**算法基本概念**

程序=数据结构+算法​

**算法的定义**

算法是若干指令的有穷序列，满足性质：

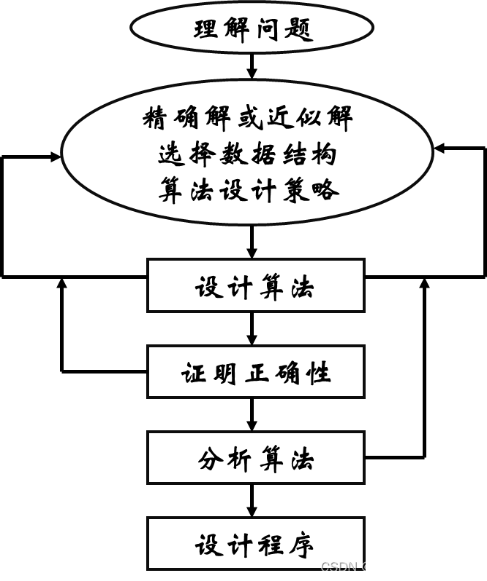
①输入：有外部提供的量作为算法的输入。

②输出：算法产生至少一个量作为输出。

③确定性：组成算法的每条指令是清晰，无歧义的。

④有限性：算法中每条指令的执行次数是有限的，执行每条指令的时间也是有限的。

⑤可行性: 算法是能够有效解决问题的。

​

**算法复杂度分析**

一个算法的运行时间是指在特定输入时所执行的基本操作数或步数。

插入排序例子

假定每次执行第*i*行所花的时间是常量*ci*；对 *j* = 2, 3, … *n*, 假设tj表示对那个值 *j* 执行*while*循环测试的次数。

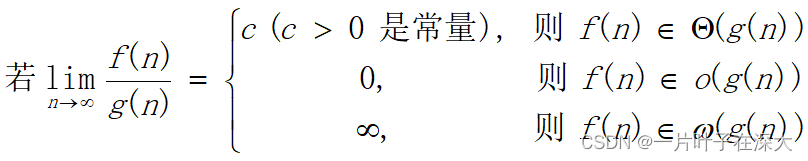
当一个*for*或*while*循环按通常的方式（由于循环头中的测试）退出时，执行测试的次数比执行循环体的次数多1。

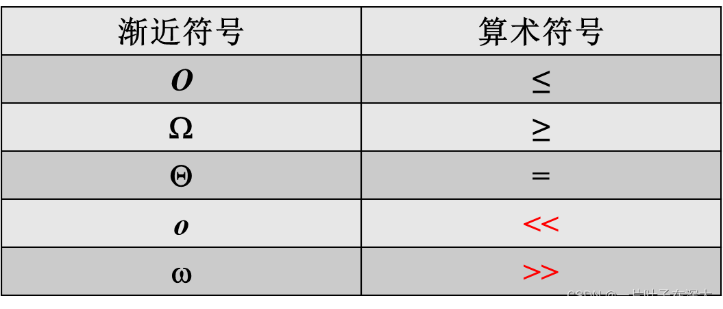
则插入排序的运行时间为所有times与对应cost之积的和，即取决于不确定的tj。

最好情况下tj的值为1，最坏情况下tj的值为j，平均情况下tj的值为j/2。

**渐近记号**

**渐进记号极限定义**

​

​

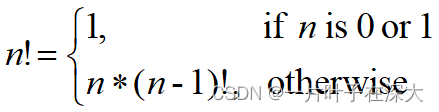
**分治步骤**

①Divide（分解）：将问题划分为若干个子问题

②Conquer（求解）：递归地求解子问题；若子问题规模足够小，则直接解决之

③Combine（组合）：将子问题的解结合成原问题的解

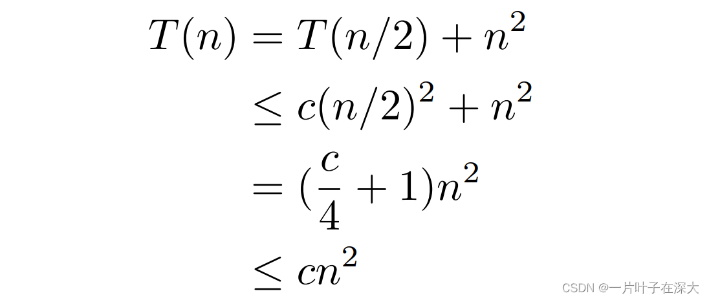
***n*!的递归式**

​

**代入法**

T(n) = T(n/2) + n²

假设T(n)∈O(n²)，证明T(n)≤cn²：

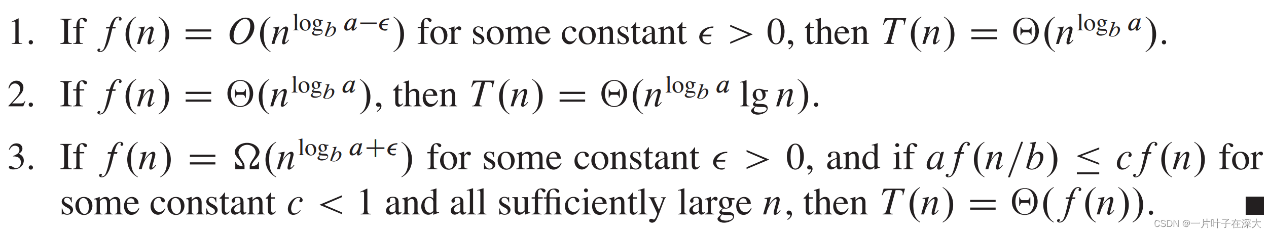
​

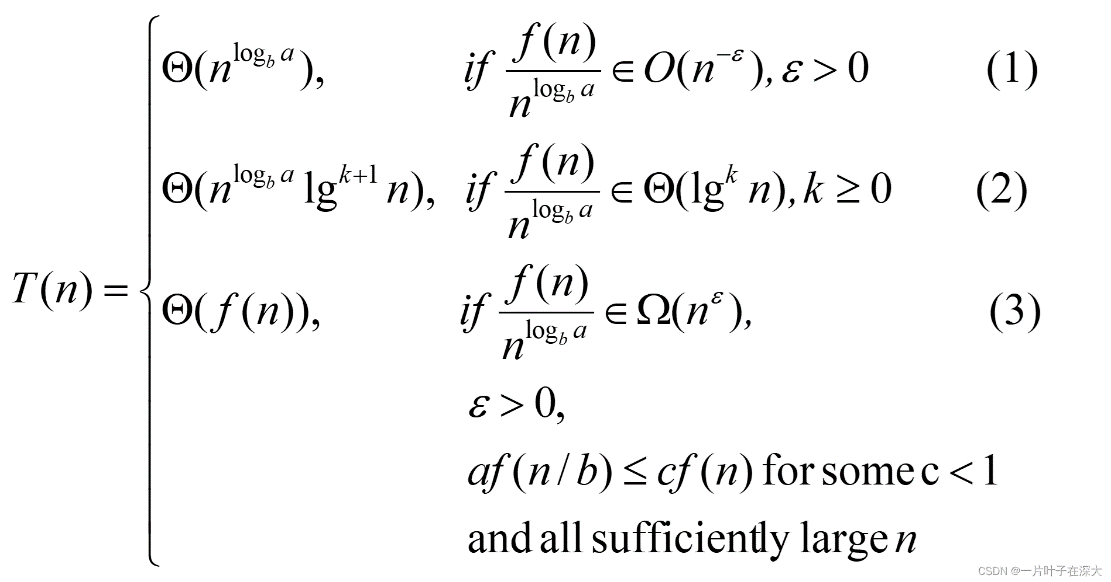
**主方法**

主方法可解如下形式的递归式

*T*(*n*) = *aT*(*n*/*b*) + *f*(*n*)

