stl

标准模板库，即标准模板库。虽然一般说起来 stl 只能想起 `vector` 之类的，但是 stl 实际上包括了容器（和容器适配器）、算法、迭代器、仿函数、内存分配器五个。
所有 stl 都在 `std` 命名空间里，使用时要使用此命名空间 `using namespace std;` 或者使用这个命名空间里的某个东西比如 `using std::vector` 或者在使用时附加 `std::` 比如 `std::vector<int>v;`。另外，如果可以 adl 也可以不写 `std`，后面的其他有关命名空间也是同理，不作阐述了。

容器

封装的数据结构，都具有某些相同的成员函数和非成员函数。

```
\begin{array}{|c|c|c|c|c|}
\hline
begin & \quad & \quad & \cdots & \quad
\\
\hline
\end{array}
~end
```

```
rend~
\begin{array}{|c|c|c|c|c|}
\hline
\quad & \quad & \cdots & \quad & rbegin
\\
\hline
\end{array}
```

- `begin`：返回起始位置的迭代器。
- `rbegin`：返回起始位置的反向迭代器。
- `end`：返回结束位置的迭代器。（左闭右开，实际上越界）
- `rend`：返回结束位置的反向迭代器。（左闭右开，实际上越界）
- `size`：返回容器中的元素数量，注意类型是 `size_t`，属于 `unsigned`。
- `empty`：返回容器是否是空。
- `clear`：清空容器。

顺序容器

顺序容器是能按顺序访问的数据结构。

- `insert`：在迭代器前面插入元素。
- `erase`：删除这个迭代器，返回后面那个元素。

vector

需要 `#include<vector>`。
保证 a_i 的地址一定和 a_{i+1} 是连续的。可以实现动态扩容。具有 $\mathcal{O}(1)$ 的随机访问，均摊 $\mathcal{O}(1)$ 的在尾部添加一个元素，非常小常数 $\mathcal{O}(n)$ 的在内部插入或删除元素（n 为插入删除的位置距离 `vector` 尾部的距离）。
内部实现是储存 3 个指针，第一个指针与第三个元素储存了一段连续的内存空间的开始和结束。第二个指针表示 `vector` 里面最后一个元素的位置。记录三个指针为 `p1` `p2` `p3`。

当你在 vector 末尾插入元素时，会先判断插入元素后 vector 有没有满，就是说把 p2 增加后是否超过第三个指针，如果没有超过就直接自增 p2，否则就要对 vector 进行重构。设原内部元素数量为 n，重构会新开辟一段长度为 $2n$ 的空间，把原来那些元素移动拷贝到这段新的空间，然后把原来那段空间释放掉。
来计算一下在末尾插入均摊的复杂度，vector 中共插入有 n 个元素，由机制可知里面容量最多是 $2n$，那么每次扩容转移就是 $1+2+4+\cdots+n+2n<4n$ ，插入 n 个元素最多也只要 $\mathcal{O}(4n)$ 的时间，均摊下来是 $\mathcal{O}(1)$ 了。
因为 vector 只存了 3 个指针，所以交换 2 个 vector 写了特化，复杂度 $\mathcal{O}(1)$。
vector 具有成员函数 shrink_to_fit，由 vector 的机制可以知道 p2 到 p3 有一段预分配的空间预留下来应对可能的末端操作，如果不去用就被浪费了，shrink_to_fit 可以释放掉这些空间，新开一段长度为 p1 到 p2 的空间把现在这些元素移动拷贝进去。但是 shrink_to_fit 是否会实现是由定义实现的，谁知道会怎么搞呢?
有的人认为 vector 的成员函数 clear 清空是 $\mathcal{O}(1)$ 的，但是并不是这样的。因为要把里面所有元素析构一次，所以 vector clear 的时间和里面元素的数量是成正比的。但是无所谓，均摊到插入上还是 $\mathcal{O}(1)$ 的。如果类型是基础类型编译器可能还会把 clear 优化成常数。由 vector 的内存分配可知 clear 只会去析构元素而不是释放
使用 vector 时容易产生迭代器失效或者改变内容，胡乱使用会 WA 甚至 RE。
swap 会让 end 迭代器失效。clear operator= assign 全部迭代器失效。reserve shrink_to_fit 如果更改容量则全部失效。erase 使删除点及之后的所有迭代器失效。push_back emplace_back 如果更改容量全部迭代器失效否则仅 end 失效。insert emplace 如果更改容量全部迭代器失效否则仅插入点及插入点后元素失效。resize 如果更改容量全部迭代器失效否则仅 end 与被删除元素失效。pop_back 仅被删除元素和 end 迭代器失效。
vector<bool>，什么，这不是 container，这是一个压缩 bool block。我们这个压缩 bool block 体积小方便存储，声明一个，push_back 会变大，怎么 push_back 都 push_back 不满，用来压位，压位，压位都是很好用的，你看内存分布像满天繁星一样，放在电脑里 push_back 变多变乱，碎片很多的。operator[] 以后，是一条变小变 safety 的 vector<bool>::reference，你看它怎么引用都引用不出来，不可修改随时销毁，使用 10^8 次都没问题，赛时切题带上它非常方便，用它 auto&，再 range_based_for，
allocator_traits::construct，world leg tree。什么？在哪里写？下方 Dev-c++，写五次 CE 五次，还包 WA。
省流：想要节省空间就用 bitset，否则可以使用 basic_string<bool> 或者 vector<char> 或者手写。
hash 提供了 vector<bool> 的哈希支持。成员函数 flip 可以翻转里面的所有位。

