目录

[C++ STL和常用算法模板 3](#_Toc21192)

[1 容器 Container 3](#_Toc25421)

[1.1向量 vector 3](#_Toc13750)

[1.2栈 stack 4](#_Toc20158)

[1.3队列 queue 4](#_Toc15948)

[1.4优先队列 priority\_queue 4](#_Toc15718)

[#include <queue> 5](#_Toc7609)

[1.5集合Set 5](#_Toc20237)

[#include <set> 5](#_Toc17051)

[1.6映射Map 7](#_Toc5635)

[#include <map> 7](#_Toc896)

[1.6.1 常用方法 7](#_Toc28187)

[1.6.2 适用情形 7](#_Toc5766)

[1.6.3 注意事项 7](#_Toc8259)

[1.7字符串String 8](#_Toc1794)

[1.7.1 常用方法 8](#_Toc24599)

[1.7.2 适用情形 8](#_Toc8528)

[1.7.3 注意事项 8](#_Toc23272)

[1.8 二元组 pair 9](#_Toc20033)

[2 算法 Algorithm 10](#_Toc3079)

[2.1 swap() 10](#_Toc25269)

[2.2 sort() 10](#_Toc26708)

[2.3 lower\_bound() / upper\_bound() 10](#_Toc19825)

[2.4 reverse() 11](#_Toc17248)

[2.5 max() / min() 11](#_Toc7211)

[2.6 unique() 11](#_Toc8881)

[2.7 Math 11](#_Toc23133)

[2.8 gcd() / lcm() 12](#_Toc6100)

[3 算法模板 Template 13](#_Toc8040)

[3.1 埃式筛 埃拉托斯特尼筛法 Eratosthenes 13](#_Toc16495)

[3.2 欧拉筛 线性筛 Euler 13](#_Toc3847)

[3.3 二分查找 Binary Search 13](#_Toc17260)

[3.3.1 ≥x 13](#_Toc26148)

[3.3.2 ≤x≤ 13](#_Toc7411)

[3.3.3 实数 13](#_Toc16270)

[3.4 三分查找 14](#_Toc22230)

[3.5 深度优先搜索 DFS 14](#_Toc22304)

[3.6 广度优先搜索 BFS 14](#_Toc31117)

[3.7 辗转相除法 欧几里得算法 Euclidean algorithm 15](#_Toc1877)

[3.8 快速幂 Exponentiation by squaring 15](#_Toc8704)

[3.8.1 不取模 15](#_Toc23655)

[3.8.2 取模 15](#_Toc19115)

[3.9 KMP 算法 The Knuth-Morris-Pratt Algorithm 15](#_Toc12642)

[3.9.1 类封装 15](#_Toc2790)

[3.9.1 旧模板 16](#_Toc11988)

[3.10 Dijkstra 算法 17](#_Toc25933)

[3.10.1 朴素（适合稠密图） 18](#_Toc4)

[3.10.2 堆优化（适合稀疏图） 18](#_Toc18088)

[3.11 Floyd-Warshall 算法 18](#_Toc31444)

[3.12 最短路径快速算法 SPFA 19](#_Toc8377)

[3.12.1 最短路 19](#_Toc6073)

[3.12.2 判负权回路 19](#_Toc10463)

[3.13 克鲁斯卡尔算法 Kruskal 20](#_Toc17005)

[3.14 普林姆算法 Prim 20](#_Toc16496)

[3.14.1 朴素（适合稠密图） 20](#_Toc4430)

[3.14.2 堆优化 21](#_Toc32049)

[3.15 排序算法 Sort 21](#_Toc8320)

[2.16.2 最大匹配 匈牙利算法 Hungarian Algorithm 21](#_Toc14944)

[3.17 背包模型 22](#_Toc30536)

[3.17.1 0/1 背包 22](#_Toc7971)

[3.17.2 完全背包 23](#_Toc20811)

[3.17.3 多重背包 24](#_Toc11611)

[3.17.4 分组背包 25](#_Toc12931)

[3.17.5 混合背包 26](#_Toc17455)

[3.18 高精度 Big Integer 26](#_Toc9265)

[3.18.1 I/O 26](#_Toc7112)

[3.18.2 BI + BI 26](#_Toc10777)

[3.18.3 BI - BI 27](#_Toc11287)

[3.18.4 BI \* I 27](#_Toc15038)

[3.18.5 BI / I 27](#_Toc16025)

[3.19 贝尔曼-福特算法 Bellman-Ford 28](#_Toc6789)

[3.20 矩阵加速算法 28](#_Toc11128)

[3.20.1 n 阶方阵乘法 28](#_Toc14566)

[3.20.2 矩阵快速幂 28](#_Toc1149)

[3.21 最长上升子序列 Longest Increasing Subsequence 29](#_Toc26633)

[3.21.1 动态规划 29](#_Toc17570)

[3.21.2 贪心、二分、单调栈 29](#_Toc19085)

[3.22 最长公共上升子序列 Longest Common Increasing Subsequence 29](#_Toc4438)

[3.22.1 三重循环 DP 29](#_Toc28749)

[3.22.2 将 DP 进行简化 30](#_Toc17063)

[3.23 最近公共祖先 Lowest Common Ancestor 30](#_Toc8781)

[3.23.1 倍增法 30](#_Toc20973)

[3.23.2 树链剖分 31](#_Toc8008)

[3.24 树链剖分（重链剖分） 31](#_Toc2738)

[3.24.1 剖分 31](#_Toc21875)

[3.24.2 操作 32](#_Toc30692)

[3.25 字符串哈希 32](#_Toc19299)

[3.25.1 哈希 32](#_Toc6042)

[3.25.2 子串哈希 33](#_Toc14666)

[3.25.3 允许 k 次失配的匹配 33](#_Toc29165)

[3.26 Manacher 算法 34](#_Toc7388)

[2.26.1 预处理 34](#_Toc21431)

[3.26.2 马拉车 34](#_Toc9629)

[3.27 拓扑排序 Topo Sort 34](#_Toc13370)

[3.28 莫队 Mo's Algorithm 35](#_Toc1129)

[3.28.1 普通莫队 35](#_Toc17653)

[3.28.2 带修改莫队 35](#_Toc16503)

# C++ STL和常用算法模板

#include<iostream> #include<algorithm> #include<string.h> #include<stdlib.h>

#include<queue> #include<stack> #include<vector> #include<set>

#include<map>

using namespace std;

# 1 容器 Container

**1.1向量 [vector](https://zh.cppreference.com/w/cpp/container/vector" \t "https://io.zouht.com/_blank)**

**#include <vector>**

连续的顺序的储存结构（和数组一样的类别），但是有长度可变的特性。

#### 1.1.1 常用方法

##### (1)构造

**vector<类型> arr(长度, [初值])**

常用的一维和二维数组构造示例，高维也是一样的（就是会有点长）.

vector<int> arr; // 构造int数组

vector<int> arr(100); // 构造初始长100的int数组

vector<int> arr(100, 1); // 构造初始长100的int数组，初值为1

vector<vector<int>> mat(100, vector<int> ()); // 构造初始100行，不指定列数的二维数组

vector<vector<int>> mat(100, vector<int> (666, -1)) // 构造初始100行，初始666列的二维数组，初值为-1

##### (2)尾接 & 尾删

* **.push\_back(元素)**：在 vector 尾接一个元素，数组长度 +1.
* **.pop\_back()**：删除 vector 尾部的一个元素，数组长度 -1

// init: arr = []

arr.push\_back(1);// after: arr = [1]

arr.push\_back(2);// after: arr = [1, 2]

arr.pop\_back();// after: arr = [1]

arr.pop\_back();// after: arr = []

##### (3)中括号运算符

和一般数组一样的作用

##### (4)获取长度

**.size()**

获取当前 vector 的长度

for (int i = 0; i < arr.size(); i++)

cout << a[i] << endl;

##### (5)清空

**.clear()**

清空 vector

##### (6)判空

**.empty()**

如果是空返回 true 反之返回 false.

##### (7)改变长度

**.resize(新长度, [默认值])**

修改 vector 的长度

* 如果是缩短，则删除多余的值
* 如果是扩大，且指定了默认值，则新元素均为默认值****（旧元素不变）****

#### 1.1.2 适用情形

一般情况 vector 可以替换掉普通数组，除非该题卡常。

有些情况普通数组没法解决：n×m的矩阵，1≤n,m≤10^6 且 n×m≤10^6

* 如果用普通数组 int mat[1000010][1000010]，浪费内存，会导致 MLE。
* 如果使用 vector<vector<int>> mat(n + 10, vector<int> (m + 10))，完美解决该问题。

另外，vector 的数据储存在堆空间中，不会爆栈。

#### 1.1.3 注意事项

##### (1)提前指定长度

如果长度已经确定，那么应当直接在构造函数指定长度，而不是一个一个 .push\_back(). 因为 vector 额外内存耗尽后的重分配是有时间开销的，直接指定长度就不会出现重分配了。

// 优化前: 522ms

vector<int> a;for (int i = 0; i < 1e8; i++)

a.push\_back(i);

// 优化后: 259msvector<int> a(1e8);for (int i = 0; i < a.size(); i++)

a[i] = i;

##### (2)当心 size\_t 溢出

vector 获取长度的方法 .size() 返回值类型为 size\_t，通常 OJ 平台使用的是 32 位编译器（有些平台例如 cf 可选 64 位），那么该类型范围为 [0,2^32).

