PLAGIARISM CHECKER

-Satwik Banchhor
-2018CS10385

Algorithm Gist:

I used a space-optimized dynamic programming based algorithm to detect similarity between the sections of the 2 texts. I defined a similarity score for each section of the text and defined a cutoff based on which I decide whether a section of the text is plagiarized or not. In the end I took union of all the plagiarized sections of the input text and found the percentage of the input file that is plagiarized.

Details:

Hashing:

First I took each word of the text and hashed it with a hash function that took O(n) time to hash a word with n characters. Further capital and small letters were considered same for the purpose of hashing, and characters other than english letters and decimal numbers were ignored while hashing (as addition of a number of such letters could be used to disguise plagiarism). Since we have only **26 (englilsh alphabet) + 10 (decimal numbers) = 36** the appropriate base for hashing was chosen as 37 and mod as $10^9 + 9$

Hash Function:

Consider a string s = s[0..n]
the hash value of the string s is
hash_value = (s[0] * base^n + s[1] * base^n + . . . + s[n] * base^0) % mod
where, **base = 37** and **mod = 10^9 + 9**

After hashing each word into an integer I stored the integers in an array, this was for the input file and one-by-one for all the text files in corpus_files folder. Then the two arrays of integers were compared for checking plagiarism.

Dynamic Programming:

I maintined the following 3 matrices for every i, j where 1 <= i <= lenght(input text) & 1 <= j <= length(text for checking) i.e. for each combiantion of the prefixed I calculated the following (prifix of length i for the input and length j for the comparison texts):

1. The maximum similarity score matrix (S[i][j]):
   The similarity score of input text[Oi[i][j] . . i] and some section of the comparison string that ends at j is given by S[i][j], since we are only concerned with the section of the input string that is plagiarized we do not need to remember the exact section of the comparison string from where the matching score is measured (which is why we do not have an Oj[i][j]).

The similarity score of each combination is 0 if any 1 of them has lenght 0 and the similarity score is never less than 0.
Each matching makes a positive contribution to the score, and each symbol that has to be inserted, deleted or substituted to transform X 0 to Y 0 should make a negative contribution.

2. The Previous Maximum (M[i][j]):
   The previous maximum represents the maximum similarity score in the trace back from i, j to any of its origin.
3. The Origin in input after which there is similarity (Oi[i][j]):
   For any i, j the value S[i][j] represents the similarity score of the section of Oi[i][j] . . i of the input string with some sectin of the comparison string (contiguous) that we choose not to remember as we are concerned only with the percentage of the similarity in the input string.

Contrubution to similarity score:

If 2 symbols are same (i.e. the i^{th} word of the input is the same as the j^{th} word of the comparison text): +1
Insertion: -1
Deletion: -1
Replacement: -1

Predomination:

A similarity score is said to be pre-dominated if the value of M[i][j] is greater than the value of S[i][j] which means that the sections were more similar earlier in the trace back and not they have become less similar.

Cutoff:

If the M[i][j] – S[i][j] becomes equal to the CUTOFF value then we set S[i][j] to 0 as the in the string sections similarity has reduced enough for us to consider the strings as dissimilar, further M[i][j] is set to 0 and Oi[i][j] is set to i, because from here on we will measure the similarity from this position in the input text and we decided the previous section to be dissimilar enough.

Measure of Similarity:

For every i, 1 <= i <= n where n is the size of the input we check the section that has the maximum similarity score, if the similarity score for 2 secitons is the same then we choose the section which begins later (i.e. has larger Oi[i][j] => is shorter) because the larger section must have had more dissimilarities as it gives the same score as the shorter section and is not predominated i.e. it should be the largest in its traceback: **This section is the shortest section of the input string that ends at index i (i^{th} word) and has the largest similarity score**.
Plagiarism Check: For such a section if the similarity score is greater than the CUTOFF value then I consider the section from (origin + 1) to index (i) of the input text as plagiarized.

In the end I take the union of all the plagiarized sections for all i (if they exist for any i).

