

Jerzy Pokojski (red.)
Janusz Bonarowski, Jacek Jusis

Algorytmy

Warszawa 2010



KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY



Politechnika Warszawska
Wydział Samochodów i Maszyn Roboczych
Kierunek studiów "Edukacja techniczno informatyczna"
02-524 Warszawa, ul. Narbutta 84, tel. (22) 849 43 07, (22) 234 83 48
ipbmvr.simr.pw.edu.pl/spin/, e-mail: sto@simr.pw.edu.pl

Opiniodawca: dr hab. inż. Wojciech SKARKA prof. nzw. w Politechnice Śląskiej

Projekt okładki: Norbert SKUMIAŁ, Stefan TOMASZEK

Projekt układu graficznego tekstu: Grzegorz LINKIEWICZ

Skład tekstu: Janusz BONAROWSKI

Publikacja bezpłatna, przeznaczona dla studentów kierunku studiów
"Edukacja techniczno informatyczna"

Copyright © 2011 Politechnika Warszawska

Utwór w całości ani we fragmentach nie może być powielany
ani rozpowszechniany za pomocą urządzeń elektronicznych, mechanicznych,
kopiujących, nagrywających i innych bez pisemnej zgody posiadacza praw
autorskich.

ISBN 83-8970357-2

Druk i oprawa: Drukarnia Expol P. Rybiński, J. Dąbek Spółka Jawna,
87-800 Włocławek, ul. Brzeska 4

Spis treści

Wstęp.....	5
1. Podstawowe cechy algorytmów.....	7
1.1. Wstęp.....	8
1.2. Podstawowe cechy algorytmów i ich formy zapisu.....	9
2. Zmienne, typy danych, operatory, instrukcja warunkowa, instrukcja cyklu	11
3. Podstawowe algorytmy obliczeniowe. Przykłady z mechaniki i PKM	23
3.1. Wprowadzenie.....	24
3.2. Przykład zamodelowania zadania z mechaniki	26
3.3. Przykład zamodelowania zadania z Podstaw Konstrukcji Maszyn.....	32
3.4. Podsumowanie	40
4. Algorytmy generujące	41
4.1. Wprowadzenie.....	42
4.2. Metoda Monte Carlo – generowanie liczb losowych w zadanym zakresie.....	43
4.3. Metoda Monte Carlo – prosty generator liczb losowych (pseudolosowych).....	46
4.4. Złoty podział odcinka	50
4.5. Wygenerowanie n wyrazów ciągu Fibonacciego.....	53
4.6. Napisanie tekstu wspak	56
4.7. Zamiana liczby dziesiętnej na liczbę o innej podstawie i odwrotnie	58
5. Operacje geometryczne	63
5.1. Wprowadzenie.....	64
5.2. Budowanie trójkątów	64

6. Selekcja 69

- 6.1. Wprowadzenie.....70
- 6.2. Szukanie najmniejszej lub największej liczby w zbiorze70
- 6.3. Największa objętość stożka.....74

7. Algorytmy matematyczne 77

- 7.1. Wprowadzenie.....78
- 7.2. Obliczenie wartości liczby PI z zadaną dokładnością ...78
- 7.3. Liczby pierwsze.....81
- 7.4. Obliczenie wartości pierwiastka kwadratowego metodą Newtona z zadaną dokładnością.....90
- 7.5. Znajdowania pierwiastków równania (kwadratowego) drugiego stopnia.....93
- 7.6 Obliczenie silni.....97
- 7.7. Algorytm Euklidesa znajdowania największego wspólnego dzielnika100
- 7.8. Szyfrowanie i rozszyfrowywanie tekstu przy wykorzystaniu „Kodu Cezara”.....103

8. Algorytmy numeryczne 109

- 8.1. Wprowadzenie.....110
- 8.2. Metoda Monte Carlo – obliczenie całki oznaczonej110
- 8.3. Metoda bisekcji (połowienia przedziału) znajdowania pierwiastków algebraicznego równania nieliniowego ..115
- 8.4. Metoda fałsi (siecznej) znajdowania pierwiastków algebraicznego równania nieliniowego121
- 8.5. Metoda Monte Carlo – znajdowanie współczynników funkcji aproksymującej127

9. Sortowanie 133

- 9.1. Wprowadzenie.....134
- 9.2. Sortowanie przez wybieranie.....134
- 9.3. Sortowanie - algorytm bąbelkowy138

10. Literatura.....	143
----------------------------	------------

Wstęp

Niniejsze materiały zostały opracowane w ramach realizacji Programu Rozwojowego Politechniki Warszawskiej współfinansowanego ze środków PROGRAMU OPERACYJNEGO KAPITAŁ LUDZKI. Przeznaczone są dla studentów studiów inżynierskich na kierunku „Edukacja techniczno-informatyczna” na Wydziale Samochodów i Maszyn Roboczych Politechniki Warszawskiej.

Celem opracowania było przedstawienie algorytmów programów, które mogą być wykorzystywane przez inżynierów zajmujących się problematyką projektową w budowie maszyn.

W koncepcji doboru treści oraz sposobu prezentacji poszczególnych zagadnień przyjęto również założenie, że czytelnik poza elementarną znajomością języka MS Visual Basic, wyniesioną z laboratorium Technik Komputerowych na I roku studiów, posiada także pewne umiejętności w zakresie pisania niewielkich programów w tym języku. W rozdziałach 1, 2 zawarto powtórzenia oraz przypomnienia z zakresu podstawowych zagadnień programowania algorytmicznego. Rozdział 3 to wprowadzenie w obszar tworzenia aplikacji algorytmicznych wspomagających prace typowo inżynierskie.

Koncepcja rozdziałów 1, 2 i 3 zakłada stopniowe opanowywanie zagadnień niezbędnych w procesie budowy własnego oprogramowania. W zasadzie wątek rozwoju poszczególnych klas aplikacji inżynierskich kończy się w rozdziale 3. Dalej zrezygnowano z prezentacji coraz bardziej złożonych i skomplikowanych, i na ogół obszernych przykładów tych aplikacji na rzecz szczegółowego omówienia grup algorytmów szczególnie przydatnych w rozpatrywanej klasie zastosowań. Są to rozdziały 4 - 9.

Przyjęte w pracy założenia umożliwiają stosunkowo szybkie i skuteczne opanowanie, na niezbędnym poziomie strony warsztatowej, procesu tworzenia algorytmów. Dalsze poznawanie tej dziedziny powinno się opierać na projektach realizowanych indywidualnie przez studentów pod opieką prowadzących. Tematyka tych projektów powinna nawiązywać do zadań typowo inżynierskich, w miarę możliwości zaczerpniętych z realnej praktyki przemysłowej. Zróżnicowanie tematów, konwencja indywidualnie realizowanego projektu na ogół sprzyjają zarówno jakości opanowania podstaw jak i rozwojowi kreatywności samych słuchaczy.

Wspomniana koncepcja prezentacji zagadnień związanych z budową algorytmów i pisaniem oprogramowania, oraz powiązania tych zagadnień z zadaniami projektowymi zastała wypracowana przez autorów w trakcie blisko 25 lat nauczania zagadnień dotyczących tworzenia algorytmów i programów komputerowych na Wydziale Samochodów i Maszyn Roboczych Politechniki Warszawskiej w ramach Studiów Podyplomowych: „Komputerowo Wspomagane Projektowanie Maszyn”, „Informatyki dla Nauczycieli” oraz przedmiotów realizowanych, na studiach stacjonarnych i niestacjonarnych, specjalności „Wspomaganie Komputerowe Prac Inżynierskich”.

W opracowaniu uwzględniono następujące grupy zagadnień: podstawy języków algorytmicznych programowania, algorytmy tworzone na potrzeby mechaniki i wspomagania procesów projektowych, algorytmy generowania, algorytmy selekcji i optymalizacji, algorytmy związane z modelowaniem geometrycznym, algorytmy problemów matematycznych, algorytmy numeryczne.

W pracy przyjęto, że algorytmy prezentowane są w postaci pseudokodu, schematów blokowych i przykładowych programów napisanych w języku MS Visual Basic. Ze względu na dużą popularność języka programowania MS Visual Basic oraz fakt, że jest to jedno z popularniejszych narzędzi używanych do tworzenia aplikacji inżynierskich, często zintegrowanych z innym oprogramowaniem - m.in. komercyjnymi systemami CAD/CAE/CAM, znaczną część przykładów zaprezentowano właśnie w tym języku. Zastosowanie języka MS Visual Basic wzięło się także stąd, że przykłady napisane w konkretnym języku programowania odznaczają się wysokim poziomem jednoznaczności co jest szczególnie ważne w procesie tworzenia i testowania algorytmów w zastosowaniach inżynierskich.

Autorami poszczególnych rozdziałów są: Jerzy Pokojski (rozdziały 1, 2, 3,), Janusz Bonarowski (rozdziały 4 - 9), Jacek Jusis (rozdziały 4, 6 - 8).



Podstawowe cechy algorytmów

1.1. Wstęp

II połowa XX wieku przyniosła powszechność zastosowań komputerów w różnych obszarach działalności człowieka. Zmiany te poważnie wpłynęły na jego życie zawodowe i prywatne. Obecnie, komputery stosowane są w przechowywaniu i udostępnianiu informacji, w projektowaniu, wytwarzaniu, zarządzaniu, komunikowaniu się, itd.

Komputery i implementowane na nich oprogramowanie stały się nieodłącznym elementem praktycznie każdego warsztatu zawodowego. Rozwinięte metody i oprogramowanie poważnie wpłynęły na postać realizacyjną podejmowanych zadań. W wielu przypadkach skróceniu uległy czasy ich wykonywania, poprawiona została jakość przedsięwzięcia, całość stała się bardziej efektywna i mniej podatna na popełnianie błędów.

W charakteryzowanym okresie nastąpiło szereg zmian w koncepcjach budowy oprogramowania komputerowego. Początkowo oprogramowanie było tworzone, przede wszystkim przez poszczególne firmy, z myślą o zaspokojeniu własnych potrzeb. Oprogramowanie było w znacznym stopniu oparte na firmowym know-how i stanowiło jego realną egzemplifikację. Prowadziło to w wielu przypadkach do dublowania wysiłków - w różnych instytucjach powstawały podobne lub bardzo zbliżone rozwiązania software'owe.

Z czasem, na rynku zaistnieli komercyjni dostawcy oprogramowania. Zaczęto oferować rozwiązania bardziej uniwersalne. Ich propozycje były na ogół tańsze od oprogramowania wykonywanego własnymi, firmowymi siłami. W praktyce rozwiązanie to stawało się coraz bardziej popularne. Bardzo szybko okazało się jednak, że po to aby efektywnie implementować oprogramowanie komercyjne konieczne jest jego przystosowanie do indywidualnych, firmowych potrzeb. Zaczęto oferować narzędzia programistyczne pozwalające na rozbudowę systemów komercyjnych o nowe moduły uwzględniające specyfikę indywidualnych rozwiązań.

Obecnie, najczęściej możemy spotkać koegzystencję modułów software'owych będących oprogramowaniem komercyjnym z modułami zbudowanymi przez użytkowników lub w oparciu o ich szczegółowe koncepcje. Stąd też w edukacji inżynierów ważną rolę odgrywa umie-

jętność tworzenia koncepcji modułów programistycznych, które stanowią odbicie ich własnego, inżynierskiego know-how. Umiejętność ta może ograniczać się do budowy samych algorytmów wyrażanych za pomocą pseudokodu lub schematów blokowych, może również być poszerzona na obszar pisania aplikacji modelowych, pilotażowych, prototypowych w konkretnym języku programowania. Ostatnie rozwiązanie jest dzisiaj szeroko praktykowane. Dlatego w niniejszym opracowaniu przyjęto, że dominującą formą prezentacji będą przykłady przygotowane w języku MS Visual Basic. Dla zrozumienia przedstawianych zagadnień wystarczy elementarna znajomość tego języka.

1.2. Podstawowe cechy algorytmów i ich formy zapisu

Stosowanie środków komputerowych we wspomaganiu określonych obszarów ludzkiej działalności wiąże się z koniecznością zapewnienia zarówno elementów hardware'owych jak i software'owych. O ile elementy hardware'owe, w przypadku danego projektu, przeważnie dobierane są w oparciu o ofertę rynkową konkretnych producentów to oprogramowanie może stanowić przedmiot dosyć złożonych rozważań i decyzji. Stosowane oprogramowanie może być: 1) oprogramowaniem komercyjnym, już istniejącym, oferowanym na rynku, 2) oprogramowaniem tworzonym przez inny podmiot na zamówienie, 3) oprogramowaniem wykonywanym przez przyszłego jego użytkownika.

Właściwe rozwiązanie jest dobierane pod kątem jego dostosowania do realizacji konkretnych zadań. Ważną rolę odgrywa w tym procesie również aspekt ekonomiczny. Najczęściej brane są pod uwagę możliwości merytoryczne samego oprogramowania, zastosowane metody przetwarzania, oferowany serwis producenta oprogramowania, jego pozycja rynkowa, itd. Jeżeli jest to oprogramowanie komercyjne to bardzo trudne jest w tym przypadku określenie dokładnego sposobu przetwarzania realizowanego przez to oprogramowanie. Przeważnie producenci nie ujawniają szczegółów swoich dokonań programistycznych. Oprogramowanie oferowane komercyjnie znamy najczęściej z opisów pochodzących od producenta, charakterystyk prób jego stosowania, z opinii innych użytkowników, własnych doświadczeń. Właściwie możemy się

jedynie domyślać jak jest ono zbudowane z punktu widzenia zastosowanych rozwiązań programistycznych. Równie rzadko producenci oprogramowania udostępniają informacje na temat metod i metodologii, które zostały zaimplementowane w ich produktach.

Oprogramowanie oferowane komercyjnie najczęściej pretenduje do uniwersalności proponowanych podejść. Na ogół również takie możliwości ma. Często zdarza się jednak, że problemy, do których realnie ma być zastosowane to oprogramowanie są znacznie bardziej złożone, czasem mogą zawierać również jakieś specyficzne aspekty nieobecne w oprogramowaniu standardowym. Zachodzi wówczas potrzeba rozbudowy oprogramowania komercyjnego o nowe możliwości, o nowe jego funkcjonalności. Taka rozbudowa może być zrealizowana na drodze zamówienia, u kogoś, właściwego rozszerzenia programu komercyjnego, które ma być zintegrowane z programem komercyjnym w określony sposób. Może się również zdarzyć, że zachodzi potrzeba stworzenia oprogramowania, które ma być wykonane od podstaw ze względu na jego stronę merytoryczną czy też ekonomiczną.

Programy komputerowe to ciągi instrukcji, które są kolejno wykonywane przez komputer. Te ciągi instrukcji muszą nawiązywać do rzeczywistości, do tego jak ma być realizowane konkretne przetwarzanie. Zachodzi zatem potrzeba uchwycenia tej rzeczywistości, matematycznego jej opisanie. Dalszy krok to zaproponowanie sposobu przetwarzania krok po kroku. Można ten sposób przetwarzania opisywać bezpośrednio, używając rozkazów określonego języka programowania – budując program w tym języku. Można posłużyć się pewną formą zapisu, która służy przede wszystkim wyrażaniu głównych idei przetwarzania, a która zwana jest algorytmem. Zwykle w opisie postępowania - algorytmie zawarte są dwie grupy składników: 1) składniki opisujące obiekty, które są przetwarzane, 2) składniki, które stanowią opis działań, które mają być kolejno wykonywane.

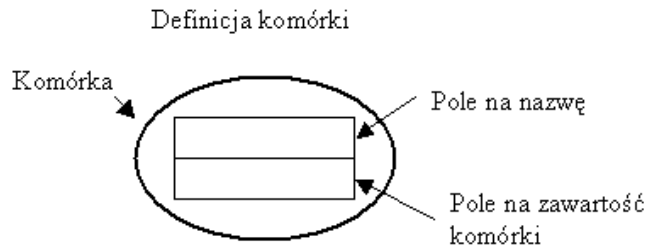
Algorytmy można przedstawiać w formie schematów blokowych. Jest to postać graficzna, składająca się z figur geometrycznych, które zawierają opisy kolejnych działań, a z kolei które są ze sobą powiązane za pomocą sieci połączeń. W sumie pozwala to na graficzne zilustrowanie proponowanego sposobu przetwarzania. Innym sposobem prezentacji algorytmu jest stosowanie tzw. pseudokodu tj. sztucznego języka, który zawiera struktury zbliżone do struktur dostępnych w językach algorytmicznych. Najczęściej pseudokod jest tworzony, w przypadku polskiego czytelnika, w języku polskim.

2

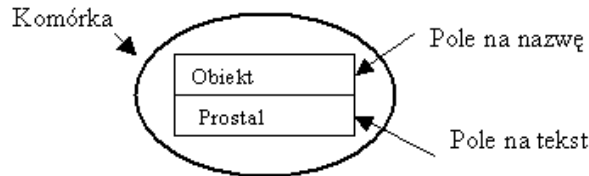
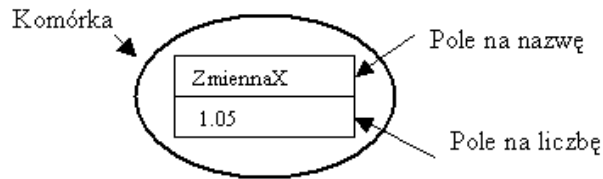
**Zmienne, typy danych,
operatory,
instrukcja warunkowa,
instrukcja cyklu**

W rozdziale przedstawimy za pomocą przykładów podstawowe elementy składowe algorytmicznych języków programowania. Kolejno omawiane przykłady zilustrują w jaki sposób budowane są i z jakich składników tworzone algorytmy konkretnych programów.

Na początek wyjaśnimy w jaki sposób w programowaniu funkcjonują składniki opisujące obiekty, które są przedmiotem przetwarzania. Pojęciem pierwotnym jest w tym przypadku pojęcie komórki (rysunek 2.1). Komórka stanowi zapis w pamięci komputera, który posiada dwa obszary: 1) obszar przeznaczony na przechowywanie nazwy komórki, będącej jej identyfikatorem, 2) obszar przeznaczony na przechowywanie zawartości komórki – może to być na przykład zawartość liczbowa jak na rysunku 2.1. Zawartością komórki może być także zawartość tekstowa (rysunek 2.2).

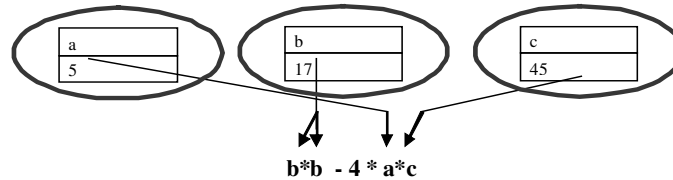


Rysunek 2.1. Ilustracja pojęcia komórki



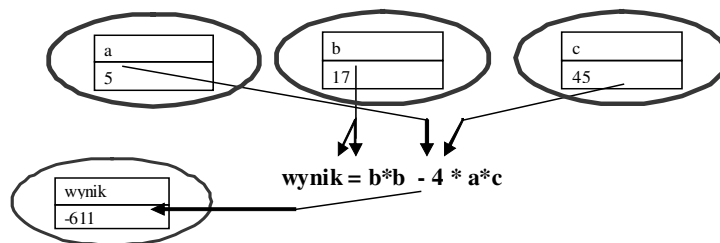
Rysunek 2.2. Przykłady komórek; u góry: przeznaczona do przechowywania liczby, na dole: przeznaczona do przechowywania opisu tekstowego

Zwykle komórki używane w programach identyfikowane są po ich nazwach przez to nazwy nadawane są w ten sposób, że nawiązują do nazw używanych w potocznym opisie modelowanej sytuacji. Odwołując się do komórki po nazwie np. wstawiając nazwę komórki do wyrażenia arytmetycznego (rysunek 2.3) odwołujemy się do zawartości tej komórki w momencie obliczania wartości tego wyrażenia.



Rysunek 2.3. Przykład wyrażenia arytmetycznego, w którym wykorzystano komórki **a**, **b**, **c**

Zatem jeżeli obliczamy wyrażenie ($b*b - 2*a*c$) - oznacza to kolejne odwołanie się do komórek **b**, **a** i **c** - pobranie przechowywanych przez nie wartości liczbowych i policzenie wartości wyrażenia. Jeżeli natomiast mamy sytuację, gdzie komórce **wynik** przypisywana jest określona wartość liczbową to oznacza to, że wartość liczbową jest wprowadzana do komórki o nazwie **wynik**. Czyli w zależności od roli komórki w danym wyrażeniu jest pobierana bądź wprowadzana do niej określona wartość liczbową. Zwykle w programowaniu posługujemy się dużą ilością komórek, ich nazwy tak jak wspomniano, nawiązują do nazw używanych potocznie w ich opisie. Na rysunku 2.4 przedstawiono przykłady komórek.



Rysunek 2.4. Przykład wyrażenia arytmetycznego, w którym wykorzystano komórki **a**, **b**, **c** i komórkę **wynik**

Pojęcie komórki dobrze ilustruje sens merytoryczny obiektu, który służy do składowania wielkości przetwarzanych. Generalnie, wielkości te

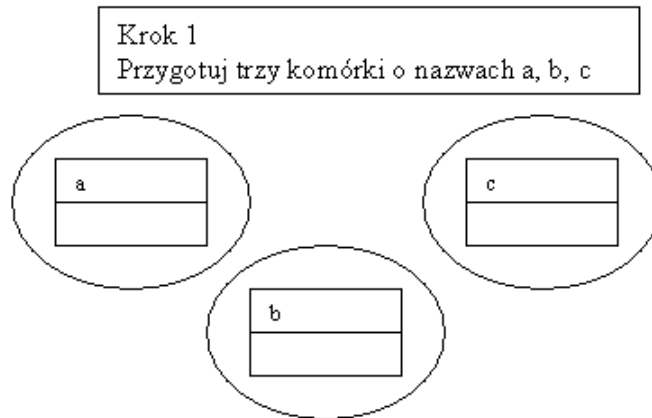
w językach programowania nazywane są zmiennymi. Poniżej przedstawimy prosty przykład, który zilustruje w jaki sposób realizowane jest przetwarzanie przez komputer i w jaki sposób możemy budować algorytmy. Opisywane zadanie to problem dodania do siebie dwóch liczb. Poniżej omówiono poszczególne etapy procesu przetwarzania.

Przykład 2.1

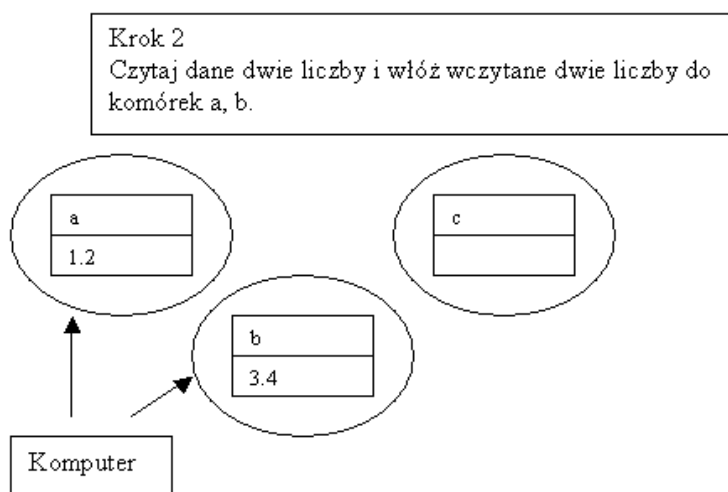
Rozwiązujący problem:

- budujemy algorytm programu, który ma dodawać do siebie dowolne dwie, wczytane przez komputer liczby.
- ciąg działań – algorytm:
 - krok 1, przygotowanie zmiennych (komórek) o nazwach a, b, c; a i b przeznaczone na dodawane liczby, c przeznaczona na przechowywanie wyniku,
 - krok 2, czytaj dane dwie liczby i zapisz je do zmiennych (komórek) a i b,
 - krok 3, wykonaj dodawanie liczb zawartych w zmiennych (komórkach) a i b, wynik zapisz w zmiennej c,
 - krok 4, wyświetl wynik zawarty w zmiennej (komórce) c.

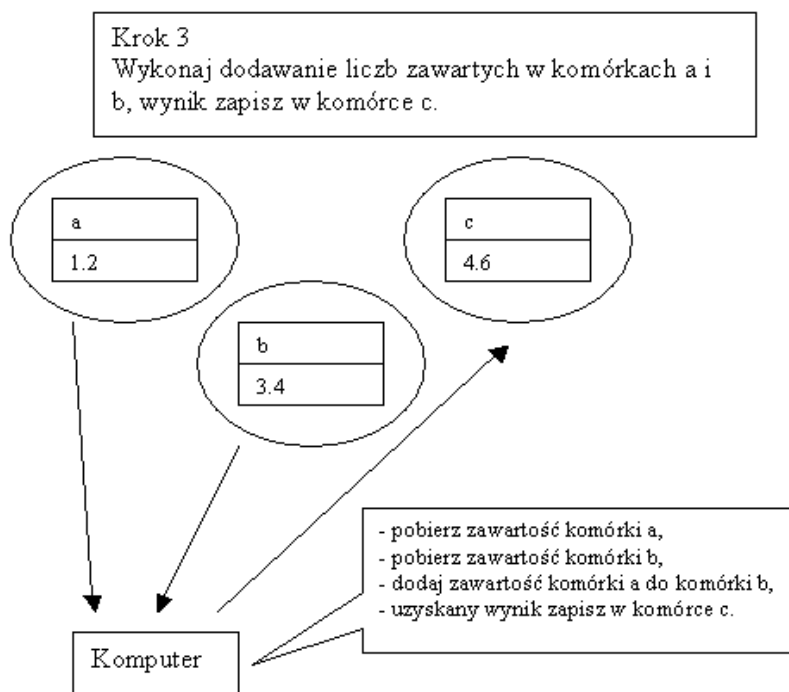
Kolejne kroki zostały przedstawione graficznie na rysunkach 2.5-2.8.



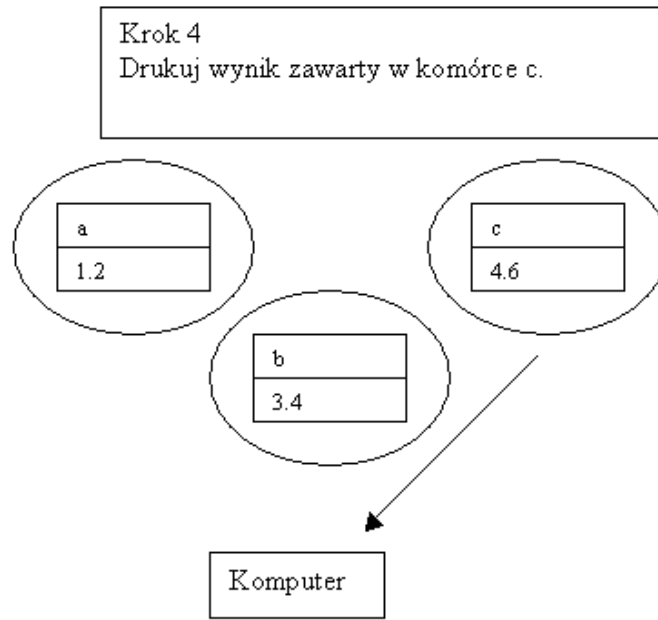
Rysunek 2.5. Ilustracja kroku 1 proponowanego algorytmu



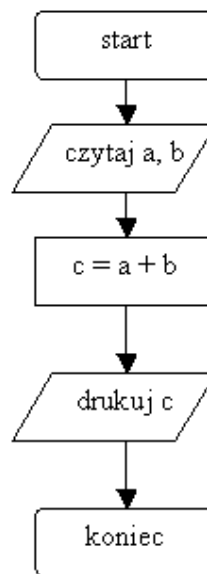
Rysunek 2.6. Ilustracja kroku 2 proponowanego algorytmu



Rysunek 2.7. Ilustracja kroku 3 proponowanego algorytmu



Rysunek 2.8. Ilustracja kroku 4 proponowanego algorytmu



Rysunek 2.9. Algorytm zaprezentowany opisowo na rysunkach 2.5-2.8 w postaci schematu blokowego

Na rysunku 2.9 pokazano algorytm rozwiązania przykładu 2.1 w formie schematu blokowego. Zaproponowany schemat blokowy jest wiernie przetworzonym algorytmem przedstawionym wcześniej w formie poszczególnych kroków. Poniżej na rysunku 2.10 ten sam problem zilustrowano w formie algorytmu zapisanego za pomocą pseudokodu. Na rysunku 2.11. widzimy omawiane zadanie zrealizowane jako aplikacja konsolowa w języku MS Visual Basic.

- określenie rezerwacji komórek a, b, c	}	- deklaracje
- wczytanie a i b	}	- część wykonawcza
- obliczenie c = a + b		
- wydrukowanie c		
- koniec pracy		

Rysunek 2.10. Algorytm z przykładu 2.1. w formie pseudokodu

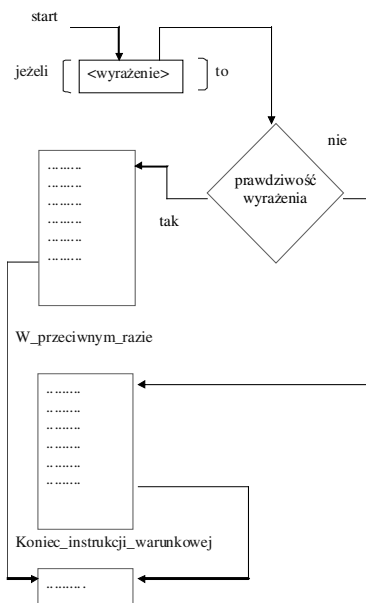
```
Sub Main()  
Dim a, b, c As Single  
  
Console.WriteLine("-> podaj 2 liczby")  
a = CSng(Console.ReadLine())  
b = CSng(Console.ReadLine())  
  
c = a + b  
  
Console.WriteLine("-> wynik :")  
Console.WriteLine(c)  
  
End Sub
```

Rysunek 2.11. Program napisany w języku MS Visual Basic w oparciu o algorytm przedstawiony na rysunku 2.10

W językach algorytmicznych programowania bardzo przydatną i często stosowaną konstrukcją jest instrukcja warunkowa. Na rysunku 2.12 pokazano schemat ogólny instrukcji warunkowej. Instrukcja składa się z linii-rozkazu zaczynającej się od **jeżeli**, w której znajduje się również warunek, którego prawdziwość jest sprawdzana. Po prawej stronie rysunku pokazano za pomocą strzałek jak odbywa się przetwarzanie w przypadku spełnienia warunku a jak w przypadku niespełnienia. W strukturze instrukcji występują dwa ciągi instrukcji: a) wykonywany w przypadku spełnienia warunku, b) w przypadku jego niespełnienia.

Przykładem dwukrotnego wykorzystania instrukcji warunkowej jest program wykonany w języku MS Visual Basic (rysunek 2.13), którego ce-

lem jest policzenie pierwiastków trójmianu kwadratowego. W programie odbywa się analiza wyróżnika trójmianu kwadratowego i w zależności od przypadku podejmowana jest decyzja odnośnie właściwego toku obliczeń.



Rysunek 2.12. Schemat struktury i działania instrukcji warunkowej

Kolejną często stosowaną w językach algorytmicznych konstrukcją jest instrukcja cyklu. Powszechnie jest ona nazywana pętlą. Jej schemat strukturalny powiązany ze schematem funkcjonowania przedstawiono na rysunku 2.14. Na schemacie założono dla uproszczenia, że licznik, który steruje krotnością wykonania zmienia się od **1** do **100** z krokiem **1**. Ogólnie zarówno wartość początkowa licznika jak i jej wartość końcowa, oraz krok mogą przyjmować dowolne wartości. Na rysunku 2.15 pokazano przykład programu napisanego w języku Visual Basic, który wykorzystuje instrukcję cyklu do zliczania sumy liczb od 1 do 100.

