**算法总论**

[基本算法 4](#_Toc72313294)

[1. 递推与递归 4](#_Toc72313295)

[2. 前缀和与差分 4](#_Toc72313296)

[3. 二分 4](#_Toc72313297)

[4. 排序 4](#_Toc72313298)

[5. 倍增 4](#_Toc72313299)

[6. 贪心 5](#_Toc72313300)

[7. 区间和的最值问题 5](#_Toc72313301)

[8. 总结 5](#_Toc72313302)

[基本数据结构 6](#_Toc72313303)

[1. 栈 6](#_Toc72313304)

[2. 队列 7](#_Toc72313305)

[3. 链表和邻接表 8](#_Toc72313306)

[4. 哈希 8](#_Toc72313307)

[5. 字符串 8](#_Toc72313308)

[6. 字典树 9](#_Toc72313309)

[7. 二叉堆 9](#_Toc72313310)

[8. 总结 10](#_Toc72313311)

[搜索 10](#_Toc72313312)

[1. 树与图的遍历 10](#_Toc72313313)

[2. 深度优先搜索 10](#_Toc72313314)

[3. 剪枝 10](#_Toc72313315)

[4. 迭代加深 11](#_Toc72313316)

[5. 广度优先搜索 12](#_Toc72313317)

[6. 广搜变形 12](#_Toc72313318)

[7. A\* 13](#_Toc72313319)

[8. IDA\* 13](#_Toc72313320)

[9. 总结 13](#_Toc72313321)

[数学知识 13](#_Toc72313322)

[1. 质数 13](#_Toc72313323)

[2. 约数 13](#_Toc72313324)

[3. 同余 13](#_Toc72313325)

[4. 矩阵乘法 13](#_Toc72313326)

[5. 高斯消元与线性空间 14](#_Toc72313327)

[6. 组合计数 14](#_Toc72313328)

[7. 容斥原理与莫比乌斯函数 14](#_Toc72313329)

[8. 概率与数学期望 14](#_Toc72313330)

[9. 0/1分数规划 14](#_Toc72313331)

[10. 博弈论 14](#_Toc72313332)

[11. 总结 16](#_Toc72313333)

[数据结构进阶 16](#_Toc72313334)

[1. 并查集 16](#_Toc72313335)

[2. 树状数组 16](#_Toc72313336)

[3. 线段树 17](#_Toc72313337)

[4. 分块 17](#_Toc72313338)

[5. 二叉搜索树与平衡树初步 17](#_Toc72313339)

[6. 离线分治算法 17](#_Toc72313340)

[7. 可持久化数据结构 17](#_Toc72313341)

[8. 总结 17](#_Toc72313342)

[动态规划 18](#_Toc72313343)

[1. 思想 18](#_Toc72313344)

[2. 线性DP 19](#_Toc72313345)

[3. 背包问题 19](#_Toc72313346)

[4. 区间DP 19](#_Toc72313347)

[5. 树形DP 19](#_Toc72313348)

[6. 环形与后效性处理 19](#_Toc72313349)

[7. 状态压缩DP 19](#_Toc72313350)

[8. 倍增优化DP 19](#_Toc72313351)

[9. 数据结构优化DP 19](#_Toc72313352)

[10. 单调队列优化DP 19](#_Toc72313353)

[11. 斜率优化DP 19](#_Toc72313354)

[12. 四边形不等式优化DP 19](#_Toc72313355)

[13. 计数类DP 19](#_Toc72313356)

[14. 数位统计类DP 20](#_Toc72313357)

[15. 总结 20](#_Toc72313358)

[图论 20](#_Toc72313359)

[1. 最短路 20](#_Toc72313360)

[2. 最小生成树 21](#_Toc72313361)

[3. 树的直径与最近公共祖先 21](#_Toc72313362)

[4. 基环树 21](#_Toc72313363)

[5. 负环与差分约数 21](#_Toc72313364)

[6. Tarjan算法与无向图的连通性 21](#_Toc72313365)

[7. Tarjan算法与有向图的连通性 21](#_Toc72313366)

[8. 二分图的匹配 21](#_Toc72313367)

[9. 二分图的覆盖与独立集 21](#_Toc72313368)

[10. 网络流 21](#_Toc72313369)

[11. 总结 21](#_Toc72313370)

[C++ STL用法总结 21](#_Toc72313371)

[**一、** **容器：** 21](#_Toc72313372)

[序列式容器： 21](#_Toc72313373)

[1. Vector 21](#_Toc72313374)

[2. Array 21](#_Toc72313375)

[3. List 21](#_Toc72313376)

[关联式容器： 21](#_Toc72313377)

[1. Set:底层结构式红黑树 21](#_Toc72313378)

[2. Multiset：底层结构是红黑树 22](#_Toc72313379)

[3. Map 22](#_Toc72313380)

[4. multimap 22](#_Toc72313381)

[5. Unordered\_set 22](#_Toc72313382)

[6. Unordered\_map 22](#_Toc72313383)

[**二、** **算法：** 22](#_Toc72313384)

[**三、** **迭代器：** 22](#_Toc72313385)

[**四、** **仿函数：** 22](#_Toc72313386)

[**五、** **适配器：** 22](#_Toc72313387)

[**六、** **配置器：** 22](#_Toc72313388)

# 基本算法

1. 位运算

位运算本身往往不会影响程序运行结果的正确性，但是合理地使用位运算的技巧，会使得程序变得非常高效。而之所以使用位运算的技巧，其实是与计算机设计的思想有关，我们都知道，计算机中的数字通过二进制串来表示，而二进制运算的过程其实就是一些电位开关的变化情况，这其实是计算机运行最快的方式，因为底层运行逻辑本就如此，所以为了优化程序执行的性能，我们可以使用位运算的技巧来优化程序结构。

位运算对于程序的优化我在这里主要分为两种情况：

* 1. 优化计算

顾名思义，就是使用位运算的技巧来优化计算速度，常用的技巧就是与、或、异或三个操作，此外还有左移、右移两种运算技巧。

运用以上技巧我们还可以实现子集枚举的功能。

* 1. 优化存储

使用位运算来优化存储，通常指的是使用多进制来保存一种全局的状态，这样我们可以以较小的空间存储足够多的状态。这种技巧我们通常称之为状态压缩，常用于动态规划问题中的优化。

* 1. 优化结构

1. 递推与递归

递推与递归思想，可能就是整个

1. 前缀和与差分
2. 前缀和
3. [激光炸弹](https://www.acwing.com/problem/content/101/)
4. [Max Add](https://atcoder.jp/contests/arc120/tasks/arc120_a)
5. [使数组元素相等的减少操作次数](https://leetcode-cn.com/problems/reduction-operations-to-make-the-array-elements-equal/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E7%AE%97%E6%B3%95/%E5%89%8D%E7%BC%80%E5%92%8C/%E4%BD%BF%E6%95%B0%E7%BB%84%E5%85%83%E7%B4%A0%E7%9B%B8%E7%AD%89%E7%9A%84%E6%9C%80%E5%B0%91%E6%93%8D%E4%BD%9C%E6%AC%A1%E6%95%B0)
6. 差分
7. [增减序列](https://www.acwing.com/problem/content/description/102/) [最高的牛](https://www.acwing.com/problem/content/103/)
8. 二分

**二分法是一种非常精妙的算法，很多算法的优化就来源于此方法，这里这要讲述两种常用的二分适用场景**

* 1. **对于明显的有序序列，我们可以使用二分快速查找。**

**例题：**

* 1. **对于隐含的有序序列，我们可以将搜索问题转化为判定问题快速求解，转化的关键在于判定时，我们判定的时间复杂度至少要低于搜索的时间复杂度log(n)级别的，否则将没有意义。**

**例题：**

1. [最佳牛围栏](https://www.acwing.com/problem/content/description/104/)
2. [向下取整整数对和](https://leetcode-cn.com/problems/sum-of-floored-pairs/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E7%AE%97%E6%B3%95/%E4%BA%8C%E5%88%86%E6%90%9C%E7%B4%A2/%E5%90%91%E4%B8%8B%E5%8F%96%E6%95%B4%E6%95%B0%E5%AF%B9%E5%92%8C)
3. 排序
4. 离散化：离散化指的是将大数映射到小数上，并保持他们的大小关系不变的操作。
5. 大数映射到小数是为了便于操作。
6. 保持大小关系是为了等比压缩，可以按照小数操作。

[电影](https://www.acwing.com/problem/content/105/)

1. 排序：
2. 选择、插入、冒泡
3. 不常用，基于比较的O(n^2)的排序算法
4. 堆排序，快排，归并
5. 快排：第k大的数
6. 归并：求解逆序对数目的问题 [超快速排序](https://www.acwing.com/problem/content/109/)
7. 堆排序：[动态中位数](https://www.acwing.com/problem/content/108/)
8. 计数，基数，桶排序
9. 倍增
10. 思想

我们设当前扩充的区间长度为p，区间的左右端点分别是L，R，那么倍增的主要操作为以下三步：

1. 初始化p=1,R=L.
2. 验证区间[L,R+p]是否满足条件，满足条件时更新R=R+p，p\*=2;不满足条件时R不变，p/=2;
3. 当p=0时，一次倍增操作结束，此时R即为所求区间的右端点。
4. 单调递增：
5. 快速幂+倍增：[graph smoothing](https://atcoder.jp/contests/abc199/tasks/abc199_f)
6. ST算法：[数列区间的最大值](https://www.acwing.com/problem/content/1272/)
7. 单调递减：
8. LCA：
9. 先增后减
10. 倍增搜索：[天才ACM](https://www.acwing.com/problem/content/description/111/)
11. [向下取整整数对和](https://leetcode-cn.com/problems/sum-of-floored-pairs/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E7%AE%97%E6%B3%95/%E5%80%8D%E5%A2%9E/%E5%90%91%E4%B8%8B%E5%8F%96%E6%95%B4%E6%95%B0%E5%AF%B9%E5%92%8C)
12. 贪心
13. 介绍：贪心的思想具有很大的技巧性在里面，我们通常需要找到某一个角度，由局部最优推导出全局最优，这样我们才可以使用贪心的思想。
14. 微扰（邻项交换）

