Manual de Informatică pentru licenţă iulie și septembrie 2017

Specializarea Matematică-Informatică

Tematica generală:

Algoritmică şi programare.

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

Cuprins

ALGORITMICĂ ŞI PROGRAMARE **3**

1. CĂUTĂRI ŞI SORTĂRI **3**

1.1. CĂUTĂRI 3

1.1.1. Căutare secvenţială *3*

1.1.2. Căutare binară *4*

1.2. SORTĂRI 5

1.2.1. Sortare prin selecţie *6*

1.2.2. Bubble sort *6*

1.2.3. Quicksort *7*

2. METODELE BACKTRACKING ŞI DIVIDE ET IMPERA **8**

2.1. METODA BACKTRACKING 8

2.2. METODA "DIVIDE ET IMPERA" 12

3. ALGORITMI ŞI SPECIFICĂRI **13**

3.1. SCRIEREA UNUI ALGORITM PORNIND DE LA O SPECIFICAŢIE DATĂ 13

4. CONCEPTE OOP ÎN LIMBAJE DE PROGRAMARE **14**

4.1. NOŢIUNEA DE CLASĂ 14

4.1.1. Realizarea protecţiei datelor prin metoda programării modulare *14*

4.1.2. Tipuri abstracte de date *16*

4.1.3. Declararea claselor *17*

4.1.4. Membrii unei clase. Pointerul this *18*

4.1.5. Constructorul *19*

4.1.6. Destructorul *22*

5. PROBLEME PROPUSE **23**

6. BIBLIOGRAFIE GENERALĂ **24**

Algoritmică şi programare

1. Căutări şi sortări

1.1. Căutări

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

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

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

$$k_1 < k_2 < .... < k_n .$$

Uneori este util să aflăm nu numai dacă există un articol cu cheia dorită ci şi 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 $a,n,(k_i, i=1,n);$

Precondiţia: $n \in N, n \geq 1$ *şi* $k_1 < k_2 < .... < k_n$;

Rezultate $p;$

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

1.1.1. Căutare secvenţială

O primă metodă este căutarea **secvenţială**, 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∈N, n≥1 şi k1 < k2 < .... < kn}
                                                  {Se caută p astfel ca: (p=1 şi a ≤ k1) sau}
                                                  { (p=n+1 şi a>kn) sau (1<p≤n) şi (kp-1 < a ≤ kp).
  Fie p := 0;                                                    {Cazul "încă negasit"}
  Dacă a ≤ k1 atunci p := 1 altfel
     Dacă a > kn atunci p := n + 1 altfel
          Pentru i := 2; n execută
               Dacă (p = 0) şi (a ≤ ki) 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 comparări, întrucât contorul *i* va lua toate valorile de la 2 la *n*. Cele *n* chei împart axa reală în *n*+1 intervale. Tot atâtea comparări se vor efectua în *n*-1 din cele n+1 intervale în care se poate afla cheia căutată, deci complexitatea medie are acelaşi ordin de mărime ca şi complexitatea în cel mai rău caz.

Evident că în multe situaţii 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∈N, n≥1 şi k1 < k2 < .... < kn}
                                                  {Se caută p astfel ca: p=1 şi a ≤ k1) sau }
                                                  {(p=n+1 şi a>kn) sau (1<p≤n) şi (kp-1 < a ≤ kp).
   Fie p:=1;
   Dacă a>k1 atunci
      Câttimp p≤n şi a>kp execută p:=p+1 sfcât
   sfdacă
sf-CautSucc
```

În cel mai rău caz şi acest algoritm face acelaşi număr de operaţii ca şi subalgoritmul *Cautsecv*. În medie numărul operaţiilor este jumătate din numărul mediu de operaţii efecuat de subalgoritmul *Cautsecv* deci complexitatea este aceeaşi.

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 relaţie se află cheia articolului aflat în mijlocul colecţiei cu cheia de căutare. În urma acestei verificări căutarea se continuă doar într-o jumătate a colecţiei. În acest mod, prin înjumătăţiri succesive se micşorează volumul colecţiei rămase pentru căutare. Căutarea binară se poate realiza practic prin apelul funcţiei `BinarySearch(a, n, K, 1, n)`, descrisă mai jos, folosită în subalgoritmul dat în continuare.