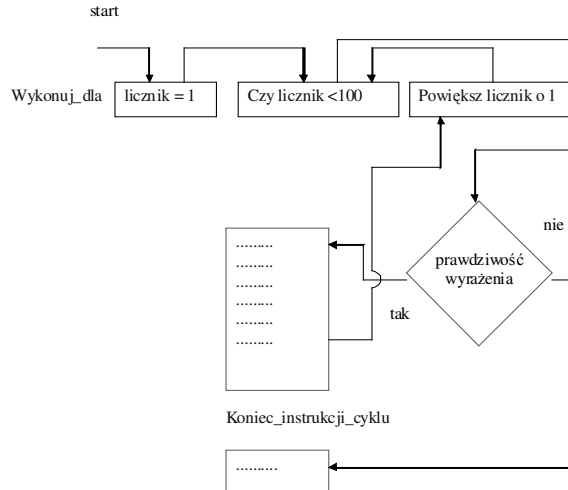
W programowaniu bardzo często używa się obiektów, które posiadają jedną wspólną nazwę powiązaną indeksem. Tych obiektów jest oczywiście tyle ile wynosi maksymalna wartość indeksu. Może to być np. obiekt: **tablica (100)** – zapis ten oznacza, że obiekt tablica posiada 100 elementów. Do konkretnego elementu możemy się odwoływać podając nazwę **tablica** i konkretną, odpowiednią wartość indeksu np. **tablica (53)**.

Do wskazania indeksu tablicy możemy wykorzystać również zmienną np. **i = 3** i dalej odwołujemy się do **tablica (i)**.

Na rysunku 2.16 pokazano przykład programu ilustrującego proces tablicowania funkcji, gdzie poszczególne obliczone wartości funkcji są składowane w odpowiednich elementach tablicy. Na rysunku 2.17 przedstawiono rozwiniętą wersję przykładu z rysunku 2.16. Dodano moduł wyszukujący najmniejsze i największe wartości elementów tablicy. Wykorzystano w tym celu zmienne **imax** i **imin**, które są porównywane z kolejnymi elementami tablicy. W przypadku, gdy porównywany element tablicy jest lepszy od dotychczasowego **imax** lub **imin** odpowiednio **imin** lub **imax** nadawana jest nowa jego wartość.

```
Sub Main()  
    Dim a, b, c As New Single  
    Dim delta, x0, x1, x2 As New Single  
    Console.WriteLine("program liczy pierwiastki  
trójmianu")  
    Console.WriteLine("podaj a:")  
    a = CStr(Console.ReadLine())  
    Console.WriteLine("podaj b:")  
    b = CStr(Console.ReadLine())  
    Console.WriteLine("podaj c:")  
    c = CStr(Console.ReadLine())  
    delta = b * b - 4 * a * c  
    If (delta < 0.0) Then  
        Console.WriteLine("nie ma pierwiastków")  
    Else  
        If (delta = 0.0) Then  
            x0 = -b / (2 * a)  
            Console.WriteLine(Format(x0, _  
                "1 pierwiastek = 00000.00"))  
        Else  
            x1 = (-b - Math.Sqrt(delta)) / (2 * a)  
            x2 = (-b + Math.Sqrt(delta)) / (2 * a)  
            Console.WriteLine(Format(x1, _  
                "1 pierwiastek = 00000.00"))  
            Console.WriteLine(Format(x2, _  
                "2 pierwiastek = 00000.00"))  
        End If  
    End If  
End Sub
```

Rysunek 2.13. Program w języku MS Visual Basic obliczający pierwiastki trójmianu kwadratowego



Rysunek 2.14. Schemat struktury i działania instrukcji cyklu

```
Sub Main()  
  Dim i As New Integer  
  Dim suma As New Single  
  suma = 0.0  
  For i = 1 To 100 Step 1  
    suma = suma + i  
  Next  
  Console.WriteLine(Format(suma, _  
    " wynik dodawania = 000.0"))  
End Sub
```

Rysunek 2.15. Program napisany w języku Visual Basic zliczający liczby od 1 do 100

```
Sub Main()  
  Dim i As New Integer  
  Dim tablica(100) As Single  
  Dim i_max, i_min As New Single  
  
  For i = 1 To 100 Step 1  
    tablica(i) = i * i - 2 * i  
  Next i  
  
  i_max = tablica(1)  
  i_min = tablica(1)  
  For i = 2 To 100 Step 1  
    If (tablica(i) > i_max) Then
```

ROZDZIAŁ 2

```
        i_max = tablica(i)
    End If
    If (tablica(i) < i_min) Then
        i_min = tablica(i)
    End If
Next

For i = 1 To 100 Step 1
    Console.WriteLine(Format(i, _
        "argument = 000.0"))
    Console.WriteLine(Format(tablica(i), _
        "wartość funkcji = 000.0"))
Next

    Console.WriteLine(Format(i_max, _
        "wartość maksymalna = 00000.0"))
    Console.WriteLine(Format(i_min, _
        "wartość minimalna = 00000.0"))
End Sub
```

Rysunek 2.16. Instrukcja cyklu w powiązaniu obiektem- tablicą

Niniejszy rozdział miał charakter wprowadzający, omówiono w nim podstawowe konstrukcje języków algorytmicznych programowania. W rozdziale następnym 3 przedstawiono podstawowe zagadnienia związane z budową algorytmów w mechanice i projektowaniu maszyn.

3

**Podstawowe algorytmy
obliczeniowe.**

**Przykłady z mechaniki
i PKM**

3.1. Wprowadzenie

W rozdziale przedstawimy kilka przykładowych zadań projektowania algorytmów, które pokazują w jaki sposób możemy uchwycić realne procesy, realne zjawiska i zamodelować jako problemy typowe dla programowania algorytmicznego.

Zanim przejdziemy do ww. przykładów zwrócimy uwagę na zagadnienia przetwarzania iteracyjnego w przetwarzaniu algorytmicznym. Zrobimy to na przykładzie całkowania numerycznego.

Jednym z zagadnień bardzo często spotykanych w projektowaniu maszyn, opartych na przetwarzaniu iteracyjnym, jest obliczanie wartości całek oznaczonych. Zwykle, stosuje się podejście numeryczne, które zastępuje problem pierwotny problemem zastępczym. Obliczanie całki oznaczonej sprowadza się do obliczenia pola pod funkcją podcałkową w określonych granicach – przedziale całkowania. Budując zadanie zastępcze zastępujemy obszar pod funkcją podcałkową zbiorem obszarów o jednakowej szerokości i o różnych wysokościach odpowiadających wartościom funkcji podcałkowej we właściwych punktach. Zadanie całkowania w tym przypadku stanowi zadanie zliczania sumy pól znajdujących się pod krzywą. Poszczególne pola to trapezy. Zadanie polega na obliczaniu pól poszczególnych trapezów i ich sumowaniu.

Na rysunku 3.1 przedstawiono program napisany w języku MS Visual Basic realizujący proces obliczania całki oznaczonej w sposób iteracyjny. Zadanie wykonano dla funkcji:

$$y(x) = x*x$$

Oznaczenia poszczególnych zmiennych:

- x_p – wartość początkowa przedziału całkowania,
- x_k - wartość końcowa przedziału całkowania,
- dx – krok całkowania, odpowiada wysokości pola obliczanego trapezu,
- x - bieżąca wartość zmiennej zmieniająca się wraz z iteracjami instrukcji cyklu,

ROZDZIAŁ 3

- x_1 – wartość x dla pierwszej podstawy trapezu, którego pole jest obliczane,
- x_2 – wartość x dla drugiej podstawy trapezu, którego pole jest obliczane,
- a, b – długości podstaw, pierwszej i drugiej, danego trapezu,
- `pole_trapezu` – zmienna przeznaczona do składowania obliczanego w danej chwili pola trapezu,
- `pole` – zmienna przeznaczona do bieżącego składowania obliczonej aktualnie wartości sumy pól poszczególnych trapezów.

```
Sub Main()  
    Dim x, x1, x2, a, b, dx, xp, xk, _  
        pole_trapezu, pole As Single  
    xp = 10  
    xk = 100  
    dx = 0.01  
    pole = 0.0  
    For x = xp To xk Step dx  
        x1 = x  
        x2 = x + dx  
        a = x1 * x1  
        b = x2 * x2  
        pole_trapezu = (a + b) * dx / 2  
        pole = pole + pole_trapezu  
    Next  
    Console.WriteLine("-> pole:")  
    Console.WriteLine(Format(pole, "000.0"))  
End Sub
```

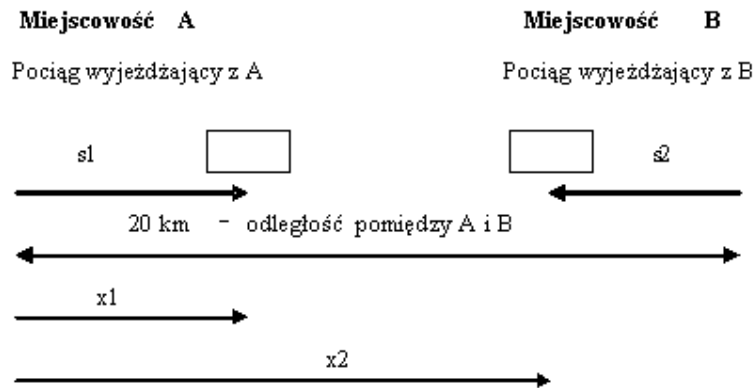
Rysunek 3.1. Program napisany w języku MS Visual Basic obliczający w sposób iteracyjny (metodą trapezów) całkę oznaczoną.

Metody całkowania numerycznego wraz z przykładami przedstawiono w rozdziale 8.

3.2. Przykład zamodelowania zadania z mechaniki

Jednym z często analizowanych zagadnień mechaniki są zadania z kinematyki. W ramach rozwiązywania tej klasy zadań można dokładnie określać położenia, prędkości i przyspieszenia przemieszczających się obiektów. Zastosowanie komputera pozwala na wykorzystanie podejścia symulacyjnego w tych zagadnieniach. Można określać równoległe ruch większej liczby przemieszczających się obiektów. W ten sposób symulujemy ich realne zachowanie. Można także badać relacje zachodzące pomiędzy symulowanymi obiektami. Zagadnienie to pokażemy na przykładzie przemieszczających się pociągów.

Na rysunku 3.2. pokazano koncepcję zadania.



Rysunek 3.2. Ilustracja graficzna zadania opisującego ruch pociągów

Na rysunku 3.3 przedstawiono program wyliczający kolejne położenia pociągu wyjeżdżającego z miejscowości A do miejscowości B. Pociąg porusza się ze stałą prędkością v_1 (np. 80 km/godz.), droga pociągu jest oznaczona przez s_1 (w km). Przez t oznaczono bieżący czas, który jest zmieniany z krokiem dt (w godzinach). W podobny sposób można wyliczać położenia pociągu przy bardziej realistycznym przebiegu zmienności jego prędkości. Można również zadbać o precyzyjne określenie związków pomiędzy czasem realnym i czasem symulowanym. W algorytmie badamy pierwsze 30 iteracji.

```
Sub Main()  
    Dim s1, t, v1, dt, i As Single  
    v1 = 80  
    dt = 0.01  
    t = 0  
    For i = 1 To 30 Step 1  
        t = t + dt  
        s1 = v1 * t  
        Console.WriteLine("-> czas t:")  
        Console.WriteLine(Format(t, "000.0000"))  
        Console.WriteLine("-> droga s1:")  
        Console.WriteLine(Format(s1, "000.0"))  
    Next  
End Sub
```

Rysunek 3.3. Program w MS Visul Basic'u obliczający kolejne położenia przemieszczającego się pociągu (wyjeżdżającego z miejscowości A i zmierzającego do miejscowości B)

Na rysunku 3.4. przedstawiono rozszerzoną wersję poprzedniego programu. Pojawia się jeszcze pociąg przemieszczający się miejscowości **B** do **A**. Pociąg ten rusza po czasie **t2p** od momentu ruszenia pierwszego pociągu. Kolejno wyliczane są położenia dla każdego z obu pociągów. Pociąg drugi porusza się ze stałą prędkością **v2**, jego drogę oznaczono przez **s2**. Dodanie kolejnego pociągu to dodanie nowego procesu. Możemy oczywiście wprowadzić wiele takich procesów, które jak w tym przykładzie pozostają w stosunku do siebie niezależne (np. kolejne pociągi poruszające się po różnych torach). Możemy także zamodelować związki pomiędzy procesami wynikające np. ze spotkań pociągów na dworcach czy też korzystania przez różne pociągi z tych samych torów.

Na rysunku 3.5. pokazano wersję tego samego programu gdzie odbywa się sprawdzanie zdarzenia polegającego na spotkaniu obu pociągów. Przyjęto, że strefa spotkania obu pociągów obejmuje określony odcinek drogi mniejszy od **0.5 km**. Przez **x1** i **x2** oznaczono położenia obu pociągów w tym samym układzie współrzędnych. Zdarzenie w postaci spotkania pociągów zależy od prędkości obu pociągów, kroku przyrostu czasu oraz wielkości strefy spotkania. Może się zdarzyć, że zdarzenie nie zostanie wykryte w programie mimo, że pociągi przemieszczają się koło siebie.

```

Sub Main()
    Dim s1, s2, v2, t2p, t, v1, dt, i As Single
    v1 = 80
    v2 = 100
    t2p = 0.2
    dt = 0.01
    t = 0
    For i = 1 To 30 Step 1
        t = t + dt
        s1 = v1 * t
        Console.WriteLine("-> czas t:")
        Console.WriteLine(Format(t, "000.0000"))
        Console.WriteLine("-> droga s1:")
        Console.WriteLine(Format(s1, "000.0"))
        If (t - t2p > 0.0) Then
            s2 = v2 * (t - t2p)
            Console.WriteLine("-> czas t-t2p:")
            Console.WriteLine(Format(t - t2p, _
                "000.0000"))
            Console.WriteLine("-> droga s2:")
            Console.WriteLine(Format(s2, _
                "000.0"))
        End If
    Next
End Sub

```

Rysunek 3.4. Program w MS Visual Basic'u obliczający kolejne położenia obu pociągów

Na rysunkach 3.6 i 3.7 przedstawiono wersję programu z rysunku 3.5. napisaną jako aplikacja z okienkowym interfejsem graficznym w języku MS Visual Basic. Na rysunku 3.6 pokazano interfejs graficzny programu z zaznaczonymi nazwami obiektów graficznych. W dolnej części okna umieszczono przyciski uruchamiające/zatrzymujące i inicjujące animację przemieszczających się pociągów. Poza tym wyświetlana jest bieżąca informacja o położeniach pociągów i o aktualnych zdarzeniach. Część środkową okna zajmuje obszar zajęty przez obiekty animowane. Górna część służy do wprowadzania danych.

Procedura `btnSTART_Click` przeznaczona jest do uruchamiania/zatrzymywania procesu animacji. Procedura `TEST_DANYCH()` służy do merytorycznego sprawdzania poprawności danych – na wydruku pozostawiono jedynie sprawdzanie zmiennej `strefa` określającej odcinek spotkania pociągów (Przyjęto, że maksymalna wartość strefy wynosi 4 km). Procedura `Timer1_Tick` steruje bezpośrednio procesem animacji. Natomiast procedury `btnPOCZATEK_Click` i `btnSTOP_Click` odpowiednio: pierwsza inicjuje animację, druga kończy pracę programu.

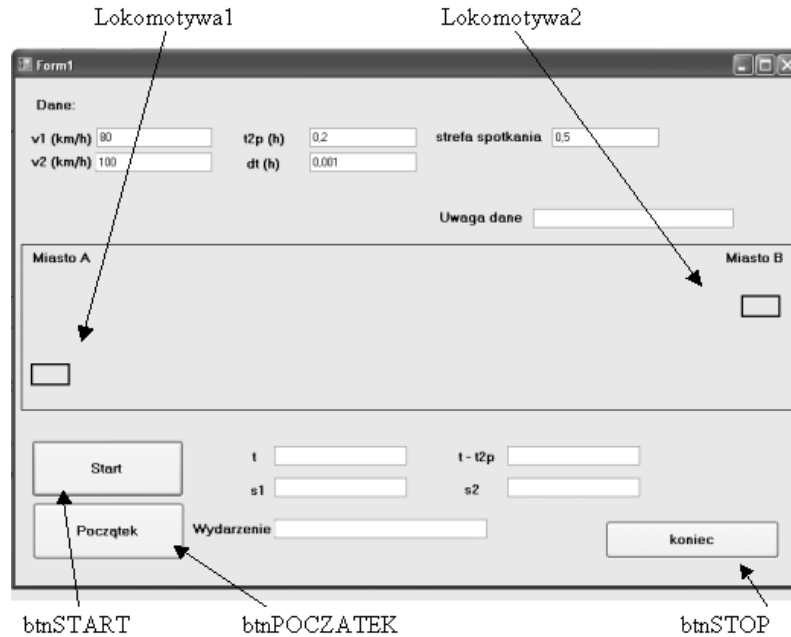
ROZDZIAŁ 3

```
Sub Main()  
    Dim s1, s2, v2, t2p, t, v1, dt, i, x1, x2 As _  
                                                Single  
  
    v1 = 80  
    v2 = 100  
    t2p = 0.2  
    dt = 0.01  
    t = 0  
    For i = 1 To 50 Step 1  
        t = t + dt  
        s1 = v1 * t  
        Console.WriteLine("-> czas t:")  
        Console.WriteLine(Format(t, "000.0000"))  
        Console.WriteLine("-> droga s1:")  
        Console.WriteLine(Format(s1, "000.0"))  
        If (t - t2p > 0.0) Then  
            s2 = v2 * (t - t2p)  
            Console.WriteLine("-> czas t-t2p:")  
            Console.WriteLine(Format(t - t2p, _  
                                    "000.0000"))  
            Console.WriteLine("-> droga s2:")  
            Console.WriteLine(Format(s2, "000.0"))  
            x1 = s1  
            x2 = 20 - s2  
            If (Math.Abs(x1 - x2) < 0.5) Then  
                Console.WriteLine("-> SPOTKANIE")  
            End If  
        End If  
    Next  
End Sub
```

Rysunek 3.5. Program w MS Visul Basic'u obliczający kolejne położenia obu pociągów i „wykrywający” moment ich spotkania

Przykładowe zadanie dotyczy problematyki przemieszczających się obiektów. Podobne modele mogą powstać w przypadku analizy ergonomicznej określonych konstrukcji, obsługi, monitoringu pewnych klas maszyn. Poziom komplikacji modeli może oczywiście być znacznie podniesiony w zależności od przeznaczenia budowanego oprogramowania. Wykorzystywane są wówczas bardziej zaawansowane metody, modele i podejścia mechaniki. Jeżeli oprogramowanie budowane jest w celach poznawczych przeważnie próbujemy uchwycić rzeczywistość w jak najdoskonalszy sposób. Jeżeli jednak musimy odwoływać się do realnej rzeczywistości – mamy wyniki konkretnych, realnych badań, nasz model komputerowy ma być wykorzystywany w procesach monitorowania lub sterowania to na ogół dużą rolę odgrywają jakość wprowadzanych danych (czy te dane są poprawne i czy są zawsze dostępne),

i realizowalny komputerowo czas przetwarzania (czy jest on akceptowalny).



Rysunek 3.6. Opis wybranych obiektów wykorzystanych w okienkowej wersji programu symulującego ruch pociągów

Dobrym rozwinięciem zaprezentowanego programu mogą być aplikacje symulujące inne realne zjawiska np. proces sterowania pracą grupy urządzeń, których warunki uruchomienia, pracy zależą od czynników zewnętrznych (np. sekcje silników/pomp, różne urządzenia stosowane w samochodach, przemysłowa aparatura pomiarowo-sterująca, itp.). Zwykle rozwiązanie zadania tej klasy zaczyna być budowane od określenia zbioru koniecznych do zamodelowania stanów układu oraz zasad przechodzenia od stanu do stanu. Stosowane formalizmy mogą przyjmować bardzo różną, zależną od przypadku, postać.

```
Public Class POCIAGI
    Dim t As Single = 0.0
    Private Sub btnSTART_Click(ByVal sender As _
        System.Object, ByVal e As System.EventArgs) _
        Handles btnSTART.Click
        Dim tak As Integer = 0
        Call TEST_DANYCH(tak)
        If tak = 0 Then
```

```
        If btnSTART.Text = "Start" Then
            btnSTART.Text = "Stop"
            Timer1.Enabled = True
        Else
            btnSTART.Text = "Start"
            Timer1.Enabled = False
        End If
    End If
End Sub

Private Sub TEST_DANYCH(ByRef tak As Integer)
    Dim v2, t2p, v1, dt, strefa As Single
    tak = 0
    txtinfo.Text = ""
    v1 = CSng(txtV1.Text)
    v2 = CSng(txtV2.Text)
    t2p = CSng(txtt2p.Text)
    dt = CSng(txttdt.Text)
    strefa = CSng(txtStrefa.Text)
    If strefa < 0.0 Or strefa > 4 Then
        txtinfo.Text = "strefa poza zakresem: " & _
            CStr(strefa) & " , (0. ; 4.)"
        tak = 1
    End If
End Sub

Private Sub Timer1_Tick(ByVal sender As _
    System.Object, _
    ByVal e As System.EventArgs) Handles _
    Timer1.Tick
    Dim s1, s2, v2, t2p, v1, dt, x1, x2, strefa _
        As Single
    v1 = CSng(txtV1.Text)
    v2 = CSng(txtV2.Text)
    t2p = CSng(txtt2p.Text)
    dt = CSng(txttdt.Text)
    strefa = CSng(txtStrefa.Text)
    t = t + dt
    s1 = v1 * t
    txtT.Text = CStr(t)
    txtS1.Text = CStr(s1)
    If s1 > 20 Then s1 = 20
    lokomotywa1.Left = 17 + 37 * s1
    If (t - t2p > 0.0) Then
        s2 = v2 * (t - t2p)
        txtT2p.Text = CStr(t - t2p)
        txtS2.Text = CStr(s2)
        If s2 > 20 Then s2 = 20
    End If
End Sub
```

```
        lokomotywa2.Left = 761 - 37 * s2
        x1 = s1
        x2 = 20 - s2
        txtWYD.Text = ""
        If (Math.Abs(x1 - x2) < strefa) Then
            txtWYD.Text = "Spotkanie pociągów"
        End If
    End If
End Sub

Private Sub btnPOCZATEK_Click(ByVal sender As _
    System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnPOCZATEK.Click
    t = 0.0
    lokomotywa1.Left = 17
    lokomotywa2.Left = 761
    txtT.Text = ""
    txtS1.Text = ""
    txtTT2p.Text = ""
    txtS2.Text = ""
    txtWYD.Text = ""
End Sub

Private Sub btnSTOP_Click(ByVal sender As _
    System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnSTOP.Click
End
End Sub
```

Rysunek 3.7. Okienkowa wersja programu symulującego ruch pociągów

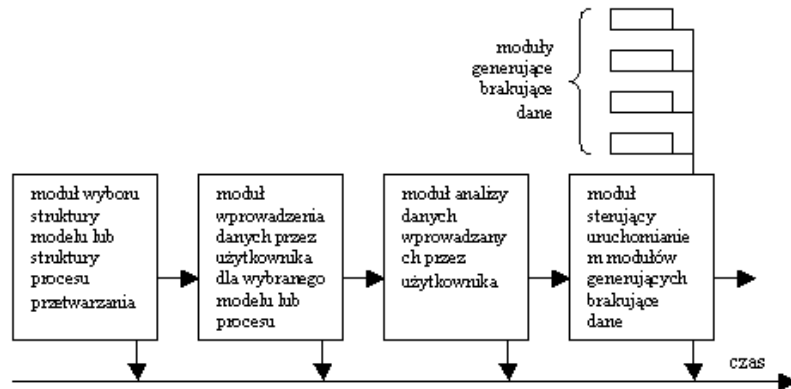
3.3. Przykład zamodelowania zadania z Podstaw Konstrukcji Maszyn

W poprzednim rozdziale przedstawiliśmy przykład algorytmu dla problemu opartego na mechanice, który w odniesieniu do zagadnień inżynierskich, projektowych może być określony jako analizy inżynierskie.

Osobną grupę zagadnień stanowią problemy projektowe, które również mogą być wspomagane określonymi narzędziami komputerowymi.

Obecnie do wspomagania procesów projektowych wykorzystujemy oferowane komercyjnie systemy CAD/CAE (Computer Aided Design/Computer Aided Engineering). Systemy te zawierają bardzo dużo modułów wspomagających proces rozwiązywania różnych klas problemów. Systemy CAD pozwalają tworzyć modele 3D projektowanych konstrukcji, systemy CAE umożliwiają wykonanie niezbędnych analiz. Zwykle oprogramowanie to charakteryzuje się dużym uniwersalizmem. Jednak w konfrontacji z procesami inżynierskimi realizowanymi w dzisiejszym przemyśle często okazuje się, że potrzebne są inne, nowe funkcjonalności, których w tych systemach nie ma np. może chodzić o powiązanie procesów wspomaganych za pomocą komercyjnych narzędzi CAD/CAE z wybitnie firmowym know-how. Skutkiem takiej sytuacji jest zwykle hybrydowa struktura programistyczna zawierająca zarówno moduły komercyjne jak i oprogramowanie własne, firmowe. To oprogramowanie własne może być budowane samodzielnie przez firmy, może być również zlecane do wykonania podmiotom zewnętrznym.

Oprogramowanie oparte na firmowym know-how najczęściej wspomaga proces realizacji ściśle określonych aktywności projektowych. W strukturze pojedynczej aktywności projektowej przeważnie można wyodrębnić jej fragment związany ze strukturą modelu oraz postacią procesu rozwiązywania zadania projektowego (rysunek 3.8). Można w tym przypadku spotkać dwa rozwiązania:



Rysunek 3.8. Struktura programu komputerowego – etap wyboru struktury i wprowadzania parametrów

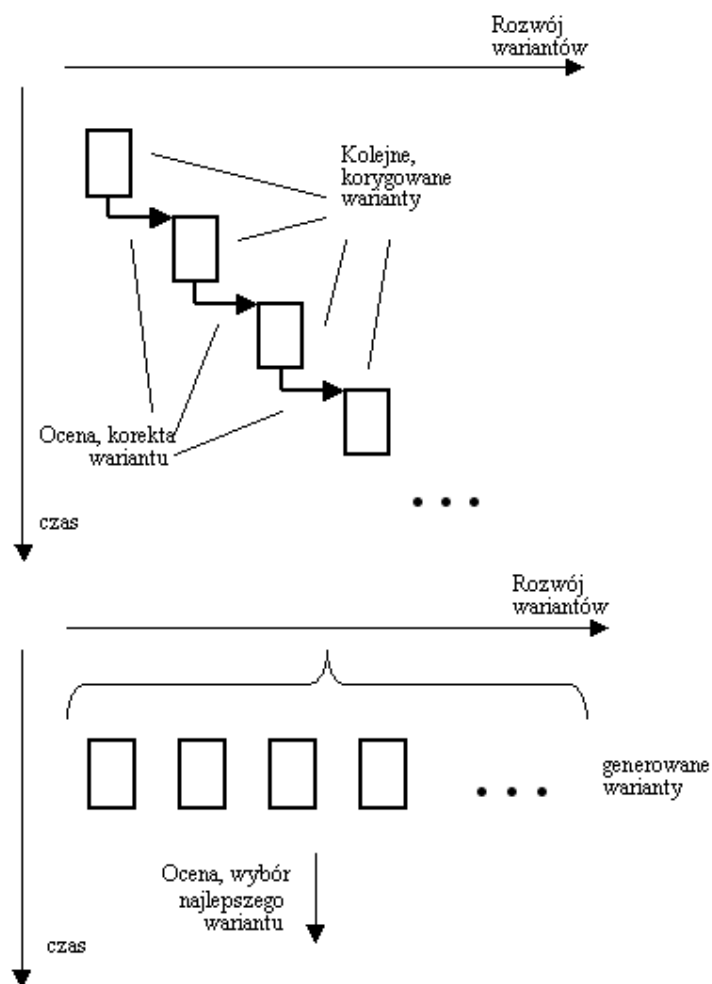
1. zastosowanie narzędzia pozwalającego modelować modele produktu czy też procesy projektowe za pomocą zbioru elementów podstawowych (prymitywów) - jest to koncepcja oparta na budowie specjalizowanego edytora,
2. przygotowanie zbioru zdefiniowanych wstępnie wariantów modeli produktów czy też procesów projektowych.

W niniejszym opracowaniu nie będziemy zajmować się przypadkiem 1) – generalnie jest on dosyć złożony. W przypadku 2) użytkownik programu wybiera wariant strukturalny modelu czy też procesu i dalej wprowadzając odpowiednie dane tworzy kompletny opis rozwiązywanego zadania.

Wspomagany proces projektowy może składać się z kilku takich etapów jak powyżej i w każdym z nich może występować fragment związany z doбором jego postaci strukturalnej.

Po etapie specyfikacji struktury następuje etap wprowadzania danych typowo parametrycznych. Może być on realizowany ręcznie przez projektującego. Może też zawierać elementy funkcjonujące automatycznie (rysunek 3.8). Proces automatycznego generowania wybranych parametrów opiera się na zamodelowanej w systemie wiedzy oraz na danych już wprowadzonych do systemu przez projektującego np. ustawień domyślnych, danych wprowadzanych aktualnie, itd.

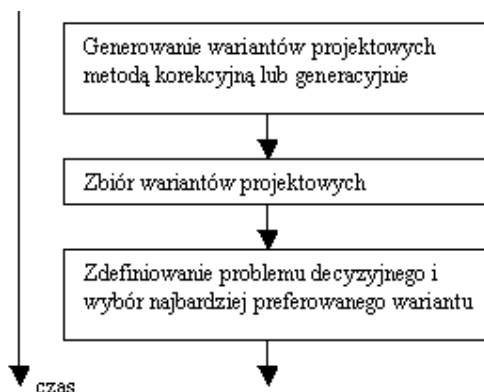
W procesie projektowania na etapie dopracowywania konstrukcji, najczęściej stosowana jest jedna z dwóch strategii (rysunek 3.9):



Rysunek 3.9. Korekcyjne (u góry) i generacyjne (u dołu) tworzenie kolejnych wariantów projektowych

1. realizowany jest jeden wariant projektowy, który jest następnie korygowany przez projektującego, w rezultacie powstaje nowy wariant i sytuacja jest dalej powtarzana – może powstać kolejno więcej, stopniowo ulepszanych, wariantów,
2. projektujący jednorazowo generuje określony zbiór wariantów projektowych, które następnie ocenia, porównuje i dokonuje selekcji najbardziej preferowanego rozwiązania.

W przypadku powstania więcej niż jednego wariantu projektowego zawsze konieczne jest porównanie wariantów. Stosowane są w tym celu metody wspomaganie procesów decyzyjnych, których zadaniem jest uszeregować wygenerowane rozwiązania zgodnie z preferencjami projektującego (rysunek 3.10).



Rysunek 3.10. Proces podejmowania decyzji projektowych

Powyższe zagadnienia począwszy od wyboru wariantu strukturalnego modelu produktu jak i procesu projektowego, sposobu generowania kolejnych wariantów projektowych, zestawiania ze sobą wariantów projektowych oraz wyboru rozwiązania ostatecznego przedstawimy poniżej na przykładach.

Zajmijmy się sytuacją gdzie oprogramowanie powstaje w firmie. Rozważmy konkretny przykład. Przykład dotyczy procesu doboru reduktora jedno-stopniowego (przekładnia walcowa o zębach skośnych) z katalogu producenta. Zakładamy, że dysponujemy bazą danych reduktorów oferowanych przez producenta, w której dostępne są podstawowe dane reduktorów oraz ich charakterystyki. Przyjmujemy, że możemy pobrać odpowiednie dane z bazy do naszego programu, który ma zapewnić możliwość selekcji odpowiedniego egzemplarza reduktora. Dane te zostaną wykorzystane w procesie selekcji i pozwolą wprowadzić właściwy model reduktora do dokumentacji 3D systemu CAD.

