2026 数据结构期末复习讲座

曹泽阳

Z4T155664@163.com

2026 年 1 月 15 日

Overview

基础语法要点

程序设计基础

程序设计基础

- 基础语法要点
- 代码和笔记详见 for25 目录

基础语法要点

- C++ 是一种静态类型的、编译式的、通用的、大小写敏感的、不规则的编程语言，支持过程化编程、面向对象编程和泛型编程。
- C++ 是 C 语言的一个超集，事实上任何合法的 C 程序都是合法的 C++ 程序。
- C++ 中的标准库是 STL。

头文件

头文件

- 头文件是一种包含 C++ 函数声明和宏定义的文件, 通常具有.h 扩展名。
- 常用的头文件有 iostream、fstream、cmath、string,algorithm,vector,map 等。
- 推荐使用万能头:#include<bits/stdc++.h>, 包含了所有的标准库。

变量类型

变量类型

常用变量类型

1. C++ 中的基本数据类型有 int、char、float、double、bool,long long 等。
2. STL 中的数据类型有 string、vector、map、set、queue、stack,pair<>,tuple<> 等。
3. 自定义变量类型:struct 和 class。

技巧

1. int 范围太小, 有时候需要使用 long long。一般建议无脑使用 long long。
2. 使用 auto 关键字可以自动推断变量类型。
3. 使用 const 关键字可以定义常量。
4. 使用 static 关键字可以定义静态变量。

输入输出

输入输出

- cin 和 cout 是 C++ 中的标准输入输出流对象。
- cin 用于从标准输入设备 (键盘) 读取数据,cout 用于向标准输出设备 (显示器) 输出数据。
- cin 和 cout 是 C++ 中的标准输入输出流对象。
- cin,cout 需要使用 using namespace std; 来使用。

字符串读取

字符串读取

- cin 读取字符串时，遇到空格、制表符、换行符就会停止读取。通常来说,cin 会将换行符留在输入缓冲区中。这与 getline 函数连用时会产生问题。
- getline 函数可以读取一整行字符串，包括空格、制表符、换行符等。getline 会将换行符从输入缓冲区中取出，但会扔掉。

字符串读取

- cin 和 getline 可以混用，但是需要注意 cin 留在输入缓冲区的换行符可能会被 getline() 读取。也就是说，如果在 cin 后，先 cin.ignore() 再 getline，就可以避免这个问题。
- cin.ignore() 函数用于清除输入缓冲区中的字符，通常用于清除换行符。
- 请同学们搭配代码 testio.cpp 学习

格式化输出

格式化输出-流操纵符

- 格式化输出是指按照一定的格式输出数据，可以控制输出的宽度、精度、对齐方式等。
- 格式化输出可以使用流操纵符 setw、setfill、setprecision、setiosflags 等。
- setw(n): 设置输出宽度为 n 个字符。
- setfill(c): 设置填充字符为 c。
- setprecision(n): 设置浮点数的精度为 n 位。
- setiosflags(ios::fixed): 设置浮点数的输出格式为定点表示法。
- 然而，相比于 printf,cout 的格式化输出功能较弱，所以在 ACM 竞赛中,printf 更常用。

printf 格式化输出

- printf 是 C 语言中的标准输出函数，可以用于格式化输出。
- printf 的格式化输出功能非常强大，可以控制输出的宽度、精度、对齐方式等。
- printf 的格式化输出使用格式字符串，格式字符串中的格式控制符用于指定输出的格式。
- printf 的格式控制符包括:%d、%f、%s、%c、%x、%o、%u、%e、%g 等。

printf 格式化输出示例

占位符

- %d: 整数
- %f: 浮点数
- %c: 字符
- %s: 字符串
- %x: 十六进制
- %o: 八进制
- %u: 无符号整数
- %e: 科学计数法
- %g: 自动选择 f 或 e

修饰符

- %md: 输出宽度为 m，右对齐，左边补空格
- %0md: 输出宽度为 m，右对齐，左边补 0
- %-md: 输出宽度为 m，左对齐，右边补空格
- %m.nf: 宽度 m，小数点后 n 位
- %.nf: 小数点后保留 n 位
- %m.ne: 宽度 m，科学计数法，小数点后 n 位
- %m.ng: 宽度 m，自动选择格式，小数点后 n 位

注: 实际输出内容超过指定宽度时会完整显示，对于整数是这样，而浮点数不是。可查看 testprintf.cpp

printf 格式化输出示例

```
%d: 占位符示例：整数：42
%ld: 占位符示例：长整数：1234567890
%lld: 占位符示例长长整数：123456789012345
%f: 占位符示例浮点数：3.141590
%c: 占位符示例字符：A
%s: 占位符示例字符串：Hello, World!
%x: 占位符示例十六进制：2a
%o: 占位符示例八进制：52
%u: 占位符示例无符号整数：300
%e: 占位符示例科学计数法：1.230000e-04
%g: 占位符示例自动格式：123457

修饰符示例%5d: 修饰符示例宽度为5的整数：    42
%-5d: 修饰符示例左对齐宽度为5的整数：42
%05d: 修饰符示例宽度为5的整数，前面补0：00042
%+d: 修饰符示例带符号整数：+42
%.2f: 修饰符示例保留两位小数的浮点数：3.14
%10.2f: 修饰符示例宽度为10，保留两位小数的浮点数，右对齐：      3.14
%-10.2f: 修饰符示例宽度为10，保留两位小数的浮点数，左对齐：3.14
%010.2f: 修饰符示例宽度为10，保留两位小数的浮点数，前面补0：0000003.14
测试超过指定宽度：宽度为5，实际长度为10的整数：1234567890
宽度为5，实际长度为10的浮点数：12345.68
```

四舍五入

四舍五入

- 四舍五入是指将一个数值保留到指定的小数位数，并根据下一位的数值决定是否进位。
- 在 C++ 中，可以使用 round 函数进行四舍五入。round 函数返回最接近的整数。
- 例如:round(3.14) 返回 3,round(3.56) 返回 4。
- 如果需要保留小数位，可以将数值乘以 10 的幂次方，进行四舍五入后再除以 10 的幂次方。
- 例如: 保留两位小数的四舍五入:round(3.14159 * 100) / 100.0 返回 3.14。

数据结构串讲

数据结构串讲

- 基础数据结构
- 基础算法
- 代码详见 for24 目录，部分代码来自网络题解

基础数据结构

基础数据结构

- 线性表:
  - 顺序表、链表、栈、队列
- 树:
  - 二叉树、二叉搜索树、平衡二叉树
- 图:
  - 邻接矩阵、邻接表、深度优先搜索、广度优先搜索, 最短路径, 最小生成树, 拓扑排序,Tarjan 算法
- 哈希表:
  - 哈希函数、冲突解决、哈希集合、哈希映射

线性表与双指针

线性表与双指针技巧

线性表

- 顺序表
  - 基于数组实现，支持随机访问，插入与删除操作时间复杂度高
- 单链表
  - 每个节点包含数据部分和指向下一个节点的指针
  - 动态内存分配，插入与删除操作高效
- 双链表
  - 每个节点包含数据部分、指向前一个节点和后一个节点的指针
  - 支持双向遍历，操作更加灵活
- 循环链表

