Conceptul de ALGORITM - abordare MODERNĂ

Marin Vlada

În cadrul acestui al doilea articol dedicat unei abordări moderne a conceptului de algoritm va fi prezentat detaliat modul în care ar trebui privit algoritmul luând în considerare diferite aspecte care sunt practic desconsiderate de abordarea clasică.

Practica rezolvării problemelor folosind un limbaj de programare a determinat de-a lungul timpului diverse abordări în funcție de performanța limbajului de programare, performanța calculatorului și, nu în ultimul rând, în funcție de metodele și tehnicile avansate privind implementarea raționamentelor pentru demonstrațiile corespunzătoare problemelor.

În [13] se arată că "*un limbaj de programare acționează ca o interfață între universul real al problemei de rezolvat și programul de rezolvare, punând la dispoziție o serie de elemente constructive (entități) și legi de combinare a acestora, prin care elementele problemei și acțiunile de rezolvare pot fi reprezentate și prelucrate la nivelul programului. A construi un program de rezolvare a problemei înseamnă, în esență, a găsi modalitatea de agregare a acestor entități, în așa fel încât rezultatul să constituie o replică a procesului pe care un rezolvitor uman l-ar executa pentru a rezolva problema*". Rezolvarea teoretică a unei probleme nu garantează și rezolvarea ei practică folosind calculatorul.

În general, un limbaj de programare este menit să faciliteze rezolvarea unor clase de probleme și se pretează mai bine anumitor *tipuri de algoritmi*. Este nevoie de experiență în utilizarea și cunoașterea calculatorului, de competență și intuiție, este nevoie de inspirație și creație. În astfel de situații este nevoie de cunoașterea mai multor limbaje de programare pentru a alege limbajul de programare adecvat pentru clasa de probleme din care face parte problema de rezolvat.

Experiența a arătat că, atunci când nu este ales limbajul de programare corespunzător, dacă totuși se ajunge să se rezolve problema, s-a făcut risipă de resurse (timp, memorie, finanțe etc.) și, prin urmare, eficiența și performanța au avut de suferit.

Conform [13], rezolvarea unei probleme folosind un limbaj de programare modern trebuie să urmeze următoarele etape:

- ***etapa de analiză și abstractizare a problemei*** care constă în identificarea *obiectelor* implicate în rezolvare și a *acțiunilor de transformare* corespunzătoare acestora. Un *obiect* este un element esențial al unei probleme, element care poate fi analizat independent, prelucrabil prin *operatori specifici*, în funcție de *tipul* său, și a cărui *stare curentă* influențează *starea problemei* din punct de vedere al rezolvării acesteia. Rezultatul acestei prime etape este *universul abstract al problemei* (***UAP***), care scoate în evidență *mulțimea tipurilor de obiecte*, *mulțimea obiectelor*, *mulțimea relațiilor între obiecte* și *mulțimea restricțiilor de prelucrare* necesare rezolvării problemei;
- ***etapa găsirii unei metode acceptabile de rezolvare*** care constă în precizarea exactă a *operatorilor de prelucrare* proprii obiectelor din *UAP*, *elaborării algoritmului de rezolvare* și, în final, *codificării algoritmului* și *reprezentării obiectelor* din *UAP* folosind entitățile și legile de combinare oferite de limbajul de programare.

Trebuie să precizăm faptul că aceste etape sunt influențate de natura și performanța limbajului de programare folosit, de entitățile prin care un program poate fi construit folosind limbajul ales.

În domeniul *inteligenței artificiale* este utilizat limbajul ***Prolog*** care oferă facilități de programare declarativă / obiectuală și care este un exemplu concludent de limbaj care impune parcurgerea celor două etape precizate mai

sus. De asemenea, limbajul *C++* oferă facilități moderne pentru rezolvarea diverselor probleme, respectând principiile *OOP* ([28, 29, 30]).

De asemenea, în domeniul *graficii pe calculator*, aceste etape sunt frecvente. Limbajul ***Java*** este un limbaj modern care utilizează din plin conceptele *programării orientate obiect*.

