Analysis of Algorithms II
Homework 3
Oguzhan Karabacak 150170021

compile : g++ -std=c++11 -Wall -Werror 150170021.cpp -o 150170021

run : ./150170021 <input_file_name>.txt <output_file_name>.txt

run type : second type calico (e2-files_as_main_parameters.t)

1. Report Problem Formulation

a.) Smith Class

```
class Smith{  //Smith-Waterman class
    private:
        int match;  //match number
        int unmatch;  //unmatch number
        int indel;  //gap penalty number
        int score;  //total score
    public:
        Smith(int,int,int);  //constructor
        int findScore(char, char);  //for calculate cell of matrix
        int findMax(int array[], int length); // to find Max score
        int findMaxIndex(int array[], int,int);  //to find max index
        vector<string> findCommon(string,string); //find common sequences
        vector<string> printCommon(string,string);  //for print
        int getScore(){  //getscore function
            return this->score;
        };

};
Smith::Smith(int ma,int unma,int gap){  //constructor
    match=ma;  //match number
    unmatch=unma; //unmatch number
    indel=gap; //gap penalty
    score=0;  //first score
}
```

The Smith class forms the structure of the entire algorithm.
“match”, “unmatch”, “indel” variables holds the match, unmatch and gap penalty numbers.

b.) findScore function

```
int Smith::findScore(char a, char b)
{
    int result;
    if(a==b) result=this->match;  //match  //if match return match number
    else result=this->unmatch;  //unmatch //if unmatch return unmatch number

    return result;
}
```

findScore function is basic function. It cheks if the char's are equal. If equal, return match number, if not return unmatch number.

Pseudo-Code

```
Findscore(a,b)
    If a = b
            result -> match
    else
            result -> unmatch
```

Time Complexity

findScore function is basic function. Just check chars is equal. So the time complexity is O(1).

c.) findMax function

```
int Smith::findMax(int array[], int length){  //find max score
    int max = array[0];

    for(int i=1; i<length; i++){  //traverse all matrix
        if(array[i] > max)  max = array[i];
    }
    return max;  //return max
}
```

findMax function is to find maximum entry score of matrix. And return.

Pseudo-Code

```
findMax(array[],length):
    max -> array[0]  //array first element
    for (i -> 0 to length):
            if (array[i] > max):
                    max -> array.index(i)
    return max
```

Time Complexity

At the beginning of the function, we do the operation max = array [0]. This is a simple operation so the time complexity is O (1). Then we find the maximum value in the array with the for loop. In the worst case, the maximum value is at the end of the array, so the time complexity is O (n), where n = length.

d.) findCommon function

The findCommon function is a very long function, so we will break down the code and calculate the time complexity of each code block separately.

```
vector<string> Smith::findCommon(string s1,string s2){  //for find most common sequences

    int s1_len = s1.length(); // initialize some variables
    int s2_len = s2.length();


    int matrix[s1_len+1][s2_len+1]; // initialize matrix, all entries is 0 initially
    for(int i=0;i<=s1_len;i++){
        for(int j=0;j<=s2_len;j++)  matrix[i][j]=0;
    }

    int neighbors[4];

    for (int i=1;i<=s1_len;i++) {
        for(int j=1;j<=s2_len;j++) {
            neighbors[0] = matrix[i-1][j-1]+findScore(s1[i-1],s2[j-1]);
            neighbors[1] = matrix[i-1][j]+this->indel; //left entry + gap penalty
            neighbors[2] = matrix[i][j-1]+this->indel; //up entry + gap penalty
            neighbors[3] = 0;  //last value is 0
            matrix[i][j] = findMax(neighbors,4);  //find max value and assign to current entry
        }
    }
```

In the first part, we first get the size of the strings. Then we create a 2d array using these sizes and make all the values of the matrix 0. Then we create the neighbors array. This array is used to calculate the value of the current entry. We calculate all entries of the Matrix. The first element of the neighbors with up-left specifies the match-unmatch state of the letters in the string. The second element of the neighbors is the sum of the left entry and the gap penalty. The third element of the neighbors is the sum of the up entry and the gap penalty. Neighbors' last member is directly 0.

