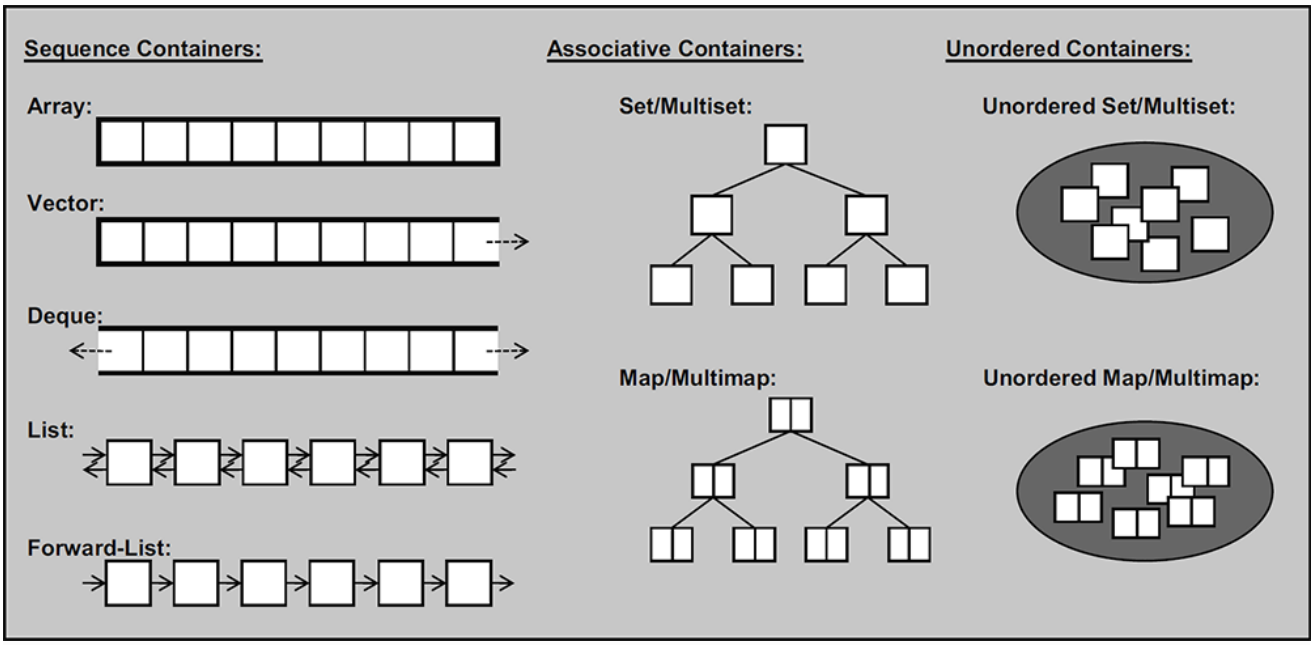
STL容器

分类



序列式容器

- **向量**(`vector`) 后端可高效增加元素的顺序表。
- **数组**(`array`)**C++11**，定长的顺序表，C 风格数组的简单包装。
- **双端队列**(`deque`) 双端都可高效增加元素的顺序表。
- **列表**(`list`) 可以沿双向遍历的链表。
- **单向列表**(`forward_list`) 只能沿一个方向遍历的链表。

关联式容器

- **集合**(`set`) 用以有序地存储 **互异** 元素的容器。其实现是由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种比较元素大小的谓词进行排列。
- **多重集合**(`multiset`) 用以有序地存储元素的容器。允许存在相等的元素。
- **映射**(`map`) 由 {键，值} 对组成的集合，以某种比较键大小关系的谓词进行排列。
- **多重映射**(`multimap`) 由 {键，值} 对组成的多重集合，亦即允许键有相等情况的映射。

无序（关联式）容器

- **无序（多重）集合**(`unordered_set` / `unordered_multiset`)**C++11**，与 `set` / `multiset` 的区别在与元素无序，只关心”元素是否存在“，使用哈希实现。
- **无序（多重）映射**(`unordered_map` / `unordered_multimap`)**C++11**，与 `map` / `multimap` 的区别在与键 (key) 无序，只关心 "键与值的对应关系"，使用哈希实现。

容器适配器

容器适配器其实并不是容器。它们不具有容器的某些特点（如：有迭代器、有 `clear()` 函数.....）。

> ”适配器是使一种事物的行为类似于另外一种事物行为的一种机制”，适配器对容器进行包装，使其表现出另外一种行为。

- **栈** (`stack`) 后进先出 (LIFO) 的容器。
- **队列**(`queue`) 先进先出 (FIFO) 的容器。
- **优先队列**(`priority_queue`) 元素的次序是由作用于所存储的值对上的某种谓词决定的的一种队列。

容器声明

都是 `containerName<typeName,...> name` 的形式，但模板参数（`<>` 内的参数）的个数、形式会根据具体容器而变。

本质原因：STL 就是“标准模板库”，所以容器都是模板类。

迭代器

在 STL 中，迭代器（Iterator）用来访问和检查 STL 容器中元素的对象，它的行为模式和指针类似，但是它封装了一些有效性检查，并且提供了统一的访问格式。类似的概念在其他很多高级语言中都存在，如 Python 的 `__iter__` 函数，C# 的 `IEnumerator`。

基础使用

迭代器听起来比较晦涩，其实迭代器本身可以看作一个数据指针。迭代器主要支持两个运算符：自增 (`++`) 和解引用（单目 `*` 运算符），其中自增用来移动迭代器，解引用可以获取或修改它指向的元素。

指向某个 **STL 容器** `container` 中元素的迭代器的类型一般为 `container::iterator`。

迭代器可以用来遍历容器，例如，下面两个 for 循环的效果是一样的：

```cpp
vector<int> data(10);

for (int i = 0; i < data.size(); i++)
  cout << data[i] << endl;  // 使用下标访问元素

for (vector<int>::iterator iter = data.begin(); iter != data.end(); iter++)
  cout << *iter << endl;  // 使用迭代器访问元素
// 在C++11后可以使用 auto iter = data.begin() 来简化上述代码
```

分类

在 STL 的定义中，迭代器根据其支持的操作依次分为以下几类：

- InputIterator（输入迭代器）：只要求支持拷贝、自增和解引访问。
- OutputIterator（输出迭代器）：只要求支持拷贝、自增和解引赋值。
- ForwardIterator（向前迭代器）：同时满足 InputIterator 和 OutputIterator 的要求。
- BidirectionalIterator（双向迭代器）：在 ForwardIterator 的基础上支持自减（即反向访问）。
- RandomAccessIterator（随机访问迭代器）：在 BidirectionalIterator 的基础上支持加减运算和比较运算（即随机访问）。

其实这个“分类”并不互斥——一个“类别”是可以包含另一个“类别”的。例如，在要求使用向前迭代器的地方，同样可以使用双向迭代器。

不同的 **STL 容器** 支持的迭代器类型不同，在使用时需要留意。

指针满足随机访问迭代器的所有要求，可以当作随机访问迭代器使用。

相关函数

很多 **STL 函数** 都使用迭代器作为参数。

可以使用 `std::advance(it, n)` 获取迭代器 `it` 的增加 `n` 步的迭代器；若 `n` 为负数，则尝试获取迭代器的前驱，此时若迭代器不满足双向迭代器，行为未定义。

在 C++11 以后可以使用 `std::next(it)` 获取向前迭代器 `it` 的后继，`std::next(it, n)` 获取向前迭代器 `it` 的第 `n` 个后继。

在 C++11 以后可以使用 `std::prev(it)` 获取双向迭代器 `it` 的前驱，`std::prev(it, n)` 获取双向迭代器 `it` 的第 `n` 个前驱。

STL 容器 一般支持从一端或两端开始的访问，以及对 **const 修饰符** 的支持。例如容器的 `begin()` 函数可以获得指向容器第一个元素的迭代器，`rbegin()` 函数可以获得指向容器最后一个元素的反向迭代器，`cbegin()` 函数可以获得指向容器第一个元素的 const 迭代器，`end()` 函数可以获得指向容器尾端（“尾端”并不是最后一个元素，可以看作是最后一个元素的后继；“尾端”的前驱是容器里的最后一个元素，其本身不指向任何一个元素）的迭代器。

共有函数

`=`：有赋值运算符以及复制构造函数。

`begin()`：返回指向开头元素的迭代器。

`end()`：返回指向末尾的下一个元素的迭代器。`end()` 不指向某个元素，但它是末尾元素的后继。

`size()`：返回容器内的元素个数。

`max_size()`：返回容器 **理论上** 能存储的最大元素个数。依容器类型和所存储变量的类型而变。

`empty()`：返回容器是否为空。

`swap()`：交换两个容器。

`clear()`：清空容器。

`==` / `!=` / `<` / `>` / `<=` / `>=`：按 **字典序** 比较两个容器的大小。（比较元素大小时 `map` 的每个元素相当于 `set<pair<key, value> >`，无序容器不支持 `<` / `>` / `<=` / `>=`。）

序列式容器

vector

`std::vector` 是 STL 提供的 **内存连续的**、**可变长度** 的数组（亦称列表）数据结构。能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问。

为什么要使用 `vector`

作为 Oler，对程序效率的追求远比对工程级别的稳定性要高得多，而 `vector` 由于其对内存的动态处理，时间效率在部分情况下低于静态数组，并且在 OJ 服务器不一定开全优化的情况下更加糟糕。所以在正常存储数据的时候，通常不选择 `vector`。下面给出几个 `vector` 优秀的特性，在需要用到这些特性的情况下，`vector` 能给我们带来很大的帮助。

`vector` 可以动态分配内存

很多时候我们不能提前开好那么大的空间（eg：预处理 1~n 中所有数的约数）。尽管我们能知道数据总量在空间允许的级别，但是单份数据还可能非常大，这种时候我们就需要 `vector` 来把内存占用量控制在合适的范围内。`vector` 还支持动态扩容，在内存非常紧张的时候这个特性就能派上用场了。

`vector` 重写了比较运算符及赋值运算符

`vector` 重载了六个比较运算符，以字典序实现，这使得我们可以方便的判断两个容器是否相等（复杂度与容器大小成线性关系）。例如可以利用 `vector<char>` 实现字符串比较（当然，还是用 `std::string` 会更快更方便）。另外 `vector` 也重载了赋值运算符，使得数组拷贝更加方便。

`vector` 便利的初始化

由于 `vector` 重载了 `=` 运算符，所以我们可以方便的初始化。此外从 C++11 起 `vector` 还支持 **列表初始化**，例如 `vector<int> data {1, 2, 3};`。

`vector` 的使用方法

以下介绍常用用法，详细内容 **请参见 C++ 文档**。

构造函数

用例参见如下代码（假设你已经 `using` 了 `std` 命名空间相关类型）：

```
// 1. 创建空vector; 常数复杂度
vector<int> v0;
// 1+. 这句代码可以使得向vector中插入前3个元素时，保证常数时间复杂度
v0.reserve(3);
// 2. 创建一个初始空间为3的vector，其元素的默认值是0; 线性复杂度
vector<int> v1(3);
// 3. 创建一个初始空间为3的vector，其元素的默认值是2; 线性复杂度
vector<int> v2(3, 2);
// 4. 创建一个初始空间为3的vector，其元素的默认值是1,
// 并且使用v2的空间配置器; 线性复杂度
vector<int> v3(3, 1, v2.get_allocator());
// 5. 创建一个v2的拷贝vector v4,  其内容元素和v2一样; 线性复杂度
vector<int> v4(v2);
// 6. 创建一个v4的拷贝vector v5，其内容是{v4[1], v4[2]}; 线性复杂度
vector<int> v5(v4.begin() + 1, v4.begin() + 3);
// 7. 移动v2到新创建的vector v6，不发生拷贝; 常数复杂度; 需要 C++11
vector<int> v6(std::move(v2));  // 或者 v6 = std::move(v2);
```

可以利用上述的方法构造一个 `vector`，足够我们使用了。

元素访问

`vector` 提供了如下几种方法进行元素访问

1. `at()`

   `v.at(pos)` 返回容器中下标为 `pos` 的引用。如果数组越界抛出 `std::out_of_range` 类型的异常。

2. `operator[]`

   `v[pos]` 返回容器中下标为 `pos` 的引用。不执行越界检查。

3. `front()`

   `v.front()` 返回首元素的引用。

4. `back()`

   `v.back()` 返回末尾元素的引用。

5. `data()`

`v.data()` 返回指向数组第一个元素的指针。

迭代器

vector 提供了如下几种 **迭代器**

1. `begin()/cbegin()`

   返回指向首元素的迭代器，其中 `*begin = front`。

2. `end()/cend()`

   返回指向数组尾端占位符的迭代器，注意是没有元素的。

3. `rbegin()/crbegin()`

   返回指向逆向数组的首元素的逆向迭代器，可以理解为正向容器的末元素。

4. `rend()/crend()`

   返回指向逆向数组末元素后一位置的迭代器，对应容器首的前一个位置，没有元素。

以上列出的迭代器中，含有字符 `c` 的为只读迭代器，你不能通过只读迭代器去修改 `vector` 中的元素的值。如果一个 `vector` 本身就是只读的，那么它的一般迭代器和只读迭代器完全等价。只读迭代器自 C++11 开始支持。

长度和容量

`vector` 有以下几个与容器长度和容量相关的函数。注意，`vector` 的长度（size）指有效元素数量，而容量（capacity）指其实际分配的内存长度，相关细节请参见后文的实现细节介绍。

与长度相关：

- `empty()` 返回一个 `bool` 值，即 `v.begin() == v.end()`，`true` 为空，`false` 为非空。
- `size()` 返回容器长度（元素数量），即 `std::distance(v.begin(), v.end())`。
- `resize()` 改变 `vector` 的长度，多退少补。补充元素可以由参数指定。
- `max_size()` 返回容器的最大可能长度。

与容量相关：

- `reserve()` 使得 `vector` 预留一定的内存空间，避免不必要的内存拷贝。
- `capacity()` 返回容器的容量，即不发生拷贝的情况下容器的长度上限。
- `shrink_to_fit()` 使得 `vector` 的容量与长度一致，多退但不会少。

元素增删及修改

- `clear()` 清除所有元素
- `insert()` 支持在某个迭代器位置插入元素、可以插入多个。**复杂度与 `pos` 距离末尾长度成线性而非常数的**
- `erase()` 删除某个迭代器或者区间的元素，返回最后被删除的迭代器。复杂度与 `insert` 一致。
- `push_back()` 在末尾插入一个元素，均摊复杂度为 **常数**，最坏为线性复杂度。
- `pop_back()` 删除末尾元素，常数复杂度。
- `swap()` 与另一个容器进行交换，此操作是 **常数复杂度** 而非线性的。

`vector` 的实现细节

`vector` 的底层其实仍然是定长数组，它能够实现动态扩容的原因是增加了避免数量溢出的操作。首先需要指明的是 `vector` 中元素的数量（长度） 与它已分配内存最多能包含元素的数量（容量） 是不一致的，`vector` 会分开存储这两个量。当向 `vector` 中添加元素时，如发现 ，那么容器会分配一个尺寸为 的数组，然后将旧数据从原本的位置拷贝到新的数组中，再将原来的内存释放。尽管这个操作的渐进复杂度是 ，但是可以证明其均摊复杂度为 。而在末尾删除元素和访问元素则都仍然是 的开销。 因此，只要对 `vector` 的尺寸估计得当并善用 `resize()` 和 `reserve()`，就能使得 `vector` 的效率与定长数组不会有太大差距。

`vector<bool>`

标准库特别提供了对 `bool` 的 `vector` 特化，每个“`bool`”只占 1 bit，且支持动态增长。但是其 `operator[]` 的返回值的类型不是 `bool&` 而是 `vector<bool>::reference`。因此，使用 `vector<bool>` 使需谨慎，可以考虑使用 `deque<bool>` 或 `vector<char>` 替代。而如果你需要节省空间，请直接使用 **`bitset`**。

`array`(C++11)

`std::array` 是 STL 提供的 **内存连续的、固定长度** 的数组数据结构。其本质是对原生数组的直接封装。

为什么要用 `array`

`array` 实际上是 STL 对数组的封装。它相比 `vector` 牺牲了动态扩容的特性，但是换来了与原生数组几乎一致的性能（在开满优化的前提下）。因此如果能使用 C++11 特性的情况下，能够使用原生数组的地方几乎都可以直接把定长数组都换成 `array`，而动态分配的数组可以替换为 `vector`。

成员函数

隐式定义的成员函数

函数	作用
`operator=`	以来自另一 `array` 的每个元素重写 `array` 的对应元素

元素访问

函数	作用
`at`	访问指定的元素，同时进行越界检查
`operator[]`	访问指定的元素，**不** 进行越界检查
`front`	访问第一个元素
`back`	访问最后一个元素
`data`	返回指向内存中数组第一个元素的指针

`at` 若遇 `pos >= size()` 的情况会抛出 `std::out_of_range`。

容量

函数	作用
`empty`	检查容器是否为空
`size`	返回容纳的元素数
`max_size`	返回可容纳的最大元素数

由于每个 `array` 都是固定大小容器，`size()` 返回的值等于 `max_size()` 返回的值。

操作

函数	作用
`fill`	以指定值填充容器
`swap`	交换内容

注意，交换两个 `array` 是 的，而非与常规 STL 容器一样为 。

非成员函数

函数	作用
`operator==` 等	按照字典序比较 `array` 中的值
`std::get`	访问 `array` 的一个元素
`std::swap`	特化的 `std::swap` 算法

下面是一个 `array` 的使用示例：

```cpp
// 1. 创建空array，长度为3；常数复杂度
std::array<int, 3> v0;
// 2. 用指定常数创建array；常数复杂度
std::array<int, 3> v1{1, 2, 3}; v0.fill(1);
// 填充数组 // 访问数组
for (int i = 0; i != arr.size(); ++i) cout << arr[i] << " ";
```

deque

`std::deque` 是 STL 提供的 **双端队列** 数据结构。能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问。

`deque` 的使用方法

以下介绍常用用法，详细内容 **请参见 C++ 文档**。`deque` 的迭代器函数与 `vector` 相同，因此不作详细介绍。

构造函数

参见如下代码（假设你已经 `using` 了 `std` 命名空间相关类型）：

```cpp
// 1. 定义一个int类型的空双端队列 v0
deque<int> v0;
// 2. 定义一个int类型的双端队列 v1，并设置初始大小为10；线性复杂度
deque<int> v1(10);
// 3. 定义一个int类型的双端队列 v2，并初始化为10个1；线性复杂度
deque<int> v2(10, 1);
// 4. 复制已有的双端队列 v1；线性复杂度
deque<int> v3(v1);
// 5. 创建一个v2的拷贝deque v4，其内容是v4[0]至v4[2]；线性复杂度
deque<int> v4(v2.begin(), v2.begin() + 3);
// 6. 移动v2到新创建的deque v5，不发生拷贝；常数复杂度；需要 C++11
deque<int> v5(std::move(v2));
```

元素访问

与 `vector` 一致，但无法访问底层内存。其高效的元素访问速度可参考实现细节部分。

- `at()` 返回容器中指定位置元素的引用，执行越界检查，**常数复杂度**。
- `operator[]` 返回容器中指定位置元素的引用。不执行越界检查，**常数复杂度**。
- `front()` 返回首元素的引用。
- `back()` 返回末尾元素的引用。

迭代器

与 `vector` 一致。

长度

与 `vector` 一致，但是没有 `reserve()` 和 `capacity()` 函数。（仍然有 `shrink_to_fit()` 函数）

元素增删及修改

与 `vector` 一致，并额外有向队列头部增加元素的函数。

- `clear()` 清除所有元素
- `insert()` 支持在某个迭代器位置插入元素、可以插入多个。**复杂度与 `pos` 与两端距离较小者成线性。**
- `erase()` 删除某个迭代器或者区间的元素，返回最后被删除的迭代器。复杂度与 `insert` 一致。
- `push_front()` 在头部插入一个元素，**常数复杂度**。
- `pop_front()` 删除头部元素，**常数复杂度**。
- `push_back()` 在末尾插入一个元素，**常数复杂度**。
- `pop_back()` 删除末尾元素，**常数复杂度**。
- `swap()` 与另一个容器进行交换，此操作是 **常数复杂度** 而非线性的。

`deque` 的实现细节

`deque` 通常的底层实现是多个不连续的缓冲区，而缓冲区中的内存是连续的。而每个缓冲区还会记录首指针和尾指针，用来标记有效数据的区间。当一个缓冲区填满之后便会在之前或者之后分配新的缓冲区来存储更多的数据。更详细的说明可以参考 **STL 源码剖析——deque 的实现原理和使用方法详解**。

