Simon Chau
SID: 862048772

Assignment 1

Exercise 1:

A. See “main1.cpp”

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

string RemoveSpaces(string input);
void CheckValidPoly(string input);
void CheckValidElement(string element);
int HighestDegree(vector<string> elementsVector);

int main() {
    try{
        string inputString;
        cout << "Enter a polynomial: " ;
        getline(cin, inputString);
        string cleanString = RemoveSpaces(inputString);
        CheckValidPoly(cleanString);
    }
    catch(runtime_error& excpt) {
        cout << "Invalid Polynomial" << endl;
        cout << excpt.what() << endl;
    }
    catch(invalid_argument& excpt) {
        cout << "Invalid Polynomial" << endl;
        cout << excpt.what() << endl;
    }
    return 0;
};

string RemoveSpaces(string input) {
    string::size_type i;
    string newInput;
    for(i = 0; i < input.size(); i++) {
        if(input[i] != ' ') {
            newInput += input[i];
        }
    }
    return newInput;
};

void CheckValidPoly(string input) {
    //check for negatives or exponentials
    //output big-o if valid polynomial
    string::size_type i, j;
    string element;
    int substringSize;
    vector<string> elementsVector;
    for(i = 0; i < input.size(); i++) {
        for(j = i + 1; j < input.size(); j++) {
            if(input[i] != '+' && input[j] == '+') {
                substringSize = j - i;
                element = input.substr(i, substringSize);
                elementsVector.push_back(element);
                break;
            }
            else if(input[j] == '-') {
                throw runtime_error("No Negation");
            }
        }
        if(j == input.size()) {
            substringSize = j - i;
```

```
            element = input.substr(i, substringSize);
            elementsVector.push_back(element);
        }
        i = j;
    }
    cout << "Parsed Elements:" << endl;
    for(i = 0; i < elementsVector.size(); i++) {
        string currElement = elementsVector.at(i);
        cout << currElement << endl;
        CheckValidElement(currElement);
    }
    cout << "Valid Polynomial!" << endl;

    int highestDegree = HighestDegree(elementsVector);
    if(highestDegree > 0) {
        cout << "Big-O complexity: O(n^" << highestDegree << ")" << endl;
    }
    else if(highestDegree == 0) {
        cout << "Big-O complexity: O(1)" << endl;
    }

    return;
};

void CheckValidElement(string element) {
    bool containsMult = false;
    bool containsExp = false;
    string exponent;
    string::size_type i, j, k; //j is the index of the '^' char, i and k are just iterators
    for (i = 0; i < element.size(); i++) {
        if(element[i] == '*') {
            containsMult = true;
        }
        else if(element[i] == '^') {
            containsExp = true;
            j = i;
        }
    }
    if(!containsMult) {
        for (k = 1; k < element.size(); k++) {
            if(element[k] == 'n') {
                throw invalid_argument("Need Multiplication Operator");
            }
        }
    }
    if(containsExp) {
        for(i = j + 1; i < element.size(); i++) {
            if(!isdigit(element[i])) {
                throw invalid_argument("Exponent must be integer");
            }
        }
    }
};

int HighestDegree(vector<string> elementsVector) {
    string currElement;
    string exponent;
    bool containsExp;
    int maxDegree = 0;
    string::size_type i, j;
    for(i = 0; i < elementsVector.size(); i++) {
        currElement = elementsVector.at(i);
        //iterate through current element and find index of '^'
        for(j = 0; j < currElement.size(); j++) {
            if(currElement[j] == '^') {
                containsExp = true;
                break;
            }
        }
```

```
        //create substring of everything after the '^' and compare to maxDegree
        if(containsExp) {
            exponent = currElement.substr(j + 1, currElement.size() - j - 1);
            if(stoi(exponent) > maxDegree) {
                maxDegree = stoi(exponent);
            }
        }
    }
    return maxDegree;
};
```

B.

The big-O complexity of my program with respect to m is:
O(m^2)
My most inefficient function “CheckValidPoly” contains a nested loop which iterates m! times to parse the polynomial into individual elements.
m! = m(m-1)/2, which is O(m^2).

C.

See main2.cpp for updated code.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

string RemoveSpaces(string input);
void CheckValidPoly(string input);
void CheckValidElements(vector<string> elementsVector);
int HighestDegree(vector<string> elementsVector);

int main() {
    try{
        string inputString;
        cout << "Enter a polynomial: " ;
        getline(cin, inputString);
        string cleanString = RemoveSpaces(inputString);
        CheckValidPoly(cleanString);
    }
    catch(runtime_error& excpt) {
        cout << "Invalid Polynomial" << endl;
        cout << excpt.what() << endl;
    }
    catch(invalid_argument& excpt) {
        cout << "Invalid Polynomial" << endl;
        cout << excpt.what() << endl;
    }
    return 0;
};

string RemoveSpaces(string input) {
    string::size_type i;
    string newInput;
    for(i = 0; i < input.size(); i++) {
        if(input[i] != ' ') {
            newInput += input[i];
        }
    }
    return newInput;
};
```

