0x00 STL函数

0x01 vector

> 可理解为变长数组，它的内部实现基于倍增思想。

- **两个数组进行比较**
  - 和 `string` 的比较规则相同，在数组长度相同的情况下比较。
- **定义和初始化**
  - 一维：`vector<int> vec(n)`
  - 二维：`vector g(n, vector<int> (m))`
  - 三维：`vector<vector<vector<int>>> f(n, vector<vector<int>>(m, vector<int>(k)))`
  - 赋值：`vector<int> vec(n, -1)`，另同。
- **`vec.push_back(x)/pop_back()`**
  - `vec.push_back(x)` 把元素 `x` 插入到 `vec` 的尾部。
  - `vec.pop_back()` 删除 `vec` 的最后一个元素。
- **`vec.size()`**
  - 统计元素个数，返回 `int` 型变量，时间复杂度 $O(1)$。
- **`vec.empty()`**
  - 检查 `vec` 是否为空，返回 `bool` 型变量，时间复杂度 $O(1)$。
- **`vec.clear()`**
  - 清空 `vec`，无返回值。
- **`vec.erase()`**
  - `vec.erase(vec.begin(), vec.begin() + 1)` 删除 $[vec.begin(), vec.begin()+1)$ 中的元素。
  - `vec.erase(x)` 删除迭代器 `x` 指向的元素。
- **`vec.begin()/end()`**
  - `vec.begin()` 返回第一个元素的迭代器。
  - `vec.end()` 返回最后一个元素的下一个位置的迭代器。
- **`vec.front()/back()`**
  - `vec.front()` 返回第一个元素。
  - `vec.back()` 返回最后一个元素。
- **`vec.assign(a.begin(), a.end())`**

    - 将数组 `a` 中的每个元素按顺序拷贝到 `vec` 中。
- `vec.swap(a)`
    - 将两数组中的元素进行整体性调换。

0x02 queue

> 队列，先进先出

- `q.push(x)/pop()`
    - 入队尾/出队头，时间复杂度 $O(1)$。
- `q.front()/back()`
    - 返回队首元素/队尾元素，时间复杂度 $O(1)$。
- `q.empty()`
    - 检查队列是否为空，时间复杂度 $O(1)$。
- `q.size()`
    - 返回队列中元素个数，时间复杂度 $O(1)$。
- `q.clear()`
    - 清空队列，时间复杂度 $O(1)$。

0x03 priority_queue

> 优先队列，可看作堆

- **声明**
    - `priority_queue<int> heap`
        - 默认为大根堆，元素越大，优先级越大。
        - 也可称写成 `queue<int, vector<int>, less<int>> heap`
    - `priority_queue<int, vector<int>, greater<int>> heap`
        - 小根堆，元素越小，优先级越大。
        - 其中第二个参数填写的是来承载底层数据结构堆（heap）的容器，如果第一个参数是 `PII` 型，则此处只需要填写 `vector<PII>`。
- `heap.top()`
    - 返回队首（堆顶/优先级最高的）元素，时间复杂度 $O(1)$。
- `heap.push(x)/pop()`
    - 令 `x` 入队/队首（堆顶/优先级最高的）元素出队，时间复杂度 $O(logn)$。
- `heap.empty()`

    - 检查队列是否为空，时间复杂度 $O(1)$。

- `heap.size()`
    - 返回队列中元素个数，时间复杂度 $O(1)$。

0x04 deque

> 双端队列，可在队头或队尾插入和弹出元素

- `q.push_front(x)/pop_front()`
    - 在队头插入/弹出元素，时间复杂度 $O(1)$。
- `q.push_back(x)/pop_back()`
    - 在队尾插入/弹出元素，时间复杂度 $O(1)$。
- `q.front()/back()`
    - 返回队首元素/队尾元素，时间复杂度 $O(1)$。
- `q.empty()`
    - 检查队列是否为空，时间复杂度 $O(1)$。
- `q.size()`
    - 返回队列中元素个数，时间复杂度 $O(1)$。
- `q.clear()`
    - 清空队列，时间复杂度 $O(1)$。

