Contents

[TOC]

参考tianjinsa/cpp-md: cpp 算法的相关资料

1 C++ 基础类型

1.1 变量和常量

1.1.1 变量类型

1.1.1.1 内建类型

分类	类型/别名	典型大小（字节）	常用头文件	说明
布尔	`bool`	1	*(内建)*	取值 `true/false`
字符	`char`	1	*(内建)*	符号性由实现决定（有的为有符号）
字符	`signed char` / `unsigned char`	1	*(内建)*	明确有/无符号
宽字符	`wchar_t`	2(Win)/4(Unix)	`<cwchar>`	宽字符编码实现相关
UTF 字符	`char8_t`(C++20), `char16_t`, `char32_t`	1/2/4	`<cuchar>`	分别用于 UTF-8/16/32 单元
整型	`short` / `unsigned short`	2	*(内建)*	
整型	`int` / `unsigned int`	4	*(内建)*	
整型	`long` / `unsigned long`	8(LP64)/4(LLP64)	*(内建)*	注意平台差异
整型	`long long` / `unsigned long long`	8	*(内建)*	
大小/指针差	`size_t`	8(64 位)	`<cstddef>`	无符号，容器大小
大小/指针差	`ptrdiff_t`	8(64 位)	`<cstddef>`	有符号，指针差值
浮点	`float`	4	*(内建)*	IEEE-754 单精度（常见）
浮点	`double`	8	*(内建)*	IEEE-754 双精度（常见）
浮点	`long double`	8/16（实现依赖）	*(内建)*	x86 Linux 常见 16B，很多实现为 8B
指针/引用	`T*` / `T&`	指针 8(64 位)	*(内建)*	引用本身无独立大小；指针大小 = 地址宽度
空指针常量	`nullptr_t` (`nullptr`)	—	*(内建)*	C++11 起

1.1.1.2 固定宽度整数

类型	典型大小	头文件
`std::int8_t` / `std::uint8_t`	1	`<cstdint>`
`std::int16_t` / `std::uint16_t`	2	`<cstdint>`
`std::int32_t` / `std::uint32_t`	4	`<cstdint>`
`std::int64_t` / `std::uint64_t`	8	`<cstdint>`
对应最大最小宏	如 `INT32_MAX/INT32_MIN/UINT32_MAX`	`<cstdint>`

1.1.2 常量类型

1.1.2.1 宏

宏	含义	头文件
`CHAR_BIT`	`char` 的位数（通常 8）	`<climits>`
`SCHAR_MIN/SCHAR_MAX/UCHAR_MAX`	`signed/unsigned char` 极值	`<climits>`
`CHAR_MIN/CHAR_MAX`	`char` 极值（实现相关）	`<climits>`
`SHRT_MIN/SHRT_MAX/USHRT_MAX`	`short` 极值	`<climits>`
`INT_MIN/INT_MAX/UINT_MAX`	`int` 极值	`<climits>`
`LONG_MIN/LONG_MAX/ULONG_MAX`	`long` 极值	`<climits>`
`LLONG_MIN/LLONG_MAX/ULLONG_MAX`	`long long` 极值	`<climits>`

宏	含义	头文件
MB_LEN_MAX	多字节字符最大字节数	<climits>
FLT_MIN/FLT_MAX/DBL_MIN/DBL_MAX/LDBL_MIN/LDBL_MAX	浮点正最小/最大规格化数	<cfloat>
FLT_EPSILON/DBL_EPSILON/LDBL_EPSILON	机器精度	<cfloat>
FLT_DIG/DBL_DIG/LDBL_DIG	十进制有效位数	<cfloat>
SIZE_MAX	size_t 最大值	<cstddef>（或 <stdint.h>）
PTRDIFF_MIN/PTRDIFF_MAX	ptrdiff_t 极值	<cstddef>
WCHAR_MIN/WCHAR_MAX	wchar_t 极值	<climits>

```cpp
#include <limits>
int imin = std::numeric_limits<int>::min();
double dmax = std::numeric_limits<double>::max();
float eps = std::numeric_limits<float>::epsilon();
```

1.1.2.2 limits

1.1.3 stl 数据结构

1.1.3.1 序列式容器

容器	头文件	典型操作复杂度/特点
vector<T>	<vector>	动态数组；push_back 摊还 $O(1)$，随机访问 $O(1)$
deque<T>	<deque>	分段连续；首尾插入 $O(1)$，随机访问 $O(1)$
list<T>	<list>	双向链表；任意位置插入/删除 $O(1)$（已定位迭代器）
forward_list<T>	<forward_list>	单向链表；更轻量
array<T,N>	<array>	固定大小、栈分配，随机访问 $O(1)$
basic_string<Ch> / string/u8string…	<string>	连续存储、面向文本

1.1.3.2 关联式容器

容器	头文件	典型复杂度
set<T> / multiset<T>	<set>	查找/插入/删除 $O(\log N)$
map<Key,T> / multimap<Key,T>	<map>	查找/插入/删除 $O(\log N)$

1.1.3.3 无序容器

容器	头文件	典型复杂度
unordered_set<T> / unordered_multiset<T>	<unordered_set>	均摊 $O(1)$ 查找/增删
unordered_map<K,T> / unordered_multimap<K,T>	<unordered_map>	均摊 $O(1)$ 查找/增删

1.1.3.4 容器适配器

适配器	头文件	基于	特点
stack<T>	<stack>	默认 deque	LIFO
queue<T>	<queue>	默认 deque	FIFO
priority_queue<T>	<queue>	默认 vector+ 堆	取顶 $O(1)$，入堆 $O(\log N)$

1.1.3.5 结构容器

组件	头文件	用途
pair / make_pair	<utility>	二元组
tuple / make_tuple / tie / get<N>	<tuple>	多元组
optional<T>	<optional>	可空值 (C++17)
variant<Ts...>	<variant>	类型安全联合 (C++17)
any	<any>	运行时类型擦除 (C++17)
bitset<N>	<bitset>	固定大小位集
span<T>		非 owning 视图 (C++20)
string_view	<string_view>	只读字符串视图 (C++17)
initializer_list<T>	<initializer_list>	花括号初始化支持
array 重申	<array>	固定数组容器

迭代器与算法：算法在 <algorithm>、数值算法在 <numeric>；迭代器工具在 <iterator> (如 back_inserter、iota 在 <numeric>)。

1.2 ASCII 表

二进制	八进制	十进制	十六进制	字符/缩写	解释
00000000	000	0	00	NUL (NULL)	空字符
00000001	001	1	01	SOH (Start Of Headling)	标题开始
00000010	002	2	02	STX (Start Of Text)	正文开始
00000011	003	3	03	ETX (End Of Text)	正文结束
00000100	004	4	04	EOT (End Of Transmission)	传输结束
00000101	005	5	05	ENQ (Enquiry)	请求
00000110	006	6	06	ACK (Acknowledge)	回应/响应/收到通知
00000111	007	7	07	BEL (Bell)	响铃
00001000	010	8	08	BS (Backspace)	退格
00001001	011	9	09	HT (Horizontal Tab)	水平制表符
00001010	012	10	0A	LF/NL(Line Feed/New Line)	换行键
00001011	013	11	0B	VT (Vertical Tab)	垂直制表符
00001100	014	12	0C	FF/NP (Form Feed/New Page)	换页键
00001101	015	13	0D	CR (Carriage Return)	回车键
00001110	016	14	0E	SO (Shift Out)	不用切换
00001111	017	15	0F	SI (Shift In)	启用切换
00010000	020	16	10	DLE (Data Link Escape)	数据链路转义
00010001	021	17	11	DC1/XON (Device Control 1/Transmission On)	设备控制 1/传输开始
00010010	022	18	12	DC2 (Device Control 2)	设备控制 2
00010011	023	19	13	DC3/XOFF (Device Control 3/Transmission Off)	设备控制 3/传输中断
00010100	024	20	14	DC4 (Device Control 4)	设备控制 4
00010101	025	21	15	NAK (Negative Acknowledge)	无响应/非正常响应/拒绝接收
00010110	026	22	16	SYN (Synchronous Idle)	同步空闲
00010111	027	23	17	ETB (End of Transmission Block)	传输块结束/块传输终止
00011000	030	24	18	CAN (Cancel)	取消
00011001	031	25	19	EM (End of Medium)	已到介质末端/介质存储已满/介质中断
00011010	032	26	1A	SUB (Substitute)	替补/替换
00011011	033	27	1B	ESC (Escape)	逃离/取消
00011100	034	28	1C	FS (File Separator)	文件分割符
00011101	035	29	1D	GS (Group Separator)	组分隔符/分组符
00011110	036	30	1E	RS (Record Separator)	记录分离符
00011111	037	31	1F	US (Unit Separator)	单元分隔符
00100000	040	32	20	(Space)	空格
00100001	041	33	21	!	
00100010	042	34	22	”	
00100011	043	35	23	#	

二进制	八进制	十进制	十六进制	字符/缩写	解释
00100100	044	36	24	$	
00100101	045	37	25	%	
00100110	046	38	26	&	
00100111	047	39	27	’	
00101000	050	40	28	(	
00101001	051	41	29	)	
00101010	052	42	2A	*	
00101011	053	43	2B	+	
00101100	054	44	2C	,	
00101101	055	45	2D	-	
00101110	056	46	2E	.	
00101111	057	47	2F	/	
00110000	060	48	30	0	
00110001	061	49	31	1	
00110010	062	50	32	2	
00110011	063	51	33	3	
00110100	064	52	34	4	
00110101	065	53	35	5	
00110110	066	54	36	6	
00110111	067	55	37	7	
00111000	070	56	38	8	
00111001	071	57	39	9	
00111010	072	58	3A	:	
00111011	073	59	3B	;	
00111100	074	60	3C	<	
00111101	075	61	3D	=	
00111110	076	62	3E	>	
00111111	077	63	3F	?	
01000000	100	64	40	@	
01000001	101	65	41	A	
01000010	102	66	42	B	
01000011	103	67	43	C	
01000100	104	68	44	D	
01000101	105	69	45	E	
01000110	106	70	46	F	
01000111	107	71	47	G	
01001000	110	72	48	H	
01001001	111	73	49	I	
01001010	112	74	4A	J	
01001011	113	75	4B	K	
01001100	114	76	4C	L	
01001101	115	77	4D	M	
01001110	116	78	4E	N	
01001111	117	79	4F	O	
01010000	120	80	50	P	
01010001	121	81	51	Q	
01010010	122	82	52	R	
01010011	123	83	53	S	
01010100	124	84	54	T	
01010101	125	85	55	U	
01010110	126	86	56	V	
01010111	127	87	57	W	
01011000	130	88	58	X	
01011001	131	89	59	Y	
01011010	132	90	5A	Z	

二进制	八进制	十进制	十六进制	字符/缩写	解释
01011011	133	91	5B	[	
01011100	134	92	5C	\	
01011101	135	93	5D	]	
01011110	136	94	5E	^	
01011111	137	95	5F	_	
01100000	140	96	60	‘	
01100001	141	97	61	a	
01100010	142	98	62	b	
01100011	143	99	63	c	
01100100	144	100	64	d	
01100101	145	101	65	e	
01100110	146	102	66	f	
01100111	147	103	67	g	
01101000	150	104	68	h	
01101001	151	105	69	i	
01101010	152	106	6A	j	
01101011	153	107	6B	k	
01101100	154	108	6C	l	
01101101	155	109	6D	m	
01101110	156	110	6E	n	
01101111	157	111	6F	o	
01110000	160	112	70	p	
01110001	161	113	71	q	
01110010	162	114	72	r	
01110011	163	115	73	s	
01110100	164	116	74	t	
01110101	165	117	75	u	
01110110	166	118	76	v	
01110111	167	119	77	w	
01111000	170	120	78	x	
01111001	171	121	79	y	
01111010	172	122	7A	z	
01111011	173	123	7B	{	
01111100	174	124	7C	|	
01111101	175	125	7D	}	
01111110	176	126	7E	~	
01111111	177	127	7F	DEL (Delete)	删除

1.3 Struct

- 一个典型的 struct 包括如下部分：

```
// O(1) 可删除/插入/存在检查的“索引集合”
struct IndexedSet {
    vector<int> items = vector<int>(10, 0);       // 对象（初始化）
    vector<int> pos;         // 对象（不初始化）
    IndexedSet() {}
    IndexedSet(int n): pos(n, -1) {} //构造函数

    bool operator < (const IndexedSet & target){ // 重载运算符
        return (this->items[0] < target.items[0]);
    }

    void ensure(int n) { if ((int)pos.size() < n) pos.assign(n, -1); } //声明函数
```

```
    bool has(int x) const { return x >= 0 && x < (int)pos.size() && pos[x] != -1; }

    //其他的函数
};
```

1.3.1 1. 树节点 TreeNode（常用于二叉树题目）

```
template<typename T>
struct TreeNode {
    T val;
    TreeNode<T>* left;
    TreeNode<T>* right;
    TreeNode(T x) : val(x), left(nullptr), right(nullptr) {}
};
```

常用场景：二叉树遍历（前序/中序/后序/层次）、路径和、LCA。

1.3.2 2. N 叉树节点 NaryNode

```
template<typename T>
struct NaryNode {
    T val;
    std::vector<NaryNode<T>*> children;
    NaryNode(T x) : val(x) {}
};
```

常用场景：多叉树 DFS/BFS、树的层序遍历。

1.3.3 3. 图的邻接表 Graph

```
template<typename T>
struct Graph {
    int n;  // 节点数
    std::vector<std::vector<T>> adj; // 邻接表
    Graph(int n): n(n), adj(n) {}
    void addEdge(int u, int v, bool directed=false){
        adj[u].push_back(v);
        if(!directed) adj[v].push_back(u);
    }
};
```

常用场景：DFS/BFS 遍历图，判环，连通分量。

1.3.4 4. 带权图（边权 w）

```
template<typename T>
struct WeightedGraph {
    int n;
    std::vector<std::vector<std::pair<int,T>>> adj;
    WeightedGraph(int n): n(n), adj(n) {}
    void addEdge(int u,int v,T w,bool directed=false){
        adj[u].push_back({v,w});
        if(!directed) adj[v].push_back({u,w});
    }
};
```

常用场景：0-1 BFS、Dijkstra、最短路。

1.3.5 5. 并查集（Union-Find，用于连通性）

```
struct UnionFind {
    std::vector<int> parent, rank;
    UnionFind(int n): parent(n), rank(n,0){
        for(int i=0;i<n;i++) parent[i]=i;
    }
    int find(int x){
        if(parent[x]!=x) parent[x]=find(parent[x]);
        return parent[x];
    }
    void unite(int x,int y){
        int rx=find(x), ry=find(y);
        if(rx==ry) return;
        if(rank[rx]<rank[ry]) parent[rx]=ry;
        else if(rank[rx]>rank[ry]) parent[ry]=rx;
        else { parent[ry]=rx; rank[rx]++; }
    }
};
```

常用场景：岛屿问题（并查集解法）、判连通性、最小生成树。

1.3.6 6. 通用队列节点（用于 BFS 状态搜索）

```
template<typename T>
struct State {
    T x;          // 核心状态（数字/位置/字符串）
    int step;     // 当前步数
    State(T _x,int _step):x(_x),step(_step){}
};
```

常用场景：数字变换题、棋盘最短步数、数码难题。

1.3.7 7. 网格点（二维坐标）

```
struct Point {
    int x,y;
    Point(int _x,int _y):x(_x),y(_y){}
};
```

常用场景：迷宫 BFS/DFS，最短路。

1.3.8 8. 带状态的迷宫点（位置 + 钥匙/能力）

```
struct Node {
    int x,y;
    int mask;    // 额外状态，例如钥匙集合
    Node(int _x,int _y,int _mask):x(_x),y(_y),mask(_mask){}
};
```

常用场景：迷宫带钥匙、带层次 BFS

1.3.9 9. 通用边（用于 Kruskal、Prim）

```cpp
template<typename T>
struct Edge {
    int u,v;
    T w;
    Edge(int _u,int _v,T _w):u(_u),v(_v),w(_w){}
};
```

常用场景：最小生成树、最短路、图论题。

2 String

C++ 标准库中的 `string` 类是字符串操作的核心工具。本指南全面介绍其功能和用法，帮助你高效处理字符串。

2.1 基本操作

2.1.1 构造与初始化

```cpp
// 多种初始化方式
string s1;                          // 空字符串
string s2("Hello");                 // C 风格字符串初始化
string s3 = "Hello";                // 赋值初始化
string s4(5, 'a');                  // 创建字符串 "aaaaa"
string s5(s2, 1, 3);                // 从 s2 的位置 1 开始的 3 个字符 ("ell")
string s6(s2.begin() + 1, s2.end() - 1); // 使用迭代器 ("ell")

// C++11 统一初始化
string s7 = {"Hello"};
string s8{"World"};
```

2.1.2 赋值操作

```cpp
string s;
s = "Hello";                    // C 风格字符串赋值
s = another_string;             // string 对象赋值
s = 'A';                        // 单个字符赋值

// assign 方法
s.assign("Hello");              // 替换内容
s.assign(5, 'a');               // 替换为 5 个'a'
s.assign(str, 2, 3);            // 替换为 str 从位置 2 开始的 3 个字符
s.assign(str.begin(), str.begin() + 3); // 使用迭代器范围赋值
```

2.1.3 访问元素

```cpp
string str = "Hello";
char c1 = str[0];               // 'H'，不检查边界
char c2 = str.at(1);            // 'e'，会检查边界，越界时抛出 out_of_range 异常

// C++11 新增
char& front = str.front();      // 引用首字符 'H'
char& back = str.back();        // 引用尾字符 'o'

// C 字符串转换
const char* c_str = str.c_str();  // 以空字符结尾的 C 风格字符串
```

```cpp
const char* data = str.data();     // 获取内部数据指针 (C++11 前不保证空字符结尾)

// C++17 数据视图
string_view sv(str);               // 轻量级、非拥有的字符串视图
```

2.2 字符串操作

2.2.1 添加与连接

```cpp
string s = "Hello";
s += " World";                     // 追加 C 风格字符串，现在 s 为"Hello World"
s += '!';                          // 追加字符，现在 s 为"Hello World!"

// append 方法
s.append(" Welcome");              // 追加字符串
s.append(3, '!');                  // 追加 3 个感叹号
s.append(another_str, 0, 5);       // 追加 another_str 的前 5 个字符

// 单字符添加
s.push_back('.');                  // 在末尾添加一个点，C++11

// 字符串连接
string s1 = "Hello";
string s2 = "World";
string s3 = s1 + " " + s2;         // "Hello World"
```

2.2.2 插入操作

```cpp
string s = "Hello";
s.insert(5, " World");             // 在位置 5 插入字符串，结果"Hello World"
s.insert(5, 3, '!');               // 在位置 5 插入 3 个'!'，结果"Hello!!! World"
s.insert(0, another_str, 0, 3); // 在开头插入 another_str 的前 3 个字符

// 使用迭代器插入
auto it = s.begin() + 5;
s.insert(it, 'X');                 // 在位置 5 插入字符'X'
s.insert(it, 3, 'Y');              // 在位置 5 插入 3 个'Y'
```

2.2.3 删除操作

```cpp
string s = "Hello World";
s.erase(5, 1);                     // 删除位置 5 的一个字符 (空格)，结果"HelloWorld"

// 使用迭代器删除
auto it = s.begin() + 5;
s.erase(it);                       // 删除迭代器指向的字符
s.erase(s.begin(), s.begin() + 5); // 删除范围，结果为"World"

// C++11
s.pop_back();                      // 删除最后一个字符
s.clear();                         // 清空整个字符串
```

2.2.4 替换操作

```
string s = "Hello World";
s.replace(0, 5, "Hi");         // 用"Hi" 替换前 5 个字符，结果"Hi World"
s.replace(3, 0, "Beautiful "); // 在位置 3 插入字符串，结果"Hi Beautiful World"
s.replace(s.begin(), s.begin() + 2, "Hey"); // 使用迭代器替换，结果"Hey Beautiful World"
```

2.2.5 子串提取

```
string s = "Hello World";
string sub = s.substr(0, 5);  // "Hello"，从位置 0 开始的 5 个字符
string sub2 = s.substr(6);    // "World"，从位置 6 到结尾
```

2.3 字符串信息与搜索

2.3.1 大小与容量

```
string s = "Hello World";
size_t len = s.length();       // 11，字符串长度
size_t size = s.size();        // 11，同 length()
bool empty = s.empty();        // false，检查是否为空

// 容量管理
size_t cap = s.capacity();     // 获取当前容量（可能大于 size）
s.reserve(100);                // 预留至少 100 个字符的空间
s.shrink_to_fit();             // 释放多余容量，C++11
s.resize(5);                   // 调整大小为 5，结果"Hello"
s.resize(10, '*');             // 调整大小为 10，不足部分用 * 填充，结果"Hello*****"
```

2.3.2 查找与搜索

```
string s = "Hello World, Hello C++";

// 正向查找
size_t pos1 = s.find("Hello");        // 0，第一次出现的位置
size_t pos2 = s.find("Hello", 1);     // 13，从位置 1 开始查找
size_t pos3 = s.find('W');            // 6，查找字符
size_t pos4 = s.find_first_of("aeiou"); // 1，第一个元音字母的位置 ('e')

// 反向查找
size_t rpos1 = s.rfind("Hello");      // 13，最后一次出现的位置
size_t rpos2 = s.find_last_of("aeiou"); // 19，最后一个元音字母的位置 ('e')

// 查找不在集合中的字符
size_t pos5 = s.find_first_not_of("Helo "); // 6，第一个不是"Helo " 中字符的位置 ('W')
size_t pos6 = s.find_last_not_of("+ "); // 19，最后一个不是"+ " 中字符的位置 ('e')

// 检查是否找到
if (s.find("Java") == string::npos) {
    cout << " 未找到 Java" << endl;
}
```

2.3.3 比较操作

```
string s1 = "Apple";
string s2 = "Banana";

// 比较方法
int comp1 = s1.compare(s2);           // 负值，s1 < s2
int comp2 = s1.compare(0, 1, "A");    // 0，s1 的第一个字符等于"A"
int comp3 = s1.compare(0, 3, s2, 0, 3); // 比较部分子串

// 比较运算符
bool b1 = (s1 == s2);  // false
bool b2 = (s1 != s2);  // true
bool b3 = (s1 < s2);   // true，按字典序比较
bool b4 = (s1 > s2);   // false

// C++20 前缀后缀检查
// bool b5 = s1.starts_with("App");  // true，C++20
// bool b6 = s1.ends_with("le");     // true，C++20

// C++20 之前的前缀后缀检查实现
bool starts_with(const string& str, const string& prefix) {
    return str.size() >= prefix.size() &&
           str.compare(0, prefix.size(), prefix) == 0;
}

bool ends_with(const string& str, const string& suffix) {
    return str.size() >= suffix.size() &&
           str.compare(str.size() - suffix.size(), suffix.size(), suffix) == 0;
}
```

2.4 高级功能

2.4.1 字符串与数值转换 (C++11)

```
// 【字符串转数值】- ACM 比赛常用功能
string num_str = "123.456";

// 整数转换 - 掌握这三个基本够用
int i = stoi("42");               // 字符串转 int: 42
long l = stol("1234567890");      // 字符串转 long
long long ll = stoll("1234567890123456789"); // 字符串转 long long (ACM 中常用)

// 浮点数转换 - 记住这两个最常用
double d = stod("2.71828");       // 字符串转 double (ACM 中常用)
float f = stof("3.14");           // 字符串转 float

// 其他整数转换函数
unsigned long ul = stoul("12345");
unsigned long long ull = stoull("12345678901234567890");

// 进制转换 - ACM 比赛非常实用
// 第二个参数是指针，用于存储成功解析的字符数量
// 第三个参数是进制，默认为 10
size_t pos = 0;
int hex_value = stoi("0xFF", &pos, 16);  // 十六进制转换: 255, pos = 4
```

```cpp
int bin_value = stoi("101010", nullptr, 2);  // 二进制转换: 42
int oct_value = stoi("052", nullptr, 8);  // 八进制转换: 42

// 错误处理 - 别忘了这点!
try {
    int invalid = stoi("not_a_number");  // 会抛出 std::invalid_argument 异常
    int overflow = stoi("999999999999");  // 太大的数会抛出 std::out_of_range 异常
} catch (const invalid_argument& e) {
    cout << " 无效输入: " << e.what() << endl;
} catch (const out_of_range& e) {
    cout << " 数值超出范围: " << e.what() << endl;
}

// 【数值转字符串】- 简单好记
string s1 = to_string(42);        // 整数转字符串: "42"
string s2 = to_string(3.14159);  // 浮点数转字符串: "3.14159"

// 【ACM 比赛技巧】: 当需要控制输出格式时, 使用 stringstream 比 to_string 更灵活
#include <iomanip>
#include <sstream>
ostringstream oss;
oss << fixed << setprecision(3) << 3.14159;  // 控制小数点后位数
string formatted = oss.str();    // "3.142"

// 输出特定进制
ostringstream hex_oss;
hex_oss << hex << uppercase << 255;  // 十六进制大写
string hex_str = hex_oss.str();      // "FF"

ostringstream bin_oss;
bin_oss << bitset<8>(42);            // 二进制, 8 位
string bin_str = bin_oss.str();      // "00101010"
```

ACM 比赛提示: 1. stoi, stod 和 to_string 是最常用的转换函数, 重点掌握 2. 处理大整数时, 记得使用 stoll 而不是 stoi 3. 字符串数值转换错误会抛出异常, 在稳定性要求高的场景需要处理 4. 自定义格式化数值时, 使用 stringstream 比 to_string 更灵活 5. 进制转换功能在位运算和编码题中非常有用

字符串流操作

```cpp
#include <sstream>

// 字符串解析
string data = "123 3.14 Hello";
istringstream iss(data);
int i; double d; string s;
iss >> i >> d >> s;  // i=123, d=3.14, s="Hello"

// 字符串构建
ostringstream oss;
oss << "Value: " << 42 << ", Pi: " << 3.14;
string result = oss.str();  // "Value: 42, Pi: 3.14"

// 数值格式化
ostringstream oss2;
oss2 << fixed << setprecision(2) << 3.14159;  // 需要 #include <iomanip>
string formatted = oss2.str();  // "3.14"
```

2.4.2 内存管理与性能优化

```
// 预分配内存避免频繁重新分配
string result;
result.reserve(1000);  // 预留 1000 个字符的空间
for (int i = 0; i < 100; i++) {
    result += "Some text ";  // 不会导致频繁重新分配内存
}

// 移除多余容量
result.shrink_to_fit();  // C++11

// 使用 swap 技巧释放内存
{
    string(result).swap(result);  // C++11 前清空并最小化容量的技巧
    // C++11 后可以直接用: result.clear(); result.shrink_to_fit();
}

// 移动语义 (C++11), 避免不必要的拷贝
string build_message() {
    string msg = "Hello " + user_name + "!";
    return msg;  // 编译器可能使用移动而非拷贝
}

string message = std::move(build_message());  // 显式移动
```

2.5 实用案例

2.5.1 案例 1: 分割字符串

```
// 按分隔符分割字符串
vector<string> split(const string& s, char delimiter) {
    vector<string> tokens;
    istringstream tokenStream(s);
    string token;

    while (getline(tokenStream, token, delimiter)) {
        if (!token.empty()) {  // 忽略空标记 (如连续分隔符)
            tokens.push_back(token);
        }
    }

    return tokens;
}

// 使用示例
string csv = "apple,banana,cherry,date";
vector<string> fruits = split(csv, ',');
// fruits 包含: "apple", "banana", "cherry", "date"
```

2.5.2 案例 2: 字符串替换

```
// 替换字符串中的所有匹配项
string replace_all(string str, const string& from, const string& to) {
    size_t pos = 0;
```

```
    while ((pos = str.find(from, pos)) != string::npos) {
        str.replace(pos, from.length(), to);
        pos += to.length();  // 跳过刚刚插入的内容
    }
    return str;
}

// 使用示例
string text = "Hello [name], welcome to [city]!";
text = replace_all(text, "[name]", "Alice");
text = replace_all(text, "[city]", "Wonderland");
// 结果: "Hello Alice, welcome to Wonderland!"
```

2.5.3 案例 3: URL 编码/解码

```
// URL 编码
string url_encode(const string &value) {
    ostringstream escaped;
    escaped.fill('0');
    escaped << hex;

    for (char c : value) {
        // 保留字母数字和一些特殊字符
        if (isalnum(c) || c == '-' || c == '_' || c == '.' || c == '~') {
            escaped << c;
        } else {
            // 其他字符编码为%XX 形式
            escaped << uppercase;
            escaped << '%' << setw(2) << int((unsigned char)c);
            escaped << nouppercase;
        }
    }

    return escaped.str();
}

// URL 解码
string url_decode(const string &encoded) {
    string result;
    for (size_t i = 0; i < encoded.length(); ++i) {
        if (encoded[i] == '%') {
            if (i + 2 < encoded.length()) {
                int value;
                istringstream is(encoded.substr(i + 1, 2));
                if (is >> hex >> value) {
                    result += static_cast<char>(value);
                    i += 2;
                } else {
                    result += '%';
                }
            } else {
                result += '%';
            }
        } else if (encoded[i] == '+') {
            result += ' ';
        } else {
```

```
            result += encoded[i];
        }
    }
    return result;
}
```

2.5.4 案例 4: 文本修剪 (Trim)

```
// 去除字符串首尾的空白字符
string trim(const string& str) {
    // 找到第一个非空白字符
    const string whitespace = " \t\n\r\f\v";
    size_t start = str.find_first_not_of(whitespace);

    // 如果字符串全是空白字符
    if (start == string::npos)
        return "";

    // 找到最后一个非空白字符
    size_t end = str.find_last_not_of(whitespace);

    // 提取并返回子串
    return str.substr(start, end - start + 1);
}

// 去除字符串左侧空白
string ltrim(const string& str) {
    const string whitespace = " \t\n\r\f\v";
    size_t start = str.find_first_not_of(whitespace);
    return (start == string::npos) ? "" : str.substr(start);
}

// 去除字符串右侧空白
string rtrim(const string& str) {
    const string whitespace = " \t\n\r\f\v";
    size_t end = str.find_last_not_of(whitespace);
    return (end == string::npos) ? "" : str.substr(0, end + 1);
}
```

2.5.5 案例 5: 大小写转换

```
// 将字符串转换为大写
string to_upper(string str) {
    transform(str.begin(), str.end(), str.begin(),
              [](unsigned char c){ return toupper(c); });
    return str;
}

// 将字符串转换为小写
string to_lower(string str) {
    transform(str.begin(), str.end(), str.begin(),
              [](unsigned char c){ return tolower(c); });
    return str;
}
```

```cpp
// 将字符串首字母大写
string capitalize(string str) {
    if (!str.empty()) {
        str[0] = toupper(str[0]);
    }
    return str;
}
```

2.6 常见错误与最佳实践

2.6.1 潜在问题

1. **越界访问** - 使用 [] 不进行边界检查，可能导致未定义行为

```cpp
string s = "Hi";
char c = s[10];  // 未定义行为！应使用 s.at(10) 触发异常
```

2. **迭代器失效** - 修改字符串可能导致迭代器失效

```cpp
string s = "Hello";
auto it = s.begin() + 2;
s += " World";   // 可能导致 it 失效
*it = 'X';       // 危险操作，可能导致未定义行为
```

3. **忽略 string::npos 检查** - 未检查 find 结果

```cpp
string s = "Hello";
size_t pos = s.find("World");
string sub = s.substr(pos);     // 错误！未检查 pos 是否为 npos
```

4. **C 字符串转换问题** - c_str() 指针在 string 修改后可能失效

```cpp
string s = "Hello";
const char* ptr = s.c_str();
s += " World";       // ptr 可能失效
cout << ptr;         // 不安全的操作
```

2.6.2 最佳实践

1. **使用合适的方法进行边界检查**

```cpp
string s = "Hello";
// 安全访问 - 越界会抛出异常
try {
    char c = s.at(10);
} catch (const out_of_range& e) {
    cerr << " 越界访问: " << e.what() << endl;
}
```

2. **字符串连接效率优化**

```cpp
// 低效：每次 += 都可能重新分配内存
string result;
for (int i = 0; i < 1000; ++i) {
    result += to_string(i);
}
```

```cpp
// 高效：预先分配足够空间
string result;
result.reserve(10000);  // 预估大小
for (int i = 0; i < 1000; ++i) {
    result += to_string(i);
}

// 或使用字符串流
ostringstream oss;
for (int i = 0; i < 1000; ++i) {
    oss << i;
}
string result = oss.str();
```

3. **安全地使用 find 结果**

```cpp
string s = "Hello World";
size_t pos = s.find("World");
if (pos != string::npos) {
    string sub = s.substr(pos);  // 安全
}
```

4. **使用 string_view 减少拷贝 (C++17)**

```cpp
// 使用 string_view 作为函数参数，避免字符串拷贝
bool starts_with(string_view str, string_view prefix) {
    return str.substr(0, prefix.size()) == prefix;
}
```

5. **避免不必要的临时对象**

```cpp
// 低效：创建多个临时 string 对象
string result = string("Hello ") + string("World");

// 高效：直接使用字符串字面值
string result = "Hello " + string("World");
```

2.7 总结

C++ string 类提供了丰富而强大的字符串处理功能。掌握这些操作可以帮助你更高效地处理各种文本处理任务。关键是理解字符串的内存管理方式，选择合适的方法进行操作，并注意潜在的陷阱。

随着 C++ 标准的发展，string 类不断获得新功能（如 C++17 的 string_view 和 C++20 的 starts_with/ends_with 方法），使字符串处理变得更加便捷和高效。

3 STL 容器

3.1 序列式容器

3.1.1 vector-动态数组

核心特点：连续内存、动态扩容、高效随机访问、尾部操作高效

动态数组（vector）是最常用的容器。它的内存是连续分配的，因此支持快速随机访问；但当容量不足时，会按照一定的扩展策略（通常是倍增扩容）进行内存重新分配，这可能导致所有迭代器失效。

3.1.1.1 详细说明

- **内存管理**：内存扩容通常遵循几何级增长（如 1.5 倍或 2 倍），确保插入元素的均摊时间复杂度为 $\mathrm{O}(1)$
- **迭代器稳定性**：迭代器和指针在重新分配时全部失效，需要注意使用返回的迭代器或重新获取索引
- **异常安全**：提供如 `at()` 的边界检查函数，在访问越界时抛出 `std::out_of_range` 异常
- **性能特点**：随机访问 $\mathrm{O}(1)$，尾部插入均摊 $\mathrm{O}(1)$，中间或头部插入 $\mathrm{O}(\mathrm{n})$

小贴士：适用于需要快速随机访问和尾部操作的场景。例如，存储一组数据，并需要频繁访问其中的元素。

3.1.1.2 使用场景

- **需要频繁随机访问元素**：`vector` 提供了高效的随机访问能力，可以通过索引直接访问元素
- **主要在尾部进行插入和删除操作**：在尾部插入和删除元素的时间复杂度为 $\mathrm{O}(1)$，效率很高
- **需要动态改变容器大小**：`vector` 可以根据需要动态调整大小，方便存储不确定数量的元素
- **需要与 C 风格数组互操作**：通过 `data()` 方法可获取底层连续内存指针

3.1.1.3 常用成员函数及使用说明

函数	描述	示例	复杂度
`push_back(const T& value)`	将元素添加到容器末尾	`vec.push_back(5);` // 在末尾添加数值 5	均摊 $\mathrm{O}(1)$
`pop_back()`	移除末尾元素（不返回值）	`vec.pop_back();` // 删除最后一个元素	$\mathrm{O}(1)$
`size()`	返回容器中的元素数量	`size_t len = vec.size();` // 获取元素个数	$\mathrm{O}(1)$
`empty()`	检查容器是否为空	`if(vec.empty()) {...}` // 判断是否为空	$\mathrm{O}(1)$
`at(size_t pos)`	访问指定位置元素（带边界检查）	`int val = vec.at(3);` // 访问索引 3 的元素，越界会抛出异常	$\mathrm{O}(1)$
`operator[]`	访问指定位置元素（无边界检查）	`int val = vec[0];` // 访问首个元素，越界行为未定义	$\mathrm{O}(1)$
`clear()`	清空所有元素	`vec.clear();` // 移除所有元素，容量不变	$\mathrm{O}(\mathrm{n})$
`insert(iterator pos, const T& value)`	在指定位置插入元素	`auto it = vec.insert(vec.begin()+2, 10);` // 在索引 2 的位置插入值 10	$\mathrm{O}(\mathrm{n})$
`erase(iterator pos)`	删除指定位置的元素	`auto it = vec.erase(vec.begin()+1);` // 删除索引 1 位置的元素	$\mathrm{O}(\mathrm{n})$
`emplace_back(Args&&... args)`	在末尾原位构造元素	`vec.emplace_back(42);` // 直接构造值为 42 的元素	均摊 $\mathrm{O}(1)$
`reserve(size_t capacity)`	预留容量空间但不创建元素	`vec.reserve(100);` // 预留 100 个元素的空间，减少重新分配次数	最坏 $\mathrm{O}(\mathrm{n})$
`resize(size_t count)`	调整容器大小	`vec.resize(5);` // 调整大小为 5 个元素，多余的会被移除，不足的用默认值填充	最坏 $\mathrm{O}(\mathrm{n})$
`capacity()`	返回当前分配的容量	`size_t cap = vec.capacity();` // 获取容器当前容量	$\mathrm{O}(1)$
`front()`	返回首元素的引用	`int first = vec.front();` // 获取第一个元素值	$\mathrm{O}(1)$
`back()`	返回末元素的引用	`int last = vec.back();` // 获取最后一个元素值	$\mathrm{O}(1)$

函数	描述	示例	复杂度
`data()`	返回指向数据的指针	`int* ptr = vec.data();` // 获取底层数组指针，方便与 C 函数交互	O(1)
`shrink_to_fit()`	减少内存到适合大小	`vec.shrink_to_fit();` // 释放未使用的内存空间	O(n)
`assign(size_t count, const T& value)`	替换容器内容	`vec.assign(3, 0);` // 用 3 个 0 替换当前内容	O(n)
`swap(vector& other)`	与另一容器交换内容	`vec1.swap(vec2);` // 交换两个 vector 的内容	O(1)

```
// 多种构造方式
std::vector<int> vec1;                      // 默认构造，空容器
std::vector<int> vec2(5, 10);               // 构造包含 5 个值为 10 的元素
std::vector<int> vec3 = {1, 2, 3, 4, 5};    // 使用初始化列表
std::vector<int> vec4(vec3.begin(), vec3.end());  // 使用迭代器范围构造
```

3.1.1.4 构造和初始化示例

```
// 添加元素
vec1.push_back(15);           // 在末尾添加元素 15
vec1.emplace_back(20);        // 在末尾直接构造元素 20（通常比 push_back 高效）

// 在指定位置添加元素
auto it = vec1.begin() + 1;   // 指向第二个元素的迭代器
vec1.insert(it, 25);          // 在第二个位置插入 25
vec1.emplace(it, 30);         // 在第二个位置原位构造 30

// 删除元素
vec1.pop_back();              // 移除最后一个元素
vec1.erase(vec1.begin());     // 删除第一个元素
vec1.erase(vec1.begin(), vec1.begin() + 2);  // 删除前两个元素

// 清空容器
vec1.clear();                 // 移除所有元素
```

3.1.1.5 添加删除元素示例

```
// 多种元素访问方式
int first = vec1.front();     // 获取第一个元素
int last = vec1.back();       // 获取最后一个元素
int third = vec1[2];          // 直接访问索引 2 的元素（不检查边界）
int fourth = vec1.at(3);      // 访问索引 3 的元素（带边界检查）

// 使用迭代器遍历
for (auto it = vec1.begin(); it != vec1.end(); ++it) {
    std::cout << *it << " ";
}
// 使用范围 for 循环（C++11 起）
for (const auto& item : vec1) {
```

```
    std::cout << item << " ";
}
```

3.1.1.6 访问元素示例

```
// 容量管理
vec1.reserve(100);              // 预分配存储空间，减少重新分配
size_t current_capacity = vec1.capacity();  // 获取当前容量
vec1.resize(50);               // 调整大小为 50 个元素
vec1.resize(75, -1);           // 增大到 75 个元素，新元素用-1 填充
vec1.shrink_to_fit();          // 释放多余的内存
```

3.1.1.7 容量操作示例

```
// 排序和反转
std::sort(vec1.begin(), vec1.end());              // 升序排序整个 vector
std::sort(vec1.begin(), vec1.end(), std::greater<int>()); // 降序排序
std::reverse(vec1.begin(), vec1.end());           // 反转元素顺序

// 查找元素
auto it = std::find(vec1.begin(), vec1.end(), 42);
if (it != vec1.end()) {
    std::cout << " 找到元素 42，位置: " << std::distance(vec1.begin(), it) << std::endl;
}

// 拼接操作
std::vector<int> vec4 = {6, 7, 8};
vec1.insert(vec1.end(), vec4.begin(), vec4.end());  // 将 vec4 的所有元素追加到 vec1 末尾

// 使用 data() 与 C 风格函数交互
void legacy_function(int* array, size_t size) { /*...*/ }
legacy_function(vec1.data(), vec1.size());
```

3.1.1.8 其他常用操作

3.1.1.9 vector 迭代器失效情况总结

1. **添加元素时**：
   - 如果导致内存重新分配，则所有迭代器和引用都会失效
   - 如果没有导致重新分配，则插入点之前的迭代器有效，之后的失效
2. **删除元素时**：
   - 被删除位置之前的迭代器有效
   - 被删除位置及之后的迭代器全部失效

```
// 处理迭代器失效的正确方法
auto it = vec.begin() + 3;
it = vec.insert(it, 42);  // insert 返回指向新插入元素的迭代器
it = vec.erase(it);       // erase 返回指向被删除元素之后的迭代器
```

3.1.1.10 deque-双端队列

核心特点：分段连续内存、两端高效插入删除、随机访问

双端队列采用分段连续内存存储，每段固定大小，这使得两端的插入及删除操作非常高效。

3.1.1.11 详细说明

- **内存管理**：deque 采用分段连续内存存储，支持在两端 $O(1)$ 时间复杂度的插入和删除操作
- **迭代器稳定性**：分段存储使得 deque 可以在两端进行常数时间的操作，缺点是在中间插入时需要移动块指针，可能丢失部分迭代器
- **异常安全**：与 vector 不同，deque 不支持 reserve 操作，原因在于其非连续内存结构
- **性能特点**：两端插入删除 $O(1)$，中间插入删除 $O(n)$，随机访问 $O(1)$

注意：deque 不支持 reserve 操作，请参照示例使用。

3.1.1.12 使用场景

- **需要在两端频繁插入和删除元素**：deque 的两端操作都是 $O(1)$ 复杂度
- **需要随机访问元素**：支持下标访问，与 vector 类似
- **对中间元素的插入删除操作相对较少**：中间操作效率低于两端操作
- **不希望因容量改变导致迭代器全部失效**：相较于 vector，deque 在容量变化时只有部分迭代器失效

3.1.1.13 常用成员函数及使用说明

函数	描述	示例	复杂度
`push_front(const T& value)`	在容器前端添加元素	`deq.push_front(10); //` 在开始位置插入值 10	$O(1)$
`push_back(const T& value)`	在容器末尾添加元素	`deq.push_back(20); //` 在末尾添加值 20	$O(1)$
`pop_front()`	移除前端元素	`deq.pop_front(); //` 删除第一个元素	$O(1)$
`pop_back()`	移除末尾元素	`deq.pop_back(); //` 删除最后一个元素	$O(1)$
`size()`	获取元素个数	`size_t len = deq.size(); //` 获取元素数量	$O(1)$
`empty()`	检查是否为空	`if(deq.empty()) {...} //` 判断容器是否为空	$O(1)$
`at(size_t pos)`	访问指定位置元素（带边界检查）	`int val = deq.at(2); //` 安全访问索引 2 的元素	$O(1)$
`operator[size_t pos]`	访问指定位置元素（不检查边界）	`int val = deq[0]; //` 直接访问首个元素	$O(1)$
`clear()`	清空所有元素	`deq.clear(); //` 移除所有元素	$O(n)$
`insert(iterator pos, const T& value)`	在指定位置插入元素	`auto it = deq.insert(deq.begin()+1, 15); //` 在第二个位置插入 15	$O(n)$
`erase(iterator pos)`	删除指定位置的元素	`auto it = deq.erase(deq.begin()); //` 删除首个元素	$O(n)$
`emplace_front(Args&&... args)`	在前端原位构造元素	`deq.emplace_front(5); //` 在开头直接构造元素 5	$O(1)$
`emplace_back(Args&&... args)`	在末尾原位构造元素	`deq.emplace_back(25); //` 在末尾直接构造元素 25	$O(1)$
`resize(size_t count)`	调整容器大小	`deq.resize(10); //` 调整大小为 10 个元素	最坏 $O(n)$
`front()`	返回首元素的引用	`int first = deq.front(); //` 获取第一个元素	$O(1)$
`back()`	返回末元素的引用	`int last = deq.back(); //` 获取最后一个元素	$O(1)$
`shrink_to_fit()`	减少内存到适合大小	`deq.shrink_to_fit(); //` 释放未使用的内存（C++11 起）	$O(n)$

函数	描述	示例	复杂度
`assign(size_t count, const T& value)`	替换容器内容	`deq.assign(5, 1);` // 用 5 个值为 1 的元素替换内容	O(n)
`swap(deque& other)`	与另一容器交换内容	`deq1.swap(deq2);` // 交换两个 deque 的内容	O(1)

```
// 多种构造方式
std::deque<int> deq1;                       // 默认构造，空队列
std::deque<int> deq2(5, 10);                // 5 个值为 10 的元素
std::deque<int> deq3 = {1, 2, 3, 4, 5};     // 使用初始化列表
std::deque<int> deq4(deq3.begin(), deq3.begin() + 3);  // 使用迭代器范围构造
```

3.1.1.14 构造和初始化示例

```
// 前端操作
deq1.push_front(50);            // 在前端添加元素 50
deq1.emplace_front(40);         // 在前端原位构造元素 40
deq1.pop_front();               // 移除前端元素

// 后端操作
deq1.push_back(60);             // 在末尾添加元素 60
deq1.emplace_back(70);          // 在末尾原位构造元素 70
deq1.pop_back();                // 移除末尾元素
```

3.1.1.15 两端添加删除示例

```
// 中间位置操作
auto mid_it = deq1.begin() + deq1.size() / 2;  // 中间位置迭代器
deq1.insert(mid_it, 30);        // 在中间插入元素 30
deq1.erase(mid_it);             // 删除中间位置的元素
```

3.1.1.16 中间插入删除示例

```
// 访问元素
int first = deq1.front();       // 获取第一个元素
int last = deq1.back();         // 获取最后一个元素
int third = deq1[2];            // 通过索引访问（不检查边界）
int fourth = deq1.at(3);        // 通过 at 访问（会检查边界）

// 遍历元素
for (auto it = deq1.begin(); it != deq1.end(); ++it) {
    std::cout << *it << " ";
}
// 使用范围 for 循环
for (const auto& elem : deq1) {
    std::cout << elem << " ";
}
```

3.1.1.17 访问元素示例

```cpp
// 容量相关操作
size_t sz = deq1.size();       // 获取元素数量
bool is_empty = deq1.empty(); // 检查是否为空
deq1.resize(10);               // 调整大小为 10 个元素
deq1.resize(15, -1);           // 调整为 15 个元素，新元素用-1 填充
deq1.shrink_to_fit();          // 减少内存占用（C++11 起）
```

3.1.1.18 容量管理示例

```cpp
// 排序和反转
std::sort(deq1.begin(), deq1.end());  // 升序排序
std::reverse(deq1.begin(), deq1.end()); // 反转元素顺序

// 查找操作
auto find_it = std::find(deq1.begin(), deq1.end(), 30);
if (find_it != deq1.end()) {
    std::cout << " 找到元素 30" << std::endl;
}

// 拼接操作
std::deque<int> deq5 = {100, 200, 300};
deq1.insert(deq1.end(), deq5.begin(), deq5.end());  // 将 deq5 拼接到 deq1 末尾
```

3.1.1.19 其他常用操作

3.1.1.20 deque 与 vector 的区别

- **内存分配**：deque 使用分段连续内存，vector 使用单一连续内存
- **扩容策略**：deque 不需要像 vector 那样整体重新分配
- **两端操作**：deque 支持高效的两端操作，vector 只支持高效的尾部操作
- **迭代器失效**：deque 的迭代器在插入删除时部分失效，vector 可能全部失效

```cpp
// 何时选择 deque 而非 vector?
// 1. 需要频繁在两端添加删除元素
// 2. 不希望因容量变化导致全部迭代器失效
// 3. 元素数量较大且内存分配问题影响性能
```

3.1.2 list-双向链表

核心特点：节点分散存储、双向链接、高效中间插入删除

双向链表采用节点连续的内存链接，每个节点储存前后指针，允许在任意位置 $O(1)$ 插入或删除，但不支持索引访问。

3.1.2.1 详细说明

- **内存管理**：由于节点分散存储，list 在迭代时的局部性不如 vector，但在频繁中间修改时非常高效
- **迭代器稳定性**：即使插入或删除操作不会使其他迭代器失效，操作前仍需定位目标位置，时间复杂度为 $O(n)$
- **异常安全**：list 的插入和删除操作不会抛出异常，除非内存分配失败
- **性能特点**：中间插入删除 $O(1)$，随机访问 $O(n)$

优点：高效的中间插入删除；**缺点**：不支持随机访问。

3.1.2.2 使用场景

- **需要频繁在任意位置插入和删除元素**：list 的插入删除不会导致其他迭代器失效
- **不需要随机访问元素**：list 不支持下标访问，只能顺序遍历
- **需要频繁的元素重排**：list 的 splice 操作可以高效地合并、分割链表
- **元素需要在容器中保持稳定位置**：元素的位置不会因为插入删除而改变

3.1.2.3 常用成员函数及使用说明

函数	描述	示例	复杂度
`push_front(const T& value)`	在链表开头添加元素	`lst.push_front(10); //` 在开头插入值 10	O(1)
`push_back(const T& value)`	在链表末尾添加元素	`lst.push_back(20); //` 在末尾插入值 20	O(1)
`pop_front()`	移除链表开头元素	`lst.pop_front(); //` 删除第一个元素	O(1)
`pop_back()`	移除链表末尾元素	`lst.pop_back(); //` 删除最后一个元素	O(1)
`size()`	获取元素个数	`size_t len = lst.size(); //` 获取链表长度	O(1)
`empty()`	检查是否为空	`if(lst.empty()) {...} //` 判断链表是否为空	O(1)
`clear()`	清空所有元素	`lst.clear(); //` 移除所有元素	O(n)
`insert(iterator pos, const T& value)`	在指定位置插入元素	`auto it = lst.insert(pos, 30); //` 在 pos 位置插入值 30	O(1)
`erase(iterator pos)`	删除指定位置的元素	`auto next_it = lst.erase(it); //` 删除 it 指向的元素	O(1)
`emplace_front(Args&&... args)`	在开头原位构造元素	`lst.emplace_front(5); //` 在开头直接构造元素 5	O(1)
`emplace_back(Args&&... args)`	在末尾原位构造元素	`lst.emplace_back(25); //` 在末尾直接构造元素 25	O(1)
`resize(size_t count)`	调整链表大小	`lst.resize(10); //` 调整大小为 10 个元素	O(n)
`front()`	返回首元素的引用	`int first = lst.front(); //` 获取第一个元素	O(1)
`back()`	返回末元素的引用	`int last = lst.back(); //` 获取最后一个元素	O(1)
`assign(size_t count, const T& value)`	替换链表内容	`lst.assign(5, 1); //` 用 5 个值为 1 的元素替换内容	O(n)
`swap(list& other)`	与另一链表交换内容	`lst1.swap(lst2); //` 交换两个链表的内容	O(1)
`splice(iterator pos, list& other)`	将另一链表的所有元素移动到此链表	`lst1.splice(lst1.begin(), lst2); //` 将 lst2 的所有元素移到 lst1 开头	O(1)
`merge(list& other)`	合并两个有序链表	`lst1.merge(lst2); //` 将有序链表 lst2 合并入 lst1	O(n)
`unique()`	移除相邻重复元素	`lst.unique(); //` 删除连续重复的元素	O(n)
`sort()`	对链表进行排序	`lst.sort(); //` 按升序排序链表	O(n log n)
`reverse()`	反转链表	`lst.reverse(); //` 翻转链表元素顺序	O(n)

```cpp
// 多种构造方式
std::list<int> lst1;                        // 默认构造，空链表
std::list<int> lst2(5, 10);                 // 5 个值为 10 的元素
std::list<int> lst3 = {1, 2, 3, 4, 5};      // 使用初始化列表
std::list<int> lst4(lst3.begin(), std::next(lst3.begin(), 3)); // 使用迭代器范围构造
```

