Mr. Agana, The Anagram Decider
Program Analysis

Samy Masadi

Program Summary

Mr. Agana is a program designed to determine, using two different methods, whether a pair of input strings are anagrams. The user can use it not only to determine anagrams, but also to compare the efficiency of the two methods to each other.

The two anagram deciding algorithms are as follows:

Brute Force:

Search through all possible permutations of the first string supplied and compare each permutation to the second string until a match is found or all permutations are exhausted.

Brute Force Pseudocode:

```
def permute(first string, second string, first character index, last character index):
    if first character == last character:        # base case
        return second string == current permutation
    else:    # recursive case
        for i in range (first index of current recursion level to last index +1):
            swap i character with first character of current recursion level's string
            if (make recursive permute call here) returns True:
                match has been found, return True to level above
            swap i character with first character again to backtrack
        return False if no match was found
```

Linear:

Iterate through each string, and count instances of each letter within. Save the counts in a separate array (one array per string) containing 26 slots corresponding to each letter in alphabetical order. Compare the arrays to each other to determine if the strings are anagrams.

Linear Pseudocode:

```
def letterCount(string):
    initialize counts array with 26 instances of 0 for each alphabetical letter
    for i in range(0 to string length):
        determine i^th character
        determine index in counts array corresponding to the difference of the i^th
            character's ASCII value and the base ASCII value of 'a'
        increment the current sum of i^th character in the proper index in counts array
    return counts array for comparison
```

Program Efficiency

Brute Force Worst Case Time Efficiency: **O(n!)**

```
def permute(a, b, l, r):                          Const   Time
    if l==r:                                      c1      n!
        return b == toString(a)                   c2      n!
    else:                                         c3      n!+1
        for i in range(l,r+1):                    c4      (n+1)!
            a[l], a[i] = a[i], a[l]               c5      n!
            if permute(a, b, l+1, r):             c6      n!
                return True                       c7      1
            a[l], a[i] = a[i], a[l] # backtrack   c8      n!
        return False                              c9      n!
```

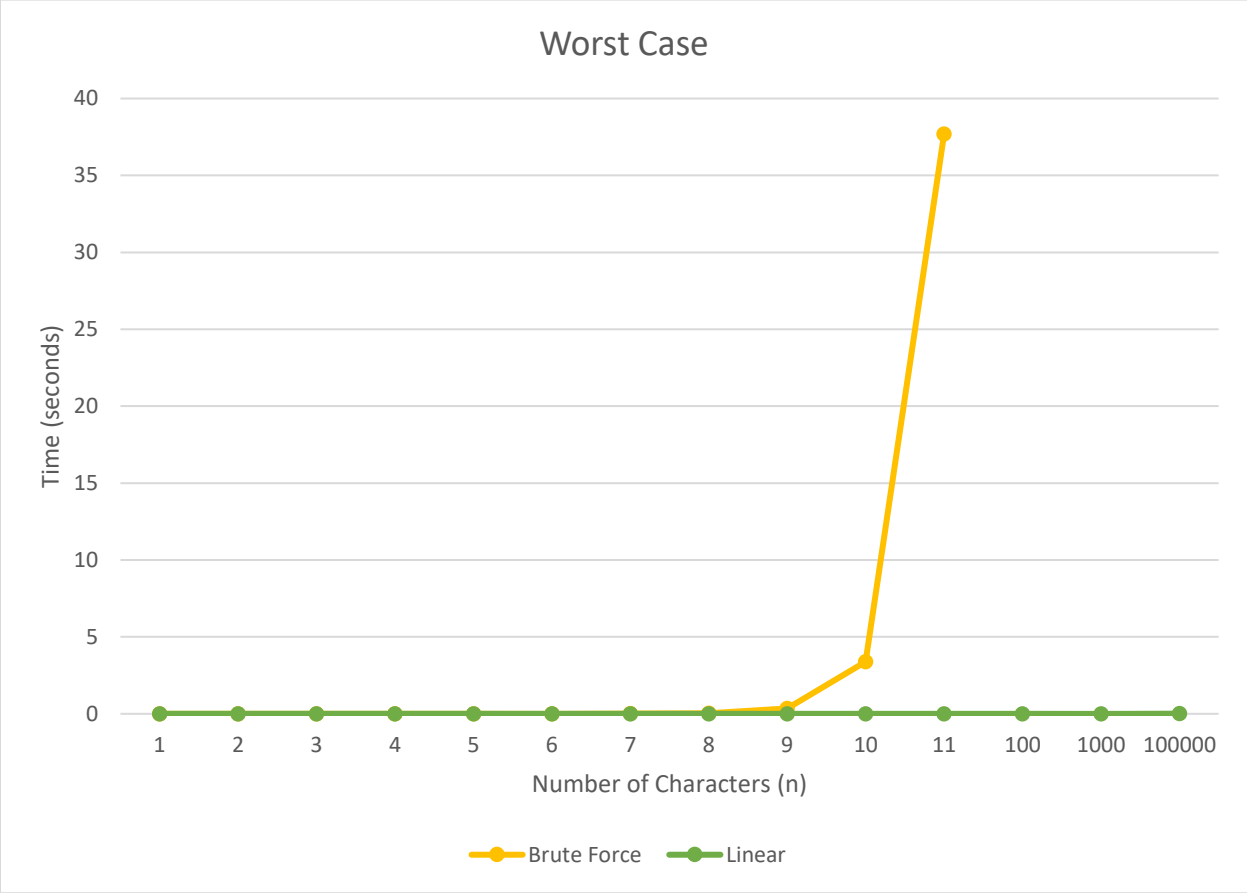
The worst case occurs when the two input strings are not anagrams, and thus the brute force algorithm must permute through every ordering of the first string's letters. Examining the permute function at the core of algorithm reveals recursive code that gets called the factorial of the number of characters, n. Each level of recursion changes l and r indices for iterating and swapping, and calls recursively until l and r are equal. Character swaps thus occur in a patter of n * (n-1) * (n-2) * … * 1, and generates n! permutations. The brute force algorithm therefore has an order of growth of O(n!).

