



Leonardo 的 ICPC 模板

模板用的好，金牌少不了

作者：列奥那多是勇者

组织：狗头吧

时间：August 24, 2022

版本：0.1.0

邮箱：azraelzarkli@gmail.com



狗头吧在任何时候都是 acmer 最好的救济

写在前面

模板使用须知

本模板使用了 elegantbook-magic-revision latex 模板 (<https://github.com/Azure1210/elegantbook-magic-revision>)，在此对其表达感谢。

本文档的题显版本相关题目中包含 AcWing 题目，部分是需要买课才能查看的，这类题目可以到其它算法网站(如洛谷)上查找。



目录

1

第 1 部分 * STL

第 1 章	STL	2
1.1	pair	2
1.2	vector	2
1.3	array	3
1.4	deque	3
1.5	list	3
1.6	set / unordered_set	4
1.7	map / unordered_map	4
1.8	string	5
1.9	stack	5
1.10	queue	5
1.11	priority_queue	6
1.12	bitset	6
1.13	iterator	7
1.14	algorithm	7
1.14.1	Non-modifying sequence operations	7
1.14.2	Modifying sequence operations	8

2

第 2 部分 * 数据结构

第 2 章	数据结构	11
2.1	并查集 DSU/UF	11

3

第 3 部分 * 基础算法

第 3 章	基础算法	14
3.1	排序算法	14
3.1.1	快速排序	14



3.1.2 归并排序	14
3.2 二分	15
3.2.1 整数二分	15
3.2.2 浮点数二分	15
3.3 高精度	16
3.3.1 加法	16
3.3.2 减法	16
3.3.3 乘法	17
3.3.4 除法	17
3.4 前缀和	18
3.5 差分	18

4

第 4 部分 * 进阶算法

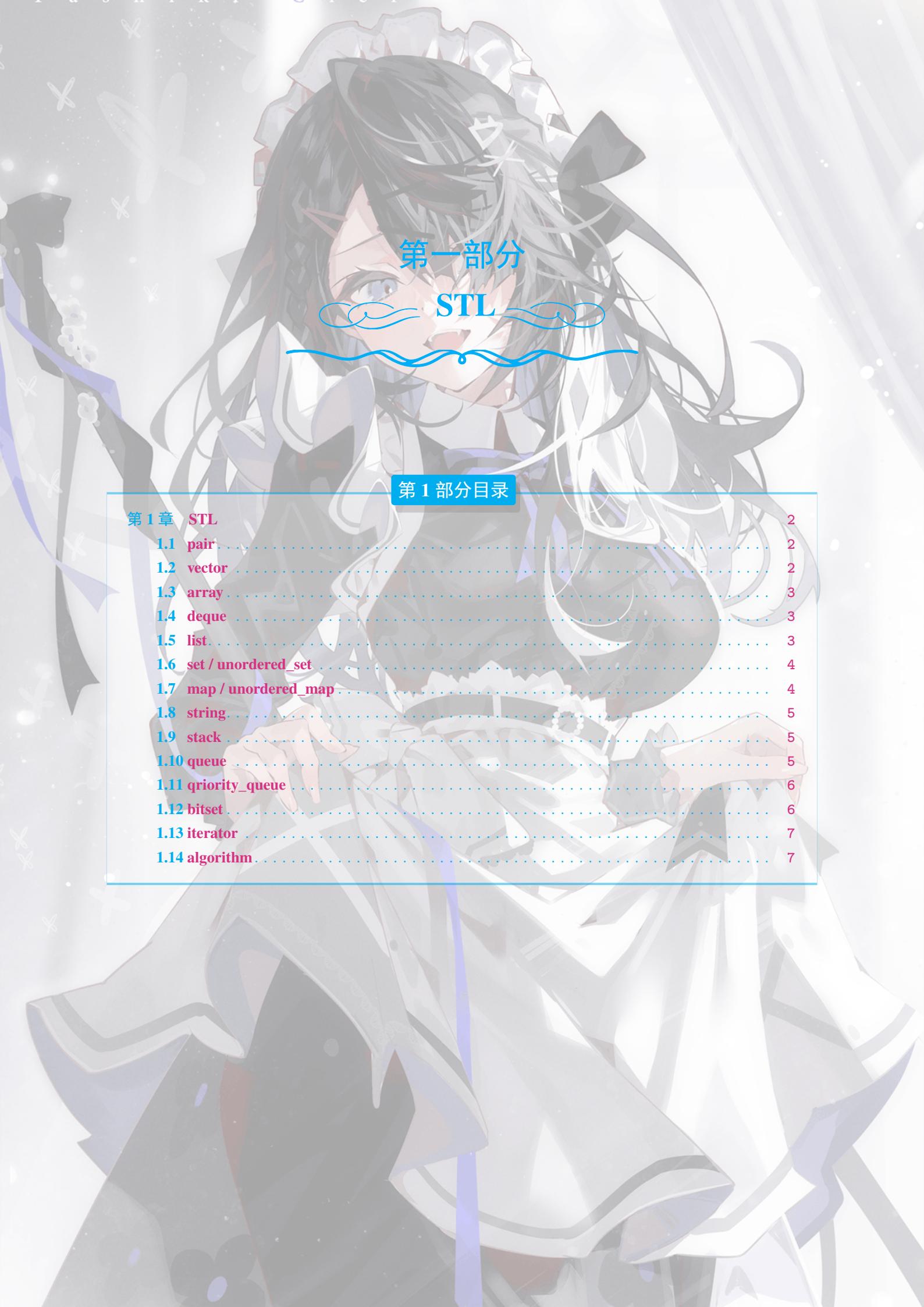
第 4 章 图论	20
4.1 二分图	20
4.1.1 检测二分图	20
4.2 拓扑排序	21