Na początku naszego zadania, zgodnie z sugestiami zamieszczonymi powyżej, musimy podjąć decyzje dotyczące struktury reduktora. Reduktor jest jednostopniowy. W związku z tym zakładamy, że możliwy jest jedynie wybór struktury układu łożyskowania. Zakładamy możliwość wyboru jednego spośród trzech wariantów:

1. łożyskowanie wału dwu-podporowego, za pomocą dwóch łożysk tocznych zwykłych,
2. łożyskowanie wału dwu-podporowego za pomocą dwóch łożysk kulkowych skośnych,
3. łożyskowanie wału dwu-podporowego za pomocą dwóch łożysk skośnych.

Na rysunku 3.11 przedstawiono fragment programu napisanego w języku MS Visual Basic pozwalający na wybór określonego rozwiązania w zakresie struktury łożyskowania.

Po etapie wyboru struktury łożyskowania dalszy wybór oczekiwanych parametrów reduktora jest realizowany przez projektującego (rysunek 3.12). Zakładamy, że na tym etapie, przystępując do doboru reduktora projektujący określa jego podstawowe dane:

P- przenoszona moc,

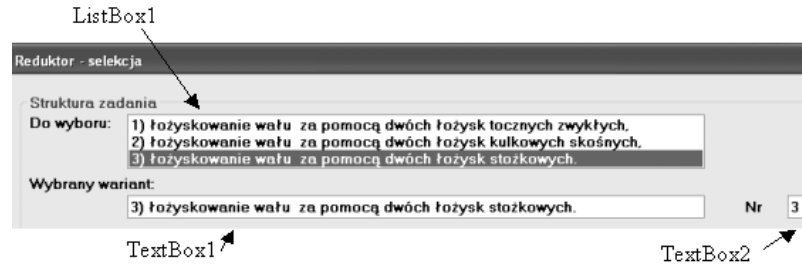
n – prędkość obrotową na wejściu,

i – przełożenie reduktora,

delta_i – odchyłkę przełożenia,

a, b, c – trzy wymiary gabarytowe w układzie x, y, z.

Na formularzu widoczne są dwa przyciski „**Sprawdź**” i „**Sprawdź/generuj**”. Przycisk „**Sprawdź**” powoduje sprawdzenie czy wprowadzone w poszczególnych polach dane wejściowe spełniają wymagania merytoryczne – wyniki sprawdzenia są zapisywane poniżej każdej wprowadzanej wielkości wejściowej. Przycisk „**Sprawdź/generuj**” powoduje realizację tej samej akcji jak powyżej – w przypadkach wymagających ingerencji dokonuje także korekty wprowadzonych wartości i skorygowane wartości liczbowe są wpisywane w oknach obszaru „**Parametry wprowadzone i generowane**”.



```
Public Class Form1
    Dim Numer_wariantu As Integer
    Private Sub ListBox1_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
ListBox1.SelectedIndexChanged
        TextBox1.Text = ListBox1.SelectedItem
        Numer_wariantu = ListBox1.SelectedIndex + 1
        TextBox2.Text = Numer_wariantu.ToString
    End Sub
    .....
End Class
```

Rysunek 3.11. Fragment programu pozwalający na określenie struktury łożyskowania

Zadanie generowania wariantów reduktora i doboru określonego wariantu w naszym przypadku realizujemy w ten sposób, że pobieramy z bazy danych dane dotyczące parametrów np. 100 reduktorów i następnie poszukujemy, wśród nich reduktora, który najlepiej spełnia nasze oczekiwania. W prezentacji przykładu na rysunkach 3.11. 3.11, 3.13 pomijamy etap pobierania danych o reduktorach z bazy danych. Generowanie wariantów może odbywać się również poprzez przeliczanie parametrów kolejnych wariantów dla danych z pewnego przedziału zmienności. Konstrukcję tę można zrealizować za pomocą instrukcji cyklu lub generatorów liczb losowych.

Przystępując do realizacji zadania doboru reduktora, spośród reduktorów, których dane pobrano z bazy, musimy określić tok postępowania związany z doбором. Możliwych jest co najmniej kilka rozwiązań. Poniżej przedstawiono trzy przykładowe propozycje:

1. podstawą do podjęcia decyzji jest warunek na przenoszoną moc – zakładamy, że wybierzemy reduktor o mocy wyższej od oczekiwanej, najbliższy tej wartości,
2. podstawą do podjęcia decyzji jest warunek na przenoszoną moc (tak jak powyżej) rozpatrywany w powiązaniu z warunkiem na przełożenie z uwzględnieniem jego odchyłki – oba warunki muszą być spełnione jednocześnie,

- zakładamy, że najważniejsze są warunki gabarytowe związane z zabudową reduktora – dopiero potem sprawdzane są warunki jak w punkcie 2).

Oczywiście możliwych jest więcej przypadków scenariuszy.

Zajmijmy się pierwszym przypadkiem. Zakładamy, że dane poszczególnych reduktorów z katalogu pobrano z bazy danych i są one dostępne w postaci tablic parametrów reduktora: moc: **P_katalog (100)**, prędkość obrotowa: **n_katalog (100)**, przełożenie: **i_katalog(100)**, wymiary gabarytowe w układzie **x, y, z**: **a_katalog(100)**, **b_katalog (100)**, **c_katalog(100)** w programie napisanym w języku MS Visual Basic. Przyjmujemy, że dane k-tego reduktora oznaczone są tą samą wartością indeksu we wszystkich tablicach.

Reduktor - selekcja

Struktura zadania

Do wyboru:

- 1) łożyskowanie wału za pomocą dwóch łożysk tocznych zwykłych.
- 2) łożyskowanie wału za pomocą dwóch łożysk kulkowych skośnych.
- 3) łożyskowanie wału za pomocą dwóch łożysk stożkowych.

Wybrany wariant:

3) łożyskowanie wału za pomocą dwóch łożysk stożkowych. Nr 3

Parametry wprowadzane

Moc P	20	Prędkość obrotowa n	2000	Wysokość c	brak dane
Przełożenie i	3	Długość a	poza zakr		
Błąd przełożenia delta_i	brak dane	Szerokość b	brak dane		

Parametry wprowadzane i generowane

Moc P	20	Prędkość obrotowa n	20	Wysokość c	36
Przełożenie i	20	Długość a	40		
Błąd przełożenia delta_i	44	Szerokość b	38		

Sprawdź

Sprawdź/generuj

Rysunek 3.12. Okno programu służącego do wyboru struktury zadania (fragment przedstawiono na rysunku 3.11), wprowadzania parametrów, sprawdzania merytorycznej poprawności parametrów i generowania zbioru poprawnych parametrów

Na rysunku 3.13 pokazano program pozwalający wybrać najlepsze rozwiązanie przy strategii postępowania nr 1). Strategie 2) i 3) wymagają obliczenia odległości pomiędzy wariantem oczekiwanym i każdym z wariantów możliwych do realizacji. Tak obliczone odległości mogą się

stać podstawą do uszeregowania branych pod uwagę wariantów. W przypadku strategii 3) dochodzi jeszcze warunek na odfiltrowanie wariantów nie spełniających wymagań geometrycznych.

```
Sub Main()  
    Dim P_katalog(100) As Single  
    Dim P, P_odleglosc As Single  
    Dim k_wybrane, k As Integer  
    ' Pobieranie, z bazy, danych katalogowych reduktorów:  
    ' P_katalog(100)(moc) oraz informacji odnośnie  
    ' żądanej mocy reduktora P  
    ' P_odleglosc - parametr używany w procesie selekcji  
    ' wariantu katalogowego najbliższego poszukiwanemu,  
    P_odleglosc = 1000.0  
    For k = 1 To 100 Step 1  
        If (P < P_katalog(k)) Then  
            If (Math.Abs(P - P_katalog(k)) < _  
                P_odleglosc) Then  
                k_wybrane = k  
                P_odleglosc = _  
                    Math.Abs(P - P_katalog(k))  
            End If  
        End If  
    Next  
    ' Wydruk informacji na temat wybranego wariantu  
    ' reduktora k_wybrane, jego moc P_katalog(k_wybrane)  
End Sub
```

Rysunek 3.13. Program wyszukujący wariant najbliższy poszukiwanemu wg strategii 1)

3.4. Podsumowanie

W kolejnych rozdziałach niniejszego opracowania zamieszczono szczegółowe algorytmy dotyczące szerokiej grupy problemów. Dobór i sposób grupowania algorytmów uwzględnia przede wszystkim potrzeby inżynierów zajmujących się specjalistycznymi zagadnieniami spotykanymi w budowie maszyn.

4

Algorytmy generujące

4.1. Wprowadzenie

W rozdziale zebrano i przedstawiono kilka algorytmów, które mogą być przydatne w procesie tworzenia oprogramowania, a których wspólnym mianownikiem jest fakt generowania przez algorytm określonych obiektów o określonych cechach matematycznych.

Dwa pierwsze algorytmy dotyczą generowania liczb losowych. Algorytmy tej klasy pozwalają generować losowo, na ogół przy rozkładzie równomiernym, wartości liczbowe z określonego przedziału. Stosujemy je w procesie optymalizacji gdzie interesują nas określone przedziały zmienności wybranych zmiennych decyzyjnych i ich wpływ na inne wielkości wynikowe (funkcje kryterialne), przy czym nie ma żadnych innych przesłanek przemawiających za doбором określonych ich wartości. Aby uzyskać pełniejszy obraz skutków doboru różnych wartości zmiennych decyzyjnych stosujemy właśnie generatory liczb losowych.

Proces generowania liczb losowych może być prowadzony dla pojedynczej zmiennej lub dla wielu zmiennych.

Zwykle zadania optymalizacyjne, w których stosowanie generatorów liczb losowych jest przydatne, polegają na wyborze określonych wariantów rozwiązania problemu z obszernej przestrzeni decyzyjnej. Mogą to być np. charakterystyki modeli mechanicznych układów dynamicznych. W przypadku zadań projektowych są to najczęściej wartości parametrów projektowych, które mogą przyjmować wartości z określonych przedziałów zmienności.

Kolejne dwa algorytmy dotyczą: złotego podziału - możliwości iteracyjnego generowania odpowiednich wartości liczbowych, oraz tworzenia kolejnych wyrazów określonego ciągu. Oba algorytmy mogą być przydatne w zadaniach optymalizacji czy też zadaniach optymalizacji powiązanych z próbami wygenerowania ograniczonego zbioru rozwiązań standardowych.

Następny z prezentowanych algorytmów pokazuje możliwości w zakresie operowania informacją tekstową. Pokazano jedną z możliwych operacji. Struktury programistyczne tego typu leżą u podstaw modułów języków problemowo-zorientowanych często stosowanych do sterowa-

nia oprogramowaniem inżynierskim. Mogą również wystąpić w opisach struktur danych używanych w oprogramowaniu.

Ostatni z przedstawianych algorytmów ilustruje w jaki sposób można zamienić liczbę o podstawie dziesiętnej na liczbę o innej podstawie. Praktyczne stosowanie tej klasy podejść zdarza się dzisiaj rzadko w rozważanej w pracy klasie zastosowań. Może być jednak przydatne w przypadku rozwoju oprogramowania, które powstało w przeszłości.

4.2. Metoda Monte Carlo – generowanie liczb losowych w zadanym zakresie

Generator liczb pseudolosowych generuje kolejne liczby losowe z przedziału 0 – 1. Aby uzyskać liczbę losową stosujemy funkcję wewnętrzną Visual Basic'a: Rnd():

Funkcja Rnd ([liczba])

gdzie argument *liczba* może być dowolnym wyrażeniem liczbowym

Funkcja Rnd zwraca liczbę przypadkową z przedziału:

$$0 \leq \text{liczba przypadkowa} < 1$$

Tabela 4.1

Jeśli argument <i>liczba</i> ma wartość	Rnd(<i>liczba</i>) zwraca
Mniejszą od zera	cały czas tę samą liczbę przypadkową używając do jej wyliczenia argumentu <i>liczba</i> jako „ziarno”.
Większą od zera	następną liczbę przypadkową w kolejności.
Równą zero	ostatnio wygenerowaną liczbę przypadkową.
Brak wartości	kolejną liczbę przypadkową w obliczanej sekwencji liczb.

Sekwencja liczb przypadkowych generowanych przez funkcję Rnd jest zawsze taka sama. Ma to dobrą i złą stronę.

Jeśli testujemy oprogramowanie wykorzystujące tę funkcję to najczęściej zależy nam na powtarzalności obliczeń i fakt, że sekwencja generowanych liczb przypadkowych jest za każdym razem identyczna jest pożyteczny.

Jeśli jednak aplikacja ma spełniać swoją rolę i wygenerowana liczba przypadkowa ma być trudna do przewidzenia, to fakt powtarzalności generatora jest sytuacją niezadowalającą.

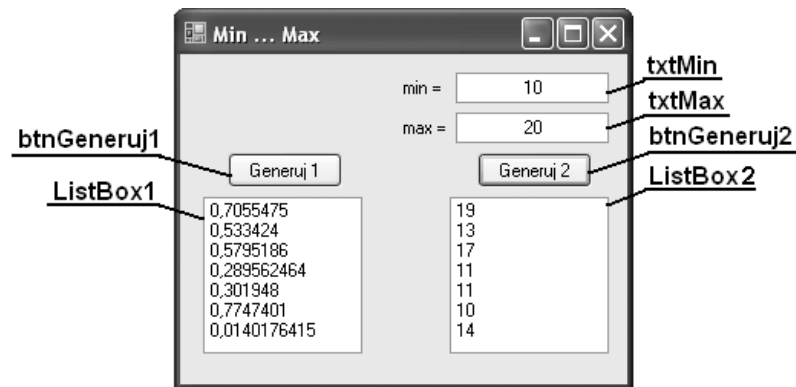
Aby sekwencja generowanych liczb była trudna do przewidzenia i za każdym uruchomieniem programu inna, należy wywołać jeden raz, najlepiej podczas uruchamiania aplikacji instrukcję Random, bez argumentów. Funkcja ta dostarczy funkcji Rnd wartość „ziarna” pobraną z dziesiętnych części sekund czasu systemowego, co gwarantuje niepowtarzalność startu generatora.

Aby uzyskać liczbę przypadkową z zadanego przedziału, np. z przedziału od *minimum* do *maksimum* należy posłużyć się wyrażeniem:

```
Liczba=Int((maksimum–minimum+1) * Rnd + minimum)
```

Przykład aplikacji w języku Visual Basic

Zbudujmy aplikację, rysunek 4.1, która wyposażona będzie w dwa przyciski Generuj 1 i Generuj 2.



Rysunek 4.1. Postać formularza

Przycisk Generuj 1 niech zapełnia listę liczbami przypadkowymi z przedziału 0 – 1, a przycisk Generuj 2 niech zapełnia drugą listę liczbami przypadkowymi z przedziału określonego na formularza przez wartości wpisane w pola tekstowe txtMin i txtMax.

Kod programu

```
Private Sub Form1_Load(ByVal sender As _
    System.Object, ByVal e As _
    System.EventArgs) Handles MyBase.Load
    'Randomize()
End Sub
-----
Private Sub btnGeneruj1_Click(ByVal sender _
    As System.Object, ByVal e As _
    System.EventArgs) _
    Handles btnGeneruj1.Click
    Dim LiczbaLosowa As Single
    LiczbaLosowa = Rnd()
    ListBox1.Items.Add(LiczbaLosowa)
End Sub
-----
Private Sub btnGeneruj2_Click(ByVal sender _
    As System.Object, ByVal e As _
    System.EventArgs) _
    Handles btnGeneruj2.Click
    Dim min, max As Single
    Dim LiczbaLosowa As New Random
    min = Single.Parse(txtMin.Text)
    max = Single.Parse(txtMax.Text)
    ListBox2.Items.Add(LiczbaLosowa.Next _
        (min, max + 1))
End Sub
```

Proszę zwrócić uwagę, że w procedurze *Private Sub Form1_Load* instrukcja *Randomize* nie działa – jest „zakomentowana” (patrz znak apostrofu wstawiony jako pierwszy znak w wierszu). Oznacza to, że po uruchomieniu aplikacji zawsze uzyskamy identyczne wartości liczb losowych.

Jeśli aplikacja, po jej sprawdzeniu, działa prawidłowo – należy usunąć apostrof przed instrukcją *Randomize*, rozpocznie ona wtedy swoje działanie i liczby pseudolosowe będą przy każdym uruchomieniu aplikacji generowane z innym „ziarnem”, czyli w innej kolejności.

4.3. Metoda Monte Carlo – prosty generator liczb losowych (pseudolosowych)

W różnych zastosowaniach programistycznych (gry komputerowe, zastosowania kryptograficzne do generowania haseł, programy komputerowe metody Monte Carlo) istnieje potrzeba uzyskania ciągu liczb o wartościach losowych.

Generatory liczb losowych mogą być sprzętowe - działające na zasadzie generowania szumu elektronicznego i programowe - będące procedurą obliczającą ciąg liczb. Liczby te nie są w rzeczywistości liczbami przypadkowymi lecz w pewnym zakresie spełniają potrzebę przypadkowości, stąd ich nazwa – generatory liczb pseudolosowych. Generator liczb pseudolosowych generuje liczby z przedziału 0 – 1.

Najprostszy algorytm generatora ma następującą postać:

Wybieramy dwie liczby $a < 1$ i $b \gg 1$

Mnożymy $c = a * b$

Pobieramy ułamek dziesiętny z liczby c , który jest liczbą pseudolosową.

Przykład

Na początku przyjmujemy dwie dowolne liczby $b = 98765$, $a = 0,758493$ jedną dużo większą od 1, drugą mniejszą od 1.

$$c = 98765 * 0,758493 = 74912,561145$$

$$a = 0,561145$$

$$c = 98765 * 0,561145 = 55421,485925$$

$$a = 0,485925$$

$$c = 98765 * 0,485925 = 47992,382625$$

$$a = 0,382625$$

ROZDZIAŁ 4

Otrzymujemy ciąg liczb pseudolosowych:

$a = 0,758493$

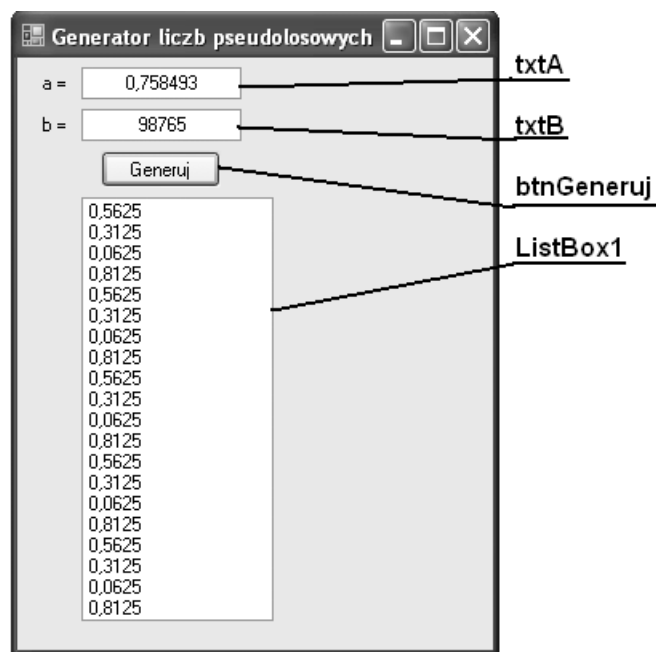
$a = 0,561145$

$a = 0,485925$

$a = 0,382625$

Program generujący liczby pseudolosowe zaczyna po pewnej liczbie cykli generować ponownie te same wartości. Na długość cyklu i równomierność rozkładu generatora mają wpływ przyjęte dwie początkowe wartości a i b .

Przykład aplikacji w języku Visual Basic



Rysunek 4.2. Postać formularza

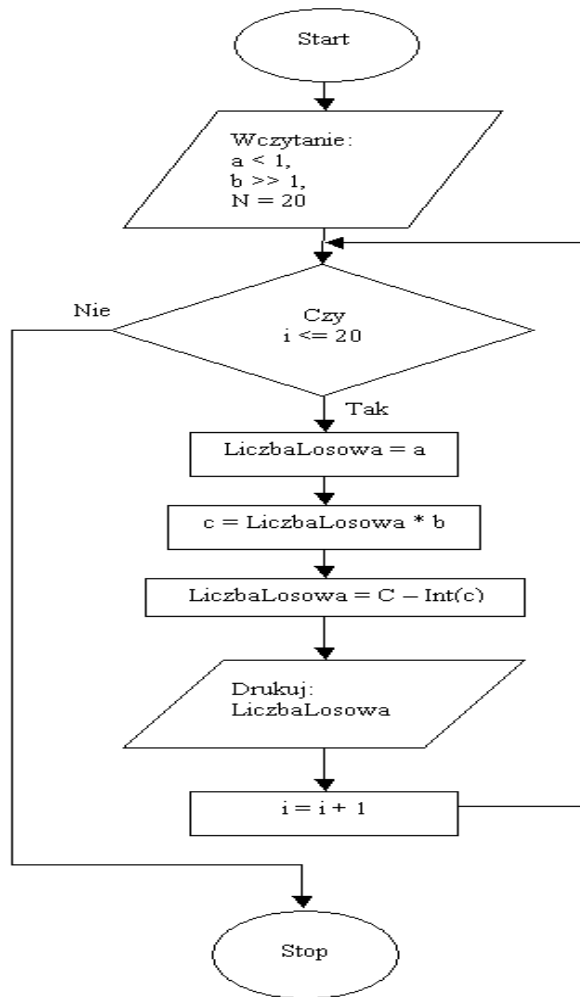
Kod programu

```
Private Sub btnGeneruj_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs)
    Handles btnGeneruj.Click
    Dim a, b, LiczbaLosowa As Single
    a = Single.Parse(txtA.Text)
    LiczbaLosowa = a
    b = Single.Parse(txtB.Text)
    For i = 1 To 20
        Call Generator(LiczbaLosowa, b)
        ListBox1.Items.Add(LiczbaLosowa.ToString)
    Next
End Sub
```

```
Private Sub Generator(ByRef a, ByVal b)
    Dim c As Single
    c = a * b
    a = c - Int(c)
End Sub
```

W tak prostym generatorze ujawnia się wiele jego ograniczeń i niedoskonałości. Np. na rysunku formularza widać, że przy liczbach początkowych $a = 0,758493$ $b = 98765$ okres generatora wynosi 4 liczby, a przy liczbach: $a = 0,758493$, $b = 987$ okres ten jest dużo większy.

Inną niedoskonałością tak przyjętego algorytmu jest niebezpieczeństwo, że jeśli kolejna liczba losowa przyjmie wartość zero, to wszystkie następane też będą miały wartość zerową np. dla wartości $a = 0,758493$ $b = 9876$.



Rysunek 4.3. Algorytm generowania 20 liczb pseudolosowych

4.4. Złoty podział odcinka

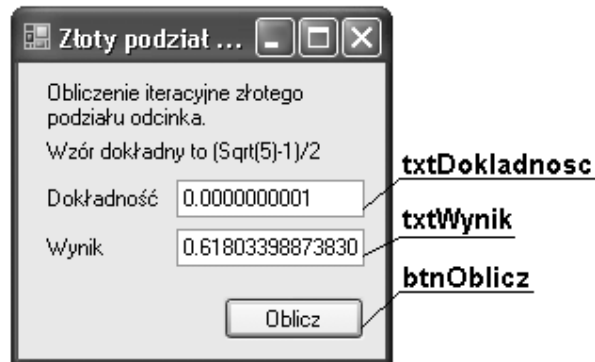
Wartość złotego podziału odcinka wyliczana jest iteracyjnie ze wzoru:

$$Y = \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \dots}}}}$$

Daną wejściową jest wymagana dokładność obliczeń. Algorytm porównuje oszacowania podziału odcinka z dwóch kolejnych kroków i kończy pracę gdy różnica kolejnych oszacowań jest mniejsza od założonej dokładności.

Dla sprawdzenia obliczeń można użyć wzoru $\frac{\sqrt{5}-1}{2}$.

Przykład aplikacji w języku Visual Basic

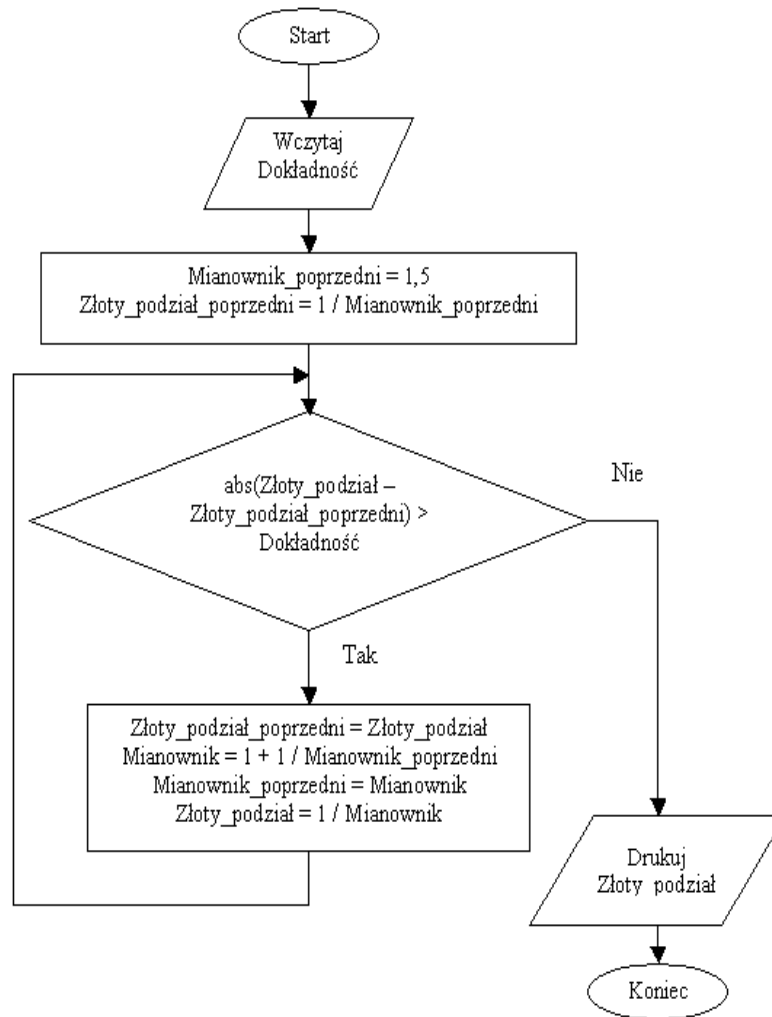


Rysunek 4.4

Kod programu

```
Private Sub btnOblicz_Click(ByVal sender As _  
    System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnOblicz.Click  
    Dim Dokladnosc As Double  
    Dokladnosc = Cdbl(txtDokladnosc.Text)  
    txtWynik.Text = CStr(ZlotyPodzial(Dokladnosc))  
End Sub
```

```
Private Function ZlotyPodzial(ByVal Dokladnosc As _  
    Double) As Double  
    Dim mian_p, mian, ZlotyPodzial_p As Double  
    mian_p = 1.5  
    ZlotyPodzial = 1 / mian_p  
    Do While Math.Abs(ZlotyPodzial - ZlotyPodzial_p) _  
        > Dokladnosc  
        ZlotyPodzial_p = ZlotyPodzial  
        mian = 1 + 1 / mian_p  
        mian_p = mian  
        ZlotyPodzial = 1 / mian  
    Loop  
End Function
```

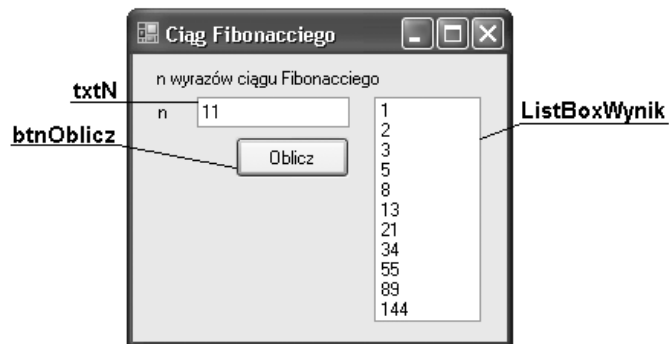


Rysunek 4.5. Schemat algorytmu

4.5. Wygenerowanie n wyrazów ciągu Fibonacciego

Program generuje n wyrazów ciągu Fibonacciego i wpisuje je do formantu listy. Kolejny wyraz ciągu jest sumą dwóch poprzednich. Daną wejściową jest liczba wyrazów n. Algorytm generowania kolejnego wyrazu ciągu sumuje dwa poprzednie wyrazy. W związku z tym jest problem z wygenerowaniem pierwszych dwóch wyrazów. Dlatego program przyjmuje, że pierwszy wyraz jest równy 1 oraz poprzedzający go również jest równy 1. Dzięki temu możliwe jest wygenerowanie następujących wyrazów ciągu.

Przykład aplikacji w języku Visual Basic

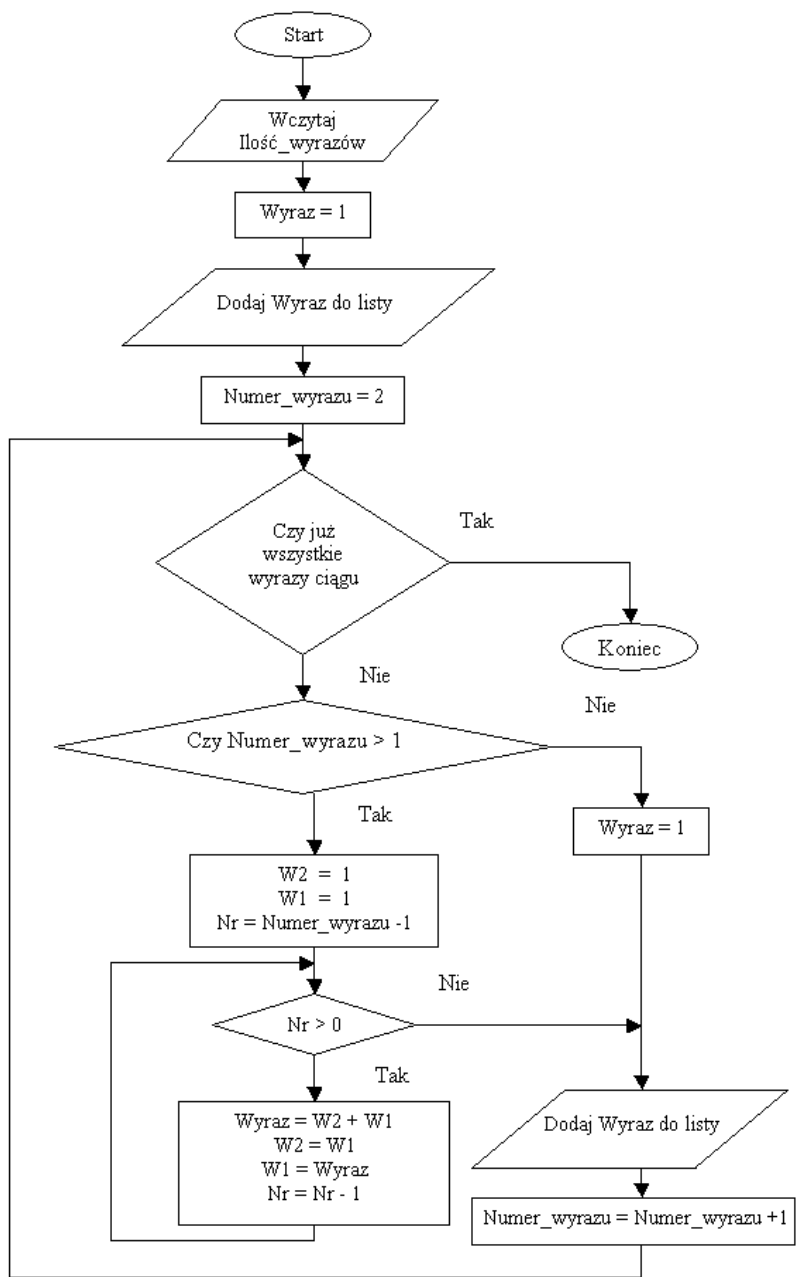


Rysunek 4.5

Kod programu

```
Private Sub btnOblicz_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnOblicz.Click
    Dim n As Integer
    n = CInt(txtN.Text)
    ListBoxWynik.Items.Add(1)
    For i = 2 To n
        ListBoxWynik.Items.Add(Fibonacci(i))
    Next
End Sub
```

```
Private Function Fibonacci(ByVal n As Integer) _
    As Integer
    Dim W_2, W_1, nr As Integer
    If n = 0 Or n = 1 Then
        Fibonacci = 1
    Else
        W_2 = 1
        W_1 = 1
        nr = n - 1
        Do While nr > 0
            Fibonacci = W_2 + W_1
            W_2 = W_1
            W_1 = Fibonacci
            nr = nr - 1
        Loop
    End If
End Function
```



Rysunek 4.6. Schemat algorytmu

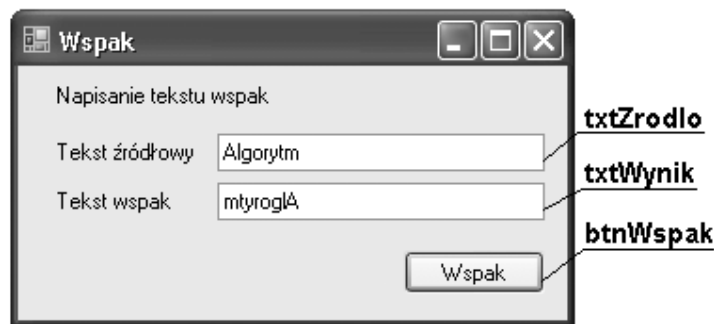
4.6. Napisanie tekstu wspak

Program zadany tekst wypisuje wspak. Daną wejściową jest tekst źródłowy. W procedurze zastosowano funkcje operacji na ciągach znakowych:

- *Len*(ciąg_znakowy) – zwraca długość ciągu znakowego
- *Mid*(ciąg_znakowy, pozycja_startowa, długość_ciagu) – funkcja wycina z ciągu znakowego podciąg zaczynając od pozycji startowej o określonej w trzecim parametrze długości ciągu.

