Curs Pregatire Admitere UBB 2023

Cursul 4 – Fundamente de Programare

Fundamente de Programare

- In acest curs vom discuta despre limitele tipurilor de date in C si C++ (necesare unei bune calculari a dimensiunilor variabilelor care incap pe un anumit numar de biti)
- De asemenea, vom aborda si tema calcularii complexitatii unui algoritm si vom vedea cum identificam corect algoritmul in functie de dimensiunile datelor de intrare

Complexitatea Algoritmilor

- Toți algoritmii informatici sunt caracterizați de o complexitate. Aceasta poate fi de 2 tipuri: Complexitate TIMP si Complexitate SPATIU.
- Este foarte important sa cunoaștem complexitățile algoritmilor noștri deoarece in acest mod putem estima timpul in care algoritmul va duce la bun sfârșit "Task-ul" si de asemenea putem estima spațiul(memoria) de care are nevoie programul pentru a putea finaliza sarcina.
- Complexitatea unui program este in general estimata de programator încă din faza de proiectare si acest lucru este foarte important pentru a afla limitele in care programul funcționează.

Complexitatea algoritmilor

- Pentru a rezolva corect o problema de informatica, este extrem de important ca complexitatea algoritmilor pe care ii proiectam sa se încadreze in limitele impuse de problema.
- "Time limit exceeded" este o eroare des întâlnita care semnifica faptul ca complexitatea timp a algoritmului proiectat de noi nu este suficient de buna, motiv pentru care trebuie sa proiectam un nou algoritm mai bun din acest punct de vedere.
- "Caught fatal signal 11" este in general eroarea semnalata de evaluatorul de probleme in momentul in care complexitatea spațiu este prea mare, adică intenționam sa folosim mai multa memorie decât ne este alocata.

Complexitatea Timp

- Complexitatea timp se exprima in general sub forma O(x) unde x reprezintă numărul total de operații pe care programul nostru le va face. "X" nu este exprimat niciodată ca valoare absoluta ci mai mult ca si o suma de alte variabile care reprezintă datele de intrare ale programului. Exemplu: O(n + m) reprezintă faptul ca programul nostru va face n + m pași unde n si m sunt date de intrare.
- In cazul in care ne confruntam cu o suma de mai multe astfel de variabile, pentru a nu lungi foarte mult formula de calcul a complexității, se considera o notație corecta selectarea celei mai mari valori. Exemplu: O(m + n^2) va fi considerate O(n ^ 2) daca n^2 > m. De asemenea si O(n + m) poate fi considerata O(n) daca n > m sau invers in caz contrar.

Complexitatea Timp

- In calculul complexitatilor, constantele nu se iau in considerare. Exemplu: O(2 * n) este egal cu O(n)
- Tipuri de complexitati timp si algoritmii aferenti:
  - O(log2(n)) – se numeste complexitatea logaritmica. Un exemplu in acest sens este algoritmul de cautare binara
  - O(sqrt(n)) – se numeste complexitate radical. Un exemplu in acest sens este algoritmul de determinarea divizorilor unui numar
  - O(n) – se numeste complexitate liniara. Un exemplu in acest sens este algoritmul de cautare secventiala pe vector
  - O(n^2) – se numeste complexitate patratica. Un exemplu in acest sens este algoritmul de sortare prin insertie / bubble sort
  - O(n^n) – se numeste complexitate exponentiala. Un exemplu in acest sens este algoritmul backtracking care genereaza toate multimile de n
  - Altele: O(n * log2(n)), O(n!) etc.

Complexitatea Spatiu

- Complexitatea spatiu se exprima in general identic cu cea timp sub forma O(x) unde x reprezintă numărul total de variabile pe care programul nostru le va stoca. "X" nu este exprimat niciodată ca valoare absoluta ci mai mult ca si o suma de alte variabile care reprezintă dimensiunile datelor de intrare ale programului. Exemplu: O(n + m) reprezintă faptul ca programul nostru va retine n + m variabile unde n si m sunt dimensiunile datelor de intrare. (In general acest exemplu semnifica retinarea a 2 vectori de n si, respectiv, m elemente)
- In cazul in care ne confruntam cu o suma de mai multe astfel de variabile, pentru a nu lungi foarte mult formula de calcul a complexității, se considera o notație corecta selectarea celei mai mari valori. Exemplu: O(m + n^2) va fi considerate O(n ^ 2) daca n^2 > m. De asemenea si O(n + m) poate fi considerata O(n) daca n > m sau invers in caz contrar.