3.1.2.4 构造和初始化示例

```
// 前端操作
lst1.push_front(50);            // 在开头添加元素 50
lst1.emplace_front(40);         // 在开头原位构造元素 40
lst1.pop_front();               // 移除开头元素

// 后端操作
lst1.push_back(60);             // 在末尾添加元素 60
lst1.emplace_back(70);          // 在末尾原位构造元素 70
lst1.pop_back();                // 移除末尾元素
```

3.1.2.5 两端添加删除示例

```
// 中间位置操作
auto mid_it = std::next(lst1.begin(), lst1.size() / 2);  // 中间位置迭代器
lst1.insert(mid_it, 30);        // 在中间插入元素 30
lst1.erase(mid_it);             // 删除中间位置的元素
```

3.1.2.6 中间插入删除示例

```
// 访问元素
int first = lst1.front();       // 获取第一个元素
int last = lst1.back();         // 获取最后一个元素
int second = *(++lst1.begin()); // 访问第二个元素

// 遍历元素
for (auto it = lst1.begin(); it != lst1.end(); ++it) {
    std::cout << *it << " ";
}
// 使用范围 for 循环
for (const auto& elem : lst1) {
    std::cout << elem << " ";
}
```

3.1.2.7 访问元素示例

```
// 容量相关操作
size_t sz = lst1.size();        // 获取元素数量
bool is_empty = lst1.empty(); // 检查是否为空
lst1.resize(10);                // 调整大小为 10 个元素
lst1.resize(15, -1);            // 调整为 15 个元素，新元素用-1 填充
```

3.1.2.8 容量管理示例

```cpp
// 排序和反转
lst1.sort();                    // 升序排序
lst1.reverse();                 // 反转元素顺序

// 查找操作
auto find_it = std::find(lst1.begin(), lst1.end(), 30);
if (find_it != lst1.end()) {
    std::cout << " 找到元素 30" << std::endl;
}

// 拼接操作
std::list<int> lst5 = {100, 200, 300};
lst1.splice(lst1.end(), lst5);  // 将 lst5 拼接到 lst1 末尾
```

3.1.2.9 其他常用操作

3.1.2.10 list 与 vector/deque 的区别

- **内存分配**：list 使用节点分散存储，vector 和 deque 使用连续内存
- **访问方式**：list 只支持顺序访问，vector 和 deque 支持随机访问
- **插入删除**：list 在任意位置插入删除都是 $O(1)$，vector 和 deque 在中间位置插入删除是 $O(n)$
- **迭代器失效**：list 的迭代器在插入删除时不会失效，vector 和 deque 可能失效

```cpp
// 何时选择 list 而非 vector/deque?
// 1. 需要频繁在中间位置插入删除元素
// 2. 不需要随机访问元素
// 3. 需要稳定的迭代器
```

3.1.3 set-有序集合

核心特点：自动排序、唯一元素、基于红黑树

有序集合基于平衡二叉树（通常为红黑树）实现，所有操作均具有 $O(\log n)$ 的时间复杂度。

3.1.3.1 详细说明

- **内存管理**：元素在插入时自动排好序，内存占用相对较大
- **迭代器稳定性**：迭代器稳定性较好，删除一个元素仅使该元素的迭代器失效
- **异常安全**：所有操作均具有强异常安全性
- **性能特点**：插入删除查找 $O(\log n)$

自动排序，适合存储唯一且有序的元素。

3.1.3.2 使用场景

- **需要维护有序的唯一元素集合**：set 自动排序且不允许重复元素
- **需要快速查找元素**：set 的查找操作是 $O(\log n)$ 复杂度
- **需要范围查询功能**：set 提供 lower_bound 和 upper_bound 等接口

3.1.3.3 常用成员函数及使用说明

函数	描述	示例	复杂度
`insert(const T& value)`	插入元素	`s.insert(10); // 插入值 10`	$O(\log n)$
`erase(const T& value)`	删除指定元素	`s.erase(10); // 删除值 10`	$O(\log n)$
`find(const T& value)`	查找元素	`auto it = s.find(10); // 查找值 10`	$O(\log n)$

函数	描述	示例	复杂度
`count(const T& value)`	计算元素出现次数	`size_t cnt = s.count(10);` // 计算值 10 的出现次数	O(log n)
`size()`	获取元素个数	`size_t len = s.size();` // 获取元素数量	O(1)
`empty()`	检查是否为空	`if(s.empty()) {...}` // 判断容器是否为空	O(1)
`clear()`	清空所有元素	`s.clear();` // 移除所有元素	O(n)
`emplace(Args&&... args)`	原位插入元素	`s.emplace(15);` // 直接构造值为 15 的元素	O(log n)
`lower_bound(const T& key)`	返回第一个不小于 key 的迭代器	`auto it = s.lower_bound(10);` // 查找第一个不小于 10 的元素	O(log n)
`upper_bound(const T& key)`	返回第一个大于 key 的迭代器	`auto it = s.upper_bound(10);` // 查找第一个大于 10 的元素	O(log n)
`equal_range(const T& key)`	返回等于 key 的范围	`auto range = s.equal_range(10);` // 查找值 10 的范围	O(log n)
`swap(set& other)`	与另一容器交换内容	`s1.swap(s2);` // 交换两个 set 的内容	O(1)

```
// 多种构造方式
std::set<int> s1;                          // 默认构造，空集合
std::set<int> s2 = {1, 2, 3, 4, 5};     // 使用初始化列表
std::set<int> s3(s2.begin(), s2.end()); // 使用迭代器范围构造
```

3.1.3.4 构造和初始化示例

```
// 添加元素
s1.insert(10);              // 插入值 10
s1.emplace(15);             // 直接构造值为 15 的元素

// 删除元素
s1.erase(10);               // 删除值 10
s1.erase(s1.begin());       // 删除第一个元素
```

3.1.3.5 添加删除元素示例

```
// 查找元素
auto it = s1.find(15);      // 查找值 15
if (it != s1.end()) {
    std::cout << " 找到元素 15" << std::endl;
}

// 计算元素出现次数
size_t cnt = s1.count(15); // 计算值 15 的出现次数

// 范围查询
```

```cpp
auto lower = s1.lower_bound(10); // 查找第一个不小于 10 的元素
auto upper = s1.upper_bound(10); // 查找第一个大于 10 的元素
auto range = s1.equal_range(10); // 查找值 10 的范围
```

3.1.3.6 查找元素示例

```cpp

// 访问元素
for (auto it = s1.begin(); it != s1.end(); ++it) {
    std::cout << *it << " ";
}
// 使用范围 for 循环
for (const auto& elem : s1) {
    std::cout << elem << " ";
}
```

3.1.3.7 访问元素示例

```cpp

// 清空集合
s1.clear();                 // 移除所有元素

// 交换集合
s1.swap(s2);                // 交换 s1 和 s2 的内容
```

3.1.3.8 其他常用操作

3.1.3.9 set 与 unordered_set 的区别

- **内存分配**：set 使用红黑树，unordered_set 使用哈希表
- **访问方式**：set 自动排序，unordered_set 无序
- **查找效率**：set 的查找是 O(log n)，unordered_set 是 O(1)
- **迭代器失效**：set 的迭代器在插入删除时不会失效，unordered_set 可能失效

```cpp
// 何时选择 set 而非 unordered_set?
// 1. 需要有序存储元素
// 2. 需要范围查询功能
// 3. 需要稳定的迭代器
```

3.1.4 multiset-有序多重集合

允许存储重复的元素，并按照升序自动排序。使用场景：需要记录重复值且希望元素自动排序时。

```cpp
std::multiset<int> ms;
ms.insert(10);
ms.insert(10); // 重复元素允许
```

3.1.5 map-有序键值对

核心特点：键值对、自动排序、基于红黑树

有序映射和 set 类似，底层也使用红黑树。

3.1.5.1 详细说明

- **内存管理**：当使用 operator[] 进行元素访问时，如果键不存在，则会自动插入默认值，可能引起额外的内存分配和树重平衡
- **迭代器稳定性**：提供 lower_bound、upper_bound 等接口，便于高效的范围查询
- **异常安全**：所有操作均具有强异常安全性
- **性能特点**：插入删除查找 $O(\log n)$

用于建立键值对应关系，并保持键的排序。

3.1.5.2 使用场景

- **需要建立键到值的映射关系**：map 提供键值对存储
- **需要按键的顺序访问元素**：map 自动按键排序
- **需要高效的键值查找**：map 的查找操作是 $O(\log n)$ 复杂度

3.1.5.3 常用成员函数及使用说明

函数	描述	示例	复杂度
`insert(const std::pair<const Key, T>& value)`	插入键值对	`m.insert({3, "three"});` // 插入键值对 (3, “three”)	$O(\log n)$
`erase(const Key& key)`	删除指定键的元素	`m.erase(3);` // 删除键为 3 的元素	$O(\log n)$
`find(const Key& key)`	查找键对应的元素	`auto it = m.find(3);` // 查找键为 3 的元素	$O(\log n)$
`count(const Key& key)`	计算键出现次数	`size_t cnt = m.count(3);` // 计算键为 3 的出现次数	$O(\log n)$
`size()`	获取元素个数	`size_t len = m.size();` // 获取元素数量	$O(1)$
`empty()`	检查是否为空	`if(m.empty()) {...}` // 判断容器是否为空	$O(1)$
`clear()`	清空所有元素	`m.clear();` // 移除所有元素	$O(n)$
`operator[](const Key& key)`	访问或插入键对应的值	`m[5] = "five";` // 访问或插入键为 5 的值	$O(\log n)$
`emplace(Args&&... args)`	原位插入键值对	`m.emplace(4, "four");` // 直接构造键值对 (4, “four”)	$O(\log n)$
`lower_bound(const Key& key)`	返回第一个不小于 key 的迭代器	`auto it = m.lower_bound(3);` // 查找第一个不小于 3 的元素	$O(\log n)$
`upper_bound(const Key& key)`	返回第一个大于 key 的迭代器	`auto it = m.upper_bound(3);` // 查找第一个大于 3 的元素	$O(\log n)$
`equal_range(const Key& key)`	返回等于 key 的范围	`auto range = m.equal_range(3);` // 查找键为 3 的范围	$O(\log n)$
`swap(map& other)`	与另一容器交换内容	`m1.swap(m2);` // 交换两个 map 的内容	$O(1)$

```
// 多种构造方式
std::map<int, std::string> m1;                          // 默认构造，空映射
std::map<int, std::string> m2 = {{1, "one"}, {2, "two"}}; // 使用初始化列表
std::map<int, std::string> m3(m2.begin(), m2.end()); // 使用迭代器范围构造
```

3.1.5.4 构造和初始化示例

```
// 添加元素
m1.insert({3, "three"});      // 插入键值对 (3, "three")
m1.emplace(4, "four");        // 直接构造键值对 (4, "four")
m1[5] = "five";               // 访问或插入键为 5 的值

// 删除元素
m1.erase(3);                  // 删除键为 3 的元素
m1.erase(m1.begin());         // 删除第一个元素
```

3.1.5.5 添加删除元素示例

```
// 查找元素
auto it = m1.find(4);         // 查找键为 4 的元素
if (it != m1.end()) {
    std::cout << " 找到键 4 对应的值: " << it->second << std::endl;
}

// 计算键出现次数
size_t cnt = m1.count(4);     // 计算键为 4 的出现次数

// 范围查询
auto lower = m1.lower_bound(3); // 查找第一个不小于 3 的元素
auto upper = m1.upper_bound(3); // 查找第一个大于 3 的元素
auto range = m1.equal_range(3); // 查找键为 3 的范围
```

3.1.5.6 查找元素示例

```
// 访问元素
for (auto it = m1.begin(); it != m1.end(); ++it) {
    std::cout << it->first << ": " << it->second << " ";
}
// 使用范围 for 循环
for (const auto& elem : m1) {
    std::cout << elem.first << ": " << elem.second << " ";
}
```

3.1.5.7 访问元素示例

```
// 清空映射
m1.clear();                   // 移除所有元素

// 交换映射
m1.swap(m2);                  // 交换 m1 和 m2 的内容
```

3.1.5.8 其他常用操作

3.1.5.9 map 与 unordered_map 的区别

- **内存分配**：map 使用红黑树，unordered_map 使用哈希表
- **访问方式**：map 自动排序，unordered_map 无序

- **查找效率**：map 的查找是 $O(\log n)$，unordered_map 是 $O(1)$
- **迭代器失效**：map 的迭代器在插入删除时不会失效，unordered_map 可能失效

```
// 何时选择 map 而非 unordered_map?
// 1. 需要有序存储键值对
// 2. 需要范围查询功能
// 3. 需要稳定的迭代器
```

3.1.6 multimap-有序多重映射

允许相同键对应多个值，自动按键的升序排序。使用场景：需要建立一个键映射多个值时。

```
std::multimap<int, std::string> mm;
mm.insert({1, "one"});
mm.insert({1, "uno"}); // 同一键的重复插入
```

3.2 无序关联式容器

3.2.1 unordered_set-哈希集合

核心特点：基于哈希表、无序存储、快速查找

基于哈希表实现，所有查找、插入操作在平均情况下为 $O(1)$。

3.2.1.1 详细说明

- **内存管理**：元素无序存储，内存占用相对较小
- **迭代器稳定性**：内部采用开放地址或链地址法解决冲突，不同实现对内存和性能的要求不同
- **异常安全**：当负载因子过高时，rehash 操作会导致所有元素重新分布，迭代器也可能失效
- **性能特点**：插入删除查找平均 $O(1)$

优点：查找速度快，无须保持顺序。

3.2.1.2 使用场景

- **只需要判断元素是否存在**：unordered_set 提供高效的查找操作
- **不需要元素保持有序**：unordered_set 无序存储
- **追求最快的元素查找速度**：unordered_set 的查找操作是 $O(1)$ 复杂度

3.2.1.3 常用成员函数及使用说明

函数	描述	示例	复杂度
`insert(const T& value)`	插入元素	`us.insert(10); // 插入值 10`	平均 $O(1)$
`erase(const T& value)`	删除指定元素	`us.erase(10); // 删除值 10`	平均 $O(1)$
`find(const T& value)`	查找元素	`auto it = us.find(10); // 查找值 10`	平均 $O(1)$
`count(const T& value)`	计算元素出现次数	`size_t cnt = us.count(10); // 计算值 10 的出现次数`	平均 $O(1)$
`size()`	获取元素个数	`size_t len = us.size(); // 获取元素数量`	$O(1)$
`empty()`	检查是否为空	`if(us.empty()) {...} // 判断容器是否为空`	$O(1)$
`clear()`	清空所有元素	`us.clear(); // 移除所有元素`	$O(n)$
`emplace(Args&&... args)`	原位插入元素	`us.emplace(15); // 直接构造值为 15 的元素`	平均 $O(1)$

函数	描述	示例	复杂度
`bucket_count()`	返回当前哈希桶的数量	`size_t n = us.bucket_count(); // 获取哈希桶数量`	O(1)
`load_factor()`	返回当前负载因子	`float lf = us.load_factor(); // 获取负载因子`	O(1)
`rehash(size_t n)`	重新配置哈希桶数量	`us.rehash(20); // 重新配置哈希桶数量`	O(n)
`swap(unordered_set& other)`	与另一容器交换内容	`us1.swap(us2); // 交换两个 unordered_set 的内容`	O(1)

```
// 多种构造方式
std::unordered_set<int> us1;                          // 默认构造，空集合
std::unordered_set<int> us2 = {1, 2, 3, 4, 5};     // 使用初始化列表
std::unordered_set<int> us3(us2.begin(), us2.end()); // 使用迭代器范围构造
```

3.2.1.4 构造和初始化示例

```
// 添加元素
us1.insert(10);              // 插入值 10
us1.emplace(15);             // 直接构造值为 15 的元素

// 删除元素
us1.erase(10);               // 删除值 10
us1.erase(us1.begin());      // 删除第一个元素
```

3.2.1.5 添加删除元素示例

```
// 查找元素
auto it = us1.find(15);      // 查找值 15
if (it != us1.end()) {
    std::cout << " 找到元素 15" << std::endl;
}

// 计算元素出现次数
size_t cnt = us1.count(15); // 计算值 15 的出现次数
```

3.2.1.6 查找元素示例

```
// 访问元素
for (auto it = us1.begin(); it != us1.end(); ++it) {
    std::cout << *it << " ";
}
// 使用范围 for 循环
for (const auto& elem : us1) {
    std::cout << elem << " ";
}
```

3.2.1.7 访问元素示例

```cpp
// 清空集合
us1.clear();                    // 移除所有元素

// 交换集合
us1.swap(us2);                  // 交换 us1 和 us2 的内容

// 获取哈希桶数量
size_t n = us1.bucket_count(); // 获取哈希桶数量

// 获取负载因子
float lf = us1.load_factor();  // 获取负载因子

// 重新配置哈希桶数量
us1.rehash(20);                 // 重新配置哈希桶数量
```

3.2.1.8 其他常用操作

3.2.1.9 unordered_set 与 set 的区别

- **内存分配**：unordered_set 使用哈希表，set 使用红黑树
- **访问方式**：unordered_set 无序，set 自动排序
- **查找效率**：unordered_set 的查找是 $O(1)$，set 是 $O(\log n)$
- **迭代器失效**：unordered_set 的迭代器在插入删除时可能失效，set 不会失效

```cpp
// 何时选择 unordered_set 而非 set?
// 1. 只需要判断元素是否存在
// 2. 不需要元素保持有序
// 3. 追求最快的查找速度
```

3.2.2 unordered_map-哈希表

核心特点：基于哈希表、键值对、快速查找

详细说明与 unordered_set 类似，另外键值对的存储允许快速的键值映射。

3.2.2.1 详细说明

- **内存管理**：注意在使用 operator[] 时，如果键不存在会创建新项，可能带来意外的内存分配
- **迭代器稳定性**：内部采用开放地址或链地址法解决冲突，不同实现对内存和性能的要求不同
- **异常安全**：当负载因子过高时，rehash 操作会导致所有元素重新分布，迭代器也可能失效
- **性能特点**：插入删除查找平均 $O(1)$

常用于快速键值查找和数据插入。

3.2.2.2 使用场景

- **需要快速的键值查找**：unordered_map 提供高效的查找操作
- **不需要键值对保持有序**：unordered_map 无序存储
- **存储大量数据时需要高效查找**：unordered_map 的查找操作是 $O(1)$ 复杂度

3.2.2.3 常用成员函数及使用说明

函数	描述	示例	复杂度
`insert(const std::pair<const Key, T>& value)`	插入键值对	`um.insert({3, "three"});` // 插入键值对 (3, “three”)	平均 $O(1)$

函数	描述	示例	复杂度
`erase(const Key& key)`	删除指定键的元素	`um.erase(3); // 删除键为 3 的元素`	平均 $O(1)$
`find(const Key& key)`	查找键对应的元素	`auto it = um.find(3); // 查找键为 3 的元素`	平均 $O(1)$
`count(const Key& key)`	计算键出现次数	`size_t cnt = um.count(3); <br// 计算键为 3 的出现次数`	平均 $O(1)$
`size()`	获取元素个数	`size_t len = um.size(); <br// 获取元素数量`	$O(1)$
`empty()`	检查是否为空	`if(um.empty()) {...} <br// 判断容器是否为空`	$O(1)$
`clear()`	清空所有元素	`um.clear(); <br// 移除所有元素`	$O(n)$
`operator[](const Key& key)`	访问或插入键对应的值	`um[5] = "five"; <br// 访问或插入键为 5 的值`	平均 $O(1)$
`emplace(Args&&... args)`	原位插入键值对	`um.emplace(4, "four"); <br// 直接构造键值对 (4, “four”)`	平均 $O(1)$
`bucket_count()`	返回当前哈希桶的数量	`size_t n = um.bucket_count(); <br// 获取哈希桶数量`	$O(1)$
`load_factor()`	返回当前负载因子	`float lf = um.load_factor(); <br// 获取负载因子`	$O(1)$
`rehash(size_t n)`	重新配置哈希桶数量	`um.rehash(20); <br// 重新配置哈希桶数量`	$O(n)$
`swap(unordered_map& other)`	与另一容器交换内容	`um1.swap(um2); <br// 交换两个 unordered_map 的内容`	$O(1)$

```cpp
// 多种构造方式
std::unordered_map<int, std::string> um1;                          // 默认构造，空映射
std::unordered_map<int, std::string> um2 = {{1, "one"}, {2, "two"}}; // 使用初始化列表
std::unordered_map<int, std::string> um3(um2.begin(), um2.end()); // 使用迭代器范围构造
```

3.2.2.4 构造和初始化示例

```cpp
// 添加元素
um1.insert({3, "three"});      // 插入键值对 (3, "three")
um1.emplace(4, "four");        // 直接构造键值对 (4, "four")
um1[5] = "five";               // 访问或插入键为 5 的值

// 删除元素
um1.erase(3);                  // 删除键为 3 的元素
um1.erase(um1.begin());        // 删除第一个元素
```

3.2.2.5 添加删除元素示例

```cpp
// 查找元素
auto it = um1.find(4);          // 查找键为 4 的元素
if (it != um1.end()) {
    std::cout << " 找到键 4 对应的值: " << it->second << std::endl;
}

// 计算键出现次数
size_t cnt = um1.count(4);      // 计算键为 4 的出现次数
```

3.2.2.6 查找元素示例

```cpp
// 访问元素
for (auto it = um1.begin(); it != um1.end(); ++it) {
    std::cout << it->first << ": " << it->second << " ";
}
// 使用范围 for 循环
for (const auto& elem : um1) {
    std::cout << elem.first << ": " << elem.second << " ";
}
```

3.2.2.7 访问元素示例

```cpp
// 清空映射
um1.clear();                    // 移除所有元素

// 交换映射
um1.swap(um2);                  // 交换 um1 和 um2 的内容

// 获取哈希桶数量
size_t n = um1.bucket_count(); // 获取哈希桶数量

// 获取负载因子
float lf = um1.load_factor();  // 获取负载因子

// 重新配置哈希桶数量
um1.rehash(20);                 // 重新配置哈希桶数量
```

3.2.2.8 其他常用操作

3.2.2.9 unordered_map 与 map 的区别

- **内存分配**: unordered_map 使用哈希表，map 使用红黑树
- **访问方式**: unordered_map 无序，map 自动排序
- **查找效率**: unordered_map 的查找是 $O(1)$，map 是 $O(\log n)$
- **迭代器失效**: unordered_map 的迭代器在插入删除时可能失效，map 不会失效

```cpp
// 何时选择 unordered_map 而非 map?
// 1. 需要快速的键值查找
// 2. 不需要键值对保持有序
// 3. 存储大量数据时需要高效查找
```

3.3 容器适配器

3.3.1 queue-队列

核心特点：FIFO 结构、基于 deque 实现、只允许访问队首和队尾

队列是一种 FIFO 数据结构，内部一般使用 deque 实现。

3.3.1.1 详细说明

- **内存管理**：由于只允许访问队首和队尾，不支持迭代器，因此无法直接对队列内部数据做复杂操作
- **迭代器稳定性**：在多线程环境中，队列必须进行额外的同步来保证数据的一致性
- **异常安全**：所有操作均具有强异常安全性
- **性能特点**：插入删除 $O(1)$，访问队首队尾 $O(1)$

先进先出（FIFO）结构，常用于任务调度及广度优先搜索。

3.3.1.2 使用场景

- **需要按照先进先出的顺序处理元素**：queue 提供 FIFO 结构
- **实现广度优先搜索**：queue 常用于 BFS 算法
- **需要按照时间顺序处理任务**：queue 适合任务调度

3.3.1.3 常用成员函数及使用说明

函数	描述	示例	复杂度
`push(const T& value)`	添加元素到队列末尾	`q.push(10);` <br// 在末尾添加值 10	$O(1)$
`pop()`	移除队列前端元素	`q.pop();` <br// 删除第一个元素	$O(1)$
`front()`	访问队列前端元素	`int front = q.front();` <br// 获取第一个元素	$O(1)$
`back()`	访问队列末尾元素	`int back = q.back();` <br// 获取最后一个元素	$O(1)$
`empty()`	检查是否为空	`if(q.empty()) {...}` <br// 判断容器是否为空	$O(1)$
`size()`	获取元素个数	`size_t len = q.size();` <br// 获取元素数量	$O(1)$
`emplace(Args&&... args)`	原位添加元素到队列末尾	`q.emplace(15);` <br// 直接构造值为 15 的元素	$O(1)$
`swap(queue& other)`	与另一容器交换内容	`q1.swap(q2);` <br// 交换两个 queue 的内容	$O(1)$

```
// 多种构造方式
std::queue<int> q1;                    // 默认构造，空队列
std::queue<int> q2;                    // 使用默认构造
q2.push(1);                            // 添加元素
q2.push(2);                            // 添加元素
```

3.3.1.4 构造和初始化示例

```
// 添加元素
q1.push(10);              // 在末尾添加值 10
q1.emplace(15);           // 直接构造值为 15 的元素
```

```
// 删除元素
q1.pop();                    // 删除第一个元素
```

3.3.1.5 添加删除元素示例

```
// 访问元素
int front = q1.front();  // 获取第一个元素
int back = q1.back();    // 获取最后一个元素
```

3.3.1.6 访问元素示例

```
// 容量相关操作
size_t sz = q1.size();   // 获取元素数量
bool is_empty = q1.empty(); // 检查是否为空
```

3.3.1.7 容量管理示例

```
// 交换队列
q1.swap(q2);                 // 交换 q1 和 q2 的内容
```

3.3.1.8 其他常用操作

3.3.1.9 queue 与 deque 的区别

- **内存分配**：queue 使用 deque 作为底层容器
- **访问方式**：queue 只允许访问队首和队尾，deque 支持随机访问
- **插入删除**：queue 只允许在队首和队尾插入删除，deque 支持在任意位置插入删除
- **迭代器失效**：queue 不支持迭代器，deque 支持迭代器

```
// 何时选择 queue 而非 deque?
// 1. 需要先进先出的顺序处理元素
// 2. 不需要随机访问元素
// 3. 需要简单的接口
```

3.3.2 priority_queue-优先队列

核心特点：基于堆、默认最大堆、支持自定义比较器

优先队列基于堆实现，默认是最大堆。

3.3.2.1 详细说明

- **内存管理**：内部维护堆结构，使得每次获取 top 操作都是 $O(1)$，而插入和删除操作都是 $O(\log n)$
- **迭代器稳定性**：可通过自定义比较器改变排序规则，例如实现最小堆
- **异常安全**：所有操作均具有强异常安全性
- **性能特点**：插入删除 $O(\log n)$，访问 top $O(1)$

默认实现最大堆，可根据需定制排序规则。

3.3.2.2 使用场景

- **需要维护一个动态的有序序列**：priority_queue 提供堆结构
- **实现堆排序**：priority_queue 常用于堆排序算法
- **需要频繁获取最大/最小元素**：priority_queue 提供高效的 top 操作

3.3.2.3 常用成员函数及使用说明

函数	描述	示例	复杂度
`push(const T& value)`	添加元素	`pq.push(10);` <br// 添加值 10	O(log n)
`pop()`	移除优先级最高的元素	`pq.pop();` <br// 删除优先级最高的元素	O(log n)
`top()`	访问优先级最高的元素	`int top = pq.top();` <br// 获取优先级最高的元素	O(1)
`empty()`	检查是否为空	`if(pq.empty()) {...}` <br// 判断容器是否为空	O(1)
`size()`	获取元素个数	`size_t len = pq.size();` <br// 获取元素数量	O(1)
`emplace(Args&&... args)`	原位添加元素	`pq.emplace(15);` <br// 直接构造值为 15 的元素	O(log n)
`swap(priority_queue& other)`	与另一容器交换内容	`pq1.swap(pq2);` <br// 交换两个 priority_queue 的内容	O(1)

```
// 多种构造方式
std::priority_queue<int> pq1;                    // 默认构造，空优先队列
std::priority_queue<int> pq2;                    // 使用默认构造
pq2.push(1);                                     // 添加元素
pq2.push(2);                                     // 添加元素
```

3.3.2.4 构造和初始化示例

```
// 添加元素
pq1.push(10);              // 添加值 10
pq1.emplace(15);           // 直接构造值为 15 的元素

// 删除元素
pq1.pop();                 // 删除优先级最高的元素
```

3.3.2.5 添加删除元素示例

```
// 访问元素
int top = pq1.top();       // 获取优先级最高的元素
```

3.3.2.6 访问元素示例

```
// 容量相关操作
size_t sz = pq1.size();    // 获取元素数量
bool is_empty = pq1.empty(); // 检查是否为空
```

3.3.2.7 容量管理示例

```
// 交换优先队列
pq1.swap(pq2);              // 交换 pq1 和 pq2 的内容
```

3.3.2.8 其他常用操作

3.3.2.9 priority_queue 与 queue 的区别

- **内存分配**：priority_queue 使用堆结构，queue 使用 deque 作为底层容器
- **访问方式**：priority_queue 只允许访问优先级最高的元素，queue 只允许访问队首和队尾
- **插入删除**：priority_queue 的插入删除是 $O(\log n)$，queue 的插入删除是 $O(1)$
- **迭代器失效**：priority_queue 不支持迭代器，queue 不支持迭代器

```
// 何时选择 priority_queue 而非 queue?
// 1. 需要维护一个动态的有序序列
// 2. 需要频繁获取最大/最小元素
// 3. 需要实现堆排序
```

3.3.3 stack-栈

核心特点：LIFO 结构、基于 deque 实现、只允许访问栈顶

栈是一种 LIFO 数据结构，通常使用 deque 实现。

3.3.3.1 详细说明

- **内存管理**：栈中的所有操作均为常数时间，但由于不支持迭代器，其调试和错误检查较为困难
- **迭代器稳定性**：主要适用于递归调用、表达式求值等场景
- **异常安全**：所有操作均具有强异常安全性
- **性能特点**：插入删除 $O(1)$，访问栈顶 $O(1)$

后进先出 (LIFO) 结构，适合实现深度优先搜索或函数调用栈。

3.3.3.2 使用场景

- **需要按照后进先出的顺序处理元素**：stack 提供 LIFO 结构
- **实现深度优先搜索**：stack 常用于 DFS 算法
- **处理括号匹配等问题**：stack 适合括号匹配等问题
- **实现函数调用栈**：stack 适合函数调用栈

3.3.3.3 常用成员函数及使用说明

函数	描述	示例	复杂度
`push(const T& value)`	添加元素到栈顶	`s.push(10);` <br// 在栈顶添加值 10	$O(1)$
`pop()`	移除栈顶元素	`s.pop();` <br// 删除栈顶元素	$O(1)$
`top()`	访问栈顶元素	`int top = s.top();` <br// 获取栈顶元素	$O(1)$
`empty()`	检查是否为空	`if(s.empty()) {...}` <br// 判断容器是否为空	$O(1)$
`size()`	获取元素个数	`size_t len = s.size();` <br// 获取元素数量	$O(1)$
`emplace(Args&&... args)`	原位添加元素到栈顶	`s.emplace(15);` <br// 直接构造值为 15 的元素	$O(1)$
`swap(stack& other)`	与另一容器交换内容	`s1.swap(s2);` <br// 交换两个 stack 的内容	$O(1)$

```cpp
// 多种构造方式
std::stack<int> s1;                         // 默认构造，空栈
std::stack<int> s2;                         // 使用默认构造
s2.push(1);                                 // 添加元素
s2.push(2);                                 // 添加元素
```

3.3.3.4 构造和初始化示例

```cpp
// 添加元素
s1.push(10);              // 在栈顶添加值 10
s1.emplace(15);           // 直接构造值为 15 的元素

// 删除元素
s1.pop();                 // 删除栈顶元素
```

3.3.3.5 添加删除元素示例

```cpp
// 访问元素
int top = s1.top();       // 获取栈顶元素
```

3.3.3.6 访问元素示例

```cpp
// 容量相关操作
size_t sz = s1.size();    // 获取元素数量
bool is_empty = s1.empty(); // 检查是否为空
```

3.3.3.7 容量管理示例

```cpp
// 交换栈
s1.swap(s2);              // 交换 s1 和 s2 的内容
```

3.3.3.8 其他常用操作

3.3.3.9 stack 与 deque 的区别

- **内存分配**：stack 使用 deque 作为底层容器
- **访问方式**：stack 只允许访问栈顶元素，deque 支持随机访问
- **插入删除**：stack 只允许在栈顶插入删除，deque 支持在任意位置插入删除
- **迭代器失效**：stack 不支持迭代器，deque 支持迭代器

```cpp
// 何时选择 stack 而非 deque?
// 1. 需要后进先出的顺序处理元素
// 2. 不需要随机访问元素
// 3. 需要简单的接口
```

3.4 位集合

3.4.1 bitset-固定大小位容器

核心特点：固定大小、位操作高效、适合布尔标志存储

用于存储固定数量的二进制位，支持高效位运算。

3.4.1.1 详细说明

- **内存管理**：因为固定大小，bitset 的操作在编译期间就确定，性能和内存使用非常优化
- **迭代器稳定性**：在处理大批量布尔值数据时，比 vector 更加高效且易于理解
- **异常安全**：所有操作均具有强异常安全性
- **性能特点**：位操作 $O(1)$

用于存储固定数量的二进制位，并支持高效的位操作。

3.4.1.2 使用场景

- **需要按位处理或存储大量布尔标志**：bitset 提供高效的位操作

3.4.1.3 常用成员函数及使用说明

函数	描述	示例	复杂度
`set()`	将所有位或指定位置的位设置为 1	`bs.set();` <br// 将所有位设置为 1	O(1)
`reset()`	将所有位或指定位置的位重置为 0	`bs.reset();` <br// 将所有位重置为 0	O(1)
`flip()`	将所有位或指定位置的位取反	`bs.flip();` <br// 将所有位取反	O(1)
`test(size_t pos)`	检查指定位置的位是否为 1	`bool bit3 = bs.test(3);` <br// 检查第 3 位是否为 1	O(1)
`all()`	检查所有位是否都为 1	`bool all = bs.all();` <br// 检查所有位是否都为 1	O(1)
`any()`	检查是否存在为 1 的位	`bool any = bs.any();` <br// 检查是否存在为 1 的位	O(1)
`none()`	检查是否所有位都为 0	`bool none = bs.none();` <br// 检查是否所有位都为 0	O(1)
`count()`	返回为 1 的位的个数	`size_t cnt = bs.count();` <br// 返回为 1 的位的个数	O(1)
`size()`	返回位的总数	`size_t sz = bs.size();` <br// 返回位的总数	O(1)
`to_string()`	返回位的字符串表示	`std::string str = bs.to_string();` <br// 返回位的字符串表示	O(n)
`to_ulong()`	返回位的无符号长整型表示	`unsigned long ul = bs.to_ulong();` <br// 返回位的无符号长整型表示	O(1)
`to_ullong()`	返回位的无符号长长整型表示	`unsigned long long ull = bs.to_ullong();` <br// 返回位的无符号长长整型表示	O(1)

```cpp
// 多种构造方式
std::bitset<8> bs1;                     // 默认构造，空位集合
std::bitset<8> bs2(0b10101010);         // 使用二进制字面量
std::bitset<8> bs3("1100");             // 使用字符串
```

3.4.1.4 构造和初始化示例

```
// 设置和重置位
bs1.set();                  // 将所有位设置为 1
bs1.reset();                // 将所有位重置为 0
bs1.set(3);                 // 将第 3 位置为 1
bs1.reset(3);               // 将第 3 位重置为 0
bs1.flip();                 // 将所有位取反
bs1.flip(2);                // 将第 2 位取反
```

3.4.1.5 设置和重置位示例

```
// 检查位
bool bit3 = bs1.test(3);  // 检查第 3 位是否为 1
bool all = bs1.all();     // 检查所有位是否都为 1
bool any = bs1.any();     // 检查是否存在为 1 的位
bool none = bs1.none();   // 检查是否所有位都为 0
size_t cnt = bs1.count(); // 返回为 1 的位的个数
size_t sz = bs1.size();   // 返回位的总数
```

3.4.1.6 检查位示例

```
// 转换为字符串和整数
std::string str = bs1.to_string(); // 返回位的字符串表示
unsigned long ul = bs1.to_ulong(); // 返回位的无符号长整型表示
unsigned long long ull = bs1.to_ullong(); // 返回位的无符号长长整型表示
```

3.4.1.7 转换为字符串和整数示例

```
std::bitset<4> bs(0b1010); // 表示：第 3 位为 1，第 2 位为 0，第 1 位为 1，第 0 位为 0
// 例如：测试最高位
bool highest = bs.test(3); // 返回 true
```

3.4.1.8 示例：bitset 默认索引从右到左（低位到高位）

3.5 性能对比

容器类型	随机访问	插入/删除（开始）	插入/删除（中间）	插入/删除（结尾）	查找
vector	O(1)	O(n)	O(n)*	O(1)	O(n)
deque	O(1)	O(1)	O(n)*	O(1)	O(n)
list	O(n)	O(1)	O(n)（遍历定位 + O(1)）	O(1)	O(n)
set	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
map	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
unordered_set	N/A	O(1)	O(1)	O(1)	O(1)
unordered_map	N/A	O(1)	O(1)	O(1)	O(1)

* 注：中间插入/删除需要先遍历定位，list 操作定位为 O(n)

3.6 空间复杂度

容器类型	额外空间
vector	O(n)
deque	O(n)
list	O(n)
set	O(n)
map	O(n)
unordered_set	O(n)
unordered_map	O(n)

3.7 最佳实践

1. 默认优先使用 vector
2. 需要在两端操作时使用 deque
3. 频繁插入删除使用 list
4. 需要有序唯一元素使用 set
5. 需要键值对使用 map
6. 追求查找效率使用 unordered_set/map
7. 需要 FIFO 使用 queue
8. 需要维护有序序列使用 priority_queue

补充说明

- 使用 emplace 系列函数可以避免构造临时对象，从而提高性能。
- 对于可能抛出异常的操作（如 vector::at()、map::at()），建议使用 try-catch 捕获异常。

3.8 容器适用场景对比

容器类型	内存占用	查找效率	插入/删除效率	是否有序	允许重复元素
vector	较低	O(n)	尾部 O(1)，中间 O(n)	是	否
deque	较低	O(n)	两端 O(1)，中间 O(n)	是	否
list	较高	O(n)	O(1)	否	是
set	中等	O(log n)	O(log n)	是	否
multiset	中等	O(log n)	O(log n)	是	是
map	中等	O(log n)	O(log n)	是	键唯一
multimap	中等	O(log n)	O(log n)	是	键可重复
unordered_set	较低	平均 O(1)	平均 O(1)	否	否
unordered_map	较低	平均 O(1)	平均 O(1)	否	键唯一

3.9 迭代器失效说明

在 **vector** 和 **deque** 中，插入或删除元素可能导致迭代器失效：
详细说明：

- 对 vector：插入或删除元素可能导致重新分配内存，从而使所有迭代器失效。建议使用返回值或重新获取迭代器。
- 对 deque：插入/删除头尾元素通常安全，但中间操作可能使部分迭代器失效。

示例：

```
// 避免迭代器失效的示例
auto it = vec.begin();
it = vec.insert(it, 42); // 使用返回的迭代器继续操作
```

3.10 std::array 的说明

虽然 std::array 大小固定，但适用于需要在栈上分配小型固定数组的场景，性能较高且安全：

```
#include <array>
std::array<int, 5> arr = {1, 2, 3, 4, 5};
```

3.11 查找操作详解

详细查找操作部分增加对异常安全性、底层实现（例如哈希桶分布、红黑树平衡规则）的解释，帮助你更好地选择合适的查找算法和数据结构进行优化。

3.11.1 顺序容器查找

```
std::vector<int> vec = {1, 2, 3, 4, 5};

// 1. 使用 find 算法（需要包含 <algorithm>）
auto it = std::find(vec.begin(), vec.end(), 3);
if (it != vec.end()) {
    std::cout << " 找到元素： " << *it << "\n";
} else {
    std::cout << " 未找到元素\n";
}

// 2. 使用 find_if 查找满足条件的元素
auto it2 = std::find_if(vec.begin(), vec.end(), [](int x) {
    return x > 3;
});
if (it2 != vec.end()) {
    std::cout << " 找到第一个大于 3 的元素： " << *it2 << "\n";
}

// 3. 二分查找（要求容器已排序）
if (std::binary_search(vec.begin(), vec.end(), 3)) {
    std::cout << " 元素 3 存在\n";
}

// 4. 获取位置
auto idx = std::distance(vec.begin(), it);
std::cout << " 元素位置： " << idx << "\n";
```

3.11.1.1 vector/deque 的查找

```
std::list<int> lst = {1, 2, 3, 4, 5};

// 由于 list 不支持随机访问，通常使用迭代器
auto it = std::find(lst.begin(), lst.end(), 3);
if (it != lst.end()) {
    std::cout << " 找到元素： " << *it << "\n";
    // 前移/后移迭代器
    ++it; // 下一个元素
    --it; // 上一个元素
}
```

3.11.1.2 list 的查找

3.11.2 关联容器查找

```
std::set<int> s = {1, 2, 3, 4, 5};

// 1. 使用 find 成员函数
auto it = s.find(3);
if (it != s.end()) {
    std::cout << " 找到元素: " << *it << "\n";
}

// 2. 使用 count 检查元素是否存在（set 中要么是 0 要么是 1）
if (s.count(3) > 0) {
    std::cout << " 元素 3 存在\n";
}

// 3. 使用 lower_bound 和 upper_bound
auto lower = s.lower_bound(3); // 大于等于 3 的第一个元素
auto upper = s.upper_bound(3); // 大于 3 的第一个元素

if (lower != s.end()) {
    std::cout << " 第一个大于等于 3 的元素: " << *lower << "\n";
}

// 4. 使用 equal_range 获取范围
auto range = s.equal_range(3);
for (auto it = range.first; it != range.second; ++it) {
    std::cout << *it << " ";
}
```

3.11.2.1 set 的查找

```
std::map<std::string, int> m = {
    {"one", 1}, {"two", 2}, {"three", 3}
};

// 1. 使用 find 成员函数
auto it = m.find("two");
if (it != m.end()) {
    std::cout << " 找到键 two 对应的值: " << it->second << "\n";
}

// 2. 使用 count 检查键是否存在
if (m.count("two") > 0) {
    std::cout << " 键 two 存在\n";
}

// 3. 使用 at 访问（不存在则抛出异常）
try {
    int value = m.at("two");
    std::cout << " 键 two 的值: " << value << "\n";
} catch (const std::out_of_range& e) {
    std::cout << " 键不存在\n";
}
```

```cpp
// 4. 使用 operator[] （不存在则插入默认值）
int value = m["two"]; // 如果不存在，会插入并初始化为 0

// 5. 使用 lower_bound 和 upper_bound（基于键排序）
auto lower = m.lower_bound("two");
if (lower != m.end()) {
    std::cout << " 第一个大于等于 two 的键值对: "
              << lower->first << ":" << lower->second << "\n";
}
```

3.11.2.2 map 的查找

3.11.3 无序容器查找

```cpp
std::unordered_set<int> us = {1, 2, 3, 4, 5};

// 1. 使用 find 成员函数
auto it = us.find(3);
if (it != us.end()) {
    std::cout << " 找到元素: " << *it << "\n";
}

// 2. 使用 count 检查元素是否存在
if (us.count(3) > 0) {
    std::cout << " 元素 3 存在\n";
}

// 注意：无序容器不支持 lower_bound/upper_bound
```

3.11.3.1 unordered_set 的查找

```cpp
std::unordered_map<std::string, int> um = {
    {"one", 1}, {"two", 2}, {"three", 3}
};

// 1. 使用 find 成员函数
auto it = um.find("two");
if (it != um.end()) {
    std::cout << " 找到键 two 对应的值: " << it->second << "\n";
}

// 2. 使用 count 检查键是否存在
if (um.count("two") > 0) {
    std::cout << " 键 two 存在\n";
}

// 3. 安全的值访问模式
auto it2 = um.find("two");
if (it2 != um.end()) {
    std::cout << " 值: " << it2->second << "\n";
} else {
    std::cout << " 键不存在\n";
}
```

```cpp
// 4. 使用 at（不存在则抛出异常）
try {
    int value = um.at("two");
} catch (const std::out_of_range& e) {
    std::cout << " 键不存在\n";
}
```

3.11.3.2 unordered_map 的查找

3.11.4 容器适配器的查找

注意：stack、queue 和 priority_queue 不支持直接查找，只能访问特定位置的元素（如栈顶、队首等）

```cpp
std::stack<int> s;
s.push(1);
s.push(2);
s.push(3);

// 只能访问栈顶元素
if (!s.empty()) {
    std::cout << " 栈顶元素: " << s.top() << "\n";
}

std::queue<int> q;
q.push(1);
q.push(2);
q.push(3);

// 只能访问队首和队尾元素
if (!q.empty()) {
    std::cout << " 队首元素: " << q.front() << "\n";
    std::cout << " 队尾元素: " << q.back() << "\n";
}

std::priority_queue<int> pq;
pq.push(1);
pq.push(3);
pq.push(2);

// 只能访问最高优先级的元素（默认最大值）
if (!pq.empty()) {
    std::cout << " 最高优先级元素: " << pq.top() << "\n";
}
```

3.11.5 查找最佳实践

1. 对于有序序列，优先使用二分查找（binary_search, lower_bound, upper_bound）
2. 对于关联容器，优先使用成员函数 find 而不是算法 std::find
3. 需要频繁查找时，考虑使用哈希容器（unordered_set/map）
4. 需要查找范围时，使用 equal_range
5. 对于 map/unordered_map，优先使用 find 而不是 operator[] 检查键是否存在

3.12 扩展说明及用法

3.12.1 1. STL 各容器的选择建议

- 对于需要动态扩展且随机访问要求高的场景，尽可能使用 vector。
- 当两端插入删除频繁时，优先考虑 deque；而中间频繁插入删除建议使用 list。

- 对于需要快速查询，且数据无须顺序时，可选择 unordered_set 或 unordered_map。

3.12.2 2. 详细使用示例

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    vector<int> vec = {5, 3, 8, 1};
    // 排序并查找第一个大于 4 的元素
    sort(vec.begin(), vec.end());
    auto it = find_if(vec.begin(), vec.end(), [](int n){ return n > 4; });
    if(it != vec.end()){
        cout << " 第一个大于 4 的元素是: " << *it << endl;
    }
    return 0;
}
```

3.12.2.1 示例：使用 vector 与算法配合

3.12.3 3. 扩展用法说明

- 为提高代码复用性，建议将常用操作封装为函数或类模板。
- 利用 Lambda 表达式进行复杂条件查找，是 C++11 后的常用技巧。
- STL 算法与容器的灵活组合，能够大幅简化代码且提高性能，应多加实践。

3.12.4 4. 迭代器类别

- **vector / deque**：随机访问迭代器（支持 +、-、[] 运算符）
- **list / set / map**：双向迭代器（仅支持 ++ 和--）
- **forward_list**：前向迭代器（仅支持 ++）

C++ STL 中的迭代器按照功能强大程度可以分为以下几类：

1. 随机访问迭代器
    - 适用容器：vector、deque
    - 支持的操作：

```
it + n;       // 向前移动 n 个位置
it - n;       // 向后移动 n 个位置
it[n];        // 访问第 n 个元素
it1 < it2;    // 迭代器比较
```

2. 双向迭代器
    - 适用容器：list、set、map
    - 支持的操作：

```
++it;     // 向前移动一个位置
--it;     // 向后移动一个位置
*it;      // 解引用访问元素
```

3. 前向迭代器
    - 适用容器：forward_list
    - 支持的操作：

```
++it;     // 只能向前移动
*it;      // 解引用访问元素
```

```
// 随机访问迭代器示例
std::vector<int> vec = {1, 2, 3, 4, 5};
auto it = vec.begin();
it += 2;       // 可以直接跳转
std::cout << it[1] << std::endl;  // 可以用 [] 访问

// 双向迭代器示例
std::list<int> lst = {1, 2, 3, 4, 5};
auto it2 = lst.begin();
++it2;     // 只能一步一步移动
--it2;     // 可以向后移动

// 前向迭代器示例
std::forward_list<int> fl = {1, 2, 3, 4, 5};
auto it3 = fl.begin();
++it3;     // 只能向前移动
```

3.12.4.1 使用示例

4 STL 常用函数

4.1 算法相关函数

使用前需要包含头文件 `#include <algorithm>`

4.1.1 排序与查找

```
// 对容器内的元素进行排序
// 返回值：无
// 参数说明：
// - first: 指向要排序范围首元素的迭代器
// - last: 指向要排序范围末尾的下一个元素的迭代器
// - comp: 比较函数（可选），接收两个元素并返回 bool 值，表示第一个参数是否应排在第二个参数前面
sort(first, last, comp);
```

4.1.1.1 sort

参数详解：- `first`：指向要排序范围首元素的迭代器。必须是随机访问迭代器 - `last`：指向要排序范围末尾的下一个元素的迭代器（即排序区间为 [first, last)) - `comp`：比较函数（可选），是一个返回 bool 值的二元谓词，形式为 `bool cmp(const Type1 &a, const Type2 &b)` - 当 `comp(a, b)` 返回 true 时，a 会排在 b 前面 - 如果不提供，默认使用 `operator<` 进行升序排序 - 可以是函数指针、函数对象或 Lambda 表达式

返回值：无。排序会直接修改原容器中的元素顺序。

时间复杂度：$O(N \log N)$，其中 N 为排序元素的个数。实际实现通常使用快速排序、堆排序和插入排序的混合算法，在最坏情况下保证 $O(N \log N)$。

空间复杂度：$O(\log N)$，用于递归调用栈。

使用场景：- 需要对数组、vector 等随机访问容器进行快速排序 - 可以用于基本数据类型和自定义类型的排序 - 当需要稳定排序（保持相等元素的相对顺序）时，应使用 `stable_sort` 而非 `sort`

示例：

```
vector<int> v = {5, 2, 8, 1, 9};
// 升序排序 - sort 将 v 中的元素按从小到大排序
// 调用后 v 中的元素会被重新排列为 {1, 2, 5, 8, 9}
sort(v.begin(), v.end());
// 结果: {1, 2, 5, 8, 9}
```

```cpp
// 降序排序 - 使用 greater<int>() 作为比较函数
// greater<int>() 是一个函数对象，当 a>b 时返回 true，实现从大到小排序
sort(v.begin(), v.end(), greater<int>());
// 结果：{9, 8, 5, 2, 1}

// 使用自定义比较函数 - lambda 表达式实现降序排序
// [](int a, int b) { return a > b; } 定义了一个匿名函数，当 a>b 时返回 true
// 这使得排序算法将较大的元素放在前面，实现降序排序
sort(v.begin(), v.end(), [](int a, int b) {
    return a > b;  // 降序排序：当 a>b 时返回 true，表示 a 应排在 b 前面
});

// 结构体排序 - 根据 Person 的 age 字段升序排序
struct Person {
    string name;
    int age;
};
vector<Person> people = {{"Alice", 25}, {"Bob", 20}, {"Charlie", 30}};
// 使用 Lambda 表达式作为比较函数，比较两个 Person 对象的 age 字段
// 当 a.age < b.age 时返回 true，这样年龄小的 Person 将排在前面
sort(people.begin(), people.end(), [](const Person& a, const Person& b) {
    return a.age < b.age;  // 按年龄升序排序
});

// 多字段排序 - 先按年龄升序，年龄相同时按姓名字母顺序排序
sort(people.begin(), people.end(), [](const Person& a, const Person& b) {
    if (a.age != b.age) return a.age < b.age;  // 主要排序键：年龄
    return a.name < b.name;  // 次要排序键：姓名
});
```

错误处理：- 如果提供的迭代器不是随机访问迭代器，将导致编译错误 - 比较函数必须提供严格弱序关系，否则行为未定义 - 使用有问题的比较函数（如不满足传递性）可能导致排序结果不确定

注意事项：- `sort` 是不稳定的排序算法，不保证相等元素的相对顺序 - 比较函数必须定义**严格弱序**关系，即满足以下性质：- 非自反性：对于任意 x，不能有 $x<x$ 为真 - 非对称性：如果 $x<y$ 为真，则 $y<x$ 为假 - 传递性：如果 $x<y$ 为真且 $y<z$ 为真，则 $x<z$ 为真 - 只适用于随机访问迭代器的容器，如 vector、array、deque 等 - 对于 list 等非随机访问的容器，应使用其成员函数 sort - 在处理大型对象时，尽量使用引用作为比较函数的参数类型，避免不必要的拷贝

```cpp
// 稳定排序，保持相等元素的相对顺序
// 返回值：无
// 参数说明：
// - first: 指向要排序范围首元素的迭代器
// - last: 指向要排序范围末尾的下一个元素的迭代器
// - comp: 比较函数（可选），接收两个元素并返回 bool 值
stable_sort(first, last, comp);
```

4.1.1.2 stable_sort

参数和用法同 `sort`，但保证相等元素的相对顺序不变。

时间复杂度：- $O(N \log N)$（如果有足够额外内存）- $O(N (\log N)^2)$（在最坏情况下）

示例：

