. Mama chciała wiedzieć, jaka jest ostatnia cyfra otrzymanej liczby. Jasio usiadł przed komputerem i po 10 minutach powiedział mamie, że to 8. Czy mógł sobie kupić grę?

1.7.6. Chińskie twierdzenie o resztach (jedno z najważniejszych w teorii liczb i kryptografii)

W I wieku naszej ery chińscy generałowie wymyślili prosty sposób sprawdzania liczebności swoich żołnierzy. Żołnierzom wydawano rozkaz ustawienia się w kolumnach 3-, 5-, 7-osobowych i liczono tylko niepełną kolumnę (resztę z dzielenia). Przy kolejnym liczeniu wystarczyło porównać otrzymane reszty, by stwierdzić, czy liczba żołnierzy uległa zmianie.

Niech $n1$, $n2$, ..., nk będą dodatnimi liczbami względnie pierwszymi (dla każdej pary NWD wynosi 1) oraz niech $y1$, $y2$, ..., yk będą dowolnymi resztami. Wtedy istnieje dokładnie jedna liczba całkowita x spełniająca układ równań:

$a1 = x\ (\%n1)$

$a2 = x\ (\%n2)$

...

$ak = x\ (\%nk)$

gdzie x jest większe lub równe 1 i mniejsze lub równe nk.

Algorytm wyliczania x:

- $N = n_1 * \ldots * n_k$
- $N_i = N/n_i$
- $N_i * x_i = 1(\%n_i)$ (Korzystamy z rozszerzonego algorytmu Euklidesa).
- $x = \sum_{i=1}^{k} y_i * x_i * N_i (\%N)$

Przykład

Rozwiążmy układ równań:

$x = 3(\%10)$

$x = 4(\%11)$

$x = 5(\%7)$

- Sprawdzamy warunek, czy liczby są parami względnie pierwsze: NWD(10, 11) = 1, NWD(10, 7) = 1, NWD(11, 7) = 1.
- Obliczamy $N = 10*11*7 = 770$.
- $N_1 = 770/10 = 77$, $N_2 = 770/11 = 70$, $N_1 = 770/7 = 110$.
- Rozwiązujemy równania $N_i * x_i = 1\%n_i$

$x_1 * 77 = 1(\%10)$	$77\%10 = 7$	$7 * x_1 = 1(\%10)$	**$x1 = 3$**
$x_2 * 70 = 1\ (\%11)$	$70\%11 = 4$	$4 * x_2 = 1(\%11)$	**$x2 = 3$**
$x_3 * 110 = 1(\%7)$	$110\%7 = 5$	$5 * x_3 = 1(\%7)$	**$x3 = 3$**

Wskazówka

Wartości x obliczamy metodą podstawiania kolejnych liczb o najmniejszej wartości bezwzględnej.

$x = (3*3*77+4*3*70+5*3*110)\%770 = 103$

Sprawdzamy:

$x = 3(\%10)$

$103\%10 = 3\%10$

$x = 4$ (%11)

103%11 = 4%11

$x = 5$(%7)

103%7 = 5%7

1.7.7. Potęgowanie modularne

Jeżeli chcemy obliczyć resztę z dzielenia a^x przez b i a^x jest bardzo dużą liczbą, możemy zastosować wzór:

$a^x\%b = (...(((a\%b)^*a\%b)^*a\%b)...^*a\%b)$ — wykonujemy x mnożeń.

Jeżeli $a > b$, to najpierw obliczamy $a\%b$, a potem podnosimy do potęgi.

Przykład

Obliczmy resztę z dzielenia 17^4 przez 5.

Najpierw obliczamy a%b = 17%5 = 2.

1.	1*2 = 2	2%5 = 2
2.	2*2 = 4	4%5 = 4
3.	4*2 = 8	8%5 = 3
4.	3*2 = 6	6%5 = 1

Wynik: $17^4\%5 = 1$.

Powyższy algorytm pozwala uniknąć bardzo dużych liczb, ale wykonujemy w nim x operacji mnożenia i tyle samo operacji dzielenia. Co jednak zrobić w przypadku, gdy sam wykładnik jest bardzo duży, jak np. w algorytmach kryptograficznych?

Zanim przejdziemy do algorytmu szybkiego potęgowania, zastanów się:

- Jak obliczyć potęgę o wykładniku, który jest potęgą dwójki?
- Czy każda liczba naturalna jest sumą potęg dwójki?

Algorytm szybkiego potęgowania

Wykorzystując spostrzeżenie, że każda liczba naturalna jest potęgą dwójki, i wartość potęgi, której wykładnik j jest potęgą dwójki, wykonujemy:

```
y = podstawa;
for (int i = 1; i <=j; i++)
   y=y*y;
```

j — wykładnik potęgi dwójki (2^j).

W algorytmie szybkiego potęgowania wykładnik jest w postaci binarnej.

Algorytm szybkiego potęgowania dla wykładnika równego 5 (w systemie binarnym 101)

```
#include <iostream>
using namespace std;
int main()
{int dl, p, y=1;
 int wyk[3]={1,0,1};
 cin>>p;
 if (wyk[0]==1) y=y*p;
 for (int i = 1; i <3; i++)
 { p=p*p;
   if (wyk[i]==1) y=y*p;
 }
 cout<<y<<endl;
system("pause");
 return 0;
}
```

Przykład wykorzystania algorytmu do obliczania reszty z dzielenia

Obliczmy $r = 17^{23}\%5$.

a = 17, b = 5, czyli a > b, więc przypisujemy *a* resztę z dzielenia: a = a%b = 17%5 = 2.

- Zamieniamy 23 na postać binarną: $23 = (10111)_B$.
  Będziemy musieli wymnażać tylko te bity, których wartość wynosi 1 (każda wartość podniesiona do potęgi 0 daje 1).
- Na początku *r* przypisujemy wartość 1.
  $b_0 = 1\ r = (r^*a)\%b = 1^*2\%5 = 2a = a^*a\%b = 2^*2\%5 = 4$
  $b_1 = 1\ r = (r^*a)\%b = 2^*4\%5 = 3a = a^*a\%b = 4^*4\%5 = 1$
  $b_2 = 1\ r = (r^*a)\%b = 3^*1\%5 = 3a = a^*a\%b = 1^*1\%5 = 1$
  $b_3 = 0\ r = 3a = a^*a\%b = 1^*1\%5 = 1$
  $b_4 = 1\ r = (r^*a)\%b = 3^*1\%5 = 3$

FTP **Zadanie 1.8** Stosując powyższą metodę, oblicz resztę z dzielenia 12^{53} przez 7.

FTP **Zadanie 1.9** Napisz program, który dla *n* (1 <= *n* <= 1 000 000 000) wypisuje resztę z dzielenia *n*! przez 1 000 000 000.

Małe twierdzenie Fermata i funkcja Eulera

Fermat był dosyć ekscentrycznym matematykiem, bo podawał swoje twierdzenia... bez dowodów, pozostawiając innym matematykom dowodzenie poprawności tych twierdzeń lub ich zaprzeczenia.

Małe twierdzenie Fermata

Dla dowolnej liczby pierwszej p i dowolnego a:

$$a^p \% p = a \% p$$

Przykład

$a = 3\ p = 2$

$a^p \% p = 3^2 \% 2 = 9 \% 2 = 1$

$a \% p = 3 \% 2 = 1$

$a^p \% p = a \% p$

FTP **Zadanie 1.10** Napisz program, który wczytuje liczbę n i wypisuje ostatnią cyfrę liczby 2^n.

Wejście:

Liczba naturalna $1 \leq n \leq 30\,000$.

Wyjście:

Ostatnia cyfra 2^n.

Przykład:

Dla danych wejściowych:

8

Poprawna odpowiedź:

6

FTP **Zadanie 1.11** Kasia ma 5 lat i zbiera kolorowe, szklane kuleczki. Dziewczynka dostała od starszego brata pięć jednakowych pudełeczek i chciałaby rozmieścić kuleczki tak, żeby w każdym pudełku było ich tyle samo. Niestety, umie liczyć tylko do pięciu i nie potrafi dzielić. Najpierw układała kuleczki, wkładając po kolei jedną do każdego pudełeczka. Zostały jej trzy. Spróbowała do dwóch — została jej jedna. W przypadku trzech też została jedna. Kasia zastanawia się teraz, czy uda jej się podzielić kuleczki równo na cztery pudełka. A Ty jak myślisz?

Wejście:

Pary liczb:

5 3

2 1

3 1

Wyjście:

TAK — jeżeli kuleczki można podzielić równo na cztery pudełka.

NIE — w przeciwnym wypadku.

TP **Zadanie 1.12** Kasia, Ania i Janek dostali trzy identyczne czekolady. Każde dziecko wzięło swoją do szkoły, żeby podzielić się z przyjaciółmi. Najwięcej koleżanek ma Kasia — aż sześć. Po równym podziale czekolady pomiędzy wszystkie dziewczynki zostały dwie kostki. Janek podzielił się z dwoma kolegami i została mu jedna kostka. Ani zostały trzy kostki po podzieleniu się z czterema koleżankami. Po ile kostek miały czekolady?

Funkcja Eulera

Funkcja φ (Eulera) lub tocjent — funkcja nosząca nazwisko Eulera przypisująca każdej liczbie naturalnej ilość liczb względnie z nią pierwszych, nie większych od niej samej (liczby względnie pierwsze to takie, które nie mają wspólnych dzielników oprócz jedynki).

$$\varphi(n) = n*(1-\frac{1}{p_1})*(1-\frac{1}{p_2})*\ldots*(1-\frac{1}{p_1 k}),$$

gdzie p_1, p_2, ..., p_k są czynnikami (czynnik pierwszy — dowolna liczba pierwsza dzieląca daną liczbę) liczby n (bez powtórzeń).

Przykład

$\varphi(4) = 4*(1-1/2) = 2$	Są to liczby: 1, 3.
$\varphi(6) = 6*(1-1/2)*(1-1/3) = 2$	Są to liczby: 1, 5.

FTP **Zadanie 1.13** Napisz program, który zwraca resztę z dzielenia x^m przez $n <= 20$, gdzie x i n są względnie pierwsze.

Dane wejściowe:

x, m, n

Wynik:

r — reszta z dzielenia x^m przez n.

Przykładowy rezultat:

Dla danych $x = 13$, $m = 101$, $n = 16$ program zwraca 13.

Mnożenie rosyjskich chłopów

Na zakończenie algorytmów numerycznych coś prostego, czyli mnożenie... bez mnożenia. To algorytm znany pod nazwą mnożenia po etiopsku lub mnożenia rosyjskich chłopów. Sprawdza się przy działaniu z dużymi liczbami.

Algorytm

Jeżeli chcemy pomnożyć *a* przez *b*, przypisujemy na początku wynikowi wartość 0 i wykonujemy następujące kroki tak długo, aż *b* wyniesie 0.

1. Jeżeli *b*%2! = 0, to do wyniku dodajemy *a*.
2. Mnożymy *a* przez 2.
3. Dzielimy *b* przez 2 (dzielenie całkowitoliczbowe).

```
#include <iostream>
using namespace std;
int main()
{ int a,b,wynik=0;
  cin>>a>>b;
  while (b!=0)
  {if (b%2!=0) wynik+=a;
    a=2*a;
    b=b/2;
  }
 cout<<wynik<<endl;
  system("pause");
return 0;
}
```

Proste, prawda? I działa! Czy potrafisz wytłumaczyć dlaczego?

1.8. Algorytmy rekurencyjne

„Należy najpierw zrozumieć rekurencję, żeby zrozumieć rekurencję."

— Anonim

Definicja

Rekurencją nazywamy metodę, w której definiując procedurę lub funkcję, odwołujemy się do niej samej.

Brzmi to trochę jak pleonazm, ale w rzeczywistości wygląda inaczej.

W celu lepszego zrozumienia rekurencji warto zajrzeć do podręcznika *Informatyka Europejczyka* dla szkół ponadgimnazjalnych — zakres rozszerzony, część 1, rozdział 1.6, „Rekurencja".

Definiując problem rekurencyjnie, musimy podać wartość brzegową (początkową) i równanie wyrażające pozostałe wartości za pomocą wcześniej obliczonych elementów.

Przykład

Obliczanie silni liczb naturalnych

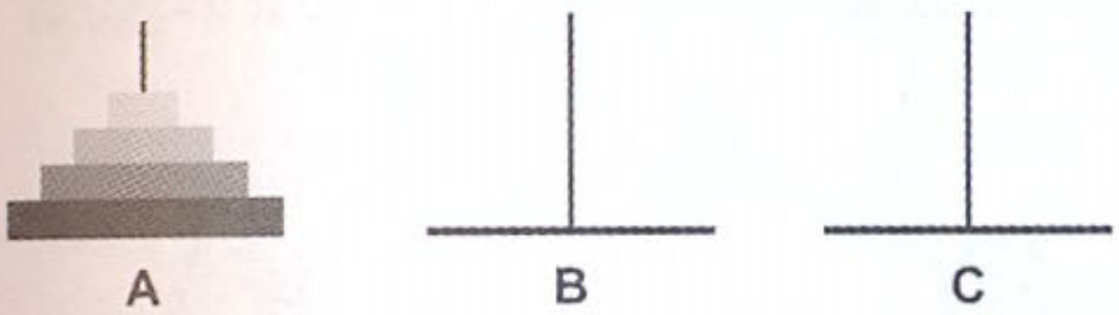
Dla $n <= 1$ silnia(n) = 1 (wartość brzegowa).

Dla $n > 1$ silnia(n) = silnia(n–1)*n (pozostałe wartości).

Rekurencyjne rozwiązywanie problemu sprowadza się na ogół do rozwiązania tego samego problemu, tyle że dla danych o mniejszych wartościach.

Przeanalizujmy pod tym kątem klasyczną łamigłówkę — wieże Hanoi.

Naszym zadaniem jest przeniesienie wszystkich krążków z palika A na palik C z wykorzystaniem palika B w taki sposób, by większy krążek nigdy nie znalazł się na mniejszym. Staramy się wykonać jak najmniej ruchów (rysunek 1.3).



Rysunek 1.3. Wieże Hanoi

Proponuję rozwiązać tę łamigłówkę z wykorzystaniem symulacji dostępnych w internecie lub przygotowanego modelu (albo wieży z klocków, którą bawi się młodsze rodzeństwo ☺) i zapisywać minimalną liczbę ruchów, które można wykonać przy każdym przeniesieniu elementów.

Zgodnie z zasadą rekurencji zaczynamy od minimalnej liczby krążków: n–3. Następnie 4, 5 itd.

Czy zauważyłeś, na czym polega optymalna strategia? Potrafisz zapisać matematycznie związek między otrzymanymi wynikami a liczbą krążków? Mamy tu do czynienia z drugą zasadą rozwiązywania problemów rekurencyjnych — próbujemy otrzymane wyniki zapisać za pomocą wzoru matematycznego i sprawdzamy, czy jest on prawdziwy dla wszystkich przypadków.

Przyjmujemy numerację: A — 0, B — 1, C — 2.

Optymalna strategia dla parzystej liczby krążków

Krążki o indeksach nieparzystych przesuwają się w prawo, wracając skokiem z ostatniego palika {1, 2, 0, 1, 2, ...}, a krążki o indeksach parzystych odwrotnie — najpierw skaczą na ostatni palik, potem wracają w lewo {2, 1, 0, 2, 1, …} (to oznacza, że najmniejszy obiekt z indeksem 1 rozpoczyna, przechodząc z palika A na środkowy palik B).

Optymalna strategia dla nieparzystej liczby krążków

Krążki o indeksach nieparzystych najpierw skaczą na ostatni palik, potem wracają w lewo {2, 1, 0, 2, 1, …}, a krążki o indeksach parzystych odwrotnie — krążą w prawo i wracają skokiem z ostatniego palika {1, 2, 0, 1, 2, …}.

Czy zauważyłeś, że liczbę ruchów można wyrazić iteracyjnie: 2^n-1?

Algorytm rekurencyjny:

- przenosimy (rekurencyjnie) $n-1$ krążków z A na B, wykorzystując C;
- przenosimy n-ty krążek z A na C;
- przenosimy (rekurencyjnie) $n-1$ krążków z B na C, wykorzystując A.

Teraz wystarczy napisać prosty program i na pozór skomplikowane zadanie będzie rozwiązane!

```
#include <iostream>
using namespace std;
void wieze_hanoi(int n, char A, char B, char C)
{
  if (n>0)
  { wieze_hanoi(n-1, A, C, B);
    cout << A << " -> " << C << endl;
    wieze_hanoi(n-1, B, A, C);
 }
}
int main()
{int n;
 cin>>n;
 wieze_hanoi(n, 'A1', 'B2', 'C3');
 system ("pause");
 return 0;
}
```

Z wieżami Hanoi jest związana tybetańska legenda, której autorstwo przypisuje się matematykowi Lucasowi. Według niej mnisi w świątyni Brahmy rozwiązują zagadkę z 64 krążkami (ze szczerego złota!), przestrzegając niezłomnych (znanych nam już) zasad. Kiedy ją rozwiążą, nastąpi koniec świata ☺

Tym, którzy się zastanawiają, czy koniec świata nastąpi przed klasówką z matematyki i czy opłaca się w związku z tym uczyć, czym prędzej podpowiadam: po podstawieniu danych do wzoru ułożenie wieży zajmie 18 446 744 073 709 551 615 sekund, czyli około 584 542 miliardów lat. Klasówki nie da się uniknąć...