vector<int> a(65536);long long a = a.size() \* a.size(); // 直接溢出变成0了

**1.2栈** **[stack](https://zh.cppreference.com/w/cpp/container/stack" \t "https://io.zouht.com/_blank)**

**#include <stack>**

通过二次封装双端队列 (deque) 容器，实现先进后出的栈数据结构。

#### 1.2.1 常用方法

| **作用** | **用法** | **示例** |
| --- | --- | --- |
| 构造 | stack<类型> stk | stack<int> stk; |
| 进栈 | .push(元素) | stk.push(1); |
| 出栈 | .pop() | stk.pop(); |
| 取栈顶 | .top() | int a = stk.top(); |
| 查看大小 / 判空 | .size() .empty() | stk.size() stk.empty() |

#### 1.2.2 适用情形

如果不卡常的话，就可以直接用它而不需要手写栈了。

另外，vector 也可以当栈用，vector 的 .back() 取尾部元素，就相当于取栈顶，.push\_back() 相当于进栈，.pop\_back() 相当于出栈。

#### 1.2.3 注意事项

不可访问内部元素！****下面都是错误用法****

for (int i = 0; i < stk.size(); i++)

cout << stk[i] << endl;for (auto ele : stk)

cout << stk << endl;

**1.3队列 [queue](https://zh.cppreference.com/w/cpp/container/queue" \t "https://io.zouht.com/_blank)**

**#include <queue>**

通过二次封装双端队列 (deque) 容器，实现先进先出的队列数据结构。

#### 1.3.1 常用方法

| **作用** | **用法** | **示例** |
| --- | --- | --- |
| 构造 | queue<类型> que | queue<int> que; |
| 进队 | .push(元素) | que.push(1); |
| 出队 | .pop() | que.pop(); |
| 取队首 | .front() | int a = que.front(); |
| 取队尾 | .back() | int a = que.back(); |
| 查看大小 / 判空 | 略 | 略 |

#### 1.3.2 适用情形

如果不卡常的话，就可以直接用它而不需要手写队列了。

#### 1.3.3 注意事项

不可访问内部元素！****下面都是错误用法****

for (int i = 0; i < que.size(); i++)

cout << que[i] << endl;for (auto ele : que)

cout << ele << endl;

**1.4优先队列 [priority\_queue](https://zh.cppreference.com/w/cpp/container/priority_queue" \t "https://io.zouht.com/_blank)**

**#include <queue>**

提供常数时间的最大元素查找，对数时间的插入与提取，底层原理是二叉堆。

#### 1.4.1 常用方法

##### (1)构造

**priority\_queue<类型, 容器, 比较器> pque**

* 类型：要储存的数据类型
* 容器：储存数据的底层容器，默认为 vector<类型>，竞赛中保持默认即可
* 比较器：比较大小使用的比较器，默认为 less<类型>，可自定义

priority\_queue<int> pque1; // 储存int的大顶堆

priority\_queue<int, vector<int>, greater<int>> pque2; // 储存int的小顶堆

##### (2)函数

| **作用** | **用法** | **示例** |
| --- | --- | --- |
| 进堆 | .push(元素) | que.push(1); |
| 出堆 | .pop() | que.pop(); |
| 取堆顶 | .top() | int a = que.top(); |
| 查看大小 / 判空 | 略 | 略 |

进出队复杂度 O(logn)，取堆顶 O(1).

#### 1.4.2 适用情形

持续维护元素的有序性：每次向队列插入大小不定的元素，或者每次从队列里取出大小最小/最大的元素，元素数量 n，插入操作数量 k.

* 每次插入后进行快速排序：k⋅nlogn
* 使用优先队列维护：k⋅logn

#### 1.4.3 注意事项

##### (1)仅堆顶可读

只可访问堆顶，其他元素都无法读取到。****下面是错误用法：****

cout << pque[1] << endl;

##### (2)所有元素不可写

堆中所有元素是不可修改的。****下面是错误用法：****

pque[1] = 2;

pque.top() = 1;

如果你恰好要修改的是堆顶元素，那么是可以完成的：

int tp = pque.top();

pque.pop();

pque.push(tp + 1);

**1.5集合Set**

**#include <set>**

提供对数时间的插入、删除、查找的集合数据结构。底层原理是红黑树。

| **集合三要素** | **解释** | **set** | **multiset** | **unordered\_set** |
| --- | --- | --- | --- | --- |
| 确定性 | 一个元素要么在集合中，要么不在 | ✔ | ✔ | ✔ |
| 互异性 | 一个元素仅可以在集合中出现一次 | ✔ | ❌（任意次） | ✔ |
| 无序性 | 集合中的元素是没有顺序的 | ❌（从小到大） | ❌（从小到大） | ✔ |

#### 1.5.1 常用方法

##### (1)构造

**set<类型, 比较器> st**

* 类型：要储存的数据类型
* 比较器：比较大小使用的比较器，默认为 less<类型>，可自定义

set<int> st1; // 储存int的集合（从小到大）

set<int, greater<int>> st2; // 储存int的集合（从大到小）

##### (2)遍历

可使用迭代器进行遍历：

for (set<int>::iterator it = st.begin(); it != st.end(); ++it)

cout << \*it << endl;

基于范围的循环（C++ 11）：

for (auto &ele : st)

cout << ele << endl;

##### (3)函数

| **作用** | **用法** | **示例** |
| --- | --- | --- |
| 插入元素 | .insert(元素) | st.insert(1); |
| 删除元素 | .erase(元素) | st.erase(2); |
| 查找元素 | .find(元素) | auto it = st.find(1); |
| 判断元素是否存在 | .count(元素) | st.count(3); |
| 查看大小 / 清空 / 判空 | 略 | 略 |

增删查时间复杂度均为 O(logn)

#### 1.5.2 适用情形

* 元素去重：[1,1,3,2,4,4]→[1,2,3,4][1,1,3,2,4,4]→[1,2,3,4]
* 维护顺序：[1,5,3,7,9]→[1,3,5,7,9][1,5,3,7,9]→[1,3,5,7,9]
* 元素是否出现过：元素大小 [−10^18,10^18]，元素数量 10^6，vis 数组无法实现，通过 set 可以完成。

#### 1.5.3 注意事项

##### (1)不存在下标索引

set 虽说可遍历，但仅可使用迭代器进行遍历，它不存在下标这一概念，无法通过下标访问到数据。****下面是错误用法：****

cout << st[0] << endl;

##### (2)元素只读

set 的迭代器取到的元素是只读的（因为是 const 迭代器），不可修改其值。如果要改，需要先 erase 再 insert. ****下面是错误用法：****

cout << \*st.begin() << endl; // 正确。可读。

\*st.begin() = 1; // 错误！不可写！

##### (3)不可用迭代器计算下标

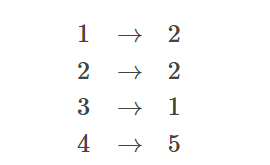
set 的迭代器不能像 vector 一样相减得到下标。****下面是错误用法：****

auto it = st.find(2); // 正确，返回2所在位置的迭代器。int idx = it - st.begin(); // 错误！不可相减得到下标。

**1.6映射Map**

**#include <map>**

提供对数时间的有序键值对结构。底层原理是红黑树。

映射：

| **性质** | **解释** | **map** | **multimap** | **unordered\_map** |
| --- | --- | --- | --- | --- |
| 互异性 | 一个键仅可以在映射中出现一次 | ✔ | ❌（任意次） | ✔ |
| 无序性 | 键是没有顺序的 | ❌（从小到大） | ❌（从小到大） | ✔ |

### 1.6.1 常用方法

#### (1)构造

**map<键类型, 值类型, 比较器> mp**

* 键类型：要储存键的数据类型
* 值类型：要储存值的数据类型
* 比较器：键比较大小使用的比较器，默认为 less<类型>，可自定义

map<int, int> mp1; // int->int 的映射（键从小到大）

map<int, int, greater<int>> st2; // int->int 的映射（键从大到小）

#### (2)遍历

可使用迭代器进行遍历：

for (map<int, int>::iterator it = mp.begin(); it != mp.end(); ++it)

cout << it->first << ' ' << it->second << endl;

基于范围的循环（C++ 11）：

for (auto &pr : mp)

cout << pr.first << ' ' << pr.second << endl;

结构化绑定 + 基于范围的循环（C++17）：

for (auto &[key, val] : mp)

cout << key << ' ' << val << endl;

#### (3)函数

| **作用** | **用法** | **示例** |
| --- | --- | --- |
| 增 / 改 / 查元素 | 中括号 | mp[1] = 2; |
| 查元素（返回迭代器）  若没找到，返回map.end() | .find(元素) | auto it = mp.find(1); |
| 删除元素 | .erase(元素) | mp.erase(2); |
| 判断元素是否存在 | .count(元素) | mp.count(3); |
| 查看大小 / 清空 / 判空 | .size() .clear()  .empty() | 略 |

增删改查时间复杂度均为 O(logn)

### 1.6.2 适用情形

需要维护映射的场景可以使用：输入若干字符串，统计每种字符串的出现次数。(map<string, int> mp)

