Optimizarea programelor folosind operaţii pe biţi

Responsabili:

- Darius Neațu (CA 2019-2021) [mailto:neatudarius@gmail.com]
- Dorinel Filip (CA 2019-2021) [mailto:ion_dorinel.filip@cti.pub.ro]
- Rareș Cheșeli (2018, 2017) [mailto:rares96cheseli@gmail.com]

Obiective

În urma parcurgerii acestui laborator, studentul va fi capabil:

- să înțeleagă conceptele legate de operații pe biți
- să utilizeze operații pe biți pentru optimizare (când este cazul)
- să înteleagă mai multe despre organizarea datelor în memorie
- să gestioneze mai bine memoria folosită într-un anumit program C
- să implementeze o structură de date in C

- Laboratorul curent introduce mai multe noțiuni. Pe multe dintre acestea le veți mai întâlni și la alte cursuri din următorii ani, precum: IOCLA, CN/CN2, AA/PA, PC, PM, SO.
- Secțiunile **Studiu de caz** și **Probleme de interviu** nu sunt obligatorii, dar recomandăm parcurgerea întregului material pentru o viziune de ansamblu mai bună. Atenție! Problemele de interviu sunt mult peste nivelul așteptat la această materie!

Motivație

În cadrul acestui articol vă vom prezenta câteva metode prin care programele se pot optimiza dacă utilizăm eficient operaţiile pe biţi. Un lucru foarte important de reținut este că nu întotdeauna putem folosi aceste operații pentru optimizare, iar laboratorul are ca scop ilustrarea câtorva **exemple**, pe care le puteți întâlni și pe viitor.

Veți învăța în anul 2 la Analiza Algoritmilor despre complexitatea unui algoritm. Vom considera momentan că un algoritm este mai rapid decât altul dacă are mai puțini pași (exemplu un for cu n = 100 de pași este mai rapid decât un for cu 1000 de pași).

În exemplul anterior **performanța** se referă la timp (dacă executăm mai puține instrucțiuni într-un program, ne așteptăm să se termine mai repede). Acest aspect va fi abordat pe larg la materiile AA și PA.

În acest laborator vom vorbi despre altă metrică de măsurare a performanței unui program, mai exact despre **memoria folosită** de un program.

De ce este **important** și acest aspect? Dacă din punct de vedere al timpului de execuție, sunt situații în care putem aștepta mai mult timp pentru a se termina programul, din punctul de vedere al memoriei folosite avem o limitare exactă. Un exemplu simplu este calculatorul nostru, care are 4GB/8GB/16GB. Dacă mașina noastră are X GB RAM, dintre care o parte importantă o ocupă sistemul de operare, asta înseamnă că într-un anumit program nu putem folosi o cantitate nelimitată de RAM (mai multe detalii la CN2, SO). Pentru **simplitate**, momentan presupunem că programul nostru nu poate rula pe o mașină cu X GB, dacă are nevoie de mai mult de X GB.

Dacă ajungem într-o astfel de situație în mod evident trebuie să schimbăm ceva, însă de multe ori putem păstra algoritmul și să facem câteva modificări în implementare, care exploatează anumite abilități ale limbajului C (ex. operații pe biți).

Dimensiunea tipurilor implicite în C. Calculul memoriei unui program

În laboratorul 2 [https://ocw.cs.pub.ro/courses/programare/laboratoare/lab02] au fost prezentate tipurile de date implicite din C și dimensiunea acestora.

Pentru a afla **dimensiunea în bytes** a unei variabile se poate folosi operatorul **sizeof**.

