CSCI-570 Spring 2025
Analysis of Algorithms
Final Project

Due: May 12, 2025

1 Guidelines

You can work on this project in groups of no more than 3 people.

2 Project Description

Implement the basic Dynamic Programming solution to the Sequence Alignment problem. Run the test set provided and show your results.

2.1 Algorithm Description

Suppose we are given two strings X and Y, where X consists of the sequence of symbols $x_1, x_2, \ldots, x_m$ and Y consists of the sequence of symbols $y_1, y_2, \ldots, y_n$.

Consider the sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ as representing the different positions in the strings X and Y, and consider a matching of these sets; Recall that a matching is a set of ordered pairs with the property that each item occurs in at most one pair. We say that a matching M of these two sets is an alignment if there are no "crossing" pairs: if $(i, j), (i', j') \in M$ and $i < i'$, then $j < j'$. Intuitively, an alignment gives a way of lining up the two strings, by telling us which pairs of positions will be lined up with one another.

Our definition of similarity will be based on finding the optimal alignment between X and Y, according to the following criteria. Suppose M is a given alignment between X and Y:

1. First, there is a parameter $\delta_e > 0$ that defines a gap penalty. For each position of X or Y that is not matched in M – it is a gap – we incur a cost of δ.
2. Second, for each pair of letters p, q in our alphabet, there is a mismatch cost of α_{pq} for lining up p with q. Thus, for each $(i, j) \in M$, we pay the appropriate mismatch cost $\alpha_{x_i y_j}$ for lining up x_i with y_j. One generally assumes that $\alpha_{pp} = 0$ for each letter p – there is no mismatch cost to line up a letter with another copy of itself – although this will not be necessary in anything that follows.
3. The cost of M is the sum of its gap and mismatch costs, and we seek an alignment of minimum cost.

2.2 Input string Generator

The input to the program would be a text file containing the following information:

1. First base string (s_1)

2. Next j lines consist of indices after which the copy of the previous string needs to be inserted in the cumulative string. (eg given below)
3. Second base string (s_2)
4. Next k lines consist of indices after which the copy of the previous string needs to be inserted in the cumulative string. (eg given below)

This information would help generate 2 strings from the original 2 base strings. This file could be used as an input to your program, and your program could use the base strings and the rules to generate the actual strings. Also note that the numbers j and k correspond to the first and the second string, respectively. Make sure you validate the length of the first and the second string to be $2^j * \text{len}(s_1)$ and $2^k * \text{len}(s_2)$. Please note that the base strings need not be of equal length and similarly, j need not be equal to k.

Example:

```
ACTG
3
6
1
TACG
1
2
9
```

Using the above numbers, the generated strings would be `ACACTGACTACTGACTGGTGACTACTGACTGG` and `TATTATACGCTATTATACGCGACGCGGACGCG`.

Following is the step by step process on how the above strings are generated:

ACTG
ACTGACTG
ACTGACTACTGACTGG
ACACTGACTACTGACTGGTGACTACTGACTGG
TACG
TATACGCG
TATTATACGCGACGCGGACGCG

2.3 Values for Delta and Alphas

Values for α's are as follows. δ_e is equal to 30.

	A	C	G	T
A	0	110	48	94
C	110	0	118	48
G	48	118	0	110
T	94	48	110	0

2.4 Programming/Scripting Languages

Following are the list of languages which could be used:

1. C

2. C++
3. Java
4. Python
5. Python3

2.5 Bounds

1. Basic Algorithm
   - $0 \leq j, k \leq 10$
   - $1 \leq \text{len}(s_1), \text{len}(s_2) \leq 2000$
   - $1 \leq 2^j * \text{len}(s_1), 2^k * \text{len}(s_2) \leq 2000$
2. Memory Efficient Algorithm
   - $0 \leq j, k \leq 20$
   - $1 \leq \text{len}(s_1), \text{len}(s_2) \leq 20000$
   - $1 \leq 2^j * \text{len}(s_1), 2^k * \text{len}(s_2) \leq 20000$

3 Goals

Following are the goals to achieve for your project:

3.1

Your program should take 2 arguments:

1. Input file path
2. Output file path (If path is valid and file not found, your program should create it)

Examples:

```
python2 basic_2.py input.txt output.txt
java Basic input.txt output.txt
python3 basic_3.py input.txt output.txt
```

Note: As mentioned in 2.2, input file will have data to generate strings. Since Gap penalty (δ_e) and Mismatch penalty (α_{pq}) are FIXED, you have to hardcode them in your program.

You are not allowed to use any libraries.

3.2

Implement the Dynamic Programming algorithm. Your program should print the following information at the respective lines in the output file:

1. Cost of the alignment (Integer)
2. First string alignment (Consists of A, C, T, G, _ (gap) characters)
3. Second string alignment (Consists of A, C, T, G, _ (gap) characters)

4. Time in Milliseconds (Float)
5. Memory in Kilobytes (Float)

Note: There can be multiple alignments that have the same cost. You can print ANY alignment generated by your program. The only condition is it should have a minimum cost.

e.g. For strings s_1: A and s_2: C, alignments A_, _C and _A, C_ both have alignment cost 60 which is minimum. You can print any one of them.

