目录

[TOC]

1 STL 和一些函数

1.1 vector

用法	含义
v[i]	直接以下标方式访问容器中的元素
v.front()	返回首元素
v.back()	返回尾元素
v.push_back(x)	向表尾插入元素 x

用法	含义
v.pop_back()	删除表尾元素
v.begin()	返回指向首元素的迭代器
v.end()	返回指向尾元素的下一个位置的迭代器
v.size()	返回表长
v.empty()	当表空时，返回真，否则返回假
v.clear()	删除容器中的所有的元素
v.insert(it, val)	向迭代器 it 指向的元素前插入新元素 val
v.insert(it, n, x)	向迭代器 it 指向的元素前插入 n 个 x
v.insert(it, first, last)	将由迭代器 first 和 last 所指定的序列 [first, last) 插入到迭代器 it 指向的元素前面
v.erase(it)	删除由迭代器 it 所指向的元素
v.erase(first, last)	删除由迭代器 first 和 last 所指定的序列 [first, last)
v.resize(n)	改变序列的长度，超出的元素将会被删除，如果序列需要扩展（原空间小于 n），元素默认值将填满扩展出的空间
v.resize(n, val)	改变序列的长度，超出的元素将会被删除，如果序列需要扩展（原空间小于 n），将用 val 填满扩展出的空间
v.swap(v2)	将 s 与另一个 vector 对象 v2 进行交换
v.assign(first, last)	将序列替换成由迭代器 first 和 last 所指定的序列 [first, last), [first, last) 不能是原序列中的一部分

1.1.1 注意

- vector 不自带 v.find() 函数

1.2 set & multiset & unordered_set

用法	含义
s.begin()	返回指向第一个元素的迭代器
s.end()	返回指向最后一个元素的迭代器
s.clear()	清除所有元素
s.empty()	如果集合为空，返回 true
s.count(val)	返回值为 val 的元素的个数
s.erase(val)	删除集合中**所有**值为 val 的元素
s.erase(it)	删除集合中迭代器 it 指向的元素
s.erase(first,last)	删除由迭代器 first 和 last 所指定的子集 [first, last)
s.equal_range(val)	返回有序/升序集合中 val 元素第一次和最后一次出现的位置
s.find()	返回一个指向被查找到元素的迭代器
s.insert(val)	在集合中插入值为 val 的元素, 返回值为 pair<set::iterator, bool>,pair::first 被设置为指向新插入元素的迭代器或指向等值的已经存在的元素的迭代器。成员 pair::second 是一个 bool 值，如果新的元素被插入，返回 true，如果等值元素已经存在（即无新元素插入），则返回 false。(set.insert(x).second)

用法	含义
s.max_size()	返回集合能容纳的元素的最大限值
s.rbegin()	返回指向集合中最后一个元素的反向迭代器
s.rend()	返回指向集合中第一个元素的反向迭代器
s.size()	集合中元素的数目
s.swap(s2)	交换两个集合变量
s.upper_bound(val)	返回大于 val 值元素的迭代器
s.lower_bound(val)	返回指向大于（或等于）val 值的第一个元素的迭代器

1.2.1 注意

- set 无法用下标访问，迭代器也不能进行数的加减，无法直接访问第 k 个元素
- set/multiset 自动有序，无法使用排序函数
- unordered_set 只能使用前向迭代器
- set、multiset 头文件均为 set，unordered_set 头文件为 unordered_set
- multiset 的 erase, **当传入的是值，将删除所有相同的值**; 若传入的是迭代器, 则只删除对应元素

1.3 stack

用法	含义
st.empty()	堆栈为空则返回真
st.pop()	移除栈顶元素
st.push(val)	在栈顶增加元素 val
st.size()	返回栈中元素数目
st.top()	返回栈顶元素

