Using Hashed Storage in Dictionaries to Avoid or Reduce Nested loops

Problem Statement

We have two lists, list1 and list2

We would like to know the indices in each list of any value that is present in both

We can do this by looping through the list twice, one loop per index

In [1]:
```
list1 = [3,5,10,3,7,4,6]
list2 = [3,4,2,10,3,8,9]
work_on2 = 0
for i in range(len(list1)):
    for j in range(len(list2)):
        work_on2+=1
        if list1[i] == list2[j]:
            print("Both elements are the same. " + '\nlist1 indice is: '+ str(i) +
```

```
Both elements are the same.
list1 indice is: 0
 list2 indice is: 0
Both elements are the same.
list1 indice is: 0
 list2 indice is: 4
Both elements are the same.
list1 indice is: 2
 list2 indice is: 3
Both elements are the same.
list1 indice is: 3
 list2 indice is: 0
Both elements are the same.
list1 indice is: 3
 list2 indice is: 4
Both elements are the same.
list1 indice is: 5
 list2 indice is: 1
```

We will use hashed storage (a python dictionary) to reduce the number of steps needed

First we create a dictionary that indicates whether each item in list1 is also in list2 The key is the index in loop 1, the value is true or false depending on whether the value at that index is in list2

Then we have a loop that goes through the key, value pairs. For the cases where the list1 value is in list2, we loop through loop2 to find the matching index

In [7]:
```
from collections import defaultdict
work_on    = 0
hash_map = defaultdict(int)

for i in range(len(list1)):
    work_on += 1
    hash_map[i] = list1[i] in list2

for key, value in hash_map.items():
   if value == True:
        for j in range(len(list2)):
            work_on += 1
            if list1[key] == list2[j]:
                print("Both elements are the same. " + '\nlist1 indice is: '+ str(k
   else:
        work_on += 1
```

```
Both elements are the same.
list1 indice is: 0
 list2 indice is: 0
Both elements are the same.
list1 indice is: 0
 list2 indice is: 4
Both elements are the same.
list1 indice is: 2
 list2 indice is: 3
Both elements are the same.
list1 indice is: 3
 list2 indice is: 0
Both elements are the same.
list1 indice is: 3
 list2 indice is: 4
Both elements are the same.
list1 indice is: 5
 list2 indice is: 1
```

In [8]:
```
print('nested loop number of steps: ' + str(work_on2))
print('hash map number of steps: ' + str(work_on))
```

```
nested loop number of steps: 49
hash map number of steps: 38
```

* Question/Action*

Can you estimate the order of the first approach, if the loop sizes are M and N?

For the second approach, the number of loops depends on slightly different factors. What would be the order if there are no matches in the lists? What would be the order if all values in the two list had a match?

1. This requires M*N operations. Because M and N are the same length, the time complexity would be O(n^2)

2. No matches would be M items, or O(n) time complexity. All matches would be M+M*N operations, or O(n^2 + n) because M and N would be equivalent.

Question/Action

Add the timing measurement tools from the other pair programming exercises to this notebook to get the elapsed time as well as the number of steps.

Which is actually more important, the elapsed time or the order?

In [5]:
```
def nested():
    work_on2 = 0
    for i in range(len(list1)):
        for j in range(len(list2)):
            work_on2+=1
            #if list1[i] == list2[j]:
                #print("Both elements are the same. " + '\nlist1 indice is: '+ str(

def hashed():
    work_on    = 0
    hash_map = defaultdict(int)

    for i in range(len(list1)):
        work_on += 1
        hash_map[i] = list1[i] in list2

    for key, value in hash_map.items():
        if value == True:
                for j in range(len(list2)):
                    work_on += 1
                    #if list1[key] == list2[j]:
                        #print("Both elements are the same. " + '\nlist1 indice is:
        else:
                work_on += 1

%timeit nested()
%timeit hashed()
```

```
1.46 µs ± 7.93 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
2.03 µs ± 43.7 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

The critical factor is the order. Elapsed time for a given sorting algorithm can be deceiving, depending on the input size of the array. Order is a tool that can be used to show how efficient an algorithm is depending on the scale of the array, which is far more generalized than elapsed time.