PPT 内容总结

- L01-环路检测
  - 算法
  - 数据结构
- L02-约瑟夫环
  - 数组和链表
  - 递归
  - 函数调用时的机制
- L03-两数之和
  - 两数之和问题
  - 哈希
  - 时空复杂度
  - 排序
- L04-表达式求值
- L05-part1-最大子数组和
  - 最大区间和问题
  - 分治
- L06-公共祖先
  - 寻找 LCA 的算法
  - BST 操作
  - 遍历树
  - AVL
- L07-树的扩展
  - B 树
  - 2-3 树
  - 红黑树
  - B+ 树
- L08-农羊狼菜
  - DFS/BFS 解决问题和剪枝策略
  - 状态机
  - 搜索策略
- L09-优化问题
  - 贪心解法的案例
  - 贪心 DP 对比
  - 旅行商问题解法
  - 分支限界法(Branch and Bound)
- L10-六度空间
  - 图的存储

- DFS 的应用
- BFS 的应用
- 求最小生成树
- SSSP 问题
- APSP 问题

L12-NP问题

- 优化问题和判定问题
- P,NP,NP-hard 和 NP-complete 问题

PPT 问题解答/部分内容的详述

L02-约瑟夫环

- 42

L05-part1-最大子数组和

- 7
- 13

L06-公共祖先

- 4
- 21
- 26
- AVL rotation 促进平衡的证明

L08-农羊狼菜

- 18
- 19
- 43

L10-六度空间

- 10
- 13
- 14

L12-NP问题

- 图灵机
- 多项式时间归约

> “PPT 内容总结”尽量涵盖了所有内容，但过于平凡或者单纯列举的不展开讲，用 [·] 中括号概述。如果真的要看这些，容易在相应 PPT 中找到。
>
> “PPT 问题解答/部分内容的详述” 详细讲了一些细节，但考试不一定那么深入，~~虽说我写的也并不深入~~

PPT 内容总结

Lo1-环路检测

算法

> 对于实际问题，将其建模为形式化的模型，再应用算法得到求解的程序。

算法是一些逻辑层面上的指令，而程序是对其使用代码来实现的执行。[算法和程序的各项差异、常见的算法例子]

衡量算法的方式：

- 实验分析，使用不同的输入来看运行时间。依赖HW/OS，难以准确预测。
- 理论分析，考虑所有输入，在算法执行上分析。无关于HW/OS.

数据结构

数据结构是高效地组织数据的多种方式，e.g. 使其便于存储、管理、获取 [数据结构的例子]

数据结构处理数据如何被实际存储和组织，以及支持各个操作的算法实现。
而在接口层面，关心的是能存储什么数据、能用什么操作、需要什么参数

数据结构做到了封装，让用户只关注后者的"做什么"而不必在意"怎么做"、在背后实现改变时也不影响用户使用、防止不封装时用户将数据破坏。
它的优势还有：

- 把复杂系统分解成便于处理的部分
- 可复用性
- 便于维护和 debug

Lo2-约瑟夫环

数组和链表

数组，可以连续地存储任意类型数据，每个元素都有索引。
它的插入、遍历都是 O(n)；对于 2D 的，插入列、插入行都是 O(m·n)。由于只需要遍历的指针/交换临时变量/不需要额外空间，这些操作的空间复杂度 $O(1)$.

数组的局限是尺寸固定，并且在存储中是连续的。而实际上对数量的需求会发生变化，并且对内存的使用也应该更高效，链表解决了这些问题。[链表的类型和基本操作]

链表便于扩容，数组也可以，比如动态数组（Python 的 `list` ）在大小为 1,2,4 等时候扩容为两倍，并且拷贝前一半。而插入 n 个元素时总的用于拷贝的成本，均摊代价是 O(n)：

$$\sum_{i=0}^{\log_2 n} 2^i = 2^{\log_2 n+1} - 1 = 2n - 1 = O(n)$$

递归

递归是在不同输入下多次进行相同操作、逐渐把问题规模缩小，直到达到终止条件。它在一些数据结构（树、图）和算法（分治、贪心、动规）中常用，能让我们不断分解问题到更小规模。
写递归要处理终止条件、递归过程、边界输入情况。[一些递归的例子、递归的种类]

由于需要栈内存和对栈内存的操作，递归相比迭代有更高的时空复杂度而不安全性（栈溢出）。
当我们不在意额外的时空开销，或者只是要一个快速的可行解时，可以使用。此外，在问题本身很适合分解成更小子问题，或者一些如配合记忆化使用的场景里，也适合使用。

函数调用时的机制

- Passing Control（传递控制权）

  函数调用时，CPU 的执行流程要"跳转"。调用某函数时，CPU 跳转到该函数代码的起始地址；函数执行完 return 时，CPU 跳回执行调用函数的下一行继续执行。这种 CPU 要记住"从哪里来，回哪里去"就是返回地址压栈的原因

- Passing Data（传递数据）

  调用函数需要传参数和返回值。

- Memory Management（内存管理）

  函数的局部变量需要临时内存。比如函数内部有局部变量，在调用时会在栈上分配空间来存储，在

执行 return 后都释放。

这些机制都是由机器指令实现的。实际使用时，处理器很节约，用不到的机制就不用，比如一个函数没有局部变量，就不分配栈空间。
机器指令如，`call` 跳转到函数入口地址 + 压返回地址；`ret` 弹出返回地址 + 跳回。
机器指令提供了实现这些机制的能力，但具体怎么用（比如参数放寄存器还是栈）是由设计者规定的规范，这套规范就叫 ABI.

有关函数调用时栈的细节，见这里

Lo3-两数之和

两数之和问题

在两数之和问题中，

- 纯暴力方法，直接双循环枚举，时间复杂度 $O(n^2)$，不额外消耗空间；
- 哈希表方法，记录有某数值的元素的下标，每次判断补数是否存在于之前的哈希中，时空复杂度都为 $O(n)$.
- 排序加双指针法，排序后两侧移动指针，每次判断是否是目标数值，在已经排好序的情况下，时空复杂度都为 $O(n)$.

