**Введение**

Олимпиадное программирование состоит из двух частей – проектирования алгоритмов и реализации алгоритмов.

Проектирование алгоритмов. По сути своей, олимпиадное программирование – это придумывание эффективных алгоритмов решения корректно поставленных вычислительных задач. Для проектирования алгоритмов необходимы навыки в решении задач и знание математики. Зачастую решение появляется в результате сочетания хорошо известных методов и новых идей.

Важную роль в олимпиадном программировании играет математика. На самом деле четких границ между проектированием алгоритмов и математикой не существует. Эта книга не требует глубокой математической подготовки. В приложении, которое можно использовать как справочник, описаны некоторые встречающиеся в книге математические понятия и методы, например: множества, математическая логика и функции.

Реализация алгоритмов. В олимпиадном программировании решение задачи оценивается путем проверки реализованного алгоритма на ряде тестов. Поэтому придумать алгоритм недостаточно, его еще нужно корректно реализовать, для чего требуется умение программировать. Олимпиадное программирование сильно отличается от традиционной программной инженерии: программы короткие (несколько сотен строк – уже редкость), писать их нужно быстро, а сопровождение после соревнования не требуется.

В настоящее время на соревнованиях по программированию популярнее всего языки C++, Python и Java. Например, среди 3000 лучших участников Google Code Jam 2017 79% писали на C++, 16% – на Python и 8% – на Java. Многие считают C++ самым подходящим выбором для олимпиадного программиста. К его достоинствам можно отнести очень высокую эффективность и тот факт, что в стандартной библиотеке много разнообразных структур данных и алгоритмов.

Все примеры в разделах теории написаны на C++, в них часто используются структуры данных и алгоритмы из стандартной библиотеки. Код отвечает стандарту C++11, который разрешен в большинстве современных соревнований.

**Глава 1**

**Техника программирования**

**Тема 1.1 Языковые средства**

Типичная олимпиадная программа на C++ устроена по такому образцу:

#include <bits/stdc++.h>

using namespace std;

int main() {

// здесь находится решение

}

Строка #include в начале программы – специфика компилятора g++, она служит для включения всей стандартной библиотеки. Таким образом, не нужно по отдельности включать такие библиотеки, как iostream, vector и algorithm; все они автоматически становятся доступны.

В строке using объявляется, что все классы и функции из стандартной библиотеки можно использовать без указания пространства имен. Не будь using, нужно было бы писать, к примеру, std::cout, а так достаточно просто cout.

Для компиляции программы используется команда вида:

g++ -std=c++11 -02 -Wall test.cpp -o test

Она порождает двоичный файл test из исходного файла test.cpp. Флаги означают, что компилятор должен следовать стандарту C++11 (-std=c++11), оптимизировать код (-02) и выводить предупреждения о возможных проблемах (-Wall).

**Подтема 1.1.1. Ввод и вывод**

В большинстве олимпиадных задач для ввода и вывода используются стандартные потоки. В C++ стандартный поток ввода называется cin, а стандартный поток вывода – cout. Можно также использовать C-функции, например scanf и printf.

Входными данными для программы обычно являются числа и строки, разделенные пробелами и знаками новой строки. Из потока cin они читаются следующим образом:

int a, b;

string x;

cin >> a >> b >> x;

Такой код работает в предположении, что элементы потока разделены хотя бы одним пробелом или знаком новой строки. Например, приведенный выше код может прочитать как входные данные:

123 456 monkey

так и входные данные:

123 456

monkey

Поток cout используется для вывода:

int a = 123, b = 456;

string x = "monkey";

cout << a << " " << b << " " << x << "\n";

Ввод и вывод часто оказываются узкими местами программы. Чтобы повысить эффективность ввода-вывода, можно поместить в начало программы такие строки:

ios::sync\_with\_stdio(0);

cin.tie(0);

Отметим, что знак новой строки "\n" работает быстрее, чем endl, потому что endl всегда сопровождается сбросом буфера.

