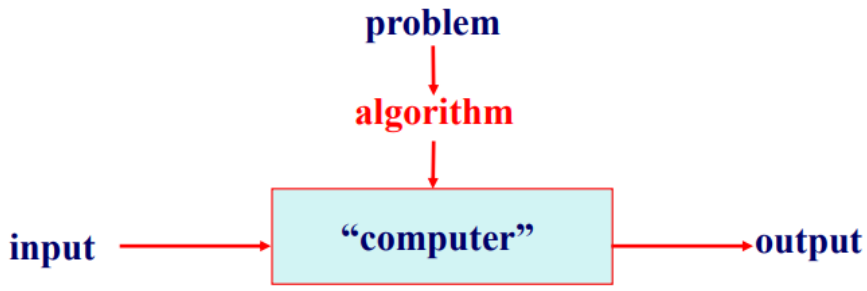
算法复习提纲

一、算法概述

算法基本概念与思想

算法：是一系列解决问题的清晰指令，也就是说，对于符合一定规范的输入，算法能够在有限时间内获得所要求的输出。
算法是解决问题的一种方法或过程，它是由若干条指令组成的有穷序列。



特征：

- **输入**：有零或多个外部量作为算法的输入。
- **输出**：算法产生至少一个量作为输出。
- **确定性**：组成算法的每条指令清晰、无歧义。
- **有效性**：算法中执行的任何计算步骤都是可分解为基本的可执行的操作步。
- **有限性**：算法中每条指令的执行次数有限，执行每条指令的时间也有限。

算法和程序的区别：

一、形式不同

1、算法：算法在描述上一般使用半形式化的语言。

2、程序：程序是用形式化的计算机语言描述的。

二、性质不同

1、算法：算法是解决问题的步骤。

2、程序：程序是算法的代码实现。

三、特点不同

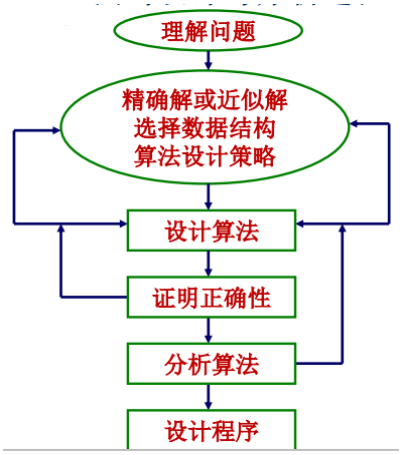
1、算法：算法要依靠程序来完成功能。

2、程序：程序需要算法作为灵魂。

程序是算法用某种编程语言的具体实现，不满足有限性。

常见问题类型：
排序、查找、字符串处理、图的问题、组合问题、几何问题、数值问题、方程问题等。

算法求解问题基本过程



二、算法效率分析基础

算法效率：算法效率的高低体现在运行该算法所需要耗费资源的多少。可分为时间效率和空间效率。

算法效率分析基本方法：
N：要解决的问题的规模

I：算法的输入

A：算法本身

C：复杂性

时间复杂性**T**：F（N，I，A）

空间复杂性**S**：F（N，I，A）

T = T（N，I），S = S（N，I）

函数渐近的界

设f 和g 是定义域为自然数集N上的函数

(1) f(n)=O(g(n)) 若存在正数c和n0使得对一切n≥n0有0≤f(n)≤cg(n)

(2) f(n)= Ω(g(n)) 若存在正数c和n0使得对一切n≥n0有0≤cg(n)≤ f(n)

(3) f(n)=o(g(n)) 对任意正数c存在n0使得对一切n≥n0有0≤f(n)

(4) f(n)=ω(g(n)) 对任意正数c存在n0使得对一切n≥n0有0≤cg(n)

(5) f(n)=Θ(g(n)) ⇔ f(n)=O(g(n)) 且f(n)=Ω(g(n))

(6) O(1)表示常数函数

符号	含义	通俗理解 f(x)在左
Θ	精确的渐近行为	相当于“=”
O	上界	相当于“<=”
o	松上界	相当于“<”
Ω	下界	相当于“>=”
ω	松下界	相当于“>”

(1) 若 $\lim_{n\to\infty}\frac{f(n)}{g(n)}=c$ ，c为大于0的常数，那么

$f(n)=\Theta(g(n))$

(2) 若 $\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$ ，那么

$f(n)=o(g(n))$

(3) 若 $\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$ ，那么

$f(n)=\omega(g(n))$

- 设f, g, h 是定义域为自然数集N上的函数：

(1) 如果$f=O(g)$且$g=O(h)$，那么$f=O(h)$.

(2) 如果$f=\Omega(g)$且$g=\Omega(h)$，那么$f=\Omega(h)$.

(3) 如果$f=\Theta(g)$和$g=\Theta(h)$，那么$f=\Theta(h)$.

(4) $O(f(n))+O(g(n)) = O(\max\{f(n),g(n)\})$

(5) $O(f(n))+O(g(n)) = O(f(n)+g(n))$

(6) $O(f(n))*O(g(n)) = O(f(n)*g(n))$

算法基本渐进效率类型：

函数渐进的界

算法基本渐进效率类型

类 型	名 称	注 释
1	常量	为数极少的效率最高的算法，难以举出几个例子。通常，当输入规模变得无穷大的时候，算法的执行时间也会趋于无穷大。
logn	对数	算法每次循环都会消去问题规模的一个常数因子。
n	线性	扫描规模为n的列表（如顺序查找）算法。
nlogn	n-logn-n	诸多分治算法，包括合并排序和快速排序平均效率属于此类型。
n^2	二次	一般来说，包含两重嵌套循环的算法。n 阶方阵的某些特定算法。
n^3	三次	一般来说，包含三重嵌套循环的算法。
2^n	指数	求n个元素集合的所有子集的算法。“指数”这个术语常用在更广泛的层面上，不仅包括这种类型，还包括那些增长速度更快的算法如阶乘。
n!	阶乘	求n个元素集合的完全排列的算法。

13

算法复杂性分析基本步骤：

- 确定表示输入**规模的参数**。
- 找出算法的**基本操作**。
- 检查基本操作的执行次数是否只依赖于输入规模。这决定**是否需要考虑最差、平均以及最优情况**下的复杂性。
- 对于非递归算法，建立算法基本操作执行次数的求和表达式；对于递归算法，建立算法基本操作执行次数的递推关系及其初始条件。
- 利用求和公式和法则建立一个操作次数的闭合公式，或者求解递推关系式，确定增长的阶。

非递归算法复杂性分析常见求和公式：

- 等差数列：$\sum_{k=1}^{n} a_k$ ，Sn=n*a1+n(n-1)d/2或Sn=n(a1+an)/2
- 等比数列：$\sum_{k=0}^{n} aq^k$ ，$S_n = a_1 \frac{1-q^n}{1-q}$
- 调和级数（发散的）：$\sum_{k=1}^{n} \frac{1}{k}$
- 对数求和：$\sum_{i=1}^{n} \log i$

非递归算法的复杂性分析：

- **算法输入规模：**可以用数组元素个数n度量
- **基本操作：**比较与赋值两种，选择比较
- 比较操作执行次数与输入相关，需要考虑**最坏、平均 、最好**情况
- 建立基本操作执行次数求和表达式（最好，最坏情况的表达式）
- 确定增长的阶

递归算法的复杂性分析：

方程一（a为每一个函数中递归函数使用次数）：

$$T(n)=\begin{cases}O(1) & n=1\\ aT(n\text{-}1)+f(n) & n>1\end{cases}$$

$$T(n)=a^{n-1}\,T(1)+\sum_{i=2}^{n}a^{n-i}f(i)$$

示例：

（1）若取$a=2$，$f(n)=O(1)$，汉诺塔问题
$T(n)=O(2^n-1)$
（2）若取$a=1$，$f(n)=n-1$，插入排序最坏情况
$T(n)=O(n^2)$

方程二（b为当前问题的规模为上一轮次的b分之一）：

$$T(n)=\begin{cases}O(1) & n=1\\ aT(\frac{n}{b})+f(n) & n>1\end{cases}$$

$$T(n)=n^{\log_b a}T(1)+\sum_{j=0}^{\log_b n-1}a^j f(\frac{n}{b^j})$$

示例：

（1）若$f(n)=c$ 实例

$$T(n)=\begin{cases}O(n^{\log_b a}) & a\neq 1\\ O(\log n) & a=1\end{cases}$$

（2）若$f(n)=cn$ 实例

$$T(n)=\begin{cases}O(n) & a<b\\ O(n\log n) & a=b\\ O(n^{\log_b a}) & a>b\end{cases}$$

