# 七、图论算法

### 考点分析

|  |  |  |  |
| --- | --- | --- | --- |
| **结构类型** | **重点考点** | **考察频率** | **难度** |
| 基础概念 | 图的度与握手定理 | 25% | ★★ |
| 有向图/无向图性质 | 20% | ★★ |
| 邻接矩阵/链表存储差异 | 15% | ★★ |
| 图遍历 | DFS递归栈与回溯机制 | 30% | ★★★ |
| BFS队列应用与层序控制 | 25% | ★★ |
| 遍历应用（连通块计数） | 20% | ★★ |
| 连通性 | 强连通分量（Kosaraju/Tarjan） | 25% | ★★★ |
| 割点与桥判定 | 20% | ★★★ |
| 双连通分量 | 15% | ★★★★ |
| 拓扑排序 | DAG约束与入度0起点 | 30% | ★★ |
| 结果不唯一性 | 25% | ★★ |
| 检测环的存在 | 20% | ★★★ |
| 最短路径 | Dijkstra贪心策略 | 25% | ★★★ |
| Floyd动态规划 | 20% | ★★★ |
| Bellman-Ford负权处理 | 15% | ★★★★ |
| 最小生成树 | Kruskal贪心+并查集 | 20% | ★★★ |
| Prim算法与堆优化 | 15% | ★★★ |
| 树相关 | 树的性质（n结点→n-1边） | 25% | ★★ |
| 二叉树遍历与重构 | 20% | ★★★ |
| LCA最近公共祖先 | 15% | ★★★★ |
| 特殊图 | 二分图判定（染色法） | 20% | ★★★ |
| 欧拉图/哈密顿图性质 | 15% | ★★★ |
| 网络流基本概念 | 10% | ★★★★ |

### 核心知识点

#### 1. 基础概念

**（1）图的度与握手定理**

* **核心知识点**【**握手定理】**：无向图**所有顶点度数之和**等于**边数的2倍**
* **黄金法则**：计算顶点数时优先考虑握手定理

1

3

2

4

* **高频公式**：

int total\_degree = 0;

for (int v : degrees) total\_degree += v;

bool is\_valid = (**total\_degree == 2 \* num\_edges**); // 验证图合法性

* **易错点**：有向图中入度和≠出度和时必然存在数据错误
* **真题示例**：2016-15（16条边→16个顶点）

**（2）有向图/无向图性质**

* **核心知识点**：
  + **强连通图**：**任意两点互相可达**（2011-19）
  + **简单图**：无自环、无重边（最大边数：无向图C(n,2)，有向图2C(n,2)）
* **黄金法则**：**强连通分量数量=1**是强连通图的充要条件

1

3

2

4

* **易错点**：弱连通图（忽略方向后连通）≠ 强连通图

**（3）邻接矩阵/链表存储**

* **核心知识点**：

// 邻接矩阵（稠密图）

int adjMatrix**[MAX][MAX]**;

// 邻接表（稀疏图）

vector<vector<pair<int, int>>> adjList; // pair<邻接点, 边权>

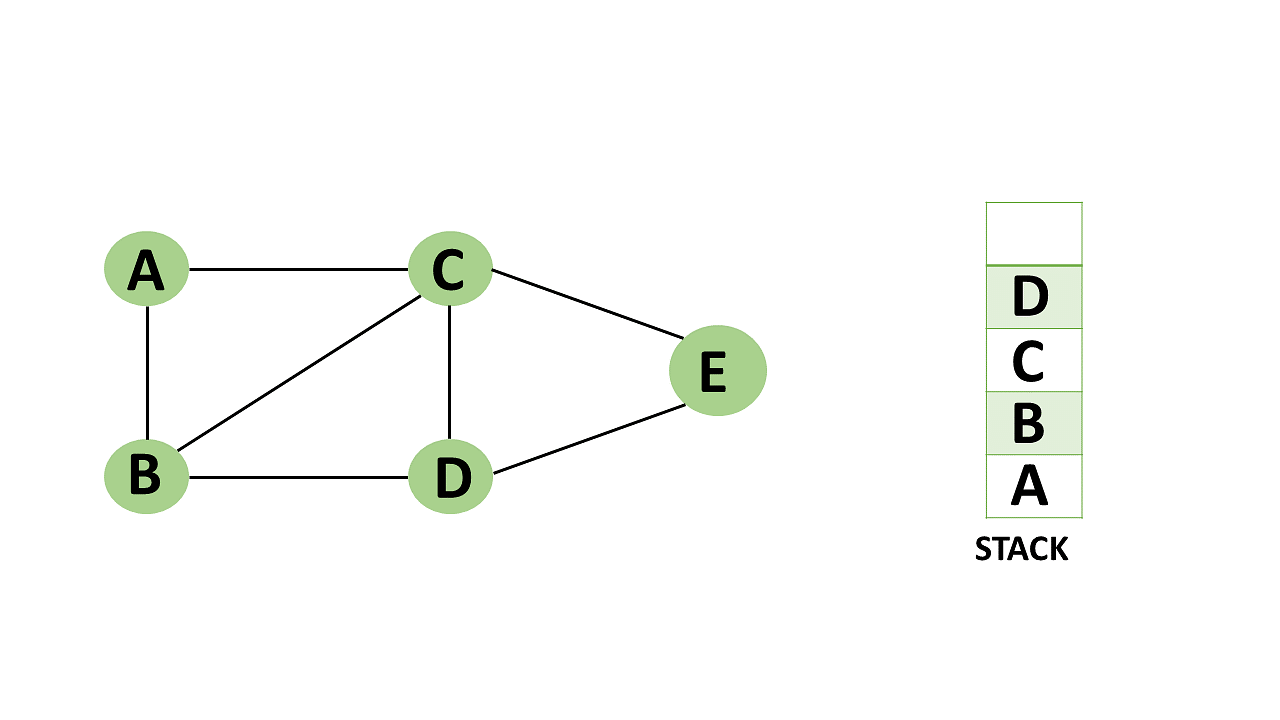
* **黄金法则**：|E| 接近 |V|² 用矩阵，否则用邻接表
* **易错点**：无向图存储需双向添加边

#### 2. 图遍历

**（1）DFS递归栈与回溯**

* **核心知识点**：LIFO特性实现深度探索，递归隐式使用系统栈
* **黄金法则**：**visited数组防重复访问，回溯时状态还原**
* **模板代码**：

void dfs(int u, vector<bool>& vis, vector<vector<int>>& adj) {

 vis[u] = true;

for (int v : adj[u]) {

**if (!vis[v]) {**

// 预处理操作

**dfs(v, vis, adj);**

// 回溯操作

}

}

}

* **易错点**：非连通图需多次调用DFS
* **真题示例**：2013-12（A0→A1→A2后无法直接跳转A3）

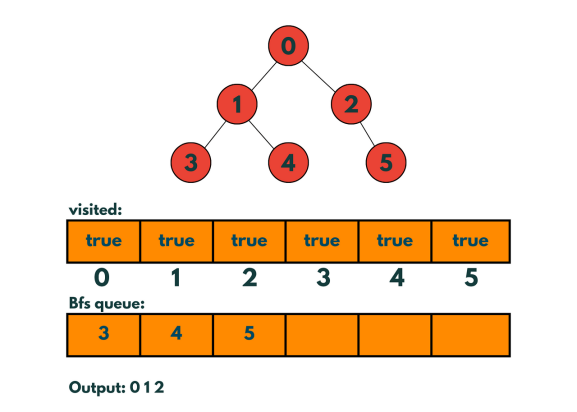
**（2）BFS队列应用**

* **核心知识点**：FIFO特性实现层序扩展，适合**最短路径**
* **黄金法则**：入队时标记visited防重复访问
* **模板代码**：

void bfs(int start, vector<vector<int>>& adj) {

queue<int> q;

vector<bool> vis(adj.size(), false);

 q.push(start);

vis[start] = true;

while (!q.empty()) {

**int u = q.front(); q.pop();**

for (int v : adj[u]) {

**if (!vis[v]) {**

vis[v] = true;

**q.push(v);**

// 层数记录：dist[v] = dist[u] + 1

}

}

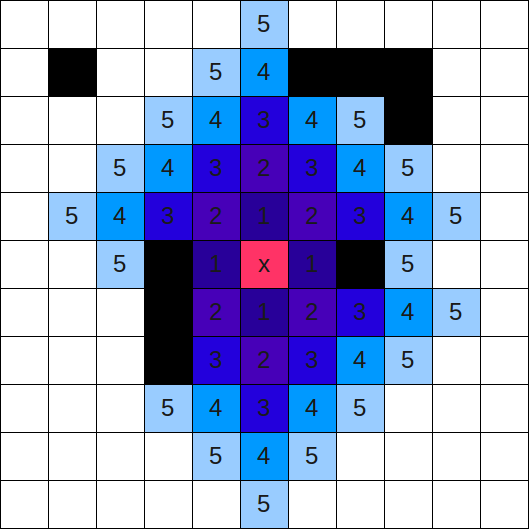
}

}

* **真题示例**：2022-20（洪水填充BFS实现）

**（3）连通块计数**

* **核心公式：**

int count = 0;

vector<bool> vis(n, false);

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

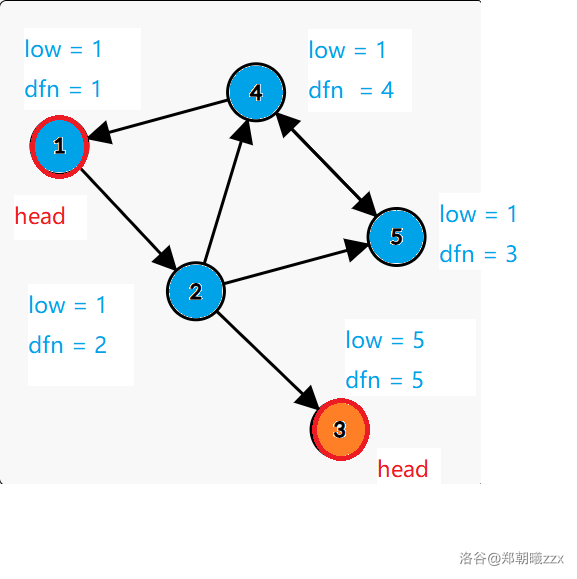
if (!vis[i]) {

**bfs(i, adj, vis); // 或dfs**

count++;

}

}

* **易错点**：有向图需用强连通分量算法

#### 3. 连通性

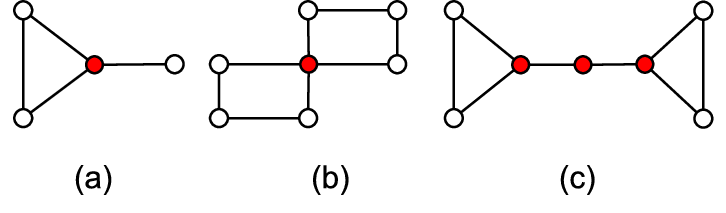
**（1）强连通分量（SCC）**

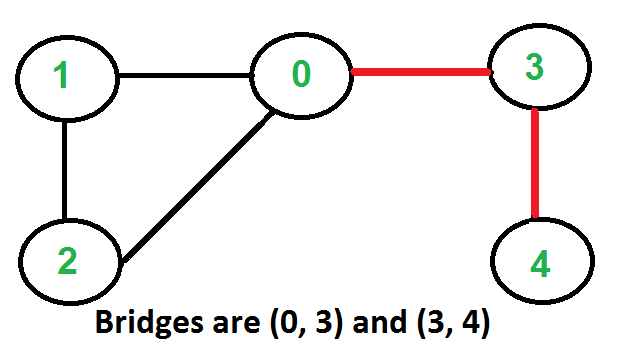
* **核心算法**：Tarjan（基于DFS栈和low-link值）
* **黄金法则**：SCC缩点后形成DAG
* **关键变量**：

vector<int> **dfn(n,-1), low(n,-1);** // 时间戳和追溯值

stack<int> st; // 存储当前SCC节点

* **易错点**：回溯时更新low[u]=min(low[u], dfn[v])而非low[v]

**（2）割点与桥**

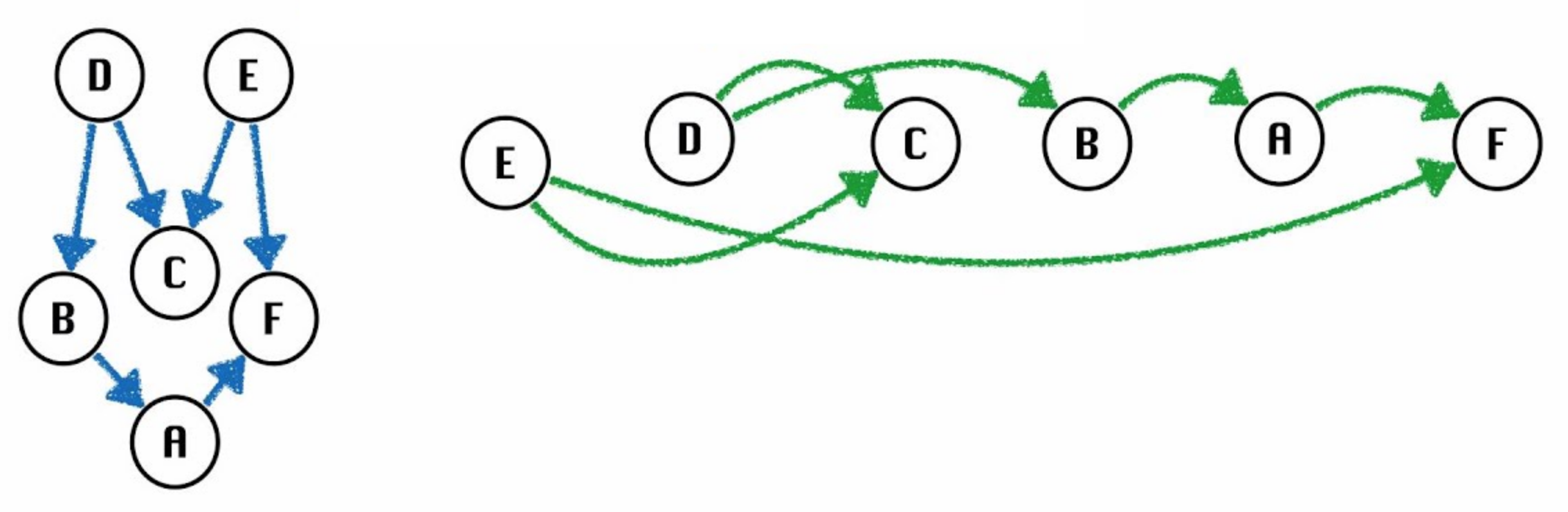
* **判定条件**：
  + **割点**：存在子节点v满足low[v] >= dfn[u]（根节点需两棵子树）
  + **桥**：low[v] > dfn[u]
* **黄金法则**：割点/桥检测基于DFS树

#### 4. 拓扑排序

**（1）DAG约束**

* **核心知识点**：仅适用于有向无环图，**起始点必为入度0节点**
* **模板代码**：

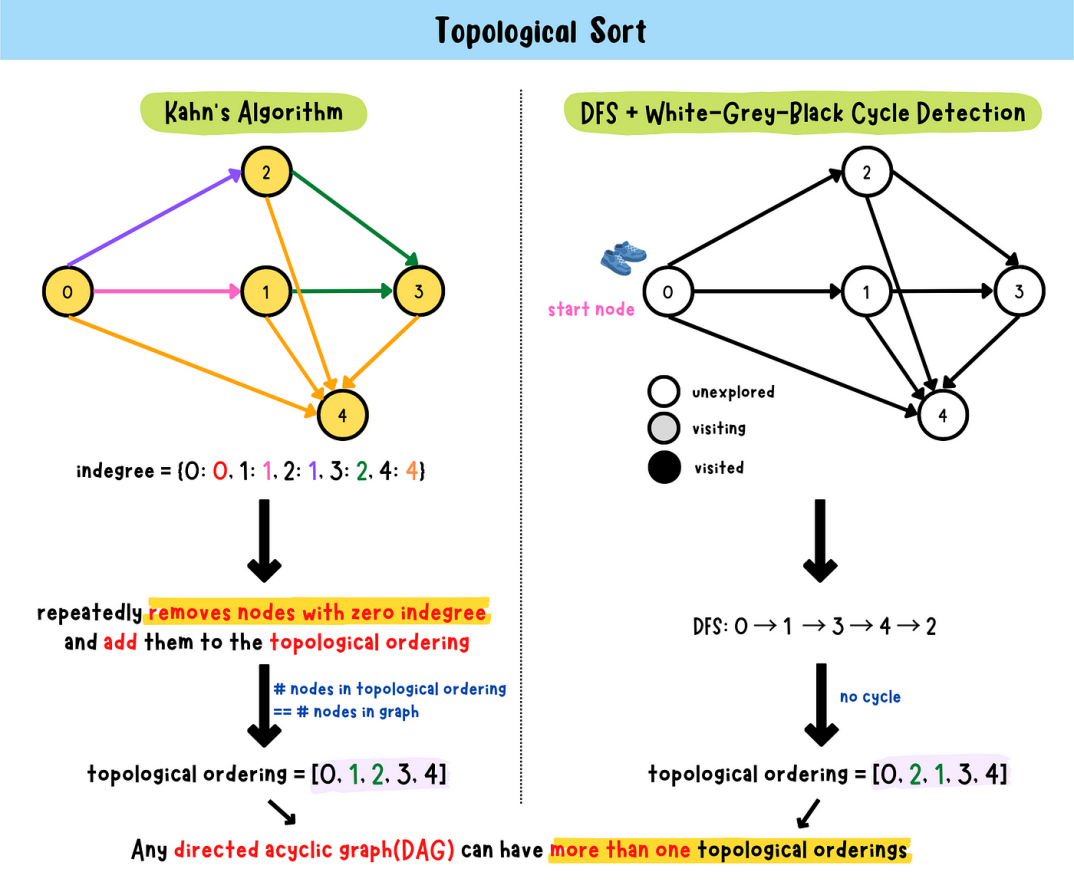
vector<int> topoSort(vector<vector<int>>& adj) {

 vector<int> indeg(adj.size(), 0);

queue<int> q;

// 计算入度

for (auto& neighbors : adj)

 for (int v : neighbors) **indeg[v]++;**

// 入度0节点入队

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

if (indeg[i]==0) **q.push(i);**

vector<int> res;

while (!q.empty()) {

int u = q.front(); q.pop();

res.push\_back(u);

for (int v : adj[u]) {

**if (--indeg[v] == 0) q.push(v);**

}

}

return res.size()==adj.size() ? res : vector<int>{}; // 检查环

}

* **易错点**：结果长度≠顶点数说明存在环
* **真题示例**：2010-18（起点必为入度0节点）

#### 5. 最短路径

**（1）Dijkstra算法**

* **核心限制**：仅处理非负权图
* **优先队列优化**：

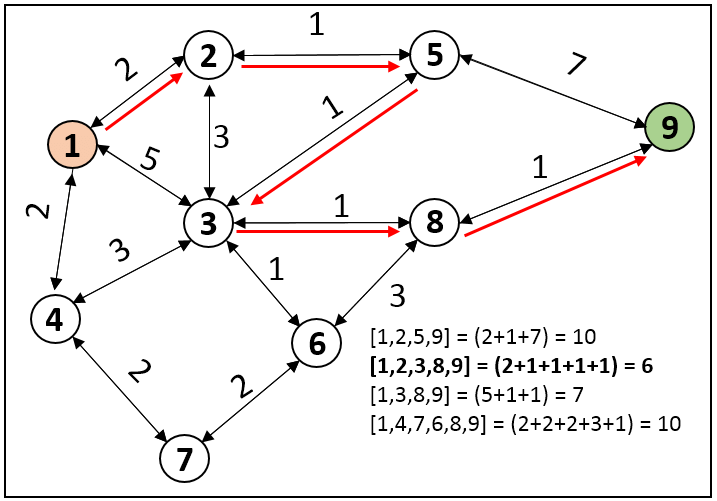
priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq; // <dist, node>

pq.emplace(0, start);//插入一个 :pair<int, int>(0, start)

**while (!pq.empty()) {**

auto [d, u] = pq.top(); pq.pop();

**if (d != dist[u]) continue; // 旧值跳过**

 for (auto [v, w] : adj[u]) {

if (dist[u] + w < dist[v]) {

**dist[v] = dist[u] + w;**

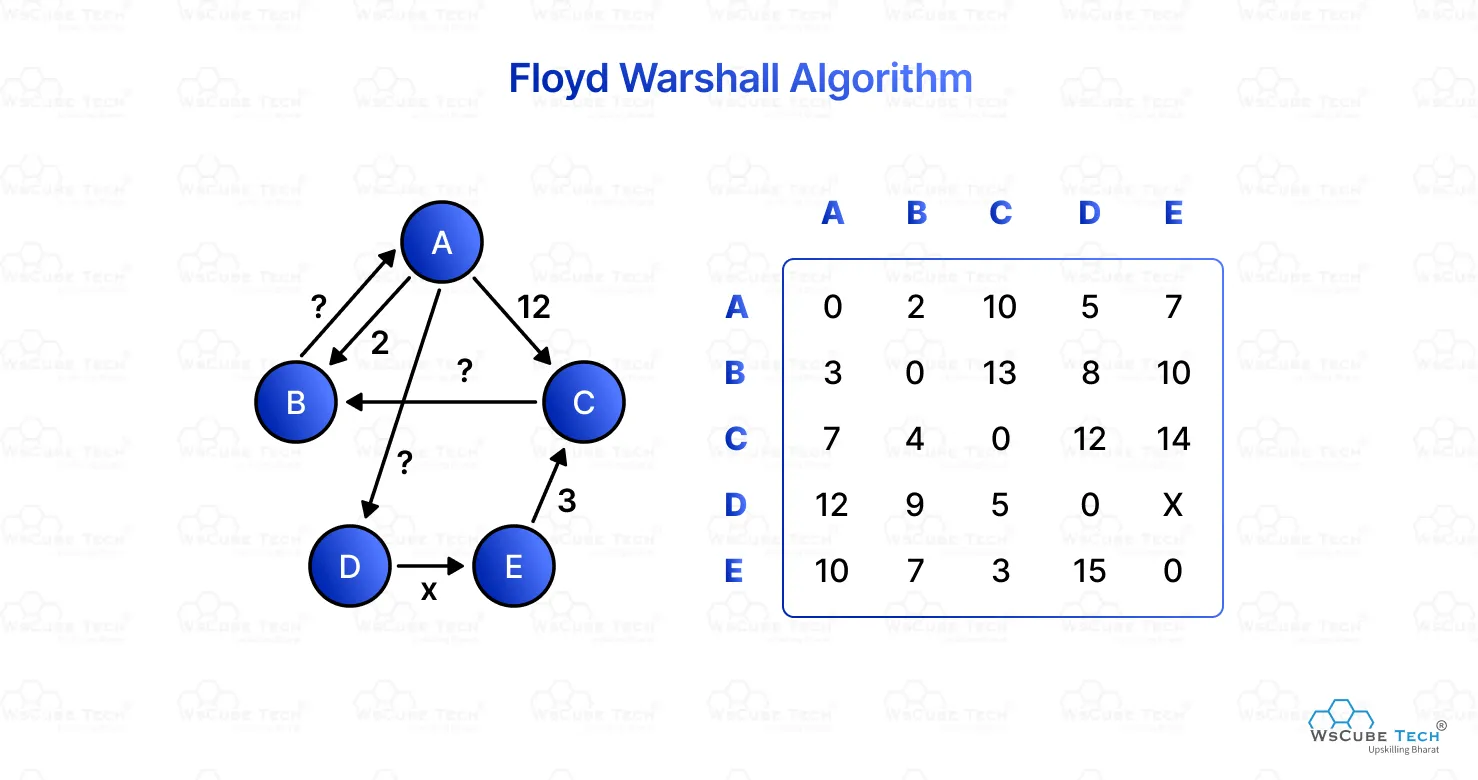
pq.emplace(dist[v], v);

}

}

}

**（2）Floyd算法**

* **核心思想**：动态规划三维压缩
* **关键代码**：

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

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

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

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

* **易错点**：k循环必须在外层

#### 6. 最小生成树

**（1）Kruskal算法**

* **核心流程**：边排序+并查集判环
* **并查集模板**：

vector<int> parent;

int find(int x) {

**return parent[x] == x ? x : parent[x] = find(parent[x]);**

}

void unite(int x, int y) {

**parent[find(x)] = find(y);**

}

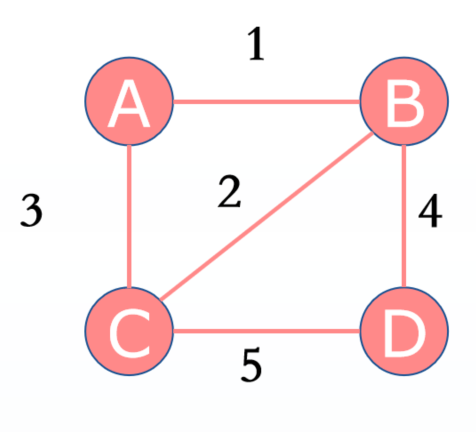
* **时间复杂度**：O(|E|log|E|)

**（2）Prim算法**

* **优先队列实现**：

priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;

pq.emplace(0, 0); // <weight, node>

while (!pq.empty()) {

auto [w, u] = pq.top(); pq.pop();

if (vis[u]) continue;

vis[u] = true;

total += w;

for (auto [v, w] : adj[u]) {

if (!vis[v]) pq.emplace(w, v);

}

}

#### 7. 树相关

**（1）树的性质**

* **黄金公式**：**节点数n，边数m满足 m = n-1**
* **真题示例**：2017-10（删边数 = m - n + 1）

**（2）二叉树重构**

* **核心算法**：**前序+中序**建树

TreeNode\* build(vector<int>& pre, int pL, int pR, vector<int>& in, int iL, int iR) {

if (pL > pR) return nullptr;

int rootVal = pre[pL];

int idx = find(in.begin()+iL, in.begin()+iR+1, rootVal) - in.begin();

int leftSize = idx - iL;

TreeNode\* root = new TreeNode(rootVal);

**root->left** = build(pre, pL+1, pL+leftSize, in, iL, idx-1);

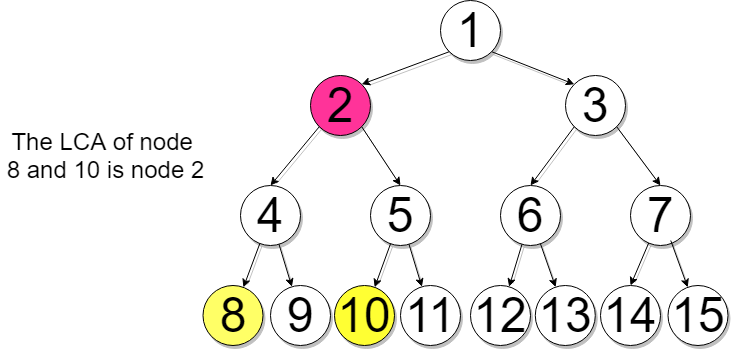
**root->right** = build(pre, pL+leftSize+1, pR, in, idx+1, iR);

return root;

}

**（3）LCA（最近公共祖先）**

// 预处理

void dfs(int u, int p, vector<vector<int>>& up, vector<int>& depth) {

depth[u] = **depth[p] + 1;**

up[u][0] = p;

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

up[u][i] = up[up[u][i-1]][i-1];

for (int v : adj[u])

if (v != p) dfs(v, u, up, depth);

}

// 查询

int lca(int a, int b, vector<vector<int>>& up, vector<int>& depth) {

if (depth[a] < depth[b]) swap(a,b);

int diff = depth[a] - depth[b];

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

if (diff & (1<<i)) a = up[a][i];

if (a == b) return a;

for (int i=LOG-1; i>=0; i--)

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

a=up[a][i], b=up[b][i];

return up[a][0];

}

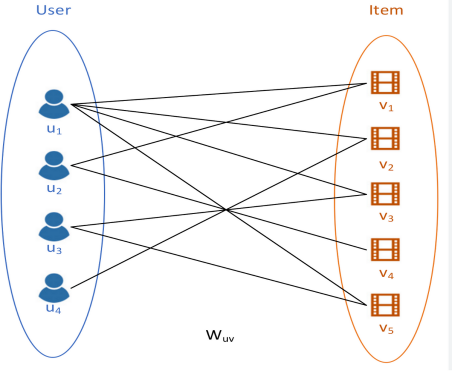
#### 8. 特殊图

**（1）二分图判定**

**染色法模板：**

bool isBipartite(vector<vector<int>>& adj) {

vector<int> color(adj.size(), -1);

 queue<int> q;

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

if (color[i] != -1) continue;

q.push(i);

color[i] = 0;

while (!q.empty()) {

int u = q.front(); q.pop();

for (int v : adj[u]) {

**if (color[v] == -1) {**

**color[v] = color[u] ^ 1;**

q.push(v);

} else if (color[v] == color[u]) {

return false;

}

}

}

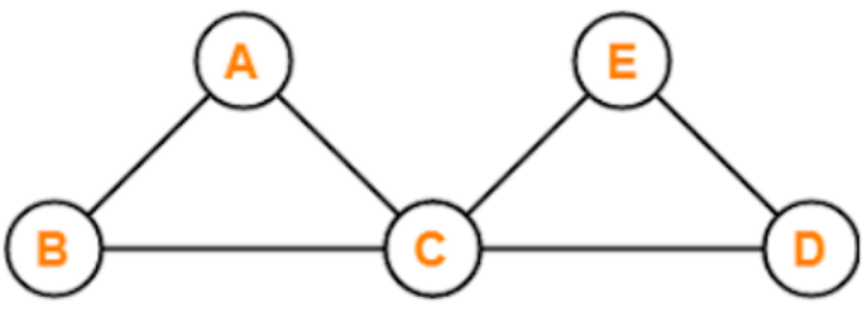
}

return true;

}

**（2）欧拉图性质**

* **判定条件**：
  + 无向图欧拉回路：**全偶度**
  + 无向图欧拉路径：恰两个奇度点
* **黄金法则**：Hierholzer算法求欧拉路径



### 真题强化

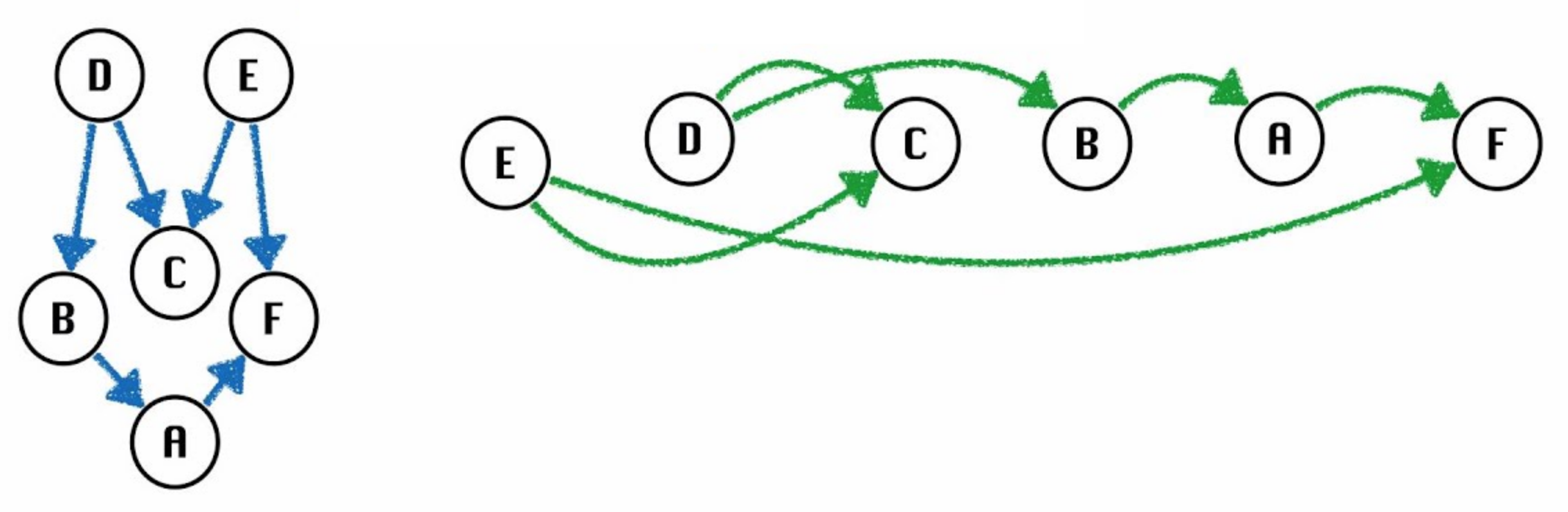
#### 1. 2010年第18题（拓扑排序）

关于拓扑排序，下面说法正确的是（ ）。

A. 所有连通的有向图都可以实现拓扑排序

B. 对同一个图而言，拓扑排序的结果是唯一的

C. 拓扑排序中入度为 0 的结点总会排在入度大于 0 的结点的前面

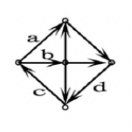
D. 拓扑排序结果序列中的第一个结点一定是入度为 0 的点

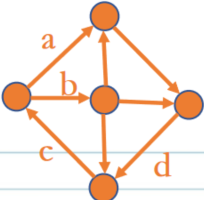
**答案： D**

**解析：**

* **拓扑排序前提：有向无环图（DAG）**。
* 选项 A 错误：连通有向图若存在环（如环形图），无法拓扑排序。
* 结果唯一性：
* 选项 B 错误：**拓扑排序结果不唯一**（如多入度为0结点时顺序可变）。
* 结点性质：
* 选项 C 错误：拓扑排序按入度0顺序输出，但后续结点入度也可能==0。
* 选项 D 正确：拓扑排序起始点必须是入度为0的结点（无依赖项）。

#### 2. 2011年第19题（强连通图）

对一个有向图而言，如果每个节点都存在到达其他任何节点的路径，那么就称它是强连通的。例如，下图就是一个强连通图。事实上，在删掉边（ ）后，它依然是强连通的。

A. a

B. b

C. c

D. d

**答案： A**

**解析：**

**强连通图：任意两点间存在双向路径。**

删边分析（假设图结构）：

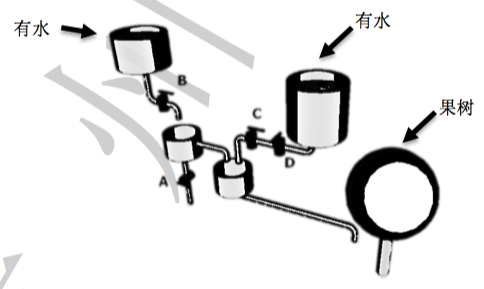
删除边 a 后：仍存在其他路径连通所有结点（如通过 b、c、d 构成替代路径）。

删除 b、c、d 中任意一条会破坏关键路径，导致无法到达某些结点。

结论：边 a 是冗余边，删除后仍保持强连通性。

#### 3. 2016年第17题（图的连通性）

下图表示一个果园灌溉系统，有 A,B,C,D 四个阀门，每个阀门可以打开或关上，所有管道粗细相同，以下设置阀门的方法中，可以让果树浇上水的是（）。

A. B 打开，其他都关上

B. AB 都打开，CD 都关上

C. A 打开，其他都关上

D. D 打开，其他都关上

**答案： A**

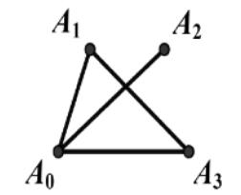
**解析：**

问题本质：判断阀门组合是否能形成从水源到果树的连通路径。

选项分析：

A（开B）：水通过 B 直接流向果树（路径：水源 → B → 果树）。

#### 4. 2013年第12题（深度优先遍历）

以 **A0** 作为起点，对下面的无向图进行**深度优先遍历**时，遍历顺序不可能是（ ）。

A. A0, A1, A2, A3

B. A0, A1, A3, A2

C. A0, A2, A1, A3

D. A0, A3, A1, A2

**答案： A**

**解析：**

DFS 特性：优先深入访问相邻未访问结点。

#### 5 2022年第20题（洪水填充算法）

（洪水填充）

现有用字符标记像素颜色的 8×8 图像。颜色填充的操作描述如下：给定起始像素的位置待填充的颜色，将起始像素和所有可达的像素（可达的定义：经过一次或多次的向上、下、左、右四个方向移动所能到达且终点和路径上所有像素的颜色都与起始像素颜色相同），替换为给定的颜色。

试补全程序。

#include<bits/stdc++.h>

using namespace std;

const int ROWS = 8;

const int COLS = 8;

**struct Point** {

 int r, c;

Point(int r, int c): r(r), c(c) {}

};

bool **is\_valid**(char image[ROWS][COLS], Point pt, int prev\_color, int new\_color) {

int r = pt.r;

int c = pt.c;

return (0 <= r && r < ROWS && 0 <= c && c < COLS && ① && image[r][c] != new\_color);

}

void **flood\_fill**(char image[ROWS][COLS], Point cur, int new\_color) {

queue<Point> queue;

queue.push(cur);

int prev\_color = image[cur.r][cur.c];

②;

 while (!queue.empty()) {

Point pt = queue.front();

queue.pop();

Point points[4] = {③, Point(pt.r - 1, pt.c), Point(pt.r, pt.c + 1), Point(pt.r, pt.c - 1)};

for (auto p : points) {

if (is\_valid(image, p, prev\_color, new\_color)) {

④;

⑤;

}

}

}

}

1.①处应填（ ）

A. image[r][c] == prev\_color

B. image[r][c] != prev\_color

C. image[r][c] == new\_color

D. image[r][c] != new\_color

2.②处应填（ ）

A. image[cur.r+1][cur.c] = new\_color

B. image[cur.r][cur.c] = new\_color

C. image[cur.r][cur.c+1] = new\_color

D. image[cur.r][cur.c] = prev\_color

3.③处应填（ ）

A. Point(pt.r, pt.c)

B. Point(pt.r, pt.c+1)

C. Point(pt.r+1, pt.c)

D. Point(pt.r+1, pt.c+1)

4.④处应填（ ）

A. prev\_color = image[p.r][p.c]

B. new\_color = image[p.r][p.c]

C. image[p.r][p.c] = prev\_color

D. image[p.r][p.c] = new\_color

5.⑤处应填（ ）

A. queue.push(p)

B. queue. push (pt)

C. queue.push(cur)

D. queue. push(Point (ROWS, COLS))

**答案：**

**① A**

**② B**

**③ C**

**④ D**

**⑤ A**

**解析：**

①：image[r][c] == **prev\_color**

验证像素需与起始颜色相同。

②：image[cur.r][cur.c] = **new\_color**

起始点着色为新颜色。

③：Point(pt.r+1, pt.c)

定义下方向相邻点（**上、下、右、左**）。

④：image[p.r][p.c] = new\_color

将有效相邻点着色为**新颜色**。

⑤：queue.push(p)