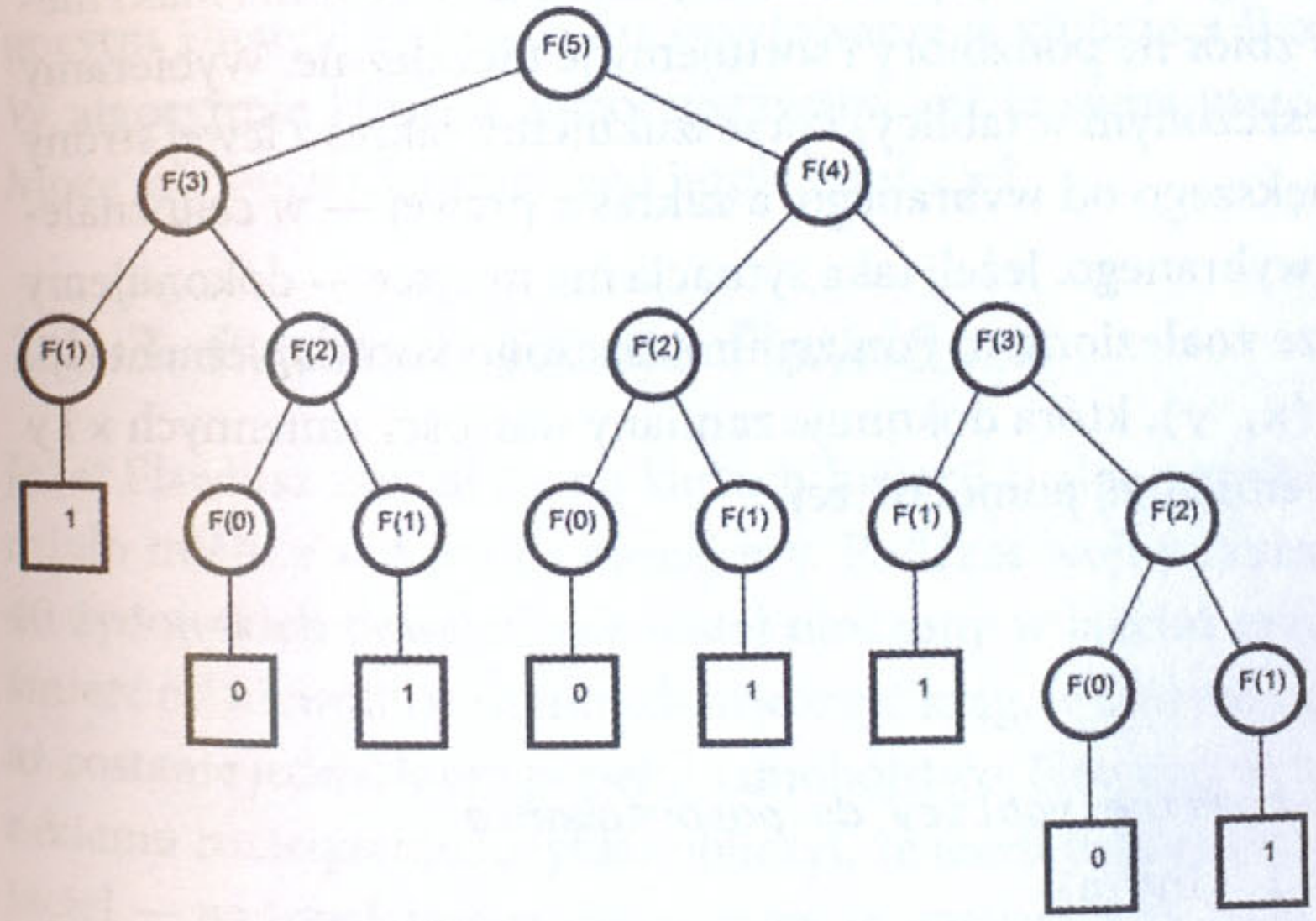
1.8.1. Ciąg Fibonacciego

Rekurencja jest stosowana w wielu algorytmach, ale nie zawsze rozwiązanie rekurencyjne jest najefektywniejsze. Możesz się o tym przekonać, porównując czas działania funkcji obliczającej n-ty wyraz ciągu Fibonacciego (jest to ciąg liczb naturalnych, w którym kolejne wyrazy ciągu są wyrażone wzorem: $F_0 = 0, F_1 = 1, F_n = F_{n-1}+F_{n-2}$).

```
//Funkcja obliczająca n-ty wyraz ciągu metodą rekurencyjną
long long FibR (int n)
{
 if (n<2)
  return 1;
 else return FibR(n-1)+FibR(n-2)
}
```

TP **Zadanie 1.14** Napisz program, w którym n-ty wyraz ciągu jest obliczany metodą iteracyjną i rekurencyjną. Porównaj czas działania obu metod dla $n = 10$.

Różnice w czasie działania pomoże zrozumieć schemat przedstawiony na rysunku 1.4.



Rysunek 1.4. Schemat wywołań rekurencyjnych funkcji Fibonacci dla n = 5

Czy po analizie schematu potrafisz uzupełnić poniższą tabelkę?

n	Fib(n+1)	Liczba dodawań	Liczba wywołań
4	5		
5	8	7	15
10			
20			
30			
50			

FTP **Zadanie 1.15** Mamy 50 wylosowanych liczb z zakresu od 1 do 1000. Jak, stosując rekurencję, możemy znaleźć wśród nich podaną wartość?

Wskazówka

Posortuj tablicę i zastosuj wyszukiwanie binarne.

FTP **Zadanie 1.16** Oblicz sumę *n* początkowych wyrazów następującego ciągu 1+1/2–1/3+1/4–1/5+…+1/*n*.

Zadanie 1.17 Na ile sposobów można ułożyć kostki domina na prostokącie o rozmiarze 2 na *n*?

1.8.2. QuickSort

QuickSort jest uważany za najszybciej działający algorytm sortowania losowych danych. Wykorzystuje znaną już z innych algorytmów (np. szukania jednoczesnego min i max) metodę dziel i zwyciężaj. Dzielimy zbiór na podzbiory i sortujemy je niezależnie. Wybieramy element w ciągu (najlepiej umieszczonym w tablicy) i przeszukujemy zakres z lewej strony w celu znalezienia elementu większego od wybranego, a zakres z prawej — w celu znalezienia elementu mniejszego od wybranego. Jeżeli taka sytuacja ma miejsce — dokonujemy zamiany wybranego elementu ze znalezionym. Poniżej umieszczono kod implementacji. Zwróć uwagę na funkcję `swap (x, y)`, która dokonuje zamiany wartości zmiennych `x` i `y` bez konieczności deklarowania zmiennej pomocniczej.

```
#include <iostream>
using namespace std;
int tab[10000000];
//Deklarujemy maksymalny rozmiar tablicy do posortowania
void qsort(int tab[],int l, int p)
{
 if (l<p)
 {
  int s=l;
  for(int i=l+1;i<=p;i++)
  if (tab[i]<tab[l])
//tab{++s] – inkrementacja prefiksowa – zwiększa wartość zmiennej o 1
//oraz zwraca jej nową wartość
swap(tab[++s],tab[i]);
swap(tab[l],tab[s]);
qsort(tab,l,s-1);
```

```
 qsort(tab,s+1,p);
 }
}
int main()
{int r;
 cin>>r;
 for (int i=0;i<r;i++)
 cin>>tab[i];
 qsort(tab,0,r-1);
 for (int i=0;i<r;i++)
  cout << tab[i] <<"  ";
 cout << endl;
 system("pause");
 return 0;
}
```

Twórcą tego algorytmu jest sir Charles Antony Richard Hoare. Opracował on również algorytm Hoare'a —służący do znajdowania w zbiorze *n* liczb liczby *k*-tej co do wielkości. W algorytmie Hoare'a wykorzystywana jest ta sama metoda co w sortowaniu szybkim. Może spróbujesz napisać jego implementację?

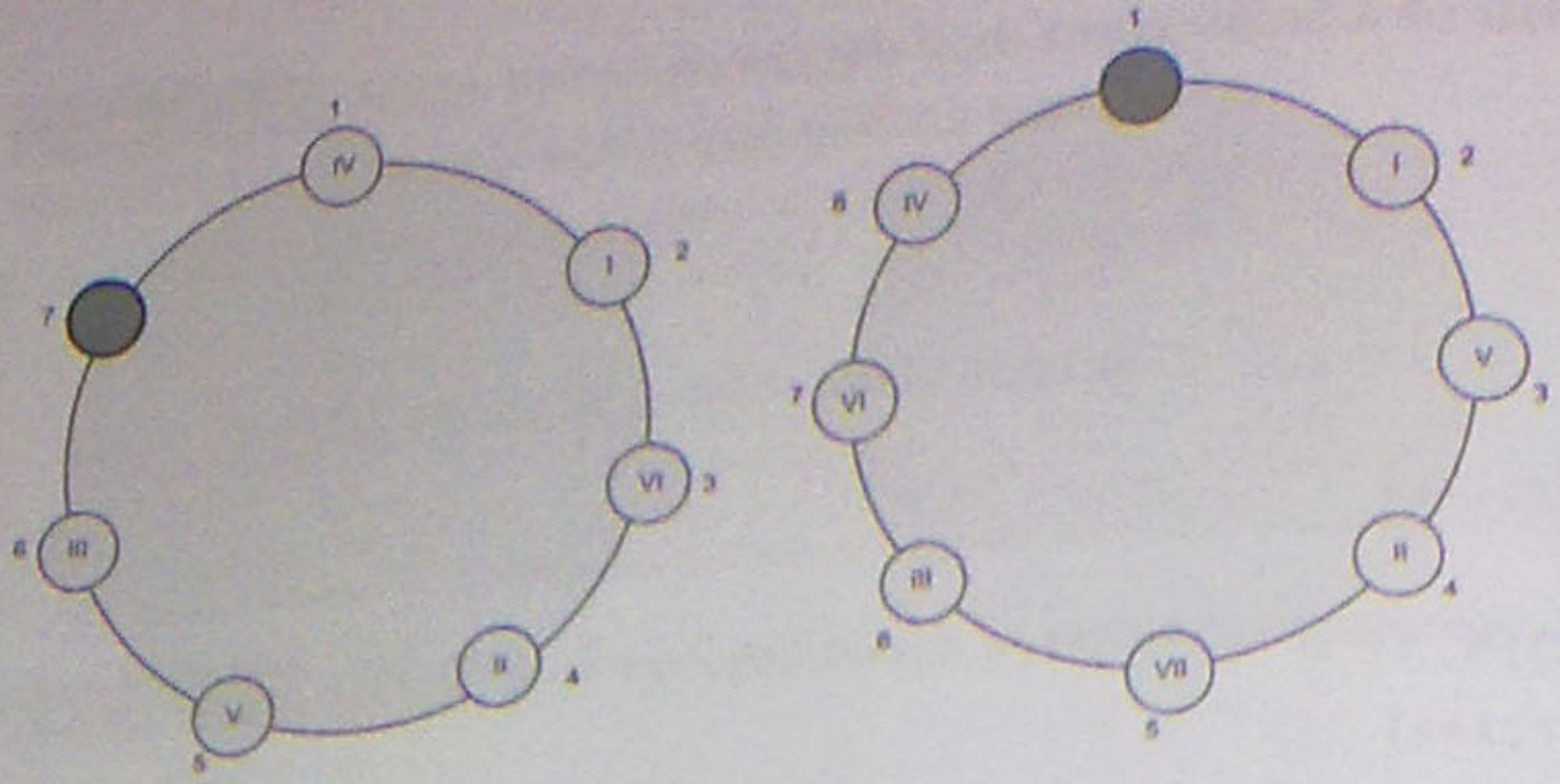
1.8.3. Problem Józefa Flawiusza

Józef Flawiusz zapisał się na kartach historii (i algorytmiki) w wyniku wydarzenia, które miało miejsce w I wieku naszej ery. Podczas wojny rzymsko-żydowskiej wraz z grupą 40 żydowskich powstańców został otoczony w jaskini przez Rzymian. Powstańcy woleli śmierć od niewoli i postanowili utworzyć krąg, w którym będzie zabijana co trzecia osoba, aż zostanie jedna, która popełni samobójstwo. Flawiusz wraz z przyjacielem był przeciwny takiemu rozwiązaniu. Szybko obliczył, że jeżeli ustawi się na szesnastym miejscu, a przyjaciel — na trzydziestym pierwszym, to zostaną jako dwaj ostatni i będą mieli szansę na przeżycie, co też się stało.

Definicja

Problem wskazania ostatniego elementu, który pozostanie z *n*-elementowego okręgu po wyeliminowaniu co *k*-tego, nazywa się **problemem lub permutacją Józefa Flawiusza**.

Na rysunku 1.5 przedstawiono analizę tego problemu dla parzystej (osiem) i nieparzystej (siedem) liczby elementów dla k = 2.



Rysunek 1.5. Permutacja Flawiusza

Liczbami arabskimi oznaczono pozycje, a rzymskimi — kolejność eliminowania. Ciemnoszarym kolorem zostały zaznaczone elementy, które „przeżyły".

Na okręgu z lewej strony mamy nieparzystą liczbę elementów — 7:

Kolejność eliminowania: 2, 4, 6, 1, 5.

Okręg z prawej strony ma parzystą liczbę elementów — 8:

Kolejność eliminowania: 2, 4, 6, 8, 5.

FTP **Zadanie 1.18** Wypełnij poniższą tabelę, w której n oznacza liczbę elementów w kole, a $L(n)$ jest numerem ostatniego elementu. Otrzymane wyniki spróbuj zapisać wzorem matematycznym. Sprawdź, czy jest on prawdziwy również dla dużych n. Zapisz otrzymany wzór rekurencyjnie i wykorzystaj go w funkcji `flawiusz(n)`.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$L(n)$																

1.9. Struktury danych

Wiemy już, że warunkiem pomyślnego rozwiązania problemu jest właściwy algorytm. Nie mniej ważna jest kwestia jego implementacji. Na etapie projektowania istotny jest również wybór odpowiedniej struktury danych, która pozwoli skrócić czas wykonywania programu albo uprościć (czasami wręcz umożliwić) jego zapis.

Tymczasem zajmiemy się praktycznym zastosowaniem takich struktur jak stos, kolejka, kopiec, drzewo licznikowe i potęgowe.

Opis wybranych dynamicznych struktur danych znajduje się w podręczniku *Informatyka Europejczyka* dla szkół ponadgimnazjalnych — zakres rozszerzony, część 1, rozdział 3.7, „Dynamiczne struktury danych".

1.9.1. Drzewa

Skoro o informatykach opowiada się anegdoty, że na prośbę o pożyczenie 1000 zł proponują 1024 dla równego rachunku, nie powinno budzić zdziwienia to, że informatyczne drzewa rosną… korzeniami do góry ☺

Co to jest drzewo?

Definicja

Drzewo jest to struktura reprezentująca drzewo matematyczne — obiekty (węzły) ułożone hierarchicznie. Węzeł nadrzędny jest nazywany korzeniem, węzły podrzędne są liśćmi (synami).

Podstawowe pojęcia

Ścieżka — ciąg krawędzi łączących węzły.

Stopień wierzchołka — liczba wychodzących z niego krawędzi.

Wysokość węzła — długość najdłuższej ścieżki prowadzącej od węzła do liści.

Głębokość — długość ścieżki łączącej węzeł z korzeniem.

Rozmiar drzewa — liczba węzłów.

Strukturę drzewiastą, w której każdy węzeł może mieć maksymalnie dwóch potomków (lewego i prawego syna), nazywamy drzewem binarnym.

