1. Algoritmul și etapele rezolvării unei probleme

> **Definiție.** Un algoritm este o succesiune finită și bine determinată de pași prin care, pornind de la datele de intrare, se obțin datele de ieșire.

1.1. Datele unei probleme

Felul datelor	Ce sunt
date de intrare	valorile cunoscute, care se citesc
date de ieșire	rezultatele cerute, care se afișează
date de manevră	valorile intermediare, folosite doar în timpul calculului

1.2. Etapele rezolvării

1. analiza enunțului și stabilirea datelor;
2. elaborarea algoritmului;
3. reprezentarea algoritmului în pseudocod;
4. implementarea într-un limbaj de programare;
5. compilarea și rularea;
6. testarea și depanarea.

1.3. Însușirile unui algoritm

Însușire	Ce înseamnă
claritate	fiecare pas este fără dubiu
finitudine	se termină după un număr finit de pași
generalitate	rezolvă toate cazurile problemei, nu unul singur
corectitudine	dă rezultatul cerut pentru orice date valide
eficiență	se încadrează în timpul și memoria disponibile

> Anul acesta se adaugă subprogramele și recursivitatea. Toți algoritmii de mai jos vor fi rescriși ca subprograme: în loc să copiezi testul de primalitate de fiecare dată, îl scrii o dată și îl apelezi.

---

2. Date, operatori, expresii

2.1. Tipuri simple de date

Tip	Exemple	Utilizare
`int`	`-12, 0, 35`	numere întregi
`long long`	`4000000000LL`	numere întregi mari
`float`	`2.5f`	numere reale
`double`	`3.14159`	numere reale, precizie mai bună
`char`	`'A'`	un caracter
`bool`	`true, false`	valori logice

2.2. Operatorii

Categorie	Operatori
aritmetici	`+ - * / %`
relaționali	`< <= > >= == !=`
logici	`&& || !`
de atribuire	`= += -= *= /=`
de incrementare	`++ --`

> Pentru operanzi întregi, `/` face împărțirea întreagă, iar `%` dă restul. Cele două operații stau la baza aproape tuturor algoritmilor care urmează.

2.3. Citirea și afișarea

```
#include <iostream>
using namespace std;

int main() {
    int a, b;
    cin >> a >> b;

    cout << "Suma este " << a + b << '\n';
    cout << "Catul intreg este " << a / b << '\n';
    cout << "Restul este " << a % b;

    return 0;
}
```

3. Structurile programării structurate

Orice algoritm se construiește din trei structuri, care se pot combina și imbrica: **liniară**, **alternativă** și **repetitivă**.

3.1. Structura alternativă

În pseudocod:

```
dacă a > b atunci
    scrie a
altfel
    scrie b
sfârșit dacă
```

Program — maximul a trei numere:

```
#include <iostream>
using namespace std;

int main() {
    int a, b, c;
    cin >> a >> b >> c;

    int maxim;

    if (a >= b && a >= c)
        maxim = a;
    else if (b >= a && b >= c)
        maxim = b;
    else
        maxim = c;

    cout << maxim;

    return 0;
}
```

3.2. Structurile repetitive

Situație	Structură potrivită
numărul repetărilor este cunoscut dinainte	`for`
se repetă cât timp condiția este adevărată	`while`
corpul trebuie executat cel puțin o dată	`do…while`

---

4. Algoritmi pentru interschimbare

Interschimbarea valorilor a două variabile nu se poate face prin simpla atribuire a noilor valori: `a = b` urmat de `b = a` pierde valoarea veche a lui `a`, iar la final ambele variabile au aceeași valoare.

4.1. Cu variabilă intermediară

```
int a, b, x;
cin >> a >> b;

x = a;      // se salveaza prima valoare
a = b;      // prima primeste valoarea celei de-a doua
b = x;      // a doua primeste valoarea salvata

cout << a << ' ' << b;
```

4.2. Fără variabilă intermediară

Se folosesc identitățile `a = (a - b) + b` și `b = ((a - b) + b) - (a - b)`: valoarea `a - b` se atribuie întâi lui `a`.

```
int a, b;
cin >> a >> b;

a = a - b;
b = a + b;      // b = (a-b) + b = a vechi
a = b - a;      // a = a vechi - (a-b) = b vechi

cout << a << ' ' << b;
```

> Varianta a doua arată isteață, dar în practică se folosește prima: pentru numere mari, `a - b` sau `a + b` poate depăși tipul, iar rezultatul iese greșit.