C-функции scanf и printf – альтернатива стандартным потокам C++. Обычно они работают немного быстрее, но и использовать их сложнее. Вот как можно прочитать из стандартного ввода два целых числа:

int a, b;

scanf("%d %d", &a, &b );

А вот как их можно напечатать:

int a = 123, b = 456;

printf("%d %d\n", a, b);

Иногда программа должна прочитать целую входную строку, быть может, содержащую пробелы. Это можно сделать с помощью функции getline:

string s;

getline(cin, s);

Если объём данных заранее неизвестен, то полезен такой цикл:

while (cin >> x) {

// код

}

Этот цикл читает из стандартного ввода элементы один за другим, пока входные данные не закончатся.

В некоторых олимпиадных системах для ввода и вывода используются файлы. В таком случае есть простое решение: писать код так, будто работаешь со стандартными потоками, но в начало программы добавить такие строки:

freopen("input.txt", "r", stdin);

freopen("output.txt", "w", stdout);

После этого программа будет читать данные из файла «input.txt», а записывать в файл «output.txt».

**Подтема 1.1.2. Работа с числами**

Целые числа. Из целых типов в олимпиадном программировании чаще всего используется int – 32-разрядный тип, принимающий значения из диапазона −231…231 − 1 (приблизительно −2 · 109 …2 · 109 ). Если типа int недостаточно, то можно взять 64-разрядный тип long long. Диапазон его значений

−263…263 − 1 (приблизительно −9 · 1018…9 · 1018). Ниже определена переменная типа long long:

long long x = 123456789123456789LL;

Суффикс LL означает, что число имеет тип long long.

Типичная ошибка при использовании типа long long возникает, когда где-то в программе встречается еще и тип int. Например, в следующем коде есть тонкая ошибка:

int a = 123456789;

long long b = a\*a;

cout << b << "\n"; // -1757895751

Хотя переменная b имеет тип long long, оба сомножителя в выражении a\*a имеют тип int, поэтому тип результата тоже int. Из-за этого значение b оказывается неверным. Проблему можно решить, изменив тип a на long long или изменив само выражение на (long long)a\*a.

Обычно олимпиадные задачи формулируются так, что типа long long достаточно. Но все же полезно знать, что компилятор g++ поддерживает также 128-разрядный тип \_\_int128\_t с диапазоном значений −2127…2127 − 1 (приблизительно −1038…1038). Однако этот тип доступен не во всех олимпиадных системах.

Арифметика по модулю. Иногда ответом является очень большое число, но достаточно вывести результат «по модулю m», т. е. остаток от деления на m (например, «7 по модулю 109 »). Идея в том, что даже когда истинный ответ очень велик, типов int и long long все равно достаточно.

Остаток x от деления на m обозначается x mod m. Например, 17 mod 5 = 2, поскольку 17 = 3 · 5 + 2. Важные свойства остатков выражаются следующими формулами:

(a + b) mod m = (a mod m + b mod m) mod m;

(a − b) mod m = (a mod m − b mod m) mod m;

(a · b) mod m = (a mod m · b mod m) mod m.

Это значит, что можно брать остаток после каждой операции, поэтому числа никогда не станут слишком большими.

Например, следующий код вычисляет n! (n факториал) по модулю m:

long long x = 1;

for (int i = 1; i <= n; i++) {

x = (x\*i)%m;

}

cout << x << "\n";

Обычно мы хотим, чтобы остаток находился в диапазоне 0…m − 1. Но в C++ и в других языках остаток от деления отрицательного числа равен нулю или отрицателен. Чтобы не возникали отрицательные остатки, можно поступить следующим образом: сначала вычислить остаток, а если он отрицателен, прибавить m:

x = x%m;

if (x < 0) x += m;

Но это стоит делать, только если в программе встречается операция вычитания, так что остаток может стать отрицательным.

Числа с плавающей точкой. В большинстве олимпиадных задач целых чисел достаточно, но иногда возникает потребность в числах с плавающей точкой. В C++ наиболее полезен 64-разрядный тип double, а в компиляторе g++ имеется расширение – 80-разрядный тип long double. Чаще всего типа double достаточно, но long double дает более высокую точность.

