# Sample Problem

Given two strings or lists of elements, determine whether or not they are anagrams – exactly the same bag of elements.

To simplify the problem, consider lists where the elements are selected from integers between 0 and 9999 inclusive.

## First Attempt

Step through one list and check the other.

```
for each item in word:
    i = 0
    found = false

    while not found and i < length(word2):
        if item == word2[i]:
            found = true
            word2[i] = None
        else:
            i = i + 1

    if found == false:
        return false

return true
```

### Analysis

Triangular number formula: 0.5 * n * (n + 1)

In the worst case (more or less), we will have to step through the inner loop approximately this many times. This is slow.

# Second Attempt

Maintain a dictionary of elements seen in the first, reduce the count by one at each letter. If the dictionary returns to 0 afterwards, they are anagrams.

```
for each i from 0 to len - 1:
    dict[word1[i]] += 1
    dict[word2[i]] -= 1

for each value in dict:
    if value != 0:
        return false
return true
```

## Analysis

The first loop is on the order of 2n, the second is on the order of 10000. Overall, 2n + 10000. This is slow for small inputs but far faster for large inputs.

If n is 1000, the second process gives 12000 instead of 500000.

You could improve this by building the dictionary as you go instead of beforehand, so that it's rarely 10000 elements in size.

# Third Attempt

Sort both words first, then step through each simultaneously and make sure they're identical.

```
sort(word1)
sort(word2)
for each i from 0 to n-1:
    if word1[i] != word2[i]:
        return false
return true
```

### Analysis

The loop takes n steps in the worst case. Python's sort algorithm is of the order nlogn, giving 2nlogn + n.

## Summary

For large input, the second attempt (dictionary) is best. For small input, the third attempt (sort function) is best.

Two things to take away:

- We need to analyse how complex our algorithms are and see if they're too slow.
- We need to run stress tests to analyse the run times of our algorithms.