Algorytmy i struktury danych

wykład 2

Plan wykładu:

- Pojęcie algorytmu.
- Projektowanie wstępujące i zstępujące.
- Rekurencja.

Pojęcie algorytmu

Algorytm – skończony zbiór operacji, koniecznych do wykonania zadania z pewnej klasy zadań, przetwarzając dane należące do pewnej klasy danych.

Ciekawostka:
Słowo „algorytm” pochodzi od łacińskiej wersji nazwiska perskiego matematyka Muhammad ibn Musa al-Chuwarizmi, żyjącego w IX wieku.

Algorytm ma przeprowadzić system z pewnego stanu początkowego do stanu końcowego. Algorytm może zostać zaimplementowany jako program komputerowy lub ciąg czynności dla innego urządzenia. Przykładem może tu być maszyna tkająca, wykorzystywana przez chińskich rzemieślników.

Najprostszy przykład algorytmu – przepis kuchenny:
- dane: lista produktów,
- zbiór czynności,
- wynik: potrawa.

Algorytm – skończony zbiór operacji, koniecznych do wykonania zadania z pewnej klasy zadań, przetwarzając dane należące do pewnej klasy danych.

Algorytmy wykonywane w życiu codziennym:

- przygotowywanie potraw,
- jazda samochodem,
- wchodzenie/wychodzenie z pomieszczenia,
- kupowanie w sklepach,
- obsługa pilota telewizyjnego,
- włączanie/wyłączanie urządzeń,
- … i wiele innych.

Algorytm – skończony zbiór operacji, koniecznych do wykonania zadania z pewnej klasy zadań, przetwarzając dane należące do pewnej klasy danych.

Algorytmy wykonywane w życiu codziennym:
- przygotowywanie potraw,
- jazda samochodem,
- wchodzenie/wychodzenie z pomieszczenia,
- kupowanie w sklepach,
- obsługa pilota telewizyjnego,
- włączanie/wyłączanie urządzeń,
- … i wiele innych.

Gdzie można znaleźć algorytmy?:
- w systemach komputerowych,
- w telefonach komórkowych,
- w aparatach fotograficznych,
- w urządzeniach mechanicznych,
- … wszędzie tam, gdzie wykonywane czynności można podzielić na powtarzające się czynności.

Czynności związane z opracowaniem algorytmu:

1. określenie problemu,
2. określenie danych wejściowych,
3. określenie wyniku, jaki powinien zostać wyznaczony,
4. opracowania rozwiązania problemu.

Algorytm powinien spełniać następujące założenia:

1. poprawność
2. jednoznaczność
3. skończoność
4. uniwersalność

Czynności związane z opracowaniem algorytmu:

1. określenie problemu,
2. określenie danych wejściowych,
3. określenie wyniku, jaki powinien zostać wyznaczony,
4. opracowania rozwiązania problemu.

Algorytm powinien spełniać następujące założenia:

1. poprawność
2. jednoznaczność
3. skończoność
4. uniwersalność

dla każdego właściwego zestawu danych, po wykonaniu skończonej liczby czynności, prowadzi do poprawnych wyników

Czynności związane z opracowaniem algorytmu:
1. określenie problemu,
2. określenie danych wejściowych,
3. określenie wyniku, jaki powinien zostać wyznaczony,
4. opracowania rozwiązania problemu.

Algorytm powinien spełniać następujące założenia:
1. poprawność
2. jednoznaczność
3. skończoność
4. uniwersalność

dla każdego właściwego zestawu danych, po wykonaniu skończonej liczby czynności, prowadzi do poprawnych wyników

wykonanie algorytmu dla tych samych danych powoduje wyznaczenie tego samego wyniku

Czynności związane z opracowaniem algorytmu:

1. określenie problemu,
2. określenie danych wejściowych,
3. określenie wyniku, jaki powinien zostać wyznaczony,
4. opracowania rozwiązania problemu.

Algorytm powinien spełniać następujące założenia:

1. poprawność
2. jednoznaczność
3. skończoność
4. uniwersalność

dla każdego właściwego zestawu danych, po wykonaniu skończonej liczby czynności, prowadzi do poprawnych wyników

wykonanie algorytmu dla tych samych danych powoduje wyznaczenie tego samego wyniku

każdy algorytm musi kiedyś zakończyć się

Czynności związane z opracowaniem algorytmu:

1. określenie problemu,
2. określenie danych wejściowych,
3. określenie wyniku, jaki powinien zostać wyznaczony,
4. opracowania rozwiązania problemu.

Algorytm powinien spełniać następujące założenia:

1. poprawność – dla każdego właściwego zestawu danych, po wykonaniu skończonej liczby czynności, prowadzi do poprawnych wyników
2. jednoznaczność – wykonanie algorytmu dla tych samych danych powoduje wyznaczenie tego samego wyniku
3. skończoność – każdy algorytm musi kiedyś zakończyć się
4. uniwersalność – algorytm rozwiązuje dowolne zadanie z pewnej grupy zadań

Sposoby zapisywania algorytmów:

1. język naturalny
2. schematy blokowe
3. języki formalne
4. funkcje rekurencyjne

Sposoby zapisywania algorytmów:

1. język naturalny
2. schematy blokowe
3. języki formalne
4. funkcje rekurencyjne

Sposób najbardziej rozpowszechniony.
Zalety:

- prostota zapisu,
- brak specyficznych konstrukcji formalnych,
- szeroki zasób słownictwa.

Wady:

- mała precyzja,
- możliwe błędy interpretacji,
- miejscami zbyt mała zwięzłość.

