Introducere

În primul curs vom ne vom familiariza cu principiile, notațiile folosite de-a lungul întregului curs. Vom începe printr-o plasare a conceptului de algoritm în raport cu problemele de calcul utilizând un exemplu clasic. Vom prezenta conceptul de pseudocod și de asemenea vom exemplifica. Vom analiza timpul de execuție al unui anume algoritm prezentând notațiile generale ce vor fi utilizate în continuare.

Algoritmi: prezentare generală

Un algoritm este o procedură bine definită ce are ca intrare un set de valori sau una singură, și va produce o ieșire sau o mulțime de ieșiri. În această formă de exprimare, un algoritm este o secvență de calcule ce transformă o valoare sau mai multe valori: intrarea, în altă valoare sau mai multe valori: ieșirea. Terminologia de algoritm este folosită în ingineria calculatoarelor, pentru a descrie o metodă de rezolvarea unei probleme.

O perspectivă structurală asupra termenului de algoritm va indica faptul că această procedură este formată dintr-o mulțime *finită* de pași unde fiecare pas este compus dintr-una sau mai multe operații. Pentru a putea fi implementate într-un limbaj recunoscut de computer, aceste operații trebuie să fie *definite* – adică trebuie să fie foarte clar ce anume trebuie executat . De asemenea, aceste operații trebuie să fie *efective*, adică o persoană să poată efectua aceste operații, în scris, intr-un timp finit.

Programul este forma "fizică" a unui algoritm exprimat folosind un anumit limbaj de programare. Atunci când trebuie dezvoltat un program complex, enorm, foarte multe bătăi de cap apar încă de la înțelegerea și definirea problemei. Mai apoi apar probleme privind mentenanța acelui program, ceea ce duce la necesitatea divizării în subprograme numite *module*. Altfel spus, programul poate fi privit ca o îmbinare a mai multor algoritmi având ca scop rezolvarea multiplelor sarcini prevăzute de dezvoltatori.

Revenind la algoritmi, putem diviza studiul acestora în mai multe etape:

1. Conceperea algoritmilor. Crearea unui algoritm este un proces ce nu va putea fi niciodată automatizat, fiind vorba de creativitatea umană ce produce *noul* conform următoarei sinteze:

   reguli (automatizabile) + creativitate(neautomatizabilă) = soluție

   Scopul acestui curs este de a prezenta tehnici fundamentale de elaborare a algoritmilor , tehnici ce pot fi folosite pentru a dezvolta algoritmi eficienți.

2. Exprimarea algoritmilor. Formularea unui algoritm trebuie să fie clară și precisă. Ca mod de structurare se folosește programarea orientare pe obiect.
3. Validarea algoritmilor. Un algoritm, după ce a fost elaborat, poate fi testat pentru a ne asigura că, indiferent de limbajul în care va fi implementat, acesta va funcționa corect.
4. Analiza algoritmilor. Pentru aceeași problemă, putem găsi mai multe rezolvări exprimate prin diverși algoritmi. Pentru a putea decide care dintre aceștia este cel mai bun, va trebui să definim ce înseamnă bun, adică o serie de criterii de apreciere a algoritmilor.
5. Testarea programelor. Odată ce avem forma finală a programului, ce conține un anumit algoritm, trecem la ultima fază și anume rularea programului. În timpul aceste rulări, sau depanări putem evidenția și ulterior corecta eventualele erori. Presupunând că această etapă este executată cu minuțiozitate, nu putem afirma totuși, că am eliminat 100% din erori. O altă metodă de testare a programul este *profiling*, sau trasarea unu program, adică executarea acestuia pe diferite mulțimi de intrare pentru a determina timpul de calcul, memoria și alte resurse utilizate. Rezultatele pot fi ulterior comparate pentru a stabili *calitatea* algoritmului dar și a modului de implementare.

Algoritmii prezentați vor fi descriși sub formă de pseudocod, și pe alocuri sub formă de schemă logică. De asemenea, se vor întâlni și implementări concrete în limbajul Java ale unor algoritmi. Pseudocodul se diferențiază de un program prin absența unor instrucțiuni concise, ce respectă sintaxa unui anumit limbaj de programare, fiind apropiat mai mult de limba vorbită. Acesta permite abstractizarea datelor, prin ignorarea necesității tratării erorilor.

Mai jos avem o simplă catalogare a algoritmilor după modul de implementare al acestora:

- recursiv-iterativ. Recursivitatea înseamnă pe scurt apelarea aceleiași funcții chiar din interiorul ei în mod repetat. Pe de cealaltă parte, un algoritm iterativ, presupune execuția succesivă a instrucțiunilor.
- serial-paralel
- deterministic-aleatoriu. Un algoritm deterministic va furniza pentru aceleași intrări un set de ieșiri care nu se va schimba la orice rulare. Pe de altă parte un algoritm aleatoriu, va produce pentru aceleași intrări, la rulări diferite, ieșiri diferite.

Exemplu introductiv

Începem studiul algoritmilor cu următoarea problemă de sortare a unei secvențe de numere. Problema sortării este foarte des întâlnită în practică și de aceea se va insista asupra acestui aspect ceva mai mult.