证明在任意局面下，任意对局部最优策略的微小改变都会造成整体结果变差。经常用于以排序为贪心策略的证明。

1. [均分纸牌](https://www.acwing.com/problem/content/1538/)
2. 范围缩放

证明任何对局部最优策略作用范围的扩展都不会造成整体结果变差。

1. 决策包容

证明在任意局面下，做出局部最优决策后，在问题状态空间中的可达集合中包含了作出其他任何决策后的可达集合。换言之，这个局部最优策略提供的可能性包含其它所有策略提供的可能性。

1. [七夕祭](https://www.acwing.com/problem/content/107/)
2. 反证法
3. 数学归纳法
4. [货仓选址](file:///C:\Users\windows\Desktop\acwing.com\problem\content\106\)（中位数）

其他例题：

1. [构成交替字符串的最少交换次数](https://leetcode-cn.com/problems/minimum-number-of-swaps-to-make-the-binary-string-alternating/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E7%AE%97%E6%B3%95/%E8%B4%AA%E5%BF%83/%E6%9E%84%E6%88%90%E4%BA%A4%E6%9B%BF%E5%AD%97%E7%AC%A6%E4%B8%B2%E7%9A%84%E6%9C%80%E5%B0%91%E4%BA%A4%E6%8D%A2%E6%AC%A1%E6%95%B0)

经典例题，其实也谈不上是贪心问题，者更偏向与一个脑筋急转弯。对于对于一个交替字符串来说，当我们确定奇数和偶数位置的数是什么后，奇数位置不满足条件的字符数量一定等于偶数位置不满足条件的字符数量，我们只需要统计记录较小的一种分配方案即可。

1. 区间和的最值问题

给定一个包含负数的数组，求解区间和的最大值。

* 1. **利用滑动窗口机制，及时剪枝的贪心做法**
  2. **将问题转化为扫描并维持一个前缀和最小的的方法，具体方法对于每个扫描到的位置i，以该点为右端点的区间和的最值，一定是sum[i]与sum[i]之前的前缀和的最小值之间的差值。**

[最大子序和](https://leetcode-cn.com/problems/maximum-subarray/submissions/) [最佳牛围栏](https://www.acwing.com/problem/content/description/104/)

1. 总结

任何问题都有其可行解的范围空间，每个可行解有一系列的状态组成，所有状态空间组成的空间就是问题搜索过程中的状态空间，我们实现算法的目的就是在可行

# 基本数据结构

1. 栈
2. 括号匹配+后缀表达式计算

栈的最经典应用就是括号匹配以及表达式计算。

1. [有效的括号](https://leetcode-cn.com/problems/valid-parentheses/)
2. 单调栈问题

对于区间的一个端点不变，而另一个端点变化的区间最值问题，我们可以使用单调栈求解。

1. [包含min的栈](https://leetcode-cn.com/problems/bao-han-minhan-shu-de-zhan-lcof/) （哈希+单调栈）
2. [下标对中的最大距离](https://leetcode-cn.com/problems/maximum-distance-between-a-pair-of-values/submissions/)(双指针 单调栈+双指针 单调栈+二分)
3. [子数组最小乘积的最大值](https://leetcode-cn.com/problems/maximum-subarray-min-product/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E6%A0%88/%E5%AD%90%E6%95%B0%E7%BB%84%E4%B8%AD%E6%9C%80%E5%B0%8F%E4%B9%98%E7%A7%AF%E7%9A%84%E6%9C%80%E5%A4%A7%E5%80%BC)
4. [直方图中最大矩形面积](https://www.acwing.com/problem/content/133/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E6%A0%88/%E5%8D%95%E8%B0%83%E6%A0%88-%E7%9F%A9%E5%BD%A2%E9%9D%A2%E7%A7%AF.cpp)
5. 卡特兰数
6. [火车进出栈问题](https://www.acwing.com/problem/content/132/)

方法一：枚举 [卡特兰数-枚举法](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E6%A0%88/%E5%8D%A1%E7%89%B9%E5%85%B0%E6%95%B0-%E6%9E%9A%E4%B8%BE%E6%B3%95.cpp)

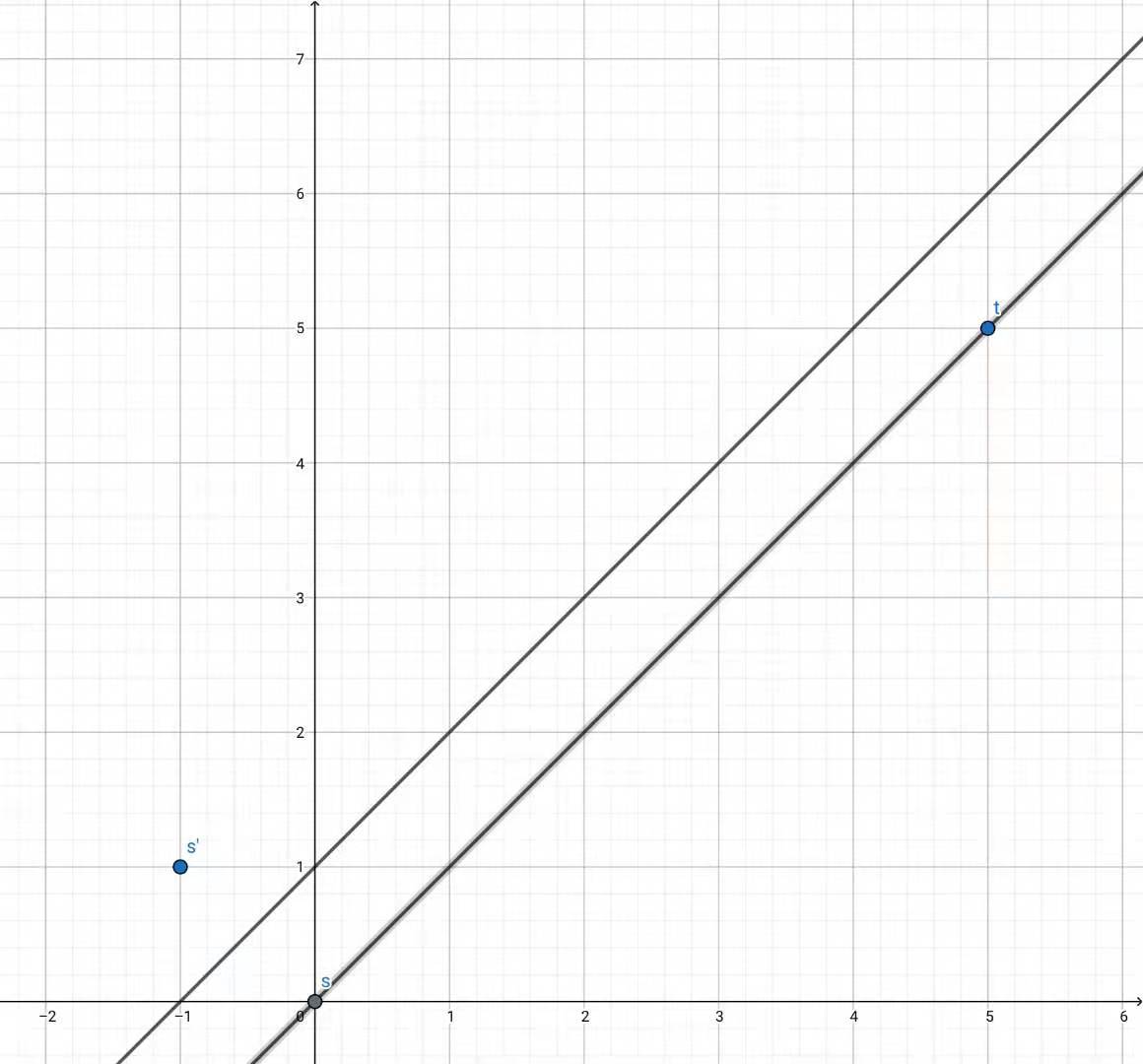
方法二：递推 [卡特兰数-递推法](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E6%A0%88/%E5%8D%A1%E7%89%B9%E5%85%B0%E6%95%B0-%E9%80%92%E6%8E%A8%E6%B3%95.cpp)

实际上满足卡特兰数性质的问题有很多，进栈出栈问题，二叉树子树问题，凸多边形的三角形划分等。值为n的卡特兰数我们定义为，0和1的卡特兰数为1，通过以下递推公式我们可以由子问题推导出。

方法三：DP [卡特兰数-dp法](https://github.com/run916/-/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E6%A0%88/%E5%8D%A1%E7%89%B9%E5%85%B0%E6%95%B0-dp%E6%B3%95.cpp)

方法四：数学

1. 非降路径问题。



对应于进栈出栈问题，实际上就是上图中点s到点t的所有路径中只位于x=y这条直线以下的路径总数。而不考虑路径限制的条件下，点s到点t的路径总数为组合数C(2n,n)，我们要从这些路径里面剔除不满足条件的路径，而不满足条件的路径一定是经过直线y=x+1的路径，我们如何计算所有经过直线y=x+1的路径呢？这里的一个技巧，我们取s关于直线y=x+1的对称点s’，s到t经过直线y=x+1的路径总数与s’到t的路径总数相同，这样我们就得到了卡特兰数的数学解：

