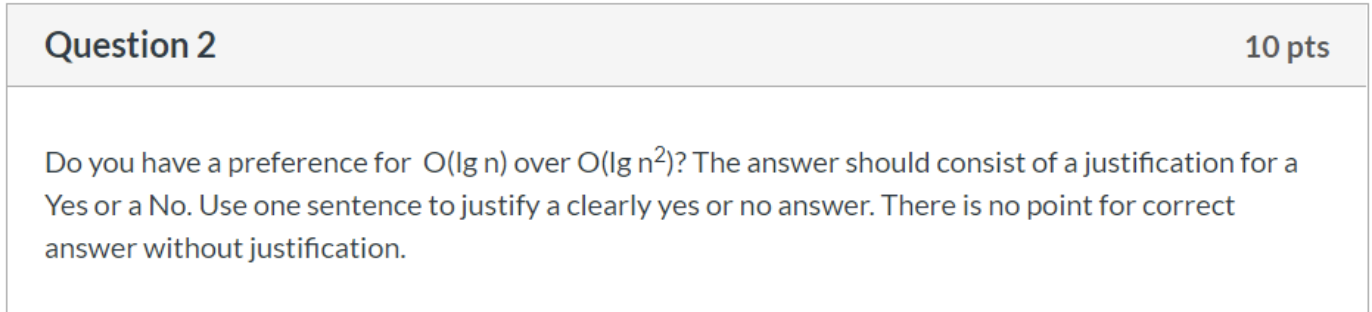
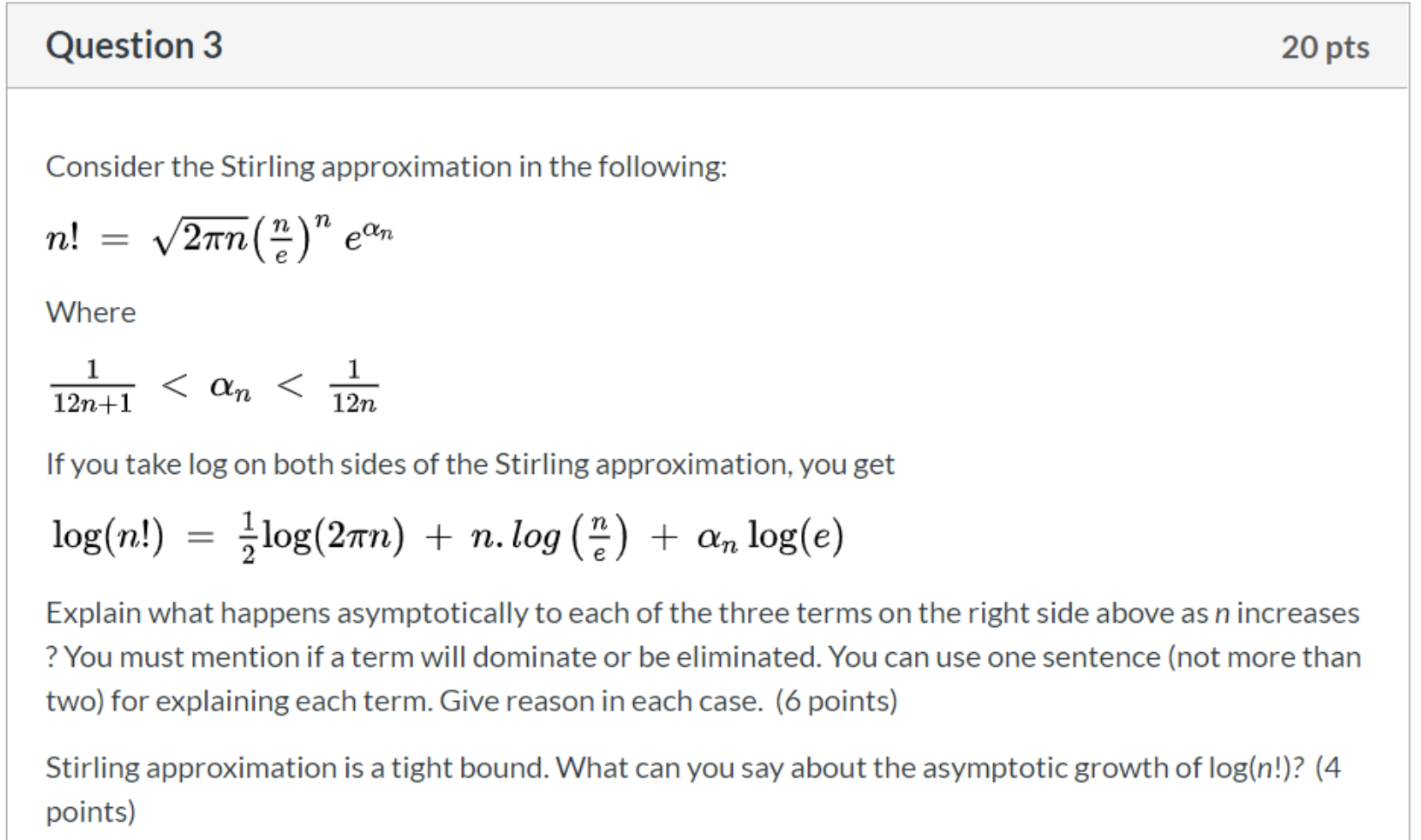