Fie codul din următorul cod.

```
#include <stdio.h>

int main() {
    // afiseaza dimensiunea tipurilor si a unor variabile de un anumit tip
    char xc;
    printf("sizeof(unsigned char) = %ld B\n", sizeof(unsigned char));
    printf("sizeof(char) = %ld B\n", sizeof(char));
    printf("sizeof(xc) = %ld B\n", sizeof(xc));

    short int xs;
    printf("sizeof(unsigned short int) = %ld B\n", sizeof(unsigned short int));
    printf("sizeof(short int) = %ld B\n", sizeof(short int));
    printf("sizeof(xs) = %ld B\n", sizeof(xs));

    int xi;
    printf("sizeof(unsigned int) = %ld B\n", sizeof(unsigned int));
    printf("sizeof(int) = %ld B\n", sizeof(int));
    printf("sizeof(xi) = %ld B\n", sizeof(xi));

    // afiseaza dimensiunea unor tablouri cu dimensiune cunoscuta
    char vc[100];
    short int vs[100];
    int vi[100];
    printf("sizeof(vc) = %ld B\n", sizeof(vc));
    printf("sizeof(vs) = %ld B\n", sizeof(vs));
    printf("sizeof(vi) = %ld B\n", sizeof(vi));

    return 0;
}
```

În urma executării acestui program pe o arhitectură de **32 biți** (ceea ce folosim la PC) vom vedea următorul rezultat.

```
sizeof(unsigned char) = 1 B
sizeof(char) = 1 B
sizeof(xc) = 1 B

sizeof(unsigned short int) = 2 B
sizeof(short int) = 2 B
sizeof(xs) = 2 B

sizeof(unsigned int) = 4 B
sizeof(int) = 4 B
sizeof(xi) = 4 B

sizeof(vc) = 100 B
sizeof(vs) = 200 B
sizeof(vi) = 400 B
```

Putem afla dimensiunea unui tip de date / unei variabile de un anumit tip la **compile time** folosind operatorul sizeof care returnează dimensiunea în bytes a parametrului dat.

sizeof poate fi folosit **și** pentru măsurarea dimesiunii unui vector / matrice alocat(a) static.

Memoria totală folosită de un program poate fi calculată ca **suma** tuturor dimensiunilor ocupate de variabilele din program.

De obicei, ne interesează să știm **ordinul de mărime** al spațiului de memorie alocat, astfel, de cele mai multe ori, putem contoriza doar tablourile.

Un caz special îl poate reprezenta recursivitatea! Punerea parametrilor pe stivă de un număr foarte mare de ori, este echivalent cu declararea unui tablou de valori pe stivă. Aceste variabile nu pot fi neglijate în calculul memoriei!

Vom descoperi mai multe în următoare laboratoare. 

Operatori pe biți în C

Operatorii limbajului C pot fi unari, binari sau ternari, fiecare având o precedenţă şi o asociativitate bine definite (vezi lab02 [https://ocw.cs.pub.ro/courses/programare/laboratoare/lab02]).

În tabelul următor reamintim operatorii limbajului C care sunt folosiți la nivel de bit.

Operator	Descriere	Asociativitate
~	Complement faţă de 1 pe biţi	dreapta-stânga
<< si >>	Deplasare stânga/dreapta a biţilor	stânga-dreapta
&	ŞI pe biţi	stânga-dreapta
^	SAU-EXCLUSIV pe biţi	stânga-dreapta
|	SAU pe biţi	stânga-dreapta
&= și **|=**	Atribuire cu ŞI/SAU	dreapta-stânga
^=	Atribuire cu SAU-EXCLUSIV	dreapta-stânga
<<= şi **>>=**	Atribuire cu deplasare de biţi	dreapta-stânga

Trebuie avută în vedere precedenţa operatorilor pentru obţinerea rezultatelor dorite!

Dacă nu sunteți sigur de precendența unui operator, folosiți o pereche de paranteze rotunde în plus în expresia voastră! Nu exagerați cu parantezele, codul poate deveni ilizibil.

Bitwise NOT

Bitwise NOT (**complement față de 1**) este operația la nivel de bit care următorul tabel de adevăr.

x	~x
0	1
1	0

Evident putem extinde această operație și la nivel de număr. Operația se aplică separat pentru fiecare rang binar.

	b_2	b_1	b_0
x = 3	0	1	1
~x = 4	1	0	0

Bitwise AND

Bitwise AND (**ȘI pe biți**) este operația la nivel de bit care următorul tabel de adevăr.

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Evident putem extinde această operație și la nivel de număr. Operația se aplică separat pentru fiecare rang binar.

	b_2	b_1	b_0
x = 3	0	1	1
y = 7	1	1	1
x & y = 3	0	1	1

Bitwise OR

Bitwise OR (**SAU pe biți**) este operația la nivel de bit care următorul tabel de adevăr.

