Chapter 10

Bit manipulation

All data in computer programs is internally stored as bits, i.e., as numbers 0 and 1. This chapter discusses the bit representation of integers, and shows examples of how to use bit operations. It turns out that there are many uses for bit manipulation in algorithm programming.

Bit representation

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.

Here is the bit representation of the `int` number 43:

00000000000000000000000000101011

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

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

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}.

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

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

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 | y produces a number that has one bits in positions where at least one of x and y have one bits. For example, 22 | 26 = 30, because

$$\begin{array}{rll} & 10110 & (22) \\ | & 11010 & (26) \\ \hline = & 11110 & (30) \end{array}$$

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

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

Not operation

The **not** operation $\sim x$ produces a number where all the bits of x have been inverted. The formula $\sim x = -x - 1$ holds, for example, $\sim 29 = -30$.

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:

$$\begin{array}{rrrl} x & = & 29 & 00000000000000000000000000011101 \\ \sim x & = & -30 & 11111111111111111111111111100010 \end{array}$$

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.

Applications

A number of the form $1 << k$ has a one bit in position k and all other bits are zero, so we can use such numbers 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 i = 31; i >= 0; i--) {
    if (x&(1<<i)) cout << "1";
    else cout << "0";
}
```

It is also possible to modify single bits of numbers using similar ideas. For example, the formula x | $(1 << k)$ sets the kth bit of x to one, the formula x & $\sim(1 << k)$ sets the kth bit of x to zero, and the formula x ^ $(1 << k)$ inverts the kth bit of x.

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 | $(x-1)$ inverts all the bits after the last one bit. Also note that a positive number x is a power of two exactly when x & $(x-1) = 0$.

Additional functions

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

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

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
```

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

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 $2^8+2^4+2^3+2^1=282$.

Set implementation

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

Set operations can be implemented as follows as bit operations:

	set syntax	bit syntax
intersection	$a \cap b$	a & b
union	$a \cup b$	a | b
complement	$\bar{a}$	$\sim a$
difference	$a \setminus b$	a & $(\sim b)$

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 $z = x \cup y = \{1,3,4,6,8,9\}$:

```
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
```

Iterating through subsets

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

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

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
    }
}
```

The following code goes through the subsets of a set x:

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

Bit optimizations

Many algorithms can be optimized using bit operations. Such optimizations do not change the time complexity of the algorithm, but they may have a large impact on the actual running time of the code. In this section we discuss examples of such situations.

Hamming distances

The **Hamming distance** $\texttt{hamming}(a,b)$ between two strings a and b of equal length is the number of positions where the strings differ. For example,

$$\texttt{hamming}(01101, 11001) = 2.$$

Consider the following problem: Given a list of n bit strings, each of length k, calculate the minimum Hamming distance between two strings in the list. For example, the answer for $[00111, 01101, 11110]$ is 2, because

- $\texttt{hamming}(00111, 01101) = 2$,
- $\texttt{hamming}(00111, 11110) = 3$, and
- $\texttt{hamming}(01101, 11110) = 3$.

A straightforward way to solve the problem is to go through all pairs of strings and calculate their Hamming distances, which yields an $O(n^2k)$ time algorithm. The following function can be used to calculate distances:

```
int hamming(string a, string b) {
    int d = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
    }
    return d;
}
```

However, if k is small, we can optimize the code by storing the bit strings as integers and calculating the Hamming distances using bit operations. In particular, if $k \leq 32$, we can just store the strings as `int` values and use the following function to calculate distances:

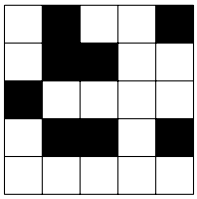
```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

In the above function, the xor operation constructs a bit string that has one bits in positions where a and b differ. Then, the number of bits is calculated using the `__builtin_popcount` function.

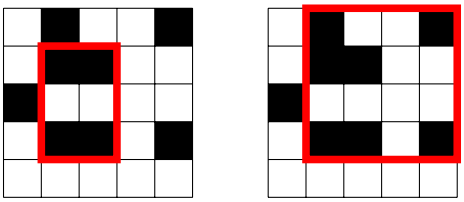
To compare the implementations, we generated a list of 10000 random bit strings of length 30. Using the first approach, the search took 13.5 seconds, and after the bit optimization, it only took 0.5 seconds. Thus, the bit optimized code was almost 30 times faster than the original code.

Counting subgrids

As another example, consider the following problem: Given an $n \times n$ grid whose each square is either black (1) or white (0), calculate the number of subgrids whose all corners are black. For example, the grid



contains two such subgrids:



There is an $O(n^3)$ time algorithm for solving the problem: go through all $O(n^2)$ pairs of rows and for each pair (a,b) calculate the number of columns that contain a black square in both rows in $O(n)$ time. The following code assumes that color[y][x] denotes the color in row y and column x:

```
int count = 0;
for (int i = 0; i < n; i++) {
    if (color[a][i] == 1 && color[b][i] == 1) count++;
}
```

Then, those columns account for count(count $-$ 1)/2 subgrids with black corners, because we can choose any two of them to form a subgrid.

