Author: Hanes Camelia-Andreea Group 914

**Problem statement**

Optional homework – implementation operations + conversions

● share in the final grade: 10%

● deadline for homework submission: the second week of December 2015

The application must implement algorithms for:

○ arithmetic operations: addition, subtraction, multiplication and division by one

digit, in a base p from the set {2,3,...,9,10,16}

○ conversions of natural numbers between two bases p,q from {2,3,...,9,10,16}

using the substitution method or successive divisions and rapid conversions

between two bases p,q{2, 4, 8, 16}.

and must have a menu such that all operations and conversion methods to be verified

separately.

The mark is computed as follows:

10%: by default

70% : the application (the author name will be found in code and will be visible at run too)

1p ­ algorithm for the method of successive divisions

1p ­ algorithm for the substitution method

1p – algorithm for conversion using 10 as an intermediate base

2p ­ rapid conversions (executable form) between two bases p,q from {2, 4, 8, 16}.

1p addition of two numbers in a base

1p subtraction of two numbers in a base

1p multiplication of a number by a digit in a base

1p division of a number by a digit in a base

1p code quality (indentation, use of comments, suggestive variables names)

20%: documentation

1p problem statement

1p sub­algorithm’s diagram

1p used data type specification

3p specification and pseudo­code for the important algorithms used (input, output,

preconditions, post­conditions ­1p; pseudo­code 2p)

3p at least a set of test data for the complete application, more data sets where is needed

1p documentation clearness (structured, well written, ...)

**Numerical Converter and Calculator**

**Features:**

**Feature 1: Add two numbers of the same numerical base**

**Feature 2: Subtract two numbers of the same numerical base**

**Feature 3: Multiply two numbers represented in the same base**

**Feature 4: Divide a number by a small number (represented on one allowed digit)**

**Feature 5: Successive divisions algorithm**

**Feature 6: Substitution Method algorithm**

**Feature 7: Conversion using base 10 as intermediate base algorithm**

**Feature 8: Rapid conversions between bases {2, 4, 8, 16} algorithm**

**Feature 9: User interface provided**

**Feature 10: Tests and specifications**

**Implementation plan:**

1. **IntegerNumber entity class to represent the integer number**
2. **Functions to define all operations (addition between integers, subtraction, multiplication and division by one digit)**
3. **Functions to implement the algorithms for conversions**
4. **Console menu-based user interface**

The project is compounded by four Python modules described below:

1. The coordinator of the application: application\_manager.py

This module starts and runs the application, instantiates two entities:

1. The test entity, to run default and random data input (tests the application)
2. The ActionCalculator entity which contains the run() function
3. Console\_ui module:

It is a class to coordinate the interaction between the user and the program

It contains a list of commands allowed for the user interaction

1. Test module, which unit tests the operations and conversions; it contains a list of allowed bases
2. IntegerNumber class, representing the entity containing all data necessary for performing the operatons and conversions on an integer number

Properties:

1. \_numberBase – the base of the number
2. \_numberLength – the number of digits the number has
3. \_digits – a list containing the digits in reverse order compared to the usual structure of a number
4. Exception class (exception.py) representing the exception occurred which we catch in the program running process

(Every module and every function has its own detailed specifications)

**Implementation plan**

**=Item 1=**

**IntegerNumber entity**

class **IntegerNumber**():

*'''*

*Class to represent an integer number*

*Fields:*

*self.\_numberBase : the base of the integer number*

*self.\_numberLength : the number of digits the number has*

*self.\_digits : the list containing the digits of the integer, with the*

*precision that the least significant digits of the number are placed*

*at the lower index in the array of digits*

*'''*

def **\_\_init\_\_**(*self*, numberBase, numberRepr):

*'''*

*This is the contructor for the IntegerNumber class*

**:parameter** *numberBase: an integer number - the base of the number's representation*

**:parameter** *numberRepr: a string - the number represented in the base above described*

**:raises** *IntegerNumberException if the base doesn't belong to the list of allowed numerical bases*

*'''*

if numberBase not in IntegerNumber.AllowedNumericalBases:

raise IntegerNumberException(*"Base not allowed."*)

if numberRepr == *""*:

numberRepr = *"0"*

*self*.\_numberBase = numberBase

*self*.\_numberLength = 0

*self*.\_digits = []

The constructor is used when we create a new IntegerNumber object which has the properties described above. In the initializing function we also have the validator, where the number is parsed and each of its digits and base are checked for obeying the necessary mathematical conditions in order to be part of a number in the base given as parameter. The number class is created if and only if it’s attributes had passed the checking process.