哈希

哈希是一种分配和索引数据的方法，通常期望能把大量数据索引到固定大小的空间。
哈希表通过 hashing function 来把输入的 key 转化成索引值。一个好的哈希函数应该计算轻量、把 key 均匀地索引、避免冲突。[哈希的术语和常见哈希函数]

使用 hash 的案例：

- Password verification. 登录时比较密码哈希的结果和注册时的哈希结果（fingerprint）。
  btw，avalanche effect 指的是，微小的密码变动会导致哈希结果显著变化。
- File Systems location. 对于一个文件，计算哈希并用哈希值构建路径。

对于哈希冲突，一些解决方法：

- 直接链接：同一个位置挂一条链表，冲突的元素都串在链表上。

  > 这样做，哈希表永远不会满，但是在链太长时会有性能退化。
  >
  > > 下面的开放寻址的实现，元素删除时比较麻烦（遇到空，不知道是不存在还是需要继续跳），需要特殊处理，而链表的删除比较方便。所以，在删除频繁的时候，通常使用它。

- 开放寻址：冲突了以后，在数组里继续找下一个空位

  > 找下一个空位的方法：
  >
  > - 线性探测，冲突了就往后找最近的空格
  > - 双重哈希，冲突时用第二个哈希函数算出"跳几步"来找空位
  >
  > 在表满了的时候，创建一个二倍大小的新表。但是，复制已有元素需要 $O(n)$ 的时间复杂度。在提前知道数据量大小的情况下，扩容不会触发，我们通常使用它。

时空复杂度

[记号$\boldsymbol{O}$(·)、 Ω(·)、Θ(·) 的定义、常见的时间复杂度类型和例子]

哈希的搜索、插入和删除都是平均 $O(1)$ 时间复杂度，只有在哈希函数冲突严重的时候会有 $O(n)$.

排序

[In/Out Place、稳定/不稳定 排序的定义]

一些排序算法：

- 冒泡：比较相邻元素，大小关系错了就交换
- 选择：一直在无序部分中选择最小的，放进有序部分
- 插入：一直在无需部分中取出元素，放进有序部分中的合适位置
- 桶：创建桶来记录各个值的元素的数量，结束后按值的顺序遍历桶
- 计数：与桶排序类似，确认最大最小值以后，记录每个值出现的次数
- 归并：把序列平分，分别调用自己来排序，再合并两个有序序列
- 快速：选择一个分点，把序列分成大于/小于这个分点的两部分，分别调用自己来排序，再直接合并
- 基数：把每个数字拆分成若干位（个位、十位...），从最低/高位开始，依次按每一位的值进行稳定的排序（比如计数）再收集回来，重复这个过程直到处理完所有位。

堆排序

二叉堆是完全二叉树，可能是大/小根堆。大根堆的根节点是全堆中最大的，并且递归地，对于每个节点为根的子堆也是如此。
作为完全二叉树，可以直接计算孩子/父亲的节点索引。

- 堆的 max_heapify 操作：假设一个节点的左右子树都已经是最大堆，但是节点本身可能违反堆性质时，可以 $O(\log n)$ 地恢复最大堆性质。
  做法是，把节点和其左右子节点中更大的那个交换顺序，再递归地处理交换后的该节点，直到某次比左右节点都大。
- 建堆操作：先直接把数组填进完全二叉树中，再从 $\frac{n}{2}$ 到 1 逐个进行 max_heapify.

这就给出了堆排序的流程：先建堆，再每次把（值最大/小的）根节点与最后一个叶子节点交换并从堆中拿掉（放进有序序列），之后对新的根节点进行 max_heapify.

在流式数据中维护 top-k，使用小根堆是非常高效的操作。

[所有排序的时空复杂度、best/worst case 的时空复杂度、是否稳定]

L04-表达式求值

[用栈计算中缀表达式、中缀转后缀代码；栈的常见用途；数学归纳法和强数学归纳法的定义]

L05-part1-最大子数组和

最大区间和问题

- 暴力法（以及其前缀和优化）都是 $O(n^2)$ 时间复杂度；
- 计算以某个元素结尾的最大区间和：要么是只取它自己（前面的和是负贡献不如重新开始），要么是接上前面的最大子数组和。时间复杂度 $O(n)$；
- 递归求出左、右半部分的最大子数组和，再考虑跨越中点的最大子数组和，去最大者。时间复杂度 $O(n \log n)$

分治

分治递归地把问题分解为相似的子问题，直到它们简单到可以直接解决，再把子问题的结果合并得到最终解。[分治的例子]

主定理的结论：

1. 对于 $T(n) = aT(n-b) + f(n), f(n) = n^c$ ：
   - a>1 时 $T(n) = O(a^{n/b} f(n))$
   - a=1 时 $T(n) = O(nf(n))$
   - a<1 时 $T(n) = O(f(n))$
2. 对于 $T(n) = aT(n/b) + f(n), f(n) = n^c$ ：
   - $\log_b a >> c$ 时，$T(n) = O(n^{\log_b a})$
   - $\log_b a \approx c$ 时，$T(n) = O(n^{\log_b a} \log n)$
   - $\log_b a << c$ 时，$T(n) = O(f(n))$

Lo6-公共祖先

寻找 LCA 的算法

朴素的做法是先将深度对齐，再一起向上跳。每次查询 $O(n)$.
LCA 倍增法的做法与之本质上一致，但是提前计算和存储了每个节点向上跳 2^k 步到达的祖先（这个过程时空复杂度 $O(n \log n)$），两个过程都可以二进制拆分步数，单次查找 $O(\log n)$.

BST 操作

在 BST 中搜索、插入、删除节点的操作都是平均 $O(\log n)$，最坏 $O(n)$.
删除和插入的指针操作本身都是 $O(1)$ 的，但是需要搜索，所以有如上的时间复杂度。
如果搜索、插入、删除的实现是递归调用函数，每次还需要 $O(\log n)$ 的空间复杂度。

遍历树

前序根左右、中序左根右（只能BT）、后续左右根、层序（BFS实现）

AVL

旋转

在 AVL 树中任一节点的两子树高度之差都不大于1. 对于大于1的节点，根据失衡类型进行旋转：

- LL(RR)：左子树更深，且左孩子的左子树更深。

  想法是，让 LL 不平衡的节点 n 的左孩子 l 做为新节点：

  1. n 的左孩子赋值为 l 的右孩子
  2. l 的右孩子赋值为 n
  3. 更新高度
- LR(RL)：左子树更深，且左孩子的右子树更深。

  让 LL 不平衡的节点 n 的左孩子 l 做一次右旋（上述操作左右互换，也就是 RR 失衡时会做的旋转），把问题转化为 LL 失衡从而重复上述过程。

插入的全流程

1. 像普通 BST 一样递归插入到正确位置（成为叶子节点）
2. 递归返回时，沿着插入路径自底向上回溯，每到一个节点：
   - 更新 height，计算平衡因子 BF = height(left) - height(right)
   - 若 |BF| > 1，说明此节点失衡，并且这是"离插入点最近的失衡祖先"。

