BLG372E

Analysis of Algorithms

2013 Spring

Report of Project 3

Date of Submission : 04.05.2013

Student Name: Volkan İlbeyli

Student Number : 040100118

Instructor : Hazım Ekenel

Introduction

We are asked to implement the combinatorial solution and the pascal triangle by a class we define as BigUnsignedInteger which can operate on relatively very much larger unsigned integers compared to built-in types of C++.

Data Structures

Data structures used in the implementation is as follows:

- BigUnsignedInteger

```cpp
class BigUnsignedInteger {
public:
    //constructors & destructors
    BigUnsignedInteger();
    BigUnsignedInteger(const BigUnsignedInteger& orig);
    virtual ~BigUnsignedInteger();

    //getters & setters
    vector<unsigned int> getValue() const { return value; }
    void setValue(vector<unsigned int> value) { this->value = value; }

    //operators
    BigUnsignedInteger operator=(const unsigned int&);
    BigUnsignedInteger operator+(const BigUnsignedInteger&);
    BigUnsignedInteger operator+=(const BigUnsignedInteger&);
    BigUnsignedInteger operator+(const unsigned int&);
    BigUnsignedInteger operator+=(const unsigned int&);
    BigUnsignedInteger operator*(const BigUnsignedInteger&);
    BigUnsignedInteger operator*=(const BigUnsignedInteger&);
    BigUnsignedInteger operator*(const unsigned int&);
    BigUnsignedInteger operator*=(const unsigned int&);
    BigUnsignedInteger operator-(const unsigned int&);
    BigUnsignedInteger operator-=(const unsigned int&);
    BigUnsignedInteger operator-(const BigUnsignedInteger&);
    BigUnsignedInteger operator-=(const BigUnsignedInteger&);
    BigUnsignedInteger operator/(const unsigned int&);
    BigUnsignedInteger operator/=(const unsigned int&);
    BigUnsignedInteger operator/(const BigUnsignedInteger&);
    BigUnsignedInteger operator/=(const BigUnsignedInteger&);
    bool operator==(const BigUnsignedInteger&);
    bool operator==(const unsigned int&);
    bool operator<(const BigUnsignedInteger&);
    bool operator<(const unsigned int&);
    bool operator<=(const BigUnsignedInteger&);
    bool operator<=(const unsigned int&);
    BigUnsignedInteger operator>>(const unsigned int);
    BigUnsignedInteger operator<<(const unsigned int);

    //member functions
    void print() const;
    unsigned int length() const;
    unsigned long long val() const;

private:
    vector<unsigned int> value;
};
```

The operators are all overloaded to implement the operations on the class objects.

- CombinatorialSolution

```
class CombinatorialSolution {
public:
    //constructors & destructors
    CombinatorialSolution();
    CombinatorialSolution(const CombinatorialSolution& orig);
    CombinatorialSolution(unsigned const int, unsigned const int);
    virtual ~CombinatorialSolution();

    //getters & setters

    //member functions
    void print() const;
private:
    vector< vector<BigUnsignedInteger> > triangle;
};
```

There is only a 2D vector which holds the triangle data in this class. Notice that only the constructor is used for creating the triangle and writing them to file. There is also a print member function to print to screen.

- PascalTriangle

```
class PascalTriangle {
public:
    //constructors & destructors
    PascalTriangle();
    PascalTriangle(const unsigned int, const unsigned int);
    PascalTriangle(const PascalTriangle& orig);
    virtual ~PascalTriangle();

    //getters & setters

    //member functions
    void print() const;

private:
    vector< vector<BigUnsignedInteger> > triangle;
};
```

This class differs only by the algorithm which creates the triangle of pascal.

Algorithm

PascalTriangle class simply builds up the triangle from 0 by adding previous rows to build up the triangle. Initially the triangle will contain zeros and the edges (first and last element) will be 1. Then during each iteration, previous row will be summed accordingly to create the current row until desired row is created. Then the program will write the triangle in the desired form to the output file.

On the other hand, CombinatorialSolution class will create the triangle using provided dimensions, with all values being 1. How combination will be used is explained in the comment section of the constructor, where all the triangle-building-up occurs.

