一 算法

1. 算法的五个特性：**输入**（0-n）、**输出**（1-n）、**确定性**（无歧义）、

   可行性（可表达、可计算、简单）、**有穷性**（有限时间）
2. 算法的概念：在有限时间内，对问题求解的一个清晰的指令序列

二 算法分析

1. 渐进时间复杂度（三个符号）

   （1）渐进上界 O

   （2）渐进下界Ω

   （3）紧渐进符号θ

渐近分析的记号 $\theta\ O\ \Omega\ o\ \omega$

先看结论：a 代表$f(n)$，b 代表$g(n)$

$$f(n)=O(g(n)) \approx a \le b;$$

$$f(n)=\Omega(g(n)) \approx a \ge b;$$

$$f(n)=\Theta(g(n)) \approx a = b;$$

$$f(n)=o(g(n)) \approx a < b;$$

$$f(n)=\omega(g(n)) \approx a > b.$$

2. 求算法时间复杂度

   （1）非递归：找基本语句，看循环嵌套

   （2）递归：通项公式、展开、主定理
3. 比较时间复杂度：比值，不行就洛必达

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{order of growth of } g(n) \\ c>0 & \text{order of growth of } T(n) = \text{order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{order of growth of } g(n) \end{cases}$$

2-4. 使用O、Ω和Θ的非正式定义判断以下断言是否为真

a) $n(n+1)/2 \in O(n^3)$

解答:

$n(n+1)/2 = (n^2 + n)/2 \approx n^2/2$
当$n \to \infty$时，$n^2/2 \le n^3$
因此该断言为真

b) $n(n+1)/2 \in \Theta(n^2)$

解答:

$n(n+1)/2 = (n^2 + n)/2$
存在$c_1=1/4$, $c_2=1/2$, $n_0=1$使得
$c_1 n^2 \le (n^2 + n)/2 \le c_2 n^2$对于所有$n \ge n_0$
因此该断言为真

三 蛮力法

案例：选择排序、冒泡排序、旅行商穷举、背包穷举、分配工作穷举

四 递归算法

1. 递归的两个条件：**边界条件 递归方程**
2. 递归的类型：**减一**（斐波那契）、**减常数因子**（二分）、**减不固定因子**（汉诺塔）
3. 递归的实例：斐波那契数列、阿克曼函数、汉诺塔、排列问题
4. 递归通常会消耗更多的内存，因为每次函数调用都需要在**调用栈**上保存信息
5. 主定理（哪个大是哪个）

定理： 设$a \ge 1$, $b>1$为常数, $f(n)$为函数, $T(n)$为非负整数，且$T(n)=aT(n/b)+f(n)$，则

简单记法：用$f(n)$和$n^{\log_b a}$比较

- 若$n^{\log_b a}$大，则$T(n) = \theta(n^{log_b a})$
- 若$f(n)$大，则$T(n) = \theta(f(n))$
- 若二者同阶，则$T(n) = \theta(n^{log_b a} \log n) = \theta(f(n) \log n)$

注意 F（n）是多项式增长

五 分治法

1. 分治法三个步骤:

   将问题划分为两个或多个更小的**子问题**

   递归地解决这些子问题

   将子问题的解**合并成为原问题**的解

2. 分治法要求：所有问题都是**相同类型**、**相互独立**、**规模均衡**

3. **大整数乘法：**乘数都分半形成 4 个子问题，再通过表达式变换分成 3 个

$$X * Y = A * C \cdot 10^{n} + (A * D + B * C) \cdot 10^{n/2} + B * D$$

Since we have: $(A + B) * (C + D) = AC + (AD + BC) + BD$

$$(AD + BC) = (A + B) * (C + D) - A^{*} C - B^{*} D$$

$$X * Y = A * C \cdot 10^{n} + [(A + B) * (C + D) - A^{*} C - B^{*} D] \cdot 10^{n/2} + B * D$$

$$T(n) = \begin{cases} O(1) & n = 1 \\ 3T(n/2) + O(n) & n > 1 \end{cases}$$

$$T(n) = O(3^{\log_2 n}) = O(n^{\log_2 3}) \approx O(n^{1.585})$$

4. **矩阵乘法的 Strassen 算法：**传统分成 8 个子矩阵相乘，通过变换使用了 7 次乘法

   设 $T(n)$ 为乘两个 $n \times n$ 矩阵所需的时间，则Strassen算法的递归关系式可以表示为：
   $T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$

   因此，Strassen算法的渐进时间复杂度为 $O(n^{2.807})$，这比传统矩阵乘法的 $O(n^3)$ 更优。

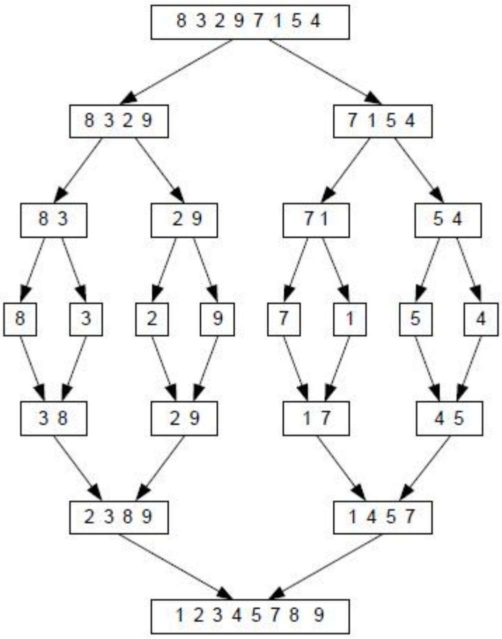
5. **二分搜索：**low high mid

```
ALGORITHM BinarySearch(A[0..n-1], K)
l ← 0;  r ← n-1
while l ≤ r do                     // l and r crosses over→ can't find K
    m ← ⌊(l+r)/2⌋
    if K = A[m] return m           //the key is found
    else if K < A[m] r ← m-1       //the key is on the left half of the array
        else l ← m+1               // the key is on the right half of the array
return -1
```

   最好情况：第一次比较后直接找到目标，时间复杂度为 O（1）

   最坏情况和平均情况 $O(\log n)$

6. **归并排序（Merge Sort）：**不断地将一个数组分成两半来进行递归排序，然后**合并**已排序的部分



$$C(n) = 2C(n/2) + C_{merge}(n) \text{ for } n>1$$

$$C(1) = 0$$

$$\text{Where } C_{merge}(n) = n-1$$

$$C(n) = \Theta\ (n \log n)$$

（1）性能非常稳定，无论输入数据的初始顺序如何，归并排序都将**执行相同数量的比较和移动操作**，时间复杂度都是 O（nlogn）

（2）它的键值比较次数非常接近理论的最小值，缺点是需要大量的额外存储空间。

7. **快速排序（Quick Sort）：**通过选择基准值（pivot）将数组分为两部分，小于基准的元素放在左边，大于的放在右边，然后递归地对两部分分别排序，平均 O (n log n)，最坏 O (n^2)