To optimize this algorithm, we divide the grid into blocks of columns such that each block consists of N consecutive columns. Then, each row is stored as a list of N-bit numbers that describe the colors of the squares. Now we can process N columns at the same time using bit operations. In the following code, color[y][k] represents a block of N colors as bits.

```
int count = 0;
for (int i = 0; i <= n/N; i++) {
    count += __builtin_popcount(color[a][i]&color[b][i]);
}
```

The resulting algorithm works in $O(n^3/N)$ time.

We generated a random grid of size 2500×2500 and compared the original and bit optimized implementation. While the original code took 29.6 seconds, the bit optimized version only took 3.1 seconds with $N = 32$ (int numbers) and 1.7 seconds with $N = 64$ (long long numbers).

Dynamic programming

Bit operations provide an efficient and convenient way to implement dynamic programming algorithms whose states contain subsets of elements, because such states can be stored as integers. Next we discuss examples of combining bit operations and dynamic programming.

Optimal selection

As a first example, consider the following problem: We are given the prices of k products over n days, and we want to buy each product exactly once. However, we are allowed to buy at most one product in a day. What is the minimum total price? For example, consider the following scenario ($k = 3$ and $n = 8$):

	0	1	2	3	4	5	6	7
product 0	6	9	5	2	8	9	1	6
product 1	8	2	6	2	7	5	7	2
product 2	5	3	9	7	3	5	1	4

In this scenario, the minimum total price is 5:

	0	1	2	3	4	5	6	7
product 0	6	9	5	**2**	8	9	1	6
product 1	8	**2**	6	2	7	5	7	2
product 2	5	3	9	7	3	5	**1**	4

Let $\texttt{price}[x][d]$ denote the price of product x on day d. For example, in the above scenario $\texttt{price}[2][3] = 7$. Then, let $\texttt{total}(S,d)$ denote the minimum total price for buying a subset S of products by day d. Using this function, the solution to the problem is $\texttt{total}(\{0 \ldots k-1\}, n-1)$.

First, $\texttt{total}(\emptyset, d) = 0$, because it does not cost anything to buy an empty set, and $\texttt{total}(\{x\}, 0) = \texttt{price}[x][0]$, because there is one way to buy one product on the first day. Then, the following recurrence can be used:

$$\texttt{total}(S,d) = \min(\texttt{total}(S,d-1), \min_{x \in S}(\texttt{total}(S \setminus x, d-1) + \texttt{price}[x][d]))$$

This means that we either do not buy any product on day d or buy a product x that belongs to S. In the latter case, we remove x from S and add the price of x to the total price.

The next step is to calculate the values of the function using dynamic programming. To store the function values, we declare an array

```
int total[1<<K][N];
```

where K and N are suitably large constants. The first dimension of the array corresponds to a bit representation of a subset.

First, the cases where $d = 0$ can be processed as follows:

```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

Then, the recurrence translates into the following code:

```
for (int d = 1; d < n; d++) {
    for (int s = 0; s < (1<<k); s++) {
        total[s][d] = total[s][d-1];
        for (int x = 0; x < k; x++) {
            if (s&(1<<x)) {
                total[s][d] = min(total[s][d],
                                  total[s^(1<<x)][d-1]+price[x][d]);
            }
        }
    }
}
```

The time complexity of the algorithm is $O(n2^k k)$.

From permutations to subsets

Using dynamic programming, it is often possible to change an iteration over permutations into an iteration over subsets[1]. The benefit of this is that $n!$, the number of permutations, is much larger than 2^n, the number of subsets. For example, if $n = 20$, then $n! \approx 2.4 \cdot 10^{18}$ and $2^n \approx 10^6$. Thus, for certain values of n, we can efficiently go through the subsets but not through the permutations.

As an example, consider the following problem: There is an elevator with maximum weight x, and n people with known weights who want to get from the ground floor to the top floor. What is the minimum number of rides needed if the people enter the elevator in an optimal order?

For example, suppose that $x = 10$, $n = 5$ and the weights are as follows:

person	weight
0	2
1	3
2	3
3	5
4	6

In this case, the minimum number of rides is 2. One optimal order is $\{0,2,3,1,4\}$, which partitions the people into two rides: first $\{0,2,3\}$ (total weight 10), and then $\{1,4\}$ (total weight 9).

[1] This technique was introduced in 1962 by M. Held and R. M. Karp [34].

The problem can be easily solved in $O(n!n)$ time by testing all possible permutations of n people. However, we can use dynamic programming to get a more efficient $O(2^n n)$ time algorithm. The idea is to calculate for each subset of people two values: the minimum number of rides needed and the minimum weight of people who ride in the last group.

Let $\texttt{weight}[p]$ denote the weight of person p. We define two functions: $\texttt{rides}(S)$ is the minimum number of rides for a subset S, and $\texttt{last}(S)$ is the minimum weight of the last ride. For example, in the above scenario

$$\texttt{rides}(\{1,3,4\}) = 2 \quad \text{and} \quad \texttt{last}(\{1,3,4\}) = 5,$$

because the optimal rides are $\{1,4\}$ and $\{3\}$, and the second ride has weight 5. Of course, our final goal is to calculate the value of $\texttt{rides}(\{0 \ldots n-1\})$.

We can calculate the values of the functions recursively and then apply dynamic programming. The idea is to go through all people who belong to S and optimally choose the last person p who enters the elevator. Each such choice yields a subproblem for a smaller subset of people. If $\texttt{last}(S \setminus p) + \texttt{weight}[p] \leq x$, we can add p to the last ride. Otherwise, we have to reserve a new ride that initially only contains p.

To implement dynamic programming, we declare an array