```cpp
struct Student {
    string name;
    int grade;
};
vector<Student> students = {
    {"Alice", 90}, {"Bob", 85}, {"Charlie", 90}, {"David", 85}
```

```
};

// 按成绩降序排序，成绩相同时保持原有顺序
// 当 grade 相同时，stable_sort 保证了元素的相对顺序不变
// 例如，Alice 和 Charlie 的 grade 都是 90，排序后 Alice 仍在 Charlie 之前
stable_sort(students.begin(), students.end(), [](const Student& a, const Student& b) {
    return a.grade > b.grade;
});
// 结果：[{"Alice", 90}, {"Charlie", 90}, {"Bob", 85}, {"David", 85}]
```

```
// 将范围中最小的 N 个元素排序放在开头
// 返回值：指向排好序部分末尾的迭代器（即 middle）
// 参数说明：
// - first: 范围起始迭代器
// - middle: 部分排序结束位置（排序 [first,middle) 区间）
// - last: 范围结束迭代器
// - comp: 比较函数（可选）
partial_sort(first, middle, last, comp);
```

4.1.1.3 partial_sort

参数说明：- `first`：范围起始迭代器 - `middle`：部分排序结束位置（排序 $[\text{first},\text{middle})$ 区间）- `last`：范围结束迭代器 - `comp`：比较函数（可选）

时间复杂度：$O(N \log M)$，其中 N 是范围大小，M 是要排序的元素个数

使用场景：当只需要前 k 小/大元素时，比完全排序更高效

示例：

```
vector<int> v = {5, 2, 8, 1, 9, 3, 7, 6, 4};
// 对前 5 个最小元素排序
// 此操作将 v 中最小的 5 个元素（即 1,2,3,4,5）排序并放在 v 的前 5 个位置
// v.begin() + 5 表示排序截止的位置，v.end() 表示要考虑的全部元素范围
partial_sort(v.begin(), v.begin() + 5, v.end());
// 可能结果：{1, 2, 3, 4, 5, ?, ?, ?, ?}，其中? 表示未排序的元素
// 注意：后面的元素不保证任何特定顺序，只确保它们都大于或等于前 5 个元素
```

```
// 将第 n 小的元素放在正确位置，并将小于它的元素放在它前面，大于它的放在后面
// 返回值：无
// 参数说明：
// - first: 范围起始迭代器
// - nth: 指向要定位的元素的迭代器
// - last: 范围结束迭代器
// - comp: 比较函数（可选）
nth_element(first, nth, last, comp);
```

4.1.1.4 nth_element

参数说明：- `first`：范围起始迭代器 - `nth`：指向要定位的元素的迭代器 - `last`：范围结束迭代器 - `comp`：比较函数（可选）

时间复杂度：平均 $O(N)$，最坏 $O(N^2)$

使用场景：- 查找中位数、分位数 - 需要将数组划分为两部分，但不需要完全排序

示例：

```
vector<int> v = {5, 2, 8, 1, 9, 3, 7, 6, 4};
// 找出第 5 小的元素（索引为 4）
// 此操作将把第 5 小的元素（值为 5）放在 v[4] 的位置
// 保证 v[0] 到 v[3] 的所有元素都小于等于 v[4]，v[5] 到 v[8] 的所有元素都大于等于 v[4]
nth_element(v.begin(), v.begin() + 4, v.end());
```

```
// v[4] 是第 5 小的元素，前面的元素都小于等于它，后面的元素都大于等于它
cout << " 第 5 小的元素是: " << v[4] << endl;

// 找中位数
// 将 v.size()/2 位置的元素（即中位数）放在正确的位置
// 对于含有 9 个元素的 v，中位数索引为 4，值为 5
nth_element(v.begin(), v.begin() + v.size()/2, v.end());
cout << " 中位数是: " << v[v.size()/2] << endl;
```

```
// 二分查找，返回元素是否存在
// 返回值: bool 值，表示值是否在有序区间中
// 参数说明:
// - first, last: 指定查找范围的迭代器
// - value: 要查找的值
// - comp: 比较函数（可选）
bool binary_search(first, last, value, comp);
```

4.1.1.5 binary_search

参数说明: - first、last: 指定查找范围的迭代器 - value: 要查找的值 - comp: 比较函数（可选）

时间复杂度: $O(\log N)$

前提条件: - 区间必须已经排序 - 对于自定义比较函数，区间必须按照该函数规定的顺序排序

示例:

```
vector<int> v = {1, 2, 5, 8, 9};
// 在已排序的 vector 中查找值 5
// binary_search 返回 true，因为 5 存在于 v 中
bool found = binary_search(v.begin(), v.end(), 5); // 返回 true
// 查找不存在的值 6
// binary_search 返回 false，因为 6 不存在于 v 中
found = binary_search(v.begin(), v.end(), 6);     // 返回 false

// 使用自定义比较函数进行查找
// 这里使用 lambda 表达式定义比较函数，当 a<b 时返回 true
bool found_custom = binary_search(v.begin(), v.end(), 5, [](int a, int b) {
    return a < b;  // 升序排序下的查找
});
```

注意事项: - 仅返回是否存在，不返回元素位置 - 如果需要获取位置，应使用 lower_bound 或 upper_bound

```
// 返回指向不小于指定值的第一个元素的迭代器
// 返回值: 迭代器，指向第一个大于等于 value 的元素；如果所有元素都小于 value，则返回 last
// 参数说明:
// - first, last: 指定查找范围的迭代器
// - value: 查找的值
// - comp: 比较函数（可选）
auto it = lower_bound(first, last, value, comp);
```

4.1.1.6 lower_bound

时间复杂度: $O(\log N)$

使用场景: - 查找第一个大于等于某值的元素 - 在有序数组中查找插入位置 - 处理有重复元素的有序区间

示例:

```
vector<int> v = {1, 2, 2, 3, 3, 3, 4, 5};
// 查找第一个大于等于 3 的元素
```

```cpp
// lower_bound 返回指向第一个值为 3 的迭代器（索引为 3 的位置）
auto it = lower_bound(v.begin(), v.end(), 3);
int pos = it - v.begin(); // 返回位置 3

// 查找第一个不小于 2.5 的元素
// 在 v 中没有 2.5，所以 lower_bound 返回指向第一个大于 2.5 的元素，即索引为 2 的第一个 3
auto it2 = lower_bound(v.begin(), v.end(), 2.5);
int pos2 = it2 - v.begin(); // 返回位置 2（指向第一个 3）

// 计算有序容器中元素的出现次数
// 使用 upper_bound 和 lower_bound 的差值计算元素 3 出现的次数
// upper_bound(v, 3) - lower_bound(v, 3) = 索引 6 - 索引 3 = 3
int val = 3;
int count = upper_bound(v.begin(), v.end(), val) - lower_bound(v.begin(), v.end(), val);
// count = 3，表示值 3 出现了 3 次
```

注意事项：- 如果没找到（所有元素都小于 value），返回 last 迭代器 - 区间必须已排序 - 通常与 upper_bound 配合使用

```cpp
// 返回指向大于指定值的第一个元素的迭代器
// 返回值：迭代器，指向第一个大于 value 的元素；如果所有元素都不大于 value，则返回 last
// 参数说明：
// - first, last: 指定查找范围的迭代器
// - value: 查找的值
// - comp: 比较函数（可选）
auto it = upper_bound(first, last, value, comp);
```

4.1.1.7 upper_bound 时间复杂度：O(log N)

使用场景：- 查找第一个大于某值的元素 - 与 lower_bound 配合查找某个值的范围

示例：

```cpp
vector<int> v = {1, 2, 2, 3, 3, 3, 4, 5};
// 查找第一个大于 3 的元素
// upper_bound 返回指向第一个大于 3 的元素，即值为 4 的位置（索引为 6）
auto it = upper_bound(v.begin(), v.end(), 3);
int pos = it - v.begin(); // 返回位置 6（指向 4）

// 查找一个不存在的值的上界
// upper_bound 返回指向第一个大于 3.5 的元素，即值为 4 的位置（索引为 6）
auto it2 = upper_bound(v.begin(), v.end(), 3.5);
int pos2 = it2 - v.begin(); // 返回位置 6（指向 4）

// 配合 lower_bound 获取等值区间
// 获取所有值为 3 的元素的范围
auto range_begin = lower_bound(v.begin(), v.end(), 3); // 索引 3
auto range_end = upper_bound(v.begin(), v.end(), 3);   // 索引 6
// 现在 [range_begin, range_end) 包含所有值为 3 的元素（索引 3,4,5）
// 可以用于遍历所有值为 3 的元素
for (auto it = range_begin; it != range_end; ++it) {
    // 对每个值为 3 的元素进行操作
    cout << *it << " at position " << (it - v.begin()) << endl;
}
```

注意事项：- 如果没找到（所有元素都不大于 value），返回 last 迭代器 - 区间必须已排序

```
// 返回等于指定值的元素范围，返回 pair< 迭代器, 迭代器 >
// 返回值：pair 对象，first 成员是 lower_bound 的结果，second 成员是 upper_bound 的结果
// 参数说明：
// - first, last: 指定查找范围的迭代器
// - value: 查找的值
// - comp: 比较函数（可选）
auto range = equal_range(first, last, value, comp);
```

4.1.1.8 equal_range 时间复杂度：O(log N)

使用场景：- 需要同时获取 lower_bound 和 upper_bound 时 - 在排序容器中寻找等值区间

示例：

```
vector<int> v = {1, 2, 2, 3, 3, 3, 4, 5};
// 获取所有等于 3 的元素范围
// equal_range 返回的 pair 中，first 指向第一个值为 3 的元素（索引 3）
// second 指向第一个大于 3 的元素（索引 6）
auto range = equal_range(v.begin(), v.end(), 3);
int first_pos = range.first - v.begin();   // 返回 3
int last_pos = range.second - v.begin();   // 返回 6
// [range.first, range.second) 区间内的所有值都等于 3
// 可以用于检查元素数量或遍历所有匹配元素
int count = range.second - range.first; // 值为 3 的元素数量

// 检查元素是否存在更简洁的方法
// 对于不存在的值 7，equal_range 返回的 range.first 和 range.second 相等
auto range2 = equal_range(v.begin(), v.end(), 7);
bool exists = (range2.first != range2.second); // 返回 false，因为 7 不存在
// 如果元素存在，range2.first 和 range2.second 之间会有至少一个元素
```

注意事项：- equal_range 相当于同时调用 lower_bound 和 upper_bound - 区间必须已排序

4.1.2 集合操作

```
// 合并两个已排序的区间到目标区间
// 返回值：指向目标区间末尾的迭代器
// 参数说明：
// - first1, last1: 第一个已排序范围的迭代器
// - first2, last2: 第二个已排序范围的迭代器
// - result: 目标范围的起始迭代器
// - comp: 比较函数（可选）
auto result_end = merge(first1, last1, first2, last2, result, comp);
```

4.1.2.1 merge 参数说明：- first1, last1：第一个已排序范围 - first2, last2：第二个已排序范围 - result：目标范围的起始迭代器 - comp：比较函数（可选）

时间复杂度：O(N+M)，其中 N 和 M 是两个输入范围的大小

前提条件：两个输入范围都必须已经按相同顺序排序

使用场景：- 合并两个有序数组 - 归并排序算法实现 - 在保持顺序的前提下组合两个数据集

示例：

```
vector<int> v1 = {1, 3, 5, 7};
vector<int> v2 = {2, 4, 6, 8};
vector<int> result(v1.size() + v2.size());
// 合并两个有序向量 v1 和 v2 到 result 中
// merge 函数保持元素的有序性，按照从小到大排序合并
```

```cpp
// 返回值是指向结果序列末尾的迭代器
auto end_it = merge(v1.begin(), v1.end(), v2.begin(), v2.end(), result.begin());
// 结果: {1, 2, 3, 4, 5, 6, 7, 8}
// 可以用返回值检查合并的元素个数
int merged_count = end_it - result.begin(); // 应为 8

// 使用比较函数进行降序合并
vector<int> desc_v1 = {7, 5, 3, 1};
vector<int> desc_v2 = {8, 6, 4, 2};
// 使用 greater<int>() 作为比较函数，表示在比较时认为 greater 返回 true 的元素排在前面
// 因此合并的结果是降序的
auto desc_end_it = merge(desc_v1.begin(), desc_v1.end(), desc_v2.begin(), desc_v2.end(),
      result.begin(), greater<int>());
// 结果: {8, 7, 6, 5, 4, 3, 2, 1}
```

注意事项: - 目标区间必须有足够的空间 - 对于目标与源重叠的情况，结果不确定

```cpp
// 将同一序列中的两个连续有序子序列合并为一个有序序列
// 返回值: 无
// 参数说明:
// - first: 序列起始位置
// - middle: 第一个子序列的结束位置（也是第二个子序列的起始位置）
// - last: 序列结束位置
// - comp: 比较函数（可选）
inplace_merge(first, middle, last, comp);
```

4.1.2.2 inplace_merge

参数说明: - `first`: 序列起始位置 - `middle`: 第一个子序列的结束位置（也是第二个子序列的起始位置） - `last`: 序列结束位置 - `comp`: 比较函数（可选）

时间复杂度: - 如果有足够的额外内存: $O(N)$ - 否则: $O(N \log N)$

使用场景: - 归并排序的一部分 - 当已经有两个相邻的有序子序列需要合并时

示例:

```cpp
vector<int> v = {1, 3, 5, 7, 2, 4, 6, 8};
// v 中 [0,4) 和 [4,8) 两个子序列都已排序
// inplace_merge 将这两个有序子序列原地合并成一个有序序列
// 操作会修改 v，使整个范围变为有序
inplace_merge(v.begin(), v.begin() + 4, v.end());
// 结果: {1, 2, 3, 4, 5, 6, 7, 8}

// 使用自定义比较函数进行降序合并
vector<int> desc_v = {7, 5, 3, 1, 8, 6, 4, 2};
// 两个子序列分别是 [0,4) 和 [4,8)，都是降序排列的
inplace_merge(desc_v.begin(), desc_v.begin() + 4, desc_v.end(), greater<int>());
// 结果: {8, 7, 6, 5, 4, 3, 2, 1}
```

```cpp
// 计算两个已排序区间的并集
// 返回值: 指向结果集合末尾的迭代器
// 参数说明:
// - first1, last1: 第一个已排序范围
// - first2, last2: 第二个已排序范围
// - result: 存储结果的起始迭代器
// - comp: 比较函数（可选）
auto end_it = set_union(first1, last1, first2, last2, result, comp);
```

4.1.2.3 set_union 时间复杂度：$O(N+M)$，其中 N 和 M 是两个输入范围的大小

使用场景：- 需要合并两个集合且不重复 - 在有序集合上执行集合运算

示例：

```
vector<int> v1 = {1, 2, 3, 4, 5};
vector<int> v2 = {3, 4, 5, 6, 7};
vector<int> result(10);  // 足够大以容纳所有可能结果
// 计算 v1 和 v2 的并集
// set_union 返回一个指向结果范围末尾的迭代器
auto it = set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), result.begin());
// 调整 result 的大小为实际元素个数
result.resize(it - result.begin());
// 结果：{1, 2, 3, 4, 5, 6, 7}
// 注意：并集中的每个元素只出现一次

// 处理重复元素
vector<int> v3 = {1, 2, 2, 3, 4};
vector<int> v4 = {2, 3, 3, 4, 5};
result.resize(10);  // 重置 result 大小
it = set_union(v3.begin(), v3.end(), v4.begin(), v4.end(), result.begin());
result.resize(it - result.begin());
// 结果：{1, 2, 2, 3, 3, 4, 5}
// 注意重复元素在结果中的出现次数是两个输入序列中该元素出现次数的最大值
// v3 中有 2 个'2'，v4 中有 1 个'2'，并集中有 2 个'2'
// v3 中有 1 个'3'，v4 中有 2 个'3'，并集中有 2 个'3'
```

注意事项：- 两个输入区间必须已排序 - 结果区间需要有足够空间 - 函数返回的是结果序列的末尾迭代器

```
// 计算两个已排序区间的交集
// 返回值：指向结果集合末尾的迭代器
// 参数说明：
// - first1, last1: 第一个已排序范围
// - first2, last2: 第二个已排序范围
// - result: 存储结果的起始迭代器
// - comp: 比较函数（可选）
auto end_it = set_intersection(first1, last1, first2, last2, result, comp);
```

4.1.2.4 set_intersection 时间复杂度：$O(N+M)$，其中 N 和 M 是两个输入范围的大小

使用场景：- 查找两个集合的共有元素 - 确定共同兴趣、共同特征等

示例：

```
vector<int> v1 = {1, 2, 3, 4, 5};
vector<int> v2 = {3, 4, 5, 6, 7};
vector<int> result(5);  // 足够大以容纳可能的结果
// 计算 v1 和 v2 的交集
// set_intersection 返回指向结果范围末尾的迭代器
auto it = set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), result.begin());
// 调整 result 的大小为实际元素个数
result.resize(it - result.begin());
// 结果：{3, 4, 5} - 这些是 v1 和 v2 中共有的元素

// 处理重复元素
vector<int> v3 = {1, 2, 2, 3, 4};
vector<int> v4 = {2, 3, 3, 4, 5};
result.resize(5);  // 重置 result 大小
```

```
it = set_intersection(v3.begin(), v3.end(), v4.begin(), v4.end(), result.begin());
result.resize(it - result.begin());
// 结果: {2, 3, 4}
// 注意重复元素在结果中的出现次数是两个输入序列中该元素出现次数的最小值
// v3 中有两个 2, v4 中有一个 2, 交集中有一个 2
// v3 中有一个 3, v4 中有两个 3, 交集中有一个 3
```

```
// 计算两个已排序区间的差集 (在区间 1 中但不在区间 2 中)
// 返回值: 指向结果集合末尾的迭代器
// 参数说明:
// - first1, last1: 第一个已排序范围
// - first2, last2: 第二个已排序范围
// - result: 存储结果的起始迭代器
// - comp: 比较函数 (可选)
auto end_it = set_difference(first1, last1, first2, last2, result, comp);
```

4.1.2.5 set_difference 时间复杂度: $O(N+M)$, 其中 N 和 M 是两个输入范围的大小

使用场景: - 找出第一个集合中独有的元素 - 移除共有元素

示例:

```
vector<int> v1 = {1, 2, 3, 4, 5};
vector<int> v2 = {3, 4, 5, 6, 7};
vector<int> result(5);
// 计算 v1 和 v2 的差集 (在 v1 中但不在 v2 中)
// set_difference 返回指向结果范围末尾的迭代器
auto it = set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), result.begin());
// 调整 result 的大小为实际元素个数
result.resize(it - result.begin());
// 结果: {1, 2}

// 注意差集不是对称的
// 计算 v2 和 v1 的差集 (在 v2 中但不在 v1 中)
it = set_difference(v2.begin(), v2.end(), v1.begin(), v1.end(), result.begin());
result.resize(it - result.begin());
// 结果: {6, 7}

// 处理重复元素
vector<int> v3 = {1, 2, 2, 3, 4};
vector<int> v4 = {2, 3, 3, 4, 5};
result.resize(5);
it = set_difference(v3.begin(), v3.end(), v4.begin(), v4.end(), result.begin());
result.resize(it - result.begin());
// 结果: {1, 2}
// v3 中有两个 2, v4 中有一个 2, 差集中保留一个 2
```

```
// 计算两个已排序区间的对称差集 (在区间 1 或区间 2 中, 但不同时在两者中)
// 返回值: 指向结果集合末尾的迭代器
// 参数说明:
// - first1, last1: 第一个已排序范围
// - first2, last2: 第二个已排序范围
// - result: 存储结果的起始迭代器
// - comp: 比较函数 (可选)
auto end_it = set_symmetric_difference(first1, last1, first2, last2, result, comp);
```

4.1.2.6 set_symmetric_difference 时间复杂度：O(N+M)，其中 N 和 M 是两个输入范围的大小

使用场景：- 寻找两个集合中不共有的元素 - XOR 类型的集合运算

示例：

```
vector<int> v1 = {1, 2, 3, 4, 5};
vector<int> v2 = {3, 4, 5, 6, 7};
vector<int> result(10);
// 计算 v1 和 v2 的对称差集（在 v1 或 v2 中，但不同时在两者中）
// set_symmetric_difference 返回指向结果范围末尾的迭代器
auto it = set_symmetric_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), result.begin());
// 调整 result 的大小为实际元素个数
result.resize(it - result.begin());
// 结果：{1, 2, 6, 7}

// 处理重复元素
vector<int> v3 = {1, 2, 2, 3, 4};
vector<int> v4 = {2, 3, 3, 4, 5};
result.resize(10);
it = set_symmetric_difference(v3.begin(), v3.end(), v4.begin(), v4.end(), result.begin());
result.resize(it - result.begin());
// 结果：{1, 2, 3, 5}
// v3 有两个 2，v4 有一个 2，对称差集中有一个 2
// v4 有两个 3，v3 有一个 3，对称差集中有一个 3
```

注意事项：- 对称差集是元素在一个集合但不在另一个集合中的所有元素 - 可以看作是 (A-B) ∪ (B-A)

4.1.3 数值算法

```


// 计算指定元素在范围中的出现次数
// 返回值：int，表示元素出现的次数
// 参数说明：
// - first, last: 指定范围的迭代器
// - value: 要计数的值
int num = count(first, last, value);

// 计算满足谓词条件的元素个数
// 返回值：int，表示满足条件的元素个数
// 参数说明：
// - first, last: 指定范围的迭代器
// - pred: 谓词函数，接收一个元素并返回 bool 值
int num = count_if(first, last, pred);
```

4.1.3.1 count / count_if 时间复杂度：O(N)，其中 N 是范围内的元素个数

使用场景：- 统计特定元素出现次数 - 统计满足特定条件的元素个数

示例：

```
vector<int> v = {1, 2, 3, 2, 5, 2, 7};
// 计算值为 2 的元素出现次数
int cnt = count(v.begin(), v.end(), 2); // 返回 3

// 计算满足条件的元素个数
// 这里使用 lambda 表达式定义谓词函数，判断元素是否为偶数
int even_count = count_if(v.begin(), v.end(), [](int x) {
    return x % 2 == 0;
}); // 返回 3（2 出现三次）
```

```cpp
// 在字符串中计数
string str = "Hello World";
// 计算字符 'l' 出现的次数
int l_count = count(str.begin(), str.end(), 'l'); // 返回 3

// 复杂条件统计
vector<pair<string, int>> students = {
    {"Alice", 85}, {"Bob", 92}, {"Charlie", 85}, {"David", 78}
};
// 计算成绩大于等于 85 的学生数量
int high_score = count_if(students.begin(), students.end(), [](const pair<string, int>& s) {
    return s.second >= 85;
}); // 返回 3
```

```cpp
// 返回范围中最小/最大元素的迭代器
// 返回值：迭代器，指向最小/最大元素
// 参数说明：
// - first, last: 指定范围的迭代器
// - comp: 比较函数（可选）
auto min_it = min_element(first, last, comp);
auto max_it = max_element(first, last, comp);

// 返回范围中最小和最大元素的迭代器
// 返回值：pair 对象，first 成员是最小元素的迭代器，second 成员是最大元素的迭代器
// 参数说明：
// - first, last: 指定范围的迭代器
// - comp: 比较函数（可选）
auto minmax = minmax_element(first, last, comp);
```

4.1.3.2 min_element / max_element / minmax_element

时间复杂度：- min_element/max_element: $O(N)$ - minmax_element: $O(N)$，但比单独调用 min_element 和 max_element 更高效

使用场景：- 寻找数组中的最值 - 寻找最大/最小对象 - 同时需要最大和最小值时

示例：

```cpp
vector<int> v = {5, 2, 8, 1, 9};
// 查找最小元素
auto min_it = min_element(v.begin(), v.end()); // 指向 1
// 查找最大元素
auto max_it = max_element(v.begin(), v.end()); // 指向 9

cout << " 最小值: " << *min_it << " 位置: " << (min_it - v.begin()) << endl;
cout << " 最大值: " << *max_it << " 位置: " << (max_it - v.begin()) << endl;

// 同时获取最小和最大元素
auto minmax = minmax_element(v.begin(), v.end());
int min_val = *minmax.first;   // 1
int max_val = *minmax.second;  // 9

// 使用自定义比较函数
struct Person {
    string name;
    int age;
};
vector<Person> people = {{"Alice", 25}, {"Bob", 20}, {"Charlie", 30}};
```

```cpp
// 查找年龄最大的 Person
auto oldest = max_element(people.begin(), people.end(),
    [](const Person& a, const Person& b) { return a.age < b.age; });
cout << " 最年长的人是: " << oldest->name << ", " << oldest->age << " 岁" << endl;
```

注意事项: - 返回的是迭代器，需要解引用获取值 - 如果有多个最值元素，返回第一个最小值和最后一个最大值

```cpp
// 计算范围内元素的累加和
// 需引入头文件 #include <numeric>
// 返回值: 累加结果
// 参数说明:
// - first, last: 指定范围的迭代器
// - init_val: 初始值
// - op: 二元操作符（可选），默认为加法
T sum = accumulate(first, last, init_val, op);
```

4.1.3.3 accumulate 参数说明: - first, last: 指定范围 - init_val: 初始值 - op: 二元操作符（可选），默认为加法

时间复杂度: $O(N)$

使用场景: - 计算元素总和 - 将容器中所有元素连接起来 - 执行自定义累积操作

示例:

```cpp
#include <numeric>
#include <functional> // 提供 multiplies 等函数对象

vector<int> v = {1, 2, 3, 4, 5};
// 计算累加和
int sum = accumulate(v.begin(), v.end(), 0); // 返回 15

// 计算乘积
int product = accumulate(v.begin(), v.end(), 1, multiplies<int>()); // 返回 120

// 字符串连接
vector<string> words = {"Hello", "World", "C++"};
// 使用 accumulate 将字符串连接起来
// next(words.begin()) 跳过第一个元素，避免在结果前面多一个空格
string sentence = accumulate(next(words.begin()), words.end(), words[0],
    [](const string& a, const string& b) { return a + " " + b; });
// 返回 "Hello World C++"

// 更复杂的累积
vector<pair<string, int>> sales = {{"A", 100}, {"B", 200}, {"C", 150}};
// 计算所有销售额的总和
int total = accumulate(sales.begin(), sales.end(), 0,
    [](int sum, const pair<string, int>& p) { return sum + p.second; });
// 返回 450
```

注意事项: - 初始值的类型决定了累积结果的类型 - 操作必须满足结合律才能保证结果正确

```cpp
// 计算两个区间的内积，需引入头文件 #include <numeric>
// 返回值: 内积结果
// 参数说明:
// - first1, last1: 第一个区间的迭代器
// - first2: 第二个区间的起始迭代器
```

```
// - init: 初始值
// - binary_op1: 累计操作（默认为加法）
// - binary_op2: 元素间操作（默认为乘法）
T result = inner_product(first1, last1, first2, init, binary_op1, binary_op2);
```

4.1.3.4 inner_product

参数说明：- first1, last1：第一个区间 - first2：第二个区间的起始位置 - init：初始值 - binary_op1：累计操作（默认为加法）- binary_op2：元素间操作（默认为乘法）

时间复杂度：$O(N)$

使用场景：- 计算向量点积 - 计算加权和 - 比较两个序列的相似度

示例：

```
#include <numeric>

vector<int> v1 = {1, 2, 3};
vector<int> v2 = {4, 5, 6};

// 计算点积: 1*4 + 2*5 + 3*6 = 4 + 10 + 18 = 32
int dot_product = inner_product(v1.begin(), v1.end(), v2.begin(), 0);
cout << " 点积: " << dot_product << endl;

// 自定义操作：计算匹配元素数量
int matches = inner_product(v1.begin(), v1.end(), v2.begin(), 0,
                            std::plus<int>(),
                            [](int a, int b) { return a == b ? 1 : 0; });

// 计算欧几里得距离的平方
int euclidean_dist_squared = inner_product(
    v1.begin(), v1.end(), v2.begin(), 0,
    std::plus<int>(),
    [](int a, int b) { return (a - b) * (a - b); });
```

注意事项：- 第二个区间至少需要与第一个区间相同长度 - 操作的顺序是先应用 $\mathrm{binary\_op2}$ 到对应元素，然后用 $\mathrm{binary\_op1}$ 累积结果

```
// 计算部分和序列，需引入头文件 #include <numeric>
// 返回值：无
// 参数说明：
// - first, last: 输入范围的迭代器
// - result: 输出范围的起始迭代器
// - op: 二元操作（可选），默认为加法
partial_sum(first, last, result, op);
```

4.1.3.5 partial_sum

时间复杂度：$O(N)$

参数说明：- first, last：输入范围 - result：输出范围的起始位置 - op：二元操作（可选），默认为加法

使用场景：- 计算前缀和 - 计算累积乘积 - 动态规划中的状态转移

示例：

```
#include <numeric>

vector<int> v = {1, 2, 3, 4, 5};
vector<int> result(v.size());

// 计算部分和
partial_sum(v.begin(), v.end(), result.begin());
// 结果: {1, 3, 6, 10, 15}
```

```
// 使用自定义操作：累积乘积
partial_sum(v.begin(), v.end(), result.begin(), multiplies<int>());
// 结果：{1, 2, 6, 24, 120}
```

注意事项：- 输出序列可以与输入序列相同 - 一般用于快速计算前缀和，避免重复累加计算

```
// 计算相邻元素的差，需引入头文件 #include <numeric>
// 返回值：无
// 参数说明：
// - first, last: 输入范围的迭代器
// - result: 输出范围的起始迭代器
// - op: 二元操作（可选），默认为减法
adjacent_difference(first, last, result, op);
```

4.1.3.6 adjacent_difference 时间复杂度：$O(N)$

参数说明：- first, last：输入范围 - result：输出范围的起始位置 - op：二元操作（可选），默认为减法

使用场景：- 计算序列的差分 - 寻找序列中的变化 - 时间序列分析

示例：

```
#include <numeric>

vector<int> v = {1, 3, 6, 10, 15};
vector<int> result(v.size());

// 计算相邻元素的差
adjacent_difference(v.begin(), v.end(), result.begin());
// 结果：{1, 2, 3, 4, 5}
// 第一个元素保持不变，其余元素为当前元素减前一个元素

// 使用自定义操作：计算比值
adjacent_difference(v.begin(), v.end(), result.begin(),
                    [](int a, int b) { return a / b; });
// 结果：{1, 3, 2, 10/6, 15/10} = {1, 3, 2, 1.67, 1.5}
```

注意事项：- 第一个元素直接复制，不应用操作 - 输入输出范围可以是同一序列

```
// 用连续递增的值填充范围，需引入头文件 #include <numeric>
// 返回值：无
// 参数说明：
// - first, last: 要填充的范围的迭代器
// - value: 起始值
iota(first, last, value);
```

4.1.3.7 iota 时间复杂度：$O(N)$

参数说明：- first, last：要填充的范围 - value：起始值

使用场景：- 创建一个递增序列 - 生成索引数组 - 初始化容器

示例：

```
#include <numeric>

vector<int> v(10);
// 用 0 到 9 填充数组
iota(v.begin(), v.end(), 0);
```

```
// 结果: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

// 创建自定义序列
vector<int> seq(5);
iota(seq.begin(), seq.end(), 100);
// 结果: {100, 101, 102, 103, 104}
```

4.1.4 修改序列

```
// 复制元素到目标范围
// 返回值: 指向目标范围末尾的迭代器
// 参数说明:
// - first, last: 输入范围的迭代器
// - result: 目标范围的起始迭代器
auto end_it = copy(first, last, result);

// 根据条件复制元素
// 返回值: 指向目标范围末尾的迭代器
// 参数说明:
// - first, last: 输入范围的迭代器
// - result: 目标范围的起始迭代器
// - pred: 谓词函数，接收一个元素并返回 bool 值
auto end_it = copy_if(first, last, result, pred);

// 复制指定数量的元素
// 返回值: 指向目标范围末尾的迭代器
// 参数说明:
// - first: 输入范围的起始迭代器
// - count: 要复制的元素数量
// - result: 目标范围的起始迭代器
auto end_it = copy_n(first, count, result);

// 从后向前复制（避免覆盖）
// 返回值: 指向目标范围末尾的迭代器
// 参数说明:
// - first, last: 输入范围的迭代器
// - result_end: 目标范围的末尾迭代器
auto end_it = copy_backward(first, last, result_end);
```

4.1.4.1 copy / copy_if / copy_n / copy_backward 时间复杂度: $O(N)$

使用场景: - 将一个容器的内容复制到另一个容器 - 选择性地复制满足条件的元素 - 处理重叠区域的复制

示例:

```
vector<int> v1 = {1, 2, 3, 4, 5};
vector<int> v2(5);

// 基本复制
copy(v1.begin(), v1.end(), v2.begin());
// v2: {1, 2, 3, 4, 5}

// 条件复制: 只复制偶数
vector<int> even_numbers;
copy_if(v1.begin(), v1.end(), back_inserter(even_numbers), [](int x) {
    return x % 2 == 0;
});
```

```
// even_numbers: {2, 4}

// 复制前 3 个元素
vector<int> v3(3);
copy_n(v1.begin(), 3, v3.begin());
// v3: {1, 2, 3}

// 从后向前复制（适用于目标和源重叠的情况）
vector<int> v4 = {1, 2, 3, 4, 5};
copy_backward(v4.begin(), v4.begin() + 3, v4.begin() + 4);
// v4: {1, 2, 1, 2, 3}
```

注意事项：- 目标区域必须有足够空间 - 使用 back_inserter 可以避免预分配空间 - 处理重叠区域时，copy 从前向后复制，copy_backward 从后向前复制

```
// 移动元素到目标范围
// 返回值：指向目标范围末尾的迭代器
// 参数说明：
// - first, last: 输入范围的迭代器
// - result: 目标范围的起始迭代器
auto end_it = move(first, last, result);

// 从后向前移动（避免覆盖）
// 返回值：指向目标范围末尾的迭代器
// 参数说明：
// - first, last: 输入范围的迭代器
// - result_end: 目标范围的末尾迭代器
auto end_it = move_backward(first, last, result_end);
```

4.1.4.2 move / move_backward 时间复杂度：$O(N)$

使用场景：- 高效转移资源所有权 - 避免不必要的拷贝操作 - 实现移动语义

示例：

```
vector<string> v1 = {"Hello", "World", "C++"};
vector<string> v2(3);

// 移动元素（避免拷贝）
move(v1.begin(), v1.end(), v2.begin());
// v2 包含移动后的字符串，v1 中的字符串可能变为空

// 处理重叠区域
vector<string> v3 = {"A", "B", "C", "D", "E"};
move_backward(v3.begin(), v3.begin() + 3, v3.begin() + 4);
// v3 可能变为：{"A", "A", "B", "C", "E"}
```

注意事项：- 移动后源对象处于有效但未指定状态 - 移动语义通常比复制更高效，特别是对于大型对象

```
// 用指定值填充范围
// 返回值：无
// 参数说明：
// - first, last: 要填充的范围的迭代器
// - value: 填充的值
fill(first, last, value);
```

```
// 用指定值填充指定数量的元素
// 返回值：指向目标范围末尾的迭代器
// 参数说明：
// - first: 要填充的范围的起始迭代器
// - count: 要填充的元素数量
// - value: 填充的值
fill_n(first, count, value);
```

4.1.4.3 fill / fill_n 时间复杂度：$O(N)$

使用场景：- 初始化数组或向量 - 重置容器的所有值 - 填充特定区域

示例：

```
vector<int> v(5);
// 用 7 填充整个范围
fill(v.begin(), v.end(), 7);
// 结果：{7, 7, 7, 7, 7}

// 只填充部分元素
fill_n(v.begin(), 3, 9);
// 结果：{9, 9, 9, 7, 7}

// 二维数组的初始化
vector<vector<int>> matrix(3, vector<int>(3));
for (auto& row : matrix) {
    fill(row.begin(), row.end(), 0);
}
// 所有元素都被设置为 0
```

注意事项：- fill_n 不检查空间是否足够，调用者需确保有足够空间 - 对于自定义类型，要确保赋值操作是高效的

```
// 对范围内的元素应用函数，结果放入目标范围
// 返回值：指向目标范围末尾的迭代器
// 参数说明：
// - first1, last1: 输入范围的迭代器
// - result: 目标范围的起始迭代器
// - unary_op: 一元操作函数，接收一个元素并返回转换后的值
auto end_it = transform(first1, last1, result, unary_op);

// 对两个范围内的对应元素应用二元函数
// 返回值：指向目标范围末尾的迭代器
// 参数说明：
// - first1, last1: 第一个输入范围的迭代器
// - first2: 第二个输入范围的起始迭代器
// - result: 目标范围的起始迭代器
// - binary_op: 二元操作函数，接收两个元素并返回转换后的值
auto end_it = transform(first1, last1, first2, result, binary_op);
```

4.1.4.4 transform 时间复杂度：$O(N)$

使用场景：- 元素级别的数据转换 - 对两个序列执行元素对应的运算 - 应用映射或过滤函数

示例：

```
vector<int> v = {1, 2, 3, 4, 5};
vector<int> result(v.size());

// 一元操作：每个元素平方
```

```cpp
transform(v.begin(), v.end(), result.begin(), [](int x) {
    return x * x;
});
// 结果: {1, 4, 9, 16, 25}

// 二元操作: 对应元素相加
vector<int> v2 = {10, 20, 30, 40, 50};
transform(v.begin(), v.end(), v2.begin(), result.begin(), plus<int>());
// 结果: {11, 22, 33, 44, 55}

// 复杂变换: 结构转换
struct Person { string name; int age; };
struct PersonView { string display; };

vector<Person> people = {{"Alice", 25}, {"Bob", 30}};
vector<PersonView> views(people.size());

transform(people.begin(), people.end(), views.begin(), [](const Person& p) {
    return PersonView{p.name + " (" + to_string(p.age) + ")"};
});
// 结果: [{"Alice (25)"}, {"Bob (30)"}]
```

注意事项: - 目标范围必须有足够空间 - 源和目标可以是同一范围 - 可以使用标准库中的函数对象或自定义函数

```cpp
// 替换范围内的指定值
// 返回值: 无
// 参数说明:
// - first, last: 输入范围的迭代器
// - old_value: 要替换的值
// - new_value: 替换后的值
replace(first, last, old_value, new_value);

// 根据条件替换
// 返回值: 无
// 参数说明:
// - first, last: 输入范围的迭代器
// - pred: 谓词函数, 接收一个元素并返回 bool 值
// - new_value: 替换后的值
replace_if(first, last, pred, new_value);

// 替换并复制到新范围
// 返回值: 指向目标范围末尾的迭代器
// 参数说明:
// - first, last: 输入范围的迭代器
// - result: 目标范围的起始迭代器
// - old_value: 要替换的值
// - new_value: 替换后的值
auto end_it = replace_copy(first, last, result, old_value, new_value);
```

4.1.4.5 replace / replace_if / replace_copy 时间复杂度: $O(N)$

使用场景: - 数据清洗和转换 - 条件替换 - 生成替换后的新序列

示例:

```cpp
vector<int> v = {1, 2, 3, 2, 5};
// 替换值为 2 的元素为 99
replace(v.begin(), v.end(), 2, 99);
// 结果: {1, 99, 3, 99, 5}

// 条件替换：将所有负数替换为 0
vector<int> v2 = {-1, 2, -3, 4, -5};
replace_if(v2.begin(), v2.end(), [](int x) { return x < 0; }, 0);
// 结果: {0, 2, 0, 4, 0}

// 替换并复制
vector<int> v3 = {1, 2, 3, 2, 5};
vector<int> result(v3.size());
replace_copy(v3.begin(), v3.end(), result.begin(), 2, 99);
// v3 不变，result: {1, 99, 3, 99, 5}
```

```cpp
// 移除连续重复元素，返回新逻辑结尾的迭代器
// 返回值：指向新逻辑结尾的迭代器
// 参数说明：
// - first, last: 输入范围的迭代器
// - pred: 比较函数（可选），接收两个元素并返回 bool 值
auto new_end = unique(first, last, pred);

// 复制不重复的元素到目标范围
// 返回值：指向目标范围末尾的迭代器
// 参数说明：
// - first, last: 输入范围的迭代器
// - result: 目标范围的起始迭代器
// - pred: 比较函数（可选），接收两个元素并返回 bool 值
auto result_end = unique_copy(first, last, result, pred);
```

4.1.4.6 unique / unique_copy 时间复杂度：$O(N)$

使用场景：- 移除序列中的连续重复元素 - 创建不包含连续重复元素的副本 - 数据去重（需要先排序）

示例：

```cpp
vector<int> v = {1, 1, 2, 2, 3, 3, 3, 4, 5, 5};
// 移除连续重复元素
auto it = unique(v.begin(), v.end());
v.resize(it - v.begin());
// 结果: {1, 2, 3, 4, 5}

// 使用自定义比较
vector<string> words = {"apple", "APPLE", "banana", "BANANA", "orange"};
auto it2 = unique(words.begin(), words.end(), [](const string& a, const string& b) {
    return a.size() == b.size();  // 根据长度判断是否相同
});
words.resize(it2 - words.begin());
// 可能的结果: {"apple", "banana", "orange"}

// 复制不重复元素到新容器
vector<int> v2 = {1, 1, 2, 2, 3, 3, 3, 4, 5, 5};
vector<int> result;
unique_copy(v2.begin(), v2.end(), back_inserter(result));
// 结果: {1, 2, 3, 4, 5}
```

注意事项：- unique 只移除连续重复的元素，所以通常先排序 - 不会改变容器大小，需手动调整（通常用 resize） - 被”移除”的元素仍在容器中，但处于未指定状态

```
// 反转序列中的元素顺序
// 返回值：无
// 参数说明：
// - first, last: 输入范围的迭代器
reverse(first, last);

// 将反转后的序列复制到目标范围
// 返回值：指向目标范围末尾的迭代器
// 参数说明：
// - first, last: 输入范围的迭代器
// - result: 目标范围的起始迭代器
auto result_end = reverse_copy(first, last, result);
```

4.1.4.7 reverse / reverse_copy 时间复杂度：$O(N)$

使用场景：- 反转数组、字符串或向量 - 创建反转的副本而不修改原序列 - 算法中需要处理反向序列

示例：

```
vector<int> v = {1, 2, 3, 4, 5};
// 反转整个序列
reverse(v.begin(), v.end());
// 结果：{5, 4, 3, 2, 1}

// 反转部分序列
string s = "Hello World";
reverse(s.begin() + 6, s.end());
// 结果："Hello dlroW"

// 创建反转副本
vector<int> v2 = {1, 2, 3, 4, 5};
vector<int> result(v2.size());
reverse_copy(v2.begin(), v2.end(), result.begin());
// v2 不变，result: {5, 4, 3, 2, 1}
```

```
// 旋转序列中的元素，将 middle 指向的元素放到首位
// 返回值：指向旋转后原来首元素的新位置的迭代器
// 参数说明：
// - first, middle: 输入范围的起始迭代器和旋转点
// - last: 输入范围的结束迭代器
auto new_first = rotate(first, middle, last);

// 旋转并复制到目标范围
// 返回值：指向目标范围末尾的迭代器
// 参数说明：
// - first, middle: 输入范围的起始迭代器和旋转点
// - last: 输入范围的结束迭代器
// - result: 目标范围的起始迭代器
auto result_end = rotate_copy(first, middle, last, result);
```

4.1.4.8 rotate / rotate_copy 时间复杂度：$O(N)$

使用场景：- 循环移位 - 重新排列数据 - 实现循环缓冲区

示例：

```cpp
vector<int> v = {1, 2, 3, 4, 5};
// 将 3 放到开头（向左旋转 2 个位置）
rotate(v.begin(), v.begin() + 2, v.end());
// 结果：{3, 4, 5, 1, 2}

// 复制旋转结果到新容器
vector<int> v2 = {1, 2, 3, 4, 5};
vector<int> result(v2.size());
rotate_copy(v2.begin(), v2.begin() + 1, v2.end(), result.begin());
// 结果：{2, 3, 4, 5, 1}
```

注意事项：- `middle` 必须在 `first` 和 `last` 之间 - 返回值是旋转后原来首元素的新位置 - 可以用于高效实现循环队列和缓冲区

```cpp
// 随机打乱序列，使用随机数生成器
// 返回值：无
// 参数说明：
// - first, last: 输入范围的迭代器
// - gen: 随机数生成器
shuffle(first, last, gen);
```

4.1.4.9 `shuffle` 时间复杂度：$O(N)$

使用场景：- 随机化数据集 - 实现洗牌算法 - Monte Carlo 方法

示例：

```cpp
#include <random>

vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8};
// 使用随机数引擎
random_device rd;
mt19937 g(rd());
shuffle(v.begin(), v.end(), g);
// 结果：序列随机排列，如 {3, 1, 7, 8, 4, 2, 5, 6}
```

注意事项：- `shuffle` 比 `random_shuffle` 提供更好的随机性，推荐使用 - 需要高质量随机性时，避免使用 `rand()` - 推荐使用 `<random>` 头文件中的随机数生成器

4.1.5 排列组合

```cpp
// 生成下一个字典序排列，成功返回 true，否则返回 false
// 返回值：bool 值，表示是否生成了下一个排列
// 参数说明：
// - first, last: 输入范围的迭代器
// - comp: 比较函数（可选）
bool has_next = next_permutation(first, last, comp);

// 生成上一个字典序排列，成功返回 true，否则返回 false
// 返回值：bool 值，表示是否生成了上一个排列
// 参数说明：
// - first, last: 输入范围的迭代器
// - comp: 比较函数（可选）
bool has_prev = prev_permutation(first, last, comp);
```

4.1.5.1 `next_permutation` / `prev_permutation` 时间复杂度：平均 $O(N)$，最坏 $O(N^2)$

使用场景：- 枚举所有可能的排列 - 组合优化问题 - 生成下一个排列
示例：

```
vector<int> v = {1, 2, 3};
// 枚举所有排列
do {
    // 处理当前排列
    for (int n : v) cout << n << ' ';
    cout << '\n';
} while (next_permutation(v.begin(), v.end()));

// 输出：
// 1 2 3
// 1 3 2
// 2 1 3
// 2 3 1
// 3 1 2
// 3 2 1

// 对于已排序的最大排列，下一个排列会回到开始
vector<int> v2 = {3, 2, 1};
bool has_next = next_permutation(v2.begin(), v2.end());
// has_next 为 false，v2 变为 {1, 2, 3}

// 使用 prev_permutation
vector<int> v3 = {3, 2, 1};
do {
    // 处理当前排列
    for (int n : v3) cout << n << ' ';
    cout << '\n';
} while (prev_permutation(v3.begin(), v3.end()));

// 输出与 next_permutation 相反的顺序
```

注意事项：- 初始序列需要是有效的排列（没有重复元素）- 想要生成所有排列，起始序列应该是最小排列（已排序）- 函数会修改序列，将其变为下一个/上一个排列

4.1.6 堆操作

```
// 创建堆
// 返回值：无
// 参数说明：
// - first, last: 输入范围的迭代器
// - comp: 比较函数（可选）
make_heap(first, last, comp);

// 添加元素到堆
// 返回值：无
// 参数说明：
// - first, last: 输入范围的迭代器
// - comp: 比较函数（可选）
push_heap(first, last, comp);

// 移除堆顶元素
// 返回值：无
// 参数说明：
// - first, last: 输入范围的迭代器
// - comp: 比较函数（可选）
```

```
pop_heap(first, last, comp);

// 堆排序
// 返回值：无
// 参数说明：
// - first, last: 输入范围的迭代器
// - comp: 比较函数（可选）
sort_heap(first, last, comp);
```

时间复杂度：- make_heap: $O(N)$ - push_heap: $O(\log N)$ - pop_heap: $O(\log N)$ - sort_heap: $O(N \log N)$
使用场景：- 优先队列实现 - 堆排序 - 动态维护最大/最小值
示例：

```
vector<int> v = {3, 1, 4, 1, 5, 9};

// 构建最大堆
make_heap(v.begin(), v.end());
// v 可能变为：{9, 5, 4, 1, 1, 3}

// 添加元素
v.push_back(6);
push_heap(v.begin(), v.end());
// 先添加元素到末尾，然后调整堆

// 移除堆顶元素
pop_heap(v.begin(), v.end());
int top = v.back(); // 获取堆顶元素
v.pop_back();       // 从容器中移除
// pop_heap 将堆顶元素移到末尾，并重新调整堆

// 堆排序
sort_heap(v.begin(), v.end());
// 结果：升序排列
```

详细用法：

```
// 创建最小堆
vector<int> min_heap = {6, 1, 2, 8, 3, 4};
make_heap(min_heap.begin(), min_heap.end(), greater<int>());
// min_heap 可能为：{1, 3, 2, 8, 6, 4}

// 弹出最小元素
pop_heap(min_heap.begin(), min_heap.end(), greater<int>());
int min_val = min_heap.back();
min_heap.pop_back();
// min_val = 1

// 添加元素到最小堆
min_heap.push_back(0);
push_heap(min_heap.begin(), min_heap.end(), greater<int>());
// 0 现在是堆顶最小元素

// 自定义堆比较函数
struct Person {
    string name;
    int priority;
};
```

```cpp
vector<Person> persons = {
    {"Alice", 3}, {"Bob", 1}, {"Charlie", 2}
};

make_heap(persons.begin(), persons.end(),
    [](const Person& a, const Person& b) {
        return a.priority < b.priority;  // 优先级高的放堆顶
    });
```

注意事项：- 默认创建的是最大堆 - 常用于实现优先队列和堆排序 - 需要手动维护容器大小

4.2 迭代器相关函数

使用前需要包含头文件 #include <iterator>

```cpp
// 将迭代器向前或向后移动 n 个位置
// 返回值：无
// 参数说明：
// - it: 要移动的迭代器
// - n: 移动的距离，可以为负
advance(it, n);
```

4.2.0.1 advance 时间复杂度：- 随机访问迭代器：$O(1)$ - 其他迭代器：$O(N)$

示例：

```cpp
list<int> lst = {1, 2, 3, 4, 5};
auto it = lst.begin();
advance(it, 2); // 现在 it 指向 3
```

注意事项：- 没有边界检查，调用者必须确保移动后迭代器仍然有效 - 对于双向迭代器，n 可以为负 - 对于单向迭代器，n 必须为非负

```cpp
// 计算两个迭代器之间的距离
// 返回值：int，表示两个迭代器之间的距离
// 参数说明：
// - first, last: 要计算距离的两个迭代器
int d = distance(first, last);
```

4.2.0.2 distance 时间复杂度：- 随机访问迭代器：$O(1)$ - 其他迭代器：$O(N)$

示例：

```cpp
vector<int> v = {1, 2, 3, 4, 5};
int d = distance(v.begin(), v.end()); // 返回 5

list<int> lst = {10, 20, 30, 40};
auto it = lst.begin();
advance(it, 2);
int d2 = distance(lst.begin(), it); // 返回 2
```

注意事项：- first 必须可达 last，否则行为未定义 - 对于非随机访问迭代器，需要遍历计算距离

```
// 返回前进 n 个位置的迭代器（不修改原迭代器）
// 返回值：前进 n 个位置后的迭代器
// 参数说明：
// - it: 要前进的迭代器
// - n: 前进的距离，默认为 1
auto next_it = next(it, n = 1);

// 返回后退 n 个位置的迭代器（不修改原迭代器）
// 返回值：后退 n 个位置后的迭代器
// 参数说明：
// - it: 要后退的迭代器
// - n: 后退的距离，默认为 1
auto prev_it = prev(it, n = 1);
```

4.2.0.3 next / prev 时间复杂度：与 advance 相同

示例：

```
list<int> lst = {1, 2, 3, 4, 5};
auto it = lst.begin();
auto it2 = next(it, 2); // 指向 3，不修改 it
auto it3 = prev(lst.end(), 1); // 指向 5
```

注意事项：- 与 advance 不同，不会修改原迭代器 - C++11 引入，提供更安全的迭代器移动方式

```
// 创建插入迭代器
// 返回值：插入迭代器
// 参数说明：
// - container: 要插入元素的容器
back_inserter(container);   // 在容器末尾插入
front_inserter(container);  // 在容器开头插入
inserter(container, it);    // 在指定位置插入
```

4.2.0.4 back_inserter / front_inserter / inserter 使用场景：- 在 STL 算法中动态扩展容器 - 避免手动调整容器大小 - 实现流式处理

示例：

```
vector<int> source = {1, 2, 3, 4, 5};
vector<int> dest;

// 使用 back_inserter 避免预先分配空间
copy(source.begin(), source.end(), back_inserter(dest));
// dest: {1, 2, 3, 4, 5}

// 过滤并插入
copy_if(source.begin(), source.end(), back_inserter(dest),
        [](int x) { return x % 2 == 0; });
// 添加到 dest 后面: {1, 2, 3, 4, 5, 2, 4}

// front_inserter 适用于双端队列和列表
list<int> lst;
copy(source.begin(), source.end(), front_inserter(lst));
// lst: {5, 4, 3, 2, 1} (注意顺序反转)

