

Algorytmy i złożoność



KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY



*Projekt „Modernizacja i rozwój systemu nauczania informatyki w Społecznej Wyższej Szkole Przedsiębiorczości i Zarządzania” współfinansowany przez Unię Europejską ze środków Europejskiego Funduszu Społecznego w ramach Priorytetu IV Szkolnictwo wyższe i nauka
Działania 4.1 Wzmocnienie i rozwój potencjału dydaktycznego uczelni oraz zwiększenie liczby absolwentów kierunków o kluczowym znaczeniu dla gospodarki opartej na wiedzy Poddziałania 4.1.1 Wzmocnienie potencjału dydaktycznego uczelni Programu Operacyjnego Kapitał Ludzki*

Literatura

1. Harold Abelson, Gerald J. Sussman, Struktura i interpretacja programów komputerowych, WNT 2002
2. Knuth Donald, The Art of Computer programming,
http://en.wikipedia.org/wiki/The_Art_of_Computer_Programming
3. Kenneth A. Ross, Charles R.B. Wright, Matematyka dyskretna, Wydawnictwo Naukowe PWN 1996
4. Wirth N., Algorytmy + struktury danych = programy, Wydawnictwa Naukowo-Techniczne, Warszawa 2004
5. E. Reingold, J. Nievergelt, N. Deo, Algorytmy kombinatoryczne, PWN 1985
6. S.E. Goodman, S.T. Hedetniemi, Introduction to The Design and Analysis of Algorithms, McGraw-Hill 1977
7. Witold Lipski, Kombinatoryka dla programistów, WNT 1982
8. Jacek Błazewicz, Złożoność obliczeniowa algorytmów kombinatorycznych, WNT 1988

Uruchamiamy naszą intuicję

Każdy z nas rozwiązywał jakieś problemy z wykorzystaniem komputera.

Jedno z centralnych pojęć w dziedzinie określanej jako „computer science”.

Dziedziny zbliżone: algebra, arytmetyka.

Algorytm (definicja „luźna”) – *niedwuznaczna* procedura rozwiązania problemu.

Procedura – skończona sekwencja *dobrze określonych* kroków lub operacji, z których każda wymaga jedynie skończonej ilości pamięci lub przestrzeni roboczej i zajmuje skończoną ilość czasu do pełnego wykonania.

Wymaganie. Algorytm powinien zakończyć się w skończonym czasie dla dowolnych danych wejściowych.

Precyzujemy nasze rozważania

Poprzednia definicja ma kilka wad:

- Termin „niedwuznaczny” jest sam w sobie dwuznaczny.
„Niedwuznaczny”, ale dla kogo. Przykład : obliczanie pochodnej wielomianu kwadratowego przez osobę nie znającą rachunku różniczkowego.
- Algorytm rozwiązania danego zadania może istnieć, lecz być trudny lub niemożliwy do opisanie w pewnym podanym formacie lub notacji.
Przykład – sznurowanie butów wyrażone wyłącznie w postaci słowa mówionego.

Możliwe rozwiązanie: zdefiniowanie *maszyny matematycznej* jako środowiska do uruchamiania i wykonywania algorytmów.

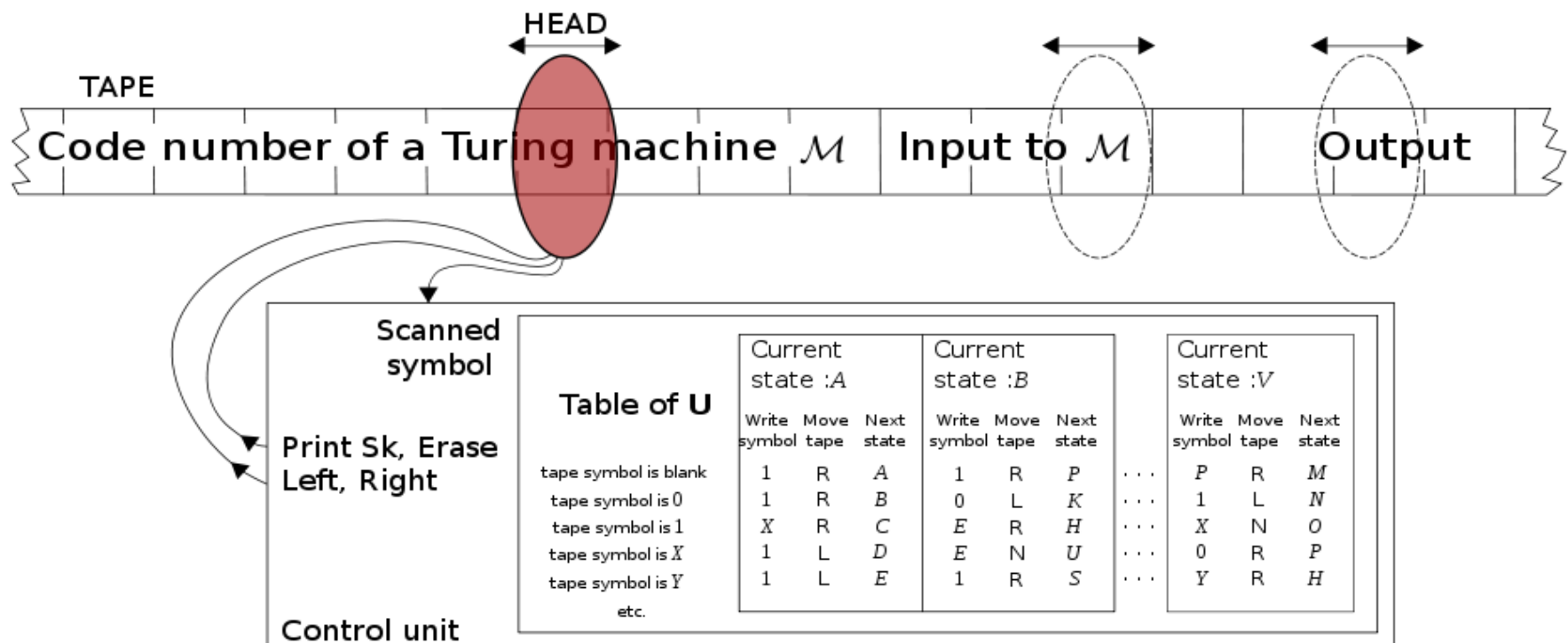
Uniwersalna maszyna Turinga

Maszyna Turinga - abstrakcyjny model komputera służący do wykonywania algorytmów.

Maszyna Turinga składa się z nieskończenie długiej taśmy podzielonej na pola. Taśma może być nieskończona jednostronnie lub obustronnie. Każde pole może znajdować się w jednym z M stanów. Maszyna zawsze jest ustawiona nad jednym z pól i znajduje się w jednym z N stanów. Zależnie od kombinacji stanu maszyny i pola maszyna zapisuje nową wartość w polu, zmienia stan i przesuwa się o jedno pole w prawo, w lewo lub pozostaje na miejscu. Taka operacja nazywana jest rozkazem. Maszyna Turinga jest sterowana listą zawierającą dowolną ilość takich rozkazów. Liczby N i M mogą być dowolne, byle skończone. Czasem dopuszcza się też stan $(N+1)$, który oznacza zakończenie pracy maszyny.

Źródło: <http://en.wikipedia.org>
<http://pl.wikipedia.org>

Uniwersalna maszyna Turinga - schemat



Źródło:

http://upload.wikimedia.org/wikipedia/commons/thumb/4/43/Universal_Turing_machine.svg/1000px-Universal_Turing_machine.svg.png

Uniwersalna maszyna Turinga - formalnie

Formalnie Maszynę Turinga opisuje się poprzez siódemkę:

$MT = \langle Q, \Sigma, \delta, \Gamma, q_0, B, F \rangle$, gdzie:

Q - skończony zbiór stanów

q_0 - stan początkowy, $q_0 \in Q$

F - zbiór stanów końcowych

Γ - skończony zbiór dopuszczalnych symboli

B - symbol pusty, $B \in \Gamma$

Σ - zbiór symboli wejściowych - podzbiór zbioru Γ nie zawierający B

δ - funkcja opisana następująco:

$$\delta : \Gamma \times Q \rightarrow Q \times \Gamma \times \{L, P, -\}$$

co oznacza że jest to funkcja pobierająca aktualny stan maszyny oraz symbol wejściowy a dającą w wyniku symbol jaki ma się pojawić na taśmie, kolejny stan maszyny oraz przesunięcie głowicy maszyny (lewo, prawo lub bez przesunięcia).

Źródło: http://pl.wikipedia.org/wiki/Maszyna_Turinga

John von Neumann

John von Neumann (ur. 28 grudnia 1903 w Budapeszcie, zm. 8 lutego 1957 w Waszyngtonie) – matematyk, inżynier chemik, fizyk i informatyk. Wniósł znaczący wkład do wielu dziedzin matematyki - w szczególności był głównym twórcą teorii gier, teorii automatów komórkowych (w które znaczący wkład miał także **Stanisław Ulam**), i stworzył formalizm matematyczny mechaniki kwantowej. Uczestniczył w projekcie Manhattan. Przyczynił się do rozwoju numerycznych prognoz pogody.

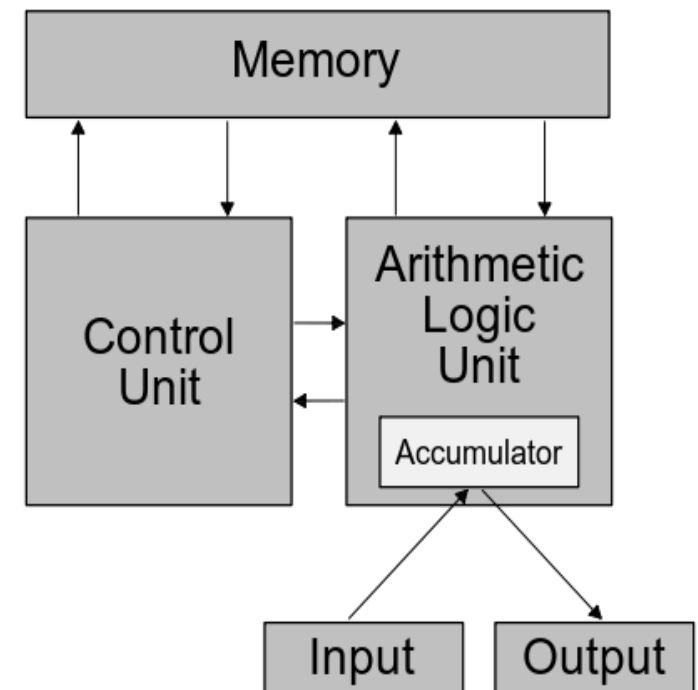
Architektura komputera w-g von Neumanna

Opracowana na projekcie EDVAC (Electronic Discrete Variable Automatic Computer), będącym następcą projektu ENIAC. Przedstawiona po raz pierwszy w roku 1945 w artykule *First Draft of a Report on the EDVAC*.

John von Neumann oparł się na pomysłe Turinga.

Jest to architektura, która – z pewnymi modyfikacjami – jest architekturą obecnych komputerów.

Fundamentalną cechą komputera von Neumanna jest fakt **modyfikacji** pamięci. Jest to pochodna cechy maszyny Turinga – głowica mogła **modyfikować stan taśmy**.



Sumowanie liczb całkowitych – problem

Zadanie. Posługując się urządzeniem o architekturze von Neumanna (Turinga) dodać do siebie dwie liczby zapisane w pewnym systemie liczbowym:

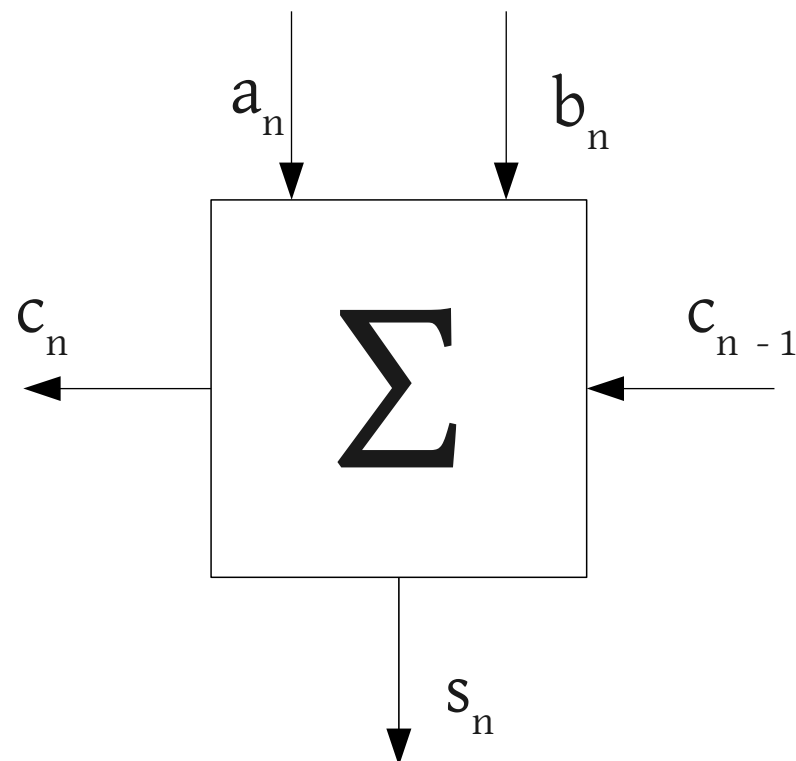
$$A = a_n a_{n-1} \dots a_2 a_1$$

$$B = b_n b_{n-1} \dots b_2 b_1$$

Wynikiem ma być suma:

$$S = s_n s_{n-1} \dots s_2 s_1$$

Sumator elementarny



a_n n-ty wyraz rozwinięcia liczby A

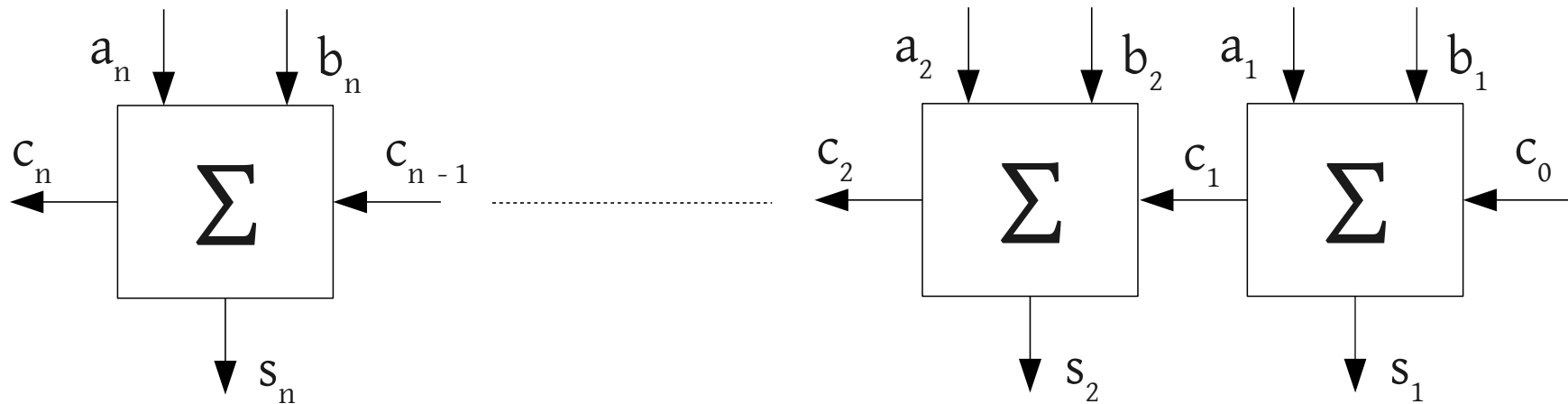
b_n n-ty wyraz rozwinięcia liczby B

c_{n-1} przeniesienie z poprzedniego sumatora, c_0 zwyczajowo jest równe 0

s_n n-ty składnik sumy

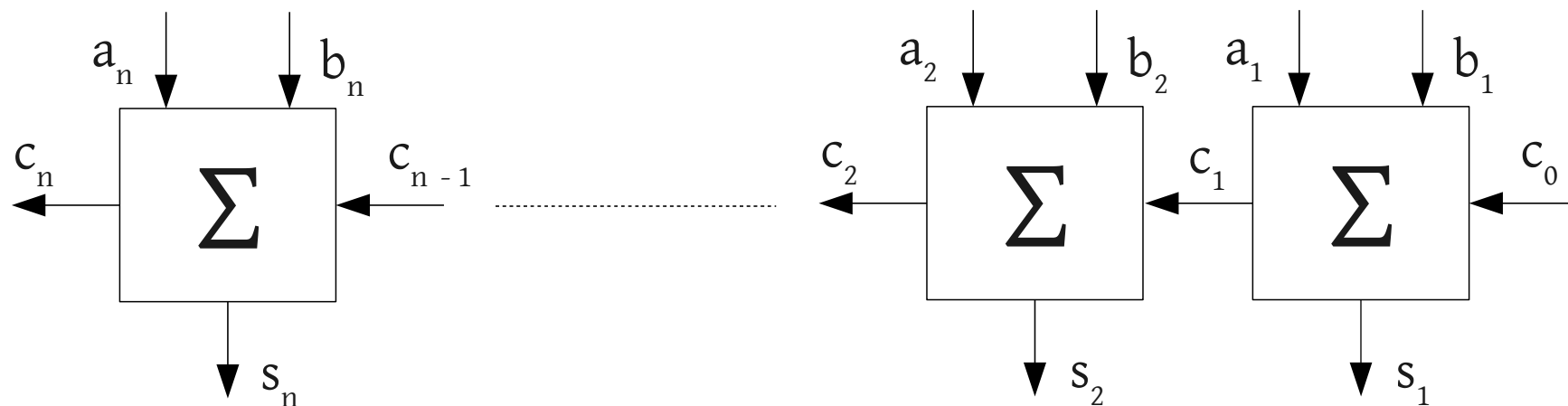
c_n aktualne przeniesienie

Sumator z przeniesieniami szeregowymi



Sumator szeregowy w systemie dziesiętnym

$$A = 949, B = 176, S = (1)125$$



$$a_3 = 9$$

$$b_3 = 1$$

$$c_2 = 1$$

$$s_3 = 1$$

$$c_3 = 1$$

$$a_2 = 4$$

$$b_2 = 7$$

$$c_1 = 1$$

$$s_2 = 2$$

$$c_2 = 1$$

$$a_1 = 9$$

$$b_1 = 6$$

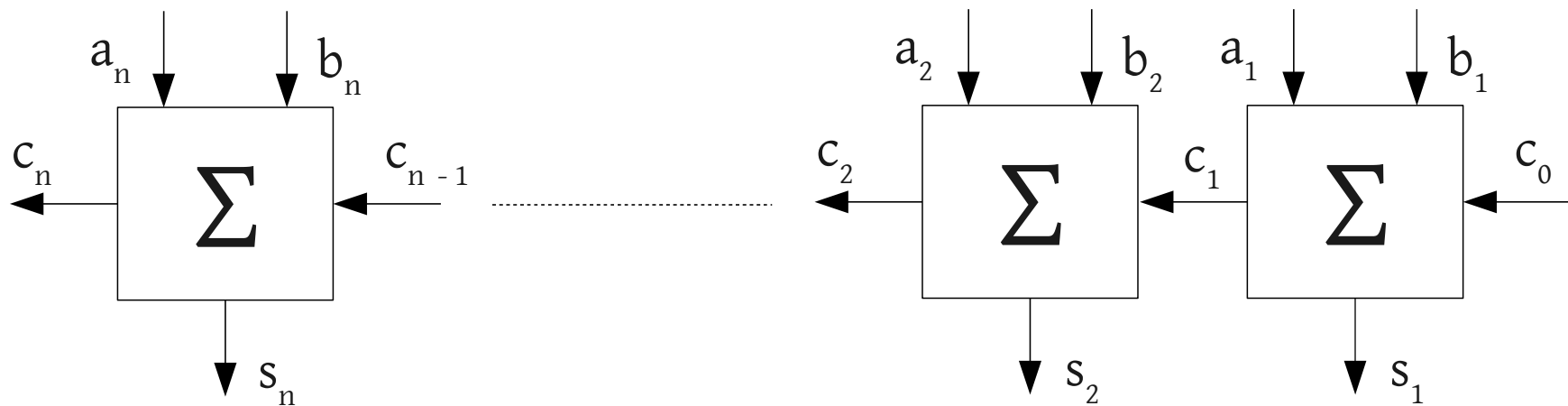
$$c_0 = 0$$

$$s_1 = 5$$

$$c_1 = 1$$

Sumator szeregowy w systemie dwójkowym

$$A = 101_2 (5_{10}), B = 110_2 (6_{10}), S = (1)011_2 (11_{10})$$



$$a_3 = 1$$

$$b_3 = 1$$

$$c_2 = 0$$

$$s_3 = 0$$

$$c_3 = 1$$

$$a_2 = 0$$

$$b_2 = 1$$

$$c_1 = 0$$

$$s_2 = 1$$

$$c_2 = 0$$

$$a_1 = 1$$

$$b_1 = 0$$

$$c_0 = 0$$

$$s_1 = 1$$

$$c_1 = 0$$

Ograniczenia

Pytanie: Co będzie, gdy nasza (ograniczona z natury) architektura sumatora składa się np. z 32 sumatorów elementarnych i próbujemy dodać do siebie dwie liczby A i B, takie, że $C_3 = 1$?

Odpowiedź:

```
unsigned int a, b, s;  
a = 4294967295;  
b = 1;  
s = a + b;  
printf("%d\n", s);
```

daje w wyniku na konsoli:
0

Dlaczego się tak dzieje ... ?

Maszyny matematyczne i algorytmy

- Mając zdefiniowaną maszynę matematyczną mianem *algorytmu* możemy określić **każdą procedurę**, którą można **uruchomić i wykonać** na takiej maszynie. **WYKONAĆ PROCEDURĘ TO ZNACZY OBLICZYĆ WARTOŚĆ jej zastosowania do parametrów aktualnych (operandów)** (ang. *computer* – „obliczacz” od słowa *compute* - obliczyć).
- Próby takiego formalnego zdefiniowania algorytmów są obarczone trudnościami związanymi z opanowaniem matematycznego aparatu, którym musimy się posłużyć.
- W praktyce programistycznej (przemysłowej) jest to podejście trudne w stosowaniu.
- Maszyn matematycznych używa się natomiast często do udowadniania rozstrzygalności i nierozstrzygalności problemów, do oceny ich złożoności obliczeniowej oraz do rozważań na temat obliczalności funkcji.

Dalsze precyzowanie rozważań

Dążymy do zachowania elastyczności i pewnej intuicyjności pierwszej definicji i jednocześnie przynajmniej częściowego usunięcia jej niejednoznaczności.

Wyobraźmy sobie „typowy” komputer cyfrowy, język do komunikowania się z nim oraz *procedurę* wyrażoną w tym języku, której nadajemy rangę ***algorytmu***. Komputer ten będzie posiadał nieograniczoną pamięć operacyjną, w której mogą być zapisywane liczby rzeczywiste, całkowite i stałe logiczne. Jedno słowo tej pamięci może pomieścić liczbę o dowolnym lecz skończonym rozmiarze. Dostęp do wartości umieszczonych w słowach tej pamięci możemy uzyskiwać w ustalonym, skończonym czasie (założenie wygodne i w praktyce *niemalże* prawdziwe – dlaczego?).

Dalsze precyzowanie rozważań – c.d.

Zdefiniowany komputer jest zdolny do wykonywania zapisanego programu, który składa się z ograniczonej kolekcji *instrukcji* określonego typu, ze standardowymi operatorami arytmetycznymi, operatorami porównania, rozgałęzieniami itd.. Zwykle jednej instrukcji przypiszemy jedną jednostkę czasu wykonania.

Pamięć naszego komputera może być zorganizowana w jedno-, dwu- i wielowymiarowe tablice (macierze), co będziemy zapisywać w postaci prostej deklaracji.

Ważne. Mówimy, że posiadamy *algorytm* rozwiązujący problem, jeśli możemy napisać program w języku komunikacji z komputerem, który to komputer będzie dany problem rozwiązywał. Kwestia do dalszej dyskusji, poza zakresem naszego wykładu – patrz przykład z wiązaniem sznurówek. Człowiek może wykonać ogromną różnorodność subtelnych operacji pozostających poza zakresem możliwości naszego komputera lub maszyny matematycznej. Pozostajemy więc w zakresie ograniczonej definicji algorytmu.

Algorytm MAX

Mając dane N liczb rzeczywistych w jednowymiarowej tablicy $R(1), R(2), \dots, R(N)$, znajdź M i J takie, że:

$$M = R(J) = \max_{1 \leq K \leq N} R(K)$$

W przypadku, gdy dwa lub więcej elementów z R mają wartość największą, wartość J będzie najmniejsza z możliwych (najmniejszy indeks tablicy).

Krok 0. [Inicjalizacja]. **set** $M := R(1)$; **and** $J := 1$.

Krok 1. [$N = 1$?] **if** $N = 1$ **then** STOP **fi**.

Krok 2. [Analizuj każdą wartość] **for** $K := 2$ **to** N **step** 1 **do** krok 3 **od**; **and** STOP.

→ Krok 3. [Porównaj] **If** $M < R(K)$ **then set** $M := R(K)$; **and** $J := K$ **fi**.

M jest teraz największą z przeanalizowanych liczb, znajduje się na pozycji J – tej.

Algorytm MAX - implementacja

```
r = [1, 2, -34, 16, 3, 8, 4]
n = len(r)
liczba = MAX(R = r, N = n)
```

```
def MAX (R, N):
    M = R[0]
    J = 0
    if N == 1:
        return (M, J)
    else:
        for K in range(1, N):
            if M < R[K]:
                M = R[K]
                J = K
        return (M, J)
```

- Implementacja algorytmu MAX w języku programowania Python

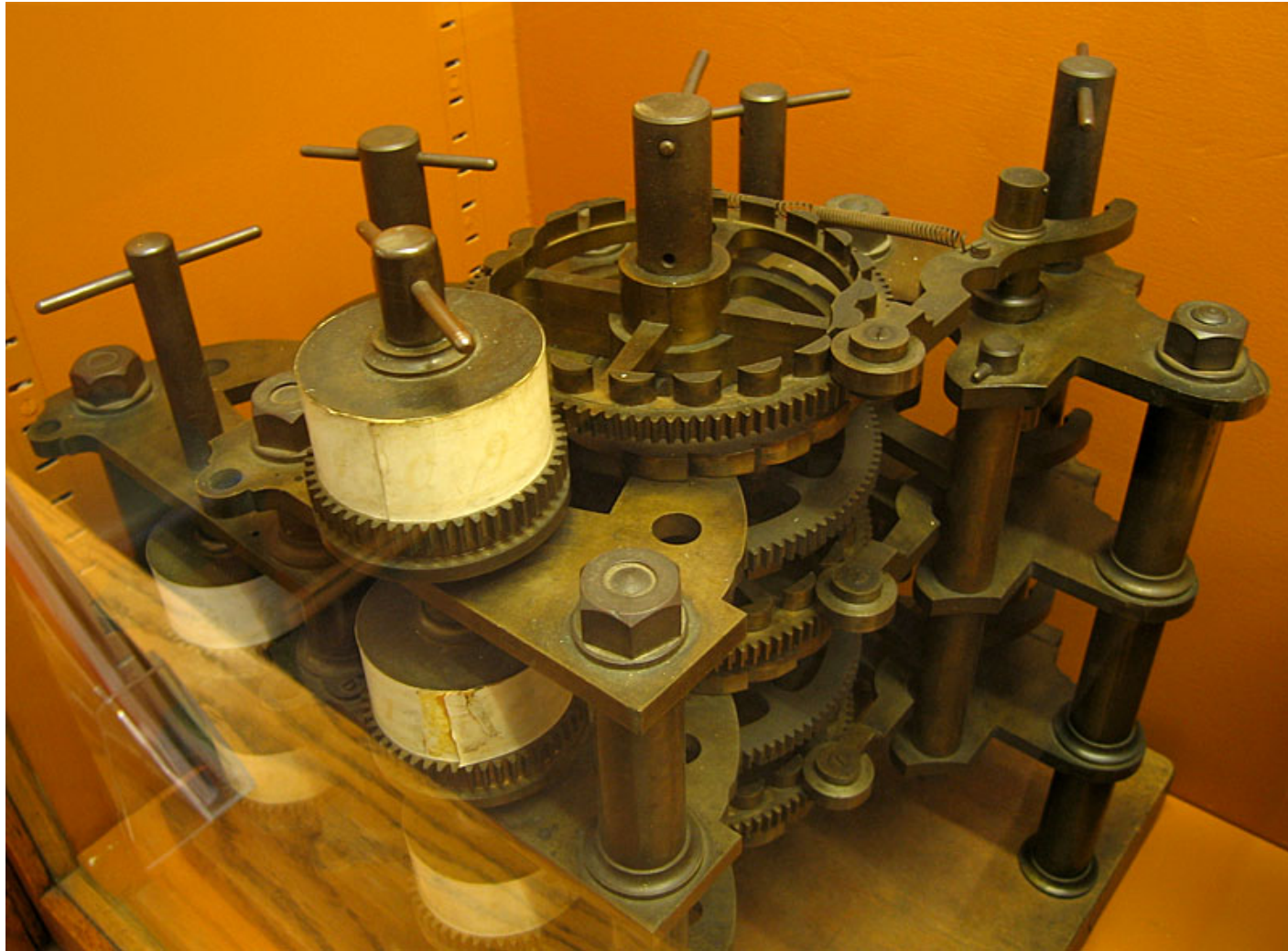
Etymologia słowa *algorytm*

Słowo *algorytm* pochodzi od nazwiska arabskiego matematyka z IX wieku Muhammeda ibn Musa Alchwarizmiego. Początkowo słowem *algorism* nazywano czynności konieczne do wykonywania obliczeń z użyciem dziesiętnego systemu liczbowego.

Obecne znaczenie słowa *algorytm* jako zestawu ścisłych reguł powstało wraz z rozwojem matematyki i techniki. Wynalezienie zbiorów zasad pozwalających na obliczanie parametrów konstruowanych maszyn, stało się podstawą rewolucji przemysłowej zapoczątkowanej w końcu XVIII stulecia. Jednak dopiero zbudowanie maszyn, które same mogły realizować pewne proste algorytmy, stało się przełomem. Początkowo miały one postać układów mechanicznych mogących dokonywać prostych obliczeń. Ogromnego postępu dokonał w tej dziedzinie w 1842 roku *Charles Babbage*, który na podstawie swoich doświadczeń sformułował ideę maszyny analitycznej zdolnej do realizacji złożonych algorytmów matematycznych.

Źródło: <http://pl.wikipedia.org/wiki/Algorytm>

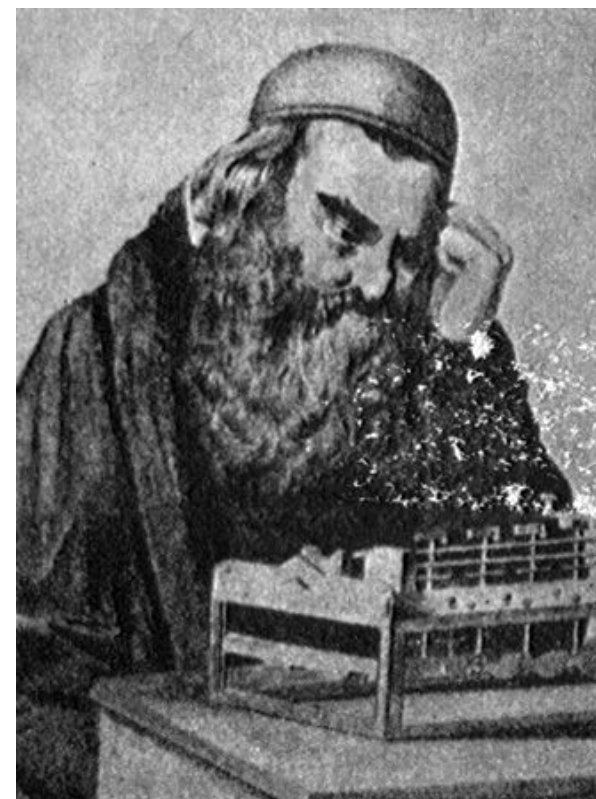
Maszyna różnicowa Babbage'a (1842)



Źródło: <http://upload.wikimedia.org/wikipedia/commons/5/53/BabbageDifferenceEngine.jpg>

Maszyna Abrahama Sterna (1814, 1817)

Abraham Stern – warszawski zegarmistrz, który skonstruował urządzenie mechaniczne zdolne, m.in. do obliczania wartości pierwiastków kwadratowych. Podstawowa różnica pomiędzy maszyną Sterna a (późniejszą) maszyną Babbage'a była taka, że maszyna Sterna nie była podlegała rekonfiguracji (programowaniu).



Źródło: <http://www.rechenmaschinen-illustrated.com/>

Etapy konstruowania algorytmu

1. Sformułowanie problemu.
2. Opracowanie modelu matematycznego.
3. Projekt algorytmu.
4. Ocena poprawności.
5. Implementacja.
6. Ocena złożoności obliczeniowej.
7. Testowanie.

Etapy 5-7 mogą być ułożone również w nieco innej kolejności:

5. Ocena złożoności obliczeniowej.
6. Implementacja.
7. Testowanie.

Proces konstruowania algorytmu może mieć (i na ogół ma) charakter iteracyjny.

Język C – historia

Opracowany w *Laboratoriach Bella* (należących do firmy *AT&T*) w latach 1969 – 1972 (1973). Jego autorami byli *Ken Thompson i Dennis Ritchie*, twórcy systemu operacyjnego *UNIX*, którzy potrzebowali języka programowania stosunkowo niskiego poziomu (abstrakcji) mogącego zastąpić assembler. Protoplastami języka *C* są języki *B* oraz *BCPL* – pierwszy język w historii, który posługiwał się nawiasami klamrowymi do wyznaczania początków i końców bloków instrukcji (*Algol* posługiwał się zwyczajowo słowami *begin ... end*).

Język C jest przykładem *języka dziedzinowego* (ang. *domain-specific-language*). Został on stworzony po to, by napisać w nim jądro systemu operacyjnego UNIX.

Zadziwia to, do jakiego stopnia C oraz jego pochodne (C++, Objective-C) „opanowały” branżę IT, będąc używane w zastosowaniach, którym zupełnie nie odpowiadają ich cechy.

Język C – literatura

1. Język ANSI C, Brian W. Kernighan, Dennis M. Ritchie, WNT Warszawa 2000, wyd. V, ISBN 83-204-2620-0
2. <http://pl.wikibooks.org/wiki/C>
3. http://en.wikibooks.org/wiki/C_Programming
4. A. D. Marshall. Programming in C. UNIX System Calls and Subroutines using C. <http://www.cs.cf.ac.uk/Dave/C/>

Język C – cechy

- Podlega **kompilacji** do kodu maszynowego (pierwsza wersja była interpretowana).
- Posiada **statyczny i słaby system typów**.
- Dominującym stylem programowania jest **styl imperatywny**, polegający na zmianie stanu procesu obliczeniowego przy użyciu zmiennych i operatora przypisania (=).
- Podstawowym środkiem do realizacji abstrakcji są **procedury** (zwyczajowo nieściśle zwane funkcjami), **tablice** oraz złożone typy danych – **struktury** w połączeniu z możliwością definiowania własnych, nazwanych typów z wykorzystaniem wyrażenia **typedef ...** ;
- **Programowanie strukturalne**, tzn. bez instrukcji **goto** z wykorzystaniem instrukcji warunkowych **if** i **case** oraz pętli **for**, **while**, **do-while**.
- **Typ wskaźnikowy** do budowania **relacji** pomiędzy obiektami.
 - **Wskaźniki** w C to cecha języka przesądzająca o jego elastyczności, ale i powodująca wiele zagrożeń. **Arytmetyka wskaźnikowa** jest głównym powodem powstawania **luk i błędów** w systemach. Można się pokusić o stwierdzenie, że bez wskaźników **wirusy** by nie istniały. Lub przynajmniej byłoby ich mniej.
- **Preprocesor** z **makrodefinicjami** działającymi w czasie kompilacji jak generatory kodu.

Co to jest abstrakcja ?