Complexity Analysis

The program contains many additions, subtractions, multiplications and divisions. First, the regular multiplication and addition algorithms are used. Therefore the complexities are as follows:

Length of the big unsigned integer (number of digits in base 2^{32}): L
Value of the big unsigned integer: V

- **Addition**: $O(L)$

```
for(i=0 ; i<val.size() ; i++){                  O(L)
        dummy = val[i];                         rest is O(1)
        dummy += number.getValue()[i];
        dummy += carry;
        if(dummy > LIMIT){  //if carry
            val[i] = val[i] - ((LIMIT - number.getValue()[i])+1);
            carry = 1;
        }
        else{   //if no carry
            val[i] += carry;
            val[i] += number.getValue()[i];
            carry = 0;
        }
    }
```

- **Subtraction:** $O(L^2)$

```
for(i=0 ; i<number.getValue().size() ; i++){    //for each digit         O(L)
        if((signed long int)(val[i])-number.getValue()[i] < 0){ //borrow occurs
            //set the digit
            val[i] = val[i]+(LIMIT+1-number.getValue()[i]);

            if(val.size() > i+1){   //if there are more than 1 digit     O(L)
                int j;
                for(j=i+1; j<val.size() ; j++){   //iterate through digits
                    if(val[j] == 0){    //if next digit is zero
                        val[j] = LIMIT; //set its value
                        continue;       //move to next digit
                    }
                    else{               //if not zero
                        val[j] = val[j]-1;  //decrement by 1
                        break;          //borrow found, stop iteration
                    }
                }

                //if the last digit (that is borrowed) is zero
                if(j == val.size()-1 && val[j] == 0)
                    val.pop_back();     //remove the last digit
            }
        }
        else{
            val[i] -= number.getValue()[i];
            if(val[i] == 0){
//                  cout << "val[" << i << "]=0" << endl;
//                  cout << "****" << endl << "valsize=" << val.size() << endl;
                if(i==val.size()-1){
                    val.pop_back();
                }
            }
        }
    }
```

- **Multiplication:** $O(Lx(Vnumber + \ LxVlimit))$

```
//get the mid-results of multiplication
    for(int i=0; i<val.size() ; i++){                           O(L)
        result = dummyInt*number.getValue()[i];                 O(V_number)
        for(int j=0 ; j<i ; j++){   //optimization IS required  O(L)
            result = result*LIMIT + result;                     O(V_LIMIT)
        }
        midResults.push_back(result);
    }

    //add mid-results up
    result = 0;
    Iterate(it, midResults){
        result += *it;
    }
```

- **Division:** $O(Vnumber)$

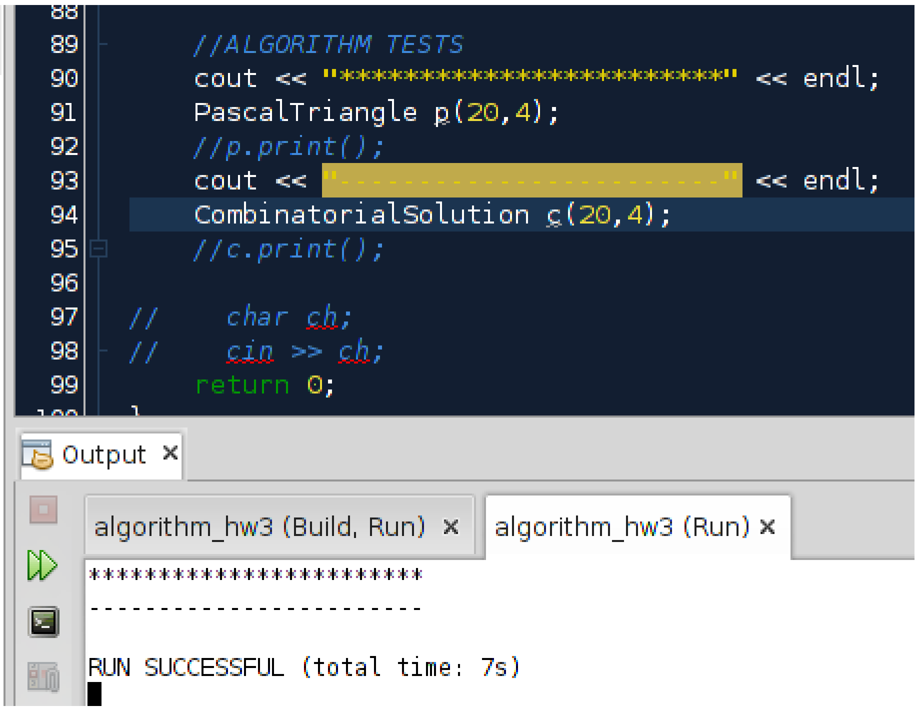
```
    while(dummy.length() > 32){  //subtract until it has unsigned int's size
        dummy -= number;
        division += 1;
    }

    while(dummy.val() > 0){ //dummy will either be 0 or (0 < dummy < number)
        if(dummy.val() < number.val())
            break;
        else{
            dummy -= number;
            division += 1;
        }
    }
```