// 在特定位置插入
list<int> lst2 = {10, 20, 30, 40};
```

```cpp
auto pos = next(lst2.begin(), 2); // 指向 30
copy(source.begin(), source.end(), inserter(lst2, pos));
// lst2: {10, 20, 1, 2, 3, 4, 5, 30, 40}
```

注意事项：- back_inserter 调用 push_back，适用于 vector、deque、list 等 - front_inserter 调用 push_front，只适用于 deque、list 等 - inserter 调用 insert，适合所有支持插入的容器

4.3 函数对象

需包含头文件 #include <functional>

STL 提供了许多预定义的函数对象，它们是实现了函数调用运算符（operator()）的类。这些对象可以像函数一样使用，但有更多灵活性。

4.3.1 预定义函数对象

```cpp
// 算术运算
plus<T>()           // 函数对象，实现 a + b 的加法操作
minus<T>()          // 函数对象，实现 a - b 的减法操作
multiplies<T>()     // 函数对象，实现 a * b 的乘法操作
divides<T>()        // 函数对象，实现 a / b 的除法操作
modulus<T>()        // 函数对象，实现 a % b 的取模操作
negate<T>()         // 函数对象，实现 -a 的取负操作

// 比较运算
equal_to<T>()       // 函数对象，检查 a == b 是否成立
not_equal_to<T>()   // 函数对象，检查 a != b 是否成立
greater<T>()        // 函数对象，检查 a > b 是否成立
less<T>()           // 函数对象，检查 a < b 是否成立
greater_equal<T>() // 函数对象，检查 a >= b 是否成立
less_equal<T>()     // 函数对象，检查 a <= b 是否成立

// 逻辑运算
logical_and<T>()    // 函数对象，实现 a && b 的逻辑与操作
logical_or<T>()     // 函数对象，实现 a || b 的逻辑或操作
logical_not<T>()    // 函数对象，实现 !a 的逻辑非操作
```

函数对象的优点：1. 比函数指针更高效，因为编译器可以内联它们 2. 可以有状态（成员变量）3. 可以与 STL 算法无缝配合 4. 便于组合和自定义
使用场景：- 作为排序和比较操作的谓词 - 参与 STL 算法中的函数组合 - 实现自定义的操作，尤其是需要保存状态的操作

4.3.2 基本用法示例

```cpp
#include <functional>
#include <algorithm>
#include <vector>
#include <iostream>

vector<int> v = {5, 2, 8, 1, 9};

// 使用 greater 函数对象进行降序排序
// greater<int>() 创建一个函数对象，当第一个参数大于第二个参数时返回 true
// 这会导致较大的元素排在前面，实现降序排序
sort(v.begin(), v.end(), greater<int>());
// 结果: {9, 8, 5, 2, 1}

// 使用 less 函数对象进行升序排序（这实际上是默认行为）
// less<int>() 创建一个函数对象，当第一个参数小于第二个参数时返回 true
```

```
sort(v.begin(), v.end(), less<int>());
// 结果：{1, 2, 5, 8, 9}
```

4.3.3 函数对象用于算术运算

```
#include <functional>
#include <algorithm>
#include <numeric>
#include <vector>

// 准备两个向量
vector<int> v1 = {1, 2, 3, 4};
vector<int> v2 = {10, 20, 30, 40};
vector<int> result(v1.size());

// 使用 plus 函数对象对应元素相加
// transform 以 v1 和 v2 为输入，以 result 为输出，对每对对应元素应用 plus<int>() 操作
// plus<int>() 会将两个参数相加并返回和
transform(v1.begin(), v1.end(), v2.begin(), result.begin(), plus<int>());
// 结果：{11, 22, 33, 44}

// 使用 multiplies 函数对象对应元素相乘
// multiplies<int>() 会将两个参数相乘并返回乘积
transform(v1.begin(), v1.end(), v2.begin(), result.begin(), multiplies<int>());
// 结果：{10, 40, 90, 160}

// 使用 accumulate 计算乘积
// 这里用 multiplies<int>() 替代默认的加法操作，实现连乘
int product = accumulate(v1.begin(), v1.end(), 1, multiplies<int>());
// 结果：24 (= 1*1*2*3*4)
```

4.3.4 创建自定义函数对象

```
// 自定义函数对象：计算平方
struct Square {
    // 重载函数调用运算符，使 Square 对象可以像函数一样被调用
    int operator()(int x) const {
        return x * x;
    }
};

vector<int> nums = {1, 2, 3, 4, 5};
vector<int> squares(nums.size());

// 使用自定义函数对象
// 创建一个 Square 对象并将其传给 transform
// transform 会对 nums 中每个元素调用 Square 对象，计算其平方值
transform(nums.begin(), nums.end(), squares.begin(), Square());
// 结果：{1, 4, 9, 16, 25}

// 带状态的函数对象
class Accumulator {
private:
    int sum; // 状态：保存累计和
public:
```

```
    // 构造函数初始化状态
    Accumulator() : sum(0) {}

    // 函数调用运算符：累加并返回当前和
    int operator()(int x) {
        sum += x;
        return sum;
    }

    // 获取当前累计结果
    int getSum() const {
        return sum;
    }
};

// 使用带状态的函数对象
vector<int> cumulative(nums.size());
Accumulator acc;
// transform 调用 acc 处理 nums 中每个元素，acc 会维护累计和
transform(nums.begin(), nums.end(), cumulative.begin(), ref(acc));
// 结果：{1, 3, 6, 10, 15}
// 注：使用 ref() 包装函数对象以传引用，确保状态被正确累计
```

4.3.5 绑定参数与函数组合

```
#include <functional>

// 使用 bind 创建新的函数对象
// bind 将第一个参数 (multiplies<int>()) 与常量 5 绑定，生成新的函数
auto times5 = bind(multiplies<int>(), placeholders::_1, 5);
// times5(x) 等价于 x*5

vector<int> v = {1, 2, 3, 4};
vector<int> result(v.size());

// 使用绑定后的函数对象
transform(v.begin(), v.end(), result.begin(), times5);
// 结果：{5, 10, 15, 20}

// 更复杂的函数组合
vector<int> nums = {1, -2, 3, -4, 5};

// 使用 lambda 表达式与 accumulate 组合
// 这个 lambda 仅累加正数，忽略负数
int positive_sum = accumulate(nums.begin(), nums.end(), 0,
                              [](int sum, int x) {
                                  return sum + (x > 0 ? x : 0);
                              });
// positive_sum = 9 (= 1 + 3 + 5)
```

4.3.6 C++14 透明函数对象

C++14 引入了透明的函数对象，无需指定类型，编译器可以从上下文推导：

```
// C++14 特性：透明函数对象
sort(v.begin(), v.end(), greater<>()); // 不指定类型，自动推导
```

```
sort(v.begin(), v.end(), less<>());     // 不指定类型，自动推导

// 这样写可以处理不同类型的比较
vector<string> vs = {"apple", "Orange", "banana"};
// 使用 less<> 可以直接比较不同类型
sort(vs.begin(), vs.end(), less<>()); // 默认字典序
```

4.3.7 函数包装器 std::function

std::function 是一个通用的函数包装器，可以存储、复制和调用任何可调用目标：

```
#include <functional>

// 创建一个接受两个 int 并返回 int 的函数类型
function<int(int, int)> op;

// 存储函数指针
int add(int a, int b) { return a + b; }
op = add;
int result1 = op(2, 3); // 结果: 5

// 存储函数对象
op = multiplies<int>();
int result2 = op(2, 3); // 结果: 6

// 存储 lambda 表达式
op = [](int a, int b) { return a - b; };
int result3 = op(5, 3); // 结果: 2

// 用于回调函数
void processNumbers(vector<int>& v, function<void(int&)> processor) {
    for (auto& num : v) {
        processor(num);
    }
}

// 使用不同的处理函数
vector<int> numbers = {1, 2, 3, 4, 5};
processNumbers(numbers, [](int& x) { x *= 2; });  // 所有元素乘 2
// 结果: {2, 4, 6, 8, 10}
```

4.4 数值处理函数

使用前需包含头文件 #include <cmath>

STL 提供了丰富的数学函数，这些函数广泛用于科学计算、几何处理和数值分析等场景。

4.4.1 基本数学函数

```
// 基本运算
abs(x)        // 整数绝对值，参数和返回值类型相同
fabs(x)       // 浮点数绝对值，返回 double 类型
sqrt(x)       // 平方根，参数可以是任何数值类型，返回浮点类型
cbrt(x)       // 立方根，参数可以是任何数值类型，返回浮点类型
hypot(x, y)   // 计算 sqrt(x*x + y*y)，避免中间溢出
```

```
// 指数和对数
pow(x, y)     // x 的 y 次幂，两个参数可以是不同类型
exp(x)        // e 的 x 次幂
log(x)        // 自然对数 (以 e 为底)，参数必须为正
log10(x)      // 以 10 为底的对数，参数必须为正
log2(x)       // 以 2 为底的对数，参数必须为正
```

4.4.2 取整函数

```
ceil(x)       // 向上取整，返回大于等于 x 的最小整数
floor(x)      // 向下取整，返回小于等于 x 的最大整数
round(x)      // 四舍五入到最接近的整数
trunc(x)      // 截断取整（向 0 取整），丢弃小数部分
```

4.4.3 三角函数和双曲函数

```
// 三角函数（参数为弧度）
sin(x)        // 正弦
cos(x)        // 余弦
tan(x)        // 正切
asin(x)       // 反正弦，参数范围 [-1,1]，返回 [- /2, /2]
acos(x)       // 反余弦，参数范围 [-1,1]，返回 [0, ]
atan(x)       // 反正切，返回 [- /2, /2]
atan2(y, x)   // 计算 y/x 的反正切，考虑象限，返回 [- , ]

// 双曲函数
sinh(x)       // 双曲正弦
cosh(x)       // 双曲余弦
tanh(x)       // 双曲正切
asinh(x)      // 反双曲正弦
acosh(x)      // 反双曲余弦，参数必须 1
atanh(x)      // 反双曲正切，参数范围 (-1,1)
```

4.4.4 特殊数值判断和操作

```
isnan(x)      // 检查 x 是否为 NaN(不是一个数)，返回 bool
isinf(x)      // 检查 x 是否为无穷大，返回 bool
isfinite(x)   // 检查 x 是否为有限值，返回 bool
signbit(x)    // 检查 x 的符号位是否被设置（是否为负），返回 bool
copysign(x,y) // 返回带有 y 符号的 x 的绝对值
nextafter(x,y) // 返回 x 向着 y 方向的下一个可表示的数
```

4.4.5 实际使用示例

```
#include <cmath>
#include <iostream>
#include <iomanip>

// 基本计算
double a = sqrt(16);              // 计算 16 的平方根: 4.0
double b = pow(2, 3);             // 计算 2 的 3 次方: 8.0
double c = log10(100);            // 计算 log (100): 2.0
```

```cpp
// 取整操作
double d = ceil(3.14);            // 向上取整: 4.0
double e = floor(3.14);           // 向下取整: 3.0
double f = round(3.5);            // 四舍五入: 4.0
double g = trunc(-2.7);           // 截断取整: -2.0

// 三角函数（注意使用弧度）
double h = sin(M_PI / 2);         // sin(90°): 1.0
double i = cos(M_PI);             // cos(180°): -1.0
double j = atan2(1, 1);           // 计算 1/1 的反正切: /4

// 特殊值判断
double inf = 1.0 / 0.0;           // 创建无穷大值
double nan_val = 0.0 / 0.0;       // 创建 NaN 值
bool is_inf = isinf(inf);         // 检查是否为无穷大: true
bool is_nan = isnan(nan_val);     // 检查是否为 NaN: true
bool is_finite = isfinite(inf);   // 检查是否为有限值: false

// 高精度浮点数比较
// 浮点数不应直接用 == 比较，而应考虑误差范围
const double epsilon = 1e-9;      // 误差容忍度
bool is_equal = fabs(a - 4.0) < epsilon;  // 检查 a 是否等于 4.0: true

// 实际应用：计算两点之间的欧几里得距离
double distance(double x1, double y1, double x2, double y2) {
    // hypot 函数避免了中间计算可能的溢出问题
    return hypot(x2 - x1, y2 - y1);
}
double dist = distance(0, 0, 3, 4);  // 计算原点到 (3,4) 的距离: 5.0

// 使用三角函数计算角度（弧度转角度）
double angle_rad = atan2(4, 3);       // 计算向量 (3,4) 与 x 轴的角度（弧度）
double angle_deg = angle_rad * 180 / M_PI;  // 将弧度转换为角度: 约 53.13°
```

4.4.6 复数相关函数

对于复数操作，需要包含头文件 <complex>:

```cpp
#include <complex>

// 定义复数
complex<double> z1(3, 4);  // 实部 3, 虚部 4
complex<double> z2(1, 2);

// 复数运算
auto sum = z1 + z2;         // 复数加法: (4,6)
auto product = z1 * z2;     // 复数乘法: (3*1-4*2, 3*2+4*1)=(-5,10)

// 复数函数
double mag = abs(z1);       // 复数的模: 5
double arg = arg(z1);       // 复数的辐角
complex<double> conj = conj(z1);  // 共轭复数: (3,-4)
complex<double> sqr = sqrt(z1);   // 复数的平方根
```

4.5 字符串处理函数

C++ 提供了两种处理字符串的方式：C 风格的字符串函数和 C++ string 类。对于现代 C++ 编程，推荐使用 C++ string 类，它更安全，更灵活。

4.5.1 C 风格字符串函数 (<cstring>)

```
// 字符串长度和比较
strlen(str)                  // 计算字符串长度（不包括 null 结束符）
strcmp(s1, s2)               // 比较字符串，相等返回 0，s1<s2 返回负值，s1>s2 返回正值
strncmp(s1, s2, n)           // 比较最多 n 个字符

// 字符串复制和连接
strcpy(dest, src)            // 将 src 复制到 dest，包括 null 结束符
strncpy(dest, src, n)        // 复制最多 n 个字符，不保证 null 结束符
strcat(dest, src)            // 将 src 连接到 dest 末尾
strncat(dest, src, n)        // 连接最多 n 个字符

// 字符串搜索
strchr(str, c)               // 查找字符 c 首次出现的位置，返回指针或 NULL
strrchr(str, c)              // 查找字符 c 最后出现的位置，返回指针或 NULL
strstr(str, substr)          // 查找子串 substr 首次出现的位置，返回指针或 NULL
strpbrk(str, charset)        // 查找 charset 中任意字符首次出现的位置

// 字符串分割和转换
strtok(str, delim)           // 按分隔符分割字符串，返回下一个标记或 NULL
atoi(str)                    // 将字符串转换为 int
atof(str)                    // 将字符串转换为 double
```

详细示例：

```
#include <cstring>
#include <iostream>

// 基本字符串操作
char s1[100] = "Hello";
char s2[] = "World";

// 获取字符串长度
size_t len = strlen(s1);         // 返回 5（不计算结束符 '\0'）

// 字符串比较
int cmp = strcmp(s1, s2);        // 返回负值，因为 'H' 的 ASCII 码小于 'W'
bool is_equal = (strcmp(s1, "Hello") == 0); // 检查 s1 是否等于 "Hello"：true

// 字符串复制和连接
strcpy(s1 + 5, " ");             // 将空格复制到 s1[5] 位置，s1 变为 "Hello "
strcat(s1, s2);                  // 将 s2 连接到 s1 末尾，s1 变为 "Hello World"

// 字符串搜索
char* found = strchr(s1, 'o');  // 查找 'o' 首次出现位置，found 指向 "o World"
char* last_o = strrchr(s1, 'o'); // 查找 'o' 最后出现位置，指向 "orld" 中的 'o'
char* substr = strstr(s1, "World"); // 查找 "World"，substr 指向 "World"

// 字符串分割
char csv[] = "apple,banana,orange";
char* token = strtok(csv, ","); // 获取第一个标记：apple
while (token != nullptr) {
    std::cout << token << std::endl; // 输出当前标记
    token = strtok(nullptr, ",");    // 获取下一个标记
}
// 输出：
```

```
// apple
// banana
// orange

// 注意：strtok 会修改原字符串，插入 null 字符 ('\0') 来分隔标记
// 如果需要保留原字符串，应事先创建副本

// 安全的字符串复制（防止缓冲区溢出）
char dest[10];
strncpy(dest, "This is a very long string", 9); // 只复制前 9 个字符
dest[9] = '\0'; // 手动添加结束符确保安全
// dest 包含"This is a"

// 字符串转数字
int num = atoi("123");          // 将字符串转换为整数：123
double dbl = atof("3.14159");   // 将字符串转换为浮点数：3.14159
```

注意事项：- C 风格字符串函数不检查边界，可能导致缓冲区溢出和安全问题 - 使用时必须确保目标数组有足够空间 - 函数如 `strncpy` 不会自动添加 null 结束符，需手动添加 - 推荐使用 C++ string 类替代这些函数，更安全可靠

4.5.2 C++ string 类函数 (<string>)（见第 2 节）

注意事项：- C++17 标准库没有内建的字符串 split、join 功能，需要自行实现 - C++20 引入了 `starts_with`、`ends_with` 方法和 `<format>` 库，但在竞赛中可能不可用 - 处理大量字符串连接时，使用 `stringstream` 或 `reserve`+`append` 比反复 `+` 更高效 - 字符串查找失败时返回 `string::npos`，记得检查这种情况 - C++ 中不同于某些语言，字符串是可变的，许多操作会直接修改原字符串

4.6 时间复杂度总结

下面是常见 STL 函数的时间复杂度：

4.6.1 容器操作

操作	vector	deque	list	set/map	unordered_set/map
随机访问 []	O(1)	O(1)	O(N)	N/A	N/A
插入/删除（末尾）	O(1)*	O(1)	O(1)	O(logN)	O(1) 平均
插入/删除（开头）	O(N)	O(1)	O(1)	O(logN)	O(1) 平均
插入/删除（中间）	O(N)	O(N)	O(1)**	O(logN)	O(1) 平均
查找	O(N)	O(N)	O(N)	O(logN)	O(1) 平均

* 对于 vector，如果需要扩容则为 O(N) ** 对于 list，如果已知位置（迭代器）则为 O(1)，否则需要 O(N) 查找

4.6.2 常用算法

算法	时间复杂度	空间复杂度
sort	O(N log N)	O(log N)
stable_sort	O(N log N)	O(N) 或 O(1)
nth_element	O(N) 平均	O(1)
binary_search	O(log N)	O(1)
merge	O(N+M)	O(N+M)
accumulate	O(N)	O(1)
find/count	O(N)	O(1)
min/max_element	O(N)	O(1)
copy/transform	O(N)	O(1)
unique	O(N)	O(1)
next_permutation	O(N)	O(1)

4.7 实用技巧与最佳实践

4.7.1 1. 使用合适的容器

- 需要频繁随机访问元素：vector
- 需要频繁在两端操作：deque
- 需要频繁在中间插入/删除：list
- 需要保持元素排序：set, map
- 需要快速查找但不关心顺序：unordered_set, unordered_map

4.7.2 2. 迭代器优化

```
// 避免重复计算 end()
vector<int> v = {1, 2, 3, 4, 5};
// 较差的方式
for (int i = 0; i < v.size(); ++i) {
    // v.size() 每次循环都会调用
}

// 更好的方式
for (vector<int>::size_type i = 0, size = v.size(); i < size; ++i) {
    // size 只计算一次
}

// 现代 C++ 的方式
for (const auto& item : v) {
    // 使用范围 for 循环
}

// 迭代器失效问题
auto it = v.begin();
v.erase(it);      // it 现在无效
it = v.begin();   // 重新获取有效迭代器
```

4.7.3 3. 算法组合技巧

```
// 寻找前 k 大/小元素
vector<int> v = {5, 2, 8, 1, 9, 3, 7, 6, 4};
int k = 3;

// 第一种方法：排序后取前 k 个
sort(v.begin(), v.end());
// 前 k 小：v[0] 到 v[k-1]

// 第二种方法：使用 nth_element 更高效
nth_element(v.begin(), v.begin() + k - 1, v.end());
// 现在前 k 个元素是最小的 k 个（不一定有序）

// 第三种方法：使用 partial_sort
partial_sort(v.begin(), v.begin() + k, v.end());
// 前 k 个元素是有序的最小 k 个

// 查找特定范围的元素
auto range = equal_range(v.begin(), v.end(), 5);
bool found = (range.first != range.second);
```

4.7.4 4. 使用合适的函数

```
// 计算乘积：使用 accumulate 而非循环
vector<int> v = {1, 2, 3, 4, 5};
int product = accumulate(v.begin(), v.end(), 1, multiplies<int>());

// 高效寻找最大值和位置
vector<int> v = {5, 2, 8, 1, 9, 3};
auto max_it = max_element(v.begin(), v.end());
int max_value = *max_it;
int max_position = distance(v.begin(), max_it);

// 去除重复元素（需要先排序）
sort(v.begin(), v.end());
auto new_end = unique(v.begin(), v.end());
v.erase(new_end, v.end());
```

4.7.5 5. 性能优化

```
// 预分配容器空间
vector<int> v;
v.reserve(10000); // 避免频繁重新分配内存
for (int i = 0; i < 10000; ++i) {
    v.push_back(i); // 不会导致重新分配
}

// 批量插入
vector<int> src = {1, 2, 3, 4, 5};
vector<int> dest;
// 比循环中 push_back 更高效
dest.insert(dest.end(), src.begin(), src.end());

// 交换技巧清空容器
vector<int> v = {1, 2, 3, 4, 5};
vector<int>().swap(v); // 高效清空并释放内存
```

4.7.6 6. 常见错误避免

```
// 避免在循环中调整容器大小
vector<int> v;
for (int i = 0; i < 10000; ++i) {
    v.push_back(i);  // 可能导致多次重新分配
}
// 修正：使用 reserve() 或 resize() 预先分配空间

// 避免迭代器失效
vector<int> v = {1, 2, 3, 4, 5};
// 错误方式
for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it % 2 == 0) {
        v.erase(it);  // 迭代器失效!
    }
}
// 正确方式
for (auto it = v.begin(); it != v.end(); ) {
```

```
    if (*it % 2 == 0) {
        it = v.erase(it); // erase 返回下一个有效迭代器
    } else {
        ++it;
    }
}

// 或使用 remove_if 和 erase 的组合
v.erase(
    remove_if(v.begin(), v.end(), [](int n){ return n % 2 == 0; }),
    v.end()
);
```

5 手搓算法

5.1 5.1 快速排序和归并排序

```
int part(int low, int high){ //以 price 为索引
    int i=low, j=high, pivot = price[low];
    while(i<j){
        while(i<j && price[j]>pivot){
            j--;
        }
        if(i<j){
            std::swap(price[i], price[j]);
            std::swap(value[i], value[j]);
            i++;
        }
        while(i<j&&price[i]<=pivot){
            i++;
        }
        if(i<j){
            std::swap(price[i],price[j]);
            std::swap(value[i],value[j]);
            j--;
        }
    }
    return i;
}

void sort(int low, int high){
    int mid;
    if(low<high){
        mid = part(low,high);
        sort(low,mid-1);
        sort(mid+1,high);
    }
}
```

```
void Merge(int a[], int left, int mid, int right){
    int temp[right - left + 1];                    //临时数组用于存储排序时的数
    int i = left;                                  //分成两块 i 指向左边的数字 j 指向右边的数字
    int j = mid + 1;
    int k = 0;                                     //k 用于存储数字到临时数组
```

```
    while( i <= mid && j <= right ){
        if(a[i] < a[j])                         //永远都是 i 和 j 指向的数进行比较
            temp[k++] = a[i++];                     //谁小，谁就先放到临时数组中
        else
            temp[k++] = a[j++];
    }

    while( i <= mid )                               //如果左边还有数没放上去，就依次放上去
        temp[k++] = a[i++];
    while( j <= right )                             //如果是右边还有同上
        temp[k++] = a[j++];

    for(int m = left, n = 0; m <= right; m++, n++)//读取临时数组中的数
        a[m] = temp[n];
}

void Merge_Sort(int a[], int left, int right){
    if( left == right )
        return;
    int mid = (left + right)/2;
    //递归拆分成较小规模子序列排序
    Merge_Sort(a, left, mid);
    Merge_Sort(a, mid + 1, right);
    Merge(a, left, mid, right);       //合并较小规模问题解
}
```

5.2 5.2 矩阵

- 矩阵的定义

```
struct Matrix {
    int rows, cols;
    vector<vector<long long>> data;
    Matrix(int r, int c) : rows(r), cols(c) {
        data.resize(r, vector<long long>(c, 0));
    }
};
```

- 矩阵乘法

```
// 矩阵乘法
Matrix multiply(const Matrix& A, const Matrix& B, long long mod = 1e9 + 7) {
    if (A.cols != B.rows) {
        throw invalid_argument(" 矩阵维度不匹配，无法相乘");
    }
    Matrix C(A.rows, B.cols);

    for (int i = 0; i < A.rows; i++) {
        for (int j = 0; j < B.cols; j++) {
            for (int k = 0; k < A.cols; k++) {
                C.data[i][j] = (C.data[i][j] + A.data[i][k] * B.data[k][j]) % mod;
            }
        }
    }
    return C;
}
```

- 快速幂

```
Matrix matrixPow(Matrix A, long long n, long long mod = 1e9 + 7) {// 矩阵快速幂（O(log n×m³) 时间复杂度）
    if (A.rows != A.cols) {
        throw invalid_argument(" 只有方阵才能进行幂运算");
    }
    int m = A.rows;
    // 创建单位矩阵作为结果
    Matrix result(m, m);
    for (int i = 0; i < m; i++) {
        result.data[i][i] = 1;
    }
    // 快速幂算法
    while (n > 0) {
        if (n & 1) { // 如果 n 的二进制最低位为 1
            result = multiply(result, A, mod);
        }
        A = multiply(A, A, mod);
        n >>= 1; // n 右移一位，相当于 n = n/2
    }
    return result;
}
```

- 快速递推（以计算斐波那契数列为例）

```
//斐波那契数列可以用矩阵形式表示：
//[F(n+1), F(n)] = [[1, 1], [1, 0]] × [F(n), F(n-1)]
// 使用矩阵快速幂计算斐波那契数列第 n 项
long long fibonacci(long long n, long long mod = 1e9 + 7) {
    if (n <= 1) return n;

    // 定义转移矩阵 [[1,1],[1,0]]
    Matrix A(2, 2);
    A.data = {{1, 1}, {1, 0}};

    // 计算矩阵的 n-1 次幂
    Matrix result = matrixPow(A, n - 1, mod);

    // 初始状态 [F(1), F(0)] = [1, 0]
    // 结果为 result × [1, 0]^T
    return result.data[0][0]; // F(n) = result[0][0]*F(1) + result[0][1]*F(0) = result[0][0]*1 + result[0][1]*0
}
```

5.3 5.3 快速幂（指数分解）

- 迭代式

```
long long quickPow(long long a, long long b) {
    if (b == 0) return 1;

    long long half = quickPow(a, b / 2);

    if (b % 2 == 0) {
        return half * half;
    } else {
        return half * half * a;
    }
}
```

- 递推式

```
// 计算 a^b 的值
long long quickPow(long long a, long long b) {
    long long res = 1; // 初始化结果为 1

    while (b > 0) {
        if (b & 1) {  // 如果 b 的当前最低位为 1
            res = res * a;  // 将当前的 a 累乘到结果中
        }
        a = a * a;  // a 平方
        b >>= 1;  // b 右移一位，即 b = b/2
    }

    return res;
}
```

- 带取模的

```
// 计算 (a^b) % mod
long long quickPowMod(long long a, long long b, long long mod) {
    a %= mod;
    long long res = 1;

    while (b > 0) {
        if (b & 1) {
            res = (res * a) % mod;
        }
        a = (a * a) % mod;
        b >>= 1;
    }

    return res;
}
```

5.4 5.4 高精度运算和筛

- 长整数加

```
string addStrings(string num1, string num2) {
    string result = "";
    int carry = 0;
    int i = num1.length() - 1;
    int j = num2.length() - 1;

    while (i >= 0 || j >= 0 || carry > 0) {
        int sum = carry;
        if (i >= 0) sum += (num1[i--] - '0');
        if (j >= 0) sum += (num2[j--] - '0');

        carry = sum / 10;
        result = to_string(sum % 10) + result;
    }

    return result;
}
```

- 长整数模

```
const int MOD = 1e9 + 7;
// 模运算下的加法
int addMod(int a, int b) {
    return (a + b) % MOD;
}
// 模运算下的乘法
int mulMod(int a, int b) {
    return (1LL * a * b) % MOD;  // 使用 long long 避免溢出
}
// 模运算下的快速幂
int powMod(int base, int exponent) {
    int result = 1;
    base %= MOD;

    while (exponent > 0) {
        if (exponent & 1) {
            result = mulMod(result, base);
        }
        exponent >>= 1;
        base = mulMod(base, base);
    }
    return result;
}
```

- 高精度平方

```
// 高精度计算示例 - 二分法求平方根
double sqrt(double x, double epsilon = 1e-9) {
    double low = 0, high = max(1.0, x);

    while (high - low > epsilon) {
        double mid = low + (high - low) / 2;
        if (mid * mid > x) {
            high = mid;
        } else {
            low = mid;
        }
    }

    return low;
}
```

- 求大素数

```
// 埃氏筛法求素数
vector<bool> sieveOfEratosthenes(int n) {
    vector<bool> isPrime(n+1, true);
    isPrime[0] = isPrime[1] = false;  // 0 和 1 不是素数

    for (int i = 2; i * i <= n; i++) {
        if (isPrime[i]) {
            // 将 i 的所有倍数标记为非素数
            for (int j = i * i; j <= n; j += i) {
                isPrime[j] = false;
            }
```

```
        }
    }

    return isPrime;
}

// 线性筛法 （欧拉筛）求素数
vector<int> linearSieve(int n) {
    vector<bool> isPrime(n+1, true);
    vector<int> primes;
    isPrime[0] = isPrime[1] = false;

    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) {
            primes.push_back(i);
        }

        // 用已知的素数去筛
        for (int j = 0; j < primes.size() && i * primes[j] <= n; j++) {
            isPrime[i * primes[j]] = false;
            if (i % primes[j] == 0) break;  // 保证每个合数只被其最小质因子筛掉
        }
    }

    return primes;
}
```

- 高精度 2 除法

```
static inline i64 floor_div(i64 a, i64 b) {
    i64 q = a / b;
    i64 r = a % b;
    if (r != 0 && ((r > 0) != (b > 0))) --q; //负数反转
    return q;
}
static inline i64 ceil_div(i64 a, i64 b) {
    return -floor_div(-a, b);
}

static inline i64 floor_div_pow2(i64 a, long long k) {
    if (k >= 62) {  //除数位数太大直接归 0
        if (a >= 0) return 0;
        return -1;
    }
    i64 d = 1LL << k; //位移生成除数
    return floor_div(a, d);
}
static inline i64 ceil_div_pow2(i64 a, long long k) {
    if (k >= 62) {
        if (a <= 0) return 0;
        return 1;
    }
    i64 d = 1LL << k;
    return ceil_div(a, d);
}
```

- 高精度求大组合数

```cpp
// 动态规划法计算组合数
vector<vector<int>> generateCombinations(int n, int mod = 1e9 + 7) {
    vector<vector<int>> C(n+1, vector<int>(n+1, 0));

    for (int i = 0; i <= n; i++) {
        C[i][0] = 1;  // 从 i 个元素中选 0 个的方法数为 1
        for (int j = 1; j <= i; j++) {
            // C(i,j) = C(i-1,j-1) + C(i-1,j)
            C[i][j] = (C[i-1][j-1] + C[i-1][j]) % mod;
        }
    }

    return C;
}

// 逆元法计算大组合数（适用于 mod 为质数的情况）
int combination(int n, int k, int mod) {
    if (k > n) return 0;
    if (k == 0 || k == n) return 1;

    // 计算阶乘及其逆元
    vector<long long> fact(n+1), invFact(n+1);
    fact[0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = (fact[i-1] * i) % mod;
    }

    // 费马小定理：a^(p-1)  1 (mod p) => a^(p-2)  a^(-1) (mod p)
    invFact[n] = fastPowMod(fact[n], mod - 2, mod);
    for (int i = n - 1; i >= 0; i--) {
        invFact[i] = (invFact[i+1] * (i+1)) % mod;
    }

    // C(n,k) = n! / (k! * (n-k)!)
    return (fact[n] * ((invFact[k] * invFact[n-k]) % mod)) % mod;
}
```

5.5 5.5 素数专用算法

- 分段筛

```cpp
// 分段筛法，计算区间 [l, r] 中的素数
vector<bool> segmentedSieve(long long l, long long r) {
    // 生成 sqrt(r) 以内的所有素数
    int sqrtR = sqrt(r);
    vector<bool> basePrime = sieveOfEratosthenes(sqrtR);
    vector<int> primes;
    for (int i = 2; i <= sqrtR; i++) {
        if (basePrime[i])
            primes.push_back(i);
    }
    // 分段标记 [l, r] 区间内的合数
    vector<bool> isPrime(r - l + 1, true);

    // 处理 0 和 1 的特殊情况
    if (l <= 1)
        isPrime[1 - l] = false;
```

```
    // 用小素数筛掉区间内的合数
    for (int p : primes) {
        // 找到区间内 p 的第一个倍数
        long long start = max(p * p, ((l + p - 1) / p) * p);

        for (long long j = start; j <= r; j += p) {
            isPrime[j - l] = false;
        }
    }
    return isPrime;
}
// 获取区间内的素数列表
vector<long long> segmentedPrimeList(long long l, long long r) {
    vector<bool> isPrime = segmentedSieve(l, r);
    vector<long long> result;

    for (long long i = 0; i < r - l + 1; i++) {
        if (isPrime[i])
            result.push_back(l + i);
    }

    return result;
}
```

- 整数分解

```
vector<int> factorize(int n) {
    vector<int> factors;
    vector<bool> isPrime = sieveOfEratosthenes(sqrt(n) + 1);
    vector<int> primes;

    for (int i = 2; i <= sqrt(n); i++) {
        if (isPrime[i])
            primes.push_back(i);
    }

    for (int p : primes) {
        while (n % p == 0) {
            factors.push_back(p);
            n /= p;
        }
    }

    if (n > 1) // 如果 n 是素数或者有大于 sqrt(n) 的素因子
        factors.push_back(n);

    return factors;
}
```

5.6 5.6 中国剩余定理 (CRT)

中国剩余定理用于求解如下形式的同余方程组：x a (mod m) x a (mod m) ...x a (mod m) 其中 m , m , ..., m 两两互质。

```
// 中国剩余定理
int CRT(vector<int>& remainders, vector<int>& moduli) {
    int n = remainders.size();
```

```
    int product = 1;
    for (int mod : moduli) {
        product *= mod;
    }

    int result = 0;
    for (int i = 0; i < n; i++) {
        int m = moduli[i];
        int pp = product / m;
        int inv = modInverse(pp, m);

        result = (result + 1LL * remainders[i] * pp % product * inv % product) % product;
    }

    return result;
}
```

5.7 5.7 扩展中国剩余定理 (EXCRT)

当模数不互质时，需要使用扩展 CRT。

```
// 合并两个方程: x ≡ a1 (mod m1) 和 x ≡ a2 (mod m2)
pair<int, int> merge(int a1, int m1, int a2, int m2) {
    int x, y;
    int g = extGcd(m1, m2, x, y);

    if ((a2 - a1) % g != 0) {
        return {-1, -1};  // 无解
    }

    int lcm = m1 / g * m2;  // 最小公倍数
    int mod_val = ((a2 - a1) / g * x % (m2 / g) + (m2 / g)) % (m2 / g);
    int ans = (a1 + m1 * mod_val) % lcm;

    return {ans, lcm};
}

// 扩展中国剩余定理
int EXCRT(vector<int>& remainders, vector<int>& moduli) {
    int n = remainders.size();
    int a = remainders[0], m = moduli[0];

    for (int i = 1; i < n; i++) {
        auto [new_a, new_m] = merge(a, m, remainders[i], moduli[i]);
        if (new_a == -1) return -1;  // 无解
        a = new_a;
        m = new_m;
    }

    return a;
}
```

5.8 5.8 组合数学

- 计算排列数

```cpp
// 计算排列数 P(n,k)
long long permutation(int n, int k) {
    long long result = 1;
    for (int i = 0; i < k; i++) {
        result *= (n - i);
    }
    return result;
}
```

- 计算组合数

从 n 个不同元素中取出 k 个元素，不考虑顺序的选择方式数量称为组合数，记作 $\mathrm{C(n,k)}$ 或 $\binom{n}{k}$。

$$C(n,k) = \binom{n}{k} = \frac{P(n,k)}{k!} = \frac{n!}{k!(n-k)!}$$

组合数满足以下性质：1. $\mathrm{C(n,k)} = \mathrm{C(n,n\text{-}k)}$ 2. $\mathrm{C(n,0)} = \mathrm{C(n,n)} = 1$ 3. $\mathrm{C(n,k)} = \mathrm{C(n\text{-}1,k\text{-}1)} + \mathrm{C(n\text{-}1,k)}$ （杨辉三角性质）

```cpp
// 计算组合数 C(n,k)（可能溢出）
long long combination(int n, int k) {
    if (k > n) return 0;
    if (k > n - k) k = n - k;  // 利用 C(n,k) = C(n,n-k) 优化

    long long result = 1;
    for (int i = 1; i <= k; i++) {
        result = result * (n - k + i) / i;  // 每次乘以 (n-k+i) 再除以 i
    }

    return result;
}
```

- 带模计算组合数

```cpp
// Lucas 定理计算 C(n,m) % p，其中 p 为素数
long long lucas(long long n, long long m, int p) {
    if (m == 0) return 1;
    return C(n % p, m % p, p) * lucas(n / p, m / p, p) % p;
}

// 计算小组合数 C(n,m) % p
long long C(int n, int m, int p) {
    if (m > n) return 0;

    long long res = 1;
    for (int i = n - m + 1; i <= n; i++) {
        res = res * i % p;
    }
    for (int i = 1; i <= m; i++) {
        res = res * pow_mod(i, p - 2, p) % p;  // 逆元
    }

    return res;
}
```

- 计算卡特兰数

卡特兰数是组合数学中的一个数列，可表示多种计数问题的解。第 n 个卡特兰数的计算公式为：

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

递推关系：$C_0 = 1, C_{n+1} = \sum_{i=0}^{n} C_i \times C_{n-i}$

```
// 计算第 n 个卡特兰数
long long catalan(int n) {
    // 使用组合数公式：C(n) = C(2n,n) / (n+1)
    return combination(2 * n, n) / (n + 1);
}

// 使用递推公式计算卡特兰数（避免溢出）
vector<long long> catalanDP(int n) {
    vector<long long> cat(n + 1, 0);
    cat[0] = 1;

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < i; j++) {
            cat[i] += cat[j] * cat[i - j - 1];
        }
    }

    return cat;
}
```

卡特兰数可以计算：1. n 个节点可以组成多少种不同的二叉搜索树 2. n 对括号的合法序列数量 3. n 个点的凸多边形，通过对角线划分为三角形的不同方法数 4. 从 (0,0) 到 (n,n) 的格点路径，不越过对角线的路径数

5.9 5.9 公约数和公倍数

- gcd 算法

```
//递归
int gcd(int a, int b) {
    if (b == 0) return a;  // 基础情况：当 b 为 0 时，a 即为最大公约数
    return gcd(b, a % b);  // 递归调用：将 b 和 a%b 作为新的输入
}
//迭代
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;  // 计算余数
        a = temp;   // 更新 a 为原来的 b
    }
    return a;  // 当 b 为 0 时，a 即为最大公约数
}
```

- 最小公倍数

```
int lcm(int a, int b) {
    return a / gcd(a, b) * b;  // 先除后乘，避免溢出
}
```

- 分数加法

```
struct Fraction {
    int num, den;  // 分子 (numerator) 和分母 (denominator)

    // 约分，使分数保持最简形式
    void simplify() {
        int g = gcd(abs(num), den);
        num /= g;
        den /= g;
    }

    // 分数加法
    Fraction add(Fraction other) {
        int lcm_val = lcm(den, other.den);
        int new_num = num * (lcm_val / den) + other.num * (lcm_val / other.den);
        Fraction result = {new_num, lcm_val};
        result.simplify();
        return result;
    }
};
```

5.10 5.10 手搓栈 (U416829)

```
#include <iostream>
#include <climits>
#include <vector>
#include <deque>
#include <algorithm>
using namespace std;

using namespace std;

using ll = long long;

struct package {
    ll count;
    ll single;
    package(ll c = 0, ll s = 0) : count(c), single(s) {}
};

class Stack {
private:
    // 用 deque 连续存放“段”(从底到顶: front -> ... -> back)
    deque<package> runs;
    ll sum_ = 0;              // 栈内元素总和 (addall O(1))
public:
    Stack() = default;

    // ——基础接口 (与题意一致)——
    // 单次 push (为兼容保留; 性能靠 push_repeat)
    void push(ll v) {
        if (!runs.empty() && runs.back().single == v) {
            runs.back().count += 1;
        } else {
            runs.emplace_back(1, v);
        }
        sum_ += v;
```

```
}

// 批量 push：把同一数值 v 压入 times 次（O(1)）
void push_repeat(ll v, ll times) {
    if (times <= 0) return;
    if (!runs.empty() && runs.back().single == v) {
        runs.back().count += times;
    } else {
        runs.emplace_back(times, v);
    }
    sum_ += v * times;
}

// 返回栈顶元素值；空栈抛异常或按题面自行处理
ll top() const {
    if (isEmpty()) throw runtime_error("top() on empty stack");
    return runs.back().single;
}

// 单次 pop（为兼容保留；性能靠 pop_sum_k）
void pop() {
    if (isEmpty()) return;
    sum_ -= runs.back().single;
    if (--runs.back().count == 0) runs.pop_back();
}

// 批量弹出 k 个并返回它们的和（O(触达段数)）
ll pop_sum_k(ll k) {
    if (k <= 0 || isEmpty()) return 0;
    ll got = 0;
    while (k > 0 && !runs.empty()) {
        ll take = min(k, runs.back().count);
        got  += take * runs.back().single;
        sum_ -= take * runs.back().single;
        runs.back().count -= take;
        if (runs.back().count == 0) runs.pop_back();
        k -= take;
    }
    return got;
}

// 把 x 的全部元素移动到 *this 的顶部（维持整体顺序）
// 内置“小并大”与两种合并路径，减少移动与拷贝
void move_all_from(Stack& x) {
    if (&x == this || x.isEmpty()) return;

    // 谁短谁移动：减少总搬运次数
    if (x.runs.size() <= runs.size()) {
        // 直接从 x 顶部一段段“接到” this 顶部
        while (!x.runs.empty()) {
            package p = x.runs.back();
            x.runs.pop_back();
            if (!runs.empty() && runs.back().single == p.single) {
                runs.back().count += p.count;
            } else {
                runs.push_back(p);
            }
```

```
            }
        } else {
            // 反转 x 后把 this 的内容前插到 x，再交换到 this
            // 1) 反转 x 的段顺序（使原顶在 front）
            std::reverse(x.runs.begin(), x.runs.end());
            // 2) 把当前 this 的内容并到 x 的前端
            while (!runs.empty()) {
                package p = runs.back();
                runs.pop_back();
                if (!x.runs.empty() && x.runs.front().single == p.single) {
                    x.runs.front().count += p.count;
                } else {
                    x.runs.push_front(p);
                }
            }
            // 3) 交换容器，使结果在 this
            runs.swap(x.runs);
        }
        // 维护总和
        sum_ += x.sum_;
        x.sum_ = 0;
    }

    // 查询
    int size() const {
        // 元素总数 = 各段 count 之和；如需 O(1) 可额外维护 elems_
        long long s = 0;
        for (auto &p : runs) s += p.count;
        return (int)min<long long>(s, INT_MAX);
    }
    bool isEmpty() const { return runs.empty(); }
    ll addall() const { return sum_; }

    // 可选：释放容量（避免高水位常驻）
    void shrink_to_fit() {
        deque<package> tmp;
        tmp.swap(runs);
        runs.swap(tmp);
    }
};
```

5.11 5.11 两流水线 johnson 最佳分配 (U419509)

1. Johnson 法则是：

- 把任务分成两组：ai b 的放到前半段，按 ai 升序；ai>bi 的放到后半段，按 bi 降序。
- 或者说：反复在所有未安排任务中选出 **min(ai,bi)** 最小的那一个；若是来自 ai (即 ai bi)，把它放到队列**最前**；若是来自 bi (即 ai>bi)，把它放到队列**最后**。

```
#include <iostream>
#include <climits>
#include <vector>
#include <algorithm>
using namespace std;

struct Job {
    long long a, b;
};
```

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    if (!(cin >> T)) return 0;
    while (T--) {
        int n;
        cin >> n;
        vector<Job> jobs(n);
        for (int i = 0; i < n; ++i) cin >> jobs[i].a >> jobs[i].b;

        // Johnson's rule: split and sort
        vector<Job> L, R;
        L.reserve(n); R.reserve(n);
        for (auto &j : jobs) {
            if (j.a <= j.b) L.push_back(j);
            else R.push_back(j);
        }
        sort(L.begin(), L.end(), [](const Job& x, const Job& y){
            if (x.a != y.a) return x.a < y.a;
            return x.b < y.b; // 次要键，随意但稳定
        });
        sort(R.begin(), R.end(), [](const Job& x, const Job& y){
            if (x.b != y.b) return x.b > y.b; // 降序
            return x.a < y.a;
        });

        // Concatenate: L then R
        vector<Job> order;
        order.reserve(n);
        order.insert(order.end(), L.begin(), L.end());
        order.insert(order.end(), R.begin(), R.end());

        // Simulate two machines
        long long endA = 0, endB = 0;
        for (auto &j : order) {
            endA += j.a;                    // 洗发机先做完这个
            endB = max(endB, endA) + j.b; // 剪发机必须等洗发完成 & 自身空闲
        }
        cout << endB << "\n";
    }
    return 0;
}
```

6 二分查找

6.1 算法概述

【二分查找】(Binary Search) 是一种高效的查找算法，用于在【有序数组】中查找特定元素。其核心思想是每次将查找区间一分为二，通过与中间元素比较，排除一半的搜索空间，从而大幅减少查找次数。

6.2 算法设计思路

二分查找的基本思路是：1. 确定查找区间的左右边界 2. 计算区间中点，将待查找值与中点值比较 3. 根据比较结果，将搜索范围缩小到左半部分或右半部分 4. 重复上述步骤，直到找到目标值或确定目标值不存在

6.3 代码实现与解析

6.3.1 基本二分查找

```cpp
// 在有序数组 nums 中查找目标值 target，返回索引，若不存在则返回-1
int binarySearch(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size() - 1; // 注意这里是闭区间 [left, right]

    while (left <= right) { // 注意这里是 <=，因为是闭区间
        int mid = left + (right - left) / 2; // 防止整数溢出的写法

        if (nums[mid] == target) {
            return mid; // 找到目标值，返回索引
        } else if (nums[mid] < target) {
            left = mid + 1; // 目标在右半部分
        } else {
            right = mid - 1; // 目标在左半部分
        }
    }

    return -1; // 目标不存在
}
```

6.3.2 查找第一个等于目标值的位置

```cpp
// 在有序数组 nums 中查找第一个等于 target 的位置，不存在则返回-1
int findFirstEqual(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size() - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] >= target) {
            if (nums[mid] == target) {
                result = mid; // 记录当前找到的位置
            }
            right = mid - 1; // 继续在左侧查找
        } else {
            left = mid + 1;
        }
    }

    return result;
}
```

6.3.3 查找最后一个等于目标值的位置

```cpp
// 在有序数组 nums 中查找最后一个等于 target 的位置，不存在则返回-1
int findLastEqual(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size() - 1;
    int result = -1;
```

```
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] <= target) {
            if (nums[mid] == target) {
                result = mid; // 记录当前找到的位置
            }
            left = mid + 1; // 继续在右侧查找
        } else {
            right = mid - 1;
        }
    }

    return result;
}
```

6.3.4 查找第一个大于等于目标值的位置（下界）

```
// 在有序数组 nums 中查找第一个大于等于 target 的位置，如果所有元素都小于 target 则返回 nums.size()
int lowerBound(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size(); // 注意这里是开区间 [left, right)

    while (left < right) { // 注意这里是 <，因为是开区间
        int mid = left + (right - left) / 2;

        if (nums[mid] >= target) {
            right = mid; // 缩小右边界，但仍包含 mid
        } else {
            left = mid + 1;
        }
    }

    return left; // 返回下界
}
```

6.3.5 查找第一个大于目标值的位置（上界）

```
// 在有序数组 nums 中查找第一个大于 target 的位置，如果所有元素都小于等于 target 则返回 nums.size()
int upperBound(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size(); // 开区间 [left, right)

    while (left < right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] > target) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left; // 返回上界
}
```

6.4 逻辑流程图解

基本二分查找流程：

```
初始区间：[0, n-1]
  |
  v
计算中点：mid = (left + right) / 2
  |
  v
比较nums[mid]和target
  |
  +----> nums[mid] == target：返回mid
  |
  +----> nums[mid] < target：搜索区间变为 [mid+1, right]
  |
  +----> nums[mid] > target：搜索区间变为 [left, mid-1]
  |
  v
重复上述步骤直到找到target或区间为空
```

6.5 复杂度分析

- 时间复杂度：$O(\log n)$，其中 n 是数组的长度
  - 每次迭代将搜索空间减半，因此最多需要 $\log n$ 次比较
- 空间复杂度：$O(1)$，只需要常数级别的额外空间

6.6 常见的二分查找变形

二分查找有很多变形，主要用于解决以下问题：

1. 【精确查找】：查找等于目标值的元素
2. 【范围查找】：查找第一个/最后一个等于目标值的元素
3. 【边界查找】：查找第一个大于等于/大于目标值的元素
4. 【旋转数组查找】：在旋转有序数组中查找元素
5. 【二分答案】：使用二分查找确定答案范围

6.7 典型例题分析

6.7.1 例题 1：搜索插入位置（LeetCode 35）

问题描述：给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

```cpp
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size(); // 注意是开区间 [left, right)

        while (left < right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] >= target) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
```

```
        return left; // 这里等价于下界 (lower_bound)
    }
};
```

分析：- 时间复杂度：$O(\log n)$ - 空间复杂度：$O(1)$ - 核心思想：查找第一个大于等于 target 的位置，即 lowerBound 实现

6.7.2 例题 2：在排序数组中查找元素的第一个和最后一个位置（LeetCode 34）

问题描述：给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。如果数组中不存在目标值 target，返回 [-1, -1]。

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        if (nums.empty()) return {-1, -1};

        // 查找第一个等于 target 的位置
        int first = findFirst(nums, target);

        // 如果没找到，直接返回 [-1, -1]
        if (first == -1) return {-1, -1};

        // 查找最后一个等于 target 的位置
        int last = findLast(nums, target);

        return {first, last};
    }

private:
    int findFirst(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size() - 1;
        int result = -1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] >= target) {
                if (nums[mid] == target) {
                    result = mid;
                }
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        return result;
    }

    int findLast(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size() - 1;
        int result = -1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
```

```
            if (nums[mid] <= target) {
                if (nums[mid] == target) {
                    result = mid;
                }
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return result;
    }
};
```

分析：- 时间复杂度：$O(\log n)$ - 空间复杂度：$O(1)$ - 核心思想：分别使用两次二分查找，找到目标值的第一个和最后一个位置

6.7.3 例题 3：搜索旋转排序数组（LeetCode 33）

问题描述：给你一个整数数组 $nums$，它原来是一个升序排序的数组，但在预先未知的某个点上进行了旋转。找出并返回数组中的 $target$，如果未找到返回 -1。

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size() - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid;
            }

            // 判断 mid 是在左侧递增区间还是右侧递增区间
            if (nums[left] <= nums[mid]) {
                // mid 在左侧递增区间
                if (nums[left] <= target && target < nums[mid]) {
                    // target 在左侧递增区间内
                    right = mid - 1;
                } else {
                    // target 在右侧区间
                    left = mid + 1;
                }
            } else {
                // mid 在右侧递增区间
                if (nums[mid] < target && target <= nums[right]) {
                    // target 在右侧递增区间内
                    left = mid + 1;
                } else {
                    // target 在左侧区间
                    right = mid - 1;
                }
            }
        }

        return -1;
```

```
    }
};
```

