1295. Find Numbers with Even Number of Digits

Given an array `nums` of integers, return how many of them contain an **even number** of digits.

Example 1:

```
Input: nums = [12,345,2,6,7896]
Output: 2
Explanation:
12 contains 2 digits (even number of digits).
345 contains 3 digits (odd number of digits).
2 contains 1 digit (odd number of digits).
6 contains 1 digit (odd number of digits).
7896 contains 4 digits (even number of digits).
Therefore only 12 and 7896 contain an even number of digits.
```

Example 2:

```
Input: nums = [555,901,482,1771]
Output: 1
Explanation:
Only 1771 contains an even number of digits.
```

Constraints:

- `1 <= nums.length <= 500`
- `1 <= nums[i] <=` 10^5

Approach

This code identifies how many numbers in a list have an **even amount of digits** (for example: 12 has two digits, 4444 has four digits).

1. Main Function: `findNumbers(nums)` This function iterates through the list.

- **Counter:** It starts with `evenDigitCount = 0`.
- **The Loop:** It checks every `num` in the array.
- **The Condition:** If the helper function `hasEvenDigits` returns **true**, it increments the counter.
- **The Result:** It returns the total count of "even-digit" numbers.

2. Helper Function: `hasEvenDigits(num)` This function calculates the number of digits in a single number.

The Logic

1. **Start at 0:** It creates a variable `digitCount`.
2. **Divide by 10:** It uses a `while` loop to divide the number by 10 repeatedly.
    - *Example:* 123 → 12 → 1 → 0.
3. **Count:** Each time it divides, it adds +1 to `digitCount`.
4. **The Bitwise Trick:** Finally, it checks if the count is even using `(digitCount & 1) == 0`.

---

3. 🧠 Key Concept: Bitwise AND (&) The expression `(digitCount & 1) == 0` is a fast way to check if a number is **Even**.

- **Even numbers** (2, 4, 6...) always end in `0` in binary.
- **Odd numbers** (1, 3, 5...) always end in `1` in binary.
- Comparing a number with `& 1` looks only at that last bit.

1351. Count Negative Numbers in a Sorted Matrix

Given a `m x n` matrix `grid` which is sorted in non-increasing order both row-wise and column-wise, return *the number of* ***negative*** *numbers in* `grid`.

Example 1:

```
Input: grid = [[4,3,2,-1],[3,2,1,-1],[1,1,-1,-2],[-1,-1,-2,-3]]
Output: 8
Explanation: There are 8 negatives number in the matrix.
```

Example 2:

```
Input: grid = [[3,2],[1,0]]
Output: 0
```

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 100`
- `-100 <= grid[i][j] <= 100`

Follow up: Could you find an `O(n + m)` solution?

Approach

📌 1. Propiedades Clave de la Matriz

El ejercicio especifica que la matriz está ordenada de forma **descendente** tanto en filas como en columnas.

- **Horizontal:** Los valores disminuyen de izquierda a derecha.
- **Vertical:** Los valores disminuyen de arriba hacia abajo.

> 💡 **Insight clave:** Si encuentras un número negativo en una celda, todos los números que estén en esa misma columna pero en filas inferiores **también son negativos**.

🚀 2. Estrategia: Búsqueda en Escalera (Saddleback Search)

En lugar de un doble bucle `for` que tardaría $O(m \times n)$, usamos un enfoque de "frontera" que tarda $O(m + n)$.

El Punto de Partida

Iniciamos en la **esquina superior derecha**:

- `row = 0`
- `col = n - 1`

🛠️ 3. Lógica del Algoritmo

Mientras estemos dentro de los límites de la matriz, evaluamos el valor actual:

CASO A: El número es NEGATIVO (`grid[row][col] < 0`)

Si el número actual es negativo, aprovechamos el orden de la columna:

1. **Contar:** Sumamos todos los elementos restantes de esa columna: `count += (m - row)` .
2. **Mover:** Como ya procesamos toda esa columna, nos movemos a la izquierda: `col--` .

CASO B: El número es POSITIVO o CERO (`grid[row][col] >= 0`)

Si el número es positivo, sabemos que hacia la izquierda los números son aún mayores:

1. **Mover:** Bajamos de fila para buscar el límite de los negativos: `row++` .

📊 4. Análisis de Complejidad

- **Tiempo:** $O(m + n)$. En el peor de los casos, recorres el ancho y el alto de la matriz una sola vez.
- **Espacio:** $O(1)$. Solo utilizas variables constantes para los punteros y el contador.