Abstrakcja (łac. *abstractio* - *oderwanie*) – w sztukach plastycznych: taka realizacja dzieła, w której jest ono pozbawione wszelkich cech ilustracyjności, a artysta nie stara się naśladować natury. Autorzy stosują różne środki wyrazu, dzięki którym "coś przedstawiają".

Kompozycja VII (1913).
Wasilij Wasiljewicz
Kandinski (ur. 4 grudnia 1866 w Moskwie, zm. 13 grudnia 1944 w Neuilly-sur-Seine) – rosyjski malarz, grafik i teoretyk sztuki. Współtwórca i jeden z najważniejszych przedstawicieli abstrakcjonizmu.



[http://pl.wikipedia.org/wiki/Abstrakcja_\(sztuka\)](http://pl.wikipedia.org/wiki/Abstrakcja_(sztuka))
http://en.wikipedia.org/wiki/Wassily_Kandinsky

Abstrakcja w filozofii, matematyce i psychologii

1. (łac. ***abstractio*** – ***oderwanie***). Proces budowania pojęć, w którym od rzeczy jednostkowych (konkretnych) dochodzi się do pojęcia ogólnego poprzez wyłanianie tego, co dla tych rzeczy jest wspólne. ***Uogólnianie***.

2. Potężny element naszego procesu myślowego pozwalający na niwelowanie ograniczeń naszego umysłu. Polega na ***pomijaniu*** (abstrahowaniu od) ***szczegółów pojęć***, którymi się posługujemy w ***procesie wnioskowania***.

Funkcja jako pojęcie abstrakcyjne

Funkcja - relacja określona na zbiorach A (dziedzinie) i B (przeciwdziedzinie), w której elementowi ze zbioru A odpowiada jeden i tylko jeden element ze zbioru B. Właściwość ta zachodzi **zawsze**. W matematyce **czas nie istnieje**.

Funkcja = tablica o potencjalnie nieskończonych (nieprzeliczalnych) rozmiarach (zależnie od właściwości dziedziny).

$1 \rightarrow 5$

$2 \rightarrow 7$

$3 \rightarrow 9$

$\cdot \rightarrow \cdot$

Abstrakcyjne ujęcie funkcji – receptura (przepis) na generowanie kolejnych elementów tablicy.

$f: x \rightarrow 2x + 3$ dla x należących do zbioru \mathbf{R} .

Obiekt

„**Byt** posiadający *tożsamość*, to znaczy – *jednoznacznie odróżnialny* od wszystkich innych bytów.”

Na podstawie: Harold Abelson, Gerald J. Sussman, Struktura i interpretacja programów komputerowych, WNT 2002

Na czym polega tożsamość kredy ? A na czym tożsamość człowieka ... ?

Rozważania o przestrzeniach

Wyobraźmy sobie przestrzeń obiektów:

3 1 „Pada deszcz”
2 + „Świeci Słońce”
e Π
- * cos
sin

Wszystkie elementy tej przestrzeni są równorzędne pod względem istotności. Jako takie mogą podlegać złożeniom. Np.

$$1 + 2 ==> 3$$

$$\sin 0 ==> 0$$

Rozważania o przestrzeniach

Wyobraźmy sobie przestrzeń obiektów:

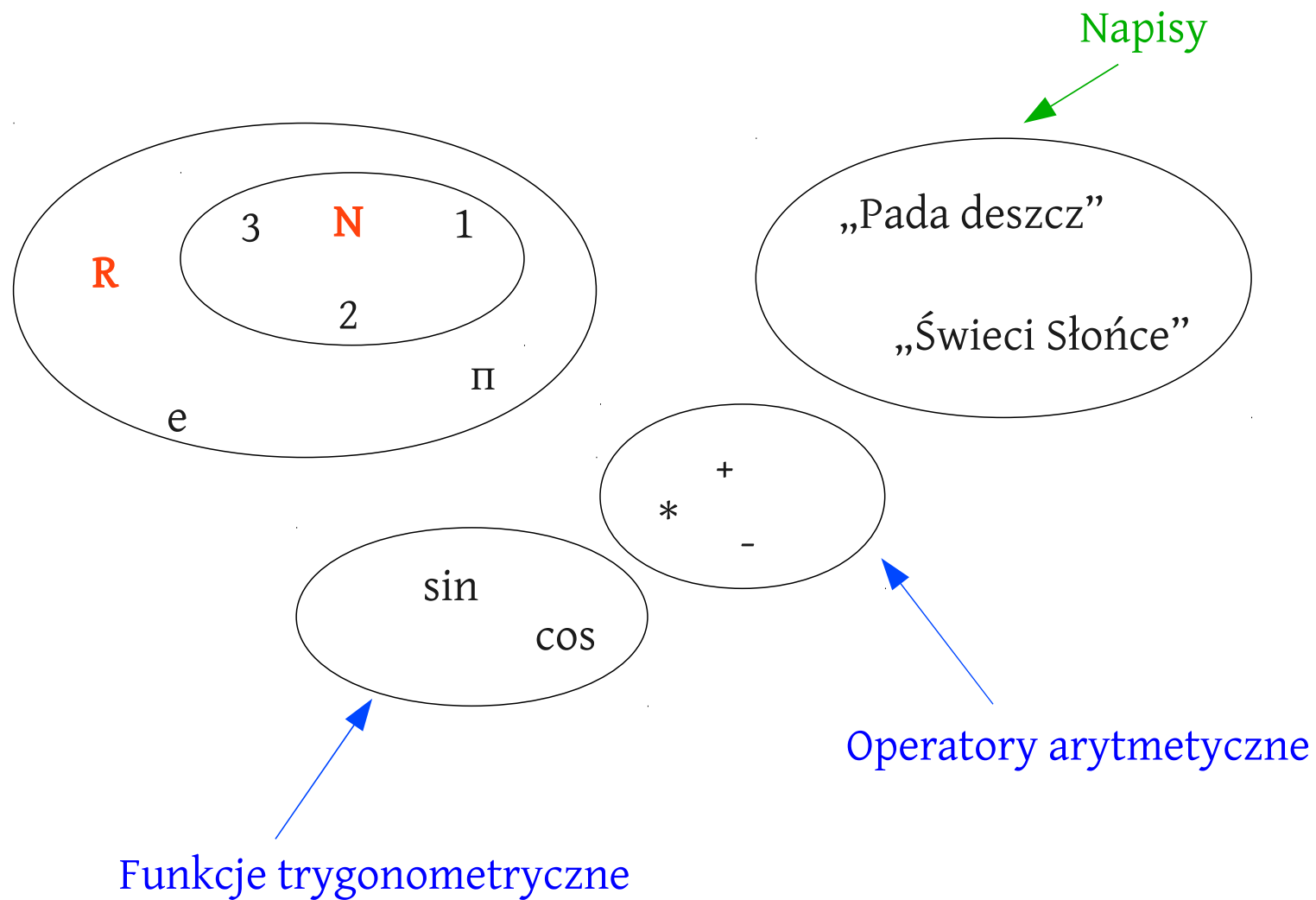
$$\begin{array}{ccccc}
 3 & 1 & & & \text{„Pada deszcz”} \\
 & 2 & & + & \text{„Świeci Słońce”} \\
 & & \Pi & & \\
 e & & & & \\
 & - & & * & \cos \\
 & & \sin & &
 \end{array}$$

Co jest wartością wyrażeń:

\sin „Pada deszcz”

$2 + +$

Klasyfikacja



TYP

1. **Zbiór więzów i ograniczeń** nałożonych na **dane (obiekty)** w celu **zagwarantowania** ich **poprawnego użycia**. Chodzi o użycie w zestawieniu z funkcjami (operatorami) w zdaniach pewnego języka (zwykle formalnego).

2. **Zbiór danych (obiektów)** charakteryzujących się tym, że wszystkie one **spełniają** ten sam zestaw **więzów i ograniczeń**.

Powyższe komplementarne definicje zostały wypracowane przez znanych matematyków amerykańskich: **Danę Scotta i Christophera Strachey**.

W niektórych językach programowania – zwanych zorientowanymi obiektowo – występuje pojęcie **klasy**. Jest ono pojęciem zgodnym co do joty z powyższą ogólną definicją.

KLASA = TYP !!!

Najważniejsze typy danych w języku C

W dalszych rozważaniach będziemy wykorzystywali następujące typy danych:

int – zbiór liczb całkowitych (ujemnych, dodatnich oraz 0) z zakresu $-2^{31}..2^{31}-1$

unsigned int – zbiór liczb naturalnych z zakresu $0..2^{32}-1$

char – zbiór liczb całkowitych z zakresu $-128..127$. Wartości tego typu są wykorzystywane do reprezentowania znaków alfabetu ASCII.

UWAGA. Wartości typów całkowitych mogą być (i zwykle są) wykorzystywane do reprezentowania prawdy lub fałszu (patrz ***algebra Boole'a***). Zwyczajowo 0 = fałsz, 1 = prawda.

double – zbiór liczb rzeczywistych (w istocie – wymiernych, dlaczego?). Zakres rzędu $\pm 1.79769e+308$, zdolność do reprezentacji małych liczb (okolice zera) rzędu $2.22507e-308$.

Powyższe zakresy dotyczą platformy 32-bitowej i kompilatora gcc.

Procedury a funkcje

Procedura to obiekt będący reprezentantem (abstrakcją, uogólnieniem) **ciągu kroków obliczeniowych**. Kroki te mogą być wyrażone w postaci dosłownej – jako tzw. **instrukcje** ustawione w odpowiednim **porządku** (w czasie) lub niejawnie – jako etapy wyznaczania wartości (**ewaluacji**) **wyrażeń** (zdań języka posiadających wartość) których kolejność wynika z semantyki języka.

Niektóre procedury służą do obliczania wartości **funkcji** matematycznych z wykorzystaniem automatu – maszyny Turinga (w praktyce – komputera von Neumanna).

Turing pokazał w roku 1936, że w matematyce istnieją funkcje, których wartości **nie da się obliczyć** przy użyciu maszyny. Jego rozważania dotyczyły więc obliczalności funkcji.

Kiedy procedura nie jest funkcją ?

- **Procedura** jest więc w pewnych warunkach **maszynową realizacją funkcji** matematycznej.
- **Procedura** jest jednocześnie **maszynową realizacją algorytmu**.
- W ujęciu matematycznym **ALGORYTM** jest więc **FUNKCJĄ**. Funkcja dla dowolnego elementu należącego do dziedziny podaje jeden i zawsze ten sam element należący do zbioru wartości. Stąd nasze początkowe wymaganie dot. **jednoznaczności** algorytmu.

Czasem procedury **modyfikują stan** środowiska, w którym są uruchamiane. Ta modyfikacja jest **efektem ubocznym** zadziałania procedury. O procedurze, która zachowuje się w ten sposób nie wolno powiedzieć, że jest to funkcja (w sensie matematycznym).

Bardzo wiele programów posługuje się masowo pojęciem stanu, modyfikacji, zmiennej. Zjawisko to powoduje utratę matematycznego piękna w programach. Czasem jest niezbędne, lecz liczba takich koniecznych sytuacji jest zdecydowanie mniejsza niż wskazuje na to stopień użycia programistycznych technik wykorzystujących modyfikację stanu w zastosowaniach przemysłowych.

Procedura (nie funkcja) *printf* w języku C

Jest przykładem procedury uruchamianej dla uzyskania efektu ubocznego – wypisania na standardowym wyjściu (zwykle konsoli) pewnego tekstu.

```
#include <stdio.h> /* Plik nagłówkowy zawierający deklarację printf  
                      /usr/include/stdio.h */
```

```
printf("Koniec.");  
printf("Wartość pewnej liczby całkowitej to %d.\n", 256);
```

Przydatne przełączniki:

\n - symbol przejścia do nowej linii, \t - symbol tabulacji (tab)

%d - liczba całkowita ze znakiem w formacie dziesiętnym

%i - synonim dla %d

%x - liczba całkowita bez znaku w formacie szesnastkowym, z użyciem małych liter

%X - liczba całkowita bez znaku w formacie szesnastkowym, z użyciem wielkich liter

%o - liczba całkowita bez znaku w formacie oktalnym

%u - liczba całkowita bez znaku w formacie dziesiętnym

%e - liczba zmiennoprzecinkowa w zapisie naukowym (1.2345e+3)

%E - liczba zmiennoprzecinkowa w zapisie naukowym (1.2345E+3)

%f - liczba zmiennoprzecinkowa typu double w zapisie dziesiętnym (123.45)

%s - łańcuch tekstowy

%p - wskaźnik

Na podstawie: <http://pl.wikipedia.org/wiki/Printf>

Instrukcja warunkowa *if*

Instrukcja – nie posiada wartości. Jest używana do manipulowania przepływem sterowania w programie w zależności od spełnienia bądź niespełnienia warunków – wyrażeń warunkowych.

```
if (wyrażenie-warunkowe) {  
    wykonaj-gdy-prawda  
}  
else {  
    wykonaj-gdy-nieprawda  
}
```

wyrażenie-warunkowe jest wyrażeniem oznaczającym **prawdę**, gdy **wynik** jego wykonania (ewaluacji) jest **niezerowy** i **fałsz** – gdy wartość tego wyrażenia wynosi **0**.

W powyższym przykładzie, gdy **wyrażenie-warunkowe** przyjmie wartość różną od 0, wykonane zostaną instrukcje i wyrażenia oznaczone tutaj symbolicznie jako **wykonaj-gdy-prawda**, w przeciwnym wypadku zostaną wykonane instrukcje i wyrażenia oznaczone jako **wykonaj-gdy-nieprawda**.

Blok **else {}** jest opcjonalny; możliwe jest jego pominięcie, gdy chcemy jedynie wykonać instrukcje i wyrażenia, gdy warunek jest prawdziwy:

```
if (wyrażenie-warunkowe) {  
    wykonaj-gdy-prawda  
}
```


Instrukcja warunkowa *if*

Bardziej skomplikowana wersja instrukcji warunkowej zakłada, że weryfikujemy kolejno kilka warunków (wyrażeń warunkowych), znajdujemy pierwszy warunek o niezerowej wartości (prawdziwy) i wykonujemy odpowiadający mu blok instrukcji i wyrażeń. Pozostałych warunków już nie weryfikujemy.

```
if (warunek-1) {  
    wykonaj-gdy-prawda-1  
}  
else if (warunek-2) {  
    wykonaj-gdy-prawda-2  
}  
...  
else if (warunek-N) {  
    wykonaj-gdy-prawda-N  
}  
else {  
    wykonaj-gdy-żaden-niespełniony  
}
```

Blok *else* {} jest tak samo jak poprzednio opcjonalny.

Operatory w wyrażeniach logicznych

Tak jak wyrażenia o charakterze obliczeniowym mogą składać się z elementów składowych, pod-wyrażeń, np.

$x + y$

tak również wyrażenia logiczne mogą składać się z pod-wyrażeń łączonych operatorami logicznej:

- negacji !
- sumy (alternatywy, ang. *or*) //
- koniunkcji (iloczynu, ang. *and*) &&

Negacja logiczna

Zaprzeczenie, w rachunku zdań zapisywane jako $\sim p$ lub $\neg p$ dla pewnego zdania p . Oznacza „nieprawda, że p ”. Gdy p jest prawdziwe, zdanie to przyjmuje wartość fałsz, gdy p jest fałszywe – prawda.

p	$\neg p$
0	1
1	0

W powyższej tabeli przyjmujemy, że **1** oznacza *prawdę* logiczną, **0** – *fałsz*.

W języku C negację oznacza się symbolem **!**, np.

```
!(x == 4)
```

Powyższe zdanie jest prawdziwe, gdy wartość symbolu x nie jest równa 4.

Alternatywa logiczna

Operator, którego użycie ze zdaniami p i q – zdanie „ p **lub** q ” – przyjmuje wartość **prawda**, wtedy i tylko wtedy, gdy **którekolwiek** ze zdań p , q **lub obydwa** są **prawdziwe**. W rachunku zdań wyrażenie zbudowane z wykorzystaniem tego operatora zapisujemy jako $p \vee q$.

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

W języku C alternatywę logiczną oznacza się symbolem `||`, np.

```
X == 3 || x > 5
```

Powyższe zdanie jest prawdziwe, gdy wartość symbolu x jest równa 3 lub większa od 5.

Short-circuit evaluation

Jeśli mamy do czynienia z alternatywą logiczną postaci

`warunek-1 || warunek-2`

i **warunek-1** jest spełniony, prawdziwy, to **warunek-2** w ogóle nie zostanie poddany wartościowaniu (dlaczego można tak postąpić?) i całe wyrażenie zyskuje wartość **prawda**. Technika ta pojawia się w zdecydowanej większości popularnych języków programowania. Jej umiejętne użycie może skutkować znacznymi optymalizacjami szybkości działania naszych algorytmów – np. kiedy wyznaczenie wartości wyrażenia **warunek-2** jest bardzo kosztowne, a **warunku-1** – nie. Pozwala ona również konstruować wyrażenia jak poniżej:

```
if (x == 0 || 1 / x < epsilon) { ... }
```



Kiedy x przyjmuje wartość 0, drugie **wyrażenie** w ogóle nie jest poddane wartościowaniu. Dlatego tylko nie otrzymujemy błędu.

Warunkiem uzyskania korzyści z zastosowania tej techniki jest to, by wyrażenia były ze sobą logicznie powiązane.

Iloczyn logiczny

Operator, którego użycie ze zdaniami p oraz q – zdanie „ p i q ” – przyjmuje wartość **prawda**, wtedy i tylko wtedy, gdy **obydwa zdania** są **prawdziwe**. W rachunku zdań wyrażenie zbudowane z wykorzystaniem tego operatora zapisujemy jako $p \wedge q$.

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

W języku C iloczyn logiczny oznacza się symbolem `&&`, np.

`x > 3 && x < 5`

Powyższe zdanie jest prawdziwe, gdy wartość symbolu x jest większa niż 3 i mniejsza od 5.

Iloczyn logiczny

Podobnie, jak to miało miejsce w przypadku alternatywy logicznej, tutaj również występuje skrócona ewaluacja. Jeśli w wyrażeniu:

warunek-1 && warunek-2

warunek-1 jest **falszywy**, to warunek-2 nie jest w ogóle poddany wartościowaniu i całe wyrażenie przyjmuje wartość **falsz**. Pozwala to np. napisać:

```
if (x != 0 && 1 / x < epsilon) { ... }
```

Dzielenie 1 przez x zostanie wykonane jedynie wówczas, gdy x nie narusza warunków prawidłowości wykonania tego wyrażenia.

