1. **Se dă un număr natural x. Să se calculeze x div 4 și x*6 folosind operatori pe biți.**

Soluție:

Amintim că înmulțirea lui n cu 2^k se poate face folosind operatorul de shiftare la stânga (cu k poziții). Programul Python este următorul:

```
n = int(input("n="))
print(n,'div 4 =',n>>2) #4=2**2
print(n,'* 6 =',(n<<2)+(n<<1)) #n*6=n*2**2+n*2
```

2. **Se dă un număr natural n. Să se determine dacă acesta este par sau impar folosind operatori pe biți.**

Soluție:

Un număr n este par dacă și numai dacă ultimul bit din reprezentarea sa binară este 0. Programul Python este următorul:

```
n = int(input("n="))
if n&1 == 0:  #testam ultimul bit
    print("par")
else:
    print("impar")
```

3. **Se dau 2 numere naturale x și k. Să se afișeze al k-lea bit din dreapta din reprezentarea binară a lui x.**

Soluție:

```
n=int(input("n="))
k=int(input("k="))
print( (n >> (k-1)) & 1)
```

4. **Se dau 2 numere întregi x și k. Să se afișeze numărul obținut din x astfel:**
   - **se setează bitul numărul k la valoarea 1**
   - **se setează bitul k la valoarea 0**
   - **se complementează bitul k**

Soluție:

Pentru a accesa biți din reprezentarea binară a lui x putem folosi operatori pe biți și numere alese m (numite măști) astfel încât prin operații de tipul **x&m**, **x^m**, **x|m** să păstrăm din **x** doar biții care ne interesează.

   a) **se setează bitul numărul k la valoarea 1**

x	= 0b	$a_1 \dots a_{n-k}$	$a_{n-(k-1)}$	$a_{n-(k-2)}$	…	a_n
m	= 0b	0. …. 0	1	0	…	0
x|m	= 0b	$a_1 \dots a_{n-k}$	1	$a_{n-(k-2)}$	…	a_n

Avem m = 0b0010…0 = 2^{k-1} = (1<< (k-1)), deci soluția la a) este x = x | (1 << (k-1)).

```
x=0b1001
k=2
x = x | (1 << (k-1))
print(bin(x))
```

b) se setează bitul k la valoarea 0

$$
\begin{array}{rcl}
x & = & 0b\ a_1 \dots a_{n-k}\ \ a_{n-(k-1)}\ \ a_{n-(k-2)} \dots a_n \\
m & = & 0b\ 1. \dots 1\ \ \ 0\ \ \ 1\ \ \dots 1 \\
\hline
\color{red}{x\&m} & = & 0b\ a_1 \dots a_{n-k}\ \ \ 0\ \ \ a_{n-(k-2)} \dots a_n
\end{array}
$$

Avem m = ~2^{k-1} =~(1<< (k-1)) deci soluția la b) este x = x & (~ (1 << (k-1)))

```
x=0b1011
k=2
x = x & (~ (1 << (k-1)))
print(bin(x))
```

c) se complementează bitul k

$$
\begin{array}{rcl}
x & = & a_1 \dots a_{n-k}\ \ a_{n-(k-1)}\ \ a_{n-(k-2)} \dots a_n \\
m & = & 0 \dots 0\ \ \ 1\ \ \ 0\ \ \dots 0 \\
\hline
\color{red}{x\wedge m} & = & a_1 \dots a_{n-k}\ \ \mathbf{\sim a_{n-(k-1)}}\ \ a_{n-(k-2)} \dots a_n
\end{array}
$$

Avem m = 2^{k-1} = (1<< (k-1)) deci soluția la c) este x= x ^ (1 << (k-1)).

```
x=0b1001 # x=0b1011
k=2
x = x ^ (1 << (k-1))
print(bin(x))
```

5. Să se verifice dacă un număr natural pozitiv n este de forma 2^k. În caz afirmativ să se afișeze valoarea k (folosind operatori pe biți).

Soluție:

Un număr natural n este de forma 2^k dacă reprezentarea sa binară conține un singur bit nenul (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 îl obține pe n, de exemplu 0b1001 = 2^3+2^0 = 9).