💻 5. Implementación en JavaScript

```javascript /**

- @param {number[][]} grid
- @return {number}
- / var countNegatives = function(grid) { const m = grid.length; const n = grid[0].length;

  let count = 0; let row = 0; let col = n - 1;

  while (row < m && col >= 0) {

  ```
  if (grid[row][col] < 0) {
      // Sumamos el resto de la columna actual
      count += (m - row);
      // Movemos hacia la izquierda
      col--;
  } else {
      // Bajamos a la siguiente fila
      row++;
  }
  ```

  } return count; };

# 1491. Average Salary Excluding the Minimum and Maximum Salary

You are given an array of **unique** integers `salary` where `salary[i]` is the salary of the `i`$^{th}$ employee.

Return *the average salary of employees excluding the minimum and maximum salary*. Answers within $10^{-5}$ of the actual answer will be accepted.

**Example 1:**
  ```

```
Input: salary = [4000,3000,1000,2000]
Output: 2500.00000
Explanation: Minimum salary and maximum salary are 1000 and 4000 respectively.
Average salary excluding minimum and maximum salary is (2000+3000) / 2 = 2500
```

Example 2:

```
Input: salary = [1000,2000,3000]
Output: 2000.00000
Explanation: Minimum salary and maximum salary are 1000 and 3000 respectively.
Average salary excluding minimum and maximum salary is (2000) / 1 = 2000
```

Constraints:

- `3 <= salary.length <= 100`
- `1000 <= salary[i] <=` 10^6
- All the integers of `salary` are **unique**.

Approach

1. Comenzamos el ejercicio declarando las variables minimum maximum.
2. Declaramos la variable sum con valor inicial 0.
3. Inicializamos las variables minimum y maximum con el valor del primer elemento del arreglo.
4. Iteramos el arreglo salary con un ciclo for.
5. Dentro del ciclo evaluamos si la cantidad actual salary[i] es mayor al valor de maximum, si es mayor, asignamos el valor a maximum.
6. Debajo del condicional anterior, evaluamos si la cantidad actual salary[i], es menor al minimum, si es menor, asignamos el valor a minimum.
7. Dentro del ciclo for sumamos todos los valores en sum.
8. Una vez terminado el ciclo for, restamos el valor minimum y maximum de la suma total.
9. La suma total la dividiremos entre los elementos del arreglo sobrante, en este caso podemos usar la longitud del arreglo menos los dos elementos que ya restamos.
10. Retornamos la operación de promedio.

Code Explanation (Easy English)

1. First, we declare the variables minimum and maximum.
2. We declare the variable sum and give it a starting value of 0.
3. We initialize minimum and maximum with the value of the first element in the array.
4. We loop through the salary array using a for loop.
5. Inside the loop, we check if the current salary is greater than maximum. If it is, we update maximum.
6. Then, we check if the current salary is smaller than minimum. If it is, we update minimum.
7. Inside the loop, we add all the values to the sum.

8. After the loop ends, we subtract the minimum and maximum values from the total sum.
9. We divide the total sum by the number of remaining elements (the array length minus two).
10. Finally, we return the average.

```
var average = function(salary) {
    let minimum = salary[0];
    let maximum = salary[0];
    let sum = 0;

    for (let i = 0; i < salary.length; i++) {
        if (salary[i] > maximum) {
            maximum = salary[i]
        }
        if ( salary[i] < minimum ) {
            minimum = salary[i]
        }
        sum += salary[i]
    }
    sum = sum - (minimum + maximum)
    return sum / (salary.length - 2)
};
```

1480. Running Sum of 1d Array

Given an array `nums` . We define a running sum of an array as `runningSum[i] = sum(nums[0]…nums[i])` .

Return the running sum of `nums` .

Example 1:

```
Input: nums = [1,2,3,4]
Output: [1,3,6,10]
Explanation: Running sum is obtained as follows: [1, 1+2, 1+2+3, 1+2+3+4].
```

Example 2:

```
Input: nums = [1,1,1,1,1]
Output: [1,2,3,4,5]
Explanation: Running sum is obtained as follows: [1, 1+1, 1+1+1, 1+1+1+1, 1+1+1+1+1].
```

Example 3:

```
Input: nums = [3,1,2,10,1]
Output: [3,4,6,16,17]
```

Constraints:

- `1 <= nums.length <= 1000`
- `-10^6 <= nums[i] <= 10^6`

Approach

The function takes a list of numbers (called nums) and creates a new list. In the new list, each number is the sum of all previous numbers.

Step 1: The code creates an empty list called result. This is where we will store the answers.

Step 2: It creates a variable called sum and starts it at 0.

Step 3: The code uses a loop to look at every number (element) in the input list.

Step 4: Inside the loop, it adds the current number to the sum.

Example: If the current number is 3 and the sum was 2, the new sum is 5.

Step 5: It puts the new sum into the result list using .push().