算法思想与特点、适用条件、核心要素、算法基本步骤、程序设计、效率分析、具体应用

四、分治法

思想：

将一个难以直接解决的大问题，**分解**成规模较小的**相同子问题**，直至这些子问题容易直接求解，并可以利用这些子问题的解求出原问题的解。

适用条件/特点：

- 问题的规模小到一定程度就可以容易解决。
- 问题可以分解成若干小问题。
- 子问题的解可以合并成原问题的解。
- 子问题相对独立，子问题之间不包含公共子问题。

如果各子问题是不独立的，则 分治法要做许多不必要的工作，重复地解公共的子问题，此时 虽然也可用分治法，但一般用**动态规划**较好。

基本步骤：

- **划分**：把规模为n的原问题划分为k个规模大致相同的子问题，并尽量使这k个子问题的规模大致相同。
- **求解子问题**：求解各个子问题的解。各子问题的解法与原问题的解法通常是相同的，可以用递归的方法求解各个子问题，有时递归处理也可以用循环来实现。
- **合并**：将各个子问题的解合并，得出原问题的解。合并的代价因情 况不同有很大差异，分治算法的有效性很大程度上依赖于合并的实现。

案例：

排列问题（全排列）：

输出：n个数字的所有排列方法。

例 2.4 排列问题

设 $R=\{r_1,r_2,\cdots,r_n\}$ 是要进行排列的 n 个元素，$R_i=R-\{r_i\}$。集合 X 中元素的全排列记为 perm(X)。(r_i)perm(X) 表示在全排列 perm(X) 的每一个排列前加上前缀 r_i 得到的排列。R 的全排列可归纳定义如下：

当 $n=1$ 时，perm$(R)=(r)$，其中 r 是集合 R 中唯一的元素；

当 $n>1$ 时，perm(R) 由 (r_1)perm(R_1)，(r_2)perm(R_2)，$\cdots$，(r_n)perm(R_n) 构成

伪代码：

```
算法 perm(Type list[], int k, int m)
//生成列表list的全排列
//输入：一个全排列元素列表list[0..n-1]
//输出：list的全排列集合
//k为最左边数字的序号，m为最右边数字的序号
if k == m
    for i←0 to m do
        输出list[i]
else
    for i←k to m do
        swap list[k] and list[i]
        perm(list, k+1, m)
        swap list[k] and list[i]

```

复杂度分析： $T(n)=\begin{cases}O(1) & n=1\\ nT(n-1)+O(n) & n>1\end{cases}$

整数划分：

问题：将给定正整数n表示成一系列正整数之和 n=n1+n2+...+nk，其中n1≥n2≥...≥nk≥1，k≥1。 求正整数n的**不同划分个数**p(n)。

$$q(n,m)=\begin{cases}1 & n=1,m=1\\ q(n,n) & m>n\\ 1+q(n,n-1) & m=n\\ q(n,m-1)+q(n-m,m) & 1<m<n\end{cases}$$

n为需要划分的数字，m为最大加数。

```
int q(int n, int m) {
    if ((n < 1)||(m < 1)) return 0;
    if ((n == 1)||(m == 1)) return 1;
    if (n < m) return q(n,n);
    if (n==m) return q(n, m-1) + 1;
    return q(n, m-1) + q(n-m, m);
}
```

二分搜索：

问题：给定已按升序排好序的n个元素a[0:n-1]，现要在这n个 元素中找出一特定元素x。

如果是无序数组，从第0个位置开始遍历搜索，平均时间复杂度：O(n)，平均时间复杂度：O(n)。

如果是有序数组，可以使用二分搜索，最坏时间复杂度为O(logn)。

大整数乘法：

问题：设X和Y都是n位二进制数，设计一个有效算法计算它们的乘积XY。

直接加运算时**复杂度**：O（n^2）。

使用分治法：

X = AB；Y = CD

$$XY=\left(A*10^{\frac{n}{2}}+B\right)\left(C*10^{\frac{n}{2}}+D\right)=AC*10^{n}+(AD+BC)*10^{\frac{n}{2}}+BD$$

$$XY=AC*10^{n}+((A-B)(D-C)+AC+BD)*10^{\frac{n}{2}}+BD$$

$$T(n)=\begin{cases}O(1) & n=1\\ 3T(n/2)+O(n) & n>1\end{cases}$$ （系数3是因为进行了3次n/2位乘法）

$T(n)=O(n^{\log 3})=O(n^{1.59})$

n大于600位时， 分治法性能才超越传统算法。

矩阵乘法：

问题：计算A和B的乘积矩阵C。

传统方法： 依据定义来计算A和B的乘积矩阵C，则每计算C的一个元 素C[i][j]，需要做n次乘法和n-1次加法。因此，算出矩阵C 的 n^2个元素所需的计算时间为O（n^3）。

使用分治法：

问题足够小情况：如果n=2，则2个2阶方阵的乘积可以直接计算出来，共需要8次乘法和4次加法。

分治思想：将矩阵A，B和C中每一矩阵都分块成4个大小相等的子矩阵。由此可将方程 C=AB 重写为：

$$\begin{bmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{bmatrix}=\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix}$$

$$\begin{aligned}C_{11}&=A_{11}B_{11}+A_{12}B_{21}\\ C_{12}&=A_{11}B_{12}+A_{12}B_{22}\\ C_{21}&=A_{21}B_{11}+A_{22}B_{21}\\ C_{22}&=A_{21}B_{12}+A_{22}B_{22}\end{aligned}$$

计算2个n阶方阵的乘积可以转化为计算8个n/2阶方阵的乘 积和4个n/2阶方阵的加法。加法可以在O (n^2)时间内完成。

复杂度分析： $T(n)=\begin{cases}O(1) & n=2\\ 8T(n/2)+O(n^2) & n>2\end{cases}$

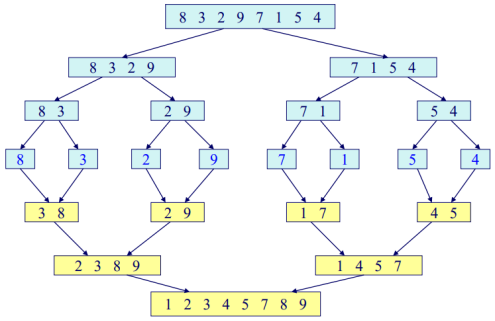
Strassen乘法:

复杂度分析： $T(n)=\begin{cases}O(1) & n=2\\ 7T(n/2)+O(n^2) & n>2\end{cases}$

合并排序:

问题：给定一个可排序的n个元素序列（数字、字符 或字符串），对它们按照非降序方式重新排列。

基本思想：将待排序元素分成大小大致相同的2个子集合，分 别对2个子集合进行排序，最终将排好序的子集合合并成为所 要求的排好序的集合。



复杂度分析： $T(n)=\begin{cases}O(1) & n\le 1\\ 2T(n/2)+O(n) & n>1\end{cases}$

快速排序:

问题：给定一个可排序的n个元素序列（数字、字符 或字符串），对它们按照非降序方式重新排列。

```
quicksort(A, lo, hi)
  if lo < hi
    p = partition(A, lo, hi)
    quicksort(A, lo, p - 1)
    quicksort(A, p + 1, hi)

partition(A, lo, hi)
    pivot = A[hi]
    i = lo //place for swapping
    for j = lo to hi - 1
        if A[j] <= pivot
            swap A[i] with A[j]
            i = i + 1
    swap A[i] with A[hi]
    return i
```

复杂度分析：O(nlogn)

线性时间选择:

定义：给定线性序集中n个元素和一个整数k，1≤k≤n，要求找出这n个元素中第k小的元素。

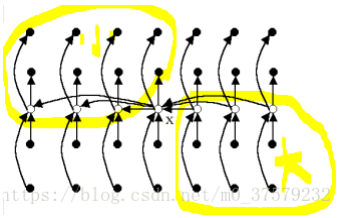
步骤：

（1）将n个输入元素划分成n/5（向上取整）个组，每组5个元素，最多只可能有一个组不是5个元素。用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共n/5（向上取整）个。
（2）递归调用select来找出这n/5（向上取整）个元素的中位数。如果n/5（向上取整）是偶数，就找它的2个中位数中较大的一个。以这个元素作为划分基准。

设中位数的中位数是x，比x小和比x大的元素至少3*(n-5)/10个，原因：

3*(n/5-1)*1/2
3—中位数比x小的每一组中有3个元素比x小
n/5-1—有5个数的组数
1/2—大概有1/2组的中位数比x小

而当n≥75时，3(n-5)/10≥n/4所以按此基准划分所得的2个子数组的长度都至少缩短1/4，也就是说，长度最长为原长度的3/4。



如图，划分的部分左上是肯定比x小的（大概占1/4）右下是肯定比x大的（大概占1/4）左下和右上不确定，就算这两部分同时不比x小或比x大，划分成的子区间也能至少缩短1/4!

