大厂学院
ITDACHANG.COM

大厂算法与数据结构解析之

算法基础

讲师：武晟然

大厂学院
ITDACHANG.COM

1 什么是算法

2 算法的复杂度

3 算法的分类

4 一些经典算法

为什么要学习算法

- 程序 = 数据结构 + 算法
- 算法——大厂面试的必备主菜
  - 算法可以衡量程序员的技术功底
  - 算法可以体现程序员的学习能力和成长潜力
  - 学习算法有助于提高分析解决问题的能力
  - 学习算法是做性能优化、成长为架构师的必经之路

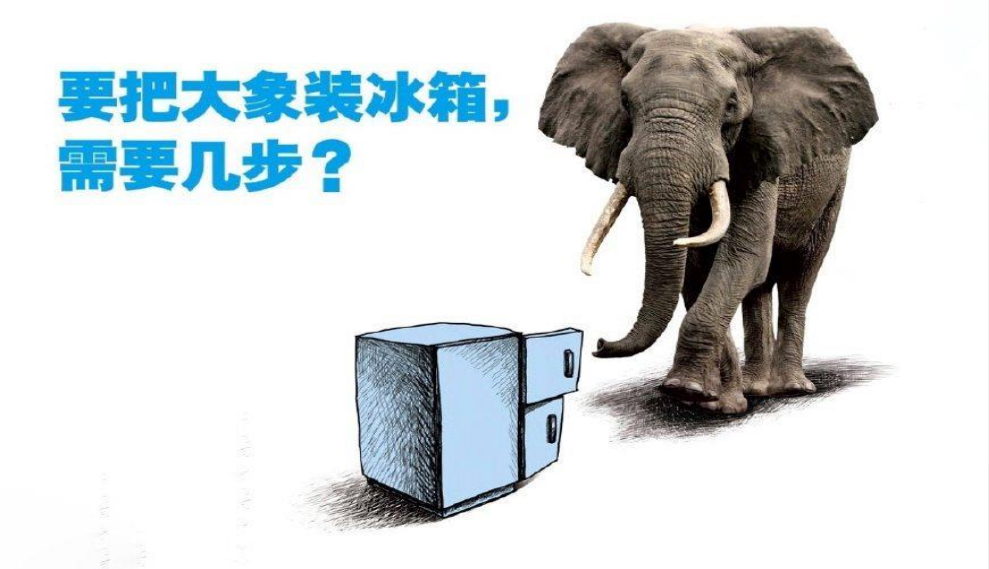
怎样学习算法

- 前置知识
  - 熟练掌握一门编程语言，了解数据结构相关知识
- 分类学习
  - 按照数据结构、应用场景、实现策略
- 学习算法的捷径——刷题
  - 题目来源：LeetCode

什么是算法

- 算法（Algorithm）

  —— 解决问题的步骤和方法



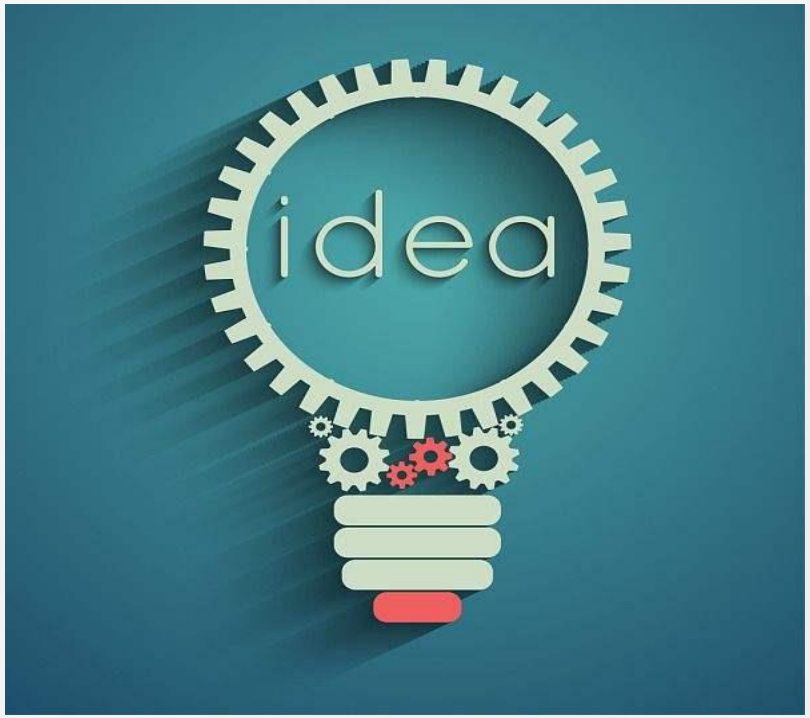
- 算法，是指解题方案的准确而完整的描述，是一系列**解决问题的清晰指令**，算法代表着用系统的方法描述解决问题的策略机制。也就是说，能够对一定规范的**输入**，在**有限时间**内获得所要求的**输出**。