Finally, we find the greatest value among these values and equate it to the current entry of the matrix.

Pseudo-Code

```
findCommon(s1,s2):
        s1_len -> s1.length
        s2_len -> s2.length
        matrix[s1_len+1] [s2_len+1]
        for i->0 to s1_len:
                for j -> 0 to s2_len:
                        matrix[i][j] -> 0
        neighbors[4]
        for(i -> 1 to s1_len):
                for(j->1 to s2_len):
                        neighbors[0] -> diagonal_neighbor.value  + findscore(s1[i-1],s2[j-1])
                        neighbors[1] -> up_neighbor.value + gap
                        neighbors[2] -> left_neighbor.value + gap
                        neighbors[3] -> 0
                        matrix[i][j] -> findMax(neighbors,4)
```

Time Complexity:

Since we determine the value of all entries in a 2 dimensional array, time complexity becomes O (m.n). m is the length of the first string, n is the length of the second string.

```
int matrix_max = 0;  //find max score
int i_max=0, j_max=0;  //max score indexes
for(int i=0;i<s1_len+1;i++) {
    for(int j=0;j<s2_len+1;j++) {  //traverse all matrix
        if(matrix[i][j]>matrix_max) {
            matrix_max = matrix[i][j];
            i_max=i;
            j_max=j;
        }
    }
}
int size = 1;   //how many most common
vector<int> first;  //first indexes of matrix for max score
first.push_back(i_max);
first.push_back(j_max);
vector<vector<int> > locations_max;
locations_max.push_back(first);  //add to vector first max score
int i_max2=0, j_max2=0;
for(int i=0;i<s1_len+1;i++){
    for(int j=0;j<s2_len+1;j++) {  //traverse all matrix and find score which is equal to max score
        if(matrix[i][j]==matrix_max && i_max != i) {
            size += 1;
            i_max2=i;
            j_max2=j;
            vector<int> a;
            a.push_back(i_max2);
            a.push_back(j_max2);
            locations_max.push_back(a);  //add to vector
        }
    }
}
```

In this code block, we find the maximum values in the matrix and the indexes of these values. First we find the largest value by going through the entire matrix. Then we assign the indexes of this value to i_max and j_max. Then we create the variable for you. This variable specifies how many maximum values are in the matrix. The locations_max vector, on the other hand, stores the indexes of all maximum points. We use the second for loop to find other maximum indexes. We traverse the entire matrix again and add all indexes whose value is matrix_max but whose indexes are different from matrix_max, to the locations_max vector.

Pseudo-Code

```
matrix_max -> 0
max_index -> 0  // i_max , j_max
for (i -> 0 to s1_len):
        for(j -> 0 to s2_len):
                if matrix[i][j] > matrix_max :
                        matrix_max -> matrix[i][j]
                        max_index -> current.indexes
size -> 1
vector first
first.add(max_index)
vector locations_max
locations_max.add(first)
other_max_index // i_max2 , j_max2
for (i -> 0 to s1_len):
        for(j -> 0 to s2_len):
                if(matrix[i][j] > matrix_max and i_max = i):
                        increment size by 1
                        other_max_index -> current.indexes
                        locations_max.add (other_max_index)
```

```
vector<string> common_seq;  //common sequences vector
cout << "Score: " << matrix_max << " ";  //print score

score=matrix_max;
if(matrix_max == 0){  //if matrix is 0 , return empty common_seq
    return common_seq;
}
for(int i=0;i<size;i++){  //loop by size number for all common sequences

    vector<int>  st_index;
    int current_i=locations_max[i][0];
    int current_j=locations_max[i][1];
    st_index.push_back(current_i-1);
    while (matrix[current_i][current_j] != 0 ){  //traceback,
        current_i = current_i-1;
        current_j = current_j-1;
        if (matrix[current_i][current_j] != 0)  st_index.push_back(current_i-1);
    }
    string total_com="";
    for(int i=st_index.size()-1;i != -1; i--){
        total_com += s1[st_index[i]];  //add to all letter in traceback path
    }
    common_seq.push_back(total_com);  //add to vector
}
return common_seq;
```

We do traceback in this code block. First, we check whether the maximum value is 0 or not, if it is 0, we return the empty vector. If the maximum value is not 0, we do traceback as many times as there are size of maximum value. In the for block, the st_index vector stores the index of common letters.

