# 3011. Find if Array can be sorted Example

| | |
|---|---|
| Created | @November 6, 2024 12:48 AM |
| Class | LeetCode |

## Initial Variables:

```
let previousMax = 0;         // Maximum value in the previous group
let currentMin = Infinity;   // Minimum value in the current group
let currentMax = -Infinity; // Maximum value in the current group
let previousBitCount = 0;    // Bit count of the previous number
```

**Explanation:**

- `previousMax`: Stores the maximum value from the previous group of numbers that had the same number of '1's in their binary representation.
- `currentMin`: Stores the minimum value in the current group.
- `currentMax`: Stores the maximum value in the current group.
- `previousBitCount`: Keeps track of the bit count of the previous number to detect when the bit count changes.

---

Now, we'll iterate over each number in the array `[8, 4, 2, 30, 15]`.

## Iteration 1: `num = 8`

```
const currentBitCount = num.toString(2).split('1').length -
1;
```

**Calculations:**

- **Binary Representation:** `8` in binary is `'1000'`.
- **Splitting by '1':** `'1000'.split('1')` results in `['', '000']`.
- **Length of Array:** `['', '000']` has a length of `2`.
- **`currentBitCount`**: `2 - 1 = 1`.

**Variables Before Conditional:**

- `previousBitCount = 0`
- `currentBitCount = 1`
- `currentMin = Infinity`
- `currentMax = -Infinity`
- `previousMax = 0`

**Conditional Check:**

- `previousBitCount (0) === currentBitCount (1)` → **False**

**Else Block Execution:**

- **Check if `currentMin < previousMax`:**
  - `currentMin (Infinity) < previousMax (0)` → **False**
- **Update Variables:**
  - `previousMax = currentMax (-Infinity)` → `previousMax = -Infinity`
  - `currentMin = currentMax = num (8)`
  - `previousBitCount = currentBitCount (1)`

**Variables After Update:**

- `previousMax = -Infinity`
- `currentMin = 8`

- `currentMax = 8`
- `previousBitCount = 1`

---

## Iteration 2: `num = 4`

### Calculations:

- **Binary Representation**: `4` in binary is `'100'`.
- **Splitting by '1'**: `'100'.split('1')` results in `['', '00']`.
- **Length of Array**: `['', '00']` has a length of `2`.
- **`currentBitCount`**: `2 - 1 = 1`.

### Variables Before Conditional:

- `previousBitCount = 1`
- `currentBitCount = 1`
- `currentMin = 8`
- `currentMax = 8`
- `previousMax = -Infinity`

### Conditional Check:

- `previousBitCount (1) === currentBitCount (1)` → **True**

### If Block Execution:

- **Update `currentMin` and `currentMax`:**
  - `currentMin = Math.min(8, 4) = 4`
  - `currentMax = Math.max(8, 4) = 8`

### Variables After Update:

- `currentMin = 4`
- `currentMax = 8`

---

## Iteration 3: `num = 2`

**Calculations:**

- **Binary Representation**: `2` in binary is `'10'`.
- **Splitting by '1'**: `'10'.split('1')` results in `['', '0']`.
- **Length of Array**: `['', '0']` has a length of `2`.
- **`currentBitCount`**: `2 - 1 = 1`.

**Variables Before Conditional:**

- `previousBitCount = 1`
- `currentBitCount = 1`
- `currentMin = 4`
- `currentMax = 8`
- `previousMax = -Infinity`

**Conditional Check:**

- `previousBitCount (1) === currentBitCount (1)` → **True**

**If Block Execution:**

- **Update `currentMin` and `currentMax`:**
  - `currentMin = Math.min(4, 2) = 2`
  - `currentMax = Math.max(8, 2) = 8`

**Variables After Update:**

- `currentMin = 2`
- `currentMax = 8`

---

## Iteration 4: `num = 30`

**Calculations:**

- **Binary Representation**: `30` in binary is `'11110'`.
- **Splitting by '1'**: `'11110'.split('1')` results in `['', '', '', '', '0']`.
- **Length of Array**: `['', '', '', '', '0']` has a length of `5`.

- `currentBitCount`: `5 - 1 = 4`.

### Variables Before Conditional:

- `previousBitCount = 1`
- `currentBitCount = 4`
- `currentMin = 2`
- `currentMax = 8`
- `previousMax = -Infinity`

### Conditional Check:

- `previousBitCount (1) === currentBitCount (4)` → **False**

### Else Block Execution:

- **Check if `currentMin < previousMax`:**
  - `currentMin (2) < previousMax (-Infinity)` → **False**
- **Update Variables:**
  - `previousMax = currentMax (8)`
  - `currentMin = currentMax = num (30)`
  - `previousBitCount = currentBitCount (4)`

### Variables After Update:

- `previousMax = 8`
- `currentMin = 30`
- `currentMax = 30`
- `previousBitCount = 4`

---

## Iteration 5: `num = 15`

### Calculations:

- **Binary Representation**: `15` in binary is `'1111'`.
- **Splitting by '1'**: `'1111'.split('1')` results in `['', '', '', '', '']`.

