Objectives

* Developing alternative, more efficient algorithms.
* Estimating the efficiency of an algorithm (Big-O)
* Measuring the time to execute code.

Due Date

This assignment is due on Tuesday, February 6, 2024, by 11:00 pm.

**Remarks:**

* It is expected you develop test cases for your code.
* When you are asked to hand in code, you cut-and-paste the definition(s) from your code and paste into this Word document immediately following the activity.
* Include with your code several test case examples and your results.
* This homework is to be individual work.

Value

This assignment is worth 20 points.

Activities

You will be developing multiple algorithms for one problem: Given a list of unique integers and a target integer, find all pairs in the list that add up to the target. For example, given a list of integers [1, 2, 3, 4, 5] and a target integer 5, the function should return   
{(1, 4), (2, 3)} (a *set* of *tuples*). NOTE: It does not, and it should not, return the reverse pairs. It should *not* return {(1, 4), (2, 3) , (3, 2) , (4, 1)}.

1. Use test-driven development as you go - write unittests (using the unittest module) in a file TestHw3.py. To fully test the find\_pairs functions, you should look for at a minimum:

* Test the functions with different inputs and check if the output is correct.
* Test the function with edge cases and different inputs to check the correctness of the functions. For example: test case with empty list; test case with target = 0; test case with no pairs that meet the target.
* A test case with target that equals to double number in the list. For example, For example if input list is [1, 2, 3, 4, 5] and target is 6, the output should be {(2, 4), (1, 5)}. Pair (3,3) should not be included.
* A test case with only expected duplicate values as the target. For example, if input list is [1, 2, 3, 4, 5] and target is 10, the output should be an empty set set(). (i.e., there is only one solution for the target but it requires duplicating a value which is not allowed so the result is the empty set).

1. Be sure to identify at least one more test than the above. Paste your test code below this bullet.

import unittest  
from hw3 import find\_pairs\_naive, find\_pairs\_optimized  
  
class TestPairFindingAlgorithms(unittest.TestCase):  
  
 def test\_basic\_functionality(self):  
 *"""Using a basic list, test hw3"""* self.assertEqual(find\_pairs\_naive([1, 2, 3, 4, 5], 5), {(1, 4), (2, 3)})  
 self.assertEqual(find\_pairs\_optimized([1, 2, 3, 4, 5], 5), {(1, 4), (2, 3)})  
  
 def test\_with\_negative\_numbers(self):  
 *"""Using a list with negative numbers"""* self.assertEqual(find\_pairs\_naive([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5], 0), {(-5, 5), (-4, 4), (-3, 3), (-2, 2), (-1, 1)})  
 self.assertEqual(find\_pairs\_optimized([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5], 0), {(-5, 5), (-4, 4), (-3, 3), (-2, 2), (-1, 1)})  
  
 def test\_no\_pairs(self):  
 *"""Using a list and target where no two numbers can sum up to the target"""* self.assertEqual(find\_pairs\_naive([1, 2, 3, 4], 10), set())  
 self.assertEqual(find\_pairs\_optimized([1, 2, 3, 4], 10), set())  
  
 def test\_large\_list(self):  
 *"""Using a somewhat large list"""* large\_list = list(range(100))  
 target = 50  
 expected\_pairs = {(10, 40), (11, 39), (1, 49), (2, 48), (18, 32), (17, 33), (8, 42), (24, 26), (9, 41), (15, 35), (0, 50), (16, 34), (19, 31), (6, 44), (22, 28), (7, 43), (23, 27), (14, 36), (5, 45), (13, 37), (20, 30), (21, 29), (12, 38), (3, 47), (4, 46)}  
 self.assertEqual(find\_pairs\_naive(large\_list, target), expected\_pairs)  
 self.assertEqual(find\_pairs\_optimized(large\_list, target), expected\_pairs)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

1. Create a file hw3.py. In this file, you need to write two functions that behave as described above: find\_pairs\_naive(L, n) and find\_pairs\_optimized(L,n).
2. Write the function find\_pairs\_naive(L, n). The implementation of this algorithm should iterate over the entire list using two nested loops to check for pairs that add up to the target. Note that sorting the list may be of value. Provide a listing of your find\_pairs\_naive(L, n) below this line.
3. Write the function find\_pairs\_optimized(L, n). The implementation of this algorithm should make use of a data structure or other methods to improve the efficiency of the algorithm. Each of these algorithms will be timed in follow-on steps. Provide a listing of your find\_pairs\_optimized(L, n) below this line.

