算法复习

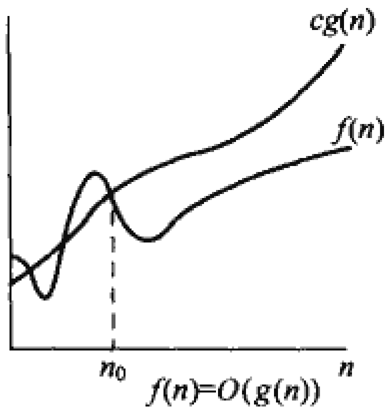
第一章：复杂度分析

输入规模

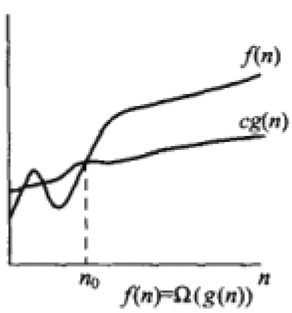
- 输入规模不是指输入数据的个数，而是数据在**内存中所占空间的大小**
- 通常情况下，计算数据存储位数的结果可以**约等于输入数据的个数**
- 对于大数运算，不能只考虑输入数据的个数，也要考虑**输入数据的长度**

各类复杂度

O：上界（**小于等于，最坏复杂度**）

> 当且仅当存在正的常数C和n0，使得对于所有的n>=n0，有f(n)≤Cg(n)
>
> 
>

Ω ：下界（**大于等于**）

> 当且仅当存在正的常数C和n0，使得对于所有的n>=n0，有f(n)≥Cg(n)
>
> 
>

θ：确界（等于）

> 当且仅当存在正的常数C1和C2和n0，使得对于所有的n>=n0，有 C1g(n)≤ f(n) ≤ C2g(n)

影响时间复杂度的因素

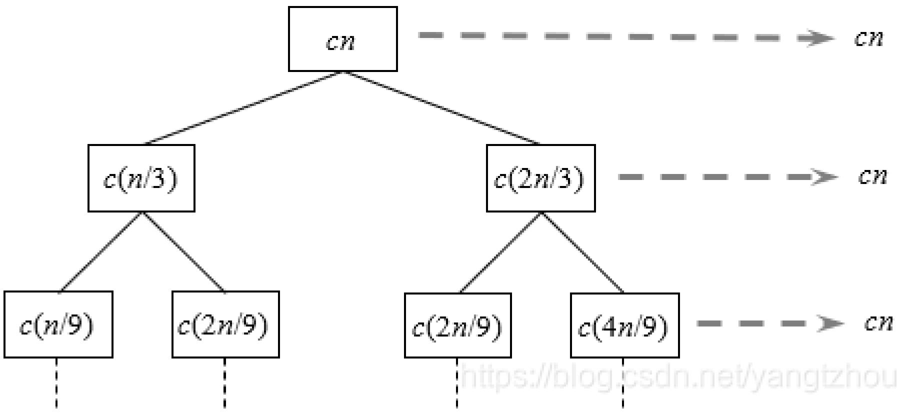
- 算法执行次数
- 数据规模

- 算法基本操作
- 输入数据特征（是否有序）

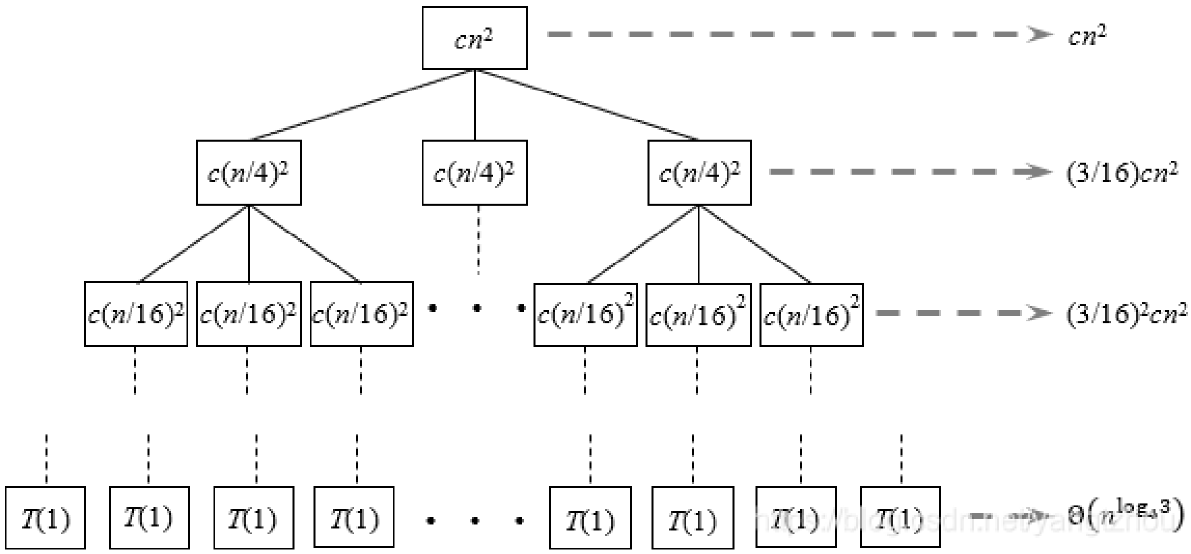
复杂度计算方法

递归树

eg.$T(n) = T(n/3) + T(2n/3) + cn$



eg.$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$



主定理法

T(n) = a T(n/b) + f(n)

计算**logb a**，和**k**比较

- **如果f(n) = O(n^k) 且k < logb a, T(n) =** $\theta(n^{log_b a})$
- **如果f(n) = Θ(n^k) 且k = logb a, T(n) =** $\theta(n^k logn)$
- **如果f(n) = Ω(n^k) 且k> logb a, T(n) =** $\theta(f(n))$

主定理法扩展

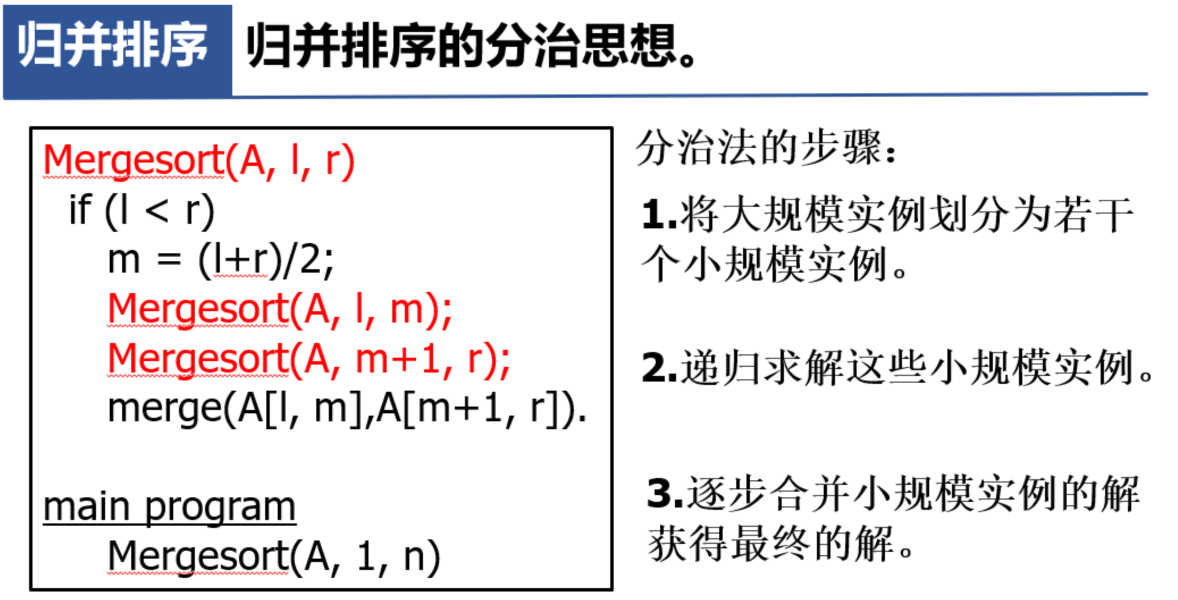
对于常数c和a1,...,ak 满足**a1+...+ak <1**, 如果有递归式**T(n) ≤ T(a1n)+T(a2n)+...T(akn)+cn**, 则**T(n) = O(n)**.