Astăzi, perfomanța unui *informatician / programator* este determinată de experiența și competența obținute în desfășurarea celor două etape (***Analiză***, ***Programare***):

- ***etapa gândirii obiectuale*** (***Analiză / Proiectare***) - modul de analiză și abstractizare a problemelor prin definirea corectă a obiectelor, a tipurilor de obiecte, a relațiilor dintre obiecte și a operatorilor specifici (*elaborarea UAP*; *etapa de concepție și analiză - proiectare*);
- ***etapa gândirii algoritmice*** (***Programare / Execuție***) - alegerea și aplicarea corectă a unor metode de rezolvare prin precizarea exactă a operatorilor de prelucrare a obiectelor, reprezentarea corectă a strategiilor algoritmice, reprezentarea codificată a obiectelor și a prelucrărilor conform unui limbaj de programare (*elaborarea algoritmului și programului; etapa de programare - codificare, implementare și execuție*).

În prezent este tot mai des invocată reprezentarea problemelor folosind concepte ***OOP*** (***O**bject **O**riented **P**rogramming*). Conceptul de ***obiect*** (**M. Minsky**, *The Society of Mind*, Touchstone Books, New York, 1986) are un rol important în știința cunoașterii și educației. Un *obiect* modelează o *entitate* din *lumea reală sau virtuală / imaginară* [14]. Prin definirea obiectelor se modelează diferite tipuri de entități și concepte din lumea reală sau imaginară.

În *activitatea de rezolvare a problemelor* trebuie să se *identifice / definească* obiectele din cadrul problemelor care provin din diferite domenii: științifice, economice, sociale etc.

Identificarea obiectelor este echivalentă cu determinarea entităților și conceptelor care reprezintă forme *fizice / grafice*, *fapte*, *evenimente*, *procese, stări* etc. Un *obiect* este caracterizat în mod unic prin *identificare*, *comportament* (caracteristică dinamică) și *stare* (caracteristică statică). O clasă este o familie de obiecte cu structură și comportamente identice.

Conform ***OMG*** (***O**bject **M**anagement **G**roup*) o clasă este o implementare care poate fi instanțiată în vederea generării de obiecte cu același comportament.

Comportamentul unui obiect corespunde cu *setul de mesaje* care se transmit obiectului, iar pentru fiecare mesaj există o *metodă* care are acces la starea obiectului. Metodele sunt de mai multe tipuri: *constructori*, *destructori*, *selectori* și *modificatori*.

În esență, rezolvarea unei probleme se va exprima printr-o codificare a *universului problemei* și a raționamentelor pentru procesul demonstrativ. Programul elaborat pentru rezolvarea problemei va fi conceput din două părți:

- date (de intrare și / sau ieșire);
- operații (acestea vor acționa asupra datelor).

Un limbaj de programare modern trebuie să ofere posibilitatea de a construi *tipuri de date* diferite de cele predefinite / standard. *Clasele* reprezintă implementarea unor tipuri abstracte de date (limbajul *C++*). O ***clasă*** reunește datele și operațiile în cadrul aceluiași tip de date:

Clasă = ***Structuri de date*** + ***Operații***

În limbajul *C++* definirea unei clase constă din două părți:

- ***declarații*** - lista elementelor componente ale clasei (date membre și metode);
- ***implementarea*** - implementarea metodelor clasei.

O ***structură de date*** reprezintă o structură destinată să stocheze *datele* precum și *metodele* pentru a *crea*, *modifica* și *accesa* datele [12]. Practica *elaborării programelor* pentru aplicațiile simple sau complexe a determinat *utilizarea abstractizării datelor* care constă în a separa algoritmii (procesele de calcul) de structurile de date. În acest fel, rezolvarea problemelor trebuie gândită în *termenii tipurilor abstracte de date* care vor fi folosite în elaborarea aplicațiilor.

Un ***tip abstract de date*** specifică ceea ce se poate face cu o structură (ce operații suportă). O structură de date are trei componente:

- ***tip abstract de date*** (***TAD***) - mulțime de operații care manipulează obiecte abstracte;
- ***structură de stocare*** - suport de memorare a obiectelor abstracte;
- ***implementarea operațiilor*** - definirea operațiilor *TAD* folosind structura de stocare.