     此时，做相应旋转来让此子树平衡。(旋转后全树已平衡，后续回溯过程中不会有 |BF| > 1)

有关 AVL 能够促进平衡的数学证明，见这里

Lo7-树的扩展

B 树

B 树是自平衡的多路树，面向磁盘设计。
一棵 m 阶 B 树（m 是阶数，决定每个节点最多能塞多少个分支，在PPT中 $m = 2B$）要求：

- 每个节点最多有 m 个子节点（叶子节点没有子节点，对它们来说是最多有 m-1 个键）；除根节点和叶节点外，每个内部节点最少有 $\lceil m/2 \rceil$ 个子节点
- 节点内的键严格升序排列：$k_1 < k_2 < ... < k_n$
- 设某节点有 k 个子节点，那么它必然有 $k - 1$ 个键。
  若节点有键 $k_1, k_2, ..., k_n$ 和子指针 $p_0, p_1, ..., p_n$，
  则 p_0 指向的子树中所有键 < k_1，p_i （0 < i < n）指向的子树中所有键满足 $k_i < x < k_{i+1}$，p_n 指向的子树中所有键 > k_n
- 所有叶子节点处于同一深度

所有叶子节点处于同一深度，导致树高 $O(\log_m n)$，由于 m 比 2 大得多，与二叉树的 $O(\log_2 n)$ 相比显著降低了磁盘 IO.

2-3 树

性质

2-3 树是 m=3 的 B 树。直接使用 B 树的性质即可推出其性质：

每个内部节点是**2节点**（1个键，2个子指针）或**3节点**（2个键，3个子指针）。

- 2节点的键 k：左子树所有键 $< k$，右子树所有键 $> k$
- 3节点的键 $k_1 < k_2$：左子树键 $< k_1$，中子树键在 k_1 和 k_2 之间，右子树键 $> k_2$

并且所有叶子节点在同一层，跟B树性质一致。

操作

- 搜索：类似二叉树，在每个节点都比较大小来确认走哪个子树，$O(\log n)$.
- 中序遍历：对于2节点，和二叉树一样；对于3节点，做法是 左子树 → 第一个键 → 中子树 → 第二个键 → 右子树。
- 插入：从根开始，按键值比较，像查找一样走到对应的叶子节点。把新键插入这个叶子，如果插入后变成了3个键，就发生分裂：

  取三个键中中间的那个，提升到父节点，原节点剩下左右两个键，分别独立成为两个新节点（中间的那个键所提升到的父节点的两个子指针分别指向它们）
  父节点接收了这个提升上来的键后，自己也可能因此超员，需要重复这个步骤（最坏情况会一直往上传到根节点，导致树长高一层）

- 删除：

  如果删除的不是叶子节点，找到其左子树最大值或右子树最小值，用该值覆盖它，然后转去删除那个前驱/后继本身（它必然是叶子）。问题转化为删除一个叶子节点。
  对于这个叶子节点 N，如果在删除后键数 ≥ 1，结束；否则，
  　　如果 N 是根节点，直接移除它，树整体降一层；
  　　如果不是，查看它的相邻兄弟 S
  　　　　如果 S 是3节点，让 N 和 S 的父节点中分隔 N 和 S 的键下放给N，并让 S 的一个键提升给父节点，结束
  　　如果 S 也是2节点，合并 N、S、父节点的分隔键成一个3节点，此时父节点失去一个键，对父节点递归执行从头开始的修复

红黑树

红黑树是用普通二叉树的结构去模拟2-3树。2-3树有2节点和3节点两种，前者天然就是二叉树节点，但3节点有2个键、3个子树。对此，红黑树的做法是**把一个3节点拆成两个二叉节点，用红色标记，红色节点和其父节点是同一个3节点拆出来的两半。**

这是背后的本质。形式化地定义，一棵红黑树是一棵二叉搜索树，每个节点带一个颜色标记（红或黑），满足：

1. 每个节点是红色或黑色
2. 根节点是黑色
3. 所有NIL节点（每当一个真实节点缺少左孩子或右孩子，那个缺失的位置就指向一个NIL节点）是黑色
4. 红色节点的两个子节点都必须是黑色
5. 从根到任意NIL节点的路径上，黑色节点数量相同

第3,4条性质保证了**从根到任意叶子的最长路径，不会超过最短路径的2倍**，防止了如普通二叉搜索树最坏情况下退化成一条链的情况，保证了性能。

B+树

它和 B 树的区别只在于：

1. 只有叶子节点存数据，内部节点只存"路标"用的键，不存实际记录。任何一个内部节点的键，都会在叶子节点中重复一次。
2. 所有叶子节点用指针串成一条链表，可以从最左边一路扫到最右边。每个叶子节点持有指向下一个

叶子节点的指针。

所有叶子形成的链表，让**范围查询**效率很高：定位起点后变成链表遍历，一次 $O(\log_m n)$ 之后就一直链表遍历即可。但是 B 树就需要多次走从根到目标的路径。
事实上，数据库索引里实际用的几乎都是它。

Lo8-农羊狼菜

DFS/BFS 解决问题和剪枝策略

农羊狼菜

形式化状态：`0` = 起始岸，`1` = 对岸，四个分量分别代表 f,w,g,c

初始状态为 (0,0,0,0)，
每次尝试所有可能的下一步过河方式（农夫自己或者带一个），要求 1. 过河后状态安全 2. 过河后的状态没访问过，
如果达到了 (1,1,1,1) 就完成。

8-puzzle 问题和剪枝策略

> 8 数码问题有一个数学性质：初始状态和目标状态的逆序数奇偶性相同时才有解。所以先检查初始状态的逆序对数量，是偶数再继续。

一个 move 存储为状态+当前 0 的位置+移动后 0 的位置。
每次尝试 0 上下左右四种 move，要求 1. 不越过边界 2. 移动后的状态没访问过，
如果达到了目标状态就完成。

几种剪枝策略：

- A* 算法，在所有可能的 BFS 尝试中优先选择那些目前看起来更接近目标的移动。（对于给定初始状态，再移动到合法的八皇后问题也可以用）
  “更接近”的启发式衡量指标可以是每个数字和目标状态的曼哈顿距离之和、或者是已经匹配好的数字数量。
- 双向 BFS，既从始点向终点搜，也同时从终点向始点搜。

[八皇后、water jug 问题的解决]

