1.Answer

a) The upperbound running time is **O(n^3)**.

Given that we need to calculate the upperbound running time, we can assume that the times complexity of loops take the maximum (for example "*for j in range (i, n - 1)*", we can assume that every times of loop is n).

In this case, there are 3 iterations,while other lines take O(1) times.

Firstly, *for i = 0 to n -1*, which gones on **n times complexity**.

Secondly, *for j = i to n – 1*, which should counts **n-i** in every iteration**,** considering the worst situation, everytime it can run **n times. (Also, you can say that everytime it runs at most n times)**

Lastly, compute the product of entires A[i] to A[j], which should counts **j-i+1 in every iteration**(Because need to use *for x in range(i,j)* to do that), it can be counts as **n complexity** in this situation. **(Also, you can say that everytime it runs at most n times)**

In math, the time complexity can be showed like that:

$$\mathrm{T(n)} = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1} j - i + 1 \le \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n \ = \mathrm{O(n^3)}$$

In conclusion, the whole time complexity is **O(n^3).**

b) The lowerbound running time is **Ω(n^3)**

We can only look at the last $\frac{n}{2}$ iterations, since it is the part of the whole execution, it can be seen as the lowerbound of the whole function.

We can put i = $\frac{n}{2}$ into the T(n) above, which is:

$$\mathrm{T(n / 2)} = \sum_{i=n/2}^{n-1}\sum_{j=i}^{n-1} j - i + 1$$

Then,

$$\mathrm{T(n / 2)} = \sum_{i=n/2}^{n-1}\frac{(n-i)*(1+n-i)}{2}$$

The result must include n^3, thus, the Time complexity of T($\frac{n}{2}$) is n^3.

In conclusion, given that the a half of the iteration has a n^3 complexity, the whole time complexity must bigger than n^3, so the lowerbound **Ω(n^3)**

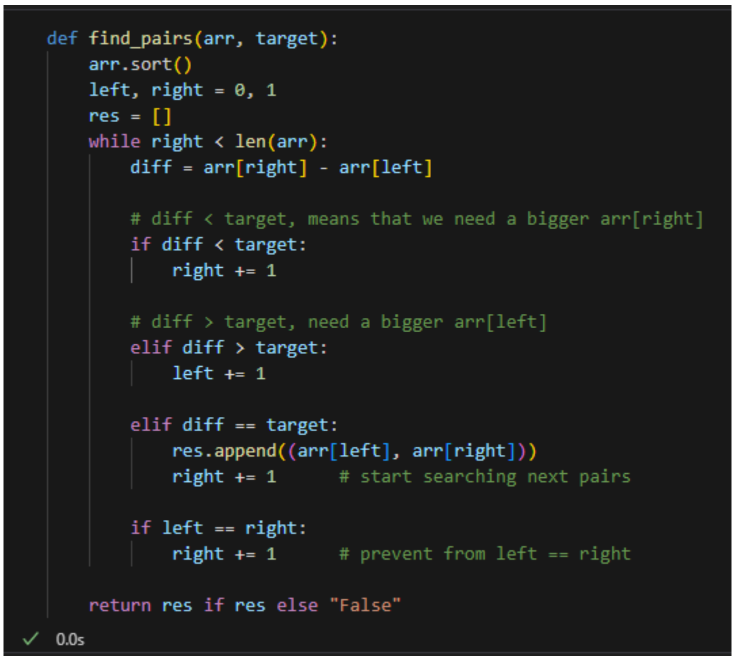
c) The time complexity is **O(n)**

To compute the time complexity of append a new element to the end of the linked list, you need to scan the whole list from the head, then visit the `next` attribute of each node, until you find the a node which `next` is null. Appending the element to the next of the last node, the time complexity will be **O(n)** due to scan all the node to find the last one.