array

需要 #include<array>。
是对数组的封装，该结构体结合了 C 风格数组的性能、可访问性与容器的优点，比如可获取大小、支持赋值、随机访问迭代器等。
array 的 swap 不是 $\mathcal{O}(1)$ 的，而是和 array 的大小正比。
array 的只要不被析构，迭代器就永远不会失效。
可以采用成员函数 data 返回底层数组上的头指针。函数 make_array 可以按其中元素的数量构造一个 array。函数 to_array 可以用数组创建一个 array。

deque

需要 #include<deque>。
你说得对，但是我们这个 deque 体积小方便携带，拆开一包，放评测机里就变大，怎么放都放不下，用来打 CSP，打 NOIP ，打 NOI 都是很好用的，你看打开以后像 1KB一样大小，放在评测机里遇水变大变高，占空间很大的。打开以后，是一个加大加长的双端队列，你看他怎么开都开不下，使用七八次都没问题，出差打比赛

带上它非常方便。
什么？在哪里买？下方 @NOI2022 D1T1 出题人，买 1 GB送一块铁牌，还包邮。
`deque` 具有 $\mathcal{O}(1)$ 的首位插入删除，较大常数的 $\mathcal{O}(1)$ 随机访问，$\mathcal{O}(n)$ 的插入删除（n 为插入删除的位置距离 `deque` 头部尾部较近的那个距离）。
`deque` 采取了类似分块的方式保存。用类似 `vector` 的方式开辟一段内存连续的中控器，每个中控器存指针，记录一段内存连续的缓冲区（缓冲区的大小取决于编译器的实现，例如 64 位 libstdc++ 上是对象尺寸的 8 倍；64 位 libc++ 上是对象尺寸的 16 倍和 4096 字节中的较大者）。在末尾插入元素时，看看最后一个中控器管理的缓冲区满了没，要是没满直接插入，满了就新开辟一个缓冲区，在中控器末端加入指针去管理这片缓冲区就行了。
这么做虽然可以方便在首位添加删除元素，但是也有很大的副作用。`array` 通过头指针的直接加减就可以，0 个中间商；`vector` 要经过那 3 个指针，1 个中间商，`deque` 要到中控器再到缓冲区，2 个中间商，常数更大。而且 `deque` 这种记录方式需要消耗大量的内存空间。
`deque` 的储存方式复杂，迭代器失效也复杂。`swap` 能使 `end` 迭代器失效，`shrink_to_fit` `clear` `operator=` `assign` 所有迭代器和引用都会失效。`push_front` `push_back` `emplace_front` `emplace_back` 会使所有迭代器和引用失效。对于 `insert` 和 `emplace`，一定会使迭代器失效，除非在 首尾操作。`erase` 如果在首部，只有被删除迭代器失效，如果在尾部是删除点及 `end` 迭代器失效，否则所有迭代器失效。`resize` 如果新尺寸小于旧尺寸，只有被删除元素和尾后迭代器失效 如果新尺寸大于旧尺寸，所有迭代器失效，否则没有失效。
`pop_front` 只有被删除元素失效，`pop_back` 被删除元素和 `end` 失效。

