3.4. An Anagram Detection Example

A good example problem for showing algorithms with different orders of magnitude is the classic anagram detection problem for strings. One string is an anagram of another if the second is simply a rearrangement of the first. For example, `'heart'` and `'earth'` are anagrams. The strings `'python'` and `'typhon'` are anagrams as well. For the sake of simplicity, we will assume that the two strings in question are of equal length and that they are made up of symbols from the set of 26 lowercase alphabetic characters. Our goal is to write a boolean function that will take two strings and return whether they are anagrams.

3.4.1. Solution 1: Checking Off

Our first solution to the anagram problem will check the lengths of the strings and then to see that each character in the first string actually occurs in the second. If it is possible to “checkoff” each character, then the two strings must be anagrams. Checking off a character will be accomplished by replacing it with the special Python value `None`. However, since strings in Python are immutable, the first step in the process will be to convert the second string to a list. Each character from the first string can be checked against the characters in the list and if found, checked off by replacement. ActiveCode 1 shows this function.

Save & Run | Load History | Show CodeLens

```
            else:
                pos2 = pos2 + 1

        if found:
            alist[pos2] = None
        else:
            stillOK = False

        pos1 = pos1 + 1

    return stillOK

print(anagramSolution1('abcd','dcba'))

```

Activity: 3.4.1.1 Checking Off (active5)

To analyze this algorithm, we need to note that each of the *n* characters in `s1` will cause an iteration through up to *n* characters in the list from `s2`. Each of the *n* positions in the list will be visited once to match a character from `s1`. The number of visits then becomes the sum of the integers from 1 to *n*. We stated earlier that this can be written as

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

(BigONotation.html) (PerformanceofPy

As n gets large, the n^2 term will dominate the n term and the $\frac{1}{2}$ can be ignored. Therefore, this solution is $O(n^2)$.

3.4.2. Solution 2: Sort and Compare

Another solution to the anagram problem will make use of the fact that even though `s1` and `s2` are different, they are anagrams only if they consist of exactly the same characters. So, if we begin by sorting each string alphabetically, from a to z, we will end up with the same string if the original two strings are anagrams. ActiveCode 2 shows this solution. Again, in Python we can use the built-in `sort` method on lists by simply converting each string to a list at the start.

Save & Run Load History Show CodeLens

```
def anagramSolution2(s1,s2):
    alist1 = list(s1)
    alist2 = list(s2)

    alist1.sort()
    alist2.sort()

    pos = 0
    matches = True

    while pos < len(s1) and matches:
        if alist1[pos]==alist2[pos]:
            pos = pos + 1
        else:
            matches = False
```

Activity: 3.4.2.1 Sort and Compare (active6)

At first glance you may be tempted to think that this algorithm is $O(n)$, since there is one simple iteration to compare the *n* characters after the sorting process. However, the two calls to the Python `sort` method are not without their own cost. As we will see in a later chapter, sorting is typically either $O(n^2)$ or $O(n \log n)$, so the sorting operations dominate the iteration. In the end, this algorithm will have the same order of magnitude as that of the sorting process.

3.4.3. Solution 3: Brute Force

A **brute force** technique for solving a problem typically tries to exhaust all possibilities. For the anagram detection problem, we can simply generate a list of all possible strings using the characters from `s1` and then see if `s2` occurs. However, there is a difficulty with this approach. When generating all possible strings from `s1` , there are *n* possible first characters, $n-1$ possible characters for the second position, $n-2$ for the third, and so on. The total number of candidate strings is $n*(n-1)*(n-2)*...*3*2*1$, which is $n!$. Although some of the strings may be duplicates, the program cannot know this ahead of time and so it will still generate $n!$ different strings.

(BigONotation.html) (PerformanceofPy

It turns out that $n!$ grows even faster than 2^n as *n* gets large. In fact, if `s1` were 20 characters long, there would be $20! = 2,432,902,008,176,640,000$ possible candidate strings. If we processed one possibility every second, it would still take us 77,146,816,596 years to go through the entire list. This is

probably not going to be a good solution.

3.4.4. Solution 4: Count and Compare

Our final solution to the anagram problem takes advantage of the fact that any two anagrams will have the same number of a's, the same number of b's, the same number of c's, and so on. In order to decide whether two strings are anagrams, we will first count the number of times each character occurs. Since there are 26 possible characters, we can use a list of 26 counters, one for each possible character. Each time we see a particular character, we will increment the counter at that position. In the end, if the two lists of counters are identical, the strings must be anagrams. ActiveCode 3 shows this solution.

Save & Run Load History Show CodeLens

```
def anagramSolution4(s1,s2):
    c1 = [0]*26
    c2 = [0]*26

    for i in range(len(s1)):
        pos = ord(s1[i])-ord('a')
        c1[pos] = c1[pos] + 1

    for i in range(len(s2)):
        pos = ord(s2[i])-ord('a')
        c2[pos] = c2[pos] + 1

    j = 0
    stillOK = True
    while j<26 and stillOK:
```

Activity: 3.4.4.1 Count and Compare (active7)

Again, the solution has a number of iterations. However, unlike the first solution, none of them are nested. The first two iterations used to count the characters are both based on *n*. The third iteration, comparing the two lists of counts, always takes 26 steps since there are 26 possible characters in the strings. Adding it all up gives us $T(n) = 2n + 26$ steps. That is $O(n)$. We have found a linear order of magnitude algorithm for solving this problem.

Before leaving this example, we need to say something about space requirements. Although the last solution was able to run in linear time, it could only do so by using additional storage to keep the two lists of character counts. In other words, this algorithm sacrificed space in order to gain time.

This is a common occurrence. On many occasions you will need to make decisions between time and space trade-offs. In this case, the amount of extra space is not significant. However, if the underlying alphabet had millions of characters, there would be more concern. As a computer scientist, when given a choice of algorithms, it will be up to you to determine the best use of computing resources given a particular problem.

(BigONotation.html) (PerformanceofPy

Self Check

Q-4: Given the following code fragment, what is its Big-O running time?

```
test = 0
for i in range(n):
   for j in range(n):
      test = test + i * j
```

- A. O(n)
- B. O(n^2)
- C. O(log n)
- D. O(n^3)

Check Me Compare me

Activity: 3.4.4.2 Multiple Choice (analysis_1)

Q-5: Given the following code fragment what is its Big-O running time?

```
test = 0
for i in range(n):
   test = test + 1

for j in range(n):
   test = test - 1
```

- A. O(n)
- B. O(n^2)
- C. O(log n)
- D. O(n^3)

Check Me Compare me

Activity: 3.4.4.3 Multiple Choice (analysis_2)

Q-6: Given the following code fragment what is its Big-O running time?

```
i = n
while i > 0:
   k = 2 + 2
   i = i // 2
```

- A. O(n)
- B. O(n^2)
- C. O(log n)
- D. O(n^3)

Check Me Compare me

(BigONotation.html) (PerformanceofPy

Activity: 3.4.4.4 Multiple Choice (analysis_3)

You have attempted 1 of 7 activities on this page

user not logged in

(BigONotation.html) (PerformanceofPy