Linear Worst Case Time Efficiency: **Θ(n)**

```
def letterCount(string):                  Const   Time
    counts = [0] * 26                     c1      1
    for i in range(0, len(string)):       c2      n+1
        ch = string[i:i+1]                c3      n
        ind = ord(ch) - 97                c4      n
        counts[ind] += 1                  c5      n
    return counts                         c6      1
```

For the linear algorithm, the worst case also occurs when the input strings are not anagrams; however, it only needs to iterate through n number of characters for each string to tally the character counts. When the counts arrays for each string are compared, Python will internally visit and compare the values in each of the 26 array spaces. In the counts comparison, iteration thus also occurs, but always at a constant of 26. As linear's best case will show, its efficiency scales linearly with n regardless of input, so it has an order of growth of Θ(n) as both an upper and lower bound.

Worst Case Time Comparison:



Timing each algorithm meant including calls to a Python time function in the program, marking the start and end of algorithm execution. I then could record and compare times taken for each in seconds.

To test the worst case, I chose values of n characters starting at 1 and continuing until the time for the brute force algorithm became too prohibitive to test. Only for the linear algorithm did I continue on, and specifically chose increasingly large values of n to see how the algorithm can scale with such large values.

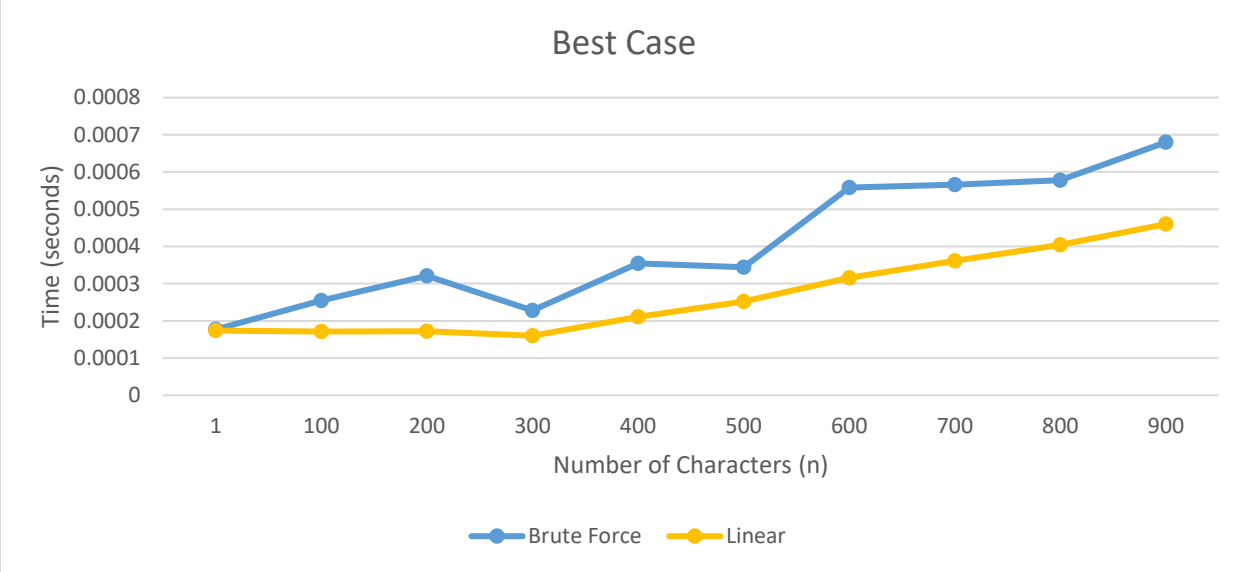
As far as the data itself, both algorithms have comparable efficiency for strings up to 5 characters in length. On my test system, lengths of 6 and beyond showed the brute force pull away from the linear, and values greater than 10 revealed brute force's practical limits. The linear algorithm only started scaling at lengths greater than 100 characters, so larger values of 1000 and 100000 were chosen to give further insight into its efficiency.