4.3. Aplicație: numărul minim din cifrele unui număr

Se citește un număr natural de trei cifre. Cifrele se extrag, se ordonează crescător prin interschimbări, apoi se compune numărul nou. Ordonarea a trei valori se face prin trei comparații.

```
#include <iostream>
using namespace std;

int main() {
    int n, a, b, c, x;
    cin >> n;

    a = n / 100;            // cifra sutelor
    b = (n / 10) % 10;      // cifra zecilor
    c = n % 10;             // cifra unitatilor

    if (a > b) { x = a; a = b; b = x; }
    if (b > c) { x = b; b = c; c = x; }
    if (a > b) { x = a; a = b; b = x; }

    n = a * 100 + b * 10 + c;
    cout << n;

    return 0;
}
```

> Pentru 312 se afișează 123. A treia comparație nu e de prisos: după ce `b` și `c` se schimbă între ele, `a` poate ajunge din nou mai mare decât `b`.

---

5. Algoritmi pentru determinarea maximului (minimului)

Primul număr citit devine valoarea de pornire, iar celelalte se compară cu ea. Nu se pornește de la 0: un șir numai cu valori negative ar da maximul 0, care nu apare în șir.

5.1. Când se știe câte numere sunt

```
#include <iostream>
using namespace std;

int main() {
    int n, a, maxim;
    cin >> n;

    cin >> a;
    maxim = a;              // primul numar da valoarea de pornire

    for (int i = 2; i <= n; i++) {
        cin >> a;
        if (a > maxim)
            maxim = a;
    }

    cout << maxim;

    return 0;
}
```

5.2. Când se citește până la 0

```
#include <iostream>
using namespace std;

int main() {
    int a, maxim;

    cin >> a;
    maxim = a;

    while (a != 0) {
        if (a > maxim)
            maxim = a;
        cin >> a;
    }

    cout << maxim;

    return 0;
}
```

5.3. Maximul și de câte ori apare

Contorul `k` se pune pe 1 la fiecare maxim nou și crește doar la egalitate.

```
#include <iostream>
using namespace std;

int main() {
    int n, a, maxim, k;
    cin >> n;

    cin >> a;
    maxim = a;
    k = 1;

    for (int i = 2; i <= n; i++) {
        cin >> a;

        if (a == maxim)
            k++;
        else if (a > maxim) {
            maxim = a;
            k = 1;          // maxim nou: numaratoarea o ia de la capat
        }
    }

    cout << maxim << " apare de " << k << " ori";

    return 0;
}
```

> Ordinea celor două ramuri contează, iar a doua trebuie să fie `else if`, nu un `if` separat: altfel, la un maxim nou, contorul se pune pe 1 și imediat crește din nou.

---

6. Algoritmi pentru prelucrarea cifrelor unui număr

Trei algoritmi, construiți din aceleași două operații:

- `n % 10` — dă cifra cea mai nesemnificativă;
- `n / 10` — o elimină din număr.

6.1. Extragerea cifrelor

```
int n, c;
cin >> n;

while (n != 0) {
    c = n % 10;         // se extrage cifra cea mai nesemnificativa
    cout << c << ' ';   // se prelucreaza
    n = n / 10;         // se elimina din numar
}
```

Urmărim algoritmul pentru `n = 4567`:

Pas	c = n % 10	n = n / 10
1	`7`	456
2	`6`	45
3	`5`	4
4	`4`	0 — bucla se oprește

Suma și produsul cifrelor

```
#include <iostream>
using namespace std;

int main() {
    int n, s = 0, p = 1;
    cin >> n;

    while (n != 0) {
        s = s + n % 10;
        p = p * (n % 10);
        n = n / 10;
    }

    cout << "Suma = " << s << ", produsul = " << p;

    return 0;
}
```

> Suma pornește de la 0, produsul de la 1. Un produs pornit de la 0 rămâne 0, orice s-ar înmulți după aceea.

