Programming Techniques

2

This chapter presents some of the features of the C++ programming language that are useful in competitive programming, and gives examples of how to use recursion and bit operations in programming.

Section 2.1 discusses a selection of topics related to C++, including input and output methods, working with numbers, and how to shorten code.

Section 2.2 focuses on recursive algorithms. First we will learn an elegant way to generate all subsets and permutations of a set using recursion. After this, we will use backtracking to count the number of ways to place n non-attacking queens on an $n \times n$ chessboard.

Section 2.3 discusses the basics of bit operations and shows how to use them to represent subsets of sets.

2.1 Language Features

A typical C++ code template for competitive programming looks like this:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    // solution comes here
}
```

The `#include` line at the beginning of the code is a feature of the `g++` compiler that allows us to include the entire standard library. Thus, it is not needed to separately

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate
Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_2

include libraries such as `iostream`, `vector`, and `algorithm`, but rather they are available automatically.

The `using` line declares that the classes and functions of the standard library can be used directly in the code. Without the `using` line we would have to write, for example, `std::cout`, but now it suffices to write `cout`.

The code can be compiled using the following command:

```
g++ -std=c++11 -O2 -Wall test.cpp -o test
```

This command produces a binary file `test` from the source code `test.cpp`. The compiler follows the C++11 standard (`-std=c++11`), optimizes the code (`-O2`), and shows warnings about possible errors (`-Wall`).

2.1.1 Input and Output

In most contests, standard streams are used for reading input and writing output. In C++, the standard streams are `cin` for input and `cout` for output. Also C functions, such as `scanf` and `printf`, can be used.

The input for the program usually consists of numbers and strings separated with spaces and newlines. They can be read from the `cin` stream as follows:

```
int a, b;
string x;
cin >> a >> b >> x;
```

This kind of code always works, assuming that there is at least one space or newline between each element in the input. For example, the above code can read both the following inputs:

```
123 456 monkey
```

```
123   456
monkey
```

The `cout` stream is used for output as follows:

```
int a = 123, b = 456;
string x = "monkey";
cout << a << " " << b << " " << x << "\n";
```

Input and output is sometimes a bottleneck in the program. The following lines at the beginning of the code make input and output more efficient:

```
ios::sync_with_stdio(0);
cin.tie(0);
```

Note that the newline `"\n"` works faster than `endl`, because `endl` always causes a flush operation.

The C functions `scanf` and `printf` are an alternative to the C++ standard streams. They are usually slightly faster, but also more difficult to use. The following code reads two integers from the input:

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

The following code prints two integers:

```
int a = 123, b = 456;
printf("%d %d\n", a, b);
```

Sometimes the program should read a whole input line, possibly containing spaces. This can be accomplished by using the `getline` function:

```
string s;
getline(cin, s);
```

If the amount of data is unknown, the following loop is useful:

```
while (cin >> x) {
    // code
}
```

This loop reads elements from the input one after another, until there is no more data available in the input.

In some contest systems, files are used for input and output. An easy solution for this is to write the code as usual using standard streams, but add the following lines to the beginning of the code:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

After this, the program reads the input from the file “input.txt” and writes the output to the file “output.txt”.

2.1.2 Working with Numbers

Integers The most used integer type in competitive programming is `int`, which is a 32-bit type[1] with a value range of $-2^{31} \ldots 2^{31} - 1$ (about $-2 \cdot 10^9 \ldots 2 \cdot 10^9$). If the type `int` is not enough, the 64-bit type `long long` can be used. It has a value range of $-2^{63} \ldots 2^{63} - 1$ (about $-9 \cdot 10^{18} \ldots 9 \cdot 10^{18}$).

The following code defines a `long long` variable:

```
long long x = 12345678912345678LL;
```

The suffix `LL` means that the type of the number is `long long`.

A common mistake when using the type `long long` is that the type `int` is still used somewhere in the code. For example, the following code contains a subtle error:

```
int a = 123456789;
long long b = a*a;
cout << b << "\n"; // -1757895751
```

Even though the variable `b` is of type `long long`, both numbers in the expression `a*a` are of type `int`, and the result is also of type `int`. Because of this, the variable `b` will have a wrong result. The problem can be solved by changing the type of `a` to `long long` or by changing the expression to `(long long)a*a`.

Usually contest problems are set so that the type `long long` is enough. Still, it is good to know that the `g++` compiler also provides a 128-bit type `__int128_t` with a value range of $-2^{127} \ldots 2^{127} - 1$ (about $-10^{38} \ldots 10^{38}$). However, this type is not available in all contest systems.

Modular Arithmetic Sometimes, the answer to a problem is a very large number, but it is enough to output it "modulo m", i.e., the remainder when the answer is divided by m (e.g., "modulo $10^9 + 7$"). The idea is that even if the actual answer is very large, it suffices to use the types `int` and `long long`.

We denote by $x \bmod m$ the remainder when x is divided by m. For example, $17 \bmod 5 = 2$, because $17 = 3 \cdot 5 + 2$. An important property of remainders is that the following formulas hold:

$$\begin{aligned}(a + b) \bmod m &= (a \bmod m + b \bmod m) \bmod m \\ (a - b) \bmod m &= (a \bmod m - b \bmod m) \bmod m \\ (a \cdot b) \bmod m &= (a \bmod m \cdot b \bmod m) \bmod m\end{aligned}$$

Thus, we can take the remainder after every operation, and the numbers will never become too large.

[1] In fact, the C++ standard does not exactly specify the sizes of the number types, and the bounds depend on the compiler and platform. The sizes given in this section are those you will very likely see when using modern systems.

For example, the following code calculates $n!$, the factorial of n, modulo m:

```
long long x = 1;
for (int i = 1; i <= n; i++) {
    x = (x*i)%m;
}
cout << x << "\n";
```

Usually we want the remainder to always be between $0 \ldots m-1$. However, in C++ and other languages, the remainder of a negative number is either zero or negative. An easy way to make sure there are no negative remainders is to first calculate the remainder as usual and then add m if the result is negative:

```
x = x%m;
if (x < 0) x += m;
```

However, this is only needed when there are subtractions in the code, and the remainder may become negative.

Floating Point Numbers In most competitive programming problems, it suffices to use integers, but sometimes floating point numbers are needed. The most useful floating point types in C++ are the 64-bit `double` and, as an extension in the `g++` compiler, the 80-bit `long double`. In most cases, `double` is enough, but `long double` is more accurate.

The required precision of the answer is usually given in the problem statement. An easy way to output the answer is to use the `printf` function and give the number of decimal places in the formatting string. For example, the following code prints the value of x with 9 decimal places:

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

A difficulty when using floating point numbers is that some numbers cannot be represented accurately as floating point numbers, and there will be rounding errors. For example, in the following code, the value of x is slightly smaller than 1, while the correct value would be 1.

```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

It is risky to compare floating point numbers with the == operator, because it is possible that the values should be equal but they are not because of precision errors. A better way to compare floating point numbers is to assume that two numbers are equal if the difference between them is less than ε, where ε is a small number. For example, in the following code $\varepsilon = 10^{-9}$:

```
if (abs(a-b) < 1e-9) {
    // a and b are equal
}
```

Note that while floating point numbers are inaccurate, integers up to a certain limit can still be represented accurately. For example, using `double`, it is possible to accurately represent all integers whose absolute value is at most 2^{53}.

2.1.3 Shortening Code

Type Names The command `typedef` can be used to give a short name to a data type. For example, the name `long long` is long, so we can define a short name `ll` as follows:

```
typedef long long ll;
```

After this, the code

```
long long a = 123456789;
long long b = 987654321;
cout << a*b << "\n";
```

can be shortened as follows:

```
ll a = 123456789;
ll b = 987654321;
cout << a*b << "\n";
```

The command `typedef` can also be used with more complex types. For example, the following code gives the name `vi` for a vector of integers, and the name `pi` for a pair that contains two integers.

```
typedef vector<int> vi;
typedef pair<int,int> pi;
```

Macros Another way to shorten code is to define *macros*. A macro specifies that certain strings in the code will be changed before the compilation. In C++, macros are defined using the `#define` keyword.