5

第 5 部分 * 附录

第 5 章 附录	24
5.1 比赛初始代码模板	24
5.2 unordered_set/map 哈希函数	25
5.3 Big-O Cheat Sheet	26
5.3.1 Common Data Structure Operations	26
5.3.2 Array Sorting Algorithms	26
5.3.3 Growth Rates	27
5.4 数据范围 => 算法复杂度及算法内容	27





第一部分

STL

第1部分目录

第1章 STL	2
1.1 pair	2
1.2 vector	2
1.3 array	3
1.4 deque	3
1.5 list	3
1.6 set / unordered_set	4
1.7 map / unordered_map	4
1.8 string	5
1.9 stack	5
1.10 queue	5
1.11 priority_queue	6
1.12 bitset	6
1.13 iterator	7
1.14 algorithm	7



第 1 章 STL

备注：以 c++14 为标准

◆ 1.1 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。

```
1 pair<int, int> a{1,2};  
2 cout << a.first << " " << a.second;  
3 pair<int, int> b{3,1};  
4 cout << (a < b) << endl;  
5 // vector<pair<int, int>> c{a, b};  
6 vector<pair<int, int>> c{{1,2}, {1,1}, {0,3}};  
7 // sort c according to the first param ascendingly (升序);  
8 sort(c, c + 3);  
9 // sort c according to the second param ascendingly (升序); 返回值为false交换  
10 sort(c.begin(), c.end(), [](const pair<int,int> &a, const pair<int, int> &b){return a.second < b.second;});
```

◆ 1.2 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。

```
1 #include <vector>  
2 vector<pair<int,int>> a;  
3 vector<int> b(3, 2); // 默认0, 手动2  
4 vector<int> d(b); // 用b创建d, 线性复杂度  
5 vector<int> e(move(a)); // e = move(a), 常数复杂度  
6 d.resize(20); // 重新分配空间, 若空间小于原本, 则删除
```

```

7 vector<vector<int>> c{vector<int>{1,2}};
8 vector<vector<int>> e(10,vector<int>(10));
9 c.resize(2, vector<int>(2)); // 分配二维空间
10 c.empty(); c.size(); c.clear();
11 c.erase(c.begin() + 1); c.erase(c.begin(), c.end());
12 a.emplace_back(1, 2);
13 a.swap(d);
14 a.pop_back();
15 a.insert(a.begin(), {{1,2}, {3,4}}); // can insert multiple elements
16 a.begin(); a.end(); a.rbegin(); a.rend(); // iterator
17 a.front(); a.back(); // reference
18 a.data(); // pointer to the first element, error when empty

```

1.3 array

`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。

```

1 #include <array>
2 array<int, 3> a {0};
3 a.fill(1);
4 a.size(); a.empty();
5 a.front(); a.back();
6 a.swap(c); // same type and length

```

1.4 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。

```

1 #include <deque>
2 deque<int> a {1,2,3};
3 a.emplace_back(4);
4 a.pop_back();
5 a.emplace_front(0);
6 a.pop_front();
7 a.begin(); a.end(); a.rbegin(); a.rend();
8 a.back(); a.front();
9 a.clear(); // delete all element
10 a.insert(a.begin(), {1,2}); // can insert multiple elements

```

1.5 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` 的使用方法与 `deque` 基本相同，但是增删操作和访问的复杂度不同。此处列下特殊的（下列函数 `list` 提供了特别的实现以供使用）：

```

1 #include <list>
2 list<int> a{1,2,3};
3 list<int> b{4,6,4};
4 a.splice(a.begin(), b); // 4 6 5 1 2 3
5 a.merge(b); // append b to a
6 a.remove(4); // remove all elements equals to 4
7 a.remove_if([](int x){return x < 3;});
8 a.sort();
9 a.unique(); // remove consecutive duplicate (连续重复) elements