list

需要 `#include<list>`。
`list` 具有 $\mathcal{O}(1)$ 的首尾插入删除，$\mathcal{O}(n)$ 随机访问，$\mathcal{O}(1)$ 的插入删除。
一般采取的实现就是双向链表，没什么好讲的。
高贵的链表优点是插入永远不会使任何迭代器失效，删除只会使被删除元素的迭代器删除。
为了保证链表的每个节点的迭代器不被失效，所以 `list` 自带些函数，能够归并两个有序链表的 `merge`，能够把另一个链表的所有节点移动到这个链表的 `splice`，能够翻转所有节点的 `reverse`，排序的 `sort`，去重的 `unique`。

forward_list

需要 `#include<forward_list>`。
`forward_list` 具有 $\mathcal{O}(1)$ 的首插入删除，$\mathcal{O}(n)$ 随机访问，$\mathcal{O}(1)$ 的插入删除。
一般采取的实现就是单向链表，没什么好讲的。
高贵的链表优点是插入永远不会使任何迭代器失效，删除只会使被删除元素的迭代器删除。
为了保证链表的每个节点的迭代器不被失效，所以 `forward_list` 自带些函数，能够归并两个有序链表的 `merge`，能够把另一个链表的所有节点移动到这个链表的 `splice_after`，能够翻转所有节点的 `reverse`，排序的 `sort`，去重的 `unique`。

关联容器

能够实现 $\mathcal{O}(\log n)$ 查找的数据结构，不过这些容器里面的提供的迭代器都是只读迭代器，所以想要修改里面的元素只能依靠删除再插入，或者是 `mutable`。

set

需要 `#include<set>`。
具有 $\mathcal{O}(\log n)$ 的插入查找删除。

通常采取红黑树维护，里面的元素是有序的，不过由于 set 的实现元素是不可重的，这里如果 a 不小于 b 且 b 不小于 a 就表示 a 和 b 是重复的。
迭代器是指节点的，表示 set 的迭代器失效情况和 list 是差不多的。
emplace_hint 的效果是在插入元素的时候提示一个迭代器，如果提示的对，新元素正好插在提示迭代器之前，复杂度就是均摊 $\mathcal{O}(1)$ 的。
iterator 的 operator++ 和 operator-- 最坏复杂度是 $O(\log n)$，均摊复杂度（把整个 set 遍历一遍）是 $O(1)$。 算法中 lower_bound 和 upper_bound 那两个函数对 vector 这种迭代器支持随机访问的才可以有 $\mathcal{O}(\log n)$ 复杂度。 set 迭代器只会 operator++ operator-- 的，复杂度就不对了，复杂度为线性。 与后面的无序关联容器都有相同的成员函数。

