Ezoteryczne Kartki
Operacje bitowe

Jakub Bachurski

wersja 1.2.0.3

1 Podstawy

Operacje bitowe to operacje działające na reprezentacjach bitowych liczb. Obsługują je wszystkie typy liczbowe.[1]

Binarne operacje (tzn. z dwoma parametrami) bitowe działają jak odpowiednie operacje logiczne na każdej parze odpowiadających bitów obu liczb.

Operator	Operator logiczny	Warunek `1`
`&`	`&&, and`	oba bity to `1`
`|`	`||, or`	co najmniej jeden z bitów to `1`
`^`	`(xor)`	dokładnie jeden z bitów to `1`

Przykłady (liczymy `155 & 25`, `100 | 172`, `87 ^ 248`):

`1001 1011`	`0110 0100`	`0101 0111`
`& 0001 1001`	`| 1010 1100`	`^ 1111 1000`
`0001 1001`	`1110 1100`	`1010 1111`

Do tego dochodzą przesunięcia bitowe *(bitshifty)*, które działają na trochę innej zasadzie:

- `x << s` to liczba x przesunięta o s bitów w lewo. `5 << 2 = 20`
- `x >> s` to liczba x przesunięta o s bitów w prawo. `42 >> 3 = 5`

Jeżeli jakiś bit „wyjdzie” poza wielkość typu (np. 32 bity `unsigned`), to przepada. `(3 >> 1) << 1 = 2`.

Jest jeden operator unarny (tzn. z jednym parametrem): `~` (operator negacji, czyli `not`), odpowiadający `!`: neguje *wszystkie* bity.

Uwaga. Operacje bitowe mają bardzo niski priorytet w kolejności operatorów, więc warto opatrzyć je nawiasami. Więcej o priorytetach tutaj: [operator precedence].

[1] W tej kartce „typy liczbowe” rozumiane są jako typy reprezentujące liczby całkowite: `int`, `unsigned`, `long` itd.

2 Dodatkowe operacje – rozszerzenia g++

`g++` udostępnia nam jeszcze trochę dodatkowych operacji, dostępnych jako funkcje. Nie wszystkie procesory je obsługują, ale w razie czego `g++` zapewni własną, szybką implementację.

Funkcja	Działanie
`__builtin_clz(x)`	`count leading zeroes`. Liczba zer wiodących.
`__builtin_ctz(x)`	`count trailing zeroes`. Liczba zer kończących.
`__builtin_popcount(x)`	Liczba bitów zapalonych (czyli `1`).

Każda z funkcji działa domyślnie na `unsigned int`, ale ma odpowiedniki działające na `unsigned long` i `unsigned long long` – mają one dodatkowy sufiks `l` lub `ll`. Przykładowo, rodzina `clz` to `__builtin_clz`, `__builtin_clzl`, `__builtin_clzll`. Wszystkie funkcje zwracają `int`-a. Zachowanie `clz` i `ctz` na `0` jest niezdefiniowane.

3 Ciekawe własności

- Bitshifty o k odpowiadają pomnożeniu/podzieleniu (z podłogą) przez 2^k. Zatem potęgi dwójki można liczyć przez `1 << k`. *Uwaga.* Jeżeli kompilator ma wyłączone optymalizacje (`-O0`) to bitshifty są szybsze od operacji arytmetycznych. To znaczy, że w takich dziwnych warunkach `x >> 1` jest szybsze od `x / 2`. Dotyczy to tylko dzielenia/mnożenia przez stałe.
- W drzewie binarnym, w którym indeksujemy dzieci wierzchołka x jako $2x$ oraz $2x + 1$, aby dostać dziecko o takim samym ojcu jak `y` można użyć `y^1`.
- `xor` ma ciekawe własności z tego względu, że jeżeli rzucamy w niego losowymi liczbami to wyniki też są całkiem losowe. Zatem ma zastosowanie w pewnych odmianach hashowania.
- `and` oraz `or` są idempotentne (`x | y | y = x | y`) i nie mają odwrotności. `xor` jest swoją własną odwrotnością (`x ^ x = 0`).
- Liczba `x` jest potęgą dwójki jeżeli `x & (x - 1) == 0`.
- Ponieważ `int`-y w dzisiejszych czasach używają U2 ([two's complement]), aby implementować ujemne liczby, można korzystać z takiej tożsamości: `-x == ~x+1`.
- Żeby sprawdzić czy k-ty bit liczby x jest włączony, można użyć formułki `(x >> k) & 1`. Dostaniemy 0 lub 1. Jeżeli wystarczy nam 0 i pewna niezerowa wartość, można napisać `x & (1 << k)` (otrzymamy 0 lub 2^k).