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Evident putem extinde această operație și la nivel de număr. Operația se aplică separat pentru fiecare rang binar.

	b_2	b_1	b_0
x	**0**	**1**	**1**
y	**1**	**0**	**1**
x | y = 7	**1**	**1**	**1**

Bitwise XOR

Bitwise XOR (**SAU-EXCLUSIV pe biți**) este operația la nivel de bit care următorul tabel de adevăr.

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Evident putem extinde această operație și la nivel de număr. Operația se aplică separat pentru fiecare rang binar.

	b_2	b_1	b_0
x = 3	**0**	**1**	**1**
y = 5	**1**	**0**	**1**
x ^ y = 6	**1**	**1**	**0**

Bit LOGICAL SHIFT

În C sunt definite doar shiftări logice. Acestea pot fi la stânga (<<) sau la dreapta (>>), reprezentând deplasarea in binar a cifrelor și completarea pozițiilor "golite" cu zerouri.

LEFT SHIFT

Efectul unei deplasări la stânga cu un rang binar este echivalent cu înmulțirea cu 2 a numărului din baza 10. Dacă rezultatul nu are loc pe tipul de date folosit, atunci se pot pierde din biți!

Se poate deduce următoarea relație: $n << k = n * 2^k$.

Fie un exemplu de deplasarea la stânga, pentru un număr **pe 3 biți** .

	b_2	b_1	b_0
x = 3	**0**	**1**	**1**
x « 1 = 6	**1**	**1**	**0**
x « 2 = 8	**1**	**0**	**0**

RIGHT SHIFT

Efectul unei deplasări la dreapta cu un rang binar este echivalent cu împărțirea întreagă la 2 a numărului din baza 10.

Se poate deduce următoarea relație: $n >> k = [n/2^k]$.

Fie un exemplu de deplasarea la dreapta, pentru un număr **pe 3 biți** .

	b_2	b_1	b_0
x = 3	**0**	**1**	**1**
x » 1 = 1	**0**	**0**	**1**
x » 2 = 0	**0**	**0**	**0**

Lucrul cu măști

Având la dispoziție operațiile prezentate mai sus, putem răspunde la următoarele întrebări.

- Cum verificăm dacă **bitul i** dintr-un număr n este **setat** ?
- Cum **setăm bitul i** dintr-un număr n?
- Cum **resetăm bitul i** dintr-un număr n?

Pentru a răspunde ușor, pentru fiecare întrebare vom aplica o operație pe biți între n și o valoarea numită **mască** .

Cum verificăm dacă bitul i dintr-un număr n este setat?

Detectarea bitului:

- pas 1: se aplică următoarea operația **x = n & mask** , unde **mask = (1 « i)**

	b_7	...	b_{i+1}	b_i	b_{i-1}	...	b_0	
n	*	...	*	?	*	...	*	
mask	0	...	0	1	0	...	0	op
x	0	...	0	?	0	...	0	

- pas 2: deoarece **?** poate avea două valori, x poate fi 0 sau 2^i
  - dacă **x == 0** , atunci **bitul i este 0**
  - dacă **x > 0** , atunci **bitul i este 1**

```
// is_set
// byte    - byte de intrare pentru care vreau sa verific un bit
// i       - indexul bitului din byte
// @return - 1, daca bitul este 1
//           0, daca bitul este 0
int is_set(char byte, int i) {
    int mask = (1 << i);
    return (byte & mask) != 0;
}

...
if (is_set(mybyte, i)) {
    printf("bitul %d din byteul %d este setat!\n", i, mybyte);
} else {
    printf("bitul %d din byteul %d NU este setat!\n", i, mybyte);
}
...
```

Această întrebare ne oferă valoarea bitului i.

Dacă "valoarea este 1", atunci vom spune că "bitul este setat".

Dacă "valoarea este 0", atunci vom spune că "bitul nu este "setat".

