STL

vector

创建

可笼统理解为变长数组

```
vector<int> a; vector<char> aa; vector<long long> aaa;
vector<int> b(5, 10);
int m = 5;
vector<int> bb(m, 5);
vector<int> g[10];
vector<vector<int>> c(10, vector<int>(10, 10));
```

常用函数

```
vector<int> a;
a.push_back(); //插入一个数
a.emplace_back(); //插入一个数，据说比push_back快点
a.clear(); //清除所有元素
a.pop_back(); //删除最后一个元素
a.size(); //返回a中所含元素的数量
a.empty(); //布尔类型，表示a是否非空
a.front(); //第一个元素的值
a.back(); //最后一个元素的值
sort(a.begin(), a.end()); //升序排序
sort(a.begin(), a.end(), greater<>()); //降序排序

bool cmp()
sort(a.begin(), a.end(), cmp); //自定义排序
```

string

字符串类型，比C语言的char数组好用（除了从0开始得把习惯转换一下）

```
string s = "";
string s = "xxx";
s = s + s;
s = s + s + t;

s.insert(x, "xxx"); //在指定位置处插入一个字符串
s.erase(x, n); //删除指定位置开始的长度为n的字符串
s.size()/s.length(); //返回字符串的长度
s.empty();

字符串重载了运算符，实现了字符串之间的比较<, >, =

int pos = s.find("xxx"); //查找原串中第一次出现该字符串的位置，若没有返回-1
int pos = s.rfind("xxx"); //查找原串中最后一次出现该字符串的位置，若没有返回-1
```

```
string t = s.substr(x, y); //返回原串从第x+1个字符开始，长度为y的一个子串
```

queue

普通队列，先进先出的数据结构

```
queue<int> q; //创建队列
q.push(1); //把元素推入队尾
q.pop(); //移出队头元素
q.front(); //得到队头元素
q.back(); //得到队尾元素
q.empty(); //判断队列是否为空
q.size(); //得到队列中已有元素的数量
```

stack

栈，先进后出的数据结构

```
stack<int> st; //创建队列
st.push(1); //把元素推入栈顶
st.pop(); //弹出栈顶元素
st.top(); //访问栈顶元素
st.empty(); //判断栈是否为空
st.size(); //栈中已有元素的数量
```

双端队列deque

```
deque<int> dq; //创建双端队列
dp.push_back(1); //在队尾插入元素
dq.push_front(1); //在队头插入元素
dq.pop_back(); //删除队尾元素
dq.pop_front(); //删除队头元素
dq.front(); //访问队头元素
dq.back(); //访问队尾元素

dq.size();
dq.empty();
dq.clear();
```

pair

二元组，很好用。

```
一个二元组，与结构体类似，不过STL已经帮我们重载好了运算符
pair<int, int> a = pair<int, int> (x, y); //定义方式1
pair<int, int> b = {x, y}; //定义方式2
pair<int, int> c = make_pair(x, y); //定义方式3
pair<int, pair<int, int>> d = make_pair(1, make_pair(2, 3));
typedef pair<int, int> pii; //简写为三个字母，第一个字母表示pair，第二和第三个表示元素的类型

pair<int, int> //第一个元素叫做first，第二个元素叫做second
```

```
int x = a.first, y = a.second;
int da = a.first, db = a.second.first, dc = a.second.second;

//pair已经重载完了运算符，以first为第一关键字，second为第二关键字，升序排序
vector<pii> a; sort(a.begin(), a.end());
比如(1, 2), (2, 3), (1, 3) 排序后结果为 (1, 2), (1, 3), (2, 3)
```

tuple

不咋常见，除非要用到运算符，三个以上的元素一般我都是直接定义结构体的

```
类似于pair，不过元组数量不限，可以是任意个
tuple<int, string, int> a = make_tuple(1, "ame", 3);
tuple<int, string, int> b = {2, "maybe", 4};
tuple<int, string, string> c = {2, "csdx", "oldchicken"};
typedef tuple<int, int, int> tiii;
tiii aa = {1, 2, 3};

get<x>(y); //取出y这个tuple的第x+1个元素的值（最小的x从0开始）
int xx = get<0>(aa), xxx = get<1>(aa), xxxx = get<2>(aa);
xx = 1, xxx = 2, xxxx = 3;

//tuple也重载了运算符，排序规则和pair一样，也可以放入vector排序
```

priority_queue优先队列

优先队列每次只能取出队头元素，“队头”元素是队列中优先级最高的元素。

优先级的定义默认值越大优先级越高（大根堆），可以在定义的时候多加几个参数改成小根堆（值越小优先级越高）。

优先级内部的元素必须拥有或者已经重载好了比较运算符，否则程序会报错。

```
priority_queue<int> q; //默认定义方式
priority_queue<int, vector<int>, greater<int>> pq; //更改优先级，小的优先级更高，不同类型只需把int换成对应类型即可，如double, string, pair<int, int>

q.push(x); //把一个元素推入优先队列，时间复杂度O(logN)
q.pop(); //弹出“队头”元素
q.top(); //访问队头元素
q.size(); //优先队列元素个数
q.empty(); //判断优先队列是否为空