数独问题的剪枝策略

- 填完一个数字后，更新受影响空位的候选集
- 优先填候选集大小较小的
- 如果某个位置候选集为空，停止搜索

状态机

状态机是一个抽象计算模型，包含一个状态集合、初始状态和一系列变换规则（这个规则也可能是非确定的，参考非确定性图灵机）。可以把问题抽象成状态。
不变式是一个对初始状态和经过变换不改变正确性的谓词。

[一些不变式的例子]

搜索策略

衡量搜索策略时空复杂度的指标有：

- 最大分支因子 b：每个节点最多能展开出多少个子节点
- 最优解所在的深度 d：从根节点走到"最优解"这个节点,需要经过多少层
- 状态空间的最大深度 m：整棵搜索树最深能延伸到第几层(可能是无穷大)

常见的各种搜索有：

- BFS：优先扩展深度最小的
- DFS：优先扩展最深的
- DLS：除了无法扩展这个终止条件以外，如果过深也停止扩展
- IDS：从 0 到 ∞ 增加 DLS 的扩展深度，直到找到解

	完整性	**最优性**	**时间复杂度**	**空间复杂度**
BFS	如果 b 有限，是的	通常不是，除非问题本身深度最小就是最优	$O(b^{d+1})$ 或者说 $O(b^d)$	$O(b^d)$
DFS	如果空间有限，并且标记重复状态，是的	不是	$O(b^m)$	$O(bm)$

	完整性	最优性	时间复杂度	空间复杂度
DLS	如果深度限制$l \geq d$，是的	不是	$O(b^l)$	$O(bl)$
IDS	是的	是的	$O(b^d)$	$O(bd)$

> 注意这里 DFS,DLS,IDS 的空间复杂度是因为把所有路径上的节点和其待展开的兄弟节点都存储了。
> 做法是，一次性生成全部子节点,全部压栈，回到之前的节点时直接从栈里取下一个就行；如果用游标/迭代器,只记"我展开到第几个子节点了"，就 $O(m)$

双向 BFS 可以让时空复杂度都降到 $O(b^{d/2})$，但是需要知道目标在哪里，以及怎么反向搜索。

Log-优化问题

贪心解法的案例

- 节点合并：合并两个节点为新节点，权值为二者之和，求合并到只剩一个节点时的最小权值

  > 每次都取权值最小的两个合并。

- 作业排序：每个作业都有收益和 ddl 两项指标，都能在 ddl 前的某个单位时间完成，求最大收益

  > 先处理目前收益最大的，并且将其完成实现选择为可行时间范围的最后。

- 哈夫曼编码：给每个字符一个二进制编码，求 bit 数最小的编码

  > 把字符的出现次数作为权重，同“节点合并”得到二叉树

  btw，码表也要占空间，因此在和直接编码比较时要考虑码表

- 0/1背包问题：每个物品有重量和价值，在不超过某重量限制时价值最大

  > 每次取剩余物品中价值密度最高的 （不保证最优，物品是金粉/油这种的分数背包可以保证）

- 硬币面值：有若干面值的硬币，用最小的数量来凑出指定面值

  > 取不超过剩余面值中面值最大的（不保证最优）

贪心 DP 对比

贪心	DP
单步地选择局部最优，得到的子问题也递归地这样解决	用状态转移方程逐步解决所有子问题，直到原问题
在有最优子结构的同时还有局部最优性质才能保证解是全局最优的	在有最优子结构时就能保证解是全局最优的

贪心需要更低的时空复杂度，并且不需要记忆化。

旅行商问题解法

常用的解法是 swap-2/swap-3.

swap-2 的做法是，任选路径中的两条边，把它们都去掉，然后用唯一的另一种方式重新连接剩下的两段，看总长度是否变短。
比如说，把 2-3 和 5-6 的边切断后，比较原来的 `12-3456-781` 和 `12-6543-781` 哪个更短。

swap-3 与之同理，把三条边去掉，然后看剩下七种有没有路径更短的，比如：

```
去掉三条边后被切成三段 [段1] [段2] [段3]，可以尝试的重组方式包括：
- [段1] [段2反转] [段3]
- [段1] [段3] [段2]
- [段1] [段3反转] [段2反转]
- ...
```

分支限界法(Branch and Bound)

通过计算上下界来排除那些确认不可能是最优的情况，从而达到剪枝。
需要好的限界函数，配合搜索顺序来更高效地寻找最优解。相比之下，回溯通常是找全部解或任意可行解。

以求最大值举例来说流程：
先计算根节点的下界（或者快速找到一个可行解得到下界），并放入“活节点集(live node set)”，
之后每次从活节点集中取一个节点来处理（通常选上界最大的那个），对其进行扩展，生成所有合法的子节点，分别计算上界。如果某个子节点的上界小于目前的最优下界，就丢弃这个节点；否则加入活节点集。

（注意上述叙述中只涉及了对上界的计算，什么时候计算下界呢？通常，在根节点计算下界以后，剩下只有在已经得到可行解了之后，用可行解的结果作为下界。但是PPT没有提到什么时候计算下界，这只是通常做法）

一些上下界计算的例子：

- 背包问题：上界是已取得的价值+剩余可用重量*剩余物品中最大的价值密度
- 含有惩罚的工作排序：下界是所有未选择的工作的惩罚之和
- 单源最短路问题：下界是从起点走到当前节点的最短路径长度+从当前出发的所有出边中权重最小的那条边
- 无向图旅行商问题：下界是对每个顶点都计算含有它的最便宜的两条边，求和/2
- 有向图旅行商问题：

  写出点之间互相到达的距离，得到 n*n 代价矩阵。
  先对每一行，都找出该行最小值，并把该行所有元素减去该值；之后在新矩阵中，对每一列都找出最小值，下界是求和后再加上先前的各行最小值之和。

[DP、贪心和 branch bound 的特点；branch bound 和回溯的区别]

L10-六度空间

图的存储

邻接表存储的缺点：寻找边的存在性需要 $O(degree)$；
邻接矩阵的缺点：对于稀疏图浪费空间，添加或删除顶点不方便。

邻接矩阵的一个性质：$(A^k)_{i,j}$ 是从 i 到 j 的长度为 k 的路径总数。

[子图、补图、同胚、遍历的定义]

DFS 的应用

找割点

定义 `dfn[u]` 是顶点在 DFS 遍历中被访问到的顺序编号，`low[u]` 为从 u 开始 dfs，除了父亲以外所能走到的 `dfn` 最小的节点。

low 是从子节点向父节点，在回溯的时候计算的，

```
low[u] = min(
    dfn[u],
    low[v]  对所有 u 的孩子 v,
    dfn[w]  对所有(u,w), 除非这是dfs时访问 u 的边
)
```