1. 01序列问题。

假设我们用0代表进站，1代表出站，那么01序列问题中，满足进栈出栈限制的序列是从左往右数，数字0的数目总是大于等于1的序列数目。

证明：

显然我们只需要从所有可能的01序列中减去不满足条件的01序列即可。那么如何求解不满足条件的01序列呢？对于一个不满足条件的01序列一定存在某一个奇数前缀是满足条件的，而后半部分不满足条件，那么如果我们把后半部分的01序列中的0和1互换，那么每一个不满足条件的01序列就对应着一个由n+1个1和n-1个0对应的01序列。

另一种思路是，不合法的序列是某一个前缀中1的数目多于0的数目的序列，那么我们从2n个位置中选取n-1位置作为0，那么剩余的n+1个位置都是1，如果我们将剩余n+1个位置中的最后一个位置变成0，那么此时该序列一定是不满足条件的。

1. 队列

队列是一种先进先出结构，常用来做层次遍历的结构。

1. 队列例题。
2. [小组队列](https://www.acwing.com/problem/content/description/134/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E9%98%9F%E5%88%97/%E9%98%9F%E5%88%97%E4%BE%8B%E9%A2%98/%E9%98%9F%E5%88%97%E4%BE%8B%E9%A2%98)
3. 单调队列问题
4. [滑动窗口最大值](https://leetcode-cn.com/problems/sliding-window-maximum/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E9%98%9F%E5%88%97/%E5%8D%95%E8%B0%83%E9%98%9F%E5%88%97%E4%BE%8B%E9%A2%98/%E6%9C%80%E5%A4%A7%E5%AD%90%E5%BA%8F%E5%92%8C%E5%8F%98%E5%BD%A2)
5. [最大子序和变形](https://www.acwing.com/problem/content/description/137/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E9%98%9F%E5%88%97/%E5%8D%95%E8%B0%83%E9%98%9F%E5%88%97%E4%BE%8B%E9%A2%98/%E6%BB%91%E5%8A%A8%E7%AA%97%E5%8F%A3%E6%9C%80%E5%A4%A7%E5%80%BC)
6. 优先级队列问题：

采用优先级队列的问题，往往与贪心的思想是相互配合的，例如非常经典的Dijkstra算法，

例题：

1. [包含每个查询的最小区间](https://leetcode-cn.com/problems/minimum-interval-to-include-each-query/)

**方法一：**

这道题的贪心策略相对简单一些，对于将要查询的点i，我们要查找覆盖该点的最小区间的编号，所以我们只需要将所有区间按照左端点排序，将所有左端点小于查询点i，右端点大于等于查询点i的区间加入优先级队列（优先级队列以区间长度为比较维度），即可求解出一个查询的最优值，而对于所有查询，我们可以将查询排序，在按照单个查询相同的思路，从左到右遍历每一个查询，同时将满足要求的区间加入，不满足要求的区间删除即可求解该问题。

**方法二：**

除了方法一以外，根据数据范围，我们还可以利用映射的关系进行求解，首先将所有的区间排序，然后建立一个 10^7的数组用来记录每个位置对应的区间最小值，然后再将所有的区间覆盖的区域进行更新，最终得到每一个位置的最小值，然后我们直接在处理后的数组上查询即可。

**方法三：**

线段树，采用线段树这一数据结构来完成方法二中对于区间的更新，然后直接返回要查询的值即可。

1. [批处理任务](https://leetcode-cn.com/problems/t3fKg1/)
2. 链表和邻接表
3. 链表是一种非常重要的结构，我们要熟练掌握建立链表的细节。
4. 链接链表是一种表示图的数据结构，通常我们建立领接链表的方式有以下两种：
5. 用vector动态数组模拟链表。优点是便于理解实现，缺点是时间复杂度稍高。

模板

1. 用兄弟链表法，优点是速度快，缺点是不便于理解。

[模板](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E9%93%BE%E8%A1%A8%E5%92%8C%E9%82%BB%E6%8E%A5%E9%93%BE%E8%A1%A8/%E9%82%BB%E6%8E%A5%E9%93%BE%E8%A1%A8-%E5%85%84%E5%BC%9F%E9%93%BE%E8%A1%A8%E6%B3%95)

1. 哈希
2. 哈希表是一种非常有用的数据结构，利用哈希表我们可以设计出很多巧妙的结构和算法。
3. STL里面的unordered\_map以及unordered\_set就是哈希表，这两者是利用开散列的方式解决冲突的，也即通过拉链法解决冲突。
4. 哈希表用于解决某些构造性问题时会有奇效。
5. 计数排序就是基于哈希思想的。

例题：

1. [找出和为指定值得下标对](https://leetcode-cn.com/problems/finding-pairs-with-a-certain-sum/submissions/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E7%AE%97%E6%B3%95/%E5%93%88%E5%B8%8C/%E6%89%BE%E5%87%BA%E5%92%8C%E4%B8%BA%E6%8C%87%E5%AE%9A%E5%80%BC%E7%9A%84%E4%B8%8B%E6%A0%87%E5%AF%B9)
2. [装包裹的最小浪费空间](https://leetcode-cn.com/problems/minimum-space-wasted-from-packaging/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%93%88%E5%B8%8C/%E8%A3%85%E5%8C%85%E8%A3%B9%E7%9A%84%E6%9C%80%E5%B0%8F%E6%B5%AA%E8%B4%B9%E7%A9%BA%E9%97%B4)

**字符串哈希：**

1. 字符串哈希是将字符串映射成一个k进制数的过程，不过映射后的数可能比较大，我们通常取这个字符串对应的数的hash值，k我们通常取为131或者13331，此时哈希冲突的概率极低。
2. 此时我们可以用字符串的哈希值来比较两个字符串是否相等。这个思想其实就是RK字符串匹配的思想。
3. 最长回文子串问题可以中字符串哈希+二分的方式将时间复杂度维持在O(nlogn)级别。
4. 字符串

字符串处理是计算机里面一个非常重要的工作，本质上字符串问题没有特定的算法，我们只是利用已有的算法来解决字符串问题。最为经典的字符串问题应该是以下几个：

1. 字符串匹配
2. KMP算法：[KMP模板](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%AD%97%E7%AC%A6%E4%B8%B2/KMP)

[周期](https://www.acwing.com/problem/content/description/143/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%AD%97%E7%AC%A6%E4%B8%B2/%E5%91%A8%E6%9C%9F-KMP%E4%BE%8B%E9%A2%98)

这里简单地谈一谈我对KMP算法的理解。首先从问题本身出发，对于一个字符串匹配问题来说，假设将要匹配的字符串为s，模式串为p，我们在匹配过程中的想法是，如果p的第k位与s的第i位匹配失败后，我们希望尽快得到下一个可以继续比较的p的最大前缀的长度，假设我们用pi[k-1]来表示这一长度，那么当失败的情况发生后，我们可以快速得到p[1,k-1]与s[1,i-1]匹配的次大的长度，此时我们可以尝试继续比较，这一过程是一个可以不断迭代的过程，也即如果新的长度依然匹配不成功我们可以继续尝试上述操作，那么整个操作过程可以总结为以下两步：

1. 加入当前要比较的位置是s的第i个位置，p已经与s匹配的前缀长度为k，那么比较第k+1位时，如果匹配成功则匹配s的第i+1个位置的字符，否则转步骤二。
2. 用pi[k]获得与p[1,k]的真后缀匹配的p的前缀的最大长度。
3. 继续步骤一、二。

观察后发现，这实际上就是一个动态规划的过程，我们只需要求解处模式串p的pi数组即可，按照pi的定义，p与自身匹配的过程来计算出这个数组。

1. RK算法
2. 最长公共子序列（后续DP部分介绍）
3. 最长回文子串（后续DP部分介绍）
4. 字符串的最小表示
5. [隐藏密码](https://www.acwing.com/problem/content/description/1412/)（模板题）[答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%AD%97%E7%AC%A6%E4%B8%B2/%E9%9A%90%E8%97%8F%E5%AF%86%E7%A0%81-%E6%9C%80%E5%B0%8F%E8%A1%A8%E7%A4%BA%E6%B3%95%E6%A8%A1%E6%9D%BF%E9%A2%98)

最小表示法还是一道比较精妙的题，该问题的技巧就是在比较过程中进行最优性剪枝，从而是的算法的时间复杂度退化到O(n)。

1. 字典树
2. 字典树是另一种快速实现字符串匹配的数据结构，他是一个多叉树结构，这个多叉树结构是比较松散的，与我们平常理解的树结构是一致的。
3. 字典树中保存字符串每个字符的实际上是多叉树中的边，每个节点链出的边就代表了一个字符。

例题：[模板](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%AD%97%E5%85%B8%E6%A0%91/%E5%AD%97%E5%85%B8%E6%A0%91-%E6%A8%A1%E6%9D%BF)

1. [前缀统计](https://www.acwing.com/problem/content/description/144/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%AD%97%E5%85%B8%E6%A0%91/%E5%89%8D%E7%BC%80%E7%BB%9F%E8%AE%A1)
2. [数组中两个数的最大异或值](https://www.acwing.com/problem/content/145/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%AD%97%E5%85%B8%E6%A0%91/%E6%95%B0%E7%BB%84%E4%B8%AD%E4%B8%A4%E4%B8%AA%E6%95%B0%E7%9A%84%E6%9C%80%E5%A4%A7%E5%BC%82%E6%88%96%E5%80%BC)
3. [最长异或值路径](https://www.acwing.com/problem/content/146/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%AD%97%E5%85%B8%E6%A0%91/%E6%9C%80%E9%95%BF%E5%BC%82%E6%88%96%E8%B7%AF%E5%BE%84)
4. 二叉堆
5. 实际上就是堆，也被称作是优先级队列，前面在介绍优先级队列时已经提到过了。