`list`

`std::list` 是 STL 提供的 **双向链表** 数据结构。能够提供线性复杂度的随机访问，以及常数复杂度的插入和删除。

`list` 的使用方法

`list` 的使用方法与 `deque` 基本相同，但是增删操作和访问的复杂度不同。详细内容 **请参见 C++ 文档**。`list` 的迭代器、长度、元素增删及修改相关的函数与 `deque` 相同，因此不作详细介绍。

元素访问

由于 `list` 的实现是链表，因此它不提供随机访问的接口。若需要访问中间元素，则需要使用迭代器。

- `front()` 返回首元素的引用。
- `back()` 返回末尾元素的引用。

操作

`list` 类型还提供了一些针对其特性实现的 STL 算法函数。由于这些算法需要 **随机访问迭代器**，因此 `list` 提供了特别的实现以便于使用。这些算法有 `splice()`、`remove()`、`sort()`、`unique()`、`merge()` 等。

`forward_list`（C++11）

`std::forward_list` 是 STL 提供的 **单向链表** 数据结构，相比于 `std::list` 减小了空间开销。

`forward_list` 的使用方法

`forward_list` 的使用方法与 `list` 几乎一致，但是迭代器只有单向的，因此其具体用法不作详细介绍。详细内容 **请参见 C++ 文档**

关联式容器

`set`

`set` 是关联容器，含有键值类型对象的已排序集，搜索、移除和插入拥有对数复杂度。`set` 内部通常采用红黑树实现。平衡二叉树的特性使得 `set` 非常适合处理需要同时兼顾查找、插入与删除的情况。

和数学中的集合相似，`set` 中不会出现值相同的元素。如果需要有相同元素的集合，需要使用 `multiset`。`multiset` 的使用方法与 `set` 的使用方法基本相同。

插入与删除操作

- `insert(x)` 当容器中没有等价元素的时候，将元素 x 插入到 `set` 中。
- `erase(x)` 删除值为 x 的 **所有** 元素，返回删除元素的个数。
- `erase(pos)` 删除迭代器为 pos 的元素，要求迭代器必须合法。
- `erase(first,last)` 删除迭代器在 范围内的所有元素。
- `clear()` 清空 `set`。

insert 函数的返回值

insert 函数的返回值类型为 pair<iterator, bool>，其中 iterator 是一个指向所插入元素（或者是指向等于所插入值的原本就在容器中的元素）的迭代器，而 bool 则代表元素是否插入成功，由于 set 中的元素具有唯一性质，所以如果在 set 中已有等值元素，则插入会失败，返回 false，否则插入成功，返回 true；map 中的 insert 也是如此。

迭代器

`set` 提供了以下几种迭代器：

1. `begin()/cbegin()`

   返回指向首元素的迭代器，其中 `*begin = front`。

2. `end()/cend()`

   返回指向数组尾端占位符的迭代器，注意是没有元素的。

3. `rbegin()/crbegin()`

   返回指向逆向数组的首元素的逆向迭代器，可以理解为正向容器的末元素。

4. `rend()/crend()`

   返回指向逆向数组末元素后一位置的迭代器，对应容器首的前一个位置，没有元素。

以上列出的迭代器中，含有字符 `c` 的为只读迭代器，你不能通过只读迭代器去修改 `set` 中的元素的值。如果一个 `set` 本身就是只读的，那么它的一般迭代器和只读迭代器完全等价。只读迭代器自 C++11 开始支持。

查找操作

- `count(x)` 返回 `set` 内键为 x 的元素数量。
- `find(x)` 在 `set` 内存在键为 x 的元素时会返回该元素的迭代器，否则返回 `end()`。
- `lower_bound(x)` 返回指向首个不小于给定键的元素的迭代器。如果不存在这样的元素，返回 `end()`。
- `upper_bound(x)` 返回指向首个大于给定键的元素的迭代器。如果不存在这样的元素，返回 `end()`。
- `empty()` 返回容器是否为空。
- `size()` 返回容器内元素个数。

lower_bound 和 upper_bound 的时间复杂度

set 自带的 lower_bound 和 upper_bound 的时间复杂度为 `O(log n)`。

但使用 algorithm 库中的 lower_bound 和 upper_bound 函数对 set 中的元素进行查询，时间复杂度为 `O(n)`。

nth_element 的时间复杂度

set 没有提供自带的 nth_element。使用 algorithm 库中的 nth_element 查找第 k大的元素时间复杂度为 `O(n)`。

如果需要实现平衡二叉树所具备的 `O(log n)` 查找第k大元素的功能，需要自己手写平衡二叉树或权值线段树，或者选择使用 pb_ds 库中的平衡二叉树。

使用样例

`set` 在贪心中的使用

在贪心算法中经常会需要出现类似 **找出并删除最小的大于等于某个值的元素**。这种操作能轻松地通过 `set` 来完成。

```
// 现存可用的元素
set<int> available;
// 需要大于等于的值
int x;

// 查找最小的大于等于x的元素
set<int>::iterator it = available.lower_bound(x);
if (it == available.end()) {
  // 不存在这样的元素，则进行相应操作……
} else {
  // 找到了这样的元素，将其从现存可用元素中移除
  available.erase(it);
  // 进行相应操作……
}
```

`map`

`map` 是有序键值对容器，它的元素的键是唯一的。搜索、移除和插入操作拥有对数复杂度。`map` 通常实现为红黑树。

你可能需要存储一些键值对，例如存储学生姓名对应的分数：`Tom 0`，`Bob 100`，`Alan 100`。但是由于数组下标只能为非负整数，所以无法用姓名作为下标来存储，这个时候最简单的办法就是使用 STL 中的 `map` 了！

`map` 重载了 `operator[]`，可以用任意定义了 `operator <` 的类型作为下标（在 `map` 中叫做 `key`，也就是索引）：

```
map<Key, T> yourMap;
```

其中，`Key` 是键的类型，`T` 是值的类型，下面是使用 `map` 的实例：

```
map<string, int> mp;
```

`map` 中不会存在键相同的元素，`multimap` 中允许多个元素拥有同一键。`multimap` 的使用方法与 `map` 的使用方法基本相同。

Warning

正是因为 `multimap` 允许多个元素拥有同一键的特点，`multimap` 并没有提供给出键访问其对应值的方法。

插入与删除操作

- 可以直接通过下标访问来进行查询或插入操作。例如 `mp["Alan"]=100`。
- 通过向 `map` 中插入一个类型为 `pair<Key, T>` 的值可以达到插入元素的目的，例如 `mp.insert(pair<string,int>("Alan",100));`；
- `erase(key)` 函数会删除键为 `key` 的 **所有** 元素。返回值为删除元素的数量。
- `erase(pos)`：删除迭代器为 pos 的元素，要求迭代器必须合法。
- `erase(first,last)`：删除迭代器在 范围内的所有元素。
- `clear()` 函数会清空整个容器。

下标访问中的注意事项

在利用下标访问 map 中的某个元素时，如果 map 中不存在相应键的元素，会自动在 map 中插入一个新元素，并将其值设置为默认值（对于整数，值为零；对于有默认构造函数的类型，会调用默认构造函数进行初始化）。

当下标访问操作过于频繁时，容器中会出现大量无意义元素，影响 map 的效率。因此一般情况下推荐使用 find() 函数来寻找特定键的元素。

查询操作

- `count(x)`：返回容器内键为 x 的元素数量。复杂度为 （关于容器大小对数复杂度，加上匹配个数）。
- `find(x)`：若容器内存在键为 x 的元素，会返回该元素的迭代器；否则返回 `end()`。
- `lower_bound(x)`：返回指向首个不小于给定键的元素的迭代器。
- `upper_bound(x)`：返回指向首个大于给定键的元素的迭代器。若容器内所有元素均小于或等于给定键，返回 `end()`。
- `empty()`：返回容器是否为空。
- `size()`：返回容器内元素个数。

使用样例

`map` 用于存储复杂状态

在搜索中，我们有时需要存储一些较为复杂的状态（如坐标，无法离散化的数值，字符串等）以及与之有关的答案（如到达此状态的最小步数）。`map` 可以用来实现此功能。其中的键是状态，而值是与之相关的答案。下面的示例展示了如何使用 `map` 存储以 `string` 表示的状态。

```
// 存储状态与对应的答案
map<string, int> record;

// 新搜索到的状态与对应答案
string status;
int ans;
```

```
// 查找对应的状态是否出现过
map<string, int>::iterator it = record.find(status);
if (it == record.end()) {
  // 尚未搜索过该状态，将其加入状态记录中
  record[status] = ans;
  // 进行相应操作……
} else {
  // 已经搜索过该状态，进行相应操作……
}
```

遍历容器

可以利用迭代器来遍历关联式容器的所有元素。

```
set<int> s;
typedef set<int>::iterator si;
for (si it = s.begin(); it != s.end(); it++) cout << *it << endl;
```

需要注意的是，对 `map` 的迭代器解引用后，得到的是类型为 `pair<Key, T>` 的键值对。

在 C++11 中，使用范围 for 循环会让代码简洁很多：

```
set<int> s;
for (auto x : s) cout << x << endl;
```

对于任意关联式容器，使用迭代器遍历容器的时间复杂度均为 `O(n)`。

自定义比较方式

`set` 在默认情况下的比较函数为 `<`（如果是非内置类型需要 **重载 `<` 运算符**）。然而在某些特殊情况下，我们希望能自定义 `set` 内部的比较方式。

这时候可以通过传入自定义比较器来解决问题。

具体来说，我们需要定义一个类，并在这个类中 **重载 `()` 运算符**。

例如，我们想要维护一个存储整数，且较大值靠前的 `set`，可以这样实现：

```
struct cmp {
  bool operator()(int a, int b) { return a > b; }
};
set<int, cmp> s;
```

对于其他关联式容器，可以用类似的方式实现自定义比较，这里不再赘述。

无序关联式容器

概述

自 C++11 标准起，四种基于哈希实现的无序关联式容器正式纳入了 C++ 的标准模板库中，分别是：`unordered_set`，`unordered_multiset`，`unordered_map`，`unordered_multimap`。

它们与相应的关联式容器在功能，函数等方面有诸多共同点，而最大的不同点则体现在普通的关联式容器一般采用红黑树实现，内部元素按特定顺序进行排序；而这几种无序关联式容器则采用哈希方式存储元素，内部元素不以任何特定顺序进行排序，所以访问无序关联式容器中的元素时，访问顺序也没有任何保证。

采用哈希存储的特点使得无序关联式容器 **在平均情况下** 大多数操作（包括查找，插入，删除）都能在常数时间复杂度内完成，相较于关联式容器与容器大小成对数的时间复杂度更加优秀。

Warning

在最坏情况下，对无序关联式容器进行插入、删除、查找等操作的时间复杂度会 **与容器大小成线性关系**！这一情况往往在容器内出现大量哈希冲突时产生。

同时，由于无序关联式容器的操作时通常存在较大的常数，其效率有时并不比普通的关联式容器好太多。

因此应谨慎使用无序关联式容器，尽量避免滥用（例如懒得离散化，直接将 `unordered_map<int, int>` 当作空间无限的普通数组使用）。

由于无序关联式容器与相应的关联式容器在用途和操作中有很多共同点，这里不再介绍无序关联式容器的各种操作，这些内容读者可以参考 **关联式容器**。

制造哈希冲突

上文中提到了，在最坏情况下，对无序关联式容器进行一些操作的时间复杂度会与容器大小成线性关系。

在哈希函数确定的情况下，可以构造出数据使得容器内产生大量哈希冲突，导致复杂度达到上界。

在标准库实现里，每个元素的散列值是将值对一个质数取模得到的，更具体地说，是 **这个列表** 中的质数（g++ 6 及以前版本的编译器，这个质数一般是 ，g++ 7 及之后版本的编译器，这个质数一般是 ）。

因此可以通过向容器中插入这些模数的倍数来达到制造大量哈希冲突的目的。

自定义哈希函数

使用自定义哈希函数可以有效避免构造数据产生的大量哈希冲突。

要想使用自定义哈希函数，需要定义一个结构体，并在结构体中重载 `()` 运算符，像这样：

```
struct my_hash {
  size_t operator()(int x) const { return x; }
};
```

当然，为了确保哈希函数不会被迅速破解（例如 Codeforces 中对使用无序关联式容器的提交进行 hack），可以试着在哈希函数中加入一些随机化函数（如时间）来增加破解的难度。

例如，在 **这篇博客** 中给出了如下哈希函数：

```
struct my_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }

  size_t operator()(uint64_t x) const {
    static const uint64_t FIXED_RANDOM =
        chrono::steady_clock::now().time_since_epoch().count();
    return splitmix64(x + FIXED_RANDOM);
  }

  // 针对 std::pair<int, int> 作为主键类型的哈希函数
  size_t operator()(pair<uint64_t, uint64_t> x) const {
    static const uint64_t FIXED_RANDOM =
        chrono::steady_clock::now().time_since_epoch().count();
    return splitmix64(x.first + FIXED_RANDOM) ^
           (splitmix64(x.second + FIXED_RANDOM) >> 1);
  }
};
```

写完自定义的哈希函数后，就可以通过 `unordered_map<int, int, my_hash> my_map;` 或者 `unordered_map<pair<int, int>, int, my_hash> my_pair_map;` 的定义方式将自定义的哈希函数传入容器了。

容器适配器

栈

STL 栈（`std::stack`）是一种后进先出 (Last In, First Out) 的容器适配器，仅支持查询或删除最后一个加入的元素（栈顶元素），不支持随机访问，且为了保证数据的严格有序性，不支持迭代器。

头文件

```
#include <stack>
```

定义

```
std::stack<TypeName> s;  // 使用默认底层容器 deque，数据类型为 TypeName
std::stack<TypeName, Container> s;  // 使用 Container 作为底层容器
std::stack<TypeName> s2(s1);        // 将 s1 复制一份用于构造 s2
```

成员函数¶

以下所有函数均为常数复杂度¶

- `top()` 访问栈顶元素（如果栈为空，此处会出错）
- `push(x)` 向栈中插入元素 x
- `pop()` 删除栈顶元素
- `size()` 查询容器中的元素数量
- `empty()` 询问容器是否为空

简单示例

```
std::stack<int> s1;
s1.push(2);
s1.push(1);
std::stack<int> s2(s1);
s1.pop();
std::cout << s1.size() << " " << s2.size() << endl;  // 1 2
std::cout << s1.top() << " " << s2.top() << endl;    // 2 1
s1.pop();
std::cout << s1.empty() << " " << s2.empty() << endl;  // 1 0
```

队列

STL 队列 (`std::queue`) 是一种先进先出 (First In, First Out) 的容器适配器，仅支持查询或删除第一个加入的元素（队首元素），不支持随机访问，且为了保证数据的严格有序性，不支持迭代器。

头文件

```
#include <queue>
```

定义

```
std::queue<TypeName> q;  // 使用默认底层容器 deque，数据类型为 TypeName
std::queue<TypeName, Container> q;  // 使用 Container 作为底层容器

std::queue<TypeName> q2(q1);  // 将 s1 复制一份用于构造 q2
```

成员函数

以下所有函数均为常数复杂度

- `front()` 访问队首元素（如果队列为空，此处会出错）
- `push(x)` 向队列中插入元素 x
- `pop()` 删除队首元素
- `size()` 查询容器中的元素数量
- `empty()` 询问容器是否为空

简单示例

```
std::queue<int> q1;
q1.push(2);
q1.push(1);
std::queue<int> q2(q1);
q1.pop();
std::cout << q1.size() << " " << q2.size() << endl;    // 1 2
std::cout << q1.front() << " " << q2.front() << endl;  // 1 2
q1.pop();
std::cout << q1.empty() << " " << q2.empty() << endl;  // 1 0
```

优先队列

头文件

```
#include <queue>
```

定义

```
std::priority_queue<TypeName> q;             // 数据类型为 TypeName
std::priority_queue<TypeName, Container> q;  // 使用 Container 作为底层容器
std::priority_queue<TypeName, Container, Compare> q;
// 使用 Container 作为底层容器，使用 Compare 作为比较类型

// 默认使用底层容器 vector
// 比较类型 less<TypeName> (此时为它的 top() 返回为最大值)
// 若希望 top() 返回最小值，可令比较类型为 greater<TypeName>
// 注意：不可跳过 Container 直接传入 Compare

// 从 C++11 开始，如果使用 lambda 函数自定义 Compare
// 则需要将其作为构造函数的参数代入，如：
auto cmp = [](const std::pair<int, int> &l, const std::pair<int, int> &r) {
  return l.second < r.second;
};
std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int> >,
                    decltype(cmp)>
    pq(cmp);
```

成员函数

以下所有函数均为常数复杂度

- `top()` 访问堆顶元素（此时优先队列不能为空）
- `empty()` 询问容器是否为空
- `size()` 查询容器中的元素数量

以下所有函数均为对数复杂度

- `push(x)` 插入元素，并对底层容器排序
- `pop()` 删除堆顶元素（此时优先队列不能为空）

简单示例

```
std::priority_queue<int> q1;
std::priority_queue<int, std::vector<int> > q2;
// C++11 后空格可省略
std::priority_queue<int, std::deque<int>, std::greater<int> > q3;
// q3 为小根堆
for (int i = 1; i <= 5; i++) q1.push(i);
// q1 中元素 :  [1, 2, 3, 4, 5]
std::cout << q1.top() << std::endl;
// 输出结果 : 5
q1.pop();
// 堆中元素 : [1, 2, 3, 4]
std::cout << q1.size() << std::endl;
// 输出结果 : 4
for (int i = 1; i <= 5; i++) q3.push(i);
// q3 中元素 :  [1, 2, 3, 4, 5]
std::cout << q3.top() << std::endl;
// 输出结果 : 1
```

STL算法

STL 提供了大约 100 个实现算法的模版函数，基本都包含在 `<algorithm>` 之中，还有一部分包含在 `<numeric>` 和 `<functional>`。完备的函数列表请 **参见参考手册**，排序相关的可以参考 **排序内容的对应页面**。

- `find`：顺序查找。`find(v.begin(), v.end(), value)`，其中 `value` 为需要查找的值。
- `find_end`：逆序查找。`find_end(v.begin(), v.end(), value)`。
- `reverse`：翻转数组、字符串。`reverse(v.begin(), v.end())` 或 `reverse(a + begin, a + end)`。
- `unique`：去除容器中相邻的重复元素。`unique(ForwardIterator first, ForwardIterator last)`，返回值为指向 **去重后** 容器结尾的迭代器，原容器大小不变。与 `sort` 结合使用可以实现完整容器

去重。

- `random_shuffle`：随机地打乱数组。`random_shuffle(v.begin(), v.end())` 或 `random_shuffle(v + begin, v + end)`。
- `sort`：排序。`sort(v.begin(), v.end(), cmp)` 或 `sort(a + begin, a + end, cmp)`，其中 `end` 是排序的数组最后一个元素的后一位，`cmp` 为自定义的比较函数。
- `stable_sort`：稳定排序，用法同 `sort()`。
- `nth_element`：按指定范围进行分类，即找出序列中第 大的元素，使其左边均为小于它的数，右边均为大于它的数。`nth_element(v.begin(), v.begin() + mid, v.end(), cmp)` 或 `nth_element(a + begin, a + begin + mid, a + end, cmp)`。
- `binary_search`：二分查找。`binary_search(v.begin(), v.end(), value)`，其中 `value` 为需要查找的值。
- `merge`：将两个（已排序的）序列 **有序合并** 到第三个序列的 **插入迭代器** 上。`merge(v1.begin(), v1.end(), v2.begin(), v2.end() ,back_inserter(v3))`。
- `inplace_merge`：将两个（已按小于运算符排序的）：`[first,middle), [middle,last)` 范围 **原地合并为一个有序序列**。`inplace_merge(v.begin(), v.begin() + middle, v.end())`。
- `lower_bound`：在一个有序序列中进行二分查找，返回指向第一个 **大于等于** 的元素的位置的迭代器。如果不存在这样的元素，则返回尾迭代器。`lower_bound(v.begin(),v.end(),x)`。
- `upper_bound`：在一个有序序列中进行二分查找，返回指向第一个 **大于** 的元素的位置的迭代器。如果不存在这样的元素，则返回尾迭代器。`upper_bound(v.begin(),v.end(),x)`。
- `next_permutation`：将当前排列更改为 **全排列中的下一个排列**。如果当前排列已经是 **全排列中的最后一个排列**（元素完全从大到小排列），函数返回 `false` 并将排列更改为 **全排列中的第一个排列**（元素完全从小到大排列）；否则，函数返回 `true`。`next_permutation(v.begin(), v.end())` 或 `next_permutation(v + begin, v + end)`。
- `partial_sum`：求前缀和。设源容器为 ，目标容器为 ，则令 。`partial_sum(src.begin(), src.end(), back_inserter(dst))`。