分析：- 时间复杂度：$O(\log n)$ - 空间复杂度：$O(1)$ - 核心思想：分析中间点在哪个递增区间，然后判断目标值在哪个区间，缩小搜索范围

6.8 二分答案

除了在数组中查找元素外，二分查找还经常用于"二分答案"，即当问题的答案具有单调性时，可以用二分查找确定最优解。

6.8.1 例题 4：分割数组的最大值（LeetCode 410）

问题描述：给定一个非负整数数组 $nums$ 和一个整数 m，你需要将这个数组分成 m 个非空的连续子数组。设计一个算法使得这 m 个子数组各自和的最大值最小。

```
class Solution {
public:
    int splitArray(vector<int>& nums, int m) {
        int left = *max_element(nums.begin(), nums.end()); // 最小可能的最大值
        int right = accumulate(nums.begin(), nums.end(), 0); // 最大可能的最大值

        while (left < right) {
            int mid = left + (right - left) / 2;

            // 检查是否可以将数组分成 m 个子数组，使得每个子数组的和不超过 mid
            if (canSplit(nums, m, mid)) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        return left;
    }

private:
    // 判断是否可以将数组分成 m 个子数组，使得每个子数组的和不超过 maxSum
    bool canSplit(vector<int>& nums, int m, int maxSum) {
        int count = 1; // 子数组数量
        int currentSum = 0; // 当前子数组的和

        for (int num : nums) {
            if (currentSum + num > maxSum) {
                count++;
                currentSum = num;

                if (count > m) {
                    return false;
                }
            } else {
                currentSum += num;
            }
        }

        return true;
    }
};
```

分析：- 时间复杂度：$O(n * \log(sum))$，其中 n 是数组长度，sum 是数组元素和 - 空间复杂度：$O(1)$ - 核心思想：二分答案，猜测最大子数组和，然后验证是否可行

6.9 易错点与调试技巧

1. 【区间定义】不同的区间定义（开/闭）需要不同的循环条件和更新方式

```cpp
// 闭区间 [left, right]
int left = 0, right = nums.size() - 1;
while (left <= right) {
    // ...
    if (condition) right = mid - 1;
    else left = mid + 1;
}

// 开区间 [left, right)
int left = 0, right = nums.size();
while (left < right) {
    // ...
    if (condition) right = mid;
    else left = mid + 1;
}
```

2. 【中点计算】防止整数溢出

```cpp
// 错误写法，可能导致整数溢出
int mid = (left + right) / 2;

// 正确写法
int mid = left + (right - left) / 2;
```

7 BFS

7.1 算法概述

【广度优先搜索】是一种图搜索算法，用于遍历或搜索图中的节点。它从起始节点开始，逐层向外扩展，直到找到目标节点或遍历完所有节点。BFS 通常用于寻找最短路径、连通性等问题。

7.2 算法设计思路

广度优先搜索的核心思想是：使用队列来实现层次遍历，确保首先访问距离起点最近的节点，然后逐层向外扩展。其基本步骤如下：1. 将起始节点加入队列，并标记为已访问 2. 当队列不为空时，重复以下步骤：- 从队列中取出一个节点 - 访问该节点的所有未被访问的邻居节点，并将它们加入队列 3. 直到找到目标节点或队列为空

7.3 代码实现与解析

7.3.1 基本 BFS 实现

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>

using namespace std;

// 广度优先搜索函数
void bfs(const vector<vector<int>>& graph, int start) {
    queue<int> q; // 队列用于存储待访问的节点
    unordered_set<int> visited; // 集合用于记录已访问的节点

    q.push(start); // 将起始节点加入队列
```

```
    visited.insert(start); // 标记起始节点为已访问

    while (!q.empty()) {
        int node = q.front(); // 取出队首节点
        q.pop();

        cout << " 访问节点: " << node << endl;

        // 访问所有邻居节点
        for (int neighbor : graph[node]) {
            if (visited.find(neighbor) == visited.end()) { // 如果邻居节点未被访问
                q.push(neighbor); // 加入队列
                visited.insert(neighbor); // 标记为已访问
            }
        }
    }
}

int main() {
    // 示例图的邻接表表示
    vector<vector<int>> graph = {
        {1, 2},    // 节点 0 的邻居
        {0, 3, 4}, // 节点 1 的邻居
        {0, 4},    // 节点 2 的邻居
        {1, 5},    // 节点 3 的邻居
        {1, 2, 5}, // 节点 4 的邻居
        {3, 4}     // 节点 5 的邻居
    };

    bfs(graph, 0); // 从节点 0 开始进行广度优先搜索

    return 0;
}
```

7.3.2 时间复杂度分析

- 时间复杂度：$O(V + E)$，其中 V 是节点数，E 是边数
- 空间复杂度：$O(V)$，需要存储队列和访问标记

7.4 优化策略

1. 【双向 BFS】当同时知道起点和终点时，可以从两端同时开始 BFS，直到两个搜索相遇

```
bool bidirectionalBFS(int start, int end) {
    if (start == end) return true;

    queue<int> q1, q2;
    unordered_set<int> visited1, visited2;

    q1.push(start);
    q2.push(end);
    visited1.insert(start);
    visited2.insert(end);

    while (!q1.empty() && !q2.empty()) {
        // 总是从较小的队列扩展
        if (q1.size() > q2.size()) {
```

```
            swap(q1, q2);
            swap(visited1, visited2);
        }

        int size = q1.size();
        for (int i = 0; i < size; i++) {
            int node = q1.front();
            q1.pop();

            for (int neighbor : graph[node]) {
                if (visited1.count(neighbor)) continue; // 已访问

                if (visited2.count(neighbor)) {
                    return true; // 两个搜索相遇
                }

                visited1.insert(neighbor);
                q1.push(neighbor);
            }
        }
    }

    return false; // 无法到达
}
```

2. 【01BFS】处理边权为 0 或 1 的最短路径问题时，可以用双端队列优化

```
vector<int> bfs01(vector<vector<pair<int, int>>>& graph, int start) {
    int n = graph.size();
    vector<int> dist(n, INT_MAX);
    deque<int> dq; // 使用双端队列

    dist[start] = 0;
    dq.push_front(start);

    while (!dq.empty()) {
        int node = dq.front();
        dq.pop_front();

        for (auto& [neighbor, weight] : graph[node]) {
            if (dist[neighbor] > dist[node] + weight) {
                dist[neighbor] = dist[node] + weight;

                if (weight == 0) {
                    dq.push_front(neighbor); // 权重为 0 的边，加入队首
                } else {
                    dq.push_back(neighbor);  // 权重为 1 的边，加入队尾
                }
            }
        }
    }

    return dist;
}
```

3. 【状态压缩 BFS】当需要记录多个状态时，可以使用位运算进行状态压缩

```cpp
// 使用位运算表示状态，比如在迷宫中记录已访问的物品
int bfsWithBitmask(vector<vector<int>>& grid, int startX, int startY) {
    int rows = grid.size();
    int cols = grid[0].size();
    int totalItems = 5; // 假设有 5 个物品需要收集

    // visited[x][y][state] 表示在 (x,y) 位置，已收集物品状态为 state 时是否访问过
    vector<vector<vector<bool>>> visited(rows, vector<vector<bool>>(cols, vector<bool>(1 << totalItems, false)));

    queue<tuple<int, int, int>> q; // (x, y, state)
    q.push({startX, startY, 0});
    visited[startX][startY][0] = true;

    int steps = 0;
    vector<pair<int, int>> dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    while (!q.empty()) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            auto [x, y, state] = q.front();
            q.pop();

            // 如果所有物品都收集了
            if (state == (1 << totalItems) - 1) {
                return steps;
            }

            for (auto& dir : dirs) {
                int nx = x + dir.first;
                int ny = y + dir.second;

                if (nx < 0 || nx >= rows || ny < 0 || ny >= cols || grid[nx][ny] == -1) continue;

                int newState = state;
                if (grid[nx][ny] > 0) {
                    // 收集物品，物品编号为 grid[nx][ny]-1
                    newState |= (1 << (grid[nx][ny] - 1));
                }

                if (!visited[nx][ny][newState]) {
                    visited[nx][ny][newState] = true;
                    q.push({nx, ny, newState});
                }
            }
        }
        steps++;
    }

    return -1; // 无法收集所有物品
}
```

7.5 A* 搜索算法（BFS 的扩展）

A* 算法是 BFS 的一种启发式扩展，它使用估价函数来引导搜索方向，优先探索更有希望到达目标的路径。

```cpp
// A* 算法寻找最短路径
vector<pair<int, int>> aStarSearch(vector<vector<int>>& grid, pair<int, int> start, pair<int, int> goal) {
    int rows = grid.size();
    int cols = grid[0].size();

    // 优先队列，按 f 值（g+h）排序，最小值在队首
    priority_queue<tuple<int, int, int>, vector<tuple<int, int, int>>, greater<>> pq;

    // g[r][c] 表示从起点到 (r,c) 的已知最短距离
    vector<vector<int>> g(rows, vector<int>(cols, INT_MAX));

    // parent 记录路径
    vector<vector<pair<int, int>>> parent(rows, vector<pair<int, int>>(cols));

    // 启发函数 h：曼哈顿距离
    auto h = [&goal](int r, int c) -> int {
        return abs(r - goal.first) + abs(c - goal.second);
    };

    g[start.first][start.second] = 0;
    pq.push({h(start.first, start.second), start.first, start.second}); // {f, r, c}

    vector<pair<int, int>> dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    while (!pq.empty()) {
        auto [f, r, c] = pq.top();
        pq.pop();

        // 达到目标
        if (r == goal.first && c == goal.second) {
            // 重建路径
            vector<pair<int, int>> path;
            for (auto pos = make_pair(r, c); pos != start; pos = parent[pos.first][pos.second]) {
                path.push_back(pos);
            }
            path.push_back(start);
            reverse(path.begin(), path.end());
            return path;
        }

        // f 值已经不是最小值，跳过
        if (f > g[r][c] + h(r, c)) continue;

        for (auto& dir : dirs) {
            int nr = r + dir.first;
            int nc = c + dir.second;

            if (nr < 0 || nr >= rows || nc < 0 || nc >= cols || grid[nr][nc] == 1) continue;

            int ng = g[r][c] + 1; // 假设步长为 1

            if (ng < g[nr][nc]) {
                g[nr][nc] = ng;
                parent[nr][nc] = {r, c};
                pq.push({ng + h(nr, nc), nr, nc});
            }
```

```
        }
    }

    return {}; // 无法到达目标
}
```

7.6 0. 通用 BFS 框架

场景：最短路（无权图）、层次遍历、状态搜索 **要点**：用队列、用 dist 或 visited 记录。

```
#include <bits/stdc++.h>
using namespace std;

int n;
vector<vector<int>> g;
vector<int> dist;

void bfs(int s){
    dist.assign(n, -1);
    queue<int> q;
    dist[s]=0; q.push(s);
    while(!q.empty()){
        int u=q.front(); q.pop();
        for(int v: g[u]){
            if(dist[v]==-1){
                dist[v]=dist[u]+1;
                q.push(v);
            }
        }
    }
}
```

7.7 1. 无权图最短路

场景：图/网格最短路径，迷宫走最少步数

```
int n,m; vector<string> maze;
int dx[4]={1,-1,0,0}, dy[4]={0,0,1,-1};

int bfs_grid(int sx,int sy,int tx,int ty){
    vector<vector<int>> dist(n, vector<int>(m,-1));
    queue<pair<int,int>> q;
    dist[sx][sy]=0; q.push({sx,sy});
    while(!q.empty()){
        auto [x,y]=q.front(); q.pop();
        if(x==tx && y==ty) return dist[x][y];
        for(int k=0;k<4;k++){
            int nx=x+dx[k], ny=y+dy[k];
            if(nx<0||nx>=n||ny<0||ny>=m) continue;
            if(maze[nx][ny]=='#'||dist[nx][ny]!=-1) continue;
            dist[nx][ny]=dist[x][y]+1;
            q.push({nx,ny});
        }
    }
    return -1; // 不可达
}
```

7.8 2. 多源 BFS

场景：火焰扩散、腐烂的橘子（LeetCode 994）、最近距离 **要点**：初始时把所有源点同时入队。

```
int n,m; vector<string> grid;
int dx[4]={1,-1,0,0}, dy[4]={0,0,1,-1};

int multi_bfs(vector<pair<int,int>> src){
    vector<vector<int>> dist(n, vector<int>(m,-1));
    queue<pair<int,int>> q;
    for(auto [x,y]: src){ dist[x][y]=0; q.push({x,y}); }
    while(!q.empty()){
        auto [x,y]=q.front(); q.pop();
        for(int k=0;k<4;k++){
            int nx=x+dx[k], ny=y+dy[k];
            if(nx<0||nx>=n||ny<0||ny>=m) continue;
            if(grid[nx][ny]=='#'||dist[nx][ny]!=-1) continue;
            dist[nx][ny]=dist[x][y]+1;
            q.push({nx,ny});
        }
    }
    // dist[i][j] 表示到最近源点的距离
    return 0;
}
```

7.9 3. 层次遍历（树/图分层）

场景：二叉树层序遍历、计算最短层数

```
struct TreeNode{ int val; TreeNode*left,*right; };
vector<vector<int>> levelOrder(TreeNode* root){
    vector<vector<int>> res;
    if(!root) return res;
    queue<TreeNode*> q; q.push(root);
    while(!q.empty()){
        int sz=q.size(); vector<int> level;
        while(sz--){
            auto* u=q.front(); q.pop();
            level.push_back(u->val);
            if(u->left) q.push(u->left);
            if(u->right) q.push(u->right);
        }
        res.push_back(level);
    }
    return res;
}
```

7.10 4. 双向 BFS

场景：起点-终点图搜索（如单词接龙 LeetCode 127）**要点**：正反两队列，谁小扩展谁。

```cpp
int bidir_bfs(int s,int t){
    if(s==t) return 0;
    vector<int> dist1(n,-1), dist2(n,-1);
    queue<int> q1,q2; dist1[s]=0; dist2[t]=0;
    q1.push(s); q2.push(t);
    while(!q1.empty() && !q2.empty()){
        if(q1.size()<=q2.size()){
            int u=q1.front(); q1.pop();
            for(int v:g[u]) if(dist1[v]==-1){
                dist1[v]=dist1[u]+1; q1.push(v);
                if(dist2[v]!=-1) return dist1[v]+dist2[v];
            }
        }else{
            int u=q2.front(); q2.pop();
            for(int v:g[u]) if(dist2[v]==-1){
                dist2[v]=dist2[u]+1; q2.push(v);
                if(dist1[v]!=-1) return dist1[v]+dist2[v];
            }
        }
    }
    return -1;
}
```

7.11 5. 0-1 BFS（边权 0/1）

场景：最短路，但边权只有 0 或 1 **要点**：用 deque，0 权边从队首进，1 权边从队尾进。

```cpp
int n; vector<vector<pair<int,int>>> g; // {v, w=0/1}
vector<int> dist;
int bfs01(int s){
    dist.assign(n, 1e9); deque<int> dq;
    dist[s]=0; dq.push_back(s);
    while(!dq.empty()){
        int u=dq.front(); dq.pop_front();
        for(auto [v,w]: g[u]){
            if(dist[v]>dist[u]+w){
                dist[v]=dist[u]+w;
                if(w==0) dq.push_front(v);
                else dq.push_back(v);
            }
        }
    }
    return 0;
}
```

7.12 6. 状态 BFS（最小操作步数）

场景：数字变换、棋盘/数码难题、最短编辑操作 **要点**：状态 + 判重，用 unordered_set 或 map。

```cpp
string start="12345678x", goal="12345678x";
vector<vector<int>> moves={{1,3},{0,2,4},{1,5},{0,4,6},{1,3,5,7},{2,4,8},{3,7},{4,6,8},{5,7}};
unordered_map<string,int> dist;
```

```cpp
int bfs_state(){
    queue<string> q; q.push(start); dist[start]=0;
    while(!q.empty()){
        auto cur=q.front(); q.pop();
        if(cur==goal) return dist[cur];
        int idx=cur.find('x');
        for(int nxt:moves[idx]){
            string t=cur;
            swap(t[idx],t[nxt]);
            if(!dist.count(t)){
                dist[t]=dist[cur]+1;
                q.push(t);
            }
        }
    }
    return -1;
}
```

7.13 7. 多层状态 BFS（带钥匙/层次）

场景：迷宫里带钥匙，状态 = 位置 + 钥匙集合

```cpp
int n,m; vector<string> maze;
struct Node{int x,y,mask;};
int bfs_keys(int sx,int sy){
    vector<vector<vector<int>>> dist(n, vector<vector<int>>(m, vector<int>(1<<6,-1)));
    queue<Node> q; dist[sx][sy][0]=0;
    q.push({sx,sy,0});
    int dx[4]={1,-1,0,0}, dy[4]={0,0,1,-1};
    while(!q.empty()){
        auto [x,y,mask]=q.front(); q.pop();
        for(int k=0;k<4;k++){
            int nx=x+dx[k], ny=y+dy[k], nmask=mask;
            if(nx<0||nx>=n||ny<0||ny>=m) continue;
            char c=maze[nx][ny];
            if(c=='#') continue;
            if(c>='a'&&c<='f') nmask|=(1<<(c-'a'));      // 拿钥匙
            if(c>='A'&&c<='F' && !(nmask>>(c-'A')&1)) continue; // 没钥匙打不开
            if(dist[nx][ny][nmask]==-1){
                dist[nx][ny][nmask]=dist[x][y][mask]+1;
                q.push({nx,ny,nmask});
            }
        }
    }
    return -1;
}
```

7.14 8. 拓扑排序 (Kahn's BFS)

场景：有向无环图的排序、课程表

```cpp
int n; vector<vector<int>> g; vector<int> indeg;
vector<int> topo_sort(){
```

```
    queue<int> q; for(int i=0;i<n;i++) if(indeg[i]==0) q.push(i);
    vector<int> order;
    while(!q.empty()){
        int u=q.front(); q.pop(); order.push_back(u);
        for(int v:g[u]) if(--indeg[v]==0) q.push(v);
    }
    return order;
}
```

7.15 9. 迷宫问题 (BFS/DFS)

```
// BFS 解决迷宫问题
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// 四个方向：上、右、下、左
const int dx[4] = {-1, 0, 1, 0};
const int dy[4] = {0, 1, 0, -1};

bool bfs(vector<vector<char>>& maze, pair<int, int> start, pair<int, int> end) {
    int n = maze.size();
    int m = maze[0].size();
    vector<vector<bool>> visited(n, vector<bool>(m, false));
    queue<pair<int, int>> q;

    q.push(start);
    visited[start.first][start.second] = true;

    while (!q.empty()) {
        pair<int, int> curr = q.front();
        q.pop();

        // 到达终点
        if (curr.first == end.first && curr.second == end.second) {
            return true;
        }

        // 尝试四个方向
        for (int i = 0; i < 4; i++) {
            int nx = curr.first + dx[i];
            int ny = curr.second + dy[i];

            // 检查新位置是否有效且未访问
            if (nx >= 0 && nx < n && ny >= 0 && ny < m &&
                maze[nx][ny] != '#' && !visited[nx][ny]) {
                q.push({nx, ny});
                visited[nx][ny] = true;
            }
        }
    }

    return false; // 无法到达终点
}
```

8 DFS

8.1 算法概述

【深度优先搜索】(Depth-First Search, DFS) 是一种用于遍历或搜索树/图数据结构的算法。其特点是沿着一条路径尽可能深入地搜索，直到不能再深入为止，然后回溯到前一个节点，继续搜索其他路径。

8.2 算法设计思路

DFS 的核心思想是：1. 从起点开始，选择一个方向深入探索 2. 不断深入，直到无法继续（达到目标或碰到边界）3. 回溯到上一个状态，选择另一个方向继续探索 4. 重复上述过程直到所有可能的路径都被探索完毕

DFS 通常使用函数递归或显式栈来实现回溯机制。

8.3 代码实现与解析

8.3.1 递归实现（以图的遍历为例）

```
// 邻接表表示的图
vector<vector<int>> graph;
// 访问标记数组
vector<bool> visited;

// DFS 函数定义
void dfs(int node) {
    // 标记当前节点为已访问
    visited[node] = true;
    cout << " 访问节点: " << node << endl;

    // 遍历所有相邻节点
    for (int neighbor : graph[node]) {
        // 如果相邻节点未被访问，则递归访问它
        if (!visited[neighbor]) {
            dfs(neighbor);
        }
    }
}

// 图的 DFS 遍历，处理非连通图的情况
void dfsTraversal(int n) {
    visited.resize(n, false);
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i);
        }
    }
}
```

8.3.2 非递归实现（使用栈）

```
void dfsIterative(int start) {
    stack<int> s;
    s.push(start);
    visited[start] = true;

    while (!s.empty()) {
        int node = s.top();
        s.pop();
```

```
        cout << " 访问节点: " << node << endl;

        // 注意：与递归 DFS 相比，迭代版本的访问顺序可能不同
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                s.push(neighbor);
            }
        }
    }
}
```

8.4 流程图解析

DFS过程示意（以二叉树为例）：

```
      1          第1步：访问节点1
     / \
    2   3        第2步：访问节点2
   / \           第3步：访问节点4
  4   5          第4步：访问节点5（节点4没有子节点）
                 第5步：返回节点2
                 第6步：访问节点3（节点2的所有路径都已探索）
```

8.5 复杂度分析

- 时间复杂度：$O(V + E)$，其中 V 是节点数，E 是边数
  - 在邻接表表示下，每个节点和每条边都会被访问一次
  - 在邻接矩阵表示下，时间复杂度为 $O(V^2)$
- 空间复杂度：$O(V)$，递归实现时，递归栈的最大深度为图的深度，最坏情况为 $O(V)$

8.6 典型应用场景

DFS 在算法竞赛中有广泛应用，常见的应用场景包括：

1. 【图的遍历】与节点的发现/访问
2. 【连通分量】的识别
3. 【拓扑排序】（通过结束时间的逆序）
4. 【路径查找】问题，特别是所有路径的枚举
5. 【迷宫问题】和网格搜索
6. 【回溯算法】的框架，如排列组合、数独解决等
7. 【图的环检测】

8.7 典型例题分析

8.7.1 例题 1：岛屿的数量（LeetCode 200）

问题描述：给你一个由 '1'（陆地）和 '0'（水）组成的二维网格，请你计算网格中岛屿的数量。岛屿由相邻的陆地连接而成，上、下、左、右四个方向视为相邻。

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        if (grid.empty() || grid[0].empty()) return 0;

        int rows = grid.size();
        int cols = grid[0].size();
        int islands = 0;
```

```
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (grid[i][j] == '1') {
                    islands++; // 发现一个新岛屿
                    dfs(grid, i, j); // 用 DFS 将整个岛屿标记为已访问
                }
            }
        }

        return islands;
    }

private:
    void dfs(vector<vector<char>>& grid, int row, int col) {
        int rows = grid.size();
        int cols = grid[0].size();

        // 边界检查和已访问检查
        if (row < 0 || col < 0 || row >= rows || col >= cols || grid[row][col] != '1')
            return;

        grid[row][col] = '2'; // 标记为已访问（可以设为任何非'1'值）

        // 探索上下左右四个方向
        dfs(grid, row + 1, col);
        dfs(grid, row - 1, col);
        dfs(grid, row, col + 1);
        dfs(grid, row, col - 1);
    }
};
```

分析：- 时间复杂度：$O(M \times N)$，其中 M 是行数，N 是列数 - 空间复杂度：$O(M \times N)$，最坏情况下，整个网格都是陆地，递归调用深度为 $M \times N$ - 核心思想：当发现一个陆地格子时，使用 DFS 将整个相连的陆地区域标记为已访问，避免重复计数

8.7.2 例题 2：全排列（LeetCode 46）

问题描述：给定一个不含重复数字的数组，返回其所有可能的全排列。

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> result;
        vector<int> current;
        vector<bool> used(nums.size(), false);

        dfs(nums, used, current, result);
        return result;
    }

private:
    void dfs(vector<int>& nums, vector<bool>& used, vector<int>& current, vector<vector<int>>& result) {
        // 达到排列长度，记录结果
        if (current.size() == nums.size()) {
            result.push_back(current);
            return;
        }
```

```
        // 尝试每个数字
        for (int i = 0; i < nums.size(); i++) {
            // 跳过已使用的数字
            if (used[i]) continue;

            // 选择当前数字
            used[i] = true;
            current.push_back(nums[i]);

            // 递归生成后续排列
            dfs(nums, used, current, result);

            // 回溯：撤销选择
            current.pop_back();
            used[i] = false;
        }
    }
};
```

分析：- 时间复杂度：$O(n!)$，其中 n 是数组长度，全排列的数量为 $n!$ - 空间复杂度：$O(n)$，递归调用栈的深度为 n，额外使用了 used 数组和 current 数组 - 核心思想：使用 DFS 和回溯技术生成所有可能的排列

8.8 代码模板

8.8.1 1. 通用模板

```
// 邻接表：vector<vector<int>> g(n);
int n; vector<vector<int>> g;
vector<int> vis;              // 0= 未访问,1= 访问中,2= 访问完
vector<int> parent;

void dfs_graph(int u, int p=-1){
    vis[u] = 1; parent[u] = p;
    for(int v: g[u]){
        if(v == p) continue;          // 无向图避免走回父边
        if(!vis[v]) dfs_graph(v, u); // 树/连通性/遍历
        // 判有向环：if(vis[v]==1) 有回边
    }
    vis[u] = 2;
}

// 迭代版（避免深递归栈）
void dfs_iter(int s){
    vector<int> st; st.push_back(s);
    parent[s]=-1; vis[s]=1;
    while(!st.empty()){
        int u = st.back(); st.pop_back();
        for(int v: g[u]){
            if(v==parent[u]) continue;
            if(!vis[v]){ parent[v]=u; vis[v]=1; st.push_back(v); }
        }
    }
}
```

8.8.2 2. 连通模板（岛屿问题）

```
int n,m; vector<string> grid;
int dx[4]={1,-1,0,0}, dy[4]={0,0,1,-1};

void dfs_cell(int x,int y){
    if(x<0||x>=n||y<0||y>=m||grid[x][y]!='1') return;
    grid[x][y]='0'; // 染色为已访问
    for(int k=0;k<4;k++) dfs_cell(x+dx[k], y+dy[k]);
}

int count_islands(){
    int cnt=0;
    for(int i=0;i<n;i++)for(int j=0;j<m;j++)
        if(grid[i][j]=='1'){ cnt++; dfs_cell(i,j); }
    return cnt;
}
```

8.8.3 3. 搜索图上所有路径

```
int n; vector<vector<int>> g;
vector<int> path, used;
vector<vector<int>> allPaths;

void dfs_paths(int u, int t){
    if(u==t){ allPaths.push_back(path); return; }
    for(int v: g[u]){
        if(used[v]) continue;
        used[v]=1; path.push_back(v);
        dfs_paths(v,t);
        path.pop_back(); used[v]=0;
    }
}
// 使用: used.assign(n,0); used[s]=1; path={s}; dfs_paths(s,t);
```

8.8.4 4. 回溯剪枝

```
// 全排列
vector<int> a, perm, used;
vector<vector<int>> ans_perm;
void dfs_perm(){
    if(perm.size()==a.size()){ ans_perm.push_back(perm); return; }
    for(int i=0;i<(int)a.size();i++){
        if(used[i]) continue;
        used[i]=1; perm.push_back(a[i]);
        dfs_perm();
        perm.pop_back(); used[i]=0;
    }
}

// 组合（从候选 a 中选 k 个）
int k; vector<int> comb; vector<vector<int>> ans_comb;
void dfs_comb(int start){
    if((int)comb.size()==k){ ans_comb.push_back(comb); return; }
    for(int i=start;i<(int)a.size();i++){
```

```
        comb.push_back(a[i]);
        dfs_comb(i+1);
        comb.pop_back();
    }
}

// 子集（幂集）
vector<int> subset; vector<vector<int>> ans_subset;
void dfs_subset(int i){
    if(i==(int)a.size()){ ans_subset.push_back(subset); return; }
    dfs_subset(i+1);                // 不选
    subset.push_back(a[i]);         // 选
    dfs_subset(i+1);
    subset.pop_back();
}
```

8.8.5 5. N 皇后剪枝

```
int n; vector<int> col, d1, d2; // d1: r-c 偏移 n-1; d2: r+c
vector<int> place; vector<vector<string>> sols;

void dfs_queens(int r){
    if(r==n){
        vector<string> board(n, string(n,'.'));
        for(int i=0;i<n;i++) board[i][place[i]]='Q';
        sols.push_back(board); return;
    }
    for(int c=0;c<n;c++){
        if(col[c]||d1[r-c+n-1]||d2[r+c]) continue;
        col[c]=d1[r-c+n-1]=d2[r+c]=1; place[r]=c;
        dfs_queens(r+1);
        col[c]=d1[r-c+n-1]=d2[r+c]=0;
    }
}
```

8.8.6 6. 树题：深度/路径和/根到叶路径

```
// 树深度
int n; vector<vector<int>> tree;
int dfs_depth(int u,int p){
    int h=0;
    for(int v: tree[u]) if(v!=p) h=max(h, 1+dfs_depth(v,u));
    return h;
}

// 路径和（根到叶 =targetSum 条数）
int target, ways=0; vector<int> val;
void dfs_sum(int u,int p,long long s){
    s += val[u];
    if((int)tree[u].size()==(p==-1?0:1)){ // 叶子
        if(s==target) ways++;
    }
    for(int v: tree[u]) if(v!=p) dfs_sum(v,u,s);
}
```

8.8.7 7. 最近公共祖先（LCA，DFS 版）

小数据：记录根到节点路径，或父数组回溯。**更稳**（建议准备）：倍增 + DFS 进出时序。

```
// 倍增模板
const int LOG=20;
int n, up[200005][LOG], depth_[200005];
vector<vector<int>> tr;

void dfs_lca(int u,int p){
    up[u][0]=p==-1?u:p;
    for(int j=1;j<LOG;j++) up[u][j]=up[up[u][j-1]][j-1];
    for(int v: tr[u]) if(v!=p){
        depth_[v]=depth_[u]+1;
        dfs_lca(v,u);
    }
}

int lca(int a,int b){
    if(depth_[a]<depth_[b]) swap(a,b);
    int diff=depth_[a]-depth_[b];
    for(int j=0;j<LOG;j++) if(diff>>j & 1) a=up[a][j];
    if(a==b) return a;
    for(int j=LOG-1;j>=0;j--)
        if(up[a][j]!=up[b][j]){ a=up[a][j]; b=up[b][j]; }
    return up[a][0];
}
```

8.8.8 8. Tarjan：割点与桥（无向图）

要点：dfn/low；根特判子树数；low[v]>dfn[u] 为桥。

```
int n, timer_=0;
vector<vector<int>> g;
vector<int> dfn, low, isCut;
vector<pair<int,int>> bridges;

void dfs_tarjan(int u,int p){
    dfn[u]=low[u]=++timer_;
    int child=0;
    for(int v: g[u]){
        if(!dfn[v]){
            child++; dfs_tarjan(v,u);
            low[u]=min(low[u], low[v]);
            if(p!=-1 && low[v]>=dfn[u]) isCut[u]=1;   // 割点
            if(low[v]>dfn[u]) bridges.push_back({u,v}); // 桥
        }else if(v!=p) low[u]=min(low[u], dfn[v]);
    }
    if(p==-1 && child>1) isCut[u]=1; // 根的特判
}
```

8.8.9 9. Tarjan：强连通分量（SCC，有向图）

要点：栈内标记；low[u]==dfn[u] 时弹栈成分。

```
int n, timer_=0, scc_cnt=0;
vector<vector<int>> g;
vector<int> dfn, low, inSt, comp;
```

```
vector<int> st;

void dfs_scc(int u){
    dfn[u]=low[u]=++timer_; st.push_back(u); inSt[u]=1;
    for(int v: g[u]){
        if(!dfn[v]){ dfs_scc(v); low[u]=min(low[u], low[v]); }
        else if(inSt[v]) low[u]=min(low[u], dfn[v]);
    }
    if(low[u]==dfn[u]){
        while(true){
            int x=st.back(); st.pop_back(); inSt[x]=0; comp[x]=scc_cnt;
            if(x==u) break;
        }
        scc_cnt++;
    }
}
```

8.8.10 10. 拓扑排序（DFS 版，DAG）

要点：后序入栈；若需判环，另做有向环检测。

```
int n; vector<vector<int>> g;
vector<int> vis, order; // 0 未,1 中,2 完

bool dfs_topo(int u){
    vis[u]=1;
    for(int v: g[u]){
        if(vis[v]==1) return false;      // 有向环
        if(!vis[v]) if(!dfs_topo(v)) return false;
    }
    vis[u]=2; order.push_back(u);        // 后序
    return true;
}
// 使用：对所有未访问 u 调 dfs_topo(u)，最后 reverse(order)
```

8.8.11 11. DAG 最长路径（DFS + 记忆化）

```
int n; vector<vector<int>> g; vector<int> dp; // -1 未算
int dfs_longest(int u){
    if(dp[u]!=-1) return dp[u];
    int best=0;
    for(int v: g[u]) best=max(best, 1+dfs_longest(v));
    return dp[u]=best;
}
```

8.8.12 12. 状态空间 DFS + 剪枝（通用骨架）

场景：选择/排列/搜索最优解 **要点**：界限函数 bound；先排序以利剪枝。

```
long long best = LLONG_MIN;
void dfs_state(int i, long long cur){
    // 估价剪枝
    // if(cur + optimistic_upper_bound(i) <= best) return;

    if(i==N){ best=max(best, cur); return; }
    // 分支 1：选
```

```
    // dfs_state(i+1, cur + gain[i]);
    // 分支 2: 不选
    // dfs_state(i+1, cur);
}
```

8.8.13 13. 位压缩 DFS（如旅行/匹配小规模）

```
int n; vector<vector<int>> w; // 代价/连通
int dp[1<<20][20]; // 记忆化，或 unordered_map<int,int> 更省内存
int INF=1e9;

int tsp(int mask,int u){
    if(mask==(1<<n)-1) return 0;
    int &res=dp[mask][u];
    if(res!=-1) return res;
    res=INF;
    for(int v=0; v<n; v++){
        if(!(mask>>v&1) && w[u][v]<INF)
            res=min(res, w[u][v] + tsp(mask|1<<v, v));
    }
    return res;
}
```

8.8.14 14. DFS 序/进出时刻（树上区间技巧）

要点：Euler Tour，用于子树转区间、判断祖先关系。

```
int n, timer_=0;
vector<vector<int>> tr;
vector<int> tin, tout;

void dfs_euler(int u,int p){
    tin[u]=++timer_;
    for(int v: tr[u]) if(v!=p) dfs_euler(v,u);
    tout[u]=timer_;
}
bool is_ancestor(int a,int b){ // a 是否 b 的祖先
    return tin[a]<=tin[b] && tout[b]<=tout[a];
}
```

8.8.15 15. 矩阵优化（U560764）

- 类似 DFS+ 贪心

```
class matrixOptimizer{
private:
    int row, col;
    vector<vector<int>> A;
    vector<vector<int>> S;
public:
    matrixOptimizer(const vector<vector<int>>& matrix): A(matrix), row(matrix.size()), col(matrix[0].size()){S=A
    std::pair<vector<int>,vector<int>> CalViolate(){
        vector<int> row_v(row,0);
        vector<int> col_v(col,0);
        for(int i=0; i<row; i++){
            for(int j=0; j<col; j++){
```

```
                row_v[i] ^= S[i][j];
                col_v[j] ^= S[i][j];
            }
        }
        return{row_v, col_v};
    }
    int CalBenefit(int r, int c){
        return (S[r][c] == A[r][c])?(-1):1;
    }
    void Fix(){
        //计算需要修复的行和列
        auto [row_v, col_v] = CalViolate();
        vector<int> rows_to_fix, cols_to_fix;
        for(int i=0; i<row; i++){
            if(row_v[i] != 0){
                rows_to_fix.push_back(i);
            }
        }
        for(int j=0; j<col; j++){
            if(col_v[j] != 0){
                cols_to_fix.push_back(j);
            }
        }
        //先修行列都违反的
        while((!rows_to_fix.empty())&&(!cols_to_fix.empty())){
            int bestr = -1;
            int bestc = -1;
            int benefit = INT_MIN;
            for(int r : rows_to_fix){
                for(int c: cols_to_fix){
                    int calbenefit = CalBenefit(r,c);
                    if(calbenefit > benefit){
                        bestr = r;
                        bestc = c;
                        benefit = calbenefit;
                    }
                }
            }
            if(bestr >=0 && bestc >= 0){
                S[bestr][bestc] = 1- S[bestr][bestc];
                rows_to_fix.erase(std::find(rows_to_fix.begin(), rows_to_fix.end(), bestr));
                cols_to_fix.erase(std::find(cols_to_fix.begin(), cols_to_fix.end(), bestc));
            }
            else{
                break;
            }
        }
        //对单列进行优化
        while(rows_to_fix.size() >= 2){
            int r1 = rows_to_fix[0];
            int r2 = rows_to_fix[1];
            int bestc = -1;
            int benefit = INT_MIN;
            for(int c=0; c<col; c++){
                int colbenefit = CalBenefit(r1, c)+CalBenefit(r2,c);
                if(benefit < colbenefit){
                    benefit = colbenefit;
```

```
                bestc = c;
            }
        }
        S[r1][bestc] = 1-S[r1][bestc];
        S[r2][bestc] = 1-S[r2][bestc];
        rows_to_fix.erase(rows_to_fix.begin(), rows_to_fix.begin()+2);
    }
    //对单行进行优化
    while(cols_to_fix.size() >= 2){
        int c1 = cols_to_fix[0];
        int c2 = cols_to_fix[1];
        int bestr = -1;
        int benefit = INT_MIN;
        for(int r=0; r<row; r++){
            int rowbenefit = CalBenefit(r, c1)+CalBenefit(r, c2);
            if(benefit < rowbenefit){
                benefit = rowbenefit;
                bestr = r;
            }
        }
        S[bestr][c1] = 1-S[bestr][c1];
        S[bestr][c2] = 1-S[bestr][c2];
        cols_to_fix.erase(cols_to_fix.begin(), cols_to_fix.begin()+2);
    }
}
vector<vector<int>> Optimize(){
    int iter_n = std::max(col, row);
    for(int iter=0; iter<iter_n; iter++){
        Fix();
        bool improved = false;
         for (int i = 0; i < row && !improved; i++) {
            for (int j = 0; j < col && !improved; j++) {
                // 尝试翻转这个位置
                S[i][j] = 1 - S[i][j];

                // 检查是否仍然满足约束且汉明距离更小
                auto [rowViol, colViol] = CalViolate();
                bool valid = true;
                for (int v : rowViol) if (v != 0) valid = false;
                for (int v : colViol) if (v != 0) valid = false;

                if (!valid) {
                    S[i][j] = 1 - S[i][j];  // 撤销翻转
                } else {
                    improved = true;
                }
            }
        }

        if (!improved) break;
    }
    return S;
}
int getHammingDistance() {
    int distance = 0;
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
```

```
                if (S[i][j] != A[i][j]) distance++;
            }
        }
        return distance;
    }
};
```

8.9 易错点与调试技巧

1. 【访问标记】忘记标记节点为已访问，导致无限递归

```
// 错误写法
void dfs_wrong(int node) {
    // 忘记标记为已访问
    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            dfs_wrong(neighbor);
        }
    }
}

// 正确写法
void dfs_correct(int node) {
    visited[node] = true; // 立即标记为已访问
    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            dfs_correct(neighbor);
        }
    }
}
```

2. 【边界条件】检查不充分，特别是在网格问题中

```
// 错误写法
void dfs_grid_wrong(vector<vector<int>>& grid, int row, int col) {
    // 缺少边界检查
    if (grid[row][col] != 1) return; // 可能导致数组越界

    grid[row][col] = 2;
    dfs_grid_wrong(grid, row+1, col);
    // ... 其他方向
}

// 正确写法
void dfs_grid_correct(vector<vector<int>>& grid, int row, int col) {
    int rows = grid.size();
    int cols = grid[0].size();

    // 完整的边界检查
    if (row < 0 || row >= rows || col < 0 || col >= cols || grid[row][col] != 1)
        return;

    grid[row][col] = 2;
    dfs_grid_correct(grid, row+1, col);
    // ... 其他方向
}
```

3. 【递归深度】过深导致栈溢出

```
// 处理递归深度过大的情况
void dfs_with_depth_check(int node, int depth) {
    if (depth > MAX_DEPTH) return; // 限制递归深度

    visited[node] = true;
    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            dfs_with_depth_check(neighbor, depth + 1);
        }
    }
}
```

8.10 优化策略

1. 【记忆化搜索】避免重复计算

```
// 使用记忆化数组避免重复状态计算
vector<int> memo;

int dfs_with_memo(int state) {
    if (memo[state] != -1) return memo[state]; // 已计算过的状态直接返回

    // 计算结果
    int result = 0;
    // ... 计算逻辑

    memo[state] = result; // 存储结果
    return result;
}
```

2. 【剪枝】提前终止无效搜索路径

```
void dfs_with_pruning(int node, int target, int current_sum) {
    if (current_sum > target) {
        return; // 剪枝：当前和已超过目标，无需继续
    }

    if (current_sum == target) {
        // 找到解
        return;
    }

    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            visited[neighbor] = true;
            dfs_with_pruning(neighbor, target, current_sum + value[neighbor]);
            visited[neighbor] = false;
        }
    }
}
```

9 回溯法

9.1 算法概述

【回溯法】是一种通过穷举所有可能性来找到问题解的算法。它采用试错的思想，尝试分步解决问题，当发现当前尝试不能得到有效解时，会撤销上一步或几步的计算，再尝试其他的可能性。这种”尝试-回退”的策略，使得回溯法特别适合解决组合、排列、选择类问题。

9.2 算法设计思路

回溯法的核心思想是：1. 定义问题的解空间 2. 采用深度优先策略，系统地搜索整个解空间 3. 搜索过程中，根据约束条件进行剪枝，避免无效的搜索路径 4. 当找到合法解或遍历完整个解空间时结束搜索

回溯法常常可以表示为一个递归函数，每一次递归调用都表示一次选择。

9.3 代码实现与解析

回溯法的一般框架：

```
void backtrack(参数列表) {
    if (终止条件) {
        记录解;
        return;
    }

    for (选择 : 选择列表) {
        做选择;
        backtrack(参数列表); // 递归进入下一层决策树
        撤销选择; // 回溯到上一步
    }
}
```

9.3.1 经典问题：N 皇后问题

N 皇后问题要求在 $N \times N$ 的棋盘上放置 N 个皇后，使得它们不能互相攻击（即不能同行、同列、同对角线）。

```
class Solution {
public:
    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> result;
        vector<string> board(n, string(n, '.'));

        backtrack(board, 0, result);
        return result;
    }

private:
    void backtrack(vector<string>& board, int row, vector<vector<string>>& result) {
        // 终止条件：所有行都放置了皇后
        if (row == board.size()) {
            result.push_back(board);
            return;
        }

        int n = board.size();
        // 尝试在当前行的每一列放置皇后
        for (int col = 0; col < n; col++) {
            if (!isValid(board, row, col)) continue;

            // 做选择
```

```
            board[row][col] = 'Q';

            // 递归到下一行
            backtrack(board, row + 1, result);

            // 撤销选择（回溯）
            board[row][col] = '.';
        }
    }

    // 检查在 (row, col) 位置放置皇后是否合法
    bool isValid(vector<string>& board, int row, int col) {
        int n = board.size();

        // 检查列
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 'Q') return false;
        }

        // 检查左上方对角线
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q') return false;
        }

        // 检查右上方对角线
        for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
            if (board[i][j] == 'Q') return false;
        }

        return true;
    }
};
```

9.4 回溯法与深度优先搜索的关系

回溯法可以看作是一种特殊的深度优先搜索 (DFS)，但有以下区别：1. 回溯法强调的是试探与回退的过程，主要用于求解问题的所有解 2. DFS 是一种搜索策略，强调的是搜索的顺序 3. 回溯法中通常涉及到状态的选择、探索和撤销，而 DFS 主要关注访问顺序

9.5 复杂度分析

- 时间复杂度：O(k * n^k)，其中 n 是选择数，k 是解的长度
  - 实际时间复杂度取决于剪枝效率，上述是最坏情况
- 空间复杂度：O(k)，主要是递归调用栈的深度

9.6 典型应用场景

回溯法适用于以下问题类型：

1. 【组合问题】：从 n 个数中找出 k 个数的所有组合
2. 【排列问题】：n 个数的所有排列
3. 【切割问题】：将字符串切割成满足特定条件的片段
4. 【子集问题】：求一个集合的所有子集
5. 【棋盘问题】：如 N 皇后、数独等
6. 【图的着色问题】
7. 【迷宫问题】：寻找所有可行路径

9.7 典型例题分析

9.7.1 例题 1：全排列问题（LeetCode 46）

问题描述：给定一个不含重复数字的数组，返回其所有可能的全排列。

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> result;
        vector<int> current;
        vector<bool> used(nums.size(), false);

        backtrack(nums, used, current, result);
        return result;
    }

private:
    void backtrack(const vector<int>& nums, vector<bool>& used, vector<int>& current, vector<vector<int>>& resul
        // 终止条件：排列长度等于数组长度
        if (current.size() == nums.size()) {
            result.push_back(current);
            return;
        }

        // 尝试每个数字
        for (int i = 0; i < nums.size(); i++) {
            // 跳过已经使用的数字
            if (used[i]) continue;

            // 做选择
            used[i] = true;
            current.push_back(nums[i]);

            // 递归
            backtrack(nums, used, current, result);

            // 撤销选择（回溯）
            used[i] = false;
            current.pop_back();
        }
    }
};
```

分析：- 时间复杂度：$O(n * n!)$，其中 n 是数组长度 - 空间复杂度：$O(n)$，递归深度和存储当前排列的空间 - 核心思想：通过标记数组记录已使用元素，递归生成所有可能的排列

9.7.2 例题 2：子集（LeetCode 78）

问题描述：给定一组不含重复元素的整数数组，返回该数组所有可能的子集（幂集）。

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> result;
        vector<int> current;

        backtrack(nums, 0, current, result);
        return result;
```

```
    }

private:
    void backtrack(const vector<int>& nums, int start, vector<int>& current, vector<vector<int>>& result) {
        // 每个递归状态都是一个有效的子集，直接加入结果
        result.push_back(current);

        // 从 start 开始，避免重复
        for (int i = start; i < nums.size(); i++) {
            // 做选择
            current.push_back(nums[i]);

            // 递归，注意从 i+1 开始，避免重复使用元素
            backtrack(nums, i + 1, current, result);

            // 撤销选择（回溯）
            current.pop_back();
        }
    }
};
```

分析：- 时间复杂度：$O(n * 2^{n)}$，共有 2^n 个子集，每个子集需要 $O(n)$ 时间生成 - 空间复杂度：$O(n)$，递归深度和存储当前子集的空间 - 核心思想：通过递归构建包含或不包含每个元素的所有组合

9.7.3 例题 3：组合总和（LeetCode 39）

问题描述：给定一个无重复元素的数组 candidates 和一个目标数 target，找出 candidates 中所有可以使数字和为 target 的组合。candidates 中的数字可以无限制重复被选取。

```
class Solution {
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> result;
        vector<int> current;

        backtrack(candidates, target, 0, current, result);
        return result;
    }

private:
    void backtrack(const vector<int>& candidates, int remain, int start,
                   vector<int>& current, vector<vector<int>>& result) {
        // 终止条件
        if (remain < 0) return; // 超过目标和，无效
        if (remain == 0) {
            result.push_back(current); // 找到一个有效组合
            return;
        }

        // 尝试从 start 开始的每个候选数
        for (int i = start; i < candidates.size(); i++) {
            // 做选择
            current.push_back(candidates[i]);

            // 递归，注意仍从 i 开始，因为可以重复使用元素
            backtrack(candidates, remain - candidates[i], i, current, result);
```

```
            // 撤销选择（回溯）
            current.pop_back();
        }
    }
};
```

分析：- 时间复杂度：$O(n^(T/M))$，其中 T 是目标和，M 是最小的候选数 - 空间复杂度：$O(T/M)$，递归深度最多为 T/M - 核心思想：通过回溯构建所有和为目标值的组合，允许重复使用元素

9.8 易错点与调试技巧

1. 【边界条件处理】确保递归的终止条件正确

```
// 错误写法：可能导致无限递归
void backtrack() {
    if (someCondition) {
        // 找到解
    }
    // 缺少最终的终止条件
    for (选择 : 选择列表) {
        backtrack();
    }
}

// 正确写法：有明确的终止条件
void backtrack() {
    if (终止条件) {
        return; // 明确终止递归
    }
    for (选择 : 选择列表) {
        做选择;
        backtrack();
        撤销选择; // 确保每次递归后都撤销选择
    }
}
```

2. 【回溯操作】确保每次选择后都正确回溯

```
// 错误写法：没有回溯
void backtrack() {
    for (选择 : 选择列表) {
        做选择;
        backtrack();
        // 缺少撤销选择步骤，会导致状态混乱
    }
}

// 正确写法：选择与撤销选择对应，确保状态一致性
void backtrack() {
    for (选择 : 选择列表) {
        做选择;
        backtrack();
        撤销选择; // 确保回溯到选择前的状态
    }
}
```

3. 【状态管理】注意引用传递和值传递的区别

```cpp
// 使用引用传递 current，确保回溯时正确撤销选择
void backtrack(vector<int>& current) {
    // ...
    current.push_back(val);
    backtrack(current);
    current.pop_back(); // 回溯
}
```

4. 【剪枝优化】提前排除无效路径

```cpp
void backtrack(int remain) {
    if (remain < 0) return; // 剪枝：和已经超过目标
    if (remain == 0) {
        // 找到解
        return;
    }
    // 继续搜索
}
```

9.9 优化策略

1. 【排序预处理】通过预先排序提高剪枝效率

```cpp
vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
    sort(candidates.begin(), candidates.end()); // 预先排序
    // ... 继续回溯
}
```

2. 【位运算优化】对于某些问题，可以用位运算表示状态

```cpp
// 使用位运算表示访问状态
void backtrack(int state) {
    for (int i = 0; i < n; i++) {
        if ((state & (1 << i)) == 0) { // 检查第 i 位是否已访问
            backtrack(state | (1 << i)); // 设置第 i 位为已访问
        }
    }
}
```

3. 【记忆化搜索】缓存已经计算过的结果

```cpp
unordered_map<string, bool> memo;

bool backtrackWithMemo(string& s, int start, vector<string>& wordDict) {
    if (start == s.size()) return true;

    string key = to_string(start);
    if (memo.count(key)) return memo[key];

    bool result = false;
    // ... 执行回溯

    memo[key] = result; // 记忆结果
    return result;
}
```

10 动态规划

10.1 思路与推导（金色传说）

目标是选出长度为 m 的互不相同下标序列

$$[m_1, \dots, m_m]$$

，最大化

$$\sum_{i=1}^{m} A[m_i] \; - \; \sum_{i=1}^{m-1} |B[m_i] - B[m_{i+1}]|$$

10.2 关键化简：按 BBB 排序、罚分望文生义“望两端”

对任意给定的集合

$$S = \{m_1, \dots, m_m\} S = m1, ..., mm$$

_{i=1}^{m-1} |B[m_i]-B[m_{i+1}]|$$ 尽量小，最佳排列是让 B **单调**（不降或不升）排序（由三角不等式可证）。当按 BBB 非降顺序排列时，绝对差分项会**望两端伸缩并且望两端相消**：

$$\sum_{i=1}^{m-1} (B[i_{k+1}] - B[i_k]) \; = \; B[i_m] - B[i_1]$$

因此**最优序的罚分只与端点的 B 值有关**，问题等价为在按 B 升序的序列中，选出 m 个位置

$$i_1 < \cdots < i_m$$

最大化

$$\sum_{t=1}^{m} A[i_t] \; - \; (B[i_m] - B[i_1])$$

进一步把罚分“分摊”为逐步转移的边代价（会望两端相消）：

$$\sum_{t=1}^{m} A[i_t] \; - \; \sum_{t=1}^{m-1} (B[i_{t+1}] - B[i_t])$$

这与原式在单调序中等价。

10.3 DP 设计

设按 B 升序后得到对

$$(B_i, A_i)$$

的序列（遇到相同 B 时顺序任意）。定义：

- $$dp[k][i]$$

  ：在前 i 个位置里，**以第 i 个位置结尾**、长度为 k 的最优值。

  转移（只允许从更早位置

$$j < i$$

连接到

$$i$$

）：

$$dp[k][i] \; = \; A_i \; + \; \max_{j<i} \Big(dp[k-1][j] - (B_i - B_j) \Big)$$

将与 i 相关的项拆开：

$$dp[k][i] \; = \; A_i - B_i \; + \; \max_{j<i} \big(dp[k-1][j] + B_j \big)$$

故对每个层数 k，维护一个前缀最大值

$$pref\_max[k-1][i-1] = \max_{j \le i-1} (dp[k-1][j] + B_j)$$

即可 $O(1)$ 得到转移：

$$dp[k][i] = A_i - B_i + pref\_max[k-1][i-1]$$

边界：

- $$dp[1][i] = A_i$$

  （只选自己，无边代价）。
- 答案为

  $$\max_i dp[m][i]$$

  。

  说明：相同 B 值间转移时，边代价是 0，等式同样成立；互异下标由

  $$j < i$$

  自然保证。

10.3.1 参考实现（C++17）