```

1.6 set / unordered_set

关于 `unordered_set` 的哈希冲突，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，代码实现见本册附录。

`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。（当然也有 `unordered_multiset`）

```

1 #include <set>
2 // elements in set are sorted accendingly, usually implemented by red-black tree;
3 set<int> a {1, 2, 3, 7}, b{1, 5, 6};
4 a.insert(4); a.count(10); a.empty(); a.size(); a.erase(7);
5 a.merge(b); // c++17 => a: {1,2,3,5,6,7}; b: {1};
6 a.find(7); // iterator
7 a.upper_bound(3); // <= 3, iterator
8 a.lower_bound(2); // >= 2, iterator
9 *a.rbegin(); // value of last element
10
11 set<int> d{0,1,2};
12 set<int> res;
13 // intersection && union
14 // the two functions int <algorithm> do not work nomally with unordered_set
15 set_intersection(a.begin(), a.end(), d.begin(), d.end(), inserter(res, res.end()));
16 set_union(a.begin(), a.end(), d.begin(), d.end(), inserter(res, res.end()));

```

1.7 map / unordered_map

关于 `unordered_map` 的哈希冲突，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，代码实现见本册附录。

`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。`multimap` 则允许有重复的 Key 值。（当然也有 `unordered_multimap`）

```

1 #include <map>
2 map<int, int> a;
3 a[1] = 2; // if not exist, then create

```

```

4 cout << a[0]; // if not exist, then return default value(here is 0);
5 a.count(1); // whether exist
6 a.find(1); // iterator
7 cout << (*a.lower_bound(0)).first; // iterator of type pair
8 cout << (*a.upper_bound(2)).second;
9 a.emplace(1,3);
10 a.empty(); a.size();

```

1.8 string

`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。

```

1 #include <string>
2 string a = "abcdef123456";
3 string b(2, '2'); // 22;
4 printf("%s\n", a.c_str()); // for printf
5 a.length(); // a.size();
6 a.find('c'); // return position number, unfind: -1
7 a.find('c', 4); // start find from position 4
8 a.substr(0, 2); // "ab"
9 a.erase(2); // delete all from pos 2
10 a.erase(0, 2); // delete 2 characters from pos 0
11 a.insert(0, "hello");
12 a.insert(2, 3, 'u'); // insert char 'u' 3 times
13 a.replace(2, 5, ""); // replace whole 5 characters with a string ""
14 string s1 = to_string(12.05); // Converts number to string
15 string s;
16 getline(cin, s); // Read line ending in '\n'(not include)

```

1.9 stack

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。

```

1 #include <stack>
2 stack<int> s;
3 stack<int> s2(s);
4 s.push(1); s.pop();
5 cout << s.top(); // error, can't use top() when s is empty
6 s.size(); s.empty();

```

1.10 queue

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。

```

1 #include <queue>
2 queue<int> q;
3 queue<int> q2(q);
4 q.push(1); q.pop();
5 cout << q.front(); // error, can't use front() when q is empty
6 q.size(); q.empty();

```

1.11 priority_queue

默认容器为 vector, 默认算子为 less, 也就是最大堆

```

1 #include <queue>
2 // big heap & small heap
3 priority_queue<int> big_heap;
4 priority_queue<int,vector<int>,less<int> > big_heap;
5 priority_queue<int,vector<int>,greater<int> > small_heap;
6
7 // operations
8 priority_queue<int> q;
9 q.push(1);
10 cout << q.top(); // not empty
11 q.empty(); q.size();
12 q.pop();
13
14 // custom structure
15 struct Status {
16     int val;
17     ListNode *ptr;
18     bool operator < (const Status &rhs) const { // 必须定义<运算符
19         return val > rhs.val; // rhs 小于 当前 val 就交换, 说明是最小堆
20     }
21 };
22 priority_queue<Status> q;
23 q.push({l->val, l}); // l 是 ListNode* 类型

```

1.12 bitset

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。

```

1 #include <bitset>
2 bitset<3> a; // 000
3 bitset<3> b(14); // 14->1(110) lower 3 bits
4 bitset<4> c("1010"); // 1010
5 string astr = "10101";
6 bitset<4> d(astr, 1, 4); // astr must be a string(not char[] "10101"), from pos 1, length 4
7 d.any(); // exists 1 ?
8 d.none(); // doesn't exists 1 ?

