



列奥那多的 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 27, 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	7
1.13 iterator	7
1.14 algorithm	7
1.14.1 Non-modifying sequence operations	7
1.14.2 Modifying sequence operations	8
1.14.3 Partitioning operations	10
1.14.4 Sorting operations	11
1.14.5 Binary search operations (on sorted ranges)	11
1.14.6 Other operations on sorted ranges	11
1.14.7 Set operations (on sorted ranges)	11
1.14.8 Heap operations	12
1.14.9 Minimum/maximum operations	12
1.14.10 Comparison operations	12
1.14.11 Permutation operations	13
1.14.12 Numeric operations	13



2

第 2 部分 * 数据结构

第 2 章	数据结构	15
2.1	单链表	15
2.2	双链表	16
2.3	栈	16
2.3.1	数组模拟栈	16
2.3.2	单调栈	17
2.4	队列	17
2.4.1	普通队列	17
2.4.2	循环队列	18
2.4.3	单调队列	18
2.5	Tire 树	18
2.6	并查集 DSU/UF	20

3

第 3 部分 * 基础算法

第 3 章	基础算法	22
3.1	排序算法	22
3.1.1	快速排序	22
3.1.2	归并排序	22
3.2	二分	23
3.2.1	整数二分	23
3.2.2	浮点数二分	23
3.3	高精度	24
3.3.1	加法	24
3.3.2	减法	24
3.3.3	乘法	25
3.3.4	除法	25
3.4	前缀和	26
3.5	差分	26
3.6	位运算	26
3.7	双指针算法	27
3.8	离散化	27
3.9	区间合并	28
3.10	KMP	28