Требуемая точность ответа обычно указывается в формулировке задачи. Проще всего для вывода ответа использовать функцию printf и указать количество десятичных цифр в форматной строке. Например, следующий код печатает значение x с 9 цифрами после запятой:

printf("%.9f\n", x);

С использованием чисел с плавающей точкой связана одна сложность: некоторые числа невозможно точно представить в таком формате, поэтому неизбежны ошибки округления. Например, в следующем коде получается значение x, немного меньшее 1, тогда как правильное значение равно в точности 1.

double x = 0.3\*3+0.1;

printf("%.20f\n", x); // 0.99999999999999988898

Числа с плавающей точкой рискованно сравнивать с помощью оператора ==, потому что иногда равные значения оказываются различны из-за ошибок округления. Более правильно считать, что два числа равны, если разность между ними меньше ε, где ε мало. Например, в следующем коде ε = 10-9:

if (abs(a-b) < 1e-9) {

// a и b равны

}

Отметим, что хотя числа с плавающей точкой, вообще говоря, не точны, не слишком большие целые числа представляются точно. Так, тип double позволяет точно представить все целые числа, по абсолютной величине не большие 253.

Подтема 1.1.3. Сокращение кода

Имена типов. Ключевое слово typedef позволяет сопоставить типу данных короткое имя. Например, имя long long слишком длинное, поэтому можно определить для него короткий псевдоним ll:

typedef long long ll;

После этого код

long long a = 123456789;

long long b = 987654321;

cout << a\*b << "\n";

можно немного сократить:

ll a = 123456789;

ll b = 987654321;

cout << a\*b << "\n";

Ключевое слово typedef применимо и к более сложным типам. Например, ниже мы сопоставляем вектору целых чисел имя vi, а паре двух целых чисел – тип pi.

typedef vector<int> vi;

typedef pair<int, int> pi;

Макросы. Еще один способ сократить код – макросы. Макрос говорит, что определенные строки кода следует подменить до компиляции. В C++ макросы определяются с помощью ключевого слова #define. Например, мы можем определить следующие макросы:

#define F first

#define S second

#define PB push\_back

#define MP make\_pair

После чего код

v.push\_back(make\_pair(y1,x1));

v.push\_back(make\_pair(y2,x2));

int d = v[i].first+v[i].second;

можно сократить до:

v.PB(MP(y1,x1));

v.PB(MP(y2,x2));

int d = v[i].F+v[i].S;

У макроса могут быть параметры, что позволяет сокращать циклы и другие структуры. Например, можно определить такой макрос:

#define REP(i,a,b) for (int i = a; i <= b; i++)

После этого код

for (int i = 1; i <= n; i++) {

search(i);

}

можно сократить следующим образом:

REP(i,1,n) {

search(i);

}

**Тема 1.2. Рекурсивные алгоритмы**

Благодаря рекурсии часто удается элегантно реализовать алгоритм. В этом разделе мы обсудим рекурсивные алгоритмы, которые систематически перебирают потенциальные решения задачи. Сначала рассмотрим порождение подмножеств и перестановок множества, а затем обсудим более общую технику перебора с возвратом.

**Подтема 1.2.1. Порождение подмножеств**

В качестве первого применения рекурсии рассмотрим порождение всех подмножеств множества из n элементов. Например, подмножествами {1, 2, 3} являются 0, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3} и {1, 2, 3}. Следующая рекурсивная функция search генерирует подмножества. Функция манипулирует вектором

vector<int>subset;

который содержит элементы подмножества. Чтобы начать поиск, следует вызвать функцию с параметром 1.

void search(int k) {

if (k == n+1) {

// обработать подмножество

} else {

// включить k в подмножество

subset.push\_back(k);

search(k+1);

subset.pop\_back();