第二章：分治

归并排序

时间复杂度：T(n) = 2·T(n/2) + O(n)

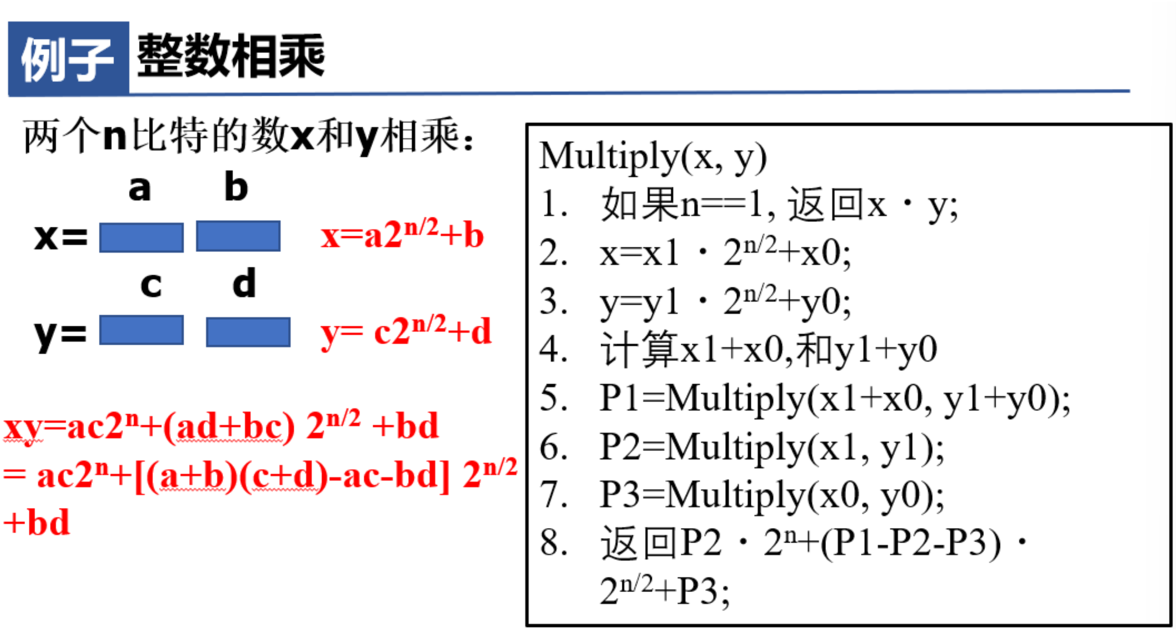
T(n)=O(nlogn)



整数相乘

时间复杂度：T(n) = 3·T(n/2) + O(n)

T(n)=O($n^{log_2 3}$)



Strassen矩阵相乘

时间复杂度：T(n) = 7·T(n/2) + O(n^2)

T(n)=O($n^{log_2 7}$)

例子 斯特拉森（Strassen）矩阵相乘

两个**n x n** 的矩阵**A**，**B**相乘，**n**为**2**的次幂：

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

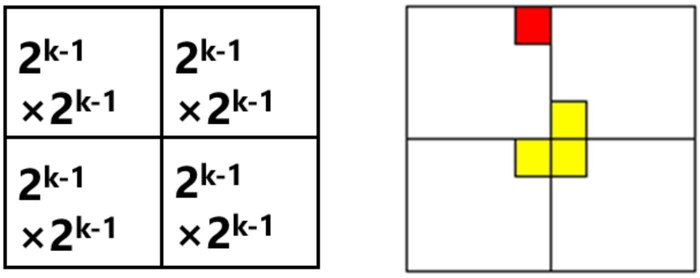
$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

- $M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$
- $M_2 = (A_{10} + A_{11}) * B_{00}$
- $M_3 = A_{00} * (B_{01} - B_{11})$
- $M_4 = A_{11} * (B_{10} - B_{00})$
- $M_5 = (A_{00} + A_{01}) * B_{11}$
- $M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$
- $M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

棋盘覆盖

例子 棋盘覆盖

方法：如果k>0，使用一个L型骨牌覆盖在中间，将$2^k \times 2^k$的棋盘划分为4个大小$2^{k-1} \times 2^{k-1}$为的棋盘。



(a) (b)

程序输入参数:

(1) 左上网格的行号tr;
(2) 左上网格的列号tc;
(3) 特殊网格的行号dr;
(4) 特殊网格的列号dc;
(5) 棋盘的大小size。

```
chBoard(tr, tc, dr, dc, size)
 if (size == 1) return;
 tile++;
 s = size/2;
// 覆盖左上网格
 if 包含特殊网格:
   chBoard(tr, tc, dr, dc, s);
 else
   Board[tr+s-1][tc+s-1]=t;
  chBoard(tr, tc, tr+s-1, tc+s-1, s);
```

```
// 覆盖右上网格
 if 包含特殊网格
   chBoard(tr, tc+s, dr, dc, s);
else
   Board[tr+s-1][tc+s]=t;
  chBoard(tr, tc+s, tr+s-1, tc+s, s);
```

```
// 覆盖左下网格
If 包含特殊网格
   chBoard(tr+s, tc, dr, dc, s);
else
   Board[tr+s][tc+s-1]=t;
   chBoard(tr+s, tc, tr+s, tc+s-1, s);
```

```
// 覆盖右下网格
if 包含特殊网格
   chBoard(tr+s, tc+s, dr, dc, s);
else
   Board[tr+s][tc+s]=t;
   chBoard(tr+s, tc+s, tr+s, tc+s, s);
```

递归关系：$T(k) = 4T(k-1)+O(1)$

$T(0) = 1$

时间复杂度：$O(4^k)$

寻找第k小元素

例子 元素查找

给定数组**a[]**，**a[]**中存有**n**个具有不同值的元素，给定整数**k**，**1 <= k <= n**，以**O(n)**的时间复杂度查找**a[]**中第**k**小的元素。

划分：
以**x**为参考元素，将数组**a[]**划分为左右两个子数组，使得左边数组中的元素小于等于**x**，右边数组中的元素大于等于**x**。

- **快速选择算法**

```
Find_kth_element(A, p, q, k)
  if (p = q) return a[p];
  r = Partition(A, p, q);
  j=r-p+1;
  if k≤j  return Find_kth_element(A, p, r, k);
  else  return Find_kth_element(A, r+1, q, k-j);
```

最差情况下的时间复杂度：$O(n^2)$

目标：最差情况下的时间复杂度到**O(n)**

- **线性时间选择法**

(1) 将a[]中的元素划分到 $\lceil n/5 \rceil$ 组中；

(2)使用任意排序算法对每一组中的元素排序；

(3)找到每一个组中的中值元素，共有 $\lceil n/5 \rceil$ 个中值元素；

(4) 使用M表示中值元素的集合, |M|= $\lceil n/5 \rceil$，在M中求解中值元素，表示为x；

(5) 使用x作为参考值进行划分。

Select(a[],k) //T(n)

1. 如果a[]的长度小于75, 使用任意一种排序算法对a[]进行排序;
2. 将a[]划分到M=$\lfloor n/5 \rfloor$组中, 每组中都有5个数;
3. 找出每一组中的中值组成集合M;
4. x←Select(M,$\lceil |M|/2 \rceil$); //**T(n/5)**
5. 将a[]中的元素与x进行比较;
   将小于x的元素放入集合S1中,将大于x的元素放入集合S2中；
6. 如果k ≤ |S1|，Select(S1,k) //**T(|S1|)**
7. 否则Select(S2,k-|S1|); //**T(|S2|)**

(1)在每一组中有两个元素小于该组的中值。

(2)在$\lfloor n/5 \rfloor$个中值中，最少有$3\lfloor (n-5)/10 \rfloor$个元素小于x。

如果n ≥75，$3\lfloor (n-5)/10 \rfloor \geq n/4$。

$$T(n) \leq \begin{cases} C_1 & n < 75 \\ C_2 n + T(n/5) + T(3n/4) & n \geq 75 \end{cases}$$

T(n)=O(n)

每组中的元素数量是5，75是递归的关键点。

n/5+3n/4=19n/20=εn， 0<ε<1

除了5和75以外，还有其它值的设置选择。

最近点对问题

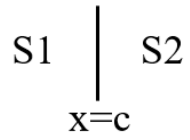
时间复杂度：T(n) = 2·T(n/2) + O(n)

T(n)=O(nlogn)

例子 最近点对问题

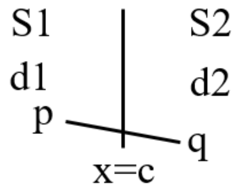
给定平面上n个点的集合S，其中n为2的次幂，每个点分别表示为c1=(x1, y1), c2=(x2, y2), …, cn=(xn, yn)，这些点按对应的x坐标以升序排列，找出具有最近距离的一对点。

方法:

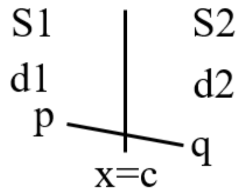


(1) 将点集S划分为两个子集S1, S2, |S1| = |S2| = n/2, 绘制一条垂直线x=c。

(2) 在S1和S2中分别递归的求解最小距离，设为d1和d2。

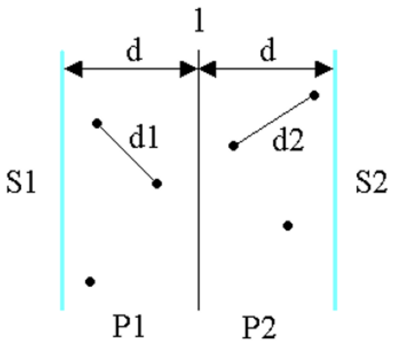


假设**d=min{d1, d2}，S**的最小距离是**d** 或者样本对**(p, q)**之间的距离**,** 其中**p∈S1, q∈S2**。



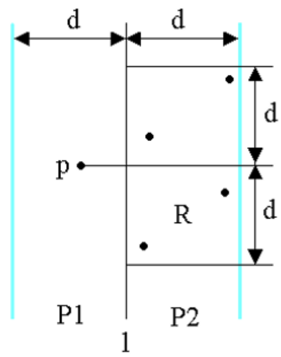
(1) p和q应该在什么位置?

p和q 位于宽度为2d的垂直条带中。



(2)对于集合P1中的固定点p, P2中的哪个点可以被用来与p计算最小距离?