复杂度分析：

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

$T(n)=O(n)$

最近点对问题：

问题：给定平面上的n个点，找其中的一对点，使得在n个点组成的所有点对中，该点对之间的距离最小。

分治思想：将平面上n个点的集合S划分为两个子集S_1和S_2， 每个子集约n/2个点，然后在每个子集中寻找最接近点对。 问题在于合并：如何根据子集中的最接近点对求得原集合中的最接近点对。

棋盘覆盖问题

问题：在一个2^k x 2^k (k ≥0) 个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为特殊方格。显然，特殊方格在棋盘中可能出现的位置有4^k种，因而有4^k种不同的棋盘。棋盘覆盖问题要求用4种不同形状的L型骨牌覆盖给定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。

当$n = 2^k$

$$T(n) = \begin{cases} O(1) & n = 1 \\ 4T(n/2) + O(1) & n > 1 \end{cases}$$

$T(n) = O(n^2) = O(4^k)$

$(4^k - 1) / 3 = \Omega(4^k) = \Omega(n^2)$

覆盖一个2k x 2k期盼所需的L型骨牌个数为（4k-1)/3，该算法为渐近意义下的最优算法。

五、动态规划

思想：和分治法类似，将原问题分解成多个阶段或子问题，然后按顺序求解各子问题。
最后一个阶段或子问题的解就是初始问题的解。
DP的子问题往往不是相互独立。
DP的不同子问题的数目常常只有多项式级。分治法求解时，有些子问题被重复计算了许多次，从而导致分治法求解问题时间复杂度较高。
DP的基本思想是保留已解决的问题的子问题，在需要时再查找已求得的解，避免重复计算。

特点/适用条件：
求解问题是组合优化问题
求解过程需要多步判断，从小到大以此求解
子问题目标函数最优解之间存在依赖关系（原问题最优解是由子问题最优解构成）

基本步骤：
找出最优解性质，并刻画其结构特征
递推的定义最优值
自底向上计算出最优解
根据计算最优解时得到的信息，构造最优解，递推方程

要素：
最优子结构
重叠子问题
备忘录（存放解的额外空间）

案例：

矩阵连乘问题：

问题：给定n个矩阵{A1，A2，…，An }，其中，Ai与Ai+1是可乘 的，i=1, 2, …，n-1。确定矩阵乘法顺序，使得元素相乘的次数最少。（矩阵连乘次序对计算量有很大影响。）

蛮力法：

搜索所有可能的计算次序，并计算出每种计算次序相应需要的数乘次数，从中找出一种数乘次数最少的计算次序。设不同计算次序为P(n)。

复杂度分析

$$P(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & n>1\end{cases}$$

该递归方程解为Catalan数（ $C(n)=\frac{1}{n+1}\binom{2n}{n}$ ）
$P(n)=\Omega(4^n/n^{3/2})$

动态规划：

说明：
将矩阵连乘积$A_iA_{i+1}\dots A_j$简记为A[i:j]，i≤j。
考察计算A[i:j]的最优计算次序。设这个计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开，i≤k<j，则其相应完全加括号方式为$(A_iA_{i+1}\dots A_k)(A_{k+1}A_{k+2}\dots A_j)$
计算量：
A[i:k]的计算量加上A[k+1:j]的计算量，再加上A[i:k]和A[k+1:j]相乘的计算量。

（1）分析最优解结构:
➢计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的。
➢矩阵连乘计算次序问题的最优解包含着其子问题的最优解，满足最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法求解的显著特征。

（2）建立递推关系

1.设计算A[i:j]，1≤i≤j≤n，所需要的最少数乘次数m[i,j]，则原问题的最优值为m[1,n]。

2.递推方程

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

推导

K的位置只有j-i种可能

（1）分析最优解结构:
➢计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的。
➢矩阵连乘计算次序问题的最优解包含着其子问题的最优解，满足最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法求解的显著特征。

（2）建立递推关系

1.设计算A[i:j]，1≤i≤j≤n，所需要的最少数乘次数m[i,j]，则原问题的最优值为m[1,n]。

2.递推方程

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

推导

K的位置只有j-i种可能

(3) 计算最优值—递归求解

```
int RecurMatrixChain(int i, int j, int *p, int **s) {
    if(i == j) return 0;
    int u = RecurMatrixChain(i, i) + RecurMatrixChain(i+1, j) +
p[i1]*p[i]*p[j];
    s[i][j] = i;
    for(int k = i+1; k < j; k++){
        int t = RecurMatrixChain(i, k) + RecurMatrixChain(k+1, j) +
p[i1]*p[k]*p[j];
    if(t < u) {u = t; s[i][j] = k;}
    }
    return u;
}

```

复杂度分析：

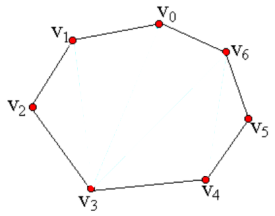
$$T(n)\geq\begin{cases}O(1) & n=1\\ \sum_{k=1}^{n-1}(T(k)+T(n-k)+O(1)) & n>1\end{cases}$$

$$T(n)\geq O(n)+\sum_{k=1}^{n-1}T(k)+\sum_{k=1}^{n-1}T(n-k)=O(n)+2\sum_{k=1}^{n-1}T(k)$$

$$T(n)\geq 2^{n-1}$$

凸多边形最优三角剖分：

凸多边形的表示：多边形顶点的逆时针序列表示。 例：P={v0，v1，v2，v3，v4，v5，v6 }



弦：两个不相邻顶点间的直 线线段。弦将多边形分割为 两个子多边形。

问题：给定凸多边形P={v0，v1，… ，vn-1 }，以及定义在由凸多边形的边和弦组成的三角形上的权函数w。要求确定该凸多边 形的三角剖分，使得该三角剖分所对应的权，即三角剖分中诸三角形上权之和为最小。

复杂度分析：同矩阵连乘（只需改符号种类）

最长公共子序列：

若给定序列X={x1 ,x2 ,...,xm}，则另一序列Z={z1 ,z2 ,...,zk }是X的子序列是指，存在一个严格递增下标序列{i1 ,i2 ,...,ik }使得对于所有 j=1,2,...,k有：$z_j = x_{ij}$ 。

给定2个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的公共子序列。

X={A，B，C，B，D，A，B}，Y={B，D，C，A，B，A}，则序列Z={B， C，A}是X和Y的一个公共子序列。 序列W={B，C，B，A}是X和Y的一个公共子序列，并且是X和Y的最长公共子序列。

问题：给定2个序列X={x1 ,x2 ,...,xm}和Y={y1 ,y2 ,...,yn }，找出X和 Y的最长公共子序列。

蛮力法：求X和Y的所有公共子序列，找出最长的。 判断X一个子序列是否是Y子序列时间O(n)。 X有2^m个子序列。 最坏情况下时间复杂度O(n^{2m})。

分治法：

（1）最优子结构性质
设序列$X=\{x_1,x_2,\ldots,x_m\}$和$Y=\{y_1,y_2,\ldots,y_n\}$的最长公共子序列为$Z=\{z_1,z_2,\ldots,z_k\}$，则
(a)若$x_m=y_n$，则$z_k=x_m=y_n$，且z_{k-1}是x_{m-1}和y_{n-1}的最长公共子序列。
(b)若$x_m\neq y_n$且$z_k\neq x_m$，则Z是x_{m-1}和Y的最长公共子序列。
(c)若$x_m\neq y_n$且$z_k\neq y_n$，则Z是X和y_{n-1}的最长公共子序列。