Ciąg wynikowy uzyskujemy poprzez przestawienie w pętli po jednym znaku zaczynając od końca ciągu źródłowego.

Przykład aplikacji w języku Visual Basic



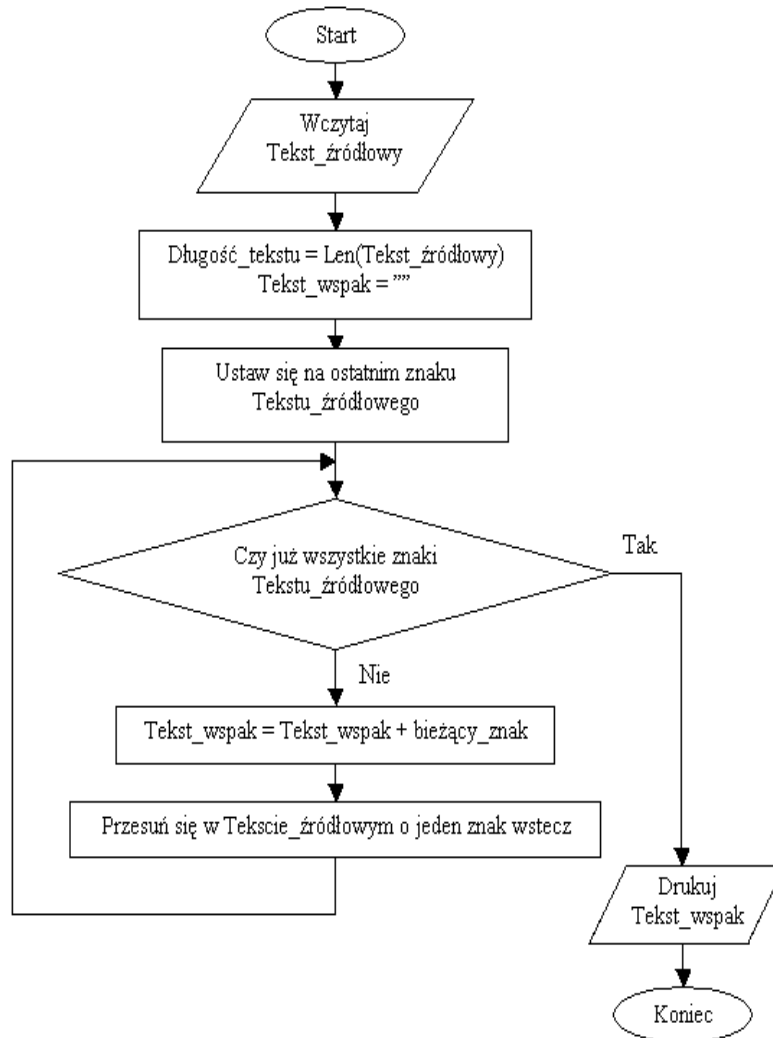
Rysunek 4.6

Kod programu

```
Private Sub btnWspak_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnWspak.Click
    Dim Zrodlo As String
    Zrodlo = txtZrodlo.Text
    txtWynik.Text = Wspak(Zrodlo)
End Sub
```

ROZDZIAŁ 4

```
Private Function Wspak(ByVal Tekst_zrodlowy As _  
                        String) As String  
    Dim Dlugosc As Integer  
    Dlugosc = Len(Tekst_zrodlowy)  
    Wspak = ""  
    For i = Dlugosc To 1 Step -1  
        Wspak = Wspak & Mid(Tekst_zrodlowy, i, 1)  
    Next  
End Function
```



Rysunek 4.7. Schemat algorytmu

4.7. Zamiana liczby dziesiętnej na liczbę o innej podstawie i odwrotnie

Algorytm przekształca liczbę dziesiętną na liczbę wyrażoną w innym systemie liczbowym np. dwójkową, ósemkową itp. Podstawa tego systemu musi być w zakresie od dwóch do dziewięciu. Możliwa jest również konwersja odwrotna. Danymi wejściowymi przy zamianie liczby dziesiętnej są:

- liczba dziesiętna
- podstawa innego systemu liczbowego

Algorytm zamiany liczby dziesiętnej na liczbę o innej podstawie jest następujący:

- obliczamy resztę z dzielenia liczby dziesiętnej przez podstawę innego systemu liczbowego i otrzymany rezultat mnożymy przez podstawę
- uzyskany wynik przekształcamy na znak i zapisujemy jako najmniej znaczącą pozycją wyniku
- od przekształcanej liczby odejmujemy uzyskaną resztę i dzielimy przez podstawę innego systemu liczbowego; rezultat tej operacji staje się nową liczbą do przekształcania
- proces prowadzimy w pętli dopóki liczba przekształcana jest większa od zera dopisując nowe znaki wyniku po lewej stronie ciągu znakowego

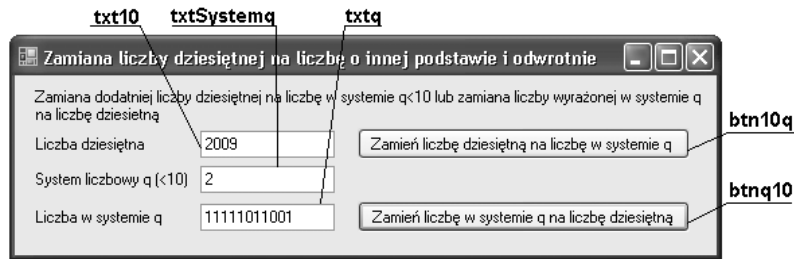
Danymi wejściowymi przy zamianie liczby z innego systemu liczbowego na dziesiętną są:

- liczba w innym systemie (np. dwójkowa)
- podstawa tego systemu (w tym przypadku 2)

Algorytm zamiany liczby o innej podstawie na liczbę dziesiętną jest następujący:

- określamy z ilu znaków składa się dana liczba
- wykonujemy pętlą dla każdego znaku liczby
- podstawę innego systemu liczbowego podnosimy do potęgi odpowiadającej pozycji znaku w liczbie i mnożymy przez wartość jaką reprezentuje ten znak
- rezultaty poprzedniej operacji sumujemy w pętli

Przykład aplikacji w języku Visual Basic



Rysunek 4.8

Kod programu

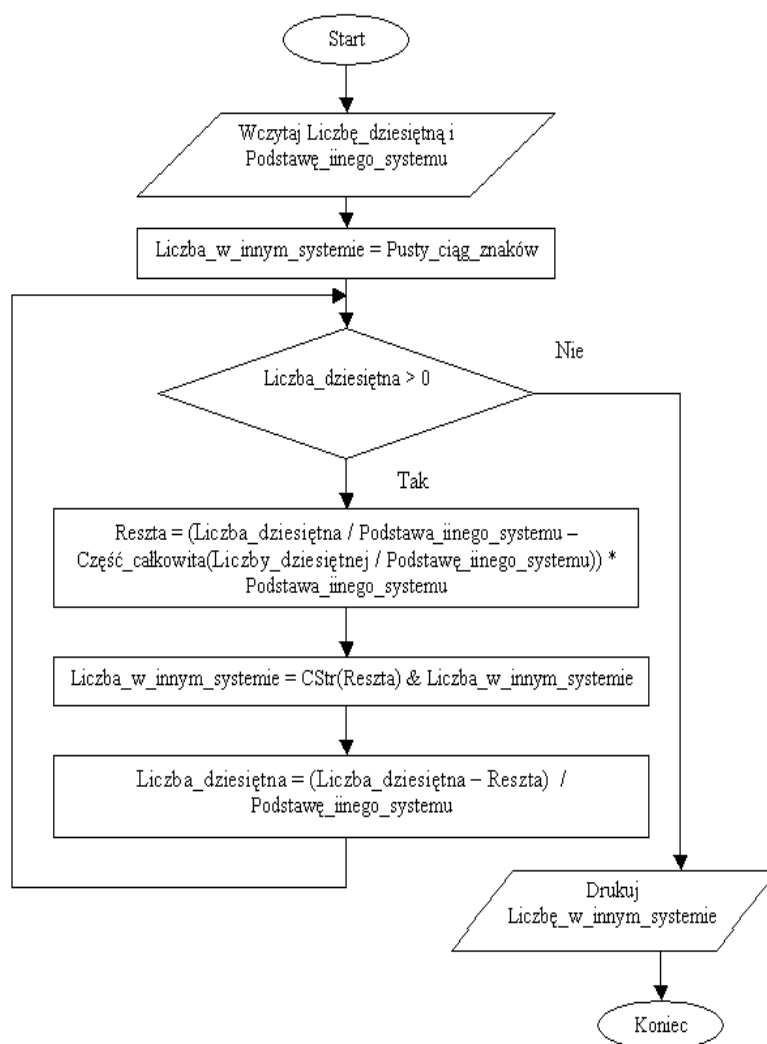
```

Private Sub btn10q_Click(ByVal sender As _
    System.Object, _
    ByVal e As System.EventArgs) _
    Handles btn10q.Click
    Dim L10, LSq As Integer
    L10 = CInt(txt10.Text)
    LSq = CInt(txtSystemq.Text)
    txtq.Text = K10q(L10, LSq)
End Sub
-----
Private Sub btnq10_Click(ByVal sender As
System.Object, _
    ByVal e As System.EventArgs) Handles
btnq10.Click
    Dim LSq, Lq As Integer
    Lq = CInt(txtq.Text)
    LSq = CInt(txtSystemq.Text)
    txt10.Text = CStr(Kq10(Lq, LSq))
End Sub
-----

```

```
Private Function K10q(ByVal Liczba10 As Integer, _  
                    ByVal Systemq As Integer) As String  
    Dim reszta As Integer  
    K10q = ""  
    Do While Liczba10 > 0  
        reszta = (Liczba10 / Systemq - _  
                Math.Truncate(Liczba10 / Systemq))  
    * Systemq  
        K10q = CStr(reszta) & K10q  
        Liczba10 = (Liczba10 - reszta) / Systemq  
    Loop  
End Function
```

```
Private Function Kq10(ByVal Liczbaq As Integer, _  
                    ByVal Systemq As Integer) As Integer  
    Dim Lq_dlugosc, i As Integer  
    Lq_dlugosc = Len(CStr(Liczbaq))  
    For i = 1 To Lq_dlugosc  
        Kq10 = Kq10 + _  
            CInt(Mid(CStr(Liczbaq), i, 1)) * Systemq ^  
            (Lq_dlugosc - i)  
    Next i  
End Function
```



Rysunek 4.9. Schemat algorytmu



Operacje geometryczne

5.1. Wprowadzenie

W rozdziale przedstawiono przykład algorytmu, który ilustruje w jaki sposób modelować i w jaki sposób przetwarzać modele geometryczne. Zwrócono uwagę na problem opisu obiektów geometrycznych jak i problem komputerowego badania relacji pomiędzy istniejącymi obiektami.

5.2. Budowanie trójkątów

Sprawdzić, czy z danych 3 boków a , b , c można zbudować trójkąt. Sprawdzić czy może to być trójkąt prostokątny:

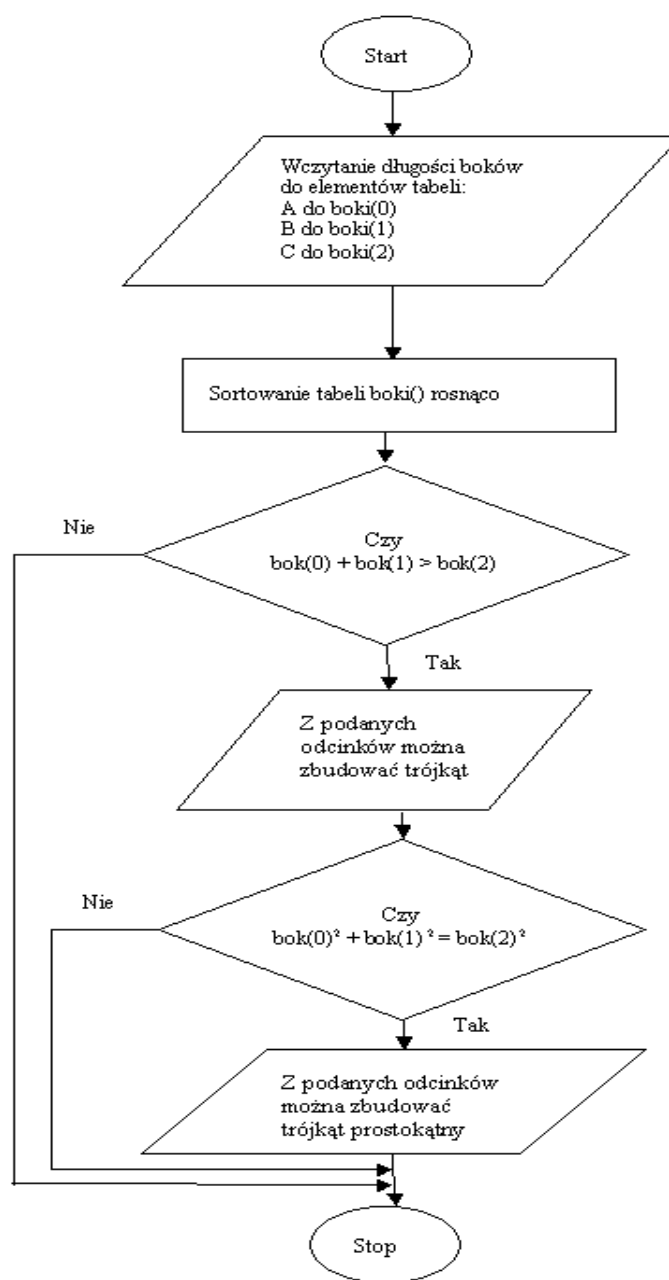
Propozycja algorytmu:

1. Wczytać długości boków do tablicy, np. o nazwie $boki$.
2. Posortować tablicę rosnąco.
3. Jeśli z odcinków można zbudować trójkąt to musi być spełniony warunek:

$$boki(0) + boki(1) > boki(2) \quad (5.1)$$

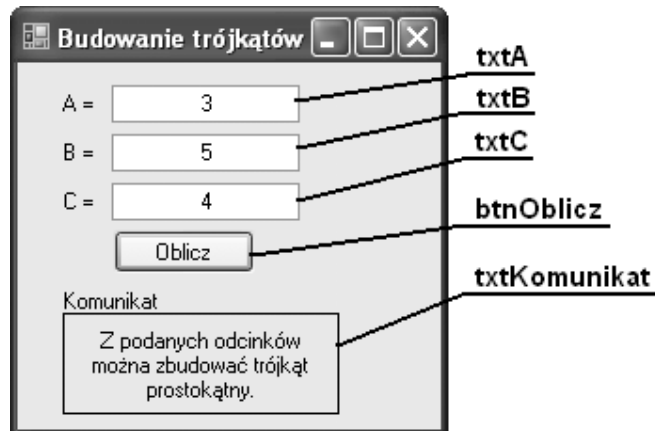
4. Jeśli warunek (1) jest spełniony należy sprawdzić warunek (5.2) decydujący o tym, że trójkąt zbudowany z danych boków jest trójkątem prostokątnym

$$boki(0)^2 + boki(1)^2 = boki(2)^2 \quad (5.2)$$



Rysunek 5.1. Algorytm programu

Przykład aplikacji w języku Visual Basic



Rysunek 5.2.. Propozycja formularza

Kod programu

```

Private Sub btnOblicz_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnOblicz.Click
    Dim boki(2) As Double
    Dim Info As String = ""
    boki(0) = Double.Parse(txtA.Text)
    boki(1) = Double.Parse(txtB.Text)
    boki(2) = Double.Parse(txtC.Text)
    Call SortowanieRosnaco(boki)
    If boki(0) + boki(1) > boki(2) Then
        Info = "Z podanych odcinków" & _
            " można zbudować trójkąt"
        If boki(0) ^ 2 + boki(1) ^ 2 = _
            boki(2) ^ 2 Then
            Info = Info & " prostokątny."
        End If
    Else
        Info = "Z podanych odcinków" & _
            " nie można zbudować trójkąta."
    End If
    lblKomunikat.Text = Info
End Sub

```

ROZDZIAŁ 5

```
Private Sub SortowanieRosnaco(ByRef boki)
    Dim tmp As Double
    For k As Integer = 0 To 1
        For i = 0 To 1
            If boki(i) > boki(i + 1) Then
                tmp = boki(i)
                boki(i) = boki(i + 1)
                boki(i + 1) = tmp
            End If
        Next i
    Next k
End Sub
```

Szczegółowe omówienie procedury sortowania znajduje się w rozdziale 6



Selekcja

6.1. Wprowadzenie

W rozdziale przedstawiono algorytmy, które pokazują w jaki sposób można dokonać selekcji spośród dostępnych rozwiązań.

Pierwszy algorytm ilustruje w jaki sposób można wyznaczyć wartość najmniejszą i największą w zbiorze. Algorytm ten jest podstawą wielu innych algorytmów.

Kolejny algorytm dotyczy problemów, których celem jest selekcja najbardziej pożądanego – ekstremalnego wariantu spośród wielu możliwych wariantów. Zadania są typowo geometryczne o różnym stopniu komplikacji opisu modelu.

6.2. Szukanie najmniejszej lub największej liczby w zbiorze

Szukanie najmniejszej lub największej liczby w zadanym zbiorze liczb bardzo często jest elementem większego zadania, np. w przykładzie „Sortowanie przez wybieranie” wielokrotnie szukamy liczby najmniejszej w malejącym cyklicznie zbiorze liczb.

Poniżej przedstawiono algorytmy szukania najmniejszej lub największej liczby w zbiorze.

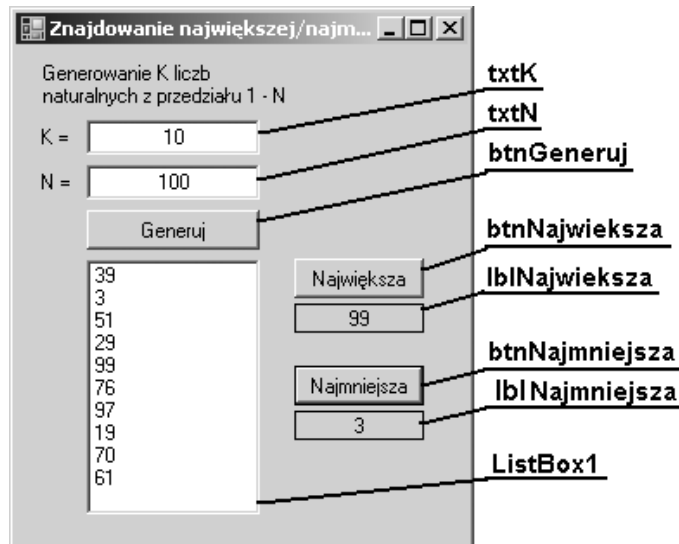
Algorytm znajdowania najmniejszej liczby w zbiorze

1. Przyjęcie za najmniejszą liczbę pierwszą liczbę w zbiorze liczb.
2. Porównywanie kolejno tej liczby z następnymi w zbiorze, poczynając od drugiej liczby do ostatniej.
3. Jeśli okaże się, że kolejna liczba jest mniejsza od przyjętej w punkcie 1, to za najmniejszą przyjmuje się tę kolejną.

Algorytm szukania największej liczby w zbiorze

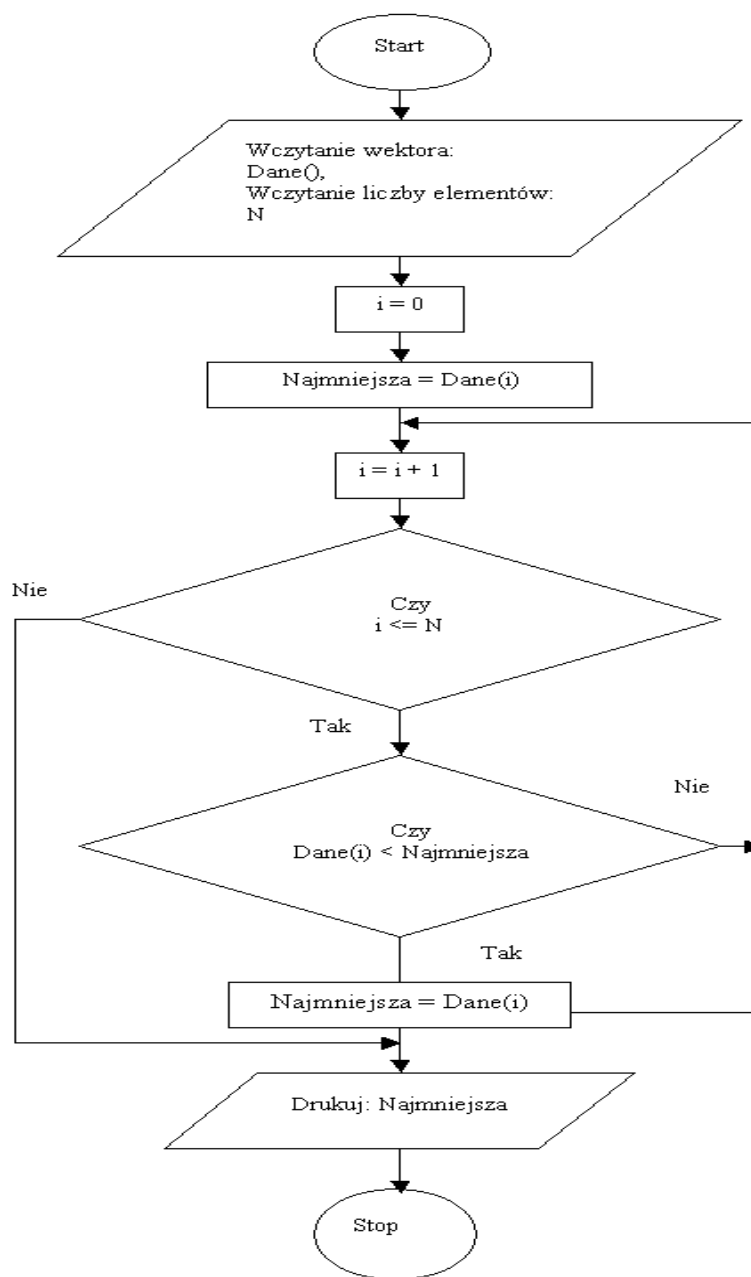
1. Przyjęcie za największą liczbę pierwszą liczbę w zbiorze liczb.
2. Porównywanie kolejno tej liczby z następnymi w zbiorze, poczynając od drugiej liczby do ostatniej.
3. Jeśli okaże się, że kolejna liczba jest większa od przyjętej w punkcie 1, to za najmniejszą przyjmuje się tę kolejną.

Przykład aplikacji w języku Visual Basic



Rysunek 6.1. Propozycja formularza

Aplikacja jest rozbudowana o część kodu generującą zbiór liczb całkowitych przypadkowych z przedziału [1, N]. Zbiór ten wizualizowany jest na formularzu w obiekcie Lista. Liczba elementów zbioru liczb i górna granica przedziału mogą być wprowadzane z klawiatury.



Rysunek 6.2. Algorytm szukania najmniejszej liczby w zbiorze liczb Dane(N)

Kod programu

```
Private Sub Form1_Load(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles MyBase.Load
    Randomize()
End Sub


---


Private Sub btnGeneruj_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnGeneruj.Click
    Dim K, N, i As Integer
    Dim LiczbaPrzypadkowa As Integer
    K = Integer.Parse(txtK.Text)
    N = Integer.Parse(txtN.Text)
    ListBox1.Items.Clear()
    For i = 1 To K
        LiczbaPrzypadkowa = Int(N * Rnd() + 1)
        ListBox1.Items.Add(LiczbaPrzypadkowa.ToString)
    Next
    lblNajmniejsza.Text = ""
    lblNajwieksza.Text = ""
End Sub


---


Private Sub btnNajmniejsza_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnNajmniejsza.Click
    Dim M As Integer
    Dim Najmniejsza As Integer
    Dim Liczba As Integer
    M = ListBox1.Items.Count
    Najmniejsza = _
        Integer.Parse(ListBox1.Items.Item(0))
    For i = 1 To M - 1
        Liczba = _
            Integer.Parse(ListBox1.Items.Item(i))
        If Najmniejsza > Liczba Then
            Najmniejsza = Liczba
        End If
    Next
    lblNajmniejsza.Text = Najmniejsza.ToString
End Sub


---


Private Sub btnNajwieksza_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnNajwieksza.Click
    Dim M As Integer
    Dim Najwieksza As Integer
    Dim Liczba As Integer
    M = ListBox1.Items.Count
```

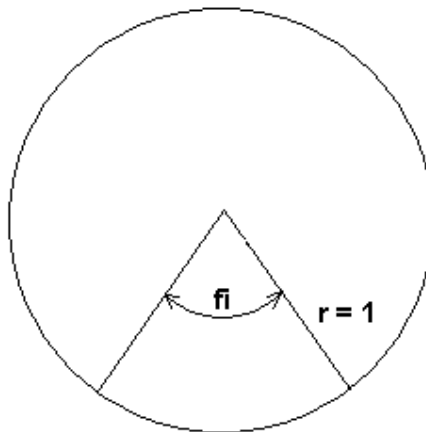
```

Najwieksza = _
    Integer.Parse(ListBox1.Items.Item(0))
For i = 1 To M - 1
    Liczba = _
        Integer.Parse(ListBox1.Items.Item(i))
    If Najwieksza < Liczba Then
        Najwieksza = Liczba
    End If
Next
lblNajwieksza.Text = Najwieksza.ToString
End Sub

```

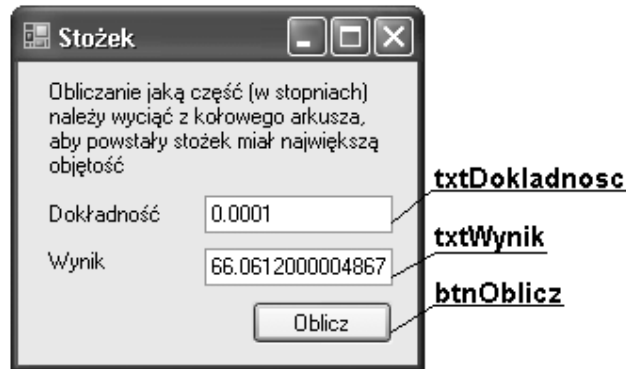
6.3. Największa objętość stożka

Z kołowego arkusza usuwamy wycinek i składamy arkusz uzyskując stożek. Algorytm wylicza jaką część (wynik w stopniach) należy wyciąć, aby uzyskać maksymalną objętość stożka. Daną wejściową jest dokładność obliczeń. Dla uproszczenia przyjęto, że promień okręgu na arkuszu jest równy 1. W pętli zmieniany jest kąt ϕ_i w zakresie od zera do $2*\pi$ z krokiem odpowiadającym wymaganej dokładności. W każdym kroku obliczana jest objętość stożka. Jeśli otrzymana wartość jest większa od największej dotychczas uzyskanej wówczas zapamiętywana jest bieżąca objętość jako objętość maksymalna i notowana jest wartość kąta, przy którym taka sytuacja wystąpiła.



Rysunek 6.3

Przykład aplikacji w języku Visual Basic



Rysunek 6.4

Kod programu

```

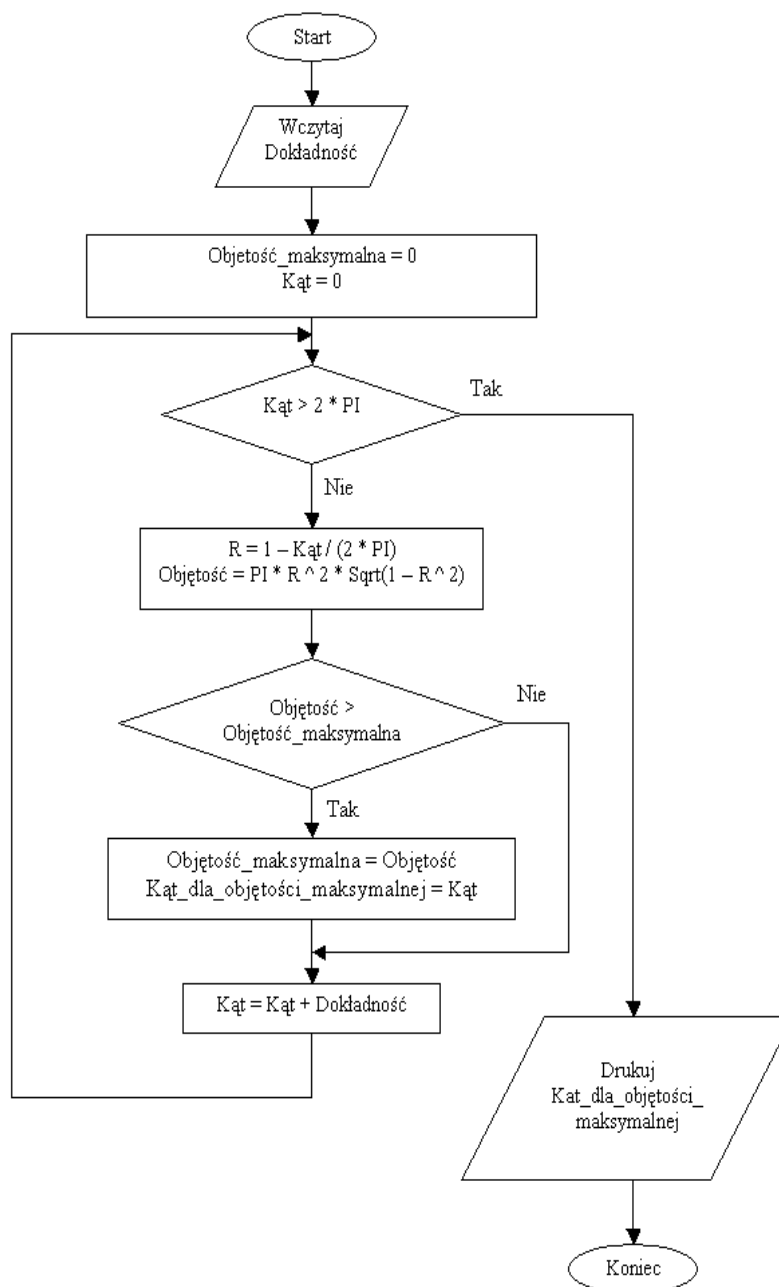
Private Sub btnOblicz_Click(ByVal sender As _
    System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnOblicz.Click
    Dim Dokladnosc_radiansy As Double
    Dokladnosc_radiansy = CDb1(txtDokladnosc.Text) * _
        Math.PI / 180
    txtWynik.Text = _
        CStr(KatObjetoscStozkaMax(Dokladnosc_radiansy) _
            * 180 / Math.PI)
End Sub

```

```

Private Function KatObjetoscStozkaMax(ByVal _
    Dokladnosc As Double) _
    As Double
    Dim i, Objetosc, ObjetoscMax, r As Double
    ObjetoscMax = 0
    For i = 0 To 2 * Math.PI Step Dokladnosc
        r = 1 - i / (2 * Math.PI)
        Objetosc = Math.PI * r ^ 2 * _
            Math.Sqrt(1 - r ^ 2) / 3
        If Objetosc > ObjetoscMax Then
            ObjetoscMax = Objetosc
            KatObjetoscStozkaMax = i
        End If
    Next
End Function

```



Rysunek 6.5. Schemat algorytmu



Algorytmy matematyczne

7.1. Wprowadzenie

W rozdziale przedstawiono szereg algorytmów, które ilustrują w jaki sposób modelować i rozwiązywać problemy matematyczne na komputerze.

