

W2D1 solution

Question 1: Design and Analysis of Algorithms

(a) Wooden Blocks Sorting Problem

Problem

We have an array of wooden blocks that are painted either Blue or Red. We need to arrange them so all Blue toys are at one end and all Red toys are at the other end.

My Algorithm

I'll use a two-pointer approach to solve this problem. The idea is to have one pointer starting from the left and another from the right, and swap elements when needed.

Algorithm:

1. Initialize `left = 0` and `right = n-1`
2. While `left < right`:
 - a. Move `left` pointer forward while `blocks[left]` is Blue
 - b. Move `right` pointer backward while `blocks[right]` is Red
 - c. If `left < right`, swap `blocks[left]` and `blocks[right]`
 - d. Increment `left` and decrement `right`
3. Done

Example

Let's say we have: [Red, Blue, Red, Blue, Red, Blue, Blue, Red]

Step-by-step execution:

Step	Array	left	right	What happened
Start	[Red, Blue, Red, Blue, Red, Blue, Blue, Red]	0	7	Initial state
1	[Red, Blue, Red, Blue, Red, Blue, Blue, Red]	0	6	Move right (skip Red at position 7)
2	[Blue, Blue, Red, Blue, Red, Blue, Red, Red]	0	6	Swap positions 0 and 6
3	[Blue, Blue, Red, Blue, Red, Blue, Red, Red]	2	5	Skip Blue at 1, now at position 2
4	[Blue, Blue, Blue, Blue, Red, Red, Red, Red]	2	5	Swap positions 2 and 5
5	[Blue, Blue, Blue, Blue, Red, Red, Red, Red]	4	3	Pointers crossed, stop

Final Result: [Blue, Blue, Blue, Blue, Red, Red, Red]

Is it In-Place?

Yes, my algorithm is in-place because:

- I only use two variables (left and right pointers)
- No extra array is created
- All operations are done on the original array itself

- Only constant extra space is used

Space Complexity

Space Complexity = O(1)

I'm only using a constant amount of extra space (two pointer variables and one temporary variable for swapping).

Time Complexity

Let me analyze different cases:

Best Case: O(n)

- This happens when the array is already sorted (all Blues on left, all Reds on right)
- Example: [Blue, Blue, Blue, Red, Red, Red]
- We still need to check all elements once

Average Case: O(n)

- Blues and Reds are randomly distributed
- Each element is examined once
- Some swaps occur but doesn't change the overall complexity

Worst Case: O(n)

- Maximum number of swaps needed
- Example: [Red, Blue, Red, Blue, Red, Blue] (alternating pattern)
- Even though we do many swaps, we still only visit each element once
- Total work = n comparisons + at most $n/2$ swaps = O(n)

Therefore, Time Complexity = O(n) in all cases

Summary

- **Time Complexity:** O(n)
- **Space Complexity:** O(1)
- **In-Place:** Yes
- **Number of passes:** Single pass through the array

This algorithm is efficient because it solves the problem in linear time with constant space.

(b) Three Colors: Blue, Red, and Green

Problem

Now we have three colors: Blue, Red, and Green. We need to arrange them so all blocks of the same color are together.

- Blue blocks at the beginning
- Green blocks in the middle

- Red blocks at the end

Algorithm:

1. Initialize three pointers:
 - low = 0 (boundary for Blue)
 - mid = 0 (current element being examined)
 - high = n-1 (boundary for Red)
2. While mid <= high:
 - a. If blocks[mid] = Blue:
 - Swap blocks[low] and blocks[mid]
 - Increment both low and mid
 - b. Else if blocks[mid] = Green:
 - Just increment mid (Green stays in middle)
 - c. Else (blocks[mid] = Red):
 - Swap blocks[mid] and blocks[high]
 - Decrement high (don't increment mid yet!)
3. Done

Example

Let's say we have: [Red, Blue, Green, Red, Blue, Green, Blue, Red, Green]

