PROGRAMARE ȘI STRUCTURI DE DATE

CURS 3

Lect. dr. Oneț-Marian Zsuzsanna

Facultatea de Matematică și Informatică UBB
în colaborare cu NTT Data

În cursul 1 și 2:

- Algoritmi și programe
- Componentele de bază ale unui algoritm:
  - Variabile
  - Instrucțiunea de citire (date de la tastatură) și afișare (mesaje pe ecran)
  - Instrucțiunea condițională
  - Instrucțiuni repetitive (ciclu *pentru* și ciclu *câttimp*)
- Tablouri
- Vizibilitatea variabilelor

Cuprins

Tipuri definite de utilizator I

- Să ne imaginăm că trebuie să facem un program care să ajute la gestionarea notelor și absențelor elevilor într-o școală generală. Vrem să reținem toți elevii, pentru fiecare elev notele primite și absențele avute. Vrem să structurăm elevii pe clase și pentru fiecare clasă să avem și numele dirigintelor.
- Înainte de a începe orice implementare, trebuie să ne decidem cum vrem să reprezentăm datele (informațiile) din problemă.

Gestionarea elevilor

- O notă trebuie reprezentată prin:

Gestionarea elevilor

- O notă trebuie reprezentată prin:
  - Nota efectivă, de ex. 8
  - Disciplina la care a fost primită, de ex. Matematică
  - Data la care a fost primită, de ex. 5.12.2016
  - Poate și o valoare booleană care să arate dacă e vorbă de o teză sau nu
- O absență trebuie reprezentată prin:

Gestionarea elevilor

- O notă trebuie reprezentată prin:
  - Nota efectivă, de ex. 8
  - Disciplina la care a fost primită, de ex. Matematică
  - Data la care a fost primită, de ex. 5.12.2016
  - Poate și o valoare booleană care să arate dacă e vorbă de o teză sau nu
- O absență trebuie reprezentată prin:
  - Data la care a fost primită, de ex. 5.12.2016
  - Disciplina la care a fost primită, de ex. Istorie
  - O valoare booleană care să arate dacă a fost motivată sau nu

Gestionarea elevilor

- Un elev trebuie reprezentat prin:

Gestionarea elevilor

- Un elev trebuie reprezentat prin:
  - Nume și prenume, de ex. Adam Corina
  - Inițiala părintelui, de ex. S.
  - Data nașterii, CNP, adresă, etc.

Gestionarea elevilor

- Un elev trebuie reprezentat prin:
  - Nume și prenume, de ex. Adam Corina
  - Inițiala părintelui, de ex. S.
  - Data nașterii, CNP, adresă, etc.
  - Fiecare elev trebuie să aibă mai multe note (o listă de note). *Lista* în cazul nostru înseamnă tablou. Dar, din moment ce fiecare notă este reprezentată de 4 valori, ne trebuie de fapt 4 tablouri:
    - Notele [10, 7, 4, 9]
    - Disciplinele [“Fizică”, “Matematică”, “Chimie”, “Română”]
    - Date [“5.12.2016”, “10.12.2016”, “7.1.2017”, “21.11.2016”]
    - Teză sau nu [Fals, Adevărat, Fals, Fals]
    - Ne trebuie și numărul de note, ca să știm câte elemente sunt în aceste tablouri

Gestionarea elevilor

- În mod similar ar trebui să facem cu absențe, ne trebuie încă 3 tablouri, plus o variabilă care să arate câte absențe are elevul.