Step 6: Finally, when the loop finishes, it returns the complete result list.

```
var runningSum = function (nums) {
    const result = []
    var sum = 0;
    for (const element of nums) {
        sum = element + sum;
        result.push(sum)
    }
    return result

};
```

1672. Richest Customer Wealth

You are given an `m x n` integer grid `accounts` where `accounts[i][j]` is the amount of money the `i`th customer has in the `j`th bank. Return *the **wealth** that the richest customer has.*

A customer's **wealth** is the amount of money they have in all their bank accounts. The richest customer is the customer that has the maximum **wealth**.

Example 1:

```
Input: accounts = [[1,2,3],[3,2,1]]
Output: 6
Explanation:
1st customer has wealth = 1 + 2 + 3 = 6
2nd customer has wealth = 3 + 2 + 1 = 6
Both customers are considered the richest with a wealth of 6 each, so return 6.
```

Example 2:

```
Input: accounts = [[1,5],[7,3],[3,5]]
Output: 10
Explanation:
1st customer has wealth = 6
2nd customer has wealth = 10
3rd customer has wealth = 8
The 2nd customer is the richest with a wealth of 10.
```

Example 3:

```
Input: accounts = [[2,8,7],[7,1,3],[1,9,5]]
Output: 17
```

Constraints:

- `m == accounts.length`
- `n == accounts[i].length`
- `1 <= m, n <= 50`
- `1 <= accounts[i][j] <= 100`

💰 Richest Customer Wealth: Approach Explanation

Approach

We'll use a **maximum tracking** approach combined with a **nested sum calculation**:

1. **Initialize maximum tracker:** Start with `richest = 0` as the baseline wealth to compare others against.

2. **Customer iteration:** Use a `for...of` loop to process each individual customer's account array inside the main accounts matrix.
3. **Wealth calculation:** For every customer, initialize a local `customerWealth` sum at 0 and add every bank account balance they own.
4. **Maximum comparison:** Use an `if` statement to compare the current customer's total wealth with the current `richest` value.
5. **Update tracking:** If the current customer is wealthier, update the `richest` variable with the new, higher value.
6. **Return result:** Once all customers are processed, return the final `richest` value representing the maximum wealth found.

```
var maximumWealth = function (accounts) {
    let richest = 0
    for (customer of accounts) {
        let customerWealth = 0
        for (wealth of customer) {
            customerWealth = customerWealth + wealth
        }

        if (customerWealth >= richest) {
            richest = customerWealth
        }
    }

    return richest
};
```

2529. Maximum Count of Positive Integer and Negative Integer

Given an array `nums` sorted in **non-decreasing** order, return *the maximum between the number of positive integers and the number of negative integers.*

- In other words, if the number of positive integers in `nums` is `pos` and the number of negative integers is `neg`, then return the maximum of `pos` and `neg`.

Note that `0` is neither positive nor negative.

Example 1:

```
Input: nums = [-2,-1,-1,1,2,3]
Output: 3
Explanation: There are 3 positive integers and 3 negative integers. The maximum count a
```

Example 2:

```
Input: nums = [-3,-2,-1,0,0,1,2]
Output: 3
Explanation: There are 2 positive integers and 3 negative integers. The maximum count a
```

Example 3:

```
Input: nums = [5,20,66,1314]
Output: 4
Explanation: There are 4 positive integers and 0 negative integers. The maximum count a
```

Constraints:

- `1 <= nums.length <= 2000`
- `-2000 <= nums[i] <= 2000`
- `nums` is sorted in a **non-decreasing order**.

Follow up: Can you solve the problem in `O(log(n))` time complexity?

Approach

We'll use a **dual-counter approach** with a final **maximum comparison**:

1. **Initialize counters:** Start two separate variables, `positiveCount` and `negativeCount`, at 0 to track the numbers.
2. **Array iteration:** Use a `for` loop to examine every number in the `nums` array from the beginning to the end.
3. **Positive check:** If the current number is greater than 0, increment the `positiveCount`.
4. **Negative check:** If the current number is less than 0, increment the `negativeCount`. *(Note: The code ignores zeros because 0 is neither positive nor negative).*
5. **Maximum comparison:** Compare the final `positiveCount` and `negativeCount` using an `if-else` structure.
6. **Return result:** Return the larger of the two counts as the final answer.

💡 Pro-tip (Simplified Step 5 & 6): You can replace the final if-else block with a single line using Math.max(): return Math.max(positiveCount, negativeCount);

```
var maximumCount = function(nums) {
    let positiveCount = 0;
    let negativeCount = 0;

    for (let i = 0; i < nums.length; i++ ) {
        if ( nums[i] > 0) {
            positiveCount++;
        } else if ( nums[i] < 0) {
            negativeCount++;
        }
    }

    if (positiveCount >= negativeCount) {
        return positiveCount
    } else {
        return negativeCount
    }
};
```