算法的特性

- 有穷性（Finiteness）
  - 算法必须能在执行有限个步骤之后终止；
- 确切性（Definiteness）
  - 算法的每一步骤必须有确切的定义；
- 输入项（Input）
  - 一个算法有0个或多个输入，以描述运算对象的初始情况
- 输出项（Output）
  - 一个算法有一个或多个输出，以反映对输入数据加工后的结果
- 可行性（Effectiveness）
  - 算法中每个计算步骤都可以在有限时间内完成（也称之为有效性）

算法的复杂度

- 如何分析算法的优劣
  - 求和：1 + 2+ 3 + ... + 100
- 时间复杂度
  - 执行算法所需要的计算工作量
- 空间复杂度
  - 算法需要消耗的内存空间



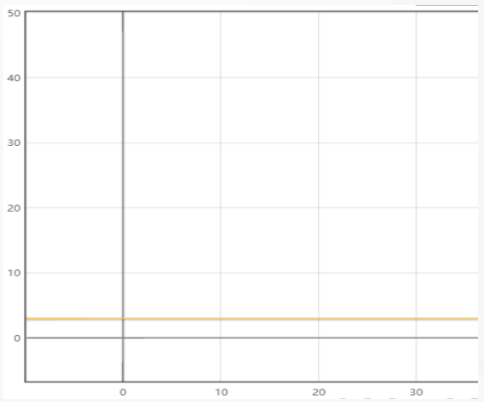
算法的时间复杂度

- 我们考虑的问题
  - 不同的机器，计算速度不同
  - 如果我们的计算机无限快，那只要确定算法能够终止就可以，所有算法性能没有比较的必要了
  - 如果输入数据量很小，那一般机器也都可以在非常短的时间内完成，算法性能也没有必要比较了
  - 计算资源有限，输入数据量很大的情况下，不同算法耗费的时间差异极大
- 基本指令——程序执行消耗的时间单位
  - 算术指令、数据移动指令、控制指令
  - int a = 1; 运行时间 1
  - if (a > 1) {} 运行时间 1
  - for (int i = 0; i < N; i++) { System.out.println(i); } 运行时间 3N+2

时间复杂度

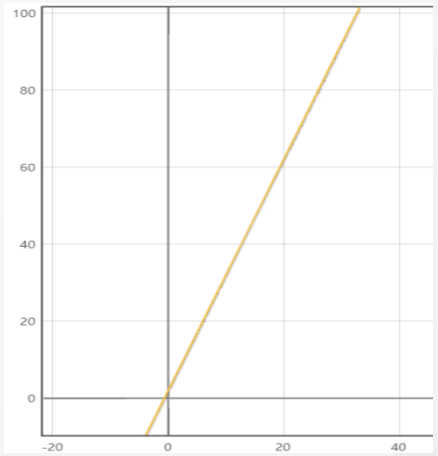
- 不同的算法，运行时间 T(n) 随着输入规模 n 的增长速度，是不同的

```
int a = 1;
a = 2;
a = 3;
```



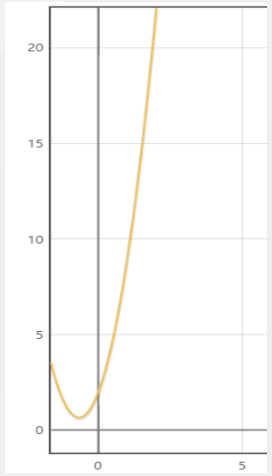
T(n) = 3

```
for (int i = 0; i < n; i++) {
    System.out.println(“test");
}
```