    （1）**指针移动与交换：**

    - 左指针向右移动，直到找到大于基准的值。
    - 右指针向左移动，直到找到小于基准的值。
    - 交换左右指针所指的元素，继续移动指针。

    （2）**最终交换：**当左右指针相遇或交叉时，将基准值与右指针位置的元素交换。

（3）**递归**地对基准元素（Pivot）左右两侧的**子数组**继续执行相同的操作

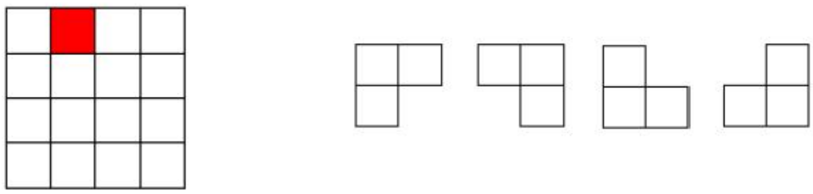
平均情况：快速排序的平均时间复杂度是 $O(n\log n)$。

最坏情况：如果每次分区操作都将列表分成两部分不平均，如当数组已经是有序的或完全逆序时，最坏的时间复杂度是 $O(n^2)$。

最好情况：最好情况发生在每次分区都能将数组均等地分成两部分，此时时间复杂度为 $O(n\log n)$。

8. **棋盘覆盖**:

在一个 $2^k\times2^k$ 个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。



$$T(k)=\begin{cases} O(1) & k=0 \\ 4T(k-1)+O(1) & k>0 \end{cases}$$

$T(n)=O(4^k)$ 渐进意义下的最优算法 $\text{所需骨牌数}=\dfrac{4^k-1}{3}$

六 减治法

1. 减治法：减治法是一种解决问题的策略，其核心思想是通过不断迭代逐步减小问题的规模，在每一次迭代中，问题的结构一般保持不变。

减一个常量，常常是减 1：如插入排序

减一个常因子，常常是减去因子 2（一半）：如折半查找

减可变规模：欧几里得算法（碾转相除法）

2. 插入排序（扑克牌排序，减常量）：

（1）将数组分为已排序部分和未排序部分，取出未排序部分的第一个元素，将其插入到已排序部分中的适当位置

（2）输入数组已经是排序好的，最好 O (n)；最坏、平均 O (n²)

时间复杂度的计算主要考虑每个元素在插入过程中需要比较的次数。在最坏情况下，第 i 个元素需要 $i-1$ 次比较，因此总的比较次数是 $\frac{n(n-1)}{2}$，这是一个二次函数，因此渐进时间复杂度为 $O(n^2)$。

3. **拓扑排序（减常量）：**

（1）有向无环图，依此删除入度为 0 的节点，删除顺序就是拓扑排序

（2）实现拓扑排序的方法（时间复杂度均为 O (V+E) 顶点数边数）:

使用队列（减 1 技术）：利用队列来存储所有入度为零的顶点。每次从队列中取出一个顶点，更新其相邻顶点的入度，若更新后某个顶点的入度变为零，则将其加入队列。

使用深度优先搜索（DFS）：对每个未访问的顶点执行 DFS，访问结束后将顶点推入栈中。最后从栈中依次弹出顶点即可得到一个拓扑排序。

4. **二分搜索（减常因子）**

5. **假币问题（减常因子）：**从一堆硬币中找出唯一的一个较轻的假币，思路是分成 k 组，时间复杂度就是 O（logkn）

6. **俄罗斯农民乘法（减常因子）：**手算两个数的乘积，它通过重复的加法和减半操作来实现乘法。

$$n \cdot m = \frac{n}{2} \cdot 2m.$$

If n is odd, we need only a slight adjustment of this formula:

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m.$$

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	+(130 + 1040) = 3250

时间复杂度：O(logA)，其中 A 是较小的乘数。每次操作 A 至少减少一半，因此操作的次数取决于 A 的二进制位数。

7. **欧几里得算法（减可变因子）：**辗转相除法，将较大数除以较小数，得到余数，余数置为新的较小数，重复直到余数为 0。

假设需要计算 252 和 105 的最大公约数：

- $252 \div 105 = 2$ 余数 42
- $105 \div 42 = 2$ 余数 21
- $42 \div 21 = 2$ 余数 0
- 余数为 0，最大公约数是 21。

时间复杂度：最坏情况下为 O（log（min（a，b）)），每次操作都至少把问题规模减半

七 变治法

1. 变治法是一种基于变换思想，把问题变换成一种更容易解决的类型
2. 变治法的类型：**实例化简、改变表现、问题化简**
3. **预排序算法(实例化简)：**

应用：

统计众数：先对数组进行排序，然后通过线性扫描来统计每个元素快速找出众数。

快速查找：排序后的数组可以使用二分查找

数据去重：在排序数组中，重复的元素会被排在一起，可以通过线性扫描去除重

Selection Sort	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$
Insertion Sort	$C_{worst}(n) = \frac{(n-1)n}{2}$ $\quad C_{best}(n) = n-1$ $\quad C_{avg}(n) \approx \frac{n^2}{4}$
Mergesort	$\Theta(n \log n)$
Quicksort	$C_w(n) = \Theta(n^2)$ $\quad C_b(n) = \Theta(n \log n)$ $\quad C_{avg}(n) = O(n \log n)$

6. **高斯消元法(实例化简)：**

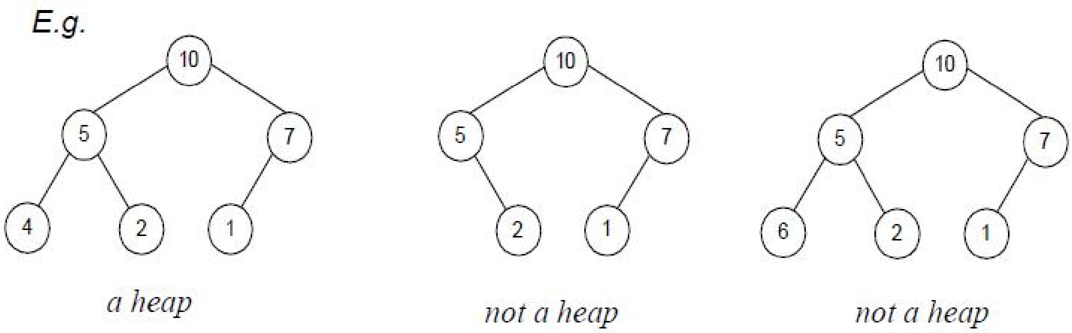
（1）解线性方程组，通过行交换和行变换，逐步**简化（变换）原始的方程组**，最终将其

转换成一个更易于解决的形式（就是求解方程的过程）。

时间复杂度：高斯消元法的时间复杂度大致为 $O(n^3)$，其中 n 是方程组中未知数的数量。这是因为每个消元步骤涉及到对 $n-1$ 行进行 n 次操作，而对于每个未知数都需要进行此操作。

