```
from collections import Counter

# Question 6
def bubble(frequencies, index):
    left = 2 * index + 1
    right = 2 * index + 2
    if left < len(frequencies) and frequencies[left][0] >
frequencies[index][0]:
        largest = left
    else:
        largest = index

    if right < len(frequencies) and frequencies[right][0] >
frequencies[largest][0]:
        largest = right
    if largest != index:
        frequencies[index], frequencies[largest] = frequencies[largest],
frequencies[index]
        bubble(frequencies, largest)

def max_heap(frequencies):
    for frequency in range(len(frequencies)//2 -1, -1, -1):
        bubble(frequencies,frequency)

def heapPush(frequencies,pair):
    frequencies.append(pair)
    max_heap(frequencies)

def heapPop(frequencies):
    value = frequencies.pop(0)
    frequencies = max_heap(frequencies)
    return value

def isRearrangePossible(s,k):
    heap = [(frequency,character) for character,frequency in
Counter(s).items()]
    max_heap(heap)
    queue = list()
    result = list()
    while heap:
        frequency,character = heapPop(heap)
        result.append(character)
        queue.append((frequency-1,character))

        if len(queue) >= k:
            updatedFreq, character = queue.pop(0)
            if updatedFreq:
                heapPush(heap,(updatedFreq, character))

    return False if len(result) != len(s) else True

#Question 5
def determineStandardRadius(houses,heaters):
    heaters.sort()
```

```
    houses.sort()
    result,index = 0,0

    for i in range(len(heaters)):
        minimum = houses[0] if i == 0 else (heaters[i-1]+heaters[i])//2
        maximum = houses[-1] if i == len(heaters)-1 else
(heaters[i+1]+heaters[i])//2
        while index < len(houses) and minimum <= houses[index] <= maximum:
            result = max(result,abs(heaters[i] - houses[index]))
            index+=1
    return result
#Question 4
def createMerge(arr, ans, low, mid, high):
    temp = list()
    index = low
    second = mid+1
    while (index < mid+1 and second <= high):
        if arr[index][0] > arr[second][0]:
            ans[arr[index][1]] += (high-second+1)
            temp.append(arr[index])
            index += 1
        else:
            temp.append(arr[second])
            second += 1
    while (index <= mid):
        temp.append(arr[index])
        index += 1
    while second <= high:
        temp.append(arr[second])
        second += 1

    k = 0
    index = low
    while (index <= high):
        arr[index] = temp[k]
        index += 1
        k += 1

def mergeRight(arr, ans, i, j):
    if i < j:
        mid = (i+j)//2
        mergeRight(arr, ans, i, mid)
        mergeRight(arr, ans, mid + 1, j)
        createMerge(arr, ans, i, mid, j)

def shorterBuildings(heights):
    heightIndex = [[heights[i],i] for i in range(len(heights))]
    results = [0 for _ in range(len(heights))]
    mergeRight(heightIndex,results,0,len(heights)-1)
    return results

# Question 3
import heapq
from collections import defaultdict
```

```
def find_median(maxHeap, minHeap, windowSize):
        if windowSize % 2 == 1:
            return maxHeap[0] * -1
        else:
            return (maxHeap[0] * -1 + minHeap[0]) / 2

def findMedianPrice(prices, k):
    maxHeap = list()
    minHeap = list()
    valDict = defaultdict(int)
    result = list()

    for i in range(k):
        heapq.heappush(maxHeap, -prices[i])
        heapq.heappush(minHeap, -heapq.heappop(maxHeap))
        if len(minHeap) > len(maxHeap):
            heapq.heappush(maxHeap, -heapq.heappop(minHeap))

    median = find_median(maxHeap, minHeap, k)
    result.append(median)

    for i in range(k, len(prices)):
        prev = prices[i - k]
        valDict[prev] += 1
        difference = -1 if prev <= median else 1
        if prices[i] <= median:
            difference += 1
            heapq.heappush(maxHeap, -prices[i])
        else:
            difference -= 1
            heapq.heappush(minHeap, prices[i])
        if difference < 0:
            heapq.heappush(maxHeap, -heapq.heappop(minHeap))
        elif difference > 0:
            heapq.heappush(minHeap, -heapq.heappop(maxHeap))
        while maxHeap and valDict[-maxHeap[0]] > 0:
            valDict[-maxHeap[0]] -= 1
            heapq.heappop(maxHeap)
        while minHeap and valDict[minHeap[0]] > 0:
            valDict[minHeap[0]] -= 1
            heapq.heappop(minHeap)

        median = find_median(maxHeap, minHeap, k)
        result.append(median)
    return result

# Question 2
def solvePuzzle(numbers):
    result = 0
    visited = [0 for _ in range(len(numbers))]
    significantNumbers = list()
    for i in range(0,len(numbers)):
        heapq.heappush(significantNumbers, (numbers[i],i))
```

```
    while significantNumbers:
        number, position = heapq.heappop(significantNumbers)
        if not visited[position]:
            visited[position] = 1
            if position:
                visited[position - 1] = 1
            if position < len(numbers)-1:
                visited[position + 1] = 1
            result += number
    return result

# Question 1
def busRemaining(busStation):
    busStation.sort()
    routes = list()
    if len(busStation) > 0:
        routes.append(busStation[0])
    else:
        return 0
    for route in busStation[1:]:
        if routes[-1][0] <= route[0] <= routes[-1][-1]:
            routes[-1][-1] = max(routes[-1][-1],route[-1])
        else:
            routes.append(route)
    return len(routes)

# Question 8

class Node:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
class Wavelet_Tree:
    def __init__(self, A):
        vals = ["" for i in range(10)]
        keys = [i for i in range(10)]
        self.numDictionary = dict(zip(keys,vals))