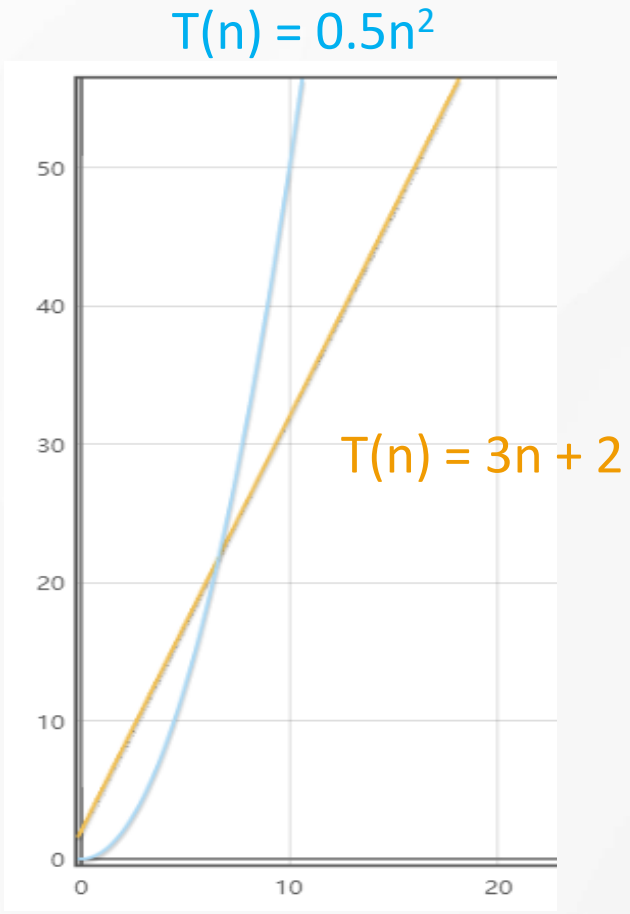
T(n) = 3n + 2

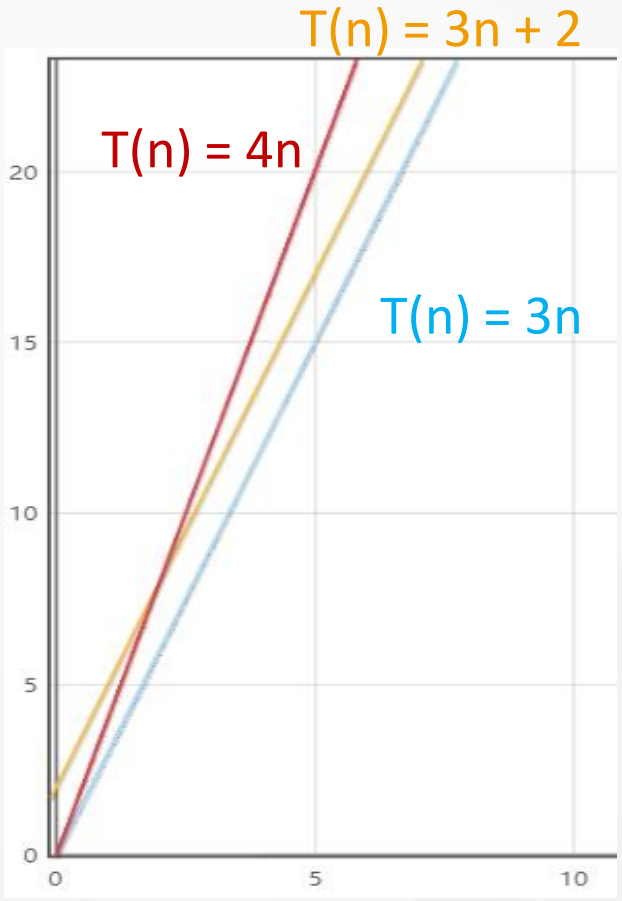
```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        System.out.println(“test");
}
```