双指针技巧

- 快慢指针
  - 用于检测链表中的环
  - 快指针每次移动两步，慢指针每次移动一步
- 对撞指针
  - 从序列的两端同时移动指针
  - 常用于查找特定元素对，如两数之和
- 滑动窗口
  - 动态维护一个窗口来解决子数组或子字符串问题
  - 常用于最小覆盖子串、最长无重复子串等问题

寻找链表中的中间节点

- 方法: 使用快慢指针，快指针每次移动两步，慢指针每次移动一步。
- 步骤:
  1. 初始化快指针和慢指针均指向链表头部。
  2. 快指针每次移动两步，慢指针每次移动一步。
  3. 当快指针到达链表末尾时，慢指针所在的位置即为中间节点。
- 公式证明: 设链表长度为 n。快指针每次移动 2 步，慢指针每次移动 1 步。当快指针移动到链表末尾时，快指针总共移动了 $2k$ 步，慢指针移动了 k 步。因为快指针到达末尾时，$2k \geq n$，且 $k = \left\lfloor \frac{n}{2} \right\rfloor$。因此，慢指针移动了 $\left\lfloor \frac{n}{2} \right\rfloor$ 步，位于中间节点。

检测链表中的环

- 给定一个链表，判断其中是否存在环。(力扣 141. 环形链表)
- 即，是否存在某个节点，使得从该节点开始的后继节点会回到该节点本身。

检测链表中的环

- **方法:** 使用快慢指针，快指针每次移动两步，慢指针每次移动一步。
- **步骤:**
  1. 初始化快指针和慢指针均指向链表头部。
  2. 快指针每次移动两步，慢指针每次移动一步。
  3. 若快指针追上慢指针，说明链表中存在环。
  4. 若快指针到达链表末尾，说明链表中不存在环。

检测链表中的环-找到环的入口

- **问题:** 如何找到环的入口？(力扣 142. 环形链表 2)
- **方法:** 在两个指针相遇后，将其中一个指针指向链表头部，两个指针同时移动一步，再次相遇的节点即为环的入口。

检测链表中的环-找到环的入口

- 设:
  - 链表头部到环入口的距离为 a。
  - 慢指针进入环后走 b 步与快指针相遇。
  - 环的长度位 L。
- 在他们第一次相遇时:
  - 慢指针走过的距离为 $a + b$。
  - 因为速度时二倍, 所以快指针走过的距离为 $2 * (a + b)$
- 快指针比慢指针多走了 n 圈:

$$2 * (a + b) = a + b + nL$$

$$a = nL - b$$

- 即, 慢指针再走 $nL - b$ 步就会走环的整数倍, 即走到环的入口。
- 那么令一个指针指向头部, 另一个指针指向相遇点, 两个指针同时移动一步, 再次相遇的节点即为环的入口。

检测链表中的环-找到环的长度

- **方法:** 在两个指针相遇后，将其中一个指针保持不动，另一个指针继续移动，每次一步，直到再次回到相遇点，移动的步数即为环的长度。

两数之和问题

- **方法:** 使用对撞指针在有序数组中查找目标和。
- **步骤:**
  1. 初始化左指针指向数组开始，右指针指向数组末尾。
  2. 计算当前左右指针指向元素的和。
  3. 若和等于目标值，返回结果。
  4. 若和小于目标值，左指针右移。
  5. 若和大于目标值，右指针左移。
  6. 重复上述步骤，直到找到目标或指针交错。
- **时间复杂度:** $O(n)$，相比于暴力解法的 $O(n^2)$ 更加高效。
- **扩展:** 三数之和问题。则固定一个数，然后使用对撞指针查找另外两个数。22 级题目

最长无重复子串

- **方法:** 利用滑动窗口维护当前子串，无重复字符。
- **步骤:**
  1. 初始化左指针和右指针均指向字符串开始。
  2. 移动右指针，扩展窗口，直到遇到重复字符。
  3. 若遇到重复字符，移动左指针，缩小窗口，直到重复字符被移除。
  4. 更新最长子串长度。
  5. 重复上述步骤，直到右指针遍历完整个字符串。
- **时间复杂度:** $O(n)$，每个字符最多被访问两次。

排序与查找

排序与查找

常见排序算法

- 冒泡排序 - $O(n^2)$
  - 比较相邻元素并交换顺序
  - 简单但效率低下，适用于小规模数据
- 快速排序 - $O(nlogn)$
  - 分治法，通过基准元素划分子数组
  - 平均情况下性能优越，但最坏情况为 $O(n^2)$
- 归并排序 - $O(nlogn)$
  - 分治法，将数组分成两半，分别排序后合并
  - 稳定排序，适用于链表
- 堆排序 - $O(nlogn)$
  - 利用堆数据结构实现
  - 原地排序，无需额外空间

排序与查找

二分查找

- 基本二分查找
  - 在有序数组中查找目标元素
  - 每次比较中间元素，缩小搜索范围
- 二分答案
  - 在搜索最优解时使用，如最小最大值
- 左侧边界二分
  - 查找第一个满足条件的元素
- 右侧边界二分
  - 查找最后一个满足条件的元素

二分模板

模板 1

```
int l,r,mid;

while(l<r){
    mid = (l+r)/2;
    if(check(mid)) r = mid;
    else l = mid + 1;
}
return l;
```

二分模板

模板 2

```
int l,r,mid;

while(l<r){
    mid = (l+r+1)/2;
    if(check(mid)) l=mid;
    else r=mid-1;
}
return l;
```

要点

- 模板 1,2 的本质区别在于 l 的更新方式, 模板 1 是 l=mid+1, 模板 2 是 l=mid
- 这是为了应对这样的一种情况, 只有两个元素的时候, 如果取 mid=(l+r)/2, 那么 mid 永远等于 l, 可能会造成死循环
- 二分不是针对单调性, 而是针对能够将问题划分为两个部分, 然后通过判断中间值来缩小搜索范围的问题, 并不一定是狭义上单调的

排序与查找的应用案例

- 合并两个有序数组
  - 使用双指针技巧，逐一比较并合并
  - 也就是归并排序的 merge 操作
  - 时间复杂度为 O(n)
- 二分答案
  - 最大值最小或者最小值最大

树

树

二叉树

- 完全二叉树
  - 除最后一层外，每层节点数达到最大
  - 最后一层节点集中在左侧
- 满二叉树
  - 每个节点都有 0 或 2 个子节点
  - 所有叶子节点深度相同
- 二叉搜索树 (BST)
  - 左子树所有节点值小于根节点值
  - 右子树所有节点值大于根节点值

树

二叉树 (续)

- AVL 树
  - 自平衡二叉搜索树
  - 每个节点的左右子树高度差不超过 1

树的遍历

- 前序遍历
  - 访问根节点，遍历左子树，遍历右子树
- 中序遍历
  - 遍历左子树，访问根节点，遍历右子树
  - 对于 BST，中序遍历得到有序序列

树

树的遍历 (续)

- 后序遍历
  - 遍历左子树，遍历右子树，访问根节点
  - 常用于释放内存
- 层序遍历
  - 按层级顺序逐层访问节点
  - 通常使用队列实现

AVL 树（旋转）

AVL 树：为什么要旋转?

