Using Bitmask

1 Motivation

Suppose you have a set of objects and you want some way to represent which objects to pick and which ones not to pick. How do you represent that in in a program? More generally, how do you represent a subest of a set?One way is to use a Map to associate with each object a boolean value indicating whether the object is picked. Alternatively, if the object can be indexed by integers, you can use a boolean array. However, this takes up a lot of memory and can be slow due to the overhead of Map and array. If the size of the set is not too large, a bitmask is much more efficient (and convenient)!

2 What is a Bitmask?

We can represent whether an object is picked or not by a single bit! Using a boolean to represent this is an overkill in terms of memory usage. However, neither C++ nor Java has any data type representing a single bit, so how can we cut down the memory usage?

The answer is to use an integer! We know an integer is just a bunch of bits stringed together, so why don't we use the integer to represent the entire set? The 1^{st} bit will represent whether the 1^{st} object is picked, the 2^{nd} bit will represent whether the 2^{nd} object is picked or not, etc. For example, suppose in a set of 5 objects, we have picked the 1^{st}, 3^{rd}, and 4^{th} object. The bitmask to represent this in binary is *01101* or 13 in decimal (in the notes, the 1^{st} bit will always be the least significant bit and will always appear at the very right). We have just cut down the memory usage from five booleans to a single integer!

This sounds nice, but there is a serious limitation to this approach. An *int* is 32-bit, so what if the number of objects is greater than 32? In this case, we will have to use a *long long* in C++ or *long* in Java. So what if the number of object is greater than 64? Unfourtunately, bitmask becomes infeasible.

3 Manipulating Bitmask

Bitmask are not only memory efficient, they are also easy to manipulate. Both C++ and Java supports a variety of bitwise operators:

Bitwise Operators			
Operator	Name	Description	Example
`>>`	Right Shift	Shift every bit to the right, discard the lowest significant bit, and extend the sign bit	`-8 >> 3 = -1`
`<<`	Left Shift	Shift every bit to the left, discard the highest significant bit (sign is preserved), and add a 0 as the lowest significant bit	`-8 << 1 = -16`
`&`	Bitwise And	"And" every bit individually	`9 & 3 = 1`
`|`	Bitwise Or	"Or" every bit individually	`9 | 3 = 11`
`^`	Bitwise Xor	"Xor" every bit individually	`9 ^ 3 = 10`
`~`	Bitwise Not	Flip every bit	`~9 = -10`

With these operations, we can manipulate the bitmask in many ways:

Description	Code
Add the i^{th} object to the subset (set the i^{th} bit from 0 to 1)	`x = (x | (1 << i))`
Remove the i^{th} object from the subset (set the i^{th} bit from 1 to 0)	`x = (x - (1 << i))`
Check whether the i^{th} object is in the subset (check whether i^{th} bit is 1)	`(x & (1 << i)) != 0`
Iterate through all subsets of a set of size n	`for (x = 0; x < (1 << n); ++x)`
Iterate through all subsets of a subset y (not including empty set)	`for (x = y; x > 0; x = (y & (x-1)))`
Find the lowest index that is set to 1	`x & (-x)`

4 Final Remarks

Bitmask is an efficient and convenient way to represent subsets. For example, given a set of numbers, we want to find the sum of all subsets. This is easy to code using bitmasks. Furthermore, we can use an array to store all the results (imagine storing the results when you are using an array to represent the subset)!

```
int sum_of_all_subset(vector<int> s) {
        int n = s.size();
        int results[(1 << n)];

        // initialize results to 0
        memset(results, 0, sizeof(results));

        // iterate through all subsets
        for (int i = 0; i < (1 << n); ++i) {
                for (int j = 0;  j < n; ++j) {
                        if ((i & (1 << j)) != 0)
                                results[i] += s[j];
                }
        }
}
```

Finally, two word of caution on using bitmasks. First, always check the size of the set to determine whether to use an *int* or *long long* (*long* in Java) or not using bitmask at all. Secondly, always use parenthesis to indicate the precedence of operations when doing bitwise operations! The order of operations is very screwed up when it involves bitwise operators and not putting parenthesis when yield undesirable results! For example, let `x = 5`. Then `x - 1 << 2 = 16`, but `x - (1 << 2) = 1`.