For example, we can define the following macros:

```
#define F first
#define S second
#define PB push_back
#define MP make_pair
```

After this, the code

```
v.push_back(make_pair(y1,x1));
v.push_back(make_pair(y2,x2));
int d = v[i].first+v[i].second;
```

can be shortened as follows:

```
v.PB(MP(y1,x1));
v.PB(MP(y2,x2));
int d = v[i].F+v[i].S;
```

A macro can also have parameters, which makes it possible to shorten loops and other structures. For example, we can define the following macro:

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

After this, the code

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

can be shortened as follows:

```
REP(i,1,n) {
    search(i);
}
```

2.2 Recursive Algorithms

Recursion often provides an elegant way to implement an algorithm. In this section, we discuss recursive algorithms that systematically go through candidate solutions to a problem. First, we focus on generating subsets and permutations and then discuss the more general backtracking technique.

2.2.1 Generating Subsets

Our first application of recursion is generating all subsets of a set of n elements. For example, the subsets of $\{1, 2, 3\}$ are $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$, and $\{1, 2, 3\}$. The following recursive function `search` can be used to generate the subsets. The function maintains a vector

```
vector<int> subset;
```

that will contain the elements of each subset. The search begins when the function is called with parameter 1.

```
void search(int k) {
    if (k == n+1) {
        // process subset
    } else {
        // include k in the subset
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
        // don't include k in the subset
        search(k+1);
    }
}
```

When the function `search` is called with parameter k, it decides whether to include the element k in the subset or not, and in both cases, then calls itself with parameter $k+1$. Then, if $k = n+1$, the function notices that all elements have been processed and a subset has been generated.

Figure 2.1 illustrates the generation of subsets when $n = 3$. At each function call, either the upper branch (k is included in the subset) or the lower branch (k is not included in the subset) is chosen.

2.2.2 Generating Permutations

Next we consider the problem of generating all permutations of a set of n elements. For example, the permutations of {1, 2, 3} are (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), and (3, 2, 1). Again, we can use recursion to perform the search. The following function `search` maintains a vector

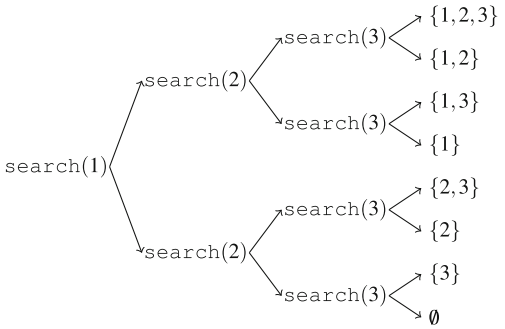

Fig. 2.1 The recursion tree when generating the subsets of the set {1, 2, 3}

```
vector<int> permutation;
```

that will contain each permutation, and an array

```
bool chosen[n+1];
```

which indicates for each element if it has been included in the permutation. The search begins when the function is called without parameters.

```
void search() {
    if (permutation.size() == n) {
        // process permutation
    } 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();
        }
    }
}
```

Each function call appends a new element to `permutation` and records that it has been included in `chosen`. If the size of `permutation` equals the size of the set, a permutation has been generated.

Note that the C++ standard library also has the function `next_permutation` that can be used to generate permutations. The function is given a permutation, and it produces the next permutation in lexicographic order. The following code goes through the permutations of $\{1, 2, \ldots, n\}$:

```
for (int i = 1; i <= n; i++) {
    permutation.push_back(i);
}
do {
    // process permutation
} while (next_permutation(permutation.begin(),
                          permutation.end()));
```

2.2.3 Backtracking

A *backtracking* algorithm begins with an empty solution and extends the solution step by step. The search recursively goes through all different ways how a solution can be constructed.

As an example, consider the problem of calculating the number of ways n queens can be placed on an $n \times n$ chessboard so that no two queens attack each other. For example, Fig. 2.2 shows the two possible solutions for $n = 4$.

