

## SEMINAR 1

### Operatori pe biți

1. Să se interschimbe valorile a două variabile de tip întreg folosind operatorul  $\wedge$  (XOR/"sau exclusiv" pe biți).

#### Rezolvare:

Operatorul  $\wedge$  are următoarele proprietăți:

- a)  $x \wedge x = 0$
- b)  $x \wedge 0 = x$
- c)  $x \wedge y = y \wedge x$
- d)  $(x \wedge y) \wedge z = x \wedge (y \wedge z)$

Folosind proprietățile enunțate anterior, putem interschimba valorile a două variabile  $x$  și  $y$  folosind operatorul  $\wedge$ , astfel (corectitudinea algoritmului este demonstrată în comentarii):

$$\begin{aligned} x &= x \wedge y \\ y &= x \wedge y \quad \# \quad y = (x \wedge y) \wedge y = x \wedge (y \wedge y) = x \wedge 0 = x \\ x &= x \wedge y \quad \# \quad x = (x \wedge y) \wedge x = x \wedge (y \wedge x) = (x \wedge x) \wedge y = 0 \wedge y = y \end{aligned}$$

De asemenea, putem observa faptul că expresia  $x \wedge y$  furnizează diferențele dintre valorile  $x$  și  $y$  din punct de vedere al biților aflați pe aceeași poziție. Argumentați corectitudinea algoritmului folosind această observație!

2. Să se verifice dacă un număr natural nenul  $n$  este de forma  $2^k$  sau nu. În caz afirmativ să se afișeze exponentul  $k$ , altfel să se afișeze un mesaj corespunzător.

#### Rezolvare:

Un număr natural  $n$  este de forma  $2^k$  dacă reprezentarea sa binară conține un singur bit nenul. De exemplu, reprezentarea binară a numărului  $n = 512 = 2^9$  este  $0b100000000$ , în timp ce numărul  $n = 600$  are reprezentarea binară  $0b1001011000$ . Practic, pozițiile biților nenuli din reprezentarea binară a unui număr natural  $n$  (numerotate începând cu 0 de la dreapta spre stânga) indică puterile lui 2 pe care trebuie să le însumăm pentru a-l obține pe  $n$  (în exemplul următor am considerat numărul  $n = 600$ ):

| Poziția bitului                        | 9     | 8     | 7     | 6     | 5     | 4     | 3     | 2     | 1     | 0     |
|--|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| Valoarea bitului                       | 1     | 0     | 0     | 1     | 0     | 1     | 1     | 0     | 0     | 0     |
| Puterea lui 2 corespunzătoare poziției | $2^9$ | $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |

Astfel, obținem faptul că  $n = 600 = 2^9 + 2^6 + 2^4 + 2^3 = 512 + 64 + 16 + 8$ .

O primă variantă de rezolvare a problemei constă în eliminarea din numărul  $n$  a bitului de pe poziția 0 (denumit și *bitul cel mai puțin semnificativ* sau *bitul de paritate*) cât timp acesta este nul, iar dacă la sfârșit valoarea lui  $n$  a devenit 1 atunci  $n$  este de forma  $2^k$ :

```
n = int(input("n = "))

aux = n
k = 0
while aux & 1 == 0:
    aux = aux >> 1
    k = k + 1

if aux == 1:
    print("Numarul", n, "este egal cu 2**", k)
else:
    print("Numarul", n, "nu este o putere a lui 2")
```

Deoarece lungimea reprezentării binare a unui număr natural  $n$  este egală cu  $1 + \lceil \log_2 n \rceil$ , rezultă faptul că algoritmul anterior are complexitatea  $\mathcal{O}(\log_2 n)$ .

