

Manual de Informatică pentru licență valabil începând cu sesiunea iulie 2013

Specializarea Matematică-Informatică

Tematica generală:

Algoritmică și programare.

1. Căutari (secvențială și binară), sortări (selecție, bubblesort, quicksort).
2. Metodele backtracking și divide et impera.
3. Algoritmi și specificări. Scrierea unui algoritm pornind de la o specificație dată. Se dă un algoritm; se cere rezultatul execuției lui.
4. Concepte OOP în limbaje de programare (C++, Java, C#): Clase și obiecte. Membrii unei clase și specificatorii de acces. Constructori și destructori.

Cuprins

ALGORITMICĂ ȘI PROGRAMARE.....	3
1. CĂUTĂRI ȘI SORTĂRI.....	3
1.1. CĂUTĂRI.....	3
1.1.1. Căutare secvențială.....	3
1.1.2. Căutare binară.....	4
1.2. SORTĂRI.....	5
1.2.1. Sortare prin selecție.....	6
1.2.2. Bubble sort.....	6
1.2.3. Quicksort.....	7
2. METODELE BACKTRACKING ȘI DIVIDE ET IMPERA.....	8
2.1. METODA BACKTRACKING.....	8
2.2. METODA "DIVIDE ET IMPERA".....	12
3. ALGORITMI ȘI SPECIFICĂRI.....	13
3.1. SCRIEREA UNUI ALGORITM PORNIND DE LA O SPECIFICATIE DATĂ.....	13
4. CONCEPTE OOP ÎN LIMBAJE DE PROGRAMARE.....	17
4.1. NOȚIUNEA DE CLASĂ.....	17
4.1.1. Realizarea protecției datelor prin metoda programării modulare.....	17
4.1.2. Tipuri abstracte de date.....	18
4.1.3. Declararea claselor.....	19
4.1.4. Membrii unei clase. Pointerul this.....	21
4.1.5. Constructorul.....	21
4.1.6. Destructorul.....	25
5. PROBLEME PROPUSE.....	26
6. BIBLIOGRAFIE GENERALĂ.....	27

Algoritmică și programare

1. Căutări și sortări

1.1. Căutări

Datele se află în memoria internă, într-un șir de articole. Vom căuta un articol după un câmp al acestuia pe care îl vom considera cheia de căutare. În urma procesului de căutare va rezulta poziția elementului căutat (dacă acesta există).

Notând cu k_1, k_2, \dots, k_n cheile corespunzătoare articolelor și cu a cheia pe care o căutăm, problema revine la a găsi (dacă există) poziția p cu proprietatea $a = k_p$.

De obicei articolele sunt păstrate în ordinea crescătoare a cheilor, deci vom presupune că

$$k_1 < k_2 < \dots < k_n.$$

Uneori este util să aflăm nu numai dacă există un articol cu cheia dorită ci și să găsim în caz contrar locul în care ar trebui inserat un nou articol având cheia specificată, astfel încât să se păstreze ordinea existentă.

Deci *problema căutării* are următoarea specificare:

Date $a, n, (k_i, i=1, n)$;

Precondiția: $n \in \mathbb{N}, n \geq 1$ și $k_1 < k_2 < \dots < k_n$;

Rezultate p ;

Postcondiția: $(p=1$ și $a \leq k_1)$ sau $(p=n+1$ și $a > k_n)$ sau $(1 < p \leq n)$ și $(k_{p-1} < a \leq k_p)$.

1.1.1. Căutare secvențială

O primă metodă este căutarea **secvențială**, în care sunt examinate succesiv toate cheile. Sunt deosebite trei cazuri: $a \leq k_1$, $a > k_n$, respectiv $k_1 < a \leq k_n$, căutarea având loc în al treilea caz.

Subalgoritmul CautSecv(a, n, K, p) este: $\{n \in \mathbb{N}, n \geq 1$ și $k_1 < k_2 < \dots < k_n\}$

{Se caută p astfel ca: $(p=1$ și $a \leq k_1)$ sau}

{ $(p=n+1$ și $a > k_n)$ sau $(1 < p \leq n)$ și $(k_{p-1} < a \leq k_p)$.

{Cazul "încă negăsit"}

Fie $p := 0$;

Dacă $a \leq k_1$ atunci $p := 1$ altfel

Dacă $a > k_n$ atunci $p := n + 1$ altfel

Pentru $i := 2$; n execută

Dacă $(p = 0)$ și $(a \leq k_i)$ atunci $p := i$ sfdacă

sfpentru

sfdacă

sfdacă
sf-CautSecv

Se observă că prin această metodă se vor executa în cel mai nefavorabil caz $n-1$ comparații, întrucât contorul i va lua toate valorile de la 2 la n . Cele n chei împart axa reală în $n+1$ intervale. Tot atâtea comparații se vor efectua în $n-1$ din cele $n+1$ intervale în care se poate afla cheia căutată, deci complexitatea medie are același ordin de mărime ca și complexitatea în cel mai rău caz.

Evident că în multe situații acest algoritm face calcule inutile. Atunci când a fost deja găsită cheia dorită este inutil a parcurge ciclul pentru celelalte valori ale lui i . Cu alte cuvinte este posibil să înlocuim ciclul **PENTRU** cu un ciclu **CÂTTIMP**. Ajungem la un al doilea algoritm, dat în continuare.

Subalgoritmul CautSucc(a, n, K, p) este: $\{n \in \mathbb{N}, n \geq 1 \text{ și } k_1 < k_2 < \dots < k_n\}$
 $\{ \text{Se caută } p \text{ astfel ca: } p=1 \text{ și } a \leq k_1 \text{ sau } \}$
 $\{ (p=n+1 \text{ și } a > k_n) \text{ sau } (1 < p \leq n) \text{ și } (k_{p-1} < a \leq k_p) \}$

```

Fie p:=1;
Dacă a>k1 atunci
    Cât timp p≤n și a>kp execută p:=p+1 sfcă
sfdacă
sf-CautSucc

```

În cel mai rău caz și acest algoritm face același număr de operații ca și subalgoritmul *Cautsecv*. În medie numărul operațiilor este jumătate din numărul mediu de operații efectuat de subalgoritmul *Cautsecv* deci complexitatea este aceeași.

1.1.2. Căutare binară

O altă metodă, numită **căutare binară**, care este mult mai eficientă, utilizează tehnica "divide et impera" privitor la date. Se determină în ce relație se află cheia articolului aflat în mijlocul colecției cu cheia de căutare. În urma acestei verificări căutarea se continuă doar într-o jumătate a colecției. În acest mod, prin înjumătățiri succesive se micșorează volumul colecției rămase pentru căutare. Căutarea binară se poate realiza practic prin apelul funcției $\text{BinarySearch}(a, n, K, 1, n)$, descrisă mai jos, folosită în subalgoritmul dat în continuare.