例题：

1. [超市](https://www.acwing.com/problem/content/description/147/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%A0%86/%E8%B6%85%E5%B8%82) [方法二](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%A0%86/%E8%B6%85%E5%B8%82-%E7%AC%AC%E4%BA%8C%E7%A7%8D%E6%80%9D%E6%83%B3)
2. [增长的内存泄露](https://leetcode-cn.com/problems/incremental-memory-leak/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%A0%86/%E5%A2%9E%E9%95%BF%E7%9A%84%E5%86%85%E5%AD%98%E6%B3%84%E9%9C%B2)
3. [使用服务器处理任务](https://leetcode-cn.com/problems/process-tasks-using-servers/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/%E5%A0%86/%E4%BD%BF%E7%94%A8%E6%9C%8D%E5%8A%A1%E5%99%A8%E5%A4%84%E7%90%86%E4%BB%BB%E5%8A%A1)
4. 霍夫曼树——霍夫曼树是对堆的一个应用，通常用来做字符串压缩的工作。霍夫曼树的构建过程我们可以用以下几个步骤来描述：
5. 首先将所有的字符权值加入到堆中。
6. 当堆中元素为1时，算法结束，返回该节点为霍夫曼树根节点。
7. 从堆中连续弹出两个最小权值的点，合并这两个点为根的子树，并生成一个新的节点作为两个子树根节点的父节点，新生成的节点的权值为两个子树权值和。将合并后的节点插入到堆中。
8. 重复步骤B，C。
9. 多叉霍夫曼树的构建方式与二叉霍夫曼树稍稍不同，我们需要先补充一些权值为0的节点，使得最后一步时可以恰好只剩1个根节点。

**总结：**

优先级队列有几个非常重要的应用场景：

1. 任务安排问题。此类问题通常要求按照某个权值优先的顺序，按照时间顺序安排任务，此时我们通常需要动态维持一个有序数组，这是优先级队列最为擅长的能力。抽象后我们其实得到这样一条结论——维持动态有序数组的问题我们需要使用优先级队列。
2. 最小生成树，单源最短路等经典算法也使用了优先级队列。
3. 维持top k的元素。

其实总结起来就一句话——**维持动态有序数组的问题**，我们都可以使用优先级队列。

1. 总结

学习算法总结不能少，这一章结束后我们算是稍微进了算法的大门，了解了一些基础的数据结构，本章知识点主要如下：

栈、队列、链表与邻接链表、哈希、字符串、字典树、二叉堆。

1. 活学活用才是硬道理。

# 搜索

1. 树与图的遍历
2. 树的遍历是处理图问题最为核心的一种手段，我们可以使用一些而外的标记数组来记录遍历过程中的数据。
3. 除了显式的图结构、树形结构以外，实际上现实中的很多问题都可以抽象成对图的搜索，计算机解决问题的本质其实就是搜索可行解，如果我们把搜索过程中的每个中间状态表示为图中的一个节点，而将不同中间态之间的转化关系看作是图中的边，那么所有问题都可以抽象为使用搜索算法在可行空间里面搜索解的过程。
4. 深度优先搜索
5. 深度优先遍历需要借助递归来实现，最基本的深度优先搜索问题我主要列出了四种，其中树的深度是基于自顶向下的思想实现的，数的重心是基于自底向上的思想实现的，无向图的连通分支则是最朴素的dfs搜索过程，拓扑排序是对dfs的扩展应用，指出了标记数组的作用。

例题：

1. 树的深度：[模板](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E6%A0%91%E4%B8%8E%E5%9B%BE%E7%9A%84%E9%81%8D%E5%8E%86/%E6%A0%91%E7%9A%84%E6%B7%B1%E5%BA%A6-dfs)
2. 树的重心：[模板](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E6%A0%91%E4%B8%8E%E5%9B%BE%E7%9A%84%E9%81%8D%E5%8E%86/%E6%A0%91%E7%9A%84%E9%87%8D%E5%BF%83-dfs)
3. 无向图的连通分支数：[村村通](https://www.luogu.com.cn/problem/P1536) [答案](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E6%A0%91%E4%B8%8E%E5%9B%BE%E7%9A%84%E9%81%8D%E5%8E%86/%E6%9D%91%E6%9D%91%E9%80%9A-%E8%BF%9E%E9%80%9A%E5%88%86%E6%94%AF)
4. 拓扑排序：
5. [模板题](http://poj.org/problem?id=2367) [dfs](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E6%B7%B1%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2/%E5%AE%B6%E6%97%8F%E8%B0%B1-%E6%8B%93%E6%89%91%E6%8E%92%E5%BA%8F-dfs) [bfs](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2/%E5%AE%B6%E6%97%8F%E8%B0%B1-%E6%8B%93%E6%89%91%E6%8E%92%E5%BA%8F-bfs)
6. [确定比赛名次](http://acm.hdu.edu.cn/showproblem.php?pid=1285) [答案](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E6%B7%B1%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2/%E6%8B%93%E6%89%91%E6%8E%92%E5%BA%8F-%E6%96%B9%E6%B3%95%E4%B8%80)
7. 搜索与剪枝往往是相辅相成的，合理的剪枝会显著地降低搜索的时间，但是剪枝不会改变问题的复杂度级别，原来的算法是什么样的的时间复杂度，剪枝后也依然是该时间复杂度。
8. [将字符串拆分成递减的连续值](https://leetcode-cn.com/problems/splitting-a-string-into-descending-consecutive-values/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E5%89%AA%E6%9E%9D/%E5%B0%86%E5%AD%97%E7%AC%A6%E4%B8%B2%E6%8B%86%E5%88%86%E6%88%90%E9%80%92%E5%87%8F%E7%9A%84%E8%BF%9E%E7%BB%AD%E5%80%BC)
9. 剪枝

剪枝并不会改变一个算法固有的时间复杂度，但是基于随机性理论，剪枝可以排除很多不必要的搜索，实际的运行时间可能很快。如果搜索的每一个中间状态都看作一个节点，那么搜索过程中会形成一个树。剪枝操作就是在这个树上做判断，及时避免一些不必要的搜索。那么在这个树上搜索时，减少节点遍历次数的手段就是剪枝。常见的剪枝手段有以下5种：（注：下文中节点都代表搜索过程中的一个状态）

* **优化搜索顺序**

一般而言，对于搜索一种可行解的问题，我们通过优化搜索顺序的方式，可以尽快得到一种可行解，从而避免一些无意义节点的遍历。

例题：

1. [小猫爬山](https://www.acwing.com/problem/content/167/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E5%89%AA%E6%9E%9D/%E5%B0%8F%E7%8C%AB%E7%88%AC%E5%B1%B1)
2. [数独](https://www.acwing.com/problem/content/168/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E5%89%AA%E6%9E%9D/%E6%95%B0%E7%8B%AC)
3. [使二进制字符串字符交替的最少反转次数](https://leetcode-cn.com/problems/minimum-number-of-flips-to-make-the-binary-string-alternating/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E5%89%AA%E6%9E%9D/%E4%BD%BF%E4%BA%8C%E8%BF%9B%E5%88%B6%E5%AD%97%E7%AC%A6%E4%B8%B2%E5%AD%97%E7%AC%A6%E4%BA%A4%E6%9B%BF%E7%9A%84%E6%9C%80%E5%B0%91%E5%8F%8D%E8%BD%AC%E6%AC%A1%E6%95%B0)

* 排除等效冗余

前面我们说搜索过程形成的结构是一颗搜索树，但是实际上我们可以将一些子问题相同的状态合并，最后搜索结构就成了一个有向无环图。在这个有向无环图上搜索时，某些位置我们可能多次遍历，而实际上我们只需要遍历一次即可，所以可以排除这些节点的冗余遍历。

* 可行性剪枝

按照字面意思理解即可，对于不可行的分支，我们及时排除，例如问题对搜索的步数有限制，资源的消耗有限制，通过对当前状态的粗略分析，我们发现即使在最优条件下，也无法满足要求，那么这个分支的搜索我们就可以剪去。

* 最优性剪枝

与可行性剪枝非常类似，但是有一些区别，最优性剪枝是在已经有一个当前最优结果下的剪枝，当我们已经搜索到一种可行解后，如果我们要求解的是最优解，那么后面搜索过程中只要搜索到的状态的值大于等于最优解，那么这条分支搜索后的结果不会优于当前最优解，所以我们也可以及时剪枝。

* 记忆化

记忆化是将我们搜索过的相同分支进行记录，比如说斐波那契堆问题，f(n)会同时被f(n+1)和f(n+2)搜索，那么如果我们每次都从头到尾搜索一遍，花费的代价是很大的。基于此，我们可以将f(n)的第一次搜索的结果记录下来，后续搜索到时，直接返回结果即可。

* 总结

各种剪枝方式是可以相互配合使用的，比如最优性剪枝和可行性剪枝就可以相互配合，实现更为精妙的剪枝方法。

实际上搜索算法面对的状态可以看作一个多元组，其中每一元都是问题状态空间中的一个维度。每个维度发生变化都会使搜索移动到一个新的状态。搜索过程中的剪枝，其实就是针对每个维度与该维度的边界条件，加以放缩，推导，得到一个不等式，来减少搜索分支的扩张。

1. 迭代加深

深度优先搜索有一个不是缺点的缺点，那就是，它在搜索过程中会优先往下搜索。如果有一个问题它的结果在某个比较浅层的节点上，那么我们在搜索到结果之前可能要经过很多不必要的搜索。这个时候如果我们按照深度递增的顺序加以限制搜索，那么时间复杂度可能会降低很多。这种搜索方式就是迭代加深。

迭代加深的搜索过程中浅层节点可能被多次搜索，但是相比于搜索树指数级别的扩充，这些额外的搜索又不值一提。可能这个时候有人就想到了，为什么不用广度优先搜索呢？我们当然可以使用广度优先搜索解决该问题，但是深度优先搜索的实现方式通常是用递归方式实现的，我们很容易就可以通过递归的形式写出一个搜索结构，然而广度优先搜索可能需要在队列中维持较多的状态，不便于实现。

