# Assignment 2: Answer key and Common mistakes

November 16, 2015

(The numbers shown after each mistake is the *maximum* marks deducted for that mistake.)
**Answer 1.**

Book problem 1.3.3:

(b) is not possible because we can't pop 0 before 1.
(f) is not possible because we can't pop 1 before 7 and 2 is popped.
(g) is not possible because we can't pop 0 before 2.

Book problem 1.3.13:

Only (a) could occur. Because queues preserve the first in first out (FIFO) property. So, the numbers 0–9 should be dequeued in the same order as they were enqueued.

**Common mistakes in Answer 1.**

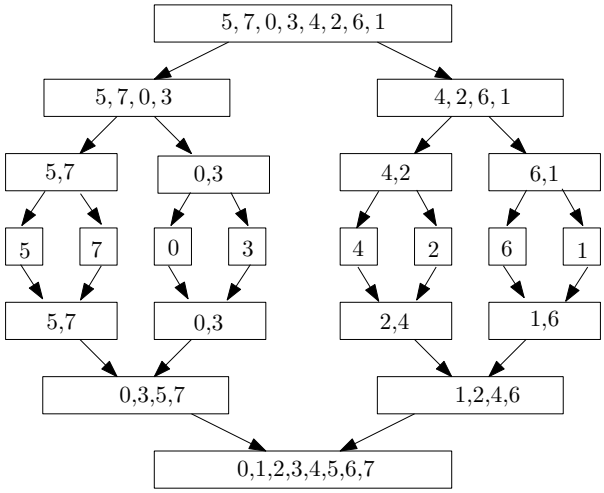
1.1 Didn't explain why a sequence can't occur. $-2$

**Answer 2.**

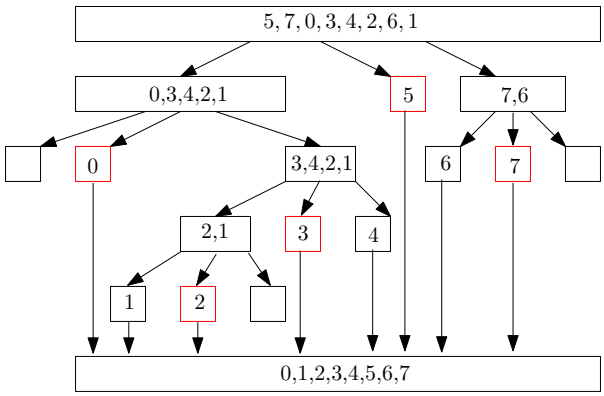
Inserion Sort: Here, red means the element that we will put in place next, green means the element that has just been placed in the correct spot.

$5, 7, 0, 3, 4, 2, 6, 1$
$5, 7, 0, 3, 4, 2, 6, 1$
$0, 5, 7, 3, 4, 2, 6, 1$
$0, 3, 5, 7, 4, 2, 6, 1$
$0, 3, 4, 5, 7, 2, 6, 1$
$0, 2, 3, 4, 5, 7, 6, 1$
$0, 2, 3, 4, 5, 6, 7, 1$
$0, 2, 3, 4, 5, 6, 1, 7$

Merge Sort:




Quick Sort: Red box means the pivot. People used different partitioning algorithm, but if they did it correctly no marks were taken off. The most common algorithm is shown below.




**Common mistakes in Answer 2.**

2.1 Mergesort tree is not correct. $-8$

2.2 Partitioning not shown correctly. $-5$

**Answer 3.**

It is actually insertion sort, where you increment a "count" variable every time you do a swap.

---

```
count = 0
for i = 1 to length(A) - 1
    j = i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j = j - 1
        count++
    end while
end for
```

---

The for loop takes $O(n)$ time. As the while loop only goes back until there is an inversion, the total time needed for the while loop is $O(k)$. So the total time complexity becomes $O(n + k)$. Note that if you just count back the inversions but do not actually swap the elements to put them in the right place, then your algorithm won't give you the correct number of inversions.