由于距离(p, q)<d, 所有满足条件的点一定在d×2d的矩阵上。



- 在d×2d的矩阵中**最多包含6个点**

MinDist(S)
1. 如果|S|≤3,计算所有样本对之间的距离;
2. 找出这些点中x坐标的中值m;
3. 使用m将S划分到S1和S2中，其中S1中的x坐标小于等于m, S2中的x坐标大于m;
4. d=min{MinDist(S1), MinDist(S2)};
5. B_{s1} =与m的距离小于等于d的S1中的点;
6. B_{s2} =与m的距离小于d的S2中的点;
7. 根据y坐标对B_{s2}排序;
8. 对 B_{s1}中的点p，通过二分搜索在B_{s2}中对应的d ×2d范围内搜索最大y值的q ∈ B_{s2}，从q开始，最多和p比较6个点。

EX1：堆排序

堆排序的性质

具有n个节点的堆的高度≤ $\log_2 n$

FixHeap的时间复杂度为O(log n)

MakeHeap的时间复杂度为 O(n)

堆排序的时间复杂度为 O(n log n)

堆排序的过程

建立堆，交换根节点和最后一个叶子节点的值，调整堆

- 定理
  - **基于比较的排序算法的时间复杂度是Ω(n log n)**
  - **基于比较的查找算法的时间复杂度是Ω(log n)**

EX2：线性排序算法

计数排序

T(n)=O(n+k)

6.1计数排序 问题和思路

思路: 对于每个元素A[i], 统计A[0..n-1] 中不大于A[i]的元素数量，表示为h_i 。在输出结果中，A[i]应当位于 h_i的位置上。

算法： CountingSort

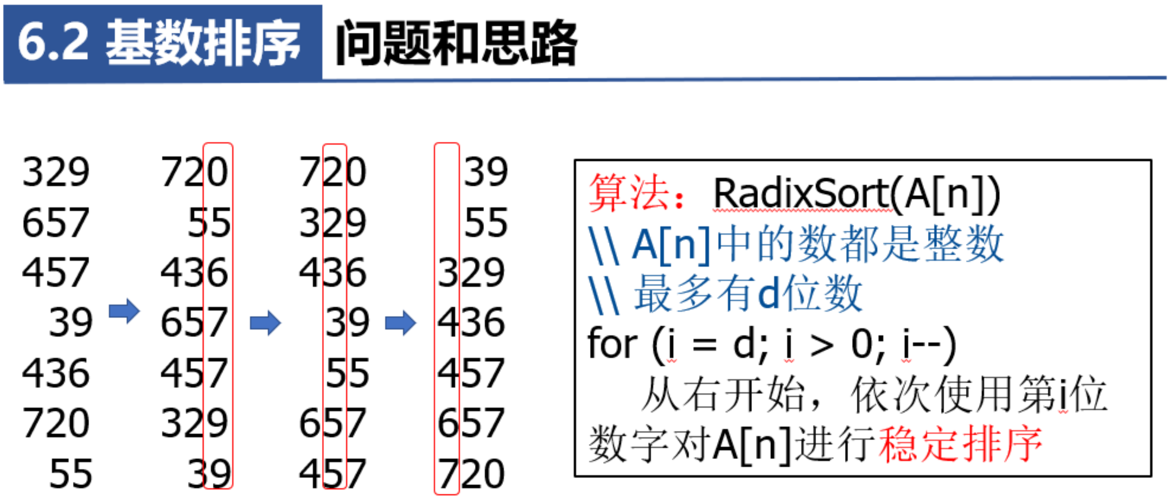
Step 1： 对于每一个值v, 统计A[0..n-1]中v出现的次数。

Step 2： 对于每一个A[i], 统计A[0..n-1]中不大于A[i]的元素数量。

Step 3. 使用步骤2的信息输出有序列表。

基数排序

T(n)=O(d*n)



> 计数排序是**稳定排序**算法，基数排序也是**稳定排序**算法。
>
> 稳定排序算法：冒泡排序，插入排序，归并排序。
>
> 不稳定排序算法：快速排序，堆排序，选择排序。

第三章：贪心

特点

- 贪心选择：
  - 整体的最优解可通过一系列局部最优解达到
  - 每次的选择可依赖以前作出的选择，但不依赖后续选择
- （完美）最优子结构：问题的整体最优解中包含子问题的最优解

案例

- 活动选择
- 任务调度
- 哈夫曼编码

贪心与动态规划的区别

- 相同点：都是推导算法，两者都具有最优子结构性质
- 不同点：

  贪心算法：

  - 贪心算法中，做出的每步贪心决策都没法改变，由于贪心策略是由**上一步的最优解推导下一步的最优解，而上一步以前的最优解则不做保留。**
  - 贪心法正确的条件是：**每一步的最优解必定包含上一步的最优解。**
  - 贪心是**自顶向下**的，当前的求解需依赖于下面全部子树的状况，只需要在当前选择局部最优解即可
  - **贪心只有一个子问题**

  动态规划：

  - 全局最优解中必定包含某个局部最优解，但不必定包含前一个局部最优解，所以**须要记录以前的全部最优解** （无后效性）

- 动态规划的关键是状态转移方程，即如何**由已求出的局部最优解来推导全局最优解。**
- 动归是**自底向上**的，通过子节点求出根节点，相对于贪心算法来说**开销比较大。**
- **动态规划有多个子问题**

第四章：动态规划

特点

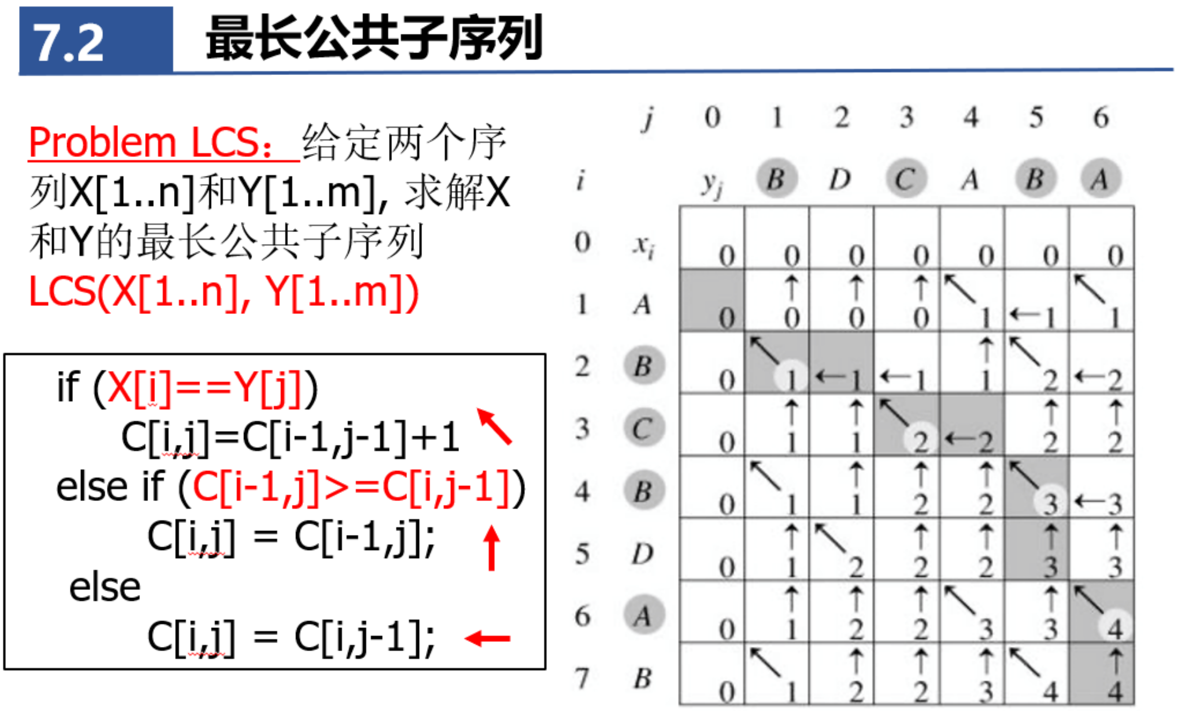
- 具有最优子结构
  - 子问题的最优解可以用于求解原始问题的最优解
- 具有重叠子问题
  - 子问题不断重复出现

实现dp的方法

- 自顶向下（递归求解子问题）
  - 和分治的不同点：避免多次重新计算相同的子问题
- 自底向上

最长公共子序列

T(n)=O(n*m)



0/1背包问题（!!!）

- 子问题：OPT(i, w) = 可装物品为1, ..., i，约束重量不超过w时，背包问题的最优值
- **T(n)=O(n*W)**

Case1:如果wi > w, OPT(i, w) = OPT(i-1, w)

Case2:如果wi <= w:

如果第i个物品不装入包里，OPT(i, w)在{ 1, 2, ..., i – 1 }中选择重量之和小于w的物品使得价值最优；OPT(i,w)=OPT(i-1,w)

如果第i个物品装入包里，OPT(i, w)在{ 1, 2, ..., i – 1 }中选择重量之和小于w-wi的物品使得价值最优；OPT(i,w)=OPT(i-1,w-wi)+vi

- 完全背包问题（每个物品可以取无限次）：就是在wi和vi前面都乘上一个k

加权区间调度

T(n)=O(nlogn)