Iată cum putem defini problema sortării:

Intrare: O secvență de *n* numere $(a_1, a_2, \dots, a_n)$

Ieșire: O permutare $(a'_1, (a'_2, \dots, a'_n)$ a secvenței de intrare astfel încât $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Fie secvența $(31,45,58,23,45,59)$, un algoritm de sortare va returna la ieșire secvența $(23,31,45,45,58,59)$. Intrarea se numește o *instanță* a problemei de sortare. În general o instanță a unei probleme reprezintă toate intrările ce sunt necesare pentru a calcula o soluție a unei probleme. Se spune că dacă un algoritm rezolvă o problemă dată atunci el se consideră *corect*. Pe de altă parte, un algoritm incorect nu va rezolva toate instanțele unei probleme, având alte rezultate decât cele scontate. Ca exemplu introductiv am ales sortarea prin *inserție*, deoarece este simplu de înțeles și de aplicat. Vom începe cu formularea problemei, apoi descrierea algoritmului, și mai apoi testarea lui.

Se dă un șir de numere, ce nu sunt neapărat ordonate în nici un fel. Se cere sortarea acestui șir. Pseudocodul pentru acest algoritm este format dintr-o procedură denumită INSERTION-SORT ce ia un parametru, și anume $A[1..n]$ ce reprezintă un șir de *n* elemente ce trebuie sortate. Acest număr de elemente poate fi reprezentat sub forma unui număr *length[A]*. La finalul execuției procedurii INSERTION-SORT se consideră că șirul *A*, va fi sortat.

$INSERTION - SORT(A)$

1. $\boldsymbol{for}\ j \leftarrow 2\ \boldsymbol{to}\ length[A]$
2. $\quad \boldsymbol{do}\ key \leftarrow A[j]$
3. $\quad \triangleright\ insert\ A[j]\ into\ the\ sorted\ sequence\ A[i..j-1]$
4. $\quad i \leftarrow j - 1$
5. $\quad \boldsymbol{while}\ i > 0\ and\ A[i] > key$
6. $\quad\quad \boldsymbol{do}\ A[i+1] \leftarrow A[i]$
7. $\quad\quad i \leftarrow i - 1$
8. $\quad A[i+1] \leftarrow key$

Acesta este pseudocodul ce descrie algoritmul de sortare prin inserție. Mai jos avem un exemplu pentru a testa exprimarea algoritmului prin acest pseudocod.

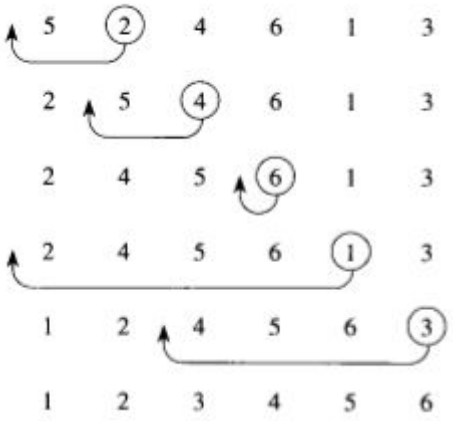


Figura 1 Operațiile de INSERTION-SORT pentru un șir $A = (5,2,4,6,1,3)$

Figura 1 prezintă modul cum lucrează acest algoritm pentru șirul ales. Practic algoritmul va parcurge întregul șir de la stânga la dreapta prin $\boldsymbol{for}\ j \leftarrow 2\ \boldsymbol{to}\ length[A]$ unde *j* este numărul curent ce va fi inserat în șirul nou format care este și sortat. Elementele $A[2..j-1]$ reprezintă elementele deja sortate iar elementele $A[j+1..n]$ sunt elementele ce urmează a fi parcurse și evident introduse în șirul final care va fi și sortat. Indexul *j* se deplasează de la stânga la dreapta prin întregul șir. La fiecare iterație din *for*, elementul $A[j]$ este selecționat. Apoi, începând cu poziția *j-1,* elementele sunt mutate cu o poziție la dreapta, până când se găsește o poziție potrivită (liniile 4-7) pentru inserarea elementului $A[j]$. Apoi acesta este inserat (linia 8).

Conventiile pentru pseudocod

Pentru a folosi corect și constant, în cele ce urmează, limbajul pseudocod, trebuie să ținem cont de anumite reguli și anume:

1. Identarea indică o structură bloc (while, if etc). De exemplu, corpul instrucțiunii **for** ce începe pe linia 1 constă din liniile 2-8, sau corpul **while-ului** ce începe pe linia 5 și conține liniile 6-7 dar fără linia 8.

   În loc de identare se pot folosi paranteze, acolade sau specificatori **begin**, **end** pentru a spori claritatea codului.
2. Construcțiile de tip *while*, *for* sau *repeat* și structurile condiționale *if*, au aceleași reprezentări ca în Pascal.