Taking union of plagiarized sections:

I maintain an array of lenght n (plagiarized_sections array) which is initialized with 0s.
For each i if we detect a plagiarized section then we mark it in an the plagiarized sections array by adding 1 to the index where the section the section begins and -1 to the index after the index where the section ends i.e. (i + 1).
At the end I take the prefix sum of the plagiarized sections array and if any index has a non-zero value then that index (word in the text) is considered plagiarized.

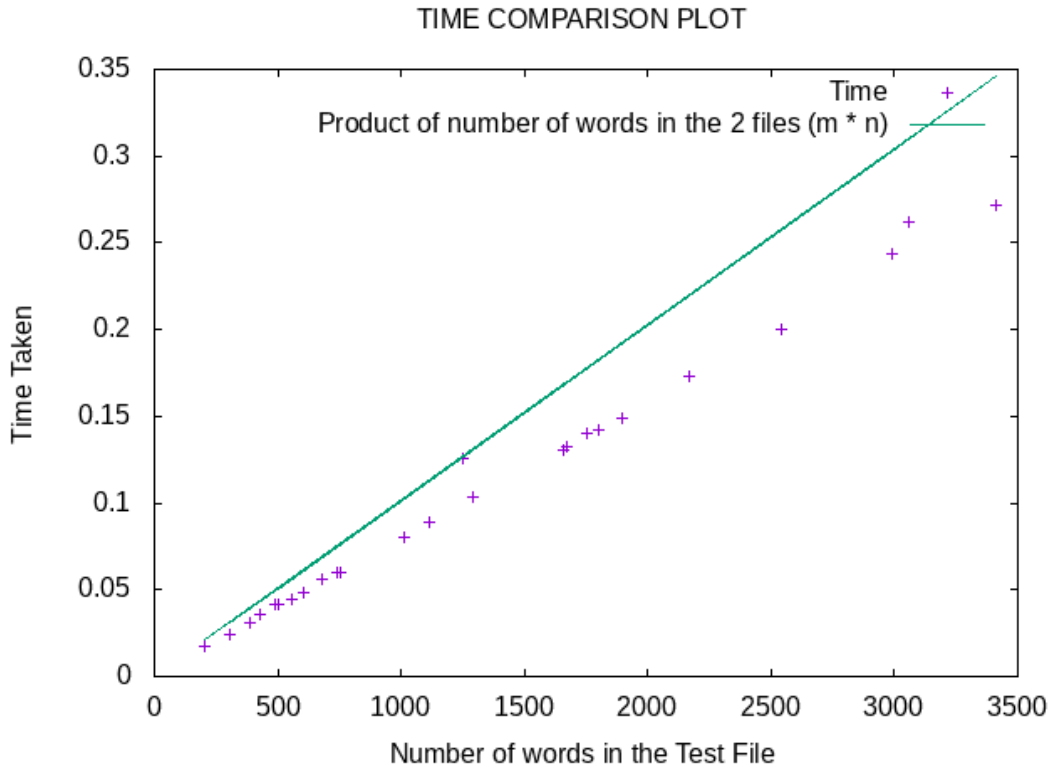
Quantification of plagiarism:

After the union is taken I know which word is plagiarized and which isn't so depending on the lenght of the word I calculate its contrubution to the text and count it as plagiarized and calculate the persentage of plagiarism.

Time Complexity:

In the above algorithm we are filling the dynamic programming table of size m*n and we take O(1) time to fill each cell. Therefore the time complexity of the algorithm for similarity comparison is
Time Complexity: **O(m * n)** where,
n is the number of word in the input text and
m is the number of words in the comparison text

The above analysis can be proven by the following GNU plot that compares the time taken by the algorithm to test an **input file of size 1800** with comparison files of various sizes



Space Complexity:

Note that to fill the 3 table that we are using to detect plagiarism in the text files we only need the previous row to fill the next row therefore there is scope for space optimization i.e. we need not store the entire table and store only 2 rows: the row that we filled previously and the row that we have to fill now,

Therefore the **space taken by the DP tables is: O(m)** where m is the size of the comparison text.

Further we need to store the **hashes input text and the hashed comparison text** requiring **O(m + n)** space.

Therefore the overall space complexity is: **O(m + n).**