> 如果所有工作的**报酬都相同**，则直接利用**贪心算法**即可得到结果；
>
> 如果不同工作的**报酬不同**，则无法直接通过贪心算法得到结果；

- 按工作的完成时间以升序进行排序

定义： OPT(j) = 在工作1, 2, ..., j之间，选择可完成的工作所获得的最大报酬；
目标： OPT(n) =所有工作之间，选择可完成的工作所获得最大报酬；
Case 1： OPT(j)没有选择工作j:
- 在工作1, 2, ..., j – 1之间选择可完成的工作所获得的最大报酬；

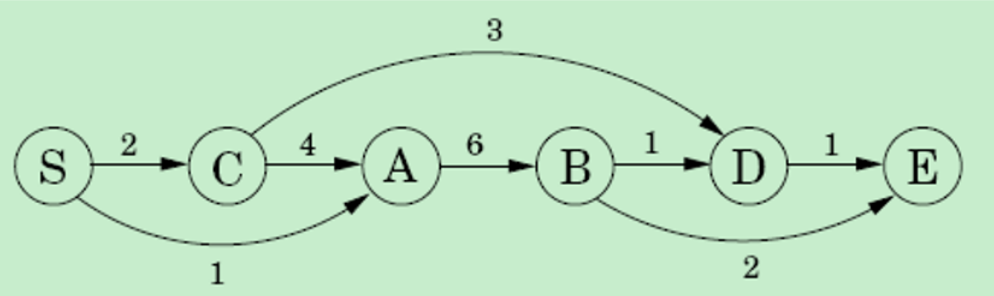
Case 2： OPT(j)选择工作j:
- 获得报酬wj；
- 不能选择OPT(j)之外的其它工作{ p(j) + 1, p(j) + 2, ..., j – 1 }；
- 必须在工作1, 2, ..., p(j)之间选择可完成的工作获得最大报酬；

$$OPT(j) = \begin{cases} 0, & \text{if } j = 0; \\ \max\{OPT(j-1),\ w_j + OPT(p(j))\}, & \text{if } j > 0; \end{cases}$$

	Interval-Scheduling(n, s1...sn, f1...fn,w1...wn)
O(n log n)	1. 根据工作的完成时间进行排序，并重新编号使得 f1<=f2<=...<=fn;
O(n log n)	2. 计算p[1], p[2],...,p[n]; 二分搜索
O(1)	3. Opt[0] = 0;
O(n)	4. for(j = 1; j<= n; j++)
O(1)	5. Opt[j] = max{Opt[j-1], wj+ Opt[p[j]]};
	6. return Opt[n].

有向无环图（DAG）中的最长路径问题

给定一个加权的有向无环图G=(V,E,W)，求图G中的最长路径。



动态规划方程：

对于任意顶点v，$dilg(v)=\max_{(u,v)\in E}\{dilg(u)+w(u,v)\}$

Dplongestpath(G)

1. 初始化所有节点 dilg(.)的值为-1;
2. S是所有入度为0的节点的结合;
3. for 节点v in S do
   dilg(v)=0;
4. For $v\in V\backslash S$ 按拓扑排序 do
   $dilg(v)=\max_{(u,v)\in E}\{dilg(u)+w(u,v)\}$
5. Return dilg(.)的最大值和对应的节点。

编辑距离问题

T(n)=O(n*m)

给定两个字符串x[1..n], y[1..m], 找出x和y之间的编辑距离(E(n, m)).

对齐：将一个字符串列在其它字符串的上面;

x = SNOWY　y = SUNNY

S	–	N	O	W	Y
S	U	N	N	–	Y

–	S	N	O	W	–	Y
S	U	N	–	–	N	Y

"－"表示"可编辑";

任意可编辑的位置都可以使用其它的字符代替;

对齐字符串所需要修改的字符数是具有不同字母的列的数量;

给定两个字符串x[1..n], y[1..m], 找出x和y之间的编辑距离(E(n, m))。

子问题：E[i,j]

$$\begin{matrix} x[i] \\ - \end{matrix} \quad \text{or} \quad \begin{matrix} - \\ y[j] \end{matrix} \quad \text{or} \quad \begin{matrix} x[i] \\ y[j] \end{matrix}$$

E(i, j) E(i-1, j)	E(i, j) E(i, j-1)	E(i, j) E(i-1, j-1)
↓关系	↓关系	↓关系
E(i,j)=1+E(i-1,j)	E(i,j)=1+E(i,j-1)	E(i,j)=1/0+E(i-1,j-1)

$$E(i,j) \;=\; \min\{1+E(i-1,j),\ 1+E(i,j-1),\ \text{diff}(i,j)+E(i-1,j-1)\}$$

如果x[i]=y[j]则diff(i, j)=0, 否则diff(i, j)=1;

E(0, j)=j; E(i, 0)=i;

旅行商问题（TSP）

[旅行商问题（动态规划方法，超级详细的）-CSDN博客]

动态规划与分治的区别

- 相同点：都是把原问题划分为多个子问题求解，都是将大问题拆分成小问题逐个击破
- 不同点：
  - **动态规划的子问题并不是互相独立，是有交集的；分治的子问题是相互独立的**（独立指的是子问题只用来求解一个问题，不独立指的是同一个子问题可以同来求解多个问题）
  - **动态规划多数是自底而上**来求解问题的,先解决小问题，再由小问题解决大问题，利用额外的空间存储前面子问题的计算；**分治法自顶向下**，**一般递归求解。**
  - 动态规划一般用于解决**最优问题**，而分治法用于解决**通用问题**

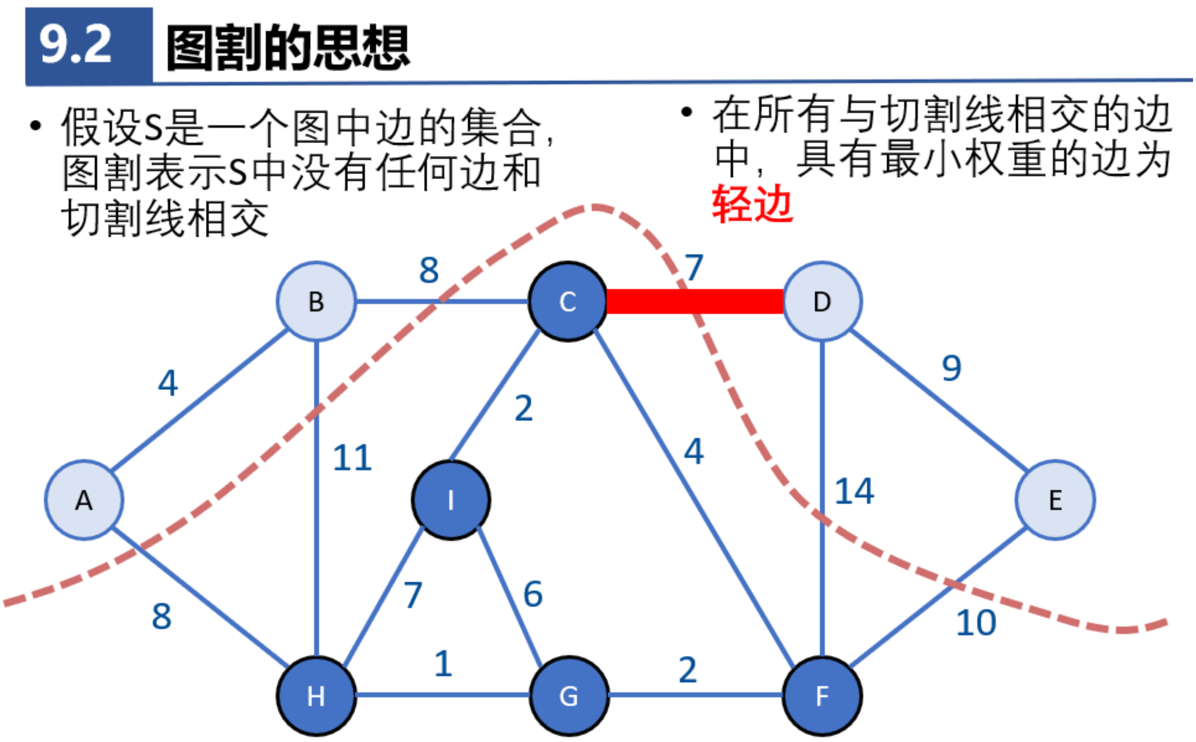
	标准分治	动态规划	贪心算法
适用类型	通用问题	优化问题	优化问题
子问题结构	每个子问题不同	很多子问题重复（不独立）	只有一个子问题
最优子结构	不需要	必须满足	必须满足
子问题数	全部子问题都要解决	全部子问题都要解决	只要解决一个子问题
子问题在最优解里	全部	部分	部分
选择与求解次序	先选择后解决子问题	先解决子问题后选择	先选择后解决子问题