```

```

9 d.count(); // count of 1
10 d.size(); // size
11 d[0];
12 d.test(0); // 1 at pos 0?
13 d.set(); // set all to 1
14 d.set(2); // set 1 to pos 2
15 d.set(2, 0); // set 0 to pos 2
16 d.reset(); // set all to 0;
17 d.reset(2); // set 0 to pos 2
18 d.flip(); // flip all pos
19 d.flip(2);
20 d[0].flip();
21 d.to_string();
22 d.to_ulong();
23 d.to_ullong();
24 cout << d._Find_first(); // the pos of the first true (start from 1); if all false, return the size;
25 cout << d._Find_next(1); // the pos of the next true ...

```

1.13 iterator

```

1 vector<int> a{1,2,3};
2 auto it = a.begin() + 1; // *it = 2;
3 advance(it, 1); // *it = 3;
4 distance(a.begin(), it); // pos it (*it == 3) - a.begin() = 2
5 prev(it); // *x = 2, it no change
6 next(it); // *x = 0; exceed; it no change

```

1.14 algorithm

1.14.1 Non-modifying sequence operations

表 1.1: Non-modifying sequence operations

operations	functions
all_of any_of none_of	checks if a predicate is true for all, any or none of the elements in a range
for_each	applies a function to a range of elements
count count_if	returns the number of elements satisfying specific criteria
mismatch	finds the first position where two ranges differ
find find_if find_if_not	finds the first element satisfying specific criteria
find_end	finds the last sequence of elements in a certain range
find_first_of	searches for any one of a set of elements
adjacent_find	finds the first two adjacent items that are equal (or satisfy a given predicate)
search	searches for a range of elements
search_n	searches a range for a number of consecutive copies of an element

```

1 vector<int> a{1, 2, 3, 1};
2
3 all_of(a.begin(), a.end(), [](int x){return x % 2 == 0;}); // false
4 // any_of, none_of 同理
5
6 for_each(a.begin(), a.end(), [](int x){cout << x << " "});
7
8 count(a.begin(), a.end(), 3); // num of value which is 3
9 count_if(a.begin(), a.end(), [](int x){return x % 2 == 1;});
10
11 find(a.begin(), a.end(), 2); // first pos (iterator)
12 find_if(a.begin(), a.end(), [](int x){return x % 2 == 1;}); // first pos (iterator)
13 // find_if_not 同理
14
15 vector<int> b{2,3};
16 // find begining of the last matched pattern(b) in a
17 find_end(a.begin(), a.end(), b.begin(), b.end()); // in this example: return the pos of 2 in a
18 // find_fist_of 同理
19 // search 作用和用法和 find_first_of 类似
20 search_n(a.begin(), a.end(), 1, 1); // 1,1 : count, value
21
22 adjacent_find(a.begin(), a.end()); // 相邻元素相同 first pos (iterator)
23
24 // need to be added:
25 // mismatch, search

```

1.14.2 Modifying sequence operations

表 1.2: Modifying sequence operations

operations	functions
copy copy_if	copies a range of elements to a new location
copy_n	copies a number of elements to a new location
copy_backward	copies a range of elements in backwards order
move	moves a range of elements to a new location
move_backward	moves a range of elements to a new location in backwards order
fill	copy-assigns the given value to every element in a range
fill_n	copy-assigns the given value to N elements in a range
transform	assigns the results of successive function calls to every element in a range
generate	assigns the results of successive function calls to every element in a range
generate_n	assigns the results of successive function calls to N elements in a range
remove remove_if	removes elements satisfying specific criteria
remove_copy remove_copy_if	copies a range of elements omitting those that satisfy specific criteria
replace replace_if	replaces all values satisfying specific criteria with another value
replace_copy replace_copy_if	copies a range, replacing elements satisfying specific criteria with another value

swap	swaps the values of two objects
swap_ranges	swaps two ranges of elements
iter_swap	swaps the elements pointed to by two iterators
reverse	reverses the order of elements in a range
reverse_copy	creates a copy of a range that is reversed
rotate	rotates the order of elements in a range
rotate_copy	copies and rotate a range of elements
shuffle	randomly re-orders elements in a range
unique	removes consecutive duplicate elements in a range
unique_copy	creates a copy of some range of elements that contains no consecutive duplicates

第二部分 数据结构

第 2 部分目录

第 2 章 数据结构	11
2.1 并查集 DSU/UF	11



第 2 章 数据结构

2.1 并查集 DSU/UF

DSU: Disjoint Set Union; **UF**: Union Find 并查集是一种树形的数据结构，顾名思义，它用于处理一些不交集的合并及查询问题。它支持两种操作：