- AVL：自平衡的二叉搜索树（BST）。
- 维护每个节点的 高度 h，并定义 平衡因子

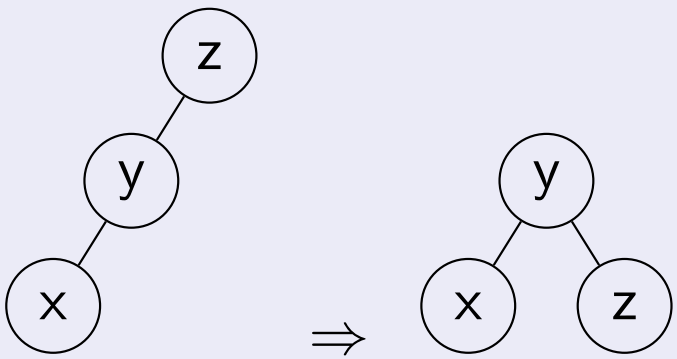
$$bf(u) = h(\text{left}(u)) - h(\text{right}(u))$$

- AVL 的约束：对任意节点 u，都有 $|bf(u)| \leq 1$。
- 插入/删除后，某条祖先链上可能出现 $|bf| = 2$ 的节点，需要通过 旋转 恢复平衡。
- 关键点：旋转只改变局部形状，不改变中序序列（BST 性质仍成立）。

单旋：LL 与 RR

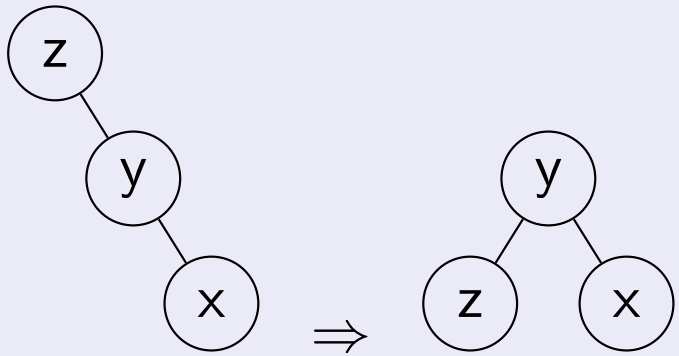
LL：右旋（Right Rotation）

- 失衡点 z：$bf(z) = +2$，且插入发生在 z 的 左子树的左侧。
- 操作：对 z 做 右旋。



RR：左旋（Left Rotation）

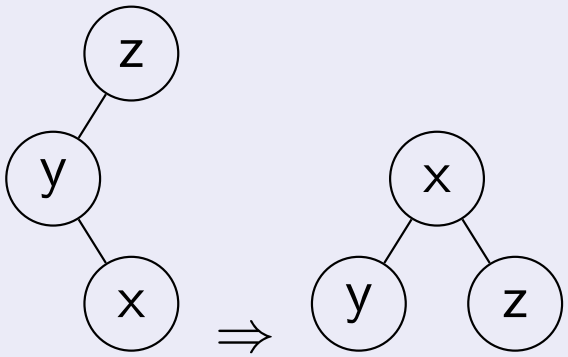
- 失衡点 z：$bf(z) = -2$，且插入发生在 z 的 右子树的右侧。
- 操作：对 z 做 左旋。



双旋：LR 与 RL

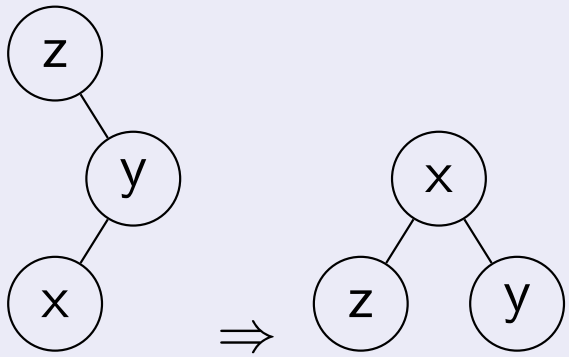
LR：先左旋子树，再右旋根

- $bf(z) = +2$，插入发生在 z 的左子树的右侧。
- 操作：LeftRotate(y) 再 RightRotate(z)。



RL：先右旋子树，再左旋根

- $bf(z) = -2$，插入发生在 z 的右子树的左侧。
- 操作：RightRotate(y) 再 LeftRotate(z)。



记忆方法：看“路径”就能秒判旋转

三步口诀

1. 找到 第一个失衡点 z（从插入点往上，遇到 $|bf(z)| = 2$ 的第一个）。
2. 看插入点（或更高的“重子树”）相对 z 的 两步方向路径：L/R + L/R。
3. 同向（LL / RR） $\Rightarrow$ 单旋；异向（LR / RL） $\Rightarrow$ 双旋。

一句话记住“转哪边”

LL 向右转，RR 向左转；LR = 左旋子树再右旋根，RL = 右旋子树再左旋根。

四种情况速查表

按“路径”分类

路径	失衡类型	旋转操作
LL	左左	RightRotate(z)
RR	右右	LeftRotate(z)
LR	左右	LeftRotate(y) → RightRo
RL	右左	RightRotate(y) → LeftRo

“同向/异向”

- LL、RR：同向（L+L / R+R）
- LR、RL：异向（L+R / R+L）
- 同向做 一次，异向做 两次

插入后如何恢复平衡（思路）

- 像 BST 一样插入新节点（叶子位置）。
- 从插入点一路 回溯 到根：
  - 先更新当前节点高度（pull）。
  - 计算 bf，若 $|bf| \le 1$ 继续回溯。
  - 若出现 $bf = +2$ 或 $bf = -2$，按 4 种情况旋转一次（或双旋），该子树恢复平衡。
- AVL 树的高度为 $O(\log n)$，所以插入/删除/查找都是 $O(\log n)$。