Step-by-step execution:

Step	Array	low	mid	high	Action
Start	[Red, Blue, Green, Red, Blue, Green, Blue, Red, Green]	0	0	8	Initial
1	[Green, Blue, Green, Red, Blue, Green, Blue, Red, Red]	0	0	7	Red: swap(0,8), high--
2	[Green, Blue, Green, Red, Blue, Green, Blue, Red, Red]	0	0	6	Red: swap(0,7), high--
3	[Blue, Blue, Green, Red, Blue, Green, Green, Red, Red]	0	0	5	Blue: swap(0,6), high--
4	[Blue, Blue, Green, Red, Blue, Green, Green, Red, Red]	1	1	5	Blue: swap(0,0), low++, mid++
5	[Blue, Blue, Green, Red, Blue, Green, Green, Red, Red]	2	2	5	Blue: swap(1,1), low++, mid++
6	[Blue, Blue, Green, Red, Blue, Green, Green, Red, Red]	2	3	5	Green: mid++
7	[Blue, Blue, Green, Green, Blue, Red, Green, Red, Red]	2	3	4	Red: swap(3,5), high--
8	[Blue, Blue, Blue, Green, Green, Red, Green, Red, Red]	2	3	3	Blue: swap(2,4), low++, mid++
9	[Blue, Blue, Blue, Green, Green, Red, Green, Red, Red]	3	4	3	Green: mid++
10	Done	3	5	3	mid > high, stop

Wait, let me recalculate this more carefully...

Actually, let me trace through it properly:

Step	Array	low	mid	high	blocks[mid]	Action
0	[Red, Blue, Green, Red, Blue, Green, Blue, Red, Green]	0	0	8	Red	Swap(0,8), high=7
1	[Green, Blue, Green, Red, Blue, Green, Blue, Red, Red]	0	0	7	Green	mid=1
2	[Green, Blue, Green, Red, Blue, Green, Blue, Red, Red]	0	1	7	Blue	Swap(0,1), low=1, mid=2
3	[Blue, Green, Green, Red, Blue, Green, Blue, Red, Red]	1	2	7	Green	mid=3
4	[Blue, Green, Green, Red, Blue, Green, Blue, Red, Red]	1	3	7	Red	Swap(3,7), high=6
5	[Blue, Green, Green, Red, Blue, Green, Blue, Red, Red]	1	3	6	Red	Swap(3,6), high=5
6	[Blue, Green, Green, Blue, Blue, Green, Red, Red, Red]	1	3	5	Blue	Swap(1,3), low=2, mid=4
7	[Blue, Blue, Green, Green, Blue, Green, Red, Red, Red]	2	4	5	Blue	Swap(2,4), low=3, mid=5
8	[Blue, Blue, Blue, Green, Green, Green, Red, Red, Red]	3	5	5	Green	mid=6
9	Done	3	6	5	-	mid > high

Final Result: [Blue, Blue, Blue, Green, Green, Green, Red, Red, Red]

Is it In-Place?

Yes, the algorithm is still in-place because:

- I only use three pointer variables (low, mid, high)
- No additional array is created
- All swaps are done within the original array
- Space usage is constant

Space Complexity

Space Complexity = O(1)

Only using three pointer variables and one temporary variable for swapping - constant space.

Time Complexity

Best Case: O(n)

- Array is already sorted: [Blue, Blue, Green, Green, Red, Red]
- We still examine each element once as mid goes from 0 to n

Average Case: O(n)

- Colors are randomly distributed
- Each element is examined exactly once
- Some swaps occur but mid pointer goes through array once

Worst Case: O(n)

- Maximum swaps needed
- Example: [Red, Red, Red, Green, Green, Blue, Blue, Blue]
- Even with many swaps, each element is visited exactly once by the mid pointer
- mid pointer travels from 0 to n: O(n) operations

Therefore, Time Complexity = O(n) in all cases

Comparison with Two Colors

Aspect	Two Colors (Part a)	Three Colors (Part b)
In-Place	Yes	Yes
Space Complexity	O(1)	O(1)
Time Complexity	O(n)	O(n)
Pointers Used	2 (left, right)	3 (low, mid, high)
Passes	Single pass	Single pass

Final Answer

- Is the algorithm in-place? YES
- Space Complexity: O(1) - constant extra space
- Time Complexity: O(n) - linear time

The algorithm remains efficient even with three colors because we still only need one pass through the array!