第五章：图算法

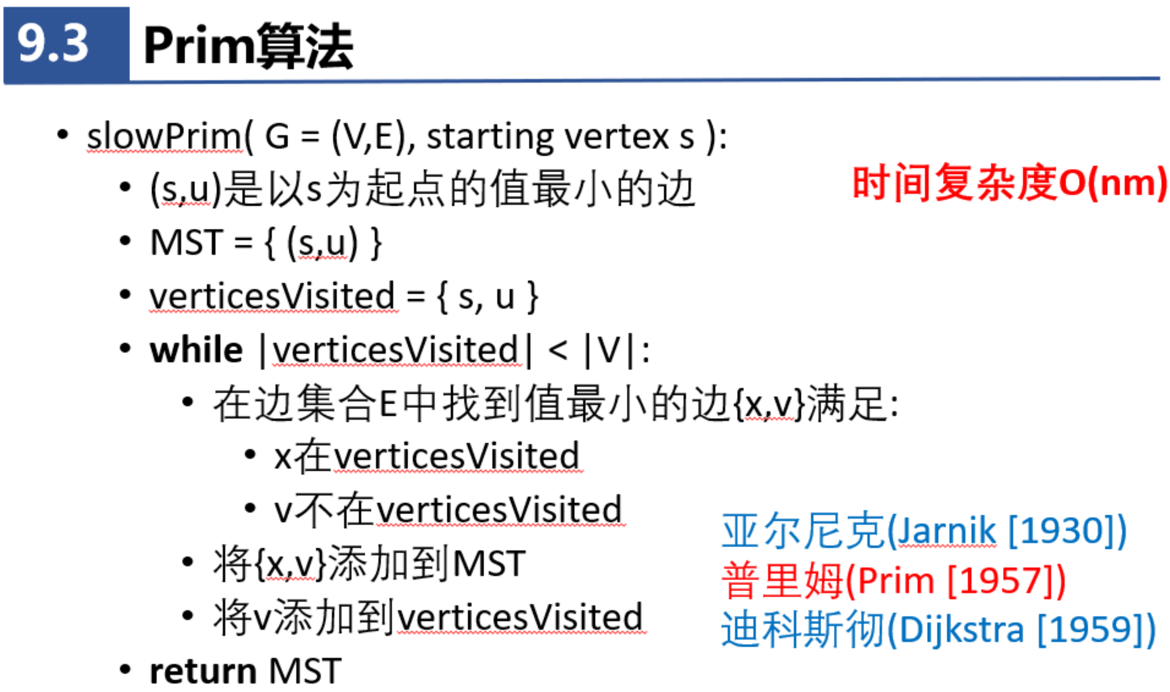
最小生成树（贪心算法）

[【算法】最小生成树——Prim和Kruskal算法_简述最小生成树的两种算法思想(prim算法、kruskal算法),给定一个带权图,根据算法-CSDN博客]

- 图割



Prim算法



> 如果使用类似堆结构，复杂度为**O(m + nlog(n))**

Kruskal算法

9.4 Kruskal算法

- kruskal(G = (V,E)):
  - 以非递增的方式对E中边的权重进行排序
  - MST = {}   // 初始化一棵空树
  - **for** v in V:
    - **makeSet**(v)   // 将每个顶点放在自己所在的树中
  - **for** (u,v) in E: // 按排序的顺序访问边
    - **if find**(u) != **find**(v):   // 如果u和v不在相同的树中
      - add (u,v) to MST
      - **union**(u,v)   // 合并u所在的树和v所在的树
  - **return** MST

- 对边进行排序的时间复杂度 O(m log(n))
  - 在实际中，如果权重是小的整数，可以使用基数排序，时间复杂度为O(m)
- 剩下的部分:
  - 调用n次**makeSet**：将每个顶点放在自己的集合中
  - 调用2m次**find**：对于每条边, **find**终点
  - 调用n-1次**union**：树上最多有n-1条边
- 总的时间复杂度:
  - 最差情况下 O(mlog(n))
  - 如果调用基数排序，接近O(m)

算法证明

- 假设当前的选择与最小生成树是一致的
- 下一个贪心选择依然与最小生成树是一致的

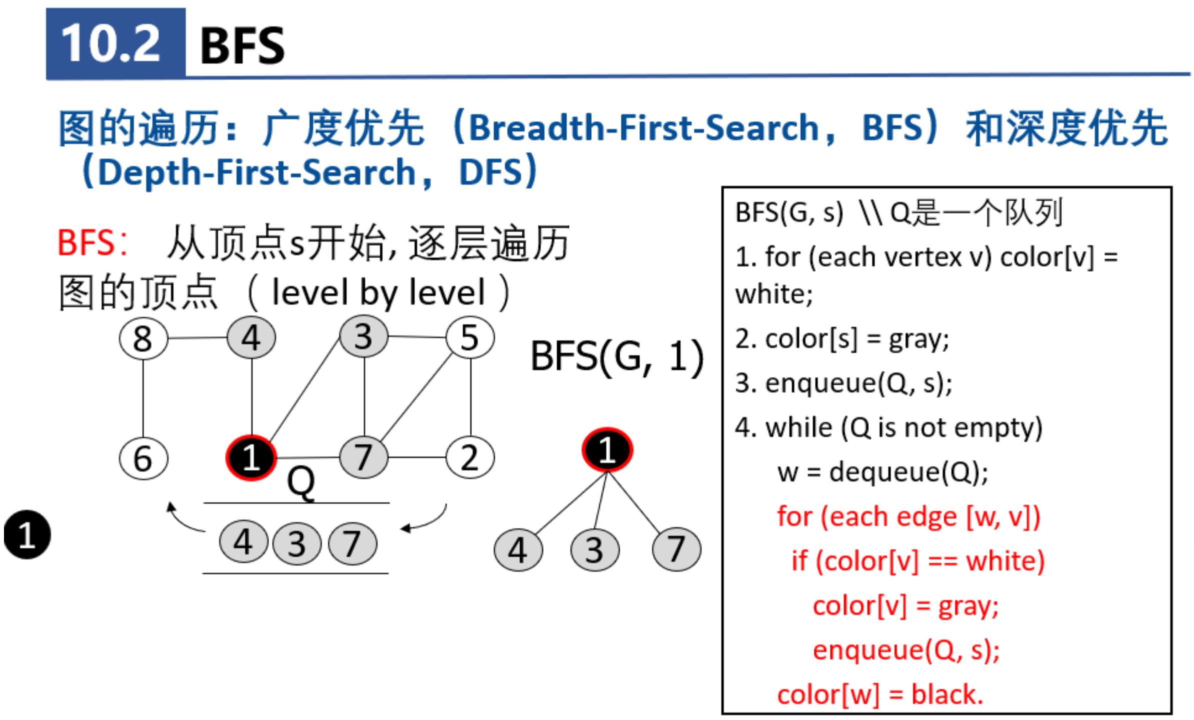
总 结

- Prim:
  - 逐渐扩展一棵树.
  - 基于堆的时间复杂度为O(m + nlog(n))
- Kruskal:
  - 逐渐扩展一个森林
  - 使用并查集数据结构的时间复杂度为O(mlog(n))
  - 如果使用基数排序对边的权重进行排序，则为O(m)

对于密集连接的图，如果无法使用基数排序，则选择Prim算法更好	对于稀疏连接的图，如果可以使用基数排序对边的权重进行排序，则Kruskal算法更好

图搜索算法

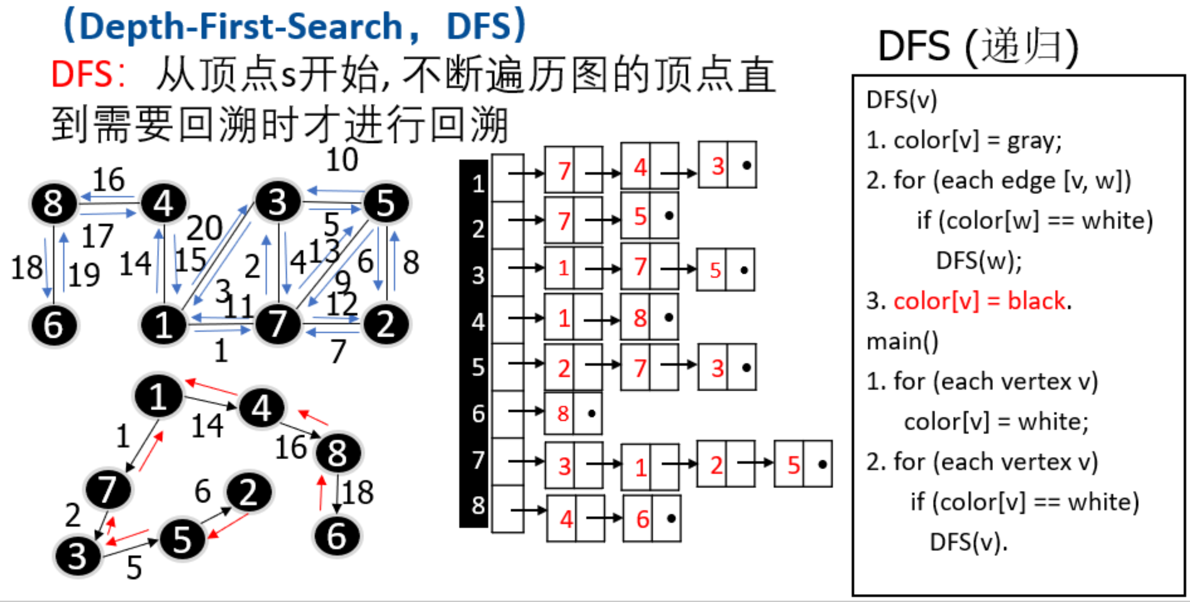
BFS



T(n)=O(m+n)

- BFS得到了一棵**BFS树**
- 一条边的两个终点不在BFS的同一个分支上可能存在两种情况：它们在相同的层或者是相邻层
- BFS 可以用来查找两个节点之间的**最短路径**
- BFS可以用来测试图的连接情况
- BFS可以用来测试**二部图**

DFS



T(n)=O(m+n)