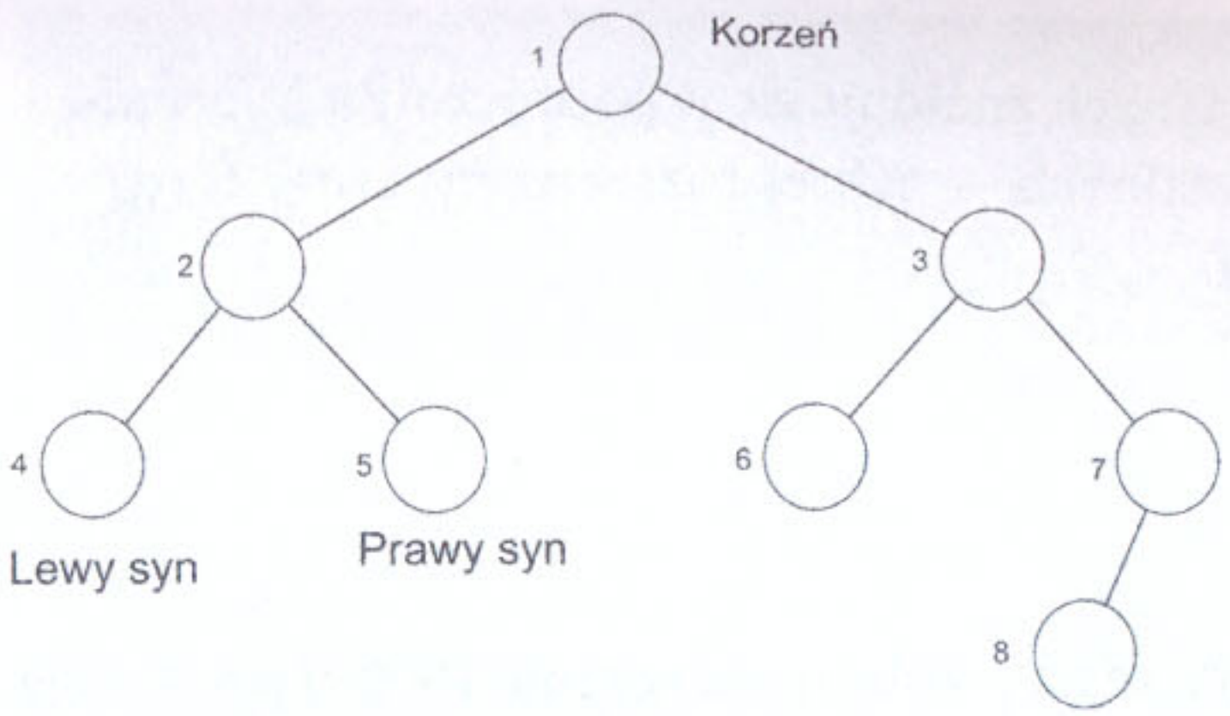
W drzewie binarnym węzły potomne i węzeł nadrzędny dla węzła k mają indeksy:

lewy syn — $2k$,

prawy syn — $2k+1$,

nadrzędny — $k/2$ (część całkowita z dzielenia).

Jaki rozmiar i wysokość ma drzewo przedstawione na rysunku 1.6?



Rysunek 1.6.
Drzewo binarne

Przechodzenie drzewa

Pre-order: przejście wzdłuż

Korzeń — lewe poddrzewo — prawe poddrzewo

1, 2, 4, 5, 3, 6, 7, 8

In-order: poprzeczne

Poddrzewo lewe — korzeń — poddrzewo prawe

4, 2, 5, 1, 6, 3, 8, 7

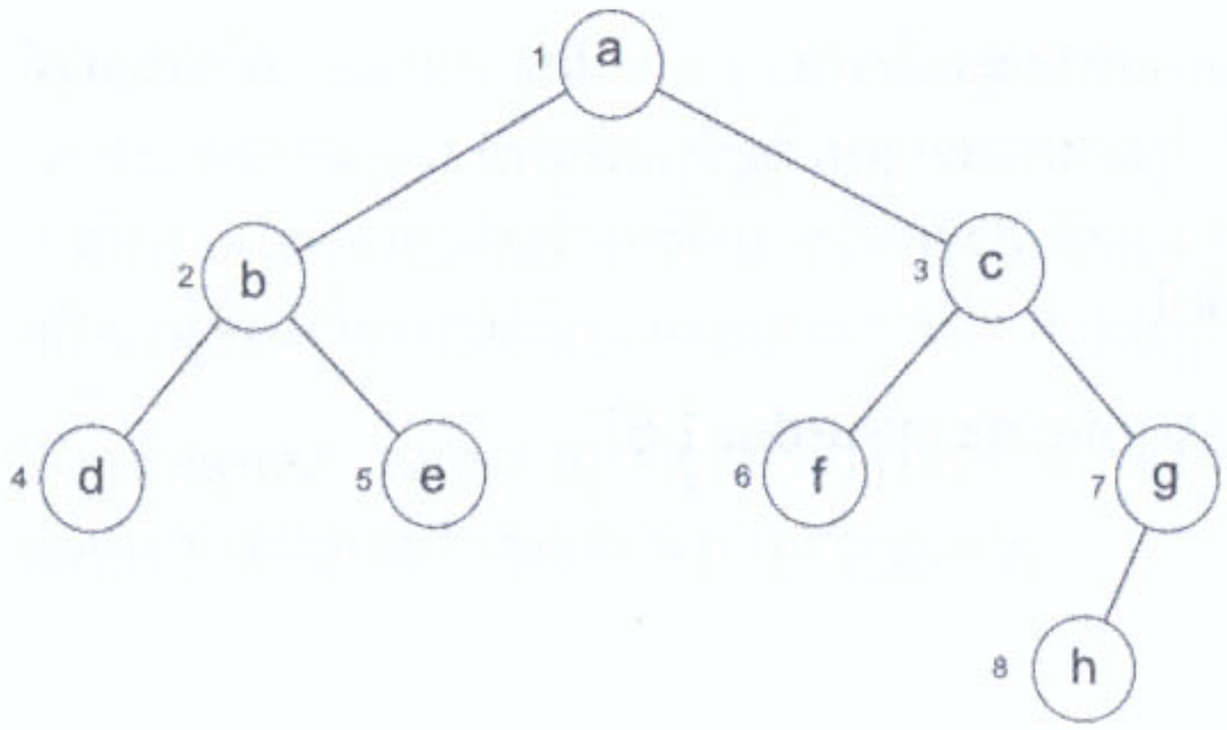
Post-order: wsteczne

Poddrzewo lewe — poddrzewo prawe — korzeń

4, 5, 2, 6, 8, 7, 3, 1

Umiejętność „chodzenia po drzewach" przyda nam się przy korzystaniu z algorytmów grafowych.

Do implementacji drzew binarnych w komputerze najprościej jest zastosować tablice. Zapiszemy w tablicy drzewo z rysunku 1.7.



Rysunek 1.7.
Drzewo binarne

Oto implementacja tablicowa drzewa.

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*

Zadanie 1.19 Czy potrafisz na podstawie tablicy odtworzyć strukturę drzewa? Czy umiesz odpowiedzieć na pytanie, jaką zawartość ma lewy lub prawy syn na podanym poziomie?

Drzewo licznikowe

Definicja

Drzewo licznikowe jest drzewem binarnym, które pozwala przypisywać elementom wartości ze zbioru liczb całkowitych oraz wyznaczać wartość sumy w zadanym przedziale elementów. Węzły drzewa przechowują sumę wartości swoich podzbiorów (wartość każdego wierzchołka jest sumą jego synów).

Do zapamiętania drzewa wykorzystujemy tablicę o rozmiarze równym dwukrotnej wartości potęgi liczby 2, która jest większa lub równa maksymalnemu elementowi zbioru (np. dla 6 jest to 8). Wyznaczoną potęgę nazywa się CZAPĄ lub BAZĄ. Wartość ta jest równa numerowi pierwszego wierzchołka ostatniego poziomu (liścia).

Dodawanie elementów

Węzły przechowują sumę elementów swoich podzbiorów, dlatego jeżeli dodamy 1 do elementu o indeksie 8 na naszym drzewie, to wartość węzłów: 4, 2 i 1 (wszystkich nadrzędnych) powinna automatycznie wzrosnąć. Zwróć uwagę, że numer węzła nadrzędnego obliczamy, wykonując dzielenie całkowitoliczbowe.

Jeżeli numer węzła (indeks tablicy) oznaczymy przez *k*, a liczbę dodawanych elementów przez *ile*, funkcje dodające, zmniejszające lub podające liczbę elementów z zadanego przedziału będą wyglądać następująco:

Funkcja dodaj

```
void dodaj (int k, int ile))
{
int p=BAZA + k;
while (p>0)
{
 t[p]++;
 p/=2;
}
```

Funkcja zmniejsz

```
void zmniejsz (int k, int ile)
{
```

```
    int p=BAZA + k;
    while (p>0)
     {
      t[p]=t[p]-ile;
      p=p/2;
     }
  }
```

Obliczanie liczby elementów w zadanym przedziale <a, b>

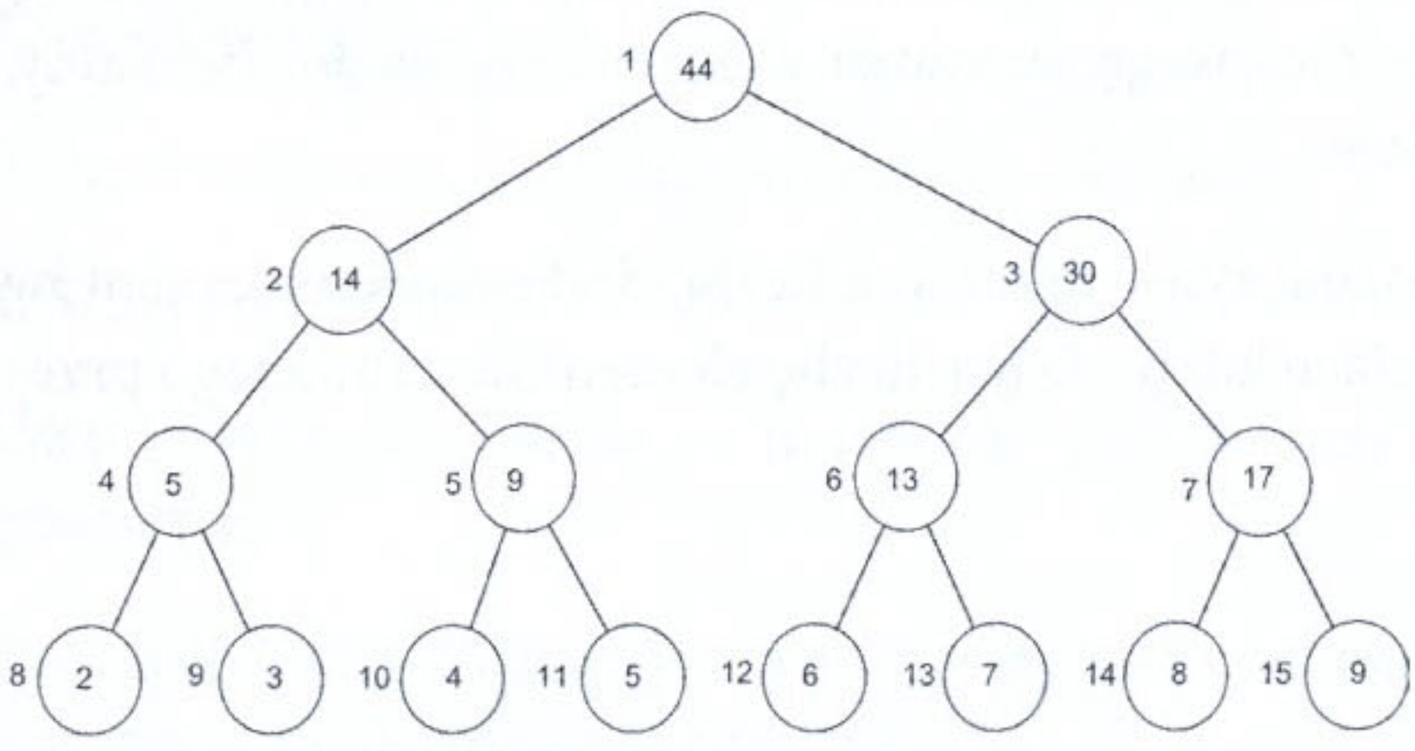
$a = a$+BAZA

$b = b$+BAZA

Jeżeli a jest równe b, to wiadomo, że jako liczbę elementów możemy zwrócić $t[a]$ lub $t[b]$. W przeciwnym wypadku zliczamy sumę $t[a]+t[b]$ i dopóki $a/2$ jest różne od $b/2$ (nie mają wspólnego ojca), dodajemy elementy tablicy: dla a — $t[a+1]$, jeżeli reszta z dzielenia $a\%2 = = 0$ ($t[a]$ jest lewym synem i trzeba dodać prawego brata), a dla b — $t[b-1]$, jeżeli reszta z dzielenia $b\%2 = = 1$ ($t[b]$ jest prawym synem i trzeba dodać lewego brata).

Przykład

Oto omówienie funkcji dla drzewa z rysunku 1.8.



Rysunek 1.8.
Drzewo licznikowe

Tak wygląda tablica.

	44	14	30	5	9	13	17	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

a+BAZA = 9

b+BAZA =13

Krok 1.: wynik = $t[9]+t[13] = 3+7 = 10$.

Krok 2.: sprawdzamy, czy a i b mają wspólnego ojca ($a!/2 = b/2$) i czy a było lewym synem (nie, bo reszta z dzielenia 9%2 nie jest równa 0), a b — prawym (tak, bo reszta z dzielenia 13/2 jest równa 1); wynik+ $t[b-1] = 10+6 = 16$, $a = 9/2 = 4$, $b = 13/2 = 6$ (idziemy w górę).

Krok 3.: $a/2! = b/2$ — tak, a jest lewym synem ($4\%2 = 0$), czyli do wyniku dodajemy jego prawego brata $t[a+1]$, wynik = wynik+$t[5]$ = 16+9 = 25; b jest prawym synem — nie dodajemy, $a = a/2 = 4/2 = 2$, b = $b/2 = 6/2 = 3$.

Krok 4.: $a/2! = b/2$ — nie, kończymy — wynik = 25.

```
int ile_p (int a, int b)
  {
   a=a+baza;
   b=b+baza;
   int wynik=0;
   wynik+=znaczki[a];
  if (a!=b)
   wynik+=t[b];
  while (a/2!=b/2)
  {
   if (a%2==0) wynik+ti[a+1];
   if(b%2==1) wynik+=t[b-1];
   a=a/2;
   b=b/2;
  }
   return wynik;
   }
```

Zadanie 1.20 Spróbuj napisać:

- funkcję zliczającą, ile jest elementów mniejszych, a ile większych od podanej wartości,
- funkcję znajdującą liczbę minimalną i maksymalną w drzewie licznikowym.

Zadanie 1.21 Pan Kazimierz kolekcjonuje znaczki pocztowe, których cena wyraża się liczbą całkowitą $1 \le c \le 1000$. Bardzo często wymienia się z innymi kolekcjonerami. Duży zbiór znaczków wymaga częstego liczenia i zapamiętywania aktualnego stanu po każdej operacji. Pan Kazimierz potrzebuje programu, który zapamiętuje aktualny stan wszystkich znaczków, daje możliwość szybkiej aktualizacji po każdej operacji, a w przypadku gdy wymiana nie może być zrealizowana (brak wymaganej liczby znaczków), wyświetla komunikat „TRANSAKCJA NIE MOŻE BYĆ ZREALIZOWANA". Dasz radę napisać taki program?

Drzewo potęgowe

Drzewo potęgowe najłatwiej przedstawić w postaci tablicy, w której pod i-tym indeksem jest zapisywana informacja o k ostatnich elementach (łącznie z elementem i), gdzie k jest najwyższą potęgą dwójki będącej dzielnikiem i.

Wszystkie komórki o nieparzystych indeksach pamiętają tylko siebie.

Jeżeli do komórki $i = 1$ wpiszemy jakąś wartość, np. 2, to w tablicy T:

$T[1]$: $T[1] = 1$

$T[2]$: $k = 2$ $T[2] = T[1]+T[2] = 1$

$T[3]$: $T[3]$

$T[4]$: $k = 4 T[4] = T[1]+T[2]+T[3]+T[4] = 1$

$T[5]$: $T[5]$

$T[6]$: $k = 2 T[6] = T[5]+T[6]$

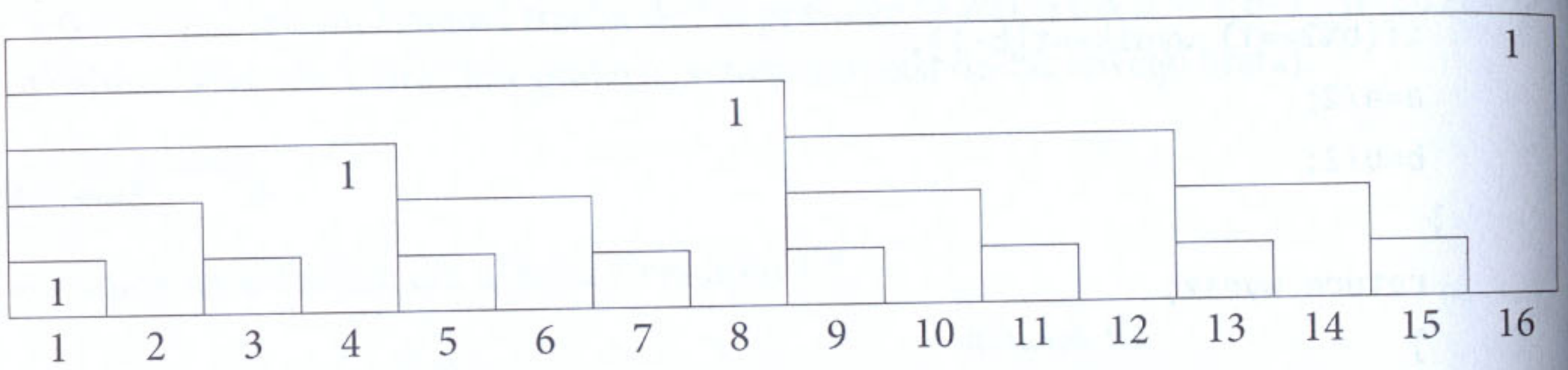
$T[7]$: $T[7]$

$T[8]$: $k = 8 T[8] = T[1]+T[2]+T[3]+T[4]+T[5]+T[6]+T[7]+T[8] = 1$

$T[9]$: $T[9]$

...

