Assignement 1

Polynomials and product subset

**Source code is included at the top of this file as well as in the zip folder*

```
#include <iostream>
#include <string>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::string;
using std::vector;
/**
 * * Function for exercise 1
 * Function for spliting polynomial and removing spaces
 */
vector<string> SplitStr(string polynomial, char splitChar)
{
    vector<string> splitVector;
    string splitString = "";
    for (int position = 0; (unsigned int)position < polynomial.size(); positio
n++)
    {
        if (polynomial[position] != ' ')
        {
            if (polynomial[position] != splitChar)
                splitString += polynomial[position];
            else
            {
                splitVector.push_back(splitString);
                splitString = "";
            }
        }
    }
    splitVector.push_back(splitString);
    return splitVector;
}

/**
 * * Function for exercise 1
 * Function for checking invalid characters in polynomial
 */
bool findNativeError(string polynomial, vector<char> exception)
```

```
{
    bool hasError = false;
    int position = 0;
    while (!hasError && (unsigned int)position < polynomial.size())
    {
        int exceptionIndex = 0;
        while (!hasError && (unsigned int)exceptionIndex < exception.size())
        {
            hasError = polynomial[position] == exception[exceptionIndex];
            exceptionIndex++;
        }
        position++;
    }
    return hasError;
}

/**
 * * Function for exercise 1
 * Function for checking if a vector contains a value
 */
bool VectorContains(vector<int> vec, int searchedVal)
{
    for (int value : vec)
        if (value == searchedVal)
            return true;
    return false;
}

/**
 * * Function for exercise 1
 * Function for getting the maximum value of a vector
 */
int GetMaxVector(vector<int> vec)
{
    int max = vec[0];
    for (unsigned int i = 1; i < vec.size(); i++)
        if (vec[i] > max)
            max = vec[i];
    return max;
}

/**
 * * Function for exercise 1
 * Function checking if a member of the polynomial is valid and return his com
plexity
 */
int CheckMember(string member)
{
```

```
    int complexity = -1;
    int i = 0;
    int last = member.size() - 1;
    while (i <= last && ((member[i] >= '0' && member[i] <= '9') || member[i] =
= '.'))
        i++;
    if (i - 1 == last)
        complexity = 0;
    else
    {
        bool cont = false;
        if (member[i] == 'n')
        {
            if (i == 0)
                cont = true;
            else if (member[i - 1] == '*')
                cont = true;
        }
        else if (member[i] == '*' && i != last)
        {
            if (member[++i] == 'n')
                cont = true;
        }
        if (cont)
        {
            if (i == last)
                complexity = 1;
            else
            {
                if (member[++i] == '^')
                {
                    string lastDigits = "";
                    i++;
                    while (i <= last && (member[i] >= '0' && member[i] <= '9')
)
                        lastDigits += member[i++];
                    if (i == last + 1)
                        complexity = stoi(lastDigits);
                }
            }
        }
    }
    return complexity;
}

/**
 * * Function for exercise 1
 * Function for checking validity of a polynomial
 */
```

```
void checkPolynomial(string polynomial)
{
    cout << "Testing for " << polynomial << endl;
    if (findNativeError(polynomial, {'(', ')', '-'}))
    {
        cout << "Invalid polynomial expression" << endl;
        return;
    }
    vector<string> members = SplitStr(polynomial, '+');
    vector<int> degres;

    for (string member : members)
    {
        int degre = CheckMember(member);
        if (degre == -1 || VectorContains(degres, degre))
        {
            cout << "Invalid polynomial expression.\n"
                 << endl;
            return;
        }
        else
            degres.push_back(degre);
    }
    int complexity = GetMaxVector(degres);
    cout << "Polynomial is valid, and it's complexity is O(n^" << complexity <
< ").\n"
         << endl;

    return;
}

/**
 * * First exercise of the assignement
 */
void Exercise1()
{
    char polynomial[100];
    cout << "Enter a polynomial : " << endl;
    cin.getline(polynomial, sizeof(polynomial));

    checkPolynomial(polynomial);
}

/**
 * * Helper function that prints the content of a vector<T>
 */
template <typename T>
void PrintVector(vector<T> vec)
{
```

```
    for (const T elm : vec)
    {
        cout << elm << " ";
    }
    cout << endl;
}

/**
 * * Function for exercise 2
 * Find a subset in input array that multiplies to given target
 */
vector<int> findMultiplyingSubset(vector<int> integers, int target)
{
    // Find divisibles of target
    vector<int> divisibles;
    for (const int v : integers)
    {
        if (target % v == 0)
            divisibles.push_back(v);
    }
    // Traverse them to find subset
    vector<int> result;
    int divisiblesSize = divisibles.size();
    int i = 0, j = 0;
    int copyTarget = target;

    while (i < divisiblesSize)
    {
        j = i;
        while (j < divisiblesSize)
        {
            int currentDiv = divisibles[j];
            // we have to check here for further loops iterations with target
being modified hereunder
            if (target % currentDiv == 0)
            {
                target /= currentDiv;
                result.push_back(currentDiv);
            }
            // if target is 1 then we found a subset
            if (target == 1)
                return result;
            ++j;
        }
        // reseting variables
        ++i;
        result.clear();
        target = copyTarget;
    }
```

```
    return result;
}

/**
 * * Second exercise of the assignement
 * ! Modify the inputs directly in the function
 */
void Exercise2()
{
    vector<int> integers = {2, 27, 18, 5, 7, 23, 9, 8, 2, 12, 10, 17, 5, 2, 7,
 9, 8, 10, 4, 10, 8, 5, 4, 8, 9, 10, 7, 6, 5, 4, 1, 24, 5, 8, 9, 7, 2, 5, 11,
4, 2, 5, 2, 5, 2, 5, 25, 2, 5, 52};
    int target(2100);
    vector<int> result = findMultiplyingSubset(integers, target);
    if (result.size() == 0)
    {
        cout << "There is no subset of your input that multiplies to the given
 target." << endl;
    }
    else
    {
        cout << "The subset of your input that multiplies to " << target << "
is :" << endl;
        PrintVector(result);
    }
}

int main()
{
    cout << "EXERCISE 1: \n"
         << endl;
    Exercise1();
    cout << "EXERCISE 2: \n"
         << endl;
    Exercise2();
    cout << system("pause");
    return 0;
}
```