(c) Four Colors: Blue, Red, Green, and Yellow

Problem

Now we have four colors: Blue, Red, Green, and Yellow. We need to arrange them so all blocks of the same color are together.

My Algorithm - Extended Partitioning

- Blue blocks at the beginning
- Red blocks after Blue
- Green blocks after Red
- Yellow blocks at the end

Algorithm:

1. Initialize four pointers:

- blue = 0 (boundary for Blue region)
- red = 0 (boundary for Red region)
- green = 0 (current element being examined)
- yellow = n-1 (boundary for Yellow region)

2. While green <= yellow:

a. If blocks[green] = Blue:

- Swap blocks[blue] and blocks[green]
- If blocks[green] is now Red, swap blocks[red] and blocks[green]
- Increment blue, red, and green

b. Else if blocks[green] = Red:

- Swap blocks[red] and blocks[green]
- Increment red and green

c. Else if blocks[green] = Green:

- Just increment green (Green stays here)

d. Else (blocks[green] = Yellow):

- Swap blocks[green] and blocks[yellow]
- Decrement yellow (don't increment green!)

3. Done

Example

Let's say we have: [Yellow, Red, Blue, Green, Yellow, Red, Blue, Green]

Step-by-step execution:

Step	Array	blue	red	green	yellow	blocks[green]	Action
0	[Yellow, Red, Blue, Green, Yellow, Red, Blue, Green]	0	0	0	7	Yellow	Swap(0,7), yellow=6
1	[Green, Red, Blue, Green, Yellow, Red, Blue, Yellow]	0	0	0	6	Green	green=1
2	[Green, Red, Blue, Green, Yellow, Red, Blue, Yellow]	0	0	1	6	Red	Swap(0,1), red=1, green=2
3	[Red, Green, Blue, Green, Yellow, Red, Blue, Yellow]	0	1	2	6	Blue	Swap(0,2), then swap(1,2), blue=1, red=2, green=3
4	[Blue, Red, Green, Green, Yellow, Red, Blue, Yellow]	1	2	3	6	Green	green=4
5	[Blue, Red, Green, Green, Yellow, Red, Blue, Yellow]	1	2	4	6	Yellow	Swap(4,6), yellow=5
6	[Blue, Red, Green, Green, Blue, Red, Yellow, Yellow]	1	2	4	5	Blue	Swap(1,4), then swap(2,4), blue=2, red=3, green=5
7	[Blue, Blue, Red, Green, Green, Red, Yellow, Yellow]	2	3	5	5	Red	Swap(3,5), red=4, green=6
8	[Blue, Blue, Red, Red, Green, Green, Yellow, Yellow]	2	4	6	5	-	green > yellow, Done

Final Result: [Blue, Blue, Red, Red, Green, Green, Yellow, Yellow]

Is it In-Place?

Yes, the algorithm is still in-place because:

- I only use four pointer variables (blue, red, green, yellow)
- No additional array is needed
- All operations happen within the original array
- Space usage remains constant regardless of input size

Space Complexity

Space Complexity = O(1)

Only using four pointer variables and one temporary variable for swapping - still constant space!