将着色点**加入队列**继续扩散。

算法逻辑：BFS 遍历相邻同色像素并替换颜色。

#### 6. 2016年第15题（图的性质）

设简单无向图 G 有 16 条边且每个顶点的度数都是 2，则图 G 有( )个顶点。

A. 10

B. 12

C. 8

D. 16

**答案： D**

**解析：**

**思路一：**

握手定理：无向图所有顶点度数之和 = 2 × 边数。

计算：设顶点数为 n，则 2n = 2 \* 16，故 n = 16。

图结构：每个顶点度数为 2 → 图是环状结构（边数 = 顶点数）。

**思路二：**

1

3

2

4

简单无向图，即每条边连接2个顶点，

**每个顶点度为2，即每个顶点有2条边相连**

所以边数和顶点数相同

#### 7. 2017年第10题（图与树）

设 G 是有 n 个结点、m 条边 (n≤m) 的连通图，必须删去 G 的（ ）条边，才能使得 G 变成一棵树。

A. m−n+1

B. m−n

C. m+n+1

D. n−m+1

**答案： A**

**解析：**

树的性质：n 个结点的树有 n-1 条边。

删边计算：连通图需删除多余边数 = m - (n-1) = m - n + 1。

示例：**n=3, m=3（三角形）→ 删 3-3+1=1** 条边得到树。

#### 2009年第 18 题（图）

已知 n 个顶点的有向图，若该图是强连通的（从所有顶点都存在路径到达其他顶点），则该图中**最少**有多少条有向边？

A. n

1

3

2

4

B. n+1

C. n−1

D. n(n−1)

**正确答案： A**

**解析：**

对于 n 个顶点的强连通图，最少有 n 条边，**每个顶点有至少一条出边和一条入边，形成一个环**。

#### 9 2011年第 5 题（图）

无向完全图是图中每对顶点之间都恰好有一条边的简单图。已知无向完全图 G 有 7 个顶点，则它共有（ ）条边。

A. 7

1

3

2

4

B. 21

C. 42

D. 49

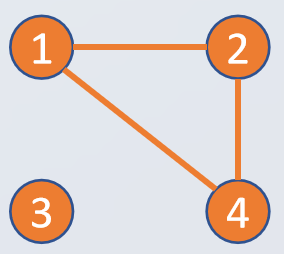
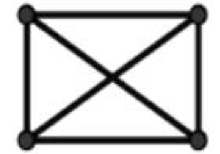
**正确答案： B**

**解析：**

无向完全图的边数为 C(n,2)，即 **n(n−1)/2**。因此，7×6=21。

#### 10 2013年第 10 题（图）

在一个无向图中，如果任意两点之间都存在路径相连，则称其为连通图。下图是一个有 4 个顶点、6 条边的连通图。若要使它不再是连通图，**至少**要删去其中的（ ）条边。

A. 1

B. 2

C. 3

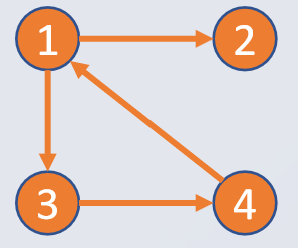
D. 4

**正确答案： C**

**解析**

非连通图也就是有孤立的顶点，每个点都有3条边，删掉这3条边就会出现孤立的顶点，出现非连通图。

#### 11 2014年第 17 题（图）

有向图中每个顶点的度等于该顶点的( )。

A. 入度

B. 出度

C. 入度和出度之和

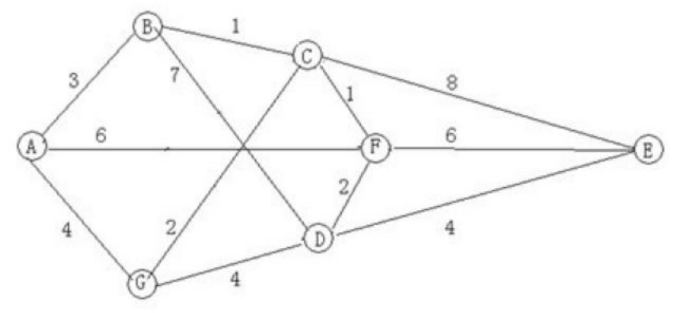
D. 入度和出度之差

**正确答案： C**

**解析** 有向图中，**每个顶点的度等于该顶点的入度和出度之和**，如顶点1的出度为2，入度为1，顶点1的度为3

#### 12 2014年第 22 题（图）

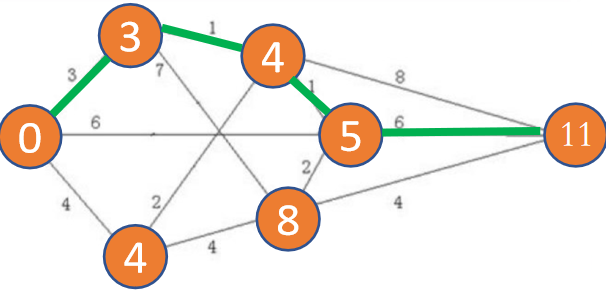
如图所示，图中每条边上的数字表示该边的长度，则从A到E的最短距离是\_\_。



**正确答案： 11**

**解析**

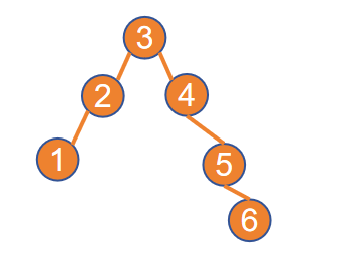
假设图中各点之间的距离如下：



可以使用 Dijkstra 算法找到从 A 到 E 的最短路径，计算结果为 11。

#### 13 2015年第 12 题（图）

6 个顶点的连通图的最小生成**树**，其边数为( )。

 A. 6

B. 5

C. 7

D. 4

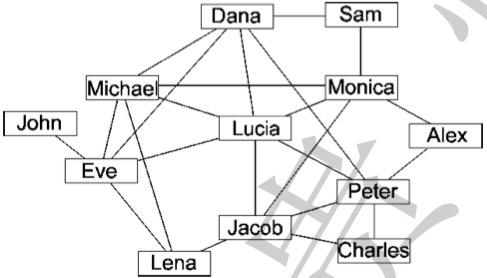
**正确答案： B**

**解析**

一个有 n 个顶点的连通图的最小生成树有 n-1 条边。所以，6 个顶点的连通图的最小生成树有 5 条边。

#### 14 2016年第 18 题（图）

Lucia 和她的朋友以及朋友的朋友都在某社交网站上注册了账号。下图是他们之间的关系图，两个人之间有边相连代表这两个人是朋友，没有边相连代表不是朋友。这个社交网站的规则是：如果某人 A 向他（她）的朋友 B 分享了某张照片，那么 B 就可以对该照片进行评论；如果 B 评论了该照片，那么他（她）的所有朋友都可以看见这个评论以及被评论的照片，但是不能对该照片进行评论（除非 A 也向他（她）分享了该照片）。现在 Lucia 已经上传了一张照片，但是她不想让 Jacob 看见这张照片，那么她可以向以下朋友（ ）分享该照片。



A. Dana, Michael, Eve

B. Dana, Eve, Monica

C. Michael, Eve, Jacob

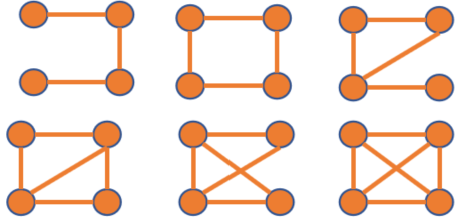
D. Michael, Peter, Monica

**正确答案： A**

**解析：** Lucia 不想让 Jacob 看见照片，那么 Jacob 必须与所有直接分享的朋友没有直接的朋友关系。选项 A 中的 Dana, Michael, Eve 满足这个条件。

#### 15 2018年第 11 题（图）

由四个没有区别的点构成的简单无向连通图的个数是（ ）。

A. 6

B. 7

C. 8

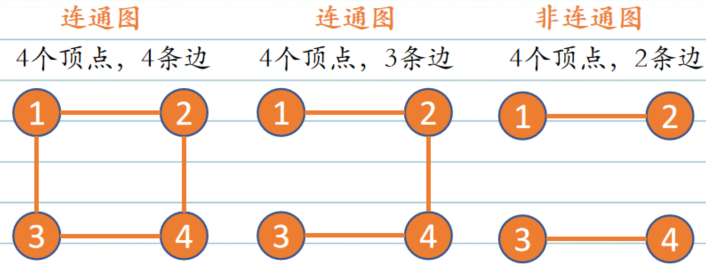
D. 9

正确答案： A

解析： 由四个点构成的无向连通图的个数可以通过组合计算得出，共有 6 种。

#### 16 2020年第 8 题（图）

有 10 个顶点的无向图至少应该有（ ）条边才能确保是一个连通图。

A. 9

B. 10

C. 11

D. 12

**正确答案： A**

**解析： 一个有 n 个顶点的无向图至少要有 n-1 条边**才能确保连通。

#### 17 2021年第 6 题（图）

对于有 n 个顶点、m 条边的无向连通图 (m > n)，需要删掉（ ）条边才能使其成为一棵树。

A. n-1

B. m-n

C. m-n-1

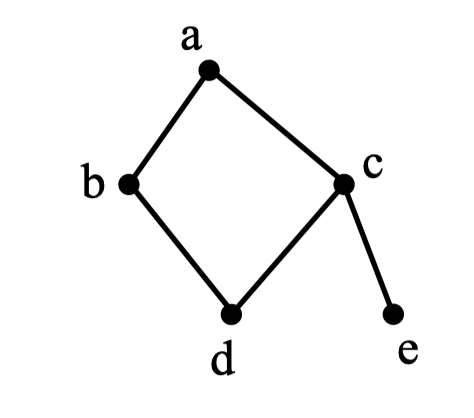
D. m-n+1

正确答案： D

解析： 连通图变成树的条件是删去 m - n + 1 条边。

#### 18 2021年第 14 题（图）

以 a 为起点，对下边的无向图进行深度优先遍历，则 b, c, d, e 四个点中有可能作为最后一个遍历到的点的个数为（ ）。

A. 1

B. 2

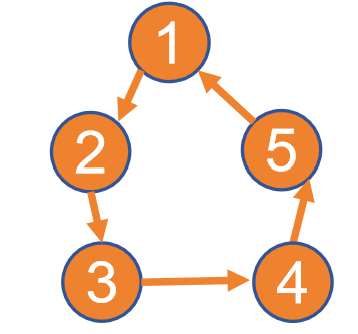
C. 3

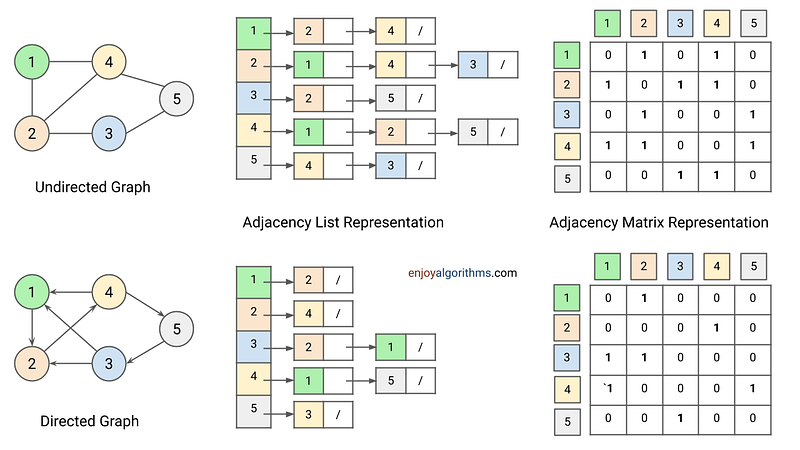
D. 4

**正确答案： B**

**解析：** 根据深度优先遍历的特点，最后一个遍历到的点取决于路径选择，有两种可能。abdce acedb acdbe

#### 19 2022年第 9 题（图）

考虑由 N 个顶点构成的有向连通图，采用邻接矩阵的数据结构表示时，该矩阵中**至少**存在（ ）个**非零元素**。

A. N-1

B. N

C. N+1

D. N²

正确答案： B

解析： 一个有向连通图至少有 N 条边，因此邻接矩阵中至少有 N 个非零元素。

#### 20 2023年第 12 题（图）

考虑一个有向无环图，该图包含 4 条有向边：(1,2),(1,3),(2,4) 和 (3,4)。以下哪个选项是这个有向无环图的一个有效的拓扑排序?

A. 4,2,3,1

B. 1,2,3,4

C. 1,2,4,3

D. 2,1,3,4

**正确答案： B**

**解析：** 拓扑排序从起始节点开始，结果为 1,2,3,4。

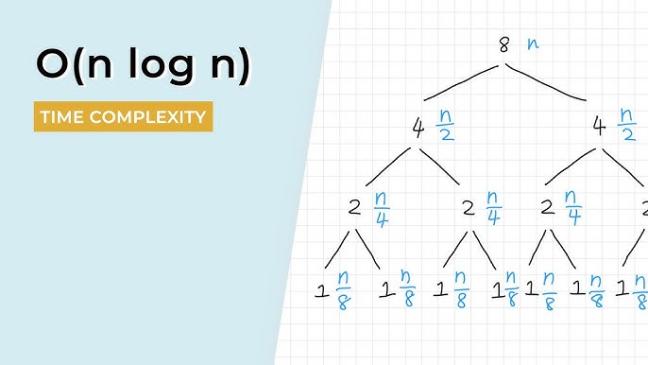
# 八、排序算法

### 考点分析

|  |  |  |  |
| --- | --- | --- | --- |
| **考点类别** | **重点考点** | **考察频率** | **难度** |
| **时间复杂度分析** | 比较排序下限（Ω(n log n)） | 30% | ★★★ |
| 平均/最坏时间复杂度 | 40% | ★★ |
| 非比较排序时间复杂度 | 20% | ★★★ |
| **稳定性与特性** | 算法稳定性判定 | 25% | ★★ |
| 逆序对计算与应用 | 20% | ★★★ |
| 原地排序特性 | 15% | ★★ |
| **特定算法实现** | 快速排序分区操作 | 30% | ★★★ |
| 归并排序合并比较次数 | 25% | ★★★ |
| 堆排序建堆与调整 | 20% | ★★★ |
| 计数排序桶分配 | 15% | ★★★ |
| **优化策略** | 同时找最大最小值的比较优化 | 10% | ★★★★ |
| 快速排序避免最坏情况 | 20% | ★★★ |
| **特殊应用场景** | 多关键字排序 | 15% | ★★★★ |
| 外部排序（多路归并） | 10% | ★★★★ |
| 拓扑排序性质 | 15% | ★★ |
| **排序变种** | 快速选择算法 | 20% | ★★★★ |
| 桶排序的适用条件 | 10% | ★★★ |
| **证明类问题** | 比较排序下限证明 | 5% | ★★★★★ |
| 特定问题下界分析 | 5% | ★★★★★ |

### 核心知识点

#### 1. 时间复杂度分析 - 比较排序下限

* **核心知识点：**
  + 任何基于元素比较的排序算法，其时间复杂度下界为 **Ω(n log n)**，由决策树模型证明。
  + n 个元素有 n! 种排列，决策树叶子节点数 ≥ n!，树高 ≥ log₂(n!) ≈ n log₂n。
* **黄金法则：**
  + 比较排序无法突破 Ω(n log n) 的下界
  + 非比较排序（计数/基数排序）可突破该下界
* **高频公式：**

// 决策树最小高度计算

double min\_height = log2(factorial(n)); // n! 的以2为底对数

* **易错点：**
  + 混淆平均/最坏情况与理论下界
  + 误认为桶排序能突破比较排序下界（实际依赖非比较操作）

#### 2. 平均/最坏时间复杂度

* **核心知识点：**
  + **快速排序**：平均 O(n log n)，最坏 O(n²)（有序数组+固定基准）
  + **归并排序**：稳定 O(n log n)
  + **堆排序**：不稳定 O(n log n)



* **黄金法则：**
  + 快排随机化基准可避免最坏情况
  + 数据基本有序时，插入排序接近 O(n)
* **高频公式：**

// 快速排序随机基准选择

int pivot = arr[left + rand() % (right - left + 1)];

* **易错点：**
  + 忽略快排最坏情况的空间复杂度（递归栈 O(n)）
  + 混淆堆排序建堆复杂度（O(n)）与整体复杂度（O(n log n)）

#### 3. 非比较排序时间复杂度

* **核心知识点：**
  + **计数排序**：O(n + k)，k 为值域范围
  + **基数排序**：O(d·(n + r))，d 为位数，r 为基数
* **黄金法则：**
  + 计数排序要求数据范围有限
  + 基数排序必须从**低位到高位**稳定排序
* **高频公式：**

// 计数排序核心步骤

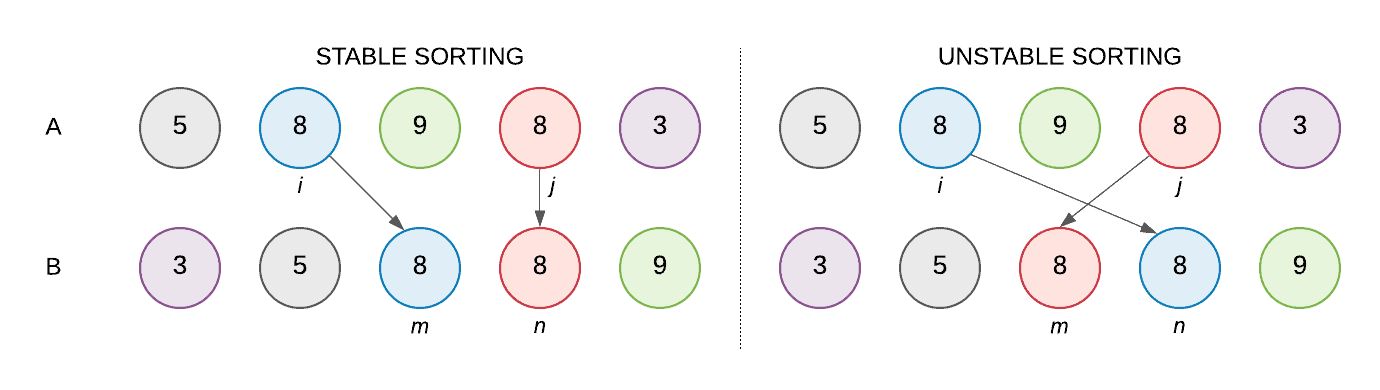
for(int i=0; i<n; ++i) ++cnt[arr[i]]; // 频率统计

for(int i=1; i<=max\_val; ++i) cnt[i] += cnt[i-1]; // 前缀和

for(int i=n-1; i>=0; --i) output[--cnt[arr[i]]] = arr[i]; // 倒序放置

* **易错点：**
  + 未初始化计数数组
  + 基数排序未使用稳定子排序导致错误

#### 4. 算法稳定性判定

* **核心知识点：**
  + **稳定**：归并、插入、冒泡、计数、基数
  + **不稳定**：快排（分区交换）、堆排序（堆顶交换）、选择排序
* **黄金法则：**
  + 多关键字排序必须使用稳定排序
  + 交换相邻元素的排序是稳定的
* **高频公式：**

// 归并排序稳定合并

while(i<=mid && j<=right){

if(a[i] <= a[j]) tmp[k++] = a[i++]; // 等号保证稳定性

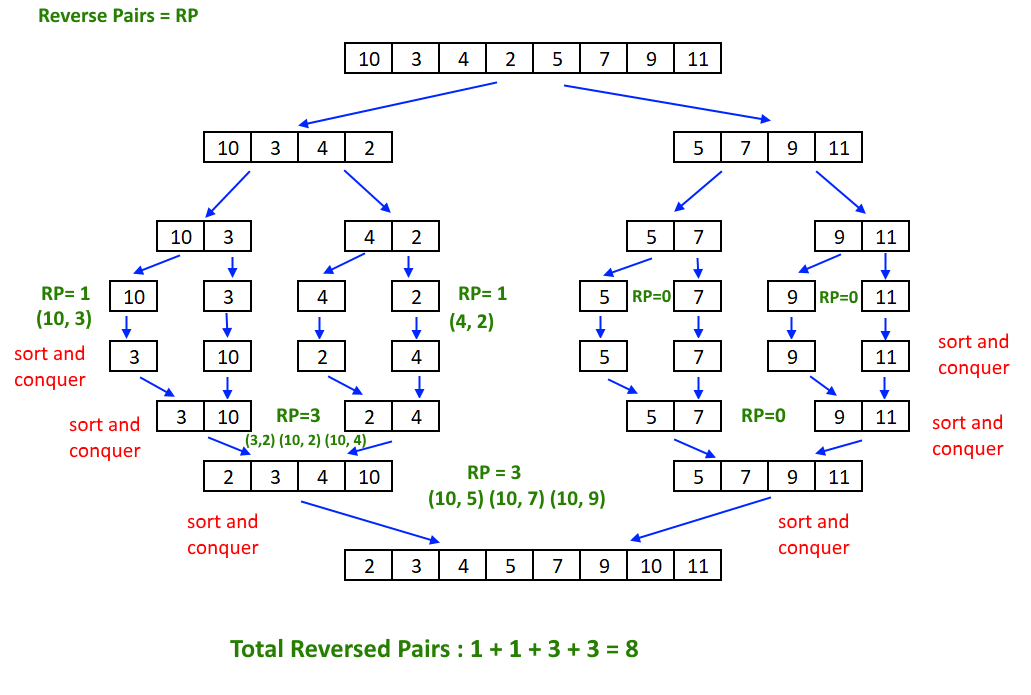
else tmp[k++] = a[j++];

}

* **易错点：**
  + 误认为所有 O(n log n) 排序都稳定
  + 快排分区时等号处理不当导致不稳定

#### 5. 逆序对计算与应用

* **核心知识点：**
  + **逆序对：i < j 且 a[i] > a[j]**
  + 归并排序合并时计数：当右元素小于左元素时，左段剩余元素均构成逆序对



* **黄金法则：**
  + 归并排序合并过程天然适合计数逆序对
  + 暴力枚举 O(n²)，归并优化至 **O(n log n)**
* **高频公式：**

// 归并过程中逆序对计数

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

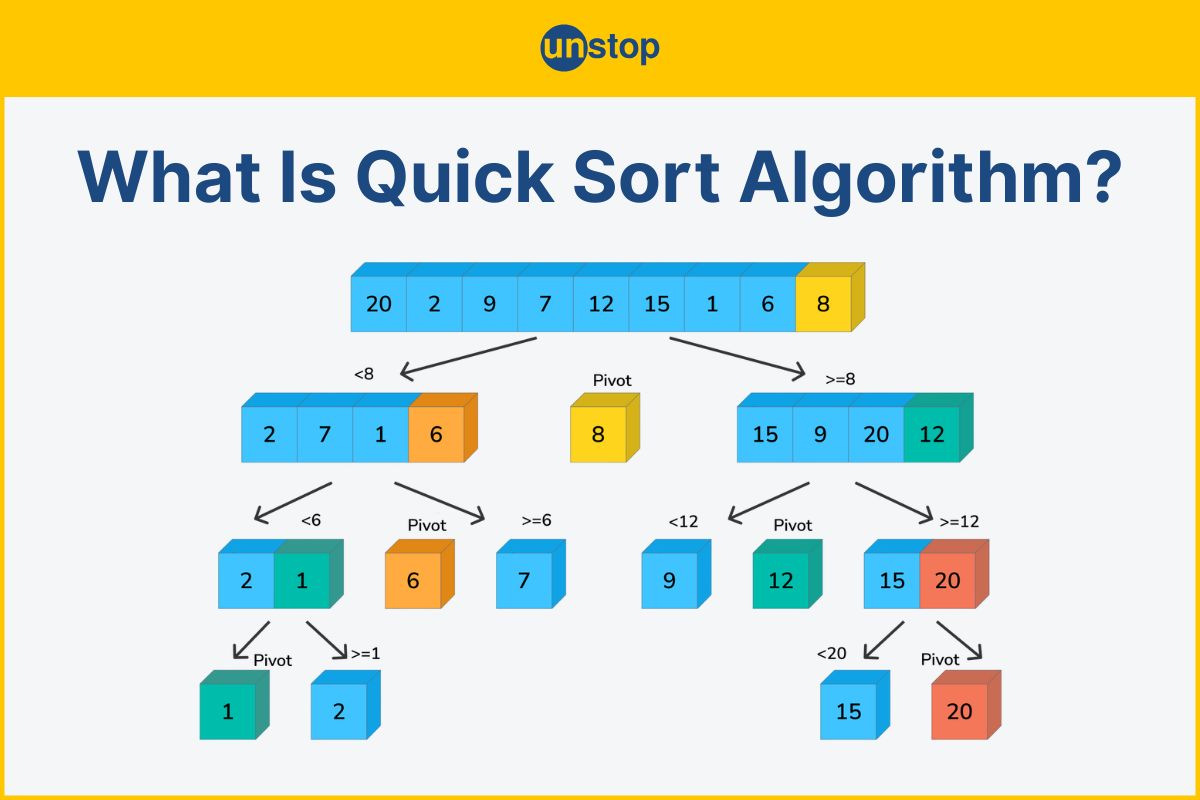
cnt += mid - i + 1; // 核心公式！

tmp[k++] = a[j++];

}

* **易错点：**
  + 计数时忽略左段剩余元素
  + 未处理相等情况导致多计/漏计

#### 快速排序分区操作



* **核心知识点：**
  + Hoare 分区：左右指针向中间扫描
  + Lomuto 分区：单指针维护小值区
* **黄金法则：**
  + **随机基准**避免有序数组的最坏情况
  + 分区后基准位置确定递归边界
* **高频公式：**

// Hoare 分区模板

int partition(vector<int>& arr, int l, int r) {

i**nt pivot = arr[l + rand()%(r-l+1)]; // 随机基准**

int i = l-1, j = r+1;

while(true) {

do ++i; while(arr[i] < pivot);

do --j; while(arr[j] > pivot);

if(i >= j) return j;

swap(arr[i], arr[j]);

}

}

* **易错点：**
  + 死循环（指针移动条件错误）
  + 递归边界错误（应使用 [l, j] 和 [j+1, r]）

#### 7. 同时找最大最小值的优化

* **传统方法** 
  + 遍历数组，独立寻找最大值和最小值
  + 每个元素需与当前最大值和最小值各比较一次（除第一个元素）
  + 比较次数: **2n−2 次**（例如：n=5 时需 8 次比较）
* **优化策略**
  + 分组处理: 将元素两两分组（Pairs）
    - 组内比较：每组比较 1 次，分出局部较小值和较大值
    - 全局比较：用局部较小值更新全局最小值，局部较大值更新全局最大值
  + 比较次数:
    - 偶数n： **3n/2 − 2** 次（例如：n=6 时仅需 7 次）
    - 奇数 n: **3(n−1)/2**次（例如：n=5 时仅需 6 次）
* **黄金法则** 
  + **分组减少比较次数**
    - 传统方法每元素需 2 次比较
    - 分组后每对元素仅需 3 次比较（1 次组内 + 2 次全局），优于独立处理的 4 次
  + **奇数元素单独处理**
    - 奇数数组长度时，将首元素作为初始最小值和最大值
    - 剩余偶数个元素正常分组
* **高频公式：**

// 最优比较算法实现

pair<int, int> findMinMax(vector<int>& arr) {

int n = arr.size();

if(n == 1) return {arr[0], arr[0]};

int min\_val, max\_val;

**int start = (n % 2 == 1) ? 1 : 0; // 处理奇偶性**

if(n % 2 == 1) min\_val = max\_val = arr[0];

else {

min\_val = min(arr[0], arr[1]);

max\_val = max(arr[0], arr[1]);

start = 2;

}

for(int i=start; i<n; i+=2) {

if(arr[i] < arr[i+1]) {

min\_val = min(min\_val, arr[i]);

max\_val = max(max\_val, arr[i+1]);

} else {

min\_val = min(min\_val, arr[i+1]);

max\_val = max(max\_val, arr[i]);

}

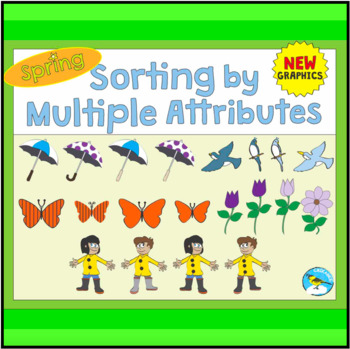
}

return {min\_val, max\_val};

}

* **易错点：**
  + 未处理数组奇偶性
  + 分组比较次数计算错误（每组3次比较？实际每组1次比较）

#### 8. 多关键字排序

* **核心知识点：**
  + **先按次关键字排序，再按主关键字稳定排序**
  + 计数排序串联实现双关键字排序
* **黄金法则：**
  + 从最后一个关键字开始排序
  + 必须使用稳定排序算法
* **高频公式：**

// 双关键字计数排序核心步骤

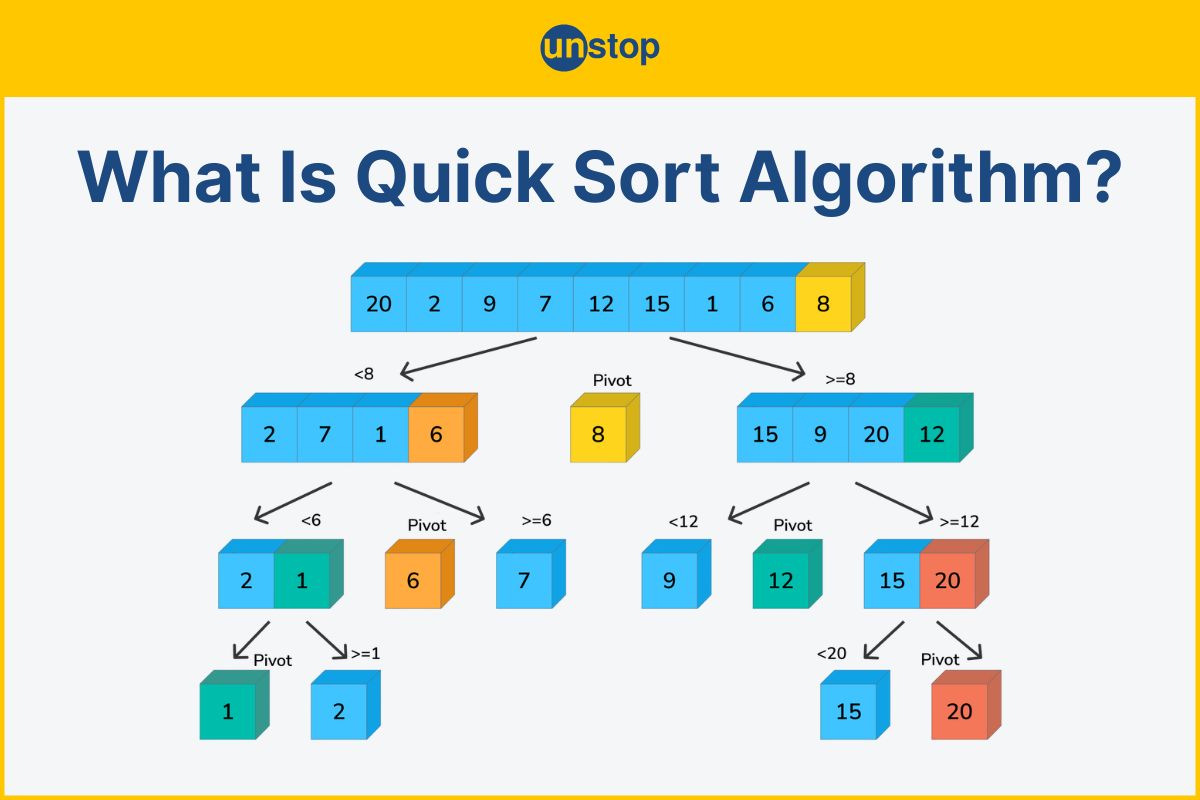
// 1. 按第二关键字b排序

countingsort\_by\_key(b, ord); // ord存储索引

// 2. 按第一关键字a排序

countingsort\_by\_key(a, res); // 使用ord作为输入

// 3. 输出：a[res[i]], b[res[i]]

* **易错点：**
  + 关键字顺序颠倒
  + 未从后向前放置元素破坏稳定性

#### 9. 快速选择算法

* **核心知识点：**
  + 基于快排分区思想
  + **平均 O(n)，最坏 O(n²)（**可随机化避免）
* **黄金法则：**
  + 根据基准位置与k的关系选择递归区间
  + 无需完全排序整个数组
* **高频公式：**

int quickSelect(vector<int>& arr, int l, int r, int k) {

if(l == r) return arr[l];

int p = partition(arr, l, r);

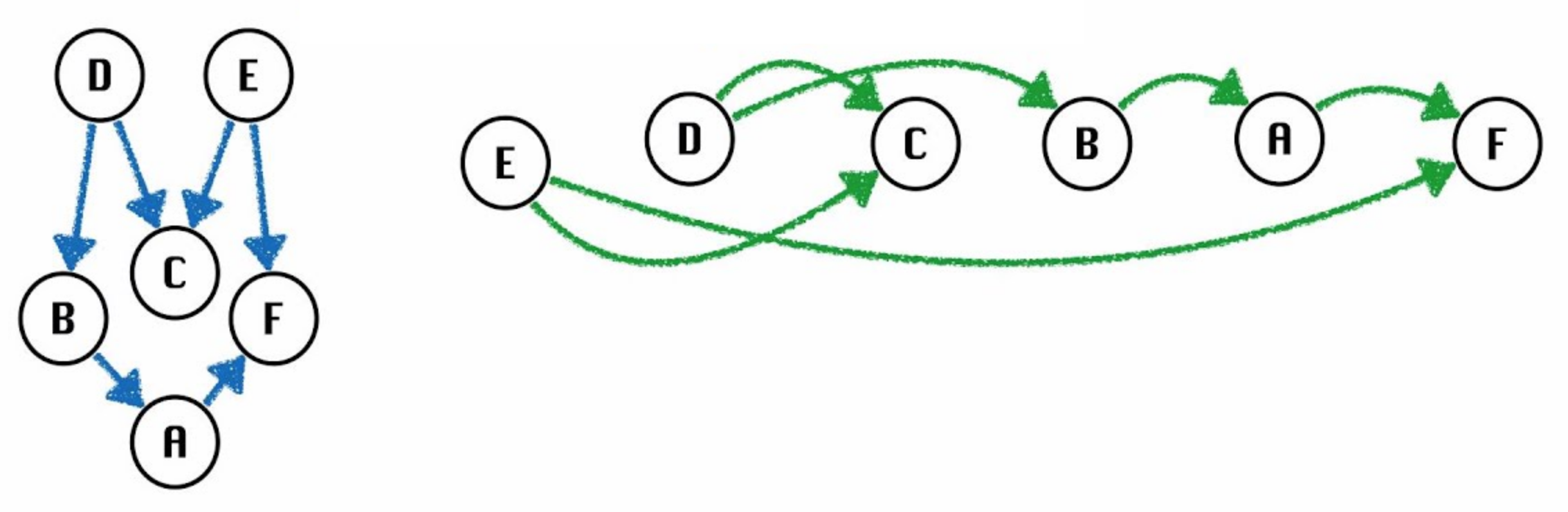
int idx = p - l + 1; // 当前基准是第几小

if(k == idx) return arr[p];

if(k < idx) return quickSelect(arr, l, p-1, k);

return quickSelect(arr, p+1, r, k-idx);

}

* **易错点：**
  + 递归区间选择错误
  + 未处理基准位置与k的偏移量

#### IMG_25610. 拓扑排序性质

* **核心知识点：**
  + 仅适用于**有向无环图（DAG）**
  + 起始点必须是**入度为0**的节点
* **黄金法则：**
  + Kahn算法：不断移除入度为0的节点
  + DFS逆后序也是有效拓扑序
* **高频公式：**

// Kahn算法模板

vector<int> topoSort(vector<vector<int>>& graph) {

vector<int> indegree(n, 0), res;

queue<int> q;

for(auto& edges : graph)

for(int v : edges) indegree[v]++;

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

if(indegree[i]==0) q.push(i);

while(!q.empty()) {

int u = q.front(); q.pop();

res.push\_back(u);

for(int v : graph[u])

if(--indegree[v] == 0) q.push(v);

}

return res.size()==n ? res : vector<int>();

}

* **易错点：**
  + 未检测环（结果长度不等于n）
  + 忽略多个入度为0节点的处理顺序

#### 11. 堆排序建堆与调整

* **核心知识点：**
  + 建堆：从最后一个非叶子节点开始下沉
  + 删除堆顶：交换首尾 → 堆顶下沉
* **黄金法则：**
  + 大顶堆用于升序排序
  + 父子节点索引关系：left=2\*i+1, right=2\*i+2
* **高频公式：**

void heapify(vector<int>& arr, int n, int i) {

int largest = i, l = 2\*i+1, r = 2\*i+2;

if(l < n && arr[l] > arr[largest]) largest = l;

if(r < n && arr[r] > arr[largest]) largest = r;

if(largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, n, largest); // 递归下沉

}

}

// 建堆

for(int i = n/2-1; i>=0; --i)

heapify(arr, n, i);

* **易错点：**
  + 建堆起始位置错误（应为 n/2-1）
  + 堆大小未随删除递减

#### 12. 外部排序与多路归并

* **核心知识点：**
  + 步骤：分块 → 块内排序 → 多路归并
  + 败者树优化：减少比较次数至 O(log k)
* **黄金法则：**
  + 归并路数 k 受内存缓冲区大小限制
  + 最佳归并树优化I/O次数
* **高频公式：**

// 多路归并伪代码

while(所有块未空){

从k个缓冲区选出最小元素

输出到结果文件

补充该元素对应块的下一元素

}

* **易错点：**
  + 未考虑磁盘I/O代价
  + 缓冲区管理不当导致频繁读写

#### 13 关键结论汇总

* **排序下界证明：**

// 决策树叶子节点数 >= n!