图论基础

图论基础概念

基本定义

- 图 $G = (V, E)$：点集 V、边集 E。
- 无向/有向、带权/无权。
- 度：无向图是 deg(v)；有向图分 入度/出度。

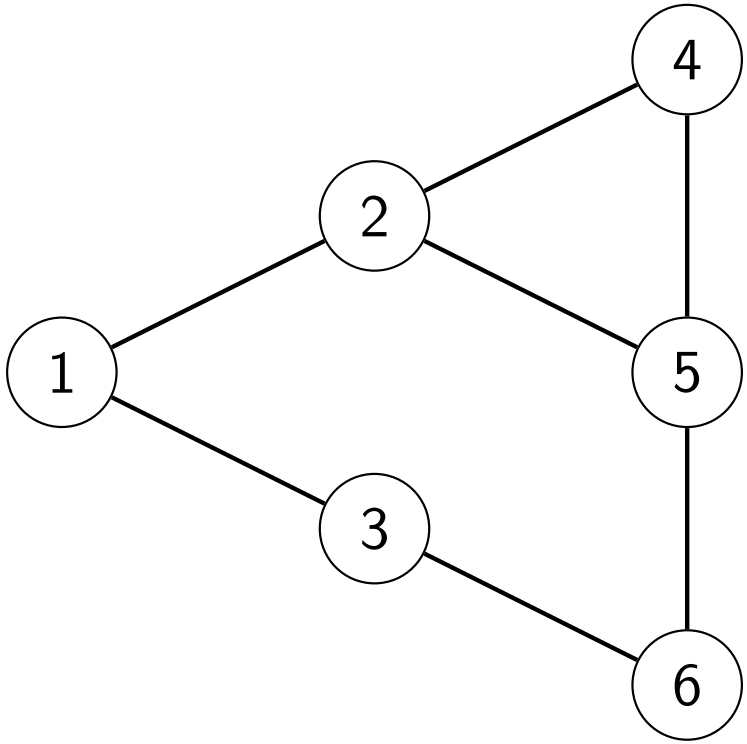
路径与连通

- 路径、简单路径、回路(环)。
- 连通块：无向图的“分组”。
- DAG：有向无环图（拓扑排序的对象）。

图论题目

图论很多题 = 建图 + 遍历/最短路/生成树/强连通。

示例图



约定（避免“遍历顺序不一样”争议）

从 1 号点开始；邻接表按编号从小到大访问。

图论基础

图的表示

- 邻接矩阵
  - 使用二维数组表示图的边
  - 适合稠密图，空间复杂度 $O(n^2)$
- 邻接表
  - 使用链表或数组表示每个节点的邻居
  - 适合稀疏图，空间复杂度 $O(n + m)$
- 前向星
  - 高效的图边存储方式，适用于竞赛编程

邻接矩阵实现

```cpp
// 邻接矩阵:二维数组g[N][N]存储
const int N = 1010; // 最大节点数
int g[N][N];  // 邻接矩阵
int n, m;     // n个节点,m条边

void init() {
    // 读入n个节点,m条边
    cin >> n >> m;
    // 初始化邻接矩阵
    memset(g, 0, sizeof g);
    // 读入m条边
    for(int i = 0; i < m; i++) {
        int a, b, w;
        cin >> a >> b >> w; // 起点a,终点b,权值w
        g[a][b] = w;  // 存储边权
        //g[b][a] = w; // 无向图时需要
    }
}
```

邻接表实现

```cpp
int n, m;       // n个节点,m条边

vector<vector<int>> g(n);
void init() {
    // 读入n个节点,m条边
    cin >> n >> m;
    // 初始化邻接表
    g.resize(n);
    // 读入m条边
    for(int i = 0; i < m; i++) {
        int a, b, w;
        cin >> a >> b >> w; // 起点a,终点b,权值w
        g[a].push_back(b);  // 存储边权
        //g[b].push_back(a); // 无向图时需要
    }
    // 遍历节点a的所有邻接边
    for (int i=0; i<g[a].size(); i++) {....}
}
```

链式前向星

```
const int N = 1010,M=2*N; // 最大节点数,最大边数=2*最大节点数
int h[N],e[M],ne[M],w[M],idx; // 邻接表存储
memset(h, -1, sizeof h); // 邻接表初始化
int n, m;      // n个节点,m条边
void add(int a, int b, int c) {
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}
// 遍历一个节点a的所有邻接边
for (int i = h[a]; i != -1; i = ne[i]) {
    int b = e[i], c = w[i];
    // 遍历a的所有邻接边
}
```

图论基础

图的算法

- Tarjan 算法 (强连通分量)
  - 用于分割图中的强连通子图
  - 基于 dfn 和 low 数组实现
- 最短路算法
  - Dijkstra: 适用于非负权重图
  - Floyd: 适用于所有节点对之间的最短路径
- 最小生成树
  - Prim: 逐步扩展生成树, 适合稠密图
  - Kruskal: 按边权排序, 适合稀疏图（通常配合并查集判环/连通）

DFS：一条路走到黑

核心思想

- 从起点出发，优先深入到不能再深入，再回溯。
- 实现方式：递归 或 显式栈。

能解决什么

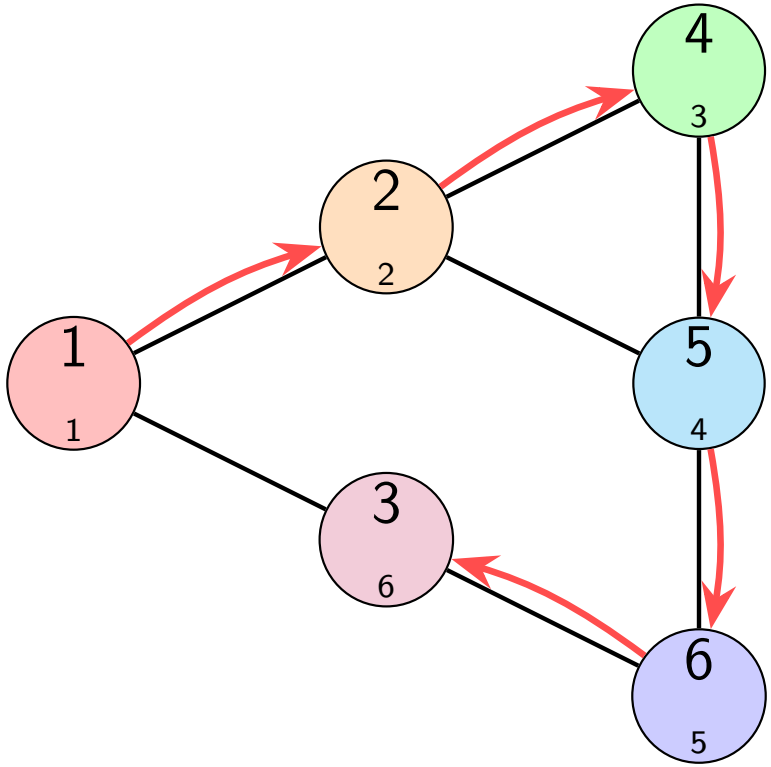
- 连通性/连通块、无向图判环、拓扑排序（DAG）、树的遍历、Flood Fill。
- 时间复杂度：邻接表存图时 $\mathcal{O}(n+m)$。

DFS 模板（邻接表）

```cpp
int n, m;
vector<vector<int>> g;
vector<int> vis;

void dfs(int u){
    vis[u] = 1;
    for(int v: g[u]){
        if(!vis[v]) dfs(v);
    }
}
// 如果图不一定连通:
for(int i=1;i<=n;i++) if(!vis[i]) dfs(i);
```