```
pair<int,int> best[1<<N];
```

that contains for each subset S a pair $(\texttt{rides}(S), \texttt{last}(S))$. We set the value for an empty group as follows:

```
best[0] = {1,0};
```

Then, we can fill the array as follows:

```
for (int s = 1; s < (1<<n); s++) {
    // initial value: n+1 rides are needed
    best[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s&(1<<p)) {
            auto option = best[s^(1<<p)];
            if (option.second+weight[p] <= x) {
                // add p to an existing ride
                option.second += weight[p];
            } else {
                // reserve a new ride for p
                option.first++;
                option.second = weight[p];
            }
            best[s] = min(best[s], option);
        }
    }
}
```

Note that the above loop guarantees that for any two subsets S_1 and S_2 such that $S_1 \subset S_2$, we process S_1 before S_2. Thus, the dynamic programming values are calculated in the correct order.

Counting subsets

Our last problem in this chapter is as follows: Let $X = \{0 \ldots n-1\}$, and each subset $S \subset X$ is assigned an integer $\texttt{value}[S]$. Our task is to calculate for each S

$$\texttt{sum}(S) = \sum_{A \subset S} \texttt{value}[A],$$

i.e., the sum of values of subsets of S.

For example, suppose that $n = 3$ and the values are as follows:

- $\texttt{value}[\emptyset] = 3$
- $\texttt{value}[\{0\}] = 1$
- $\texttt{value}[\{1\}] = 4$
- $\texttt{value}[\{0,1\}] = 5$
- $\texttt{value}[\{2\}] = 5$
- $\texttt{value}[\{0,2\}] = 1$
- $\texttt{value}[\{1,2\}] = 3$
- $\texttt{value}[\{0,1,2\}] = 3$

In this case, for example,

$$\begin{aligned}\texttt{sum}(\{0,2\}) &= \texttt{value}[\emptyset] + \texttt{value}[\{0\}] + \texttt{value}[\{2\}] + \texttt{value}[\{0,2\}] \\ &= 3 + 1 + 5 + 1 = 10.\end{aligned}$$

Because there are a total of 2^n subsets, one possible solution is to go through all pairs of subsets in $O(2^{2n})$ time. However, using dynamic programming, we can solve the problem in $O(2^n n)$ time. The idea is to focus on sums where the elements that may be removed from S are restricted.

Let $\texttt{partial}(S,k)$ denote the sum of values of subsets of S with the restriction that only elements $0 \ldots k$ may be removed from S. For example,

$$\texttt{partial}(\{0,2\},1) = \texttt{value}[\{2\}] + \texttt{value}[\{0,2\}],$$

because we may only remove elements $0 \ldots 1$. We can calculate values of $\texttt{sum}$ using values of $\texttt{partial}$, because

$$\texttt{sum}(S) = \texttt{partial}(S, n-1).$$

The base cases for the function are

$$\texttt{partial}(S,-1) = \texttt{value}[S],$$

because in this case no elements can be removed from S. Then, in the general case we can use the following recurrence:

$$\texttt{partial}(S,k) = \begin{cases} \texttt{partial}(S,k-1) & k \notin S \\ \texttt{partial}(S,k-1) + \texttt{partial}(S \setminus \{k\},k-1) & k \in S \end{cases}$$

Here we focus on the element k. If $k \in S$, we have two options: we may either keep k in S or remove it from S.

There is a particularly clever way to implement the calculation of sums. We can declare an array

```
int sum[1<<N];
```

that will contain the sum of each subset. The array is initialized as follows:

```
for (int s = 0; s < (1<<n); s++) {
    sum[s] = value[s];
}
```

Then, we can fill the array as follows:

```
for (int k = 0; k < n; k++) {
    for (int s = 0; s < (1<<n); s++) {
        if (s&(1<<k)) sum[s] += sum[s^(1<<k)];
    }
}
```

This code calculates the values of $\texttt{partial}(S,k)$ for $k = 0 \ldots n-1$ to the array `sum`. Since $\texttt{partial}(S,k)$ is always based on $\texttt{partial}(S,k-1)$, we can reuse the array `sum`, which yields a very efficient implementation.