

Question 1: Radix Sort (Radix = 9)

Array: {179, 721, 639, 549, 292, 427, 335, 435, 62}

Pass 1: Sort by (value mod 9)

- 179→8, 721→1, 639→0, 549→0, 292→4, 427→4, 335→2, 435→3, 62→8

After Pass 1: {639, 549, 721, 335, 435, 292, 427, 179, 62}

Pass 2: Sort by ($\lfloor \text{value}/9 \rfloor$ mod 9)

- 639→8, 549→7, 721→8, 335→1, 435→3, 292→5, 427→2, 179→1, 62→6

After Pass 2: {335, 179, 427, 435, 292, 62, 549, 639, 721}

Pass 3: Sort by ($\lfloor \text{value}/81 \rfloor$ mod 9)

- 335→4, 179→2, 427→5, 435→5, 292→3, 62→0, 549→6, 639→7, 721→8

Final Result: {62, 179, 292, 335, 427, 435, 549, 639, 721} ✓

Question 2: Sorting 4 Elements with 5 Comparisons

Algorithm:

Let the 4 elements be A, B, C, D.

1. Compare A vs B → Order as (small₁, large₁)
2. Compare C vs D → Order as (small₂, large₂)
3. Compare small₁ vs small₂ → Identify smallest overall
4. Compare large₁ vs large₂ → Identify largest overall
5. Compare the two middle elements (the loser of comparison 3 and winner of comparison 4)

After these 5 comparisons, we know the complete order.

Does this violate the lower bound?

No, this does NOT violate the theoretical lower bound. Here's why:

- **Theoretical lower bound:** For n elements, we need at least $\lceil \log_2(n!) \rceil$ comparisons
- For n = 4: $\lceil \log_2(4!) \rceil = \lceil \log_2(24) \rceil = \lceil 4.585 \rceil = 5$ comparisons

This algorithm is **optimal** because it achieves the theoretical lower bound exactly. The bound of $\lceil \log_2(n!) \rceil$ represents the minimum number of comparisons needed in the worst case to distinguish

between all $n!$ possible permutations. With 5 comparisons, we can distinguish between $2^5 = 32$ possible outcomes, which is sufficient for $4! = 24$ permutations.

Question 3: FBS Array Algorithm

Algorithm: FBSSort

```
FBSSort(A[1..n]):
```

1. Separate array into two parts:
 - ODD: elements at odd indices (1, 3, 5, ...)
 - EVEN: elements at even indices (2, 4, 6, ...)
2. Sort ODD in ascending order
3. Sort EVEN in descending order
4. Merge and adjust:

```
While there exists even[i] > odd[j] for any valid i, j:  
    Swap elements between EVEN and ODD to satisfy condition (3)  
    Re-sort ODD (ascending) and EVEN (descending) if needed
```
5. Interleave ODD and EVEN back into original positions

More Efficient Algorithm:

```
FBSSort(A[1..n]):
```

1. Sort entire array A in ascending order - $O(n \log n)$
2. Let $k = \lfloor n/2 \rfloor$ (number of even positions)
Let $m = \lceil n/2 \rceil$ (number of odd positions)
3. Place smallest k elements in even positions (in descending order)
Place largest m elements in odd positions (in ascending order)

Even positions: $A[k], A[k-1], \dots, A[1]$

Odd positions: $A[k+1], A[k+2], \dots, A[n]$

Example verification with {7, 20, 10, 19, 10, 17, 14, 15, 15}:

- Sorted: {7, 10, 10, 14, 15, 15, 17, 19, 20}
- $k = 4, m = 5$
- Even positions get {7, 10, 10, 14} in descending order $\rightarrow \{14, 10, 10, 7\}$
- Odd positions get {15, 15, 17, 19, 20} in ascending order $\rightarrow \{15, 15, 17, 19, 20\}$
- Result: {15, 14, 15, 10, 17, 10, 19, 7, 20} (not exactly the example, but valid FBS)

Asymptotic Running Time

My algorithm: $O(n \log n)$

- Sorting the array takes $O(n \log n)$
- Partitioning and placement takes $O(n)$
- Total: $O(n \log n)$

Fastest Possible Running Time: $\Omega(n \log n)$

Proof that $\Omega(n \log n)$ is the lower bound:

FBS array sorting is at least as hard as regular sorting because:

1. **Any FBS array can be converted to a fully sorted array in $O(n)$ time:** Simply read the odd positions (already sorted ascending) and even positions (reverse to get ascending), then merge.
2. **If FBSSort could be done in $o(n \log n)$, we could sort in $o(n \log n)$:**
 - Run FBSSort: $o(n \log n)$
 - Extract and merge odd/even: $O(n)$
 - Total: $o(n \log n)$
3. This would **contradict the comparison-based sorting lower bound** of $\Omega(n \log n)$.
4. Therefore, **any comparison-based FBSSort algorithm requires $\Omega(n \log n)$ time in the worst case.**

Conclusion: The fastest possible asymptotic running time is $\Theta(n \log n)$, and the algorithm I provided achieves this optimal bound.