5. **堆、堆排序、堆维护（改变表现）：**

（1）变治法的原因是转换成了堆的树形结构，注意堆是**完全二叉树**



（2）构建堆

- 构建堆的时间复杂度为 $O(n)$。
- 排序过程中，每次重新恢复堆性质的时间复杂度为 $O(\log n)$，共需要进行 $n-1$ 次操作。因此，总的时间复杂度为 $O(n\log n)$。

（3）堆插入：添加到末尾**上浮**

- 最坏情况下，新元素可能需要从堆底浮到堆顶，时间复杂度为 $O(\log n)$。

（4）堆删除（堆顶）：将堆顶与最后一个元素交换，然后移除最后一个元素进行**下沉**

6. **霍纳法则（改变表现）：**

（1）多项式求值，通过多项式变形，减少乘法操作来提高计算多项式的效率

（2）通过嵌套式的连续乘加操作，霍纳法则减少了多项式计算中不必要的重复乘法，使得计算更为高效，有线性时间复杂度，对于高度多项式尤为有效。

考虑一个多项式：
$P(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_2x^2+a_1x+a_0$

霍纳法则将其重写为嵌套的形式：
$P(x)=(((a_nx+a_{n-1})x+a_{n-2})x+\ldots+a_1)x+a_0$

初始化：设定结果变量 $b=a_n$。

假设我们要计算多项式 $P(x)=2x^3-6x^2+2x-1$ 在 $x=3$ 时的值：

- 初始化：$b=2$
- $b=2\times3+(-6)=0$
- $b=0\times3+2=2$
- $b=2\times3-1=5$

因此，$P(3)=5$。

时间复杂度：霍纳法则的时间复杂度为 $O(n)$，其中 n 是多项式的度。这是因为每个系数只参与一次乘法和一次加法。

7. **问题化简**的实例

（1）几何问题转变成代数问题——线性规划（背包）

（2）转化成图问题（过河）

八 回溯法和分支限界法

（一）搜索算法

1. **穷举搜索**：组合优化问题（排列组合）、密码破解、排列组合问题、TSP、子集和问题

步骤

1. **列出所有可能的解**：生成所有可能的候选解。
2. **逐一验证**：对每个候选解进行验证，检查它是否满足问题的条件。
3. **选择最佳解**：从满足条件的解中选择最佳解（如果有）。

2. **DFS**：从一个起点开始，沿着一条路径一直往深处走，直到不能继续为止，然后回溯到最近的分支点继续搜索，直到所有节点都被访问

3. **BFS**：从一个起点开始，首先访问所有与起点直接相连的节点，然后依次访问这些节点的邻接节点，逐层扩展（队列实现）

- **DFS**：适用于需要深入搜索解决的问题，如路径问题、连通分量检测等。采用邻接链表时时间复杂度为 $O(V+E)$，采用邻接矩阵时时间复杂度为$O(V^2)$
- **BFS**：适用于需要逐层搜索的问题，如最短路径问题。采用邻接链表时时间复杂度为 $O(V+E)$，采用邻接矩阵时时间复杂度为$O(V^2)$