主定理

​

​

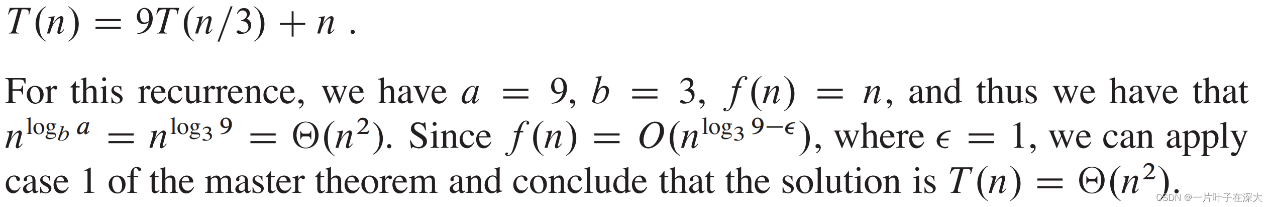
 关键是比较 *f*(*n*) 和 *nlogba，*看谁大：

①f(n)小，case1成立

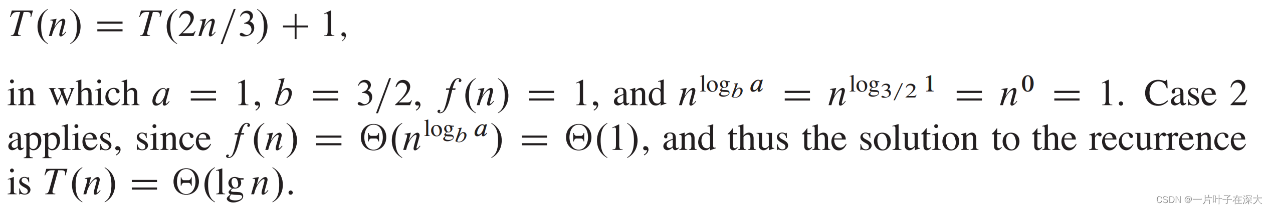
②差不多大，case2成立

③f(n)大，case3成立

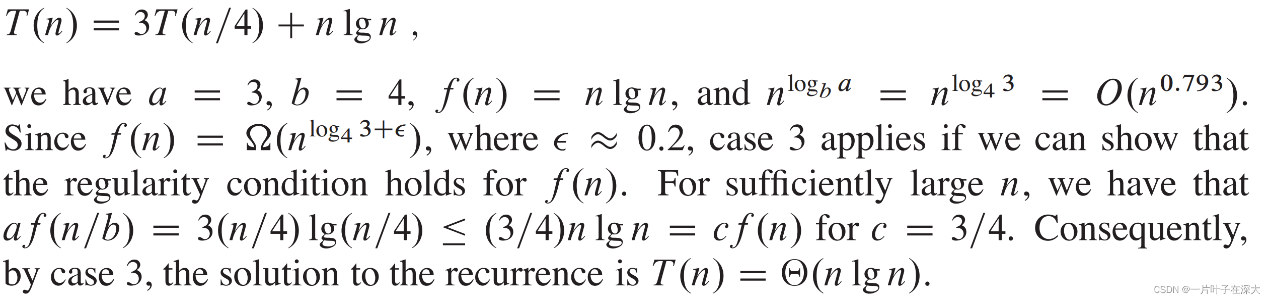
case1例子：

​

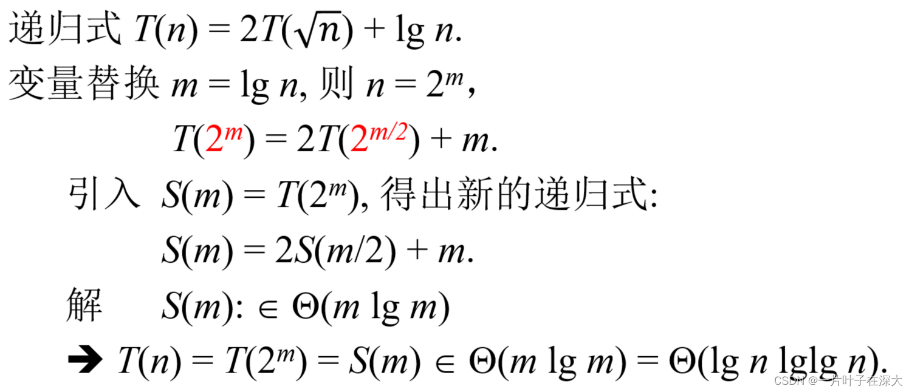
case2例子：

​

case3例子：

​

**改变变量**

​

**二叉树**

***有序树***是一棵有根结点的树，其中每个结点的孩子结点都是有序的 (第一、二个孩子结点等等)。

***二叉树***是一棵有根结点的有序树，其中每个结点最多有两个孩子结点，并且左孩子结点和右孩子结点可区分 (也就是说他们有不同属性)。

***结点 的深度*** 是从这个结点到根结点的简单路径上边的数目**。**

***树 的深度*** 是树中所有结点最大的深度**。**

***结点 的高度*** 是从该结点到一个叶子结点的最长简单路径的边数**。**

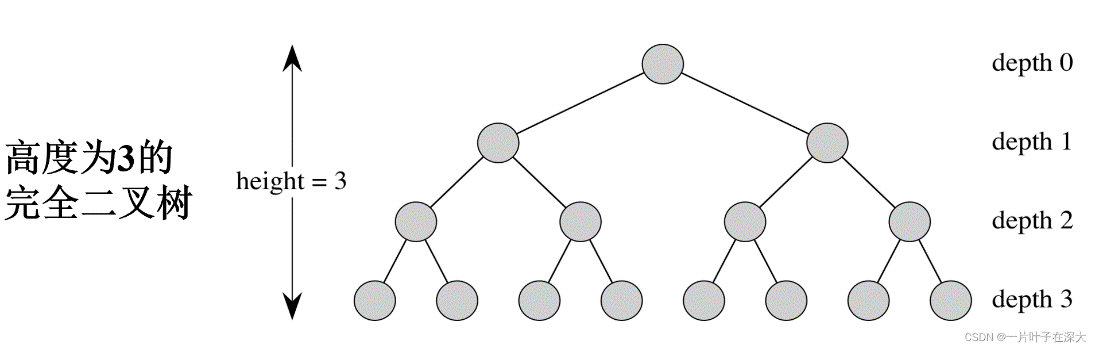
***树  的高度*** 是树的根结点的高度= 树的深度。

**叶子结点：**没有孩子的结点，也称外部结点。

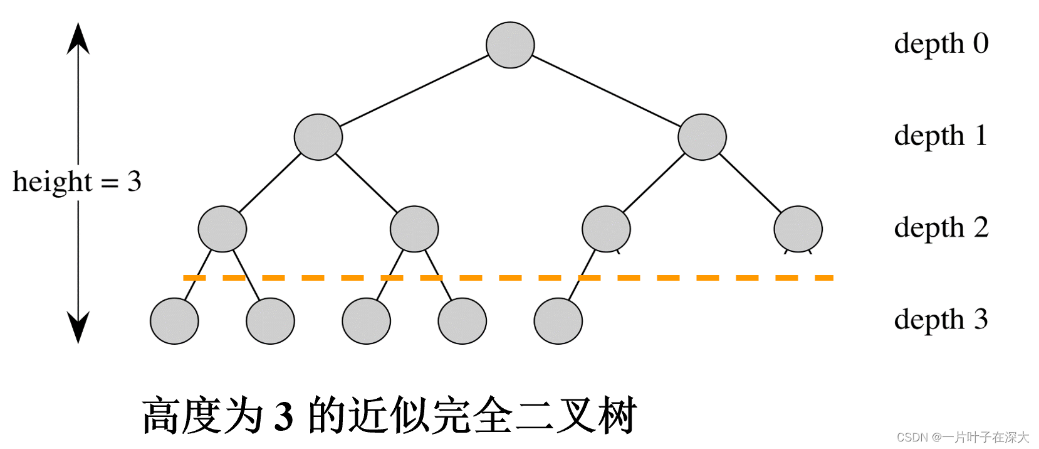
**内部结点：**非叶子结点。

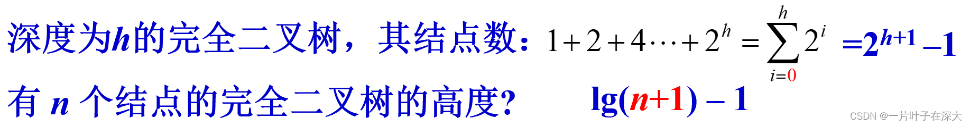
**结点的度：**结点孩子的数目。

***完全二叉树*** 是一棵所有叶子结点在同一深度，而且每个非叶节点都有两个孩子结点的二叉树。

​编辑

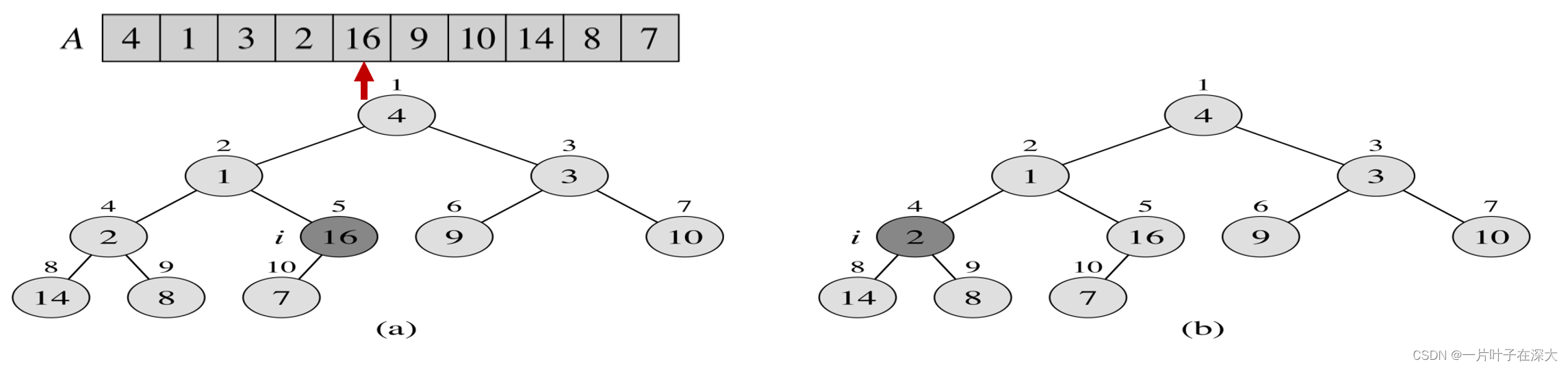