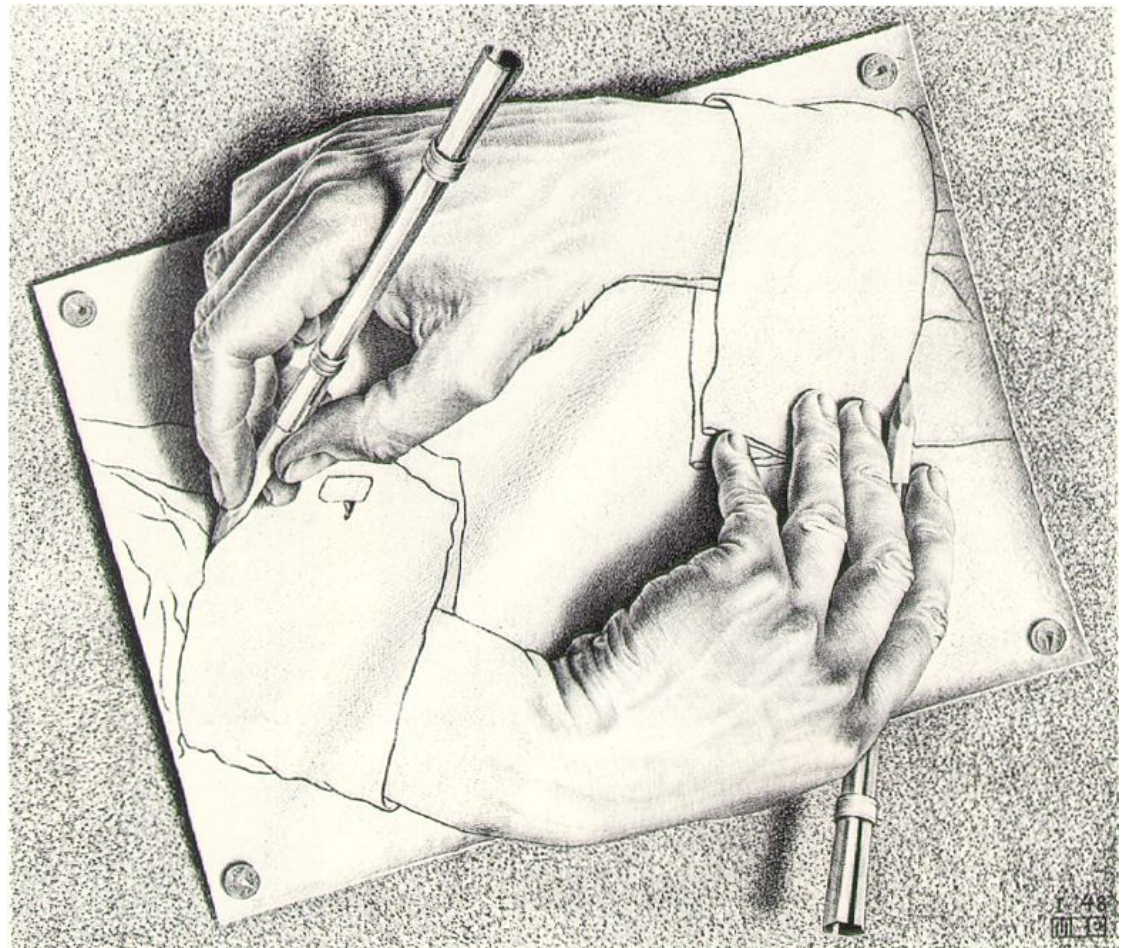
Short-circuit evaluation jest jedną z najważniejszych technik stosowanych przy **walidacji**, tj. weryfikacji poprawności danych, na których działają nasze algorytmy.

Rekurencja i iteracja

„Iteracja jest rzeczą ludzką, rekurencja – boską.” - *L (Lawrence) Peter Deutsch*, hacker, twórca Ghostscripta (interpreter/przeglądarka plików Postscript i PDF).

Drawing Hands, litografia (1948).

Maurits Cornelis Escher (1898 – 1972), holenderski grafik znany ze swoich matematycznych inklinacji.



Rekurencja

(łac. *recurrere* – przybiec z powrotem). Zjawisko występujące, gdy **definicja** pewnego bytu **odwołuje się do** tegoż **definiowanego bytu**. Rodzaj intelektualnego „zapętlenia”. W praktyce przyjmuje ono następujące kształty:

1. W matematyce i informatyce (ang. *computer science*) rekurencja polega na **wywoływaniu funkcji** (procedury) **przez samą siebie**.
2. Rekurencja może również polegać na tym, że złożony typ danych (struktura) T określa jedną ze swoich składowych jako odwołanie (referencję, wskaźnik) do bytu (obiektu, wartości) o typie T. Mamy tutaj do czynienia z **rekurencyjnymi typami danych**.

Silnia

Funkcja określona na zbiorze liczb naturalnych N , zdefiniowana następująco:

$$n! = 1 \quad \text{dla } n = 0$$


$$n! = n (n - 1)! \quad \text{dla } n > 0$$

Wykorzystywana szeroko w m. in. kombinatoryce (i innych dyscyplinach wchodzących w skład dziedziny dyskretnej) oraz teorii liczb.

Silnia – implementacja

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

```
unsigned int silnia(unsigned int n)
{
    if (n == 0) {
        return 1;
    }
    else {
        return n * silnia (n - 1);
    }
}
```



```
int main(void)
{
    unsigned int x, n;
    n = 0;
    x = silnia(n);
    printf("Silnia z %d wynosi %d\n", n, x);


    n = 1;
    x = silnia(n);
    printf("Silnia z %d wynosi %d\n", n, x);

    return 0;
}
```

Silnia – implementacja – narzędzia

```
void wypisz_silnie(unsigned int n)
{
    printf("Silnia z %d wynosi %d\n", n, silnia(n));
}
```

```
void wypisz_silnie_z_zakresu(unsigned int start,
                             unsigned int koniec)
{
    if(start <= koniec) {
        wypisz_silnie(start);
        wypisz_silnie_z_zakresu(start+1, koniec);
    }
}
```



```
int main(void)
{
    wypisz_silnie_z_zakresu(0, 10);
    return 0;
}
```

```
Silnia z 0 wynosi 1
Silnia z 1 wynosi 1
Silnia z 2 wynosi 2
Silnia z 3 wynosi 6
Silnia z 4 wynosi 24
Silnia z 5 wynosi 120
Silnia z 6 wynosi 720
Silnia z 7 wynosi 5040
Silnia z 8 wynosi 40320
Silnia z 9 wynosi 362880
Silnia z 10 wynosi 3628800
```

Silnia – jak to działa

`silnia(5) =`

`5 * silnia(4) =`

`5 * 4 * silnia(3) =`

`5 * 4 * 3 * silnia(2) =`

`5 * 4 * 3 * 2 * silnia(1) =`

`5 * 4 * 3 * 2 * 1 * silnia(0) =`

`5 * 4 * 3 * 2 * 1 * 1 = 120`

Powyższe rozumowanie „przeprowadziło” nas od *rekurencyjnej* definicji funkcji $n!$ oraz procedury *silnia* do wersji *jawnej*.

$$n! = \prod_{k=1}^n k \quad \forall n \in \mathbb{N}$$

Zachodzi to przy koniecznym i przyjętym w matematyce założeniu, że iloczyn zera liczb (pusty) jest równy 1.

Iteracja

(łac. *iteratio*) – powtarzanie operacji, szczególnie z każdorazowym przekształceniem w analogiczny sposób.

Procedura

```
void wypisz_silnie_z_zakresu(unsigned int start,  
                             unsigned int koniec) { ... }
```

jest przykładem *algorytmu iteracyjnego*, ponieważ jego celem jest **wielokrotne powtórzenie** operacji wypisywania wartości procedury *silnia* na ekran, dla wartości parametru formalnego *n* należących do podanego zakresu.

Procedura ta jest **zrealizowana** z wykorzystaniem **rekurencji**.

Iteracja

- Wielu programistów milcząco przyjmuje, że *iteracja* jest równoznaczna z realizacją powtarzania operacji z wykorzystaniem *instrukcji skoku (pętli) i zmiany stanu*. Vide L Peter Deutsch.
- Jest to rozumowanie *nieściśle* ! Wielokrotne powtarzanie (iterowanie) może być zrealizowane z użyciem zarówno instrukcji skoku, jak i *rekurencji*.

Z uwagi jednak na rozpowszechnienie tego poglądu, wygodnie jest przyjąć rozpowszechnioną choć nieściłą terminologię. Od tej pory mówiąc „*iteracyjny*” mamy na myśli – *zrealizowany z wykorzystaniem zmiany stanu i instrukcji skoku* (pętli). Chyba, że zostanie to jawnie określone inaczej.

Problemy rekurencji

Są dwa zasadnicze:

1. Rekurencyjna postać funkcji może być mało czytelna dla człowieka nie przyzwyczajonego do myślenia w ten sposób o problemach. Fakt ten może utrudnić analizę algorytmu, ocenę jego poprawności. Bardzo częstym błędem popełnianym przy tworzeniu procedur rekurencyjnych jest pominięcie warunku prowadzącego do zatrzymania procesu obliczeniowego (**warunek „stopu”**) lub jego nieumiejętne użycie. W wyniku tego proces obliczeniowy może „wpaść” w nieskończoną iterację. Objawi się to komunikatem **STACK-OVERFLOW-ERROR** z uwagi na zjawisko opisane w punkcie 2.
2. **PRYMITYWNA** realizacja rekurencji w obecnie stosowanych językach programowania – oparta na **skończonym stosie** – powoduje, że zwykle mamy do czynienia z ograniczeniem głębokości drzewa wywołań rekurencyjnych do kilkuset lub kilku tysięcy. Iteracja zrealizowana **poprawnie** z wykorzystaniem rekurencji jest więc narażona na **STACK-OVERFLOW-ERROR** dla nietrywialnych rozmiarów problemu, który adresuje.

Dojrzała realizacja rekurencji

Dojrzałe języki programowania oferują przeźroczystą (ang. *transparent*) konwersję wywołań rekurencyjnych na postać, która nie wykorzystuje fatalnego stosu.

Technika, o której mowa, stosowana jest na etapie kompilacji i nosi nazwę **optymalizacji rekurencji krańcowej** (dosł. „ogonowej” z ang. *tail recursion optimization*). Języki, które tę technikę oferują:

§ Dialekty języka **LISP**, w szczególności

- Scheme – zawsze, domyślnie
- Common Lisp – przy założeniu niektórych ustawień optymalizacji
- Clojure – poprzez jawne wywołanie (**recur** ...) pomimo braku tej techniki w maszynie wirtualnej Javy (JVM)

§ Niektóre „pochodne” języka ML, np. **Ocaml**

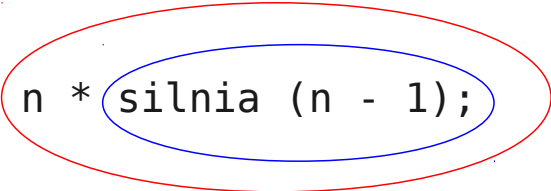
Doskonały język funkcyjny **Haskell** jest językiem wyposażonym w techniki **odraczania wykonania** (ewaluacji) **wyrażeń** (ang. *lazy evaluation*). Z tego powodu opisywane problemy z rekurencją w ogóle go nie dotyczą, bo stos nie jest wykorzystywany w sposób tradycyjny. W **Clojure** późne wykonanie jest również dostępne, ale jego zastosowanie musi być jawnie zadekretowane, poprzez użycie odpowiednich form składniowych.

Optymalizacja rekurencji krańcowej

Warunkiem koniecznym zastosowania tej techniki przez kompilator jest taka postać procedury rekurencyjnej, w której **wartość wywołania rekurencyjnego** jest jednocześnie **wartością procedury**. Innymi słowy – po obliczeniu wartości wywołania rekurencyjnego wartość ta nie jest dalej używana do obliczeń, lecz jest natychmiast zwracana jako wartość procedury. Stąd nazwa „krańcowa” - nic więcej nie robimy, **zejście (wywołanie) rekurencyjne** jest **ostatnim etapem** wyznaczania wartości wywołania procedury.

Silnia w postaci

```
unsigned int silnia(unsigned int n)
{
    if (n == 0) {
        return 1;
    }
    else {
        return n * silnia (n - 1);
    }
}
```



nie posiada rekurencji krańcowej. Wartość wyrażenia `silnia (n - 1)` jest użyta do wyznaczenia wartości wyrażenia `n * silnia (n - 1)`.

Silnia z rekurencją krańcową

```
unsigned int silnia(unsigned int n,  
                    unsigned int wartosc)  
{  
    if(n == 0) {  
        return wartosc;  
    }  
    else {  
        return silnia(n-1, wartosc * n);  
    }  
}
```

`int x = silnia(10, 1);` ==> 3628800 (10!), proszę zwrócić uwagę na dodatkowy konieczny parametr równy zawsze 1.

Teraz wartość wywołania rekurencyjnego `silnia(n-1, wartosc * n)` jest wartością wywołania procedury `silnia`. Wywołanie `silnia(n-1, wartosc * n)` jest krańcowe, po nim nie ma już żadnych innych obliczeń, jego wartość jest natychmiast zwracana.

Kompilator języka C (GCC) nie posiada optymalizacji rekurencji krańcowej. Technika ta była dotychczas odrzucana przez twórców kompilatorów języka C z uwagi na brak automatycznego zarządzania pamięcią (szczególnie przestrzenią stosu) oraz ogólny, niskopoziomowy charakter języka.

Co zrobiłby dojrzały kompilator GCC ?

```
unsigned int silnia(unsigned int n,  
                    unsigned int wartosc)  
{  
    unsigned int tmp;  
    START:  
    if(n == 0) {  
        return wartosc;  
    }  
    else {  
        tmp = n;  
        n = n - 1;  
        wartosc = wartosc * tmp;  
        goto START;  
    }  
}
```

Postać ta opiera się na zastosowaniu instrukcji skoku (*goto*) oraz na *modyfikacji stanu* procesu obliczeniowego przy użyciu *operatora przypisania* (=) przykładanego do symboli *zmiennych* *n*, *wartosc* i *tmp*.

GOTO Considered Harmful ...

„Nieostrożne użycie polecenia goto skutkuje natychmiastowymi konsekwencjami; szalenie trudne staje się ustalenie zbioru koordynatów, przy użyciu których można byłoby określić kierunek zmian procesu (obliczeniowego). [...] Polecenie goto jest zbyt prymitywne, jest zaproszeniem do bałaganienia w kodzie programu.”

Edsger Dijkstra (March 1968). "Go To Statement Considered Harmful". Communications of the ACM 11 (3): 147–148. doi:10.1145/362929.362947

```
unsigned int silnia(unsigned int n,  
                    unsigned int wartosc)  
{  
    while(1) {  
        if(n == 0) {  
            return wartosc;  
        }  
        else {  
            wartosc = wartosc * n;  
            n = n - 1;  
        }  
    }  
}
```

Edsger Dijkstra (ur. 11 maja 1930 w Rotterdamie, zm. 6 sierpnia 2002 w Neunen). Matematyk, jeden z pionierów informatyki, twórca algorytmu znajdującego najkrótszą ścieżkę pomiędzy węzłami grafu (**algorytm Dijkstry**), wynalazca **semafora** w prog. współbieżnym, laureat Nagrody Turinga (1972).

Ciąg Fibonacciego

Ciąg zdefiniowany dla liczb naturalnych, jak następuje:

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_n = fib_{n-1} + fib_{n-2} \text{ dla } n > 1$$

Fibonacci używał wyrazów tego ciągu do opisu wzrostu liczebności populacji królików (dokładnie – ilości par królików w kolejnych miesiącach) przy pewnych wyidealizowanych założeniach (m.in. nieśmiertelności królików).

Ciąg ten był znany już w starożytnych Indiach jeszcze przed Chrystusem, a więc na długo zanim „zagościł” on w matematycznych rozważaniach umysłów Zachodu.

Fibonacci (Leonardo z Pizy; ur. około 1175 r. - zm. 1250 r.) - włoski matematyk. .

Na podstawie: <http://pl.wikipedia.org/wiki/Fibonacci>
http://pl.wikipedia.org/wiki/Ciąg_Fibonacciego

Zadanie

- a. Zaimplementować rekurencyjną procedurę ***fib***, która wywołana z argumentem n typu ***int*** zwróci n -ty wyraz ciągu Fibonacciego.
- b. Zaimplementować rekurencyjnie proces wypisywania na ekran wyrazów ciągu Fibonacciego dla n z zakresu *start ... end* (jak w przypadku procedury *silnia*).
- c. Doprowadzić rekurencyjną procedurę ***fib*** do postaci z rekurencją krańcową.
- d. Przedstawić wersję procedury ***fib*** z wykorzystaniem instrukcji skoku lub pętli.

Ternary conditional

W językach funkcyjnych, np. w Lispie obowiązuje zasada, że każde zdanie posiada wartość. Np.

```
(if (parzysta x)
    (inc x)
    (dec x))
```

ma wartość będącą wynikiem inkrementacji (zwiększenia o 1) wartości symbolu *x*, kiedy jest ona liczbą parzystą, i dekrementacji (zmniejszenia o 1), gdy jest ona nieparzysta.

Zdanie takie może być użyte w innych wyrażeniach, np.

```
(+ 256 (if (parzysta x)
            (inc x)
            (dec x)))
```

Dodajemy tutaj wartość całego wyrażenia (if ...) do liczby 256. Spróbujcie to wykonać w C z wykorzystaniem *bezwartościowej instrukcji*

```
if ( ) { ... } else { ... } ...
```

Ternary conditional

C posiada jednak konstrukcję składniową, która jest echem podejścia do konstrukcji języków programowania opartego na wartościowaniu. Jest nim trójargumentowe wyrażenie warunkowe postaci

warunek ? **wartość-gdy-prawda** : **wartość-gdy-fałsz**

warunek ma takie samo znaczenie, jak odpowiadający mu element instrukcji warunkowej **if** () { ... }. Teraz możemy już zapisać np.

double y = 256 + parzysta(x) ? x + 1 : x - 1;

Zdefiniujmy teraz funkcję, która modeluje bardzo przydatną w matematyce funkcję $| \cdot |$ (wartość bezwzględna).