1.3.0.1 注意

- 栈满足先入后出原则

1.4 queue

用法	含义
q.push()	入队
q.pop()	出队
q.front()	返回首元素
q.back()	返回末元素
q.size()	输出现有元素的个数
q.empty()	队列为空返回 1，反之返回 0

1.4.0.1 注意

- 队列满足先入先出原则
- 队列可以用 [] 访问

1.5 priority_queue（优先队列）

用法	含义
p.empty()	判断是否为空
p.push(val)	队列尾部增加元素 val
p.pop()	队列头部数据出队
p.top()	返回头部数据
p.size()	返回栈中元素个数

1.6 deque（双向队列）

用法	含义
d.assign(first, last)	将 [first, last) 区间中的元素赋值给 d
d.assign(n,val)	将 n 个 val 赋值给 d
d.at(index)	传回索引 index 所指的元素，如果 index 越界，抛出 out_of_range
d.begin()	返回首元素地址
d.end()	返回尾元素地址
d.front()	返回首元素
d.back()	返回尾元素
d.clear()	移除容器中所有元素
d.empty()	判断容器是否为空
d.erase(pos)	删除 pos 位置的元素，传回下一个元素的位置
d.erase(first, last)	删除 [first, last) 区间的元素，传回下一个元素的位置
d.insert(pos,val)	在 pos 位置插入 val，传回新元素位置
d.insert(pos,n,val)	在 pos 位置插入 n 个 val 元素，无返回值
d.insert(pos,first, last)	在 pos 位置插入在 [first, last) 区间的元素，无返回值
d.pop_back()	删除最后一个元素
d.pop_front()	删除头部元素
d.push_back(val)	在尾部加入一个元素
d.push_front(val)	在头部插入一个元素
d.rbegin()	传回一个逆向队列的第一个元素
d.rend()	传回一个逆向队列的最后一个元素的下一个位置
d.resize(num)	重新指定队列的长度
d.size()	返回容器中实际元素的个数

1.7 map & multimap & unordered_map

用法	含义
mp[0] = x	利用数组方式插入数据，0 是键，x 是值
mp.at(0) = x	利用 at 执行插入操作
mp.insert(make_pair(key,x))	利用 insert 插入 pair(键，值) 数据
mp.emplace(make_pair(key,x))	在映射中不存在主键 key 时执行插入操作
mp.size()	返回 mp 的大小
mp.count(key)	统计键为 key 的元素存在的映射数，存在返回 1，不存在返回 0
mp.erase(it)	根据迭代器删除元素
mp.clear()	清空映射
mp.empty()	判断映射是否为空
mp.find(key)	根据键 key 查找元素，找到以后返回迭代器
mp.rbegin()	返回反向迭代器
mp.rend()	返回反向迭代器
mp.swap(mp2)	将 mp 和 mp2 进行交换
mp.lower_bound(key)	返回 map 中第一个大于或等于 key 的迭代器指针
mp.upper_bound(key)	返回 map 中第一个大于 key 的迭代器指针

```
map<int, string> mapStudent;
mapStudent[0] = "student_zero";
mapStudent.insert(pair<int, string>(1, "student_one"));
mapStudent.insert(map<int, string>::value_type (2, "student_two"));
```

1.7.0.1 三种插入操作示例

1.7.0.2 注意

- map 的键值 key 不可重复，具有严格的一一对应关系，而 multimap 可以一个键对应多个值
- map 支持 [ ] 运算符，multimap 不支持 [ ] 运算符
- map/multimap 中的元素是自动按 key 升序排序，不能使用 sort 函数

1.8 list

用法	含义
l.begin()	返回链表首地址
l.end()	返回链表尾地址
l.front()	返回首元素
l.back()	返回尾元素
l.push_back(elem)	插入元素到链表尾部