Pierwszy przykład pokazuje jak można obliczać iteracyjnie wartość liczby PI i jak można decydować o uzyskiwanej dokładności obliczeń. Drugi przykład przedstawia problem badania czy dana liczba jest liczbą pierwszą. Przykład trzeci pokazuje algorytm obliczania pierwiastka kwadratowego z zadaną dokładnością.

Kolejny algorytm dotyczy klasycznego problemu obliczania pierwiastków trójmianu kwadratowego.

W następnych podrozdziałach pokazano w jaki sposób obliczyć wartość silni, sposób znajdowania największego wspólnego dzielnika za pomocą algorytmu Euklidesa oraz algorytm szyfrowania.

Przedstawione algorytmy i przykłady ilustrują kluczowe dla wielu przetwarzanych problemów inżynierskich zagadnienia związane z dokładnością przeprowadzanych obliczeń, możliwością wpływania na ich jakość, klasyfikowania uzyskiwanych rozwiązań czy też stosowania bardzo specyficznych algorytmów pozwalających wyznaczyć określone rozwiązanie.

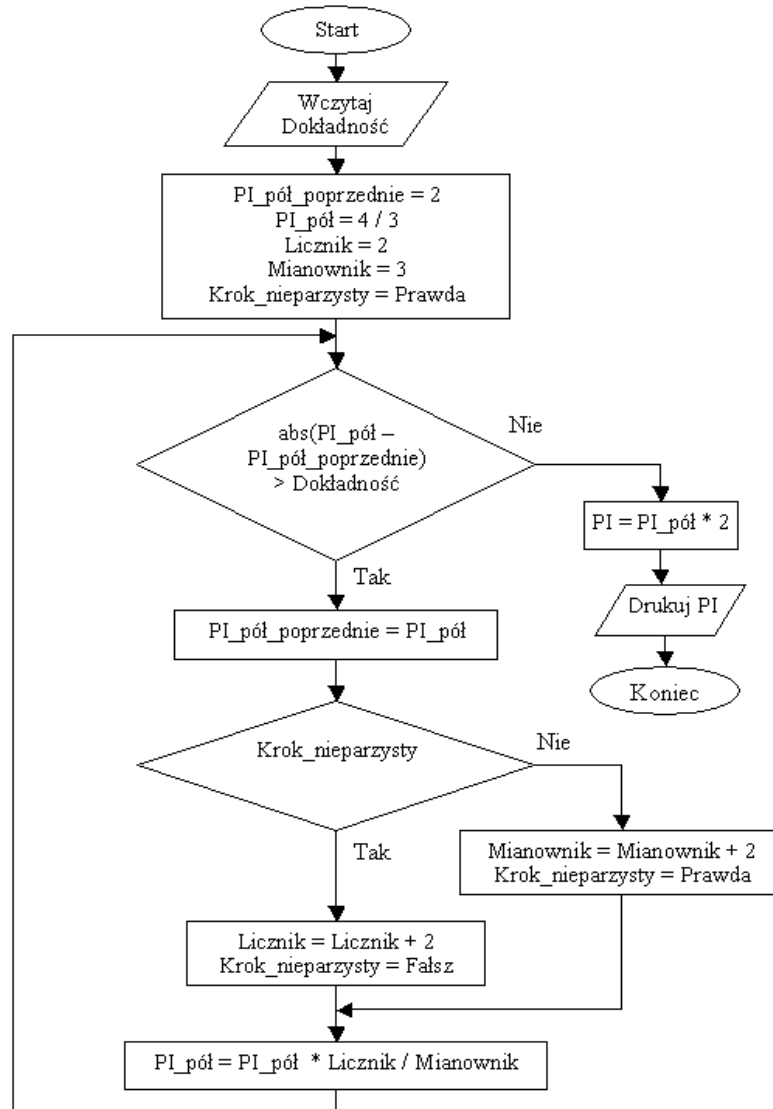
7.2. Obliczenie wartości liczby PI z zadaną dokładnością

Wartość liczby PI wyliczana jest z iteracyjnego wzoru:

$$\pi = 2 * \frac{2 * 2 * 4 * 4 * 6 * 6 * 8 * 8 * \dots}{1 * 3 * 3 * 5 * 5 * 7 * 7 * 9 * \dots} \quad (7.1)$$

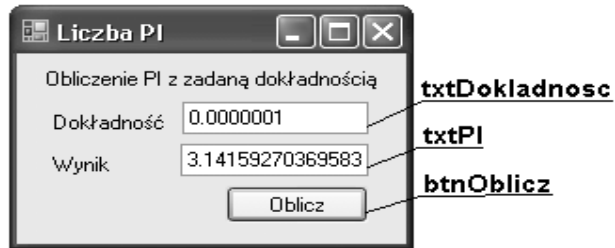
Daną wejściową jest wymagana dokładność oszacowania liczby PI. Algorytm porównuje oszacowania liczby PI z dwóch kolejnych kroków i

kończy pracę, gdy różnica kolejnych oszacowań jest mniejsza od założonej dokładności. Przy dodawaniu w pętli kolejnych czynników do licznika i mianownika trzeba odróżnić kroki parzyste i nieparzyste. W parzystych zwiększany o 2 jest kolejny czynnik mianownika a przy nieparzystych licznika.



Rysunek 7.1. Schemat algorytmu

Przykład aplikacji w języku Visual Basic



Rysunek 7.2

Kod programu

```

Private Sub btnOblicz_Click(ByVal sender As _
    System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnOblicz.Click
    Dim Dokladnosc As Double
    Dokladnosc = CDb1(txtDokladnosc.Text)
    txtPI.Text = CStr(PI(Dokladnosc))
End Sub

Private Function PI(ByVal Dokladnosc As Double) As _
    Double

    Dim Licznik, Mianownik As Integer
    Dim krok_nieparzysty As Boolean
    Dim PI_p_2, PI_2 As Double
    PI_p_2 = 2
    PI_2 = 4 / 3
    Licznik = 2
    Mianownik = 3
    krok_nieparzysty = True
    Do While Math.Abs(PI_2 - PI_p_2) * 2 > Dokladnosc
        PI_p_2 = PI_2
        If krok_nieparzysty Then
            Licznik = Licznik + 2
            krok_nieparzysty = False
        Else
            Mianownik = Mianownik + 2
            krok_nieparzysty = True
        End If
        PI_2 = PI_2 * Licznik / Mianownik
    Loop
    PI = PI_2 * 2
End Function

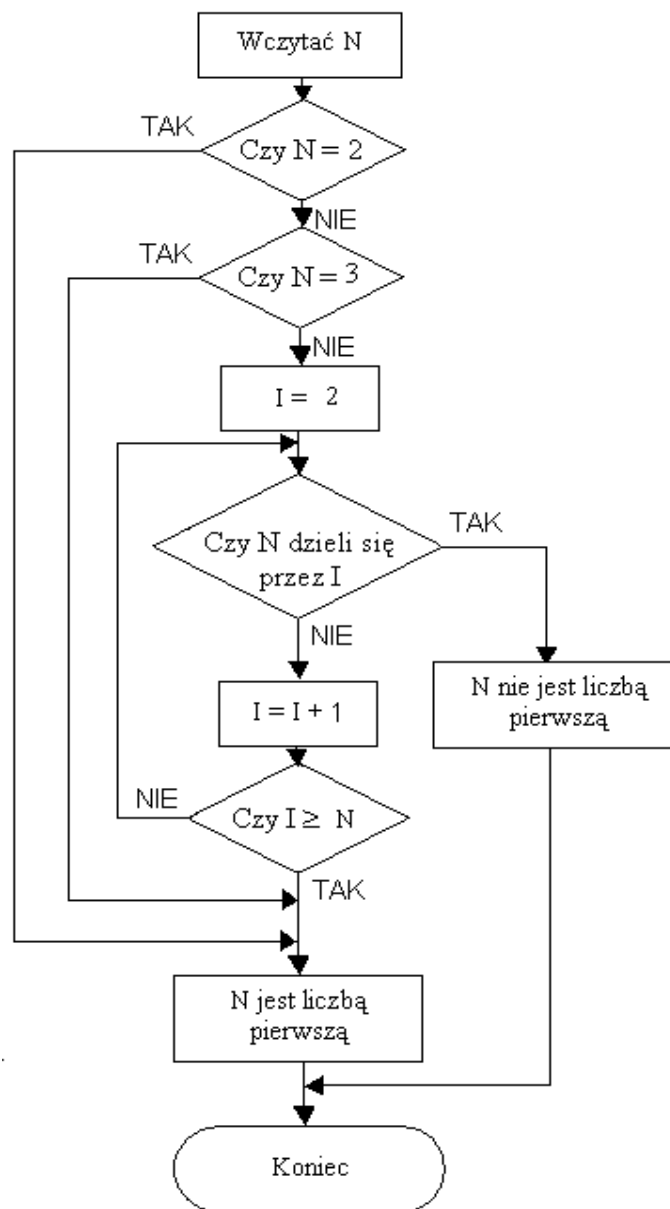
```

7.3. Liczby pierwsze

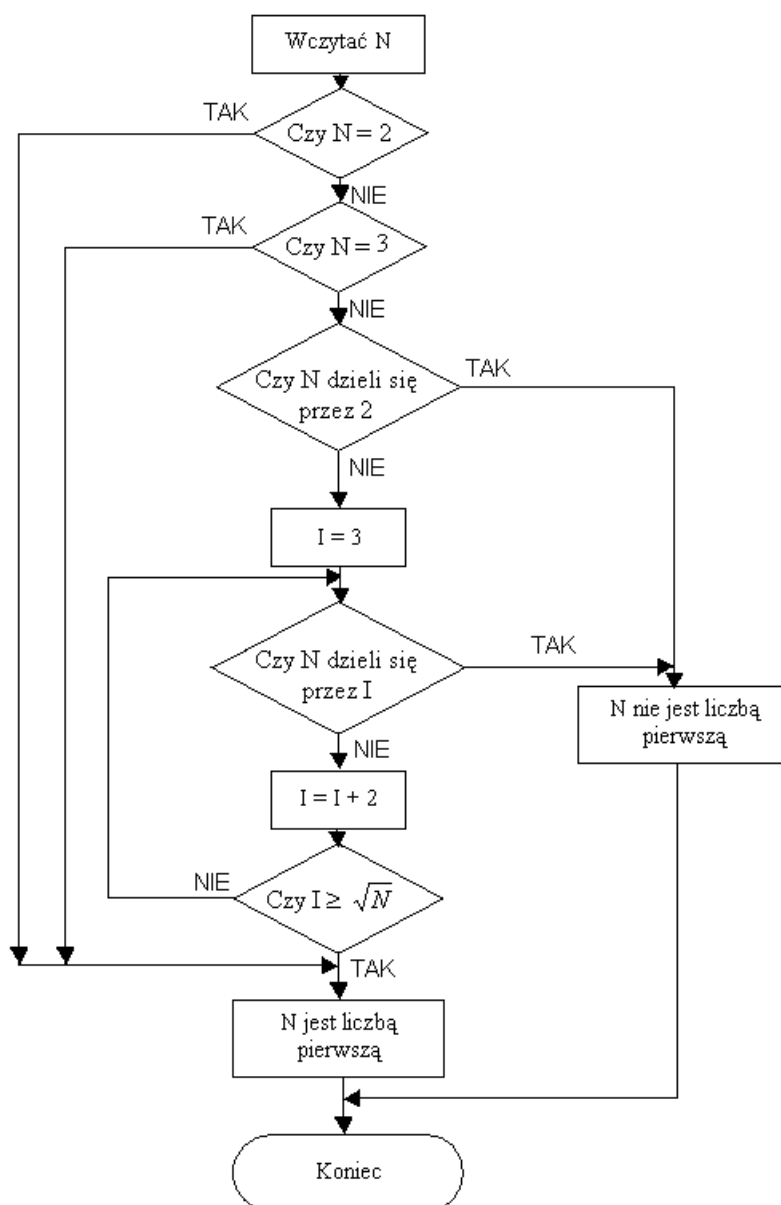
Liczba pierwsza, to taka liczba naturalna, która dzieli się tylko przez 1 i przez samą siebie. Np. liczby 3, 11, 29 to liczby pierwsze, a 8, 9, 10 nie są liczbami pierwszymi. Najprostszy (lecz wcale nie najlepszy) algorytm badania, czy dana liczba N jest liczbą pierwszą może mieć postać jak na rysunku 7.3 [5]:

Algorytm ten polega na sprawdzaniu, czy N dzieli się bez reszty przez kolejne liczby naturalne poczynając od 2 a kończąc na N .

Algorytm ten jest zbyt „rozrzutny”, wykonuje niepotrzebnie (wydłużając czas działania) zbyt wiele sprawdzeń. Można zauważyć [5], że jeśli liczba nie dzieli się przez 2, to nie będzie też dzieliła się przez następne liczby parzyste. Czyli po sprawdzeniu podzielności przez 2 należy sprawdzać dalej jedynie podzielność przez liczby nieparzyste. Sprawdzenie należy zakończyć nie w momencie gdy i osiągnie wartość N lecz \sqrt{N} . Uwagi te komplikują nieco algorytm lecz przyspieszają jego działanie dwudziestokrotnie, rysunek 7.4.

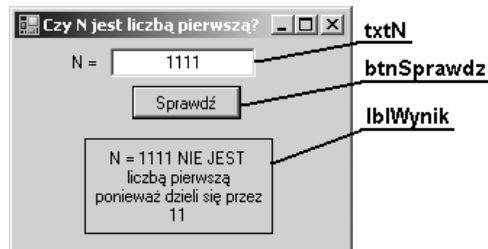


Rysunek 7.3. Najprostszy algorytm badania, czy dana liczba N jest liczbą pierwszą [5].
(Zmodyfikowany przez dodanie dwu sprawdzeń: Czy $N=2$ i Czy $N=3$)



Rysunek 7.4. Poprawiony algorytm sprawdzania, czy N jest liczbą pierwszą [5].
 (Zmodyfikowany przez dodanie dwu sprawdzeń: Czy N=2 i Czy N=3)

Przykład aplikacji w języku Visual Basic



Rysunek 7.5. Propozycja formularza

Aplikacja sprawdza, czy wprowadzona liczba N jest liczbą pierwszą.

Kod programu

```
Private Sub btnSprawdz_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnSprawdz.Click
    Dim N As Integer
    N = Integer.Parse(txtN.Text)
    lblWynik.Text = CzyPierwsza(N)
End Sub

Private Function CzyPierwsza(ByVal N) As String
    Dim info As String = ""
    Dim i As Integer
    If N = 2 Then
        info = "N = " & N.ToString & _
            " JEST liczbą pierwszą"
    ElseIf N = 3 Then
        info = "N = " & N.ToString & _
            " JEST liczbą pierwszą"
    Else
        If N Mod 2 = 0 Then
            info = "N = " & N.ToString & _
                " nie jest liczbą pierwszą"
        Else
            i = 3
            Do
                If N Mod i = 0 Then
                    info = "N = " & N.ToString & _
                        " NIE JEST liczbą pierwszą" & _
                        vbCrLf & _
                        "ponieważ dzieli się przez " & _
                        i.ToString
                    Return info
                End If
                i += 1
            Loop
        End If
    End If
    Return info
End Function
```

```
Exit Function
Else
    info = "N = " & N.ToString & _
        " JEST liczbą pierwszą"
End If
i = i + 2
Loop While i <= Math.Sqrt(N)
End If
End If
Return info
End Function
```

Generowanie liczb pierwszych w zadanym przedziale [2, N]

Zagadnieniem bardzo podobnym do sprawdzania, czy dana liczba jest liczbą pierwszą - jest generowanie liczb pierwszych w zadanym zakresie, na ogół od 2 do N.

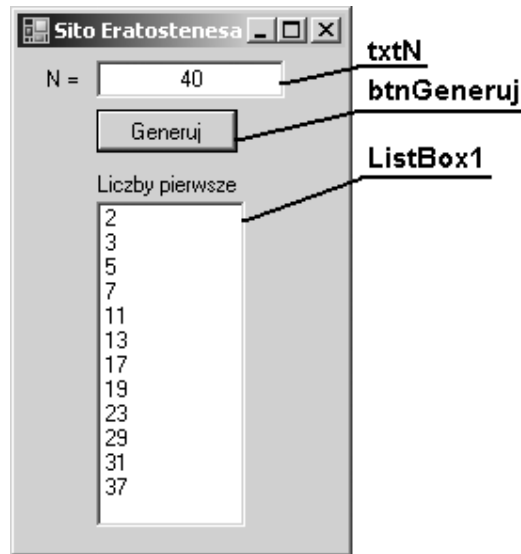
Do tego celu można wykorzystać funkcję utworzoną w aplikacji powyżej `CzyPierwsza(N)`, ale bardziej efektywny jest algorytm o nazwie *sito Erastotenesa*.

Opis algorytmu sito Erastotenesa wyznaczania liczb pierwszych z przedziału [2, N].

1. Ze zbioru liczb naturalnych od 2 do N usuwamy liczby podzielne przez $i = 2$ (zastępując je w Tablicy zerem).
2. Z nowo otrzymanego zbioru liczb usuwamy (zastępując je w Tablicy zerem) liczby podzielne przez $i = 3$ (czyli przez następną po 2 liczbę nieusuniętej ze zbioru), .
3. Z nowo otrzymanego zbioru liczb usuwamy (zastępując je w Tablicy zerem) liczby podzielne przez $i = 5$ (następną po 3 liczbę nieusuniętej ze zbioru).

Usuwanie kontynuujemy dla i od 2 do \sqrt{N} .

Przykład aplikacji w języku Visual Basic



Rysunek 7.6. Propozycja formularza

Kod programu

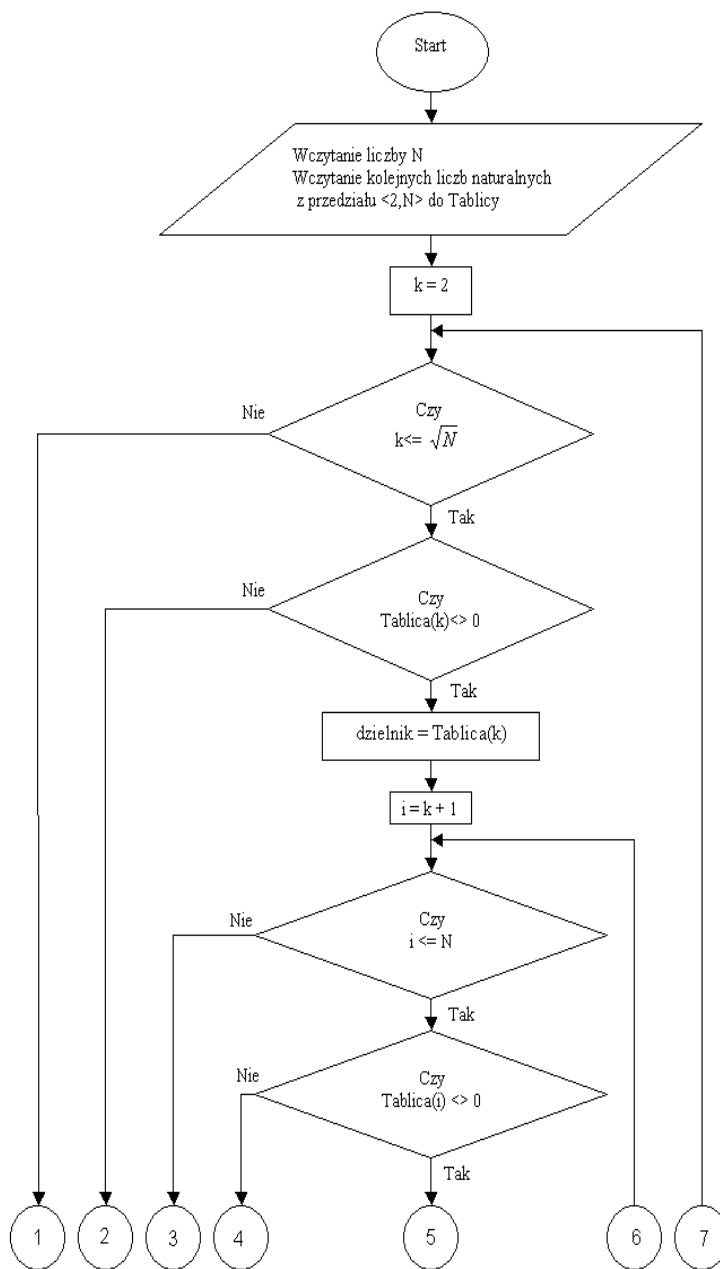
```

Private Sub btnGeneruj_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnGeneruj.Click
    Dim i, N As Integer
    N = Integer.Parse(txtN.Text)
    Dim Tablica(N) As Integer
    For i = 2 To N
        Tablica(i) = i
    Next
    Call LiczbyPierwsze(N, Tablica)
    ListBox1.Items.Clear()
    For i = 2 To N
        If Tablica(i) <> 0 Then
            ListBox1.Items.Add(Tablica(i))
        End If
    Next
End Sub

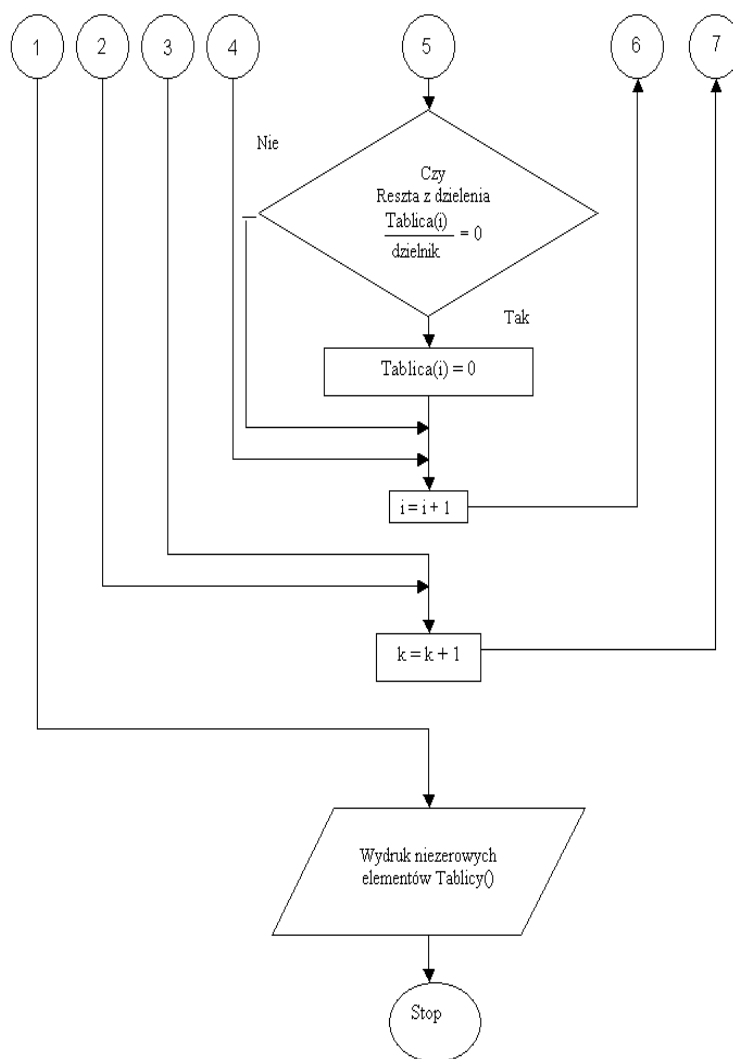
```

ROZDZIAŁ 7

```
Private Sub LiczbyPierwsze(ByVal N, ByRef Tablica)
    Dim i, k, dzielnik As Integer
    For k = 2 To Math.Sqrt(N)
        If Tablica(k) <> 0 Then
            dzielnik = Tablica(k)
        End If
        For i = (k + 1) To N
            If Tablica(i) <> 0 Then
                If Tablica(i) Mod dzielnik = 0 Then
                    Tablica(i) = 0
                End If
            End If
        Next
    Next
End Sub
```



Rysunek 7.7. Algorytm sita Erastotenesa, część 1



Rysunek 7.8. Algorytm sita Erastotenesa, część 2

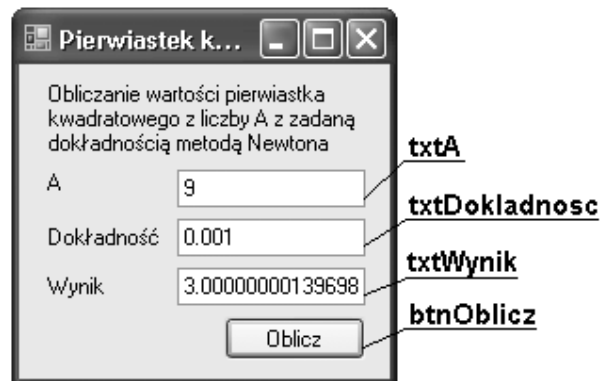
7.4. Obliczenie wartości pierwiastka kwadratowego metodą Newtona z zadaną dokładnością

Wartość pierwiastka kwadratowego z liczby A wyliczana jest z iteracyjnego wzoru:

$$x_n = x_{n-1} + \frac{1}{2} \cdot \left(\frac{A}{x_{n-1}} - x_{n-1} \right) \quad (7.2)$$

Danymi wejściowymi są: liczba A i wymagana dokładność oszacowania wartości pierwiastka. Algorytm porównuje obliczenia pierwiastka kwadratowego z dwóch kolejnych kroków i kończy pracę, gdy różnica kolejnych oszacowań jest mniejsza od założonej dokładności.

Przykład aplikacji w języku Visual Basic

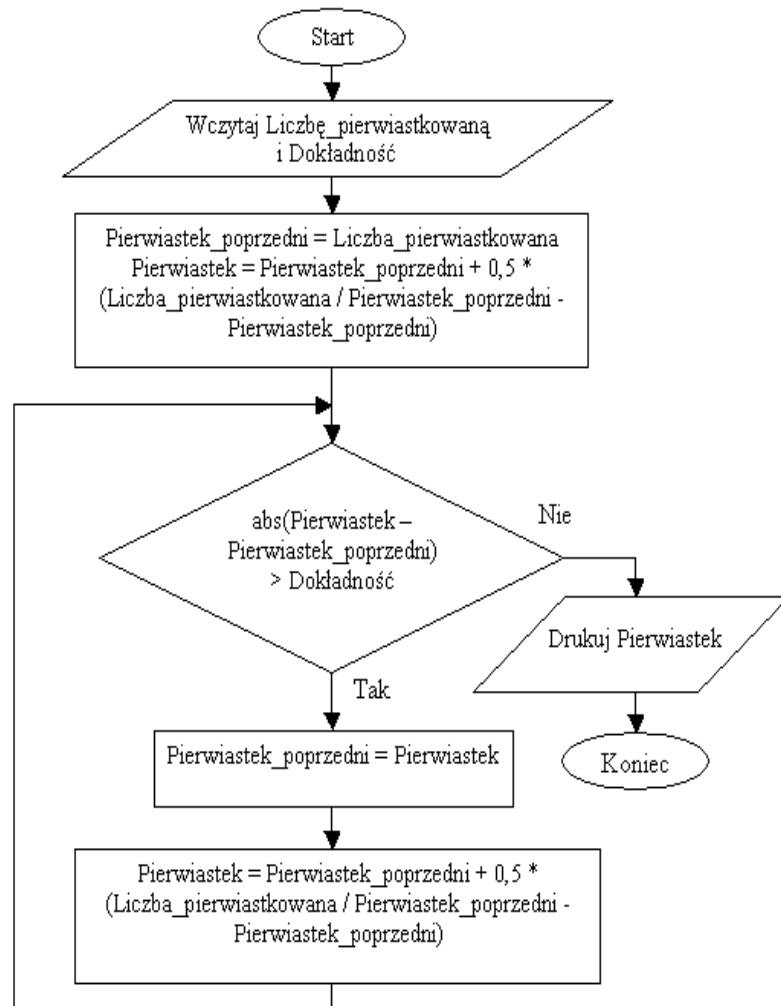


Rysunek 7.9

Kod programu

```
Private Sub btnOblicz_Click(ByVal sender As _  
    System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnOblicz.Click  
    Dim A, Dokladnosc As Double  
    A = CDb1(txtA.Text)  
    Dokladnosc = CDb1(txtDokladnosc.Text)  
    txtWynik.Text = CStr(PierwKwadr(A, Dokladnosc))  
End Sub
```

```
Private Function PierwKwadr(ByVal LiczbaA, _  
    ByVal Dokladnosc) _  
    As Double  
    Dim PierwKwadr_p As Double  
    PierwKwadr_p = LiczbaA  
    PierwKwadr = PierwKwadr_p + 0.5 * _  
        (LiczbaA / PierwKwadr_p - _  
        PierwKwadr_p)  
    Do While Math.Abs(PierwKwadr - PierwKwadr_p) > _  
        Dokladnosc  
        PierwKwadr_p = PierwKwadr  
        PierwKwadr = PierwKwadr_p + 0.5 * _  
            (LiczbaA / PierwKwadr_p - _  
            PierwKwadr_p)  
    Loop  
End Function
```



Rysunek 7.10. Schemat algorytmu

7.5. Znajdowania pierwiastków równania (kwadratowego) drugiego stopnia

Równanie drugiego stopnia ma postać:

$$A \cdot x^2 + B \cdot x + C = 0 \quad (7.3)$$

Obliczanie pierwiastków równania w dziedzinie liczb rzeczywistych rozpoczynamy obliczając wyróżnik, bardzo często oznaczany grecką literą delta Δ

$$\Delta = B^2 - 4 \cdot A \cdot C \quad (7.4)$$

Jeśli $\Delta > 0$ równanie ma dwa pierwiastki

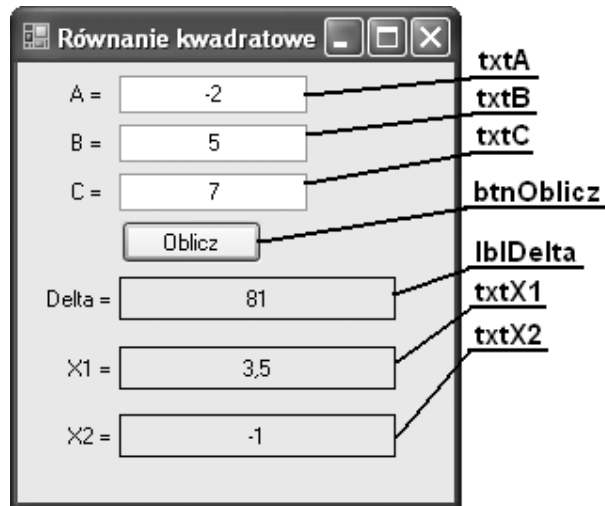
$$x_1 = \frac{-B - \sqrt{\Delta}}{2 \cdot A}; \quad x_2 = \frac{-B + \sqrt{\Delta}}{2 \cdot A} \quad (7.5)$$

Jeśli $\Delta = 0$ równanie ma pierwiastek podwójny

$$x_1 = x_2 = -\frac{B}{2 \cdot A} \quad (7.6)$$

Jeśli $\Delta < 0$ równanie nie ma pierwiastków w dziedzinie liczb rzeczywistych.