$|x|$ - implementacja

Matematyczna definicja wartości bezwzględnej z x będącego elementem zbioru liczb rzeczywistych \mathbf{R} :

$$\begin{aligned} |x| &= -x \text{ dla } x < 0 \\ &0 \text{ dla } x = 0 \\ &x \text{ dla } x > 0 \end{aligned}$$

znajduje swoją realizację w postaci procedury:

```
double bezwzgledna(double x)
{
    return x < 0 ? -x : x;
}
```

Inaczej moglibyśmy zapisać oczywiście:

```
double bezwzgledna(double x)
{
    if (x < 0) {
        return -x;
    }
    else {
        return x;
    }
}
```

Pierwiastkowanie metodą Newtona

Pierwiastek **kwadratowy** z liczby rzeczywistej nieujemnej, to funkcja, którą można zdefiniować, jak następuje:

$$\sqrt{x} = \text{takie } y, \text{ że } y \geq 0 \text{ i } y^2 = x \quad \forall_{x \in \mathbb{R} \wedge x \geq 0}$$

Zapis deklaratywny tej funkcji mógłby wyglądać następująco:

```
double pierwiastek(double x)
{
    return taki_ze(y >= 0 && y * y == x);
}
```

Niestety, język C jest przykładem języka imperatywnego, tj. takiego, w którym musimy dokładnie „powiedzieć” maszynie, jak ma dla nas policzyć interesującą nas wartość. Powyższa konstrukcja jest nieprawidłowa.

Dla zainteresowanych: pakiet **Mathematica** posiada mechanizmy deklaratywnego definiowania funkcji jak powyższa. Polecam Państwa uwadze !!!

Pierwiastkowanie metodą Newtona

Użyjemy więc metody wyznaczania pierwiastków funkcji ciągłych, zwanej **metodą Newtona-Raphsona** lub **metodą stycznych**. Zainteresowanych zgłębieniem tej metody (przy okazji pięknego przedmiotu „Metody numeryczne”) odsyłam do:

http://pl.wikipedia.org/wiki/Metoda_Newtona

My poprzestaniemy tutaj na stwierdzeniu, będącym wnioskiem ze wskazanych rozważań, że ilekroć mamy przybliżoną wartość y pierwiastka kwadratowego liczby x , możemy w prosty sposób uzyskać lepsze przybliżenie pierwiastka: licząc średnią arytmetyczną y i x/y .

Uwaga. Ten algorytm pierwiastkowania został opracowany przez **Herona z Aleksandrii** w I w. n.e. Kilkanaście stuleci później okazało się, że istnieje uogólnienie tej metody, zwane właśnie **metodą stycznych**. Abstrahowanie to naprawdę JEST wychodzenie od przypadków szczególnych w kierunku uogólnień. Obydwa rozwiązania wymagały jednak istnienia prawdziwego geniuszu dwóch wybitnych matematyków.

Liczymy pierwiastek kwadratowy z 2

<i>Wartość przybliżona y</i>	<i>Iloraz x / y</i>	<i>Średnia z x / y i y</i>
1	$2 / 1 = 2$	$(2 + 1) / 2 = 1,5$
1,5	$2 / 1,5 = 1,3333$	$(1,3333 + 1,5) / 2 = 1,4167$
1,4167	$2 / 1,4167 = 1,4118$	$(1,4118 + 1,4167) / 2 = 1,4142$
1,4142

Wzór Herona/Newtona - implementacja

```
double pierwiastek(double przyblizenie, double x)
{
    if (wystarczajace(przyblizenie, x)) {
        return przyblizenie;
    }
    else {
        return pierwiastek(ulepsz(przyblizenie, x),
                           x);
    }
}
```

Proszę zwrócić uwagę na fakt, że procedura powyższa posiada *rekurencję krańcową*.

Wzór Herona/Newtona - implementacja

```
double srednia(double a, double b)
{
    return (a + b) / 2;
}
```

```
double ulepsz(double przyblizenie, double x)
{
    return srednia(przyblizenie, x / przyblizenie);
}
```

```
typedef int BOOL;
```

```
BOOL wystarczajace(double przyblizenie, double x)
{
    return bezwzgledna(przyblizenie * przyblizenie - x) < 0.001;
}
```

- TUTAJ MIEŚCI SIĘ WARUNEK ZATRZYMANIA ALGORYTMU !!!

Wzór Herona/Newtona - użycie

```
int main(void)
{
    printf("Pierwiastek z 2 to %f\n", pierwiastek(1.0, 2.0));
    printf("Pierwiastek z 2 do kwadratu to %f\n",
           pierwiastek(1.0, 2.0) * pierwiastek(1.0, 2.0));
    return 0;
}
```

daje w wyniku na konsoli:

```
Pierwiastek z 2 to 1.414216
Pierwiastek z 2 do kwadratu to 2.000006
```

Zadanie

a. Zmodyfikować procedurę ***pierwiastek*** (oraz pozostałe, jeśli jest potrzeba) tak, aby możliwe było przekazanie wartości wyznaczającej moment zatrzymania algorytmu – dodatkowy parametr ***epsilon***:

```
pierwiastek(1.0, 2.0, 0.0001);
```

b. Zmodyfikować procedurę ***pierwiastek*** (oraz pozostałe, jeśli jest potrzeba) tak, aby możliwe było przekazanie wartości wyznaczającej moment zatrzymania algorytmu – dodatkowy parametr ***n*** określający maksymalną liczbę kroków do wykonania przez algorytm:

```
pierwiastek(1.0, 2.0, 30);
```

c. Doprowadzić rekurencyjną procedurę ***pierwiastek*** do postaci wykorzystującej instrukcję skoku lub pętli.

Kwadrat podanej liczby

Możliwe jest zdefiniowanie wersji tej procedury dla kilku typów parametrów. Na przykład dla typu całkowitoliczbowego:

```
unsigned int kwadrat_naturalnej(unsigned int x)
{
    return x * x;
}
```

lub dla liczb zmiennoprzecinkowych

```
double kwadrat_rzeczywistej(double x)
{
    return x * x;
}
```

Możliwe jest zdefiniowanie tylu wersji procedury, ile mamy interesujących nas typów. Ważne jest tylko, aby za każdym razem nadawać procedurze inną nazwę; w języku C nie ma przeciążania nazw procedur (w odróżnieniu od C++, Javy).

Użyjmy preprocesora

W języku C **preprocesor** jest znakomitym mechanizmem pozwalającym na generowanie kodu w trakcie kompilacji, a dokładnie w fazie bezpośrednio poprzedzającej analizę zdań – zwanej **preprocesingiem**.

Zdefiniujmy makrodefinicję preprocesora:

```
#define KWADRAT(nazwa_funkcji,          \
                typ_parametru )         \
    typ_parametru nazwa_funkcji(typ_parametru x) { \
        return x * x;                  \
    }                                   \
```

Teraz możemy napisać po prostu:

```
KWADRAT(kwadrat_rzeczywistej, double)
KWADRAT(kwadrat_naturalnej, unsigned int)
```

a to z kolei w trakcie rozwinięcia uzyska formy:

```
double kwadrat_rzeczywistej(double x) { return x * x; }
unsigned int kwadrat_naturalnej(unsigned int x) { return x * x; }
```


Jeszcze na temat makr

W języku C symbol \ (ang. *backslash*) użyty poza literałem łańcuchowym (ang. *string*) oznacza kontynuację linii. Makrodefinicje mają to do siebie, że „wymagają” zdefiniowania w jednej linii. Po prostu taka jest natura preprocesora. Dlatego makro

```
#define KWADRAT(nazwa_funkcji,          \
                    typ_parametru )      \
    typ_parametru nazwa_funkcji(typ_parametru x) { \
        return x * x;                    \
    }
```

użyte w wyrażeniu:

```
KWADRAT(kwadrat_rzeczywistej, double)
```

rozwija się do jednolinijkowej postaci:

```
double kwadrat_rzeczywistej(double x) { return x * x; }
```

Nie mówiąc już o tym, że przy definiowaniu makr po prostu **musimy** użyć symbolu \, jeśli chcemy rozłożyć kod makra na wiele linii w celu np. podniesienia czytelności.

Jeszcze na temat makr

Dojrzałe języki programowania, np. Common Lisp, Clojure posiadają potężny mechanizm definiowania makr, które tam w istocie są **procedurami generującymi kod** na etapie kompilacji (dokładnie – na etapie makro-rozwinięć, ang. *macro-expansion*).

W języku C mechanizm ten posiada jedynie szcątkową funkcjonalność w zestawieniu z Lispami. Pomimo tego może się przydać do generowania nieraz obszernych fragmentów „uciążliwego”, a niezbyt skomplikowanego kodu.

Możemy więc mówić, że **makro** jest **uogólnieniem**, abstrakcją, pewnego **zbioru konstrukcji składniowych** języka.

- UWAGA. Nieumiejętne, nieostrożne użycie makrodefinicji może prowadzić do powstania niezwykle trudnych do odnalezienia błędów w programie.

Jakie mogą być problemy z makrami ?

Wracając do naszego przykładu z podnoszeniem do kwadratu ...
Możemy się poczuć skuszeni możliwością zdefiniowania makra:

```
#define KWADRAT(x) (x * x)
```

Jest ono faktycznie dość uniwersalne, bo „załatwia” problem wielokrotnego definiowania tego samego kodu dla różnych typów argumentu i pod różnymi nazwami.

Dla prostych wartości działa, np.

```
printf(„%d”, KWADRAT(4));
```

 daje w wyniku 16

Ale zobaczmy, co będzie, gdy

```
printf(„%d”, KWADRAT(4 + 1));
```

Niestety, otrzymujemy 9 zamiast 25. Jest tak dlatego, że rozwinięcie ma postać:

```
printf(„%d”, (4 + 1 * 4 + 1));
```

Rozwiązanie ...

... polega na pewnej „niewinnej” przeróbce naszej definicji:

```
#define KWADRAT(x) ((x) * (x))
```

Teraz jest ok. Można sprawdzić, że

```
printf(„%d”, KWADRAT(4 + 1));
```

zostaje rozwinięte do:

```
printf(„%d”, ((4 + 1) * (4 + 1)));
```

W ten sposób, umiejętnie używając () do zabezpieczenia naszych wyrażeń przed zgubnym wpływem niejednoznaczności gramatyk możemy pisać całkiem efektywne makrodefinicje. Z tym tylko, że ...

Wpływ operatora przypisania i tutaj jest zgubny

W języku C wyrażenie $i += 1$ jest tożsame z zapisem $i = i + 1$ dla pewnego symbolu i . Teraz spróbujemy posłużyć się naszym makrem w sposób następujący:

```
int i = 4;  
printf(„%d”, KWADRAT(i += 1));
```

Oczekujemy, że wypisaną na ekranie wartością będzie 25. Tymczasem ...

```
printf(„%d”, ((i += 1) * (i += 1)));
```

Uzyskujemy wartość 36. Okazuje się, że wartością wyrażenia składowego $i += 1$ stojącego po obydwu stronach operatora mnożenia $*$ jest wartość symbolu i w chwili ewaluacji operacji mnożenia, tj. już po dwukrotnym zinkrementowaniu i . Mylące, prawda ?

Okazuje się więc, że makrodefinicje to potężne narzędzie, które używane nieostrożnie może nas mocno zranić, a użyte w zestawieniu z operatorem przypisania – naszym śmiertelnym wrogiem – staje się prawdziwą zmorą. Tak, że niektórzy wolą w ogóle z niego nie korzystać. A może należałoby wysnuć kilka refleksji na temat „niewinnego” symbolu $=$... ?

Potęgi o stopniach wyższych niż 2

Skoro w kwestii podnoszenia do kwadratu powiedzieliśmy sobie немало, możemy teraz pokusić się o zdefiniowanie procedury liczącej wartości wyrażeń postaci a^n . Załóżmy od tej pory, że a oraz n są liczbami naturalnymi

typedef unsigned long long Naturalna;

i a jest różne od 0. Można zdefiniować a^n jako:

$$\begin{aligned} a^n &= 1 && \text{dla } n = 0 \\ &a a^{n-1} && \text{dla } n > 0 \end{aligned}$$

8 bajtów może „unieść dla nas”
wartość $2^{64} - 1 =$
18446744073709551615

Procedura odpowiadająca tej definicji ma postać:

```
Naturalna potega(Naturalna a, Naturalna n) {  
    return n == 0 ? 1 : a * potega(a, n-1);  
}
```

W miejscu wywołania wstawić 1.

W formie z rekurencją krańcową:

```
Naturalna potega(Naturalna a, Naturalna n, Naturalna wartosc) {  
    return n == 0 ? wartosc : potega(a, n-1, wartosc * a)  
}
```

Czy można coś przyspieszyć ?

Dotychczasowa wersja procedury **potega** wymaga n kroków do wykonania. Jest to dość optymistyczne, ale da się szybciej.

Spróbujmy np. policzyć $a^8 = (a \ a \ (a \ (a \ (a \ (a \ a))))$). Zapiszmy to raczej jako:

$a^8 = (((a \ a) \ (a \ a)) \ ((a \ a) \ (a \ a)))$, w takim razie

$a^8 = ((a^2 \ a^2) \ (a^2 \ a^2))$ i

$a^8 = a^4 \ a^4$

De facto potrzebujemy więc tylko 3 mnożeń:

$a^2 = a \ a$

$a^4 = a^2 \ a^2$

$a^8 = a^4 \ a^4$

i ogólnie dla n parzystych $a^n = (a^{n/2})^2$.

Dla nieparzystych n przyjmujemy $a^n = a \ a^{n-1}$.

Dla $n = 0$ mamy 1 ...

... i to dopełnia naszej definicji.

Na podstawie: „Struktura i interpretacja programów komputerowych” H. Abelson, G.J. Sussmann

Implementacja w języku C

```
BOOL parzysta(Naturalna x)
{
    return x % 2 == 0;
}
```

```
Naturalna potega(Naturalna a, Naturalna n)
{
    if(n == 0)
        return 1;
    else if(parzysta(n))
        return kwadrat_naturalnej(potega(a, n / 2));
    else
        return a * potega(a, n-1);
}
```

Powyższa procedura nie posiada rekurencji krańcowej.

Zadanie

- a. Zaimplementować rekurencyjną procedurę szybkiego potęgowania. Uruchomić dla kilku wartości a i n .
- b. Zaproponować wersję powyższej procedury wykorzystującą jedynie instrukcje warunkowe oraz instrukcję skoku (pętlę `while`).

Rozważania o szybkości działania algorytmu

W języku C bardzo trudno jest zbadać rzeczywisty zysk z użycia szybszego algorytmu. Nie mamy typu danych zdolnego do reprezentowania liczby np. 2^{100000} . A interesujący nas zysk można obserwować dla wartości n tego rzędu. W Clojure:

```
(defn expt
  [a n]
  (reduce * (repeat n a)))

(defn square [x] (* x x))

(defn fast-expt
  [a n]
  (cond (zero? n) 1
        (even? n) (square (fast-expt a (/ n 2)))
        :else (* a (fast-expt a (dec n)))))
```

Iteraz:

```
(time (= (fast-expt 2 100000) (fast-expt 2 100000)))
"Elapsed time: 64.339332 msecs"

(time (= (expt 2 100000) (expt 2 100000)))
"Elapsed time: 9256.313825 msecs"
```

Problem komiwojażera

Znany również pod nazwą TSP (ang. *Travelling Salesman Problem*).

Jacek jest przedstawicielem handlowym w branży komputerowej (sprzedawcą), którego rejon działania obejmuje 5 miejscowości w centralnej Polsce. Jego pracodawca zwróci mu jedynie 50 procent całkowitego kosztu jego służbowych przejazdów. Jacek stoi przed problemem wyboru trasy podróży pomiędzy wszystkimi dwudziestoma miejscowościami, rozpoczynając i kończąc na miejscowości początkowej, tak, aby koszty jego podróżowania były jak najmniejsze.

Jest to klasyczny przykład problemu komiwojażera, zagadnienia z teorii grafów polegające na znalezieniu minimalnego **cyklu Hamiltona** w grafie.

Cykl Hamiltona – cykl w grafie, w którym każdy wierzchołek (węzeł) grafu występuje dokładnie jeden raz.

Cykl – zamknięta ścieżka w grafie tzn. taka, której koniec (ostatni wierzchołek) jest identyczny z początkiem.

Sformułowanie problemu

Aby zrozumieć problem musimy go precyzyjnie sformułować. Warunek ten nie jest (niestety) wystarczający do zrozumienia problemu, lecz jest absolutnie konieczny. Precyzyjne sformułowanie problemu jest zwykle kwestią *zadania właściwych pytań*:

- Czy rozumiem słownictwo użyte w pierwotnym opisie problemu ?
- Jaka informacja o problemie została nam podana ?
- Czego naprawdę poszukuję, co chcę znaleźć ?
- Jak rozpoznam rozwiązanie ?
- Jakich informacji brakuje i czy jakiekolwiek z tych informacji będą użyteczne ?
- Czy któraś z podanych informacji jest bezużyteczna ?
- Czy i jakie założenia zostały przyjęte ?

...

TSP – sformułowanie problemu

Co jest podane ? Lista miast w rejonie Jacka i skojarzona z nią macierz kosztów, tj. tablica dwuwymiarowa o wyrazach $C_{i,j}$ równych kosztowi przejazdu z miasta i do miasta j . W naszym przykładzie macierz kosztów ma 5 wierszy i 5 kolumn.

Co chcemy znaleźć ? Poszukujemy listy miast, która zawiera dokładnie jeden raz każde miasto, poza pierwszym (bazowym) miastem, które znajdzie się również na pozycji ostatniej. Kolejność miast na liście odzwierciedla kolejność, w której Jacek ma odbyć swoją podróż poprzez miasta. Suma kosztów podróży pomiędzy każdą kolejną parą miast należących do listy jest całościowym kosztem trasy reprezentowanej przez listę.

- **Konkluzja – rozwiążemy problem Jacka dostarczając mu listę miast o najmniejszym koszcie całościowym.**

Skonstruowanie modelu matematycznego

Automatyzacja tego procesu praktycznie nie jest możliwa, bo niemożliwe jest podanie zbioru reguł automatyzujących ten etap. Większość problemów wymaga indywidualnego podejścia. Modelowanie jest bardziej sztuką niż nauką i prawdopodobnie takim pozostanie.