### 1.6.3 注意事项

#### (1)中括号访问时默认值

如果使用中括号访问 map 时对应的键不存在，那么会新增这个键，并且值为默认值，因此中括号会影响键的存在性。

map<char, int> mp;

cout << mp.count('a') << endl; // 0

mp['a']; // 即使什么都没做，此时mp['a']=0已经插入了

cout << mp.count('a') << endl; // 1

cout << mp['a'] << endl; // 0

#### (2)不可用迭代器计算下标

map 的迭代器不能像 vector 一样相减得到下标。****下面是错误用法：****

auto it = mp.find('a'); // 正确，返回2所在位置的迭代器。int idx = it - mp.begin(); // 错误！不可相减得到下标

**1.7字符串String**

**#include <string>**

顾名思义，就是储存字符串的。

### 1.7.1 常用方法

#### (1)构造

构造函数：string(长度, 初值)

string s1; // 构造字符串，为空

string s2 = "awa!"; // 构造字符串，并赋值awa!

string s3(10, '6'); // 构造字符串，通过构造函数构造为6666666666

#### (2)输入输出

C++

string s;

cin >> s;

cout << s;

C

string s;char buf[100];scanf("%s", &buf);

s = buf;printf("%s", s.c\_str());

#### (3)函数

| **作用** | **用法** | **示例** |
| --- | --- | --- |
| 修改、查询指定下标字符 | [] | s[1] = 'a'; |
| 是否相同 | == | if (s1 == s2) ... |
| 字符串连接 | + | string s = s1 + s2; |
| 尾接字符串 | += | s += "awa"; |
| 取子串 | .substr(起始下标, 子串长度) | string sub = s.substr(2, 10); |
| 查找字符串 | .find(字符串, 起始下标) | int pos = s.find("awa"); |
| 查找首次和末次出现的位置 | .find\_first\_of（str）  .find\_last\_of（str） | int index=s.find\_last\_of(c) |

#### 如果没有找到，那么会返回一个特别的标记npos，一般写作string::npos。

#### (4)数值与字符串互转（C++11）

| **源** | **目的** | **函数** |
| --- | --- | --- |
| int / long long / float / double / long double | string | to\_string() |
| string | int | stoi() |
| string | long long | stoll() |
| string | float | stof() |
| string | double | stod() |
| string | long double | stold() |

### 1.7.2 适用情形

非常好用！~~建议直接把字符数组扔了，赶快投入 string 的怀抱~~

### 1.7.3 注意事项

#### 尾接字符串一定要用 +=

string 的 += 运算符，将会在原字符串原地尾接字符串。而 + 了再 = 赋值，会先生成一个临时变量，在复制给 string.

通常字符串长度可以很长，如果使用 + 字符串很容易就 TLE 了。

// 优化前: 15139ms

string s;for (int i = 0; i < 5e5; i++)

s = s + "a";

// 优化后: < 1ms (计时器显示0)

string s;for (int i = 0; i < 5e5; i++)

s += "a";

#### .substr() 方法的奇葩参数

一定要注意，C++ string 的取子串的第一个参数是****子串起点下标****，第二个参数是****子串长度****。

第二个参数不是子串终点！不是子串终点！要与 java 等其他语言区分开来。

#### .find() 方法的复杂度

该方法实现为暴力实现，时间复杂度为 O(n^2)

### **1.8 二元组** [pair](https://zh.cppreference.com/w/cpp/utility/pair" \t "https://io.zouht.com/_blank)

#include <utility>

顾名思义，就是储存二元组的。

#### 1.8.1 常用方法

##### (1)构造

**pair<第一个值类型, 第二个值类型> pr**

* 第一个值类型：要储存的第一个值的数据类型
* 第二个值类型：要储存的第二个值的数据类型

pair<int, int> p1;

pair<int, long long> p2;

pair<char, int> p3;// ...

##### (2)赋值

老式

pair<int, char> pr = make\_pair(1, 'a');

列表构造 C++11

pair<int, char> pr = {1, 'a'};

##### (3)取值

直接取值

* 取第一个值：.first
* 取第二个值：.second

pair<int, char> pr = {1, 'a'};int awa = pr.first;char bwb = pr.second;

结构化绑定 C++17

pair<int, char> pr = {1, 'a'};auto &[awa, bwb] = pr;

##### (4)判同

直接用 == 运算符

pair<int, int> p1 = {1, 2};

pair<int, int> p2 = {1, 3};if (p1 == p2) { ... } // false

#### 1.8.2 适用场景

所有需要二元组的场景均可使用，效率和自己定义结构体差不多。

#### 1.8.3 注意事项

无

# 2 算法 Algorithm

## **2.1**swap()

交换两个变量的值

****用法示例****

template< class T >void swap( T& a, T& b );

int a = 0, b = 1;swap(a, b);

// now a = 1, b = 0

int arr[10] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

swap(arr[4], arr[6]);

// now arr = {0, 1, 2, 3, 6, 5, 4, 7, 8, 9}

## **2.2**sort()

使用快速排序给一个可迭代对象排序

****用法示例****

template< class RandomIt, class Compare >void sort( RandomIt first, RandomIt last, Compare comp );

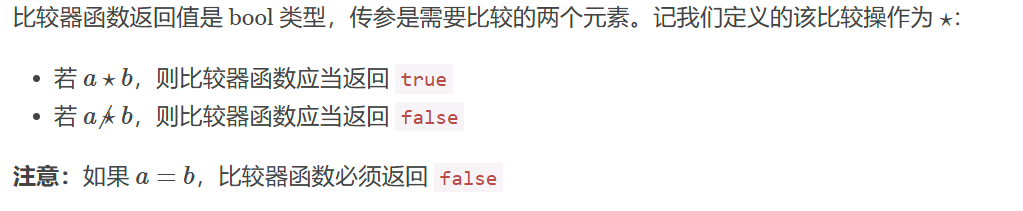
默认排序从小到大

vector<int> arr{1, 9, 1, 9, 8, 1, 0};sort(arr.begin(), arr.end());// arr = [0, 1, 1, 1, 8, 9, 9]

如果要从大到小，则需要传比较器进去。

vector<int> arr{1, 9, 1, 9, 8, 1, 0};sort(arr.begin(), arr.end(), greater<int>());// arr = [9, 9, 8, 1, 1, 1, 0]

如果需要完成特殊比较，则需要手写比较器。



bool cmp(pair<int, int> a, pair<int, int> b){

if (a.second != b.second)

return a.second < b.second;

return a.first > b.first;

}

int main(){

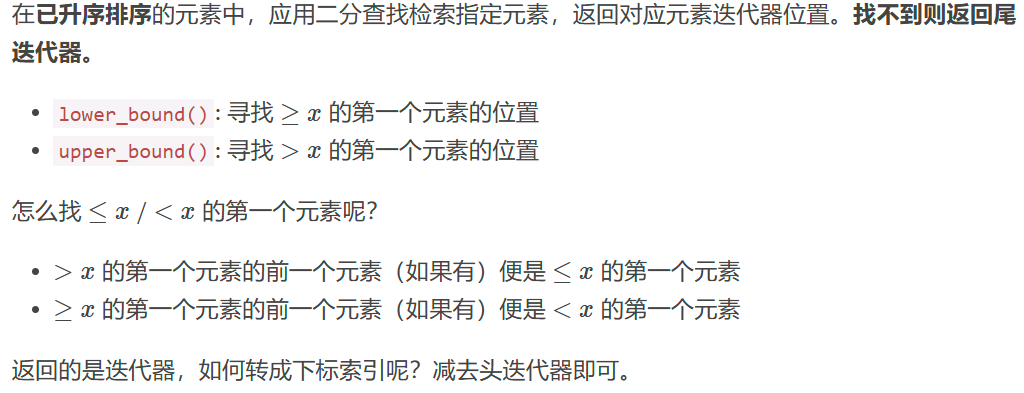
vector<pair<int, int>> arr{{1, 9}, {2, 9}, {8, 1}, {0, 0}};

sort(arr.begin(), arr.end(), cmp);

// arr = [(0, 0), (8, 1), (2, 9), (1, 9)]

}

## **2.3** lower\_bound() / upper\_bound()



****用法示例****

template< class ForwardIt, class T >

ForwardIt lower\_bound( ForwardIt first, ForwardIt last, const T& value );

vector<int> arr{0, 1, 1, 1, 8, 9, 9};

vector<int>::iterator it = lower\_bound(arr.begin(), arr.end(), 7);int idx = it - arr.begin();// idx = 4

我们通常写成一行：

vector<int> arr{0, 1, 1, 1, 8, 9, 9};

idx = lower\_bound(arr.begin(), arr.end(), 7) - arr.begin(); // 4

idx = lower\_bound(arr.begin(), arr.end(), 8) - arr.begin(); // 4

idx = upper\_bound(arr.begin(), arr.end(), 7) - arr.begin(); // 4

idx = upper\_bound(arr.begin(), arr.end(), 8) - arr.begin(); // 5

## **2.4**reverse()

反转一个可迭代对象的元素顺序,注意arr.end()是终点的后一位

****用法示例****

template< class BidirIt >void reverse( BidirIt first, BidirIt last );

vector<int> arr(10);iota(arr.begin(), arr.end(), 1); // 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

reverse(arr.begin(), arr.end()); // 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

## **2.5** max() / min()

返回最大值 / 最小值的****数值****

****用法示例****

int mx = max(1, 2); // 2int mn = min(1, 2); // 1

在 C++11 之后，可以使用列表构造语法传入一个列表，这样就能一次性给多个元素找最大值而不用套娃了：

// Before C++11int mx = max(max(1, 2), max(3, 4)); // 4int mn = min(min(1, 2), min(3, 4)); // 1

// After C++11int mx = max({1, 2, 3, 4}); // 4int mn = min({1, 2, 3, 4}); // 1

## **2.6** unique()

消除数组的重复相邻元素，数组长度不变，但是有效数据缩短，返回的是有效数据位置的结尾迭代器。

例如：[1,1,4,5,1,4]→[1,4,5,1,4,?–][1,1,4,5,1,4]→[1,4,5,1,4,?\_]，下划线位置为返回的迭代器指向。

template< class ForwardIt >

ForwardIt unique( ForwardIt first, ForwardIt last );

****用法示例****

单独使用 unique 并不能达成去重效果，因为它只消除相邻的重复元素。但是如果序列有序，那么它就能去重了。

但是它去重后，序列尾部会产生一些无效数据：[1,1,2,4,4,4,5]→[1,2,4,5,?\_,?,?]，为了删掉这些无效数据，我们需要结合 erase.