O-notation is an asymptotic upper bound which cannot represent the least running time but more suitable for the most time.



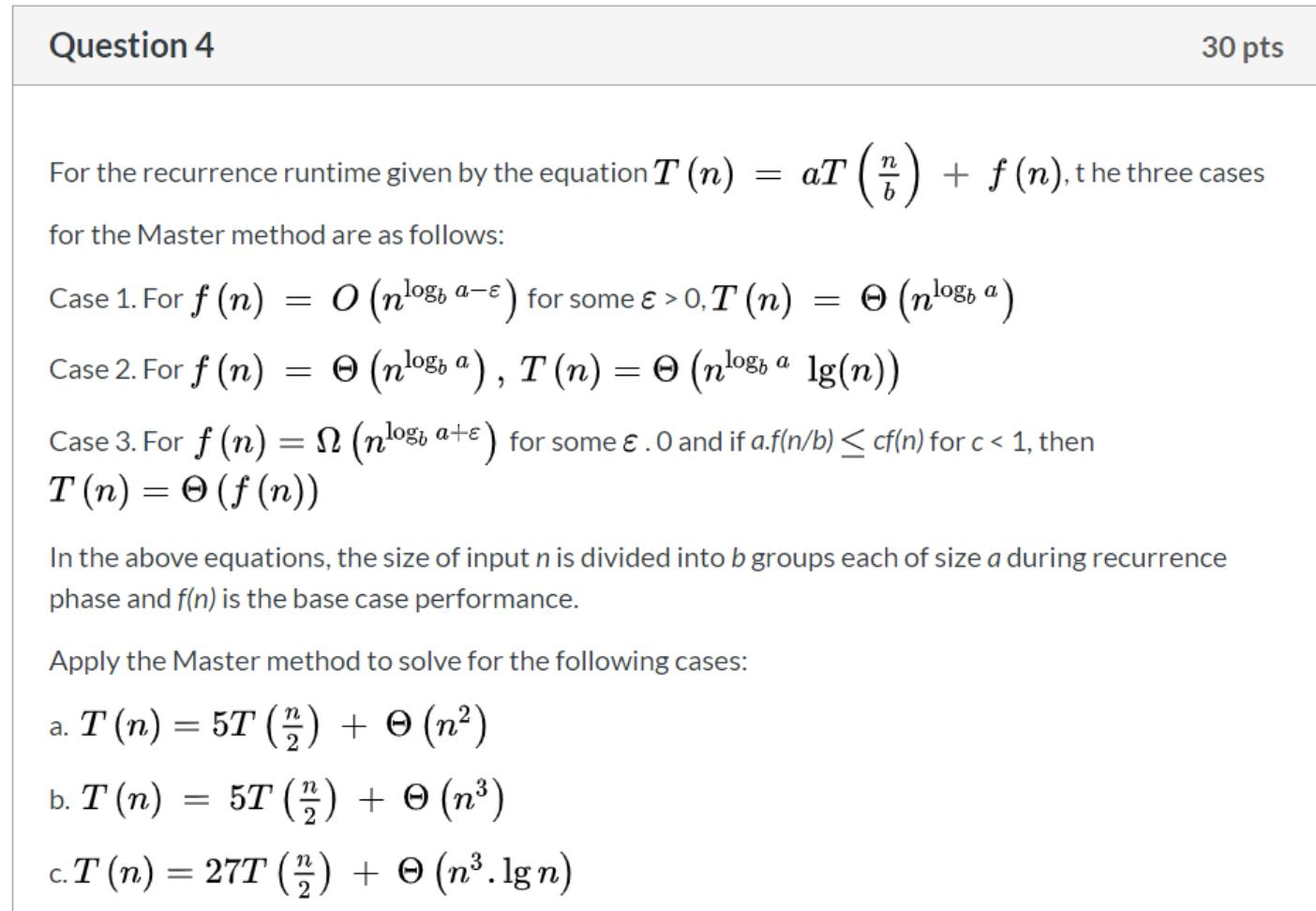
No. lg n^2 = 2\* lg n. And obviously O(2\* lg n)= O(lg n) . So O(lg n) and O(lg n^2) are same to some extends.