- În total pentru un elev vom avea nevoie de
  - 8 variabile *simple* (nume, prenume, inițiala părintelui, data nașterii, CNP, adresă, numărul notelor și numărul absențelor)
  - 7 tablouri (4 pentru note și 3 pentru absențe)

Gestionarea elevilor

- O clasă trebuie reprezentată prin:

Gestionarea elevilor

- O clasă trebuie reprezentată prin:
  - Numele dirigintelui
  - Lista de elevi. Cum un elev are nume, inițiala tatălui, și alte valori ne trebuie de fapt mai multe tablouri. Dar ce facem cu notele (unde am deja 4 tablouri), și cu absențe? Ne trebuie câte un tablou bidimensional (matrice): pe prima linie am notele pentru primul elev, pe a 2-a linie notele pentru al 2-lea elev, etc.
  - Deci, pentru a reprezenta o clasă ne trebuie: un string (numele dirigintelui), 8 tablouri (numele și prenumele elevilor, inițiala tatălui, data nașterii, CNP, adresă, numărul notelor, numărul absențelor) și 7 matrici (4 pentru note - nota, disciplină, data, teză - și 3 pentru absențe - data, disciplina, motivat -).

Gestionarea elevilor

- O clasă trebuie reprezentată prin:
  - Numele dirigintelui
  - Lista de elevi. Cum un elev are nume, inițiala tatălui, și alte valori ne trebuie de fapt mai multe tablouri. Dar ce facem cu notele (unde am deja 4 tablouri), și cu absențe? Ne trebuie câte un tablou bidimensional (matrice): pe prima linie am notele pentru primul elev, pe a 2-a linie notele pentru al 2-lea elev, etc.
  - Deci, pentru a reprezenta o clasă ne trebuie: un string (numele dirigintelui), 8 tablouri (numele și prenumele elevilor, inițiala tatălui, data nașterii, CNP, adresă, numărul notelor, numărul absențelor) și 7 matrici (4 pentru note - nota, disciplină, data, teză - și 3 pentru absențe - data, disciplina, motivat -).
- Și ne-ar mai trebui să reprezentăm școala, care e o listă de clase...

Gestionarea elevilor

- Este o reprezentare care e complicată, e ușor să greșim ceva, dar greu de găsit greșeala, codul necesar (de exemplu pentru a determina pentru fiecare clasă cine are media generală cea mai mare) este greu de scris și de înțeles.

Gestionarea elevilor

- Este o reprezentare care e complicată, e ușor să greșim ceva, dar greu de găsit greșeala, codul necesar (de exemplu pentru a determina pentru fiecare clasă cine are media generală cea mai mare) este greu de scris și de înțeles.
- În asemenea situații, pentru a reprezenta date din lumea reală, ne pot ajuta foarte mult tipurile de date definite de utilizator. Folosind tipuri definite de utilizator putem structura informațiile într-un fel în care înțelegem mai bine structura lor și algoritmii care le prelucrează.

Gestionarea elevilor

- Este o reprezentare care e complicată, e ușor să greșim ceva, dar greu de găsit greșeala, codul necesar (de exemplu pentru a determina pentru fiecare clasă cine are media generală cea mai mare) este greu de scris și de înțeles.
- În asemenea situații, pentru a reprezenta date din lumea reală, ne pot ajuta foarte mult tipurile de date definite de utilizator. Folosind tipuri definite de utilizator putem structura informațiile într-un fel în care înțelegem mai bine structura lor și algoritmii care le prelucrează.
- Un tip de date definit de utilizator (numit *record* sau *struct*) este un tip nou de date alcătuit din câmpuri (adică variabile). Câmpurile unui tip de date pot fi de tipuri predefinite (întreg, real, boolean, string) sau pot fi alte tipuri de date definite de utilizator.

Tipuri definite de utilizator

- Pentru a defini un tip nou trebuie să-i dăm un nume și să enumerăm câmpurile tipului. Pentru fiecare câmp trebuie să specificăm numele câmpului și tipul.

Tipuri definite de utilizator

- Pentru a defini un tip nou trebuie să-i dăm un nume și să enumerăm câmpurile tipului. Pentru fiecare câmp trebuie să specificăm numele câmpului și tipul.
- În pseudocod un tip nou este definit în modul următor:

```
NumeTipNou:
  nume_câmp1: tip
  nume_câmp2: tip
  nume_câmp3: tip
etc.
```

- După ce definim un tip nou, putem defini variabile care să aibă tipul respectiv. Pentru a accesa un câmp pentru o variabilă de un tip definit de utilizator, folosim punctul și numele câmpului.