- 合并 (Union): 将两个子集合并成一个集合。
- 查找 (Find): 确定某个元素处于哪个子集；

```
1 // https://codeforces.com/contest/1691/submission/159022997
2 // 自己改进版本：加入cnt变量表示有几组，查询时间复杂度 O(1)
3 struct DSU {
4     std::vector<int> f, siz;
5     int cnt;
6     DSU(int n) : f(n), siz(n, 1), cnt(n) { std::iota(f.begin(), f.end(), 0); } // f记录parent
7     int leader(int x) { // to implement other funcs
8         while (x != f[x]) x = f[x] = f[f[x]];
9         return x;
10    }
11    bool same(int x, int y) { return leader(x) == leader(y); } // 是否一组
12    bool merge(int x, int y) { // 合并
13        x = leader(x);
14        y = leader(y);
15        if (x == y) return false;
16        siz[x] += siz[y];
17        f[y] = x;
18        cnt--;
19        return true;
20    }
21    int size(int x) { return siz[leader(x)]; } // 一组多少个元素
22    int count() {return cnt;} // 几组
23};
```

讲解: <https://labuladong.github.io/algo/2/22/53/>

相关题目:

1. LeetCode 990. Satisfiability of Equality Equations: <https://leetcode.cn/problems/satisfiability-of-equality-equations/>



第三部分

基础算法

第3部分目录

第3章 基础算法	14
3.1 排序算法	14
3.2 二分	15
3.3 高精度	16
3.4 前缀和	18
3.5 差分	18



第3章 基础算法

◆ 3.1 排序算法

3.1.1 快速排序

```
1 void quick_sort(int q[], int l, int r)
2 {
3     if (l >= r) return;
4
5     int i = l - 1, j = r + 1, x = q[l + r >> 1];
6     // x取l和r的中间值避免了某些情况下x取l或者r导致的无限循环问题
7     while (i < j)
8     {
9         do i ++ ; while (q[i] < x);
10        do j -- ; while (q[j] > x);
11        if (i < j) swap(q[i], q[j]);
12    }
13    quick_sort(q, l, j), quick_sort(q, j + 1, r);
14 }
```

3.1.2 归并排序

```
1 // 需要自己定义tmp数组(n)
2 void merge_sort(int q[], int l, int r)
3 {
4     if (l >= r) return;
5
6     int mid = l + r >> 1;
7     merge_sort(q, l, mid);
```

```

8     merge_sort(q, mid + 1, r);
9
10    int k = 0, i = l, j = mid + 1; // j 只能取 mid+1
11    while (i <= mid && j <= r)
12        if (q[i] <= q[j]) tmp[k ++] = q[i ++];
13        else tmp[k ++] = q[j ++];
14
15    while (i <= mid) tmp[k ++] = q[i ++];
16    while (j <= r) tmp[k ++] = q[j ++];
17
18    for (i = l, j = 0; i <= r; i ++, j ++) q[i] = tmp[j];
19}

```

相关题目：

- AcWing 788. 逆序对的数量: <https://www.acwing.com/problem/content/790/>

3.2 二分

3.2.1 整数二分

```

1 bool check(int x) {/* ... */} // 检查x是否满足某种性质
2 // 下面两者初始值一般情况下 l = 0 和 r = n - 1, 取决于集合的开闭
3 // 区间 [l, r] 被划分成 [l, mid] 和 [mid + 1, r] 时使用: (也就是r=mid)
4 int bsearch_1(int l, int r)
5 {
6     while (l < r)
7     {
8         int mid = l + r >> 1;
9         if (check(mid)) r = mid; // 如 q[mid] >= x 寻找第一个大于等于x的位置
10        else l = mid + 1;
11    }
12    return l;
13}
14 // 区间 [l, r] 被划分成 [l, mid - 1] 和 [mid, r] 时使用: (也就是l=mid)
15 int bsearch_2(int l, int r)
16 {
17     while (l < r)
18     {
19         int mid = l + r + 1 >> 1; // 注意 "+ 1"
20         if (check(mid)) l = mid; // 如 q[mid] <= x 寻找第一个小于等于x的位置
21         else r = mid - 1;
22    }
23    return l;
24}

```

3.2.2 浮点数二分

```

1 bool check(double x) {/* ... */} // 检查x是否满足某种性质
2
3 double bsearch_3(double l, double r)
4 {
5     const double eps = 1e-6;    // eps 表示精度, 取决于题目对精度的要求, 一般比题目要求精度更严格两位
6     while (r - l > eps)
7     {
8         double mid = (l + r) / 2;
9         if (check(mid)) r = mid;
10        else l = mid;
11    }
12    return l;
13}