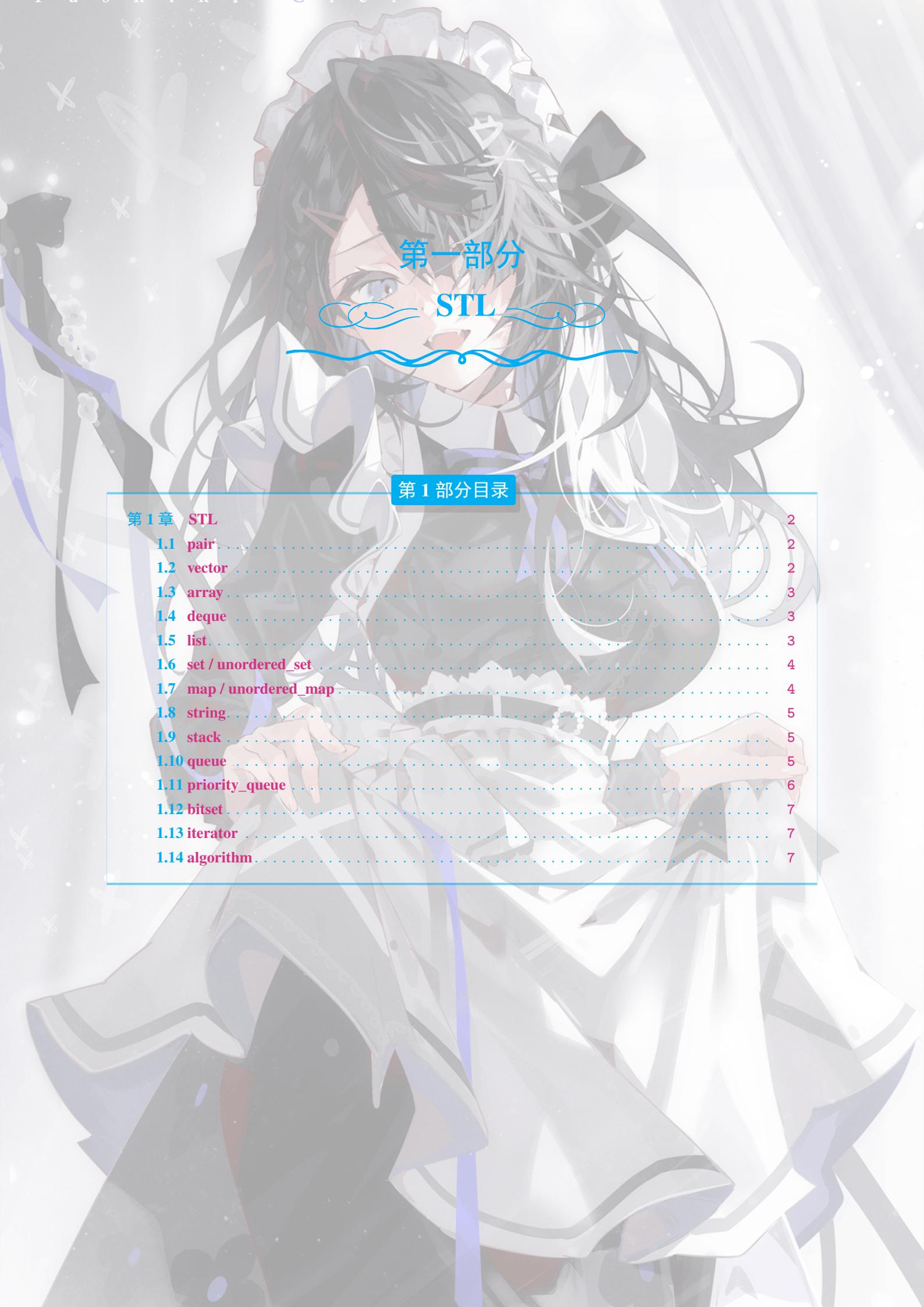
4

第 4 部分 * 进阶算法

第 4 章	图论	31
4.1	二分图	31
4.1.1	检测二分图	31
4.2	拓扑排序	32



第 5 章 附录	35
5.1 比赛初始代码模板	35
5.2 unordered_set/map 哈希函数	36
5.3 Big-O Cheat Sheet	37
5.3.1 Common Data Structure Operations	37
5.3.2 Array Sorting Algorithms	37
5.3.3 Growth Rates	38
5.4 数据范围 => 算法复杂度及算法内容	38



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	7
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 typedef pair<int, int> PII;
3 // big heap & small heap
4 priority_queue<int> big_heap;
5 priority_queue<PII, vector<PII>, less<PII> > big_heap;
6 priority_queue<int, vector<int>, greater<int> > small_heap;
7
8 // operations
9 priority_queue<int> q;
10 q.push(1); q.emplace(1);
11 cout << q.top(); // not empty
12 q.empty(); q.size();
13 q.pop();
14
15 // custom comparator
16 class Node;
17 bool Compare(Node a, Node b);
18 std::priority_queue<Node, std::vector<Node>, decltype(&Compare)> openSet(Compare); // notice & notation
19 // or auto cmp = [] (const auto &a, const auto &b) { return a.second > b.second; }; // small_heap according
20 // to the second element in a pair
21 priority_queue<PII, vector<PII>, decltype(cmp)> pq(cmp); // small_heap // without & notation
22
23 // custom structure
24 struct Status {
25     int val;
26     ListNode *ptr;
27     bool operator < (const Status &rhs) const { // 必须定义<运算符
28         return val > rhs.val; // rhs 小于 当前 val 就交换, 说明是最小堆
29     }
30 };
31 priority_queue<Status> q;
32 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
------------	-----------

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

```

1 vector<int> a(10);
2 iota(a.begin(), a.end(), 0);
3 vector<int> to_vector(10);
4 copy(a.begin(), a.end(), to_vector.begin()); // need to give a size to to_vector
5 // copy(a.begin(), a.end(), back_inserter(to_vector)); // not needed
6 copy_if(a.begin(), a.end(), to_vector.begin(), [](int x){return x % 2 == 0;});
7 copy_n(a.begin(), 2, to_vector.begin()); // copy exactly n element
8 copy_backward(a.begin(), a.end(), to_vector.end()); // note: to_vector.end(), copy direction is backward,
    so need end()
9 // see also: move, move_backward
10
11 fill(a.begin(), a.end(), -1);
12 fill_n(a.begin(), 3, 1); // the first 3 elements of a
13
14 iota(a.begin(), a.end(), 0);
15 transform(a.begin(), a.end(), to_vector.begin(), [](int a){ return a * a; });
16
17 generate(a.begin(), a.end(), [](){ return 1;});
18 generate_n(a.begin(), 2, [](){ return 2;});
19
20 a = {1,2,4,1};