// не включать k в подмножество

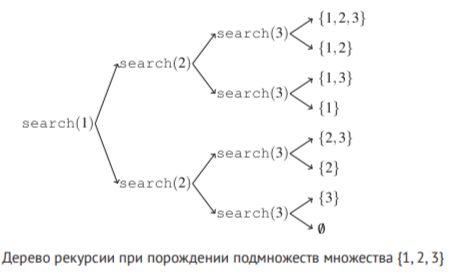
search(k+1);

}

}

При вызове с параметром k функция search решает, следует ли включать элемент k в множество или нет, но в обоих случаях вызывает себя с параметром k + 1. Если оказывается, что k = n + 1, то функция понимает, что все элементы обработаны и подмножество сгенерировано.

На рис. показано порождение подмножеств при n = 3. При каждом вызове функции выбирается либо верхняя ветвь (k включается в подмножество), либо нижняя (k не включается в подмножество).



**Пожтема 1.2.2. Порождение перестановок**

Теперь рассмотрим задачу о порождении всех перестановок множества из n элементов. Например, перестановками {1, 2, 3} будут (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2) и (3, 2, 1). И снова для решения можно применить рекурсию. Следующая функция манипулирует вектором

vector<int>permutation;

который содержит перестановку, и массивом

bool chosen[n+1];

который для каждого элемента показывает, включен он уже в перестановку или нет. Поиск начинается, когда эта функция вызывается без параметров.

void search() {

if (permutation.size() == n) {

// обработать перестановку

} else {

for (int i = 1; i <= n; i++) {

if (chosen[i]) continue;

chosen[i] = true;

permutation.push\_back(i);

search();

chosen[i] = false;

permutation.pop\_back();

}

}

}

При каждом вызове функция добавляет новый элемент в вектор permutation и запоминает в массиве chosen, что он был добавлен. Если размер permutation оказывается равен размеру множеству, то генерируется перестановка.

Отметим, что в стандартной библиотеке C++ имеется функция next\_permutation, которую также можно использовать для порождения перестановок. Этой функции передается перестановка, а она порождает следующую за ней в лексикографическом порядке. Следующая программа перебирает все перестановки множества {1, 2, …, n}:

for (int i = 1; i <= n; i++) {

permutation.push\_back(i);

}

do {

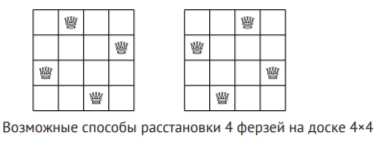
// обработать перестановку

} while (next\_permutation(permutation.begin(), permutation.end()));

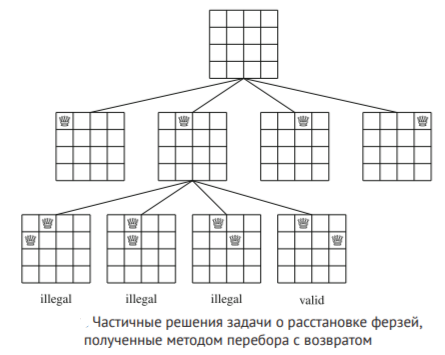
**Подтема 1.2.3. Перебор с возвратом**

Алгоритм перебора с возвратом начинает работу с пустого решения и шаг за шагом расширяет его. Рекурсивно перебираются все возможные способы построения решения.

В качестве примера рассмотрим задачу о вычислении количества способов расставить n ферзей на доске n×n, так чтобы никакие два не били друг друга. Так, на рис. показано два возможных решения для n = 4.



Задачу можно решить методом перебора с возвратом, рассматривая горизонтали одну за другой. Точнее, на каждой горизонтали можно разместить ровно одного ферзя, так чтобы он не оказался под боем ранее поставленных ферзей. Очередное решение будет найдено, когда на доску поставлены все n ферзей. Например, на рис. показано несколько частичных решений, сгенерированных алгоритмом перебора с возвратом при n = 4. Первые три расстановки на нижнем уровне недопустимы, поскольку ферзи бьют друг друга. Но четвертая расстановка допустима, и ее можно дополнить до решения, поставив на доску еще двух ферзей. Сделать это можно единственным способом.