**=Item 2=**

**Provided functions to define all required operations**

**Addition – performed in the IntegerNumber class**

def **\_\_add\_\_**(*self*, nextOperand):

*""""*

*This function represents the addition between two integer number objects*

**:param** *nextOperand representing the second operand for the addition process*

*while the first one is represented by self*

**:return** *a new integer number representing the sum between self and the*

*nextOperand*

**:raises** *ValueError if the second operand is not of the type integerNumber*

*or it is represented in other base, different from self*

*"""*

if not isinstance(nextOperand, IntegerNumber):

raise ValueError(*"Addition between different objects, can't perform."*)

if *self*.\_numberBase != nextOperand.\_numberBase:

raise ValueError(*"Impossible addition between numbers in different bases."*)

operatingBase = *self*.getNumericalBase()

newRepresentation = IntegerNumber(operatingBase, repr(*self*))

if len(newRepresentation) < len(nextOperand):

for i in range(0, len(nextOperand) - len(newRepresentation)):

newRepresentation.appendToRepresentation(0)

else:

for i in range(0, len(newRepresentation) - len(nextOperand)):

nextOperand.appendToRepresentation(0)

carryDigit = 0

for i in range(0, newRepresentation.\_numberLength):

value = newRepresentation[i] + nextOperand[i] + carryDigit

newRepresentation[i] = value % operatingBase

carryDigit = value // operatingBase

if carryDigit:

newRepresentation.appendToRepresentation(carryDigit)

return newRepresentation

The algorithm parses the numbers and add them as we do „by hand”, after making sure that the input numbers have the same number of digits by appending auxiliary zero values where it is necessary. If there is extra transport, the parameter defined in the function body assures to append the carry element to the result.

**Subtraction**

def **\_\_sub\_\_**(*self*, nextOperand):

*"""*

*This function implements the subtraction between self and a new operand*

*given as parameter with the name nextOperand, with the precision*

*that the parameter nextOperand is the subtrahend and self is the minuent*

**:return** *an IntegerNumber representing the difference between the numbers*

*described above*

**:raise** *ValueError if the operand given as parameter is not an instance*

*of integer numbers or if the base of this operand is different from the*

*one self has in its representation*

*"""*

if not isinstance(nextOperand, IntegerNumber):

raise ValueError(*"Can't subtract two different objects."*)

if *self*.getNumericalBase() != nextOperand.getNumericalBase():

raise ValueError(*"Can't subtract two numbers with different base representation."*)

newRepresentation = IntegerNumber(*self*.\_numberBase, repr(*self*))

#nextOperand = IntegerNumber(self.getNumericalBase(), repr(nextOperand))

carryDigit = 0

for i in range(len(newRepresentation) - len(nextOperand)):

nextOperand.appendToRepresentation(0)

for i in range(0, len(*self*)):

newRepresentation[i] = newRepresentation[i] - (nextOperand[i] + carryDigit)

if newRepresentation[i] < 0:

carryDigit = 1

else:

carryDigit = 0

if carryDigit:

newRepresentation[i] += newRepresentation.getNumericalBase()

newRepresentation.removeLeadingZeros()

return newRepresentation

The subtraction algorithm has the same structure as the addition algorithm with the precision that here the transport digit is used for „renting digits” as we usually perform by hand these operations, having in account both the numerical values and the bases we are working with and performing the opration accordingly. The algorithm called in the last line before returning the value makes sure we do not have leading zeroes as the first number digits after performing the operation.

