

DIDACTICA INFORMATICII

2019-2020

Problema 54

nume student
Grupa
adresa email

Enuntul problemei:

54. Un număr natural se numește perfect dacă este egal cu suma divizorilor săi, fără el însuși.

Exemplu: $6=1+2+3$.

Să se verifice dacă un număr natural dat este perfect.

Breviar teoretic

Subprogramele sunt parti ale unui program care contin tipuri de date, variabile si instructiuni destinate unei anumite prelucrari, identificabile prin nume, care se pot activa la cerere prin intermediul acestor nume.

Un program C++ este alcatuit din una sau mai multe functii: una principala (main), respectiv niciuna sau mai multe functii secundare(subordonate).


Amintim cum se defineste o functie:

```
tip_returnat_de_functie nume_functie (lista_parametrilor_formali) // antetul  
{ instructiune; // corpul functiei  
}
```

Amintim cum se apeleaza o functie:

 prin valoare:

```
tip_returnat nume_variabila = nume_functie(lista_parametrilor_efectivi);
```

 prin referinta:

```
nume_functie(lista_parametrilor_efectivi);
```

Amintim cum se face revenirea in programul principal(main) din functie:

- dupa executia ultimei instructiuni din corpul functiei
- la intalnirea unei instructiuni **return** : return; sau return expresie;

Cunostinte matematice minime necesare rezolvarii problemei:

 Notiunea de **divizor**: Un numar natural a este divizibil cu un numar natural b daca exista un numar natural c astfel incat $a=b*c$.

Notam: $a \bmod b = 0$ (restul impartirii lui a la b este 0) si $a/b = c$ (catul impartirii lui a la b este c).

✚ Notiunea de **multimea divizorilor unui numar a**: Multimea divizorilor lui a este formată din toți divizorii ai lui a.

✚ Notiunea de **numar perfect**: Numarul perfect este un numar intreg egal cu suma divizorilor sai, din care se exclude numarul insusi

✚ Optional: notiunea de **radical**.

Cunostinte minime din punct de vedere al algoritmilor:

✚ Calculul divizorilor unui numar.

✚ Calcului sumei a n(n dat) numere.

Analiza enuntului problemei:

Problema ne cere sa testam daca un numar natural dat (citit de la tastatura) este numar perfect sau nu. Recitind cu atentie, putem identifica doua cuvinte cheie: suma si divizori. Astfel, vine ideea de a imparti problema in doua subprobleme independente, mai usor de rezolvat: programul principal si un subprogram care va calcula suma divizorilor numarului citit in programul principal.

Rezolvarea problemei:

1. Incercam rezolvarea problemei doar in cadrul **programului principal**, fara a ne folosi de subprograme: consideram intr-o structura repetitiva o variabila numita **divizor**, initializata cu 1, avand limita de oprire **numar-1**, unde **numar** reprezinta numarul citit. In acest interval, testam la fiecare pas daca **numar mod divizor == 0**. In caz afirmativ, adaugam la **suma** (initializata cu 0 la inceputul corpului functiei), altfel nu efectuam nicio atune, ci doar trecem mai departe. La final testam daca **suma == numar**. In caz afirmativ, afisam un raspuns pozitiv, altfel negativ.

```
int main(){
    int numar, divizor, suma; // declaram variabilele
    suma = 0; // initializam cu 0 suma
    cin>>numar; // citim numarul
    for(divizor = 2; divizor < n; divizor++) // parcurgem numarul
    {
        if(numar % divizor == 0) // daca numar se imparte exact la divizor
        {
            suma = suma + divizor; // adaugam la suma
        }
    }
    if(suma == numar) // verificam egalitatea dintre numar si suma
    {
        cout<< numar<<" este numar perfect."; // in caz pozitiv
    }
    else
    {
        cout<<numar<<" nu este numar perfect."; // in caz negativ
    }
}
```

```

    return 0; // incheiem programul principal
}