O rezolvare mai eficientă a acestei probleme se bazează pe următoarea idee: dacă dintr-un număr natural nenul  $n$  scădem 1, atunci în reprezentarea sa binară toți biții egali cu 0 de la sfârșit vor deveni 1 până se va întâlni primul bit egal cu 1, iar acesta va deveni 0. Astfel, valoarea expresiei  $n \& (n-1)$  va fi egală cu 0 dacă și numai dacă reprezentarea binară a numărului natural nenul  $n$  conține un singur bit nenul (i.e., numărul  $n$  este o putere a lui 2), așa cum se poate observa din următoarele exemple:

|                             |  |                             |
|-----------------------------|--|-----------------------------|
| $n = 512 = 0b1000000000$    |  | $n = 600 = 0b1001011000$    |
| $n-1 = 511 = 0b0111111111$  |  | $n-1 = 599 = 0b1001010111$  |
| $n \& (n-1) = 0b0000000000$ |  | $n \& (n-1) = 0b1001010000$ |

În plus, dacă  $n = 2^k$ , atunci putem calcula direct exponentul  $k = \log_2 n$ . Un program Python în care sunt implementate aceste idei este următorul:

```
import math

n = int(input("n = "))

if n & (n-1) == 0:
    print("Numarul", n, "este egal cu 2**", int(math.log2(n)))
else:
    print("Numarul", n, "nu este o putere a lui 2")
```

Deoarece în majoritatea limbajelor de programare actuale funcțiile care calculează logaritmi sunt foarte rapide, putem considera faptul că algoritmul de mai sus are complexitatea  $\mathcal{O}(1)$ .

3. Să se determine în mod eficient numărul de biți nenuli din reprezentarea binară a unui număr natural nenul. De exemplu, reprezentarea binară a numărului 600 este  $0b1001011000$  și conține 4 biți nenuli.

#### Rezolvare:

O variantă directă de rezolvare a problemei constă în numărarea biților nenuli din reprezentarea binară a numărului dat:

```
n = int(input("n = "))
print("Reprezentarea binară a numărului", n, "este:\n", bin(n))
aux = n
k = 0
while aux != 0:
    if aux & 1 == 1:
        k = k + 1
    aux = aux >> 1
print("Reprezentarea binară a numărului", n, "conține", k, "biți nenuli")
```

Evident, algoritmul de mai sus are complexitatea  $\mathcal{O}(\log_2 n)$ .

Așa cum deja am menționat, orice număr natural  $n$  poate fi scris ca o sumă de puteri ale lui 2, iar pozițiile biților nenuli din reprezentarea sa binară reprezintă, de fapt, exponenții puterilor respective. Astfel, numărul biților nenuli din reprezentarea binară a unui număr natural  $n$  este egal cu numărul puterilor lui 2 utilizate pentru scrierea lui  $n$  în baza 2 (vezi rezolvarea problemei anterioare).

Pentru a calcula în mod eficient numărul  $k$  de biți nenuli din reprezentarea binară a numărului natural  $n$  vom folosi ideea din problema anterioară. Astfel, dacă  $n \& (n-1)$  este o valoare nenulă, înseamnă că mai există puteri ale lui 2 pe care trebuie să le însumăm pentru a-l obține pe  $n$ , deci mai există biți nenuli în reprezentarea binară a lui  $n$ , ceea ce înseamnă că trebuie să reluăm procedeul până când  $n$  devine 0 (exemplul de mai jos este dat pentru  $n = 600$ ):

```

n = 0b1001011000
n - 1 = 0b1001010111
n = n & (n-1) = 0b1001010000

n = 0b1001010000
n - 1 = 0b1001001111
n = n & (n-1) = 0b1001000000

n = 0b1001000000
n - 1 = 0b1000111111
n = n & (n-1) = 0b1000000000

n = 0b1000000000
n - 1 = 0b0111111111
n = n & (n-1) = 0b0000000000
```

Practic, prin fiecare instrucțiune de atribuire  $n = n \& (n-1)$  eliminăm câte un bit nenul din reprezentarea binară a lui  $n$ , fără a mai lua în considerare și biții nuli!

