

Programare procedurală – M1

Algoritmi: Caracteristici. Descriere. Complexitate. Corectitudine

Grigore ALBEANU

<http://www.researcherid.com/rid/H-4522-2011>

Agenda

- ▶ Obiectivele disciplinei
 - ▶ Conținutul tematic
 - ▶ Lucrări de laborator
 - ▶ Evaluare
 - ▶ Bibliografie
-
- ▶ C1 – Algoritmi: Caracteristici. Descriere. Complexitate

Obiectivele disciplinei

OBIECTIVELE DISCIPLINEI

1. Formarea deprinderilor de programare structurata in limbaje de programare clasice si moderne.
2. Insusirea instructiunilor de programare procedurala in limbajul C
3. Deprinderea tehnicilor de testare si verificare a corectitudinii programelor.
4. Asigurarea compatibilitatii cu invatamantul de excelenta : Oxford University (http://web2.comlab.ox.ac.uk/oucl/prospective/ugrad/csatox/cs_core1.html), California State University (<http://csc.csudh.edu/jhan/Spring2008/csc321/CSC321-syllabus.htm>), University of Cambridge (<http://www.cl.cam.ac.uk/teaching/0809/CST/node50.html>).

A **programming paradigm** is a fundamental style of [computer programming](#). (Compare with a [methodology](#), which is a style of solving specific [software engineering](#) problems). Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints, etc.) and the steps that compose a computation (assignment, evaluation, continuations, data flows, etc.).

Wiki: Imperative programmingIn [computer science](#), **imperative programming** is a [programming paradigm](#) that describes computation in terms of [statements](#) that change a program [state](#). In much the same way that [imperative mood](#) in [natural languages](#) expresses commands to take action, imperative programs define sequences of commands for the computer to perform. (<http://wapedia.mobi/en>)

Conținutul tematic

- ▶ Algoritmi: Caracteristici. Descriere. Complexitate [Complexitatea programelor (time.h)]. Corectitudine [Corectitudinea programelor C. Metoda aserțiunilor (assert.h)].
- ▶ Limbaje de programare. Caracteristici. Exemple: FORTRAN, C, Pascal...
- ▶ Limbajul de programare C: Entități sintactice. Operatori. Expresii. Instrucțiuni. Funcții (definire și declarare, transferul parametrilor).
- ▶ Directive de preprocesare. Tablouri și Pointeri. Funcția main cu argumente. Pachetele: stdio.h, math.h, string.h
- ▶ Alocare statică – Alocare dinamică. Structuri de date dinamice (liste și arbori). Aplicații ale utilizării tipurilor de date structurate (struct, union, typedef) cu ajutorul pointerilor: crearea și explorarea structurilor de date. Pachetele: stdlib.h, alloc.h
- ▶ Operații de intrare–ieșire. Fișiere în C și aplicații. Pachetul iostream.h (C++).
- ▶ Testarea programelor C.
- ▶ Utilizarea bibliotecilor statice (.LIB) și dinamice (.DLL).
- ▶ Metode de proiectarea programelor.

Lucrări de laborator

- ▶ Prezență (Realizarea lucrărilor practice–proiectelor) obligatorie !
- ▶ Tematica:
 1. *Structura programelor C. Instrucțiuni decizionale. Instrucțiuni repetitive*
 2. *Tablouri unidimensionale. Tablouri bidimensionale*
 3. *Tablouri și pointeri*
 4. *Transferul parametrilor*
 5. *Unități de traducere. Funcția main cu argumente. Comunicare cu module scrise în limbaj de asamblare sau alte limbi de programare.*
 6. *Funcții cu număr variabil de argumente. Pointeri*
 7. *Aplicații ale pointerilor (liste, arbori, grafuri). Fișiere*
 8. *Corectitudinea și complexitatea programelor*
 9. *Testarea programelor*
 10. *Biblioteci statice și biblioteci dinamice. Realizarea aplicațiilor complexe.*

Evaluare

- ▶ 30% Laborator
- ▶ 70% Evaluare finală în (pre)sesiune
- ▶ Sesiuni de examinare: iarnă, vară, toamnă/mărire de notă
- ▶ Refacerea lucrărilor de laborator condiționează intrarea în evaluarea finală.
- ▶ Structura lucrării finale: Sintaxa/semantica C, algoritmi, complexitate (teorie și aplicații), corectitudine (aserțiuni), directive de preprocesare, pointeri, fișiere, funcții cu număr variabil de argumente, *main* cu parametri, generarea sevențelor de test.