Sposoby zapisywania algorytmów:
1. język naturalny
2. schematy blokowe
3. języki formalne
4. funkcje rekurencyjne

Sposób najbardziej rozpowszechniony.
Zalety:
- prostota zapisu,
- brak specyficznych konstrukcji formalnych,
- szeroki zasób słownictwa.

Wady:
- mała precyzja,
- możliwe błędy interpretacji,
- miejscami zbyt mała zwięzłość.

Jest to sformalizowany zapis algorytmów.
Zalety:
- stosowanie symboli graficznych i linii połączeń,
- łatwość interpretacji,
- łatwość wyznaczania bloków funkcjonalnych.

Wady:
- techniczny brak możliwości przedstawiania dużych algorytmów.

Sposoby zapisywania algorytmów:

1. język naturalny
2. schematy blokowe
3. języki formalne
4. funkcje rekurencyjne

Sposób najbardziej rozpowszechniony.
Zalety:
- prostota zapisu,
- brak specyficznych konstrukcji formalnych,
- szeroki zasób słownictwa.

Wady:
- mała precyzja,
- możliwe błędy interpretacji,
- miejscami zbyt mała zwięzłość.

Jest to sformalizowany zapis algorytmów.
Zalety:
- stosowanie symboli graficznych i linii połączeń,
- łatwość interpretacji,
- łatwość wyznaczania bloków funkcjonalnych.

Wady:
- techniczny brak możliwości przedstawiania dużych algorytmów.

Sposób najczęściej stosowany w praktyce.
Zalety:
- stosowanie dowolnego języka programowania lub specjalnej notacji,
- zapewnia jednoznaczność zapisu.

Sposoby zapisywania algorytmów:

1. język naturalny
2. schematy blokowe
3. języki formalne
4. funkcje rekurencyjne

Sposób najbardziej rozpowszechniony.
Zalety:
- prostota zapisu,
- brak specyficznych konstrukcji formalnych,
- szeroki zasób słownictwa.

Wady:
- mała precyzja,
- możliwe błędy interpretacji,
- miejscami zbyt mała zwięzłość.

Jest to sformalizowany zapis algorytmów.
Zalety:
- stosowanie symboli graficznych i linii połączeń,
- łatwość interpretacji,
- łatwość wyznaczania bloków funkcjonalnych.

Wady:
- techniczny brak możliwości przedstawiania dużych algorytmów.

Sposób najczęściej stosowany w praktyce.
Zalety:
- stosowanie dowolnego języka programowania lub specjalnej notacji,
- zapewnia jednoznaczność zapisu.

Sposób stosowany wszędzie tam, gdzie łatwo można przedstawić rozwiązanie problemu przy pomocy rekurencji.

Symbole używane do tworzenia schematów blokowych:

Symbol	Znaczenie
START	Symbol określa miejsce rozpoczęcia czynności związanych z algorytmem. Można zdefiniować tylko jedno takie miejsce w całym schemacie.
STOP	Symbol określa miejsce zakończenia algorytmu. Możliwe jest definiowanie wielu miejsc zakończenia.
A:= A + 10;	Blok operacyjny, definiuje czynności wykonywane na danym etapie algorytmu.
TAK A = B ? NIE	Symbol określa blok decyzyjny (rozgałęzienie).
Pisz n	Symbol określa blok wejście i wyjścia.
	Symbole połączeń między blokami (linia). Czasem po wyjściu z węzła używa się strzałki w celu określenia kierunku przepływu sterowania.

Przykład algorytmu:
Przedstawić algorytm obliczania $y=\prod_{i=1}^{n} x(i)$

Przykład algorytmu:

Przedstawić algorytm obliczania $y = \prod_{i=1}^{n} x(i)$

Rozwiązanie:

a. opis za pomocą notacji:

1. y = 1 : idź do 2
2. i = 1 : idź do 3
3. y = y *x(i) : idź do 4
4. czy i = n ? : tak_(idź do 6)
   nie_(idź do 5)
5. i = i + 1 : idź do 3
6. koniec

Przykład algorytmu:

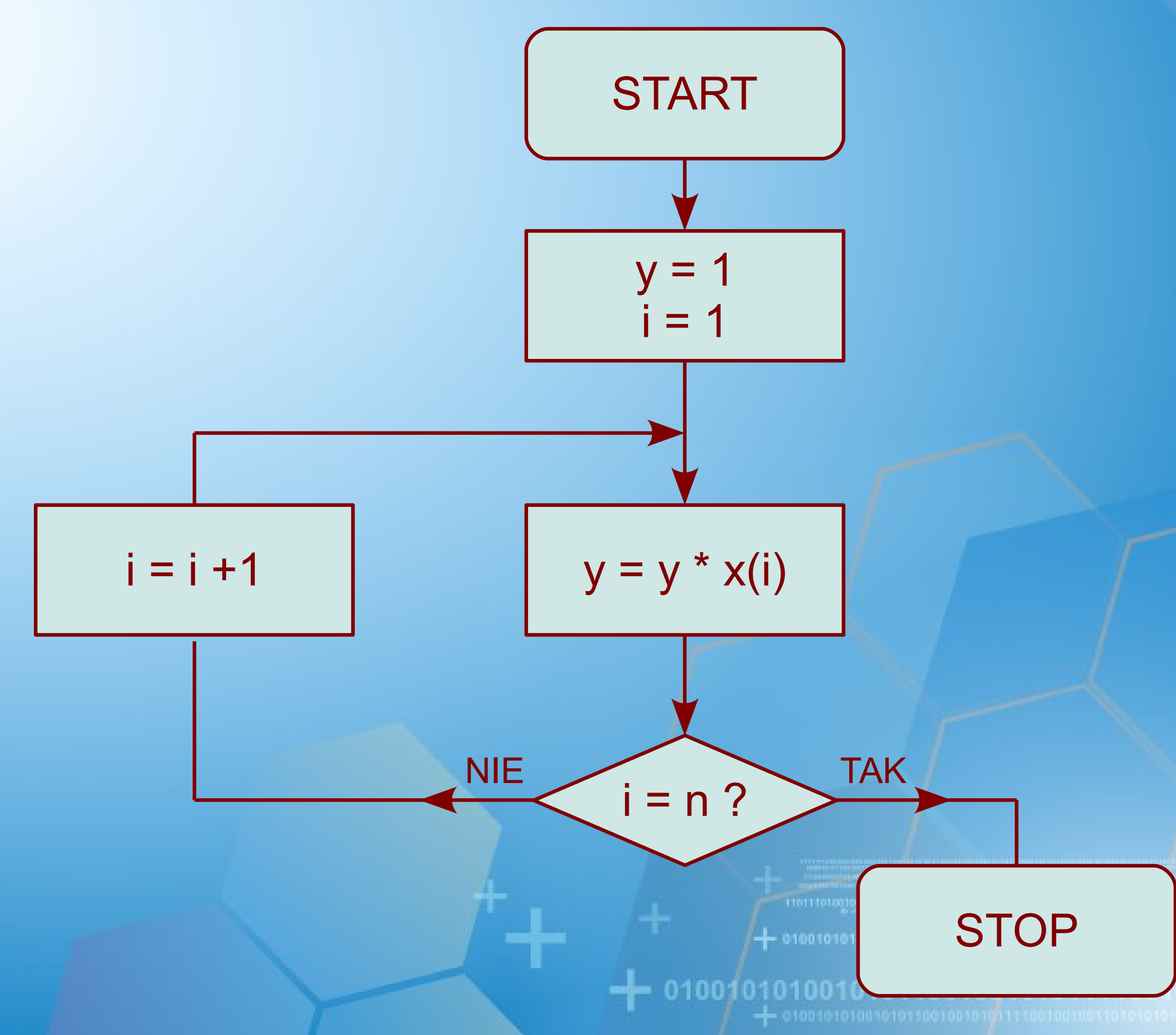
Przedstawić algorytm obliczania $y = \prod_{i=1}^{n} x(i)$

Rozwiązanie:

a. opis za pomocą notacji:

1. y = 1 : idź do 2
2. i = 1 : idź do 3
3. y = y *x(i) : idź do 4
4. czy i = n ? : tak_(idź do 6)
   nie_(idź do 5)
5. i = i + 1 : idź do 3
6. koniec

b. opis przy pomocy schematu blokowego:



Maszyna Turinga

Maszyna Turinga – jest to abstrakcyjny automat, służący do analizy algorytmów.

Maszyna Turinga składa się nieskończenie długiej taśmy, podzielonej na kratki oraz głowicy przesuwającej się nad taśmą. Głowica ma możliwość odczytywania i zapisywania symboli.

Pewnym przybliżeniem maszyny Turinga jest komputer klasy PC.

Maszyna Turinga – jest to abstrakcyjny automat, służący do analizy algorytmów.

Maszyna Turinga składa się nieskończenie długiej taśmy, podzielonej na kratki oraz głowicy przesuwającej się nad taśmą. Głowica ma możliwość odczytywania i zapisywania symboli.

Pewnym przybliżeniem maszyny Turinga jest komputer klasy PC.

Podstawowe definicje:

Zbiór symboli S={s_i}	zbiór symboli, które będą przetwarzane przez maszynę Turinga – alfabet,
Napis	ciąg symboli alfabetu S,
Głowica	rozróżnia, odczytuje i zapisuje symbole alfabetu S na taśmie, przyjmuje jeden ze stanów q_1,..., q_m,
Stan maszyny	określa stan głowicy q_j i odczytany przez nią symbol s_i – $S_{ij}=(s_i, q_j)$,
Ruch głowicy	Opisuje akcje, jaką ma wykonać głowica maszyny Turinga. Ruch głowicy jest reakcją maszyny na stan S_{ij} i jest opisany przez $R_{ij}=(s_k, K, q_l)$, gdzie: s_k – jest symbolem, który ma być zapisany przez głowicę, K – określa sposób przesunięcia głowicy, q_l – jest stanem do którego ma przejść głowica.

Tablica charakterystyczna maszyny Turinga – jest to funkcja jednoznacznie określająca ruch R_{ij} głowicy G, związany ze stanem S_{ij}.

Tablica charakterystyczna maszyny Turinga – jest to funkcja jednoznacznie określająca ruch R_{ij} głowicy G, związany ze stanem S_{ij}.

