# 算法概念整理

目录

[算法概念整理 1](#_Toc1188)

[I. 前置基础知识 2](#_Toc12487)

[II. 排序 2](#_Toc30311)

[III. 搜索 4](#_Toc23134)

[IV. 二分/三分 6](#_Toc5921)

[V. 前缀和/差分 6](#_Toc20817)

[VII. 数据结构 8](#_Toc4617)

[VIII. 动态规划（DP）—— dynamic programming 8](#_Toc32763)

[IX. 图论 11](#_Toc27652)

[X. 数论 21](#_Toc29635)

## 前置基础知识

* 1. 复杂度
     1. 基本操作数
     2. 时间复杂度
     3. 空间复杂度
  2. 子问题
  3. （暴力）枚举
     1. 横向划分子问题
  4. 递归&分治
     1. 逐级划分子问题
  5. 贪心策略
     1. 子问题最优解（不可回退，DP可回退）
  6. 模拟

## 排序

* 1. 影响性质
     1. 稳定性
        1. 相等的元素经过排序之后相对顺序是否发生了改变
     2. 复杂度
        1. 一般默认为时间复杂度
        2. 部分会涉及空间复杂度
  2. 冒泡排序
     1. 稳定排序
     2. O(n^2)
  3. 选择排序
     1. 不稳定排序
     2. O(n^2)
  4. 插入排序
     1. 稳定排序
     2. O(n)~O(n^2)
  5. 快速排序
     1. 不稳定排序
     2. O(n \* log(n))~O(n^2)
  6. 归并排序
     1. 稳定排序
     2. 复杂度：
        1. 时间：O(n \* log(n))~O(n^2)
        2. 空间：O(n)
     3. 过程
        1. 检查&分治
           1. 当数组长度为1时，数组有序
           2. 当数组长度>1时，将数组分为两段，分别检查是否有序
           3. 如果有序，则将它们合并为一个有序数组；

否则对不有序的数组重复第 2 条，再合并。

* + - 1. 合并

为保证排序的稳定性

前段首元素小于或等于后段首元素时作为最小值放入有序数组

* 1. 堆排序
     1. 不稳定排序
     2. O(n \* log(n))
     3. 过程：
        1. 建立大顶堆（小顶堆则反向）
        2. 取堆顶值与数组相应位置交换
        3. 维持剩余堆性质
        4. 重复(2)(3)至堆中元素全部取出

## 搜索

* 1. 深度优先搜索(DFS)
     1. 枚举搜索
        1. 概念阐述
           1. 利用递归函数便捷实现暴力枚举
        2. 过程
           1. 将搜索目标分成若干“层”
           2. 每层基于前几层的状态进行决策，直到达到目标状态
     2. 图论搜索

见图论DFS

* 1. 迭代加深搜索(IDS)

Iterative Deepening Search

* + 1. 概念阐述
       1. 每次限制搜索深度的深度优先搜索(DFS)
       2. 对比说明：

# BFS 的基础是一个队列，队列的空间复杂度很大

# 当状态比较多或者单个状态比较大时，BFS 显劣势

# 类似于用 DFS 方式实现的 BFS，其空间复杂度相对较小

# 搜索树分支较多时，每增加一层复杂度会出现指数级爆炸式增长，这时前面重复进行的部分所带来的复杂度几乎可以忽略，因此迭代加深搜索可近似为BFS

* + 1. 过程：
       1. 设定一个较小的深度进行DFS
       2. 若在搜索途中发现答案即可进行回溯并记录路径（答案）
       3. 若没有发现答案则继续搜索直到深度达到设定值
       4. 将深度限制加1，重新DFS
  1. 广度优先搜索(BFS)

见图论BFS

* 1. 双向同时搜索
     1. 概念阐述：
     2. 基本思路：
        + 1. 状态图上的起点和终点同时开始进行搜索（DFS/BFS）
          2. 搜索两端相遇即视为可行解
  2. Meet in the middle
     1. 概念阐述：
        1. 暴力枚举算法优化
     2. 基本思路：
        1. 搜索过程分成两半分别搜索
        2. 将两半的结果合并
  3. A\*(A-star)
     1. ?
     2. Dijkstra/BFS

## 二分/三分

## 前缀和/差分

* 1. 前缀和
     1. 概念阐述：
        1. 数列（集合类）的前n项和
        2. 数据处理方式
        3. 降低查询时间复杂度
     2. 表达式：
        1. 一维

设前缀和数列为pre\_sum[n]

令pre\_sum[0]前有pre\_sum[-1] = 0

设原数列为 org\_array[n]

* + - * 1. 求和

pre\_sum[i] = org\_array[i] + pre\_sum[i-1]