Subalgoritmul CautBin(a, n, K, p) este: $\{n \in \mathbb{N}, n \geq 1 \text{ și } k_1 < k_2 < \dots < k_n\}$
 $\{ \text{Se caută } p \text{ astfel ca: } (p=1 \text{ și } a \leq k_1) \text{ sau } \}$
 $\{ (p=n+1 \text{ și } a > k_n) \text{ sau } (1 < p \leq n) \text{ și } (k_{p-1} < a \leq k_p) \}$

```

Dacă a ≤ k1 atunci p := 1 altfel
    Dacă a > kn atunci p := n+1 altfel
    P := BinarySearch(a, n, K, 1, n)
sfdacă
sf-CautBin

```

Funcția BinarySearch(a, n, K, St, Dr) este:

```
Dacă St ≥ Dr - 1
    atunci BinarySearch := Dr
    altfel m := (St+Dr) Div 2;
        Dacă a ≤ km
            atunci BinarySearch := BinarySearch(a, n, K, St, m)
            altfel BinarySearch := BinarySearch(a, n, K, m, Dr)
        sfdacă
    sfdacă
sf-BinarySearch
```

În funcția BinarySearch descrisă mai sus, variabilele St și Dr reprezintă capetele intervalului de căutare, iar m reprezintă mijlocul acestui interval. Prin această metodă, într-o colecție având n elemente, rezultatul căutării se poate furniza după cel mult $\log_2 n$ comparații. Deci complexitatea în cel mai rău caz este direct proporțională cu $\log_2 n$. Fără a insista asupra demonstrației, menționăm că ordinul de mărime al complexității medii este același.

Se observă că funcția BinarySearch se apelează recursiv. Se poate înlătura ușor recursivitatea, așa cum se poate vedea în următoarea funcție:

Funcția BinSeaNerec(a, n, K, St, Dr) este:

```
Cât timp Dr - St > 1 execută
    m := (St+Dr) Div 2;
    Dacă a ≤ km atunci Dr := m altfel St := m sfdacă
sfcât
BinSeaNerec := Dr
sf-BinSeaNerec
```

1.2. Sortări

Prin sortare internă vom înțelege o rearanjare a unei colecții aflate în memoria internă astfel încât cheile articolelor să fie ordonate crescător (eventual descrescător).

Din punct de vedere al complexității algoritmilor problema revine la ordonarea cheilor. Deci specificarea problemei de **sortare internă** este următoarea:

Date n, K ;

$\{K=(k_1, k_2, \dots, k_n)\}$

Precondiția: $k_i \in R, i=1, n$

Rezultate K' ;

Postcondiția: K' este o permutare a lui K , dar ordonată crescător.

Deci $k_1 \leq k_2 \leq \dots \leq k_n$.

1.2.1. Sortare prin selecție

O primă tehnică numită "*Selecție*" se bazează pe următoarea idee: se determină poziția elementului cu cheie de valoare minimă (respectiv maximă), după care acesta se va interschimba cu primul element. Acest procedeu se repetă pentru subcolecția rămasă, până când mai rămâne doar elementul maxim.

Subalgoritmul *Selectie*(n, K) este:

```
{Se face o permutare a celor}
{n componente ale vectorului K astfel}
{ca  $k_1 \leq k_2 \leq \dots \leq k_n$ }
```

Pentru $i := 1$; $n-1$ execută
 Fie $ind := i$;
 Pentru $j := i + 1$; n execută
 Dacă $k_j < k_{ind}$ atunci $ind := j$ sfdacă
 sfpentru
 Dacă $i < ind$ atunci $t := k_i$; $k_i := k_{ind}$; $k_{ind} := t$ sfdacă
 sfpentru
sf-*Selectie*

Se observă că numărul de comparații este:

$$(n-1)+(n-2)+\dots+2+1=n(n-1)/2$$

indiferent de natura datelor. Deci complexitatea medie, dar și în cel mai rău caz este $O(n^2)$.

1.2.2. Bubble sort

Metoda "*BubbleSort*", compară două câte două elemente consecutive iar în cazul în care acestea nu se află în relația dorită, ele vor fi interschimbate. Procesul de comparare se va încheia în momentul în care toate perechile de elemente consecutive sunt în relația de ordine dorită.

Subalgoritmul *BubbleSort*(n, K) este:

```
Repetă
  Fie kod := 0; {Ipoteza "este ordine"}
  Pentru i := 2; n execută
    Dacă  $k_{i-1} > k_i$  atunci
       $t := k_{i-1}$ ;
       $k_{i-1} := k_i$ ;
       $k_i := t$ ;
      kod := 1 {N-a fost ordine!}
    sfdacă
  sfpentru
  pânăcând kod = 0 sfrep {Ordonare}
sf-BubbleSort
```

Acest algoritm execută în cel mai nefavorabil caz $(n-1)+(n-2)+ \dots +2+1 = n(n-1)/2$ comparații, deci complexitatea lui este $O(n^2)$.

O variantă optimizată a algoritmului "*BubbleSort*" este :

Subalgoritmul BubbleSort(n, K) este:

```
Fie s := 0
Repetă
  Fie kod := 0;                                     {Ipoteza "este ordine"}
  Pentru i := 2; n-s execută
    Dacă  $k_{i-1} > k_i$  atunci
      t :=  $k_{i-1}$ ;
       $k_{i-1} := k_i$ ;
       $k_i := t$ ;
      kod := 1                                     {N-a fost ordine!}
    sfdacă
  sfpentru
  s := s + 1
pânăcând kod = 0 sfrep                             {Ordonare}
sf-BubbleSort
```

1.2.3. Quicksort

O metodă mai performantă de ordonare, care va fi prezentată în continuare, se numește "QuickSort" și se bazează pe tehnica "divide et impera" după cum se poate observa în continuare. Metoda este prezentată sub forma unei proceduri care realizează ordonarea unui subșir precizat prin limita inferioară și limita superioară a indicilor acestuia. Apelul procedurii pentru ordonarea întregului șir este : QuickSort(n, K, 1, n), unde n reprezintă numărul de articole ale colecției date. Deci

Subalgoritmul SortareRapidă(n, K) este:

```
Cheamă QuickSort(n, K, 1, n)
sf-SortareRapidă
```

Procedura QuickSort(n, K, St, Dr) va realiza ordonarea subșirului $k_{St}, k_{St+1}, \dots, k_{Dr}$. Acest subșir va fi rearanjat astfel încât k_{St} să ocupe poziția lui finală (când șirul este ordonat). Dacă i este această poziție, șirul va fi rearanjat astfel încât următoarea condiție să fie îndeplinită:

$$k_j \leq k_i \leq k_l, \text{ pentru } st \leq j < i < l \leq dr \quad (*)$$

Odată realizat acest lucru, în continuare va trebui doar să ordonăm subșirul $k_{St}, k_{St+1}, \dots, k_{i-1}$ prin apelul recursiv al procedurii QuickSort(n, K, St, i-1) și apoi subșirul k_{i+1}, \dots, k_{Dr} prin apelul QuickSort(n, K, i+1, Dr). Desigur ordonarea acestor două subșiruri (prin apelul recursiv al procedurii) mai este necesară doar dacă acestea conțin cel puțin două elemente.