While tracebacking, while going back from the maximum score, the traceback continues until it reaches 0. When tracebacking, go to the up-left (diagonal) among the neighbors and we find the next indexes. Then, if the value indicated by these indexes is 0, we do not add it to the vector.

Finally, we use these indexes to find common letters and find the most common sequence with them. It then returns the common sequence.

Pseudo-Code

```
vector common_seq
score -> matrix_max
if (matrix_max  = 0 ):
        return common_seq
for ( i -> 0 to size):
        vector st_index
        current_index -> locations_max[i].indexes
        max_index[3]
        st_index.add(current_index - 1)
        while(matrix[current_index] != 0):
                max_index[0] -> matrix[current_index– 1 ]
                current_index -> diagonal_neighbor.indexes
```

```
                    if(matrix[current_index] != 0):
                              st_index.add(current_index - 1)
          total_com-> “”
          for( i -> st_index.size – 1 to -1 ):
                    total_com -> total_com + s1[st_index[i]]
          common_seq.add(total_com)
return common_seq
```

Time Complexity

When doing traceback, we first find the maximum value in the Matrix, then using the indexes of this value, we go back diagonally until the current value is 0. In the worst case, we have to traverse the entire matrix. So time complexity is O (m.n).

As a result, time complexity in Smith Waterman algorithm is O(m.n) to create matrix and O(m.n) to traceback.

2.) Analyze and compare the algorithm results with assessing all possible alignments one by one in terms of:

a.) The Calculations Made

The biggest difference between brute force and smith-waterman is efficieny.
The brute force solution iterates over the array many times to get every possible solution but in smith-waterman algorithm, solution only iterates through the array once.
That's why the Smith-Waterman algorithm is much faster than Brute-Force.
It is enough to create an m x n matrix in the Smith-Waterman algorithm and fill this matrix and traceback it. (m is first string, n is second string) (m.n)
But in the Brute-Force algorithm, we need to operate on the array as much as the n combination of (m + n). ($C\binom{m+n}{m}$)

For example lets compare brute force and smith-waterman using SAD-SD strings.
(Since it is difficult to implement the sample strings in the homework with brute force, I chose 2 short strings.)

Smith-Waterman Implement for match=1,unmatch=-2, gap = -4 scores

		S	D
	0	0	0
S	0	1	0
A	0	0	0
D	0	0	1

According to this table the most sequences is : ‘S’ and ‘D’

Now, implement with brute-force
S – S (Match, +1)
A – D (Unmatch, -2)
D – (Gap, -4)
Now All possible alignments of “SAD” and “SD”
(SAD, SD-) -> Score : -5 -> 0
(SAD, S-D) -> Score : -2 -> 0
(SAD-, S - - D)
(SA-D , S - D -)
…. and continues

As it can be seen, while the smith-Waterman algorithm finds all the most sequences in a table, the brute-force method takes a very long time.

b.) The Maximum number of calculation results kept in the memory

In the Smith-Waterman algorithm, a matrix of size m.n is created for calculation. m is the length of first string , and n is the length of second string. We also define some variables such as max to keep the results, but they are constant so the space complexity is O (1). The total space complexity of the algorithm is O (m.n).
In brute force, on the other hand, since it iterates every time, there is no need to keep a matrix in memory, we just need to keep the maximum value, so space complexity is O(1).

c.) The Running Time

In the Time complexity of Smith-Waterman is O(n.m) (Question 1).

In the Brute-Force , We number all possible alignments, score each alignment, and choose the alignment with the maximum score.

In the Brute-Force , If we convert the alignment of two strings into a single string (1-1 Correspondence). There are m + n positions in total. Each character in each string takes a position.

So combination of this position $\binom{m+n}{m}$ so time complexity is O$\binom{m+n}{m}$.