最终，给 vector 去重的写法便是：

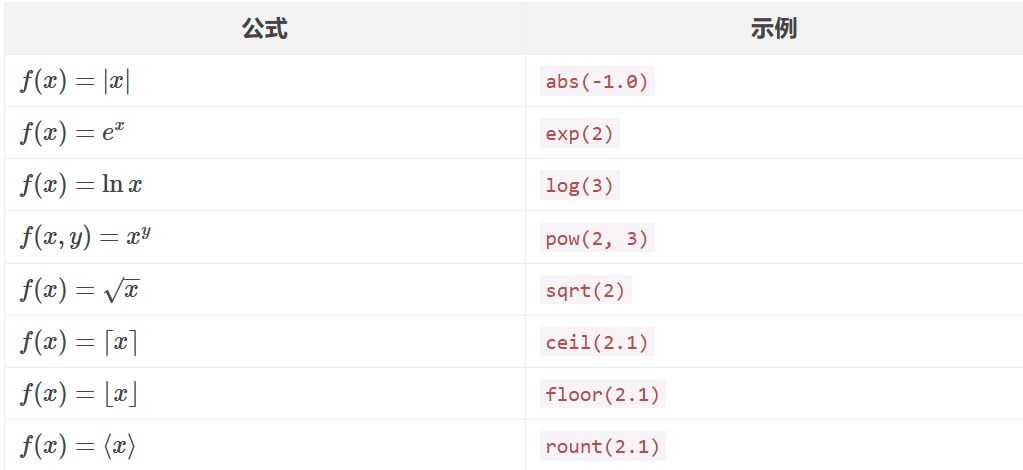
vector<int> arr{1, 2, 1, 4, 5, 4, 4};

sort(arr.begin(), arr.end());//先排序

arr.erase(unique(arr.begin(), arr.end()), arr.end());

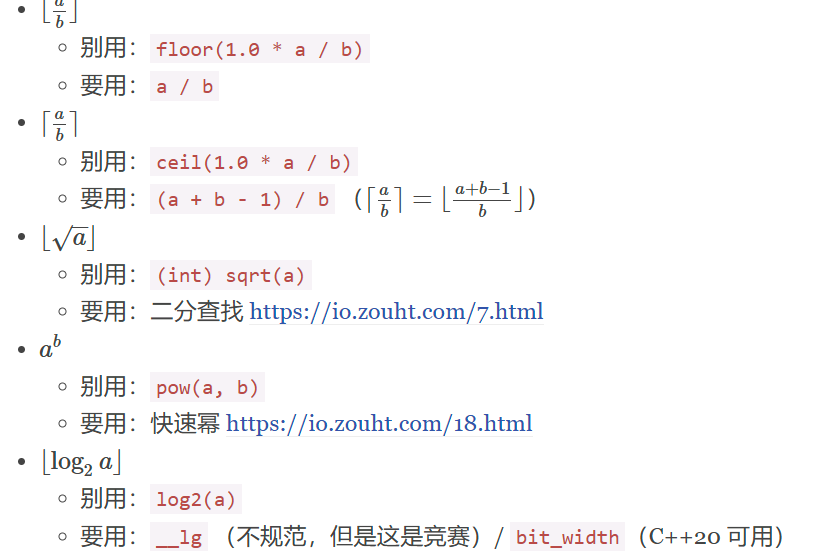
## **2.7** Math

所有函数参数均支持 int / long long / float / double / long double



****注意事项****

由于浮点误差，有些的数学函数的行为可能与预期不符，导致 WA。如果你的操作数都是整型，那么用下面的写法会更稳妥。



## **2.8** gcd() / lcm()

（C++17）返回最大公因数 / 最小公倍数

int x = gcd(8, 12); // 4

int y = lcm(8, 12); // 24

如果不是 C++17，但是是 GNU 编译器（g++），那么可以用内置函数 \_\_gcd().

当然，gcd / lcm 函数也挺好写，直接写也行（欧几里得算法）：

int gcd(int a, int b){

if (!b)

return a;

return gcd(b, a % b);

}

int lcm(int a, int b){

return a / gcd(a, b) \* b;

}

# 3 算法模板 Template

## 3.1 埃式筛 埃拉托斯特尼筛法 Eratosthenes

时间复杂度：O(nloglogn)

const int MAXN = 1e5 + 10;bool is\_prime[MAXN]; // is\_prime储存该数是否为素数

void init\_prime(){

memset(is\_prime, true, sizeof(is\_prime));

is\_prime[0] = is\_prime[1] = false; // 特判0、1不是素数

for (int i = 2; i < MAXN; i++) // 使用埃氏筛处理>1的情况

if (is\_prime[i])

for (int j = 2; j \* i < MAXN; j++)

is\_prime[i \* j] = false;

}

## 3.2 欧拉筛 线性筛 Euler

时间复杂度：O(n)

// prime储存素数序列，primesize即素数数量，not\_prime储存该数是否不是素数const int MAXN = 1e5 + 10;int prime[MAXN], primesize = 0;bool not\_prime[MAXN];

void init\_prime(){

not\_prime[0] = not\_prime[1] = true; // 特判0、1不是素数

for (int i = 2; i < MAXN; i++) // 使用欧拉筛处理>1的情况

{

if (!not\_prime[i])

prime[primesize++] = i;

for (int j = 0; j < primesize && i \* prime[j] < MAXN; j++)

{

not\_prime[i \* prime[j]] = true;

if (i % prime[j] == 0)

break;

}

}

}

## 3.3 二分查找 Binary Search

时间复杂度：O(logn)

### 3.3.1 ≥x

// a[ ]为储存数据的有序递增数组// l ~ r为二分查找的数组范围int l = 0, r = n - 1;while (l < r)

{

int mid = (l + r) / 2;

if (a[mid] >= x)

r = mid;

else

l = mid + 1;

}

### 3.3.2 ≤x≤

// a[ ]为储存数据的有序递增数组// l ~ r为二分查找的数组范围int l = 0, r = n - 1;while (l < r)

{

int mid = (l + r + 1) / 2;

if (a[mid] <= x)

l = mid;

else

r = mid - 1;

}

### 3.3.3 实数

// check(int x): 判断x是否满足条件// eps: 精度（因为浮点数误差，不可直接比大小）// BEGIN: 查找左边界// END: 查找右边界bool check(int x);double eps = 1e-6;double l = BEGIN, r = END;while (r - l > eps)

{

double mid = (l + r) / 2;

if (check(mid))

l = mid;

else

r = mid;

}

## 3.4 三分查找

时间复杂度：O(logn)

// f(double x): 给定区间内的凹函数或凸函数// eps: 精度（因为浮点数误差，不可直接比大小）// BEGIN: 查找左边界// END: 查找右边界

double f(double x);double eps = 1e-6;

double l = BEGIN, r = END;

while(r - l > eps)

{

double m1 = l + (r - l) / 3, m2 = r - (r - l) / 3;

if(f(m1) > f(m2)) // 此为找最小值，若要找最大值，则改为<

l = m1;

else

r = m2;

}

## 3.5 深度优先搜索 DFS

type dfs(type x, ... ) // 可以存在多个变量{

if( ... ) // 达成目标，找到答案

{

... // 输出答案或判断最优解等等

return;

}

if( ... ) // 达到搜索边界（即到边界了还没搜到，有时没有此步骤）

{

return;

}

for( ... ) // 遍历所有子节点

{

if( ... ) // 可以转移状态，一般用标志变量判断

{

... // 修改标志变量，表明此节点不可转移

dfs( ... ) // 搜索子节点，经常为x+1

... // 还原标志变量，表面此节点可转移

}

}

}

## 3.6 广度优先搜索 BFS

bool vis[MAXN]; // 标记是否搜索过，有时也可直接用depth来判断

int depth[MAXN]; // 储存搜索深度，有时可能为二维数组或map

queue<type> que; // STL队列，不过数组模拟队列效率更高

type bfs(type start){

que.push(start); // 起点入队

depth[start] = 0; // 起点深度0

vis[start] = true; // 标记起点

while (!que.empty())

