

Wstęp do Informatyki i Programowania

Jacek Cichoń Przemysław Kobylański

Katedra Informatyki W11/K2
Politechnika Wrocławска



Politechnika Wrocławskiego

Plan wykładu

- 1 Algorytmy i programy
- 2 Projektowanie algorytmów
- 3 Elementy języka C
- 4 Podstawowe typy danych w C
- 5 Czasowa złożoność obliczeniowa
- 6 Rekurencja
- 7 Zasada dziel i zwyciężaj
- 8 Przeszukiwanie z nawrotami
- 9 Obliczenia na stosie
- 10 Programowanie dynamiczne
- 11 Systemy regułowe
- 12 Automaty skończone
- 13 Rozstrzygalność i obliczalność
- 14 Nierozstrzygalność i nieobliczalność
- 15 Klasy złożoności

Literatura

- Lektura podstawowa

- ① S. Alagić, M.A. Arbib. **Projektowanie programów poprawnych i dobrze zbudowanych.** WNT, Warszawa, 1982.
- ② D. Harel, Y. Feldman. **Rzecz o istocie informatyki.** WNT, Warszawa, 2008.
- ③ B.W. Kernighan, D.M. Ritchie. **Język ANSI C.** WNT, Warszawa, 2002.

- Lektura uzupełniająca

- ① M. Kotowski. **Wysokie C.** LUPUS, Warszawa, 1998.
- ② A. Hunt, D. Thomas. **Pragmatyczny programista. Od czeladnika do mistrza.** WNT, Warszawa, 2002.

Wykład 1

Algorytmy i programy

Algorytmy + struktury danych = programy

Niklaus Wirth

Algorytm = logika + sterowanie

Robert Kowalski

Algorytmy i programy

- algorytm
- program
- język programowania
- interpreter
- kompilator
- języki algorytmiczne
- języki deklaratywne

Algorytmy i programy

Algorytm

Algorytm = opis sposobu postępowania



Abū ‘Abdallāh Muhammād ibn Mūsā al-Khwārizmī (780-850 n.e., Persja)

Algorytmy i programy

Program

Program = formalny zapis algorytmu

Program składa się z ciągu operacji jakie ma wykonać komputer.

Algorytmy i programy

Język programowania

- program zapisany jest w języku programowania
- każdy komputer zna tylko jeden język programowania – swój własny język maszynowy
- wykaz ok. 2500 języków programowania:
<http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>
- aby komputer mógł wykonać program w innym języku niż maszynowy potrzebna jest interpretacja programu albo jego kompilacja na język maszynowy

Algorytmy i programy

Interpreter

Przykład w języku BASIC¹:

```
10 FOR I=1 TO 10  
20 PRINT I  
30 NEXT I
```

Instrukcja PRINT I jest analizowana dziesięciokrotnie.

¹ Język programowania opracowany przez Johna Kemeny i Thomasa Kurtza w roku 1964. Jego nazwa jest akronimem **B**eginner's **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode.

Algorytmy i programy

Interpreter

Przykłady zastosowania interpreterów:

- powłoki systemów operacyjnych (np. bash w Linuxie),
- języki skryptowe (np. PHP i Javascript),
- języki operujące na napisach (np. Perl)
- języki obiektowe z dynamicznym typowaniem (np. Python i Ruby),
- kalkulatory programowalne

Algorytmy i programy

Kompilator

Translator = program tłumaczący język programowania na język maszynowy

Kompilator = program konsolidujący przekład z funkcjami bibliotecznymi

Algorytmy i programy

Języki algorytmiczne

Algorytmy + struktury danych = programy

Język algorytmiczny = język programowania, w którym wyraża się algorytm

Główne cechy:

- zmienne będące miejscem przechowywania danych
- instrukcja przypisania
- zestaw instrukcji sterujących (np. warunkowe oraz iteracji)

Algorytmy i programy

Języki algorytmiczne: Fortran

FORmula TRANslator (1954; najnowszy standard 2008)

```
DO 10 I=1, 10
      PRINT 2, I
10      CONTINUE
```

Algorytmy i programy

Języki algorytmiczne: Fortran (przykład z sondy Mariner 1)

```
IF (TVAL .LT. 0.2E-2) GOTO 40
DO 40 M = 1, 3
W0 = (M-1)*0.5
X = H*1.74533E-2*W0
DO 20 NO = 1, 8
EPS = 5.0*10.0**(-NO-7)
CALL BESJ(X, 0, B0, EPS, IER)
IF (IER .EQ. 0) GOTO 10
20 CONTINUE
DO 5 K = 1, 3
T(K) = W0
Z = 1.0/(X**2)*B1**2+3.0977E-4*B0**2
D(K) = 3.076E-2*2.0*(1.0/X*B0*B1+3.0977E-4*
*(B0**2-X*B0*B1))/Z
E(K) = H**2*93.2943*W0/SIN(W0)*Z
H = D(K)-E(K)
5 CONTINUE
10 CONTINUE
Y = H/W0-1
40 CONTINUE
```

Algorytmy i programy

Języki algorytmiczne: C

Dennis Ritchie z AT&T Bell Laboratories (1971)

```
for(i=1; i <= 10; i++)
    printf("%d\n", i);
```

Algorytmy i programy

Języki algorytmiczne: C (przykład literówki)

Przykład programu zawierającego błąd:

```
x = 10.0;  
while(x > 2,3)  
{  
    printf("%f\n", x);  
    x = x - 0.1;  
}
```

Algorytmy i programy

Języki algorytmiczne: Ada

Język na zamówienie Departamentu Obrony Stanów Zjednoczonych (1979; najnowszy standard 2012)

```
for I in 1..10 loop
    put(I);
    new_line;
end loop;
```

Samolot Boeing 777 jest w 99% oprogramowany w języku Ada:

<http://archive.adaic.com/projects/atwork/boeing.html>

Algorytmy i programy

Języki algorytmiczne: Ada 2012 (przykład weryfikacji poprawności kodu)

$$|x| = \begin{cases} -x & \text{gdy } x < 0 \\ x & \text{gdy } x \geq 0 \end{cases}$$

Funkcja w C z biblioteki **glibc** obliczająca wartość bezwzględna liczby całkowitej:

```
int absolute(int x)
{
    return x < 0 ? -x : x;
}
```

Czy powyższa funkcja jest poprawna?

Algorytmy i programy

Języki algorytmiczne: Ada 2012 (przykład weryfikacji poprawności kodu)

```
package absolute is
    function av(x: in Integer) return Integer with
        post => av'Result >= 0;
end absolute;
package body absolute is
    function av(x: in Integer) return Integer is
        begin
            return (if x < 0 then -x else x);
        end av;
end absolute;
```

Wynik analizy programem **GNATprove**:

```
analyzing absolute.av, 2 checks
absolute.adb:6:29: overflow check not proved [overflow_check]
absolute.ads:4:14: postcondition not proved, requires av'Result >= 0 [postcondition]
```

Algorytmy i programy

Języki algorytmiczne: Python

Interpretowany język programowania obiektowego z dynamicznym typowaniem (1991)

```
for i in range(10):  
    print(i)
```

Powyższa pętla drukuje wartości od 0 do 9 gdyż właśnie taki zakres jest wartością funkcji range(10).

Algorytmy i programy

Języki deklaratywne

Język deklaratywny = język programowania, w którym opisuje się problem

Algorytm = logika + sterowanie

Algorytmy i programy

Języki deklaratywne: SQL

Język zapytań do relacyjnych baz danych (lata 70-te XX wieku)

```
SELECT CompanyName, ContactName FROM customers WHERE  
    CompanyName LIKE 'a%'
```

W wyniku wykonania powyższej instrukcji zostają odnalezione w bazie **customers** informacje o **CompanyName** i **ContactName**, przy czym nazwa firmy powinna zaczynać się od litery a.

Algorytmy i programy

Języki deklaratywne: Prolog

PROgramming in LOGic (Alain Colmerauer 1972)

Język programowania logicznego:

```
lubi(przemko, programowanie).
```

```
lubi(przemko, X) :- lubi(X, programowanie).
```

Na pytanie co lubi kogo lubi Przemko są dwie odpowiedzi:

```
?- lubi(przemko, X).
```

```
X = programowanie ;
```

```
X = przemko ;
```

```
no
```

Z powodu przyjętej w Prologu **zasady zamkniętego świata** nie ma innych odpowiedzi, bo prawdziwe jest tylko to, co wyrażono w programie.

Wykład 2

Projektowanie algorytmów

It is easier to write an incorrect program than understand a correct one.

Alan J. Perlis „Epigrams in Programming”

Projektowanie algorytmów

- schemat blokowy
- pseudokod
- poprawność algorytmu
- projektowanie zstępujące

Projektowanie algorytmów

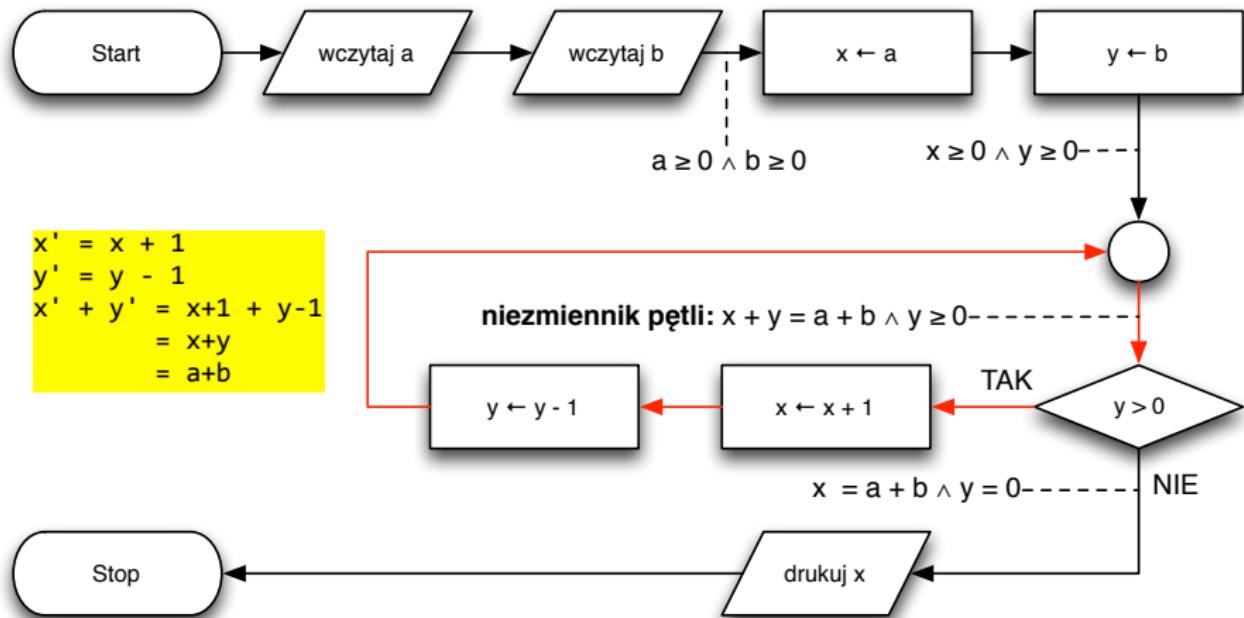
Schemat blokowy

Element schematu	Znaczenie	Strzałki dochodzące	Strzałki wychodzące
	początek schematu	0	1
	koniec schematu	1	0
	operacja wejścia/wyjścia	1	1
	przetwarzanie danych	1	1
	warunkowe rozwidlenie	1	2
	połączenie ścieżek	2 lub więcej	1

Projektowanie algorytmów

Schemat blokowy

Example (Suma liczb naturalnych)



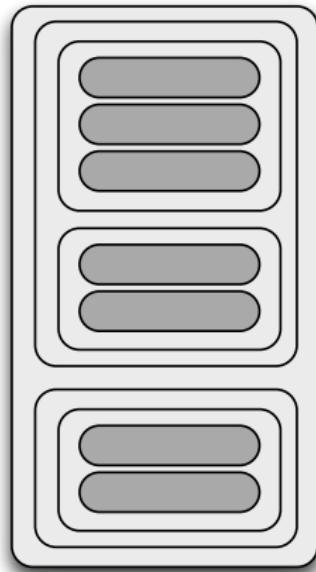
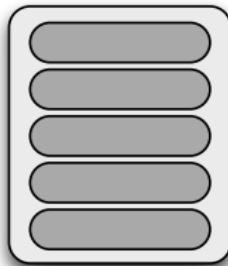
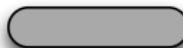
Projektowanie algorytmów

Programowanie strukturalne

- *Programowanie strukturalne* opiera się na zasadzie, że w programie można wyróżnić strukturę zagnieżdżonych w sobie jednostek programowych.
- Wielkim propagatorem programowania strukturalnego jest Niklaus Wirth z Politechniki Federalnej w Zurychu (twórca takich języków programowania jak Pascal i Modula).

Projektowanie algorytmów

Programowanie strukturalne



a)

b)

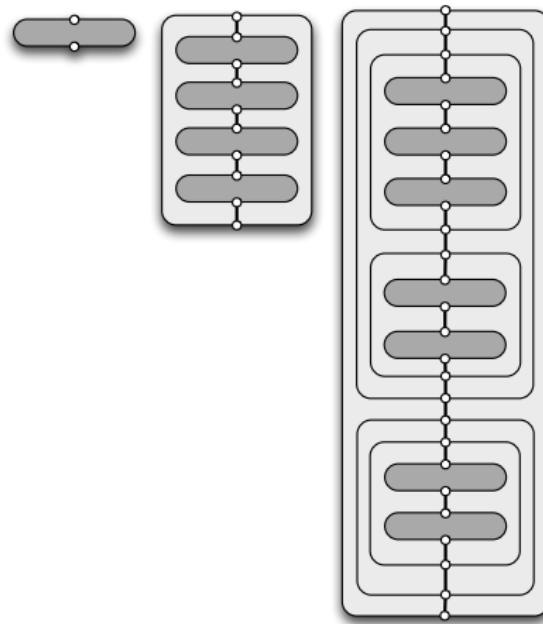
c)

- a) prosta instrukcja b) instrukcja złożona c) instrukcja jeszcze bardziej złożona

Projektowanie algorytmów

Programowanie strukturalne

Każda instrukcja (prosta i złożona) ma jeden punkt wejścia i jeden punkt wyjścia:



Projektowanie algorytmów

Pseudokod: zmienne

- ① Wartość może być zapisywana w zmiennej dzięki instrukcji podstawienia $x \leftarrow W$, gdzie x jest nazwą zmiennej a W jest wyrażeniem, którego wartość zostanie zapisana w zmiennej.
- ② Aktualna wartość zmiennej może być pobrana i użyta poprzez użycie nazwy zmiennej w wyrażeniu.
- ③ Zapisanie wartości w zmiennej niszczy bezpowrotnie poprzednią jej wartość.
- ④ Zmienne mają nieokreślone początkowe wartości dlatego użycie ich bez wcześniejszego podstawienia uważa się za błędne.

Projektowanie algorytmów

Pseudokod: tablice

- Jeśli program przetwarza dużą liczbę danych, szczególnie gdy ich liczba nie jest z góry znana i zostaje podana w danych do programu, wówczas wygodnie jest użyć tablice.
- Niech t będzie tablicą o pięciu elementach. Kolejne elementy tablicy będziemy oznaczać jako $t[1]$, $t[2]$, $t[3]$, $t[4]$ i $t[5]$.

Projektowanie algorytmów

Pseudokod: tablice

Example (Indeksowanie tablicy)

Niech t będzie pięcioelementową tablicą o następujących wartościach:

i	1	2	3	4	5
$t[i]$	2	1	5	3	4

Dla $i = 2$ mamy następujące wartości elementów:

$$t[i] = t[2] = 1$$

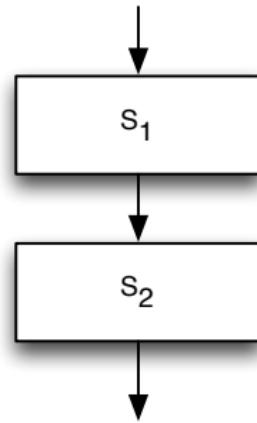
$$t[t[i]] = t[t[2]] = t[1] = 2$$

$$t[t[i] + 3] = t[t[2] + 3] = t[1 + 3] = t[4] = 3$$

Projektowanie algorytmów

Pseudokod: sekwencja instrukcji

Schemat blokowy dwóch instrukcji S_1 i S_2 :



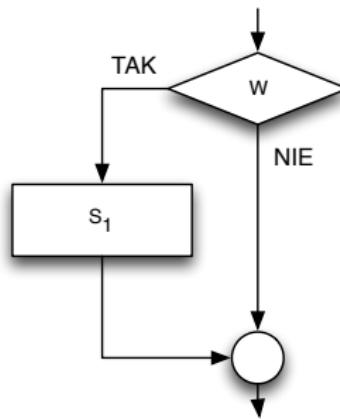
Pseudokod:

- 1: S_1 ;
- 2: S_2

Projektowanie algorytmów

Pseudokod: instrukcja warunkowa

Warunkowe wykonanie instrukcji S_1 gdy spełniony jest warunek W :



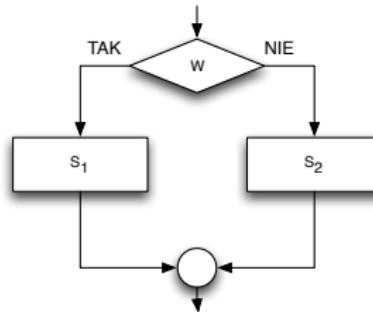
Pseudokod:

```
1: if  $W$  then  
2:      $S_1$   
3: end if
```

Projektowanie algorytmów

Pseudokod: instrukcja warunkowa

Warunkowe wykonanie instrukcji S_1 , gdy spełniony jest warunek W , oraz S_2 w przeciwnym przypadku:



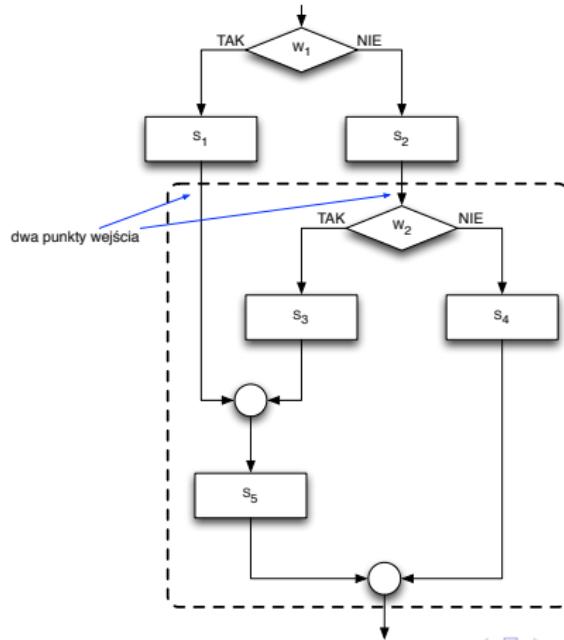
Pseudokod:

```
1: if W then  
2:     S1  
3: else  
4:     S2  
5: end if
```

Projektowanie algorytmów

Pseudokod: instrukcja warunkowa

Nie każdy schemat blokowy da się zapisać bez powtórzeń w postaci instrukcji strukturalnej:



Projektowanie algorytmów

Pseudokod: instrukcja warunkowa

Gdy zapisze się w postaci strukturalnej:

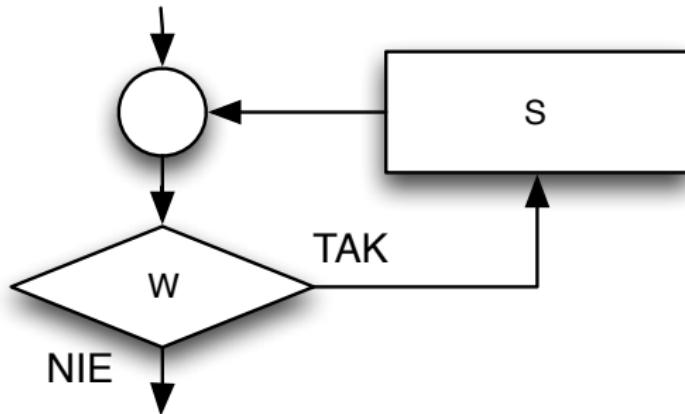
```
1: if  $W_1$  then  
2:    $S_1$ ;  
3:    $S_5$   
4: else  
5:    $S_2$ ;  
6:   if  $W_2$  then  
7:      $S_3$ ;  
8:      $S_5$   
9:   else  
10:     $S_4$   
11: end if  
12: end if
```

instrukcja S_5 występuje dwukrotnie.

Projektowanie algorytmów

Pseudokod: instrukcja pętli

Powtarzanie instrukcji S dopóki spełniony jest warunek W :



Pseudokod:

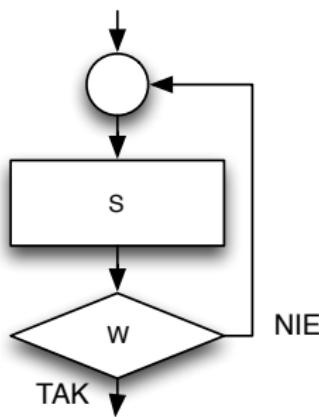
```
1: while W do  
2:   S  
3: end while
```

▷ może nie wykonać się ani razu

Projektowanie algorytmów

Pseudokod: instrukcja pętli

Powtarzanie instrukcji S aż spełniony będzie warunek W :



Pseudokod:

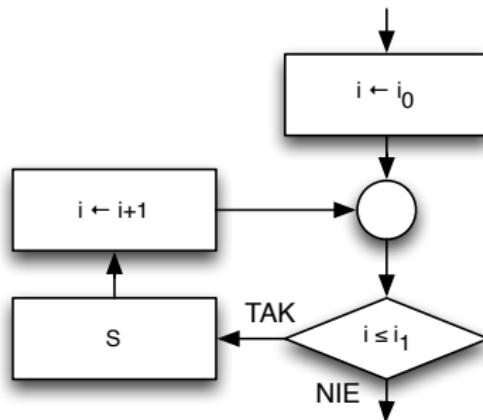
```
1: repeat  
2:   S  
3: until W
```

▷ co najmniej raz jest wykonywana

Projektowanie algorytmów

Pseudokod: instrukcja pętli

Powtarzanie wykonania instrukcji S dla wartości zmiennej sterującej $i = i_0, i_0 + 1, \dots, i_1$:



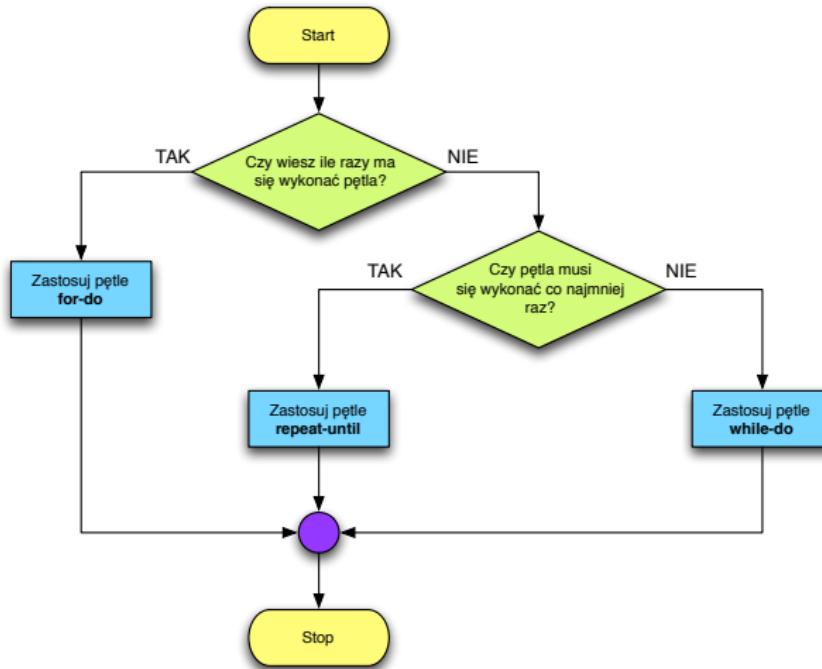
Pseudokod:

```
1: for  $i \leftarrow i_0, i_1$  do
2:    $S$ 
3: end for
```

Projektowanie algorytmów

Pseudokod: instrukcja pętli

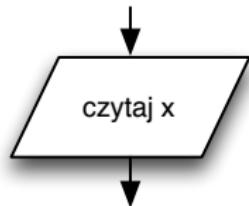
Która pętlę zastosować?



Projektowanie algorytmów

Pseudokod: operacje wejścia i wyjścia

Przeczytanie wartości i umieszczenie jej w zmiennej:



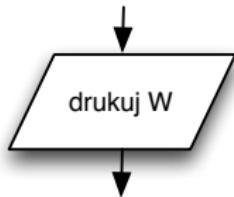
Pseudokod:

1: **read x**

Projektowanie algorytmów

Pseudokod: operacje wejścia i wyjścia

Wyliczenie wartości wyrażenia W i wydrukowanie jej:



Pseudokod:

1: **write** W

Projektowanie algorytmów

Pseudokod: funkcje

Pseudokod:

```
1: function f(x, y, ...)  
2:     S  
3:     f ← W  
4: end function
```

Projektowanie algorytmów

Pseudokod: funkcje

Example (Podnoszenie do kwadratu)

```
1: function kwadrat(x)
2:     y ← x * x
3:     KWADRAT ← y
4: end function
5:
6: x ← 2;
7: y ← 3;
8: z ← KWADRAT(x) + KWADRAT(y);           ▷  $z = x^2 + y^2 = 13$ 
9: z ← KWADRAT(x + y)                      ▷  $z = x^2 + 2xy + y^2 = 25$ 
```

Projektowanie algorytmów

Pseudokod: funkcje

Example (Błędne wywołanie funkcji)

Funkcja powinna być wywołana w kontekście wyrażenia a nie instrukcji, jak w poniższym fragmencie kodu:

1: KWADRAT(1)

Projektowanie algorytmów

Pseudokod: procedury

Pseudokod:

```
1: procedure p(x, y, ...)  
2:     S  
3: end procedure
```

Projektowanie algorytmów

Pseudokod: procedury

Example (Przykład procedury i jej wywołania)

```
1: procedure operacje(x, y)
2:   write x + y;
3:   write x - y;
4:   write x * y;
5:   write x/y
6: end procedure
7:
8: OPERACJE(2,4);
9: OPERACJE(4,2)
```

W wyniku wykonania powyższego programu zostaną wydrukowane kolejno następujące wartości:

6 -2 8 0.5 6 2 8 2

Projektowanie algorytmów

Pseudokod: procedury

Example (Przykład błędnego wywołania procedury)

Procedura powinna być wywołana w kontekście instrukcji a nie wyrażenia, jak w poniższym fragmencie kodu:

- 1: $x \leftarrow \text{OPERACJE}(2, 4);$
- 2: $\text{OPERACJE}(4, \text{OPERACJE}(2, 4))$

Projektowanie algorytmów

Pseudokod: procedury

Example (Przykład wywołania procedury)

```
1: procedure P(x)
2:     write x; Q(x + 1); R(x + 1); write x
3: end procedure
4: procedure Q(x)
5:     write x; R(x + 1); write x
6: end procedure
7: procedure R(x)
8:     write x
9: end procedure
```

W wyniku wywołania $P(1)$ zostaną wydrukowane kolejno następujące liczby:

1 2 3 2 2 1

Projektowanie algorytmów

Pseudokod: przekazywanie parametrów

- We wszystkich dotychczas rozpatrywanych przykładach, za pomocą parametrów dostarczane były do procedury wartości.
- Jeśli wartość parametru procedury, podczas jej działania, zmieniała się, to ta zmiana nie miała żadnego efektu na zewnątrz procedury.

1: **procedure** P(x)

2: $x \leftarrow x + 1$

3: **end procedure**

4:

5: $y \leftarrow 1;$ $\triangleright y = 1$

6: P(y)

$\triangleright y = 1$

$\triangleright y = 1$

Projektowanie algorytmów

Pseudokod: przekazywanie parametrów

- W pewnych sytuacjach chcielibyśmy aby zmiany wartości parametru w procedurze miały efekty na zewnątrz, w miejscu jej wywołania.
 - Wówczas przed nazwą parametru podawać będziemy dodatkowe słowa **in** lub **out** oznaczające, że parametr taki nie tylko dostarcza dane (jest wejściowy **in**) ale również oddaje wyniki (jest wyjściowy **out**).

1: procedure Q(**in out** x)

2: $x \leftarrow x + 1$

3: end procedure

4:

5: $y \leftarrow 1;$

▷ $y = 1$

6: $Q(y)$

▷ $y = 2$

Projektowanie algorytmów

Pseudokod: przekazywanie parametrów

- Jeśli parametr nie wymaga dostarczenia wartości, bo jest wyliczany podczas działania procedury, a jego wartość oddawana jest na zewnątrz, to taki parametr będziemy nazywać parametrem wyjściowym i deklarować poprzedzając go jednym słowem **out**.

```
1: procedure R(out x)
2:     x ← 1
3: end procedure
4:                                     ▷ zmienna y nie jest ustalona
5: R(y)                                ▷  $y = 1$ 
```

Projektowanie algorytmów

Pseudokod: przykłady programów

Example (Dodawanie dwóch liczb naturalnych)

Require: dwie liczby naturalne a i b

Ensure: suma $a + b$ w zmiennej x

```
1:  $x \leftarrow a;$ 
2:  $y \leftarrow b;$                                       $\triangleright x \geq 0 \wedge y \geq 0$ 
3: while  $y > 0$  do                          $\triangleright$  niezmiennik:  $x + y = a + b \wedge y \geq 0$ 
4:      $x \leftarrow x + 1;$ 
5:      $y \leftarrow y - 1$ 
6: end while                                      $\triangleright x = a + b \wedge y = 0$ 
```

Projektowanie algorytmów

Pseudokod: przykłady programów

Example (Mnożenie dwóch liczb naturalnych)

Require: dwie liczby naturalne a i b

Ensure: iloczyn $a \cdot b$ w zmiennej x

```
1:  $x \leftarrow 0;$ 
2:  $y \leftarrow b;$ 
3: while  $y > 0$  do                                ▷ ćwiczenie: jaki niezmiennik?
   4:      $z \leftarrow a;$ 
   5:     while  $z > 0$  do                      ▷ ćwiczenie: jaki niezmiennik?
       6:          $x \leftarrow x + 1;$ 
       7:          $z \leftarrow z - 1$ 
   8:     end while
   9:      $y \leftarrow y - 1$ 
10: end while
```

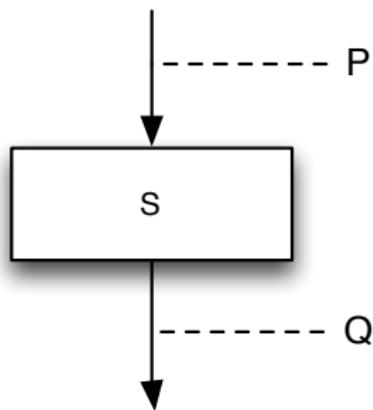
Projektowanie algorytmów

Poprawność algorytmu

- W komentarzach umieszczać warunki jakie spełniają wartości zmiennych w danym miejscu.
- Pokazać wynikanie kolejnych warunków stosując odpowiednie schematy wnioskowania.