**Multiplication**

def **\_\_mul\_\_**(*self*, nextOperand):

*"""*

*This function implements the multiplication between self and the other*

*operand represented as input parameter wth the nextOperand name*

**:return** *an IntegerNumber representing the number resulted from the*

*multiplicaton above described*

**:raises** *ValueError if the nextOperand is not of the type IntegerNumber*

*or if the base doesn't coincide with the self base*

*"""*

if not isinstance(nextOperand, IntegerNumber):

raise ValueError(*"Can't multiply objects of different types."*)

if *self*.getNumericalBase() != nextOperand.getNumericalBase():

raise ValueError(*"Can't multiply numbers with different base representations."*)

newRepresentation = IntegerNumber(*self*.\_numberBase, *"0"*\*(len(*self*) + len(nextOperand) -1))

#nextOperand = IntegerNumber(self.\_numberBase, repr(nextOperand))

carryDigit = 0

for i in range(0, len(*self*)):

for j in range(0, len(nextOperand)):

newRepresentation[i+j] += *self*[i] \* nextOperand[j]

for i in range(0, len(newRepresentation)):

newRepresentation[i] += carryDigit

carryDigit = newRepresentation[i] // *self*.getNumericalBase()

newRepresentation[i] = newRepresentation[i] % *self*.getNumericalBase()

while carryDigit > 0:

newRepresentation.appendToRepresentation(carryDigit%*self*.\_numberBase)

carryDigit = carryDigit // *self*.\_numberBase

return newRepresentation

The multiplication algorithm is implemented also similarly, providing the same attention to the base and to detail we used to maintain.

**Division**

**The implementation of division algorithm needed more precision for the implementation, having that the result provides a quotient and a remainder, presented below as separate module functions**

**Function to return the quotient**

def **\_\_floordiv\_\_**(*self*, nextOperand):

*"""*

*Function which implements the division of an object of the IntegerNumber*

*type and one digit integerNumber object*

*nextOperand is the divisor*

**:return** *an IntegerNumber representing the quotient of the division*

**:raises** *valueerror if nextOperand is not a ne-digit number*

*"""*

if not isinstance(nextOperand, int):

raise ValueError(*"Division is only permitted with a one digit integer as divisor."*)

auxiliary = 0

newRepresentation = IntegerNumber(*self*.getNumericalBase(), repr(*self*))

for i in reversed(range(0, len(newRepresentation))):

auxiliary = newRepresentation.getNumericalBase() \* auxiliary + newRepresentation[i]

newRepresentation[i] = auxiliary // nextOperand

auxiliary = auxiliary % nextOperand

newRepresentation.removeLeadingZeros()

return newRepresentation

**Function to return the remainder**

def **\_\_mod\_\_**(*self*, nextOperand):

*""""*

*This function implements the division between two numbers of the type IntegerNumber*

*from which the second one fgiven as parameter is a one-digit-type number*

**:return** *an IntegerNumber representing the remainder of the diviosion*

*described above*

**:raies** *ValueError if the number given as parameter has more than one digit*

*"""*

if not isinstance(nextOperand, int):

raise ValueError(*"Division is only permitted with an one digit divisor."*)

auxiliary = 0

for i in reversed(range(0, len(*self*))):

