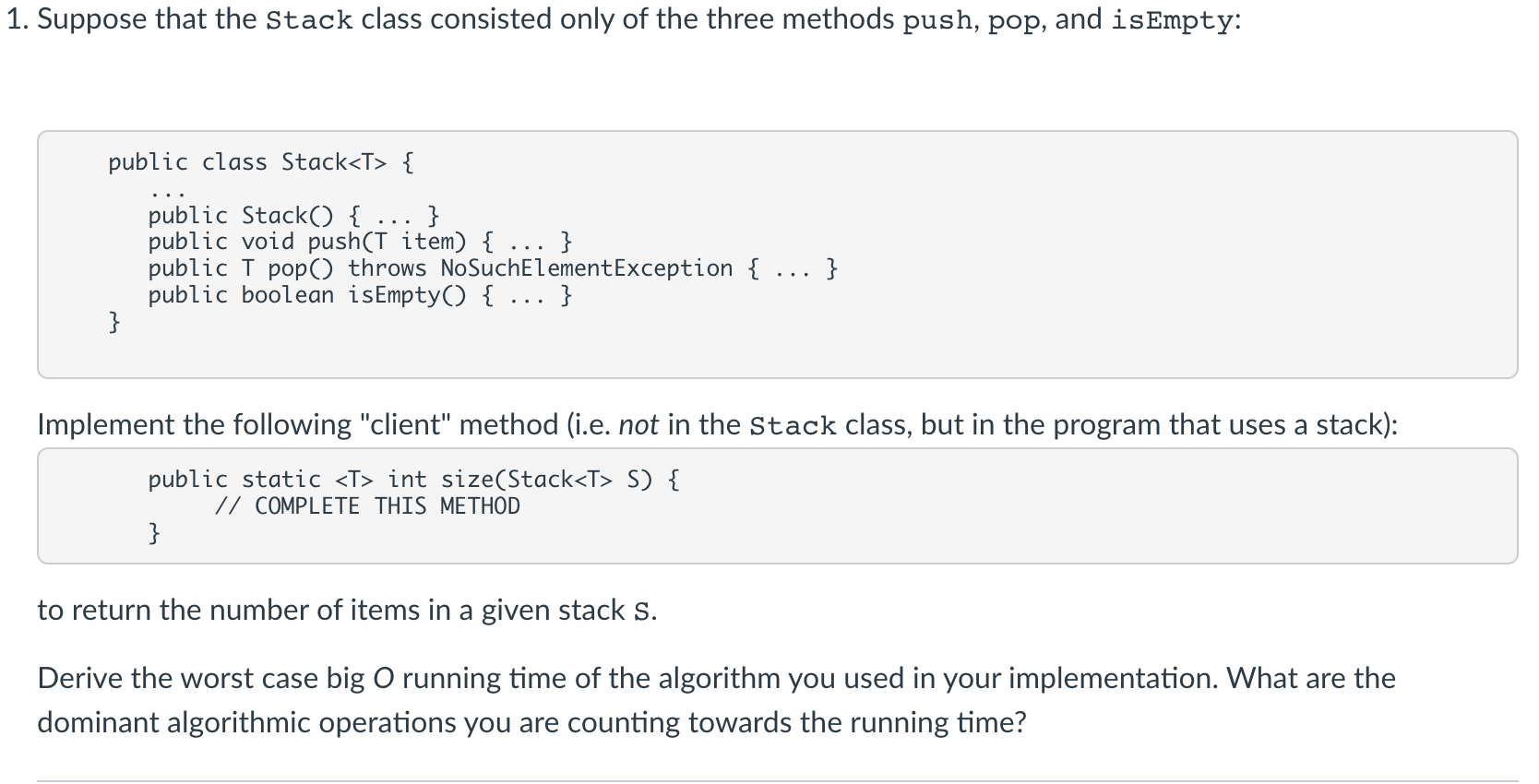