Projektowanie algorytmów

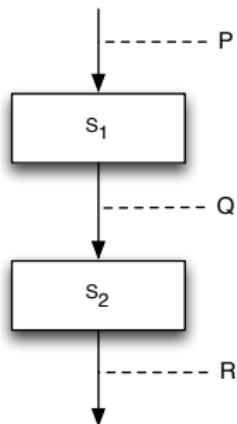
Poprawność algorytmu: częściowa poprawność



- **Częściowa poprawność:** Jeśli przed wykonaniem instrukcji S spełniony jest warunek wstępny P i instrukcja S kończy działanie, to po wykonaniu instrukcji S spełniony jest warunek końcowy Q .
- Dla pokazania **pełnej poprawności** konieczne byłoby udowodnienie jeszcze, że jeśli dane spełniają warunek P , to instrukcja S skończy działanie.

Projektowanie algorytmów

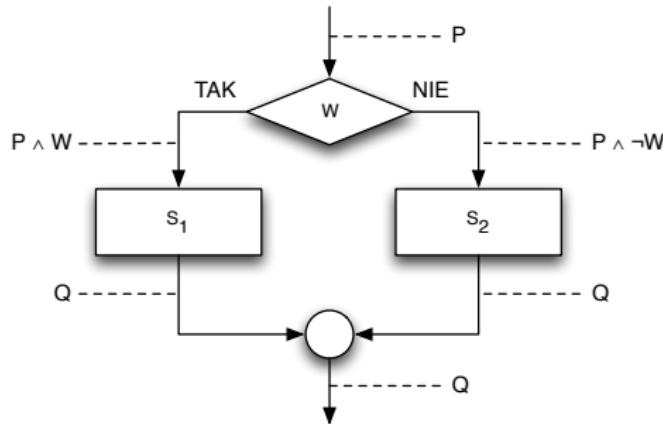
Poprawność algorytmu: sekwencja instrukcji



- Warunek P jest warunkiem wstępny dla instrukcji S_1 , natomiast warunek Q jest jej warunkiem końcowym.
- Warunek Q jest warunkiem wstępny dla instrukcji S_2 , natomiast warunek R jest jej warunkiem końcowym.
- Warunek P jest warunkiem wstępny dla sekwencji instrukcji S_1 i S_2 , natomiast warunek R jest warunkiem końcowym dla tej sekwencji.

Projektowanie algorytmów

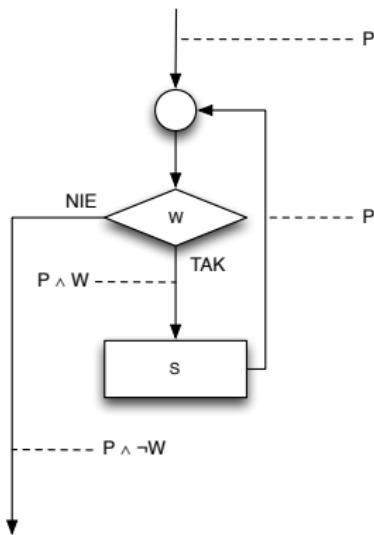
Poprawność algorytmu: instrukcja warunkowa



- Warunek P jest warunkiem wstępny dla instrukcji ***if W then S₁ else S₂ end if***, natomiast warunek Q jest jej warunkiem końcowym.
- Warunek $P \wedge W$ jest warunkiem wstępny dla instrukcji S_1 , natomiast Q jest jej warunkiem końcowym.
- Warunek $P \wedge \neg W$ jest warunkiem wstępny dla instrukcji S_2 , natomiast Q jest jej warunkiem końcowym.

Projektowanie algorytmów

Poprawność algorytmu: instrukcja pętli



- Warunek P jest warunkiem wstępny dla instrukcji **while** W do S **end while**, natomiast warunek $P \wedge \neg W$ jest jej warunkiem końcowym.
- Warunek $P \wedge W$ jest warunkiem wstępny dla instrukcji S , natomiast P jest jej warunkiem końcowym.

Projektowanie algorytmów

Poprawność algorytmu: instrukcja pętli

Warunek P , ze specyfikacji pętli, nazywany jest niezmiennikiem pętli **while-do**.

Aby udowodnić, że ma on tę własność, należy pokazać, że:

- ① Jest on spełniony przy pierwszym wejściu do pętli.
- ② Jeśli był spełniony warunek $P \wedge W$, to po wykonaniu instrukcji S nadal będzie spełniony warunek P .

Projektowanie algorytmów

Poprawność algorytmu

- Programy, które nie zawierają pętli są trywialne do analizy poprawności.
- Często bardzo trudno pokazać, że pętla w ogóle kończy pracę.

Hipoteza: poniższa pętla kończy pracę dla dowolnej początkowej wartości całkowitej $n \geq 1$.

```
1: while n > 1 do
2:   if n mod 2 = 0 then
3:     n ← n div 2
4:   else
5:     n ← 3 * n + 1
6:   end if
7: end while
```

Projektowanie algorytmów

Projektowanie zstępujące

Definition (Projektowanie zstępujące – ang. top-down design)

Projektowanie algorytmu rozwiązującego problem przez jego rozkład na ścisłe określone podproblemy i przez sprawdzanie, że jeśli każdy z podproblemów jest rozwiązany poprawnie, a ich rozwiązania są zestawione w określony sposób, to problem początkowy jest rozwiązany poprawnie. Ten proces uściślania powtarza się aż do osiągnięcia właściwego stopnia szczegółowości.

Projektowanie algorytmów

Projektowanie zstępujące: największy wspólny dzielnik

Przykład zaczerpnięty z książki Alagića i Arbiba.

Problem (1)

Dane są liczby całkowite a i b . Znajdź ich największy wspólny dzielnik.

Podstawowe własności największego wspólnego dzielnika dwóch liczb:

$$nwd(0, 0) = 0$$

$$nwd(u, v) = nwd(v, u)$$

$$nwd(u, v) = nwd(-u, v)$$

$$nwd(u, 0) = |u|$$

Projektowanie algorytmów

Projektowanie zstępujące: największy wspólny dzielnik

Opierając się na własnościach największego wspólnego dzielnika możemy problem (1) przeformułować do następującego:

Problem (2)

Dane są liczby całkowite nieujemne a i b . Znajdź ich największy wspólny dzielnik.

Rozwiązywanie problemu (1) można znaleźć rozwiązując problem (2) dla wartości $|a|$ i $|b|$.

Projektowanie algorytmów

Projektowanie zstępujące: największy wspólny dzielnik – pierwsze przybliżenie

Require: dane są dwie liczby całkowite nieujemne a i b

Ensure: wartość $nwd(a, b)$ w zmiennej x

1: $x \leftarrow a;$

2: $y \leftarrow b;$

3: **while** $y > 0$ **do** \triangleright niezmiennik: $nwd(x, y) = nwd(a, b) \wedge y \geq 0$

4: zmniejsz y i zmień x w ten sposób, aby obie te liczby pozostały
nieujemne i żeby wartość $nwd(x, y)$ nie zmieniła się;

5: **end while** $\triangleright x = nwd(a, b)$

Projektowanie algorytmów

Projektowanie zstępujące: największy wspólny dzielnik – drugie przybliżenie

Oznaczmy symbolami $x \text{ div } y$ i $x \text{ mod } y$ odpowiednio iloraz i resztę z dzielenia x przez y .

Dla dowolnych x i y zachodzi następująca relacja:

$$x = (x \text{ div } y) * y + x \text{ mod } y \quad (1)$$

Zatem:

$$x \text{ mod } y = x - (x \text{ div } y) * y$$

Dowolny wspólny dzielnik x i y (np. $x = c_1 * k, y = c_2 * k$), jest również wspólnym dzielnikiem y i $x - (x \text{ div } y) * y$, bo:

$$x - (x \text{ div } y) * y = c_1 \cdot k - (x \text{ div } y) * c_2 * k = k * (c_1 - (x \text{ div } y) * c_2).$$

Projektowanie algorytmów

Projektowanie zstępujące: największy wspólny dzielnik – drugie przybliżenie

Dowolny wspólny dzielnik y i $x - (x \text{ div } y) * y$ (np.

$y = c_1 * k, x - (x \text{ div } y) * y = c_2 * k$), jest również wspólnym dzielnikiem x i y , bo:

$$x = c_2 * k + (x \text{ div } y) * c_1 * k = k * (c_2 + (x \text{ div } y) * c_1).$$

Zatem największy wspólny dzielnik liczb x i y jest równy największemu wspólnemu dzielnikowi liczb y i $x - (x \text{ div } y) * y$:

$$\text{nwd}(x, y) = \text{nwd}(y, x \text{ mod } y).$$

Wartość $x \text{ mod } y$ jest mniejsza od y , więc za nową wartość y można przyjąć $x \text{ mod } y$ (zmniejszenie y) a za nową wartość x przyjąć starą wartość y (zmiana x tak by zachować wartość największego wspólnego dzielnika $\text{nww}(x, y)$).

Projektowanie algorytmów

Projektowanie zstępujące: największy wspólny dzielnik – drugie przybliżenie

- Niezmiennikiem pętli **while-do** jest warunek $nwd(x, y) = nwd(a, b) \wedge y \geq 0$.
- Wartość y jest nieujemna i za każdym razem maleje, więc kiedyś osiągnie 0 i pętla przestanie się wykonywać.
- Po jej zakończeniu zachodzić będzie warunek $y = 0 \wedge x = nwd(x, 0) = nwd(a, b)$.

Projektowanie algorytmów

Projektowanie zstępujące: największy wspólny dzielnik – drugie przybliżenie

Require: dane są dwie liczby całkowite nieujemne a i b

Ensure: wartość $nwd(a, b)$ w zmiennej x

- 1: $x \leftarrow a;$
- 2: $y \leftarrow b;$
- 3: **while** $y > 0$ **do** ▷ niezmiennik: $nwd(x, y) = nwd(a, b) \wedge y \geq 0$
- 4: pod r podstaw $x \bmod y$;
- 5: $x \leftarrow y$;
- 6: $y \leftarrow r$
- 7: **end while** ▷ $x = nwd(a, b)$

W powyższym programie uszczegółowienia wymaga wyliczenie reszty z dzielenia x przez y i podstawienie jej pod zmienną r .

Projektowanie algorytmów

Projektowanie zstępujące: największy wspólny dzielnik – trzecie przybliżenie

Require: dane są dwie liczby całkowite nieujemne a i b

Ensure: wartość $nwd(a, b)$ w zmiennej x

- 1: $x \leftarrow a;$
- 2: $y \leftarrow b;$
- 3: **while** $y > 0$ **do** ▷ niezmiennik: $nwd(x, y) = nwd(a, b) \wedge y \geq 0$
- 4: $r \leftarrow x;$
- 5: **while** $r \geq y$ **do** ▷ niezmiennik: $r = x \bmod y + (r \bmod y) \cdot y$
- 6: $r \leftarrow r - y$
- 7: **end while** ▷ $r = x \bmod y$
- 8: $x \leftarrow y;$
- 9: $y \leftarrow r$
- 10: **end while** ▷ $x = nwd(a, b)$

Projektowanie algorytmów

Projektowanie zstępujące: największy wspólny dzielnik – trzecie przybliżenie

W trzecim przybliżeniu, niezmiennikiem wewnętrznej pętli **while-do** jest warunek $r = x \text{ mod } y + (r \text{ div } y) \cdot y$, co wykażemy indukcyjnie.

- Za pierwszym wejściem do pętli r ma wartość równą x , zatem zgodnie z (1) warunek niezmiennika jest spełniony.
- Założmy, że przy kolejnym przebiegu pętli spełniony jest niezmiennik. Pokażemy, że po wykonaniu instrukcji $r \leftarrow r - y$, niezmiennik nadal będzie spełniony. Niech r' będzie nową wartością zmiennej r . Wówczas $r = r' + y$ i otrzymujemy:

$$r = x \text{ mod } y + (r \text{ div } y) \cdot y, \quad (\text{z zał. ind.})$$

$$r' + y = x \text{ mod } y + ((r' + y) \text{ div } y) \cdot y,$$

$$r' = x \text{ mod } y + ((r' + y) \text{ div } y) \cdot y - y,$$

$$r' = x \text{ mod } y + (r' \text{ div } y + 1) \cdot y - y,$$

$$r' = x \text{ mod } y + (r' \text{ div } y) \cdot y + y - y,$$

$$r' = x \text{ mod } y + (r' \text{ div } y) \cdot y.$$

Projektowanie algorytmów

Projektowanie zstępujące: największy wspólny dzielnik – wersja ostateczna

Require: dane są dwie liczby całkowite a i b

Ensure: wartość $nwd(a, b)$ w zmiennej x

```
1:  $x \leftarrow |a|;$ 
2:  $y \leftarrow |b|;$ 
3: while  $y > 0$  do            $\triangleright$  niezmiennik:  $nwd(x, y) = nwd(a, b) \wedge y \geq 0$ 
4:    $r \leftarrow x;$ 
5:   while  $r \geq y$  do        $\triangleright$  niezmiennik:  $r = x \bmod y + (r \bmod y) \cdot y$ 
6:      $r \leftarrow r - y$ 
7:   end while                    $\triangleright r = x \bmod y$ 
8:    $x \leftarrow y;$ 
9:    $y \leftarrow r$ 
10: end while                   $\triangleright x = nwd(a, b)$ 
```

Wykład 3

Elementy języka C

A C program is like a fast dance on a newly waxed dance floor by people carrying razors.

Waldi Ravens

Elementy języka C

- deklaracje zmiennych
- instrukcja podstawienia
- instrukcje wejścia/wyjścia
- instrukcje warunkowe
- instrukcje pętli
- deklaracja, definicja i wywołanie funkcji

Elementy języka C

Deklaracje zmiennych

- Każda użyta w programie zmienna musi być zadeklarowana.
- Deklaracja w języku C ma następującą postać:
Typ NazwaZmiennej ;
- W deklaracji zmiennej możliwe jest zainicjowanie jej wartości:
Typ NazwaZmiennej = WartośćPoczątkowa ;

Elementy języka C

Deklaracje zmiennych: zasięg deklaracji

```
01 #include <stdio.h>
02 int i =1;
03 int main()
04 {
05     printf("%d\n", i);
06     int i = 2;
07     printf("%d\n", i);
08     {
09         int i = 3;
10         printf("%d\n", i);
11     }
12     printf("%d\n", i);
13     return 0;
14 }
15
16
```

Elementy języka C

Instrukcja podstawienia

Instrukcja podstawienia ma w C następującą postać:

NazwaZmiennej = Wyrażenie;

Elementy języka C

Instrukcje wejścia/wyjścia

- Dołączenie do źródeł pliku nagłówkowego zawierającego deklaracje funkcji dostępnych w standardowej bibliotece wejścia/wyjścia:

```
#include <stdio.h>
```

- Wywołanie funkcji scanf() ma następującą postać:

```
scanf(Format, &NazwaZmiennej);
```

gdzie Format jest łańcuchem znaków ujętym między dwoma cudzysłowami.

Elementy języka C

Instrukcje wejścia/wyjścia

Tabela: Wybrane formaty do czytania danych

Format	Opis
%d	Przeczytanie liczby całkowitej.
%f	Przeczytanie liczby rzeczywistej pojedynczej precyzji.
%lf	Przeczytanie liczby rzeczywistej podwójnej precyzji.
%s	Przeczytanie łańcucha znaków do białego znaku (spacja, tabulacja albo nowa linia).

Elementy języka C

Instrukcje wejścia/wyjścia

Example

Rozpatrzmy następujący fragment programu w C:

```
int wiek;  
char imie[10];  
scanf("%d", &wiek);  
scanf("%s", imie);
```

- Przed zmienną wiek należy podać ampersand.
- Nie trzeba podawać ampersanda przed nazwą tablicy znaków imie.
- Na końcu przeczytanego napisu zostanie dodany znak '\0' o kodzie 0.

Elementy języka C

Instrukcje wejścia/wyjścia

Do drukowania wartości wyrażeń służy funkcja printf():

```
printf(Format, Wyrażenie1, Wyrażenie2, ..., Wyrażenien);
```

gdzie Format jest łańcuchem znaków ujętym w cudzysłowy a Wyrażenie_i, dla $i = 1, 2, \dots, n$, jest wyrażeniem, którego wartość będzie drukowana.

Elementy języka C

Instrukcje wejścia/wyjścia

Tabela: Wybrane formaty do drukowania danych

Format	Opis
%c	Drukowanie jednego znaku.
%d	Drukowanie liczby całkowitej.
%f	Drukowanie liczby rzeczywistej.
%s	Drukowanie łańcucha znaków zakońzonego znakiem o kodzie 0.

Format drukowania może zawierać dowolny tekst oraz następujące znaki specjalne:

- \n przejście do nowego wiersza,
- \t przejście do kolejnej pozycji tabulacji,
- \a krótki sygnał dźwiękowy (ang. bell character).

Elementy języka C

Instrukcje wejścia/wyjścia

Przy każdym formacie można podać informację ile znaków ma zajmować drukowana wartość.

Example

W wyniku wykonania następującego fragmentu kodu:

```
double d = 3.14159267;
```

```
printf("                  Liczba d=%f\n", d);
printf(" Ta sama liczba na polu dziesięciu znaków d=%10f\n", d);
printf("Ta sama liczba z trzema cyframi po przecinku d=%.3f\n", d);
```

zostaną wydrukowane następujące wiersze:

```
                  Liczba d=3.141593
Ta sama liczba na polu dziesięciu znaków d= 3.141593
Ta sama liczba z trzema cyframi po przecinku d=3.142
```

Elementy języka C

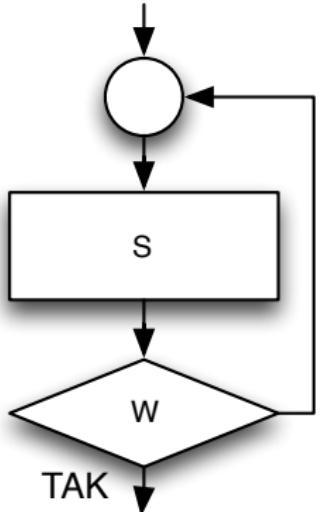
Instrukcja pętli: pętla while-do

- Warunek W sterujący wykonywaniem się pętli może być w języku C dowolnym wyrażeniem dostarczającym wartość będącą liczbą całkowitą.
- Jeśli wartość W jest równa zero, to warunek nie jest spełniony, natomiast każda wartość różna od zera oznacza spełnienie warunku.

Schemat blokowy	Pseudokod	Język C
<pre> graph TD S1[] --> W{W} W -- TAK --> S1 W -- NIE --> End(()) </pre>	<pre> 1: while W do 2: S 3: end while </pre>	<pre> while(W) S; </pre>

Elementy języka C

Instrukcja pętli: pętla do-while

Schemat blokowy	Pseudokod	Język C
 <p>NIE</p> <p>TAK</p>	<ol style="list-style-type: none">1: repeat2: S3: until <i>W</i>	do <i>S</i> ; while (<i>!W</i>);

Elementy języka C

Instrukcja pętli: pętla for

Schemat blokowy	Pseudokod	Język C
<pre> graph TD Init[i ← i₀] --> Cond{i ≤ i₁} Cond -- TAK --> Body[S] Body --> Inc[i ← i + 1] Inc --> Cond Cond -- NIE --> End </pre>	<pre> 1: for i ← i₀, i₁ do 2: S 3: end for </pre>	<pre> for(i=i0; i<=i1; i++) S; </pre>

Elementy języka C

Instrukcja pętli: pętla for

Pętla **for** w języku C jest jedynie skróconym zapisem pętli **while-do**.

Example

Rozpatrzmy następującą pętlę **for**:

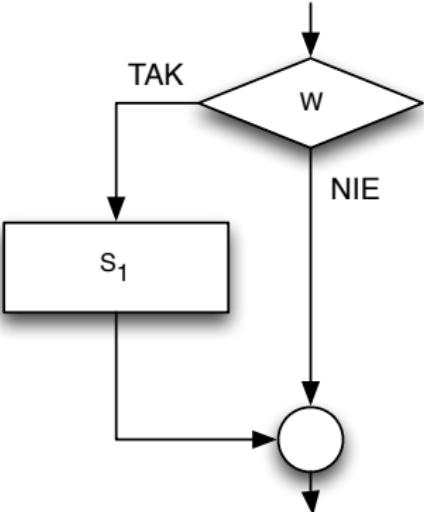
```
for(S1; W; S2)  
    S;
```

Można zapisać ją jako następująca pętla **while-do**:

```
S1;  
while(W)  
{  
    S;  
    S2;  
}
```

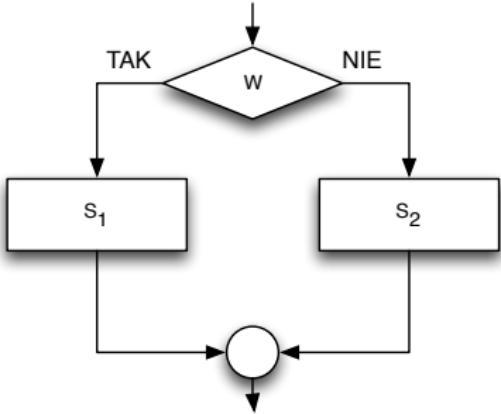
Elementy języka C

Instrukcje warunkowe: instrukcja if-then

Schemat blokowy	Pseudokod	Język C
 <pre>graph TD; Start(()) --> W{W}; W -- TAK --> S1[S1]; S1 --> W; W -- NIE --> End(());</pre>	<ol style="list-style-type: none">1: if <i>W</i> then2: <i>S₁</i>3: end if	if(<i>W</i>) <i>S₁</i> ;

Elementy języka C

Instrukcje warunkowe: instrukcja if-then-else

Schemat blokowy	Pseudokod	Język C
 <pre>graph TD; W{W} -- TAK --> S1[S1]; W -- NIE --> S2[S2]; S1 --> End(()); S2 --> End;</pre>	<ol style="list-style-type: none">1: if <i>W</i> then2: <i>S</i>₁3: else4: <i>S</i>₂5: end if	<pre>if(W) S1; else S2;</pre>

Elementy języka C

Instrukcje warunkowe: instrukcja switch

W języku C dostępna jest instrukcja **switch**, która przenosi sterowanie w różne miejsca w zależności od wartości danego wyrażenia *W*:

```
switch(W)
{
    case W1: S1;
                break;
    case W2: S2;
                break;
    ...
    case Wn: Sn;
                break;
    default: S;
}
```

Elementy języka C

Instrukcje warunkowe: instrukcja switch

Fraza case Wi: jest etykietą miejsca gdzie nastąpi skok gdy wyrażenie W będzie miało wartość W_i .

```
#include <stdio.h>
int main()
{
    int i, n;
    scanf("%d", &n);
    switch(n)
    {
        case 0: for(i=0; i<10; i++)
        {
            case 1: printf("i=%d\n", i); // co gdy n = 1?
        }
    }
    return 0;
}
```

Elementy języka C

Deklaracje funkcji

- Niech T będzie typem wartości n -argumentowej funkcji f a T_1, T_2, \dots, T_n będą typami kolejnych parametrów formalnych x_1, x_2, \dots, x_n .
- W następujący sposób deklaruje się funkcję f :

$$T\ f(T_1\ x_1, T_2\ x_2, \dots, T_n\ x_n);$$

Elementy języka C

Deklaracje funkcji

Example

Niech w pliku dekl.c znajduje się następująca deklaracja i wywołanie funkcji suma():

```
int suma(int x, int y);
```

```
int main()
{
    int n;
    n = suma(2, 2);
    return 0;
}
```

Elementy języka C

Deklaracje funkcji

Example (cd.)

Próba skompilowania źródeł programu dekl.c kończy się niepowodzeniem:

```
$ gcc -o dekl dekl.c
Undefined symbols for architecture x86_64:
  "_suma", referenced from:
    _main in ccRA9xzb.o
ld: symbol(s) not found for architecture x86_64
collect2: ld returned 1 exit status
```

Elementy języka C

Definicja funkcji

- Założmy, że po wykonaniu instrukcji S , wartość wyrażenia W jest wartością n -argumentowej funkcji o typie wyniku T i parametrach x_1, x_2, \dots, x_n typów T_1, T_2, \dots, T_n .
- Definicję funkcji f zapisuje się w C w sposób następujący:

```

$$\begin{array}{c} T\ f(T_1\ x_1, T_2\ x_2, \dots, T_n\ x_n) \\ \{ \\ \quad S; \\ \quad \mathbf{return}\ W; \\ \} \end{array}$$

```

- W szczególnym przypadku, jeśli samo wyrażenie W definiuje wartość funkcji f , instrukcje S można pominąć:

```

$$\begin{array}{c} T\ f(T_1\ x_1, T_2\ x_2, \dots, T_n\ x_n) \\ \{ \\ \quad \mathbf{return}\ W; \\ \} \end{array}$$

```

Elementy języka C

Definicja funkcji

Example

Niech w pliku def.c znajduje się następująca definicja i wywołanie funkcji suma():

```
int suma(int x, int y)
{
    return x+y;
}
```

```
int main()
{
    int n;
    n = suma(2, 2);
    return 0;
}
```

Elementy języka C

Definicja funkcji

Example

Czasami nie da się zdefiniować funkcji bez ich deklaracji:

```
double f()
{
    ... g() ... // nieznana funkcja g()
}
```

```
float g()
{
    ... f() ...
}
```

Elementy języka C

Definicja funkcji

Example (cd.)

Należy zadeklarować funkcję g() przed jej pierwszym użyciem:

```
float g();  
  
double f()  
{  
    ... g() ...  
}  
  
float g()  
{  
    ... f() ...  
}
```

Elementy języka C

Definicja funkcji

- W języku C nie ma procedur takich jak w pseudokodzie.
- Aby zapisać w języku C procedurę p o parametrach x_1, x_2, \dots, x_n typów T_1, T_2, \dots, T_n , definiujemy funkcję o pustym typie wartości void:

```
void  $p(T_1 x_1, T_2 x_2, \dots, T_n x_n)$ 
{
    S;
}
```

Elementy języka C

Wywołanie funkcji

Example

Rozpatrzmy następujący fragment kodu dla funkcji `suma()` zwracającej sumę dwóch swoich argumentów będących liczbami całkowitymi:

```
int n;  
n = suma(suma(1, 2), suma(4, 5));
```

W wyniku jego wykonania pod zmienną `n` zostanie podstawiona wartość 12:

- ➊ Zostanie policzona wartość `suma(1, 2)` równa 3.
- ➋ Zostanie policzona wartość `suma(4, 5)` równa 9.
- ➌ Zostanie policzona wartość `suma(3, 9)` równa 12.
- ➍ Wyliczona wartość 12 zostanie podstawiona pod `n`.

Elementy języka C

Przekazywanie parametrów

- W C jest tylko jeden sposób przekazywania parametrów: **przez wartość** (odpowiednik parametru **in** z pseudokodu).
- Jeśli w parametrze wywołania y ma się znaleźć odebrana z funkcji wartość, to należy zamiast wartości zmiennej y przekazać jej adres &y.
- Deklaracja parametru T *y oznacza, że parametr y jest wskazaniem na wartość typu T (adresem miejsca, gdzie przechowywana jest wartość typu T), natomiast *y jest wskazywaną przez y wartością typu T.

Elementy języka C

Przekazywanie parametrów

Example

Rozpatrzmy następującą funkcję (procedurę) podwajacz:

```
void podwajacz(int x, int y)
{
    y = x+x;
}
```

Jeśli zmienna a ma wartość 2 i zmienna b ma wartość 3, to po takim wywołaniu funkcji podwajacz:

```
podwajacz(a, b);
```

wartość zmiennej b nie ulegnie zmianie (nadal będzie równa 3).

Elementy języka C

Przekazywanie parametrów

Example (cd.)

Aby wyliczona wartość została przekazana na zewnątrz, funkcja podwajacz musi dostać nie wartość drugiego parametru ale miejsce gdzie wyliczoną wartość należy odłożyć:

```
void podwajacz(int x, int *y)
{
    *y = x+x;
}
```

Jeśli teraz wywołamy funkcję podwajacz w następujący sposób:

```
podwajacz(a, &b);
```

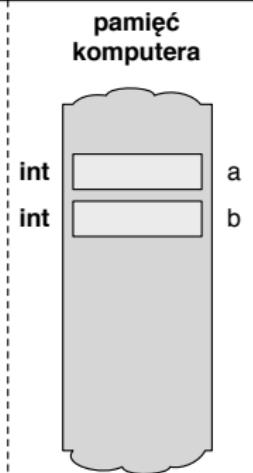
to w zmiennej b znajdzie się podwojona wartość zmiennej a.