```

```

21 auto b = remove(a.begin(), a.end(), 1); // 把不等于1的数放到前面，返回多余(就是要删除)的元素的开始的指针
22 b = remove_if(a.begin(), a.end(), [](int x){return x % 2 == 0;});
23 a.erase(b);
24
25 a = {1,2,4,1}; to_vector = {};
26 remove_copy(a.begin(), a.end(), back_inserter(to_vector), 1);
27 // remove_copy_if
28
29 a = {1,2,4,1}; to_vector = {};
30 replace(a.begin(), a.end(), 1, 2);
31 replace_if(a.begin(), a.end(), [](int a){return a % 2 == 0; }, 4);
32 replace_copy_if(a.begin(), a.end(), back_inserter(to_vector), [](int a){return a % 2 == 0; }, 4);
33 // replace_copy
34
35 a = {-1,-1,-1}; to_vector = {1,1,1,1};
36 swap(a, to_vector);
37 swap_ranges(a.begin(), a.begin() + 2, to_vector.begin()); // only swap the first 2 elements
38
39 a = {-1,3,2};
40 iter_swap(a.begin(), a.end() - 1); // swap two elements by passing their iterator
41
42 a = {-1,3,2}; to_vector = {};
43 reverse(a.begin(), a.end());
44 reverse_copy(a.begin(), a.end(), back_inserter(to_vector));
45
46 a = {1,2,3,4,5,6};
47 rotate(a.begin(), a.begin() + 2, a.end()); // 3,4,5,6,1,2
48 rotate(a.rbegin(), a.rbegin() + 1, a.rend()); // 6,1,2,3,4,5
49 // rotate_copy
50
51 a = {1,2,3,4,5,6};
52 default_random_engine defaultEngine;
53 shuffle(a.begin(), a.end(), defaultEngine);
54
55 a = {1,1,1,4,5,6};
56 auto x = unique(a.begin(), a.end()); // a: 1,4,5,6,5,6 ; x points to pos -2
57 // unique_copy

```

1.14.3 Partitioning operations

表 1.3: Modifying sequence operations

operations	functions
is_partitioned	determines if the range is partitioned by the given predicate
partition	divides a range of elements into two groups
partition_copy	copies a range dividing the elements into two groups
stable_partition	divides elements into two groups while preserving their relative order

<code>partition_point</code>	locates the partition point of a partitioned range
------------------------------	----------------------------------------------------

1.14.4 Sorting operations

表 1.4: Modifying sequence operations

operations	functions
<code>is_sorted</code>	checks whether a range is sorted into ascending order
<code>is_sorted_until</code>	finds the largest sorted subrange
<code>sort</code>	sorts a range into ascending order
<code>partial_sort</code>	sorts the first N elements of a range
<code>partial_sort_copy</code>	copies and partially sorts a range of elements
<code>stable_sort</code>	sorts a range of elements while preserving order between equal elements
<code>nth_element</code>	partially sorts the given range making sure that it is partitioned by the given element

1.14.5 Binary search operations (on sorted ranges)

表 1.5: Modifying sequence operations

operations	functions
<code>lower_bound</code>	returns an iterator to the first element not less than the given value
<code>upper_bound</code>	returns an iterator to the first element greater than a certain value
<code>binary_search</code>	determines if an element exists in a partially-ordered range
<code>equal_range</code>	returns range of elements matching a specific key

1.14.6 Other operations on sorted ranges

表 1.6: Modifying sequence operations

operations	functions
<code>merge</code>	merges two sorted ranges
<code>inplace_merge</code>	merges two ordered ranges in-place

1.14.7 Set operations (on sorted ranges)

表 1.7: Modifying sequence operations

operations	functions
------------	-----------

<code>includes</code>	returns true if one sequence is a subsequence of another
<code>set_difference</code>	computes the difference between two sets
<code>set_intersection</code>	computes the intersection of two sets
<code>set_symmetric_difference</code>	computes the symmetric difference between two sets
<code>set_union</code>	computes the union of two sets