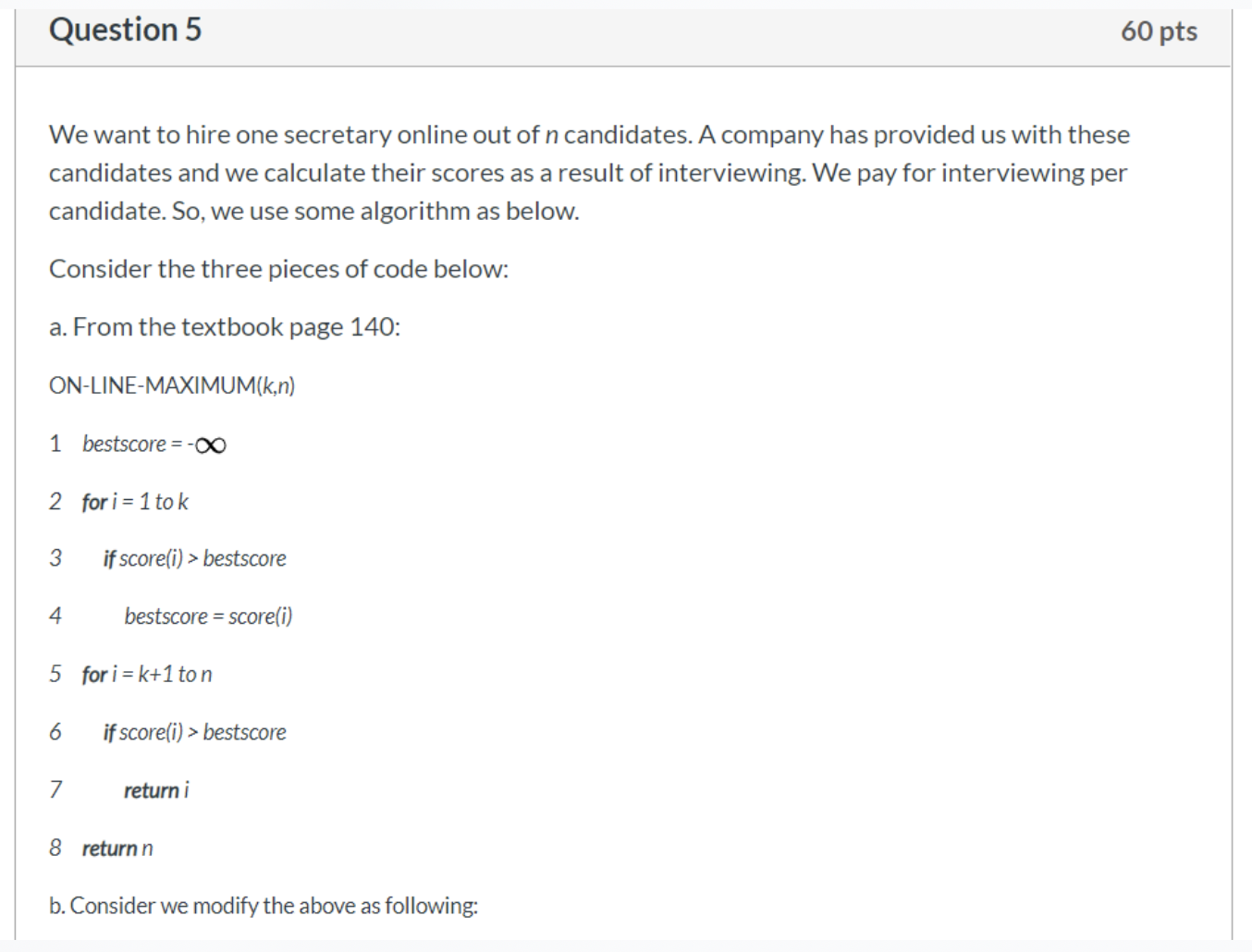
第一项增，第二项增，第三项减到无穷小。 然后log(n!) 近似 等于 nlogn。