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}
```

```
int a, b;
a = 2;
b = 1;
p(a, &b);
```



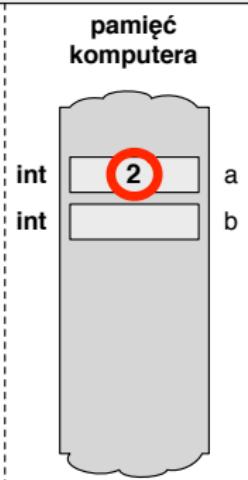
1. w pamięci znajdują się zmienne a i b typu int

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```



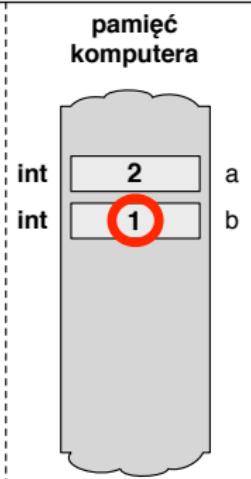
1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```



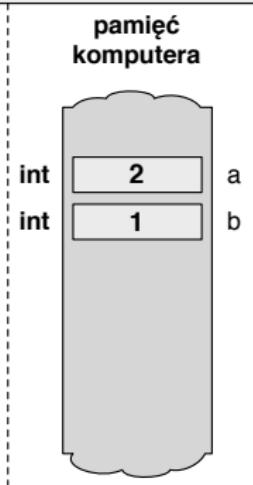
1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```



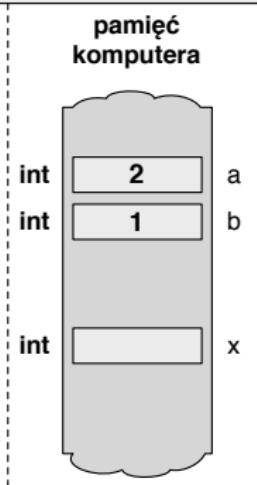
1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```



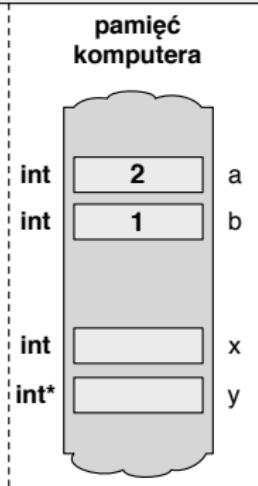
1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```



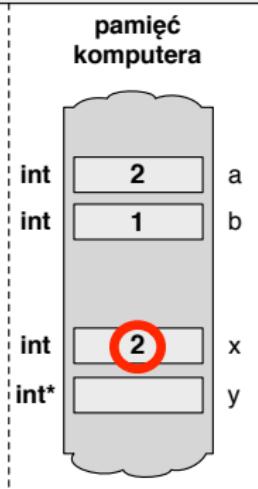
1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```



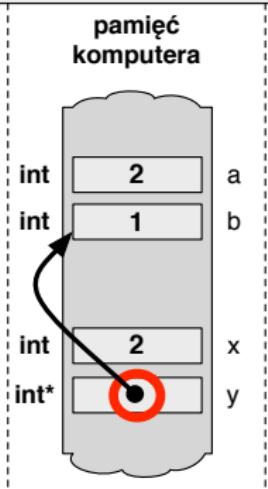
1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```



1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b

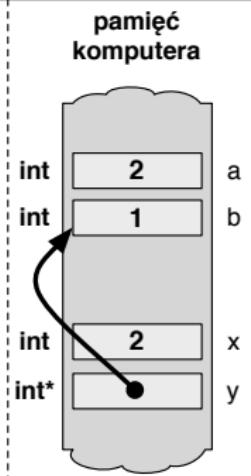
Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
```

```
    x = x+1;
    *y = *y+1;
}
```

```
int a, b;
a = 2;
b = 1;
p(a, &b);
```



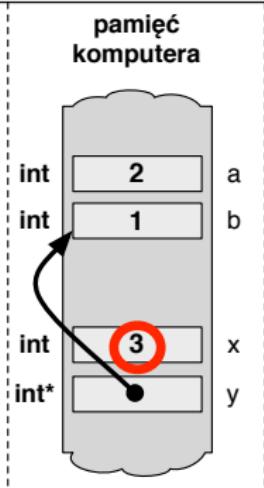
1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}
```

```
int a, b;
a = 2;
b = 1;
p(a, &b);
```



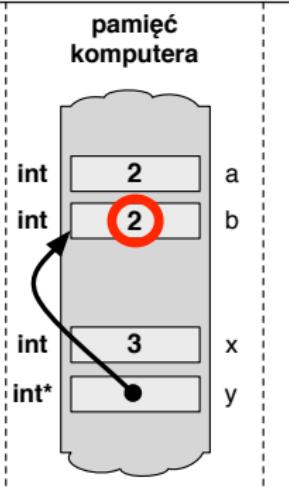
1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```



1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1

Elementy języka C

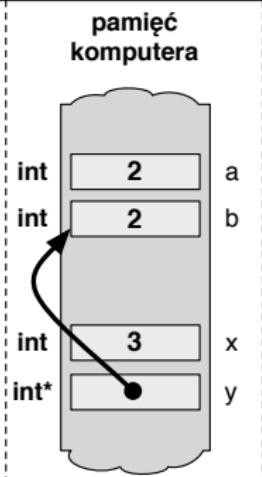
Przekazywanie parametrów

```
void p(int x, int *y)
{

```

```
    x = x+1;
    *y = *y+1;
}
```

```
int a, b;
a = 2;
b = 1;
p(a, &b);
```



- w pamięci znajdują się zmienne a i b typu int
- pod a podstawiana jest wartość 2
- pod b podstawiana jest wartość 1
- wywonywana jest funkcja p
- tworzona jest zmienna lokalna x typu int
- tworzona jest zmienna lokalna y typu int*
- pod x podstawiana jest wartość zmiennej a
- pod y podstawiany jest adres zmiennej b
- rozpoczyna się wykonywanie funkcji p
- zwiększa się wartość x o 1
- zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1
- kończy się wykonanie funkcji p

Elementy języka C

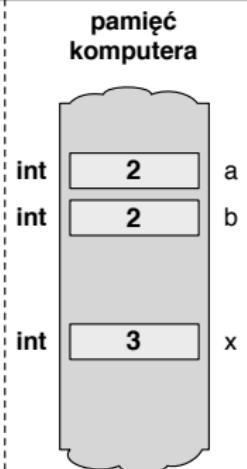
Przekazywanie parametrów

```
void p(int x, int *y)
{

```

```
    x = x+1;
    *y = *y+1;
}
```

```
int a, b;
a = 2;
b = 1;
p(a, &b);
```



1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y

Elementy języka C

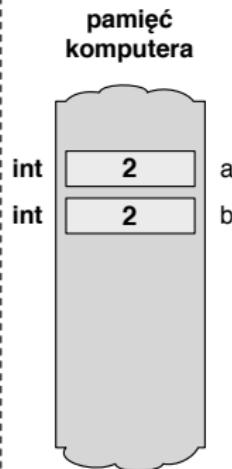
Przekazywanie parametrów

```
void p(int x, int *y)
{

```

```
    x = x+1;
    *y = *y+1;
}
```

```
int a, b;
a = 2;
b = 1;
p(a, &b);
```



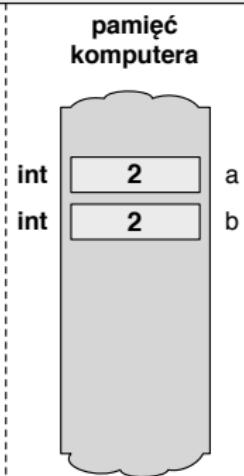
1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y
14. usuwana jest zmienna lokalna x

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```



1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y
14. usuwana jest zmienna lokalna x
15. sterowanie powraca do miejsca za wywołaniem funkcji p

Elementy języka C

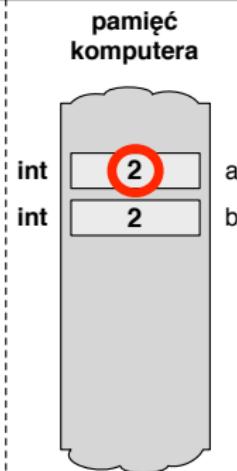
Przekazywanie parametrów

```
void p(int x, int *y)
{

```

```
    x = x+1;
    *y = *y+1;
}
```

```
int a, b;
a = 2
b = 1;
p(a, &b);
```



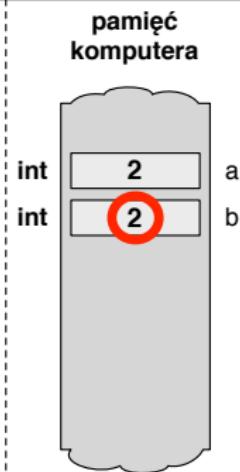
1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y
14. usuwana jest zmienna lokalna x
15. sterowanie powraca do miejsca za wywołaniem funkcji p
16. zmienna a nie zmieniła wartości

Elementy języka C

Przekazywanie parametrów

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1
p(a, &b);
```



1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętanym w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y
14. usuwana jest zmienna lokalna x
15. sterowanie powraca do miejsca za wywołaniem funkcji p
16. zmieniona a nie zmieniła wartości
17. zmieniona b zmieniła wartość z 1 na 2

Wykład 4

Podstawowe typy danych w C

Podstawowe typy danych w C

- typ całkowity
- typ rzeczywisty
- typ wskaźnikowy
- typ tablicowy
- struktury i unie

Podstawowe typy danych w C

Typ całkowity

Standard języka C określa tylko relacje między rozmiarami danych w typach całkowitych:

$$\text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short int}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long int})$$

Na komputerze 64-bitowym rozmiary te są następujące 1, 2, 4 i 8 bajtów.
Każdy z powyższych typów występuje w dwóch odmianach:

- ze znakiem do wyrażania wartości dodatnich i ujemnych, wtedy można poprzedzić nazwę typu słowem kluczowym **signed**,
- bez znaku do wyrażania wartości nieujemnych, wtedy trzeba poprzedzić nazwę typu słowem kluczowym **unsigned**.

Podstawowe typy danych w C

Typ całkowity

Jeśli n jest liczbą bitów w binarnej reprezentacji wartości całkowitej, to

- wartości ze znakiem są z zakresu od -2^{n-1} do $2^{n-1} - 1$
- wartości bez znaku są z zakresu od 0 do $2^n - 1$

Tabela: Zakresy typów dla liczb całkowitych na komputerze 64-bitowym

Typ	min	max
(signed) char	-128	127
unsigned char	0	255
(signed) short int	-32 768	32 767
unsigned short int	0	65 535
(signed) int	-2 147 483 648	2 147 483 647
unsigned int	0	4 294 967 295
(signed) long int	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long int	0	18 446 744 073 709 551 615

Podstawowe typy danych w C

Typ całkowity

Tabela: Wybrane operacje na liczbach całkowitych

Operator	Działanie	Przykład
+	dodawanie	2 + 2 == 4
-	odejmowanie	2 - 4 == -2
*	możenie	2 * 4 == 8
/	dzielenie całkowitoliczbowe	4 / 3 == 1
%	reszta z dzielenia	5 % 2 == 1
++	pre-inkrementacja	++n == n+1
++	post-inkrementacja	n++ == n
--	pre-dekrementacja	--n == n-1
--	post-dekrementacja	n-- == n

Podstawowe typy danych w C

Typ całkowity

Tabela: Wybrane operacje na liczbach całkowitych (cd.)

Operator	Działanie	Przykład
<code>~</code>	negacja bitowa	<code>~2 == -3</code>
<code>&</code>	koniunkcja bitowa	<code>5 & 2 == 0</code>
<code> </code>	alternatywa bitowa	<code>5 3 == 7</code>
<code>^</code>	alternatywa rozłączna	<code>5 ^ 3 == 6</code>
<code><<</code>	przesunięcie bitów w lewo	<code>3 << 2 == 12</code>
<code>>></code>	przesunięcie bitów w prawo	<code>5 >> 2 == 1</code>
<code>==</code>	równe (1–spełnienie, 0–niespełnienie)	<code>1 == 2 == 0</code>
<code>!=</code>	nie równe	<code>1 != 2 == 1</code>
<code><</code>	mniejsze	<code>1 < 2 == 1</code>
<code><=</code>	mniejsze lub równe	<code>1 <= 2 == 1</code>
<code>></code>	większe	<code>1 > 2 == 0</code>
<code>>=</code>	większe lub równe	<code>1 >= 2 == 0</code>

Podstawowe typy danych w C

Typ całkowity

Tabela: Wybrane operacje na liczbach całkowitych (cd.)

Operator	Działanie	Przykład
!	logiczna negacja (0–fałsz, w pp.–prawda)	<code>! 2 == 0</code>
&&	logiczna koniunkcja	<code>5 && 2 == 1</code>
 	logiczna alternatywa	<code>2 0 == 1</code>
? :	wyrażenie warunkowe	<code>n>=0?n:-n == abs(n)</code>
()	rzutowanie	<code>(int) 3.75 == 3</code>
= op=	podstawienie $x \text{ op=} w$ odpowiada $x = x \text{ op } w$, dla operacji op	<code>(x=2) + (y=3) == 5</code>

Podstawowe typy danych w C

Typ całkowity

Example (Zliczanie jedynek)

Założymy, że obliczenia odbywają się na komputerze 64-bitowym.
Policzmy liczbę jedynek w binarnej reprezentacji wartości zmiennej u:

```
unsigned long int u;
int i, n = 0;
for(i = 0; i<64; i++)
{
    if(u & 1) n++;
    u = u >> 1;
}
```

Powyższa pętla wykonuje się zawsze 64 razy bez względu na to ile jest bitów równych 1 w wartości zmiennej u.

Podstawowe typy danych w C

Typ całkowity

Example (Zliczanie jedynek cd.)

Zmodyfikujemy pętlę zliczającą jedynki:

```
n = 0;  
while(u > 0)  
{  
    n++;  
    u = u & (u-1);  
}
```

Powyższa pętla wykonuje się tyle razy ile jest jedynek w binarnej reprezentacji u.

Ćwiczenie: dlaczego $u \& (u-1)$ ma o jedną jedynkę mniej niż u?

Podstawowe typy danych w C

Typ rzeczywisty

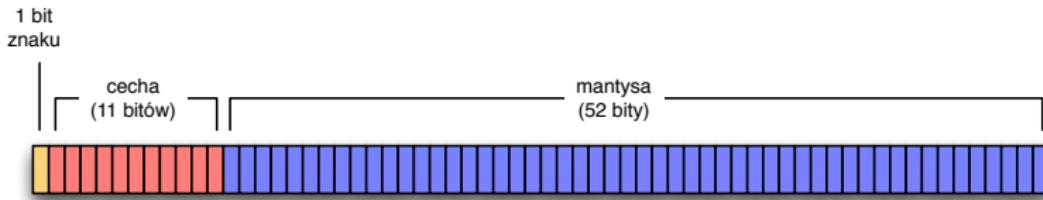
- Liczby zmiennoprzecinkowe reprezentowane są jako liczby pojedynczej (typ **float**) lub podwójnej (typ **double**) precyzji zgodnie ze standardem **IEEE 754**.
- Liczba zmiennoprzecinkowa pojedynczej precyzji zajmuje 32 bity.
- Liczba zmiennoprzecinkowa podwójnej precyzji zajmuje 64 bity.

typ	min neg	max neg	min pos	max pos	prec
float	-3.4×10^{38}	-1.2×10^{-38}	1.2×10^{-38}	3.4×10^{38}	6
double	-1.8×10^{308}	-2.2×10^{-308}	2.2×10^{-308}	1.8×10^{308}	15

Podstawowe typy danych w C

Typ rzeczywisty: **double**

- 64 bity reprezentujące liczbę podwójnej precyzji podzielone są na:
 - bit znaku,
 - bity cechy (11 bitów),
 - bity mantysy (52 bity).



Podstawowe typy danych w C

Typ rzeczywisty: **double**

- Niech z będzie bitem znaku, $c_{10}c_9 \dots c_0$ będą bitami cechy², natomiast $m_{51}m_{50} \dots m_0$ bitami mantysy.
- Reprezentowana liczba ma wartość:

$$(-1)^z \times (1.m_{51}m_{50} \dots m_0)_2 \times 2^{(c_{10}c_9 \dots c_0)_2 - 1023}$$

- Jest skończenie wiele liczb zmiennoprzecinkowych (co najwyżej 2^{64}).
- Liczby zmiennoprzecinkowe rozmieszczone są symetrycznie względem zera.
- Jest najmniejsza dodatnia liczba zmiennopozycyjna ($= 2^{-1022} \approx 2.2 \times 10^{-308}$).
- Jest największa dodatnia liczba zmiennopozycyjna ($= (2 - 2^{-52}) \times 2^{1023} \approx 2^{1024} \approx 1.8 \times 10^{308}$).

²Cechy 000000000000 i 111111111111 mają specjalne znaczenie.

Podstawowe typy danych w C

Typ rzeczywisty: **double**

- Epsilon maszynowym nazywa się najmniejszą liczbę zmiennoprzecinkową, która dodana do 1 jest większa od 1 ($= 2^{-52} \approx 2.22 \times 10^{-16}$).
- W przedziale $[2^n, 2^{n+1}]$ liczby zmiennoprzecinkowe odległe są od siebie o 2^{n-52} .

Example $(2^{53} + 1 = 2^{53})$

W zakresie $[2^{53}, 2^{54}] = [9007199254740992, 18014398509481984]$ odległość między kolejnymi liczbami zmiennoprzecinkowymi wynosi $2^{53-52} = 2$.

Podstawowe typy danych w C

Typ wskaźnikowy

- Często konieczne jest operowanie nie tylko na danych ale również na adresach pamięci, gdzie dane te są przechowywane.
- Niech Typ będzie dowolnym typem. Wówczas deklaracja zmiennej:

Typ *NazwaZmiennej ;

określa zmienną która przechowuje adres miejsca pamięci, w której przechowywana jest wartość zadanego typu.

Podstawowe typy danych w C

Typ wskaźnikowy

Example

Rozpatrzmy deklarację:

```
int *p;
```

Wówczas zmienna p jest typu *wskazanie na wartość całkowitą*. Stosując operator wyłuskania * do zmiennej typu wskaźnikowego otrzymuje się wskazywaną wartość. Zatem *p jest wartością typu całkowitego, której adres przechowywany jest w zmiennej p.

Podstawowe typy danych w C

Typ wskaźnikowy

Aby zarezerwować miejsce w pamięci o zadanym rozmiarze należy skorzystać z funkcji:

```
void *malloc(int RozmiarObszaru)
```

Do określenia rozmiaru pamięci potrzebnej do przechowania danej typu T można użyć operacji **sizeof(T)**.

Podstawowe typy danych w C

Typ wskaźnikowy

Example

Rozpatrzmy następujący fragment kodu w języku C na komputerze 64-bitowym:

```
double *ptr;  
ptr = malloc(8);  
*ptr = 3.14159267;
```

Zmienna `ptr` została zadeklarowana jako wskazanie na wartość rzeczywistą podwójnej precyzji. Wywołując funkcję `malloc(8)` rezerwowany zostaje obszar ośmiu bajtów a jego adres w pamięci zwracany jest jako wartość funkcji. Po podstawieniu za zmienną `ptr` adresu zarezerwowanego obszaru można wpisać do niego wartość rzeczywistą `3.14159267` wykonując instrukcję podstawienia `*ptr = 3.14159267`.

Podstawowe typy danych w C

Typ wskaźnikowy

- *Stertą* nazywa się obszar pamięci, w którym rezerwowane są za pomocą funkcji `malloc()` fragmenty pamięci.
- Jeśli na stercie nie ma wolnego obszaru wystarczającego by pomieścić rezerwowane miejsce (rozmiar podawany jako argument wywołania funkcji `malloc()`), funkcja `malloc()` zwraca specjalną wartość `NULL`.

Podstawowe typy danych w C

Typ wskaźnikowy

- Obszar zarezerwowany za pomocą funkcji `malloc()` należy niezwłocznie po tym jak przestaje być potrzebnym do dalszych obliczeń zwolnić.
- Do zwalniania zarezerwowanego obszaru pamięci służy funkcja:

```
void free(void *ptr)
```

- Wywołując funkcję `free()` należy podać w argumencie wywołania wskazanie na zwalniane miejsce.
- Aby korzystać z funkcji `malloc()` i `free()` należy dołączyć do pliku ze źródłami programu plik nagłówkowy zawierający odpowiednie deklaracje:

```
#include <stdlib.h>
```

Podstawowe typy danych w C

Typ tablicowy

- Tablica jest uporządkowaną kolekcją obiektów tego samego typu.
- Następująca instrukcja deklaruje N -elementową tablicę tab o elementach typu T:

T tab[N] ;

- Do każdego elementu tablicy można się odwołać podając jego indeks z zakresu od 0 do $N-1$.
- Rozmiar tablicy musi być stały i często definiuje się go za pomocą dyrektywy `#define` jak w poniższym przykładzie:

```
#define MAXN 100  
...  
int main()  
{  
    int t[MAXN];  
    ...  
}
```

Podstawowe typy danych w C

Typ tablicowy

- Deklarując tablicę można ją jednocześnie zainicjować wartościami umieszczonymi między nawiasami klamrowymi.
- Inicjując tablicę nie trzeba podawać liczby jej elementów gdyż kompilator sam policzy liczbę wartości podanych przy inicjowaniu:

```
int tab[] = {1, 2, 3, 5, 8};
```

Podstawowe typy danych w C

Typ tablicowy

- Pamiętać należy, że kiedy przekazuje się tablicę do funkcji jako parametr, to przekazywany jest jedynie jej adres.
- Należy do funkcji przesyłać osobnym parametrem liczbę elementów tablicy.

Example

```
double srednia(double tablica[], int n)
{
    double suma = 0.0;
    int i;
    for(i=0; i<n; i++)
        suma += tablica[i];
    return suma/n;
}
```

Podstawowe typy danych w C

Typ tablicowy

- Często parametr będący adresem tablicy deklaruje się nie w postaci typ nazwa[] ale jako typ *nazwa.
- W wywołaniu funkcji nie trzeba dla tablicy będącej parametrem podawać ampersand przed nazwą tablicy, gdyż nazwa tablicy jest jednocześnie adresem jej początku .

Podstawowe typy danych w C

Struktury

- W przeciwieństwie do tablic, struktury są krotkami elementów o różnych typach.
- Elementy struktury nazywa się polami. Niech p_1, p_2, \dots, p_n będą nazwami pól a t_1, t_2, \dots, t_n typami tych pól. Wówczas strukturę S o tych polach deklaruje się następująco:

```
struct S {t1 p1; t2 p2; ... tn pn;};
```

- Powyższa instrukcja deklaruje jedynie typ struct S.
- Zmienną x będącą strukturą S deklaruje się następująco:

```
struct S x;
```

Podstawowe typy danych w C

Struktury

- Do pola p_i struktury x typu struct S odwołuje się za pomocą operatora selekcji zapisywanego w postaci kropki:

$$x.p_i$$

- Każdy z typów t_1, t_2, \dots, t_n musi być wcześniej zdefiniowany zatem nie można zdefiniować struktury S o polu typu struktura S (kompilator nie byłby w stanie określić rozmiaru pamięci zajmowanego przez to pole).
- Można jednak deklarować pole o typie wskazanie na strukturę S :

```
struct S { ...; struct S * p_i; ...};
```

- W powyższej deklaracji pole p_i przechowuje wskazanie do struktury typu S zatem jego rozmiar w pamięci wyznacza architektura komputera (długość adresu).

Podstawowe typy danych w C

Struktury

Example

Zdefiniujemy strukturę służącą do reprezentowania liczb zespolonych i podamy przykład funkcji obliczającą sumę takich liczb:

```
struct Zespolona
{
    double re; double im;
};

struct Zespolona suma(struct Zespolona x, struct Zespolona y)
{
    struct Zespolona z;
    z.re = x.re + y.re;  z.im = x.im + y.im;
    return z;
}
```

Podstawowe typy danych w C

Struktury

Jeśli t_1, t_2, \dots, t_n są typami pól w strukturze S a T_1, T_2, \dots, T_n są zbiorami wartości typów t_1, t_2, \dots, t_n , to typ struct S ma następujący zbiór wartości będący iloczynem kartezjańskim:

$$T_1 \times T_2 \times \cdots \times T_n.$$

Podstawowe typy danych w C

Unie

- W przeciwieństwie do struktur będących krotkami elementów (mogą mieć różne typy), unie wyrażają alternatywę elementów (mogą mieć różne typy).
- Elementy unii nazywa się składowymi. Niech s_1, s_2, \dots, s_n będą nazwami składowych a t_1, t_2, \dots, t_n typami tych składowych. Wówczas unię U o tych składowych deklaruje się następująco:

union $U \{t_1\ s_1; t_2\ s_2; \dots t_n\ s_n; \};$

- Powyższa instrukcja deklaruje jedynie typ union U .
- Zmienną x będącą unią U deklaruje się następująco:
 $\text{union } U\ x;$
- Do składowej s_i unii x typu union U odwołuje się za pomocą operatora selekcji zapisywanego w postaci kropki:

$x.s_i$

Podstawowe typy danych w C

Unie

Example

Zdefiniujemy struktury Mężczyzna i Kobieta do przechowywania danych o imieniu, nazwisku i nazwisku panieńskim. Struktury te będą składowymi unii Osoba, gdyż w zależności od płci osoba ma lub nie ma nazwiska panieńskiego.

```
struct Mężczyzna {char imię[20]; char nazwisko[20];};  
struct Kobieta {char imię[20]; char nazwisko[20];  
                char nazwisko_panienskie[20];};  
  
union Osoba {struct Mężczyzna m; struct Kobieta k;};  
  
union Osoba pracownik;
```

Podstawowe typy danych w C

Unie

Example (cd.)

Możliwe pola w unii pracownik typu union Osoba:

- dla osoby będącej mężczyzną: pracownik.m.imie i pracownik.m.nazwisko,
- dla osoby będącej kobietą: pracownik.k.imie, pracownik.k.nazwisko i pracownik.k.nazwisko_panienskie.

Jeśli t_1, t_2, \dots, t_n są typami składowych w unii U a T_1, T_2, \dots, T_n są zbiorami wartości typów t_1, t_2, \dots, t_n , to typ union U ma następujący zbiór wartości będący sumą mnogościową:

$$T_1 \cup T_2 \cup \dots \cup T_n.$$

Podstawowe typy danych w C

Unie

Example

Niech t_1, t_2, t_3, t_4 będą typami o zbiorach wartości, odpowiednio, T_1, T_2, T_3, T_4 . Wówczas poniższa unia:

```
union U
{
    struct S1 {t1 p1; t2 p2;} s1;
    struct S2 {t3 p3; t4 p4;} s2;
};
```

reprezentuje typ o zbiorze wartości:

$$(T_1 \times T_2) \cup (T_3 \times T_4).$$

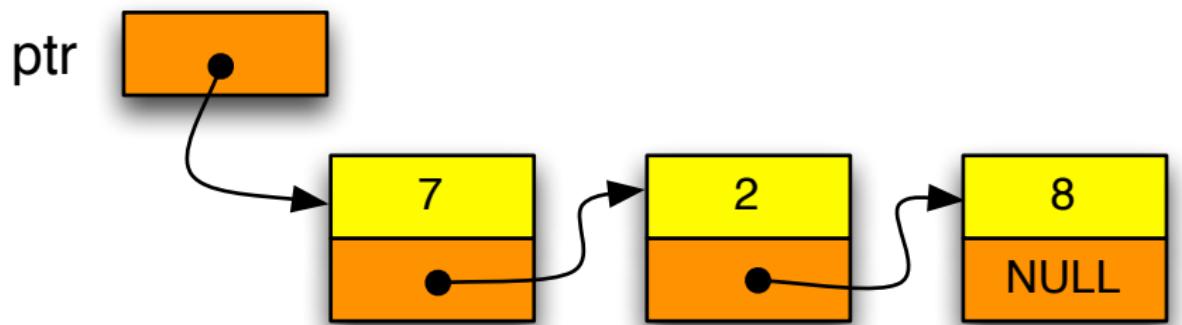
Podstawowe typy danych w C

Struktury dynamiczne

- Przedstawione wcześniej typy złożone mają ustaloną liczbę składowych (elementów) i nie mogą jej zmienić w trakcie działania programu.
- Często istnieje konieczność przetwarzania takich struktur danych, dla których przybywa i ubywa elementów.
- Służą do tego struktury dynamiczne tworzone za pomocą wskaźników.

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa



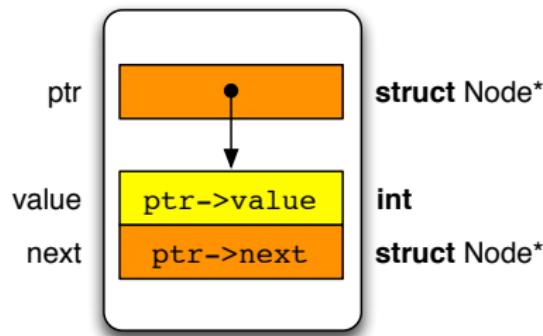
- Aby stworzyć w pamięci komputera listę jednokierunkową należy zadeklarować odpowiednią strukturę.
- Struktura będzie miała dwa pola:
 - ➊ Pole przechowujące pamiętaną w elemencie wartość (ang. value).
 - ➋ Pole przechowujące wskazanie na następny element listy (ang. next).

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa

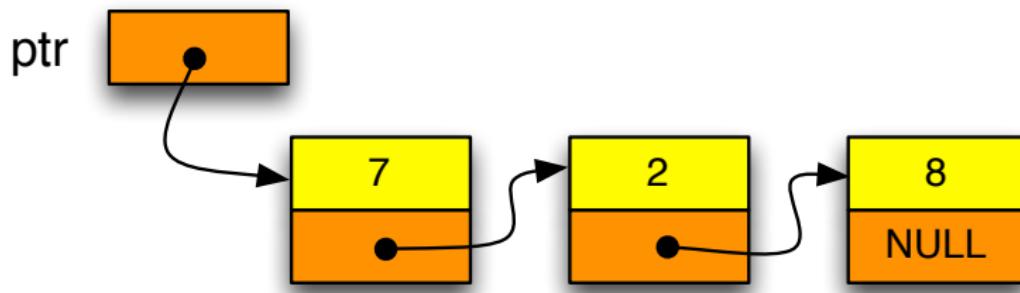
Poniżej przedstawiono deklarację struktury Node służącej przechowywaniu elementu listy:

```
struct Node
{
    int value;
    struct Node *next;
};
```



Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa



Jeśli zmienna `ptr` wskazuje na początek listy, to

- `ptr->value` ma wartość 7,
- `ptr->next` jest wskazaniem na drugi element listy,
- `ptr->next->value` ma wartość 2,
- `ptr->next->next` jest wskazaniem na trzeci element listy,
- `ptr->next->next->value` ma wartość 8,
- `ptr->next->next->next` ma wartość `NULL` (nie ma czwartego elementu listy).

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa

Często korzystać będziemy z typu `struct Node*` dlatego zdefiniujemy go jako typ `List`:

```
typedef struct Node *List;
```

- Zmienna typu `List` przechowuje wskazanie na elementy listy (dokładniej na strukturę przechowującą element listy).
- Zmienna typu `List*` przechowuje wskazanie na miejsce gdzie przechowywane jest wskazanie na elementy listy (dokładniej na strukturę przechowującą element listy).

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa – drukowanie wartości

Do drukowania elementów listy możemy posłużyć się następującą funkcją drukuj():

```
void drukuj(List ptr)
{
    while(ptr != NULL)
    {
        printf("%d ", ptr->value);
        ptr = ptr->next;
    }
    printf("\n");
}
```

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa – wstawianie na początek

Zawsze znany jest początek listy, zatem najprościej jest wstawiać kolejne elementy listy właśnie na jej początek.