- insert：在容器中插入元素。
- erase：在容器中插入元素或者迭代器指向的元素。
- find：返回查找元素的迭代器，没有就返回 end。
- count：返回查找元素的数量。
- contains：返回有无这个元素。
- merge：把另一个关联容器的所有节点放到这个里面，迭代器不失效。
- extract：返回键为给定值的结点柄。
- try_emplace：试图在里面插入给定键的元素，如果有了就跳过。
- swap：$\mathcal{O}(1)$ 交换。

multiset

需要 #include<set>。
和 set 基本一样，但是可以插入多个相同的元素，可以看成是 set 插入以后带入了个时间戳。
所以 try_emplace 被羊了。
注意到：记 c 为容器中某个键的出现个数，那么 count 的复杂度为 $O(c+\log n)$，contains 的复杂度为 $O(\log n)$，所以用 count 小心被卡。

map

需要 #include<map>。
和 set 基本一样，但是插入的东西可以带一个对应的值（需要在定义的时候指定键值的类型）。我们把插入的元素叫做键（key），元素对应的值叫做键值（value）。可以通过成员函数 at 或 operator[] 调出键值。单次复杂度为 $O(\log n)$。
注意到 operator[] 会返回一个迭代器的引用，所以会进行一个键的插入。

multimap

需要 #include<map>。
和 map 基本一样，但是因为同一个键值可以出现多次，所以 operator[] 和 at 被羊了。

无序关联容器

能够实现随机情况下 $\mathcal{O}(1)$ 插入、删除、查找的数据结构，内部实现显然是哈希。共同成员函数如上。
建议阅读：Blowing up unordered_map, and how to stop getting hacked on it
省流：126271 (GCC 6-)，107897 (GCC 7+) 及其倍数能卡死这一系列的容器。要不被卡，可以加或者异或一个 FIXED_RANDOM，就不太会被卡了。

unordered_set

需要 `#include<unordered_set>`。
用法类 `set`。不过插入导致重哈希会导致迭代器失效。

unordered_multiset

需要 `#include<unordered_set>`。
用法类 `multiset`。

unordered_map

需要 `#include<unordered_map>`。
用法类 `map`。不过插入导致重哈希会导致迭代器失效。

unordered_multimap

需要 `#include<unordered_map>`。
用法类 `multimap`。

容器适配器

容器适配器实际上没写什么代码，需要提供一个容器，容器适配器只是对这个容器的包装。

stack

需要 `#include<stack>`。
相当于只会 `push_back` 和 `pop_back`（也就是 LIFO）并且没有迭代器的 `deque`。内部默认实现也确实是 `deque`。

queue

需要 `#include<queue>`。
相当于只会 `push_back` 和 `pop_front`（也就是 FIFO）并且没有迭代器的 `deque`。内部默认实现也确实是 `deque`。

priority_queue

需要 `#include<queue>`。
提供了 $\mathcal{O}(1)$ 的最值元素查找，$\mathcal{O}(\log n)$ 的插入元素和删除最值元素。
默认实现是 `vector` 实现的二叉堆，不过由于实现问题默认反而是大根堆，也就是说这个的 cmp 是反一下的。

字符串库

虽然 `basic_string` 并不算是 stl，但是和 stl 高度重合，也有迭代器内存分配器什么的，和算法库也能通用，就在这里讲一下吧。

basic_string

需要 #include<string>。
内存连续，动态开空间方式类似于 vector。里面要放平凡类型（即没有构造函数的类型）。
basic_string 为了兼容 c 的字符串，可以使用成员函数 c_str 返回头指针，而且绝对能传递出空终止（即 end 的返回值为 0），这是 vector 做不到的。
把一个 basic_string 复制到另一个后面直接使用 opetraor+=，在末端插入元素可以使用 operator+，不过删除末端元素也只能老实 pop_back，重载了字面量 operator""s 把字符数组字面量转为 basic_string。另外，basic_string 重载的所有运算符都重载了对应的字符数组类型。就算 b 是 T* a 是 basic_string<T> 那么 a+=b 是没问题的。
如果你比较好奇去比较 vector 和 basic_string 的空间消耗会发现 basic_string 需要 32 字节，而 vector 只要 24 字节，怎么回事呢?
有人认为 c++ 的 basic_string 效率低下，需要额外进行一次指针解引用。所以 c++ 提供了小字符串优化。如果这个字符串本身长度较长（超过 24 个字节），那么采用 vector 的储存方式，也就是 3 个指针共 24 字节。否则仅采用 1 个指针，用 24 字节在**栈**上开内存。由于一个 basic_string 不可能同时两种储存方式，所以使用 union 绑在一起，内存消耗是 $\max(3\times 8, 8+24)=32$ 字节。
string 完全等价于 basic_string<char>。

basic_string_view

需要 #include<string_view>。
给 basic_string 准备的视图，不开辟内存空间，只记录头指针和长度，由于没储存什么东西，所以建议值传递。在 OI 界用处不大。
重载了字符数组字面量 operator""sv 用来表示字符串视图。
string_view 完全等价于 basic_string_view<char>。