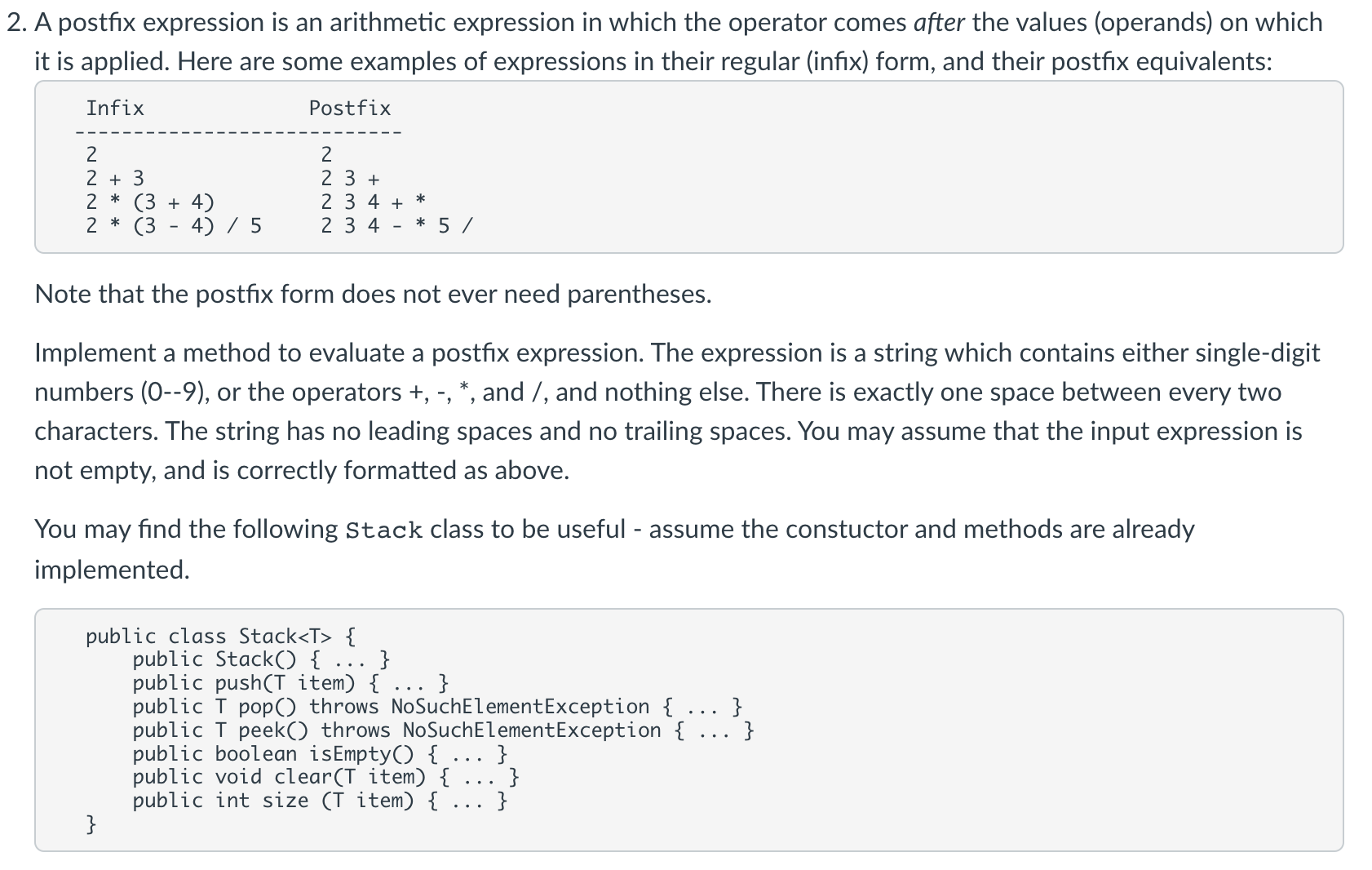