auxiliary = (auxiliary \* *self*.getNumericalBase() + *self*[i]) % nextOperand

return auxiliary

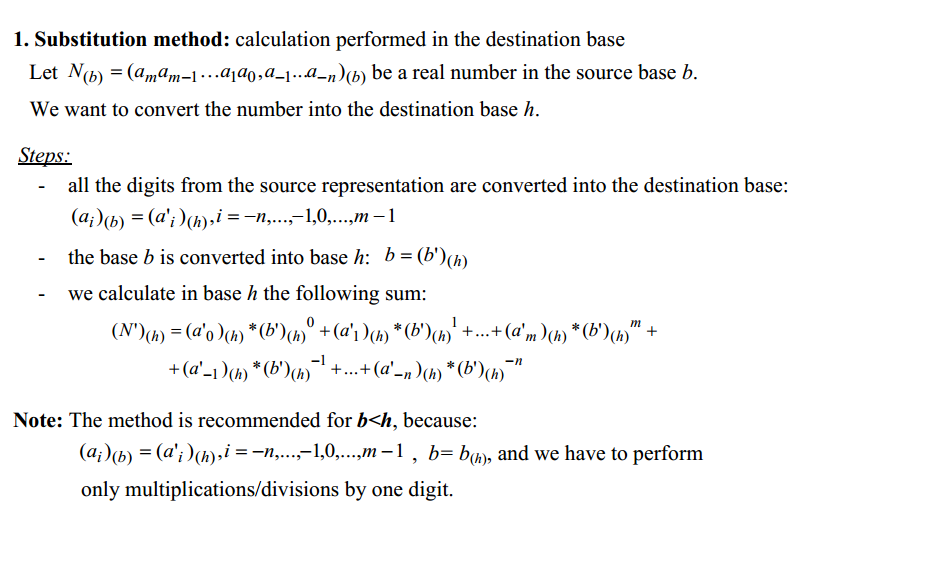
It is easy to recall that all algorthms for performing the operations are overriding the Python methods, which eases the structure of the algorithms and the complexity. Moreover, it makes the code more readable, easier to test and doesn’t need to be more specified.

**=Item 3=**

**Conversion methods algorithms**

**Substitution Method Algorithm**

First, let us remember the definition of the method and follow the steps of the algorithm accordingly.

****

def **substitutionMethodBaseConversion**(*self*, destinationBase):

*"""*

*This function implements the conversion of self in another given base as parameter*

*using the substitution base which is more often reccomended in the case*

*when the actual base is less than the base we want to convert the number to*

**:return** *an IntegerNUmber representing the number converted to the destinationBase*

*described above*

*"""*

destinationNumber = IntegerNumber(destinationBase, *"0"*)

basePwr = IntegerNumber(destinationBase, *"1"*)

for i in range(len(*self*)):

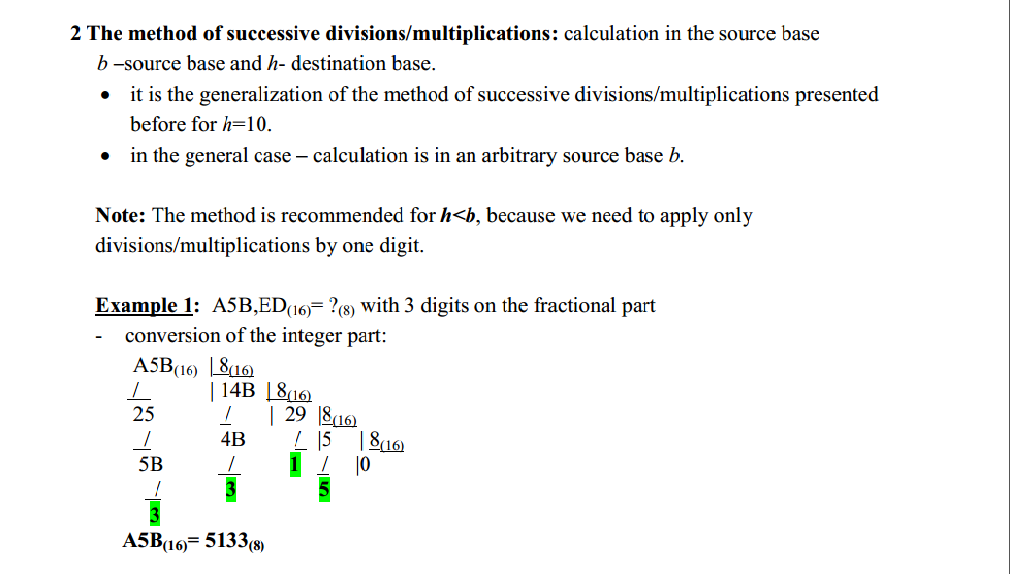
destinationNumber = destinationNumber + basePwr \* IntegerNumber(destinationBase, IntegerNumber.NumericalSymbols[*self*[i]])

basePwr = basePwr \* IntegerNumber(destinationBase, IntegerNumber.NumericalSymbols[*self*.\_numberBase])

return destinationNumber

**Successive divisions/multiplications algorithm**

(in our case we are talking only about divisions because the numbers are integers)



def **successiveDivisipnConversionMethod**(*self*, destinationBase):