2.Answer

a) Using double pointer (or slow, fast pointer)

The main idea is that use the left pointer pointing at the lower postion, while the right pointer pointing at the higher postion. Comparing the *arr[right] – arr[left]* and target value, we can change the left pointer (left += 1) and right pointer (right += 1) until we find all the answers.



```python
def find_pairs(arr, target):
    arr.sort()
    left, right = 0, 1
    res = []
    while right < len(arr):
        diff = arr[right] - arr[left]

        # diff < target, means that we need a bigger arr[right]
        if diff < target:
            right += 1

        # diff > target, need a bigger arr[left]
        elif diff > target:
            left += 1

        elif diff == target:
            res.append((arr[left], arr[right]))
            right += 1      # start searching next pairs

        if left == right:
            right += 1      # prevent from left == right

    return res if res else "False"
```

```
✓ 0.0s
```

Using this algorithm, we need to sort the whole array firstly, we can use bubble sort to do that, swap *a[j]* and *a[j + 1]* while *j* is from 0 to *n – i – 1*.

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Then, we need to define *left* and *right* pointer, starting from 0 and 1 (to make sure that the right > left), and *res* to store the result.

Then we need to start the loop, the condition is that *while right < len(arr)* , we assign *diff* as the difference of arr[right] and arr[left]. Comparing every *diff* with target number, if it is bigger, make *right + 1*, if it is smaller, make *left + 1,* while *diff == target*, we can store the result.

Considering that after *left + 1*, it may cause *left == right*, we need to do *right += 1* to make sure that *right* is always bigger than *left*.

Finally, return res if there are results else return `false`

b) The algorithm is correct.

The algorithm is correct due to the **basic movement of 2 pointers** and **sorting the array**, and it pass all the example input.

Sorting the whole array is first steps, which make sure that the array is ascending order so that *diff(arr[right] – arr[left])* can be bigger or smaller with the increasing *right* and *left*.

The basic rule of pointers movement depends on the *diff*. *diff* equals to *arr[right]* minus *arr[left]*. For a random stage, assumed that *right* = a, *left* = b, the *diff* = *arr[a] – arr[b]*.

There are 3 situations:

If *diff* < *target value*: which means that the arr[a] is too small, while we sort the array into ascending order, we can just make *right = a + 1* (the right pointer) to do that.

If *diff* > *target* : which means that the arr[b] is too small, so we can make *left = b + 1* (the left pointer) to do that.

If *diff* == *target* : which means that we get the right answer, given that *arr[a] > arr[b]*, we can store (*arr[b], arr[a]*) into result.

What's more, if the *arr[right]* is too small, *left* will always + 1, which will eventually make *left == right.* Considering we want all the *right* bigger than *left* to operate 3 situations above, we need to make *right + 1.*

In conclusion, this algorithm travesel all the elements using pointer *right*, and also find the possible pairs with *arr[right],* The bigger *arr[right]* needs a bigger *arr[left]*, so when moving into a bigger right position, element before left pointer can be ignored.

c) The time complexity is **O(n^2).**

Given that the length of the whole array is *n*, the time complexity of sorting is O(n^2) (Some algorithm can be quicker, but in this case we only use the basic one).

Some lines like assigning the *res, left, right* goes O(1) time complexity.

In the loop of *while right < len(arr),* we can see that if *right* add one for *n – 1* times, the loop will end. Considering about the worst situation, *left* and *right* alternately + 1, which will still be O(n) time complexity.