The problem can be solved using backtracking by placing queens on the board row by row. More precisely, exactly one queen will be placed on each row so that no queen attacks any of the queens placed before. A solution has been found when all n queens have been placed on the board.

For example, Fig. 2.3 shows some partial solutions generated by the backtracking algorithm when $n = 4$. At the bottom level, the three first configurations are illegal, because the queens attack each other. However, the fourth configuration is valid, and it can be extended to a complete solution by placing two more queens on the board. There is only one way to place the two remaining queens.

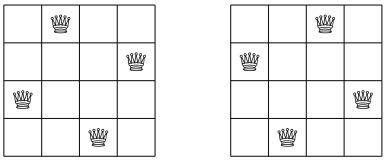

Fig. 2.2 The possible ways to place 4 queens on a 4×4 chessboard

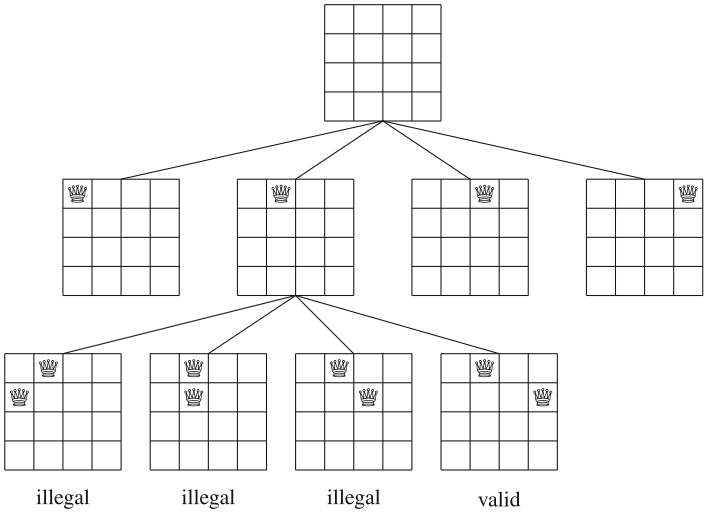

Fig. 2.3 Partial solutions to the queen problem using backtracking

The algorithm can be implemented as follows:

```
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;
    }
}
```

The search begins by calling `search(0)`. The size of the board is n, and the code calculates the number of solutions to `count`. The code assumes that the rows and columns of the board are numbered from 0 to $n-1$. When `search` is called with parameter y, it places a queen on row y and then calls itself with parameter $y+1$. Then, if $y = n$, a solution has been found, and the value of `count` is increased by one.

The array `col` keeps track of the columns that contain a queen, and the arrays `diag1` and `diag2` keep track of the diagonals. It is not allowed to add another queen to a column or diagonal that already contains a queen. For example, Fig. 2.4 shows the numbering of columns and diagonals of the 4×4 board.

The above backtracking algorithm tells us that there are 92 ways to place 8 queens on the 8×8 board. When n increases, the search quickly becomes slow, because the number of solutions grows exponentially. For example, it takes already about a minute on a modern computer to calculate that there are 14772512 ways to place 16 queens on the 16×16 board.

In fact, nobody knows an *efficient* way to count the number of queen combinations for larger values of n. Currently, the largest value of n for which the result is known is 27: there are 234907967154122528 combinations in this case. This was discovered in 2016 by a group of researchers who used a cluster of computers to calculate the result [25].