Programul Python în care este implementată această idee este următorul:

```
n = int(input("n = "))
print("Reprezentarea binară a numărului", n, "este:\n", bin(n))

aux = n
k = 0
while aux != 0:
    k = k + 1
    aux = aux & (aux - 1)

print("Reprezentarea binară a numărului", n, "conține", k, "biți nenuli")
```

Complexitatea acestui algoritmul este dată de numărul  $k$  de biți nenuli din reprezentarea binară a numărului natural  $n$ . Deoarece valoarea lui  $k$  nu se poate calcula direct pe baza valorii lui  $n$ , putem afirma doar faptul că algoritmul are complexitatea maximă  $\mathcal{O}(\log_2 n)$ . Totuși, pentru majoritatea valorilor numărului  $n$ , această variantă va fi mai eficientă decât prima variantă de rezolvare prezentată!

Observați faptul că problema anterioară este un caz particular al acestei probleme, respectiv cazul în care suma puterilor lui 2 conține un singur termen!

**4.** Să se găsească lungimea maximă a unei secvențe de biți egali cu 1 din reprezentarea binară a unui număr natural dat.

#### Rezolvare:

O variantă directă de rezolvare a acestei probleme constă în actualizarea lungimii  $lcrt$  a secvenței curente de biți egali cu 1 și a lungimii maxime  $lmax$  a unei secvențe de biți egali cu 1 în funcție de valoarea bitului de paritate, astfel:

```
n = int(input("n = "))
print("Reprezentarea binară a numărului", n, "este:\n", bin(n))
aux = n
lcrt = lmax = 0
while aux != 0:
    if aux & 1 == 1:
        lcrt = lcrt + 1
    else:
        lcrt = 0
    if lcrt > lmax:
        lmax = lcrt
    aux = aux >> 1

print("Lungimea maximă a unei secvențe de biți nenului este", lmax)
```

Evident, algoritmul de mai sus are complexitatea  $\mathcal{O}(\log_2 n)$ .

Pentru a calcula în mod eficient lungimea maximă a unei secvențe de biți nenuli din reprezentarea binară a unui număr natural  $n$  vom folosi următoarea idee: prin deplasarea cu o poziție spre stânga a biților numărului  $n$  toate secvențele de biți egali cu 1 se vor deplasa cu o poziție spre stânga, deci primului bit din fiecare secvență de biți nenuli din  $n \ll 1$  îi va corespunde unui bit egal cu 0 în reprezentarea binară a numărului  $n$ . În consecință, prin instrucțiunea de atribuire  $n = n \& (n \ll 1)$  vom elimina primul bit al fiecărei secvențe de biți nenuli din reprezentarea binară a numărului  $n$ , deci lungimile tuturor secvențelor de biți nenuli se vor micșora cu 1, așa cum se poate observa în exemplul de mai jos, în care  $n = 14130$ :

```

n = 0b11011100110010
n << 1 = 0b10111001100100
n = n & (n << 1) = 0b10011000100000

n = 0b10011000100000
n << 1 = 0b00110001000000
n = n & (n << 1) = 0b00010000000000

n = 0b00010000000000
n << 1 = 0b00100000000000
n = n & (n << 1) = 0b00000000000000

```

Se observă cu ușurință faptul că numărul  $n$  va deveni egal cu 0 după un număr de iterații egal cu lungimea maximă a unei secvențe de biți nenuli din reprezentarea sa binară!

Programul Python în care este implementată această rezolvare este următorul:

```

n = int(input("n = "))
print("Reprezentarea binară a numărului", n, "este:\n", bin(n))

aux = n
lmax = 0
while aux != 0:
    aux = aux & (aux << 1)
    lmax = lmax + 1

print("Lungimea maximă a unei secvențe de biți nenului este", lmax)

```

Complexitatea acestui algoritmul este dată de lungimea maximă  $lmax$  a unei secvențe de biți nenuli din reprezentarea binară a numărului natural  $n$ . Deoarece valoarea lui  $lmax$  nu se poate calcula direct pe baza valorii lui  $n$ , putem afirma doar faptul că algoritmul are complexitatea maximă  $\mathcal{O}(\log_2 n)$ . Totuși, pentru majoritatea valorilor numărului  $n$ , această variantă va fi mai eficientă decât prima variantă prezentată!