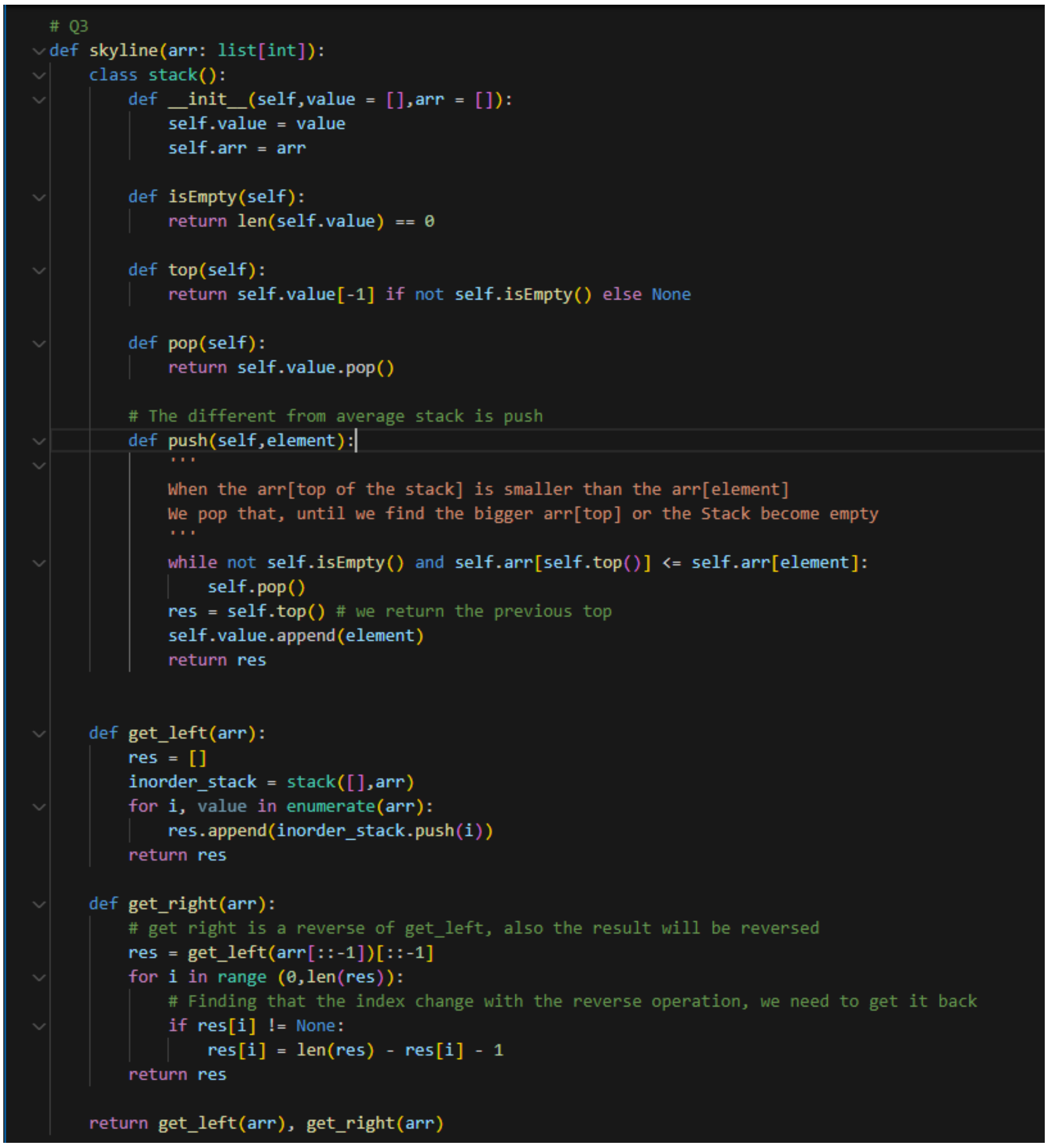
In conclusion, the time complexity is:

$$T(n) = O(n\text{^}2) + O(1) + O(n) = O(n\text{^}2)$$

3.Answer

a) Using Increasing Stack

The code is like that



```python
# Q3
def skyline(arr: list[int]):
    class stack():
        def __init__(self,value = [],arr = []):
            self.value = value
            self.arr = arr

        def isEmpty(self):
            return len(self.value) == 0

        def top(self):
            return self.value[-1] if not self.isEmpty() else None

        def pop(self):
            return self.value.pop()

        # The different from average stack is push
        def push(self,element):
            '''
            When the arr[top of the stack] is smaller than the arr[element]
            We pop that, until we find the bigger arr[top] or the Stack become empty
            '''
            while not self.isEmpty() and self.arr[self.top()] <= self.arr[element]:
                self.pop()
            res = self.top() # we return the previous top
            self.value.append(element)
            return res


    def get_left(arr):
        res = []
        inorder_stack = stack([],arr)
        for i, value in enumerate(arr):
            res.append(inorder_stack.push(i))
        return res

    def get_right(arr):
        # get right is a reverse of get_left, also the result will be reversed
        res = get_left(arr[::-1])[::-1]
        for i in range (0,len(res)):
            # Finding that the index change with the reverse operation, we need to get it back
            if res[i] != None:
                res[i] = len(res) - res[i] - 1
        return res

    return get_left(arr), get_right(arr)
```

We define a special stack, in which all the element *e* means the index of array. The *arr[e]* in it shows increasing trend from the top of the stack to the bottom of the stack.

All the operation is similar to normal stack except the *push()*. When pushing the element to the stack, we need to compare the value in array between *stack top* and the *element*, if *arr[stack_top]* is smaller than *arr[element],* we pop the *stack top* and compare the next *arr[new_stack_top]* with *arr[element]*; if *arr[element]* is bigger than *arr[stack_top]* or the stack is empty, we put the element on the top of the stack. Every time we push the element, we return the old stack top (the top before append operation, not the original stack top)

Then, we need to calculate the result of left taller buildings and right taller buildings.

For left taller building, we traversel all the element in the array, and push all the element into the increasing stack and store the result into result list *res*.

For right taller buildings, we can first reverse the array and put it into left taller buildings function, after getting the result, we reverse it again and change the index with *res[i] = len(res) – res[i] – 1.*

b) The algorithm is correct.

The algorithm is correct due to **the defination and function** of increasing stack, also for the correct operation to get left taller buildings and right taller buildings.

The stack actually store the possible result of the left taller buildings. Originally, it store nothing, so when we push the first element into it or the stack is empty, the old stack top that the function *push(element)* return *None*, and we also push the first element on the top of the stack.

Then, when we continually push element *e* into the stack, we need to compare the *arr[e]* with the *arr[stack_top]*. Considering the we push every element sides by sides, the closer an element reach the top, the closer it near to *e*. So, comparing the *arr[e]* with the *arr[stack_top]* means comparing the *arr[e]* with the previous index *arr[j] (0<=j < e)* from *e* to *0*.

If *arr[stack_top]* is bigger, which means that *stack_top* is the taller buildings that near to *e* , and the function can return the *stack_top* as the answer, then it will push the *e* in the stack for the next element. Else we continally pop the stack top and compare the *arr[new_stack_top]* with the *arr[e]* untill we find the *arr[new_stack_top]* which bigger than *arr[e]*.

By the way, it may be confusing about why `removing` the *new_stack_top* from stack if *arr[new_stack_top] < arr[e]*. It is because *e* (which is an index) is always bigger than *new_stack_top*, so if the *arr[new_stack_top] < arr[e],* it will never become the answer, so we can easily pop (or remove, delete) it from the stack.

Then, the get_left() and get_right() function, it can be seen that the push function of the stack can always return the left taller building, so we can use it for it. As for right taller buildings, it can be seen as the reverse of the get left taller buildings. Due to the index will be reversed and the result is reversed to, we need to do a little transfomeration as above.

c) The time complexity is O(n)

Considering about the time complexity, we can assume the worst situation, that every element of the array will push in the stack and then pop in the next iteration. So the highest time complexity of stack-operation is

$$O(2 * n) = O(n)$$

The get_left will get O(n) complexity, and get_right need to do trasnformation for each element after the stack operation, so the time complexity of all the code have a

$$O(n) + O(n) + O(n) + O(1) = O(n)$$

O(n) complexity. While the O(1) above represents other linear time complexity code.