***近似完全二叉树***深度为 *d，*只考虑深度为 *d* – 1 的部分是完全二叉树，深度为 *d* 的结点都在靠左部分。

​

​

**建堆**

从n/2向下取整开始调整堆



建堆的代价为O(n)。

**堆排序**

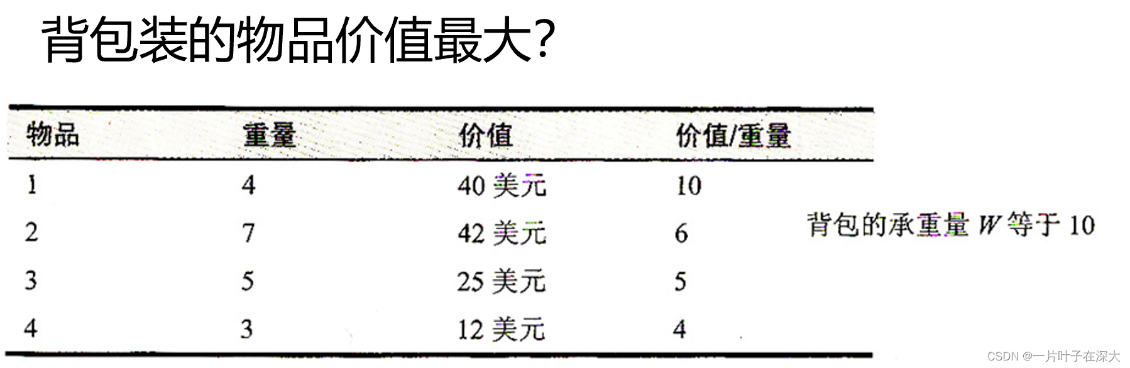
在数组上建一个最大堆。

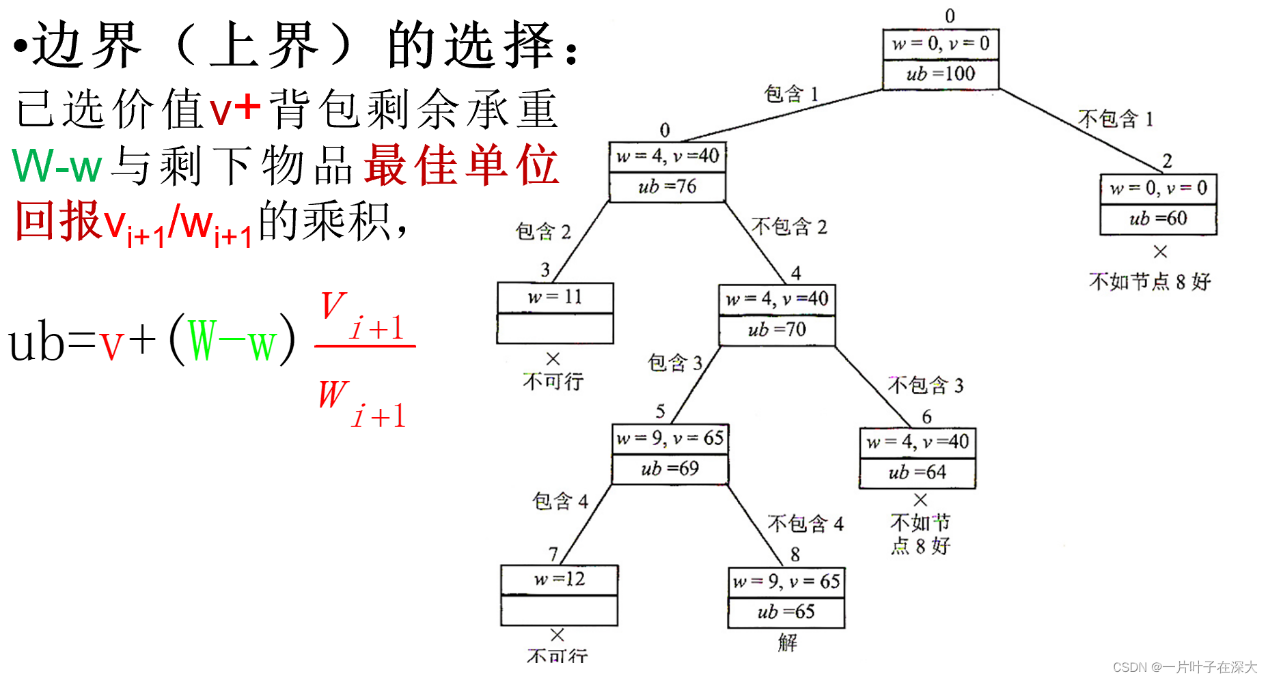
从根结点开始 (它的值最大), 算法将最大值放到数组中正确的地方，例如将它与数组中最后一个元素交换位置。

“去掉” 数组中最后一个元素 (已经在正确的位置)， 在新的根结点上调用 Max-Heapify，新的根结点满足最大堆性质。

重复“去掉” 操作直到只剩一个结点 (也就是最小值), 得到已经排序的数组。

**0-1背包问题**

​

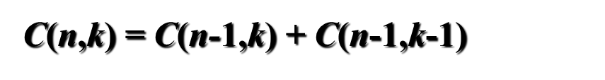
​

**动态规划步骤**

动态规划的思想实质是**分治思想**和**解决冗余**。

求解步骤

①找出最优解的性质，并刻画其结构特征；

​

②递归地**定义最优值**（写出动态规划方程）；

③以**自底向上**的方式计算出最优值；

④根据计算最优值时记录的信息，构造最优解。

注：

－步骤①~③是动态规划算法的基本步骤。如果只需要求出最优值的情形，步骤④可以省略

－若需要求出问题的一个最优解，则必须执行步骤④，步骤③中记录的信息是构造最优解的基础。

动态规划的有效性依赖于问题具有两个重要性质

**最优子结构**

问题的最优解是由其子问题的最优解来构造，则称该问题具有最优子结构性质。

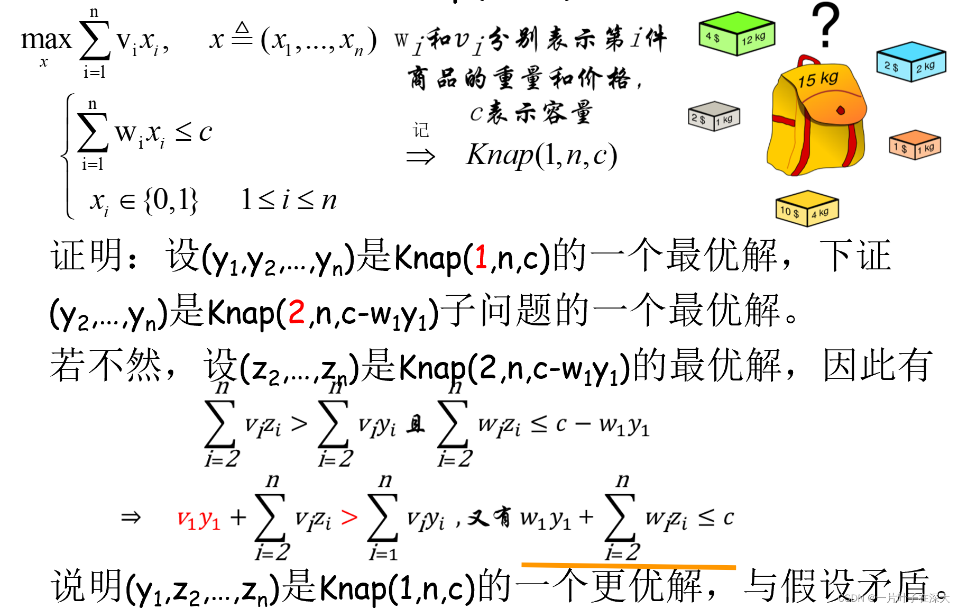
**重叠子问题**

在用递归算法自顶向下解问题时，每次产生的子问题并不总是新问题，有些子问题被**重复计算多次**。动态规划算法利用子问题的重叠性质，**对相同子问题只求解一次**，将其解保存在一个表格中，以后该子问题的解直接查表。