**双向搜索：**

除了迭代加深以外，双向搜索也可以避免在深层子树上浪费时间。而要使用双向搜索，我们需要保证问题里面不但有一个初始状态，还有一个终点状态，这样我们就可以通过初始状态与终点状态的同时搜索，来减少搜索空间的大小。

1. 广度优先搜索

广度优先搜索是搜索的另一种方式，它的搜索偏好是优先搜索一个状态的兄弟节点，为了快速得到兄弟节点，我们需要在遍历的过程中用一个队列来保存兄弟节点的数据。

实际使用过程中广度优先搜索常用来实现层次遍历。而层次遍历的过程有一个最短路的性质，所以广度优先搜索也常用来做走地图类似的最短路问题。

例题：

1. 推箱子
2. [立体推箱子](https://www.acwing.com/problem/content/174/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2/%E7%AB%8B%E4%BD%93%E6%8E%A8%E7%AE%B1%E5%AD%90)

这道题的思想并不难，但是实现很复杂，代码中有很多做类似问题的遍历技巧。

1. 记忆化优化。遍历的过程中我们通常需要记录一些已经遍历到的状态，避免同一个局面被重复遍历，这可以保证算法的复杂度上界。
2. 迷宫问题每一步的遍历选择通常有上下左右四个方向，我们如果用if-else的形式遍历，会让代码显得比较冗余，同时难以维护，美观性也不好，这里我们采用DFA的思想，用一个状态转换表，来实现每一步遍历时，x和y的变化情况，我们只需要用一个统一的for循环即可实现所有可能的遍历。
3. 进一步扩充，本题中每一步变化时，x和y的变化形式还与当前箱子的状态有关，这样我们可以进一步扩充状态转换矩阵，从而在一个简单的for循环内部完成所有可能的变化。
4. 广搜变形
5. **双端队列BFS**

例题：

1. [电路维修](https://www.acwing.com/problem/content/description/177/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E5%B9%BF%E6%90%9C%E5%8F%98%E5%BD%A2/%E5%8F%8C%E7%AB%AF%E9%98%9F%E5%88%97BFS/%E7%94%B5%E8%B7%AF%E7%BB%B4%E4%BF%AE)
2. **优先级队列BFS**

双端队列BFS可以看作是优先级队列BFS的特例，上面的例题我们完全可以使用优先级队列BFS实现，但是双端队列BFS的时间复杂度是线性的，而优先级队列BFS的复杂度则是nlogn的。这是因为双端队列中我们利用了队列中的特性，即对头队尾的层次差最多为1，而在更为一般的问题中，每次扩充的元素，其层次增加可能并不是1，此时我们要维持队列中元素的有序性，就需要用优先级队列这一数据结构进行辅助。

按照我们之前描述的搜索树的结构，实际上采用优先级队列求解最优解的过程，就是在一个图上求解最短路的过程。

例题：[单源最短路](https://www.acwing.com/problem/content/submission/852/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/%E5%B9%BF%E6%90%9C%E5%8F%98%E5%BD%A2/%E4%BC%98%E5%85%88%E7%BA%A7%E9%98%9F%E5%88%97/%E5%8D%95%E6%BA%90%E6%9C%80%E7%9F%AD%E8%B7%AF)

2. **总结：**

实际解决问题过程中，我们可以根据图上边权的情况选择合适的BFS形式。

1. 边权抽象为1，求解某个状态到其他状态的最少步数，我们可以使用普通BFS形式。
2. 边权为0或者1，求解某个状态当其他状态的最少步数，我们可以使用双端队列BFS形式。特点是每个状态入队多次，但是只会扩展1次，我们需要一个vise数组标记某个状态是否被扩展过，并用一个距离数组存储最短路的长度。
3. 边权任意，求解某个状态到其他状态的最短距离，此时使用优先级队列BFS形式。优先级队列BFS的特点是，每个状态入队多次，但是只会被扩展1次。这需要一个标记数组vise来标记每个节点是否被扩展过，还要使用一个记录最短路的数组。
4. **双向BFS**

与之前双向搜索的思想类似，两者的思想本质相同，只有搜索的手段不同。

1. A\*

给定一个目标状态，BFS搜索时，是按照与源点的距离最近的贪心思想扩展的，这种思想是一种贪心策略的思想。贪心的策略是优先扩充距离源点更近的节点，但是最优解可能是距离源点比较远的节点，之前我们提到的双向搜索的思想就是减少这种贪心策略使得我们搜索过多无用的状态。但是这并不够完善，实际上，如果我们能够预估出当前点到目标点的距离，那么我们可以优先选择预估值与当前值的和较小的状态优先扩展。

这实际上就是利用预估值，按照所有可能路线的权值大小的顺序进行扩展的思想，避免我们只考虑路径的一部分时，陷入局部最优的扩展中。我们将这种利用“**当前代价+预估代价**”进行扩展的BFS搜索算法，称之为A\*算法。

为了保证算法的正确性，我们要保证当前代价+预估代价要小于等于最优值，否则扩展可能是不正确的。

**证明：**

A\*算法的正确性其实是通过最优性不等式保证的，因为估价函数与当前代价的和不可能超过最优解，那么随着算法的扩展，非最优分支的扩展，会在某个点发生当前代价+预估代价大于最优解搜索路径上某个点的当前代价+预估代价的情况，那么此时扩展就会从最优解这条分支进行，最终保证目标状态第一次被扩展时一定是最优解。

例题：

1. [第K短路](https://www.acwing.com/problem/content/180/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%90%9C%E7%B4%A2/A*/%E7%AC%ACK%E7%9F%AD%E8%B7%AF)
2. IDA\*

IDA\*算法本质上与A\*算法的思想相似，但是使用了迭代加深的搜索方式，而迭代加深我更喜欢将其看作是BFS算法的DFS近似实现，优点是编程容易，代码简单，缺点是会有一个不必要的遍历。这里我不在介绍其思想。

例题：

1. 总结

# 数学知识

1. 质数
2. 约数
3. 同余
4. 矩阵运算
5. 矩阵转置
6. 矩阵旋转

旋转可以通过转置+交换的方式来实现。

1. [旋转盒子](https://leetcode-cn.com/problems/rotating-the-box/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E7%9F%A9%E9%98%B5%E8%BF%90%E7%AE%97/%E7%9F%A9%E9%98%B5%E6%97%8B%E8%BD%AC/%E6%97%8B%E8%BD%AC%E7%9B%92%E5%AD%90)
2. [判断矩阵轮转后是否一致](https://leetcode-cn.com/problems/determine-whether-matrix-can-be-obtained-by-rotation/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E7%9F%A9%E9%98%B5%E8%BF%90%E7%AE%97/%E7%9F%A9%E9%98%B5%E6%97%8B%E8%BD%AC/%E5%88%A4%E6%96%AD%E7%9F%A9%E9%98%B5%E8%BD%AE%E8%BD%AC%E5%90%8E%E6%98%AF%E5%90%A6%E4%B8%80%E8%87%B4)
3. 高斯消元与线性空间
4. 组合计数
5. 容斥原理与莫比乌斯函数
6. 概率与矩阵乘法
7. 数学期望
8. 0/1分数规划
9. 博弈论

博弈论是起源于微观经济学，是以数学为基础，与经济学相结合的边缘学科。既然称之为博弈论，那么它研究的问题中设计的主体至少有两个。博弈论研究的是参与博弈的多方都站在各自最优的角度博弈时，最终可以达到一个均衡状态，达到这个状态所做的一系列决策就是博弈的结果。

我们来介绍一些关于博弈中的术语，首先加入参与博弈的只有两方，我们将先行动的人称之为先手，而将后行动的人称之为后手。如果存在某一种先手行动方式，使得后手不管怎么操作都面临必败局面，那么我们成先手必胜，反之我们也称之为后手必胜。

当然很多问题并不是询问博弈双方谁获胜，另一类问题的目标是询问达到博弈均衡状态是先手获得最大收益，我会在后续的例题中介绍。

1. NIM博弈

NIM博弈是一个非常经典的博弈问题，问题是这样描述的，给定n堆物品，第i堆物品有个。两名玩家轮流行动，每次可以任选一堆，取走任意多个物品，可以把一堆都取光，但是不能不取。取走最后一件物品者获胜。两人都采取最优策略，问先手是否能够必胜。

NIM博弈有一个非常巧妙的结论——当时，先手必胜。

反证法：

对于任意一个局面，如果，设x的最高位1在第k位，那么必然存在一个，它的第k位时1，此时我们得到，那么我们可以从中取掉若干个石子，使得剩余石子数目为个，此时就得到了一个各堆石子异或值等于0的局面。

对于一个各堆石子异或值为0的局面，我们按照NIM游戏的规则，不可能在一次取石子时，使得下一个局面中各堆石子的异或值为0，所以这个局面一定是必输的局面。因为每次必须至少取一个石子，那么对于有限的石子来说，所有的石子一定会被取完，而各堆石子异或值为0的局面下，玩家不可能一次使得石子数变为0，所以当前局面一定不是最后拿石子的人，所以当前局面一定是必输局面。

这样我们可以得到各堆石子异或值不为0的局面是必胜局面。反之则是必输局面。

例题：

1. [NIM博弈](https://www.luogu.com.cn/problem/P2197) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/NIM%E5%8D%9A%E5%BC%88)
2. 公平组合游戏ICG