0x05 set 与 multiset

> 两者的内部实现是一颗红黑树（平衡树的一种），会自动进行排序（从小到大），支持的函数基本相同。
>
> - `set`是数学上的有序集合——具有唯一性，即每个元素只出现一次。
> - `multiset`是有序多重集合，每个元素可以出现若干次。

- `se.insert(x)`
    - 插入元素 `x`，返回插入地址的迭代器和是否插入成功的 `bool` 并成的 `pair` ，时间复杂度为 $O(logn)$。
    - `set` 在进行插入的时候是不允许有重复的键值的，如果新插入的键值与原有的键值重复则插入无效（`multiset` 可以重复）。
- `se.erase(x)`
    - 删除元素 `x`，参数可以是**元素**或者**迭代器**，返回下一个元素的迭代器，时间复杂度为 $O(logn)$ 。
    - **注意在 `multiset` 中 `se.erase(x)` 会删除所有值为 `x` 的元素。**若只想删除 `multiset` 中的一个 `x` 元素，删除其中一个等于 `x` 的迭代器即可。

- `se.size()`
  - 统计元素个数，返回 `int` 型变量，时间复杂度 $O(1)$。
- `se.empty()`
  - 检查 `set` 是否为空，返回 `bool` 型变量，时间复杂度 $O(1)$。
- `se.clear()`
  - 清空 `se`，无返回值。
- `se.count(x)`
  - 查找元素 `x` 的个数，返回 `x` 的个数，时间复杂度 $O(logn)$。
- `se.find(x)`
  - 查找元素 `x` ，并返回指向第一个等于 `x` 的迭代器，若不存在，则返回 `se.end()`，时间复杂度为 $O(logn)$。
- `se.begin()/end()`
  - 返回 `set` 的第一个元素/最后一个元素的下一个位置的迭代器。
- `se.rbegin()/rend()`
  - 返回 `set` 的最后一个元素/第一个元素的迭代器。
- `prev(it)/next(it)`
  - 获取某个迭代器的上一个/下一个迭代器，返回迭代器。
- `se.lower_bound(x)/upper_bound(x)`
  - 用法与 `find` 类似，但查找的条件略有不同，时间复杂度 $O(logn)$。
  - `se.lower_bound(x)` 表示查找第一个 $\geq x$ 的元素，并返回指向该元素的迭代器。
  - `se.upper_bound(x)` 表示查找第一个 $> x$ 的元素，并返回指向该元素的迭代器。
  - 对于 $>$ `set` 中最大的元素的元素，返回 `se.end()`。

0x06 map 与 unordered_map

> 映射，内部元素唯一。
>
> - `map` 可看作Hash表使用，建立从复杂信息key到简单信息value的映射。由于是基于红黑树（平衡树）实现，所以大部分操作时间复杂度在 $O(logn)$ 级别，略慢于使用哈希函数实现的传统哈希表。 `unoredered_map` 基于Hash函数的容器，内部元素是无序的。

- **赋值**
  - `mp[key] = value`, `mp[key] ++` 等。
  - `mp.insert({key, value})`，插入元素，若容器中已经存在键值相同的元素，当前插入作废。
- `mp.find(key)`

    - 返回键为 `key` 的映射的迭代器，若不存在，返回 `mp.end()`，时间复杂度 $O(logn)$。

- `mp.erase(it/{key, value})`
    - 参数可以是 `pair` 或迭代器。
- `mp.size()`
    - 获得 `mp` 中映射的对数，时间复杂度 $O(1)$。
- `mp.clear()`
    - 清空所有元素，复杂度为 $O(N)$，其中 `N` 为 `mp` 中元素的个数。
- `mp.empty()`
    - 判断容器是否为空。
- `mp.count(key)`
    - 如果该键存在于容器中，则此函数返回布尔数1；如果该键不存在于容器中，则返回0。

0x07 bitset

> bitset 可看作一个多为二进制数，每8位占用1个字节，相当于采用了状态压缩的二进制数组，并支持简单的位运算。在估算程序运行时间是，我们一般采用以32位整数的运算次数为基准。