Przykład tablicy charakterystycznej:

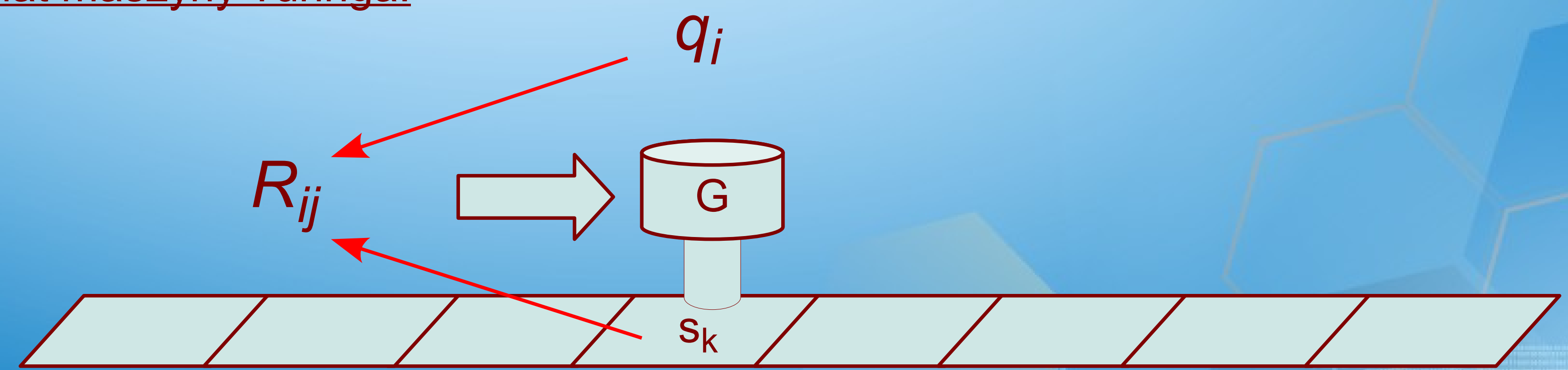
T	q_1	...	q_j	...	q_m
s_1	R_{11}	...	R_{1j}	...	R_{1m}
...	...	...	...	...	...
s_i	R_{i1}	...	R_{ij}	...	R_{im}
...	...	...	...	...	...
s_n	R_{n1}	...	R_{nj}	...	R_{nm}

Tablica charakterystyczna maszyny Turinga – jest to funkcja jednoznacznie określająca ruch R_{ij} głowicy G, związany ze stanem S_{ij}.

Przykład tablicy charakterystycznej:

T	q_1	...	q_j	...	q_m
s_1	R_{11}	...	R_{1j}	...	R_{1m}
...	...	...	...	...	...
s_i	R_{i1}	...	R_{ij}	...	R_{im}
...	...	...	...	...	...
s_n	R_{n1}	...	R_{nj}	...	R_{nm}

Schemat maszyny Turinga:



Twierdzenie:

Każdy algorytm jest deterministyczny, jeśli można go zrealizować na odpowiednio zaprogramowanej maszynie Turinga.

Przykład:

W zbiorze napisów trzyliterowych utworzonych z liter „a”, „b” i „c” poprawny jest tylko napis „abc”.Opracować algorytm rozpoznawania takiego napisu.

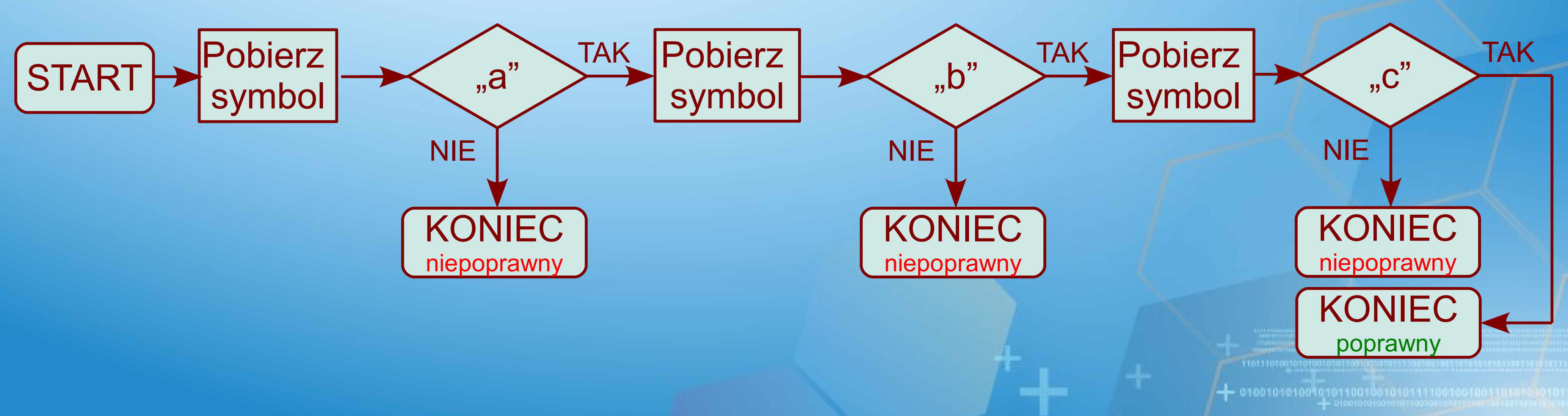
Przykład:
W zbiorze napisów trzyliterowych utworzonych z liter „a”, „b” i „c” poprawny jest tylko napis „abc”.Opracować algorytm rozpoznawania takiego napisu.

Rozwiązanie:
1. Pobierz symbol. Jeśli pobrano „a”, to pobierz następny symbol i idź do 2, w przeciwnym przypadku idź do 4.
2. Jeśli pobrany został symbol „b”, to pobierz następny symbol i przejdź do 3, w przeciwnym przypadku idź do 4.
3. Jeśli pobranym symbolem jest „c”, to idź do 5, w przeciwnym przypadku idź do 4.
4. Wyprowadź informację – napis nieprawidłowy.
5. Wyprowadź informację – napis prawidłowy.

Przykład:
W zbiorze napisów trzyliterowych utworzonych z liter „a”, „b” i „c” poprawny jest tylko napis „abc”.Opracować algorytm rozpoznawania takiego napisu.