Алгоритм можно реализовать следующим образом:

void search(int y) {

if (y == n) {

count++;

return;

}

for (int x = 0; x < n; x++) {

if (col[x] || diag1[x+y] || diag2[x-y+n-1]) continue;

col[x] = diag1[x+y] = diag2[x-y+n-1] = 1;

search(y+1);

col[x] = diag1[x+y] = diag2[x-y+n-1] = 0;

}

}

Поиск начинается вызовом search(0). Размер доски равен n, функция подсчитывает количество решений в переменной count. В коде предполагается, что горизонтали и вертикали доски пронумерованы от 0 до n − 1. Будучи вызвана с параметром y, функция search помещает ферзя на горизонталь y и вызывает себя с параметром y + 1. Когда y = n, решение найдено, поэтому значение count увеличивается на 1.

В массиве col запоминаются вертикали, уже занятые ферзями, а в массивах diag1 и diag2 запоминаются диагонали. Запрещается ставить ферзя на вертикаль или диагональ, в которой уже находится другой ферзь. На рис. показана нумерация вертикалей и диагоналей для доски 4×4.



Показанный выше алгоритм перебора с возвратом говорит, что существует 92 способа расставить 8 ферзей на доске 8×8. С ростом n поиск быстро замедляется, поскольку число решений возрастает экспоненциально. Так, на современном компьютере требуется около минуты, чтобы вычислить количество расстановок 16 ферзей на доске 16×16 – 14 772 512.

На самом деле до сих пор неизвестен эффективный способ подсчета числа расстановок ферзей для больших n. В настоящее время наибольшее n, для которого результат известен, равно 27: в этом случае существует 234 907 967 154 122 528 расстановок. Это было установлено в 2016 году группой исследователей, использовавших для решения кластер компьютеров [25].

**Тема 1.3. Поразрядные операции**

В программировании n-разрядное целое число хранится в виде двоичного числа, содержащего n бит. Например, тип int в C++ 32-разрядный, т. е. любое число типа int содержит 32 бита. Так, двоичное представление числа 43 типа int имеет вид

00000000000000000000000000101011.

Биты в этом представлении нумеруются справа налево. Преобразование двоичного представления bk … b2 b1 b0 в десятичное число производится по формуле

bk 2k + … + b2 22 + b1 21 + b0 20 .

Например: 1 · 25 + 1 · 23 + 1 · 21 + 1 · 20 = 43. Двоичное представление числа может быть со знаком и без знака. Обычно используется представление со знаком, позволяющее представить положительные и отрицательные числа. n-разрядная переменная со знаком может содержать любое целое число в диапазоне от −2n-1 до 2n-1 − 1. Например, тип int в C++ знаковый, поэтому переменная типа int может содержать любое целое число от −231 до 231 − 1.

Первый разряд в представлении со знаком содержит знак числа (0 для неотрицательных чисел, 1 – для отрицательных), а остальные n − 1 разрядов – абсолютную величину числа. Используется дополнительный код, т. е. для получения противоположного числа нужно сначала инвертировать все его биты, а затем прибавить к результату единицу. Например, двоичное представление числа –43 типа int имеет вид

11111111111111111111111111010101.

Представление без знака позволяет представить только неотрицательные числа, но верхняя граница диапазона больше. n-разрядная переменная без знака может содержать любое целое число от 0 до 2n − 1. Например, в C++ переменная типа unsigned int может содержать любое целое число от 0 до 232 − 1. Между обоими представлениям существует связь: число со знаком –x равно числу без знака 2n − x. К примеру, следующая программа показывает, что число со знаком x = −43 равно числу без знака y = 232 − 43:

int x = -43;

unsigned int y = x;

cout << x << "\n"; // -43

cout << y << "\n"; // 4294967253

Если число больше верхней границы допустимого диапазона, то возникает переполнение. В представлении со знаком число, следующее за 2n-1 – 1, равно −2n-1, а в представлении без знака за 2n-1 следует 0. Рассмотрим следующий код:

int x = 2147483647;

cout << x << "\n"; // 2147483647

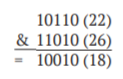
x++;

cout << x << "\n"; // -2147483648

Первоначально x принимает значение 231 − 1. Это наибольшее значение, которое можно сохранить в переменной типа int, поэтому следующее за 231 − 1 значение равно −231.