- **声明**
    - `bitset<length> bs`
        - 表示一个 `length` 位的二进制数，`<>` 中写位数。
    - `bitset<length> bs(x)`
        - 将一个十进制数 `x` 表示成一个 `length` 位的二进制数。
- **位运算操作符**
    - `~bs` 返回对 `bs` 按位取反的结果。
    - `&, |, ^` 返回对两个位数相同的 `bitset` 执行按位与、或、异或运算的结果。
    - `>>, <<` 返回把一个 `bitset` 右移、左移若干位的结果。
    - `==, !=` 比较两个 `bitset` 代表的二进制数是否相等。
- **[]操作符**
    - `bs[k]` 表示 `bs` 的第 `k` 位，既可以取值，也可以赋值。
    - 在 10000 位的二进制表示中，最低位为 `bs[0]`，最高位为 `bs[9999]`。
- `bs.count()`
    - 返回值有多少位为1。
- `bs.any()/none()`
    - 若 `bs` 所有位都为0，则 `bs.any()` 返回 `false`，`bs.none()` 返回 `true`。
    - 若 `bs` 至少一位为1，则 `bs.any()` 返回 `true`，`bs.none()` 返回 `false`。

- `set`
  - `bs.set()` 把 `bs` 的所有位变为1。
  - `bs.set(k, v)` 把 `bs` 的第 `k` 位改为 `v`，即`bs[k] = v`。
- `reset`
  - `bs.reset()` 把 `bs` 的所有位变为0。
  - `bs.reset(k)` 把 `bs` 的第 `k` 位改为0，即 `bs[k] = 0`。
- `flip`
  - `bs.flip()` 把 `bs` 的所有位取反，即 `s = ~s`。
  - `bs.flip(k)` 把 `bs` 的第 `k` 位取反，即 `bs[k] ^= 1`。
- **结论**
  - 连续的与运算的结果一定不大于参与运算的所有数中的最小值。
  - 一个数异或上一个奇数，其奇偶性改变；一个数加上一个奇数，其奇偶性改变。
  - $48 \oplus 1 = 49$，$48 \oplus 1 = 48$。

0x08 #include<algorithm>

- `accumulate` **求和**
  - `int sum = accumulate(vec.begin(), vec.end(), 0)` 求和。
  - 返回和。
- `ceil/floor` **向上/向下取整**
  - **参数是** `double` **类型**，输出整数。
- `count` **计数**
  - `count(vec.begin(), vec.end(), x)`，计算 `x` 在 `vec` 中出现的次数，**`x` 可以是数字，也可以是字符。**
  - 返回 `x` 出现的次数。
- `is_sorted` **是否有序**
  - 检测序列是否有有序，返回值为`bool`型。
  - `is_sorted(iterator start, iterator end)` 默认非降序。
  - `is_sorted(iterator start, iterator end, greater())` 非升序。
- `is_sorted_until` **是否有序 + 找到第一个破坏有序状态的元素**
  - 检测序列是否有序，并且返回一个正向迭代器，该迭代器指向的是当前序列中第一个破坏有序状态的元素，若没有返回`vec.end()`。
  - 语法与 `is_sorted()` 相同。
- `iota` **连续填充序列**
  - `iota(vec.begin(), vec.end(), x)`，数组 `vec` 从 `x` 开始填充，填充一次后 `x ++` 。