杂七杂八

由于过于杂七杂八，所以~~值得我们足足半日的停王缅怀~~特地分一块。

span

需要 #include。
c++20 的贵物。本身什么也没有实现，是一段视图用来表示一段连续的内存比如 array vector 之类的，使用时可以有静态范围和动态范围（本质上是记录头指针和长度，静态范围表示长度是编译期常量），用来弥补顺序容器没有视图的缺陷，由于没储存什么东西，所以建议值传递。在 OI 界用处不大。

mdspan

c++23，高级。是一段视图用来表示一段连续的内存比如 array vector 之类的，但是会把这段连续的内存解释为多维数组。定义 mdspan 后面可以带上一串数字，每个数字表示每个维度的大小。由于没储存什么东西，所以建议值传递。
在 OI 界中，如地图只给出 n,m 的时候就可以 mdspan(a,n+1,m+1) 把静态一维数组包装成二维数组。调用的时候是用到 operator[] 的，c++23 的 operator[] 里面可以放多个形参。比如 b[2,3]。

valarray

需要 #include<valarray>。
能够把所有元素一直操作的贵物。是用两个指针记录一段内存，所以内部元素连续，但是缺点是不能在末尾插入元素但是可以 resize。

特点是可以把所有元素一起操作，比如 a+=c/2 相当于对于每个 i a[i]+=c[i]/2，其他运算也支持，还有 pow sin log 这种数学函数。成员函数 swap 特化是 $\mathcal{O}(1)$ 的，sum 返回所有元素总和，min max 用来返回最值，shift 是位移元素，cshift 就是循环位移了，apply 对所有元素执行函数。
operator[] 并不紧紧能返回第几个元素的引用，比如 a 是 valarray，那么 a>4 就是 mask_array，这个时候调用 a[a>4]++ 就相当于对每个下标大于 4 的元素自增。还可以在里面塞 slice，这个 slice 就类似于 Python 里面的切片，slice(a,b,c) 就相当于下标为 $i\in[0,b-1),a+ic$ *的一起操作。更加变态的是* gslice，*gslice(a,b,c) 就相当于下表为 $i\in n,a+b_ic_i$* 的一起操作，注意 b c 都是 valarray，如果 gslice 算出相同下标且进行修改是未定义行为。里面还可以放 valarray<bool> 表示每位是否可行。里面还可以放 valarray<size_t> 表示应该才做的下标。
注意：valarray 本质上就是一个循环，不会把本体复制一份进行运算，某些对于本体的非顺序修改会使结果错误，慎用，或者说别用 valarray，答应我，别用 valarray 好吗?

bitset

需要 #include<bitset>。
不是贵物，但是也快算是贵物了。valarray 虽好但是效率并不高可能还不如 vector 手写循环了，更别说对于 bool 类型不能压缩空间或者增加效率，vector<bool> 压了空间却失去了效率，bitset 就挺身而出了。
对于 bitset 是编译器确定长度的，后期无法插入删除 resize，8 位压一个字节，基本位运算可以全部一起操作，但是不带 operator+ operator< 之类的，成员函数 all any none 可以返回所有，任一或没有位为 true，count 可以返回 true 的数量，可以使用 set reset flip 可以把指定位 true false 或翻转。
由于 bitset 压了位所以不能对每位直接取引用，而是通过成员类 reference 代理，对 reference 的读写会潜在读写底层 bitset。
另外，hash 提供了 bitset 的哈希支持，bitset 不存在 swap 特化。