- DFS可以用来实现**拓扑排序**
- DFS可以用来找**连通分量**（强连通分量）

[强连通分量（超详细！！！） - endl\n - 博客园 (cnblogs.com)]

1.DFS时间复杂度

- **DFS算法是一一个递归算法**，需要借助一个递归工作栈，故它的空间复杂度为O(N)。遍历图的过程实质上是对每个顶点查找其邻接点的过程，其耗费的时间取决于所采用结构。
- **邻接表**表示时，查找所有顶点的邻接点所需时间为O(E)，访问顶点的邻接点所花时间为O(N),此时，总的时间复杂度为O(N+E)。
- **邻接矩阵**表示时，查找每个顶点的邻接点所需时间为O(N)，要查找整个矩阵，故总的时间度为O(N^2)。

2.BFS时间复杂度

- **BFS是一种借助队列来存储的过程**，分层查找，优先考虑距离出发点近的点。无论是在邻接表还是邻接矩阵中存储，都需要借助一个辅助队列，N个顶点均需入队，最坏的情况下，空间复杂度为O(N)。
- **邻接表**形式存储时，每个顶点均需搜索一次，时间复杂度T1=O(N)，从一个顶点开始搜索时，开始搜索，访问未被访问过的节点。最坏的情况下，每个顶点至少访问一次，每条边至少访问1次，这是因为在搜索的过程中，若某结点向下搜索时，其子结点都访问过了，这时候就会回退，故时间复杂度为O(E)，算法总的时间复 度为O(N+E)。
- **邻接矩阵**存储方式时，查找每个顶点的邻接点所需时间为O(N)，即该节点所在的该行该列。又有n个顶点，故算总的时间复杂O(N^2)。

单源节点最短路径问题

Dijkstra算法

[图详解第四篇：单源最短路径--Dijkstra算法-腾讯云开发者社区-腾讯云 (tencent.com)]

T(n)=min(O(n^2)，O(mlogn))

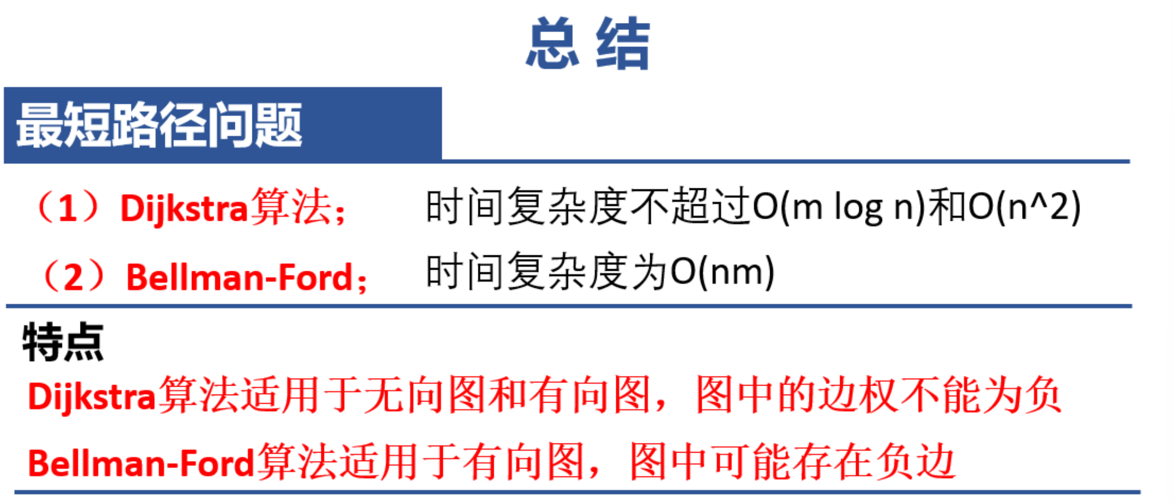
如果图中的边存在负值，Dijkstra算法可能不正确

Bellman-Ford算法

[图详解第五篇：单源最短路径--Bellman-Ford算法-腾讯云开发者社区-腾讯云 (tencent.com)]

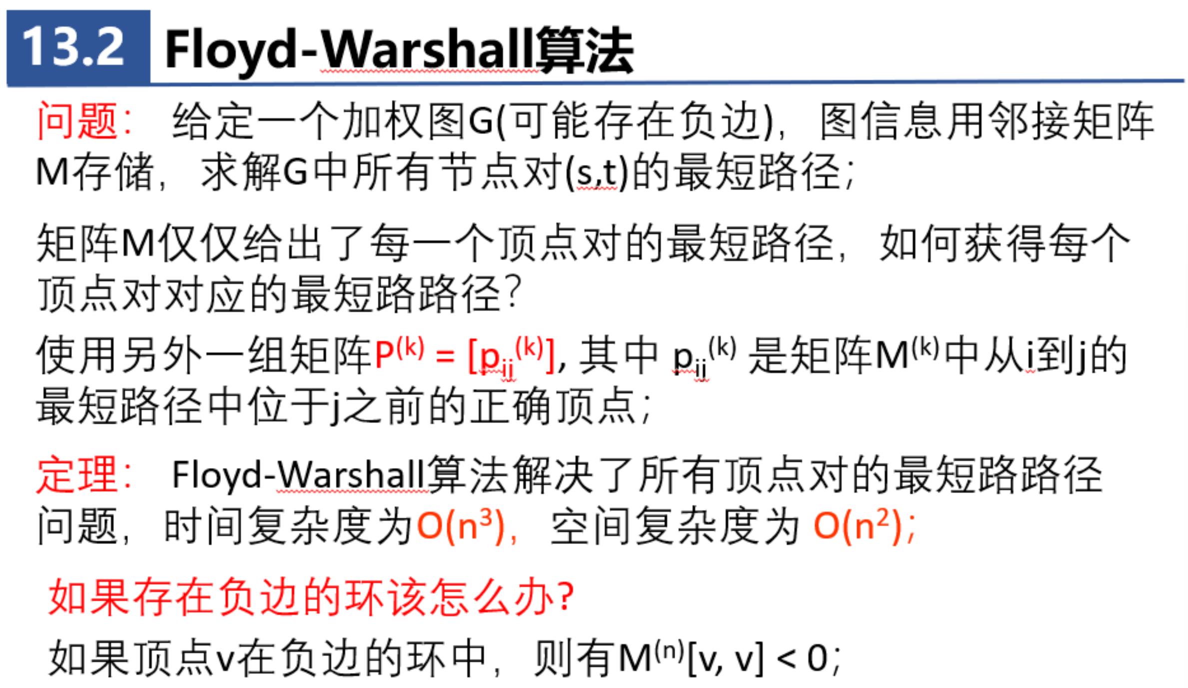
T(n)=O(nm)

算法对比



多源最短路径（Floyd-Warshall算法）

[图详解第六篇：多源最短路径--Floyd-Warshall算法（完结篇）-腾讯云开发者社区-腾讯云 (tencent.com)]

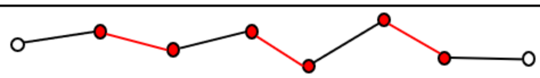


图匹配问题

14.1 图匹配问题和定义

定义： 如果在无向图G中的边集合M中，没有两条边共享一个顶点，则边集合M是一个**图匹配**。

定义： 增广路径 (w.r.t 一个匹配M)是一个始点与终点都为未匹配点的交错路径。



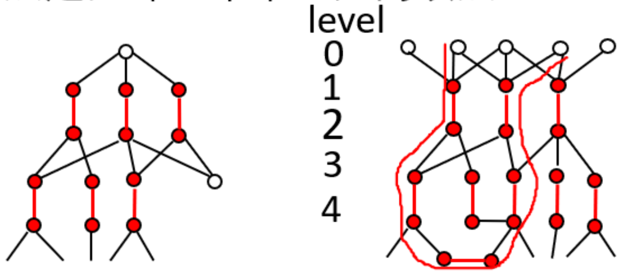
定理： 如果在G中，M没有增广路径，则匹配M是最大的。

证明： 如果M是最大的，则满足没有增广路径，否则，可以构建一个更大的匹配；

14.2 增广路径求解

定理： 如果在G中，M没有增广路径，则匹配M是最大的。

求解增广路径: 从一个未匹配的点开始，找到一条增广路径，其终点是另外一个未匹配的顶点。



与BFS很像，但是: 在偶数层，扩展所有可能的方式，在奇数层，只在单个匹配的边上扩展

T(n)=O(n+m)

> 此算法仅仅适用于**二部图**

第六章：回溯和分支限界

回溯

N皇后

> 将n个皇后放到一n x n的棋盘,满足没有两个皇后会互相攻击

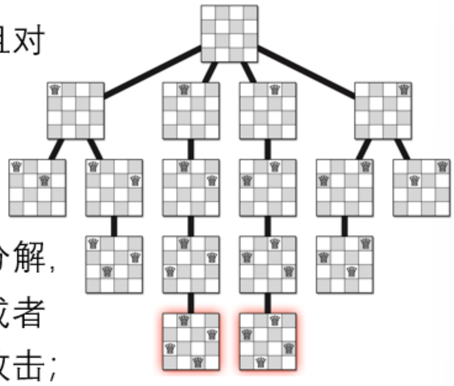
回溯算法的一般思路：

- 从第一行开始，每次放一个皇后到棋盘上；
- 在放置第r个皇后时，需要从左到右系统地循环尝试第r行中所有n个位置；
  - 如果某个位置被已经放置的皇后攻击，则跳过该位置；
  - 否则, 暂时将一个皇后放置到该位置，然后递归地摸索后续的行中可行的皇后放置位置。

15.2 N皇后

N皇后的回溯算法

- 算法PQ的执行过程可以通过一棵递归树来描述;
- 树上的每个节点对应一个递归子问题，并且对应一个合法的部分解;
- 根节点对应一个空棋盘;
- 递归树上的边对应递归调用;
- 如果某个叶子节点对应一个不能扩展的部分解，则要么每一行已经有一个皇后（完整解）或者在下一个空行钟每个位置都能被已有皇后攻击;
- 利用回溯搜索完整解的过程等价于在该树上进行深度优先搜索.