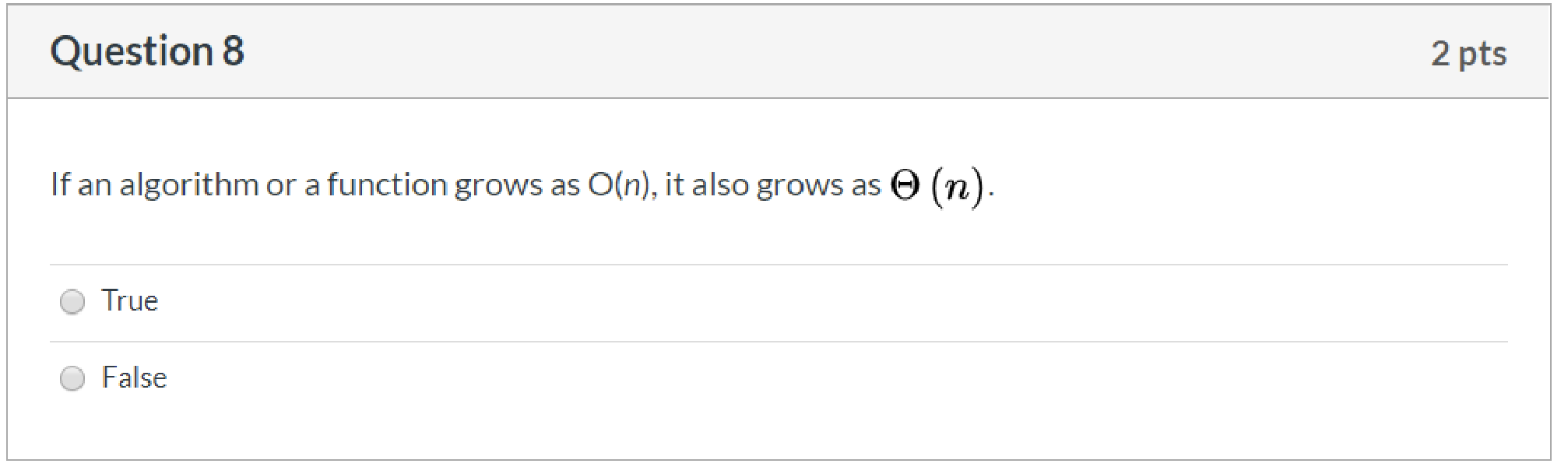
½\*log(2πn) : like log(x), this term will increase while n increases. But it’s growth rate will decrease.

n\*log(n/e): n\*log(n/e)