Simon Chau
SID: 862048772

```
void CheckValidPoly(string input) {
    //check for negatives or exponentials
    //output big-o if valid polynomial
    string::size_type i, j;
    string element;
    int substringSize;
    vector<string> elementsVector;
    for(i = 0; i < input.size(); i++) {
        for(j = i + 1; j < input.size(); j++) {
            if(input[i] != '+' && input[j] == '+') {
                substringSize = j - i;
                element = input.substr(i, substringSize);
                elementsVector.push_back(element);
                break;
            }
            else if(input[j] == '-') {
                throw runtime_error("No Negation");
            }
        }
        if(j == input.size()) {
            substringSize = j - i;
            element = input.substr(i, substringSize);
            elementsVector.push_back(element);
        }
        i = j;
    }
    CheckValidElements(elementsVector);
    cout << "Valid Polynomial!" << endl;

    int highestDegree = HighestDegree(elementsVector);
    if(highestDegree > 0) {
        cout << "Big-O complexity: O(n^" << highestDegree << ")" << endl;
    }
    else if(highestDegree == 0) {
        cout << "Big-O complexity: O(1)" << endl;
    }

    return;
};

void CheckValidElements(vector<string> elementsVector) {
    bool containsMult = false;
    bool containsExp = false;
    bool isConstant = true;
    int currDegree = 0;
    int compareDegree = 0;
    string currElement;
    string compareElement;
    string::size_type i, j, k, l;
    for(i = 0; i < elementsVector.size(); i++) {
        currElement = elementsVector.at(i);
        for(j = 0; j < currElement.size(); j++) {
            if(currElement[j] == 'n') {
                isConstant = false;
            }
            else if(currElement[j] == '^') {
                containsExp = true;
                k = j + 1;
            }
            else if(currElement[j] == '*') {
                containsMult = true;
            }
        }
        if(!containsMult && !isConstant) {
            throw invalid_argument("Need Multiplication Operator");
        }
        if(containsExp) {
            currDegree = stoi(currElement.substr(k, currElement.size() - k - 1));
            for(k; k < currElement.size(); k++) {
```

```
                if(!isdigit(currElement[k])) {
                    throw invalid_argument("Exponent must be integer");
                }
            }

        }
        for(l = i + 1; l < elementsVector.size(); l++) {
            compareElement = elementsVector.at(l);
            for(j = 0; j < compareElement.size(); j++) {
                if(compareElement[j] == '^') {
                    compareDegree = stoi(compareElement.substr(j + 1, compareElement.size() - j - 1));
                    break;
                }
            }
            if(currDegree == compareDegree) {
                throw invalid_argument("Only one term for each degree");
            }
        }
    }
};

int HighestDegree(vector<string> elementsVector) {
    string currElement;
    string exponent;
    bool containsExp;
    int maxDegree = 0;
    string::size_type i, j;
    for(i = 0; i < elementsVector.size(); i++) {
        currElement = elementsVector.at(i);
        //iterate through current element and find index of '^'
        for(j = 0; j < currElement.size(); j++) {
            if(currElement[j] == '^') {
                containsExp = true;
                break;
            }
        }
        //create substring of everything after the '^' and compare to maxDegree
        if(containsExp) {
            exponent = currElement.substr(j + 1, currElement.size() - j - 1);
            if(stoi(exponent) > maxDegree) {
                maxDegree = stoi(exponent);
            }
        }
    }
    return maxDegree;
};
```

The big-O complexity of my updated program is O(m^2) still.

Simon Chau
SID: 862048772

Exercise 2:

A. See main3.cpp for source code!

```cpp
#include <iostream>

using namespace std;

void SubsetProduct(int A[], int s, int n);

int main() {
    int size;
    int key;
    cout << "Enter number of integers:";
    cin >> size;
    int a[size];
    cout << "\nEnter array values:" << endl;
    for(int i = 0; i < size; i++) {
        cin >> a[i];
    }
    cout << "Enter s:";
    cin >> key;
    SubsetProduct(a, key, size);
    return 0;
}

void SubsetProduct(int A[], int s, int n) {
    cout << "Subset with product " << s << ":" << endl;
    if((n < 3) && (A[0] * A[1]) != s) {
        cout << "No possible subset." << endl;
    }
    else {
        for(int i = 0; i < n; i++) {
            for(int j = i + 1; j < n; j++) {
                if((A[i] * A[j]) == s) {
                    cout << "{" << A[i] << ", " << A[j] << "}" << endl;
                    return;
                }
            }
        }
        cout << "No possible subset." << endl;
    }
}
```

B.

The asymptomatic complexity of this program is O(n^2) since there is a nested loop which iterates n! times. n! = (n)(n-1)/2 => O(n^2).