Exercise 1:

A. Using only core C++ (no special libraries, except STL vector or string if you want), write a C++ program that allows a user to input a string and

(a) Checks if the expression is a valid polynomial. Parentheses or negation are not allowed. Spaces should be ignored. E.g., the following are valid
   i. n^2+2*n+5
   ii. 2*n + 4.54* n^5 +4 +5*n

   and the following are invalid
   iii. n^3n
   iv. n^4.2
   v. 5n
   vi. n^3 -3*n

(b) If the polynomial is valid, outputs its big-Oh class. E.g., for (ii) above it is O(n^5).

B. If the length of the input expression is m chars, what is the big-Oh complexity of your program with respect to m?

C. What if we require that there is only one term for each degree? That is, (ii) above is invalid because it has two terms for degree 1 (n^1).

Modify your program accordingly.

What is the asymptotic complexity of the new program?

Throughout the exercise, make any assumptions necessary.

Answers Part 1:

B. In this first case we assume that the polynomial entered by the user has m characters. In order to determine the big-Oh complexity of our program, we must start by analyzing the inner loop and its *primitive* operations. The checkPolynomial algorithm has a main for loop iterating over the m characters:

```
for (string member : members)
    {
        int degre = CheckMember(member);
        if (degre == -1)
        {
            cout << "Invalid polynomial expression.\n"
                 << endl;
            return;
        }
        else
            degres.push_back(degre);
    }
```

Here the primitive operation is the assignment of degre to checkMember, and the condition evaluation. The condition is done is constant time whereas the checkMember must be evaluated. The std::push_back() has amortized $O(1)$ complexity and since we have relatively small inputs we can assume that its time complexity is constant. So that leaves us with checkMember function. Most of the function is the evaluation of conditions which are again running in constant time, excepts for two loops :

```
while (i <= last && ((member[i] >= '0' && member[i] <= '9') || member[i] == '.'))
        i++;
```

and

```
 while (i <= last && (member[i] >= '0' && member[i] <= '9'))
                        lastDigits += member[i++];
```

Those two can theoretically be considered $O(m)$ in a worst-case scenario. That would make the function run in $O(2m + some\ constant\ k)$ which leaves us a time complexity of $O(m)$ for the check member function.