Wybierając lub opracowując model możemy sobie jednak pomóc odpowiadając na dwa pytania:

1. Które ze znanych struktur matematycznych wydają się najlepiej pasować do problemu ?
2. *Czy istnieją inne, rozwiązane już problemy, które przypominają nasz ?*

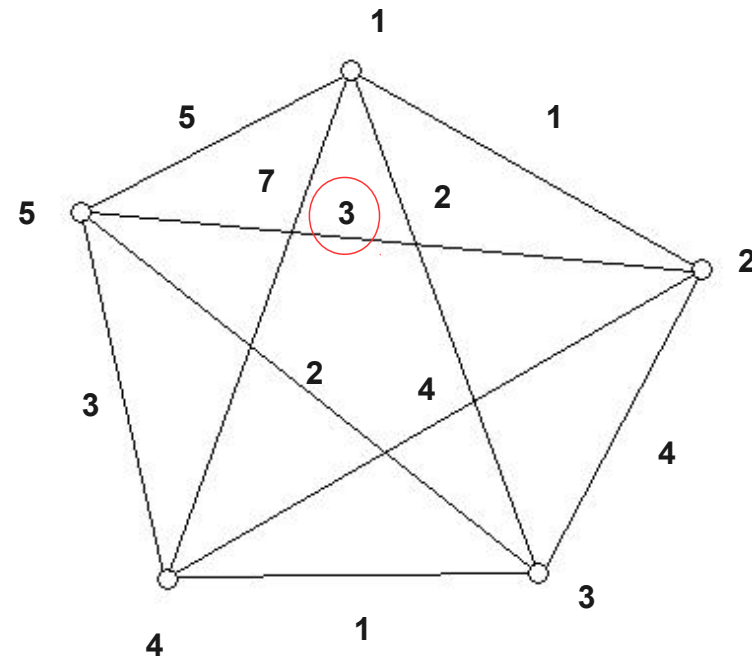
- Bardzo ważny etap w całym procesie znajdowania rozwiązania.
- Wybór właściwego modelu ma poważny i konkretny wpływ na pozostałe fazy procesu.

TSP – model matematyczny

Numer
miasta

	1	2	3	4	5
1	—	1	2	7	5
2	1	—	4	4	3
3	2	4	—	1	2
4	7	4	1	—	3
5	5	3	2	3	—

Koszt drogi pomiędzy
miastami 2 i 5



Trasa 1-5-3-4-2-1 ma koszt $5 + 2 + 1 + 4 + 1 = 13$. Czy jest to koszt najmniejszy ?

TSP – projekt algorytmu

Rozpocznijmy od ponumerowania n miast liczbami całkowitymi z zakresu od 1 do N , każdemu z miast przypisując inną wartość. Miasto, z którego rozpoczynamy podróż oznaczmy jako N .

Zauważmy, że każda trasa odpowiada pewnej unikalnej **permutacji** liczb całkowitych $1, 2, \dots, N-1$ (dlaczego nie $1, 2, \dots, N$?) i **permutacja** ta opisuje **unikalną trasę**. Jest to odpowiedniość **jeden-do-jednego**.

Dla podanej permutacji możemy łatwo oznaczyć trasę na modelu grafowym i – równocześnie – obliczyć jego **koszt**.

Możemy w łatwy sposób rozwiązać problem generując wszystkie permutacje $N-1$ dodatnich liczb całkowitych. Dla każdej permutacji konstruujemy odpowiadającą jej trasę i obliczamy jej **koszt**.

Postępujemy przez listę wszystkich permutacji pamiętając tę trasę, która – jak dotąd – miała **najmniejszy koszt**. Jeśli znajdziemy trasę o mniejszym koszcie – uznajemy ją za trasę do kolejnych porównań.

- Wybór techniki algorytmicznej na etapie projektowania algorytmu, która bardzo często ściśle zależy od wyboru modelu matematycznego, może ogromnie wpłynąć na efektywność rozwiązania.
- Dwa różne algorytmy rozwiązujące poprawnie ten sam problem mogą się znacznie różnić pod względem efektywności.

TSP – algorytm wyczerpujący (ang. *exhaustive*)

Krok 0. [Inicjalizacja]. **set** TRASA := \emptyset ; **and** MIN := INF (ang. *infinity* - nieskończoność).

Krok 1. [Wygeneruj wszystkie permutacje] **for** I := 1 to (N - 1)! **do through** Krok 4 **od**; **and** STOP.

→ Krok 2. [Pobierz kolejną permutację] **set** P := I-ta permutacja liczb (1, ..., N-1).
(potrzebny algorytm generowania permutacji !!!)

→ Krok 3. [Skonstruuj nową trasę] T(P) odpowiadającą permutacji P; **and** oblicz koszt KOSZT(T(P))

(potrzebne dwa kolejne algorytmy składowe – T oraz KOSZT !!!)

→ Krok 4. [Porównaj] **if** KOSZT(T(P)) < MIN **then set** TRASA := T(P); **and** MIN := KOSZT(T(P)) **fi**.

Naturalną tendencją panującą wśród programistów jest spędzanie jak najmniejszej ilości czasu na projektowaniu algorytmu. Interesuje nas przede wszystkim kodowanie. Ta przemożna chęć powinna być powstrzymana !

Faza projektowania powinna być przeprowadzona z wielką rozważą. Powinniśmy również rozważyć 2 fazy ją poprzedzające i 1 po niej następująca.

Poprawność algorytmu

Dowodzenie lub upewnienie się, że algorytm jest poprawny, jest jednym z najtrudniejszych i często najnudniejszych kroków w całym procesie projektowania algorytmu.

Możliwe podejście (najłatwiejsze?): uruchomienie procedury dla różnych przypadków testowych (*test-case*). Jeśli odpowiedzi generowane przez program mogą być porównane z „ręcznymi” obliczeniami lub wartościami znanymi, jesteśmy skuszeni do konkludowania, że program „działa”. Technika ta jednak bardzo rzadko usuwa wszelkie wątpliwości w stosunku do twierdzenia, że nie istnieje przypadek, dla którego program zadziała błędnie (lub w ogóle nie zadziała).

Zaoferujmy ogólny przepis na dowodzenie poprawności algorytmu:

1. Przypuśćmy, że algorytm jest wyrażony poprzez serię kroków, powiedzmy od 0 do m . Spróbuj zaproponować pewne uzasadnienie (usprawiedliwienie) **każdego** kroku. W szczególności może to wymagać posłużenia się *lematem* dot. warunków istniejących przed i po wykonaniu kroku.

2. Następnie spróbuj udowodnić, że algorytm się zakończy. W ten sposób przeegzaminowaliśmy **wszystkie** stosowne dane wejściowe i wyprodukowaliśmy **wszystkie** możliwe dane wyjściowe.

Poprawność algorytmu ETS

Algorytm ETS jest tak prosty, że jego poprawność jest łatwa do udowodnienia. Ponieważ analizujemy każdą możliwą trasę, trasa o minimalnym koszcie również musi być przeanalizowana w jednym z kroków. Kiedy zostanie „zauważona”, pozostanie zapamiętana przez algorytm. Nigdy nie zostanie porzucona, ponieważ mogłoby się to wydarzyć jedynie w sytuacji, gdyby istniała trasa o mniejszym koszcie, co nie jest z kolei prawdą. Algorytm musi się zakończyć, ponieważ istnieje skończona ilość tras do przeanalizowania.

Przedstawiona tu metoda jest zwyczajowo nazywana *dowodem przez wyczerpanie* (ang. „proof by exhaustion”). Jest to technika najbardziej „surowa”, „brutalna”.

Wypada podkreślić fakt, że poprawność algorytmu nie musi koniecznie sugerować niczego na temat jego stosowalności ani wydajności. Algorytmy „wyczerpujące” rzadko są bardzo dobrymi algorytmami w jakimkolwiek sensie.

Implementacja - trudności

Bardzo często krok algorytmu będzie przedstawiony w formie, która nie jest bezpośrednio przekładalna na kod (np. jeden krok może być tak sformułowany, że wymagać będzie całej podprocedury do jego zaimplementowania).

Zanim rozpoczniemy pisać kod, niejednokrotnie musimy zaprojektować cały system struktur danych w pamięci komputera do reprezentowania ważnych aspektów używanego modelu matematycznego. Przykład: konieczność posługiwania się bibliotekami realizującymi elementarne struktury danych i algorytmy z zakresu algebry liniowej w programach opartych o algebraiczne struktury matematyczne.

Implementacja

W fazie implementacji stawiamy sobie (i poszukujemy odpowiedzi) na pytania:

- Jakie symbole wprowadzić ?
- Jaki ma być ich typ ?
- Czy potrzebujemy tablic, jeśli tak – to jaki ma być ich rozmiar ?
- Czy warto posłużyć się dynamicznymi strukturami danych, np. listami połączonymi, drzewami itd. ?
- Jakich procedur składowych (podprocedur) będziemy potrzebowali ?
- Jakiego języka programowania użyć ?

Konkretna implementacja może w sposób znaczący wpłynąć zarówno na wymagania pamięciowe jak i szybkość działania programu.

WAŻNE ! Jedną rzeczą jest dowodzenie poprawności algorytmu wyrażonego w formie werbalnej (lub w pseudokodzie). Zupełnie **OSOBNĄ RZECZĄ** jest dowiedzenie, że dany **PROGRAM KOMPUTEROWY**, który ma być implementacją algorytmu jest **RÓWNIEŻ POPRAWNY**.

Implementacja algorytmu ETS

```
def ets (citiesMatrix, baseCityIndex):  
    TOUR = [] # Pusta ścieżka  
    INF = sys.maxint + 1  
    MIN = INF # Nieskończoność  
  
    cities = range(0, len(citiesMatrix))  
    cities.remove(baseCityIndex)  
  
    for path in permutations(cities):  
        path.insert(0, baseCityIndex)  
        path.append(baseCityIndex)  
  
        cost = tourCost(citiesMatrix, path)  
  
        if cost < MIN:  
            TOUR = path  
            MIN = cost  
  
    return (TOUR, MIN)
```

Przykład użycia

```
citiesMatrix = [[-1, 1, 2, 7, 5],  
                [ 1, -1, 4, 4, 3],  
                [ 2, 4, -1, 1, 2],  
                [ 7, 4, 1, -1, 3],  
                [ 5, 3, 2, 3, -1]]
```

```
tour, min = ets(citiesMatrix, 0)
```

- Implementacja algorytmu ETS w języku programowania Python

Dlaczego analizujemy algorytmy ?

Z praktycznego punktu widzenia potrzebujemy oszacowania lub ograniczenia z góry ilości pamięci, przestrzeni dyskowej lub czasu wykonania, których to zasobów nasz algorytm będzie potrzebował do udanego przetworzenia konkretnych danych wejściowych. Dobra analiza pozwala znaleźć tzw. „wąskie gardła”, tj. sekcje w naszym programie, w których „spędzane” jest najwięcej czasu.

Powody teoretyczne. Potrzebny jest ilościowy standard porównywania dwóch różnych algorytmów, z których każdy realizuje ten sam problem. „Słabszy” algorytm powinien zostać ulepszony lub odrzucony. Pożądane byłoby istnienie mechanizmu wyboru najbardziej wydajnych algorytmów. Wydawanie klarownych sądów o względnej wydajności algorytmów czasami nie jest w ogóle możliwe (jeden działa lepiej, kiedy uśrednimy wyniki działania dla różnych przypadków testowych, inny – lepiej dla konkretnych danych wejściowych).

Jest również niezmiernie ważne, aby wypracować absolutny standard definiowania jakości algorytmów w ogóle. Kiedy problem jest optymalnie rozwiązany ? Tj. kiedy mamy do czynienia z algorytmem, który jest tak dobry, że **nie jest możliwe** stworzenie innego, znacząco lepszego ?

Dygresja o znaczeniu efektywności



- Wbrew obiegowym opiniom (formułowanym na bazie słusznego zresztą prawa Moore'a) czas procesora i pamięć komputera są stosunkowo deficytowymi (i kosztownymi) zasobami.
- Zadziwiające jest, jak wielu menadżerów, projektantów i programistów z mozółem odkrywa, że ich system po prostu nie może obsłużyć podanych danych wejściowych w czasie poniżej dosłownie kilku dni roboczych.
- Lepiej byłoby przewidywać takie rzeczy przy użyciu kartki i ołówka w celu zapobieżenia katastrofalnym zachowaniom.
- Jest to wręcz niezbędne i absolutnie wymagane w przypadku tzw. *systemów krytycznych*.

Złożoność obliczeniowa

Niech **A** oznacza algorytm rozwiązujący konkretną *klasę problemów*. Niech **n** będzie miarą wielkości konkretnego problemu należącego do tej klasy (**n** może być np. ilością wierzchołków grafu, rozmiarem tablicy itd.).

Określmy funkcję $f_A(n)$ jako funkcję roboczą, zwracającą górne ograniczenie liczby podstawowych operacji (dodawania, porównań itd.), które algorytm **A** musi wykonać w celu rozwiązania problemu o rozmiarze **n**. Używamy następującego kryterium oceny *jakości* algorytmu. Algorytm **A** jest **wielomianiowy** jeśli $f_A(n)$ rośnie nie szybciej niż wielomian zmiennej **n**. W przeciwnym razie o algorytmie **A** mówimy, że jest **wykładniczy**.

Funkcja $f(n)$ jest zdefiniowana jako $O[g(n)]$ i mówi się o niej, że jest rzędu $g(n)$ dla dużych **n** jeśli:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \text{const} \neq 0$$

Zapisujemy to jako $f(n) = O[g(n)]$

Złożoność obliczeniowa

Funkcja $h(n)$ jest określana jako $o[z(n)]$ dla dużych n , jeśli:

$$\lim_{n \rightarrow \infty} \frac{h(n)}{z(n)} = 0$$

Powyższe symbole są często wymawiane jako „duże o” i – odpowiednio – „małe o”. Intuicyjnie, jeśli $f(n)$ jest $O[g(n)]$, wówczas te dwie funkcje f i g rosną „podobnie” szybko, gdy n dąży do nieskończoności. Gdy $f(n)$ jest $o[g(n)]$, wówczas g rośnie znacznie szybciej niż f .

Notacja dużego O (wielka literą O, nie zero), zwana również notacją Landaua, to symbolika używana w teorii złożoności obliczeniowej, informatyce i matematyce do opisu asymptotycznego zachowania funkcji. Notacja Landaua opisuje, jak szybko dana funkcja rośnie lub maleje, abstrahując od konkretnej postaci tych zmian.

Źródło: http://pl.wikipedia.org/wiki/Notacja_Landaua

Przykłady złożoności obliczeniowej

(a) Wielomian $f(n) = 2n^5 + 6n^4 + 6n^2 + 18$ jest $O(n^5)$, ponieważ:

$$\lim_{n \rightarrow \infty} \frac{2n^5 + 6n^4 + 6n^2 + 18}{n^5} = 2$$

Jest on również $o(n^{5.1}), o(e^n), o(2^n)$. Faktycznie jest on „małym o” każdej funkcji, która rośnie szybciej niż wielomian stopnia piątego.

(b) Funkcja $f(n) = 2^n$ jest $o(n!)$.

(c) Funkcja $f(n) = 1000\sqrt{n}$ jest $o(n)$.

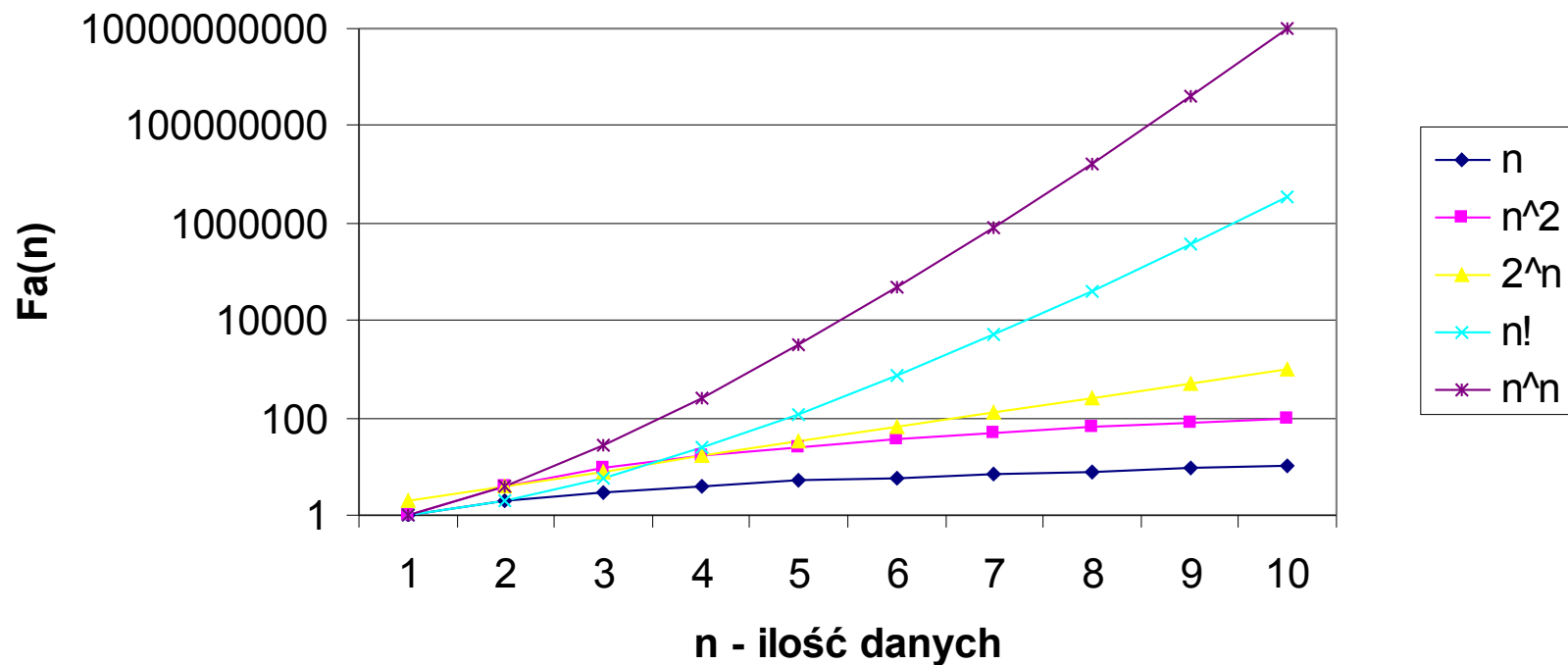
Algorytm jest **wielomianowy**, jeśli $f_A(n) = O[P_k(n)]$ lub $f_A(n) = o[P_k(n)]$ gdzie $P_k(n)$ jest dowolnym wielomianem zmiennej n o ustalonym stopniu k . W przeciwnym wypadku algorytm jest **wykładniczy**.