```

2. Modularizam programul prezentat anterior pentru a trece in revista utilizarea **subprogramelor cu transfer prin valoare**: subprogramul va primi un singur parametru, numarul citit in main; consideram intr-o structura repetitiva o variabila numita **divizor**, initializata cu 1, avand limita de oprire **numar-1**, unde **numar** reprezinta numarul citit. In acest interval, testam la fiecare pas daca **numar mod divizor == 0**. In caz afirmativ, adaugam la **suma** (initializata cu 0 la inceputul corpului functiei), altfel nu efectuam nicio atiune, ci doar trecem mai departe. La final, returnam in programul principal **suma** calculata de subprogram.

!! Obs. : Putem considera tipul_returnat, respectiv tipul parametrului sa fie **unsigned**?
(R.: Da. Enuntul specifica numere **naturale**.)

```

int suma_divizorilor(int numar)
{
    int divizor, suma; // declaram variabilele
    suma = 0; // initializam suma cu 0
    for( divizor = 1; divizor < numar; divizor++) // parcurgem: 1 la numar-1

    {
        if(numar % divizor == 0) // daca numar se imparte exact la divizor
        {
            suma = suma + divizor; // adaugam la suma
        }
    }
    return suma; // dupa executarea repetitivei, returnam suma gasita
}

int main(){
    int numar, suma; // declaram variabilele
    cin>>numar; // citim numarul
    suma = suma_divizorilor(numar); // apelam functia si pastram in suma
    if(suma == numar) // verificam egalitatea dintre numar si suma calculata
    {
        cout<< numar<<" este numar perfect."; // in caz pozitiv
    }
    else
    {
        cout<<numar<<" nu este numar perfect."; // in caz negativ
    }
    return 0; // incheiem programul principal
}

```

3. Modularizam programul prezentat la pasul 1. pentru a trece in revista utilizarea **subprogramelor cu transfer prin referinta**: subprogramul va primi doi parametri: numarul citit si o variabila prin care furnizam suma divizorilor; consideram intr-o structura repetitiva o variabila numita **divizor**, initializata cu 1, avand limita de oprire **numar-1**, unde **numar** reprezinta numarul citit. In acest interval, testam la fiecare pas daca **numar mod divizor == 0**. In caz afirmativ, adaugam la **suma** (initializata cu 0 la inceputul corpului functiei), altfel nu


efectuam nicio atiune, ci doar trecem mai departe. La final, returnam in programul principal **suma** calculata de subprogram.

```
int suma_divizorilor(int numar, int &suma) // antetul functiei
{
    int divizor, suma; // declaram variabilele
    suma = 0; // initializam suma cu 0
    for( divizor = 1; divizor < numar; divizor++) // parcurgem de la 1 la
numar-1
    {
        if(numar % divizor == 0) // daca numar se imparte exact la divizor
        {
            suma = suma + divizor; // adaugam la suma
        }
    }
}

int main(){
    int numar, suma; // declaram variabilele
    cin>>numar; // citim numarul
    suma_divizorilor(numar,suma); // apelam functia cu numarul si suma
    if(suma == numar) // verificam egalitatea dintre numar si suma
    {
        cout<< numar<<" este numar perfect."; // in caz pozitiv
    }
    else
    {
        cout<<numar<<" nu este numar perfect."; // in caz negativ
    }
    return 0; // incheiem programul principal
}
```

!! Obs. : Mai este necesar sa initializam **suma** cu 0 in main, daca o initializam in subprogram?
(R.: Nu, deoarece transferul se face prin referinta.)

4. Modularizam programul prezentat la pasul 1. pentru a trece in revista utilizarea **subprogramelor cu transfer prin valoare cu tip_returnat logic**: subprogramul va primi un singur parametru, numarul citit in main; consideram intr-o structura repetitiva o variabila numita **divizor**, initializata cu 1, avand limita de oprire **numar-1**, unde **numar** reprezinta numarul citit. In acest interval, testam la fiecare pas daca **numar mod divizor == 0**. In caz afirmativ, adaugam la **suma** (initializata cu 0 la inceputul corpului functiei), altfel nu efectuam nicio atiune, ci doar trecem mai departe. La final, testam in cadrul subprogramului daca numarul este sau nu perfect si returnam in programul principal raspunsul.