子集和

15.4 子集和

问题定义: 给定一个正整数集合X和一个目标整数T，X中是否存在一个子集元素加起来为T。

例子： 如果X={8,6,7,5,3,10,9}， T=15; 则{8,7}，{7,5,3}，{6,9}和{5,10} 的和都是15， 如果X={11,6,5,1,7,13,12}， T=15， 则没有满足条件的集合。

- 对于一般情况, 考虑任意元素x ∈ X;
- 只有满足下列条件时，才存在X的子集，其和为T:
  - X的子集包含x，并且和为T;
  - X的子集不包含x，而和为T;
- 对于第一种情况, 存在一个子集X - {x} ，其和为T - x;
- 对于第二种情况, 存在一个子集X - {x} ，其和为T;
- 因此，SubsetSum(X,T) 可以简化为如下两种情形:
  - SubsetSum(X - {x}, T - x) 和 SubsetSum(X - {x}, T)

最长递增子序列

> 给定整数序列a[1...N]，寻找元素递增的最长子序列。

- 法一

15.5 最长递增子序列

问题定义:给定整数序列a[1…N]，寻找元素递增的最长子序列。

给定一个整数prev和一个向量A[1 … n],求解A中最长递增子序列，满足子序列的每个元素都大于prev。

- 定义LISbigger(i，j）表示A[j…n]的最长递增子序列的长度，满足每个元素都大于A[i].
- 递归策略给出了如下递归方程:

$$LISbigger(i,j)=\begin{cases}0 & \text{if } j>n\\ LISbigger(i,j+1) & \text{if } A[i]\geq A[j]\\ \max\left\{\begin{array}{c}LISbigger(i,j+1)\\ 1+LISbigger(j,j+1)\end{array}\right\} & \text{otherwise}\end{cases}$$

- 法二

> 定义LISfirst(i) 表示以A[i]开始的集合A[i...n]的最长递增子序列的长度

方法2：除了每次考虑输入序列的一个元素之外，可以尝试每次求解输出序列的一个元素：

给定一个索引i，求解从A[i]开始的A[i…n]的最长递增子序列。

$$LISfirst(i)=1+\max\left\{LISfirst(j)\mid j>i \text{ and } A[j]>A[i]\right\}$$

```
LISFIRST(i):
  best ← 0
  for j ← i+1 to n
      if A[j] > A[i]
          best ← max{best, LISFIRST(j)}
  return 1 + best
```

```
LIS(A[1..n]):
  best ← 0
  for i ← 1 to n
      best ← max{best, LISFIRST(i)}
  return best
```

分支-限界

> 分支限界法按**广度优先策略**搜索问题的解空间树，在搜索过程中，对待处理的节点根据限界函数估算目标函数的可能取值，从中选取使目标函数取得极值（极大或极小）的节点优先进行广度优先搜索，从而不断调整搜索方向，尽快找到问题的解。
>
> 分支限界法适用于求解**最优化问题**的**一个最优解**。

设计思想

（1）确定一个**合理的界限函数**，并根据界限函数确定目标函数的**界[down,up]**

（2）按照**广度优先策略**搜索问题的解空间树，在分支节点上依次扩展该节点的所有孩子节点，分别估算这些孩子节点的目标函数的可能取值，某孩子节点的目标函数的可能取值**超出目标函数的界**，则将其**丢弃**；否则，将其加入待处理节点表（**活节点表**）中。

（3）依次从**活结点表**中选取使目标函数**取得极值的终点**成为**当前扩展节点**，重复上述过程，直到找到最优解。

- 如何确定合适的界限函数？
    - **计算简单，减少搜索空间，不丢最优解**

多阶段图最短路径

16.2 多阶段图最短路径问题

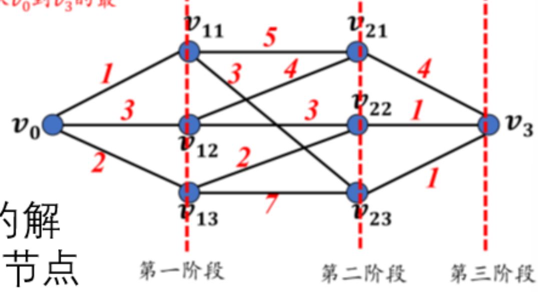
（1）确定合理的界限函数：

UP=1+3+1

Down=?

输入：多阶段图

输出：从v_0到v_3的最短路径



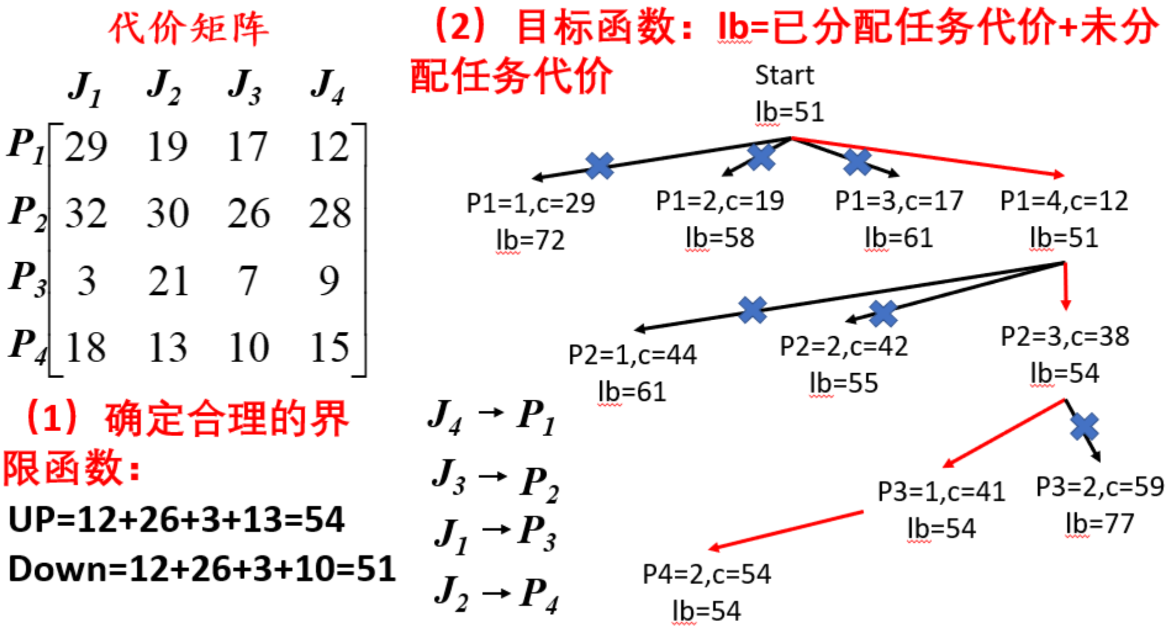
（2）按照**广度优先策略**搜索问题的解空间树，在分支节点上依次扩展该节点的所有孩子节点，分别估算这些孩子节点的目标函数的可能取值，某孩子节点的目标函数的可能取值**超出目标函数的界**，则将其**丢弃**；否则，将其加入待处理节点表（**活节点表**）中。