6.2. Compunerea unui număr din cifrele sale

Cifrele se citesc începând cu cea mai semnificativă. La fiecare pas, ce s-a construit până atunci se mută cu o poziție la stânga și se adaugă cifra nouă.

```
#include <iostream>
using namespace std;

int main() {
    int c, nr = 0;

    cin >> c;
    while (c >= 0 && c <= 9) {      // cat timp se citeste o cifra
        nr = nr * 10 + c;
        cin >> c;
    }

    cout << nr;

    return 0;
}
```

6.3. Inversul unui număr

```
#include <iostream>
using namespace std;

int main() {
    int n, inv = 0;
    cin >> n;

    while (n != 0) {
        inv = inv * 10 + n % 10;
        n = n / 10;
    }

    cout << inv;

    return 0;
}
```

> Un număr care se termină în 0 are inversul mai scurt: pentru 1230 se obține 321, nu 0321 — nu există numere scrise cu zerouri în față.

6.4. Aplicație: palindrom

Un număr este palindrom dacă este egal cu inversul lui. Aici apare capcana clasică: bucla îl consumă pe `n`, deci înainte de ea trebuie păstrată o copie.

```
#include <iostream>
using namespace std;

int main() {
    int n, copie, inv = 0;
    cin >> n;

    copie = n;              // n se pierde in timpul calculului

    while (copie != 0) {
        inv = inv * 10 + copie % 10;
        copie = copie / 10;
    }

    if (n == inv)
        cout << "palindrom";
    else
        cout << "nu este palindrom";

    return 0;
}
```

---

7. Algoritmi pentru calcularea c.m.m.d.c.

7.1. Algoritmul lui Euclid, cu împărțiri

Cel mai mare divizor comun a două numere îl divide și pe restul împărțirii lor. Deci perechea se poate înlocui cu una mai mică, până când restul devine 0.

```
citește a, b
cât timp b <> 0 execută
    r ← a mod b
    a ← b
    b ← r
sfârșit cât timp
scrie a
```

```
#include <iostream>
using namespace std;

int main() {
    int a, b, r;
    cin >> a >> b;

    while (b != 0) {
        r = a % b;
        a = b;
        b = r;
    }

    cout << "cmmdc = " << a;

    return 0;
}
```

Urmărim algoritmul pentru a = 18 și b = 12:

a	b	r = a % b
`18`	`12`	6
`12`	`6`	0
`6`	`0`	— bucla se oprește, cmmdc = 6

7.2. Algoritmul cu scădere repetată

```
#include <iostream>
using namespace std;

int main() {
    int a, b;
    cin >> a >> b;

    while (a != b)
        if (a > b)
            a = a - b;
        else
            b = b - a;

    cout << "cmmdc = " << a;

    return 0;
}
```

> Varianta cu scăderi se blochează dacă unul dintre numere este 0: condiția `a != b` nu se atinge niciodată. Varianta cu împărțiri nu are problema asta.

7.3. Cel mai mic multiplu comun

Nu se calculează separat, ci din c.m.m.d.c., folosind că produsul celor două numere este egal cu produsul dintre c.m.m.d.c. și c.m.m.m.c.

```
#include <iostream>
using namespace std;

int main() {
    int a, b, x, y;
    cin >> a >> b;

    x = a;                  // a si b se pierd, deci lucram pe copii
    y = b;

    while (y != 0) {
        int r = x % y;
        x = y;
        y = r;
    }

    cout << "cmmdc = " << x << '\n';
    cout << "cmmmc = " << 1LL * (a / x) * b;

    return 0;
}
```

> Se împarte întâi și abia apoi se înmulțește — `(a / x) * b`, nu `a * b / x`: așa produsul rămâne mic și nu depășește tipul. Pentru siguranță se lucrează tot în `long long`.

---

8. Algoritmi pentru testarea unui număr prim

Un număr prim are exact doi divizori: 1 și el însuși. Se caută primul divizor între 2 și radical din `n`; dacă nu există niciunul, numărul este prim.

```
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;

    bool prim = true;

    if (n < 2)
        prim = false;

    for (int i = 2; i <= n / i; i++)
        if (n % i == 0) {
            prim = false;
            break;
        }

    if (prim)
        cout << "Numarul este prim";
    else
        cout << "Numarul nu este prim";

    return 0;
}
```

> Condiția se scrie `i <= n / i`, nu `i <= sqrt(n)`: împărțirea se face între întregi și nu are erori de rotunjire, iar `i * i` ar putea depăși tipul `int` pentru valori mari.