n	Brute	Linear
1	0.000174	0.000176
2	0.000174	0.000175
3	0.000178	0.000176
4	0.000171	0.000178
5	0.000174	0.000175
6	0.000764	0.000176
7	0.004913	0.000177
8	0.038228	0.000176
9	0.337447	0.000176
10	3.390803	0.000176
11	37.69525	0.000175
100	n/a	0.000171
1000	n/a	0.000494
100000	n/a	0.007955

Best Case Time Efficiency: **Ω(n)**

For a best case scenario where both input strings are identical, both algorithms show an order of growth with a lower bound of Ω(n). Even in the best case, the linear algorithm will iterate through each of the strings and count characters in the same manner, thus scaling according to n characters.

The Brute force algorithm, meanwhile, can only attain best case scaling when the first permutation generated by the recursive calls is an exact match to the second string. Since the first permutation generated is always a copy of the input string, the best case will only occur if the two strings are already identical. The maximum depth of recursion down to the base case is always equal to the number of characters n. If it stops at the first permutation, it is essentially the equivalent of a single loop that iterates through n characters.



Values of n from 1 to 900 at 100-character intervals were chosen to effectively show the linear scaling in both algorithms. Interestingly, attempting to test n values of 1000 or greater showcased another shortcoming of the brute force algorithm: Python imposes a maximum limit on levels of recursion that it will execute. It halts execution and returns an error whenever levels of recursion exceed the limit. Unless it is forcibly changed, the limit ultimately makes high values of n impossible for the brute force algorithm under all circumstances.

n	Brute	Linear
1	0.000177	0.000174
100	0.000255	0.000171
200	0.000321	0.000172
300	0.000228	0.00016
400	0.000354	0.000211
500	0.000344	0.000252
600	0.000558	0.000316
700	0.000566	0.000361
800	0.000578	0.000404
900	0.00068	0.00046

Space Efficiency: **Θ(n)**

Both algorithms have space efficiency with an order of growth of Θ(n). The brute force algorithm does not require any extra space beyond what it allocates for the input strings. It does not save all generated permutations, which would cause it to have space efficiency of O(n!). The linear algorithm, meanwhile, does require extra space for two additional arrays, but since they only contain counts of letters, they are fixed to a constant length of 26. The linear algorithm thus also has space efficiency that scales with the length of input strings.

Sample Program Runs

Brute Force and Linear Worst Case (n = 11):