1.14.8 Heap operations

表 1.8: Modifying sequence operations

operations	functions
<code>is_heap</code>	checks if the given range is a max heap
<code>is_heap_until</code>	finds the largest subrange that is a max heap
<code>make_heap</code>	creates a max heap out of a range of elements
<code>push_heap</code>	adds an element to a max heap
<code>pop_heap</code>	removes the largest element from a max heap
<code>sort_heap</code>	turns a max heap into a range of elements sorted in ascending order

1.14.9 Minimum/maximum operations

表 1.9: Modifying sequence operations

operations	functions
<code>max</code>	returns the greater of the given values
<code>max_element</code>	returns the largest element in a range
<code>min</code>	returns the smaller of the given values
<code>min_element</code>	returns the smallest element in a range
<code>minmax</code>	returns the smaller and larger of two elements
<code>minmax_element</code>	returns the smallest and the largest elements in a range

1.14.10 Comparison operations

表 1.10: Modifying sequence operations

operations	functions
<code>equal</code>	determines if two sets of elements are the same
<code>lexicographical_compare</code>	returns true if one range is lexicographically less than another

1.14.11 Permutation operations

表 1.11: Modifying sequence operations

operations	functions
is_permutation	determines if a sequence is a permutation of another sequence
next_permutation	generates the next greater lexicographic permutation of a range of elements
prev_permutation	generates the next smaller lexicographic permutation of a range of elements

1.14.12 Numeric operations

表 1.12: Modifying sequence operations

operations	functions
iota	fills a range with successive increments of the starting value
accumulate	sums up a range of elements
inner_product	computes the inner product of two ranges of elements
adjacent_difference	computes the differences between adjacent elements in a range
partial_sum	computes the partial sum of a range of elements

第二部分 数据结构

第 2 部分目录

第 2 章 数据结构	15
2.1 单链表	15
2.2 双链表	16
2.3 栈	16
2.4 队列	17
2.5 Tire 树	18
2.6 并查集 DSU/UF	20



第 2 章 数据结构

2.1 单链表

```
1 // head存储链表头, e[]存储节点的值, ne[]存储节点的next指针, idx表示当前用到了哪个节点
2 int head, e[N], ne[N], idx;
3
4 // 初始化
5 void init()
6 {
7     head = -1;
8     idx = 0;
9 }
10
11 // 在链表头插入一个数a
12 void insert(int a)
13 {
14     e[idx] = a, ne[idx] = head, head = idx++;
15 }
16
17 // 将头结点删除, 需要保证头结点存在
18 void remove()
19 {
20     head = ne[head];
21 }
```