3.3

Implement the memory-efficient version of this solution and repeat the tests in 3.2.

3.4

Plot the results of 3.2 and 3.3 using a line graph.

Please use the provided input files in the 'datapoints' folder for generating the data points to plot the graph.

1. Single plot of *CPU time* vs *problem size* for the two solutions.
2. Single plot of *Memory usage* vs *problem size* for the two solutions.

Units: CPU time - milliseconds, Memory in KB, problem size m+n

4 Submission

4.1

You should submit the ZIP file containing the following files:

a. Basic algorithm file
   Name of the program file should be 'basic.c' / 'basic.cpp' / 'Basic.java' / 'basic_2.py' (Python 2.7) / 'basic_3.py' (Python 3)

b. Memory efficient algorithm file
   Name of the program file should be 'efficient.c' / 'efficient.cpp' / 'Efficient.java' / 'efficient_2.py'(Python 2.7) / 'efficient_3.py' (Python 3)

c. Summary.pdf
   It must contain the following details:

   1. Datapoints output table (which are generated from provided input files)
   2. Two graphs and Insights
   3. Contribution from each group member e.g. coding, testing, report preparation, etc. if everybody did not have equal contribution (Please use the provided Summary.docx file, fill in the details and upload it as PDF)

d. 2 Shell files 'basic.sh' and 'efficient.sh' with the commands to compile and run your basic and efficient version. These are needed to provide you with flexibility in passing any additional compiler/run arguments that your programs might need. See More Hints (VII part E for more details)

   Example: `basic.sh`

```
javac Basic.java
java Basic "$1" "$2"
```

Execution: `./basic.sh input.txt output.txt`
`./efficient.sh input.txt output.txt`

4.2

The name of your zip file should have the USC IDs (not email ids) of everyone in your group separated by an underscore. e.g.

- `1234567890_1234567890_1234567890.zip`
- Contents (example):

```
1234567890_1234567890_1234567890
- basic_2.py
- efficient_2.py
- Summary.pdf
- basic.sh
- efficient.sh
```

5 Grading

Please read the following instructions to understand how your submission will be evaluated.

5.1 Correctness of algorithms - 70 points

1. Both programs (basic/ efficient) are correctly outputting files having all 5 lines in the correct order: 15 points
2. Basic Algorithm: 25 points
3. Memory Efficient Algorithm: 30 points

Note: Graders will execute your program on the 'Linux' OS. The goal of this part is to check correctness. The program should output a valid alignment having a minimum cost. **We will check whether the answer is valid, the cost is optimal and aligned with the answer.** Memory and Time will be evaluated in the next part.

5.2 Plots, analysis of results, insights and observations: 30 points

1. Your program will be run on the input files (provided by us in the 'data points' folder) to generate output files. ~~The memory and time in the output files should be in a "close range" to what is given by you in the Summary.pdf.~~ **Memory usage and time should follow some certain patterns, but they are no longer required to be within the range of the sample files.**
2. Correctness of the graph
3. Correctness of Analysis/ Insights about the graph and the algorithms.

Note: Unlike 5.1, the evaluation of Part B is subjective so it will be done manually. So it is alright if your graphs/data points have 'some' outliers.

6 What is provided to you in the zip file?

A. SampleTestCases folder containing sample input and output files

B. Datapoint folder containing input files to generate graph data points.

C. Summary.docx file for reference

7 HINTS, NOTES, and FAQs

7.1 Regarding Input and string generation

1. We will never give an invalid input to your program. Input strings will always contain A, C, G, T only.
2. The length of the final input string should be equal to $2^{\text{number of lines}} * \text{len(base string)}$, as mentioned in the document.
3. The string generation mechanism is the same irrespective of the basic or the efficient version of the algorithm.
4. The entire program (string generation, solution, write output) should be written in a single file. You may break those functions into different classes to make the code modular, but there should be only one file. Your program won't be evaluated based on how modular it is.

7.2 Regarding Algorithm and output

1. DO NOT REFER TO THE PSEUDOCODE PROVIDED IN KLEINBERG AND TARDOS.
2. DO NOT USE ANY LIBRARIES FOR WRITING YOUR ALGORITHMS.
3. Samples for time and memory calculation are provided. ~~Please use them for consistency.~~ **You can refer to them. However, their memory reporting may not be accurate, which is acceptable. While more techniques exist for obtaining precise memory readings, code simplicity is prioritized to ensure easy execution without complex environment setup.**
4. Your solutions for the regular and memory-efficient algorithms should be in two different programs.
5. There can be multiple valid sequences with the same optimal cost; you can output any of those. All of them are valid.
6. You should code both the basic version and memory-efficient algorithm. Even though the memory-efficient version will pass all the bounds of the simple version, you must not use the memory-efficient version in both sub-problems.
7. Your program should not print anything when it runs. It should only write to the output file.
8. There is no specific requirement for the precision of Time and Memory float values.
9. Time and Memory depend on so many factors such as CPU, Operating System, etc. So there might be differences in the output. Therefore, it will be evaluated subjectively. There must be a clear distinction in behavior between programs whose Time/ Memory complexity is $O(n)$ vs $O(n^2)$ vs $O(\log n)$.