Comparing Combinatorial solution and Pascal solution, since combinatorial solution has *lots* of multiplications and divisions, it has a much slower running time while on the other hand since pascal triangle solution only has additions, it offers a much faster solution.

During the tests, pascal triangle solution would go up to n=75, completing the solution correctly in a few seconds while combinatorial solution would finish computation in 6 seconds when n=20.

Below is a screenshot for optimization comparison for combinatorial solution.



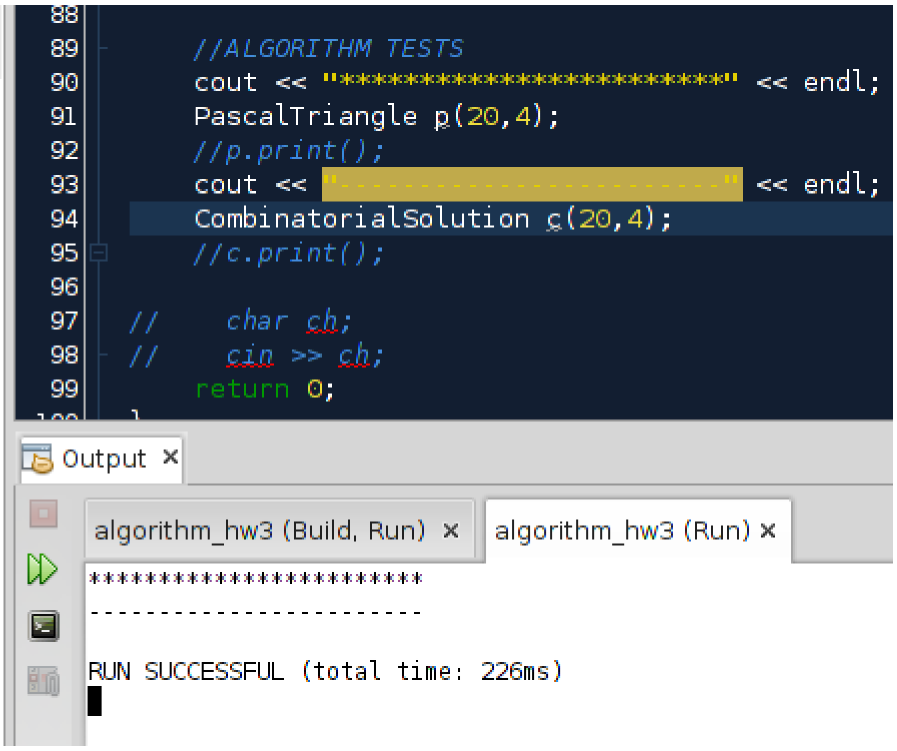
```
    //ALGORITHM TESTS
    cout << "************************" << endl;
    PascalTriangle p(20,4);
    //p.print();
    cout << "------------------------" << endl;
    CombinatorialSolution c(20,4);
    //c.print();

//    char ch;
//    cin >> ch;
    return 0;
```

```
************************
------------------------

RUN SUCCESSFUL (total time: 7s)
```

NO OPTIMIZATION



```
    //ALGORITHM TESTS
    cout << "************************" << endl;
    PascalTriangle p(20,4);
    //p.print();
    cout << "------------------------" << endl;
    CombinatorialSolution c(20,4);
    //c.print();

//    char ch;
//    cin >> ch;
    return 0;
```

```
************************
------------------------

RUN SUCCESSFUL (total time: 226ms)
```

OPTIMIZED RESULTS

Pascal Solution

```
//build the triangle up
for(int i=0; i<=n ; i++){   //for each row
    triangle[i][0] = 1; //set first and last elements to 1
    triangle[i][i] = 1;
    for(int j=1; j<i ; j++){
        //no need for optimization for the if block below
        //unless branch prediction is disabled
        if(i!=0){
            triangle[i][j] += triangle[i-1][j] + triangle[i-1][j-1];
        }
    }
}
```

As it is seen this method only uses multiplication to build up the triangle, thus having a complexity of O(n^2).