使用样例

- 使用 `next_permutation` 生成 1 到9的全排列。例题：**Luogu P1706 全排列问题**

```
int N = 9, a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
do {
  for (int i = 0; i < N; i++) cout << a[i] << " ";
  cout << endl;
} while (next_permutation(a, a + N));
```

- 使用 `lower_bound` 与 `upper_bound` 查找有序数组a中小于x，等于x，大于x元素的分界线。

```
int N = 10, a[] = {1, 1, 2, 4, 5, 5, 7, 7, 9, 9}, x = 5;
int i = lower_bound(a, a + N, x) - a, j = upper_bound(a, a + N, x) - a;
// a[0] ~ a[i - 1] 为小于x的元素， a[i] ~ a[j - 1] 为等于x的元素， a[j] ~ a[N -
// 1] 为大于x的元素
cout << i << " " << j << endl;
```

- 使用 `partial_sum` 求解src中元素的前缀和，并存储于dst中。

```
vector<int> src = {1, 2, 3, 4, 5}, dst;
// 求解src中元素的前缀和，dst[i] = src[0] + ... + src[i]
// back_inserter 函数作用在 dst 容器上，提供一个迭代器
partial_sum(src.begin(), src.end(), back_inserter(dst));
for (unsigned int i = 0; i < dst.size(); i++) cout << dst[i] << " ";
```

- 使用 `lower_bound` 查找有序数组a中最接近x的元素。例题：**UVA10487 Closest Sums**

```
int N = 10, a[] = {1, 1, 2, 4, 5, 5, 8, 8, 9, 9}, x = 6;
// lower_bound将返回a中第一个大于等于x的元素的地址，计算出的i为其下标
int i = lower_bound(a, a + N, x) - a;
// 在以下两种情况下，a[i]（a中第一个大于等于x的元素）即为答案：
// 1. a中最小的元素都大于等于x；
// 2. a中存在大于等于x的元素，且第一个大于等于x的元素（a[i]）
// 相比于第一个小于x的元素（a[i - 1]）更接近x；
// 否则，a[i - 1]（a中第一个小于x的元素）即为答案
if (i == 0 || (i < N && a[i] - x < x - a[i - 1]))
  cout << a[i];
else
  cout << a[i - 1];
```

- 使用 `sort` 与 `unique` 查找数组 中 **第 小的值**（注意：重复出现的值仅算一次，因此本题不是求解第 小的元素）。例题：**Luogu P1138 第 k 小整数**

```
int N = 10, a[] = {1, 3, 3, 7, 2, 5, 1, 2, 4, 6}, k = 3;
sort(a, a + N);
// unique将返回去重之后数组最后一个元素之后的地址，计算出的cnt为去重后数组的长度
int cnt = unique(a, a + N) - a;
cout << a[k - 1];
```

bitset

介绍¶

`std::bitset` 是标准库中的一个存储 `0/1` 的大小不可变容器。严格来讲，它并不属于 STL。

由于内存地址是按字节即 `byte` 寻址，而非比特 `bit`，一个 `bool` 类型的变量，虽然只能表示 `0/1`，但是也占了 1 byte 的内存。

`bitset` 就是通过固定的优化，使得一个字节的八个比特能分别储存 8 位的 `0/1`。

对于一个 4 字节的 `int` 变量，在只存 `0/1` 的意义下，`bitset` 占用空间只是其 ，计算一些信息时，所需时间也是其 。

在某些情况下通过 `bitset` 可以优化程序的运行效率。至于其优化的是复杂度还是常数，要看计算复杂度的角度。一般 `bitset` 的复杂度有以下几种记法：（设原复杂度为 ）

1. ，这种记法认为 `bitset` 完全没有优化复杂度。
2. ，这种记法不太严谨（复杂度中不应出现常数），但体现了 `bitset` 能将所需时间优化至 。
3. ，其中 （计算机的位数），这种记法较为普遍接受。
4. ，其中 为计算机一个整型变量的大小。

当然， `vector` 的一个特化 `vector<bool>` 的储存方式同 `bitset` 一样，区别在于其支持动态开空间，`bitset` 则和我们一般的静态数组一样，是在编译时就开好了的。

然而，`bitset` 有一些好用的库函数，不仅方便，而且有时可以避免使用 for 循环而没有实质的速度优化。因此，一般不使用 `vector<bool>`。

使用

头文件

```
#include <bitset>
```

指定大小

```
bitset<1000> bs;  // a bitset with 1000 bits
```

构造函数

- `bitset()`：每一位都是 `false`。
- `bitset(unsigned long val)`：设为 `val` 的二进制形式。
- `bitset(const string& str)`：设为 串 `str`。

运算符

- `operator []`：访问其特定的一位。
- `operator ==/!=`：比较两个 `bitset` 内容是否完全一样。
- `operator &/&=/|/| =/^/^=/~`：进行按位与/或/异或/取反操作。**`bitset` 只能与 `bitset` 进行位运算**，若要和整型进行位运算，要先将整型转换为 `bitset`。
- `operator <>/<<=/>>=`：进行二进制左移/右移。
- `operator <>`：流运算符，这意味着你可以通过 `cin/cout` 进行输入输出。

成员函数

- `count()`：返回 `true` 的数量。
- `size()`：返回 `bitset` 的大小。
- `test(pos)`：它和 `vector` 中的 `at()` 的作用是一样的，和 `[]` 运算符的区别就是越界检查。
- `any()`：若存在某一位是 `true` 则返回 `true`，否则返回 `false`。
- `none()`：若所有位都是 `false` 则返回 `true`，否则返回 `false`。
- `all()`：**C++11**，若所有位都是 `true` 则返回 `true`，否则返回 `false`。
- 1. `set()`：将整个 `bitset` 设置成 `true`。
  2. `set(pos, val = true)`：将某一位设置成 `true` / `false`。
- 1. `reset()`：将整个 `bitset` 设置成 `false`。
  2. `reset(pos)`：将某一位设置成 `false`。相当于 `set(pos, false)`。
- 1. `flip()`：翻转每一位。（，相当于异或一个全是 的 `bitset`）
  2. `flip(pos)`：翻转某一位。
- `to_string()`：返回转换成的字符串表达。
- `to_ulong()`：返回转换成的 `unsigned long` 表达（`long` 在 NT 及 32 位 POSIX 系统下与 `int` 一样，在 64 位 POSIX 下与 `long long` 一样）。
- `to_ullong()`：**C++11**，返回转换成的 `unsigned long long` 表达。

一些文档中没有的成员函数：

- `_Find_first()`：返回 `bitset` 第一个 `true` 的下标，若没有 `true` 则返回 `bitset` 的大小。
- `_Find_next(pos)`：返回 `pos` 后面（下标严格大于 `pos` 的位置）第一个 `true` 的下标，若 `pos` 后面没有 `true` 则返回 `bitset` 的大小。

string

`string` 是什么

`std::string` 是在标准库 `<string>`（注意不是 C 语言中的 `<string.h>` 库）中提供的一个类，本质上是 `std::basic_string<char>` 的别称。

为什么要使用 `string`

在 C 语言中，提供了字符串的操作，但只能通过字符数组的方式来实现字符串。而 `string` 则是一个简单的类，使用简单，在 OI 竞赛中被广泛使用。并且相较于其他 STL 容器，`string` 的常数可以算是非常优秀的，基本与字符数组不相上下。

`string` 可以动态分配空间

和许多 STL 容器相同，`string` 能动态分配空间，这使得我们可以直接使用 `std::cin` 来输入，但其速度则同样较慢。这一点也同样让我们不必为内存而烦恼。

`string` 重载了加法运算符和比较运算符

`string` 的加法运算符可以直接拼接两个字符串或一个字符串和一个字符。和 `std::vector` 类似，`string` 重载了比较运算符，同样是按字典序比较的，所以我们可以直接调用 `std::sort` 对若干字符串进行排序。

使用方法

下面介绍 `string` 的基本操作，具体可看 **C++ 文档**。

声明

```
std::string s;
```

转 char 数组

在 C 语言里，也有很多字符串的函数，但是它们的参数都是 char 指针类型的，为了方便使用，`string` 有两个成员函数能够将自己转换为 char 指针——`data()` / `c_str()`（它们几乎是一样的，但最好使用 `c_str()`，因为 `c_str()` 保证末尾有空字符，而 `data()` 则不保证），如：

```
printf("%s", s);          // 编译错误
printf("%s", s.data());   // 编译通过，但是是 undefined behavior
printf("%s", s.c_str());  // 一定能够正确输出
```

获取长度¶

很多函数都可以返回 string 的长度：

```
printf("s 的长度为 %d", s.size());
printf("s 的长度为 %d", s.length());
printf("s 的长度为 %d", strlen(s.c_str()));
```

寻找某字符（串）第一次出现的位置

```
printf("字符 a 在 s 的 %d 位置第一次出现", s.find('a'));
printf("字符串 t 在 s 的 %d 位置第一次出现", s.find(t));
printf("在 s 中自 pos 位置起字符串 t 第一次出现在 %d 位置", s.find(t, pos));
```

截取子串

`substr(pos, len)`，这个函数的参数是从 `pos` 位置开始截取最多 `len` 个字符（如果从 `pos` 开始的后缀长度不足 `len` 则截取这个后缀）。

```
printf("从这个字符串的第二位开始的最多三个字符构成的子串是 %s",
       s.substr(1, 3).c_str());
```

pair

`std::pair` 是标准库中定义的一个类模板。用于将两个变量关联在一起，组成一个“对”，而且两个变量的数据类型可以是不同的。

使用

初始化

可以在定义时直接完成 `pair` 的初始化。

```
pair<int, double> p0(1, 2.0);
```

也可以使用先定义，后赋值的方法完成 `pair` 的初始化。

```
pair<int, double> p1;
p1.first = 1;
p1.second = 2.0;
```

还可以使用 `std::make_pair` 函数。该函数接受两个变量，并返回由这两个变量组成的 `pair`。

```
pair<int, double> p2 = make_pair(1, 2.0);
```

一种常用的方法是使用宏定义 `#define mp make_pair`，将有些冗长的 `make_pair` 化简为 `mp`。

在 C++11 以及之后的版本中，`make_pair` 可以配合 `auto` 使用，以避免显式声明数据类型。

```
auto p3 = make_pair(1, 2.0);
```

访问

通过成员函数 `first` 与 `second`，可以访问 `pair` 中包含的两个变量。

```
int i = p0.first;
double d = p0.second;
```

也可以对其进行修改。

```
p1.first++;
```

比较

`pair` 已经预先定义了所有的比较运算符，包括 `<`、`>`、`<=`、`>=`、`==`、`!=`。当然，这需要组成 `pair` 的两个变量所属的数据类型定义了 `==` 和/或 `<` 运算符。

其中，`<`、`>`、`<=`、`>=` 四个运算符会先比较两个 `pair` 中的第一个变量，在第一个变量相等的情况下再比较第二个变量

```
if (p2 >= p3) {
  cout << "do something here" << endl;
}
```

由于 `pair` 定义了 STL 中常用的 `<` 与 `==`，使得其能够很好的与其他 STL 函数或数据结构配合。比如，`pair` 可以作为 `priority_queue` 的数据类型。

```
priority_queue<pair<int, double> > q;
```

赋值与交换

可以将 `pair` 的值赋给另一个类型一致的 `pair`。

```
p0 = p1;
```

也可以使用 `swap` 函数交换 `pair` 的值。

```
swap(p0, p1);
p2.swap(p3);
```

应用举例

离散化

`pair` 可以轻松实现离散化。

我们可以创建一个 `pair` 数组，将原始数据的值作为每个 `pair` 第一个变量，将原始数据的位置作为第二个变量。在排序后，将原始数据值的排名（该值排序后所在的位置）赋给该值原本所在的位置即可。

```
// a为原始数据
```

```
pair<int, int> a[MAXN];
// ai为离散化后的数据
int ai[MAXN];
for (int i = 0; i < n; i++) {
  // first为原始数据的值，second为原始数据的位置
  scanf("%d", &a[i].first);
  a[i].second = i;
}
// 排序
sort(a, a + n);
for (int i = 0; i < n; i++) {
  // 将该值的排名赋给该值原本所在的位置
  ai[a[i].second] = i;
}
```

Dijkstra

如前所述，`pair` 可以作为 `priority_queue` 的数据类型。

那么，在 Dijkstra 算法的堆优化中，可以使用 `pair` 与 `priority_queue` 维护节点，将节点当前到起点的距离作为第一个变量，将节点编号作为第二个变量。

```
priority_queue<pair<int, int>, std::vector<pair<int, int> >,
               std::greater<pair<int, int> > >
    q;
... while (!q.empty()) {
  // dis为入堆时节点到起点的距离，i为节点编号
  int dis = q.top().first, i = q.top().second;
  q.pop();
  ...
}
```

pair 与 map

`map` 的是 C++ 中存储键值对的数据结构。很多情况下，`map` 中存储的键值对通过 `pair` 向外暴露。

```
map<int, double> m;
m.insert(make_pair(1, 2.0));
```

> HUST
>
> 算法必会模板
>
> 必会！

佚名
2021-11-26

目录

省 选 模 板

1 数学知识

//组合数

```
//C(n,m) 在 n 个数中选 m 个的方案数
ll C[N][N];
void get_C(int n)
{
        for(int i=1;i<=n;i++)
        {
                C[i][i]=C[i][0]=1;
                for(int j=1;j<i;j++)
                        C[i][j]=(C[i-1][j]+C[i-1][j-1])%mod;
        }
}
```

//欧几里得算法

```
//(a,b)
ll gcd(ll a,ll b)
{
        return b==0? a:gcd(b,a%b);
}
```

//拓展欧几里得算法

```
//解同余方程 a*x+b*y = (a,b)
ll exgcd(ll a,ll b,ll& d,ll& x,ll& y)
{
        if(!b) { d=a; x=1; y=0; }
        else { exgcd(b,a%b,d,y,x); y-=x*(a/b); }
}
```

//逆元

```
//a*inv(a,n) = 1 mod n
ll inv(ll a,ll n)
{
        ll d,x,y;
        exgcd(a,n,d,x,y);
        return d==1? (x+n)%n:-1;
}
```

//lucas 定理

```
//计算较大的组合数
ll fac[N];
void get_pre(int n)
{
        for(int i=1;i<=n;i++)
                fac[i]=(fac[i-1]*i)%mod;
```

```
}
ll C(ll n,ll m,ll mod)
{
        if(n<m) return 0;
        if(n<mod&&m<mod)
                return fac[n]*inv(fac[m],mod)%mod*inv(fac[n-m],mod)%mod;
        return C(n/mod,m/mod,mod)*C(n%mod,m%mod,mod)%mod;
}
```

//快速幂

```
//a^p % mod
ll pow(ll a,ll p,ll mod)
{
        ll ans=1;
        while(p)
        {
                if(p&1) ans=(ans*a)%mod;
                a=(a*a)%mod; p>>=1;
        }
        return ans;
}
```

//中国剩余定理

```
//解线性同余方程组
//sigma{ ai*(1-ai*mi) } % M  , ai*mi+wi*y=1
ll a[N],m[N];
ll china(int n)
{
        ll M=1,d,x=0,y;
        for(int i=1;i<=n;i++) M*=m[i];
        for(int i=1;i<=n;i++)
        {
                ll w=M/m[i];
                exgcd(m[i],w,d,d,y);
                x=(x+y*w*a[i])%M;
        }
        return (x+M)%M;
}
```

//大步小步算法

```
//计算 a^x=b mod n 中的最小 x
map<int,int> mp;
int BSGS(int a,int b,int n)
{
        int m=sqrt(n)+1,e=1,i;
```

```
        int v=inv(pow(a,m,n),n);
        mp[e]=0;
        for(i=1;i<m;i++)
        {
                e=(e*m)%n;
                if(!mp.count(e)) mp[e]=i;
        }
        for(i=0;i<m;i++)
        {
                if(mp.count(b)) return i*m+mp[b];
                b=(b*v)%mod;
        }
        return -1;
}
```

//快速筛法求素数表

```
int su[N],vis[N];
void get_su(int n)
{
        for(int i=2;i<=n;i++)
        {
                if(!vis[i]) su[++su[0]]=i;
                for(int j=1;j<=su[0]&&i*su[j]<=n;j++)
                {
                        vis[i*su[j]]=1;
                        if(i%su[j]==0) break;
                }
        }
}
```

//欧拉函数

```
//phi(n)小于 n 的数中与 n 互素的数的个数
ll get_phi(int n)
{
        int m=sqrt(n)+1;
        ll ans=n;
        for(int i=2;i<=m;i++) if(n%i==0)
        {
                ans=ans/i*(i-1);
                while(n%i==0) n/=i;
        }
        if(n>1) ans=ans/n*(n-1);
        return ans;
}
ll phi[N];
```

```
void get_phi_table(int n)
{
	phi[1]=1;
	for(int i=2;i<=n;i++) if(!phi[i])
	{
		for(int j=i;j<=n;j+=i)
		{
			if(!phi[j]) phi[j]=j;
			phi[j]=phi[j]/i*(i-1);
		}
	}
}
```

//莫比乌斯函数

```
int mu[N],su[N],vis[N];
void get_mu(int n)
{
	mu[1]=1;
	for(int i=2;i<=n;i++)
	{
		if(!vis[i]) mu[i]=-1,su[++su[0]]=i;
		for(int j=1;j<=su[0]&&i*su[j]<=n;j++)
		{
			vis[i*su[j]]=1;
			if(i%su[j]==0) mu[i*su[j]]=0;
			else mu[i*su[j]]=-mu[i];
		}
	}
}
```

//高斯消元

```
//解线性方程组
double a[N][N];
void gause(int n)
{
	for(int i=1;i<=n;i++)
	{
		int r=i;
		for(int j=i+1;j<=n;j++)
			if(fabs(a[j][i])>fabs(a[r][i])) r=i;
		for(int j=1;j<=n+1;j++) swap(a[i][j],a[r][j]);
		for(int j=n+1;j>=i;j--)
			for(int k=i+1;k<=n;k++)
				a[k][j]-=a[k][i]/a[i][i]*a[i][j];
	}
```

```
        for(int i=n;i;i--)
        {
                for(int j=i+1;j<=n;j++)
                        a[i][n+1]-=a[j][n+1]*a[i][j];
                a[i][n+1]/=a[i][i];
        }
}
```

//高精度

```
//高精度
const int maxn = 1e3+10;
const int base = 1e8;
const int wlen = 8;

int trans(char* s,int st,int ed)
{
        int x=0;
        for(int i=st;i<ed;i++) x=x*10+s[i]- '0 ';
        return x;
}

struct Bign
{
        int len; ll N[maxn];
        Bign() { len=1; memset(N,0,sizeof(N)); }
        Bign(ll num) {  *this=num; }
        Bign(const char* s) { *this=s; }
        void print()
        {
                printf("%d",N[len-1]);
                for(int i=len-2;i>=0;i--)
                        printf("%08d",N[i]);
                puts("");
        }
        Bign operator = (const ll x)
        {
                ll num=x;
                while(num>base)
                {
                        N[len++]=num%base;
                        num/=base;
                }
                N[len++]=num;
                return *this;
```

```
        }
        Bign operator = (char* s)
        {
                int L=strlen(s);
                len=(L-1)/wlen+1;
                for(int i=0;i<len;i++)
                {
                        int ed=L-i*wlen;
                        int st=max(0,ed-wlen);
                        N[i]=trans(s,st,ed);
                }
                return *this;
        }
        bool operator < (const Bign& B) const
        {
                if(len!=B.len) return len<B.len;
                for(int i=len-1;i>=0;i--)
                        if(N[i]!=B.N[i]) return N[i]<B.N[i];
                return 0;
        }
        bool operator <= (const Bign& B) const
        {
                return !(B<(*this));
        }

        void clear()
        {
                while(len>1&&N[len-1]==0) len--;
        }

        Bign operator + (const Bign& B) const
        {
                Bign C;
                C.len=max(len,B.len)+10;
                for(int i=0;i<C.len;i++)
                {
                        C.N[i]+=N[i]+B.N[i];
                        C.N[i+1]+=C.N[i]/base;
                        C.N[i]%=base;
                }
                C.clear();
                return C;
        }
        Bign operator - (Bign B)
```

```
	{
		Bign C;
		C.len=max(len,B.len);
		for(int i=0;i<C.len;i++)
		{
			if(N[i]<B.N[i]) {
				N[i+1]--;
				N[i]+=base;
			}
			C.N[i]=N[i]-B.N[i];
		}
		C.clear();
		return C;
	}
	Bign operator * (const Bign& B) const
	{
		Bign C;
		C.len=len+B.len;
		for(int i=0;i<len;i++)
			for(int j=0;j<B.len;j++)
				C.N[i+j]+=N[i]*B.N[j];
		for(int i=0;i<C.len;i++)
		{
			C.N[i+1]+=C.N[i]/base;
			C.N[i]%=base;
		}
		C.clear();
		return C;
	}
	Bign operator / (const Bign& B)
	{
		Bign C,F;
		C.len=len;
		for(int i=len-1;i>=0;i--)
		{
			F=F*base;
			F.N[0]=N[i];
			while(B<=F)
			{
				F=F-B;
				C.N[i]++;
			}
		}
		C.clear();
```

```
                return C;
        }
        Bign operator % (const Bign& B)
        {
                Bign r=*this/B;
                return *this-r*B;
        }

}A,B;
```