Conceptul de algoritm

Practica *dezvoltării aplicațiilor software* (care necesită rezolvarea diverselor tipuri de probleme) a scos la iveală următoarele etape importante [12] (*gândirea obiectuală*: primele două etape ↔ *analiză-proiectare*; *gândirea algoritmică*: ultimele patru etape ↔ *programare-execuție*):

- ***specificarea problemelor*** - descrierea clară și precisă a problemelor indiferent din ce domeniu provin acestea;
- ***proiectarea soluțiilor*** - includerea problemelor în clasa corespunzătoare și alegerea modului de reprezentare a problemelor prin formularea *etapelor* și *procedeelor* corespunzătoare pentru procesele de rezolvare;
- ***implementarea soluțiilor*** - *elaborarea algoritmilor* și *codificarea* acestora într-un limbaj de programare modern;
- ***analiza soluțiilor*** - *eficiența soluțiilor* raportată la resursele utilizate: *memorie*, *timp*, *utilizarea dispozitivelor I/O* etc.;
- ***testarea și depanarea*** - verificarea execuției programului cu diverse seturi de date de intrare pentru a putea răspunde rezolvării oricărei probleme pentru care a fost elaborată aplicația;

- ***actualizarea și întreținerea*** - adaptarea soluțiilor implementate pentru *eliminarea erorilor* în rezolvarea unei anumite probleme și *compatibilitatea* cu sistemul de calcul și sistemul de operare folosite.

Conform acestor faze iese în evidență interdependența între următoarele activități importante: *Reprezentare* ↔ *Elaborare / Proiectare* ↔ *Execuție*.

Această interdependență este sintetizată prin schema prezentată în figura 1.

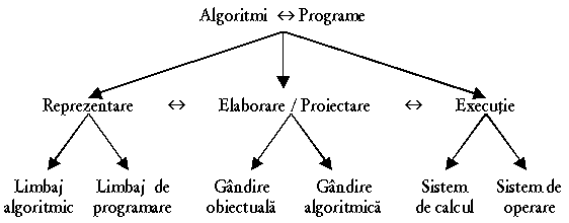

Figura 1

De asemenea, interdependența precizată mai sus se transmite și între componentele de pe nivelul inferior din schema arborescentă alaturată.

Competența și ***experiența*** în rezolvarea problemelor se pot obține doar dacă permanent se are în vedere această interdependență și dacă se întreprind eforturi pentru însușirea de noi cunoștințe și pentru cunoașterea corespunzătoare a tuturor aspectelor privind ***modelul fizic***, respectiv ***modelul virtual***, aspecte determinate de interdependența ***Sistem de calcul ↔ Algoritmică ↔ Programare.*** Aceasta este ilustrată în figura 2.

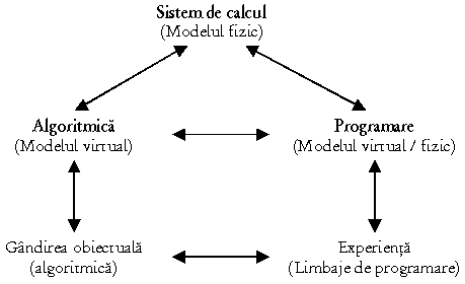

Figura 2

Practica și experiența elaborării programelor pentru rezolvarea problemelor scot în evidență următoarele aspecte foarte importante:
- ***modelul fizic***;
- ***modelul virtual***;
- ***modelul program***.

Modelul fizic

Acest model este dat de *sistemul de calcul*. Modelul trebuie luat în considerare atunci când se proiectează și se elaborează o aplicație; acest aspect reclamă competență în domeniul sistemelor de calcul și perfecționare continuă pentru cel care proiectează și elaborează aplicația.