Trochę arytmetyki

n	n^2	2^n	$n!$	n^n
1	1	2	1	1
2	4	4	2	4
3	9	8	6	27
4	16	16	24	256
5	25	32	120	3125
6	36	64	720	46656
7	49	128	5040	823543
8	64	256	40320	16777216
9	81	512	362880	387420489
10	100	1024	3628800	10000000000

Jeden rysunek jest więcej wart ...

**Zachowanie funkcji określających złożoność algorytmu
- skala logarytmiczna.**



Złożoność obliczeniowa – rozważania

Chcielibyśmy uzyskać tak precyzyjną analizę algorytmów, jak to jest tylko możliwe, tj. chcemy określać funkcję $f_A(n)$ tak precyzyjnie, jak się tylko da.

Szczegółowa informacja o algorytmie może być uzyskana wyłącznie na podstawie specyficznej implementacji. Np. charakterystyka $O(n^5)$ jest zwykle nierozzerwalnie związana z algorytmem, natomiast $2n^5 + 6n^2 + 4$ będąca opisem algorytmu może zostać określona dopiero przy analizie konkretnej implementacji.

Ważne jest również spostrzeżenie, jak złymi algorytmami są algorytmy zachowujące się wykładniczo. Istnieje wiele ważnych problemów kombinatorycznych, dla których znamy obecnie wyłącznie algorytmy wykładnicze (mowa tu o algorytmach dokładnych, wyczerpujących, nie zaś przybliżonych). Problem TSP należy do tego rodzaju problemów.

Problemy tego typu muszą być rozwiązane, ponieważ mają one ważne zastosowania praktyczne. W takiej sytuacji należy zawsze wybierać „mniejsze zło”, czyli taki algorytm, który jest najwydajniejszy. Np. algorytm $O(2^n)$ jest lepszy niż $O(n!)$.

Złożoność obliczeniowa algorytmu ETS

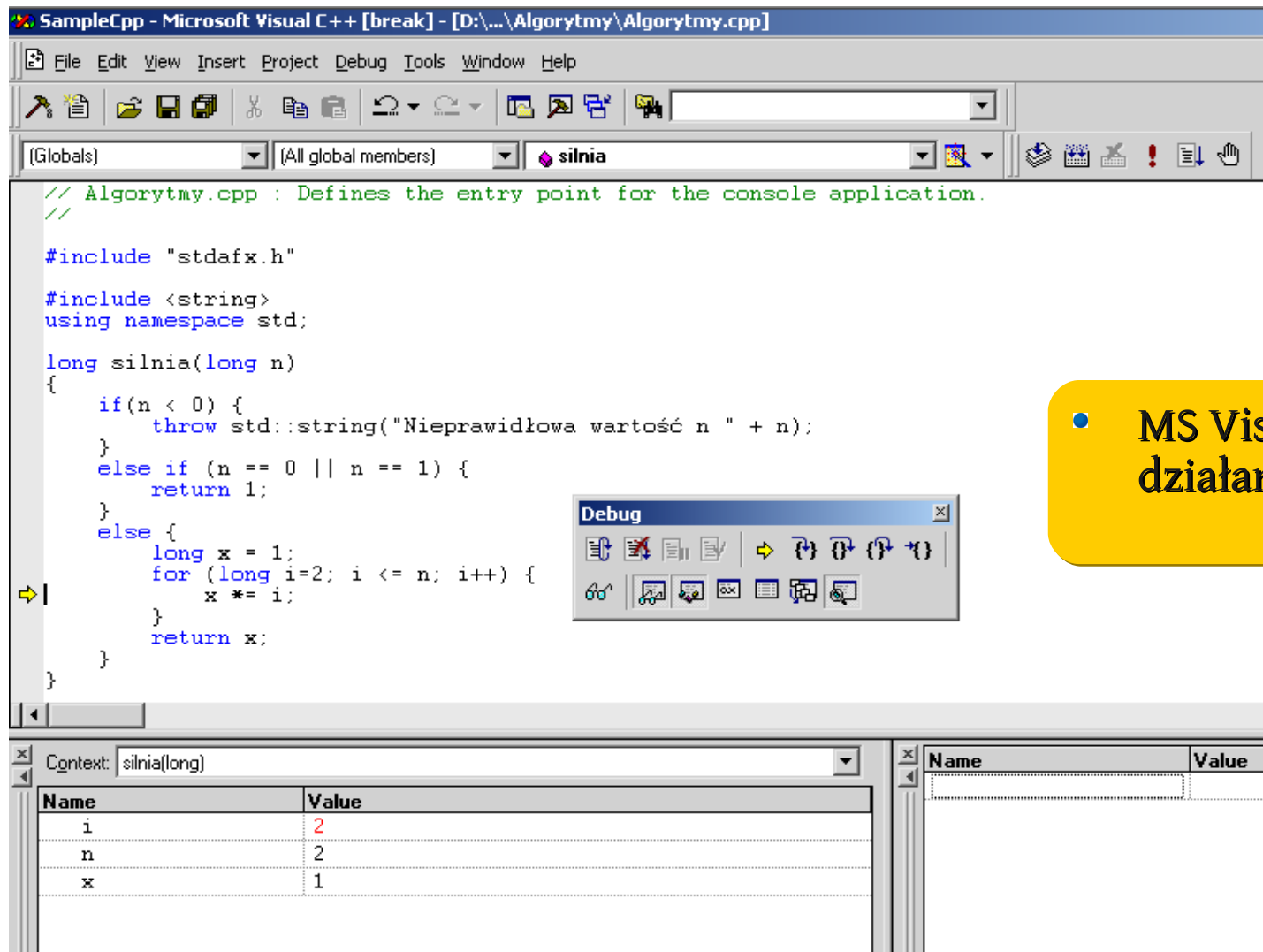
W problemie n miast, algorytm ETS wymaga od nas wyliczenia i iteracyjnego przeanalizowania permutacji $n-1$ pierwszych liczb całkowitych dodatnich. Istnieje $(n-1)!$ takich permutacji. Nawet, jeśli potrzebowalibyśmy jedynie 1 operacji dla każdej z permutacji, ta część algorytmu już miałaby złożoność $O[(n-1)!]$. Kiedy pojedyncza permutacja jest brana pod uwagę, znalezienie odpowiadającej jej trasy i obliczenie jej kosztu ma złożoność $O[n]$. Wobec tego górnym ograniczeniem całkowitego czasu wykonania musi być $O[n!]$.

Założmy, że nasz sprzedawca ma do odwiedzenia 20 miast i że dysponujemy fenomenalnie dobrym algorytmem składowym który generuje kolejną permutację w jednym kroku. Założmy, że dysponujemy komputerem, która wykonuje jeden krok podstawowy algorytmu w czasie 0,0000001 s. Wówczas rozwiązanie problemu TSP przy użyciu algorytmu ETS zajęłoby nieco poniżej 70 stuleci !!!

Testowanie zaimplementowanego algorytmu

- Uruchomienie programu powinno być poprzedzone debugowaniem („odpluskwianiem” – ang. *debugging*).
- Testowanie programu przypomina nieco eksperymentowanie w naukach przyrodniczych. Możemy myśleć o tej fazie, jak o eksperymentalnej weryfikacji, czy program robi dokładnie to, co powinien (lub jak o eksperymentalnym sprawdzeniu granic używalności programu / algorytmu).
- Samo udowodnienie, że algorytm jest poprawny, nie wystarczy. Każdy program powinien zostać intensywnie i kompleksowo przetestowany. Stwarzamy sobie szansę dostrzeżenia niuansów związanych z działaniem programu, których z natury rzeczy nie mogliśmy uwzględnić wcześniej np. wpływ systemu operacyjnego na obliczenia w czasie rzeczywistym.
- Sztuką jest wybranie i przeprowadzenie grupy wystarczająco reprezentatywnych przypadków testowych i dobranie dla nich danych wejściowych dla naszego programu.
- W najlepszym przypadku powinniśmy stworzyć sobie warunki do przetestowania wszystkich segmentów kodu źródłowego w naszym programie. Warunek ten bywa trudny do osiągnięcia w przypadku złożonych systemów.

Debugowanie programu liczącego $n!$

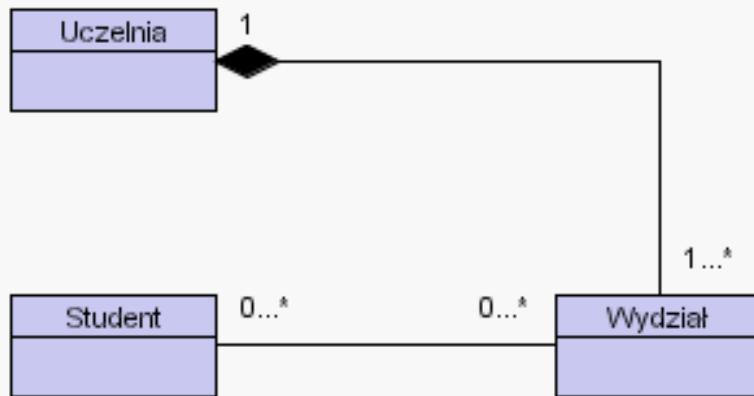


- MS Visual C++ 6.0 w działaniu.

Dokumentowanie

Złota reguła: Dokumentuj swoje programy tak, jak chciałbyś, aby były udokumentowane programy, których kod źródłowy czytasz.

*Rys. Unified Modelling Language (UML)
„w natarciu” - diagram klas.*



Proces dokumentowania powinien być wpleciony we wszystkie fazy tworzenia algorytmu, szczególnie zaś w te, które są związane z projektowaniem i implementacją.

Najlepszym sposobem dokumentowania projektu informatycznego jest pisanie kodu tak, aby był on samo-dokumentujący się. Jest to szczególnie ważne w przypadku programów o nietrywialnych rozmiarach (100 tys. linii i więcej) oraz bibliotek.

Dotknęliśmy zaledwie wierzchołka góry lodowej.

Programowanie strukturalne „z góry na dół” i poprawność programów

Podstawowym celem algorytmu jest *prawidłowe zaimplementowanie* specyfikacji podanej funkcji $f : X \rightarrow Y$. Innymi słowy, każdy algorytm określa funkcję $a : X' \rightarrow Y'$ z dziedziny X' dopuszczalnych wartości wejściowych do zbioru Y' możliwych wartości wyjściowych. Algorytm jest poprawną implementacją specyfikacji podanej funkcji f jeśli $X \subseteq X'$ oraz $Y \subseteq Y'$ i gdy zachodzi $f(x) = a(x)$ dla każdego x należącego do X .

Drugim celem algorytmu jest zaimplementowanie funkcji tak, aby nakład czasu i środków na obliczenie wartości $f(x) = a(x)$ był jak najmniejszy.

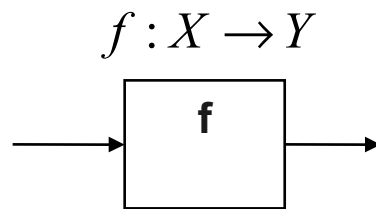
Kolejny cel algorytmu, którego znaczenie wzrasta, to implementacja funkcji a w taki sposób, aby była ona łatwa do zrozumienia, aby możliwe było jak najłatwiejsze dowiedzenie jej poprawności i aby można było ją łatwo zmienić, gdy zmianie ulegnie jej *specyfikacja*.

50 do 90 % czasu programistów jest poświęcane na poprawę, utrzymanie i modyfikację kodu. Co zrobić, aby zmniejszyć wysiłek i uczynić pracę bardziej usystematyzowaną ?

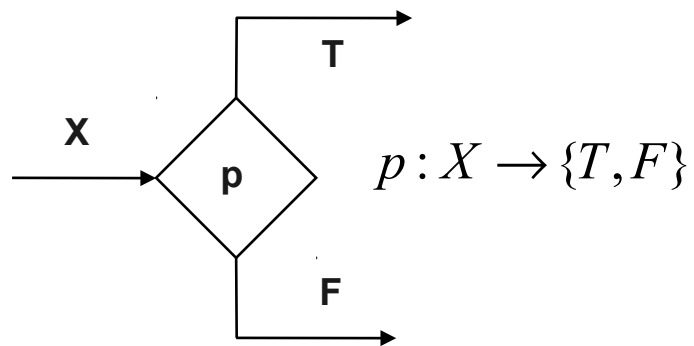
Odpowiedź 1. Programowanie strukturalne „z góry na dół”.

Schemat blokowy algorytmu - Flowchart

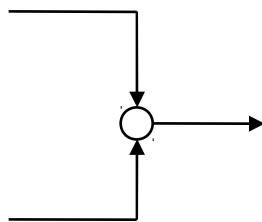
1. Wierzchołek funkcyjny



2. Wierzchołek warunkowy (predykatowy)



3. Wierzchołek łączący



Schemat blokowy (ang. flowchart)
– graf skierowany posiadający 3
możliwe rodzaje wierzchołków.
Jest to schematyczna
reprezentacja procesu, przebiegu
działania procedury, systemu lub
programu komputerowego

Schemat blokowy strukturalny

Jest to schemat blokowy, który może być wyrażony jako złożenie czterech podstawowych (elementarnych) schematów blokowych.

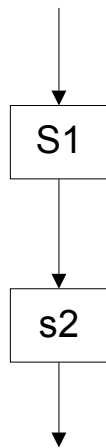
O ile stosunkowo łatwym zadaniem jest pokazanie, że **każdy** program komputerowy może zostać przedstawiony w postaci schematu blokowego, o tyle nie jest rzeczą oczywistą, że z kolei każdy schemat blokowy może być reprezentowany przez schemat strukturalny.

Pokazali to Corrado Bohm i Giuseppe Jacopini w 1966 r. Jest to tzw. *teoria programowania strukturalnego*.

Natychmiastowym następstwem powyższego stwierdzenia jest to, że cztery *elementarne schematy blokowe* są wystarczające do przedstawienia *dowolnego* algorytmu.

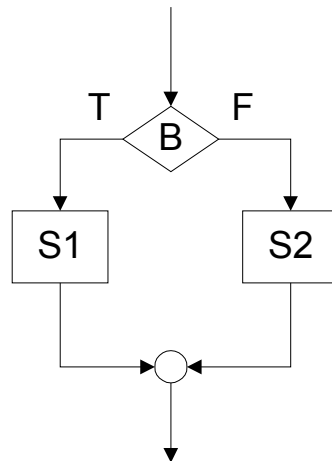
Podstawowe schematy blokowe

Kompozycja



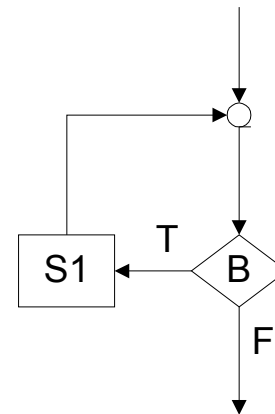
do S1; S2; od

Selekcja



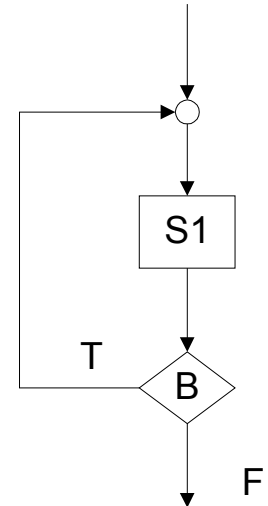
**if B then S1
else S2**

Iteracja



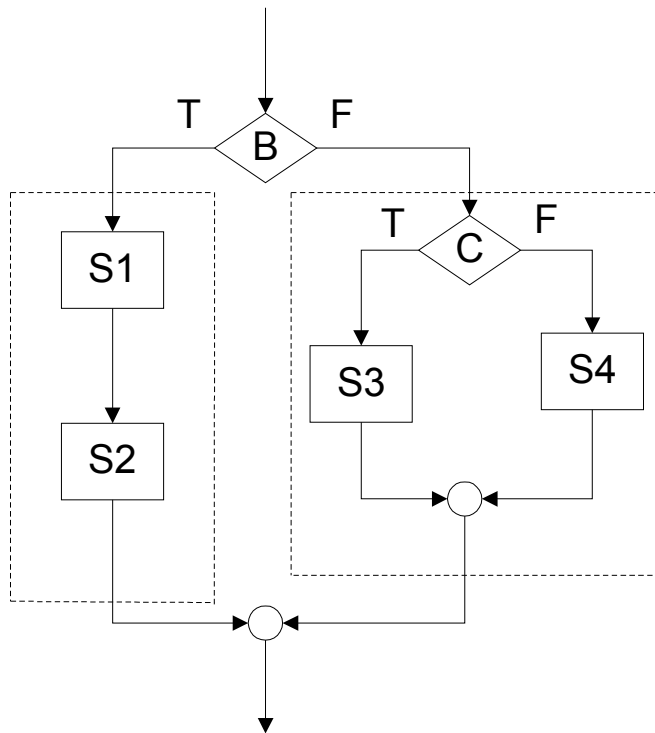
while B do S1 od;

Iteracja

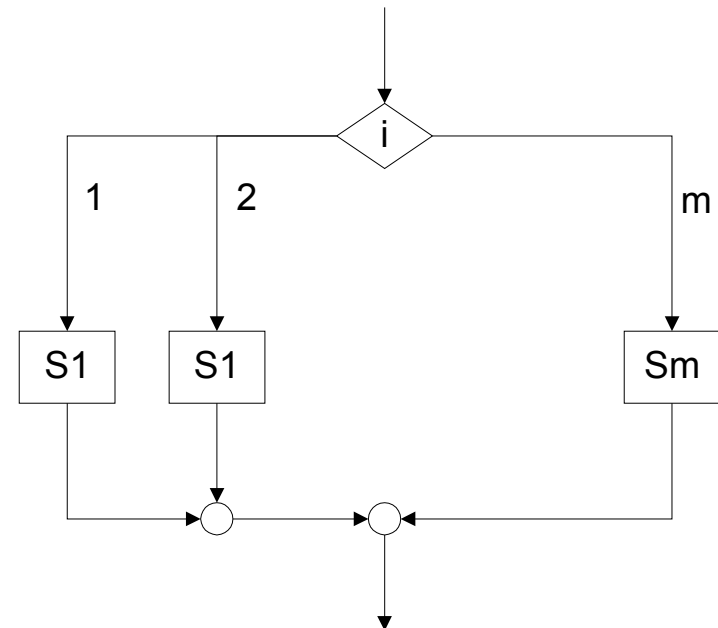


do S1 while B od;

Przykłady schematów strukturalnych [1]

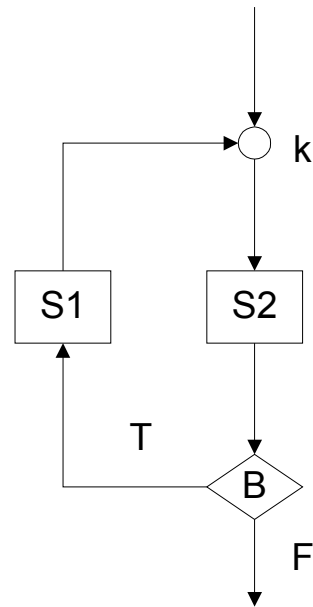


**if B then do S1; S2 od
else if C then S3
else S4 fi fi;**



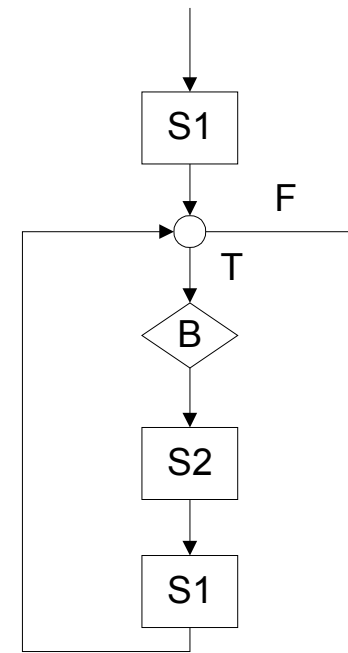
case i of S1; S2; ...; Sm fo

Przykłady schematów strukturalnych [2]



Schemat niestukturalny.