//矩阵乘法

```
//矩阵乘法
struct Mat
{
        int r,c; ll N[maxn][maxn];
        Mat(int r=0,int c=0)
        {
                this->r=r,this->c=c;
                memset(N,0,sizeof(N));
        }
        Mat operator * (const Mat& B) const
        {
                Mat C(r,B.c);
                for(int i=0;i<r;i++)
                        for(int j=0;j<B.c;j++)
                                for(int k=0;k<c;k++)

        C.N[i][j]=(C.N[i][j]+N[i][k]*B.N[k][j])%mod;
                return C;
        }
        Mat operator ^ (int p)
        {
                Mat ans(r,r),tmp=*this;
                for(int i=0;i<r;i++) ans.N[i][i]=1;
                while(p)
                {
                        if(p&1) ans=ans*tmp;
                        tmp=tmp*tmp; p>>=1;
                }
                return ans;
        }

};
```

2. 数据结构

//树状数组

```
int C[N],mx;
void Add(int x,int v)
{
    for(int i=x;i<=mx;i+=i&-i) C[i]+=v;
}
int query(int x)
{
    int ans=0;
    for(int i=x;i;i-=i&-i) ans+=C[i];
    return ans;
}
```

//线段树

```
//区间加,区间乘,区间求和
int mod;
struct Tnode
{
        int u,l,r;
        ll sum,add,mul;
        void mulv(ll x) {
                sum=(sum*x)%mod;
                mul=(mul*x)%mod;
                add=(add*x)%mod;
        }
        void addv(ll x) {
                sum=(sum+(r-l+1)*x%mod)%mod;
                add=(add+x)%mod;
        }
        void pushdown() ;
        void maintain() ;
}T[N];
void Tnode::pushdown() {
        if(mul^1) {
                T[u<<1].mulv(mul);
                T[u<<1|1].mulv(mul);
                mul=1;
        }
        if(add) {
                T[u<<1].addv(add);
                T[u<<1|1].addv(add);
```

```
                add=0;
        }
}
void Tnode::maintain() {
        sum=(T[u<<1].sum+T[u<<1|1].sum)%mod;
}

void update(int u,int L,int R,int x,int f)
{
        T[u].pushdown();
        if(L<=T[u].l&&T[u].r<=R) {
                if(!f) T[u].addv(x);
                else T[u].mulv(x);
        } else {
                int mid=T[u].l+T[u].r>>1;
                if(L<=mid) update(u<<1,L,R,x,f);
                if(mid<R) update(u<<1|1,L,R,x,f);
                T[u].maintain();
        }
}
ll query(int u,int L,int R)
{
        T[u].pushdown();
        if(L<=T[u].l&&T[u].r<=R)
                return T[u].sum;
        else {
                int mid=T[u].l+T[u].r>>1;
                ll ans=0;
                if(L<=mid) ans=(ans+query(u<<1,L,R))%mod;
                if(mid<R) ans=(ans+query(u<<1|1,L,R))%mod;
                return ans;
        }
}
ll a[N];

void build(int u,int l,int r)
{
        T[u]=(Tnode){ u,l,r,0,0,1 };
        if(l==r) {
                T[u].sum=a[l];
        } else {
                int mid=l+r>>1;
                build(u<<1,l,mid);
                build(u<<1|1,mid+1,r);
```

```
                T[u].maintain();
        }
}
```

//Treap

```
struct Node
{
        Node *ch[2];
        int v,r,m,w,s;
        Node(int v):v(v) { ch[0]=ch[1]=NULL; r=rand(); s=w=1; }
        int cmp(int x) {
                if(v==x) return -1;
                return x<v? 0:1;
        }
        void maintain() {
                s=w;
                if(ch[0]!=NULL) s+=ch[0]->s;
                if(ch[1]!=NULL) s+=ch[1]->s;
        }
};

void rotate(Node* &o,int d)
{
        Node* k=o->ch[d^1];o->ch[d^1]=k->ch[d];k->ch[d]=o;
        o->maintain(); k->maintain(); o=k;
}
void insert(Node *&o,int x)
{
        if(o==NULL) o=new Node(x);
        int d=o->cmp(x);
        if(d==-1) o->w++;
        else {
                insert(o->ch[d],x);
                if(o->ch[d]->r > o->r) rotate(o,d^1);
        }
        o->maintain();
}
void remove(Node *&o,int x)
{
        int d=o->cmp(x);
        if(d==-1)
        {
                if(o->s>1) { o->w--; o->maintain(); return ; }
                else {
```

```
                        if(o->ch[0]!=NULL&&o->ch[1]!=NULL) {
                                int d2=o->ch[0]->r > o->ch[1]->r ? 1:0;
                                rotate(o,d2); remove(o->ch[d2],x);
                        } else {
                                if(o->ch[0]!=NULL) o=o->ch[0]; else
o=o->ch[1];
                                delete o;
                        }
                }
        } else
                remove(o->ch[d],x);
        if(o!=NULL) o->maintain();
}
int kth(Node* o,int rk)
{
        if(o==NULL) return 0;
        int s=o->ch[0]==NULL? 0:o->ch[0]->s;
        if(rk==s+1) return o->v;
        else if(rk<=s) return kth(o->ch[0],rk);
        else return kth(o->ch[1],rk-s-o->w);
}
int rank(Node* o,int x)
{
        if(o==NULL) return 0;
        int s=o->ch[0]==NULL? 0:o->ch[0]->s;
        int d=o->cmp(x);
        if(d==-1) return 1;
        else if(d==0) return rank(o->ch[0],x);
        else return s+o->w+rank(o->ch[1],x);
}
int tmp;
void before(Node* o,int x)
{
        if(o==NULL) return ;
        if(o->v<x) { tmp=max(tmp,o->v); before(o->ch[1],x); }
        else before(o->ch[0],x);
}
void after(Node* o,int x)
{
        if(o==NULL) return ;
        if(o->v>x) { tmp=min(tmp,o->v); after(o->ch[0],x); }
        else after(o->ch[1],x);
}
```

//splay 自上而下

```
struct Node
{
	Node *ch[2];
	int s;
	int cmp(int x)
	{
		int d=x-ch[0]->s;
		if(d==1) return -1;
		return d<=0? 0:1;
	}
	void maintain()
	{
		s=ch[0]->s+ch[1]->s;
	}
	void pushdown() {}
}mempool[N],*G=mempool;

Node* null=new Node();
void rotate(Node* &o,int d)
{
	Node* k=o->ch[d^1]; o->ch[d^1]=k->ch[d],k->ch[d]=o;
	o->maintain(); k->maintain(); o=k;
}
void splay(Node* &o,int k)
{
	o->pushdown();
	int d=o->cmp(k);
	if(d==1) k-=o->ch[0]->s+1;
	if(d!=-1) {
		Node* p=o->ch[d];
		p->pushdown();
		int d2=p->cmp(k),k2=d2==0? k:k-p->ch[d]->s-1;
		if(d2!=-1) {
			splay(p->ch[d2],k2);
			if(d==d2) rotate(o,d^1); else rotate(o->ch[d],d);
		}
		rotate(o,d^1);
	}
}
Node* merge(Node* left,Node* right)
{
	splay(left,left->s);
	left->ch[1]=right,left->maintain();
```

```
        return left;
}
void split(Node* o,int k,Node*&left,Node*&right)
{
        splay(o,k);
        left=o,right=left->ch[1],left->ch[1]=NULL;
        left->maintain();
}
Node* build(int l,int r)
{
        if(r<l) return null;
        int mid=l+r>>1;
        G->s=1;
        G->ch[0]=build(l,mid-1);
        G->ch[1]=build(mid+1,r);
        G->maintain();
        return G++;
}
```

//主席树

```
struct Tnode
{
        Tnode *ls,*rs;
        int sum;
} *T[N*50],mempool[N*50],*G=mempool;

Tnode* Nw(Tnode* l,Tnode* r,int x)
{
        G->ls=l,G->rs=r,G->sum=x;
        return G++;
}
Tnode* build(Tnode* p,int l,int r,int pos)
{
        if(l==r)
                return Nw(T[0],T[0],p->sum+1);
        else {
                int mid=l+r>>1;
                if(pos<=mid) Nw(build(p->ls,l,mid,pos),p->rs,p->sum+1);
                else return Nw(p->ls,build(p->rs,mid+1,r,pos),p->sum+1);
        }
}
int query(Tnode* x,int l,int r,int pos)
{
        if(l==r) return x->sum;
        else {
```

```
                int mid=l+r>>1;
                if(pos<=mid) return query(x->ls,l,mid,pos);
                else return query(x->rs,mid+1,r,pos);
        }
}
```

//Link-Cut-Tree

```
namespace LCT
{

        struct Node {
                Node *ch[2],*fa;
                int rev;
                //others v
                Node() {};
                Node(int x) ;
                void reverse() {
                        swap(ch[0],ch[1]);
                        rev^=1;
                }
                void up_push() {
                        if(fa->ch[0]==this||fa->ch[1]==this)
                                fa->up_push();
                        if(rev) {
                                ch[0]->reverse();
                                ch[1]->reverse();
                                rev=0;
                        }
                }
                void maintain() { }
        } T[N<<1],*null=&T[0];
        Node::Node(int x) {
                ch[0]=ch[1]=fa=null;
                rev=0; //v=x;
        }
        void rot(Node* o,int d) {
                Node* p=o->fa;
                p->ch[d]=o->ch[d^1];
                o->ch[d^1]->fa=p;
                o->ch[d^1]=p;
                o->fa=p->fa;
                if(p==p->fa->ch[0])
                        p->fa->ch[0]=o;
                else if(p==p->fa->ch[1])
                        p->fa->ch[1]=o;
```

```
        p->fa=o;
        p->maintain();
    }
    void splay(Node* o) {
        o->up_push();
        Node *nf,*nff;
        while(o->fa->ch[0]==o||o->fa->ch[1]==o) {
            nf=o->fa,nff=nf->fa;
            if(o==nf->ch[0]) {
                if(nf==nff->ch[0]) rot(nf,0);
                rot(o,0);
            } else {
                if(nf==nff->ch[1]) rot(nf,1);
                rot(o,1);
            }
        }
        o->maintain();
    }
    void Access(Node* o) {
        Node *son=null;
        while(o!=null) {
            splay(o);
            o->ch[1]=son;
            o->maintain();
            son=o; o=o->fa;
        }
    }
    void evert(Node* o) {
        Access(o);
        splay(o);
        o->reverse();
    }
    void Link(Node* u,Node* v) {
        evert(u);
        u->fa=v;
    }
    void Cut(Node* u,Node* v) {
        evert(u);
        Access(v),splay(v);
        v->ch[0]=u->fa=null;
        v->maintain();
    }
    Node* find(Node* o) {
        while(o->fa!=null) o=o->fa;
```

```
                return o;
        }

}
using namespace LCT ;
```

3 图

//2-sat

```
struct TwoSAT {
        int n;
        vector<int> g[N<<1];
        int st[N<<1],mark[N<<1],top;

        bool dfs(int x) {
                if(mark[x^1]) return 0;
                if(mark[x]) return 1;
                mark[x]=1;
                st[++top]=x;
                for(int i=0;i<g[x].size();i++)
                        if(!dfs(g[x][i])) return 0;
                return 1;
        }
        void init(int n) {
                this->n=n;
                for(int i=0;i<2*n;i++) g[i].clear();
                memset(mark,0,sizeof(mark));
        }
        void addc(int x,int xval,int y,int yval) {
                x=x*2+xval;
                y=y*2+yval;
                g[x^1].push_back(y);
                g[y^1].push_back(x);
        }
        bool solve() {
                for(int i=0;i<2*n;i+=2) {
                        if(!mark[i]&&!mark[i+1]) {
                                top=0;
                                if(!dfs(i)) {
                                        while(top) mark[st[top--]]=0;
                                        if(!dfs(i+1)) return 0;
                                }
                        }
```

```
                }
                return 1;
        }

} s;
```

//有向图的强联通分量

```
//tarjan 求 SCC
struct Edge {
        int v,nxt;
}e[M];
int en=1,front[N];
void adde(int u,int v)
{
        e[++en]=(Edge){v,front[u]}; front[u]=en;
}

int n,top,dfn;
int st[N],sccno[N],scc_cnt,pre[N],lowlink[N];

void tarjan(int u)
{
        pre[u]=lowlink[u]=++dfn;
        st[++top]=u;
        trav(u,i) {
                int v=e[i].v;
                if(!pre[v]) {
                        tarjan(v);
                        lowlink[u]=min(lowlink[u],lowlink[v]);
                } else
                if(!sccno[v])
                        lowlink[u]=min(lowlink[u],pre[v]);
        }
        if(lowlink[u]==pre[u]) {
                scc_cnt++;
                for(;;) {
                        int x=st[top--];
                        sccno[x]=scc_cnt;
                        if(x==u) break;
                }
        }
}
```

//无向图的边的双连通分量

```
//BCC
struct Edge {
	int u,v,nxt;
}e[M];
int en=1,front[N];
void adde(int u,int v)
{
	e[++en]=(Edge){u,v,front[u]}; front[u]=en;
}

Edge st[N];
vector<int> bcc[N];
int pre[N],iscut[N],bccno[N],top,dfn,bcc_cnt;

int dfs(int u,int fa)
{
	int lowu=pre[u]=++dfn;
	int child=0;
	trav(u,i) {
		int v=e[i].v;
		Edge E=e[i];
		if(!pre[v]) {
			st[++top]=E;
			child++;
			int lowv=dfs(v,u);
			lowu=min(lowu,lowv);
			if(lowv>=pre[u]) {
				iscut[u]=1;
				bcc_cnt++;
				for(;;) {
					Edge x=st[top--];
					if(bccno[x.u]!=bcc_cnt) {
						bccno[x.u]=bcc_cnt;

	bcc[bcc_cnt].push_back(x.u);
					}
					if(bccno[x.v]!=bcc_cnt) {
						bccno[x.v]=bcc_cnt;

	bcc[bcc_cnt].push_back(x.v);
					}
					if(x.u==u&&x.v==v) break;
				}
```

```
                        }
                } else
                if(pre[v]<pre[u] && v!=fa) {
                        st[++top]=E;
                        lowu=min(lowu,pre[v]);
                }
        }
        if(fa<0&&child==1) iscut[u]=0;
        return lowu;
}
```

//最短路
//spfa

```
struct Edge {
        int v,w,nxt;
}e[M];
int en=1,front[N];
void adde(int u,int v,int w)
{
        e[++en]=(Edge){v,w,front[u]}; front[u]=en;
}

queue<int> q;
int inq[N],dis[N];
void spfa(int s)
{
        dis[s]=0; inq[s]=1;
        q.push(s);
        while(!q.empty()) {
                int u=q.front(); q.pop();
                inq[u]=0;
                trav(u,i) {
                        int v=e[i].v;
                        if(dis[v]>dis[u]+e[i].w) {
                                dis[v]=dis[u]+e[i].w;
                                if(!inq[v]) {
                                        inq[v]=1;
                                        q.push(v);
                                }
                        }
                }
        }
}
```

//dijkstra

```
struct Node {
	int id,dis;
	bool operator < (const Node& rhs) const
	{
		return dis>rhs.dis;
	}
};
priority_queue<Node> q;

int vis[N],dis[N];
void dijkstra(int s)
{
	q.push((Node){s,0});
	while(!q.empty()) {
		int u=q.top().id;
		if(vis[u]) continue;
		vis[u]=1;
		trav(u,i) {
			int v=e[i].v;
			if(dis[v]>dis[u]+e[i].w) {
				dis[v]=dis[u]+e[i].w;
				q.push((Node){v,dis[v]});
			}
		}
	}
}
```

//最小生成树
//Kruskal

```
int fa[N];
int find(int u)
{
	if(!fa[u] || u==fa[u]) return fa[u]=u;
	return fa[u]=find(fa[u]);
}

struct Edge {
	int u,v,w;
	bool operator < (const Edge& rhs) const
```

```
        {
                return w<rhs.w;
        }
}e[M];
int tot;

void Kruskal()
{
        sort(e+1,e+tot+1);
        for(int i=1;i<=tot;i++) {
                int u=e[i].u,v=e[i].v;
                int x=find(u),y=find(v);
                if(x!=y) {
                        fa[x]=y;
                        //加入树边(u,v)
                }
        }
}
```

//最大流
//Dinic 算法求最大流

```
struct Edge {
        int u,v,cap,flow;
};
struct Dinic {
        int d[N],cur[N],vis[N];
        vector<Edge> es;
        vector<int> g[N];
        queue<int> q;

        void AddEdge (int u,int v,int w) {
                es.push_back((Edge){u,v,w,0});
                es.push_back((Edge){v,u,0,0});
                int m=es.size();
                g[u].push_back(m-2);
                g[v].push_back(m-1);
        }
        bool bfs(int s,int t) {
                memset(vis,0,sizeof(vis));
                d[s]=0; vis[s]=1;
                q.push(s);
                while(!q.empty()) {
                        int u=q.front(); q.pop();
                        FOR(i,0,(int)g[u].size()-1) {
```

```
                                Edge& e=es[g[u][i]];
                                int v=e.v;
                                if(e.cap>e.flow&&!vis[v]) {
                                        vis[v]=1;
                                        d[v]=d[u]+1;
                                        q.push(v);
                                }
                        }
                }
                return vis[t];
        }
        int dfs(int u,int a,int t) {
                if(u==t||a==0) return a;
                int flow=0,f;
                for(int& i=cur[u];i<g[u].size();i++) {
                        Edge& e=es[g[u][i]];
                        int v=e.v;
                        if(d[v]==d[u]+1&&(f=dfs(v,min(a,e.cap-e.flow),t))>0)
{
                                e.flow+=f;
                                es[g[u][i]^1].flow-=f;
                                flow+=f,a-=f;
                                if(!a) break;
                        }
                }
                return flow;
        }
        int maxflow(int s,int t) {
                int flow=0;
                while(bfs(s,t)) {
                        memset(cur,0,sizeof(cur));
                        flow+=dfs(s,inf,t);
                }
                return flow;
        }
} dc;
```

//最小费用最大流

//最短路算法求最小费用最大流

```
struct Edge {
        int u,v,cap,flow,cost;
        Edge(int _,int __,int ___,int ____,int _____)
        { u=_,v=__,cap=___,flow=____,cost=_____; }
};
```

```
struct MCMF {
	int n,m,s,t;
	int d[N],p[N],a[N],inq[N];
	vector<Edge> es;
	vector<int> g[N];
	queue<int> q;
	void init(int n) {
		this->n=n;
		es.clear();
		for(int i=0;i<=n;i++) g[i].clear();
	}
	void AddEdge(int u,int v,int w,int c) {
		es.push_back(Edge(u,v,w,0,c));
		es.push_back(Edge(v,u,0,0,-c));
		int m=es.size();
		g[u].push_back(m-2);
		g[v].push_back(m-1);
	}
	bool spfa(int s,int t,ll& flow,ll& cost) {
		memset(inq,0,sizeof(inq));
		for(int i=0;i<=n;i++) d[i]=inf;
		inq[s]=1; d[s]=p[s]=0; a[s]=inf;
		q.push(s);
		while(!q.empty()) {
			int u=q.front(); q.pop();
			inq[u]=0;
			for(int i=0;i<g[u].size();i++) {
				Edge& e=es[g[u][i]];
				int v=e.v;
				if(d[v]>d[u]+e.cost && e.cap>e.flow) {
					d[v]=d[u]+e.cost;
					a[v]=min(a[u],e.cap-e.flow);
					p[v]=g[u][i];
					if(!inq[v])
						inq[v]=1 , q.push(v);
				}
			}
		}
		if(d[t]==inf) return 0;
		flow+=a[t],cost+=a[t]*d[t];
		for(int x=t;x!=s;x=es[p[x]].u) {
			es[p[x]].flow+=a[t];
			es[p[x]^1].flow-=a[t];
```

```
                }
                return 1;
        }
        void mcmf(int s,int t,ll& cost,ll& flow) {
                flow=cost=0;
                while(spfa(s,t,cost,flow)) ;
        }
} mc;
```