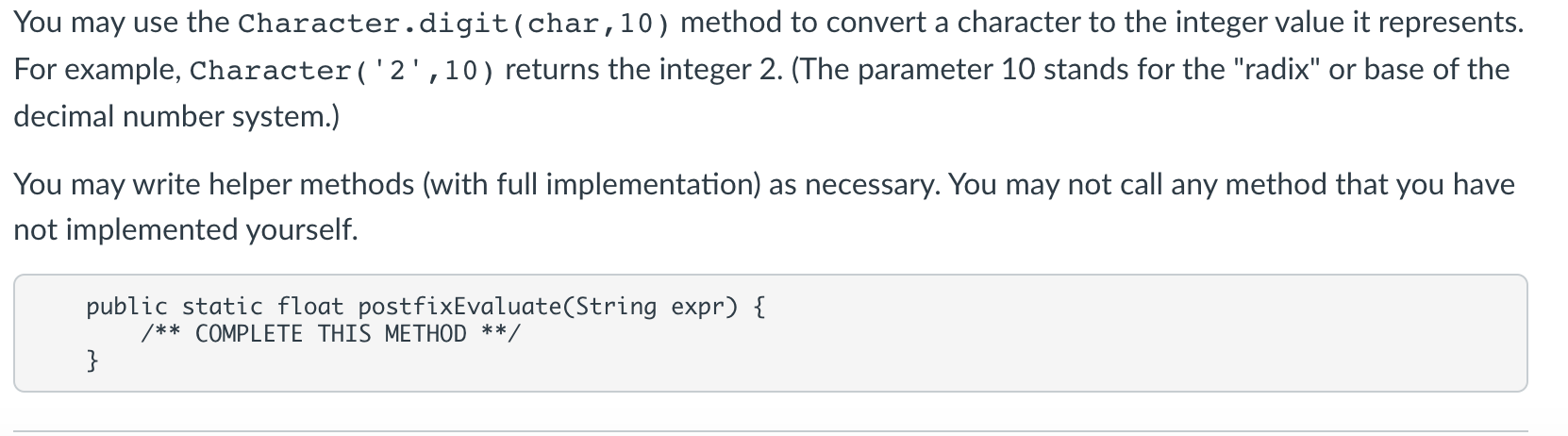
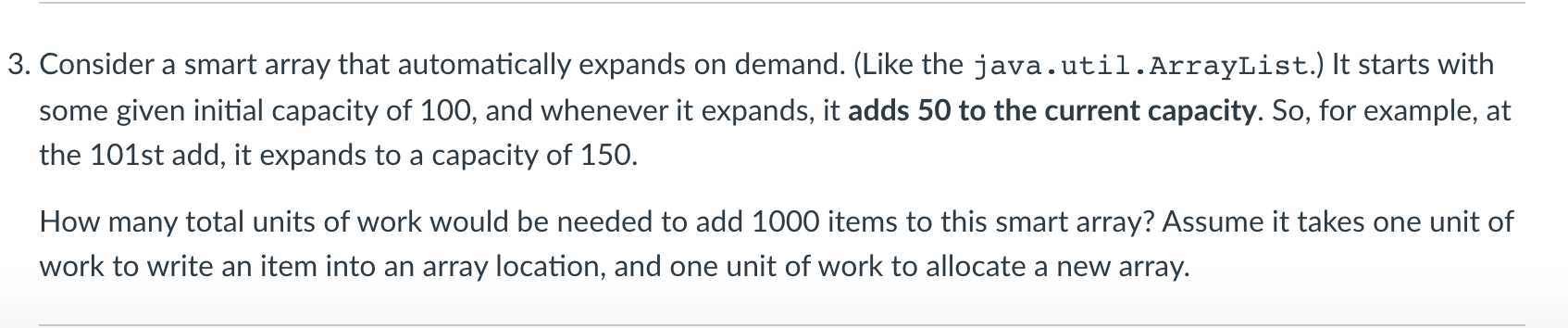