Tipur definite de utilizator II

- Pentru un exemplu concret, presupunem că avem nevoie de un tip care să reprezinte un număr rațional (o fracție p/q). Pentru o fracție avem nevoie de 2 valori întregi, numărător și numitor.

Tipur definite de utilizator II

- Pentru un exemplu concret, presupunem că avem nevoie de un tip care să reprezinte un număr rațional (o fracție p/q). Pentru o fracție avem nevoie de 2 valori întregi, numărător și numitor.

Rațional:
  p: întreg
  q: întreg

numar: Rațional
numar.p = 2
numar.q = 3
//numar reprezintă fracția 2/3

Gestionarea elevilor I

- Dacă ne întoarcem la problema cu gestionarea elevilor, putem să reprezentăm datele definind tipuri noi de date:

Nota:
valoare_nota: întreg
disciplina: string
data: string
teza: boolean

Gestionarea elevilor II

Absență:
- data: string
- disciplina: string
- motivat: boolean

Gestionarea elevilor III

Elev:

nume: string
prenume: string
initTată: string
dataNașterii: string
CNP: string
adresa: string
note: Nota[] *//tablou cu notele*
nrNote: întreg *//lungimea tabloului cu note*
absente: Absență[] *//tablou cu absențe*
nrAbs: întreg *//lungimea tabloului cu absențe*

Gestionarea elevilor IV

Clasă:
diriginte: string
elevi: Elev[] *//tablou cu elevi*
nrElevi: întreg *//numărul elevilor*

nouaA: Clasă
numărul de absențe pentru primul elev din clasa nouaA este:
nouaA.elevi[0].nrAbs
și prima nota a celui de al 3-lea elev a fost la disciplina:
nouaA.elevi[2].note[0].disciplina
//presupunem că există al 3-lea elev și are măcar o notă

Șosete I

- Să considerăm următoarea problemă: *Un vânzător de șosete găsește o pungă cu tot felul de șosete, dar care nu sunt aranjate în perechi. Fiecare șosetă are o culoare (de exemplu: negru, marou, gri, etc.) și un model (de exemplu: buline, carouri, dungi, etc.). Vânzătorul vrea să vândă șosetele, dar știe că poate să le vândă doar în perechi (care au aceeași culoare și model). Citind de la tastatură culoarea și modelul pentru n șosete să calculăm câte perechi pot fi vândute.*

Șosete II

- De exemplu dacă avem următoarele șosete:
  - negru-buline,
  - alb-dungi,
  - negru-dungi,
  - negru-buline,
  - gri-carouri,
  - alb-dungi,
  - alb-dungi,
  - gri-buline

Șosete III

- vânzătorul poate să vândă doar 2 perechi: o pereche negru-buline și o pereche alb-dungi.

Șosete IV

- Problema nu este complicată, trebuie să vedem câte șosete avem din fiecare tip, și din acest număr câte perechi de șosete putem face. Dar cum reprezentăm datele?

Șosete V

- Putem defini un tip nou, care să reprezinte o șosetă și numărul de apariții pentru șoseta respectivă.

Șosetă:
culoare: string
model: string
cantitate: întreg

Șosete VI

- Când se citesc datele de intrare, vom construi un tablou cu șosete. În acest tablou fiecare tip de șosetă apare o singură dată. De exemplu:
- negru-buline, alb-dungi, negru-dungi, negru-buline, gri-carouri, alb-dungi, alb-dungi, gri-buline
- va fi reprezentat ca: [negru-buline-2, alb-dungi-3, negru-dungi-1, gri-carouri-1, gri-buline-1] (elementele legate cu liniuță reprezintă valorile câmpurilor).

Șosete VII

- Câte elemente ar trebui să aibă tabloul în care păstrăm șosetele?

