TASK 1

Write code that allows to use individuals as well as the three functions `OneMax`, `LeadingOnes`, and `Jump_k` . For individuals, do not use libraries but implement a data type that fully utilizes the memory. That is, do not store each bit value of an individual in a byte but in an actual bit.

Individual data type

An individual is a potential solution $x = (x_1, ..., x_n) \in \{0, 1\}^n$. We wish to implement a data type that stores each $x_i \in \{0, 1\}$ using a single bit. Since the smallest addressable unit of memory is a byte, this means that we cannot represent each x_i using a basic type such as `bool`, because a `bool` uses 1 byte of memory.

However, we can use a basic type such as `int` to represent multiple x_i at once. The `int` data type has 32 bits, so we can represent 32 x_i's using a single integer. Using an array of integers, we can represent all x_i's. If we suppose that $n \in \mathbb{N}$ is a multiple of 32, then this representation will fully utilize memory.

The following procedures are implemented in `Individual.nim` :

- `void newIndividual(int size)` which creates a new individual and allocates the necessary memory to represent all bits.
- `int get(int idx)` which returns the value of the bit at the index `idx`.
- `void set(int idx)` which updates the value of the bit at the index `idx` to 1. If the value is already 1, no change occurs.
- `void reset(int idx)` which updates the value of the bit at the index `idx` to 0. If the value is already 0, no change occurs.
- `void flip(int idx)` which flips the value of the bit at the index `idx`.
- `int count()` which counts the number of bits equal to 1.

Code (implemented in Nim)

- Individual.nim

```
# Individual data type

type
    Individual* = ref object
        #[ represents candidate solutions x = (x1, ..., xn) ]#
        size*: int
        bits*: seq[int]
```

```
8
9  proc newIndividual(size: int): Individual =
10     #[ constructor for new Individual ]#
11
12     #number of integers necessary to represent all xi's
13     let necessaryIntegers = (size + 31) div 32
14
15     result = new(Individual)
16     result.size = size
17     result.bits = newSeq[int](necessaryIntegers)
18
19     return result
20
21 proc get*(individual: Individual, idx: int): int =
22     #[ get bit at index idx ]#
23     let testBit = individual.bits[idx div 32] and (1 shl (idx mod 32))
24
25     if testBit > 0:
26         return 1
27     else:
28         return 0
29
30 proc set*(individual: Individual, idx: int): void =
31     #[ set bit at index idx to 1 ]#
32     individual.bits[idx div 32] = individual.bits[idx div 32] or (1 shl (idx
       mod 32))
33     return
34
35 proc reset*(individual: Individual, idx: int): void =
36     #[ set bit at index idx to 0 ]#
37     individual.bits[idx div 32] = individual.bits[idx div 32] and not (1 shl
       (idx mod 32))
38     return
39
40 proc flip*(individual: Individual, idx: int): void =
41     #[ flip bit at index idx ]#
42     let bitI = get(individual, idx)
43     if bitI == 0:
44         set(individual, idx)
45     else:
46         reset(individual, idx)
47     return
48
49 proc count*(individual: Individual): int =
50     #[ count number of bits equal to 1 ]#
51     result = 0
52     for i in countup(1, individual.size):
53         let bitI = get(individual, i)
54         result += bitI
55     return result
```

- BenchmarkFunctions.nim

```nim
import Individual

# Benchmark functions

proc OneMax(individual: Individual): int =
    #[ returns the number of 1s of the input ]#
    return count(individual)

proc LeadingOnes(individual: Individual): int =
    #[ returns the length of the longest consecutive prefix of 1s ]#
    let n = individual.size
    result = 0
    for i in countup(1, n):
        var prefixProduct = 1
        for j in countup(1, i):
            prefixProduct = prefixProduct*get(individual, j)
        result += prefixProduct
    return result

const k = 50 #we set k value as a constant
proc JumpK(individual: Individual): int =
    #[ analog to OneMax but penalises individuals with number of ones in n-k
   +1,...,n-1 ]#
    let n = individual.size
    let OneMax_x = OneMax(individual)
    if OneMax_x <= n - k or OneMax_x == n:
        return k + OneMax_x
    return n - OneMax_x
```