4.Answer

a) **Use special double link list**

First define the `Order` class, which has attributes of

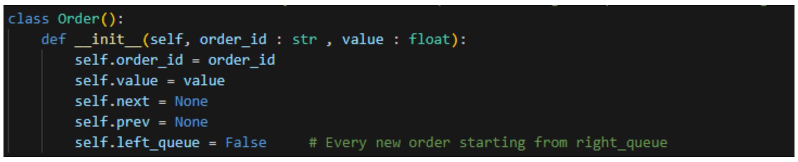
order_id,

value,

next(pointing at the next Order),

prev(pointing at the previous Order),

left_queue(means that whether the Order in the left half queue)



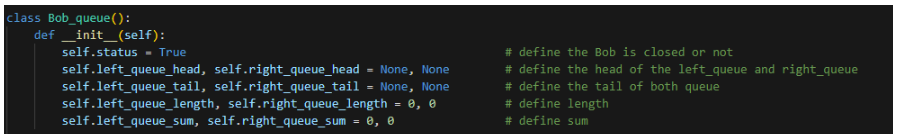
```python
class Order():
    def __init__(self, order_id : str , value : float):
        self.order_id = order_id
        self.value = value
        self.next = None
        self.prev = None
        self.left_queue = False      # Every new order starting from right_queue
```

The class `Bob_queue` is defined as follows:

status(whether the bob`s coffee shop is open),

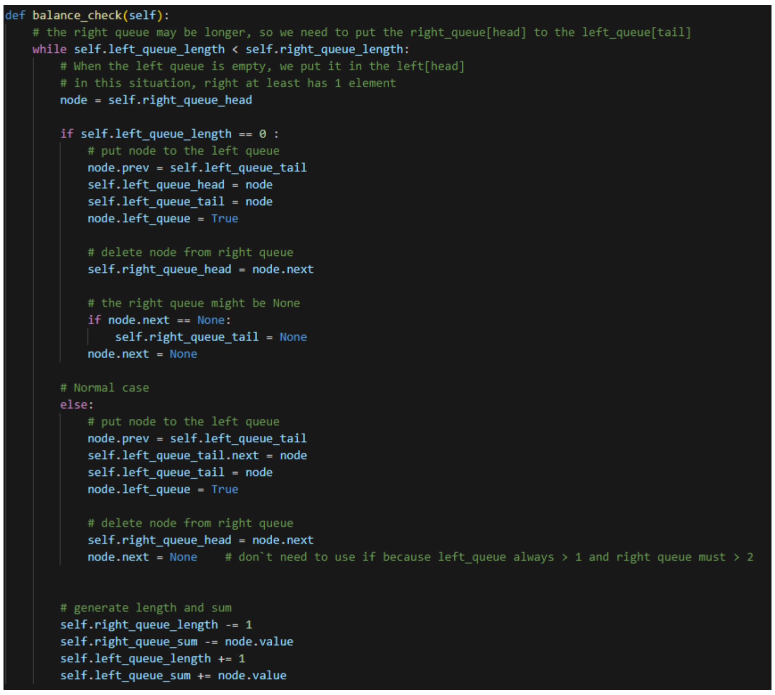
left half queue with head, tail, length, sum,

right half queue with head, tail, length, sum.



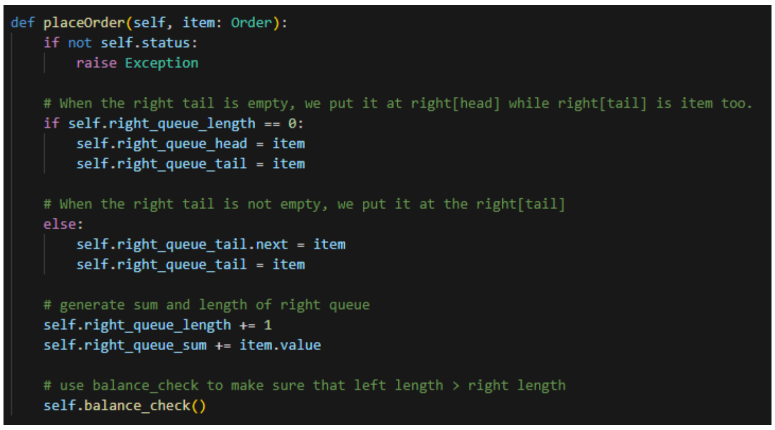
```python
class Bob_queue():
    def __init__(self):
        self.status = True                                              # define the Bob is closed or not
        self.left_queue_head, self.right_queue_head = None, None       # define the head of the left_queue and right_queue
        self.left_queue_tail, self.right_queue_tail = None, None       # define the tail of both queue
        self.left_queue_length, self.right_queue_length = 0, 0         # define length
        self.left_queue_sum, self.right_queue_sum = 0, 0               # define sum
```