Șosete VIII

- Știm că vor fi citite n șosete, dar nu știm de câte tipuri vor fi ele (poate toate sunt la fel și ne-ar ajunge un tablou cu 1 element, poate toate sunt diferite și ne-ar trebui un tablou cu n elemente). Pentru a fi siguri că avem un tablou suficient de mare, vom crea un tablou cu n elemente. Dar va trebui să reținem exact câte elemente sunt în tablou într-o variabilă separată.
  - A decide *capacitatea* unui tablou (adică numărul maxim de elemente care pot fi stocate în el), nu este o problemă simplă. În general, când nu știm câte elemente vor fi stocate, alegem o capacitate suficient de mare încât să încape toate elementele. Dar pentru că numărul *real* de elemente e probabil mai mic decât capacitatea, vom reține separat și numărul real de elemente.

Șosete IX

Șosete

algoritm Șosete **este**

Șosete IX

Șosete

algoritm Șosete **este**
n, nrCurent, nr: întreg
cul, mod: string
scrie "Numărul de șosete:"
citește n
sosete: Șosetă[n] *//tablou pentru max n tipuri de șosete*
nrCurent = 0 *//câte tipuri de șosete sunt momentan*

Șosete IX

Șosete

algoritm Șosete **este**
    n, nrCurent, nr: întreg
    cul, mod: string
    scrie "Numărul de șosete:"
    citește n
    sosete: Șosetă[n] *//tablou pentru max n tipuri de șosete*
    nrCurent = 0 *//câte tipuri de șosete sunt momentan*
    //începem să citim șosetele
    pentru nr = 0, n, 1 **execută**
        scrie "Dați culoarea și modelul:"
        citește cul
        citește mod
//continuăm pe pagina următoare

Șosete X

//trebuie să vedem dacă avem deja acest tip în tablou
găsit: boolean
găsit = fals *//dacă am găsit în tablou acest tip*

Șosete X

```
//trebuie să vedem dacă avem deja acest tip în tablou
găsit: boolean
găsit = fals //dacă am găsit în tablou acest tip
i: întreg
i = 0
câttimp găsit == fals ȘI i < nrCurent execută
    dacă sosete[i].culoare == cul ȘI sosete[i].model == mod atunci
```

Șosete X

```
//trebuie să vedem dacă avem deja acest tip în tablou
găsit: boolean
găsit = fals //dacă am găsit în tablou acest tip
i: întreg
i = 0
cât timp găsit == fals ȘI i < nrCurent execută
    dacă sosete[i].culoare == cul ȘI sosete[i].model == mod atunci
        //am gasit în tablou acest tip. Trebuie să incrementăm cantitatea...
        sosete[i].cantitate = sosete[i].cantitate + 1
        //și să setăm variabila găsit la adevărat
        găsit = adevărat
    sf_dacă
    //incrementăm i-ul să trecem la poziția următoare
    i = i + 1
sf_cât timp
```

Șosete XI

//daca nu am găsit, trebuie să adaug un element nou în tablou cu cantitate 1
dacă gasit == fals **atunci**
element: Șosete
element.culoare = cul
element.model = mod
element.cantitate = 1

Șosete XI

```
    //daca nu am găsit, trebuie să adaug un element nou în tablou cu
cantitate 1
    dacă gasit == fals atunci
        element: Șosete
        element.culoare = cul
        element.model = mod
        element.cantitate = 1
        //elementul nou trebuie adăugat pe poziția nrCurent
        sosete[nrCurent] = element
         //am adăgat un element nou, trebuie să incrementăm nrCurent
        nrCurent = nrCurent + 1
    sf_dacă
  sf_pentru
  //continuăm pe pagina următoare...
```

```
   //am terminat de citit șosete și am construit tabloul.
   //Acum numărăm perechi
   perechi: întreg
   perechi = 0
   pentru i = 0, nrCurent, 1 execută
      perechi = perechi + (sosete[i].cantitate / 2)
      //împărțire întreagă
   sf_pentru
   scrie "Numărul de perechi: " + perechi
sf_algoritm
```

Subprograme I

- Cum crește complexitatea problemelor de rezolvat, crește și lungimea codului de scris. Pentru a evita bucăți foarte mari și lungi de cod, vom împărți soluția noastră în bucăți mai mici, numite *subprograme*.
- Un subprogram este o bucată de cod care rezolvă o parte din problema principală. Avantajele subprogramelor (și a împărțirii codului în subprograme):
  - Codul este mai ușor de înțeles (mai ales dacă folosim nume sugestive pentru subprograme).
  - Putem refolosi codul scris (dacă avem un subprogram scris, îl putem folosi de mai multe ori în programul nostru, dar și în alte programe).