目标函数：当前路径和

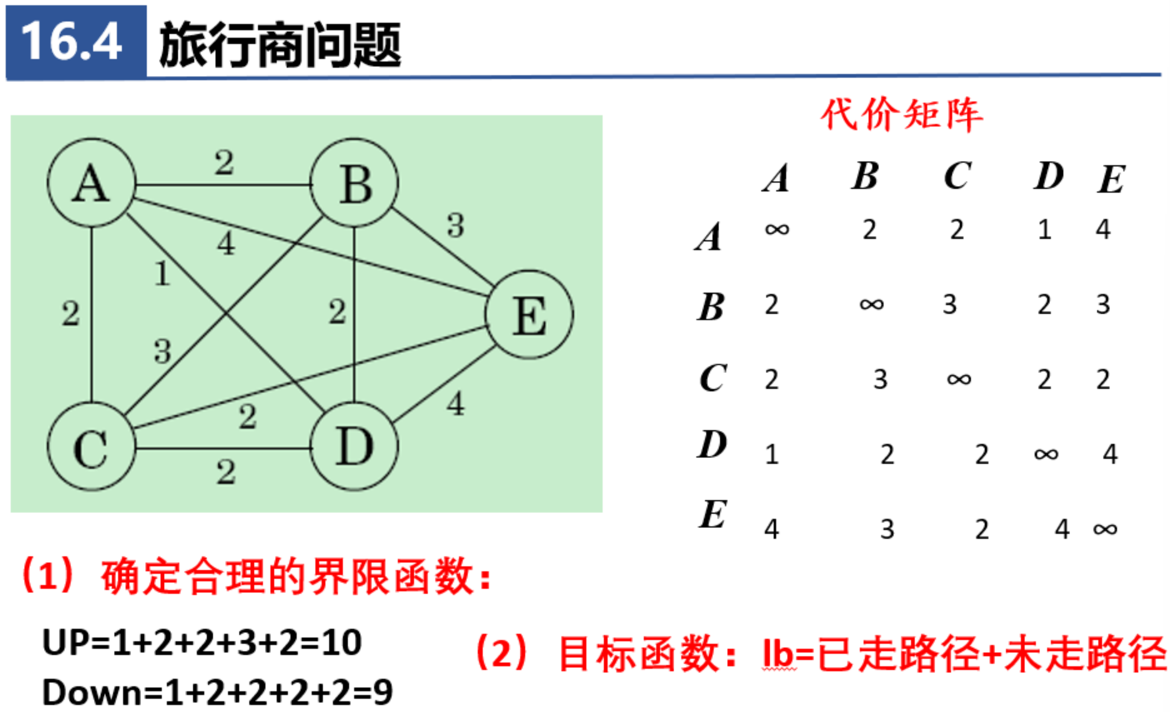
人员安排问题

16.3 人员安排问题

- 输入
    - 人的集合$P=\{P_1,P_2,\cdots,P_n\}$，$P_1<P_2<\cdots<P_n$，工作的集合$J=\{J_1,J_2,\cdots,J_n\}$
    - 矩阵$[C_{ij}]$，C_{ij}是工作J_j分配到P_i的代价
- 输出
    - 矩阵$[X_{ij}]$，$X_{ij}=1$表示P_i被分配J_j，$\sum_{ij}C_{ij}X_{ij}$最小
    - 每个人被分配一种工作，不同人分配不同工作



旅行商问题



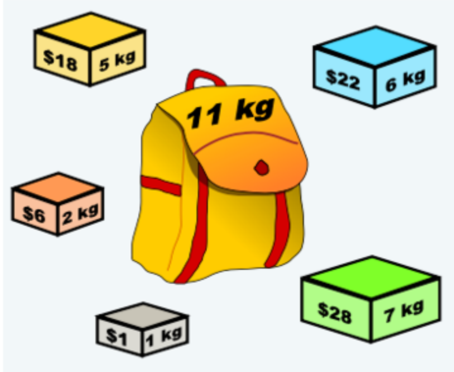
0/1背包问题

[0-1背包问题-分支限界法(优先队列分支限界法)_01背包优先队列式分界法-CSDN博客]

16.5 0/1背包问题

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

weights and values can be arbitrary positive integers



（1）确定合理的界限函数：

Down=28+6*2=40

Up=28/7×11=44

（2）目标函数：lb=已装物品价值+未装物品最优价值

lb=v+(11-w) ×vi/wi

分支界限法和回溯法的区别

1、求解目标不同

- 回溯法的求解目标一般是找出解空间树中**满足条件的所有解**。
- 分支限界法则是尽快找出**满足约束条件的一个解**，或是在满足约束条件的解中找出在某种意义下的最优解。

2、搜索方式不同

- 回溯法——>**深度优先** 遍历结点搜索解空间树。
- 分支限界法——>**广度优先**或最小耗费优先搜索解空间树。

3、存储空间不同

分支限界法由于加入了**活结点表**，所以**存储空间比回溯法大得多**。因此当内存容量有限时，回溯法的成功率要大一些。

4、扩展结点的方式不同

分支限界法中，每个活结点只有一次机会变成扩展结点，一旦成为扩展结点便一次性生成其所有子结点。

区别小结：回溯法空间效率更高，分支限界法由于只需要求到一个解，所以往往更"快"。

第七章：NP问题

[NP问题总结（概念+例子+证明）-CSDN博客]

NP完全性理论

如果问题Q可以在多项式时间内解决，即对于常数c，解决的时间复杂度为O(n^c)，则认为问题Q是可解的。

定义**P** 为在多项式时间内可解的所有问题类别。

- 判定性问题：需要给出**是或者否**的答案
- NP问题

- 定义：如果Q是一个判定性问题，并且对于Q的任何一个实例x，存在算法A_Q满足：
    1. 如果x是一个"yes"实例，则存在y满足AQ(x, y) = 1;
    2. 如果x是一个"no"实例，则对于所有的y，满足AQ(x, y) = 0;
    3. AQ(x, y) 在|x|范围内运行的时间复杂度是多项式的。

> 顶点覆盖，独立集合， 可满足性问题， 和集合划分问题都在NP问题的范畴内
>
> P中的判定性问题Q也是NP问题，即，**P ⊆ NP**。

独立集合问题（IS）：给定图G和整数k，在G中是否可以找到有k个顶点的集合C，满足C中任意两个顶点都不相邻?

最大独立集合（MaxIS）：给定图G，建立最大的独立集合。

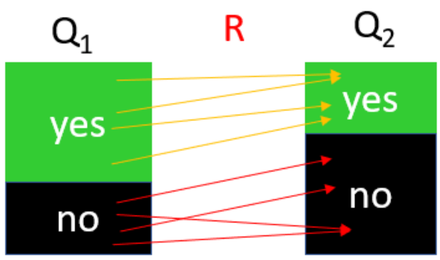
如果MaxIS 可以在$O(n^c)$的时间复杂度内求解，则独立集也是如此。

如果IS可以在$O(n^c)$的时间复杂度内求解，则算法max-size(G)可以在时间复杂度$O(n^{c+1})$内求解IS的最大值。

NP-Hard问题的判别

- 比较问题的难度

定义：Q_1和Q_2为判定性问题。如果存在一个多项式时间算法R，对于Q_1的任意实例x_1，当且仅当$R(x_1)$对于Q2是一个"yes"的实例时，x_1对于Q_1也是一个"yes"的实例。则问题Q_1可以在多项式时间内归约到问题Q_2，记作$Q_1 \le_m^p Q_2$。



引理：假设$Q_1 \le_m^p Q_2$，且Q_2在$\mathcal{P}$中，则Q_1也在$\mathcal{P}$。

> 如果**Q2 是容易问题，则Q1也是容易问题**，否则，**如果Q1 是难问题，则 Q2 也是难问题。**
>
> 即**Q1 的难度不会超过Q2**

- 库克定理

库克定理：对于$\mathcal{NP}$中的每一个问题Q，Q $\leq_m^p$ SAT。

可满足性问题（Satisfiability，SAT）：给定一个合取范式的可满足性问题（CNF），用公式F = F($x_1, x_2, \ldots, x_n$)表示，是否存在一种赋值方案$x_1, x_2, \ldots, x_n$，使得F = true?

> **SAT是NP中最难的问题**，即SAT是NP-完全问题

定义：如果对于$\mathcal{NP}$中的每一个问题Q'，满足Q' $\leq_m^p$ Q，则问题Q是$\mathcal{NP}$-难问题；如果该问题在$\mathcal{NP}$中且是$\mathcal{NP}$–难的，则问题Q是$\mathcal{NP}$–完全问题。

引理：如果$Q_1 \leq_m^p Q_2$，且Q_1是$\mathcal{NP}$–难问题，则Q_2也是$\mathcal{NP}$–难问题

证明：对于$\mathcal{NP}$中的任意问题Q'，由于Q_1是$\mathcal{NP}$–难问题，且$Q' \leq_m^p Q_1$，因此$Q' \leq_m^p Q_1 \leq_m^p Q_2$，可得$Q' \leq_m^p Q_2$，因此$Q_2$是$\mathcal{NP}$–难问题。

$$Q' \xrightarrow{R'} Q_1 \xrightarrow{\quad R \quad} Q_2$$

$$x \longrightarrow R'(x) \longrightarrow R(R'(x))$$

证明问题Q_2的$\mathcal{NP}$–难度的一般过程:

1. 从一个已知的$\mathcal{NP}$–难问题Q_1开始(例如，SAT);
2. 说明$Q_1 \leq_m^p Q_2$，从而给出Q_2的$\mathcal{NP}$–难度。

> 独立集合问题（IS）、顶点覆盖问题（VC）、子集和问题（subset-sum）、集合划分问题（Partitition）是NP-完全问题

总 结

NP完全性理论I

（1）$\mathcal{P}$：多项式时间内可解的所有问题类别；

（2）$\mathcal{NP}$：非确定性多项式时间；

NP–完全问题

SAT，IS，VC，Subset-Sum和Partition

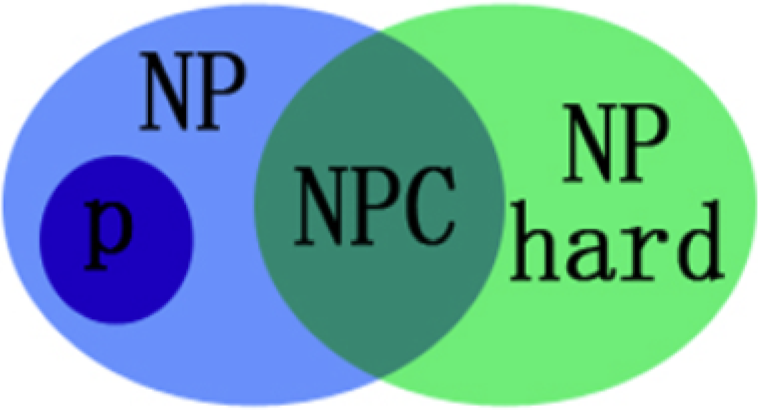


图1 P NP NPC NPhard关系的图形表示