8.1. Optimizarea: se sar numerele pare

Dacă `n` nu se divide cu 2, nu se divide nici cu vreun alt număr par. Deci după ce se tratează separat cazul lui 2, șirul divizorilor încercați poate merge din 2 în 2, ceea ce înjumătățește numărul de pași.

```
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;

    bool prim = true;

    if (n < 2)
        prim = false;
    else if (n > 2 && n % 2 == 0)
        prim = false;                    // numerele pare, in afara de 2
    else
        for (int i = 3; i <= n / i; i = i + 2)
            if (n % i == 0) {
                prim = false;
                break;
            }

    if (prim)
        cout << "Numarul este prim";
    else
        cout << "Numarul nu este prim";

    return 0;
}
```

> `break` nu e un rafinament: fără el, un număr cu mulți divizori continuă să fie testat degeaba, iar la un fișier cu un milion de numere diferența se vede.

---

9. Algoritmi pentru prelucrarea divizorilor unui număr

9.1. Toți divizorii

1 și `n` sunt divizori pentru orice număr, iar ceilalți nu pot depăși `n / 2`:

```
int n;
cin >> n;

cout << 1 << ' ' << n << ' ';

for (int i = 2; i <= n / 2; i++)
    if (n % i == 0)
        cout << i << ' ';
```

9.2. Varianta cu radical

Divizorii vin în perechi: dacă `i` îl divide pe `n`, atunci și `n / i` îl divide. Pentru 36:

Divizorul mic	Perechea lui
`1`	`36`
`2`	`18`
`3`	`12`
`4`	`9`
`6`	`6`

În fiecare pereche, unul dintre numere este cel mult radical din `n`. E de ajuns să căutăm până acolo, iar perechea o obținem prin împărțire:

```
int n;
cin >> n;

for (int i = 1; i <= n / i; i++)
    if (n % i == 0) {
        cout << i << ' ';

        if (i != n / i)         // la 36, perechea lui 6 e tot 6
            cout << n / i << ' ';
    }
```

9.3. Divizorii primi

Se elimină din număr toate puterile fiecărui divizor găsit. Atunci următorul divizor găsit nu mai poate fi compus, deci este prim.

```
int n;
cin >> n;

int i = 2;

while (n > 1) {
    if (n % i == 0) {
        cout << i << ' ';

        while (n % i == 0)      // se elimina toate puterile lui i
            n = n / i;
    }
    i++;
}
```

9.4. Descompunerea în factori primi

Aceeași idee, dar se numără și de câte ori se împarte:

```
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;

    int i = 2;

    while (n > 1) {
        if (n % i == 0) {
            int k = 0;

            while (n % i == 0) {
                k++;
                n = n / i;
            }

            cout << i << " la puterea " << k << '\n';
        }
        i++;
    }

    return 0;
}
```

> Pentru n = 20 se afișează `2 la puterea 2` și `5 la puterea 1`.

---

10. Algoritmi pentru conversii între sisteme de numerație

10.1. Din baza 10 în baza q

Numărul se împarte întreg la `q` până când câtul devine mai mic decât `q`. Resturile obținute sunt cifrele reprezentării în baza `q`, primul rest fiind cifra cea mai puțin semnificativă.

```cpp
#include <iostream>
using namespace std;

int main() {
    int n10, q;
    cin >> n10 >> q;

    int nq = 0, p = 1;

    while (n10 != 0) {
        nq = nq + p * (n10 % q);    // restul e cifra din baza q
        n10 = n10 / q;
        p = p * 10;                 // se aseaza cu o pozitie mai la stanga
    }

    cout << nq;

    return 0;
}
```

Urmărim conversia lui 11 în baza 2:

Pas	Restul	nq	n10
1	`c = 11 mod 2 = 1`	`1`	5
2	`c = 5 mod 2 = 1`	`11`	2
3	`c = 2 mod 2 = 0`	`011`	1
4	`c = 1 mod 2 = 1`	`1011`	0 — gata

> Rezultatul `nq` nu e numărul în baza q, ci un număr scris în baza 10 care are aceleași cifre. Pentru q mai mare decât 10 metoda nu mai merge — ar trebui litere, deci un șir de caractere.