- `lower/upper_bound` **二分查找**
  - 返回迭代器，时间复杂度 $O(logn)$。
  - `*lower_bound(a.begin(), a.end(), x)`，找到第一个 $\geq x$ 的元素。
  - `*lower_bound(a.begin(), a.end(), x) - 1`，找到最后一个 $< x$ 的元素。
  - `*upper_bound(a.begin(), a.end(), x)`，找到第一个 $> x$ 的元素。
  - `*upper_bound(a.begin(), a.end(), x) - 1`，找到最后一个 $\leq x$ 的元素。
- `max/min_element` **查找最大/最小值**
  - 返回**迭代器**，时间复杂度 $O(n)$。
  - `*max_element(vec.begin(), vec.end())`，在数组中查找最大值。
  - `*min_element(vec.begin(), vec.end())`，在数组中查找最小值。
- `next/pre_permutation` **下一个/上一个排列**
  - 把两个迭代器指定的部分看作一个排列，求出这些元素构成的全排列中，字典序排在下一个/上一个的排列，并直接在序列上更新。
  - 若不存在排名更靠后/靠前的的排列，则返回 `false`，负责返回 `true`。
  - `do{...} while(next_permutation(vec.begin(), vec.end()))`
- `reverse` **翻转**
  - 翻转数组，时间复杂度 $O(n)$。
  - `reverse(vec.begin(), vec.end())`
- `random_shuffle` **随机打乱**
  - `random_shuffle(vec.begin(), vec.end())`
- `rotate` **旋转序列**
  - 想象数组是一个闭合的圆形回路，将数组进行旋转，`vec.begin() + k` 变成数组首，`vec.begin() + k - 1` 变成数组尾。时间复杂度 $O(n)$。
  - `rotate(vec.begin(), vec.begin() + k, vec.end())`
- `sort` **快排**
  - 时间复杂度 $O(nlogn)$。
  - `sort(vec.begin(), vec.end())` 默认从小到大排序。
  - `sort(vec.begin(), vec.end(), greater()/cmp)` 从大到小排序。

```
auto cmp = [&] (int x, int y)
{
    return x > y;
};
```

- `unique` **去重**
  - 去掉相邻的重复元素，一般搭配 `sort` 一起使用，返回最后一个元素的下一个位置的迭代器。

    - `int n = unique(vec.begin(), vec.end()) - a.begin()`，计算去重后剩余元素个数。

0x09 #include<string.h>

- `substr` **截取字符串**
    - `substr(str, pos, len)`，从 `pos` 开始，截取长度为 `len` 的字符串，返回这个截取的字符串。

0x0A #include<math.h>

- `abs/fabs` **取绝对值**
    - `abs(x)` 返回**整数** x 的绝对值。
    - `fabs(x)` 返回**浮点数** x 的绝对值。
- `sqrt` **平方根**
    - `sqrt(x)` 参数 x 为非负数，返回浮点数。
- `pow` **次方**
    - `pow(x, y)` x、 y 和返回值是同类型的浮点数。
- `sin/cos/tan` **三角函数**
    - `sin(x) / cos(x) / tan(x)`
    - 参数是由弧度表示的浮点数，返回浮点数。
- `asin/acos/atan/atan2` **反三角函数**
    - `asin(y / 斜边) / acos(x / 斜边) / atan(y / x)` 参数为浮点数，返回浮点数弧度数。
    - `atan2(y, x)` 返回返回浮点数弧度数。
- `log` **对数函数**
    - `log(x)` 返回以 e 为底的对数。
    - `log2(x)` 返回以 2 为底的对数。
    - `log10(x)` 返回以 10 为底的对数。

0x0B __builtin 位运算函数

> **所有带 `ll` 的名字，均为 `long long` 类型下运算，否则将当作 `int` 来算。**

- `__builtin_clz()/__builtin_clzll()` **前缀零个数**
    - 返回括号内数的二进制表示形式中前导 0 的个数。
    - `long long` 转换成二进制有 64 位，`int` 转换成二进制有 32 位。
- `__builtin_ctz()/__builtin_ctzll()` **后缀零个数**
    - 返回括号内数的二进制表示形式中末尾 0 的个数。
- `__builtin_popcount()` **二进制表示中** 1 **的个数**
    - 返回括号内数的二进制表示形式中 1 的个数。

- `__builtin_parity()` **二进制表示中 1 的个数的奇偶性**
  - 返回括号内数的二进制表示形式中 1 的个数的奇偶性（偶：0，奇：1）。
- `__builtin_ffs()` **二进制表示中最后一位 1 的位置**
  - 返回括号内数的二进制表示形式中最低位的 1 在第几位（从后往前），下标从 1 开始。