 **Obs.:** Acelasi lucru se poate face si cu subprograme cu transfer prin referinta. Lasam elevului ca exercitiu sa rezolve problema in aceasta maniera.

```
bool este_perfect(int numar) // antetul functiei
{
    int divizor, suma; // declaram variabilele
    suma = 0; // initializam suma cu 0
    for( divizor = 1; divizor < numar; divizor++) // de la 1 la numar-1
```

```

{
    if(numar % divizor == 0) // daca numar se divide cu divizor
    {
        suma = suma + divizor; // adaugam la suma
    }
}
if(suma == numar)
    {return true;}
else
    {return false;}
}

int main(){
    int numar, suma; // declaram variabilele
    bool verifica; // variabila de tip boolean in care pastram
//rezultatul returnat de functie
    cin>>numar; // citim numarul
    verifica = este_perfect(numar); // apelam functia
    if(verifica == true) // testam
    {
        cout<< numar<<" este numar perfect."; // in caz pozitiv
    }
    else
    {
        cout<<numar<<" nu este numar perfect."; // in caz negativ
    }


    return 0; // incheiem programul principal
}

```

Concluzii

Problema nu prezinta dificultate sporita in niciunul dintre modurile alese pentru a fi rezolvata, insa diferente se pot observa. In cazul primei variante, programul principal este incarcat, dar usor de urmarit. Pe de alta parte, in cazul utilizarii subprogrameelor observam un numar de avantaje: modularizarea problemei (posibilitatea unei analize logice facile, descompunere in subprobleme, obisnuirea elevului sa lucreze curat si modularizat), reutilizarea acelorasi secvente de mai multe ori (la nevoie, de exemplu prin dezvoltarea dificultatii problemei: citirea mai multor numere in cadrul aceluiasi program), depanarea si intretinerea mai usoara (in cazul aparitiei erorilor de programare/sintaxa). De asemenea, rezolvarea in mai multe moduri, principal diferite sporesc capacitatea de analiza a elevului, gandirea logica si analitica si abilitatile de problem solving (scop: gasirea mai multor solutii pentru o problema).

Din punct de vedere al complexitatii, al eficientei, intre variantele prezentate mai sus nu poate fi vorba de o diferenta remarcabila, insa propunem urmatorul program spre a fi o **varianta eficienta** (in special, in cazul numerelor mari, mai ales de tip **long long**):

 **divizor** va lua valori doar intre 1 si **numar/2**, inclusiv; prin exemple elevul va observa singur ca numerele parcurse de la **numar/2** in sus nu vor aparea niciodata in **suma**; (exemplificam pe una dintre variantele mai sus prezentate)

```
int suma_divizorilor(int numar) // antetul functiei
{
    int divizor, suma; // declaram variabilele
    suma = 0; // initializam suma cu 0
    for( divizor = 1; divizor <= numar/2; divizor++) // :1 la numar/2

    {
        if(numar % divizor == 0) // daca numar se divide cu divizor
        {
            suma = suma + divizor; // adaugam la suma
        }
    }
    return suma; // dupa executarea repetitivei, returnam suma gasita
}
... (main)
```

 In cazul in care elevii au cunostintele necesare apare intrebarea-observatie:

Putem eficientiza programul mai mult prin limitarea lui **divizor** la **sqrt(n)**?

(R.: Nu. Aceasta limitare functioneaza doar in cazul descompunerii unui numar in factori primi (ex: 8))

Barem

Din oficiu	1p
Cunoștințe generale necesare	1p
Modularizare	2p
Subprogramul pt divizori	3p
Programul principal	1p
Complexitate	1p
Stil (comentarii, indentare)	1p