Poniższa funkcja dopisz() wstawia kolejną wartość całkowitą przed pierwszy element listy i zwraca wskazanie na nowy początek listy:

```
List dopisz(List ptr, int i)
{
    List nowy;
    nowy = malloc(sizeof(struct Node));
    nowy->value = i;
    nowy->next = ptr;
    return nowy;
}
```

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa – wstawianie na początek

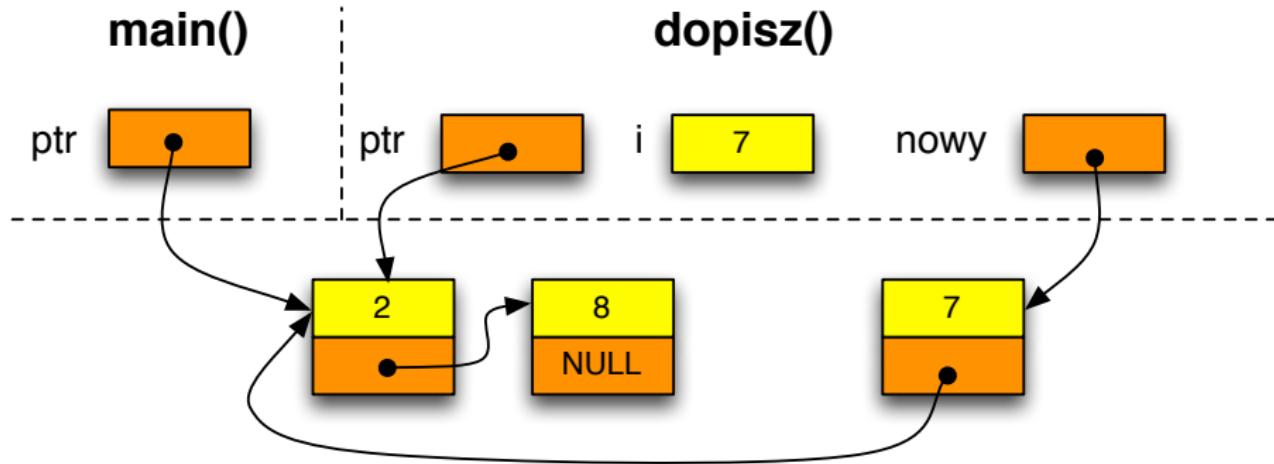
Aby przetestować powyższą funkcję można napisać następującą funkcję main():

```
int main()
{
    List ptr;
    ptr = NULL;
    ptr = dopisz(ptr, 8);
    ptr = dopisz(ptr, 2);
    ptr = dopisz(ptr, 7);
    drukuj(ptr);
    return 0;
}
```

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa – wstawianie na początek

Sytuacja na chwilę przed zakończeniem działania funkcji dopisz() gdy wstawiano liczbę 7:



Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa – wstawianie na początek

Po skompilowaniu i uruchomieniu programu zostaną wydrukowane liczby:

7 2 8

Zwrót uwagę na to, że liczby drukowane są w odwrotnej kolejności niż były dodawane na początek listy.

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa – wstawianie na koniec

Aby wstawić element na koniec listy należy odnaleźć ostatni element doiązać za nim nowy element.

Poniższa funkcja dopisz_na_koniec() wstawia kolejną wartość całkowitą na koniec listy:

```
void dopisz_na_koniec(List *ptr, int i)
{
    List nowy, pomoc;
    nowy = malloc(sizeof(struct Node));
    nowy->value = i;
    nowy->next = NULL;
    if(*ptr == NULL)
        *ptr = nowy;
    else
    {
        pomoc = *ptr;
        while(pomoc->next != NULL) pomoc = pomoc->next;
        pomoc->next = nowy;
    }
}
```

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa – wstawianie na koniec

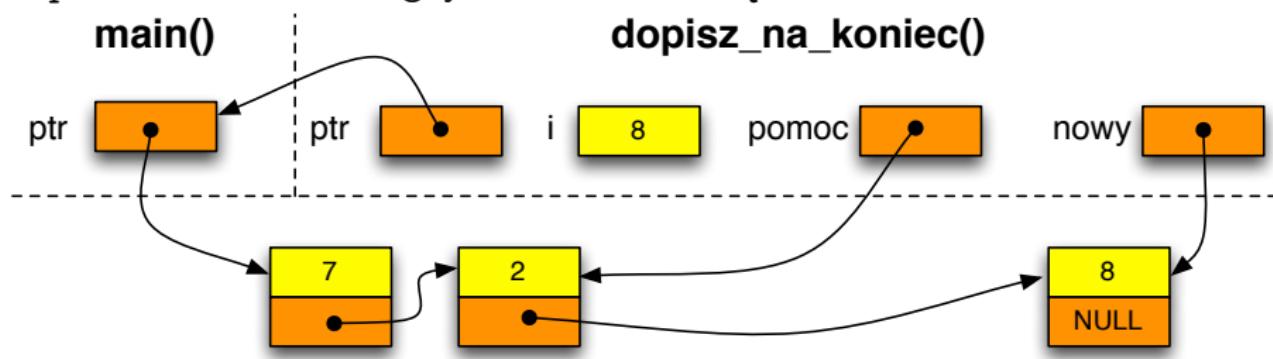
Tym razem parametr formalny ptr nie jest typu List ale typu List*. Aby przetestować powyższą funkcję można napisać następującą funkcję main():

```
int main()
{
    List ptr;
    ptr = NULL;
    dopisz_na_koniec(&ptr, 7);
    dopisz_na_koniec(&ptr, 2);
    dopisz_na_koniec(&ptr, 8);
    drukuj(ptr);
    return 0;
}
```

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa – wstawianie na koniec

Sytuacja na chwilę przed zakończeniem działania funkcji `dopisz_na_koniec()` gdy wstawiano liczbę 8:



Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa – wstawianie na koniec

Po skompilowaniu i uruchomieniu programu zostaną wydrukowane liczby:

7 2 8

Zwrć uwagę na to, że liczby drukowane są w tej samej kolejności w jakiej były wstawiane na koniec listy.

Podstawowe typy danych w C

Struktury dynamiczne: lista jednokierunkowa – usuwanie całej listy

Do usunięcia całej listy z pamięci można użyć poniższą funkcję usun():

```
void usun(List *ptr)
{
    List pomoc;
    while(*ptr != NULL)
    {
        pomoc = (*ptr)->next;
        free(*ptr);
        *ptr = pomoc;
    }
}
```

Funkcję usun() wywołuje się przekazując jej adres &ptr zmiennej ptr wskazującej na początek listy po to aby funkcja mogła zmienić wartość tej zmiennej na NULL.

Wykład 5

Czasowa złożoność obliczeniowa

Czasowa złożoność obliczeniowa

- notacja wielkie O
- notacja małe o
- notacja wielkie Θ
- analiza czasowej złożoności obliczeniowej

Czasowa złożoność obliczeniowa

Notacja O duże

Definition (Notacja O duże)

Niech $f, g : N \mapsto R$. Wtedy

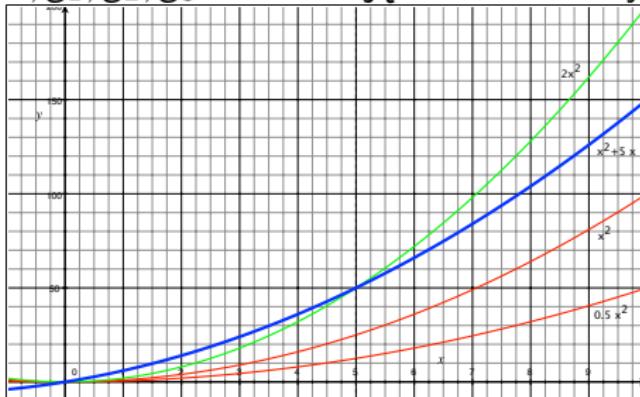
$$f = O(g) \leftrightarrow (\exists c > 0)(\exists k \in N)(\forall n > k)(f(n) < c \cdot g(n)).$$

Czasowa złożoność obliczeniowa

Notacja O duże

Example

Rozpatrzmy funkcję $f(n) = n^2 + 5 \cdot n$ oraz funkcje $g_1(n) = \frac{1}{2} \cdot n^2$, $g_2(n) = n^2$, $g_3(n) = 2 \cdot n^2$. Poniżej przedstawiono wykresy funkcji f , g_1 , g_2 , g_3 rozszerzając ich dziedziny na liczby rzeczywiste.



Dla $n > 5$, $f(n) < 2 \cdot n^2$. Zatem $f(n) = O(n^2)$.

Czasowa złożoność obliczeniowa

Notacja Θ duże

Definition

Niech $f, g : N \mapsto R$. Wtedy

$$f = \Theta(g) \leftrightarrow (f = O(g) \wedge g = O(f)).$$

Czasowa złożoność obliczeniowa

Notacja Θ duże

Theorem

Niech $f, g : N \mapsto R$. Załóżmy, że istnieje granica $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$.

- ① Jeżeli $0 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$, to $f = O(g)$.
- ② Jeżeli $0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$, to $f = \Theta(g)$.

Dowód.

Niech $c = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$. Załóżmy, że $0 \leq c < \infty$. Wówczas dla dostatecznie dużych n , $0 \leq \frac{f(n)}{g(n)} < c + 1$.

Zatem dla dostatecznie dużych n , $f(n) < (c + 1) \cdot g(n)$, więc $f = O(g)$.

Jeśli ponadto $c > 0$, to $0 \leq \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \frac{1}{c} < \infty$, więc z (1) wynika, że $g = O(f)$, czyli $f = \Theta(g)$.



Czasowa złożoność obliczeniowa

Notacja o małe

Definition

Niech $f, g : N \rightarrow R^+$. Wtedy

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

Zamiast pisać $f = o(g)$ będziemy przez chwilę stosować zapis $f \ll g$.
Łatwo sprawdzić, że prawdziwe są następujące zależności:

$$\dots \ll \sqrt[4]{n} \ll \sqrt[3]{n} \ll \sqrt{n} \ll n \ll n^2 \ll n^3 \ll n^4 \ll \dots$$

Czasowa złożoność obliczeniowa

Notacja o małe

Theorem (Reguła de l'Hospitala)

Założymy, że f i g są funkcjami różniczkowalnymi takimi, że $\lim_{x \rightarrow \infty} f(x) = \infty$ oraz $\lim_{x \rightarrow \infty} g(x) = \infty$. Wtedy

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}.$$

Czasowa złożoność obliczeniowa

Notacja o małe

Theorem

$$(\forall \alpha > 0) \log_2 n = o(n^\alpha).$$

Dowód.

Niech $\alpha > 0$. Funkcja $f(x) = \log_2 x$ oraz $g(x) = x^\alpha$ spełniają oba założenia twierdzenia de l'Hospitala. Zatem

$$\lim_{x \rightarrow \infty} \frac{\log_2 x}{x^\alpha} = \lim_{x \rightarrow \infty} \frac{\frac{1}{x \cdot \ln 2}}{\alpha \cdot x^{\alpha-1}} = \lim_{x \rightarrow \infty} \frac{1}{\alpha \cdot \ln 2 \cdot x^\alpha} = 0.$$



Funkcja $\log_2 n$ rośnie więc znacznie wolniej od dowolnego pierwiastka z liczby n :

$$\log_2 n \ll \dots \ll \sqrt[4]{n} \ll \sqrt[3]{n} \ll \sqrt{n} \ll n.$$

Czasowa złożoność obliczeniowa

Analiza

Analiza czasowej złożoności obliczeniowej algorytmu bada jak zależy czas działania algorytmu od rozmiaru jego danych.

Example

Jeśli algorytm przetwarza dane w tablicy, to najczęściej za rozmiar danych przyjmuje się liczbę elementów tej tablicy. Dla przykładu, rozmiarem danych w zagadnieniu sortowania n -elementowej tablicy jest n . Natomiast algorytm przetwarzający sieć złożoną z n węzłów połączonych m łukami ma dane rozmiaru $n + m$.

Z kolei algorytm, który przetwarza tylko jedną liczbę (np. rozkłada ją na czynniki pierwsze), ma dane rozmiaru długości reprezentacji tej liczby. Jeśli zatem nieujemna liczba całkowita n jest jedyną daną dla algorytmu, to przyjmujemy, że dane mają rozmiar $\log_2 n$.

Czasowa złożoność obliczeniowa

Analiza: sortowanie bąbelkowe

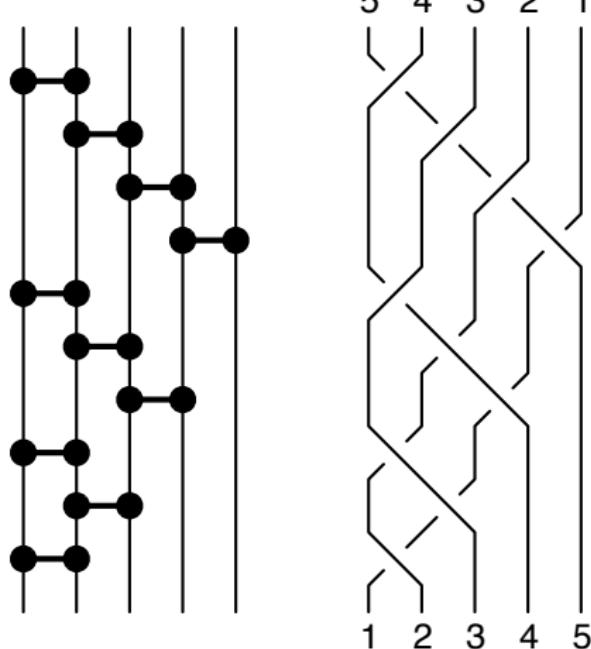
Przeprowadzimy przykładową analizę złożoności czasowej na przykładzie *sortowania bąbelkowego*:

```
1: for  $i \leftarrow n - 1, 1$  do
2:   for  $j \leftarrow 1, i$  do
3:     if  $t[j] > t[j + 1]$  then
4:        $temp \leftarrow t[j];$ 
5:        $t[j] \leftarrow t[j + 1];$ 
6:        $t[j + 1] \leftarrow temp$ 
7:     end if
8:   end for
9: end for
```

Czasowa złożoność obliczeniowa

Analiza: sortowanie bąbelkowe

Poniżej przedstawiono ideę sortowania bąbelkowego w postaci sieci komparatorów.



Czasowa złożoność obliczeniowa

Analiza: sortowanie bąbelkowe

Komparatorowi z sieci sortującej odpowiada instrukcja **if-then** z wierszy od 3 do 7.

W sieci sortującej pięć liczb przedstawionej na rysunku można wyróżnić cztery fazy działania algorytmu:

- faza 1 złożona z czterech porównań;
- faza 2 złożona z trzech porównań;
- faza 3 złożona z dwóch porównań;
- faza 4 złożona z jednego porównania.

Każdej takiej fazie odpowiada wewnętrzna pętla **for** ze zmienną sterującą **j** realizowana w wierszach od 2 do 8.

Czasowa złożoność obliczeniowa

Analiza: sortowanie bąbelkowe

Oznaczmy przez $C(n)$ liczbę porównań wykonywanych w wierszu 3 przy sortowaniu n liczb.

Theorem

$$C(n) = \Theta(n^2).$$

Dowód.

Podczas wykonywania pętli w wierszach 2-8 wykonuje się i porównań.

Zatem łączna liczba porównań jest równa: $C(n) = \sum_{i=1}^{n-1} i$.

Wiemy, że $\sum_{i=1}^n i = \frac{n(n+1)}{2}$, zatem $C(n) = \frac{(n-1)\cdot n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$.

Na mocy wcześniejszego twierdzenia:

$$\lim_{n \rightarrow \infty} \frac{\frac{1}{2}n^2 - \frac{1}{2}n}{n^2} = \lim_{n \rightarrow \infty} \frac{\frac{1}{2}n^2/n^2 - \frac{1}{2}n/n^2}{n^2/n^2} = \lim_{n \rightarrow \infty} \left(\frac{1}{2} - \frac{1}{2n} \right) = \frac{1}{2} < \infty$$

wnioskujemy, że $C(n) = \Theta(n^2)$.



Czasowa złożoność obliczeniowa

Analiza: sortowanie bąbelkowe

Na dalszych wykładach zostanie przedstawiony szybszy algorytm sortowania, który wymaga maksymalnie $O(n \log n)$ operacji porównania.

Czasowa złożoność obliczeniowa

Analiza: wyszukiwanie binarne

Problem (Wyszukiwanie)

- Dana jest n -elementowa tablica zawierająca liczby w porządku niemalejącym tj. $t[1] \leq t[2] \leq \dots \leq t[n]$.
- Dla zadanej wartości x rozstrzygnąć czy znajduje się ona w tablicy.

- ➊ Jeśli porównuje się wartość x z elementem $t[1]$, to w przypadku $x \neq t[1]$ wielkość przeszukiwanego obszaru redukuje się z n do $n - 1$.
- ➋ Jeśli porównuje się wartość x z elementem środkowym $t[(n + 1) \text{ div } 2]$, to w przypadku $x > t[(n + 1) \text{ div } 2]$ wielkość przeszukiwanego obszaru redukuje się o połowę!

Czasowa złożoność obliczeniowa

Analiza: wyszukiwanie binarne

- Jeśli przez $D_0 = n$ przyjmiemy długość początkowego obszaru (rozmiar całej tablicy), to obszary w kolejnych iteracjach algorytmu mają długości D_0, D_1, D_2, \dots w przybliżeniu równe:

$$n, \frac{n}{2}, \frac{n}{4}, \dots, \frac{n}{2^i}, \dots, 2, 1.$$

- Niech $B(n)$ będzie maksymalną liczbą iteracji jaka wykonywana jest podczas binarnego wyszukiwania w tablicy o n elementach.
- Oczywiście $B(n)$ jest nie większe niż maksymalna liczba połowień prowadzących do obszaru rozmiaru 1.
- Można pokazać, że $B(n) = O(\log n)$.
- Ćwiczenie:** Udowodnij, że $B(n) = \lceil \log_2(n + 1) \rceil$, zatem $B(n) = \Theta(\log n)$.

Wykład 6

Rekurencja

Iterować jest rzeczą ludzką, wykonywać rekursywnie – boską.

L. Peter Deutsch

Smok – podsuszony zmok (patrz: Zmok).

Zmok – zmoczony smok (patrz: Smok).

Stanisław Lem „Wizja lokalna”

Rekurencja

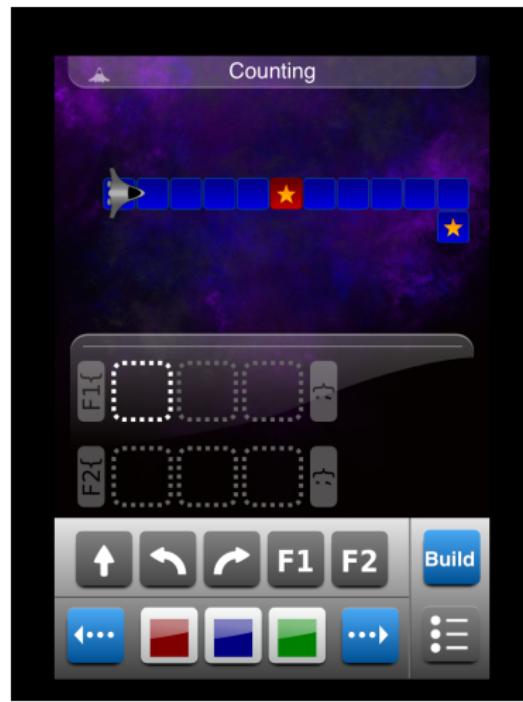
- funkcje rekurencyjne
- rozwiązywanie problemów
- niewłaściwe użycie rekurencji
- postać ogonowa rekurencji
- przekształcenie do postaci ogonowej
- zamiana postaci ogonowej na iterację

Rekurencja

- Rekurencją nazywamy metodę specyfikacji procesu w terminach definiowanego procesu.
- Dokładniej mówiąc, „skomplikowane” przypadki procesu redukowane są do „prostszych” przypadków.
- Stałymi składnikami metod rekurencyjnych są określenie warunku zatrzymania, oraz określenie metody redukcji problemu do prostszych przypadków.

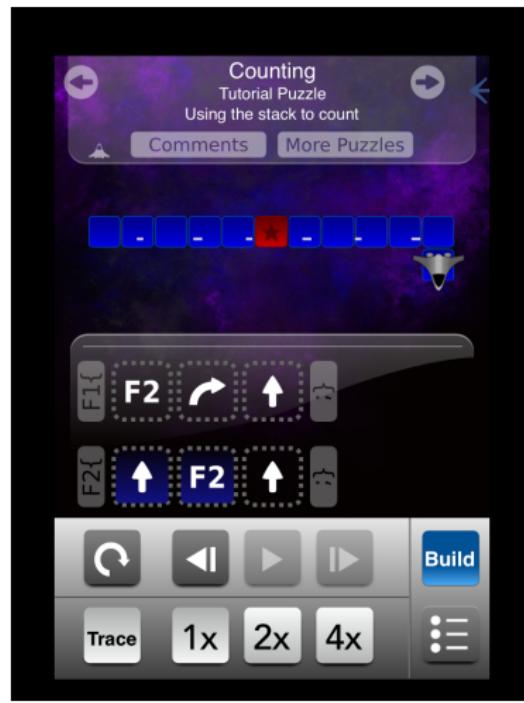
Rekurencja

Przykład z gry Robozzle: jak zebrać obie gwiazdki?



Rekurencja

Przykład z gry Robozzle: rozwiązańie



Rekurencja

Funkcje rekurencyjne

Example (Funkcja silnia)

Klasycznym przykładem funkcji zdefiniowanej rekurencyjnie jest silnia:

$$n! = 1 \cdot 2 \cdot \dots \cdot n$$

Jeśli jednak zauważymy, że $0! = 1$ oraz, że $n! = n \cdot (n - 1)!$, to możemy napisać następującą funkcję:

```
1: function Silnia(n)
2:   if n = 0 then
3:     Silnia ← 1
4:   else
5:     Silnia ← n * Silnia(n - 1)
6:   end if
7: end function
```

Rekurencja

Funkcje rekurencyjne

Example (Funkcja Ackermanna)

Funkcją Ackermanna nazywamy funkcję dwóch zmiennych naturalnych zdefiniowaną wzorem

$$A(n, m) = \begin{cases} m + 1 & \text{gdy } n = 0, \\ A(n - 1, 1) & \text{gdy } n > 0 \wedge m = 0, \\ A(n - 1, A(n, m - 1)) & \text{gdy } n > 0 \wedge m > 0. \end{cases}$$

Uwaga: powyższa funkcja bardzo szybko rośnie. Dla przykładu

$$A(4, 2) = 2^{65536} - 3$$

ma 19729 cyfr.

Ćwiczenie: udowodnij, że dla każdych nieujemnych całkowitych wartości n i m obliczenia funkcji Ackermanna $A(n, m)$ kończą się.

Rekurencja

Funkcje rekurencyjne

Example (Funkcja Ackermana cd.)

W pseudokodzie możemy zapisać ją następująco:

```
1: function A(n, m)
2:   if n = 0 then
3:     A ← m + 1
4:   else
5:     if m = 0 then
6:       A ← A(n - 1, 1)
7:     else
8:       A ← A(n - 1, A(n, m - 1))
9:     end if
10:   end if
11: end function
```

Rekurencja

Rozwiązywanie problemów

- Rekursję można stosować nie tylko do definiowania funkcji.
- Można ją stosować do rozwiązywania problemów.
- W tym przypadku metoda polega na redukcji złożonego problemu na problemy prostsze.

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

- Mamy trzy patyki ponumerowane liczbami 1, 2 i 3 oraz n krążków.
- Krążki są różnych rozmiarów.
- Początkowo wszystkie krążki nawleczone są na pierwszy patyk w malejącej kolejności: największy leży na dole a najmniejszy na samej górze.
- Należy przenieść krążki na trzeci patyk przestrzegając następujących dwóch reguł:
 - ① wolno przekładać tylko jeden krążek w jednym ruchu,
 - ② nigdy większy krążek nie może zostać położony na mniejszym krążku.

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

- Zadanie jest banalne gdy $n = 0$, bo nic nie trzeba wtedy przenosić.
- Rozwiążanie całego zadania stanie się jasne, gdy zauważmy, że jeśli potrafimy przenieść $n - 1$ krążków z patyka i -tego na patyk j -ty, to potrafimy zadanie to rozwiązać dla n krążków.
- Oto strategia:
 - ① wyznacz pomocniczy patyk $k = 6 - (i + j)$;
 - ② przenieś górne $n - 1$ krążków z patyka i -tego na patyk k -ty;
 - ③ przenieś krążek z patyka i -tego na patyk j -ty;
 - ④ przenieś górne $n - 1$ krążków z patyka k -tego na patyk j -ty

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

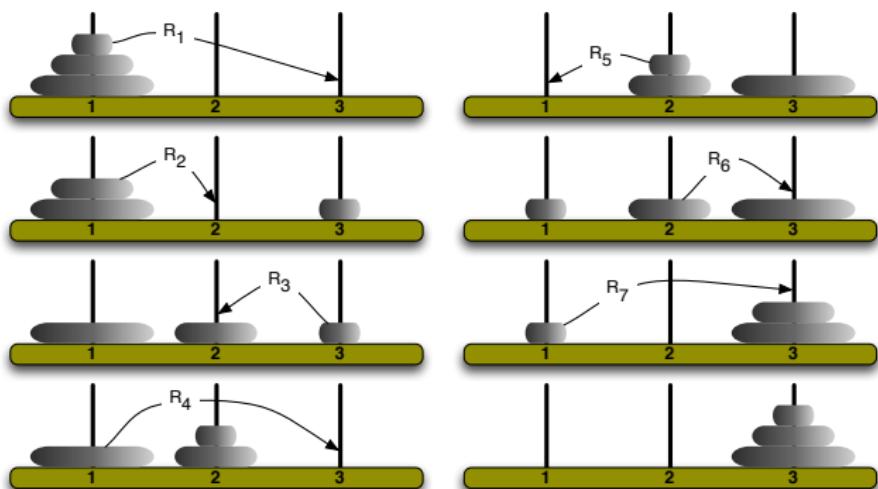
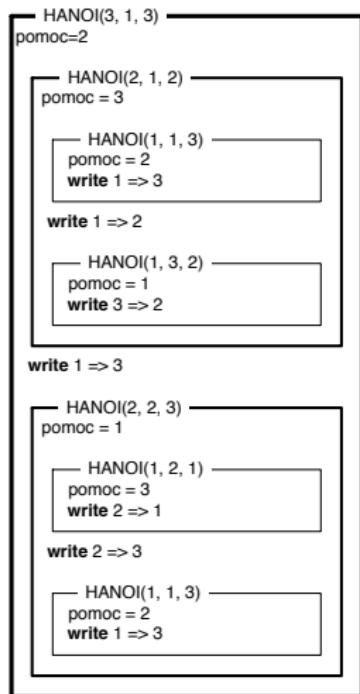
Oto jak tę strategię możemy zamienić na program w pseudokodzie:

```
1: procedure Hanoi(n, skąd, dokąd)
2:   if n > 0 then
3:     pomoc ← 6 – (skąd + dokąd);
4:     Hanoi(n – 1, skąd, pomoc);
5:     write skąd ⇒ dokąd;
6:     Hanoi(n – 1, pomoc, dokąd)
7:   end if
8: end procedure
```

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

Przykład obliczeń Hanoi(3, 1, 3):



Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

- Problem Wież z Hanoi wymyślił Edouard Lucas w roku 1883.
- Według legendy którą wtedy opowiedział, w pewnej hinduskiej świątyni mnisi przekładają bez przerwy układ 64 krążków zgodnie z regułami wież Hanoi.
- Po zrealizowaniu zadania świat ma się zakończyć.

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

- Niech $H(n)$ oznacza ilość operacji przeniesienia krążka, którą wykonuje nasza procedura dla konfiguracji n krążków.
- Oczywiście $H(1) = 1$ oraz $H(n + 1) = H(n) + 1 + H(n) = 2H(n) + 1$.
- Zajmiemy się teraz wyznaczeniem wzoru na $H(n)$.
- Wypiszmy kilka pierwszych wartości:

$$\begin{cases} H(1) = 1 \\ H(2) = 2H(1) + 1 \\ H(3) = 2H(2) + 1 \\ H(4) = 2H(3) + 1 \end{cases}$$

Rekurencja

Rozwiązywanie problemów: wieże z Hanoi

- Pomnóżmy pierwszą równość przez 8, drugą przez 4 a trzecią przez 2.
Otrzymamy

$$\begin{cases} 8H(1) = 8 \\ 4H(2) = 8H(1) + 4 \\ 2H(3) = 4H(2) + 2 \\ H(4) = 2H(3) + 1 \end{cases}$$

- Po zsumowaniu wszystkich równości stronami otrzymamy

$$H(4) = 1 + 2 + 4 + 8 = 1 + 2 + 2^2 + 2^3 = 2^4 - 1,$$

co powinno nam nasunąć następującą hipotezę:

$$(\forall n > 0)(H(n) = 2^n - 1).$$

- Jest ona prawdziwa, co łatwo możemy sprawdzić indukcją matematyczną.

Rekurencja

Niewłaściwe użycie rekurencji

- Pisząc procedurę rekurencyjną musimy zadać o to aby przy każdym dopuszczalnym zestawie parametrów wejściowych zatrzymywała się ona po skończonej liczbie kroków.

```
1: function R(n)
2:     R  $\leftarrow$  R( $n - 1$ )
3: end function
```

- W powyższej funkcji brak przypadku kończącego wywołania rekurencyjne (najprostszego przypadku, który nie wymaga wywołania rekurencyjnego).

Rekurencja

Niewłaściwe użycie rekurencji: współczynnik Newtona

Założymy, że chcemy wyznaczyć wartość *współczynnika Newtona*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

korzystając z dobrze znanej równości Pascala

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

oraz z dwóch dodatkowych wzorów $\binom{n}{0} = \binom{n}{n} = 1$.

Rekurencja

Niewłaściwe użycie rekurencji: współczynnik Newtona

Zapiszemy to w pseudokodzie:

```
1: function Newton0(n, k)
2:   if  $k = 0 \vee k = n$  then
3:     Newton0  $\leftarrow 1$ 
4:   else
5:     Newton0  $\leftarrow$  Newton0( $n - 1, k - 1$ ) + Newton0( $n - 1, k$ )
6:   end if
7: end function
```

Rekurencja

Niewłaściwe użycie rekurencji: współczynnik Newtona

Przyjrzyjmy się jednak, jakie obliczenia będą wykonywane przez tak zaprogramowaną funkcję dla parametrów (7,4):

$$\binom{7}{4} \rightarrow \left\{ \binom{6}{3}, \binom{6}{4} \right\} \rightarrow \left\{ \left\{ \binom{5}{2}, \binom{5}{3} \right\}, \left\{ \binom{5}{3}, \binom{5}{4} \right\} \right\} \rightarrow$$

$$\left\{ \left\{ \left\{ \binom{4}{1}, \binom{4}{2} \right\}, \left\{ \binom{4}{2}, \binom{4}{3} \right\} \right\}, \left\{ \left\{ \binom{4}{2}, \binom{4}{3} \right\}, \left\{ \binom{4}{3}, \binom{4}{4} \right\} \right\} \right\} \rightarrow \dots$$

Prowadzi to do wykładniczej liczby wywołań rekurencyjnych, które wielokrotnie liczą tę samą wartość.

Rekurencja

Niewłaściwe użycie rekurencji: współczynnik Newtona

Można skorzystać z innej zależności:

$$\binom{n}{k} = \binom{n-1}{k-1} \cdot \frac{n}{k},$$

który prowadzi do następującego kodu:

```
1: function Newton(n, k)
2:   if  $k = 0 \vee k = n$  then
3:     Newton  $\leftarrow 1$ 
4:   else
5:     Newton  $\leftarrow (\text{Newton}(n-1, k-1) * n) \text{ div } k$ 
6:   end if
7: end function
```

o czasowej złożoności obliczeniowej $O(n)$.