* + - * 1. 解算

org\_array[i] = pre\_sum[i] - pre\_sum[i-1]

* + - 1. 二维

设前缀和数列为pre\_sum[n][m]

设原数列为 org\_array[n][m]

* + - * 1. 求和

pre\_sum[i][j] = org\_array[i][j]

+ pre\_sum[i-1][j] + pre\_sum[i][j-1]

- pre\_sum[i-1][j-1]

* + - * 1. 解算

org\_array[i][j] = pre\_sum[i][j]

- pre\_sum[i-1][j] - pre\_sum[i][j-1]

+ pre\_sum[i-1][j-1]

* + - 1. 多维

基于动态规划(DP)高维前缀和

* 1. 差分
     1. 概念阐述：
        1. 数列的前n项差
        2. 前缀和逆运算
     2. 表达式

设 下标为n的原数组（或其他数据容器）为 a[n]

下标为n的差分项为 b[n]

* + - 1. 求差

b[i] = { a[i], i = 1 }

{ a[i] - a[i-1], i > 1 }

* + - 1. 解算

a[i] = { b[i], i = 1 }

{ a[i-1] + b[i], i > 1 }

1. **字符串**

## 数据结构

* 1. 链表
     1. 概念阐述：
        1. 单向链表/双向链表
        2. 循环链表
  2. 栈
     1. 概念阐述：
        1. 线性数据结构
        2. 后进先出原则(LIFO-Last In First Out)
     2. 实现思路：
        1. 数组模拟栈
        2. C++ STL容器(stack)
        3. Python 列表模拟(list)
  3. 队列
     1. 概念阐述：
        1. 线性数据结构
        2. 先进先出原则(FIFO-First In First Out)
        3. 循环队列
           1. 假溢出
           2. 后继坐标[(index+1) % deque.size]
     2. 实现思路：
        1. 数组模拟队列
        2. C++ STL容器(deque)
        3. Python 双端队列(queue)
  4. 堆
     1. 概念阐述：
  5. 并查集
     1. 概念阐述：
        1. 管理元素所属集合的数据结构
        2. 树（图论）实现
        3. 相关操作
           1. 合并
           2. 启发式合并
           3. 查询
           4. 路径压缩
           5. 删除
           6. 移动
  6. 哈希表（散列表）
     1. 概念阐述：
        1. [key-value]（键值对）储存数据
        2. 键值联系唯一（key指向唯一内存）
        3. 类似高级数组
        4. 哈希函数
        5. 存储冲突
        6. 拉链法（开散列法 open hashing）
        7. 闭散列法
  7. 单调栈
  8. 单调队列
  9. 线段树
  10. 二叉搜索树
  11. 平衡树

## 动态规划（DP）—— dynamic programming

动态规划是通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法

* 1. **原理**
     1. 最优子结构
        1. 证明问题最优解的第一个组成部分是做出一个选择；
        2. 对于一个给定问题，在其可能的第一步选择中，假定你已经知道哪种选择才会得到最优解。
        3. 给定可获得的最优解的选择后，确定这次选择会产生哪些子问题，以及如何最好地刻画子问题空间；
        4. 证明作为构成原问题最优解的组成部分，每个子问题的解就是它本身的最优解。（方法是反证法）
     2. 无后效性
        1. 已经求解的子问题，不会再受到后续决策的影响。
     3. 子问题重叠
        1. 如果有大量的重叠子问题，我们可以用空间将这些子问题的解存储下来，避免重复求解相同的子问题，从而提升效率。
  2. **概念**
     1. 阶段
     2. 状态
     3. 状态转移方程
  3. **基本思路**
     1. 将原问题划分为若干**阶段**，每个阶段对应若干个子问题，提取这些子问题的特征（**状态**）
     2. 寻找每一个状态的可能**决策**，或者说是各状态间的**相互转移方式**（**状态转移方程**）
     3. 按顺序求解每一个阶段的问题。
  4. **记忆化搜索**

记录已经遍历过的状态的信息，来避免对同一状态重复遍历的搜索实现方式

添加记忆化数组

* 1. 0-1**背包**

####

每个物体只有两种可能的状态（取与不取）即对应二进制中的 0 和 1

####

设DP状态f{i,j}

即在只能放前 i 个物品的情况下，容量为 j 的背包所能达到的最大总价值

####

由此可以得出状态转移方程：

f[i][j] = max{ f[i-1][j] , f[i−1][j−w[i]] + v[i] }

改用滚动数组的形式来优化：

f[j] = max{ f[j] , f[j−w[i]] + v[i] }

**大部分背包问题的转移方程都是在此基础上推导出来的！！！**

####

在对第i个物品遍历时，按照背包容量逆序遍历

在对第i个物品进行枚举时，容量从背包总容量j遍历至该物品的占用空间 w[i]，以保证同一物品只被选一次即f[j`] = f[j-w[i]]时，后者会先 被更新。

* 1. **多重背包**

####

0-1背包问题变种

区别在于同种物品有k[i](k[i]>=1)个

####

DP状态定义：

设 f{i,j}

选前 i 个物品时，容量为 j 的背包可以达到的最大价值。

####

状态转移方程（朴素做法）：

f[i][j] = max(f[i-1][j-k\*w[i]]+v[i]\*k) , k∈[0,k[i]]

####

优化?