若一个游戏满足以下三个条件，那么它是一个公平组合游戏。

1. 有两名玩家交替行动。
2. 在游戏进程的任意时刻，可以执行的合法行动与轮到那名玩家无关。
3. 不能行动的玩家判负。
4. 有向图游戏

对于一个有向无环图，图中有一个唯一的起点，在起点上有一枚棋子，两名玩家交替地将这枚棋子向前移动，每次可以移动一步，不能移动的玩家判负。

公平组合游戏都可以转化为有向图游戏。具体过程就是将博弈过程中的每个局面抽象成一个点，而将可能的行动抽象成每一条边。

1. SG函数

我们首先定义mex运算，mex(S)定义为不属于集合S的最小非负整数。

根据以上定义我们将SG函数定义为下面的表达式：

上式中，表示从可以到达的局面。特别的整个有向无环图的SG函数定义为：

1. 有向图游戏的和

设是m个有向图游戏。定义有向图游戏*G*，它的行动规则是任选某个有向图游戏，并在上行动一步。*G*被称为有向图游戏的和。

有向图游戏的和的SG函数等于它包含的各个子游戏的SG函数的异或和。

1. 定理
2. 有向图游戏的某个局面必胜，当且仅当该局面对应的SG函数的值大于0.
3. 有向图游戏的某个局面必败，当且仅当该局面对应的SG函数的值等于0.
4. 有向图没有出度的点的SG函数值可以直观求解到，为0，是必胜局面。
5. 实际上如果我们能将问题抽象成为有向图游戏，那么我们可以按照拓扑逆序做dp倒推出初始状态是否为必胜局面。
6. 最优状态博弈

例题：

1. [石子游戏1](https://leetcode-cn.com/problems/stone-game/) [dp法](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/%E7%9F%B3%E5%AD%90%E6%B8%B8%E6%88%8F1-dp) [脑筋急转弯](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/%E7%9F%B3%E5%AD%90%E6%B8%B8%E6%88%8F1-%E8%84%91%E7%AD%8B%E6%80%A5%E8%BD%AC%E5%BC%AF)
2. [石子游戏2](https://leetcode-cn.com/problems/stone-game-ii/submissions/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/%E7%9F%B3%E5%AD%90%E6%B8%B8%E6%88%8F2-dp)
3. [石子游戏3](https://leetcode-cn.com/problems/stone-game-iii/submissions/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/%E7%9F%B3%E5%AD%90%E6%B8%B8%E6%88%8F3-dp)
4. [石子游戏4](https://leetcode-cn.com/problems/stone-game-iv/submissions/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/%E7%9F%B3%E5%AD%90%E6%B8%B8%E6%88%8F4-dp)
5. [石子游戏5](https://leetcode-cn.com/problems/stone-game-v/submissions/) [正向dp](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/%E7%9F%B3%E5%AD%90%E6%B8%B8%E6%88%8F5-dp) [记忆化搜索](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/%E7%9F%B3%E5%AD%90%E6%B8%B8%E6%88%8F5-%E8%AE%B0%E5%BF%86%E5%8C%96%E6%90%9C%E7%B4%A2)
6. [石子游戏6](https://leetcode-cn.com/problems/stone-game-vi/) [贪心](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/%E7%9F%B3%E5%AD%90%E6%B8%B8%E6%88%8F6-%E8%B4%AA%E5%BF%83)
7. [石子游戏7](https://leetcode-cn.com/problems/stone-game-vii/submissions/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/%E7%9F%B3%E5%AD%90%E6%B8%B8%E6%88%8F7-dp)
8. [石子游戏8](https://leetcode-cn.com/problems/stone-game-viii/) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E5%AD%A6%E7%9F%A5%E8%AF%86/%E5%8D%9A%E5%BC%88%E8%AE%BA/%E7%9F%B3%E5%AD%90%E6%B8%B8%E6%88%8F8-dp)

**总结：**

**通过上面一系列例题我们可以总结出来这样几个规律。**

1. **博弈论的最优化问题通常是通过动态规划求解的。动态规划中，dp数组不具体指某一个人的收益，而是任何一个当前玩家在当前状态下可以获得的最大收益，这样对于先手操作的人，初始状态对饮的dp数组的值就是我们要求解的目标值。**
2. **想要求解此类博弈问题，我们最重要的是明白问题怎么定义，以及如何进行递推。石子游戏是一个典型的问题，直观地讲我们很难按照正向的方式去思考求解出最优解。因为最优博弈过程可能在前面一些步骤并不是最优的，类似于强化学习的原理，我们要的是在长期动作累计的情况下，最终结果的最优性。这实际上告诉我们，实际上最优博弈的过程是后边的步骤影响前面的操作，而非由前面的操作决定后面的操作。**
3. **我们要求解此类问题，最重要的是搞清楚什么是子问题，什么是父问题。此时我们就可以通过子问题递推父问题的方式来求解。回想一下石子问题1，最大的父问题是长度为n的区间中当前玩家获得的最大收益，而最小的子问题可能是任意长度为1的区间的玩家获得的总收益。最开始时我们并不知道最优博弈会到哪一个终点，但是我们可以假设最开始时任意一个区间都可能是最优博弈的最后一步，那么我们就可以以此为基准，以及求解出更大区间的子问题对应的最优解。可以看出博弈论中的子问题实际上就是更大的博弈问题操作几步后的问题。**
4. **确定父子问题更无脑一点的想法是，最大的父问题往往是博弈的初始状态对应的问题，例如石子游戏2中，博弈的最初始状态就是位于第一堆石子，M=1时博弈的最优价值。假设其那面的部分已经被取完，那么从任意一堆石子开始，M=j时的状态就是一个子问题，而子问题递推过程实际上是从后往前递推的。**
5. **博弈论的问题我们用记忆化搜索往往比较容易实现，这是因为这本身就与博弈论的思路契合，但是从记忆化搜索我们也可以反推出递推方式的动态规划过程。**
6. 总结

# 数据结构进阶

1. 并查集

并查集实际上是一个孩子节点指向父节点的有向树结构，这使得我们可以通过孩子节点不断查找父节点的过程来实现快速确定根节点的操作，如果我们将根节点选择为这个集合的代表元素，那么我们可以快速确定一个元素属于那个集合。

并查集有两个主要的优化：

1. 按秩合并

并查集实际上是一个孩子节点指向父亲节点的树，我们查询的时候根据连接关系查找这个树的根节点，这样树的深度越低，那么并查集查询时间复杂度越小，基于这一理论，我们保存每个节点的深度，在合并时将深度作为孩子节点指向深度深的父亲节点。

1. 路径压缩

不合理的树形结构会导致树形结构接近于一个线型结构，基于这一考虑，我们在查询的过程中，将所有节点的父节点动态地更改为这个树形结构的根节点，那么在后续的相同查询中，查询的时间复杂度就会变得很低。

例题：

[模板题](https://www.luogu.com.cn/problem/P3367) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84%E8%BF%9B%E9%98%B6/%E5%B9%B6%E6%9F%A5%E9%9B%86/%E6%A8%A1%E6%9D%BF%E9%A2%98.cpp)

1. 树状数组

树状数组的思想使用数组来模拟二叉树的结构，该数据结构巧妙地利用了数据的二进制分解的技巧，来使得每个二进制数对应的数组位置保存一个与该二进制数本身相关的数据。二进制数对应的数组位置，与存储的数据有什么关系呢？以二进制数11000为例，该位置存储所有小于等于11000，大于10000区间内的对应位置上所有数据的和。值得注意的是，实际上每个位置本身只对应一个值，但是为了有效地利用信息，我们通过这种错位存储的方式，来尽可能多地保存了有效信息，而每个位置本身对应的值，我们通过一个两个查询之间的差值来确定。

树状数组的主要操作技巧就是对二进制数的一些操作，更具二进制位的特点，每个位置存储了不同位数的和，例如我们在查询前10011个数的和时，可以利用10011,10010,以及10000三个位置保存的值得和来确定，因为100011只保存了它自己位置的值，10010保存了它自己以及前一位的和，而10000保存了所有小于等于10000位置的和，将这三个位置的值加起来就是前10011个位置的和。可以证明的是，这种查询的时间复杂度不超过O(logn)，但是该结构可以快速通过类似于前缀和的思想来获得区间的和，同时可以在O(logn)的时间内更新某一个位置的值。具体对应的结构我们用下面的图来表示：

图示

描述已自动生成

上图中，每个二进制数对应的位置保存了所有子树中元素的总和，上面的树形结构实际上并不存在，只是为了便于理解虚拟出来的图，实际上真正的结构就是最下面一层的数组，我们只需要将标有二进制数的节点中应该保存的数放到真实数组对应的位置即可。

* 1. 树状数组单点修改

[模板题](https://www.luogu.com.cn/problem/P3374) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84%E8%BF%9B%E9%98%B6/%E6%A0%91%E7%8A%B6%E6%95%B0%E7%BB%84/%E6%A8%A1%E6%9D%BF%E9%A2%98)

* 1. 树状数组区间修改

1. 线段树

线段树的原理与堆的思想类似，是对二进制数对应的二叉树分解方式的机制利用，不同于树状数组，线段树使用了额外的空间来真正模拟了一个二叉树，在这个二叉树中，用完全二叉树编号对应的数组位置的方式来确定父子节点之间的关系，这样我们就可以通过二叉树查询的方式来修改很更新这个结构。线段树的原理其实比较简单，但是线段树的模板往往比较大，通常使用递归方式的模板已然能够满足大多数问题，所有后续的代码中，线段树的实现方式也是递归方式。

例题：

1. [模板题](https://www.luogu.com.cn/problem/P3372) [答案](https://github.com/run916/Algorithms/blob/main/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84%E8%BF%9B%E9%98%B6/%E7%BA%BF%E6%AE%B5%E6%A0%91/%E6%A8%A1%E6%9D%BF%E9%A2%98.cpp)
2. 分块
3. 二叉搜索树与平衡树初步
4. 离线分治算法
5. 可持久化数据结构
6. 总结