Pentru a **verifica** valoarea bitului i din numărul n, practic noi ar trebui să privim numărul astfel:

	b_7	b_6	...	b_i	...	b_1	b_0
n	*	*	...	?	...	*	*

unde * înseamnă **don't care** (de la PL), iar **?** este valoarea pe care o căutăm.

Deci am vrea să facem, după cum am zis mai sus, o operație de tipul "scoate" doar bitul i din număr, iar în rest lasă 0 (pentru a evidenția bitul nostru).

	b_7	...	b_{i+1}	b_i	b_{i-1}	...	b_0	
n	*	...	*	?	*	...	*	
mask	m_7	...	m_{i+1}	m_i	m_{i-1}	...	m_0	op
n op mask	0	...	0	?	0	...	0	

op este o operație, iar **mask** un număr. Să analizăm cine pot fi **op** și **biții din mască** (m_i).

Dorim ca:

- $?\ op\ m_i = ?$, adică operația op aplicată pe ? și m_i, va avea mereu ca rezultat pe ?
- $*\ op\ m_j = 0$ (unde i != j), adică operația op aplicată pe orice valoare și m_j, va da 0

Observăm că:

- 1 este elementul neutru pentru **ȘI**, ceea ce verifică **? & 1 = ?** , oricare are fi ? un bit
- 0 este elementul care poate "șterge" un bit prin **Și**, ceea ce verifică *** & 0 = 0** , oricare ar fi * un bit

Cum setăm (valoarea devine 1) bitul i dintr-un număr n?

Setarea bitului:

- pas 1: se aplică următoarea operația **n = n | mask** , unde **mask = (1 « i)**

	b_7	...	b_{i+1}	b_i	b_{i-1}	...	b_0	
n	n_7	...	n_{i+1}	*	n_{i-1}	...	n_0	
mask	0	...	0	1	0	...	0	**op**
n op mask	n_7	...	n_{i+1}	**1**	n_{i-1}	...	n_0	

- explicație:
  - orice valoare are avea bitul $*$ va fi suprascris cu 1
  - ceilalti biți vor fi copiați

```
// set
// byte    - byte de intrare pentru care vreau sa setez un bit
// i       - indexul bitului din byte
// @return - noul byte
char set(char byte, int i) {
    int mask = (1 << i);
    return (byte | mask);
}

...
mybyte = set(mybyte, i);
...
```

Dorim să facem următoarea operație: schimba doar bitul i in 1, iar pe ceilalți lasă-i neschimbați.

	b_7	...	b_{i+1}	b_i	b_{i-1}	...	b_0	
n	n_7	...	n_{i+1}	*	n_{i-1}	...	n_0	
mask	m_7	...	m_{i+1}	m_i	m_{i-1}	...	m_0	**op**
n op mask	n_7	...	n_{i+1}	**1**	n_{i-1}	...	n_0	

op este o operație, iar **mask** un număr. Să analizăm cine pot fi **op** și **biții din mască** (m_i).

Dorim ca:

- $*\ op\ m_i = 1$, adică operația op aplicată pe $*$ (orice) și m_i , va avea mereu ca rezultat pe 1
- $n_j\ op\ m_j = n_j$ (unde i != j), adică operația op aplicată pe n_j și m_j, va da n_j

Observăm că:

- 1 este elementul care poate "umple" un bit prin **SAU**, ceea ce verifică $*|1 = 1$, oricare ar fi * un bit
- 0 este elementul neutru pentru **SAU**, ceea ce verifică $n_j|0 = n_j$, oricare are fi n_j un bit

Cum resetăm (valoarea devine 0) bitul i dintr-un număr n?

Resetarea bitului:

- pas 1 / 1: se aplică următoarea operația **n = n & mask** , unde **mask = ~(1 « i)**

	b_7	...	b_{i+1}	b_i	b_{i-1}	...	b_0	
n	n_7	...	n_{i+1}	*	n_{i-1}	...	n_0	
mask	1	...	1	0	1	...	1	**op**
n op mask	n_7	...	n_{i+1}	**0**	n_{i-1}	...	n_0	