Modelul virtual

Acest model este dat de *gândirea obiectuală și algoritmică*, *de modul de reprezentare a algoritmilor* și *de mașina virtuală pe care trebuie să se execute algoritmul elaborat*; în timp, acest model a suferit schimbări majore deoarece a fost influențat în continuu de *modelul fizic* și de clasa problemelor care urmau să fie rezolvate.

Modelul program

Acest model este reprezentat *de o îmbinare între modelul fizic* (sistemul de calcul) *și modelul virtual* (***algoritmul***); întotdeauna un program se elaborează într-un limbaj de programare care trebuie să respecte *restricțiile modelului fizic* și *restricțiile modelului virtual*.

Conform acestor aspecte trebuie să existe o echivalență între *algoritm* și *program* indiferent de restricțiile particulare ale modelelor. În esență, un ***program*** reprezintă codificarea într-un limbaj de programare a unui ***algoritm***, iar un ***algoritm*** reprezintă codificarea într-un limbaj de reprezentare a unui ***raționament***.

Programul se va executa pe o mașină fizică / reală reprezentată de un sistem de calcul (SC), iar *algoritmul* va trebui să simuleze execuția pe o mașină virtuală asemănătoare mașinii reale. Acest lucru înseamnă că restricțiile modelului fizic trebuie considerate și în modelul virtual, prin urmare execuția algoritmului trebuie să simuleze execuția programului pe mașina reală.

Comentarii metodologice și pedagogice

- Conform celor de mai sus *este inadecvată utilizarea schemelor logice* în reprezentarea algoritmilor; autorii de manuale de informatică fac o mare greșeală dacă nu țin seama de acest lucru; inventarea și apariția *structurilor de control* în programare au eliminat teoretic schemele logice;
- Utilizarea unui *pseudocod* în reprezentarea algoritmilor care se exprimă cu cuvinte-cheie din limba română nu este oportună dacă se ține seama că algoritmul va fi codificat într-un limbaj de programare modern: ***C***, ***C++***, ***Java***, ***Pascal***, ***Foxpro***, ***Oracle***, ***Prolog*** etc.; încă mai există manuale care utilizează această formă de reprezentare; pe viitor astfel de manuale nu trebuie să mai existe;
- ***Activitățile de laborator sunt esențiale în procesul de învățare***; profesorii de informatică sunt obligați să îmbine cât mai eficient *orele de predare cu orele de laborator* prin care de fapt se încheie ciclul pentru rezolvarea unei probleme cu calculatorul: *problema → modelul matematic → algoritmul → programul → calculator → rezultate → verificare soluții*.

Se observă echivalența *algoritm ↔ program ↔ SC* modulo operațiile realizate de algoritm în spațiul virtual, respectiv operațiile realizate de *SC* prin execuția programului corespunzător algoritmului. Algoritmul va trebui să simuleze toate operațiile declanșate prin lansarea în execuție a programului corespunzător algoritmului.

Definiția conceptului de algoritm

Un *algoritm* este sistemul virtual $A = (M, V, P, R, Di, De, Mi, Me)$ caracterizat de următoarele aspecte:

- *elaborare*;
- *reprezentare*;
- *execuție*;
- *corectitudine și analiză*.

Algoritmul

Sistemul *A* este constituit din următoarele elemente:

