Anthony Gringeri
Algorithms: Project 1
Executive Summary Report

Introduction

For this project, two algorithms were written and analyzed to determine whether two given strings are anagrams. If two strings are anagrams, then they contain the same letters – but not necessarily in the same arrangement. The first algorithm was written using a brute force approach, meaning that the algorithm is written in such a way that it exhausts all possibilities before concluding. The second algorithm was one that I designed myself, which is much more efficient than using a brute force approach. My algorithm has a hash map for each lowercase letter in the alphabet (a-z) and stores the counts of each letter respectively.

Summary of time and space efficiency analysis

Analyzing the brute force algorithm was difficult because of modern computational limits. I was not able to test strings longer than eleven characters because of the resource intensiveness of the brute force approach. The brute force algorithm that I wrote creates a list of all possible permutations of the first given string, and then traverses the list one-by-one, comparing it to the second given string. The brute force algorithm that I wrote also does not stop when it finds a match; the algorithm keeps track of how many matches there were and will continue to traverse the list until it ends. In theory, this algorithm is in the efficiency class of $O(n!)$. This is due to the nature of permutations. A permutation is calculated using the formula: $P(n,r) = \frac{n!}{(n-r)!}$ where n is the number of objects (characters) and r is the sample (number of characters to consider). In my brute force algorithm, n and r are always the same, because it only produces strings that are of the same length as the given string. Thus, simplifying the equation gives $P(n,r) = \frac{n!}{1!} = n!$, which is the efficiency class of the brute force algorithm. The space complexity of the brute force algorithm is the same, because it creates a list of all permutations before comparing them. This massive consumption of space was made apparent when my laptop began to give warnings implying that space on my hard drive was running out when I tested large strings. Put simply, the space complexity class of the brute force approach is $O(n!)$.

Analyzing the algorithm that I wrote proved to be much easier in practice. I could test strings up to 1000 characters in size in well under a second. In fact, I ran into the exact opposite problem testing this algorithm, because the terminal I was using was unable to input large enough strings to test the algorithm with much larger strings. Still, I could collect enough data to conclude that my algorithm was in the efficiency class of $O(n)$. My algorithm creates a hash table with every character (a-z) as keys, mapping to the number of occurrences in each word. It iterates through every character in the given string, and adds an occurrence accordingly. After both hash tables are constructed and both strings are iterated through, it compares the hash tables. If they are the equal, then the strings are anagrams, and vice versa. The space complexity of my algorithm is $O(1)$, because the size of the hash table is constant for both strings. It never adds or deletes a mapped entry in either of the hash tables.

Pseudocode

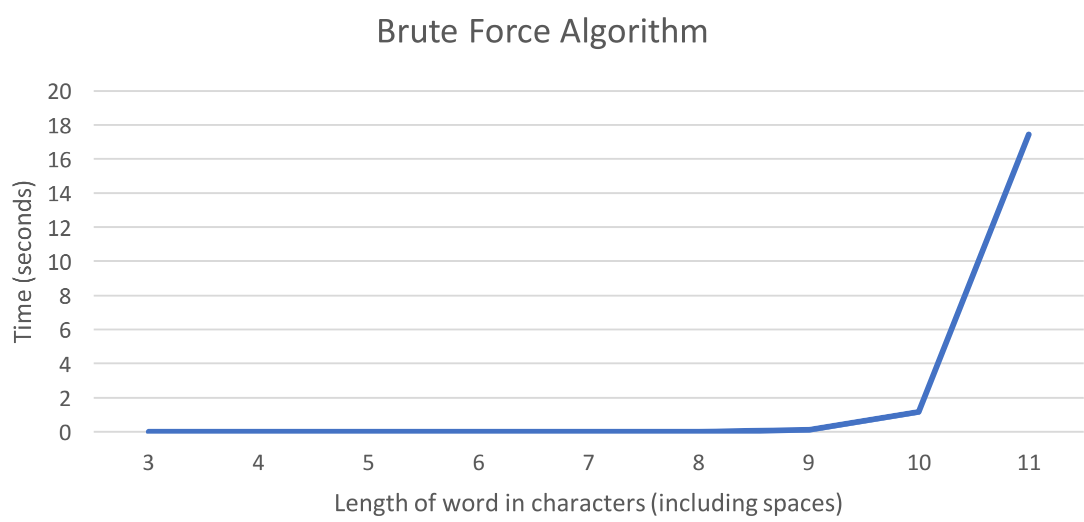
For the brute force algorithm:

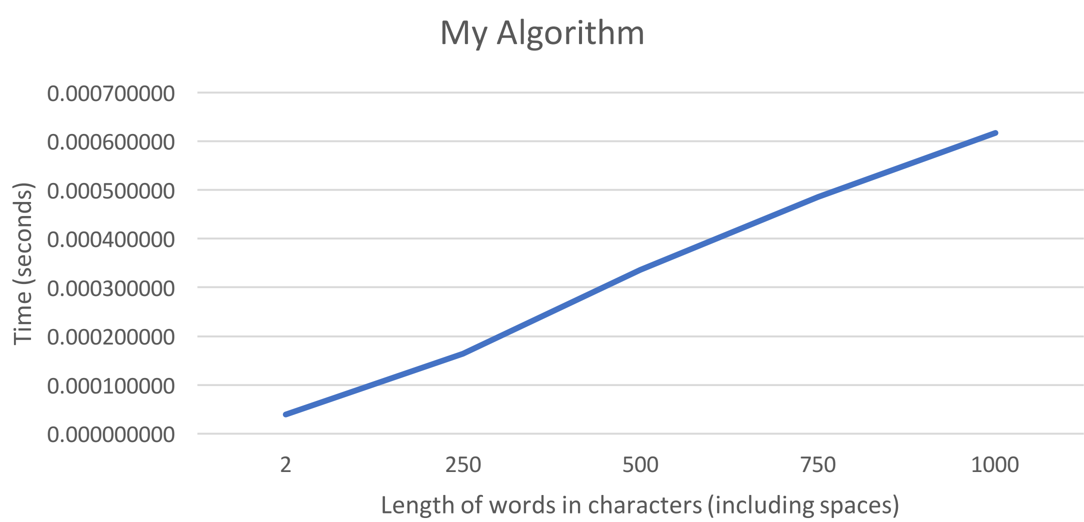
```
BruteForceAlg(s1, s2)
    matches = 0
    permutations = permutations(s1)
    for x in permutations:
        if x == s2:
            matches++
    if matches > 0:
        return true
    else:
        return false
```

For my algorithm:

```
MyAlgorithm(s1, s2)
    if (length(s1) != length(s2)):        // does not include spaces
        return false
    counts1 = hashmap<char(a-z), occurences>
    counts2 = hashmap<char(a-z), occurences>

    for c1 in s1:
        if c1 == (a-z)
            counts1.get(char) + counts1.get(char) + 1

    for c2 in s2:
        if c2 == (a-z)
            counts2.get(char) + counts2.get(char) + 1

    if counts1 == counts2:
        return true
    else:
        return false
```

Graphs





Looking at the graphs, it is very apparent that the brute force algorithm resembles a $O(n!)$, and my algorithm resembles a linear graph, or $O(n)$.

Explanation of timing algorithms

To time my algorithms, I used the timeit class in Python 3. In detail, I initiated a timeit.default_timer() (for portability) before the algorithm started, and another after it ended. To report the computation time, I simply subtracted the start time from the end time.

Choosing a sample size *n*

I chose my sample size simply by trial and error. This task was simple because I could not sufficiently test any strings over size eleven for the brute force algorithm, and could not test any strings over size 1024 for my algorithm due to limits in my command line terminal. This limited my possible sample size significantly, making the task of choosing a sample size much easier.

Testing

Brute Force Algorithm					
string1	**s1 size**	**string2**	**s2 size**	**anagram?**	**Time (secs)**
abc	3	cba	3	YES	0.000064556
abcd	4	dcba	4	YES	0.000061536
hello	5	bello	5	NO	0.000129562
silent	6	listen	6	YES	0.000407076
rentals	7	antlers	7	YES	0.002343573
roasting	8	organist	8	YES	0.014466582
auctioned	9	cautioned	9	YES	0.110277547
harmonicas	10	maraschino	10	YES	1.152062148
harmonicaso	11	maraoschino	11	YES	17.426338449
harmonicasoo	12	maraoschinoo	12	YES	N/A (Could not compute)
to be or not to be that is the question whether tis nobler in the mind to suffer the slings and arrows of outrageous fortune	124	in one of the bards best thought of tragedies our insistent hero hamlet queries on two fronts about how life turns rotten	121	YES	N/A (Could not compute)
aaaaaaaaaaaaaaaaaaaaa	250	aaaaaaaaaaaaaaaaaaaaaaaaaaaaa		YES	N/A (Could not compute)
aaaaaaaaaaaaaaaaaaaaa	500	aaaaaaaaaaaaaaaaaaaaa	500	YES	N/A (Could not compute)
aaaaaaaaaaaaaaaaaaaaa	750	aaaaaaaaaaaaaaaaaaaaa	750	YES	N/A (Could not compute)
aaaaaaaaaaaaaaaaaaaaa	1000	aaaaaaaaaaaaaaaaaaaaa	1000	YES	N/A (Could not compute)

My Algorithm					
string1	**s1 size**	**string2**	**s2 size**	**anagram?**	**Time (secs)**
abc	3	cba	3	YES	0.000038870
abcd	4	dcba	4	YES	0.000031730
hello	5	bello	5	NO	0.000038420
silent	6	listen	6	YES	0.000050006
rentals	7	antlers	7	YES	0.000049072
roasting	8	organist	8	YES	0.000034997
auctioned	9	cautioned	9	YES	0.000037538
harmonicas	10	maraschino	10	YES	0.000041579
harmonicaso	11	maraoschino	11	YES	0.000200259
harmonicasoo	12	maraoschinoo	12	YES	0.000079730
to be or not to be that is the question whether tis nobler in the mind to suffer the slings and arrows of outrageous fortune	124	in one of the bards best thought of tragedies our insistent hero hamlet queries on two fronts about how life turns rotten	121	YES	0.000334316
aaaaaaaaaaaaaaaaaaaaa	250	aaaaaaaaaaaaaaaaaaaaa	250	YES	0.000163997
aaaaaaaaaaaaaaaaaaaaa	500	aaaaaaaaaaaaaaaaaaaaa	500	YES	0.000335916
aaaaaaaaaaaaaaaaaaaaa	750	aaaaaaaaaaaaaaaaaaaaa	750	YES	0.000485352
aaaaaaaaaaaaaaaaaaaaa	1000	aaaaaaaaaaaaaaaaaaaaa	1000	YES	0.000616766