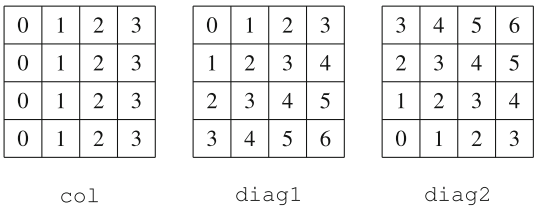

Fig. 2.4 Numbering of the arrays when counting the combinations on the 4×4 board

2.3 Bit Manipulation

In programming, an n-bit integer is internally stored as a binary number that consists of n bits. For example, the C++ type `int` is a 32-bit type, which means that every `int` number consists of 32 bits. For example, the bit representation of the `int` number 43 is

$$00000000000000000000000000101011.$$

The bits in the representation are indexed from right to left. To convert a bit representation $b_k \ldots b_2 b_1 b_0$ into a number, the formula

$$b_k 2^k + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

can be used. For example,

$$1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 43.$$

The bit representation of a number is either *signed* or *unsigned*. Usually a signed representation is used, which means that both negative and positive numbers can be represented. A signed variable of n bits can contain any integer between -2^{n-1} and $2^{n-1} - 1$. For example, the `int` type in C++ is a signed type, so an `int` variable can contain any integer between -2^{31} and $2^{31} - 1$.

The first bit in a signed representation is the sign of the number (0 for nonnegative numbers and 1 for negative numbers), and the remaining $n - 1$ bits contain the magnitude of the number. *Two's complement* is used, which means that the opposite number of a number is calculated by first inverting all the bits in the number and then increasing the number by one. For example, the bit representation of the `int` number -43 is

$$11111111111111111111111111010101.$$

In an unsigned representation, only nonnegative numbers can be used, but the upper bound for the values is larger. An unsigned variable of n bits can contain any integer between 0 and $2^n - 1$. For example, in C++, an `unsigned int` variable can contain any integer between 0 and $2^{32} - 1$.

There is a connection between the representations: a signed number $-x$ equals an unsigned number $2^n - x$. For example, the following code shows that the signed number $x = -43$ equals the unsigned number $y = 2^{32} - 43$:

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

If a number is larger than the upper bound of the bit representation, the number will overflow. In a signed representation, the next number after $2^{n-1} - 1$ is -2^{n-1},

and in an unsigned representation, the next number after $2^n - 1$ is 0. For example, consider the following code:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Initially, the value of x is $2^{31} - 1$. This is the largest value that can be stored in an `int` variable, so the next number after $2^{31} - 1$ is -2^{31}.

2.3.1 Bit Operations

And Operation The *and* operation x & y produces a number that has one bits in positions where both x and y have one bits. For example, 22 & 26 = 18, because

$$\begin{array}{r} 10110\ (22) \\ \& \ 11010\ (26) \\ \hline = 10010\ (18)\ . \end{array}$$

Using the and operation, we can check if a number x is even because x & $1 = 0$ if x is even, and x & $1 = 1$ if x is odd. More generally, x is divisible by 2^k exactly when x & $(2^k - 1) = 0$.

Or Operation The *or* operation $x \mid y$ produces a number that has one bits in positions where at least one of x and y have one bits. For example, $22 \mid 26 = 30$, because

$$\begin{array}{r} 10110\ (22) \\ \mid 11010\ (26) \\ \hline = 11110\ (30)\ . \end{array}$$

Xor Operation The *xor* operation x ^ y produces a number that has one bits in positions where exactly one of x and y have one bits. For example, 22 ^ 26 = 12, because

$$\begin{array}{r} 10110\ (22) \\ \hat{} \ 11010\ (26) \\ \hline = 01100\ (12)\ . \end{array}$$

Not Operation The *not* operation ~x produces a number where all the bits of x have been inverted. The formula ~$x = -x - 1$ holds, for example, ~29 = −30. The result of the not operation at the bit level depends on the length of the bit representation,

because the operation inverts all bits. For example, if the numbers are 32-bit `int` numbers, the result is as follows:

$$\begin{aligned} x &= \quad 29 \; 00000000000000000000000000011101 \\ \tilde{}x &= -30 \; 11111111111111111111111111100010 \end{aligned}$$

Bit Shifts The *left bit shift* $x << k$ appends k zero bits to the number, and the *right bit shift* $x >> k$ removes the k last bits from the number. For example, $14 << 2 = 56$, because 14 and 56 correspond to 1110 and 111000. Similarly, $49 >> 3 = 6$, because 49 and 6 correspond to 110001 and 110. Note that $x << k$ corresponds to multiplying x by 2^k, and $x >> k$ corresponds to dividing x by 2^k rounded down to an integer.