Procedura QuickSort este prezentată în continuare :

```

Subalgoritmul QuickSort (n, K, St, Dr) este:
  Fie i := St; j := Dr; a := ki;
  Repetă
    Cât timp kj ≥ a și (i < j) execută j := j - 1 sfcăț
    ki := kj;
    Cât timp ki ≤ a și (i < j) execută i := i + 1 sfcăț
    kj := ki;
  pânăcând i = j sfrep
  Fie ki := a;
  Dacă St < i-1 atunci Cheamă QuickSort(n, K, St, i - 1) sfdacă
  Dacă i+1 < Dr atunci Cheamă QuickSort(n, K, i + 1, Dr) sfdacă
sf-QuickSort

```

Complexitatea algoritmului prezentat este $O(n^2)$ în cel mai nefavorabil caz, dar complexitatea medie este de ordinul $O(n \log_2 n)$.

2. Metodele backtracking și divide et impera

2.1. Metoda backtracking

Metoda backtracking (căutare cu revenire) este aplicabilă în general unor probleme ce au mai multe soluții.

Vom considera întâi un exemplu, după care vom indica câțiva algoritmi generali pentru această metodă.

Problema 1. (Generarea permutărilor) Fie n un număr natural. Determinați permutările numerelor $1, 2, \dots, n$.

O soluție pentru generarea permutărilor, în cazul particular $n = 3$, ar putea fi:

```

Subalgoritmul Permutări1 este:
  Pentru i1 := 1; 3 execută
    Pentru i2 := 1; 3 execută
      Pentru i3 := 1; 3 execută
        Fie posibil := (i1, i2, i3)
        Dacă componentele vectorului posibil sunt distincte
          atunci
            Tipărește posibil
          sfdacă
        sfpentru
      sfpentru
    sfpentru
  sf-Permutări1

```

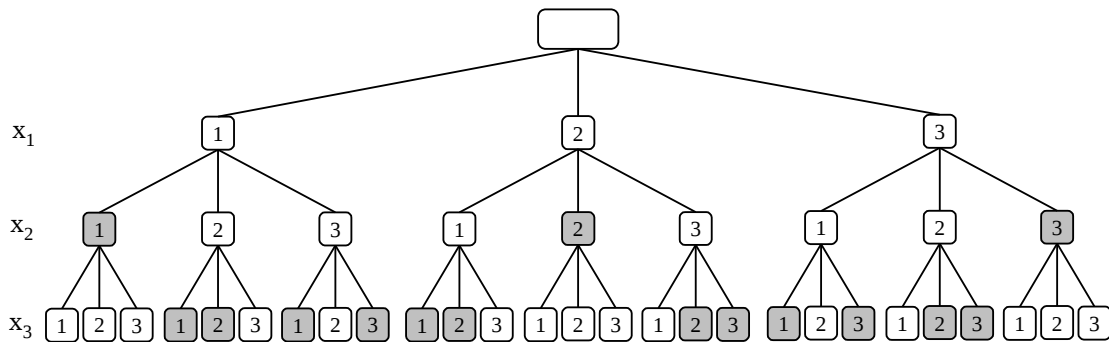



Figura 1.1. Reprezentare grafică a produsului cartezian $\{1, 2, 3\}^3$

Observații privind subalgoritmul *Permutări1*:

- Pentru n oarecare nu putem descrie un algoritm care să conțină n cicluri în textul sursă.
- Numărul total de vectori verificați este 3^3 , iar în general n^n . Vectorii *posibil* verificați sunt reprezentați grafic în Figura 1.1 - fiecare vector este un drum de la rădăcină (de sus) spre frunze (baza arborelui).
- Algoritmul *atribuie valori tuturor componentelor vectorului x , apoi verifică dacă vectorul este o permutare.*

O îmbunătățire a acestor algoritmi ar consta în a verifica anumite condiții din problemă în timp ce se construiesc vectorii, evitând completarea inutilă a unor componente.

De exemplu, dacă prima componentă a vectorului construit (*posibil*) este 1, atunci este inutil să atribuim celei de a doua componente valoarea 1, iar componentei a treia oricare din valorile 1, 2 sau 3. Dacă n este mare se evită completarea multor vectori ce au prefixul (1, ...). În acest sens, (1, 3, ...) este un *vector promițător* (pentru a fi o permutare), în schimb vectorul (1, 1, ...) nu este. Vectorul (1, 3, ...) satisface anumite *condiții de continuare* (pentru a ajunge la soluție) - are componente distincte. Nodurile hașurate din Figura 1.1 constituie valori care nu conduc la o soluție.

Vom descrie un algoritm general pentru metoda Backtracking după care vom particulariza acest algoritm pentru problemele enunțate la începutul secțiunii. Pentru început vom face câteva observații și notații privind metoda Backtracking aplicată unei probleme în care soluțiile se reprezintă pe vectori, nu neapărat de lungime fixă:

- spațiul de căutare a soluțiilor (spațiul soluțiilor posibile): $S = S_1 \times S_2 \times \dots \times S_n$;
- *posibil* este vectorul pe care se reprezintă soluțiile;
- *posibil[1..k]* $\in S_1 \times S_2 \times \dots \times S_k$ este un vector care poate conduce sau nu la o soluție; k reprezintă indice pentru vectorul *posibil*, respectiv nivel în arborele care redă grafic procesul de căutare (Figura 1.2).
- *posibil[1..k]* este promițător dacă satisface condiții care pot conduce la o soluție;
- *soluție(n, k, posibil)* funcție care verifică dacă vectorul (promițător) *posibil[1..k]* este soluție a problemei.