# 动态规划

1. 思想

在具体介绍动态规划之前我先介绍一下动态规划的思想，以及遇到一个问题我们该如何思考，得到动态规划的递推顺序以及状态转移方程。

动态规划实际上是对搜索问题的剪枝，我们在搜索那一节中提到的记忆化搜索就是动态规划的思想。首先介绍一些书上对于动态规划的抽象性描述。

**能够使用动态规划的问题，具有三个重要的特点，①问题可以划分成等价的子问题，同时子问题必须有重叠性，否则搜索过程中用空间换时间就没有意义，因为对于每个问题来说，它的子问题都需要单独计算，因为没有重叠性。②递归过程无后效性。③最优子结构性质。**

**①问题可以划分成子问题，是可以使用动态规划技术的基本要求，因为动态规划的本质是将搜索过程中的中间态进行存储，用时间换空间的技术。如果问题不可划分，且子问题不重叠，那么dp数组也就无用了，这样动态规划实际上就是无效的。**

**②动态规划递推过程要求无后效性，所谓无后效性就是递推关系形成的图是一个有向无环图。怎么来看待递推关系形成的图呢？如果我们把搜索过程中的每一个数据稳定的中间状态看做是一个节点，那么这些节点之间的搜索关系的反向关系构成的图就是一个有向无环图。**

**举个例子，对于最为经典的斐波那契数列来说，如果我们用搜索的方式进行求解，那么我们是从n开始搜索的，而每个一个数据稳定的状态我们可以看做是某一个递归栈内保存的数据，也即每个递推体里的状态，对于斐波那契数列问题来说，每个递归体里面稳定的数据只有传入参数n，所以我们可以用dp[i]来记录i=n时这个递归体里的状态。**

**状态之间的关系其实就是递归调用的关系，我们发现对于dp[n]这个状态来说，它在递归函数体内调用了dp[n-1]以及dp[n-2]，所以dp[n]有对dp[n-1]和dp[n-2]的搜索关系，反过来将dp[n-1]和dp[n-2]到dp[n]就有了子问题到父问题的递推关系，也即我们上面描述的搜索关系的反向关系，可以发现这个例子递推关系形成的图是一个有向无环图。**

**那么我们该如何递推呢？根据我们上面的描述，要递推出原始问题的解，就要从它的子问题进行倒推。这样实际上动态规划的过程，就是我们是从最初的子问题开始逐步递推出每个更大子问题的过程。而这个递推顺序实际上就是有向无环图的拓扑序。**

**动态规划的第二个条件是“最优子结构性质”，因为动态规划总是被用来求解最优解，那么我们在递推的过程中要保证可以从子问题的最优解地推出更大子问题的最优解，否则也不能使用动态规划。对于最优性问题来说，我们在表述其状态时要做到不漏即可，而对于计数类动态规划问题，我们要保证不重不漏。**

**动态规划我自己总结为三步：**

**中间态表示：所谓中间态表示，就是用一个dp数组来表示记录每一个数据稳定的状态的过程。值得注意的是这里的中间态不是针对结果数据规模的中间态，而是分析动态规划问题时，由子问题推导到结果问题时的各个中间态。例如在斐波那契数列这个例子中，中间态指的是有规模为0,1的子问题推导出规模为n的问题时的每一个中间态k，其中k∈[2,n]。**

**离散值决策：对于每一个状态，它可行的决策可能只有几种，也即从它导出的有向边有多少，我们就需要在这里进行决策。**

**确定递推顺序：得到了递推顺序以后，实际上我们通常可以根据一二两步得到递推顺序，但是有些问题可能需要先预处理数据，然后再做这三步。**

**当然上述三个步骤不一定完全按照描述的一二三步进行思考，但是大部分问题是符合这一特征的**

1. 线性DP
2. 背包问题

背包问题

背包思想的例题：

1. [准时抵达会场的最少跳跃休息次数](https://leetcode-cn.com/problems/minimum-skips-to-arrive-at-meeting-on-time/submissions/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92/%E8%83%8C%E5%8C%85%E9%97%AE%E9%A2%98/%E5%87%86%E6%97%B6%E6%8A%B5%E8%BE%BE%E4%BC%9A%E5%9C%BA%E7%9A%84%E6%9C%80%E5%B0%91%E8%B7%B3%E8%B7%83%E4%BC%91%E6%81%AF%E6%AC%A1%E6%95%B0)
2. 区间DP
3. 树形DP
4. 树形DP是指在树形结构上进行DP的问题，一定要注意是在树形结构中，而不是任意的图结构，所以我们也可以基于这一性质来判断是否可以用树形DP求解该题。
5. 树形DP通常使用DFS遍历的方式进行DP，DP的思路通常是有子树推导出更大的树的DP值，最后得到整个树的DP值，当然我们也可以反其道而行，但是需要我们给子节点传递父节点的信息，不如自底向上的推导。
6. 例题：
7. [有向图中的最大颜色值](https://leetcode-cn.com/problems/largest-color-value-in-a-directed-graph/)(与求解树的直径的问题类似)
8. 环形与后效性处理
9. 状态压缩DP

状态压缩类动态规划问题，往往解决的是数据范围比较小，但是需要枚举子集的问题。这个时候如果我们使用一个数组来表示dp的中间态时，空间复杂度比较大，从而导致时间复杂度也相对较大。

最大权匹配类例题：

1. [两个数组最小的异或值之和](https://leetcode-cn.com/problems/minimum-xor-sum-of-two-arrays/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92/%E7%8A%B6%E6%80%81%E5%8E%8B%E7%BC%A9DP/%E4%B8%A4%E4%B8%AA%E6%95%B0%E7%BB%84%E6%9C%80%E5%B0%8F%E7%9A%84%E5%BC%82%E6%88%96%E5%80%BC%E4%B9%8B%E5%92%8C)
2. 倍增优化DP
3. 数据结构优化DP
4. 单调队列优化DP
5. 斜率优化DP
6. 四边形不等式优化DP
7. 计数类DP
8. 计数类DP是动态规划中特殊的一类，在求解最值的动态规划问题中，子问题之间是可以重叠的，这不会影响最终结果的最优性。但是在计数类DP问题中，我们要保证统计的过程不重不漏。
9. 计数类DP与组合数学的关系比较紧密，我们要对组合数学中的组合数，阶乘非常熟悉。因为组合数之间有很多变换规则，同一个问题不同的思考方式也会带来截然不同的效果，我们用例题来统计分析出一些规律。

例题：

1. [恰有K根木棍可以被看到的排列总数](https://leetcode-cn.com/problems/number-of-ways-to-rearrange-sticks-with-k-sticks-visible/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92/%E8%AE%A1%E6%95%B0%E7%B1%BBDP/%E6%81%B0%E6%9C%89K%E4%B8%AA%E6%9C%A8%E6%A3%8D%E5%8F%AF%E4%BB%A5%E8%A2%AB%E7%9C%8B%E5%88%B0%E7%9A%84%E6%8E%92%E5%88%97%E6%80%BB%E6%95%B0)

**思路：**

本题的思路还是比较有技巧性的，对于没有接触过此类问题的人来说，还是很难在短时间内想到解法。从常规的动态规划思路出发，搜索的中间态是数量为i,可以看到的木棍数为j时的方案数目，也即我们可以用dp[i][j]表示搜索过程中的一个中间态。

那么我们如何决策呢？首先考虑数量为i的木棍，可以看到j个木棍的状态是什么样的。显然这种状态下，最高的木棍一定是最后一个可以被看到的，在它后面的木棍我们就无法看到了。那么dp[i][j]可能由那些状态转化过来呢？根据前面的分析，我们需要按照最高的木棍的位置来分类讨论。

首先，如果最高的木棍在最后一个，那么此时dp[i][j]可以由所有dp[i-1][j-1]的可行方案最后面添加一个高度为i的木棍转化而来。

其次，如果最后一个木棍看不到，那么它的高度只能是[1,i-1]，此时，除了最后一个木棍外，剩余的木棍是一个木棍数量为i-1的子问题，只要该子问题中可以看到的木棍数量是j，那么按照最高木棍一定是最后一个被看到的，而最高的木棍又在木棍数量为i-1的排列中，那么只需要把最后一个木棍加入木棍数量为i-1的子问题的可行解的后面，即可得到一个木棍数量为i的子问题的一个可行解。按照组合数原理我们有dp[i][j]+=dp[i-1][j]\*(i-1)。

综上，根据我们分析得到的两个决策，我们可以得到状态转移方程dp[i][j]=(dp[i-1][j-1]+dp[i-1][j]\*(i-1))。

可以看出以上思路与我们求解组合数时考虑的种种情况相似，所以对于计数类DP问题，我们需要与组合计数的思想结合，才能较快地解决问题。

除了上述思想外，其实还有另一种思路，我们之前的思考中子问题是前i个木棍可以看到j个的方案数的问题，但是实际上我们可以不用定序，而将子问题描述为，i个木棍可以看到j个木棍的方案数，那么对于dp[i][j]来说，任意从i个木棍中挑出i-1个木棍的问题都可以作为更小规模的子问题。根据我们之前的分析，最高的木棍一定是最后一个被看到，那么如果从i个木棍中挑选的木棍包括i，那么剩余的木棍一定不能被看到，那么此时i-1个木棍中必须要有j个被看到，如果高度为i的木棍没有别挑选，那么挑选的i-1个木棍里面只有j-1可以被看到才能满足dp[i][j]的要求。

1. 数位统计类DP
2. 总结

# 图论

1. 最短路
2. Dijkstra算法

Dijkstra用来处理边权非负的的图中的单元最短路。以节点1为例，求解单元最短路的算法流程如下：

1. 初始化dist[1]=0，其余节点的dist设置为无穷大，同时将1加入到优先级队列中。
2. 找出一个未标记的dist[x]的值最小的点，然后标记该点为已经标记的点。
3. 扫描x的所有出边(x,y,z)，如果dist[y]>dist[x]+z，则更新dist[y]，同时将y加入到优先级队列中。（这里采用惰性删除的方式，来避免判断一个节点是否在有限级队列中。）
4. 重复上述步骤，直到队列为空（所有节点已经被标记）