Przykład aplikacji w języku Visual Basic



Rysunek 7.11. Propozycja formularza

Kod programu

```

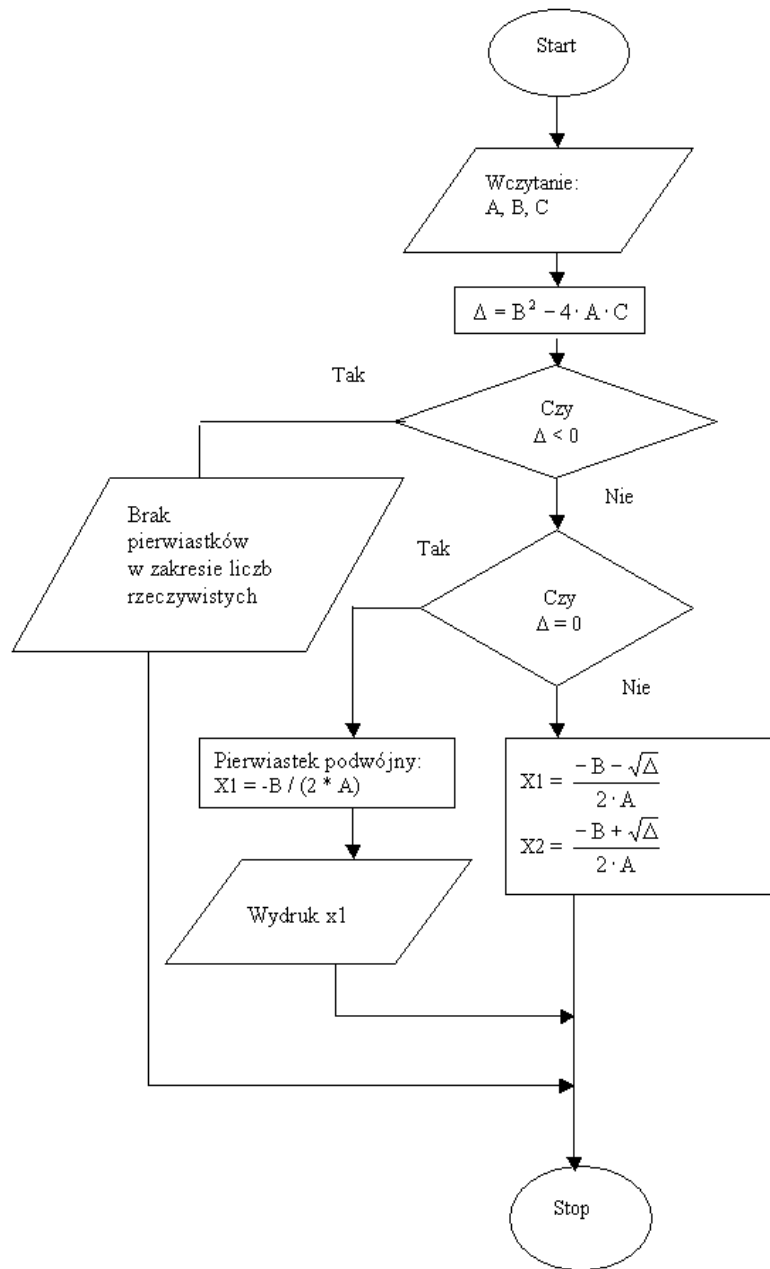
Private Sub btnOblicz_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnOblicz.Click
    Dim A, B, C, Delta, X1, X2 As Double
    Dim Info As String = ""
    A = Double.Parse(txtA.Text)
    B = Double.Parse(txtB.Text)
    C = Double.Parse(txtC.Text)
    lblDelta.Text = ""
    lblX1.Text = ""
    lblX2.Text = ""
    Call PierwiaskiRownania _
        (A, B, C, Delta, X1, X2, Info)
    lblDelta.Text = Delta.ToString
    If Info <> "" Then
        MessageBox.Show(Info, "Uwaga, Delta ujemna", _
            MessageBoxButtons.OK, _
            MessageBoxIcon.Information)
    Else
        lblX1.Text = X1.ToString
        lblX2.Text = X2.ToString
    End If
End Sub

```

ROZDZIAŁ 7

```
Private Sub PierwiaskiRownania(ByVal A, ByVal B, _  
    ByVal C, ByRef Delta, _  
    ByRef X1, ByRef X2, ByRef Info)  
    Delta = B ^ 2 - 4 * A * C  
    If Delta < 0 Then  
        Info = "Delta ujemna" & vbCrLf & _  
            "Brak pieriastków"  
    ElseIf Delta = 0 Then  
        X1 = -B / (2 * A)  
        X2 = X1  
    Else  
        X1 = (-B - Math.Sqrt(Delta)) / (2 * A)  
        X2 = (-B + Math.Sqrt(Delta)) / (2 * A)  
    End If  
End Sub
```

Można dodać do aplikacji sprawdzenie, jeszcze przed wywołaniem procedury obliczającej pierwiastki, czy wartość współczynnika A nie jest zerowa. W takim przypadku równanie nie jest równaniem drugiego stopnia i obliczenia należy przerwać, ewentualnie informując użytkownika o niepełnych danych.

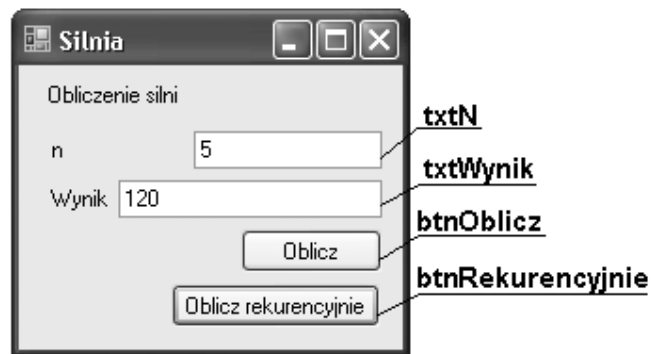


Rysunek 7.12. Znajdowanie pierwiastków równania drugiego stopnia

7.6. Obliczenie silni

Przedstawiono dwa algorytmy wyliczające silnię. W funkcji *Silnia* wykorzystano pętlę For... Next. W funkcji *SilniaRekurencyjnie* zastosowano rekurencyjne wywoływanie funkcji. W obu przypadkach wymagane jest początkowe przypisanie wartości równej 1 zmiennej wynikowej. Daną wejściową jest wartość n.

Przykład aplikacji w języku Visual Basic



Rysunek 7.13

Kod programu

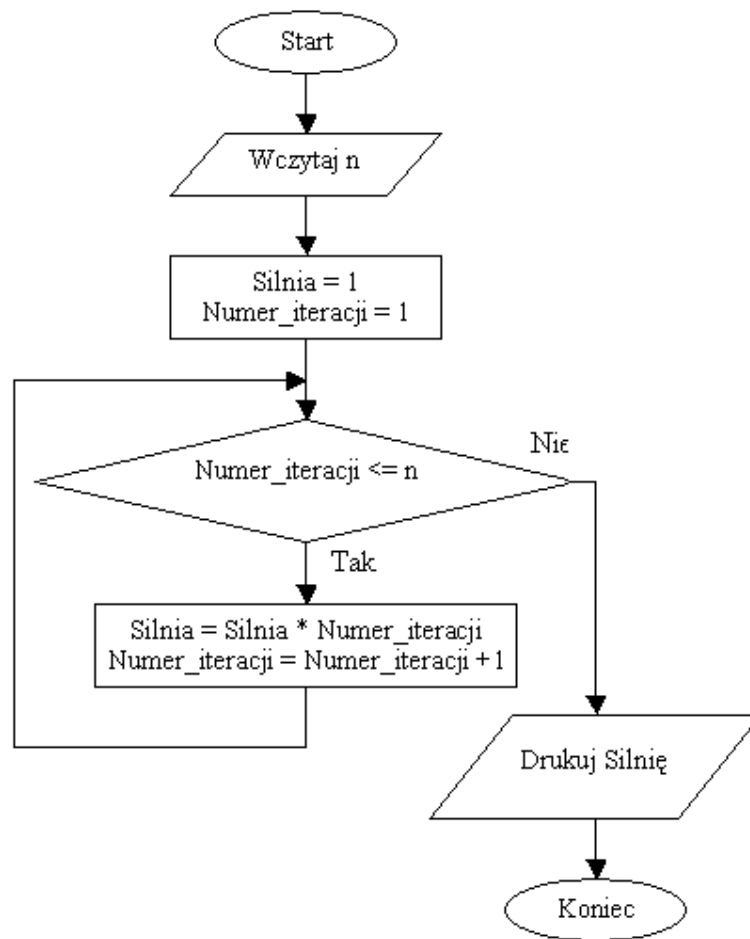
```
Private Sub btnOblicz_Click(ByVal sender As _
    System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnOblicz.Click
    Dim n As Integer
    n = CInt(txtN.Text)
    txtWynik.Text = CStr(Silnia(n))
End Sub
```

```
Private Sub btnRekurencyjnie_Click(ByVal sender As _
    System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnRekurencyjnie.Click
    Dim n As Integer
    n = CInt(txtN.Text)
    txtWynik.Text = CStr(SilniaRekurencyjnie(n))
End Sub
```

```
Private Function Silnia(ByVal n As Integer) As Double
    Dim i As Integer
    Silnia = 1
    For i = 1 To n
        Silnia = Silnia * i
    Next
End Function


---


Private Function SilniaRekurencyjnie(ByVal i As
Integer) As Double
    SilniaRekurencyjnie = 1
    If i > 0 Then
        SilniaRekurencyjnie = _
            SilniaRekurencyjnie(i - 1) * i
    End If
End Function
```



Rysunek 7.14. Schemat algorytmu

7.7. Algorytm Euklidesa znajdowania największego wspólnego dzielnika

Algorytm Euklidesa pozwala na znajdowania największego wspólnego dzielnika (NWD) dwóch liczb naturalnych. Algorytm powstał około IV wieku p.n.e., opracowany został przez Eudoksosa z Knidos i opublikowany przez Euklidesa w jego słynnym dziele Elementy, księga VII [7].

„Szkolny” sposób znajdowania NWD dwóch liczb naturalnych polega na ich rozłożeniu na czynniki i wymnożeniu przez siebie tych czynników, które powtarzają się w obu liczbach: Wykonajmy go dla liczb 60 i 24:

$$\begin{array}{r|l}
 60 & 2 \\
 30 & 2 \\
 15 & 3 \\
 5 & 5 \\
 1 &
 \end{array}
 \qquad
 \begin{array}{r|l}
 24 & 2 \\
 12 & 2 \\
 6 & 2 \\
 3 & 3 \\
 1 &
 \end{array}$$

Ponieważ dla obu liczb czynniki występujące jednocześnie to: 2, 2 i 3 zatem

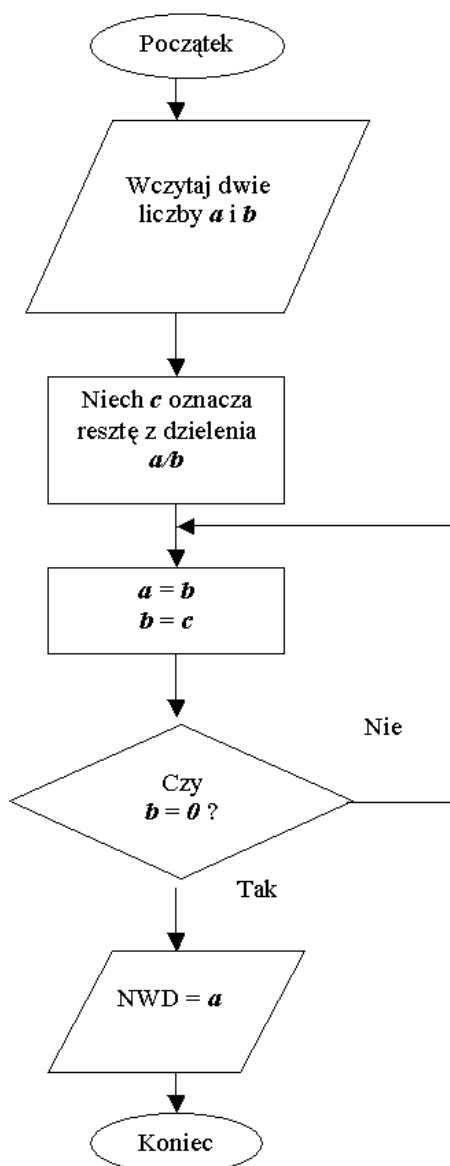
$$\text{NWD} = 2 * 2 * 3 = 12$$

Algorytm Euklidesa umożliwia znalezienie NWD bez rozkładania liczb na czynniki. Opis algorytmu:

Niech będą dane dwie liczby naturalne a i b , (gdzie $a > b$) których NWD poszukujemy.

1. Podziel a przez b . Niech c zawiera resztę z tego dzielenia.
2. Pod a podstaw b , a pod b podstaw c .
3. Jeśli b nie równa się 0 idź do 1.
4. Jeśli $b = 0$, to $\text{NWD} = a$. Koniec algorytmu

Postać skrzynkową algorytmu Euklidesa przedstawia rysunek 7.9.



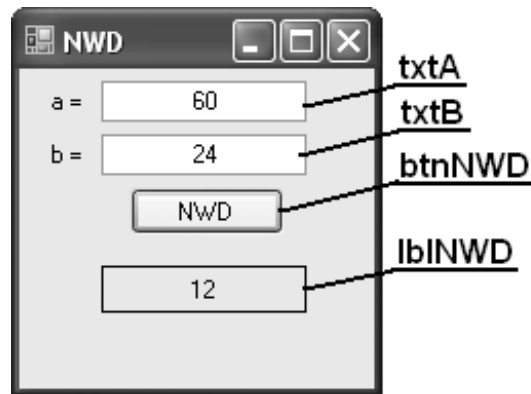
Rysunek 7.15. Schemat blokowy algorytmu Euklidesa

Znajdowanie NWD dla liczb 60 i 24 - kolejność działań będzie następująca:

1. $a = 60, b = 24$

2. $a/b = 60/24 = 2$ reszta, czyli $c = 12$
3. $a = 24, b = c$, czyli $b = 12$
4. czy $b = 0$? nie – a zatem kontynuacja działań
5. $a/b = 24/12 = 2$ reszta, czyli $c = 0$
6. $a = 12, b = c$, czyli $b = 12$
7. czy $b = 0$? tak, zatem $NWD = a$, czyli $NWD = 12$.

Przykład aplikacji w języku Visual



Rysunek 7.16. Postać formularza

Kod programu

```

Private Sub btnNWD_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnNWD.Click
    Dim a, b, c, wynik As Integer
    a = Integer.Parse(txtA.Text)
    b = Integer.Parse(txtB.Text)
    Call NWD(a, b, wynik)
    lblNWD.Text = wynik.ToString
End Sub

```

```

Private Sub NWD(ByRef a, ByRef b, ByRef wynik)
    Dim c As Integer
    c = a Mod b
    a = b
    b = c

```

```
    If b <> 0 Then
        Call NWD(a, b, wynik)
    Else
        wynik = a
    End If
End Sub
```

7.8. Szyfrowanie i rozszyfrowywanie tekstu przy wykorzystaniu „Kodu Cezara”

Tekst kodowany jest przy pomocy metody „Kod Cezara”. Metoda wymaga podania słowa szyfrującego (Kod), które musi być znane również przy operacji rozkodowywania. Dla uproszczenia w przykładzie pokazano wyłącznie 26 dużych liter alfabetu angielskiego: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z. Litery te zajmują kolejne pozycje w kodzie ASCII (A-65, B-66, ...).

Kodowanie

Metoda polega na dodaniu numeru pozycji tekstu kodowanego i numeru pozycji tekstu kodu. Jeśli otrzymana suma przekracza ilość dostępnych znaków (26), wówczas od sumy odejmowana jest 26. Otrzymane wartości sum określają tekst zakodowany. Danymi wejściowymi przy kodowaniu są: tekst do zakodowania i tekst kodu szyfrującego.

Przykład

Tekst	I	N	F	O	R	M	A	C	J	A
Pozycja	9	14	6	15	18	13	1	3	10	1
Kod	S	Z	Y	F	R	S	Z	Y	F	R
Pozycja	19	26	25	6	18	19	26	25	6	18
Sumy	28	40	31	21	36	32	27	28	16	19
Po odjęciu 26	2	14	5	21	10	6	1	2	16	19
Zakodowany	B	N	E	U	J	F	A	B	P	S

Rozkodowywanie

Przy rozkodowywaniu należy od numeru pozycji tekstu zakodowanego odjąć numer pozycji tekstu kodu. Jeśli otrzymana wartość jest mniejsza od 1, wówczas należy dodać do niej 26. Danymi wejściowymi przy rozkodowywaniu są: tekst zakodowany i tekst kodu szyfrującego.

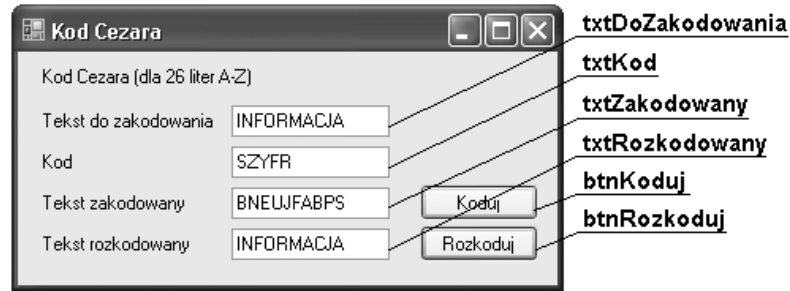
Przykład

Zakodowany	B	N	E	U	J	F	A	B	P	S
Pozycja	2	14	5	21	10	6	1	2	16	19
Kod	S	Z	Y	F	R	S	Z	Y	F	R
Pozycja	19	26	25	6	18	19	26	25	6	18
Różnica	-17	-12	-20	15	-8	-13	-25	-23	10	1
Po dodaniu	9	14	6	15	18	13	1	3	10	1
Rozkodow.	I	N	F	O	R	M	A	C	J	A

W programie zastosowano funkcje operacji na ciągach znakowych:

- *Len*(ciąg_znakowy) – funkcja zwraca długość ciągu znakowego
- *Mid*(ciąg_znakowy, pozycja_startowa, długość_ciagu) – funkcja wycina z ciągu znakowego podciąg zaczynając od pozycji startowej o określonej w trzecim parametrze długości.
- *UCase*(ciąg_znakowy) – funkcja zamienia wszystkie litery na duże
- *Asc*(znak) – funkcja podaje kod ASCII znaku
- *Chr*(liczba) – funkcja zwraca znak odpowiadający kodowi ASCII dla podanej liczby

Przykład aplikacji w języku Visual Basic



Rysunek 7.17

Kod programu

```

Private Sub btnKoduj_Click(ByVal sender As _
    System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnKoduj.Click
    Dim TekstDo, Kod As String
    TekstDo = UCase(txtDoZakodowania.Text)
    Kod = UCase(txtKod.Text)
    txtZakodowany.Text = Tekst_zakodowany(TekstDo, _
        Kod)
End Sub

```

```

Private Sub btnRozkoduj_Click(ByVal sender As _
    System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnRozkoduj.Click
    Dim Kod, Zakodowany As String
    Zakodowany = UCase(txtZakodowany.Text)
    Kod = UCase(txtKod.Text)
    txtRozkodowany.Text = _
        Tekst_rozkodowany(Zakodowany, Kod)
End Sub

```

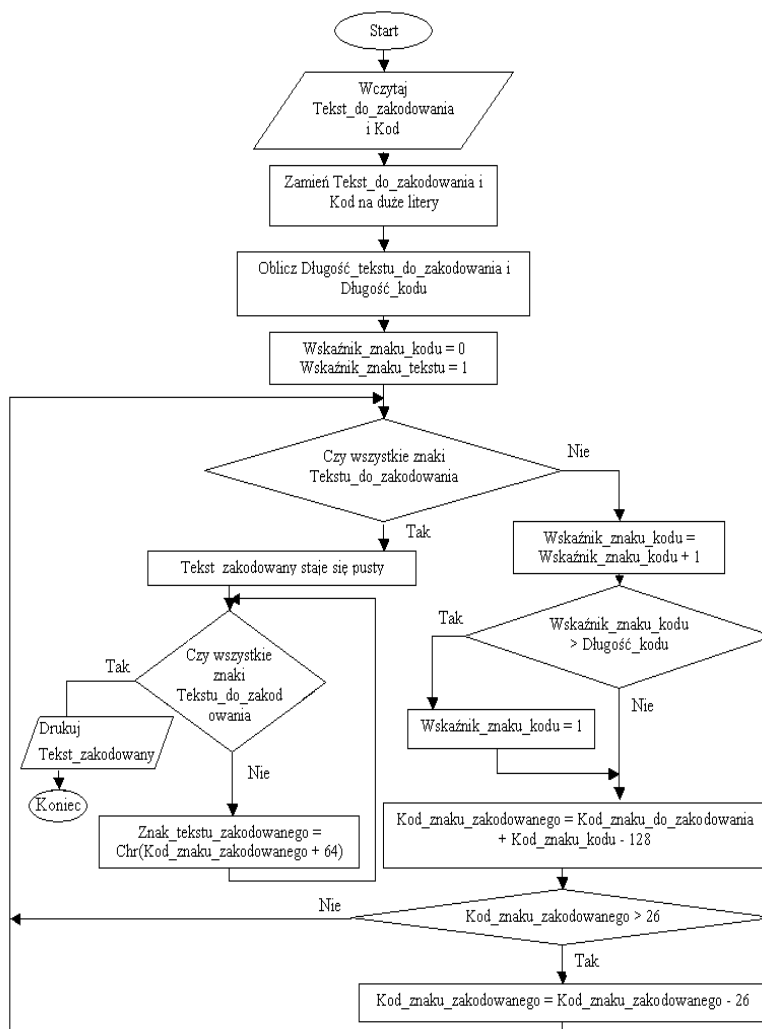
```

Private Function Tekst_zakodowany(ByVal TekstDo _
    As String, _
    ByVal Kod As String) As String
    Dim i, ik, Dlugosc_Do, Dlugosc_Kod, _
        Tab_Zakodowany(100) As Integer
    Dlugosc_Do = Len(TekstDo)
    Dlugosc_Kod = Len(Kod)
    ik = 0
    For i = 1 To Dlugosc_Do
        ik = ik + 1

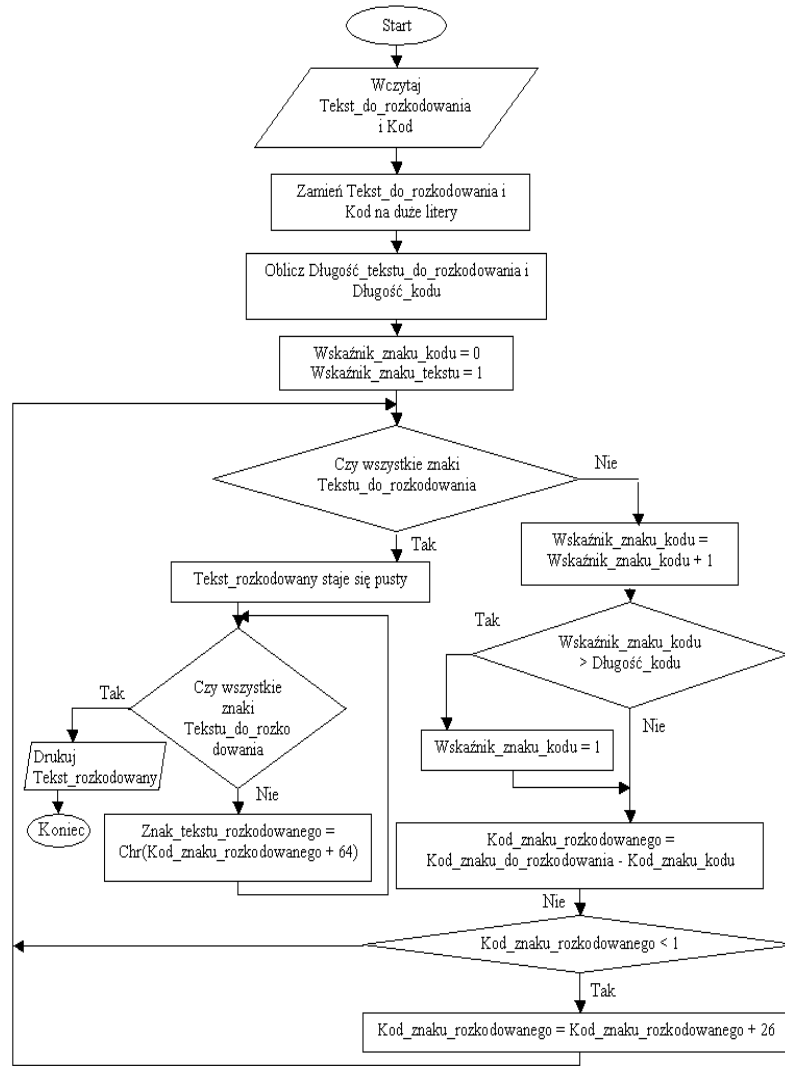
```

```
    If ik > Dlugosc_Kod Then
        ik = 1
    End If
    Tab_Zakodowany(i) = _
        Asc(Mid(TekstDo, i, 1)) + _
        Asc(Mid(Kod, ik, 1)) - 128
    If Tab_Zakodowany(i) > 26 Then
        Tab_Zakodowany(i) = _
            Tab_Zakodowany(i) - 26
    End If
Next
Tekst_zakodowany = ""
For i = 1 To Dlugosc_Do
    Tekst_zakodowany = Tekst_zakodowany & _
        Chr(Tab_Zakodowany(i) + 64)
Next
End Function
```

```
Private Function Tekst_rozkodowany(ByVal _
    Zakodowany As String, _
    ByVal Kod As String) As String
    Dim i, ik, Dlugosc_Zakodowany, Dlugosc_Kod, _
        Tab_Rozkodowany(100) As Integer
    Zakodowany = UCase(txtZakodowany.Text)
    Kod = UCase(txtKod.Text)
    Dlugosc_Zakodowany = Len(Zakodowany)
    Dlugosc_Kod = Len(Kod)
    ik = 0
    For i = 1 To Dlugosc_Zakodowany
        ik = ik + 1
        If ik > Dlugosc_Kod Then
            ik = 1
        End If
        Tab_Rozkodowany(i) = Asc(Mid(Zakodowany, _
            i, 1)) - Asc(Mid(Kod, ik, 1))
        If Tab_Rozkodowany(i) < 1 Then
            Tab_Rozkodowany(i) = _
                Tab_Rozkodowany(i) + 26
        End If
    Next
    Tekst_rozkodowany = ""
    For i = 1 To Dlugosc_Zakodowany
        Tekst_rozkodowany = Tekst_rozkodowany & _
            Chr(Tab_Rozkodowany(i) + 64)
    Next
End Function
```



Rysunek 7.18. Kod Cezara, kodowanie



Rysunek 7.19. Schemat algorytmu Kod Cezara, rozkodowanie



Algorytmy numeryczne

8.1. Wprowadzenie

W rozdziale przedstawiono algorytmy, które funkcjonują iteracyjnie i pozwalają numerycznie z określoną dokładnością rozwiązać wybrane problemy matematyczne.

Pierwszy algorytm ilustruje w jaki sposób można rozwiązywać numerycznie zagadnienia obliczania całki oznaczonej. Pokazano pięć przykładów algorytmów.

Kolejne dwa przykłady dotyczą znajdowania numerycznego pierwiastków funkcji nieliniowej. Ostatni przykład przedstawia zagadnienie aproksymacji funkcji.

8.2. Metoda Monte Carlo – obliczenie całki oznaczonej

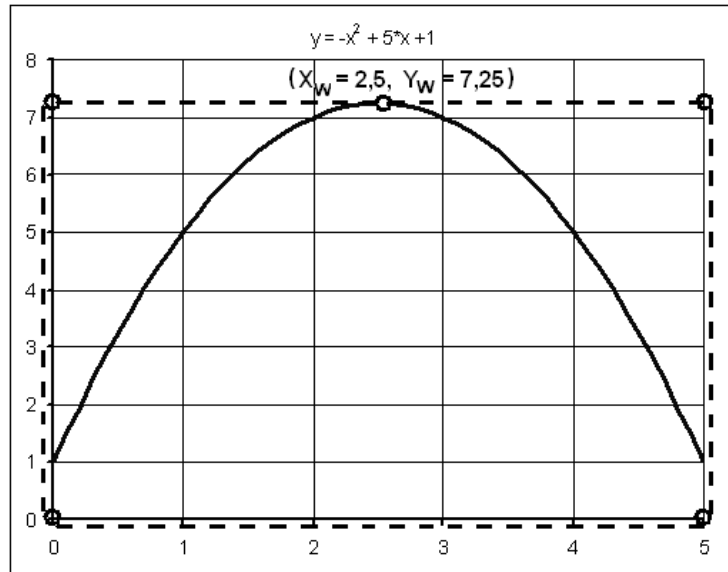
Obliczono, metodą Monte Carlo, całkę oznaczoną funkcji $y = f(x)$, tzn.

$$S = \int_a^b (A \cdot x^2 + B \cdot x + C) \cdot dx = \int_0^5 (-x^2 + 5 \cdot x + 1) \cdot dx \quad (8.1)$$

Wybrano funkcję na tyle prostą, aby łatwo można było tę całkę obliczyć metodą algebraiczną:

$$S = \int_0^5 (-x^2 + 5 \cdot x + 1) = \left(-\frac{x^3}{3} + \frac{5}{2} \cdot x^2 + x \right) \Big|_0^5 = \frac{155}{6} = 25,833 \quad (8.2)$$

Przedstawiono zagadnienie graficznie, rysunek 8.1



Rysunek 8.1. Obliczenia pola pod krzywą

Obliczyć całkę oznaczoną, to obliczyć pole pod krzywą w zadanym przedziale całkowania.

Współrzędne wierzchołka paraboli wynoszą:

$$X_w = \frac{-B}{2 \cdot A}, \quad Y_w = \frac{-\Delta}{4 \cdot A} \quad (8.3)$$

gdzie $\Delta = B^2 - 4 \cdot A \cdot C$

stąd: $X_w = 2,5$, a $Y_w = 7,25$,

Pole prostokąta, oznaczonego na rysunku 8.5 można zatem obliczyć i wynosi ono $P_p = 5 * 7,25 = 36,25$.