DFS 遍历示意（从 1 开始）



访问序

一种可能的 DFS 序：$1 \to 2 \to 4 \to 5 \to 6 \to 3$（取决于邻接表顺序）。

BFS：按层扩展

核心思想

- 从起点出发，一圈一圈向外扩展。
- 实现方式：队列。

最重要的性质

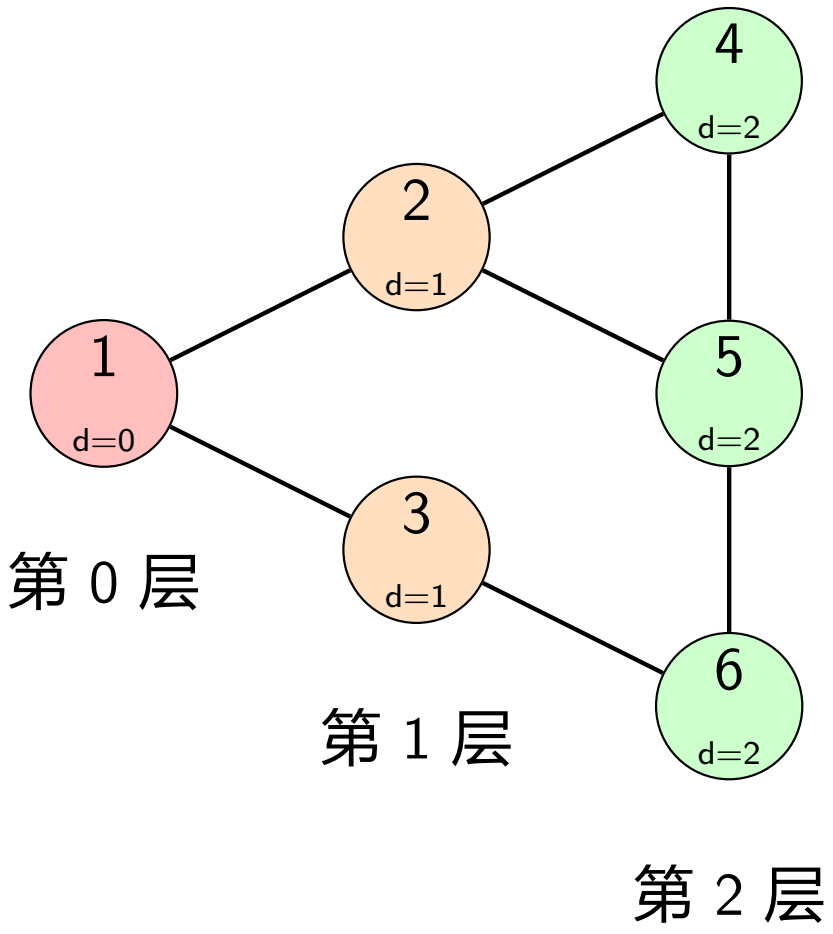
- 在无权图/边权相同时，BFS 求得的就是最短路 (边数最少)。
- 时间复杂度：邻接表存图时 $\mathcal{O}(n+m)$。

BFS 模板（无权最短路）

```
vector<int> dist(n + 1, -1);
queue<int> q;
dist[s] = 0;
q.push(s);

while(!q.empty()){
    int u = q.front(); q.pop();
    for(int v: g[u]){
        if(dist[v] != -1) continue;
        dist[v] = dist[u] + 1;
        q.push(v);
    }
}
```

BFS 分层示意（从 1 开始）



访问顺序（队列）

常见输出序：$1, 2, 3, 4, 5, 6$（同样依赖邻接表顺序）。

图论算法怎么选?

- 连通性/连通块：DFS/BFS。
- 无权最短路：BFS。
- 非负权最短路：Dijkstra（优先队列）。
- 任意两点最短路：Floyd（$O(n^3)$，点数小才用）。
- 最小生成树：Prim / Kruskal（Kruskal 常配并查集）。
- 强连通分量：Tarjan。
- DAG 拓扑序：入度法 (Kahn) 或 DFS。

并查集 (DSU)

并查集 (Union-Find / DSU)

它解决什么问题?

- 维护一堆元素的若干个不相交集合（连通块/帮派/组件）。
- 支持两类操作：
  - Find(x)：查询 x 属于哪个集合（集合代表元）。
  - Union(a,b)：把 a 所在集合与 b 所在集合合并。
- 常见场景：动态连通性、无向图判环、Kruskal 最小生成树、统计连通块个数。

并查集：两大优化（推荐）

- 路径压缩 (Path Compression)：在 Find 时把访问路径上的点直接挂到根上。
- 按大小合并 (Union by Size)：总是把“小树”挂到“大树”上，让树更矮。
- 复杂度直觉：
  - 只用按大小/按秩合并：树高 $\leq \log n$，单次 Find 为 $\mathcal{O}(\log n)$。
  - 只用路径压缩：实际通常很快，但理论上不如“两者一起”的保证好。
  - 两者一起用：均摊接近常数，严格地说是 $\mathcal{O}(\alpha(n))$。

按秩合并（可替代按大小合并）

- rank 表示树的“高度上界”（不一定等于真实高度）。
- 合并时把低秩树挂到高秩树；若相等，任选一个做根并把它的 rank+1。

```
vector<int> p, rk;

int find(int x){
    if(p[x]!=x) p[x]=find(p[x]); // 路径压缩
    return p[x];
}
bool unite(int a,int b){ // 按秩合并
    a=find(a); b=find(b);
    if(a==b) return false;
    if(rk[a]<rk[b]) swap(a,b);
    p[b]=a;
    if(rk[a]==rk[b]) rk[a]++;
    return true;
}
```

并查集 C++ 模板（推荐背熟）

```cpp
vector<int> p, sz;
void init(int n) {
    p.resize(n + 1);
    sz.assign(n + 1, 1);
    for (int i = 0; i <= n; i++) p[i] = i;
}
int find(int x) {
    if(p[x]!=x) p[x]=find(p[x]); // 路径压缩
    return p[x];
}
bool unite(int a, int b) { // 按大小合并
    a = find(a), b = find(b);
    if (a == b) return false;
    if (sz[a] < sz[b]) swap(a, b); // 小挂大
    p[b] = a;
    sz[a] += sz[b];
    return true;
}
```

并查集的典型用法

1) 无向图判环

- 逐条处理边 (u, v)：
- 若 same(u,v) 为真，说明 u 和 v 已经连通，再连一条边就会形成环。
- 否则 unite(u,v)。

2) Kruskal 最小生成树

- 把边按权值从小到大排序，依次尝试加入。
- 若加入后会成环（same 为真）就跳过，否则加入并 unite。

例题：动态连通性（并查集经典题）

题目

给定 n 个点（编号 $1 \sim n$），有 m 次操作：

- 1 a b：把 a 和 b 连起来（合并集合）。
- 2 a b：询问 a 和 b 是否连通，是输出 Yes 否输出 No。

样例

输入： $n = 5$，操作：1 1 2, 2 1 3, 1 2 3, 2 1 3
输出： No, Yes

例题思路

- 初始化：每个点自成一个集合（父亲指向自己）。
- 遇到 1 a b：执行 unite(a,b)。
- 遇到 2 a b：判断 same(a,b)，输出 Yes/No。
- 核心点：find 一定要写路径压缩，否则大数据会慢。

例题参考实现（C++）

```cpp
#include <bits/stdc++.h>
using namespace std;
vector<int> p, sz;

void init(int n) {
    p.resize(n + 1);
    sz.assign(n + 1, 1);
    for (int i = 0; i <= n; i++) p[i] = i;
}
int find(int x) {
    if (p[x] == x) return x;
    return p[x] = find(p[x]);
}
```

例题参考实现（C++）

```cpp
void unite(int a, int b) {
    a = find(a), b = find(b);
    if (a == b) return;
    if (sz[a] < sz[b]) swap(a, b);
    p[b] = a;
    sz[a] += sz[b];
}
bool same(int a, int b) { return find(a) == find(b); }
int main() {
    ios::sync_with_stdio(false);cin.tie(nullptr);
    int n, m;cin >> n >> m;
    init(n);
    while (m--) {
        int op, a, b;cin >> op >> a >> b;
        if (op == 1) unite(a, b);
        else cout << (same(a, b) ? "Yes\n" : "No\n");
    }
    return 0;
```