$T[16]$: $k = 16 T[16]=T[1]+...+T[16]$



Jeżeli uaktualnimy dowolny element tablicy o indeksie i, to zmiana musi nastąpić w elemencie o indeksie $i+p$, gdzie p jest największą potęgą dwójki dzielącą indeks pola aktualnego.

p wyznaczamy jako iloczyn binarny i & $-i$.

Przykład

Modyfikujemy $T[6]$. Zgodnie ze schematem zmianie powinny ulec również $T[8]$ i $T[16]$ (i <= rozmiar naszej tablicy).

Sprawdzamy.

6	110
–6	010 (kod uzupełnień do dwóch)
i & $-i$	010
$p = 2i+p = 8$	
8	100
–8	100
i & $-i$	100
$p = 8i+p = 16$	

1.9.2. Stos

Nazwa struktury w pełni oddaje to, co wyobrażamy sobie pod tym pojęciem (np. stos talerzy, książek, płyt itp.).

Definicja

Stos jest to struktura, w której mamy dostęp tylko do ostatniego elementu zwanego wierzchołkiem. Jej rozmiar zmienia się dynamicznie: dodając — zwiększamy, zdejmując — zmniejszamy. Element, który umieszczamy na stosie jako pierwszy, zdejmujemy jako ostatni — LIFO (*Last In First Out*). Na stos możemy kłaść elementy dowolnego typu, określonego przy definiowaniu zmiennej.

Do korzystania ze stosu potrzebujemy biblioteki `stack`.

Dodanie elementu: *`nazwa_stosu`*`.push(`*`element`*`).`

Zdjęcie elementu: *`nazwa_stosu`*`.pop().`

Funkcja zwracająca wartość `true`, gdy stos jest pusty: *`nazwa_stosu`*`.empty().`

Funkcja zwracająca liczbę elementów (rozmiar) stosu: *`nazwa_stosu`*`.size().`

Funkcja zwracająca wartość wierzchołka: *`nazwa_stosu`*` .top().`

Zastosowanie stosu prześledzimy w znanym zapewnie wszystkim algorytmie zamiany liczby dziesiętnej na binarną.

```
#include <iostream>
#include <stack>
using namespace std;
main ()
{int liczba,znak;
//Deklarujemy zmienną binarną jako stos liczb całkowitych
 stack <int> binarna;
 cin>>liczba;
while (liczba>0)
{znak=liczba%2;
 liczba=liczba/2;
//Wkładamy na stos otrzymaną resztę z dzielenia (0 lub 1)
 binarna.push(znak);
    }
//Możemy wyświetlić rozmiar naszego stosu
cout<<binarna.size()<<endl;
//Dopóki stos nie jest pusty...
while (!binarna.empty())
```

```
{
//Wyświetlamy wartość wierzchołka
cout<<binarna.top()<<" ";

//Zdejmujemy element ze stosu
binarna.pop();
}
 system("pause");
 return 0;
}
```

Zastosowanie stosu w ONP

Czy wiesz, co to jest odwrotna notacja polska (ONP)? To sposób zapisu wyrażenia arytmetycznego pozwalający na opuszczenie nawiasów, przedstawiony przez Jana Łukasiewicza. Jest to tzw. notacja postfiksowa — operatory są umieszczane po argumentach.

Wyrażenie	ONP
4+12	4 12+
(2+3)*(4–2)	2 3+4 2–*
(2+3)/2*(3+5)	?

Jak widać, zasada jest prosta — najpierw wprowadzamy liczby, a następnie znak działania, które należy wykonać.

Przeanalizujmy zastosowanie stosu dla wyrażenia zapisanego w drugim wierszu tabeli:

- odkładamy na stos 2;
- odkładamy na stos 3;
- mamy znak +;
- zdejmujemy ze stosu 2 elementy;
- dodajemy je: 2+3;
- odkładamy na stos wynik 5;
- dodajemy do stosu 4;
- dodajemy do stosu 2 (mamy teraz na stosie 2, 4, 5);
- mamy znak –;
- zdejmujemy ze stosu 2 elementy (2, 4);
- obliczamy ich różnicę: 4–2 (ostatni od poprzedniego);
- odkładamy wynik na stos — 2 (na stosie: 2, 5);
- mamy znak *;
- pobieramy 2 elementy ze stosu;
- mnożymy je: 2*5;
- otrzymujemy wynik 10.

FTP **Zadanie 1.22** Napisz program, który oblicza wartość wyrażenia zapisanego w ONP.

Zadanie 1.23 Napisz program, który zamienia wyrażenie podane z użyciem nawiasów na ONP.

1.9.3. Kolejka

Definicja

Kolejka jest liniową strukturą danych, w której elementy są dopisywane na końcu, a pobierane z początku — FIFO (*First In First Out*).

Do korzystania z kolejki potrzebna jest biblioteka: `queue`.

Deklaracja kolejki: `queue <typ_elementów> Nazwa_kolejki`.

Dodanie elementu: `Nazwa_kolejki.push(element)`.

Przód kolejki: `Nazwa_kolejki.front()`.

Usunięcie elementu: `Nazwa_kolejki.pop()`.

Funkcja zwracająca prawdę, jeżeli kolejka jest pusta: `Nazwa_kolejki.empty()`.

Wykorzystamy kolejkę do rozwiązania problemu Józefa Flawiusza:

```
#include <iostream>
#include <queue>
using namespace std;
int main ()
{
 int n;
 int k;
 queue <int> Q;
 cin>>n>>k;
//Dodajemy n elementów do kolejki
 for (int i = 1; i <= n; i++)
  Q.push(i);
 while ( !Q.empty() )
  {
//Przestawiamy k elementów z początku na koniec kolejki
   for (int j = 0; j <k - 1; j++)
    {Q.push(Q.front());
     Q.pop();
    }
```

```
//Wypisujemy numer usuwanego elementu na ekranie
 cout<<Q.front()<<" ";
//i usuwamy go z kolejki
 Q.pop();
 }
 cout<<endl;
 system ("pause");
 return 0;
}
```

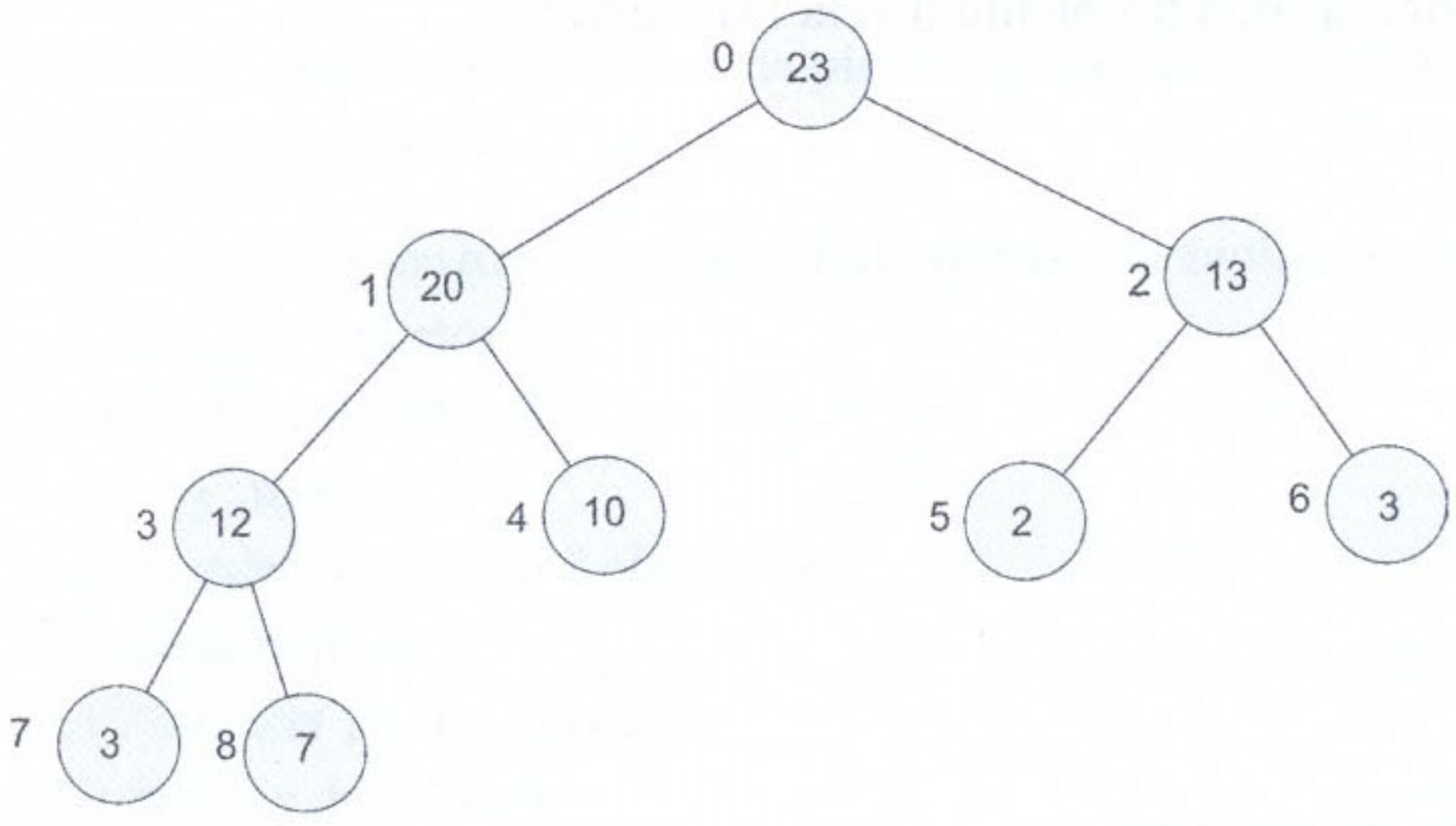
1.9.4. Kopiec

Definicja

Kopiec jest drzewem binarnym, które spełnia następujące właściwości:

- jest uporządkowane, tzn. wartość przechowywana przez rodzica jest większa od wartości przechowywanej przez synów;
- wszystkie poziomy są zapełnione; jeżeli nie, to na ostatnim poziomie „dosuwamy" do lewej strony.

Przykład kopca pokazano na rysunku 1.9.



Rysunek 1.9. Kopiec

Oto jego implementacja tablicowa.

23	20	13	12	10	2	3	3	7

Funkcja zwracająca indeks rodzica i-tego elementu

```
int rodzic(int i)
 {
  return (i-1)/2;
 }
```

Funkcja zwracająca indeks lewego syna

```
int lewy_syn(int i)
{
  return 2*i+1;
}
```

Wskazówka

Pamiętajmy, że pierwszy element tablicy ma indeks 0.

Funkcja zwracająca indeks prawego syna

```
int prawy_syn (int i)
 {
 return 2*i+2;
//Albo po prostu lewy(i)+1
 }
```

Dodawanie elementów do kopca odbywa się „od dołu". Dodajemy element, zwiększając rozmiar tablicy o 1. Syn musi być mniejszy od rodzica, a więc dokonujemy porównania z elementem nadrzędnym w drzewie i jeżeli ten warunek nie jest spełniony, dokonujemy zamiany. Czynność powtarzamy, aż warunek będzie prawdziwy lub dojdziemy na szczyt.

Funkcja dodająca element do kopca

```
void wstaw_element(int x)
 {int i=rozmiar;
  rozmiar++;
  while ((i>0) && (x>kopiec[rodzic(i)]))
   {kopiec[i]=kopiec[rodzic(i)];
   i=rodzic(i);
  }
  kopiec[i]=x;
}
```

Wskazówka

Jeżeli wykorzystujemy funkcję do wpisywania wszystkich elementów kopca, przyjmujemy rozmiar równy 0.

FTP **Zadanie 1.24** Napisz program, który zapisuje w tablicy strukturę kopca z rysunku 1.9.

Zastanów się, co powinniśmy zrobić, jeżeli usuwamy element z kopca, ale chcemy nadal utrzymać strukturę.

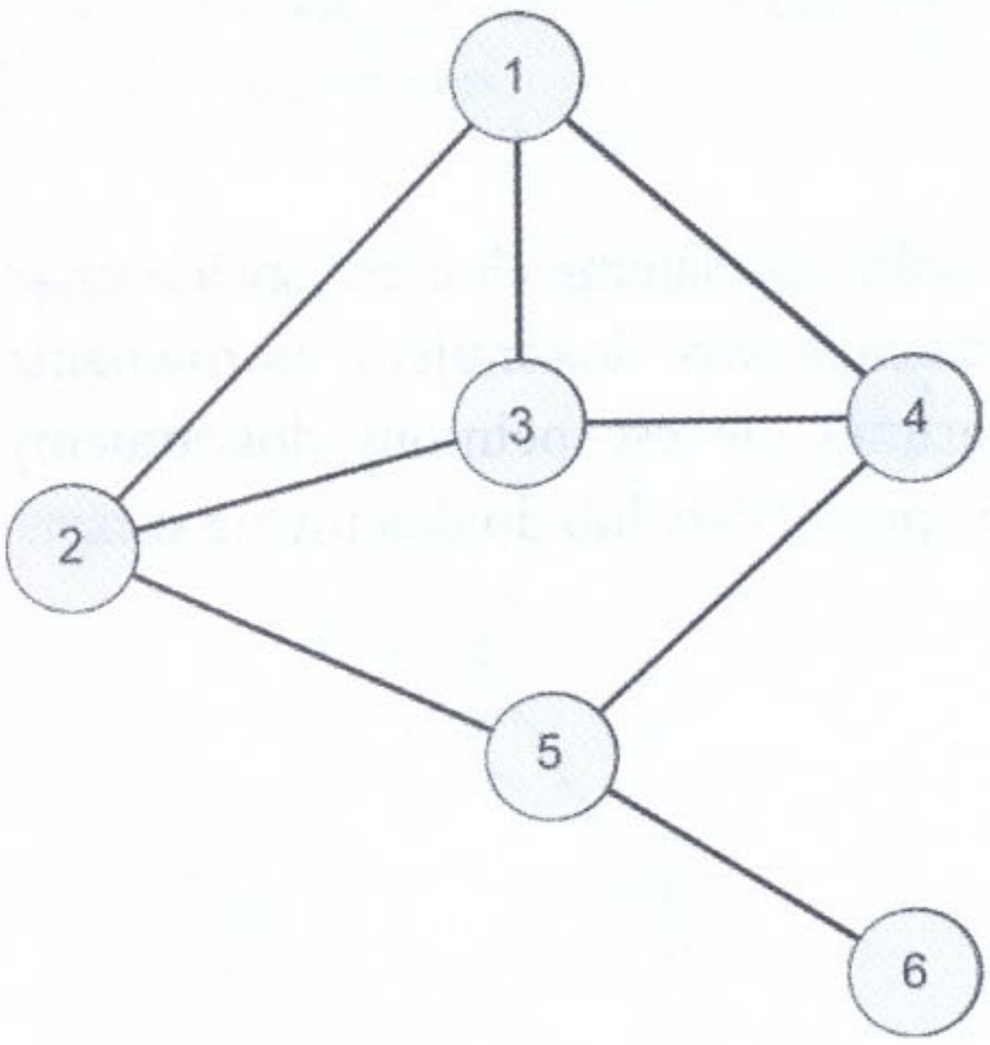
1.10. Algorytmy grafowe

Zrozumienie struktury i możliwości stosowania grafów w dużym stopniu ułatwia modelowanie i rozwiązywanie wielu problemów.

Definicja

Graf jest rozszerzeniem struktury drzewiastej i stanowi obiekt składający się ze zbioru wierzchołków i z krawędzi łączących niektóre wierzchołki. Jest wykorzystywany do modelowania różnego rodzaju sieci połączeń (m.in. komunikacyjnych).

Graficznie jest przedstawiany w postaci zbioru wierzchołków i zbioru łączących je krawędzi (rysunek 1.10).



Rysunek 1.10. Graf G(6, 8)

Graf najczęściej jest przedstawiany za pomocą symboli $G = (V, E)$:

V — zbiór wierzchołków (*vertex*),

E — zbiór krawędzi (*edge*).

Grafy mogą być skierowane (krawędzie mają jeden określony kierunek, np. jednokierunkowe ulice) lub nieskierowane (krawędzie „dwukierunkowe").

Jeżeli krawędziom są przypisane wartości (np. długość drogi, czas lub koszty przejazdu), mamy do czynienia z grafami ważonymi.

