from itertools import combinations  
*'''  
에라토스테네스의 체에 의한 소수 구하기  
1. 2부터 n 까지의 소수를 구한다 하였을 때, n의 최대약수는 sqrt(n)이다  
2. 2부터 시작하여 수의 배수를 모두 걸러준다.  
3. 즉, 2 부터 sqrt(n)까지 반복하여 약수들의 배수를 모두 걸러준다  
4. 최종적으로 걸러지지 않은 수들은 모두 소수이다  
'''*nums = [1,2,7,6,4]  
  
def prime\_list(n):  
 # 에라토스테네스의 체 초기화: n개 요소에 True 설정(소수로 간주)  
 isPrime = [True] \* n  
 # n의 최대 약수가 sqrt(n) 이하이므로 i=sqrt(n)까지 검사  
 m = int(n \*\* 0.5)+1  
 for i in range(2, m):  
 if isPrime[i] == True: # i가 소수인 경우  
 for j in range(i+i, n, i): # i를 제외하고 i의 배수들은 모두 걸러준다  
 isPrime[j] = False  
 # 소수 목록 산출  
 return [i for i in range(2, n) if isPrime[i] == True]  
  
def solution(nums):  
 primeList = prime\_list(3000)  
 # nums 의 각 원소는 1 이상 1000 이하의 자연수이며 3개를 뽑기 때문에 최대 nums의 합은 3000이다.  
 # 즉 2에서 3000까지의 소수들을 모두 구한뒤 시작한다  
 primeCnt =0  
 # combinations 를 이용하여 n개의 수중에서 r개를 뽑는 nCr을 만든다  
 cases = list(combinations(nums,3)) #[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]  
 # 각 케이스 별로 그들의 합이 소수인지 판별하여 소수인 경우의 수를 카운트 하여 리턴하자  
 for case in cases:  
 if sum(case) in primeList: # 만약 케이스의 합이 소수리스트에 있다면 카운트 +1  
 primeCnt+=1  
 return primeCnt

이정우/ 파이썬/ 이전에 소수를 구하는 문제를 어느정도 접해봐서 생각했던 대로 구현할 수 있었습니다. 그리고 경우의 수를 구하는 것을 쉽게 해주는 combinations 라이브러리는 모든 경우의 수를 탐색하여 판별하는 문제에서 자주 등장하기 때문에 이번 기회에 다시 사용하면서 좀 익숙해진것 같습니다. 에라토스테네스의 체 같은 경우 일반적으로 떠올릴 수 있는 소수 판별법인 2부터 타겟 숫자까지를 나눴을 때 한번 밖에 안나눠떨어지면 소수이다 하는 방법은 O(n^2)인 반면 에라토스테네스의 체를 이용할 경우 O(Nlog(logN)으로 줄일 수 있기 때문에 확연한 차이를 보여준다. 이는 1000 단위를 넘어서는 순간부터는 O(n^2)의 방법을 사용할 경우 제시간에 문제가 도출되지 않는 점을 어느정도 개선할 수 있다 생각하여 좋았다. n까지의 최대약수가 루트n 이라는 점은 다소 이해하기 어려운 개념이였지만 계속 활용해나가며 익숙해지도록 노력해야겠다

def solution(numbers):  
 answer = []  
 for target in numbers: # numbers에 저장된 타겟 값을 받아온다  
 bit = bin(target).lstrip('0b')  
 # bit값으로 변환한 target을 변환한다. 기본적으로 '0bXX'가 되는데 앞의 0b를 빼준다  
 i = 1  
 bzc = bit.count('0')  
 boc = bit.count('1')  
 while True:  
 comp = bin(target+i).lstrip('0b') # 비교할 대상  
 czc = comp.count('0') # comp의 1의 개수와 0의 개수  
 coc = comp.count('1')  
 if abs(boc-coc) + abs(bzc-czc) <=2: # 1의 개수의 차 + 0의 개수의 차가 2이하일 때 탈출한다  
 break  
 # 아니라면 i 증가  
 i+=1  
 answer.append(target+i) # 타겟 + i는 다른 지점 2개 이하이면서 제일 작은수  
 return answer