```cpp
#include <bits/stdc++.h>
using namespace std;

using i64 = long long;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    if (!(cin >> n >> m)) return 0;
    vector<i64> A(n), B(n);
    for (int i = 0; i < n; ++i) cin >> A[i];
    for (int i = 0; i < n; ++i) cin >> B[i];

    // 将 (B, A) 按 B 升序排列
    vector<pair<i64,i64>> v(n);
    for (int i = 0; i < n; ++i) v[i] = {B[i], A[i]};
    sort(v.begin(), v.end()); // sort by B

    // 提取排序后的数组，便于书写
    vector<i64> Bb(n), Aa(n);
    for (int i = 0; i < n; ++i) {
        Bb[i] = v[i].first;
        Aa[i] = v[i].second;
    }

    // 特判：m==1 直接取最大 A
    if (m == 1) {
        i64 ans = *max_element(Aa.begin(), Aa.end());
```

```
        cout << ans << "\n";
        return 0;
    }

    // dp_prev[i] 表示 dp[k-1][i], dp_cur[i] 表示 dp[k][i]
    const i64 NEG_INF = numeric_limits<i64>::min()/4;
    vector<i64> dp_prev(n, NEG_INF), dp_cur(n, NEG_INF);

    // 初始化 k=1 层: dp[1][i] = A[i]
    for (int i = 0; i < n; ++i) dp_prev[i] = Aa[i];

    // 逐层推进 k = 2..m
    for (int k = 2; k <= m; ++k) {
        // 维护前缀最大: pref_max[j] = max_{t<=j}( dp_prev[t] + B[t] )
        i64 pref_max = NEG_INF;

        for (int i = 0; i < n; ++i) {
            // 在更新 dp_cur[i] 之前, pref_max 应该是 j<i 的最大值
            if (i > 0) {
                // dp[k][i] = A[i] - B[i] + max_{j<i}(dp[k-1][j] + B[j])
                dp_cur[i] = Aa[i] - Bb[i] + pref_max;
            } else {
                dp_cur[i] = NEG_INF; // 不能从空集转移来形成长度 >=2
            }
            // 然后用当前位置 i 更新 pref_max, 以供下一个 i 使用
            pref_max = max(pref_max, dp_prev[i] + Bb[i]);
        }
        dp_prev.swap(dp_cur);
        fill(dp_cur.begin(), dp_cur.end(), NEG_INF);
    }

    // 答案: max_i dp[m][i]
    i64 ans = *max_element(dp_prev.begin(), dp_prev.end());
    cout << ans << "\n";
    return 0;
}
```

10.3.2 使用说明与要点

- 输入格式：第一行 n m，第二行 n 个 A，第三行 n 个 B（可按你题库的既定格式微调）。
- 数据范围建议使用 long long。
- 复杂度：

$$O(n \log n + nm)$$

。当 nnn 上万、mmm 上百仍可接受；若 mmm 接近 nnn 且都很大，可考虑剪枝或启发式，但这已经是较紧凑的精确解法。

10.4 典型例题分析

10.4.1 例题 1：最长递增子序列 (LIS)(线性 DP)

```
// 动态规划解法，时间复杂度 O(n^2)
int lengthOfLIS(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;

    // dp[i] 表示以 nums[i] 结尾的最长递增子序列的长度
    vector<int> dp(n, 1);
```

```
    int maxLen = 1;

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        maxLen = max(maxLen, dp[i]);
    }

    return maxLen;
}

// 优化解法，使用二分查找，时间复杂度 O(nlogn)
int lengthOfLIS_optimized(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;

    // tails[i] 表示长度为 i+1 的递增子序列的末尾元素的最小值
    vector<int> tails(n, 0);
    int len = 0;

    for (int num : nums) {
        // 二分查找，找到第一个大于等于 num 的位置
        int left = 0, right = len;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (tails[mid] < num) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        tails[left] = num;
        if (left == len) len++;
    }

    return len;
}
```

10.4.2 例题 2：0-1 背包问题（背包 DP

10.4.3 ）

```
// 0-1 背包问题，二维 DP 数组实现
int knapsack(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    // dp[i][j] 表示前 i 个物品，背包容量为 j 时的最大价值
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= capacity; j++) {
            if (weights[i-1] <= j) {
                // 选择第 i 个物品或不选
```

```
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-weights[i-1]] + values[i-1]);
            } else {
                // 不能选择第 i 个物品
                dp[i][j] = dp[i-1][j];
            }
        }
    }

    return dp[n][capacity];
}

// 0-1 背包问题，一维 DP 数组优化实现
int knapsack_optimized(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    // dp[j] 表示背包容量为 j 时的最大价值
    vector<int> dp(capacity + 1, 0);

    for (int i = 0; i < n; i++) {
        // 必须逆序遍历，防止物品被重复使用
        for (int j = capacity; j >= weights[i]; j--) {
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
        }
    }

    return dp[capacity];
}
```

10.4.4 例题 3：编辑距离

```
// 编辑距离问题
int minDistance(string word1, string word2) {
    int m = word1.length();
    int n = word2.length();

    // dp[i][j] 表示 word1 前 i 个字符转换到 word2 前 j 个字符需要的最少操作数
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    // 边界条件
    for (int i = 0; i <= m; i++) {
        dp[i][0] = i;  // 删除 word1 的 i 个字符
    }
    for (int j = 0; j <= n; j++) {
        dp[0][j] = j;  // 插入 j 个字符到 word1
    }

    // 状态转移
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (word1[i-1] == word2[j-1]) {
                dp[i][j] = dp[i-1][j-1];  // 无需操作
            } else {
                dp[i][j] = min(dp[i-1][j-1],  // 替换操作
                               min(dp[i-1][j],    // 删除操作
                                   dp[i][j-1]));  // 插入操作
                dp[i][j] += 1;  // 操作数 +1
            }
```

```
        }
    }

    return dp[m][n];
}
```

10.4.5 例题 4：旅行商问题（状态压缩 DP）

```
// 旅行商问题简化版
int tsp(vector<vector<int>>& graph) {
    int n = graph.size();
    int all = (1 << n) - 1; // 所有城市都被访问的状态

    // dp[state][i] 表示当前在城市 i，已访问的城市集合为 state 时的最短路径长度
    vector<vector<int>> dp(1 << n, vector<int>(n, INT_MAX));

    // 初始状态：从城市 0 出发
    dp[1][0] = 0;

    // 枚举所有状态
    for (int state = 1; state < (1 << n); state++) {
        for (int i = 0; i < n; i++) {
            // 如果城市 i 不在当前 state 中，跳过
            if (!(state & (1 << i))) continue;

            // 枚举前一个城市 j
            for (int j = 0; j < n; j++) {
                // 如果城市 j 不在前一个状态中，或者 j==i，跳过
                if (i == j || !(state & (1 << j))) continue;

                // 状态转移
                dp[state][i] = min(dp[state][i], dp[state ^ (1 << i)][j] + graph[j][i]);
            }
        }
    }

    // 所有城市都访问后，最后回到城市 0
    int result = INT_MAX;
    for (int i = 1; i < n; i++) {
        if (graph[i][0] != INT_MAX) {
            result = min(result, dp[all][i] + graph[i][0]);
        }
    }

    return result;
}
```

10.5 常见错误和注意事项

1. **状态定义不清晰**：
   - 确保状态能够唯一表示子问题
   - 状态包含足够的信息
2. **边界条件处理不当**：
   - 仔细分析初始情况
   - 处理特殊输入，如空序列

3. **状态转移方程不正确**：
    - 考虑所有可能的转移路径
    - 确保递推关系的正确性
4. **计算顺序错误**：
    - 确保计算某状态时，其依赖的状态已计算
    - 注意数组的遍历顺序
5. **溢出问题**：
    - 注意大数运算可能的溢出
    - 必要时采用适当的取模操作
6. **优化过度导致错误**：
    - 在优化空间前确保基本算法正确
    - 空间优化后仔细检查依赖关系

10.6 01 背包问题

【问题定义】有 N 件物品和一个容量为 V 的背包，每件物品有且只有一件。第 i 件物品的重量是 $w[i]$，价值是 $v[i]$。求解将哪些物品装入背包，可使这些物品的总重量不超过背包容量，且总价值最大。

10.6.1 基本思路

使用动态规划求解，设 $dp[i][j]$ 表示考虑前 i 件物品，背包容量为 j 时能达到的最大价值。

10.6.2 状态转移方程

对于第 i 件物品，有两种选择：不选择该物品或选择该物品。- 不选择第 i 件物品：$dp[i][j] = dp[i-1][j]$ - 选择第 i 件物品（前提是能装下）：$dp[i][j] = dp[i-1][j-w[i]] + v[i]$

因此，完整的状态转移方程为：

```
dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])  (if j   w[i])
dp[i][j] = dp[i-1][j]  (if j < w[i])
```

10.6.3 代码实现

二维数组实现：

```
// 01 背包问题的二维数组实现
int knapsack01(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    // dp[i][j]: 考虑前 i 个物品，背包容量 j 下的最大价值
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= capacity; j++) {
            if (j < weights[i-1]) {
                // 背包容量不足，无法放入第 i 个物品
                dp[i][j] = dp[i-1][j];
            } else {
                // 可以选择放或不放第 i 个物品
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-weights[i-1]] + values[i-1]);
            }
        }
    }

    return dp[n][capacity];
}
```

10.6.4 空间优化

由于 dp[i][j] 只与 dp[i-1][j] 和 dp[i-1][j-w[i]] 有关，可以使用一维数组优化空间：

```
// 01 背包问题的一维数组优化
int knapsack01_optimized(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    vector<int> dp(capacity + 1, 0);

    for (int i = 0; i < n; i++) {
        for (int j = capacity; j >= weights[i]; j--) {
            // 注意这里的遍历顺序：从大到小逆序遍历
            dp[j] = max(dp[j], dp[j-weights[i]] + values[i]);
        }
    }

    return dp[capacity];
}
```

【注意】一维数组实现中，必须从大到小逆序遍历容量 j，确保每个物品只被考虑一次。

10.7 完全背包问题

【问题定义】有 N 种物品和一个容量为 V 的背包，每种物品有无限件。第 i 种物品的重量是 w[i]，价值是 v[i]。求解将哪些物品装入背包，可使这些物品的总重量不超过背包容量，且总价值最大。

10.7.1 状态转移方程

与 01 背包不同，完全背包问题中每种物品可以选择多次，因此状态转移方程为：

```
dp[i][j] = max(dp[i-1][j], dp[i][j-w[i]] + v[i])  (if j   w[i])
dp[i][j] = dp[i-1][j]  (if j < w[i])
```

注意区别：01 背包中是 dp[i-1][j-w[i]] + v[i]，而完全背包中是 dp[i][j-w[i]] + v[i]，因为物品 i 可以被重复选择。

10.7.2 代码实现

```
// 完全背包问题
int unboundedKnapsack(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= capacity; j++) {
            if (j < weights[i-1]) {
                dp[i][j] = dp[i-1][j];
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-weights[i-1]] + values[i-1]);
            }
        }
    }

    return dp[n][capacity];
}
```

10.7.3 空间优化

完全背包的一维数组优化与 01 背包类似，但遍历顺序不同：

```cpp
// 完全背包问题的一维数组优化
int unboundedKnapsack_optimized(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    vector<int> dp(capacity + 1, 0);

    for (int i = 0; i < n; i++) {
        for (int j = weights[i]; j <= capacity; j++) {
            // 注意这里的遍历顺序：从小到大正序遍历
            dp[j] = max(dp[j], dp[j-weights[i]] + values[i]);
        }
    }

    return dp[capacity];
}
```

【注意】与 01 背包不同，完全背包需要从小到大正序遍历容量 j，确保每个物品可以被多次选择。

10.8 多重背包问题

【问题定义】有 N 种物品和一个容量为 V 的背包。第 i 种物品最多有 $k[i]$ 件，每件重量为 $w[i]$，价值为 $v[i]$。求解将哪些物品装入背包，可使这些物品的总重量不超过背包容量，且总价值最大。

10.8.1 基本思路

多重背包可以看作是 01 背包和完全背包的混合：每种物品有限定数量。

10.8.2 朴素解法

通过枚举每种物品选择的数量来解决：

```cpp
// 多重背包问题的朴素解法
int multiplePack(vector<int>& weights, vector<int>& values, vector<int>& counts, int capacity) {
    int n = weights.size();
    vector<int> dp(capacity + 1, 0);

    for (int i = 0; i < n; i++) {
        for (int j = capacity; j >= 0; j--) {
            for (int k = 1; k <= counts[i] && k * weights[i] <= j; k++) {
                dp[j] = max(dp[j], dp[j - k * weights[i]] + k * values[i]);
            }
        }
    }

    return dp[capacity];
}
```

10.8.3 二进制优化

当物品数量较大时，可以使用二进制优化来降低时间复杂度：

```cpp
// 多重背包问题的二进制优化
int multiplePack_binary(vector<int>& weights, vector<int>& values, vector<int>& counts, int capacity) {
    int n = weights.size();
    vector<int> dp(capacity + 1, 0);

    // 将多个相同物品拆分成二进制组
    vector<int> new_weights, new_values;
```

```
    for (int i = 0; i < n; i++) {
        int count = counts[i];
        for (int k = 1; k <= count; k *= 2) {
            new_weights.push_back(weights[i] * k);
            new_values.push_back(values[i] * k);
            count -= k;
        }
        if (count > 0) {
            new_weights.push_back(weights[i] * count);
            new_values.push_back(values[i] * count);
        }
    }

    // 转化为 01 背包问题
    for (int i = 0; i < new_weights.size(); i++) {
        for (int j = capacity; j >= new_weights[i]; j--) {
            dp[j] = max(dp[j], dp[j - new_weights[i]] + new_values[i]);
        }
    }

    return dp[capacity];
}
```

10.9 混合背包问题

【问题定义】有 N 种物品和一个容量为 V 的背包。物品有三类：- 第一类物品只能用 1 次（01 背包）- 第二类物品可以用无限次（完全背包）- 第三类物品最多用 k 次（多重背包）

10.9.1 解题思路

根据物品的类型分别处理：

```
// 混合背包问题
int mixedKnapsack(vector<int>& weights, vector<int>& values, vector<int>& types, vector<int>& counts, int capaci
    int n = weights.size();
    vector<int> dp(capacity + 1, 0);

    for (int i = 0; i < n; i++) {
        if (types[i] == 0) {  // 01 背包
            for (int j = capacity; j >= weights[i]; j--) {
                dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
            }
        } else if (types[i] == 1) {  // 完全背包
            for (int j = weights[i]; j <= capacity; j++) {
                dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
            }
        } else {  // 多重背包
            // 二进制优化
            int count = counts[i];
            for (int k = 1; k <= count; k *= 2) {
                int new_weight = weights[i] * k;
                int new_value = values[i] * k;
                for (int j = capacity; j >= new_weight; j--) {
                    dp[j] = max(dp[j], dp[j - new_weight] + new_value);
                }
                count -= k;
            }
```

```
            if (count > 0) {
                int new_weight = weights[i] * count;
                int new_value = values[i] * count;
                for (int j = capacity; j >= new_weight; j--) {
                    dp[j] = max(dp[j], dp[j - new_weight] + new_value);
                }
            }
        }
    }

    return dp[capacity];
}
```

10.10 分组背包问题

【问题定义】有 N 组物品和一个容量为 V 的背包。每组物品有若干个，同一组内的物品最多只能选一个。每件物品的重量是 $w[i][j]$，价值是 $v[i][j]$，其中 i 是组号，j 是组内编号。求解将哪些物品装入背包，可使物品总重量不超过背包容量，且总价值最大。

10.10.1 解题思路

针对每一组物品，枚举该组选择哪件物品：

```
// 分组背包问题
int groupKnapsack(vector<vector<int>>& weights, vector<vector<int>>& values, int capacity) {
    int n = weights.size();  // 组数
    vector<int> dp(capacity + 1, 0);

    for (int i = 0; i < n; i++) {  // 枚举每一组
        for (int j = capacity; j >= 0; j--) {  // 枚举背包容量
            for (int k = 0; k < weights[i].size(); k++) {  // 枚举该组的每个物品
                if (j >= weights[i][k]) {
                    dp[j] = max(dp[j], dp[j - weights[i][k]] + values[i][k]);
                }
            }
        }
    }

    return dp[capacity];
}
```

10.11 二维背包问题

【问题定义】有 N 件物品和一个容量为 (V,M) 的背包，这里有两个限制维度（如体积和重量）。第 i 件物品需要消耗背包的空间为 $(v[i],m[i])$，价值是 $w[i]$。求解将哪些物品装入背包，可使物品的总体积不超过 V，总重量不超过 M，且总价值最大。

10.11.1 解题思路

扩展到三维 DP 数组，$dp[i][j][k]$ 表示考虑前 i 个物品，体积不超过 j，重量不超过 k 的最大价值：

```
// 二维费用背包问题
int twoDimensionKnapsack(vector<int>& weights, vector<int>& volumes, vector<int>& values, int capacity_weight, i
    int n = weights.size();
    vector<vector<int>> dp(capacity_weight + 1, vector<int>(capacity_volume + 1, 0));

    for (int i = 0; i < n; i++) {
        for (int j = capacity_weight; j >= weights[i]; j--) {
            for (int k = capacity_volume; k >= volumes[i]; k--) {
```

```
                dp[j][k] = max(dp[j][k], dp[j - weights[i]][k - volumes[i]] + values[i]);
            }
        }
    }

    return dp[capacity_weight][capacity_volume];
}
```

10.12 背包问题求方案数

【问题变形】除了求最大价值外，还可能需要求达到最大价值的方案数量。

10.12.1 解题思路

维护两个 DP 数组，一个记录最大价值，一个记录达到该价值的方案数：

```
// 背包问题求方案数
pair<int, int> knapsackWithCount(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    int mod = 1e9 + 7;
    vector<int> dp(capacity + 1, 0);  // 最大价值
    vector<int> count(capacity + 1, 0);  // 方案数量
    count[0] = 1;  // 初始状态有 1 种方案

    for (int i = 0; i < n; i++) {
        for (int j = capacity; j >= weights[i]; j--) {
            int new_value = dp[j - weights[i]] + values[i];
            if (new_value > dp[j]) {
                // 找到更大的价值，更新最大价值和方案数
                dp[j] = new_value;
                count[j] = count[j - weights[i]];
            } else if (new_value == dp[j]) {
                // 相同价值，方案数累加
                count[j] = (count[j] + count[j - weights[i]]) % mod;
            }
        }
    }

    return {dp[capacity], count[capacity]};
}
```

10.13 背包问题输出方案

【问题变形】需要输出具体选择了哪些物品。

10.13.1 解题思路

记录每一步的选择，然后回溯：

```
// 背包问题输出方案
vector<int> knapsackWithSolution(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));
    vector<vector<bool>> selected(n + 1, vector<bool>(capacity + 1, false));

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= capacity; j++) {
```

```
            if (j < weights[i-1]) {
                dp[i][j] = dp[i-1][j];
                selected[i][j] = false;
            } else {
                int not_take = dp[i-1][j];
                int take = dp[i-1][j-weights[i-1]] + values[i-1];
                if (take > not_take) {
                    dp[i][j] = take;
                    selected[i][j] = true;
                } else {
                    dp[i][j] = not_take;
                    selected[i][j] = false;
                }
            }
        }
    }

    // 回溯找出选择的物品
    vector<int> solution;
    int i = n, j = capacity;
    while (i > 0) {
        if (selected[i][j]) {
            solution.push_back(i-1);  // 选择了第 i 个物品
            j -= weights[i-1];
        }
        i--;
    }

    return solution;
}
```

10.14 多维背包、泛化背包问题

背包问题可以扩展到多个维度，或者添加更多的约束条件。解决这类问题的核心思想是理解状态定义和转移方程，然后根据具体问题调整 DP 数组的维度和转移方式。

10.15 背包问题变种

10.15.1 恰好装满背包

当要求恰好装满背包时，初始化方式有所不同：- $dp[0] = 0$ - $dp[j] = -\infty$ $(j > 0)$

```
// 恰好装满背包的 01 背包问题
int knapsack01_exact(vector<int>& weights, vector<int>& values, int capacity) {
    int n = weights.size();
    vector<int> dp(capacity + 1, INT_MIN); // 除了 dp[0]=0 外，其他初始化为负无穷
    dp[0] = 0;

    for (int i = 0; i < n; i++) {
        for (int j = capacity; j >= weights[i]; j--) {
            dp[j] = max(dp[j], dp[j-weights[i]] + values[i]);
        }
    }

    // 如果结果为负无穷，表示无法恰好装满
    return dp[capacity] < 0 ? -1 : dp[capacity];
}
```

10.15.2 最小代价问题

有时背包问题可以反向思考，寻找达到某一目标的最小代价：

```
// 最小代价背包问题
int minCostKnapsack(vector<int>& weights, vector<int>& costs, int target_weight) {
    int n = weights.size();
    vector<int> dp(target_weight + 1, INT_MAX);
    dp[0] = 0;

    for (int i = 0; i < n; i++) {
        for (int j = target_weight; j >= weights[i]; j--) {
            if (dp[j - weights[i]] != INT_MAX) {
                dp[j] = min(dp[j], dp[j - weights[i]] + costs[i]);
            }
        }
    }

    return dp[target_weight] == INT_MAX ? -1 : dp[target_weight];
}
```

10.16 最长递增子序列

```
// 使用二分查找并输出最长递增子序列
vector<int> findLIS_optimized(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return {};

    vector<int> tails(n, 0);  // tails[i] 表示长度为 i+1 的 LIS 的末尾最小值
    vector<int> indices(n, 0);  // indices[i] 表示长度为 i+1 的 LIS 的末尾元素在 nums 中的索引
    vector<int> prev(n, -1);  // prev[i] 表示 nums[i] 前一个元素在 LIS 中的索引

    int len = 0;  // 当前 LIS 长度

    for (int i = 0; i < n; i++) {
        int left = 0, right = len;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (tails[mid] < nums[i]) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        // 如果当前位置前有元素，记录前驱
        if (left > 0) {
            prev[i] = indices[left - 1];
        }

        tails[left] = nums[i];
        indices[left] = i;

        if (left == len) {
            len++;
        }
    }
```

```
    // 回溯构建 LIS
    vector<int> lis;
    int index = indices[len - 1];
    while (index != -1) {
        lis.push_back(nums[index]);
        index = prev[index];
    }

    reverse(lis.begin(), lis.end());
    return lis;
}
```

10.17 变种问题

10.17.1 最长递减子序列 (LDS)

只需将原序列取负或反向比较即可转化为 LIS 问题。

```
// 最长递减子序列
int lengthOfLDS(vector<int>& nums) {
    vector<int> reversed(nums);
    for (int& num : reversed) {
        num = -num;  // 取负转换为 LIS 问题
    }
    return lengthOfLIS(reversed);
}
```

10.17.2 最长不降子序列（可以包含相等元素）

只需修改比较条件：

```
// 最长不降子序列
int lengthOfLNDS(vector<int>& nums) {
    int n = nums.size();
    vector<int> dp(n, 1);

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] >= nums[j]) {  // 允许相等
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
    }

    return *max_element(dp.begin(), dp.end());
}
```

10.17.3 最长交替子序列（最长摆动子序列）

找出序列中最长的子序列，使得子序列中相邻元素的差交替出现正负。

```
// 最长交替子序列
int lengthOfLAS(vector<int>& nums) {
    int n = nums.size();
    if (n < 2) return n;

    // dp[i][0] 表示以 nums[i] 结尾且最后一个差为负的最长交替子序列长度
```

```
    // dp[i][1] 表示以 nums[i] 结尾且最后一个差为正的最长交替子序列长度
    vector<vector<int>> dp(n, vector<int>(2, 1));

    int result = 1;

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j]) {
                // 当前差为正，前一个差应为负
                dp[i][1] = max(dp[i][1], dp[j][0] + 1);
            } else if (nums[i] < nums[j]) {
                // 当前差为负，前一个差应为正
                dp[i][0] = max(dp[i][0], dp[j][1] + 1);
            }
        }
        result = max({result, dp[i][0], dp[i][1]});
    }

    return result;
}
```

10.17.4 二维 LIS（信封问题）

给定一组信封的宽度和高度，一个信封可以放入另一个信封当且仅当它的宽度和高度都小于另一个信封。求最多可以嵌套的信封数量。

```
// 信封嵌套问题
int maxEnvelopes(vector<vector<int>>& envelopes) {
    // 按宽度升序排序，宽度相同时按高度降序排序
    sort(envelopes.begin(), envelopes.end(), [](const vector<int>& a, const vector<int>& b) {
        return a[0] < b[0] || (a[0] == b[0] && a[1] > b[1]);
    });

    // 在高度上应用 LIS 算法
    vector<int> heights;
    for (auto& env : envelopes) {
        heights.push_back(env[1]);
    }

    return lengthOfLIS_optimized(heights);
}
```

10.18 最长公共子序列

10.18.1 状态定义

设 $\mathrm{dp[i][j]}$ 表示序列 X 的前 i 个字符与序列 Y 的前 j 个字符的最长公共子序列长度。

10.18.2 状态转移方程

对于每一对位置 $(\mathrm{i,j})$，有以下情况：

1. 如果 $\mathrm{X[i] = Y[j]}$，则 $\mathrm{dp[i][j] = dp[i\text{-}1][j\text{-}1] + 1}$（当前字符相同，最长公共子序列可以加上这个字符）
2. 如果 $\mathrm{X[i] \neq Y[j]}$，则 $\mathrm{dp[i][j] = \max(dp[i\text{-}1][j], dp[i][j\text{-}1])}$（当前字符不同，最长公共子序列取决于去掉 $\mathrm{X[i]}$ 或 $\mathrm{Y[j]}$ 后的最长公共子序列）

10.18.3 初始条件

- $\mathrm{dp[0][j] = 0}$（X 为空序列时，与 Y 的任何子序列的 LCS 长度为 0）
- $\mathrm{dp[i][0] = 0}$（Y 为空序列时，与 X 的任何子序列的 LCS 长度为 0）

10.18.4 代码实现

```
// 计算两个字符串的最长公共子序列长度
int longestCommonSubsequence(string X, string Y) {
    int m = X.length();
    int n = Y.length();

    // dp[i][j] 表示 X[0...i-1] 和 Y[0...j-1] 的 LCS 长度
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i-1] == Y[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }

    return dp[m][n];
}
```

10.19 输出最长公共子序列

除了计算 LCS 的长度外，有时还需要输出具体的 LCS 内容。可以通过回溯 DP 表格来实现：

```
// 计算并输出最长公共子序列
string getLCS(string X, string Y) {
    int m = X.length();
    int n = Y.length();

    // 计算 dp 数组
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i-1] == Y[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }

    // 回溯构造 LCS
    string lcs;
    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (X[i-1] == Y[j-1]) {
            // 当前字符是 LCS 的一部分
            lcs = X[i-1] + lcs;
            i--; j--;
        } else if (dp[i-1][j] > dp[i][j-1]) {
            // LCS 来自于去掉 X[i]
            i--;
        } else {
```

```
            // LCS 来自于去掉 Y[j]
            j--;
        }
    }

    return lcs;
}
```

11 并查集

11.1 算法概述

【并查集】是一种树形数据结构，用于处理一些不相交集合的合并及查询问题。主要支持两种操作：- 查找（Find）：确定元素属于哪一个子集 - 合并（Union）：将两个子集合并成一个集合

11.2 算法设计思路

并查集主要用于解决连接性问题，如”网络中节点 A 和节点 B 是否连通”等。它的核心思想是：1. 每个集合用一棵树表示，树中的节点表示元素 2. 树的根节点作为集合的代表元素 3. 通过”压缩”查找路径和”按秩合并”来优化操作效率

11.3 代码实现与解析

11.3.1 基本并查集实现

```
// 初始化并查集，parent[i] = i 表示每个元素初始时都是独立的集合
void init(int n) {
    for (int i = 0; i < n; i++) {
        parent[i] = i; // 每个节点的父节点初始为自己
    }
}

// 查找操作 - 寻找元素 x 所属集合的代表元素
int find(int x) {
    if (parent[x] == x) return x; // 如果 x 是根节点，则返回 x
    return find(parent[x]); // 递归查找 x 的父节点
}

// 合并操作 - 将元素 x 所在集合与元素 y 所在集合合并
void union_sets(int x, int y) {
    int root_x = find(x); // 找到 x 的根节点
    int root_y = find(y); // 找到 y 的根节点

    if (root_x != root_y) {
        parent[root_x] = root_y; // 将 x 的根节点指向 y 的根节点
    }
}
```

11.3.2 优化后的并查集实现

```
// 初始化带秩的并查集
void init(int n) {
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 1; // 初始化每个节点的秩为 1
    }
```

```
}

// 路径压缩优化的查找操作
int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]); // 路径压缩：直接指向根节点
    }
    return parent[x];
}

// 按秩合并优化的合并操作
void union_sets(int x, int y) {
    int root_x = find(x);
    int root_y = find(y);

    if (root_x == root_y) return; // 已经在同一集合中

    // 按秩合并：将较小的树连接到较大的树下
    if (rank[root_x] < rank[root_y]) {
        parent[root_x] = root_y;
    } else if (rank[root_x] > rank[root_y]) {
        parent[root_y] = root_x;
    } else {
        parent[root_y] = root_x;
        rank[root_x]++; // 秩相同时，新树的秩加 1
    }
}
```

11.4 逻辑图解

```
初始状态：[1] [2] [3] [4] [5]  （5个独立元素）
执行union(1,2)：[1,2] [3] [4] [5]
执行union(3,4)：[1,2] [3,4] [5]
执行union(1,3)：[1,2,3,4] [5]
```

11.5 复杂度分析

- 时间复杂度：
  - 未优化时：查找操作 $O(n)$，合并操作 $O(n)$
  - 路径压缩 + 按秩合并后：均摊近乎 $O(1)$，严格来说是 $O(\ (n))$， 是阿克曼函数的反函数，在实际使用中可视为常数
- 空间复杂度：$O(n)$ 存储父节点数组和秩数组

11.6 典型应用场景

1. 判断网络连接性问题
2. 最小生成树的 Kruskal 算法
3. 等价类划分
4. 动态连通性问题

11.7 典型例题分析

11.7.1 例题：朋友圈问题

有 n 个人，给出他们之间的朋友关系，求朋友圈的数量（朋友的朋友也是朋友）。

```cpp
int findCircleNum(vector<vector<int>>& M) {
    int n = M.size();
    vector<int> parent(n);
    vector<int> rank(n, 1);

    // 初始化并查集
    for (int i = 0; i < n; i++) {
        parent[i] = i;
    }

    // 合并朋友关系
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (M[i][j] == 1) {
                union_sets(parent, rank, i, j);
            }
        }
    }

    // 计算朋友圈数量
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (parent[i] == i) { // 根节点的数量就是集合的数量
            count++;
        }
    }
    return count;
}

// 查找函数
int find(vector<int>& parent, int x) {
    if (parent[x] != x) {
        parent[x] = find(parent, parent[x]); // 路径压缩
    }
    return parent[x];
}

// 合并函数
void union_sets(vector<int>& parent, vector<int>& rank, int x, int y) {
    int root_x = find(parent, x);
    int root_y = find(parent, y);

    if (root_x == root_y) return;

    // 按秩合并
    if (rank[root_x] < rank[root_y]) {
        parent[root_x] = root_y;
    } else if (rank[root_x] > rank[root_y]) {
        parent[root_y] = root_x;
    } else {
        parent[root_y] = root_x;
        rank[root_x]++;
    }
}
```

11.7.2 易错点与调试技巧

1. 【路径压缩】是关键优化，不使用可能导致树非常高

```
// 错误写法（没有路径压缩）
int find(int x) {
    if (parent[x] == x) return x;
    return find(parent[x]); // 没有更新 parent[x]
}

// 正确写法
int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]); // 路径压缩
    }
    return parent[x];
}
```

2. 【按秩合并】对性能也很重要

```
// 不加判断的合并可能导致树很高
void union_wrong(int x, int y) {
    int root_x = find(x);
    int root_y = find(y);
    parent[root_x] = root_y; // 总是将 x 连到 y 下，可能导致很高的树
}
```

12 堆

12.1 基本概念

【定义】堆是一种满足堆属性的完全二叉树：- **最大堆**：每个节点的值都大于或等于其子节点的值 - **最小堆**：每个节点的值都小于或等于其子节点的值

【关键特性】1. 堆是一个完全二叉树，通常使用数组实现 2. 根节点是整个堆中的最大值（最大堆）或最小值（最小堆）3. 插入和删除操作的时间复杂度为 $O(\log n)$ 4. 查找最大/最小元素的时间复杂度为 $O(1)$

12.2 堆的实现

12.2.1 数组表示

在数组实现中，对于索引从 0 开始的数组：- 父节点：`parent(i) = (i - 1) / 2` - 左子节点：`left(i) = 2 * i + 1` - 右子节点：`right(i) = 2 * i + 2`

对于索引从 1 开始的数组：- 父节点：`parent(i) = i / 2` - 左子节点：`left(i) = 2 * i` - 右子节点：`right(i) = 2 * i + 1`

12.2.2 基本操作

12.2.2.1 上浮操作 (Sift Up)

当一个节点的值大于其父节点（最大堆）或小于其父节点（最小堆）时，需要将该节点上浮：

```
// 最大堆的上浮操作（索引从 0 开始）
void siftUp(vector<int>& heap, int i) {
    int parent = (i - 1) / 2;
    while (i > 0 && heap[i] > heap[parent]) {
        swap(heap[i], heap[parent]);
        i = parent;
        parent = (i - 1) / 2;
    }
}
```

12.2.2.2 下滤操作 (Sift Down)

当一个节点的值小于其子节点（最大堆）或大于其子节点（最小堆）时，需要将该节点下沉：

```
// 最大堆的下沉操作（索引从 0 开始）
void siftDown(vector<int>& heap, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // 找出当前节点和其子节点中的最大值
    if (left < n && heap[left] > heap[largest]) {
        largest = left;
    }
    if (right < n && heap[right] > heap[largest]) {
        largest = right;
    }

    // 如果最大值不是当前节点，则交换并继续下沉
    if (largest != i) {
        swap(heap[i], heap[largest]);
        siftDown(heap, n, largest);
    }
}
```

12.2.2.3 插入操作

将新元素添加到堆的末尾，然后执行上浮操作：

```
void insert(vector<int>& heap, int val) {
    // 将新元素添加到末尾
    heap.push_back(val);

    // 执行上浮操作
    siftUp(heap, heap.size() - 1);
}
```

12.2.2.4 删除最大/最小元素

将根节点与最后一个节点交换，移除最后一个节点，然后对新的根节点执行下沉操作：

```
int extractMax(vector<int>& heap) {
    if (heap.empty()) {
        throw runtime_error("Heap is empty");
    }

    int maxVal = heap[0];  // 保存最大值

    // 将最后一个元素放到根节点，并缩小堆
    heap[0] = heap.back();
    heap.pop_back();

    // 如果堆不为空，对新的根节点执行下沉操作
    if (!heap.empty()) {
        siftDown(heap, heap.size(), 0);
    }

    return maxVal;
}
```

12.2.3 堆的构建

从无序数组构建堆的方式有两种：

```
vector<int> buildHeap(const vector<int>& arr) {
    vector<int> heap;
    for (int val : arr) {
        insert(heap, val);
    }
    return heap;
}
```

12.2.3.1 1. 逐个插入（时间复杂度 $O(n \log n)$）

12.2.3.2 2. 原地建堆（时间复杂度 $O(n)$） 从最后一个非叶节点开始，依次向前执行下沉操作：

```
// 原地建堆，时间复杂度 O(n)
void buildHeap(vector<int>& arr) {
    int n = arr.size();

    // 从最后一个非叶节点开始，依次向前执行下沉操作
    for (int i = n / 2 - 1; i >= 0; i--) {
        siftDown(arr, n, i);
    }
}
```

12.3 堆排序

堆排序是一种基于比较的排序算法，它利用堆的性质进行排序：

```
void heapSort(vector<int>& arr) {
    int n = arr.size();

    // 构建最大堆
    buildHeap(arr);

    // 一个个取出堆顶元素
    for (int i = n - 1; i > 0; i--) {
        // 将堆顶元素（当前最大值）与数组末尾交换
        swap(arr[0], arr[i]);

        // 对剩余元素进行下沉操作，重新构建最大堆
        siftDown(arr, i, 0);
    }
}
```

堆排序的时间复杂度为 $O(n \log n)$，空间复杂度为 $O(1)$，是一种原地排序算法。

12.4 STL 中的优先队列

C++ 标准库提供了优先队列容器 `priority_queue`，它是基于堆实现的：

```
#include <queue>
#include <vector>
using namespace std;

// 最大堆（默认）
priority_queue<int> maxHeap;

// 最小堆
```

```cpp
priority_queue<int, vector<int>, greater<int>> minHeap;

// 自定义比较函数
struct compare {
    bool operator()(const pair<int, int>& a, const pair<int, int>& b) {
        return a.first < b.first;  // 按 first 降序排列
    }
};
priority_queue<pair<int, int>, vector<pair<int, int>>, compare> customHeap;
```

12.4.1 基本操作

```cpp
// 插入元素
maxHeap.push(10);

// 访问堆顶元素
int top = maxHeap.top();

// 移除堆顶元素
maxHeap.pop();

// 获取堆的大小
int size = maxHeap.size();

// 判断堆是否为空
bool isEmpty = maxHeap.empty();
```

12.5 堆的应用

12.5.1 1. Top-K 问题

在 n 个元素中找出最大/最小的 k 个元素:

```cpp
// 找出数组中最大的 k 个元素
vector<int> findTopK(const vector<int>& nums, int k) {
    // 使用最小堆
    priority_queue<int, vector<int>, greater<int>> minHeap;

    // 只保留 k 个最大元素
    for (int num : nums) {
        minHeap.push(num);
        if (minHeap.size() > k) {
            minHeap.pop();  // 弹出最小的元素
        }
    }

    // 将结果转换为数组
    vector<int> result;
    while (!minHeap.empty()) {
        result.push_back(minHeap.top());
        minHeap.pop();
    }

    // 按升序排序结果
    reverse(result.begin(), result.end());
    return result;
}
```

12.5.2 2. 堆优化的 Dijkstra 算法

使用堆（优先队列）可以优化 Dijkstra 算法，将时间复杂度从 $O(V^2)$ 降低到 $O(E \log V)$：

```
// 堆优化的 Dijkstra 算法（参见图论算法章节）
void dijkstra(vector<vector<pair<int, int>>>& graph, int start) {
    int n = graph.size();
    vector<int> dist(n, INT_MAX);
    dist[start] = 0;

    // {距离, 顶点} 的优先队列（最小堆）
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, start});

    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();

        if (d > dist[u]) continue;  // 跳过已经处理过的节点

        for (auto [v, w] : graph[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
}
```

12.5.3 3. 中位数维护

使用一个最大堆和一个最小堆可以高效地维护数据流的中位数：

```
class MedianFinder {
private:
    priority_queue<int> maxHeap;  // 存储较小的一半元素
    priority_queue<int, vector<int>, greater<int>> minHeap;  // 存储较大的一半元素

public:
    void addNum(int num) {
        // 始终保持 maxHeap.size() >= minHeap.size()
        if (maxHeap.empty() || num <= maxHeap.top()) {
            maxHeap.push(num);
        } else {
            minHeap.push(num);
        }

        // 调整两个堆的大小，使它们的大小差不超过 1
        if (maxHeap.size() > minHeap.size() + 1) {
            minHeap.push(maxHeap.top());
            maxHeap.pop();
        } else if (maxHeap.size() < minHeap.size()) {
            maxHeap.push(minHeap.top());
            minHeap.pop();
        }
    }
```

```
    double findMedian() {
        if (maxHeap.size() > minHeap.size()) {
            return maxHeap.top();
        } else {
            return (maxHeap.top() + minHeap.top()) / 2.0;
        }
    }
};
```

12.5.4 4. 合并 K 个有序链表/数组

使用最小堆可以高效地合并多个有序数据结构：

```
ListNode* mergeKLists(vector<ListNode*>& lists) {
    // 自定义比较函数
    auto compare = [](ListNode* a, ListNode* b) {
        return a->val > b->val;  // 最小堆
    };

    // 创建最小堆
    priority_queue<ListNode*, vector<ListNode*>, decltype(compare)> minHeap(compare);

    // 将每个链表的头节点加入堆
    for (ListNode* list : lists) {
        if (list) minHeap.push(list);
    }

    // 哨兵节点
    ListNode dummy(0);
    ListNode* tail = &dummy;

    // 依次取出堆顶元素，并将下一个节点加入堆
    while (!minHeap.empty()) {
        ListNode* curr = minHeap.top();
        minHeap.pop();

        tail->next = curr;
        tail = tail->next;

        if (curr->next) {
            minHeap.push(curr->next);
        }
    }

    return dummy.next;
}
```

12.6 相关变种和扩展

12.6.1 1. 索引堆

索引堆维护元素的索引而非元素本身，适用于需要随机访问堆中元素的场景：

```
class IndexMinHeap {
private:
    vector<int> data;    // 原始数据
    vector<int> indexes; // 索引数组，indexes[i] 表示堆中第 i 个位置存储的元素在 data 中的索引
```

```
    vector<int> reverse; // 反向索引，reverse[i] 表示索引 i 在 indexes 中的位置，如果不在堆中则为-1

    // 比较函数
    bool less(int i, int j) {
        return data[indexes[i]] < data[indexes[j]];
    }

    // 交换 indexes[i] 和 indexes[j]
    void swap(int i, int j) {
        std::swap(indexes[i], indexes[j]);
        reverse[indexes[i]] = i;
        reverse[indexes[j]] = j;
    }

    // 上浮操作
    void siftUp(int k) {
        while (k > 0) {
            int parent = (k - 1) / 2;
            if (!less(k, parent)) break;
            swap(k, parent);
            k = parent;
        }
    }

    // 下沉操作
    void siftDown(int k) {
        int n = indexes.size();
        while (2 * k + 1 < n) {
            int j = 2 * k + 1; // 左子节点
            if (j + 1 < n && less(j + 1, j)) j++; // 取两个子节点中较小者
            if (!less(j, k)) break;
            swap(k, j);
            k = j;
        }
    }

public:
    IndexMinHeap(int capacity) {
        data.resize(capacity);
        indexes.clear();
        reverse.resize(capacity, -1);
    }

    int size() {
        return indexes.size();
    }

    bool isEmpty() {
        return indexes.empty();
    }

    // 插入元素
    void insert(int i, int val) {
        data[i] = val;
        indexes.push_back(i);
        reverse[i] = indexes.size() - 1;
        siftUp(indexes.size() - 1);
```

```
    }

    // 弹出最小元素
    int extractMin() {
        int minIndex = indexes[0];
        swap(0, indexes.size() - 1);
        reverse[indexes.back()] = -1;
        indexes.pop_back();
        siftDown(0);

        return minIndex;
    }

    // 更新元素值
    void update(int i, int val) {
        data[i] = val;
        siftUp(reverse[i]);
        siftDown(reverse[i]);
    }

    // 判断索引 i 是否在堆中
    bool contains(int i) {
        return reverse[i] != -1;
    }

    // 获取最小元素的索引
    int getMinIndex() {
        return indexes[0];
    }

    // 获取最小元素的值
    int getMinValue() {
        return data[indexes[0]];
    }
};
```

12.6.2 2. 二项堆和斐波那契堆

这些是更高级的堆结构，支持更多高效的操作，例如合并两个堆和减小键值。

12.7 堆的常见面试题和竞赛题

1. **合并 K 个有序数组/链表**：使用堆来高效地合并
2. **第 K 大/小元素**：使用堆来维护前 K 大/小的元素
3. **滑动窗口最大/最小值**：使用堆来跟踪窗口内的最大/最小值
4. **任务调度**：使用堆来选择优先级最高的任务
5. **丑数**：使用堆来生成按顺序的丑数
6. **K 近邻算法**：使用堆来找到最近的 K 个点
7. **流量控制**：在网络流量监控中维护高流量连接
8. **多路归并**：合并多个有序数据源时的选择策略

12.8 堆的实际应用

1. **优先级队列**：操作系统中的任务调度
2. **图算法**：Dijkstra、Prim 等算法的优化
3. **事件驱动模拟**：离散事件模拟
4. **数据流处理**：维护数据流中的统计信息，如中位数、Top-K 等

5. **定时任务管理**：操作系统中的定时器实现
6. **带宽管理**：网络路由器中的数据包优先级处理
7. **内存管理**：某些垃圾回收算法中的引用计数管理
8. **游戏开发**：寻路算法的开销优化、决策树剪枝

12.9 堆的性能分析与比较

12.9.1 时间复杂度

操作	数组实现	STL priority_queue	链表实现	斐波那契堆
建堆	O(n)	O(n)	O(n log n)	O(n)
插入	O(log n)	O(log n)	O(log n)	O(1)*
查找最值	O(1)	O(1)	O(1)	O(1)
删除最值	O(log n)	O(log n)	O(log n)	O(log n)*
合并	O(n)	O(n)	O(n)	O(1)
修改键值	O(log n)	不支持	O(log n)	O(1)*
删除任意元素	O(log n)	不支持	O(log n)	O(log n)

* 斐波那契堆的摊销时间复杂度

12.9.2 空间复杂度

- **数组实现**：O(n)，紧凑但需要连续内存
- **链表实现**：O(n)，但有额外指针开销
- **斐波那契堆**：O(n)，但常数因子较大

12.9.3 优缺点比较

数组实现 - 优点：内存紧凑，缓存友好，实现简单 - 缺点：合并操作不高效，需要连续内存

STL priority_queue - 优点：标准库实现，稳定可靠，使用方便 - 缺点：功能有限，不支持修改键值和迭代

二项堆/斐波那契堆 - 优点：某些操作的理论复杂度更优，支持高效合并 - 缺点：实现复杂，常数因子大，实际性能可能不如简单堆

12.10 技巧与总结

1. 对于需要频繁获取最大/最小值的场景，优先考虑使用堆
2. 对于 Top-K 问题，使用大小为 K 的堆可以高效解决
3. STL 中的 `priority_queue` 已经提供了完整的堆实现，无需自行编写
4. 使用自定义比较函数可以灵活地改变堆的排序方式
5. 索引堆是处理需要随机访问和修改堆元素的有力工具
6. 在图算法中，堆常用于优化最短路径和最小生成树算法
7. 当需要频繁修改和合并堆时，考虑使用更高级的堆结构
8. 在多线程环境中使用堆时，需要考虑并发安全问题

12.11 常见错误与陷阱

1. **忘记更新堆属性**：修改堆中元素后未执行上浮或下沉操作
2. **比较函数设置错误**：导致创建了最小堆而期望最大堆，或反之
3. **堆空检查**：在执行弹出操作前未检查堆是否为空
4. **自定义类型缺少比较运算符**：使用自定义类型时未提供比较方法
5. **误解 STL 容器行为**：`priority_queue` 默认是最大堆，而非最小堆
6. **重复索引问题**：在索引堆中误用了相同的索引值
7. **线程安全问题**：在并发环境中未对堆操作进行适当的同步

13 简单二叉树

13.1 基本概念

【定义】二叉树是一种树形结构，其中每个节点最多有两个子节点，通常称为左子节点和右子节点。

二叉树的主要类型包括：1. **满二叉树**：除叶节点外的所有节点都有两个子节点，且所有叶节点都在同一层 2. **完全二叉树**：除最后一层外，其他层的节点都是满的，且最后一层的节点都靠左排列 3. **平衡二叉树**：任意节点的左右子树高度差不超过 1 4. **二叉搜索树**：左子树上所有节点的值都小于根节点，右子树上所有节点的值都大于根节点

13.2 二叉树的表示

13.2.1 链式表示

```
struct TreeNode {
    int val;                // 节点值
    TreeNode *left;         // 左子节点指针
    TreeNode *right;        // 右子节点指针

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

【链式表示分析】- **优点**：结构灵活，易于实现动态操作（插入、删除节点）- **缺点**：额外的指针开销，内存分散可能导致缓存不友好 - **适用场景**：一般用于结构不规则的二叉树，如 BST、平衡树等 - **ACM 中常见用法**：大多数树题目都使用此表示方法，因为操作更为灵活

13.2.2 数组表示

对于完全二叉树，可以使用数组进行高效存储：- 根节点存储在索引 1（或 0，取决于实现）- 对于索引为 i 的节点，其左子节点的索引为 $2i$，右子节点的索引为 $2i+1$（如果索引从 1 开始）

```
const int MAXN = 1005;
int tree[MAXN];  // 使用数组存储完全二叉树
```

【数组表示分析】- **优点**：内存连续，访问迅速，无需存储指针 - **缺点**：对非完全二叉树会浪费空间 - **适用场景**：堆、线段树等完全二叉树结构 - **ACM 常见应用**：cpp // 索引从 1 开始的完全二叉树操作示例 inline int left(int i) { return i << 1; } // 左子节点：2*i inline int right(int i) { return (i << 1) | 1; } // 右子节点：2*i+1 inline int parent(int i) { return i >> 1; } // 父节点：i/2 - **易错点**：索引计算时的起始值（0 或 1）混淆会导致严重错误

13.3 二叉树的遍历

13.3.1 深度优先遍历

```
void preorder(TreeNode* root) {
    if (root == nullptr) return;

    cout << root->val << " ";  // 访问根节点
    preorder(root->left);      // 遍历左子树
    preorder(root->right);     // 遍历右子树
}

// 非递归实现
vector<int> preorderIterative(TreeNode* root) {
    vector<int> result;
    if (root == nullptr) return result;

    stack<TreeNode*> s;
    s.push(root);

    while (!s.empty()) {
```

```
        TreeNode* node = s.top();
        s.pop();

        result.push_back(node->val);  // 访问节点

        // 注意：先压入右子节点，再压入左子节点，这样出栈时会先处理左子节点
        if (node->right) s.push(node->right);
        if (node->left) s.push(node->left);
    }

    return result;
}
```

13.3.1.1 前序遍历（根-左-右）

【前序遍历分析】- **实现思路**：先访问当前节点，再递归访问左右子树 - **ACM 应用场景**：构建树的复制、序列化二叉树 - **递归 vs 迭代**：递归代码简洁但有栈溢出风险；迭代需要手动管理栈但更安全 - **时间复杂度**：$O(n)$，每个节点恰好被访问一次 - **空间复杂度**：递归 $O(h)$，迭代 $O(h)$，h 为树高，最坏情况 $O(n)$

```
void inorder(TreeNode* root) {
    if (root == nullptr) return;

    inorder(root->left);        // 遍历左子树
    cout << root->val << " ";   // 访问根节点
    inorder(root->right);       // 遍历右子树
}

// 非递归实现
vector<int> inorderIterative(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> s;
    TreeNode* curr = root;

    while (curr != nullptr || !s.empty()) {
        // 一直向左遍历，将所有节点入栈
        while (curr != nullptr) {
            s.push(curr);
            curr = curr->left;
        }

        // 弹出栈顶节点并访问
        curr = s.top();
        s.pop();
        result.push_back(curr->val);

        // 转向右子树
        curr = curr->right;
    }

    return result;
}
```

13.3.1.2 中序遍历（左-根-右）

【中序遍历分析】- **实现思路**：递归或迭代方式按"左-根-右"顺序访问节点 - **ACM 应用特点**：对 BST 进行中序遍历可以得到有序序列，常用于判断 BST 合法性 - **易错点**：迭代版本的双重循环逻辑较复杂，要注意内外循环的结束条件 - **优化思路**：可使用 Morris 遍历降低空间复杂度至 $O(1)$ - **测试用例设计**：- 单节点树：检验基本功能 - 链状树（只有左子树或只有右子树）：测试极端情况 - 满二叉树：测试平衡情况

```cpp
void postorder(TreeNode* root) {
    if (root == nullptr) return;

    postorder(root->left);      // 遍历左子树
    postorder(root->right);     // 遍历右子树
    cout << root->val << " ";  // 访问根节点
}

// 非递归实现
vector<int> postorderIterative(TreeNode* root) {
    vector<int> result;
    if (root == nullptr) return result;

    stack<TreeNode*> s1, s2;
    s1.push(root);

    // 第一次遍历，将节点按根-右-左的顺序压入 s2
    while (!s1.empty()) {
        TreeNode* node = s1.top();
        s1.pop();
        s2.push(node);

        if (node->left) s1.push(node->left);
        if (node->right) s1.push(node->right);
    }

    // 从 s2 中弹出节点，顺序即为左-右-根
    while (!s2.empty()) {
        result.push_back(s2.top()->val);
        s2.pop();
    }

    return result;
}
```

13.3.1.3 后序遍历（左-右-根）

【后序遍历分析】- **实现思路**：使用双栈法是非递归实现中最直观的方式 - **ACM 应用场景**：树的删除操作、表达式计算树求值 - **复杂度优化**：双栈法空间复杂度为 $O(n)$，但实现简单易懂 - **单栈实现提示**：需要记录上一个访问的节点，判断右子树是否已访问，实现更复杂 - **易错点**：直接修改前序遍历代码（调整左右子树压栈顺序，再反转结果）不足以实现后序遍历

