PairProgramming_Hashes_and_Loops

May 1, 2025

1 Using Hashed Storage in Dictionaries to Avoidor Reduce Nested loops

1.1 Beija Richardson 5/1/2025

1.2 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

[3]:
```
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) + '\n list2 indice is: ' + str(j) )
```

```
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

[8]:
```
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(key) + '\n list2 indice is: ' + str(j))
   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
```

[10]:
```
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
```

2 * 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?

3 The outer loop runs M times.For each iteration of the outer loop, the inner loop runs N times.The total number of iterations = M * N.

for item1 in list1:
if item1 == item2: # No Matches = O(M × N) # Matches = O(M)

4 *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?

4.1 Elapsed time is useful for benchmarking real performance,comparing two concrete implementations or profiling hotspots in code. Factors that could vary include hardware (CPU speed, memory), programming language or compiler, or input data patterns.

4.2 Order of complexity is useful for predicting scalability or comparing algorithims regardless of hardware

4.3 Order of complexity matters more for understanding performance growth, while elapsed time matters more for real-world user experience.