Stopień wierzchołka v — liczba wychodzących z niego krawędzi.

Ścieżka — ciąg od v_0 do v_n, w którym każde dwa kolejne wierzchołki są połączone krawędzią.

Z wierzchołka 1 do 6 mamy kilka ścieżek (np. 1-2-5-6).

Zamiast numerów węzłów możemy posługiwać się numerami krawędzi (jeżeli oczywiście istnieją ☺).

Przy planowaniu tras przejazdów korzystamy na ogół z numerów dróg zamiast nazw miejscowości. Jest to praktyczniejsze, ponieważ do tego samego miasta możemy najczęściej dojechać na kilka sposobów.

Długość ścieżki — liczba krawędzi na ścieżce.

Cykl — ścieżka zaczynająca się i kończąca w tym samym wierzchołku.

Sąsiad wierzchołka v — każdy wierzchołek połączony bezpośrednio krawędzią z v.

Graf spójny — wszystkie wierzchołki są połączone.

Graf skierowany — krawędzie mają wyróżniony początek i koniec.

Graf ważony — każda krawędź ma swoją wagę (np. długość drogi).

1.10.1. Reprezentacja grafu w pamięci komputera

Tablica sąsiedztwa

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

Zbiór krawędzi jest reprezentowany tablicą kwadratową o rozmiarze `w` na `w` (`w` — liczba wierzchołków).

`tablica[i][j]` — liczba krawędzi między wierzchołkami `i` – `j`.

```
#include <iostream>
using namespace std;
int main ()
{


 //a, b – numery wierzchołków;
 //w – liczba wierzchołków, k – liczba krawędzi
```

```
    int a,b,w,k;
    cin>>w>>k;
    int t[w+1][w+1];
    for (int i=0; i<=w; i++)
     {  for (int j=0; j<=w; j++)
        t[i][j]=0;
      }
//Tworzymy tablicę sąsiedztwa
    for (int i=1; i<=k; i++)
   { cin>>a>>b;
//Podajemy numer wierzchołków połączonych krawędzią
     t[a][b]++;
     t[b][a]++;
    }
//Wypisujemy tablicę
      for (int i=1; i<=w; i++)
    { for (int j=1; j<=w; j++)
       cout<<t[i][j]<<" ";
     cout<<endl;
      }
    system ("pause");
    return 0;
  }
```

Lista sąsiedztwa

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

Do przechowywania listy sąsiedztwa wykorzystamy klasę `vector` — dynamiczne tablice dowolnego typu, które mają własny zestaw funkcji.

Biblioteka: `<vector>`.

Deklaracja pustego wektora `V`: `vector < int > V`.

Dodanie nowego elementu `x` na koniec: `V.push_back(x)`.

Usunięcie ostatniego elementu: `V.pop_back()`.

Odwołanie do pierwszego elementu: `V[0]`.

Funkcja zwracająca liczbę elementów: `V.size()`.

Wykorzystamy wektor do opisu grafu:

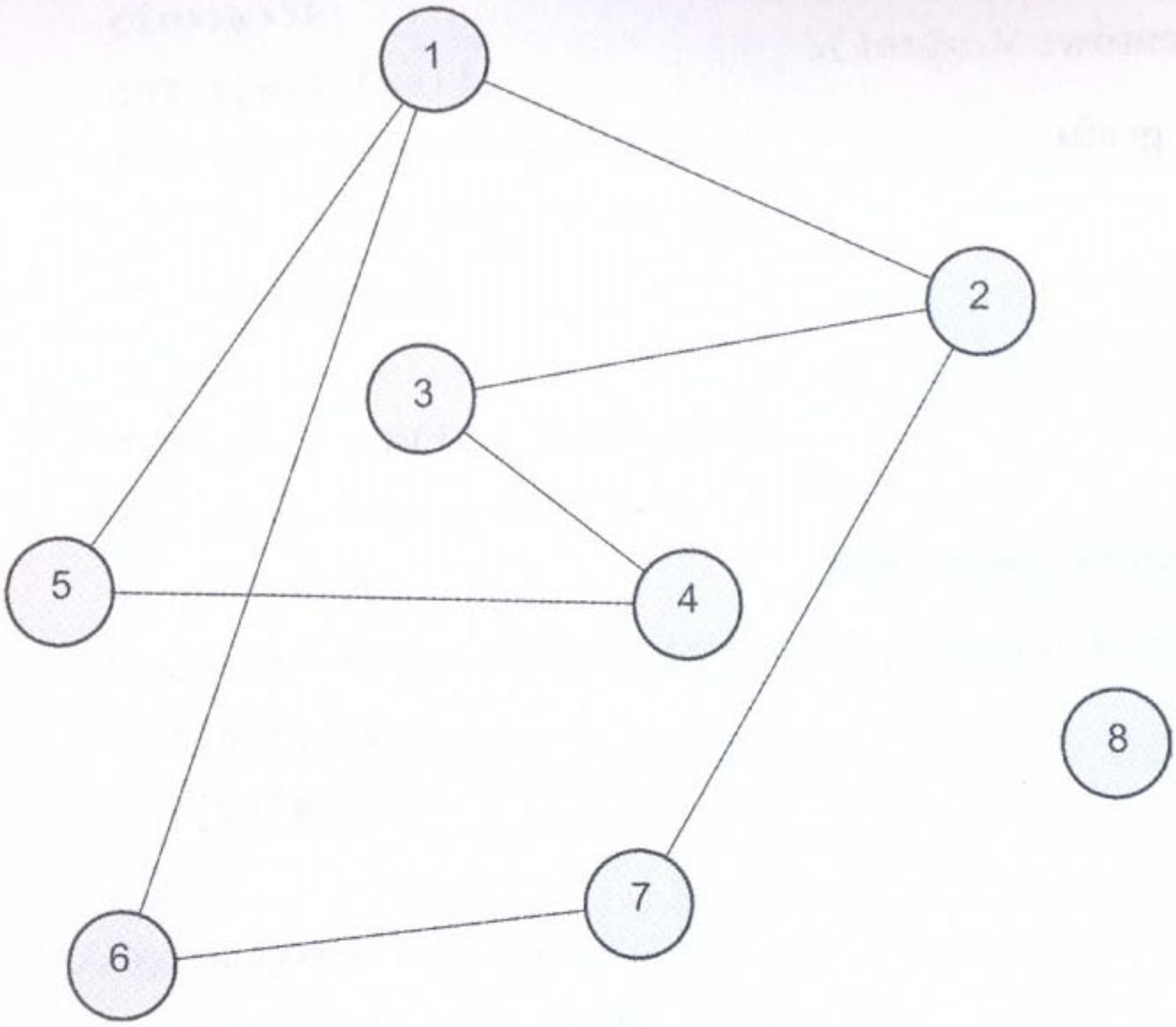
```
#include <iostream>
#include <vector>
using namespace std;
int main ()
{vector <int> G;
//Oznaczenia jak w poprzednim programie
int w,k,a,b;
cin>>w>>k;
for (int i=0; i<k;i++)
 {cin>>a>>b;

//Dodajemy element
 G[a].push_back(b);

//Dodajemy element
 G[b].push_back(a);
 }
//Wypisujemy elementy wektora (listę sąsiedztwa)
for( int j=1; j<=w; j++)
 {
  cout << j << ":";
  for (int i=0; i<G[j].size(); i++)
   cout <<' '<<G[j][i];
  cout << endl;
 }
system("pause");
return 0;
}
```

Zadanie 1.25 Przedstaw graf z rysunku 1.11 w postaci tablicy sąsiedztwa i listy sąsiedztwa.

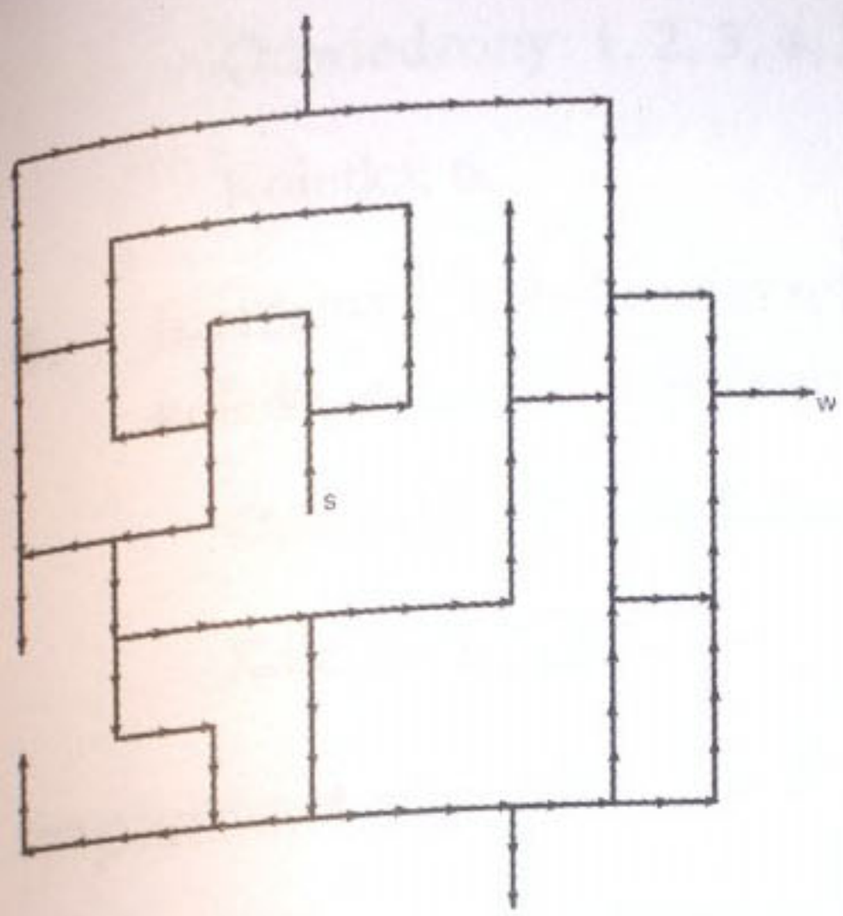
Rysunek 1.11. Graf



Zadanie 1.26 Klasa bardzo chce jechać na biwak, podczas którego uczniowie mają nocować w 2-osobowych domkach połączonych ścieżkami. Każdy domek jest połączony maksymalnie z dwoma innymi. Wychowawca obawia się, że może mieć w nocy problem z dyscypliną, szczególnie chłopców. Ponieważ dziewczynki nie sprawiają żadnych kłopotów, zgodził się na biwak pod warunkiem, że uda się tak ulokować młodzież, by chłopcy sąsiadowali tylko z dziewczynkami. W klasie jest tyle samo chłopców i dziewczynek. Ich suma jest liczbą parzystą. Do wyboru jest kilka wariantów zawierających numery domków połączonych ścieżkami. Napisz program, który dla dostępnego zestawu danych da odpowiedź na pytanie, czy uda się ulokować młodzież zgodnie z życzeniem wychowawcy. Rozpatrz wyłącznie te przypadki, w których domków wystarczy dla wszystkich dzieci.

1.10.2. Przeszukiwanie grafu

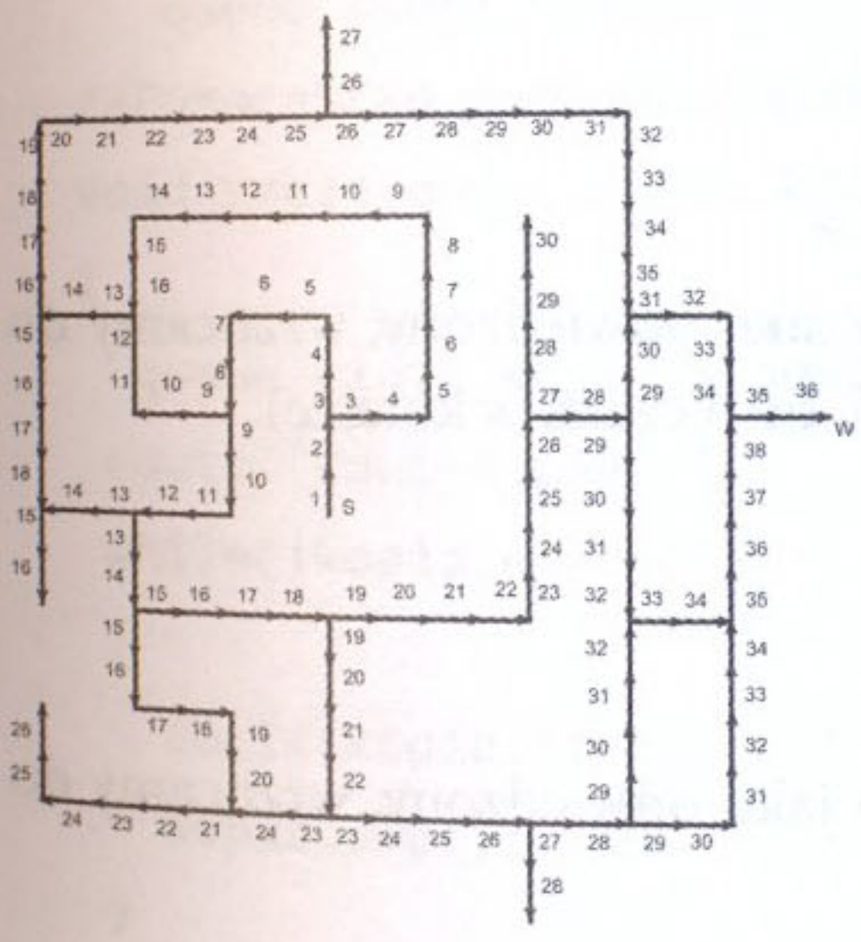
Przeszukiwanie grafu polega na odwiedzeniu wszystkich wierzchołków w celu zebrania potrzebnych informacji. Najlepiej przedstawić to na przykładzie. Wyobraźmy sobie, że rysunek 1.12 przedstawia sieć dróg w labiryncie, a naszym zadaniem jest dotarcie z punktu S (start) do punktu W (wyjście). Wiadomo, że zawsze staramy się wybrać optymalną drogę, co w konkretnym przypadku oznacza najkrótszą.



Rysunek 1.12. Labirynt (sieć dróg)

1.10.3. Przeszukiwanie wszerz BFS

Zaznaczamy na rysunku odległości od punktu startowego do każdego kolejnego węzła (wierzchołka grafu). Oznaczamy każdy kolejny wierzchołek liczbą reprezentującą odległość. Jeżeli do węzła prowadzą dwie różne drogi, to przypisujemy mu zawsze mniejszą wartość (rysunek 1.13).



Rysunek 1.13.
Wychodzenie z labiryntu BFS

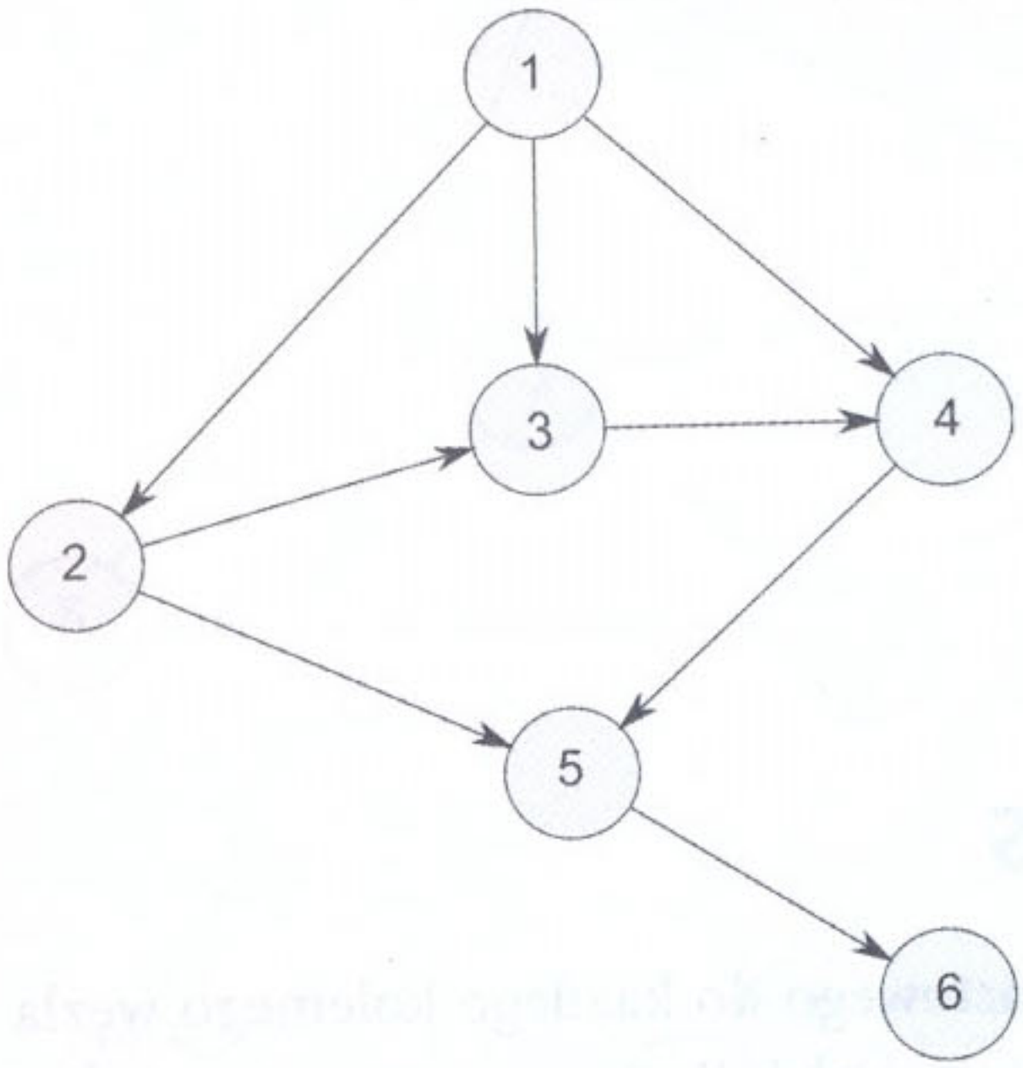
Taki algorytm przeszukiwania grafu nazywa się przeszukiwaniem wszerz — BFS (*Breadth — First — Search*). Nazwa jest jak najbardziej adekwatna do tego, co robimy — na każdym kroku poszukujemy kolejnych wierzchołków, których jeszcze nie odwiedziliśmy (rozglądamy się wszerz).