//KM 算法
//KM 算法求二分图的最佳完美匹配

```
struct KM {
        int slack[N],res[N];
        int l[N],r[N],lx[N],rx[N],g[N][N];

        void clear(int n) {
                for(int i=1;i<=n;i++) {
                        res[i]=0;
                        for(int j=1;j<=n;j++) g[i][j]=-1;
                }
        }
        bool find(int x,int n) {
                lx[x]=1;
                for(int i=1;i<=n;i++)
                        if(!rx[i]&&g[x][i]!=-1) {
                                int tmp=g[x][i]-l[x]-r[i];
                                if(!tmp) {
                                        rx[i]=1;
                                        if(!res[i]||find(res[i],n)) {
                                                res[i]=x;
                                                return 1;
                                        }
                                } else
                                        slack[i]=min(slack[i],tmp);
                        }
                return 0;
        }
        int solve(int n) {
                if(!n) return 0;
                for(int i=1;i<=n;i++) r[i]=0;
                for(int i=1;i<=n;i++) {
                        l[i]=INF;
                        for(int j=1;j<=n;j++) if(g[i][j]!=-1)
                                l[i]=min(l[i],g[i][j]);
```

```
                }
                for(int i=1;i<=n;i++) {
                        for(int j=1;j<=n;j++) slack[j]=INF;
                        for(;;) {
                                for(int j=1;j<=n;j++) lx[j]=rx[j]=0;
                                if(find(i,n)) break;
                                int mini=INF;
                                for(int i=1;i<=n;i++) if(!rx[i])
                                        mini=min(mini,slack[i]);
                                for(int i=1;i<=n;i++) {
                                        if(lx[i]) l[i]+=mini;
                                        if(rx[i]) r[i]-=mini;
                                        else slack[i]-=mini;
                                }
                        }
                }
                int ans=0;
                for(int i=1;i<=n;i++)
                        ans+=l[i]+r[i];
                return ans;
        }
} km;
```

4 树

//LCA

//倍增法求 LCA

```
//倍增法可以在线构造 比较灵活
struct Edge {
        int v,nxt;
}e[M];
int en=1,front[N];
void adde(int u,int v)
{
        e[++en]=(Edge){v,front[u]}; front[u]=en;
}

int fa[N][D],dep[N];

void dfs(int u)
{
        for(int i=1;i<D;i++)
                fa[u][i]=fa[fa[u][i-1]][i-1];
        trav(u,i) {
                int v=e[i].v;
                if(v!=fa[u][0]) {
```

```
                        fa[v][0]=u;
                        dep[v]=dep[u]+1;
                        dfs(v);
                }
        }
}
int lca(int u,int v)
{
        if(dep[u]<dep[v]) swap(u,v);
        int t=dep[u]-dep[v];
        for(int i=0;i<D;i++)
                if(t&(1<<i)) u=fa[u][i];
        if(u==v) return u;
        for(int i=D-1;i>=0;i--)
                if(fa[u][i]!=fa[v][i])
                        u=fa[u][i],v=fa[v][i];
        return fa[u][0];
}
```

//树链剖分求 LCA

```
//比较快
struct Edge {
        int v,nxt;
}e[M];
int en=1,front[N];
void adde(int u,int v)
{
        e[++en]=(Edge){v,front[u]}; front[u]=en;
}

int fa[N],top[N],siz[N],dep[N],son[N];
void dfs1(int u)
{
        siz[u]=1; son[u]=0;
        trav(u,i) {
                int v=e[i].v;
                if(v!=fa[u]) {
                        fa[v]=u;
                        dep[v]=dep[u]+1;
                        dfs1(v);
                        siz[u]+=siz[v];
                        if(siz[v]>siz[son[u]]) son[u]=v;
                }
        }
```

```
}
void dfs2(int u,int tp)
{
	top[u]=tp;
	if(son[u]) dfs2(son[u],tp);
	trav(u,i)
		if(e[i].v!=fa[u]&&e[i].v!=son[u])
			dfs2(e[i].v,e[i].v);
}
int lca(int u,int v)
{
	while(top[u]!=top[v]) {
		if(dep[top[u]]<dep[top[v]]) swap(u,v);
		u=fa[top[u]];
	}
	return dep[u]<dep[v]? u:v;
}
```

//树链剖分
//树链剖分

```
struct Edge {
	int v,nxt;
}e[M];
int en=1,front[N];
void adde(int u,int v)
{
	e[++en]=(Edge){v,front[u]}; front[u]=en;
}

int fa[N],top[N],siz[N],dep[N],son[N],bl[N],dfn;
void dfs1(int u)
{
	siz[u]=1; son[u]=0;
	trav(u,i) {
		int v=e[i].v;
		if(v!=fa[u]) {
			fa[v]=u;
			dep[v]=dep[u]+1;
			dfs1(v);
			siz[u]+=siz[v];
			if(siz[v]>siz[son[u]]) son[u]=v;
		}
	}
}
```

```
void dfs2(int u,int tp)
{
        top[u]=tp; bl[u]=++dfn;
        if(son[u]) dfs2(son[u],tp);
        trav(u,i)
                if(e[i].v!=fa[u]&&e[i].v!=son[u])
                        dfs2(e[i].v,e[i].v);
}
//以合适的数据结构 T 维护重链
int ans;
int query(int u,int v)
{
        while(top[u]!=top[v]) {
                if(dep[top[u]]<dep[top[v]]) swap(u,v);
                ans<-query(T,bl[top[u]],bl[u]);
                u=fa[top[u]];
        }
        if(u==v) return ;
        if(dep[u]>dep[v]) swap(u,v);
        ans<-query(T,bl[u],bl[v]);
        <-ans
}
//类似-查询树上任意两节点的方法
void modify() {}
```

//点分治
//点分治

```
struct Edge {
        int v,nxt;
}e[M];
int en=1,front[N];
void adde(int u,int v)
{
        e[++en]=(Edge){v,front[u]}; front[u]=en;
}

int rt,n,size,vis[N],siz[N],f[N],dep[N];

void get_root(int u,int fa)
{
        siz[u]=1; f[u]=0;
        trav(u,i) {
                int v=e[i].v;
                if(v!=fa) {
```

```
                        get_root(v,u);
                        siz[u]+=siz[v];
                        if(siz[v]>f[u]) f[u]=siz[v];
                }
        }
        f[u]=max(f[u],size-siz[u]);
        if(f[u]<f[rt]) rt=u;
}
void solve(int u)
{
        vis[u]=1;
        //计算经过根 u 的信息
        trav(u,i) if(!vis[e[i].v])
        {
                //统计当前子树信息
                //与前 i-1 个子树信息结合计算贡献
                //将当前子树信息加入前 i-1 个子树信息
        }
        trav(u,i) if(!vis[e[i].v]) {
                int v=e[i].v;
                size=siz[v]; rt=0;
                get_root(v,-1);
                solve(rt);
        }
}
int main()
{
        //blabla
        size=f[0]=n;
        rt=0; get_root(rt,-1);
        solve(rt);
}
```

5 字符串

//KMP

```
//KMP 算法
//s[s]==s[fail[i]] , fail[i]!=0
int f[N]; char s[N];
void get_fail()
{
        int j=0;
        int n=strlen(s);
        for(int i=2;i<n;i++) {
                while(j&&s[j+1]!=s[i]) j=f[j];
                if(s[j+1]==s[i]) j++;
```

```
                f[i]=j;
        }
}
```

//AC 自动机

```
//AC 自动机
struct AC_auto {
        int sz,ch[N][26],f[N],val[N];
        AC_auto() {
                sz=1;
                memset(ch,0,sizeof(ch));
        }
        void insert(char* s) {
                int u=0;
                for(int i=0;s[i];i++) {
                        int c=s[i]- ‘a ‘;
                        if(!ch[u][c]) ch[u][c]=++sz;
                        u=ch[u][c];
                }
                val[u]=1;
        }
        void get_fail() {
                queue<int> q;
                f[0]=0;
                for(int c=0;c<26;c++)
                        if(ch[0][c]) f[ch[0][c]]=0,q.push(ch[0][c]);
                while(!q.empty()) {
                        int qr=q.front(); q.pop();
                        for(int c=0;c<26;c++) {
                                int u=ch[qr][c];
                                if(!u) continue;
                                q.push(u);
                                int v=f[qr];
                                while(v&&!ch[v][c]) v=f[v];
                                if(val[ch[v][c]]) val[u]=1;
                                f[u]=ch[v][c];
                        }
                }
        }
};
```

//后缀自动机 SAM

```
struct SAM {
        int sz,last,fa[N],ch[N][26],l[N];
```

```
        SAM() {
                sz=0; last=++sz;
                memset(l,0,sizeof(l));
                memset(fa,0,sizeof(fa));
        }
        void Add(int c) {
                int np=++sz,p=last; last=np;
                l[np]=l[p]+1;
                for(;p&&!ch[p][c];p=fa[p]) ch[p][c]=np;
                if(!p) fa[np]=1;
                else {
                        int q=ch[p][c];
                        if(l[q]==l[p]+1) fa[np]=q;
                        else {
                                int nq=++sz; l[nq]=l[p]+1;
                                memcpy(ch[nq],ch[q],sizeof(ch[q]));
                                fa[nq]=fa[q];
                                fa[q]=fa[np]=nq;
                                for(;q==ch[p][c];p=fa[p]) ch[p][c]=nq;
                        }
                }
        }
        //do some other things

} sam;
```

//后缀数组

```
#define rep(a,b,c) for(int a=(b);a>=(c);a--)
#define FOR(a,b,c) for(int a=(b);a<=(c);a++)
char s[N];
int c[N],t[N],t2[N],height[N],rank[N],sa[N];
void build_sa(int m,int n) {
        int *x=t,*y=t2,p,k;
        FOR(i,0,m-1) c[i]=0;
        FOR(i,0,n-1) c[x[i]=s[i]]++;
        FOR(i,0,m-1) c[i]+=c[i-1];
        rep(i,n-1,0) sa[--c[x[i]]]=i;

        for(k=1;k<=n;k<<=1) {
                p=0;
                FOR(i,n-k,n-1) y[p++]=i;
                FOR(i,0,n-1) if(sa[i]>=k) y[p++]=sa[i]-k;

                FOR(i,0,m-1) c[i]=0;
```

```
                FOR(i,0,n-1) c[x[y[i]]]++;
                FOR(i,0,m-1) c[i]+=c[i-1];
                rep(i,n-1,0) sa[--c[x[y[i]]]]=y[i];

                swap(x,y);
                p=1; x[sa[0]]=0;
                FOR(i,1,n-1)
                        x[sa[i]]=y[sa[i]]==y[sa[i-1]]&&y[sa[i]+k]==y[sa[i-
1]+k]? p-1:p++;
                if(p>=n) break;
                m=p;
        }
}
void get_height(int n)
{
        FOR(i,0,n-1) rank[sa[i]]=i;
        int k=0;
        FOR(i,0,n-1) {
                if(k) k--;
                int j=sa[rank[i]-1];
                while(s[i+k]==s[j+k]) k++;
                height[rank[i]]=k;
        }
}
```

Manacher：

//Manacher 算法

```
char s[N],a[N];
int p[N];
void Add(int l,int r)
{
        l=l/2,r=r/2-1;
        if(l>r) return ;
        //q[++tot]=(Seg){l,r};
        //[l,r]为一个回文串
}
void Manacher()
{
        int n=strlen(s+1);
        int m=n*2+1;
        for(int i=1;i<=n;i++) {
                a[i<<1]=s[i];
                a[i<<1|1]= '# ';
        }
```

```
        a[0]= '+ ',a[1]= '# ',a[m+1]= '- ';
        int mx=0,id;
        for(int i=1;i<=m;i++) {
                if(mx>i) p[i]=min(mx-i,p[id*2-i]);
                else p[i]=1;
                while(a[i-p[i]]==a[i+p[i]]) p[i]++;
                Add(i-p[i],i+p[i]);
                if(p[i]+i>mx) mx=i+p[i],id=i;
        }
}
```

6 计算几何

计算几何基础知识：

//计算几何基础

```
const double eps = 1e-10;
int dcmp(double x) {
        if(fabs(x)<eps) return 0; else return x<0? -1:1;
}

struct Pt {
        double x,y;
        Pt(double x=0,double y=0):x(x),y(y) {}
};
typedef Pt vec;

vec operator - (Pt A,Pt B) { return vec(A.x-B.x,A.y-B.y); }
vec operator + (vec A,vec B) { return vec(A.x+B.x,A.y+B.y); }
vec operator * (vec A,double p) { return vec(A.x*p , A.y*p); }
bool operator < (const Pt& a,const Pt& b) {
        return a.x<b.x || (a.x==b.x && a.y<b.y);
}
bool operator == (const Pt& a,const Pt& b) {
        return dcmp(a.x-b.x)==0 && dcmp(a.y-b.y)==0;
}

double cross(vec A,vec B) { return A.x*B.y-A.y*B.x; }
double Dot(vec A,vec B) { return A.x*B.x+A.y*B.y; }
double Len(vec A) { return sqrt(Dot(A,A)); }
double Angle(vec A,vec B) { return acos(Dot(A,B)/Len(A)/Len(B)); }
```

//逆时针旋转 rad 角度

```
vec rotate(vec A,double rad) {
```

```
        return vec(A.x*cos(rad)-A.y*sin(rad) , A.x*sin(rad)+A.y*cos(rad));
}
```

//法向量 左转 90 度 长度归 1

```
vec Normal(vec A)
{
        double L=Len(A);
        return vec(-A.y/L,A.x/L);
}
```

//判断点在线段上

```
bool OnSeg(Pt P,Pt a1,Pt a2) {
        return dcmp(cross(a1-P,a2-P))==0 && dcmp(Dot(a1-P,a2-P))<0;
}
```

//直线交点

```
Pt LineIntersection(Pt P,vec v,Pt Q,vec w) {
        vec u=P-Q;
        double t=cross(w,u)/cross(v,w);
        return P+v*t;
}
double DistoLine(Pt P,Pt A,Pt B) {
        vec v1=B-A,v2=P-A;
        return fabs(cross(v1,v2))/Len(v1);
}
```

//线段不含端点 判断相交

```
bool SegIntersection(Pt a1,Pt a2,Pt b1,Pt b2) {
        double c1=cross(a2-a1,b1-a1) , c2=cross(a2-a1,b2-a1) ,
                    c3=cross(b2-b1,a1-b1) , c4=cross(b2-b1,a2-b1);
        return dcmp(c1)*dcmp(c2)<0 && dcmp(c3)*dcmp(c4)<0;
        // b1 b2 在线段 a1a2 的两侧  a1 a2 在线段 b1b2 的两侧  规范相交
}
```

//线段含端点 判断线段严格相交

```
bool SegInter(Pt s1, Pt e1, Pt s2, Pt e2) {
    if( min(s1.x, e1.x) <= max(s2.x, e2.x) &&
        min(s1.y, e1.y) <= max(s2.y, e2.y) &&
        min(s2.x, e2.x) <= max(s1.x, e1.x) &&
        min(s2.y, e2.y) <= max(s1.y, e1.y) &&
        cross(e1-s1,s2-s1) * cross(e1-s1,e2-s1) <= 0 &&
        cross(e2-s2,s1-s2) * cross(e2-s2,e1-s2) <= 0
          ) return true;
    return false;
}
```

//点到线段的距离

```
double DistoSeg(Pt P,Pt A,Pt B) {
        if(A==B) return Len(P-A);
        vec v1=B-A , v2=P-A , v3=P-B;
        if(dcmp(Dot(v1,v2))<0) return Len(v2);
        else if(dcmp(Dot(v1,v3))>0) return Len(v3);
        else return fabs(cross(v1,v2))/Len(v1);
}
```

//多边形面积

```
double PolygonArea(Pt* p,int n)
{
        double S=0;
        for(int i=1;i<n-1;i++)
                S+=cross(p[i]-p[0],p[i+1]-p[0]);
        return S/2;
}
```

//凸包

```
const int N = 400000+10;
const double PI = acos(-1.0);
const double eps = 1e-12;

int dcmp(double x) {
        if(fabs(x)<eps) return 0; else return x<0? -1:1;
}

struct Pt {
        double x,y;
        Pt(double x=0,double y=0) :x(x),y(y) {};
};
typedef Pt vec;

vec operator - (Pt a,Pt b) { return vec(a.x-b.x,a.y-b.y); }
vec operator + (vec a,vec b) { return vec(a.x+b.x,a.y+b.y); }
bool operator == (Pt a,Pt b) {
        return dcmp(a.x-b.x)==0 && dcmp(a.y-b.y)==0;
}
bool operator < (const Pt& a,const Pt& b) {
        return a.x<b.x || (a.x==b.x && a.y<b.y);
}

vec rotate(vec a,double x) {
```

```
        return vec(a.x*cos(x)-a.y*sin(x),a.x*sin(x)+a.y*cos(x));
}
double cross(vec a,vec b) { return a.x*b.y-a.y*b.x; }
double dist(Pt a,Pt b) {
        return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}

vector<Pt> ConvexHull(vector<Pt> p) {
        sort(p.begin(),p.end());
        p.erase(unique(p.begin(),p.end()),p.end());
        int n=p.size() , m=0;
        vector<Pt> ch(n+1);
        for(int i=0;i<n;i++) {
                while(m>1 && cross(ch[m-1]-ch[m-2],p[i]-ch[m-2])<=0) m--;
                ch[m++]=p[i];
        }
        int k=m;
        for(int i=n-2;i>=0;i--) {
                while(m>k && cross(ch[m-1]-ch[m-2],p[i]-ch[m-2])<=0) m--;
                ch[m++]=p[i];
        }
        if(n>1) m--;
        ch.resize(m); return ch;
}
```

//半平面交

```
const int N =  305;
const double bond = 100001;
const double eps = 1e-10;

struct Pt {
        double x,y;
        Pt (double x=0,double y=0):x(x),y(y){}
};
typedef Pt vec;

vec operator + (Pt a,Pt b) { return vec(a.x+b.x,a.y+b.y); }
vec operator - (Pt a,Pt b) { return vec(a.x-b.x,a.y-b.y); }
vec operator * (Pt a,double p) { return vec(a.x*p,a.y*p); }

double cross(Pt a,Pt b) { return a.x*b.y-a.y*b.x; }

struct Line {
        Pt p; vec v; double ang;
```

```
        Line () {}
        Line (Pt p,vec v) :p(p),v(v){ ang=atan2(v.y,v.x); }
        bool operator < (const Line& rhs) const {
                return ang<rhs.ang;
        }
};

bool onleft(Line L,Pt p) { return cross(L.v,p-L.p)>0; }
Pt LineInter(Line a,Line b)
{
        vec u=a.p-b.p;
        double t=cross(b.v,u)/cross(a.v,b.v);
        return a.p+a.v*t;
}
vector<Pt> HPI(vector<Line> L)
{
        int n=L.size();
        sort(L.begin(),L.end());
        int f,r;
        vector<Pt> p(n) , ans;
        vector<Line> q(n);
        q[f=r=0]=L[0];
        for(int i=1;i<n;i++) {
                while(f<r&&!onleft(L[i],p[r-1])) r--;
                while(f<r&&!onleft(L[i],p[f])) f++;
                q[++r]=L[i];
                if(fabs(cross(q[r].v,q[r-1].v))<eps) {
                        r--;
                        if(onleft(q[r],L[i].p)) q[r]=L[i];
                }
                if(f<r) p[r-1]=LineInter(q[r-1],q[r]);
        }
        while(f<r&&!onleft(q[f],p[r-1])) r--;
        if(r-f<=1) return ans;
        p[r]=LineInter(q[r],q[f]);
        for(int i=f;i<=r;i++) ans.push_back(p[i]);
        return ans;
}
```

预设

```
#include <iostream>
#include <vector>
#define rep(i,l,r) for(int i=(l);i<=(r);++i)
#define rpe(i,r,l) for(int i=(r);i>=(l);--i)
using namespace std;
typedef long long ll;
inline ll max(ll x,ll y){return x>y?x:y;}
inline ll read(){
    ll x=0,f=1;char ch=' ';
    while(ch<'0' || ch>'9'){if(ch=='-')f=-1;ch=getchar();}
    while(ch>='0' && ch<='9'){x=x*10+(ch^48);ch=getchar();}
    return f==1?x:-x;
}
```

平面分割线

我们看到过很多直线分割平面的题目，今天的这个题目稍微有些变化，我们要求的是n条折线分割平面的最大数目

```
#include <stdio.h>

int main(){
    int n;
    scanf("%d",&n);
    for(int i = 0;i < n;i++){
        int a;
        scanf("%d",&a);
        printf("%lld\n",2 * a * a - a + 1);
    }
    return 0;
}
```