Bit Masks A *bit mask* of the form $1 << k$ has a one bit in position k, and all other bits are zero, so we can use such masks to access single bits of numbers. In particular, the kth bit of a number is one exactly when $x \,\&\, (1 << k)$ is not zero. The following code prints the bit representation of an `int` number x:

```
for (int k = 31; k >= 0; k--) {
    if (x&(1<<k)) cout << "1";
    else cout << "0";
}
```

It is also possible to modify single bits of numbers using similar ideas. The formula $x \mid (1 << k)$ sets the kth bit of x to one, the formula $x \,\&\, \tilde{}(1 << k)$ sets the kth bit of x to zero, and the formula $x \,\hat{}\, (1 << k)$ inverts the kth bit of x. Then, the formula $x \,\&\, (x-1)$ sets the last one bit of x to zero, and the formula $x \,\&\, -x$ sets all the one bits to zero, except for the last one bit. The formula $x \mid (x-1)$ inverts all the bits after the last one bit. Finally, a positive number x is a power of two exactly when $x \,\&\, (x-1) = 0$.

One pitfall when using bit masks is that `1<<k` is always an `int` bit mask. An easy way to create a `long long` bit mask is `1LL<<k`.

Additional Functions The g++ compiler also provides the following functions for counting bits:

- `__builtin_clz`(x): the number of zeros at the beginning of the bit representation
- `__builtin_ctz`(x): the number of zeros at the end of the bit representation
- `__builtin_popcount`(x): the number of ones in the bit representation
- `__builtin_parity`(x): the parity (even or odd) of the number of ones in the bit representation

The functions can be used as follows:

```
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
```

Note that the above functions only support `int` numbers, but there are also `long long` versions of the functions available with the suffix `ll`.

2.3.2 Representing Sets

Every subset of a set $\{0, 1, 2, \ldots, n-1\}$ can be represented as an n bit integer whose one bits indicate which elements belong to the subset. This is an efficient way to represent sets, because every element requires only one bit of memory, and set operations can be implemented as bit operations.

For example, since `int` is a 32-bit type, an `int` number can represent any subset of the set $\{0, 1, 2, \ldots, 31\}$. The bit representation of the set $\{1, 3, 4, 8\}$ is

$$00000000000000000000000100011010,$$

which corresponds to the number $2^8 + 2^4 + 2^3 + 2^1 = 282$.

The following code declares an `int` variable x that can contain a subset of $\{0, 1, 2, \ldots, 31\}$. After this, the code adds the elements 1, 3, 4, and 8 to the set and prints the size of the set.

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

Then, the following code prints all elements that belong to the set:

```
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
// output: 1 3 4 8
```

Set Operations Table 2.1 shows how set operations can be implemented as bit operations. For example, the following code first constructs the sets $x = \{1, 3, 4, 8\}$ and $y = \{3, 6, 8, 9\}$ and then constructs the set $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$:

Table 2.1 Implementing set operations as bit operations

Operation	Set syntax	Bit syntax
Intersection	$a \cap b$	a & b
Union	$a \cup b$	a | b
Complement	$\bar{a}$	~a
Difference	$a \setminus b$	a & (~b)

```
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
```

The following code goes through the subsets of $\{0, 1, \ldots, n-1\}$:

```
for (int b = 0; b < (1<<n); b++) {
    // process subset b
}
```

Then, the following code goes through the subsets with exactly k elements:

```
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // process subset b
    }
}
```

Finally, the following code goes through the subsets of a set x:

```
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);
```

C++ Bitsets The C++ standard library also provides the `bitset` structure, which corresponds to an array whose each value is either 0 or 1. For example, the following code creates a bitset of 10 elements:

```
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
```

The function `count` returns the number of one bits in the bitset:

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

Also bit operations can be directly used to manipulate bitsets:

```
bitset<10> a, b;
// ...
bitset<10> c = a&b;
bitset<10> d = a|b;
bitset<10> e = a^b;
```