Bibliografie

- ▶ G. Albeanu, Algoritmi și limbaje de programare, Editura Fundației România de Mâine, București, 2000.
- ▶ B.W. Kernighan, D.M. Ritchie, The C programming language, Prentice Hall, 1988 (2nd ed.).
- ▶ Peter Salus, Handbook of Programming Languages: Vol. II: Imperative Programming Languages, Macmillan Technical Publishing, 1998.
- ▶ C.A. Giumele, Introducere în Analiza Algoritmilor, Polirom, 2004.
- ▶ H.S. Wilf, Algorithmes et complexité, Masson / Prentice+Hall, 1989.
- ▶ Bălănescu T., Corectitudinea Algoritmilor, Editura Tehnică, 1995.
- ▶ B.W. Kernighan, R. Pike, The practice of programming, Addison-Wesley, 1999.
- ▶ P. van der Linden, Expert C programming, Prentice Hall, 1994.
- ▶ G. Perry, C by examples, Que, 2000.
- ▶ P.S. Deshpande, O.G. Kakde, C and Data structures, Charles River Media, 2004.

Algoritmi: Caracteristici. Descriere. Complexitate. Corectitudine.

Prin **algoritm** vom înțelege o secvență finită de comenzi explicite și neambigue care executate pentru o mulțime de date (ce satisfac anumite condiții inițiale), conduce în timp finit la rezultatul corespunzător.

Caracteristici: Generalitate, Claritate, Finititudine, Corectitudine, Performanță, Robustete

Descriere: Limbaj natural / Pseudocod, Diagramă (schemă logică), program, ...

Complexitatea algoritmilor – 1

- ▶ Analiza complexității unui algoritm presupune determinarea resurselor de care acesta are nevoie pentru a produce datele de ieșire. Prin resursă înțelegem *timpul de executare*, dar uneori este necesar să analizăm și *alte resurse* precum: memoria internă, memoria externă etc.
- ▶ Modelul mașinii pe care va fi executat algoritmul nu presupune existența operațiilor paralele; operațiile se execută secvențial.
- ▶ În analiza complexității unui algoritm avem în vedere *cazul cel mai defavorabil* din mai multe motive:
 - a) Timpul de executare în cazul cel mai defavorabil oferă o limită superioară a timpului de executare (avem certitudinea că executarea algoritmului nu va dura mai mult). Această situație va fi acoperitoare pentru algoritmii cu funcționare în cadrul sistemelor pentru controlul proceselor în timp real.
 - b) Situația cea mai defavorabilă este întâlnită des.
 - c) *Timpul mediu de executare* este, uneori, apropiat de timpul de executare în cazul cel mai defavorabil, dar dificil de estimat.

Complexitatea algoritmilor – 2

Analiza exactă a complexității unui algoritm conduce la formule extrem de complicate. De aceea se folosesc notații asimptotice, care evidențiază doar termenul preponderent al formulei. Fie f și g două funcții reale. Atunci:

- $f(x) = o(g(x))$ dacă există și este egală cu 0 (zero) limita: $\lim_{x \rightarrow \infty} f(x)/g(x)$.
- $f(x) = O(g(x))$ dacă există c și x_0 astfel încât $|f(x)| < Cg(x)$ ($\forall x > x_0$).
- $f(x) = \Theta(g(x))$ dacă există constantele C_1 și C_2 pozitive și x_0 astfel încât pentru orice $x \geq x_0$: $C_1g(x) \leq f(x) \leq C_2g(x)$.
- $f(x) \sim g(x)$ dacă $\lim_{x \rightarrow \infty} f(x)/g(x) = 1$.
- $f(x) = \Omega(g(x))$ dacă există $\varepsilon > 0$ și un sir $x_1, x_2, \dots, x_n, \dots \rightarrow \infty$ astfel încât pentru oricare $j \geq 1$ are loc inegalitatea $|f(x_j)| > \varepsilon g(x_j)$.

În general, se consideră că un algoritm este mai rapid decât altul dacă are un ordin de mărime pentru timpul de executare mai mic. Pentru o dimensiune mică a datelor de prelucrat, aceste comparații pot fi (însă) eronate.