（每条边每个结点都被访问）

（二）回溯法

1. 通过递归来解决问题的算法，其基本思想是在解空间树中进行深度优先搜索（DFS），从而寻找问题的所有解。

2. 剪枝包括两种：

（1）**约束函数**：在扩展结点处剪去**不满足约束**的子树

（2）**限界函数**：剪去**得不到最优解**的子树

3. **0-1背包问题**：无限界（直接看解空间树的叶节点）

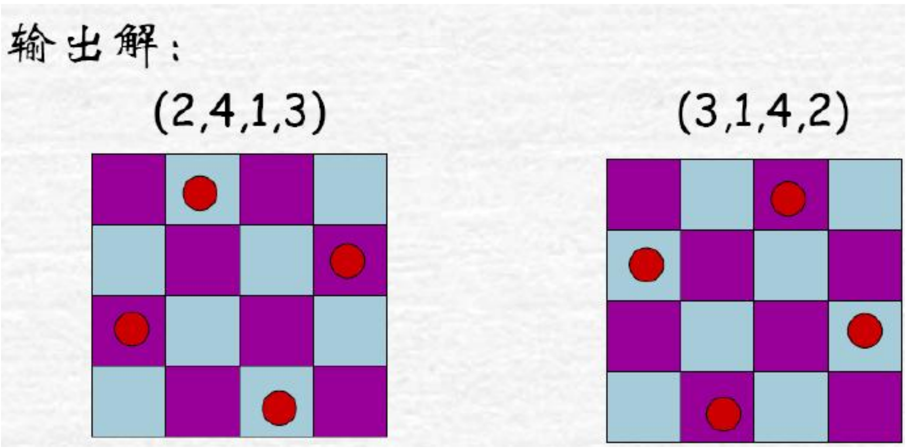
有限界（大题）

4. **TSP问题**：构建一个“**排列树**”，从根节点开始（通常为空或仅包含起始城市），通过逐步扩展到所有未访问的城市来探索所有可能的路径（DFS）。

5. **N皇后问题：**

（1）在一个N×N的棋盘上放置N个皇后，使得它们不能互相攻击。皇后可以攻击同一行、同一列或同一对角线上的其他皇后。

（2）通常使用一个一维数组 board[N] ，其中 board[i] 表示第 i 行的皇后放置在哪一列。



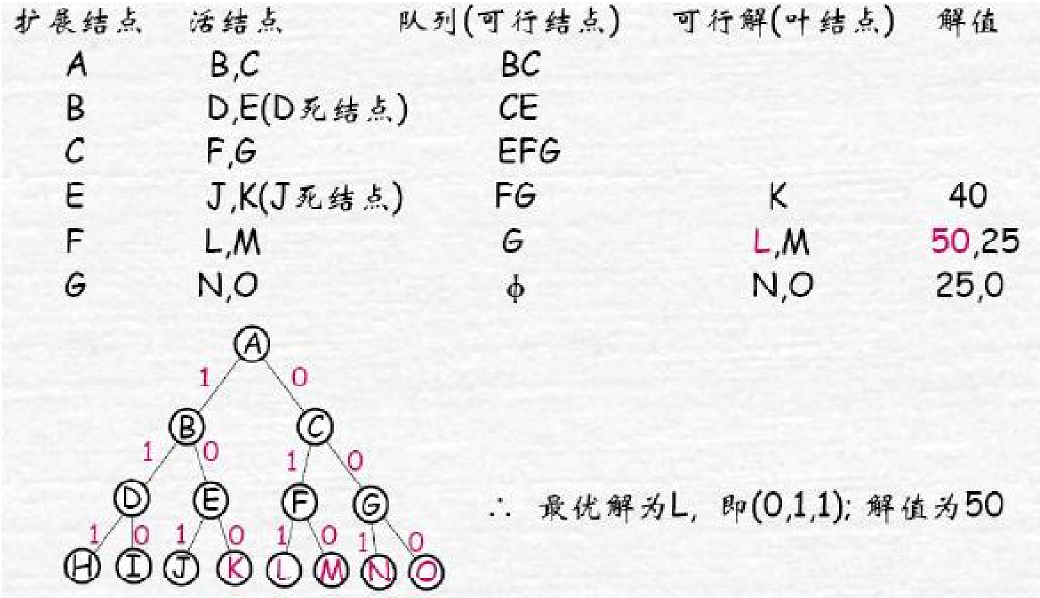
（三）分支限界法

1. 核心思想是系统地**遍历或构建决策树的所有可能的候选解**，通过**“分支”**来生成子候选解，并使用**“限界”**来避免不必要的搜索，即进行剪枝。

2. 分支限界法使用**广度优先**或**最佳优先**搜索策略

3. 常见的两种分支限界法：

(1) **队列式（FIFO）分支限界法**：BFS，逐层扩展解空间树的所有节点，保证了每层的节点都被完全扩展

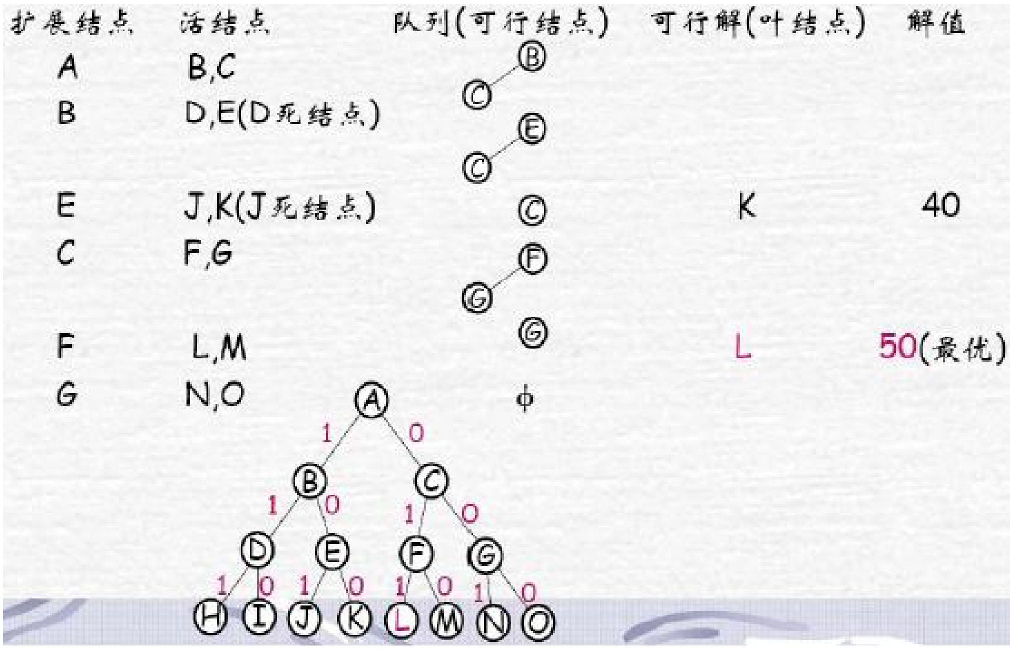


扩展结点	活结点	队列(可行结点)	可行解(叶结点)	解值
A	B,C	BC		
B	D,E(D死结点)	CE		
C	F,G	EFG		
E	J,K(J死结点)	FG	K	40
F	L,M	G	L,M	50,25
G	N,O	ϕ	N,O	25,0

∴ 最优解为L，即(0,1,1)；解值为50

(2) **优先队列分支限界法**：最优优先，根据节点的优先级（如成本、估值等）来选择下一个扩展的节点

优先队列：按照价值率优先



扩展结点	活结点	队列(可行结点)	可行解(叶结点)	解值
A	B,C	B C		
B	D,E(D死结点)	E C		
E	J,K(J死结点)	C	K	40
C	F,G	F G		
F	L,M	G	L	50(最优)
G	N,O	ϕ		

4．**上界计算方法**

（1）贪心法计算上界（背包的价值密度）

（2）线性松弛法

5．**回溯法和分支限界法比较**

（1）**求解目标**不同：一般而言，回溯法求解目标是找出解空间树中满足约束条件的**所有解**，而分支限界法的求解目标则是尽快找出满足约束条件的**一个解**

（2）**搜索方法**不同：回溯法采用**深度优先搜索算法**，分支限界法一般使用**广度优先算法或最佳优先算法**来搜索

（3）对扩展结点的扩展方式不同：分支限界法中每一个活结点只有一次机会成为扩展结点，活结点一旦成为扩展结点就一次性产生其所有儿子结点

（4）存储空间要求不同：**分支限界法的存储空间比回溯法大得多**，因此当内存容量有限时，回溯法成功的可能性更大

6. **装载问题**：有一批共n个集装箱要装上2艘载重量分别为C1和C2的轮船

容易证明：如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。

(1)首先将第一艘轮船尽可能装满；

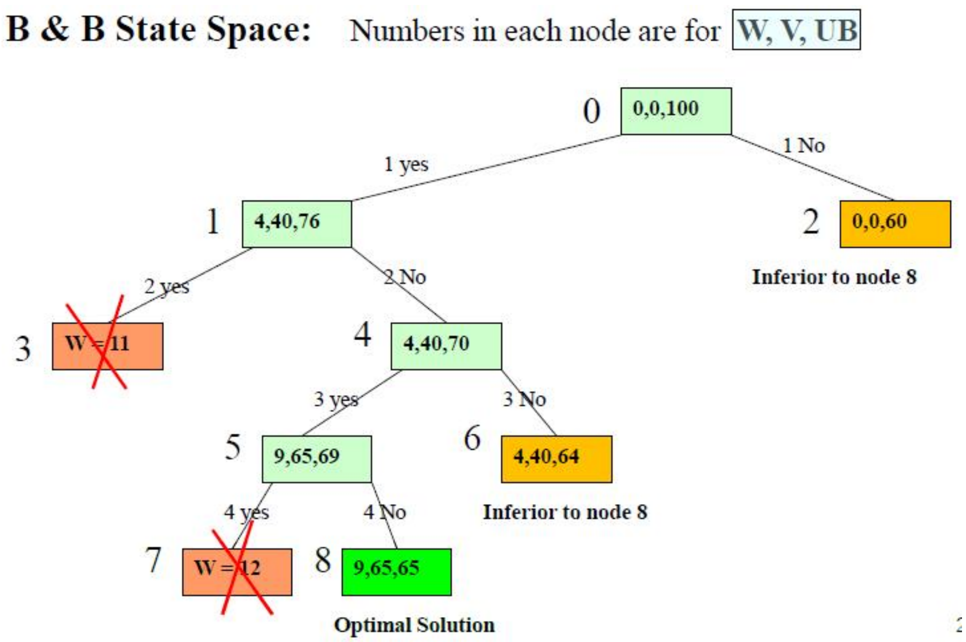
(2)将剩余的集装箱装上第二艘轮船。

后面以装满任意一个集装箱c为例，进行算法阐述.