```
Subalgoritmul CautBin(a, n, K, p) este:           {n∈N, n≥1 şi k1 < k2 < .... < kn}
                                                  {Se caută  p astfel ca: (p=1 şi a ≤ k1) sau}
                                                  {(p=n+1 şi a>kn) sau (1<p≤n) şi (kp-1 < a ≤ kp)}
 Dacă a ≤ k1 atunci p := 1 altfel
     Dacă a > kn atunci p := n+1 altfel
         P := BinarySearch(a, n, K, 1, n)
     sfdacă
 sfdacă
sf-CautBin
```

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

În funcţia `BinarySearch` descrisă mai sus, variabilele `St` şi `Dr` reprezintă capetele intervalului de căutare, iar `m` reprezintă mijlocul acestui interval. Prin această metodă, într-o colecţie având `n` elemente, rezultatul căutării se poate furniza după cel mult $log_2 n$ comparări. Deci complexitatea în cel mai rău caz este direct proporţională cu $log_2 n$. Fără a insista asupra demonstraţiei, menţionăm că ordinul de mărime al complexităţii medii este acelaşi.

Se observă că funcţia `BinarySearch` se apelează recursiv. Se poate înlătura uşor recursivitatea, aşa cum se poate vedea în următoarea funcţie:

```
Funcţia BinSeaNerec(a, n, K, St, Dr) este:
 Câttimp Dr - St > 1 execută
         m := (St+Dr) Div 2;
         Dacă a ≤ k_m atunci Dr := m altfel St := m sfdacă
 sfcât
 BinSeaNerec := Dr
sf-BinSeaNerec
```

1.2. Sortări

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

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

Date n,K; $\{K=(k_1,k_2,...,k_n)\}$

Precondiţia: $k_i \in R$, $i=1,n$

Rezultate K';

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

Deci $k_1 \le k_2 \le ... \le k_n$.

1.2.1. Sortare prin selecţie

O primă tehnică numită "*Selecţie*" se bazează pe următoarea idee: se determină poziţia elementului cu cheie de valoare minimă (respectiv maximă), după care acesta se va interschimba cu primul element. Acest procedeu se repetă pentru subcolecţia 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 k1 ≤ k2 ≤ .... ≤ kn }
 Pentru i := 1; n-1 execută
   Fie ind := i;
   Pentru j := i + 1; n execută
      Dacă kj < kind atunci ind := j sfdacă
   sfpentru
   Dacă i < ind atunci t := ki; ki := kind; kind := t sfdacă
 sfpentru
sf-Selectie
```

Se observă că numărul de comparări este:

$$(n-1)+(n-2)+...+2+1=n(n-1)/2$$

indiferent de natura datelor. Deci complexitatea medie, dar şi 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 relaţia dorită, ele vor fi interschimbate. Procesul de comparare se va încheia în momentul în care toate perechile de elemente consecutive sunt în relaţia de ordine dorită.

```
Subalgoritmul BubbleSort(n, K) este:
  Repetă
    Fie kod := 0;                                  {Ipoteza "este ordine"}
    Pentru i := 2; n execută
      Dacă ki-1 > ki atunci
        t := ki-1;
        ki-1 := ki;
        ki := 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)+ ... +2+1 = n(n-1)/2 comparări, 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 numeşte "*QuickSort*" şi 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 subşir precizat prin limita inferioară şi limita superioară a indicilor acestuia. Apelul procedurii pentru ordonarea întregului şir este : `QuickSort(n, K, 1, n)`, unde `n` reprezintă numărul de articole ale colecţiei date. Deci

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

Procedura `QuickSort(n, K, St, Dr)` va realiza ordonarea subşirului k_{St}, k_{St+1}, ..., k_{Dr}. Acest subşir va fi rearanjat astfel încât k_{St} să ocupe poziţia lui finală (când şirul este ordonat). Dacă `i` este această poziţie, şirul va fi rearanjat astfel încât următoarea condiţie să fie îndeplinită:

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

Odată realizat acest lucru, în continuare va trebui doar să ordonăm subşirul k_{St}, k_{St+1}, ... , k_{i-1} prin apelul recursiv al procedurii `QuickSort(n, K, St, i-1)` şi apoi subşirul k_{i+1}, ..., k_{Dr} prin apelul `QuickSort(n, K, i+1, Dr)`. Desigur ordonarea acestor două subşiruri (prin apelul recursiv al procedurii) mai este necesară doar dacă acestea conţin cel puţin două elemente.

Procedura `QuickSort` este prezentată în continuare :