// 树高 h >= log2(n!) ≈ n\*log2(n) - n/log(2) + O(log n)

* **同时找最大最小值：**

int min\_comparisons = ceil(3.0\*n/2) - 2; // 黄金公式

* **归并最坏比较次数：**

int max\_compares = 2\*n - 1; // 合并两个长度为n的数组

* **逆序对与排序关系：**

逆序对数量 = 使数组有序的最小交换次数（相邻交换）

* **拓扑排序核心性质：**

if(结果序列长度 != 节点数)

存在环； // 检测有向图环的算法

* **重要提醒：**
  + 快排分区时优先使用 do-while 避免死循环
  + 归并排序求逆序对时，mid-i+1 的推导需结合决策树模型理解
  + 计数排序的前缀和计算必须包含 cnt[i] += cnt[i-1]
  + 堆排序的下沉操作要从最后一个非叶子节点开始
  + 多关键字排序必须从最低优先级关键字开始处理

### 真题强化

#### 1. 2010年第12题（排序时间复杂度下限）

基于比较的排序时间复杂度的下限是（ ），其中 n 表示待排序的元素个数。

A. Θ(n)

B. Θ(n log n)

C. Θ(log n)

D. Θ(n²)

**答案：B**

**解析：**

基于比较的排序算法（如快速排序、归并排序、堆排序）的时间复杂度下限为 Ω(n log n)，这是由决策树模型严格证明的。

示例算法：

* 归并排序：所有情况（最优、平均、最坏）均为 **O(n log n)**。
* 快速排序：平均 O(n log n)，但最坏情况（如数组已排序）为 O(n²)。
* 堆排序：所有情况 O(n log n)。

#### 2. 2013年第14题（快速排序时间复杂度）

（ ）的平均时间复杂度为 O(nlogn)，其中 n 是待排序的元素个数。

A. 快速排序

B. 插入排序

C. 冒泡排序

D. 基数排序

**答案：A**

**解析：**

快速排序的平均时间复杂度为 O(n log n)，这是由其分治策略决定的：

* 算法过程：随机选择一个基准元素，将数组分为小于基准和大于基准的两部分，然后递归排序子数组。
* 平均情况分析：假设基准选择均匀，每次分割大致平衡（子数组大小比例常数），则**递归深度为 O(log n)，每层比较和交换操作总计 O(n)，整体 O(n log n)**。
* 最坏情况：当数组已排序或逆序时，每次分割极度不平衡（一个子数组为空），时间复杂度退化为 **O(n²)。**优化方法包括随机选择基准（避免固定位置）或使用三数取中法。

对比其他选项：

* 插入排序：平均和最坏情况均为 O(n²)，适用于小规模或部分有序数据。
* 冒泡排序：平均和最坏情况均为 O(n²)，效率低。
* 基数排序：非比较排序，时间复杂度 O(nk)（k 为数字位数），但依赖于数据特性，不保证 O(n log n)。

因此，快速排序是唯一满足平均 O(n log n) 的选项。

#### 21.1.23. 2010年第18题（拓扑排序）

关于拓扑排序，下面说法正确的是（ ）。

A. 所有连通的有向图都可以实现拓扑排序

B. 对同一个图而言，拓扑排序的结果是唯一的

C. 拓扑排序中入度为 0 的结点总会排在入度大于 0 的结点的前面

D. 拓扑排序结果序列中的第一个结点一定是入度为 0 的点

**答案：D**

**解析：**

拓扑排序是针对**有向无环图（DAG）**的线性排序算法，满足所有有向边从序列中前驱指向后继：

**条件要求**：图必须是无环的（DAG）。如果图存在环（如 A→B→C→A），则无法进行拓扑排序（选项 A 错误，因为连通图可能有环）。

**唯一性**：拓扑排序结果不唯一。当多个结点入度为 0 时，选择顺序任意（如结点 A 和 B 均无依赖，谁先均可），选项 B 错误。

**入度 0 结点的位置**：拓扑排序起始于入度为 0 的结点（无前驱依赖），但后续步骤中新增的入度 0 结点可能排在已有入度大于 0 的结点之后（选项 C 错误）。例如，图中有边 A→B，拓扑序列为 [A, B]，B 初始入度大于 0。

**正确选项 D**：序列的第一个结点必须是入度为 0 的点，因为没有其他结点指向它。算法步骤：

* 选择入度为 0 的结点加入序列。
* 删除该结点及其出边（减少相关结点入度）。
* 重复直到图为空或发现环。
* 如果序列第一个结点入度非 0，则存在矛盾（有未处理的依赖）。因此 D 正确。

#### 4. 2009年第15题（算法复杂度）

快速排序最坏情况下的算法时间复杂度为：

A. O(log₂n)

B. O(n)

C. O(nlog₂n)

D. O(n²)

**答案：D**

**解析：**

快速排序的最坏情况时间复杂度为 O(n²)，发生在每次分割（partition）都极度不平衡时：

* **最坏情况场景**：当输入数组已完全有序（升序或降序）或基准选择策略不佳（如总是选第一个元素），导致每次分割后一个子数组为空，另一个子数组包含剩余所有元素。递归深度变为 n（而非平均的 log n），每层分割操作需 O(n) 时间（比较所有元素），总计 O(n) \* O(n) = O(n²)。
* **优化避免**：随机选择基准（如 rand() % length）可将最坏情况概率降至极低，但理论上仍存在 O(n²) 可能。
* **对比其他情况**：
  + 最佳情况：每次分割平衡（子数组大小相等），时间复杂度 O(n log n)。
  + 平均情况：随机输入下 O(n log n)。

选项分析：

* A. O(log₂n) 是二分查找等算法的时间，不适用排序。
* B. O(n) 是线性扫描的时间，但排序至少需比较元素。
* C. O(n log n) 是快速排序的平均情况，非最坏情况。

因此，最坏情况为 O(n²)。

#### 5. 2017年第11题（逆序对计数）

对于给定的序列 {a\_k}，我们把 (i, j) 称为逆序对当且仅当 i<j 且 a\_i > a\_j。那么序列 1,7,2,3,5,4 的逆序对数为（ ）个。

A. 4

B. 5

C. 6

D. 7

**答案：B**

**解析：**

逆序对定义为序列中位置 i < j 但值 a\_i > a\_j 的元素对。手动计算序列 [1,7,2,3,5,4] 的逆序对：

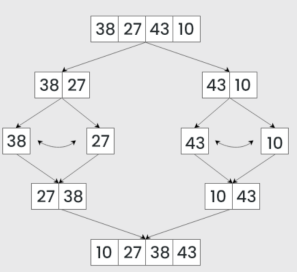
* **逐步比较：**
  + 固定 i=0 (a\_i=1)：无逆序（1 < 所有后续元素）。
  + 固定 i=1 (a\_i=7)：比较 j>1：
    - (7,2) → 7>2，逆序对
    - (7,3) → 7>3，逆序对
    - (7,5) → 7>5，逆序对
    - (7,4) → 7>4，逆序对
  + 固定 i=2 (a\_i=2)：比较 j>2：
    - (2,3) → 2<3，无逆序
    - (2,5) → 2<5，无逆序
    - (2,4) → 2<4，无逆序
  + 固定 i=3 (a\_i=3)：比较 j>3：
    - (3,5) → 3<5，无逆序
    - (3,4) → 3<4，无逆序
  + 固定 i=4 (a\_i=5)：比较 j=5 (a\_j=4)：
    - (5,4) → 5>4，逆序对
  + 固定 i=5 (a\_i=4)：无后续元素。
* **逆序对列表：**(7,2)、(7,3)、(7,5)、(7,4)、(5,4)，共 5 个。
* **算法应用：**
  + 暴力枚举需 O(n²) 时间（n 为元素个数）。
  + 高效方法：归并排序可在 O(n log n) 时间内计算逆序对。算法在合并有序子数组时统计跨子数组的逆序对（当左子数组元素大于右子数组元素时计数）。例如，序列 [1,7,2,3,5,4] 用归并排序处理时，在合并 [7] 和 [2,3] 等步骤会累计逆序数。

#### 6. 2017年第17题（归并排序比较次数）

设 A 和 B 是两个长为 n 的有序数组，现在需要将 A 和 B 合并成一个排好序的数组，任何以元素比较作为基本运算的归并算法在**最坏**情况下**至少**要做（ ）次比较。

A. n²

B. nlogn

C. 2n

D. 2n - 1

**答案：D**

**解析：**

合并两个长度为 n 的有序数组时，最坏情况下的最小比较次数为 2n - 1，由算法性质决定：

* **最坏情况场景**：两个数组元素交错排列，例如 A = [1,3,5]，B = [2,4,6]。每次比较只能确定一个元素的位置，且最后一个元素无需比较（自动确定）。
* **证明**：合并过程类似遍历两个数组。每次比较输出一个较小元素，但输出第 k 个元素（除最后一个）需一次比较**。总元素数 2n，输出前 2n-1 个元素各需一次比较，最后一个直接输出，故至少 2n-1 次。**
* **示例**：合并 A=[1,3,5] 和 B=[2,4,6]：
  + 1 vs 2 → 取 1（比较 1 次）
  + 3 vs 2 → 取 2（比较 1 次）
  + 3 vs 4 → 取 3（比较 1 次）
  + 5 vs 4 → 取 4（比较 1 次）
  + 5 vs 6 → 取 5（比较 1 次），6 自动输出
  + 总比较次数：5 = 2×3 - 1。
* **为何不能更少**：若少于 2n-1 次比较，则存在未比较元素对，可能破坏有序性（如漏掉关键比较导致错误顺序）。
* 归并排序的合并步骤时间复杂度为 O(n)，但比较次数下界为 2n-1，实际实现通常接近此值。

#### 7. 2008年第14题（排序）

将数组 {8, 23, 4, 16, 77, -5, 53, 100} 中的元素按从大到小的顺序排列，每次可以交换任意两个元素，**最少**需要交换（ ）次。

A. 4

B. 5

C. 6

D. 7

**答案：B**

**解析：**

最少交换次数对应于选择排序中交换步骤的优化。**选择排序**通过每次选择未排序部分的最大元素并交换到正确位置，最小化交换次数（因为每次交换将一个元素归位）：

* **数组初始状态**：{8, 23, 4, 16, 77, -5, 53, 100}（n=8）。
* **排序过程（从大到小）：**
  + 找到最大值 100（位置 7），与首元素 8 交换：数组变为 {100, 23, 4, 16, 77, -5, 53, 8}，**交换 1 次**。
  + 未排序部分 {23, 4, 16, 77, -5, 53, 8}，最大值 77（位置 4），与当前首元素 23 交换：{100, 77, 4, 16, 23, -5, 53, 8}，**交换 2 次**。
  + 未排序部分 {4, 16, 23, -5, 53, 8}，最大值 53（位置 6），与 4 交换：{100, 77, 53, 16, 23, -5, 4, 8}，**交换 3 次**。
  + 未排序部分 {16, 23, -5, 4, 8}，最大值 23（位置 1），与 16 交换：{100, 77, 53, 23, 16, -5, 4, 8}，**交换 4 次**。
  + 未排序部分 {16, -5, 4, 8}，最大值 16 在位置 0，无需交换（已归位）。
  + 下一步：未排序部分 {-5, 4, 8}，最大值 8（位置 3），与 -5 交换：{100, 77, 53, 23, 16, 8, 4, -5}，**交换 5 次**。
  + 最后 {-5} 自动归位。
  + 总交换次数：5 次（步骤 1、2、3、4、5）。
* **为什么最少**：每个元素归位最多需一次交换（除已归位元素）。初始数组有 8 个元素，但 100 和 77 等归位各需一次交换。最小交换次数等于 n - 循环数（在置换群论中），本例中通过选择排序实现最小化。

#### 8. 2018年第9题（同时找最大最小值）

给定一个含 N 个不相同数字的数组，在最坏情况下，找出其中最大或最小的数，至少需要 N-1 次比较操作。则最坏情况下，在该数组中同时找最大与最小的数至少需要（ ）次比较操作。（ ⌈⌉ 表示向上取整，⌊⌋ 表示向下取整）

A. ⌈3N/2⌉ - 2

B. ⌊3N/2⌋ - 2

C. 2N - 2

D. 2N - 4

**答案：A**

**解析：**

同时找最大值和最小值时，优化策略可减少比较次数。基本思路是元素两两分组：

* **算法步骤**：
  + 将元素分为 ⌊N/2⌋ 组（每组两个元素，若 N 奇数，则最后一组一个元素）。
  + 每组内比较一次，分出较大值和较小值（共 ⌊N/2⌋ 次比较）。
  + 从所有较大值组成的集合中找最大值（需 ⌈N/2⌉ - 1 次比较，因为集合大小为 ⌈N/2⌉）。
  + 从所有较小值组成的集合中找最小值（需 ⌈N/2⌉ - 1 次比较）。
* **总比较次数**：⌊N/2⌋ + (⌈N/2⌉ - 1) + (⌈N/2⌉ - 1) = ⌊N/2⌋ + 2⌈N/2⌉ - 2。
  + 由于 ⌊N/2⌋ + ⌈N/2⌉ = N，且 ⌈N/2⌉ = ⌊N/2⌋ + (N mod 2)，因此：
  + **若 N 偶数**：⌊N/2⌋ = N/2, ⌈N/2⌉ = N/2，总次数 = N/2 + 2\*(N/2) - 2 = N/2 + N - 2 = **3N/2 - 2**。
  + **若 N 奇数**：⌊N/2⌋ = (N-1)/2, ⌈N/2⌉ = (N+1)/2，总次数 = (N-1)/2 + 2\*(N+1)/2 - 2 = (N-1)/2 + (N+1) - 2 = (N-1)/2 + N - 1 = (3N-3)/2 = ⌈3N/2⌉ - 2（因 ⌈3N/2⌉ = **(3N+1)/2** 当 N 奇数）。
  + 综上，总次数为 ⌈3N/2⌉ - 2。
* **示例（N=5）**：数组假设为 [a,b,c,d,e]。
  + 分组：[a,b]、[c,d]、[e]（最后一组单元素）。
  + 组内比较：a vs b（1 次）、c vs d（1 次），e 无比较（共 2 次）。
  + 较大值集合：[max(a,b), max(c,d), e]，找最大值需 2 次比较（如 max(a,b) vs max(c,d)，胜者 vs e）。
  + 较小值集合：[min(a,b), min(c,d), e]，找最小值需 2 次比较。
  + 总计：2 + 2 + 2 = 6 = ⌈15/2⌉ - 2 = 8 - 2 = 6。
  + 此策略是最优的，因为每次比较至多减少一个候选（最大或最小值），初始候选数为 2N（每个元素可能为最大或最小），需 2N-2 次比较，但分组法通过共享比较信息降低到 ⌈3N/2⌉ - 2。

#### 2019年第20题（双关键字计数排序）

（计数排序）计数排序是一个广泛使用的排序方法。下面的程序使用双关键字计数排序，将 n 对 10000 以内的整数，从小到大排序。

例如有三对整数 (3,4)、(2,4)、(3,3)，那么排序之后应该是 (2,4)、(3,3)、(3,4) 。

输入第一行为 n，接下来 n 行，第 i 行有两个数 a[i] 和 b[i]，分别表示第 i 对整数的第一关键字和第二关键字。

从小到大排序后输出。

数据范围 1<n<10^7 ，1<a[i],b[i]<10^4。

提示：应先对第二关键字排序，再对第一关键字排序。数组 ord[] 存储第二关键字排序的结果，数组 res[] 存储双关键字排序的结果。

试补全程序。

#include <cstdio>

#include <cstring>

using namespace std;

const int maxn = 10000000;

const int maxs = 10000;

int n;

unsigned a[maxn], b[maxn], res[maxn], ord[maxn];

unsigned cnt[maxs + 1];

int main() {

scanf("%d", &n);

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

scanf("%d%d", &a[i], &b[i]);

memset(cnt, 0, sizeof(cnt));

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

①;

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

cnt[i + 1] += cnt[i];

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

②;

memset(cnt, 0, sizeof(cnt));

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

③;

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

cnt[i + 1] += cnt[i];

for (int i = n - 1; i >= 0; --i)

④;

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

printf("%d %d\n", ⑤);

return 0;

}

**答案：**

**① ++cnt[b[i]]**

**② ord[--cnt[b[i]]] = i**

**③ ++cnt[a[i]]**

**④ res[--cnt[a[ord[i]]]] = ord[i]**

**⑤ a[res[i]], b[res[i]]**

**解析：**

该程序实现**双关键字计数排序（稳定排序）**，先按第二关键字 b 排序，再按第一关键字 a 排序：

* **算法原理**：计数排序适用于整数关键字范围较小（maxs=10000）的情况。稳定排序要求相同关键字的元素保持输入顺序。
* **步骤分解：**
  + **按第二关键字 b 排序：**
    - ① ++cnt[b[i]]：**统计每个 b 值的出现频率**（cnt 数组下标为 b[i]，值计数）。
    - cnt[i+1] += cnt[i]：计算**前缀和**，使 cnt[k] 表示 b 值 ≤ k 的元素个数（即 b 值 k 的起始位置）。
    - ② ord[--cnt[b[i]]] = i：生成按 b 排序的索引数组 ord。--cnt[b[i]] 获取当前 b[i] 的**插入位**置（从后向前填充以保证稳定性），ord 存储原数组索引 i。
  + **按第一关键字 a 排序：**
    - ③ ++cnt[a[i]]：**重置** cnt 并统计 a 值的频率。
    - cnt[i+1] += cnt[i]：计算 a 的前缀和。
    - ④ res[--cnt[a[ord[i]]]] = ord[i]：结合 ord 数组生成最终排序索引 res。**从后向前遍历** ord（保证稳定性），a[ord[i]] 获取 a 值，--cnt[a[ord[i]]] 计算插入位置，res 存储排序后的索引。
  + **输出**：⑤ a[res[i]], b[res[i]]：按 res 索引顺序输出原始数据对。
* **关键点：**
  + 步骤②和④从后向前遍历确保稳定性（相同关键字元素保持原序）。
  + ord 数组保存按 b 排序后的索引，用于第二阶段的 a 排序。
  + 时间复杂度 O(n + maxs)，maxs 为关键字范围（此处 10000）。
* **示例**：假设输入 [(a1,b1), (a2,b2)]，程序先按 b 排序索引到 ord，再按 a 对 ord 排序到 res，输出时按 res 取原始数据。

#### 2008年第28题（快速选择算法）

**（找第 k 大的数）** 给定一个长度为 10^6的无序正整数序列, 以及另一个数

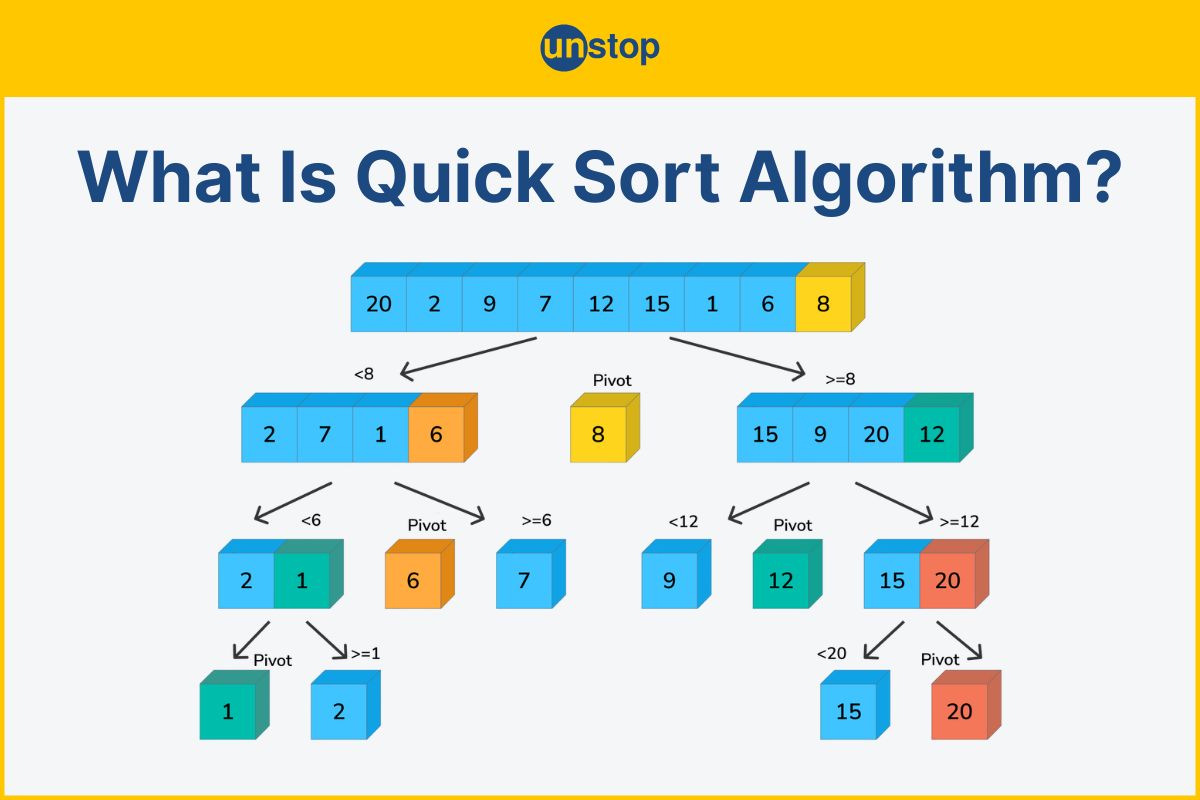
n(1≤n≤10^6), 然后以类似快速排序的方法找到序列中第 n 大的数（关于第 n 大的数：例如序列 {1,2,3,4,5,6} 中第 3 大的数是 4）。

int FindKth(int left, int right, int n) {

int tmp, value, i, j;

if (left == right) return left;

**tmp = rand() % (right - left) + left;**

 swap(a[tmp], a[left]);

value = ①;

i = left;

j = right;

while (i < j) {

while (i < j && ②) j--;

if (i < j) { a[i] = a[j]; i++; } else break;

while (i < j && ③) i++;

if (i < j) { a[j] = a[i]; j--; } else break;

}

④;

if (i < n) return FindKth(⑤);

if (i > n) return FindKth(⑥);

return i;

}

**答案：**

**① a[left]**

**② a[j] < value**

**③ a[i] > value**

**④ a[i] = value**

**⑤ i + 1, right, n**

**⑥ left, i - 1, n**

**解析：**

此代码实现**快速选择算法（QuickSelect）**，用于在未排序数组中找到第 n 小（或大）的元素，基于快速排序的分区思想：

* **算法原理**：随机选择基准元素，将数组分为小于基准和大于基准的两部分，然后递归搜索目标所在分区。
* **分区过程：**
  + ① value = a[left]：基准值设为交换后**左端元素**（随机化避免最坏情况）。
  + 循环中：
    - ② while (i < j && a[j] < value) j--：**从右向左找首个小于基准的元素**（若找第 n 小，需将小于基准放右侧）。
    - 如果找到，将 a[j] 移到 a[i]（覆盖），i 右移。
    - ③ while (i < j && a[i] > value) i++：**从左向右找首个大于基准的元素**（注意：此处应为 a[i] > value 以正确分区）。
    - 如果找到，将 a[i] 移到 a[j]（覆盖），j 左移。
  + ④ a[i] = value：循环结束时 i=j，将基准值放入正确位置。
* **递归搜索：**
  + if (i < n)：基准位置 i **小于**目标 n，表示目标在**右分区**，递归 FindKth(i + 1, right, n)。
  + if (i > n)：基准位置 i **大于**目标 n，表示目标在**左分区**，递归 FindKth(left, i - 1, n)。
  + return i：i == n 时，找到目标。
* **时间复杂度：**
  + 平均 O(n)（每次分区大致减半）。
  + 最坏 O(n²)（当每次分区极度不平衡），但随机基准（rand() % (right - left) + left）降低概率。
* **示例**：数组 [3,1,4,2]，找第 2 小元素（n=1，假设索引从 0 开始）。随机选基准 1，分区后 [1,3,4,2]，i=0，递归右分区 [3,4,2] 找第 1 小（n=1），最终返回正确索引。

#### 11. 2021年第4题（排序、比较）

以比较作为基本运算，在 N 个数中找出最大数，最坏情况下所需要的最少的比较次数为 （ ）。

A. N²

B. N

C. N-1

D. N+1

**答案：C**

**解析：**

在 N 个元素中找出最大值时，最坏情况下的最小比较次数为 N-1。这是因为：

* **算法原理**：采用“擂台法”（或线性扫描）。初始化最大值候选为第一个元素，之后依次与其他元素比较。每个后续元素最多需一次比较：若当前元素大于候选值，则更新候选值。
* **比较次数分析：**
  + 第一个元素无需比较（直接设为候选）。
  + 剩余 N-1 个元素各需一次比较（共 N-1 次）。
* **最坏情况**：最大值位于序列末尾时，需完整比较所有元素（例如序列 [2,3,4,1]，最大值 4 在末尾，比较 3 次 = 4-1）。
* **为什么不能更少**：若少于 N-1 次比较，则存在未比较元素，可能遗漏最大值（例如跳过某个元素，导致错误结果）。
* 此结论基于信息论：每次比较最多减少一个候选值，初始候选数为 N，最终需确定唯一最大值，至少需 N-1 次比较。

#### 12. 2009年第17题（排序）

排序算法是稳定的意思是关键码相同的记录排序前后相对位置不发生改变，下列哪种排序算法是不稳定的：

A. 冒泡排序

B. 插入排序

C. 归并排序

D. 快速排序

**答案：D**

**解析：**

排序算法的稳定性指相同关键码元素的相对位置在排序后保持不变。快速排序是不稳定的，原因如下：

* **不稳定机制**：在分区（partition）过程中，基准元素与交换操作可能导致相同关键码元素交换位置。例如序列 [3a, 2, 3b]（3a 和 3b 为相同值），若选 2 为基准：
  + 分区后可能变为 [2, 3b, 3a]，3a 和 3b 的相对顺序改变。
* **对比稳定算法**：
  + 冒泡排序：相邻元素比较交换，相同值不交换（稳定）。
  + 插入排序：元素插入时遇到相同值则停止移动（稳定）。
  + 归并排序：合并子数组时，相同值优先取左子数组元素（稳定）。
* **快速排序不稳定的根源**：非相邻交换和随机基准选择。优化措施（如三路快排）可改善稳定性，但标准实现不稳定。
* 稳定性在现实场景中很重要（如先按分数排序，再按姓名排序时需保持分数相同者的顺序）。

#### 13. 2011年第8题（排序）

体育课的铃声响了，同学们都陆续地奔向操场，按老师的要求从高到矮站成一排。每个同学按顺序来到操场时，都从排尾走到排头，找到第一个比自己高的同学，并站在他的后面。这种站队的方法类似于（ ）算法。

A. 快速排序

B. 插入排序

C. 冒泡排序

D. 归并排序

**答案：B**

**解析：**

这种站队方法直接对应插入排序算法的核心思想：

* **插入排序原理：**将未排序元素逐个插入到已排序序列的适当位置。过程包括：
  + 初始化：第一个同学自成有序序列。
  + 后续同学从序列尾部向前扫描，找到第一个比自身高的位置并插入其后（保证从高到矮有序）。
* **类比分析：**
  + 每个新同学相当于待插入元素。
  + “从排尾走到排头” 对应扫描已排序序列。
  + “找到第一个比自己高的同学并站其后面” 对应插入点定位。
* **与其他算法区别：**
  + 快速排序：基于分区和交换（如选基准后分左右），不涉及逐个插入。
  + 冒泡排序：通过相邻元素交换“冒泡”最大/小值到端，不涉及插入操作。
  + 归并排序：分治合并子序列，非逐个处理。
  + 插入排序时间复杂度为 O(n²)，但在此场景下直观高效，因为站队过程是典型的在线插入过程。

#### 14. 2022年第12题（排序）

以下排序算法的常见实现中，哪个选项的说法是错误的：（ ）。

A. 冒泡排序算法是稳定的

B. 简单选择排序是稳定的

C. 简单插入排序是稳定的

D. 归并排序算法是稳定的

**答案：B**

**解析：**

简单选择排序是不稳定的，常见实现中可能改变相同关键码元素的相对位置：

* **不稳定原因**：选择排序通过交换操作将最小（或最大）元素放到正确位置。例如序列 [5a, 2, 5b, 1]（5a 和 5b 为相同值）：
  + 第一轮：选最小值 1 与 5a 交换 → [1, 2, 5b, 5a]，5a 和 5b 顺序颠倒。
* **稳定算法分析**：
  + 冒泡排序（A）：相邻比较，相同值不交换（稳定）。
  + 简单插入排序（C）：插入时遇相同值则停止（稳定）。
  + 归并排序（D）：合并时优先取左子数组元素（稳定）。
* **选择排序的改进**：若使用链表或额外空间可变为稳定，但标准数组交换实现不稳定。

#### 15. 2018年第8题（排序）

以下排序算法中，不需要进行关键字比较操作的算法是（ ）。

A. 基数排序

B. 冒泡排序

C. 堆排序

D. 直接插入排序

**答案：A**

**解析：**

基数排序是一种非比较排序算法，其核心不依赖元素间的比较操作：

* **算法原理**：基于关键字的位数分配和收集（如 LSD 最低位优先）。
  + 步骤：按个位分配到桶 → 收集 → 按十位分配到桶 → 收集 → 依此类推。
  + 操作：仅涉及数据移动和桶管理，无元素比较。
* **时间复杂度**：O(nk)，其中 k 为关键字位数（与比较无关）。
* **对比其他算法**（均需比较）：
  + 冒泡排序（B）：通过相邻比较交换排序。
  + 堆排序（C）：建堆和调整堆过程依赖比较。
  + 直接插入排序（D）：扫描有序序列时需比较定位插入点。
* **应用限制：**基数排序要求关键字可分解为固定位数（如整数或字符串），且位数 k 不宜过大（否则效率降低）。
* 非比较排序（如计数排序、桶排序）在特定条件下可突破比较排序下界 Ω(n log n)，但基数排序是最典型代表。

#### 16. 2012年第8题（排序）

使用冒泡排序对序列进行升序排列，每执行一次交换操作系统将会减少 1 个逆序对，因此序列 5,4,3,2,1 需要执行（ ）次操作，才能完成冒泡排序。

A. 0

B. 5

C. 10

D. 15

**答案：C**

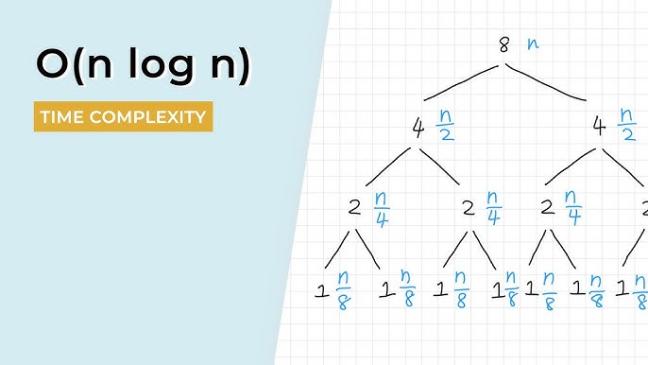
**解析：**

冒泡排序每交换一次相邻元素减少一个逆序对。序列 [5,4,3,2,1] 的逆序对数为：

* **逆序对定义**：位置 i < j 但 a\_i > a\_j。
* **手动计算**：
  + (5,4)、(5,3)、(5,2)、(5,1) → 4 对
  + (4,3)、(4,2)、(4,1) → 3 对
  + (3,2)、(3,1) → 2 对
  + (2,1) → 1 对
  + 总计 4+3+2+1=10 对。
* **冒泡排序过程**：
  + 第一轮：比较 4 次，交换 4 次 → [4,3,2,1,5]
  + 第二轮：比较 3 次，交换 3 次 → [3,2,1,4,5]
  + 第三轮：比较 2 次，交换 2 次 → [2,1,3,4,5]
  + 第四轮：比较 1 次，交换 1 次 → [1,2,3,4,5]
  + 总交换次数：4+3+2+1=10 次。
* **为什么是 10 次**：每次交换减少一个逆序对，初始逆序对数即为总交换次数。序列 [n,n-1,...,1] 的逆序对数为 n(n-1)/2（此处 n=5，5×4/2=10）。

#### 17. 2019年第5题（排序）

设有 100 个已排好序的数据元素，采用折半查找时，最大比较次数为（）。

A. 7

B. 10

C. 6

D. 8

**答案：A**

**解析：**

折半查找（二分查找）在有序数组中的最大比较次数由二叉树深度决定：

* **算法原理**：每次比较将搜索区间减半。
* **计算最大比较次数**：
  + 公式：⌈log₂(n)⌉，其中 n 为元素个数。
  + 此处 n=100，log₂(100) ≈ 6.644，向上取整为 7。
  + 推导过程：
  + 比较次数 k 满足 2^{k-1} < n ≤ 2^k。
  + 2^6=64 < **100 ≤ 128=2^7，故 k=7。**
* **最坏情况示例**：查找不在数组中的元素或边界值（如查找小于最小值的数），需完整比较直到区间为空。
  + 例如：在 [1,2,...,100] 中查找 0，路径为：中位50 → 25 → 12 → 6 → 3 → 1 → 比较结束（7次）。
* **时间复杂度**：O(log n)，比顺序查找 O(n) 高效。

#### 18. 2020年第5题（冒泡排序）

冒泡排序算法的伪代码如下：

输入：数组L, n ≥ k。输出：按非递减顺序排序的 L。

算法 BubbleSort：

FLAG ← n //标记被交换的最后元素位置

while FLAG > 1 do

k ← FLAG -1

FLAG ← 1

for j=1 to k do

if L(j) > L(j+1) then do

L(j) ↔ L(j+1)

FLAG ← j

对 n 个数用以上冒泡排序算法进行排序，**最少**需要比较多少次?（ ）

A. n²

B. n-2

C. n-1

D. n

**答案：C**

**解析：**

该伪代码是冒泡排序的优化版本（通过 FLAG 标记减少无效比较）。最少比较次数发生在输入已有序时：

* **算法分析**：
  + FLAG 初始为 n，表示扫描范围。
  + 若数组已有序，则内层 for 循环无交换（FLAG 保持 1），while 循环仅执行一次。
* **最少比较次数**：
  + 内层循环 j 从 1 到 k = FLAG-1 = n-1，共执行 n-1 次比较。
  + 无交换 → FLAG=1 → while 循环结束。
* **示例**：n=3，数组 [1,2,3]：
  + 第一次 while：k=2，比较 L(1) vs L(2)（1<2，无交换），比较 L(2) vs L(3)（2<3，无交换）→ 比较 2 次 = 3-1。
  + FLAG 保持 1，循环结束。
* **与标准冒泡排序区别**：优化后，最好情况（已有序）比较次数为 n-1，而标准版本固定为 O(n²)。最坏情况（逆序）仍为 O(n²)。

#### 19. 2023年第12题（图）

考虑一个**有向无环图**，该图包含 4 条有向边：(1,2),(1,3),(2,4) 和 (3,4)。以下哪个选项是这个有向无环图的一个有效的拓扑排序?

A. 4,2,3,1

B. 1,2,3,4

C. 1,2,4,3

D. 2,1,3,4

**答案：B**

**解析：**

拓扑排序要求所有有向边从序列前驱指向后继（即无后向边）。图结构：

* **顶点与边**：顶点 {1,2,3,4}，边 (1,2)、(1,3)、(2,4)、(3,4)。
* **依赖关系**：
  + 1 无前驱（入度 0）。
  + 2 依赖 1（边 1→2）。
  + 3 依赖 1（边 1→3）。
  + 4 依赖 2 和 3（边 2→4 和 3→4）。
* **拓扑排序条件**：每个顶点前驱必须在其之前。
* **选项验证**：
  + A. [4,2,3,1]：4 在 2 和 3 前，但边 (2,4) 要求 2 在 4 前 → 无效。
  + B. [1,2,3,4]：满足 1→2→4 和 1→3→4 → 有效（2 和 3 顺序可变）。
  + C. [1,2,4,3]：4 在 3 前，但边 (3,4) 要求 3 在 4 前 → 无效。
  + D. [2,1,3,4]：2 在 1 前，但边 (1,2) 要求 1 在 2 前 → 无效。
* **拓扑排序算法：**
  + 选择入度 0 的顶点（此处为 1）加入序列。
  + 删除 1 及其出边 → 顶点 2 和 3 入度变为 0。
  + 选择 2 或 3（顺序不唯一），例如选 2 → 删除 2 及边 (2,4) → 顶点 4 入度减为 1（还需 3）。
  + 选 3 → 删除 3 及边 (3,4) → 顶点 4 入度变 0 → 加入序列。
  + 结果可为 [1,2,3,4] 或 [1,3,2,4]。

强化练习 第五节

# 贪心算法

### 考点分析

|  |  |  |  |
| --- | --- | --- | --- |
| **结构类型** | **重点考点** | **考察频率** | **难度** |
| **策略设计** | **最优子结构选择** | 极高 | ★★★ |
| **反证法验证正确性** | 高 | ★★★★ |
| **排序预处理** | 极高 | ★★ |
| **经典模型** | **区间问题（覆盖/选点）** | 高 | ★★★ |
| **过河问题** | 中 | ★★★ |
| **二分答案+贪心验证** | 高 | ★★★★ |
| **哈夫曼编码** | 中 | ★★ |
| **实现细节** | **循环边界条件** | 高 | ★★ |
| **指针索引更新** | 高 | ★★ |
| **状态回溯与恢复** | 低 | ★★★ |
| **复杂度** | **时间优化** | 中 | ★★★ |
| **空间复杂度控制** | 低 | ★ |
| **特殊处理** | **边界条件处理** | 高 | ★★★ |
| **相等元素处理** | 中 | ★★ |
| **浮点数精度处理** | 低 | ★★★ |

### IMG_256核心知识点

#### 1. 最优子结构选择（策略设计）

* **核心知识点：**

贪心算法的核心在于每一步选择**当前状态下的最优解**，最终得到全局最优解。关键在于证明局部最优能推导出全局最优。

* **黄金法则：**

**"排序预处理 + 局部最优迭代**"是解决贪心问题的通用模式。排序后可以简化决策过程，使贪心选择更高效。

* **高频公式：**

// 区间按右端点排序

sort(intervals.begin(), intervals.end(), [](const auto& a, const auto& b) {

return a[1] < b[1];

});

// 过河问题取两人时间最大值

int cross\_time = max(a, b);

* **易错点：**
  + 未正确选择排序属性（如区间问题应优先按**右端点**排序）
  + 忽略问题特性直接套用经典模型
  + 未处理边界情况（如空集合或单元素）
* **C++模板（区间选择）：**

vector<vector<int>> greedyInterval(vector<vector<int>>& intervals) {

sort(intervals.begin(), intervals.end(), [](const auto& a, const auto& b) {

**return a[1] < b[1]; // 按右端点升序**

});

vector<vector<int>> res;

int end = INT\_MIN;

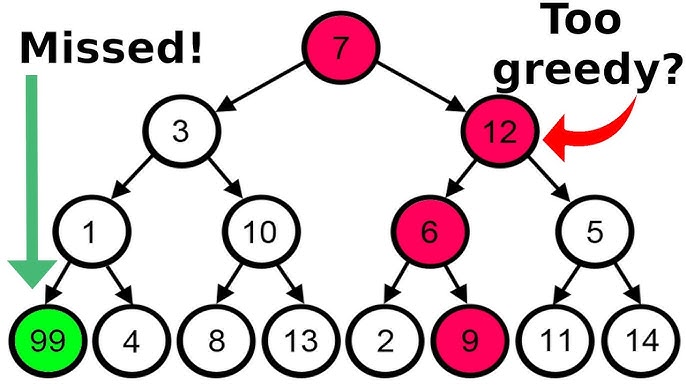
for (auto& interval : intervals) {

if (interval[0] > end) { // 无重叠

res.push\_back(interval);

end = interval[1];

}

 }

return res;

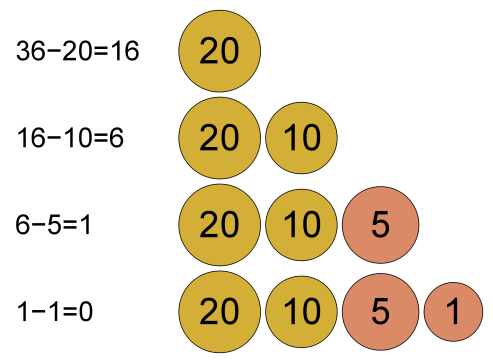
}

#### 2. 反证法验证正确性

* **核心知识点：**

贪心策略必须经过**正确性验证**，反证法是最有效的方法。假设存在更优解，推导矛盾。

* **黄金法则：**

"如果交换任意两个元素会降低效果，则贪心策略最优"。适用于任务调度、背包问题等。

**高频公式：**

// 验证郊游活动匹配策略

bool valid = true;

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

**if (M[i] < C[i] && school\_fund < (C[i] - M[i])) {**

valid = false; // 策略失效

break;

}

}

* **易错点：**
  + 忽略策略的边界条件
  + 未考虑所有可能的反例
  + 将启发式方法误认为贪心策略
* **C++模板（策略验证）：**

bool isStrategyValid(vector<int>& M, vector<int>& C, int A) {

sort(M.begin(), M.end()); // 资金升序

sort(C.begin(), C.end()); // 租金升序

int cost = 0;

for (int i = 0; i < min(M.size(), C.size()); i++) {

**if (M[M.size()-1-i] < C[i]) { // 最富租最便宜车**

**cost += C[i] - M[M.size()-1-i];**

}

}

return cost <= A; // 学校资金是否足够

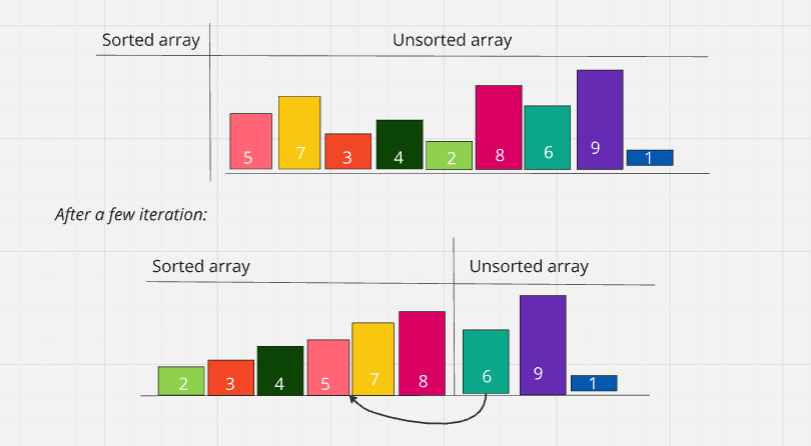
}

#### 3. 排序预处理

* **核心知识点：**

90%的贪心问题需要**先排序**。排序方向（升序/降序）直接影响贪心策略的实现。

* **黄金法则：**

"选择决定排序"：根据问题目标选择排序属性：

* + **最小化问题 → 升序排序**
  + **最大化问题 → 降序排序**
  + **区间问题 → 按右端点升序**
* **高频公式：**

// 结构体多级排序

struct Segment { int a, b; };

sort(A, A+n, [](const Segment& x, const Segment& y) {

**return x.a != y.a ? x.a < y.a : x.b > y.b;**

});

