Binary

Thomas Li

Outline

- Binary Representation
- Bitwise operations
- Applications

Binary Representation Recap

The binary number $(a_k a_{k-1} \ldots a_1 a_0)_2$ represents the number:

$$(a_k a_{k-1} \ldots a_1 a_0)_2 = a_k \cdot 2^k + a_{k-1} \cdot 2^{k-1} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0.$$

Ex: 1101 = 2^3 + 2^2 + 2^0 = 13

We'll say each digit in binary is a **bit**, and a bit is **set** if it's 1, and **cleared** if it's 0.

The least significant bit (**LSB**) is the smallest **set** bit in the number

The most significant bit (**MSB**) is the largest **set** bit in the number

Data Types

Computers represent integers in **fixed width** binary, typically up to 64 bits

=> Bit operations are extremely fast

Commonly used in C++:

int (32 bit **signed** integer)

long long (64 bit **signed** integer)

(u)int32_t (u stands for **unsigned**)

(u)int64_t

Two’s Complement

How do you represent negative numbers in binary?

Suppose we have fixed width of *w* bits

Negate the largest bit. So if the largest bit in a binary representation is set, subtract 2^(w-1) instead of adding it.

Ex: w=5, 11001 = -16 + 8 + 1 = -7

Two’s Complement 2

The w-1 smallest bits can represent the range [0,2^(w-1)-1]

If we use the largest bit, we’ll create the shifted range [-2^(w-1),-1]

So we can represent [-2^(w-1),2^(w-1)-1] in total

How to convert value x to -x?

-x = ~x + 1

Intuition: x + ~x = every bit is 1, +1 causes overflow to 0. So x + (~x + 1) = 0, so (~x+1) represents additive inverse of x.

Left as an exercise to show that two’s complement has nice properties

Bitwise operations

Logical operators applied on each **bit** of a binary number. It doesn’t care about signedness

Bitwise AND: 11001 & 10100 = 10000

Bitwise OR: 11001 | 10100 = 11101

Bitwise XOR: 11001 ^ 10100 = 01101

Bitwise NOT: ~11001 = 00110

Shift operations

Also have operations that shift bits left or right.

Ex: 11001 << 2 = 00100, 11001 >> 1 = 01100

Effect on **non negative** numbers:

x << k = x * 2^k

x >> k = floor(x / 2^k)

Effect on **negative** numbers???

Takeaway

Avoid using bitwise operations on values that could be negative, as representation could be confusing / undefined

Operations

How to get a power of 2?

How to get the value of a bit?

How to set a bit? How to clear a bit?

Count the number of set bits?

Find LSB? MSB?

Binary Applications

Representing sets

We can represent **subsets** of a given set as a bitmask.

Order the elements S0,S1,...,Sn, and consider a binary number, where

the i-th bit (bit corresponding to 2^i) is set iff Si is in the subset.

Ex: {A,B,D} ⊆ {A,B,C,D} -> 1011

Then **OR** = set union, **AND** = set intersection, **XOR** = ???

??? = Set difference

Subsets of a set of *n* elements biject with the integers [0,2^n-1], perfectly compact

Application to DP

We have a nice way to *efficiently* represent a subset

Now we can do DP with a subset state, and easily store the state in an array.

Travelling Salesman Problem

Given a weighted graph (V,E), what's the minimum cost to visit every node exactly once and return to the origin node?

Well known NP-hard problem

Do DP: dp[S][u] = minimum cost to reach nodes in set S $\subseteq$ V if the last visited node is u.

Represent S as bitmask, so, dp is an (2^|V|) x |V| 2D array

Implementation

Assuming a complete graph in adjacency matrix *adj*, starting at *start*

Set all dp entries to infty

```
dp[1<<start][start] = 0
for msk in range(1<<len(adj)):
        for u in range(len(adj)):
                if (msk >> u & 1) == 1:
                        for v in range(len(adj)):
                                if (msk >> v & 1) == 0:
                                        dp[msk | (1<<v)][v] <- dp[msk][u] + adj[u][v]
```

Questions

What if we can visit nodes multiple times?

What if we can start from any node?

What if n is too big?

Binary lifting

Suppose we have a monotonic predicate F on non-negative integers

Ex: F(0) = F(1) = F(2) = F(3) = 0, 1 = F(4) = F(5) = …

We want to find the first 1 in [0,N]

Consider the following algorithm:

Binary Lifting 2

```
int cum = 0;
if(F(0)) return 0;
for(int i = highest bit of N; i >= 0; i--){
        if(!F(cum+(1<<i))) cum += 1<<i;
}
return cum+1;
```

Binary Lifting 3

Correctness:

Let k be the largest integer where F(k) = 0, exists based on the if statement

Look at binary representation of k.

Can show inductively that after the *x*-th iteration of the loop, the top *x* bits of **k** and **cum** will be identical

Finally add 1 to get first F(x) = 1

Binary Lifting 4

Given a functional graph (a graph where each node has **exactly** one outgoing edge) represented as an array **nxt**

Answer queries (u,k) which asks what's the node reached after going to the "next" node **k** times. k <= 10^18

Solution

Compute an array jump[u][p] = the node reached if we apply **nxt** *2^p* times.

Array has size N x L

Base case: jump[u][0] = nxt[u]

If we have jump[u][p], 2^(p+1) = 2^p + 2^p, so

jump[u][p+1] = jump[jump[u][p]][p] (we apply a 2^p jump twice)

Can compute array in O(NL)

Solution 2

To answer a full query **(u,k)**, consider binary representation of k.

We can apply the **jump** array on each **set** bit of k.

```
int ans = u;
for(int i = 0; i <= highest bit of k; i++){
        if(k >> i & 1) ans = jump[ans][i];
}
return ans;
```

We need L >= max(log2(k)), so complexity is O(N log maxk) preprocess, O(log k) per query

Questions

Does it matter what order we iterate the bits?

There exists an O(log N) per query solution (and even an O(1) per query way). How to do it?

Practice Contest

https://vjudge.net/contest/669082

Sum over subsets

We'll represents subsets as a bitmask for this section.

We want to compute $dp[S] = \sum_{U \subseteq S} a[U]$ for some array a of size 2^N.

On Multidimensional Prefix Sums

How do you compute 2D prefix sums?

One way: inclusion exclusion – takes exponential time relative to dimension

Other way: build sums over a prefix of dimensions

Let psa[0][x][y] = original array

Let psa[1][x][y] = sum(psa[0][w][y]) for all w <= x – **Row sums**

We have **psa[1][x][y] = psa[1][x-1][y]+psa[0][x][y]**

Let psa[2][x][y] = sum(psa[0][w][z]) for all w <= x, z <= y – **Prefix sums**

We have psa[2][x][y] = psa[1][x][y]+psa[2][x][y-1] (we add our row to the rest of the sum)

2

Let dp[i][x] = the partial sum applied on the first **i** dimensions on “coordinate” **x**

We compute it in the same way as 2D case, takes O(N * # elements) time, if we can compute a coordinate index in O(1)

We can also drop the dimensional dimension.

Observe that sum of subsets is just a N-dimensional prefix sum over {0,1}^N !

So we have the algorithm:

```
dp = a
for d in range(N):
    for msk in range(1<<N):
        if msk >> d & 1:
            dp[msk] += dp[msk ^ (1<<d)]
```

Questions

How to compute the sums for each k x k … k cube?