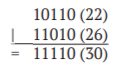
**Подтема 1.3.1. Поразрядные операции**

Операция И. Результатом операции И x & y является число, двоичное представление которого содержит единицы в тех позициях, на которых в представлениях x и y находятся единицы. Например, 22 & 26 = 18, поскольку

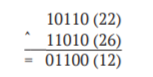


С помощью операции И можно проверить, является ли число x четным, т. к. x & 1 = 0, если x четно, и x & 1 = 1, если x нечетно. Вообще, x нацело делится на 2k , если x & (2k − 1) = 0.

Операция ИЛИ. Результатом операции ИЛИ x | y является число, двоичное представление которого содержит единицы в тех позициях, на которых хотя бы в одном из представлений x и y находятся единицы. Например, 22 | 26 = 30, поскольку



Операция ИСКЛЮЧАЮЩЕЕ ИЛИ. Результатом операции ИСКЛЮЧАЮЩЕЕ ИЛИ x ^ y является число, двоичное представление которого содержит единицы в тех позициях, на которых ровно в одном из представлений x и y находятся единицы. Например, 22 ^ 26 = 12, поскольку



Операция НЕ. Результатом операции НЕ ~x является число, в двоичном представлении которого все биты x инвертированы. Справедлива формула ~x = −x – 1, например ~29 = −30. Результат операции НЕ на битовом уровне зависит от длины двоичного представления, поскольку инвертируются все биты. Например, в случае 32-разрядных чисел типа int имеем:

x = 29 00000000000000000000000000011101

~x = −30 11111111111111111111111111100010

Поразрядный сдвиг. Операция поразрядного сдвига влево x << k дописывает в конец числа k нулей, а операция поразрядного сдвига вправо x >> k удаляет k последних бит. Например, 14 << 2 = 56, поскольку двоичные представления 14 и 56 равны соответственно 1110 и 111000. Аналогично 49 >> 3 = 6, потому что 49 и 6 в двоичном виде равны соответственно 110001 и 110. Отметим, что операция x << k соответствует умножению x на 2k, а x >> k – делению x на 2k с последующим округлением с недостатком до целого.

Битовые маски. Битовой маской называется число вида 1 << k, содержащее в позиции k единицу, а во всех остальных позициях – нули. Такую маску можно использовать для выделения одиночных битов. В частности, k-й бит числа равен единице тогда и только тогда, когда x & (1 << k) не равно нулю. В следующем фрагменте печатается двоичное представление

числа x типа int:

for (int k = 31; k >= 0; k--) {

if (x&(1<<k)) cout << "1";

else cout << "0";

}

Аналогичным образом можно модифицировать отдельные биты числа. Выражение x | (1 << k) устанавливает k-й бит x в единицу, выражение x & ~(1 << k) сбрасывает k-й бит x в нуль, а выражение x ˆ (1 << k) инвертирует k-й бит x. Далее выражение x & (x − 1) сбрасывает последний единичный бит x в нуль, а выражение x & −x сбрасывает в нуль все единичные биты, кроме последнего. Выражение x | (x − 1) инвертирует все биты после последнего единичного. Наконец, положительное число x является степенью

двойки тогда и только тогда, когда x & (x − 1) = 0.

При работе с битовыми масками нужно помнить, что 1<<k всегда имеет тип int. Самый простой способ создать битовую маску типа long long – написать 1LL<<k.