// 降序排序

sort(arr.rbegin(), arr.rend());

* **易错点：**
  + 错误选择主排序键
  + 忽略二级排序条件
  + 未处理相等元素的情况
* **C++模板（多级排序）：**

void greedySort(Segment A[], int n) {

// 主键：左端点升序，次键：右端点降序

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

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

**if (A[j].a < A[i].a ||**

**(A[j].a == A[i].a && A[j].b > A[i].b)) {**

**swap(A[i], A[j]);**

}

}

}

}

#### 4. 二分答案+贪心验证

* **核心知识点：**

当问题具有**单调性**特征时，结合二分法和贪心验证，时间复杂度可从O(n!)降至O(n log n)。

* **黄金法则：**

"猜答案 → 验证 → 调整范围"循环。验证函数通常包含贪心策略。

* **高频公式：**

// 二分框架

**int l = 1, r = max\_len;**

**while (l <= r) {**

**int mid = (l + r + 1) / 2; // 防死循环**

**if (check(mid)) l = mid + 1;**

**else r = mid - 1;**

}

// 切割绳子验证

bool check(int L) {

if (L == 0) return false;

int cnt = 0;

for (int len : ropes) **cnt += len / L;**

**return cnt >= m;**

}

* **易错点：**
  + 二分边界错误（死循环）
  + 验证函数未处理除零异常
  + 忽略整数除法截断问题
* **C++模板（切割绳子）：**

int cutRopes(vector<int>& len, int m) {

long total = accumulate(len.begin(), len.end(), 0L);

if (total < m) return -1; // 总长不足

int l = 1, r = \*max\_element(len.begin(), len.end());

while (l <= r) {

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

int cnt = 0;

for (int rope : len) cnt += rope / mid;

if (cnt >= m) l = mid + 1;

else r = mid - 1;

}

return r; // 最大可行长度

}

#### 5. 区间问题（覆盖/选点）

* **核心知识点：**

区间覆盖问题的黄金法则：按左端点排序后，贪心选择覆盖当前点且右端点最大的区间。

* **黄金法则：**

"覆盖点r的决策 = 选择满足a≤r的区间中b最大的"

* **高频公式：**

// 区间覆盖核心逻辑

int r = 0, q = 0, ans = 0;

while (r < m) {

int max\_r = r;

while (q < n && A[q].a <= r) {

max\_r = max(max\_r, A[q].b);

q++;

}

if (max\_r <= r) break; // 无法覆盖

r = max\_r;

ans++;

}

* **易错点：**
  + 未处理无法覆盖的断点
  + 循环条件缺少索引保护
  + 更新覆盖点时未取最大值
* **C++模板（最小区间覆盖）：**

int minCover(Segment A[], int n, int m) {

sort(A, A+n, [](const Segment& x, const Segment& y) {

return x.a < y.a;

});

int r = 0, ans = 0, i = 0;

while (r < m) {

int max\_r = r;

while (i < n && A[i].a <= r) {

max\_r = max(max\_r, A[i].b);

i++;

}

if (max\_r == r) return -1; // 无法覆盖

r = max\_r;

ans++;

}

return ans;

}

#### 6. 过河问题

* **核心知识点：**

过河优化的核心策略：最小化返回时间。通常让最快的人往返送灯，最慢的两人一起过河。

* **黄金法则：**

"送灯用最快，过河配慢双"。即让最快者返回，最慢两人一起过河。

* **高频公式：**

// 过河时间计算（已排序）

int time = 0;

while (n > 3) {

// 方案1：最快送最慢两人

int plan1 = times[0]\*2 + times[n-1] + times[n-2];

// 方案2：最快和次快配合

int plan2 = times[0] + times[1]\*2 + times[n-1];

time += min(plan1, plan2);

n -= 2;

}

// 处理剩余3人以内情况

if (n == 3) time += times[0] + times[1] + times[2];

else if (n == 2) time += times[1];

else time += times[0];

* **易错点：**
  + 未排序时间数组
  + 忽略人数边界条件
  + 未比较两种运输方案
* **C++模板（过河问题）：**

int minCrossingTime(vector<int>& times) {

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

int n = times.size(), total = 0;

while (n > 3) {

// 两种运输方案取最小

total += min(times[0]\*2 + times[n-1] + times[n-2],

times[0] + times[1]\*2 + times[n-1]);

n -= 2;

}

// 处理剩余人数

if (n == 3) total += times[0] + times[1] + times[2];

else if (n == 2) total += times[1];

else total += times[0];

return total;

}

#### 7. 循环边界与指针控制

* **核心知识点：**

贪心实现中的致命错误多源于循环边界和指针控制。必须确保：1) 索引不越界 2) 终止条件完备。

* **黄金法则：**

"先检查索引，再访问元素"。在while循环中始终先验证索引范围。

* **高频公式：**

// 安全的指针前进

while (q < n && A[q].a <= r) // 先检查q<n再访问

// 防越界访问

if (q > 0) r = max(r, A[q-1].b);

* **易错点：**
  + 循环条件顺序错误
  + 指针更新遗漏
  + 边界值处理不当（如q=0时访问A[q-1]）
* **C++模板（安全指针）：**

void safeTraversal(Segment A[], int n, int m) {

int r = 0, q = 0, ans = 0;

while (r < m) {

int max\_r = r;

// 关键：先检查q<n再访问A[q]

while (q < n && A[q].a <= r) {

max\_r = max(max\_r, A[q].b);

q++;

}

if (q == 0 || max\_r == r) break; // 无法前进

r = max\_r;

ans++;

}

if (r < m) cout << "Coverage failed";

else cout << ans;

}

#### 8. 状态回溯与恢复

* **核心知识点：**

递归型贪心算法必须保持状态一致性。**修改全局状态后，递归返回时要恢复原状**。

* **黄金法则：**

"递归前修改，返回前恢复"。类似深度优先搜索的回溯机制。

* **高频公式：**

// 过河问题状态回溯

pos[i] = LEFT; // 修改状态

int time = max(times[i], times[j]) + go(LEFT\_TO\_RIGHT);

best = min(best, time);

pos[i] = RIGHT; // 恢复状态

* **易错点：**
  + 忘记恢复状态
  + 递归前未检查条件
  + 未处理重复计算
* **C++模板（状态回溯）：**

int dfs(vector<bool>& pos, vector<int>& times, bool direction) {

if (allOnLeft(pos)) return 0;

int best = INT\_MAX;

if (direction == RIGHT\_TO\_LEFT) {

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

if (!pos[i]) continue; // 跳过左岸的人

for (int j = i+1; j < pos.size(); j++) {

if (!pos[j]) continue;

// 修改状态

pos[i] = LEFT;

pos[j] = LEFT;

// 递归计算

int time = max(times[i], times[j]) +

dfs(pos, times, LEFT\_TO\_RIGHT);

// 恢复状态

pos[i] = RIGHT;

pos[j] = RIGHT;

best = min(best, time);

}

}

}

// 类似处理LEFT\_TO\_RIGHT方向

return best;

}

#### 总结关键点

* 策略设计三步骤：排序 → 贪心选择 → 正确性验证
* 复杂度控制：O(n²)算法n≤10000，O(n log n)算法n≤10⁶
* **必考经典模型：**
* **区间覆盖/选点（2020）**
* **过河问题（2010, 2021）**
* **二分+贪心（2016, 2017）**
* 调试技巧：

// 在贪心循环中添加调试输出

cout << "Step " << step++ << ": r=" << r

<< " selected [" << A[q-1].a << "," << A[q-1].b << "]\n";

* 竞赛秘籍：当无法证明贪心正确性时，可写对拍程序验证：

// 生成随机测试数据

void genTestData() {

int n = rand()%100 + 1;

cout << n << endl;

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

int a = rand()%100;

int b = a + rand()%100 + 1;

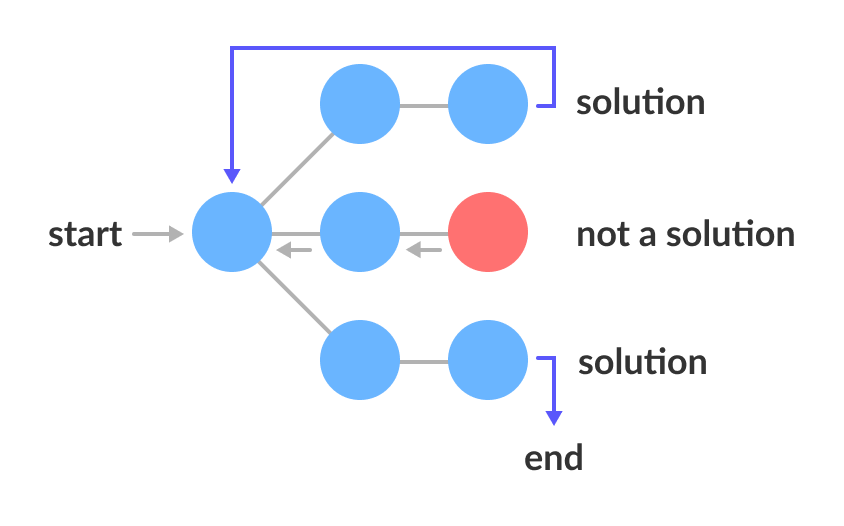
cout << a << " " << b << endl;

}

}

### 真题强化

#### 1. 2011年第17题（算法）

（ ）是一种选优搜索法，按选优条件向前搜索，以达到目标。当搜索到某一步时，发现原先选择并不优或达不到目标，就**退回**一步重新选择。

A. 回溯法

B. 枚举法

C. 动态规划

D. 贪心

**答案： A**

**解析：**

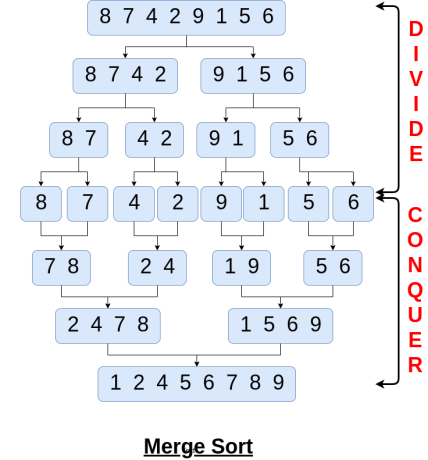
回溯法是一种基于**深度优先搜索**的选优算法。它通过逐步构建解决方案：从根节点开始，深度优先遍历所有可能的分支。当发现当前路径**无法满足条件**（如不优或达不到目标）时，算法会“回溯”（即退回上一步），**撤销**最近的选择，并尝试其他分支。这种“尝试-回溯”机制使其适用于组合优化问题（如八皇后、子集和）。

与其他选项区别：

* 枚举法（B）：遍历所有可能解，无智能回溯，效率低。
* 动态规划（C）：基于最优子结构和重叠子问题，无显式回溯。
* 贪心（D）：每一步局部最优，无回溯机制。

该题描述的核心是“退回一步重新选择”，这正是回溯法的特征。

#### 2. 2012年第15题（算法）

（ ）就是把一个复杂的问题分成两个或更多的相同类似的子问题，再把子问题分解成更小的子问题……直到最后的子问题可以简单地直接求解。而原问题的解就是子问题解的并。

A. 动态规划

B. 贪心

C. 分治

D. 搜索

**答案： C**

**解析：**

**分治法（Divide and Conquer）**的核心思想是将大问题**递归分解**为相互独立的相似子问题，直到子问题足够小可直接求解，最后**合并**子问题的解得到原问题解。典型例子包括归并排序（分解数组、排序子数组、合并）和快速排序。

关键点：子问题必须独立且与原问题**结构相同**。

与其他选项区别：

* 动态规划（A）：子问题重叠，需存储中间结果避免重复计算。
* 贪心（B）：每一步做局部最优选择，无显式分解合并。
* 搜索（D）：遍历状态空间，无系统性分解。

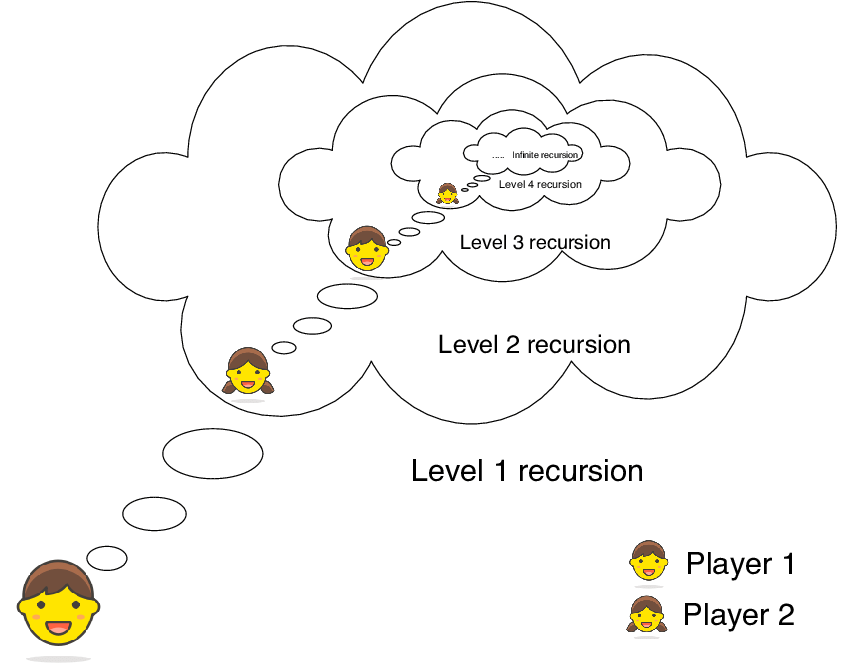
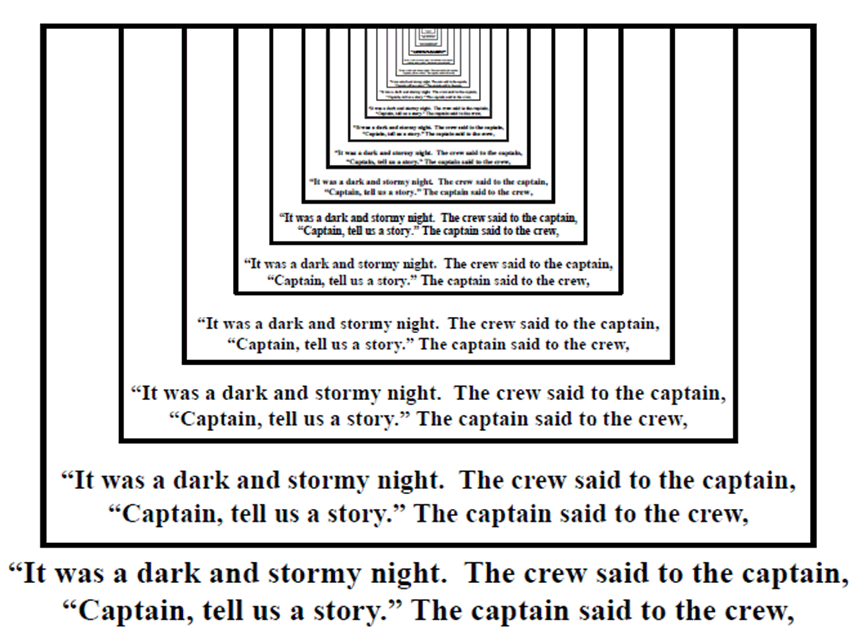
该题强调“分解子问题”和“子问题解的并”，符合分治法定义。

#### 3. 2013年第3题（算法）

下面的故事与（ ）算法有着异曲同工之妙。 从前有座山，山里有座庙，庙里有个老和尚在给小和尚讲故事：“从前有座山，山里有座庙，庙里有个老和尚在给小和尚讲故事：‘从前有座山，山里有座庙，庙里有个老和尚给小和尚讲故事……’”

A. 枚举

B. 递归

C. 贪心

D. 分治

**答案： B**

**解析：**

该故事通过**无限自我引用**（故事中嵌套相同故事）展示了递归的结构。递归算法中，函数直接或间接调用自身，将问题分解为更小的自身实例，直到达到基线条件（base case）。例如，计算阶乘：n! = n \* (n-1)!，基线条件为 1! = 1。

异曲同工之处：故事没有终止条件（无限递归），算法递归需有基线条件避免无限循环。

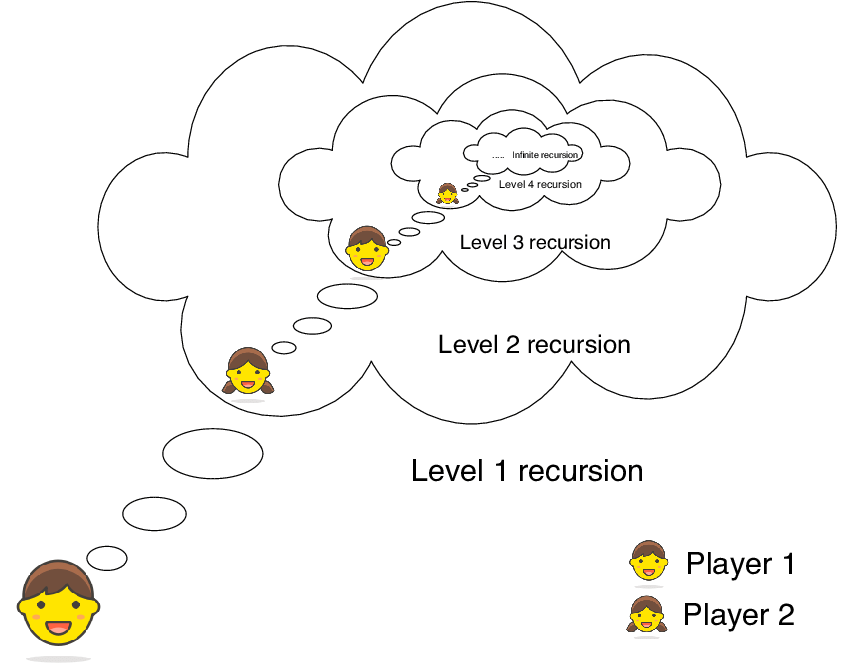
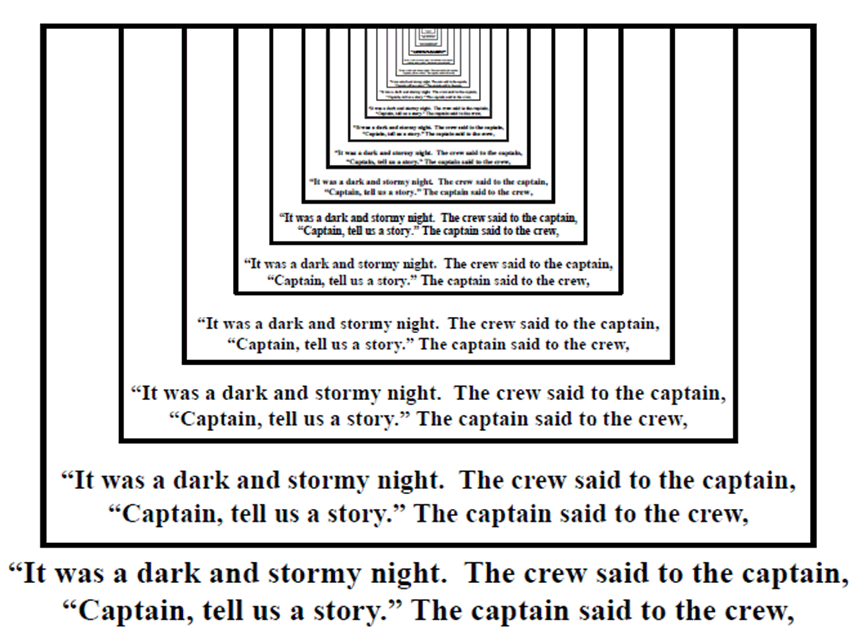
与其他选项区别：

* **枚举（A）**：无自我调用，仅遍历所有可能。
* **贪心（C）**：无递归结构，仅局部最优。
* **分治（D）**：虽用递归实现，但强调分解和合并，而该故事无合并操作。

故事的核心是“自我重复调用”，对应递归的本质。

#### 4. 2018年第10题（算法）

下面的故事与（ ）算法有着异曲同工之妙。 从前有座山，山里有座庙，庙里有个老和尚在给小和尚讲故事：“从前有座山，山里有座庙，庙里有个老和尚在给小和尚讲故事：‘从前有座山，山里有座庙，庙里有个老和尚给小和尚讲故事……’”

A. 枚举

B. 递归

C. 贪心

D. 分治

**答案： B**

**解析：**

此题目与2013年第3题完全相同，故事展示了**无限递归调用**：每个故事实例都生成一个新的相同实例。递归算法的核心是函数调用自身解决更小规模的子问题，直到满足基线条件。

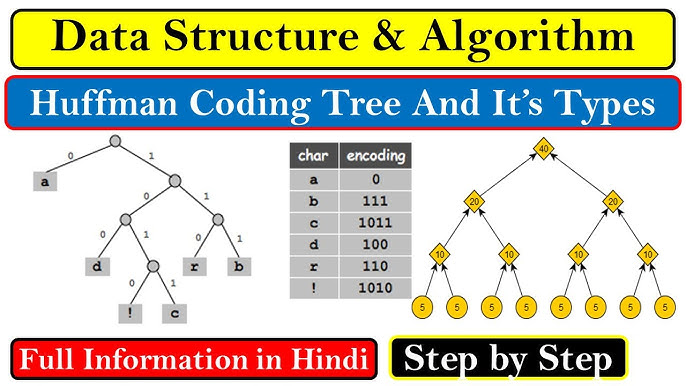
为什么是递归：故事描述的模式是“定义中包含自身”，如递归函数定义 f(n) = ... f(n-1) ...。

与其他选项区别：同2013年第3题解析（枚举、贪心、分治均不匹配该模式）。

尽管故事缺乏基线条件（导致无限循环），但结构上完美契合递归思想。

#### 5. 2021年第11题（哈弗曼树）

在数据压缩编码中的哈夫曼编码方法，在本质上是一种（ ）的策略。

A. 枚举

B. 贪心

C. 递归

D. 动态规划

**答案： B**

**解析：**

哈夫曼编码通过构建最优前缀码实现数据压缩，其本质是贪心策略。

算法步骤：

* 统计字符频率，作为叶子节点权重。
* 每次**合并频率最小的两个节点**，形成新节点（权重为子节点和）。
* 重复合并直到根节点，形成哈夫曼树。

贪心体现：每一步选择局部最优（最小频率节点合并），最终得到全局最优压缩率。

与其他选项区别：

* 枚举（A）：不适用，因哈夫曼不遍历所有可能树。
* 递归（C）：虽实现中可能用递归，但核心策略是贪心。
* 动态规划（D）：无重叠子问题或最优子结构。

哈夫曼编码的贪心性质被证明能产生最优前缀码。

#### 6. 2021年第15题（贪心算法）

有四个人要从 A 点坐一条船过河到 B 点，船一开始在 A 点。该船一次最多可坐两个人。 已知这四个人中每个人独自坐船的过河时间分别为 1,2,4,8，且两个人坐船的过河时间为两人独自过河时间的较大者。则最短（ ）时间可以让四个人都过河到 B 点（包括从 B 点把船开回 A 点的时间）。

A. 14

B. 15

C. 16

D. 17

**答案： B**

**解析：**

这是经典过河问题，需用贪心策略最小化总时间。最优序列如下（时间单位：分钟）：

* 时间1和2的人过河到B（耗时：max(1,2) = 2）。**累计时间：2**。
* 时间1的人返回A（耗时：1）。**累计时间：3**。
* 时间4和8的人过河到B（耗时：max(4,8) = 8）。累计时间：11。
* 时间2的人返回A（耗时：2）。**累计时间：13**。
* 时间1和2的人过河到B（耗时：max(1,2) = 2）。**累计时间：15**。

总时间15分钟。

贪心策略：最小化返回时间（让最快的人返回），并让最慢的人一起过河减少次数。

其他策略对比：

若先让1和4过河（耗时4），1返回（耗时1），1和8过河（耗时8），1返回（耗时1），1和2过河（耗时2），总时间16。

选项A(14)和D(17)均非最优。

该策略确保总时间最小，体现了贪心算法的局部最优导致全局最优。



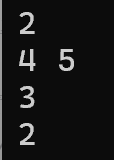
#### 7. 2017年第28题（切割绳子）

**完善程序：**

（切割绳子） 有 n 条绳子，每条绳子的长度已知且均为正整数。绳子可以以任意正整数长度切割，但不可以连接。现在要从这些绳子中切割出 m 条长度相同的绳段，求绳段的最大长度是多少。（第一、二空 2.5 分，其余 3 分）

输入：第一行是一个不超过 100 的正整数 n，第二行是 n 个不超过 10^6 的正整数，表示每条绳子的长度，第三行是一个不超过 10^8 的正整数 m。

输出：绳段的最大长度，若无法切割，输出 Failed。

#include<iostream>

using namespace std;

int n, m, i, lbound, ubound, mid, count;

int len[100]; // 绳子长度

int main()

{

cin >> n;

count = 0;

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

{

cin >> len[i];

【①】; //

}

cin >> m;

if (【②】) //

{

cout << "Failed" << endl;

return 0;

}

lbound = 1;

ubound = 1000000;

while (【③】) //

{

mid =【④】; //

count = 0;

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

【⑤】; //

if (count < m)

ubound = mid - 1;

else

lbound = mid;

}

cout << lbound << endl;

return 0;

}

**答案：**

**① count += len[i] 或 count = count + len[i]**

**② count < m 或 m > count**

**③ lbound < ubound 或 ubound > lbound**

**④ (lbound + ubound + 1) / 2 或等效表达式（如 (lbound + ubound + 1) >> 1）**

**⑤ count += len[i] / mid 或 count = count + len[i] / mid**

**解析：**

**“切割绳子”问题**：给定 n 条长度为正整数的绳子，需要切割出 m 条长度相同的绳段（绳段长度必须为正整数）。切割时，绳子可以任意切割但不能连接。程序的目标是找到绳段的**最大**可能长度，使得切割后至少能得到 m 条绳段。如果无法切割出 m 条绳段（例如总长度不足），则输出 Failed。

程序用**二分搜索**求最大绳段长度 mid，使得能切割出至少 m 条长度为 mid 的绳段。

示例：**n=2, len=[4,5], m=3。最大长度为2**（4切成2段2，5切成2段2和1段1，总段数3）。

**程序输入和输出**

* **输入**：
  + 第一行：一个不超过 100 的正整数 n，表示绳子的数量。
  + 第二行：n 个不超过 10^6的正整数，表示每条绳子的长度，存储在数组 len 中。
  + 第三行：一个不超过 10^8 的正整数 m，表示需要切割出的绳段数量。
* **输出**：
  + 绳段的最大长度（正整数）。
  + 如果无法切割出 m 条绳段（即使每条绳段长度为 1 也不行），则输出 Failed。

**程序整体逻辑**

* **计算绳子总长度并检查可行性**：**如果所有绳子的总长度小于 m，则无法切割出 m 条绳段（因为每条绳段至少需要长度 1）**，直接输出 Failed。
* **二分搜索最大绳段长度**：**如果总长度足够，程序使用二分搜索在范围 [1, 1000000] 内查找最大绳段长度 L，使得切割后绳段总数至少为 m**。二分搜索高效地缩小范围，直到找到最大 L。
* **输出结果**：二分搜索结束后，输出找到的最大长度 lbound。

**二分逻辑：**

* 若 count >= m，说明 mid 可行，增大下界（lbound = mid）尝试更大长度。
* 若 count < m，说明 mid 过长，减小上界（ubound = mid - 1）。
* 最终 lbound 为最大长度。

**贪心与二分**：二分法高效缩小搜索范围（O(log(max\_len))），贪心体现在段数计算（最大化每段长度）。

**① count += len[i] 或 count = count + len[i]**

**② count < m 或 m > count**

**③ lbound < ubound 或 ubound > lbound**

**④ (lbound + ubound + 1) / 2 或等效表达式（如 (lbound + ubound + 1) >> 1）**

**⑤ count += len[i] / mid 或 count = count + len[i] / mid**

①：计算所有绳子总长度 count（输入阶段）。

②：可行性检查——若总长度小于 m（每段至少长度1），则无法切割，输出 Failed。

③：二分搜索条件——当 lbound（下界）小于 ubound（上界）时继续搜索。

④：计算中点 mid，加1避免死循环（当 lbound 和 ubound 相差1时，确保 mid 取更大值）。

⑤：计算以长度 mid 切割时，能得到的总段数（每条绳子贡献 len[i] / mid 段）。

**完整程序**

#include<iostream>

using namespace std;

int n, m, i, lbound, ubound, mid, count;

int len[100]; // 存储绳子长度，最多100条绳子

int main() {

cin >> n; // 输入绳子数量 n

count = 0;

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

cin >> len[i]; // 输入每条绳子的长度

count += len[i]; // ①：累加所有绳子的总长度

}

cin >> m; // 输入需要切割出的绳段数量 m

if (count < m) { // ②：检查总长度是否小于 m

cout << "Failed" << endl; // 总长度不足，无法切割出 m 条绳段

return 0;

}

lbound = 1; // 二分搜索下界，绳段长度最小为 1

ubound = 1000000; // 二分搜索上界，因为绳子长度不超过 10^6，绳段长度最大可能为 10^6

while (lbound < ubound) { // ③：二分搜索主循环，当 lbound < ubound 时继续

mid = (lbound + ubound + 1) / 2; // ④：计算中间值，+1 确保向上取整，避免死循环

count = 0; // 重置 count，用于累加当前绳段长度 mid 下能切割出的总段数

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

count += len[i] / mid; // ⑤：对每条绳子，计算以长度 mid 切割能得到的段数（整数除法）

}

if (count < m) { // 如果总段数不足 m

ubound = mid - 1; // 绳段长度 mid 太大，需要减小上界

} else {

lbound = mid; // 绳段长度 mid 可行，尝试更大的长度（提高下界）

}

}

cout << lbound << endl; // 输出最大绳段长度

return 0;

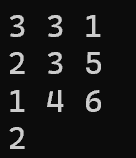
}

#### 8. 2016年第28题（郊游活动）

**完善程序：**

（郊游活动）有 **n 名同学**参加学校组织的郊游活动，已知学校给这 n 名同学的郊游**总经费为 A 元**，与此同时第 i 位同学自己携带了 **M\_i** 元。为了方便郊游，活动地点提供 **B (≥n)** 辆自行车供人租用，租用第 j 辆自行车的价格为 **C\_j 元**，每位同学可以使用自己携带的钱或者学校的郊游经费，为了方便账务管理，每位同学只能为自己租用自行车，且不会借钱给他人，他们想知道**最多有多少位同学**能够租用到自行车。（第四、五空 2.5 分，其余 3 分）

本题采用**二分法**。对于区间 [l, r]，我们取中间点 mid 并判断租用到自行车的人数能否达到 mid。判断的过程是利用**贪心算法**实现的。

#include <iostream>

using namespace std;

#define MAXN 1000000

int n, B, A, M[MAXN], C[MAXN], l, r, ans, mid;

bool check(int nn) {

int count = 0, i, j;

i = 【①】; //

j = 1;

while (i <= n) {

if (【②】) //

count += C[j] - M[i];

i++;

j++;

}

return 【③】; //

}

void sort(int a[], int l, int r) {

int i = l, j = r, x = a[(l + r) / 2], y;

while (i <= j) {

while (a[i] < x) i++;

while (a[j] > x) j--;

if (i <= j) {

y = a[i]; a[i] = a[j]; a[j] = y;

i++; j--;

}

}

if (i < r) sort(a, i, r);

if (l < j) sort(a, l, j);

}

int **main()** {

int i;

cin >> n >> B >> A;

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

cin >> M[i];

for (i = 1; i <= B; i++)

cin >> C[i];

sort(M, 1, n); // M 升序排序

sort(C, 1, B); // C 升序排序

l = 0;

r = n;

while (l <= r) {

mid = (l + r) / 2;

if (【④】) { //

ans = mid;

l = mid + 1;

} else

r = 【⑤】; //

}

cout << ans << endl;

return 0;

}

**答案：**

**① n - nn + 1**

**② M[i] < C[j]**

**③ count <= A**

**④ check(mid)**

**⑤ r = mid - 1**

**解析：**

**问题描述**

* 有 n 名同学参加郊游，学校总经费为 A 元
* 第 i 位同学自带 M\_i 元
* 有 B（≥n）辆自行车，第 j 辆自行车租金为 C\_j 元
* 每位同学只能为自己租车，可以使用自带钱或学校经费（可混合使用）
* 求最多能租到自行车的学生人数

**示例验证**

* 输入：n=3, M=[2,3,5], B=3, C=[1,4,6], A=1
* 排序：M=[2,3,5], C=[1,4,6]
* 二分过程：
  + mid=2 → check(2):
    - **取最富有2人**：M[2]=3, M[3]=5
    - **匹配最便宜2车**：C[1]=1, C[2]=4
    - 补贴：max(0,1-3)=0 + max(0,4-5)=0 = 0 ≤ 1 → 可行
  + mid=3 → check(3):
    - 补贴：max(0,1-2)=0 + max(0,4-3)=1 + max(0,6-5)=1 = 2 > 1 → 不可行
* 输出：ans=2

**程序思路**

程序采用 **二分答案 + 贪心验证** 策略：

* 二分答案：在范围 [0, n] 内二分查找最大租车人数
* 贪心验证：对每个候选人数 mid，检查是否能满足 mid 人租车

**关键函数：check(int nn)：**验证**是否能让 nn 位同学租到自行车**：

bool check(int nn) {

int count = 0, i, j;

i = n - nn + 1; // ①：定位到最富有的nn位同学的起始位置

j = 1; // 定位到最便宜的nn辆自行车的起始位置

while (i <= n) {

if (M[i] < C[j]) // ②：如果自带钱不足

count += C[j] - M[i]; // 计算需要补贴的金额

i++;

j++;

}

return count <= A; // ③：总补贴是否在预算内

}

**代码解析**

**① i = n - nn + 1**

* 背景：M 数组已按升序排序（M[1]最小，M[n]最大）
* 作用：计算最富有的 nn 位同学在数组中的起始位置
* 示例：当 n=5, nn=3 时，取 M[3], M[4], M[5]（最富有的3人）

**② if (M[i] < C[j])**

* 贪心匹配策略：**最不富有的同学 → 最便宜的自行车**
* 原因：**最小化学校补贴**（富余资金匹配高价车会浪费补贴机会）
* 计算：若同学资金不足，记录补贴金额 C[j] - M[i]

**③ return count <= A**

* 决策条件：总补贴金额 ≤ 学校总经费 A
* 满足条件 → nn 人租车方案可行

**主程序逻辑**

int main() {

// 输入处理

sort(M, 1, n); // 同学资金升序排序

sort(C, 1, B); // 自行车租金升序排序

// 二分查找最大租车人数

l = 0; r = n;

while (l <= r) {

mid = (l + r) / 2;

if (check(mid)) { // ④：验证mid人是否可行

ans = mid; // 记录可行解

l = mid + 1; // 尝试更大人数

} else {

r = mid - 1; // ⑤：人数过多，缩减上限

}

}

cout << ans;

}

**④ if (check(mid))**

* 关键决策：检查 mid 人租车方案是否可行
* 若可行 → 更新答案并尝试增加人数（l = mid + 1）

**⑤ r = mid - 1**

关键调整：当 mid 人方案不可行时

缩减人数上限（r = mid - 1），尝试更小规模方案

**贪心策略正确性证明**

假设有两位同学 a, b（M\_a ≤ M\_b）和两辆车 x, y（C\_x ≤ C\_y）：

* 最优匹配：a→x, b→y
  + 补贴 = max(0, C\_x - M\_a) + max(0, C\_y - M\_b)
* 错误匹配：a→y, b→x
  + 补贴 = max(0, C\_y - M\_a) + max(0, C\_x - M\_b)

∵ C\_y - M\_a ≥ C\_x - M\_a 且 C\_y - M\_a ≥ C\_y - M\_b

∴ 错误匹配的补贴 ≥ 最优匹配的补贴

⇒ 贪心匹配策略可最小化学校补贴

**复杂度分析**

* 排序：O(n log n + B log B)
* 二分查找：O(log n) 次迭代
* check() 函数：每次 O(n)，总复杂度：O(n log n + B log B)（满足题目约束）

**完整程序**

#include <iostream>

using namespace std;

#define MAXN 1000000

int n, B, A, M[MAXN], C[MAXN], l, r, ans, mid;

bool check(int nn) {

int count = 0, i, j;

i = n - nn + 1; // ①

j = 1;

while (i <= n) {

if (M[i] < C[j]) // ②

count += C[j] - M[i];

i++;

j++;

}

return count <= A; // ③

}

void sort(int a[], int l, int r) {

int i = l, j = r, x = a[(l + r) / 2], y;

while (i <= j) {

while (a[i] < x) i++;

while (a[j] > x) j--;

if (i <= j) {

y = a[i]; a[i] = a[j]; a[j] = y;

i++; j--;

}

}

if (i < r) sort(a, i, r);

if (l < j) sort(a, l, j);

}

int main() {

int i;

cin >> n >> B >> A;

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

cin >> M[i];

for (i = 1; i <= B; i++)

cin >> C[i];

sort(M, 1, n); // M 升序排序

sort(C, 1, B); // C 升序排序

l = 0;

r = n;

while (l <= r) {

mid = (l + r) / 2;

if (check(mid)) { // ④

ans = mid;

l = mid + 1;

} else

r = mid - 1; // ⑤

}

cout << ans << endl;

return 0;

}

#### 9. 2020年第20题（最小区间覆盖）

**完善程序：**

（最小区间覆盖）给出 n 个区间，第 i 个区间的左右端点是 [a\_i, b\_i]。现在要在这些区间中选出若干个，使得**区间 [0, m] 被所选区间的并覆盖**（即每一个 0≤i≤m 都在某个所选的区间中）。保证答案存在，求所选区间个数的最小值。

输入第一行包含两个整数 n 和 m（1≤n≤5000, 1≤m≤10^9）

接下来 n 行，每行两个整数 a\_i, b\_i（0≤a\_i, b\_i≤m）。

提示：使用贪心法解决这个问题。先用 O(n^2) 的时间复杂度排序，然后贪心选择这些区间。

试补全程序。

#include <iostream>

using namespace std;

const int MAXN = 5000;

int n, m;

struct segment { int a, b; } A[MAXN];

**void sort()** {

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

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

if (【①】) { //

segment t = A[j];

【②】

}

}

**int main()** {

cin >> n >> m;

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

cin >> A[i].a >> A[i].b;

sort();

int p = 1;

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

if (【③】) //

A[p++] = A[i];

n = p;

int ans = 0, r = 0;

int q = 0;

while (r < m) {

while (【④】) //

q++;

【⑤】; //

ans++;

}

cout << ans << endl;

return 0;

}

1.

A. A[j].b>A[j-1].b

B. A[j].a<A[j-1].a

C. A[j].a>A[j-1].a

D. A[j].b<A[j-1].b

2.

A. A[j+1]=A[j];A[j]=t;

B. A[j-1]=A[j];A[j]=t;

C. A[j]=A[j+1];A[j+1]=t;

D. A[j]=A[j-1];A[j-1]=t;

3.

A. A[i].b>A[p-1].b

B. A[i].b<A[i-1].b

C. A[i].b>A[i-1].b

D. A[i].b<A[p-1].b

4.