With that, we can assert that the main loop of the checkPolynomial function runs in $O(m^2)$. Let's analyze the rest. Before the loop:

```
cout << "Testing for " << polynomial << endl;
    if (findNativeError(polynomial, {'(', ')', '-'}))
    {
        cout << "Invalid polynomial expression" << endl;
        return;
    }
    vector<string> members = SplitStr(polynomial, '+');
    vector<int> degres;
```

The findNativeError function has a loop iterating over the m characters of our polynomial, and for each character, we are testing if the current character is invalid. If the vector containing the invalid characters contains k characters, then the complexity of this function is $O(m * k)$. In our case, k is constant and is equal to 3 ("(", ")" and "-"). Thus, the runtime falls back to $O(3 * m) \Rightarrow O(m)$.

The SplitStr function splits our polynomial into strings. It iterates over the m characters and performs concatenation of strings when needed (splitting after a parameter character). The string concatenation runs linearly with respect to its size, so that function would also not exceed $O(m^2)$.

```
int complexity = GetMaxVector(degres);
    cout << "Polynomial is valid, and it's complexity is O(n^" << complexity
 << ").\n"
         << endl;
    return;
```

GetMaxVector iterates over the m characters to find the maximum order of the polynomial, so it's runtime is linear as well. This leaves us with an overall $O(m^2)$ complexity for the checkPolynomial function.

C. Here the only condition that changes if that we have to check if the order read on the polynomial is not already present in our vector of orders:

```
for (string member : members)
    {
        int degre = CheckMember(member);
        if (degre == -1 || VectorContains(degres, degre)
        {
            cout << "Invalid polynomial expression.\n"
                 << endl;
            return;
        }
        else
            degres.push_back(degre);
    }
```

The vectorContains function iterates over the m characters and return true if the order we search for is already listed in our vector of orders. Its time complexity is thus $O(m)$. Since we are performing this operation in the main loop, that still leaves us with an overall $O(m^2)$ complexity for the checkPolynomial function.

Exercise 2:

Given an array A of n integers and an integer s, find a subset of the integers in A such that their product is s.

A. Write C++ function.

B. Compute asymptotic complexity.

A. Function included at the top and in the source files.

B. Analysis of the findMultiplyingSubset function:

```
// Find divisibles of target
    vector<int> divisibles;
    for (const int v : integers)
    {
        if (target % v == 0)
            divisibles.push_back(v);
    }
```

This loop puts all the divisibles of our target present in our set in a divisible vector previously created. It iterates over the n integers, and performs the push_back() operation, so its time complexity is O(n). Let's take a look at the main loop of the program:

```
    vector<int> result;
    int divisiblesSize = divisibles.size();
    int i = 0, j = 0;
    int copyTarget = target;

    while (i < divisiblesSize)
    {
        j = i;
        while (j < divisiblesSize)
        {
            int currentDiv = divisibles[j];
            // we have to check here for further loops iterations with targe
t being modified hereunder
            if (target % currentDiv == 0)
            {
                target /= currentDiv;
                result.push_back(currentDiv);
            }
            // if target is 1 then we found a subset
            if (target == 1)
                return result;
            ++j;
        }
        // reseting variables
        ++i;
        result.clear();
        target = copyTarget;
    }
```

The first 3 lines are constant in time since they are assignments. We analyze the inner while's loop primitive operations. Most of them are comparisons and conditions which evaluate in constant time. We have an "early" return if we reach our target. J increment and currentDiv assignments are also constant. If the outer loop iterates over n numbers, the inner loop iterates over $n-1, n-2, n-3$ … elements. So, the overall time complexity of the main loop is $O(k * (n * (n+1)/2))$ where k is the number of operations in the inner loop. The whole-time complexity of the findMultiplyingSubset is thus $O(n^2)$.