Notația Θ este utilizată pentru a specifica faptul că o funcție este mărginită (inferior și superior). Semnificația notației O este de limită superioară, în timp ce semnificația notației Ω este de limită inferioară.

Complexitatea algoritmilor – 3

Definiția 1. Fie A un algoritm, n dimensiunea datelor de intrare și $T(n)$ timpul de executare estimat pentru algoritmul A. Se spune că algoritmul A are comportare polinomială (apartine clasei P) dacă există $p > 0$ astfel încât $T(n) = O(n^p)$.

Definiția 2. O funcție care crește mai rapid decât funcția putere x^p , dar mai lent decât funcția exponențială a^x cu $a > 1$, se spune că este cu creștere exponențială moderată. Mai precis: f este cu creștere exponențială moderată dacă pentru oricare $p > 0$ avem $f(x) = \Omega(x^p)$ și oricare $\epsilon > 0$ avem $f(x) = o((1 + \epsilon)^x)$.

Definiția 3. O funcție f are creștere exponențială dacă există $a > 1$ astfel încât $f(x) = \Omega(a^x)$ și există $b > 1$ astfel încât $f(x) = O(b^x)$.

Exemplul 1. (Produsul a două numere complexe). Considerăm numerele complexe $a+bi$ și $c+di$ ($i^2 = -1$; a, b, c, d numere reale). Un algoritm pentru calculul produsului $(a+bi)(c+di)$ este următorul:

n	$n \cdot n$	$n \cdot n \cdot n$	$\ln n$	$\log n$ (bază 2)
2	4	8	0.693147	1
16	256	4096	2.772589	4
32	1024	32768	3.465736	5
64	4096	262144	4.158883	6
128	16384	2097152	4.85203	7
256	65536	16777216	5.545177	8
512	262144	134217728	6.238325	9
1024	1048576	1073741824	6.931472	10

P1 **real** a, b, c, d, p, q;
 real t1, t2;
 SEQ
 Read a, b, c, d;
 t1 := a*c; t2 := b*d; p := t1-t2;
 t1 := a*d; t2 := b*c; q := t1+t2;
 write p, q;
 END

Acest algoritm (notat P1) necesită 4 înmulțiri, o adunare și o scădere. În final p reprezintă partea reală, iar q furnizează partea imaginară a produsului celor 2 numere complexe. Următorul algoritm (notat P2) calculează același produs folosind 3 înmulțiri, 3 adunări și 2 scăderi:

P2 **real** a, b, c, d, p, q;
 real t1, t2, t3, t4;
 SEQ
 Read a, b, c, d;
 t1 := a+b; t2 := t1*c; t1 := d-c; t3 := a*t1; q := t2+t3; t1 := d+c; t4 := b*t1; p := t2-t4; **write** p, q;
 END

Complexitatea algoritmilor – 4

Temă:

Determinarea maximului și minimului, simultan, folosind

$3n/2 + O(1)$ operații de comparare.

Exemplul 2. (*Determinarea elementului maxim dintr-un sir (tablou)*). Fie X un tablou cu n elemente aparținând unei mulțimi total ordonate $T: X = (x_1, x_2, \dots, x_n)$ cu $x_i \in T, i = 1, 2, \dots, n$. Se caută un indice k , $1 \leq k \leq n$ astfel încât $\max \{x_i : i=1, 2, \dots, n\} = x_k$, iar k este cel mai mic număr cu această proprietate.

Pentru rezolvarea acestei probleme putem utiliza următorul algoritm pentru $T = \text{real}$.

```
procedure maxim(x,n,y,k)
integer n; real array x(n);
real y; integer i,k;
SEQ
    k:=1; y:=x[1];
    for i := 2, n, 1 do if y < x[i] then SEQ y := x[i]; k:=i END;
    return
END
```

Observăm că $T(n) = n-1$. Necesanul de memorie pentru stocarea datelor de prelucrat se exprimă în funcție de metoda de reprezentare a informației în calculator.

Propoziția 1. Pentru determinarea elementului maxim al unei mulțimi total ordonate cu n elemente sunt necesare cel puțin $n-1$ comparații ($T(n) = n-1$).