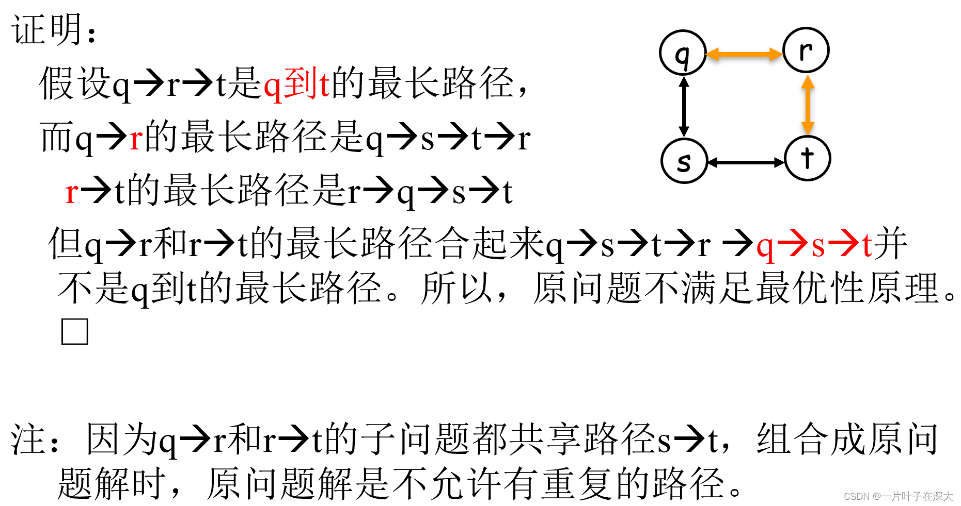
**最优性原理**

求解问题的一个最优策略的子策略序列总是最优的，则称该问题满足最优性原理。

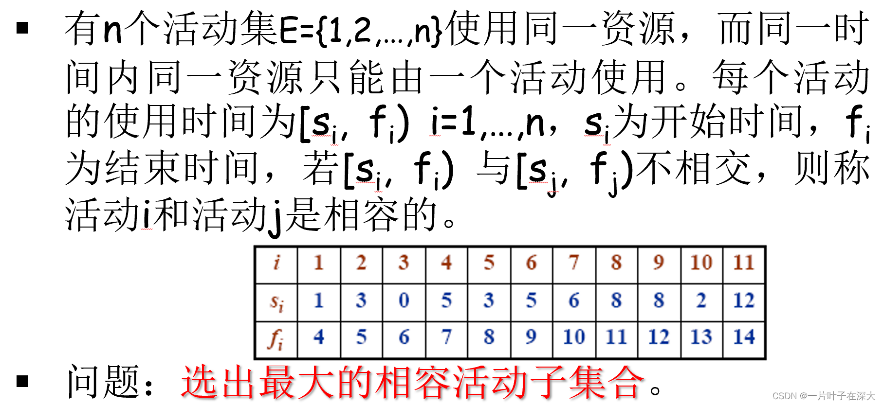
**证明：0-1背包问题Knap(1,n,c)满足最优性原理**

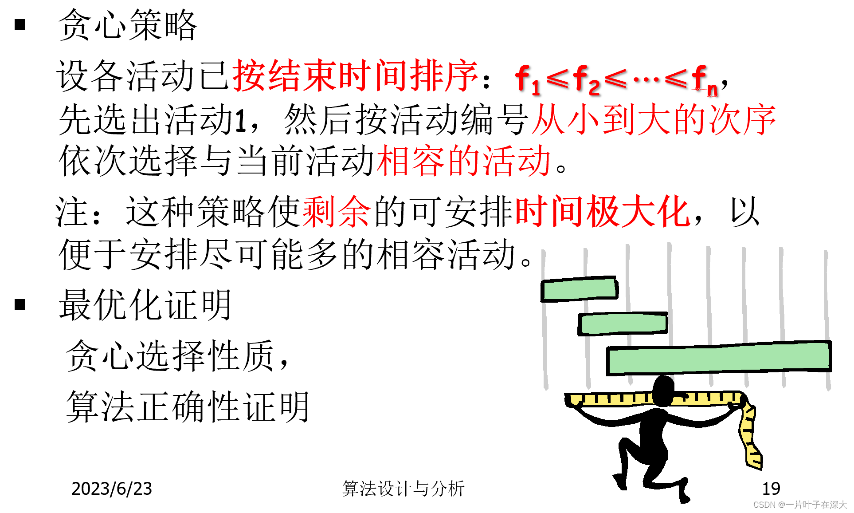
​

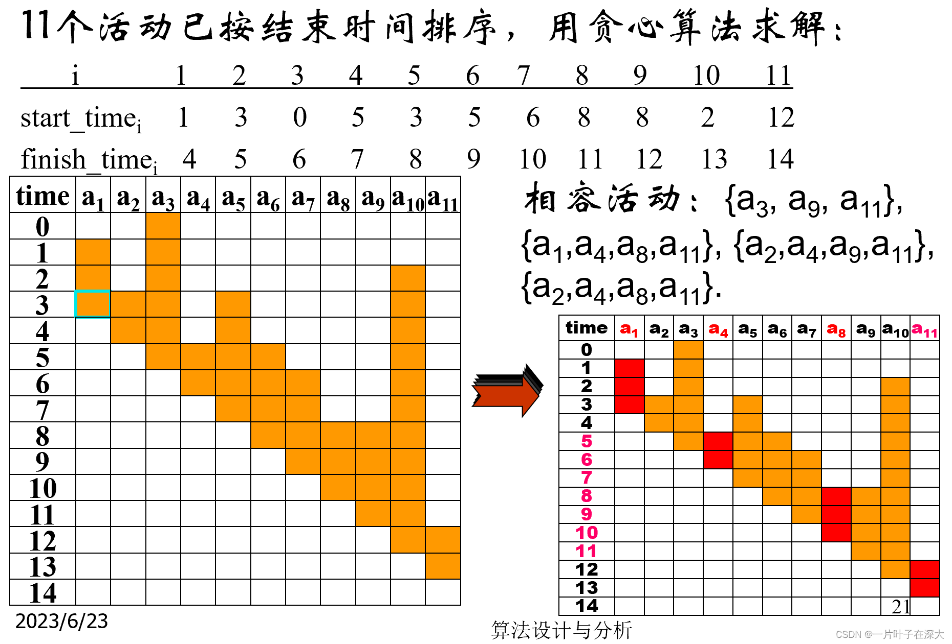
**证明：最长路径问题不满足最优性原理。**



**活动选择问题**

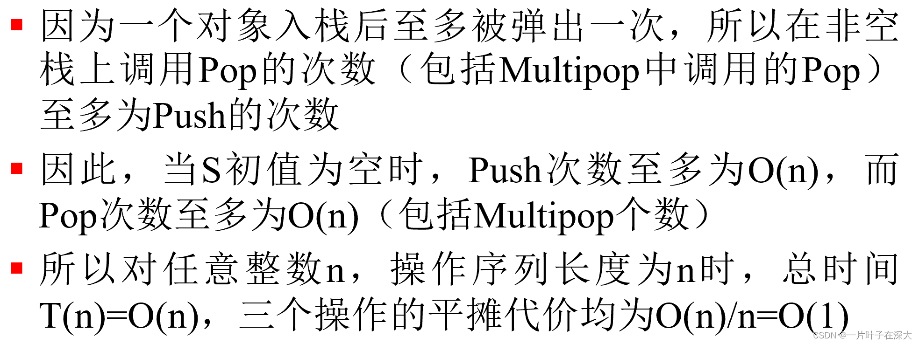
​

​

​

**聚合法/合计法**

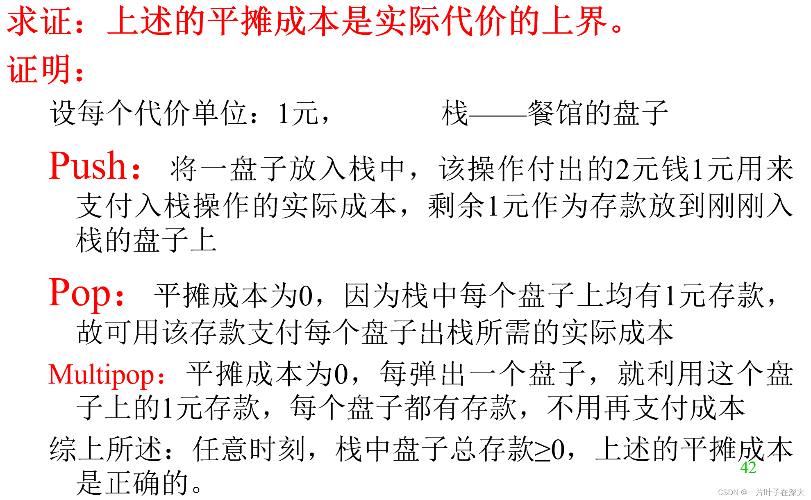
栈操作分析

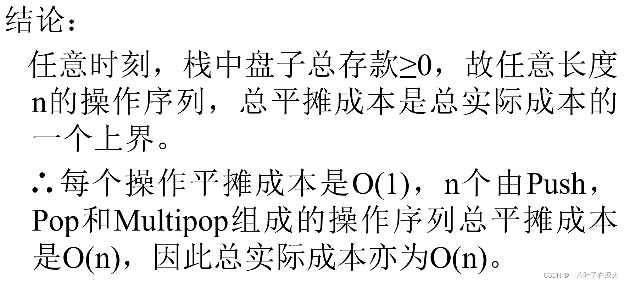
​

**核算法/记账法**

栈操作

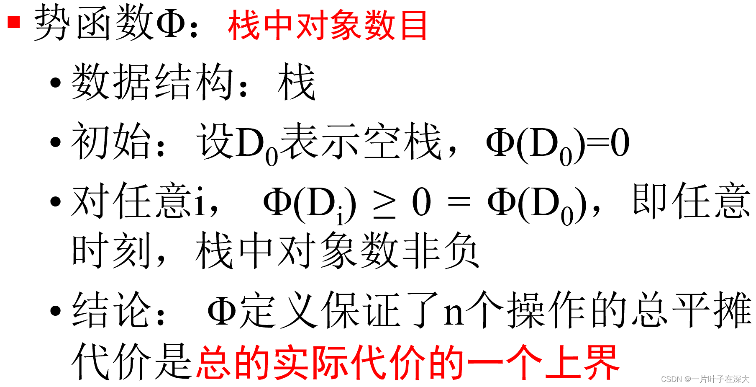
​

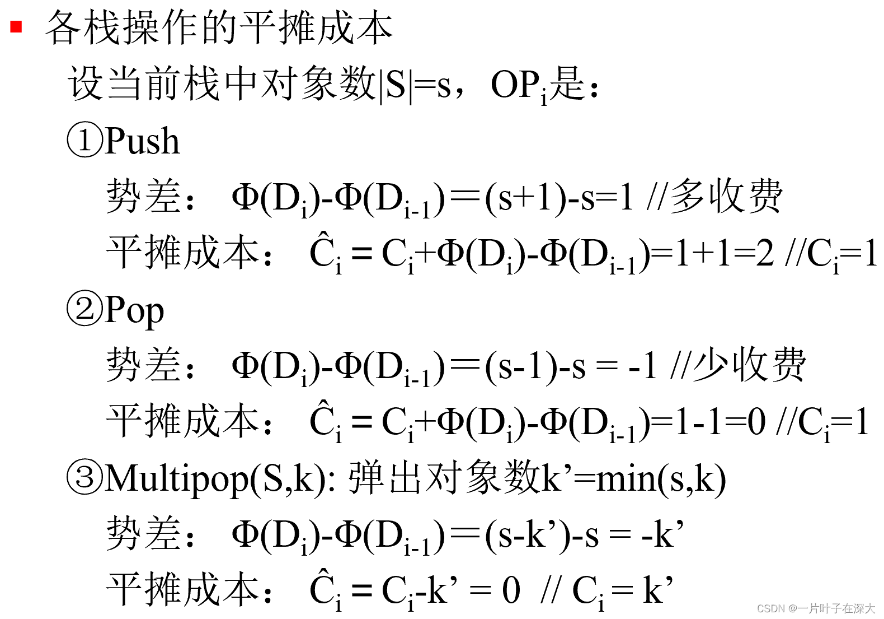
​

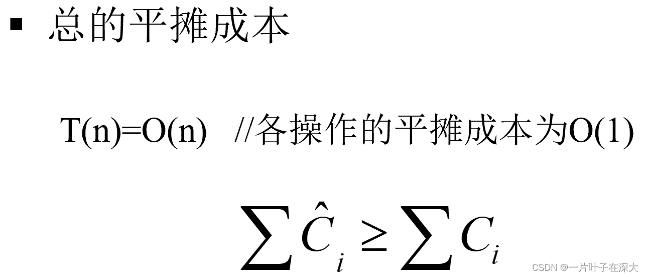