10.2. Din baza q în baza 10

Cifrele se citesc începând cu cea mai semnificativă, iar numărul se construiește ca la compunerea din cifre, doar că se înmulțește cu `q`, nu cu 10.

```
#include <iostream>
using namespace std;

int main() {
    int q, c;
    cin >> q;

    int n10 = 0;

    cin >> c;
    while (c >= 0 && c < q) {       // cat timp e cifra valida in baza q
        n10 = n10 * q + c;
        cin >> c;
    }

    cout << n10;

    return 0;
}
```

---

11. Algoritmi pentru generarea șirurilor recurente

Într-un șir definit prin recurență, fiecare termen se calculează din cei dinaintea lui. La Fibonacci, primii doi termeni sunt dați, iar fiecare următor este suma celor doi precedenți: 1, 1, 2, 3, 5, 8, 13, 21…

```
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;

    long long a1 = 1, a2 = 1, a3;

    if (n >= 1) cout << a1 << ' ';
    if (n >= 2) cout << a2 << ' ';

    for (int i = 3; i <= n; i++) {
        a3 = a1 + a2;
        cout << a3 << ' ';
        a1 = a2;            // fereastra aluneca
        a2 = a3;
    }

    return 0;
}
```

Nu se folosește niciun vector: la orice moment ne trebuie doar ultimii doi termeni. După ce s-a calculat `a3`, fereastra alunecă.

> Ordinea celor două atribuiri contează. Scrise invers — mai întâi `a2 = a3` — se pierde valoarea veche a lui `a2`, iar șirul iese greșit de la al patrulea termen.

> Termenii cresc repede: al 47-lea nu mai încape în `int`. De aceea se lucrează în `long long`.

---

12. Eficiența algoritmilor

Pentru aceeași problemă se pot scrie mai mulți algoritmi. Cel mai eficient este cel care folosește cele mai puține resurse:

Resursa	Ce se măsoară	Regula
Memoria internă	câte variabile și cât de mari sunt tipurile alese pentru ele	se alege tipul care consumă cea mai puțină memorie, dar în care încap toate valorile posibile
Procesorul	câte operații se execută până la rezultat	se caută varianta cu mai puțini pași, mai ales când datele sunt multe

12.1. Cât înseamnă, în cifre

Câți pași face căutarea divizorilor, pentru fiecare variantă:

n	Căutare până la n / 2	Căutare până la radical
`100`	50 pași	**10 pași**
`10 000`	5 000 pași	**100 pași**
`1 000 000`	500 000 pași	**1 000 pași**
`1 000 000 000`	500 de milioane de pași	**31 623 pași**

> Trei întrebări de pus la fiecare problemă, înainte de a scrie codul: câte treceri fac prin date, cât memorez și pot să mă opresc mai devreme?

13. Tablouri de memorie

> **Lungimea fizică** este cea din declarare — cât loc s-a rezervat. **Lungimea logică** este câte elemente sunt folosite efectiv. Prima nu se schimbă niciodată; a doua crește și scade în timpul programului.

13.1. Vectorul

Pentru un vector cu `n` elemente, indicii merg de la `0` la `n - 1`. Parcurgerea se poate face în ambele sensuri:

```
#include <iostream>
using namespace std;

int main() {
    const int DIM = 100;
    int a[DIM], n;

    cin >> n;

    for (int i = 0; i < n; i++)
        cin >> a[i];

    for (int i = 0; i < n; i++)         // de la primul la ultimul
        cout << a[i] << ' ';

    cout << '\n';

    for (int i = n - 1; i >= 0; i--)    // de la ultimul la primul
        cout << a[i] << ' ';

    return 0;
}
```

13.2. Matricea

Elementul `a[i][j]` se află pe linia `i` și coloana `j`. Parcurgerea cere două bucle, una în alta:

```
#include <iostream>
using namespace std;

int main() {
    const int DIM = 20;
    int a[DIM][DIM], n, m;

    cin >> n >> m;

    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> a[i][j];

    for (int i = 0; i < n; i++) {       // pe linii
        for (int j = 0; j < m; j++)
            cout << a[i][j] << ' ';
        cout << '\n';
    }

    return 0;
}
```

