**Optional Homework – Mititean Traian Adrian**

**Implementation of operations and conversions**

**Problem Statement**

Create an application that implements algorithms for:

* Arithmetic operations for positive integers: addition, subtraction, multiplication and division by one digit, in a base p∈{2, 3, …, 9, 10, 16}
* Conversions of natural numbers between two bases p, q∈{2, 3, …, 9, 10, 16} using the substitution method or successive divisions and rapid conversions between two bases p, q∈{2, 4, 8, 16}

The application must have a menu such that all operations and conversion methods can be verified separately.

To run the project you have to run ProgramRunFile.py file

**Pseudo-code**

**Conversion Methods:**

function divide\_a\_number\_to\_a\_digit\_in\_base\_p(chosen\_number, divisor\_number, base\_number):

# Define a list of digits for reference in different bases

list\_of\_digits = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F']

# Convert the divisor\_number to its equivalent base 10 value

divisor\_number = list\_of\_digits.index(divisor\_number)

# Initialize an empty string to store the result

result = ""

# Initialize the remainder to 0

remainder = 0

# Iterate through each digit in the chosen\_number

for current\_digit\_of\_number in chosen\_number:

# Convert the current digit to its equivalent base 10 value

current\_digit\_of\_number = list\_of\_digits.index(current\_digit\_of\_number)

# Calculate the corresponding value in base 10

current\_digit\_of\_number = list\_of\_digits[(remainder \* base\_number + current\_digit\_of\_number) // divisor\_number]

# Update the remainder for the next iteration

remainder = (remainder \* base\_number + current\_digit\_of\_number) % divisor\_number

# Append the current digit to the result

result = result + current\_digit\_of\_number

# Remove leading zeros from the result

while result[0] == '0' and len(result) > 1:

result = result[1:]

# Return the result and the remainder of the division

return result, remainder

function rapid\_conversion\_from\_base\_2(base\_two\_number, target\_base):

# Correspondence table is a dictionary that has all the equivalent representations for the numbers from 0 to 16 in their corresponding base

corresponding\_number\_representation\_table = {

2: ["0", "1", "10", "11", "100", "101", "110", "111", "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"],

4: ["0", "1", "2", "3", "10", "11", "12", "13", "20", "21", "22", "23", "30", "31", "32", "33"],

8: ["0", "1", "2", "3", "4", "5", "6", "7", "10", "11", "12", "13", "14", "15", "16", "17"],

16: ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", "F"],

}

# Set the index to the last digit of the base\_two\_number

index\_base\_two\_number = length(base\_two\_number) - 1

# Calculate the number of digits to group based on the target base

grouped\_digits = floor(log(target\_base, 2))

# Initialize an empty string to store the result

result = ""

# Continue processing until all digits in base\_two\_number are processed

while index\_base\_two\_number >= 0:

# Take the digits from the base 2 number, from right to left, and group them in groups of grouped\_digits digits

group = ""

for \_ in range(grouped\_digits):

# Add the digits to the group, as long as there are digits to add

if index\_base\_two\_number >= 0:

group = base\_two\_number[index\_base\_two\_number] + group

index\_base\_two\_number -= 1

# Remove leading zeros from the group

while group[0] == '0' and length(group) > 1:

group = group[1:]

# Find the equivalent representation of the group in the target base and add it to the front of the result

result = corresponding\_number\_representation\_table[target\_base][corresponding\_number\_representation\_table[2].index(group)] + result

return result

function rapid\_conversion\_to\_base\_2(number\_in\_base, from\_base\_convert):

# Correspondence table is a dictionary that has all the equivalent representations for the numbers from 0 to 16 in their corresponding base

correspondence\_table = {

2: ["0", "1", "10", "11", "100", "101", "110", "111", "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"],

4: ["0", "1", "2", "3", "10", "11", "12", "13", "20", "21", "22", "23", "30", "31", "32", "33"],

8: ["0", "1", "2", "3", "4", "5", "6", "7", "10", "11", "12", "13", "14", "15", "16", "17"],