The first function is helping balancing the left half queue and right half queue. Given that the length of left half queue need to equal or bigger (while the length of the whole queue is odd), we need to put the head of the right half queue to the tail of the left half queue. There are a lot of situations(for example, the left half queue do not have any element), the implementation in Python is below. Also, after operation, the attributes like length and sum need to be change.



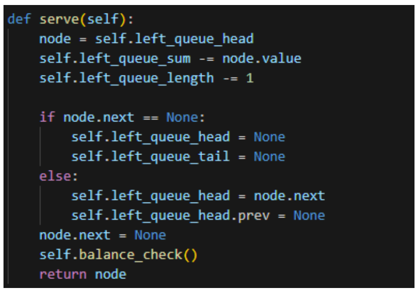
```python
def balance_check(self):
    # the right queue may be longer, so we need to put the right_queue[head] to the left_queue[tail]
    while self.left_queue_length < self.right_queue_length:
        # When the left queue is empty, we put it in the left[head]
        # in this situation, right at least has 1 element
        node = self.right_queue_head

        if self.left_queue_length == 0 :
            # put node to the left queue
            node.prev = self.left_queue_tail
            self.left_queue_head = node
            self.left_queue_tail = node
            node.left_queue = True

            # delete node from right queue
            self.right_queue_head = node.next

            # the right queue might be None
            if node.next == None:
                self.right_queue_tail = None
            node.next = None

        # Normal case
        else:
            # put node to the left queue
            node.prev = self.left_queue_tail
            self.left_queue_tail.next = node
            self.left_queue_tail = node
            node.left_queue = True

            # delete node from right queue
            self.right_queue_head = node.next
            node.next = None    # don`t need to use if because left_queue always > 1 and right queue must > 2


        # generate length and sum
        self.right_queue_length -= 1
        self.right_queue_sum -= node.value
        self.left_queue_length += 1
        self.left_queue_sum += node.value
```

The *placeOrder(item)* funciton, when the café shop is closed, it raise Exception. Else, we put the item in the right queue, the `add` operation will be different depending on whether the right_queue has element. After generating the length and sum of right_queue, using function `*balance_check()*` to ensure the left_queue and right_queue has the same length (or length of left is bigger).



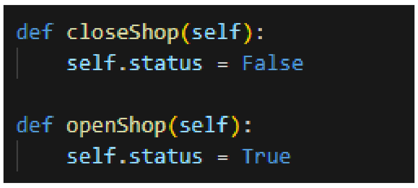
```python
def placeOrder(self, item: Order):
    if not self.status:
        raise Exception

    # When the right tail is empty, we put it at right[head] while right[tail] is item too.
    if self.right_queue_length == 0:
        self.right_queue_head = item
        self.right_queue_tail = item

    # When the right tail is not empty, we put it at the right[tail]
    else:
        self.right_queue_tail.next = item
        self.right_queue_tail = item

    # generate sum and length of right queue
    self.right_queue_length += 1
    self.right_queue_sum += item.value

    # use balance_check to make sure that left length > right length
    self.balance_check()
```

The *serve()* function delete the head of left_queue, generate sum,length, keep balance of left, right queue and return that node.