Poziție într-o matrice pătratică	Condiție
diagonala principală	`i == j`
diagonala secundară	`i + j == n - 1`
deasupra diagonalei principale	`i < j`
sub diagonala principală	`i > j`

Program — sumele diagonalelor:

```
#include <iostream>
using namespace std;

int main() {
    const int DIM = 20;
    int a[DIM][DIM], n;

    cin >> n;

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> a[i][j];

    int sp = 0, ss = 0;

    for (int i = 0; i < n; i++) {
        sp = sp + a[i][i];
        ss = ss + a[i][n - 1 - i];
    }

    cout << sp << ' ' << ss;

    return 0;
}
```

---

14. Algoritmi pentru căutarea unui element

14.1. Căutarea secvențială

Merge pe orice vector, sortat sau nu:

```
int poz = -1;

for (int i = 0; i < n; i++)
    if (a[i] == x) {
        poz = i;
        break;
    }

if (poz != -1)
    cout << "S-a gasit elementul in pozitia " << poz;
else
    cout << "Nu s-a gasit elementul";
```

14.2. Căutarea binară

> Se aplică numai unui vector **sortat** după criteriul căutării.

Se compară valoarea căutată cu elementul din mijloc: dacă nu e el, jumătate din vector se elimină dintr-o singură comparație.

```
int st = 0, dr = n - 1;
bool gasit = false;
int mijl;

while (st <= dr && !gasit) {
    mijl = (st + dr) / 2;

    if (a[mijl] == x)
        gasit = true;
    else if (x < a[mijl])
        dr = mijl - 1;      // se cauta in subvectorul din stanga
    else
        st = mijl + 1;      // se cauta in subvectorul din dreapta
}

if (gasit)
    cout << "S-a gasit elementul in pozitia " << mijl;
else
    cout << "Nu s-a gasit elementul";
```

Urmărim căutarea lui 15 în vectorul `7 9 11 14 16 18 20`:

st	dr	mijl	a[mijl]	Ce urmează
`0`	`6`	`3`	`14`	15 este mai mare decât 14 → se merge în dreapta
`4`	`6`	`5`	`18`	15 este mai mic decât 18 → se merge în stânga
`4`	`4`	`4`	`16`	15 este mai mic decât 16 → se merge în stânga
`4`	`3`	`–`	`–`	st a depășit dr → 15 nu apare

> Într-un vector cu un milion de elemente, căutarea secvențială face în cel mai rău caz un milion de comparații, iar cea binară cel mult 20.

15. Ștergerea și inserarea unui element

Un vector nu are goluri: ștergerea înseamnă mutarea elementelor de după el cu o poziție la stânga, iar inserarea înseamnă mutarea la dreapta, ca să se elibereze locul.

ȘTERGERE DIN POZIȚIA K

```
// se sterge elementul din pozitia
k
for (int i = k; i < n - 1; i++)
    a[i] = a[i + 1];

n--;                          //
lungimea logica scade cu 1
```

INSERARE ÎN POZIȚIA K

```
// se insereaza x in pozitia k
if (n + 1 <= DIM) {          //
incape in lungimea fizica?
    for (int i = n; i > k; i--)
        a[i] = a[i - 1];

    a[k] = x;
    n++;                     //
lungimea logica creste cu 1
}
```

> Sensul buclelor nu e la alegere. La ștergere se merge de la stânga la dreapta; la inserare, de la dreapta la stânga. Inversate, fiecare element se suprascrie pe cel următor și tot vectorul se umple cu aceeași valoare.

> La inserare se verifică întâi dacă mai încape: lungimea logică nu are voie să depășească lungimea fizică, altfel ultimul element se pierde.

16. Algoritmi pentru sortarea unui vector

Aranjarea elementelor se poate face în două feluri:

Unde se sortează	Metode
Într-un alt vector	metoda inserării, metoda numărării
În același vector	metoda selecției directe, metoda bulelor, metoda inserării directe, metoda inserării rapide

> Toate sortează crescător. Pentru ordonare descrescătoare e de ajuns să se schimbe sensul comparațiilor.