Rekurencja

Niewłaściwe użycie rekurencji: ciąg Fibonacciego

Rozpatrzmy przykład ciągu Fibonacciego, w którym kolejny wyraz jest sumą dwóch poprzednich:

```
1: function Fib0(n)
2:   if n = 0 then
3:     Fib0 ← 0
4:   else
5:     if n = 1 then
6:       Fib0 ← 1
7:     else
8:       Fib0 ← Fib0(n - 1) + Fib0(n - 2)
9:     end if
10:    end if
11: end function
```

Rekurencja

Niewłaściwe użycie rekurencji: ciąg Fibonacciego

$$\begin{aligned}\text{Fib0(5)} &= \text{Fib0(4)} + \text{Fib0(3)} \\&= \text{Fib0(3)} + \text{Fib0(2)} + \text{Fib0(3)} \\&= \text{Fib0(2)} + \text{Fib0(1)} + \text{Fib0(2)} + \text{Fib0(3)} \\&= \text{Fib0(1)} + \text{Fib0(0)} + \text{Fib0(1)} + \text{Fib0(2)} + \text{Fib0(3)} \\&= 1 + \text{Fib0(0)} + \text{Fib0(1)} + \text{Fib0(2)} + \text{Fib0(3)} \\&= 1 + 0 + \text{Fib0(1)} + \text{Fib0(2)} + \text{Fib0(3)} \\&= 1 + 0 + 1 + \text{Fib0(2)} + \text{Fib0(3)} \\&= 1 + 0 + 1 + \text{Fib0(1)} + \text{Fib0(0)} + \text{Fib0(3)} \\&= 1 + 0 + 1 + 1 + \text{Fib0(0)} + \text{Fib0(3)} \\&= 1 + 0 + 1 + 1 + 0 + \text{Fib0(3)} \\&= 1 + 0 + 1 + 1 + 0 + \text{Fib0(2)} + \text{Fib0(1)} \\&= 1 + 0 + 1 + 1 + 0 + \text{Fib0(1)} + \text{Fib0(0)} + \text{Fib0(1)} \\&= 1 + 0 + 1 + 1 + 0 + 1 + \text{Fib0(0)} + \text{Fib0(1)} \\&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + \text{Fib0(1)} \\&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 = 5\end{aligned}$$

Rekurencja

Niewłaściwe użycie rekurencji: ciąg Fibonacciego

```
1: function Fib(n)
2:   if n = 0 then
3:     Fib ← 0
4:   else
5:     if n = 1 then
6:       Fib ← 1
7:     else
8:       starsza ← 0; stara ← 1;
9:       for i ← 2, n do
10:         nowa ← stara + starsza;
11:         starsza ← stara; stara ← nowa
12:       end for
13:       Fib ← nowa
14:     end if
15:   end if
16: end function
```

Rekurencja

Niewłaściwe użycie rekurencji: ciąg Fibonacciego

Uwaga

Stosując operacje na macierzach, jakie poznałeś na kursie **Algebra z geometrią analityczną**, można policzyć n -ty wyraz ciągu Fibonacciego za pomocą $O(\log n)$ iteracji.

Ćwiczenie: Niech macierz A ma następujące wartości elementów:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Jakie są wartości elementów macierzy A^n , tj. w n -tej potędze macierzy A ?

Rekurencja

Postać ogonowa

Powróćmy do rekurencyjnej funkcji Silnia(n) obliczającej $n!$.

Poniżej przedstawiono przebieg obliczeń Silnia(5):

$$\begin{aligned}\text{Silnia}(5) &= 5 * \text{Silnia}(4) \\&= 5 * 4 * \text{Silnia}(3) \\&= 5 * 4 * 3 * \text{Silnia}(2) \\&= 5 * 4 * 3 * 2 * \text{Silnia}(1) \\&= 5 * 4 * 3 * 2 * 1 * \text{Silnia}(0) \\&= 5 * 4 * 3 * 2 * 1 * 1 \\&= 5 * 4 * 3 * 2 * 1 \\&= 5 * 4 * 3 * 2 \\&= 5 * 4 * 6 \\&= 5 * 24 = 120\end{aligned}$$

Rekurencja

Postać ogonowa

- Potrzeba przemnażania wartości $Silnia(n - 1)$ przez n wymaga przechowywania zmiennej n na stosie (po powrocie z rekurencyjnego wywołania zmienna n musi mieć taką wartość jak przed rekurencyjnym wywołaniem).
- Gdyby po powrocie z rekurencyjnego wywołania nie wykonywała się już żadna operacja na zmiennych lokalnych, to nie zachodziłaby potrzeba przechowywania ich na stosie.
- Aby było to możliwe funkcja rekurencyjna powinna spełniać następujące warunki:
 - ① Rekurencyjne wywołanie funkcji znajduje się w jednym miejscu jej definicji.
 - ② Po powrocie z rekurencyjnego wywołania nie mogą być już wykonywane żadne operacje na zmiennych lokalnych funkcji.

Rekurencja

Przekształcanie do postaci ogonowej

```
1: function Silnia2(n, akumulator)
2:   if n = 0 then
3:     Silnia2  $\leftarrow$  akumulator
4:   else
5:     Silnia2  $\leftarrow$  Silnia2(n - 1, n * akumulator)
6:   end if
7: end function
8:
9: function Silnia(n)
10:   Silnia  $\leftarrow$  Silnia2(n, 1)
11: end function
```

Rekurencja

Przekształcanie do postaci ogonowej

Przebieg obliczeń:

$$\begin{aligned}\text{Silnia}(5) &= \text{Silnia2}(5, 1) \\ &= \text{Silnia2}(4, 5) \\ &= \text{Silnia2}(3, 20) \\ &= \text{Silnia2}(2, 60) \\ &= \text{Silnia2}(1, 120) \\ &= \text{Silnia2}(0, 120) \\ &= 120\end{aligned}$$

Rekurencja

Zamiana postaci ogonowej na iterację

```
1: function Silnia(n)
2:     akumulator ← 1;
3:     while  $n \neq 0$  do
4:         akumulator ← akumulator * n;
5:          $n \leftarrow n - 1$ 
6:     end while
7:     Silnia ← akumulator
8: end function
```

Wykład 7

Zasada dziel i zwyciężaj

Zasada dziel i zwyciężaj

- schemat metody
- analiza złożoności
- przykłady algorytmów i ich analiza

Zasada dziel i zwyciężaj

Schemat metody

Metoda rozwiązywania problemów „dziel i zwyciężaj” (*divide and conquer*) polega na rozbiciu zadania na mniejsze pod-zadania, a po ich rozwiązaniu, przeprowadzeniu syntezy rozwiązania całego zadania z rozwiązań mniejszych pod-zadań:

```
1: function RozwiązańeZłożonegoProblemu( $P$ )
2:   if problem  $P$  ma proste rozwiązanie then
3:     RozwiązańeZłożonegoProblemu  $\leftarrow$  Rozwiązańe( $P$ )
4:   else
5:     podziel problem  $P$  na podproblemy  $P_1, P_2, \dots, P_n$ ;
6:     for  $i \leftarrow 1, n$  do
7:        $R_i \leftarrow$  RozwiązańeZłożonegoProblemu( $P_i$ )
8:     end for
9:     RozwiązańeZłożonegoProblemu  $\leftarrow$  Synteza( $R_1, R_2, \dots, R_n$ )
10:    end if
11:  end function
```

Zasada dziel i zwyciężaj

Analiza złożoności

Theorem (O rekurencji uniwersalnej)

Założymy, że a i b są liczbami dodatnimi oraz c dodatnią liczbą naturalną takimi, że T jest funkcją, która dla potęgi liczby c spełnia równanie:

$$T(n) = \begin{cases} b & \text{gdy } n = 1, \\ a \cdot T\left(\frac{n}{c}\right) + b \cdot n & \text{gdy } n > 1. \end{cases}$$

Wtedy ze względu na relację między wartościami a i c mamy:

- ① Jeśli $a < c$, to $T = O(n)$.
- ② Jeśli $a = c$, to $T = O(n \log n)$.
- ③ Jeśli $a > c$, to $T = O(n^{\log_c a})$.

Zasada dziel i zwyciężaj

Analiza złożoności

Dowód.

- ① Indukcyjnie pokazuje się, że dla $n \in N$:

$$T(c^n) = bc^n \sum_{i=0}^n \left(\frac{a}{c}\right)^i.$$

- ② Dla $k = c^n$, w zależności od wartości a/c :

$a/c < 1$ wówczas $\sum_{i=0}^{\infty} \left(\frac{a}{c}\right)^i < \infty$, więc $T(k) = C \cdot k$

$a/c = 1$ wówczas $\sum_{i=0}^{\log_c k} \left(\frac{a}{c}\right)^i = \log_c k + 1$, więc $T(k) = O(k \log k)$

$a/c > 1$ wówczas $T(k) = bk \frac{\left(\frac{a}{c}\right)^{\log_c k+1} - 1}{\frac{a}{c} - 1} \simeq Ck \left(\frac{a}{c}\right)^{\log_c k} = Ckc^{\log_c(a/c) \cdot \log_c k} = D \cdot k^{\log_c a}$



Zasada dziel i zwyciężaj

Analiza złożoności

- Funkcja $T(n)$ z twierdzenia o rekurencji uniwersalnej, może wyrażać liczbę operacji wykonywanych przez funkcję RozwiążanieZłożonegoProblemu(P), dla problemu P rozmiaru n .
- Stała b odpowiada liczbie operacji wykonywanych przez funkcję Rozwiążanie(P).
- Stała a odpowiada liczbie podproblemów na jaką dzielimy problem P , natomiast stała c wskazuje ilu krotnie mniejszych rozmiarów są to podproblemy w stosunku do rozmiaru problemu P .

Example

Jeśli dzielimy problem P na trzy podproblemy o połowę mniejszych rozmiarów, to $a = 3$ i $c = 2$.

Zasada dziel i zwyciężaj

Przykłady analizy: minimalny i maksymalny element

Wyszukiwanie najmniejszej wartości \min w tablicy $A[1..n]$ można wykonać za pomocą $n - 1$ porównań:

```
1: min  $\leftarrow A[1];
2: for i  $\leftarrow 2, n$  do
3:   if A[i]  $< \min$  then
4:     min  $\leftarrow A[i]
5:   end if
6: end for$$ 
```

Dla znalezienie wartości największej można wykonać kolejnych $n - 1$ porównań.

Wówczas wykonanych zostanie $2 \cdot n - 2$ porównań.

Zasada dziel i zwycięzaj

Przykłady analizy: minimalny i maksymalny element

```
1: procedure MinMax(A, L, P, out min, out max)
2:   if L = P then
3:     min ← A[L]; max ← A[L]
4:   else
5:     if L + 1 = P then
6:       min ← min(A[L], A[P]);
7:       max ← max(A[L], A[P]);
8:     else
9:       S ← (L + P) div 2;
10:      MinMax(A, L, S, min1, max1);
11:      MinMax(A, S + 1, P, min2, max2);
12:      min ← min(min1, min2);
13:      max ← max(max1, max2);
14:    end if
15:  end if
16: end procedure
```

Zasada dziel i zwyciężaj

Przykłady analizy: minimalny i maksymalny element

Założmy, że mamy znaleźć minimalne i maksymalne wartości z tablicy rozmiaru $n = 2^k$. Niech $T(n)$ oznacza ilość potrzebnych do tego celu porównań.

Jeśli $k = 1$, to wystarczy jedno porównanie, zatem $T(2) = 1$.

Założmy teraz, że mamy daną tablicę A długości $2n$. Za pomocą $T(n)$ porównań możemy wyznaczyć minimum \min_1 i maksimum \max_1 z $A[1..n]$ oraz za pomocą tej samej liczby porównań możemy wyznaczyć odpowiednie wartości \min_2 i \max_2 z tablicy $A[n + 1..2n]$.

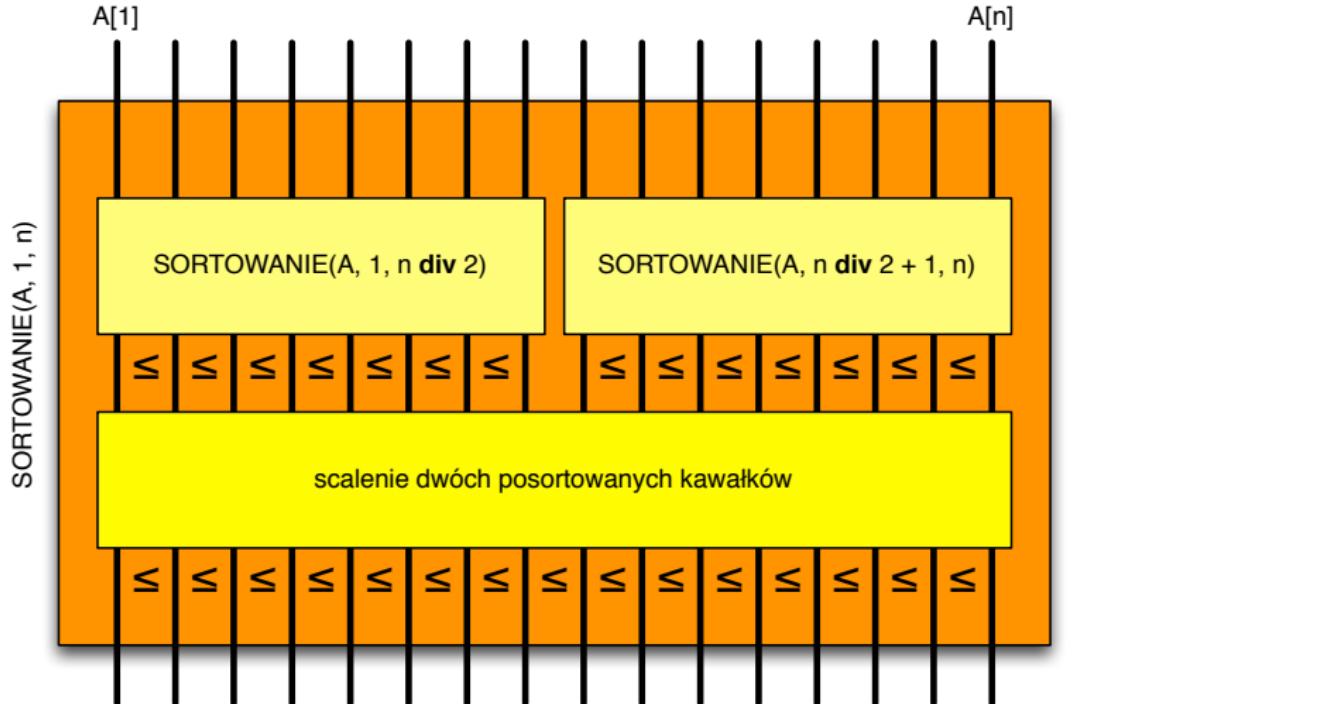
Jeśli $\min_1 < \min_2$ to minimalną wartością jest \min_1 a w przeciwnym przypadku jest nią \min_2 . Podobnie, jeśli $\max_1 > \max_2$, to maksymalną wartością jest \max_1 a w przeciwnym wypadku jest nią \max_2 . Zatem

$$T(2n) = 2 \cdot T(n) + 2.$$

Łatwo sprawdzić metodą indukcji matematycznej, że dla n będących potęgami dwójką mamy $T(n) = \frac{3}{2}n - 2 < 2n - 2$.

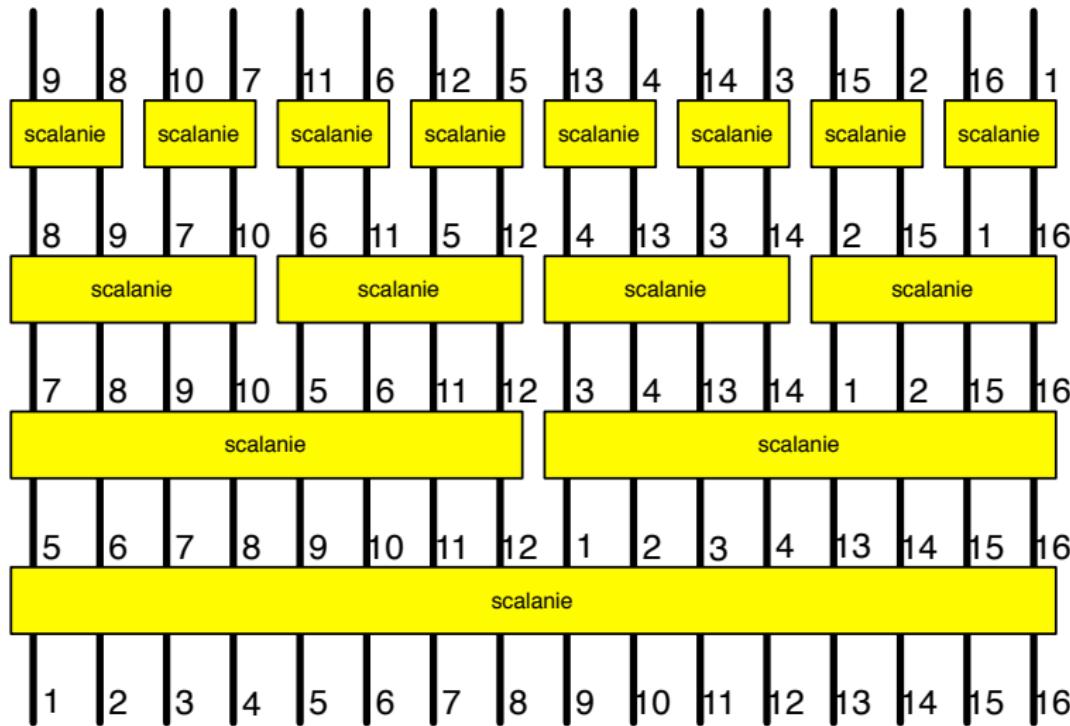
Zasada dziel i zwycięzaj

Przykłady analizy: sortowanie przez scalanie



Zasada dziel i zwycięzaj

Przykłady analizy: sortowanie przez scalanie



Zasada dziel i zwycięzaj

Przykłady analizy: sortowanie przez scalanie

```
1: procedure Sortowanie(A, L, P)
2:   if L < P then
3:     S ← (L + P) div 2; Sortowanie(A, L, S); Sortowanie(A, S + 1, P);
4:     i ← L; j ← S + 1; k ← L;
5:     while i ≤ S ∧ j ≤ P do
6:       if A[i] < A[j] then
7:         Rob[k] ← A[i]; k ← k + 1; i ← i + 1
8:       else
9:         Rob[k] ← A[j]; k ← k + 1; j ← j + 1
10:      end if
11:      end while
12:      while i ≤ S do
13:        Rob[k] ← A[i]; k ← k + 1; i ← i + 1
14:      end while
15:      while j ≤ P do
16:        Rob[k] ← A[j]; k ← k + 1; j ← j + 1
17:      end while
18:      for k ← L, P do
19:        A[k] ← Rob[k]
20:      end for
21:    end if
22: end procedure
```

Zasada dziel i zwyciężaj

Przykłady analizy: sortowanie przez scalanie

Theorem

Tablicę rozmiaru n można posortować za pomocą $O(n \log n)$ porównań i podstawień.

Dowód.

- Niech $C(n)$ oznacza liczbę porównań i podstawień podczas sortowania n elementów.
- Założmy, że n jest potęgą liczby 2.
- Wówczas $C(1) = 0$, natomiast $C(2 \cdot n) = 2 \cdot C(n) + D \cdot n$.
- Zatem, z twierdzenia o rekurencji uniwersalnej, $C(n) = O(n \log n)$.



Zasada dziel i zwyciężaj

Przykłady analizy: operacje na dużych liczbach

- Nieujemne liczby całkowite można przechowywać w tablicy n -elementowej, w której $A[0]$ jest najmniej znaczącą cyfrą dziesiętną, natomiast $A[n - 1]$ jest najbardziej znaczącą cyfrą dziesiętną.
- Wartość zapisana w takiej tablicy jest równa:

$$\sum_{i=0}^{n-1} 10^i \cdot A[i].$$

Zasada dziel i zwyciężaj

Przykłady analizy: operacje na dużych liczbach

Wynik dodawania dwóch n -cyfrowych liczb może mieć $n + 1$ cyfr i dlatego poniższa funkcja zwraca ostatnie przeniesienie, na które nie ma miejsca w tablicy C , jako swoją wartość:

```
1: function Dodaj(A, B, out C, n)
2:     przeniesienie ← 0;
3:     for  $i \leftarrow 0, n - 1$  do
4:         suma ←  $A[i] + B[i] + \text{przeniesienie}$ ;
5:          $C[i] \leftarrow \text{suma} \bmod 10$ ;
6:         przeniesienie ← suma div 10
7:     end for
8:     Dodaj ← przeniesienie
9: end function
```

Funkcja zawiera jedną pętlę wykonywaną n razy, zatem jej czasowa złożoność obliczeniowa jest $O(n)$.

Zasada dziel i zwyciężaj

Przykłady analizy: operacje na dużych liczbach

76147654

x 2611

$$\begin{array}{r} 76147654 \\ \times 2611 \\ \hline \end{array} = 1 * 76147654 * 1$$

$$\begin{array}{r} 761476540 \\ \times 2611 \\ \hline \end{array} = 1 * 76147654 * 10$$

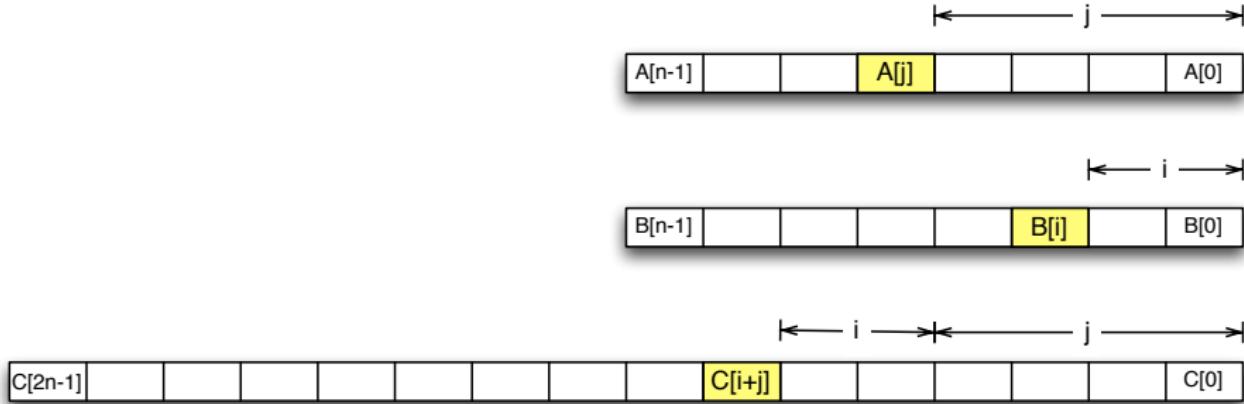
$$\begin{array}{r} 45688592400 \\ \times 2611 \\ \hline \end{array} = 6 * 76147654 * 100$$

$$\begin{array}{r} + 152295308000 \\ \hline \end{array} = 2 * 76147654 * 1000$$

198821524594

Zasada dziel i zwyciężaj

Przykłady analizy: operacje na dużych liczbach



suma $\leftarrow C[i+j] + A[j]*B[i] + \text{przeniesienie}$
 $C[i+j] \leftarrow \text{suma mod } 10$
 $\text{przeniesienie} \leftarrow \text{suma div } 10$

Zasada dziel i zwyciężaj

Przykłady analizy: operacje na dużych liczbach

Wynik mnożenia dwóch liczb n -cyfrowych może mieć $2 \cdot n$ cyfr:

```
1: procedure Pomnóż(A, B, out C, n)
2:   for  $i \leftarrow 0, 2 * n - 1$  do
3:      $C[i] \leftarrow 0$ 
4:   end for
5:   for  $i \leftarrow 0, n - 1$  do
6:     przeniesienie  $\leftarrow 0$ ;
7:     for  $j \leftarrow 0, n - 1$  do
8:       suma  $\leftarrow C[i + j] + B[i] * A[j] + przeniesienie$ ;
9:        $C[i + j] \leftarrow suma \bmod 10$ ;
10:      przeniesienie  $\leftarrow suma \bmod 10$ ;
11:    end for
12:  end for
13: end procedure
```

Zasada dziel i zwyciężaj

Przykłady analizy: algorytm szybkiego mnożenia Karatsuby

- Omówimy teraz szybszą metodę, wymyśloną przez Karatsubę w roku 1962.
- Założmy, że chcemy pomnożyć dwie długie liczby x i y całkowite zapisane w układzie o podstawie 10.
- Założmy, że obie liczby mają parzystą liczbę $n = 2m$ cyfr (jeśli nie, to dodajmy zera z ich lewej strony). Możemy je zapisać w postaci

$$x = 10^m \cdot x_1 + x_2, y = 10^m \cdot y_1 + y_2.$$

gdzie wszystkie liczby x_1, x_2, y_1, y_2 są już m -cyfrowe.

- Wtedy

$$x \cdot y = 10^{2m} x_1 y_1 + 10^m(x_1 y_2 + x_2 y_1) + x_2 y_2,$$

więc powinniśmy umieć szybko wyznaczyć liczby $x_1 y_1$, $x_1 y_2 + x_2 y_1$ i $x_2 y_2$.

Zasada dziel i zwyciężaj

Przykłady analizy: algorytm szybkiego mnożenia Karatsuby

- Karatsuba zauważył, że cztery iloczyny x_1y_1 , $x_1y_2 + x_2y_1$ i x_2y_2 można wyznaczyć wykonując trzy mnożenia.
- Niech $A = x_1y_1$, $B = x_2y_2$ oraz $C = (x_1 + x_2)(y_1 + y_2)$.
- Liczbę $x_1y_2 + x_2y_1$ można wyliczyć z liczb A , B i C za pomocą jednego dodawania i jednego odejmowania:

$$C - (A + B) = x_1y_2 + x_2y_1.$$

Zasada dziel i zwyciężaj

Przykłady analizy: algorytm szybkiego mnożenia Karatsuby

- Aby wyznaczyć produkty liczb m -cyfrowych można zastosować rekurencyjnie ten sam trik.
- Niech $T(n)$ oznacza czas potrzebny do pomnożenia dwóch n -cyfrowych liczb metodą Karatsuby. Wtedy

$$T(n) \leq 3 \cdot T\left(\frac{n}{2}\right) + b \cdot n.$$

dla pewnej stałej b .

Theorem

Algorytm Karatsuby działa w czasie $O(n^{\log_2 3})$.

Zasada dziel i zwyciężaj

Przykłady analizy: algorytm szybkiego mnożenia Karatsuby

Dowód.

Na mocy twierdzenia o rekursji uniwersalnej, jeśli funkcja T spełnia równanie $T(n) = 3 \cdot T\left(\frac{n}{2}\right) + b \cdot n$, to $T(n) = O(n^{\log_2 3})$ dla liczb n będących potęgą liczby c .



Zauważ, że $\log_2 3 = \frac{\log_{10} 3}{\log_{10} 2} \simeq 1.585$, zatem $n^{\log_2 3} = o(n^2)$. Algorytm Karatsuby jest więc znacznie szybszy od prostego „ręcznego” algorytmu działającego w czasie $O(n^2)$.

Znane są jeszcze szybsze metody mnożenia dużych liczb naturalnych. Algorytm Schönhagena - Strassena z roku 1972 działa w czasie $O(n \cdot \log n \cdot \log \log n)$. Wykorzystuje on stosunkowo zaawansowane narzędzia matematyczne, a mianowicie transformacje Fouriera.

Wykład 8

Przeszukiwanie z nawrotami

Przeszukiwanie z nawrotami

- sformułowanie problemu
- przegląd drzewa poszukiwań
- przykłady problemów
- wybrane narzędzia programistyczne

Przeszukiwanie z nawrotami

Sformułowanie problemu

- Dany zbiór wartości D nazywany dziedziną.
- Niech $\Omega = D^n$.
- Na zbiorze Ω określona jest funkcja $C : \Omega \mapsto \{\text{true}, \text{false}\}$ wyznaczająca zbiór rozwiązań dopuszczalnych $S \subseteq \Omega$:

$$S = \{\langle x_1, x_2, \dots, x_n \rangle \in \Omega : C(x_1, x_2, \dots, x_n)\}.$$

- Warunek logiczny C , wyznaczający zbiór rozwiązań dopuszczalnych, nazywać będziemy *ograniczeniami*.

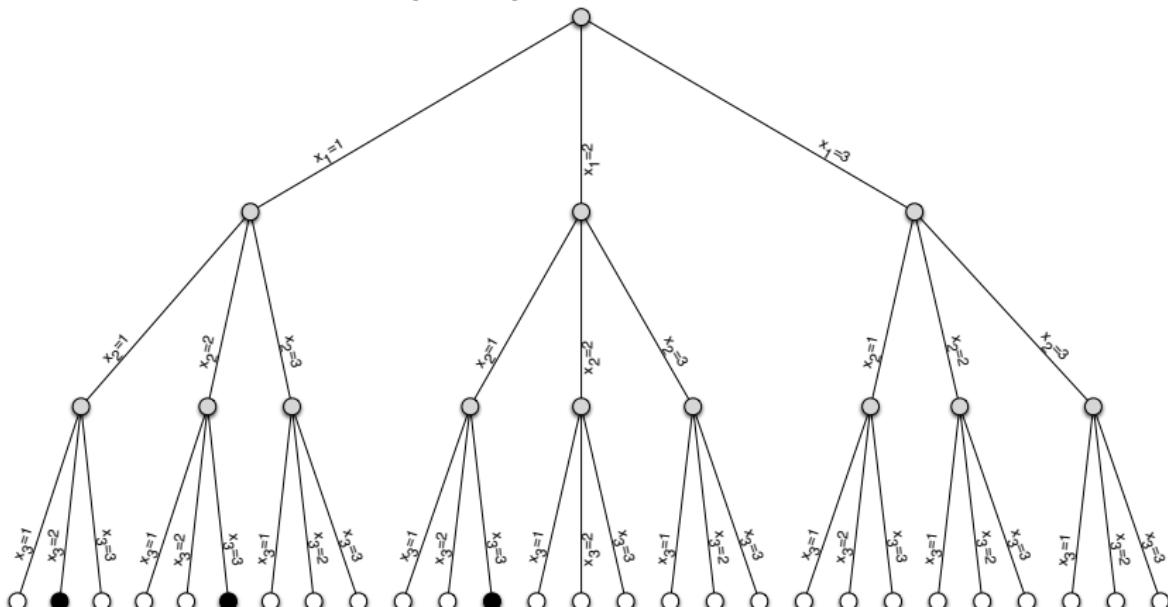
Problem (Problem spełnienia ograniczeń)

Interesuje nas znalezienie elementów zbioru S (wszystkich albo co najmniej jednego).