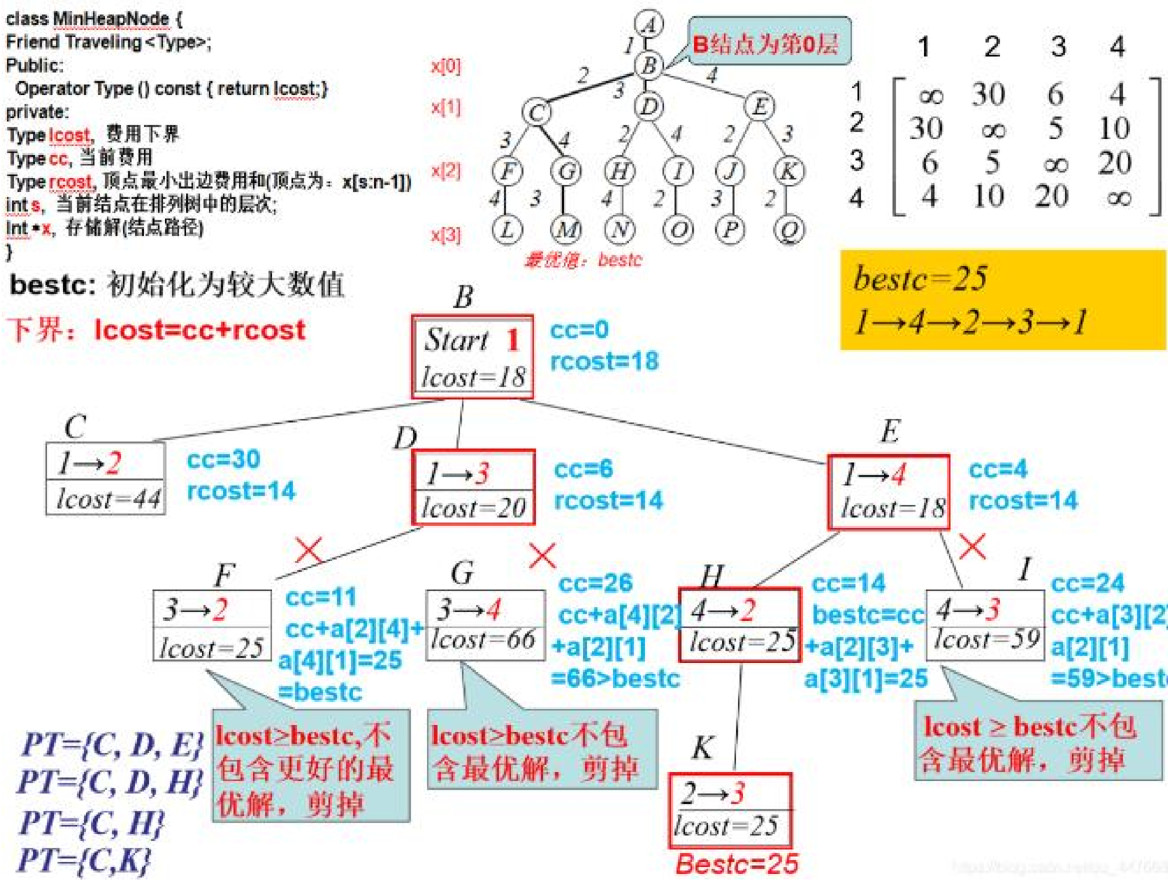
7. 0-1背包分支限界法

$$W = 10, n = 4$$

Item	Weight	Value	Value Density
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4



8. TSP分支限界法



rcost 通常表示剩余未访问节点的最小出边费用和

九 动态规划

1. 动态规划的两个关键属性：**重叠子问题**和**最优子结构**

2. 动态规划三要素：阶段（划分求解阶段）、状态 （无后效性）、策略 （决策序列）

3. **状态**的**无后效性**：如果某阶段状态给定后，则该阶段以后过程的发展不受该阶段以前各阶段状态的影响，也就是说状态具有**马尔科夫性。**

4. 动态规划的思想实质：**分治**和**解决冗余。**

➤**与分治法相同点：**

将原问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。

➤**与分治法不相同点：**

经分解的子问题往往不是互相独立的。若用分治法来解，有些共同部分（子问题或子子问题）被重复计算了很多次。

5. Bellman最优性原理：求解问题的一个最优策略序列时，该最优策略序列的子策略序列总是最优的，则称该问题满足**最优性原理。**

6. 多阶段决策问题（MDP）：求解的问题可以划分为一系列相互联系的 阶段，在每个阶段都需要作出决策，且一个阶段决策的选择会影响下一个阶段的决策 ，从而影响整个过程的活动路线

7. 动态规划两种求解方法：自顶向下（备忘录）和自底向上

8. 动态规划实例：

（1） 计算二项式系数

$C(n, k) = C(n-1, k-1) + C(n-1, k)$

1. **选第 n 个物品**：再从剩下的 $n-1$ 个中选 $k-1$ 个（即 $C(n-1,k-1)$）。
2. **不选第 n 个物品**：直接从剩下的 $n-1$ 个中选 k 个（即 $C(n-1,k)$）。

each row i (0<=*i* <=*n*) *from left to right, starting with C(n, 0)=* 1,
row 0 *through k, end with* 1 *on the table's diagonal, C(i, i)=* 1

	0	1	2	3	4	5	……	k-1	k
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
k	1								1
…									
n-1	1							$C(n\text{-}1,k\text{-}1)$	$C(n\text{-}1,k)$
n	1								$C(n,k)$

看图填表，等于某个元素等于上边加对角

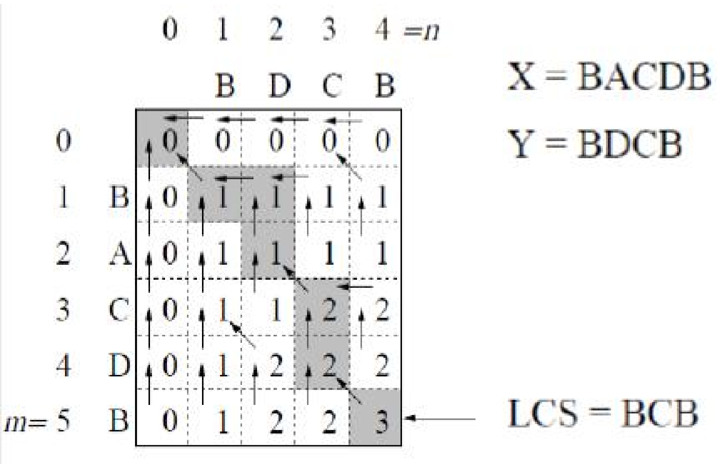
（2） 最长公共子序列（LCS）

LCS问题概述

最长公共子序列问题是在两个序列中找到一个最长的子序列（不要求连续），这个子序列在两个原始序列中都出现。

$$c[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0\ ,\\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j\ ,\\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i\neq y_j\ .\end{cases}$$