（2）建立递归关系
用c[i][j]记录序列X_i和Y_j的最长公共子序列的长度。其中，$X_i=\{x_1,x_2,\ldots,x_i\}$；$Y_j=\{y_1,y_2,\ldots,y_j\}$。当i=0或j=0时，空序列是$X_i$和$Y_j$的最长公共子序列。故此时C[i][j]=0。其它情况下，由最优子结构性质可建立递归关系如下：

$$c[i][j]=\begin{cases}0 & i=0,j=0\\ c[i-1][j-1]+1 & i,j>0;x_i=y_j\\ \max\{c[i][j-1],c[i-1][j]\} & i,j>0;x_i\neq y_j\end{cases}$$

（3）计算最优值

C[i][j]存储X_i和Y_j的最长公共子序列的长度，b[i][j]记录C[i][j]的值是由哪一个子问题的解得到的，后面构造最长公共子序列时需要用到。

```
void LCSLength(int m, int n, char *x, char *y, int **c, int **b) {
    int i, j;
    for (i = 1; i <= m; i++) c[i][0] = 0;
    for (i = 1; i <= n; i++) c[0][i] = 0;
    for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++) {
            if (x[i]==y[j]) {
                c[i][j]=c[i-1][j-1]+1; b[i][j]=1;}
            else if (c[i-1][j]>=c[i][j-1]) {
                c[i][j]=c[i-1][j]; b[i][j]=2;}
            else { c[i][j]=c[i][j-1]; b[i][j]=3; }
    }
}
```

算法时间复杂度为O(mn)

（4）构造最优解

从b[m][n]开始，在数组b中搜索，当b[i][j]=1时，表示X_i和Y_j的最长公共子序列是由X_{i-1}和Y_{j-1}的最长公共子序列在尾部加上x_i所得到的子序列；当b[i][j]=2时，表示表示X_i和Y_j的最长公共子序列与X_{i-1}和Y_j的最长子序列相同；当b[i][j]=3时，表示表示X_i和Y_j的最长公共子序列与X_i和Y_{j-1}的最长子序列相同。

```
void LCS(int i, int j, char *x, int **b) {
    if (i ==0 || j==0) return;
    if (b[i][j]== 1){ LCS(i-1, j-1, x, b); cout<<x[i]; }
    else if (b[i][j]== 2) LCS(i-1, j, x, b);
    else LCS(i, j-1, x, b);
}
```

算法时间复杂度为O(m+n)。

- 最长公共子序列

		B	D	C	A	B	A
	c[m][n]	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

X={A，B，C，B，D，A，B}，Y={B，D，C，A，B，A}

b[m][n]	1B	2D	3C	4A	5B	6A
1A	2	2	2	1	3	1
2B	1	3	3	2	1	3
3C	2	2	1	3	2	2
4B	2	2	2	2	1	3
5D	2	2	2	2	2	2
6A	2	2	2	1	2	1
7B	2	2	2	2	2	2

1 2 3

最优二叉搜索树：

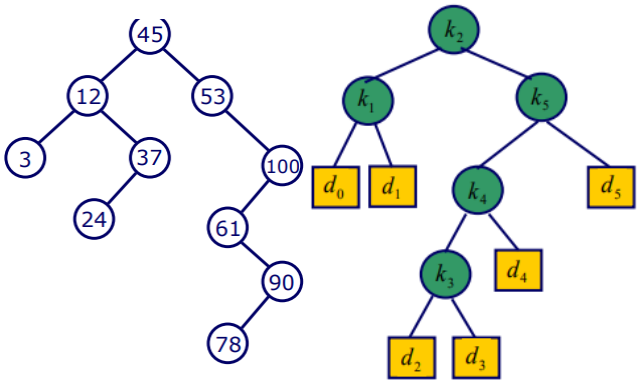
二叉排序树：

(1)若它的左子树不空，则左子树上所有 节点的值均小于它的根节点的值；

(2)若它的右子树不空，则右子树上所有 节点的值均大于它的根节点的值；

(3)它的左、右子树也分别为二叉排序树。

随机的情况下，二叉查找树的平均查找长度和 logn 是等数量级的。



设树的根节点深度为0。

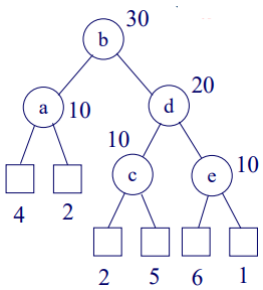
二叉搜索树的叶结点是形如(x_i , x~i+1 ~)的开区间。 在二叉搜索树中搜索一个元素x，返回结果 有两种情形：

(1)在树的内结点找到 x = x_i ；

(2)在树的叶结点中确定 x ∈ (x_i , x_{i+1}) 。

实例分析：

设数据集 S = { a, b, c, d, e } ， 存储S的二叉搜索树如图所示。 设被查找的 x 值对应树中各结点的分布 概率为 P = (4, **10**, 2, **30**, 2, **10**, 5, **20**, 6, **10**, 1) / 100



平均查找长度： L=0.3×1+(0.1+0.2) ×2+(0.1+0.1) ×3 +(0.04+0.02) ×2+(0.02+0.05+0.06+0.01) ×3 =2.04

问题描述：

给定数据集合S与存取概率分布P，求一颗平均查找次数（平 均查找长度）最小的二叉搜索树，即最优二叉搜索树。

> 蛮力法：
> 枚举具有n个结点的所有二分搜索树，计算平均查找长度，从中找出最优树。
> 复杂度与矩阵链相乘问题相同，为Catalan数，下界为$\Omega(4^n/n^{3/2})$

动态规划：

时间复杂度O(n^3) ,空间复杂度 O(n^2)

0-1背包问题：

设子问题最优值为 m(i，j)，即m(i，j)是背包容量为j，可选择物品为1, 2, … , i时0-1背包问题的最优值。由0-1背包问题的最 优子结构性质，可以建立计算m(i，j)的递归式如下：

$$m(i,j)=\begin{cases}\max\{m(i\text{-}1,j),m(i\text{-}1,j-w_i)+v_i\} & j\geq w_i\\ m(i\text{-}1,j) & 0\leq j<w_i\end{cases}$$

$$\begin{cases}m(0,j)=0\\ m(i,0)=0\end{cases}$$

实例求解分析：

n=5, c=10，w={2, 2, 6, 5, 4} v={6, 3, 5, 4, 6}

k \ c	1	2	3	4	5	6	7	8	9	10
$F_k(y)$	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9
3	0	0	6	6	9	9	9	11	14	14
4	0	0	6	6	9	9	9	10	11	14	14
5	0	0	6	6	9	9	12	12	15	15	15

最大子段和问题：

> 问题：给定n个整数（可以为负数）的序列（$a_1, a_2, \ldots, a_n$），求其最大子段和 $\max\{0, \max\limits_{1\leq i\leq j\leq n}\sum\limits_{k=i}^{j} a_k\}$

实例：

给定序列A = (-2, 11, -4, 13, -5, -2)

长度为1的子段和有： -2, 11, -4, 13, -5, -2

长度为2的子段和有：9, 7, 9, 8, -7

长度为3的子段和有：5, 20, 4, 6

长度为4的子段和有：18, 15, 2

长度为5的子段和有：13, 13

长度为6的子段和有：11

最大子段和为11 – 4 + 13 = 20

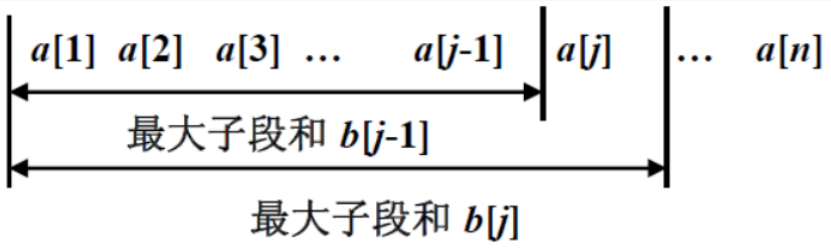
蛮力法：时间复杂度O(n^3)

设b[j]表示最后一项为a[j]的序列构成的最大的子段和最优解
b[1],b[2],…,b[n]中的最大值，即

$$b[j]=\max_{1\le i\le j}\{\sum_{k=i}^{j}a[k]\},1\le j\le n$$

可建立b[j]的递推方程

$$b[j]=\max\{b[j-1]+a[j],a[j]\}$$



54

```
int MaxSum (int n, int* a) {
    int sum = MIN, b = 0;
    for (i = 1; i <= n; i++) {
    if (b > 0) b += a[i];
    else b = a[i];
    if (b > sum) sum = b;
    }
    return sum;
}
```

实例：

给定序列A = (2, -5, 8, 11, -3, 4, 6)

i=1: b=a[i]=2, sum=b=2;

i=2: b=2+a[2]=-3, sum=2;

i=3: b=a[3]=8, sum=8;

i=4: b=8+a[4]=19, sum=19;

i=5: b=19+a[5]=16, sum=19;

i=6: b=16+a[6]=20, sum=20;

i=7: b=20+a[7]=26, sum=26.