- *M - memorie virtuală (internă)* utilizată pentru stocarea temporară a informațiilor destinate variabilelor din mulțimea de variabile *V*; asupra variabilelor acționează procesul de calcul *P* care are acces la memorie prin rezervarea de memorie pentru variabilele din mulțimea *V*; inițial memoria virtuală este folosită pentru rezervare de memorie pentru variabilele din mulțimea *V*, după care este utilizată pentru scrierea și citirea de informații în locațiile corespunzătoare variabilelor utilizate în procesul de calcul *P*;
- *V - mulțime de variabile / structuri de date* definite conform raționamentului *R* corespunzător rezolvării unei probleme și care utilizează memoria *M* prin locații de memorie pentru fiecare tip de variabilă din *V*; locațiile de memorie rezervate variabilelor din *V* sunt utilizate de procesul de calcul *P* care prin execuția instrucțiunilor care constituie *P*, schimbă valorile (starea) locațiilor de memorie corespunzătoare variabilelor în conformitate cu implementarea raționamentului *R*;
- *P - proces de calcul* care este reprezentat de o colecție de instrucțiuni / comenzi exprimate într-un limbaj de reprezentare (cel mai utilizat fiind *pseudocodul*); folosind memoria virtuală *M* și mulțimea de variabile *V*, colecția de instrucțiuni implementează / codifică tehnicile și metodele care constituie raționamentul *R* conceput special pentru rezolvarea unei clase de probleme; execuția instrucțiunilor din *P* determină o dinamică a valorilor locațiilor de memorie corespunzătoare din *V*; după execuția tuturor instrucțiunilor din *P*, soluția / soluțiile problemei se află în anumite locații de memorie care constituie datele de ieșire *De*;
- *R - raționament de rezolvare* exprimat prin diverse tehnici și metode specifice domeniului din care face parte clasa de probleme supuse rezolvării (matematică, fizică, chimie etc.), care îmbinate cu tehnici de programare corespunzătoare realizează acțiuni / procese logice prin utilizarea memoriei virtuale *M* și a mulțimii de variabile *V*;
- *Di - date de intrare* care reprezintă valori ale unor parametri care caracterizează ipoteze de lucru / stări inițiale; valorile datelor de intrare sunt stocate în memoria *M* prin intermediul instrucțiunilor de citire / intrare care utilizează mediul de intrare *Mi*; acesta este un dispozitiv virtual pentru citirea datelor de intrare;
- *De - date de ieșire* care reprezintă valori ale unor parametri care caracterizează soluția / soluțiile problemei invocate de cerințele problemei / stările finale; valorile datelor de ieșire sunt obținute din valorile intermediare ale unor variabile generate de execuția instrucțiunilor din procesul de calcul *P* și care în final sunt stocate în memoria *M* în vederea transmiterii / scrierii lor către mediul de ieșire *Me*; acesta este un dispozitiv virtual pentru reprezentarea / scrierea datelor de ieșire sub forma grafică sau alfanumerică;
- *Mi - mediu de intrare* care este un dispozitiv virtual de intrare / citire pentru citiri virtuale ale valorilor datelor de intrare pentru a fi stocate în memoria virtuală *M*;
- *Me - mediu de ieșire* care este un dispozitiv de ieșire / scriere pentru preluarea datelor de ieșire din memoria virtuală *M* și care au fost obținute prin execuția procesului de calcul *P*; datele de ieșire sunt transmise / scrise sub formă grafică sau alfanumerică pe un suport virtual (ecran virtual, hârtie virtuală, disc magnetic virtual etc.).

Elaborarea

Elaborarea / conceperea algoritmului înseamnă construirea unui proces demonstrativ / computațional care va constitui raționamentul de rezolvare *R* și care va îngloba metode și tehnici eficiente pentru găsirea soluției / soluțiilor problemelor pentru care se elaborează algoritmul; elaborarea raționamentului de rezolvare *R* constă din următoarele:

- definirea *datelor de intrare* (*Di*) și a *datelor de ieșire* (*De*);
- aplicarea metodelor și tehnicilor utilizate pentru rezolvare;
- definirea mulțimii variabilelor *V* utilizate în rezolvare;
- codificarea raționamentului de rezolvare *R* într-un limbaj de reprezentare (de tip *pseudocod*) va constitui procesul de calcul *P*.

Reprezentarea

Prin reprezentarea algoritmului se înțelege exprimarea formalizată într-un limbaj de reprezentare (în general, de tip *pseudocod*) a legăturii dintre memoria *M*, mulțimea de variabile *V* și procesul de calcul P.

Forma generală a unui algoritm:

```
algoritm <nume_algoritm>
<declaratii_variabile>   secțiunea de declarații
start
   <procesul_de_calcul_P>   corpul algoritmului
stop
```

Execuția

Algoritmii se consideră executați pe *mașini abstracte / virtuale* (ale căror caracteristici le "abstractizează" pe cele ale mașinilor de calcul / sistemelor de calcul existente la un moment dat).