1. **基本情况**：当其中一个序列长度为0时，LCS长度为0。
2. **字符匹配**：当 `xᵢ = yⱼ` 时，LCS长度等于去掉这两个字符后的子问题的LCS长度加1。
3. **字符不匹配**：当 `xᵢ ≠ yⱼ` 时，LCS长度等于两个子问题的较大值：
   - 去掉 `xᵢ` 后的LCS长度
   - 去掉 `yⱼ` 后的LCS长度



（3） 动态矩阵乘法

问题描述

给定一系列矩阵 $A_1, A_2, \ldots, A_n$，其中矩阵 A_i 的维度为 $p_{i-1} \times p_i$，找到最优的矩阵乘法顺序，使得计算这些矩阵乘积所需的标量乘法次数最少。

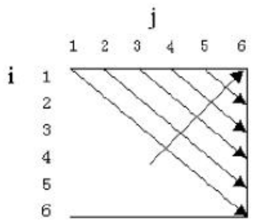
$$m[i,j]=\begin{cases} 0 & i=j \\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j \end{cases}$$

参数说明:

- $m[i,j]$：计算矩阵 A_i 到 A_j 乘积所需的最少乘法次数
- p_{i-1}, p_i, p_j：矩阵的维度参数
  - 矩阵 A_i 的维度是 $p_{i-1} \times p_i$
  - 矩阵 A_j 的维度是 $p_{j-1} \times p_j$
- k：分割点，表示在 A_k 和 A_{k+1} 之间进行分割

- $p_{i-1}p_kp_j$：将这两个结果矩阵相乘所需的乘法次数
  - 因为 $A_i \ldots A_k$ 的结果矩阵维度是 $p_{i-1} \times p_k$
  - $A_{k+1} \ldots A_j$ 的结果矩阵维度是 $p_k \times p_j$
  - 它们相乘需要 $p_{i-1} \times p_k \times p_j$ 次标量乘法

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25



(a) 计算次序

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(b) m[i][j]

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i][j]

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000 \\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125 \\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375 \end{cases}$$

$$=7125$$

$$\therefore\ s[2][5]=3$$

s[i][j] 表示最优分割点

（4） 0-1 背包问题

$$V(i,j)=\begin{cases}\max\{V(i-1,j),V(i-1,j-w_i)+v_i\} & j\ge w_i\\ V(i-1,j) & 0\le j<w_i\end{cases}$$

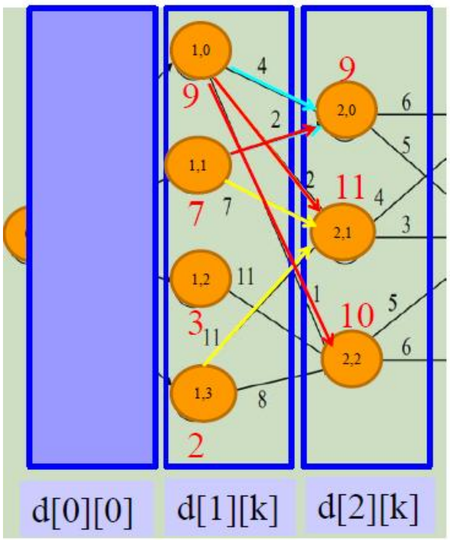
1. 当 $j \geq w_i$ **（当前背包可以装下第 i 个物品）**：
   - 我们需要在两种选择中取最大值：
     - **不选第 i 个物品**：价值保持为 $V(i-1,j)$
     - **选第 i 个物品**：价值为 $V(i-1,j-w_i)+v_i$（前 $i-1$ 个物品在容量 $j-w_i$ 下的最大价值加上当前物品的价值）
2. 当 $0 \leq j < w_i$ **（当前背包无法装下第 i 个物品）**：
   - 只能选择不装第 i 个物品，价值保持为 $V(i-1,j)$

	i	0	1	2	3	4	5			
	0	0	0	0	0	0	0	$V(i\text{-}1, j\text{-}w_1)+v_1$		$V(i\text{-}1, j)$
$w_1=2.\ v_1=12$	1	0	0	12	12	12	12	$V(i\text{-}1, j\text{-}w_2)+v_2$	$V(i\text{-}1, j)$	$V(i, j)$
$w_2=1.\ v_2=10$	2	0	10	12	22	22	22		$V(i, j)$	
$w_3=3\ \ v_3=20$	3	0	10	12	22	30	32			
$w_4=2.\ v_4=15$	4	0	10	15	25	30	37			

列是背包容量，最大为5；填表顺序是一行一行从左到右填写

37是用min{（3,3）+15，（3,5）}填写的

（5） 多阶段决策过程



d[1][k]:
d[1][0]= 9; d[1][1]= 7
d[1][2]= 3; d[1][3]= 2

d[2][k]:
d[2][0]=min{d[1][0]+w[1][0][0], d[1][1]+w[1][1][0]}}
=min{d[1][k]+w[1][k][0]}
=min{9+4;7+2} =9

d[2][1]=min{d[1][k]+w[1][k][1]}
=min{9+2;7+7;2+11} =11

d[2][2]=min{d[1][k]+w[1][k][2]}
=min{9+1;3+11;2+8}=10

w[0][0][0]=9　　w[1][3][2]=8

$$d[\mathrm{i}][\mathrm{j}] = \min_{\substack{j \in V_i \\ (\mathrm{k},\mathrm{j}) \in \mathrm{E}}} \{\mathrm{d}[\mathrm{i}-1][\mathrm{k}] + \mathrm{w}[\mathrm{i}-1][\mathrm{k}][\mathrm{j}]\}$$

$$path[\mathrm{i}][\mathrm{j}] = \arg\min_{k} \{\mathrm{d}[\mathrm{i}-1][\mathrm{k}] + \mathrm{w}[\mathrm{i}-1][\mathrm{k}][\mathrm{j}]\}$$

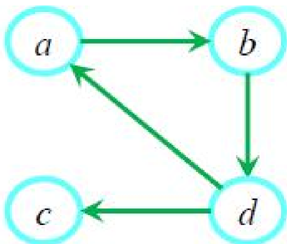
参数说明:

- i：当前阶段（从1开始）
- j：当前阶段的状态
- $d[i][j]$：到达第i阶段状态j的最小成本/最短距离
- V_j：可以转移到状态j的前一阶段状态的集合
- E：状态转移边的集合
- $w[i-1][k][j]$：从第i-1阶段状态k转移到第i阶段状态j的成本/距离

