Description | Editorial | Solutions (3.5K) | Submissions

190. Reverse Bits

Easy | 4.4K | 1.2K

Companies

Reverse bits of a given 32 bits unsigned integer.

Note:

- Note that in some languages, such as Java, there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in **Example 2** above, the input represents the signed integer `-3` and the output represents the signed integer `-1073741825`.

Example 1:

```
Input: n = 00000010100101000001111010011100
Output:    964176192 (00111001011110000010100101000000)
Explanation: The input binary string
00000010100101000001111010011100 represents the unsigned integer
43261596, so return 964176192 which its binary representation is
00111001011110000010100101000000.
```

Example 2:

```
Input: n = 11111111111111111111111111111101
Output:   3221225471 (10111111111111111111111111111111)
Explanation: The input binary string
11111111111111111111111111111101 represents the unsigned integer
4294967293, so return 3221225471 which its binary representation
is 10111111111111111111111111111111.
```

Java | Auto

```java
public class Solution {
    // you need treat n as an unsigned value
    public int reverseBits(int n) {
        int rev = 0;
        for (int i = 0; i < 32; i++) {
            rev <<= 1;
            rev |= (n & 1);
            n >>= 1;
        }
        return rev;
    }
}
```

Testcase | Result

Accepted Runtime: 0 ms

Case 1 | Case 2

Input

```
00000010100101000001111010011100
```

Output

```
964176192 (00111001011110000010100101000000)
```

Expected

```
964176192 (00111001011110000010100101000000)
```

Contribute a testcase

Console | Run | Submit

Description | Editorial | Solutions (1.3K) | Submissions

2090. K Radius Subarray Averages

Hint

Medium | 1.4K | 75

Companies

You are given a **0-indexed** array `nums` of `n` integers, and an integer `k`.

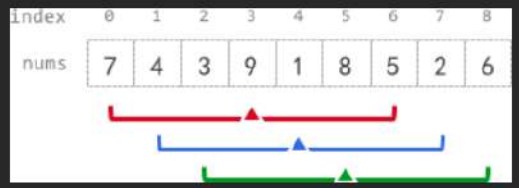
The **k-radius average** for a subarray of `nums` **centered** at some index `i` with the **radius** `k` is the average of **all** elements in `nums` between the indices `i - k` and `i + k` (**inclusive**). If there are less than `k` elements before **or** after the index `i`, then the **k-radius average** is `-1`.

Build and return *an array* `avgs` *of length* `n` *where* `avgs[i]` *is the* ***k-radius average*** *for the subarray centered at index* `i`.

The **average** of `x` elements is the sum of the `x` elements divided by `x`, using **integer division**. The integer division truncates toward zero, which means losing its fractional part.

- For example, the average of four elements `2`, `3`, `1`, and `5` is `(2 + 3 + 1 + 5) / 4 = 11 / 4 = 2.75`, which truncates to `2`.

Example 1:




```
Input: nums = [7,4,3,9,1,8,5,2,6], k = 3
Output: [-1,-1,-1,5,4,4,-1,-1,-1]
Explanation:
- avg[0], avg[1], and avg[2] are -1 because there are less than k elements before each index.
- The sum of the subarray centered at index 3 with radius 3 is: 7 + 4 + 3 + 9 + 1 + 8 + 5 = 37.
  Using integer division, avg[3] = 37 / 7 = 5.
```

Java | Auto

```java
class Solution {
    public int[] getAverages(int[] nums, int k) {
        if (k == 0) return nums;
        int[] res = new int[nums.length];
        Arrays.fill(res, -1);
        if (2*k + 1 > nums.length) return res;
        long[] prefixSum = new long[nums.length];
        prefixSum[0] = nums[0];
        for (int i = 1; i < prefixSum.length; i++) {
            prefixSum[i] = prefixSum[i - 1] + nums[i];
        }

        for (int i = k; i < nums.length - k; i++) {
            long sum = prefixSum[i + k] - prefixSum[i - k] + nums[i - k];
            res[i] = (int) (sum / (2 * k + 1));
        }

        return res;
    }
}
```

Testcase | Result

Accepted Runtime: 0 ms

Case 1 | Case 2 | Case 3

Input

nums =
[7,4,3,9,1,8,5,2,6]

k =
3

Output

Console | Run | Submit