Subprograme II

- În pseudocod avem 2 tipuri de subprograme: *proceduri* și *funcții*.
- O *procedură* este un subprogram care efectuează anumite instrucțiuni, dar nu returnează niciun rezultat. De exemplu un subprogram care afișează elementele unui tablou pe ecran.
- O *funcție* este un subprogram care efectuează anumite instrucțiuni și returnează un rezultat. De exemplu, un subprogram care calculează suma elementelor dintr-un tablou.

Subprograme III

- Pentru a transmite unui subprogram datele de care are nevoie, folosim *parametri*. De exemplu, dacă avem o procedură care afișează elementele unui tablou pe ecran, vom transmite acest tablou ca parametru la procedura noastră.
- De aceea, la definirea unui subprogram, după numele subprogramului avem lista de parametri: enumerăm pe rând toți parametrii care trebuie transmiși subprogramului în momentul apelului. În pseudocod lista parametrilor conține numele și tipul parametrilor.
- Parametrii pot fi folosiți într-un subprogram ca orice variabilă definită în subprogram.

Proceduri I

- În pseudocod o procedură se definește în modul următor:

subalgoritm numeProcedura (lista parametri formali) **este**
    instrucțiuni
sf_subalgoritm

Proceduri II

- Faptul că definim o procedură (sau o funcție) nu înseamnă că instrucțiunile din procedura respectivă vor fi executate.
- Pentru a executa o procedură, trebuie să o *apelăm*.
- Apelul unei proceduri se face în modul următor:

algoritm nume **este**
    instrucțiuni ...
    numeProcedura(lista parametri actuali)
    instrucțiuni ...
sf_algoritm

Proceduri III

- *Lista parametri formali* reprezintă parametrii care trebuie transmiși procedurii, dar care pot avea orice nume în definiția procedurii, numele lor nu trebuie neapărat să coincidă cu numele variabilelor din algoritmul principal.
- În momentul apelului trebuie să specificăm parametrii actuali, adică acele variabile care trebuie să fie transmiși procedurii. Trebuie să folosim numele exact al variabilelor care vor fi transmise.

Proceduri IV

AfisareElementeTablou

```
subalgoritm AfisElemTablou (tablou: întreg[], lungime:întreg)
este
    i:întreg
    pentru i = 0, lungime, 1 execută
        scrie tablou[i]
    sf_pentru
sf_subalgoritm
```

Proceduri V

ProgramPrincipal

```
algoritm TotFelulDeTablouri este
    numerePare: întreg[10]
    nrPare, nrImpare, i: întreg
    nrPare = 10
    pentru i = 0, 10, 1 execută
        numerePare[i] = i * 2
    sf_pentru
    numereImpare: întreg[15]
    nrImpare = 15
    pentru i = 0, 15, 1 execută
        numereImpare[i] = i * 2 - 1
    sf_pentru
    AfisElemTablou(numerePare, nrPare)
    AfisElemTablou(numereImpare, nrImpare)
sf_algoritm
```

Proceduri VI

- Parametrii formali pentru procedura *AfisElemTablou* sunt *tablou* și *lungime*
- La apelul *AfisElemTablou(numerePare, nrPare)* parametri actuali sunt *numerePare* și *nrPare*. În momentul apelului, parametrul *tablou* din procedură primește ca valoare tabloul *numerePare*, iar parametrul *lungime* primește ca valoare numărul *nrPare*.
- În mod similar se întâmplă și la apelul *AfisElemTablou(numereImpare, nrImpare)*.

