Program 3: Maximal Substring

Members: Daniel Deal, Mikhal Filippov, Samuel Hohenstein

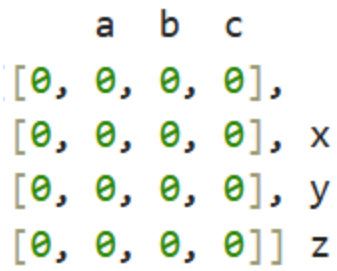
In this programming assignment, we are to use the ideas of **dynamic programming** in order to optimize the `MaximalSubstring` algorithm. We will then test it, prove its correctness, and to runtime analysis.

Solution Approach

In our solution we use a similar method to memoization in recursion, but use loops instead of the recursion. The lengths array allows for the program to prevent repeated computations, and allows for following logic to use the values from the array already computed. The lengths array is made based on the sizes of the strings A and B, where if A was 4 letters, and B was 5 letters, the lengths array will have 5 (4+1) arrays of 6 (5+1) 0s when initiated. The first array and first index of each array are there to allow for the actual first letters of the comparisons to have something to be built off of, as for each following letter check, if the letters are the same, they add 1 to the sum of the previous value stored in the previous array and previous index. Using this when there are multiple letters inside of a word, it still allows for individual letter checks to not interfere with each other, and allows for us to save the longest substring and how long it was. When a new longest substring is found, we record how long that substring is and where it ends, which also allows us to compute where the substring starts. This substring is then what is returned at the end of our function.

```
     p  i  n  e  a  p  p  l  e
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], s
 [0, 0, 0, 0, 0, 1, 0, 0, 0, 0], a
 [0, 1, 0, 0, 0, 0, 2, 1, 0, 0], p
 [0, 1, 0, 0, 0, 0, 1, 3, 0, 0], p
 [0, 0, 0, 0, 0, 0, 0, 0, 4, 0], l
 [0, 0, 0, 0, 1, 0, 0, 0, 0, 5]] e
```

In this picture string A (sapple) is compared to string B (pineapple), and the length of substrings is displayed in a matrix look at lenghts. e * e is the longest substring length is 5, where end length would be 6 (sappl(e)), using e and the previous 4 letters, the longest substring would be computed to be "apple".



In this picture string A (xyz) is compared to string B (abc), and the length of substrings is displayed in a matrix look at lenghts. The longest substring is 0, where end length would be 0 ((x)yz), using no letters making the longest substring computed to be "".

Pseudocode

```
FUNCTION MaxSubstring(A, B):
    n = A.length
    m = B.length
    lengths = [ [0]*(m+1) ] * (n+1)
    max_length = 0
    end_index = 0

    FOR i = 1...n DO:
        FOR j = 1...m DO:
            IF A[i-1] == B[j-1] THEN:
                lengths[i][j] = lengths[i-1][j-1] + 1

                IF lengths[i][j] > max_length THEN:
                    max_length = lengths[i][j]
                    end_index = i
                END IF

            ELSE:
                lengths[i][j] = 0
            END IF
        END FOR
    END FOR

    start_index = end_index-max_length
    RETURN A[start_index:end_index]
```

Proof of Correctness

- ☑ always generates a correct result when it halts (end of string comparisons)
- ☑ proven to terminate

The algorithm is proven to be correct when it halts as it checks every substring comparison between the two strings by recording the place of the individual substring comparisons and

building upon it. The first longest substring will be returned upon termination, which is what the Maxmimal Substring function seeks to accomplish. This algorithm will always terminate once it has looped through each letter in A, and compared it to each letter in B. Once this is accomplished, the loops terminate and the longest substring is calculated and returned.

More formally, we can define the **invariant** to be: At any `(i, j)`, `lengths[i][j]` contains the length of the longest common substring ending at `A[i-1]` and `B[j-1]`.

Base case states: When `i = 0` and `j = 0`, `lengths[i][j] = 0` will hold true because empty strings do not have any matching substrings.

Maintenance states: We assume, through strong induction, that all `k` and `g`, for `k < i` and `g < j`, are correct. Then, at `i` and `j`, we reach two cases:

1. If `A[i-1] == B[j-1]`, the longest substring adds on at `lengths[i][j]` by using the previous `lengths[i-1][j-1]`, which are included in `k` and `g` as stated in the strong induction. Furthermore, we compare `max_length` against the current `lengths[i][j]` and store that in `max_length` if the current length is longer, storing the end index as well.
2. If `A[i-1] != B[j-1]`, the previous substring ends and we update `lengths[i][j] = 0`.

Termination: `k` and `g` eventually reach `i` and `j`, proving the loops terminate as they have a set end condition to count up towards.