리뷰: 처음으로 bin 이라는 함수에 대해 알게 되었다. 비트를 다루는 일이 흔하지 않다보니 좋은 경험이 되었던것 같다. 문제의 난점이라고 생각되는 점이 두가지 있는데 첫번째 2진수로 변환이며 두번째는 비트가 다른 지점의 판단이였다. 처음에는 Counter 를 떠올려 딕셔너리 형태로 만들어values들의 차를 이용하여 구현하려고 하였지만 bin을 이용한 값은 앞의 0을 무시하고 1부터 반환하기에 7인 경우 {'1':3} 의 형태로 나와 0의 값을 비교할 수 없었다. 그래서 비트 값임을 이용하여 두가지 값밖에 존재하지 않기때문에 각수를 이진수로 변환한뒤 1의 개수와 0의 개수를 count 메서드를 활용하여 개수를 구하였다. 생소한 문제이지만 bin 이라는 함수에 대해 알수 있어 뜻깊었다.

from queue import PriorityQueue  
  
print('문자 개수? ', end='')  
n = int(input()) # 문자개수  
# ================================ 파일 열어서 기존에 있던거 저장하기 ================================  
try:  
 f = open('huff.txt', 'rt', encoding='utf-8') # Open file with 'UTF-8' 인코딩  
 line = f.readline()  
 l = int(line.strip('\n')) # 가장 첫째줄에는 문자의 개수가 있다  
  
 charL = {}  
 for i in range(l):  
 line = f.readline().strip('\n').split()  
 charL[line[0]] = int(line[1]) # 딕셔너리 형태로 만들어 저장한다  
 if not line: break  
 f.close()  
except FileNotFoundError:  
 print('디스크에 해당 파일이 없습니다')  
# ================================ 파일 열어서 새로운 문자 추가하기 ================================  
for i in range(n):  
 flag = False # 기존 리스트에 새로받은 문자가 있는지 판단  
 print('문자? ', end='')  
 char = input()  
 print('빈도수? ', end='')  
 freq = int(input())  
 if char[i] in charL.keys(): # 문자가 이미 딕셔너리에 있는 상태라면  
 charL[char] = charL[char] + freq # 기존에 있던 빈도수에서 증가 시켜준다  
 else:  
 charL[char] = freq # 새로운 문자라면 딕셔너리에 추가해준다  
l = len(charL) # 새롭게 라인수를 초기화 한다  
f = open('huff.txt', 'w', encoding='utf-8')  
f.write('{}\n'.format(l)) # 가장 앞줄에 문자의 개수 초기화  
for i in range(l):  
 f.write('{} {}\n'.format(list(charL.keys())[i],list(charL.values())[i]))  
print('파일 덮어쓰기 완료!')  
f.close()  
  
# 파이썬에서 객체지향 프로그래밍을 위해 클래스를 생성한다  
class HuffNode: # 4개의 멤버 변수를 갖는다  
 def \_\_init\_\_(self, char, freq): # 문자와 빈도수를 파라미로 받는다  
 self.char = char  
 self.freq = freq  
 self.left = None # left, right 를 null로 초기화 한다  
 self.right = None  
  
 def \_\_eq\_\_(self, other): # 우선 순위가 같다면  
 return self.freq == other.freq  
  
 def \_\_ne\_\_(self, other): # 우선 순위가 다르다면  
 return self.freq != other.freq  
  
 def preorder(self): # root - left - right 순으로 채운다  
 print(self.char, end=' ')  
 if self.left is not None:  
 self.left.preorder() # 만약 왼쪽이 비지 않았다면 다시 재귀호출하여 왼쪽부터 채운다  
 if self.right is not None:  
 self.right.preorder() # 똑같이 오른쪽이 비지 않았다면 재귀호출하여 오른쪽채운다  
  
 def enbit(self, bit, target): # 문자가 같을때 까지 루트에서 left - right 순으로 순회  
 if self.char == target:  
 print(self.char + ': ' + bit)  
 return  
 if self.left is not None:  
 bit += '0'  
 self.left.enbit(bit, target)  
 bit = bit[:-1] # 만약 좌측에 자식노드가 없다면 마지막 비트 값은 삭제하고 오른쪽 순회  
 if self.right is not None:  
 bit += '1'  
 self.right.enbit(bit, target)  
  