Funcții I

- În pseudocod o funcție se definește în modul următor:

funcție nume (lista parametri formali) **este**
    instrucțiuni
    //pentru a returna rezultatul, folosim cuvântul returni
    returnează rezultatul_funcției
sf_funcție

Funcții II

- Iar apelul unei funcții se face în modul următor:

algoritm nume **este**
    instrucțiuni ...
    rez = numeFuncție(lista parametri actuali)
    instrucțiuni ...
sf_algoritm

Funcții III

SumaElementeTablou

```
funcție SumaElemente (tablou:întreg[], lungime:întreg) este
    suma,i: întreg
    suma = 0
    pentru i = 0, lungime, 1 execută
        suma = suma + tablou[i]
    sf_pentru
    returnează suma
sf_funcție
```

Funcții VI

```
algoritm TotFeluIDeTablouri este
    numerePare: întreg[10]
    nrPare, nrImpare, i: întreg
    nrPare = 10
    pentru i = 0, 10, 1 execută
        numerePare[i] = i * 2
    sf_pentru
    numereImpare: întreg[15]
    nrImpare = 15
    pentru i = 0, 15, 1 execută
        numereImpare[i] = i * 2 - 1
    sf_pentru
    sumaPare = SumaElemente(numerePare, nrPare)
    scrie "Suma elementelor din numerePare: " + sumaPare
    sumaImpare = SumaElemente(numereImpare, nrImpare)
    scrie "Suma elementelor din numereImpare: " + sumaImpare
sf_algoritm
```

Transmiterea parametrilor I

- Să considerăm codul următor:

```
funcție Dublează(nr: întreg) este:
    nr = nr * 2
    returneaza nr
sf_funcție
```

```
algoritm ProgramPrincipal este
    valoare, rezultat: întreg
    valoare = 10
    rezultat = Dublează(valoare)
    scrie "Rezultatul apelului: " + rezultat
    scrie "Valoarea variabilei: " + valoare
sf_algoritm
```

- Ce va fi afișat, dacă apelăm algoritmul *ProgramPrincipal*?

Transmiterea parametrilor II

- Când un parametru este transmis **prin valoare** unui subprogram, modificările făcute de subprogram asupra valorii parametrului, **nu sunt vizibile în programul de unde subprogramul a fost apelat**.
- Practic, în momentul în care subprogramul este apelat, se face o copie a parametrului transmis prin valoare, și orice modificare făcută de subprogram se face pe această copie, de aceea modificările nu se văd în programul de unde se apelează subprogramul.
- Dacă folosim transmitere prin valoare, algoritmul *ProgramPrincipal* va afișa valorile 20 și 10.

Transmiterea parametrilor III

- Când un parametru este transmis **prin referință** unui subprogram, modificările făcute de subprogram asupra valorii parametrului, **sunt vizibile în programul de unde subprogramul a fost apelat**.
- Practic, în acest caz, subprogramul primește adresa de memorie a variabilei respective, și orice modificare este făcută direct la aceea adresă de memorie.
- Dacă folosim transmitere prin referință, algoritmul *ProgramPrincipal* va afișa valorile 20 și 20.

Transmiterea parametrilor IV

- Anumite limbaje de programare permit alegerea modului de transmitere a parametrilor, în alte limbaje de programare modul de transmitere a parametrilor este predefinită.
- **În pseudocod vom presupune că tablourile și structurile definite de utilizator sunt transmise prin referință iar variabilele de tip număr (întreg sau real), caracter, string sau boolean sunt transmise prin valoare.**

Sume egale

- Să considerăm umătoarea problemă: *Să se citească un tablou de n elemente și să verifice dacă există în tablou o poziție i, astfel că suma elementelor în stânga poziției este egală cu suma elementelor în dreapta poziției. Dacă într-o parte nu există elemente, suma se consideră 0.*
- De exemplu pentru tabloul [10, 7, 3, 6, 8, 3]