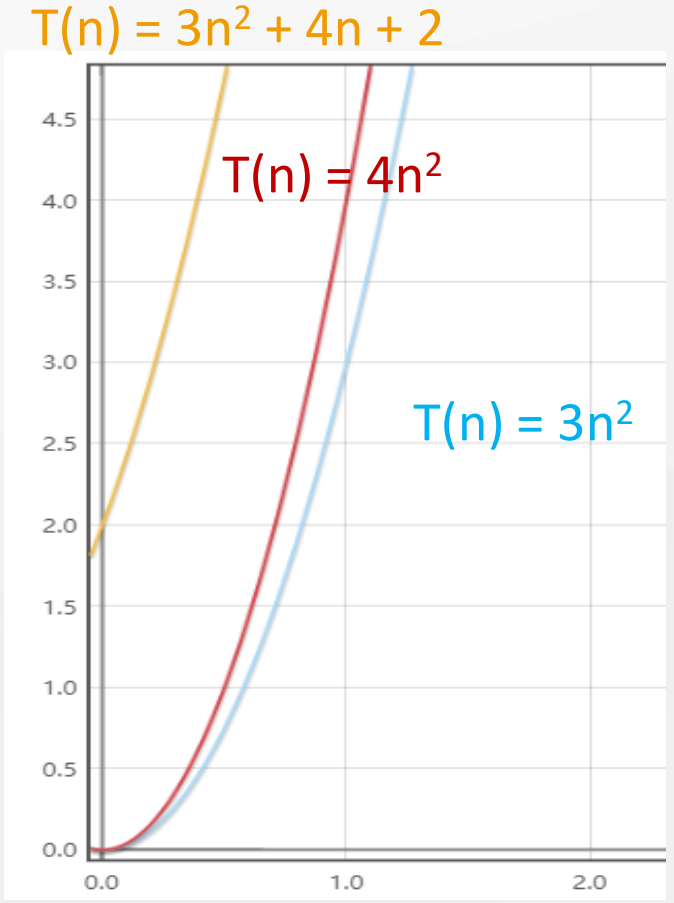
T(n) = $3n^2 + 4n + 2$

复杂度的大0表示法





T(n) = O(n)



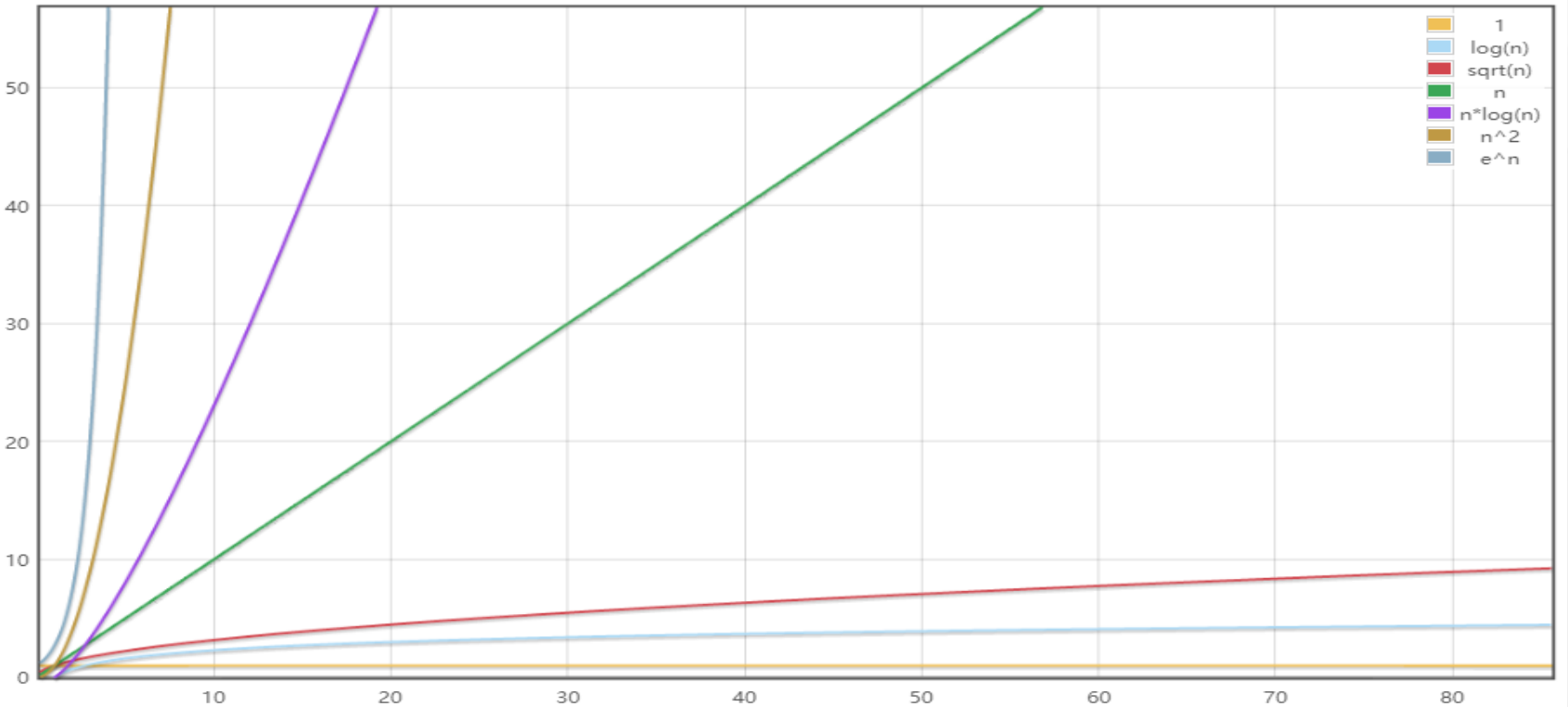
T(n) = O(n²)

