Bit Manipulation_1

@kbbhatt04

@August 29, 2023

Operations on ith bit (Check, Set & Clear)

- Approach
    - Optimal
        - Using shift (<< or >>) operators
        - Time Complexity: $O(1)$
        - Space Complexity: $O(1)$

```
# Python3
# Optimal Solution
def bitManipulation(num : int, i : int) -> List[int]:
    i -= 1
    ans = []
    ans += 1 if (num & (1<<i)) else 0,
    num |= (1<<i)
    ans += num,
    num ^= (1<<i)
    ans += num,
    return ans
```

```
// C++
// Optimal Solution
vector<int> bitManipulation(int num, int i){
    i--;
    int a = num & (1<<i) ? 1 : 0;
    int b = num | (1<<i);
    int c = num & (~(1<<i));
    return {a, b, c};
}
```

Check if number is odd

- Approach
    - Optimal
        - If rightmost bit is 1 then odd else even
        - Time Complexity: $O(1)$
        - Space Complexity: $O(1)$

```
# Python3
# Optimal Solution
from typing import *

def oddEven(N : int) -> str:
    return "odd" if N & 1 else "even"
```

```
// C++
// Optimal Solution
string oddEven(int N){
    return (N & 1) ? "odd" : "even";
}
```

Check if number is power of 2

- Approach
    - Optimal
        - Powers of 2 have leftmost bit 1 and rest all bits set to 0
        - Therefore if we apply bitwise and between n and n-1 i.e. `n & (n-1)` we can check if the number is a power of 2 or not
        - Time Complexity: $O(1)$

        - Space Complexity: $O(1)$

```
# Python3
# Optimal Solution
class Solution:
    def isPowerOfTwo(self, n: int) -> bool:
        return False if n & (n-1) or n == 0 else True
```

```
// C++
// Optimal Solution
class Solution {
public:
    bool isPowerOfTwo(int n) {
        return (n & (long(n)-1)) || (n == 0) ? false : true;
    }
};
```

Swap 2 numbers

- Approach
    - Optimal
        - Using xor (^) and its properties i.e. a ^ a = 0
        - Time Complexity: $O(1)$
        - Space Complexity: $O(1)$

```
# Python3
# Optimal Solution
def swapNumber(a, b):
    a ^= b
    b ^= a
    a ^= b
    return a, b
```

```
// C++
// Optimal Solution
void swapNumber(int &a, int &b) {
    a ^= b;
    b ^= a;
    a ^= b;
}
```

Minimum Bit Flips to Convert Number X to Y

- Approach
    - Optimal
        - Using xor
        - x^y gives us the bits which are different in both numbers
        - return the count of 1's in x^y
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$

```
# Python3
# Optimal Solution
class Solution:
    def minBitFlips(self, start: int, goal: int) -> int:
        n = start ^ goal
        ans = 0
        while n:
            ans += 1 if n&1 else 0
            n >>= 1
        return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def minBitFlips(self, start: int, goal: int) -> int:
        return (start ^ goal).bit_count()
```

```
// C++
// Optimal Solution
class Solution {
public:
    int minBitFlips(int start, int goal) {
        int n = start ^ goal;
        int ans = 0;
        while(n) {
            ans += (n & 1) ? 1 : 0;
            n >>= 1;
        }
        return ans;
    }
};
```

```
// C++
// Optimal Solution
class Solution {
public:
    int minBitFlips(int start, int goal) {
        return __builtin_popcount(start^goal);
    }
};
```

L to R XOR

- Approach
    - Brute-force
        - Xor numbers from L to R
        - Time Complexity: $O(R - L + 1)$

        - Space Complexity: $O(1)$
    - Optimal
        - Observe the pattern: xor till 1 = 1 (1), till 2 = 3 (n+1), till 3 = 0 (0), till 4 = 4 (n)
        - This pattern repeats itself
        - Therefore
            - if num % 4 == 0, then num,
            - elif num % 4 == 1, then 1,
            - elif num % 4 == 2, then num + 1,
            - else 0
        - Now we need to just do xor of nums up to R and nums up to L - 1 i.e. x_R ^ x_L-1
        - Time Complexity: $O(1)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
def findXOR(L : int, R : int) -> int:
    ans = 0
    for i in range(L, R+1):
        ans ^= i
    return ans
```

```
# Python3
# Optimal Solution
def findXOR(L, R):
    def solve(num):
        if num % 4 == 0:    return num
        if num % 4 == 1:    return 1
        if num % 4 == 2:    return num + 1
        return 0

    return solve(R)^solve(L-1)
```

```
// C++
// Optimal Solution
int solve(int num)
{
    if(num%4==0)return num;
    else if(num%4==1)return 1;
    else if(num%4==2)return num+1;
    else return 0;
}

int findXOR(int L, int R){
    return solve(R)^solve(L-1);
}
```

Template

- Approach
    - Brute-force
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$
    - Better
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$

- Optimal
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```