initializer_list

分布很杂，许多头文件里都包括了这个。
没有用，初始化列表（不是成员初始化器列表），你想写构造函数大概就要写这个了，不允许随机访问，但是给了你迭代器，底层是临时只读数组。

pair

分布很杂，许多头文件里都包括了这个。
将两个类型打包成一个，第一个叫做 x.first，第二个叫做 x.second，比较方式是从第零个开始往后比较。

tuple

分布很杂，许多头文件里都包括了这个。
将 n 个类型打包成一个，第 i 个叫做 get<i>(x)，比较方式是从第零个开始往后比较。
具有把空类型优化内存的空间，比如 unique_ptr 包括指针和空类型析构器，如果直接封装由于内存对齐和不允许空类型需要 16 字节，但是 tuple 奇妙优化只要 8 字节。

complex

需要 #include<complex>。
实现的复数类（只有 float double long double 三种类型是明确可以实例化），效率不高，想要效率可以使用对应类型的 pair 重载。
real 返回实部，imag 返回虚部。非成员函数 abs 返回模，arg 返回辐角，conj 返回复共轭，polar 用模和辐

角构造复数。另外 `pow sin` 也是可以的。
`operator<<,>>` 可以输入输出复数，格式是 `(real,imag)`。
重载了字面量 `operator""if,i,il` 表示纯虚数的 `float double long double`。

算法

算法就是一堆写完的函数模板，由于太多了，我就讲几个比较常用的，以下的复杂度均指内存连续的情况。
要用就直接 `#include<algorithm>`。

- `swap`：交换两个元素，对某些容器特化。
- `reverse`：翻转一段元素，$\mathcal{O}(n)$。
- `sort`：排序一段元素，$\mathcal{O}(n\log n)$。
- `is_sort`：检查一段元素是否有序，$\mathcal{O}(n)$。
- `unique`：去重一段元素，$\mathcal{O}(n)$。
- `nth_element`：获得一段元素第 n 项，$\mathcal{O}(n)$
- `min_element`：求一段元素最小值的迭代器，$\mathcal{O}(n)$。
- `max_element`：求一段元素最大值的迭代器，$\mathcal{O}(n)$。
- `fill`：推平一段元素，$\mathcal{O}(n)$。
- `count`：计算一段元素某个值的数量，$\mathcal{O}(n)$。
- `shuffle`：打乱一段元素，$\mathcal{O}(n)$。
- `lower_bound`：有序序列找到第一个大于等于目标元素的迭代器，$\mathcal{O}(\log n)$。
- `upper_bound`：有序序列找到第一个大于目标元素的迭代器，$\mathcal{O}(\log n)$。
- `iota`：用给定的初值自增填充一段元素，$\mathcal{O}(n)$。
- `accumulate`：求给定一段元素的和，$\mathcal{O}(n)$。
- `next_permutaion`：使给定元素变为下个全排列，$\mathcal{O}(n)$。
- `prev_permutaion`：使给定元素变为上个全排列，$\mathcal{O}(n)$。

好像，其实也不是很多很常用的?

迭代器

虽然需要 `#include<iterator>` 但是已经被包括在别的里面了。
迭代器有 6 种，输入输出向前双向随机连续，还分为老式的，c++20 给出了另外一种更加复杂的迭代器分类方式，何况，你知道那么多迭代器你也没用啊QwQ。只要知道迭代器能够 `operator++` 访问下个位置，有些可以 `operator--` 访问上一个元素，有的可以直接快速多次 `operator++ operator--`，想要解引用就直接 `operator*` 就行了。另外，想要访问迭代器 `it` 的下一个位置且不改变 `it` 可以采用 `next(it)`，上一个位置且不改变 `it` 可以采用 `prev(it)`。