Przeszukiwanie z nawrotami

Przegląd drzewa poszukiwań

Przykład dla $n = 3$, $D = \{1, 2, 3\}$ i ograniczenia $x_1 + x_2 = x_3$:



Kolorem czarnym zaznaczono liście odpowiadające zbiorowi rozwiązań:

$$S = \{\langle x_1, x_2, x_3 \rangle \in D^3 : x_1 + x_2 = x_3\}$$

Przeszukiwanie z nawrotami

Przegląd drzewa poszukiwań

Poniższy program dokonuje przeglądu wszystkich dopuszczalnych rozwiązań dla przykładu z poprzedniego slajdu.

```
1: for  $X[1] \leftarrow 1, 3$  do
2:   for  $X[2] \leftarrow 1, 3$  do
3:     for  $X[3] \leftarrow 1, 3$  do
4:       if  $X[1] + X[2] = X[3]$  then
5:         write  $X[1], X[2], X[3]$ 
6:       end if
7:     end for
8:   end for
9: end for
```

Przeszukiwanie z nawrotami

Przegląd drzewa poszukiwań

W przypadku dowolnej liczby zmiennych n i dziedziny k -elementowej $D = \{1, 2, \dots, k\}$ potrzebna jest procedura dokonująca przeglądu z nawrotami (zmienna j jest lokalna w tej procedurze):

```
1: procedure PEŁENPRZEGŁĄD(n, i) ▷ ustalenie wartości  $X[1], \dots, X[n]$ 
2:   if  $i = n + 1$  then
3:     if  $C(X[1], X[2], \dots, X[n])$  then
4:       drukuj wartości  $X[1], X[2], \dots, X[n]$ 
5:     end if
6:   else
7:     for  $X[i] \leftarrow 1, k$  do
8:       PEŁENPRZEGŁĄD( $n, i + 1$ )
9:     end for
10:   end if
11: end procedure
```

Przeszukiwanie z nawrotami

Przegląd drzewa poszukiwań

PEŁENPRZEGŁĄD(n , i):

for $X[i] \leftarrow 1, k$ do

 PEŁENPRZEGŁĄD(n , $i+1$)

end for

PEŁENPRZEGŁĄD($3, 1$):

for $X[1] \leftarrow 1, k$ do

 PEŁENPRZEGŁĄD($3, 2$):

 for $X[2] \leftarrow 1, k$ do

 PEŁENPRZEGŁĄD($3, 3$):

 for $X[3] \leftarrow 1, k$ do

 PEŁENPRZEGŁĄD($3, 4$):

 if $C(X[1], X[2], X[3])$ then

 write $X[1], X[2], X[3]$

 end if

 end for

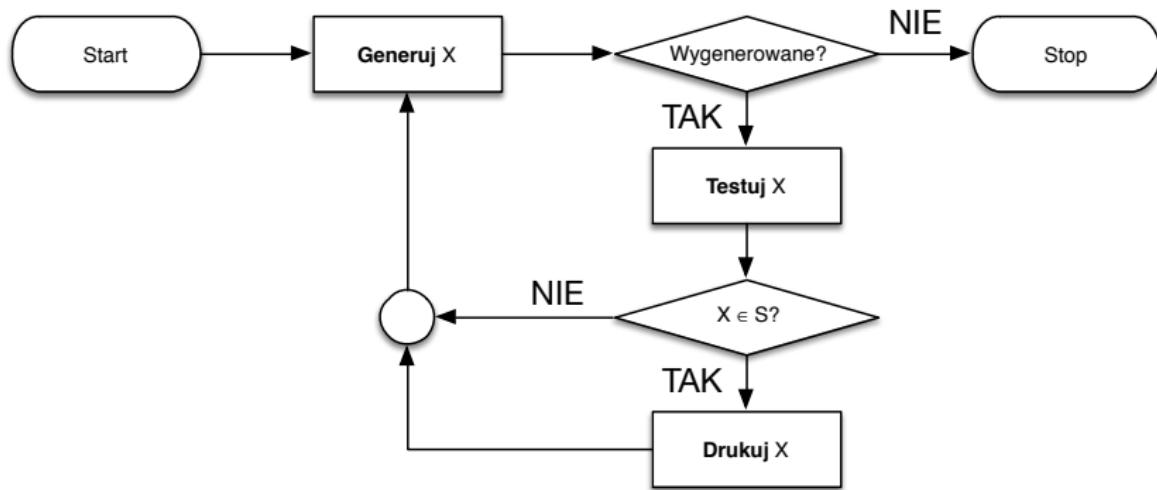
 end for

 end for

Przeszukiwanie z nawrotami

Przegląd drzewa poszukiwań

Procedura PEŁENPRZEGŁĄD pracuje bardzo nieefektywnie bo najpierw generuje cały układ wartości $X[1], X[2], \dots, X[n]$ i dopiero na koniec sprawdza czy spełnia on ograniczenia C (takie przeszukiwanie nazywa się *generowaniem i testowaniem*).



Przeszukiwanie z nawrotami

Przegląd drzewa poszukiwań

- Niech warunek $\text{MOŻLIWE}(X[1], X[2], \dots, X[i])$, dla $i \leq n$, jest spełniony gdy istnieją takie wartości $X[i+1], X[i+2], \dots, X[n]$, że $C(X[1], X[2], \dots, X[i], \dots, X[n])$.
- Warunek MOŻLIWE można zinterpretować w ten sposób, że ocenia on węzeł drzewa poszukiwań i przewiduje, czy w poddrzewie zakorzenionym w ocenianym węźle znajduje się liść odpowiadający rozwiązaniu dopuszczalnemu (elementowi ze zbioru S).
- Ponieważ najczęściej aby być pewnym odpowiedzi z warunku MOŻLIWE musiałby on przejrzeć całe rozpatrywane poddrzewo, więc będziemy zakładać, że warunek MOŻLIWE udziela bez pełnego przeglądu jedynie **niepewnej** odpowiedzi tj.
 - ❶ jeśli warunek nie jest spełniony, to w analizowanym poddrzewie **na pewno** nie ma rozwiązania dopuszczalnego;
 - ❷ jeśli warunek jest spełniony, to w analizowanym poddrzewie **może** jest rozwiązanie dopuszczalne.

Przeszukiwanie z nawrotami

Przegląd drzewa poszukiwań

Poniższa procedura dokonuje częściowego przeglądu, co może w wielu sytuacjach bardzo przyspieszyć poszukiwania:

```
1: procedure CZĘŚCIOWYPRZEGŁĄD(n, i)
2:   if  $i = n + 1$  then
3:     drukuj wartości  $X[1], X[2], \dots, X[n]$ 
4:   else
5:     for  $X[i] \leftarrow 1, k$  do
6:       if MOŻLIWE( $X[1], X[2], \dots, X[i]$ ) then
7:         CZĘŚCIOWYPRZEGŁĄD( $n, i + 1$ )
8:       end if
9:     end for
10:    end if
11: end procedure
```

Przeszukiwanie z nawrotami

Przegląd drzewa poszukiwań

- Procedura CZĘŚCIOWYPRZEGŁĄD działa przy dodatkowym założeniu, że warunek

$$\text{MOŻLIWE}(X[1], X[2], \dots, X[n])$$

poprawnie sprawdza spełnienie ograniczeń C przy ustalonych wszystkich n wartościach.

- Często łatwiej rozstrzygnąć warunek $\text{NIEMOŻLIWE}(X[1], X[2], \dots, X[i])$, który zachodzi gdy nie można rozszerzyć wartości $X[1], X[2], \dots, X[i]$ do n wartości spełniających ograniczenia C .
- Oczywiście taka odpowiedź od warunku NIEMOLIWE również nie jest pewna tj.
 - jeśli warunek nie jest spełniony, to w analizowanym poddrzewie **może** jest rozwiązanie dopuszczalne;
 - jeśli warunek jest spełniony, to w analizowanym poddrzewie **na pewno** nie ma rozwiązania dopuszczalnego.

Przeszukiwanie z nawrotami

Przegląd drzewa poszukiwań

- Jeśli potrafimy sformułować warunek NIEMOŻLIWE, to warunek

$$\text{MOŻLIWE}(X[1], X[2], \dots, X[i]),$$

z wiersza 6 w procedurze CZĘŚCIOWYPRZEGŁĄD, można zastąpić warunkiem:

$$\neg \text{NIEMOŻLIWE}(X[1], X[2], \dots, X[i]).$$

Przeszukiwanie z nawrotami

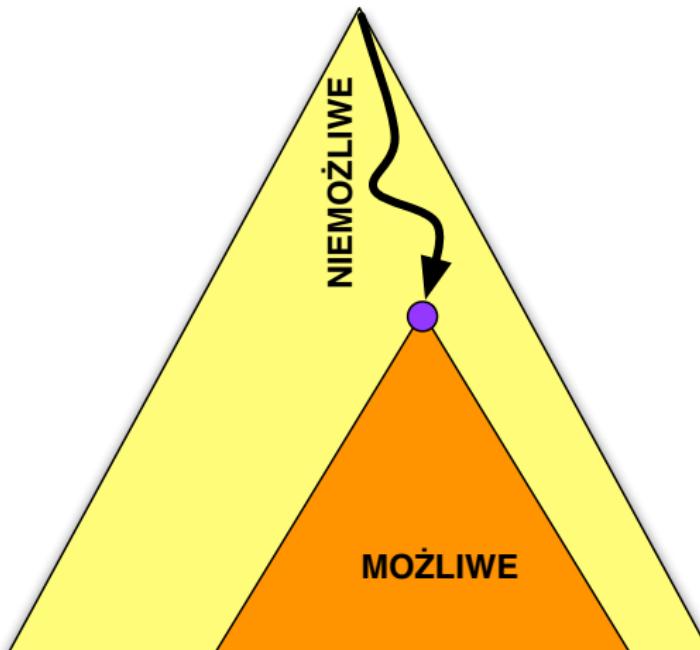
Przegląd drzewa poszukiwań

- MOŻLIWE – stara się przewidzieć czy w analizowanym poddrzewie jest jakieś rozwiązanie dopuszczalne;
- NIEMOŻLIWE – analizując dotychczas ustalone wartości $X[1], X[2], \dots, X[i]$ sprawdza czy nie naruszają one ograniczeń i uniemożliwiają znalezienie w rozpatrywanym poddrzewie rozwiązania dopuszczalnego.

Przeszukiwanie z nawrotami

Przegląd drzewa poszukiwań

Warunek MOŻLIWE "przewiduje" przyszłość, natomiast warunek NIEMOŻLIWE analizuje przeszłość:



Przeszukiwanie z nawrotami

Przykłady problemów: ciąg bitów

Poniższa procedura drukuje wszystkie n -elementowe ciągi zero-jedynkowe, zatem nie ma potrzeby sprawdzania warunku MOŻLIWE ani NIEMOŻLIWE.

```
1: procedure PODZBIÓR(n, i)                                ▷  $n$ -elementowe ciągi bitów
2:   if  $i = n + 1$  then
3:     drukuj wartości  $X[1], X[2], \dots, X[n]$ 
4:   else
5:      $X[i] \leftarrow 0$ ;                                     ▷ gdy nie biorę  $i$ -tego elementu
6:     PODZBIÓR( $n, i + 1$ );
7:      $X[i] \leftarrow 1$ ;                                     ▷ gdy biorę  $i$ -ty element
8:     PODZBIÓR( $n, i + 1$ )
9:   end if
10: end procedure
```

Przeszukiwanie z nawrotami

Przykłady problemów: ciąg bitów

Example

Po wywołaniu PODZBIÓR(3, 1) drukowane są następujące wartości:

000
001
010
011
100
101
110
111

Przeszukiwanie z nawrotami

Przykłady problemów: kombinacje bez powtórzeń

```

1: procedure KOMBINACJE( $n, k, i$ )                                ▷ wybór  $k$  spośród  $n$ 
2:   if  $i = k + 1$  then
3:     drukuj wartości  $X[1], X[2], \dots, X[k]$ 
4:   else
5:     if  $i = 1$  then                                     ▷ wyliczenie wartości początkowej dla  $X[i]$ 
6:        $start \leftarrow 1$ 
7:     else
8:        $start \leftarrow X[i - 1] + 1$ 
9:     end if
10:    for  $X[i] \leftarrow start, n$  do
11:      if  $k - i \leq n - X[i]$  then                      ▷ MOŻLIWE  $\equiv k - i \leq n - X[i]$ 
12:        KOMBINACJE( $n, k, i + 1$ )
13:      end if
14:    end for
15:  end if
16: end procedure

```

Przeszukiwanie z nawrotami

Przykłady problemów: kombinacje bez powtórzeń

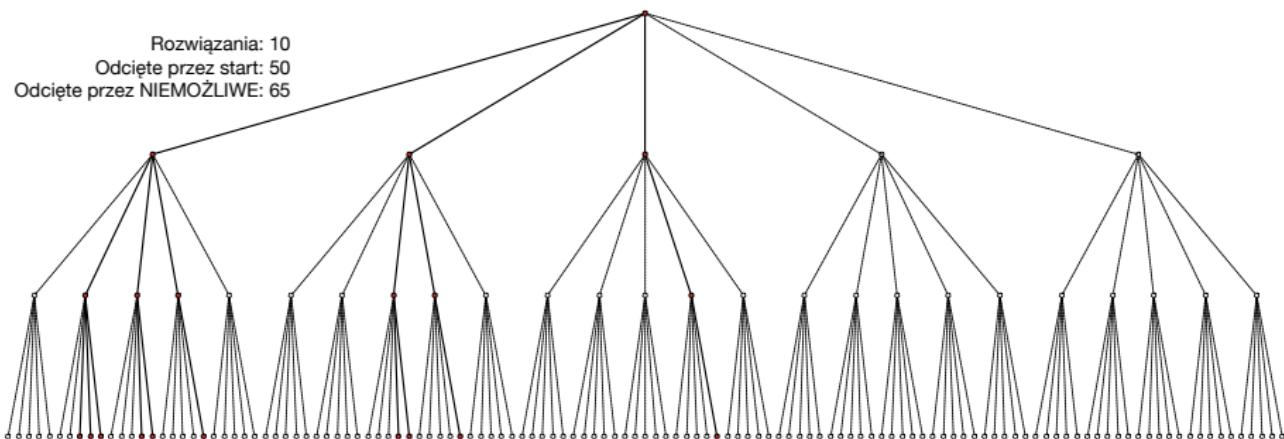
Example

Po wywołaniu KOMBINACJE(5, 3, 1) drukowane są następujące wartości:

```
1  2  3  
1  2  4  
1  2  5  
1  3  4  
1  3  5  
1  4  5  
2  3  4  
2  3  5  
2  4  5  
3  4  5
```

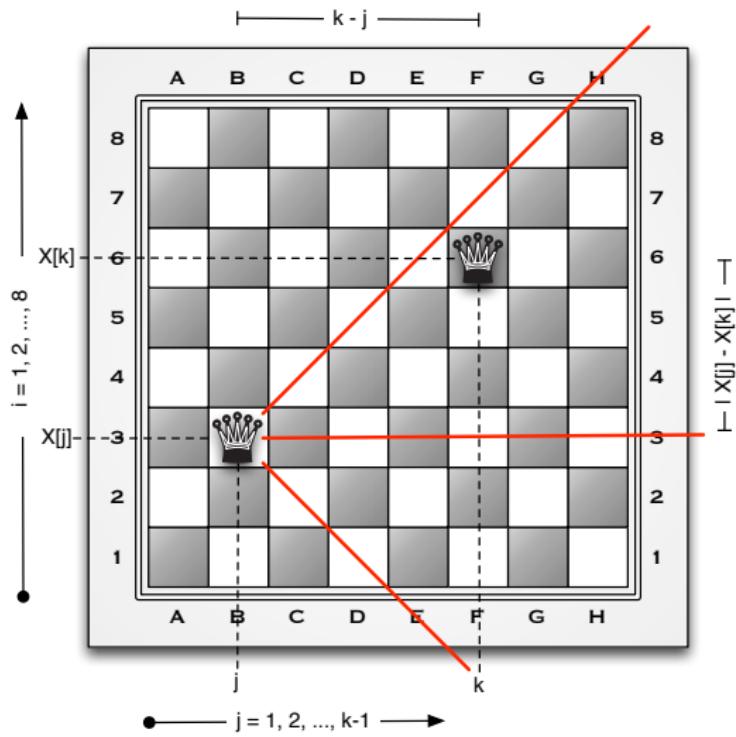
Przeszukiwanie z nawrotami

Przykłady problemów: kombinacje bez powtórzeń



Przeszukiwanie z nawrotami

Przykłady problemów: problem hetmanów



Przeszukiwanie z nawrotami

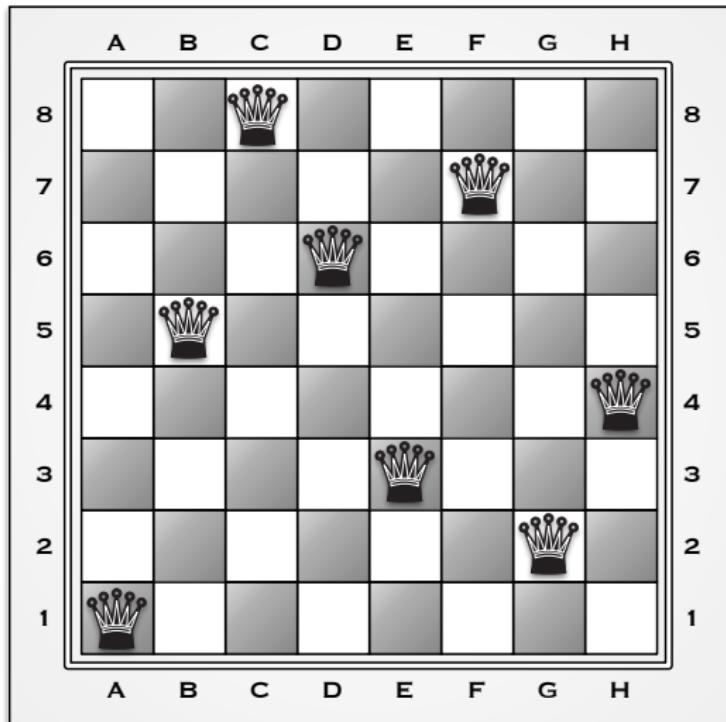
Przykłady problemów: problem hetmanów

```
1: procedure HETMANY(k)           ▷ dostawienie hetmanów w kolumnach k..8
2:   if k = 9 then
3:     drukuj rozwiązanie X[1], X[2], ..., X[8]
4:   else
5:     for X[k] ← 1, 8 do           ▷ wybierz kolejny wiersz
6:       OK ← true;
7:       for j ← 1, k - 1 do
8:         OK ← OK ∧ X[j] ≠ X[k] ∧ |X[j] - X[k]| ≠ k - j
9:       end for
10:      if OK then
11:        HETMANY(k + 1)
12:      end if
13:    end for
14:  end if
15: end procedure
```

Przeszukiwanie z nawrotami

Przykłady problemów: problem hetmanów

Pierwsze z 92 rozwiązań znalezionych procedurą HETMANY:



Przeszukiwanie z nawrotami

Przykłady problemów: problem hetmanów

Fragment kodu w wierszach 6-9 odpowiada sprawdzeniu warunku NIEMOŻLIWE. Jeśli po wyjściu z pętli **for-do** wartość zmiennej *OK* jest równa *false*, to znaczy, że hetman stawiany na przecięciu kolumny *k-tej* i wiersza *i-tego* jest bity przez któregoś z wcześniej postawionych hetmanów w kolumnach od 1 do *k – 1*.

Ćwiczenie

Przekształć pętlę w wierszach 7-9 procedury HETMANY aby kończyła się zaraz po wykryciu hetmana bijącego pozycję na przecięciu *k-tej* kolumny z *i-tym* wierszem (zmienna *OK* powinna przyjąć wtedy wartość *false*).

Przeszukiwanie z nawrotami

Wybrane narzędzia programistyczne

- Dla efektywnego rozwiązywania problemów sformułowanych w postaci problemu spełnienia ograniczeń opracowano specjalną metodologię programowania nazywaną *technologią więzów*³.
- Technologię więzów opracowano początkowo dla języka programowania Prolog⁴ ale dostępna jest ona dla innych języków programowania jak np. Java czy C++.

³Na studiach drugiego stopnia jest możliwość wyboru kursu **Technologia więzów**.

⁴Na studiach pierwszego stopnia istnieje możliwość wyboru kursu **Programowanie w logice**, na którym wykładany jest język programowania Prolog, natomiast na studiach drugiego stopnia jest możliwość wyboru kursu **Metody programowania w logice**, na którym omawiane są teoretyczne podstawy programowania w logice.

Przeszukiwanie z nawrotami

Wybrane narzędzia programistyczne

Tabela: Wybrane narzędzia dostarczające technologię więzów

Narzędzie	Język programowania
IBM ILOG Solver	C++
JaCoP	Java
B-Prolog	Prolog
Ciao Prolog	Prolog
ECLiPSe	Prolog
GNU-Prolog	Prolog
IF/Prolog	Prolog
SICStus Prolog	Prolog
SWI-Prolog	Prolog
Yap	Prolog
Emma	Python
python-constraint	Python

Przeszukiwanie z nawrotami

Wybrane narzędzia programistyczne

Example

Prolog jest językiem deklaratywnym, zatem opisuje się w nim problem bez podawania sposobu (algorytmu) rozwiązywania. Rozpatrzmy problem generowania wszystkich permutacji liczb od 1 do n . Poniższy predykat perm(N , X) opisuje w GNU-Prologu ten problem:

```
perm(N, X) :- length(X, N),
              fd_domain(X, 1, N),
              fd_all_different(X),
              fd_labeling(X).
```

Przeszukiwanie z nawrotami

Wybrane narzędzia programistyczne

Example (cd.)

Predykat $\text{perm}(N, X)$ wyraża warunki jakie powinny być spełnione aby X było permutacją liczb od 1 do N :

- ① X powinno być listą długości N .
- ② Elementy listy X powinny być liczbami z zakresu od 1 do N .
- ③ Elementy listy X powinny być parami różne.
- ④ Należy podstawić za elementy X takie wartości aby spełniały powyższe trzy warunki.

Przeszukiwanie z nawrotami

Wybrane narzędzia programistyczne

Example (cd.)

Dostępny w GNU-Prologu predykat `fd_labeling(X)` dokonuje przeglądu drzewa poszukiwań, przy czym jest on wykonywany w bardzo wyrafinowany sposób z uwzględnieniem warunku MOŻLIWE (na podstawie warunków 1, 2 i 3) oraz warunku NIEMOŻLIWE (na podstawie warunku 3). To jak system GNU-Prolog uwzględnia je podczas przeszukiwania stanowi kwintesencję technologii więzów. Przykład odpowiedzi udzielonych przez GNU-Prolog na pytanie `perm(3, X)`:

```
| ?- perm(3, X).  
X = [1,2,3] ? ;  
X = [1,3,2] ? ;  
X = [2,1,3] ? ;  
X = [2,3,1] ? ;  
X = [3,1,2] ? ;  
X = [3,2,1]
```

Wykład 9

Obliczenia na stosie

Obliczenia na stosie

- stos i operacje na stosie
- odwrotna notacja polska
- języki oparte na ONP
- przykłady programów

Obliczenia na stosie

Stos i operacje na stosie

Stos jest strukturą danych, na której dostępne są dwie operacje:

- wstawienie elementu,
- wyjęcie elementu.

Najważniejszą właściwością stosu jest to, że wyjmowany jest element jaki został ostatnio wstawiony. Z tego powodu nazywa się go kolejką **LIFO** od angielskiego **last in, first out**.

Obliczenia na stosie

Stos i operacje na stosie

- Niech $STACK[1..n]$ będzie globalną tablicą a SP globalną zmienną o wartościach całkowitych (początkowa wartość SP równa jest 1).
- Wartość zmiennej SP (ang. stack pointer) wskazuje aktualny szczyt stosu.
- Poniższa procedura wstawia wartość argumentu x na szczyt stosu:

```
1: procedure Push(x)
2:   STACK[SP] ← x;
3:   SP ← SP + 1
4: end procedure
```

Obliczenia na stosie

Stos i operacje na stosie

- Poniższa funkcja zdejmuje wartość ze szczytu stosu i zwraca ją:
 - 1: **function** Pop
 - 2: $SP \leftarrow SP - 1;$
 - 3: Pop $\leftarrow STACK[SP]$
 - 4: **end function**

Ćwiczenie

Rozszerz procedurę Push o obsługę przepełnienia stosu a funkcję Pop o reakcję na pusty stos.

Obliczenia na stosie

Odwrotna notacja polska: notacja beznawiasowa

- W roku 1920 polski matematyk Łukasiewicz opracował beznawiasową notację formuł rachunku zdań, którą nazwano **notacją polską**.
- W notacji polskiej operatory Negacji, Alternatywy i Koniunkcji poprzedzają ich argumenty.

Example

Formułę $p \wedge q$ zapisuje się jako Kpq , formułę $p \wedge q \vee r$ jako $AKpqr$ a formułę $p \wedge \neg q$ jako $KpNq$.

Obliczenia na stosie

Odwrotna notacja polska: notacja odwrotna

- Przy korzystaniu ze stosu wygodniej operatory pisać za argumentami.
- Beznawiasową notację, w której argumenty poprzedzają operatory, nazywa się **odwrotną notacją polską**.

Obliczenia na stosie

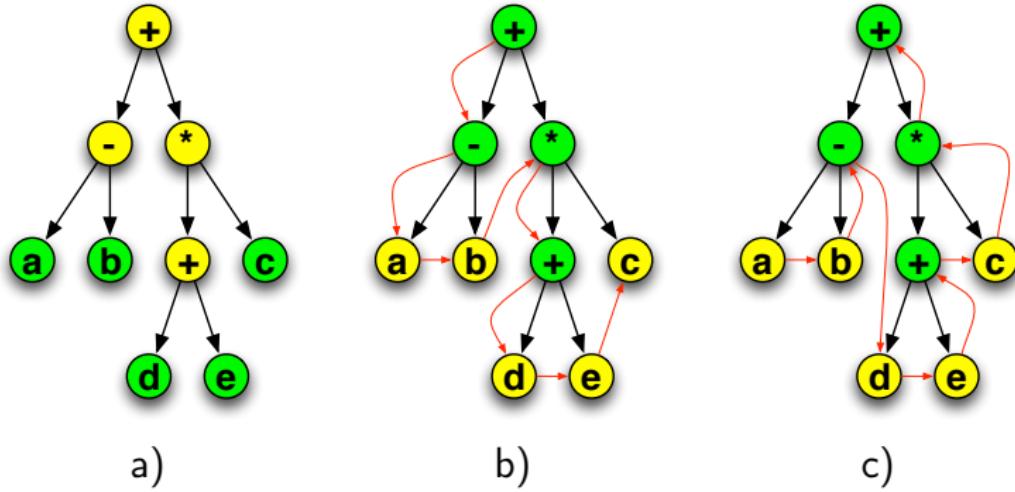
Odwrotna notacja polska: notacja odwrotna

Rozróżnia się trzy porządkи przeglądu drzewa:

- pre-order** najpierw korzeń a potem lewe i prawe poddrzewo w porządku **pre-order**;
- in-order** najpierw lewe poddrzewo w porządku **in-order**, potem korzeń a na koniec prawe poddrzewo w porządku **in-order**;
- post-order** najpierw lewe i prawe poddrzewo w porządku **post-order** a na końcu korzeń.

Obliczenia na stosie

Odwrotna notacja polska: notacja odwrotna



Rysunek: a) Drzewo wyrażenia $a - b + (d + e) * c$; b) jego przegląd w porządku **pre-order** $+ - ab * + dec$; c) jego przegląd w porządku **post-order** $ab - de + c * +$

Obliczenia na stosie

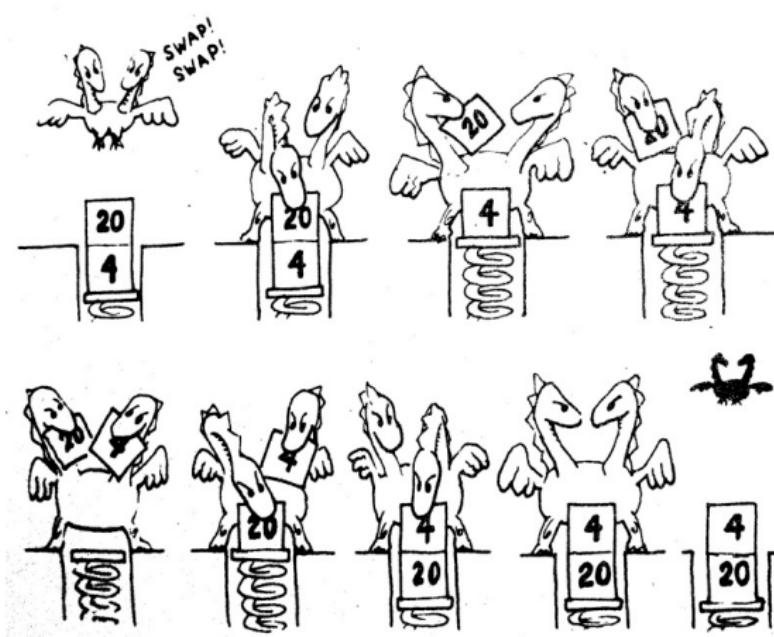
Odwrotna notacja polska: języki oparte na ONP

FORTH

n	(-- n)	wstawienie liczby na szczyt stosu
+	(n1 n2 -- (n1+n2))	suma dwóch liczb
-	(n1 n2 -- (n1-n2))	różnica dwóch liczb
*	(n1 n2 -- (n1*n2))	iloczyn dwóch liczb
/	(n1 n2 -- (n1 div n2))	iloraz dwóch liczb
.	(n --)	drukowanie wartości ze szczytu stosu
SWAP	(n1 n2 -- n2 n1)	zamiana dwóch elementów
DUP	(n -- n n)	skopiowanie elementu
OVER	(n1 n2 -- n1 n2 n1)	skopiowanie elementu
ROT	(n1 n2 n3 -- n2 n3 n1)	rotacja trzech elementów
DROP	(n --)	usunięcie elementu

Obliczenia na stosie

Odwrotna notacja polska: języki oparte na ONP



Ilustracja z książki L. Brodie. *Starting FORTH. An Introduction to the FORT Language and Operating System for Beginners and Professionals.* Prentice-Hall Inc., 1987.

Obliczenia na stosie

Odwrotna notacja polska: języki oparte na ONP

Example

Poniższy program w języku FORTH wstawia dwie liczby na stos, oblicza ich sumę a następnie drukuje wartość sumy (usuwając ją ze szczytu stosu):

2 2 + .