[模板题](https://www.luogu.com.cn/problem/P4779) [模板](https://github.com/run916/Algorithms/blob/main/%E5%9B%BE%E8%AE%BA/%E5%8D%95%E5%85%83%E6%9C%80%E7%9F%AD%E8%B7%AF/dijkstra)

1. SPFA算法

SPFA是Bellman-Ford算法的优化版本，实际上就是优先级队列优化的Bellman-Ford算法。算法流程如下：

1. 建立一个队列，初始时只有一个点。
2. 取出队头元素，扫描它的所有出边(x,y,z)，若dist[y]>dist[x]+z，则更新dist[y]的值，同时如果y不在队列中，则将y加入到队列中。
3. 重复上述步骤直到队列为空。

[模板题](https://www.luogu.com.cn/problem/P3371) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9B%BE%E8%AE%BA/%E5%8D%95%E5%85%83%E6%9C%80%E7%9F%AD%E8%B7%AF/SPFA.cpp)

1. Floyd算法

Floyd算法利用的是DP思想求解的，DP的思想比较简单，但是怎么找到一个重叠子问题可以扩展到全问题的方法才是难点，Floyd的DP思想中，子问题是求解(i,j)之间节点编号不超过k的最短路的长度，最终不断扩大决策层k的值，即可求解出最终答案。

[牛的旅行](https://www.acwing.com/problem/content/description/1127/) [答案](https://github.com/run916/Algorithms/blob/main/%E5%9B%BE%E8%AE%BA/%E5%8D%95%E5%85%83%E6%9C%80%E7%9F%AD%E8%B7%AF/Floyd.cpp)

1. 最小生成树
2. Kruskal算法

Kruskal算法的正确性基于这样一个简单的事实——任意一颗最小生成树种一定包含无向图中权值最小的边。该结论可以扩展到更一般性的情况，也即对于任何MST的生成子图来说，将每个连通分支缩点后的图中依然满足该结论。

[模板题](https://www.luogu.com.cn/problem/P3366) [模板](https://github.com/run916/Algorithms/blob/main/%E5%9B%BE%E8%AE%BA/%E6%9C%80%E5%B0%8F%E7%94%9F%E6%88%90%E6%A0%91/kruskal)

1. Prim算法

Prim算法同样基于上述理论，但是prim算法的时间复杂度是O(n^2)的，所以prim算法比较适合于稠密图。Prim算法可以用二叉堆优化成O(mlogn)的时间复杂度，但是在稠密图上反而时间复杂度可能更高。

[模板题](https://www.luogu.com.cn/problem/P3366) [模板](https://github.com/run916/Algorithms/blob/main/%E5%9B%BE%E8%AE%BA/%E6%9C%80%E5%B0%8F%E7%94%9F%E6%88%90%E6%A0%91/prim-%E5%88%9D%E5%A7%8B%E6%A8%A1%E5%BC%8F) [堆优化](https://github.com/run916/Algorithms/blob/main/%E5%9B%BE%E8%AE%BA/%E6%9C%80%E5%B0%8F%E7%94%9F%E6%88%90%E6%A0%91/prim-%E5%A0%86%E4%BC%98%E5%8C%96)

1. 树的直径与最近公共祖先
2. 基环树
3. 负环与差分约数
4. Tarjan算法与无向图的连通性
5. Tarjan算法与有向图的连通性
6. 二分图的匹配
7. 二分图的覆盖与独立集
8. 网络流

网络流是图论里面一个非常重要的分支，很多图论里面的问题都可以转化为一个网络流问题进行求解。关于网络流的知识，可以查询算法导论，或者图论与网络流这本书学习，这里只对几个比较经典的网络流算法做一个实现以及分析。

1. EK算法

Emonds-Krap算法的思想很简单，主要包括两个流程：

* 寻找一条增广路。
* 更新最大流。

不断重复上面两个流程，直到不能找到一条新的增广路为止，此时最大流就确定了。该算法的主要关键点在于如何寻找增光路，我们可以使用dfs或者bfs的思想寻找，但是dfs寻找时该算法是一个伪多项式算法。所以通常程序上实现时我们都采用bfs寻找增广路的思想来实现。

算法的难点主要有这么几个，首先就是寻找一条增广路，增广路要求流量大于0，所以我们需要用一个数组来记录bfs搜索到的节点的流入流量的最大可行值得大小，该值遵循流量守恒的原则。

第二个难点是如何更新增广路上的边流量，这里我们需要用一个前驱数组记录增广路，同时利用构建邻接链表时的成对变换技巧来更新边的流量。

1. Dinic算法
2. Isap算法
3. 推拉流算法
4. 总结

# C++ STL用法总结

1. **容器：**

### 序列式容器：

1. Vector
2. Array
3. List

### 关联式容器：

1. Set:底层结构式红黑树

红黑树是一种特殊的二叉搜索树，具有以下特点：

1. 根节点为黑色。
2. 不能存在连续两个点都为红色。
3. 任意一条从根节点到叶子节点的路径上黑色节点的数目相等。
4. 所有叶子节点都是黑色的。
5. 树的高度查不超过两倍，这样查询次数为log n级别的。

这里再额外介绍一下set的非精确查找的两种方式：

* 1. Lower\_bound()方法，查找的是大于等于目标值的第一个元素。
  2. Upper\_bound()方法，查找的是大于目标值的第一个元素。
  3. 在使用这两个方法时，我们需要注意判断边界条件，避免超出范围。

[最近的房间](https://leetcode-cn.com/problems/closest-room/submissions/)

对于使用set的问题，除了离散化以外，如果单纯地想要使用平衡二叉树的结构来实现快速插入和删除，并维持数据有序的问题，我们可以用set。

1. Multiset：底层结构是红黑树
2. Map
3. multimap
4. Unordered\_set
5. Unordered\_map
6. **算法：**
7. **迭代器：**
8. **仿函数：**
9. **适配器：**
10. **配置器：**

# 计算智能算法

计算智能算法是一类采用仿生思想设计的算法，相比于传统的数学方法，计算智能方法的有点在于它不需要对问题的数学模型有深的了解，也不需要函数的导数等信息。而是通过择优迭代的过程，利用数据自身属性来获得局部最优解的过程。

在我的理解中，我更愿意将计算只能的方法看做是类似于牛顿法等迭代方法的扩展，但是又和这些方法有所不同，牛顿法要求区间连续可到。而斜率迭代的要求没那么高，但是还是需要知晓曲线的形状。而计算智能类方法连函数表达式这一步骤也省略了，我们只需要知晓问题本身的一些局部最优性的性质即可。

1. 爬山算法

其实就是类似于牛顿迭代法的过程，我们不断地朝着当前最优的方向前进，直到达到局部最优。

1. 模拟退火
2. 遗传算法

遗传算法应该是进化算法里面最为经典的一类算法。它的基本思想其实来源于达尔文进化论的物竞天择、适者生存的理论。遗传算法的优点包括我们在本章一开始描述的优点，此外遗传算法还具有其他一些优点。例如实现简单，思路简单。但是也存在一些缺点，就是我们很难从理论上说明这一算法的理论保证，即使在主观经验上，我们可以想象得到这其实就是一个并行随机，迭代择优的随机化算法。

遗传算法主要思想是，初始时生成一定数量的初始种群（问题的初始随机解），然后通过杂交，变异获得下一代种群。通过已有的一些择优标准，我们删除种群中表现比较差的个体，然后继续下一轮迭代即可。为了使得算法有效运行，告诉迭代，需要我们在每一轮考量生成的后代是否具有代表性，数量是否足够多，同时每一轮择优之后，获胜的后代是否是最优的等问题。

**执行过程：**

1. 从问题的随机解集开始迭代。为了使用计算机求解，我们首先需要将问题的解集编码，参与计算机能够识别的运算。目前的编码方式主要有三种——二进制编码、浮点数编码、符号编码（图灵机理论）。
2. 二进制编码：缺点是对连续函数优化问题，由于其随机性，和表示范围的跨越度可能比较大，在一些高精度问题上达不到最优。
3. 浮点数编码：指的是每个基因值用某一个范围内的一个浮点数来表示。在浮点数编码中，必须保证基因值在给定的区间限制范围内、遗传算法中所使用的交叉、变异等遗传算子也必须保证其运算结果所产生的新个体的基因值在这个区间限制范围内。

主要优点：

1. 使用与在遗传算法中表示范围较大的数。
2. 适用于精度要求较高的遗传算法。
3. 便于较大空间的遗传搜索。
4. 改善了遗传算法的计算复杂性，提高了运算交率。
5. 便于遗传算法与经典优化方法的混合使用。
6. 便于设计针对问题的专门知识的知识型遗传算子。
7. 便于处理复杂的决策变量约束条件。
8. 符号编码法：指的是个体染色体编码串中的基因值取自一个无数值含义，只有代码含义的符号集如{A,B ,C…}

优点：

* 1. 符合有意义技术块编码原则。
  2. 便于在遗传算法中利用所求问题的专门知识。
  3. 便于遗传算法和相关近似算法之间的混合使用。

1. 交叉、变异生成后代种群，也即在当前这些解上执行交叉，和变异的某些操作，生成一些随机的解集。这里我们要保证后代是合理的。
2. 择优。按照适应度函数择优，选择出适应度高的一组个体参与后续的遗传。
3. 末代种群解码后获得一个较优解。

例题：

1. 粒子群算法
2. 蚁群算法
3. 鱼群算法

# 人工智能算法