3. Simbolul ▷ indică faptul că ceea ce urmează după el este un comentariu
4. Atribuirea se face folosind operatorul „←" . de exemplu *i* ia valoarea lui *j* se va scrie $i \leftarrow j$.
5. Variabilele (de exemplu *i* sau *j*) sunt locale în procedura respectivă. Nu vom folosi variabile globale în pseudocod.
6. Șirurile se vor reprezenta prin litere mari: A,B, accesarea unui element din șir se face utilizând paranteze pătrate ca de exemplu $A[3]$. Aici a fost accesat elementul cu indexul 3. Notația ".." este folosită pentru a indica un domeniu de valori din cadrul șirului. De exemplu $A[1..j]$ indică subșirul ce este format din elementele $A[1], A[2], \dots, A[j]$.
7. Atributele unor obiecte vor fi indicate prin cuvinte în engleză cu font italic. De exemplu dacă tratăm un șir ca obiect, atunci atributul *length* va indica lungimea acestuia, adică numărul total de elemente ale șirului. Dacă dorim să reprezentăm un pointer, va trebui să marcăm acest lucru prin operația de atribuire. Dacă avem două obiecte *x* și *y,* iar operația $y \leftarrow x$ produce egalitatea $f[x] = f[y]$ unde *f* este orice atribut al acestui obiect și apoi setăm $f[x] \leftarrow 3$, iar aceste operații vor face ca și $f[y] = 3$, atunci spunem că *x* și *y* sunt pointeri ai aceluiași obiect. Un obiect null va fi marcat cu valoarea NIL.
8. Parametrii sunt transmiși prin valoare, adică procedura primește copii ale parametrilor de la apel.

Analiza algoritmilor

A analiza un algoritm poate însemna a prezice resursele necesare ca un algoritm să ruleze. Uneori ne interesează resurse ca memorie, lățime de bandă, etc. De cele mai multe ori, măsura eficienței unui algoritm este reprezentată prin timpul de calcul. Practic cel mai rapid algoritm este și cel mai eficient în cele mai multe situații.

Înainte de a analiza un algoritm, trebuie să avem un model al tehnologiei pe care va fi implementat. Vom presupune faptul că va rula pe un sistem cu un procesor, iar instrucțiunile vor fi executate una după alta, fără operații concurente.

Analiza unui algoritm implică unele cunoștințe de matematică, combinări și probabilități. în cele ce urmează, vom încerca aplicarea unui model matematic pentru a reprezenta eficiența algoritmului de inserție de mai sus.

Analiza sortării prin insertie

Timpul necesar sortării unui șir folosind această metodă, depinde de intrare: sortarea unui șir de mii de elemente este evident, mai lentă, decât sortarea unui șir cu zece elemente. Pentru a putea cuantifica acest timp, în raport cu intrarea, va trebui să definim noțiunea de *timp de rulare* și noțiunea de *intrare* mai amănunțit.

Noțiunea de *mărime de intrare* depinde de problema studiata. De exemplu înmulțirea a doi întregi, este în strânsă relație cu numărul de biți necesari reprezentării acelor numere. Pe de altă parte sortarea unui șir depinde de numărul de elemente ale acelui șir.

Noțiunea de *timp de rulare* înseamnă mai degrabă numărul de operații (pași) care trebuie executate pentru a rula algoritmul. Este mai bine să folosim termenul de *pas* pentru a nu lega noțiunea de operație de o arhitectură anume. Pentru a cuantifica cât mai corect timpul de rulare, vom introduce și noțiunea de constantă ce reprezintă timpul necesar execuției unei instrucțiuni.

De exemplu pentru a executa linia *i*, putem preciza constanta c_i ca fiind timpul necesar execuției acelei linii. Vom denumi această constantă, *costul* unei operații. În continuare vom analiza, pornind de la algoritmul descris anterior timpul de execuție al sortării prin inserție:

$INSERTION - SORT(A)$	*cost*	*timpi*
1. $\boldsymbol{for}\ j \leftarrow 2\ \boldsymbol{to}\ length[A]$	c_1	n
2. $\quad \boldsymbol{do}\ key \leftarrow A[j]$	c_2	$n-1$
3. $\quad \triangleright\ insert\ A[j]\ into\ the\ sorted\ sequence\ A[i..j-1]$	0	$n-1$
4. $\quad i \leftarrow j-1$	c_4	$n-1$
5. $\quad \boldsymbol{while}\ i > 0\ and\ A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6. $\quad\quad \boldsymbol{do}\ A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7. $\quad\quad i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8. $\quad A[i+1] \leftarrow key$	c_8	$n-1$

Timpul de rulare al algoritmului este suma timpilor necesari rulării fiecărei operații. În cazul unei operații simple (ca 1. se va înmulți costul cu numărul de execuții al acelei operații). Numărul de execuții este ceva mai complicat în cazul operației 5. de exemplu, și anume *while*, deoarece nu putem spune cu siguranță de câte ori se va executa fiecare *while*, aceasta depinzând de **intrare** și anume de șirul de sortat. De aceea presupunem că t_j este numărul de execuții în while pentru un anumit *j*.

Putem încerca în continuare să stabilim numărul total de execuții, și ca atare timpul de rulare total $T(n)$.