**Common mistakes in Answer 3.**

3.1 Time complexity was not explained correctly or not mentioned at all. $-5$

3.2 The divide and conquer method gives you the correct number of inversions, but the time complexity is $O(n \log n)$, not $O(n + k)$. $-10$

3.3 Your algorithm does not output the correct number of inversions. $-10$

**Answer 4.**

Strategy 1:

Suppose $c = 4$.

| n | Regular push | Special push | Stack size |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | (0+4)+0+1=5 | 4 |
| 2 | 1 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 1 | 0 | 4 |
| 5 | 0 | (4+4) + 4 + 1=13 | 8 |
| 6 | 1 | 0 | 8 |
| 7 | 1 | 0 | 8 |
| 8 | 1 | 0 | 8 |
| 9 | 0 | (8+4) + 8 + 1=21 | 12 |
| 10 | 1 | 0 | 12 |
| . | . | . | . |
| . | . | . | . |
| 13 | 0 | (12+4) + 12 + 1 = 29 | 16 |
| . | . | . | . |
| . | . | . | . |

So the total number of pushes is

$$
\begin{aligned}
&=8 + 16 + 24 + \dots \\
&=2c + 4c + 6c + \dots \\
&=2c \sum_{i=1}^{\frac{n}{c}} i \\
&=2c \frac{\frac{n}{c}(\frac{n}{c} + 1)}{2} \\
&=c \frac{n(n+c)}{c} \\
&=n^2 + nc \\
&=O(n^2)
\end{aligned}
$$

Strategy 2:

| n | Regular push | Special push | Stack size |
|---|---|---|---|
| 1 | 1 | 0 | 1 |
| 2 | 0 | (2*1)+1+1 = 4 | 2 |
| 3 | 0 | (2*2)+2+1 =7 | 4 |
| 4 | 1 | 0 | 4 |
| 5 | 0 | (2*4) + 4 + 1=13 | 8 |
| 6 | 1 | 0 | 8 |
| 7 | 1 | 0 | 8 |
| 8 | 1 | 0 | 8 |
| 9 | 0 | (2*8) + 8 + 1=25 | 16 |
| 10 | 1 | 0 | 16 |
| . | . | . | . |
| . | . | . | . |
| 17 | 0 | (2*16) + 16 + 1 = 49 | 32 |
| . | . | . | . |
| . | . | . | . |

So the total number of pushes is

$$
\begin{aligned}
&=1+4+8+16+32+\ldots \\
&=1+2^2+2^3+2^4+2^5+\ldots \\
&=(2^0+2^1+2^2+2^3+2^4+2^5+\ldots)-1 \\
&=\sum_{i=0}^{\log n} 2^i - 1 \\
&=(2^{\log n+1}-1)-1 \\
&=2\times 2^{\log n}-2 \\
&=2n-2 \\
&=O(n)
\end{aligned}
$$

Therefore, strategy 2 is better.

**Common mistakes in Answer 4.**

4.1 A closed form or a formula was required for both the strategies. $-5$

4.2 Strategy 2 is better, not strategy 1. $-5$

4.3 The calculation is not fully correct. $-5$

4.4 You need to show your work in details, how you got the formula and how you came to the conclusion that one strategy is better than the other. $-10$

**Answer 5.**

The function peek() shows you the top element of the stack. It is basically one pop and then one push operation.

```
Create stack S
push(0)
span[0] = 1
for i = 1 to length(A) - 1
    if (P[i] > P[i-1])
        while (S is not empty and P[S.peek()<P[i]]
            pop()
        end while
        if (S is empty)
            span[i] = i+1
        else
            span[i] = i-S.peek()
        endif
        push(i)
    endif
end for
```

Since every element is pushed and popped at most once, the number of stack operations is at most $2n$ (at most $4n$ if you count the peek() operations), where $n$ is the size of $A$. Therefore, the total time required is $O(n)$.

**Common mistakes in Answer 5.**

4.1 You can't use stack.size(), because to know the size of a stack, you have to pop all the elements and then push them back. That screws up the time complexity. We use stack to increase efficiency, so when you increase the time complexity like that it just doesn't make sense. $-10$

4.2 Algorithm is not correct $-10$

4.3 Didn't explain time complexity clearly (basically, did not count the push and pop operations correctly). $-5$