13.3.2 广度优先遍历（层序遍历）

```cpp
vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (root == nullptr) return result;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();  // 当前层的节点数量
        vector<int> currentLevel;

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();
```

```
            currentLevel.push_back(node->val);

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        result.push_back(currentLevel);
    }

    return result;
}
```

【层序遍历分析】- **算法思路**：使用队列按层从上到下遍历节点 - **数据结构**：必须使用队列 (FIFO)，不能用栈 - **标准实现**：通过记录当前层的节点数量来分离不同层的节点 - **ACM 实战应用**：- 计算二叉树的宽度 - 寻找最近公共祖先 - 判断是否为完全二叉树 - **实现变种**：“‘cpp // 简化版层序遍历 (仅输出值) void simpleLevelOrder(TreeNode* root) { if (!root) return; queue<TreeNode*> q; q.push(root);

```
  while (!q.empty()) {
      TreeNode* node = q.front();
      q.pop();

      cout << node->val << " "; // 访问节点

      if (node->left) q.push(node->left);
      if (node->right) q.push(node->right);
  }
```

} “‘ - **时间复杂度**：O(n) - **空间复杂度**：O(w)，其中 w 是树的最大宽度，最坏情况 O(n/2) O(n)

13.4 二叉树的构造

13.4.1 从前序和中序遍历构造二叉树

```
/**
 * 根据前序遍历和中序遍历序列构造二叉树
 * 前序遍历的第一个元素是根节点，中序遍历中根节点左侧是左子树，右侧是右子树
 * @param preorder 前序遍历序列 [根节点, [左子树的前序遍历], [右子树的前序遍历]]
 * @param inorder 中序遍历序列 [[左子树的中序遍历], 根节点, [右子树的中序遍历]]
 * @return 构造的二叉树根节点
 */
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    return buildTreeHelper(preorder, 0, preorder.size() - 1,
                           inorder, 0, inorder.size() - 1);
}

/**
 * 递归辅助函数，用于构造二叉树的子树
 * @param preorder 前序遍历序列
 * @param preStart 当前子树在前序序列中的起始索引
 * @param preEnd 当前子树在前序序列中的结束索引
 * @param inorder 中序遍历序列
 * @param inStart 当前子树在中序序列中的起始索引
 * @param inEnd 当前子树在中序序列中的结束索引
 * @return 当前子树的根节点
 */
TreeNode* buildTreeHelper(vector<int>& preorder, int preStart, int preEnd,
                          vector<int>& inorder, int inStart, int inEnd) {
    // 递归终止条件：如果索引范围无效，返回空节点
```

```
    if (preStart > preEnd || inStart > inEnd) return nullptr;

    // 根节点是前序遍历的第一个节点
    int rootVal = preorder[preStart];
    TreeNode* root = new TreeNode(rootVal);

    // 在中序遍历中找到根节点的位置
    // 中序遍历中，根节点左侧的节点属于左子树，右侧的节点属于右子树
    int rootIndex = inStart;
    for (int i = inStart; i <= inEnd; i++) {
        if (inorder[i] == rootVal) {
            rootIndex = i;
            break;
        }
    }

    // 计算左子树的节点数量
    // 这个数量用于确定前序遍历中左子树的范围
    int leftSubtreeSize = rootIndex - inStart;

    // 递归构造左右子树
    // 左子树:
    // - 前序范围: [preStart+1, preStart+leftSubtreeSize]
    // - 中序范围: [inStart, rootIndex-1]
    root->left = buildTreeHelper(preorder, preStart + 1, preStart + leftSubtreeSize,
                                 inorder, inStart, rootIndex - 1);

    // 右子树:
    // - 前序范围: [preStart+leftSubtreeSize+1, preEnd]
    // - 中序范围: [rootIndex+1, inEnd]
    root->right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1, preEnd,
                                  inorder, rootIndex + 1, inEnd);

    return root;
}
```

【构造二叉树分析】- **算法原理**：1. 前序遍历的第一个元素是根节点 2. 在中序遍历中找到根节点位置，划分左右子树 3. 递归构建左右子树

- **ACM 常见题型**：根据各种遍历序列重建二叉树
  - 前序 + 中序 → 二叉树（经典题型）
  - 后序 + 中序 → 二叉树
  - 前序 + 后序 → 完全二叉树（解不唯一）
- **性能优化**：

```
// 使用哈希表优化根节点查找
unordered_map<int, int> buildIndexMap(vector<int>& inorder) {
    unordered_map<int, int> map;
    for (int i = 0; i < inorder.size(); i++) {
        map[inorder[i]] = i;
    }
    return map;
}

// 优化的构建函数
TreeNode* buildTreeOptimized(vector<int>& preorder, vector<int>& inorder) {
    unordered_map<int, int> indexMap = buildIndexMap(inorder);
```

```
    return buildTreeHelper(preorder, 0, preorder.size() - 1,
                           inorder, 0, inorder.size() - 1, indexMap);
}

TreeNode* buildTreeHelper(..., unordered_map<int, int>& indexMap) {
    // 使用 indexMap 直接查找根节点位置，避免 O(n) 的线性查找
    int rootIndex = indexMap[rootVal];
    // 其他代码不变...
}
```

- **复杂度分析**：
  - 原始实现：时间 $O(n^2)$，每层递归中查找 rootIndex 需要 $O(n)$
  - 哈希表优化：时间 $O(n)$，空间 $O(n)$
- **易错点**：
  - 索引计算错误导致子树划分不正确
  - 没有考虑到树中可能有重复值的情况
  - 递归基础情况处理不当可能导致无限递归

13.4.2 从后序和中序遍历构造二叉树

```
/**
 * 根据后序遍历和中序遍历序列构造二叉树
 * 后序遍历的最后一个元素是根节点，中序遍历中根节点左侧是左子树，右侧是右子树
 * @param postorder 后序遍历序列 [[左子树的后序遍历], [右子树的后序遍历], 根节点]
 * @param inorder 中序遍历序列 [[左子树的中序遍历], 根节点, [右子树的中序遍历]]
 * @return 构造的二叉树根节点
 */
TreeNode* buildFromPostAndIn(vector<int>& postorder, vector<int>& inorder) {
    // 构建中序遍历值到索引的映射，用于快速查找根节点在中序遍历中的位置
    unordered_map<int, int> indexMap;
    for (int i = 0; i < inorder.size(); i++) {
        indexMap[inorder[i]] = i;
    }

    return buildFromPostAndInHelper(postorder, 0, postorder.size() - 1,
                                    inorder, 0, inorder.size() - 1, indexMap);
}

/**
 * 递归辅助函数，用于构造二叉树的子树
 * @param postorder 后序遍历序列
 * @param postStart 当前子树在后序序列中的起始索引
 * @param postEnd 当前子树在后序序列中的结束索引
 * @param inorder 中序遍历序列
 * @param inStart 当前子树在中序序列中的起始索引
 * @param inEnd 当前子树在中序序列中的结束索引
 * @param indexMap 中序遍历值到索引的映射表
 * @return 当前子树的根节点
 */
TreeNode* buildFromPostAndInHelper(vector<int>& postorder, int postStart, int postEnd,
                                   vector<int>& inorder, int inStart, int inEnd,
                                   unordered_map<int, int>& indexMap) {
    // 递归终止条件：如果索引范围无效，返回空节点
    if (postStart > postEnd || inStart > inEnd) return nullptr;
```

```
    // 后序遍历的最后一个元素是当前子树的根节点
    int rootVal = postorder[postEnd];
    TreeNode* root = new TreeNode(rootVal);

    // 在中序遍历中找到根节点的位置 - 使用哈希表实现 O(1) 查找
    int rootIndex = indexMap[rootVal];

    // 计算左子树的节点数量
    // 这个数量用于确定后序遍历中左子树的范围
    int leftSubtreeSize = rootIndex - inStart;

    // 递归构造左右子树
    // 左子树:
    // - 后序范围: [postStart, postStart+leftSubtreeSize-1]
    // - 中序范围: [inStart, rootIndex-1]
    root->left = buildFromPostAndInHelper(postorder, postStart, postStart + leftSubtreeSize - 1,
                                          inorder, inStart, rootIndex - 1, indexMap);

    // 右子树:
    // - 后序范围: [postStart+leftSubtreeSize, postEnd-1]
    // - 中序范围: [rootIndex+1, inEnd]
    // 注意: 后序遍历中, 根节点在最后, 其前面是右子树, 再前面是左子树
    root->right = buildFromPostAndInHelper(postorder, postStart + leftSubtreeSize, postEnd - 1,
                                           inorder, rootIndex + 1, inEnd, indexMap);

    return root;
}
```

【后序 + 中序构造分析】- **算法原理**: 1. 后序遍历的最后一个元素是根节点 2. 在中序遍历中找到根节点位置，划分左右子树 3. 递归构建左右子树 - **实现难点**: 后序遍历中子树范围的计算较前序遍历更复杂 - 左子树: $\text{postStart} \sim (\text{postStart} + \text{leftSize} - 1)$ - 右子树: $(\text{postStart} + \text{leftSize}) \sim (\text{postEnd} - 1)$ - **优化策略**: - 使用哈希表存储中序遍历的值到索引的映射，将查找根节点位置的复杂度从 $O(n)$ 降低到 $O(1)$ - 直接传递子树大小而不是重新计算 - **ACM 应用场景**: 解析表达式树、处理语法分析树 - **易错点**: - 后序遍历的索引划分容易出错，注意右子树结束位置是 $\text{postEnd}-1$ - 处理空树或只有一个节点的特殊情况

13.4.3 从层序遍历构造完全二叉树

```
/**
 * 根据层序遍历序列构造完全二叉树
 * 层序遍历按照从上到下、从左到右的顺序访问节点
 * @param levelOrder 层序遍历序列 [根节点, 第二层节点..., 第 n 层节点...]
 * @return 构造的二叉树根节点
 */
TreeNode* buildFromLevelOrder(vector<int>& levelOrder) {
    // 空序列返回空树
    if (levelOrder.empty()) return nullptr;

    // 创建根节点
    TreeNode* root = new TreeNode(levelOrder[0]);
    queue<TreeNode*> q;  // 使用队列按层构建树节点
    q.push(root);

    int i = 1;  // 从第二个元素开始 (索引 1)
    // 按照完全二叉树的性质, 使用层序遍历构建树
    while (!q.empty() && i < levelOrder.size()) {
        // 获取当前要处理的父节点
        TreeNode* current = q.front();
        q.pop();
```

```
        // 处理左子节点 (2*i)
        if (i < levelOrder.size()) {
            if (levelOrder[i] != -1) {  // 假设-1 表示空节点
                // 创建左子节点并加入队列
                current->left = new TreeNode(levelOrder[i]);
                q.push(current->left);
            }
            i++;  // 移动到下一个元素

            // 处理右子节点 (2*i+1)
            if (i < levelOrder.size()) {
                if (levelOrder[i] != -1) {  // 假设-1 表示空节点
                    // 创建右子节点并加入队列
                    current->right = new TreeNode(levelOrder[i]);
                    q.push(current->right);
                }
                i++;  // 移动到下一个元素
            }
        }
    }

    return root;
}
```

【层序构造分析】- **算法原理**：利用队列按层构建树节点 1. 首先创建根节点（层序遍历的第一个元素）2. 使用队列跟踪当前层的节点 3. 为队列中的每个节点分配其左右子节点（如果存在）

- **数据结构**：
  - 队列用于按层次顺序处理节点
  - 层序遍历数组中的空节点通常使用特殊值（如-1 或 null）表示
- **时间复杂度**：$O(n)$，每个节点只处理一次
- **空间复杂度**：$O(w)$，其中 w 是树的最大宽度（最后一层的节点数），最坏情况下约为 $n/2$
- **实现技巧**：
  - 使用索引 i 跟踪层序序列中的当前位置
  - 对于每个父节点，先处理左子节点，再处理右子节点
  - 确保索引不超出数组范围
- **适用场景**：
  - 从数组表示转换为链式表示的树结构
  - 解析堆的数组表示并构建树形结构
  - 生成完全二叉树进行可视化或分析
- **易错点**：
  - 忘记检查空节点标记
  - 索引增量计算错误
  - 未正确处理数组边界条件 ### 从字符串表示构造二叉树

处理括号表示法的二叉树字符串，如："$1(2)(3(4)(5))$"

```
TreeNode* buildFromBracketNotation(string s) {
    if (s.empty()) return nullptr;

    int i = 0;
    return parseNode(s, i);
}

TreeNode* parseNode(const string& s, int& i) {
    // 解析节点值
    string num;
    while (i < s.size() && s[i] >= '0' && s[i] <= '9') {
        num += s[i++];
    }

    if (num.empty()) return nullptr;

    TreeNode* node = new TreeNode(stoi(num));

    // 解析左子树
    if (i < s.size() && s[i] == '(') {
        i++; // 跳过左括号
        node->left = parseNode(s, i);
        i++; // 跳过右括号
    }

    // 解析右子树
    if (i < s.size() && s[i] == '(') {
        i++; // 跳过左括号
        node->right = parseNode(s, i);
        i++; // 跳过右括号
    }

    return node;
}
```

【字符串构造分析】- **算法原理**：递归解析括号表示的树结构 - **实现技巧**：使用引用传递索引位置，跟踪当前解析位置 - **ACM 应用**：解析输入的树形结构字符串表示 - **注意事项**：需处理多位数字、空节点和格式错误等边界情况

13.4.4 构造特殊的二叉树

13.4.4.1 构造平衡二叉搜索树 从排序数组构造平衡 BST：

```
TreeNode* sortedArrayToBST(vector<int>& nums) {
    return buildBST(nums, 0, nums.size() - 1);
}

TreeNode* buildBST(vector<int>& nums, int left, int right) {
    if (left > right) return nullptr;

    // 总是选择中间位置的元素作为根节点
    int mid = left + (right - left) / 2;
    TreeNode* root = new TreeNode(nums[mid]);

    // 递归构造左右子树
    root->left = buildBST(nums, left, mid - 1);
    root->right = buildBST(nums, mid + 1, right);
```

```
    return root;
}
```

【构造平衡 BST 分析】- **核心思想**：始终选择数组中间元素作为子树根节点 - **时间复杂度**：$O(n)$ - **特点**：构造出的树高度最小，约为 $\log(n)$ - **应用场景**：需要快速构建平衡搜索树的场景

```
// 线段树节点
struct SegmentTreeNode {
    int start, end;  // 区间范围
    int sum;         // 区间和（可根据需要改为其他聚合值）
    SegmentTreeNode *left, *right;

    SegmentTreeNode(int s, int e) : start(s), end(e), sum(0), left(nullptr), right(nullptr) {}
};

// 从数组构建线段树
SegmentTreeNode* buildSegmentTree(vector<int>& nums, int start, int end) {
    if (start > end) return nullptr;

    SegmentTreeNode* root = new SegmentTreeNode(start, end);

    if (start == end) {
        // 叶节点，代表单个元素
        root->sum = nums[start];
    } else {
        int mid = start + (end - start) / 2;
        root->left = buildSegmentTree(nums, start, mid);
        root->right = buildSegmentTree(nums, mid + 1, end);

        // 合并子节点信息
        root->sum = (root->left ? root->left->sum : 0) +
                    (root->right ? root->right->sum : 0);
    }

    return root;
}
```

13.4.4.2 构造线段树

【线段树构造分析】- **应用场景**：区间查询和修改问题 - **构造思路**：自顶向下分割区间，自底向上聚合信息 - **时间复杂度**：$O(n)$ - **ACM 实战提示**：线段树是解决区间问题的强大工具，掌握其构造和操作是算法竞赛的基本要求

13.5 二叉树的应用

13.5.1 二叉搜索树 (BST)

二叉搜索树是特殊的二叉树，它满足以下性质：- 左子树上所有节点的值都小于根节点的值 - 右子树上所有节点的值都大于根节点的值 - 左右子树也都是二叉搜索树

```
TreeNode* search(TreeNode* root, int key) {
    if (root == nullptr || root->val == key) return root;

    if (key < root->val) {
        return search(root->left, key);  // 在左子树中查找
    } else {
        return search(root->right, key); // 在右子树中查找
    }
```

```
}

// 非递归实现
TreeNode* searchIterative(TreeNode* root, int key) {
    TreeNode* curr = root;

    while (curr != nullptr && curr->val != key) {
        if (key < curr->val) {
            curr = curr->left;
        } else {
            curr = curr->right;
        }
    }

    return curr;  // 如果找不到，返回 nullptr
}
```

13.5.1.1 BST 的查找操作

【BST 查找分析】- **算法思路**：利用 BST 的有序性质，通过比较节点值与目标值大小决定搜索方向 - **时间复杂度**：平均 $O(\log n)$，最坏 $O(n)$（当树退化为链表时）- **递归与迭代**：迭代版本避免了函数调用开销，更适合 ACM 比赛 - **ACM 实战应用**：实现各种集合、映射操作 - **优化建议**：对于不平衡的 BST，考虑使用平衡树如 AVL 或红黑树

```
TreeNode* insert(TreeNode* root, int key) {
    if (root == nullptr) return new TreeNode(key);

    if (key < root->val) {
        root->left = insert(root->left, key);
    } else if (key > root->val) {
        root->right = insert(root->right, key);
    }

    return root;  // 返回更新后的树的根节点
}

// 非递归实现
TreeNode* insertIterative(TreeNode* root, int key) {
    TreeNode* newNode = new TreeNode(key);

    if (root == nullptr) return newNode;

    TreeNode* curr = root;
    TreeNode* parent = nullptr;

    while (curr != nullptr) {
        parent = curr;

        if (key < curr->val) {
            curr = curr->left;
        } else if (key > curr->val) {
            curr = curr->right;
        } else {
            // 键已存在，不进行插入
            delete newNode;
            return root;
        }
    }
```

```
    // 确定新节点是父节点的左子节点还是右子节点
    if (key < parent->val) {
        parent->left = newNode;
    } else {
        parent->right = newNode;
    }

    return root;
}
```

13.5.1.2 BST 的插入操作

【BST 插入分析】- **算法原理**：通过比较寻找合适的插入位置 - **关键点**：BST 不允许重复值（标准定义下），需要特殊处理 - **时间复杂度**：平均 $O(\log n)$，最坏 $O(n)$ - **ACM 应用**：动态维护有序数据集合，multiset 实现 - **易错点**：- 插入操作可能改变根节点，需要正确处理返回值 - 不正确处理树中已存在的重复值会导致内存泄漏 - 迭代版本必须记录 parent 节点才能连接新节点

```
TreeNode* deleteNode(TreeNode* root, int key) {
    if (root == nullptr) return nullptr;

    // 寻找要删除的节点
    if (key < root->val) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->val) {
        root->right = deleteNode(root->right, key);
    } else {
        // 找到要删除的节点

        // 情况 1: 叶节点（无子节点）
        if (root->left == nullptr && root->right == nullptr) {
            delete root;
            return nullptr;
        }
        // 情况 2: 只有一个子节点
        else if (root->left == nullptr) {
            TreeNode* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            TreeNode* temp = root->left;
            delete root;
            return temp;
        }
        // 情况 3: 有两个子节点
        else {
            // 找到右子树中的最小节点（或左子树中的最大节点）
            TreeNode* temp = findMin(root->right);

            // 用该节点的值替换当前节点的值
            root->val = temp->val;

            // 删除右子树中的最小节点
            root->right = deleteNode(root->right, temp->val);
        }
    }
```

```
    return root;
}

TreeNode* findMin(TreeNode* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}
```

13.5.1.3 BST 的删除操作

【BST 删除分析】- **算法思路**：分三种情况处理删除操作 1. 叶节点：直接删除 2. 有一个子节点：用子节点替换 3. 有两个子节点：用后继节点（或前驱节点）替换，再删除后继

- **复杂度分析**：
  - 时间复杂度：平均 $O(\log n)$，最坏 $O(n)$
  - 空间复杂度：递归实现 $O(h)$，h 为树高
- **ACM 实战技巧**：
  - 题目中如果只需要考虑插入和查找操作，可省略删除操作，简化代码
  - 删除是 BST 中最复杂的操作，实现时考虑用标记删除（懒删除）简化
- **常见错误**：
  - 忘记释放被删除节点的内存
  - 处理有两个子节点的情况时，没有正确递归删除替换节点
  - 对根节点的改变处理不当

13.6 二叉树的常见操作

13.6.1 计算二叉树的高度

```
int height(TreeNode* root) {
    if (root == nullptr) return 0;

    int leftHeight = height(root->left);
    int rightHeight = height(root->right);

    return max(leftHeight, rightHeight) + 1;
}
```

13.6.2 检查二叉树是否平衡

```
bool isBalanced(TreeNode* root) {
    return checkHeight(root) != -1;
}

// 返回树的高度，如果不平衡则返回-1
int checkHeight(TreeNode* root) {
    if (root == nullptr) return 0;

    int leftHeight = checkHeight(root->left);
    if (leftHeight == -1) return -1;

    int rightHeight = checkHeight(root->right);
    if (rightHeight == -1) return -1;

    // 检查高度差是否超过 1
```

```
    if (abs(leftHeight - rightHeight) > 1) return -1;

    return max(leftHeight, rightHeight) + 1;
}
```

【平衡检查分析】- **算法原理**：自底向上检查每个节点的左右子树高度差 - **优化思路**：使用一个函数同时计算高度和判断平衡性，避免重复计算 - **时间复杂度**：$O(n)$，每个节点只访问一次 - **对比分析**：“‘cpp // 朴素解法（低效） - $O(n^2)$ bool isBalancedNaive(TreeNode* root) { if (root == nullptr) return true;

```
  int leftHeight = height(root->left);
  int rightHeight = height(root->right);

  return abs(leftHeight - rightHeight) <= 1 &&
         isBalancedNaive(root->left) &&
         isBalancedNaive(root->right);
```

} “‘ - **ACM 应用**：AVL 树相关问题，验证树的合法性

13.6.3 二叉树的序列化与反序列化

```
// 序列化二叉树
string serialize(TreeNode* root) {
    if (root == nullptr) return "X,";  // 使用 X 表示空节点

    string result = to_string(root->val) + ",";
    result += serialize(root->left);
    result += serialize(root->right);

    return result;
}

// 反序列化二叉树
TreeNode* deserialize(string data) {
    queue<string> nodes;
    string val;

    // 将序列化字符串分割为 token
    for (char c : data) {
        if (c == ',') {
            nodes.push(val);
            val.clear();
        } else {
            val.push_back(c);
        }
    }

    return deserializeHelper(nodes);
}

TreeNode* deserializeHelper(queue<string>& nodes) {
    string val = nodes.front();
    nodes.pop();

    if (val == "X") return nullptr;  // 空节点

    TreeNode* root = new TreeNode(stoi(val));
    root->left = deserializeHelper(nodes);
    root->right = deserializeHelper(nodes);
```

```
    return root;
}
```

【序列化分析】- **实现思路**：使用前序遍历序列化，空节点用特殊符号表示 - **ACM 应用场景**：树的持久化、树的传输与重建 - **多种序列化方案对比**：1. 前序遍历序列化（常用）：实现简单，易于理解 2. 层序遍历序列化：适合表示广度优先遍历序列 3. 括号表示法：如"1(2)(3(4)(5))"，人类可读性好 - **注意事项**：- 确保特殊字符（如 X）不会与节点值冲突 - 处理节点值可能包含的特殊字符 - 序列化和反序列化算法必须配对使用 - **优化提示**：对于大型树，考虑使用二进制序列化以节省空间

13.7 二叉树的高级应用

13.7.1 线索二叉树

线索二叉树是一种优化的二叉树，它利用叶节点的空指针存储前驱和后继信息，以便更高效地进行树的遍历。

```
// 线索二叉树节点定义
struct ThreadedNode {
    int val;
    ThreadedNode *left, *right;
    bool leftThread;   // 左指针是否为线索
    bool rightThread;  // 右指针是否为线索

    ThreadedNode(int x) : val(x), left(nullptr), right(nullptr),
                          leftThread(false), rightThread(false) {}
};

// 中序线索化二叉树
void inorderThreading(ThreadedNode* root) {
    ThreadedNode* prev = nullptr; // 记录前一个访问的节点
    inorderThreadingHelper(root, prev);
}

void inorderThreadingHelper(ThreadedNode* node, ThreadedNode*& prev) {
    if (node == nullptr) return;

    // 线索化左子树
    inorderThreadingHelper(node->left, prev);

    // 处理当前节点
    if (node->left == nullptr) {
        node->leftThread = true;
        node->left = prev; // 左指针指向前驱
    }

    // 处理前一个节点的右线索
    if (prev != nullptr && prev->right == nullptr) {
        prev->rightThread = true;
        prev->right = node; // 右指针指向后继
    }

    prev = node;

    // 线索化右子树
    inorderThreadingHelper(node->right, prev);
}
```

【线索二叉树分析】- **基本原理**：利用空指针存储前驱后继，加速遍历 - **优点**：无需栈或递归即可进行遍历，空间复杂度 O(1) - **缺点**：修改树结构更复杂，需要维护线索信息 - **适用场景**：需要频繁遍历但较少修改的二叉树 - **ACM 竞赛应用**：了解即可，实际竞赛中较少直接使用

13.7.2 二叉树的 Morris 遍历

Morris 遍历是一种不使用栈和递归，空间复杂度为 $O(1)$ 的二叉树遍历算法。

```
vector<int> morrisInorder(TreeNode* root) {
    vector<int> result;
    TreeNode* curr = root;

    while (curr != nullptr) {
        if (curr->left == nullptr) {
            // 如果没有左子树，访问当前节点并转向右子树
            result.push_back(curr->val);
            curr = curr->right;
        } else {
            // 找到当前节点在中序遍历下的前驱节点
            TreeNode* predecessor = curr->left;
            while (predecessor->right != nullptr && predecessor->right != curr) {
                predecessor = predecessor->right;
            }

            if (predecessor->right == nullptr) {
                // 建立线索（前驱节点的右指针指向当前节点）
                predecessor->right = curr;
                curr = curr->left;
            } else {
                // 已经访问过左子树，恢复树结构
                predecessor->right = nullptr;
                result.push_back(curr->val);
                curr = curr->right;
            }
        }
    }

    return result;
}
```

【Morris 遍历分析】- **算法原理**：动态建立和拆除线索，无需额外空间 - **关键步骤**：1. 找到当前节点的前驱 2. 建立临时线索 3. 遍历完左子树后拆除线索 - **复杂度分析**：- 时间复杂度：虽然有嵌套循环，但总体仍为 $O(n)$ - 空间复杂度：$O(1)$，不使用额外空间 - **应用场景**：内存受限环境，如嵌入式系统 - **ACM 提示**：这是一种高级技巧，了解原理即可，实际比赛中根据题目选择合适的遍历方式

13.8 习题推荐

1. LeetCode 94: 二叉树的中序遍历
2. LeetCode 102: 二叉树的层序遍历
3. LeetCode 105: 从前序与中序遍历序列构造二叉树
4. LeetCode 236: 二叉树的最近公共祖先
5. LeetCode 662: 二叉树最大宽度
6. POJ 1330: 树的最近公共祖先
7. Codeforces 380C: Sereja and Brackets（使用二叉树实现的线段树）
8. SPOJ QTREE: 树上路径查询系列问题

13.9 ACM 竞赛实用技巧

1. **二叉树问题解题框架**：

    - 确定遍历方式（前序、中序、后序、层序）
    - 明确是自顶向下还是自底向上解决问题
    - 选择合适的递归/迭代方式

2. **输入优化**：

```
// 读入二叉树优化
ios::sync_with_stdio(false);
cin.tie(nullptr);

// 构建二叉树的常用方法
TreeNode* buildTreeFromArray(const vector<int>& arr, int index = 0) {
    if (index >= arr.size() || arr[index] == -1) return nullptr; // -1 表示空节点

    TreeNode* root = new TreeNode(arr[index]);
    root->left = buildTreeFromArray(arr, 2*index + 1);
    root->right = buildTreeFromArray(arr, 2*index + 2);
    return root;
}
```

3. **内存管理**：

```
// 清理二叉树内存
void freeTree(TreeNode* root) {
    if (root == nullptr) return;
    freeTree(root->left);
    freeTree(root->right);
    delete root;
}

// 比赛中更简单的方法：使用静态数组预分配节点
const int MAXN = 1e5 + 5;
TreeNode nodes[MAXN];
int nodeCount = 0;

TreeNode* newNode(int val) {
    nodes[nodeCount].val = val;
    nodes[nodeCount].left = nodes[nodeCount].right = nullptr;
    return &nodes[nodeCount++];
}
```

4. **调试技巧**：

```
// 打印二叉树结构（调试用）
void printTree(TreeNode* root, string prefix = "", bool isLeft = true) {
    if (root == nullptr) return;

    cout << prefix;
    cout << (isLeft ? "   " : "   ");
    cout << root->val << endl;

    printTree(root->left, prefix + (isLeft ? "    " : "    "), true);
    printTree(root->right, prefix + (isLeft ? "    " : "    "), false);
}
```

5. **记忆化搜索**：对于需要多次重复计算的树形结构问题，使用哈希表缓存中间结果

13.10 复杂度分析

- **平衡二叉搜索树**：
  - 查找、插入、删除操作的平均时间复杂度：$O(\log n)$

- 最坏情况（如退化为链表）时间复杂度：$O(n)$
- 存储空间：$O(n)$

- **树遍历算法**：
  - 递归/迭代实现的时间复杂度：$O(n)$
  - 递归/栈实现的空间复杂度：$O(h)$，h 为树高，最坏 $O(n)$
  - Morris 遍历空间复杂度：$O(1)$

- **性能优化策略**：
  1. 对于高度不平衡的树，考虑重新平衡或使用平衡树结构
  2. 对于频繁查询但很少修改的树，考虑缓存结果
  3. 递归层数太深时，转为迭代实现避免栈溢出
  4. 对于大规模数据，使用节点池管理内存，减少动态分配开销

13.10.1 前序和后序遍历确定二叉树数量

当已知二叉树的前序遍历和后序遍历时，可能存在多种不同的二叉树结构。这是因为前序和后序遍历无法唯一确定一棵二叉树（除非是满二叉树）。

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MOD = 1000000007;

// 快速幂：计算 a^e % MOD
long long mod_pow(long long a, long long e) {
    long long r = 1;
    while (e) {
        if (e & 1) r = r * a % MOD;
        a = a * a % MOD;
        e >>= 1;
    }
    return r;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    string line;
    vector<int> pre, post;

    // 读入前序遍历
    getline(cin, line);
    {
        istringstream iss(line);
        int x;
        while (iss >> x) pre.push_back(x);
    }
    // 读入后序遍历
    getline(cin, line);
    {
        istringstream iss(line);
        int x;
        while (iss >> x) post.push_back(x);
    }

    int n = pre.size();
    stack<int> st;
    st.push(pre[0]);
```

```
    int j = 0, m = 0;
    for (int i = 1; i < n; i++) {
        // 如果栈顶已经遍历完，弹出
        while (!st.empty() && st.top() == post[j]) {
            st.pop();
            j++;
        }
        // 栈非空表示当前节点是单子树节点，增加二义性
        if (!st.empty()) m++;
        st.push(pre[i]);
    }

    // 答案 = 2^m % MOD
    cout << mod_pow(2, m) << "\n";
    return 0;
}
```

14 字典树

【字典树】，又称前缀树 (Prefix Tree) 或 Trie 树，是一种专门用于处理字符串检索的树形数据结构。它能够高效地存储和查找字符串集合，特别适合用于前缀匹配和单词查询场景。

14.1 基本原理

字典树的核心思想是利用字符串的公共前缀来减少查询时间，通过树形结构将字符串的公共前缀合并在一起。每个节点代表一个字符，从根节点到某一节点的路径上经过的字符连接起来，就是该节点对应的字符串。

14.1.1 字典树的特性

1. **根节点不包含字符**，除根节点外每个节点只包含一个字符
2. **从根节点到特定节点的路径上的字符连接**即为该节点对应的字符串
3. **每个节点的所有子节点包含的字符都不相同**
4. **通常在词尾节点标记单词结束**

14.2 数据结构定义

一个基本的 Trie 节点通常包含以下信息：

```
struct TrieNode {
    bool isEndOfWord;  // 标记该节点是否是一个单词的结束
    TrieNode* children[26];  // 假设只包含小写英文字母，每个节点最多有 26 个子节点

    // 构造函数
    TrieNode() {
        isEndOfWord = false;
        for (int i = 0; i < 26; i++) {
            children[i] = nullptr;
        }
    }
};
```

14.3 基本操作

14.3.1 1. 插入操作

向 Trie 中插入一个单词：

```
// 插入单词
void insert(TrieNode* root, const string& word) {
    TrieNode* node = root;
    for (char c : word) {
        int index = c - 'a';  // 转换为 0-25 的索引
        if (!node->children[index]) {
            node->children[index] = new TrieNode();  // 如果子节点不存在，创建新节点
        }
        node = node->children[index];  // 移动到子节点
    }
    node->isEndOfWord = true;  // 标记单词结束
}
```

14.3.2 2. 查找操作

在 Trie 中查找一个完整的单词：

```
// 查找单词
bool search(TrieNode* root, const string& word) {
    TrieNode* node = root;
    for (char c : word) {
        int index = c - 'a';
        if (!node->children[index]) {
            return false;  // 字符不存在，查找失败
        }
        node = node->children[index];  // 移动到子节点
    }
    return node->isEndOfWord;  // 只有节点标记为单词结束才返回 true
}
```

14.3.3 3. 前缀查找操作

检查 Trie 中是否存在以给定前缀开始的单词：

```
// 前缀查找
bool startsWith(TrieNode* root, const string& prefix) {
    TrieNode* node = root;
    for (char c : prefix) {
        int index = c - 'a';
        if (!node->children[index]) {
            return false;  // 字符不存在，查找失败
        }
        node = node->children[index];  // 移动到子节点
    }
    return true;  // 前缀存在
}
```

14.3.4 4. 删除操作

从 Trie 中删除一个单词（相对复杂）：

```
// 辅助函数 - 判断节点是否可以删除
bool isEmpty(TrieNode* node) {
    for (int i = 0; i < 26; i++) {
        if (node->children[i]) return false;
    }
    return true;
```

```
}

// 递归删除
bool remove(TrieNode* &node, const string& word, int depth = 0) {
    // 如果树为空
    if (!node) return false;

    // 如果已经到达单词末尾
    if (depth == word.length()) {
        // 如果此节点不是单词结尾，删除失败
        if (!node->isEndOfWord) return false;

        // 取消标记为单词结尾
        node->isEndOfWord = false;

        // 如果此节点没有子节点，可以删除
        if (isEmpty(node)) {
            delete node;
            node = nullptr;
            return true;
        }
        return false;
    }

    // 递归删除
    int index = word[depth] - 'a';
    bool shouldDeleteCurrentNode = remove(node->children[index], word, depth + 1);

    // 如果子节点被删除且当前节点不是单词结尾且没有其他子节点
    if (shouldDeleteCurrentNode && !node->isEndOfWord && isEmpty(node)) {
        delete node;
        node = nullptr;
        return true;
    }

    return false;
}
```

14.4 完整实现

以下是一个完整的 Trie 类实现：

```
#include <iostream>
#include <string>
using namespace std;

class Trie {
private:
    struct TrieNode {
        bool isEndOfWord;
        TrieNode* children[26]; // 假设只处理小写英文字母

        TrieNode() {
            isEndOfWord = false;
            for (int i = 0; i < 26; i++) {
                children[i] = nullptr;
            }
```

```
        }
    };

    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }

    ~Trie() {
        // 递归删除所有节点
        clear(root);
    }

    // 释放内存的辅助函数
    void clear(TrieNode* node) {
        if (!node) return;
        for (int i = 0; i < 26; i++) {
            if (node->children[i]) {
                clear(node->children[i]);
            }
        }
        delete node;
    }

    // 插入单词
    void insert(const string& word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new TrieNode();
            }
            node = node->children[index];
        }
        node->isEndOfWord = true;
    }

    // 查找单词
    bool search(const string& word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                return false;
            }
            node = node->children[index];
        }
        return node->isEndOfWord;
    }

    // 前缀查找
    bool startsWith(const string& prefix) {
        TrieNode* node = root;
        for (char c : prefix) {
            int index = c - 'a';
```

```
            if (!node->children[index]) {
                return false;
            }
            node = node->children[index];
        }
        return true;
    }

    // 删除单词
    void remove(const string& word) {
        removeHelper(root, word, 0);
    }

private:
    bool removeHelper(TrieNode* node, const string& word, int depth) {
        if (!node) return false;

        // 到达单词末尾
        if (depth == word.length()) {
            if (!node->isEndOfWord) return false;

            node->isEndOfWord = false;

            // 检查是否可以删除此节点
            return isEmpty(node);
        }

        int index = word[depth] - 'a';
        if (removeHelper(node->children[index], word, depth + 1)) {
            delete node->children[index];
            node->children[index] = nullptr;

            // 如果没有其他子节点且不是单词结尾，可以删除
            return !node->isEndOfWord && isEmpty(node);
        }

        return false;
    }

    // 检查节点是否没有子节点
    bool isEmpty(TrieNode* node) {
        for (int i = 0; i < 26; i++) {
            if (node->children[i]) return false;
        }
        return true;
    }
};

int main() {
    Trie trie;

    // 插入单词
    trie.insert("apple");
    trie.insert("app");
    trie.insert("application");

    // 查找单词
```

```
    cout << " 搜索 'apple': " << (trie.search("apple") ? " 找到" : " 未找到") << endl;
    cout << " 搜索 'app': " << (trie.search("app") ? " 找到" : " 未找到") << endl;
    cout << " 搜索 'apricot': " << (trie.search("apricot") ? " 找到" : " 未找到") << endl;

    // 前缀查找
    cout << " 前缀 'app': " << (trie.startsWith("app") ? " 存在" : " 不存在") << endl;
    cout << " 前缀 'apr': " << (trie.startsWith("apr") ? " 存在" : " 不存在") << endl;

    // 删除单词
    trie.remove("apple");
    cout << " 删除后搜索 'apple': " << (trie.search("apple") ? " 找到" : " 未找到") << endl;
    cout << " 删除后搜索 'app': " << (trie.search("app") ? " 找到" : " 未找到") << endl;

    return 0;
}
```

14.5 字典树的优化与变种

14.5.1 1. 压缩前缀树 (Compressed Trie)

合并只有一个子节点的节点，以节省空间：

```
struct CompressedTrieNode {
    bool isEndOfWord;
    string prefix;  // 存储从父节点到当前节点的字符串片段
    unordered_map<char, CompressedTrieNode*> children;

    CompressedTrieNode() : isEndOfWord(false) {}
};
```

14.5.2 2. 三分搜索树 (Ternary Search Tree)

每个节点有三个子节点，减少空间浪费：

```
struct TSTreeNode {
    char data;
    bool isEndOfWord;
    TSTreeNode *left, *mid, *right;

    TSTreeNode(char c) : data(c), isEndOfWord(false), left(nullptr), mid(nullptr), right(nullptr) {}
};
```

14.5.3 3. 双数组 Trie (Double-Array Trie)

使用两个数组来表示 Trie，空间效率更高，适合处理大规模字典：

- base 数组：存储当前状态的基础值
- check 数组：存储当前状态的有效性检查

14.6 时间和空间复杂度

- **插入操作**：$O(m)$，其中 m 是单词长度
- **查找操作**：$O(m)$，其中 m 是单词长度
- **前缀查找**：$O(m)$，其中 m 是前缀长度
- **删除操作**：$O(m)$，其中 m 是单词长度
- **空间复杂度**：

- 最坏情况：O(ALPHABET_SIZE * m * n)，其中 n 是单词数量，m 是单词的平均长度，ALPHABET_SIZE 是字符集大小
- 实际情况往往好很多，因为有公共前缀

14.7 字典树的应用场景

1. **自动补全/自动建议**：根据用户输入的前缀，快速找到所有可能的完整单词
2. **拼写检查**：检查单词是否在字典中
3. **最长公共前缀**：查找一组字符串的最长公共前缀
4. **字符串匹配**：在文本中查找多个模式串
5. **IP 路由表查找**：使用前缀匹配来查找 IP 地址对应的路由
6. **单词游戏**：如 Boggle，快速检查是否有单词以特定字母序列开始

14.8 典型例题

14.8.1 例题 1：实现一个字典

问题：设计一个数据结构，支持以下操作：1. 添加一个单词 2. 判断一个单词是否在字典中 3. 判断是否有以特定前缀开始的单词

解决方案：使用上面实现的 Trie 类

14.8.2 例题 2：单词搜索 II

问题：给定一个 m×n 的字母矩阵和一个单词列表，找出所有在矩阵中出现的单词。单词可以由相邻单元格（水平或垂直）中的字母构成，但不能重复使用同一单元格。

解决方案：使用 Trie 进行 DFS 搜索

```
vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
    // 构建字典树
    Trie trie;
    for (const auto& word : words) {
        trie.insert(word);
    }

    int m = board.size();
    int n = board[0].size();
    vector<string> result;
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    string current;

    // 从每个单元格开始 DFS
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            dfs(board, visited, trie, i, j, current, result);
        }
    }

    return result;
}

// DFS 搜索
void dfs(vector<vector<char>>& board, vector<vector<bool>>& visited, Trie& trie,
         int i, int j, string& current, vector<string>& result) {
    // 超出边界或已访问
    if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size() || visited[i][j]) {
        return;
    }

    // 添加当前字符
    current.push_back(board[i][j]);
```

```
    // 检查前缀是否存在
    if (!trie.startsWith(current)) {
        current.pop_back();
        return;
    }

    // 标记为已访问
    visited[i][j] = true;

    // 如果当前是一个完整单词，加入结果
    if (trie.search(current) && find(result.begin(), result.end(), current) == result.end()) {
        result.push_back(current);
    }

    // 探索四个方向
    dfs(board, visited, trie, i + 1, j, current, result);
    dfs(board, visited, trie, i - 1, j, current, result);
    dfs(board, visited, trie, i, j + 1, current, result);
    dfs(board, visited, trie, i, j - 1, current, result);

    // 回溯
    visited[i][j] = false;
    current.pop_back();
}
```

14.8.3 例题 3：最长公共前缀

问题：找出一组字符串的最长公共前缀。

解决方案：将所有字符串插入 Trie，然后从根开始遍历，直到遇到分叉或结束。

```
string longestCommonPrefix(vector<string>& strs) {
    if (strs.empty()) return "";

    // 构建前缀树
    Trie trie;
    for (const auto& str : strs) {
        trie.insert(str);
    }

    // 获取第一个字符串
    string first = strs[0];
    string prefix;

    TrieNode* node = trie.getRoot();
    for (char c : first) {
        int index = c - 'a';
        int count = 0;

        // 检查是否只有一个子节点
        for (int i = 0; i < 26; i++) {
            if (node->children[i]) {
                count++;
            }
        }

        // 如果有多个分支或已到达单词结尾，则结束
        if (count > 1 || node->isEndOfWord) {
```

```
            break;
        }

        node = node->children[index];
        prefix.push_back(c);
    }

    return prefix;
}
```

14.9 注意事项

1. **内存管理**：字典树可能消耗大量内存，尤其是当字符集较大时
2. **字符集考虑**：根据实际应用选择合适的字符集表示
3. **空间优化**：考虑使用压缩前缀树或双数组 Trie 来优化空间
4. **内存泄漏**：在删除字典树时，确保正确释放所有节点的内存

15 字符串算法

15.1 字符串算法的基本操作

1. **字符串匹配**：在文本中查找特定的模式串。
   - 朴素算法：$O(n*m)$ 时间复杂度
   - KMP 算法：$O(n+m)$ 时间复杂度
   - Rabin-Karp 算法：平均 $O(n+m)$，最坏 $O(n*m)$
   - Boyer-Moore 算法：平均情况下比 KMP 更快
2. **字符串处理**：
   - 分割与合并
   - 替换与删除
   - 大小写转换
   - 子串提取
3. **字符串存储**：
   - 字典树 (Trie)
   - 后缀树与后缀数组
   - 哈希表

15.2 常用字符串算法

15.2.1 1. KMP 算法

Knuth-Morris-Pratt 算法是一种高效的字符串匹配算法，通过预处理模式串，避免了不必要的比较。

核心思想：利用已经匹配过的信息，避免回溯文本串的指针。

时间复杂度：$O(n+m)$，其中 n 和 m 分别是文本串和模式串的长度。

主要步骤：1. 预处理模式串，计算 next 数组 2. 使用 next 数组指导匹配过程

```
// KMP 算法实现
void computeLPS(string pat, int* lps) {
    int m = pat.length();
    int len = 0;  // 前缀长度

    lps[0] = 0;  // lps[0] 总是 0

    int i = 1;
    while (i < m) {
        if (pat[i] == pat[len]) {
```

```
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

vector<int> KMP(string txt, string pat) {
    int n = txt.length();
    int m = pat.length();
    vector<int> result;  // 存储所有匹配位置

    // 创建 lps 数组存储最长匹配前缀后缀值
    int lps[m];
    computeLPS(pat, lps);

    int i = 0;  // txt 的索引
    int j = 0;  // pat 的索引

    while (i < n) {
        // 当前字符匹配
        if (pat[j] == txt[i]) {
            i++;
            j++;
        }

        // 找到完整匹配
        if (j == m) {
            result.push_back(i - j);  // 记录匹配位置
            j = lps[j - 1];  // 寻找下一个匹配
        }
        // 不匹配的情况
        else if (i < n && pat[j] != txt[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }

    return result;
}
```

15.2.2 2. 字典树 (Trie)

字典树是一种树形数据结构，用于高效地存储和检索字符串集合。

特点：- 根节点不包含字符 - 每个节点包含多个子节点，对应不同的字符 - 从根节点到某一节点的路径上的字符连接起来形成一个字符串

应用场景：- 前缀匹配 - 自动补全 - 字符串集合的快速查找

```cpp
// 字典树实现
struct TrieNode {
    bool isEndOfWord;  // 标记该节点是否为某个单词的结尾
    TrieNode* children[26];  // 假设只包含小写英文字母

    TrieNode() {
        isEndOfWord = false;
        for (int i = 0; i < 26; i++) {
            children[i] = nullptr;
        }
    }
};

class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }

    // 插入单词
    void insert(string word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new TrieNode();
            }
            node = node->children[index];
        }
        node->isEndOfWord = true;  // 标记单词结尾
    }

    // 搜索单词
    bool search(string word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                return false;  // 找不到对应字符
            }
            node = node->children[index];
        }
        return node->isEndOfWord;  // 只有到达单词结尾才算找到
    }

    // 前缀搜索
    bool startsWith(string prefix) {
        TrieNode* node = root;
        for (char c : prefix) {
            int index = c - 'a';
            if (!node->children[index]) {
                return false;
            }
            node = node->children[index];
```

```
        }
        return true;  // 只要能走完前缀就算找到
    }

    // 析构函数
    ~Trie() {
        deleteTrie(root);
    }

private:
    // 递归删除树节点
    void deleteTrie(TrieNode* node) {
        for (int i = 0; i < 26; i++) {
            if (node->children[i]) {
                deleteTrie(node->children[i]);
            }
        }
        delete node;
    }
};
```

15.2.3 3. Rabin-Karp 算法

Rabin-Karp 算法使用哈希函数计算字符串的哈希值，通过比较哈希值来进行字符串匹配。

核心思想：- 计算模式串的哈希值 - 计算文本串中每个长度为 m 的子串的哈希值 - 比较哈希值，如果相同则进一步验证

特点：- 平均时间复杂度：$O(n+m)$ - 最坏时间复杂度：$O(n*m)$ - 适合多模式串匹配

```
// Rabin-Karp 算法实现
vector<int> rabinKarp(string txt, string pat) {
    int n = txt.length();
    int m = pat.length();
    vector<int> result;

    // 基数，通常选择一个比字符集大小大的素数
    int d = 256;
    // 一个大素数
    int q = 101;

    int h = 1;
    // 计算 h = d^(m-1) % q
    for (int i = 0; i < m - 1; i++) {
        h = (h * d) % q;
    }

    // 计算模式串和文本串第一个窗口的哈希值
    int p = 0;  // 模式串哈希值
    int t = 0;  // 文本串当前窗口的哈希值

    for (int i = 0; i < m; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    // 滑动窗口并比较
    for (int i = 0; i <= n - m; i++) {
        // 如果哈希值相等，进一步比较字符
        if (p == t) {
```

```
            bool match = true;
            for (int j = 0; j < m; j++) {
                if (txt[i + j] != pat[j]) {
                    match = false;
                    break;
                }
            }
            if (match) {
                result.push_back(i);
            }
        }

        // 计算下一个窗口的哈希值
        if (i < n - m) {
            t = (d * (t - txt[i] * h) + txt[i + m]) % q;
            if (t < 0) {
                t += q;
            }
        }
    }

    return result;
}
```

15.3 字符串算法优化技巧

1. **字符串哈希**：将字符串转换为整数，便于快速比较。
   - 常用哈希函数：$h = (h * base + s[i]) \% mod$
   - 常见的 $base$ 值：$31, 131, 1331$ 等
   - 常见的 mod 值：$10^9+7, 10^9+9$ 等
2. **位运算优化**：利用位运算优化字符串处理。
   - 例如，用一个整数表示字符集，利用位操作进行字符集合运算
3. **滑动窗口**：在需要连续处理子串的问题中使用滑动窗口技术。
   - 维护一个窗口，通过添加和删除元素更新窗口状态
4. **双指针技术**：使用两个指针处理字符串。
   - 如对撞指针、快慢指针等

15.4 常见字符串问题类型

1. **模式匹配**：在文本中查找特定的模式串。
   - 精确匹配：KMP, Rabin-Karp 等
   - 近似匹配：编辑距离，最长公共子序列等
2. **字符串统计**：统计字符串中的特定字符或模式。
   - 字符频率统计
   - 最长重复子串
   - 最长回文子串
3. **字符串转换**：将一个字符串转换为另一个字符串。
   - 编辑距离问题
   - 字符串压缩与解压
4. **字符串排序**：特定条件下的字符串排序。
   - 字典序排序
   - 按特定条件排序

15.5 典型例题分析

15.5.1 例题 1：最长回文子串

```
// Manacher 算法求最长回文子串
string longestPalindrome(string s) {
    // 预处理，在字符间插入特殊字符
    string t = "$#";
    for (char c : s) {
        t += c;
        t += '#';
    }
    t += '@';

    int n = t.length();
    vector<int> p(n, 0);  // p[i] 表示以 i 为中心的回文半径

    int center = 0, maxRight = 0;
    int maxLen = 0, start = 0;

    for (int i = 1; i < n - 1; i++) {
        // 初始化 p[i]
        if (i < maxRight) {
            p[i] = min(p[2 * center - i], maxRight - i);
        } else {
            p[i] = 1;
        }

        // 中心扩展
        while (t[i + p[i]] == t[i - p[i]]) {
            p[i]++;
        }

        // 更新 maxRight
        if (i + p[i] > maxRight) {
            center = i;
            maxRight = i + p[i];
        }

        // 更新最长回文子串
        if (p[i] - 1 > maxLen) {
            maxLen = p[i] - 1;
            start = (i - p[i] + 1) / 2;
        }
    }

    return s.substr(start, maxLen);
}
```

15.5.2 例题 2：字符串哈希应用

```
// 使用字符串哈希检查重复子串
bool hasRepeatingSubstring(string s, int len) {
    int n = s.length();
    if (len * 2 > n) return false;  // 无法形成重复子串
```

```
    unordered_map<long long, vector<int>> hash_pos;
    const int base = 31;
    const int mod = 1e9 + 7;

    long long hash = 0;
    long long power = 1;

    // 计算第一个窗口的哈希值
    for (int i = 0; i < len; i++) {
        hash = (hash * base + (s[i] - 'a' + 1)) % mod;
        if (i < len - 1) {
            power = (power * base) % mod;
        }
    }

    hash_pos[hash].push_back(0);

    // 滑动窗口计算后续哈希值
    for (int i = 1; i <= n - len; i++) {
        hash = (hash - (s[i - 1] - 'a' + 1) * power % mod + mod) % mod;
        hash = (hash * base + (s[i + len - 1] - 'a' + 1)) % mod;