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$.

În cel mai bun caz, și anume când șirul de intrare este deja sortat instrucțiunea 5. este executată doar pentru a efectua verificare, și nu se vor executa niciodată instrucțiunile 6. și 7. În acest caz putem calcula $T(n)$ considerând $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} 1 = n - 1$.

În cazul acesta timpul total de rulare va fi:

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$.

Aceasta poate fi scrisă sub forma $an + b$ unde *a* și *b* sunt constante ce depind de c_i. Aceasta înseamnă că $T(n)$ este o funcție liniară de *n*.

În cazul în care șirul este sortat descrescător, calculând $T(n)$ obținem cel mai rău timp cu putință. În cazul în care șirul este sortat, *while*-ul va fi parcurs de fiecare dată, ceea ce înseamnă că t_j va fi egal cu *j*. În acest caz instrucțiunea while va fi executată de *j* ori și timpul va fi:

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

și

$$\sum_{j=2}^{n}(t_j - 1) = \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

Înlocuind în formula de calcul a lui $T(n)$ obținem

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$.

Acest timp poate fi exprimat sub forma $an^2 + bn + c$, cu a, b *şi* c depinzând de c_i. Aceasta înseamnă că funcția este una de gradul 2.

Cel mai rău caz şi cazul mediu

În analiza de mai sus a algoritmului de sortare, am calculat timpii de rulare pentru cel mai bun caz, în care șirul este deja sortat, și cel mai rău caz, în care șirul este sortat descrescător. În continuarea

cursului, ne vom axa pe căutarea celui mai rău caz, adică cel cu timpul de rulare maxim. Există trei motivații pentru această decizie:

- Cazul cel mai rău posibil este limita superioară a algoritmului pentru orice intrare. Aceasta este o garanție că algoritmul nu va dura mai mult.
- Pentru unii algoritmi, cazul cel mai rău cu putință, are loc adesea. De exemplu căutarea într-o bază de date înseamnă interogarea fiecărui element în cazul în care informația căutată nu se află în baza de date.
- Cazul mediu, este deseori la fel de nepotrivit ca și cel mai rău caz. Presupunem că vom căuta *n* numere și vom aplica sortarea prin inserție. Pentru determina locul unde inserăm elementul $A[j]$ în subșirul $A[1..j-1]$ va costa, în medie jumătate din subșir. În medie, va trebui să verificăm jumătate din subșir deci $t_j = \frac{j}{2}$. Dacă vom calcula $T(n)$ luând în considerare această presupunere, vom descoperi tot un timp pătratic, ca și în cazul cel mai rău cu putință.

Ordinul de creştere

Am simplificat unele calcule pentru a ușura cât mai mult analiza algoritmului de inserție. De exemplu, am ignorat costul concret al fiecărei instrucțiuni folosind constante de tip c_i. Apoi, în cel mai rău caz, am ales câteva constante a, b $și$ c pentru a întări faptul că timpul este pătratic. Pentru a simplifica și mai mult lucrurile vom proceda la următoarea simplificare. Se numește **ordin de creștere**, acel timp de rulare care interesează cel mai mult. Vom considera din funcția obținută, doar cel mai reprezentativ termen și anume an^2 din moment ce restul termenilor vor fi mai mici decât acesta pentru un n suficient de mare. Mai mult, vom ignora coeficientul, din moment ce aceștia sunt mai puțin semnificativi în determinarea ordinului de creștere. De aceea, obținem pentru acest algoritm, în cel mai rău caz, timpul de rulare de $\Theta(n^2)$, notație pe care o vom detalia în curând.

Vom considera că un algoritm este mai eficient decât altul dacă timpul de rulare în cel mai rău caz, are un ordin de creștere, mai mic. Această abordare poate eșua în cazul în care n este mic, dar pentru valori suficient de mari ale acestuia, un algoritm cu ordinul $\Theta(n^2)$ va fi mai bun decât unul cu ordinul $\Theta(n^3)$.

Motivele pentru care este necesar să efectuăm o analiză matematică a algoritmilor sunt:

- compararea diferiților algoritmi ce rezolvă aceeași sarcină
- predicția performanței în anumite contexte
- setarea valorii parametrilor algoritmului

Luând în considerare o serie de factori precum limbajul în care este scris algoritmul, mediul în care va rula programul ce implementează algoritmul, putem extinde analiza de baza care va fi prezentată în acest curs ,pentru a prezice cu acuratețe timpul necesar rulării unui program, pentru diferite intrări. Totuși, în cadrul acestui curs vom ignora aceste aspecte, pentru a păstra simplitatea calculului, și a ușura înțelegerea modelelor descrise. În cele ce urmează, vom studia fundamentele matematice care ne vor fi de mare folos în analiza tuturor algoritmilor ce vor fi studiați.

Operatie elementară

O operație elementară este o operație al cărei timp de execuție poate fi mărginit, superior de o constantă ce depinde doar de particularitatea implementării. Deoarece suntem interesați de timpul de execuție în limita unei constante multiplicative, vom considera numai numărul de operații elementare executate într-un algoritm nu si timpul exact al acelei operații.