16.1. Metoda selecției directe

Se aduce pe prima poziție cel mai mic element, apoi pe a doua cel mai mic dintre cele rămase, și tot așa.

```
for (int i = 0; i < n - 1; i++)
    for (int j = i + 1; j < n; j++)
        if (a[j] < a[i]) {
            int aux = a[i];
            a[i] = a[j];
            a[j] = aux;
        }
```

16.2. Metoda bulelor

Se compară elementele vecine și se interschimbă dacă nu sunt în ordine. Vectorul se parcurge de mai multe ori, până când la o trecere completă nu se mai face nicio mutare.

```
bool terminat = false;

while (!terminat) {
    terminat = true;

    for (int i = 0; i < n - 1; i++)
        if (a[i] > a[i + 1]) {
            int aux = a[i];
            a[i] = a[i + 1];
            a[i + 1] = aux;
            terminat = false;   // s-a mai facut o mutare, mai trecem o data
        }
}
```

Pentru vectorul `4 3 2 1` — cazul cel mai dezavantajos:

Trecerea	Vectorul	terminat
prima	`4 3 2 1 → 3 2 1 4`	`false`
a doua	`3 2 1 4 → 2 1 3 4`	`false`
a treia	`2 1 3 4 → 1 2 3 4`	`false`
a patra	`1 2 3 4 — nicio mutare`	`true → gata`

> Variabila `terminat` se pune pe `true` la începutul **fiecărei** treceri, înăuntrul lui `while`. Pusă în afară, programul face o singură trecere și vectorul rămâne nesortat.

16.3. Metoda inserării directe

Vectorul se împarte în doi subvectori: cel din stânga, deja sortat, și cel din dreapta, încă nesortat. Primul element din partea nesortată se inserează la locul lui în partea sortată.

```
for (int i = 1; i < n; i++) {
    int aux = a[i];         // elementul care se insereaza
    int j = i - 1;

    while (j >= 0 && aux < a[j]) {
        a[j + 1] = a[j];    // se deplaseaza spre dreapta
        j--;
    }

    a[j + 1] = aux;         // se aseaza pe locul gasit
}
```

16.4. Metoda inserării rapide

Aceeași împărțire, dar poziția de inserare nu se mai caută element cu element: partea din stânga e sortată, deci se poate căuta binar.

```
for (int i = 1; i < n; i++) {
    int aux = a[i];
    int st = 0, dr = i - 1;

    while (st <= dr) {              // pozitia se cauta binar
        int mijl = (st + dr) / 2;

        if (aux < a[mijl])
            dr = mijl - 1;
        else
            st = mijl + 1;
    }

    for (int j = i - 1; j >= st; j--)
        a[j + 1] = a[j];

    a[st] = aux;
}
```

16.5. Metoda inserării, într-un alt vector

Elementele se copiază pe rând din vectorul sursă în vectorul destinație, fiecare fiind așezat la locul lui, astfel încât destinația să fie tot timpul sortată.

```
b[0] = a[0];

for (int i = 1; i < n; i++) {
    int j = 0;

    while (j <= i - 1 && a[i] > b[j])    // se cauta pozitia
        j++;

    for (int k = i; k > j; k--)          // se face loc
        b[k] = b[k - 1];

    b[j] = a[i];
}
```

16.6. Metoda numărării

Pentru fiecare element se numără câte elemente sunt mai mici decât el. Numărul acela este chiar poziția lui în vectorul sortat, deci fiecare element sare direct pe locul lui.

```
int k[DIM] = {0};           // k[i] = cate elemente sunt mai mici decat a[i]

for (int i = 0; i < n - 1; i++)
    for (int j = i + 1; j < n; j++)
        if (a[i] > a[j])
            k[i]++;
        else
            k[j]++;

for (int i = 0; i < n; i++)
    b[k[i]] = a[i];         // fiecare element sare direct pe locul lui
```

> Vectorul `k` nu ține valori, ci contoare. E aceeași idee ca la vectorii de frecvență și se întâlnește des la problemele care cer un algoritm eficient.