def huffMan(PQ):  
 depth = 1  
 while PQ.qsize() >= 2: # 0~n-1까지 호출한다 (n개의 요소를 2개씩 짝지어 반환하기 때문에 n-1번 반복한다)  
 p = PQ.get()[1] #  
 q = PQ.get()[1] # 빈도수가 가장 작은 값 두개를 갖고 root를 만든다  
 r = HuffNode('H-' + str(depth), p.freq + q.freq) # root 빈도 = p빈도 + q빈도  
 r.left = p  
 r.right = q  
 PQ.put((r.freq, r))  
 depth += 1  
 return PQ.get()[1] # 마지막 노드를 반환한다  
  
  
PQ = PriorityQueue() # 우선순위 큐를 생성한다  
  
for i in range(len(charL)):  
 node = HuffNode(list(charL.keys())[i], list(charL.values())[i]) # 허프 노드의 생성자로 문자와 빈도수를 준다  
 PQ.put((node.freq, node)) # 빈도수를 기준으로 우선순위 큐를 생성한다  
  
root = huffMan(PQ)  
print('\n허프만 트리 생성 (Preorder): ')  
root.preorder()  
print('\n\n각 문자당 비트')  
for c in sorted(list(charL.keys()), key=lambda x:charL[x], reverse=True):  
 root.enbit('', c) # 빈도수가 가장 큰수 부터 차례로 출력한다  
  
fourB = 1  
while fourB <= l:  
 fourB \*= 2  
  
print('\n4-비트 코드 시: {} bits'.format(fourB)) # 4비트 사용시 n 보다 큰 2^n 만큼 필요하다  
print('허프만 코드 시: {} bits'.format(l)) # 허프만 사용시 문자의 개수 만큼만 있으면 된다

허프만 텍스트

17  
g 21  
n 60  
t 4  
k 6  
c 15  
i 20  
d 34  
a 10  
u 9  
p 7  
f 14  
s 30  
x 25  
e 4  
l 16  
z 2  
o 12

리뷰: 사실 어느정도는 알고리즘에 길들여져 있다 생각했지만 최근 치룬 코딩테스트나 이번 문제를 되돌아보며 많은것을 느끼고 아직 부족함을 몸소 체험했다. 오기와 비슷한 심정으로 극복하고자 밤새가며 이리저리 영상보고 소스들 보면서 시간 가는줄 모르고 코딩했던것 같다. 문제에서 제시하였던 파일 읽기 쓰기 기능과 허프만 코드 인코딩 부분이 살짝 이해가 가지 않아 헤매이는 감이 없지 않아 있었다. 우선 이해한 바에 의하면 빈도수를 기준으로 우선순위를 메겨 리프노드에 각 문자들이 오게끔 하고 루트에서 부터 내려오면서 깊이에 따라 각 리프노드의 비트를 메기는 것이였는데 객체들을 다루는 일이 많지 않아 시행착오가 많았었다. 아직까지 이해 안되는것은 class 에서 반환한 값을 메인함수에서 호출하면 None 이라고 뜨는 문제로 인해 직접 출력을 해야했던점… 해싱으로 비트 값을 삽입하는 방법도 좋았을것 같다는 생각을 이제야 해본다. 인코딩 뿐만아니라 디코딩 부분을 정말 해보고 싶었는데 시간이 날 때 조금 더 개선하여 디코딩 까지 해보려고 한다. 하프만 코드는 기존 문자열에 할당되는 비트값을 최소화 하는것에서 효율성이 매우 좋다고 알려져 있다. 다른 방면으로는 픽셀값등을 허프만 트리를 이용하여 압축할 수 있다. 마지막으로 우선순위를 비교할 때 빈도수를 기준으로 우선순위를 메겼는데 객체끼리는 같은 우선순위를 갖게 되면 충돌하여 에러가 발생하는 것을 해결하고자 발견했던 \_\_eq\_\_ 나 \_\_ne\_\_ 생성자가 다소 신기하고 유용할 것 같다는 생각이 들었다. 앞으로도 조금 더 자신을 발전시킬 수 있는 문제들이 자주 나왔으면 한다.