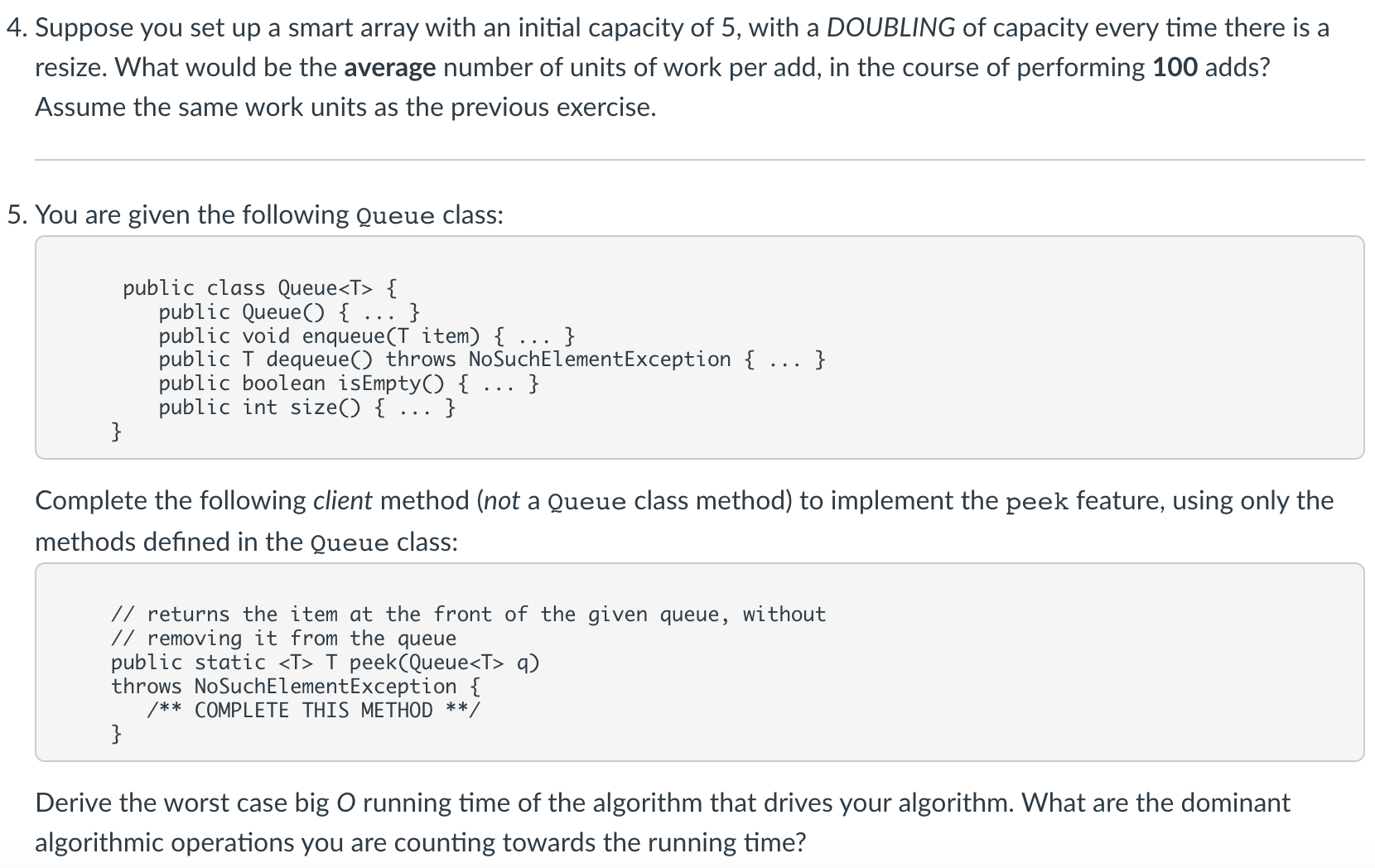