Rozwiązanie:
1. Pobierz symbol. Jeśli pobrano „a”, to pobierz następny symbol i idź do 2, w przeciwnym przypadku idź do 4.
2. Jeśli pobrany został symbol „b”, to pobierz następny symbol i przejdź do 3, w przeciwnym przypadku idź do 4.
3. Jeśli pobranym symbolem jest „c”, to idź do 5, w przeciwnym przypadku idź do 4.
4. Wyprowadź informację – napis nieprawidłowy.
5. Wyprowadź informację – napis prawidłowy.

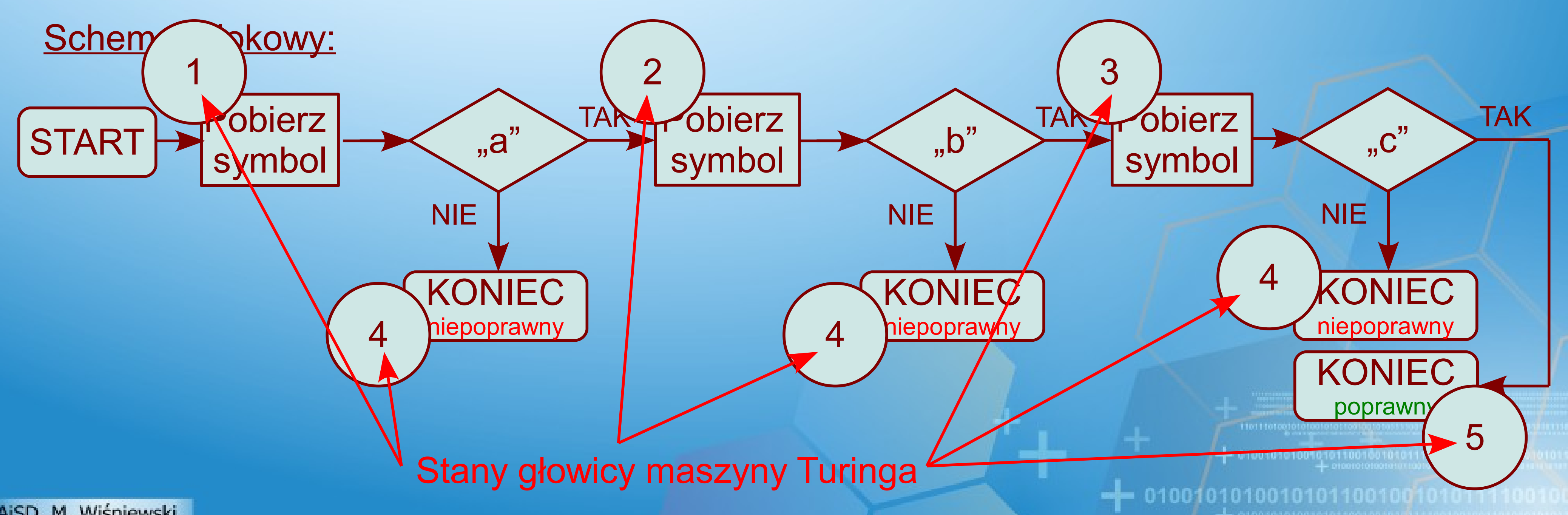
Schemat blokowy:



Przykład:
W zbiorze napisów trzyliterowych utworzonych z liter „a", „b" i „c" poprawny jest tylko napis „abc".Opracować algorytm rozpoznawania takiego napisu.

Rozwiązanie:
1. Pobierz symbol. Jeśli pobrano „a", to pobierz następny symbol i idź do 2, w przeciwnym przypadku idź do 4.
2. Jeśli pobrany został symbol „b", to pobierz następny symbol i przejdź do 3, w przeciwnym przypadku idź do 4.
3. Jeśli pobranym symbolem jest „c", to idź do 5, w przeciwnym przypadku idź do 4.
4. Wyprowadź informację – napis nieprawidłowy.
5. Wyprowadź informację – napis prawidłowy.

Schemat blokowy:



Przykład:
W zbiorze napisów trzyliterowych utworzonych z liter „a”, „b” i „c” poprawny jest tylko napis „abc”.Opracować algorytm rozpoznawania takiego napisu.

Oznaczenia:
„a”, „b”, „c” - symbole alfabetu,
P – operacja „pobierz symbol”,
N – operacja „nie rób nic”,
q_n – przejdź do stanu *n*.

Tablica charakterystyczna maszyny Turinga:

	q_1	q_2	q_3	q_4	q_5
„a”	Pq_2	Nq_4	Nq_4	Nq_4	Nq_5
„b”	Nq_4	Pq_3	Nq_4	Nq_4	Nq_5
„c”	Nq_4	Nq_4	Nq_5	Nq_4	Nq_5

Programowanie wstępujące

Programowanie wstępujące – (ang. bottom-up design) polega na określeniu podstawowych zadań, których wykonanie będzie konieczne do realizacji algorytmu.

Tak zdefiniowane zadania wykorzystuje się do budowy większych fragmentów algorytmu, aż do zbudowania modułowego rozwiązania problemu.

Ten typ projektowania stosuje się najczęściej w językach proceduralnych, gdzie Podstawowe problemy są rozwiązywane przez procedury, a cały algorytm jest realizowany przez kompletny program.

Programowanie zstępujące

Programowanie zstępujące – (ang. top-down design) polega na dekompozycji zadań wykonywanych przez algorytm na podzadania, które są łatwe do implementacji

W ogólności problem dzieli się na mniejsze podproblemy, które są następnie dzielone na kolejne podproblemy, aż do podproblemów, których rozwiązanie jest oczywiste.