并查集

非带权模板

```
#include<vector>
using namespace std;
#define maxn 100005
typedef long long ll;
// 连通分量个数
ll cnt;
// 存储一棵树
ll parent[maxn]={0};
// 记录树的“重量”
ll weight[maxn]={0};
void init(ll n) {
```

```
    cnt = n;
    for (ll i = 1; i <= n; i++) {
        parent[i] = i;
        weight[i] = 1;
    }
}
ll find(ll x) {
    while (parent[x] != x) {
        // 进行路径压缩
        parent[x] = parent[parent[x]];
        x = parent[x];
    }
    return x;
}

void u(ll p, ll q) {
    ll rootP = find(p);
    ll rootQ = find(q);
    if (rootP == rootQ)
        return;

    // 小树接到大树下面，较平衡
    if (weight[rootP] > weight[rootQ]) {
        parent[rootQ] = rootP;
        weight[rootP] += weight[rootQ];
    } else {
        parent[rootP] = rootQ;
        weight[rootQ] += weight[rootP];
    }
    cnt--;
}

bool connected(ll p, ll q) {
    return find(p) == find(q);
}
```

银河英雄传说

描述

公元 5801 年，地球居民迁至金牛座 α 第二行星，在那里发表银河联邦创立宣言，同年改元为宇宙历元年，并开始向银河系深处拓展。

宇宙历 799 年，银河系的两大军事集团在巴米利恩星域爆发战争。泰山压顶集团派宇宙舰队司令莱因哈特率领十万余艘战舰出征，气吞山河集团点名将杨威利组织麾下三万艘战舰迎敌。

杨威利擅长排兵布阵，巧妙运用各种战术屡次以少胜多，难免恣生骄气。在这次决战中，他将巴米利恩星域战场划分成 30000 列，每列依次编号为 $1, 2, \ldots, 30000$。之后，他把自己的战舰也依次编号为 $1, 2, \ldots, 30000$，让第 i 号战舰处于第 i 列，形成“一字长蛇阵”，诱敌深入。这是初始阵形。当进犯之敌到达时，杨威利会多次发布合并指令，将大部分战舰集中在某几列上，实施密集攻击。合并指令为 `M i j`，含义为第 i 号战舰所在的整个战舰队列，作为一个整体（头在前尾在后）接至第 j 号战舰所在的战舰队列的尾部。显然战舰队列是由处于同一列的一个或多个战舰组成的。合并指令的执行结果会使队列增大。

然而，老谋深算的莱因哈特早已在战略上取得了主动。在交战中，他可以通过庞大的情报网络随时监听杨威利的舰队调动指令。

在杨威利发布指令调动舰队的同时，莱因哈特为了及时了解当前杨威利的战舰分布情况，也会发出一些询问指令：`C i j`。该指令意思是，询问电脑，杨威利的第 i 号战舰与第 j 号战舰当前是否在同一列中，如果在同一列中，那么它们之间布置有多少战舰。

作为一个资深的高级程序设计员，你被要求编写程序分析杨威利的指令，以及回答莱因哈特的询问。

输入

第一行有一个整数 T（$1 \leq T \leq 5 \times 10^5$），表示总共有 T 条指令。

以下有 T 行，每行有一条指令。指令有两种格式：

1. `M i j`：i 和 j 是两个整数（$1 \leq i, j \leq 30000$），表示指令涉及的战舰编号。该指令是莱因哈特窃听到的杨威利发布的舰队调动指令，并且保证第 i 号战舰与第 j 号战舰不在同一列。
2. `C i j`：i 和 j 是两个整数（$1 \leq i, j \leq 30000$），表示指令涉及的战舰编号。该指令是莱因哈特发布的询问指令。

输出

依次对输入的每一条指令进行分析和处理：

- 如果是杨威利发布的舰队调动指令，则表示舰队排列发生了变化，你的程序要注意到这一点，但是不要输出任何信息。
- 如果是莱因哈特发布的询问指令，你的程序要输出一行，仅包含一个整数，表示在同一列上，第 i 号战舰与第 j 号战舰之间布置的战舰数目。如果第 i 号战舰与第 j 号战舰当前不在同一列上，则输出 -1。

```
//带权并查集模板题
#include<iostream>
#define rep(i,l,r) for(ll i=(l);i<=(r);++i)
#define rpe(i,r,l) for(ll i=(r);i>=(l);--i)
typedef long long ll;
using namespace std;
inline ll max(ll x,ll y){return x>y?x:y;}
inline ll read(){
    ll x=0,f=1;char ch=' ';
    while(ch<'0' || ch>'9'){if(ch=='-')f=-1;ch=getchar();}
    while(ch>='0' && ch<='9'){x=x*10+(ch^48);ch=getchar();}
```

```
    return f==1?x:-x;
}
#define maxn 30005

// 连通分量个数
ll cnt;
// 存储一棵树
ll parent[maxn]={0};
//编号为i的战舰到parent[i]的距离
ll d[maxn] = {0};
//编号为i的战舰所在的一列有多少战舰
ll s[maxn] = {0};
void init(ll n) {
    cnt = n;
    for (ll i = 1; i <= n; i++) {
        parent[i] = i;
        s[i] = 1;
    }
}
ll find(ll x) {
    if (parent[x] != x) {
        int k = parent[x];
        //每次find的时候都更新d和s
        parent[x] = find(parent[x]);
        d[x] += d[k];
        s[x] = s[parent[x]];
    }
    return parent[x];
}

void u(ll p, ll q) {
    ll rootP = find(p);
    ll rootQ = find(q);
    if (rootP == rootQ)
        return;
    //p放到q的后面
    parent[rootP] = rootQ;

    d[rootP] = d[rootQ] + s[rootQ];
    s[rootQ] += s[rootP];
    s[rootP] = s[rootQ];
    cnt--;
}

bool connected(ll p, ll q) {
    return find(p) == find(q);
}

int main(){
    init(30000);
    ll t = read();
    char c;
    ll a,b;
    rep(i,1,t){
        cin>>c;
        a = read();
        b = read();
        if(c == 'M'){
```

```
            u(a,b);
        }else{
            if(connected(a,b)){
                printf("%lld\n",abs(d[a]-d[b])-1);
            }else{
                printf("-1\n");
            }
        }
    }
    return 0;
}
```

单调栈

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
int n,a[3000005],f[3000005];//a是需要判断的数组（即输入的数组），f是存储答案的数组
stack<int> S;
int main() {
    scanf("%d",&n);
    for (int i = 1; i <= n; ++i) scanf("%d",&a[i]);

    for (int i = 1; i <= n; ++i) {
        //栈从栈底到栈顶递减
        //从前往后扫 遇到比栈顶元素大的 就是栈顶的“下一个更大的数”
        while (!S.empty() && a[S.top()] < a[i]) {
            f[S.top()] = i;
            S.pop();
        }
        S.push(i);
    }
    for (int i = 1; i <= n; ++i) printf("%d ",f[i]);
    return 0;
}
```

动态规划

低价购买

“低价购买”这条建议是在奶牛股票市场取得成功的一半规则。要想被认为是伟大的投资者，你必须遵循以下的问题建议:“低价购买；再低价购买”。每次你购买一支股票,你必须用低于你上次购买它的价格购买它。买的次数越多越好!你的目标是在遵循以上建议的前提下，求你最多能购买股票的次数。你将被给出一段时间内一支股票每天的出售价(≤2^16范围内的正整数)，你可以选择在哪些天购买这支股票。每次购买都必须遵循“低价购买；再低价购买”的原则。写一个程序计算最大购买次数。

这里是某支股票的价格清单：

日期1,2,3,4,5,6,7,8,9,10,11,12

价格68,69,54,64,68,64,70,67,78,62,98,87

最优秀的投资者可以购买最多4次股票，可行方案中的一种是：

日期2,5,6,10

价格69,68,64,62

输入

第1行:N(1≤N≤5000)，股票发行天数

第2行:N个数，是每天的股票价格。

输出

两个数:

最大购买次数和拥有最大购买次数的方案数（<2^63），当二种方案“看起来一样”时（就是说它们构成的价格队列一样的时候）,这2种方案被认为是相同的。

```
#include <iostream>

using namespace std;
int a[5005] = {0};
int dp[5005] = {0};
long long cnt[5005] = {0};
int main(){
    int n;
    cin>>n;
    for(int i = 1;i <= n;i ++){
        cin>>a[i];
    }
    int maxlen = 0;
    long long ans = 0;
    for(int i = 1;i <= n;i++){
        dp[i] = 1;
        cnt[i] = 1;
        for(int j = 1; j < i;j++){
            if(a[i] < a[j]){
                if(dp[j] + 1 > dp[i]){
                    dp[i] = dp[j] + 1;
                    cnt[i] = cnt[j];
                }else if(dp[j] + 1 == dp[i]){
                    cnt[i] += cnt[j];
                }
            }
        }
        if(dp[i] > maxlen){
            maxlen = dp[i];
            ans = cnt[i];
        }else if(dp[i] == maxlen){
            ans += cnt[i];
        }
    }
    printf("%d %lld",maxlen,ans);
```

```
    return 0;
}
```

装箱问题

描述

有一个箱子容量为V（正整数，$0 \leq V \leq 20000$），同时有n个物品（$0 < n \leq 30$，每个物品有一个体积（正整数）。

要求n个物品中，任取若干个装入箱内，使箱子的剩余空间为最小。

输入

1个整数，表示箱子容量

1个整数，表示有n个物品

接下来n行，分别表示这n个物品的各自体积

输出

1个整数，表示箱子剩余空间。

```
#include <iostream>
#include <cmath>
using namespace std;
int a[35] = {0};
int dp[30005] = {0};
int main(){
    int V,n;
    cin>>V>>n;
    for(int i = 1;i <= n;i++){
        cin>>a[i];
    }
    for(int i = 1;i <= n;i++){
        for(int j = V;j >= a[i];j--){
            dp[j] = max(dp[j],dp[j-a[i]]+a[i]);
        }
    }
    cout<<V - dp[V]<<endl;
    return 0;
}
```

分组背包

描述

自01背包问世之后，小 A 对此深感兴趣。一天，小 A 去远游，却发现他的背包不同于01背包，他的物品大致可分为k组，每组中的物品相互冲突，现在，他想知道最大的利用价值是多少。

输入

两个数m,n，表示一共有n件物品，总重量为m。

接下来n行，每行3个数ai,bi,ci，表示物品的重量，利用价值，所属组数。

输出

一个数，最大的利用价值。

```
#include <iostream>
#include <cmath>
#include <map>
#include <vector>
using namespace std;
int a[1500]={0};
int b[1500]={0};
int c[1500]={0};
int main(){
    //n个物品总重m
    map<int,vector<int>> map;
    int m,n;
    //共k组
    int k = 0;
    cin>>m>>n;
    for(int i = 1;i <= n;i++){
        //重量 价值 组别
        cin>>a[i]>>b[i]>>c[i];
        k = max(k,c[i]);
        if(map.find(c[i]) == map.end()){
            vector<int> v;
            v.push_back(i);
            map.insert(pair<int,vector<int>>(c[i],v));
        }else{
            map[c[i]].push_back(i);
        }
    }
    //dp[i][j]表示前i组 重量j 最大价值
    vector<vector<int>> dp(k+1,vector<int>(m+1));
    for(int i = 1;i <= k;i++){
        vector<int> v = map[i];
        for(int j = 1;j <= m;j++){
            for(int l : v){
                if(j >= a[l]){
                    dp[i][j] = max(max(dp[i-1][j],dp[i-1][j-a[l]]+b[l]),dp[i][j]);
                }else{
                    dp[i][j] = max(dp[i-1][j],dp[i][j]);
                }
            }
        }
    }
    cout<<dp[k][m]<<endl;

    return 0;
}
```

石子合并

描述

在一个园形操场的四周摆放N堆石子,现要将石子有次序地合并成一堆.规定每次只能选相邻的2堆合并成新的一堆，并将新的一堆的石子数，记为该次合并的得分。

试设计出一个算法,计算出将N堆石子合并成1堆最大得分.

输入

第1行一个正整数N,1≤N≤2000,表示有N堆石子.

第2行有N个数,分别表示每堆石子的个数.

输出

共1行,最大得分

```
#include<iostream>
#include<cstdio>
#include<cmath>
using namespace std;
int n,maxl,f[4005][4005],num[4005];
int s[4005];
inline int d(int i,int j){return s[j]-s[i-1];}
//O(n^3)的转移方程: f[i][j] = max(f[i][k]+f[k+1][j]+d[i][j];
int main()
{
    scanf("%d",&n);
    for(int i=1;i<=n+n;i++)
    {
        /**
     * 输入只到n 后面后续处理n~2n的数
     */
        scanf("%d",&num[i]);
        num[i+n]=num[i];
        s[i]=s[i-1]+num[i];
    }
    // 枚举区间长度
    // 区间长度从1到n-1枚举 不能枚举i从1到2n j从1到2n 要按长度从1到n-1
    // 因为如果按前者枚举的话 有的前面的可能没dp到
    // 比如dp[1][3]要用到dp[2][3] 但是dp[2][3]还没有计算出 所以要按i和j之前的长度枚举
    for(int p=1;p<n;p++)
    {
        for(int i=1,j=i+p;(j<n+n) && (i<n+n);i++,j=i+p)
        {
            //超时
            // for(int k=i;k<j;k++)
            // {
            //     f[i][j] = max(f[i][j], f[i][k]+f[k+1][j]+d(i,j));
            // }
            f[i][j] = max(f[i+1][j],f[i][j-1])+d(i,j);
        }
    }
    for(int i=1;i<=n;i++)
```

```
    {
        maxl=max(maxl,f[i][i+n-1]);
    }
    printf("%d\n",maxl);
    return 0;
}
```

高精度

```
#include <algorithm> // max
#include <cassert>   // assert
#include <cstdio>    // printf,sprintf
#include <cstring>   // strlen
#include <iostream>  // cin,cout
#include <string>    // string类
#include <vector>    // vector类
using namespace std;
typedef unsigned long long LL;
struct BigInteger {


    static const int BASE = 100000000;
    static const int WIDTH = 8;
    vector<int> s;

    BigInteger& clean(){while(!s.back()&&s.size()>1)s.pop_back(); return *this;}
    BigInteger(LL num = 0) {*this = num;}
    BigInteger(string s) {*this = s;}
    BigInteger& operator = (long long num) {
        s.clear();
        do {
            s.push_back(num % BASE);
            num /= BASE;
        } while (num > 0);
        return *this;
    }
    BigInteger& operator = (const string& str) {
        s.clear();
        int x, len = (str.length() - 1) / WIDTH + 1;
        for (int i = 0; i < len; i++) {
            int end = str.length() - i*WIDTH;
            int start = max(0, end - WIDTH);
            sscanf(str.substr(start,end-start).c_str(), "%d", &x);
            s.push_back(x);
        }
        return (*this).clean();
    }

    BigInteger operator + (const BigInteger& b) const {
        BigInteger c; c.s.clear();
        for (int i = 0, g = 0; ; i++) {
            if (g == 0 && i >= s.size() && i >= b.s.size()) break;
            int x = g;
            if (i < s.size()) x += s[i];
            if (i < b.s.size()) x += b.s[i];
```

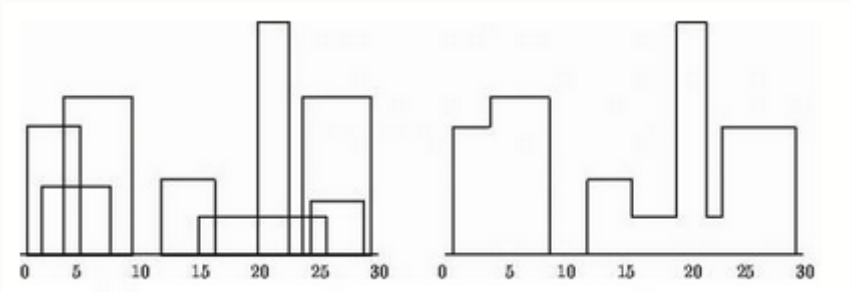
```
            c.s.push_back(x % BASE);
            g = x / BASE;
        }
        return c;
    }
    BigInteger operator - (const BigInteger& b) const {
        assert(b <= *this); // 减数不能大于被减数
        BigInteger c; c.s.clear();
        for (int i = 0, g = 0; ; i++) {
            if (g == 0 && i >= s.size() && i >= b.s.size()) break;
            int x = s[i] + g;
            if (i < b.s.size()) x -= b.s[i];
            if (x < 0) {g = -1; x += BASE;} else g = 0;
            c.s.push_back(x);
        }
        return c.clean();
    }
    BigInteger operator * (const BigInteger& b) const {
        int i, j; LL g;
        vector<LL> v(s.size()+b.s.size(), 0);
        BigInteger c; c.s.clear();
        for(i=0;i<s.size();i++) for(j=0;j<b.s.size();j++) v[i+j]+=LL(s[i])*b.s[j];
        for (i = 0, g = 0; ; i++) {
            if (g ==0 && i >= v.size()) break;
            LL x = v[i] + g;
            c.s.push_back(x % BASE);
            g = x / BASE;
        }
        return c.clean();
    }
    BigInteger operator / (const BigInteger& b) const {
        assert(b > 0);  // 除数必须大于0
        BigInteger c = *this;       // 商:主要是让c.s和(*this).s的vector一样大
        BigInteger m;               // 余数:初始化为0
        for (int i = s.size()-1; i >= 0; i--) {
            m = m*BASE + s[i];
            c.s[i] = bsearch(b, m);
            m -= b*c.s[i];
        }
        return c.clean();
    }
    BigInteger operator % (const BigInteger& b) const { //方法与除法相同
        BigInteger c = *this;
        BigInteger m;
        for (int i = s.size()-1; i >= 0; i--) {
            m = m*BASE + s[i];
            c.s[i] = bsearch(b, m);
            m -= b*c.s[i];
        }
        return m;
    }
    // 二分法找出满足bx<=m的最大的x
    int bsearch(const BigInteger& b, const BigInteger& m) const{
        int L = 0, R = BASE-1, x;
        while (1) {
            x = (L+R)>>1;
            if (b*x<=m) {if (b*(x+1)>m) return x; else L = x;}
            else R = x;
```

```
        }
    }
    BigInteger& operator += (const BigInteger& b) {*this = *this + b; return *this;}
    BigInteger& operator -= (const BigInteger& b) {*this = *this - b; return *this;}
    BigInteger& operator *= (const BigInteger& b) {*this = *this * b; return *this;}
    BigInteger& operator /= (const BigInteger& b) {*this = *this / b; return *this;}
    BigInteger& operator %= (const BigInteger& b) {*this = *this % b; return *this;}

    bool operator < (const BigInteger& b) const {
        if (s.size() != b.s.size()) return s.size() < b.s.size();
        for (int i = s.size()-1; i >= 0; i--)
            if (s[i] != b.s[i]) return s[i] < b.s[i];
        return false;
    }
    bool operator >(const BigInteger& b) const{return b < *this;}
    bool operator<=(const BigInteger& b) const{return !(b < *this);}
    bool operator>=(const BigInteger& b) const{return !(*this < b);}
    bool operator!=(const BigInteger& b) const{return b < *this || *this < b;}
    bool operator==(const BigInteger& b) const{return !(b < *this) && !(b > *this);}
};

int main(){
    BigInteger a("123"),b("456");
    a += b;
    for(auto i : a.s){
        cout<<i;
    }
    return 0;
}
```

优先队列

轮廓线

怎样描述一个城市的轮廓呢？我们知道 Genoa 所有的建筑共享一个地面，你可以认为它是水平的。所有的建筑用一个三元组(Li,Hi,Ri)，其中Li和Ri分别是建筑的左坐标和右坐标，Hi就是建筑的高度。在下方所示的图表中左边建筑物描述如下(1,11,5)，(2,6,7)，(3,13,9)，(12,7,16)，(14,3,25)，(19,18,22)，(23,13,29)，(24,4,28)，右边轮廓线用(1,11,3,13,9,0,12,7,16,3,19,18,22,3,23,13,29,0)表示：



输入

在输入数据中，你将得到一系列表示建筑的三元组。在输入数据中所有建筑的坐标中的数值都是小于1000000的正整数，且至少有1幢建筑，最多有1000000幢建筑。在输入中每幢建筑的三元组各占一行。三元组中的所有整数应由一个或多个空格分开。

输出

在输出数据中，你被要求给出城市的轮廓线。你可以这样来描述：对于所有轮廓线上的折点，按顺序排好，第奇数个点输出x坐标，第偶数个点输出y坐标，两个数之间用空格分开。

