Data Structure, Week 2 – Solution

Array

1. Polynomial 에서 두 다항식을 곱하는 Multiply 함수를 작성하라.
   이 함수의 시간 복잡도는 어떻게 되는가?

```
const Polynomial Polynomial::Multiply(const Polynomial& b)
{       // Return *this * b
        Polynomial c;

        for (int aPos = 0; aPos < terms; aPos++)
        {
                Polynomial subResult;
                for (int bPos = 0; bPos < b.terms; bPos++)
                {
                        subResult.NewTerm(termArray[aPos].coef * b.termArray[bPos].coef,
termArray[aPos].exp + b.termArray[bPos].exp);
                }

                c = c.Add(subResult);
        }

        return c;
}
```

다항식 A 의 항의 갯수를 m, 다항식 B 의 항의 갯수를 n이라고 하자.
이 때, Multiply 함수의 시간 복잡도는 $O(m(n+(m+n)) = O(mn+m^2+mn) = O(m^2)$이 된다.

2. SparseMatrix 에서 희소 행렬을 읽고 출력하는 C++ 함수를 작성하라.
   <<와 >>를 연산자 오버로딩해서 구현해야 된다.

```
std::istream& DataStructure::operator>>(std::istream& is, SparseMatrix& sm)
{
	is >> sm.rows >> sm.cols >> sm.terms;

	sm.capacity = sm.terms;
	sm.smArray = new MatrixTerm[sm.terms];

	for (int i = 0; i < sm.terms; i++)
	{
		is >> sm.smArray[i];
	}

	return is;
}

std::istream& DataStructure::operator>>(std::istream& is, MatrixTerm& mt)
{
	is >> mt.row >> mt.col >> mt.value;
	return is;
}

std::ostream& DataStructure::operator<<(std::ostream& os, const SparseMatrix& sm)
{
	os << "Rows: " << sm.rows << ", Cols: " << sm.cols << ", Terms: " << sm.terms << std::endl;

	for (int i = 0; i < sm.terms; i++)
	{
		os << sm.smArray[i];
	}

	return os;
}

std::ostream& DataStructure::operator<<(std::ostream& os, const MatrixTerm& mt)
{
	os << mt.row << ' ' << mt.col << ' ' << mt.value << std::endl;
	return os;
}
```

3. SparseMatrix 에서 두 행렬을 더하고 빼는 Add, Subtract 함수를 작성하라.
   이 함수의 시간 복잡도는 어떻게 되는가?

```
SparseMatrix SparseMatrix::Add(const SparseMatrix& b) const
{       // Return added matrix of two sparse matrices *this and b
        if (rows != b.rows || cols != b.cols) throw "Incompatible matrices";

        SparseMatrix result(rows, cols, 0);

        int idxMatrixA = 0, idxMatrixB = 0;

        while ((idxMatrixA < terms) && (idxMatrixB < b.terms))
        {
                if (smArray[idxMatrixA].row == b.smArray[idxMatrixB].row)
                {
                        if (smArray[idxMatrixA].col == b.smArray[idxMatrixB].col)
                        {
                                result.StoreTerm(smArray[idxMatrixA].row, smArray[idxMatrixA].col,
smArray[idxMatrixA].value + b.smArray[idxMatrixB].value);
                                idxMatrixA++; idxMatrixB++;
                        }
                        else if (smArray[idxMatrixA].col < b.smArray[idxMatrixB].col)
                        {
                                result.StoreTerm(smArray[idxMatrixA].row, smArray[idxMatrixA].col,
smArray[idxMatrixA].value);
                                idxMatrixA++;
                        }
                        else
                        {
                                result.StoreTerm(b.smArray[idxMatrixB].row, b.smArray[idxMatrixB].col,
b.smArray[idxMatrixB].value);
                                idxMatrixB++;
                        }
                }
                else if (smArray[idxMatrixA].row < b.smArray[idxMatrixB].row)
                {
                        result.StoreTerm(smArray[idxMatrixA].row, smArray[idxMatrixA].col,
smArray[idxMatrixA].value);
                        idxMatrixA++;
                }
                else
                {
                        result.StoreTerm(b.smArray[idxMatrixB].row, b.smArray[idxMatrixB].col,
```

```
b.smArray[idxMatrixB].value);
                        idxMatrixB++;
                }
        }

        // Add rest terms of *this
        for (; idxMatrixA < terms; idxMatrixA++)
                result.StoreTerm(smArray[idxMatrixA].row, smArray[idxMatrixA].col,
smArray[idxMatrixA].value);
        // Add rest terms of matrix b
        for (; idxMatrixB < b.terms; idxMatrixB++)
                result.StoreTerm(b.smArray[idxMatrixB].row, b.smArray[idxMatrixB].col,
b.smArray[idxMatrixB].value);

        return result;
}

SparseMatrix SparseMatrix::Subtract(const SparseMatrix& b) const
{       // Return subtracted matrix of two sparse matrices *this and b
        if (rows != b.rows || cols != b.cols) throw "Incompatible matrices";

        SparseMatrix result(rows, cols, 0);

        int idxMatrixA = 0, idxMatrixB = 0;

        while ((idxMatrixA < terms) && (idxMatrixB < b.terms))
        {
                if (smArray[idxMatrixA].row == b.smArray[idxMatrixB].row)
                {
                        if (smArray[idxMatrixA].col == b.smArray[idxMatrixB].col)
                        {
                                result.StoreTerm(smArray[idxMatrixA].row, smArray[idxMatrixA].col,
smArray[idxMatrixA].value - b.smArray[idxMatrixB].value);
                                idxMatrixA++; idxMatrixB++;
                        }
                        else if (smArray[idxMatrixA].col < b.smArray[idxMatrixB].col)
                        {
                                result.StoreTerm(smArray[idxMatrixA].row, smArray[idxMatrixA].col,
smArray[idxMatrixA].value);
                                idxMatrixA++;
                        }
                        else
                        {
```

```
                                        result.StoreTerm(b.smArray[idxMatrixB].row, b.smArray[idxMatrixB].col,
b.smArray[idxMatrixB].value);
                                        idxMatrixB++;
                                }
                        }
                        else if (smArray[idxMatrixA].row < b.smArray[idxMatrixB].row)
                        {
                                result.StoreTerm(smArray[idxMatrixA].row, smArray[idxMatrixA].col,
smArray[idxMatrixA].value);
                                idxMatrixA++;
                        }
                        else
                        {
                                result.StoreTerm(b.smArray[idxMatrixB].row, b.smArray[idxMatrixB].col,
b.smArray[idxMatrixB].value);
                                idxMatrixB++;
                        }
                }

                // Add rest terms of *this
                for (; idxMatrixA < terms; idxMatrixA++)
                        result.StoreTerm(smArray[idxMatrixA].row, smArray[idxMatrixA].col,
smArray[idxMatrixA].value);
                // Add rest terms of matrix b
                for (; idxMatrixB < b.terms; idxMatrixB++)
                        result.StoreTerm(b.smArray[idxMatrixB].row, b.smArray[idxMatrixB].col,
b.smArray[idxMatrixB].value);

                return result;
}
```

