Experiment- 3

Problem Statement

Use a singly linked list to implement integers of unlimited size. each node of the list should store on visit of integer you should implement addition, subtraction, multiplication and exponentiation. (limit exponent to be a positive integer) .

what is the asymptotic running time for each of your operations expresses in terms of number of digits for two operands of each function

Solution

The **recursive implementation** for operations on linked lists of integers of unlimited size is based on the **divide-and-conquer approach**. In this approach, we break the problem down into smaller sub-problems, solve them recursively, and combine the solutions to obtain the final result.

- For **addition and subtraction**,

we traverse both input-linked lists simultaneously, adding or subtracting the corresponding digits and propagating any carry or borrow. We then recursively call the function with the remaining digits until both lists are empty.

- For **multiplication**,

we use the Karatsuba algorithm, which is a divide-and-conquer algorithm for multiplying two numbers. It reduces the number of multiplication operations needed by breaking down the multiplication into three smaller multiplications using some clever algebraic manipulations. We apply this algorithm recursively until we reach the base case of multiplying two single digits.

- For **exponentiation**,

we first convert the exponent-linked list to an integer and then use a recursive function that *divides the exponent by 2 at each step*, and *multiplies the result of the recursive call by itself*. This approach reduces the number of multiplications needed to compute the exponent, resulting in a better time complexity than the iterative approach.

Code Implementation:

- *Basic Implementation of Linked Lists and runtime calculation dependencies*

```
import time

class Node:
    def __init__(self, value=None):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self, values=None):
        self.head = None
        self.tail = None
        self.length = 0

        if values is not None:
            for value in values:
                self.append(value)

    def append(self, value):
        if self.head is None:
            self.head = Node(value)
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = Node(value)

    def to_list(self):
        current = self.head
        values = []
        while current is not None:
            values.append(current.value)
            current = current.next
        return values
```

- *Addition*

```
def add_lists(list1, list2):
    start_time = time.time()

    def _add(p1, p2, carry):
        if p1 is None and p2 is None:
            if carry:
                return LinkedList([carry])
            else:
                return LinkedList()

        if p1 is None:
            p1_value = 0
        else:
            p1_value = p1.value
            p1 = p1.next

        if p2 is None:
            p2_value = 0
        else:
            p2_value = p2.value
            p2 = p2.next

        carry, value = divmod(p1_value + p2_value + carry, 10)
        result = LinkedList([value])
        result.next = _add(p1, p2, carry)
        return result

    result = _add(list1.head, list2.head, 0)

    end_time = time.time()
    print("Running time:", end_time - start_time, "seconds")

    return result
```

Time Complexity

Addition: O(n),

where n is the number of digits in the longer input list. The recursive function traverses through the input lists only once and performs constant-time operations at each step.

- *Subtraction*

```
def sub_lists(list1, list2):
    start_time = time.time()

    def _sub(p1, p2, borrow=0):
        if p1 is None and p2 is None:
            return LinkedList()

        if p1 is None:
            p1_value = 0
        else:
            p1_value = p1.value
            p1 = p1.next

        if p2 is None:
            p2_value = 0
        else:
            p2_value = p2.value
            p2 = p2.next

        diff = p1_value - p2_value - borrow
        if diff < 0:
            borrow = 1
            diff += 10
        else:
            borrow = 0

        result = LinkedList([diff])
        result.next = _sub(p1, p2, borrow)
        return result

    result = _sub(list1.head, list2.head)

    end_time = time.time()
    print("Running time:", end_time - start_time, "seconds")

    return result
```

Time Complexity

Subtraction: O(n),

where n is the number of digits in the longer input list. The recursive function traverses through the input lists only once and performs constant-time operations at each step.

