# Примеры использования битового сжатия

---

## Содержание

# 1 Битовое сжатие и хранение множеств

Пусть нам нужно оперировать с подмножествами множества $\{0, 1, 2, \ldots, n-1\}$, где $n \le 64$. Закодируем множество последовательностью $n$ нулей и единиц. Если число $i$ присутствует в множестве, на $i$-й позиции будет единица, иначе ноль. Последовательность из $n$ нулей и единиц можно воспринимать, как последовательность $n$ бит, как целое число, записанное в двоичной системе счисления. Итого получаем:

```
unsigned long long x; // множество из не более чем 64 элементов
```

Теперь пусть у нас есть два множества $A$ и $B$. Выразим стандартные операции над множествами через операции над целыми числами:

<table>
<tr><td>$A \cap B$</td><td><code>A & B</code></td><td>битовый AND</td></tr>
<tr><td>$A \cup B$</td><td><code>A | B</code></td><td>битовый OR</td></tr>
<tr><td>дополнение $A$</td><td><code>~A</code></td><td>битовый NOT</td></tr>
<tr><td>$A \setminus B$</td><td><code>A & ~B</code></td><td></td></tr>
<tr><td>добавить в множество элемент $i$</td><td><code>A | (1ull « i)</code></td><td>битовый сдвиг влево</td></tr>
<tr><td>удалить из множества элемент $i$</td><td><code>A & ~(1ull « i)</code></td><td></td></tr>
</table>

Здесь `1ull` — единица типа `unsigned long long`. Во всех примерах мы делали $O(1)$ арифметических операций.

Самая сложная операция из часто используемых — посчитать размер множества, то есть число единичных бит в числе. В `C++` для этого есть специальная функция `__builtin_popcount`, которая работает за $O(\log w)$ арифметических операций, где $w$ — длина машинного слова (обычно 32 или 64). Для совсем маленьких множеств, например, до 16 или 32 элементов можно посчитать число бит за $O(1)$, используя предподсчитанную таблицу размера $2^{16}$ байт:

```
char bn[1«16];
for (i = 0; i < (1«16); i++)
    bn[i] = (i & 1) + bn[i»1];
int bit_count_16(unsigned int x) { return bn[x]; }
int bit_count_32(unsigned int x) { return bn[x»16] + bn[x&65535]; }
```

Для хранения больших множеств можно использовать массив:

```
const int n1 = n/32 + 1; // n - размер множества
unsigned int a[n1];
```

Основные функции работы с новой структурой данных:

```
int get( int i ) { return (a[i>>5] >> (i & 31)) & 1; }
void set_0( int i ) { a[i>>5] &= ~(1u << (i & 31)); }
void set_1( int i ) { a[i>>5] |= 1u << (i & 31); }
int count() {
    int sum = 0;
    for (int i = 0; i < n1; i++)
        sum += bit_count_32(a[i]);
    return sum;
}
```

В `C++/STL` подобная структура данных уже реализована и называется `bitset`. Подключение: `#include <bitset>`. Основные примеры использования:

```
bitset<100> a, b; // объявить два множества из 100 элементов
a[3] = 1; // записать 1 в 3-й бит
a[3] = 0; // записать 0 в 3-й бит
int x = a[3]; // посмотреть значение в 3-м бите, при присваивании
  оно скастится к int
printf("%d\n", (int)a[3]); // а здесь нужно кастить руками
a = a | b; a |= b; // объединение множеств
a = a & b; a &= b; // пересечение множеств
a = b >> 10; b = a << 10; // сдвиги вправо-влево
a = a & ~b; // разность множеств
int c = a.count(); // число единичных бит
a.reset(0); b.reset(1); // присвоить всем битам нужное значение
```

Обращение к $i$-му элементу (и просмотр, и присваивание) работает за $O(1)$, остальные операции для `bitset<n>` работают за $n/w$, где $w$ — длина машинного слова.

Внутри `bitset` лежит обычный массив, поэтому можно делать так:

```
bitset<100> a;
unsigned int *b = &a; // хакнули bitset!
b[0] |= 1 << 7; // a[7] = 1
```

В последующих разделах вы найдете многочисленные примеры использования `bitset`. Для краткости в примерах кода везде, где не сказано обратного, предполагается, что массивы заполнены нулями.

## 2 Умножение булевых матриц за $O(n^3/\log n)$

Стандартное умножение матриц за $O(n^3)$:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            c[i][j] += a[i][k] * b[k][j];
```

Версия над $\mathbb{F}_2$ (т.е. для булевых матриц):

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            c[i][j] ^= a[i][k] & b[k][j];
```

Заметим, что циклы `for` можно выполнять в любом удобном нам порядке. Упрощаем и ускоряем, не меняя асимптотику.

```
for (i = 0; i < n; i++)
    for (k = 0; k < n; k++)
        if (a[i][k])
            for (j = 0; j < n; j++)
                c[i][j] ^= b[k][j];
```

Используем bitset, получаем меньше кода и лучшее время работы.

```
bitset<N> a[N], b[N], c[N]; // n <= N
for (i = 0; i < n; i++)
    for (k = 0; k < n; k++)
        if (a[i][k])
            c[i] ^= b[k];
```

Новый код работает за $n^3/w$ операций, где $w$ — длина машинного слова (обычно 32 или 64). С точки зрения асимптотики мы предполагаем, что с числами не более $n$ все операции происходят за $O(1)$, т.е. на самом деле $n \leq 2^w \Rightarrow \log n \leq w$. Поэтому мы честно говорим, что получили алгоритм умножения матриц за $O(n^3/\log n)$.