5. Se citește un șir format din cel puțin 3 numere naturale cu proprietatea că fiecare valoare distinctă apare de exact două ori în șir, mai puțin una care apare o singură dată. Să se afișeze valoarea care apare o singură dată în șir. De exemplu, în șirul 100, -7, 20, 20, 1, -7, 1, 3, 100 valoarea 3 apare o singură dată.

**Rezolvare:**

Prin aplicarea operatorului  $\wedge$  între perechi de numere egale se obține valoarea 0, deci aplicând acest operator între toate numerele date vom obține chiar numărul  $x$  care apare o singură dată în șir:

```
n = int(input("Numărul de valori: "))
x = 0
for i in range(n):
    v = int(input("Valoare: "))
    x = x ^ v
print("Numărul care apare o singură dată în șir:", x)
```

Complexitatea acestui algoritmul este, evident,  $\mathcal{O}(n)$ .

6. Să se calculeze numărul  $x$  obținut prin aplicarea operatorului XOR între toate elementele tuturor submulțimilor mulțimii  $A = \{1, 2, \dots, n\} \subset \mathbb{N}$ , mai puțin mulțimea vidă. De exemplu, dacă  $n = 3$  obținem  $x = (1) \wedge (2) \wedge (3) \wedge (1 \wedge 2) \wedge (1 \wedge 3) \wedge (2 \wedge 3) \wedge (1 \wedge 2 \wedge 3) = 0$  (am folosit parantezele doar pentru a pune în evidență elementele submulțimilor lui  $A$ ).

**Indicație de rezolvare:**

O rezolvare a acestei probleme folosind forța brută (i.e., generăm toate submulțimile lui  $A$  și calculăm numărul  $x$ ) va avea complexitatea exponențială  $\mathcal{O}(2^n)$ .

O variantă de rezolvare mult mai eficientă se obține observând faptul că, pentru  $n \geq 2$ , fiecare element al mulțimii  $A$  va apărea într-un număr par de submulțimi (orice element  $k \geq 2$  al mulțimii  $A$  va fi adăugat tuturor submulțimilor care se pot forma cu numerele  $1, 2, \dots, k-1$ , deci elementul  $k$  va apărea în  $2^{k-1}$  submulțimi), deci vom obține  $x = 0$ . Evident, pentru  $n = 1$  numărul cerut este  $x = 1$ . Complexitatea acestei variante de rezolvare este  $\mathcal{O}(1)$ !

7. Se citesc  $n - 1$  numere naturale distincte dintre primele  $n$  numere naturale nenule. Să se afișeze numărul lipsă  $x$ .

**Exemple:**

- Dacă se citesc numerele 5, 1, 4, 2, atunci numărul lipsă este  $x = 3$ .
- Dacă se citesc numerele 1, 2, 3, 4, atunci numărul lipsă este  $x = 5$ .
- Dacă se citesc numerele 2, 3, 4, 5, atunci numărul lipsă este  $x = 1$ .

**Rezolvare:**

Există mai multe metode de rezolvare pentru această problemă, având diverse complexități:

- a) *se caută căutarea fiecare număr de la 1 la  $n$  într-o listă cu cele  $n - 1$  numere citite*
- complexitate computațională:  $\mathcal{O}(n^2)$
  - spațiu de memorie utilizat:  $\mathcal{O}(n)$

- b) se sortează cele  $n - 1$  numere citite și se caută prima pereche de numere aflate pe poziții consecutive care nu sunt consecutive (atenție la cazurile particulare când lipsește numărul 1 sau numărul  $n$ )
- complexitate computațională:  $\mathcal{O}(n \log_2 n)$  – dacă folosim Quicksort sau Mergesort
  - spațiu de memorie utilizat:  $\mathcal{O}(n)$
- c) se citesc, pe rând, cele  $n - 1$  numere și se marchează într-un vector de apariții/frecvențe, după care se caută primul număr nemarcat
- complexitate computațională:  $\mathcal{O}(n)$
  - spațiu de memorie utilizat:  $\mathcal{O}(n)$
- d) calculăm diferența dintre suma primelor  $n$  numere naturale și suma numerelor citite
- complexitate computațională:  $\mathcal{O}(n)$
  - spațiu de memorie utilizat:  $\mathcal{O}(1)$