对于非根节点，如果有 (u,v) 使得 low[v] ≥ dfn[u]，那么 u 是割点；
对于根节点：如果它有 ≥ 2 个孩子（在 DFS 树中），那么根节点是割点

找环

DFS 遍历过程中，给每个顶点标记一种颜色：

- 白色：还没被访问过
- 灰色：已经进入了 DFS 的递归调用栈，但还没有完成对它的所有邻居的探索（还没回溯）
- 黑色：这个顶点和它所有能到达的子节点都已经探索完，已经从递归栈中退出

当 DFS 正在探索顶点 u 的某条边 (u, v) 时，如果发现 v 是灰色，说明存在环。

找 SCC

在和之前割点算法一样地计算 low/dfn的同时，加一个访问顶点时压栈：DFS 访问一个新顶点，就把它压入栈中。

回溯时，到某个顶点 u，发现 `low[u] == dfn[u]`，则从栈顶开始往下弹，一直弹到 u（包含u）为止，这些被弹出的顶点就构成一个完整的SCC

求拓扑排序

遍历所有顶点，如果它没被访问过，就做 DFS（通常开局选一个入度为零的）。
　　某个顶点做 DFS 的做法是，一个个递归地 DFS 它指向的每一个未访问过的顶点。等所有依赖它的顶点都处理完了，就把当前顶点压入栈。
所有顶点都访问过以后，栈顶到栈底正好是从早期任务到晚期任务的合法拓扑顺序。

BFS 的应用

- 求单源最短路径：第一次访问到某个顶点的那一刻，就已经找到了从起点到这个顶点的最短路径。
- 二分图检测（染色法）：BFS 过程中，给每一层的顶点交替染色。如果发现某条边的两个端点颜色相同，说明这个图不是二分图。
- 求树的直径（所有点对的最短路径里，最长的那一条）：

  从任意顶点 a 出发，做一次 BFS，找到离它最远的顶点 b；再从 b 出发，做第二次 BFS，找到离 b 最远的顶点 c。b 和 c 之间的距离，就是整个树的直径

求最小生成树

kruskal

1. 把所有边按权重从小到大排序
2. 依次处理每条边，用并查集判断是否成环
3. 直到MST含有 $|V|-1$ 条边时停止

时间复杂度：

- 排序所有边：O(E log E)；
- 遍历每条边，配合并查集的Find/Union操作（接近O(1)均摊）：O(E)

空间复杂度：

- 存储每条边：O(E)（排序的额外空间一般不会超过这个）
- 并查集存储每个顶点的父节点：O(V)

prim

维护一个已经在生成树中的顶点集合 S，每一步都从 S 到 V\S 的所有边里，挑选权重最小的那条边，把它的另一端顶点加入 S，直到 S 包含所有顶点。
btw，这个和dijkstra很像，一个维护从起点到每个外部顶点的最短路径长度，一个维护从S集合到每个外部顶点的最小连接边权重

实现方式：

```
初始化：S = {起点}，minEdge[起点] = 0，其余顶点 minEdge = ∞
将所有顶点按 minEdge 放入小根堆

while S 不等于 V:
    从优先队列取出 minEdge 最小的顶点 u（且u不在S中）
    将 u 加入 S
    for u 的每个邻居 v（v不在S中）:
        if weight(u,v) < minEdge[v]:
            minEdge[v] = weight(u,v)
            min_heapify()
```

- 时间复杂度：`while S 不等于 V` 和 `for u 的每个邻居 v（v不在S中）` 共同构成了 $O(E)$ 次循环，循环中每次 `min_heapify()` 是 $O(\log V)$.
- 空间复杂度：`minEdge` 等 $O(V)$，存图 $O(V+E)$.

SSSP 问题

dijkstra

维护一个"已确定最短距离"的集合，每一步从未确定的顶点中，挑出当前距离最小的那个，将它"确定下来"，然后用它去更新邻居的距离。
实现：

1. 初始化：起点距离为0，其余所有顶点距离为∞
2. 把其余顶点建最小堆，维护"未确定顶点的当前最短距离估计"
3. 重复以下操作，直到所有顶点都确定：
    - 取出当前堆中（未确定的）距离最小的顶点 u
    - 将 u 标记为已确定（它的最短距离不会再变了）
    - 对 u 的每个邻居 v：如果 `dist[u] + weight(u,v) < dist[v]`，就更新 `dist[v]`

时间复杂度 $O(E \log V)$

bellman-ford

1. 初始化：起点距离为0，其余顶点距离为∞
2. 重复 V-1 次：遍历图中所有边 (u, v, weight)，如果 `dist[u] + weight < dist[v]`，就更新

`dist[v] = dist[u] + weight`

3. （可选）第 V 次再扫一遍所有边：如果还有更新，说明图中存在负权环，最短路径不存在（无下界）

时间复杂度 $O(EV)$

APSP 问题

Floyd-Warshall

> 核心递推公式是 `dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] )`，不断更新。

实现：

1. 初始化距离矩阵 `dist[][]`：
   - `dist[i][i] = 0`
   - `dist[i][j] = weight(i,j)`（如果i,j之间有直接边）
   - `dist[i][j] = ∞`（如果i,j之间没有直接边）
2. 三层循环，依次把每个顶点 k 当作中转点：

```
for k = 1 to V:        # 依次尝试每个中转点
    for i = 1 to V:    # 枚举起点
        for j = 1 to V:  # 枚举终点
            if dist[i][k] + dist[k][j] < dist[i][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
```

3. 循环结束后，`dist[i][j]` 就是 i 到 j 的最短路径长度

它可以正确处理负权边，也可以进行负权环检测：算法结束后，检查对角线 `dist[i][i]`，如果有任何 `dist[i][i] < 0`，说明存在负权环。
需要 $O(V^2)$ 空间复杂度。

[各种最短路算法的各项指标总结]

L12-NP问题

优化问题和判定问题

对于离散且有界的优化问题，可以通过多次询问判定问题来求解优化问题，而且询问的次数通常是对数级别的（这得益于二分搜索的思想）。所以，如果你能高效解决判定问题,你就能高效解决优化问题(只需要多调用对数次)