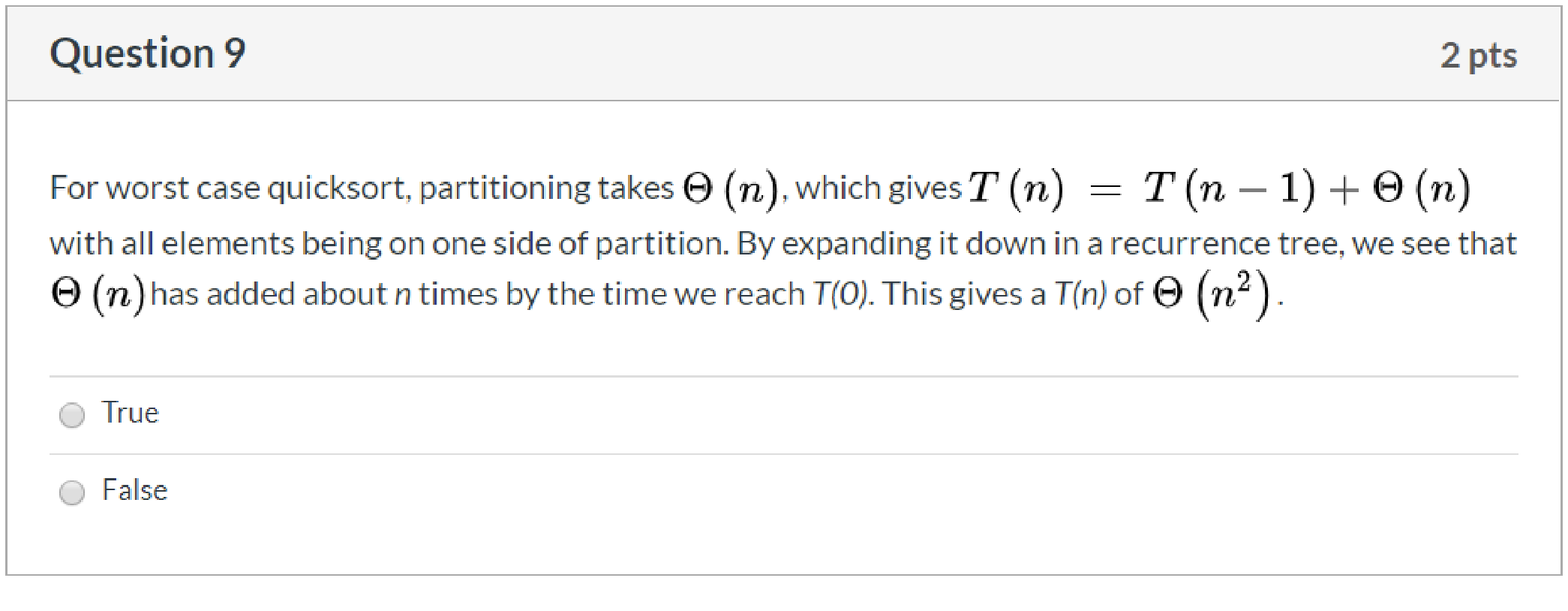


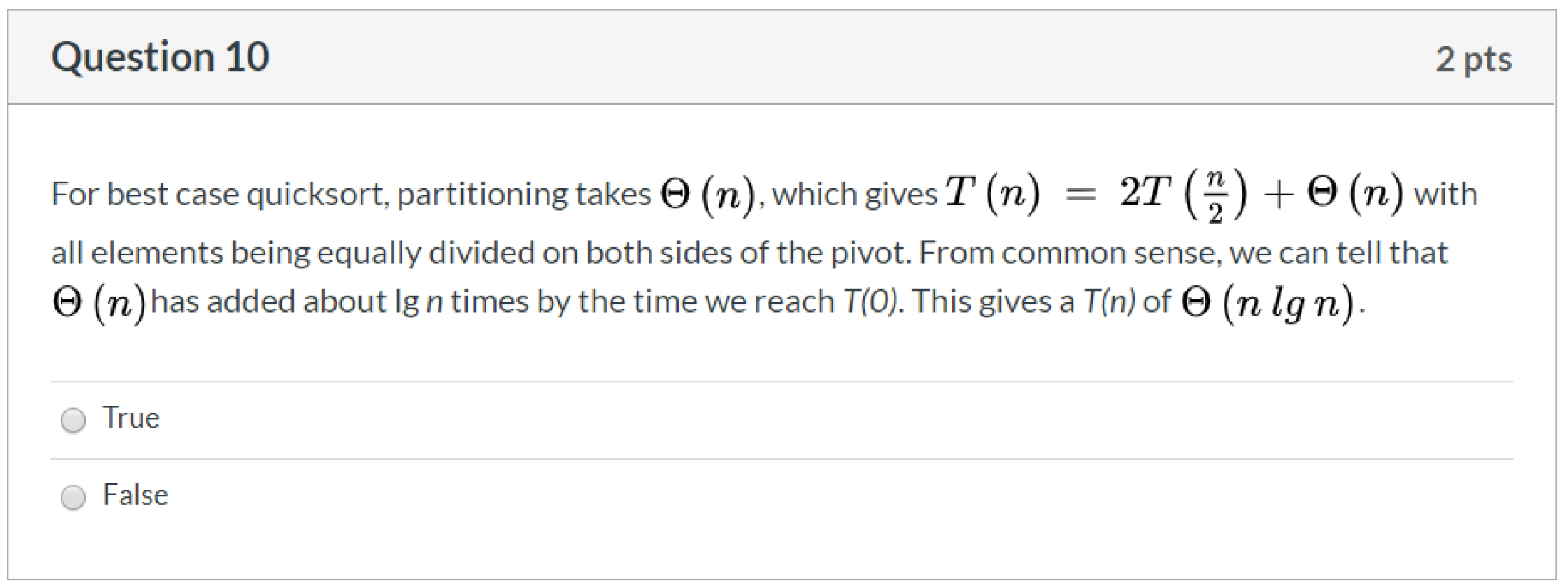
