STL 应用实例

模板

STL=Standard Template Library，标准模板库.

什么是模板（template）？

当函数参数的类型不确定时，可以声明 template<class/typename T>，用 T 指代这个不确定的类型。

调用函数时，编译器会自动将 T 替换为参数的类型。

```
template<typename T> //T即不确定的类型
T min(T x,T y){
    return (x<y)?x:y;
}
//这个 min 函数就可以同时用于 float/double/int/long long
```

同样地，模板也可以用于定义结构体：

```
template<class T1,class T2> //T即不确定的类型
struct pair{
    T1 first;
    T2 second;
};
pair<int,string> pis;
//pis.first 是 int 型的，而pis.second 是 string 型的。
```

迭代器

传统数组使用指针访问地址，而 STL 容器使用迭代器，功能是一样的。

使用`container<Type>::iterator`声明一个迭代器。

C++11 中也可以使用`auto`自动判别类型。

迭代器分为单向移动迭代器、双向移动迭代器、随机访问迭代器。

vector 的迭代器是第三种，map/list/set 的为第二种。

随机迭代器可以加减常数进行移动，也可以比较大小、做差。

双向移动迭代器只能加减 1，判断相等。

单向移动迭代器只能加或减 1（只包括一种），判断相等。

常见应用：

- 遍历容器：`for(map<int>::iterator it=mp.begin();it!=mp.end();++it)`

  使用迭代器访问 map/set 即可从小到大遍历。
- 减小常数：`auto it=mp[key];`

若连续多次使用 `mp[key]`，访问迭代器每次是 $O(1)$ 的，而直接访问 `mp[key]` 是 $O(\log n)$ 的。

注意：所有容器的 end() 函数返回值均为其最后一个元素 **之后** 的地址。

STL 函数

STL 中的函数需要的区间基本都是左闭右开的。

原因：迭代器并不一定能比较大小，左闭右开可以写出循环：`for(it=begin;it!=end;++it)`

sort

应该都会用，使用 快排+插入排序+堆排序，保证最坏复杂度是 $O(n\log n)$ 的。

要保证相等的元素按原来顺序放置，应使用 `stable_sort`，用法与复杂度均相同。

lower_bound/upper_bound

在一个递增数组中二分查找不小于某数的第一个/最后一个数。复杂度 $O(\log n)$

unique

删除有序数组中重复的部分，并返回新的 end。

示例：基于 STL 的离散化。

```
int v[N],id[N];
void Discretization(){
    for(int i=1;i<=n;++i)
        id[i]=v[i];
    sort(id+1,id+n+1);
    int *tail=unique(id+1,id+n+1);
    for(int i=1;i<=n;++i)
        v[i]=lower_bound(id+1,tail,v[i])-id;
}
```

nth_element

在乱序数组中查找第 k 大的数 x，并将其置于第 k 位，并保证其前面的数小于 x，后面的数大于 x。

next_permutation

生成按字典序的下一个全排列。

random_shuffle

随机打乱数组。

STL 容器

不推荐使用：deque（占空间奇大还不释放，经常MLE），stack/list（直接数组写就很简单）

参数传输：容器作为函数参数时，应当使用引用，即 `container<type>& name`，否则传参会产生 $O(n)$ 的时间开销。

string

理论上也是 STL 容器，其实内置了很多函数。

可以直接比较（字典序）、拼接（`+/+=`）、输入、输出，不多赘述。

注意比较是 $O(m)$ 的，也就是说 `set<string>` 复杂度可能变成 $O(m \log n)$。

有 string 作参数的函数通常也可以用 char* 作参数。

下面提到的 “第 n 个字符” 均为从 0 编号。

- `bool empty(void)`：返回字符串是否非空。
- `unsigned int size(void)`：返回字符串大小。
- `void clear(void)`：清空。
- `iterator begin(void)`：返回指向第一个字符的迭代器。
- `iterator end(void)`：返回指向最后一个字符 **之后** 的迭代器（即左闭右开）。
- `char front/back(void)`：返回第一个/最后一个字符。

  易混：begin(),end() 是首尾迭代器，front(),back() 是首尾元素值。

  其实还有 rbegin()，rend()，容易 CE，也并不必需。

  以上几个函数大部分 STL 容器均具有。
- `Type at(int)`：相当于 a[int]，但会在越界时自动 RE，可用于调代码。
- `unsigned int length(void)`：返回字符串长度，与 size() 的值精确一致。
- `int find(char/string,int n)`：从第 n 位开始向后查找一个字符或字符串（默认从头），rfind 为向前查找。
- `string& replace(int n,int len,string str)`：从第 n 位开始的 len 个字符替换为 str。
- `string substr(int n,int len)`：返回从第 n 位开始截取 len 位的子串。
- `void push_back(char)`：向串后连接一个字符 c，相当于 `s+=c`。

下面的不常用，只做了解。

- `int compare(string str)`：将字符串与 str 比较，大于返回 1，小于返回 -1，等于返回 0 .
- `string& insert(int n,string/char)`：向第 n 个字符后插入字符串或单个字符。
- `string& erase(int n,int num)`：删除第 n 位开始的 num 个字符（默认全部删除）。

  注意上面两个函数有返回值。
- `char* c_str(void)`：返回指向字符串开头的字符指针（即传统字符数组）。**此函数一般不应使用**。

vector

支持操作：at,empty,clear,size,push_back,front,back,begin,end，同 string .