- explicație:
  - orice valoare are avea bitul $*$ va fi suprascris cu 0
  - ceilalti biți vor fi copiați

```
// reset
// byte    - byte de intrare pentru care vreau sa resetez un bit
// i       - indexul bitului din byte
// @return - noul byte
char reset(char byte, int i) {
    int mask = ~(1 << i);
    return (byte & mask);
}

...
mybyte = reset(mybyte, i);
...
```

Dorim să facem următoarea operație: schimba doar bitul i in 0, iar pe ceilalți lasă-i neschimbați.

	b_7	...	b_{i+1}	b_i	b_{i-1}	...	b_0	
n	n_7	...	n_{i+1}	*	n_{i-1}	...	n_0	
mask	m_7	...	m_{i+1}	m_i	m_{i-1}	...	m_0	**op**
n op mask	n_7	...	n_{i+1}	**0**	n_{i-1}	...	n_0	

op este o operație, iar **mask** un număr. Să analizăm cine pot fi **op** și **biții din mască** (m_i).

Dorim ca:

- $*\ \ op\ \ m_i = 0$, adică operația op aplicată pe $*$ (orice) și m_i, va avea mereu ca rezultat pe 0
- $n_j\ \ op\ \ m_j = n_j$ (unde i != j), adică operația op aplicată pe n_j și m_j, va da n_j

Observăm că:

- 0 este elementul care poate "șterge" un bit prin **ȘI**, ceea ce verifică $\boxed{* \ \& \ 0 = 0}$, oricare ar fi * un bit
- 1 este elementul neutru pentru **ȘI**, ceea ce verifică $n_j | 1 = n_j$, oricare are fi n_j un bit

Exerciții

Checker laborator 7 CB\CD [https://drive.google.com/drive/u/1/folders/1qB6EZLGVubKbuTXMtMue06egH_8fo25M] Teste Problema 1 [https://drive.google.com/open?id=1oBRfwcYfuosDaBRkeq7f_XmQYayOqbZq] Teste Problema 2 [https://drive.google.com/open?id=16kc4UllXMXb62KYIyZ8AndclaRjBTQ0Q] Teste Problema 3 [https://drive.google.com/open?id=1kO2oT9cF9Oih0Jav00Ji9gHuhSnsMo5R] Teste Problema 4 [https://drive.google.com/open?id=1MVOlZdZv7aZ3DAdMRn2QQL2DkjkCb6Xh] Teste Problema 5 [https://drive.google.com/open?id=1TsXI7HkbLOR2kwM0lLe-1z3qcSs3eT0z] Teste Problema 6 [https://drive.google.com/open?id=1QjYcHcDSApXZ1ZTloxqibOAU9GSckaxE]

Precizari CB\CD

- Arhivele 4, 5, 6 testeaza reuniunea, intersectia respectiv diferenta seturilor.
- Intrarea corespunde functiei set_read (n, urmat de n elemente)
- Ref corespunde functiei set_print aplicata pe setul obtinut (cardinalul setului, urmat pe urmatoarea linie de elementele din set)

- Arhiva 7 corespunde problemei de bonus B1 (.ref contine rezultatul pentru get_lsb urmat de rezultatul pentru get_msb)
- Arhiva 8 corespunde problemei B2.

Precizari generale

Rezolvați împreună cu asistentul pe tablă, exercițiile 0-4, apoi rezolvați invidual exercițiul 5.

0: Verificare că un număr e par

Să se verifice folosind operații pe biți că un număr natural n e par.

```
int is_even(int n);
```

1. Calcul putere a lui 2 (0p)

Să se scrie o funcție care să calculeze 2^n, unde $n <= 30$.

```
int pow2(int n);
```

Răspundeți la întrebarea: **are sens** să scriem o funcție?

2. Negarea biților unui număr n (0p)

Să se scrie o funcție care să nege biții unui număr n (32 biți).

```
int flip_bits(int n);
```

Răspundeți la întrebarea: **are sens** să scriem o funcție?

3. Afișarea biților unui număr n (0p)

Să se scrie o funcție care să afișeze toți biții unui număr întreg pe 32 biți.

```
void print_bits(int n);
```

4. Verificare că un număr este putere al lui 2 (0p)

Să se scrie o funcție care verifică dacă un număr întreg n pe 32 biți este puterea a lui 2. Funcția va returna 1 dacă n este putere a lui 2, 0 altfel.

```
int is_power2(int n);
```

Analizați reprezentarea în baza 2 a lui n (ex. n = 16 si n = 5).