2.2 双链表

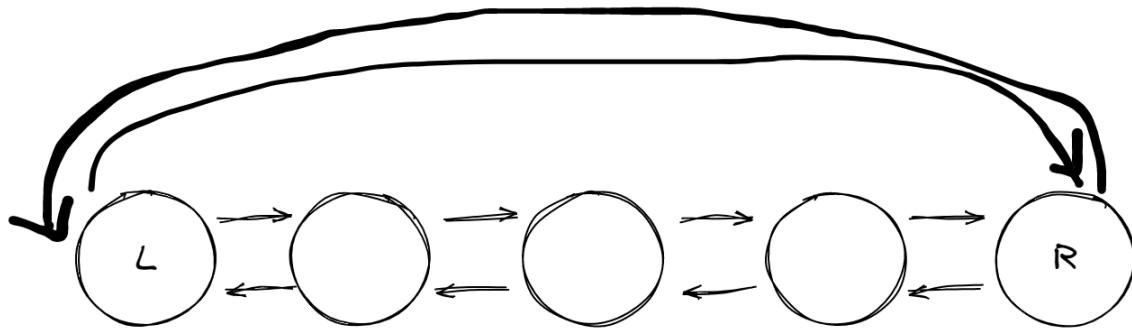


图 2.1: 双链表示意

```

1 // e[] 表示节点的值, l[] 表示节点的左指针, r[] 表示节点的右指针, idx 表示当前用到了哪个节点
2 int e[N], l[N], r[N], idx;
3
4 // 初始化
5 void init()
6 {
7     // 0 是左端点, 1 是右端点
8     r[0] = 1, l[1] = 0;
9     idx = 2;
10 }
11
12 // 在节点a的右边插入一个数x
13 void insert(int a, int x)
14 {
15     e[idx] = x;
16     l[idx] = a, r[idx] = r[a];
17     l[r[a]] = idx, r[a] = idx++;
18 }
19
20 // 删除节点a
21 void remove(int a)
22 {
23     l[r[a]] = l[a];
24     r[l[a]] = r[a];
25 }
```

2.3 栈

2.3.1 数组模拟栈

使用数组模拟栈的代码如下 (stl 中有现成的栈的实现) :

```
1 // tt 表示栈顶
```

```

2 int stk[N], tt = 0;
3
4 // 向栈顶插入一个数
5 stk[ ++ tt] = x;
6
7 // 从栈顶弹出一个数
8 tt -- ;
9
10 // 栈顶的值
11 stk[tt];
12
13 // 判断栈是否为空
14 if (tt > 0)
15 {
16
17 }

```

2.3.2 单调栈

常见模型：找出每个数左边离它最近的比它大/小的数

```

1 // 常见模型：找出每个数左边离它最近的比它大/小的数
2 int tt = 0;
3 for (int i = 1; i <= n; i++)
4 {
5     while (tt && check(stk[tt], i)) tt--;
6     stk[ ++ tt] = i;
7 }

```

2.4 队列

2.4.1 普通队列

```

1 // hh 表示队头，tt表示队尾
2 int q[N], hh = 0, tt = -1;
3
4 // 向队尾插入一个数
5 q[ ++ tt] = x;
6
7 // 从队头弹出一个数
8 hh++;
9
10 // 队头的值
11 q[hh];
12
13 // 判断队列是否为空
14 if (hh <= tt)

```

```
15 {
16
17 }
```

2.4.2 循环队列

```
1 // hh 表示队头, tt表示队尾的后一个位置
2 int q[N], hh = 0, tt = 0;
3
4 // 向队尾插入一个数
5 q[tt ++ ] = x;
6 if (tt == N) tt = 0;
7
8 // 从队头弹出一个数
9 hh ++ ;
10 if (hh == N) hh = 0;
11
12 // 队头的值
13 q[hh];
14
15 // 判断队列是否为空
16 if (hh != tt)
17 {
18 }
```

2.4.3 单调队列

常见模型：找出滑动窗口中的最大值/最小值

```
1 // 常见模型：找出滑动窗口中的最大值/最小值
2 int hh = 0, tt = -1;
3 for (int i = 0; i < n; i ++ )
4 {
5     while (hh <= tt && check_out(q[hh])) hh ++ ; // 判断队头是否滑出窗口
6     while (hh <= tt && check(q[tt], i)) tt -- ;
7     q[ ++ tt] = i;
8 }
```

2.5 Tire 树

字典树，是一种空间换时间的数据结构，又称 Trie 树、前缀树，是一种树形结构（字典树是一种数据结构），典型用于统计、排序、和保存大量字符串。所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。