Demonstrație. Pentru $n = 2$, este evident că $T(2) = 1$. Presupunem că propoziția este adevărată pentru oricâte k elemente, $2 \leq k \leq n$ și considerăm o mulțime oarecare (total ordonată) cu $n+1$ elemente. Fie A un algoritm oarecare ce rezolvă problema propusă. Printr-o renumerotare a elementelor putem presupune că prima comparație efectuată de A este cea dintre x_1 și x_2 și că $x_1 \leq x_2$. Rezultă că $\max \{x_1, x_2, \dots, x_{n+1}\} = \max \{x_2, \dots, x_{n+1}\}$ pentru a căruia determinare sunt necesare cel puțin $n-1$ comparații, deci algoritmul A necesită cel puțin n comparații. Prin *inducție matematică*, rezultă că propoziția este adevărată pentru oricare n .

Complexitatea algoritmilor – 5

Exemplul 3. (*Determinarea celui mai mare divizor comun*). Fie m și n două numere întregi pozitive, iar q și r cîtul, respectiv restul împărțirii lui n la m , adică $n = qm + r$ ($0 \leq r < m$). Spunem că m divide n dacă restul împărțirii lui n la m este zero ($q = 0$).

Pentru determinarea celui mai mare divizor comun (cmmdc) a două numere se poate utiliza algoritmul lui Euclid. Dacă d este un divizor oarecare al numerelor n și m , atunci d divide restul r . Reciproc, dacă d este un divizor al numerelor m și r , relația $n = mq + r$ arată că d este divizor al numărului n . Deci: $cmmdc(n, m) = cmmdc(m, r)$. Dacă $r = 0$ atunci $n = qm$. Deci $cmmdc(n, m) = m$. Folosind notația $n \text{ mod } m$ pentru r , putem scrie: $cmmdc(n, m) = cmmdc(m, n \text{ mod } m)$. Necessarul de memorie pentru stocarea numerelor n și m se poate exprima prin $\Theta(\log m) + \Theta(\log n) \approx \Theta(\log(mn))$ biți. Timpul necesar executării algoritmului este dat de următorul rezultat.

Propoziția 2. Fie n și m numere întregi pozitive. Algoritmul lui Euclid pentru a determina $cmmdc(m, n)$ efectuează cel mult $[2\log_2 M] + 1$ operații de împărțire întreagă, unde $M = \max(m, n)$.

Demonstratie. Presupunem că $n \geq m$. Conform schemei de mai sus, algoritmul generează un sir x_0, x_1, \dots , cu $x_0 = n$, $x_1 = m$ și $x_{j+1} = x_{j-1} \text{ mod } x_j$, $j \geq 1$. Cum pentru oricare două numere întregi $1 \leq b \leq a$ avem

$a \text{ mod } b \leq (a-1)/2$ obținem $x_{j+1} \leq \frac{x_{j-1} - 1}{2} \leq \frac{x_{j-1}}{2}$. Rezultă (prin recurență) că $x_{2j} \leq \frac{x_0}{2^j}$ ($j \geq 0$) și $x_{2j+1} \leq \frac{x_1}{2^j}$ ($j \geq 0$), deci $x_k \leq 2^{-[k/2]}M$ ($k = 0, 1, \dots$). Algoritmul se încheie dacă $2^{-[k/2]}M < 1 \Leftrightarrow k > 2\log_2 M$.

Complexitatea algoritmilor – 6

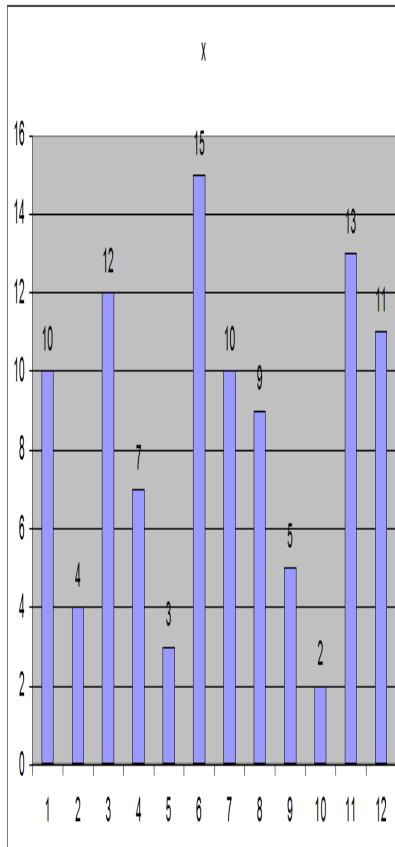
Exemplul 4. (*Sortare prin inserție*). Fiind dată o secvență de elemente caracterizate de valorile x_1, x_2, \dots, x_n aparținând unei mulțimi total ordonate T , să se determine o permutare $x_{i(1)}, x_{i(2)}, \dots, x_{i(n)}$ a secvenței date, astfel încât $x_{i(j)} \leq x_{i(k)}$ pentru $i(j) \leq i(k)$, unde “ \leq ” este relația de ordine pe mulțimea T . Metoda ce va fi prezentată în continuare se numește “*metoda jucătorului de cărți*” și este o metodă de sortare prin inserție.