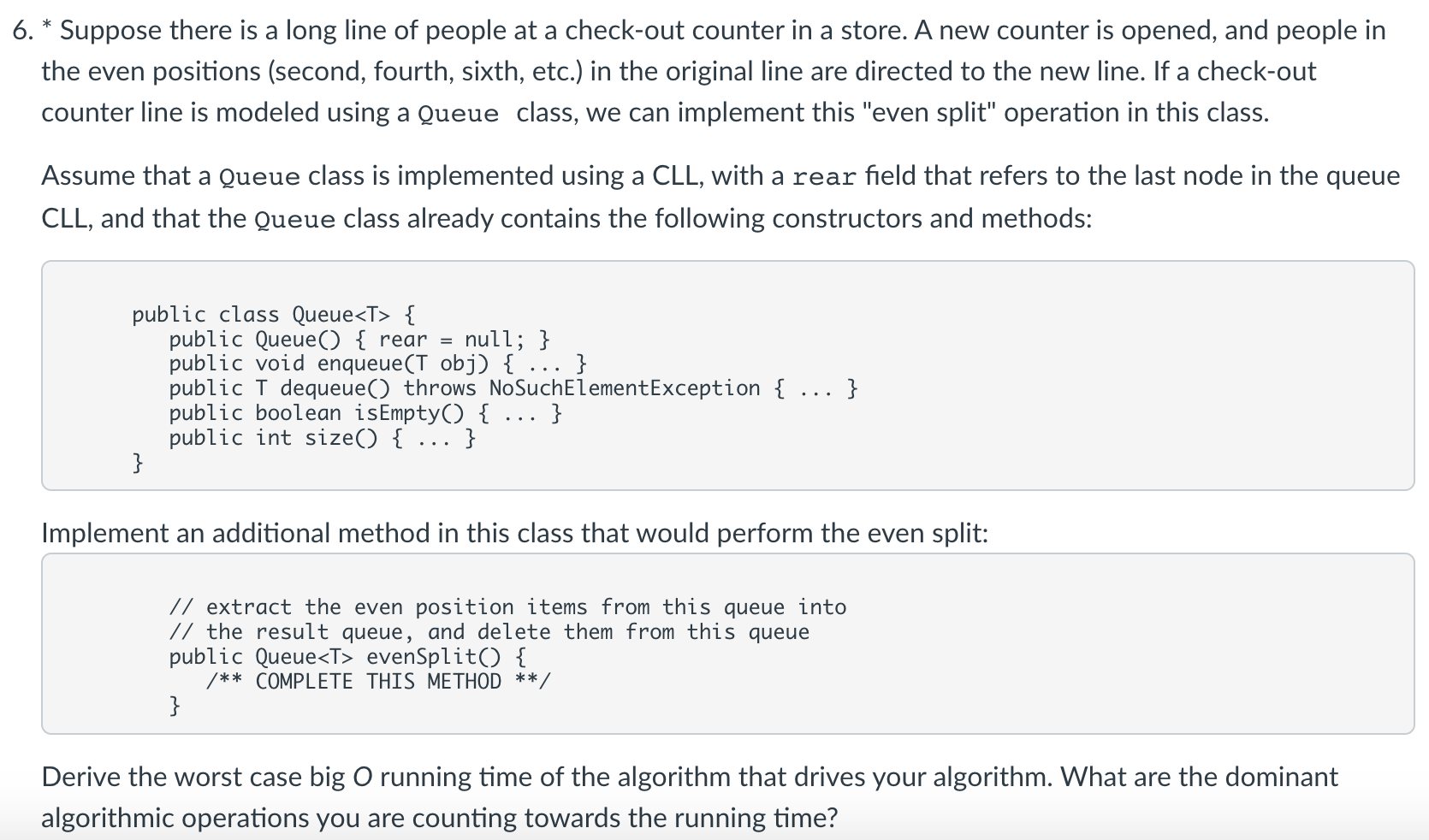
****