```

相关题目：

1. AcWing 790. 数的三次方根: <https://www.acwing.com/activity/content/problem/content/824/>

3.3 高精度

也就是数比较大, 不能放在通常的类型中的时候(像 python, java 这类的语言本身就实现了这类性质)

3.3.1 加法

```

1 // C = A + B, A >= 0, B >= 0
2 // 所有都是数的低位放在数组前面的
3 vector<int> add(vector<int> &A, vector<int> &B)
4 {
5     if (A.size() < B.size()) return add(B, A);
6
7     vector<int> C;
8     int t = 0;
9     for (int i = 0; i < A.size(); i++)
10    {
11        t += A[i];
12        if (i < B.size()) t += B[i];
13        C.push_back(t % 10);
14        t /= 10;
15    }
16
17    if (t) C.push_back(t);
18    return C;
19}

```

3.3.2 减法

```

1 // C = A - B, 满足A >= B, A >= 0, B >= 0
2 // 所有都是数的低位放在数组前面的

```

```

3 vector<int> sub(vector<int> &A, vector<int> &B)
4 {
5     vector<int> C;
6     for (int i = 0, t = 0; i < A.size(); i++)
7     {
8         t = A[i] - t;
9         if (i < B.size()) t -= B[i];
10        C.push_back((t + 10) % 10);
11        if (t < 0) t = 1;
12        else t = 0;
13    }
14
15    while (C.size() > 1 && C.back() == 0) C.pop_back();
16    return C;
17 }

```

3.3.3 乘法

高精度乘低精度如 10000 位的乘以一个大小为 100 的

```

1 // C = A * b, A >= 0, b >= 0
2 // 所有都是数的低位放在数组前面的
3 vector<int> mul(vector<int> &A, int b)
4 {
5     vector<int> C;
6
7     int t = 0;
8     for (int i = 0; i < A.size() || t; i++)
9     {
10         if (i < A.size()) t += A[i] * b;
11         C.push_back(t % 10);
12         t /= 10;
13     }
14
15     while (C.size() > 1 && C.back() == 0) C.pop_back();
16
17     return C;
18 }

```

3.3.4 除法

高精度除以低精度如 10000 位的除以一个大小为 100 的

```

1 // A / b = C ... r, A >= 0, b > 0 注意余数r是引用类型，随便传一个变量即可
2 // 所有都是数的低位放在数组前面的
3 vector<int> div(vector<int> &A, int b, int &r)
4 {
5     vector<int> C;
6     r = 0;

```

```

7   for (int i = A.size() - 1; i >= 0; i -- )
8   {
9       r = r * 10 + A[i];
10      C.push_back(r / b);
11      r %= b;
12  }
13  reverse(C.begin(), C.end());
14  while (C.size() > 1 && C.back() == 0) C.pop_back();
15  return C;
16 }
```

3.4 前缀和

```

1 // 一维前缀和
2 // S[i] = a[1] + a[2] + ... a[i]
3 // a[1] + ... + a[r] = S[r] - S[l - 1]
4
5 // 二维前缀和
6 // S[i, j] = 第i行j列格子左上部分所有元素的和
7 // 以(x1, y1)为左上角, (x2, y2)为右下角的子矩阵的和为:
8 // S[x2, y2] - S[x1 - 1, y2] - S[x2, y1 - 1] + S[x1 - 1, y1 - 1]
```

3.5 差分

```

1 // 一维差分
2 // 给区间[l, r]中的每个数加上c: B[l] += c, B[r + 1] -= c
3
4 // 二维差分
5 // 给以(x1, y1)为左上角, (x2, y2)为右下角的子矩阵中的所有元素加上c:
6 // S[x1, y1] += c, S[x2 + 1, y1] -= c, S[x1, y2 + 1] -= c, S[x2 + 1, y2 + 1] += c
```

第四部分 进阶算法

第 4 部分目录

第 4 章 图论	20
4.1 二分图	20
4.2 拓扑排序	21



第4章 图论

4.1 二分图

二分图常用来判断是否能分成两组

4.1.1 检测二分图

```
1 // 自己写的，经过leetcode检验
2 int mark[N]{0}; // 记录分组
3
4 bool traverse(vector<vector<int>> &graph, int root, int flag = 1, int prev = -1)
5 { // graph: 无向图(双向图)的邻接矩阵; flag: 判断是否是哪组; prev 防止两者来回访问
6     if (!mark[root]) mark[root] = flag;
7     else if (mark[root] != flag) return false;
8     else return true;
9
10    for (auto &node: graph[root])
11        if (node != prev && !traverse(graph, node, -flag, root)) return false;
12
13    return true;
14 }
15
16 bool isBipartite(vector<vector<int>>& graph)
17 { // 主函数
18     int n = graph.size();
19     for (int i = 0; i < n; i++)
20         if (!mark[i] && !traverse(graph, i)) return false;
21     return true;
22 }
```