Pentru a motiva această alegere vom considera următorul exemplu:testul lui Wilson de primalitate:

function Wilson(n)

return *true* dacă și numai dacă *n* este prim

if n divide ((n-1)!+1) **then return** *true*

else return *false*

Dacă vom considera calculului factorialului și testul de divizibilitate drept operații elementare, atunci eficiența este foarte mare. Dacă luăm în considerare faptul că factorialul este un algoritm cu un ordin în funcție de *n*, și anume unul foarte mare, atunci eficiența testului este redusă.

De aceea este important să definim ce înseamnă o operație elementară. Adunarea, teoretic nu este o operație elementară deoarece, depinde de lungimea operanzilor, și , ca în cazul de mai sus ar putea prezenta deficiențe. Totuși, adunarea poate fi considerată o operație elementară pentru operanzi de lungime rezonabilă. În continuare, vom considera adunările, scăderile, înmulțirile, împărțirile, operațiile booleene și atribuirile drept operații elementare.

Algoritmi fundamentali

În continuare, vom prezenta câțiva algoritmi nu foarte complicați, pentru a crea o baza și o introducere pentru acest curs și pentru a exersa conceptele de mai sus și nu numai.

Înmultirea *a la russe*

Este o operație de matematică inventată în antichitate, și este o metodă ce nu implică folosirea tabelului înmulțirii, ci doar divizarea cu 2 și adăugarea. Marele avantaj al acestei metode este că poate fi implementată ușor în tehnica de calcul modernă, deoarece implică folosirea acestor operații aritmetice de bază. În cele ce urmează vom prezenta algoritmul:

$RUSSE(A, B)$

1. $\boldsymbol{arrays}\ X, Y$
2. $X[1] \leftarrow A; Y[1] \leftarrow B$
3. $i \leftarrow 1$
4. $\triangleright\ se\ construiesc\ cele\ două\ coloane$
5. $\boldsymbol{while}\ X[i] > 1\ \boldsymbol{do}$
6. $\quad X[i+1] \leftarrow X[i]\ div\ 2$
7. $\quad Y[i+1] \leftarrow Y[i] + Y[i]$
8. $\quad i \leftarrow i + 1$
9. $prod \leftarrow 0$
10. $\triangleright\ se\ calculeaza\ produsul\ final$
11. $\boldsymbol{while}\ i > 0\ \boldsymbol{do}$
12. $\quad \boldsymbol{if}\ X[i]\ este\ impar\ \boldsymbol{then}\ prod \leftarrow prod + Y[i]$
13. $\quad i \leftarrow i - 1$
14. $\boldsymbol{return}\ prod$

Pentru a exemplifica acest algoritm alegem două numere *45* deînmulțitul și *19* înmulțitorul și vom aplica în continuare conform regulii: se împarte deînmulțitul cu 2 și se înmulțește cu 2 înmulțitorul. Se aplică regula până când deînmulțitul este 1. Pentru a calcula produsul final vom aduna toate numerele din coloana înmulțitorului care corespund, pe linie, numerelor impare de pe coloana deînmulțitului.

Iată un exemplu pentru a demonstra acest principiu, înmulțim numerele 52 și 15:

52	15	-
26	30	-
13	60	60
6	120	-
3	240	240
1	480	480

Rezultat		780

Implementarea acestui algoritm în limbaj Java, este dată mai jos:

```
public class ALaRusse
{
    public static int russe(int a, int b)
    {
        int[] X, Y;
        X = new int[1000];
        Y = new int[1000];
        X[1] = a;
        Y[1] = b;
        int i = 1;
        while (X[i] > 1)
        {
            X[i + 1] = X[i] / 2;
            Y[i + 1] = Y[i] + Y[i];
            i = i + 1;
        }
        int prod = 0;
        while (i > 0) {
            if ((X[i] % 2) != 0)
            {
                prod = prod + Y[i];
            }
            i = i - 1;
        }
        return prod;
    }

    public static void main(String[] args)
    {
        System.out.println(russe(52, 15));
    }
}
```

Motivul pentru care am prezentat implementarea concretă, este de a corela forma pseudocodului cu implementarea într-un limbaj concret, Java, și pentru a avea pentru laboratoare, un

model de urmat. Observați că mărimea șirurilor *X*, și *Y* este arbitrar aleasă și anume 1000. Aceste valori pot fi setate individual după necesitate, este recomandat să folosim valori dinamice (de exemplu introduse de cel ce rulează programul, sau citite dintr-un fisier) și nu valori “hardcoded”, ca în cazul de mai sus (numerele 1000, 52, 15 de exemplu).