16: ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", "F"],

}

# grouped\_digits represent how many digits are in base 2 from base p in a group

# Example: For converting from base 16, a digit in base 16 corresponds to 4 digits in base 2

grouped\_digits = floor(log(from\_base\_convert, 2))

# Set the index to the last digit of the number\_in\_base

index\_number\_in\_base = length(number\_in\_base) - 1

# Initialize an empty string to store the result

result = ""

# Loop through each digit in number\_in\_base from left to right

while index\_number\_in\_base >= 0:

# Find the equivalent representation in base 2 for the current digit

group = correspondence\_table[2][correspondence\_table[from\_base\_convert].index(str(number\_in\_base[index\_number\_in\_base]))]

index\_number\_in\_base -= 1

# Add leading zeros to the group as needed

while length(group) < grouped\_digits:

group = "0" + group

# Add the group to the front of the result

result = group + result

# Remove leading zeros from the result if they are present

while length(result) > 1 and result[0] == '0':

result = result[1:]

return result

function convert\_using\_substitution\_method(converting\_number, source\_base, destination\_base):

list\_of\_digits = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F']

# Find the equivalent representation of the source base in the digit set

source\_base = list\_of\_digits[source\_base]

# Set the index to the last digit of the converting\_number

index = length(converting\_number) - 1

# Initialize an empty string to store the result

res = ""

# Initialize the multiplier\_number to "1"

multiplier\_number = "1"

# Iterate through each digit in converting\_number from right to left

while index >= 0:

# Take the digit from the number

dig = converting\_number[index]

# Multiply the digit by the multiplier\_number and add it to the result

current\_number = multiply\_number\_by\_a\_digit(multiplier\_number, dig, destination\_base)

res = add\_two\_numbers(res, current\_number, destination\_base)

# Multiply the multiplier\_number by the source\_base for the next iteration

multiplier\_number = multiply\_number\_by\_a\_digit(multiplier\_number, source\_base, destination\_base)

# Move to the next digit

index -= 1

# Remove leading zeros from the result if they are present

while length(res) > 0 and res[0] == '0':

res = res[1:]

return res

function convert\_using\_successive\_divisions\_method(converting\_number, source\_base, destination\_base):

list\_of\_digits = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F']

# Find the equivalent representation of the source base in the digit set

source\_base = list\_of\_digits[source\_base]

# Initialize an empty string to store the result

res = ""

# Continue the loop until converting\_number becomes "0"

while converting\_number != "0":

# Divide the number by the destination\_base and find the remainder

converting\_number, remainder = divide\_a\_number\_to\_a\_digit\_in\_base\_p(converting\_number, source\_base, destination\_base)

# Add the remainder at the front of the result

res = list\_of\_digits[remainder] + res

# Remove leading zeros from the result if they are present

while length(res) > 0 and res[0] == '0':

res = res[1:]

return res

function convert\_using\_base\_10\_as\_intermediary\_base(converting\_number, source\_base, destination\_base):

# Convert the number from the source\_base to base 10

# Convert the number from base 10 to the destination\_base

# Return the result

if source\_base == 16:

# If the source\_base is 16, use successive divisions method for conversion to base 10

result\_in\_base\_10 = convert\_using\_successive\_divisions\_method(converting\_number, source\_base, 10)

return result\_in\_base\_10, convert\_using\_successive\_divisions\_method(result\_in\_base\_10, 10, destination\_base)

if destination\_base == 16:

# If the destination\_base is 16, use substitution method for conversion to base 10

result\_in\_base\_10 = convert\_using\_substitution\_method(converting\_number, source\_base, 10)

return result\_in\_base\_10, convert\_using\_substitution\_method(result\_in\_base\_10, 10, destination\_base)

# Use substitution method for both conversions if neither source\_base nor destination\_base is 16

result\_in\_base\_10 = convert\_using\_substitution\_method(converting\_number, source\_base, 10)

return result\_in\_base\_10, convert\_using\_substitution\_method(result\_in\_base\_10, 10, destination\_base)