Дополнительные функции. Компилятор g++ предлагает также следующие функции для подсчета битов:

• \_\_builtin\_clz(x): количество нулей в начале двоичного представления;

• \_\_builtin\_ctz(x): количество нулей в конце двоичного представления;

• \_\_builtin\_popcount(x): количество единиц в двоичном представлении;

• \_\_builtin\_parity(x): четность количества единиц в двоичном представлении.

Эти функции используются следующим образом:

int x = 5328; // 00000000000000000001010011010000

cout << \_\_builtin\_clz(x) << "\n"; // 19

cout << \_\_builtin\_ctz(x) << "\n”; // 4

cout << \_\_builtin\_popcount(x) << "\n"; // 5

cout << \_\_builtin\_parity(x) << "\n"; // 1

Отметим, что данные функции поддерживают только числа типа int, но есть также их варианты для типа long long, их имена заканчиваются суффиксом ll.

**Подтема 1.3.2. Представление множеств**

Любое подмножество множества {0, 1, 2, …, n − 1} можно представить n-разрядным целым числом, в котором единичные биты соответствуют элементам, принадлежащим подмножеству. Такой способ представления эффективен, поскольку для каждого элемента требуется всего один бит памяти, а теоретико-множественные операции можно реализовать с помощью поразрядных операций.

Например, поскольку тип int 32-разрядный, числом типа int можно

представить любое подмножество множества {0, 1, 2, …, 31}. Двоичное

представление множества {1, 3, 4, 8} имеет вид

00000000000000000000000100011010,

что соответствует числу 28 + 24 + 23 + 21 = 282.

В показанном ниже коде объявлена переменная x типа int, которая может содержать подмножество множества {0, 1, 2, …, 31}. Затем в это подмножество добавляются элементы 1, 3, 4, 8 и печатается его размер.

int x = 0;

x |= (1<<1);

x |= (1<<3);

x |= (1<<4);

x |= (1<<8);

cout << \_\_builtin\_popcount(x) << "\n"; // 4

Далее печатаются все элементы, принадлежащие множеству:

for (int i = 0; i < 32; i++) {

if (x&(1<<i)) cout << i << " ";

}

// выводится: 1 3 4 8

Операции над множествами. В табл. показано, как реализовать теоретико-множественные операции с помощью поразрядных. Так, следующий код сначала конструирует множества x = {1, 3, 4, 8} и y = {3, 6, 8, 9},

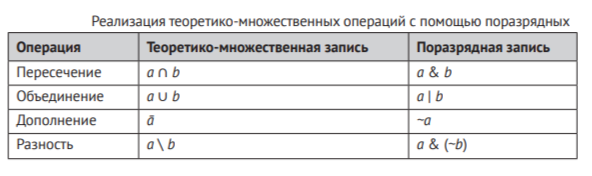
а затем множество z = x ∪ y = {1, 3, 4, 6, 8, 9}:

int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);

int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);

int z = x|y;

cout << \_\_builtin\_popcount(z) << "\n"; // 6



Следующий код перебирает подмножества множества {0, 1, …, n − 1}:

for (int b = 0; b < (1<<n); b++) {

// обработать подмножество b

}

А этот код перебирает только подмножества, содержащие ровно k элементов:

for (int b = 0; b < (1<<n); b++) {

if (\_\_builtin\_popcount(b) == k) {

// обработать подмножество b

}

}

Наконец, следующий код перебирает все подмножества множества x:

int b = 0;

do {

// обработать подмножество b

} while (b=(b-x)&x);

Битовые множества в C++. В стандартной библиотеке C++ имеется структура bitset, соответствующая массиву, все элементы которого равны 0 или 1. Например, следующий код создает битовое множество из 10 элементов:

bitset<10> s;

s[1] = 1;

s[3] = 1;

s[4] = 1;

s[7] = 1;

cout << s[4] << "\n"; // 1

cout << s[5] << "\n"; // 0

Функция count возвращает количество единичных битов в битовом множестве:

cout << s.count() << "\n"; // 4

К битовым множествам также применимы поразрядные операции:

bitset<10> a, b;

// ...

bitset<10> c = a & b;

bitset<10> d = a|b;

bitset<10> e