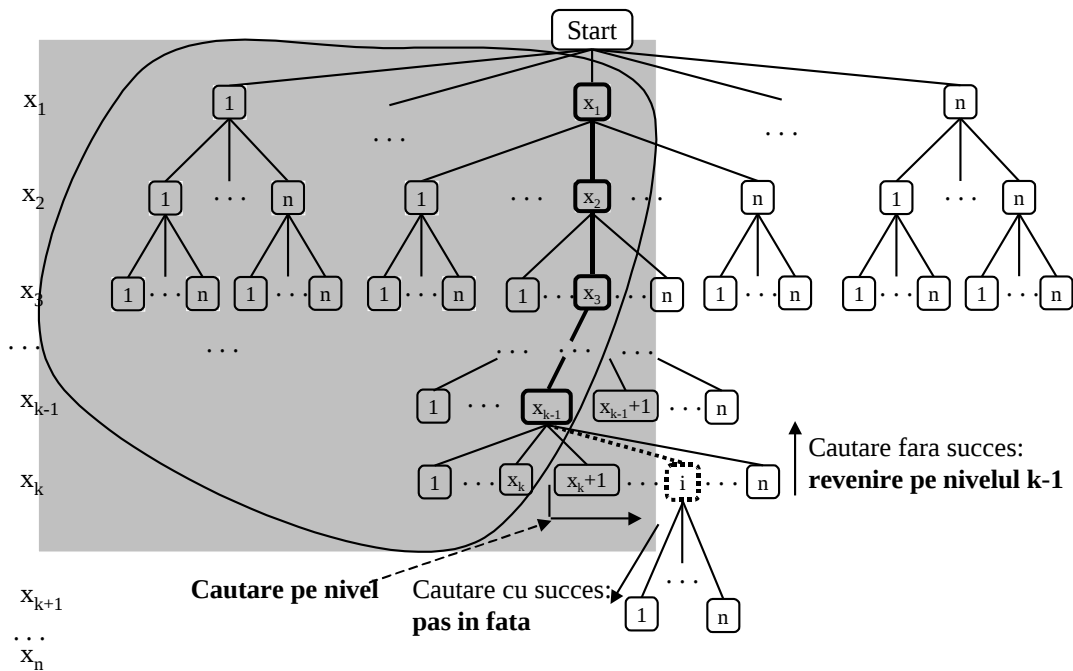


Figura 1.2. Spațiul soluțiilor posibile pentru generarea permutărilor

Procesul de căutare poate fi urmărit în algoritmul care urmează:

```

Algoritmul Backtracking este: {varianta nefinisată}
Fie k := 1
@Inițializează căutarea pe nivelul k (= 1)
Cât timp k > 0 execută {posibil[1..k-1] este promițător}
  @Caută (secvențial) pe nivelul k o valoare v, pentru a completa în
  continuare vectorul posibil[1..k-1] astfel încât posibil[1..k] să
  fie promițător
  Dacă căutarea este cu succes
    atunci Fie posibil[k] := v {posibil[1..k] este promițător}
    Dacă soluție(n, k, posibil)
      atunci {o soluție! (rămânem pe nivelul k)}
      Tiparește posibil[1..k]
    altfel {e doar un vector promițător}
    @Initializeaza cautarea pe nivelul k+1
    Fie k := k + 1 {pas în față (pe nivelul k+1)}
  sfdacă
  altfel {pas în spate (revenire pe nivelul k-1)}
  k := k - 1
sfdacă
sfcât
sf-Backtracking
  
```

Pentru a finisa acest algoritm trebuie să precizăm elementele nstandard prezente. Astfel, avem nevoie de funcția booleană

condiții-continuare(k, posibil, v)

funcție care verifică dacă vectorul promițător *posibil*[1..k-1] completat cu valoarea v conduce la un vector promițător.

Apoi, pentru a inițializa căutarea la nivelul j avem nevoie de a alege un element fictiv din mulțimea S_j , activitate realizată de funcția

init(j)

care returnează acest element fictiv, care are rolul de a indica faptul că din mulțimea S încă nu s-a ales nici un element, deci după el urmează primul element propriu din această mulțime. Pentru a căuta o valoare pe nivelul j , în ipoteza că valoarea curentă nu e bună, avem nevoie de funcția booleană

următor(j, v, nou)

care este *True* dacă poate alege o valoare din S_j care urmează după valoarea v , valoare notată prin *nou* și *False* în cazul în care nu mai există alte valori în S_j , deci nu mai poate fi făcută alegerea. Cu aceste notații algoritmul devine:

```
Algoritmul Backtracking este: {versiune finală}
Fie k := 1;
posibil[1] := init(1);
Cât timp k > 0 execută {posibil[1..k-1] este promițător}
    Fie Găsit := false; v := posibil[k] ;
    Cât timp Următor(k, v, urm) și not Găsit execută
        Fie v := urm;
        Dacă condiții-continuare(k, posibil, v) atunci
            Găsit := true
        sfdacă
    sfcât
Dacă Găsit
    atunci Fie posibil[k] := v; {posibil[1..k] este promițător}
        Dacă soluție(n, k, posibil)
            atunci {o soluție! (rămânem pe nivelul k)}
                Tiparește posibil[1..k]
            altfel {e doar un vector promițător}
                Fie k := k + 1; {pas în față (pe nivelul k+1)}
                posibil[k] := init(k)
        sfdacă
    altfel {pas în spate (revenire pe nivelul k-1)}
        k := k - 1;
    sfdacă
sfcât
sf-Backtracking
```

Procesul de căutare a unei valori pe nivelul k și funcțiile *condiții-continuare* și *soluție* sunt dependente de problema care se rezolvă. De exemplu, pentru generarea permutărilor funcțiile menționate sunt:

Funcția *init(k)* este:

```
Init := 0
sf-init;
```

Funcția *Următor(k, v, urm)* este:

```
Dacă v < n
    atunci Următor := True; urm := v + 1
    altfel Următor := False
sfdacă
sf-urmator
```

Funcția *conditii-continuare(k, posibil, v)* este:

```
Kod := True; i := 1;
Cât timp kod și (i < k) execută
    Dacă posibil[i] = v atunci kod := False sfdacă
    i := i + 1;
```

```

    sfcât
    conditii-continuare:=kod
sf-conditii

```

```

Funcția soluții(n, k, posibil) este:
    Soluții := (k = n)
sf-soluții

```

În încheiere, menționăm că explorarea backtracking poate fi descrisă de asemenea recursiv. Dăm în acest scop următoru subalgoritm:

```

Subalgoritmul Backtracking(k, posibil) este:
    {posibil[1..k] este promițător}
    Dacă soluție(n, k, posibil) atunci
        {o soluție! terminare apel recursiv, astfel}
        Tipareste posibil[1..k]
        {rămânem pe același nivel}
    altfel
        Pentru fiecare v valoare posibilă pentru posibil[k+1] execută
            Dacă condiții-continuare(k + 1, posibil, v) atunci
                posibil[k + 1] := v
                Backtracking(k + 1, posibil)
                {pas în față}
            sfdacă
        sfpentru
    sfdacă
    {terminare apel Backtracking(k, posibil)}
sf-Backtracking {deci, pas în spate (revenire)}

```

cu apelul inițial **Cheamă** Backtracking(0, posibil).

2.2. Metoda "divide et impera"

Strategia "Divide et Impera" în programare presupune împărțirea datelor ("divide and conquer") și împărțirea problemei în subprobleme ("top-down").