- **Length of Array**: `['', '', '', '', '']` has a length of `5`.
- `currentBitCount`: `5 - 1 = 4`.

### Variables Before Conditional:

- `previousBitCount = 4`
- `currentBitCount = 4`
- `currentMin = 30`
- `currentMax = 30`
- `previousMax = 8`

### Conditional Check:

- `previousBitCount (4) === currentBitCount (4)` → **True**

### If Block Execution:

- **Update `currentMin` and `currentMax`:**
  - `currentMin = Math.min(30, 15) = 15`
  - `currentMax = Math.max(30, 15) = 30`

### Variables After Update:

- `currentMin = 15`
- `currentMax = 30`

---

## After the Loop:

### Final Check:

- **Ensure Last Group's `currentMin` ≥ `previousMax`:**
  - `currentMin (15) >= previousMax (8)` → **True**

### Return Value:

- Since the final condition is met, the function returns `true`.

---

## Summary of Variable Values:

- **Group 1 (Bit Count = 1):** Numbers `[8, 4, 2]`
  - `currentMin = 2`
  - `currentMax = 8`
  - `previousMax` (after group) = `8`
- **Group 2 (Bit Count = 4):** Numbers `[30, 15]`
  - `currentMin = 15`
  - `currentMax = 30`

---

## Explanation of Each Variable:

- `previousMax`: The maximum value from the previous group. It's used to ensure that the current group's minimum value is not less than the previous group's maximum, maintaining a non-decreasing order across groups.
- `currentMin` **and** `currentMax`: The minimum and maximum values within the current group. They help track the range of values that can be rearranged within the group.
- `previousBitCount`: Tracks the number of '1's in the binary representation of the previous number. It's used to detect when we transition to a new group with a different bit count.

---

## Detailed Breakdown of `currentBitCount` Calculation:

For each `num`, here's how `currentBitCount` is calculated step by step:

1. **Convert `num` to Binary String:**
   - `num.toString(2)`
2. **Split the Binary String by '1's:**
   - `binaryString.split('1')`
3. **Calculate Length of the Resulting Array:**
   - `splitArray.length`
4. **Compute `currentBitCount`:**

- `currentBitCount = splitArray.length - 1`

**Let's apply this to each number:**

### `num = 8`

- **Binary:** `'1000'`
- **Split:** `['', '000']`
- **Length:** `2`
- `currentBitCount`: `2 - 1 = 1`

### `num = 4`

- **Binary:** `'100'`
- **Split:** `['', '00']`
- **Length:** `2`
- `currentBitCount`: `2 - 1 = 1`

### `num = 2`

- **Binary:** `'10'`
- **Split:** `['', '0']`
- **Length:** `2`
- `currentBitCount`: `2 - 1 = 1`

### `num = 30`

- **Binary:** `'11110'`
- **Split:** `['', '', '', '', '0']`
- **Length:** `5`
- `currentBitCount`: `5 - 1 = 4`

### `num = 15`

- **Binary:** `'1111'`

- **Split**: `['', '', '', '', '']`
- **Length**: `5`
- `currentBitCount`: `5 - 1 = 4`

---

## Understanding the Logic:

- **Grouping Numbers**: The code groups numbers based on the number of '1's in their binary representation (`currentBitCount`).
- **Within Groups**: Numbers within the same group can be rearranged freely (since swapping is allowed within the group) to achieve any order.
- **Between Groups**: To maintain non-decreasing order across different groups, the minimum value of the current group (`currentMin`) must not be less than the maximum value of the previous group (`previousMax`).

---

## Why This Works:

- **Swap Constraints**: You can only swap numbers that have the same number of '1's in their binary representation.
- **Ensuring Order**: By ensuring that `currentMin >= previousMax` when transitioning between groups, we guarantee that even after swapping within groups, the overall array can be sorted in non-decreasing order.
- **Failure Condition**: If `currentMin < previousMax`, it means there is a number in the current group that is less than a number in the previous group, and since we can't swap between groups, the array cannot be sorted.

---

## Conclusion:

- **Result**: The function returns `true` for the array `[8, 4, 2, 30, 15]`, indicating that it's possible to sort the array into non-decreasing order under the given swap constraints.
- **Key Takeaway**: The code effectively checks the possibility of sorting the array by ensuring the ordering between different bit count groups and utilizing swapping within those groups.

---

I hope this detailed step-by-step explanation clarifies how the code works and how the variables are updated at each step. Let me know if you have any more questions or need further clarification!