7.3 Regarding the plot

1. Both the graphs are line graphs. The X-axis represents problem size as m+n, where m and n are the lengths of the generated string, and the Y-axis of the Memory plot represents memory in KB, and the Y-axis of the Time Plot represents time in milliseconds. The 2 lines in the graph will represent stats of basic and memory-efficient algorithms.
2. You can use any libraries/packages in any language to plot the graphs.
3. You do not have to provide code for generating the plots. Only add images in the Summary.pdf.

7.4 Regarding Submission

1. Only 1 person in the group needs to submit the project. We'll get the USC IDs of all the other team members from the filenames.
2. To allow for grading the whole class in a reasonable amount of time, we'll kill your program if it runs for more than a minute on a single input file.

7.5 Regarding Shell File

To make the evaluation seamless on our end, please make sure you also have a shell script named 'basic.sh' and 'efficient.sh' with the commands required to run your program. For example, the contents of this file can be one of the following:

C

```
basic.sh
gcc basic.c -o basic
./basic "$1" "$2"

efficient.sh
gcc efficient.c -o efficient
./efficient "$1" "$2"
```

Listing 1: C Shell Script

C++

```
basic.sh
g++ basic.cpp -o basic
./basic "$1" "$2"

efficient.sh
g++ efficient.cpp -o efficient
./efficient "$1" "$2"
```

Listing 2: C++ Shell Script

Java

```
basic.sh
javac Basic.java
java Basic "$1" "$2"

efficient.sh
javac Efficient.java
java Efficient "$1" "$2"
```

Listing 3: Java Shell Script

python 2.7

```
basic.sh
python2 basic_2.py "$1" "$2"

efficient.sh
python2 efficient_2.py "$1" "$2"
```

Listing 4: Python 2.7 Shell Script

python 3

```
basic.sh
python3 basic_3.py "$1" "$2"

efficient.sh
python3 efficient_3.py "$1" "$2"
```

Listing 5: Python 3 Shell Script

Note that the above are just examples. You can modify them according to your convenience. The goal is to have a language-independent mechanism to run your script to get your outputs. Also note that for python2 or python3 users, it's important that they have 2 or 3 suffixes at the end.

7.6 Sample code for memory and time calculation

Python

```python
import sys
from resource import *
import time
import psutil

def process_memory():
    process = psutil.Process()
    memory_info = process.memory_info()
    memory_consumed = int(memory_info.rss/1024)
    return memory_consumed

def time_wrapper():
    start_time = time.time()
    call_algorithm()  # Replace with your algorithm function call
    end_time = time.time()
    time_taken = (end_time - start_time)*1000
    return time_taken
```

Listing 6: Python Time and Memory

Java

```java
private static double getMemoryInKB() {
    double total = Runtime.getRuntime().totalMemory();
    return (total - Runtime.getRuntime().freeMemory()) / 1024;
}

private static double getTimeInMilliseconds() {
    return System.nanoTime() / 1000000.0;
}

double beforeUsedMem = getMemoryInKB();
```

```java
double startTime = getTimeInMilliseconds();
String[] alignment = basicSolution(firstString, secondString, delta, alpha);
double afterUsedMem = getMemoryInKB();
double endTime = getTimeInMilliseconds();
double totalUsage = afterUsedMem - beforeUsedMem;
double totalTime = endTime - startTime;
```

Listing 7: Java Time and Memory

C/C++

```cpp
#include <iostream>
#include <sys/resource.h>
#include <errno.h>
#include <stdio.h>
#include <sys/time.h>

extern int errno;

// getrusage() is available in Linux. Your code will be evaluated in Linux OS.
long getTotalMemory() {
    struct rusage usage;
    int returnCode = getrusage(RUSAGE_SELF, &usage);
    if(returnCode == 0) {
        return usage.ru_maxrss; // In KB
    } else {
        // It should never occur. Check man getrusage for more info to debug.
        // printf("error %d", errno); // Use perror for error messages
        return -1;
    }
}

int main() {
    struct timeval begin, end;
    gettimeofday(&begin, 0);
    //write your solution here
    //Please call getTotalMemory() only after calling your solution function. It
        calculates max memory used by the program.

    double totalmemory = getTotalMemory();

    gettimeofday(&end, 0);
    long seconds = end.tv_sec - begin.tv_sec;
    long microseconds = end.tv_usec - begin.tv_usec;
    double totaltime = seconds * 1000 + microseconds / 1000.0;

    printf("%f\n", totaltime);
    printf("%f\n", totalmemory);
    return 0;
}
```

Listing 8: C/C++ Time and Memory