- `iterator insert(iterator,Type)`：向指定位置 **前** 插入元素。$O(n)$（与插入位置有关）,返回一个指向该元素的迭代器。
- `iterator erase(iterator)`：删除指定位置的元素。$O(n)$（与删除位置有关），返回一个指向被删除元素后一个元素的迭代器。

  虽然这两个是 $O(n)$，但是常数极小（尤其是 O2 以后），于是诞生了 vector 假平衡树

所以遇到某些题拿 vector 硬做说不定有不错的分。

注意 erase(it) 后 it 会失效，不能再使用。

- `void resize(int n,Type ele)`：大小改变为 n，缩小则删除多余部分，扩大则以 ele（不填则为默认值）填充新增部分。
- `void assign(int n,Type ele)`：将容器内容改为 n 个 ele 。

无 O2 下 vector 效率为普通数组的 0.7 倍，应根据实际需要使用，避免滥用。

有 O2 时 vector 效率玄学，吊打数组可能性微存，但不建议赌。

pair

严格来说，这个不是容器。

就是一个简简单单的二元组，pair<type1,type2> 即可定义，内含 first , second 两个成员。（参考第一部分的例子）

自带一个比较函数，先比较 first ，再比较 second .

可以使用 make_pair(x,y) 构造一个 pair。

建议使用时定义一些宏以方便使用（取决于实际使用次数）。

eg. `typedef pii pair<int,int>` 、`#define make_pair mp`

和自己定义结构体没啥区别，因为自带比较函数有时候还算方便。

queue/priority_queue/stack

这三个容器比较特殊，其本身并不存放元素。元素实际存放在一个底层容器中（默认为 vector），它们只是提供一些操作。

它们并没有迭代器，不能直接访问/遍历（因为要保证内部元素的有序性）。

它们没有什么操作，大伙用的几乎就是全部了。

应用：基于 STL 的dijkstra：（其实就是比较常规的写法）

```
typedef pair<int,int> pii;
vector<pii> edge[N];
priority_queue<pii,vector<pii>,greater<pii> > q;
/*
*某些版本的编译器会将vector<vector<int>>之类的模板的最后两个尖括号
*识别为 >>（右移运算），现在基本没有这个问题了,但仍建议加空格分开。
*/
void dijkstra(int s){
    vector<int> dis(N,0x3f3f3f3f);
    vector<bool> vis(N,0);
    dis[s]=0;
    q.push(make_pair(0,s));
    int h=0;
    while(!q.empty()){
        h=q.top().second;
        q.pop();
        if(vis[h])continue;
        vis[h]=1;
        //auto 遍历语法需要 C++11
```

```
        for(auto e:edge[h])
          if(dis[h]+e.second<dis[e.first]){
            dis[e.first]=dis[h]+e.second;
            q.push(make_pair(dis[e.first],e.first));
        }
    }
}
```

map/multimap

内置了一颗红黑树，因为过度封装，仅支持一部分平衡树操作（查找，插入，删除），复杂度与红黑树相同。

使用 map<type1,type2> mp定义，相当于离散数组，下标可以是任何玩意，直接使用 mp[key] 修改/查询 key 对应的值。

mp[key]=value 中 key 称为键值。

- 访问 mp[key]：$O(\log n)$

  注意，若 mp 如果没有 key 键值，它会自动插入一个。如果你询问的键值不一定存在，查询后你的 map 可能会莫名其妙多一个值。

- `iterator find(type1 key)`：返回一个迭代器，指向 pair<key,value>，不存在则返回 end()。$O(\log n)$
- `pair<iterator,bool> insert(pair<type1,type2>)`：即 `mp[key]=value`。
- `void erase(iterator beg,iterator ed)`：删除 $[beg,ed)$ 间的元素。
- `void erase(iterator it)`：删除 it 指向的元素。
- `iterator lower_bound/upper_bound(type1 key)`：含义同原函数。
- 遍历时键值递增。（二叉平衡搜索树基本特性）

map 效率其实还不错，会额外占用一些空间。

multimap 允许重复值 ~~这对map有任何意义吗~~，map不允许。

应用：计数居多，少数情况作为数组使用。

eg. 给出一长度为 n 的数组 a_n，求满足 $a_i+a_j=K$, $(i\neq j)$ 的无序数对 (i,j) 对数： $O(n\log n)$

```
map<int,int> cnt;
long long match(vector<int>& a,int K){
    long long ans=0;
    for(auto x:a){
        ans+=cnt[K-x];
        cnt[x]++;
    }
    return ans;
}
```