> 举个例子，“图中最短的哈密顿回路长度是多少？”可以被转化为对数次数的“图中是否存在长度 ≤ k 的哈密顿回路？”问题。

P,NP,NP-hard 和 NP-complete 问题

（如果有不了解的术语，见这里）

- P 问题：能被确定性图灵机在多项式时间内求解的判定问题。
- NP 问题：能被非确定性图灵机在多项式时间内求解的判定问题 <=> 解可以在多项式时间内被验证的问题
- NP-hard 问题：NP 中每一个问题都能在多项式时间内归约到它
  它不要求问题属于 NP，比如说，可能是优化问题、非判定问题
- NP-complete 问题：属于 NP，且 NP 中的每一个问题都能在多项式时间内归约到它。

一些例子：

问题	类型	复杂度(不一定最优)
排序	P	$O(n \log n)$
二分图匹配	P	$O(VE)$
SAT	NPC	$O(2^n \cdot m)$，其中 n 是变量数，m 是子句数
哈密顿回路	NPC	$O(n!)$（暴力枚举）
3-SAT	NPC	$O(2^n \cdot m)$
Clique	NPC	$O(2^n \cdot n^2)$（暴力枚举所有子集并检查）
Circuit-SAT	NPC	$O(2^n)$（枚举所有输入组合）

比如说，0/1背包问题（判定版，不是优化问题）、SAT问题和子集和问题，三者都是 NPC，所以彼此之间理论上存在多项式时间的相互归约

PPT 问题解答/部分内容的详述

Lo2-约瑟夫环

42

FP标记当前执行的函数的栈帧底部，在一个函数执行期间保持不动；SP标记栈顶，会随着压栈/弹栈一直移动
Caller是发起调用的函数；Callee 是被调用的函数

假设 P 是 caller, Q 时 callee，调用的全流程是（x86 从高地址往低地址增长）：

```
高地址
┌──────────────────────┐ ← FP（P的帧底部）
│    P 的栈帧          │
│    P 的局部变量      │
└──────────────────────┘ ← SP（栈顶）
低地址
```

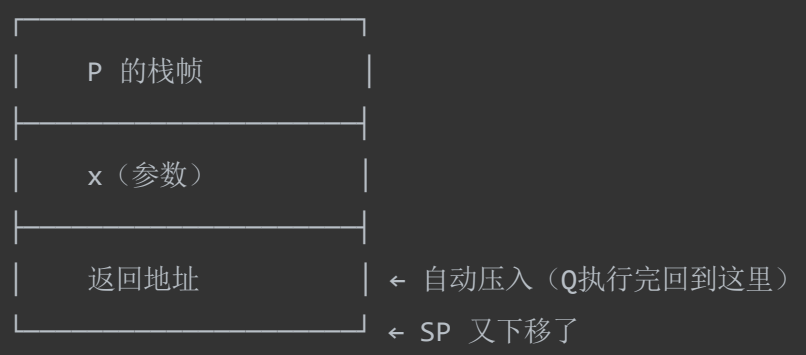
```
P 执行：把 x 的值放到栈上或寄存器里


┌──────────────────────┐
│    P 的栈帧          │
├──────────────────────┤
│    x（参数）         │ ← 新压入
└──────────────────────┘ ← SP 下移了
```

call 指令自动做两件事：

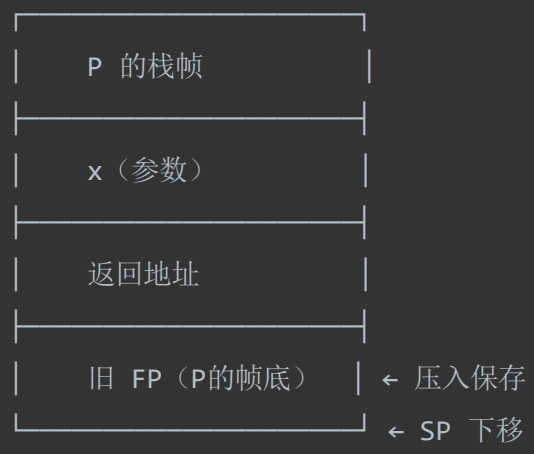
a. 把"P中call的下一行地址"压栈

b. 跳转到 Q 的代码

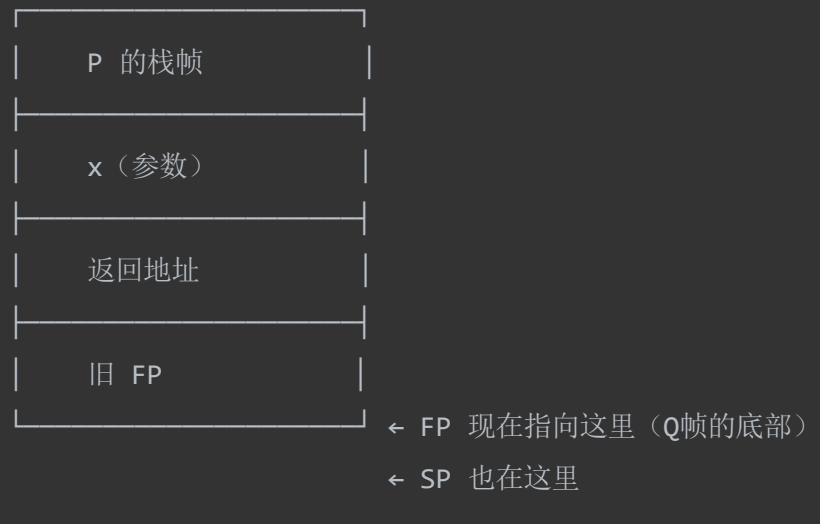


（可选）把 P 的 FP 保存起来

因为等会儿FP要指向Q的栈帧，P的FP就会丢失



现在 FP 下移，指向 Q 的栈帧底部



```
SP 继续下移，为 Q 的局部变量开辟空间


┌──────────────────────┐
│    P 的栈帧          │
├──────────────────────┤
│    x（参数）         │
├──────────────────────┤
│    返回地址          │
├──────────────────────┤
│    旧 FP             │ ← FP 在这里（不动了）
├──────────────────────┤
│    int t             │ ← Q 的局部变量
│    int v[10]         │ ← Q 的局部数组
└──────────────────────┘ ← SP 在这里
```