时间复杂度O(n)

电路布线问题
图像压缩问题
最大子段和问题
投资问题

0-N背包问题
流水作业调度
序列匹配

六、贪心

基本思想：总是做出当前最优解，即局部最优解。贪心算法不能对所有问题都得到整体最优解，但是结果一般也是近似最优解。

在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

贪心算法中，较大子问题的解恰好包含了较小子问题的解作为子集，这与动态规划算法设计中的优化原则本质上是一致的。

动态规划算法在某一步决定优化函数的最大或最小值时，需要考虑到它的所有子问题的优化函数值，然后从中选出最优的结果；贪心算法的每步判断时，不考虑子问题的计算结果，而是根据当时情况采取"只顾眼前"的贪心策略决定取舍。

适用条件/特点：

- 最优子结构性质：一个问题的最优解包含其子问题的最优解。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。
- 贪心选择性质：所求问题的整体最优解可以通过一系列局部最优的选择来达到。

动态规划算法通常以自底向上的方式求解各子问题，而贪心算法则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。

对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。

> 假设有4种硬币，它们的面值分别为二角五分、一角、五分和一分，现要找给顾客四角八分钱，问如何选取个数最少的硬币？
>
> 如硬币选择动态规划，由小推大（小已知）；
>
> 而贪心由大推小（先取硬币面值大的看能取多少个，取完后问题缩小，再看下一个面值）
> 贪心性质就是局部最优选择可以得到全局最优结果！

案例：

活动安排：

问题：有n个活动申请使用同一个礼堂，每项活动有一个开始时间和一个截止时间，如果任何两个活动不能同时举行，问如何选择这些活动，从而使得被安排的活动数量达到最多？

策略：早完成的活动先安排。
把活动按照截止时间从小到大排序，使得f1 ≤ f2 ≤... ≤fn ，然后从前向后挑选，只要与前面选择的活动相容，便将这项活动选入最大相容集合A。

- 活动安排问题—实例计算过程

设待安排的11个活动的开始时间和结束时间按结束时间的非减序排列如下：

i	1	2	3	4	5	6	7	8	9	10	11
s[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

最终解为：A={1, 4, 8, 11}

背包问题和0-1背包问题：

背包问题可用贪心，0-1背包不行

0-1背包问题：给定n个物品和一个背包。物品 i 的重量为 w_i，其价值为 v_i，背包的容量为 c 。问应如何选择装入背包中的物品，使得装入背包中的物品总价值最大?
说明：同一物品不能装入多次，也不能装入物品的一部分。

背包问题：与 0-1背包问题类似，所不同的是在选择物品 i 装入背包时，可以选择物品的一部分，而不一定要全部装入背包。

背包问题的贪心算法：

首先计算每种物品单位重量的价值 v_i / w_i ，然后，依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。若将这种物品全部装入背包后，背包内的物品总重量未超过c，则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直地进行下去，直到背包装满为止。

算法的计算时间上界为 O(nlogn) 。

对于**0-1背包问题**，**贪心选择之所以不能得到最优解**是因为在这种情况下，它无法保证最终能将背包装满，部分闲置的背包空间使每公斤背包空间的价值降低了。

最优装载问题
最优前缀码问题（哈夫曼）
最小生成树问题（Prim（加点）和Kruskal（加边））
单源最短路径问题（Dijkstra算法）
多机调度问题
硬币找零问题

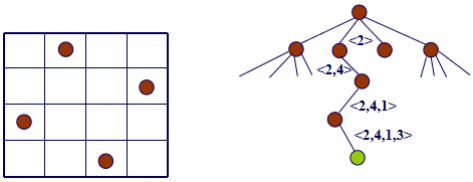
七、回溯

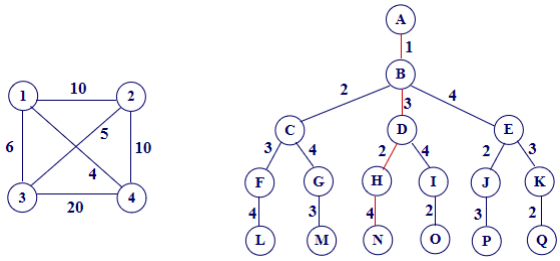
回溯法也称**试探法。**

思想：从初始状态出发，搜索所能达到的所有状态；当一条路走到头，再后退一步或若干步，从另一个状态出发继续搜索，直到所有路径都搜索过。（不断前进 、不断回溯寻找解）

核心元素：

- 将解空间组织成树，采用系统的方法搜索解空间树，有排列树和子集树。
- 搜索策略：深度优先为主，也可以采用广度优先、函数优先、广度深度结合等。
- 避免无效搜索策略：
  - 约束函数：在扩展节点处剪去不满足约束条件的子树。
  - 界限函数：在扩展节点处剪去得不到最优解的子树。

> N后问题：
> 在N×N的棋盘中放置N个皇后，使得任何两个皇后之间不能相互攻击（不在同一行，同一列，同一对角线），试给出所有的放置方法。
>
> 
>
>
> 4皇后问题的解空间可以表示为5层的树，每个节点都有0或者4个子节点，N皇后问题的一个解可以表示为树中的一条路径，从而可以将N皇后问题转换为树的搜索问题。N皇后问题，每一行都有N个可选的位置，所以问题的规模是n^n。

> 旅行售货员（货郎担）问题：某售货员要到若干城市去推销商品，已知各城市间的路程耗费（代价），如何选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使得总路程耗费最小。
>
> 
>
>
> TSP问题的解空间可以表示为N层的树，每个节点最多有N-1个子节点，TSP问题的一个解可以表示为树中的一条长度为N+1的路径，从而可以将TSP问题转换为树的搜索问题。通过这两个问题的回顾，我们可以知道，大部分问题的解空间，都可以将其表示为树状结构，从而将优化问题，转化为搜索问题，即可以用回溯法来解决。因此回溯法也被称为问题的通解。

> N皇后问题不能用贪心，动态规划，因为不是一个求最优值的问题，只能用蛮力法求解。
>
> TSP问题不能用贪心，动态规划求解，因为它不具备最优子结构，不满足最短路径上的一部分也一定是最短路径，
>
> 举个反例，上图中的1,3,2,4,1是最短路径，但1,3,2不是1到2的最短路径。得出结论这两个问题只能用已学的蛮力法求解，而蛮力法的复杂度是N^N的，很显然不太合适，
>
> 那回溯法相对会有什么好处呢？--回溯法能够方便的剔除不满足条件的解，从而避免不必要的搜。虽然回溯法能够避免不必要的搜索，但是它的复杂度还是很高的，所以对于大规模问题依然没法解决，所以需要新的方法。
>
> N后问题和TSP问题都可以用树表示其解空间，从而可以转换为一棵树的搜索问题，这种用搜索的方式解决问题的思路就是回溯法的思想。
>
> 思想：有些位置的可能解有多个，我们试探一个，然后假设我们试探的解为正确解，然后再重复前面的步骤。一条路走不通了，就掉头，回到上一次做选择的位置，重复这样的步骤，直到得到所有的路径都试探过。

解题思想：

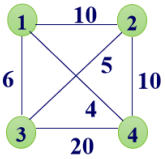
问题解的表示：将问题的解表示成一个n元式 (x1, x2, ..., xn) 的形式。

显示约束：对xi的取值约束。

隐式约束：为满足问题的解而对不同分量之间加的约束。

解空间：解向量满足约束条件的所有多元组组成的空间。

例如：



问题解的表示： (x1, x2, ..., x4)

显示约束：xi∈{1, 2, 3, 4}。

◆隐示约束：$(x_i, x_{i+1}) \in E$和$\forall i, k \in \{1,2,3,4\}, x_i \neq x_k$， $(x_n, x_1) \in E$

◆解空间：满足$x_i \in \{1,2,3,4\}$，$(x_i, x_{i+1}) \in E, \forall i, k \in \{1,2,3,4\}, x_i \neq x_k$，且$(x_n, x_1) \in E$的所有$(x_1, x_2, \ldots, x_4)$。

结点状态：

白结点（尚未访问）

灰结点（正在访问以该结点为根的子树）

黑结点（以该结点为根的子树遍历完成）

活结点：白节点+灰节点；

死节点：黑节点

存储：当前路径

回溯法适用问题：搜索问题和优化问题。

必要条件：多米诺性质

如果子结点满足约束条件能够推导出其父结点满足约束条件,那么就满足多米诺性质.