Algorytm jaki zastosujemy będzie następujący:

1. Wygenerujemy liczbę przypadkową z przedziału 0 – 5 i będzie to X_i .
2. Zwiększymy licznik ogólny o 1.
3. Dla X_i obliczymy punkt leżący na paraboli $Y_i = f(X_i)$

4. Wygenerujemy liczbę przypadkową z przedziału 0 – 7,25 i będzie to Y_{los} .
5. Sprawdzimy czy $Y_{los} \leq Y_i$, tzn, czy wygenerowany punkt leży pod (i na) krzywej, czy też leży nad krzywą.
6. Jeśli punkt Y_{los} leży na krzywej lub pod krzywą, zwiększymy licznik trafień o 1.

Istnieje zależność, że

$$(\text{Pole pod krzywą})/(\text{Pole prostokąta}) = (\text{Licznik trafień})/(\text{Licznik ogólny})$$

stąd obliczymy całkę:

$$\text{Pole pod krzywą} = (\text{Pole prostokąta}) * (\text{Licznik trafień})/(\text{Licznik ogólny})$$

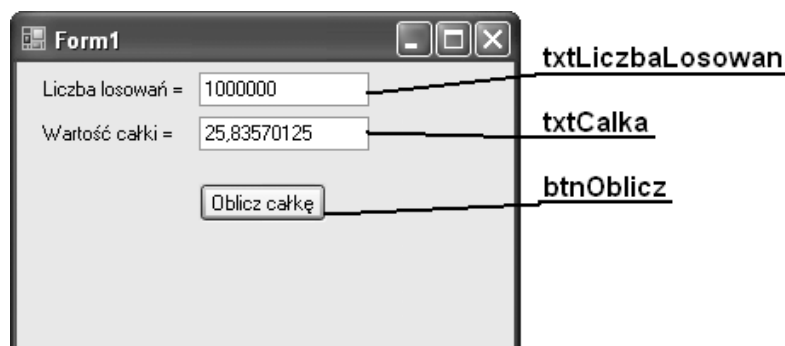
Błąd metody jest proporcjonalny do $\frac{1}{\sqrt{n}}$, gdzie n = liczba prób [3]

(czyli Licznik ogólny)

Należy zatem przewidzieć liczbę losowań i ustalać ją odpowiednio dużą. Ponieważ LiczbaLosowan zadeklarowano jako Integer liczba losowań nie może być większa niż 2 147 483 647 (około dwa miliardy). Przyjęcie zbyt dużej liczby prób może znacznie wydłużyć czas oczekiwania na wynik.

W powyższym przypadku wyjaśnienia trwały dłużej niż analityczne obliczenie całki, ale tak jest tylko wtedy, gdy równanie krzywej jest znane i jest bardzo proste. Przewagą metody Monte Carlo jest to, że możemy ją stosować także wtedy, gdy krzywa jest nieregularna, czy też całka nie daje się obliczyć z innych powodów.

Przykład aplikacji w języku Visual Basic



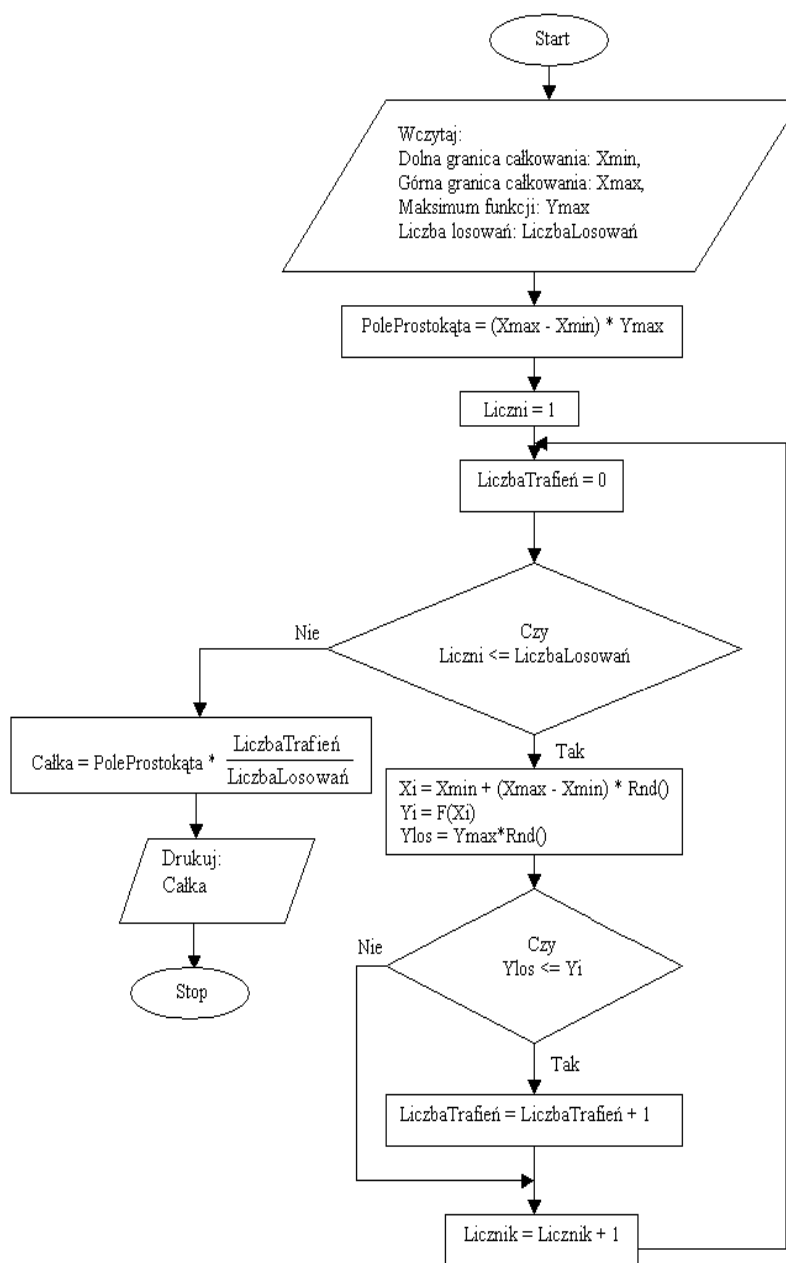
Rysunek 8.2. Postać formularza

Kod programu

```

Private Sub Form1_Load(ByVal sender As _
                        System.Object, ByVal e As _
                        System.EventArgs) Handles _
                        MyBase.Load
    Randomize ()
End Sub
-----
Private Sub btnOblicz_Click(ByVal sender As _
                            System.Object, ByVal e As _
                            System.EventArgs) Handles btnOblicz.Click
    Dim LiczbaLosowan As Integer
    Dim LiczbaTrafien As Integer
    Dim PoleProstokata, Calka As Double
    Dim Xi, Ylos, Yi As Double
    LiczbaLosowan = _
        Integer.Parse(txtLiczbaLosowan.Text)
    PoleProstokata = 5 * 7.25
    For i = 1 To LiczbaLosowan
        Xi = 5 * Rnd()
        Yi = -Xi ^ 2 + 5 * Xi + 1
        Ylos = 7.25 * Rnd()
        If Ylos <= Yi Then
            LiczbaTrafien = LiczbaTrafien + 1
        End If
    Next
    Calka = PoleProstokata * LiczbaTrafien _
        / LiczbaLosowan
    txtCalka.Text = Calka.ToString
End Sub

```



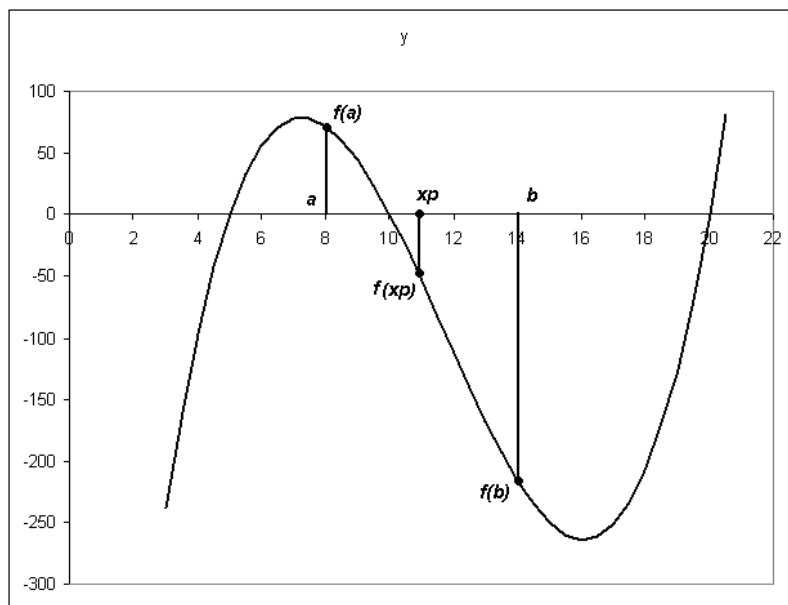
Rysunek 8.3. Schemat algorytmu

8.3. Metoda *bisekcji* (połowienia przedziału) znajdowania pierwiastków algebraicznego równania nieliniowego

Przybliżone rozwiązywanie równania algebraicznego nieliniowego w metodzie połowienia przedziału. Jeśli przyjmiemy dokładność obliczeń za eps, odbywa się według następującego algorytmu:

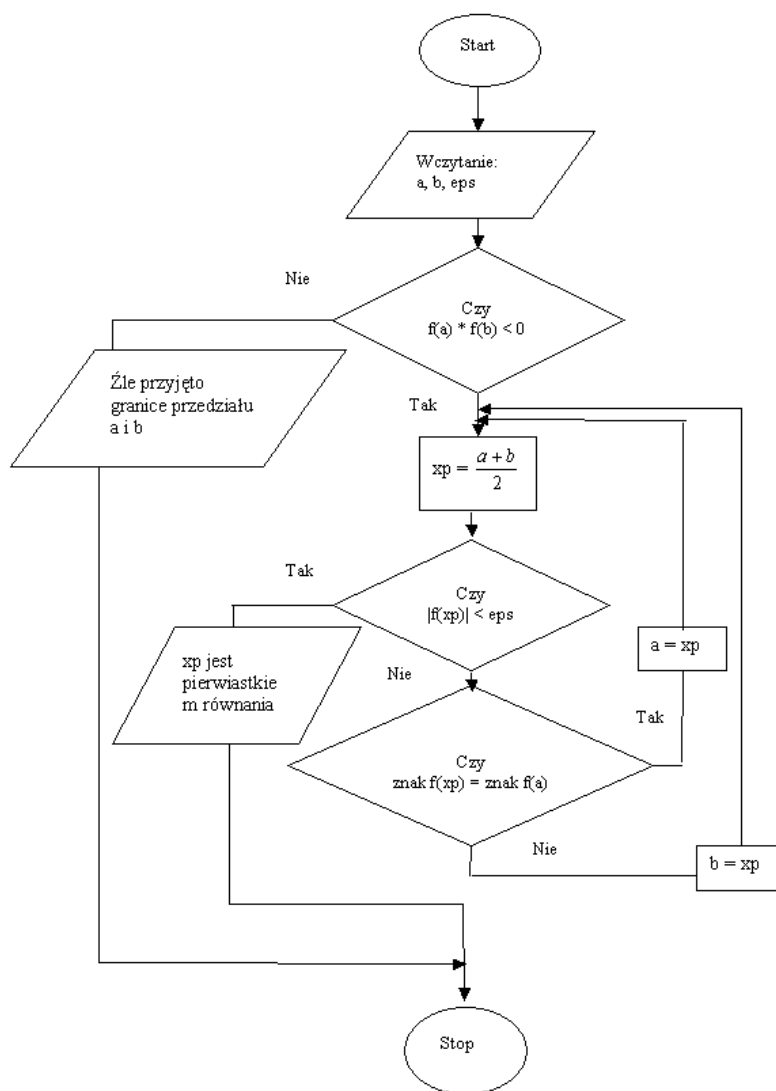
1. Przyjąć (arbitralnie) przedział $[a, b]$ i dokładność eps
2. Sprawdzamy, czy $f(a)*f(b) < 0$.
3. Odpowiedź TAK oznacza, że dobrze przyjęliśmy granice przedziału i pierwiastek równania znajduje się wewnątrz przedziału i można kontynuować działania.
4. Odpowiedź NIE oznacza, że granice przedziału zostały przyjęte błędnie i należy przerwać obliczenia.
5. Obliczyć (dzieląc przedział na połowę – stąd nazwa „metoda połowienia przedziału”) wartość wyrażenia

$$xp = \frac{a + b}{2}$$
6. Jeżeli $|f(xp)| < \text{eps}$ zakończenie programu, xp jest wartością pierwiastka.
7. Jeśli $|f(xp)| \geq \text{eps}$ należy sprawdzić:
 - 7.1. Czy znak $f(xp)$ jest taki sam jak znak $f(a)$
 - 7.2. Jeśli TAK to pierwiastek znajduje się w przedziale $[xp, b]$, czyli $a = xp$ i dalsze obliczenia od p-tu 5.
 - 7.3. Jeśli NIE, to pierwiastek znajduje się wewnątrz przedziału $[a, xp]$, czyli $b = xp$ i dalsze obliczenia od p-tu 5.



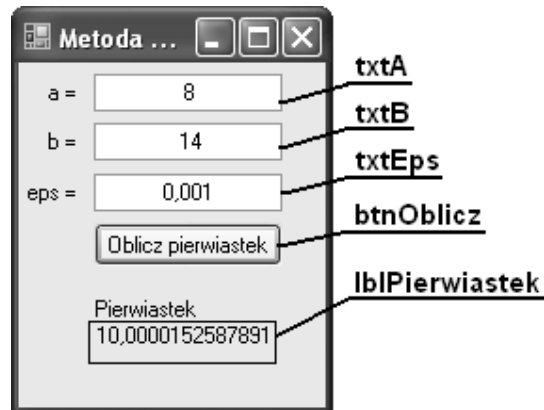
Rysunek 8.4. Graficzna ilustracja metody połowienia przedziału

Funkcja $y(x) = A \cdot x^3 + B \cdot x^2 + C \cdot x + D$, dla $A = 1$, $B = -35$, $C = 350$, $D = -1000$ posiada trzy pierwiastki $x_1 = 5$, $x_2 = 10$ i $x_3 = 20$. Dla przedziału $[8, 14]$ i dokładności $\text{eps} = 0,001$ program oblicza pierwiastek $x_p = 10,000015$, ale wystarczy przyjąć przedział $[8, 12]$ i pierwiastek zostanie wyliczony na $x_p = 10$.



Rysunek 8.5. Algorytm metody połowienia przedziału

Przykład aplikacji w języku Visual Basic



Rysunek 8.6. Propozycja formularza

Kod programu

```

Private Sub btnOblicz_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnOblicz.Click
    Dim a, b, xp, eps As Double
    Dim info As String = ""
    a = Double.Parse(txtA.Text)
    b = Double.Parse(txtB.Text)
    eps = Double.Parse(txtEps.Text)
    Call Pierwiastek(a, b, xp, eps, info)
    If info = "OK" Then
        lblPierwiastek.Text = xp
    Else
        lblPierwiastek.Text = info
    End If
End Sub

```

```

Private Sub Pierwiastek(ByVal a, ByVal b, ByRef xp, _
    ByVal eps, ByRef info)
    If F(a) * F(b) < 0 Then
        xp = (a + b) / 2
        If Math.Abs(F(xp)) < eps Then
            info = "OK"
            Exit Sub
        Else
            If Math.Sign(F(xp)) = Math.Sign(F(a)) Then
                a = xp
            Else

```

ROZDZIAŁ 8

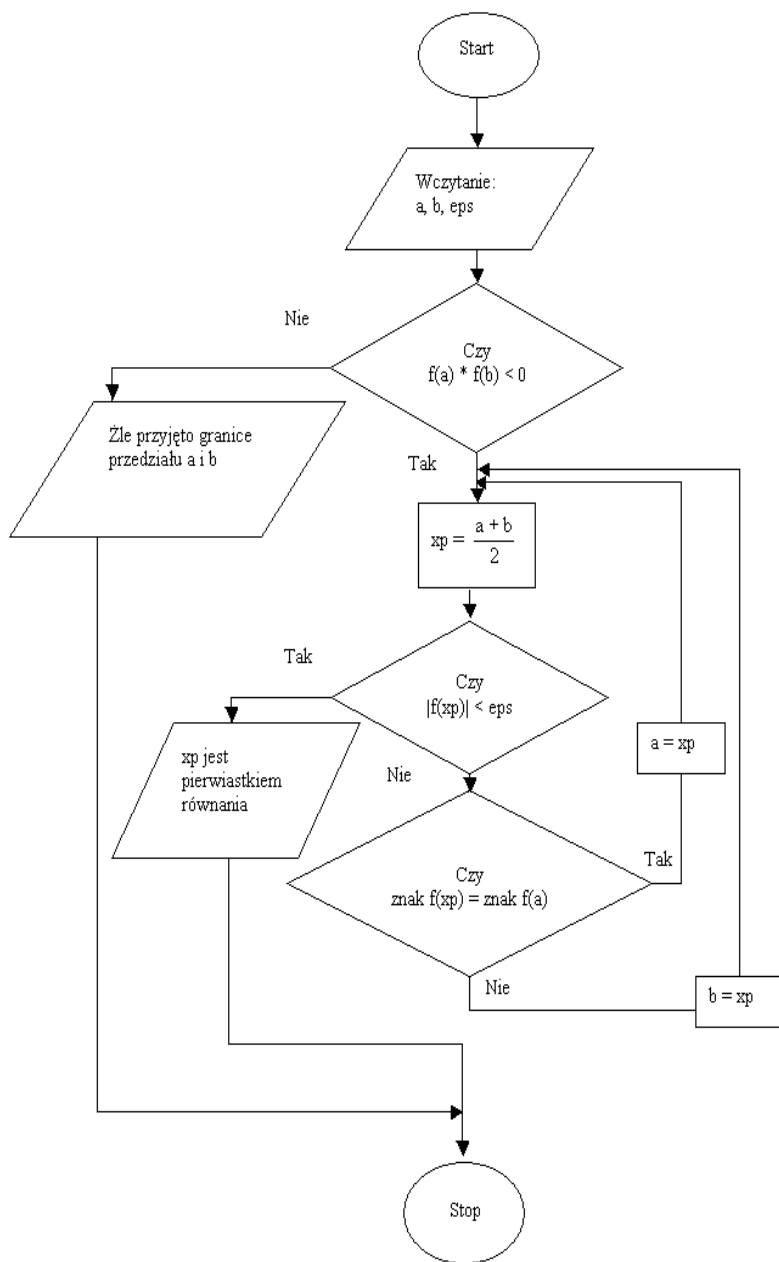
```
        b = xp
    End If
    Call Pierwiastek(a, b, xp, eps, info)
End If
Else
    info = "Błąd przedziału"
    Exit Sub
End If
End Sub


---


Private Function F(ByVal x As Double) As Double
    'Własna funkcja
    Dim A, B, C, D As Double
    A = 1
    B = -35
    C = 350
    D = -1000
    F = A * x ^ 3 + B * x ^ 2 + C * x + D
End Function
```

Podobnie jak metodę fałsi – metodę bisekcji można wykorzystać do znajdowania pierwiastka rzeczywistego równania algebraicznego dowolnego stopnia, należy jedynie samodzielnie napisać procedurę Function F(x) pozwalającą obliczyć wartość funkcji dla danego x.

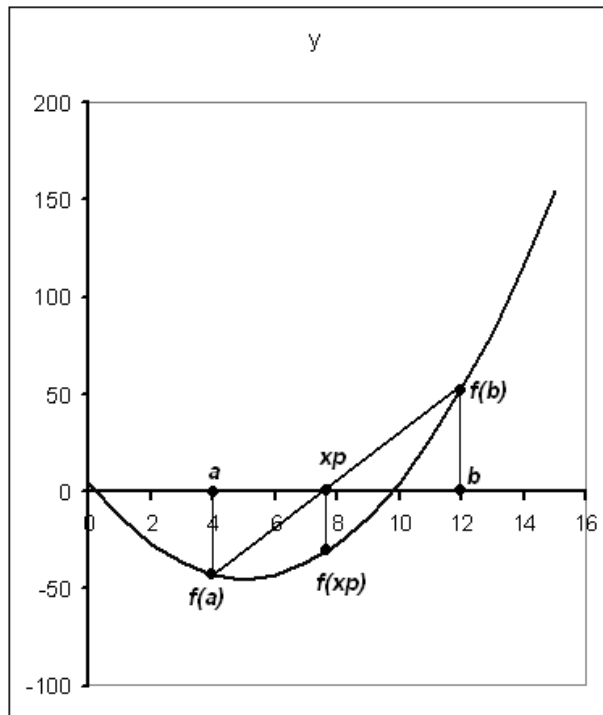
Wynik metody bisekcji – podobnie jak w metodzie fałsi zależy od przyjętych granic przedziału [a, b].



Rysunek 8.7. Schemat algorytmu znajdowanie pierwiastka rzeczywistego równania algebraicznego metodą połowienia przedziału

8.4. Metoda *falsi* (siecznej) znajdowania pierwiastków algebraicznego równania nieliniowego

Przybliżone rozwiązywanie równania algebraicznego [4] nieliniowego w metodzie *falsi* – interpolacji liniowej – zakłada, że za przybliżoną wartość pierwiastka można przyjąć, x_p , czyli punkt przecięcia osi x sieczną, rysunek 8.12.



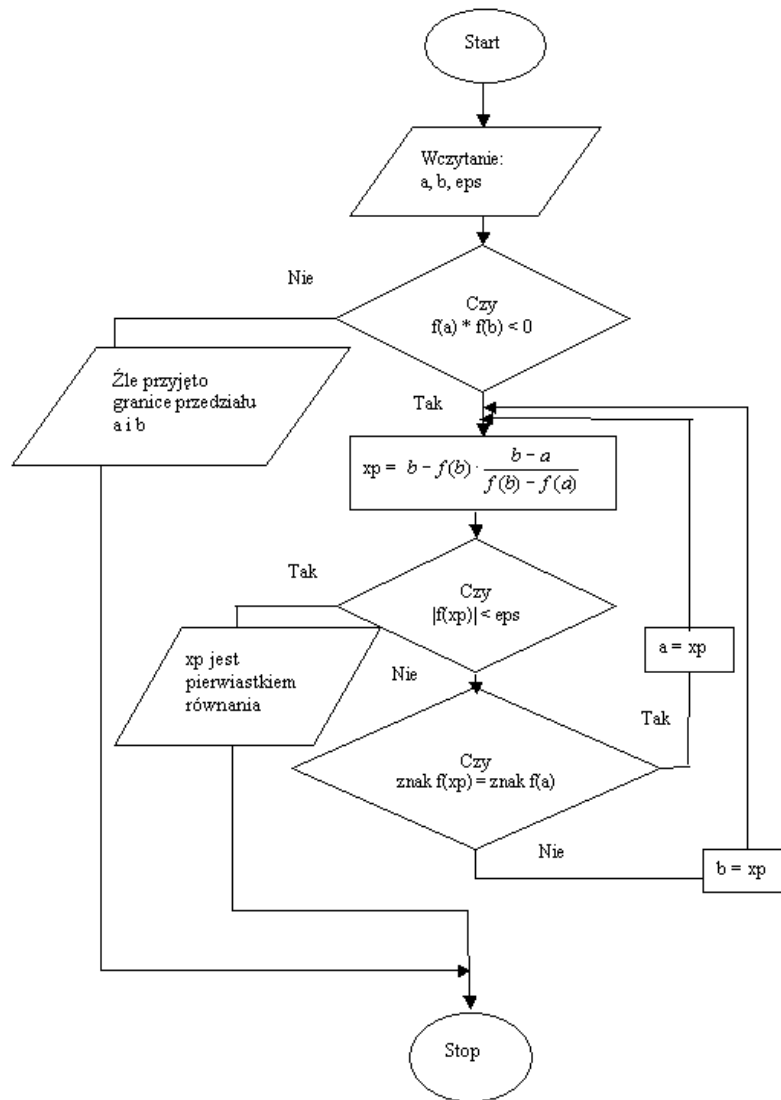
Rysunek 8.8. Graficzna ilustracja metody siecznych

Gdzie x_p , punkt przecięcia osi x sieczną wyraża się wzorem:

$$xp = b - f(b) \cdot \frac{(b-a)}{f(b) - f(a)} \quad (8.6)$$

Jeśli przyjmiemy dokładność obliczeń za eps, to algorytm, rysunek 8.9, może mieć postać:

1. Przyjąć (arbitralnie) przedział [a, b] i eps
2. Sprawdzamy, czy $f(a) \cdot f(b) < 0$.
3. Odpowiedź TAK oznacza, że dobrze przyjęliśmy granice przedziału i pierwiastek równania znajduje się wewnątrz przedziału i można kontynuować działania.
4. Odpowiedź NIE oznacza, że granice przedziału zostały błędnie przyjęte i należy przerwać obliczenia.



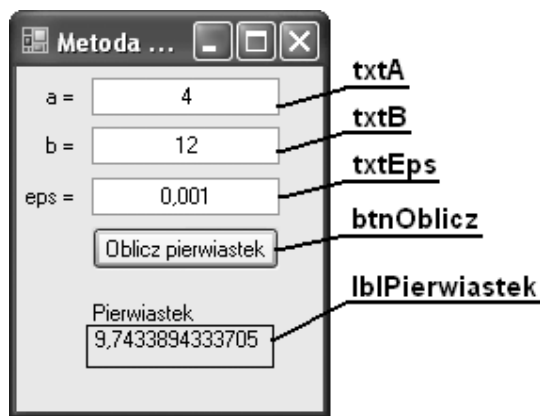
Rysunek 8.9. Algorytm metody siecznej

5. Obliczamy x_p ze wzoru 8.6
6. Jeżeli $|f(x_p)| < \text{eps}$ zakończenie programu, x_p jest wartością pierwiastka.
7. Jeśli $|f(x_p)| \geq \text{eps}$ należy sprawdzić:

- 7.1. Czy znak $f(x_p)$ jest taki sam jak znak $f(a)$
- 7.2. Jeśli TAK to pierwiastek znajduje się w przedziale $[x_p, b]$, czyli $a=x_p$ i dalsze obliczenia od p-tu 5.
- 7.3. Jeśli NIE, to pierwiastek znajduje się wewnątrz przedziału $[a, x_p]$, czyli $b=x_p$ i dalsze obliczenia od p-tu 5.

Funkcja $y(x) = A \cdot x^2 + B \cdot x + C$, dla $A = 2$, $B = -20$, $C = 5$ posiada dwa pierwiastki $x_1 = 0,25658351$, $x_2 = 9,74341649$. Dla przedziału $[4, 12]$ i dokładności $\text{eps} = 0,001$ program oblicza pierwiastek $x_p = 9,74338943$

Przykład aplikacji w języku Visual Basic



Rysunek 8.10. Propozycja formularza

Kod programu

```
Private Sub btnOblicz_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnOblicz.Click
    Dim a, b, xp, eps As Double
    Dim info As String = ""
    a = Double.Parse(txtA.Text)
    b = Double.Parse(txtB.Text)
    eps = Double.Parse(txtEps.Text)
    Call Pierwiastek(a, b, xp, eps, info)
    If info = "OK" Then
        lblPierwiastek.Text = xp
    Else
```

```
        lblPierwiastek.Text = info
    End If
End Sub


---


Private Sub Pierwiastek(ByVal a, ByVal b, ByRef xp, _
    ByVal eps, ByRef info)
    If F(a) * F(b) < 0 Then
        xp = b - (F(b) * (b - a)) / (F(b) - F(a))
        If Math.Abs(F(xp)) < eps Then
            info = "OK"
            Exit Sub
        Else
            If Math.Sign(F(xp)) = Math.Sign(F(a)) Then
                a = xp
            Else
                b = xp
            End If
            Call Pierwiastek(a, b, xp, eps, info)
        End If
    Else
        info = "Błąd przedziału"
        Exit Sub
    End If
End Sub


---


Private Function F(ByVal x As Double) As Double
    'Własna funkcja
    Dim A, B, C As Double
    A = 2
    B = -20
    C = 5
    F = A * x ^ 2 + B * x + C
End Function
```

Aplikację można wykorzystać do znajdowania pierwiastka rzeczywistego równania algebraicznego dowolnego stopnia, należy jedynie samodzielnie napisać procedurę Function F(x) pozwalającą obliczyć wartość funkcji dla danego x.

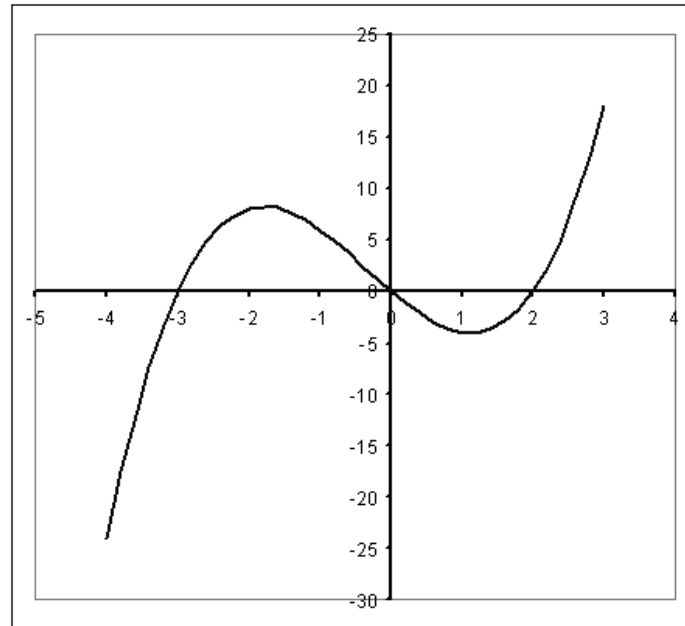
Np. dla funkcji $y(x) = A \cdot x^3 + B \cdot x^2 + C \cdot x + D$, $A = 1$, $B = 1$, $C = -6$, $D = 0$ należy napisać własną procedurę o postaci:

```
Private Function F(ByVal x As Double) As Double
    'Własna funkcja
    Dim A, B, C, D As Double
    A = 1
    B = 1
    C = -6
    D = 0
    F = A * x ^ 3 + B * x ^ 2 + C * x + D
End Function
```

Funkcja ta ma trzy pierwiastki: $x_1 = -3$, $x_2 = 0$, $x_3 = 2$, patrz rysunek 8.12, które możemy wyliczyć algebraicznie, a także obliczyć powyższą metodą, porównanie wyników - patrz tabela poniżej.

Tabela 1

Przedział [a, b]	Dokładność	Pierwiastek obliczony	Pierwiastek rzeczywisty
[-4, -2]	0,001	-2,999965	-3
[-2, 1]	0,001	0	0
[1, 3]	0,001	1,999941	2



Rysunek 8.11. Przebieg funkcji $x^3 + x^2 - 6x = 0$

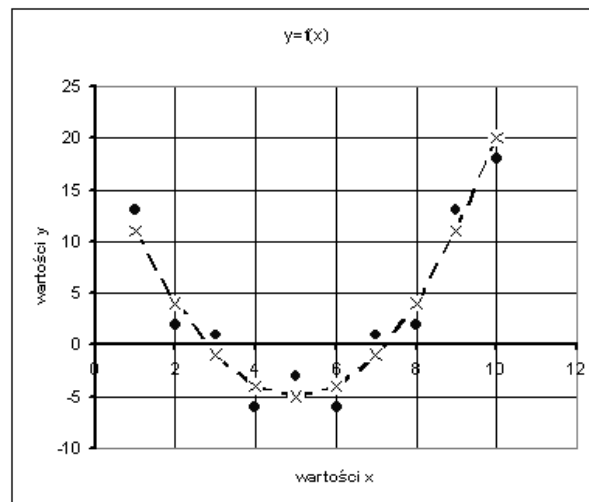
Wynik metody fałsi zależy od przyjętego przedziału, np. gdy przyjmie-
my przedział nie $[-2, 1]$ lecz $[-2, 0,5]$ uzyskujemy $x_p = 0,0000044$, a dla
przedziału $[-1, 1]$ $x_p = 0,0000075$.