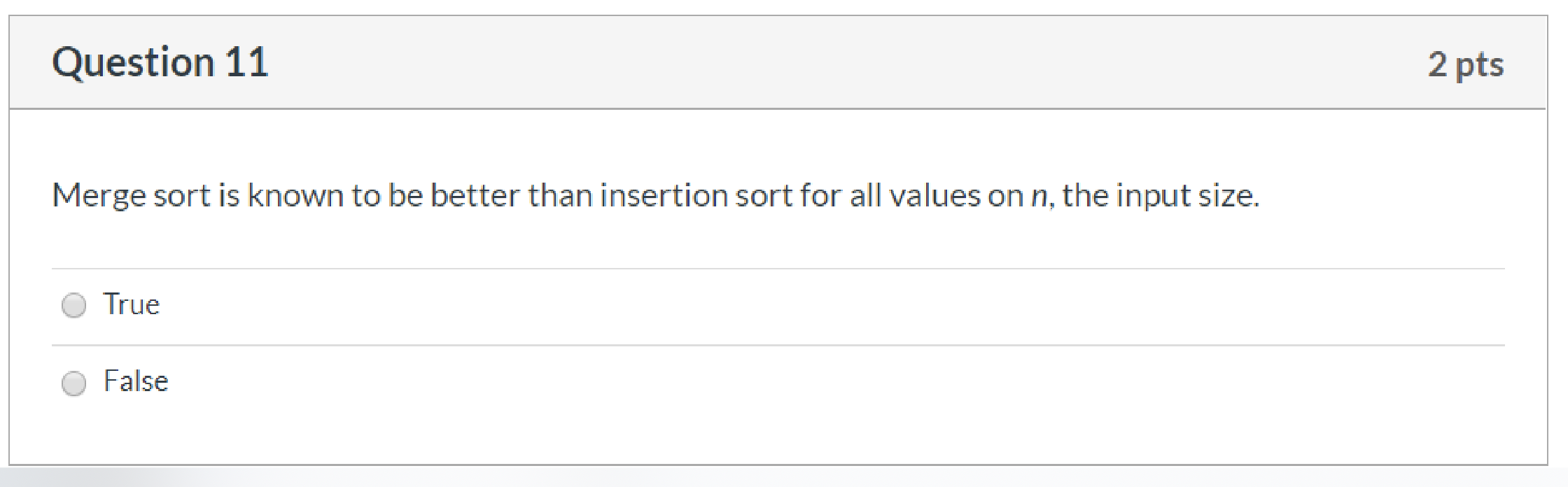
a.