Obliczenia na stosie

Odwrotna notacja polska: języki oparte na ONP

Kolejne trzy przykłady pochodzą z książki Brodie'go.

Example

Obliczmy wartość $a \cdot (a + b)$ gdy na stosie znajdują się wartości a i b . W tym celu należy wykonać operacje:

OVER + *

Prześledźmy proces obliczania:

operacja	zawartość stosu
	a b
OVER	a b a
+	a (b+a)
*	(a*(b+a))

Obliczenia na stosie

Odwrotna notacja polska: języki oparte na ONP

Example

Obliczmy wartość $b \cdot (a - c)$ gdy na stosie znajdują się wartości c , b i a . Należy wykonać operacje:

ROT - *

Przebieg obliczeń:

operacja	zawartość stosu
	c b a
ROT	b a c
-	b (a-c)
*	(b*(a-c))

Obliczenia na stosie

Odwrotna notacja polska: języki oparte na ONP

Example

Operację OVER można zdefiniować jako następujący ciąg innych operacji:

SWAP DUP ROT SWAP

Prześledźmy proces obliczania:

operacja	zawartość stosu
	n1 n2
SWAP	n2 n1
DUP	n2 n1 n1
ROT	n1 n1 n2
SWAP	n1 n2 n1

Obliczenia na stosie

Odwrotna notacja polska: języki oparte na ONP

Example (Ciąg Fibonacciego)

Na szczytach stosu znajdują się liczby 0 i 1. Każdorazowe wykonanie sekwencji trzech operacji:

SWAP OVER +

wylicza na szczytach stosu kolejny wyraz ciągu Fibonacciego:

operacja	zawartość stosu	operacja	zawartość stosu
	0 1	SWAP	2 1
SWAP	1 0	OVER	2 1 2
OVER	1 0 1	+	2 3
+	1 1	SWAP	3 2
SWAP	1 1	OVER	3 2 3
OVER	1 1 1	+	3 5
+	1 2		

Obliczenia na stosie

Odwrotna notacja polska: języki oparte na ONP

Postscript

n	(-- n)	wstawienie liczby na szczyt stosu
add	(n1 n2 -- (n1+n2))	suma dwóch liczb
sub	(n1 n2 -- (n1-n2))	różnica dwóch liczb
mul	(n1 n2 -- (n1*n2))	iloczyn dwóch liczb
div	(n1 n2 -- (n1/n2))	iloraz dwóch liczb
=	(n --)	drukowanie wartości ze szczytu stosu
dup	(n -- n n)	skopiowanie elementu
exch	(n1 n2 -- n2 n1)	zamiana dwóch elementów
pop	(n --)	usunięcie elementu
roll	(a1 a2 ... an n i -- a(n-i+1) ... an a1 ... a(n-i))	rotacja n elementów o i pozycji

Obliczenia na stosie

Odwrotna notacja polska: języki oparte na ONP

Example (Schemat Hornera)

Założmy, że na stosie znajdują się kolejno wartości współczynników a_0, a_1, \dots, a_n wielomianu n -tego stopnia a na szczycie stosu jest wartość parametru x . Aby policzyć wartość wielomianu

$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

należy n razy wykonać ciąg Postscriptowych operacji:

```
dup 4 1 roll mul add exch
```

Po ich wykonaniu na szczycie stosu znajdzie się parametr x a pod nim wyliczona wartość wielomianu.

Obliczenia na stosie

Odwrotna notacja polska: języki oparte na ONP

Example (Schemat Hornera cd.)

Poniższy program w Postscriptie wylicza wartość wielomianu

$1 + 2x + 3x^2 + 4x^3$ dla $x = 10$ a następnie drukuje wyliczoną wartość:

```
1 2 3 4 10
dup 4 1 roll mul add exch
dup 4 1 roll mul add exch
dup 4 1 roll mul add exch
pop =
```

W wyniku wykonania powyższego programu zostaje wydrukowana wartość 4321.

Obliczenia na stosie

Odwrotna notacja polska: przykład programu

- Wartości logiczne będziemy wyrażać liczbami 0 i 1 odpowiednio dla fałszu i prawdy.
- Wartość negacji wartości logicznej x można wyliczyć ze wzoru $1 - x$.
- Niech x i y będą wartościami logicznymi. Wówczas wartość alternatywy wyliczyć można ze wzoru $\max(x, y)$ a wartość koniunkcji ze wzoru $\min(x, y)$.

Obliczenia na stosie

Odwrotna notacja polska: przykład programu

```
1: function Wartość(wyrażenie, n, wartości)
2:   for i  $\leftarrow$  1, n do
3:     if wyrażenie[i] = 'N' then
4:       Push(1 - Pop)
5:     else
6:       if wyrażenie[i] = 'A' then
7:         Push(max(Pop, Pop))
8:       else
9:         if wyrażenie[i] = 'K' then
10:          Push(min(Pop, Pop))
11:        else
12:          if wyrażenie[i] = 'p' then
13:            Push(wartości[1])
14:          else
15:            ...
16:          end if
17:        end if
18:      end if
19:    end if
20:  end for
21:  Wartość  $\leftarrow$  Pop
22: end function
```

Obliczenia na stosie

Odwrotna notacja polska: przykład programu

Poniższy program służy do wydrukowania tabeli zero-jedynkowej dla formuły $p \wedge (q \vee r)$:

```
1: wyrażenie ← "pqrAK"
2: for i ← 0, 1 do
3:   wartości[1] ← i;                                ▷ ustalenie wartości zmiennej p
4:   for j ← 0, 1 do
5:     wartości[2] ← j;                                ▷ ustalenie wartości zmiennej q
6:     for k ← 0, 1 do
7:       wartości[3] ← k;                                ▷ ustalenie wartości zmiennej r
8:       write i, j, k, Wartość(wyrażenie, n, wartości)
9:     end for
10:    end for
11:  end for
```

Obliczenia na stosie

Odwrotna notacja polska: przykład programu

W wyniku wykonania tego programu zostaną wydrukowane następujące liczby:

0 0 0 0
0 0 1 0
0 1 0 0
0 1 1 0
1 0 0 0
1 0 1 1
1 1 0 1
1 1 1 1

Jak widać formuła jest spełniona przez trzy z ośmiu możliwych wartościowań (waluacjii).

Obliczenia na stosie

Zamiana rekurencji na iterację

- Nie wszystkie języki programowania umożliwiają rekurencję (np. język Basic, wczesne wersje języka Fortran, język assemblera).
- W sytuacji kiedy nie ma dostępnej rekurencji, można samemu utworzyć stos z użyciem tablicy.
- Na takim własnym stosie można odkładać wartości jakie powinny zachować się przy powtórzeniu obliczeń (odpowiednik zachowania wartości zmiennych lokalnych w funkcjach rekurencyjnych).
- Pokażemy to na przykładzie funkcji obliczającej n -ty wyraz ciągu Fibonacciego z rekurencyjnego wzoru:

$$Fib(n) = \begin{cases} 0 & \text{gdy } n = 0, \\ 1 & \text{gdy } n = 1, \\ Fib(n - 1) + Fib(n - 2) & \text{w p.p.} \end{cases}$$

Obliczenia na stosie

Zamiana rekurencji na iterację

```
1: function Fib(n)
2:     acc ← 0;
3:     PUSH(n);
4:     while stos nie jest pusty do
5:         n ← POP;
6:         if n = 1 then
7:             acc ← acc + 1
8:         else
9:             if n > 1 then
10:                PUSH(n - 1);
11:                PUSH(n - 2)
12:            end if
13:        end if
14:    end while
15:    Fib ← acc
16: end function
```

Obliczenia na stosie

Zamiana rekurencji na iterację

Ćwiczenie

Udowodnij, że funkcja $Fib(n)$, wykorzystująca stos, liczy poprawnie n -ty wyraz ciągu Fibonacciego, dla dowolnego całkowitego $n \geq 0$.

Ćwiczenie

Czy wykonując poprzednie ćwiczenie, udowodniłeś, że obliczenia funkcji $Fib(n)$ zawsze się kończą? Jeśli nie, to udowodnij to.

Wykład 10

Programowanie dynamiczne

Programowanie dynamiczne

- Złożoność obliczeniowa funkcji rekurencyjnych
- programowanie dynamiczne
- przykłady programów

Programowanie dynamiczne

Złożoność obliczeniowa funkcji rekurencyjnych

Często funkcję rekurencyjną $f(n)$ zapisać można w postaci wyrażenia:

$$g(f(n - i_1), f(n - i_2), \dots, f(n - i_k)),$$

które należy rozumieć jako wyrażenie, w którym występuje k innych wartości funkcji f .

Example

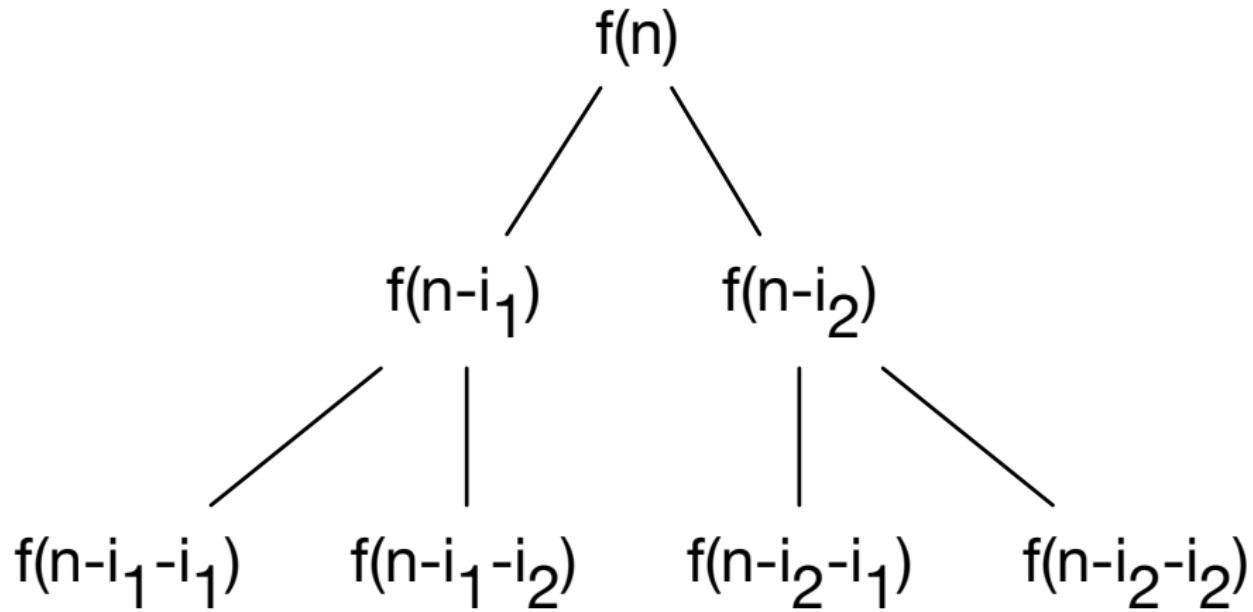
silnia ($k = 1$) $g(f(n - 1)) = n \cdot f(n - 1)$

ciąg Fibonacciego ($k = 2$) $g(f(n - 1), f(n - 2)) = f(n - 1) + f(n - 2)$

Programowanie dynamiczne

Złożoność obliczeniowa funkcji rekurencyjnych

Gdy $k \geq 2$, liczba obliczanych wyrażeń rośnie wykładniczo.



Programowanie dynamiczne

Złożoność obliczeniowa funkcji rekurencyjnych: ciąg Fibonacciego

Niech s_n oznacza liczbę operacji dodawania wykonywanych podczas obliczania n -tego wyrazu ciągu Fibonacciego zadanego wzorem rekurencyjnym:

$$Fib(n) = \begin{cases} 0 & \text{gdy } n = 0, \\ 1 & \text{gdy } n = 1, \\ Fib(n - 1) + Fib(n - 2) & \text{gdy } n > 1 \end{cases}$$

Wartość s_n można wyliczyć, z poniższych zależności:

$$s_0 = 0, \quad s_1 = 0, \quad s_n = 1 + s_{n-1} + s_{n-2} \text{ dla } n > 1$$

Programowanie dynamiczne

Złożoność obliczeniowa funkcji rekurencyjnych: ciąg Fibonacciego

W poniższej tabeli zebrano pierwsze wyrazy ciągu $Fib(n)$ i s_n :

n	0	1	2	3	4	5	6	7
$Fib(n)$	0	1	1	2	3	5	8	13
s_n	0	0	1	2	4	7	12	20

Łatwo zauważyć w niej następujące dwie prawidłowości:

$$\begin{aligned} s_n &= \sum_{i=0}^{n-1} Fib(i) \\ s_n &= Fib(n+1) - 1 \end{aligned}$$

Ćwiczenie

Udowodnij indukcyjnie oba powyższe równania.

Programowanie dynamiczne

Złożoność obliczeniowa funkcji rekurencyjnych: ciąg Fibonacciego

Stosując funkcje tworzące – wykraczające poza tematykę naszego wykładu – można wyprowadzić wzór analityczny na n -ty wyraz ciągu Fibonacciego:

$$Fib(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}} = \left\lfloor \frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

gdzie $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ i $\psi = \frac{1-\sqrt{5}}{2} \approx -0.618$.

Ćwiczenie

Udowodnij poprawność powyższej postaci analitycznej.

Granica $\lim_{n \rightarrow \infty} \frac{Fib(n)}{\varphi^n} = \frac{1}{\sqrt{5}}$, zatem

$$Fib(n) = \Theta(\varphi^n).$$

Programowanie dynamiczne

Złożoność obliczeniowa funkcji rekurencyjnych: ciąg Fibonacciego

- Z tego, że $Fib(n + 2) = \Theta(\varphi^n)$, można wywnioskować, że liczba s_n wykonywanych operacji dodawania podczas obliczania $Fib(n)$ jest rzędu $\Theta(\varphi^n)$, zatem rośnie wykładniczo wraz ze wzrostem n .
- Powrócimy jeszcze do ciągu Fibonacciego pokazując jak policzyć jego wartości wykonując $\Theta(n)$ operacji dodawania.
- Dzięki programowaniu dynamicznemu, można w prosty sposób przeformułować funkcję rekurencyjną tak aby była efektywniej obliczana (np. w czasie wielomianowym albo nawet liniowym) z użyciem iteracji i tablic do przechowywania pośrednich wyników.

Programowanie dynamiczne

Idea

- Aby uniknąć wielokrotnego obliczania wartości tych samych wyrażeń tablicuje się je.
- Programowanie dynamiczne nazywamy *d-wymiarowym*, jeśli rekurencyjna funkcja $f : N^d \mapsto R$ jest *d-argumentowa*.
- Wyniki pośrednie w *d-wymiarowym programowaniu dynamicznym* tablicuje się w *d-wymiarowej* tablicy $t[n_1, n_2, \dots, n_d]$, gdzie n_1, n_2, \dots, n_d są rozmiarami tablicy w kolejnych wymiarach.
- Cała trudność programowania dynamicznego tkwi w dobraniu odpowiedniej kolejności wyliczania elementów tablicy t .

Programowanie dynamiczne

Idea

Example ($d = 1$)

W tym przypadku najczęściej wystarcza obliczanie wartości $t[i]$ dla kolejnych wartości $i = 1, 2, 3, \dots$

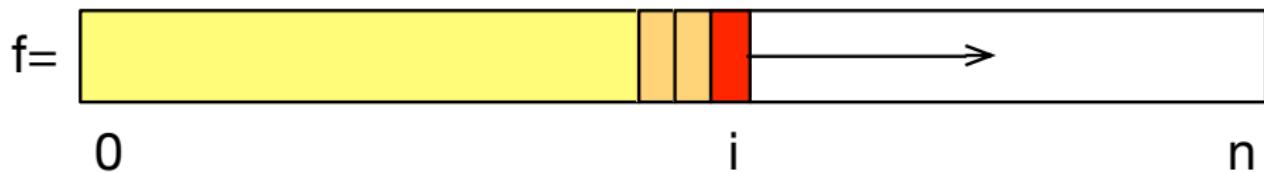
Example ($d = 2$)

W tym przypadku najczęściej oblicza się wartości $t[i, j]$ kolejnymi wierszami albo kolumnami. Czasami trzeba jednak bardziej wyrafinowanej kolejności jak np. kolejnymi coraz dłuższymi przekątnymi:

```
1: for  $k \leftarrow 1, n$  do                                ▷ przetwarzamy  $k$ -tą przekątną  
2:   for  $i \leftarrow 1, k$  do                      ▷ przetwarzamy  $i$ -ty element  $k$ -tej przekątnej  
3:      $j \leftarrow k + 1 - i$ ;    ▷ na  $k$ -tej przekątnej zachodzi warunek  $i + j = k + 1$   
4:     oblicz  $t[i, j]$   
5:   end for  
6: end for
```

Programowanie dynamiczne

Przykłady programów: ciąg Fibonacciego

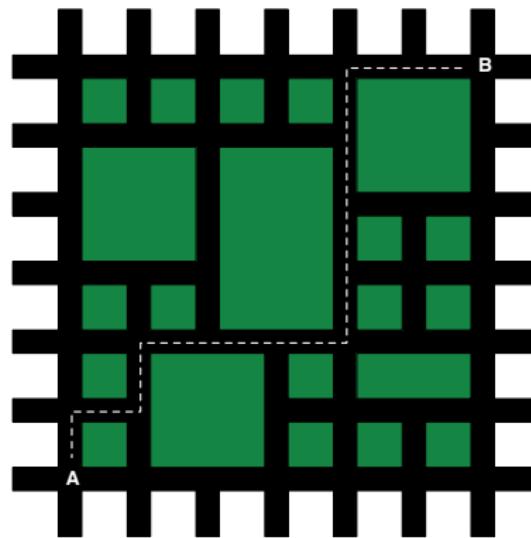


```
1: function Fib(n)
2:   f[0] ← 0;
3:   f[1] ← 1;
4:   for i ← 2, n do
5:     f[i] ← f[i - 1] + f[i - 2]
6:   end for
7:   Fib ← f[n]
8: end function
```

Programowanie dynamiczne

Przykłady programów: zliczanie dróg

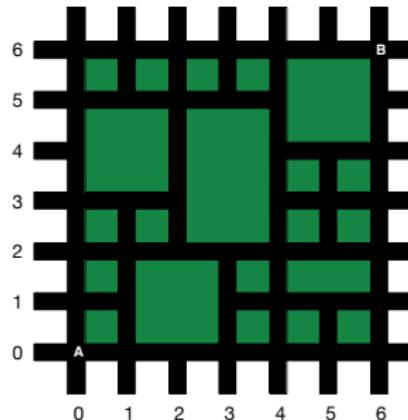
Rozpatrzmy problem zliczania najkrótszych dróg między zadanymi dwoma skrzyżowaniami A i B:



Programowanie dynamiczne

Przykłady programów: zliczanie dróg

- Ponumerujmy ulice od lewej do prawej kolejnymi liczbami naturalnymi $0, 1, 2, \dots$
- Podobnie ponumerujmy ulice od dołu do góry kolejnymi liczbami naturalnymi $0, 1, 2, \dots$
- Przy takim ponumerowaniu skrzyżowanie A ma współrzędne $(0, 0)$ a skrzyżowanie B współrzędne $(6, 6)$.



Programowanie dynamiczne

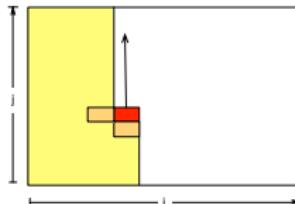
Przykłady programów: zliczanie dróg

- Symbolem $N[i, j]$ oznaczmy liczbę najkrótszych dróg od skrzyżowania A do skrzyżowania o współrzędnych (i, j) , tj. przecięcie i -tej ulicy pionowej z j -tą ulicą poziomą.
- Przyjmujemy, że $N[i, j] = 0$, gdy skrzyżowanie (i, j) jest zajęte przez budynek.
- Pozostałe wartość $N[i, j]$ można wyliczyć ze wzoru:

$$N[i, j] = \begin{cases} 1 & \text{gdy } i = 0, j = 0, \\ N[i - 1, j] + N[i, j - 1] & \text{gdy połączone z } (i - 1, j) \text{ oraz } (i, j - 1), \\ N[i - 1, j] & \text{gdy połączone z } (i - 1, j) \text{ ale nie z } (i, j - 1), \\ N[i, j - 1] & \text{gdy połączone z } (i, j - 1) \text{ ale nie z } (i - 1, j). \end{cases}$$

Programowanie dynamiczne

Przykłady programów: zliczanie dróg



```

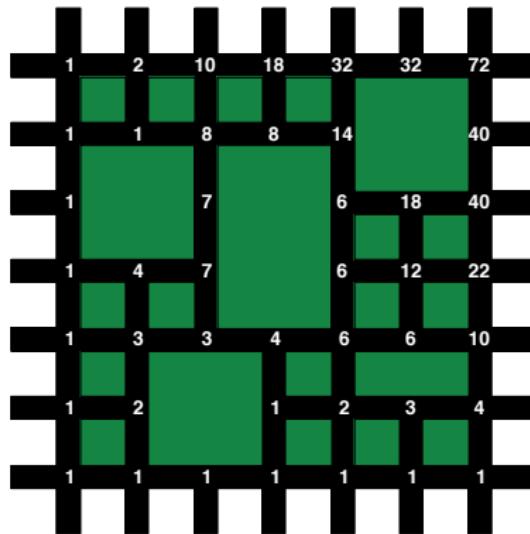
1: for  $i \leftarrow 0, m$  do
2:   for  $j \leftarrow 0, n$  do
3:     if  $(i, j)$  jest zajęte przez budynek then
4:        $N[i, j] \leftarrow 0$ 
5:     else
6:       if  $i = 0 \wedge j = 0$  then
7:          $N[i, j] \leftarrow 1$ 
8:       else
9:          $N[i, j] \leftarrow N[i - 1, j]$  gdy połączone z  $(i - 1, j)$  +  $N[i, j - 1]$  gdy połączone z  $(i, j - 1)$ 
10:        end if
11:      end if
12:    end for
13:  end for

```

Programowanie dynamiczne

Przykłady programów: zliczanie dróg

Na poniższym rysunku zaznaczono dla każdego skrzyżowania (i, j) liczbę $N[i, j]$ najkrótszych dróg prowadzących do niego ze skrzyżowania $A(0, 0)$:



Programowanie dynamiczne

Przykłady programów: zliczanie dróg

- Liczba najkrótszych dróg rośnie wykładniczo szybko wraz ze wzrostem i oraz j .
- Liczbę najkrótszych dróg można policzyć bez potrzeby ich generowania.
- Dzięki wzorowi rekurencyjnemu i programowaniu dynamicznemu możemy policzyć ile jest najkrótszych dróg w czasie rzędu $O(i \cdot j)$.
- Zauważ, że gdyby nie było budynków zajmujących część połączeń wówczas $N[i, j] = \binom{i+j}{i}$.
- Kilka pierwszych wartości $N[n, n] = \binom{2n}{n}$:

n	1	2	3	4	5	6	7	8
$N[n, n]$	2	6	20	70	252	924	3432	12870

Programowanie dynamiczne

Przykłady programów: problem plecakowy

- Danych jest n rodzajów przedmiotów, przy czym zakładamy, że jest dowolnie dużo przedmiotów każdego z tych rodzajów.
- Każdy przedmiot i -tego rodzaju ma rozmiar $\text{size}[i]$ i wartość $\text{value}[i]$.
- Należy zapakować plecak o pojemności k (suma rozmiarów nie może przekroczyć pojemności), tak aby łączna wartość spakowanych przedmiotów była jak największa.

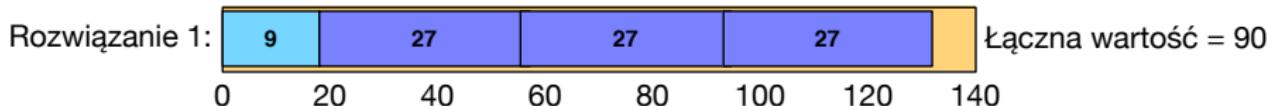
Programowanie dynamiczne

Przykłady programów: problem plecakowy

Example

$n = 3$, $size[1] = 19$, $size[2] = 29$, $size[3] = 39$, $value[1] = 9$, $value[2] = 21$, $value[3] = 27$

Rodzaje: 



Na powyższym rysunku przedstawiono przykładowe dwa rozwiązania dla pojemności $k = 140$. Drugie rozwiązanie jest optymalne.

Programowanie dynamiczne

Przykłady programów: problem plecakowy

- Niech funkcja $\text{maxknapsack}(k)$ będzie równa maksymalnej łącznej wartości przedmiotów jakie można spakować do plecaka o pojemności k .
- Założmy, że do plecaka o pojemności k zdecydujemy się włożyć przedmiot i -tego rodzaju.
- Jeśli przedmiot i -tego rodzaju mieści się w plecaku ($\text{size}[i] \leq k$), to po włożeniu go do plecaka możemy osiągnąć łączną wartość:

$$W_i = \text{maxknapsack}(k - \text{size}[i]) + \text{value}[i].$$

- Wybór rodzaju wkładanego przedmiotu dokonamy wybierając największe W_i , dla $i = 1, 2, \dots, n$.

Programowanie dynamiczne

Przykłady programów: problem plecakowy

```
1: function MaxKnapsack(k, size, value, n)
2:   knapsack[0]  $\leftarrow$  0;
3:   for i  $\leftarrow$  1, k do       $\triangleright$  rozpatrywanie plecaków o coraz większej pojemności
4:     max  $\leftarrow$  0;
5:     for t  $\leftarrow$  1, n do           $\triangleright$  rozpatrywanie przedmiotu każdego rodzaju
6:       if size[t]  $\leq$  i then            $\triangleright$  przedmiot mieści się
7:         total  $\leftarrow$  knapsack[i - size[t]] + value[t];       $\triangleright$  łączna wartość
8:         if total > max then
9:           max  $\leftarrow$  total
10:        end if
11:      end if
12:    end for
13:    knapsack[i]  $\leftarrow$  max            $\triangleright$  największa wartość przy pojemności i
14:  end for
15:  MaxKnapsack  $\leftarrow$  knapsack[k]
16: end function
```

Programowanie dynamiczne

Przykłady programów: najdłuższy wspólny podciąg

- Dane są dwa napisy: pierwszy długości m znaków (a_1, a_2, \dots, a_m) i drugi długości n znaków (b_1, b_2, \dots, b_n).
- Ciąg znaków (c_1, c_2, \dots, c_k) nazywamy wspólnym podciągiem a i b jeśli

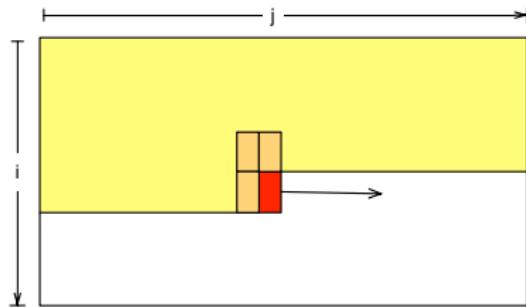
$$\exists_{1 \leq i_1 < i_2 < \dots < i_k \leq m} \forall_{l=1,2,\dots,k} c_l = a_{i_l}$$

$$\exists_{1 \leq j_1 < j_2 < \dots < j_k \leq n} \forall_{l=1,2,\dots,k} c_l = b_{j_l}$$

- Interesuje nas znalezienie jak najdłuższego wspólnego podcięgu.
- W tablicy $T[i, j]$ będziemy wyliczać długość najdłuższego wspólnego podcięgu dla fragmentów (a_1, a_2, \dots, a_i) oraz (b_1, b_2, \dots, b_j) .
- Jeśli $a_i = b_j$, to $T[i, j] = T[i - 1, j - 1] + 1$.
- Jeśli $a_i \neq b_j$, to $T[i, j] = \max\{T[i, j - 1], T[i - 1, j]\}$.

Programowanie dynamiczne

Przykłady programów: najdłuższy wspólny podciąg



```
1: for  $i \leftarrow 1, m$  do
2:   for  $j \leftarrow 1, n$  do
3:     if  $a_i = b_j$  then
4:        $T[i, j] \leftarrow T[i - 1, j - 1] + 1$ 
5:     else
6:        $T[i, j] \leftarrow \max\{T[i, j - 1], T[i - 1, j]\}$ 
7:     end if
8:   end for
9: end for
```

Programowanie dynamiczne

Przykłady programów: najdłuższy wspólny podciąg

Example (Tajemnica programowania)

	p r o g r a m o w a n i a													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	1	1	1	1	1	1	1	1
j	0	0	0	0	0	0	1	1	1	1	1	1	1	1
e	0	0	0	0	0	0	1	1	1	1	1	1	1	1
m	0	0	0	0	0	0	1	2	2	2	2	2	2	2
n	0	0	0	0	0	0	1	2	2	2	2	3	3	3
i	0	0	0	0	0	0	1	2	2	2	2	3	4	4
c	0	0	0	0	0	0	1	2	2	2	2	3	4	4
a	0	0	0	0	0	0	1	2	2	2	3	3	4	5

Wykład 11

Systemy regułowe

Systemy regułowe

- definicja systemu regułowego
- przykłady zastosowań
- implementacja systemów regułowych

Systemy regułowe

Definicja

- Alfabet Σ jest skończonym i niepustym zbiorem symboli.
- Słowo nad alfabetem Σ to skończony ciąg symboli.
- Słowo puste oznaczać będziemy grecką literą ε .
- Słowa długości k tworzą składając się na zbiór Σ^k , przy czym $\Sigma^0 = \{\varepsilon\}$.
- Domknięcie Kleene'go Σ^* to zbiór wszystkich skończonych słów:

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k.$$

Example

Niech $\Sigma = \{0, 1\}$. Wówczas:

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \dots\}$$

Systemy regułowe

Definicja

- Konkatenacją słów $a = a_1 a_2 \dots a_m \in \Sigma^m$ i $b = b_1 b_2 \dots b_n \in \Sigma^n$ jest słowo

$$ab = a_1 a_2 \dots a_m b_1 b_2 \dots b_n \in \Sigma^{m+n}.$$

- Symbolem w^n będziemy oznaczać n -krotną konkatenację słowa w :

$$w^n = \begin{cases} \varepsilon & \text{gdy } n = 0, \\ ww^{n-1} & \text{gdy } n > 0. \end{cases}$$

Example

$$(ba)^2 = baba$$

$$ab^2a = abba$$

Systemy regułowe

Definicja

- Regułą nazywać będziemy uporządkowaną parę dwóch słów nad alfabetem Σ .
- Regułę $\langle a, b \rangle \in \Sigma^* \times \Sigma^*$ będziemy zapisywać następująco:

$$a \rightarrow b$$

- Jeśli słowo $w = uav$ można przedstawić w postaci konkatenacji trzech słów $u, a, v \in \Sigma^*$, to wynikiem zastosowania reguły $a \rightarrow b$ do słowa w jest słowo ubv .