Metoda se aplica problemelor care pot fi descompuse în subprobleme independente, similar problemei inițiale, de dimensiuni mai mici și care pot fi rezolvabile foarte ușor. Ea se aplică atunci când rezolvarea problemei P pentru setul de date D se poate face prin rezolvarea aceleiași probleme P pentru alte seturi de date d_1, d_2, \dots, d_k de volum mai mic decât D .

Observații:

- o Împărțirea se face până când se obține o problemă rezolvabilă imediat.
- o Subproblemele în care se descompune problema inițială trebuie să fie independente. Dacă subproblemele nu sunt independente, se aplică alte metode de rezolvare.
- o Tehnica admite și o implementare recursivă.

Metoda poate fi descrisă în felul următor:

- **Împarte:** Dacă dimensiunea datelor este prea mare pentru a fi rezolvabilă imediat, împarte problema în una sau mai multe subprobleme independente (similare problemei inițiale).
- **Stăpânește:** Folosește recursive aceeași metodă pentru a rezolva subproblemele.
- **Combină:** Combină soluțiile subproblemelor pentru a obține soluția problemei inițiale.

Subalgoritmul S pentru rezolvarea problemei P folosind metoda ‚Divide et Impera’ are următoarea structură:

```
Sublgoritmul S(D) este:
  Dacă dim(D) ≤ a atunci
    @problema se rezolva
  altfel
    @ Descompune D in d1, d2, ..., dk
    Cheama S(d1)
    Cheama S(d2)
    .
    .
    Cheama S(dk)
    @ construiește rezultatul final prin utilizarea rezultatelor
    @ parțiale din apelurile de mai sus
  sfdacă
sf-NumeAlg
```

3. Algoritmi și specificații

Algoritmi și specificații. Scrierea unui algoritm pornind de la o specificație dată. Se dă un algoritm; se cere rezultatul execuției lui

3.1. Scrierea unui algoritm pornind de la o specificație dată

Problema 1

Scrieți o funcție care satisface următoarea specificație:

Date nr;

Precondiția: nr ∈ N, nr ≥ 1

Rezultate l₁, l₂, ..., l_n;

Postcondiția: n ∈ N, $\left\lceil \frac{nr}{l_i} \right\rceil \cdot l_i = nr \forall 1 \leq i \leq n, l_i \neq l_j \forall i \neq j, 1 \leq i, j \leq n$, n este maximal*

Problema 2

Scrieți o funcție care satisface următoarea specificație:

Date $n, L = (l_1, l_2, \dots, l_n)$;

Precondiția: $l_i \in \mathbb{R}, i=1, n$

Rezultate $R = (r_1, r_2, \dots, r_n)$;

Postcondiția: R este o permutare a lui L , $r_1 \geq r_2 \geq \dots \geq r_n$.

Problema 3

Se cere să se scrie un algoritm/program pentru rezolvarea următoarei probleme.

Când merge la cumpărături, Ana își pregătește tot timpul o listă de cumpărături: denumire, cantitate, raion (alimente, îmbrăcăminte, încălțăminte, consumabile), preț estimat. Se cere să se afișeze lista de cumpărături a Anei ordonată alfabetic după raion, lista ordonată descrescător după cantitate, precum și lista Anei de la un anumit raion. Se cere să se calculeze și un preț estimativ al cheltuielilor Anei.

Problema 4

Se cere să se scrie un algoritm/program pentru rezolvarea următoarei probleme.

Să se scrie un program care citește un șir de numere întregi nenule. Introducerea unui șir se încheie odată cu citirea valorii 0. În șirul citit programul va elimina secvențele de elemente consecutive strict pozitive de lungime mai mare decât 3 (dacă există), după care va tipări șirul obținut.

Problema 5

Ce face următorul program Pascal?

```
Program Ce_rezolva___;
```

```
Function Prim(p:Integer):Boolean;
```

```
Var d:Integer;
```

```
Begin
```

```
  d:=2;
```

```
  While (d<=Sqrt(p)) And (p Mod d>0) Do Inc(d);
```

```
  Prim:= d> Sqrt(p)
```

```
End;
```

```
Function Desc(n:Integer; Var p1,p2:Integer):Boolean;
```

```
Begin
```

```
  p1:=2;
```

```
  While (p1<=n Div 2) And (Not Prim(p1) Or Not Prim(n-p1)) Do Inc(p1);
```

```
  Desc:= p1<=n Div 2;
```

```
  p2:=n-p1
```

```
End;
```

```
Var n,p1,p2:Integer;
```

```
Begin
```

```

Repeat
  Write(' Dati n : '); Readln(n);
  If n>0 Then
    If Desc(n,p1,p2) Then Writeln(n:5,p1:7,p2:5)
      Else Writeln(' Nu ex. desc. ');
  Until n=0;
End.

```

Problema 6

Pentru următorul program Pascal precizați:

- ce face acest program;
- ce realizează fiecare subprogram;
- ce rezultate sunt tipărite pentru 12 1233 1132 2338 8533 10000 21500 0 ?

```

Program Fis_Sir_P__;
Uses Crt;
Type Sir=Array [1..100] Of Integer;
  MultimeC = Set Of 0..9;

Procedure C__S__(Var X:Sir; Var n:Byte);
Var v:Integer; f:Text;
Begin
  Assign(f, 'Sir.In'); Reset(f); n:=0;
  Writeln(' Sir.In : ');
  Repeat
    Read (f,v);
    If v>0 Then Begin Inc(n); x[n]:=v; Write(v, ',') End
  Until v=0;
  Close(f); Writeln
End;

Procedure T__S__(X:Sir; n:Byte);
Var i:Byte; f:Text;
Begin
  Assign(f, 'Sir.Out'); Rewrite(f);
  Writeln(' Sir.Out : ');
  For i:=1 To n Do Begin
    Write(f,x[i], ' ');
    Write( ' x[i], ' ')
  End;
  Writeln(f);
  Close(f); Writeln
End;

Procedure Mc(a:Integer; Var M:MultimeC);
Begin
  M:=[];
  Repeat
    M:=M+[a Mod 10];
    a:=a Div 10
  Until a=0;
End;

Function Acc(a,b:Integer):Boolean;
Var Ma, Mb:MultimeC;
Begin
  Mc(a, Ma);
  Mc(b, Mb);

```

```

        Acc:= Ma=Mb
End;

Procedure P__(Var p:Integer);
Var v:Integer;
Begin
    v:=0;
    Repeat
        v:=v*10+p Mod 10;
        p:=p Div 10
    Until p=0;
    p:=v
End;

Procedure M__S__(Var X:Sir; n:Byte);
Var i:Byte; v:Integer;
Begin
    For i:=1 To n-1 Do
        If Acc(x[i],x[i+1]) Then P__(x[i])
    End;

Var X:Sir;
    n:Byte;
Begin
    ClrScr;
    C__S__(X,n);
    M__S__(X,n);
    T__S__(X,n);
    Readln
End.