用法	含义
l.push_front(elem)	插入元素到链表头部
l.empty()	判断链表是否为空
l.insert(it, val1, val2))	在指定位置插入一个或多个元素
l.resize(n)	调整链表大小为 n，超出 n 删除，少于 n 补 0
l.clear()	清除
l.assign(len, val)	替换所有元素
l.assign(l2.begin(), l2.end())	替换所有元素为链表 l2
l.swap(l2)	交换链表
l.merge(l2)	合并两个**有序**链表中的元素，调用后 l2 为空，可用 greater()
l.erase(it)	删除（区域中的）元素
l.remove(val)	删除值为 val 的元素

1.9 bitset

其他参见《算法竞赛进阶指南》

```
// 赋初值
bitset<100>foo('1110');
bitset<100>foo(100);

foo.to_ulong() //返回它转换为 unsigned long 的结果，如果超出范围则报错
foo.to_ullong() //返回它转换为 unsigned long long 的结果，如果超出范围则报错
foo.to_string() //返回它转换为 string 的结果
```

1.10 lower_bound & upper_bound

用法	含义
lower_bound(first, last, val)	在 [first,last) 区间进行二分查找，返回**大于或等于** val 的第一个元素位置
upper_bound(first, last, val)	在 [first,last) 区间进行二分查找，返回**大于** val 的第一个元素位置

1.10.0.1 注意

- 只能作用于有序序列
- 如果所有元素都小于 val，函数返回 last 的位置，此时 last 的位置是**越界**的!
- set::lower_bound() 由于省略了再建树的过程，速度快于 lower_bound()

1.11 pair

1.11.0.1 使用 sort 对 pair 类型进行排序

- std::pair::operator< 按标准规定会在两个 std::pair 的第一个元素互不小于对方的情况下比较第二个元素

1.12 unique

- 将数组的不重复元素移到前面来，返回重复元素的首地址
- **使用前必须排序！**

```
//原型： iterator unique(iterator it_1,iterator it_2); 返回值是一个迭代器，
//它指向的是去重后容器中不重复序列的最后一个元素的下一个元素。
vector<int> a;
a.push_back(1);a.push_back(3);a.push_back(3);
a.push_back(4);a.push_back(4);a.push_back(5);
//用 erase 去重
a.erase(unique(a.begin(),a.end()),a.end());
```

1.13 string

1.13.1 substr ：

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    ios::sync_with_stdio(false);
    string s="abcdefg";

    //s.substr(pos1,n) 返回字符串位置为 pos1 后面的 n 个字符组成的串
    string s2=s.substr(1,5);//bcdef

    //s.substr(pos)//得到一个 pos 到结尾的串
    string s3=s.substr(4);//efg

    return 0;
}
```

1.13.2 replace 操作

```
#include <iostream>
#include <string>
```

```
int main ()
{
    std::string base="this is a test string.";
    std::string str2="n example";
    std::string str3="sample phrase";
    std::string str4="useful.";

    // replace signatures used in the same order as described above:

    // Using positions:                 0123456789*123456789*12345
    std::string str=base;           // "this is a test string."
    //第 9 个字符以及后面的长度 5 个字符被 str2 代替
    str.replace(9,5,str2);          // "this is an example string." (1)

    return 0;
}
```

1.13.3 find

```
//查找所有子串在母串中出现的位置
//查找 s 中 flag 出现的所有位置。没有返回 string::npos
    flag="a";
    position=0;
    int i=1;
    while((position=s.find(flag,position))!=string::npos)
    {
        cout<<"position  "<<i<<" : "<<position<<endl;
        position++;
        i++;
    }
    /*
    position 1:1
    position 2:21
    position 3:36
    */
```

1.14 其他

- ___builtin_popcount() 计算一个数字的二进制中有多少个 1，返回值 1 的个数。
- ___gcd() 自带 gcd

1.14.0.1 注意

- 以上可能不让用

1.15 一些函数

1.15.1 ___int128

___int128 在 gcc、codeblocks、vs2017 都是不被支持的，不过 ___int128 在 Linux 上可以编译并且能用。