17. Algoritm pentru interclasarea a doi vectori

Doi vectori deja sortați se pot uni într-unul sortat fără să se mai sorteze nimic: se compară capetele și se ia de fiecare dată cel mai mic.

```
int i = 0, j = 0, k = 0;

while (i < n && j < m)
    if (a[i] < b[j])
        c[k++] = a[i++];
    else
        c[k++] = b[j++];

while (i < n)               // ce a ramas din a
    c[k++] = a[i++];

while (j < m)               // sau ce a ramas din b
    c[k++] = b[j++];
```

Cele două bucle de la final nu se pot uni: când prima se termină, unul dintre vectori e gol, deci doar una dintre ele are ce copia.

> Omiterea lor e greșeala clasică: programul merge corect pe exemplele în care ultimul element vine din vectorul care se golește primul, și greșit în rest.

---

18. Fișiere text

Datele nu se mai citesc de la tastatură, ci dintr-un fișier, iar rezultatul se scrie în altul. `fin` ține locul lui `cin`, iar `fout` pe al lui `cout`.

```
#include <fstream>
using namespace std;

int main() {
    ifstream fin("date.in");
    ofstream fout("date.out");

    if (!fin)
        return 1;               // fisierul de intrare lipseste

    int x;

    while (fin >> x)            // se opreste singur la sfarsitul fisierului
        if (x % 2 == 0)
            fout << x << ' ';

    fin.close();
    fout.close();
    return 0;
}
```

`while (fin >> x)` se oprește singur la sfârșitul fișierului: citirea întoarce o valoare falsă când nu mai are ce citi. De aceea nu trebuie să știm dinainte câte numere sunt.

> Verificarea `if (!fin)` nu e de prisos. Fără ea, un fișier de intrare care lipsește nu dă nicio eroare — programul pur și simplu nu citește nimic, iar fișierul de ieșire iese gol.

---

19. Activitate practică

1. Se citește un număr natural n de trei cifre. Să se afișeze numărul minim care se poate forma din cifrele sale.
2. Se citește un număr natural. Să se afișeze cea mai mare cifră a lui și de câte ori apare.
3. Se citește un număr natural. Să se descompună în factori primi.
4. Se citește un număr natural n și o bază q. Să se afișeze reprezentarea lui n în baza q.

5. Se citesc n numere. Să se afișeze primele două valori maxime și de câte ori apare fiecare.
6. Se citește un vector. Să se șteargă toate elementele egale cu valoarea maximă.
7. Se citește un vector. Să se sorteze crescător prin metoda bulelor, apoi să se caute binar o valoare x.
8. Se citesc doi vectori sortați. Să se interclaseze, afișând valorile fără duplicate.
9. Se citește o matrice pătratică. Să se calculeze sumele celor două diagonale.
10. Fișierul numere.in conține numere naturale. Să se scrie în prime.out numai numerele prime.

---

20. Greșelile care se repetă

- confundarea operatorilor = și ==;
- interschimbarea făcută prin două atribuiri, fără variabilă auxiliară, care pierde una dintre valori;
- inițializarea maximului cu 0, deși șirul poate avea numai valori negative;
- pierderea numărului în bucla care îi consumă cifrele, fără să se fi salvat o copie;
- la testarea primalității, indicatorul nu se resetează pentru fiecare număr dintr-un șir;
- accesarea elementului a[n], deși ultimul indice este n - 1;
- omiterea lui n-- la ștergere și a lui n++ la inserare;
- deplasarea elementelor în sens greșit: la ștergere de la stânga la dreapta, la inserare invers;
- la metoda bulelor, variabila terminat pusă în afara buclei while, nu înăuntru;
- aplicarea căutării binare unui vector nesortat;
- la căutarea binară, st = mijl în loc de st = mijl + 1, ceea ce duce la buclă infinită;
- omiterea elementelor rămase după interclasare;
- folosirea condiției i + j == n pentru diagonala secundară, în loc de i + j == n - 1;
- neverificarea deschiderii fișierului de intrare;
- testarea programului pe un singur exemplu.

Ce urmează

Toți algoritmii de mai sus vor fi rescriși ca **subprograme**, apoi unele dintre ele — c.m.m.d.c., Fibonacci, factorialul — vor primi și o variantă **recursivă**, iar cele două vor fi comparate. Urmează șirurile de caractere, structurile și listele înlănțuite.