```
Subalgoritmul QuickSort (n, K, St, Dr) este:
   Fie i := St; j := Dr; a := k_i;
   Repetă
     Câttimp k_j ≥ a şi (i < j) execută j := j - 1 sfcât
     k_i := k_j;
     Câttimp k_i ≤ a şi (i < j) execută i := i + 1 sfcât
     k_j := k_i ;
  pânăcând i = j sfrep
  Fie k_i := 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 $\boldsymbol{O(n\log_2 n)}$.

2. Metodele backtracking şi divide et impera

2.1. Metoda backtracking

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

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

Problema 1. (Generarea permutărilor) Fie *n* un număr natural. Determinaţi permutările numerelor 1, 2, ..., *n*.

O soluţie 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
                              Tipăreşte posibil
                    sfdacă
               sfpentru
          sfpentru
     sfpentru
sf-Permutări1
```

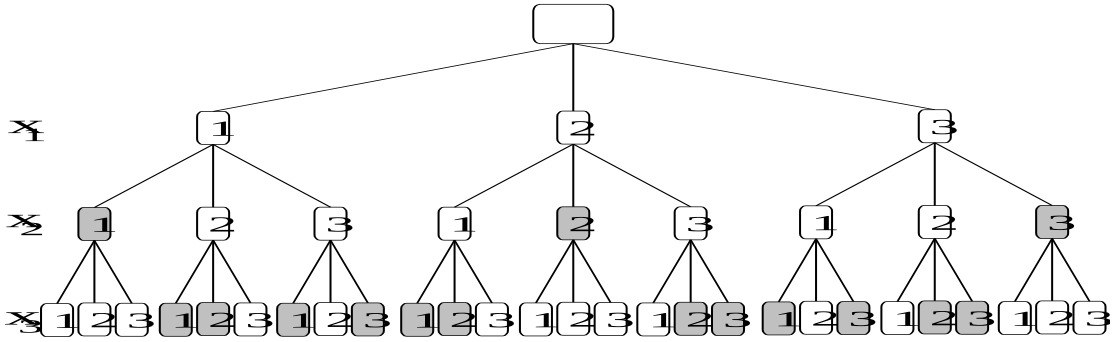

Figura 1.1. Reprezentare grafică a produsului cartezian $\{1, 2, 3\}^3$

Observaţii privind subalgoritmul *Permutări1*:

- Pentru *n* oarecare nu putem descrie un algoritm care să conţină *n* cicluri în textul sursă.
- Numărul total de vectori verificaţi este 3^3, iar în general n^n. Vectorii *posibil* verificaţi sunt reprezentaţi 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 îmbunătăţire a acestor algoritmi ar consta în a verifica anumite condiţii 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 