```
##include<iostream>
using namespace std;
inline __int128 read(){
    __int128 x = 0, f = 1;
    char ch = getchar();
    while(ch < '0' || ch > '9'){
        if(ch == '-')
            f = -1;
        ch = getchar();
    }
    while(ch >= '0' && ch <= '9'){
        x = x * 10 + ch - '0';
        ch = getchar();
    }
    return x * f;
}
inline void print(__int128 x){
    if(x < 0){
        putchar('-');
        x = -x;
    }
    if(x > 9)
        print(x / 10);
    putchar(x % 10 + '0');
}
int main(void){
    __int128 a = read();
    __int128 b = read();
    print(a + b);
    cout << endl;
    return 0;
}
```

1.15.2 strchr

C 库函数 char *strchr(const char* str, int c) 在参数 str 所指向的字符串中搜索第一次出现字符 c（一个无符号字符）的位置。

```
char *strchr(const char *str, int c)
```

1.15.2.1 函数原型： str –要被检索的 C 字符串。c –在 str 中要搜索的字符。函数返回在字符串 str 中第一次出现字符 c 的位置 (地址)，如果未找到该字符则返回 NULL。#### 示例

```
##include <stdio.h>
##include <string.h>
int main (){
   const char str[] = "http://www.runoob.com";
   const char ch = '.';
   char *ret;
   ret = strchr(str, ch);
   printf("|%c| 之后的字符串是 - |%s|\n", ch, ret);
   return(0);
}
```

结果：

```
|.| 之后的字符串是 - |.runoob.com|
```

1.15.3 关于 sscanf

一个字符串中读进与指定格式相符的数据。字符串操作是平常用途之多，截取，追加等等。也经常从文件中读取一行，取出所需要的字符串。基本有些是固定格式的。都可以用 sscanf 来得到。

例如：

```
sscanf(line, "%[A-Z]%d", t, &r);
sscanf(line,"R%dC%d", &r, &c);
```

上述知识点可参见 Codeforces 1B

https://vjudge.net/contest/387797#problem/B

1.15.4 stringstream 类

构造：

```
stringstream::str (const string& s);
```

sets s as the contents of the stream, discarding any previous contents.

3.stringstream 清空，

```
stringstream s; s.str("");
```

示例：将字符串”1 2 3 4 5”依次输入

```
while(ss>>x)a[n++]=x;
```

1.15.5 memcpy 函数

```
int src[6][3]={{1,2,3},{4,5,6},{7,8,9},{1,2,3},{4,5,6},{7,8,9}};
int des[6][3];//要小心，行数固定
memcpy(des,src,sizeof(src));
```

1.15.6 ___builtin_popcount() 用于计算一个 32 位无符号整数有多少个位为 1

Counting out the bits 可以很容易的判断一个数是不是 2 的幂次：清除最低的 1 位（见上面）并且检查结果是不是 0. 尽管如此，有的时候需要直到有多少个 1 被设置了，这就相对有点难度了。

GCC 有一个叫做 ___builtin_popcount 的内建函数，它可以精确的计算 1 的个数。尽管如此，不同于 ___builtin_ctz，它并没有被翻译成一个硬件指令（至少在 x86 上不是）。相反的，它使用一张类似上面提到的基于表的方法来进行位搜索。这无疑很**高效**并且非常方便。

其他语言的使用者没有这个选项（尽管他们可以重新实现这个算法）。如果一个数只有很少的 1 的位，另外一个方法是重复的获取最低的 1 位，并且清除它。

```
for(int i = 1;i <= 100; i++){
        cout<<__builtin_popcount(i)<<endl;
    }
```

2 计时

```
#include <chrono>
using namespace std;
using namespace chrono;

auto start = system_clock::now();
```

```
// do something...

auto end   = system_clock::now();
auto duration = duration_cast<microseconds>(end - start);
cout <<  " 花费了"
    << double(duration.count()) * microseconds::period::num / microseconds::period::den
    << " 秒" << endl;
```