Find the Missing Number in an Unsorted Array of 1 to N

`nums` = [5,2,3,1]

```
def find_missing_number(nums):


Int map<int> mp; o(nums.length)

n(N+1)/2

Int mx = -1; // o(1)
Int x = nums.length(); // o(1)

for(int i = 0; i < x; i++){
      if(mx < nums[i]) mx = nums[i]; // o(n)

for(int i = 1; i <= mx; i++){
      Mp[nums[i-1]]++; 0(log n)
}

for(int i = 1; i <= mx; i++){
      if(mp[i] == false) return i; o(log n)
o(n);
```

```
def find_missing_number(nums):
    # Calculate the expected length of the array by adding 1 to the length of the given list
    n = len(nums) + 1

    # Calculate the expected sum of all elements from 1 to n using the formula (n * (n + 1)) // 2
    expected_sum = n * (n + 1) // 2

    # Calculate the actual sum of the elements in the given list using the built-in sum() function
    actual_sum = sum(nums)

    # Return the difference between the expected sum and the actual sum, which represents the missing number
    return expected_sum - actual_sum
```

What does this function do? Rename it with proper naming convention

```
int pre_sum(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n + pre_sum(n - 1);
    }
}
```

Valid subsequence

A subsequence of an array is a set of numbers that aren't necessarily adjacent in the array but that are in the same order as they appear in the array. For instance, the numbers [1, 3, 4] form a subsequence of the array [1, 2, 3, 4], and so do the numbers [2, 4]. Note that a single number in an array and the array itself are both valid subsequences of the array.

[5, 1, 22, 25, 6, -1, 8, 10] => Main array
[1, 6, -1, 10] => valid so the function should return true
[6, 1, 10, -1] => invalid so the function should return false

```
def isValidSubsequence(array, sequence):
Int n = array.length()
Int y = sequence.length();
Int cur = sequence[0];
Int index = 0;
for(int i = 0; i < n; i++){

  if(arr[i] == cur){
      Index++;
      if(indx < y) cur = index;
}

if(cur == y-1) return true;
Return false;
```

With normal recursion

```
def fibonacci(n):
   # Base cases: Fibonacci of 0 is 0, and Fibonacci of 1 is 1
   if n == 0:
      return 0
   elif n == 1:
      return 1
   else:
      # Recursive case: Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
      return fibonacci(n - 1) + fibonacci(n - 2)

# Example usage:


n = 1000000
result = fibonacci(n)
print(f"The {n}-th Fibonacci number is: {result}")
```

```
5
Fiv(4) + Fiv(3)
Fiv(3) + FIv(2) + Fiv(2) + Fiv(1)
Fiv(2) + Fiv(1) + Fiv(1+ Fiv(0) + Fiv(1+ Fiv(0) + Fiv(1)
Fiv(1) + Fiv(0) +  Fiv(1) + Fiv(1+ Fiv(0) + Fiv(1+ Fiv(0) + Fiv(1)
```

```
{
3 => 2
}
```

null <- 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 -> null

    - **Accessing the head**:
    - **Accessing the tail**:
    - **Accessing a middle node**:
    - **Inserting / Removing the head**:
- Accessing head Time Complexity is O(1)
- Accessing tail Time Complexity is O(1)
- O(length of linked-list)
- O(1)

```
   1
  / \
 2   3
/ \
4  5

Left-root-right

4 2 5 1 3

Root-left-right

1 2 4 5 3

Left-right-root

4 5 2 3 1

1 2 3 4 5
```

Problem: Increase the salary by 10% for all employees whose experience is greater than 5 years. Let's assume that the employee table has two columns along with other columns named salary and experience.

```
Update employee_table
Set salary = salary * 1.10
Where experience > 5;
```

FROM, ORDER BY, LIMIT, WHERE, GROUP BY, HAVING, SELECT, JOIN

Select
give me the execusion preference for these commands

```
SELECT department, COUNT(*) as emp_count
FROM employees
WHERE salary > 5000
GROUP BY department
HAVING COUNT(*) > 5
ORDER BY emp_count DESC
LIMIT 10;
```

🔄 Execution order:

1. **FROM** `employees`
2. **WHERE** `salary > 5000`
3. **GROUP BY** `department`
4. **HAVING** `COUNT(*) > 5`
5. **SELECT** `department, COUNT(*)`
6. **ORDER BY** `emp_count DESC`
7. **LIMIT** `10`