Sume egale

- Să considerăm umătoarea problemă: *Să se citească un tablou de n elemente și să verifice dacă există în tablou o poziție i, astfel că suma elementelor în stânga poziției este egală cu suma elementelor în dreapta poziției. Dacă într-o parte nu există elemente, suma se consideră 0.*
- De exemplu pentru tabloul [10, 7, 3, 6, 8, 3]
  - pentru poziția 0, sumele sunt 0, 27
  - pentru poziția 1, sumele sunt 10, 20
  - pentru poziția 2, sumele sunt 17, 17
  - pentru poziția 3, sumele sunt 20, 11
  - pentru poziția 4, sumele sunt 26, 3
  - pentru poziția 5, sumele sunt 34, 0

Sume egale

- Pentru că trebuie să calculăm de foarte multe ori suma elementelor dintre 2 poziții, putem defini o funcție separată care să calculeze această sumă. Funcția va primi ca parametru tabloul și pozițiile de început și de sfârșit. Va returna suma elementelor între cele 2 poziții - inclusiv capetele.

Sume egale

- Pentru că trebuie să calculăm de foarte multe ori suma elementelor dintre 2 poziții, putem defini o funcție separată care să calculeze această sumă. Funcția va primi ca parametru tabloul și pozițiile de început și de sfârșit. Va returna suma elementelor între cele 2 poziții - inclusiv capetele.

SumaIntrePozitii

```
funcție SumaIntrePozitii (tablou: întreg[], start: întreg, end: întreg) este:
    suma, i: întreg
    suma = 0
    pentru i = start, end+1, 1 execută
        suma = suma + tablou[i]
    sf_pentru
    returnează suma
sf_funcție
```

Sume egale

```
algoritm SumeEgale este
    scrie "Dați numărul de numere:"
    nr, i, poz, st, dr: întreg
    citește nr
    elemente: întreg[nr]
    pentru i = 0, nr, 1 execută
        scrie "Elementul de pe pozitia " + i
        citește elemente[i]
    sf_pentru
    poz = -1 //dacă găsim o poziție bună, modificăm valoarea
    pentru i = 0, nr, 1 execută
        st = SumaIntrePozitii (elemente, 0, i-1) //suma din stânga
        dr = SumaIntrePozitii (elemente, i+1, nr-1) //suma din dreapta
        dacă st == dr atunci
            poz = i
        sf_dacă
    sf_pentru
```

```
   dacă poz == -1 atunci
      scrie "Nu există poziție"
   altfel
      scrie "Poziția: " + poz
   sf_dacă
sf_algoritm
```

Sume egale

- Temă de gândire: *Momentan recalculăm de foarte multe ori sumele. Cum s-ar putea rezolva problema calculând sumele de mai puține ori? Gândiți-vă cum se modifică cele două valori st și dr când trecem de la o poziție la alta.*

Recursivitate I

- Secvența Fibonacci este o secvență de numere, $F_1, F_2, F_3...$, definită în modul următor:
  - $F_1 = 1$
  - $F_2 = 1$
  - $F_n = F_{(n-1)} + F_{(n-2)}$
- Deci secvența este: 1, 1, 2, 3, 5, 8, 13, 21, ... etc.
- Dacă vreau să știu al 20-lea element din secvența Fibonacci, trebuie să calculez al 19-lea și al 18-lea element.
- Când valoarea unui element depinde de valori anterioare, spunem că avem o definiție *recursivă*.

Recursivitate II

- Problemele recursive pot fi rezolvate folosind *funcții recursive*.
- O funcție recursivă este o funcție care se autoapelează (o apelează pe ea însăși).
- De exemplu, pentru a determina al n-lea număr din secvența Fibonacci, putem implementa următoarea funcție recursivă:

Recursivitate III

```
funcție fibo(n: întreg) este
    dacă n == 1 SAU n == 2 atunci
        returnează 1
    altfel
        f1, f2: întreg
        f1 = fibo(n-1) //apel recursiv pentru n-1
        f2 = fibo(n-2) //apel recursiv pentru n-2
        returnează f1 + f2
    sf_dacă
sf_funcție
```

Recursivitate IV

- E important să avem cazuri de oprire în funcțiile recursive, altfel riscăm să intrăm într-un ciclu infinit.
- Cazul de oprire e un caz în care nu mai facem apel recursiv (la numărul Fibonacci e cazul când n este 1 sau 2).