- Podłoga z logarytmu dwójkowego to jednocześnie pozycja najbardziej znaczącej jedynki. Umiemy policzyć liczbę zer bardziej znaczących niż najbardziej znacząca jedynka za pomocą `__builtin_clz`, zatem
  `floor(log2(x)) == (W-1) - __builtin_clz(x)`
  gdzie `W` to wielkość typu – 32 dla `int`-a. Funkcja licząca podłogę z logarytmu dwójkowego jest także dostępna jako `std::__lg(x)` w `<algorithm>` i działa dokładnie w ten sposób. Implementuje ona overloady dla każdego typu liczbowego.

- Ostatni bit liczby określa jej parzystość. Można go odzyskać za pomocą `x & 1` – działa to również dla liczb ujemnych (otrzymamy 0 lub 1, a operator modulo może zwrócić 0, 1 albo -1).

4 Bitmaski – liczby jako zbiory

Możemy przyjąć, że liczba o w bitach reprezentuje pewien podzbiór zbioru $\{0, 1, 2, ..., w-1\}$. Jeżeli k-ty bit (reprezentujący 2^k) jest zapalony, to k należy do zbioru reprezentowanego przez bitmaskę. Wtedy operacje bitowe można traktować jak operacje na zbiorach:

Operacja bitowa	Operacja na zbiorach
`x & y`	$X \cap Y$
`x | y`	$X \cup Y$
`x ^ y`	$X \triangle Y$
`~x`	$\overline{X}$
`x & ~y`	$X \setminus Y$
`__builtin_popcount(x)`	$\|X\|$

W charakterze mniej konwencjonalnych operacji można dorzucić `__lg(x)` jako $\max X$ oraz `__builtin_ctz(x)` jako $\min X$.

W ten sposób będziemy mogli pisać dynamiki po podzbiorach w o wiele prostszy sposób i rozwiązać np. problem komiwojażera. Poza tym bitmaski przydają się w pisaniu backtracków i brutów.

Bardzo przydatną własnością bitmask jest to, że odwiedzając liczby w kolejności rosnącej od 0 nigdy nie odwiedzimy pewnego zbioru *po zbiorze który go zawiera.* Dzięki temu pisanie dynamików na bitmaskach jest jeszcze prostsze!

5 Bitsety

`bitset` to struktura udająca tablicę `bool`-i wykorzystująca fakt, że wiele `bool`-i można upakować do jednego machine worda[2]. (np. `int`-a). (Nawiasem mówiąc, `bool` zajmuje tyle co najmniejsza adresowalna jednostka pamięci – czyli zwykle 1 bajt).

[2] W rozumieniu jako typu standardowej wielkości na danym procesorze. Uprośćmy sobie i powiedzmy, że w architekturze 32-bitowej są to 32 bity, a 64-bitowej: 64 bity.

Za pomocą `bitset`-ów zużywamy **8** razy mniej pamięci[3], i możemy wygodnie hurtowo wykonywać operacje bitowe, co pomaga w pewnych konkretnych optymalizacjach. Typ wykorzystywany przez `bitset` to `unsigned long`, i od jego wielkości zależy, ile naraz wykonujemy porównań.

Konkretniej, `bitset` jest strukturą znaną jako `std::bitset<size_t N>` w `<bitset>`. Jako parametr szablonu podajemy ilość bitów – niestety, musi być ona stała (a operacje zawsze są wykonywane na wszystkich bitach). Przykład wykorzystania:

```
bitset<16> A, B;
A[3] = A[6] = A[8] = 1;
B[3] = B[6] = B[7] = B[10] = 1;
cout << A << " " << B << endl;
cout << (A & B) << " " << (A | B) << " " << (A ^ B) << endl;
```

[Dokumentacja bitsetów].