```

Raspuns c) 12 3321 1132 2338 8533 10000 21500

Problema 7

Precizați ce realizează următorul program, apoi scrieți programul Pascal pentru *funcția* inversă.

```

Program Str_Lit_Urm;
Uses Crt;

Function UrmL(l:Char):Char;
Begin
    Case l Of
        'Z': UrmL:='A';
        'z': UrmL:='a';
        Else UrmL:=Succ(l)
    End
End;

Function ModC(c:Char):Char;
Begin
    If UpCase(c) In ['A'..'Z'] Then ModC:=UrmL(c)
    Else ModC:=c
End;

Function Modif(s:String):String;
Var i:Byte;

```



```

Begin
    For i:=1 To Length(s) Do s[i]:=ModC(s[i]);
    Modif:=s
End;

Var s:String;
Begin
    ClrScr;
    Writeln(' Dati randurile de codif.:');
    Repeat
        Readln(s);
        Writeln(Modif(s))
    Until s=''
End.

```

Obs. Presupunând că acest program realizează o *codificare* a unui text, scrieți programul care realizează *decodificarea*!

4. Concepte OOP în limbaje de programare

4.1. Noțiunea de clasă

4.1.1. Realizarea protecției datelor prin metoda programării modulare

Dezvoltarea programelor prin programare procedurală înseamnă folosirea unor funcții și proceduri pentru scrierea programelor. În limbajul C lor le corespund funcțiile care returnează o valoare sau nu. Însă în cazul aplicațiilor mai mari ar fi de dorit să putem realiza și o protecție corespunzătoare a datelor. Acest lucru ar însemna că numai o parte a funcțiilor să aibă acces la datele problemei, acelea care se referă la datele respective. Programarea modulară oferă o posibilitate de realizare a protecției datelor prin folosirea clasei de memorie static. Dacă într-un fișier se declară o dată aparținând clasei de memorie statică în afara funcțiilor, atunci ea poate fi folosită începând cu locul declarării până la sfârșitul modulului respectiv, dar nu și în afara lui.

Să considerăm următorul exemplu simplu referitor la prelucrarea vectorilor de numere întregi. Să se scrie un modul referitor la prelucrarea unui vector cu elemente întregi, cu funcții corespunzătoare pentru inițializarea vectorului, eliberarea zonei de memorie ocupate și ridicarea la pătrat, respectiv afișarea elementelor vectorului. O posibilitate de implementare a modulului este prezentată în fișierul **vector1.cpp**:

```

#include <iostream>

using namespace std;

static int* e;           //elementele vectorului
static int d;           //dimensiunea vectorului

void init(int* e1, int d1) //initializare
{

```

```

    d = d1;
    e = new int[d];
    for(int i = 0; i < d; i++)
        e[i] = e1[i];
}

void distr()                //eliberarea zonei de memorie ocupata
{
    delete [] e;
}

void lapatrat()            //ridicare la patrat
{
    for(int i = 0; i < d; i++)
        e[i] *= e[i];
}

void afiseaza()           //afisare
{
    for(int i = 0; i < d; i++)
        cout << e[i] << ' ';
    cout << endl;
}

```

Modulul se compilează separat obținând un program obiect. Un exemplu de program principal este prezentat în fișierul **vector2.cpp**:

```

extern void init( int*, int); //extern poate fi omis
extern void distr();
extern void lapatrat();
extern void afiseaza();
//extern int* e;
int main() {
    int x[5] = {1, 2, 3, 4, 5};
    init(x, 5);
    lapatrat();
    afiseaza();
    distr();
    int y[] = {1, 2, 3, 4, 5, 6};
    init(y, 6);
    //e[1]=10;                eroare, datele sunt protejate
    lapatrat();
    afiseaza();
    distr();
    return 0;
}

```

Observăm că deși în programul principal se lucrează cu doi vectori nu putem să-i folosim împreună, deci de exemplu modulul **vector1.cpp** nu poate fi extins astfel încât să realizeze și adunarea a doi vectori. În vederea înlăturării acestui neajuns s-au introdus tipurile abstracte de date.

4.1.2. Tipuri abstracte de date

Tipurile abstracte de date realizează o legătură mai strânsă între datele problemei și operațiile (funcțiile) care se referă la aceste date. Declararea unui tip abstract de date este asemănătoare cu declararea unei structuri, care în afară de date mai cuprinde și declararea sau definirea funcțiilor referitoare la acestea.

De exemplu în cazul vectorilor cu elemente numere întregi putem declara tipul abstract:

```
struct vect {
    int* e;
    int d;
    void init(int* e1, int d1);
    void distr() { delete [] e; }
    void lapatrat();
    void afiseaza();
};
```

Funcțiile declarate sau definite în interiorul structurii vor fi numite *funcții membru* iar datele *date membru*. Dacă o funcție membru este definită în interiorul structurii (ca și funcția *distr* din exemplul de mai sus), atunci ea se consideră funcție *inline*. Dacă o funcție membru se definește în afara structurii, atunci numele funcției se va înlocui cu numele tipului abstract urmat de operatorul de rezoluție (::) și numele funcției membru. Astfel funcțiile *init*, *lapatrat* și *afiseaza* vor fi definite în modul următor:

```
void vect::init(int *e1, int d1)
{
    d = d1;
    e = new int[d];
    for(int i = 0; i < d; i++)
        e[i] = e1[i];
}

void vect::lapatrat()
{
    for(int i = 0; i < d; i++)
        e[i] *= e[i];
}

void vect::afiseaza()
{
    for(int i = 0; i < d; i++)
        cout << e[i] << ' ';
    cout << endl;
}
```

Deși prin metoda de mai sus s-a realizat o legătură între datele problemei și funcțiile referitoare la aceste date, ele nu sunt protejate, deci pot fi accesate de orice funcție utilizator, nu numai de funcțiile membru. Acest neajuns se poate înlătura cu ajutorul claselor.

4.1.3. Declararea claselor

Un tip abstract de date clasă se declară ca și o structură, dar cuvântul cheie `struct` se înlocuiește cu `class`. Ca și în cazul structurilor referirea la tipul de dată clasă se face cu numele după cuvântul cheie `class` (numele clasei). Protecția datelor se realizează cu

modificatorii de protecție: *private*, *protected* și *public*. După modificatorul de protecție se pune caracterul ':'. Modificatorul *private* și *protected* reprezintă date protejate, iar *public* date neprotejate. Domeniul de valabilitate a modificatorilor de protecție este până la următorul modificator din interiorul clasei, modificatorul implicit fiind *private*. Menționăm că și în cazul structurilor putem să folosim modificatori de protecție, dar în acest caz modificatorul implicit este *public*.