Time Complexity

Let me analyze each case:

Best Case: O(n)

- Array is already sorted: [Blue, Blue, Red, Red, Green, Green, Yellow, Yellow]
- The green pointer still traverses through the entire array once
- Each element is examined once

Average Case: O(n)

- Colors are randomly distributed
- The green pointer goes through the array from left to right
- Each element is processed exactly once
- Some swaps occur but don't affect the linear nature

Worst Case: O(n)

- Maximum number of swaps
- Example: [Yellow, Yellow, Green, Green, Red, Red, Blue, Blue]
- Even though we perform multiple swaps, the green pointer visits each position at most once
- Total operations: n comparisons + swaps (swaps don't change complexity)

Therefore, Time Complexity = O(n) in all cases

Why is it still O(n)?

The key insight is that the **green pointer** always moves forward (except when we find Yellow, but yellow pointer moves backward). Each element is examined exactly once by the green pointer as it sweeps from left to right.

Comparison Table

Aspect	2 Colors	3 Colors	4 Colors
In-Place	Yes	Yes	Yes
Space Complexity	O(1)	O(1)	O(1)

Aspect	2 Colors	3 Colors	4 Colors
Time Complexity	$O(n)$	$O(n)$	$O(n)$
Pointers Used	2	3	4
Passes	Single	Single	Single

Final Answer for Part (c)

- Is the algorithm in-place? YES
- Space Complexity: $O(1)$ - constant extra space (only 4 pointers)
- Time Complexity: $O(n)$ - linear time in all cases

The pattern is clear: even as we add more colors, the algorithm remains:

- In-place (constant space)
- $O(n)$ time complexity (single pass)
- Just need one more pointer per additional color!

Quick Sort Illustration using Median-of-Three Rule

(a) Array: {1, 2, 3, 4, 5, 6, 7, 8, 9}

Step	Subarray	First	Mid	Last	Pivot	After Partition	Works
1	[1,2,3,4,5,6,7,8,9]	1	5	9	5	[1,2,3,4] 5 [6,7,8,9]	Pivot in position 4
2	[1,2,3,4]	1	2	4	2	[1] 2 [3,4]	Left subarray
3	[1]	-	-	-	-	[1]	Single element
4	[3,4]	3	-	4	3	3 [4]	Right of 2
5	[4]	-	-	-	-	[4]	Single element
6	[6,7,8,9]	6	7	9	7	[6] 7 [8,9]	Right subarray
7	[6]	-	-	-	-	[6]	Single element
8	[8,9]	8	-	9	8	8 [9]	Right of 7
9	[9]	-	-	-	-	[9]	Single element

Final Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9]

(b) Array: {8, 7, 6, 5, 4, 3, 2, 1, 9}

Step	Subarray	First	Mid	Last	Pivot	After Partition	Notes
1	[8,7,6,5,4,3,2,1,9]	8	4	9	8	[7,6,5,4,3,2,1] 8 [9]	Pivot in position 7
2	[7,6,5,4,3,2,1]	7	4	1	4	[1,2,3] 4 [6,5,7]	Left subarray
3	[1,2,3]	1	2	3	2	[1] 2 [3]	Left of 4
4	[1]	-	-	-	-	[1]	Single element
5	[3]	-	-	-	-	[3]	Single element
6	[6,5,7]	6	5	7	6	[5] 6 [7]	Right of 4
7	[5]	-	-	-	-	[5]	Single element
8	[7]	-	-	-	-	[7]	Single element
9	[9]	-	-	-	-	[9]	Single element

Final Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9]

(c) Array: {9, 1, 8, 2, 7, 3, 6, 4, 5}

Step	Subarray	First	Mid	Last	Pivot	After Partition	Notes
1	[9,1,8,2,7,3,6,4,5]	9	7	5	7	[1,2,3,6,4,5] 7 [8,9]	Pivot in position 6
2	[1,2,3,6,4,5]	1	6	5	5	[1,2,3,4] 5 [6]	Left subarray
3	[1,2,3,4]	1	2	4	2	[1] 2 [3,4]	Left of 5
4	[1]	-	-	-	-	[1]	Single element
5	[3,4]	3	-	4	3	3 [4]	Right of 2
6	[4]	-	-	-	-	[4]	Single element
7	[6]	-	-	-	-	[6]	Single element
8	[8,9]	8	-	9	8	8 [9]	Right subarray
9	[9]	-	-	-	-	[9]	Single element