Implementați invidual următoarea problemă.

5. bitset (10p)

O mulțime de numere întregi poate fi reprezentată astfel: spunem că un număr **i** aparține unei mulțimi S dacă bit-ul al i-lea din vectorul S are valoarea 1.

Pentru eficientă, vectorul S va conține date de tipul **unsigned char** (reamintim ca **sizeof(unsigned char) == 1 byte** adică **8 biți**).

Pentru a folosi cu ușurință același cod facând schimbări minime (de exemplu schimbăm dimensiunea maximă a unei mulțimi), putem să ne definim propriul tip astfel:

```
#define SET_SIZE 100                  // aceasta este o macrodefinitie (momentan o putem privi ca pe O CONSTANTA CARE ARE VALOAREA 100
typedef unsigned char SET[SET_SIZE];   // definesc tipul SET, care este un vector cu maxim 100 de elemente de tip unsigned char
```

Cele două linii de mai sus vor fi puse imediat dupa includerea directivelor header!

5.1

Implementați următoarele funcții. Realizați un program în C prin care să demonstrați că funcțiile implementate funcționează.

Există un exemplu [https://ocw.cs.pub.ro/courses/programare/laboratoare/lab07-bitset-example] detaliat care vă explică cum functionează aceastea.

Treceți la subpunctul următor abia după ce v-ați asigurat că acestea funcționează.

- **adăugarea** unui element în mulțime

```
// insert_in_set(s, n) - adauga numarul n in multimea s
void insert_in_set(SET s, unsigned int n);
```

- **ștergerea** unui element din mulțime

```
// delete_from_set(s, n) - scoate numarul n din multime s
void delete_from_set(SET s, unsigned int n);
```

- **verificarea** faptului că un element n **aparține** unei mulțimi

```
// is_in_set(s, n) - returneaza 1 daca n este in s, 0 altfel
int is_in_set(SET s, unsigned int n);
```

- **ștergerea** tuturor elementelor din mulțime

```
// delete_all_from_set(s) - elimina toate elementele din multime
void delete_all_from_set(SET s);
```

- calcularea **cardinalul** unei mulțimi

```
// card_set(s) - returneaza cardinalul multimii s
int card_set(SET s);
```

- verificarea faptului că mulțimea este **vidă**

```
// is_empty_set(s) - verifica daca multimea este sau nu goala
// returneaza 1 daca este, 0 daca nu este
int is_empty_set(SET s);
```

- o funcție care **să citească de la tastatură** o mulțime

```
// read_set(s) - functia citeste numarul n de elemente care se afla in a
// apoi citeste cele n numere si le insereaza in a
// va returna numarul n citit (numarul de elemente)
int read_set(SET s);
```

- o funcție care **să afișeze** pe ecran elementele care se află într-o mulțime

```
// print_set(s) - functia printeaza elementele multimii s
void print_set(SET s);
```

Urmăriți acest exemplu cu bitset [https://ocw.cs.pub.ro/courses/programare/laboratoare/lab07-bitset-example] pentru a înțele cum funcționeazăaceste operații.

5.2

Realizati un program care, utilizând metodele defite anterior, citește 2 mulțimi A (n și B și afișează: $AUB, A \cap B, A - B, B - A$.

Pentru a realiza acest lucru, va trebui să implementați următoarele funcții:

- **reuniunea** a două mulțimi (**1p**)

```
// c = a U b
void merge_set(SET a, SET b, SET c);
```

- **intersecția** a două mulțimi (**1p**)

```
// c = a n b
void intersect_set(SET a, SET b, SET c);
```

- **diferența** a două mulțimi (**1p**)

```
// c = a \ b
void diff_set(SET a, SET b, SET c);
```

În final va trebui sa creați o funcție **main** și să faceți un program care rezolvă cerința folosind funcțiile implementate.