Considering space complexity, pascal solution has these variables only for initializing the triangle:

```
//variables for initializing the triangle
BigUnsignedInteger num; //number for populating vectors
vector<BigUnsignedInteger> vec; //vectors for populating triangle rows
```

Combinatorial Solution

```
// n>=k is checked from the main function, thus it is guaranteed that n>=k
//in this solution, not all factorial values are calculated.
//                        5 * 4
//since, e.g., C(5,2) = -----------
//                         2!
//therefore, less multiplication operations are made this way.


//build the triangle up
for(int i=0; i<=n ; i++){   //for each row of triangle
    for(int j=1; j<=i-1 ; j++){    //combination counter
        BigUnsignedInteger nom;     nom = 1;    //nominator of combination
        BigUnsignedInteger denom; denom = 1;  //denominator of combination

        //no need for optimization for the if block below
        //unless branch prediction is disabled
        if(i!=0){
            for(int k=0 ; k<j ; k++){ //j times
                //multiply nominator, by decrementing in each iteration by k
                //thus nominator = n!/(n-k)!
                nom *= (i-k);

                //multiply denominator by k+1, incrementing each iteration
                //thus denominator = j!
                denom *= k+1;
            }
            triangle[i][j] = nom/denom;
        }
    }
}
```

Comments make it clear. Since there are many multiplications and divisions used for this solution, it has a way more higher running time.

Optimization

For optimization, Karatsuba method is NOT used. I preferred shifting operations and therefore implemented right and left shift operators. For multiplication optimization, I used the binary multiplication method which involves shifting multiplicands and then adding up the results based on the left value of the multiplication. But to optimize the running time of combinatorial solution, optimizing division was necessary. Therefore I used a technique based on long-division method by binary numbers which involves shifting. Here is the division algorithms outline, which can also be found on the comment section of the code:

```
/***************************************************************************
 * Optimized division algorithm is based on long division in binary.    *
 * The algorithm works this way: Denominator is left shifted until      *
 * it is bigger than the nominator while the shift count is tracked.    *
 * after that denominator shifted right once for getting a smaller      *
 * value than the denominator. After that, the regular repetitive       *
 * subtraction method is used for the division. At each subtraction     *
 * coefficient is incremented (starting from 1). After the sub-division *
 * is complete, shift count * coefficient is added to result.           *
 * This goes on until the nominator is 1 or 0. Illustration is below:   *
 *                                                                      *
 *  21 / 2                                                              *
 * 2-> 4 -> 8 -> 16 -> 32 (>21) -> 16. shift count = 3                  *
 * 21 - 16 = 15 (<16). coefficient = 1. Result += 1(coef) * 2^3(dummy)  *
 * denominator -> 8; coefficient = 0; nominator = 15; shift count = 2   *
 * 15 - 8 = 7 (<8). coefficient=1, result += 1*2^2;                     *
 * goes on and on until nominator is 1 or 0. Therefore the division     *
 * will have a complexity of O(logn) instead of O(n) where n = value    *
 ***************************************************************************/
```

As it is seen, the division is improved considerably.

To apply optimization, you should define OPT and DIV in the cpp file of BigUnsignedInteger class for multiplication and division, respectively.

Compilation & Running

To compile the program, all the files should be in the same directory.

Compile: `g++ *.cpp`
Run: `./a.out 7 4`

Known Issues

For some reason I could not figure out by spending an hour, in combinatorial solution the iteration stops when value n is greater than 20. I tried many things, monitor results and such, but still I could not detect the problem. However you can see the optimization works for n=20 as I described previously, by seeing the screenshots.

Final Note

I tried to add comments everywhere as much as I could. However, some places might lack some comments. Please search for comments in the similar methods. For the possibility for if I have forgotten to comment a section completely, please feel free to e-mail ilbeyli@itu.edu.tr, I'll explain the vague parts.