​

**势能法**

栈操作

​

​

​

**图**

入度：有向图中连向该节点边的条数。

出度：有向图中从该节点连出的边的条数。

度：节点出度与入度之和，即连接该节点边的条数。

简单图：没有多重边，没有自环。

简单路径：对于一条由连续边与节点组成的路径，没有经过重复的节点。

连通图：对于一个无向图，任意两个节点之间都存在一条路径连接。

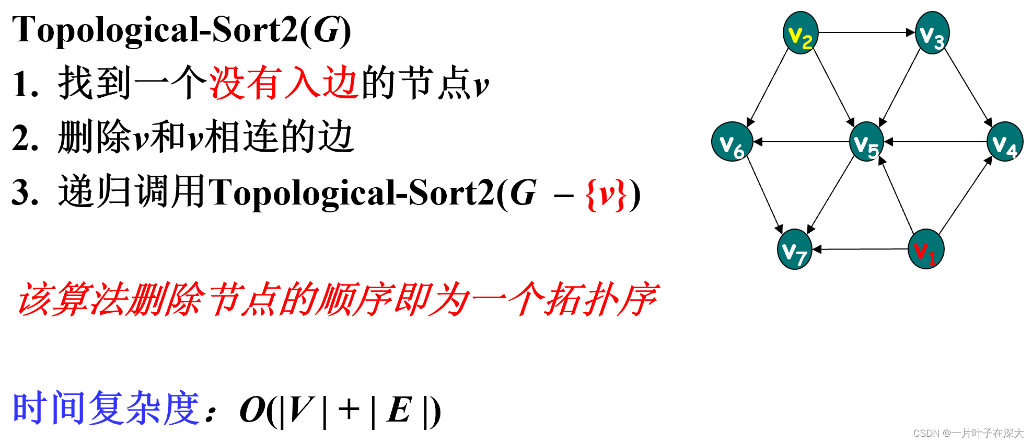
强连通图：对于一个有向图，任意2个节点之间都存在一条有向路径连接。

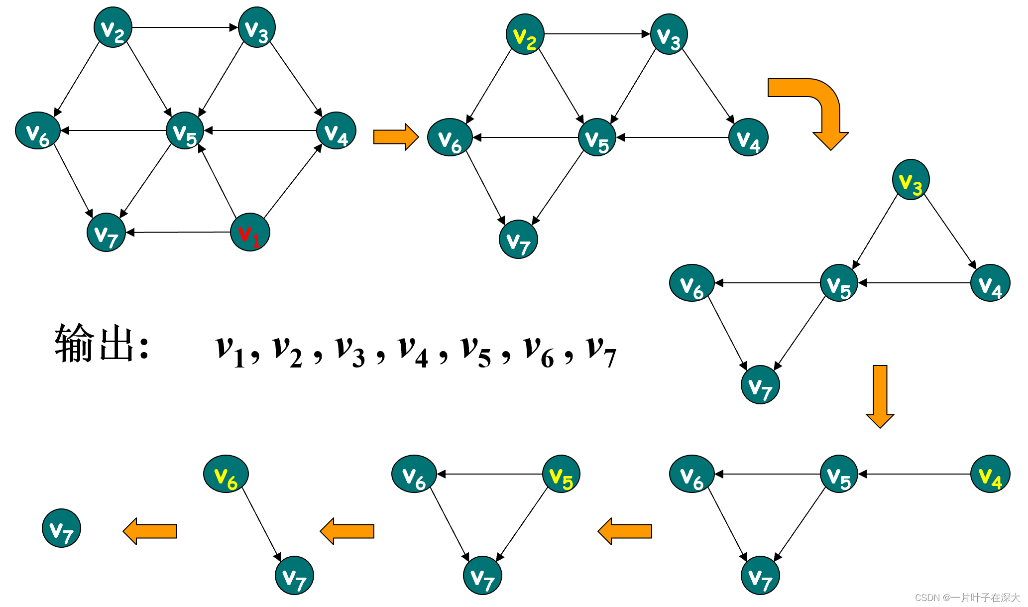
稀疏图：|E|≈|V|

稠密图：|E|≈|V|²

完全图：对于一个有向或者无向图，任意两个节点之间都有边邻接（对于有向图需要两个方向 的边）。

**拓扑排序**

​编辑

​

**Kruskal算法**

克鲁斯卡尔算法的思想是逐步选择边来构建最小生成树。具体步骤如下：

1. 将图中的所有边按照权值从小到大进行排序。
2. 从权值最小的边开始，依次考虑每条边：
   * 如果该边的两个顶点不在同一个连通分量中，则选择该边加入最小生成树，并合并这两个顶点所在的连通分量。
   * 如果该边的两个顶点已经在同一个连通分量中，则舍弃该边，以避免形成环路。