* 1. **完全背包**

####

完全背包模型与 0-1 背包类似

与 0-1 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次

####

DP状态定义：

设 f{i,j}

选前 i 个物品时，容量为 j 的背包可以达到的最大价值。

####

状态转移方程（朴素做法）：

f[i][j] = max(f[i-1][j-k\*w[i]]+v[i]\*k) , k∈[0,+∞)

滚动数组优化：

f[i][j] = max(f[i-1][j-w[i]])

####

在对第i个物品遍历时，按照背包容量顺序遍历

与0-1背包相反，容量从该物品的占用空间w[i]遍历至背包总容量j，则物品可以被多次选择即f[j`] = f[j-w[i]]时，前者会先被更新

* 1. LCS（最长公共子序列）
  2. LIS（最长递增不下降子序列）

## 图论

* 1. 基本概念
     1. 图(graph) -> G
        1. 概念阐述：

二元组：点集（非空集） + 边集

* + - 1. 组成部分：
         1. 点

又称顶点(vertex)或者节点(node)

多个点组成的集合即为点集(vertex set)->V(G)

|V(G)|称为图G的阶

* + - * 1. 边(edge)->e

多条边组成的集合即为边集(edge set)->E(G)

无向边(undirected edge)

e = (u,v)

u,v均为e的端点(endpoint)

有向边(directed edge)

e = u->v

u为起点(tail)，v为终点(head)（命名参考矢量）

u为v的 直接前驱 ，v为u的 直接后继

* + - 1. 组成联系：
         1. 相邻

边 && 点

若点v是边e的一个端点（无向图G）

则称v,e是相邻的(adjacent)/关联的(incident)

点 && 点

对于两个节点（顶点）u,v 存在边e = (u,v)

则称u,v是相邻的

邻域 (neighborhood)

对任意一个顶点v，其所有相邻的顶点构成的集合

记作 N(v)

* + - * 1. 自环

E(G)中有边e = (u,v)，且u = v，则e为自环

* + - * 1. 重边

E(G)中存在e1 = e2，则e1,e2为（一组）重边

* + - * 1. 度数

度(degree)

# 与一个顶点关联的边数称为该顶点的度

# 对于无向简单图，d(v) = |N(v)|

握手定理/图论基本定理

对于无向简单图G = (V,E)

有 Σd(v) = 2\*|E|,(v∈V)

# 推论：（任意图）度数为奇数的点必然有偶数个

顶点度数

孤立点

叶节点（悬挂点）

偶点

奇点

支配点

度数极值

有向边（矢量）

入度

出度

* + - * 1. 路径

迹

路径

回路

环（圈）

* + - * 1. 子图
        2. 连通
        3. 割
      1. 种类：
         1. 有限图

点集 && 边集 都是有限集合

* + - * 1. 无限图

点集 || 边集 是无限集合

* + - * 1. 有向图

边集为有序二元组

有向无环图(DAG) —— Directed Acyclic Graph

* + - * 1. 无向图

边集为无序二元组

* + - * 1. 混合图
        2. 简单图
        3. 多重图
        4. 稀疏图

一张图的边数 << 其点数的平方

* + - * 1. 稠密图

一张图的边数 ~ 其点数的平方

* 1. 存图方法

设n为图G的点数，m为图G的边数，d+(u)为点u的出度

* + 1. 直接存边
       1. 方法思路：

使用数组储存边信息（起点，终点，边权）

* + - 1. 复杂度;
         1. 查询边（存在）（时间）：

O(m)

* + - * 1. 遍历一个节点u的所有出度（时间）：

O(m)

* + - * 1. 遍历全图（时间）：

O(nm)

* + - * 1. 空间复杂度：

O(m)

* + 1. 邻接矩阵
       1. 方法思路：

使用二维数组adj(adjacent)储存边信息

储存格式：adj[u][v] = value

u（起点），v（终点），value（边权，0即不存在）

* + - 1. 复杂度;
         1. 查询边（存在）（时间）：

O(1)

* + - * 1. 遍历一个节点u的所有出度（时间）：

O(n)

* + - * 1. 遍历全图（时间）：

O(n^2)