$path[i][j]$：记录到达第i阶段状态j的最优路径中，前一阶段的状态k

（6） Warshall 传递闭包算法

- *Example*1



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

a -> b
d -> a ⇨ d -> a->b

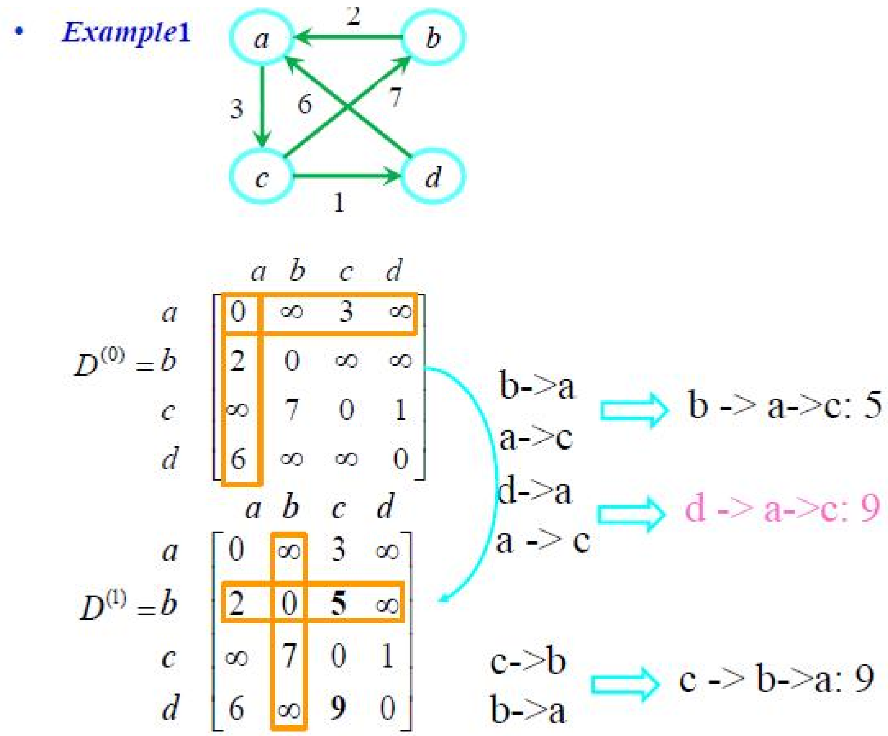
$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 0 \end{array}$$

a -> b
b ->d ⇨ a -> b->d

b ->d
d ->b ⇨ d-> b->d

$O(n^3)$, n 是顶点个数

（7）　最短路径的弗洛伊德算法



十 贪心策略

1. 贪心算法是一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是全局最佳或最优的算法策略。贪心算法不保证会得到最优解，但在某些问题中，贪心策略产生的解是最优的。

2. **贪心算法的特点：**

（1）局部最优选择（2）无回溯（3）简单高效

3. **适用场景：**具有**“贪心选择性质**”和**“最优子结构”**

- **贪心选择性质**：可以通过做出局部最优的选择来构造全局最优的解决方案。换句话说，一个全局最优解包含了它构成中的局部最优解。
- **最优子结构**：一个问题的最优解包含其子问题的最优解。这意味着问题可以通过解决其子问题并组合这些解来解决。

4.实例

(1)**活动选择问题**：给定一组活动，每个活动都有一个开始时间和结束时间。目标是选择最大

数量的互不重叠的活动。解决方案是按照活动结束时间的早晚来选择活动。

(2)**霍夫曼编码**：用于数据压缩的贪心算法。创建最优前缀代码，使得常见的字符使用较短的代码，不常见的字符使用较长的代码。

(3)**最小生成树问题**：如Prim算法和Kruskal算法，这些算法贪心地选择边来确保在满足所有顶点被包含在树中的同时，总边的权重最小。

Prim算法

核心思想：
Prim算法是一种**贪心算法**，用于求解**加权无向图的最小生成树（MST）**。它从任意一个顶点开始，逐步扩展生成树，每次选择**当前生成树连接的最小权值边**，并将该边连接的顶点加入生成树，直到所有顶点都被包含。Prim算法适用于**稠密图**，通常使用**优先队列（堆）**优化，时间复杂度为 **O(E log V)** 或 **O(V²)**（取决于实现方式）。

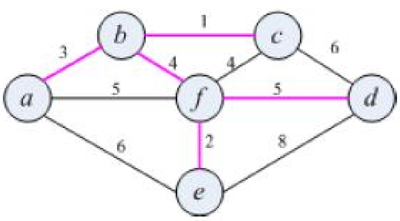
Kruskal算法

核心思想：
Kruskal算法同样用于求解**最小生成树（MST）**，但采用**边排序+并查集**的策略。它**按权值从小到大遍历所有边**，每次选择**不会形成环的最小权值边**加入生成树，直到所有顶点连通。Kruskal算法适用于**稀疏图**，时间复杂度主要由排序决定，为 **O(E log E)**（或 **O(E log V)**）。

Prim选点 Kruskal选边

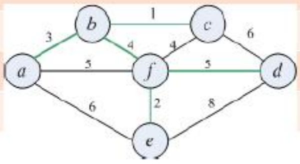
- *Example : Prim*

Tree vertices	Remaining vertices
a(-,0)	b(a,3) c(-,∞) d(-, ∞) e(a,6) f(a,5)
b(a,3)	c(b,1) d(-, ∞) e(a,6) f(b,4)
c(b,1)	d(c, 6) e(a,6) f(b,4)
f(b,4)	d(f, 5) e(f,2)
e(f,2)	d(f, 5)
d(f, 5)	



Example: Kruskal's

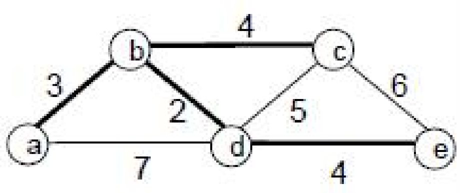
Tree edges	Sorted list of edges									
	bc	ef	ab	bf	cf	af	df	ae	cd	de
	1	2	3	4	4	5	5	6	6	8
bc										
1										
ef										
2										
ab										
3										
bf										
4										
df										
5										



(4)**单源最短路径问题**：如Dijkstra算法，该算法贪心地选择将最近的未处理顶点加入到已处理集合中，从而找到从源点到所有其他顶点的最短路径。

- ***Example : Dijkstra's***

Tree vertices	Remaining vertices
a(-,0)	b(a,3) c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) d(b,3+2) e(-,∞)
d(b,5)	c(b,7) e(d,5+4)
c(b,7)	e(d,9)
e(d,9)	



(5)找零问题

(6)分数背包问题

贪心选择性质

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。

	动态规划（DP）	贪心算法（Greedy Method）
子问题上	每步所做的选择往往依赖于子问题的解，只有在解出相关子问题后才能作出选择	仅在当前状态下作出最好选择，即局部最优选择，然后再去作出这个选择后产生的相应的子问题，不依赖于子问题的解
求解方式	通常以自底向上的方式解各子问题	通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题