3. 重复步骤2，直到最小生成树中包含图中的所有顶点为止。

通过这种方式，克鲁斯卡尔算法能够找到一个连通图的最小生成树，并且保证总权值最小。算法的关键在于选择边的过程中保证不会形成环路，以确保最终生成的树是连通的。

**prim算法**

Prim算法的思想如下：

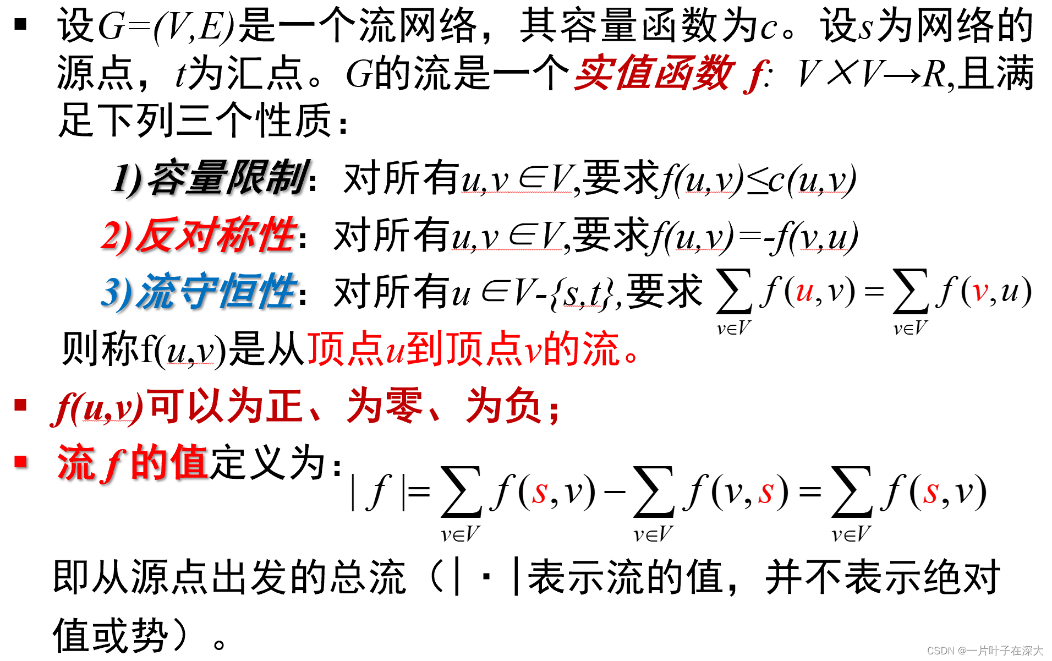
1. 选择一个起始顶点作为初始集合，可以是任意一个顶点。
2. 将该起始顶点加入到最小生成树的顶点集合中。
3. 从已加入最小生成树的顶点集合中，选择一个顶点u，将与顶点u相连且权值最小的边（u, v）加入到候选边集合。
4. 从候选边集合中选择权值最小的边（u, v），将顶点v加入到最小生成树的顶点集合中，同时将边（u, v）加入到最小生成树的边集合中。
5. 重复步骤3和步骤4，直到最小生成树包含图中的所有顶点为止。

通过这种方式，Prim算法逐渐扩展最小生成树的顶点集合，保证每一步都选择了与已加入顶点集合具有最小权值的边。最终得到的最小生成树是以起始顶点为根节点的一棵树，并且总权值最小。

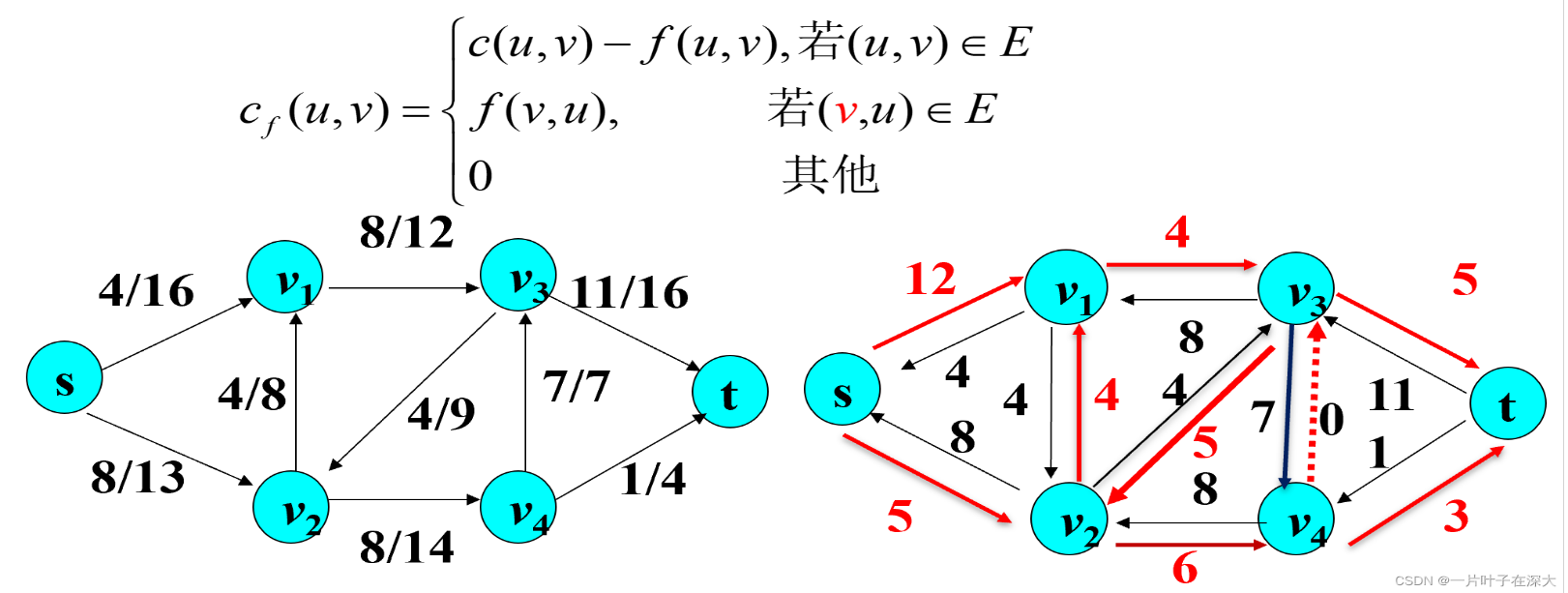
需要注意的是，Prim算法的实现通常需要使用优先队列（最小堆）来高效地选择权值最小的边。