{

type now = que.front(); // 当前节点设置为队首

que.pop(); // 弹出队首

if ( ... ) // 如果达到目标条件

{

ans = depth[now]; // 储存答案

return; // 搜索结束

}

for( ... ) // 遍历now节点的所有子节点，可用数组表示方向

{

type next = ... // 计算出子节点

if (!vis[next] && ... ) // 如果子节点未搜索过，且范围符合题目条件

{

vis[next] = true; // 标记子节点

depth[next] = depth[now] + 1; // 子节点深度+1

que.push(next); // 子节点入队

// 有时题目还需输出具体路径，可用一个数组储存每个节点的上一个节点，然后在此处对数组赋值。输出时，从结尾递归反向输出即可获得具体的路径。

}

}

}

}

## 3.7 辗转相除法 欧几里得算法 Euclidean algorithm

时间复杂度：O(log(a+b))(log⁡(+))

int gcd(int a, int b){

if (!b)

return a;

return gcd(b, a % b);

}

## 3.8 快速幂 Exponentiation by squaring

时间复杂度：O(logn)

### 3.8.1 不取模

ll fast\_pow(ll a, ll b){

ll ans = 1;

while (b)

{

if (b % 2 == 1)

ans = ans \* a;

a = a \* a;

b >>= 1;

}

return ans;

}

### 3.8.2 取模

const ll MOD = 20220128;ll fast\_pow(ll a, ll b){

a %= MOD; // 开头先取一次模

ll ans = 1;

while (b)

{

if (b % 2 == 1)

ans = ans \* a % MOD; // 每次运算都取模

a = a \* a % MOD; // 每次运算都取模

b >>= 1;

}

return ans;

}

## 3.9 KMP 算法 The Knuth-Morris-Pratt Algorithm

时间复杂度：O(n+m)

### 3.9.1 类封装

使用时先构造 KMP，传入参数为****模式串（Pattern）****.

匹配时调用 .find()，传入参数为****主串（Text）****.

class KMP

{

vector<int> nxt;

string pat;

public:

KMP(string &s)

{

pat = s;

int n = pat.length();

int j = 0;

nxt.resize(n);

for (int i = 1; i < n; i++)

{

while (j > 0 && pat[i] != pat[j])

j = nxt[j - 1];

if (pat[i] == pat[j])

j++;

nxt[i] = j;

}

}

vector<int> find(string &txt)

{

int n = pat.length(), m = txt.length();

int j = 0;

vector<int> ans;

for (int i = 0; i < m; i++)

{

while (j > 0 && txt[i] != pat[j])

j = nxt[j - 1];

if (txt[i] == pat[j])

j++;

if (j == n)

{

ans.push\_back(i - n + 1);

j = nxt[j - 1];

}

}

return ans;

}

set<int> get\_border()

{

set<int> s;

int cur = nxt.back();

while (cur)

{

s.insert(cur);

cur = nxt[cur - 1];

}

s.insert(0);

return s;

}

};

### 3.9.1 旧模板

****计算部分匹配表****

char s1[MAXN]; // 主串char s2[MAXN]; // 模式串int nxt[MAXN]; // 部分匹配表void getnext(void){

nxt[0] = -1;

int i = 0, j = -1;

int lens2 = strlen(s2);

while (i < lens2)

{

if (j == -1 || s2[i] == s2[j])

{

++i;

++j;

nxt[i] = j;

}

else

{

j = nxt[j];

}

}

}

****找所有匹配****

char s1[MAXN]; // 主串char s2[MAXN]; // 模式串int nxt[MAXN]; // 部分匹配表void kmp(void){

int i = 0, j = 0;

int lens1 = strlen(s1), lens2 = strlen(s2);

while (i < lens1)

{

if (j == -1 || s1[i] == s2[j])

{

i++;

j++;

}

else

{

j = nxt[j];

}

if (j == lens2)

{

printf("%d\n", i - j + 1);

}

}

}

****找第一个匹配****

char s1[MAXN]; // 主串char s2[MAXN]; // 模式串int nxt[MAXN]; // 部分匹配表void kmp(void){

int i = 0, j = 0;

int lens1 = strlen(s1), lens2 = strlen(s2);

while (i < lens1 && j < lens2)

{

if (j == -1 || s1[i] == s2[j])

{

i++;

j++;

}

else

{

j = nxt[j];

}

}

if (j == lens2)

{

printf("%d\n", i - j + 1);

}

else

{

printf("-1\n"); // -1表示没匹配到结果

}

}

## 3.10 Dijkstra 算法

解决赋权图的单源最短路径问题，****不能解决负边****。

### 3.10.1 朴素（适合稠密图）

时间复杂度：O(|V|2)

const int MAXN = 510, INF = 0x3f3f3f3f; // INF代表无穷大int g[MAXN][MAXN]; // 邻接矩阵存图int dist[MAXN]; // 最短距离bool vis[MAXN]; // 访问情况int v, e; // v顶点数 e边数

void dijkstra(){

memset(dist, 0x3f, sizeof(dist));

dist[1] = 0;

for (int i = 1; i <= v; i++)

{

int t = -1;

for (int j = 1; j <= v; j++)

if (!vis[j] && (t == -1 || dist[t] > dist[j]))

t = j;

vis[t] = true;

for (int j = 1; j <= v; j++)

dist[j] = min(dist[j], dist[t] + g[t][j]);

}

}

### 3.10.2 堆优化（适合稀疏图）

时间复杂度：O((|E|+|V|)log|V|)

#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> PII;const int MAXN = 100010, INF = 0x3f3f3f3f; // INF代表无穷大int E, V, S;

// E边数，V顶点数，S起点

vector<PII> edge[MAXN]; // 储存连接关系，二元组为(权值,终点)

priority\_queue<PII, vector<PII>, greater<PII>> pque; // 储存节点，节点距离小的在堆顶

int dist[MAXN]; // 储存节点距离bool vis[MAXN];

// 是否已经访问过该节点的标志

void dijkstra(){

memset(dist, 0x3f, sizeof(dist));

dist[1] = 0;

pque.push({dist[1], 1});

while (!pque.empty())

{

PII cur = pque.top();

pque.pop();

if (vis[cur.second])

continue;

vis[cur.second] = true;

for (auto next : edge[cur.second])

{

if (dist[next.second] > dist[cur.second] + next.first)

{

dist[next.second] = dist[cur.second] + next.first;

if (!vis[next.second])

pque.push({dist[next.second], next.second});

}

}

}

}

## 3.11 Floyd-Warshall 算法

解决赋权图的多源最短路径问题，能解决负边，不能解决负环。

时间复杂度：O(|V|3)

const int MAXN = 1010, INF = 0x3f3f3f3f;int e, v; // e边数 v顶点数

int dist[MAXN][MAXN]; // dist[x][y]代表x到y的距离

void floyd\_init(void){

memset(dist, 0x3f, sizeof(dist));

for (int i = 1; i <= v; i++)

dist[i][i] = 0;

}

void floyd(void){

for (int k = 1; k <= v; k++)

for (int i = 1; i <= v; i++)

for (int j = 1; j <= v; j++)

if (dist[i][k] < INF && dist[k][j] < INF)

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);

}

## 3.12 最短路径快速算法 SPFA

平均时间复杂度：O(|E|)

最差时间复杂度：O(|V|⋅|E|)

### 3.12.1 最短路

const int MAXN = 1e5 + 10, INF = 0x3f3f3f3f; // INF代表无穷大

int e, v, s; // e边数 v顶点数 s起点

vector<pair<int, int>> edge[MAXN]; // 邻接表存图 pair为(距离, 终点)

queue<int> que; // STL队列

int dist[MAXN]; // 最短距离

bool vis[MAXN]; // 是否入队

void spfa(int src){

memset(dist, 0x3f, sizeof(dist));

dist[src] = 0;

que.push(src);

vis[src] = true;

while (!que.empty())

{

int cur = que.front();

que.pop();

vis[cur] = false;

for (auto next : edge[cur])

{

if (dist[next.second] > dist[cur] + next.first)

{

dist[next.second] = dist[cur] + next.first;

if (!vis[next.second])

{

que.push(next.second);

vis[next.second] = true;

}

}

}

}

}

### 3.12.2 判负权回路

const int MAXN = 1e5 + 10, INF = 0x3f3f3f3f;

int E, V, S;

vector<pair<int, int>> edge[MAXN];

queue<int> que;int dist[MAXN], cnt[MAXN];bool vis[MAXN];

bool spfa(){

for (int i = 1; i <= V; i++)

{

que.push(i);

vis[i] = true;

}

while (!que.empty())

{

int cur = que.front();

que.pop();

vis[cur] = false;

for (auto next : edge[cur])

{

if (dist[next.second] > dist[cur] + next.first)

{

dist[next.second] = dist[cur] + next.first;

cnt[next.second] = cnt[cur] + 1;

if (cnt[next.second] >= V)

return true;

if (!vis[next.second])

{

que.push(next.second);

vis[next.second] = true;

}

}

}

}

return false;

}

## 3.13 克鲁斯卡尔算法 Kruskal

时间复杂度：O(|E|log|V|)

适合稀疏图

const int MAXN = 1e5 + 10, INF = 0x3f3f3f3f;int v, e; // v顶点数 e边数int fa[MAXN]; // 并查集struct Edge // 边的结构体，重载了<供sort()

{

int start, end, dist;

bool operator<(const Edge &x) const

{

return dist < x.dist;

}

};

vector<Edge> edges; // 邻接表存边

inline void init(int n){

for (int i = 1; i <= n; i++)

fa[i] = i;

}

int find(int x){

return x == fa[x] ? x : (fa[x] = find(fa[x]));

}

inline void merge(int i, int j){

fa[find(i)] = find(j);

}

int kruskal(){

sort(edges.begin(), edges.end());

int ans = 0, selected = 0;

bool flag = false;

for (auto ed : edges)