Astfel de modele de mașini de calcul sunt:

- *mașina Turing sau diverse automate cu stivă* (1956-1972, vezi A. Aho, J. Hopcroft, J.D. Ullman, *Data strutures and algorithms*, Addison Wesley publishing Co.,1983);
- *mașina MIX* (D. Knuth, *Knuth's MIX Language*, 1973);
- *mașina RAM* (A. Aho, J. Hopcroft, J.D. Ullman, *The Random Access Machine*, 1974);

- ***mașina ASM*** (***A**bstract **S**tate **M**achines - A Formal Method for Specification and Verification*);
- ***mașina VDM*** (**Bjorner, Jones**, *Vienna **D**evelopment **M**ethod*, *mașină abstractă Prolog*, 1980);
- ***mașina WAM***(*Warren **A**bstract **M**achine*, *mașină abstractă Prolog*, 1983);
- ***mașina pseudocod*** (**T. Cormen, L. Leiserson, R. Rivest,** *Pseudo-Code Language*, 1994).

Prin intermediul unei *mașini abstracte / virtuale* (care simulează o mașina reală / sistem de calcul) execuția algoritmului înseamnă interpretarea reprezentării algoritmului ca un mecanism virtual de tip dinamic care are o memorie virtuală *M*, un proces de calcul *P*, mediu de intrare *Mi*, mediu de ieșire *Me* și toate aceste componente se vor confunda cu elementele corespunzătoare mașinii virtuale, facând excepție procesul de calcul *P* care este "*lansat în execuție*" de unitatea centrală a mașinii virtuale.

Prin "*lansarea în execuție*" a procesului de calcul *P* de către mașina virtuală se înțelege exercitarea tuturor funcțiilor unității centrale (având memorie și procesor) pentru a simula toate procesările invocate de instrucțiunile procesului de calcul *P*.

Execuția algoritmului va însemna generarea de procese ce se vor desfășura în timp și care vor folosi memorie, dispozitive de calcul și dispozitive de intrare / ieșire la fel cum se întâmplă cu lansarea în execuție a formei executabile a unui program pe un sistem de calcul / calculator real.

În felul acesta, algoritmul, care este sistemul virtual *A* = (*M*, *V*, *P*, *R*, *Di*, *De*, *Mi*, *Me*), poate fi considerat ca un "*sistem de calcul*" *virtual* având componentele de bază: *M* (memorie internă), *P* (procesor), *Mi* (mediu de intrare) și *Me* (mediu de ieșire).

Ținând seama că un program este codificarea algoritmului într-un limbaj de programare, definiția unui program este asemănătoare cu definiția unui algoritm cu deosebirea că nu mai este necesară prezența raționamentului *R*, iar reprezentarea programului se va face conform sintaxei și semanticii limbajului de programare ales; pentru ambele concepte "*execuția*" evidențiază structura unui *sistem cibernetic* care se află la baza arhitecturii unui sistem de calcul (ansamblu de componente *hardware* (dispozitive) și *software* (programe) care oferă servicii utilizatorului pentru execuția programelor care implementează rezolvarea unor probleme).

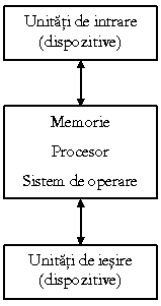

Figura 3

Mașina abstractă / virtuală va funcționa după modelul unei mașini de calcul reale.

Schema de funcționare și fluxul informațional sunt ilustrate în figura 4.

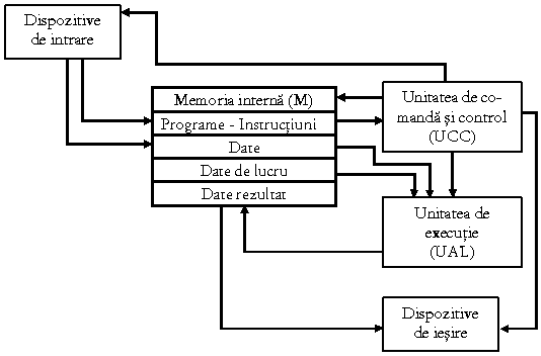

Figura 4

În concluzie, învățarea algoritmilor trebuie să fie precedată de învățarea elementelor de bază despre sistemele de calcul și să preceadă învățarea programării, și anume: *sisteme de calcul → algoritmi → programare*.

