CSCI-570 Spring 2024

Final Project

Due: May 9, 2024

The project is on the implementation of the two different solutions discussed in class for the *Sequence Alignment* problem. You can work in groups of 3 people or less. Plagiarism of any kind will be strictly monitored & reported as violation of academic integrity.

I. Project Description

Implement the two solutions to the *Sequence Alignment* problem - the basic version using Dynamic programming, and the memory efficient version which combines DP with Divide-n-conquer. Run the test set provided and show your results.

A. Problem review

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: for (i, j), $(i', j') \in M$ if $i < i'$, 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.

The similarity between X and Y is based on the optimal alignment between X and Y. Suppose M is a given alignment between X and Y:

1) First, there is a parameter $\delta > 0$ that defines a gap penalty. Each position of X or Y that is not matched in M is a *gap*, and we incur a cost of δ for it.
2) Second, for each pair of letters p, q in our alphabet, there is a mismatch cost of α_{pq} for matching p with q. Thus, for each $(i, j) \in M$, we pay the appropriate mismatch cost α_{pq} where $p = x_i$ and $q = x_j$. 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 assumption will not be necessary in anything that follows.

3) Finally, the cost of *M* is the sum of all of its gap and mismatch costs, and we seek an alignment having the minimum cost.

B. Input string Generator

In order to potentially work with long strings as test inputs, we use a string generation method. Thus, any input to the program would be generated using a provided text file as follows:

1. First line has a *base* string - let's call it s_0.
2. Next *j* lines consist of numbers that correspond to j steps we execute iteratively, each generating a new string, say strings s_1, s_2, …, s_j , across the j steps, as follows. In step 1, we take the given integer n_1 in line 1 (among the j lines), and insert a copy of s_0 within itself, right after its index n_1 (assuming 0-indexing). This gives us the string s_1. Then, in step 2, we see the integer n_2 in line 2, and insert s_1 within itself right after its index n_2, giving us string s_2. Repeating this for j steps gives us s_j which is the first string in our input for the alignment problem. Note that each step doubles the length of the string generated, thus, $len(s_j) = 2^j * len(s_0)$. See an illustrative example given below.
3. The subsequent lines contain a base string t_0 followed by k lines each containing a number. We use them as above to iteratively generate strings t_1, t_2, …, t_k, with t_k being the second string input for the alignment problem.

Thus, this text file could be used as an input to your program and your program should use the generation method above to get the actual input strings to be aligned. Note that *j* need not equal *k* for the string generation shown above.

Consider the following input as example:

ACTG

3

6

1

TACG

1

2

9

Following is the step by step process on how the final two strings are generated.

First base string: ACTG

Insertion after index 3: ACTGACTG

Insertion after index 6: ACTGACTACTGACTGG

Insertion after index 1:ACACTGACTACTGACTGGTGACTACTGACTGG

Similarly,

TACG

TATACGCG

TATTATACGCGACGCG

TATTATACGCTATTATACGCGACGCGGACGCG

Thus, using the inputs above, the generated strings are
ACACTGACTACTGACTGGTGACTACTGACTGG and
TATTATACGCTATTATACGCGACGCGGACGCG which now need to be aligned.

C. Values for Delta and Alphas

Values for α's are as follows. δ 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

D. Programming/Scripting Languages

Following are the list of languages which could be used:

1. C
2. C++
3. Java
4. Python2
5. Python3

E. Bounds

1) Basic Algorithm

   $0 <= j, k <= 10$

   $1 <= len(s_0), len(t_0) <= 2000$

   The values will also ensure that $1 <= len(s_i), len(t_k) <= 2000$

2) Memory Efficient Algorithm

   $0 <= j, k <= 20$

   $1 <= len(s_0), len(t_0) <= 20000$

   $1 <= len(s_i), len(t_k) <= 20000$

II. Final output and submission format

A. 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)

```
`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 Part II-B, input file will have data to generate input strings. Since Gap penalty (δ) and Mismatch penalties (α_{pq}) are **FIXED**, you have to **hardcode** them in your program.

You **are not** allowed to use any libraries that may be directly applicable to sequence alignment problem.

B. Implement the basic version of the solution. Your program should print the following information at the respective lines in 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)

N**ote:** There can be multiple solutions which have the same cost. You can print ANY of them as generated by your program. The only condition is, it should have a minimum cost.

E.g., For strings A and C, alignments A_, _C and _A, C_ both have alignment cost 60 which is minimum (indeed, in this case, they represent the same matching in two different ways). You can print any one of them.

C. Implement the memory efficient version of this solution and repeat the tests as in Part B and produce output in the same format.

D. Plot the results of Part B and Part C 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

III. Submission

A. 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) / '*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) / '*efficient_3.py*' (Python 3)

c. Summary.pdf

It must contain following details

1. datapoints output table (which are generated from provided input files)
2. Two graphs and your insights from them
3. Contribution from each group member, e.g., coding, testing, report etc. if everyone 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 flexibility in passing any additional compiler/run arguments that your programs might need. See More Hints (VII part E for more details)

basic.sh

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

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

./efficient.sh input.txt output.txt

B. The name of your zip file should have the USC IDs (not email ids) of everyone in your group separated by underscore. The zip file structure will look like

- 1234567891_1234567892_1234567893.zip
  - 1234567891_1234567892_1234567893
    - basic_2.py
    - efficient_2.py
    - Summary.pdf
    - basic.sh
    - efficient.sh

IV. Grading

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

A. Correctness of algorithms - **70** points

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

Note: The goal of Part A is to check the correctness of alignment having minimum cost. Memory and Time will be evaluated in Part B.

B. 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 reasonably match what you submitted in the Summary.pdf

2. Correctness of the graph
3. Correctness of analysis/ insights

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

V. What is provided to you in the zip file?

A. SampleTestCases folder containing sample input and output files
B. Datapoint folder containing files to generate graph data points.
C. Summary.docx file as template

VI. HINTS, NOTES, and FAQs

A. Regarding Input and string generation

1. We will never give an invalid input to your program. Input strings will only contain A, C, G, T. The insertion indices in the input will be valid numbers.
2. The string generation mechanism is the same irrespective of the basic or the efficient version of the algorithm.
3. The entire program (string generation, solution, write output) should be written in a single file. You may break those functions in different classes to make the code modular, but there should be only one file for consistency of submissions.

B. Regarding Algorithm and output

1. We strongly recommend to refer to lecture slides for the algorithm overview, and NOT THE PSEUDOCODE PROVIDED IN KLEINBERG AND TARDOS textbook. In our prior experience, students opting for the latter reported lots of difficulties in implementation.
2. DO NOT USE ANY LIBRARIES FOR WRITING YOUR ALGORITHMS barring the *standard* ones. If you are really unsure, ask us on piazza.
3. Samples for time and memory calculation are provided. Please use them for consistency.
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 of the sub-problems, otherwise the plots will not show the expected distinctions.
7. Your program must not print anything when it runs, 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(logn) etc.

C. Regarding the plot

1. Both the graphs are line graphs. X-axis represents problem size as m+n, where m and n are lengths of the generated input strings. Y-axis of Memory plot represents memory in KB. Y-axis of 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

D. Regarding Submission

1. Only 1 person in the group should submit the project. We will 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 will kill your program if it is stuck on a single input file for long (~few minutes).

E. 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 ./a.out "$1" "$2"
efficient.sh	gcc efficient.c ./a.out "$1" "$2"

C++

basic.sh	g++ basic.cpp ./a.out "$1" "$2"
efficient.sh	g++ efficient.cpp ./a.out "$1" "$2"

Java

basic.sh	javac Basic.java java Basic "$1" "$2"
efficient.sh	javac Efficient.java java Efficient "$1" "$2"

python 2.7

basic.sh	python2 basic_2.py "$1" "$2"
efficient.sh	python2 efficient_2.py "$1" "$2"

python 3

basic.sh	python3 basic_3.py "$1" "$2"
efficient.sh	python3 efficient_3.py "$1" "$2"

Note that the above are just examples. You can modify them as per your convenience. The goal is to have a language-independent mechanism to get your outputs.

Also note that for python2 or python3, it is important to have 2 or 3 suffix at the end.

F. Sample code for memory and time calculation

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()
    end_time = time.time()
    time_taken = (end_time - start_time)*1000 return time_taken
```

Java

```
private static double getMemoryInKB() {
    double total = Runtime.getRuntime().totalMemory(); return
    (total-Runtime.getRuntime().freeMemory())/10e3;
}
```

```
private static double getTimeInMilliseconds() { return
    System.nanoTime()/10e6;
}
```

```
double beforeUsedMem=getMemoryInKB(); double startTime
= getTimeInMilliseconds();

alignment = basicSolution(firstString, secondString, delta, alpha);

double afterUsedMem = getMemoryInKB(); double endTime =
getTimeInMilliseconds();

double totalUsage = afterUsedMem-beforeUsedMem; double totalTime =
endTime - startTime;
```

C/C++

```
#include <sys/resource.h> #include
<errno.h> #include <stdio.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;
    } else {
        //It should never occur. Check man getrusage for more info to
debug.
        // printf("error %d", errno); 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*1e-3; printf("%f\n", totaltime);
    printf("%f\n", totalmemory);
}
```