**Number Base Operations:**

# Function to add two numbers represented as strings in a given base

function add\_two\_numbers(number\_1: string, number\_2: string, base: integer) -> string:

# Initialize variables to store the sum and carry

sum\_of\_numbers = ''

carry = 0

# Equalize the lengths of the input numbers by adding leading zeros

number\_1, number\_2 = equalize\_lengths(number\_1, number\_2)

# Start from the rightmost digit of the numbers

index = length(number\_1) - 1

# Iterate through each digit of the numbers

while index >= 0:

# Add two digits and the carry from the previous step

sum\_of\_digits, carry = add\_two\_digits(number\_1[index], number\_2[index], base, carry)

# Append the sum of digits to the result

sum\_of\_numbers = sum\_of\_digits + sum\_of\_numbers

# Move to the next digit

index -= 1

# Check if there is a leftover carry after processing all digits

if carry != 0:

sum\_of\_numbers = digit\_to\_string(carry) + sum\_of\_numbers

# Return the final sum of numbers

return sum\_of\_numbers

# Function to subtract two numbers represented as strings in a given base

function subtract\_two\_numbers(number\_1: string, number\_2: string, base: integer) -> string:

# Check if the first number is greater than the second one and swap them if necessary

negative = False

if convert\_to\_decimal(number\_1, base) < convert\_to\_decimal(number\_2, base):

number\_2, number\_1 = number\_1, number\_2

negative = True

# Initialize variables to store the difference and carry

difference = ''

carry = 0

# Equalize the lengths of the input numbers by adding leading zeros

number\_1, number\_2 = equalize\_lengths(number\_1, number\_2)

# Start from the rightmost digit of the numbers

index = length(number\_1) - 1

# Iterate through each digit of the numbers

while index >= 0:

# Subtract two digits and the carry from the previous step

difference\_of\_digits, carry = subtract\_two\_digits(number\_1[index], number\_2[index], base, carry)

# Append the difference of digits to the result

difference = difference\_of\_digits + difference

# Move to the next digit

index -= 1

# Remove unnecessary leading zeroes from the difference

while difference[0] == '0':

if difference == '0':

break

difference = difference[1:]

# Check if the difference is negative and add a minus sign

if negative:

difference = '-' + difference

# Return the final difference

return difference

# Function to multiply a number represented as a string by a single-digit number in a given base

function multiply\_number\_by\_a\_digit(number: string, digit: string, base: integer) -> string:

# Initialize variables to store the product and carry

product = ''

carry = 0

# Start from the rightmost digit of the number

index = length(number) - 1

# Iterate through each digit of the number

while index >= 0:

# Multiply a digit of the number by the given single-digit digit and add the carry

product\_digits, carry = multiply\_two\_digits(number[index], digit, base, carry)

# Append the product digits to the result

product = product\_digits + product

# Move to the next digit

index -= 1

# Check if there is a leftover carry after processing all digits

if carry != 0:

product = digit\_to\_string(carry) + product

# Remove unnecessary leading zeroes from the product

if product == "0" \* length(product):

product = "0"

# Return the final product

return product

# Function to divide a number represented as a string by a single-digit number in a given base

function divide\_number\_by\_a\_digit(number: string, digit: string, base: integer) -> tuple[string, string]:

# Initialize variables to store the quotient and remainder

quotient = ''

remainder = ''

# Add a zero to the start of the number

number = '0' + number

# Continue dividing until the number is a single digit and less than the divisor

while not length(number) == 1 or not string\_to\_digit(number[0]) < string\_to\_digit(digit):

# Calculate the value of the first two digits in base 10

first\_two\_digits\_in\_base\_10 = string\_to\_digit(number[0]) \* base + string\_to\_digit(number[1])

# Calculate the quotient and remainder for the current step

quotient = quotient + digit\_to\_string(first\_two\_digits\_in\_base\_10 // string\_to\_digit(digit))

remainder = digit\_to\_string(first\_two\_digits\_in\_base\_10 % string\_to\_digit(digit))

# Update the number by removing the processed digits

number = remainder + number[2:]