Rezolvarea problemei folosind operatori pe biți constă în aplicarea operatorului  $\wedge$  între cele  $n - 1$  numere citite și toate numerele naturale de la 1 la  $n$  (vezi problema 5). De exemplu, dacă se citesc numerele 5, 1, 4, 2 (deci  $n = 5$ ), atunci numărul lipsă este  $x = (5 \wedge 1 \wedge 4 \wedge 2) \wedge (1 \wedge 2 \wedge 3 \wedge 4 \wedge 5) = 3$ .

Programul Python care implementează această rezolvare este următorul:

```
n = int(input("n = "))
x = 0
for k in range(1, n):
    v = int(input("v = "))
    x = x ^ v ^ k

x = x ^ n
print("Numărul lipsă:", x)
```

Complexitate computațională a acestui algoritm este  $\mathcal{O}(n)$ , iar spațiul de memorie utilizat este  $\mathcal{O}(1)$ .

8. Se citesc  $n - 2$  numere naturale distincte dintre primele  $n$  numere naturale nenule. Să se afișeze numerele lipsă  $x$  și  $y$ .

#### Exemple:

- Dacă se citesc numerele 5, 1, 4, atunci numerele lipsă sunt  $x = 2$  și  $y = 3$ .
- Dacă se citesc numerele 1, 3, 5, atunci numerele lipsă sunt  $x = 2$  și  $y = 4$ .
- Dacă se citesc numerele 2, 3, 4, atunci numerele lipsă sunt  $x = 1$  și  $y = 5$ .

#### Rezolvare:

Această problemă este asemănătoare cu problema anterioară, iar modalitățile de rezolvare ale problemei anterioare pot fi adaptate destul de ușor și pentru ea. Astfel, primele 3 modalități de rezolvare ale problemei anterioare, care utilizează toate un spațiu de memorie de ordinul  $\mathcal{O}(n)$ , trebuie modificate pentru a găsi două numere lipsă, ci nu

doar unul singur, iar complexitățile computaționale vor rămâne aceleași. A patra variantă de rezolvare, care utilizează un spațiu de memorie de ordinul  $\mathcal{O}(1)$  și calculează numărul lipsă ca diferența dintre suma tuturor numerelor de la 1 la  $n$  și suma numerelor citite, trebuie modificată pentru a calcula nu numai suma celor două numere lipsă, ci și produsul lor. Din acest motiv, chiar și în limbajul Python, pot să apară erori din cauza valorilor foarte mari implicate în calcule!

În continuare, vom prezenta pașii unui algoritm eficient pentru rezolvarea acestei probleme și în care nu pot să apară erorile menționate anterior. Vom considera  $n = 10$  și cele  $n-2 = 8$  numere naturale citite memorate într-o listă  $v = [3, 7, 1, 10, 5, 8, 4, 9]$ , deci numerele lipsă sunt  $x = 2$  și  $y = 6$ .

### Pasul 1:

Calculăm valoarea  $x \wedge y$  în variabila `x_xor_y`, aplicând operatorul  $\wedge$  între numerele din lista  $v$  și toate numerele naturale cuprinse între 1 și  $n = 10$ , deci vom obține  $x\_xor\_y = x \wedge y = 2 \wedge 6 = 0b010 \wedge 0b110 = 0b100 = 4$ .

### Pasul 2:

Deoarece  $x \neq y$ , rezultă că  $x\_xor\_y \neq 0$ , deci în reprezentarea binară a valorii  $x\_xor\_y$  există cel puțin un bit nenul, aflat pe o poziție  $p$ . În exemplul nostru,  $x\_xor\_y = 0b100$ , deci poziția respectivă este  $p = 2$  (biții dintr-o reprezentare binară se numără începând cu 0, de la dreapta spre stânga).