复杂度的大0表示法

- 对于给定的函数g(n)，用O(g(n))来表示以下函数的集合：
  - O(g(n)) = { f(n): 存在正常量 c 和 n_0，使对所有 $n \geqslant n_0$ ，有 $0 \leqslant f(n) \leqslant cg(n)$ }
- 算法分析中，一般用大O符号来表示函数的 **渐进上界**
- 这表示，当数据量达到一定程度时，g(n) 的增长速度不会超过 O(g(n)) 限定的范围

常见的算法复杂度

- O(1)，O(log(n))，O($\sqrt{n}$)，O(n)，O(nlog(n))，O(n^2)，O(e^n)



空间复杂度

- 算法执行所占用的空间
  - Array[100]： O(100)
  - Array[N]： O(N)
  - int val = 1; 空间复杂度 O(1)
- 有时候递归调用，需要计算调用栈所占用的空间

算法的分类

按照应用目的

- 搜索算法
- 排序算法
- 字符串算法
- 图算法
- 最优化算法
- 数学算法

按照实现策略

- 暴力法
- 增量法
- 分治法
- 贪心算法
- 动态规划法
- 回溯法
- 分支限界法

分治算法

- 分而治之
  - 问题的规模越小，越容易解决
  - 把复杂问题不断分成多个相同或相似的子问题，直到每个子问题可以简单地进行求解
  - 将所有子问题的解合并起来，就是原问题的解
- 分治和递归
  - 产生的子问题形式往往和原问题相同，只是原问题的较小规模表达
  - 使用递归手段求解子问题，可以很容易地将子问题的解合并，得到原问题的解

分治算法

- 基本步骤
  - step1：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题
  - step2：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题
  - step3：将各个子问题的解合并为原问题的解
- 应用场景
  - 二分搜索、大整数乘法、归并排序、快速排序
  - 棋盘覆盖问题、循环赛日程表问题、汉诺塔问题等

贪心算法

- 贪心（Greedy）—— 总是做出在当前看来是最好的选择
  - 不从整体考虑，只考虑眼前，得到 **局部最优解**
- 局部最优解和全局最优解
  - 要保证最终得到的是全局最优解，贪心策略必须具备 **无后效性**
- 适用场景
  - 用贪心算法直接求解全局最优，条件比较苛刻
  - 哈夫曼（Huffman）编码