# Remove unnecessary leading zeroes from the quotient

while quotient[0] == '0':

if quotient == '0':

break

quotient = quotient[1:]

# Return the final quotient and remainder

return quotient, remainder

**Assert Test Data**

import ConversionMethods  
import NumberBaseOperations  
  
  
def test\_convert\_using\_substitution\_method():  
 assert ConversionMethods.convert\_using\_substitution\_method("1010", 2, 10) == "10"  
 assert ConversionMethods.convert\_using\_substitution\_method('10001101', 2, 7) == '261'  
 assert ConversionMethods.convert\_using\_substitution\_method('24213433034122404', 5, 12) == '71713417456'  
  
  
def test\_convert\_using\_successive\_divisions\_method():  
 assert ConversionMethods.convert\_using\_successive\_divisions\_method("10", 10, 2) == "1010"  
 assert ConversionMethods.convert\_using\_successive\_divisions\_method('10001101', 2, 7) == '261'  
 assert ConversionMethods.convert\_using\_successive\_divisions\_method('71713417456', 12, 5) == '24213433034122404'  
  
  
def test\_add\_two\_numbers():  
 assert NumberBaseOperations.add\_two\_numbers("1010", "1010", 2) == "10100"  
 assert NumberBaseOperations.add\_two\_numbers('76', '90A', 11) == '985'  
 assert NumberBaseOperations.add\_two\_numbers('7C', '97', 15) == '124'  
  
  
def test\_subtract\_two\_numbers():  
 assert NumberBaseOperations.subtract\_two\_numbers("1010", "1010", 2) == "0"  
 assert NumberBaseOperations.subtract\_two\_numbers("1B23", "97", 15) == "1A7B"  
  
  
def test\_multiply\_number\_by\_a\_digit():  
 assert NumberBaseOperations.multiply\_number\_by\_a\_digit("1010", "1", 2) == "1010"  
 assert NumberBaseOperations.multiply\_number\_by\_a\_digit("53", "4", 6) == "340"  
 assert NumberBaseOperations.multiply\_number\_by\_a\_digit("53A", "7", 12) == "312A"  
  
  
def test\_divide\_number\_by\_a\_digit():  
 assert NumberBaseOperations.divide\_number\_by\_a\_digit("1010", "1010", 2) == ("1", "0")  
 assert NumberBaseOperations.divide\_number\_by\_a\_digit('67ABC56', 'A', 13) == ('877907', '1')  
  
  
def test\_rapid\_conversion\_from\_base\_2():  
 assert ConversionMethods.rapid\_conversion\_from\_base\_2("1010", 16) == "A"  
 assert ConversionMethods.rapid\_conversion\_from\_base\_2("1010", 8) == "12"  
 assert ConversionMethods.rapid\_conversion\_from\_base\_2("1010", 4) == "22"  
  
  
def test\_rapid\_conversion\_to\_base\_2():  
 assert ConversionMethods.rapid\_conversion\_to\_base\_2('A', 16) == '1010'  
 assert ConversionMethods.rapid\_conversion\_to\_base\_2('72', 8) == '111010'  
  
  
def test\_convert\_using\_base\_10\_as\_intermediary\_base():  
 assert ConversionMethods.convert\_using\_base\_10\_as\_intermediary\_base('10001101', 2, 7) == '261'  
 assert ConversionMethods.convert\_using\_base\_10\_as\_intermediary\_base('24213433034122404', 5, 12) == '71713417456'  
  
  
def test\_all\_functions():  
 test\_convert\_using\_substitution\_method()  
 test\_convert\_using\_successive\_divisions\_method()  
 test\_add\_two\_numbers()  
 test\_subtract\_two\_numbers()  
 test\_multiply\_number\_by\_a\_digit()  
 test\_divide\_number\_by\_a\_digit()  
 test\_rapid\_conversion\_from\_base\_2()  
 test\_rapid\_conversion\_to\_base\_2()  
 test\_convert\_using\_base\_10\_as\_intermediary\_base()

**Here are all the tests that verify that all the functions work properly**