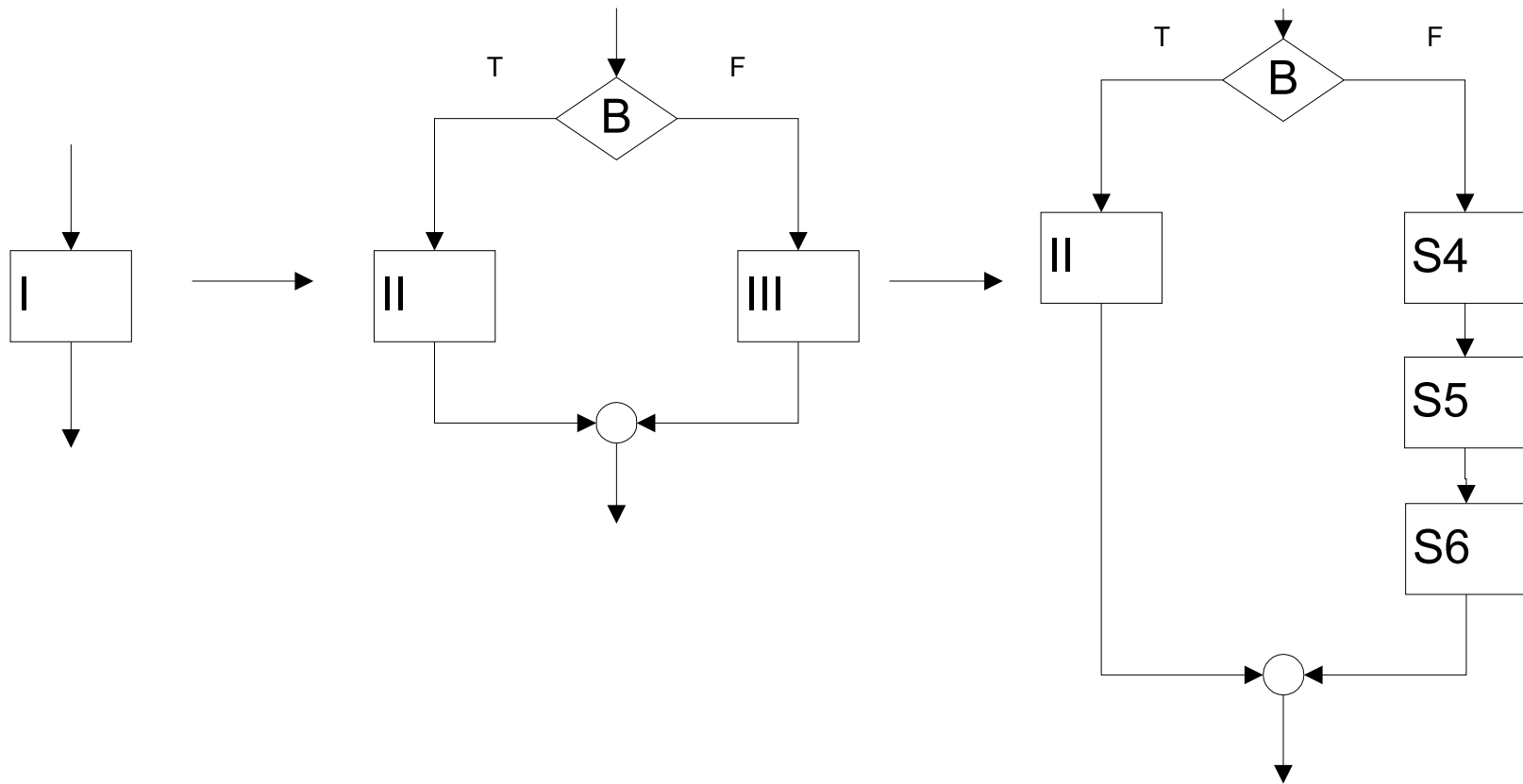
k: S1; if B then S2; goto k fi



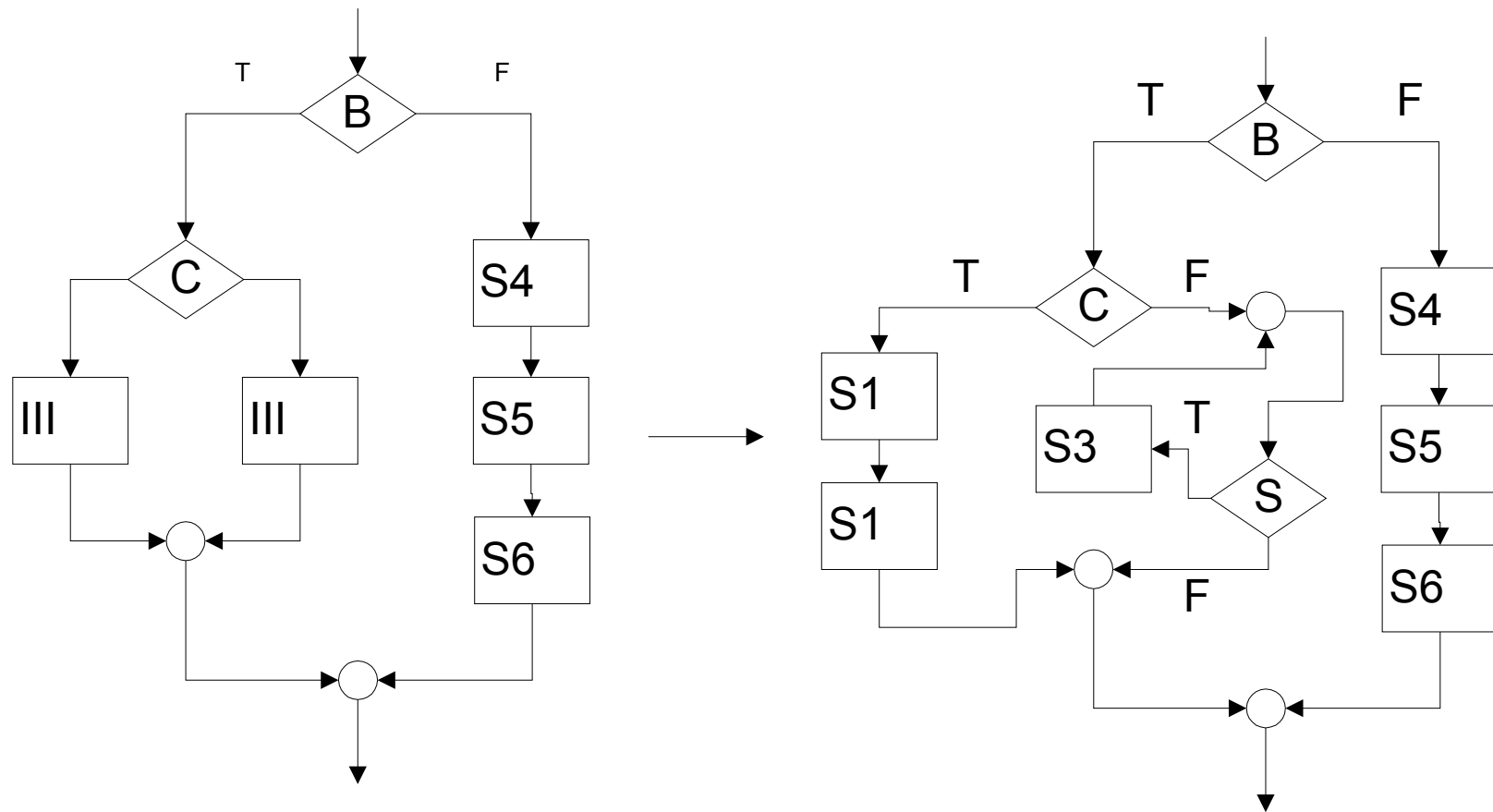
Schemat strukturalny.

do S1; while B do S2; S1 od od

Przykład dekompozycji problemu metodą „z góry na dół” [1]



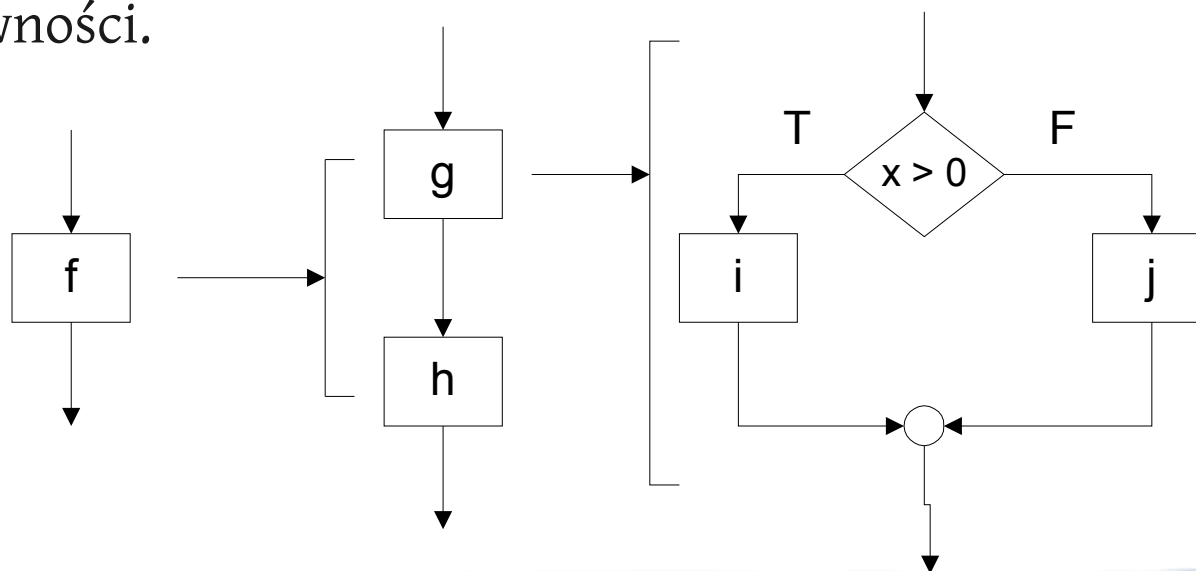
Przykład dekompozycji problemu metodą „z góry na dół” [2]



Programowanie strukturalne z wykorzystaniem metody „z góry na dół”

Termin „programowanie **top-down**” odnosi się do procesu stopniowego rozbijania (rafinowania) algorytmu na sukcesywnie coraz mniejsze elementy aż do momentu, w którym możliwe jest łatwe napisanie kodu źródłowego.

Algorytmy projektowane w taki sposób wykazują tendencje do cechowania się poprawnością wbudowaną w nie w małych krokach. Z tego powodu będą one miały zazwyczaj mniej błędów i łatwiej będzie dowodzić ich poprawności.



Pętla *while*

Ma postać:

```
while(wyrażenie-logiczne)
{
    wykonaj;
}
```

Wyrażenia i instrukcje znajdujące się w bloku { ... } podlegają wykonaniu cyklicznemu tak długo, jak długo *wyrażenie-logiczne* jest prawdziwe. Zapis równoważny z wykorzystaniem instrukcji *if* oraz instrukcji skoku *goto* wygląda jak następuje:

```
START:
if(wyrażenie-logiczne) {
    wykonaj;
    goto START;
}
```

Pętla *do-while*

Ma postać:

```
do
{
    wykonaj;
}
while(wyrażenie-logiczne);
```

Wyrażenia i instrukcje znajdujące się w bloku { ... } podlegają wykonaniu co najmniej raz. Następnie wartościowane jest wyrażenie-logiczne. Jeśli jego wartość odpowiada prawdzie logicznej, następuje powtórne wykonanie bloku. Potem sprawdzenie warunku i tak dalej. Zapis równoważny z wykorzystaniem instrukcji *if* oraz instrukcji skoku *goto* wygląda jak następuje:

```
START:
wykonaj;
if(wyrażenie-logiczne) {
    goto START;
}
```

Pętla *for*

Ma postać:

```
for(inicjalizacja; wyrażenia-logiczne; czynności-po-każdym-kroku)  
{  
    blok;  
}
```

Wykonanie pętli przebiega zgodnie ze schematem, który symbolicznie da się przedstawić w sposób następujący:

```
inicjalizacja;  
START:  
if(every in wyrażenia-logiczne) {  
    blok;  
    czynności-po-każdym-kroku;  
    goto START;  
}
```

every in ... oznacza wymaganie, by każde z wyrażeń należących do zbioru *wyrażenia-logiczne* było prawdziwe

Wyrażenia i instrukcje znajdujące się w sekcjach *inicjalizacja*, *wyrażenia-logiczne*, *czynności-po-każdym-kroku* są oddzielone przecinkami. Możliwa jest również rezygnacja z użycia tych sekcji (średniki separujące **MUSZĄ POZOSTAC**).

`for(;;) {}` jest przykładem pętli nieskończonej i nic nie robiącej.

Pętla *for*

postaci:

```
for( ; wyrażenie-logiczne ; )  
{  
    blok;  
}
```

Jest ekwiwalentem pętli while:

```
while(wyrażenie-logiczne) {  
    blok;  
}
```


Instrukcje *break* oraz *continue*

Są to w istocie warianty instrukcji *goto*. Używa się ich w blokach pętli. *break* powoduje natychmiastowe zakończenie działania pętli. *continue* zaś sprawia, że wykonywanie aktualnego kroku pętli zostaje przerwane (kolejne kroki mogą wystąpić, o ile warunki są spełnione).

```
while(wyrażenie-logiczne)
{
    wykonaj-1;
    break;
    wykonaj-2;
}
```

To forma równoważna formie:

```
while(wyrażenie-logiczne)
{
    wykonaj-1;
    goto KONIEC;
    wykonaj-2;
}
KONIEC:
```

Instrukcje *break* oraz *continue*

```
while(wyrażenie-logiczne)
{
    wykonaj-1;
    continue;
    wykonaj-2;
}
```

ma formę równoważną znaczeniowo:

```
START:
while(wyrażenie-logiczne)
{
    wykonaj-1;
    goto START;
    wykonaj-2;
}
```

Instrukcje *break* oraz *continue*

Uwaga, w przypadku pętli `for` instrukcja *continue* nie spowoduje ponownego wykonania sekcji inicjalizacyjnej. Nie jest to więc instrukcja dokładnie rozpoczynająca pętlę „od zera”, lecz mechanizm kończący aktualny krok pętli.

```
for(inicjalizacja; wyrażenia-logiczne; czynności-po-każdym-kroku)  
{  
    wykonaj-1;  
    continue;  
    wykonaj-2;  
}
```

jest odpowiednikiem:

```
inicjalizacja;  
START:  
if(every in wyrażenia-logiczne) {  
    wykonaj-1;  
  
    goto KONIEC_KROKU:  
  
    wykonaj-2;  
    KONIEC_KROKU:  
    czynności-po-każdym-kroku;  
    goto START;  
}
```

Zadanie

Napisać program zliczający sumę pierwszych 20 liczb parzystych (0, 2, 4, 6, 8, ..., 38) z wykorzystaniem pętli.

```
int main(void)
{
    int suma;
    int i;
    int licznik;

    suma = licznik = i = 0;
    while(licznik < 20) {
        if(i % 2 == 0) {
            licznik += 1;
            suma += i;
        }
        i += 1;
    }

    printf("Suma wynosi %d\n", suma);

    return 0;
}
```

... teraz z wykorzystaniem *for* ...

```
int main(void)
{
    int suma;
    int i;
    int licznik;

    for(suma = 0, licznik = 0, i = 0; licznik < 20; i++) {
        if(i % 2 == 0) {
            suma += i;
            licznik += 1;
        }
    }

    printf("Suma wynosi %d\n", suma);
    return 0;
}
```

... i funkcyjnie (Clojure)

```
(reduce + (take 20 (filter even? (iterate inc 0))))
```

Zadanie

Zaimplementować algorytmy:

- silnia
- ciąg Fibonacciego
- szybkie potęgowanie

z wykorzystaniem pętli while oraz for