Corectitudinea și analiza

Corectitudinea algoritmului este exprimată de ***corectitudinea parțială*** (procesul de calcul se termină - timpul de execuție este finit - pentru orice date de intrare dintr-un anumit domeniu de valori) și ***corectitudinea totală*** (pentru orice date de intrare dintr-un domeniu de valori, procesul de calcul determină valori corecte conform scopului / funcției algoritmului).

Există diverse metode pentru verificarea celor două componente ale corectitudinii (de exemplu, pentru *algoritmul lui Euclid* reprezentat corect în pseudocod, ***corectitudinea parțială*** este verificată de faptul că șirul valorilor resturilor obținute din împărțirile succesive este un șir convergent către 0, iar ***corectitudinea totală*** este verificată de *metoda lui Euclid* printr-o simplă demonstrație matematică).

Elaborarea aplicațiilor informatice necesită testarea programelor care implementează diverși algoritmi pentru ca programatorul să se convingă de corectitudinea algoritmilor concepuți.

Analiza algoritmului se referă la *spațiul de memorie* utilizat și la *timpul necesar executării algoritmului*; această analiză constă în măsurarea și descrierea (cantitativă) a *performanțelor* algoritmului. Astfel, este posibilă compararea diverselor soluții algoritmice pentru aceeași problemă.

De regulă, resursa ***timp*** este mai critică (importantă) decât resursa ***spațiu de memorie***. Cu toate acestea, există numeroase situații în activitatea de proiectare și implementare a unor noi algoritmi, când se stabilește un compromis între cerințele de spațiu de memorie și cele referitoare la timpul de execuție.

Analiza unui algoritm cuprinde următoarele abordări:
- stabilirea unui *model de calcul*;
- stabilirea *timpului de calcul / execuție*;
- stabilirea *ratei de creștere.*

Modelul de calcul

Acest model va specifica *operațiile fundamentale* (*elementare*) pe care le utilizează algoritmul și costul (în unități de timp) asociat fiecărei operații elementare. În continuare vom prezenta câteva exemple:
- algoritmii numerici sunt analizați prin numărarea operațiilor aritmetice mai costisitoare (costul unei înmulțiri sau al unei împărțiri este mult mai mare decât cel al unei adunări sau al unei scăderi);
- algoritmii de sortare sau căutare sunt analizați prin numărarea operațiilor de comparație;
- algoritmii din geometria computațională, care realizează operații asupra poligoanelor, sunt analizați prin numărarea vârfurilor sau muchiilor prelucrate.

Timpul de execuție

Există așa-numitele *măsuri de complexitate* care descriu aspectul de performanță care trebuie măsurat: timpul de execuție în *cazul cel mai defavorabil*, în *cazul mediu* și în *cazul amortizat* (marginea superioară în cazul cel mai defavorabil).

Aceste măsuri de complexitate depind de volumul datelor de intrare.

Rata de creștere

Prin analiza asimptotică se stabilește rata de creștere a timpului de execuție în funcție de volumul setului de date de intrare.

Analiza asimptotică exprimă creșterea timpului de execuție al unui algoritm în cazul creșterii (spre infinit) a volumului setului de date de intrare.

Timpul de execuție este exprimat de funcția $f(n)$, n fiind numărul de elemente de intrare, iar rata de creștere a funcției $f(n)$ este dată de o funcție notată prin $T(n)$.

De exemplu, dacă $f(n) = a \cdot n^2 + b \cdot n + c$, unde a, b și c sunt constante, atunci când n crește spre infinit, termenii de ordin 1 și 0 sunt nesemnificativi, și astfel, rata de creștere a lui $f(n)$ este funcția $T(n) = n^2$.