设 P(x1, x2, ..., xi) 是关于向量 <x1, x2, ..., xi> 的某个性质，那么 P (x1, x2, ..., xi+1) 真蕴含 P (x1, x2, ..., xi) 为真，即

P(x1, x2, ..., xi+1) → P(x1, x2, ..., x i) (0< i < n) (n为向量维数)

例如：

例如：求满足不等式的所有整数解问题

$5x_1 + 4x_2 - x_3 \leq 10 \quad x_i \in \{1, 2, 3\}, i = 1, 2, 3$

不满足多米诺性质

$<x_1, x_2, x_3> = <1, 2, 3>$，$P(x_1, x_2, x_3)$为真，但$P(x_1, x_2)$为假。

导致的问题：结点<1, 2>被误剪枝，导致结点<1, 2, 3>不可达，从而不能确保所有的解。

解决办法2：变量替换。替换为$5x_1 + 4x_2 + (x_4 - 4) \leq 10, x_4 \in \{1,2,3\}$，$5x_1 + 4x_2 + x_4 \leq 14; x_3 = 4 - x_4;$

回溯法设计要素

- 针对问题定义解空间：
    - 问题解向量
    - 解向量分量取值集合
    - 构造解空间树
- 判断问题是否满足多米诺性质
- 搜索解空间树，确定剪枝函数
- 确定存储搜索路径的数据结构

> 怎么缩减空间，需要剪枝函数。
> 怎么样确定最优解呢？需要存储搜索路径。

两类典型的解空间树：

子集树：当所给的问题是从n个元素的集合S中找出满足某种性质的**子集**时，相应的解空间树称为子集树。子集树通常有mn个叶结点。**(N皇后问题)**

排列树：当所给的问题是确定n个元素满足某种性质的**排列**时，相应的解空间树称为排列树。排列树通常有n!个叶结点。 **(货郎担问题)**

> 回溯法求解的问题可以根据其解空间树的类型分成两类。一种是子集树，0,1背包问题，每一个位置都有0或1两个可选项，故解空间为2^n.
>
> 另一种为排列树，货郎担相当于n个城市的一个排列。还有调度问题，也是排列树，n个任务的一个排列。
>
> 解空间树：
>
> 子集树：N皇后问题，通常有m^(n)个叶节点
> 排列树：TSP问题，通常有n!个叶节点

解题算法框架：

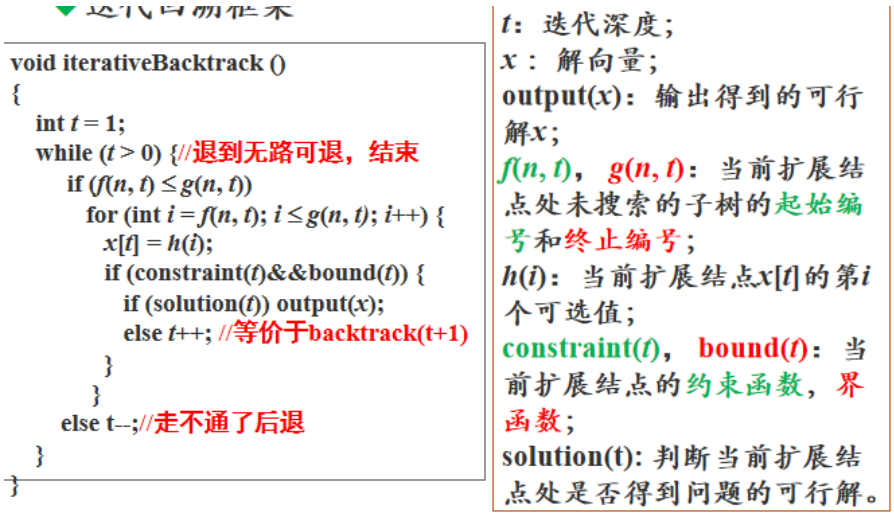
递归回溯框架：

```
void backtrack (int t)
{
    if (t > n) output(x);
    else
        for (int i = f(n, t); i ≤ g(n, t); i++)
        {
            x[t] = h(i);
            if (constraint(t)&&bound(t))
                backtrack(t+1);
        }
}
```

t：递归深度；
n：最大递归深度；
x：解向量
output(*x*)：输出得到的可行解*x*；
f(*n*, *t*)，*g*(*n*, *t*)：当前扩展结点处未搜索的子树的起始编号和终止编号；
h(*i*)：当前扩展结点*x*[*t*]的第*i*个可选值；
constraint(*t*)，bound(*t*)：当前扩展结点的约束函数，界函数；

> X为解向量，我们的目标是确定解向量中的每一个维度的分量值。深度为t时需要确定x[t]的值。典型的深度优先搜索程序，无非加了约束和界；

迭代回溯框架：：



```
void iterativeBacktrack ()
{
  int t = 1;
  while (t > 0) {//退到无路可退，结束
    if (f(n, t) ≤ g(n, t))
      for (int i = f(n, t); i ≤ g(n, t); i++) {
        x[t] = h(i);
        if (constraint(t)&&bound(t)) {
          if (solution(t)) output(x);
          else t++; //等价于backtrack(t+1)
        }
      }
    else t--;//走不通了后退
  }
}
```

t：迭代深度；
x：解向量；
output(x)：输出得到的可行解x；
f(n, t)，g(n, t)：当前扩展结点处未搜索的子树的起始编号和终止编号；
h(i)：当前扩展结点x[t]的第i个可选值；
constraint(t)，bound(t)：当前扩展结点的约束函数，界函数；
solution(t)：判断当前扩展结点处是否得到问题的可行解。

> 先做符号说明，再说明与递归的关联性，深度为t时需要确定x[t]的值，当满足约束的时候做t++，等价于backtrack(t+1)，找不到节点后，做t--
> 用while循环替代了递归过程，在while循环中先做t++，找不到结点后做t--。

在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根节点到扩展结点的路径，如果解空间树中从根节点到叶子结点的最长路径的长度为h(n)，则回溯法的空间复杂度通常为O(h(n))。

显示存储整个解空间则需要O ($m^{h(n)}$)或者O (h (n)!)。

N皇后问题：

问题：在n×n的棋盘中放置n个皇后，使得任何两个皇后之间不能相互攻击，试给出所有的放置方法。

1）每一行有且只有一个皇后，设第i行的皇后所在的列为x_i，则问题解向量：$(x_1, x_2, \dots, x_n)$

2）显约束：$x_i = 1, 2, \dots, n$

3）隐约束：

（1）不同列：$x_i \neq x_j$；

（2）不处于同一正（$x_i - i \neq x_j - j$）、反对角线（$x_i + i \neq x_j + j$）：合并后得到$|i - j| \neq |x_i - x_j|$。

```
void iterativeBacktrack (int n)
{
  int t = 1;
  int[] f;// record its initial value
  while (t > 0) {
        if(f[t] ≤ n)
          for(; f[t] ≤ n; f[t]++) {
            x[t] = f[t];
            if(check(x, t))
              if (t=n)output(x);
              else {
                t++;  f[t] = 0;
              }
          }
        else {
            t--;
            f[t]++;}
  }
}
```

TSP问题：

问题：某售货员要到若干城市去推销商品，已知各城市间的路程耗费（代价），如何选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使得总路程耗费最小。