哈希表

Hash

Hash 表

- 拉链法
  - 将哈希表中的每个桶看成一个链表
  - 适用于处理哈希冲突
  - 插入一个数的代码操作和图论建边很像，相当于添加一条边 (计算出的哈希值到这个数原本的值的边)
- 开放寻址法: 建议至少会手动模拟

Hash 表：核心概念

- 目标：把 Key 映射到 $[0, m-1]$ 的桶下标（bucket index），支持快速查找。
- 哈希函数 $h(\text{key})$：尽量 均匀、快速。
- 负载因子 $\alpha = \frac{n}{m}$：
  - n 为元素数，m 为桶数（表长）。
  - α 越大，冲突越多，操作越慢。
- 期望复杂度：在哈希均匀且 α 控制得当时，插入/查询/删除都是期望 $O(1)$。

拉链法：怎么做、复杂度怎么算

- 结构：表是一个数组，每个位置挂一个链表/动态数组，存放哈希到该桶的所有 Key。
- 查询：先算桶下标 $idx = h(key)$，再在桶内线性查找。
- 期望复杂度：桶内期望长度约为 α，所以查询期望 $O(1+\alpha)$。
- 适用：实现简单；冲突多时退化，但只退化到桶内。

拉链法：手写实现

```cpp
// 简化版：只存 int key（可扩展到 pair<key,val>）
struct HashSetChaining {
    int m; // 桶数 vector<vector<int>> b; HashSetChaining
        (int m_=100003): m(m_), b(m_) {}
    int h(int x) const {
        long long y = x;
        y %= m; if (y < 0) y += m;
        return (int)y;
    }
    bool contains(int x) const {
        int idx = h(x);
        for (int v: b[idx]) if (v == x) return true;
        return false;
    }
    void insert(int x) {
        int idx = h(x);
        if (!contains(x)) b[idx].push_back(x);
    }
```

开放寻址法：探测（建议会手动模拟）

- 思路：所有元素直接放在表数组里；冲突时沿 探测序列 找下一个空位置。
- 常见探测：
  - 线性探测：$idx, idx+1, idx+2, \ldots$
  - 二次探测：$idx+1^2, idx+2^2, \ldots$
  - 双重哈希：$idx + k \cdot h_2(key)$
- 删除：不能直接清空，需要 墓碑标记 (tombstone)，否则会断开后续查找链。
- 经验规则：一般要求 α 不要太大（如 $\alpha \leq 0.7$），否则探测长度变长。

rehash：为什么要扩容

- 当 α 过大，冲突/探测明显变多，期望 $O(1)$ 会变差。
- rehash：把桶数 m 变大（常见翻倍或换更大的素数），并把所有元素按新 m 重新分配。
- 摊还思想：扩容很贵，但不频繁；均摊到每次插入，整体仍可认为期望 $O(1)$。

算法设计

贪心与动态规划

贪心与动态规划

贪心算法

- 贪心策略
  - 每一步都选择当前最优解
  - 不回溯，期望全局最优
- 证明正确性
  - 通过贪心选择性质和最优子结构
- 经典问题
  - Huffman 编码
  - 区间调度
- 微扰法
  - 研究既有策略下，相邻两个元素交换是否更好

动态规划

- 状态定义
  - 明确定义子问题的状态
- 状态分割
  - 将问题分解为多个子问题
  - 一个状态分为多个子状态
- 状态转移
  - 描述如何从子问题的解得到当前问题的解
- 边界条件
  - 明确初始条件，确保递推的开始
- 空间优化
  - 使用滚动数组或其他技巧减少空间复杂度

贪心算法的详细讲解

- 贪心策略的选择
  - 选择当前最优，期望整体最优
  - 需要具备贪心选择性质
- 贪心算法的应用场景
  - 问题可以分解为局部最优子问题
  - 无需回溯，选择一次后不改变
- 与动态规划的区别
  - 贪心算法仅关注局部最优，动态规划关注全局最优
  - 贪心算法更高效，但适用范围有限

动态规划的详细讲解

- 分阶段解决问题
  - 将问题划分为多个阶段，每个阶段解决一个子问题
- 记忆化与递推
  - 记忆化：递归解决并存储中间结果
  - 递推：迭代解决，从小问题开始逐步构建
- 优化技巧
  - 状态压缩
  - 边界条件优化
  - 使用滚动数组减少空间

贪心算法与动态规划的比较

- 选择策略
  - 贪心: 每一步选择最优
  - 动态规划: 考虑所有可能选择
- 适用条件
  - 贪心: 需要问题具备贪心选择性质和最优子结构
  - 动态规划: 需要问题具备最优子结构
- 时间与空间复杂度
  - 贪心: 通常 $O(n)$ 或 $O(n \log n)$
  - 动态规划: 通常 $O(n^2)$ 或更高
- 实例对比
  - 区间调度: 贪心最优
  - 背包问题: 动态规划更适用

贪心与动态规划的应用案例

- 贪心算法案例: 活动选择问题
  - 选择最多不重叠的活动
  - 按结束时间排序, 优先选择最早结束的活动
- 动态规划案例: 最长公共子序列 (LCS)
  - 寻找两个序列中最长的公共子序列
  - 定义状态和转移方程, 递推计算
- 贪心算法案例:Huffman 编码
  - 构建最优二叉树
  - 频率最低的字符优先合并
- 动态规划案例: 背包问题
  - 在给定重量限制下, 选择物品使价值最大
  - 状态定义为当前重量, 选择是否放入当前物品

贪心与动态规划总结

- 贪心算法
  - 优点: 实现简单, 效率高
  - 缺点: 仅适用于具备贪心选择性质的问题
- 动态规划
  - 优点: 适用范围广, 能解决复杂问题
  - 缺点: 实现复杂, 空间与时间消耗较大
- 选择算法的依据
  - 分析问题是否具备贪心选择性质
  - 考虑算法的时间与空间需求
  - 根据具体需求和约束选择合适的算法

题目概览

经典题目分析

题目类型

- 双指针求三数目标和
- 双指针接雨水
- 动态中位数 (对顶堆)
- 根据遍历顺序构造二叉树 (单个序列, 前序中序, 后序中序)
- 最短路 dijkstra,floyd
- 最长上升子序列 (nlogn)
- 本质不同子序列个数 (动态规划)
- 区间选点问题 (贪心)

双指针：三数之和

双指针求三数目标和

题目链接：三数之和 (LeetCode 15)

题目描述

三数之和

给定一个包含 n 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 a，b，c，使得 $a+b+c=0$？请你找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

- 输入：`nums = [-1,0,1,2,-1,-4]`
- 输出：`[[-1, -1, 2], [-1, 0, 1]]`

题目分析

分析:

- 直接使用三重循环会导致时间复杂度过高，为 $O(n^3)$。
- 可以先对数组进行排序，然后固定一个数，使用双指针在剩余部分寻找另外两个数，使得三数之和为零。
- 需要注意跳过重复的元素，以避免结果中出现重复的三元组。

解题思路

做法:

1. 对数组 `nums` 进行排序。
2. 遍历数组，固定第一个数 `nums[i]`，然后使用双指针 `left` 和 `right` 在 `i+1` 到 `n-1` 的范围内寻找另外两个数。
3. 计算三数之和，如果等于零，则加入结果集，并移动指针跳过重复的数。
4. 如果三数之和小于零，移动左指针；如果大于零，移动右指针。
5. 重复以上步骤，直到所有可能的三元组都被检查。

双指针：接雨水

双指针接雨水

- 接雨水 - 力扣（LeetCode）

题目描述

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

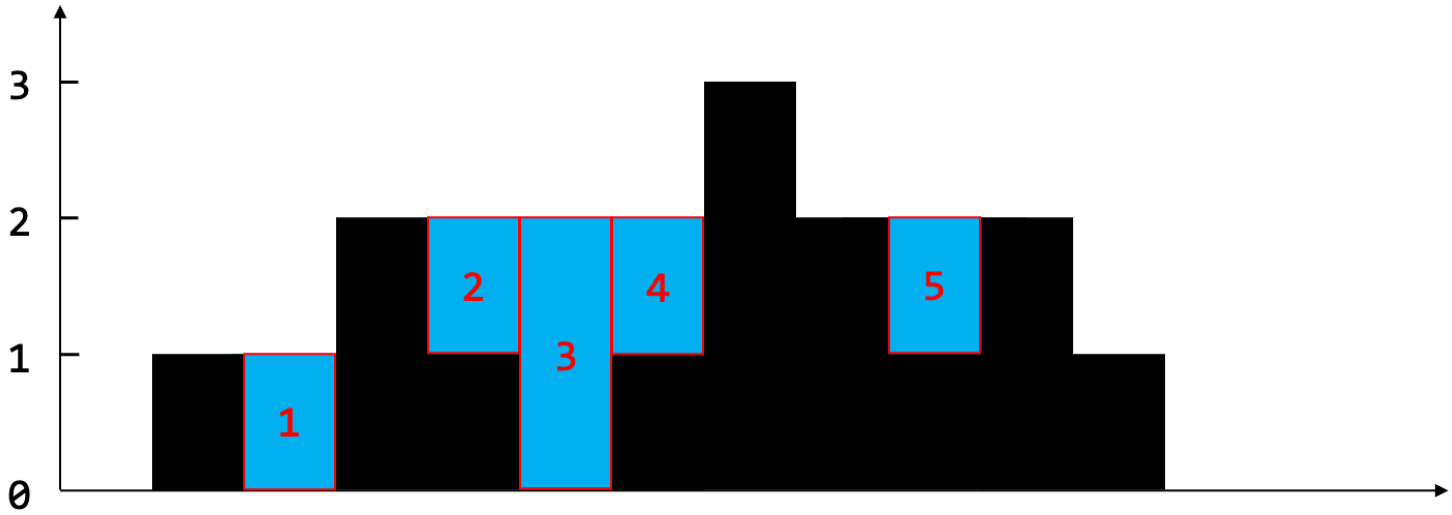
问题分析

- **直观理解**：雨水的积聚取决于每个位置左边和右边的最大高度。
- **双指针法**：
  - 初始化两个指针，left 指向数组的起始位置，right 指向数组的末尾位置。
  - 初始化两个变量，left_max 和 right_max，分别记录左边和右边的最大高度。

问题分析（续）

- 当 left 小于等于 right 时，执行以下步骤：
  - 如果 height[left] 小于 height[right]，则：
    - 如果 height[left] 大于 left_max，则更新 left_max；
    - 否则，计算当前积水量并累加到总积水量中；
    - 将 left 指针右移一位。
  - 否则：
    - 如果 height[right] 大于 right_max，则更新 right_max；
    - 否则，计算当前积水量并累加到总积水量中；
    - 将 right 指针左移一位。

接雨水示意图




```
height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
   res = [0, 0, 1, 0, 1, 2, 1, 0, 0, 1, 0, 0]
```

图: 接雨水示意图

为什么双指针有效：通过比较左右两边的高度，可以确定雨水积累的限界，从而避免使用额外的空间。

做法总结

- 使用双指针，从两端向中间移动。
- 维护左右两边的最大高度，并根据较小的一边决定移动指针。
- 时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

对顶堆：动态中位数

动态中位数 (对顶堆)

题目链接: 寻找数据流的中位数 (LeetCode 295)

题目描述

寻找数据流的中位数

中位数是有序序列中间的数。如果序列长度为偶数，中位数是中间两个数的平均值。

实现一个数据结构 `MedianFinder`，支持以下操作：

- `void addNum(int num)`：从数据流中添加一个整数到数据结构中。
- `double findMedian()`：返回当前所有元素的中位数。

题目分析

分析:

- 需要高效地找到数据流中的中位数。
- 使用两个堆 (优先队列) 可以在 $O(\log n)$ 时间内插入元素，并在 $O(1)$ 时间内找到中位数。
- 一个最大堆存储较小的一半数，一个最小堆存储较大的一半数。
- 通过维护两个堆的大小平衡，可以快速计算中位数。

解题思路

做法:

1. 使用两个优先队列 (堆):
   - `maxHeap`: 存储较小的一半数, 堆顶是这部分的最大值。
   - `minHeap`: 存储较大的一半数, 堆顶是这部分的最小值。
2. 添加数字时:
   - 先将数字添加到 `maxHeap`。
   - 将 `maxHeap` 的堆顶元素移动到 `minHeap`。
   - 如果 `minHeap` 的大小大于 `maxHeap`, 将 `minHeap` 的堆顶元素移动回 `maxHeap`。
3. 查找中位数时:
   - 如果两个堆的大小相同, 中位数是两个堆顶元素的平均值。
   - 如果不相同, 中位数是 `maxHeap` 的堆顶元素。

二叉树：遍历序构造

根据遍历顺序构造二叉树

- 根据前序和中序遍历构造二叉树 - 力扣 (LeetCode)
- 根据后序和中序遍历构造二叉树 - 力扣 (LeetCode)

题目描述

根据给定的遍历顺序构造二叉树:

- 根据前序遍历和中序遍历构造二叉树。
- 根据后序遍历和中序遍历构造二叉树。

假设树中没有重复的元素。

问题分析

1. **前序和中序遍历构造二叉树**:
   - 前序遍历的第一个元素是根节点。
   - 在中序遍历中找到根节点的位置，分割出左子树和右子树。
   - 递归构造左右子树。
2. **后序和中序遍历构造二叉树**:
   - 后序遍历的最后一个元素是根节点。
   - 在中序遍历中找到根节点的位置，分割出左子树和右子树。
   - 递归构造左右子树。
3. **优化**:
   - 使用哈希表存储中序遍历中元素的位置，以减少查找时间。
   - 时间复杂度为 $O(n)$。

做法总结

- 使用递归的方法，根据前序/后序和中序遍历信息构造二叉树。
- 使用哈希表优化中序遍历中根节点的查找。
- 时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