        // 检查哈希冲突
        for (int pos : hash_pos[hash]) {
            bool equal = true;
            for (int j = 0; j < len; j++) {
                if (s[pos + j] != s[i + j]) {
                    equal = false;
                    break;
                }
            }
            if (equal) return true;  // 找到重复子串
        }

        hash_pos[hash].push_back(i);
    }

    return false;
}
```

16 拓扑排序

16.1 基本概念

【定义】拓扑排序是将有向无环图中的所有顶点排成一个线性序列，使得图中任意一对顶点 u 和 v，如果存在一条从 u 到 v 的路径，那么在序列中 u 一定出现在 v 之前。

16.1.1 关键特性

1. 拓扑排序只适用于有向无环图（DAG）
2. 如果图中存在环，则无法进行拓扑排序
3. 对同一个图，可能存在多个合法的拓扑排序序列
4. 拓扑排序可以用来检测图中是否存在环

16.2 应用场景

1. **任务调度**：确定具有依赖关系的任务的执行顺序
2. **编译系统**：确定程序模块的编译顺序

3. **课程安排**：根据课程的先修关系安排学习顺序
4. **数据处理管道**：确定数据流的处理顺序

16.3 BFS 实现（Kahn 算法）

Kahn 算法是基于广度优先搜索的拓扑排序算法，其核心思想是不断移除没有入边的节点。

16.3.1 算法步骤

1. 计算图中每个顶点的入度
2. 将所有入度为 0 的顶点加入队列
3. 取出队列中的一个顶点，将其加入结果序列
4. 将该顶点的所有出边删除，更新相邻顶点的入度
5. 如果有新的入度为 0 的顶点，将其加入队列
6. 重复步骤 3-5，直到队列为空
7. 如果结果序列的长度等于图中顶点数，则得到一个拓扑排序；否则，图中存在环，无法进行拓扑排序

16.3.2 代码实现

```
const int MAXN = 100005;

vector<int> graph[MAXN];        // 邻接表表示的有向图
int inDegree[MAXN];             // 记录每个顶点的入度
vector<int> topologicalOrder;   // 存储拓扑排序的结果

// 返回 true 表示成功找到拓扑排序，false 表示图中存在环
bool kahnsAlgorithm(int n) {
    // 初始化入度
    memset(inDegree, 0, sizeof(inDegree));

    // 计算每个顶点的入度
    for (int u = 1; u <= n; u++) {
        for (int v : graph[u]) {
            inDegree[v]++;
        }
    }

    queue<int> q;

    // 将所有入度为 0 的顶点加入队列
    for (int i = 1; i <= n; i++) {
        if (inDegree[i] == 0) {
            q.push(i);
        }
    }

    // 处理队列中的顶点
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        topologicalOrder.push_back(u);  // 将顶点加入拓扑序列

        // 移除所有从 u 出发的边
        for (int v : graph[u]) {
            inDegree[v]--;

            // 如果 v 的入度变为 0，将其加入队列
```

```
            if (inDegree[v] == 0) {
                q.push(v);
            }
        }
    }

    // 检查是否所有顶点都已加入拓扑序列
    return topologicalOrder.size() == n;
}

// 打印拓扑排序结果
void printTopologicalOrder() {
    for (int vertex : topologicalOrder) {
        cout << vertex << " ";
    }
    cout << endl;
}
```

16.3.3 时间复杂度分析

- 时间复杂度：$O(V+E)$，其中 V 是顶点数，E 是边数
- 空间复杂度：$O(V)$，需要额外的队列和入度数组

16.4 DFS 实现

也可以使用深度优先搜索实现拓扑排序，其核心思想是在递归调用结束后将顶点加入序列。

16.4.1 算法步骤

1. 对图中每个未访问的顶点，开始一次 DFS
2. 在 DFS 过程中，递归地访问所有邻接顶点
3. 当一个顶点的所有邻接顶点都已被访问，将该顶点加入结果序列的头部
4. 最终结果序列即为拓扑排序

16.4.2 代码实现

```
const int MAXN = 100005;

vector<int> graph[MAXN];        // 邻接表表示的有向图
bool visited[MAXN];             // 记录顶点是否被访问过
bool inStack[MAXN];             // 记录顶点是否在当前 DFS 路径上（用于检测环）
vector<int> topologicalOrder;   // 存储拓扑排序的结果
bool hasCycle;                  // 标记图中是否存在环

// DFS 遍历
void dfs(int u) {
    visited[u] = true;
    inStack[u] = true;  // 标记顶点 u 在当前 DFS 路径上

    // 访问所有邻接顶点
    for (int v : graph[u]) {
        if (!visited[v]) {
            dfs(v);
        } else if (inStack[v]) {
            // 如果 v 已经在当前 DFS 路径上，说明存在环
            hasCycle = true;
            return;
```

```
        }
    }

    inStack[u] = false;  // 顶点 u 已经处理完毕，移出当前路径
    topologicalOrder.push_back(u);  // 将顶点加入拓扑序列
}

// 拓扑排序主函数
bool topologicalSort(int n) {
    // 初始化
    memset(visited, false, sizeof(visited));
    memset(inStack, false, sizeof(inStack));
    topologicalOrder.clear();
    hasCycle = false;

    // 对每个未访问的顶点开始 DFS
    for (int i = 1; i <= n; i++) {
        if (!visited[i]) {
            dfs(i);
        }
    }

    // 如果存在环，无法进行拓扑排序
    if (hasCycle) {
        return false;
    }

    // 由于 DFS 后加入的顺序与拓扑排序相反，需要反转
    reverse(topologicalOrder.begin(), topologicalOrder.end());
    return true;
}
```

16.4.3 时间复杂度分析

- 时间复杂度：$O(V+E)$
- 空间复杂度：$O(V)$，需要额外的访问标记数组和调用栈空间

16.5 实战应用

16.5.1 判断图中是否存在环

拓扑排序可以用来判断一个有向图是否包含环：- 如果能够得到一个完整的拓扑排序（排序结果包含所有顶点），则图中不存在环 - 如果无法得到完整的拓扑排序，则图中存在环

16.5.2 求解依赖问题

许多实际问题可以建模为依赖图，然后通过拓扑排序求解：

```
// 例：课程安排问题
// n 个课程，课程编号为 1 到 n
// prerequisites[i] = [a, b] 表示课程 a 依赖于课程 b（必须先学习课程 b）
bool canFinish(int n, vector<pair<int, int>>& prerequisites) {
    // 构建邻接表
    vector<int> graph[n+1];
    int inDegree[n+1] = {0};

    for (const auto& pre : prerequisites) {
        int course = pre.first;
        int prereq = pre.second;
```

```
        graph[prereq].push_back(course);
        inDegree[course]++;
    }

    queue<int> q;
    int count = 0;

    // 将所有入度为 0 的课程加入队列
    for (int i = 1; i <= n; i++) {
        if (inDegree[i] == 0) {
            q.push(i);
        }
    }

    // 拓扑排序
    while (!q.empty()) {
        int course = q.front();
        q.pop();
        count++;

        for (int next : graph[course]) {
            inDegree[next]--;
            if (inDegree[next] == 0) {
                q.push(next);
            }
        }
    }

    // 如果所有课程都能被选修，则返回 true
    return count == n;
}
```

16.6 找到所有可能的拓扑排序

在某些情况下，我们可能需要找到所有可能的拓扑排序序列。可以使用回溯法实现：

```
void allTopologicalSorts(int n, vector<int>& result, bool visited[],
                         vector<int> graph[], vector<int>& inDegree) {
    bool allVisited = true;

    // 检查是否所有顶点都已访问
    for (int i = 1; i <= n; i++) {
        if (!visited[i] && inDegree[i] == 0) {
            // 选择当前入度为 0 的顶点
            visited[i] = true;
            result.push_back(i);

            // 减少邻居的入度
            for (int neighbor : graph[i]) {
                inDegree[neighbor]--;
            }

            // 递归查找下一个顶点
            allTopologicalSorts(n, result, visited, graph, inDegree);

            // 回溯
            visited[i] = false;
```

```
            result.pop_back();
            for (int neighbor : graph[i]) {
                inDegree[neighbor]++;
            }

            allVisited = false;
        }
    }

    // 如果所有顶点都已访问，输出当前排序
    if (allVisited && result.size() == n) {
        printArrangement(result);
    }
}
```

16.7 拓扑排序的变种

16.7.1 字典序最小的拓扑排序

有时我们需要找到字典序最小的拓扑排序。只需在 BFS 实现中使用优先队列代替普通队列：

```
bool lexicographicalTopSort(int n) {
    // 使用优先队列（小顶堆）
    priority_queue<int, vector<int>, greater<int>> pq;

    // 将所有入度为 0 的顶点加入队列
    for (int i = 1; i <= n; i++) {
        if (inDegree[i] == 0) {
            pq.push(i);
        }
    }

    // 处理队列中的顶点
    while (!pq.empty()) {
        int u = pq.top();
        pq.pop();
        topologicalOrder.push_back(u);

        for (int v : graph[u]) {
            inDegree[v]--;
            if (inDegree[v] == 0) {
                pq.push(v);
            }
        }
    }

    return topologicalOrder.size() == n;
}
```

16.7.2 关键路径问题

在带权有向无环图中，关键路径是从源点到汇点的最长路径，常用于项目管理中的关键路径分析 (CPM)。可以通过拓扑排序求解：

```
void criticalPath(int n) {
    vector<int> early(n+1, 0);  // 最早开始时间
    vector<int> late(n+1, INT_MAX);  // 最晚开始时间
```

```
    // 计算最早开始时间（正向拓扑排序）
    // ... 拓扑排序代码 ...

    // 根据拓扑排序结果计算每个活动的最早开始时间
    for (int u : topologicalOrder) {
        for (auto& [v, weight] : graph[u]) {
            early[v] = max(early[v], early[u] + weight);
        }
    }

    // 初始化汇点的最晚开始时间
    int sink = n;  // 假设 n 是汇点
    late[sink] = early[sink];

    // 计算最晚开始时间（反向拓扑排序）
    for (int i = topologicalOrder.size() - 1; i >= 0; i--) {
        int u = topologicalOrder[i];
        for (auto& [v, weight] : graph[u]) {
            late[u] = min(late[u], late[v] - weight);
        }
    }

    // 找出关键活动（最早开始时间等于最晚开始时间的活动）
    for (int u = 1; u <= n; u++) {
        if (early[u] == late[u]) {
            cout << u << " is on the critical path\n";
        }
    }
}
```

17 图

17.1 邻接矩阵表示法

【定义】邻接矩阵是一个二维数组，用于表示顶点之间的连接关系。如果顶点 i 和 j 之间有一条边，则 matrix[i][j] 等于这条边的权值（无权图则为 1），否则为 0 或无穷大。

17.1.1 邻接矩阵的 C++ 实现

```
const int MAXN = 1000;  // 最大顶点数
int graph[MAXN][MAXN];  // 邻接矩阵

// 初始化图
void initGraph(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                graph[i][j] = 0;  // 自己到自己的距离为 0
            } else {
                graph[i][j] = INT_MAX;  // 初始化为无穷大，表示不可达
            }
        }
    }
}

// 添加边（有向图）
void addEdge(int from, int to, int weight) {
```

```
    graph[from][to] = weight;
}

// 添加边（无向图）
void addUndirectedEdge(int u, int v, int weight) {
    graph[u][v] = weight;
    graph[v][u] = weight;  // 无向图需要添加两条边
}
```

17.1.2 邻接矩阵的优缺点

优点： 1. 实现简单，容易理解 2. 查询两点是否有边的时间复杂度为 $O(1)$ 3. 适合稠密图（边数接近顶点数的平方）4. 方便实现 Floyd 等算法
缺点： 1. 空间复杂度为 $O(V^2)$，对于大型图会消耗大量内存 2. 对于稀疏图（边数远小于顶点数的平方）效率低下 3. 添加/删除顶点的操作复杂度高

17.2 邻接表表示法

【定义】邻接表使用一个顶点数组，每个顶点维护一个链表，存储与该顶点相连的所有顶点和边的信息。

17.2.1 邻接表的 C++ 实现

使用 STL 的 vector 实现：

```
const int MAXN = 100000;  // 最大顶点数

// 边的结构
struct Edge {
    int to;      // 目标顶点
    int weight;  // 权值

    Edge(int _to, int _weight) : to(_to), weight(_weight) {}
};

vector<Edge> graph[MAXN];  // 邻接表

// 添加边（有向图）
void addEdge(int from, int to, int weight) {
    graph[from].push_back(Edge(to, weight));
}

// 添加边（无向图）
void addUndirectedEdge(int u, int v, int weight) {
    graph[u].push_back(Edge(v, weight));
    graph[v].push_back(Edge(u, weight));
}

// 遍历从顶点 u 出发的所有边
void traverseEdges(int u) {
    for (const Edge& e : graph[u]) {
        cout << "Edge from " << u << " to " << e.to
             << " with weight " << e.weight << endl;
    }
}
```

17.2.2 邻接表的优缺点

优点： 1. 空间复杂度为 $O(V+E)$，适合稀疏图 2. 快速查找顶点的所有邻居 3. 添加边的操作简单高效 4. 适合实现 Dijkstra、Kruskal 等算法
缺点： 1. 查询两点之间是否有边需要 $O(\text{degree})$ 时间 2. 实现相对复杂 3. 不适合需要频繁查询两点连接关系的场景

17.3 其他表示方法

17.3.1 链式前向星

链式前向星是邻接表的一种静态实现，在某些竞赛中很常用：

```
const int MAXN = 10005;  // 最大顶点数
const int MAXM = 100005; // 最大边数

struct Edge {
    int to, weight, next;
} edges[MAXM];

int head[MAXN]; // 每个顶点的第一条边的索引
int cnt = 0;    // 边的计数器

// 初始化
void init() {
    cnt = 0;
    memset(head, -1, sizeof(head));
}

// 添加边
void addEdge(int from, int to, int weight) {
    edges[cnt].to = to;
    edges[cnt].weight = weight;
    edges[cnt].next = head[from];
    head[from] = cnt++;
}

// 遍历从顶点 u 出发的所有边
void traverseEdges(int u) {
    for (int i = head[u]; i != -1; i = edges[i].next) {
        int v = edges[i].to;
        int w = edges[i].weight;
        // 处理边 (u,v,w)
    }
}
```

17.3.2 链式前向星生成的树（粽子树）（U412495）

```
#include <iostream>
#include <cmath>
#include <vector>
#include <algorithm>

using namespace std;

/* ---------- 链式前向星 ---------- */
struct Edge { int to, nxt; };

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    if (!(cin >> n)) return 0;
```

```cpp
// 建图：无向树，根为 n
vector<int> head(n + 1, 0);
vector<Edge> e(2*(n-1) + 5);
int tot = 0;
auto add_edge = [&](int u, int v) {
    e[++tot] = {v, head[u]}; head[u] = tot;
    e[++tot] = {u, head[v]}; head[v] = tot;
};
for (int i = 0; i < n - 1; ++i) {
    int u, v; cin >> u >> v;
    add_edge(u, v);
}
// 读入果实种类 a[i]（可能很大，先压缩）
vector<int> a(n + 1);
for (int i = 1; i <= n; ++i) cin >> a[i];

// --- 值压缩（更快更省内存，也避免 unordered_map 常数）---
vector<int> comp = a;
sort(comp.begin() + 1, comp.end());
comp.erase(unique(comp.begin() + 1, comp.end()), comp.end());
auto get_id = [&](int x) {
    return int(lower_bound(comp.begin() + 1, comp.end(), x) - comp.begin());
};
for (int i = 1; i <= n; ++i) a[i] = get_id(a[i]);
int K = int(comp.size()) - 1;      // 压缩后颜色总数

// --- 非递归 DFS：事件栈 (u, p, entering) ---
// entering=true 表示“将要进入 u”，false 表示“将要退出 u”
struct Event { int u, p; bool in; };

vector<int> ans(n + 1, 0);
vector<int> cnt(K + 1, 0);  // 颜色计数
int distinct = 0;

vector<Event> st;
st.reserve(2*n);
const int root = n;
st.push_back({root, 0, true});

while (!st.empty()) {
    auto [u, p, in] = st.back();
    st.pop_back();
    if (in) {
        // 进入 u：路径上加入 a[u]
        if (++cnt[a[u]] == 1) ++distinct;
        ans[u] = distinct;

        // 安排“退出 u”事件
        st.push_back({u, p, false});

        // 遍历邻接边，压入“进入子节点”
        for (int i = head[u]; i; i = e[i].nxt) {
            int v = e[i].to;
            if (v == p) continue;
```

```
                st.push_back({v, u, true});
            }
        } else {
            // 退出 u：路径上去掉 a[u]
            if (--cnt[a[u]] == 0) --distinct;
        }
    }

    // 输出：每个点 i 到根的不同果实种类数
    for (int i = 1; i <= n; ++i) {
        cout << ans[i] << (i == n ? '\n' : ' ');
    }
    return 0;
}
```

18 最小生成树

最小生成树 (Minimum Spanning Tree, MST) 是图论中的一个重要概念，在网络设计、电路布线等实际问题中有广泛应用。

18.1 基本概念

【定义】给定一个带权无向连通图 $G=(V,E)$，图中每条边 e 都有一个权值 $w(e)$。最小生成树是一个包含 G 中所有顶点的无环连通子图，且所有边的权值之和最小。

关键特性：1. MST 包含图 G 的所有顶点 2. MST 是一棵树（无环连通图）3. MST 的边权之和在所有生成树中最小

18.2 Kruskal 算法

【算法】Kruskal 算法是一种贪心算法，基于边的选择策略：按照边权从小到大的顺序选择边，如果加入该边不会形成环，则将其加入生成树。

18.2.1 算法步骤

1. 将所有边按权值升序排序
2. 初始时，每个顶点构成一个独立的连通分量
3. 按排序后的顺序考察每条边 (u,v)：
    - 如果 u 和 v 不在同一连通分量，则将该边加入生成树，并合并 u 和 v 所在的连通分量
    - 否则，丢弃该边
4. 重复步骤 3，直到加入 n-1 条边（n 为顶点数）或考察完所有边

18.2.2 代码实现

Kruskal 算法通常使用**并查集**来判断两个顶点是否在同一连通分量：

```
struct Edge {
    int u, v, weight;

    Edge(int _u, int _v, int _w) : u(_u), v(_v), weight(_w) {}

    // 重载小于运算符，用于边的排序
    bool operator < (const Edge& other) const {
        return weight < other.weight;
    }
};

vector<Edge> edges; // 存储所有边
vector<Edge> mst;   // 存储最小生成树的边

// 并查集数据结构
```

```
int parent[MAXN];
int rank[MAXN];  // 用于按秩合并

// 初始化并查集
void init(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i;    // 每个节点的父节点初始为自己
        rank[i] = 0;      // 初始秩为 0
    }
}

// 查找节点所在集合的代表（带路径压缩）
int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]);  // 路径压缩
    }
    return parent[x];
}

// 合并两个集合（按秩合并）
void unionSets(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);

    if (rootX == rootY) return;

    if (rank[rootX] < rank[rootY]) {
        parent[rootX] = rootY;
    } else if (rank[rootX] > rank[rootY]) {
        parent[rootY] = rootX;
    } else {
        parent[rootY] = rootX;
        rank[rootX]++;
    }
}

// Kruskal 算法实现
int kruskal(int n) {
    // 初始化并查集
    init(n);

    // 对所有边按权值排序
    sort(edges.begin(), edges.end());

    int totalWeight = 0;  // 最小生成树的总权值
    int edgeCount = 0;    // 已加入生成树的边数

    // 按权值从小到大遍历所有边
    for (const Edge& e : edges) {
        int rootU = find(e.u);
        int rootV = find(e.v);

        // 如果加入这条边不会形成环
        if (rootU != rootV) {
            mst.push_back(e);  // 将边加入 MST
            totalWeight += e.weight;
            unionSets(rootU, rootV);  // 合并集合
```

```
            edgeCount++;

            // 如果已经有 n-1 条边，MST 构建完成
            if (edgeCount == n - 1) {
                break;
            }
        }
    }

    // 检查是否形成了一个完整的生成树
    return (edgeCount == n - 1) ? totalWeight : -1;  // -1 表示图不连通
}
```

18.2.3 时间复杂度分析

- 排序：$O(E \log E)$
- 并查集操作：近似 $O(E)$（使用路径压缩和按秩合并）
- 总时间复杂度：$O(E \log E)$ 或 $O(E \log V)$，因为 E 最多为 V^2，所以 $\log E = O(\log V)$

18.3 Prim 算法

【算法】Prim 算法也是一种贪心算法，但它基于顶点的选择策略：每次选择与已选顶点集合相邻的、权值最小的边所连接的顶点。

18.3.1 算法步骤

1. 选择任意一个顶点作为起点，加入已选顶点集合 S
2. 初始化一个数组 key，$key[v]$ 表示顶点 v 与集合 S 中顶点相连的最小边权
3. 重复以下步骤直到所有顶点都被选择：
    - 在未选择的顶点中找出 key 值最小的顶点 u，将 u 加入 S
    - 更新所有与 u 相邻的未选择顶点 v 的 key 值

18.3.2 代码实现

Prim 算法有朴素实现和堆优化两个版本，下面是堆优化的实现：

```
const int MAXN = 100005;
const int INF = 0x3f3f3f3f;

struct Edge {
    int to, weight;
    Edge(int _to, int _w) : to(_to), weight(_w) {}
};

vector<Edge> graph[MAXN];  // 邻接表表示的图
bool visited[MAXN];        // 记录顶点是否已加入 MST
int key[MAXN];             // key[i] 表示顶点 i 与已选顶点集合相连的最小边权

int prim(int n) {
    // 初始化
    for (int i = 1; i <= n; i++) {
        key[i] = INF;
        visited[i] = false;
    }

    // 从顶点 1 开始
    key[1] = 0;
```

```
    // 使用优先队列（小顶堆）
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, 1});  // {key 值, 顶点编号}

    int totalWeight = 0;
    int vertexCount = 0;  // 记录已加入 MST 的顶点数

    while (!pq.empty() && vertexCount < n) {
        int u = pq.top().second;
        int w = pq.top().first;
        pq.pop();

        if (visited[u]) continue;

        visited[u] = true;
        totalWeight += w;
        vertexCount++;

        // 更新 u 的所有邻接点的 key 值
        for (const Edge& e : graph[u]) {
            int v = e.to;
            int weight = e.weight;

            if (!visited[v] && weight < key[v]) {
                key[v] = weight;
                pq.push({key[v], v});
            }
        }
    }

    // 检查是否所有顶点都已加入 MST
    return (vertexCount == n) ? totalWeight : -1;  // -1 表示图不连通
}
```

18.3.3 时间复杂度分析

- 堆优化的 Prim 算法时间复杂度：$O(E \log V)$
- 朴素 Prim 算法时间复杂度：$O(V^2)$

18.4 Kruskal vs Prim：算法对比

算法	时间复杂度	适用情况	优点	缺点
Kruskal	$O(E \log E)$	适合稀疏图	实现简单，易于理解	需要排序所有边
Prim	$O(E \log V)$（堆优化）	适合稠密图	不需要预先排序所有边	实现相对复杂

18.5 特殊情况处理

18.5.1 次小生成树

次小生成树是指权值和仅次于最小生成树的生成树。求解次小生成树的步骤：

1. 找出最小生成树 T
2. 对于 T 中的每条边 e：
    - 暂时移除 e
    - 在不考虑 e 的情况下，找出连接 e 两端点的最小权边 e'（e'不在 T 中）
    - 计算 T - e + e'的总权值

3. 所有可能替换方案中，权值和最小的即为次小生成树

18.5.2 最小生成森林

当图不连通时，我们需要求解最小生成森林，即每个连通分量的最小生成树的集合。Kruskal 算法自然支持求解最小生成森林。

19 最短路问题

19.1 Dijkstra 算法

【算法】Dijkstra 算法是解决单源最短路径问题的经典算法，适用于**所有边权为非负数**的情况。

19.1.1 算法思想

1. 维护一个距离数组 dist，dist[i] 表示从源点 s 到顶点 i 的当前最短距离
2. 每次从未处理的顶点中选择距离最小的顶点 u
3. 更新 u 的所有邻居 v 的距离：如果 $\text{dist}[u] + \text{weight}(u,v) < \text{dist}[v]$，则更新 dist[v]
4. 重复步骤 2-3 直到所有顶点都被处理

19.1.2 代码实现

基于优先队列的 Dijkstra 算法实现：

```
const int MAXN = 100005;
const int INF = 0x3f3f3f3f;

struct Edge {
    int to, weight;
    Edge(int _to, int _weight) : to(_to), weight(_weight) {}
};

vector<Edge> graph[MAXN];  // 邻接表表示的图
int dist[MAXN];            // 存储从源点到各点的最短距离
bool visited[MAXN];        // 标记顶点是否已经处理过

void dijkstra(int start, int n) {
    // 初始化
    for (int i = 1; i <= n; i++) {
        dist[i] = INF;
        visited[i] = false;
    }
    dist[start] = 0;

    // 使用优先队列优化，pair 的 first 是距离，second 是顶点编号
    // 小顶堆，距离小的优先出队
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, start});

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        if (visited[u]) continue;  // 如果已经处理过，跳过
        visited[u] = true;

        // 更新 u 的所有邻居
        for (const Edge& e : graph[u]) {
            int v = e.to;
```

```
            int weight = e.weight;

            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
}
```

19.1.3 时间复杂度分析

- 使用邻接矩阵：$O(V^2)$
- 使用优先队列优化的邻接表：$O((V+E)\log V)$

19.1.4 应用场景

1. 路径规划
2. 网络路由算法
3. 任何需要找到最小代价路径的问题

19.2 Bellman-Ford 算法

【算法】Bellman-Ford 算法也用于解决单源最短路径问题，但其**可以处理带有负权边**的图，还能检测负权环。

19.2.1 算法思想

1. 初始化 $dist[source]=0$，其他 $dist[v]=\infty$
2. 对所有边进行 $V-1$ 次松弛操作（因为最短路径最多包含 $V-1$ 条边）
3. 再对所有边进行一次松弛操作，如果还有更新，说明存在负权环

19.2.2 代码实现

```
struct Edge {
    int from, to, weight;
};

vector<Edge> edges;  // 存储所有边
int dist[MAXN];      // 存储从源点到各点的最短距离

// 返回 true 表示没有负环，false 表示存在负环
bool bellmanFord(int start, int n, int m) {
    // 初始化
    for (int i = 1; i <= n; i++) {
        dist[i] = INF;
    }
    dist[start] = 0;

    // 进行 n-1 次松弛
    for (int i = 1; i <= n - 1; i++) {
        bool updated = false;

        // 对每条边进行松弛操作
        for (const Edge& e : edges) {
            if (dist[e.from] != INF && dist[e.from] + e.weight < dist[e.to]) {
                dist[e.to] = dist[e.from] + e.weight;
                updated = true;
```

```
            }
        }

        // 如果这一轮没有更新，提前退出
        if (!updated) break;
    }

    // 检测负环
    for (const Edge& e : edges) {
        if (dist[e.from] != INF && dist[e.from] + e.weight < dist[e.to]) {
            // 存在负环
            return false;
        }
    }

    return true;  // 不存在负环
}
```

19.2.3 时间复杂度分析

- 时间复杂度：$O(VE)$，其中 V 是顶点数，E 是边数

19.2.4 SPFA 算法

SPFA (Shortest Path Faster Algorithm) 是 Bellman-Ford 的队列优化版本，平均复杂度为 $O(kE)$，k 一般很小。

```
bool spfa(int start, int n) {
    // 初始化
    for (int i = 1; i <= n; i++) {
        dist[i] = INF;
        inQueue[i] = false;
        count[i] = 0;  // 记录顶点入队次数，用于判断负环
    }

    dist[start] = 0;
    queue<int> q;
    q.push(start);
    inQueue[start] = true;
    count[start]++;

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inQueue[u] = false;

        // 更新 u 的所有邻居
        for (const Edge& e : graph[u]) {
            int v = e.to;
            int w = e.weight;

            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;

                if (!inQueue[v]) {
                    q.push(v);
                    inQueue[v] = true;
                    count[v]++;
```

```
                    // 如果一个顶点入队次数超过 n，说明存在负环
                    if (count[v] > n) {
                        return false;  // 存在负环
                    }
                }
            }
        }
    }

    return true;  // 不存在负环
}
```

19.3 Floyd-Warshall 算法

【算法】Floyd-Warshall 算法用于解决**多源最短路径问题**，即求出图中任意两点之间的最短路径。

19.3.1 算法思想

1. 通过三重循环，考虑对于每一对顶点 (i,j)，是否存在一个顶点 k，使得从 i 到 k 再到 j 的路径比已知的从 i 到 j 的路径更短
2. 不断更新这些中间路径，最终得到任意两点间的最短路径

19.3.2 代码实现

```
const int MAXN = 505;
const int INF = 0x3f3f3f3f;

int dist[MAXN][MAXN];  // 存储任意两点间的最短距离

void floyd(int n) {
    // 初始化，dist[i][j] 已经包含了直接连接的边

    // 核心算法：考虑所有可能的中转点
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }
}
```

19.3.3 时间复杂度分析

- 时间复杂度：$O(V^3)$
- 空间复杂度：$O(V^2)$

19.3.4 应用场景

1. 需要求解所有点对之间最短路径的场景
2. 图的顶点数不是很大（小于 1000）的情况
3. 传递闭包问题

19.4 各算法对比与选择

算法	时间复杂度	适用情况	优点	缺点
Dijkstra	$O((V+E)\log V)$	无负权边的单源最短路	效率高	不能处理负权边
Bellman-Ford	$O(VE)$	可能有负权边的单源最短路	可检测负权环	效率较低
SPFA	平均 $O(kE)$	可能有负权边的单源最短路	一般情况下比 Bellman-Ford 快	最坏情况下仍是 $O(VE)$
Floyd-Warshall	$O(V^3)$	多源最短路	代码简洁，可处理负权边	顶点数大时效率低

19.5 最小最大流算法（U416838）

例题：有 10^9 台设备分布在一条数轴上，第 i 台设备的坐标为 i。有 n 位维修工，初始时第 i 位维修工的位置为 a_{i}。这些设备共发生了 m 次故障，第 j 次故障的设备为 b_{j}，你需要指定一名维修工维修设备，他将从他当前所在的位置移动到位置 b_{j}。维修工从位置 x 移动到位置 y 需要花费 x—y 的代价。

已知所有维修工的初始位置（一个数组），你需要合理调配维修工，在每次故障发生后及时完成维修，即必须依次完成 m 次维修。求所有维修的代价总和的最小值。

```cpp
# include <iostream>
# include <cmath>
# include <algorithm>
# include <vector>
# include <queue>
using namespace std;

struct MCMF {
    struct Edge { int v, rev, cap; long long cost; };
    int N;
    vector<vector<Edge>> G;
    MCMF(int n): N(n), G(n) {}
    void addEdge(int u, int v, int cap, long long cost){
        Edge a{v, (int)G[v].size(), cap, cost};
        Edge b{u, (int)G[u].size(), 0,   -cost};
        G[u].push_back(a); G[v].push_back(b);
    }
    pair<int,long long> minCostMaxFlow(int s, int t, int need){
        const long long INF = (1LL<<62);
        vector<long long> pi(N,0), dist(N);
        vector<int> pv_v(N), pv_e(N);
        auto dijkstra = [&]()->bool{
            fill(dist.begin(), dist.end(), INF);
            dist[s]=0;
            using P=pair<long long,int>;
            priority_queue<P, vector<P>, greater<P>> pq;
            pq.push({0,s});
            while(!pq.empty()){
                auto [d,u]=pq.top(); pq.pop();
                if(d!=dist[u]) continue;
                for(int i=0;i<(int)G[u].size();++i){
                    auto &e=G[u][i];
                    if(e.cap<=0) continue;
                    long long w=e.cost + pi[u]-pi[e.v];
                    if(dist[e.v]>d+w){
                        dist[e.v]=d+w; pv_v[e.v]=u; pv_e[e.v]=i;
                        pq.push({dist[e.v], e.v});
                    }
                }
            }
            if(dist[t]==INF) return false;
            for(int i=0;i<N;++i) if(dist[i]<INF) pi[i]+=dist[i];
            return true;
```

```
        };
        int flow=0; long long cost=0;
        while(flow<need && dijkstra()){
            int add = need - flow;
            int v=t;
            while(v!=s){
                auto &e=G[pv_v[v]][pv_e[v]];
                add=min(add, e.cap);
                v=pv_v[v];
            }
            v=t;
            while(v!=s){
                auto &e=G[pv_v[v]][pv_e[v]];
                auto &re=G[v][e.rev];
                e.cap-=add; re.cap+=add;
                cost += (long long)add * e.cost;
                v=pv_v[v];
            }
            flow+=add;
        }
        return {flow,cost};
    }
};

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n,m;
    if(!(cin>>n>>m)) return 0;
    vector<long long>a(n); for(int i=0;i<n;++i) cin>>a[i];
    vector<long long>b(m+1); for(int j=1;j<=m;++j) cin>>b[j];

    // Nodes:
    // S = 0
    // Workers W_i: 1..n
    // Left events E^L_j: n+1 .. n+m
    // Right events E^R_j: n+m+1 .. n+2m
    // T = n + 2m + 1
    int S=0;
    int Wbase=1;
    int Lbase=Wbase+n;
    int Rbase=Lbase+m;
    int T=Rbase+m;

    MCMF mf(T+1);

    // S -> workers (cap 1)
    for(int i=0;i<n;++i) mf.addEdge(S, Wbase+i, 1, 0);

    // S -> left events (each event can be predecessor at most once)
    for(int j=1;j<=m;++j) mf.addEdge(S, Lbase+(j-1), 1, 0);

    // Workers -> right events (start a chain at event j)
    for(int i=0;i<n;++i)
        for(int j=1;j<=m;++j)
            mf.addEdge(Wbase+i, Rbase+(j-1), 1, llabs(b[j]-a[i]));
```

```
    // Event u (as predecessor) -> right event v (u < v) (continue chain)
    for(int u=1; u<=m; ++u)
        for(int v=u+1; v<=m; ++v)
            mf.addEdge(Lbase+(u-1), Rbase+(v-1), 1, llabs(b[v]-b[u]));

    // Right events -> T (each event must be covered exactly once)
    for(int j=1;j<=m;++j) mf.addEdge(Rbase+(j-1), T, 1, 0);

    // Need exactly m units of flow (cover all m events)
    auto [flow, cost] = mf.minCostMaxFlow(S, T, m);
    if(flow!=m) cout << "-1\n";  // 理论上不会发生
    else cout << cost << "\n";
    return 0;
}
```

20 位运算优化

【位运算】是直接对整数二进制位进行操作的技术，在 ACM 竞赛中经常用于优化算法、降低时间复杂度和空间复杂度。掌握位运算技巧可以帮助我们编写更高效的代码。

20.1 基本位运算符

C++ 中常用的位运算符包括：

运算符	名称	功能
&	按位与	两个位都为 1 时，结果为 1，否则为 0
|	按位或	两个位只要有一个为 1，结果就为 1
^	按位异或	两个位相同时为 0，不同时为 1
~	按位取反	0 变 1，1 变 0
<<	左移	各二进位全部左移若干位，高位丢弃，低位补 0
>>	右移	各二进位全部右移若干位，对无符号数，高位补 0

20.2 常用位运算技巧

20.2.1 1. 判断奇偶性

使用 &1 来判断一个数是否为奇数：

```
// 检查 n 是否为奇数
bool isOdd(int n) {
    return n & 1;  // 等价于 n % 2 != 0
}
```

20.2.2 2. 乘除 2 的幂

使用位移操作可以高效实现乘以或除以 2 的幂：

```
// 乘以 2 的 k 次方，等价于 n * (2^k)
int multiplyByPowerOf2(int n, int k) {
    return n << k;
}

// 除以 2 的 k 次方，等价于 n / (2^k)
int divideByPowerOf2(int n, int k) {
    return n >> k;
}
```

20.2.3 3. 获取二进制表示中的特定位

获取第 i 位（从 0 开始，最低位为第 0 位）：

```
// 获取 n 的第 i 位（0 或 1）
int getBit(int n, int i) {
    return (n >> i) & 1;
}
```

20.2.4 4. 设置或清除特定位

```
// 将 n 的第 i 位设置为 1
int setBit(int n, int i) {
    return n | (1 << i);
}

// 将 n 的第 i 位设置为 0
int clearBit(int n, int i) {
    return n & ~(1 << i);
}

// 将 n 的第 i 位取反
int flipBit(int n, int i) {
    return n ^ (1 << i);
}
```

20.2.5 5. 计算二进制中 1 的个数

```
// 计算 n 的二进制表示中 1 的个数
int countOnes(int n) {
    int count = 0;
    while (n) {
        count += n & 1;
        n >>= 1;
    }
    return count;
}

// C++ 内置函数（更高效）
int countOnes(int n) {
    return __builtin_popcount(n);  // GCC 编译器特有
}
```

20.2.6 6. 获取最低位的 1

```
// 获取 n 二进制表示中最低位的 1
// 例如：n = 12 (1100)，结果为 4 (100)
int lowestOneBit(int n) {
    return n & -n;  // 或写作 n & (~n + 1)
}
```

20.2.7 7. 去除最低位的 1

```cpp
// 去除 n 二进制表示中的最低位 1
// 例如：n = 12 (1100)，结果为 8 (1000)
int removeLowestOneBit(int n) {
    return n & (n - 1);
}
```

20.2.8 8. 判断是否为 2 的幂

```cpp
// 判断 n 是否为 2 的幂
bool isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}
```

20.3 高级应用

20.3.1 1. 子集枚举

使用位运算可以方便地枚举一个集合的所有子集：

```cpp
void subsets(vector<int>& nums) {
    int n = nums.size();
    int subsetCount = 1 << n;  // 2^n 个子集

    for (int mask = 0; mask < subsetCount; mask++) {
        cout << " 子集 {";
        for (int i = 0; i < n; i++) {
            // 检查第 i 位是否为 1
            if (mask & (1 << i)) {
                cout << nums[i] << " ";
            }
        }
        cout << "}" << endl;
    }
}
```

20.3.2 2. 状态压缩 DP

状态压缩动态规划中，我们用整数的二进制位表示状态：

```cpp
// 旅行商问题 (TSP) 的状态压缩 DP 示例
int tsp(vector<vector<int>>& dist) {
    int n = dist.size();
    vector<vector<int>> dp(1 << n, vector<int>(n, INT_MAX));

    // 初始状态：只访问城市 0
    dp[1][0] = 0;

    // 枚举所有状态
    for (int mask = 1; mask < (1 << n); mask++) {
        for (int u = 0; u < n; u++) {
            // 如果 u 不在当前集合中，跳过
            if (!(mask & (1 << u))) continue;

            // prev_mask 表示不包含 u 的状态
            int prev_mask = mask ^ (1 << u);

            // 如果 prev_mask 为 0，说明 mask 只包含 u
```

```
            if (prev_mask == 0) continue;

            // 尝试从其他城市到达 u
            for (int v = 0; v < n; v++) {
                if (v == u) continue;
                if (prev_mask & (1 << v)) {
                    dp[mask][u] = min(dp[mask][u], dp[prev_mask][v] + dist[v][u]);
                }
            }
        }
    }

    // 计算从最后一个城市回到起点的距离
    int finalMask = (1 << n) - 1;  // 所有城市都已访问
    int minDist = INT_MAX;
    for (int u = 1; u < n; u++) {
        if (dp[finalMask][u] != INT_MAX) {
            minDist = min(minDist, dp[finalMask][u] + dist[u][0]);
        }
    }

    return minDist;
}
```

20.3.3 3. 快速幂运算

使用位运算优化快速幂算法：

```
// 计算 (a^b) % mod
long long quickPow(long long a, long long b, long long mod) {
    long long result = 1;
    a %= mod;

    while (b > 0) {
        // 如果当前二进制位为 1，将 a 累乘到结果中
        if (b & 1) {
            result = (result * a) % mod;
        }

        // 处理下一位
        a = (a * a) % mod;
        b >>= 1;
    }

    return result;
}
```

20.3.4 4. 位域数据结构

可以使用位运算来实现高效的集合操作：

```
class BitSet {
private:
    vector<unsigned int> bits;
    int size;

public:
```

```
    BitSet(int n) {
        size = n;
        // 每个 unsigned int 可以存储 32 个位
        bits.resize((n + 31) / 32, 0);
    }

    // 设置第 i 位为 1
    void set(int i) {
        if (i >= size) return;
        bits[i / 32] |= (1u << (i % 32));
    }

    // 清除第 i 位
    void clear(int i) {
        if (i >= size) return;
        bits[i / 32] &= ~(1u << (i % 32));
    }

    // 检查第 i 位是否为 1
    bool test(int i) {
        if (i >= size) return false;
        return (bits[i / 32] & (1u << (i % 32))) != 0;
    }

    // 位运算操作
    BitSet& operator&=(const BitSet& other) {
        for (int i = 0; i < bits.size(); i++) {
            bits[i] &= other.bits[i];
        }
        return *this;
    }

    BitSet& operator|=(const BitSet& other) {
        for (int i = 0; i < bits.size(); i++) {
            bits[i] |= other.bits[i];
        }
        return *this;
    }

    BitSet& operator^=(const BitSet& other) {
        for (int i = 0; i < bits.size(); i++) {
            bits[i] ^= other.bits[i];
        }
        return *this;
    }
};
```

20.3.5 5. 位掩码快速枚举 (U423279)

```
int n; // 元素总数
vector<int> arr(n); // 元素数组（这里可以是 Card）

for (int mask = 1; mask < (1 << n); mask++) {
    int cnt = __builtin_popcount(mask); // 子集大小
    if (cnt < L || cnt > R) continue;   // 控制子集大小范围，比如 L=1, R=5
```

```
    vector<int> subset;
    for (int i = 0; i < n; i++) {
        if (mask & (1 << i)) {
            subset.push_back(arr[i]);    // 把第 i 个元素放进子集
        }
    }

    // 在这里对子集 subset 做计算（比如调用 evaluate() 得分）
}
```

20.4 位运算的应用场景

1. **优化算术运算**：使用位运算代替乘除法
2. **集合操作**：高效实现集合的交、并、补等操作
3. **状态压缩**：在动态规划中表示状态
4. **权限管理**：使用位掩码表示权限组合
5. **位图排序**：处理大量整数的排序和去重
6. **散列函数**：构造高效的散列函数
7. **编码算法**：在压缩和加密算法中应用

20.5 典型例题

20.5.1 例题 1：找出唯一出现一次的数

问题：给定一个数组，其中除了一个数出现一次外，其他数都出现两次，找出这个只出现一次的数。

解法：利用异或操作的性质 `a ^ a = 0` 和 `a ^ 0 = a`

```
int findSingle(vector<int>& nums) {
    int result = 0;
    for (int num : nums) {
        result ^= num;
    }
    return result;
}
```

20.5.2 例题 2：格雷码生成

问题：生成 n 位二进制的格雷码序列。

解法：利用位运算转换二进制码到格雷码

```
vector<int> grayCode(int n) {
    vector<int> result;
    int size = 1 << n;  // 2^n

    for (int i = 0; i < size; i++) {
        // 将二进制转换为格雷码：G(i) = i ^ (i >> 1)
        result.push_back(i ^ (i >> 1));
    }

    return result;
}
```

20.6 注意事项

1. 【可读性问题】：位运算代码往往难以理解，建议添加详细注释
2. 【溢出问题】：位移操作可能导致溢出，尤其是左移
3. 【符号问题】：右移符号位的处理在不同编译器可能不同
4. 【跨平台问题】：整数大小在不同平台可能不同，影响位运算结果

5. 【优化权衡】：位运算优化有时会降低代码可读性，需要权衡

20.7 常见位运算公式

- **交换两数**：`a ^= b; b ^= a; a ^= b;`
- **取绝对值**（仅限 32 位整数）：`(n ^ (n >> 31)) - (n >> 31);`
- **取二进制的最低位 1**：`n & -n;`
- **去掉二进制末尾的 1**：`n & (n - 1);`
- **二进制中 1 的个数**：`__builtin_popcount(n);`
- **整数的平均值**：`(a & b) + ((a ^ b) >> 1);`
- **判断两个数符号是否相同**：`(a ^ b) >= 0;`

21 非位优化

21.1 非位优化方法概览

21.1.1 输入输出优化

在处理大量数据的题目中，输入输出的效率可能成为程序的瓶颈。

```
// 关闭同步，加速 cin/cout（可以使 cin/cout 比 scanf/printf 快）
ios::sync_with_stdio(false);
cin.tie(nullptr);

// 对于非常大的输入输出，可以使用快速读写函数
inline int read() {
    int x = 0, f = 1;
    char ch = getchar();
    while (ch < '0' || ch > '9') {
        if (ch == '-') f = -1;
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = x * 10 + ch - '0';
        ch = getchar();
    }
    return x * f;
}
```

21.1.2 内存优化

内存管理在竞赛中同样重要，特别是当题目有严格的内存限制时。

```
// 使用静态数组替代动态分配
const int MAXN = 1e5 + 5;
int dp[MAXN];  // 比 vector<int> dp(n) 更高效

// 动态规划中使用滚动数组
// 原始二维 DP: dp[n][m]
// 优化后: dp[2][m] 或 dp[m]（根据依赖关系）
```

21.1.3 算法设计优化

合理的算法设计可以大大提高程序效率。

```
// 预计算优化（如前缀和、差分）
vector<int> preSum(n + 1, 0);
for (int i = 1; i <= n; i++) {
```

```
    preSum[i] = preSum[i-1] + arr[i-1];
}
// 区间和查询: O(1)
int rangeSum(int l, int r) {
    return preSum[r+1] - preSum[l];
}

// 剪枝优化（以 DFS 为例）
void dfs(int state, int depth) {
    if (!promising(state)) return;  // 剪枝
    // 继续搜索...
}
```

21.1.4 常量优化

有时，即使算法复杂度相同，但通过减少常数因子也能显著提升性能。

```
// 避免重复计算
int size = v.size();  // 避免在循环中反复调用 size()
for (int i = 0; i < size; i++) {
    // 处理 v[i]...
}

// 使用 emplace_back 替代 push_back
v.emplace_back(x);  // 直接构造，避免临时对象的创建
```

21.1.5 数据结构选择

合适的数据结构选择可以大大提升算法效率。

需求	最佳选择	次优选择
快速访问任意元素	数组/vector	-
频繁在两端操作	deque	两个栈/队列模拟
需要保持元素有序	set/map	手动排序的数组
需要快速查找	unordered_map	map
区间查询和修改	线段树/树状数组	分块处理

21.2 编程习惯与技巧

良好的编程习惯能减少错误，提高效率。

1. **使用模板**：准备好常用算法和数据结构的模板，节省编码时间
2. **变量命名**：使用有意义的变量名，但保持简洁
3. **代码结构**：保持逻辑清晰，使用函数分割复杂逻辑
4. **注释关键部分**：特别是复杂算法和易错点
5. **调试技巧**：

```
#define DEBUG
#ifdef DEBUG
#define dbg(x) cout << #x << " = " << x << endl
#else
#define dbg(x)
#endif
```

21.3 常见错误及其避免方法

1. **整数溢出**：对于可能很大的结果，使用 `long long` 而不是 `int`
2. **数组越界**：总是检查索引范围，数组大小预留额外空间
3. **精度问题**：浮点数比较时使用 eps（如 `fabs(a - b) < 1e-9`）
4. **死循环**：确保循环条件最终会被满足
5. **内存限制**：注意大数组的静态分配可能导致栈溢出

21.4 案例分析：优化前后的对比

21.4.1 例 1：求区间和

未优化版本：每次查询都重新计算区间和

```
int rangeSum(vector<int>& arr, int l, int r) {
    int sum = 0;
    for (int i = l; i <= r; i++) {
        sum += arr[i];
    }
    return sum;
}
// 时间复杂度：O(n) 每次查询
```

优化版本：使用前缀和

```
vector<int> buildPrefixSum(vector<int>& arr) {
    int n = arr.size();
    vector<int> prefix(n + 1, 0);
    for (int i = 0; i < n; i++) {
        prefix[i+1] = prefix[i] + arr[i];
    }
    return prefix;
}

int rangeSum(vector<int>& prefix, int l, int r) {
    return prefix[r+1] - prefix[l];
}
// 预处理时间：O(n)，查询时间：O(1)
```

21.4.2 例 2：查找元素

未优化版本：线性查找

```
bool contains(vector<int>& arr, int target) {
    for (int x : arr) {
        if (x == target) return true;
    }
    return false;
}
// 时间复杂度：O(n)
```

优化版本：使用哈希表

```
unordered_set<int> buildSet(vector<int>& arr) {
    return unordered_set<int>(arr.begin(), arr.end());
}

bool contains(unordered_set<int>& s, int target) {
    return s.count(target) > 0;
```

```
}
// 预处理时间：O(n)，查询时间：O(1) 平均
```

22 gdb 调试指令

指令	说明
`gdb prog`	调试程序 prog
`gdb --args prog arg1 arg2`	调试带参数的程序
`run` / `r`	启动程序
`run arg1 arg2`	启动时传递参数
`start`	运行到 `main()` 并停下
`quit` / `q`	退出 gdb

指令	说明
`break func` / `b func`	在函数入口设置断点
`break file:line`	在指定文件的行设置断点
`break N`	在当前文件第 N 行设置断点
`tbreak ...`	临时断点（命中一次后删除）
`info break`	查看所有断点
`delete N`	删除编号为 N 的断点
`disable N`	禁用断点
`enable N`	启用断点
`clear`	清除当前行断点

指令	说明
`next` / `n`	单步执行（不进入函数）
`step` / `s`	单步执行（进入函数）
`finish`	运行到当前函数返回
`continue` / `c`	继续运行直到下一个断点
`until`	运行到循环结束或指定行
`jump line`	直接跳到某一行执行
`return`	强制返回当前函数

指令	说明
`print expr` / `p expr`	打印表达式/变量值
`print/x expr`	按十六进制显示
`print/d expr`	按十进制显示
`display expr`	每次停下时自动显示变量值
`undisplay N`	取消自动显示
`info locals`	查看局部变量
`info args`	查看函数参数
`backtrace` / `bt`	打印调用栈
`frame N`	切换到调用栈第 N 帧
`info registers`	查看寄存器
`info variables`	列出全局/静态变量

指令	说明
`set var x=10`	修改变量值
`call func(args)`	在调试时调用函数
`return expr`	提前返回并指定返回值

指令	说明
`x/Nfu addr`	查看内存 (N= 数量 f= 格式 u= 单位大小)
示例	`x/10xw 0x1234` → 从地址 0x1234 查看 10 个 4 字节单元，十六进制显示
`disassemble`	反汇编当前函数
`disassemble func`	反汇编指定函数

23 遍历打法

- 遍历所有长度为 n 的序列

```
string intToPermutation(int id, int n) {
    vector<int> nums(n);
    iota(nums.begin(), nums.end(), 1);
    string perm;
    vector<int> fact(n);
    fact[0] = 1;
    for (int i = 1; i < n; i++) fact[i] = fact[i - 1] * i;

    // 可选：保证 id 合法
    if (id < 0 || id >= fact[n - 1] * n) return "";

    for (int i = n - 1; i >= 0; i--) {
        int idx = id / fact[i];
        id %= fact[i];
        // 将 1..9 映射到 '1'..'9'
        perm.push_back(char('0' + nums[idx]));
        nums.erase(nums.begin() + idx);
    }
    return perm; // 例如 "12345678"
}
```

```
vector<int> intToPermutation(int id, int n) {
    vector<int> nums(n);
    iota(nums.begin(), nums.end(), 1); // nums = {1,2,...,n}
    vector<int> perm;
    vector<int> fact(n);
    fact[0] = 1;
    for (int i = 1; i < n; i++) fact[i] = fact[i-1] * i;

    for (int i = n-1; i >= 0; i--) {
        int idx = id / fact[i];
        id %= fact[i];
        perm.push_back(nums[idx]);
        nums.erase(nums.begin() + idx);
    }
    return perm;
}
```

- 遍历所有子集（见位运算）

- dfs 遍历（见 DFS）
- bfs 遍历（见 BFS）

24 如果考场只有 gcc

如果考场只给了 gcc，用下面任一做法即可：

24.0.1 一条命令编译并手动链接 C++ 标准库

```
gcc -x c++ -std=gnu++17 2024_2.cpp -lstdc++ -lm -o main
```

- -x c++：强制按 C++ 处理（避免某些环境把.cpp 当 C 处理的坑）。
- -lstdc++：**关键**，手动链接 C++ 标准库。
- -lm：如果你代码里用到 math.h/cmath 的数学函数，顺手加上。
- 注意：**库选项要放在源文件或.o 之后**（顺序很重要）。

24.0.2 分步（先编译再链接）

```
# 编译
gcc -x c++ -std=gnu++17 -O2 -c 2024_2.cpp -o 2024_2.o
# 链接
gcc 2024_2.o -lstdc++ -lm -o main
```

24.0.3 多文件示例

```
gcc -x c++ -std=gnu++17 -O2 -c a.cpp -o a.o
gcc -x c++ -std=gnu++17 -O2 -c b.cpp -o b.o
gcc a.o b.o -lstdc++ -lm -o main
```

24.0.4 需要静态链接（有些 OJ/沙箱环境要求）

```
gcc -x c++ -std=gnu++17 2024_2.cpp -o main -lstdc++ -static-libstdc++ -static-libgcc
```

（是否可用取决于考场机器是否安装了对应的静态库）

额外检查：确保源文件里包含了正确的 C++ 头（如 #include <iostream>），否则也会在链接前就报编译错。