Zalety programowania zstępującego:

- systematyzacja,
- modularność,
- przejrzystość kodu,
- przenośność rozwiązań.

Programowanie zstępujące – (ang. top-down design) polega na dekompozycji zadań wykonywanych przez algorytm na podzadania, które są łatwe do implementacji

W ogólności problem dzieli się na mniejsze podproblemy, które są następnie dzielone na kolejne podproblemy, aż do podproblemów, których rozwiązanie jest oczywiste.

Zalety programowania zstępującego:

- systematyzacja,
- modularność,
- przejrzystość kodu,
- przenośność rozwiązań.

Tego typu postępowanie pozwala na uwzględnienie wszelkich aspektów jakie mają być realizowane przez algorytm.

Programowanie zstępujące – (ang. top-down design) polega na dekompozycji zadań wykonywanych przez algorytm na podzadania, które są łatwe do implementacji

W ogólności problem dzieli się na mniejsze podproblemy, które są następnie dzielone na kolejne podproblemy, aż do podproblemów, których rozwiązanie jest oczywiste.

Zalety programowania zstępującego:

- systematyzacja,
- modularność,
- przejrzystość kodu,
- przenośność rozwiązań.

Tego typu postępowanie pozwala na uwzględnienie wszelkich aspektów jakie mają być realizowane przez algorytm.

Algorytm projektuje się w formie modułów, dzięki czemu pewne elementy można wykorzystać ponownie.

Programowanie zstępujące – (ang. top-down design) polega na dekompozycji zadań wykonywanych przez algorytm na podzadania, które są łatwe do implementacji

W ogólności problem dzieli się na mniejsze podproblemy, które są następnie dzielone na kolejne podproblemy, aż do podproblemów, których rozwiązanie jest oczywiste.

Zalety programowania zstępującego:

- systematyzacja,
- modularność,
- przejrzystość kodu,
- przenośność rozwiązań.

Tego typu postępowanie pozwala na uwzględnienie wszelkich aspektów jakie mają być realizowane przez algorytm.

Algorytm projektuje się w formie modułów, dzięki czemu pewne elementy można wykorzystać ponownie.

Dzięki modularności kod staje się czytelny – cecha pożądana w przypadku pracy zespołowej.

Programowanie zstępujące – (ang. top-down design) polega na dekompozycji zadań wykonywanych przez algorytm na podzadania, które są łatwe do implementacji

W ogólności problem dzieli się na mniejsze podproblemy, które są następnie dzielone na kolejne podproblemy, aż do podproblemów, których rozwiązanie jest oczywiste.

Zalety programowania zstępującego:

- systematyzacja,
- modularność,
- przejrzystość kodu,
- przenośność rozwiązań.

Tego typu postępowanie pozwala na uwzględnienie wszelkich aspektów jakie mają być realizowane przez algorytm.

Algorytm projektuje się w formie modułów, dzięki czemu pewne elementy można wykorzystać ponownie.

Dzięki modularności kod staje się czytelny – cecha pożądana w przypadku pracy zespołowej.

Opracowane moduły najczęściej można łatwo dostosować do podobnych problemów.

Iteracja

Iteracja – jednokrotne wykonanie jednego z elementów algorytmu.

Realizacja iteracji:

a. w języku C:

- przy pomocy pętli **for**,
- przy pomocy pętli **while**,
- przy pomocy pętli **do..while**.

Iteracja – jednokrotne wykonanie jednego z elementów algorytmu.

Realizacja iteracji:

a. w języku C:
- przy pomocy pętli **for**,
- przy pomocy pętli **while**,
- przy pomocy pętli **do..while**.

b. w języku Pascal:
- przy pomocy pętli **for**,
- przy pomocy pętli **while...do**,
- przy pomocy pętli **repeat..until**.

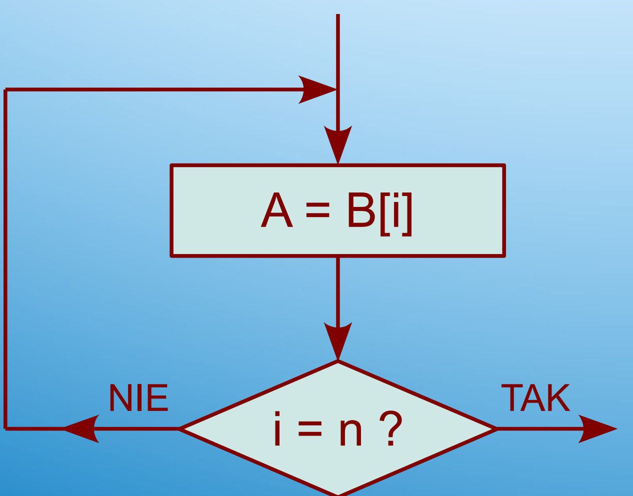
Iteracja – jednokrotne wykonanie jednego z elementów algorytmu.