promiţător* (pentru a fi o permutare), în schimb vectorul (1, 1, ...) nu este. Vectorul (1, 3, ...) satisface anumite *condiţii de continuare* (pentru a ajunge la soluţie) - are componente distincte. Nodurile haşurate din Figura 1.1 constituie valori care nu conduc la o soluţie.

Vom descrie un algoritm general pentru metoda Bactracking după care vom particulariza acest algoritm pentru problemele enunţate la începutul secţiunii. Pentru început vom face câteva observaţii şi notaţii privind metoda Backtracking aplicată unei probleme în care soluţiile se reprezintă pe vectori, nu neapărat de lungime fixă:

- spaţiul de căutare a soluţiilor (spaţiul soluţiilor posibile): $S = S_1 \times S_2 \times ... \times S_n$;
- *posibil* este vectorul pe care se reprezintă soluţiile;
- *posibil*[1..*k*] $\in S_1 \times S_2 \times ... \times S_k$ este un vector care poate conduce sau nu la o soluţie; *k* reprezintă indice pentru vectorul *posibil*, respectiv nivel în arborele care redă grafic procesul de căutare (Figura 1.2).
- *posibil*[1..*k*] este promiţător dacă satisface condiţii care pot conduce la o soluţie;
- *soluţie*(*n, k, posibil*) funcţie care verifică dacă vectorul (promiţător) *posibil*[1..*k*] este soluţie a problemei.

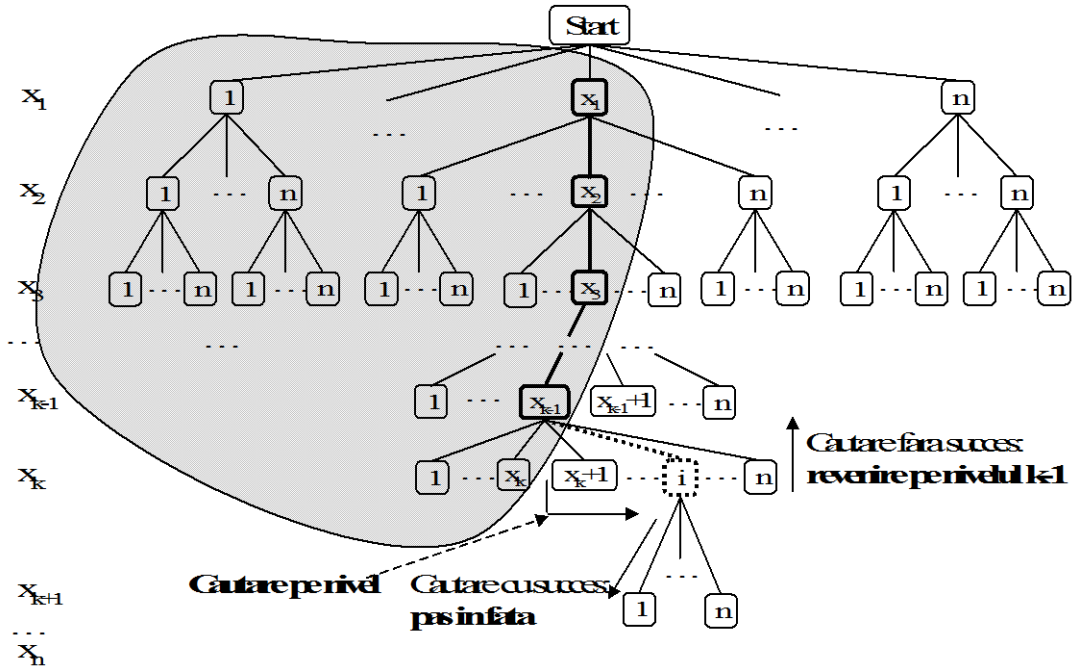

Figura 1.2. Spaţiul soluţiilor posibile pentru generarea permutărilor

Procesul de căutare poate fi urmărit în algoritmul care urmează:

```
Algoritmul Backtracking  este:                          {varianta nefinisată}
Fie k := 1
@Iniţializează căutarea pe nivelul k (= 1)
Câttimp k > 0 execută                          {posibil[1..k-1] este promiţător}
     @Caută (secvenţial)  pe nivelul  k o valoare v, pentru a completa în
          continuare vectorul posibil[1..k-1] astfel încât  posibil[1..k]  să
          fie promiţător
     Dacă căutarea este cu succes
          atunci Fie posibil[k] := v        {posibil[1..k] este promiţător}
              Dacă soluţie(n, k, posibil)
                atunci                  {o soluţie! (rămânem pe nivelul k)}
                    Tipareşte posibil[1..k]
                altfel                          {e doar un vector promiţător}
                    @Initializeaza cautarea pe nivelul k+1
                    Fie k := k + 1              {pas în faţă (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 nestandard prezente. Astfel, avem nevoie de funcţia booleană

```
condiţii-continuare(k, posibil, v)
```

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

Apoi, pentru a iniţializa căutarea la nivelul *j* avem nevoie de a alege un element fictiv din mulţimea S_j, activitate realizată de funcţia

```
init(j)
```

care returnează acest element fictiv, care are rolul de a indica faptul că din mulţimea S încă nu s-a ales nici un element, deci după el urmează primul element propriu din această mulţime. Pentru a căuta o valoare pe nivelul j, în ipoteza că valoarea curentă nu e bună, avem nevoie de funcţia 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* şi *False* în cazul în care nu mai există alte valori în S_j, deci nu mai poate fi făcută alegerea. Cu aceste notaţii algoritmul devine:

```
Algoritmul Backtracking  este:                          {versiune finală}
Fie k := 1;
posibil[1] := init(1);
Câttimp k > 0 execută                          {posibil[1..k-1] este promiţător}
     Fie Găsit := false; v := posibil[k] ;
     Câttimp Următor(k, v,urm) şi not Găsit execută
          Fie v := urm;
          Dacă condiţii-continuare(k, posibil, v) atunci
               Găsit := true
          sfdacă
     sfcât
     Dacă Găsit
          atunci Fie posibil[k] := v;       {posibil[1..k] este promiţător}
              Dacă soluţie(n, k, posibil)
                atunci                 {o soluţie! (rămânem pe nivelul k)}
                    Tipareşte posibil[1..k]
                altfel                          {e doar un vector promiţător}
                    Fie k := k + 1;             {pas în faţă (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* şi funcţiile *condiţii-continuare* şi *soluţie* sunt dependente de problema care se rezolvă. De exemplu, pentru generarea permutărilor funcţiile menţionate sunt:

```
Funcţia init(k) este:
   Init := 0
sf-init;

Funcţia Următor(k, v, urm) este:
 Dacă  v < n
   atunci Următor := True; urm := v + 1
   altfel Următor := False
 sfdacă
sf-urmator

Funcţia conditii-continuare(k, posibil, v) este:
    Kod := True;   i := 1;
    Câttimp  kod şi (i < k)  execută
        Dacă posibil[i] = v atunci kod := False sfdacă
        i := i + 1;
```

```
          sfcât
          conditii-continuare:=kod
      sf-conditii

      Funcţia soluţii(n, k, posibil) este:
         Soluţii := (k = n)
      sf-solutii
```

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

```
Subalgoritmul Backtracking(k, posibil)  este:

  {posibil[1..k] este promiţător}
  Dacă soluţie(n, k, posibil)  atunci
            {o soluţie! terminare apel recursiv, astfel}
            Tipareste posibil[1..k]
           {rămânem pe acelaşi nivel}
  altfel
     Pentru fiecare v valoare posibilă pentru posibil[k+1] execută
           Dacă condiţii-continuare(k + 1, posibil, v) atunci
                 posibil[k + 1] := v
                Backtracking(k + 1, posibil)
               {pas in faţă}
            sfdacă
       sfpentru
  sfdacă
  {terminare apel Backtracking(k, posibil)}
sf-Backtracking {deci, pas în spate (revenire)}
```

cu apelul iniţial **Cheamă** `Backtracking(0, posibil).`

2.2. Metoda "divide et impera"

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

Metoda se aplica problemelor care pot fi descompuse în subprobleme independente, similar problemei iniţiale, de dimensiuni mai mici şi care pot fi rezolvabile foarte uşor. Ea se aplică atunci când rezolvarea problemei *P* pentru setul de date *D* se poate face prin rezolvarea aceleiaşi probleme *P* pentru alte seturi de date d_1, d_2, ..., d_k de volum mai mic decât *D*.

Observaţii:

- Împărţirea se face până când se obţine o problemă rezolvabilă imediat.
- Subproblemele în care se descompune problema inițială trebuie să fie independente. Dacă subproblemele nu sunt independente, se aplică alte metode de rezolvare.
- Tehnica admite şi 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:

```
Sublalgoritmul 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)
         @ construieste rezultatul final prin utilizarea rezultatelor
         partiale din apelurile de mai sus
    sfdacă
sf-NumeAlg
```

3. Algoritmi şi specificări

Algoritmi și specificări. Scrierea unui algoritm pornind de la o specificație dată.

3.1. Scrierea unui algoritm pornind de la o specificaţie dată

Problema 1

Scrieţi o funcţie care satisface următoarea specificaţie:

Date nr;

Precondiţia: $nr \in N, nr \geq 1$

Rezultate $l_1, l_2, ..., l_n$*;*

Postcondiţia: $n \in N^*, \left[\frac{nr}{l_i}\right] \cdot l_i = nr \; \forall 1 \leq i \leq n, l_i \neq l_j \; \forall i \neq j, 1 \leq i, j \leq n$, n este maximal

Problema 2

Scrieţi o funcţie care satisface următoarea specificaţie:

Date $n, L=(l_1,l_2,...,l_n)$;
Precondiţia: $l_i \in R$, $i=1,n$
Rezultate $R=(r_1,r_2,...,r_n)$;
Postcondiţia: R este o permutare a lui L, $r_1 \geq r_2 \geq ... \geq r_n$.

Problema 3

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

Când merge la cumpărături, Ana îşi pregăteşte tot timpul o listă de cumpărături: denumire, cantitate, raion (alimente, îmbrăcăminte, încălţăminte, consumabile), preţ estimat. Se cere să se afişeze lista de cumpărături a Anei ordonată alfabetic după raion, lista ordonată descrescător după cantitate, precum şi lista Anei de la un anumit raion. Se cere să se calculeze şi un preţ estimativ al cheltuielilor Anei.

Problema 4

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

Să se scrie un program care citeşte un şir de numere întregi nenule. Introducerea unui şir se încheie odată cu citirea valorii 0. În şirul citit programul va elimina secvenţele de elemente consecutive strict pozitive de lungime mai mare decât 3 (dacă există), dupa care va tipări şirul obţinut.

4. Concepte OOP în limbaje de programare

4.1. Noţiunea de clasă

4.1.1. Realizarea protecţiei datelor prin metoda programării modulare

Dezvoltarea programelor prin programare procedurală înseamnă folosirea unor funcţii şi proceduri pentru scrierea programelor. În limbajul C lor le corespund funcţiile care returnează o valoare sau nu. Însă în cazul aplicaţiilor mai mari ar fi de dorit să putem realiza şi o protecţie corespunzătoare a datelor. Acest lucru ar însemna că numai o parte a funcţiilor să aibă acces la datele problemei, acelea care se referă la datele respective. Programarea modulară oferă o posibilitate de realizare a protecţiei datelor prin folosirea clasei de memorie static. Dacă într-un fişier se declară o dată aparţinând clasei de memorie statică în afara

funcţiilor, atunci ea poate fi folosită începând cu locul declarării până la sfârşitul modulului respectiv, dar nu şi 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 funcţii corespunzătoare pentru iniţializarea vectorului, eliberarea zonei de memorie ocupate şi ridicarea la pătrat, respectiv afişarea elementelor vectorului. O posibilitate de implementare a modulului este prezentată în fişierul **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 obţinând un program obiect. Un exemplu de program principal este prezentat în fişierul **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 deşi 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 şi 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 şi operaţiile (funcţiile) 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 şi declararea sau definirea funcţiilor 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();
};
```

Funcţiile declarate sau definite în interiorul structurii vor fi numite *funcţii membru* iar datele *date membru*. Dacă o funcţie membru este definită în interiorul structurii (ca şi funcţia *distr* din exemplul de mai sus), atunci ea se consideră funcţie *inline*. Dacă o funcţie membru se defineşte în afara structurii, atunci numele funcţiei se va înlocui cu numele tipului abstract urmat de operatorul de rezoluţie (::) şi numele funcţiei membru. Astfel funcţiile *init*, *lapatrat* şi *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;
}
```

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

4.1.3. Declararea claselor

Un tip abstract de date clasă se declară ca şi o structură, dar cuvântul cheie struct se înlocuieşte cu *class*. Ca şi în cazul structurilor referirea la tipul de dată clasă se face cu numele după cuvântul cheie class (numele clasei). Protecţia datelor se realizează cu modificatorii de protecţie: *private, protected* şi *public*. După modificatorul de protecţie se pune caracterul ':'. Modificatorul *private* şi *protected* reprezintă date protejate, iar *public* date neprotejate. Domeniul de valabilitate a modificatorilor de protecţie este până la următorul modificator din interiorul clasei, modificatorul implicit fiind *private*. Menţionăm că şi în cazul structurilor putem să folosim modificatori de protecţie, 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* şi *d* au fost declarate ca date de tip *private* (protejate), iar funcţiile membru au fost declarate publice (neprotejate). Bineînţeles, o parte din datele membru pot fi declarate publice, şi unele funcţii membru pot fi declarate protejate, dacă natura problemei cere acest lucru. În general, datele membru protejate pot fi accesate numai de funcţiile membru ale clasei respective şi eventual de alte funcţii numite *funcţii prietene (*sau funcţii *friend).*

O altă observaţie importantă referitoare la exemplul de mai sus este că iniţializarea datelor membru şi eliberarea zonei de memorie ocupată s-a făcut prin funcţii membru specifice.

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