最短路：Dijkstra / Floyd

最短路算法

- Dijkstra 算法
- Floyd 算法

Dijkstra 算法

- **适用范围**: 适用于带权图且所有边权重非负的情况。
- **基本思想**: 通过贪心选择当前最短路径的节点，逐步扩展到所有节点。
- **实现方法**:
  - 使用优先队列 (最小堆) 来选择当前距离最短的节点。
  - 初始化起点的距离为 0，其余节点为无穷大。
  - 每次从优先队列中取出距离最短的节点，更新其邻居节点的距离。
  - 重复直到所有节点都被处理。
- **时间复杂度**: $O((V+E)\log V)$，其中 V 是节点数，E 是边数。

Floyd 算法

- **适用范围**：适用于所有顶点对之间的最短路径，适用于有向图和无向图，支持负权边但不支持负权环。
- **基本思想**：动态规划，通过逐步考虑每个顶点作为中间点，更新最短路径。
- **实现方法**：
  - 初始化一个二维数组 dist，其中 dist[i][j] 表示从节点 i 到节点 j 的最短距离。
  - 对于每个中间节点 k，更新 dist[i][j] 为 min(dist[i][j], dist[i][k] + dist[k][j])。
- **时间复杂度**：$O(V^3)$。

做法总结

- **Dijkstra**:
  - 适用于单源最短路径问题。
  - 使用优先队列优化选择最短路径的节点。
- **Floyd**:
  - 适用于所有顶点对之间的最短路径。
  - 通过动态规划逐步考虑中间节点。

动态规划：最长上升子序列

最长上升子序列 (LIS)

- 最长上升子序列 - 力扣 (LeetCode)

题目描述

给定一个无序的整数数组，找到其中最长严格递增子序列的长度。

问题分析

- **动态规划方法**:
  - 定义 dp[i] 为以 nums[i] 结尾的最长上升子序列的长度。
  - 状态转移方程:dp[i] = max(dp[j] + 1), 其中 $0 \leq j < i$ 且 nums[j] < nums[i]。
  - 最终答案为 max(dp[i])。
- **二分查找优化**:
  - 维护一个数组 tails, 其中 tails[i] 表示长度为 i+1 的子序列的最小结尾元素。
  - 遍历数组, 对于每个元素 num, 使用二分查找在 tails 中找到第一个大于或等于 num 的位置 idx, 并更新 tails[idx] = num。
  - 如果 num 大于所有 tails 中的元素, 追加到 tails 末尾。
  - 最终 tails 的长度即为 LIS 的长度。

做法总结

- 动态规划方法简单直观，但时间复杂度较高。
- 使用二分查找可以将时间复杂度优化到 $O(n \log n)$。
- 适用于处理大规模数据的情况。

动态规划：本质不同子序列

本质不同子序列个数

- **问题描述**：
  - 给定一个字符串 ‘s‘，计算其所有不同的非空子序列的个数。由于结果可能非常大，返回结果需要对 $10^9 + 7$ 取余。

本质不同子序列个数 - 问题描述

给定一个字符串 ‘s‘，返回 ‘s‘ 的所有不同的非空子序列的个数。由于结果可能非常大，返回结果需要对 $10^9 + 7$ 取余。

示例:

“‘ 输入: ”abc” 输出: 7 解释: 7 个不同的非空子序列为”a”, ”b”, ”c”, ”ab”, ”ac”, ”bc”, ”abc” “‘

“‘ 输入: ”aba” 输出: 6 解释: 6 个不同的非空子序列为”a”, ”b”, ”aa”, ”ab”, ”ba”, ”aba” “‘

本质不同子序列个数 - 问题分析

- **动态规划方法**：
  - 定义 ‘dp[i]‘ 为字符串 ‘s‘ 的前 ‘i‘ 个字符所能组成的不同非空子序列的个数。
  - **状态转移方程**：
    - 对于第 ‘i‘ 个字符 ‘s[i-1]‘，每个已有的子序列都可以选择是否包含这个字符，因此总数会翻倍，即 ‘dp[i] = 2 * dp[i-1]‘。
    - 如果字符 ‘s[i-1]‘ 在之前已经出现过，那么上述方法会导致重复计算。因此，我们需要减去在之前最后一次出现 ‘s[i-1]‘ 时所增加的子序列数。
    - 设字符 ‘s[i-1]‘ 最后一次出现的位置为 ‘last[s[i-1]]‘，则

$$dp[i] = \begin{cases} 2 \times dp[i-1] & \text{如果} s[i-1] \text{ 没有出现过} \\ 2 \times dp[i-1] - dp[last[s[i-1]]-1] & \text{如果} s[i-1] \text{ 已经出现过} \end{cases}$$

本质不同子序列个数 - 问题分析（续）

- **初始化**：
  - ‘dp[0] = 0‘，表示空字符串没有非空子序列。
  - ‘dp[1] = 1‘，表示只有一个字符时，只有一个非空子序列。
- **处理重复**：
  - 使用一个哈希表 ‘last‘ 来记录每个字符最后一次出现的位置，以便在状态转移时进行调整。
- **最终结果**：
  - ‘dp[n]‘ 即为字符串 ‘s‘ 的所有不同的非空子序列的个数。

本质不同子序列个数 - 做法总结

- 使用动态规划定义子问题，逐步构建解决方案。
- 利用状态转移方程高效计算结果。
- 通过哈希表记录每个字符最后一次出现的位置，避免重复计算。
- 时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

贪心：区间选点

区间选点问题

- 最少箭矢数目 - 力扣（LeetCode）

题目描述

给定一些以不同弦线表示的气球，每个气球的弦线在水平轴上的一个区间 $points[i] = （x_{start}, x_{end}）$，找到最少数量的箭能刺爆所有气球。箭一旦被射出，可以无限地刺向某一位置的所有气球。

问题分析

- **直观理解**：要找到最少的点，使得每个区间至少包含一个点。
- **贪心算法**：
  - **排序**：按照区间的结束位置 x_end 从小到大排序。
  - **初始化**：箭的数量初始化为 1，选择第一个区间的结束位置作为箭的位置。

问题分析（续）

- **贪心算法（续）**：
  - **遍历区间**：
    - 遍历后续区间，如果当前区间的起始位置 x_start 大于箭的位置，说明需要新的箭，更新箭的位置为当前区间的结束位置，并增加箭的数量。
    - 否则，当前区间可以被已有的箭覆盖，继续。
- **为什么贪心有效**：每次选择最早结束的区间作为箭的位置，可以最大化覆盖后续的区间，减少箭的数量。

做法总结

- 按照区间结束位置排序。
- 使用贪心策略选择最早结束的位置作为箭的位置。
- 时间复杂度为 $O(n \log n)$，主要由排序决定。

刷题路线

复习方法

- **刷题**:
  - 推荐刷 PTA 作业题。
  - 参加过天梯赛的同学可以刷天梯赛题目。
  - 如果有空，还可以去做 leetcode 热题 100，对考数据结构的同学较为有帮助。
  - 选择题也是要刷的，也是 pta 上的题目。

复习方法（续）

- **大一同学**：
  - 推荐多打印一些材料，尽可能多带。
- **大二同学**：
  - 推荐把算法都过一遍，哪怕复杂的代码不会写，也要把算法弄明白，至少会手动模拟。
  - 该背的算法要背，比如最短路的 dijkstra,floyd, 动态规划的背包问题等。

问答环节

欢迎提问

感谢倾听

谢谢大家！

祝大家复习顺利，考试加油！