{

if (find(ed.start) != find(ed.end))

{

merge(ed.start, ed.end);

ans += ed.dist;

if (++selected == v - 1)

{

flag = true;

break;

}

}

}

return flag ? ans : INF;

}

## 3.14 普林姆算法 Prim

### 3.14.1 朴素（适合稠密图）

时间复杂度：O(|V|2)(||2)

const int MAXN = 510, INF = 0x3f3f3f3f; // INF代表无穷大int g[MAXN][MAXN]; // 邻接矩阵存图int dist[MAXN]; // 距离已选择点的最短距离int vis[MAXN]; // 点是否选择int v, e; // v顶点数 e边数

int prim(){

int ans = 0;

memset(dist, 0x3f, sizeof(dist));

dist[1] = 0;

for (int i = 0; i < v; i++)

{

int t = -1;

for (int j = 1; j <= v; j++)

if (!vis[j] && (t == -1 || dist[t] > dist[j]))

t = j;

if (dist[t] == INF)

return INF;

ans += dist[t];

vis[t] = true;

for (int j = 1; j <= v; j++)

dist[j] = min(dist[j], g[t][j]);

}

return ans;

}

### 3.14.2 堆优化

时间复杂度：O((|E|+|V|)log|V|)((||+||)log⁡||)

typedef pair<int, int> PII;

const int MAXN = 100010, INF = 0x3f3f3f3f; // INF代表无穷大

int E, V; // E边数 V顶点数

vector<PII> edge[MAXN]; // 储存边，二元组为(权值,终点)

priority\_queue<PII, vector<PII>, greater<PII>> pque; // 短的边在堆顶

int dist[MAXN]; // 储存节点到已选择点的最短距离

bool vis[MAXN]; // 是否已选择该点

int prim(){

int ans = 0, cnt = 0;

memset(dist, 0x3f, sizeof(dist));

dist[1] = 0;

pque.push({dist[1], 1});

while (!pque.empty())

{

PII cur = pque.top();

pque.pop();

if (vis[cur.second])

continue;

ans += cur.first;

cnt++;

vis[cur.second] = true;

for (auto next : edge[cur.second])

{

if (dist[next.second] > next.first)

{

dist[next.second] = next.first;

if (!vis[next.second])

pque.push({dist[next.second], next.second});

}

}

}

return cnt == V ? ans : INF;

}

## 3.15 排序算法 Sort

### 2.16.2 最大匹配 匈牙利算法 Hungarian Algorithm

时间复杂度：O(|V|⋅|E|)(||⋅||)

#include <bits/stdc++.h>

using namespace std;

const int SIZE = 1010;

int n1, n2, m;// n1为二分图的一个子图的顶点数，n2为另一个子图的顶点数，m为边数。

vector<int> edge[SIZE]; // 这里使用vector存图

int match[SIZE]; // 储存n2中的某个顶点匹配的n1中的顶点

bool vis[SIZE]; // 标记n2中的某个顶点是否已经匹配过

bool find(int x){

for (auto i : edge[x]) // 遍历所有与x连接的n2内的顶点i

{

if (!vis[i]) // 如果顶点i本轮还未匹配过

{

vis[i] = true; // 将其标记

// 若顶点i还没有匹配到任何n1中顶点，则直接把i与x匹配

// 如果i已经匹配上，则查询与i匹配的n1中的元素能否换一个匹配，若可以，则将i与x匹配

if (match[i] == 0 || find(match[i]))

{

match[i] = x;

return true;

}

}

}

return false; // 如果没法匹配上，返回false

}

int main(void){

cin >> n1 >> n2 >> m;

for (int i = 0; i < m; i++)

{

int u, v;

cin >> u >> v;

edge[u].push\_back(v); // 只用得到从n1到n2的边，因此只存了单向边

}

int cnt = 0; // 匹配的数量

for (int i = 1; i <= n1; i++) // 从n1第一个元素开始，尝试匹配到最后一个元素

{

memset(vis, false, sizeof(vis)); // 先清空所有n2的访问情况

if (find(i)) // 如果匹配上则匹配数+1

cnt++;

}

cout << cnt << endl;

return 0;

}

## 3.17 背包模型

### 3.17.1 0/1 背包

时间复杂度：O(N⋅V)

空间复杂度：O(N⋅V)

const int SIZE = 1010;

int N, V; // N-物品数量 V-背包容积

int v[SIZE], w[SIZE]; // v-体积 w-价值

int dp[SIZE][SIZE]; // 二维动态规划

void solve(){

cin >> N >> V;

for (int i = 1; i <= N; i++)

cin >> v[i] >> w[i];

for (int i = 1; i <= N; i++)

{

for (int j = 0; j <= V; j++)

{

dp[i][j] = dp[i - 1][j];

if (j >= v[i])

dp[i][j] = max(dp[i][j], dp[i - 1][j - v[i]] + w[i]);

}

}

cout << dp[N][V] << endl;

}

时间复杂度：O(N⋅V)

空间复杂度：O(V)()

const int SIZE = 1010;

int N, V; // N-物品数量 V-背包容积

int v[SIZE], w[SIZE]; // v-体积 w-价值

int dp[SIZE]; // 一维动态规划

void solve(){

cin >> N >> V;

for (int i = 1; i <= N; i++)

cin >> v[i] >> w[i];

for (int i = 1; i <= N; i++)

for (int j = V; j >= v[i]; j--)

dp[j] = max(dp[j], dp[j - v[i]] + w[i]);

cout << dp[V] << endl;

}

### 3.17.2 完全背包

时间复杂度：O(N⋅V2)(最坏情况)

空间复杂度：O(N⋅V)

const int SIZE = 1010;

int N, V;

int v[SIZE], w[SIZE];

int dp[SIZE][SIZE];

void solve(){

cin >> N >> V;

for (int i = 1; i <= N; i++)

cin >> v[i] >> w[i];

for (int i = 1; i <= N; i++)

for (int j = 0; j <= V; j++)

for (int k = 0; k \* v[i] <= j; k++)

dp[i][j] = max(dp[i][j], dp[i - 1][j - k \* v[i]] + k \* w[i]);

cout << dp[N][V] << endl;

}

时间复杂度：O(N⋅V)(⋅)

空间复杂度：O(N⋅V)(⋅)

const int SIZE = 1010;int N, V;int v[SIZE], w[SIZE];int dp[SIZE][SIZE];

void solve(){

cin >> N >> V;

for (int i = 1; i <= N; i++)

cin >> v[i] >> w[i];

for (int i = 1; i <= N; i++)

{

for (int j = 0; j <= V; j++)

{

dp[i][j] = dp[i - 1][j];

if (j >= v[i])

dp[i][j] = max(dp[i][j], dp[i][j - v[i]] + w[i]);

}

}

cout << dp[N][V] << endl;

}

时间复杂度：O(N⋅V)(⋅)

空间复杂度：O(V)()

const int SIZE = 1010;int N, V;int v[SIZE], w[SIZE];int dp[SIZE];

void solve(){

cin >> N >> V;

for (int i = 1; i <= N; i++)

cin >> v[i] >> w[i];

for (int i = 1; i <= N; i++)

for (int j = v[i]; j <= V; j++)

dp[j] = max(dp[j], dp[j - v[i]] + w[i]);

cout << dp[V] << endl;

}

### 3.17.3 多重背包

时间复杂度：O(N⋅V2)(⋅2) (最坏情况)

空间复杂度：O(V)()

const int SIZE = 1010;int N, V;int v[SIZE], w[SIZE], s[SIZE];int dp[SIZE];

void solve(){

cin >> N >> V;

for (int i = 1; i <= N; i++)

cin >> v[i] >> w[i] >> s[i];

for (int i = 1; i <= N; i++)

for (int j = V; j >= v[i]; j--)

for (int k = 0; k \* v[i] <= j && k <= s[i]; k++)

dp[j] = max(dp[j], dp[j - k \* v[i]] + k \* w[i]);

cout << dp[V] << endl;

}

时间复杂度：O(N⋅logS⋅V)(⋅log⁡⋅)

空间复杂度：O(V)() (已包含空间优化)

const int SIZE\_NlogS = 25000, SIZE\_V = 2010; // 需要注意各个数组的大小int N, V, idx;int v[SIZE\_NlogS], w[SIZE\_NlogS];int dp[SIZE\_V];

void solve(){

cin >> N >> V; // 物品读入

for (int i = 1; i <= N; i++)

{

int a, b, s;

cin >> a >> b >> s;

int k = 1;

while (k <= s) // 物品打包

{

v[++idx] = a \* k;

w[idx] = b \* k;

s -= k;

k \*= 2;

}

if (s)

{

v[++idx] = a \* s;

w[idx] = b \* s;

}

}

// 01-背包问题

for (int i = 1; i <= idx; i++)

for (int j = V; j >= v[i]; j--)

dp[j] = max(dp[j], dp[j - v[i]] + w[i]);

cout << dp[V] << endl;

}

时间复杂度：O(N⋅V)(⋅)

空间复杂度：O(V)() (已包含空间优化)

void solve(){

int N, V;

cin >> N >> V;

vector<int> v(N + 10), w(N + 10), s(N + 10);

for (int i = 1; i <= N; i++)

cin >> v[i] >> w[i] >> s[i];

vector<int> dp(V + 10), lst(V + 10), que(V + 10);

for (int i = 1; i <= N; i++)