### Pasul 3:

Deoarece bitul aflat pe poziția  $p$  în  $x\_xor\_y$  este egal cu 1, înseamnă că biții aflați pe poziția  $p$  în  $x$  și  $y$  sunt diferiți. În exemplul nostru, bitul aflat pe poziția  $p = 2$  în  $x = 2$  este egal cu 0, iar cel aflat pe poziția  $p = 2$  în  $y = 6$  este egal cu 1. Astfel, putem împărți toate numerele cuprinse între 1 și  $n$  în două grupe disjuncte, în funcție de valoarea bitului de pe poziția  $p$ , iar cele două numere lipsă  $x$  și  $y$  se vor găsi în grupe diferite. În tabelul de mai jos, puteți observa modul în care se împart numerele de la 1 la 10 în cele două grupe menționate, precum și faptul că  $x$  face parte din **grupa albastră** (bitul aflat pe poziția  $p = 2$  este 0), iar  $y$  face parte din **grupa roșie** (bitul aflat pe poziția  $p = 2$  este 1):

|          | Număr | Reprezentare binară |
|----------|-------|---------------------|
| <b>x</b> | 1     | 0001                |
|          | 2     | 0010                |
|          | 3     | 0011                |
|          | 4     | 0100                |
|          | 5     | 0101                |
| <b>y</b> | 6     | 0110                |
|          | 7     | 0111                |
|          | 8     | 1000                |
|          | 9     | 1001                |
|          | 10    | 1010                |

Utilizând această observație, putem să calculăm cele două numere lipsă  $x$  și  $y$ , aplicând operatorul  $\wedge$  între toate numerele cuprinse între 1 și  $n$  care fac parte din grupa sa și toate



numerele din lista  $v$  care fac parte din aceeași grupă. În exemplul dat, vom calcula valoarea  $x = (1 \wedge 2 \wedge 3 \wedge 8 \wedge 9 \wedge 10) \wedge (3 \wedge 1 \wedge 8 \wedge 10 \wedge 9) = 2$ , unde am utilizat prima pereche de paranteze doar pentru a pune în evidență numerele cuprinse între 1 și  $n$  care fac parte din grupa albastră, iar a doua pereche de paranteze am utilizat-o pentru numerele din lista  $v$  care fac parte din aceeași grupă.

#### Pasul 4:

Pentru a determina cel de-al doilea număr lipsă  $y$ , putem să procedăm la fel ca mai sus pentru grupa roșie sau, mai simplu, putem să-l calculăm cu expresia  $y = x\_xor\_y \wedge x$ . În exemplul dat, obținem  $y = x\_xor\_y \wedge x = 4 \wedge 2 = 6$ .

Înainte de a implementa acest algoritm în limbajul Python, vom prezenta o metodă eficientă de a testa din ce grupă face parte unui număr natural nenul  $t$ , în funcție de cel mai din dreapta bit egal cu 1 din reprezentarea binară a valorii  $x\_xor\_y$  (în cadrul algoritmului se poate utiliza orice alt bit nenul din reprezentarea binară a valorii  $x\_xor\_y$ , dar bitul cel mai din dreapta cu această proprietate se poate determina cel mai rapid). Practic, mai întâi vom crea o mască binară  $mb$  care va avea toți biții egali cu 0, mai puțin bitul de pe poziția  $p$  corespunzătoare celui mai din dreapta bit nenul din  $x\_xor\_y$ , astfel (am considerat  $x\_xor\_y = 424 = 0b110101000$ , deci  $p = 3$ ):

|   |  |
|---|--|
| <pre> x_xor_y = 0b110101000 x_xor_y - 1 = 0b110100111 </pre>    | <p>toți biții nuli din dreapta poziției <math>p = 3</math> și bitul nenul de pe poziția <math>p = 3</math> își comută valorile, iar restul biților nu se modifică</p>  |
| <pre> ~(x_xor_y - 1) = 0b001011000 </pre>                       | <p>toți biții nenuli din dreapta poziției <math>p = 3</math> și bitul nul de pe poziția <math>p = 3</math> își comută valorile (deci revin la valorile inițiale din <math>x\_xor\_y</math>), iar restul biților își comută valorile (deci au valori complementare celor inițiale din <math>x\_xor\_y</math>)</p> |
| <pre> mb = x_xor_y &amp; ~(x_xor_y - 1) mb = 0b000001000 </pre> | <p>calculăm masca binară <math>mb</math></p>   |

Folosind masca binară  $mb$  putem testa foarte ușor dacă un număr natural  $t$  face parte din prima grupă, respectiv verificăm dacă  $mb \& t == 0$  (pentru a testa dacă  $t$  face parte din a doua grupă folosim condiția inversă  $mb \& t != 0$ ).