Varianta 1 – Tăiem zerourile de la finalul reprezentării în baza 2 a lui n (bit cu bit, folosind shiftare la dreapta). Dacă numărul rămas este egal cu 1, atunci n era o putere a lui 2, altfel în reprezentarea lui n în baza 2 sunt mai mulți biți egali cu 1, deci nu este o putere a lui 2. Algoritmul are complexitate liniară în numărul de biți din reprezentarea binară a lui n (care este egal cu cu 1 + [$\log_2 n$]), deci $\mathcal{O}(\log_2 n)$. Programul Python este următorul:

```
n = int(input("n = "))
aux = n
k = 0
while (aux>0) and (aux & 1 == 0): #cat timp ultimul bit este 0
    aux = aux >> 1
```

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

Varianta 2 – se bazează pe următoarea observație: scăzând 1 din n, toți biții de la sfârșit devin 0, iar cel mai din dreapta bit egal cu 1 din n devine 0 (fie k poziția lui), iar restul biților rămân neschimbați. În reprezentarea binară a lui n&(n-1) toți biții începând cu poziția k devin 0, iar ceilalți sunt identici cu cei din n (și din n-1):

$$
\begin{array}{lcl}
\mathbf{n} & = & a_1 \ldots a_{k-1}\ 1\ 0\ 0\ \ldots\ 0_{(2)} \\
\mathbf{n - 1} & = & a_1 \ldots a_{k-1}\ 0\ 1\ 1\ \ldots\ 1_{(2)} \\
\mathbf{n\ \&\ (n - 1)} & = & a_1 \ldots a_{k-1}\ 0\ 0\ 0\ \ldots\ 0_{(2)}
\end{array}
$$

$\mathbf{n\ \&\ (n - 1) = 0} \Leftrightarrow \mathbf{n = 1\ 0\ 0\ \ldots\ 0_{(2)}} \Leftrightarrow \mathbf{n}$ este putere a lui 2

Astfel, pentru a testa dacă n este o putere a lui 2 este suficient să verificăm dacă n&(n-1) este egal cu 0 (n nu mai are și alți biți egali cu 1 în afară de cel de pe poziția p, altfel aceștia s-ar fi păstrat și în n&(n-1)).

Dacă știm că n este de forma 2^k putem determina k folosind funcția logaritm (care poate fi considerată având complexitate O(1)). Programul Python este următorul:

```
import math
n = int(input("n = "))
if (n>0) and (n & (n-1) == 0):
    print(n, "=2**", int(math.log2(n)), sep="")
else:
    print(n, "nu este o putere a lui 2")
```

6. Să se interschimbe valoarea a două numere întregi x, y fără a folosi alte variabile (se cer două soluții: folosind operatori aritmetici și folosind numai operatori pe biți).

Soluție:

Folosim relația x^x=0, de unde rezultă x^y^y = x.
Cu ajutorul unei variabile auxiliare putem face interschimbarea astfel:

```
aux = x ^ y
y = aux ^ y ⇒ y devine x
x = aux ^ x
```

Putem însă renunța la aux, folosindu-l în locul lui chiar pe x

```
x = x ^ y
y = x ^ y ⇒ y devine vechiul x
x = x ^ y
```

(valoarea lui x s-a modificat la prima atribuire, dar o regăsim acum în y, de aceea nu este nevoie de aux)
Avem atunci soluția:

```
#cu operatori biti
x = int(input("x="))
y = int(input("y="))
x = x^y
y = x^y # y = (x ^ y) ^ y = x ^ (y ^ y) = x ^ 0 = x
```

```python
x = x^y # x = (x ^ y) ^ x = x ^ (y ^ x) = (x ^ x) ^ y) = 0 ^ y = y
print(x,y)
```

Corectitudinea se poate demonstra și folosind observația un bit de pe poziția k din x ^ y este 1 ⇔ biții de pe poziția k din x și y sunt diferiți.

Folosind operatori aritmetici interschimbarea se poate face astfel:

`# cu operatori aritmetici`
`x = int(input("x="))`
`y = int(input("y="))`
`x=x+y`
`y=x-y`
`x=x-y`
`print(x,y)`
	`# cu operatori aritmetici - alte variante`
	`x=x*y`
	`y=x//y`
	`x=x//y`
	`print(x,y)`
	`# cu operatori aritmetici - alte variante`
	`x=x-y`
	`y=x+y`
	`x=y-x`
	`print(x,y)`

7. Se citește un șir format din 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.

Soluție:

Fie $x_1, \ldots, x_n$ numerele din șir. Fie $v = x_1 \wedge \ldots \wedge x_n$.
Deoarece ^ este asociativ și comutativ și x ^ x = 0, xor pentru valorile care apar de 2 ori este 0, deci v este egal cu numărul care apare o singură dată.
Programul Python este următorul:

`n = int(input("n (impar) ="))`
`print("sirul:")`
`v = 0`
`while n>0:`
`    x = int(input())`
`    v = v ^ x`
`    n = n - 1`
`print(v)`
	`n = int(input("n (impar) ="))`
	`print("sirul:")`
	`v = 0`
	`for i in range(n):`
	`    x = int(input())`
	`    v = v ^ x`
	`print(v)`

Complexitatea algoritmului este O(n).