Do implementacji tego algorytmu najlepiej wykorzystać kolejkę:

- zaznaczamy pierwszy wierzchołek jako odwiedzony, wrzucamy do kolejki wszystkie jego następniki w kolejności rosnącej;
- do momentu, aż kolejka będzie pusta, bierzemy kolejno każdy wierzchołek z kolejki, zaznaczamy jako odwiedzony, wrzucamy do kolejki wszystkie jego nieodwiedzone następniki.

Przykład

Przeszukajmy graf z rysunku 1.14.



Rysunek 1.14. Graf

1. Zaczynamy od wierzchołka 1. Zaznaczamy go jako odwiedzony, wrzucamy do kolejki jego następniki.

   Odwiedzony: 1.

   Kolejka: 2, 3, 4.

2. Bierzemy z kolejki wierzchołek 2. Zaznaczamy go jako odwiedzony, wrzucamy do kolejki jego nieodwiedzone następniki (jest tylko 5, bo 3 czeka w kolejce).

   Odwiedzony: 1, 2.

   Kolejka: 3, 4, 5.

3. Bierzemy z kolejki wierzchołek 3. Zaznaczamy go jako odwiedzony, wrzucamy do kolejki jego nieodwiedzone następniki.

   Odwiedzony: 1, 2, 3.

   Kolejka: 4, 5.

4. Bierzemy z kolejki wierzchołek 4. Zaznaczamy go jako odwiedzony, wrzucamy do kolejki jego nieodwiedzone następniki.

   Odwiedzony: 1, 2, 3, 4.

   Kolejka: 5.

5. Bierzemy z kolejki wierzchołek 5. Zaznaczamy go jako odwiedzony, wrzucamy do kolejki jego nieodwiedzone następniki.

Odwiedzony: 1, 2, 3, 4, 5.

Kolejka: 6.

6. Bierzemy z kolejki wierzchołek 6. Zaznaczamy go jako odwiedzony, wrzucamy do kolejki jego nieodwiedzone następniki.

Odwiedzony: 1, 2, 3, 4, 5, 6.

Kolejka: PUSTA.

Implementacja BFS

```
//a, b – numery wierzchołków;
//w – liczba wierzchołków, k – liczba krawędzi
#include <iostream>
#include <queue>
using namespace std;
   int a, b, w, k;
//Pomocniczy stos pokazujący odwiedzone
   queue <int> odwiedzone;
//Pomocnicza funkcja pokazująca wyniki kolejnych etapów
void pokaz(queue <int> kolejka)
{
  queue <int> kopia = odwiedzone;
 cout<<"Odwiedzone: ";
 while(!kopia.empty())
 {
   cout<<kopia.front()<<" ";
   kopia.pop();
 }
cout<<endl;
if(kolejka.empty())
{
   cout<<"Kolejka PUSTA!"<<endl;
   return;
}
cout<<"Kolejka: "<<kolejka.front()<<" ";
kolejka.pop();
while(!kolejka.empty())
{
   cout<<kolejka.front()<<" ";
```

```
//Usuwamy element z kolejki
   kolejka.pop();
}
 cout<<endl;
}
int main ()
{
  cin>>w>>k;
  int t[w+1][w+1];
  for (int i=0; i<=w; i++)
  {   for (int j=0; j<=w; j++)
       t[i][j]=0;
   }
 //Tworzymy tablicę sąsiedztwa
  for (int i=1; i<=k; i++)
//Podajemy numer wierzchołków połączonych krawędzią
  { cin>>a>>b;
    t[a][b]++;
    t[b][a]++;
   }
  //BFS - przeszukujemy od wierzchołka s
  queue <int> kolejka;
  int s;
  cin>>s;
  bool V[w];
  //Wierzchołki nieodwiedzone przyjmują wartość false
for(int j=0;j<=w;++j)V[j] = false;
  //Wrzucamy pierwszy wierzchołek do kolejki
kolejka.push(s);
while(!kolejka.empty())
{
 pokaz(kolejka);
 s = kolejka.front();
//Usuwamy odwiedzony element z kolejki
 kolejka.pop();
 cout<<"Kolejny wierzcholek "<<s<<endl;;
 odwiedzone.push(s);
//Zaznaczamy jako odwiedzony
 V[s] = true;
 for(int j = 0;j<=w;++j)
```

```
{
 if(t[j][s] != 0 && V[j] == false)
 {
//Wrzucamy na stos nieodwiedzonych następników
 kolejka.push(j);
 //Zaznaczamy kolejny jako odwiedzony
 V[j] = true;
  }
 }
 pokaz(kolejka);
     system ("pause");
  return 0;
 }
```

1.10.4. Przeszukiwanie w głąb DFS

Tak jak w poprzednim algorytmie rozpatrywaliśmy wszystkie możliwe drogi (wszerz), tak w tym przypadku wybieramy krawędzie w ustalonej kolejności — np. w górę, w lewo, w prawo, w dół. Każda kolejna opcja jest wybierana wtedy, gdy poprzednia nie jest możliwa.

Wykonaj symulację tego algorytmu dla labiryntu z rysunku 1.14. Przyjmij zasadę, że dochodząc do wierzchołka, wybierasz zawsze drogę w lewo. Wracając do wierzchołka, z którego możesz kontynuować ruch, poruszasz się zawsze przodem do kierunku ruchu.

Taki algorytm nazywa się przeszukiwaniem w głąb DFS (*Depth — First — Search*). Zgodnie z nazwą przy przeszukiwaniu „zagłębiamy" się w graf.

Do implementacji tego algorytmu najlepiej wykorzystać stos. Wybrany wierzchołek umieszczamy na stosie, zaznaczamy jako odwiedzony i wrzucamy na stos wszystkie jego następniki w porządku malejącym.

Zadanie 1.27 Spróbuj napisać implementację tego algorytmu. Zastosuj rekurencję.

Algorytm DFS wykorzystujemy w dalszej części podręcznika do znajdowania cyklu Eulera.

O grafach wiemy już dużo. Potrafimy je zapisywać w pamięci komputera jako tablicę sąsiedztwa i listę sąsiedztwa. Potrafimy przeszukać go wszerz i w głąb. Pora teraz na rozwiązanie nurtujących nas często problemów:

- Czy można zaplanować trasę zwiedzania miejscowości, w której chcemy zobaczyć kilka atrakcji turystycznych, w ten sposób, żeby nie przechodzić dwa razy tą samą drogą?
- Czy można tak zaplanować trasę, żeby nie oglądać dwa razy tego samego?
- I jak to zrobić optymalnie? Czyli tak, żeby było najkrócej lub najtaniej (co w przypadku wycieczek szkolnych często jest istotnym warunkiem ☺).

1.10.5. Cykl Eulera — przez każdą krawędź przechodzimy tylko raz

Definicja

Cykl Eulera — cykl, który przez każdą krawędź grafu przechodzi dokładnie jeden raz.

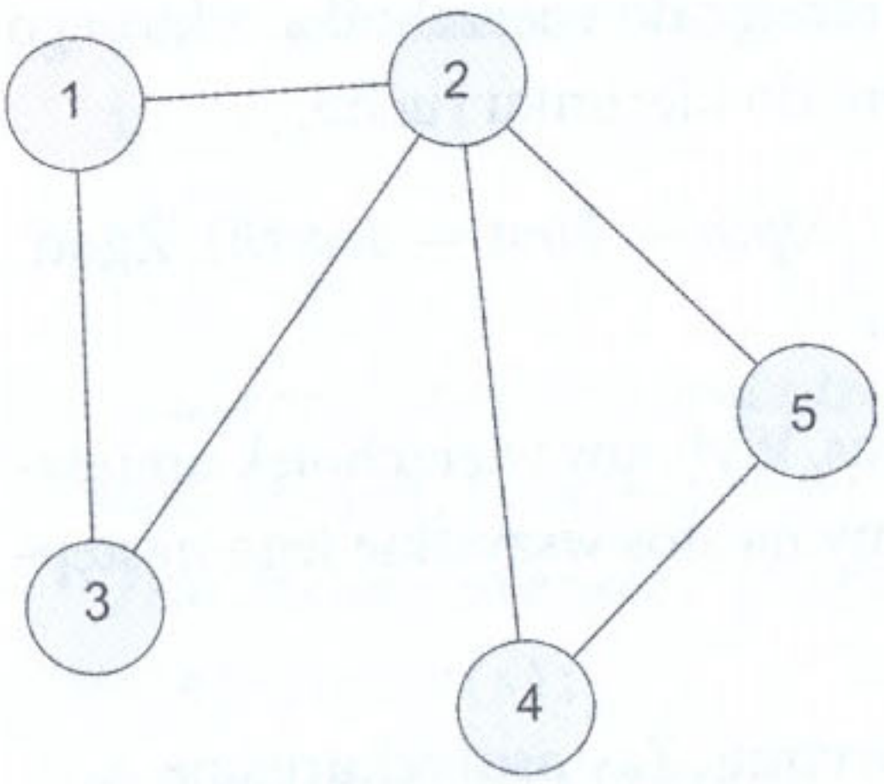
Warunki konieczne do istnienia:

- graf nieskierowany musi mieć parzysty stopień każdego wierzchołka;
- graf skierowany musi mieć taką samą liczbę krawędzi wchodzących i wychodzących.

Znajdowanie cyklu Eulera

Wykorzystujemy znany nam stos. Bierzemy pierwszy wierzchołek (ten, od którego „startujemy") i wrzucamy go na stos. Przechodzimy do kolejnego (wybieramy zawsze ten z mniejszym indeksem), wrzucamy go na stos, a drogę usuwamy. Schemat powtarzamy, aż osiągniemy wierzchołek, z którego nie możemy już nigdzie pójść. Pobieramy wtedy element ze stosu — jest on kolejnym elementem cyklu — i postępujemy identycznie jak przy poprzednim wierzchołku. Robimy tak, aż stos będzie pusty. Otrzymany ciąg wierzchołków jest cyklem Eulera.

Cykl Eulera dla grafu z rysunku 1.15



Rysunek 1.15. Graf

1. Stos 1, 2. Cykl: —. Usunięta krawędź: 1 – 2.
2. Stos 1, 2, 3. Cykl: —. Usunięta krawędź: 2 – 3.
3. Stos 1, 2, 3, 1. Cykl: —. Usunięta krawędź: 3 – 1.
4. Nie możemy nigdzie iść — pobieramy element o numerze 1 ze stosu.

   Stos 1, 2, 3. Cykl: 1.

5. Nadal nie mamy przejścia — pobieramy element 3 ze stosu. Możemy już dodać 4 i usunąć krawędź 2 – 4.

   Stos: 1, 2, 4. Cykl: 1, 3. Usunięta krawędź: 2 – 4.

6. Stos 1, 2, 4, 5. Cykl: 1, 3. Usunięta krawędź: 4 – 5.

7. Stos 1, 2, 4, 5, 2. Cykl; 1, 3. Usunięta krawędź: 5 – 2.

8. Przeszliśmy wszystkie krawędzie. Zdejmujemy teraz elementy ze stosu i dodajemy do cyklu. Cykl: 1, 3, 2, 5, 4, 2, 1.

Implementacja

```
#include <iostream>
#include <vector>
using namespace std;
vector <int> cykl;
vector <int> G[100];
int odwiedzone [100][100];
int dfs(int v)
{
  while(!G[v].empty())
  {
int t = G[v].back();
G[v].pop_back();
if(odwiedzone[v][t] == 0)
{
  odwiedzone[v][t]=1;
  odwiedzone[t][v]=1;
  dfs(t);
  cykl.push_back(t);
}
  }
}
int main ()
{

int w,k,a,b;
cin>>w>>k;
//Wprowadzamy krawędzie
for(int i=1; i<=k; i++)
  {
```

```
    cout << i << ": ";
    cin >> a >> b;
//Numer początkowego i końcowego wierzchołka
    G[a].push_back(b);
    G[b].push_back(a);
  }
  cout<<"Start: ";
  int start;
  cin>>start;
  dfs(start);
  cout<<start<<" ";
  for(int i = 0 ; i <w ;i++)
{
  cout << cykl.back() << " ";
  cykl.pop_back();
}
system("pause");
 return 0;
}
```

1.10.6. Cykl Hamiltona

Definicja

Drogę, która przechodzi przez wszystkie wierzchołki grafu dokładnie jeden raz, nazywamy **cyklem Hamiltona**.

Każdy graf pełny (czyli taki, w którym istnieją połączenia między wszystkimi wierzchołkami) o liczbie wierzchołków większej od 3 jest grafem hamiltonowskim. Wszystkie wierzchołki możemy przejść w dowolnej kolejności, ale za każdym razem mamy do czynienia z różną liczbą krawędzi, a w przypadku grafu ważonego — dodatkowo długością (kosztem) drogi. I pojawia się problem, jak znaleźć wśród nich najkrótszą (najlepszą), czyli klasyczny problem komiwojażera. Jak odwiedzić *n* miast, pokonując najkrótszą drogę, i wrócić do miejsca, z którego się wyruszyło? Zakładamy przy tym, że wszystkie miasta mają bezpośrednie połączenia. Spróbujmy rozwiązać zadanie na konkretnym przykładzie.

	Konin	Gliwice	Poznań	Warszawa
Konin	0	250	105	220
Gliwice	250	0	343	321
Poznań	105	343	0	313
Warszawa	220	321	313	0

Najprostszym sposobem wydaje się sprawdzenie wszystkich możliwych przypadków, czyli permutacji ciągów, i wybór minimum. Ponieważ jednak dla ciągu *n*-elementowego liczba permutacji wynosi *n!/2* dla dużej liczby miast, jest to praktycznie niemożliwe.

Wskazówka

Do rozwiązania tego problemu wykorzystuje się algorytmy Kruskala lub Prima, które polegają na znalezieniu tzw. minimalnego drzewa rozpinającego (drzewo rozpinające zawiera wszystkie wierzchołki grafu, a zbiór krawędzi drzewa jest podzbiorem zbioru krawędzi grafu).

Minimalne drzewo ma sumę wag krawędzi najmniejszą z możliwych. Po znalezieniu minimalnego drzewa rozpinającego przechodzimy je metodą pre-order (patrz opis pod rysunkiem 1.6), zapisując wierzchołki, jeżeli odwiedzamy je po raz pierwszy, i sumując wagi poszczególnych krawędzi. Na koniec dodajemy wagę między pierwszym i ostatnim wierzchołkiem, bo przecież chcemy wrócić, prawda?

Algorytm Kruskala w odniesieniu do naszego przykładu

1. Dla każdego wierzchołka grafu tworzone jest drzewo jednostkowe. Dla ułatwienia będziemy używać jako nazw wierzchołków pierwszych liter nazw miejscowości (K — Konin, P — Poznań, W — Warszawa, G — Gliwice). Zbiór drzew tworzy las. Nasz las składa się z czterech drzew wierzchołkowych.