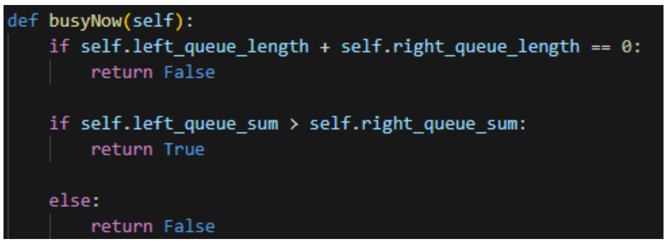
```python
def serve(self):
    node = self.left_queue_head
    self.left_queue_sum -= node.value
    self.left_queue_length -= 1

    if node.next == None:
        self.left_queue_head = None
        self.left_queue_tail = None
    else:
        self.left_queue_head = node.next
        self.left_queue_head.prev = None
    node.next = None
    self.balance_check()
    return node
```

closeShop() openShop() function set attribute *status* to False and True.



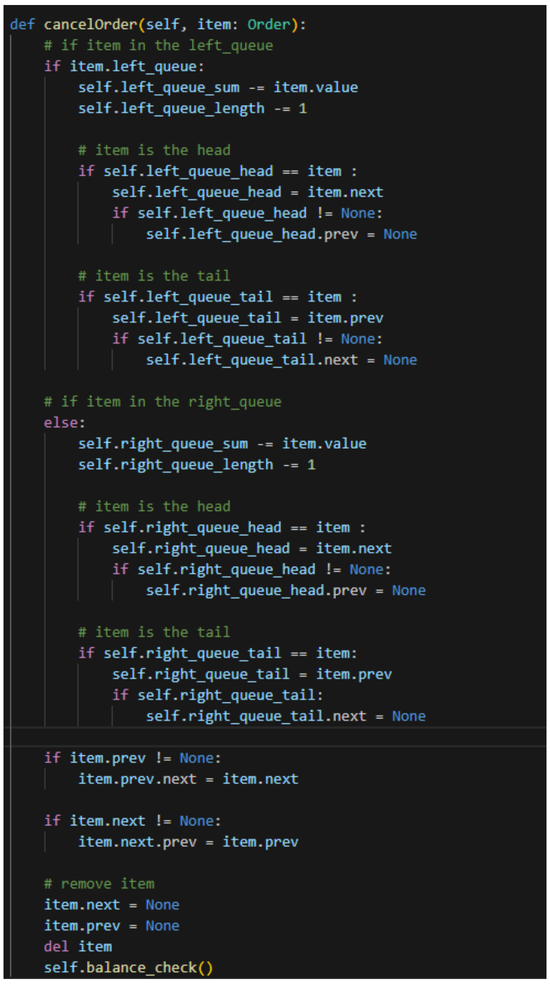
```python
def closeShop(self):
    self.status = False

def openShop(self):
    self.status = True
```

busyNow() function, when the whole length of function equal to 0, return False. When left_queue_sum > right_queue_sum, return True. When it comes to other situations, it return False.



```python
def busyNow(self):
    if self.left_queue_length + self.right_queue_length == 0:
        return False

    if self.left_queue_sum > self.right_queue_sum:
        return True

    else:
        return False
```

cancelOrder() function, considering the attributes `left_queue` of the given item, if it is True, which means that the item is in the left_queue, so we can generate the sum and length of right_queue. Else means that the item is in the right_queue. There are also a lot of situations, for example, item may be the head or tails. The details of cancel order from the left_queue or right_queue is below.



```
def cancelOrder(self, item: Order):
    # if item in the left_queue
    if item.left_queue:
        self.left_queue_sum -= item.value
        self.left_queue_length -= 1

        # item is the head
        if self.left_queue_head == item :
            self.left_queue_head = item.next
            if self.left_queue_head != None:
                self.left_queue_head.prev = None

        # item is the tail
        if self.left_queue_tail == item :
            self.left_queue_tail = item.prev
            if self.left_queue_tail != None:
                self.left_queue_tail.next = None

    # if item in the right_queue
    else:
        self.right_queue_sum -= item.value
        self.right_queue_length -= 1

        # item is the head
        if self.right_queue_head == item :
            self.right_queue_head = item.next
            if self.right_queue_head != None:
                self.right_queue_head.prev = None

        # item is the tail
        if self.right_queue_tail == item:
            self.right_queue_tail = item.prev
            if self.right_queue_tail:
                self.right_queue_tail.next = None

    if item.prev != None:
        item.prev.next = item.next

    if item.next != None:
        item.next.prev = item.prev

    # remove item
    item.next = None
    item.prev = None
    del item
    self.balance_check()
```