**流网络**

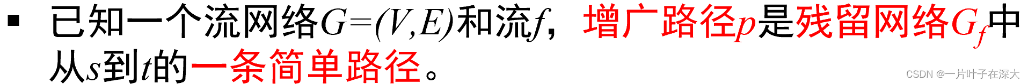
流网络是一个有向图*G=*（*V，E*），其中每条边(*u,v*)均有一**非负容量***c*(*u,v*)**≥**0。如果(*u,v*)不是*E*中的边，则假定*c*(*u,v*)=0。流网络中有两个特别的顶点：***源点s***和***汇点t***。假定每个顶点均处于从源点到汇点的某条路径上。

​

**残留网络**



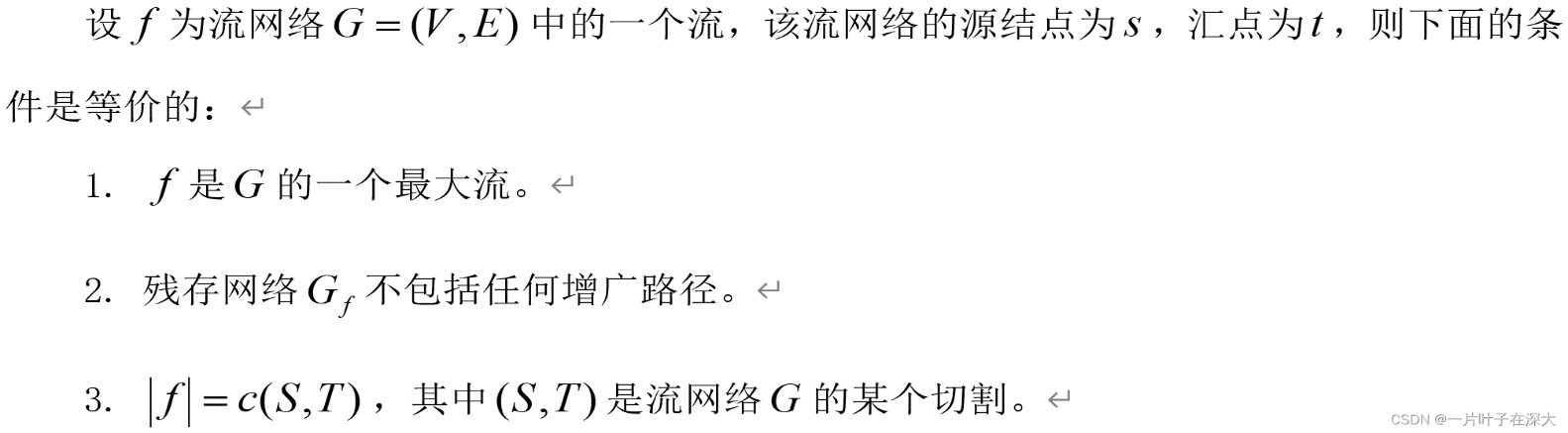
**增广路径**

​

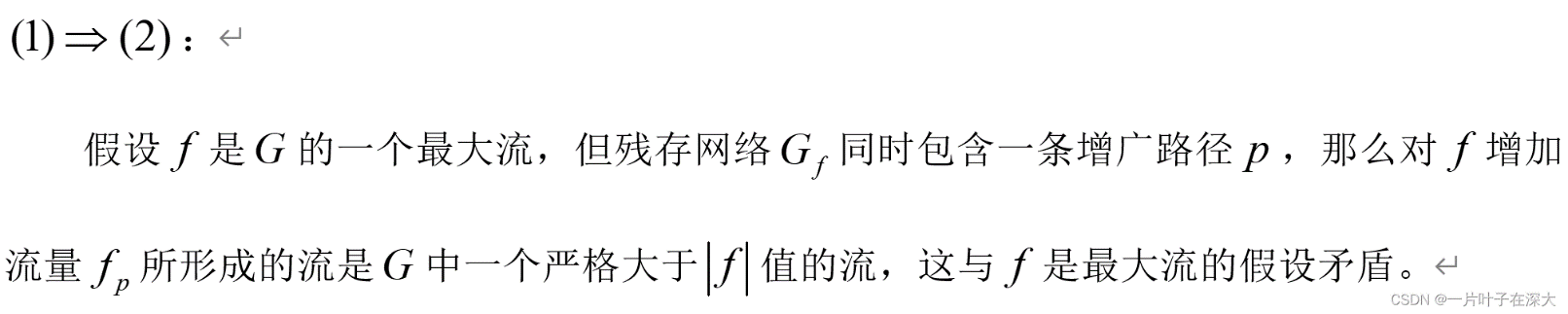
**最小割**

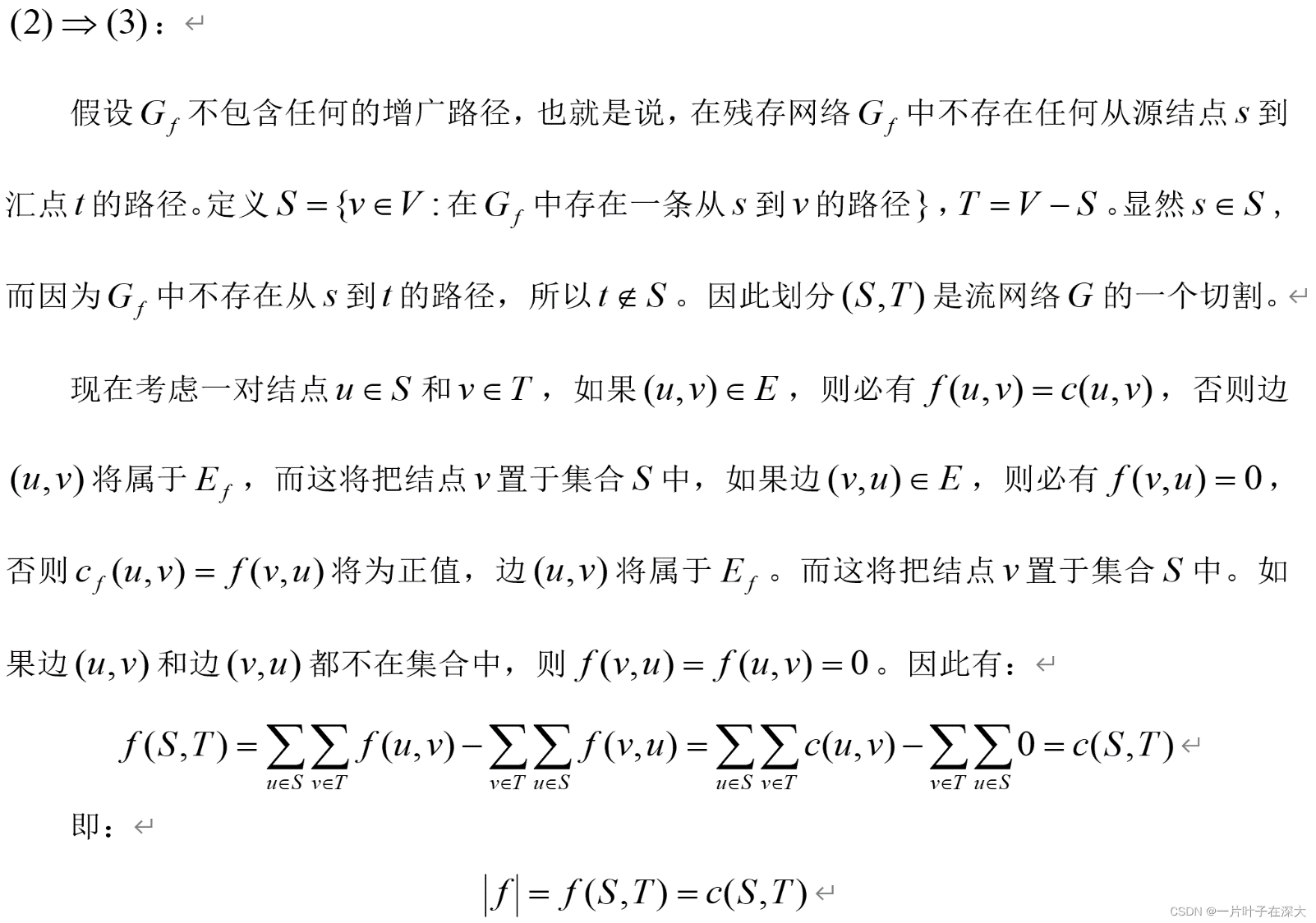
指将原有网络G(V, E)划分成两个不相交的集合(A, B)，使得A中的所有节点都无法到达B中的所有节点，在满足这一条件的情况下，将划分这两个集合的所有边的容量之和称为最小割。