仿函数

相信大家都知道函数是什么，但是也许不知道什么是仿函数。普通的函数是一块内存地址，函数名是指针指向该地址的指针，看着很好但是也有问题。众所周知 c++ 有函数模板，我需要传递一个函数模板的某个特定实例化（比如传递给类模板）就特别麻烦，函数模板不实例化就没地址，需要先实例化，再传递实例化的指针。所以需要使用仿函数（又名函数对象）。
直接 `#include<algorithm>` 就可以了。 仿函数是一个具有成员函数 `operator()` 的类模板，仿函数定义的变量就可以 `operator()` 执行“函数”了。
由于仿函数往往也是模板可以和类同时实例化，并且可以依赖、组合与继承，所以在 stl 中被大量使用。比如 stl 中的 `set priority_queue` 自定义比较顺序就用到了仿函数。

常用的比如 `less greater` 可以 `sort(all(b),greater<>())` 来调用，在 `set` 里面就是 `set<int,std::greater<>>` 了，尖括号里可以放对应的类型，也可以空着来自动推导。
就像 c 中的函数指针可以指向函数一样，c++ 提供了 `function` 来保存仿函数，需要 `#include<function>`，在 OI 界 useless。
另外 `hash` 也是个仿函数，提供了把某些类型转化为 `size_t` 的哈希值，`unordered_set` 的默认哈希就是这个（卡哈希超时警告）。

内存分配器

讲了也白讲系列。

pbds

著名的平板电视。主要是为了里面的正常平衡树（不是 stl 的阉割无法求排名的 `set`）。
就像 stl 的命名空间都在 `std` 里，pbds 的命名空间都在 `__gnu_pbds` 里面。由于这个命名空间名字太长，我直接一句 `namespace pbds=__gnu_pbds;`。
注意 pbds 中的 `tree` 和 `hash_table` 是类似 `map` 或 `unordered_map` 的。如果想实现 `set` 或 `unordered_set` 就要把第二维设定为 `null_type`。

hash_table

哈希表。
需要 `#include<ext/pb_ds/assoc_container.hpp>`。 `cc_hash_table` 是拉链法哈希。
`gp_hash_table` 是探测法哈希（一般更高效）。

tree

需要 `#include<ext/pb_ds/assoc_container.hpp>` 和 `#include<ext/pb_ds/tree_policy.hpp>`。
平衡树，可以采用 `rb_tree_tag`（红黑树，最高效） 或者 `splay_tree_tag`（伸展树，效率一般）或者 `ov_tree_tag`（贵物，慢）。
具有两种节点更新方式 `null_node_update` 和 `tree_order_statistics_node_update`，默认为 `null_node_update`，`null_node_update` 和 stl 中 `set` 类似，如果是 `tree_order_statistics_node_update` 就可以享受 `find_by_order`（给定排名返回迭代器）和 `order_of_key`（给的值返回排名）。这就是 `tree` 对于 `set` 的优点。在 NOIP 考纲中，这足以应付所有的平衡树题了（插入删除查找排名）~~学什么 Treap Splay AVL 替罪羊树啊~~。
对于更复杂的平衡树（比如文艺平衡树），`tree` 就无法或者很难维护了，所以如果你要~~薄纱 ZJOI NOI 怒砍 Au AK IOI~~那还是老实点去学 Treap Splay AVL 替罪羊树了。
所以我到现在还不会写平衡树（这就是你菜的理由？），`tree` 的常数不大，比我自己贺的指针 FHQ Treap 还快！

priority_queue

需要 `#include<ext/pb_ds/priority_queue.hpp>`。
优先队列，但是 stl 里的是二叉堆，pbds 里面有配对堆（`pairing_heap_tag`）、二叉堆（`binary_heap_tag`）、二项堆（`binomial_heap_tag`）、冗余计数二项堆（`rc_binomial_heap_tag`）、改良版斐波那契堆（`thin_heap_tag`）。不过无所谓，知道默认的配对堆最快就行了。