Test Cases

TEST CASE	EXPECTED	ACTUAL
Common Test: `"pineapple", "apple"`	`"apple"`	`"apple"`
Common Test: `"banana", "apple"`	`"a"`	`"a"`
Common Test: `"orange", "banana"`	`"an"`	`"an"`
No Common Test: `"xyz", "abc"`	`""`	`""`
Identical Test: `"hello", "hello"`	`"hello"`	`"hello"`
Single Character: `"x", "x"`	`"x"`	`"x"`
Single Character: `"x", "y"`	`""`	`""`
At Start: `"abcdef", "abc"`	`"abc"`	`"abc"`
At End: `"abcdef", "def"`	`"def"`	`"def"`
In Middle: `"abcdef", "cde"`	`"cde"`	`"cde"`

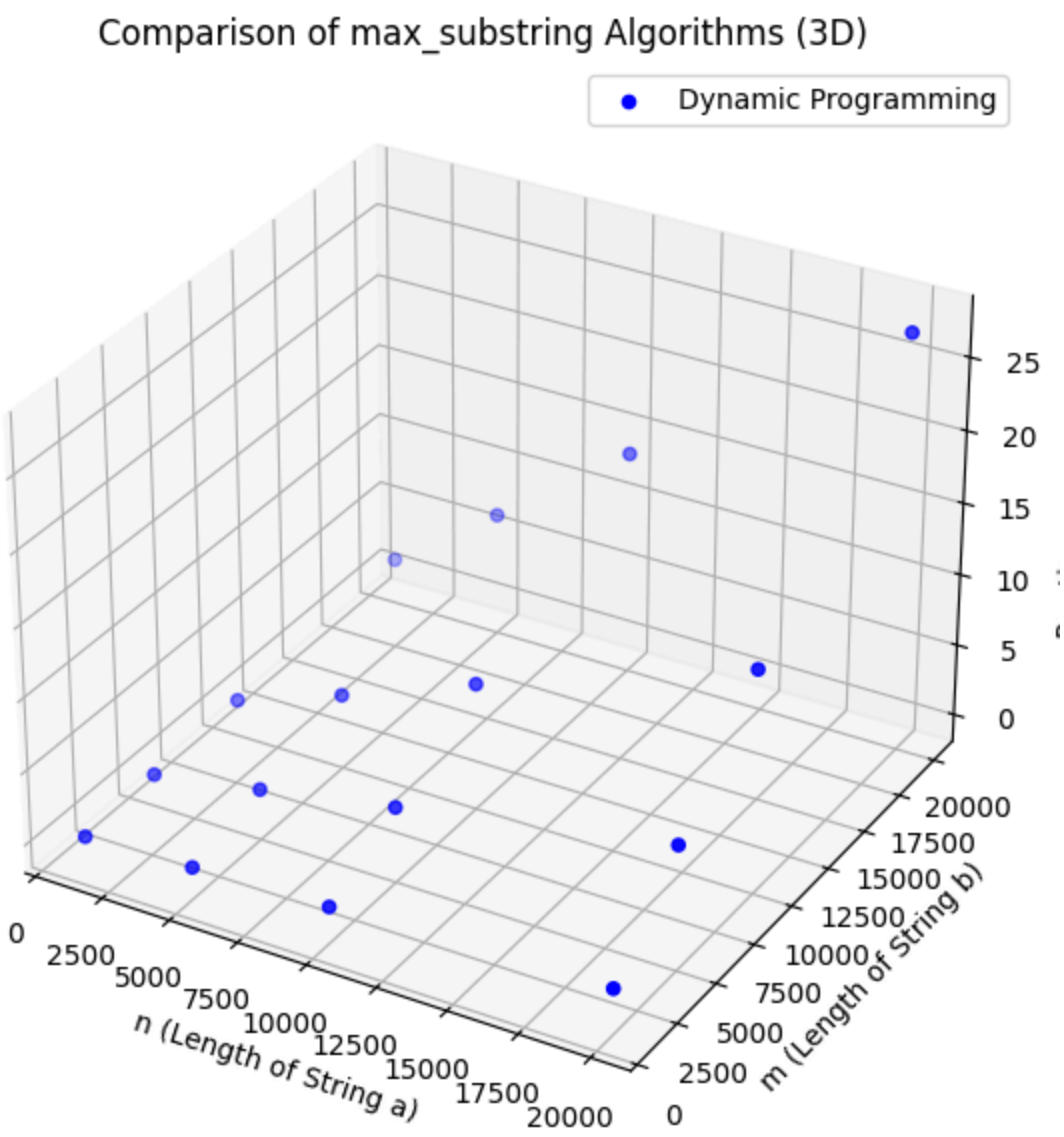
TEST CASE	EXPECTED	ACTUAL
Large String: `"a" * 1000 + "bcd", "xyz" + "bcd"`	`"bcd"`	`"bcd"`
Empty Strings: `"", "abc"`	`""`	`""`
Empty Strings: `"abc", ""`	`""`	`""`
Empty Strings: `"", ""`	`""`	`""`

Runtime Analysis

The runtime for this algorithm is trivial. We have a few initializations, then a loop looping from `1` to `n + 1` (or, `n` times). Then, for each iteration of that loop, another loop loops from `1` to `m + 1` (or, `m` times). Inside that second loop are just constant operations. So, for each `n`, we (asymptotically) run `m` operations, meaning the runtime for this algorithm is `O(n * m)`.