```
#include<cstdio>
#include<queue>
#include<algorithm>
using namespace std;
#define N 1000005
struct building{
    int h,r;//h为高度，r为右端点
    building(int h_,int r_){h=h_;r=r_;}
    bool operator <(const building& i)const{return h<i.h;}//重载运算符，以高度h来建大根堆。
};

struct sp{
    int v,w;//v指向原来位置，w为x坐标
    bool p;//p=0则是左端点，p=1是右端点
    sp(int v_=0,int w_=0,bool p_=0){v=v_;w=w_;p=p_;}
}b[N*2];


int n,a[N][3];
inline bool comp(const sp&x,const sp&y){return x.w<y.w;}

priority_queue<building>q;

int main(){
    int l,h,r;
    while (scanf("%d%d%d",&l,&h,&r)!=EOF){
        a[++n][0]=h;
        a[n][1]=l;
        a[n][2]=r;
        b[n*2-1]=sp(n,l,0);
        b[n*2]=sp(n,r,1);//把左右端点都加个b数组
    }
    sort(b+1,b+2*n+1,comp);//将b数组按x坐标排序
    int last=0;
    //遍历b数组 也即遍历所有的线
    for (int i=1;i<=2*n;i++){
        //说明如果堆顶的building的右坐标小于当前遍历的线的坐标
        //说明堆顶的building已经被遍历过去了 不应当存在在堆里了
        //这一段if是核心 如何判断堆顶的building已经成为了过去 该删掉了
        if (q.size()){
            r=(q.top()).r;
            while (q.size()&&b[i].w>=r){
                q.pop();
                if (q.empty()) break;
                r=(q.top()).r;
            }
        }
```

```
        //是左端点就把左端点代表的building入堆
        if (!b[i].p)
            q.push(building(a[b[i].v][0],a[b[i].v][2]));

        //两条线坐标重合
        if (i<2*n&&b[i].w==b[i+1].w)
            continue;

        if (q.size())
            h=(q.top()).h;
        else
            //堆空高度为0
            h=0;

        //高度变化则输出
        if (h!=last){
            printf("%d %d ",b[i].w,h);
            last=h;
        }
    }
    return 0;
}
```

序列最小和

描述

有两个长度都是N的序列A和B，在A和B中各取一个数相加可以得到N^2个和，求这N^2个和中最小的N个。

输入

第一行一个正整数N；

第二行N个整数A_i，abs(A_i)≤10^9;

第三行N个整数B_i，abs(B_i)≤10^9;

输出

输出仅一行，包含N个整数，从小到大输出这N个最小的和，相邻数字之间用空格隔开。

```
#include<iostream>
#include<queue>
#include<vector>
#include<algorithm>
#define rep(i,l,r) for(int i=(l);i<=(r);++i)
#define rpe(i,r,l) for(int i=(r);i>=(l);--i)
typedef long long ll;
using namespace std;
inline ll max(ll x,ll y){return x>y?x:y;}
inline ll read(){
    ll x=0,f=1;char ch=' ';
    while(ch<'0' || ch>'9'){if(ch=='-')f=-1;ch=getchar();}
    while(ch>='0' && ch<='9'){x=x*10+(ch^48);ch=getchar();}
    return f==1?x:-x;
}
```

```cpp
#define MAXN 1000005

ll a[MAXN]={0},b[MAXN]={0};
struct Node
{
    ll i;
    ll j;
    Node(ll a,ll b){
        i = a;
        j = b;
    }
    bool operator <(Node n) const  {  return a[i]+b[j] < a[n.i]+b[n.j]; }
    bool operator >(Node n) const  {  return a[i]+b[j] > a[n.i]+b[n.j]; }
};
//greater代表小顶堆
priority_queue<Node,vector<Node>,greater<Node>> q;
int main(){
    ll n =read();

    rep(i,1,n){
        a[i]=read();
    }
    rep(i,1,n){
        b[i]=read();
    }
    sort(a+1,a+n+1);
    sort(b+1,b+n+1);
    rep(i,1,n){
        q.push(Node(i,1));
    }
    vector<ll> v;

    while (v.size()<n)
    {
        Node top = q.top();
        q.pop();
        ll i = top.i;
        ll j = top.j;
        v.push_back(a[i]+b[j]);
        q.push(Node(i,j+1));
    }
    for (auto i : v)
    {
        printf("%lld ",i);
    }
    return 0;
}
```

树状数组

单点修改区间查询

```
//单点修改区间查询
#include<iostream>
#include<queue>
#include<vector>
#include<algorithm>
#define rep(i,l,r) for(ll i=(l);i<=(r);++i)
#define rpe(i,r,l) for(ll i=(r);i>=(l);--i)
typedef long long ll;
using namespace std;
inline ll max(ll x,ll y){return x>y?x:y;}
inline ll read(){
    ll x=0,f=1;char ch=' ';
    while(ch<'0' || ch>'9'){if(ch=='-')f=-1;ch=getchar();}
    while(ch>='0' && ch<='9'){x=x*10+(ch^48);ch=getchar();}
    return f==1?x:-x;
}
ll n,m;
ll a[500005]={0},c[500005]={0}; //对应原数组和树状数组

inline ll lowbit(ll x){
    return x&(-x);
}

void add(ll i,ll k){    //在i位置加上k
    while(i <= n){
        c[i] += k;
        i += lowbit(i);
    }
}

ll getsum(ll i){        //求A[1 - i]的和
    ll res = 0;
    while(i > 0){
        res += c[i];
        i -= lowbit(i);
    }
    return res;
}
int main(){
    n = read();
    m = read();
    rep(i,1,n){
        a[i] =read();
        //输入初值的时候，也相当于更新了值
        add(i,a[i]);
    }
    rep(i,1,m){
        int flag = read();
        if(flag == 1){
            ll x = read();
            ll k = read();
            add(x,k);
        }else{
            ll x = read();
```

```
            ll y = read();
            printf("%lld\n",getsum(y)-getsum(x-1));
        }
    }
    return 0;
}
```

区间修改单点查询

```
//区间更改 单点查询
#include<iostream>
#include<queue>
#include<vector>
#include<algorithm>
#define rep(i,l,r) for(ll i=(l);i<=(r);++i)
#define rpe(i,r,l) for(ll i=(r);i>=(l);--i)
typedef long long ll;
using namespace std;
inline ll max(ll x,ll y){return x>y?x:y;}
inline ll read(){
    ll x=0,f=1;char ch=' ';
    while(ch<'0' || ch>'9'){if(ch=='-')f=-1;ch=getchar();}
    while(ch>='0' && ch<='9'){x=x*10+(ch^48);ch=getchar();}
    return f==1?x:-x;
}
ll n,m;
ll a[500005]={0},d[500005]={0}; //对应原数组和树状数组

inline ll lowbit(ll x){
    return x&(-x);
}

void add(ll i,ll k){    //在i位置加上k
    while(i <= n){
        d[i] += k;
        i += lowbit(i);
    }
}

ll getsum(ll i){//求D[1 - i]的和 即A[i]的值
    ll res = 0;
    while(i > 0){
        res += d[i];
        i -= lowbit(i);
    }
    return res;
}

//设 D[i] = A[i] - A[i-1] 则A[i] = sigma(j from 1 to i)D[j]
//则当区间A[X,Y]改变时 D[X+1,Y]不变 只有D[X]和D[Y+1]变
int main(){
    n = read(),m = read();
    rep(i,1,n){
        a[i] = read();
        add(i,a[i]-a[i-1]);
    }
```

```
    rep(i,1,m){
        int flag = read();
        if(flag == 1){
            //[X,Y]区间内加上k
            ll x = read(), y = read();
            ll k = read();
            //A[X]-A[X-1]增加k
            add(x,k);
            //A[Y+1]-A[Y]减少k
            add(y+1,-k);
        }else{
            ll x = read();
            printf("%lld\n",getsum(x));
        }
    }
    return 0;
}
```

区间修改区间查询

```
//区间更改 单点查询
#include<iostream>
#include<queue>
#include<vector>
#include<algorithm>
#define rep(i,l,r) for(ll i=(l);i<=(r);++i)
#define rpe(i,r,l) for(ll i=(r);i>=(l);--i)
typedef long long ll;
using namespace std;
inline ll max(ll x,ll y){return x>y?x:y;}
inline ll read(){
    ll x=0,f=1;char ch=' ';
    while(ch<'0' || ch>'9'){if(ch=='-')f=-1;ch=getchar();}
    while(ch>='0' && ch<='9'){x=x*10+(ch^48);ch=getchar();}
    return f==1?x:-x;
}
ll n,m;
ll a[500005]={0},d[500005]={0}; //对应原数组和树状数组

inline ll lowbit(ll x){
    return x&(-x);
}

void add(ll i,ll k){    //在i位置加上k
    while(i <= n){
        d[i] += k;
        i += lowbit(i);
    }
}

ll getsum(ll i){//求D[1 - i]的和 即A[i]的值
    ll res = 0;
    while(i > 0){
        res += d[i];
        i -= lowbit(i);
    }
```

```
    return res;
}

//设 D[i] = A[i] - A[i-1] 则A[i] = sigma(j from 1 to i)D[j]
//则当区间A[X,Y]改变时 D[X+1,Y]不变 只有D[X]和D[Y+1]变
int main(){
    n = read(),m = read();
    rep(i,1,n){
        a[i] = read();
        add(i,a[i]-a[i-1]);
    }
    rep(i,1,m){
        int flag = read();
        if(flag == 1){
            //[X,Y]区间内加上k
            ll x = read(), y = read();
            ll k = read();
            //A[X]-A[X-1]增加k
            add(x,k);
            //A[Y+1]-A[Y]减少k
            add(y+1,-k);
        }else{
            ll x = read();
            printf("%lld\n",getsum(x));
        }
    }
    return 0;
}
```

二维

```
#include <stdio.h>
#include <string.h>

#define rep(i,l,r) for(int i=(l);i<=(r);++i)
#define rpe(i,r,l) for(int i=(r);i>=(l);--i)

typedef long long ll;

inline ll max(ll x,ll y){return x>y?x:y;}

inline ll read(){
    ll x=0,f=1;char ch=' ';
    while(ch<'0' || ch>'9'){if(ch=='-')f=-1;ch=getchar();}
    while(ch>='0' && ch<='9'){x=x*10+(ch^48);ch=getchar();}
    return f==1?x:-x;
}

#define MAX_N 5000

int tree[MAX_N][MAX_N], n;

// 单点修改 & 区间查询
void add(int x, int y, int z){ //将点(x, y)加上z
    int memo_y = y;
    while(x <= n){
```

```
        y = memo_y;
        while(y <= n)
            tree[x][y] += z, y += y & -y;
        x += x & -x;
    }
}
ll ask(int x, int y){//求左上角为(1,1)右下角为(x,y) 的矩阵和
    ll res = 0, memo_y = y;
    while(x){
        y = memo_y;
        while(y)
            res += tree[x][y], y -= y & -y;
        x -= x & -x;
    }
    return res;
}

// 区间修改 & 单点查询
// 注意此时维护的是差分数组d[i][j] = a[i][j] - a[i-1][j] - a[i][j-1] + a[i-1][j-1]
void add(int x, int y, int z){
    int memo_y = y;
    while(x <= n){
        y = memo_y;
        while(y <= n)
            tree[x][y] += z, y += y & -y;
        x += x & -x;
    }
}
void range_add(int xa, int ya, int xb, int yb, int z){
    add(xa, ya, z);
    add(xa, yb + 1, -z);
    add(xb + 1, ya, -z);
    add(xb + 1, yb + 1, z);
}
ll ask(int x, int y){
    ll res = 0, memo_y = y;
    while(x){
        y = memo_y;
        while(y)
            res += tree[x][y], y -= y & -y;
        x -= x & -x;
    }
    return res;
}

// 区间查询 & 区间修改
// 维护四个数组 O(n*m)
// 有时会爆内存或者超时，注意使用状态压缩
ll t1[MAX_N][MAX_N], t2[MAX_N][MAX_N], t3[MAX_N][MAX_N], t4[MAX_N][MAX_N], m;
void add(ll x, ll y, ll z){
    for(int X = x; X <= n; X += X & -X)
        for(int Y = y; Y <= m; Y += Y & -Y){
            t1[X][Y] += z;
            t2[X][Y] += z * x;
            t3[X][Y] += z * y;
            t4[X][Y] += z * x * y;
        }
}
```

```
void range_add(ll xa, ll ya, ll xb, ll yb, ll z){ //(xa, ya) 到 (xb, yb) 的矩形
    add(xa, ya, z);
    add(xa, yb + 1, -z);
    add(xb + 1, ya, -z);
    add(xb + 1, yb + 1, z);
}
ll ask(ll x, ll y){
    ll res = 0;
    for(int i = x; i; i -= i & -i)
        for(int j = y; j; j -= j & -j)
            res += (x + 1) * (y + 1) * t1[i][j]
                - (y + 1) * t2[i][j]
                - (x + 1) * t3[i][j]
                + t4[i][j];
    return res;
}
ll range_ask(ll xa, ll ya, ll xb, ll yb){
    return ask(xb, yb) - ask(xb, ya - 1) - ask(xa - 1, yb) + ask(xa - 1, ya - 1);
}

int main(){


    return 0;
}
```

线段树

区间加乘区间查询

已知一个数列，你需要进行下面三种操作：

将某区间每一个数乘上 x

将某区间每一个数加上 x

求出某区间每一个数的和

输入格式

第一行包含三个整数 n,m,p分别表示该数列数字的个数、操作的总个数和模数。

第二行包含 n 个用空格分隔的整数，其中第 i 个数字表示数列第 i 项的初始值。

接下来 m 行每行包含若干个整数，表示一个操作，具体如下：

操作 1： 格式：1 x y k 含义：将区间 [x,y] 内每个数乘上 k

操作 2： 格式：2 x y k 含义：将区间 [x,y] 内每个数加上 k

操作 3： 格式：3 x y 含义：输出区间 [x,y] 内每个数的和对 p 取模所得的结果(要求为正)

```
#include <iostream>
#include <cstdio>
```

```
using namespace std;
typedef long long ll;
//题目中给的p
ll p;
//暂存数列的数组
ll a[100007];
//线段树结构体，v表示此时的答案，mul表示乘法意义上的lazytag，add是加法意义上的
struct node{
    ll v, mul, add;
}st[400007];
//buildtree
void bt(ll root, ll l, ll r){
//初始化lazytag
    st[root].mul=1;
    st[root].add=0;
    if(l==r){
        st[root].v=a[l];
    }
    else{
        ll m=(l+r)/2;
        bt(root*2, l, m);
        bt(root*2+1, m+1, r);
        st[root].v=st[root*2].v+st[root*2+1].v;
    }
    st[root].v=(st[root].v+p)%p;
    return ;
}
//核心代码，维护lazytag
void pushdown(ll root, ll l, ll r){
    ll m=(l+r)/2;
//根据我们规定的优先度，儿子的值=此刻儿子的值*爸爸的乘法lazytag+儿子的区间长度*爸爸的加法lazytag
    st[root*2].v=(st[root*2].v*st[root].mul+st[root].add*(m-l+1)+p)%p;
    st[root*2+1].v=(st[root*2+1].v*st[root].mul+st[root].add*(r-m)+p)%p;
//很好维护的lazytag
    st[root*2].mul=(st[root*2].mul*st[root].mul+p)%p;
    st[root*2+1].mul=(st[root*2+1].mul*st[root].mul+p)%p;
    st[root*2].add=(st[root*2].add*st[root].mul+st[root].add+p)%p;
    st[root*2+1].add=(st[root*2+1].add*st[root].mul+st[root].add+p)%p;
//把父节点的值初始化
    st[root].mul=1;
    st[root].add=0;
    return ;
}
//update1，乘法，stdl此刻区间的左边，stdr此刻区间的右边，l给出的左边，r给出的右边
void ud1(ll root, ll stdl, ll stdr, ll l, ll r, ll k){
//假如本区间和给出的区间没有交集
    if(r<stdl || stdr<l){
        return ;
    }
//假如给出的区间包含本区间
    if(l<=stdl && stdr<=r){
        st[root].v=(st[root].v*k+p)%p;
        st[root].mul=(st[root].mul*k+p)%p;
        st[root].add=(st[root].add*k+p)%p;
        return ;
    }
//假如给出的区间和本区间有交集，但是也有不交叉的部分
//先传递lazytag
```

```
    pushdown(root, stdl, stdr);
    ll m=(stdl+stdr)/2;
    ud1(root*2, stdl, m, l, r, k);
    ud1(root*2+1, m+1, stdr, l, r, k);
    st[root].v=(st[root*2].v+st[root*2+1].v+p)%p;
    return ;
}
//update2，加法，和乘法同理
void ud2(ll root, ll stdl, ll stdr, ll l, ll r, ll k){
    if(r<stdl || stdr<l){
        return ;
    }
    if(l<=stdl && stdr<=r){
        st[root].add=(st[root].add+k+p)%p;
        st[root].v=(st[root].v+k*(stdr-stdl+1)+p)%p;
        return ;
    }
    pushdown(root, stdl, stdr);
    ll m=(stdl+stdr)/2;
    ud2(root*2, stdl, m, l, r, k);
    ud2(root*2+1, m+1, stdr, l, r, k);
    st[root].v=(st[root*2].v+st[root*2+1].v+p)%p;
    return ;
}
//访问，和update一样
ll query(ll root, ll stdl, ll stdr, ll l, ll r){
    if(r<stdl || stdr<l){
        return 0;
    }
    if(l<=stdl && stdr<=r){
        return st[root].v;
    }
    pushdown(root, stdl, stdr);
    ll m=(stdl+stdr)/2;
    return (query(root*2, stdl, m, l, r)+query(root*2+1, m+1, stdr, l, r)+p)%p;
}
int main(){
    ll n, m;
    scanf("%lld%lld%lld", &n, &m, &p);
    for(ll i=1; i<=n; i++){
        scanf("%lld", &a[i]);
    }
    bt(1, 1, n);
    while(m--){
        ll chk;
        scanf("%lld", &chk);
        ll x, y;
        ll k;
        if(chk==1){
            scanf("%lld%lld%lld", &x, &y, &k);
            ud1(1, 1, n, x, y, k);
        }
        else if(chk==2){
            scanf("%lld%lld%lld", &x, &y, &k);
            ud2(1, 1, n, x, y, k);
        }
        else{
            scanf("%lld%lld", &x, &y);
```

```
            printf("%lld\n", query(1, 1, n, x, y));
        }
    }
    return 0;
}
```

区间加区间查询

已知一个数列，你需要进行下面三种操作：

将某区间每一个数乘上 x

将某区间每一个数加上 x

求出某区间每一个数的和

输入格式

第一行包含三个整数 n,m,p分别表示该数列数字的个数、操作的总个数和模数。

第二行包含 n 个用空格分隔的整数，其中第 i 个数字表示数列第 i 项的初始值。

接下来 m 行每行包含若干个整数，表示一个操作，具体如下：

操作 1： 格式：1 x y k 含义：将区间 [x,y] 内每个数乘上 k

操作 2： 格式：2 x y k 含义：将区间 [x,y] 内每个数加上 k

操作 3： 格式：3 x y 含义：输出区间 [x,y] 内每个数的和对 p 取模所得的结果(要求为正)

```
#include <iostream>
#include <cstdio>
using namespace std;
typedef long long ll;
//题目中给的p
ll p;
//暂存数列的数组
ll a[100007];
//线段树结构体，v表示此时的答案，mul表示乘法意义上的lazytag，add是加法意义上的
struct node{
    ll v, mul, add;
}st[400007];
//buildtree
void bt(ll root, ll l, ll r){
//初始化lazytag
    st[root].mul=1;
    st[root].add=0;
    if(l==r){
        st[root].v=a[l];
    }
    else{
        ll m=(l+r)/2;
        bt(root*2, l, m);
        bt(root*2+1, m+1, r);
        st[root].v=st[root*2].v+st[root*2+1].v;
    }
    st[root].v=(st[root].v+p)%p;
    return ;
}
//核心代码，维护lazytag
void pushdown(ll root, ll l, ll r){
    ll m=(l+r)/2;
```