8.5. Metoda Monte Carlo – znajdowanie współczynników funkcji aproksymującej

Niech będą dane, w postaci tabeli i wykresu (patrz rysunek 8.12) wyniki
pomiarów.

Tablica 1

x	y
1	13
2	2
3	1
4	-6
5	-3
6	-6
7	1
8	2
9	13
10	18



Rysunek 8.12. Dane pomiarowe
i krzywa aproksymująca

Należy tak dobrać współczynniki A , B i C funkcji aproksymującej

$$y = f(x) = A \cdot x^2 + B \cdot x + C \quad (8.7)$$

aby suma kwadratów różnic

$$\sum_{i=1}^{10} (y_p - f(x_i))^2 \quad (8.8)$$

była najmniejsza (gdzie y_p to wartości z pomiarów). Są to znane założenia metody najmniejszych kwadratów. Aby jednak nie rozwiązywać układów równań, które mogą być bardziej skomplikowane niż w przedstawianym przypadku, posłużymy się metodą Monte Carlo.

Algorytm będzie następujący:

1. Musimy założyć granice dla współczynników A, B i C: $A_{min} < A < A_{max}$, $B_{min} < B < B_{max}$, $C_{min} < C < C_{max}$.
2. Generujemy, posługując się generatorem liczb przypadkowych, w tak przyjętych granicach, wartości współczynników A, B i C, które oznaczymy jako Alos, Blos i Clos.
3. Dla współczynników Alos, Blos, Clos obliczymy, dla kolejnych wartości zmiennej niezależnej x_i (pierwsza kolumna w Tabelicy 1, wartości funkcji $y_i = f(x_i)$).
4. Dla kolejnych wartości zmiennej niezależnej x_i obliczymy $(y_p - f(x_i))^2$ (kwadraty różnic).
5. Sumujemy kwadraty różnic: $\sum_{i=1}^{10} (y_p - f(x_i))^2$.
6. Porównamy, czy obecnie wyliczona suma kwadratów różnic jest mniejsza od wyliczonej poprzednio.
7. Jeśli TAK – obecnie wygenerowane współczynniki Alos, Blos, Clos są lepsze od poprzednich i należy je zapamiętać.
8. Jeśli NIE – wygenerowane współczynniki Alos, Blos Clos są pomijane
9. Sprawdzamy, czy liczba zaplanowanych losowań współczynników została wyczerpana.
10. Jeśli NIE – przechodzimy do punktu 2.
11. Jeśli TAK – kończymy obliczenia i drukujemy współczynniki Alos, Blos, Clos.

ROZDZIAŁ 8

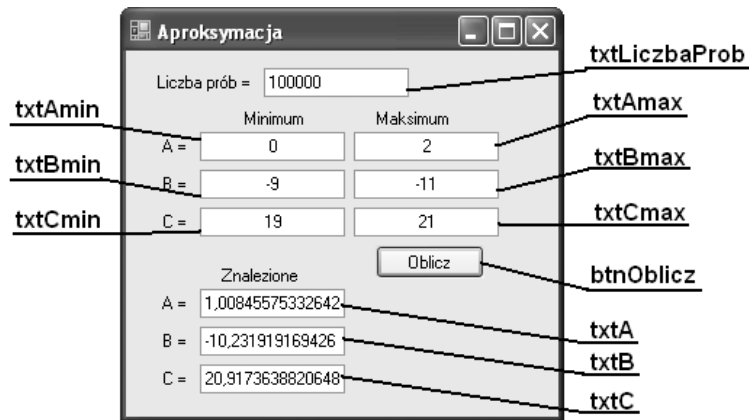
Jeden z etapów zadania, dla współczynników $A_{los}=1,05$, $B_{los}=-9,9$, $C_{los}=19,9$ pokazano w Tabeli 2

Tabela 2

x	y	f(x)	$(y-f(x))^2$
1	13	11,05	3,8025
2	2	4,30	5,2900
3	1	-0,35	1,8225
4	-6	-2,90	9,6100
5	-3	-3,35	0,1225
6	-6	-1,70	18,4900
7	1	2,05	1,1025
8	2	7,90	34,8100
9	13	15,85	8,1225
10	18	25,90	62,4100

suma = 145,5825

Przykład aplikacji w języku Visual Basic



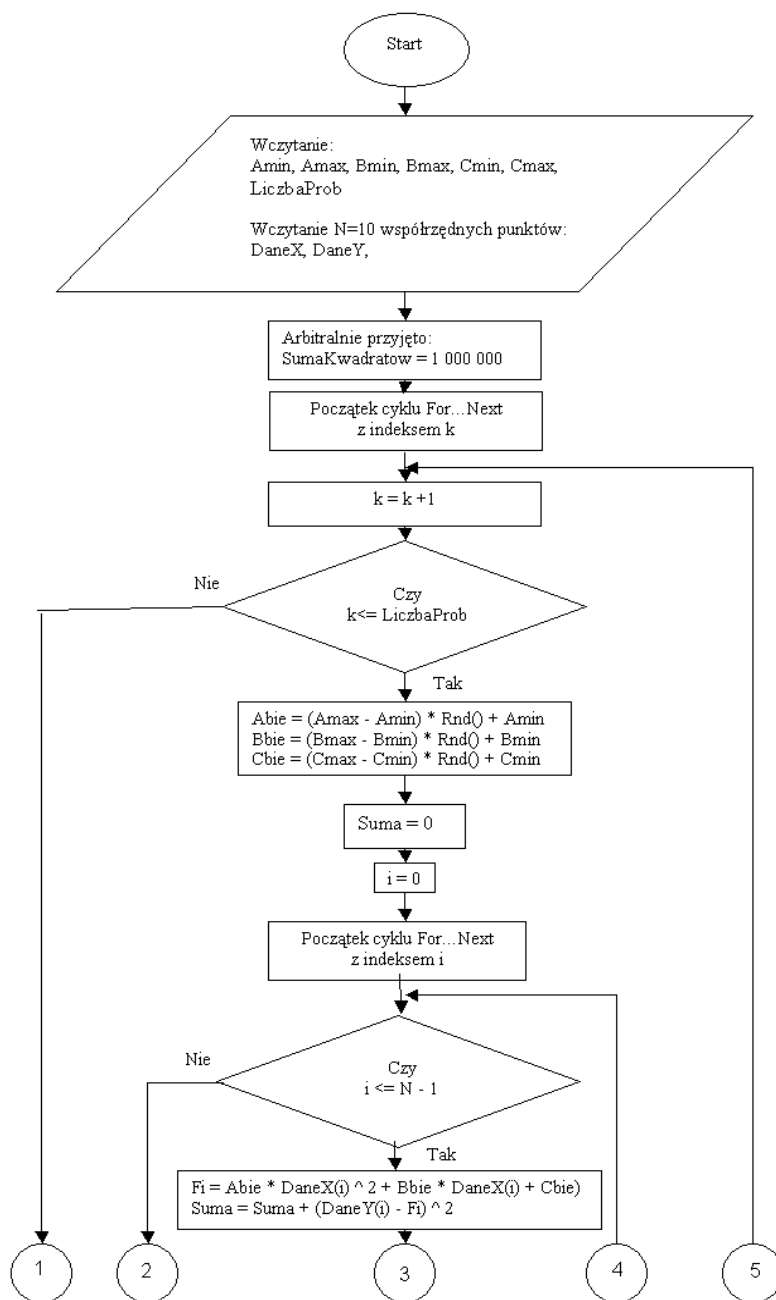
Rysunek 8.13. Postać formularza

Kod programu

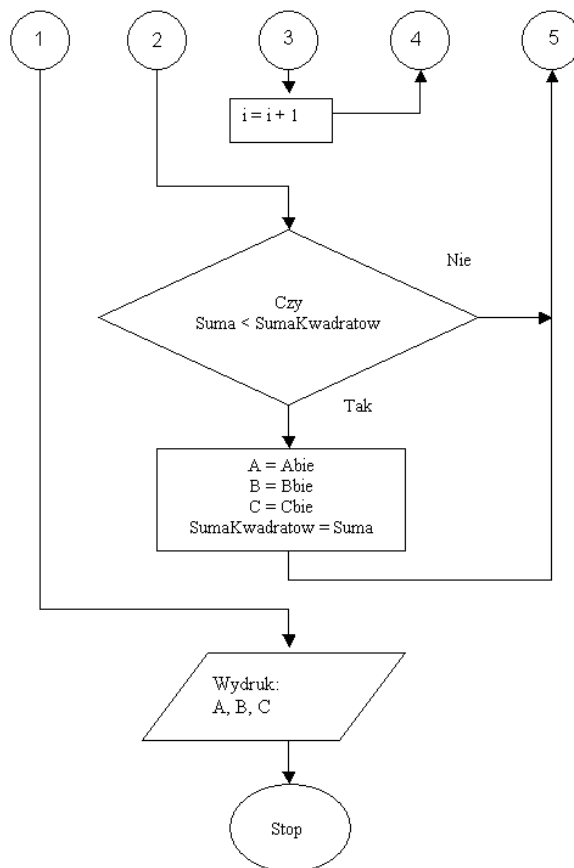
```
Private Sub Form1_Load(ByVal sender As _  
    System.Object, ByVal e As _  
        System.EventArgs) _  
    Handles MyBase.Load  
    Randomize()  
End Sub
```

```
Private Sub btnOblicz_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnOblicz.Click
    Dim LiczbaProb, i, k, N As Integer
    Dim Amin, Amax, A, Abie As Double
    Dim Bmin, Bmax, B, Bbie As Double
    Dim Cmin, Cmax, C, Cbie As Double
    Dim SumaKwadratow, Suma, Fi As Double
    Dim DaneX() As Double = _
        {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    Dim DaneY() As Double = _
        {13, 2, 1, -6, -3, -6, 1, 2, 13, 18}
    N = 10
    LiczbaProb = _
        Integer.Parse(txtLiczbaProb.Text)
    Amin = Double.Parse(txtAmin.Text)
    Amax = Double.Parse(txtAmax.Text)
    Bmin = Double.Parse(txtBmin.Text)
    Bmax = Double.Parse(txtBmax.Text)
    Cmin = Double.Parse(txtCmin.Text)
    Cmax = Double.Parse(txtCmax.Text)
    SumaKwadratow = 1000000.0
    For k = 1 To LiczbaProb
        Abie = (Amax - Amin) * Rnd() + Amin
        Bbie = (Bmax - Bmin) * Rnd() + Bmin
        Cbie = (Cmax - Cmin) * Rnd() + Cmin
        Suma = 0
        For i = 0 To N - 1
            Fi = (Abie * DaneX(i) ^ 2 + _
                Bbie * DaneX(i) + Cbie)
            Suma = Suma + (DaneY(i) - Fi) ^ 2
        Next i
        If Suma < SumaKwadratow Then
            A = Abie
            B = Bbie
            C = Cbie
            SumaKwadratow = Suma
        End If
    Next k
    txtA.Text = A.ToString
    txtB.Text = B.ToString
    txtC.Text = C.ToString
End Sub
```

W algorytmie sprawdzany jest warunek czy $Suma < SumaKwadratow$.
W pierwszym wejściu w instrukcję cyklu For...Next obliczymy wartość
zmiennej Suma natomiast wartość zmiennej SumaKwadratow musi już



Rysunek 8.14. Algorytm



Rysunek 8.15. Algorytm c.d.

istnieć i być większa od zmiennej $Suma$. Należy zatem przyjąć arbitralnie tak dużą wartość dla zmiennej $SumaKwadratow$, aby na pewno była ona większa niż spodziewana wartość zmiennej $Suma$.

W kodzie powyżej przyjęto dla zmiennej $SumaKwadratow$ wartość 1000000.



Sortowanie

9.1. Wprowadzenie

Znanych jest wiele algorytmów porządkowania szeregu elementów wg określonego kryterium czyli sortowania [8]. Algorytmy sortowania można klasyfikować według złożoności, zapotrzebowania na pamięć komputera, zmiany lub pozostawienia kolejności zbioru wyjściowego i innych kryteriów.

W rozdziale omówiono dwa algorytmy sortowania.

9.2. Sortowanie przez wybieranie

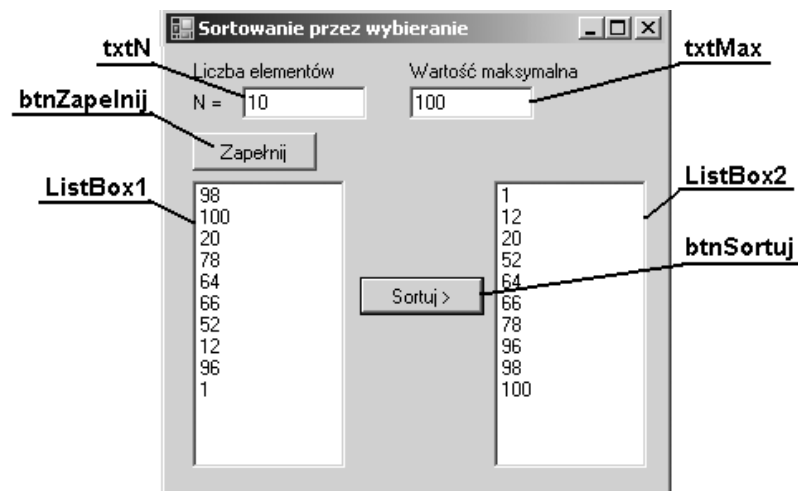
Algorytm sortowania przez wybieranie jest jednym z prostszych algorytmów sortowania. Dane do sortowania znajdują się w Tabeli z indeksami od 1 do N (patrz uwaga w algorytmie Sortowania bąbelkowego).

Przebieg algorytmu

1. Poczynając od elementu $i = 1$ Tabeli poszukujemy jej najmniejszego elementu.
2. Po znalezieniu najmniejszego elementu umieszczamy go w komórce tabeli o indeksie 1, a znajdujący się tam element umieszczamy w tej komórce, gdzie był znaleziony element najmniejszy.
3. Następnie rozpoczynając od elementu $i+1$ szukamy w tabeli najmniejszego elementu i i zamieniamy go z elementem na pozycji $i+1$
4. Powtarzamy szukanie i zamianę dla wszystkich $N-1$ elementów tabeli. Np. niech tabela składa się z elementów: (1) 9, (2) 5, (3) 8, (4) 6.
5. Poczynając od elementu (1) znajdujemy najmniejszy na pozycji (2). Dokonujemy zamiany elementów (1) i (2). Tabela będzie zatem miała postać: (1) 5, (2) 9, (3) 8, (4) 6.

6. Poczynając od elementu (2) poszukujemy najmniejszego elementu. Okazuje się, że element (4) jest w tym zbiorze elementem najmniejszym. Dokonujemy zamiany elementów (2) i (4). Tabela będzie zatem miała postać: (1) 5, (2) 6, (3) 8, (4) 9.
7. Poczynając od elementu (3) poszukujemy najmniejszego elementu. Najmniejszym elementem z tym zbiorze jest element (3). Pozostawiamy tabelę bez zmiany.
8. Ponieważ dokonaliśmy N-1 sprawdzeń kończymy algorytm. Tabela jest posortowana.

Przykład aplikacji w języku Visual Basic



Rysunek 9.1. Propozycja formularza

Kod aplikacji

```
Public N As Integer 'Liczba elementów
Private Sub Form1_Load(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles MyBase.Load
    Randomize()
End Sub
Private Sub btnZapelnij_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnZapelnij.Click
    Dim Max, index As Integer
```

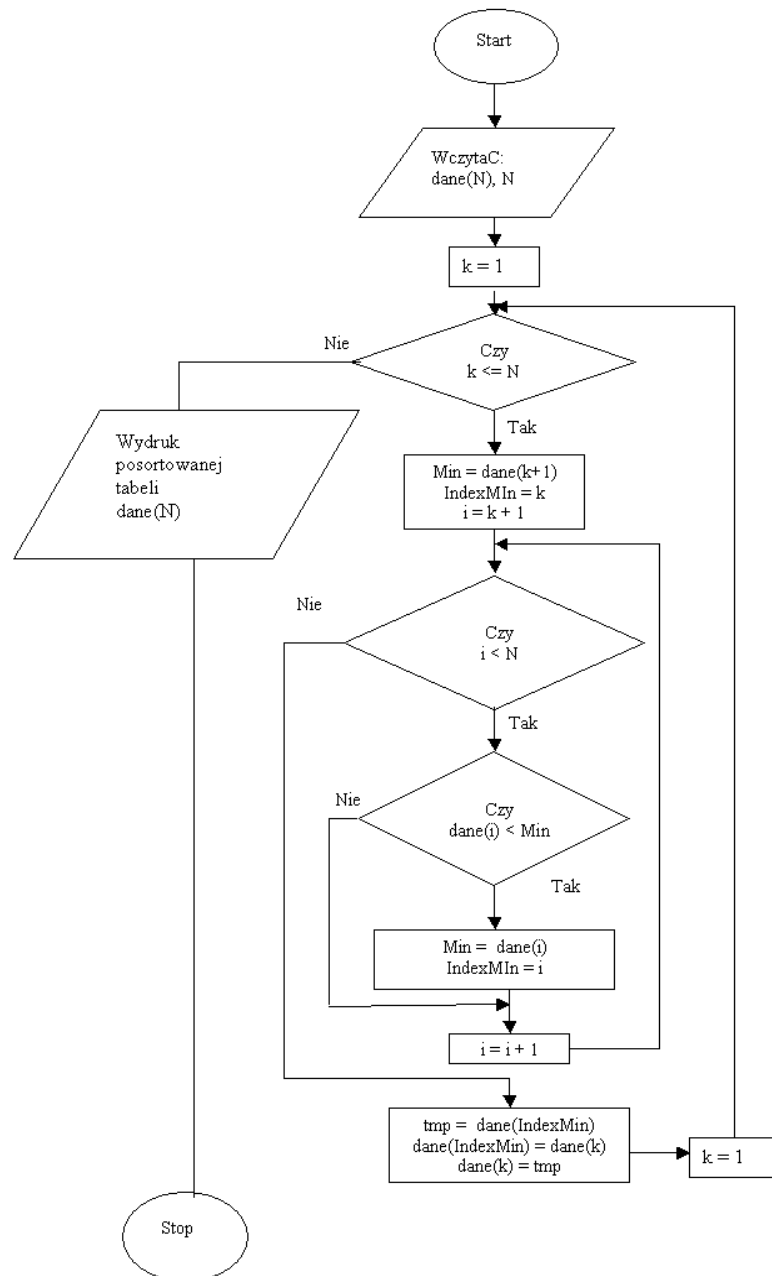
```
N = CInt(txtN.Text)
Max = CInt(txtMax.Text)
ListBox1.Items.Clear()
For index = 1 To N
    ListBox1.Items.Add(CInt(Rnd() * Max))
Next
End Sub


---


Private Sub btnSortuj_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnSortuj.Click
    Dim dane(N) As Integer
    Dim Min, tmp As Integer
    Dim IndexMin As Integer
    Dim i, k As Integer
    For i = 1 To N
        dane(i) = ListBox1.Items.Item(i - 1)
    Next
    '-----
    For k = 1 To N
        Min = dane(k)
        IndexMin = k
        For i = k + 1 To N
            If dane(i) < Min Then
                Min = dane(i)
                IndexMin = i
            End If
        Next
        tmp = dane(IndexMin)
        dane(IndexMin) = dane(k)
        dane(k) = tmp
    Next
    '-----
    ListBox2.Items.Clear()
    For i = 1 To N
        ListBox2.Items.Add(dane(i))
    Next
End Sub
```

Aplikacja działa w ten sposób, że zapełniamy, przyciskiem Zapełnij listę ListBox1 liczbami naturalnymi z przedziału [1 – Wartość maksymalna] w ilości Liczba elementów równa N.

Następnie, po kliknięciu przycisku Sortuj, wygenerowane liczby są przekładane do tablicy dane(N), sortowane i dla pokazania efektu sortowania, wyświetlane w listy ListBox2.



Rysunek 9.2. Algorytm sortowania przez wybierania

9.3. Sortowanie - algorytm bąbelkowy

Jednym z najczęściej wymienianych i stosunkowo prostym algorytmem sortowania jest algorytm bąbelkowy.

Elementy wymagające sortowania najczęściej umieszcza się w tablicy.

Algorytm polega na wielokrotnym przeglądaniu elementów tablicy i porównywaniu ich parami. Jeśli w jakiejś parze kolejność elementów jest zaburzona – wtedy elementy są zamieniane miejscami.

Niech dane znajdują się w tablicy o nazwie Tabela, o N elementach ponumerowanych od $k = 1$ do N. Należy posortować elementy rosnąco, tzn. od najmniejszego do największego.

UWAGA

W języku Visual Basic tablice numerowane są od 0. Oznacza to, że gdy zadeklarujemy tablicę: Tabela(3) – będą istniały elementy: Tabela(1), Tabela(2), Tabela(3), ale także będzie istniał element Tabela(0). W sumie zatem deklarując tablicę Tabela(3) kreujemy nie 3 lecz 4 jej elementy!

Aby jednak mówiąc: element pierwszy – posługiwać się elementem Tabela(1), a mówiąc element drugi – posługiwać się elementem Tabela(2), itd. w dalszych obliczeniach i wyjaśnieniach, dla jasności wyводу (zwłaszcza dla mniej doświadczonych programistów) – element tablicy z indeksem 0, mimo, że kreowany w momencie deklarowania – będziemy pomijali.

Nie będziemy go wykorzystywali.

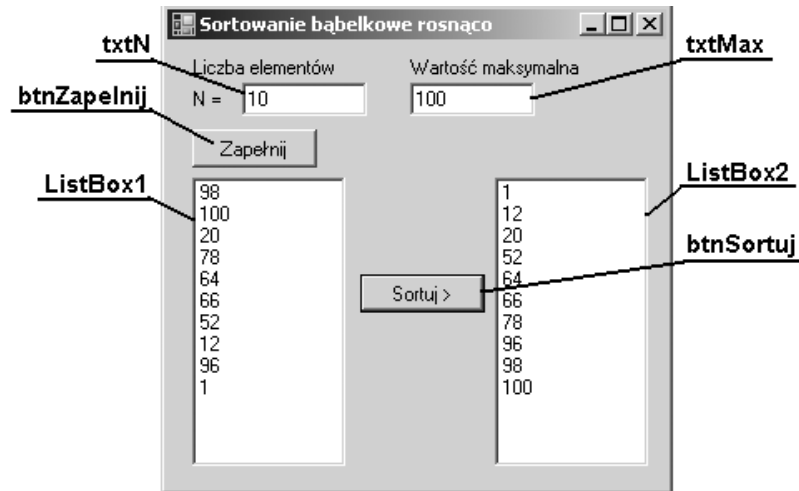
Algorytm sortowania rosnąco

Dana jest tablica Tabela zawierająca N elementów, należy je posortować rosnąco, tzn. od najmniejszego do największego. Po sortowaniu element najmniejszy powinien być elementem pierwszym, a element największy powinien być ostatni.

1. Poczynając od indeksu = 1, a kończąc na indeksie = N-1
 - 1.1. Poczynając od wskaźnika = 1, a kończąc na wskaźniku = N-1 (patrz „Dodatkowe uwagi” dalej)

- 1.1.1. Sprawdzić, czy
Tabela(wskaźnik) > Tabela(wskaźnik + 1)
- 1.1.2. Jeśli TAK – należy elementy zamienić miejscami
- 1.1.3. Jeśli NIE – elementy pozostają na swoich miejscach

Przykład aplikacji w języku Visual Basic



Rysunek 9.3. Propozycja formularza

Aplikacja jest rozbudowana o część kodu generującą zbiór liczb przeznaczonych do sortowania i część wizualizującą zbiór przed i po sortowaniu. Zbiór liczb do sortowania tworzony jest za pomocą generatora liczb przypadkowych, generującego liczby całkowite z zakresu 1 - Max, gdzie wartość Max (liczba całkowita) może być wprowadzana z klawiatury. Liczbę elementów N zbioru liczb do sortowania także możemy wprowadzać z klawiatury.

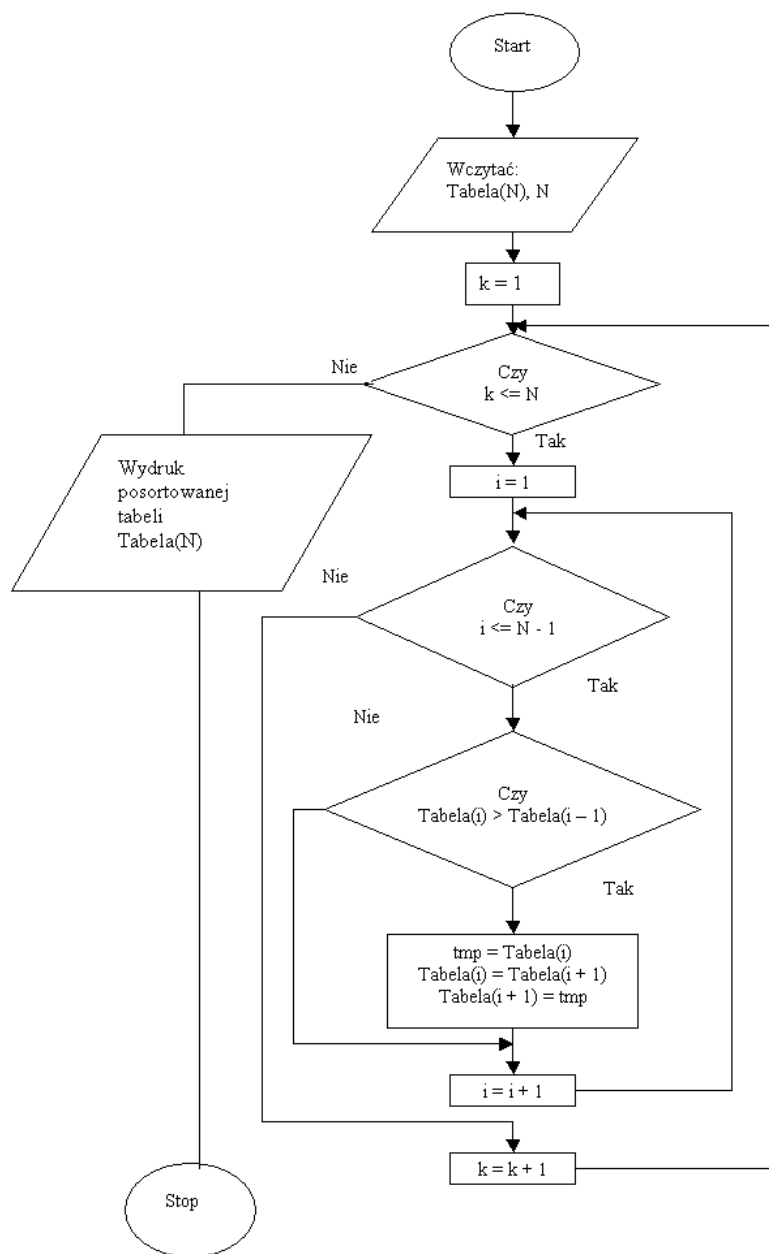
Kod programu

```
Public N As Integer
Private Sub Form1_Load(ByVal sender As _
    System.Object, ByVal e As _
    System.EventArgs) Handles MyBase.Load
    Randomize()
End Sub
```

```
Private Sub btnZapelnij_Click(ByVal sender As _
    System.Object, ByVal e As _
    System.EventArgs) Handles _
    btnZapelnij.Click
    Dim Max, index As Integer
    N = Integer.Parse(txtN.Text)
    Max = Integer.Parse(txtMax.Text)
    ListBox1.Items.Clear()
    For index = 1 To N
        ListBox1.Items.Add(CInt(Rnd() * Max))
    Next
End Sub


---


Private Sub btnSortuj_Click(ByVal sender As _
    System.Object, ByVal e As _
    System.EventArgs) Handles btnSortuj.Click
    Dim Tabela(N) As Integer
    Dim danesort(N) As Integer
    Dim tmp As Integer
    Dim i, k As Integer
    For i = 1 To N
        Tabela(i) = ListBox1.Items.Item(i - 1)
    Next
    'sortowanie
    '-----
    For k = 1 To N - 1
        For i = 1 To N - 1
            If Tabela(i) > Tabela(i + 1) Then
                tmp = Tabela(i)
                Tabela(i) = Tabela(i + 1)
                Tabela(i + 1) = tmp
            End If
        Next
    Next
    '-----
    ListBox2.Items.Clear()
    For i = 1 To N
        ListBox2.Items.Add(Tabela(i))
    Next
End Sub
```



Rysunek 9.4. Algorytm sortowania „bąbelkowego”

Dodatkowe uwagi

Aby algorytm działał sprawnie należy przedyskutować granice instrukcji cyklu.

Gdy przeglądamy parami elementy tabeli pierwszy raz, od początku do końca i zamieniamy miejscami elementy tak, że większy z nich umieszczony jest zawsze w elemencie $(i + 1)$ - powoduje to, że gdy zakończymy tę instrukcję cyklu (to ta wewnętrzna, po indeksie i) element największy w tabeli znajdzie się na końcu tabeli w elemencie Tabela(N).

Gdy po raz drugi sprawdzimy tabelę - to element drugi co do wielkości znajdzie się w elemencie tabeli Tabela(N-1).

Skoro tak działa to sortowanie, to za pierwszym razem należy porównywać elementy Tabela(N-1) i Tabela(N).

Za drugim razem ostatnie porównanie powinno dotyczyć elementów Tabela(N-2) i Tabela(N-1)

Za trzecim porównanie powinno dotyczyć już tylko elementów Tabela(N-3) i Tabela(N-2). Czyli nie musimy za każdym razem dokonywać porównań elementów aż do końca, do N-tego elementu.

Zamiast zatem kodu sortującego zamieszczonego powyżej należy zmienić granice drugiej instrukcji cyklu, patrz poniżej, wiersz wytłuszczony:

```
'sortowanie
'-----
For k = 1 To N - 1
  For i = 1 To N - 1 - (k - 1)
    If Tabela(i) > Tabela(i + 1) Then
      tmp = Tabela(i)
      Tabela(i) = Tabela(i + 1)
      Tabela(i + 1) = tmp
    End If
  Next
Next
Next
'-----
```

Poprawka ta ma ogromne znaczenie przyspieszające działanie algorytmu wtedy, gdy sortujemy bardzo duże tabele.

10

Literatura

1. Buczek B., *Ćwiczenia, Algorytmy*, Helion 2009.
2. Heineman G.T., Pollice G., Selkow S., *Algorytmy, Almanach*, Helion 2010.
3. Kalinowska-Iszkowska M., Iszkowski W., Walczak K., *Zbiór zadań do nauki programowania FORTRAN*, WPW Warszawa 1978.
4. Łubowicz J., Baraniecki M., Nosowski W., Siudak M., Żebrowski W., *Zbiór zadań z metod numerycznych ALGOL 1204*, WPW Warszawa 1974.
5. Nievergelt J., Farrar J., C., Reingold E., M., *Informatyczne rozwiązywanie zadań matematycznych*, WNT Warszawa 1978.
6. Osiński Z., Wróbel J., *Teoria konstrukcji maszyn*, PWN Warszawa 1982
7. Van Tassel, D., *Praktyka programowania*, WNT Warszawa 1978.
8. Wróblewski P., *Algorytmy, struktury danych i techniki programowania*, Helion, 2010.
9. <http://www.matematycy.interklasa.pl/euklides/index.html>
projekt badawczy „Księgi Euklidesa” (data skorzystania – listopad 2010)
10. <http://pl.wikipedia.org/wiki/Sortowanie> (data skorzystania - listopad 2010)