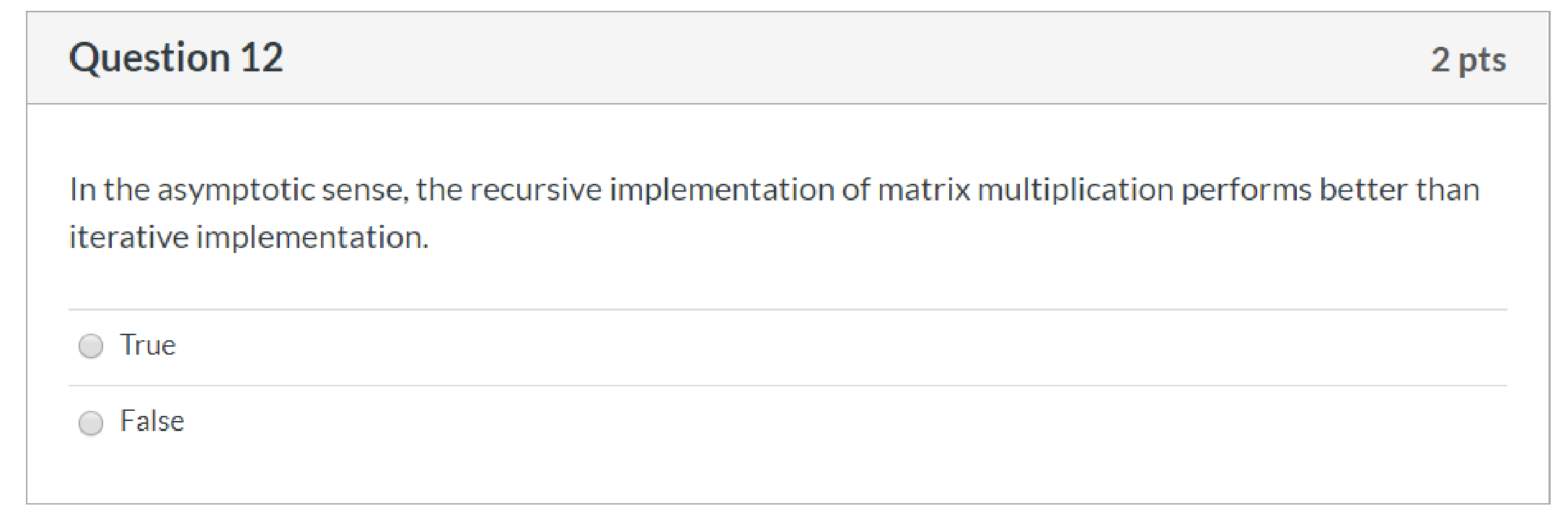
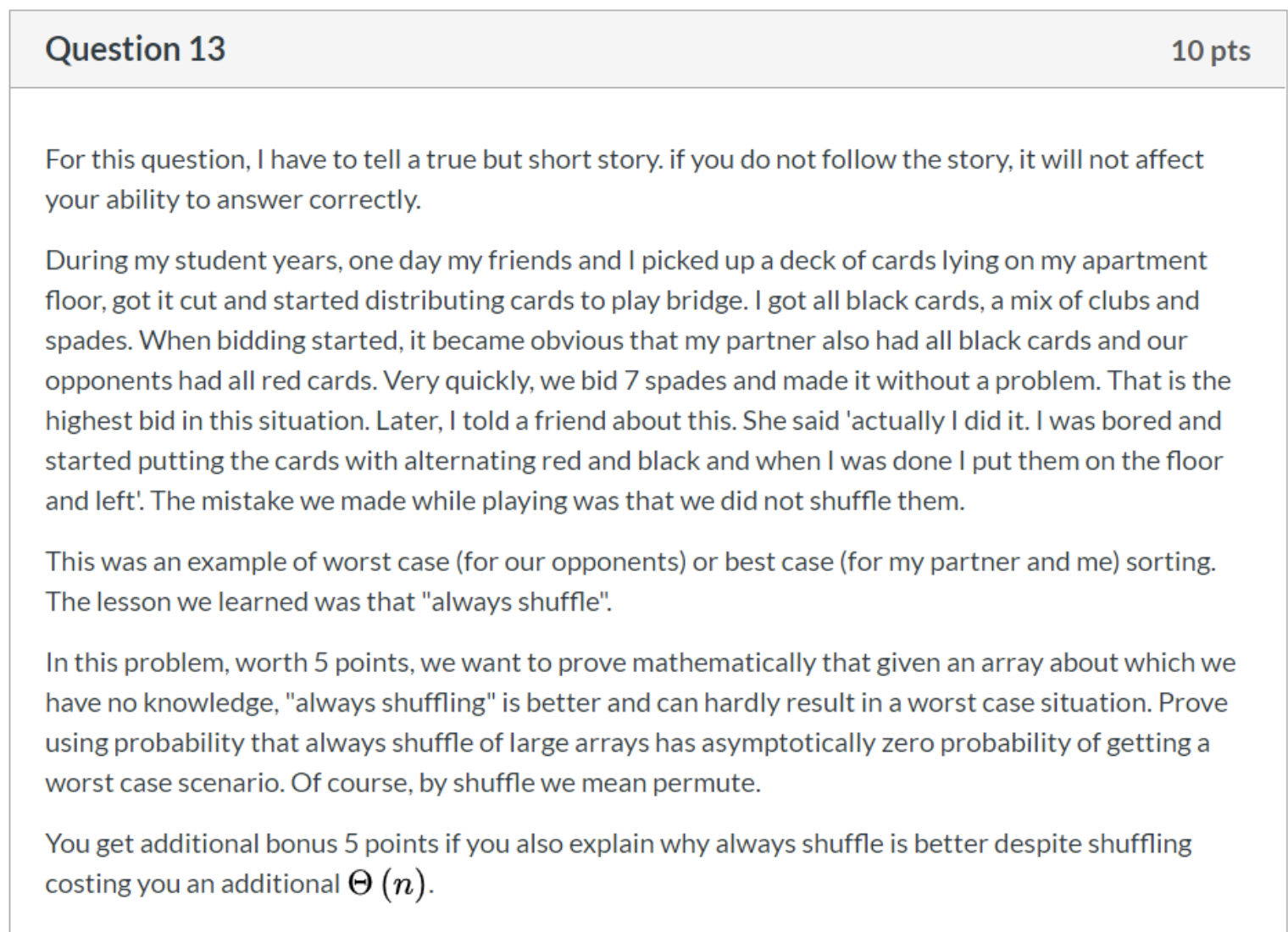




False

True

TrueFalse

False

For the first question, we can make an assumption about the worst case. For example, we think increasing sequence is the worst case in an algorithm. So, if we get an array in increasing sequence, the worst happen. But if we always shuffle the array at the begin of our algorithm, we can hardly get the worst case. What’s the probability that we can get from a shuffled array?

Assume there is an array with n element. If we want the worst case, we have to pick the smallest element randomly. The probability is 1/n. Then to pick the second-smallest element with the probability 1/(n-1). It goes on like this. At last, the probability to get the worst case for n-elements array is 1/n \* 1/(n-1) \* 1/(n-2) \* … \* 1/1 = 1/(n!). So, we have asymptotically zero probability of getting a worst case from a shuffled array.

For the second question. It’s true that the shuffling will cost us an additional Θ(n). But this additional time will be allocated to each operation in our algorithm. In most algorithm, we will have much more operation after a shuffle. So, this additional time has little impact on the running time.