贪心算法

- 具体实现框架

从问题的某一初始解出发；

while （能朝给定总目标前进一步）

{

利用可行的决策，求出可行解的一个解元素；

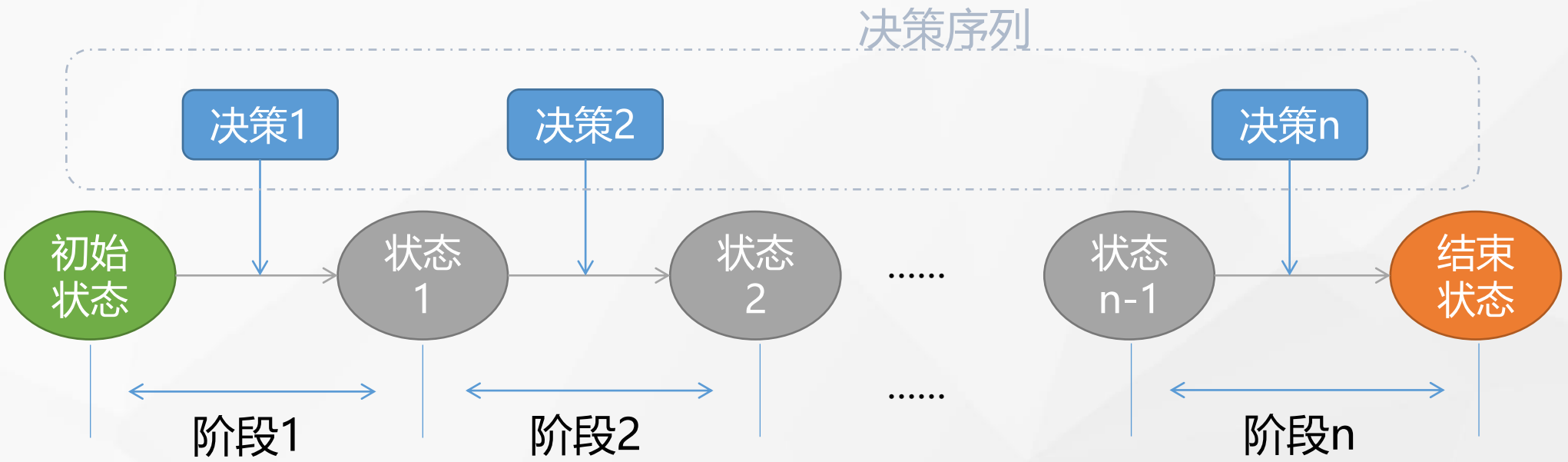
}

由所有解元素组合成问题的一个可行解；

动态规划

- 动态决策的过程
  - 把原问题划分成多个“阶段”，依次来做“决策”，得到当前的局部解
  - 每次决策依赖于当前的“状态”，随即引起状态的转移
  - 一个决策序列就是在变化的状态中产生出来的
  - 这种多阶段决策最优化，解决问题的过程就称为 **动态规划**
- 最优化问题
  - 动态规划通常用来求解最优化问题
  - 可以有很多可行解，每个解都对应一个值，我们希望找到具有最优值的解

动态规划



- 应用场景
  - 最优二叉搜索树、最长公共子序列、背包问题等等

回溯法

- 选优搜索法
  - 按照一定的选优条件，不停向前搜索，直到达到目标
  - 如果搜索到某一步，发现之前的选择并不优，就退回一步重新选择
  - 通用解题方法
- 深度优先搜索（DFS）策略
  - 在包含问题所有解的解空间树中，按照深度优先搜索的策略，从根结点出发、深度搜索解空间树
  - 回溯法就是对隐式图的深度优先搜索算法

分支限界法

- 广度优先搜索（BFS）策略
  - 所谓"分支"，就是采用广度优先的策略，依次搜索当前节点的所有分支
  - 抛弃不满足约束条件的相邻节点，其余节点加入"活节点表"
  - 然后从表中选择一个节点作为下一个扩展节点，继续搜索
- 限界策略
  - 为了加速搜索的进程，一般会在每一个活节点处，计算一个函数值
  - 根据这些已计算出的函数值，从当前活节点表中选择一个最有利的节点作为扩展节点，使搜索朝着解空间树上有最优解的分支推进，以便尽快地找出一个最优解

回溯法 vs 分支限界法

	回溯法	分支限界法
对解空间树的搜索方式	DFS	BFS
存储节点常用数据结构	堆栈	队列
应用场景	找出满足约束条件的所有解；找出全局最优解	找出满足约束条件的一个解；找出局部最优解

一些经典算法

- 二分查找
- 快速排序、归并排序
- KMP算法
- 快慢指针（双指针法）
- 普利姆（Prim）和 克鲁斯卡尔（Kruskal）算法
- 迪克斯特拉（Dijkstra）算法
- 其它优化算法：模拟退火、蚁群、遗传算法

谢 谢 观 看