Sortarea prin inserție se bazează pe următorul procedeu: Fie un tablou x cu n elemente ($x[i]$ este al i -lea element din secvența de intrare). Pomim cu subtabloul $x[1]$ și la primul pas căutăm poziția în care ar trebui să se găsească elementul $x[2]$. Dacă $x[2] < x[1]$, atunci $x[2]$ trebuie să fie mutat în locul elementului $x[1]$. La un pas i (pentru i între 1 și n), avem subtabloul $x[1..i-1]$ ordonat și încercăm să-l plasăm pe $x[i]$ astfel încât să obținem un tablou sortat între pozițiile 1 și i . Pentru aceasta, se compară succesiv $x[i]$ cu elementele tabloului $x[1..i-1]$ pentru a se determina acea poziție j pentru care $x[i] \geq x[j]$, indexul de plasare fiind $j+1$. Algoritmul prezentat în continuare utilizează inserția directă:

```
procedure insert_sort(n,x);
integer n; integer array x(n); integer i,j,temp;
SEQ
for i := 2, n, 1 do
    SEQ
    temp := x[i]; j:=i-1;
    while (j>=1) and (x[j] > temp) do
        SEQ x[j+1]:=x[j]; j:=j-1 END
    x[j+1]:=temp
END;
return
END
```

Este clar că, timpul de executare nu depinde doar de n , numărul de elemente de sortat, ci și de poziția inițială a elementelor din secvență. Fie $F(n)$ - numărul de comparații necesare, iar $G(n)$ numărul de mutări necesare algoritmului `insert_sort` pentru sortarea unui tablou cu n elemente.

Propoziția 3. Complexitatea metodei de sortare prin inserție directă este caracterizată prin: $F(n) = O(n^2/2)$, $G(n) = O(n^2/2)$.



Complexitatea algoritmilor – 7

Exemplul 5. (*Produsul a două matrice*). Fie **A** un tablou cu câte m rânduri (linii) și n coloane, $1 \leq m \leq 20$, $1 \leq n \leq 10$ și **B** un alt tablou cu n rânduri (linii) și p coloane, $1 \leq n \leq 10$, $1 \leq p \leq 30$. Se cere determinarea tabloului **C** astfel încât

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}, \quad 1 \leq i \leq m, \quad 1 \leq j \leq p.$$

Se presupune că $A = (a_{ij})_{1 \leq i \leq m, 1 \leq j \leq n}$, $B = (b_{ij})_{1 \leq i \leq m, 1 \leq j \leq n}$. Următorul algoritm îndeplinește funcția cerută.

```
integer m, n, p, i, j, k;
real array a(20,10), b(10,30), c(20, 30);
real s;
SEQ
Write "Dimensiunile m, n, p =";
Read m, n, p;
Write "Tablul A:";
for i:=1, m, 1 do for j:=1, n, 1 do Read a[i,j];
for i:=1, n, 1 do for j:=1, p, 1 do Read b[i,j];
for i := 1, m, 1 do
    for j := 1, p, 1 do
        SEQ
        s:=0;
        for k:=1, n, 1 do s := s + a[i, k] * b[k, j];
        c[i,j]:=s;
        END;
Write "Tabloul C =";
for i:=1, m, 1 do for j :=1, n, 1 do Write c[i,j];
END
```

Propoziția 4. Pentru determinarea produsului a două matrice, **A** cu m rânduri și n coloane, **B** cu n rânduri și p coloane, sunt necesare $m \times n \times p$ înmulțiri.

Temă:

Complexitatea algoritmului de înmulțire a unui sir de matrice A_1, A_2, \dots, A_k , $k > 2$.