Complexitatea Spatiu

- In calculul complexitatilor, constantele nu se iau in considerare. Exemplu: O(2 * n) este egal cu O(n)
- Tipuri de complexitati timp si algoritmii aferenti:
  - O(n) – complexitate liniara – se utilizeaza doar un vector de n elemente
  - O(n * m) – complexitate patratica – se utilizeaza o matrice cu n * m elemente
  - O(1) - nu exista complexitate spatiu – se utilizeaza doar variabile simple
  - O(2 ^ n) – complexitate exponentiala – se utilizeaza pentru memorarea arborilor de intervale

Exemple de algoritmi si complexitatile lor

- Algoritmul alaturat va citi un numar natural si va afisa scrierea lui in baza 2, utilizant o functie recursiva.
- Complexitatea Spatiu a acesuia este O(1) intrucat se declara o singura variabile simpla (n)
- Complexitatea Timp este O(log2(n)) intrucat va imparti numarul la 2 pana se obtine 1, motiv pentru care aceasta este complexitatea

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

void base2(int n){
    if(n == 0)
        return;
    base2(n / 2);
    cout << n % 2;
}

int main(){
    int n;
    cin >> n;
    base2(n);
    return 0;
}
```

Exemple de algoritmi si complexitatile lor

- Algoritmul alaturat va cauta binar o valoare citita si va afisa 1 daca o gaseste si 0 in caz contrar.
- Complexitatea Spatiu a acesuia este O(n) intrucat se declara un vector de n elemente.
- Complexitatea Timp este O(log2(n)) la functie si O(n) la intregul program.

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

int a[1001], n, val;
bool cb(int a[], int n, int val){
    int st = 1, dr = n;
    while(st <= dr){
        int mij = (st + dr) / 2;
        if(a[mij] == val)
            return true;
        else if(a[mij] < val)
            st = mij + 1;
        else dr = mij - 1;
    }
    return false;
}

int main(){
    cin >> n >> val;
    for(int i = 1; i <= n; ++i)
        cin >> a[i];
    cout << cb(a, n, val);
    return 0;
}
```

Exemple de algoritmi si complexitatile lor

- Algoritmul alaturat va calcula suma elementelor dintr-o matrice patratica
- Complexitatea Spatiu a acesuia este O(n*n) intrucat se declara o matrice cu n linii si n coloane, deci n * n elemente.
- Complexitatea Timp este O(n * n) intrucat acesta este numarul de pasi pe care ii va parcurge algoritmul nostru pentru determinarea numarului cerut.

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

int n, a[1001][1001];
int sum_elem_mat(int n, int a[][1001]){
    int sum = 0;
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= n; ++j)
            sum += a[i][j];
    return sum;
}