一. 单选题 7） 对于函数$f(n)=10^n$和$g(n)=\log(n^2)$，当n趋于无穷大时，分析函数的渐进阶，确定$f(n)$和$g(n)$的关系属于（）。**(2分)**

- $f(n)=O(g(n))$：$f(n)$ 的增长不超过 $g(n)$（上界）。
- $f(n)=\Omega(g(n))$：$f(n)$ 的增长不低于 $g(n)$（下界）。
- $f(n)=\Theta(g(n))$：$f(n)$ 和 $g(n)$ 同阶（紧确界）。
- $f(n)=o(g(n))$：$f(n)$ 的增长严格慢于 $g(n)$（高阶无穷小）。
  - 注意：题目中 $f(n)$ 是 $g(n)$ 的高阶无穷大，因此 $f(n)=\omega(g(n))$，但选项中没有 ω，所以需要反向思考。

背诵

递归算法的时间复杂度分析步骤

递归算法的时间复杂度分析通常遵循以下步骤，核心是**建立递归关系式并求解**：

---

1. 确定递归结构

- **明确递归的终止条件（Base Case）**：
  分析递归结束时的直接解（如 `n=0` 或 `n=1` 的时间复杂度 `T(1) = C`）。
- **写出递归公式（Recurrence Relation）**：
  根据递归调用关系，表达 `T(n)` 与子问题 `T(n/k)` 或 `T(n-1)` 的关系（如 `T(n) = a·T(n/b) + f(n)`）。

2. 展开递归树（Recursion Tree）

- **绘制递归调用树**：
  将递归分解过程可视化，计算每一层的总时间复杂度。
- **求和所有层的代价**：
  累加递归树中所有层的时间复杂度（可能涉及等比级数或调和级数求和）。

3. 应用主定理（Master Theorem）

若递归式为分治形式 `T(n) = a·T(n/b) + O(n^d)`，直接通过主定理判断：

- **Case 1**：若 `d < log_b a`，则 `T(n) = θ(n^{log_b a})`。
- **Case 2**：若 `d = log_b a`，则 `T(n) = θ(n^d log n)`。
- **Case 3**：若 `d > log_b a`，则 `T(n) = θ(n^d)`。

4. 替代法（Substitution Method）

- **猜测时间复杂度形式**（如 `T(n) = O(n log n)`）。
- **用数学归纳法验证**：假设子问题成立，证明原问题也成立。

5. 处理特殊递归

- **尾递归**：可优化为迭代，时间复杂度通常为 `O(n)`。
- **多重递归**（如斐波那契）：需避免重复计算（如用动态规划优化）。

三种求最短路径问题的算法的区别和联系

1. 多段图法（Multistage Graph）

- **适用场景**：适用于有向无环图（DAG），尤其是可以划分为多个阶段（stage）的图，每个阶段包含若干节点，且边仅存在于相邻阶段的节点之间。
- **算法思想**：采用动态规划的方法，从终点逆向计算每个节点到终点的最短路径，或从起点正向计算起点到每个节点的最短路径。
- **时间复杂度**：通常为$O(n^2)$，其中n为节点数。
- **特点**：只能用于特定结构的图（多段图），无法处理带环或任意结构的图。

2. 迪杰斯特拉（Dijkstra）算法

- **适用场景**：适用于带非负权重的有向或无向图，求解单源最短路径问题（从一个起点到所有其他节点的最短路径）。
- **算法思想**：采用贪心策略，每次从未处理的节点中选择距离起点最近的节点，并更新其邻居节点的最短距离。
- **时间复杂度**：使用优先队列时为$O((V+E)\log V)$，其中V为节点数，E为边数。
- **特点**：不能处理负权边，因为负权边可能导致已确定的最短路径被破坏。

3. 弗洛伊德（Floyd）算法

- **适用场景**：适用于带正权或负权的有向或无向图（不能有负权环），求解所有节点对之间的最短路径。
- **算法思想**：采用动态规划，通过三重循环逐步更新任意两点之间的最短路径，允许中间节点逐步扩展。
- **时间复杂度**：$O(V^3)$，其中V为节点数。
- **特点**：可以处理负权边，并能检测负权环，但时间复杂度较高，适合稠密图或节点数较少的情况。

联系与区别

- **联系**：
  - 三者均用于求解最短路径问题。
  - 迪杰斯特拉算法和弗洛伊德算法可以用于一般图，而多段图法仅适用于特定结构的图。
- **区别**：
  - **适用范围**：多段图法仅适用于多段图；迪杰斯特拉算法适用于单源非负权图；弗洛伊德算法适用于全源带权图（可含负权）。
  - **功能**：迪杰斯特拉算法解决单源问题，弗洛伊德算法解决全源问题，多段图法解决特定结构问题。
  - **负权处理**：迪杰斯特拉不能处理负权，弗洛伊德可以（无负权环时）。
  - **效率**：迪杰斯特拉算法在稀疏图中效率较高，弗洛伊德算法适合稠密图或小规模图。

(1) 分支限界算法的基本思想（5分）

分支限界算法是一种用于求解组合优化问题的智能搜索方法，其基本思想如下：

1. **分支（Branch）**：将问题的解空间分解为若干子集（子问题），通常通过固定部分变量的取值来实现。
2. **限界（Bound）**：对每个子问题计算一个上界（或下界），用于剪枝。若子问题的上界小于当前已知的最优解，则舍弃该子问题，避免无效搜索。
3. **优先队列**：通常使用优先队列（如最大堆）存储待处理的子问题，优先扩展上界更优的子问题，以提高搜索效率。

(2) 解空间和解空间树的定义（5分）

- **解空间**：所有可能的0-1背包问题的解，即所有物品的选择组合（每个物品选或不选），共 $2^n = 16$ 种可能。
- **解空间树**：
  - 根节点表示初始状态（未选择任何物品）。
  - 每个节点分为左子树（选择当前物品）和右子树（不选择当前物品）。
  - 树的深度为物品数量 $n = 4$，叶子节点代表完整的解。

分支限界法 核心思想是系统地**遍历或构建决策树的所有可能的候选解**，通过**“分支”**来生成子候选解，并使用**“限界”**来避免不必要的搜索，即进行剪枝。

1. 最优子结构（Optimal Substructure）

定义

- 一个问题的最优解包含其子问题的最优解。
- 即，全局最优解可以通过组合子问题的最优解得到。

2. 重叠子问题（Overlapping Subproblems）

定义

- 在递归求解过程中，**相同的子问题被多次计算。**
- 动态规划通过**记忆化存储**来避免重复计算。

在求解子问题的过程中，按照**自顶向下**的**记忆化搜索**方法或者**自底向上**的**递推**方法求解出**子问题的解**，把结果存储在表格中。当需要再次求解此子问题时，直接从表格中**查询**即可，从而避免了大量的**重复计算。**

若一个优化问题的优化解包含它的**子问题**的优化解，则称其具有**最优子结构**