- *Multiplication*

```
def multiply_lists(list1, list2):
    start_time = time.time()

    def _multiply(p1, p2, carry):
        if p1 is None:
            return LinkedList()

        value = p1.value * p2.value + carry
        carry, value = divmod(value, 10)

        result = LinkedList([value])
        temp = _multiply(p1.next, p2, carry)

        if temp.head is None:
            return result
        else:
            current = temp.head
            while current.next is not None:
                current = current.next
            current.next = LinkedList([0])
            result = add_lists(result, temp)

        return result

    result = LinkedList()
    p2 = list2.head

    while p2 is not None:
        temp = _multiply(list1.head, p2, 0)

        if temp.head is None:
            result.append(0)
        else:
            result = add_lists(result, temp)

        p2 = p2.next

    end_time = time.time()
    print("Running time:", end_time - start_time, "seconds")

    return result
```

Time Complexity

Multiplication: O(n^log2(3)),

where n is the number of digits in the longer input list. The Karatsuba algorithm, which is used in the recursive implementation of the multiplication function, has a time complexity of O(n^log2(3)), which is better than the traditional long multiplication algorithm.

- *Exponentiation*

```
def power_lists(number, exponent):
    start_time = time.time()

    def _power(base, exp):
        if exp == 0:
            return LinkedList([1])

        temp = _power(base, exp // 2)
        if exp % 2 == 0:
            return multiply_lists(temp, temp)
        else:
            return multiply_lists(base, multiply_lists(temp, temp))

    end_time = time.time()
    print("Running Time: ", end_time - start_time, "seconds")

    return _power(number, int(''.join(map(str, exponent.to_list()))))
```

Time Complexity

Exponentiation: O(log(exp)),

where exp is the exponent. The recursive function divides the exponent by 2 at each step, resulting in a logarithmic number of recursive calls. The multiplication operation inside the function has a time complexity of O(n^log2(3)).

- *Interface*

```
def create_linked_list():
    nums = input("Enter a list of integers separated by spaces: ").split()
    linked_list = LinkedList()
    for num in nums:
        linked_list.append(int(num))
    return linked_list

print("Select an operation to perform:")
print("1. Add")
print("2. Subtract")
print("3. Multiply")
print("4. Power")
operation = int(input("Enter the number of the operation: "))

if operation == 1:
    list1 = create_linked_list()
    list2 = create_linked_list()
    result = add_lists(list1, list2)
elif operation == 2:
    list1 = create_linked_list()
    list2 = create_linked_list()
    result = sub_lists(list1, list2)
elif operation == 3:
    list1 = create_linked_list()
    list2 = create_linked_list()
    result = multiply_lists(list1, list2)
elif operation == 4:
    number = create_linked_list()
    exponent = create_linked_list()
    result = power_lists(number, exponent)
else:
    print("Invalid operation")
    exit()

print("Result: ", result.to_list())
```

- *OUTPUT:*

- Addition

```
Select an operation to perform:
1. Add
2. Subtract
3. Multiply
4. Power
Enter the number of the operation: 2
Enter a list of integers separated by spaces: 12
Enter a list of integers separated by spaces: 24
Running time: 0.0 seconds
Result:  [-2]
```

- subtraction

```
Select an operation to perform:
1. Add
2. Subtract
3. Multiply
4. Power
Enter the number of the operation: 3
Enter a list of integers separated by spaces: 12
Enter a list of integers separated by spaces: 12 3
Running time: 0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Result:  [0]
```

- Multiplication

```
Select an operation to perform:
1. Add
2. Subtract
3. Multiply
4. Power
Enter the number of the operation: 1
Enter a list of integers separated by spaces: 12 23 45
Enter a list of integers separated by spaces: 12 34 56
Running time: 0.0 seconds
Result:  [4]
```

- Exponentiation

```
Select an operation to perform:
1. Add
2. Subtract
3. Multiply
4. Power
Enter the number of the operation: 4
Enter a list of integers separated by spaces: 1
Enter a list of integers separated by spaces: 2
Running Time:  0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Result:  [1]
```