2. Tworzymy zbiór Z zawierający wszystkie krawędzie oryginalnego grafu:

   KG — 250

   KP — 105

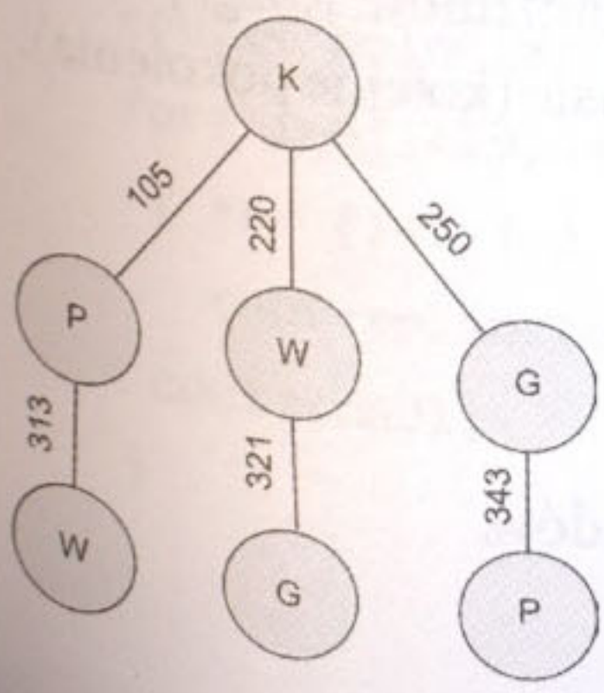
   KW — 220

   GP — 343

   GW — 321

   PW — 313

3. Do momentu, aż zbiór Z będzie pusty, wybieramy i usuwamy kolejno krawędzie o najmniejszej wadze. Jeżeli krawędź łączyła dwa różne drzewa, dodajemy ją do lasu (łączymy je). Otrzymamy w ten sposób minimalne drzewo rozpinające (rysunek 1.16).



Rysunek 1.16. Drzewo rozpinające

4. Przechodzenie pre-order — zapisujemy wierzchołki odwiedzane po raz pierwszy: K – P – W – G. Sumujemy krawędzie 105+313+321 i dodajemy wagę między K i G — 250.

Wskazówka

Dla niektórych grafów można znaleźć wiele drzew spełniających warunek drzewa minimalnego. Jeżeli myślisz, że to jeden z takich przypadków, masz rację ☺

Implementację tego algorytmu zostawiam najbardziej dociekliwym. Powodzenia!

1.11. Elementy teorii gier

Definicja

Teoria gier to dział matematyki zajmujący się badaniem optymalnych zachowań w przypadku sytuacji konfliktowych. Celem gracza w każdej grze jest oczywiście wygrana, która łączy się z osiągnięciem korzyści (zebranie skarbów, zdobycie władzy czy po prostu przetrwanie, przeżycie). Teorie gier mają zastosowanie w wielu dziedzinach. Ich zadaniem jest badanie strategii, jakie powinien przyjąć gracz, żeby osiągnąć jak najlepszy wynik.

Przy opracowywaniu strategii zawsze należy brać pod uwagę:

- zbiór graczy,
- reguły,
- możliwe straty,
- możliwe wyniki,
- wypłaty (korzyści).

Zajmiemy się analizą popularnych problemów w teorii gier — grą w życie i grą logiczną NIM.

1.11.1. Gra w życie

W drugiej połowie XX wieku John Conway wymyślił grę, która zdobyła bardzo dużą popularność. Gra jest modelem rodzenia się, ewolucji, śmierci kolonii organizmów. Rozgrywa się w przestrzeni (dwuwymiarowa plansza podzielona na komórki) i czasie (kolejne pokolenia).

Grę rozpoczynamy od „zasiedlenia" komórek.

Zasady gry według Conwaya:

- Martwa komórka, która ma trzech sąsiadów, rodzi się.
- Żywa komórka pozostaje żywa, jeżeli ma dwóch lub trzech sąsiadów.

- Jeżeli żywa komórka ma mniej niż dwóch sąsiadów, umiera z samotności.
- Jeżeli żywa komórka ma więcej niż trzech sąsiadów, umiera z zatłoczenia.

Do komputerowej realizacji tego algorytmu wykorzystamy tablicę, która będzie symulować kolonię żywych organizmów. Komórki niezamieszkałe zapełniamy literą „o", pojawienie się organizmu zaznaczamy literą „x". Potrzebna nam jest druga plansza, która przechowuje stan kolejnego pokolenia. Zawartości kolonii nie możemy zmieniać dynamicznie, bo przecież procesy „rodzenia się" i „umierania" zachodzą w tym samym czasie.

Wskazówka

Tworzymy dwie plansze o rozmiarze stosownym do symulacji, którą chcemy przeprowadzić. Wypełniamy je na początku „o" (oczyszczamy z intruzów ☺). Kontrolnie możemy wyświetlić, że wszystko jest gotowe.

```
char plansza[20][20];
char nowa_plansza [20][20];
int i,j,w,k,ilosc=0,pokolenie,p=0;
for( i=0;i<20;i++)
 { for ( j=0;j<20;j++)
  { plansza[i][j]='o';
    nowa_plansza [i][j]='o';
    cout<<plansza[i][j]<<" ";
  }
```

Następnie zasiedlamy kolonię, podając współrzędne wiersza (`w`) i kolumny (`k`), gdzie ma „pojawić się życie". Przy wyborze miejsc do zasiedlenia pamiętajmy, żeby w żadnym pokoleniu nie przekroczyć zakresu tablicy. Zmienna ilość określa, ile mamy żywych organizmów do zasiedlenia. Wyświetlimy sobie od razu schemat naszej kolonii.

```
cin>>ilosc;
for (int l=0; l<ilosc; l++)
{cin>>w;
 cin>>k;
 plansza[w][k]='x';
}
cout<<"Kolonia: "<<endl;
for( i=0;i<20;i++)
 { for (int j=0;j<20;j++)
    cout<<plansza[i][j]<<" ";
 cout<<endl;
}
```

Wskazówka

Pozostało nam napisanie fragmentu kodu, który zawiera zasady gry. Żeby nie „wyskoczyć" poza planszę, proponuję zacząć od indeksów równych 1 i skończyć na 15. Oczywiście można uzupełnić program o funkcję informującą, czy w aktualnym rozmiarze tablicy możliwy jest dalszy rozwój, do czego zachęcam.

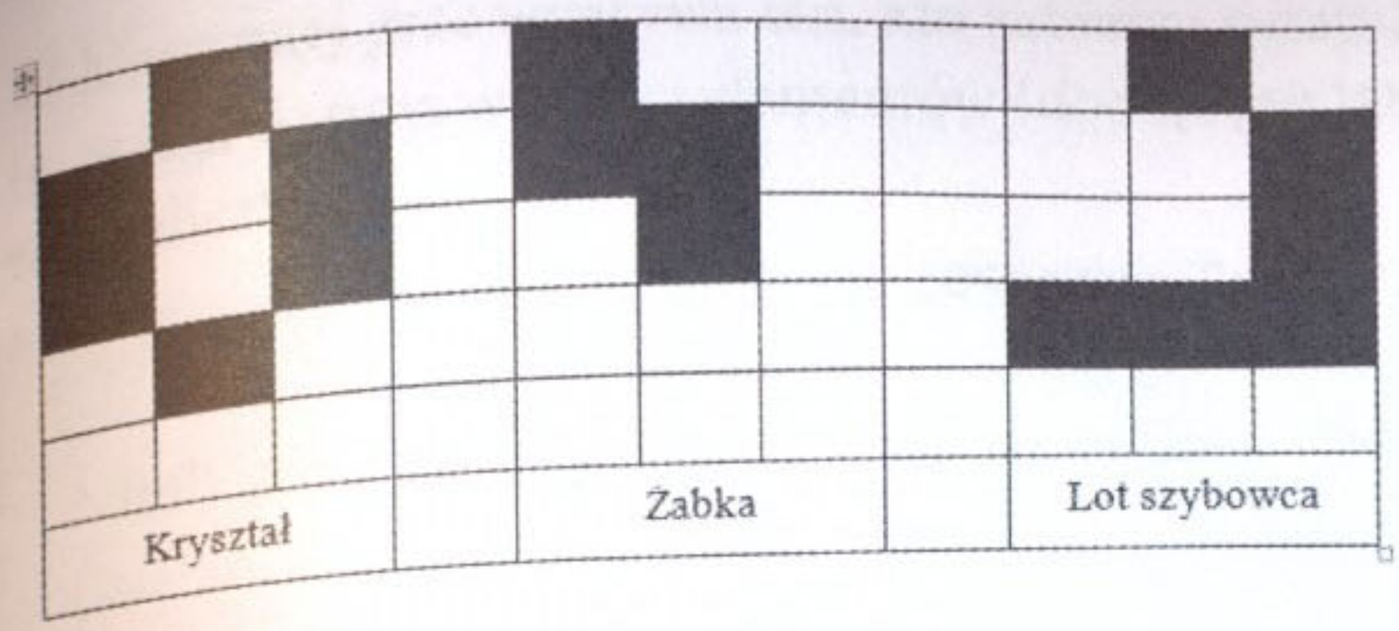
Poniższy fragment kodu wykona symulację dla jednego pokolenia. Jeżeli chcemy symulacji dla *n*-tego, należy wywołać go *n*-krotnie.

```
for (i=1; i<15;i++)
 for (j=1; j<15; j++)

//Zmienna zliczająca liczbę sąsiadów
  {ilosc=0;
  //Pojawia się nowe życie
  if (plansza[i-1][j-1]=='x') ilosc++;
  if (plansza[i-1][j]=='x') ilosc++;
  if (plansza[i-1][j+1]=='x') ilosc++;
  if (plansza[i][j-1]=='x') ilosc++;
  if (plansza[i][j+1]=='x') ilosc++;
  if (plansza[i+1][j-1]=='x') ilosc++;
  if (plansza[i+1][j]=='x') ilosc++;
  if (plansza[i+1][j+1]=='x') ilosc++;
  if (ilosc==3) nowa_plansza[i][j]='x';
  //Nic się nie zmienia
if (ilosc==2) nowa_plansza[i][j]=plansza[i][j];
//Organizm umiera z zatłoczenia
  if (ilosc>3) nowa_plansza[i][j]='o';
//Organizm umiera z samotności
 if (ilosc<2) nowa_plansza[i][j]='o';
 }
 //Funkcja kopiująca struktury, w konkretnym przypadku tablicę
 //„nowa_plansza" do „plansza" (kolejne pokolenie jest bazowym
 //dla następnego). Trzecim argumentem jest funkcja określająca
 //liczbę kopiowanych elementów
 memcpy(plansza, nowa_plansza, sizeof(plansza));
```

Mamy już wszystkie fragmenty gry. Pozostaje połączyć je w całość. Powodzenia!

Zadanie 1.28 Wykorzystując napisany program, sprawdź, jak będzie wyglądać kolonia bakterii w piątym pokoleniu dla struktur przedstawionych na rysunku 1.17.



Rysunek 1.17. Kolonie bakterii

Gra w życie jest przykładem automatu komórkowego. Automaty komórkowe stanowią już właściwie odrębny dział nauki i znajdują wiele zastosowań. Czy potrafisz wymienić przykłady?

1.11.2. Gra logiczna NIM

Czy znasz zabawę polegającą na zabieraniu przedmiotów (np. zapałek, kamieni) z kilku stosów (rzędów), taką że na starcie każdy miał różną liczbę elementów? W myśl zasad gry, wykonując ruch, gracz może zabrać dowolną liczbę elementów, ale tylko z jednej sterty. Ten, kto zabiera ostatni element, wygrywa lub przegrywa, w zależności od przyjętej koncepcji.

Zaimplementowanie takiej gry do komputera wydaje się banalne. Przeanalizuj umieszczony niżej przykład dla trzech rzędów po maksymalnie 10 elementów. Liczbę elementów do każdego rzędu losuje komputer. Podczas wyświetlania elementy są symbolizowane znakiem @. W implementacji zakładamy, że gracz potrafi liczyć i nie zabiera z rzędu więcej elementów, niż się w nim znajduje ☺

Gra NIM dla dwóch graczy; wygrywa osoba, która weźmie ostatni kamyczek

```
#include <iostream>
#include <cstdlib>
using namespace std;
void wyswietl(int i1,int i2,int i3)
{int i;
 for ( i=1; i<=i1; i++)
    cout<<"@"<<" ";
    cout<<endl;
 for (i=1; i<=i2; i++)
     cout<<"@"<<" ";
    cout<<endl;
   for (i=1; i<=i3; i++)
    cout<<"@"<<" ";
    cout<<endl;
}
```

```
int main ()
{int rzad1[10],rzad2[10],rzad3[10];;
 int rzad, ilosc;
 int i,i1=0,i2=0,i3=0,gracz1,gracz2,numer=0;
 srand(time(NULL));
 for (i=0; i<10; i++)
  {rzad1[i]=rand()%2;
   if (rzad1[i]==1) i1++;
   }
for (i=0; i<10; i++)
 {rzad2[i]=rand()%2;
  if (rzad2[i]==1) i2++;
  }
for (i=0; i<10; i++)
  {rzad3[i]=rand()%2;
   if (rzad3[i]==1) i3++;
  }
  wyswietl(i1,i2,i3);

while ((i1+i2+i3)>0)
 {numer++;
if (numer%2==0) cout<<"Gracz 2"<<endl;
else cout<<"Gracz 1"<<endl;
cout<<"podaj numer wiersza i ile zabierasz"<<endl;
cin>>rzad;
cin>>ilosc;
if (rzad==1) i1=i1-ilosc;
if (rzad==2) i2=i2-ilosc;
if (rzad==3) i3=i3-ilosc;
wyswietl(i1,i2,i3);
 if (i1+i2+i3==0) cout<<"Wygrana!!"<<endl;
}
 system("pause");
 return 0;
}
```

Przeprowadź symulację (możesz oczywiście zmienić liczbę wierszy i kamyków) i zastanów się nad zwycięską strategią. Czy potrafisz napisać program, w którym jednym z grających jest komputer?

W klasycznej grze wygrywa ten, kto zabierze ostatni element. Strategią wygrywającą jest ta, w której suma wartości elementów (dodawanie nimliczb) jest równa 0. Tylko... co to są te nimliczby?

Wskazówka

Nimliczby — liczby, które różnią się od liczb naturalnych sposobem wykonywania działań.

1.11.3. Dodawanie nimliczb

- Suma dwóch równych nimliczb wynosi 0: $7 \oplus 7 = 0$.
- Jeżeli większa z nimliczb jest potęgą dwójki, dodaje się je jak zwykłe liczby: $8 \oplus 7 = 15$.

Dodawanie nimliczb odpowiada operacji XOR (w C++ operator ^) na cyfrach rozwinięcia dwójkowego danej liczby.

Dodajmy dwie liczby:

$37_D = 100101_B$

$51_D = 110011_B$

Wskazówka

Wynikiem operacji XOR jest prawda wtedy, gdy oba elementy są różne (1 XOR 1 = FAŁSZ, 0 XOR 0 = FAŁSZ, 0 XOR 1 = PRAWDA, 1 XOR 2 = PRAWDA).

XOR	100101_B	37_D
	110011_B	51_D
NIM — suma	010110_B	22_D

$37 \oplus 51 = 22$

Powyższe działanie w C++ zapisujemy tak:

```
wynik = 37 ^ 51;
```

1.11.4. Mnożenie nimliczb

- Jeżeli większa z nimliczb jest równa elementowi ciągu 1, 2, 4, 16, 256, …, to mnożenie odbywa się na zwykłych zasadach.
- Jeżeli nimliczbę mnoży się przez siebie, to wynik jest równy sumie dwóch nimliczb — jej samej i części całkowitej jej połowy: $9 \odot 9 = 9 \oplus 9/2 = 9 \oplus 4 = 13$.

1.11.5. Potęgowanie nimliczb

- Potęgowanie odbywa się na standardowych zasadach — poprzez wielokrotne mnożenie nimliczb, czyli:

$$x^3 = x^* \xi^* x.$$

Zadanie 1.29 Napisz program — kalkulator wykonujący operacje dodawania, mnożenia i potęgowania nimliczb. Korzystanie z niego podczas gry w NIM ułatwi wygrywanie ☺

Powróćmy do strategii wygrywającej. Mamy trzy rzędy elementów.

Numer rzędu	Elementy	Ilość w systemie dziesiętnym	Ilość w systemie binarnym
1	I I I I I	5	101
2	I I I I I I I	7	111
3	I I I	3	011
NIM suma (operacja XOR)		5^7^3 = 1	001

Żeby NIM suma była równa 0, wystarczy z dowolnego rzędu zabrać jeden element. Znajdziemy się wtedy na pozycji wygrywającej.

Istnieje ryzyko, że liczba elementów do zabrania przekracza liczbę elementów w największym stosie. Jak w tym przykładzie.

Numer rzędu	Elementy	Ilość w systemie dziesiętnym	Ilość w systemie binarnym
1	I I I I I I I I I	9	1001
2	I I I I I I I	7	0111
3	I I I	3	0011
NIM suma (operacja XOR)		9^7^3 = 13	1101

W takiej sytuacji należy od otrzymanej liczby odjąć liczbę elementów w największym stosie i zostawić na nim liczbę elementów równą otrzymanej różnicy:

13–9 = 4

Zabieramy 9–4 = 5

Numer rzędu	Elementy	Ilość w systemie dziesiętnym	Ilość w systemie binarnym
1	I I I I	4	100
2	I I I I I I I	7	111
3	I I I	3	011
NIM suma (operacja XOR)		4^7^3 = 0	000

TP **Zadanie 1.30** Napisz program pt. „Gra NIM" dla trzech stosów po 10 elementów. Jednym z grających jest komputer.