{

lst = dp;

for (int r = 0; r < v[i]; r++)

{

int hh = 0, tt = -1;

for (int k = r; k <= V; k += v[i])

{

if (hh <= tt && que[hh] < k - s[i] \* v[i])

++hh;

while (hh <= tt && lst[que[tt]] - (que[tt] - r) / v[i] \* w[i] <= lst[k] - (k - r) / v[i] \* w[i])

--tt;

if (hh <= tt)

dp[k] = max(dp[k], lst[que[hh]] - (que[hh] - k) / v[i] \* w[i]);

que[++tt] = k;

}

}

}

cout << dp[V] << endl;

}

### 3.17.4 分组背包

时间复杂度：O(V⋅∑S)(⋅∑)

空间复杂度：O(V)() (已包含空间优化)

const int SIZE = 110;int N, V;int v[SIZE][SIZE], w[SIZE][SIZE], S[SIZE];int dp[SIZE];

void solve(){

cin >> N >> V;

for (int i = 1; i <= N; i++)

{

cin >> S[i];

for (int j = 0; j < S[i]; j++)

cin >> v[i][j] >> w[i][j];

}

for (int i = 0; i <= N; i++)

for (int j = V; j >= 0; j--)

for (int k = 0; k < S[i]; k++)

if (v[i][k] <= j)

dp[j] = max(dp[j], dp[j - v[i][k]] + w[i][k]);

cout << dp[V] << endl;

}

### 3.17.5 混合背包

void solve(){

int N, V;

cin >> N >> V;

vector<int> dp(V + 10);

for (int i = 1; i <= N; i++)

{

int v, w, s;

cin >> v >> w >> s;

if (s == 0) // 完全背包

{

for (int j = v; j <= V; j++)

dp[j] = max(dp[j], dp[j - v] + w);

}

else // 0-1背包 和 多重背包

{

s = abs(s); // 一律当多重背包转化成0-1背包

vector<int> vv, ww;

int k = 1;

while (k <= s)

{

vv.push\_back(v \* k);

ww.push\_back(w \* k);

s -= k;

k \*= 2;

}

if (s)

{

vv.push\_back(v \* s);

ww.push\_back(w \* s);

}

for (int j = 0; j < vv.size(); j++)

for (int k = V; k >= vv[j]; k--)

dp[k] = max(dp[k], dp[k - vv[j]] + ww[j]);

}

}

cout << dp[V] << endl;

}

## 3.18 高精度 Big Integer

### 3.18.1 I/O

/\* 变量 \*/

string a;

vector<int> A;

/\* 输入 \*/

cin >> a; // 首先以字符串形式读入for (int i = a.size() - 1; i >= 0; i--)

A.push\_back(a[i] - '0'); // 反向将字符串每位写入整型数组，注意减去偏移量‘0’

/\* 输出 \*/for (int i = A.size() - 1; i >= 0; i--)

cout << A[i]; // 反向输出整型数组每一位

### 3.18.2 BI + BI

vector<int> add(vector<int> &A, vector<int> &B){

if (B.size() > A.size())

return add(B, A);

vector<int> C;

int t = 0;

for (int i = 0; i < A.size(); i++)

{

t += A[i];

if (i < B.size())

t += B[i];

C.push\_back(t % 10);

t = t > 9 ? 1 : 0;

}

if (t)

C.push\_back(1);

return C;

}

### 3.18.3 BI - BI

vector<int> sub(vector<int> &A, vector<int> &B){

vector<int> C;

int t = 0;

for (int i = 0; i < A.size(); i++)

{

t = A[i] - t;

if (i < B.size())

t -= B[i];

C.push\_back((t + 10) % 10);

t = t < 0 ? 1 : 0;

}

while (C.size() > 1 && C.back() == 0)

C.pop\_back();

return C;

}

bool cmp(vector<int> &A, vector<int> &B) // 若A>=B返回true，否则返回false{

if (A.size() != B.size())

return A.size() > B.size();

for (int i = A.size() - 1; i >= 0; i--)

if (A[i] != B[i])

return A[i] > B[i];

return true;

}

int main(){

if (cmp(A, B))

vector<int> C = sub(A, B);

else

vector<int> C = sub(B, A); // 这种情况输出时记得先输出一个‘-’符号

}

### 3.18.4 BI \* I

vector<int> mul(vector<int> &A, int b){

vector<int> C;

int t = 0;

for (int i = 0; i < A.size() || t; i++)

{

if (i < A.size())

t += A[i] \* b;

C.push\_back(t % 10);

t /= 10;

}

while (C.size() > 1 && C.back() == 0)

C.pop\_back();

return C;

}

### 3.18.5 BI / I

vector<int> div(vector<int> &A, int b, int &r){

vector<int> C;

r = 0;

for (int i = A.size() - 1; i >= 0; i--)

{

r = r \* 10 + A[i];

C.push\_back(r / b);

r %= b;

}

reverse(C.begin(), C.end());

while (C.size() > 1 && C.back() == 0)

C.pop\_back();

return C;

}

## 3.19 贝尔曼-福特算法 Bellman-Ford

解决赋权图的单源最短路径问题，能解决负边，能解决负环。

时间复杂度：O(|V|⋅|E|)(||⋅||)

const int MAXN = 510, MAXM = 1e4 + 10, INF = 0x3f3f3f3f;int v, e, k; // v顶点数 e边数 k经过边数限制int dist[MAXN], backup[MAXN]; // dist最短距离 backup备份上一次迭代struct Edge

{

int a, b, w; // a起点 b终点 w权值

} edges[MAXM]; // 结构图数组存边

int bellman\_ford(){

memset(dist, 0x3f, sizeof(dist));

dist[1] = 0;

for (int i = 0; i < k; i++)

{

memcpy(backup, dist, sizeof(dist));

for (int j = 0; j < e; j++)

{

int a = edges[j].a, b = edges[j].b, w = edges[j].w;

dist[b] = min(dist[b], backup[a] + w);

}

}

return dist[v] > INF / 2 ? INF : dist[v];

}

## 3.20 矩阵加速算法

### 3.20.1 n 阶方阵乘法

vector<vector<ll>> mat\_mul(vector<vector<ll>> a, vector<vector<ll>> b, ll mod)

{

int n = a.size();

vector<vector<ll>> res(n, vector<ll>(n));

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

for (int k = 0; k < n; k++)

res[i][j] = (res[i][j] + a[i][k] \* b[k][j] % mod) % mod;

return res;

}

### 3.20.2 矩阵快速幂

vector<vector<ll>> mat\_pow(vector<vector<ll>> a, ll b, ll mod)

{

int n = a.size();

vector<vector<ll>> res(n, vector<ll>(n));

for (int i = 0; i < n; i++)

res[i][i] = 1;

while (b)

{

if (b % 2)

res = mat\_mul(res, a, mod);

a = mat\_mul(a, a, mod);

b /= 2;

}

return res;

}

## 3.21 最长上升子序列 Longest Increasing Subsequence

### 3.21.1 动态规划

时间复杂度：O(n2)

const int MAXN = 1010;int N, a[MAXN];int dp[MAXN];

void solve(){

a[0] = INT32\_MIN;

cin >> N;

for (int i = 1; i <= N; i++)

cin >> a[i];

for (int i = 1; i <= N; i++)

for (int j = 0; j < i; j++)

if (a[j] < a[i])

dp[i] = max(dp[i], dp[j] + 1);

int ans = 0;

for (int i = 1; i <= N; i++)

ans = max(ans, dp[i]);

cout << ans << endl;

}

### 3.21.2 贪心、二分、单调栈

时间复杂度：O(nlogn)

const int MAXN = 1e5 + 10;int N, a[MAXN];

vector<int> v;

void solve(){

cin >> N;

for (int i = 1; i <= N; i++)

cin >> a[i];

v.push\_back(a[1]);

for (int i = 2; i <= N; i++)

{

if (a[i] > v.back())

v.push\_back(a[i]);

else

v[lower\_bound(v.begin(), v.end(), a[i]) - v.begin()] = a[i];

}

cout << v.size() << endl;

}

## 3.22 最长公共上升子序列 Longest Common Increasing Subsequence

### 3.22.1 三重循环 DP

时间复杂度：O(n3)(3)

void solve(){

int N;

cin >> N;

vector<int> A(N + 10), B(N + 10);

vector<vector<int>> dp(N + 10, vector<int>(N + 10, 0));

for (int i = 1; i <= N; i++)

cin >> A[i];

for (int i = 1; i <= N; i++)

cin >> B[i];

for (int i = 1; i <= N; i++)

{

for (int j = 1; j <= N; j++)

{

dp[i][j] = max(dp[i][j], dp[i - 1][j]);

if (A[i] == B[j])

for (int k = 0; k < j; k++)

if (B[k] < B[j])

dp[i][j] = max(dp[i][j], dp[i - 1][k] + 1);

}

}

int ans = 0;

for (int i = 1; i <= N; i++)

ans = max(dp[N][i], ans);

cout << ans << endl;

}

### 3.22.2 将 DP 进行简化

时间复杂度：O(n2)(2)

void solve(){

int N;

cin >> N;

vector<int> A(N + 10), B(N + 10);

vector<vector<int>> dp(N + 10, vector<int>(N + 10, 0));

for (int i = 1; i <= N; i++)

cin >> A[i];

for (int i = 1; i <= N; i++)

cin >> B[i];

for (int i = 1; i <= N; i++)