*"""*

*This function implements the conversion of self to another base called as parameter*

*destinationBase and the succesive division method is used, recommended to be used*

*generally when the source base is greater than the destination base from reasons that*

*there is only need to divide by one-digit number*

**:return** *an IntegerNumber type object representing self converted to the destinaton base above described*

*"""*

destinationNumber = IntegerNumber(destinationBase, *"0"*)

basePwr = IntegerNumber(destinationBase, *"1"*)

interm = IntegerNumber(destinationBase, *"10"*)

selfie = deepcopy(*self*)

while len(selfie) != 0:

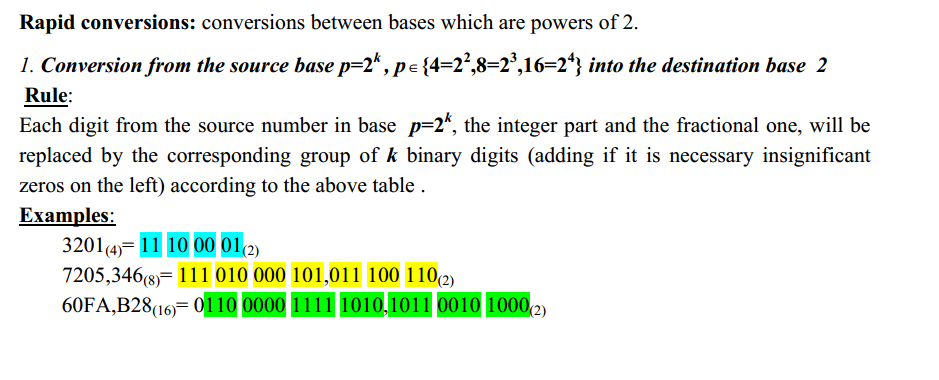
destinationNumber = destinationNumber + basePwr \* IntegerNumber(destinationBase, IntegerNumber.NumericalSymbols[selfie % destinationBase])

selfie = selfie // destinationBase

basePwr = basePwr \* interm

return destinationNumber

**Rapid conversions algorithm**

****

def **rapidConversionsMethod**(*self*, destinationBase):

auxiliary = *""*

repre = repr(*self*)

repre = repre[::-1]

if *self*.\_numberBase < destinationBase:

manipulator = int(log(destinationBase, *self*.\_numberBase))

i = 0

while i < *self*.\_numberLength:

current = repre[i:i+manipulator]

current = current[::-1]

numberDig = IntegerNumber(*self*.\_numberBase, current)

numberDig = numberDig.substitutionMethodBaseConversion(destinationBase)

auxiliary += repr(numberDig)

i += manipulator

else:

manipulator = int(log(*self*.\_numberBase, destinationBase))

for i in range(len(*self*)):

current = IntegerNumber(*self*.\_numberBase, IntegerNumber.NumericalSymbols[*self*[i]])

current = current.successiveDivisipnConversionMethod(destinationBase)

for j in range(0, len(current)):

auxiliary = auxiliary + IntegerNumber.NumericalSymbols[current[j]]

for j in range(manipulator - len(current)):

auxiliary = auxiliary + *"0"*

return IntegerNumber(destinationBase, auxiliary[::-1])

**The algorithm of conversion which uses 10 as an intermediate base**

def **ConversionIntermediateBase**(*self*, destinationBase):

number = *self*.substitutionMethodBaseConversion(10)

number = number.successiveDivisipnConversionMethod(destinationBase)

return number

This algorithm is straightforward, we can easily see the mechanism of the algorithm based on the other conversion algorithms.

**Note:** The documentation provides informations taken from the source code of the project and from the university courses of Computational Logic.