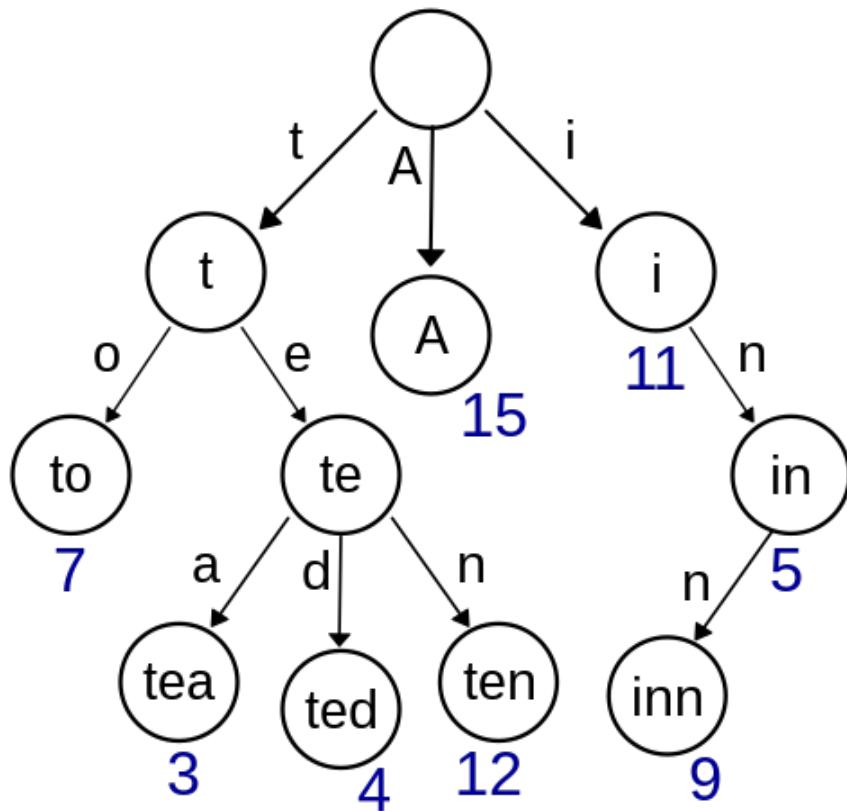


图 2.2: Tire 树

```

1 int son[N][26], cnt[N], idx;
2 // 0号点既是根节点，又是空节点
3 // son[] 存储树中每个节点的子节点
4 // cnt[] 存储以每个节点结尾的单词数量
5
6 // 插入一个字符串
7 void insert(char *str)
8 {
9     int p = 0;
10    for (int i = 0; str[i]; i++)
11    {
12        int u = str[i] - 'a';
13        if (!son[p][u]) son[p][u] = ++idx;
14        p = son[p][u];
15    }
16    cnt[p]++;
17 }
18
19 // 查询字符串出现的次数
20 int query(char *str)
21 {
22     int p = 0;
23     for (int i = 0; str[i]; i++)
24     {

```

```

25     int u = str[i] - 'a';
26     if (!son[p][u]) return 0;
27     p = son[p][u];
28 }
29 return cnt[p];
30 }
```

2.6 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	22
3.1 排序算法	22
3.2 二分	23
3.3 高精度	24
3.4 前缀和	26
3.5 差分	26
3.6 位运算	26
3.7 双指针算法	27
3.8 离散化	27
3.9 区间合并	28
3.10 KMP	28



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：

1. 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
```

3.6 位运算

oi-wiki: <https://oi-wiki.org/math/bit/>

1. 异或运算的逆运算是它本身，也就是说两次异或同一个数最后结果不变，即 $a \text{ xor } b \text{ xor } b = 0$ 。
2. 操作二进制的某位

```

1 // 以下最低位编号为0
2 // 获得a二进制的第b位
3 int getBit(int a, int b) { return (a >> b) & 1; }
4 // 把a二进制的第b位设置为0
5 int unsetBit(int a, int b) { return a & ~(1 << b); }
6 // 把a二进制的第b位设置为1
7 int setBit(int a, int b) { return a | (1 << b); }
8 // 将 a 的第 b 位取反，最低位编号为 0
```

```
9 int flapBit(int a, int b) { return a ^ (1 << b); }
```

3. 判断两非零数符号是否相同

```
1 bool isSameSign(int x, int y) { // x,y均不为0
2     return (x ^ y) >= 0;
3 }
```

4. 求 n 的第 k 位数字

```
1 n >> k & 1
```

5. 返回 n 的最后一位 1

```
1 int lowbit(int n){ return n & -n; }
2 // 例如 4 & -4 == 4 ; 5 & -5 = 1
```

6. 模拟集合操作

一个数的二进制表示可以看作是一个集合（0 表示不在集合中，1 表示在集合中）。比如集合 1,3,4,8，可以表示成 $(100011010)_2$ 。而对应的位运算也就可以看作是对集合进行的操作。更多见 [oi-wiki](#)

3.7 双指针算法

常见问题分类：

1. 对于一个序列，用两个指针维护一段区间
2. 对于两个序列，维护某种次序，比如归并排序中合并两个有序序列的操作