Realizacja iteracji:

a. w języku C:
- przy pomocy pętli **for**,
- przy pomocy pętli **while**,
- przy pomocy pętli **do..while**.

b. w języku Pascal:
- przy pomocy pętli **for**,
- przy pomocy pętli **while...do**,
- przy pomocy pętli **repeat..until**.

c. w schemacie blokowym:



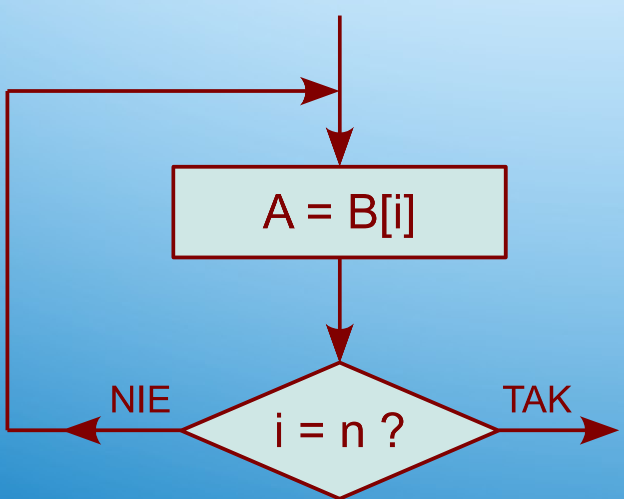
Iteracja – jednokrotne wykonanie jednego z elementów algorytmu.

Realizacja iteracji:

a. w języku C:
- przy pomocy pętli **for**,
- przy pomocy pętli **while**,
- przy pomocy pętli **do..while**.

b. w języku Pascal:
- przy pomocy pętli **for**,
- przy pomocy pętli **while...do**,
- przy pomocy pętli **repeat..until**.

c. w schemacie blokowym:



d. w opisie słownym:
- przez wskazanie kroku do którego ma zostać przekazane sterowanie.

Rozejście / rozgałęzienie

Rozejście lub **rozgałęzienie –** zmiana sterowania w algorytmie, czyli wykonanie innych, alternatywnych czynności po pojawieniu się określonych danych lub okoliczności.

W językach programowania do realizacji rozgałęzień wykorzystuje się instrukcję warunkową **if** lub instrukcję wyboru **case**.

Rozejście lub **rozgałęzienie –** zmiana sterowania w algorytmie, czyli wykonanie innych, alternatywnych czynności po pojawieniu się określonych danych lub okoliczności.

W językach programowania do realizacji rozgałęzień wykorzystuje się instrukcję warunkową **if** lub instrukcję wyboru **case**.

Przykład:

```
if liczba = 100 then
  // czynności dla tego przypadku
else
  if liczba > 100 then
    if liczba < 200 then
      // czynności, gdy liczba większa 100
      // i jednocześnie mniejsza od 200
    else
      // czynności, gdy liczba większa lub
      // równa 200
  else
    // czynności, gdy liczba mniejsza lub
    // równa 100
```

Rozejście lub **rozgałęzienie –** zmiana sterowania w algorytmie, czyli wykonanie innych, alternatywnych czynności po pojawieniu się określonych danych lub okoliczności.

W językach programowania do realizacji rozgałęzień wykorzystuje się instrukcję warunkową **if** lub instrukcję wyboru **case**.

Przykład:

```
if liczba = 100 then
  // czynności dla tego przypadku
else
  if liczba > 100 then
    if liczba < 200 then
      // czynności, gdy liczba większa 100
      // i jednocześnie mniejsza od 200
    else
      // czynności, gdy liczba większa lub
      // równa 200
  else
    // czynności, gdy liczba mniejsza lub
    // równa 100
```

```
case liczba of
0..99:
   // liczba mniejsza lub równa 100
  break;
100:
  // liczba równa 100
  break;
101..199:
  // liczba większa do 100 i mniejsza od 200
  break;
else
  // pozostałe przypadki
end;
```

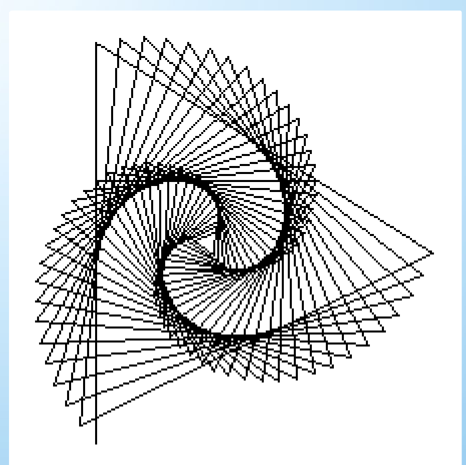
Rekurencja

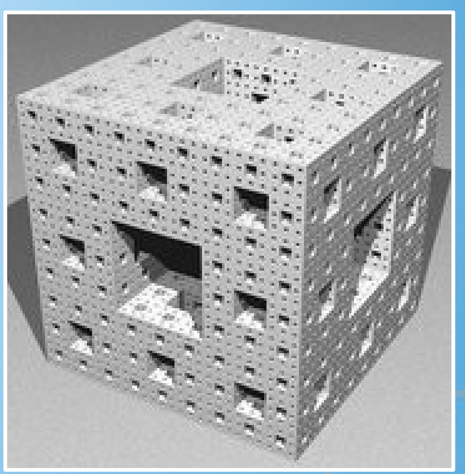
Rekurencja – polega na odwołaniu się funkcji do samej siebie.

Rekurencja jest jedną z podstawowych technik programowania. Ze względu na własności programowanie rekurencyjne może ułatwić rozwiązanie pewnej klasy zadań.

Rekurencja – polega na odwołaniu się funkcji do samej siebie.

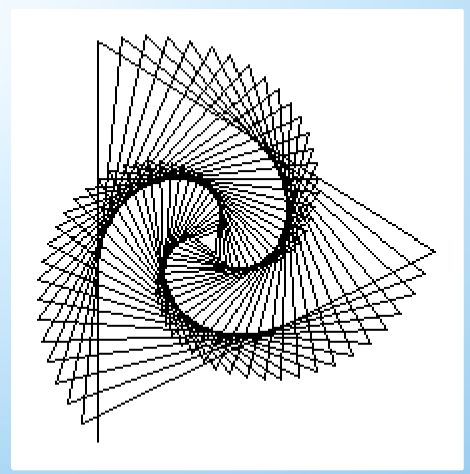
Rekurencja jest jedną z podstawowych technik programowania. Ze względu na własności programowanie rekurencyjne może ułatwić rozwiązanie pewnej klasy zadań.