```
//根据我们规定的优先度，儿子的值=此刻儿子的值*爸爸的乘法lazytag+儿子的区间长度*爸爸的加法lazytag
    st[root*2].v=(st[root*2].v*st[root].mul+st[root].add*(m-l+1)+p)%p;
    st[root*2+1].v=(st[root*2+1].v*st[root].mul+st[root].add*(r-m)+p)%p;
//很好维护的lazytag
    st[root*2].mul=(st[root*2].mul*st[root].mul+p)%p;
    st[root*2+1].mul=(st[root*2+1].mul*st[root].mul+p)%p;
    st[root*2].add=(st[root*2].add*st[root].mul+st[root].add+p)%p;
    st[root*2+1].add=(st[root*2+1].add*st[root].mul+st[root].add+p)%p;
//把父节点的值初始化
    st[root].mul=1;
    st[root].add=0;
    return ;
}
//update1，乘法，stdl此刻区间的左边，stdr此刻区间的右边，l给出的左边，r给出的右边
void ud1(ll root, ll stdl, ll stdr, ll l, ll r, ll k){
//假如本区间和给出的区间没有交集
    if(r<stdl || stdr<l){
        return ;
    }
//假如给出的区间包含本区间
    if(l<=stdl && stdr<=r){
        st[root].v=(st[root].v*k+p)%p;
        st[root].mul=(st[root].mul*k+p)%p;
        st[root].add=(st[root].add*k+p)%p;
        return ;
    }
//假如给出的区间和本区间有交集，但是也有不交叉的部分
//先传递lazytag
    pushdown(root, stdl, stdr);
    ll m=(stdl+stdr)/2;
    ud1(root*2, stdl, m, l, r, k);
    ud1(root*2+1, m+1, stdr, l, r, k);
    st[root].v=(st[root*2].v+st[root*2+1].v+p)%p;
    return ;
}
//update2，加法，和乘法同理
void ud2(ll root, ll stdl, ll stdr, ll l, ll r, ll k){
    if(r<stdl || stdr<l){
        return ;
    }
    if(l<=stdl && stdr<=r){
        st[root].add=(st[root].add+k+p)%p;
        st[root].v=(st[root].v+k*(stdr-stdl+1)+p)%p;
        return ;
    }
    pushdown(root, stdl, stdr);
    ll m=(stdl+stdr)/2;
    ud2(root*2, stdl, m, l, r, k);
    ud2(root*2+1, m+1, stdr, l, r, k);
    st[root].v=(st[root*2].v+st[root*2+1].v+p)%p;
    return ;
}
//访问，和update一样
ll query(ll root, ll stdl, ll stdr, ll l, ll r){
    if(r<stdl || stdr<l){
        return 0;
    }
    if(l<=stdl && stdr<=r){
```

```
        return st[root].v;
    }
    pushdown(root, stdl, stdr);
    ll m=(stdl+stdr)/2;
    return (query(root*2, stdl, m, l, r)+query(root*2+1, m+1, stdr, l, r)+p)%p;
}
int main(){
    ll n, m;
    scanf("%lld%lld%lld", &n, &m, &p);
    for(ll i=1; i<=n; i++){
        scanf("%lld", &a[i]);
    }
    bt(1, 1, n);
    while(m--){
        ll chk;
        scanf("%lld", &chk);
        ll x, y;
        ll k;
        if(chk==1){
            scanf("%lld%lld%lld", &x, &y, &k);
            ud1(1, 1, n, x, y, k);
        }
        else if(chk==2){
            scanf("%lld%lld%lld", &x, &y, &k);
            ud2(1, 1, n, x, y, k);
        }
        else{
            scanf("%lld%lld", &x, &y);
            printf("%lld\n", query(1, 1, n, x, y));
        }
    }
    return 0;
}
```

ST表

给定一个长度为 N 的数列，和 M 次询问，求出每一次询问的区间内数字的最大值。

输入格式

第一行包含两个整数 N,M，分别表示数列的长度和询问的个数。

第二行包含 N 个整数（记为 a_i），依次表示数列的第 i 项。

接下来 MM 行，每行包含两个整数 l_i,r_i，表示查询的区间为 [l_i,r_i]

ST表的功能很简单

它是解决RMQ问题(区间最值问题)的一种强有力的工具

它可以做到O(nlogn)预处理，O(1)查询最值

```
#include<cstdio>
#include<cmath>
#include<algorithm>
typedef long long ll;
using namespace std;
const ll MAXN=1e6+10;
inline ll read()
```

```
{
    char c=getchar();ll x=0,f=1;
    while(c<'0'||c>'9'){if(c=='-')f=-1;c=getchar();}
    while(c>='0'&&c<='9'){x=x*10+c-'0';c=getchar();}
    return x*f;
}
ll Max[MAXN][21];
ll Query(ll l,ll r)
{
    ll k=log2(r-l+1);
    return max(Max[l][k],Max[r-(1<<k)+1][k]);//把拆出来的区间分别取最值
}
int main()
{
    ll N=read(),M=read();
    for(ll i=1;i<=N;i++) Max[i][0]=read();
    for(ll j=1;j<=21;j++)
        for(ll i=1;i+(1<<j)-1<=N;i++)//注意这里要控制边界
            Max[i][j]=max(Max[i][j-1],Max[i+(1<<(j-1))][j-1]);
    for(ll i=1;i<=M;i++)
    {
        ll l=read(),r=read();
        printf("%d\n",Query(l,r));
    }
    return 0;
}
```

最小生成树

描述

如题，给出一个无向图，求出最小生成树，如果该图不连通，则输出`orz`。

输入

> 第一行包含两个整数 N, M，表示该图共有 N 个结点和 M 条无向边。
>
> 接下来 M 行每行包含三个整数 X_i, Y_i, Z_i，表示有一条长度为 Z_i 的无向边连接结点 X_i, Y_i。

输出

如果该图连通，则输出一个整数表示最小生成树的各边的长度之和。如果该图不连通则输出`orz`

KRUSKAL

```
#include <iostream>
#include <algorithm>
using namespace std;
#define re register
#define il inline
#define ll long long
```

```
il ll read(){
    ll x=0,f=1;char ch=' ';
    while(ch<'0' || ch>'9'){if(ch=='-')f=-1;ch=getchar();}
    while(ch>='0' && ch<='9'){x=x*10+(ch^48);ch=getchar();}
    return f==1?x:-x;
}
#define maxn 5005
// 连通分量个数
ll cnt;
// 存储一棵树
ll parent[maxn]={0};
// 记录树的“重量”
ll weight[maxn]={0};
il void init(ll n) {
    cnt = n;
    for (ll i = 1; i <= n; i++) {
        parent[i] = i;
        weight[i] = 1;
    }
}
il ll find(ll x) {
    while (parent[x] != x) {
        // 进行路径压缩
        parent[x] = parent[parent[x]];
        x = parent[x];
    }
    return x;
}
il void u(ll p, ll q) {
    ll rootP = find(p);
    ll rootQ = find(q);
    if (rootP == rootQ)
        return;
    // 小树接到大树下面，较平衡
    if (weight[rootP] > weight[rootQ]) {
        parent[rootQ] = rootP;
        weight[rootP] += weight[rootQ];
    } else {
        parent[rootP] = rootQ;
        weight[rootQ] += weight[rootP];
    }
    cnt--;
}

il bool connected(ll p, ll q) {
    return find(p) == find(q);
}

//u和v是节点号 w是边权重
struct Edge
{
    ll u,v,w;
}edge[200005];

ll n,m,ans;
ll counter = 0;
//按边权排序
il bool cmp(Edge a,Edge b)
```

```
{
    return a.w<b.w;
}
il void kruskal()
{
    //将边的权值排序
    sort(edge,edge+m,cmp);
    for(re ll i=0;i<m;i++)
    {
        //若出现两个点已经联通了，则说明这一条边不需要了
        if(connected(edge[i].u,edge[i].v))
        {
            continue;
        }
        //将两个点合并到一个集合
        u(edge[i].u,edge[i].v);
        //将此边权计入答案
        ans+=edge[i].w;
        counter++;
        //循环结束条件:边数为点数减一时
        if(counter==n-1)
        {
            break;
        }
    }
}
int main()
{
    n=read(),m=read();
    //初始化并查集
    init(n);
    for(re ll i=0;i<m;i++)
    {
        edge[i].u=read(),edge[i].v=read(),edge[i].w=read();
    }
    kruskal();
    //图联通的话最后对并查集只有一个连通分量
    if(cnt == 1)
    {
        printf("%lld",ans);
    }
    else
    {
        //图不连通输出orz
        printf("orz\n");
    }

    return 0;
}
```

PRIM

```
// 最小子图A问题
// 可以解决重边 负边问题
#include <iostream>
#include <cmath>
using namespace std;
#define re register
#define il inline
#define ll long long
il ll read(){
    ll x=0,f=1;char ch=' ';
    while(ch<'0' || ch>'9'){if(ch=='-')f=-1;ch=getchar();}
    while(ch>='0' && ch<='9'){x=x*10+(ch^48);ch=getchar();}
    return f==1?x:-x;
}
#define inf INT64_MAX
#define maxn 9005
#define maxm 200005
struct edge
{
    ll v,w,next;
}e[maxm<<1];//注意是无向图，开两倍数组
//已经加入最小生成树的点到没有加入的点的最短距离，比如说1和2号节点已经加入了最小生成树，那么dis[3]就等于min(1->3,2->3)
ll head[maxn],dis[maxn],cnt=0,n=0,m=0,tot=0,now=1,ans=0;

bool vis[maxn];

/*
链式前向星加边
前向星是一种特殊的边集数组,我们把边集数组中的每一条边按照起点从小到大排序,如果起点相同就按照终点从小到大排序,并记录下以某个点为起点的所有边在数组中的起始位置和存储长度,那么前向星就构造好了.
用head[i]记录以i为起点的最后一条边的储存位置（下标） 通过赋值为cnt可以看出来
*/
il void add(ll u,ll v,ll w)
{
    e[++cnt].v=v;
    e[cnt].w=w;
    e[cnt].next=head[u];
    head[u]=cnt;
}
//读入数据
il void init()
{
    n=read(),m=read();
    for(re ll i=1,u,v,w;i<=m;++i)
    {
        u=read(),v=read(),w=read();
        add(u,v,w),add(v,u,w);
    }
}
il ll prim()
{
    //先把dis数组附为极大值
    for(re ll i=2;i<=n;++i)
```

```
    {
        dis[i]=inf;
    }
    //head的下标是节点序号 dis的下标是节点序号 e的下标是边序号
    //初始化以第一个节点为起点的所有边的dis
    //通过head[1] 和e[i].next能找到所有以1号节点为起点的边 且是终点从大到小找的
    //这里要注意重边，所以要用到min
    for(re ll i=head[1];i != 0;i=e[i].next)
    {
        dis[e[i].v]=min(dis[e[i].v],e[i].w);
    }
    while(++tot<n)//最小生成树边数等于点数-1
    {
        re ll curmin=inf;//把minn置为极大值
        vis[now]=1;//标记点已经走过
        //枚举每一个没有使用的点
        //找出最小值作为新边
        //注意这里不是枚举now点的所有连边，而是1~n
        for(re ll i=1;i<=n;++i)
        {
            if(!vis[i]&&curmin>dis[i])
            {
                curmin=dis[i];
                now=i;
            }
        }
        ans+=curmin;
        //枚举now的所有连边，更新dis数组
        for(re ll i=head[now];i != 0;i=e[i].next)
        {
            // 该条边两个顶点 now和v
            re ll v=e[i].v;
            if(dis[v]>e[i].w&&!vis[v])
            {
                dis[v]=e[i].w;
            }
        }
    }
    return ans;
}
int main()
{
    init();
    printf("%lld",prim());
    return 0;
}
```

矩阵快速幂（斐波那契数列）

```
#include <iostream>
#include <cstring>

#define Max_rank 3
#define mod 1000000007
struct Matrix {
```

```
    long long a[Max_rank][Max_rank];

    Matrix() {
        memset(a, 0, sizeof(a));
    }

    void init(){
        a[1][1] = a[1][2] = a[2][1] = 1;
        a[2][2] = 0;
    }

    Matrix operator*(const Matrix b) {
        Matrix res;
        for (int i = 1; i <= 2; i++)
            for (int j = 1; j <= 2; j++)
                for (int u = 1; u <= 2; u++)
                    res.a[i][j] = (res.a[i][j] + a[i][u]*b.a[u][j])%mod;
        return res;
    }
};

long long q_pow(long long n){
    Matrix ans,base;
    ans.init();
    base.init();
    while(n > 0){
        if(n&1) ans =ans *base;
        base = base *base;
        n >>= 1;
    }
    return ans.a[1][1];
}
int main() {
    long long n;
    while(std::cin >> n){
        std::cout << q_pow(n-2) << std::endl;
    }
    return 0;
}
```

文艺平衡树

描述

您需要写一种数据结构（可参考题目标题），来维护一个有序数列。

其中需要提供以下操作：翻转一个区间，例如原有序序列是54321，翻转区间是[2,4]的话，结果是52341。

输入

第一行两个正整数n,m，表示序列长度与操作个数。序列中第i项初始为i。

接下来m行，每行两个正整数l,r，表示翻转的区间。

输出

输出一行n个正整数，表示原始序列经过m次变换后的结果。

```
#include<iostream>
using namespace std;
#define MAXN 1000007
#define INF 100000089
struct Splay_tree{
    int f,sub_size,cnt,value,tag;
    int son[2];
}s[MAXN];
int original[MAXN],root,wz;
inline bool which(int x){
    return x==s[s[x].f].son[1];
}
inline void update(int x){
    if(x){
    s[x].sub_size=s[x].cnt;
    if(s[x].son[0])s[x].sub_size+=s[s[x].son[0]].sub_size;
    if(s[x].son[1])s[x].sub_size+=s[s[x].son[1]].sub_size;
    }
}
inline void pushdown(int x){
    if(x&&s[x].tag){
        s[s[x].son[1]].tag^=1;
        s[s[x].son[0]].tag^=1;
        swap(s[x].son[1],s[x].son[0]);
        s[x].tag=0;
    }
}
inline void rotate(int x){
    int fnow=s[x].f,ffnow=s[fnow].f;
    pushdown(x),pushdown(fnow);
    bool w=which(x);
    s[fnow].son[w]=s[x].son[w^1];
    s[s[fnow].son[w]].f=fnow;
    s[fnow].f=x;
    s[x].f=ffnow;
    s[x].son[w^1]=fnow;
    if(ffnow){
        s[ffnow].son[s[ffnow].son[1]==fnow]=x;
    }
    update(fnow);
}
inline void splay(int x,int goal){
    for(int qwq;(qwq=s[x].f)!=goal;rotate(x)){
        if(s[qwq].f!=goal){//这个地方特别重要，原因是需要判断的是当前的父亲有没有到目标节点，而如果把“qwq”改成“x”……就会炸
            rotate(which(x)==which(qwq)?qwq:x);
        }
    }
    if(goal==0){
        root=x;
    }
}

int build_tree(int l, int r, int fa) {
        if(l > r) { return 0; }
```

```
        int mid = (l + r) >> 1;
        int now = ++ wz;
        s[now].f=fa;
        s[now].son[0]=s[now].son[1]=0;
        s[now].cnt++;
        s[now].value=original[mid];
        s[now].sub_size++;
        s[now].son[0] = build_tree(l, mid - 1, now);
        s[now].son[1] = build_tree(mid + 1, r, now);
        update(now);
        return now;
}
inline int find(int x){
    int now=root;
    while(1)
    {
        pushdown(now);
        if(x<=s[s[now].son[0]].sub_size){
            now=s[now].son[0];
        }
        else  {
        x-=s[s[now].son[0] ].sub_size + 1;
        if(!x)return now;
        now=s[now].son[1];
        }
    }
}
inline void reverse(int x,int y){
    int l=x-1,r=y+1;
    l=find(l),r=find(r);
    splay(l,0);
    splay(r,l);
    int pos=s[root].son[1];
    pos=s[pos].son[0];
    s[pos].tag^=1;
}
inline void dfs(int now){
    pushdown(now);
    if(s[now].son[0])dfs(s[now].son[0]);
    if(s[now].value!=-INF&&s[now].value!=INF){
        cout<<s[now].value<<" ";
    }
    if(s[now].son[1])dfs(s[now].son[1]);
}
int main(){
    int n,m,x,y;
    cin>>n>>m;
    original[1]=-INF,original[n+2]=INF;
    for(int i=1;i<=n;i++){
        original[i+1]=i;
    }
    root=build_tree(1,n+2,0);
    for(int i=1;i<=m;i++){
        cin>>x>>y;
        reverse(x+1,y+1);
    }
    dfs(root);
}
```

网络流

EK

```
#include <bits/stdc++.h>
using namespace std;
int n,m,s,t,u,v;
long long w,ans,dis[520010];
int tot=1,vis[520010],pre[520010],head[520010],flag[2510][2510];

struct node {
    int to,net;
    long long val;
} e[520010];

inline void add(int u,int v,long long w) {
    e[++tot].to=v;
    e[tot].val=w;
    e[tot].net=head[u];
    head[u]=tot;
    e[++tot].to=u;
    e[tot].val=0;
    e[tot].net=head[v];
    head[v]=tot;
}

inline int bfs() {  //bfs寻找增广路
    for(register int i=1;i<=n;i++) vis[i]=0;
    queue<int> q;
    q.push(s);
    vis[s]=1;
    dis[s]=2005020600;
    while(!q.empty()) {
        int x=q.front();
        q.pop();
        for(register int i=head[x];i;i=e[i].net) {
            if(e[i].val==0) continue;  //我们只关心剩余流量>0的边
            int v=e[i].to;
            if(vis[v]==1) continue;  //这一条增广路没有访问过
            dis[v]=min(dis[x],e[i].val);
            pre[v]=i;  //记录前驱，方便修改边权
            q.push(v);
            vis[v]=1;
            if(v==t) return 1;  //找到了一条增广路
        }
    }
    return 0;
}

inline void update() {  //更新所经过边的正向边权以及反向边权
    int x=t;
    while(x!=s) {
        int v=pre[x];
```

```
        e[v].val-=dis[t];
        e[v^1].val+=dis[t];
        x=e[v^1].to;
    }
    ans+=dis[t];   //累加每一条增广路经的最小流量值
}

int main() {
    scanf("%d%d%d%d",&n,&m,&s,&t);
    for(register int i=1;i<=m;i++) {
        scanf("%d%d%lld",&u,&v,&w);
        if(flag[u][v]==0) {  //处理重边的操作（加上这个模板题就可以用Ek算法过了）
            add(u,v,w);
            flag[u][v]=tot;
        }
        else {
            e[flag[u][v]-1].val+=w;
        }
    }
    while(bfs()!=0) {  //直到网络中不存在增广路
        update();
    }
    printf("%lld",ans);
    return 0;
}
```

DINIC

题目描述

如题，给出一个网络图，以及其源点和汇点，求出其网络最大流。

输入格式

第一行包含四个正整数 n,m,s,tn,m,s,t，分别表示点的个数、有向边的个数、源点序号、汇点序号。

接下来 m 行每行包含三个正整数 u_i,v_i,w_i，表示第 i 条有向边从 u_i出发，到达 v_i，边权为 w_i

输出格式

一行，包含一个正整数，即为该网络的最大流。

```
// DINIC算法

#include <bits/stdc++.h>
using namespace std;
typedef long long LL;

const int N = 10010, E = 200010;

int n, m, s, t; LL ans = 0;
LL cnt = 1, first[N], nxt[E], to[E], val[E];
inline void addE(int u, int v, LL w) {
    to[++cnt] = v;
    val[cnt] = w;
    nxt[cnt] = first[u];
    first[u] = cnt;
}
int dep[N], q[N], l, r;
```

```cpp
bool bfs() {//顺着残量网络，方法是 bfs；这是个bool型函数，返回是否搜到了汇点
    memset(dep, 0, (n + 1) * sizeof(int));//记得开局先初始化

    q[l = r = 1] = s;
    dep[s] = 1;
    while(l <= r) {
        int u = q[l++];
        for(int p = first[u]; p; p = nxt[p]) {
            int v = to[p];
            if(val[p] and !dep[v]) {//按照有残量的边搜过去
                dep[v] = dep[u] + 1;
                q[++r] = v;
            }
        }
    }
    return dep[t];//dep[t] != 0，就是搜到了汇点
}
LL dfs(int u, LL in/*u收到的支持（不一定能真正用掉）*/) {
//注意，return 的是真正输出的流量
    if(u == t)
        return in;//到达汇点是第一个有效return
    LL out = 0;
    for(int p = first[u]; p and in; p = nxt[p]) {
        int v = to[p];
        if(val[p] and dep[v] == dep[u] + 1) {//仅允许流向下一层
            LL res = dfs(v, min(val[p], in)/*受一路上最小流量限制*/);
            //res是v真正输出到汇点的流量
            val[p] -= res;
            val[p ^ 1] += res;
            in -= res;
            out += res;
        }
    }
    if(out == 0)//我与终点（顺着残量网络）不连通
        dep[u] = 0;//上一层的点请别再信任我，别试着给我流量
    return out;
}
int main() {
    scanf("%d %d %d %d", &n, &m, &s, &t);
    for(int i = 1; i <= m; ++i) {
        int u, v; LL w;
        scanf("%d %d %lld", &u, &v, &w);
        addE(u, v, w);
        addE(v, u, 0);
    }
    while(bfs())
        ans += dfs(s, 1e18);
    printf("%lld\n", ans);
    return 0;
}
```