int main(){
    cin >> n;
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= n; ++j)
            cin >> a[i][j];
    cout << sum_elem_mat(n, a);
    return 0;
}
```

Cum estimam timpul de executie?

- Procesoarele moderne au o capacitate teoretica de 1 MLD de operatii pe secunda. In majoritatea cazurilor ne intereseaza sa facem algoritmi incadrabili in 0.1 secunde, deci teoretic vorbind procesorul ar putea face maximum 100 mil de pasi in 0.1 secunde. Ce inseamna o operatie?
- O operatie semnifica cea mai simpla modificare produsa in memorie. Ex: i++, if(…), orice expresie care trebuie evaluta etc.
- Astfel, e clar ca daca avem un for de 1.000.000 de pasi, el va face efectiv undeva la 10.000.000 de operatii. Din acest punct de vedere, dupa ce am calculate complexitatea unui algoritm, urmatorul pas este sa inlocuim variabilele din expresia complexitatii cu maximul valorilor posibile pentru fiecare si sa vedem cat e rezultatul. DACA REZULTATUL ACESTEI EXPRESII DEPASESTE 3-5 MIL, ALGORITMUL E PREA LENT.

Care e complexitatea potrivita?

- Va spuneam ca un bun programator trebuie sa estimeze din faza de proiectarea care va fi complexitatea algoritmului lui. Astfel, e foarte simplu sa faca calculele inainte sa se apuce de treaba.
- Date pana in 1000-2000?! Ne putem gandii la o complexitate patratica.
- Date mai mari?! Incercam linear, daca nu e posibil, incercam n * log2(n). Daca nici asa nu megre, trebuie sa mearga n*sqrt(n).
- Cam asta e rationamentul.

Complexitatea memorie.

- Pentru a calcula correct complexitatea memorie, vom proceda similar, dar lucrurile nu sunt la fel de simple intrucat fiecare tip de date ocupa un anumit numar de octeti. In functie de acest numar de octeti (bytes), vom avea o complexitatea mai mica sau mai mare.
- Pentru a putea calcula corect aceste complexitati trebuie sa intelegem pe deplin cum functioneaza si cata memorie utilizeaza fiecare tip de date.
- Pentru acest lucru, in urmatoarele slide-uri vom vorbi despre limitele fiecarui tip de date si cata memorie utilizeaza fiecare.

Limitele tipurilor de date C/C++

- Desi in general sunt intalnite sub numele de C Limits, limitele limbajului C/C++ (atat C cat si C++ au aceleasi limite) se refera la limitele tipurilor de date.
- Fiecare tip de date (bool, char, int, long long) sunt scrise pe un anumit numar de octeti (grupare de cate 8 biti) si acest numar de octeti le impune niste limite.
- Datele sunt stocate in memorie ca numere in baza 2, motiv pentru care dimensiunile datelor sunt direct influentate de numarul de cifre in baza 2 din reprezentarea acestora.

Tipurile de date in C++ si memoria alocata fiecaruia

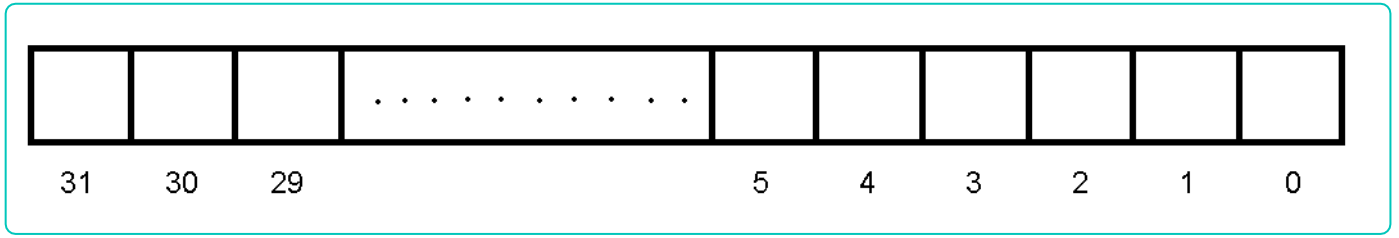
- Tipul de date “int” foloseste 4 octeti, deci 32 de biti
- Tipul de date “long long” foloseste 8 octeti, deci 64 de biti
- Tipul de date “bool” foloseste 1 octet, deci 8 biti *
- Tipul de date “char” foloseste 1 octet, deci 8 biti
- Tipul de date “short” foloseste 2 octeti, deci 16 biti
- Tipul de date “double” foloseste 8 octeti, deci 64 de biti
- Tipul de date “float” foloseste 4 octeti, deci 32 de biti

Diferenta dintre signed si unsigned

- Etichetele “signed” si “unsigned” scrise in fata variabilelor sunt asociabile tipurilor de date “int”, “long long”, “short”, “char” etc.
- Ideea este sa ii spunem compilatorului ce ne dorim de la el pentru a putea maximiza intervalul valoric al variabilelor.
- Ideea este ca atunci cand nu avem nevoie de numere negative sa declaram variabilel “unsigned” si in acest mod ni se dubleaza intervalul numeric in care sunt incadrabile tipurile de date.
- De asemenea, daca declaram variabilele “unsigned” si incercam sa memoram variabile negative, acest lucru va duce la erori de calcul.

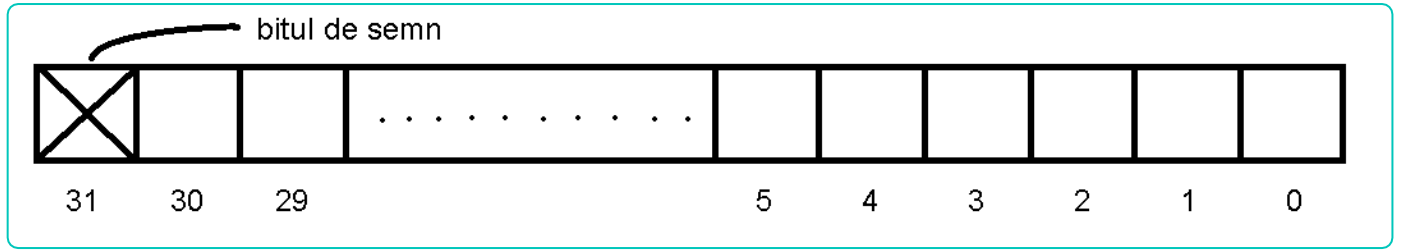
Cum se memoreaza datele?

- Tipul de date “int”:
  - - Se reprezinta pe 32 de biti, numerotati de la 0 la 31
  - - Scrierea unui numar memorat pe acest tip de date trebuie sa se incadreze in limita de 32 de biti cand este scris in baza 2
  - - Cel mai mare numar care se poate scrie pe 32 de biti este $2^{32} - 1$



Reprezentarea numerelor cu semn

- Pentru reprezentarea numerelor cu semn, se va folosi un bit dintre cei alocati scrierii numarului pentru a reprezenta semnul.
- “1” semnifica negativ si “0” semnifica pozitiv, iar avand in vedere ca am folosit un bit, vor ramane cu 1 bit mai putin pentru reprezentarea valorilor.