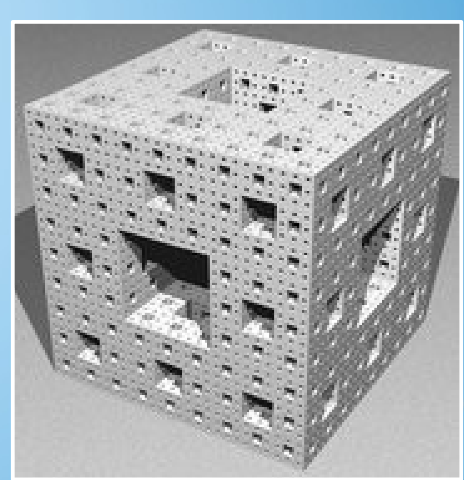




Rekurencja – polega na odwołaniu się funkcji do samej siebie.

Rekurencja jest jedną z podstawowych technik programowania. Ze względu na własności programowanie rekurencyjne może ułatwić rozwiązanie pewnej klasy zadań.



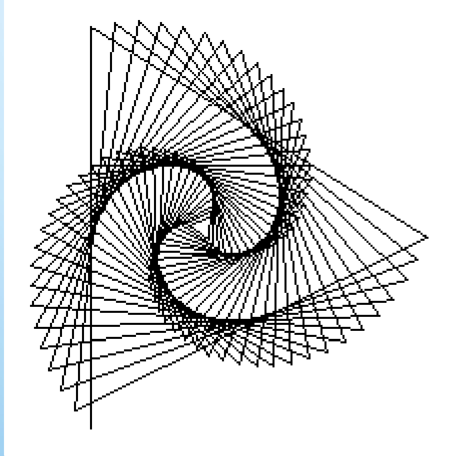


Niebezpieczeństwa związane z rekurencją:

- możliwe wielokrotne wykonywanie tych samych obliczeń,
- nieprzejrzystość kodu algorytmu,
- nieskończone wykonywanie kodu – błędny warunek zakończenia,
- możliwe większe zużycie pamięci – każde wywołanie funkcji powoduje alokacje pewnej ilości pamięci.

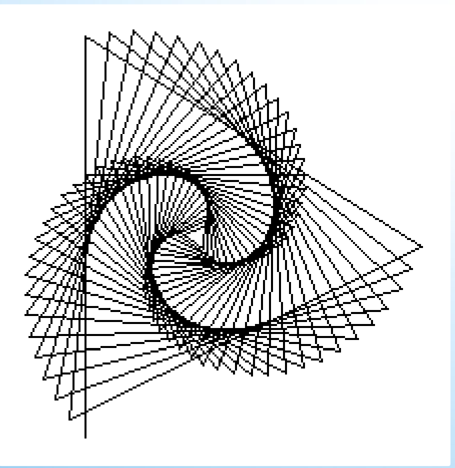
Rekurencja – polega na odwołaniu się funkcji do samej siebie.

Przykład:



Rekurencja – polega na odwołaniu się funkcji do samej siebie.

Przykład:



Algorytm rekurencyjny:

krok 1: rysuj trójkąt,

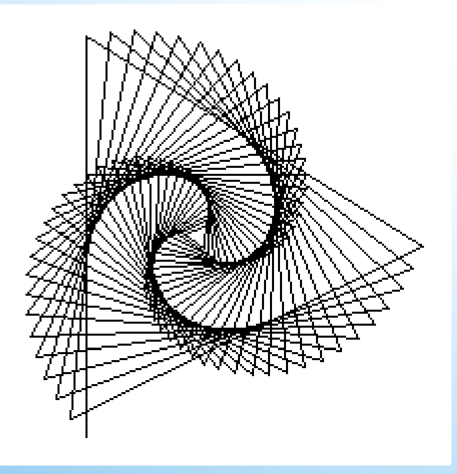
krok 2: zastosuj przekształcenie skalujące współrzędne trójkąta, powodujące zmniejszenie jego powierzchni o 5%,

krok 3: zastosuj przekształcenie obracające współrzędne trójkąta o 5 stopni względem środka trójkąta,

krok 4: wykonaj algorytm od kroku 1.

Rekurencja – polega na odwołaniu się funkcji do samej siebie.

Przykład:



Algorytm rekurencyjny:
krok 1: rysuj trójkąt,
krok 2: zastosuj przekształcenie skalujące współrzędne trójkąta, powodujące zmniejszenie jego powierzchni o 5%,
krok 3: zastosuj przekształcenie obracające współrzędne trójkąta o 5 stopni względem środka trójkąta,
krok 4: wykonaj algorytm od kroku 1.

Algorytm rekurencyjny z zakończeniem:
krok 1: przypisz poziomowi wartość 1,
krok 2: rysuj trójkąt,
krok 3: zastosuj przekształcenie skalujące współrzędne trójkąta, powodujące zmniejszenie jego powierzchni o 5%,
krok 4: zastosuj przekształcenie obracające współrzędne trójkąta o 5 stopni względem środka trójkąta,
krok 5: jeśli poziom jest mniejszy niż 20, to zwiększ poziom o 1,
krok 6: jeśli poziom jest równy 20 to zakończ algorytm,
krok 7: wykonaj algorytm od kroku 2.

Koniec wykładu