În funcție de variația funcției $T(n)$ există următoarele categorii de algoritmi: *liniari*, *pătratici*, *cubici*, *exponențiali*, *logaritmici*, *liniar-logaritmici* etc.

Notațiile utilizate în analiza asimptotică a algoritmilor sunt următoarele:
- $O(f(n))$ care reprezintă clasa funcțiilor care cresc mai puțin decât $f(n)$ când n tinde la infinit;
- $o(f(n))$ care reprezintă clasa funcțiilor care cresc strict mai lent decât $f(n)$ când n tinde la infinit;
- $\Omega(f(n))$ care reprezintă clasa funcțiilor care nu cresc mai încet decât $f(n)$ când n tinde la infinit;
- $\Theta(f(n))$ care reprezintă clasa funcțiilor care cresc cu aceeași rată ca și $f(n)$ când n tinde la infinit.

Bibliografie

1. Albeanu Gr., *Algoritmi și limbaje de programare*, Universitatea "Spiru Haret", Editura România de Mâine, București, 2000
2. Apostol C., Roșca I. Gh., Roșca V., Ghilic-Micu B., *Introducere în programare. Teorie și aplicații*, Editura București, 1993
3. Georgescu H., *Programare concurentă. Teorie și aplicații*, Editura Tehnică, București, 1996
4. Kinston J. H., *Algorithms and Structures Design. Correctness, Analysis*, Addison Wesley, Longman, 1998
5. Mitrana V., *Provocarea algoritmilor. Probleme pentru concursurile de informatică*, Editura Agni, București, 1994
6. Odăgescu I., *Optimizarea algoritmilor*, Editura Militară, București, 1991
7. Perjeriu E., Văduva I., *Îndrumar pentru lucrări de laborator la cursul de Bazele Informaticii*, Universitatea din București, 1986
8. Popovici C., Georgescu H., State L., *Bazele Informaticii*, vol. I, Universitatea din București, 1990
9. Skiena S., *The Algorithm Design Manual*, Springer Verlag, New York, Inc., 1998
10. Vlada M., *Informatica*, Universitatea din București, Editura Ars Docendi, București, 1999
11. Vlada M., *Poligoane stelate. Problema lui Hopf și Pannwitz*, Gazeta de matematică, nr. 8/1995, pag. 339-348
12. Zaharia M. D., *Structuri de date și algoritmi. Exemple în limbajele C și C++*, Editura Albastră, Cluj-Napoca, 2002
13. Cristea V., Giumale C., Kalisz E., Păunoiu Al., *Limbajul C standard*, Editura Teora, București, 1992
14. Popovici M. D., Popovici M. I., *C++. Tehnologia orientată spre obiecte. Aplicații*, Editura Teora, București, 2000
15. www.math.gatech.edu/~thomas/FC/fourcolor.html
16. mathworld.wolfram.com/Four-ColorProblem.html
17. spicerack.sr.unh.edu/~student/tutorial/fourColor/FourColor.html
18. mathcentral.uregina.ca/RR/database/RR.09.97/fisher1.html
19. www.uccs.edu/~asoifer/book5.html
20. www.uccs.edu/~asoifer/solve98.html
21. www-groups.dcs.st-and.ac.uk/~history/BigPictures/Guthrie.jpeg
22. mathworld.wolfram.com/Algorithm.html
23. www.cs.rit.edu/~atk/Java/Sorting/sorting.html
24. faculty.juniata.edu/rhodes/cs2/ch129.htm
25. www.scism.sbk.ac.uk/law/Section5/chap1/ s5c1p2.htm
26. www.ics.uci.edu/~eppstein/161/people.html
27. www.cs.pitt.edu/~kirk/algorithmcourses/
28. www.gnacademy.org/text/cc/
29. java.sun.com/docs/books/tutorial/java/concepts/
30. www.accu.org/acornsig/public/articles/oop_c.html
31. loki.cs.brown.edu:8081/webae/full.html

Domnul conf. dr. Marin Vlada este cadru didactic la Universitatea București și poate fi contactat prin e-mail la adresa vlada@chem.unibuc.ro.