Problem 2

References:

B09902100 林弘毅

1.

```
ReverseQueue(source, helper)
    n = source.size()
    for(i=0; i<n; i++)
        for(j=0; j<i-1; j++)
            tmp = source.dequeue()
            source.enqueue(tmp)
        tail = source.dequeue()
        helper.enqueue(tail)
    for(i=0; i<n; i++)
        tmp = helper.dequeue()
        helper.enqueue(tmp)
```

2.

Because `enqueue`, `dequeue` and `size` all take $O(1)$ time, the time complexity is

$$O(1) + O(n \cdot (n+1)) + O(n) = O(1) + O(n^2) + O(n) = O(n^2)$$

3.

Use one stack (`front`) to simulate the front of the deque, and the other stack (`back`) to simulate the back.

For `push_front` and `push_back` we simply push items to the corresponding stack.

`pop_front` and `pop_back` are a bit trickier. When the corresponding stack isn't empty, we can simply pop from it. However when it's empty, we dump all items from the other stack to it, pop from it, and dump all items back.

```
push_front(deque, x)
    deque.front.push(x)

push_back(deque, x)
    deque.back.push(x)

pop_front(deque)
    if deque.front is not empty
        return deque.front.pop()
    else
        while deque.back is not empty
            deque.front.push(deque.back.pop())
        frt = deque.front.pop()
        while deque.front is not empty
            deque.back.push(deque.front.pop())
```

```
        return frt

pop_back(deque)
    if deque.back is not empty
        return deque.back.pop()
    else
        while deque.front is not empty
            deque.back.push(deque.front.pop())
        bck = deque.back.pop()
        while deque.back is not empty
            deque.front.push(deque.back.pop())
        return bck
```

4.

Because `stack.push()` takes $O(1)$ time, the time complexity of `push_front()` is $O(1)$.

5.

Because `stack.push()` takes $O(1)$ time, the time complexity of `push_back()` is $O(1)$.

6.

Let n be the length of the deque. When `deque.front` is not empty, time complexity of `pop_front()` $=$ time complexity of `stack.pop()` $= O(1)$.

When `deque.front` it empty, dumping items from `deque.back` to `deque.front` takes $O(n)$ time, `stack.pop()` takes $O(1)$ time, and dumping items from `deque.front` back to `deque.back` takes another $O(n)$ time. Therefore the total time complexity of `pop_front()` is $O(n) + O(1) + O(n) = O(n)$. The performance of my implementation tends to be better if `push_front` and `pop_front` are more balanced.

7.

Since the algorithm I have for `pop_back` is basically the same as `pop_front`, they share the same time complexity, that is $O(1)$ for best and $O(n)$ for worst.

8.

The best case happens when the stack was never full during n pushes, the time complexity in this case is $n \cdot O(1) = O(n)$.

For the worst case, it happens when we start pushing from 0 element to $3^k = n$ elements, because it will trigger `enlarge()` most times. In this case, the `S->arr[++S->top] = data;` part has the same time complexity, which is $O(n)$, therefore we only needs to look at how much time complexity do all `enlarge()` add.

`enlarge()` will happen when there is $1,\ 3,\ 3^2, \cdots, 3^k$. And the enlarged size would be $3^1,\ 3^2,\ 3^3, \cdots, 3^{k+1}$.

The time complexity for that would be $O(3^1 + 3^2 + \cdots + 3^{k+1}) = O(\frac{3(3^{k+1}-1)}{3-1}) = O(3^k) = O(n)$

Therefore, the total time complexity for worst case would also be $O(n) + O(n) = O(n)$.