Sortarea prin selectie

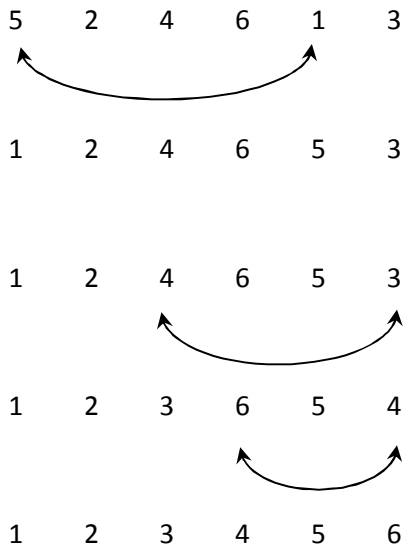
Algoritmul are ca intrare un șir de date, nu neapărat sortat, și se cere sortarea crescător a acestuia. Spre deosebire de sortarea prin inserție, sortarea prin selecție, lucrează altfel, plasând la fiecare pas un element pe poziția lui finală. Mai jos avem algoritmul în pseudocod

$SELECTION - SORT(A[1..n])$

1. $\boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ n-1\ \boldsymbol{do}$
2. $\quad minj \leftarrow i; minx \leftarrow A[i]$
3. $\quad \triangleright\ caut\ poziția\ finală\ a\ lui\ A[i]\ în\ șir$
4. $\quad \boldsymbol{for}\ j \leftarrow i+1\ to\ n\ \boldsymbol{do}$
5. $\quad\quad \boldsymbol{if}\ \ A[j] < minx\ \boldsymbol{then}$
6. $\quad\quad\quad minj\ \leftarrow j$
7. $\quad\quad\quad minx\ \leftarrow A[j]$
8. $\quad \triangleright\ și\ schimb\ elementul\ actual\ cu\ cel\ mai\ mic\ găsit$
9. $\quad A[minj] \leftarrow A[i]$
10. $\quad A[i] \leftarrow minx$

Algoritmul funcționează astfel: pornind de la primul element în șir 1. ne folosim de variabilele $minj$ și $minx$ pentru a reține poziția celui mai mic element respectiv valoarea acestuia. Apoi vom parcurge restul șirului 4. căutând cel mai mic element din subșirul care este format din elementele ce urmează elementului actual. Găsim cea mai mică valoare, reținem poziția, și facem interschimbarea cu elementul actual (daca este cazul). Continuăm până când am parcurs șirul până la penultimul element, clipă în care șirul va fi deja sortat.

Pentru exemplificare vom utiliza ca intrare același șir ca cel de la sortarea prin inserție și anume: 5 2 4 6 1 3.



Timpul de execuție al acestui algoritm este unul pătratic, independent de ordonarea inițială a a elementelor. Testul $\boldsymbol{if}\ A[j] < minx$ este executat de fiecare dată, indiferent de șirul de intrare.

Calculul șirului lui Fibonacci

Șirul lui Fibonacci este definit prin următoarea recurență:

$$\begin{cases} f_0 = 0;\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2}\ pentru\ n \geq 2 \end{cases}$$

Acest șir a fost descoperit de Leonardo Pisano, cunoscut sub numele Leonardo Fibonacci. Cel de-al *n*-lea termen din șir se poate scrie folosind definiția și anume:

$FIBONACCI1(n)$

1. $\boldsymbol{if}\ n < 2\ \boldsymbol{then\ return}\ n$
2. $\boldsymbol{else\ return}\ FIBONACCI1(n-1) + FIBONACCI1(n-2)$

Metoda este extrem de ineficientă, deoarece recalculează de mai multe ori aceleași valori, timpul fiind unul exponențial. Acest algoritm introduce noțiunea de recursivitate, asupra căreia vom insista ulterior în curs.

O altă metoda mai eficientă de a rezolva această problemă, și anume într-un timp liniar, este prezentată mai jos:

$FIBONACCI2(n)$

1. $i \leftarrow 1; j \leftarrow 0$
2. $\boldsymbol{for}\ k \leftarrow 1\ \boldsymbol{to}\ n\ \boldsymbol{do}$
3. $\quad j \leftarrow i + j$
4. $\quad i \leftarrow j - i$
5. $\boldsymbol{return}\ j$

Algoritmul lui Euclid pentru calculul celui mai mare divizor comun

Algoritmul lui Euclid este o metodă eficientă pentru calculul cmmdc-ului a două numere. Se bazează pe principiul calculului acestui număr, și anume că divizorul a două numere nu se schimbă dacă extragem numărul cel mai mic din cel mai mare. De exemplu cmmdc-ul lui 102 si 18 este 6. Dacă extragem 18 din 102 obținem 84, iar cmmdc-ul lui 18 și 84 este tot 6. Algoritmul lui Euclid se bazează pe această proprietate și spune că cmmdc-ul a două numere se poate afla astfel: se află restul împărțirii celui mai mare număr la cel mai mic și se reține acest rest, câtul și vechiul deîmpărțit devin noile numere cărora le aplicăm aceeași operație, până când restul devine zero. Penultimul rest este cmmdc-ul numerelor inițiale. Mai jos este pseudocodul acestui algoritm.