```
nume_clasă listă_de_obiecte;
```

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

```
vector v;
```

Iniţializarea obiectelor se face cu o funcţie membru specifică numită *constructor*. În cazul distrugerii unui obiect se apelează automat o altă funcţie 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 şi 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 funcţie inline. Definirea funcţiilor membru care sunt declarate, dar nu sunt definite în interiorul clasei se face ca şi în cazul tipurilor abstracte de date de tip struct, folosind operatorul de rezoluţie.

4.1.4. Membrii unei clase. Pointerul this

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

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

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

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

În interiorul funcţiilor membru însă referirea la datele respectiv funcţiile 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 funcţie membru, şi foloseşte acest pointer pentru identificarea datelor şi funcţiilor 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 funcţiei membru *afiseaza* se utilizează de exemplu data membru *d,* atunci ea este interpretată de către compilator ca şi this->d.

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

4.1.5. Constructorul

Iniţializarea obiectelor se face cu o funcţie membru specifică numită constructor. Numele constructorului trebuie să coincidă cu numele clasei. O clasă poate să aibă mai mulţi constructori. În acest caz aceste funcţii membru au numele comun, ceea ce se poate face datorită posibilităţii de supraîncărcare a funcţiilor. Bineînţeles, în acest caz numărul şi/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 mulţi constructori, având ca date membru numele şi prenumele unei persoane, şi cu o funcţie membru pentru afişarea numelui complet.

Fişierul **`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();
};
```

Fişierul **`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;
}
```

Fişierul **`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 copire
  D.afiseaza();
  return 0;
}
```

Observăm prezenţa a doi constructori specifici: *constructorul implicit* şi *constructorul de copiere*. Dacă o clasă are constructor fără parametri atunci el se va numi *constructor implicit*. *Constructorul de copiere* se foloseşte la iniţializarea obiectelor folosind un obiect de acelaşi tip (în exemplul de mai sus o persoană cu numele şi 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 conţină 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 definiţi de programator, sau obiectul care se iniţializează cu un constructor implicit, sau cu toţi parametrii impliciţi.

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

Fişierul **`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* şi *sotie* au fost declaraţi ca şi referinţe la tipul *persoana*. Dacă ar fi fost declaraţi ca parametri formali de tip *persoana*, atunci în cazul declaraţiei:

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

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

4.1.6. Destructorul

Destructorul este funcţia membru care se apelează în cazul distrugerii obiectului. Destructorul obiectelor globale se apelează automat la sfârşitul funcţiei *main* ca parte a funcţiei *exit*. Deci, nu este indicată folosirea funcţiei *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 şi un mod explicit de apelare a destructorului, în acest caz numele destructorului trebuie precedat de numele clasei şi operatorul de rezoluţie.

Numele destructorului începe cu caracterul ~ după care urmează numele clasei. Ca şi în cazul constructorului, destructorul nu returnează o valoare şi nu este permisă nici folosirea cuvântului cheie void. Apelarea destructorului în diferite situaţii este ilustrată de următorul exemplu. Fişierul **`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. Scrieţi un program într-unul din limbajele de programare C++, Java, C# care:
   a. Defineşte o clasă **Student** având:
      - un atribut ***nume*** de tip şir de caractere;
      - un atribut ***note*** conţinând un şir de note (numere întregi),

      constructori, accesori şi o metodă care calculează media notelor studentului.
   b. Defineşte o funcţie care primind un obiect de tip **Student** returnează adevărat dacă toate notele elevului sunt >4.
   c. Scrieţi specificaţiile metodelor definite în clasa **Student** precum şi a funcţiei de la punctul b.

2. Scrieţi un program într-unul din limbajele de programare C++, Java, C# care:
   a. Defineşte o clasă **Student** având:
      - un atribut ***nume*** de tip şir de caractere;
      - un atribut ***note*** conţinând un şir de note (numere întregi),

      constructori, accesori şi o metodă care calculează media notelor studentului.
   b. Defineşte un subprogram care primind un obiect de tip **Student** tipăreşte numele studentului şi notele acestuia în ordine descrescătoare.
   c. Scrieţi specificaţiile metodelor definite în clasa **Student** precum şi a subprogramului de la punctul b.

6. Bibliografie generală

1. Alexandrescu, *Programarea modernă in C++. Programare generică si 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.