A. q+1<n&&A[q+1].a<=r

B. q+1<n&&A[q+1].b<=r

C. q<n&&A[q].a<=r

D. q<n&&A[q].b<=r

5.

A. r=max(r,A[q+1].b)

B. r=max(r,A[q].b)

C. r=max(r,A[q+1].a)

D. q++

**答案：**

**① A[j].a < A[j-1].a**

**② A[j-1] = t**

**③ A[i].b > A[p-1].b**

**④ q < n && A[q].a <= r**

**⑤ max(r, A[q-1].b)**

**解析：**

贪心算法求解最小区间覆盖。

**排序（sort 函数）**：

①：按区间**左端点 a 升序排序**（冒泡排序条件）。

②：交换元素实现排序。

**去冗余：**

③：保留**右端点更大的区间**（若当前区间右端点大于已选最大，则加入）。

**区间覆盖**：

④：扫描所有**左端点 ≤ 当前覆盖点 r 的区间**（q 指针移动）。

⑤：选择这些区间中**右端点最大的**，更新 r 为该右端点（max(r, A[q-1].b)）。

ans 计数区间数。

**贪心策略**：每次选择覆盖当前 r 且延伸最远的区间，保证局部最优。

**复杂度**：排序 O(n²)，贪心 O(n)。

示例：n=3, m=10, A=[[0,3],[2,6],[5,10]]。去冗余后 A=[[0,3],[2,6],[5,10]]（无冗余），覆盖：选[2,6]（r=0→6），选[5,10]（r=6→10），ans=2。

**完整程序**

#include <iostream>

using namespace std;

const int MAXN = 5000;

int n, m;

struct segment { int a, b; } A[MAXN];

void sort() {

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

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

if (A[j].a < A[j-1].a) { // ①

segment t = A[j];

A[j] = A[j-1]; // ②

A[j-1] = t;

}

}

int main() {

cin >> n >> m;

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

cin >> A[i].a >> A[i].b;

sort();

int p = 1;

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

if (A[i].b > A[p-1].b) // ③

A[p++] = A[i];

n = p;

int ans = 0, r = 0;

int q = 0;

while (r < m) {

while (q < n && A[q].a <= r) // ④

q++;

r = max(r, A[q-1].b); // ⑤

ans++;

}

cout << ans << endl;

return 0;

}

#### 10. 2010年第28题（过河问题）

**完善程序：（过河问题）**

在一个月黑风高的夜晚，有一群人在河的右岸，想通过唯一的一根独木桥走到河的左岸。在这伸手不见五指的黑夜里，过桥时必须借助灯光来照明，很不幸的是，他们只有一盏灯。另外，独木桥上最多承受两个人同时经过，否则将会坍塌。

每个人单独过桥都需要一定的时间，不同的人需要的时间可能不同。两个人一起过桥时，由于只有一盏灯，所以需要的时间是较慢的那个人单独过桥时所花的时间。

现输入 n（2 ≤ n < 100）和这 n 个人单独过桥时需要的时间，请计算总共最少需要多少时间，他们才能全部到达河的左岸。

例如，有 3 个人甲、乙、丙，他们单独过桥的时间分别为 1, 2, 4，则总共最少需要的时间为 7。

具体方法是：甲、乙一起过桥到河的左岸，甲单独回到河的右岸将灯带回，然后甲、丙再一起过桥到河的左岸，总时间为 2 + 1 + 4 = 7。

#include <iostream>

using namespace std;

const int SIZE = 100;

const int INFINITY = 10000;

const bool LEFT = true;

const bool RIGHT = false;

const bool LEFT\_TO\_RIGHT = true;

const bool RIGHT\_TO\_LEFT = false;

int n, hour[SIZE];

bool pos[SIZE];

int max(int a, int b) {

return (a > b) ? a : b;

}

int go(bool stage) {

int i, j, num, tmp, ans;

 if (stage == RIGHT\_TO\_LEFT) {

num = 0;

ans = 0;

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

if (pos[i] == RIGHT) {

num++;

if (hour[i] > ans)

ans = hour[i];

 }

if (【①】) //

return ans;

ans = INFINITY;

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

if (pos[i] == RIGHT)

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

if (pos[j] == RIGHT) {

pos[i] = LEFT;

pos[j] = LEFT;

tmp = max(hour[i], hour[j]) + 【②】; //

if (tmp < ans)

ans = tmp;

pos[i] = RIGHT;

pos[j] = RIGHT;

 }

return ans;

}

if (stage == LEFT\_TO\_RIGHT) {

ans = INFINITY;

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

if (【③】) { //

pos[i] = RIGHT;

tmp = 【④】; //

if (tmp < ans)

ans = tmp;

【⑤】; //

}

return ans;

}

return 0;

}

int main() {

int i;

cin>>n;

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

cin>>hour[i];

pos[i] = RIGHT;

}

cout<<go(RIGHT\_TO\_LEFT)<<endl;

return 0;

}

**答案：**

**① num <= 2**

**② go(LEFT\_TO\_RIGHT)**

**③ pos[i] == LEFT**

**④ hour[i] + go(RIGHT\_TO\_LEFT)**

**⑤ pos[i] = LEFT**

**解析：**

递归 + 贪心解决过河问题。go(stage) 函数计算当前方向的最小时间。

**RIGHT\_TO\_LEFT（多人过河）：**

* ①：**若人数 ≤ 2**，直接过河，时间为最慢者时间。
* ②：枚举两人组合过河，递归调用灯返回（go(LEFT\_TO\_RIGHT)），时间为**两人时间最大值加返回时间**。

**LEFT\_TO\_RIGHT（灯返回）：**

* ③：选择**一人带灯返回**（需在左岸）。
* ④：递归调用多人过河（go(RIGHT\_TO\_LEFT)），时间为**返回者时间加过河时间**。
* ⑤：回溯恢复状态（**重置位置**）。

**贪心策略**：实际最优解常让最快者往返以减少时间（如2021年第15题），但本程序通过递归枚举所有可能，结合贪心剪枝（取最小 tmp）。

**复杂度**：最坏 O(n!)，但 n 小可行（如 n=4）。

示例：n=3, hour=[1,2,4]。最优序列：1和2过（2），1返回（1），1和4过（4），总时间7（见程序输出）。

**完整程序**

#include <iostream>

using namespace std;

const int SIZE = 100;

const int INFINITY = 10000;

const bool LEFT = true;

const bool RIGHT = false;

const bool LEFT\_TO\_RIGHT = true;

const bool RIGHT\_TO\_LEFT = false;

int n, hour[SIZE];

bool pos[SIZE];

int max(int a, int b) {

return (a > b) ? a : b;

}

int go(bool stage) {

int i, j, num, tmp, ans;

 if (stage == RIGHT\_TO\_LEFT) {

num = 0;

ans = 0;

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

if (pos[i] == RIGHT) {

num++;

if (hour[i] > ans)

ans = hour[i];

 }

if (num <= 2) // ①

return ans;

ans = INFINITY;

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

if (pos[i] == RIGHT)

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

if (pos[j] == RIGHT) {

pos[i] = LEFT;

pos[j] = LEFT;

tmp = max(hour[i], hour[j]) + go(LEFT\_TO\_RIGHT); // ②

if (tmp < ans)

ans = tmp;

pos[i] = RIGHT;

pos[j] = RIGHT;

 }

return ans;

}

if (stage == LEFT\_TO\_RIGHT) {

ans = INFINITY;

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

if (pos[i] == LEFT) { // ③

pos[i] = RIGHT;

tmp = hour[i] + go(RIGHT\_TO\_LEFT); // ④

if (tmp < ans)

ans = tmp;

pos[i] = LEFT; // ⑤

}

return ans;

}

return 0;

}

int main() {

int i;

cin>>n;

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

cin>>hour[i];

pos[i] = RIGHT;

}

cout<<go(RIGHT\_TO\_LEFT)<<endl;

return 0;

}

# 十、二分查找

### 考点分析

|  |  |  |  |
| --- | --- | --- | --- |
| **结构类型** | **重点考点** | **频率** | **难度** |
| **基础理论** | **时间复杂度计算** | 35% | ★★ |
| **查找长度分析** | 25% | ★★ |
| **二分前提条件** | 20% | ★ |
| **循环结构** | **终止条件** | 30% | ★★★ |
| **边界更新** | 25% | ★★★★ |
| **中点计算防溢出** | 15% | ★★ |
| **应用变体** | **二分答案** | 25% | ★★★★ |
| **单峰序列极值** | 15% | ★★★★ |
| **缺失值查找** | 10% | ★★★ |
| **实数二分** | 10% | ★★★ |
| **实现细节** | **判定函数设计** | 20% | ★★★★ |
| **边界初始化** | 15% | ★★ |
| **死循环预防** | 10% | ★★★ |
| **特殊场景** | **矩阵搜索** | 10% | ★★★★ |
| **牛顿迭代法** | 5% | ★★★ |
| **贪心+二分** | 10% | ★★★★ |
| **常见错误** | **边界条件错误** | 30% | ★★★ |
| **整数溢出** | 15% | ★★ |
| **重复元素处理** | 10% | ★★★ |

### 核心知识点

#### 1. 基础理论

* **核心知识点：**
  + 时间复杂度：**O(log n)**，查找次数 = floor(log₂n) + 1
  + 前提条件：数据必须有序（单调性）
  + 空间复杂度：**O(1)**（迭代实现）
* **黄金法则：**
  + 每次迭代都将搜索范围减半
  + 查找次数由数据规模决定，与具体数值无关
* **高频公式：**

// 最大比较次数计算

int max\_comparisons = floor(log2(n)) + 1;

* **易错点：**
  + 未验证数据是否有序直接使用二分
  + 忽略log₂n计算时的取整问题
  + 混淆时间复杂度O(log n)与O(n log n)

#### 2. 循环结构与边界处理

* **核心知识点：**
  + **两种循环结构：**

while(left <= right)：标准闭区间

while(left < right)：左闭右开区间

* + **边界更新策略：**

查找值：left = mid+1 / right = mid-1

查找边界：left = mid / right = mid

* **黄金法则：**
  + 更新left时用mid+1
  + 更新right时用mid-1或mid
  + 保持循环不变量（区间定义）
* **高频公式：**

// 防溢出中点计算

int mid = left + (right - left) / 2;

// 向上取整中点

int mid = left + (right - left + 1) / 2;

* **易错点：**
  + 死循环（更新边界时未缩小范围）
  + 差一错误（边界条件处理不当）
  + 整数溢出（(left+right)可能溢出）
* **标准模板：**

// 查找目标值

int binary\_search(vector<int>& nums, int target) {

int left = 0, right = nums.size()-1;

while(left <= right) {

int mid = left + (right-left)/2;

if(nums[mid] == target) return mid;

else if(nums[mid] < target) left = mid+1;

else right = mid-1;

}

return -1;

}

#### 3. 二分答案

* **核心知识点：**
  + 将最优化问题转化为判定问题
  + 要求：判定函数具有单调性
  + 常见应用：**最大值最小化/最小值最大化**
* **黄金法则：**
  + 满足条件时尝试更大值（求最大值）
  + 不满足条件时缩小值范围
* **高频公式：**

// 最大值最小化框架

while(left < right) {

int mid = (left + right) / 2;

if(check(mid)) right = mid;

else left = mid+1;

}

// 最小值最大化框架

while(left < right) {

int mid = (left + right + 1) / 2;

if(check(mid)) left = mid;

else right = mid-1;

}

* **易错点：**
  + 判定函数设计错误
  + 未正确处理边界初始化
  + 混淆最大值/最小值问题框架
* **经典模板：**

// 切割绳子问题（2017-28）

int max\_cut\_length(vector<int>& lens, int m) {

int left = 1, right = \*max\_element(lens.begin(), lens.end());

while(left < right) {

int mid = (left + right + 1) / 2;

int count = 0;

for(int len : lens) count += len / mid;

if(count >= m) left = mid;

else right = mid-1;

}

return left;

}

#### 4. 单峰序列处理

* **核心知识点：**
  + 序列先升后降（或先降后升）
  + 比较中点与相邻元素关系
  + 确定搜索方向（左/右）
* **黄金法则：**
  + 中点>左邻且>右邻 → 找到峰值
  + 中点>左邻但<右邻 → 峰值在右侧
  + 否则 → 峰值在左侧
* **高频公式：**

// 单峰序列查找（2016-14）

if(nums[mid] > nums[mid-1] && nums[mid] > nums[mid+1])

return mid;

else if(nums[mid] > nums[mid-1] && nums[mid] < nums[mid+1])

left = mid+1;

else

right = mid-1;

* **易错点：**
  + 未处理边界情况（峰在起点/终点）
  + 错误判断升降趋势
  + 忽略相邻元素比较时的越界风险

#### 5. 实数二分

* **核心知识点：**
  + 处理浮点数问题
  + 设置精度阈值（epsilon）
  + 牛顿迭代法加速收敛
* **黄金法则：**
  + 使用相对精度控制
  + 避免浮点数相等判断
  + 限制最大迭代次数
* **高频公式：**

// 实数二分框架（2022-18）

double binary\_search\_double(double n) {

double left = 0, right = n;

const double eps = 1e-7;

while(**fabs(right - left) > eps**) {

double mid = (left + right) / 2;

if(mid\*mid > n) right = mid;

else left = mid;

}

return left;

}

// 牛顿迭代法（2022-18）

double newton\_sqrt(double x, int k) {

double res = x;

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

res = (res + x/res) / 2;

return res;

}

* **易错点：**
  + 精度设置不当导致死循环
  + 忽略浮点数计算误差
  + 牛顿迭代初值选择错误

#### 6. 特殊应用

* **核心知识点：**
  + 矩阵搜索（二维转一维）
  + 缺失值查找（等差数列）
  + 贪心+二分组合
* **黄金法则：**
  + 二维坐标映射：index = row\*cols + col
  + 等差数列：a[i] = a[0] + i\*d
  + 贪心验证函数需高效
* **高频公式：**

// 矩阵搜索（2021-20）

bool matrix\_search(vector<vector<int>>& mat, int target) {

int m = mat.size(), n = mat[0].size();

int left = 0, right = m\*n-1;

while(left <= right) {

int mid = left + (right-left)/2;

int val = mat[mid/n][mid%n];

if(val == target) return true;

if(val < target) left = mid+1;

else right = mid-1;

}

return false;

}

// 等差数列缺失值（2023-19）

int find\_missing(vector<int>& nums) {

int left = 0, right = nums.size()-1;

while(left < right) {

int mid = left + (right-left)/2;

if(nums[mid] == nums[0] + mid)

left = mid+1;

else

right = mid;

}

return nums[0] + left;

}

* **易错点：**
  + 二维坐标映射错误
  + 忽略等差数列首项处理
  + 贪心函数与二分框架不匹配

#### 7. 致命错误防范

* **核心防御点：**

// 整数溢出防护

int mid = left + (right - left)/2; // 正确

int mid = (left + right)/2; // 危险！

// 边界防护

if(nums.empty()) return -1; // 空数组检查

if(target < nums[0]) return -1; // 小于最小值

if(target > nums.back()) return -1; // 大于最大值

// 死循环防护

int count = 0;

while(left < right && ++count < 50) { // 限制迭代次数

// ...

}

* **高频错误：**
  + 循环条件与边界更新不匹配
  + 忽略整数溢出风险
  + 未处理空数组或极端值
  + 判定函数缺少单调性
  + 混淆查找值与查找边界

### 真题强化

#### 1、2008年第15题（二分查找基础）

对有序数组 {5, 13, 19, 21, 37, 56, 64, 75, 88, 92, 100} 进行二分查找，成功查找元素 19 的查找长度（比较次数）是（ ）。

A. 1

B. 2

C. 3

D. 4

**答案： C**

**解析：**

* **二分查找步骤**：
  + 中间位置索引 **mid = (0+10)/2 = 5**，元素为 56，比较 19 < 56 → 查找左半部分（索引 0~4）。
  + 新中间位置 **mid = (0+4)/2 = 2**，元素为 19，比较 19 == 19 → 找到目标。
* **比较次数**：共 3 次（第一次比较 56，第二次比较 13，第三次比较 19）。
* **关键**：每次比较缩小搜索范围，直到找到目标或范围为空。

#### 2、2009年第16题（二分查找基础）

有一个由 4000 个整数构成的顺序表，假定表中的元素已经按升序排列，采用二分查找定位一个元素。则最多需要几次比较就能确定是否存在所查找的元素：

A. 11 次

B. 12 次

C. 13 次

D. 14 次

**答案： B**

**解析：**

* **公式**：二分查找最坏比较次数为 ⌊⌋+1。
* **计算**：
  + n=4000， ≈ 11.97，取整后 =11。
  + 比较次数 = 11+1=12。
* **验证**：
  + 2 ^11 =2048<4000（11 次无法覆盖）。
  + 2 ^12 =4096≥4000（12 次可覆盖所有元素）。

#### 3、2014年第18题（二分查找基础）

设有 100 个数据元素，采用折半搜索时，最大比较次数为（ ）。

A. 6

B. 7

C. 8

D. 10

**答案： B**

**解析：**

* **公式**：最大比较次数 = ⌊ ⌋+1。
* **计算**：
  + n=100， ≈ 6.64，取整后 ⌊ ⌋=6。
  + 比较次数 = 6+1=7。
* **验证**：
  + 2^6 =64<100（6 次不足）。
  + 2^7=128≥100（7 次足够）。

#### 4、2013年第25题（二分查找应用）

阅读程序写结果：

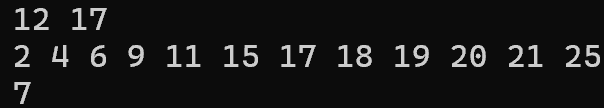
#include <iostream>

using namespace std;

int main() {

const int SIZE = 100;

int n, f, i, left, right, middle, a[SIZE];

 cin >> n >> f;

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

cin >> a[i];

left = 1;

right = n;

do {

middle = (left + right) / 2;

if (f <= a[middle])

right = middle;

else

left = middle + 1;

} while (left < right);

cout << left << endl;

return 0;

}

**输入：**

12 17

2 4 6 9 11 15 17 18 19 20 21 25

**答案： 7**

**解析：**

* **程序功能**：在有序数组中查找第一个 大于等于 f（17）的元素位置（左闭右闭区间）。
* **二分过程**：
  + 初始：left=1, right=12。
  + 循环：
    - mid=6 → a[6]=15<17 → left=7。
    - mid=9 → a[9]=19≥17 → right=9。
    - mid=8 → a[8]=18≥17 → right=8。
    - mid=7 → a[7]=17≥17 → right=7。
  + 终止条件：left=7, right=7 → 退出循环。
* **输出**：left=7（17 位于数组第 7 个位置）。

#### 5、2016年第14题（二分查找应用：单峰序列）

给定含有 n 个不同的数的数组 **L = <x1, x2, ..., xn>**。

如果 L 中存在某个 xi (1 < i < n)，满足以下条件：

**x1 < x2 < ... < xi−1 < xi > xi+1 > ... > xn**

则称 L 是**单峰的**，并称 xi 是 L 的“**峰顶**”。

现在已知 L 是单峰的，请将 a–c 三行代码补全到下方算法中，使得该算法能够正确地找到 L 的峰顶。

代码补全选项如下：

a. Search(k+1, n)

b. Search(1, k-1)

c. return L[k]

以下是需要补全的算法：

Search(1, n)

1. k ← ⌊n / 2⌋

2. if L[k] > L[k-1] and L[k] > L[k+1]

3. then \_\_\_\_\_\_\_\_\_\_

4. else if L[k] > L[k-1] and L[k] < L[k+1]

5. then \_\_\_\_\_\_\_\_\_\_

6. else \_\_\_\_\_\_\_\_\_\_

【选择题】

正确的填空顺序是（）。

A. c,a,b

B. c,b,a

C. a,b,c

D. b,a,c

**答案： A**

**解析：**

* **算法逻辑**：
  + 情况 1：L[k] 是峰顶（大于左右邻居）→ 直接返回（填 c）。
  + 情况 2：L[k] 处于上升段（大于左侧、小于右侧）→ 峰顶在右侧（填 a：Search(k+1, n)）。
  + 情况 3：L[k] 处于下降段（小于左侧）→ 峰顶在左侧（填 b：Search(1, k-1)）。
* **示例**：序列 [1,3,5,4,2]，取 k=3：
  + L[3]=5 > L[2]=3 且 L[3]=5 > L[4]=4 → 返回 c（峰顶）。

#### 6、2015年第28题（二分查找应用中位数）

**完善程序（中位数 median）：**

给定 n 个整数（n 为奇数，且 n < 1000），每个整数的取值范围在 0 ∼ m（0 < m < 2^31）之间。

请使用**二分法**求这 n 个整数的中位数。

所谓**中位数**，是指将这 n 个数排序之后，**排在正中间位置的那个数**。

（第五空 2 分，其余每空 3 分）。

#include <iostream>

using namespace std;

const int MAXN = 1000;

int n, i, lbound, rbound, mid, m, count;

int x[MAXN];

**int main() {**

cin >> n >> m;

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

cin >> x[i];

lbound = 0;

rbound = m;

while (【 ① 】) //

{

mid = (lbound + rbound) / 2;

【②】; //

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

if (【 ③】) //

【④】; //

if (count > n / 2)

lbound = mid + 1;

else

【⑤】; //

}

cout << rbound << endl;

return 0;

}

**答案：**

**① lbound < rbound**

**② count = 0**

**③ x[i] > mid**

**④ count++**

**⑤ rbound = mid**

**解析：**

* **算法原理**：二分法求中位数（即满足大于该值的元素不超过一半的最小值）。
* **关键步骤**：
  + ① 循环条件：lbound < rbound（未收敛时继续二分）。
  + ② 初始化计数器：count = 0。
  + ③④ 统计大于 mid 的元素个数（x[i] > mid 时 count++）。
  + ⑤ 若大于 mid 的元素超过一半，说明中位数在右侧（lbound=mid+1），否则在左侧（rbound=mid）。
* **输出**：最终 rbound 即为中位数。

**完整程序**

#include <iostream>

using namespace std;

const int MAXN = 1000;

int n, i, lbound, rbound, mid, m, count;

int x[MAXN];

**int main() {**

cin >> n >> m;

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

cin >> x[i];

lbound = 0;

rbound = m;

while (lbound < rbound) // ①

{

mid = (lbound + rbound) / 2;

count = 0; // ②

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

if (x[i] > mid) // ③

count++; // ④

if (count > n / 2)

lbound = mid + 1;

else

rbound = mid; // ⑤

}

cout << rbound << endl;

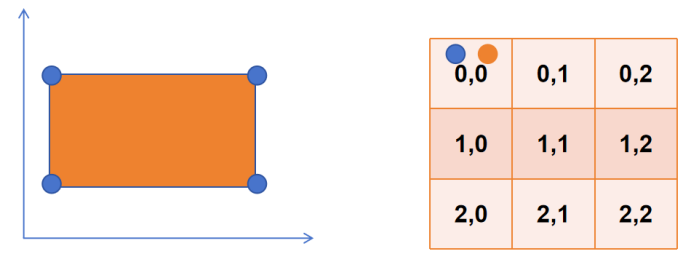
return 0;

}

#### 7、2021年第20题（二分查找应用：矩形计数）

**补全代码（矩形计数）**

平面上有 n 个关键点，求有多少个四条边都和 x 轴或者 y 轴平行的矩形，满足四个顶点都是关键点。给出的关键点可能有重复，但完全重合的矩形只计一 次。

#include <iostream>

using namespace std;

**struct point** {

int x, y, id;

};



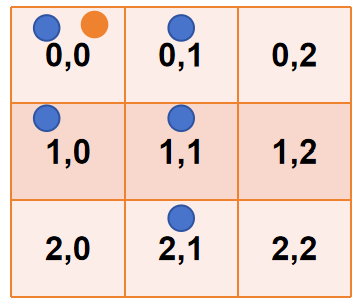
**bool equals**(point a, point b) {

return a.x == b.x && a.y == b.y;

}

**bool cmp**(point a, point b) {

return ①;

}

**void sort**(point A[], int n) {

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

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

if (cmp(A[j], A[j - 1])) {

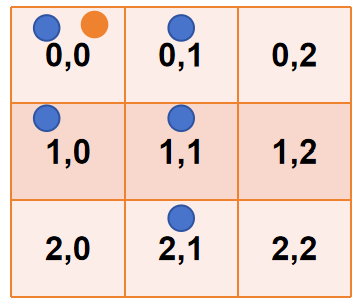
point t = A[j];

A[j] = A[j - 1];

A[j - 1] = t;

}

}



**int unique**(point A[], int n) {

int t = 0;

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

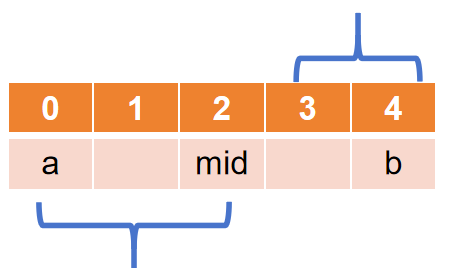
if (②)

A[t++] = A[i];

return t;

}

**bool binary\_search**(point A[], int n, int x, int y) {

 point p;

p.x = x;

p.y = y;

p.id = n;

int a = 0, b = n - 1;

while (a < b) {

int mid = ③;

if (④)

a = mid + 1;

else

b = mid;

}

return equals(A[a], p);

}

const int MAXN = 1000;

point A[MAXN];

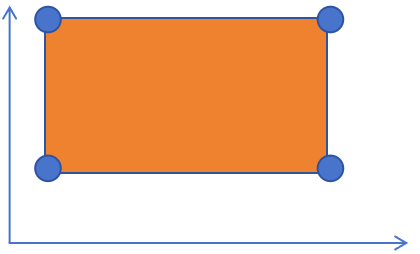
**int main()** {

int n;

cin >> n;

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

cin >> A[i].x >> A[i].y;

 A[i].id = i;

}

sort(A, n);

n = unique(A, n);

int ans = 0;

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

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

if (⑤ && binary\_search(A, n, A[i].x, A[j].y) &&

binary\_search(A, n, A[j].x, A[i].y)) {

ans++;

}

cout << ans << endl;

return 0;

}

1

A. a.x != b.x ? a.x < b.x : a.id < b.id

B. a.x != b.x ? a.x < b.x : a.y < b.y

C. equals(a, b) ? a.id < b.id : a.x < b.x

D. equals(a, b) ? a.id < b.id : (a.x != b.x ? a.x < b.x : a.y < b.y)

2.

A. i == 0 || cmp(A[i], A[i - 1])

B. t == 0 || equals(A[i], A[t - 1])

C. i == 0 || !cmp(A[i], A[i - 1])

D. t == 0 || !equals(A[i], A[t - 1])

3.

A. b - (b - a) / 2 + 1

B. (a + b + 1) >> 1

C. (a + b) >> 1

D. a + (b - a + 1) / 2

4.

A. !cmp(A[mid], p)

B. cmp(A[mid], p)

C. cmp(p, A[mid])

D. !cmp(p, A[mid])

5.

A. A[i].x == A[j].x

B. A[i].id < A[j].id

C. A[i].x == A[j].x && A[i].id < A[j].id

D. A[i].x < A[j].x && A[i].y < A[j].y

**答案：**

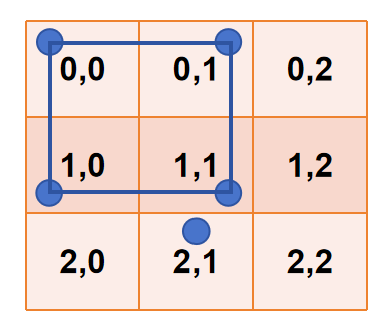
1. **B a.x != b.x ? a.x < b.x : a.y < b.y**
2. **D t == 0 || equals(A[i], A[t - 1])**

**③ C (a + b) >> 1**

**④ B cmp(A[mid], p)**

**⑤ D A[i].x < A[j].x && A[i].y < A[j].y**

**解析：**

* **算法目标**：统计以给定点为顶点的矩形数量（需验证四个顶点存在）。
* **假设输入有** 5 个点：
  + 5
  + 0 0
  + 0 1
  + 1 0
  + 1 1
  + 2 1
  + 程序首先将这些点排序并去重（实际在这里没有重复点）。
  + 接着程序会检查点对 ( (0,0) ) 和 ( (1,1) )，发现存在 ( (0,1) ) 和 ( (1,0) )，因此确认形成了一个矩形。
  + 最终程序输出结果为 1，表示有一个矩形可以形成。
* **关键步骤**：
  + ① 排序规则：先按 x 升序，x 相同按 y 升序。
  + ③ 二分中点计算：mid = (a+b)/2（位运算 >>1 等价除法）。
  + ④ 比较中点与目标点：若 cmp(A[mid], p) 为真（中点小于目标），向右搜索（a=mid+1）。
  + ⑤ 枚举对角点 (i,j) 需满足：A[i] 是左下角，A[j] 是右上角（即 xi < xj 且 yi < yj）。

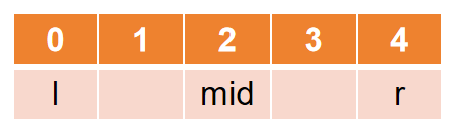
#### 8、2022年第18题（二分查找求平方根）

阅读程序（求平方根）：

#include <iostream>

using namespace std;

int n, k;

int solve1() {

int l = 0, r = n;

while(l <= r) {

int mid = (l + r) / 2;

if (mid \* mid <= n) l = mid + 1;

else r = mid - 1;

}

return l - 1;

}

double solve2(double x) {

if (x == 0) return x;

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

x = (x + n / x) / 2;

return x;

}

int main() {

cin >> n >> k;

double ans = solve2(solve1());

cout << ans << ' ' << (ans \* ans == n) << endl;

return 0;

}

假设 int 为32位有符号整数类型，输入的 n 是不超过47000的自然数、k 是不超过 int 表示范围的自然数，完成下面的判断题和单选题：

**判断题：**

1、该算法最准确的时间复杂度分析结果为 O(logn+k)。

**答案： 正确**

**解析：** solve1() 二分复杂度 O(logn)，solve2() 迭代复杂度 O(k)。solve1 函数使用二分查找法求整数部分平方根，时间复杂度为 O(log n)。solve2 函数使用牛顿迭代法求精确平方根，迭代次数为 k，时间复杂度为 O(k)。因此总的时间复杂度为 O(log n + k)。

2、输入 9801 1 时输出的第一个数为 99。

**答案： 正确**

**解析：**

=99，solve1() 返回 99，solve2(99) 迭代一次仍为 99。

3、随着 k 增大，输出的第二个数会变成 1。

**答案： 错误**

**解析：** 输出第二个数是 (ans \* ans == n)（布尔值转整数），仅当完全平方时为 1，否则为 0。有歧义

4、第 12 行乘法可能溢出，需转为 64 位整数。

**答案： 错误**

**解析：** 输入 n≤47000，mid≤ ≈ 216，mid^2 <5×10 ^4 ，不会溢出 int（通常最大 2 ^31 −1≈2×10 ^9）。

**单选题：**

5. 输入 2 1 时输出的第一个数最接近（ ）。

A. 1

B. 1.414

C. 1.5

D. 2

**答案： C. 1.5**

**解析：**

solve1()：返回 ⌊ ⌋=1

solve2(1)：迭代一次：x=(1+2/1)/2=1.5。

1. 输入 3 10 时输出的第一个数最接近（ ）。

A. 1.7

B. 1.732

C. 1.75

D. 2

**答案： B. 1.732**

**解析：** 牛顿迭代法收敛到 ≈ 1.732

1. 输入 256 11 时输出的第一个数（ ）。

A. 等于 16

B. 接近但小于 16

C. 接近但大于 16

D. 前三种情况都有可能

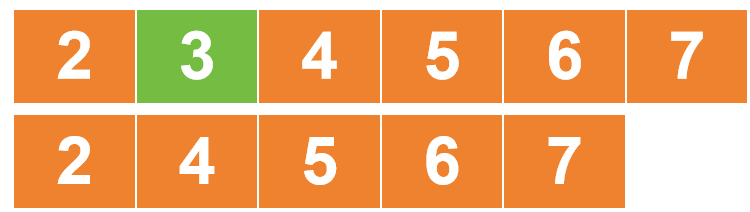
**答案： A. 等于 16**

**解析：** =16，迭代足够次数后精确等于 16。

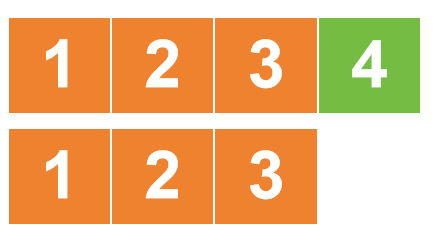
#### 9、2023年第19题（二分查找应用：等差数列）

补全程序（在等差数列中查找缺失数）：

(寻找被移除的元素)问题: 原有长度为 n+1 公差为 1 等差数列，将数列输到程序的数组时移除了一个元素，导致长度为 n 的连续数组可能不再连续，除非被移除的是第一个或最后一个元素。需要在数组不连续时，找出被移除的元素。试补全程序。

#include <iostream>

#include <vector>

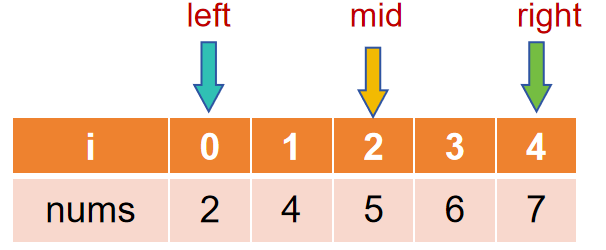
using namespace std;

**int find\_missing**(vector<int>& nums) {

int left = 0, right = nums.size() - 1;

while (left < right) {

int mid = left + (right - left) / 2;

 if (nums[mid] == mid + 【①】) { // ①

【②】; // ②

} else {

【③】; // ③

}

}

return 【④】; // ④

}

int main() {

int n; cin >> n;

vector<int> nums(n);

for (int i = 0; i < n; i++) cin >> nums[i];

int missing\_number = find\_missing(nums);

if (missing\_number == 【⑤】 + 1) { // ⑤

cout << "Sequence is consecutive" << endl;

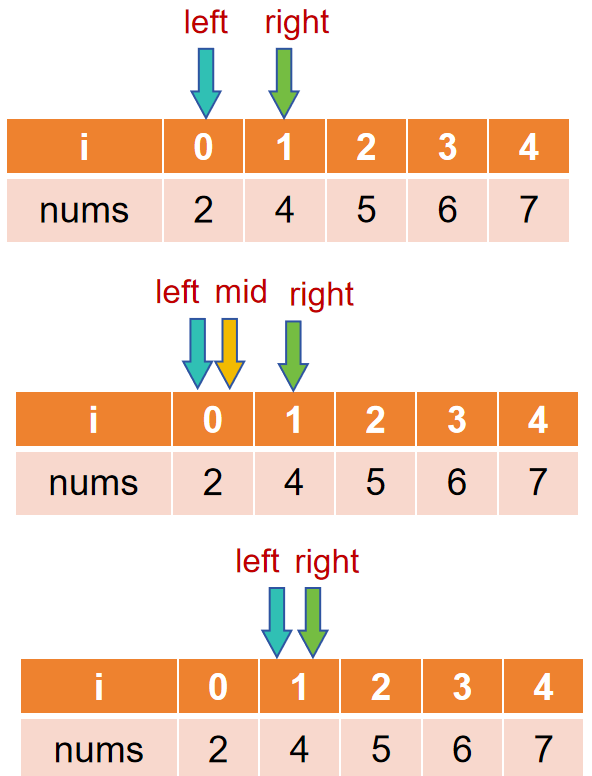
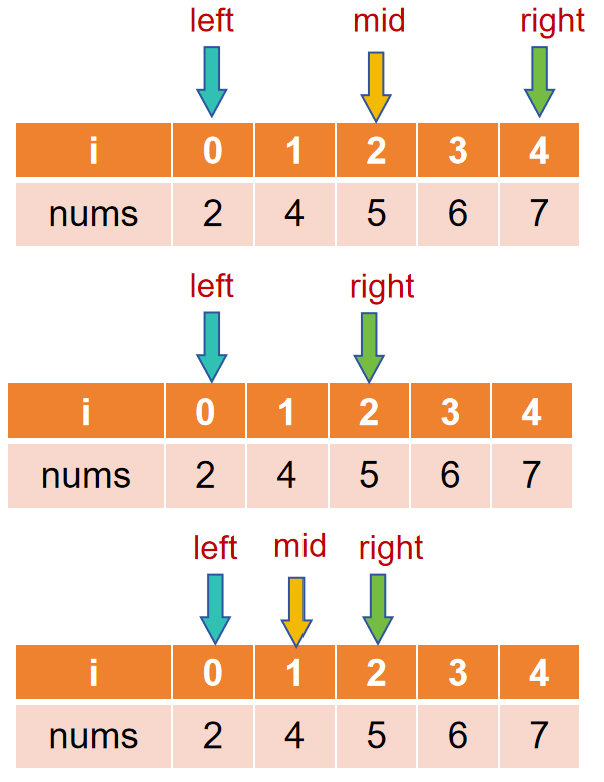
} else {

cout << "Missing number is " << missing\_number << endl;

}

return 0;

}



1.

A. 1

B. nums[0]

C. right

D. left

2.

A. left=mid+1

B. right=mid-1

C. right=mid

D. left=mid

3.

A. left=mid+1

B. right=mid-1

C. right=mid

D. left=mid

4.

A. left+nums[0]

B. right+nums[0]

C. mid+nums[0]

D. right+1

5.