희소 행렬 A 의 항의 갯수를 m, 희소 행렬 B 의 항의 갯수를 n이라고 하자.
이 때, Add 함수와 Subtract 함수의 시간 복잡도는 $O(m+n)$이 된다.

4. String 에서 입력으로 문자열을 받아 문자열 내에 각기 다른 문자가 나타나는 횟수를 구하는 함수 Frequency 를 작성하라. 적절한 데이터를 이용해 이 함수를 테스트하라.
   이 함수의 시간 복잡도는 어떻게 되는가?

```
void String::GetFrequencyEachChar()
{
	const int ALPHABET_NUM = 26;
	int alphabets[ALPHABET_NUM] = { 0 };	// Array for alphabet's frequency

	for (int i = 0; i < Length(); i++)
	{
		if ((str[i] >= 'a' && str[i] <= 'z') || (str[i] >= 'A' && str[i] <= 'Z'))
		{
			char upperChar = toupper(str[i]);  // Convert to upper char
			int idxAlphabet = static_cast<int>(upperChar) - static_cast<int>('A');
			alphabets[idxAlphabet]++;
		}
	}

	std::cout << "String::GetFrequencyEachChar()" << std::endl;
	for (int i = 0; i < ALPHABET_NUM; i++)
		std::cout << static_cast<char>(static_cast<int>('A') + i) << ": " << alphabets[i] << std::endl;
}
```

문자열의 길이를 n이라고 하자.
이 때, Frequency 함수의 시간 복잡도는 $O(n)$이 된다.

5. String 에서 문자열과 2 개의 정수(start / length)를 입력으로 받는 Delete 함수를 작성하라.
   이 함수는 start 에서 시작해 length 만큼의 문자를 원래 문자열에서 제거한 새로운 문자열을 반환해야 된다. 이 함수의 시간 복잡도는 어떻게 되는가?

```
String String::DeleteSubString(int idxStart, int deleteLength)
{
	char* deletedStr = new char[Length() + 1];// For null('\0')
	int idxResultStr = 0;

	for (int i = 0; i < Length(); i++)
	{
		// Skip to store: idxStart ~ idxStart+deleteLength-1
		if (i >= idxStart && i < idxStart + deleteLength) continue;
		else deletedStr[idxResultStr++] = str[i];
	}

	deletedStr[idxResultStr] = '\0';	// String ends with null('\0')

	String result(deletedStr, idxResultStr);

	delete[] deletedStr;
	deletedStr = nullptr;

	return result;
}
```

문자열의 길이를 n이라고 하자.
이 때, Delete 함수의 시간 복잡도는 $O(n)$이 된다.

6. String 에서 하나의 문자 c 를 입력으로 받는 CharDelete 함수를 작성하라.
   이 함수는 문자열에서 문자 c 가 나타나는 경우 모두 제거한 나머지 문자열을 반환한다.
   이 함수의 시간 복잡도는 어떻게 되는가?

```
String String::DeleteChar(char c)
{
	char* deletedStr = new char[Length() + 1];// For null('\0')
	int idxResultStr = 0;

	for (int i = 0; i < Length(); i++)
	{
		// Skip to store: specific char(c)
		if (c == str[i]) continue;
		else deletedStr[idxResultStr++] = str[i];
	}

	deletedStr[idxResultStr] = '\0';	// String ends with null('\0')

	String result(deletedStr, idxResultStr);

	delete[] deletedStr;
	deletedStr = nullptr;

	return result;
}
```

문자열의 길이를 n이라고 하자.
이 때, CharDelete 함수의 시간 복잡도는 $O(n)$이 된다.

7. 다음의 각 패턴에 대해 실패 함수를 계산하라.

A. aaaab

	A	a	a	a	b
f	-1	0	1	2	-1

B. ababaa

	a	b	a	b	a	a
f	-1	-1	0	1	2	0

C. abaabaabb

	a	b	a	a	b	a	a	b	b
f	-1	-1	0	0	1	2	3	4	-1