Systemy regułowe

Definicja

Definition (System regułowy, obliczenie, wynik)

- Systemem regułowym S nazywać będziemy uporządkowaną parę $S = \langle \Sigma, P \rangle$, gdzie P jest skończonym zbiorem reguł $P = \{P_1, P_2, \dots, P_k\} \subset \Sigma^* \times \Sigma^*$.
- Jeśli dla słowa w istnieje reguła $P_i \in P$, tż. stosując regułę P_i otrzymujemy słowo w' , to będziemy pisać $w \Rightarrow w'$.
- Jeśli dla słów w i u istnieje skończony ciąg słów w_0, w_1, \dots, w_n , tż. $w = w_0$, $w_i \Rightarrow w_{i+1}$ oraz $u = w_n$, to będziemy pisać $w \xrightarrow{*} u$.
- Jeśli $w \xrightarrow{*} u$ oraz żadnej reguły nie można już zastosować do słowa u , to ciąg przekształceń słowa w w słowo u nazywać będziemy obliczeniem a słowo u wynikiem obliczenia.

Systemy regułowe

Definicja

Example (Podwajanie)

Niech $\Sigma = \{0, 1\}$. Rozpatrzmy następujący system regułowy złożony z jednej reguły:

$$S = \langle \Sigma, \{01 \rightarrow 100\} \rangle.$$

W powyższym systemie zachodzi:

$$0^k 1 \xrightarrow{*} 10^{2k}$$

Przykładowe obliczenie:

$$\underline{0001} \Rightarrow \underline{00100} \Rightarrow \underline{010000} \Rightarrow 1000000$$

Systemy regułowe

Definicja

Example (Potęgowanie)

W systemie $\langle \{0, 1\}, \{01 \rightarrow 100\} \rangle$ zachodzi:

$$0^k 1^n \xrightarrow{*} 1^n 0^{k2^n}$$

Przykład obliczeń dla $k = 1, n = 4$:

$$01111 \Rightarrow 100111 \Rightarrow 1010011 \Rightarrow 11000011 \Rightarrow 110001001 \Rightarrow 1100100001 \Rightarrow$$

$$\Rightarrow 11010000001 \Rightarrow 111000000001 \Rightarrow 1110000000100 \Rightarrow$$

$$\Rightarrow 11100000010000 \Rightarrow 111000001000000 \Rightarrow 1110000100000000 \Rightarrow$$

$$\Rightarrow 111000100000000000 \Rightarrow 1110010000000000000 \Rightarrow$$

$$\Rightarrow 1110100000000000000 \Rightarrow 1111000000000000000000000000$$

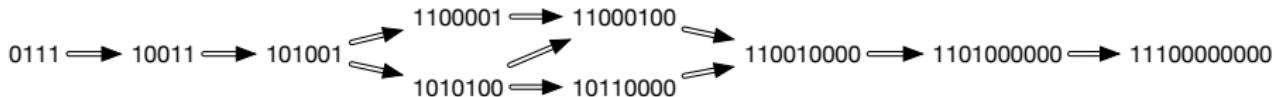
Systemy regułowe

Definicja

Example

Zwróć uwagę, że w systemie $\langle \{0, 1\}, \{01 \rightarrow 100\} \rangle$ obliczenia rozpoczęjące się tym samym słowem mogą przebiegać na więcej niż jeden sposób.

Na przykład, rozpoczynając od słowa 0111:



Systemy regułowe

Definicja

Example

Nie każdy system, ma tę własność, że każde obliczenie zaczynające się tym samym słowie daje jako wynik to samo słowo.

Dla przykładu, w systemie $\langle \{a, b, c\}, \{a \rightarrow b, a \rightarrow c\} \rangle$ obliczenia rozpoczętujące się słowem a mogą zakończyć się albo słowem b albo słowem c .

Systemy regułowe

Definicja

Definition (System konfluentny)

Mówimy, że system jest konfluentny jeśli dla dowolnego początkowego słowa w , każde możliwe obliczenie dostarcza ten sam wynik.

Ćwiczenie

Pokaż, że system $\langle \{0, 1\}, \{01 \rightarrow 100\} \rangle$ jest konfluentny.

Wskazówka 1: pomyśl o pewnym porządku leksykograficznym.

Wskazówka 2: niech w będzie początkowym słowem a n liczbą jedynek w słowie w . Gdy $n > 0$, niech a_0 będzie liczbą zer przed pierwszą jedynką; a_i , dla $i = 1, 2, \dots, n - 1$, liczbą zer między i -tą a $i + 1$ -szą jedynką; a_n liczbą zer za n -tą jedynką. Pokaż, że za każdym razem wynikiem dla słowa w jest słowo:

$$1^n 0^{\sum_{i=0}^n a_{n-i} 2^i}$$

Systemy regułowe

Przykłady

Example (Język okrągłych nawiasów)

Niech $\Sigma = \{(,)\}$. Wśród słów nad alfabetem Σ można wyróżnić słowa, w których są poprawnie sparowane nawiasy. Przykłady takich słów:

$\varepsilon, (), ()(), ((())), ()()(), ()((())), ((()), ((())()$

Systemy regułowe

Przykłady

Example (Język okrągłych nawiasów cd.)

```
1: niech  $n$  będzie długością słowa  $w$ ;  
2: licznik  $\leftarrow 0$ ;  
3: for  $i \leftarrow 1, n$  do  
4:   if  $w[i] = "("$  then  
5:     licznik  $\leftarrow$  licznik + 1  
6:   else  
7:     if licznik > 0 then  
8:       licznik  $\leftarrow$  licznik - 1  
9:     else  
10:      nawiasy niepoprawnie sparowane  
11:    end if  
12:  end if  
13: end for  
14: if licznik = 0 then  
15:   poprawnie sparowane nawiasy  
16: else  
17:   nawiasy niepoprawnie sparowane  
18: end if
```

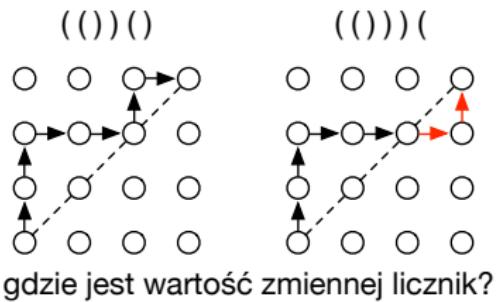
Systemy regułowe

Przykłady

Example (Język okrągłych nawiasów cd.)

Możliwych jest wiele interpretacji słów z poprawnie sparowanymi nawiasami. Jedna z nich jest następująca:

Rozpocznij wędrówkę w punkcie $\langle 0, 0 \rangle$. Ilekroć natrafisz podczas analizy słowa nawias "(" przesuń się w górę o wektor $(0, +1)$, natomiast przy nawiasie ")" przesuń się w prawo o wektor $(+1, 0)$. Nawiasy są poprawnie sparowane jeśli dojdziesz do punktu $\langle n + 1, n + 1 \rangle$ i ani razu nie zejdziesz poniżej przekątnej $y = x$.



Systemy regułowe

Przykłady

Example (Język okrągłych nawiasów cd.)

W słowie w są poprawnie sparowane nawiasy, wtedy i tylko wtedy gdy w systemie $\langle \Sigma, \{() \rightarrow \varepsilon\} \rangle$ wynikiem dla słowa w jest słowo ε . Przykłady:

$$(\underline{())})() \Rightarrow \underline{())}() \Rightarrow \underline{)} \Rightarrow \varepsilon$$

$$(\underline{))))(\Rightarrow \underline{)}) (\Rightarrow) ($$

Ćwiczenie

Udowodnij, że system $\langle \{((),)\}, \{() \rightarrow \varepsilon\} \rangle$ jest konfluentny.

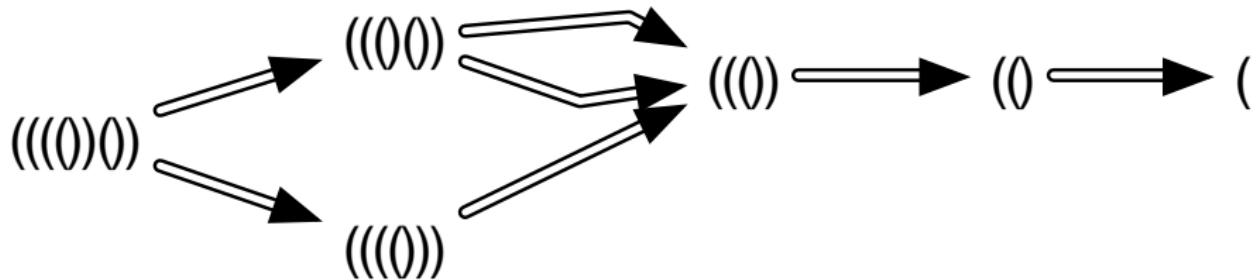
Wskazówka: przyjrzyj się diagramom z poprzedniego slajdu i pomyśl jak stosowanie reguły $() \rightarrow \varepsilon$ wpływa na przebieg ścieżki na diagramie.

Systemy regułowe

Przykłady

Example

Wszystkie możliwe obliczenia dla słowa (((())()):



Systemy regułowe

Przykłady

Example (Język dowolnych nawiasów)

Niech $\Sigma = \{(,), [], \langle \rangle\}$. Wśród słów nad alfabetem Σ można wyróżnić słowa, w których są poprawnie sparowane nawiasy. Przykłady takich słów:

([])⟨⟩, ([]⟨⟩), ([⟨⟩])

Systemy regułowe

Przykłady

Example (Język dowolnych nawiasów cd.)

W słowie w są poprawnie sparowane nawiasy, wtedy i tylko wtedy gdy w systemie $\langle \Sigma, \{() \rightarrow \varepsilon, [] \rightarrow \varepsilon, \langle \rangle \rightarrow \varepsilon\} \rangle$ wynikiem dla słowa w jest słowo ε .
Przykłady:

$$(\underline{[]} \langle \rangle) \Rightarrow (\underline{\langle \rangle}) \Rightarrow \underline{()} \Rightarrow \varepsilon$$

$$([\underline{\langle \rangle}]) \Rightarrow (\underline{[]})$$

Ćwiczenie

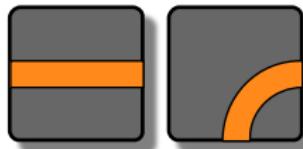
Napisz program rozstrzygający czy dowolne nawiasy są poprawnie sparowane bez używania systemu reguł.

Systemy regułowe

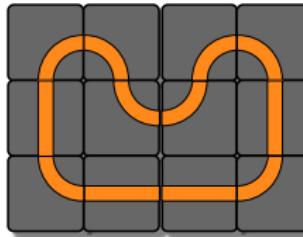
Przykłady

Example (Układanie klocków)

Mamy dwa rodzaje klocków:



Będziemy układać z nich zamknięte pętle takie jak np. poniższa:

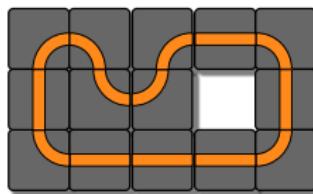


Systemy regułowe

Przykłady

Example (Układanie klocków cd.)

Chcemy umieć rozstrzygać czy pętla ułożona z klocków nie ma w swoim wnętrzu pustych pól. Poniższa pętla nie spełnia tego warunku gdyż zawiera jedno puste pole:



Ćwiczenie

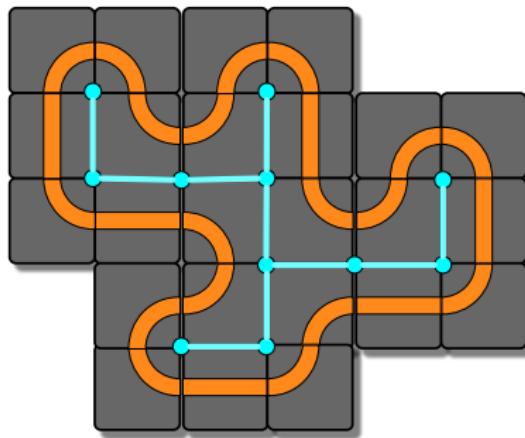
Napisz, bez użycia systemu regułowego, program rozstrzygający czy pętla nie zawiera wewnątrz pustych pól.

Systemy regułowe

Przykłady

Example (Układanie klocków cd.)

Pętle nie zawierające wewnętrz pustego pola mają ciekawą własność. Otóż przypominają one błonę opinającą szkielet będący drzewem (acyklicznym i spójnym grafem):

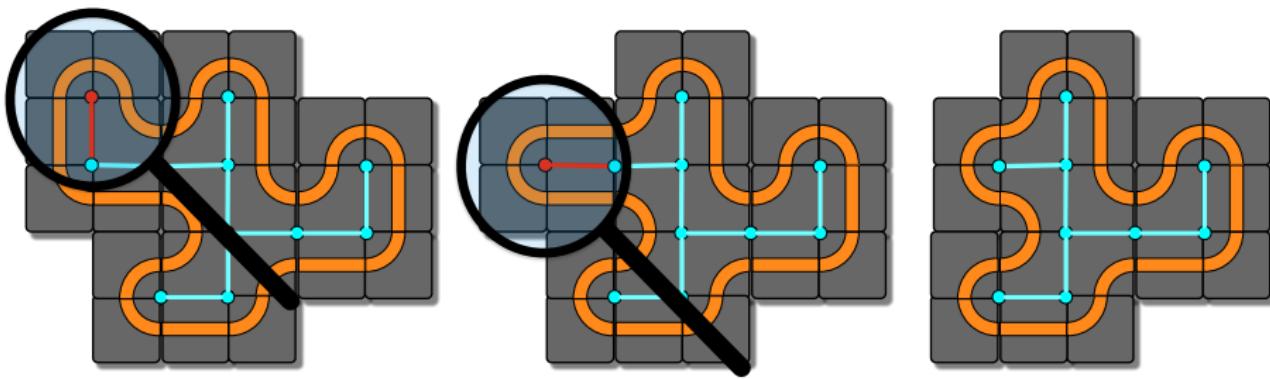


Systemy regułowe

Przykłady

Example (Układanie klocków cd.)

Rozpatrzmy następującą operację usuwania liści w drzewie stanowiącym szkielet pętli:

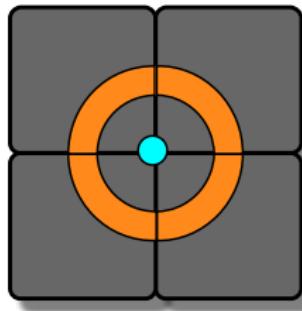


Systemy regułowe

Przykłady

Example (Układanie klocków cd.)

Jeśli uda się usuwając liście zredukować pętlę do takiego najprostszego przypadku oblekającego drzewo złożone z jednego wierzchołka:



to wyjściowa pętla nie zawierała wewnątrz pustego pola.

Systemy regułowe

Przykłady

Example (Układanie klocków cd.)

Pętlę będziemy kodować w postaci słowa nad alfabetem $\Sigma = \{F, L, R\}$ następująco:

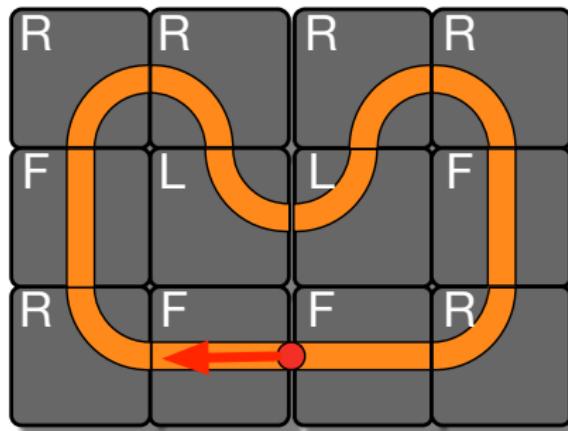
- Ustalamy dowolny punkt leżący na pętli ale na styku dwóch klocków.
- Przyjmujemy zwrot tak aby rozpoczynając ruch po pętli zgodnie z tym zwrotem obchodzić pętlę zgodnie z ruchami wskazówek zegara.
- Jeśli idziemy pętlą na wprost, to zapisujemy ten fakt literą F .
- Jeśli skręcamy w lewo, to zapisujemy ten fakt literą L .
- Jeśli skręcamy w prawo, to zapisujemy ten fakt literą R .
- Kończymy tworzenie kodu w chwili powrotu do punktu początkowego.

Systemy regułowe

Przykłady

Example (Układanie klocków cd.)

Przykład pętli i jej kodu:



kod pętli:

FRFRRLLRRFRF

długość kodu = liczba klocków

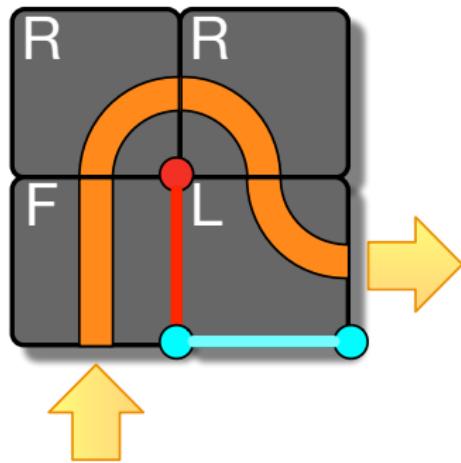
Systemy regułowe

Przykłady

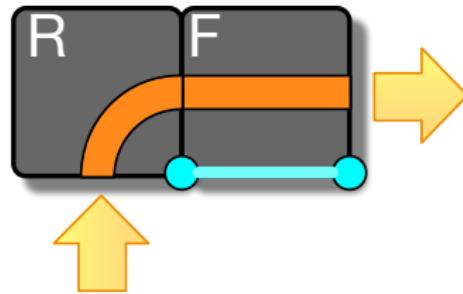
Example (Układanie klocków cd.)

Można opracować system $S = \langle \Sigma, P \rangle$, w którym reguły odpowiałyby usuwaniu liści w drzewie stanowiącym szkielet pętli.

Przykład takiej reguły:



FRRRL → RF



Systemy regułowe

Przykłady

Uwaga

- Należy pamiętać, że kod pętli w trzeba traktować cyklicznie, tzn. za ostatnim jego symbolem znajduje się ponownie pierwszy symbol.
- Próbując zastosować regułę $a \rightarrow b$ trzeba również rozpatrzyć przypadek, w którym słowo a może leżeć na granicy końca słowa w .
- Można zrobić to dzieląc słowo a na dwa słowa a' i a'' a następnie sprawdzać czy słowo w jest postaci $a''ua'$.
- Jeśli zachodzi taka sytuacja, to po zastosowaniu reguły otrzymujemy nowy kod pętli ub .

Systemy regułowe

Implementacja

Niech $S = \langle \Sigma, P \rangle$ będzie systemem regułowym ze zbiorem reguł P . Wynik dla danego słowa $w \in \Sigma^*$ można wyliczyć następująco:

```

1: function OBLICZ(w)
2:    $w_0 \leftarrow w;$ 
3:    $i \leftarrow 0;$ 
4:   while istnieje reguła  $a \rightarrow b \in P$  i takie słowa  $x$  i  $y$ , że  $w_i = xay$  do
5:      $i \leftarrow i + 1;$ 
6:      $w_i \leftarrow xby$        $\triangleright$  między kolejnymi słowami zachodzi  $w_{i-1} \Rightarrow w_i$ 
7:   end while
8:    $OBLICZ \leftarrow w_i$ 
9: end function
```

Theorem

Jeśli system S jest konfluentny, to obliczony wynik nie zależy od wyboru reguły $a \rightarrow b$ i od podziału słowa w na xay .

Systemy regułowe

Implementacja

- Jeśli każda reguła $a \rightarrow b \in P$ ma tę własność, że długość słowa a jest większa od długości słowa b , to obliczenia funkcji OBLICZ zawsze się kończą.
- Powyższy warunek jest wystarczający ale nie jest konieczny.

Example

W systemie $\langle \{0, 1\}, \{01 \rightarrow 100\} \rangle$ obliczenia kończą się dla dowolnego słowa $w \in \{0, 1\}^*$.

Wykład 12

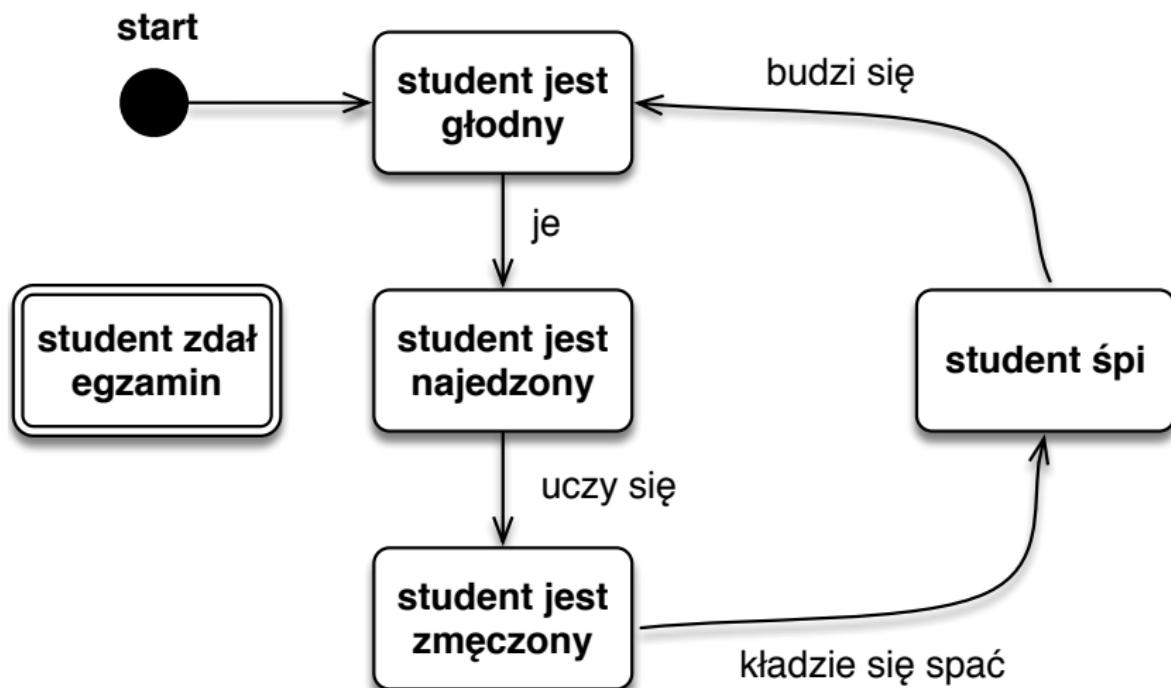
Automaty skończone

Automaty skończone

- definicja automatu
- przykłady automatów
- realizacja automatów w postaci programów

Automaty skończone

Definicja automatu



Automaty skończone

Definicja automatu

Definition (Automat skończony)

Automatem skończonym^a nazywamy uporządkowaną piątkę $\langle \Sigma, S, q_0, \delta, ACC \rangle$, w której:

Σ jest alfabetem,

S jest skończonym i niepustym zbiorem stanów,

$q_0 \in S$ jest stanem początkowym,

$\delta : S \times \Sigma \mapsto S$ jest funkcją przejścia,

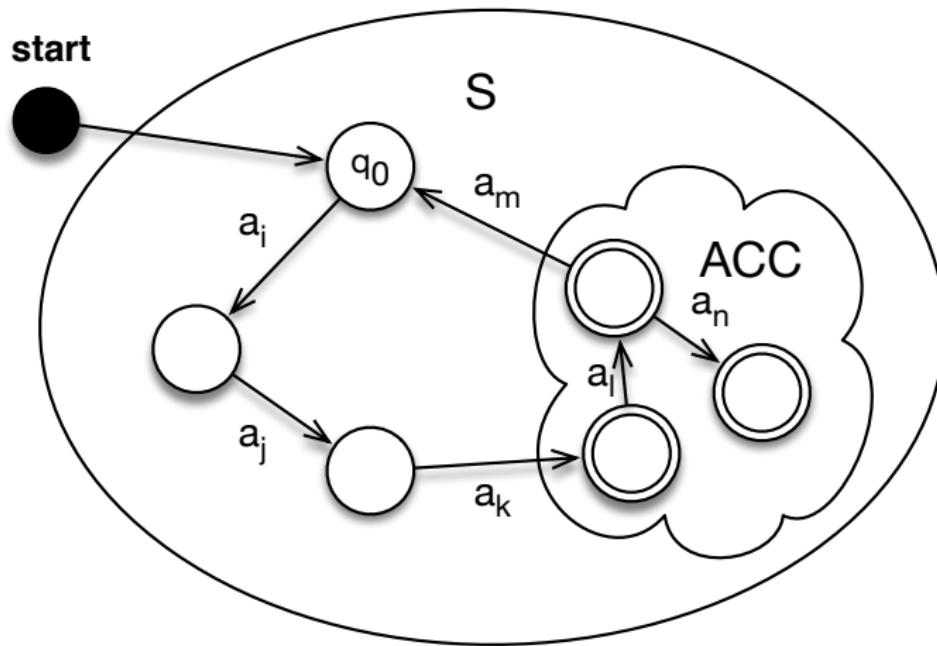
$ACC \subseteq S$ jest zbiorem stanów akceptujących.

^aDeterministycznym. O automatach niedeterministycznych będzie na innych kursach.

Automaty skończone

Definicja automatu

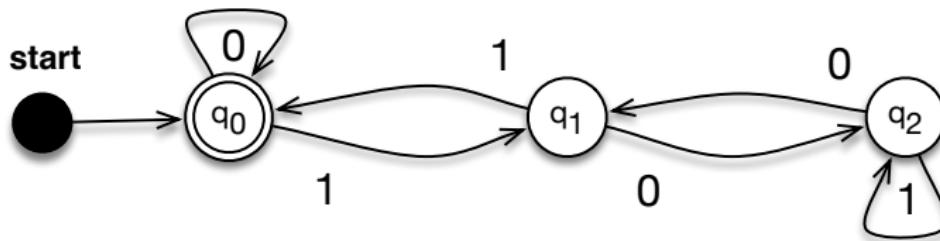
Automat przedstawiać będziemy w postaci grafu przejść między stanami:



Automaty skończone

Przykłady automatów

Example (Reszta z dzielenia)

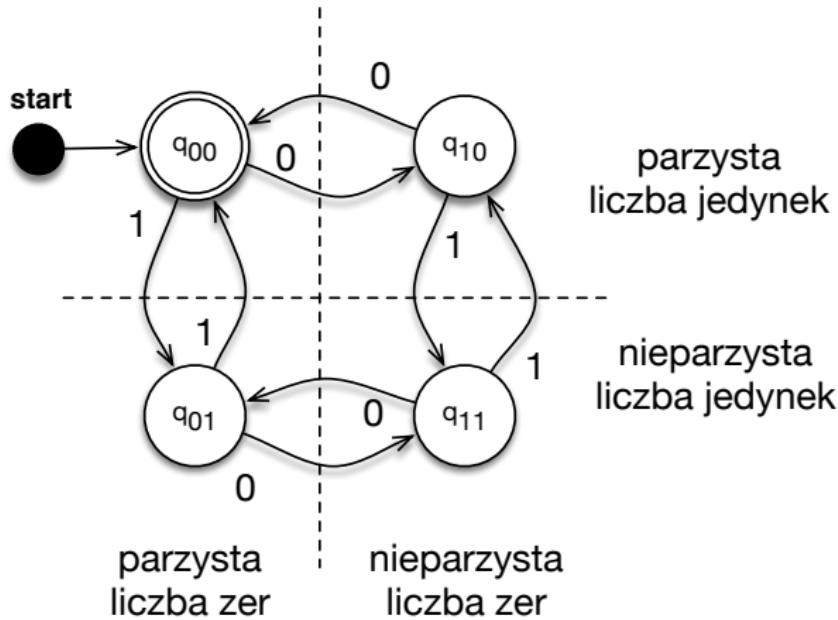


16	8	4	2	1		
1	0	1	0	1		$21 = 3 \times 7$
q_0	q_1	q_2	q_2	q_1	q_0	
					ACC	

Automaty skończone

Przykłady automatów

Example (Parzysta liczba zer i jedynek)



Automaty skończone

Przykłady automatów

Example (Jednakowa liczba zer i jedynek)

- Czy istnieje automat skończony akceptujący słowa złożone z tej samej liczby zer i jedynek?
- Na kursie **Języki Formalne i Techniki Translacji^a** dowiecie się, że nie ma takiego automatu skończonego.
- Dowiecie się również, że aby zaakceptować takie słowa potrzebny jest automat ze stosem.

^aObowiązkowy kurs na V semestrze.

Automaty skończone

Realizacja automatu (pseudokod)

```
1: function AUTOMATON( $\langle \Sigma, S, q_0, \delta, ACC \rangle$ ,  $w \in \Sigma^*$ )
2:    $t \leftarrow 0$ ;
3:   for  $i \leftarrow 1, \text{len}(w)$  do
4:      $q_{t+1} \leftarrow \delta(q_t, w_i)$ ;  $\triangleright$  wylicz nowy stan  $q_{t+1}$  na podstawie stanu
       bieżącego  $q_t$  oraz  $i$ -tego znaku ze słowa  $w$ 
5:      $t \leftarrow t + 1$ ;
6:   end for
7:    $AUTOMATON \leftarrow (q_t \in ACC)$ 
8: end function
```

Automaty skończone

Realizacja automatu (język C)

- Dwuwymiarowa tablica `delta[][]` przechowuje funkcję przejścia w następujący sposób:

$$\text{delta}[q][c] = q',$$

gdzie q i q' są numerami stanów a c jest indeksem znaku w alfabetie.

- Funkcja `decode(ch)` konwertuje kod ASCII znaku ch na jego indeks w alfabetie, np.

$$\text{decode('0')} = \text{decode}(48) = 0,$$

$$\text{decode('1')} = \text{decode}(49) = 1.$$

- Kod w języku C:

```
q = q0;  
while(*w)  
    q = delta[q][decode(*w++)];
```

Automaty skończone

Realizacja automatu (język C)

Example (Reszta z dzielenia przez 3)

```
#include <stdio.h>
int main()
{
    int q, delta[3][2] = { {0, 1}, {2, 0}, {1, 2} };
    char ch, *w, liczba[80];
    scanf("%s", liczba);
    w = liczba;
    q = 0;
    while(*w)
        q = delta[q][*w++ - '0'];
    printf("(%s) mod 3 = %d\n", liczba, q);
    return 0;
}
```

Automaty skończone

Realizacja automatu (język C)

Example (cd.)

```
$ ./automat  
110011  
(110011) mod 3 = 0  
$ ./automat  
1011  
(1011) mod 3 = 2
```

Wykład 13

Rozstrzygalność i obliczalność

Rozstrzygalność i obliczalność

-
-
-

Wykład 12

Nierozstrzygalność i nieobliczalność

Nierozstrzygalność i nieobliczalność

-
-
-

Wykład 15

Klasy złożoności

Klasy złożoności

-
-
-