**最大流最小割定理**



**最大流最小割定理的证明**

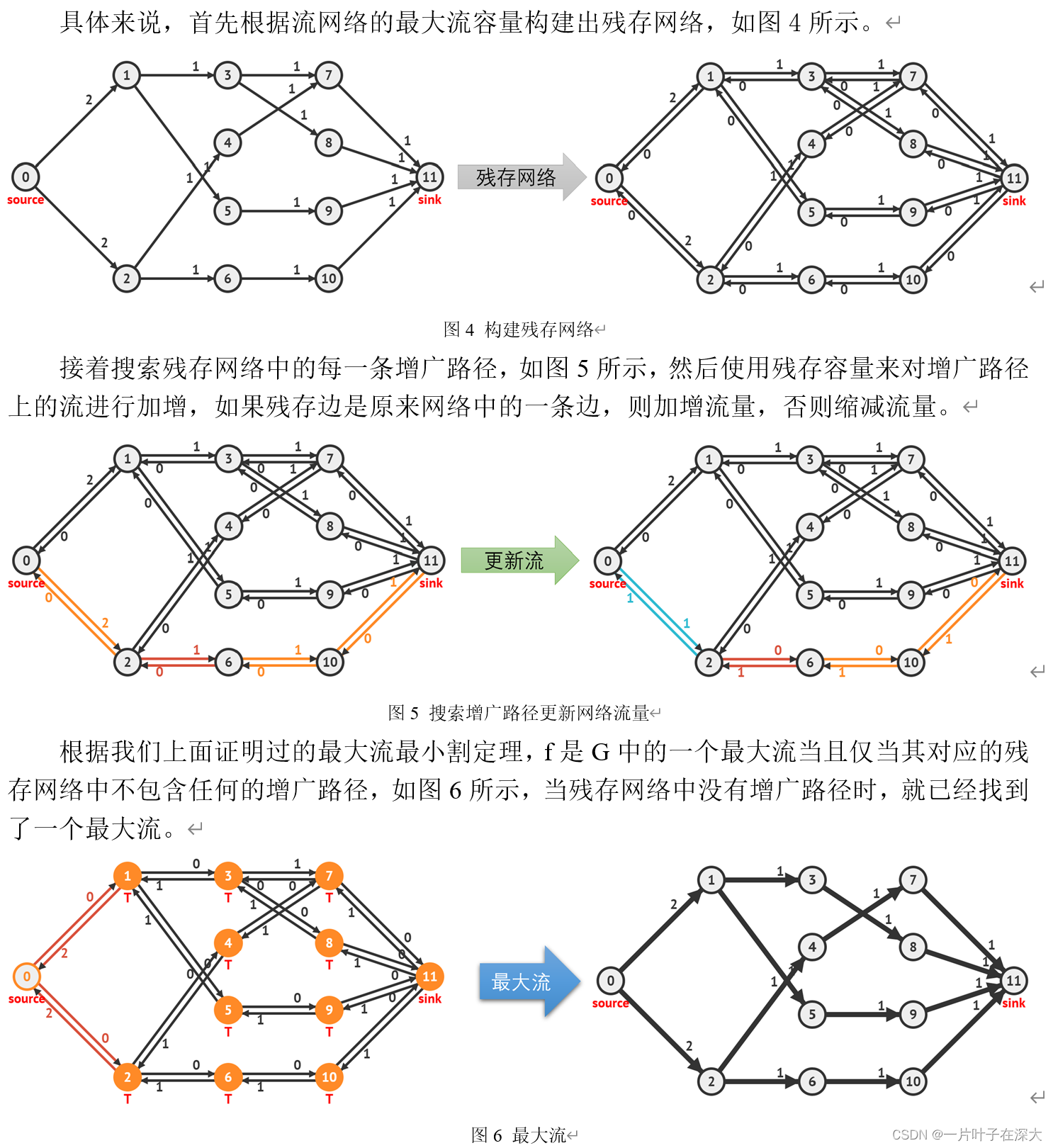
​





**Ford-Fulkerson方法**

Ford-Fulkerson方法通过不断地在残留网络中搜索出增广路径，并根据增广路径更新剩余容量的方式来寻找最大流。每次增广的过程中，都会选择一条从源点到汇点的路径，然后将这条路径上的流量增加到当前的最大流中。随着可行流的不断增加，残留网络中的剩余容量也不断减少，直到找不到增广路径为止。



**P问题和NP问题**

在计算机科学中，问题可以分为两类：P问题和NP问题。P问题是可以在多项式时间内解决的问题，也就是说存在一种算法，以输入规模的多项式函数形式运行时间来解决该问题。而NP问题则是指可以在多项式时间内验证解的问题，也就是说如果给定一个解，可以在多项式时间内验证这个解是否正确。

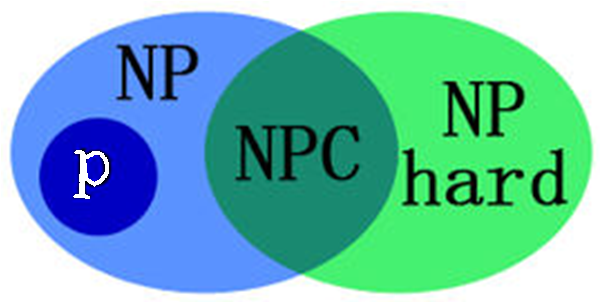
经典的NP问题包括旅行商问题（TSP）、背包问题（Knapsack Problem）、图着色问题（Graph Coloring Problem）等。这些问题在实际中非常重要，但目前还没有找到一种高效的确定性算法来解决它们。如果能够在多项式时间内找到NP问题的解，那么P问题和NP问题将等价，这是一个著名的数学难题，被称为P与NP问题的克里伯尔猜想。

NPC指的是**NP问题中最难的一部分问题**，所有的NP问题都能在**多项式时间内归约到NPC**上。

**可能**存在一些连**验证解都不能多项式解决的问题**，这些就是**NP-hard**问题。

所有的NP问题都能规约到NP-hard问题，但它不一定是NP问题。

从直觉上说，P<=NP<=NP-Complete<=NP-Hard，问题的难度递增。



**停机问题**

停机问题（Halting problem）是指确定一个给定的计算机程序能否在有限步骤内停止运行的问题。简而言之，停机问题是要判断一个程序是否会在某个时刻停止执行，或者会一直进行下去。

停机问题被证明是不可解的，也就是说，不存在一种通用算法可以判断任意程序是否会停止。这个结论由图灵在1936年提出，并通过其著名的停机问题证明（Turing's halting problem proof）得到了证实。

停机问题的重要性在于它揭示了计算机理论中的局限性。尽管现代计算机具有强大的计算能力，但某些问题是无法通过计算机自动解决的。停机问题成为计算机科学中的一个经典例子，被广泛应用于理论计算机科学和计算机算法的研究中。

**停机问题的证明**是通过构造反证法来展示停机问题的不可解性。这个证明由图灵在1936年提出，被称为图灵停机问题证明（Turing's halting problem proof）。

证明的思路如下：

1. 假设存在一个算法或程序H，可以判断任意给定程序是否会在有限步骤内停止。
2. 假设程序H接收两个输入：P（要判断是否停机的程序）和I（程序P的输入）。
3. 程序H首先尝试运行程序P并观察它的行为。如果程序P在有限步骤内停机，则程序H返回"停机"。否则，程序H进入一个无限循环。
4. 接下来，构造一个新的程序D。程序D的功能是根据输入参数来模拟程序H的行为。即，当程序D接收到输入P和I时，它会调用程序H并将输入设置为P和I。如果程序H返回"停机"，那么程序D会进入一个无限循环；如果程序H进入无限循环，那么程序D会停机。
5. 现在，我们将程序D作为自己的输入参数传递给程序D。也就是说，我们运行程序D，并将D作为输入传递给D。
6. 根据程序D的定义，它会模拟程序H的行为。如果程序H（此时是D自己）返回"停机"，那么程序D会进入无限循环；如果程序H进入无限循环，那么程序D会停机。
7. 这就导致了矛盾：根据程序D的行为，无论它是停机还是进入无限循环，都会与程序H的判断相矛盾。

由此可见，假设存在一个算法或程序H来解决停机问题是不成立的。因此，停机问题是不可解的。这个证明揭示了计算机无法判断任意程序是否会停机的困境，成为计算机科学中的经典结果之一。