TEMĂ (la laborator):

8. Scrieți un program care determină în mod eficient numărul de biți egali cu 1 din reprezentarea binară a unui număr natural n citit de la tastatură.

Soluție:

a)

Varianta 1: Putem interoga pe rând fiecare bit din reprezentare binară a lui n astfel: testăm ultimul bit din n, care este **`n&1`**, apoi îl ștergem (**`n=n>>1`**) și reluăm. Complexitatea algoritmului va fi liniară în lungimea reprezentării binare a lui n, deci O(log2(n)).

Programul Python este următorul:

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

Varianta 2: Pornim de la observația care rezultă din soluția de la problema 4: **`n&(n-1)`** are în reprezentare cu un bit nenul mai puțin decât **`n`** (v. pb 4)

```
  n           =  a1... ak-1 1 0 0 ... 0(2)
n - 1         =  a1... ak-1 0 1 1 ... 1(2)
n & (n - 1) =  a1... ak-1 0 0 0 ... 0(2)
```

Atunci, repetând atribuirea **`n = n & (n-1)`** până când n devine 0 și numărând câte atribuiri de acest tip am făcut obținem numărul de biți nenului din reprezentarea lui n.
Problema este de fapt o generalizare a problemei 4, deoarece a testa ca un număr este putere a lui 2 este echivalent cu a testa că numărul de biți nenuli din reprezentarea lui este 1.
Programul Python este următorul:

```
n = int(input("n = "))
print("Reprezentarea binară a lui ", n, "este: ", bin(n))
aux = n
k = 0
while aux != 0:
    k = k + 1
    aux = aux & (aux - 1)

print(k, "biți nenuli")
```

Complexitatea acestui algoritmul este dată de numărul k de biți nenuli din reprezentarea binară a lui n, deci complexitatea maximă va fi tot $\mathcal{O}(\log 2\ n)$. Totuși, pentru majoritatea valorilor numărului n, această variantă va fi mai eficientă decât varianta 1.

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

Soluție:

Varianta 1 – Putem interoga pe rând fiecare bit din reprezentare binară a lui n astfel: testăm ultimul bit din n, care este n&1, apoi îl ștergem (n=n>>1) și reluăm. Memorăm într-o variabilă lungimea secvenței curente de 1 și o actualizăm (o creștem sau reinițializăm cu 0) în funcție de valoarea noului bit interogat.
Programul Python este următorul:

```
n = int(input("n="))
# n = int(input("n="), 2) #putem da numarul la intrare in baza 2
aux = n
l_max = 0
l_curent = 0
while aux!=0:
    if aux&1 == 1:
```

```
        l_curent += 1
        if l_curent>l_max:
            l_max = l_curent
    else:
        l_curent = 0
    aux = aux>>1
print(l_max)
```

Complexitatea algoritmului va fi liniară în lungimea reprezentării binare a lui n, deci O(log2(n)).

Varianta 2. Pentru a calcula în mod eficient lungimea maximă a unei secvențe de biți egali cu 1 din reprezentarea binară a unui număr natural n putem folosi următoarea idee.

Dacă deplasăm cu o poziție spre stânga biții din reprezentarea lui n toate secvențele de biți egali cu 1 se vor deplasa cu o poziție spre stânga, deci primului bit de la dreapta la stânga din fiecare secvență de biți nenuli din n << 1 îi va corespunde unui bit egal cu 0 în reprezentarea binară a numărului n.

Aplicând operatorul & între n și n<<1 vom elimina cel mai din stânga bit al fiecărei secvențe de biți nenuli din reprezentarea binară a lui n, deci lungimile tuturor secvențelor de biți nenuli vor scădea cu 1.

```
           n          = 0b1011110011100110
       n << 1         = 0b0111100111001100
   n & (n << 1)      = 0b0011100011000100
```

Repetând atribuirea n = n & (n << 1) până când n devine 0 și numărând iterațiile vom obține lungimea maximă a unei secvențe de biți nenuli din reprezentarea binară a lui n.

Programul Python este următorul:

```
n = int(input("n=")) #n = int(input("n="), 2)
aux = n
l_max = 0
while aux!=0:
    aux = aux & (aux<<1)
    l_max = l_max+1
print(l_max)
```

Complexitatea acestui algoritmul este dată de lungimea maximă a unei maximă a unei secvențe de biți egali cu 1 din reprezentarea binară a lui n (lungime pe care nu o putem calcula în funcție de n) deci complexitatea maximă va fi dată de lungimea reprezentării, deci tot $\mathcal{O}(\log_2 n)$. Totuși, pentru majoritatea valorilor numărului n, această variantă va fi mai eficientă decât varianta 1.