- Din acest motiv, vom vedea ca atunci cand reprezentam numerele “signed” valorile sunt mai mici.



Limitele tipului de date “signed int”

- Avand in vedere ca “blocam” bitul 31 al reprezentarii, ne raman 31 de biti pentru valoarea efectiva. Din acest motiv, folosind doar 31 de biti, cea mai mare valoare pozitiva care se poate retine pe acest tip de date este $2^{31} - 1$
- Din acest pucnt de vedere, in domeniul numerelor negative ar trebui sa avem cel mai mic numar egal cu - $2^{31} + 1$
- Deoarece valoarea 0 este considerata pozitiva, nu mai trebuie sa reprezentam aceasta valoare si pentru numerele negative si din acest punct de vedere, putem memora cu o valoare in plus pentru numerele negative => cea mai mica valoare care poate fi memorata “signed int” este -2^{31}
- INTERVALUL FINAL DE NUMERE PENTRU SIGNED INT ESTE $[-2^{31}, 2^{31} - 1]$

Limitele tuturor tipurilor de date

- "signed int": $[-2^{31}, 2^{31} - 1]$
- "unsigned int": $[0, 2^{32} - 1]$
- "signed long long": $[-2^{63}, 2^{63} - 1]$
- "unsigned long long": $[0, 2^{64} - 1]$
- "signed short": $[-2^{15}, 2^{15} - 1]$
- "unsigned short": $[0, 2^{16} - 1]$
- "signed char": $[-2^{7}, 2^{7} - 1]$
- "unsigned char": $[0, 2^{8} - 1]$

Cand nu specificam efectiv daca un tip de date e "signed" sau "unsigned", compilatorul le considera "signed".

Revenind la MEMORIE

- Acum, dupa ce am inteles pe deplin cati bytes ocupa fiecare tip de date, putem sa afirmam ca un vector de 1.000.000 de inturi ocupa aprox 4 MB, in timp ce 1.000.000 de long-long-uri ocupa 8 MB.
- In acest mod, la fel ca la timp, inca din faza de proiectare putem sa calculam cata memorie ocupa programul nostru si astfel, aflam daca trebuie sa facem sau nu optimizari.

EXPRESIILE LOGICE

- Un alt capitol foarte important care trebuie discutat foarte bine il reprezinta cel al expresiilor informatice.
- Aceste expresii se formeaza cu ajutorul operatorilor logici si relationali. Aceste expresii se evalueaza cu TRUE / FALSE si sunt adesea folosite in structuri de decizie, structuri repetitive sau la return-ul functiilor bool.
- Operatorii logici sunt: || (sau), && (si), ! (NOT)
- Operatorii rationali sunt: == (egal), <= (mai mic, egal), >= (mai mare, egal), < (mai mic), > (mai mare), != (neegal)

Operatorii logici

- Operatorii logici inlantuiesc o multime de expresii si evalueaza, in functie de operator tipul expresiei (TRUE / FALSE)
- Exemplu:
  - TRUE && TRUE && TRUE && FALSE = FALES
  - TRUE && TRUE && TRUE && TRUE = TRUE
  - TRUE || FALSE || FALSE || FALSE = TRUE
  - FALSE || FALSE || FALSE || FALSE = FALSE
  - ! TRUE = FALSE
  - ! FALSE = TRUE

Operatorii Relationali

- Evalueaza anumite expresii non-binare cu TRUE / FALSE pentru a putea sa folosim operatorii logici dupa aceea.
- Exemple:
  - 1 <= 2 -> TRUE
  - 1 <= 1 -> TRUE
  - 0 >= 1 -> FALSE
  - 0 < 1 -> TRUE
  - 0 == 1 -> FALSE
  - 0 != 1 -> TRUE

Exemplu de expresie

- !(x <= 10 && y <= 12) || (x >= 10 && !(y<=10))
- (x > 10 || y > 12) || (x >= 10 && y > 10) – acum e mult mai usor de evaluat

Regulile de simplificare

- !(A && B) -> !A || !B
- !(A || B) -> !A && !B
- !(A <= B) -> A > B
- !(A > B) -> A <= B
- ! (A == B) -> A != B
- A <= B && A >= B -> A == B

1. Se consideră expresia logică: `(X OR Z) AND (NOT X OR Y)`. Care din variantele de mai jos fac expresia true (adevărată)?

a. X ← false; Y ←false; Z ← true;
b. X ← true; Y ←false; Z ← false;
c. X ← false; Y ←true; Z ← false;
d. X ← true; Y ←true; Z ← true;

	X	Y	Z	X or Z	NOT X or Y	(X or Z) AND (not X or Y)
1.	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
2.	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE
3.	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE
4.	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
5.	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE
6.	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
7.	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
8.	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE

2. Știind că $x < y$ (x și y sunt numere reale), care din următoarele expresii are valoarea *adevărat* dacă și numai dacă numărul memorat în t (t număr real) NU aparține intervalului (x, y)?

A. `(t > x) SAU (t < y)`
B. `(t ≤ x) SAU (t ≥ y)`
C. `(t ≤ x) ȘI (t ≥ y)`
D. `(t > x) ȘI (t < y)`

8. Care este valoarea expresiei de mai jos, dacă $x = 15$ și $y = 17$?