当 Q 执行完毕后，将会反向操作，逐步还原：

1. 把 Q 的局部变量释放
2. （可选）SP = FP，并弹出之前存的Q的FP
3. 弹出并跳转到返回地址
4. 清理调用 Q 的参数

Lo5-part1-最大子数组和

7

Q1：分治算法是可以的，`left_max` 和 `right_max` 的递归计算是互相独立的，可以分配到不同的处理器/线程并行执行；跨中点的扫描（`left_cross_max` 和 `right_cross_max`）同样可以并行计算（一个向左扫，一个向右扫）。
理论上，使用 P 个处理器，时间复杂度可以从 O(n log n) 降到 O(n log n / P)，

Q2：环形数组的最大子数组和，有两种情况：

1. 最优子数组不跨越首尾。直接用普通 Kadane 算法求一次，得到 `max_normal`。
2. 最优子数组跨越首尾。这等价于 总和 - 中间被排除的那段最小子数组和。解法是，用 Kadane 算法的"最小子数组和"版本，求出数组中和最小的子数组 `min_subarray`，则跨越首尾的最大和 =

`total_sum - min_subarray`
注意，如果数组全为负数，上面的"跨越情况"会把所有元素都排除掉（结果是空数组，不合法），此时答案应该退化为只取 `max_normal`。

Q3：核心思路是把二维问题降到一维，再用 Kadane：

1. 枚举行的范围：固定上边界 `top` 和下边界 `bottom`（两层循环，共 O(m²) 种组合）。
2. 压缩成一维数组：对每一组 `(top, bottom)`，把这些行之间、每一列的元素纵向求和，压缩成一个长度为 n 的一维数组。
3. 对压缩后的一维数组跑 Kadane 算法，求出该行范围下的最大子数组和。
4. 取所有 `(top, bottom)` 组合中的最大值，即为答案。

13

分治法解 Closest Pair of Points 问题：

1. 将所有点按 x,y 坐标排序
2. 每次按 x 坐标分中点，分别递归求解
3. 求跨越中线的最小距离。已知左右两边各自的最近距离的最小值是 `d`，那么解会出现在以中线为中心、宽度为 2d 的"带状区域内。
   对带状区域内的每个点，只需要和它后面 y 坐标差小于 d 的最多 7 个点比较（可以用几何论证证明：在宽度为 d 的带子里，若两点 y 坐标差小于 d，那么这样的点最多 7-8 个，因为再多就会有两点距离小于 d，矛盾）。

每次合并 $O(n)$，一开始的排序 $O(n \log n)$

Lo6-公共祖先

4

这个方法需要我们建一个 `self.up[u][k]` 表，表示节点 `u` 向上跳 2^k 步到达的祖先，但是具体的做法被省略了。
建此表的一个做法是，从根节点开始遍历整棵树，对每个节点记录节点的深度和节点的直接父亲，然后用

$$up[u][k] = up[\,up[u][k-1]\,][k-1]$$

来计算该节点的 up table。这个过程的时间复杂度是 $O(n \log n)$.

21

层序遍历的代码

```
    def levelOrderTraversal(rootNode):
        if not rootNode:
            return
        else:
            customQueue = queue.Queue()
            customQueue.enqueue(rootNode)
            while not(customQueue.isEmpty()):
                root = customQueue.dequeue()
                print(root.value.data)
                if root.value.leftChild is not None:
                    customQueue.enqueue(root.value.leftChild)
                if root.value.rightChild is not None:
                    customQueue.enqueue(root.value.rightChild)
```

是一个 BFS 的实现。创建队列并把根节点放进去后，
每次处理一个队列中的节点：输出自己，然后把所有孩子放进队列。

26

伪代码中的 `update height of disbalancedNode and newRoot` 这一行，如果也写成代码会是这样：

```
    disbalancedNode.height = 1 + max(height(disbalancedNode.leftChild),
    height(disbalancedNode.rightChild))
    newRoot.height = 1 + max(height(newRoot.leftChild), height(newRoot.rightChild))
```

依次计算和更新 `disbalancedNode` 和 `newRoot` 的 height.

AVL rotation 促进平衡的证明

为什么四种 rotation 能够让树变得更平衡，也就是说能让左右子树高度之差大于1的节点消失？给出一个证明：

不妨是LL或LR失衡，记

- `X = disbalancedNode`
- `Y = newRoot = X.leftChild`
- X',Y' 分别是其旋转后的结果
- $A = \text{height}(Y.\text{leftChild})$
- $B = \text{height}(Y.\text{rightChild})$
- $C = \text{height}(X.\text{rightChild})$

X 触发右旋了，是因为有：

$$\text{BF}(X) = \text{height}(Y) - C = 2$$

此时 `Y`、其两个子树都满足 AVL 性质（因为失衡只发生在最近祖先，子树本身仍平衡），有 $|A - B| \leq 1$

由 $\text{height}(Y) = 1 + \max(A, B)$ 代入：

$$\max(A, B) = C + 1$$

A,B 的大小关系决定了这是哪种失衡：LL 型 $A \geq B$，LR 型 $A < B$

旋转后：

$$\text{height}(X') = 1 + \max(B, C), \qquad \text{BF}(Y') = A - \text{height}(X') = A - 1 - \max(B, C)$$

LL 型（$A \geq B$）

由 $|A - B| \leq 1$ 得 $B \geq A - 1$，结合 $A \geq B$ 得 $A - 1 \leq B \leq A$。

由 $\max(A, B) = A = C + 1$，得 $B = C, C + 1$

- 若 $B = C$：$\text{BF}(Y') = A - 1 - C = (C + 1) - 1 - C = 0$
- 若 $B = C + 1$：$\text{BF}(Y') = A - 1 - (C + 1) = (C + 1) - 1 - (C + 1) = -1$

则 $\text{BF}(Y') \in 0, -1 \subset [-1, 1]$，该位置新的节点 Y' 是平衡的。
还需验证 X'（新的右孩子）本身也平衡：$\text{BF}(X') = B - C \in 0, 1 \subset [-1, 1]$

LR 型（$A < B$）

记：

- `Y.rightChild = Z`
- $Z_L = \text{height}(Z.leftChild)$
- $Z_R = \text{height}(Z.rightChild)$

则 $B = \text{height}(Y.rightChild) = \text{height}(Z) = 1 + \max(Z_L, Z_R)$。

完整的树结构（旋转前）：

由 $|A - B| \leq 1$ 得 $A \geq B - 1$，结合 $A < B$ 得 $A = B - 1$。

由 $\max(A, B) = B = C + 1$，得 $A = C$

由 `z` 自身满足 AVL 性质：$|Z_L - Z_R| \leq 1$

由 $B = 1 + \max(Z_L, Z_R) = C + 1$ 得 $\max(Z_L, Z_R) = C$

第一步：对 Y 做左旋，`Z` 上位成为新的子树根，`Y` 降级为 `Z` 的左孩子，`Z.leftChild` 转移给 `Y` 当右孩子。

```
左旋前：          左旋后：
    Y               Z
   / \             / \
  A   Z           Y   Z_R
     / \         / \
   Z_L  Z_R     A  Z_L
```