De exemplu clasa vector se poate declara în modul următor:

```
class vector {
    int* e; //elementele vectorului
    int d; //dimensiunea vectorului
public:
    vector(int* e1, int d1);
    ~vector() { delete [] e; }
    void lapatrat();
    void afiseaza();
};
```

Se observă că datele membru *e* și *d* au fost declarate ca date de tip *private* (protejate), iar funcțiile membru au fost declarate publice (neprotejate). Bineînțeles, o parte din datele membru pot fi declarate publice, și unele funcții membru pot fi declarate protejate, dacă natura problemei cere acest lucru. În general, datele membru protejate pot fi accesate numai de funcțiile membru ale clasei respective și eventual de alte funcții numite *funcții prietene* (sau funcții *friend*).

O altă observație importantă referitoare la exemplul de mai sus este că inițializarea datelor membru și eliberarea zonei de memorie ocupată s-a făcut prin funcții membru specifice.

Datele declarate cu ajutorul tipului de dată clasă se numesc *obiectele* clasei, sau simplu *obiecte*. Ele se declară în mod obișnuit în forma:

```
nume_clasă listă_de_obiecte;
```

De exemplu, un obiect de tip vector se declară în modul următor:

```
vector v;
```

Inițializarea obiectelor se face cu o funcție membru specifică numită *constructor*. În cazul distrugerii unui obiect se apelează automat o altă funcție membru specifică numită *destructor*. În cazul exemplului de mai sus

```
vector(int* e1, int d1);
```

este un constructor, iar

```
~vector() { delete [] e; }
```

este un destructor.

Tipurile abstracte de date de tip *struct* pot fi și ele considerate clase cu toate elementele neprotejate. Constructorul de mai sus este declarat în interiorul clasei, dar nu este definit, iar destructorul este definit în interiorul clasei. Rezultă că destructorul este o funcție inline.

Definirea funcțiilor membru care sunt declarate, dar nu sunt definite în interiorul clasei se face ca și în cazul tipurilor abstracte de date de tip struct, folosind operatorul de rezoluție.

4.1.4. Membrii unei clase. Pointerul *this*

Referirea la datele respectiv funcțiile membru ale claselor se face cu ajutorul operatorilor `.` sau `->` ca și în cazul referirii la elementele unei structuri. De exemplu, dacă se declară:

```
vector v;  
vector* p;
```

atunci afișarea vectorului *v* respectiv a vectorului referit de pointerul *p* se face prin:

```
v.afiseaza();  
p->afiseaza();
```

În interiorul funcțiilor membru însă referirea la datele respectiv funcțiile membru ale clasei se face simplu prin numele acestora fără a fi nevoie de operatorul punct (`.`) sau săgeată (`->`). De fapt compilatorul generează automat un pointer special, pointerul *this*, la fiecare apel de funcție membru, și folosește acest pointer pentru identificarea datelor și funcțiilor membru.

Pointerul *this* va fi declarat automat ca pointer către obiectul curent. În cazul exemplului de mai sus pointerul *this* este adresa vectorului *v* respectiv adresa referită de pointerul *p*.

Dacă în interiorul corpului funcției membru *afiseaza* se utilizează de exemplu data membru *d*, atunci ea este interpretată de către compilator ca și *this->d*.

Pointerul *this* poate fi folosit și în mod explicit de către programator, dacă natura problemei necesită acest lucru.

4.1.5. Constructorul

Inițializarea obiectelor se face cu o funcție membru specifică numită constructor. Numele constructorului trebuie să coincidă cu numele clasei. O clasă poate să aibă mai mulți constructori. În acest caz aceste funcții membru au numele comun, ceea ce se poate face datorită posibilității de supraîncărcare a funcțiilor. Bineînțeles, în acest caz numărul și/sau tipul parametrilor formali trebuie să fie diferit, altfel compilatorul nu poate să aleagă constructorul corespunzător.

Constructorul nu returnează o valoare. În acest caz nu este permis nici folosirea cuvântului cheie *void*.

Prezentăm în continuare un exemplu de tip clasa cu mai mulți constructori, având ca date membru numele și prenumele unei persoane, și cu o funcție membru pentru afișarea numelui complet.

Fișierul **persoana.h**:

```

class persoana {
    char* nume;
    char* prenume;
public:
    persoana();           //constructor implicit
    persoana(char* n, char* p); //constructor
    persoana(const persoana& p1); //constructor de copiere
    ~persoana();         //destructor
    void afiseaza();
};

```

Fișierul **persoana.cpp**:

```

#include <iostream>
#include <cstring>
#include "persoana.h"

using namespace std;

persoana::persoana()
{
    nume = new char[1];
    *nume = 0;
    prenume = new char[1];
    *prenume = 0;
    cout << "Apelarea constructorului implicit." << endl;
}

persoana::persoana(char* n, char* p)
{
    nume = new char[strlen(n)+1];
    prenume = new char[strlen(p)+1];
    strcpy(nume, n);
    strcpy(prenume, p);
    cout << "Apelare constructor (nume, prenume).\n";
}

persoana::persoana(const persoana& p1)
{
    nume = new char[strlen(p1.nume)+1];
    strcpy(nume, p1.nume);
    prenume = new char[strlen(p1.prenume)+1];
    strcpy(prenume, p1.prenume);
    cout << "Apelarea constructorului de copiere." << endl;
}

persoana::~persoana()
{
    delete[] nume;
    delete[] prenume;
}

void persoana::afiseaza()
{
    cout << prenume << ' ' << nume << endl;
}

```

Fișierul **persoanaTest.cpp**:

```

#include "persoana.h"

int main() {
    persoana A;           //se apeleaza constructorul implicit
    A.afiseaza();
    persoana B("Stroustrup", "Bjarne");
    B.afiseaza();
    persoana *C = new persoana("Kernighan", "Brian");
    C->afiseaza();
    delete C;
    persoana D(B);       //echivalent cu persoana D = B;
                        //se apeleaza constructorul de copiere

    D.afiseaza();
    return 0;
}

```

Observăm prezența a doi constructori specifici: *constructorul implicit* și *constructorul de copiere*. Dacă o clasă are constructor fără parametri atunci el se va numi *constructor implicit*. *Constructorul de copiere* se folosește la inițializarea obiectelor folosind un obiect de același tip (în exemplul de mai sus o persoană cu numele și prenumele identic). Constructorul de copiere se declară în general în forma:

```
nume_clasă(const nume_clasă& obiect);
```

Cuvântul cheie *const* exprimă faptul că argumentul constructorului de copiere nu se modifică.

O clasă poate să conțină ca date membru obiecte ale unei alte clase. Declarând clasa sub forma:

```

class nume_clasa {
    nume_clasa_1 ob_1;
    nume_clasa_2 ob_2;
    ...
    nume_clasa_n ob_n;
    ...
};

```

antetul constructorului clasei *nume_clasa* va fi de forma:

```

nume_clasa(lista_de_argumente):
    ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)

```

unde *lista_de_argumente* respectiv *l_arg_i* reprezintă lista parametrilor formali ai constructorului clasei *nume_clasa* respectiv ai obiectului *ob_i*.