6 *Zadanka*

6.1 Minimalna odległość Hamminga

Treść Masz dane n ciągów binarnych (czyli na alfabecie dwuznakowym) o długości k. Znajdź wśród nich parę o minimalnej odległości Hamminga (zdefiniowanej jako liczba pozycji, na której dwa ciągi różnią się).

Rozwiązanie Najprostszy brut zadziała w $O(n^2k)$. Możemy to w prosty sposób zoptymalizować wrzucając stringi do bitsetów (ręcznie zaimplementowanych bądź standardowych). Odległość Hamminga dwóch ciągów `A` i `B` jest wtedy równa `(A ^ B).count()` (analogicznie na liczbach: `__builtin_popcount(a ^ b)`). Otrzymamy w ten sposób rozwiązanie około 32 razy szybsze, bo naraz wykonujemy 32 porównania (lub tyle, ile wynosi wielkość machine worda). Złożoność tak zoptymalizowanego programu można zapisać jako $O(\frac{n^2k}{W})$, gdzie $W = 32$.

6.2 Macierze czarnorożne

Treść Masz daną macierz $n \times n$ wypełnioną zerami i jedynkami. Policz liczbę macierzy, w której narożnikach są tylko jedynki.

Rozwiązanie Możemy to rozwiązać prostym podejściem w $O(n^3)$ przechodząc wszystkie pary rzędów i zliczając, ile jest takich kolumn, że w obu rzędach jest tam jedynka. Oznaczmy tę wartość przez c – wtedy dla tej pary rzędów do wyniku dodajemy $\frac{c(c-1)}{2}$. Fazę zliczania kolumn można ulepszyć: wrzućmy rzędy macierzy do bitsetów. Liczba zapalonych bitów `and`-a pary rzędów to poszukiwana wartość c. Otrzymaliśmy rozwiązanie w $O(\frac{n^3}{W})$.

[3] W porównaniu do tablicy `bool` tej samej wielkości.

6.3 Problem wydawania reszty

Treść W klasycznym problemie wydawania reszty mamy dany zbiór nominałów i chcemy policzyć, dla jakich wartości można wybrać taki podzbiór nominałów, który sumuje się do tej wartości.

Rozwiązanie Można utrzymywać dotychczas otrzymane wartości w bitsecie. Wtedy, gdy napotykamy nowy nominał, odpowiednio przesuwamy i or-ujemy nasz bitset. *Uwaga.* Ponieważ wyrażenie `b << a[i]` może trafić na stos, to możliwe jest przekroczenie limitu stosu[4]. Aby temu zapobiec, należy zrobić tymczasowy bitset.

```
// Problem wydawania reszty dla jednorazowych nominałów a[]
bitset<M> b;
b[0] = true;
for(int i = 0; i < n; i++)
    b |= (b << a[i]);
```

W ten sposób upraszczamy implementację i znacząco zmniejszamy stałą.

6.4 Problem komiwojażera

Treść Dla odmiany dynamik na bitmaskach. Chcemy rozwiązać problem komiwojażera: dla danego zbioru miast i odległości pomiędzy nimi chcemy znaleźć najkrótszą ścieżkę która przechodzi przez każde z nich dokładnie raz.

Rozwiązanie Opisujemy dynamika o stanie $T(V, c)$, gdzie V to zbiór odwiedzonych miast, a c to miasto w którym się znajdujemy. Poszukiwana wartość to długość ścieżki odwiedzającej każde z miast w V i kończącej się w c. Przejście jest dosyć intuicyjne (niech d oznacza odległość między miastami):

$$T(V, c) = \min_{c_1 \in V \setminus \{c\}} T(V \setminus \{c\}, c_1) + d(c_1, c)$$

W prosty sposób przekłada się to na kod, jeżeli skorzystamy z bitmask:

[4] np. testując rozwiązanie lokalnie, lub na dziwnej sprawdzarce.

```
// Komiwojażer dla n wierzchołków i tablicy odległości d[][]
int T[1 << N][N];
for(int v = 1; v < (1 << n); v++)
{
    if(__builtin_popcount(v) == 1)
    {
        T[v][__lg(v)] = 0; continue;
    }
    for(int c = 0; c < n; c++)
    {
        T[v][c] = INT_MAX;
        for(int c1 = 0; c1 < n; c1++)
            if((v & (1 << c1)) and c1 != c)
                T[v][c] = min(T[v][c], T[v ^ (1 << c)][c1] + d[c1][c]);
    }
}
```

7 Triki

Bit tricks zwykle się nie pamięta, kiedy są najbardziej potrzebne, ale warto je kojarzyć.

7.1 Podzbiory bitmaski

Okazuje się, że mając pewną maskę m oraz podmaskę c, kolejno biorąc $c' = (c - 1)\ \&\ m$ będziemy okresowo dostawać bitmaski reprezentujące wszystkie podzbiory zbioru reprezentowanego przez m. Najlepiej widać to na przykładzie (kolejne komórki to kolejne c, po dekrementacji i wy`and`owaniu):

0010 0011
0010 0011
0010 0010
0010 0001
0010 0000
0000 0011
0000 0010
0000 0001
0000 0000
0010 0011
...

Formułka w C++, zaczyna od pustego i przerywa, gdy napotka pusty (`0`):

```
int c = 0;
do {
    process(c);
} while(c = (c - 1) & m);
```

7.2 Cache, cache, cache

Gdybyśmy chcieli implementować rzeczy pokroju `popcount` samodzielnie, bardzo często korzysta się z cache, które przechowują wyniki np. dla liczb 8 czy 16 bitowych. Przykładowo:

```
int byte_pop[1 << 8];

void prepare_cache()
{
    for(int i = 0; i < (1 << 8); i++)
        for(int j = 0; j < 8; j++)
            byte_pop[i] += (i >> j) & 1;
}

int popcount(int x)
{
    // 0xFF to szesnastkowo 255 = 1111 1111_2
    return byte_pop[x & 0xFF] + byte_pop[(x >> 8) & 0xFF]
         + byte_pop[(x>>16) & 0xFF] + byte_pop[(x>>24) & 0xFF];
}
```

W tym wypadku pewnie lepiej jest przechować dla liczb 16-bitowych, ale gwoli przykładu pokazuję 8-bitowe.

7.3 Upraszczanie warunków

Trudno sformalizować ten sposób. Czasami mamy dużo brzydkich `if`-ów, albo coś tego pokroju. Możemy spróbować przerzucić logikę do bitmask. Pokażę to na dwóch przykładach.

7.3.1 Samogłoska

Powiedzmy że chcemy mieć funkcję `bool is_vowel(char c)` stwierdzającą czy znak jest samogłoską. Mamy założenie, że `c` będzie wielką literą alfabetu łacińskiego. Ponieważ znaków jest mniej niż 32, możemy powrzucać do bitmaski `1` jeżeli litera na odpowiedniej pozycji jest samogłoską.

```
// 17842449 = 0b00100010000010000010001
bool is_vowel(char c)
{
    return (17842449 >> (c - 'A')) & 1;
}
```

Najlepiej po prostu jest wrzucić to do tablicy `bool`, ale jest to dobry przykład tej techniki. Kolejny przykład jest bardziej użyteczny.

7.3.2 Unikaty

Mamy trzy zmienne: `x`, `y` i `z`. Każda z nich może mieć wartości `0`, `1` albo `2`. Chcemy powiedzieć ile unikatowych wartości jest wśród zmiennych. Na przykład, dla `x = 0, y = 2, z = 0` mamy 2.

Pomysł jest taki, żeby powrzucać do bitmaski `1 << k`, jeżeli wśród liczb mamy wartość `k`, i policzyć `popcount`.

```
int count_unique(int x, int y, int z)
{
    return __builtin_popcount((1 << x) | (1 << y) | (1 << z));
}
```