# 3 Флойд за $O(n^3/\log n)$

Задача: поиск транзитивного замыкания орграфа. Для каждых $i$, $j$ хотим получить $d_{ij}$ — достижима ли из вершины $i$ вершина $j$. Пусть для удобства изначально матрица $d$ равна матрице смежности. Рассмотрим стандартный алгоритм Флойда-Уоршела.

```
int d[N][N]; // n <= N, содержит матрицу смежности
for (k = 0; k < n; k++)
    for (j = 0; j < n; j++)
        for (i = 0; i < n; i++)
            d[i][j] |= d[i][k] & d[k][j];
```

Как и в прошлой задаче избавляемся от “&”.

```
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
        if (d[i][k])
            for (j = 0; j < n; j++)
                d[i][j] |= d[k][j];
```

И для ускорения добавляем bitset.

```
bitset<N> d[N]; // n <= N
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
        if (d[i][k])
            d[i] |= d[k];
```

Получили транзитивное замыкание за $O(n^3/\log n)$.

# 4 Произведение многочленов за $O(n^2/\log n)$

Начнем с простой задачи: умножение над $\mathbb{F}_2$ . Стандартное произведение многочленов над $\mathbb{F}_2$ за $O(n^2)$:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        c[i+j] ^= a[i] & b[j];
```

Избавляемся от “&”.

```
for (i = 0; i < n; i++)
    if (a[i])
        for (j = 0; j < n; j++)
            c[i+j] ^= b[j];
```

Добавляем bitset.

```
bitset<N> a, b, c; // n <= N
for (i = 0; i < n; i++)
    if (a[i])
        c ^= b << i;
```

Упражнение: реализуйте деление с остатком многочленов над $\mathbb{F}_2$ за $O(n^2/\log n)$.

Теперь более сложная задача: умножение над $\mathbb{Z}$ многочленов с коэффициентами из $\{0, 1\}$. Для начала перевернем массив $b$.

```
for (i = 0; i < n; i++)
    b1[i] = b[n - i - 1];
```

Заметим, что $(a * b)[n-1] = \sum\limits_{i=0}^{n-1} b[n-i-1]a[i] = \sum\limits_{i=0}^{n-1} b_1[i]a[i]$.

И по аналогии $(a * b)[k] = \sum\limits_{i=0}^{n-1} b_1[i+k-n+1]a[i]$. Получаем умножение:

```
int c[2*n-1];
for (k = 0; k < n; k++)
    c[k] = ((b1 << (n-k-1)) & a).count();
for (k = n; k < 2*n-1; k++)
    c[k] = ((b1 >> (k-n+1)) & a).count();
```

# 5 Рюкзак за $O(nw/\log w)$

Задача: дан массив из $n$ чисел $a_i$, нужно найти подмножество чисел $a_i$ с суммой ровно $w$. Сперва упростим задачу и научимся отвечать на вопрос, есть ли такое множество.
Стандартная динамика за $O(nw)$:

```
is[0] = 1;
for (i = 0; i < n; i++)
    for (x = w - a[i]; x >= 0; x--)
        if (is[x])
            is[x + a[i]] = 1;
puts(is[w] ? "YES" : "NO");
```

Перепишем с использованием `bitset` за $O(nw/\log w)$:

```
is[0] = 1;
for (i = 0; i < n; i++)
    is |= is << a[i];
puts(is[w] ? "YES" : "NO");
```

Теперь научимся кроме NO/YES получать множество-ответ, пока за $O(nw)$:

```
is[0] = 1;
for (i = 0; i < n; i++)
    for (x = w-a[i]; x >= 0; x--)
        if (is[x] && !is[y = x+a[i]])
            is[y] = 1, p[y] = i;
if (!is[w])
    puts("NO");
else
    for (i = w; i > 0; i -= a[p[i]])
        printf("%d ", p[i]);
```

Теперь, чтобы получить ускорение в 32 раза, нам нужно научиться быстро перебирать все такие `x`, что `is[x]`, но не `is[x+a[i]]`.
Если `is` — `bitset`, то это ровно `is & ~(is << a[i])`.

Версия с `bitset` за $O(nw/\log w + w \log w)$:

```
bitset<W> is; // W > w
is[0] = 1;
for (i = 0; i < n; i++) {
  bitset<W> tmp = is & ~(is << a[i]); // новые единицы
  unsigned int *tmp32 = &tmp;
  for (j = w/32; j >= 0; j--)
    if (tmp32[j]) // в группе из 32 есть хотя бы одна
      for (bit = 0; bit < 32; bit++)
        if (tmp[k=(j<<5)+bit])
          is[k] = 1, p[k] = i;
}
```

К сожалению подход с использованием `bitset` не идеален. Если переписать тоже самое без `bitset`, но с битовым сжатием, код ускорится раза в 4. Основная причина в том, что создаются лишние промежуточные `bitset`-ы.

# 6 Гаусс за $O(n^3/\log n)$

Задача: посчитать определитель матрицы над $\mathbb{F}_2$ (булевой матрицы). Приведем сразу корректный код с использованием `bitset`.

```
bitset<N> a[N]; // матрица, определитель которой мы считаем
for (i = 0; i < n; i++) {
    for (j = i; j < n && !a[i][j]; j++)
        ;
    if (j == n)
        return 0;
    swap(a[i], a[j]);
    for (j = i + 1; j < n; j++)
        if (a[j][i])
            a[j] ^= a[i];
}
return 1;
```