解空间表示：

1）每个城市只出现有且仅有一次，设第i个出现的城市为x_i，则问题解向量：$(x_1, x_2, \dots, x_n)$

2）显约束：$x_i = 1, 2, \dots, n$

3）隐约束：

（1）有从x_i到x_{i+1}的边；

（2）有从x_n到x_1的边；//能回到出发城市

（3）$x_i \neq x_k$； //城市不能重复

伪代码：

```
a[n][n]；//邻接矩阵，存储任意两个城市间的代价；
cc = 0；存储当前代价；
bestc =NoEdge；//存储当前最小代价
bestx[n]; 存储当前最小代价对应的路线；
void Traveling<Type>::Backtrack(int t){ //t的初值为2；
  if (t == n) {
    if (a[x[n-1]][x[n]] != NoEdge && a[x[n]][1] != NoEdge &&
      (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc || bestc == NoEdge))
      for (int j = 1; j ≤ n; j++) bestx[j] = x[j];
      bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];}
    }
  else {//t < n
    for (int j = t; j ≤ n; j++)
      if (a[x[t-1]][x[j]] != NoEdge && // 是否可进入x[j]子树?
        (cc + a[x[t-1]][x[j]] < bestc || bestc == NoEdge)) {
        // 搜索子树
        Swap(x[t], x[j]);
        cc += a[x[t-1]][x[t]];
        Backtrack(t+1);
        cc -= a[x[t-1]][x[t]];//恢复现场
        Swap(x[t], x[j]);}
    }
}
```

时间复杂度为n!

案例：

装载问题

批作业调度问题

0-1背包问题

最大团问题

图的m着色问题

影响效率的因素：

产生x[k]的时间

满足显示约束的x[k]值的个数

计算约束函数Constraint的时间

计算上界函数Bound()的时间
满足约束函数和上界函数约束的所有x[k]的个数

八、分支限界法

思想：与回溯法类似，在问题解空间中搜索问题解

对比：

- 求解目标不同：
  回溯法可用于求解目标是找出满足约束条件的 所有解；
  分支限界法求解目标通常是满足约束条件的 一个解或最优解
- 搜索方式不同：
  回溯法主要是深度优先
  分支限界主要是广度优先或者函数优先

在分支限界中，每个活结点只有一次机会成为扩展节点，一旦成为扩展节点，
就一次性产生所有儿子节点，舍弃不可行解或非最优解，其余加入 活结点表中，
然后从活节点表取下一节点成为当前扩展节点，直到找到所需解或者表空为止。

代码框架：

```
Q = {q0};//存储所有的活结点,初始化为根节点
void Branch&Bound ()
{
    while (Q!=∅) {
        select a node q from Q;//从Q选择一个结点
        Branch(q, Q1); //对q进行分支，产生Q1，分支时利用约束和界进行剪枝
        add (Q1, Q);// 将新产生的活结点加入Q
        }
}

```

常见的两种分支搜索法（选择节点方式）：

队列式(FIFO)搜索法：按照队列先进先出（FIFO）原则选取下一个节点为扩展节点。

优先队列式搜索法：按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点。

最大堆（最大优先队列）：最大效益优先
最小堆（最小优先队列）：最小耗费优先

节点v上界(UB(v))：从v出发得到的所有叶子节点的效益值均不大于UB(v)，则UB(v)为节点v的上界；

节点v下界(LB(v))：从v出发得到的所有叶子节点的效益值均不小于LB(v)，则LB(v)为节点v的下界；

如果所有叶子节点的最大效益值等于UB(v)，则UB(v)为节点v的上确界；

如果所有叶子节点的最小效益值等于LB(v)，则LB(v)为节点v的下确界。

极小化：（极大化与之相反）

◆对于求最小值的优化问题，如果LB(v) ≥ cBest，则节点v可以加入黑名单，不再对其搜索。

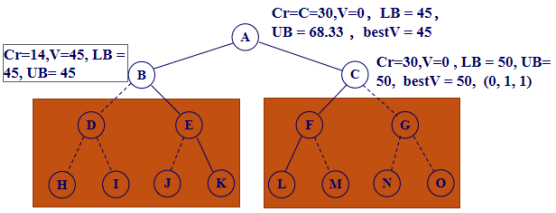
◆UB(v)通常可以利用贪心思路或者其它方式得到一个解，令其作为UB(v)，而LB(v)通常需要经过严格的证明。

◆对于求最小值的优化问题，如果$UB(v_0) = LB(v_0)$，则直接结束，输出对应于$UB(v_0)$的解即可；

◆如果节点v的UB(v) = LB(v)，则该节点不用继续搜索，直接用UB(v) 作为其可到达的最优值，对cBest进行更新；

◆如果找到节点v使得$f(v) = LB(v_0)$或者所有的节点搜索完毕则搜索完毕。

◆cBest 初始等于$UB(v_0)$，找到UB(v)小于cBest，则可以更新cBest ，另外可以减掉LB(v) ≥ cBest的点v。

> 示例：
>
> 0-1背包问题：（极大值问题）
> n=3, C=30, w={16, 15, 15}, v={28, 30, 30}
>
> 物品按照价值比重量的大小进行排序：{w2, w3, w1}.
>
> 下界：按照价值比重量的大小依次装包，在满足容量约束的前提下得到的最大值作为下界LB，即LB(v0) = 30 + 30 = 60，对应的解为（0， 1， 1）
>
> 上界：按照价值比重量的大小依次装包，没装满的按照未装的物品的最大比重进行换算，即UB(v0) = 30 + 30 = 60。
>
> LB(v0) = UB(v0)，故最优解为（0， 1， 1）。
>
> 0-1背包问题：
> n=3, C=30, w={16, 15, 15}, v={45, 25, 25}
>
> 
>
>
> 0-1背包问题：
> n=3, C=30, w={16, 15, 15}, v={45, 25, 25}
>
> 物品按照价值比重量的大小进行排序：{w1, w2, w3}.
>
> 如果n=4, C=30, w={40,16, 15, 15}, v={160,45, 25, 25}，不去掉物品1，上界为UB1(v0) = 30*160/40 = 120 》68.33。
>
> 上界：先把物品重量大于余量的除掉，剩下物品按照价值比重量的大小依次装包，最后没装满的按照未装的物品的最大比重进行换算，即UB1(v0) = 45 + 14*25/15 = 68.33。

分支限界法设计要素：

- 针对问题定义解空间：
  问题解向量
  解向量分量取值集合
  构造解空间树
- 判断问题是否满足多米诺性质
- 确定剪枝函数和界函数

适应条件：与回溯法相同，问题需要具备多米诺性质。

适用问题：求解最优解或一个可行解。

程序结构：节点扩展适合采用迭代法进行。

注意：

分支定界法剪枝可能会减掉同为最优解的解，因此不用它计算求所有问题的最优解的问题。

案例：

装载问题
0-1背包问题
TSP问题
布线问题
最大团问题
批作业调度问题

九、遗传算法

思想：借鉴 自然选择 和 自然遗传机制 的随机化搜索机制。

在每次迭代中，按 某种指标 从解群中选取较优的个体。利用

遗传算子（选择、交叉、变异）对这些个体进行组合，产生新一代解群，重复此过程，直到满足收敛指标为止。

SGA：基本遗传算法，是其他遗传算法的基础，过程简单。

基本遗传算法组成：

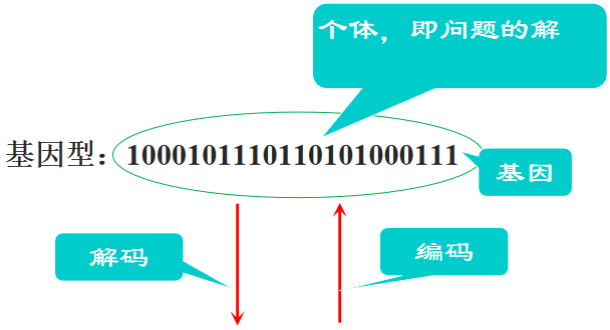
- **编码（产生初始解群）**

  GA的编码机制：把对象抽象成 由特定符号按一定顺序排成的串。

  SGA：使用 二进制 或者 数字串 进行编码。（二进制编码参考组原的地址编码）

  种群初始化：SGA用随机方法产生若干个染色体的集合，称为**初始种群**。

  初始种群中个体的数量称为**种群规模**。

  > 求下列一元函数的最大值：
  >
  > $f(x)=x\cdot\sin(10\pi\cdot x)+2.0$
  >
  > $x\in[0,3]$，求解结果精确到6位小数。
  >
  > 由于[0, 3]区间长度为3，求解结果精确到6位小数，因此可将自变量定义区间划分为3×10^6等份。又因为$2^{21}<3\times10^6<2^{22}$，所以本例的二进制编码长度至少需要22位，本例的编码过程实质上是将区间[0，3]内对应的实数值转化为一个二进制串（$b_{21}b_{20}...b_0$）。
  >
  > 
  >
  >
  > 表现型：2.288967