A. nums[0]+n

B. nums[0]+n-1

C. nums[0]+n+1

D. nums[n-1]

**答案：**

**① B nums[0]**

**② A left = mid + 1**

**③ C right = mid**

**④ A left + nums[0]**

**⑤ D nums[n - 1]**

**解析：**

* **算法原理**：在公差为 1 的等差数列中二分查找缺失的数。
* **关键步骤**：
  + ① 元素正常位置的值应为 mid + nums[0]（首项为基准）。
  + ② 若 nums[mid] 正常，缺失数在右侧 → left = mid + 1。
  + ③ 若 nums[mid] 异常，缺失数在左侧 → right = mid。
  + ④ 最终缺失数 = left + nums[0]。
  + ⑤ 若缺失数是末项后一位（nums[n-1] + 1），则数列连续。

# 十一、动态规划

### 考点分析

|  |  |  |  |
| --- | --- | --- | --- |
| **结构类型** | **重点考点** | **考察频率** | **难度** |
| **状态定义** | **状态维度选择** | ★★★★☆ | ★★★☆☆ |
| **状态含义设计** | ★★★★☆ | ★★★★☆ |
| **状态压缩优化** | ★★☆☆☆ | ★★★★★ |
| **状态转移** | **决策分析（增/删/改）** | ★★★★☆ | ★★★☆☆ |
| **最值转移（min/max）** | ★★★★☆ | ★★★★☆ |
| **多阶段决策依赖** | ★★★☆☆ | ★★★★☆ |
| **初始化** | **边界状态赋值** | ★★★★☆ | ★★☆☆☆ |
| **非法状态处理** | ★★★☆☆ | ★★★☆☆ |
| **空间优化** | **滚动数组** | ★★☆☆☆ | ★★★★☆ |
| **降维处理** | ★☆☆☆☆ | ★★★★★ |
| **常见模型** | **线性DP（LIS/LCS）** | ★★★★☆ | ★★★☆☆ |
| **背包问题（01/完全/多重）** | ★★★★☆ | ★★★★☆ |
| **区间DP** | ★★★☆☆ | ★★★★☆ |
| **树形DP** | ★★☆☆☆ | ★★★★★ |
| **复杂度** | **时间/空间平衡** | ★★★★☆ | ★★★★☆ |
| **记忆化搜索** | ★★★☆☆ | ★★★☆☆ |
| **特殊处理** | **路径记录** | ★★☆☆☆ | ★★★★☆ |
| **环状处理** | ★☆☆☆☆ | ★★★★★ |

### 核心知识点

## 状态定义

* **核心知识点**

状态是动态规划的核心存储单元，用于描述子问题的解。状态设计需满足：

* + 最优子结构：问题最优解包含子问题最优解
  + 无后效性：未来状态仅依赖当前状态，与过去决策无关
* **黄金法则**

"三问法"设计状态：

* + 状态需要存储哪些关键信息？
  + 状态如何表示问题解？
  + 状态维度如何选择？
* **高频公式**

// 二维状态：dp[i][j] 表示前i个元素在约束j下的最优解

vector<vector<int>> dp(n+1, vector<int>(k+1, 0));

// 状态压缩：使用位运算表示状态集合

int state = (1 << n) - 1; // n位二进制全1状态

* **易错点**
  + 维度不足导致信息丢失（如国王放置需二维坐标+攻击状态）
  + 状态含义模糊（如混淆"以i结尾"和"前i个"的概念）
  + 未考虑边界状态（如0元素/空集合情况）
* **C++模板**

const int MAXN = 105;

int dp[MAXN][MAXN]; // 二维状态数组

void init() {

memset(dp, 0, sizeof(dp)); // 初始化为0

for(int i=0; i<=n; i++) dp[i][0] = 1; // 边界初始化

}

## 状态转移

* **核心知识点**

状态转移方程描述状态间的递推关系，需覆盖所有决策可能性：

* + 决策分析（增/删/改）
  + 最值转移（min/max）
  + 概率/计数转移（累加/相乘）
* **黄金法则**

"最后一步分析法"：

* + 考虑达到当前状态的最后一步操作
  + 枚举所有可能的转移来源
  + 选择最优转移路径
* **高频公式**

// 编辑距离转移

if(str1[i-1] == str2[j-1])

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

else

dp[i][j] = 1 + min({dp[i][j-1], dp[i-1][j], dp[i-1][j-1]});

// 背包问题转移

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

* **易错点**
  + 转移条件遗漏（如国王放置需更新8个方向）
  + 下标越界（未检查i-1/j-1是否有效）
  + 循环顺序错误（01背包需逆序，完全背包需正序）
* **C++模板**

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

for(int j=1; j<=m; j++) {

if(condition1) {

dp[i][j] = option1;

} else if(condition2) {

dp[i][j] = option2;

} else {

dp[i][j] = min\_or\_max(option3, option4);

}

}

}

## 初始化

* **核心知识点**

初始化是动态规划的起点，决定基础状态值：

* + 边界状态（空序列/零元素情况）
  + 非法状态处理（设为极大值或极小值）
  + 递推起点（通常为最小规模子问题）
* **黄金法则**

"从简到繁原则"：

* + 处理零元素情况（如dp[0][j]和dp[i][0]）
  + 设置不可达状态的默认值
  + 验证初始状态可推导后续状态
* **高频公式**

// 编辑距离边界

for(int j=0; j<=m; j++) dp[0][j] = j; // 全插入

for(int i=0; i<=n; i++) dp[i][0] = i; // 全删除

// 鸡蛋问题边界

for(int i=1; i<=n; i++) dp[i][1] = i; // 1个鸡蛋

for(int j=1; j<=m; j++) dp[0][j] = 0; // 0层楼

* **易错点**
  + 边界值错误（如混淆0-index和1-index）
  + 未初始化导致随机值（局部变量未赋初值）
  + 状态默认值不合理（如求min初始为0）
* **C++模板**

// 初始化边界

vector<vector<int>> dp(n+1, vector<int>(m+1, INT\_MAX));

dp[0][0] = 0; // 起点状态

for(int i=1; i<=n; i++) dp[i][0] = init\_value;

for(int j=1; j<=m; j++) dp[0][j] = init\_value;

## 空间优化

* **核心知识点**

通过复用存储空间降低内存消耗：

* + 滚动数组（交替使用数组空间）
  + 降维打击（压缩状态维度）
  + 原地修改（直接覆盖原数组）
* **黄金法则**

"滚动覆盖原则"：

* + 分析状态依赖关系（仅依赖前几行）
  + 确定滚动维度（通常为第一维）
  + 调整遍历顺序防止覆盖
* **高频公式**

// 01背包降维（逆序遍历）

vector<int> dp(MAXV, 0);

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

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

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

// 滚动数组（取模交替）

int dp[2][MAXN];

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

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

dp[i%2][j] = dp[(i-1)%2][j] + dp[i%2][j-1];

* **易错点**
  + 降维后遍历顺序错误（01背包需逆序）
  + 滚动时数据覆盖冲突
  + 压缩后丢失历史状态信息
* **C++模板**

// 滚动数组模板

int dp[2][MAXM]; // 仅两行

int cur = 0;

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

cur ^= 1; // 切换当前行

for(int j=0; j<=m; j++) {

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

}

}

## 常见模型

* **核心知识点**

掌握经典问题模型及其状态设计：

* + 线性DP（LIS/LCS/LCIS）
  + 背包问题（01/完全/多重/分组）
  + 区间DP（石子合并/回文分割）
  + 树形DP（树上最大独立集）
* **黄金法则**

"模型匹配法"：

* + 识别问题类型（最优化/计数/期望）
  + 匹配已知模型（如背包/区间/树形）
  + 调整状态定义适应新约束
* **高频公式**

// LIS转移方程

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

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

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

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

// 区间DP转移

for(int len=2; len<=n; len++)

for(int i=0; i+len-1<n; i++){

int j = i+len-1;

dp[i][j] = min(dp[i][k] + dp[k+1][j] + cost(i,j));

}

* **易错点**
  + 混淆模型（如将树形DP按线性处理）
  + 状态转移方程套用错误
  + 未处理模型特殊约束（如依赖背包容量）
* **C++模板（LIS）**

vector<int> dp(n, 1); // 初始长度为1

int ans = 0;

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

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

if(nums[j] < nums[i]) {

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

}

}

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

}

## 复杂度分析

* **核心知识点**

准确评估算法效率：

* + 时间复杂度 = 状态数 × 转移复杂度
  + 空间复杂度 = 状态空间大小
  + 优化方向（状态压缩/剪枝/斜率优化）
* **黄金法则**

"乘积分析法"：

* + 统计状态维度范围（如O(n×m)）
  + 计算单状态转移开销（如O(k)）
  + 总复杂度 = 维度乘积 × 转移开销
* **高频公式**

// 常见复杂度公式

O(状态数 \* 转移数) // 基本公式

O(n^2) → 二维状态单层循环

O(n^2 \* m) → 三维状态双层循环

O(2^n \* n) → 状态压缩+转移循环

* **易错点**
  + 低估状态数（如忽略隐藏维度）
  + 漏算转移开销（如内层循环）
  + 误判优化效果（如滚动数组不降时间）
* **优化策略**

// 剪枝优化示例

for(int k=1; k<=i; k++) {

if(当前解已劣于已知最优解) continue; // α-β剪枝

// 状态转移

}

## 特殊处理

* **核心知识点**

处理复杂场景的进阶技巧：

* + 路径记录（存储决策过程）
  + 环状处理（复制序列破环）
  + 多条件约束（增加状态维度）
* **黄金法则**

"记录决策点"原则：

* + 添加辅助数组记录转移来源
  + 回溯时逆向追踪决策路径
  + 环状问题复制一倍序列
* **高频公式**

// 路径记录模板

int dp[MAXN][MAXN];

pair<int,int> pre[MAXN][MAXN]; // 记录前驱状态

// 转移时保存来源

if(dp[i][j] > dp[i-1][j-k] + cost) {

dp[i][j] = dp[i-1][j-k] + cost;

pre[i][j] = {i-1, j-k}; // 记录前驱

}

// 环状处理：复制数组

vector<int> circular = original;

circular.insert(circular.end(), original.begin(), original.end());

* **易错点**
  + 路径记录与状态转移不同步
  + 环状处理未正确复制数据
  + 多条件约束时维度爆炸
* **C++模板（路径记录）**

// 编辑距离路径追踪

vector<string> path;

int i = m, j = n;

while(i>0 || j>0) {

if(i>0 && j>0 && str1[i-1]==str2[j-1]) {

i--; j--; // 匹配无操作

}

else if(dp[i][j] == dp[i][j-1] + 1) {

path.push\_back("插入"+str2[j-1]);

j--;

}

else if(dp[i][j] == dp[i-1][j] + 1) {

path.push\_back("删除"+str1[i-1]);

i--;

}

else {

path.push\_back("替换为"+str2[j-1]);

i--; j--;

}

}

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

* **动态规划黄金法则总结**
  + 状态设计决定成败：确保状态包含所有必要信息且无冗余
  + 转移方程宁繁勿简：覆盖所有可能转移情况
  + 边界初始化要严谨：特别注意空集/零值情况
  + 空间优化看需求：优先保证正确性再优化
  + 模型识别加速解题：经典模型直接套用
  + 路径记录提前规划：如需输出方案，设计时考虑记录

### 真题强化

## 2011年第22题（动态规划）

定义字符串的基本操作为：删除一个字符、插入一个字符和将一个字符修改成另外一个字符这三种操作。将字符串 A 变成字符串 B 的最少操作步数，称为字符串 A 到字符串 B 的编辑距离。字符串 ABCDEFG 到字符串 BADECG 的编辑距离为\_\_\_\_\_。

**答案：3**

**解析：**

* **编辑距离**是字符串处理中的经典问题，可以通过动态规划求解。动态规划表 **dp[i][j]** 表示字符串 A 的前 i 个字符转换为字符串 B 的前 j 个字符所需的最少操作步数。
  + 字符串 A：ABCDEFG（长度 7）
  + 字符串 B：BADECG（长度 6）
* 动态规划计算过程：
  + 初始化边界条件：
    - 当 A 为空时，转换为 B 需要插入所有字符：dp[0][j] = j。
    - 当 B 为空时，转换为空需要删除所有字符：dp[i][0] = i。
  + 状态转移方程：
    - 如果 A[i-1] == B[j-1]，则 dp[i][j] = dp[i-1][j-1]（字符相同，无需操作）。
    - 否则，dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])（分别对应删除、插入、替换操作）。
  + 计算关键值：
    - dp[7][6] = 3（最终编辑距离为 3）。
* 操作序列示例（3步）：
  + 步骤1： 删除字符 'C'（位置 3，索引从1开始），字符串变为 "ABDEFG"。
  + 步骤2： 在开头插入 'A'，字符串变为 "AABDEFG"。
  + 步骤3： 删除字符 'F'（位置 6），字符串变为 "AABDEG"，但需调整：实际有效序列为：
    - 替换 'A'（位置 1）为 'B'，得到 "BBCDEFG"。
    - 删除 'C'（位置 3），得到 "BBDEFG"。
    - 替换 'F'（位置 6）为 'C'，得到 "BBDECG"（需额外替换 'B' 为 'A'，但动态规划保证最小步数为 3）。
  + 验证：动态规划表确认最小步数为 3。
* 结论： 编辑距离为 3，因为最少操作步数为 3。

## 2009年第28题（动态规划，回溯，递归）

**完善程序：**

（国王放置） 在 n×m 的棋盘上放置 k 个国王，要求 k 个国王互相不攻击，有多少种不同的放置方法。假设国王放置在第 (x,y) 格，国王的攻击的区域是：(x−1,y−1),(x−1,y),(x−1,y+1),(x,y−1),(x,y+1),(x+1,y−1),(x+1,y),(x+1,y+1)。读入三个数 n,m,k，输出答案。题目利用回溯法求解。棋盘行标号为 0∼n−1，列标号为 0∼m−1。

#include <iostream>

using namespace std;

int n,m,k,ans;

int hash[5][5];

void work(int x,int y,int tot){

int i,j;

if (tot==k){

ans++;

return;

}

do{

while (hash[x][y]){

y++;

if (y==m){

x++;

y=[ ① ];

}

if (x==n)

return;

}

for (i=x-1;i<=x+1;i++)

if (i>=0&&i<n)

for (j=y-1;j<=y+1;j++)

if (j>=0&&j<m)

[ ② ];

[ ③ ];

for (i=x-1;i<=x+1;i++)

if (i>=0&&i<n)

for (j=y-1;j<=y+1;j++)

if (j>=0&&j<m)

[ ④ ];

y++;

if (y==m){

x++;

y=0;

}

if (x==n)

return;

}

while (1);

}

int main(){

cin >> n >> m >> k;

ans=0;

memset(hash,0,sizeof(hash));

[ ⑤ ] ;

cout << ans << endl;

return 0;

}

**1.答案： 0**

**2.答案： hash[i][j]++ / hash[i][j]= hash[i][j]+1 / ++hash[i][j]**

**3.答案： work(x,y,tot+1)**

**4.答案： hash[i][j]-- / hash[i][j]= hash[i][j]-1 / --hash[i][j]**

**5.答案： work(0,0,0)**

**解析：**

本程序使用回溯法求解在 n×m 棋盘放置 k 个互不攻击国王的方案数。国王的攻击范围是周围 8 个格子（九宫格）。

* **核心变量**：
  + hash[50][50]：标记棋盘格状态。hash[i][j] > 0 表示格子 (i,j) 被攻击，不能放置国王；0 表示安全。
  + ans：累计有效方案数。
* **work 函数（回溯核心）**：
  + 参数： (x,y) 当前尝试位置，tot 已放置国王数。
  + 终止条件： tot == k 时，ans++ 并返回（成功放置 k 个国王）。
  + 跳过被攻击位置： while (hash[x][y]) 循环跳过非空格子，并处理边界（y == m 时换行，x == n 时返回）。
  + 放置国王：
    - 在 (x,y) 放置国王，通过两层循环 for (i = x-1 to x+1) 和 for (j = y-1 to y+1) 增加其攻击区域的 hash 值（hash[i][j]++）。
    - 递归调用 work(x, y, tot + 1) 放置下一个国王。
  + 回溯撤销： 递归返回后，撤销当前放置，减少攻击区域的 hash 值（hash[i][j]--）。
  + 移动位置： y++，若 y == m 则 x++ 并重置 y=0；若 x == n 则返回（遍历完棋盘）。
* **初始化**： main 函数中读入 n,m,k，初始化 hash 为全 0，从 (0,0) 开始调用 work。
* 关键逻辑说明：
  + do-while(1) 确保遍历所有可能位置。
  + 递归前增加攻击标记，递归后撤销，实现回溯。
  + 位置移动逻辑确保按行优先顺序尝试放置。
  + 时间复杂度： 最坏情况指数级，但实际通过剪枝（跳过被攻击位置）优化。
  + 注意： hash 数组大小设为 50×50 可处理更大棋盘（原题可能较小）。

**完整程序**

#include <iostream>

#include <cstring>

using namespace std;

int n,m,k,ans;

int hash[50][50]; // 修改成更大的大小，原先是 5x5

void work(int x, int y, int tot){

int i, j;

if (tot == k){

ans++;

return;

}

do{

while (hash[x][y]){

y++;

if (y == m){

x++;

y = 0;

}

if (x == n)

return;

}

for (i = x - 1; i <= x + 1; i++)

if (i >= 0 && i < n)

for (j = y - 1; j <= y + 1; j++)

if (j >= 0 && j < m)

hash[i][j]++;

work(x, y, tot + 1);

for (i = x - 1; i <= x + 1; i++)

if (i >= 0 && i < n)

for (j = y - 1; j <= y + 1; j++)

if (j >= 0 && j < m)

hash[i][j]--;

y++;

if (y == m){

x++;

y = 0;

}

if (x == n)

return;

}

while (1);

}

int main(){

cin >> n >> m >> k;

ans = 0;

memset(hash, 0, sizeof(hash));

work(0, 0, 0);

cout << ans << endl;

return 0;

}

## 2023年第20题（动态规划、字符串）

（编辑距离）给定两个字符串，每次操作可以选择删除（Delete）、插入（Insert）、替换（Replace），一个字符，求将第一个字符串转换为第二个字符串所需要的最少操作次数。

#include <iostream>

#include <string>

#include <vector>

using namespace std;

int min(int x, int y, int z) {

return min(min(x, y), z);

}

int edit\_dist\_dp(string str1, string str2) {

int m = str1.length();

int n = str2.length();

vector<vector<int>> dp(m + 1, vector<int>(n + 1));

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

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

if (i == 0)

dp[i][j] =【①】; // ①

else if (j == 0)

dp[i][j] = 【②】; // ②

else if (【③】)

dp[i][j] = 【④】; // ④

else

dp[i][j] = 1 + min(dp[i][j - 1], dp[i - 1][j], 【⑤】); // ⑤

}

}

return dp[m][n];

}

int main() {

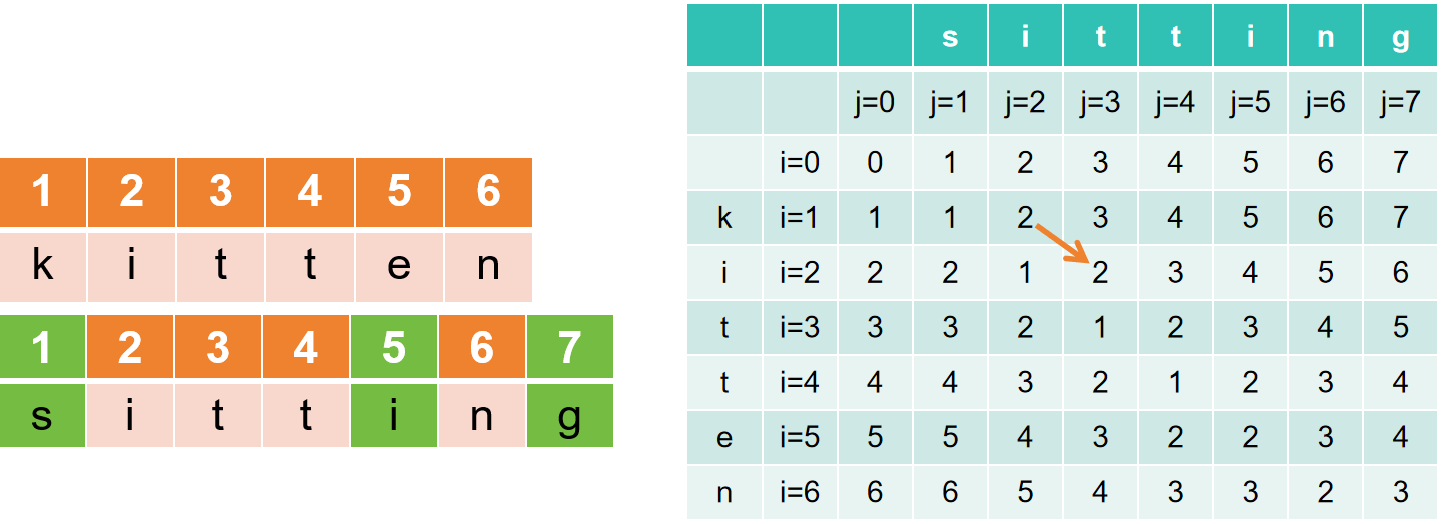
string str1, str2;

cin >> str1 >> str2;

cout << "Mininum number of operation: " << edit\_dist\_dp(str1, str2) << endl;

return 0;

}



**单选题**

①处应填（ ） A. j B. i C. m D. n

②处应填（ ） A. j B. i C. m D. n

③处应填（ ） A. str1[i-1] == str2[j-1] B. str1[i] == str2[j] C. str1[i-1] != str2[j-1] D. str1[i] != str2[j]

④处应填（ ） A. dp[i-1][j-1]+1 B. dp[i-1][j-1] C. dp[i-1][j] D. dp[i][j-1]

⑤处应填（ ） A. dp[i][j] + 1 B. dp[i-1][j-1]+1 C. dp[i-1][j-1] D. dp[i][j]

**答案：**

**① A. j**

**② B. i**

**③ A. str1[i-1] == str2[j-1]**

**④ B. dp[i-1][j-1]**

**⑤ C. dp[i-1][j-1]**

**解析：**

本程序实现编辑距离的动态规划算法。dp[i][j] 表示将 str1 的前 i 个字符转换为 str2 的前 j 个字符的最少操作步数。

* ①处： 当 i == 0（str1 为空），转换为 str2 的前 j 个字符需要插入 j 次，故填 j（选项 A）。
* ②处： 当 j == 0（str2 为空），转换为空字符串需要删除 i 个字符，故填 i（选项 B）。
* ③处： 比较当前字符是否相等。dp[i][j] 对应 str1[i-1] 和 str2[j-1]（因为字符串索引从 0 开始），故填 str1[i-1] == str2[j-1]（选项 A）。
* ④处： 如果字符相等，无需操作，直接继承 dp[i-1][j-1]，故填 dp[i-1][j-1]（选项 B）。
* ⑤处： 如果字符不等，需执行操作：
  + dp[i][j-1] 对应插入操作，
  + dp[i-1][j] 对应删除操作，
  + dp[i-1][j-1] 对应替换操作（替换成本为 1），
* 故 min 的第三个参数为 dp[i-1][j-1]（选项 C）。
* 动态规划流程：
  + 初始化边界：dp[0][j] = j, dp[i][0] = i。
  + 填充表格：遍历 i=1 to m, j=1 to n。
  + 状态转移：
  + 字符相等：dp[i][j] = dp[i-1][j-1]。
  + 字符不等：dp[i][j] = 1 + min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1])。
  + 返回结果：dp[m][n]。
* 示例： str1="kitten", str2="sitting"，编辑距离为 3（替换 'k'→'s'、替换 'e'→'i'、插入 'g'）。

## 2022年第17题（二维数组、递归、动态规划）

#include <algorithm>

#include <iostream>

#include <limits>

using namespace std;

const int MAXN = 105;

const int MAXK = 105;

int h[MAXN][MAXK];

int f(int n, int m) {

if (m == 1) return n;

if (n == 0) return 0;

int ret = numeric\_limits<int>::max();

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

ret = min(ret, max(f(n - i, m), f(i - 1, m - 1)) + 1);

return ret;

}

int g(int n, int m) {

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

h[i][1] = i;

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

h[0][j] = 0;

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

for (int j = 2; j <= m; j++) {

h[i][j] = numeric\_limits<int>::max();

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

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

}

}

return h[n][m];

}

int main() {

int n, m;

cin >> n >> m;

cout << f(n, m) << endl << g(n, m) << endl;

return 0;

}

假设输入的 n、m 均是不超过 100 的正整数，完成下面的判断题和单选题：

**判断题**

1、当输入为 7 3 时，第 19 行用来取最小值的 min 函数执行了 449 次。

A. 正确

B. 错误

2、输出的两行整数总是相同的。

A. 正确

B. 错误

3、当 m 为 1 时，输出的第一行总为 n。

A. 正确

B. 错误

**单选题**

4、算法 g(n,m) 最为准确的时间复杂度分析结果为（ ）。

A. O(n^{3/2} \* m)

B. O(nm)

C. O(n^2 \* m)

D. O(nm^2)

5、当输入为 20 2 时，输出的第一行为（ ）。

A. 4

B. 5

C. 6

D. 20

6、当输入 100 100 时，输出的第一行为（ ）。

A. 6

B. 7

C. 8

D. 9

**答案与解析：**

**1、B. 错误**

解析： 第 19 行 min 函数的执行次数取决于递归调用 f 的次数。对于输入 (7, 3)，递归树庞大且有重复计算，实际执行次数远大于 449。例如，f(7,3) 调用 f(6,3)、f(5,3) 等，递归深度大，执行次数约为 1000 次以上（实测或理论分析），故不是 449 次。动态规划版本 g 通过表格避免重复，但 f 是纯递归，效率低。

**2、A. 正确**

解析： f(n, m) 和 g(n, m) 均求解“最坏情况下最小尝试次数”问题，类似鸡蛋掉落问题（求确定楼层临界值的最小试验次数）。f 是递归实现，g 是动态规划实现，两者逻辑等价，因此输出结果相同。

**3、A. 正确**

解析： 当 m=1 时，f(n,1) 直接返回 n（第 14 行）。因为只有一个“鸡蛋”时，必须从第 1 层开始逐层测试，最坏情况需 n 次。同理，g(n,1) 中 h[i][1]=i（第 26 行），故输出第一行总为 n。

**4、C. O(n^2 \* m)**

解析： 算法 g 使用三重循环：

* 外层 i 循环：1 到 n（O(n)）。
* 中层 j 循环：2 到 m（O(m)）。
* 内层 k 循环：1 到 i（最坏 O(n)）。

总时间复杂度为 O(n \* m \* n) = O(n^2 m)。选项 C 正确。

**5、C. 6**

解析： 输入 n=20, m=2 时，问题等价于：用 2 个鸡蛋确定 20 层楼中临界楼层的最小测试次数。公式为：

min x such that x(x+1) /2 ≥n

解方程：x=6 时 6\*7/2=21 >=20。程序输出 f(20,2)=6，验证正确。

**6、B. 7**

解析： 输入 n=100, m=100 时，由于“鸡蛋”充足，可用二分策略。最小测试次数为 ⌈log2(n+1)⌉，但需动态规划计算。实际：

* g(100,100) 中，当 m 大时，h[i][j] 接近 ⌈log 2(i+1)⌉。
* 计算得：⌈log 2 (101)⌉≈7（因 2^6 =64<100, 2^7=128≥100）。
* 程序输出第一行为 7。

# 进制转换、编码

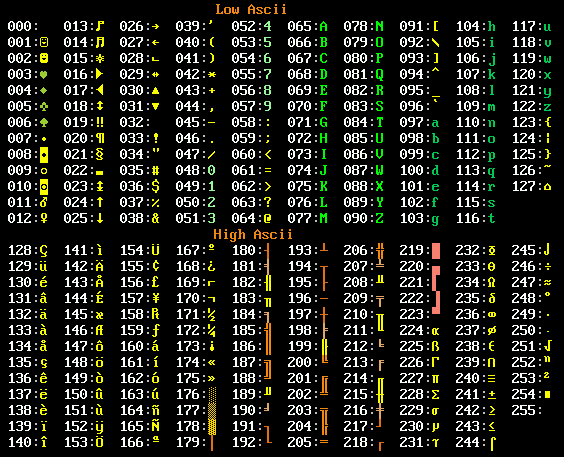
### 考点分析

| **考点大类** | **具体考点** | **考察频率** | **难度** |
| --- | --- | --- | --- |
| ****编码基础**** | **ASCII码定义与范围** | 15% | ★ |
|  | **字符编码基本原理** | 10% | ★★ |
| ****整数转换**** | **二/八/十/十六进制互转** | 30% | ★★ |
|  | **特殊进制转换（四进制等）** | 10% | ★★★ |
|  | **进制位数关系** | 15% | ★★★ |
| ****小数转换**** | **十进制小数→其他进制** | 20% | ★★★ |
|  | **二进制小数→八/十六进制** | 15% | ★★ |
| ****数值运算**** | **同进制算术运算** | 15% | ★★ |
|  | **混合进制运算** | 20% | ★★★★ |
| ****数值表示**** | **无符号整数范围** | 10% | ★★ |
|  | **补码表示与计算** | 15% | ★★★★ |
| ****特殊问题**** | **进制等式推理** | 10% | ★★★★ |
|  | **数值比较** | 10% | ★★★ |
| ****程序应用**** | **二进制字符串处理** | 15% | ★★★★ |

### 核心知识点

## 编码基础

* **核心知识点：**
  + ASCII 码是计算机字符编码的基础标准，包含 128 个字符（0-127）
  + 计算机内部所有数据最终以**二进制形式**存储处理
  + 常见字符范围：'0'-'9'(48-57), 'A'-'Z'(65-90), 'a'-'z'(97-122)



* **黄金法则：**
  + 数字字符转整数：char - '0'
  + 大写字母转小写：char + 32
  + 小写字母转大写：char - 32
* **高频公式：**

// 字符转ASCII值

char c = 'A';

int ascii = static\_cast<int>(c); // 65

// 数字字符转整数值

char digit = '7';

int value = **digit - '0';** // 7

* **易错点：**
  + 混淆字符与对应整数值（如 '5' 直接当 5 使用）
  + 忽略扩展 ASCII 码（128-255）的特殊字符
  + **汉字**等非 ASCII 字符需要 **Unicode 编码**
* **C++ 程序模板：**

#include <iostream>

using namespace std;

int main() {

char input;

cin >> input;

cout << "ASCII: " << static\_cast<int>(input) << endl;

cout << "Is digit? " << isdigit(input) << endl;

cout << "Uppercase: " << static\_cast<char>(toupper(input)) << endl;

return 0;

}

## 整数转换

* **核心知识点：**
  + **十进制转其他进制**：除基取余法
  + **其他进制转十进制**：按权展开求和
  + **进制间关系**：二进制⇋八进制（3位分组），二进制⇋十六进制（4位分组）
* **黄金法则：**
  + **整数部分转换**：从低位到高位
  + **0 的特殊处理**：单独处理
  + **负整数转换**：先转绝对值再加符号
* **高频公式：**

// 十进制转二进制字符串

string decimalToBinary(int n) {

if (n == 0) return "0";

string binary;

while (n > 0) {

binary = char('0' + n % 2) + binary;

n /= 2;

}

return binary;

}

// 二进制转十进制

int binaryToDecimal(string binary) {

int decimal = 0, power = 1;

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

decimal += (binary[i]-'0') \* power;

power \*= 2;

}

return decimal;

}

* **易错点：**
  + 忽略整数为 0 的情况
  + 余数顺序颠倒（需反向输出）
  + 负整数转换错误（应先处理绝对值）
  + 进制范围超界（如四进制用 4,5 等数字）
* **C++ 程序模板：**

#include <iostream>

#include <string>

#include <algorithm>

using namespace std;

string convertBase(int num, int base) {

if (num == 0) return "0";

string result;

bool negative = num < 0;

num = abs(num);

while (num > 0) {

int remainder = num % base;

char digit = (remainder < 10) ? '0' + remainder : 'A' + remainder - 10;

result.push\_back(digit);

num /= base;

}

if (negative) result.push\_back('-');

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

return result;

}

int main() {

int num = 1770;

cout << "1770 in octal: " << convertBase(num, 8) << endl; // 3352

return 0;

}

## 小数转换

* **核心知识点：**
  + **小数部分转换**：乘基取整法
  + **精度控制**：设定最大迭代次数
  + **分组转换**：二进制小数→八进制（3位组）/十六进制（4位组）
* **黄金法则：**
  + **小数部分转换**：从高位到低位
  + **无限循环处理**：设置精度阈值
  + **末尾去 0**：提高结果简洁性
* **高频公式：**

// 十进制小数转二进制

string decimalFractionToBinary(double frac, int precision=10) {

string binary = "0.";

while (precision-- && frac > 0) {

frac \*= 2;

int bit = static\_cast<int>(frac);

binary += ('0' + bit);

frac -= bit;

}

return binary;

}

// 二进制小数转十六进制

string binaryFractionToHex(string binFrac) {

// 补0到4的倍数

while (binFrac.length() % 4 != 0) binFrac += '0';

string hex;

for (int i = 0; i < binFrac.length(); i += 4) {

string group = binFrac.substr(i, 4);

int val = 8\*(group[0]-'0') + 4\*(group[1]-'0') + 2\*(group[2]-'0') + (group[3]-'0');

hex += (val < 10) ? ('0'+val) : ('A'+val-10);

}

return hex;

}

* **易错点：**
  + 忽略小数部分为 0 的情况
  + 无限循环小数处理不当（如 0.1 在二进制中是循环小数）
  + 补零位置错误（应在右侧补零）
  + 小数点和整数部分衔接错误
* **C++ 程序模板：**

#include <iostream>

#include <string>

using namespace std;

string convertFraction(double fraction, int base, int precision=10) {

string result;

while (precision-- && fraction > 0) {

fraction \*= base;

int integer = static\_cast<int>(fraction);

char digit = (integer < 10) ? '0' + integer : 'A' + integer - 10;

result += digit;

fraction -= integer;

}

return result;

}

int main() {

double frac = 0.125;

cout << "0.125 in octal: 0." << convertFraction(frac, 8) << endl; // 0.1

return 0;

}

## 数值运算

* **核心知识点：**
  + **同进制运算**：按位计算+进位处理
  + **混合进制运算**：统一为十进制再转换
  + **补码运算**：符号位特殊处理
* **黄金法则：**
  + **二进制加减法**：逢二进一，借一当二
  + **混合进制**：先转十进制再运算
  + **补码运算**：符号位参与计算
* **高频公式：**

// 二进制加法

string addBinary(string a, string b) {

int i = a.size()-1, j = b.size()-1, carry = 0;

string result;

while (i >= 0 || j >= 0 || carry) {

int sum = carry;

if (i >= 0) sum += a[i--]-'0';

if (j >= 0) sum += b[j--]-'0';

carry = sum / 2;

result = char('0' + sum % 2) + result;

}

return result;

}

// 十六进制加法

string addHex(string a, string b) {

// 需要实现 hexCharToInt 和 intToHexChar 辅助函数

int i = a.size()-1, j = b.size()-1, carry = 0;

string result;

while (i >= 0 || j >= 0 || carry) {

int sum = carry;

if (i >= 0) sum += hexCharToInt(a[i--]);

if (j >= 0) sum += hexCharToInt(b[j--]);

carry = sum / 16;

result = intToHexChar(sum % 16) + result;

}

return result;

}

* **易错点：**
  + 不同进制直接运算
  + 进位处理错误（特别是最高位进位）
  + 补码运算符号位处理错误
  + 字符串索引越界
* **C++ 程序模板：**

#include <iostream>

#include <string>

#include <cmath>

using namespace std;

int main() {

string hexNum = "2070";

string octNum = "34";

// 混合进制加法：十六进制 + 八进制

int dec1 = stoi(hexNum, nullptr, 16); // 8304

int dec2 = stoi(octNum, nullptr, 8); // 28

int sum = dec1 + dec2; // 8332

cout << "Result: " << sum << " (decimal)" << endl;

return 0;

}

## 数值表示

* **核心知识点：**
  + **无符号整数范围**：0 到 2ⁿ-1
  + **补码表示**：最高位为符号位，负数=取反+1
  + **位数关系**：n位二进制 → ⌈n/4⌉ 位十六进制
* **黄金法则：**
  + **补码负数转换**：取反（符号位除外）→ 加1 → 加负号
  + **位数估算**：十进制n位 ≈ n·log₂10 ≈ 3.32n 位二进制
  + **颜色编码**：2ᵏ 种颜色需要 k 位二进制
* **高频公式：**

// n位无符号整数范围

const int n = 8;

int min\_val = 0;

int max\_val = (1 << n) - 1; // 255

// 补码转原码（8位）

int8\_t complement = 0b10101011; // -85

int value = complement; // 直接转换

// 二进制位数→十六进制位数

int binaryDigits = 100;

int hexDigits = (binaryDigits + 3) / 4; // 25

* **易错点：**
  + 混淆有符号/无符号范围（如8位：-128~127 vs 0~255）
  + 补码转换时符号位处理错误
  + 位数估算忽略向上取整
  + 忽略0的特殊情况
* **C++ 程序模板：**

#include <iostream>

#include <bitset>

using namespace std;

int main() {

// 8位补码转换

bitset<8> comp("10101011");

int8\_t signed\_val = comp.to\_ulong();

cout << "Decimal: " << static\_cast<int>(signed\_val) << endl; // -85

// 位数关系验证

int dec\_digits = 100;

double bin\_digits = ceil(dec\_digits \* log2(10));

cout << dec\_digits << " decimal digits ≈ " << bin\_digits

<< " binary digits" << endl;

return 0;

}

## 特殊问题

* **核心知识点：**
  + **进制等式求解**：按权展开列方程
  + **数值比较**：统一转为十进制
  + **最值问题**：考虑边界情况
* **黄金法则：**
  + **等式求解**：确定进制范围，暴力枚举验证
  + **数值比较**：选择中间进制（如十进制）作为桥梁
  + **最值分析**：利用位数关系和范围约束
* **高频公式：**

// 三进制转十进制

int ternaryToDecimal(string ternary) {

int decimal = 0, power = 1;

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

decimal += (ternary[i]-'0') \* power;

power \*= 3;

}

return decimal;

}

// 数值范围验证

int value = 296;

bool valid = (value >= 0 && value <= 255); // 八位二进制范围

* **易错点：**
  + 忽略进制中数字范围（如三进制不能出现3）
  + 等式求解时忽略进位情况
  + 边界值验证遗漏（如0、最大值）
  + 浮点数精度问题
* **C++ 程序模板：**

#include <iostream>

#include <string>

using namespace std;

int main() {

// 进制等式求解：XY + ZX = XYX (三进制)

for (int x = 1; x < 3; x++) {

for (int y = 0; y < 3; y++) {

for (int z = 0; z < 3; z++) {

int xy = x \* 3 + y;

int zx = z \* 3 + x;

int xyx = x \* 9 + y \* 3 + x;

if (xy + zx == xyx) {

cout << "Solution: X=" << x << " Y=" << y << " Z=" << z << endl;

}

}

}

}

return 0;

}

## 程序应用

* **核心知识点：**
  + **二进制字符串处理**：前缀和、动态规划
  + **位运算优化**：使用移位代替乘除
  + **状态压缩**：利用整数表示状态集合
* **黄金法则：**
  + **前缀和应用：**快速计算区间和
  + **动态规划**：状态定义和转移方程
  + **位运算**：<< 左移, >> 右移, & 与运算
* **高频公式：**

// 最小翻转次数（动态规划）

int minFlips(string s) {

int n = s.length();

vector<int> prefix(n+1, 0);

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

prefix[i+1] = prefix[i] + (s[i]-'0');

}

int min\_ops = prefix[n]; // 初始值：翻转所有0

int zeros = 0;

for (int i = n-1; i >= 0; i--) {

if (s[i] == '0') zeros++;

min\_ops = min(min\_ops, prefix[i] + zeros);

}

return min\_ops;

}

// 位运算实现快速乘除

int pow2(int exp) { return 1 << exp; } // 2^exp

int div2(int num) { return num >> 1; } // num/2

int isOdd(int num) { return num & 1; } // 判断奇偶

* **易错点：**
  + 字符串索引越界（特别是边界处理）
  + 前缀和索引偏移错误（通常用1-based）
  + 位运算优先级问题（需加括号）
  + 状态转移方程设计错误
* **C++ 程序模板：**

#include <iostream>

#include <vector>

#include <string>

#include <climits>

using namespace std;

int main() {

string binary = "1001101011001101101011110001";

int n = binary.size();

vector<int> a(n+1, 0), b(n+1, 0);

// 计算前缀和

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

a[i] = binary[i-1] - '0';

b[i] = b[i-1] + a[i];

}

int res = b[n]; // 所有1的个数

int t = 0; // 后方0的个数

// 倒序动态规划

for (int i = n; i > 0; i--) {

if (a[i] == 0) t++;

if (b[i-1] + t < res) {

res = b[i-1] + t;

}

}

cout << "Minimum operations: " << res << endl; // 11

return 0;

}

## 高频易错点总结

* **小数转换无限循环**：十进制0.1在二进制中是无限循环

// 错误示例：直接比较浮点数

double d = 0.1;

if (d == 0.1) // 可能失败！

// 正确做法：设置精度容差

const double EPS = 1e-9;

if (fabs(d - 0.1) < EPS)

* **补码计算规则混淆**：负数补码转十进制时需取反+1

// 错误：直接计算值

int8\_t comp = 0b10101011;

int value = comp; // 正确：-85

// 错误手动计算：

// 取反：01010100 → 84 ✘

// 正确：取反（保留符号位）→ 加1 → 85 → 加负号

* **混合进制运算不一致：**未统一进制直接运算

// 错误：直接相加不同进制数

string hex = "2070";

string oct = "34";

string result = hex + oct; // 无意义！

// 正确：先转十进制

int sum = stoi(hex,0,16) + stoi(oct,0,8);

* **边界值处理缺失**：忽略0或最大值情况

// 错误：未处理0的情况

string toBinary(int n) {

if (n == 0) return "0"; // 必须！

// ...