```
(NU (x MOD 10 = 0)) ȘI (y MOD 2 = 0) ȘI (x < y)
```

A. *adevărat* B. *fals* C. Eroare D. Expresia nu poate fi evaluată

11. Care dintre următoarele expresii au valoarea *adevărat* dacă și numai dacă x este număr impar și negativ? Notăm cu $|x|$ valoarea absolută a lui x (modulul lui x).

A. `(|x| MOD 2 = 1) ȘI (x < 0)`
B. `NU ((|x| MOD 2 = 0) ȘI (x ≥ 0))`
C. `NU ((|x| MOD 2 = 0) SAU (x ≥ 0))`
D. `(|x| MOD 2 ≠ 0) SAU (x < 0)`

Raspunsuri:
2) B
8) B
11) A,C

O GRILA INTERESANTA ...

8. Fie următorul subalgoritm, având ca parametru numărul natural nenul n și care returnează un număr natural.

```
Subalgoritm f(n):
    j ← n
    CâtTimp j > 1 execută
       i ← 1
       CâtTimp i ≤ n execută
           i ← 2 * i
       SfCâtTimp
       j ← j DIV 3
    SfCâtTimp
    returnează j
SfSubalgoritm
```

În care dintre următoarele clase de complexitate se încadrează complexitatea timp a algoritmului?

A. $O(\log_2 n)$
B. $O(\log_2^2 n)$
C. $O(\log_3^2 n)$
D. $O(\log_2 \log_3 n)$

UNA SIMILARA ...

29. Fie următorul subalgoritm, având ca parametru numărul natural nenul n și care returnează un număr natural.

```
Subalgoritm f(n):
    j ← n
    CâtTimp j > 1 execută
       i ← 1
       CâtTimp i ≤ n^4 execută
           i ← 4 * i
       SfCâtTimp
       j ← j DIV 2
    SfCâtTimp
    returnează j
SfSubalgoritm
```

În care dintre următoarele clase de complexitate se încadrează complexitatea timp a algoritmului?

A. $O(\log_2 n^2)$
B. $O(\log_2^2 n^2)$
C. $O(\log_4^2 n)$
D. $O(\log_2 \log_4 n)$

Si inca una ...

14. Care dintre algoritmii următori pot fi implementați în așa fel încât să aibă complexitate de timp liniară (O(***n***))?

A. Algoritmul de căutare secvențială a unui element într-un vector de ***n*** numere
B. Algoritmul de sortare prin inserție a unui tablou unidimensional de ***n*** numere
C. Algoritmul de căutare al numărului maxim într-un vector nesortat de ***n*** numere
D. Algoritmul de determinare a sumei elementelor de pe diagonala principală a unei matrice pătratice cu ***n*** linii și ***n*** coloane.

Va multumesc!

- Aceasta prezentare apartine ACADEMIEI-ZECELAINFO
- Materiale realizate de SOMESAN PAUL-IOAN
- VA RUGAM NU DISTRIBUITI MATERIALELE!