```

set

自动排序（默认升序），去重

```
set<int> s;
s.size();
s.empty();
s.clear();
s.begin()/end();
++, --返回前驱和后继，时间复杂度O(logn)
重载运算符时要重载全
```

```
s.insert(x); //往set中插入某一个数
s.count(x); //返回某一个数的个数
s.erase(x); //若是一个数，删除这个数字 时间复杂度O(k+logn)
            //若是一个迭代器，删除这个迭代器
s.lower_bound(x); //返回大于等于x的最小的数的迭代器
s.upper_bound(x); //返回大于x的最小的数的迭代器

for (auto it : s) {
    //it是整形数字，直接拿来用就行
    cout << it << endl;
}
for (set<int>::iterator it = s.begin(); it != s.end(); it++) {
    //it是迭代器，要想获得数值得在前面加上一个符号'*'
    int x = *it;
    cout << x << endl;
}

multiset，用法和set基本一致，multiset不会去重，允许重复元素的出现
```

map

可以理解为一个比较灵活的数组，下标可以为任意值（必须要有比较运算符）

map<type1, type2>, type1称为key值，type2称为value值，根据key去找value，建立key到value的一个映射

map虽然空间大但基本够用，题目不会在这里卡。若定义域太大普通数组声明不了时，其中一种解决方法就是用map

```
map<int, int> mp;
mp.size();
mp.empty();
mp.clear();
mp.begin()/end();
++,--返回前驱和后继

mp.insert(make_pair(x, y));//插入一个二元组
mp[1] = 1; //插入的令一种方法，与insert等价
mp.count(); //返回某一个数的个数
mp.erase(); //删除一个数
lower_bount()/upper_bound();

for (auto it : mp) { //map的遍历方式
    int x = it.first;
    int y = it.second;
    cout << x << " " << y << ndl;
}

unordered_map和map一样，插入删除的时间复杂度都是O(1)，据说会更快点。
unordered_map不支持lower_bound，upper_bound，迭代器的++，-- 操作
```

bitset(了解)

```
bitset大概就是类似于bool数组一样的东西
但是它的每个位置只占1bit（特别特别小）
bitset的原理大概是将很多数压成一个，从而节省空间和时间（暴力出奇迹）
```

```
一般来说bitset会让你的算法复杂度 /32

定义方法：bitset<10000> s;

bitset类型可以用string和整数初始化（整数转化成对应的二进制）
bitset<23>bit (string("11101001"));
cout<<bit<<endl;
bit=233;
cout<<bit<<endl;
输出结果：
00000000000000011101001
00000000000000011101001

bitset支持所有位运算
bitset<23>bita(string("11101001"));
bitset<23>bitb(string("11101000"));
cout<<(bita^bitb)<<endl;
//输出00000000000000000000001
bitset<23>bita(string("11101001"));
bitset<23>bitb(string("11101000"));
cout<<(bita|bitb)<<endl;
//输出00000000000000011101001
bitset<23>bita(string("11101001"));
bitset<23>bitb(string("11101000"));
cout<<(bita&bitb)<<endl;
//输出00000000000000011101000
bitset<23>bit(string("11101001"));
cout<<(bit<<5)<<endl;
//输出00000000011110100100000
bitset<23>bit(string("11101001"));
cout<<(bit>>5)<<endl;
//输出00000000000000000000111
~, &, |, ^
>>, <<
==, !=
[]

bitset方法
bit.size()        返回大小（位数）
bit.count()      返回1的个数
bit.any()        返回是否有1
bit.none()       返回是否没有1
bit.set()        全都变成1
bit.set(p)       将第p + 1位变成1（bitset是从第0位开始的！）
bit.set(p, x)    将第p + 1位变成x
bit.reset()      全都变成0
bit.reset(p)     将第p + 1位变成0
bit.flip()       全都取反
bit.flip(p)      将第p + 1位取反
bit.to_ulong()   返回它转换为unsigned long的结果，如果超出范围则报错
bit.to_ullong()  返回它转换为unsigned long long的结果，如果超出范围则报错
bit.to_string()  返回它转换为string的结果
```