Final Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9]

Median-of-Three Rule Explanation

The **median-of-three** pivot selection works as follows:

1. Examine the **first**, **middle**, and **last** elements of the subarray
2. Choose the **median** (middle value) of these three as the pivot
3. This helps avoid worst-case $O(n^2)$ performance on already sorted or reverse-sorted data
4. **Bold** numbers indicate elements in their final sorted position

QuickSelect Illustration :Median-of-Three Rule

(a) Array: {1, 2, 3, 4, 5, 6, 7, 8, 9}, k = 4

Goal: Find the 4th smallest element (k=4 means index 3 in 0-based indexing)

Step	Subarray	First	Mid	Last	Pivot	After Partition	Pivot Index	Comparison	Action
1	[1,2,3,4,5,6,7,8,9]	1	5	9	5	[1,2,3,4] 5 [6,7,8,9]	4	k=3 < 4	Search left
2	[1,2,3,4]	1	2	4	2	[1] 2 [3,4]	1	k=3 > 1	Search right
3	[3,4]	3	-	4	3	3 [4]	2	k=3 > 2	Search right
4	[4]	-	-	-	-	4	3	k=3 = 3	Found!

Result: The 4th smallest element is **4**

(b) Array: {8, 7, 6, 5, 4, 3, 2, 1, 9}, k = 5

Goal: Find the 5th smallest element (k=5 means index 4 in 0-based indexing)

Step	Subarray	First	Mid	Last	Pivot	After Partition	Pivot Index	Comparison	Action
1	[8,7,6,5,4,3,2,1,9]	8	4	9	8	[7,6,5,4,3,2,1] 8 [9]	7	k=4 < 7	Search left
2	[7,6,5,4,3,2,1]	7	4	1	4	[1,2,3] 4 [6,5,7]	3	k=4 > 3	Search right
3	[6,5,7]	6	5	7	6	[5] 6 [7]	5	k=4 < 5	Search left
4	[5]	-	-	-	-	5	4	k=4 = 4	Found!

Result: The 5th smallest element is **5**

(c) Array: {9, 1, 8, 2, 7, 3, 6, 4, 5}, k = 6

Goal: Find the 6th smallest element (k=6 means index 5 in 0-based indexing)

Step	Subarray	First	Mid	Last	Pivot	After Partition	Pivot Index	Comparison	Action
1	[9,1,8,2,7,3,6,4,5]	9	7	5	7	[1,2,3,6,4,5] 7 [8,9]	6	k=5 < 6	Search left
2	[1,2,3,6,4,5]	1	6	5	5	[1,2,3,4] 5 [6]	4	k=5 > 4	Search right
3	[6]	-	-	-	-	6	5	k=5 = 5	Found!

Result: The 6th smallest element is **6**

QuickSelect Algorithm Explanation

QuickSelect finds the k-th smallest element without fully sorting the array:

1. **Select Pivot:** Use median-of-three (first, middle, last elements)
2. **Partition:** Rearrange array so elements < pivot are on left, elements > pivot are on right
3. **Compare Pivot Position:**
 - o If pivot index = k-1: **Found the answer!**
 - o If pivot index > k-1: Search the **left** subarray
 - o If pivot index < k-1: Search the **right** subarray
4. **Repeat** until k-th element is found

Time Complexity:

- Average case: $O(n)$
- Worst case: $O(n^2)$ - but median-of-three helps avoid this

Key Advantage: Unlike QuickSort, QuickSelect only recurses into one subarray, making it more efficient for finding a single element.