{

int maxv = 1;

for (int j = 1; j <= N; j++)

{

dp[i][j] = dp[i - 1][j];

if (A[i] == B[j])

dp[i][j] = max(dp[i][j], maxv);

if (B[j] < A[i])

maxv = max(maxv, dp[i - 1][j] + 1);

}

}

int ans = 0;

for (int i = 1; i <= N; i++)

ans = max(dp[N][i], ans);

cout << ans << endl;

}

## 3.23 最近公共祖先 Lowest Common Ancestor

### 3.23.1 倍增法

预处理：O(Vlogd)，其中 d 为树的深度。

计算：O(logd)，其中 d 为树的深度。

constexpr int MAXN = 1e6 + 10;

int h[MAXN], e[MAXN], ne[MAXN], idx;

int mylog2[MAXN]; // \lfloor log\_{2}{x} \rfloor + 1

int fa[MAXN][30], dep[MAXN];

void init(){

memset(h, -1, sizeof(h));

for (int i = 1; i < MAXN; i++)

mylog2[i] = mylog2[i - 1] + (1 << mylog2[i - 1] == i);

}

void dfs(int now, int father){

fa[now][0] = father;

dep[now] = dep[father] + 1;

for (int i = 0; i < mylog2[dep[now]]; i++)

fa[now][i + 1] = fa[fa[now][i]][i];

for (int i = h[now]; i != -1; i = ne[i])

if (e[i] != father)

dfs(e[i], now);

}

int lca(int a, int b){

if (dep[a] < dep[b])

swap(a, b); // make a >= b

while (dep[a] > dep[b])

a = fa[a][mylog2[dep[a] - dep[b]] - 1];

if (a == b)

return a;

for (int i = mylog2[dep[a]] - 1; i >= 0; i--)

{

if (fa[a][i] != fa[b][i])

{

a = fa[a][i];

b = fa[b][i];

}

}

return fa[a][0];

}

### 3.23.2 树链剖分

预处理：O(n)()

计算：O(logn)(log⁡)

int lca(int u, int v) {

while (top[u] != top[v])

{

if (dep[top[u]] > dep[top[v]])

u = fa[top[u]];

else

v = fa[top[v]];

}

return dep[u] > dep[v] ? v : u;

}

## 3.24 树链剖分（重链剖分）

链式前向星存图，下标从 11 开始。

### 3.24.1 剖分

void dfs1(int now){

son[now] = -1;

siz[now] = 1;

for (int i = h[now]; i; i = ne[i])

{

int &nxt = e[i];

if (nxt == fa[now])

continue;

fa[nxt] = now;

dep[nxt] = dep[now] + 1;

/\* 如果是边权 \*/

// val[nxt] = w[i];

dfs1(nxt);

siz[now] += siz[nxt];

if (son[now] == -1 || siz[son[now]] < siz[nxt])

son[now] = nxt;

}

}

void dfs2(int now, int tp){

top[now] = tp;

dfn[now] = ++cnt;

rnk[cnt] = now;

if (son[now] == -1)

return;

dfs2(son[now], tp);

for (int i = h[now]; i; i = ne[i])

{

int &nxt = e[i];

if (nxt == son[now] || nxt == fa[now])

continue;

dfs2(nxt, nxt);

}

}

### 3.24.2 操作

int do\_something(int a, int b){

while (top[a] != top[b])

{

int ta = top[a], tb = top[b];

if (dep[ta] >= dep[tb])

{

// do something in range [dfn[ta], dfn[a]]

a = fa[ta];

}

else

{

// do something in range [dfn[tb], dfn[b]]

b = fa[tb];

}

}

if (dep[a] > dep[b])

swap(a, b);

// do something in range [dfn[a], dfn[b]] 如果是点权

// do something in range [dfn[a] + 1, dfn[b]] 如果是边权

return ans;

}

## 3.25 字符串哈希

### 3.25.1 哈希

* p=131,13331,233,449=131,13331,233,449
* m=109+7,998244353,998244853,436522843,264=109+7,998244353,998244853,436522843,264

typedef long long ll;constexpr ll P = 449, MOD = 436522843;

ll get\_hash(string &s){

ll res = 0;

for (int i = 0; i < s.size(); i++)

res = (res \* P % MOD + s[i]) % MOD;

return res;

}

### 3.25.2 子串哈希

预处理：O(n)()

计算：O(1)(1)

struct StrHash

{

int len, base, mod;

vector<int> p, h;

void init(const string &s, int base, int mod)

{

len = s.size();

this->base = base;

this->mod = mod;

p.resize(len + 1);

h.resize(len + 1);

p[0] = 1;

for (int i = 1; i <= len; i++)

p[i] = p[i - 1] \* base % mod;

h[0] = s[0] % mod;

for (int i = 1; i < len; i++)

h[i] = (h[i - 1] \* base % mod + s[i]) % mod;

}

int get(int l, int r)

{

if (l <= 0)

return h[r];

return (h[r] - h[l - 1] \* p[r - l + 1] % mod + mod) % mod;

}

};

### 3.25.3 允许 k 次失配的匹配

/\* 依赖上文的StrHash结构体 \*/bool check(StrHash &a, StrHash &b, int toler){

if (a.len < b.len) // make a >= b

return check(b, a, toler);

int la = a.len, lb = b.len;

for (int i = 0; i <= la - lb; i++)

{

int err = 0, pos = 0;

while (err <= toler && pos < lb)

{

int l = pos, r = lb - 1;

while (l < r)

{

int mid = (l + r) / 2;

if (a.get(i + pos, i + mid) == b.get(pos, mid))

l = mid + 1;

else

r = mid;

}

if (a.get(i + pos, i + l) != b.get(pos, l))

err++;

pos = l + 1;

}

if (err <= toler)

return true;

}

return false;

}

## 3.26 Manacher 算法

### 2.26.1 预处理

string pre\_process(string &s){

string ans = "^";

for (auto &c : s)

{

ans += '#';

ans += c;

}

ans += '#';

ans += '$';

return ans;

}

### 3.26.2 马拉车

// s - 字符串（下标1开始，需要预处理）// p - 对应位置回文半径void manacher(string &s, vector<int> &p){

int r = 0, mid = 0;

for (int i = 1; i < s.size() - 1; i++)

{

p[i] = r > i ? min(p[2 \* mid - i], r - i) : 1;

while (s[i + p[i]] == s[i - p[i]])

p[i]++;

if (i + p[i] > r)

{

r = i + p[i];

mid = i;

}

}

}

## 3.27 拓扑排序 Topo Sort

时间复杂度：O(V+E)

constexpr int MAXN = 5050;

int h[MAXN], e[MAXN], d[MAXN], ne[MAXN], idx;

int que[MAXN];

void add(int a, int b){

idx++;

d[b]++;

e[idx] = b;

ne[idx] = h[a];

h[a] = idx;

}

bool topo\_sort(int n) // n - vertice cnt{

int hh = 0, tt = -1;

for (int i = 1; i <= n; i++)

if (!d[i])

que[++tt] = i;

while (hh <= tt)

{

int &now = que[hh++];

for (int i = h[now]; i; i = ne[i])

{

int &nxt = e[i];

d[nxt]--;

if (!d[nxt])

que[++tt] = nxt;

}

}

return tt == n - 1; // false if illegal

}

## 3.28 莫队 Mo's Algorithm

若存在一个长度为 n 的序列，对于序列上的 m 个区间询问问题，如果一个区间答案能够在 O(1)(1) 转移到相邻区间的答案，那么可以通过莫队算法在 O(nm−−√)() 的复杂度求出所有询问。

### 3.28.1 普通莫队

int cur\_ans = 0; // current answerint block; // block size

void add(int pos) { /\* update current answer \*/ }void del(int pos) { /\* update current answer \*/ }

bool cmp(Query a, Query b){

if (a.l / block != b.l / block)

return a.l < b.l;

return (a.l / block) % 2 ? a.r < b.r : a.r > b.r;

}

void solve(){

block = sqrt(m);

sort(query.begin(), query.end(), cmp);

int l = 1, r = 0; // initial

for (int i = 0; i < m; i++)

{

while (l > query[i].l) add(--l);

while (r < query[i].r) add(++r);

while (l < query[i].l) del(l++);

while (r > query[i].r) del(r--);

ans[query[i].idx] = cur\_ans;

}

}

### 3.28.2 带修改莫队

struct Query

{

int idx, l, r, ver;

bool operator<(Query b)

{

if (l / block != b.l / block)

return l < b.l;

else if (r / block != b.r / block)

return r < b.r;

else

return ver < b.ver;

}

};

struct Modif

{

int pos, color;

};

for (int i = 1; i <= now\_idx; i++)

{

while (l > qu[i].l) add(c[--l]);

while (r < qu[i].r) add(c[++r]);

while (l < qu[i].l) del(c[l++]);

while (r > qu[i].r) del(c[r--]);

while (time < qu[i].ver)

{

time++;

if (l <= mo[time].pos && mo[time].pos <= r)

{

add(mo[time].color);

del(c[mo[time].pos]);

}

swap(mo[time].color, c[mo[time].pos]);

}

while (time > qu[i].ver)

{

if (l <= mo[time].pos && mo[time].pos <= r)

{

add(mo[time].color);

del(c[mo[time].pos]);

}

swap(mo[time].color, c[mo[time].pos]);

time--;

}

ans[qu[i].idx] = cur;

}