b) the algorithm and data structure is correct

In order to meet the requirements of O(1) of all operations, all the possible result or other stuff should be stored in attributes in the class (for example, left_sum need to be generate when the queue is changed to prenvent from O(n) Time complexity).

The different attributes of class `Order` and `Bob_queue` have different uses. The `Order` is a set of double linked node, and has the `left_queue` attribute to indicate whether the node is in the left queue, which will help `cancelOrder()` function to generate length and sum in O(1) time complexity.

There are also some special attributes in `Bob_queue`, the `status` is to indicate whether the store is open, with the use for the function `placeOrder(item)`, `openShop()`, `closeShop()`. The attributes related to the left queue and right queue (length, sum, head, tail), which helping those return the result with O(1) time.

The function `*balance_check()*` helps the whole queue remain balanced, which means that the length of left_queue will always equal to right_queue (or left = right + 1 when the whole length is odd). So when the length of right_queue is bigger, which means that the whole queue is not equal, we put the head of the right_queue to the tail of the left_queue. The whole operation need to consider about the condition that left_queue has no element, and the right_queue has no element after removing node from the right_queue.

The function `*placeOrder(item)*` puts item to the queue. First the status of the cofe shop is needed to be check, if it is closed, the Exception should be raise accroding to the given reading materials. Then, we put the element *e* to the tail of the right_queue(need to consider if the right_queue has element). After that, the whole queue may not be balanced with the new element, so we need to run function `*balance_check()*` to make left_queue and right_queue balance.

The function `*closeShop()*`, `*openShop()*` is to transfer the status of the cofe shop, with the defined element, we can easily do that.

The `*busyNow()*` function need to consider about whether the queue has no attribute, and the length of right_queue is bigger than left_queue. With the defination of attributes `left_queue_sum`, `right_queue_sum` and the generation of this 2 values in canceling order and adding order, it can be easily done.

The `*cancelOrder(item)*` give the item wanted to be cancel. With the defination of `*Order*` class, we can find the position and previous, next element about the item, and remove it from the queue. Considering that the item may be the head or tail, we need to do operation.

In conclusion, the algorithm is efficient for the fully-defination class, and all the function can meet the functional and non-functional requirements.

c) All the required function is O(1) complexity.

The function `*placeOrder(item)*` has a linear time complexity for all the code. The time complexity needed to be mention is the *balance_check()* since it has the loop. For any situation, the length of left queue is always equals to right queue or greater than 1. If it is greater than 1, after placing the order in the right queue, the right queue has the same length to the left queue, the loop will execute 0 times. When the length is equal, after placing, the length of left queue is greater 1 than the right queue, the loop will execute 1 times. So the whole time complexity wil be O(1)

The function `*serve()*` has a O(1) time complexity. There is no loop in the code, and since it served at most one item at once, the difference between left_queue and right_queue is at most 1, so the loop in balance_check is at most 1. The whole time complexity is O(1).

The function `*closeShop()*`, `*openShop()*` just assign the self.status with a new value, which is O(1) time.

The `*busyNow()*` check the length and the sum that are defined in the attribute, so it takes O(1) time complexity.

The function `*cancelOrder(item)*` has only `if - else` check because of the attribute defined above, and it at most remove 1 element so that the loop in `*balance_check()*` execute at most 1 times. So it takse O(1) time.

In conclusion, all the function takes O(1) time.