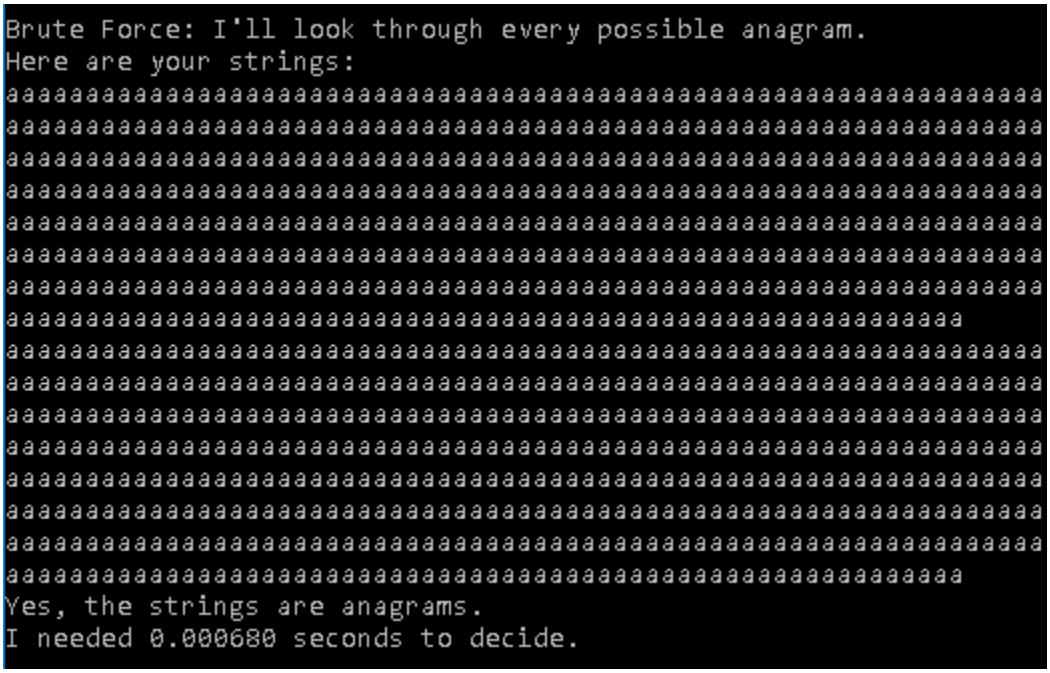
```
Brute Force: I'll look through every possible anagram.
Here are your strings:
abcdefghijk
aaaaaaaaaaa
Nope, the strings are not anagrams.
I needed 37.695250 seconds to decide.

Linear: I'll just count the letters and compare.
Here are your strings:
abcdefghijk
aaaaaaaaaaa
Nope, the strings are not anagrams.
I needed 0.000175 seconds to decide.
```

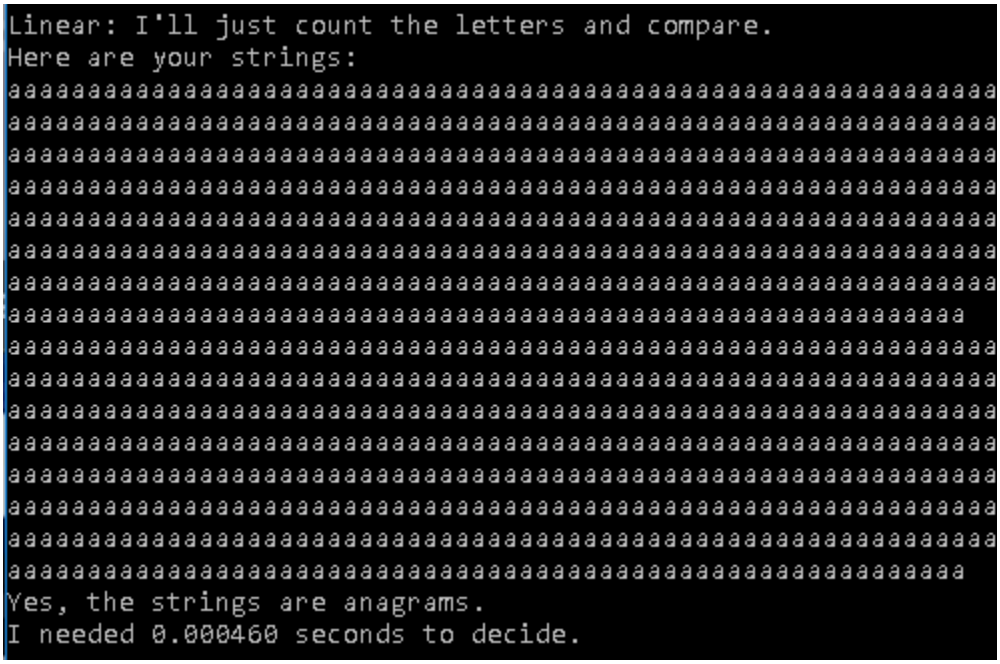
Linear Worst Case (n=100000):

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Nope, the strings are not anagrams.
I needed 0.007955 seconds to decide.
```

Brute Force Best Case (n=900):



Linear Best Case (n=900):



Realistic Example – Complete Program Run (n=8):

```
Hi, I'm Mr. Agana, The Anagram Decider

I'll figure out whether your strings are anagrams!

Which method should I use?
1: Brute Force
2: Linear
3: Brute Force and Linear
4: Exit
Enter your selection number: 3

Enter a string: silenced
Enter another string: licensed

Brute Force: I'll look through every possible anagram.
Here are your strings:
silenced
licensed
Yes, the strings are anagrams.
I needed 0.010720 seconds to decide.

Linear: I'll just count the letters and compare.
Here are your strings:
silenced
licensed
Yes, the strings are anagrams.
I needed 0.000180 seconds to decide.

Which method should I use?
1: Brute Force
2: Linear
3: Brute Force and Linear
4: Exit
Enter your selection number: 4

All right, see you later!
```