Zadanie 1.31 Zmień w swoim programie warunek wygranej — ten, kto zabiera ostatni element, przegrywa.

1.12. Sprawdź się — zadania na koniec rozdziału

Zadanie 1.32 FTP Każdy ułamek zwykły postaci $\frac{p}{q}$ (gdzie p i q są liczbami całkowitymi, względnie pierwszymi) można przedstawić w postaci ułamka łańcuchowego:

$$\frac{p}{q} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \ldots + \cfrac{1}{a_n}}}$$

Napisz program, który dla podanego p (licznika) i q (mianownika) wypisuje ciąg liczb a_0, a_1, ..., a_n.

Dane wejściowe:

Liczby naturalne p i q oznaczające odpowiednio licznik i mianownik.

Wyjście:

Ciąg ułamków postaci a/b.

Przykładowy rezultat:

Dla danych wejściowych:

9 11

Wynik:

1 4 2

Wskazówka

Funkcję najlepiej zdefiniować rekurencyjnie. Algorytm sprowadza się do „odwracania ułamka" i wyłączania całości, dopóki licznik nie osiągnie wartości 1.

Zadanie 1.33 FTP Silnię podwójną oznacza się jako $n!!$ Jest to iloczyn kolejnych liczb naturalnych do n z krokiem 2.

Na przykład:

6!! = 2*4*6

9!! = 1*3*5*7*9

Jeżeli $n = 0$ lub 1, to $n!! = 1$.

Napisz program, który oblicza podwójną silnię podanej liczby naturalnej *N*.

Dane wejściowe:

N

Wyjście:

N!!

Przykładowy rezultat:

Dla danych wejściowych:

5

Wynik:

15

TP **Zadanie 1.34** Wartość silni liczby *N* od $N = 5$ kończy się co najmniej jednym zerem. Napisz program, który oblicza liczbę zer w wartości *N*! dla $N < 1\,500\,000$.

Dane wejściowe:

N

Wyjście:

Liczba naturalna — liczba zer w *N*!

Przykładowy rezultat:

Dla danych wejściowych:

26

Wynik:

6

Wskazówka

W tym przypadku (jak i w wielu innych) czas wykonywania programu skracamy, stosując tzw. preprocessing, czyli mówiąc inaczej — przetwarzanie wstępne. Możemy wykonać obliczenia potrzebne do realizacji konkretnego algorytmu w innym programie i wykorzystać wyniki w ostatecznym.

FTP **Zadanie 1.35** Napisz program, który zwraca ostatnią niezerową cyfrę w *N*!

Dane wejściowe:

N

Wyjście:

Cyfra — ostatnia niezerowa cyfra w *N*!

Przykładowy rezultat:

Dla danych wejściowych:

26

Wynik:

4

Zadanie 1.36 Liczbę nazywamy doskonałą, jeżeli jest równa sumie swoich dzielników mniejszych od niej samej (np. 6 = 1+2+3). Napisz program, który dla podanego zakresu od 2 do *N* (2 < *N* < 10 000) wypisuje wszystkie liczby doskonałe.

Dane wejściowe:

N

Wyjście:

Ciąg liczb doskonałych oddzielonych spacjami z zakresu od 2 do *N*.

Przykładowy rezultat:

Dla danych wejściowych:

100

Wynik:

6 28

Te dwie liczby były znane już w starożytności. Istnieją teorie wiążące 6 z liczbą dni, w których Bóg stworzył świat, a 28 z liczbą nocy, których potrzebuje Księżyc na obiegnięcie Ziemi.

Zadanie 1.37 Liczby zaprzyjaźnione są to pary liczb naturalnych, w których suma dzielników jednej (oprócz samej liczby) równa się drugiej. Pierwszą taką parę znalazł Pitagoras — 220 i 284.

Napisz program, który w podanym uporządkowanym ciągu N ($2 < N < 100$) liczb $a_1, a_2, ..., a_n$ ($a_n < 10\,000$) znajduje liczby zaprzyjaźnione.

Dane wejściowe:

N — ilość liczb.

$a_1, a_2, ..., a_n$ — ciąg liczb.

Wyjście:

W kolejnych wierszach pary liczb zaprzyjaźnionych rozdzielone spacją lub komunikat „Nie ma takich liczb".

Przykładowy rezultat:

Dla danych wejściowych:

10

10 208 220 245 284 678 1001 1184 1210 2000

Wynik:

220 284

1184 1210

Zadanie 1.38 Liczbę naturalną nazywamy szczęśliwą, jeżeli ma parzystą liczbę cyfr, a suma cyfr składających się na jej pierwszą połowę jest równa sumie drugiej połowy. Napisz program, który sprawdza, czy wprowadzona liczba n jest szczęśliwa.

Dane wejściowe:

n

Wyjście:

TAK lub NIE.

Przykładowy rezultat:

Dla danych wejściowych:

123321

Wynik:

TAK

Zadanie 1.39 Dodatnią liczbę całkowitą nazywamy antypierwszą, jeżeli ma więcej dzielników niż każda liczba dodatnia mniejsza od niej.

Przykłady: 1(1), 2(2), 4(3), 6(4).

Napisz program, który dla danego przedziału od 1 do $N < 1000$ zlicza liczby antypierwsze.

Dane wejściowe:

N

Wyjście:

Liczba naturalna — ilość liczb antypierwszych.

Przykładowy rezultat:

Dla danych wejściowych:

10

Wynik:

4

Zadanie 1.40 Słowa Fibonacciego definiujemy podobnie jak liczby Fibonacciego:

FIB1 = *b* FIB2 = *a* FIBn = FIBn–1*FIBn–2 dla $k >= 1$, gdzie * jest znakiem operacji łączenia (konkatenacji) słów.

Przykład kilku kolejnych słów:

FIB3 = *ab*; FIB4 = *aba*; FIB5 = *abaab*; FIB6 = *abaababa*

Napisz program, który wczytuje słowo składające się z ciągu liter *a*, *b* oraz *n* — numeru słowa Fibonacciego i oblicza, ile razy wprowadzone słowo występuje w słowie Fibonacciego (słowa mogą na siebie nachodzić). Zakładamy, że słowo podane na wejściu jest podsłowem *n*-tego słowa.

Dane wejściowe:

tekst — podsłowo, *n* — numer słowa Fibonacciego.

Wyjście:

Liczba naturalna — liczba wystąpień podsłowa.

Przykładowy rezultat:

Dla danych wejściowych:

6 aba

Wynik:

3

Wskazówka

To zadanie pojawiło się na III Olimpiadzie Informatycznej.

Najbardziej optymalnym rozwiązaniem jest zastosowanie algorytmu Knutha-Morrisa-Pratta. Szczegółowy opis algorytmu znajdziesz w Wikipedii.

Zadanie 1.41 Jak otrzymujemy ułamki okresowe ze zwykłych, wie każdy. A jak przebiega proces odwrotny? Ponieważ każdy ułamek okresowy można zamienić na zwykły, istnieje oczywiście algorytm, który to potrafi.

Wskazówka

Wykonaj kilka prób zamiany ułamków okresowych na zwykłe (np. 0,12(5); 3,(27); 0,(1)). Czy odkryłeś algorytm?

FTP

Zadanie 1.42 Poprawność podstawowych numerów identyfikacyjnych, takich jak NIP, PESEL, REGON, sprawdza się za pomocą tzw. sum kontrolnych, które oblicza się za pomocą specjalnych algorytmów. Suma kontrolna jest przesyłana razem z danymi i służy do sprawdzania ich poprawności. W przypadku numeru PESEL są to ostatnie cyfry — 11. Dziesiąta cyfra oznacza płeć (parzysta — kobieta, nieparzysta — mężczyzna).

Sumę kontrolną w numerze PESEL obliczamy według schematu:

- cyfrę z przedziału od 1 do 10 mnożymy przez odpowiadającą jej wagę z tablicy wag postaci (1, 3, 7, 9, 1, 3, 7, 9, 1, 3);
- wagi sumujemy;
- obliczamy resztę z dzielenia sumy przez 10. Jeżeli reszta wynosi 0 — suma kontrolna równa się 0. W innym przypadku jest to wartość będąca uzupełnieniem reszty do 10.

Napisz program, który sprawdza poprawność wprowadzonego numeru PESEL i podaje płeć jej właściciela, korzystając z oznaczeń: F — kobieta, M — mężczyzna. Jeżeli wprowadzony numer jest niepoprawny, powinien pojawić się komunikat „NIEPOPRAWNE DANE".

Dane wejściowe:

Ciąg cyfr naturalnych rozdzielanych spacją — numer PESEL.

Wyjście:

F lub M — płeć właściciela lub komunikat „NIEPOPRAWNE DANE".

Przykładowy rezultat:

Dla danych wejściowych:

7 5 1 2 1 9 6 8 6 2 9

Wynik:

F

Zadanie 1.43 Mamy ciąg n liczb naturalnych $0 \leq n \leq 10^9$ $a_{1, a2,}$ ..., a_n, gdzie $a_i \leq 1000$. Napisz program sprawdzający, czy z każdej dowolnie wybranej trójki liczb można zbudować trójkąt.

TP

Dane wejściowe:

n — ilość liczb.

$a_1, a_2, \ldots, a_n$ — ciąg liczb naturalnych.

Wyjście:

TAK — jeżeli można zbudować trójkąt.

NIE — w przeciwnym wypadku.

Przykładowy rezultat:

Dla danych wejściowych:

6

12

23

45

34

56

67

Wynik:

NIE

Zadanie 1.44 Czy wiesz, w jaki sposób jest weryfikowany kod PIN karty bankowej? Sprawdzana jest zgodność numeru karty i PIN-u przez moduł HSM (*Hardware Security Model*). Oprócz kodu PIN i numeru karty moduł ten potrzebuje tablicy konwersji — 16-elementowej tablicy cyfr od 0 do 9.

Schemat działania modułu HSM:

- szyfrowany jest numer karty, w wyniku czego otrzymuje się liczbę zapisaną szesnastkowo;
- z otrzymanej liczby zostawia się tylko cztery pierwsze cyfry szesnastkowe;
- pozostawione cyfry zamienia się na dziesiętne z wykorzystaniem tablicy konwersji (cyfrze szesnastkowej przypisuje się odpowiednik dziesiętny, który znajduje się w komórce tablicy o równym jej indeksie).

Tak otrzymany numer powinien być zgodny z kodem PIN.

Przykład

Standardowa tablica konwersji T = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5).

Zaszyfrowany numer karty N — A8B3 1022 E223 1022.

Zostawiamy cztery pierwsze cyfry: A8B3.

Zamieniamy je na dziesiętne.

Znak	A	8	B	3
Indeks tablicy	10	8	11	3
Wartość	9	7	0	2

PIN — 9702

Napisz program, który na podstawie zaszyfrowanego numeru karty oblicza kod PIN.

Zadanie 1.45 ROT13 jest prostym szyfrem przesuwającym, polegającym na zamianie znaku tekstu jawnego na znak występujący 13 pozycji po nim. Napisz program szyfrujący przy użyciu ROT13.

Zadanie 1.46 Czy słyszałeś o liście Hilberta? W 1900 roku odbył się w Paryżu Międzynarodowy Kongres Matematyków, na którym David Hilbert przedstawił 23 nierozwiązane zagadnienia matematyczne. Do tej pory niektóre z nich czekają na rozwiązanie, jak słynna hipoteza Goldbacha dotycząca liczb pierwszych. Goldbach stwierdził, że każdą parzystą liczbę naturalną większą od 2 można przedstawić jako sumę dwóch liczb pierwszych[2] (np. 14 = 7+7; 20 = 13+7 lub 17+3). Ta teoria do tej pory nie została ani potwierdzona, ani obalona... Może to właśnie Twoje nazwisko pojawi się na liście Hilberta w kolumnie „aktualny status" ☺

Zanim przystąpisz do zapisywania się na liście odkryć naukowych, napisz program, który czyta parzystą liczbę naturalną i rozkłada ją na sumę liczb pierwszych.

Dane wejściowe:

Parzysta liczba naturalna *N*.

---

2 Goldbach uważał 1 za liczbę pierwszą. Ponieważ obecnie tak się nie uważa, hipoteza dotyczy liczb większych od 5.

Wyjście:

Ciąg liczb pierwszych.

Przykładowy rezultat:

Dla danych wejściowych:

46

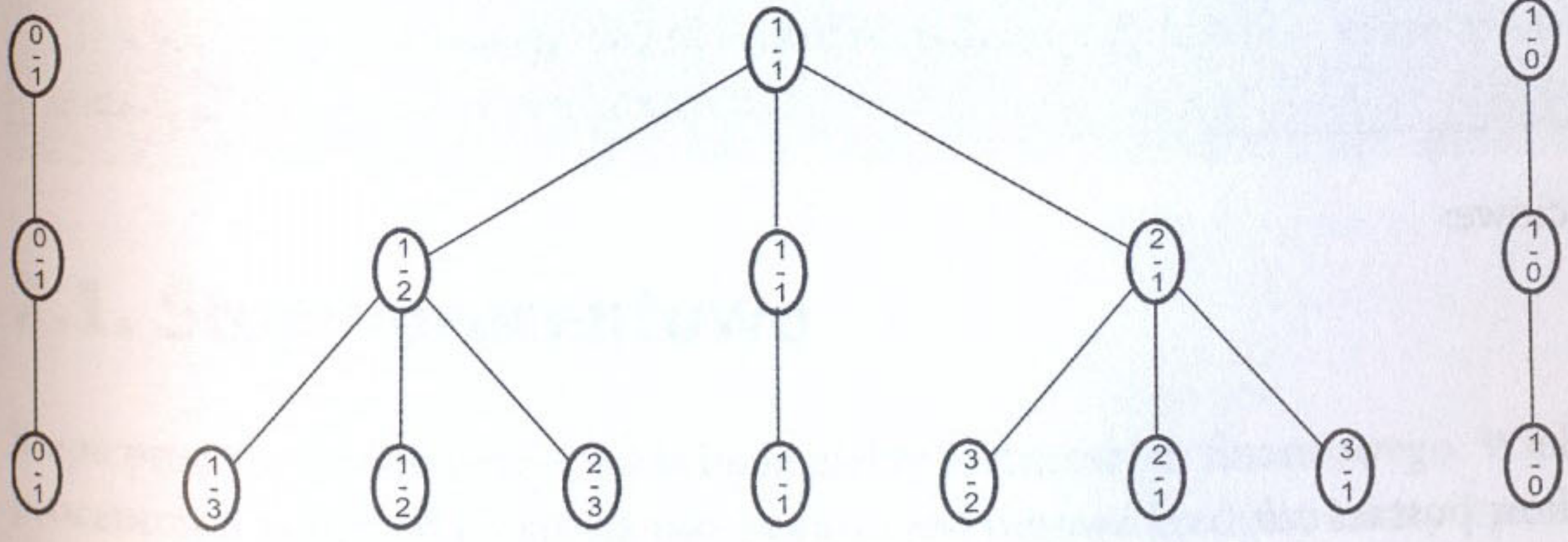
Wynik:

23 23

TP **Zadanie 1.47** Drzewo Sterna-Brocota jest to konstrukcja liczbowa pozwalająca na uzyskanie dowolnej liczby wymiernej. Zaczynamy od sekwencji dwóch ułamków symbolizujących odpowiednio 0 i nieskończoność. Następnie na kolejnym poziomie wstawiamy między nie ułamek o liczniku równym sumie liczników i mianowniku równym sumie mianowników (rysunek 1.18).

Wskazówka

Wykorzystujemy tzw. medianty. Dla ułamków $X = \frac{a}{b}$ i $Y = \frac{c}{d}$ mediantem jest ułamek $Z = \frac{a+c}{b+d}$.



Rysunek 1.18. Drzewo Sterna-Brocota

Własności drzewa:

- każda liczba wymierna występuje tylko raz;
- w drzewie występują wszystkie dodatnie liczby wymierne, czyli każdą liczbę można zapisać jako ścieżkę od korzenia do liczby.

Achille Brocot był z zawodu zegarmistrzem. Poszukaj odpowiedzi na pytanie, dlaczego zainteresował się uporządkowaniem ułamków w drzewo. Tym samym poznasz praktyczne zastosowanie tej struktury.

Napisz program, który dla podanego N wypisuje liczbę elementów (przyjmujemy, że na poziomie 0 mamy sekwencję dwóch ułamków: $\frac{0}{1}$ i $\frac{1}{0}$).

Dane wejściowe:

Liczba naturalna N ($2 \leq N \leq 20$).

Wyjście:

Liczba naturalna — liczba elementów w drzewie na N-tym poziomie.

Przykładowy rezultat:

Dla danych wejściowych:

4

Wynik:

17

TP **Zadanie 1.48** Napisz program wypisujący wszystkie ułamki z przedziału [0, 1] o mianownikach nieprzekraczających N, posortowane rosnąco.

Wskazówka

Otrzymany ciąg jest ciągiem Fareya.

Dane wejściowe:

N

Wyjście:

Ciąg ułamków postaci a/b.

Przykładowy rezultat:

Dla danych wejściowych:

8

Wynik:

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