Benchmarking Analysis

Benchmarking *only* our algorithm and graphing the results on a 3d graph (n as x, m as y, and runtime for the input as z), we see the following:



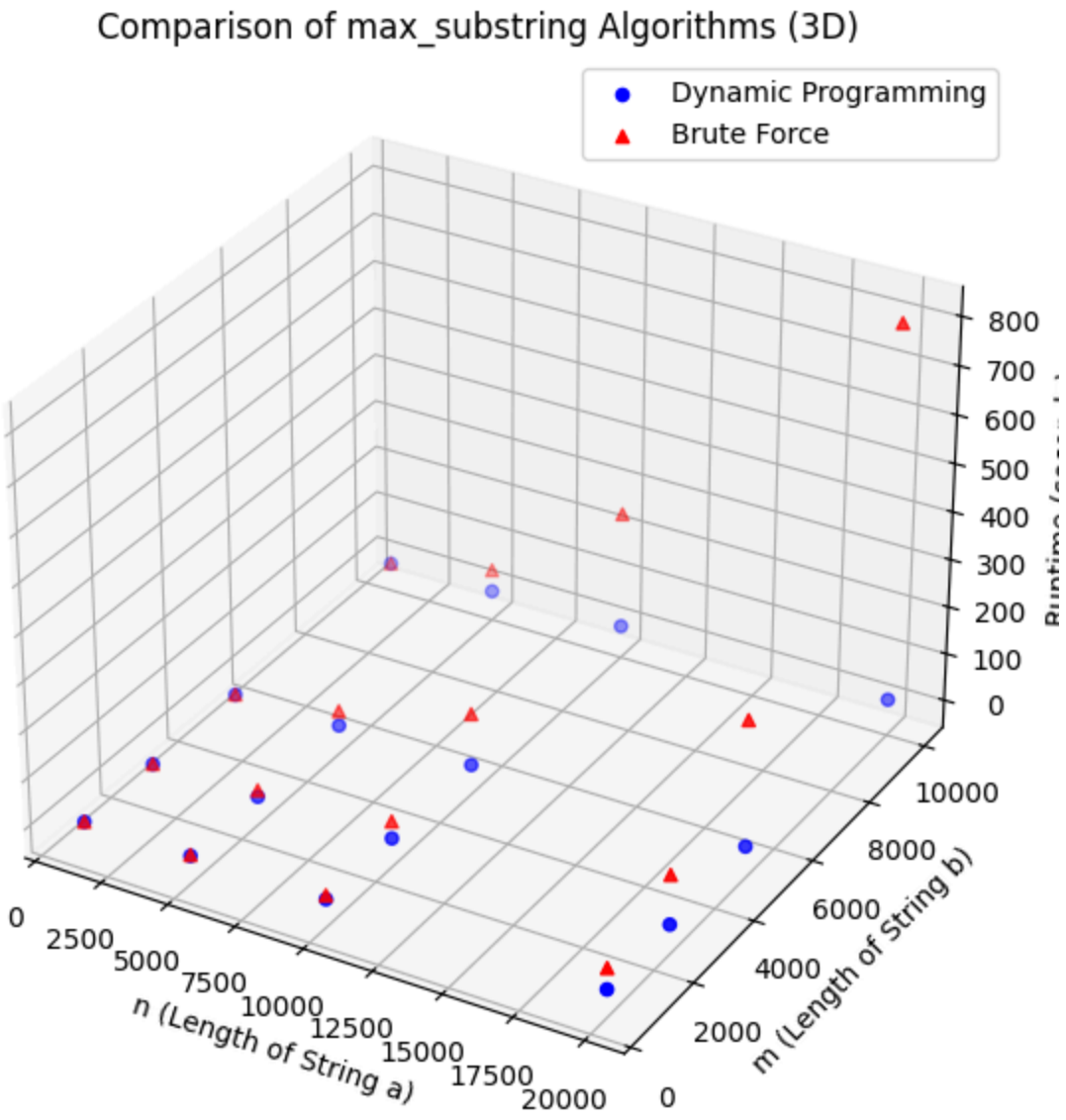
The image got a bit cut off (not sure why), but it is obvious to see that the actual runtime of the algorithm matches our theoretical runtime of `O(n * m)`.

Benchmark Results Table:

Length of a (n)	Length of b (m)	DP Time (s)
1000	1000	0.061971
1000	5000	0.338203
1000	10000	0.646735
1000	20000	1.268512
5000	1000	0.319783
5000	5000	1.642981
5000	10000	3.297931

Length of a (n)	Length of b (m)	DP Time (s)
5000	20000	6.691123
10000	1000	0.634248
10000	5000	3.365077
10000	10000	6.946464
10000	20000	13.723189
20000	1000	1.289206
20000	5000	6.793251
20000	10000	13.657638
20000	20000	27.163658

To further enforce this, we can graph (and compare) runtimes of a brute force implementation (in the appendix under `Brute Force Implementation`):



The brute force implementation is obviously `O(n^2 * m)` asymptotic runtime (explained in the `Problem` section of this assignment), and that is reflected in the graph of the algorithm. Meanwhile, our implementation sits comfortably lower than the brute force implementation, showing effective use of memoization to optimize the runtime.
