Bitmask

1. Introduction

Bitmask provide an efficient way to manipulate a small set of Booleans that is stored as a 32-(or 64-)bit signed integer in base-10 but interpreted as a short 32-(or 64-) characters string.

By using bitwise operations, each bit of the integer can be checked, turned on (or turned off) easily and quickly. It can be used in various algorithms such as the Dynamic Programming solution for Travelling Salesman Problem to speed up crucial (small) set-based operations.

2. Visualisation

The visualisation of a bitmask data structure and its operations are shown above.

The top row describes the indices of the bits of the input integer **S** starting from the 0-th bit as the rightmost bit. Note that we use 1-digit hexadecimal notation of A/B/../E for the 10/11/../14-th bit.

The second row describes the bits of the input integer **S**. In practical applications, this is usually a 32-bit signed integer (e.g., `int` in C++/most other programming languages in 2022). Sometimes, this is a 64-bit signed integer (e.g., `long long` in C++). For this visualization, we limit **S** to 16-bit signed integer.

The third row describes the bits of the **(bit)mask** that will be applied to **S** together with the associated bitwise operation.

The last row describes **the result**.

3. Bitmask Operations

In this visualization, we currently support 6 Bitmask operations (all run in O(1)):

1. Set S (several ways)
2. Set j-th Bit
3. Check j-th Bit
4. Clear j-th Bit
5. Toggle j-th Bit

6. Least Significant Bit

3-1. Set S

We can enter a (small) Integer between [0..32767 (2^{15}-1)] in Decimal (base-10) and the Binary (base-2) form of **S** will be visualized.

Alternatively, we can select any random Integer between the same range [0..32767], a random powers of 2 (with specific pattern of '10...0'), or a random powers of 2 minus 1 (with specific pattern of '11...1').

3-2. Set j-th Bit

We can enter the index of the j-th Bit of S to be set (turned on).

Note that index starts from 0 but counted from the rightmost bit (refer to the top row).

The bitwise operation is simple: **S OR (1 << j)**.

Setting a bit that is already on will not change anything.

3-3. Check j-th Bit

We can enter the index of the j-th Bit of S to be checked (on whether it is on or off).

Note that index starts from 0 but counted from the rightmost bit (refer to the top row).

The bitwise operation is simple: **S AND (1 << j)**.

3-4. Clear j-th Bit

We can enter the index of the j-th Bit of S to be cleared (turned off).

Note that index starts from 0 but counted from the rightmost bit (refer to the top row).

The bitwise operation is simple: **S AND ~(1 << j)**.

Clearing a bit that is already off will not change anything.

3-5. Toggle j-th Bit

We can enter the index of the j-th Bit of S to be toggled (on → off or off → on).

Note that index starts from 0 but counted from the rightmost bit (refer to the top row).

The bitwise operation is simple: **S XOR (1 << j)**.

3-6. Least Significant One

This operation requires no input. It is a special operation to quickly identify the rightmost bit that is on in **S**.

The bitwise operation is simple: **S AND (-S)**.

Note that in Two's complement, **-S = NOT(S)+1**.

4. Challenges

For source code example involving bitmask/bit manipulation, please review: bit_manipulation.cpp | py | java | ml.

To test your understanding about bit manipulation, try our Online Quiz on bitmask topic.

And for more challenging problems involving bitmask/bit manipulation, try the following online judge problems: UVa 11933 - Splitting Numbers and Kattis - bitbybit.

Beyond these simple applications, bitmask frequently used as low-level optimizations in a few advanced algorithms, so get ready when you encounter bitmask as sub-component of the bigger algorithms.