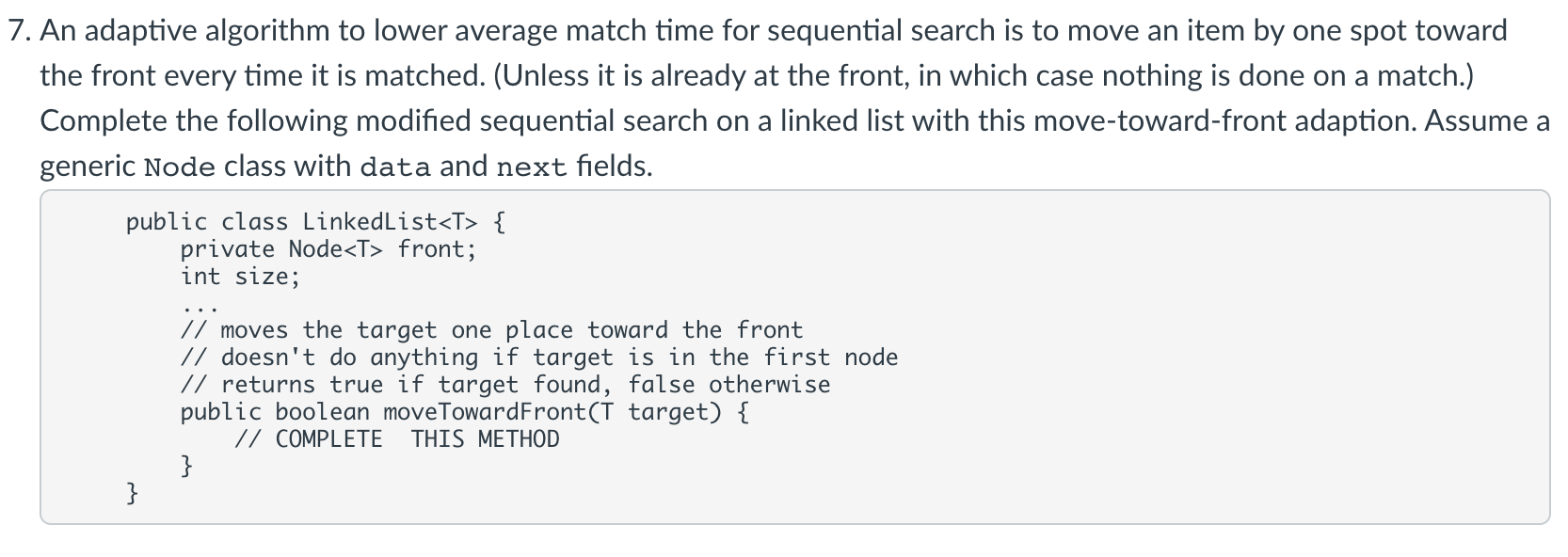
****

****

****

****

****

****

public static <T> int size(Stack<T> S) {

Stack<T> temp = new Stack<T>();

int count = 0;

while (!S.isEmpty()) {

temp.push(S.pop());

count++;

}

while (!temp.isEmpty()) {

S.push(temp.pop());

}

return count;

}

The dominant algorithmic operations for the stack are the push and pop. Elements of the stack are popped and pushed twice each for 4n total pushes and pops, meaning the Big O is O(n).

2.

public static float postfixEvaluate(String expr) {

Stack<Float> S = new Stack<Float>();

char chac=0;

for (int i=0; i < expr.length(); i++) {

chac = expr.charAt(i);

if (chac == '+' || chac == '-' || chac == '\*' || chac == '/') {

float second = S.pop();

float first = S.pop();

switch (chac) {

case '+': S.push(first + second);

case '-': S.push(first - second);

case '\*': S.push(first \* second);

case '/': S.push(first / second);

}

continue;

} else if (chac == ' ') {

continue;

}

S.push((float)Character.digit(chac,10));

}

return S.pop();

}

3.

Achieving 1000 elements would need 18 expansions of 50 units. 100(initial)+18\*50=1000

Calculating total units of work: 100(initial) + (1 (memory allocation) + 100 (writing old elements into new elements) + 50 (new units) + (1 + 150 + 50) ... (1+950+50).

4.

The array needs to be expanded 5 times in order to achieve the desired length. For every expansion, the current length is copied so we need 5, 10, 20, 40, and 80 expansions respectively.

To input 100 new values, it will take 100 units of work.

5 (initial) + 155 (added) + 100 (values written in) = 260. 260/100(total units) = 2.6 average.

5.

public static <T> T peek(Queue<T> q) {

if (q.isEmpty()) {

throw new NoSuchElementException();

}

T first = q.dequeue();

Queue<T> q2 = new Queue<T>();

q2.enqueue(first);

while (!q.isEmpty()) {

q2.enqueue(q.dequeue());

}

while (!q2.isEmpty()) {

q.enqueue(q2.dequeue());

}

return first;

}

The dominant operations are enqueue and dequeue. Everything in the queue is enqueued and dequeued twice - there are 4n operations, for a Big O time of O(n) time.

6. public Queue<T> evenSplit() {

Queue<T> evenQueue = new Queue<T>();

int qSize = size();

for (int pos=0; pos < qSize; pos += 2) {

enqueue(dequeue());

evenQueue.enqueue(dequeue());

qSize-=2;

}

if ((qSize % 2) == 1) {

enqueue(dequeue());

}

return evenQueue;

}

Enqueue and dequeue are the dominant operations. In each loop there is an enqueue and dequeue, for total 2n total operations. The Big O runtime is O(n).

7.

public boolean moveTowardFront(T target) {

Node<T> current = this.front;

Node<T> prev = null;

while (current != null) {

if (current.data.equals(target)) {

break;

} else {

prev = current;

current = current.next;

}

}

if (current == null) {

return false;

}

if (prev == null) {

T temp = prev.data;

prev.data = current.data;

current.data = temp;

return true;

}

}