def find\_pairs\_naive(L, n):  
 *"""Utilizes O(n^2) to find a par of numbers"""* pairs = set()  
 for i in range(len(L)):  
 for j in range(i + 1, len(L)):  
 if L[i] + L[j] == n:  
 pairs.add((L[i], L[j]))  
 return pairs  
  
  
def find\_pairs\_optimized(L, n):  
 *"""Utilizes Dictionary to find a pair of numbers"""* pairs = set()  
 seen = {}  
 for number in L:  
 complement = n - number  
 if complement in seen:  
 pairs.add((min(number, complement), max(number, complement)))  
 seen[number] = True  
 return pairs

1. In the same file hw3.py, write a function measure\_min\_time(func, args, n\_trials) which will measure the minimum time to execute the function over n trials. Provide a listing of your function measure\_min\_time(func, args, n\_trials) below this line.

def measure\_min\_time(func, args, n\_trials=10):  
 *"""Takes a function and argument input and  
 will keep track of the results of all n\_trials and  
 return the minimum amount of execution time as float"""* min\_time = float('inf')  
 for \_ in range(n\_trials):  
 start = time.perf\_counter()  
 func(\*args)  
 end = time.perf\_counter()  
 min\_time = min(min\_time, end - start)  
 return min\_time

1. Write a program within hw3.py that results in a table being presented along the lines as the following table is printed. Make use of properly formatted print statements so that it rounds the time to four decimal places. The number of trials (n) increases. Provide a listing of your code along with a copy of the output table.

n naive optimized

10 - -

50 - -

100 - -

150 - -

200 - -

300 - -

500 - -

def compare\_algorithms():  
 *"""Compares the naive time and optimized pair finding codes and generates a table which describes the processing   
 time of each function"""* trials = [1800, 1850, 1950, 2150, 2500]

print(f"n\tnaive\toptimized")  
 for n in trials:  
 list\_to\_test = generate\_list(n)  
 naive\_time = measure\_min\_time(find\_pairs\_naive, [list\_to\_test, n], 100)  
 optimized\_time = measure\_min\_time(find\_pairs\_optimized, [list\_to\_test, n], 100)  
 print(f"{n}\t{naive\_time:.4f}\t{optimized\_time:.4f}")

n naive optimized

1800 0.0676 0.0001

1850 0.0714 0.0001

1950 0.0793 0.0001

2150 0.0970 0.0001

2500 0.1310 0.0002

1. We can often calculate the running time of a function by adding up the execution cycles of every line. For instance, the below function has a O(n) running time to add up all the items in a list:

|  |  |
| --- | --- |
| **def** sum(L):  total **=** 0 | *# 1* |
| **for** item **in** L: | *# n* |
| total **+=** item | *# 2 (add, then assign)* |
| **return** total | *# 1*  *#--------------*  *# 1 + 2n + 1 = O(n)* |

1. Add comments to find\_pairs\_naive and find\_pairs\_optimized that show line-by-line execution cycles and the total sum, as in the example above. Be sure to insert newlines in your comments as appropriate so that the indentation of your code is not thrown off – make sure your code looks good. Provide the new listing of find\_pairs\_naive and find\_pairs\_optimized below. Include a brief explanation of the results in comments under each function.

def find\_pairs\_naive(L, n):  
 *"""Utilizes O(n^2) operations to take an input of a list and a number which will determine the pairs in the list  
 that add up to the number"""* pairs = set() # 1 (Assigns a variable)  
 for i in range(len(L)): # n (Outer loop runs n times)  
 for j in range(i + 1, len(L)): # n (Inner loop runs n times)  
 if L[i] + L[j] == n: # 1 (Assigns a variable if true)  
 pairs.add((L[i], L[j])) # 1 (Performs addition)  
 return pairs #1 (Returns a value)  
 # ------------------------  
 # 1 + 2n + 3 = O(n^2)  
  
def find\_pairs\_optimized(L, n):  
 *"""Utilizes O(n) operations to take an input of a list and a number which will determine the pairs in the list  
 that add up to the number"""* pairs = set() # 1 (Assigns a variable)  
 seen = {} # 1 (Builds blank dictionary)  
 for number in L: # n (Loop iterates n times)  
 complement = n - number # 1 (Subtraction)  
 if complement in seen: # 1 (Dictionary lookup)  
 pairs.add((min(number, complement), max(number, complement))) # 1 (Adds pairs)  
 seen[number] = True # 1 (Dictionary set operation)  
 return pairs # 1 (Returns a value)  
 # \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 # 2 + n + 5 = O(n)

1. Write a function generate\_list(n) which will return a list with n *non-repeating* integer elements. You can use the random module of python.
2. Modify the previous program which changed the number of trials so that it now changes the length of the list. Keep the number of trials constant at 100 trials. Present a similar table as before for the various values of the length of the list generated n.

n naive optimized

1800 0.0692 0.0001

1850 0.0724 0.0001

1950 0.0793 0.0001

2150 0.1035 0.0002

2500 0.1327 0.0002

1. In addition to pasting code within this Word document and submitting, also attach TestHw3.py and hw3.py to your Husky CT submission. Be sure it is a single submission in Husky CT with all three files.