(Metoda programării dinamice permite obținerea unui algoritm cu număr minim de operații de înmulțire)

Complexitatea programelor C/C++

- ▶ Pentru măsurarea timpului includeți fișierul antet `<time.h>`. Acesta conține prototipul funcției `clock()`, care întoarce numărul de tacte de ceas de timp real scurs de la începerea programului, până în punctul în care se plasează această funcție. Pentru obținerea timpului în secunde se folosește expresia `clock() / CLK_TCK`.
- ▶ Plasând două asemenea expresii, înaintea și după apelul subprogramului aflat sub măsurare, diferența lor ne dă timpul consumat de algoritm.
- ▶ *In versions of Microsoft C before 6.0, the `CLOCKS_PER_SEC` constant was called `CLK_TCK`.*

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

void sleep( clock_t wait );

int main( void ) {
    long i = 600000L;
    clock_t start, finish;
    double duration;

    /* Delay for a specified time. */
    printf( "Delay for three seconds\n" );
    sleep( (clock_t)3 * CLOCKS_PER_SEC );
    printf( "Done!\n" );

    /* Measure the duration of an event. */
    printf( "Time to do %ld empty loops is ", i );
    start = clock();
    while( i-- );
    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf( "%2.1f seconds\n", duration );
    return 0;
}

/* Pauses for a specified number of milliseconds. */
void sleep( clock_t wait ) {
    clock_t goal;
    goal = wait + clock();
    while( goal > clock() );
}
```

Corectitudinea algoritmilor / programelor C – 1

A verifica corectitudinea logică a unui program înseamnă a verifica dacă programul conduce într-un interval finit de timp la obținerea soluției corecte a problemei pentru care a fost elaborat.

Presupunem că este disponibilă funcția *assert* care pentru o expresie logică falsă determină oprirea programului. Pentru programarea în C se poate utiliza fisierul antet assert.h.

#include <assert.h>

Pentru programarea în alte limbi se poate folosi tehnologia XUnit Testing. De exemplu pentru Java este disponibil pachetul JUnit.

Notătie. Construcția

```
assert(P);  
A();  
assert(Q);
```

este numită *formulă de corectitudine totală*

și conține următoarele elemente:

P – expresie logică care descrie proprietățile datelor de intrare (*precondiția*);

A – subprogramul/funcția/modulul (secvența de instrucțiuni) supus(ă) analizei;

Q – expresie logică care descrie proprietățile datelor de ieșire (*postcondiția*).

Corectitudinea algoritmilor / programelor C – 2

Definiția 4. Un program

```
assert(P);  
A();  
assert(Q);
```

este corect când este adevărată propoziția (logică):

Dacă

datele de intrare satisfac precondiția P

Atunci

- 1) *executarea lui A se termină (într-un interval finit de timp) și*
- 2) *datele de ieșire satisfac postcondiția Q.*

Folosind elemente de logică matematică rezultă că următoarea **observație** este adevărată:

Dacă

```
assert(P);  
A();  
assert(Q);
```

și

```
assert(R);  
A();  
assert(Q);
```

sunt formule corecte atunci

```
assert (P || R);  
A();  
assert(Q);
```

este program corect

Corectitudinea algoritmilor / programelor C – 3

Regula compunerii secvențiale (CS) :

Dacă A() este de forma **SEQ** B(); C() **END** (adică B și C sunt instrucțiuni care se execută secvențial) atunci se verifica formula

```
assert(P);  
A();  
assert(Q);
```

revine la a verifica formulele

```
assert(P);  
B();  
assert(R);
```

și

```
assert(R);  
C();  
assert(Q);
```

unde R este o expresie logică asupra datelor intermediare:

CS:

Dacă A() = **SEQ** B(); C() **END**, iar assert(P);B();assert(R); și assert(R);C();assert(Q); sunt secvențe corecte **atunci** assert(P);B(); C(); assert(Q); este program corect.

Este evident că regula CS poate fi aplicată iterativ, adică “din aproape în aproape”. Mai precis, dacă A este de forma **SEQ** A₁; A₂; ...; A_n **END** atunci obținem regula compunerii secvențiale generale:

CSG:

Dacă assert(P_i);A_i;assert(P_{i+1}); i = 0, 2, ..., n sunt programe corecte, **atunci** assert(P₀);A₁; A₂; ...; A_n; assert(P_n); este program corect.

Exemplul 6. Este ușor de verificat că următorul program este corect (presupunem că x și y sunt variabile întregi, iar a și b sunt constante întregi):