B1. Optimizations

O să învățați pe viitor că nu toate înstrucțiunile sunt la fel din punct de vedere al timpului de execuției. Ca un punct de start, gândiți-vă că dacă ați face pe foaie o adunare sau o înmultire, durează mai mult să înmulțiți decât să adunați. Această dificulate o are și procesorul din laptopul vostru!

Pentru a-l ajuta și a face programul mai rapid, putem înlocui operații costisitoare (***** , **/**) cu altele mai puțin costistoare (**+** , **-**). Cele mai rapide instrucțiuni sunt cele care lucrează direct pe biți (deoarece numerele sunt stocate în memorie în binar, deci este modul natural de lucru pentru calculator).

Acest exercițiu vă propune să aplicați aceste optimizări pe codul vostru!

Pentru a completa acest bonus, **NU** aveți voie să folosiți operatorii **/, *, %** ! Încercați să folosiți operații pe biți!

B2. MSB/LSB

Să se scrie o câte o funcție pentru aflarea MSB(Most significant bit), respectiv LSB(Least significant bit), pentru un număr n pe 32 biți.

```
int get_lsb(int n);
int get_msb(int n);
```

Analizați reprezentarea în baza 2 a lui n.

Probleme de interviu

Pentru cei interesați, recomandăm rezolvarea și următoarelor probleme, care sunt des întâlnite la interviuri.

Atenție! Problemele din această categorie au un nivel de dificultate ridicat, peste cel cerut la cursul de PC.

Recomandăm totuși rezolvarea acestor probleme pentru cei care doresc să aprofundeze lucrul cu operații pe biți.

Se dă un număr n natural pe 32 de biți. Se cere să se calculeze numărul obținut prin interschimbarea biților de rang par cu cel de rang impar.

Exemplu: n = 2863311530 = > m = 1431655765

Hint: Reprezentarea numerelor în baza 2 ([http://www.binaryhexconverter.com/decimal-to-binary-converter [http://www.binaryhexconverter.com/decimal-to-binary-converter] | convertor]]).

Fie un șir cu **2n + 1** numere întregi, dintre care n numere apar de câte 2 ore, iar unul singur este unic. Să se gasească elementul unic.

Exemplu:

```
n = 5 și sirul [1, 4, 4, 1, 5]
Numărul unic este 5.
```

Hint: Încercați să nu folosiți tablouri.

Follow-up 1: Șirul are $2 * n + (2 * p + 1)$ numere. Se știe că un singur număr apare de un număr impar de ori (2p + 1), iar celelalte apar de un număr par de ori. Cum găsiți numărul care apare de un număr impar de ori?

Exemplu:

```
n = 5 și sirul [1, 1, 4, 4, 4, 4, 5, 5, 5]
Răspunsul este 5.
```

Follow-up 2: Șirul are $2n + 2$ numere, n numere apar de câte 2 ori, iar 2 numere sunt unice. Cum găsiți cele 2 numer unice?

Exemplu:

```
n = 5 și sirul [1, 4, 4, 1, 5, 6]
Numărele unice sunt 5 și 6.
```

TODO: sursă

Realizează o funcție de căutare pe binară, utilizând operații pe biți pentru optimizarea acestei implementări.

Follow up: Puteți găsi alt algoritm care să nu se bazeze pe împarțirea vectorului în două și compararea elementului din mijloc cu cel cautat?

Hint: caut bin [http://www.infoarena.ro/problema/cautbin] și Multe "smenuri" de programare in C/C++... si nu numai! [http://www.infoarena.ro/multe-smenuri-de-programare-in-cc-si-nu-numai]

Se dau n grămezi, fiecare conţinând un anumit număr de pietre. Doi jucători vor începe să ia alternativ din pietre, astfel: la fiecare pas, jucătorul aflat la mutare trebuie să îndepărteze un număr nenul de pietre dintr-o singură grămadă. Câştigătorul este cel care ia ultima piatră. Să se determine dacă jucătorul care ia primele pietre are strategie sigură de câştig.

Exemple

```
n = 4, gramezi = [1 3 5 7], raspuns = NU
n = 3, gramezi = [4 8 17],  raspuns = DA
```

Hint: nim [http://www.infoarena.ro/problema/nim]

Enunt: sushi [http://www.infoarena.ro/problema/sushi]