$EUCLID(m, n)$

1. $\boldsymbol{while}\ n \neq 0\ \boldsymbol{do}$
2. $\quad temp \leftarrow n$
3. $\quad n \leftarrow restul\ impărțirii\ lui\ m\ la\ n$
4. $\quad m \leftarrow temp$
5. $\boldsymbol{return}\ m$

Pentru exemplul ales și anume 102 și 18 algoritmul va funcționa astfel:

$temp \leftarrow 18$

$n \leftarrow 12$

$m \leftarrow 18$

$\quad temp \leftarrow 12$

$\quad n \leftarrow 6$

$\quad m \leftarrow 12$

$\quad\quad temp \leftarrow 6$

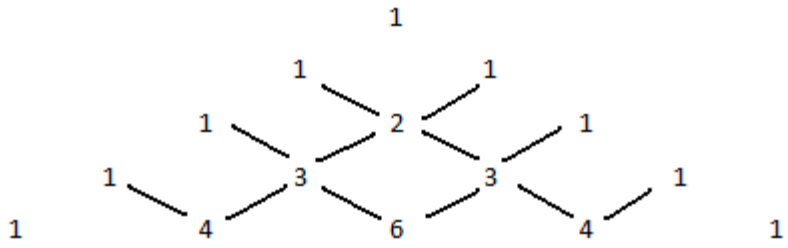
$\quad\quad n \leftarrow 0$

$\quad\quad \boldsymbol{m \leftarrow 6}$

Există și alte variante ale acestui algoritm pe care le veți aprofunda în cadrul laboratoarelor.

Triunghiul lui Pascal

Inventat de matematicianul Blaise Pascal, triunghiul reprezintă un aranjament geometric al coeficienților binomiali. Regula formării acestui triunghi este următoarea: pornind de la numărul 1 vom dispune pe linii succesive numere formate din suma numerelor alăturate de pe linia anterioară. Astfel dacă pentru primul rând avem 1, pentru al doilea rând obținem 0+1=1.
Mai jos avem primele cinci rânduri din acest triunghi și sumele efectuate.



Coeficienții binomiali sunt dați de exprimarea sumei:

$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$

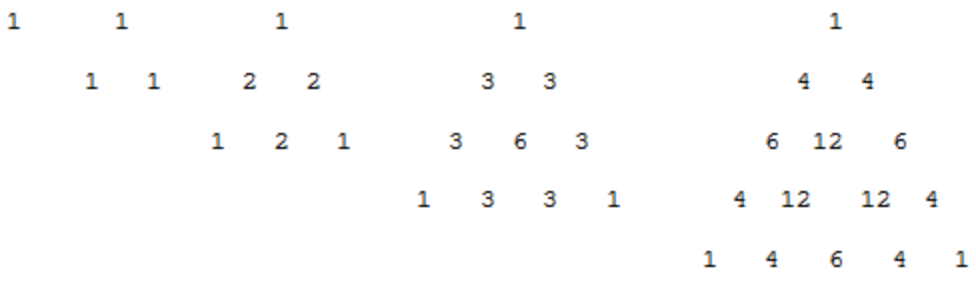
În acest caz, are loc

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

pentru orice *n* și *k* întregi, iar $k \in (0, n]$

Amintim că formula de calcul a combinărilor este $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Piramida lui Pascal este generalizarea pe trei dimensiuni a triunghiului lui Pascal. Ea este construită strat cu strat, fiecare muchie fiind un rând corespunzător din triunghiul lui Pascal iar interiorul fiind suma învelișurilor de deasupra. Iată mai jos patru secțiuni transversale prin piramidă.



Pornind de la acest triunghi există o serii de aplicații, cum ar fi serii matematice, trigonometrie etc. Asemănător cu acest triunghi este algoritmul de transformare de distanță, cu aplicații în procesarea imaginilor.

Înmultirea matricelor

Pe parcursul acestui curs, vor fi folosite des operațiile cu matrice, ca atare, pentru introducere vom prezenta una din operațiile frecvent utilizate și anume înmulțirea lor.

Condiția de bază pentru a putea efectua acest produs este ca *lățimea* primei matrice sa fie egală cu *înălțimea* celei de-a doua. Multiplicând o matrice $m \times n$ cu una $n \times p$ unde m, n, p nu sunt neapărat egale, rezultă o matrice $m \times p$.

Iată un exemplu de efectuare a acestui produs:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \end{bmatrix}$$

Elementul c_{24} se calculează după cum urmează:

$$c_{24} = (a_{21}, a_{22}) \cdot (b_{14}, b_{24}) = a_{21} \times b_{14} + a_{22} \times b_{24}$$

Parcurgerea unei matrice se realizează conform pseudocodului:

$TRAVERSEZ(A)$

1. $\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ m\ \boldsymbol{do}$
2. $\quad \boldsymbol{for}\ \boldsymbol{j} = 1\ \boldsymbol{to}\ n\ \boldsymbol{do}$
3. $\quad\quad \triangleright\ acesez\ \ valoarea\ elementul\ A[i,j]$

Fiecare element al produsului final va fi format pe baza relației:

$$(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

Pentru a realiza acest produs va trebui să parcurgem atât matricea A cât și B în același timp, si pe de altă parte să calculăm suma formată din *n* elemente.