```
1 for (int i = 0, j = 0; i < n; i ++ )
2 {
3     while (j < i && check(i, j)) j ++ ;
4
5     // 具体问题的逻辑
6 }
```

3.8 离散化

一般用于给定范围很大，但所需范围较小的题

```
1 vector<int> alls; // 存储所有待离散化的值
2 sort(alls.begin(), alls.end()); // 将所有值排序
3 alls.erase(unique(alls.begin(), alls.end()), alls.end()); // 去掉重复元素
4
5 // 二分求出x对应的离散化的值
6 int find(int x) // 找到第一个大于等于x的位置
7 {
8     int l = 0, r = alls.size() - 1;
9     while (l < r)
10    {
11        int mid = l + r >> 1;
12        if (alls[mid] >= x) r = mid;
```

```

13     else l = mid + 1;
14 }
15 return r + 1; // 映射到1, 2, ...n
16 }
```

相关题目：

- AcWing 802. 区间和: <https://www.acwing.com/problem/content/804/>

3.9 区间合并

```

1 // 将所有存在交集的区间合并
2 void merge(vector<PII> &segs)
3 {
4     vector<PII> res;
5
6     sort(segs.begin(), segs.end());
7
8     int st = -2e9, ed = -2e9;
9     for (auto seg : segs)
10    {
11        if (ed < seg.first)
12        {
13            if (st != -2e9) res.push_back({st, ed});
14            st = seg.first, ed = seg.second;
15        }
16        else ed = max(ed, seg.second);
17
18        if (st != -2e9) res.push_back({st, ed});
19
20    }
}
```

3.10 KMP

```

1 // s[]是长文本, p[]是模式串, n是s的长度, m是p的长度
2 // 求模式串的Next数组(ne):
3 for (int i = 2, j = 0; i <= m; i++)
4 {
5     while (j && p[i] != p[j + 1]) j = ne[j];
6     if (p[i] == p[j + 1]) j++;
7     ne[i] = j;
8 }
9
10 // 匹配
11 for (int i = 1, j = 0; i <= n; i++)
12 {
13     while (j && s[i] != p[j + 1]) j = ne[j];
14     if (s[i] == p[j + 1]) j++;
15 }
```

```
15 if (j == m)  
16 {  
17     j = ne[j];  
18     // 匹配成功后的逻辑  
19 }  
20 }
```



第四部分 进阶算法

第4部分目录

第4章 图论	31
4.1 二分图	31
4.2 拓扑排序	32



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。

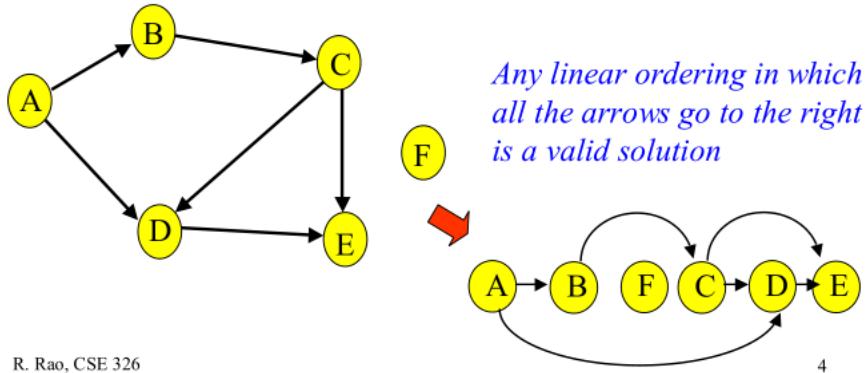


图 4.1: Topological Sorting

- 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) q.emplace_back(v);
32            }
33        }
34    }
35}
```

```
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	35
5.1 比赛初始代码模板	35
5.2 <code>unordered_set/map</code> 哈希函数	36
5.3 Big-O Cheat Sheet	37
5.4 数据范围 => 算法复杂度及算法内容	38



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