新 Y（记为 Y'）的高度 $\text{height}(Y') = 1 + \max(A, Z_L)$

新 Z（记为 Z'，即左旋后的根）的高度：$\text{height}(Z') = 1 + \max(\text{height}(Y'), Z_R)$

$\text{BF}(Y') = A - Z_L$

由 $A = C$ 和 $\max(Z_L, Z_R) = C$，$A - Z_L = C - Z_L \geq 0$。又因为 $|Z_L - Z_R| \leq 1$：

- 若 $Z_L = C$（即 $Z_L \geq Z_R$）：$\mathrm{BF}(Y') = C - C = 0$
- 若 $Z_L = C - 1$（即 $Z_R = C$）：$\mathrm{BF}(Y') = C - (C-1) = 1$

所以 $\mathrm{BF}(Y') \in 0, 1 \subset [-1, 1]$，左旋这一步产生的 Y' 子树是平衡的。

左旋完成后，把 `z'`（带着 `Y'` 作为其左孩子）作为 `X.leftChild`，

第二步：对 `X` 做右旋，等价于把上面的结构视为一个新的"LL 型"问题来处理：此时 `X` 的左孩子是 `Z'`，`Z'` 的左子树是 `Y'`、右子树是 `Z_R`。

仿照 LL 型证明的记号：

$$A_{\text{new}} := \mathrm{height}(Y'), \qquad B_{\text{new}} := Z_R, \qquad C_{\text{new}} := C$$

我们需要验证 $A_{\text{new}} \geq B_{\text{new}}$（LL 型条件），从而可以套用结论。

$A_{\text{new}} = \mathrm{height}(Y') = 1 + \max(A, Z_L) = 1 + \max(C, Z_L)$。

情况一：$Z_L = C,\ Z_R \in C-1, C$

$A_{\text{new}} = 1 + \max(C, C) = C + 1$ $B_{\text{new}} = Z_R \leq C < C + 1 = A_{\text{new}}$

满足 $A_{\text{new}} \geq B_{\text{new}}$

情况二：$Z_R = C,\ Z_L \in C-1, C$

$A_{\text{new}} = 1 + \max(C, Z_L) = 1 + C = C + 1$ $B_{\text{new}} = Z_R = C < C + 1 = A_{\text{new}}$

同样满足 $A_{\text{new}} \geq B_{\text{new}}$

因而可以使用 LL 情况的证明。证毕。

Lo8-农羊狼菜

18

核心不变量在于，$x - y$ 模 3 不变。并且可以证明，只要满足这个条件，一定可达。

因此前两个坐标是 Wall-E 走不到的，第三个可以。

19

核心不变量在于，各分量的变化模 3 完全相同。并且可以证明，只要满足这个条件，一定可达。

因此一、三可达，二不行。

43

魔方问题里，状态空间太大，使用搜索与回溯这个方法基本不可行。

L10-六度空间

10

“check if given nodes are connected” 这里指代的是检查两个顶点之间是否存在一条直接的边。

如果是一些顶点，检查子图是否连通，这个问题的时间复杂度应为 $O(V+E)$，其中 V 是 given 的顶点数。
因为不管用BFS/DFS还是并查集，**最后都要检查所有点是否 visited**，就必须有 $O(V)$ 的复杂度。

13

1. 直接暴力枚举顶点三元组，$O(V^3)$；
2. 更好的方法是对于每条边，找它的两个顶点的公共邻居，注意这个方法要把最后结果除以 3. 复杂度 $O(E\cdot \bar{degree})$，$\bar{degree}$ 是点的平均的度。

14

1. 如果三角形是无视方向怎么都行，和上面无向图的一样。
2. 如果要求是回路，
    1. 邻接矩阵乘法，$O(V^3)$
    2. 对每条边 u2v，找 u 的入邻居与 v 的出邻居的交集，再除以三去重，$O(E\cdot \bar{degree}^2)$
3. 如果要求 u2v,u2w,v2w的“传递三角形”，对每条边 u→v，看 u 和 v 的出邻居有多少个共同点 w，把这些数目累加起来就是传递三角形的总数，$O(E\cdot \bar{degree}^2_{\text{out}})$

L12-NP问题

图灵机

图灵机(Turing Machine)是一个抽象的、理想化的"计算机",用一个极简且严格定义的数学模型来精确刻画"什么是可计算的"。它的组成有，

- 一条无限长的带子，分成一个个可以存放一个符号的格子
- 一个读写头:停在带子的某个格子上,能够读取当前格子的符号,写入新符号,然后向左或右移动一格。
- 一个转移函数 $\delta(q, a) = (q', a', d)$（在非确定性图灵机中，可能等于多个值）：如果当前状态是 q,读到的符号是 a,那么就把这个格子写成新符号 a',转移到新状态 q',然后读写头往方向 d(左或右)移动一格。

工作时，机器从初始状态开始,根据当前状态和读到的符号,不断套用转移规则:读 → 写 → 移动 → 换状态,如此循环,直到进入某个终止状态为止。

图灵机重要在于，
任何"直觉上可计算"的函数,都能被某个图灵机计算。尽管它的"指令集"只有读、写、左移、右移这几个动作,但理论上能模拟任何现代计算机能做的计算。
P、NP、NP-complete、多项式时间归约等等，背后的"多项式时间"都是以图灵机的计算步数来衡量的。

图灵机可以分类为：

- 确定性图灵机：转移函数对每个 (状态, 符号) 只有唯一规则（在P的定义中使用）
- 非确定性图灵机：每步可以"猜"多个可能的转移,只要存在一条路径能接受（在NP的定义中使用）（非确定性算法:允许在某个步骤"凭空猜出"一个可能有用的答案,然后继续往下走,只要存在一种猜测方式能导向正确结果,就认为这个算法成功了）

多项式时间归约

存在一个多项式时间可计算的函数 f,把问题 A 的任意一个输入 x,转换成问题 B 的一个输入 $f(x)$,满足转换后的问题 $f(x)$ 在 B 中的答案("是"或"否")和原来 x 在 A 中的答案完全一致。

也就是说，把问题 A 转化成问题 B,如果你能解决 B,那你就能用 B 的答案解决 A