Mai concret, produsul a doua matrice de dimensiuni $m \times n$ si $n \times p$ se va calcula în acest fel:

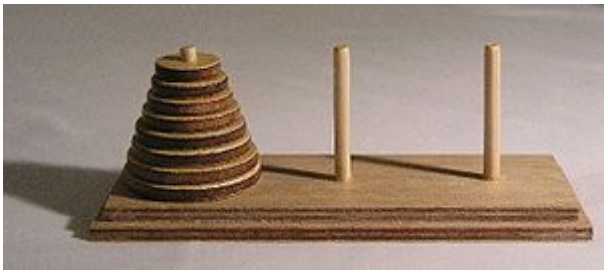
PRODUS(A,B)

1. $\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ m\ \boldsymbol{do}$
2. $\quad \boldsymbol{for}\ j = 1\ \boldsymbol{to}\ p\ \boldsymbol{do}$
3. $\quad\quad \boldsymbol{for}\ k = 1\ \boldsymbol{to}\ n\ \boldsymbol{do}$
4. $\quad\quad\quad C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$
5. $\boldsymbol{return}\ C$

Prin această abordare, am respectat formula de calcul a produsului. Eficiența este punctul slab al acestei abordări, deoarece ordinul de timp al acestui algoritm este n^3. Vom vedea, mai târziu că există un algoritm inventat de Volker Strassen ce reduce complexitatea.

Turnurile din Hanoi

Acesta este mai mult un joc matematic și constă din trei tije și un număr de N discuri de mărimi diferite care se pot plasa pe orice tijă ca în imaginea de mai jos.



Jocul pornește din starea din figură, adică toate discurile pe poziția din stânga, respectiv sursă.

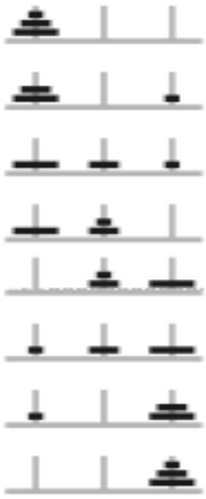
Scopul este de a muta toate discurile pe tija din dreapta, numită și destinație, utilizând tija din mijloc pentru mutări temporare. De aceea tija din mijloc se mai numește și auxiliar.

Regulile sunt următoarele:

- Doar un singur disc poate fi mutat la un pas.
- Mutarea constă în a lua discul din vârful stivei de pe o tijă și plasarea lui pe altă tijă.
- Nici un disc nu poate fi plasat deasupra unuia mai mic decât el.

Există mai multe moduri de a rezolva acest algoritm și vom aborda problema din punct de vedere recursiv și iterativ, deocamdată.

In figura de mai jos este dată soluția vizuală pentru N=3



Soluția recursivă presupune separarea problemei într-o colecție de probleme mai mici și mai departe a acestora în alte probleme mai mici până când a fost găsită soluția. Iată cum se poate rezolva problema din acest punct de vedere:

- se etichetează tijele cu A,B și C. Această notare este una inițială deoarece notările se vor schimba, spre exemplu dacă A era eticheta pentru prima tijă, mai târziu va desemna alta tijă.
- fie *n* numărul de discuri
- numerotăm discurile de la 1 – cel mai mic, la n – cel mai mare

Pașii algoritmului recursiv, pentru a muta toate *n* discurile de pe A pe C sunt:

1. mută *n*-1 discuri de pe A pe N. Al *n*-lea disc rămâne pe A.
2. mută al *n*-lea disc de pe A pe C
3. mută *n*-1 de pe tija B pe C astfel încât toate vor di deasupra discului *n*.

Algoritmul recursiv este:

HANOI(n,A,C,B)

1. $\boldsymbol{if}\ n \neq 0\ \boldsymbol{then}$
2. $HANOI(n-1, A, B, C)$
3. $\boldsymbol{a}fisez\ \ll Mută\ discul\ de\ pe \gg A\ \ll pe \gg C$
4. $HANOI(n-1, B, C, A)$

Soluția iterativă presupune crearea unei liste de mutări care vor fi efectuate după cum urmează. Contorizăm mutările pornind de la 1, numărul discului mutat este *m*. Mutările cu numere impare implică discul cel mai mic. Se poate observa că discul cel mai mic parcurge A, C,B, A,C,B, etc pentru un număr de discuri impar, și A,B,C,A,B,C, etc pentru un număr par de discuri. Următorul algoritm este mai simplu din acest punct de vedere ca și cel recursiv:

- Mută cel mai mic disc pe tija care nu a fost mai fost vizitată de cel mai mult timp.
- Mută alt disc astfel ca mutarea să fie una legală (doar o singură mutare va fi disponibilă)

Se pot observa următoarele aspecte:

1. Discurile a căror numere sunt pare se deplasează în același sens cu discul cel mai mic.
2. Discurile a căror numere sunt impare se deplasează în sens invers.
   Aplicații ale acestui tip de probleme sunt nenumărate, iar înțelegerea acestui fenomen va ajuta în diferite operații cu numere în baza doi, și nu numai.