Implementarea acestui algoritm în limbajul Python este următoarea:

```

n = int(input("n = "))

v = []
x_xor_y = 0
for i in range(1, n-1):
    t = int(input("v[" + str(i) + "] = "))
    v.append(t)
    x_xor_y = x_xor_y ^ t

```

```

for i in range(1, n+1):
    x_xor_y = x_xor_y ^ i

mb = x_xor_y & ~(x_xor_y - 1)

x = 0
for t in v:
    if t & mb == 0:
        x = x ^ t

for i in range(1, n+1):
    if i & mb == 0:
        x = x ^ i

y = x_xor_y ^ x

print("Numerele lipsă: ", x, " și ", y)

```

Complexitate computațională a acestui algoritm este  $\mathcal{O}(n)$ , iar spațiul de memorie utilizat este tot  $\mathcal{O}(n)$ .

## Probleme propuse

1. Se citește un șir format din numere naturale cu proprietatea că fiecare valoare distinctă din șir apare de un număr par de ori, mai puțin una care apare de un număr impar de ori. Să se afișeze valoarea care apare de un număr impar de ori în șirul dat.

2. Să se calculeze numărul  $x$  obținut prin aplicarea operatorului XOR între toate elementele tuturor submulțimilor mulțimii  $A = \{a_1, a_2, \dots, a_n\} \subset \mathbb{N}$ , mai puțin mulțimea vidă. De exemplu, dacă  $A = \{2, 7, 18\}$  vom obține numărul  $x = (2) \wedge (7) \wedge (18) \wedge (2 \wedge 7) \wedge (2 \wedge 18) \wedge (7 \wedge 18) \wedge (2 \wedge 7 \wedge 18) = 0$  (am folosit parantezele doar pentru a pune în evidență elementele submulțimilor lui  $A$ ).

3. Fie  $x$  și  $y$  două numere naturale nenule. Calculați numărul biților din reprezentarea binară a numărului  $x$  a căror valoare trebuie comutată pentru a obține numărul  $y$ .

**Indicație de rezolvare:** Se calculează numărul biților nenuli din numărul  $t = x \wedge y$ .

4. Fie  $n$  un număr natural. Să se determine cea mai mică putere a lui 2 mai mare sau egală decât numărul  $n$ .

**Indicație de rezolvare:** <https://www.geeksforgeeks.org/smallest-power-of-2-greater-than-or-equal-to-n/>

5. Să se determine în mod eficient numărul de biți nuli din reprezentarea binară a unui număr natural nenul. De exemplu, reprezentarea binară a numărului 600 este 0b1001011000 și conține 6 biți nuli.

**Indicație de rezolvare:**

Această problema poate fi redusă la problema rezolvată 4 prin comutarea fiecărui bit din reprezentarea binară a numărului natural nenul  $n$  dat (i.e., orice bit egal cu 0 devine 1 și orice bit egal cu 1 devine 0) și determinarea numărului de biți nenuli din noul număr obținut. Pentru a comuta toți biții numărului  $n$  vom aplica operatorul  $\wedge$  între numărul dat  $n$  și o mască binară  $mb$  având toți biții egali cu 1, așa cum se poate observa din exemplul de mai jos pentru  $n = 600$ :

```

n = 0b1001011000
mb = 0b1111111111
n ^ mb = 0b0110100111

```

Știind că lungimea reprezentării binare a numărului  $n$  este  $k = 1 + \lceil \log_2 n \rceil$ , se observă ușor faptul că masca binară  $mb$  este egală cu  $(1 \ll k) - 1$ .