相关题目：

- LeetCode 886. Possible Bipartition: <https://leetcode.cn/problems/possible-bipartition/>

4.2 拓扑排序

对于图论，直观来说，就是把一副图拉平，而且在这个拉平的图里，所有箭头方向都是一致的。

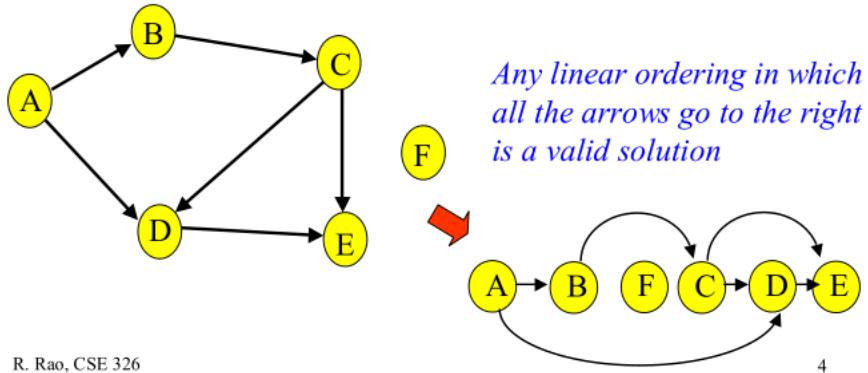


图 4.1: Topological Sorting

- 对于 **DFS**, 将后序遍历的结果进行反转, 就是拓扑排序的结果
- 对于 **BFS** (也就是 **kahn 算法**), 参考代码如下:

```

1 // leetcode problem 课程表2: https://leetcode.cn/problems/course-schedule-ii/submissions/
2 const int N = 2010;
3
4 class Solution {
5 public:
6     bool onPath[N];
7     bool visited[N];
8     int in[N]{0}; // 入度
9     vector<int> q;
10    vector<int> res; // 拓扑排序
11
12    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) { // numCourses:
13        // 所有的课程数目 (也就是生成的graph.size())
14        vector<vector<int>> graph;
15        graph.resize(numCourses);
16        for (auto &v: prerequisites)
17        {
18            graph[v[1]].emplace_back(v[0]);
19            in[v[0]]++;
20        }
21        // 前面只需实现graph以及入度in
22        for (int i = 0; i < numCourses; i++)
23            if (!in[i]) q.emplace_back(i);
24        res = q;
25        while (q.size())
26        {
27            int cur = q.back();
28            q.pop_back();
29            for (int v: graph[cur])
30            {
31                if (--in[v] == 0)
32                    q.emplace_back(v);
33            }
34        }
35    }
36}
```

```
28     for (auto &next: graph[cur])
29     {
30         in[next] --;
31         if (!in[next])
32         {
33             q.emplace_back(next);
34             res.emplace_back(next);
35         }
36     }
37 }
38 if (!(res.size() == numCourses)) return {};
39 return res;
40 }
41 };
```

相关题目：

1. LeetCode 210. Course Schedule II: <https://leetcode.cn/problems/course-schedule-ii/>

第五部分

附录

第 5 部分目录

第 5 章 附录	24
5.1 比赛初始代码模板	24
5.2 <code>unordered_set/map</code> 哈希函数	25
5.3 Big-O Cheat Sheet	26
5.4 数据范围 => 算法复杂度及算法内容	27



第 5 章 附录

◆ 5.1 比赛初始代码模板

```
1 #pragma GCC optimize(3) // 03
2
3 // #define FMT_HEADER_ONLY // comment 线上比赛调试的时候可以用下
4 // #include <fmt/ranges.h> // comment 不过提交的时候记得注释掉
5 // #define pp(v) fmt::print("{}\n", v) // comment
6
7 #include <bits/stdc++.h>
8 using namespace std;
9
10 #define rep(i, a, b) for(int i=a; i < (b); i++)
11 #define trav(a,x) for (auto& a: x)
12 #define all(x) begin(x), end(x)
13 #define rall(x) rbegin(x), rend(x)
14 #define sz(x) (int)(x).size()
15 #define endl "\n" // promote speed
16 typedef long long ll;
17 typedef long double ld;
18 typedef pair<int, int> pii;
19 typedef pair<ll, ll> pll;
20 typedef vector<int> vi;
21
22
23 void solve(int tcase){
24
25 }
26
27 int main(int argc, char *argv[]) {
```

```

28     ios::sync_with_stdio(false); cin.tie(nullptr);
29
30     int T;
31     cin >> T;
32     rep(t, 1, T + 1) {
33         solve(t);
34     }
35 }
```