Recursivitate V

Fibonacci greșit

funcție fibo(n) **este**
    f1, f2: întreg
    f1 = fibo(n-1) *//apel recursiv pentru n-1*
    f2 = fibo(n-2) *//apel recursiv pentru n-2*
    returnează f1 + f2
sf_funcție

- Pentru fibo(2) se face apel la fibo(1) și fibo(0)...pentru fibo(1) se face apel la fibo(0) și fibo(-1)...pentru fibo(0) se face apel la fibo(-1) și fibo(-2)...și apelurile niciodată nu se termină

- Temă de gândire 1: *Rescrieți funcția fibo ca să nu fie recursivă*.
- Temă de gândire 2: *Calculați în mod recursiv valoarea 2 la puterea k. Puteți folosi faptul că 2 la puterea k este de fapt de 2 ori 2 la puterea k - 1*.

Funcții recursive și nerecursive

- Sunt probleme unde definiția problemei este recursivă și pare *natural* să folosim o funcție recursivă (de ex. numerele Fibonacci)
- Câteodată și aceste probleme pot fi implementate folosind funcții nerecursive
- Și de multe ori, probleme care nu par recursive, pot fi rezolvate cu o funcție recursivă (dacă am un ciclu, probabil îl pot înlocui cu apeluri recursive).

Verificare număr prim

- Să considerăm problema următoare: *să determinăm dacă un număr este prim sau nu*.

Verificare număr prim

- Să considerăm problema următoare: *să determinăm dacă un număr este prim sau nu*.
- Un număr este prim, dacă este divizibil doar cu 1 și el însuși. De exemplu, 7, 13, 19 sunt numere prime, 15, 21, 25 nu sunt. Prin definiție,1 nu este număr prim.
- Pentru a verifica dacă un număr este prim, trebuie să vedem dacă este divizibil cu alt număr în afară de 1 și el însuși.
- Un număr *n* are divizori în intervalul [1, n]*, deci trebuie să verificăm aceste valori.
- * se pot face anumite optimizări să reducem numărul de verificări

Verificare număr prim - nerecursiv

```
funcție verifPrim(n: întreg) este
    dacă n == 1 atunci
        returnează fals
    altfel
        div: întreg
        div = 2 //primul divizor posibil
        ok: boolean
        ok = adevărat //presupunem că numărul e prim
        câttimp div < n ȘI ok execută
            dacă n mod div == 0 atunci
                ok = fals
            sf_dacă
            div = div + 1
        sf_câttimp
        returnează ok
    sf_dacă
sf_funcție
```

Verificare număr prim recursiv

- Cum putem implementa verificarea în mod recursiv?

Verificare număr prim recursiv

- Cum putem implementa verificarea în mod recursiv?
- Putem face o funcție cu 2 parametri: numărul de verificat și un divizor potențial. În funcție verificăm dacă divizorul potențial este chiar divizor. Dacă da, ne oprim (numărul are un divizor, nu poate fi prim). Dacă nu e divizor, mergem mai departe, verificăm divizorul următor într-un apel recursiv.
- Când putem spune că numărul e prim?

Verificare număr prim recursiv I

```
funcție verifPrimR (n: întreg, div: întreg) este
    dacă div == n atunci
        returnează adevărat
    altfel dacă n mod div == 0 atunci
        returnează fals
    altfel
        returnează verifPrimR(n, div+1)
    sf_dacă
sf_funcție
```

Verificare număr prim recursiv II

- Cum apelăm funcția *verifPrimR*?

Verificare număr prim recursiv III

funcție verifPrim (n: întreg) **este**
    dacă n $\leq$ 1 **atunci**
        returnează fals
    altfel
        returnează verifPrimR(n, 2)
    sf_dacă
sf_funcție