Din lista *ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)* pot să lipsească obiectele care nu au constructori definiți de programator, sau obiectul care se inițializează cu un constructor implicit, sau cu toți parametrii implicați.

Dacă clasa conține date membru de tip obiect atunci se vor apela mai întâi constructorii datelor membru, iar după aceea corpul de instrucțiuni al constructorului clasei respective.

Fișierul `pereche.cpp`:

```
#include <iostream>
#include "persoana.h"

using namespace std;

class pereche {
    persoana sot;
    persoana sotie;
public:
    pereche()                //definitia constructorului implicit
    {                        //se vor apela constructorii impliciti
    }                        //pentru obiectele sot si sotie
    pereche(persoana& sotul, persoana& sotia);
    pereche(char* nume_sot, char* prenume_sot,
            char* nume_sotie, char* prenume_sotie):
        sot(nume_sot, prenume_sot),
        sotie(nume_sotie, prenume_sotie)
    {
    }
    void afiseaza();
};

inline pereche::pereche(persoana& sotul, persoana& sotia):
    sot(sotul), sotie(sotia)
{
}

void pereche::afiseaza()
{
    cout << "Sot: ";
    sot.afiseaza();
    cout << "Sotie: ";
    sotie.afiseaza();
}

int main() {
    persoana A("Pop", "Ion");
    persoana B("Popa", "Ioana");
    pereche AB(A, B);
    AB.afiseaza();
    pereche CD("C", "C", "D", "D");
    CD.afiseaza();
    pereche EF;
    EF.afiseaza();
    return 0;
}
```

Observăm că în cazul celui de al doilea constructor, parametrii formali *sot* și *sotie* au fost declarați ca și referințe la tipul *persoana*. Dacă ar fi fost declarați ca parametri formali de tip *persoana*, atunci în cazul declarației:

```
pereche AB(A, B);
```

constructorul de copiere s-ar fi apelat de patru ori. În astfel de situații se creează mai întâi obiecte temporale folosind constructorul de copiere (două apeluri în cazul de față), după care se execută constructorii datelor membru de tip obiect (încă două apeluri).

4.1.6. Destructorul

Destructorul este funcția membru care se apelează în cazul distrugerii obiectului. Destructorul obiectelor globale se apelează automat la sfârșitul funcției *main* ca parte a funcției *exit*. Deci, nu este indicată folosirea funcției *exit* într-un destructor, pentru că acest lucru duce la un ciclu infinit. Destructorul obiectelor locale se execută automat la terminarea blocului în care s-au definit. În cazul obiectelor alocate dinamic, de obicei destructorul se apelează indirect prin operatorul *delete* (obiectul trebuie să fi fost creat cu operatorul *new*). Există și un mod explicit de apelare a destructorului, în acest caz numele destructorului trebuie precedat de numele clasei și operatorul de rezoluție.

Numele destructorului începe cu caracterul `~` după care urmează numele clasei. Ca și în cazul constructorului, destructorul nu returnează o valoare și nu este permisă nici folosirea cuvântului cheie `void`. Apelarea destructorului în diferite situații este ilustrată de următorul exemplu. Fișierul `destruct.cpp`:

```
#include <iostream>
#include <cstring>

using namespace std;

class scrie { //scrie pe stdout ce face.
    char* nume;
public:
    scrie(char* n);
    ~scrie();
};

scrie::scrie(char* n)
{
    nume = new char[strlen(n)+1];
    strcpy(nume, n);
    cout << "Am creat obiectul: " << nume << '\n';
}

scrie::~~scrie()
{
    cout << "Am distrus obiectul: " << nume << '\n';
    delete nume;
}

void functie()
{
    cout << "Apelare functie" << '\n';
    scrie local("Local");
}

scrie global("Global");

int main() {
    scrie* dinamic = new scrie("Dinamic");
    functie();
    cout << "Se continua programul principal" << '\n';
    delete dinamic;
}
```

```
    return 0;  
}
```

5. Probleme propuse

1. Scrieți un program într-unul din limbajele de programare C++, Java, C# care:
 - a. Definește o clasă **Student** având:
 - un atribut **nume** de tip șir de caractere;
 - un atribut **note** conținând un șir de note (numere întregi),
constructori, accesori și o metodă care calculează media notelor studentului.
 - b. Definește o funcție care primind un obiect de tip **Student** returnează adevărat dacă toate notele elevului sunt >4.
 - c. Scrieți specificațiile metodelor definite în clasa **Student** precum și a funcției de la punctul b.

2. Scrieți un program într-unul din limbajele de programare C++, Java, C# care:
 - a. Definește o clasă **Student** având:
 - un atribut **nume** de tip șir de caractere;
 - un atribut **note** conținând un șir de note (numere întregi),
constructori, accesori și o metodă care calculează media notelor studentului.
 - b. Definește un subprogram care primind un obiect de tip **Student** tipărește numele studentului și notele acestuia în ordine descrescătoare.
 - c. Scrieți specificațiile metodelor definite în clasa **Student** precum și a subprogramului de la punctul b.

6. Bibliografie generală

1. Alexandrescu, *Programarea modernă în C++*. Programare generică și modele de proiectare aplicate, Editura Teora, 2002.
2. Angster Erzsébet: *Objektumorientált tervezés és programozás Java*, 4KÖR Bt, 2003.
3. Bjarne Stroustrup: *A C++ programozási nyelv*, Kiskapu kiadó, Budapest, 2001.
4. Bjarne Stroustrup: *The C++ Programming Language Special Edition*, AT&T, 2000.
5. Boian F.M. Frentiu M., Lazăr I. Tambulea L. *Informatica de bază*. Presa Universitară Clujeana, Cluj, 2005
6. Bradley L. Jones: *C# mesteri szinten 21 nap alatt*, Kiskapu kiadó, Budapest, 2004.
7. Bradley L. Jones: *SAMS Teach Yourself the C# Language in 21 Days*, Pearson Education, 2004.
8. Cormen, T., Leiserson, C., Rivest, R., *Introducere în algoritmi*, Editura Computer Libris Agora, Cluj, 2000
9. Eckel B., *Thinking in C++*, vol I-II, <http://www.mindview.net>
10. Ellis M.A., Stroustrup B., *The annotated C++ Reference Manual*, Addison-Wesley, 1995
11. Frentiu M., Lazăr I. *Bazele programării*. Partea I-a: Proiectarea algoritmilor
12. Herbert Schildt: *Java. The Complete Reference*, Eighth Edition, McGraw-Hill, 2011.
13. Robert Sedgewick: *Algorithms*, Addison-Wesley, 1984
14. Simon Károly, *Kenyerünk Java. A Java programozás alapjai*, Presa Universitară Clujeană, 2010.