        def buildTreeStructure(A, left, right):
            size = len(A)
            if left == right or A == None:
                if size > 1:
                    item = 'X' * size
                    return Node(item)
                return Node('X')

            mid = (left + right)//2
            leftLeaves = list()
            rightLeaves = list()
            for i in range(len(A)):
                if A[i] <= mid:
                    leftLeaves.append(A[i])
```

```
                else:
                    rightLeaves.append(A[i])

            left_tree = buildTreeStructure(leftLeaves, min(leftLeaves),
max(leftLeaves))
            right_tree = buildTreeStructure(rightLeaves, min(rightLeaves),
max(rightLeaves))

            return Node(A, left_tree, right_tree)

        def buildSignature():
            level = 0
            selected = [(self.root, self.root.data, level)]
            while len(selected):
                current, val, rank = selected.pop(0)
                if not any(isinstance(item, str) and 'X' in item for item
in val):
                    mid = (min(val) + max(val))//2
                    for number in set(val):
                        if number <= mid:
                            self.numDictionary[number] += '0'
                        else:
                            self.numDictionary[number] += '1'

                if current.left is not None:
                    selected.append((current.left, current.left.data, rank
+ 1))

                if current.right is not None:
                    selected.append((current.right, current.right.data,
rank + 1))

        self.root = buildTreeStructure(A, min(A), max(A))
        buildSignature()

    def get_wavelet_level_order(self):
        result = list()
        curLevel = list()
        level  = 0
        prev = 0
        selected = [(self.root, self.root.data, level)]
        while len(selected) > 0:
            current, values, rank = selected.pop(0)
            temp = ""
            if any(isinstance(item, str) and 'X' in item for item in
values):
                temp += values
            else:
                mid = (min(values) + max(values))//2
                for val in values:
                    if val <= mid:
                        temp += '0'
                    else:
                        temp += '1'
```

```
            if rank != prev:
                result.append(curLevel)
                curLevel = list()
            curLevel.append(temp)

            prev = rank
            if current.left is not None:
                selected.append((current.left, current.left.data, rank +
1))

            if current.right is not None:
                selected.append((current.right, current.right.data, rank +
1))

        result.append(curLevel)
        return result

    def rank(self, character, position):
        bitDict = self.numDictionary[character]
        wavelets = self.get_wavelet_level_order()
        levels = len(bitDict)
        i,level,reps = 0,0,0
        curr_pos = position
        while i < levels:
            bit = bitDict[i]
            treeBits = 0
            for j in wavelets[i][level][:curr_pos]:
                if bit == j:
                    treeBits += 1
            if bit == '0':
                level = 2 * level
            else:
                level = 2 * level + 1

            if treeBits == 0:
                break
            else:
                curr_pos = treeBits
                i += 1
            if i == levels:
                reps = treeBits
        return reps

# Question 7
from heapq import heapify, heappush, heappop
class Huffman():
    def _init_(self):
        self.huffman_codes = {}
        self.source_string = ""

    def set_source_string(self, src_str):
        self.source_string = src_str
```

```
    def generate_codes(self):
        generated_codes = dict()
        frequencies = dict()
        for i in self.source_string:
            if i in frequencies:
                frequencies[i]+=1
            else:
                frequencies[i]=1
        heap = list()
        heapify(heap)
        for character,frequency in frequencies.items():
            heappush(heap,(frequency,[character]))
        while len(heap)>1:
            left,leftChar=heappop(heap)
            right,rightChar=heappop(heap)
            length=left+right
            charLength=leftChar+rightChar
            for i in leftChar:
                generated_codes[i] = generated_codes[i]+"0" if i in
generated_codes else "0"
                # if i in generated_codes:
                #     generated_codes[i]+="0"
                # else:
                #     generated_codes[i]="0"
            for i in rightChar:
                generated_codes[i] = generated_codes[i]+"1" if i in
generated_codes else "1"
                # if i in generated_codes:
                #     generated_codes[i]+="1"
                # else:
                #     generated_codes[i]="1"
            heappush(heap,(length,charLength))
        for freq,char in generated_codes.items():
            depth = char[::-1]
            generated_codes[i] = depth
        self.huffman_codes = generated_codes

    def encode_message(self, message_to_encode):
        encMsg = ""
        for char in message_to_encode:
            encMsg+=self.huffman_codes[char]
        return encMsg
    def decode_message(self, encoded_msg):
        decoded_msg = ""
        remain=""
        # n=len(encoded_msg)
        # i=0
        dic=self.huffman_codes
        for c in encoded_msg:
            remain=remain+c
            for k,code in dic.items():
                if remain==code:
                    decoded_msg+=k
                    remain=""
```

```
return decoded_msg
```