import time  
import random  
def find\_pairs\_naive(L, n):  
 *"""Utilizes O(n^2) operations to take an input of a list and a number which will determine the pairs in the list  
 that add up to the number"""* pairs = set() # 1 (Assigns a variable)  
 for i in range(len(L)): # n (Outer loop runs n times)  
 for j in range(i + 1, len(L)): # n (Inner loop runs n times)  
 if L[i] + L[j] == n: # 1 (Assigns a variable if true)  
 pairs.add((L[i], L[j])) # 1 (Performs addition)  
 return pairs #1 (Returns a value)  
 # ------------------------  
 # 1 + 2n + 3 = O(n^2)  
  
def find\_pairs\_optimized(L, n):  
 *"""Utilizes O(n) operations to take an input of a list and a number which will determine the pairs in the list  
 that add up to the number"""* pairs = set() # 1 (Assigns a variable)  
 seen = {} # 1 (Builds blank dictionary)  
 for number in L: # n (Loop iterates n times)  
 complement = n - number # 1 (Subtraction)  
 if complement in seen: # 1 (Dictionary lookup)  
 pairs.add((min(number, complement), max(number, complement))) # 1 (Adds pairs)  
 seen[number] = True # 1 (Dictionary set operation)  
 return pairs # 1 (Returns a value)  
 # \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 # 2 + n + 5 = O(n)  
  
def measure\_min\_time(func, args, n\_trials=10):  
 *"""Takes a function and argument input and  
 will keep track of the results of all n\_trials and  
 return the minimum amount of execution time as float"""* min\_time = float('inf')  
 for \_ in range(n\_trials):  
 start = time.perf\_counter()  
 func(\*args)  
 end = time.perf\_counter()  
 min\_time = min(min\_time, end - start)  
 return min\_time  
  
def compare\_algorithms():  
 *"""Compares the naive time and optimized pair finding codes and generates a table which describes the processing  
 time of each function"""* trials = [1800, 1850, 1950, 2150, 2500]  
 print(f"n\tnaive\toptimized")  
 for n in trials:  
 list\_to\_test = generate\_list(n)  
 target = random.randint(1, n\*10)  
 naive\_time = measure\_min\_time(find\_pairs\_naive, [list\_to\_test, target], 100)  
 optimized\_time = measure\_min\_time(find\_pairs\_optimized, [list\_to\_test, target], 100)  
 print(f"{n}\t{naive\_time:.4f}\t{optimized\_time:.4f}")  
  
def generate\_list(n):  
 *"""Generates a list of random, non-repeated numbers"""* return random.sample(range(1, n\*10), n)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 compare\_algorithms()

import unittest  
from hw3 import find\_pairs\_naive, find\_pairs\_optimized  
  
class TestPairFindingAlgorithms(unittest.TestCase):  
  
 def test\_basic\_functionality(self):  
 *"""Using a basic list, test hw3"""* self.assertEqual(find\_pairs\_naive([1, 2, 3, 4, 5], 5), {(1, 4), (2, 3)})  
 self.assertEqual(find\_pairs\_optimized([1, 2, 3, 4, 5], 5), {(1, 4), (2, 3)})  
  
 def test\_with\_negative\_numbers(self):  
 *"""Using a list with negative numbers"""* self.assertEqual(find\_pairs\_naive([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5], 0), {(-5, 5), (-4, 4), (-3, 3), (-2, 2), (-1, 1)})  
 self.assertEqual(find\_pairs\_optimized([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5], 0), {(-5, 5), (-4, 4), (-3, 3), (-2, 2), (-1, 1)})  
  
 def test\_no\_pairs(self):  
 *"""Using a list and target where no two numbers can sum up to the target"""* self.assertEqual(find\_pairs\_naive([1, 2, 3, 4], 10), set())  
 self.assertEqual(find\_pairs\_optimized([1, 2, 3, 4], 10), set())  
  
 def test\_large\_list(self):  
 *"""Using a somewhat large list"""* large\_list = list(range(100))  
 target = 50  
 expected\_pairs = {(10, 40), (11, 39), (1, 49), (2, 48), (18, 32), (17, 33), (8, 42), (24, 26), (9, 41), (15, 35), (0, 50), (16, 34), (19, 31), (6, 44), (22, 28), (7, 43), (23, 27), (14, 36), (5, 45), (13, 37), (20, 30), (21, 29), (12, 38), (3, 47), (4, 46)}  
 self.assertEqual(find\_pairs\_naive(large\_list, target), expected\_pairs)  
 self.assertEqual(find\_pairs\_optimized(large\_list, target), expected\_pairs)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()