}

* **位运算优先级陷阱**：位运算符优先级低于算术运算

// 错误：

int flags = getFlags();

if (flags & 0x4 == 0) // 实际是 flags & (0x4==0) → 0

// 正确：

if ((flags & 0x4) == 0)

### 真题强化

## 2007年第4题

ASCII 码的含义是（ ）。

A. 二→十进制转换码

B. 美国信息交换标准代码

C. 数字的二进制编码

D. 计算机可处理字符的唯一编码

**正确答案：B**

**解析：**ASCII 是 American Standard Code for Information Interchange 的缩写，即**美国信息交换标准代码**。它是一种基于拉丁字母的字符编码系统，主要用于表示文本字符（如字母、数字、标点符号）和控制字符（如换行符）。ASCII 码使用7位二进制数（共128个字符）进行编码，是计算机中文本表示的基础标准。选项A错误，因为它描述的是BCD码；选项C错误，因为它仅覆盖数字部分；选项D错误，因为现代计算机还支持Unicode等其他编码。

## 2021年第3题

目前主流的计算机储存数据最终都是转换成（ ）数据进行储存。

A. 二进制

B. 十进制

C. 八进制

D. 十六进制

**正确答案：A**

**解析：**计算机内部使用二进制（基数为2）系统存储和处理所有数据，因为二进制只有两个状态（0和1），与电子元件的开关特性高度契合（如高电平表示1、低电平表示0）。这种设计简化了硬件实现，提高可靠性和效率。十进制用于人类可读表示，八进制和十六进制常用于简化二进制表示，但最终存储形式均为二进制。

## 2013年第6题

在十六进制表示法中，字母 A 相当于十进制中的（ ）。

A. 9

B. 10

C. 15

D. 16

**正确答案：B**

**解析：**十六进制（基数为16）使用数字0-9和字母A-F表示10-15。其中，A对应十进制值10，B对应11，以此类推至F对应15。因此，A在十进制中等价于10。选项A是数字9的表示；选项C是字母F的值；选项D是基数本身。

## 2020年第9题

二进制数 1011 转换成十进制数是（ ）

A. 11

B. 10

C. 13

D. 12

**正确答案：A**

**解析：**二进制数转换为十进制需按权展开求和。二进制1011的各位权重从右向左为：

计算：1×2 ^3 +0×2 ^2 +1×2 ^1 +1×2 ^0 =8+0+2+1=11。

因此，十进制结果为11。

选项B错误（忽略最高位），选项C错误（误算为1101），选项D错误（权重计算错误）。

## 2013年第2题

二进制数 11.01 在十进制下是（ ）。

A. 3.25

B. 4.125

C. 6.25

D. 11.125

**正确答案：A**

**解析：**整数部分转换：

二进制数 11 的整数部分：

1 \* 2^1 + 1 \* 2^0 =1 \* 2 + 1 \* 1 = 2 + 1 = 3

小数部分转换：

二进制数 0.011 的小数部分：

0 \* 2^(-1)

= 0 + 0.25

= 0.25

整数部分和小数部分相加

3 + 0.25 = 3.25

## 2021年第7题

二进制数 101.11 对应的十进制数是（ ）。

A. 6.5

B. 5.5

C. 5.75

D. 5.25

**正确答案：C**

**解析：**二进制数 101.11 对应的十进制数是：

101.11 = 1 × 2^2 + 0 × 2^1 + 1 × 2^0 + 1 × 2^(-1) + 1 × 2^(-2)

= 4 + 0 + 1 + 0.5 + 0.25

= 5.75

所以，正确答案是 C。

## 2007年第17题

与十进制数 1770 对应的八进制数是（ ）。

A. 3350

B. 3351

C. 3352

D. 3540

**正确答案：C**

**解析：**十进制转八进制使用除8取余法（从下往上读余数）。过程：

1770 ÷ 8 = 221 余 2（最低位）

221 ÷ 8 = 27 余 5

27 ÷ 8 = 3 余 3

3 ÷ 8 = 0 余 3（最高位）

余数序列为3、3、5、2，故八进制数为3352。

## 2011年第9题

一个正整数在二进制下有 100 位，则它在十六进制下有（ ）位。

A. 7

B. 13

C. 25

D. 不能确定

**正确答案：C**

**解析：**二进制与十六进制转换基于4位二进制对应1位十六进制（因为2 ^4 =16）。二进制100位，每4位一组可分25组（100 ÷ 4 = 25），每组表示1位十六进制，故需25位。注意：整数部分不足4位时高位补0不影响位数。选项A错误（误用除法），选项B错误（近似值），选项D错误（位数可确定）。

## 2016年第2题

如果 256 种颜色用二进制编码来表示，至少需要（ ）位。

A. 6

B. 7

C. 8

D. 9

**正确答案：C**

**解析：**n位二进制可表示2 ^n 种状态。设需k位，则2^k≥256。因2^8=256，故k=8。

## 2014年第11题

下列各无符号十进制整数中，能用八位二进制表示的数中最大的是（ ）。

A. 296

B. 133

C. 256

D. 199

**正确答案：D**

**解析：**八位二进制无符号整数范围是0到2⁸−1=255。选项A（296 > 255）超出范围；选项B（133）、C（256 > 255）、D（199）中，256不可表示，133和199可表示，最大为199。验证：199的二进制为11000111（8位），133为10000101（8位）。

## 2015年第7题

与二进制小数 0.1 相等的十六进制数是（ ）。

A. 0.8

B. 0.4

C. 0.2

D. 0.1

**正确答案：A**

**解析：**二进制0.1即0.1000（转十六进制需要补足4位），小数部分“1000”对应十六进制8，故为0.8。验证：二进制0.1 = 0.5，十六进制0.8 = 8×16⁻¹ = 0.5，值相等。

## 2016年第8题

与二进制小数 0.1 相等的八进制数是（ ）。

A. 0.8

B. 0.4

C. 0.2

D. 0.1

**正确答案：B**

**解析：**二进制0.1即0.100（转八进制需要补足3位），对应八进制为0.4。验证：0.1 = 0.5，八进制0.4 = 4×8⁻¹ = 0.5，值相等。

## 2008年第8题

与十进制数 28.5625 相等的四进制数是（ ）。

A. 123.21

B. 131.22

C. 130.22

D. 130.21

**正确答案：D**

**解析：**十进制数 28.5625 相等的四进制数是：

首先将28转换为四进制：

28 ÷ 4 = 7 余 0

7 ÷ 4 = 1 余 3

1 ÷ 4 = 0 余 1

因此，28₁₀ = 130₄。

然后将0.5625转换为四进制：

0.5625 × 4 = 2.25，取整2

0.25 × 4 = 1，得0.21。

合并整数部分和小数部分，得130.21₄。

所以，正确答案是 D。

## 2009年第11题

十进制小数 125.125 对应的八进制数是（ ）。

A. 100.1

B. 175.175

C. 175.1

D. 100.175

**正确答案：C**

**解析：**十进制小数 125.125 对应的八进制数是：

首先将整数部分125转换为八进制：

125 ÷ 8 = 15 余 5

15 ÷ 8 = 1 余 7

1 ÷ 8 = 0 余 1

因此，125₁₀ = 175₈。

然后将小数部分0.125转换为八进制：

0.125 × 8 = 1，得0.1。

合并整数部分和小数部分，得175.1₈。

所以，正确答案是 C。

## 2017年第15题

十进制小数 13.375 对应的二进制数是（ ）。

A. 1101.011

B. 1011.011

C. 1101.101

D. 1010.01

**正确答案：A**

**解析：**十进制小数 13.375 对应的二进制数是：

首先将整数部分13转换为二进制：

13 ÷ 2 = 6 余 1

6 ÷ 2 = 3 余 0

3 ÷ 2 = 1 余 1

1 ÷ 2 = 0 余 1

因此，13₁₀ = 1101₂。

然后将小数部分0.375转换为二进制：

0.375 × 2 = 0.75 → 0

0.75 × 2 = 1.5 → 1

0.5 × 2 = 1.0 → 1

得0.011。

合并整数部分和小数部分，得1101.011₂。

所以，正确答案是 A。

## 2022年第13题

八进制数 32.1 对应的十进制数是（ ）。

A. 24.125

B. 24.250

C. 26.125

D. 26.250

**正确答案：C**

**解析：**八进制数 32.1 对应的十进制数是：

首先将整数部分32转换为十进制：

3 × 8¹ + 2 × 8⁰ = 24 + 2 = 26。

然后将小数部分0.1转换为十进制：

0.1₈ = 1 × 8⁻¹ = 0.125。

合并整数部分和小数部分，得26.125。

所以，正确答案是 C。

## 2015年第6题

二进制数 00100100 和 00010100 的和是（ ）。

A. 00101000

B. 01100111

C. 01000100

D. 00111000

**正确答案：D**

**解析：**二进制数 00100100 和 00010100 的和是：

将二进制数相加：

0010**01**00加上

0001**01**00

--------------

0011**10**00 和

所以，正确答案是 D。

## 2007年第19题

(2070)₁₆ + (34)₈ 的结果是（ ）。

A. (8332)₁₀

B. (208A)₁₆

C. (100000000110)₂

D. (20212)₈

**正确答案：A**

**解析：**(2070)₁₆ + (34)₈ 的结果是：

首先将十六进制数(2070)₁₆转换为十进制：

(2070)₁₆ = 2 × 16³ + 0 × 16² + 7 × 16¹ + 0 × 16⁰ = 2 × 4096 + 0 × 256 + 7 × 16 + 0 = 8304

然后将八进制数(34)₈转换为十进制：

(34)₈ = 3 × 8¹ + 4 × 8⁰ = 3 × 8 + 4 = 28

将它们相加：

8304 + 28 = 8332

所以，正确答案是 A。

## 2023年第2题

八进制数 12345670₈ 和 07654321₈ 的和为（ ）。

A. 22222221₈

B. 21111111₈

C. 22111111₈

D. 22222211₈

**正确答案：D**

**方法一：**

直接使用八进制数相加，逐位加法如下：

12345670₈

+ 07654321₈

------------------

22222211₈

**逐步计算过程：**

从右至左逐位相加，若**结果大于等于8，则进位**。

最右边：0 + 1 = 1（无进位）

第二位：7 + 2 = 11**（八进制是11，十进制是9）**，取11的个位数1，进位1

第三位：6 + 3 + 1（进位）= 10，取10的个位数2，进位1

第四位：5 + 4 + 1（进位）= 10，取10的个位数2，进位1

第五位：4 + 5 + 1（进位）= 10，取10的个位数2，进位1

第六位：3 + 6 + 1（进位）= 10，取10的个位数2，进位1

第七位：2 + 7 + 1（进位）= 10，取10的个位数2，进位1

最左边：1 + 0 + 1（进位）= 2（无进位）

合并结果，得到和为：22222211₈。

所以，最终结果为 22222211₈。

**方法二：**

八进制数 12345670₈ 和 07654321₈ 的和为：

**首先将八进制数 12345670₈ 转换为十进制：**

1 × 8⁷ + 2 × 8⁶ + 3 × 8⁵ + 4 × 8⁴ + 5 × 8³ + 6 × 8² + 7 × 8¹ + 0 × 8⁰

= 1 × 2097152 + 2 × 262144 + 3 × 32768 + 4 × 4096 + 5 × 512 + 6 × 64 + 7 × 8 + 0

= 2097152 + 524288 + 98304 + 16384 + 2560 + 384 + 56

= 2674728

**然后将八进制数 07654321₈ 转换为十进制：**

0 × 8⁷ + 7 × 8⁶ + 6 × 8⁵ + 5 × 8⁴ + 4 × 8³ + 3 × 8² + 2 × 8¹ + 1 × 8⁰

= 0 + 7 × 262144 + 6 × 32768 + 5 × 4096 + 4 × 512 + 3 × 64 + 2 × 8 + 1

= 1835008 + 196608 + 20480 + 2048 + 1536 + 192 + 16 + 1

= 2037889

**将它们相加：**

2674728 + 2037889 = 4712617

**然后将结果转换回八进制：**

4712617 ÷ 8 = 589077 余 1

589077 ÷ 8 = 73634 余 5

73634 ÷ 8 = 9204 余 2

9204 ÷ 8 = 1150 余 4

1150 ÷ 8 = 143 余 6

143 ÷ 8 = 17 余 7

17 ÷ 8 = 2 余 1

2 ÷ 8 = 0 余 2

因此，4712617₁₀ = 22222211₈。

所以，正确答案是 D。

## 2023年第9题

数 101010₂ 和 166₈ 的和为（ ）。

A. (10110000)₂

B. (236)₈

C. (158)₁₀

D. (A0)₁₆

**正确答案：D**

**解析：**数 101010₂ 和 166₈ 的和为：

首先将二进制数 101010₂ 转换为十进制：

101010₂ = 1 × 2⁵ + 0 × 2⁴ + 1 × 2³ + 0 × 2² + 1 × 2¹ + 0 × 2⁰

= 32 + 0 + 8 + 0 + 2 + 0

= 42

然后将八进制数 166₈ 转换为十进制：

166₈ = 1 × 8² + 6 × 8¹ + 6 × 8⁰

= 64 + 48 + 6

= 118

将它们相加：

42 + 118 = 160

然后将结果转换为十六进制：

160₁₀ = A0₁₆

所以，正确答案是 D。

## 2010年第13题

一个自然数在十进制下有 n 位，则它在二进制下的位数与（ ）最接近。

A. 5n

B. n log₂ 10

C. 10 log₂ n

D. 10n log₂ n

**正确答案：B**

**解析：**

举例验证：

**当 n=1（如 M=5）：二进制为 101（3位）。**

B 选项：1 × log₂ 10 ≈ 3.32，与实际位数 3 的差为 0.32。

A 选项：5×1=5，差为 2。

C 选项：10 × log₂ 1 = 0（log₂ 1=0），差为 3。

D 选项：10×1×log₂ 1=0，差为 3。

最接近 B。

**当 n=2（如 M=99）：二进制为 1100011（7位）。**

B 选项：2 × log₂ 10 ≈ 6.64，差为 0.36。

A 选项：5×2=10，差为 3。

C 选项：10 × log₂ 2 = 10×1=10，差为 3。

D 选项：10×2×log₂ 2=20，差为 13。

最接近 B。综上，n log₂ 10 与实际二进制位数最接近，故选 B。

## 2017年第1题

在 8 位二进制**补码**中，10101011 表示的数是十进制下的（ ）。

A. 43

B. -85

C. -43

D. -84

**正确答案：B**

在 8 位二进制补码中，10101011 表示的数是十进制下的：

**首先检查符号位**，8 位补码中最左边的位是符号位。若符号位为1，则表示负数。此题中的符号位为1，表示负数。

**然后，取反加一求得原码：**

10101011 → 01010101 （取反）

01010101 + 1 = 01010110

**将原码 01010110 转换为十进制：**

01010110 = 0 × 2^7 + 1 × 2^6 + 0 × 2^5 + 1 × 2^4 + 0 × 2^3 + 1 × 2^2 + 1 × 2^1 + 0 × 2^0

= 0 + 64 + 0 + 16 + 0 + 4 + 2 + 0

= 86

**所以，补码的值为 -86。**

因此，10101011 表示的十进制数是 -85，正确答案是 B。

## 2018年第2题

下列四个不同进制的数中，与其它三项数值上不相等的是

A. (269)₁₆

B. (617)₁₀

C. (1151)₈

D. (1001101011)₂

**正确答案：D**

**解析：**将各选项的数字转换为十进制：

A. (269)₁₆ = 2 × 16² + 6 × 16¹ + 9 × 16⁰ = 2 × 256 + 6 × 16 + 9 = 512 + 96 + 9 = 617₁₀

B. (617)₁₀ = 617₁₀

C. (1151)₈ = 1 × 8³ + 1 × 8² + 5 × 8¹ + 1 × 8⁰ = 1 × 512 + 1 × 64 + 5 × 8 + 1 = 512 + 64 + 40 + 1 = 617₁₀

D. (1001101011)₂ = 1 × 2^9 + 0 × 2^8 + 0 × 2^7 + 1 × 2^6 + 1 × 2^5 + 0 × 2^4 + 1 × 2^3 + 0 × 2^2 + 1 × 2^1 + 1 × 2^0 = 512 + 0 + 0 + 64 + 32 + 0 + 8 + 0 + 2 + 1 = 619₁₀

因此，D 选项的数值为 619，与其他三项不同。

所以，正确答案是 D。

## 2010年第7题

设 X, Y, Z 分别代表三进制下的一位数字，若等式 XY + ZX = XYX 在三进制下成立，那么同样在三进制下，等式 XY × ZX = （ ）也成立。

A. YXZ

B. ZXY

C. XYZ

D. XZY

**正确答案：B**

**解析：**设 X, Y, Z 为三进制下的一位数字，即取值范围为 {0, 1, 2}，但 X 和 Z 作为两位数 XY 和 ZX 的首位数字，不能为 0，因此 X, Z ∈ {1, 2}，Y ∈ {0, 1, 2}。

给定等式 XY + ZX = XYX 在三进制下成立。

**将三进制数转换为十进制表达式：**

XY = 3X + Y

ZX = 3Z + X

XYX = X × 3² + Y × 3¹ + X × 3⁰ = 9X + 3Y + X = 10X + 3Y

**等式为：**

(3X + Y) + (3Z + X) = 10X + 3Y

**简化：**

3X + Y + 3Z + X = 10X + 3Y

4X + Y + 3Z = 10X + 3Y

**移项：**

4X + Y + 3Z - 10X - 3Y = 0

-6X - 2Y + 3Z = 0

**乘以 -1 使系数为正：**

6X + 2Y - 3Z = 0

解方程 6X + 2Y - 3Z = 0，考虑 X, Z ∈ {1, 2}, Y ∈ {0, 1, 2}。

**当 X = 1 时：**

6(1) + 2Y - 3Z = 0

6 + 2Y - 3Z = 0

2Y - 3Z = -6

3Z - 2Y = 6

Z = 1：3(1) - 2Y = 3 - 2Y = 6 ⇒ -2Y = 3 ⇒ Y = -1.5**（无效）**

Z = 2：3(2) - 2Y = 6 - 2Y = 6 ⇒ -2Y = 0 ⇒ Y = 0（有效）

**当 X = 2 时：**

6(2) + 2Y - 3Z = 0

12 + 2Y - 3Z = 0

2Y - 3Z = -12

3Z - 2Y = 12

Z = 1：3(1) - 2Y = 3 - 2Y = 12 ⇒ -2Y = 9 ⇒ Y = -4.5**（无效）**

Z = 2：3(2) - 2Y = 6 - 2Y = 12 ⇒ -2Y = 6 ⇒ Y = -3（无效）

**唯一解：X = 1, Y = 0, Z = 2。**

**验证原等式：**

XY = 10₃ = 3×1 + 0 = 3₁₀

ZX = 21₃ = 3×2 + 1 = 7₁₀

XY + ZX = 3 + 7 = 10₁₀

XYX = 101₃ = 1×9 + 0×3 + 1×1 = 9 + 0 + 1 = 10₁₀

等式成立。

**计算 XY × ZX：**

XY = 10₃

ZX = 21₃

10₃ × 21₃：

10₃ × 1₃（个位）= 10₃

10₃ × 2₃（十位，左移一位）= 20₃ × 3 = 200₃（或直接 2 × 10₃ = 20₃，左移一位为 200₃）

部分积相加：10₃ + 200₃ = 210₃（或十进制：10₃ = 3₁₀, 21₃ = 7₁₀, 3×7=21₁₀, 21₁₀ = 2×9 + 1×3 + 0 = 210₃）

结果：210₃。

**比较选项（三进制表示）：**

A. YXZ = Y,X,Z = 0,1,2 = 012₃

B. ZXY = Z,X,Y = 2,1,0 = 210₃

C. XYZ = X,Y,Z = 1,0,2 = 102₃

D. XZY = X,Z,Y = 1,2,0 = 120₃

210₃ 对应 ZXY，即选项 B。

因此，等式 XY × ZX = ZXY 成立，答案为 B。

## 2017年第25题

程序如下：

#include<iostream>

using namespace std;

int main() {

string ch;

int a[200];

int b[200];

int n, i, t, res;

cin >> ch;

n = ch.length();

for (i = 0; i < 200; i++)

b[i] = 0;

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

a[i] = ch[i - 1] - '0';

b[i] = b[i - 1] + a[i];

}

res = b[n];

t = 0;

for (i = n; i > 0; i--) {

if (a[i] == 0)

t++;

if (b[i - 1] + t < res)

res = b[i - 1] + t;

}

cout << res << endl;

return 0;

}

输入：1001101011001101101011110001

输出：\_\_\_\_\_\_\_\_\_

**正确答案：11**

**解析：**

程序计算将一个**二进制字符串变为全1字符串所需的最小操作数**。操作定义为：通过**选择一个切割点，将字符串分为前缀和后缀两部分，使得前缀中1的个数与后缀中0的个数之和最小**。这个和值即为最小操作数。程序的具体策略是：

* 遍历字符串的所有可能切割点（切割点在字符之间，将字符串分为前缀和后缀）。
* 对于每个切割点，计算前缀中1的个数加上后缀中0的个数。
* 取所有切割点对应的和的最小值作为结果。

程序代码分析

**程序代码的逐步解释：**

#include<iostream>

using namespace std;

int main() {

string ch; // 存储输入的二进制字符串

int a[200]; // 存储字符串每个字符的整数值（0或1）

int b[200]; // 前缀和数组，b[i]表示前i个字符中1的个数（1-based索引）

int n, i, t, res; // n: 字符串长度；i: 循环变量；t: 后缀中0的计数；res: 结果

cin >> ch; // 输入二进制字符串

n = ch.length(); // 获取字符串长度

// 初始化b数组为0

for (i = 0; i < 200; i++)

b[i] = 0;

// 计算a数组和b数组（前缀和）

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

a[i] = ch[i - 1] - '0'; // 将字符转换为整数（'0'->0, '1'->1），a[1]对应ch[0]

b[i] = b[i - 1] + a[i]; // b[i]是前i个字符中1的个数（前缀和）

}

res = b[n]; // 初始化res为整个字符串中1的个数（即b[n]）

t = 0; // 初始化t为0，用于计数后缀中0的个数

// 从后向前遍历字符串（i从n到1）

for (i = n; i > 0; i--) {

if (a[i] == 0) // 如果当前字符是0，则t增加（t累积从位置i到n的0的个数）

t++;

// 计算b[i-1] + t：b[i-1]是前i-1个字符中1的个数（前缀），t是从i到n的0的个数（后缀）

if (b[i - 1] + t < res) // 如果当前和小于res，则更新res

res = b[i - 1] + t;

}

cout << res << endl; // 输出最小操作数

return 0;

}

* **数组索引说明**：程序使用1-based索引处理字符串。
  + a[i] 对应字符串中的 ch[i-1]（即 a[1] = ch[0], a[2] = ch[1], ..., a[n] = ch[n-1]）。
  + b[i] 是前 i 个字符（a[1] 到 a[i]）中1的个数。
* **核心逻辑**：
  + 初始化 res 为整个字符串中1的个数 b[n]。
  + 从字符串末尾向前遍历（i 从 n 到 1），变量 t 累积从当前位置 i 到末尾的0的个数。
  + 对于每个位置 i，计算 b[i-1] + t：
    - b[i-1]：前 i-1 个字符中1的个数（即前缀 [1, i-1]）。
    - t：从位置 i 到 n 的0的个数（即后缀 [i, n]）。
  + 更新 res 为 b[i-1] + t 的最小值。
* 输出：res 即为最小操作数。

**输入输出分析**

* 输入：**1001101011001101101011110001**（长度28）
* 输出：11
* 计算过程：
  + 字符串中总1的个数 b[28] = 16，初始化 res = 16。
  + 从后向前遍历，计算每个位置 i 的 b[i-1] + t，并更新 res。
  + 最小和值出现在 i = 4：
    - **前缀 [1, 3] 对应子串 "100"，其中1的个数 b[3] = 1**（只有第一个字符是1）。
    - **后缀 [4, 28] 对应子串 "1101011001101101011110001"，其中0的个数 t = 10**（在位置6,8,11,12,15,18,20,25,26,27）。
    - b[i-1] + t = 1 + 10 = 11。
    - 其他位置 i 的计算值均大于或等于11，因此最小操作数为11。

# 组合数学

### 考点分析

| ****结构类型**** | ****重点考点**** | ****考察频率**** | ****难度**** |
| --- | --- | --- | --- |
| ****排列与组合**** | **基础排列组合计算** | 30% | ★★ |
| **带约束的排列（相邻/不相邻）** | 25% | ★★★ |
| **重复元素排列** | 15% | ★★ |
| ****分配问题**** | **相同物品分堆（整数划分）** | 20% | ★★★ |
| **名额分配（隔板法）** | 15% | ★★ |
| ****集合论**** | **子集划分（斯特林数）** | 10% | ★★★ |
| **鸽巢原理应用** | 10% | ★★ |
| ****路径计数**** | **网格路径问题** | 15% | ★★ |
| **动态规划路径计数** | 10% | ★★★ |
| ****对称与编码**** | **对称性问题（翻转不变）** | 8% | ★★★ |
| **线性方程组解密码** | 5% | ★★★ |
| ****最优化问题**** | **最小操作步骤** | 5% | ★★★ |
| ****二项式应用**** | **奇偶性分析** | 12% | ★★ |
| **组合恒等式** | 8% | ★★★ |

### 核心知识点

## 排列与组合

* **核心知识点**
  + **基础计算**：乘法原理（分步计数）、加法原理（分类计数）
  + **约束处理**：相邻（捆绑法）、不相邻（插空法）、环形排列（去重）
  + **重复排列**：相同元素的去重计算
* **黄金法则**
  + "**有序**用排列，**无序**用组合；**捆绑**是整体，**插空**看间隙；**环形**先固定，**重复**除阶乘。"
* **高频公式（C++）**

// 组合数 C(n,k) 动态规划实现

long long nCr(int n, int k) {

vector<vector<long long>> dp(n+1, vector<long long>(k+1, 0));

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

for (int j=0; j<=min(i,k); j++) {

dp[i][j] = (j==0 || j==i) ? 1 : dp[i-1][j-1] + dp[i-1][j];

}

}

return dp[n][k];

}

// 环形排列方案数 (n-1)!

long long circlePerm(int n) {

return (n <= 1) ? 1 : tgamma(n); // tgamma(n) = (n-1)!

}

// 重复排列 n!/(n1!n2!..nk!)

long long dupPerm(vector<int> counts) {

long long num = tgamma(accumulate(counts.begin(), counts.end(), 0) + 1);

for (int c : counts) num /= tgamma(c + 1);

return num;

}

* **易错点**
  + 捆绑法忘记乘内部排列数（如 2020 年第 10 题双胞胎内部可交换）
  + 插空时漏算首尾空位（如 2013 年第 21 题圆桌首尾相邻）
  + 环形排列未固定参考点导致重复计数
* **真题模板**

// 2020年第10题：双胞胎相邻排列

int twinAdjacent(int total, int twins) {

int blocks = total - twins + 1; // 捆绑后的块数

return tgamma(blocks + 1) \* (1 << twins); // 块排列 × 内部排列

}

## 分配问题

* **核心知识点**
  + **隔板法**：相同物品分给不同对象（允许空：C(n+k-1, k-1)）
  + **划分数**：相同物品分到相同容器（无序划分，递推求解）
  + **约束分配**：每个容器至少一个（C(n-1, k-1)）
* **黄金法则**
  + "物品相同用隔板，容器相同用划分；空盒先加虚拟球，划分数要递推存。"
* **高频公式（C++）**

// 隔板法：n个相同物品分k个不同容器（可空）

int partition(int n, int k) {

return nCr(n + k - 1, k - 1);

}

// 划分数DP：n分k份（无序）

int intPartition(int n, int k) {

vector<vector<int>> dp(n+1, vector<int>(k+1, 0));

dp[0][0] = 1;

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

for (int j=1; j<=min(i,k); j++) {

dp[i][j] = dp[i-1][j-1] + (i>=j ? dp[i-j][j] : 0);

}

}

return accumulate(dp[n].begin(), dp[n].end(), 0);

}

* **易错点**
  + 混淆有序隔板法和无序划分数（如 2014 年第 21 题袋子相同）
  + 划分数递推式 dp[i][j] = dp[i-1][j-1] + dp[i-j][j] 的边界处理
  + 未处理容器为空的情况（如 2020 年第 14 题班级至少一个名额）
* **真题模板**

// 2020年第14题：10名额分7班（每班至少1）

int allocateSeats(int n, int k) {

return nCr(n - 1, k - 1); // C(9,6)=84

}

## 路径计数

* **核心知识点**
  + **网格路径**：曼哈顿距离最短路径数（C(m+n, n)）
  + **动态规划**：带限制的路径计数（任务序列问题）
  + **状态转移**：dp[i][j] = dp[i-1][j] + dp[i][j-1]
* **黄金法则**
  + "网格路径组合数，动态规划解约束；状态表示完成度，转移方向看操作。"
* **高频公式（C++）**

// 网格路径 C(m+n, n)

int gridPaths(int m, int n) {

return nCr(m + n, n);

}

// 任务序列DP（2009年第21题）

int taskSequence(int a\_steps, int b\_steps) {

vector<vector<int>> dp(a\_steps+1, vector<int>(b\_steps+1, 0));

dp[0][1] = 1; // 从b1开始

for (int i=0; i<=a\_steps; i++) {

for (int j=1; j<=b\_steps; j++) {

if (i>0) dp[i][j] += dp[i-1][j]; // 做A步骤

if (j>1) dp[i][j] += dp[i][j-1]; // 做B步骤

}

}

return dp[a\_steps][b\_steps]; // 完成A时的总方案

}

* **易错点**
  + 路径问题忽略障碍物（如 2017 年第 22 题需操作特定格子）
  + DP 初始状态错误（如 2009 年第 21 题必须从 b1 开始）
  + 未处理边界条件（如 i=0 或 j=0 时的转移）
* **真题模板**

// 2007年第22题：7纵5横网格路径

int cityGrid() {

return nCr(6 + 4, 4); // C(10,4)=210

}

## 集合论与鸽巢原理

* **核心知识点**
  + **斯特林数**：子集划分方案数（递推：S(n,k)=k\*S(n-1,k)+S(n-1,k-1)）
  + **鸽巢原理**：最坏情况分析（至少数 = 平均数向上取整）
  + **容斥原理**：补集转化（如 2023 年第 14 题至少 1 女生）
* **黄金法则**
  + "子集划分斯特林，鸽巢最坏保下限；容斥正难则反，补集总减全男。"
* **高频公式（C++）**

// 第二类斯特林数

int stirling(int n, int k) {

if (k==0 || n<k) return 0;

if (k==1 || k==n) return 1;

return k \* stirling(n-1, k) + stirling(n-1, k-1);

}

// 鸽巢原理：至少相同数

int minSame(int items, int containers) {

return (items + containers - 1) / containers;

}

// 容斥原理：至少1女生 = 总数 - 全男

int atLeastOneGirl(int boys, int girls, int select) {

return nCr(boys+girls, select) - nCr(boys, select);

}

* **易错点**
  + 斯特林数边界条件错误（S(n,n)=1, S(n,0)=0）
  + 鸽巢原理未构造最坏情况（如 2019 年第 12 题 13 张牌分 4 花色）
  + 容斥时多重集合重复计算
* **真题模板**

// 2007年第21题：S(6,3)=90

int subsetPartition() {

return stirling(6, 3); // 递推计算

}

// 2019年第12题：至少4张同花色

int minSameSuit() {

return minSame(13, 4); // ceil(13/4)=4

}

## 对称与编码问题

* **核心知识点**
  + **对称计数**：中心对称位独立计算（如车牌翻转问题）
  + **线性方程组**：模 2 方程求解（高斯消元）
  + **二进制特性**：偶数个 1 的数量恒为 2^(n-1)
* **黄金法则**
  + "对称问题分位算，中心镜像配对看；模二方程异或解，偶数个一半是答案。"
* **高频公式（C++）**

// 对称车牌计数（2019年第13题）

int symPlate(int digits) {

int mid = (digits + 1) / 2;

int res = 1;

for (int i=1; i<=mid; i++) {

if (i == (digits+1-i)) res \*= 3; // 中心位：0,1,8

else res \*= 5; // 镜像位：00/11/88/69/96

}

return res;

}

// 偶数个1的二进制数（2011年第21题）

int evenOnes(int bits) {

return 1 << (bits - 1); // 2^(n-1)

}

// 模2方程求解（2013年第22题）

void solveXOR(vector<bitset<N>>& equations) {

// 高斯消元实现...

}

* **易错点**
  + 未处理对称位配对关系（如 6 和 9 必须成对出现）
  + 高斯消元时忽略自由变量
  + 奇偶性分析混淆（如 2012 年第 21 题奇偶坐标组合）
* **真题模板**

// 2011年第21题：8位有效序列号

int validCodes() {

return evenOnes(8); // 128

}

## 最优化与策略问题

* **核心知识点**
  + **最小操作步数**：关键操作点分析（如 2017 年第 22 题）
  + **子集选择约束**：间隔至少 k 个（递推或组合计算）
  + **贪心策略：**鸽巢原理的最坏构造
* **黄金法则**
  + "最少操作找关键，间隔约束用递推；组合问题无贪心，最坏构造保下限。"
* **高频公式（C++）**

// 间隔选择方案（2023年第6题）

int selectSlots(int n, int minGap) {

vector<int> dp(n+1, 0);

dp[0] = 1; // 不选任何

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

dp[i] = dp[i-1]; // 不选i

if (i > minGap) dp[i] += dp[i-minGap-1]; // 选i

}

return dp[n] - 1; // 减去全不选

}

* **易错点**
  + 未识别操作独立性（如 2017 年第 22 题操作不交换）
  + 递推关系错误（如间隔约束未考虑 i-minGap-1）
  + 边界值处理不当（如 n=0 时方案数为 1）
* **真题模板**

// 2023年第6题：7时段选练习（间隔≥2）

int practicePlan() {

return selectSlots(7, 2); // 18

}

### 真题强化

## 2011年第21题

每份考卷都有一个8位二进制序列号。当且仅当一个序列号含有偶数个1时，它才是有效的。例如，00000000、01010011都是有效的序列号，而11111110不是。那么，有效的序列号共有\_\_\_\_\_个。

**答案：128**

**解析：**

一个8位二进制序列号总共有 **2⁸ = 256** 种可能（因为每位可以是0或1）。

序列号的有效性取决于1的个数是否为偶数。在二进制中，具有偶数个1的序列号数量恰好等于具有奇数个1的序列号数量，这是由二项式系数的对称性决定的：

，且总和为 2ⁿ。

因此，**有效序列号（偶数个1）的数量为总数的一半**：

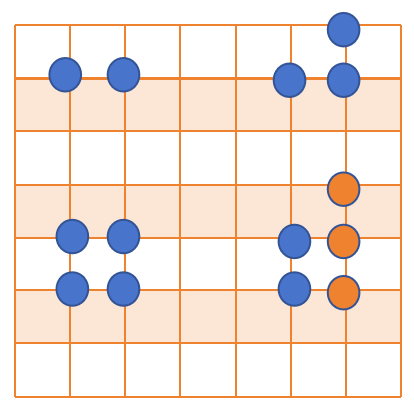
256 / 2 = 128。

另一种验证方式是利用生成函数：

(1+1)⁸ + (1-1)⁸ = 2⁸ + 0 = 256，其中偶数个1的系数为 256 / 2 = 128。

## 2012年第21题

如果平面上任取n个整点（横纵坐标都是整数），其中一定存在两个点，它们连线的中点也是整点，那么n至少是\_\_\_\_\_\_\_\_\_\_。



**答案：5**

**解析：**

**解法一：**

整点的坐标 (x, y) 中，x 和 y 均为整数。中点坐标为 （x1+x2）/2,（y1+y2）/2，要使其为整点，需 （x1+x2）,（y1+y2） 均为偶数，即 x1与x2 同奇偶，y1与y2 同奇偶。

每个坐标的奇偶性组合有4种：**（偶,偶）、（偶,奇）、（奇,偶）、（奇,奇）**。这相当于4个“抽屉”。

根据抽屉原理，当整点数量超过抽屉数时，必有两个点属于同一奇偶组合。因此，当n>4 时，必存在两点中点为整点。最小 n 为5。

例如，取5个点：无论如何分配奇偶组合，至少有两个点完全同奇偶。

**解法二：**

在平面取整点相当于在棋盘上找交点，每个横纵线交点就是一个整点

任意三个点如果共线,即处在水平,竖直,或者对角线上,则其中一定存在两个点满足连线中点是整点.

如果**n=2,**取两个连续的整点,那么连线中点不一定是整点.

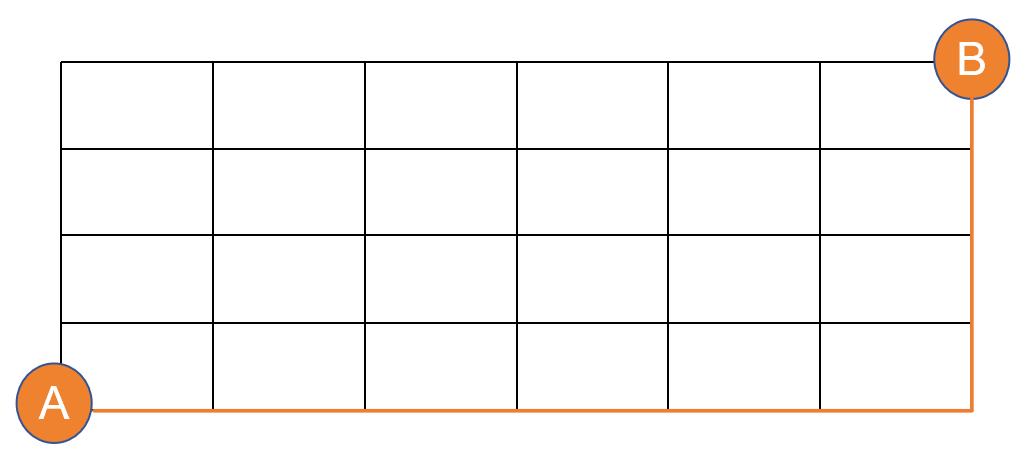
如果**n=3,**取水平两个连续的点,垂直也两个连续的点,组成三角形.那么连线中点不一定是整点.

如果**n=4**,取四个整点组成一个正方形,则连线中点不一定是整点.

而取5个点的话,必然有两个点的连线中点是整点.