* + - * 1. 空间复杂度：

O(n^2)

* + 1. 邻接表
       1. 方法思路：

使用动态容器库（vector等）来储存点信息

储存格式（举例）：

vector<vector<int> > adj ;

adj[u][i] = v

u（起点），v（终点）

* + - 1. 复杂度：
         1. 查询边(u,v)（存在）（时间）：

O(d+(u))

# 若sort则利用二分查找可做到O(log(d+(u)))

* + - * 1. 遍历一个节点u的所有出度（时间）：

O(d+(u))

* + - * 1. 遍历全图（时间）：

O(n+m)

* + - * 1. 空间复杂度：

O(m)

* + 1. 链式前向星
       1. 方法思路：

本质是用链表实现邻接表

* + - 1. 复杂度：
         1. 查询边(u,v)（存在）（时间）：

O(d+(u))

* + - * 1. 遍历一个节点u的所有出度（时间）：

O(d+(u))

* + - * 1. 遍历全图（时间）：

O(n+m)

* + - * 1. 空间复杂度：

O(m)

* 1. 拓扑排序(Topological sorting)
     1. 概念阐述：

给有向无环图所有节点排序

* + 1. 构造：卡恩算法(Kahn)
       1. 概念阐述：
          1. 检查图中是否有环
          2. 同时构造拓扑排序
       2. 时间复杂度：

对图G = (V,E)，有O(E+V)

* + - 1. 过程思路：（维护集合S）

初始状态下，集合S装着所有入度为0的点，L 是一个空列表

每次从S中任取一点u放入L，并将所有以u为起点的边删除

对于边(u, v)，若删边后点v的入度变为0，则将v放入S中

不断重复以上过程，直到集合S为空。

* + - * 1. 检查图中是否存在任何边

如果有，那么该图有环

否则返回L，L中顶点顺序就是拓扑序列

* + 1. 适用图种类：
       1. DAG
          1. 概念阐述：（有向无环图）

# 将图中的顶点以线性方式进行排序

# 排在前面的节点不能依赖于排在后面的节点

# 使对顶点u到v的有向边(u,v)，都有u在v的前面

# 若从i到j有边，则认为j依赖于i

# 若i到j有路径（i可达j），则称j间接依赖于i

* + - 1. AOVN-(Activity On Vertex Network)
         1. 概念阐述：（顶点活动网络）

# AOV 的活动都表示在顶点上

* + - * 1. 组成：

前驱活动：

# 有向边起点的活动称为终点的前驱活动

# （当一个活动的前驱全部完成后，这个活动才能进行）

后继活动：

# 有向边终点的活动称为起点的后继活动

* + - * 1. 建立过程：！
      1. AOEN-(Activity On Edge Network)
         1. 概念阐述：（顶点活动网络）

# 顶点表示事件，弧表示活动持续的时间

* + - * 1. 组成：

活动：

AOE 网中，弧表示活动

弧的权值表示活动持续的时间

事件：

顶点表示事件（可触发）

弧（活动）a\_j 的最早开始时间：

初始点到该弧起点的最长路径长度

记为 e(j)

弧（活动）a\_j 的最迟开始时间：

（在不推迟整个工期的前提下）

工程达到弧起点最晚能容忍的时间

记为 l(j)

顶点（事件）v\_j 的最早发生时间：

初始点到该顶点的最长路径长度

记为 ve(j)，ve(j)=e(j)。

顶点（事件）v\_j 的最迟发生时间：

（在不推迟整个工期的前提下）

工程达到顶点最晚能容忍的时间

记为 vl(j)， l(j)=vl(j)-dul(aj)。

关键路径：

AOE 网中从源点到汇点的最长路径的长度。

关键活动：

关键路径上的活动，最早开始时间和最迟开始时间相等。

……（递推关系）

* + - * 1. 关键路径求解

……

* 1. 遍历&搜索
     1. DFS
        1. 概念阐述
           1. 遍历或搜索树（图）的一种算法
           2. 每次都尝试向更深的节点走（深度优先）
           3. 符合以下两个特征即为广义DFS
           4. 特征一：递归调用自身
           5. 特征二：对访问过的点打上标记（每个点仅访问一次）
        2. 复杂度

n为点数，m为边数

* + - * 1. 时间复杂度：O(n + m)
        2. 空间复杂度：O(n)
    1. BFS
       1. 概念阐述
          1. 遍历或搜索树（图）的一种算法
          2. 每次都尝试访问同一层的节点（广度优先）
       2. 复杂度

n为点数，m为边数

* + - * 1. 时间复杂度：O(n + m)
        2. 空间复杂度：O(n)
  1. 路径问题

## 数论

* 1. 判断一个数是否为2的n次方