- **适应度函数**

  设计标准：适应度函数值越大，解的质量越好。

  适应度函数 是遗传算法进化过程的 驱动力 ，也是进行自然选择的 唯一标准，它的设计应结合求解问题本身的要求而定。

  > 求下列一元函数的最大值：
  >
  > $f(x)=x\cdot\sin(10\pi\cdot x)+2.0$
  >
  > 适应度可以表示为染色体（$b_{21}b_{20}...b_0$）解码后对应的值x的函数值$f(x)$。
  >
  > 求下列一元函数的最小值：
  >
  > $f(x)=x\cdot\sin(10\pi\cdot x)+2.0$
  >
  > 适应度可以表示为染色体（$b_{21}b_{20}...b_0$）解码后对应的值x的函数值$1/f(x)$ 或$MAX-f(x)$。

- **遗传算子（选择、交叉、变异）**

  遗传算法使用 选择运算 来 实现 对群体中的个体进行 优胜劣汰操作。

  优胜劣汰操作：适应度高的个体被遗传到下一代的概率高，反之则低。个体适应度越大，占总的适应度的占比越大，个体被选中的概率越高。

  选择操作的 任务 就是按某种方法从父代群体中选取一些个体，遗传到下一代群体。

  SGA的遗传算子采用 **轮盘赌** 选择、单点交叉算子。

  轮盘赌选择（比例选择算子）：各个个体被选中的概率与其适应度值大小成正比。

设群体大小为n，个体i 的适应度为 Fi，则个体i 被选中遗传到下一代群体的概率为：

$$P_i = F_i / \sum_{i=1}^{n} F_i$$

1. 计算群体中所有个体的适应度函数值；
2. 利用比例选择算子公式，计算每个个体被选中遗传到下一代群体的概率；
3. 采用模拟轮盘赌操作（即生成0到1之间的随机数与每个个体遗传到下一代群体的概率进行匹配），确定是否遗传。

其他算子：交叉算子、基本位变异算子。

交叉运算：两个相互配对的染色体依据交叉概率Pc，按某种方式相互交换部分基因形成两个新的个体。是产生新个体的主要方法。

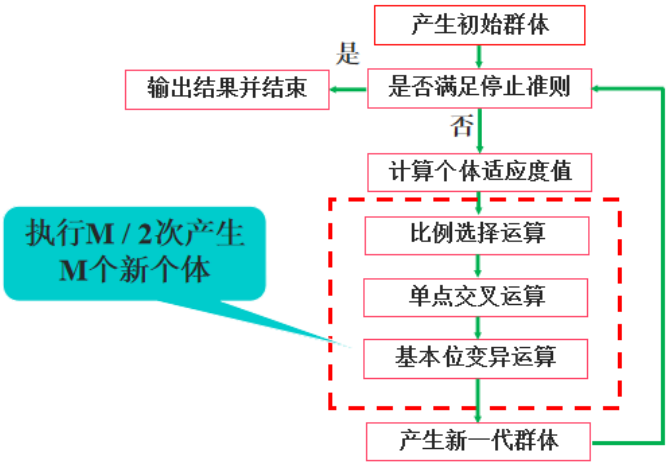
SGA中交叉算子采用**单点交叉算子。**

基本位变异算子：对个体编码串 随机指定某一位或某几位基因进行变异运算。若需要进行变异操作的某一基因的原有基因值为0，则变异操作将其变为1；反之亦然。

- **运行参数**
  M：种群规模
  Pc：交叉概率
  Pm：变异概率
  T：终止进化代数（停止规则）

> 变异的目的是产生新的特性，变异和交叉不一定能保证子代比父代一定会好。变异和交叉的过程都相当于是随机搜索的过程，只是后一代后一代能否再继续遗传到后一代取决于适应度值。

遗传算法执行流程：



特点：

1. 群体搜索，易于并行化处理。
2. 自组织、自适应和自学习特征。
3. 不需要求导或其他辅助知识，只需要知道适应度函数。
4. 强调概率转换规则，而非确定的转换规则。

收敛性分析：

首先要求任意初始种群经有限步都能达到全局最优，
其次算法必须要有保优操作，防止最优解丢失。

与收敛性相关的因素：

- 种群规模：规模太小，采样点不足，性能差；规模太大，收敛时间太长。

- 选择操作：采用最优保存策略，父代最佳个体直接遗传进入下一代，使算法以概率1收敛于全局最优解。使高适应度个体能够以更大的概率生存。
- 交叉概率：交叉操作用于个体对，产生新的个体，实质上是在解空间中进行有效搜索。交叉概率太大，种群中个体更新很快，高适应度的个体很快被破坏；概率太小，交叉操作很少进行，造成算法不收敛。
- 变异概率：是对种群模式的扰动，有利于增加种群的多样性 。变异概率太小，很难产生新模式；概率太大，使算法变成随机搜索算法。

Schaffer建议的最优参数范围是：
M = 20-100，
T = 100-500，
Pc = 0.4-0.9，
Pm = 0.001-0.01。

自适应遗传算法：

Pc和Pm可以随适应度自动改变，
当种群的各个个体适应度趋于一致或趋于局部最优时，使二者增加，
当种群适应度较分散时，两者减少，
对适应度高于群体平均适应度的个体：两者较低，使性能优良的个体进入下一代；
对适应度低于群体平均适应度的个体：两者较高使性能较差的个体被淘汰。

应用：

函数优化、组合优化、生产调度问题、自动控制、机器人学、图像处理、人工生命

> 举例：TSP问题
>
> 一个商人欲从自己所在的城市出发，到若干个城市推销商品，然后回到其所在的城市。如何选择一条周游路线使得商人经过每个城市一次且仅一次后回到起点且使他所走过的路径最短?
>
> **编码：**
>
> 假设一共有n + 1个城市，则可以对每个城市编码为1,2，..., n + 1。问题的解即为一个2到n + 1的排列。若n = 9, 则一个染色体可以表示为：
> △ = (2, 8, 5, 3, 4, 6, 10, 7, 9)
> 其含义为：1-2-8-5-3-4-6-10-7-9-1
>
> **适应度函数：**
>
> 令f(△)表示从城市1依次经过△中的城市需要走的总距离。
> 则适应度函数可以表示为：
> F(△) = 1/ f(△)
>
> **交叉操作：**
>
> 
>
>
> **修复操作：**

> **待修复个体：**
>
> $$\Pi = \{2, 8, 5, 3, 8, 5, 3, 10, 9\}$$
>
> **扫描Π ，统计基因出现的次数，将非第一次出现的基因替换为0：**
>
> $$\Pi_1 = \{2, 8, 5, 3, 0, 0, 0, 10, 9\}, \quad Count = \{1, 2, 0, 2, 0, 0, 2, 1, 1\}$$
>
> **将Count为0的基因取出来打乱顺序后，依次替换Π_1中为0的基因，假设{4, 6, 7} 打乱后为 {7, 4, 6}，则修复后的染色体为：**
>
> $$\Pi_2 = \{2, 8, 5, 3, 7, 6, 4, 10, 9\}$$
>
> **变异操作：**
>
> **变异前：**
>
> **2, 8, 5, 3, |4, 6, 7, 9, 10**
>
> 变异点
>
> **变异后：**
>
> **2, 8, 5, 3, |7, 6, 7, 9, 10**
>
> 再按照遗传算法执行流程，反复计算得到结果。

遗传算法用于神经网络的权值优化：

(1)采用某种编码方案对每个权值进行编码，随机产生一组权值编码；
(2) 计算神经网络的误差函数，确定其适应度的函数值，误差值越大，适应度值越小；
(3)选择若干适应度值大的个体直接遗传给下一代，其余按适应值确定的概率遗传；
(4)利用交叉、变异等操作处理当前种群，产生下一代种群；
(5)重复(2)～(3)，直到取得满意解。