名称	push	pop	modify	erase	join
priority_queue	最坏 $\mathcal{O}(n)$，均摊 $\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	不开放	不开放	$\mathcal{O}(n)$
手写	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
pairing_heap_tag	$\mathcal{O}(1)$	最坏 $\mathcal{O}(n)$，均摊 $\mathcal{O}(\log n)$	最坏 $\mathcal{O}(n)$，均摊 $\mathcal{O}(\log n)$	最坏 $\mathcal{O}(n)$，均摊 $\mathcal{O}(\log n)$	$\mathcal{O}(1)$
binary_heap_tag	最坏 $\mathcal{O}(n)$，均摊 $\mathcal{O}(\log n)$	最坏 $\mathcal{O}(n)$，均摊 $\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
binomial_heap_tag	最坏 $\mathcal{O}(\log n)$，均摊 $\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
rc_binomial_heap_tag	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
thin_heap_tag	$\mathcal{O}(1)$	最坏 $\mathcal{O}(n)$，均摊 $\mathcal{O}(\log n)$	最坏 $\mathcal{O}(\log n)$，均摊 $\mathcal{O}(1)$	最坏 $\mathcal{O}(n)$，均摊 $\mathcal{O}(\log n)$	$\mathcal{O}(n)$

这玩意也有迭代器失效的，`binary_heap_tag` 修改后迭代器会失效，别的怎么搞也不会迭代器失效，和 `set` 一样稳。
使用指针记录堆内节点，每次更新不入堆而是修改堆内节点就可以实现 $\mathcal{O}(n\log n+m)$ 的 dijkstra，如果使用 stl 复杂度就是 $\mathcal{O}(m\log m)$。（这里认为节点数 n 小于边数 m）

trie

你不会写 trie 吗?

```
struct node{
    node *s[26]={};
    int d=0;
}*rt=new node;
```

这玩意有啥用相信懂得都懂，怎么用我也不多讲了，直接把 `trie<string,null_type,trie_string_access_traits<>,pat_trie_tag,trie_prefix_search_node_update>` 当成一个类型。成员函数 `insert` 插入字符串，`erase` 删除字符串，`join` 合并 `trie`，`prefix_range` 返回一个 `pair`，`pair` 里面是两个迭代器，两个迭代器直接的每个元素就是以查询字符串作为前缀的所有字符串。
评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是

useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。评价是 useless。

cxx

类似 pbds，里面的东西都在 __gnu_cxx 里面，所以我直接 `namespace cxx=__gnu_cxx`。不过我也只知道里面有个 `rope`。

rope

需要 `#include<ext/rope>`。
有成员函数 `substr` `erase` `operator[]` `replace`（推平）这些操作，复杂度都可以当成 $\mathcal{O}(\sqrt{n})$。不过不支持内部翻转，可以同时维护 2 个 `rope` 一起操作。
`rope` 也是支持 $\mathcal{O}(1)$ 的可持久化的：

```
rope<char>*a,*b;
a=new rope<char>;
b=new rope<char>(*a);
```

`crope` 完全等价于 `rope<char>`。

_builtin

为啥会有这东西啊，这也不是 stl 啊，但是为了多凑点东西我就要讲！这些都是正常的函数，复杂度是比你手写优秀的。我只列出有用的，其他没用的我就不列举了。

位系列

c++20 的位操作使用 bit 更好。

- `__builtin_ffs(x)`：返回 x 最后一个 1 的位置。
- `__builtin_popcount(x)`：返回 x 中 1 的个数。
- `__builtin_ctz(x)`：返回 x 中末尾 0 的个数。
- `__builtin_clz(x)`：返回 x 中前导 0 的个数。
- `__builtin_parity(x)`：返回 x 中 1 个数的奇偶性。
- `__builtin_bswap16/32/64(x)`：返回 x 二进制下翻转的值。
- `__builtin_nearbyint(x)`：返回 x 四舍五入。

浮点系列

可以避免许多 `#include<cmath>`。

- `__builtin_floor(x)`：返回 x 向下取整。
- `__builtin_ceil(x)`：返回 x 向上取整。
- `__builtin_trunc(x)`：返回 x 去除小数位。
- `__builtin_sqrt(x)`：返回 x 的平方根。

作业

必做

CF1620E：尝试使用 list 或者 basic_string 启发式合并。