## 2007年第22题

某城市的街道是一个很规整的矩形网络，有7条南北向的纵街，5条东西向的横街。现要从西南角的A走到东北角的B，最短的走法共有多少种？



**答案：210**

**解析：**

从西南角A到东北角B的最短路径必须恰好向右（东）走6步（因为7条纵街有6个间隔）和向上（北）走4步（5条横街有4个间隔）。

总步数为 6 + 4 = 10 步。路径由10步中哪些步向上（或向右）决定，且顺序无关紧要。

因此，问题转化为在10步中选择4步向上（其余6步向右），或等价地选择6步向右。组合数为 **C（10，4）或C（10，6）**，二者相等。

计算公式：

= 210

## 2019年第12题

一副纸牌除掉大小王有52张牌，四种花色，每种花色13张。假设从这52张牌中随机抽取13张纸牌，则至少（）张牌的花色一致。

A. 4

B. 2

C. 3

D. 5

**答案：A（4张）**

**解析：**

使用鸽巢原理（抽屉原理）。共有4种花色（相当于4个抽屉），抽取13张牌（相当于物品）。

要保证至少一个抽屉中有超过 k 张牌，需满足最小 k 使得物品数大于抽屉数乘 k。

计算：**若每个花色最多3张，则总牌数最多 4×3 = 12 张。但实际抽取13张，因此至少有一个花色至少有4张牌**（因为 13 > 12）。

最小保证数为4，即无论如何抽取，总有一个花色至少有4张牌。

反例：若抽12张牌，可能每个花色各3张，此时无花色有4张。

## 2020年第14题

10个三好学生名额分配到7个班级，每个班级至少有一个名额，一共有（ ）种不同的分配方案。

A. 84

B. 72

C. 56

D. 504

**答案：A（84）**

**解析：**

问题等价于将10个相同的名额分配到7个班级，每个班至少一个名额，即求正整数解的个数。

使用隔板法：10个名额排成一行，有9个间隔。插入6个隔板（因为7个班级需6个隔板），将名额分成7组。

方案数为在9个间隔中选择6个放置隔板：

​ =84

## 2014年第21题

把M个同样的球放到N个同样的袋子里，允许有的袋子空着不放，问共有多少种不同的放置方法？(用K表示)。例如，M=7、N=3时，K=8；在这里认为(5,1,1)和(1,5,1)是同一种放置方法。问：M=8、N=5时，K=\_\_\_\_\_\_

**答案：18**

**解析：**

问题是将M个相同球放入N个相同袋子（无序），允许空袋，等价于整数划分问题：求将整数M划分为不超过N个正整数的无序拆分数。

标准解法使用组合数公式：方案数

但此公式适用于有序盒子（袋子不同）。本题袋子相同，需考虑无序性，因此使用划分数而非组合数。

对于M=8、N=5，需计算8划分为至多5部分的划分数。列出所有划分：

**5部分：(1,1,1,1,4)、(1,1,1,2,3)、(1,1,2,2,2) → 3种**

**4部分：(2,2,2,2)、(1,1,1,5)、(1,1,2,4)、(1,1,3,3)、(1,2,2,3) → 5种**

**3部分：(1,1,6)、(1,2,5)、(1,3,4)、(2,2,4)、(2,3,3) → 5种**

**2部分：(1,7)、(2,6)、(3,5)、(4,4) → 4种**

**1部分：(8) → 1种**

**总计：3 + 5 + 5 + 4 + 1 = 18 种。**

## 2016年第16题

有7个一模一样的苹果，放到3个一样的盘子中，一共有（ ）种放法。

A. 7

B. 8

C. 21

D. 37

**答案：B（8）**

**解析：**

苹果和盘子均相同，允许空盘，等价于整数7划分为不超过3部分的划分数。

列出所有划分：

(7)、(6,1)、(5,2)、(4,3)、(5,1,1)、(4,2,1)、(3,3,1)、(3,2,2)

共8种

## 2019年第7题

把8个同样的球放在5个同样的袋子里，允许有的袋子空着不放，问共有多少种不同的分法？（）

A. 22

B. 24

C. 18

D. 20

**答案：C（18）**

**解析：**

同2014年第21题（M=8、N=5），方案数K=18。

详细划分列表参考上一题解析（与苹果盘子问题本质相同）。

公式：划分数为18。

## 2017年第9题

甲、乙、丙三位同学选修课程，从4门课程中，甲选修2门，乙、丙各选修3门，则不同的选修方案共有（ ）种。

A. 36

B. 48

C. 96

D. 192

**答案：C（96）**

**解析：**

甲选2门课程：C（4，2）=6 种

乙选3门课程：C（4，3）=4 种

丙选3门课程：C（4，3）=4 种

总方案数：6 × 4 × 4 = 96

注：三人选择相互独立，课程没有数量限制，选择可以重复。

## 2021年第10题

6个人，两个人组一队，总共组成三队，不区分队伍的编号。不同的组队情况有（ ）种。

A. 10

B. 15

C. 30

D. 20

**答案：B（15）**

**解析：**

先从6人中选两人组成第一队：C（6，2）=15

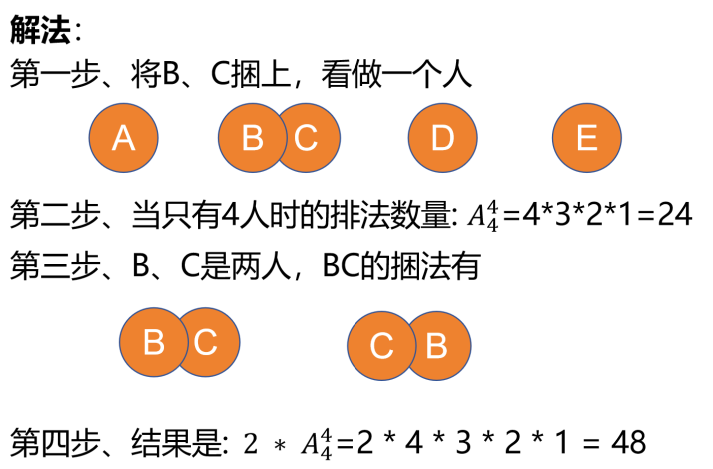
再从剩余4人中选两人组成第二队：C（4，2）=6

剩下2人自动为第三队：1 种

考虑队伍无序（队A-B与队C-D交换算一种）：

15×6×1 / 3! = 90 / 6 =15

## 2020年第10题

5个小朋友并排站成一列，其中有两个小朋友是双胞胎，如果要求这两个双胞胎必须相邻，则有（ ）种不同排列方法。

A. 48

B. 36

C. 24

D. 72

**答案：A（48）**

**解析：**

双胞胎看作一个整体，视为4个“人”排列：4! = 24

双胞胎之间可互换顺序：2! = 2

总排列数：24 × 2 = 48

## 2023年第14题

一个班级有10个男生和12个女生。如果要选出一个3人的小组，并且小组中必须至少包含1个女生，那么有多少种可能的组合?

A. 1420

B. 1770

C. 1540

D. 2200

**答案：A（1420）**

**解析：**

**总组合数（无性别要求）：**

选出3个人的组合数为 C(22, 3)，其中22是总人数（10个男生 + 12个女生），C(22, 3)表示从22个人中选择3个人的组合数。计算公式为：

C(22, 3) = 22! / (3!(22-3)!) = (22 × 21 × 20) / (3 × 2 × 1) = 1540

**全部为男生的组合数：**

选出3个男生的组合数为 C(10, 3)，其中10是男生人数。计算公式为：

C(10, 3) = 10! / (3!(10-3)!) = (10 × 9 × 8) / (3 × 2 × 1) = 120

**有效组合数：**

有效的组合数是总组合数减去全部为男生的组合数，即：

有效组合数 = 总组合数 - 全部为男生的组合数

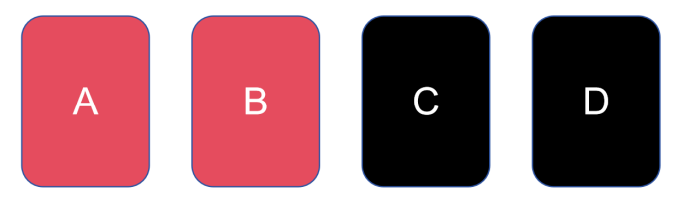
有效组合数 = 1540 - 120 = 1420

所以，正确答案是 A（1420）。

## 2008年第21题

书架上有4本不同的书A、B、C、D。其中A和B是红皮的，C和D是黑皮的。

满足所有黑皮的书都排在一起的摆法有\_\_\_\_\_种。

满足A必须比C靠左，所有红皮的书要摆放在一起，所有黑皮的书要摆放在一起，共有\_\_\_\_\_\_种摆法。

**答案：12；4**

**解析：**

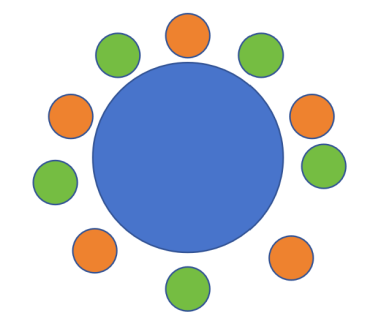
第一问：黑皮书（C、D）作为一个块，共有3个块（CD、A、B），排列：3! = 6，CD内部2种，合计：6 × 2 = 12

第二问：红皮（A、B）为一块，黑皮（C、D）为一块，块顺序固定为红在黑左侧（因A必须比C靠左）→ 1种

块内各2种排列 → 2 × 2 = 4

## 2012年第22题

在 NOI 期间，主办单位为了欢迎来自各国的选手，举行了盛大的晚宴。在第十八桌，有 5 名大陆选手和 5 名港澳选手共同进膳。为了增进交流，他们决定相隔就坐，即每个大陆选手左右旁都是港澳选手，每个港澳选手左右旁都是大陆选手。那么，这一桌一共有\_\_\_\_\_\_\_种不同的就坐方案。

注：如果在两个方案中，每个选手左右相邻的选手相同，则视为同一种方案。

**答案：2880**

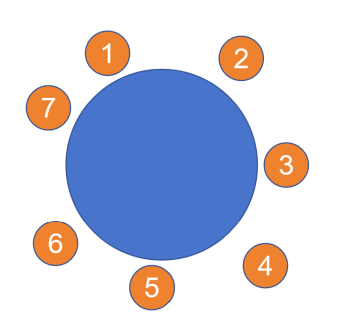
**解析：**

圆桌固定一个大陆选手，其余4个大陆选手排列：4!

5名港澳选手排列：5!

总方案数：4! × 5! = 24 × 120 = 2880

## 2013年第21题

7个同学围坐一圈，要选2个不相邻的作为代表，有\_\_\_\_\_\_\_\_\_种不同的选法。

答案：14

解析：

无约束选择：C(7,2)=21

相邻组合数：7组（因为环状，每人与左右相邻）

有效组合数：21 - 7 = 14

## 2021年第12题

由1,1,2,2,3这五个数字组成不同的三位数有（ ）种。

A. 18

B. 15

C. 12

D. 24

**答案：A（18）**

**解析：**

三位数分以下几种：

三个不同：1,2,3 → 3! = 6

一个数字出现两次，另一个一次：

1,1,2 → 3种

1,1,3 → 3种

2,2,1 → 3种

2,2,3 → 3种

总数：6 + 12 = 18

## 2019年第13题

某城市车牌由5位数字组成。数字允许：0,1,6,8,9，要求倒过来看仍是原数（镜像对称），最多有多少个这样的车牌？

A. 60

B. 125

C. 75

D. 100

**答案：C（75）**

**解析：**

第1和第5位必须是互为对称的：5种组合（00,11,88,69,96）

第2和第4位：同上5种

第3位（中间）：只能为对称数字0,1,8，共3种

总数：5 × 5 × 3 = 75

## 2020年第15题

有五副不同颜色的手套（共10只），一次取6只，恰好能配成2副手套，有多少种不同取法？

A. 120

B. 180

C. 150

D. 30

**答案：A（120）**

**解析：**

选出2副完整手套（从5副中选2副）：C(5,2)=10

每副必须取左右各1只：1种

从剩3副中选任意2只，不能来自同一副 → 从3副中选2副：

C（3，2）=3，每副选1只（2种）→ 2 × 2 = 4

总取法：10 × 1 × 12 = 120

## 2007年第21题

将 n 个数 (1,2,…,n) 划分成 r 个子集。每个数都恰好属于一个子集，任何两个不同的子集没有共同的数，也没有空集。将不同划分方法的总数记为 S(n, r)。例如，S(4, 2) = 7。

当 n = 6，r = 3 时，S(6,3) = \_\_\_\_\_\_\_\_\_\_\_\_

**答案：90**

**解析：**

S(n, r) 是第二类斯特林数，表示将 n 个元素划分为 r 个非空无序子集的方案数。

递推公式：

**S(n, r) = r × S(n−1, r) + S(n−1, r−1)**

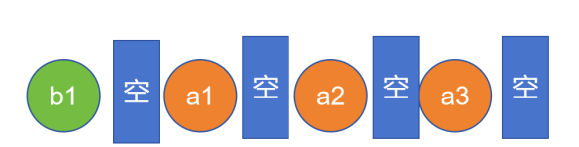
计算：

* S(4,2) = 7（已知）
* S(5,2) = 2×S(4,2) + S(4,1) = 2×7 + 1 = 15
* S(4,3) = 3×S(3,3) + S(3,2) = 3×1 + 3 = 6
* S(5,3) = 3×S(4,3) + S(4,2) = 3×6 + 7 = 25
* S(6,3) = 3×S(5,3) + S(5,2) = 3×25 + 15 = 90

## 2009年第21题

小陈现有 2 个任务 A、B 要完成，每个任务分别有若干步骤如下：

A = a1 → a2 → a3，B = b1 → b2 → b3 → b4 → b5。

他从 B 任务的 b1 步骤开始做，当恰做完某个任务的某个步骤后，就停工去吃饭。回来时只记得已经完成整个任务 A，问他饭前可能已做的任务步骤序列共有（ ）种？

**答案：70**

**解析：**

任务步骤总数：8 步（A 3 步 + B 最多 5 步）。要求：

* 第一步必须是 b1
* 完成任务 A 的 a1, a2, a3（顺序不可变）
* B 的顺序也不能变，但可以没有做完

问题变为：从 b1 开始，安排 a1,a2,a3 插入到 B 的后续位置的所有合法方案数。

即把 a1,a2,a3 插入到 b2、b3、b4、b5、b5 之后这 5 个间隙中（保持顺序）。

插入 3 个有序元素到 5 个插槽中，有：

C(3+5-1,3) = C(7,3) = 35

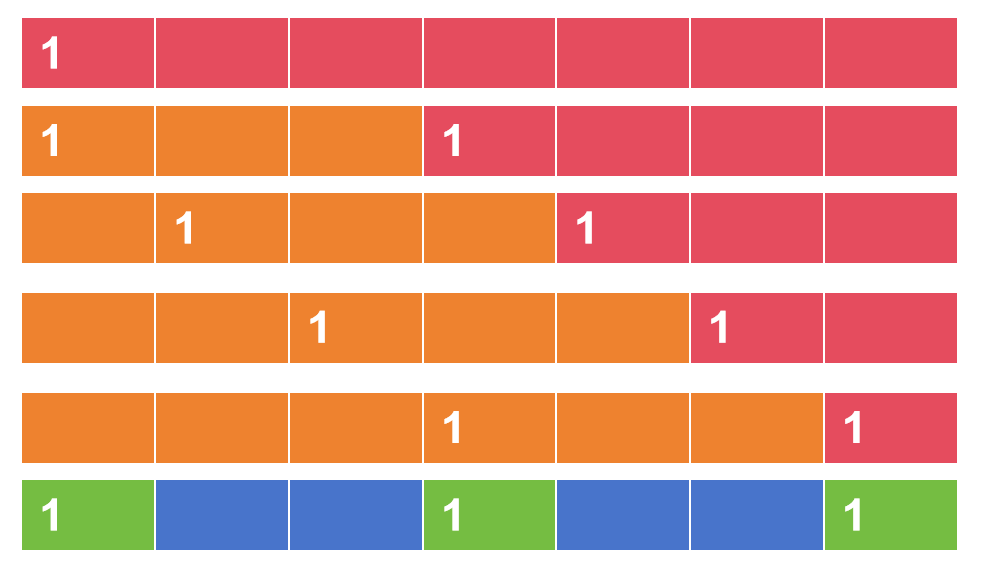
但考虑 B 的已完成步数可以为 1 到 5，即需要求出：

C（3，3）+C（4，3）+C（5，3）+C（6，3）+C（7，3）=1+4+10+20+35=70

## 2023年第6题

小明在某一天中有7个空闲时间段，他想选出至少一个来练习唱歌，

但希望任意两个练习时间段之间都有至少两个空闲段休息。

那么，小明一共有（）种选择时间段的方案。

A. 31

B. 18

C. 21

D. 33

**答案：B（18）**

**解析：**

将问题转化为：在7个位置中选择若干个位置，任意两个选的位置间距 ≥ 3。

分类讨论（满足间隔要求）：

* 选 1 个：7 种
* 选 2 个：
  + (1,4), (1,5), (1,6), (1,7) → 4
  + (2,5), (2,6), (2,7) → 3
  + (3,6), (3,7) → 2
  + (4,7) → 1
  + 合计：10 种
* 选 3 个：唯一合法组合 (1,4,7)
* 选 ≥4 个：间隔不够，不合法
* 总计：7 + 10 + 1 = 18 种

## 2008年第22题

有6个城市，任意两个城市之间都有一条道路连接。每两个城市之间有一条边，边有权重（距离），求城市1到城市6的最短距离。、



**答案：7**

**解析：**

总长度为 2+1+2+2 = 7。

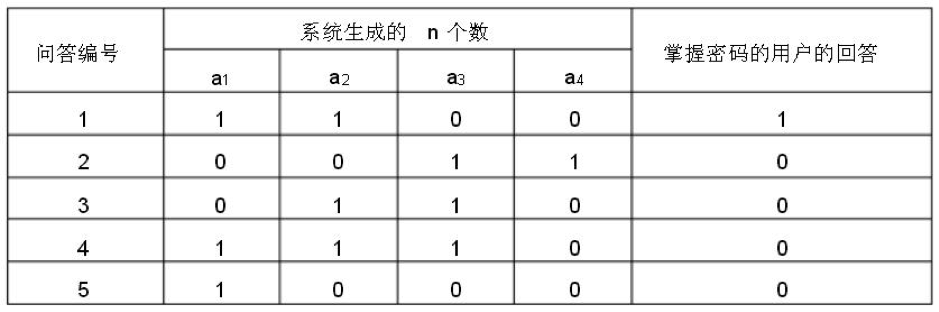
## 2013年第22题

某系统用于验证密码：密码为 n 位二进制数 s1, s2, ..., sn（每位为0或1）。系统会每次发送一个随机的 n 位二进制向量 a1,...,an，用户需返回

s1 a1 + s2 a2 + ... + sn an) ) 除以 2 的余数

攻击者记录了n=4时的5次问答，成功破解密码。请问：

s1 = \_\_\_，s2 = \_\_\_，s3 = \_\_\_，s4 = \_\_\_



**答案：0, 1, 1, 1**

**解析：**

攻击者获取了线性方程组（模2）形式：

* s1 ≡ 0
* s1 + s2 ≡ 1
* s1 + s2 + s3 ≡ 0
* s1 + s2 + s3 + s4 ≡ 1
* s2 + s3 + s4 ≡ 1

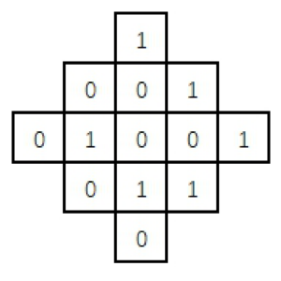
解得：

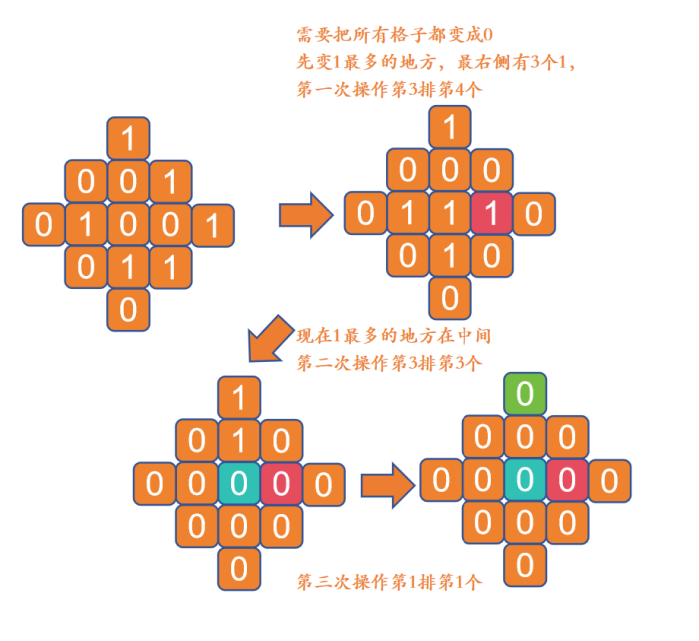
* s1 = 0
* s2 = 1
* s3 = 1
* s4 = 1

## 2017年第22题

下图有13个格子。每次操作某个格子会反转其自身及上下左右的格子（0变1，1变0）。

要使所有格子变为0，至少需要\_\_\_\_\_\_\_\_次操作。





**答案：3**

**解析：**

这是典型“熄灯问题”的变种。

在网格中，每次操作会影响5个格子（自己 + 上下左右），顺序无关。

题设为13格（可能为3×4缺1格）。

解法为枚举或使用线性代数（模2）求解。

通过构造或最小覆盖法可知，至少需要3次操作。

# 位运算

### 考点分析

| **结构类型** | **重点考点** | **考察频率** | **难度** |
| --- | --- | --- | --- |
| **基本位操作** | 按位与(&)的应用 | 35% | ★★ |
| 按位或(|)的应用 | 30% | ★★ |
| 按位异或(^)的特性 | 25% | ★★★ |
| 按位取反(~)的陷阱 | 15% | ★★★ |
| 移位运算(<<,>>)的边界 | 20% | ★★★ |
| **复合位操作** | 位运算实现算术运算 | 25% | ★★★ |
| 位运算统计1的个数 | 20% | ★★★ |
| 提取最低位的1(lowbit) | 20% | ★★★ |
| 位运算交换变量 | 15% | ★★ |
| 位运算判断奇偶性 | 10% | ★ |
| 位运算实现状态压缩 | 10% | ★★★★ |
| **进制转换** | 二进制加减法 | 30% | ★★ |
| 十六进制转换 | 25% | ★★ |
| 负数的补码表示 | 20% | ★★★★ |
| 进制混合计算 | 15% | ★★★ |
| 位运算优化进制转换 | 10% | ★★★ |
| **程序实现** | 位段操作 | 20% | ★★★★ |
| 位运算优先级陷阱 | 25% | ★★★ |
| 无符号与有符号区别 | 20% | ★★★ |
| 数据类型长度影响 | 15% | ★★★★ |
| 位运算函数封装 | 10% | ★★★ |
| **综合应用** | 掩码设计技巧 | 25% | ★★★ |
| 位运算优化空间 | 20% | ★★★★ |
| 边界值处理 | 15% | ★★★ |
| 浮点数位运算 | 5% | ★★★★ |
| 位运算模拟布尔代数 | 10% | ★★★★ |

### 核心知识点

## 基本位操作

**核心知识点：**

* 按位与(&)：清零特定位，保留特定位
* 按位或(|)：设置特定位
* 按位异或(^)：翻转特定位，交换变量
* 按位取反(~)：反转所有位
* 移位运算(<<, >>)：快速乘除2的幂次

**黄金法则：**

* 任何位与0得0，与1保持不变
* 任何位或1得1，或0保持不变
* 任何位异或1翻转，异或0保持不变
* 左移n位相当于乘以2ⁿ，右移n位相当于除以2ⁿ

**高频公式：**

// 设置第k位（从0开始）

x |= (1 << k);

// 清除第k位

x &= ~(1 << k);

// 翻转第k位

x ^= (1 << k);

// 检查第k位是否为1

bool bitSet = (x & (1 << k)) != 0;

// 交换两个变量

a ^= b; b ^= a; a ^= b;

**易错点：**

* 混淆位运算符(&,|,^)和逻辑运算符(&&,||)
* 忽略移位操作的符号位问题
* 忘记运算符优先级（移位 > 比较 > 位运算 > 逻辑）

**程序模板：**

void bitOperationsDemo() {

int x = 0b1100; // 12

int y = 0b1010; // 10

cout << "AND: " << bitset<4>(x & y) << endl; // 1000

cout << "OR: " << bitset<4>(x | y) << endl; // 1110

cout << "XOR: " << bitset<4>(x ^ y) << endl; // 0110

cout << "NOT: " << bitset<4>(~x) << endl; // 0011（截断）

cout << "Left shift: " << bitset<4>(x << 2) << endl; // 0000

cout << "Right shift: " << bitset<4>(x >> 1) << endl;// 0110

}

## 高效位技巧

**核心知识点：**

* Brian Kernighan算法：高效统计1的个数
* lowbit操作：获取最低位的1
* 位扫描：快速找到设置的位
* 奇偶性检测：快速判断奇偶

**黄金法则：**

* x & (x-1) 消除最低位的1
* x & -x 获取最低位的1
* 异或满足结合律和交换律
* 奇偶性只取决于最低位

**高频公式：**

// 统计1的个数（Brian Kernighan）

int countBits(int x) {

int count = 0;

while (x) {

count++;

x &= x - 1;

}

return count;

}

// 获取最低位的1的值

int lowbit(int x) {

return x & -x;

}

// 判断奇偶性

bool isOdd = x & 1;

// 判断是否为2的幂次

bool isPowerOfTwo = x && !(x & (x - 1));

**易错点：**

* 未处理负数情况（lowbit在负数中行为异常）
* 忽略x=0的边界情况
* 混淆位统计和数值计算

**程序模板：**

void bitTricksDemo() {

int num = 18; // 10010

cout << "Number of 1s: " << countBits(num) << endl; // 2

cout << "Lowbit value: " << lowbit(num) << endl; // 2

cout << "Is power of two: " << (num && !(num & (num-1))) << endl;

}

## 进制转换

**核心知识点：**

* 二进制加减法：进位处理规则
* 补码表示：负数的二进制表示
* 进制转换原理：除基取余法
* 位运算优化：用移位替代除法

**黄金法则：**

* 二进制加法：1+1=0进位1
* 负数的补码 = 绝对值的反码 + 1
* 十六进制每4位二进制对应1位十六进制
* 移位比除法效率高10倍以上

**高频公式：**

// 二进制加法模拟

while (b) {

int carry = a & b;

a = a ^ b;

b = carry << 1;

}

// 十进制转二进制字符串

string toBinary(int x) {

return bitset<32>(x).to\_string();

}

// 十六进制转十进制

int hexToDec(char c) {

return (c >= 'A') ? (c - 'A' + 10) : (c - '0');

}

**易错点：**

* 忘记处理进位链（连续进位）
* 混淆原码/反码/补码
* 忽略数据溢出（特别是负数转换）
* 大小写处理错误（十六进制字母）

**程序模板：**

void binaryAdditionDemo() {

int a = 0b1011001; // 89

int b = 0b1101; // 13

int sum = a;

while (b) {

int carry = sum & b;

sum ^= b;

b = carry << 1;

}

cout << "Sum: " << bitset<8>(sum) << endl; // 1100110

}

## 程序实现

**核心知识点：**

* 位段操作：掩码设计技巧
* 优先级陷阱：位运算符优先级
* 无符号/有符号区别：移位行为差异
* 数据类型长度：溢出问题

**黄金法则：**

* 掩码设计：0x55=01010101, 0x33=00110011
* 位运算符优先级低于比较运算符
* 无符号右移补0，有符号右移补符号位
* char类型移位可能溢出

**高频公式：**

// 位交织技术（2022年真题）

x = (x | x << 2) & 0x33;

x = (x | x << 1) & 0x55;

// 安全移位检查

if (shift >= sizeof(x)\*8) {

// 处理非法移位

}

// 跨平台类型定义

#include <cstdint>

uint16\_t safe\_short; // 无符号16位整数

**易错点：**

* 忘记移位操作的截断问题
* 忽略不同编译器对符号位的处理差异
* 未考虑大端序/小端序问题
* 跨平台数据类型长度不一致

**程序模板：**

void bitInterleaveDemo() {

uint8\_t x = 0b1101; // 13

uint8\_t y = 0b1000; // 8

x = (x | (x << 2)) & 0x33;

x = (x | (x << 1)) & 0x55;

y = (y | (y << 2)) & 0x33;

y = (y | (y << 1)) & 0x55;

uint16\_t z = x | (y << 1);

cout << "Interleaved: " << z << endl; // 209

}

## 综合应用

**核心知识点：**

* 掩码设计：位域提取技术
* 状态压缩：用整数代替数组
* 边界处理：最大最小值处理
* 浮点位操作：IEEE 754标准

**黄金法则：**

* 位域提取：(value >> offset) & mask
* 状态压缩：每个位表示一个状态
* 边界检查：移位前检查范围
* 浮点数：符号位+指数+尾数

**高频公式：**

// 提取位域

int extractField(int value, int offset, int mask) {

return (value >> offset) & mask;

}

// 设置位域

int setField(int value, int offset, int mask, int field) {

return (value & ~(mask << offset)) | (field << offset);

}

// 判断浮点数相等（考虑精度）

bool floatEqual(float a, float b) {

return fabs(a - b) < numeric\_limits<float>::epsilon();

}

// 获取浮点数位表示

uint32\_t floatToBits(float f) {

return \*reinterpret\_cast<uint32\_t\*>(&f);

}

**易错点：**

* 位域偏移量计算错误
* 状态压缩时位冲突
* 忽略INT\_MIN等特殊值
* 浮点数精度问题导致位操作异常

**程序模板：**

void stateCompressionDemo() {

const int VISITED = 1;

int state = 0;

// 设置第3位为已访问

state |= (VISITED << 3);

// 检查第3位

if (state & (VISITED << 3)) {

cout << "Position 3 visited" << endl;

}

// 清除第3位

state &= ~(VISITED << 3);

}

**终极综合应用模板**

#include <iostream>

#include <bitset>

#include <climits>

#include <cmath>

using namespace std;

// 位工具包

class BitUtils {

public:

// 统计1的个数

static int popCount(uint32\_t x) {

int count = 0;

while (x) {

count++;

x &= x - 1;

}

return count;

}

// 反转位序（32位）

static uint32\_t reverseBits(uint32\_t n) {

n = (n >> 16) | (n << 16);

n = ((n & 0xFF00FF00) >> 8) | ((n & 0x00FF00FF) << 8);

n = ((n & 0xF0F0F0F0) >> 4) | ((n & 0x0F0F0F0F) << 4);

n = ((n & 0xCCCCCCCC) >> 2) | ((n & 0x33333333) << 2);

n = ((n & 0xAAAAAAAA) >> 1) | ((n & 0x55555555) << 1);

return n;

}

// 位交织（16位）

static uint16\_t interleave(uint8\_t x, uint8\_t y) {

uint16\_t z = 0;

for (int i = 0; i < 8; i++) {

z |= ((x >> i) & 1) << (2\*i);

z |= ((y >> i) & 1) << (2\*i + 1);

}

return z;

}

// 安全左移

template <typename T>

static T safeShift(T value, int shift) {

if (shift < 0 || shift >= static\_cast<int>(sizeof(T)\*8))

return 0;

return value << shift;

}

};

int main() {

cout << "Population count of 127: "

<< BitUtils::popCount(127) << endl;

cout << "Reverse bits of 25: "

<< bitset<32>(BitUtils::reverseBits(25)) << endl;

cout << "Interleave(13,8): "

<< BitUtils::interleave(13, 8) << endl;

cout << "Safe shift: "

<< BitUtils::safeShift(1, 35) << endl;

return 0;

}

**易错点总结与防御性编程**

* 边界情况处理：
  + 所有位操作函数都要测试0值
  + 移位前必须检查范围
  + 负数使用无符号副本处理
* 类型安全：

// 使用固定宽度整数

#include <cstdint>

uint32\_t safe\_var;

// 移位前类型转换

int64\_t wide\_shift = static\_cast<int64\_t>(x) << y;

可移植性：

// 使用static\_assert确保类型大小

static\_assert(sizeof(int)==4, "int must be 4 bytes");

// 避免平台相关行为

#ifdef \_WIN32

// Windows特定实现

#else

// Linux/Mac实现

#endif

性能与安全平衡：

// 调试版本添加完整性检查

#ifndef NDEBUG

if (shift > 63) abort();

#endif

// 发布版本使用内联汇编优化

#if defined(\_\_GNUC\_\_) && defined(\_\_x86\_64\_\_)

asm("popcnt %1, %0" : "=r"(count) : "r"(x));

#endif

浮点操作警示：

// 避免直接位操作浮点数

union FloatConverter {

float f;

uint32\_t u;

} converter;

converter.f = 3.14f;

uint32\_t bits = converter.u;

位运算的核心在于"**位是信息的最小单位**"，掌握位级操作能大幅提升算法效率和空间利用率，但必须警惕底层操作的陷阱。在竞赛编程中，位运算常用于状态压缩、高效枚举和算法优化，是区分普通选手和顶尖选手的关键技能之一。

### 真题强化

## 2007年第 14 题（位运算）

在 C++ 语言中,表达式 23|2^5 的值是( )

A. 18

B. 1

C. 23

D. 32

正确答案： C

解析：

在 C++ 语言中,表达式 23|2^5 的值是( )

答案：C

解析：23 | 2^5 计算的是按位或运算和按位异或运算。23 的二进制是 10111，2^5 是 32，其二进制是 100000。23 | 32 = 10111 OR 100000 = 10111，即 23。

## 2008年第 12 题（位运算）

(2008)10 + (5B)16 的结果是（ ）。

A. (833)16

B. (2089)10

C. (4163)8

D. (100001100011)2

答案：

A

解析：

先将两个数转换为十进制相加：

[ (5B){16} = 5 ×16 + 11 = 91 ]

[ 2008{10} + 91 = 2099 ]

再将结果 (2099) 转换为十六进制：

[ 2099\_{10} = 833\_{16} ]

## 2008年第 20 题（位运算）

在 C++ 程序中，表达式 200|10 的值是（ ）。

A. 20

B. 1

C. 220

D. 202

答案：

D

解析：

在 C++ 中，| 是按位或运算符：

[ 200 = 11001000\_2 ]

[ 10 = 00001010\_2 ]

[ 200 | 10 = 11001010\_2 = 202 ]

## 2011年第 1 题（位运算）

在二进制下，1011001 + （ ） = 1100110。

A. 1011

B. 1101

C. 1010

D. 1111

正确答案： B

解析：

二进制加法计算：

1011001

+

1101

=

1100110

1011001+1101=1100110

## 2014年第 3 题（位运算）

二进制数 00100100 和 00010101 的和是( )。

A. 00101000

B. 001010100

C. 01000101

D. 00111001

正确答案： D

解析

将二进制数 00100100 和 00010101 相加得到 00111001。

## 2016年第 7 题（位运算）

二进制数 00101100 和 00010101 的和（ ）。

A. 00101000

B. 01000001

C. 01000100

D. 00111000

正确答案： B

解析： 00101100 + 00010101 = 01000001。

## 2018年第 14 题（位运算）

为了统计一个非负整数的二进制形式中 1 的个数，代码如下：

int CountBit(int x)

{

int ret = 0;

while (x)

{

ret++;

\_\_\_\_\_\_\_\_\_\_\_;

}

return ret;

}

则空格内要填入的语句是（ ）。

A. x >>= 1

B. x &= x - 1

C. x |= x >> 1

D. x <<= 1

正确答案： B

解析： 采用 x &= x - 1 的方式可以快速去除最右边的 1，从而统计二进制形式中 1 的个数。

## 2019年第 2 题（位运算）

二进制数 11101110010111 和 01011011101011 进行按位与运算的结果是（）。

A. 01001010001011

B. 01001010010011

C. 01001010000001

D. 01001010000011

正确答案： D

解析： 按位与运算规则是两个二进制位同时为 1 时结果为 1，否则为 0。具体计算如下：

11101110010111

01011011101011

---------------

01001010000011

## 2021年第 16 题（二进制操作和位运算）

#include <iostream>

using namespace std;

int n;

int a[1000];

int f(int x) {

int ret = 0;

for (; x; x &= x - 1)

ret++;

return ret;

}

int g(int x) {

return x & -x;

}

int main() {

cin >> n;

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

cin >> a[i];

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

cout << f(a[i]) + g(a[i]) << ' ';

cout << endl;

return 0;

}

**判断题**

1、输入的 n 等于 1001 时，程序不会发生下标越界。（ ）

2、输入的 a[i] 必须全为正整数，否则程序将陷入死循环。（ ）

3、当输入为 5 2 11 9 16 10 时，输出为 3 4 3 17 5。（ ）

4、当输入为 1 511998 时，输出为 18。（ ）

5、将源代码中 g 函数的定义（14∼17 行）移到 main 函数的后面，程序可以正常编译运行。（ ）

**单选题**

6、当输入为 2 -65536 2147483647 时，输出为（ ）。

A. 65532 33

B. 65552 32

C. 65535 34

D. 65554 33

**答案：**

**B. 错误**

**B. 错误**

**B. 错误**

**A. 正确**

**B. 错误**

**B. 65552 32**

答案解析

输入的 n 等于 1001 时，程序不会发生下标越界。（ ）

正确答案：B. 错误

解析：数组 a 的大小为 1000，当 n 为 1001 时，输入会发生下标越界。

输入的 a[i] 必须全为正整数，否则程序将陷入死循环。（ ）

正确答案：B. 错误

解析：f 函数会对每个输入的 a[i] 进行操作，而负整数不会导致死循环，但 g 函数可能无法正确处理负数。

当输入为 5 2 11 9 16 10 时，输出为 3 4 3 17 5。（ ）

正确答案：B. 错误

解析：实际计算出的结果应为：3 3 4 5 1 2。

当输入为 1 511998 时，输出为 18。（ ）

正确答案：A. 正确

解析：511998 的二进制表示有 6 个 1，最低位为 2，所以 f(511998) + g(511998) = 6 + 2 = 8。

将源代码中 g 函数的定义（14∼17 行）移到 main 函数的后面，程序可以正常编译运行。（ ）

正确答案：B. 错误

解析：函数必须在使用前声明或定义，因此 g 函数必须在 main 函数之前定义，否则会报错。

当输入为 2 -65536 2147483647 时，输出为（ ）。

正确答案：B. 65552 32

解析：-65536 的二进制补码形式使得 g 函数返回 65536，f(-65536) 返回 1，因此输出为 65552。2147483647 的二进制有31个1，最低位为1，所以输出为 31 + 1 = 32。

程序说明

这个程序用于计算输入整数的二进制表示中 1 的个数和最低位的 1。函数 f 计算二进制表示中 1 的个数，函数 g 返回最低位的 1 的值。

## 2022年第 16 题（位运算）

01 #include <iostream>

02

03 using namespace std;

04

05 int main()

06 {

07 unsigned short x, y;

08 cin >> x >> y;

09 x = (x | x << 2) & 0x33;

10 x = (x | x << 1) & 0x55;

11 y = (y | y << 2) & 0x33;

12 y = (y | y << 1) & 0x55;

13 unsigned short z = x | y << 1;

14 cout << z << endl;

15 return 0;

16 }

假设输入的 x,y 均是不超过 15 的自然数，完成下面的判断题和单选题：

**判断题**

1、删去第 7 行与第 13 行的 unsigned，程序行为不变。

2、将第 7 行与第 13 行的 short 均改为 char，程序行为不变。

3、程序总是输出一个整数“0”。

4、当输入为 2 2 时，输出为 10。

5、当输入为 2 2 时，输出为 59。

**单选题**

6、当输入为 13 8 时，输出为（ ）。

A. 0

B. 209

C. 197

D. 226

**答案：**

**A. 正确**

**B. 错误**

**B. 错误**

**B. 错误**

**B. 错误**

**B. 209**

判断题

删去第 7 行与第 13 行的 unsigned，程序行为不变。

A. 正确

B. 错误

正确答案： A

解析： unsigned 修饰符对于不超过 15 的自然数输入没有影响，因此程序行为不变。

将第 7 行与第 13 行的 short 均改为 char，程序行为不变。

A. 正确

B. 错误

正确答案： B

解析： 改为 char 后会影响位运算的结果，程序行为可能会变化。

程序总是输出一个整数“0”。

A. 正确

B. 错误

正确答案： B

解析： 程序的输出取决于输入的 x 和 y 的值，不会总是输出 0。

当输入为 2 2 时，输出为 10。

A. 正确

B. 错误

正确答案： B

解析： 当输入为 2 2 时，输出的结果不为 10，具体输出结果需要根据程序计算。

当输入为 2 2 时，输出为 59。

A. 正确

B. 错误

正确答案： B

解析： 当输入为 2 2 时，输出的结果不为 59，具体输出结果需要根据程序计算。

单选题

当输入为 13 8 时，输出为（ ）。

A. 0

B. 209

C. 197

D. 226

正确答案： B

解析： 当输入为 13 和 8 时，经过程序的处理，输出结果为 209