unordered_map

好像是 C++11 的东西，最近 NOIP 都开 C++11 了，应该能用了。

和 map 几乎一模一样，变成基于 hash 实现，复杂度全部变为期望 $O(1)$，和运气有关。

内部元素无序（因此叫 unordered），也就是不能按键值递增遍历了。

自己写一个也不难，但 hash 函数选的不好的话也不快。

因为是hash，所以可以被卡到 $O(n)$ ，请仅在数据规模超过 $O(\log n)$ 承受范围时使用。

set/multiset

功能比 map 更简单一些，只存储值并计数，也是红黑树实现。

定义：`set<type> s;`

同样地，set中元素不重复，multiset元素可重复。

函数和 map 基本一样，但不支持下标访问。

为保证有序，map 和 set 中的元素都不能通过迭代器修改，如果使用结构体，需要某个值可修改，需用 mutable 修饰。

应用：

- eg1. 查找两数列中的重复值：$O(n \log n)$

```
long long match(vector<int>& x,vector<int>& y){
    multiset<int> s;
    long long ans=0;
    for(auto i:x)
        s.insert(i);
    for(auto j:y)
        ans+=s.count(j);
    return ans;
}
```

- 珂朵莉树（ODT）

  堪称是 STL 最综合的应用之一。

  维护区间操作。需数据随机，支持区间赋值（且该操作必须占相当比例）、区间修改、区间查询（几乎支持任何你能想到的查询）。

  本质是用将区间分成相同值的若干段，用 set 存储。

  这个东西用处不大，但能加深你对 set 的理解。

  - 区间的表示：

    ```
    struct node{
        int l,r;//左右端点
        mutable long long v;//区间值，可修改。
        node(int l,int r=-1,ll v=0):l(l),r(r),v(v){}
        bool operator < (const node& p)const{return l<p.l;}
        //按左端点排序。
    }
    set<node> s;
    typedef set<node>::iterator iter;
    ```

  - 区间分裂：将区间 $[l,r]$ 从 pos 处断为 $[l,pos)$ 、$[pos,r]$。

```
iter split(int pos){
    iter mid=t.lower_bound(node(pos));//找到pos所在区间
    if(mid!=t.end()&&mid->l==pos)return mid;//pos不存在或在区间左端点不需分裂
    --mid;
    int l=mid->l,r=mid->r;
    ll v=mid->v;
    t.erase(mid);//删除原区间[l,r]
    t.insert(node(l,pos-1,v));//插入[l,pos)
    return t.insert(node(pos,r,v)).first;//插入[pos,r]，返回pos所在的新区间。
}
```

- 区间赋值：先将 l 和 r+1 处断开，然后删除 $[l,r]$ 中的小区间，插入一个值为 v 的新区间。

```
void assign(int l,int r,ll x){
    iter end=split(r+1),begin=split(l);//先断r+1,否则求end时begin会失效。
    t.erase(begin,end);//erase可以删除begin到end的所有区间
    t.insert(node(l,r,x));
}
```

- 区间查询：先将 l 和 r+1 处断开，然后迭代器遍历 $[l,r]$ 中的小区间，依次查询。

```
//查询[l,r]内每个数的k次方求和。
ll sum(int l,int r,int x,int p){
    ll ans=0;
    iter end=split(r+1),begin=split(l);
    for(iter it=begin;it!=end;++it)//迭代器遍历
        ans=(ans+1ll*(it->r-it->l+1)*fpow(it->v,x,p))%p;
    return ans;
}
```

- 区间修改：同理

```
//区间加常数。
void add(int l,int r,int x){
    iter end=split(r+1),begin=split(l);
    for(iter it=begin;it!=end;++it)
        it->v+=x;
}
```

- 区间第 k 大：将区间内的数放到vector内，排序，查找第 k 大。

```
ll kth(int l,int r,int x){
    vector<pair<ll,int> > v;
    iter end=split(r+1),begin=split(l);
    v.clear();
    for(iter it=begin;it!=end;++it)
        v.push_back(pair<ll,int>(it->v,it->r-it->l+1));
    sort(v.begin(),v.end());
    for(vector<pair<ll,int> >::iterator it=v.begin();it!=v.end();++it){
        x-=it->second;
        if(x<=0)return it->first;
    }
    return -1;
}
```

bitset

相当于一个长 01 串，支持所有位运算，复杂度 $O(\frac{n}{\omega})$ ，根据系统，$\omega = 32$ 或 64 。

多用于位运算/状压相关问题的优化，应用也简单，把 bool 数组换成 bitset 即可。

之后用到再说吧。

其实 C++ 还有更强大的 pb_ds 库，提供可并堆，多种高度扩展的平衡树，可持久化数组等高级数据结构，但使用相当复杂，竞赛也不允许使用，就不讲了。