```
#include <stdio.h>  
.....  
int x, y, a, b;  
a = 640; b = 480;  
x=a; y=b;  
assert(x == a && y == b);  
{  
    x = x+y;  
    y = x-y;  
    x = x-y;  
}  
assert(x == b && y == a);  
.....
```

Acest algoritm este o alternativă la metoda generală de interschimbarea variabilelor, valabilă pentru date întregi.

Corectitudinea algoritmilor / programelor C – 4

Regula instrucțiunii de atribuire (A):

Regula implicației (I):

Această regulă este utilă în cazul funcțiilor ce urmează să fie apelate în condiții mai târziu decât pentru cele care au fost deja verificate (testate). Vom spune că o proprietate P este mai tare decât proprietatea Q dacă este adevărată implicația: $P \rightarrow Q$.

Regula implicației se descrie astfel:

I: Dacă

assert(P); A(); assert(Q);
este program corect,
 $P_1 \rightarrow P$ și $Q \rightarrow Q_1$
sunt implicații valide
atunci
assert(P_1); A(); assert(Q_1));
este program corect

Fie notațiile:

x	Variabilă simplă
e	Expresie
Def(e)	Proprietatea satisfăcută de acele elemente pentru care evaluarea expresiei e este corectă (Exemplu: pentru int b[10], Def(b(i)) := (i=0, 2, ..., 9)); Se citește "e este bine definită".
P(x)	Formulă în care apare variabila x;
P(x/e)	Formula obținută din P(x) prin substituirea variabilei simple x cu expresia e, ori de câte ori x este variabilă liberă în P, iar dacă e conține o variabilă y care apare legată în P, înainte de substituție variabila y, din e, se înlocuiește printr-o nouă variabilă (care nu mai apare în e).

Valoarea de adevăr a propoziției $P \rightarrow Q(x/e)$ nu se schimbă dacă în $Q(x/e)$ se efectuează substituția descrisă de atribuirea $x := e$. Deci, regula atribuirii este:

A: Dacă $P \rightarrow (\text{Def}(e) \wedge Q(x/e))$ atunci secvența assert(P); $x := e$; assert(Q); este corectă.

Pentru variabile cu indici (indexate), apar complicații datorate accesului la indici, formalismul nu e simplu și optăm pentru analiza directă a executării instrucțiunilor unde apar variabile cu indici.

Exemplul 7. Următoarele secvențe sunt corecte:

- assert(x == factorial(n));
n++; x *= n;
- assert(x == factorial(n));
assert(x == a && y == b); /* a și b sunt cunoscute */
t = x; x = y; y = t;
assert(x == b && y == a);
- assert((0 ≤ i < n-1) ∧ (s = Σ {b[k] : k = 0, ..., i-1}));
s := s + b[i]; i := i + 1;
assert((0 < i < n) ∧ (s = Σ {b[k] : k = 0, ..., i-1}))
dacă b este declarat prin int b[n].

Corectitudinea algoritmilor / programelor C – 5

Regula instrucțiunii if (IF și IFR):

Dacă c este o expresie booleană, iar A și B sunt secvențe de program, pentru cele două forme ale instrucțiunii if sunt valabile două reguli de corectitudine.

IF: Dacă assert($P \&\& c$); A; assert(Q); și assert ($P \&\& \neg c$); B; assert(Q); sunt secvențe corecte, iar $P \rightarrow \text{Def}(c)$ este implicație validă
atunci assert(P); **if** (c) A; **else** B; assert(Q); este instrucțiune corectă.

IFR: Dacă assert($P \&\& c$); A; assert(Q); este secvență corectă, iar $P \wedge \neg c \rightarrow Q$ și $P \rightarrow \text{Def}(c)$ sunt implicații valide
atunci assert(P); **if** (c) A; assert(Q); este instrucțiune corectă.

Regula IFR descrie corectitudinea schemei **IF reduse**. Se poate observa că aplicarea regulilor instrucțiunii de decizie nu este în sine dificilă; corectitudinea acestei instrucțiuni se reduce la corectitudinea instrucțiunilor componente.

Exemplul 8. Sunt instrucțiuni corecte secvențele:

- a) assert(1);
 if (x>y) { t = x; x = y; y = t};
 assert(x <= y);
 - b) assert (x == a && y == b);
 if (x>=y) m = x; **else** m = y;
 assert(m == maxim (a,b));
 - c) assert(x == a);
 if (x<0) x = -x;
 assert(x == abs(a));
- adică valoarea absolută a numărului a.

Corectitudinea algoritmilor / programelor C – 6

Regula instrucției while (W):

Regula instrucției **while** trebuie să precizeze dacă nu apare fenomenul de ciclare infinită, iar prelucrările sunt corecte.

Fie o secvență de forma **assert(P); A(); while(c)S(); assert(Q);**

Presupunem că există o proprietate invariantă **I** (a se vedea mai jos) și o funcție de terminare **t** cu valori numere întregi astfel încât:

- Dacă **I** este adevărată atunci expresia booleană **c** este bine definită (adică **I → Def(c)** este formulă validă).
- Proprietatea **I** rezultă prin executarea secvenței **A** (adică **assert(P); A(); assert(I);** este instrucție corectă).
- La terminarea instrucției **while**, proprietatea finală **Q** poate fi dedusă (adică **I ∧ not C → Q** este formulă validă).
- **I** este proprietate invariantă la executarea unei iterări: dacă **I** este adevărată înainte de executarea secvenței **S** și expresia booleană **c** are valoarea **true**, atunci executarea secvenței **S** se termină într-un interval finit de timp și **I** este adevărată la sfârșit (adică **assert(I && c); S(); assert(I);** este secvență corectă).
- Dacă rezultatul evaluării expresiei **c** este **true** și proprietatea **I** este adevărată, atunci există cel puțin o iterare de efectuat (adică **I ∧ c → (t ≥ I)** este formulă validă).
- Valoarea lui **t** descrește după executarea unei iterări
(adică **assert((I && c) && (t == a)); S; assert(t < a);**).

În aceste condiții, secvența considerată inițial este corectă.

Corectitudinea algoritmilor / programelor C – 7

Exemplul 9. (*Determinarea celui mai mare divizor comun a două numere întregi*). Fie a și b două numere întregi, iar $m = |a|$ și $n = |b|$. Atunci următorul algoritm este corect:

```
assert ((x > 0) && (y > 0) && (x == m) && (y == n));
while (x <= y)
    if (x > y) x = x - y; else y = y - x;
assert( x == cmmdc(m,n));
```

Într-adevăr, există proprietatea invariantă

I: $(cmmdc(x,y) = cmmdc(m,n)) \wedge (x > 0) \wedge (y > 0)$,
iar ca funcție de terminare se poate lucra cu: $t(x,y) = x+y$. Verificarea ipotezelor structurii **While** este simplă.

Exemplul 10. (*Al n -lea termen al șirului lui Fibonacci*). Fie $f[n]$ al n -lea termen al șirului lui Fibonacci. Următorul algoritm este corect.

```
assert(n >= 0);
a = 0; b = 1; k = n;
while (k > 0) {
    temp = b; b = a + b; a = temp; k--;
}
assert(a == f[n]);
```

Într-adevăr, luăm funcția de terminare $t(k) = k$, iar proprietate invariantă este:

I: $(a = f_{n-k}) \wedge (b = f_{n-k+1}) \wedge (temp = f_{n-k}) \wedge (0 \leq k \leq n)$.

Deoarece instrucțiunile repetitive, **for** și **do while** (ciclu cu test final) se pot implementa folosind instrucțiunile anterioare, pentru demonstrarea corectitudinii acestora se aplică aceleași principii.

Algoritmi – Teme pentru lucrări practice

- ▶ Generarea numerelor pseudoaleatoare:
http://en.wikipedia.org/wiki/Random_number_generation
- ▶ Rezolvarea ecuațiilor liniare:
http://en.wikipedia.org/wiki/System_of_linear_equations
- ▶ Căutare în siruri de caractere:
http://en.wikipedia.org/wiki/String_searching_algorithm
- ▶ Generarea permutărilor, aranjamentelor, combinărilor, etc.
- ▶ Metode algoritmice în geometrie:
http://en.wikipedia.org/wiki/Computational_geometry
- ▶ Metode algoritmice în algebră:
http://en.wikipedia.org/wiki/Computer_algebra_system
- ▶ Metode algoritmice în analiza matematică:
http://en.wikipedia.org/wiki/Numerical_analysis
- ▶ Algoritmi evoluționisti:
http://en.wikipedia.org/wiki/Evolutionary_algorithm
- ▶ Alte clase de algoritmi (inclusiv aproximativi):
http://en.wikipedia.org/wiki/Approximation_algorithm