5.2 unordered_set/map 哈希函数

```

1 // https://oi-wiki.org/lang/csl/unordered-container/#_3
2 #include <chrono>
3 struct my_hash {
4     static uint64_t splitmix64(uint64_t x) {
5         x += 0x9e3779b97f4a7c15;
6         x = (x ^ (x >> 30)) * 0xbfa58476d1ce4e5b9;
7         x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
8         return x ^ (x >> 31);
9     }
10
11     size_t operator()(uint64_t x) const {
12         static const uint64_t FIXED_RANDOM =
13             chrono::steady_clock::now().time_since_epoch().count();
14         return splitmix64(x + FIXED_RANDOM);
15     }
16
17     // 针对 std::pair<int, int> 作为主键类型的哈希函数
18     size_t operator()(pair<uint64_t, uint64_t> x) const {
19         static const uint64_t FIXED_RANDOM =
20             chrono::steady_clock::now().time_since_epoch().count();
21         return splitmix64(x.first + FIXED_RANDOM) ^
22             (splitmix64(x.second + FIXED_RANDOM) >> 1);
23     }
24 };
```

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。

5.3 Big-O Cheat Sheet

5.3.1 Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity	
	Average				Worst					
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion		
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n \log(n))$	
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	
B-Tree	$\Theta(\log(n))$	$O(n)$								
Red-Black Tree	$\Theta(\log(n))$	$O(n)$								
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	
AVL Tree	$\Theta(\log(n))$	$O(n)$								
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	

图 5.1: Common Data Structure Operations

5.3.2 Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$O(n)$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$\Theta(n(\log(n))^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$\Theta(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$\Theta(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$O(n)$

图 5.2: Array Sorting Algorithms

5.3.3 Growth Rates

$n f(n)$	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	0.003ns	0.01ns	0.033ns	0.1ns	1ns	3.65ms
20	0.004ns	0.02ns	0.086ns	0.4ns	1ms	77years
30	0.005ns	0.03ns	0.147ns	0.9ns	1sec	8.4×10^{15} yrs
40	0.005ns	0.04ns	0.213ns	1.6ns	18.3min	--
50	0.006ns	0.05ns	0.282ns	2.5ns	13days	--
100	0.07	0.1ns	0.644ns	0.10ns	4×10^{13} yrs	--
1,000	0.010ns	1.00ns	9.966ns	1ms	--	--
10,000	0.013ns	10ns	130ns	100ms	--	--
100,000	0.017ns	0.10ms	1.67ms	10sec	--	--
1'000,000	0.020ns	1ms	19.93ms	16.7min	--	--
10'000,000	0.023ns	0.01sec	0.23ms	1.16days	--	--
100'000,000	0.027ns	0.10sec	2.66sec	115.7days	--	--
1,000'000,000	0.030ns	1sec	29.90sec	31.7 years	--	--

图 5.3: Growth Rates

5.4 数据范围 => 算法复杂度及算法内容

原地址: <https://www.acwing.com/blog/content/32/>

表 5.1: 由数据范围反推算法复杂度以及算法内容

数据范围	级别	算法选择
$n \leq 30$	指数级别	dfs+剪枝 状态压缩 dp
$n \leq 100$	$O(n^3)$	floyd dp 高斯消元
$n \leq 1000$	$O(n^2)O(n^2 \log n)$	dp 二分 朴素版 Dijkstra 朴素版 Prim Bellman-Ford
$n \leq 10^4$	$O(n\sqrt{n})$	块状链表 分块 莫队
$n \leq 10^5$	$O(n \log n)$	各种 sort 线段树 树状数组 set/map heap 拓扑排序 dijkstra+heap prim+heap Kruskal spfa 求凸包 求半平面交 二分 CDQ 分治 整体二分 后缀数组 树链剖分 动态树
$n \leq 10^6$	$O(n)$	单调队列 hash 双指针扫描 并查集 kmp AC 自动机
	常数较小的 $O(n \log n)$ 算法	sort 树状数组 heap dijkstra spfa
$n \leq 10^7$	$O(n)$	双指针扫描 kmp AC 自动机 线性筛素数
$n \leq 10^9$	$O(\sqrt{n})$	判断质数
$n \leq 10^{18}$	$O(\log n)$	最大公约数 快速幂 数位 DP
$n \leq 10^{1000}$	$O((\log n)^2)$	高精度加减乘除
$n \leq 10^{100000}$	$O(\log k * \log \log k)$, k 表示位数	高精度加减 FFT/NTT